Coupling Storage Systems for Efficient Management of Self-Describing Data Formats

DISSERTATION

zur Erlangung des akademischen Grades

Doktoringenieurin (Dr.-Ing.)

angenommen durch die Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg

von Kira Isabel Duwe

geb. am 25.01.1991 in Hamburg

Gutachterinnen/Gutachter

Jun.-Prof. Dr. Michael Kuhn
Prof. Dr. Thomas Ludwig
Prof. Dr. Wolfram Wingerath

Magdeburg, den 21.02.2023

Abstract

In times of continuously growing data sizes, performing insightful analysis is increasingly difficult. Therefore, it is vital to sift through the data volumes most efficiently. However, the hardware hierarchy of high-performance computing (HPC) systems complicates the data analysis. Another factor increasing the complexity is the software stack on top. The stack splits into several mostly isolated layers. While this allows exchanging them, the isolation leads to performance and management issues, e.g., how and where to optimize the data access.

To make filtering raw data easier, I/O libraries such as HDF5 (Hierarchical Data Format), and ADIOS2 (Adaptable IO System), and the according self-describing data formats are used. The goal is to provide self-explanatory data that can be exchanged between researchers easily through annotating the data, e.g. with information about the experiment. However, queries on this metadata are difficult as it is currently stored inside the corresponding data formats on data servers. Therefore, the starting point of this thesis is to dissect the HDF5 and ADIOS2's BP file formats into file metadata, such as variables and attributes, and the data. Splitting the formats enables novel and highly efficient data management techniques without redundancy. By also replacing the file system underneath with the storage framework JULEA, dedicated backends like key-value, relational databases and object stores can be used to manage the different data categories.

This approach is kept transparent to the application layer by implementing the separation inside the I/O library. Besides designing and implementing two custom storage engines, a novel data analysis interface (DAI) for JULEA has been developed. The DAI combines the possibility to pre-compute post-processing operations, e.g. computing the mean, with the option to tag specific features to reduce data access during analysis.

The evaluation showed that the format dissection alone is not always beneficial when using a slow backend. However, when bringing the format dissection, the file metadata management in a relational database and the DAI together, the performance can compete with the native ADIOS2 engines when writing data even if the backend itself is not changed.

In combination they can also outperforms the native engines by up to a factor of 60,000 when the custom metadata is queried. In conclusion, the proposed approach provides sufficient performance for the I/O typically performed in a large simulation while offering significant improvements for the post-processing. All this can be achieved without changing the application code because of the default pre-computation of statistics like the mean, thereby preserving the efforts that went into the development.

Zusammenfassung

In Zeiten ständig wachsender Datenmengen wird es immer schwieriger, aufschlussreiche Analysen durchzuführen. Daher ist wichtig, die Datenmengen möglichst effizient zu durchforsten. Die Hardware-Hierarchie von High-Performance-Computing-Systemen (HPC) erschwert jedoch die Datenanalyse. Ein weiterer Faktor, der die Komplexität erhöht, ist der darüber liegende Software-Stack. Der Stack teilt sich in mehrere meist isolierte Schichten auf. Dies ermöglicht zwar den Austausch von Daten, aber die Isolierung führt zu Leistungs- und Verwaltungsproblemen, z. B. bei der Frage, wie und wo der Datenzugriff optimiert werden kann. Um die Filterung von Rohdaten zu erleichtern, werden E/A-Bibliotheken wie HDF5 (Hierarchical Data Format) und ADIOS2 (Adaptable IO System) sowie die entsprechenden selbstbeschreibenden Datenformate verwendet. Ziel ist es, selbsterklärende Daten zur Verfügung zu stellen, die durch Annotation der Daten, z.B. mit Informationen über das Experiment, leicht zwischen Forschern ausgetauscht werden können. Die Abfrage dieser Metadaten ist jedoch schwierig, da sie derzeit innerhalb der entsprechenden Datenformate auf Datenservern gespeichert sind. Daher ist der Ausgangspunkt dieser Arbeit die Zerlegung der HDF5- und der ADIOS2 Dateiformate (BP3, BP4, BP5) in Dateimetadaten, wie Variablen und Attribute, und die Daten. Die Aufspaltung der Formate ermöglicht neue und effiziente Datenverwaltungstechniken. Indem das darunter liegende Dateisystem durch das Speicherframework JULEA ersetzt wird, können dedizierte Backends wie Key-Value-Stores, relationale Datenbanken und Objektspeicher für die Verwaltung der verschiedenen Datenkategorien verwendet werden. Dieser Ansatz wird für die Anwendungsschicht transparent gehalten, indem die Trennung innerhalb der I/O-Bibliothek implementiert wird. Neben dem Entwurf und der Implementierung von zwei Speicher-Engines wurde eine neuartige Datenanalyse-Schnittstelle (DAI) für JULEA entwickelt. Das DAI kombiniert die Möglichkeit, Post-Processing-Operationen, z.B. die Berechnung des Mittelwerts, vorzuberechnen, mit der Option, bestimmte Merkmale zu markieren, um den Datenzugriff während der Analyse zu reduzieren. Die Evaluation hat gezeigt, dass die Formatzerlegung allein nicht immer von Vorteil ist, wenn ein langsames Backend verwendet wird. Wenn jedoch die Formatzerlegung, die Verwaltung von Dateimetadaten in einer relationalen Datenbank und das DAI zusammengebracht werden, kann die Leistung beim Schreiben von Daten mit den nativen ADIOS2-Engines konkurrieren, selbst wenn das Backend selbst nicht geändert wird. In Kombination können sie auch die nativen Engines um einen Faktor von bis zu 60.000 übertreffen, wenn die benutzerdefinierten Metadaten abgefragt werden. Zusammenfassend lässt sich sagen, dass der vorgeschlagene Ansatz eine ausreichende Leistung für die E/A bietet, die typischerweise in einer großen Simulation durchgeführt wird, und gleichzeitig signifikante Verbesserungen für das Post-Processing bietet. All dies kann erreicht werden, ohne dass der Anwendungscode geändert werden muss, da Statistiken wie der Mittelwert standardmäßig vorberechnet werden, wodurch die Ressourcen, die in die Entwicklung geflossen sind, erhalten bleiben.

"So remember to look at the stars and not down at your feet. Try to make sense of what you see and wonder about what makes the universe exist.

BE CURIOUS!

And however difficult life may seem, there is always something you can do and succeed at. It matters that you don't just give up. [...]

We are all Time Travelers, journeying together into the future but let us work together to make that future a place we want to visit.

BE BRAVE, BE DETERMINED, OVERCOME THE ODDS!

It can be done!"

Last words of Stephen Hawking

Contents

1. Introduction

This chapter introduces the problems faced in today's HPC systems and why they are essential. The complexities and issues of the software and hardware stack and the unused potential of the current management of self-describing data formats are discussed. Afterwards, the challenges that are unsolved by related work are highlighted, and how they lead to developing the research questions. Then, the proposed solutions are explained in detail. Finally, the structure of the thesis is outlined. The chapter is split into the following sections:

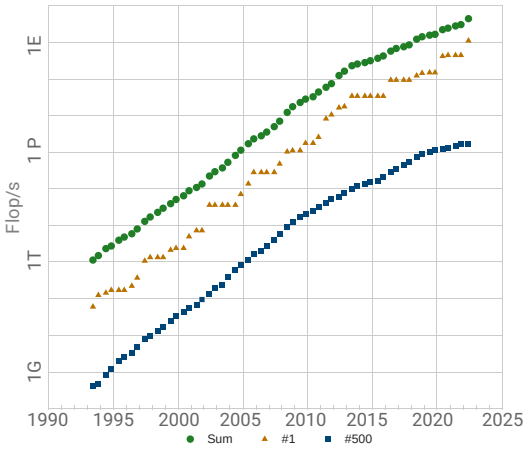
- *Section 1.1: highlights the motivation for the topic and the research area*
- *Section 1.2: provides the state of the art*
- *Section 1.3: states the derived research questions*
- *Section 1.4: presents the thesis structure.*

1.1. Motivation

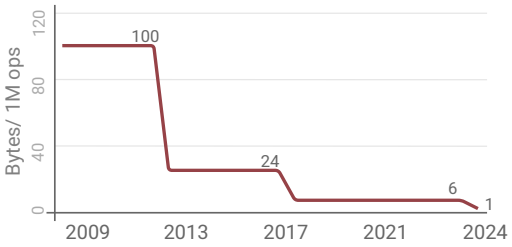
Our society today has come to increasingly rely on technology. This concerns scientific research as well. Substantial breakthroughs like the finding of the Higgs-Boson or the experimental proof of gravitational waves could not have been achieved without computers. Many open research questions require complex simulations and complicated experiment setups to gain new insights. Neither the experiment setup nor the data analysis would be possible without digital help. The presumably easy questions are already answered. Today's endeavours range from finding vaccines in a global pandemic to predicting the local impact of climate change. The high-resolution results required to investigate climate change are only possible because of the growing computing power of the CPUs and the option to combine them into increasingly large high-performance computing (HPC) clusters on the one hand and the advancements in analysing enormous data volumes on the other hand.

In Figure 1.1a the performance development of the 500 fastest supercomputers is shown. Since June 1993, the best clusters have been ranked twice yearly. While the maximum performance achieved in the High-Performance Linpack (HPL) benchmark is not the only listed feature, it is the decisive metric in determining the rank. It is measured in floating point operations per second, short flop/s. The performance developed exponentially from 59.7 Gigaflop/s in June 1993 to 1.102 Exaflop/s in June 2022. That is an increase by a factor of roughly 18,500,000. There are small plateaus in Figure 1.1a because the number one cluster often remains at the top of the list for over half a year.

In the Top500 it is observable that the growth of computational power is slowing. CPUs cannot be increased in speed arbitrarily because the energy consumption to cool them rises massively. Nonetheless, CPUs are getting more powerful by increasing the number of cores and thereby parallelism. Both the network hardware and the storage systems cannot keep up. While this is not a new problem and was already addressed in 1989 [Ousterhout and Douglis, 1989], it is still not solved [Settlemyer et al., 2021].



(a) Performance development [Top500, 2022]

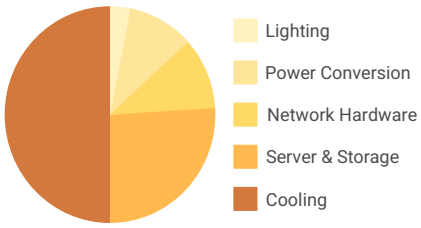


(b) Data Compute Gap:
Redrawn [Klasky et al., 2017] © 2017 IEEE

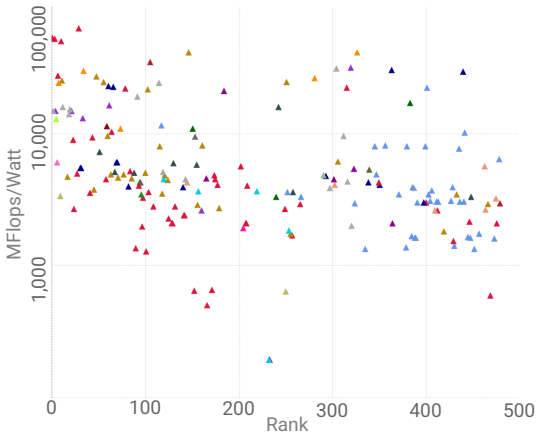
Figure 1.1.: Development of computational power: (a) The performance development in the Top500 list and (b) the different development of computation and storage hardware comparing the filesystem throughput to the total number of performed floating point operations per second.

However, different hardware categories did not develop at the same rate. While the performance of storage and network hardware did not follow this curve, it still developed exponentially. The resulting difference between computing and storage capabilities is shown in Figure 1.1b. As storage hardware performance is not growing as quickly, it becomes a severe bottleneck. With its reduced latency and increased performance, storage class memory (NVRAM and NVMe SSDs) can partly bridge this gap [van Renen et al., 2019]. However, it is not the final solution for all problems because of the small storage sizes and the high acquisition costs. This is especially true for computing sites with large-scale storage systems like the data archive at the German Climate Computing Centre (DKRZ). Currently, more than 130 Petabytes of climate data are archived. Every year 30 PB of additional data need to be archived. The system is built to deal with up to even 75 PB a year. This data must be stored for decades to validate the models because the climate is a long-term process. So, efficient data retrieval becomes even more relevant to dealing with systems of these dimensions.

Another aspect when running data centres is the ongoing costs, for example, energy. As energy consumption cannot increase indefinitely, energy efficiency becomes increasingly relevant in the Top500. It evaluates how many operations can be performed for a watt. The energy efficiency of all ranked supercomputers is shown in Figure 1.2b. As is evident, there is an enormous range. The most efficient supercomputer is rank 29, an HPE Cray (AMD EPYC)



(a) Energy consumption in data centres
Redrawn [Dayarathna et al., 2016] (CC BY 4.0)



(b) Energy efficiency in the Top500 [Top500, 2022]

Figure 1.2.: Energy consumption (a) and efficiency (b). The ranked clusters are coloured according to their vendors.

reaching 62,200 Mflops/watt, whereas the rank 232 Tundra Extreme Scale (Xeon) comes in last with only 193 Mflops/watt. This is a factor of 322. With the rise of energy prices, this aspect is of utmost importance.

In Figure 1.2a the energy consumption of different data centre components is presented. Dayarathna et al. found that about half the energy goes into cooling, yet a quarter goes into server and storage. So, reducing the data volumes is also sensible in terms of energy prices and power consumption. Here, data compression becomes increasingly important as a building block in the endeavours to decrease the data sizes [Tang et al., 2021]. Another proposal for decreasing power consumption is to use ARM clusters for storage systems. Evaluations showed such setups could outperform traditional x86 Ceph cluster [Erxleben et al., 2022].

1.2. State of the Art

Data sifting techniques are essential to all research areas with large data volumes. Managing data and metadata efficiently, however, is complicated by both the hardware and software architectures of today's systems. In the following, the challenges related to data and metadata management are presented in more detail.

Local and Parallel File Systems

Data organisation is traditionally handled by file systems. Most file systems are realised as kernel file systems, i.e. integrated into the operating system. File systems offer an interface to the underlying storage devices providing comfortable and simplified access through identifiers such as file names. File names are mapped to the physical address of the storage devices to avoid the need to handle raw storage blocks. To support various file systems, a uniform kernel API was introduced, namely the *Virtual File System* (VFS) [Bovet and Cesati, 2005]. Its

interface semantics are specified by the *POSIX* (Portable Operating System Interface) standard. Prominent examples are the ext* family for Linux systems or FAT for Windows. Local file systems manage a specific server, while large-scale systems employ parallel file systems that distribute the data over multiple devices. This technique is called striping.

The distribution can be performed at different granularity levels: complete files, single datasets, or parts of a dataset. Striping balances the workload across multiple servers, reducing the risk of uneven wear of the storage hardware. Furthermore, it increases parallel access because different servers process the Input/Output (I/O) requests. A fundamental challenge with data distribution is to keep track of which part of the data is where. Also, storing the distribution with minimal storage requirements is challenging.

There are different concepts, that can be used alone or in combination. One is using hashing to determine on which server a specific stripe is stored. A second is the separation of servers into two types; data servers and metadata servers. The data location is held in a data structure in the metadata server. A critical problem is to keep a consistent view of the data over all the servers. For example, small updates have considerable overhead when sent over the network. Therefore, atomic updates enforced by POSIX semantics are not feasible in distributed systems.

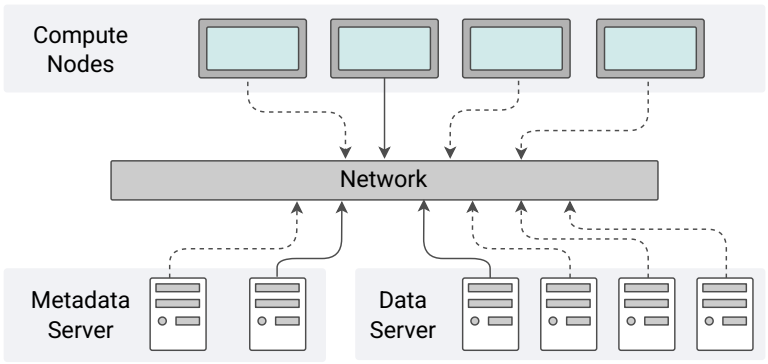
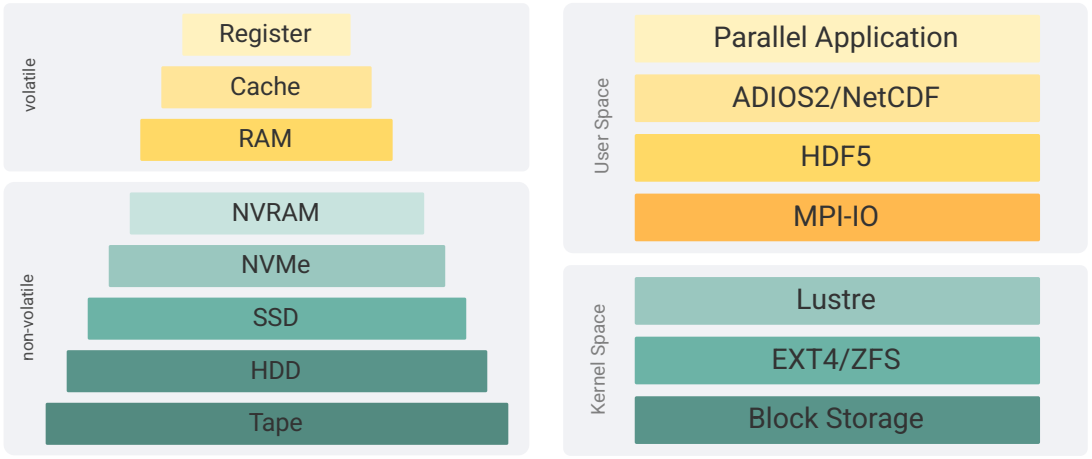


Figure 1.3.: The general system architecture of parallel distributed file systems consists of compute nodes running the application, metadata and data servers and a network for communication in between. Based on [Kuhn, 2015]

The differentiation of metadata and data is employed in many systems. Even though this concept is not new, it seems worthwhile to look further into the separation. File system metadata is information such as ownership, access rights, and time stamps. Data is the file content itself. In the case of scientific simulations, the data is typically matrices of numerical values. Research in data management has also shown that the distinction between structured and unstructured data is essential. Structured data such as file system metadata is suitable for NoSQL databases like key-value stores, while an object store better manages unstructured data like large data blobs. Access patterns differ significantly between data and metadata. The former is typically large and continuous, while the latter is accessed more randomly.

Hardware Hierarchy

A hierarchy of different storage devices is used to cover the wide range of requirements imposed on HPC systems. Thereby the demand for high capacity, on the one hand, and high velocity, on the other hand, are satisfied at the same time. By combining the respective features, the overall performance is significantly increased while reducing costs for acquisition and maintenance [Dong et al., 2016]. The result is a storage hierarchy with tape systems forming the bottom layer and cache technologies at the top.



(a) Hardware Hierarchy: Volatile layers are coloured in yellow, non-volatile layers in green

(b) I/O Software Stack: User space layers are coloured in yellow, kernel space layers in green

Figure 1.4.: The current hardware hierarchy of HPC systems (a) and the software I/O stack used on top (b).

The upper layers are volatile, meaning they do not store information persistently, so the data is lost without a constant power supply. From top to bottom, the capacity increases from 8-bit to several TB per HDD or tape while the acquisition costs decrease. Unfortunately, the latency grows dramatically. Descending through the layers, the duration that data can be stored also rises, even up to decades for tape archives. Also, the energy required to move data from lower layers to the RAM is about 100 to 1000 times higher than moving it from main memory to cache [Jacob et al., 2008, Prabhat and Quincey Koziol, 2014]. While the hardware itself is available, efficient management techniques are lacking. The selection of the appropriate storage technology for a specific piece of data must be made carefully to gain the most out of the layering. Otherwise, expensive operations have to be used to move the data across storage tiers repeatedly. Data placement decisions are often based on heuristics derived from access frequencies and related I/O patterns. However, the strict separation of layers and the loss of structural information further hinder efficient policies. These challenges are described in more detail in the following.

HPC Input/Output Stack

Besides the hardware hierarchy, the I/O software stack also interferes with efficient data sifting and the optimal utilisation of the cluster. Figure 1.4b illustrates a typical HPC I/O stack. In climate research, the applications typically directly employ I/O libraries such as NetCDF (Network Common Data Form), which depends on HDF5 (Hierarchical Data Format). They offer rich metadata and optional hints for the I/O libraries to allow performance tuning. Nevertheless, information about the data structure is lost while descending through the layers: Simulations typically pass numerical data in the form of multi-dimensional matrices to NetCDF. Currently, MPI-IO is only aware of a stream of elements, e.g. integers, and lacks critical metadata aside from the file name. At last, the POSIX interface used by Lustre and the local file systems like EXT4 only works on a byte stream [Gray et al., 2005, Lofstead, 2010].

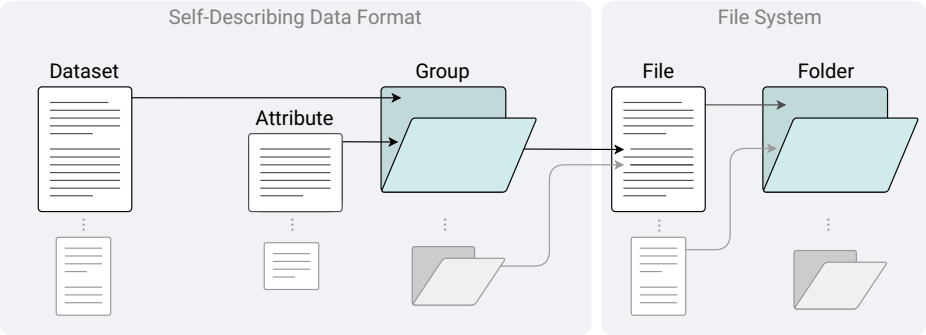
Due to the complex interplay of different components and optimisations in the I/O stack, performance issues are common. One reason is that the strict POSIX semantics, typically provided by the underlying parallel file system, is forced upon the upper layers [Stender et al., 2008]. POSIX requires changes to be visible immediately after a write call and performed in an atomic way; this can lead to performance issues in a distributed system due to synchronisation overhead. HPC applications often do not require the strict consistency and coherence semantics offered by POSIX but are not able to opt out [Wang et al., 2021]. Even though higher layers such as MPI-IO typically offer relaxed semantics, their dependence on the parallel file system's POSIX interface effectively negates these relaxations. Static semantics cannot satisfy the requirements of vastly different applications, requiring application-specific workarounds.

Self-Describing Data Formats

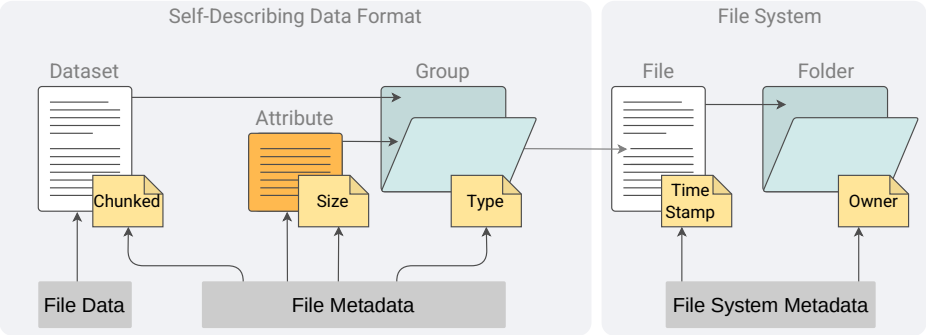
The I/O libraries make data handling easier for application developers and users. The associated file formats like HDF5, NetCDF and BP ease the data exchange between researchers. They are so-called self-describing data formats (SDDFs) that allow the option to annotate data with additional attributes, e.g. about the experiment runs. In comparison to files where only byte streams are stored, self-describing data can be accessed and managed by other users without requiring prior knowledge about the data or its structure.

In Figure 1.5a the general concept of the self-describing format is shown. While the models between common libraries such as HDF5 and ADIOS2 can be mapped onto one another, they have different terms for similar aspects. In HDF5's terminology and model, the data is stored in datasets that belong to a group. These groups, in turn, are part of a self-describing file stored in a folder. ADIOS2, on the other hand, uses data and variables that can contain additional information, for example, about the value range or the number of simulation time steps. This structuring of the namespace and the data organisation is often described as having a file system structure inside the file system because of the hierarchical nature of those formats. Gray et al. even call them "simple database systems" because they offer features like a schema language and basic indexing strategies [Gray et al., 2005].

While self-describing data formats have great capabilities, their potential is not fully used. The drawbacks and challenges of the current state are discussed next. In the past, everything inside the file was considered data compared to the file system metadata. One of the central



(a) General structure of self-describing data formats. The folders and groups (both in blue) are used for the hierarchical structuring of datasets and attributes (white).



(b) Different types of metadata are indicated: File system metadata and file metadata. The former is further split into attributes set by the user directly (orange) and structural information encoded in the format (yellow).

Figure 1.5.: Self-describing data formats offer the option to encapsulate and structure the data in concepts like datasets and groups. In this thesis, a proposal is made to differentiate between different types of metadata.

aspects of this thesis is the hypothesis that the additional metadata of self-describing data formats is better handled as metadata. Therefore, it is distinguished as a separate class of metadata called *file metadata* in the following chapters. File metadata are both the annotations, for example, *user-set attributes* and the *structural information* of the data, for example, the dimensions of a variable or any hierarchical ordering among groups as shown in Figure 1.5b.

As previously mentioned, the distinction between metadata and data is essential because they are accessed differently. Most importantly, data servers are usually optimised for streaming I/O and, therefore, can not deal well with small random accesses typical for metadata requests. Data servers often rely on sequential access patterns to achieve high performance through techniques such as read-ahead.

The rudimentary management of self-describing data files in current systems is illustrated in Figure 1.6. Assume an application requests to read part of a dataset from an HDF5 file. The call to the HDF5 library, in turn, calls the respective system call, which is then handled by the

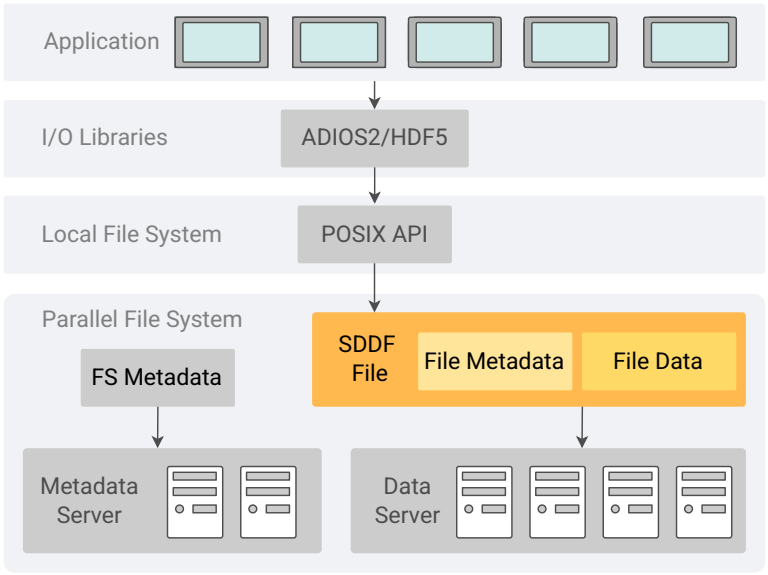


Figure 1.6.: Current management of self-describing files (orange) in a parallel file system. The file system metadata (grey) is handled separately in a dedicated metadata server. The file metadata (light yellow), on the other hand, is stored along the data (dark yellow) inside the self-describing data file on the data server.

parallel file system. The file system metadata is accessed to identify the servers holding the data. Also, further measures are taken based on the file system metadata, such as ensuring the user has appropriate access rights. Depending on the file size, the file can be striped across several servers. In the worst case, all these servers must be contacted to determine the file parts that need to be accessed to locate the HDF5 dataset. Finally, when the dataset is read, the respective data block can be returned to the application. As this procedure has blatant drawbacks, there are a lot of improvements, such as keeping a separate index for HDF5 [Zhang et al., 2019] or ADIOS2 files [Gu et al., 2018, Podhorszki et al., 2018, Wu et al., 2017].

To demonstrate the missed opportunities with the management of self-describing data formats at the moment, another example is explained, this time with a specific query. The user wants to process only the data of variables that are in a certain data range. The BP formats of ADIOS2 already store the minima and maxima of every step and block for all variables. They are part of the file metadata and could be used to discard all variables where the extrema lie outside the range. However, since this file metadata is stored along with the data, all files still have to be read. In the worst case, the tape archive needs to be accessed. By storing the file metadata separately, only accesses to the variables meeting the query expression would be required instead of scanning the complete data. Here is significant untapped potential.

In summary, there are a variety of problems. The core problem is the amount of data created by modern scientific experiments. Finding new insights in PB of data is challenging

and requires efficient data management techniques. When instruments like the SKA produce so much data, then all other problems worsen.

A significant obstacle is that hardware develops at different rates. As a result, the increase in data generation is significantly higher than the improvements in network and storage hardware. Various approaches such as data compression and advanced I/O analysis and tuning can at least partially mitigate this bottleneck posed by the hardware [Tang et al., 2021, Duwe et al., 2020, Uselton et al., 2010, Isakov et al., 2020, Bez et al., 2021].

Nonetheless, I/O remains a crucial point for optimisation and storage systems require major redesign efforts [Settlemyer et al., 2021]. A complex hardware hierarchy further hinders the efficient management of these enormous data volumes with a complicated I/O software stack on top. Their interaction leads to the loss of vital metadata information. At last, the self-describing data formats offered by the I/O libraries already provide functionality that could benefit the management. This can be summarised with the following quote: *"Metadata enables data access"* [Gray et al., 2005]. However, as the file metadata is stored inside the files on data servers, it is not accessible enough. Therefore, offering a different query option on data already present seems like a promising approach.

1.3. Research Questions

As previously detailed, the current storage stack used in high-performance computing suffers from several problems regarding data management, specifically the combined storage of file metadata with file data, static I/O semantics and the loss of structural information. Since the application data is typically stored using self-describing data formats, the central question of this thesis is: *How can self-describing data formats be managed efficiently?*

This question can be broken down into several research questions. Three are presented and discussed throughout the thesis:

- **RQ 1**: How to solve problems with state of the art?
- **RQ 2**: What optimisations become possible when the format is split?
- **RQ 3**: How to design an analysis interface for file metadata?

The first question is more general and aims to gather relevant present approaches to improving the current state. Here, the primary hypothesis of the thesis is developed, namely that the dissection of the self-describing data formats and a coupling of the formats with the storage systems is beneficial. The second question directly stems from the first and explores the new possibilities when the format is separated. As the format specifics no longer restrict management options, various possible directions arise. Especially the derivation and precomputation of additional metadata are studied. The third question deals with the data access to the separated data and whether the current I/O interfaces are sufficient. The respective details and all resulting subquestions are discussed in the following.

RQ 1: How to solve problems with the state of the art?

Two questions are derived from RQ1:

- 1.1: What optimisations exist for managing self-describing data formats and storage systems?
- 1.2: Where is untapped potential, and how can it be used?

RQ 1.1 - What optimisations exist for managing self-describing data formats and storage systems?

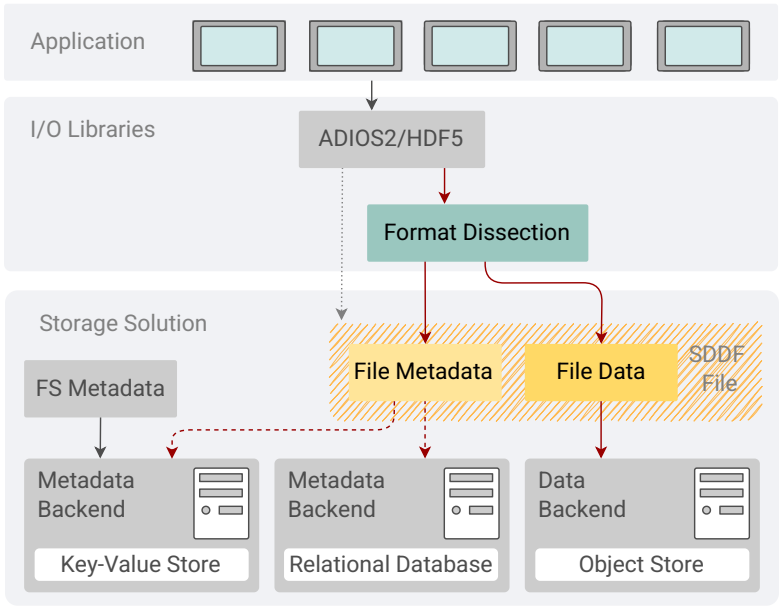
The starting point is getting an overview of related work and which of the discussed problems are addressed or solved by it and which are not. There are a lot of similar endeavours going on, as this topic is crucial for dealing with the dramatic challenges of exascale computing. The corresponding discussion can be found in Chapter 3. While there are several promising systems like MARS (Meteorological Archival and Retrieval System) from the European Centre for Medium-Range Weather Forecasts (ECMWF) [Grawinkel et al., 2015] and the ADDS (Atmospheric Data Discovery System) [Pallickara et al., 2012], they are domain or system specific and therefore not suitable for a general-purpose approach. In summary, most current proposals do not go far enough or overshoot the mark by aiming to revolutionise the complete approach to data management. Neither a new index nor the introduction of a new format is an appropriate solution. So, this thesis aims to find a middle ground between innovation and the possibility of deployment and, therefore, to inspire actual change.

RQ 1.2 - Where is untapped potential, and how can it be used?

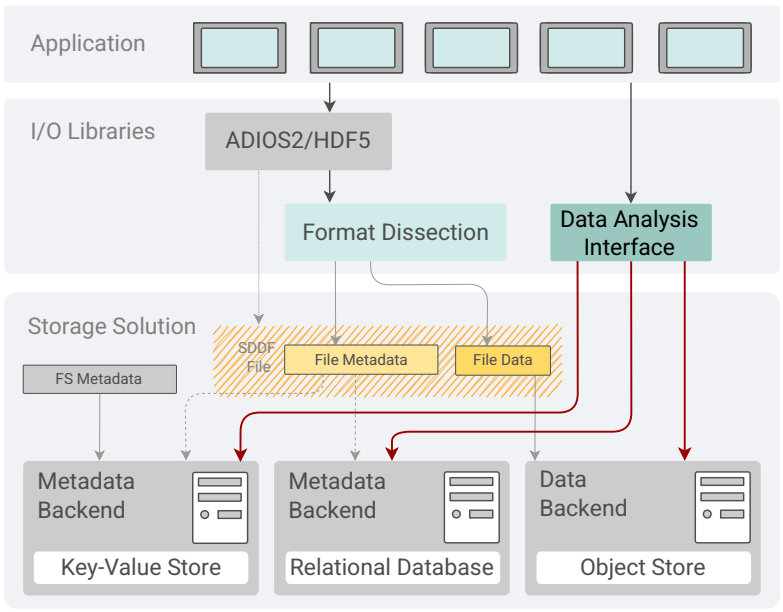
When decades of effort in application development should be preserved, the design space is limited drastically. However, altogether abandoning the time and money spent on previous achievements is hard to sell. The experience of many systems researchers is that new proposals are barely looked into when involving changes to the applications. Therefore, several previously mentioned efforts are combined to improve the current workflows transparent to the application layer. The aim is to enable significant improvements regarding performance and data management through a novel architecture using established technologies from the databases and high-performance computing domains.

The management of self-describing data formats in relational databases was evaluated in EMPRESS 1 and EMPRESS 2 and found to be useful [Lawson et al., 2017, Lawson and Lofstead, 2018]. However, Zhang et al. discussed the drawbacks in detail, among other data duplication and inconsistencies [Zhang et al., 2019, Byna et al., 2020]. So inspired by EMPRESS and dedicated storage solutions, in this thesis, the next step is taken, and the dissection of the format itself is proposed. So, the focus lies in making the file metadata more accessible that is already present in the format, in contrast to EMPRESS, which mandates building a metadata model above the I/O library's data model. In Figure 1.7a, this approach is illustrated. The original self-describing data file will no longer be stored.

Both file system metadata and file metadata will be handled by the storage system's metadata backends to enable efficient metadata management and improve metadata performance.



(a) Management of dissected formats



(b) Data Analysis Interface (DAI): A novel interface to file metadata

Figure 1.7.: Proposed management of self-describing data formats: (a) dissecting the format inside the I/O library and managing the different parts in dedicated storage backends like key-value stores, relational databases and objects stores. The original self-describing data file is no longer stored as indicated by the hatched texture. (b) provide a new interface to the original file metadata and data, and all novel derived and pre-computed metadata.

To this end, the storage system and self-describing data format will be closely coupled to make available structural information. The file metadata can be explicitly targeted by separating it from the data. A prototype, mapping file metadata to key-value stores, is explored as part of this question, whereas the mapping to relational databases will be discussed in detail in research question 2.1.

To offer the required backends, the distributed file system layer has to be switched to a different storage solution. The motivation to use the JULEA storage framework for this task is explained in detail in Chapter 4. The application layer remains untouched when the format dissection is integrated into the I/O libraries while the data storage approach is completely exchanged.

Nevertheless, some changes are necessary to address the discussed challenges. Thus, a new interface for data analysis (DAI) is developed targeting the post-processing applications that typically are considerably smaller and, therefore, easier to adapt. For example, changing a complete climate model and the respective simulation is a substantial effort. However, adapting visualisation and analysis scripts is not as invested. The DAI, as illustrated in Figure 1.7b, allows direct access to the storage backends while offering a user-friendly interface that abstracts from the underlying complexity. So, no knowledge of the database schema is required. The details of the DAI are part of the following research questions.

RQ 2: What optimisations become possible when the format is split?

The proposed separation of file metadata and file data inspires several new questions.

- 2.1: How can metadata and data be queried efficiently?
- 2.2: What custom metadata is interesting?

RQ 2.1 - How can metadata and data be queried efficiently?

To answer question 2.1, a prototype was developed that revealed suboptimal querying performance giving rise to the question of how this can be optimised. Additionally, a user survey was conducted to gain insights into essential requirements. It also entails questions about interface preferences and that will be discussed as part of research question 3.

To address the performance optimisation, further mappings to different database technologies need to be explored. As a first step, the data access patterns are analysed to find appropriate candidates. This analysis is done in the form of an in-depth examination of recent evaluations in the literature. The results show that a wide variety of patterns can be observed even for HPC workloads simply writing periodic checkpoints. Non-contiguous patterns are even more diverse and challenging [Kang et al., 2021]. Therefore, typical data warehouse approaches are not viable as they work on the assumption that queries can be clearly defined. Big data technologies such as data lakes, on the other hand, emphasise, among others, dealing with very different data forms and formats. As this thesis focuses on self-describing data formats, these capabilities are not required. Thus, relational databases arise as the most suitable technology for improved data access. As mentioned previously, a new interface was required to use the file metadata's new availability fully. Also, further information about the format

usage was vital to inform the design decisions, for example, how many variables are used or how deep the hierarchies typically are.

RQ 2.2 - What custom metadata is interesting?

"All the derived data that the scientist produces must also be carefully documented and published in forms that allow easy access. Ideally much of this metadata would be automatically generated and managed as part of the workflow, reducing the scientist's intellectual burden." [Gray et al., 2005]

At this point in the dissection, the file format no longer imposed limitations. The data characteristics stored in ADIOS2, like minima and maxima, inspired the next question: What kind of derived and novel metadata could be beneficial? To provide an answer, typical post-processing operations were studied to design several use-cases that led the design and implementation efforts. One example is to pre-compute other statistics like the average because post-processing applications in climate research often work primarily on them. When averages are computed during the I/O process and stored as part of the metadata, future data access can be reduced or even avoided altogether. In line with the above quote from Gray et al., the computation will be automatised as much as possible.

The most benefits of novel metadata are gained by focusing on cases that neither require expensive additional computation nor produce large amounts of data. As this thesis is heavily motivated by the problems found at the German Climate Computing Centre, the climate data operators (CDOs) were investigated to determine a subset that can be automatically computed during I/O. Two types of operations were selected: statistics like mean, sum and variance and the climate indices. An example of a climate index is the number of frost days in a year. Frost days are all days with a minimum temperature below 0°C.

RQ 3: How to design an analysis interface for file metadata?

- 3.1: What functionality is required for the DAI interface?
- 3.2: How can this approach be generalised?

RQ 3.1 - What functionality is required for the DAI interface

In order to provide a valuable and intuitive API for users, their preferences are vital. Therefore, a survey was performed. It investigated the usage of self-describing data formats on the one hand and interface preferences and used programming languages on the other hand. Furthermore, typical post-processing operations were evaluated as well to inform the pre-computation options. The use-cases developed as part of RQ 2.2 were also used to answer this question. Three main feature categories were designed and implemented. First, several options are offered to tell the system which operations should be pre-computed automatically. These can be done at several granularities and different time intervals; for example, compute the global mean or the sum over 24 steps. Second, the interface provides functionality to tag specific features, for example, all blocks with a maximum temperature above 25°C. Third, the DAI presents numerous ways to read file metadata from the storage backend. The user can retrieve file metadata and data without knowing about the underlying schema. Again

different granularity levels are offered in terms of data and time entities. The interface allows posing easy queries to retrieve specific entries.

RQ 3.2 - How can this approach be generalised?

Finally, the question evolved, whether and how this approach can be advanced and refined. Even though most features are explained and demonstrated using ADIOS2 and the BP formats, HDF5 versions are also implemented. Therefore, a certain level of abstraction is already offered. However, the data models differ considerably in relevant aspects, posing severe challenges. In contrast to ADIOS2, HDF5 does not offer an official step concept but allows several ways to structure data over time. For example, a specific array dimension can be used for the time. Another option is to write multiple datasets, each containing a specific step, or even to write a separate file for each step.

Thus, a generalisation of this variety is impossible without significant abstractions that, in turn, require adapting or giving up on the current I/O library interfaces. Otherwise, the ways data can be represented must be significantly restricted and limited. A third option is presented in the form of a tracking function which can cope with unintended data modelling. For example, the model of ADIOS2 is not intended to store different features in a single variable. Suppose the user wants to store the air pressure and the temperature. In that case, two variables should be used instead of storing both in one variable where every other value is the air pressure, respectively the temperature. However, assume the application already employs this usage. Then, the tracking function allows for treating the air pressure, for example, as a separate variable, thereby enabling the advantages of the format dissection.

Lastly, adjusting the DAI for different scientific fields besides climate research is possible. There are common features across various disciplines in terms of used post-processing operations, as seen in the user survey results. For example, reductions like extrema and sum are frequent, as well as comparison to observation data or creating histograms. Additionally, the option to allow the computation of arbitrary mathematical operations is discussed in detail.

1.4. Thesis Outline

In Chapter 1, the motivation of this thesis is explained by examining the state of the art and the respective problems. Furthermore, the research questions are outlined, and the developed solutions are summarised. Chapter 2 provides background information on I/O libraries and their self-describing data formats, on relational and NoSQL databases, as well as file systems and JULEA. The related work is presented in Chapter 3, mainly focusing on solutions at the file and storage system level and the improved management of self-describing data formats.

The format dissection using key-value stores is explained in Chapter 4. First, details about the format and library internals are given, which are then used to derive a subset of required features to maintain the original behaviour throughout the dissection. The implementation is demonstrated for both ADIOS2 and HDF5. Chapter 5 starts with analysing data access patterns and the first half of the user study. Afterwards, the mappings of the formats to databases are designed and implemented. In Chapter 6 the DAI and advanced functionality are presented to generalise the approach. Chapter 7 covers the evaluation in the forms of

synthetic benchmarks, a real-world simulation and a query application demonstrating the query features of the DAI. Finally, this thesis concludes with a summary and an outlook on future work in Chapter 8.

Chapter Summary

This chapter offered the motivation why dealing with HPC storage problems is a vital task. After the issues with the current I/O stack and the hardware hierarchy, the usage of self-describing data formats was detailed, and the gap left by related work such as EMPRESS or MIQS was discussed. Then, the derived research questions were introduced: How to solve problems with state of the art? What options arise when the format is split? How to design a file metadata interface? Finally, an overview of the proposed solution was given. The main idea is to dissect the self-describing format inside the I/O library and forward the file metadata and data to the JULEA storage framework. File metadata will be stored in the key-value store backend or the database backend, while the data will be stored in the object storage backend. This opened the door for further extensions, such as pre-computing statistics like the variable mean. To query the custom metadata efficiently, a new interface had to be designed that works directly on the JULEA clients.

2. Background

This chapter provides the necessary background information and terminology for the following chapters going from the application layer to the file systems. The chapter is split into the following sections:

- *Section 2.1: presents I/O Interfaces and the corresponding self-describing data formats*
- *Section 2.2: introduces databases and their data models with a focus on NoSQL databases*
- *Section 2.3: explains local as well as parallel and distributed file systems*
- *Section 2.4: describes the JULEA framework and its architecture.*

2.1. I/O Interfaces and Data Formats

In the following, different I/O interfaces and the corresponding self-describing data formats are explained. First, the results from the user survey regarding the used data formats are presented. The mentioned formats are then discussed throughout the rest of the section. As HDF5 is the most common format, it is described at the beginning, followed by ADIOS2. Here more details are given because ADIOS2 and its BP formats are the focus of this thesis.

2.1.1. HDF5

The goal of designing a new file format was to ease sharing of scientific data between different systems [Prabhat and Quincey Koziol, 2014]. Furthermore, the aim was to offer a more diverse data management functionality. Favoured goals were efficiently storing and accessing large files, the possibility for a file to contain different types of objects, the expandability of the format, and providing access to the data from C and Fortran. Eventually, the resulting format was called Hierarchical Data Format. In 1996, in order to drastically enhance the computing power of the Department of Energy's (DOE) systems, the HDF group started working together with the Advanced Simulation and Computing (ASC) program of the DOE [Prabhat and Quincey Koziol, 2014]. The rise by several orders of magnitude in computational power significantly affected data management, as HDF could neither handle the desired large file sizes nor provide parallel access at that time. To create a file format capable of fulfilling the requirements of the ASC, the three DOE laboratories and the NCSA formed a new group called Data Models and Formats (DMF). Combining HDF with the Array I/O (AIO) format from Livermore, they developed a new format they referred to as Big HDF at first. Ultimately it was called "HDF5" as the original HDF's latest version was 4.0.

HDF5 Data Model

Today, HDF5 provides a data model as well as a file format and an according library to manipulate and manage HDF5 files as discussed in the following. Figure 2.1 shows the general structure of an HDF5 file. *Files* are the fundamental object in HDF5. They contain all the information. They can be accessed using a name, in this case, the POSIX path. *Groups* and *Datasets* are so-called primary objects.[1] Groups allow hierarchical structuring of the HDF5 namespace through parent/child relations. They can contain other groups or datasets. The latter are used to store the actual data as an array along with additional metadata. They are accessed using a name. *Dataspaces* represent the dimension of a data object, whereas *datatypes* characterise the data elements. As they can become very intricate, datatypes can be defined for a file and pointed to by all datasets of this type. Thereby the redundancy in the metadata is decreased. *Attributes* can hold additional so-called user-defined or application-defined metadata.

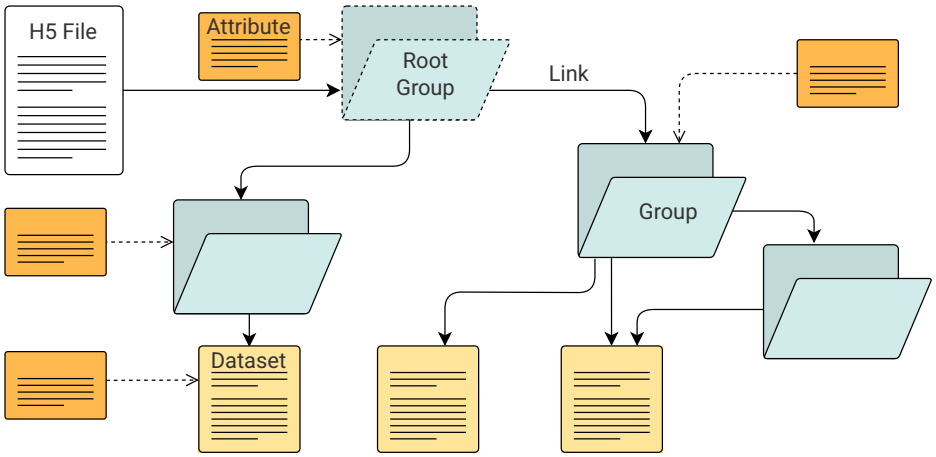


Figure 2.1.: The structure of the HDF5 data model. Groups (light teal), can contain other groups or datasets (yellow). The root group is created automatically with the name '/', as indicated by the dotted object. Attributes (orange) are not stored using links (black arrows) as marked by the dotted arrows. Instead, attributes are stored within an object and cannot be accessed separately. Based on [Prabhat and Quincey Koziol, 2014]

HDF5 Infrastructure

HDF5 offers an abstraction layer called Virtual Object Layer (VOL). VOL connectors enable redirecting the I/O calls that are made using the HDF5 interface to a different location. For example, a write call with the HDF5 interface can be intercepted in the library and the data could be stored in a database. The corresponding architecture is shown in Figure 2.2.

The Virtual File Driver Layer (VFDL) offers a similar concept with the difference, that all calls will need to pass through the native VOL connector before I/O interfaces like POSIX or

[1] "In the HDF5 file format, a primary object is an object that has an absolute address within a file associated with it. A complete description of the object, including its structural and application-defined metadata and its raw data, can be found by decoding information stored at that file address".`https://cdn.earthdata.nasa.gov/conduit/upload/492/ESDS-RFC-007v1.pdf`

MPI-I/O can be used. VOL connectors come in two versions, pass-through and terminal. Pass-through connectors can be stacked, which allows chaining of different components. They require a terminal connector to finish which cannot be stacked.

A VOL connector can be implemented in several ways:

- As a Library:
  The VOL connector can be realised as a shared or static library. The software that should use the connector has to be linked against it.
- As a Plugin:
  Plugins have to benefit that they can be loaded dynamically. By setting the VOL environment variable the plugin can be specified without any changes to the application.
- Internally:
  A connector can be implemented inside the HDF5 library which means that a library fork is required.

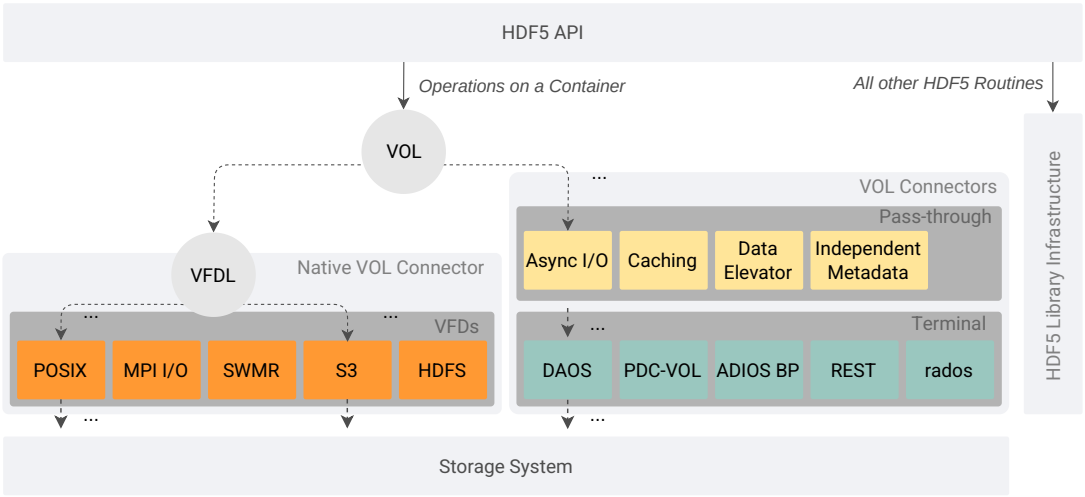


Figure 2.2.: HDF5 Architecture highlighting the different internal layers. These are the Virtual Object Layer (VOL) and the Virtual File Driver Layer (VFDL). The Virtual File Drivers (VFDs) are coloured in orange. The pass-through VOL connectors are marked in yellow, while the terminal VOL connectors are coloured in teal. Based on [Breitenfeld et al., 2020, Robinson and Lee, 2021]

2.1.2. ADIOS2

This subsection shares content with the following publications [Duwe and Kuhn, 2021a, Lüttgau et al., 2018].

The Adaptable IO System (ADIOS) has been designed to solve some of the problems that come with the approaches discussed above. The basic objective is to provide a library of tuned I/O routines to serve as a middleware on a wide range of hardware [Lofstead et al., 2008, Liu

et al., 2014]. Often, applications are highly optimized for specific system architectures to increase performance. As scientific code has a long lifetime, it is executed on several generations of supercomputers. Therefore, changes are required to fully exploit the respective system's capabilities. ADIOS offers the possibility to perform I/O using an abstract definition of the application's data structures in an XML file. This definition is then used to automatically generate code to perform the actual I/O, which allows decoupling the I/O portion of the remaining application code. Once the old I/O calls are replaced by the automatically generated code, there is no need for future recompiling as the implementation of the I/O behaviour is no longer in the application. The actual I/O functionality is realized by so-called *engines* acting as different backends for several self-describing data formats such as ADIOS's own BP (binary packed) format, HDF5, and NetCDF. Moreover, ADIOS supports advanced functionality such as on-the-fly data transformations [II et al., 2014].

Applications have different I/O patterns. So, ADIOS was designed to group I/O operations that can be performed and tuned together, such as restarting or analysis. Each group can have a different approach. The main idea is to separate the scientific source code from the actual I/O implementation. Furthermore, tool integration is required, e.g. for monitoring runs to avoid the previous workflow in Lustre which indicated the end of writing or other problems by writing specific files. That means monitoring was watching out whether specific files were created which hinders the I/O. The separation of application and I/O routines, allows to integrate analysis and visualisation routines without any changes to the source code. Using an XML file to configure I/O routines, also make the ADIOS API relatively easy to use.

A key feature is to select optimal I/O routines for a system without recompilation. Also, integration into workflow systems like Kepler and visualisation systems like Visit easier [Lofstead et al., 2008]. The library offers tuned I/O and various transport methods.

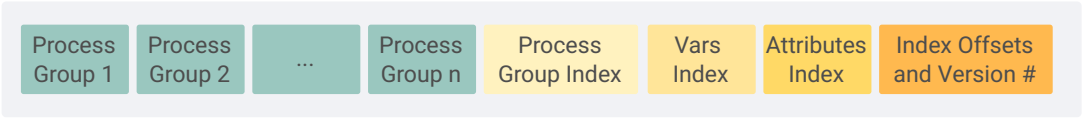


Figure 2.3.: Binary Packed (BP) file layout: Process groups (teal) combine operations that can be performed and tuned together. The different indices (light to dark yellow) to track the process groups, the variables and attributes are stored at the end of the file, together with the index offsets and the used format version (orange). Recreated from [Lofstead et al., 2009], © 2009 IEEE

BP Format BP means binary packed [Lofstead et al., 2009]. File formats like HDF5 and NetCDF owe their popularity in part to the wide variety of tools operating on these formats. One major problem is, that they were not designed with parallel access in mind. And while parallel versions have been developed which show good performance for terascale systems, design decisions for the serial access hinder the scalability required for petascale systems. Evaluations showed that ADIOS could reduce the performance by three orders of magnitude compared to parallel HDF5 [Lofstead et al., 2009]. An improved parallel HDF5 was still two orders of magnitude slower than ADIOS.

Converting BP to HDF5/NetCDF is simple and fast enough that it is often still faster to use ADIOS and BP and convert the format afterwards. One major drawback of HDF5 was the coordination required for collective output. A large number of MPI broadcasts were required to guarantee data consistency[Lofstead et al., 2009]. ADIOS offers delayed consistency. Also, while the development should have consistency checks, the ADIOS developers argue that production runs should not deploy them.

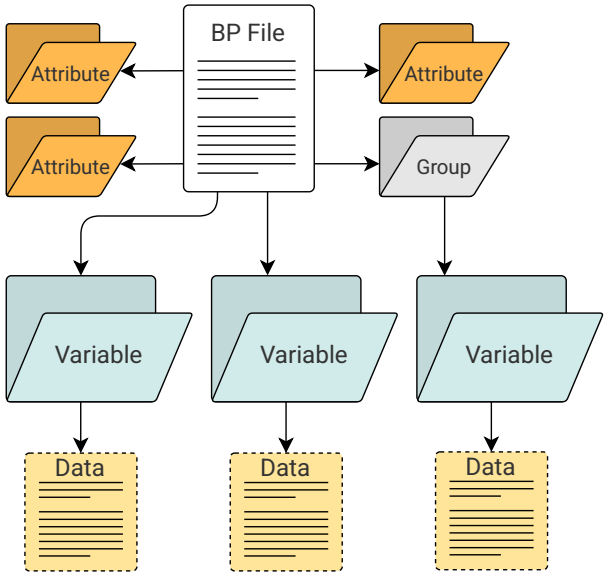


Figure 2.4.: ADIOS2 Data Model: In contrast to HDF5, data (yellow) is not a separate object to be created by the user (dotted lines). Instead, it is part of the variable. Attributes belong to the entire BP file and cannot be directly linked to other objects as in HDF5. The group concept (grey) is displayed for completeness but it was introduced recently and is not frequently used or mentioned.

One feature of large interest for this thesis is the possibility to find interesting data aspects. While content indices to index the complete data have been developed for HDF5, they have considerable space and time overhead.

ADIOS approach is to gather metadata called data characteristics[Lofstead et al., 2009]. These be can local statistics or data analysis performed during the I/O. They can help to determine the interesting data. They are cheap statistics like minima and maxima that can be collected for every process. It is also possible to perform more expensive operations like computing the standard deviation or Fourier transforms. These data characteristics can be stored in the BP format. Learning from other formats such as NetCDF, BP does not have a header but uses a footer. This allows easy data appending and faster access and avoids costly data movement when the header length is changed. The index includes the data characteristics and is part of the footer.

Data Model

The ADIOS and ADIOS2 data model has some similarities to the HDF5 data model. The largest difference is the focus of ADIOS2 on variables and attributes, originally completely discarding any other object types for hierarchical structuring such as groups or links, as found in, e.g., HDF5 [Godoy et al., 2020]. Recently, groups were introduced allowing more namespace structuring. They are however not prominently mentioned and are not frequently used yet. Attributes can only belong to a complete file, in contrast to HDF5 where an attribute can be attached to any object. That makes their usage different, as they cannot easily be mapped to the variable they might belong logically to. The data is arranged in steps which are subdivided into blocks. A block is the data portion that is written by one MPI process in a step.

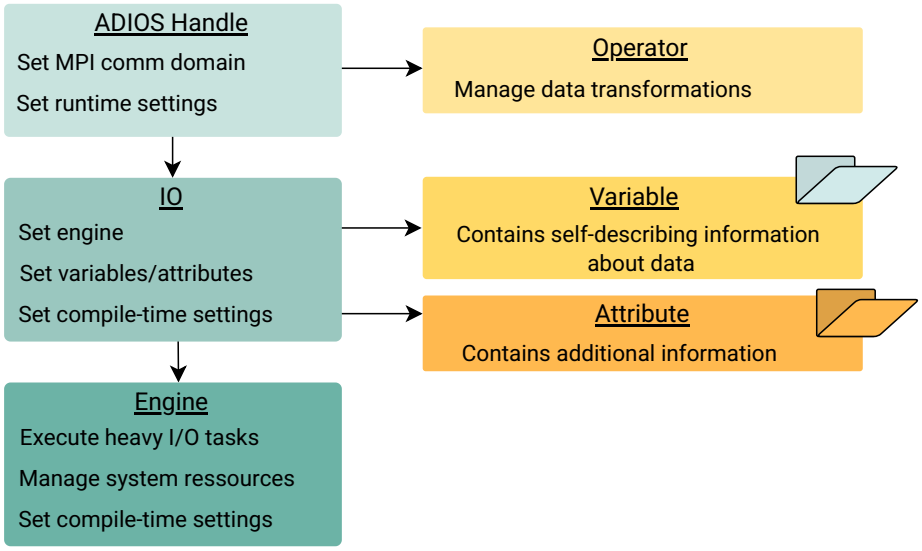


Figure 2.5.: ADIOS2 interface components and their functionality. The I/O path is coloured from light to dark teal while the transformation and data-related objects are coloured from light yellow to orange.

IO Interface Components In the following, the interface components of ADIOS2 shown in Figure 2.5 are described. ADIOS2 natively supports MPI-based applications. The respective MPI communicator needs to be passed to the library when initialising the *ADIOS2 handle*. Application-specific parameters are monitored and fine-tuned by the *IO object* that functions as a central control point for user-defined settings [ORNL, 2022]. Furthermore, the IO object handles the variable and attribute definitions and keeps track of them. The actual writing and reading behaviour is determined by the used *engine*. When opening a file at the IO object, an Engine object of the set engine is generated and returned. It will perform the data output and input and communicate with the system's I/O resources. Whether an engine serves as a writer or a reader is defined by the file open mode.

The engine component is built such that it also constitutes the point of application to implement new I/O behaviour. The format dissection is realised by implementing new ADIOS2 engines. The details are discussed in Chapter 4 and 5.

In Listing 2.1, a short example is given of how to write an ADIOS2 variable. Detailed insights into the different ADIOS2 variables are given in Section 4.1. Here a local array is used. Local in this scenario means process local. So, only one process writes this variable, which makes the definition (Line 8 - 9) simpler, as only the local dimensions (`Nx`) are required. In Line 13, the variable is written asynchronously which is termed deferred in ADIOS. Deferred operations are the default in ADIOS2. The mode can be passed as an additional third parameter to the put and get operations. This allows both to switch to synchronous I/O but also to make the deferred mode more visible.

```
1  std::vector<float> myF = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
2  const std::size_t Nx = myF.size();
3
4  adios2::ADIOS adios;                          // ADIOS class factory of IO class objects
5  adios2::IO bpIO = adios.DeclareIO("BPFileIO");
6  bpIO.SetEngine("BP4");
7
8  adios2::Variable<float> bpF = bpIO.DefineVariable<float>(
9      "bpFloats", {}, {}, {Nx}, adios2::ConstantDims);
10
11 adios2::Engine bpFileWriter = bpIO.Open("example.bp", adios2::Mode::Write);
12
13 bpFileWriter.Put<float>(bpF, myF.data());   //Write variable for buffering
14 bpFileWriter.Close();                       //Create bp file
```

Listing 2.1: Writing a ADIOS2 variable using the C++ ADIOS2 interface. A float array with the name `bpFloats` of the size `Nx` is written to the file `example.bp` using the BP4 engine. As the variable is a local array, the variable definition takes only one dimension. Apart from the engine name that has to match any of the existing engines, all other strings can be arbitrary.

Engines and the BP File Formats

In ADIOS2, the file format and the engines share the same names. So, the BP3 engine writes and reads the BP3 files. Note, that there is no file ending or suffix indicating which format version is used. Initially, this information could only be found by using a custom python script or by explicitly adding the format version to the variable name. After some online discussion, the developers agreed that knowing the format version may be useful and added this information to the output of the `bpls` command line tool. This is especially important because it is not possible to overwrite existing files of the same name with another format. For example, a BP3 file, named `test.bp` cannot be overwritten with a BP4 or BP5 file of the same name. The engines react differently to this scenario. While BP3 runs into a deadlock, BP4 crashes and BP5 throws an error.

In Figure 2.6 the different internal ADIOS2 layers are depicted. Similar to HDF5, ADIOS2 defines several interfaces that can be used to introduce new I/O behaviour. These are the

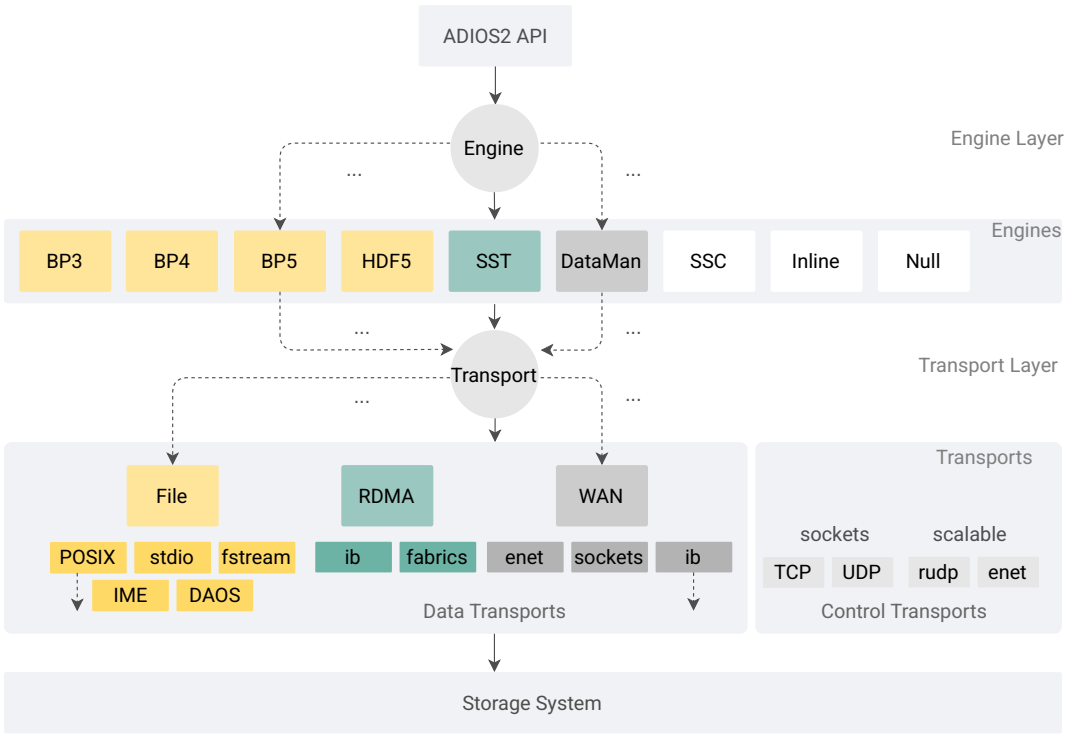


Figure 2.6.: ADIOS2 Engines and transports: They are the equivalents to VOL plugins and VFL drivers and allow to determine the I/O path and redirect the I/O operations. The file-based engines and transports are coloured yellow, the RDMA-based teal and the engines and transports for WAN data staging grey.

previously mentioned engines and transports [Klasky et al., 2010]. The native BP engines only support file-based transport types. The term *BP file* is a bit misleading. All BP formats use files and directories to store one BP file. Those for BP3 are shown in Listing 2.2.

BP4 In contrast to BP3, BP4 uses an index table to keep track of the chunked metadata and is reported to maintain a relatively constant overhead with an increasing step number [Feldman, 2019]. The parameter table can be found in the Appendix D.1. Note that each file format creates a different set of files those belonging to BP4 are shown in Listing D.1

Next, three of the advantages of BP4 over BP3 are mentioned:

- Faster appending of batched steps
- Streaming using files, that is, in-situ processing
- Burst buffer support writing the data first to the node-local file system and then asynchronously to the parallel file system later.

BP5 With version 2.8, ADIOS2 introduced a new format called BP5. It focused mainly on improving metadata management. The format file details are listed in Appendix D.4. Memory

consumption is reduced by using zero-copy whenever possible, thereby consuming about half as much as memory BP4 in ideal cases. All buffers are reused in deferred write operations. Also, MPI communication and thereby buffer copying are reduced through shared memory on a compute node. Furthermore, new files are created again as shown in Listing D.2.

```
x.bp          // collective metadata file

x.bp.dir/     // data directory and data subfiles
  x.bp.0
  x.bp.1
  ...
  x.bp.M
```

Listing 2.2: BP3 File structure: A BP3 file is not a file in the traditional sense as it consists of a file (x.bp) that stores the collective metadata and directory (x.bp.dir/) that contains the data subfiles (.bp.0 - .bp.M). The number of subfiles (M) by default equals the number of nodes that are used during writing. It can be configured using `NumAggregators`. The profiling option is set by default and a profiling.json is written as well. An example profiling.json can be found in Appendix D.

2.2. Databases

Databases have been around for a long time. Davoudian et al. gave great insight in the evolution [Davoudian et al., 2018]. Their findings are summarised in the following. The first steps in the early 1960s were general-purpose database systems. They aimed to provide convenient and efficient data access through offering hierarchical data models linking data entries together. However, the data dependence complicated application programming and lead to the development of the second generation. These had relational data models using tuples to describe relations thereby increasing the data independence. That is when SEQUEL (Structured English Query Language) was developed. It is based on relational algebra and was then adapted by Chamberlin and Boyce to be more accessible to users without intensive mathematical training [Chamberlin and Boyce, 1974]. SEQUEL was later renamed SQL due to copyright problems. Even today, SQL is still an active research topic as can be seen in the endeavours to parse natural language to SQL (NL2SQL) [Kim et al., 2020]. In the mid 1980s, the spreading of object-oriented programming resulted in the need for an appropriate database model. Combined with the limited modeling capabilities of the relational data model, this led to the third generation of database systems. Following the programming trends, they were designed to conform to the object-oriented approach, identifying objects via OID and using a hierarchical structure to model the object feature inheritance. However, the high investments into relational database (RDB) systems hindered the adoption of object-oriented databases. Through extensions to prominent RDB systems, such as DB2 and Oracle, important object-oriented features were incorporated, leading to object-relational database systems. About 20 years later, applications in the area of web technologies and the Internet of Things (IoT) started to

present another set of challenging requirements. These are in part mutually exclusive and listed in the following.

- Horizontal Scalability: making use of additional resources to deal with the increasing data sizes.
- High Availability and Fault Tolerance: guarantee stable service even in the face of hardware failure
- Transaction Reliability: provide strong consistency guarantees
- Database Schema Maintainability: make the evolution of database schema less costly

These requirements cannot be easily satisfied through RDBMSs. For one, the pre-defined schema hinders the database evolution. Also, fixed schemas are not suitable for the agile development approaches found, for example, in the area of Big Data. Additionally, scaling up needs dedicated and often specialized hardware that introduces the need for costly data movement. Lastly, scaling out complicates the join of distributed data. The use of ACID transactions also brought challenges regarding availability and performance. Sacrificing some requirements for the sake of others, more essential to the demands of the application, led to the fourth database generation and the development of numeral non-relational storage systems, namely NoSQL stores. More requirements for high scalability and reliability stimulated the development of new database technologies that could offer support for large data sizes distributed across various locations. The resulting solutions are called NewSQL stores as they offer SQL queries and ACID transactions without the need for a join operation.

2.2.1. Data Models

As previously discussed, there is a wide variety of database approaches each coming with its own data model. The separation discussed in the following was introduced by Mansouri et al. and helps to categorise the massive number of database systems [Mansouri et al., 2018]. The high-level categories are data structure, data abstraction and the data access model.

Data Structure The data structure significantly influences retrieval performance. There are three categories. (1) In *structured data* relationship is defined between fields identified through name and value, e.g., in an RDB. (2) *Semi-structured data* provides a specific form of structuring that is known to the application. Primitive operations are supported similarly to structured data. (3) *Unstructured data* does not offer any pre-defined data model. Data consistency is traded for better scalability and performance

Data Access The data access model affects what consistency can be guaranteed. It can be grouped into four categories. (1) *Key-value databases*, as mentioned before, provide a simple interface based on unique keys and arbitrary data values. As key-value stores are NoSQL databases, they support the features discussed in the third data abstraction level. (2) *Document databases* offer storage for several kinds of documents that can be indexed. They do not provide ACID guarantees for primitive operations. Therefore, it supports *eventual consistency*. (3) *Extensible record databases* (also called wide-column stores) are similar to tables

where columns can be grouped, while the rows are distributed over several storage nodes depending on the primary key. Each field offers multiple data versions ordered by timestamps. This approach is called NewSQL. (4) As discussed above, *relational databases* use pre-defined schemes and offer SQL queries and ACID transactions.

2.2.2. Relational Databases

There have been various studies trying to identify the best database system for a specific application type. Makris et al. evaluate the response time of PostgreSQL in comparison with MongoDB and find that PostgreSQL outperforms MongoDB for all queries on vessel tracking [Makris et al., 2019]. Performance bottlenecks of relational database systems, e.g., because of the read amplification and high cache miss rates, can be mitigated by employing 3D XPoint storage [Yang and Lilja, 2018].

Feser et al. propose a promising approach, Castor, by developing a new layout algebra to combine the data layout and the queries to be performed [Feser et al., 2020]. Thereby, they are able to transform both the layout and the query in a single rule and allow drastic layout transformations. Besides the layout algebra, they provide an automated deductive optimizer, a type-driven layout compiler, and a high-performance query compiler. This combination of features comes along with a significant limitation. The constructed databases are read-only and are, therefore, tailored towards specific use cases.

In the following, the 15 most popular relational database systems as of July 2021 are listed. The complete list consists of 143 entries. The order is determined by the DB-Engines ranking[2].

While MonetDB is a relational database, it can also be considered a wide-column store because it manages data in columns [solid IT, 2021]. This vertical storage allows more performant queries because CPU cache lines are used more efficiently, and new cache-conscious algorithms are provided [Boncz et al., 2008]. Furthermore, the reduced query algebra makes faster hardware implementations possible. Idreos et al. state that it is a column store that feels like a relational database to the user [Idreos et al., 2012]. Index concepts such as Elf have been integrated into MonetDB to optimize efficient querying on scientific data [Blockhaus et al., 2020]. Another variation, MonetDBLite, was developed to cater to the needs of machine learning and classifications tasks [Raasveldt and Mühleisen, 2018]. It offers fast data transfer between analytical tools and the database while also guaranteeing the ACID properties of relational systems.

2.2.3. NoSQL Databases

The list of NoSQL databases currently includes over 225 databases[3]. They can be divided into several subcategories, of which key-value stores, wide-column stores, document stores, graph stores, and object stores are highlighted. To sort the vast number of NoSQL databases the taxonomy of storage systems can be helpful that is used in big data by Siddiqa et al. [Siddiqa et al., 2017] It is shown in Figure 2.7. Most of the systems are included in the listings in

[2] `https://db-engines.com/en/ranking/relational+dbms`
[3] https://hostingdata.co.uk/nosql-database/

the following subsections. A comprehensive overview of data storage and data placement in cloud environments was published by Mazumdar et al. [Mazumdar et al., 2019].

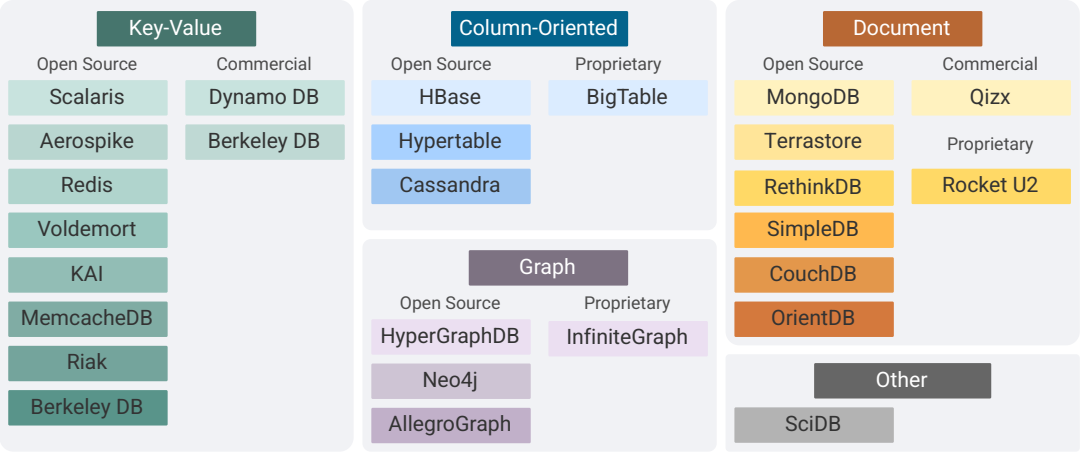


Figure 2.7.: Taxonomy of big data storage technologies based on [Siddiqa et al., 2017].
Adapted by permission (License Number: 5322100180736) from Springer Nature Customer Service Centre GmbH: Springer Nature Frontiers of Information Technology & Electronic Engineering (Big data storage technologies: a survey, Aisha Siddiqa et al.), COPYRIGHT (2017)

NoSQL stores have a wide variety of consistency models ranging from strong consistency to eventual consistency. A good overview is provided by Davoudian et al. [Davoudian et al., 2018].

Key-Value Stores The concept of key-value stores is very straightforward and popular. It consists of a value uniquely identified by a key, e.g., a name or a hash. The value can be data of any type with any structure and size. Data is stored as byte arrays requiring serialization and deserialization that is often performed on the clients. So, indexing or querying on the values is not supported. Using this simple model, access is only possible through the key, limiting the potential applications. Therefore, advanced key-value stores may support data lookup based on the value, for example, by providing list data types. Thereby, Redis and Aerospike offer data updates that do not require the replacement of the whole value. Other extensions are tabular data types with Spinnaker[4], while Riak also supports documents such as JSON. This development blurs the line between key-value stores and other approaches like document stores.

Key-value stores can be grouped according to different criteria. Often in-memory, persistent and hybrid approaches are distinguished. The former is typically used as a caching layer for transient information such as session information. Facebook uses Memcached while LinkedIn employs Voldemort[5]. Key-value stores that are common in the area of storage research are LevelDB [Ghemawat and Dean, 2020], LMDB and RocksDB [Yang et al., 2015, Cao et al., 2020].

[4] https://spinnaker.io/
[5] https://project-voldemort.com/voldemort/

Wide-Column Stores

Wide-column stores are a database concept that has its origin in Google Bigtable [Chang et al., 2008]. The idea is to focus on rows and a pre-set number of columns, often referred to as a column family. Column-families consist of logically connected columns that are stored together instead of rows [Davoudian et al., 2018]. Wide-column stores provide a more flexible schema offering to add or remove columns at runtime. Davoudian et al. call them extended key-value stores that use nested key-value pairs.

Google Bigtable is built for structured data and to provide flexibility and high performance for the variety of Google products [Chang et al., 2008]. A very popular wide-column store is Apache Cassandra [solid IT, 2021]. It was developed to meet the reliability and scalability needs of Facebook [Lakshman and Malik, 2010]. Another prominent wide-column store is Apache HBase which is built on top of the Hadoop Distributed File System (HDFS) [Vora, 2011]. Recently, ScyllaDB has been promoted to function as a drop-in replacement of Apache Cassandra promised to outperform it by up to a factor of 10. One key aspect contributing to the improvement is ScyllaDB's implementation in C++ instead of Java [Suneja, 2019]. ScyllaDB provides large data volumes, high availability, and high performance. Mahgoub reports some performance instability of ScyllaDB with server replication [Mahgoub et al., 2017]. Replacing file systems with wide-column stores is extensively studied. Santamaria et al. compare HDF5 on GPFS with Cassandra or Scylla through Hecuba[Santamaria et al., 2019].

Document Stores Document stores are adapted key-value stores where the value is managed in a document [Davoudian et al., 2018]. Typical formats include XML, JSON, and BSON. Document stores offer flexible schemas that allow updates, for example, to add new attributes. It also enables indexing and search functionality, using the attribute names and values. Some document stores like ArangoDB and orientDB also offer an additional graph representation alongside the document data model by referencing documents. Schultz et al. analyze the tunable consistency model for MongoDB [Schultz et al., 2019].

Graph Stores With the increase in graph-oriented data like the Semantic Web, a data model allowing efficient relationship traversal was required [Davoudian et al., 2018]. This lead to the concept of graph stores where vertices represent entities and edges the relationships. Property graphs are a mixture of various graph types like directed, labelled, and multigraphs, which makes them very flexible.

Time-Series DBMS With the rise of sensor storage, a new data model was required for Cloud computing and Big Data [Mazumdar et al., 2019]. Time-series DBMS typically extend key-value stores or column-stores by building an additional data model on top.

Object Stores In contrast to previous database approaches, object stores provide management solutions for unstructured data. They offer storage of variable sizes for so-called objects that typically store data as BLOBs (binary large objects). The first standardization was undertaken in late 2004 [Factor et al., 2005]. The indexing approaches of object stores are comparable to those of other databases like key-value stores. While object stores were not

designed for HPC environments, they are increasingly employed on large-scale clusters, e.g. to store meteorological and climate data [Smart et al., 2017]. In an effort to mitigate the performance limitations of current file systems, data is extracted from self-describing data formats like NetCDF and HDF5 and stored in object stores. Chu et al. demonstrate the benefits of mapping HDF5 datasets to object storage [Chu et al., 2020].

There is also a growing interest from the IoT (Internet of Things) community to use object storage in fog and edge computing infrastructure as shown by Confais et al. [Confais et al., 2017]. They studied RADOS, Cassandra, and InterPlanetary File System (IPFS) with a focus on access time for writing and reading objects, the network traffic exchanged between different geographical sites, and the influence of network latency.

Lee et al. propose SwimStore, a storage architecture providing a high write performance, low write amplification, and stable performance on top of a file system [Lee et al., 2018]. It was integrated into Ceph and performed better than other Ceph storage backends like FileStore and KStore but similar to BlueStore. BlueStore is a drastic redesign compared to FileStore and KStore. It directly operates on raw block storage, using only a thin user-space file system called BlueFS in combination with RocksDB for the metadata [Aghayev et al., 2019b].

Since there is no ranking of object stores in the DB-Engines ranking, we have compiled a list of prominent object stores in no particular order.

Relational Queries vs. Non-relational Queries In contrast to NoSQL databases, RDBs offer fixed and predefined fields for each entry. NoSQL stores build upon the key-value model or similar variants such as graphs and documents where the entry is linked to a key-value pair. NewSQL databases use directories in the top-level of their hierarchy,e.g., Customer, where each row of the directory table and all the entries in the included tables, e.g., Campaign and AdGroup, form a directory itself [Mansouri et al., 2018]. Thereby, NewSQL builds child-parent relations between all pairs where the primary key of the parents is used as a prefix in the child's primary key.

2.3. File Systems

The chapter shares content with [Lüttgau et al., 2018]

File systems are traditionally part of the operating system and are used to organise data in an accessible way. The POSIX (Portable Operating System Interface) interface was introduced in 1988 to provide standardised data access across different architectures and operating systems. The data management requires to keep additional information called file system metadata, as mentioned previously in the introduction. POSIX was designed for local file systems and thus has severe drawbacks when used in a distributed and parallel environment. With the evolution of computer architectures towards many-core solutions the concept of a local file system is not sufficient any more. Therefore, parallel and distributed file systems were introduced. Sometimes they are also referred to as either parallel or distributed file systems, even though these terms have different meanings. While a parallel file system allows concurrent access to a storage system, a distributed file system distributes the data across different machines. In HPC, both features are required.

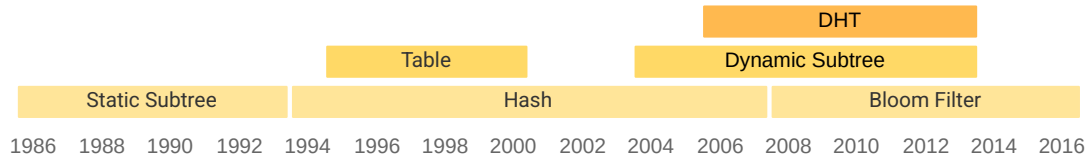


Figure 2.8.: Timeline of metadata management techniques [Singh and Bawa, 2018]. Abbreviation: DHT - Distributed Hash Table Republished with permission of ACM (Association for Computing Machinery), from Scalable Metadata Management Techniques for Ultra-Large Distributed Storage Systems – A Systematic Review, Singh and Bawa, Vol. 51, No. 4, Article 82, 2018; permission conveyed through Copyright Clearance Center, Inc.

The importance of efficient metadata management becomes very clear, by the large number of techniques that have been developed over the last decades. In Figure 2.8 a timeline of management techniques is shown which provides a good overview [Singh and Bawa, 2018]. Singh and Bawa gathered and categorised the efforts that have been performed since the mid 1980ies. The concepts have evolved from static to dynamic metadata distribution in distributed systems.

Parallel and Distributed Kernel File Systems

In the following, popular candidates are described:

Lustre Lustre is the de-facto standard for HPC file systems. The file system's architecture follows the common client-server model. Servers can be employed for different tasks, for example, Object Storage Servers (OSS) store the data as objects, whereas Metadata Servers (MDS) take care of the metadata. As Lustre is a kernel-space file system, most functionality is implemented as kernel modules. Lustre is POSIX-compliant and therefore compatible with existing applications. As mentioned previously, this convenience comes at the price of system calls for every file system operation. System calls induce a significant overhead through mode and context switches. An in-depth description of the Lustre architecture can be found in [Braam, 2019]. To make full use of node-local storage, Qian et al. proposed a hierarchical Persistent Client Caching (LPCC) mechanism, building either a local read-write cache on one SSD or a read-only cache over the SSD's of multiple clients [Qian et al., 2019].

Spectrum Scale Spectrum Scale is a scalable high-performance data management solution developed by IBM for enterprises that need to process and store massive amounts of unstructured data [IBM, 2022]. It is based on the former General Parallel File System (GPFS) [IBM, 2016]. Spectrum Scale offers hierarchical storage management for storage tiers ranging from SSDs to tape. Spectrum Scale is fully POSIX-compliant, which allows it to support many traditional HPC applications. By configuring dedicated servers for metadata updates, performance bottlenecks for metadata-intensive applications can be reduced. Spectrum Scale offers various advanced features; among others a declustered software RAID algorithm, storage pools, and the integration into OpenStack services. Vef et al. performed extensive tracing using the GPFS tracing tool and their new tracing interface prototype FlexTrace, shown to have

negligible overhead [Vef et al., 2018]. FlexTrace offers user-defined tracing profiles to allow fine-grained trace control. Unfortunately, Spectrum Scale is rather expensive and thus not a feasible solution for small research clusters.

BeeGFS Abramson et al. proposed a prototype file caching layer that makes use of the data locality across storage tiers [Abramson et al., 2020]. Also, it increases the data sharing between compute nodes and applications, while using data striping and metadata partitioning to support fast parallel I/O.

Ceph Ceph is another free and open-source system. It is built on Reliable Autonomic Distributed Object Store (RADOS) and offers data storage based on file blocks and objects [Weil et al., 2007, Weil et al., 2006]. RADOS manages replication, data migration and recovery of the system. cluster. Ceph also provides a near-POSIX-compliant file system. Its integration makes the benefits and features of the scalable environment available to applications. However, Ceph still has some drawbacks, for example, the limitation that a cluster can only deploy one CephFS. In addition, Ceph is designed for HDDs and needs further improvements to work well on SSDs and with random data access patterns [Oh et al., 2016].

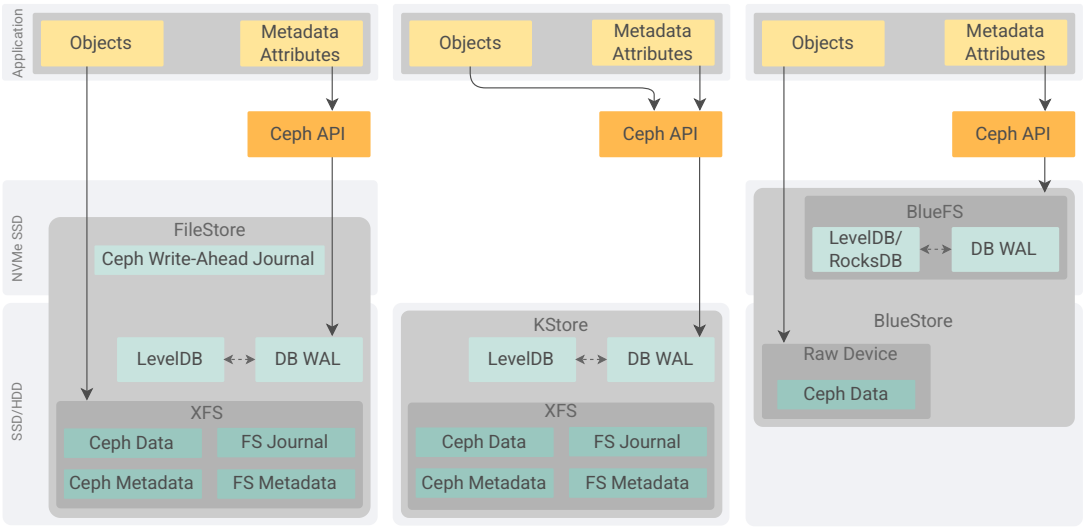


Figure 2.9.: Architecture of the Ceph storage backends FileStore, KStore, and BlueStore. All of them use a database write-ahead log (WAL). For FileStore and BlueFS the WAL that tracks the metadata and attribute calls to the Ceph API. KStore also logs the object calls in the WAL. BlueStore on the other hand runs directly on the raw block device and circumvents most layers. Redrawn from Lee et al.[Lee et al., 2017]

A major shift happened with the development of BlueStore. After more than a decade, the Ceph developers finally turned away from trying to build distributed storage backends on local file systems [Aghayev et al., 2020]. The performance problems of FileStore and KStore lead to the development of BlueStore that works directly on raw block storage, thereby reducing the I/O layers significantly [Aghayev et al., 2019b].

Ceph provides a multitude of storage backends, three of which are shown in Figure 2.9, highlighting their internal architecture differences [Lee et al., 2017]. Figure 2.9 depicts from left to right FileStore, KStore and BlueStore. FileStore is an object store built on top of POSIX where objects are mapped to files. KStore is the evolved version without a separate Ceph journal where objects as well as metadata are handled by a key-value interface. As mentioned before, BlueStore is very different in that it gives up the local file systems entirely as shown on the right in Figure 2.9.

BlueStore BlueStore was designed and implemented in 2015 and has been the default storage backend for production systems of Ceph since 2017 [Aghayev et al., 2019b]. One of the main motivations to replace local file systems completely is their lack of transactions and scalable metadata operations. Transactions in distributed file systems are typically emulated through POSIX and a write-ahead log (WAL), as can be seen in Figure 2.9.

BlueStore, however, writes the data to the raw storage device using direct I/O. The metadata is managed by RocksDB, which runs on a thin user-space file system called BlueFS [Aghayev et al., 2019b, Aghayev et al., 2019a]. Furthermore, low-level file system metadata is stored in key-value stores such as extent bitmaps. Their reference counting and the clone operation have also been optimized. Lastly, BlueStore makes use of a space allocator that uses a fixed memory size per TB of disk space. There are also plans for support of host-managed SMR (Shingled Magnetic Recording) in BlueStore using a new zone interface. Therefore, RocksDB and thereby LSM-Trees need to be adapted to run on SMR drives [Aghayev et al., 2020]. Further work to support SMRs has been done by Aghayev et al. [Aghayev et al., 2019a].

2.3.1. User-Level File Systems

In contrast to kernel space file systems such as Lustre, user-level file systems do not require any kernel modules to run. This typically makes it easier to use such file systems in a supercomputer environment, where users normally do not have root privileges.

OrangeFS OrangeFS is open-source and employs dedicated data and metadata servers. It uses local POSIX file systems for data and stores the metadata either in Berkeley DB (BDB) or Lightning Memory-Mapped Database (LMDB). OrangeFS provides MPI-IO support through a native ADIO backend and offers a wide range of user interfaces, including several Direct Interface libraries, a FUSE file system, and an optional kernel module [Vilayannur et al., 2004].

GlusterFS GlusterFS is another POSIX-compliant, free, and open-source distributed file system [Depardon et al., 2013] following the client-server model. It only offers file-based storage. Interfaces for block and object access have to be built on top. GlusterFS locates files algorithmically using an elastic hashing algorithm. By avoiding a metadata server the performance is improved while ensuring linear scalability and reliability [Selvaganesan and Liazudeen, 2016].

Adhoc File Systems Ad hoc file systems offer a way to include node-local SSDs or NVRAMs into a temporary storage system [Brinkmann et al., 2020]. There are still a lot of open questions regarding this approach, e.g., how to use these file systems in combination with a present scheduling environment. The lifetimes of ad hoc file systems vary greatly and can be as short as the runtime of a single job. They are mostly implemented in userspace. Prominent examples are **BeeOND**, **GekkoFS** [Vef et al., 2020a] and **BurstFS**. Using BeeGFS as an ad hoc file system, BeeOND was developed. It is often used for data buffering. The main idea of GekkoFS is to distribute data and metadata evenly across a cluster. GekkoFS employs hashing at the level of file inodes to determine the cluster node managing it. Spreading the metadata across all participating cluster nodes required significantly relaxed POSIX directory semantics. Therefore, applications frequently listing or renaming directories should not be run on GekkoFS. Furthermore, it does not handle conflicts like overlapping file regions and shifts the responsibility to application developers. GekkoFS outperformed Lustre by a factor of ~1400 for file creation regardless of whether a single or unique directory was used [Brinkmann et al., 2020]. BurstFS is very similar to GekkoFS. The main difference is the log-structured writing of BurstFS. Thereby, the write performance is not limited by the network latency. However, metadata directories are required to reconstruct multi-client writes to a single file.

Sirocco Sirocco takes a very different approach to any of the above. It aims at redesigning the complete storage system design[Curry et al., 2016] by using a concept similar to unstructured peer-to-peer. They position themselves against existing systems like Lustre, GPFS, and PVFS because these all are inspired by the Zebra file system [Hartman and Ousterhout, 1995] meaning that they use data striping. Sirocco has some very unconventional design principles. It values write performance and scalability over most other objectives. First, there is no central indexing for data. Data can be stored by any reachable server. Furthermore, data is continually moved within the system to offer integrity and longevity. Clients are not updated to store where the data is moved to. They can, however, specify a data protection level. Due to the write-back caching, explicit synchronization calls are required to persist data. The focus on writing comes at the cost of losing track of where data is stored and in which state. In the worst case, an extensive search over the storage servers is required. This need for a full scan is the most severe drawback of Sirocco.

2.4. JULEA

As mentioned in the introduction, applications in HPC typically do not need the strict semantics enforced by the POSIX interface [Wang et al., 2021]. JULEA was built to resolve the resulting performance problems by offering custom and dynamic storage semantics. Applications can now specify the various requirements regarding atomicity, concurrency, consistency, operation ordering, persistency and safety [Kuhn, 2017]. These detailed and configurable features allow to adjust the file system's behaviour to suit the application's needs. In Figure 2.10, the I/O stack using JULEA is shown. By replacing the distributed file system layer with JULEA more of the stack moves to the user-space.

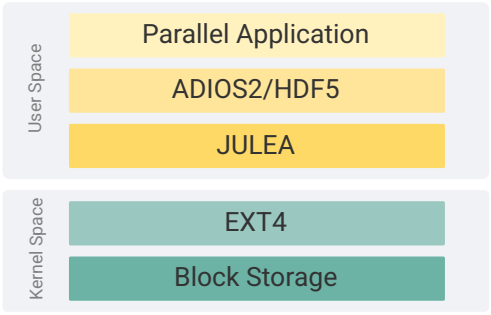


Figure 2.10.: I/O software stack with JULEA. The parallel and distributed file system has been replaced by JULEA which runs in user space (from light to dark yellow) and therefore reduces the number of layers in kernel space (from light to dark green). Furthermore, the I/O library layers are combined, resulting in a smaller stack.

2.4.1. Architecture

While JULEA was originally developed as a parallel and distributed file system it has been revised and is now a storage framework. By reworking the overall structure JULEA became more modular and customisable [Kuhn, 2017]. JULEA provides an environment to research storage systems and adopts new ideas with immensely reduced overhead in terms of the training period and implementation effort. Therefore, it is not only suitable for studying novel research approaches but also for teaching purposes. JULEA runs completely in user space, which eases development and debugging. The framework has different components that work as building blocks for setting up a custom storage system. They are shown in Figure 2.11.

JULEA offers various interfaces to serve a wide range of application types. While applications can directly interact with one or more clients, it can also make sense to use them in an I/O library. For example, HDF5 and ADIOS2 can use the JULEA clients in a VOL plugin or engine to redirect the I/O from the native path to the JULEA backends. At the beginning of this thesis, JULEA already offered object and key-value servers, which allowed tuning for their respective access patterns. In Chapter 5, another backend type is developed that works on relational databases. The introduction of a novel backend type demonstrates the flexibility of the framework over the initial file system prototype. The modularity keeps the necessary effort of adding new features to JULEA comparatively low by reducing the points of contact between the different components.

2.4.2. Clients and Backends

In contrast to introducing a new backend type, adding another backend for an existing backend type is even easier. In the following, the most important functions of the client and backend interfaces are listed and discussed.

Clients The client interface of the key-value store is shown in Listing 2.3. These are the functions that an application uses to store its data in a key-value store. The actual technology

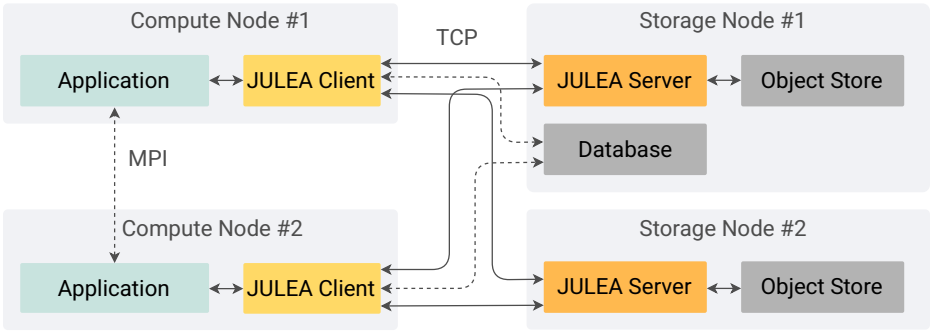


Figure 2.11.: JULEA setup with two compute and two storage nodes. The JULEA components are coloured in yellow (clients) and orange (server). The respective backends are marked in dark grey (database and object stores). The parallel application (light teal) communicates across the different nodes using MPI. The network communication is based on TCP.

that is used underneath can be exchanged easily without introducing changes to the application. For example, switching from LMDB to LevelDB only requires updating the JULEA configuration. No modifications to the application are needed. This abstraction from the underlying technology allows users to concentrate on the general functionality of their applications while the specifics are hidden in the respective backend implementation.

As shown in Listing 2.3, the interface is consisting of a small number of functions directly related to the data storage. Besides creating and deleting, putting and getting single key-value pairs (Lines 1-4), the interface offers an iterator for more efficient access to multiple pairs (Lines 6-8).

```
1 JKV* j_kv_new(gchar const* namespace, gchar const* key);
2 void j_kv_put(JKV* kv, gpointer value, guint32 len, GDestroyNotify free_func, JBatch* b);
3 void j_kv_delete(JKV* kv, JBatch* b);
4 void j_kv_get(JKV* kv, gpointer* value, guint32* len, JBatch* b);
5
6 JKVIterator* j_kv_iterator_new(gchar const* namespace, gchar const* prefix);
7 gboolean j_kv_iterator_next(JKVIterator* iterator);
8 gchar const* j_kv_iterator_get(JKVIterator* iterator, gconstpointer* value, guint32* len);
```

Listing 2.3: The most important interface functions of the JULEA key-value store client

Backends The functions that need to be implemented by the respective storage technology are presented in Listing 2.4. When adding a new key-value backend to JULEA, the definition of these functions guarantees that the existing JULEA infrastructure can talk to the new backend. In JULEA, all operations need to be added to a batch (Lines 4-5). It is possible to execute a single operation in a batch. However, to keep the management overhead small and reduce the network communications, it is sensible to gather multiple operations in a batch. The backend functions matching the client interface functions discussed previously are listed in Lines 7-9 and 11-13.

```
gboolean j_backend_kv_init(JBackend*, gchar const*);
void j_backend_kv_fini(JBackend*);

gboolean j_backend_kv_batch_start(JBackend*, gchar const*, JSemantics*, gpointer*);
gboolean j_backend_kv_batch_execute(JBackend*, gpointer);

gboolean j_backend_kv_put(JBackend*, gpointer, gchar const*, gconstpointer, guint32);
gboolean j_backend_kv_delete(JBackend*, gpointer, gchar const*);
gboolean j_backend_kv_get(JBackend*, gpointer, gchar const*, gpointer*, guint32*);

gboolean j_backend_kv_get_all(JBackend*, gchar const*, gpointer*);
gboolean j_backend_kv_get_by_prefix(JBackend*, gchar const*, gchar const*, gpointer*);
gboolean j_backend_kv_iterate(JBackend*, gpointer, gchar const**, gconstpointer*, guint32*);
```

Listing 2.4: The interface functions that need to be implemented by the specific JULEA backends, for example LMDB or RocksDB

Chapter Summary

This chapter presented the fundamental knowledge to understand the following chapter. It went through the layers of the software stack, from the application layer to file systems. In the beginning, popular I/O interfaces and the corresponding data formats were presented. As HDF5 is more commonly known than ADIOS2, the terminology was explained using the HDF5 data model. Afterwards, the motivation to develop ADIOS2 and the resulting requirements were detailed. A particular focus was on the file formats of ADIOS2, BP3, BP4 and BP5 as they form the basis for the format dissection. Section 2.2 gave an overview of database systems, their underlying data models and relational databases. A taxonomy of NoSQL storage technologies was presented, which structured the following details about several candidates, for example, key-value stores, wide-column stores and document stores. Then, local and parallel and distributed file systems were studied, comparing kernel space to user space file systems. The chapter ended with a description of JULEA and its architecture.

3. Related Work

In the previous chapters, the state of the art has been discussed with an outlook on ongoing endeavours. These are now explained in more detail. The chapter shares content with [Lüttgau et al., 2018, Duwe and Kuhn, 2021b, Duwe and Kuhn, 2021a, Kuhn and Duwe, 2020].
In the following, related approaches are examined that aim to improve one or multiple aspects of the current storage landscape to answer research question 1.1 (What optimisations exist for managing self-describing data formats and storage systems?). While some target a specific layer, there are also more general projects aiming to overhaul the entire storage system. A larger focus was put on other solutions to improve the management of self-describing data formats. Topics like the analysis of data access patterns and database research, in general, are discussed in their respective chapters. The chapter is split into the following sections:

- *Section 3.1: studies larger projects that redesign storage and file systems*
- *Section 3.2: discusses other optimisations directed at managing self-describing data formats.*

3.1. Storage and File Systems

This section discusses related work regarding the entire storage and file system. A special focus is put on the most prominent ones that likely impact future systems and on projects most closely related to this thesis. In general, the following approaches are more radical in that they sacrifice many previous efforts to advance the state of the art dramatically.

3.1.1. DAOS

The Fast Forward Storage and IO (FFSIO) project aims to provide an exascale storage system capable of dealing with the requirements of HPC applications and big data type workloads. It aims to introduce a new I/O stack and support more complex basic data types like containers and key-arrays [Lofstead et al., 2016]. Its functionality ranges from a general I/O interface at the top over an I/O forwarding and an I/O dispatcher layer to the Distributed Application Object Store (DAOS) layer, which offers a persistent storage interface and translates the object model visible to the user to the demands of the underlying infrastructure. DAOS will require large quantities of NVRAM and NVMe devices. Therefore it will not be suitable for all environments. Specifically, the high prices for these relatively new technologies will limit DAOS's use, both in data centres and especially in research, at least in the near future.

In addition to introducing a novel user-space storage system, DAOS will also support new ways of performing I/O in a scalable way [Breitenfeld et al., 2017]. From an application developer's point of view, DAOS will provide a rich I/O interface in the form of key-array objects

with support for both structured and unstructured data. Additionally, established I/O interfaces, such as a legacy POSIX interface and an HDF5 interface, will be supported natively. Similar to databases, DAOS has support for transactions. Multiple operations can be batched in a single transaction, which becomes immutable, durable, and consistent once committed as an epoch. On the one hand, this allows multiple processes to perform asynchronous write operations without having to worry about consistency problems. On the other hand, read operations will always have a consistent view because they are based on a committed (and thus immutable) epoch. First performance evaluations indicate that DAOS has a stable performance across different IO500 workloads, whereas other file systems exhibit great variation between individual tests [Liang et al., 2020].

3.1.2. ECMWF & MARS

Other storage systems focused on data archival developed by the European Centre for Medium-Range Weather Forecasts (ECMWF) are **EC**MWF's **F**ile **S**torage system (ECFS) and Meteorological Archival and Retrieval System (MARS)[Grawinkel et al., 2015]. An HPSS manages the tape archives for both systems as well as the disk cache for ECFS, where the files are accessed using a unique path. MARS is an object store providing an interface similar to a database. By using queries in a custom language, a list of relevant fields can be set, which are then joined into a package and stored in the system. The field database (FDB) stages and caches fields which are often accessed. It is both a library and a storage service working on Lustre as of now [Smart et al., 2019].

ECFS contains relatively few files which are used concurrently and experience mainly write calls. In MARS, however, the files are equally relevant and mostly read [Smart et al., 2017]. Thus, both systems provide powerful storage management for researchers interested in weather modelling. Mars allows HPC users to access vast amounts of meteorological data stored only in GRIB and BUFR formats but collected over the last 30 years. ECFS provides a huge file system which stores any kind of data that MARS can not maintain.

3.1.3. SIRIUS

The vision of SIRIUS is to design an entire software environment that builds on previous insights of efficient data management [Klasky et al., 2017]. Furthermore, the goal is to automate numerous tasks to reduce the burden on users and system administrators. An example of such optimisation could be to offer a feature to specify a maximum runtime and have the system select the data quality that can be ensured within these limits.

The authors state that despite interesting research solutions like Sirocco and Ceph, data placement optimisations still require much investigation. One of the reasons is that data access patterns vary widely, and often data is not accessed in its entirety, for example, a complete file, but in smaller subsets. This means that data placement strategies that target the file level can introduce a severe overhead. They propose to introduce a layer that internally splits the data into smaller pieces and can store them on different storage tiers [Klasky et al., 2018]. This process should be transparent to the user.

SIRIUS is, among others, funded by the U.S. Department of Energy National Nuclear Security Agency. The overarching goal is to join various efforts and sub-projects, for example, ADIOS2 and EMPRESS, into a larger system. EMPRESS possibly provides the metadata service for SIRIUS, and will be discussed in detail in the next section.

3.2. Improved Management of Self-Describing Data Formats

Next, optimisations focusing on one or more self-describing data formats are discussed.

3.2.1. EMPRESS & EMPRESS 2

The most relevant approach to improved metadata management for this thesis is *EMPRESS (Extensible Metadata PRovider for Extreme-scale Scientific Simulations)* [Lawson et al., 2022]. It offers a metadata service aimed at self-describing data formats like HDF5 and ADIOS2. The user can highlight interesting areas of data before storing it by using custom metadata tags [Lawson et al., 2017]. To achieve this, the custom metadata is stored in a relational database. The insights of EMPRESS 1 have been used to develop EMPRESS 2, which includes extensive query functionality, fault tolerance, and atomic operations [Lawson and Lofstead, 2018]. The EMPRESS2 interface will be explained after a discussion about the general approach.

Problems and Differences to this thesis While the insights of EMPRESS and EMPRESS 2 will be used to develop in the next chapters, especially for the DAI, the requirement still stands to avoid application changes at any cost. Therefore, utilising an I/O library such as ADIOS2 is convenient for making the metadata extraction transparent to the user. Also, maintaining a separate database introduces several problems, stated in detail by Zhang et al. [Zhang et al., 2019, Byna et al., 2020].

They discuss other projects using relational database management systems for metadata management like *BIMM* [Korenblum et al., 2011] and *SPOT* [Tull et al., 2013]. An approach to circumvent the required transformation of metadata is *JAMO* which uses the document database MongoDB. Zhang et al. show that they still share the same drawbacks as EMPRESS.

Their main arguments against database-based solutions are examined in the following as they closely relate to this work. First, using a database typically leads to duplication of metadata. However, in this thesis, the file system is replaced with JULEA, which allows the complete separation of the file format without storing the original file. By using JULAEA, it is possible to employ an integrated database that does not sit on top of the file layer. So, there is no duplication of file metadata. The metadata are only stored in the database, not along with a self-describing file.

Second, they further argue that the hierarchical nature of self-describing data formats cannot be modelled well in a database. However, previous experiences show that applications like Enzo use a very flat hierarchy[1]. This impression is supported by the results of the user

[1] "Enzo is a community-developed adaptive mesh refinement simulation code, designed for rich, multi-physics hydrodynamic astrophysical calculations." `https://enzo-project.org/` *accessed: 12.10.2022*

survey conducted for this thesis which is discussed in Chapter 5 and 6. In summary, users typically do not use very deep hierarchies.

Third, they state that the user needs to know details of the internal data structure in order to query it. By hiding the separation inside an HDF5 VOL plugin or an ADIOS2 engine, the user does not need to know the details of the internal data structure; neither the creation of the database indexes nor the data retrieval concern the application layer.

Fourth, Zhang et al. are concerned with the portability and mobility of database-based solutions. While the format dissection does not keep the original HDF5 or BP files, it is possible to extract all the necessary information from JULEA if the data needs to be transferred to a different research facility, not deploying JULEA. Furthermore, the database could be copied to a different system.

Zhang et al. bypass these challenges by introducing an indexing and querying service for HDF5 called *MIQS (Metadata Indexing and Querying Service)*. MIQS constitutes a schema-free indexing solution that maintains an in-memory index for each process. However, the relation between attributes and file paths needs to be stored redundantly.

EMPRESS 2 Interface In Listing 3.1 all EMPRESS 2 functions are listed that are offered for minimum, maximum and range queries. The parameters are removed from all but the first function for better readability. The complete API has 133 functions[2]. and more than 70 structs[3]. This makes the EMPRESS 2 interface very large and a bit unwieldy for potential users. The number could be reduced by adapting the parameter lists. Currently, the user has to provide, among others, the run ID, the type ID, the variable ID, and the range. The definition of the struct `md_catalog_timestep_entry` is shown in Listing D.11.

```
1 int metadata_catalog_all_timesteps_with_var_attributes_with_type_var_range(const md_server
      ↪ &server, uint64_t run_id, uint64_t type_id, uint64_t var_id, uint64_t txn_id,
      ↪ attr_data_type data_type, const std::string &data, uint32_t &count,
      ↪ std::vector<md_catalog_timestep_entry> &entries)
```

Listing 3.1: An example of an EMPRESS 2 function. It lists all timesteps that meet a condition

The general idea of EMPRESS 2 is that the user can build an additional metadata model on top of the existing one. Interesting areas can be tagged, or time steps can be added, for example. This approach essentially duplicates a large part of the efforts put into HDF5 and especially in ADIOS2. In contrast to HDF5, ADIOS2 offers time steps and additional data characteristics like the minimum and maximum for a variable and all variable blocks. Besides reimplementing existing functionality, another problem is the additional work put on the users. It is important to note that users writing scientific simulations are not necessarily computer scientists and typically do not want to be bothered by the I/O process, even more

[2] It can be found at `https://github.com/mlawsonca/empress/blob/master/include/client/my_metadata_client.h`

[3] `https://github.com/mlawsonca/empress/blob/master/include/common/my_metadata_args.h`

than today. With EMPRESS 2, they not only have to implement the I/O using a library like HDF5 or ADIOS2, but they also have to specify all information they want to tag and track explicitly. Here lies the key difference to the solution that will be developed in the following chapters. It builds upon the data model introduced by the I/O libraries and thereby hides a considerable amount of work from the user. Furthermore, by automatically pre-computing additional custom statistics like the mean value during the I/O process, the post-processing can be sped up without any changes to the application besides the engine name. The focus of this thesis is on extending the functionality offered by the libraries and adding additional query options.

3.2.2. Mapping Data to Object Stores

As mentioned before, object-based storage systems are efficient and scalable to overcome the limitations of the I/O stack. Object-centric storage systems on HSM have also been shown to improve the performance of HDF5 when its datasets are stored in the object stores [Mu et al., 2018, Mu et al., 2020]. Furthermore, by using Proactive Data Containers (PDC) as a tuning technique, the performance can be up to 47 times better than a highly optimized HDF5 implementation [Tang et al., 2019].

Another recent approach to optimize the mapping of datasets to object storage shows that pushing the I/O accesses to the object layer allows distributing it over many servers, thereby exploiting the system's parallelism[Chu et al., 2020]. Chu et al. demonstrate two possible ways of using HDF5 VOL plugins on the one hand and the SkyhookDM Ceph plugin on the other hand to reorganize the data objects.

Despite the findings of the Ceph developers, Lillaney et al. argue that dual access over file system semantics and an object store API is required to satisfy all current demands [Lillaney et al., 2019a, Lillaney et al., 2019b]. Another approach following the dual access concept is DelveFS, a user-space object store file system [Vef et al., 2020b]. It uses custom semantics to create unique views onto the object store and is aimed at large systems using even billions of objects. Strategies proposed to solve performance problems of using file systems as storage backends were implemented in SwimStore (Shadowing with Immutable Metadata Store) in Ceph [Lee et al., 2018].

NetCDF and HDF5

Using relational databases to store self-describing data formats has been discussed for a long time. For example, the developers of NetCDF stated early on that relational database systems are unsuitable for storing NetCDF data [Rew et al., 1996]. They gave various reasons. The first was that relational databases do not typically support multi-dimensional objects. This hinders intuitive data access and does not work with the abstractions used in NetCDF, e.g. coordinate systems. A second reason is the bad scalability of databases to manage large arrays efficiently. Especially indexing approaches struggle with the data sizes in large arrays. Lastly, many features that database systems offer are unnecessary for managing scientific data, such as complex update mechanisms, audit trails and elaborate transaction-support [Rew et al., 1996].

In the following, this statement is revisited and checked to determine whether it still holds and whether it can be applied to the approach developed and discussed in this thesis. One major difference is that the proposed storage of file metadata in the database is entirely different from storing all data in the database, especially in terms of size. In the proposed solution of this thesis, the data is stored in object stores, not relational databases.

While database systems have been around for a long time, they have not been widely adopted to manage scientific data. This was even noted back in 2006. At that time, Cohen et al. decided to examine again whether the extended database functionality was sufficient to manage scientific data [Cohen et al., 2006]. The extensions contained, among others, the introduction of new datatypes to represent scientific data and typical mathematical operations on them. Cohen et al. focused primarily on the area of high-performance computing and studied how the variables of the WRF (Weather Research and Forecasting) model by the National Center for Atmospheric Research (NCAR) could be managed in an object-relational database. In 2018, a case study was performed that compared NetCDF with SciDB for the management of large and multi-dimensional data. The authors acknowledge the potential benefits of a database system like transaction support and different caching strategies. However, their final evaluation showed that for large datasets, NetCDF with chunking offers the best performance [Liu et al., 2018].

In High Energy Physics experiments, data is commonly stored in a large number of small files, which introduces management overhead, as mentioned before. To reduce this overhead, files are typically merged together, which can introduce new problems for parallel access when the concatenation is not done correctly. A case study in 2022 examined how these performance problems can be mitigated [Lee et al., 2022]. The authors find that HDF5 still has restrictions with respect to collective I/O, for example, by limiting the number of datasets that can be accessed concurrently to one. They propose to change this behaviour, especially when merging small files.

3.3. Miscellaneous Storage Optimisations

In the following, various optimisations for storage systems are discussed. One common aspect is the goal to improve metadata management and lessen the performance limitations it causes.

DAMASC The *Data Management Services for Scientific Computing (DAMASC)* intend to change the file system interface by incorporating database mechanisms as the bottom layer and passing the responsibility for efficient access to the file system [Brandt et al., 2009]. The developer uses declarative queries that are evaluated and translated into efficient physical accesses to the byte stream. Therefore, the file system uses knowledge about the application and the related data formats to adapt itself and automatically install indices.DAMASC was supposed to be integrated into Ceph but has not been realised despite its promising start.

SoMeta *Scalable Object-centric Metadata Management (SoMeta)* is designed to offer the corresponding metadata infrastructure [Tang et al., 2017]. The file system metadata is managed

in a distributed hash table. Additionally, developers have the possibility to annotate this metadata with user-specific tags, such as additional information about the application.

HopsFS Other projects, such as *HopsFS*, target the previous scalability limitations of approaches, such as static namespace sharding, in terms of the number of stored objects and concurrent access to single-node services. HopsFS overcomes these constraints by introducing a distributed metadata service based on a NewSQL database for the Hadoop Distributed File System [Niazi et al., 2017]. It replaces the local in-memory metadata management resulting in a capacity improvement of at least 37 times. HopsFS-S3 is a proposal that adds POSIX-like semantics to object stores [Ismail et al., 2020]. It is built on HopsFS and updated the block storage layer to use an object store. In comparison with the EMR File System (EMRFS) [4] HopFS-S3 improves the read throughput by a factor of 3. HopsFS-S3 outperforms EMRFS in terms of metadata operations by up to 2 orders of magnitude.

Metadata Accelerator FSMAC [Chen et al., 2013, Wei et al., 2015] is a metadata accelerator that exploits byte-addressable NVM technology. Metadata is quite small, typically smaller than a file system block, which makes access to metadata inefficient. FSMAC separates metadata and data, forwarding data to the storage and metadata to the byte-addressable NVM. For synchronized I/O, this approach achieved a speed-up factor of 49.2 times and for asynchronised I/O, 7.22, respectively.

Chapter Summary

This chapter examined the current state of related work and answered research question 1.1 (What optimisations exist for managing self-describing data formats and storage systems?). It offered an overview of possible alternatives and discussed their advantages and disadvantages in the context of the goals outlined in the introduction. The project most closely related to the thesis topic is EMPRESS 2, which was developed for a similar goal. However, a different route how to achieve this was taken. The biggest difference is that the approach developed in this thesis aims to be as transparent to the application layer as possible. In contrast, EMPRESS 2 requires that users specify the metadata they need explicitly, like defining custom time steps. Aside from EMPRESS 2 projects were discussed that involve extensive changes to the current ecosystem in order to advance the storage landscape. The chapter closes with a collection of metadata management optimisations that are not format-specific nor as radical as those mentioned before.

[4] `https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-fs.html`, *accessed: 12.10.2022*

4. Format Dissection using Key-Value Stores

This chapter discusses research question 1.2 (Where is untapped potential, and how can it be used?). First, the internals of ADIOS2 are discussed in detail as they are required for understanding the following sections. Afterwards, the format dissection is introduced, and a minimal subset of ADIOS2 functionality is identified that works as the engine's basis. The resulting serialised data format and the data access granularity are explained. Furthermore, the motivation to use JULEA to provide the storage system lies in the flexibility it allows through offering storage building blocks such as different storage backends. Lastly, the implementation of the ADIOS2 engine and the HDF5 VOL plugin, as well as the effort to introduce Ceph's BlueStore as an object store backend, are described.

- *Section 4.1: gives details about ADIOS2 internals*
- *Section 4.2: presents the format dissection in general as well as design choices*
- *Section 4.3: provides the implementation for the HDF5 VOL plugin and the ADIOS2 engine that use JULEA's key-value backend.*

The thesis aims to find a better way to manage self-describing data formats. To summarise the problems presented in the introduction, the data's internal structure gets lost through the layers of the I/O stack. While the top-most layer has a clear view of the logical structure, even the MPI-I/O layer has already forgotten much information. Here, only a data type is remaining. Moreover, even lower in the stack, the information is represented as a byte stream without further metadata. Due to this loss, an HSM system cannot easily use structural information for data placement. Also, caching and similar approaches do not work well on byte streams alone.

As shown in the previous chapter, other researchers developed a wide variety of approaches to addressing these issues. However, as discussed earlier, they have drawbacks in different aspects. One major problem is introducing changes to the application layer for new features or, even worse, for keeping the current functionality. To preserve the efforts that went into developing the applications, ideally no changes are necessary. Another difficulty is data duplication when relational database systems are managed additionally besides the files. Data duplication does not only increase the space requirements but also introduces another source for data inconsistencies. Finally, application developers typically are not computer scientists and should not be bothered with additional complexity in writing their data.

Jokingly, the system's community mentions that trying to solve problems with a format inevitably leads to designing a new format. As past endeavours have shown, this approach

does not ease the situation and adds more complexity to the choice of data formats. Furthermore, there is a rich ecosystem around the formats that should ideally not be touched to avoid another hurdle for application users, developers, and their current workflows. Therefore, one priority is to avoid changing the format itself. A different way to improve the management is to make the metadata already present in the format more accessible. Thus, the answer to research question 1.2 (*Where is untapped potential, and how can it be used?*) and the proposal discussed in the following is to dissect the self-describing data format itself and couple it with the storage system.

By splitting the format, file metadata and data can be managed by different backends that cater to the specific needs. Consequently, the metadata can be accessed separately and efficiently stored on a different storage tiers and on different technologies to increase access speed further. Furthermore, structural information and user-defined attributes can be used to improve management. Information about the data model hierarchy may help determine where to place this part of the data. In this thesis, two types of backends are discussed. There are multiple ways to implement similar functionality. In this the general feasibility of the approach is demonstrated while Chapter 5 covers an improved approach using relational databases.

Where should the dissection happen? As changes to application-facing API (application programming interface) are hardly ever accepted, the approach taken in this thesis is to hide all the changes from the application layer. So, the current behaviour and functionality must be preserved throughout the development of the new approach. That leaves two general areas for changing the data handling, either inside the I/O libraries or as part of the storage or file systems. While the latter offers a greater design space, it also introduces considerable implementation overhead and possibly performance issues.

When the dissection does not happen in the I/O library, a new interface must be designed to abstract from the self-describing data formats that should be dissected. At the same time, it needs to store all relevant format information so that exporting the original format is still possible. To illustrate this aspect, a short lookahead is given. In the following, the ADIOS2 variables are examined in more detail. This will show how the data model is built directly on MPI, in contrast to HDF5, which is not. Therefore, ADIOS2 keeps metadata about local and global coordinates for the variables, whereas HDF5 stores other information, for example, about data chunking.

For an external format dissection outside the I/O library, the new interface needs to be either independent of the format in that it simply takes a metadata and a data buffer, for example, or the interface has to be tailored to the used formats. The latter will negate all benefits of not implementing the dissection in the I/O library. The underlying motivation is to offer a scalable solution that can work with other self-describing data formats with little to no integration effort. Offering a format-independent interface, however, essentially means that the API cannot be built around format specifics. So, either the library's internal logic regarding the file metadata needs to be implemented underneath the new interface to make an informed decision about the storage, or the interface does not know what is inside the buffers. This would ultimately extend the I/O stack by another layer that has no information about the data.

So, to investigate the usefulness of the dissection in general, it was implemented inside the I/O library. This way, the data access provided by the library and its formats can be used directly. By keeping the dissection transparent to the users, all the application users/developers need to change is the name of the VOL plugin or the engine, e.g. from "bp3" to "new-engine".

File Metadata Details In order to have a common ground on how the specific elements are called, a more in-depth distinction of different file metadata is needed. As can be seen in Figure 4.1, there are various kinds of file metadata for ADIOS2 too. First, there is *user-set file metadata* that the user explicitly specifies; that is, the attributes and their respective data. Second, *structural file metadata* consists of features like the variable name, data type, dimensions or namespace hierarchies. They are set by the user as well, but they belong to either a variable, group or file. Therefore, they are regarded as part of the structural file metadata. In contrast, attributes are separate objects just to define additional metadata. Furthermore, ADIOS2 also offers additional data characteristics in the form of global and local extrema. They also belong to the structural file metadata.

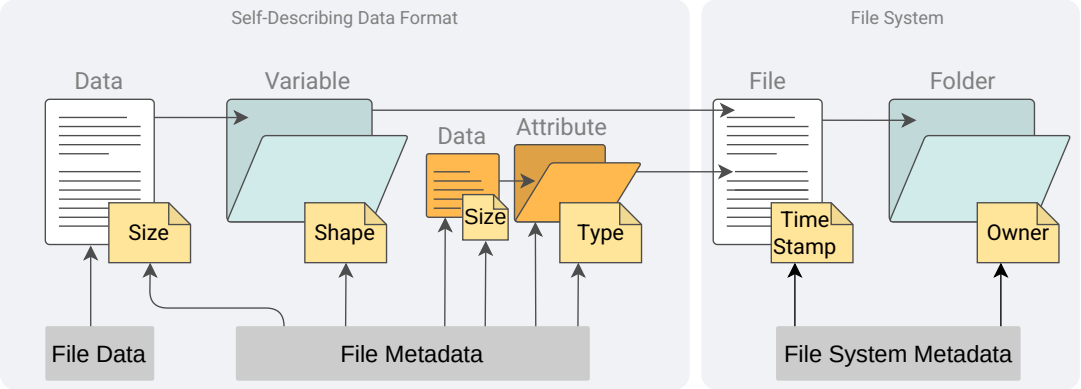


Figure 4.1.: The different metadata associated with the ADIOS2 format. Structural information is indicated by the yellow colour, while the user-set attributes are highlighted in orange. The variables are coloured in teal to differentiate them from the attributes. For completeness, the file system metadata is also added.

4.1. ADIOS2 Internals

Several constraints arise when deciding not to introduce a new format and to keep the changes transparent to the application layer. Most importantly, the previous behaviour and functionality as seen from the application need to remain the same. The first step to getting to a solution for efficiently managing self-describing data formats is to get an overview of what functionality is vital to guarantee the essential functions of the format. So, in the following, the internals of ADIOS2 are studied. The differences between global and local variables are discussed first, followed by an analysis of single values and attributes.

Global and Local Variables

In contrast, to HDF5 and NetCDF, with very detailed descriptions of the I/O libraries as well as their on-disk formats, ADIOS2 has currently rather limited documentation regarding low-level details.[1] This lack of information means a lot of trial and error in the form of reverse engineering even for BP4 where the format specifics such as the metadata index table in Section D.3 is accessible through the debugging script of Norbert Podhorszki.[2] Nonetheless, the behaviour of ADIOS2 variables is a crucial aspect of this work and will, therefore, be explained in detail in the following. Without a clear understanding of the behaviour, mimicking it is nearly impossible.

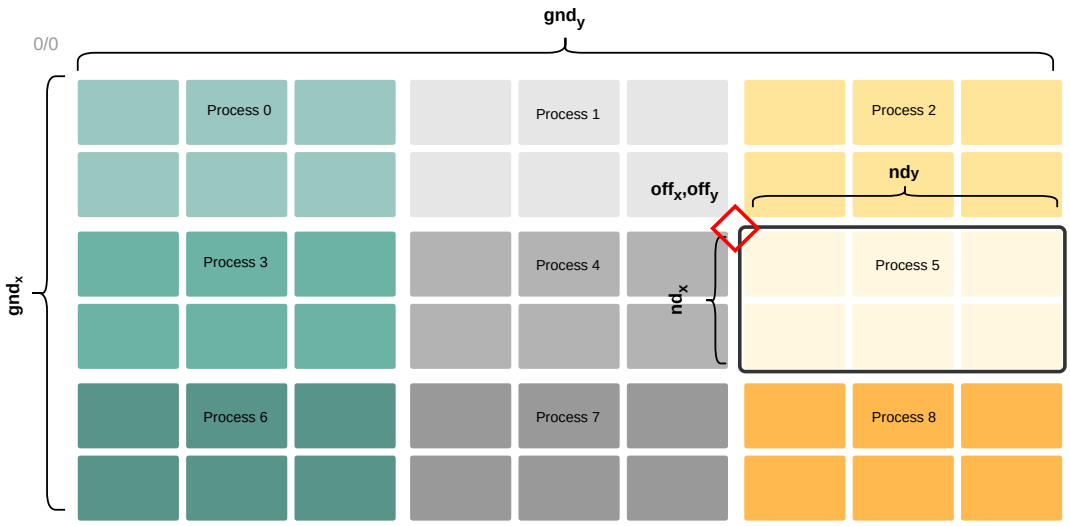


Figure 4.2.: ADIOS2 Variable: A global array is written by 9 processes (each indicated by a different colour). The global dimensions are gnd_x and gnd_y. The offset inside the global space is described by off_x and off_y, while the size the process writes is given by nd_x and nd_y. All data written in one step by one process is called a block.

MPI is fundamental In contrast to other self-describing libraries and their formats, such as HDF5 and NetCDF, ADIOS2 is designed with parallelism in mind. Rather than stacking a solution on top to enable some kind of parallelism, later on, ADIOS2 built its data model with the clear goal of targeting HPC systems and their use cases. So, MPI is deeply integrated into the library and the data model.

[1] as "hardly ever users go to that level of detail" `https://github.com/ornladios/ADIOS2/issues/2101`

[2] This experience is summarised perfectly by a user in issue 3029: "Is there any kind of reference for the BP4 file format itself? Like header structure, PGI and VMD descriptions, what part of information goes into which file (data, idx, ...)? We couldn't find any. [...] without any clear documentation of the format itself, I didn't want to bother dissecting each of the hard-coded offsets." `https://github.com/ornladios/ADIOS2/issues/3029`

ADIOS2 distinguishes global and local arrays and values, whereby local means process-local. An example of a global 2D ADIOS2 variable is given in Figure 4.2. When defining a global ADIOS2 variable, three characteristics are important: the global dimensions, the offset in the global space and the local dimensions. Their combination allows for complex data models. Note that the dimension size refers to the array size and does not mean the variable changes from 2D to 3D but rather the size of the 2D array.

The ADIOS2 developers state that the recommended and most frequent use case for a typical HPC application uses global arrays and writes these in steps. A step in ADIOS2 is an abstraction to allow the indexing of specific data aspects. Therefore, it can represent different things for different applications. It can be everything from a time step, an iteration step, an inner loop step, or an interpolation step to a variable section [ORNL, 2022].

The data one process writes at a step is called a *block*. The data of all processes for a step, that is, all blocks, are written before a new step begins. The possible data layout in storage is discussed in Section 5.2. The data of all processes is stored according to the programming language's ordering of arrays (row-major and column-major).

The curious case of process-local arrays While the dimensions of a global array have to be the same for all processes, local arrays offer much more flexibility. The dimensions do not have to be known upon definition. Furthermore, they can vary for each process, step, and even for one process over the steps. Also, a process does not need to write data for each step. Some combinations are displayed in Figure 4.3. The corresponding code for defining and writing these variables is shown in D.6. In the example, the four variables v0 to v3 are written in five output steps. However, the dimension sizes and the writing pattern change. To make this example more comprehensible, in Figure 4.3 the resulting blocks per process and step are shown.

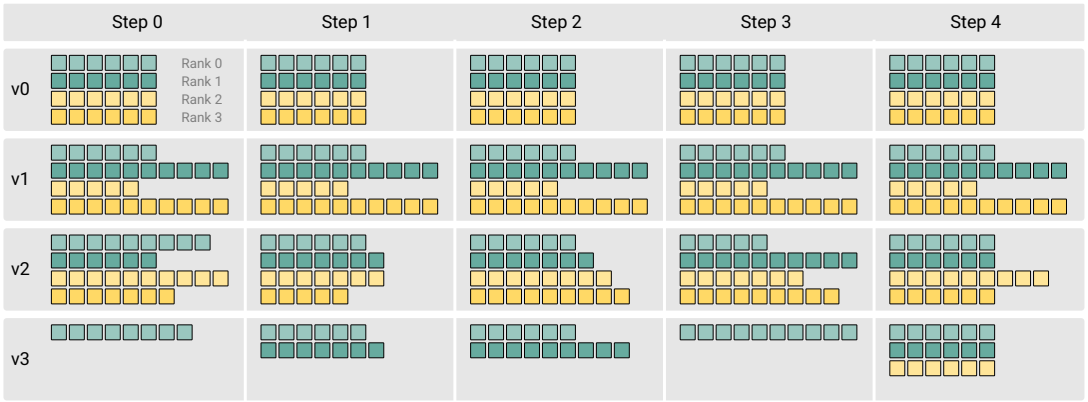


Figure 4.3.: Writing four ADIOS2 variables (v0, v1, v2, v3) for five steps with up to four MPI processes. Again, the processes are indicated using different colours. Here the local array variable type is used, which offers considerable flexibility in writing patterns.

v0 has the same size for every process at every step.
v1 has a different size for each process but the size is fixed over time.
v2 has a different size for each process and the size is also changing over time.
v3 is like v2 but also the number of processes writing the variable changes over time meaning not every process writes each step.

Naturally, one asks themselves where this type of behaviour would be needed.

This behaviour is often needed even if coming from the perspective of climate simulations considering the temperature, it is unintuitive. Neither the earth's shape is changing nor is there a scenario without a temperature. However, if you consider a particle simulation where the space is split across MPI processes, so that every process computes a specific subregion, then there might be the case that no particle is in this region. So, having the option to not need to write anything, can reduce the I/O overhead and potentially the file size. Following this direction, one can come up with scenarios for every one of these behaviour categories.

In order to support this behaviour the global and local dimensions as well as the offsets have to be stored for every step. Storing this information at the block level is not sufficient as a process may not write a block in a certain step. To account for all possible variable shapes and to make management easier, in this thesis, the metadata is split into different levels.

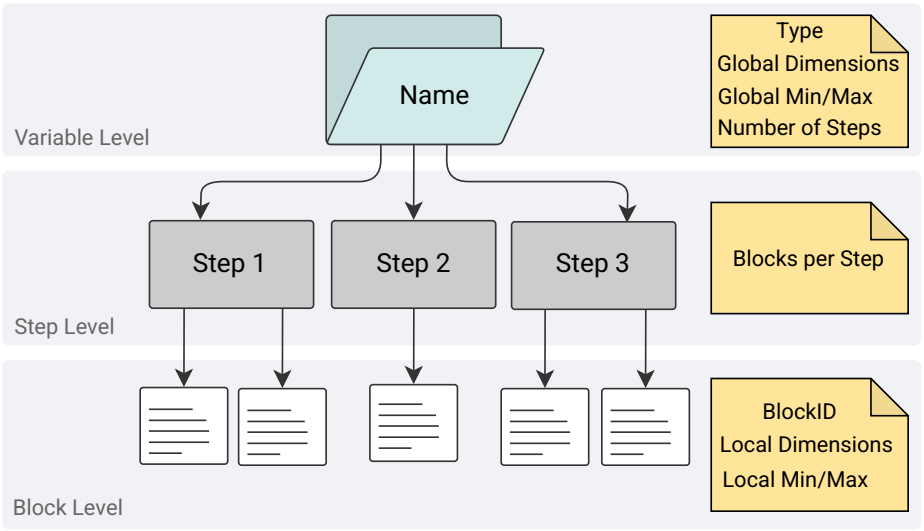


Figure 4.4.: Structural information of ADIOS2 variables on different logical levels, i.e. the variable, step and block level. Some aspects of this structural information are also referred to as data characteristics such as the global and local extrema.

Examples of the different structural metadata are given in Section 4.4. As shown, the metadata is related to various levels. Here, the focus lies on the variable, step and block level. The first includes information about the datatype and the global characteristics of the variable which is shared across all processes. At the step level, the most important detail is the number of blocks in each step. As a reminder, a block is the data of one process at a specific step. So, in contrast to the variable level, the block level holds the local characteristics.

Global and Local Single Value Variables and their Differences to Attributes

There are four shapes for variables. Besides global and local arrays, ADIOS2 also offers global and local single values. Additionally, attributes are offered as well. In the following, their intended usage and their differences are described.

Global single values store global information of one MPI that can change, for example, the number of particles. Single values are treated like metadata, so they are stored alongside the other rest of the file metadata [ORNL, 2022].

Attributes in contrast as simplified variables and are used to store additional metadata for a specific IO component. Typical use is to provide metadata for users such as information about the experiment runs. They are specified once and belong to/are associated with variables. They are time-step specific and have to be accessible in that timestep and all following steps, even if the timesteps themselves are not yet available[3].

ADIOS2 offers a lot of anticipated behaviour even though use cases are rare in production. This has both advantages and disadvantages. On the one hand, it means the API, the format and therefore applications might not need as many changes in the future. On the other hand, without a specific use case, it is hard to anticipate all requirements and a lot of semantics are not clearly defined. Most of the presented information about the distinction between attributes and global/local values is extracted in tedious hours from the issues.

For example, the difference between variables and attributes was and is still discussed and the desired semantics were unclear after version 2.5, the third production release, came out in October 2019. More specifically, when attributes should be something else than variables they should be permanently associated with a variable. However, because they are time-step specific, and belong to a variable that can have a lot of other timesteps, either the attribute needs to be stored for every timestep or the engine needs to manage the attributes outside the timestep context.

In July 2021, there was still no consensus about the semantic difference between global single-value variables and attributes. Global single values can change over time because they are still variables whereas attributes are time-step specific and part of the metadata. Also, attributes are aggregated and only written by the process with rank 0 in contrast to variables. In summary, this lack of documentation or even made decisions complicated the design and implementation from time to time. It was therefore necessary to focus on the ADIOS2 aspects that are at least clearly defined if not documented.

Groups Groups are a new concept only introduced in version 2.7.0. They are similar to HDF5 groups and can be used to structure the namespace. These groups should not be confused with the native group concept that was already present in ADIOS1, namely process groups. They were discussed in Section 2.1.2. Process groups gather different I/O routines and perform them together. So, they work as a batching mechanism.

[3] `https://github.com/ornladios/ADIOS2/issues/1912`

4.2. Format Dissection

Previously the variable behaviour was discussed with a focus on the distinct differences between global and local arrays. In order to split the format while preserving the functionality, the next step is to define a subset of functions and features that is crucial and must work. Furthermore, tools like ADIOS2's `bpls` have to produce the same output when reading bp3/bp4/bp5 files. Additionally, the original file formats must be exportable as well. A separate import/export tool has been written for HDF5 already. Another one for ADIOS2 is underway. In the meantime, the data can always be retrieved by using the native library API. For example, to export a bp4 file, all data can be read using the JULEA engine and then written to the output file with the bp4 engine. Still, to increase the convenience, a dedicated tool will be built.

4.2.1. Variable Serialisation

In ADIOS2, the variable concept is implemented using different classes. The `VariableBase` contains the information required by both compound and non-compound variables while the inheriting classes `Variable` (s. Listing 4.1) and `VariableCompound` (s. Listing D.8) specify the remaining metadata. The compound variable concept allows the definition of custom variable types. Compound variables can be helpful, for example, to store data together with indexing data. This combination is often used in particle codes, where each particle requires both float data and integer indexes[4]. However, the support for compound variables was ended[5]. Therefore, the thesis focuses on non-compound variables.

```
1     T *m_Data;  // current reference to data
2     T m_Min;    // absolute minimum
3     T m_Max;    // absolute maximum
4     T m_Value;  // current value
5
6     std::vector<Info> m_BlocksInfo;      // use for multiblock info
```

Listing 4.1: Class Members of ADIOS2 Variable

As previously stated, the structural information in ADIOS2 can be split into different levels. Most of the metadata features in the classes mentioned beforehand are at the variable level. One exception is the member `m_BlocksInfo` of the `Variable` class shown in Listing 4.1. Here the block-level details are stored like the local dimensions and extrema. A definition of the `Info` struct can be found in Appendix D.8. `T` is the ADIOS2 template type for variable types such as signed and unsigned integers of various lengths, floating point types and complex types.

[4] https://github.com/ornladios/ADIOS2/pull/1154
[5] https://github.com/ornladios/ADIOS2/pull/1225

```
1  Engine *m_Engine;
2  const std::string m_Name;    // unique identifier
3  const DataType m_Type;       // primitive from <T> or compound from struct
4  const size_t m_ElementSize;  // for compound variables
5  size_t m_BlockID;            // current block ID for local variables
6  ShapeID m_ShapeID;           // see shape types in ADIOSTypes.h
7  SelectionType m_SelectionType;
8
9  Dims m_Shape;            // total dimensions across MPI
10 Dims m_Start;            // starting point (offsets) in global shape
11 Dims m_Count;            // dimensions from m_Start in global shape
12 Dims m_MemoryStart;      // start offset
13 Dims m_MemoryCount;      // local dimensions
14
15 bool m_SingleValue;      // true: single value, false: array
16 bool m_ConstantDims;     // true: fixed m_Shape, m_Start, m_Count
17 bool m_ReadAsJoined;     // when global array written as joined array
18 bool m_ReadAsLocalValue; // when global array written as local value
19 bool m_RandomAccess;     // false: streaming
20 bool m_FirstStreamingStep;
21
22 size_t m_AvailableStepsStart;
23 size_t m_AvailableStepsCount;
24 size_t m_StepsStart;
25 size_t m_StepsCount;
26 size_t m_IndexStart;     // Index Metadata Position in serial md buffer
27 // Index to Step and blocks' characteristics position in serial md buffer
28 std::map<size_t, std::vector<size_t>> m_AvailableStepBlockIndexOffsets;
```

Listing 4.2: Selection of Class Members of VariableBase (reordered for better readability)

The most relevant class members of the variable class are discussed in the following to give an idea of how ADIOS2 realises the variable behaviour. In Listing 4.2, lines 1 to 10 manage general features like the variable type, the name or the `blockID` in case of a local variable. The global and local dimensions at the variable level can be seen in lines 9 to 13. Please note that the `Dims` type is a typedef for a C++ `std::vector<size_t>`. `m_MemoryStart` and `m_MemoryCount` are the local dimensions to contiguous memory pointers. They are required if the data contains so-called ghost cells. According to parts of the documentation, it currently only works for writing but not for reading data. Regardless, to avoid larger changes in the future, `m_MemoryStart` and `m_MemoryCount` are included in the serialised variable form.

From lines 15 to 20, different variable features are stored in the form of booleans. The features from lines 17 to 20 are only utilised for reading data. More specifically, the latter two indicate streaming access and the respective starting step. Lines 22 to 26 contain information regarding various buffers. ADIOS2 offers several functions like `SetSelection()`, `SetMemorySelection()` and `SetStepSelection()` to specify regions to write or read data from at different levels. The resulting storage regions are not necessarily contiguous and, therefore, require these additional counters, sizes and indices.

Finally, in line 28 the vital class member `m_AvailableStepBlockIndexOffsets` is listed. It is used to manage the block characteristics' position in the serial metadata buffer. The key

is a step number, i.e. a time index in bp3 format, whereas the value is a vector of block starts for that step. While this buffer is not used in JULEA, the step and block numbers have to be set correctly for the variable. This information has to be provided when initialising variables and must therefore be stored in JULEA. `bpls` for example, relies on the `m_AvailableStepBlockIndexOffsets` for the correct representation of global arrays in its output.

Serialised Metadata Format The specific information stored for the key-value store mapping can be found in Appendix D.10. This serialised format was used for the first prototype. As previously mentioned, the limitations of the self-describing data formats in terms of custom metadata can be overcome when dissecting the format. However, replacing a well-established format with a less flexible approach inside the library gains nothing. The BP format or HDF5 would be just replaced with another fixed format at a different place. So, to allow any modifications to the metadata that is stored, metadata about the metadata is required. To avoid reimplementing existing functionality, BSON was used to store the values in the key-value stores. The BSON objects function as containers, and the BSON structure holds information about its member. So, for example, it is easy to list all variables stored in the value. Without BSON, this information has to be managed by hand.

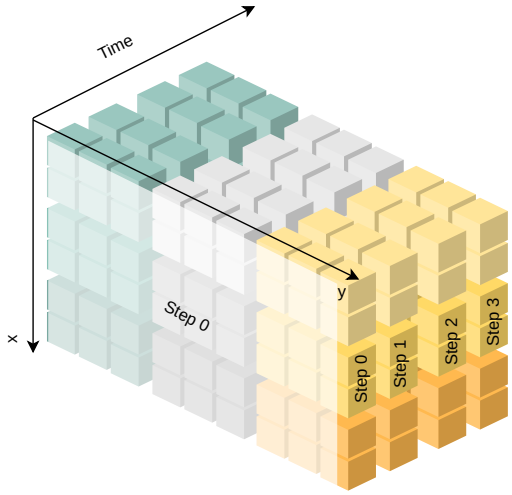


Figure 4.5.: ADIOS2 Variable Data Model used in this thesis.

Access Granularity When using key-value stores, the metadata is only available as a whole, limiting the performance when updating parts, as the entire value must be read and then written back later. This is not required when every piece of metadata is a separate key-value pair; however, this introduces overhead and a lot of string parsing to find the fitting elements in the key-value store. Managing the block metadata in 15 to 20 key-value pairs instead of one can

slow down the access significantly if no iterator is offered that can access the data at once. So, all block metadata is one key-value pair which in turn requires BSON to structure the value as mentioned previously. For consistency and simplicity, the smallest data size accessible is a block. Therefore, the metadata granularity is consistent with the data granularity. The data chunks are identified by a unique ID which is assigned to this specific combination of the file name, variable name, step and block. This naming scheme allows fine-grained access to the data. Data identification through the ID also means that migrating the database or the object store is uncomplicated as it does not require path updates. The data access will be discussed in more detail in the next chapter in Section 5.2.

To ease the discussion in the following sections and chapters, global arrays will be the focus. This means that all processes write the same data size at every step if nothing else is mentioned. The resulting data model as shown in Figure 4.5.

4.2.2. Using JULEA

It is a significant effort to design and implement a novel storage solution, even more so when it needs to run distributed over an HPC cluster. Most solutions like EMPRESS that address similar goals as in this thesis had to implement their own client/server architectures. That includes tasks such as serialising messages, setting up the network communication and other features that can quickly introduce new bottlenecks when done hastily. However, implementing such functionality does not necessarily contribute to new knowledge, even though the implementation can pose technical challenges. In order to reduce the reimplementation effort, a different path was chosen in this thesis.

As described in Section 2.4, JULEA is a storage framework that offers flexible building blocks to design and set up a custom storage system. Even though JULEA did not offer everything that was required for this thesis, it allowed reducing the implementation overhead considerably. Not all communication with key-value stores, object stores and databases had to be set up from scratch. Furthermore, it already offered benchmark functionality, a considerable number of tests and continuous integration to catch regressions with specific compilers and environments early on. The most important extension for JULEA was the client and the backend for structured metadata. These are the database client and backend discussed in Chapter 5. As JULEA is a user-space system the setup can be done without root access. Application changes can be limited to changing the engine or the VOL plugin's name. Currently, ADIOS2 has to be built with JULEA support. However, this problem can be solved through the *plugin engine* concept of ADIOS2 that was recently introduced. It works very similarly to the HDF5 VOL plugins. So, the library code does not have to be changed. Instead the external plugin can be loaded.

Architecture

The format dissection using JULEA at the node level is illustrated in Figure 4.6. The parallel application communicates across the different compute nodes through MPI. The application uses an I/O library such as ADIOS2 or HDF5 for writing and reading its data. The format is then dissected inside the library and directed forward to the JULEA clients by either the

engine or the VOL plugin. Depending on the specific backend, the following steps differ. Some backends directly communicate with the key-value store or the database. At the same time, others and also the object stores talk to the JULEA server, which in turn talks to the appropriate storage technology. The network communication happens using TCP. Support for libfabric and therefore Infiniband has been added.

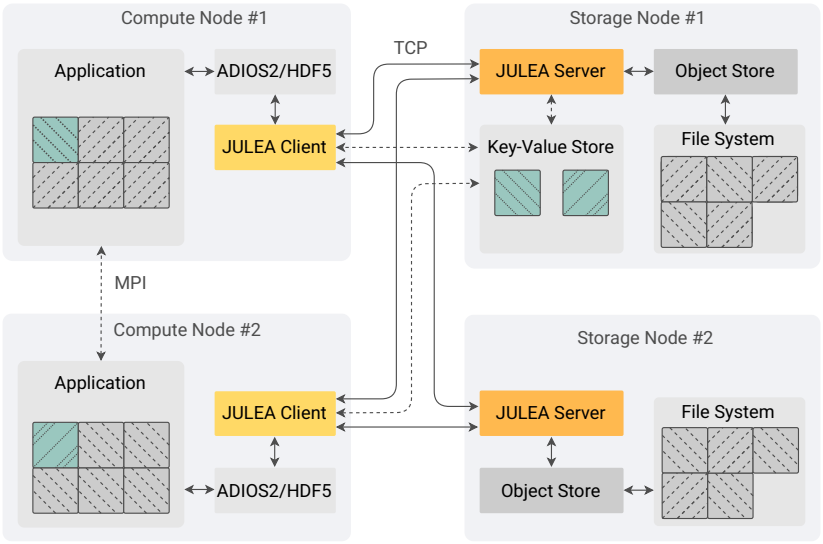


Figure 4.6.: JULEA setup with two compute nodes and two storage nodes. The JULEA components are coloured yellow (clients) and orange (server). The separation of the file metadata and file data over the different JULEA backends is illustrated using different squares. The key-value store manages the small file metadata pieces (green with fine-grained pattern), while the object store backend handles the large data chunks (grey with coarse-grained pattern). It also distributes the chunks over several storage nodes.

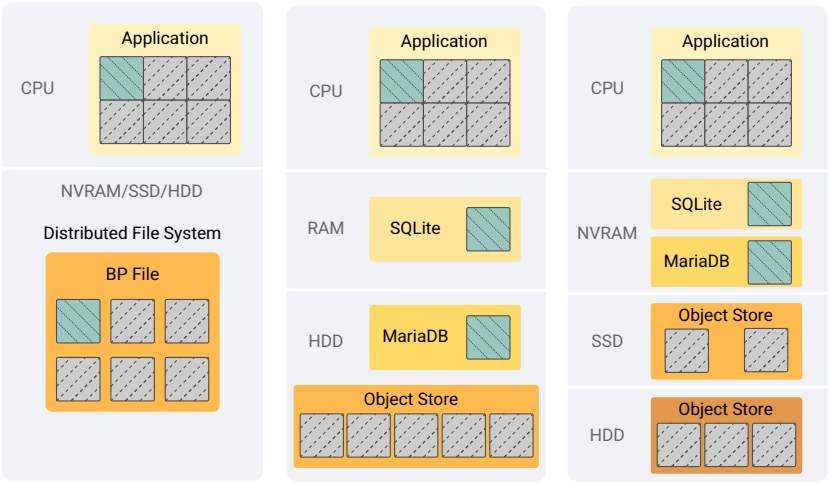
At this time, the file metadata is managed by only one key-value store to allow for an easier prototype development.

Data Distribution and HSM Support

Figure 4.6 also shows how the application output is split into file metadata and data and then stored separately. Furthermore, the data is striped across the nodes. JULEA also offers the option to distribute single objects when they reach a set size to balance the load on the system. In the evaluation, the following backends are examined: LevelDB, RocksDB, LMDB and SQLite.

The separation of file metadata and data allows using JULEA's versatile configuration features, as depicted in Figure 4.7. So, the metadata backend, that is, the key-value store or the database, can be stored on faster hardware than the object store without the requirement of large amounts of fast hardware. This is possible because the database does not hold the data and therefore is small compared to the original file size. The different backends can be put on different hardware independently already which is the foundation for an HSM system and

storage tiering. In the future, JULEA's support for storage hierarchies will be extended further to use new technologies such as NVRAM fully. This will also allow storing parts of the object onto SSDs for fast access, while the majority can be kept on HDDs (right).



(a) ADIOS2 **(b)** Current JULEA **(c)** Future JULEA

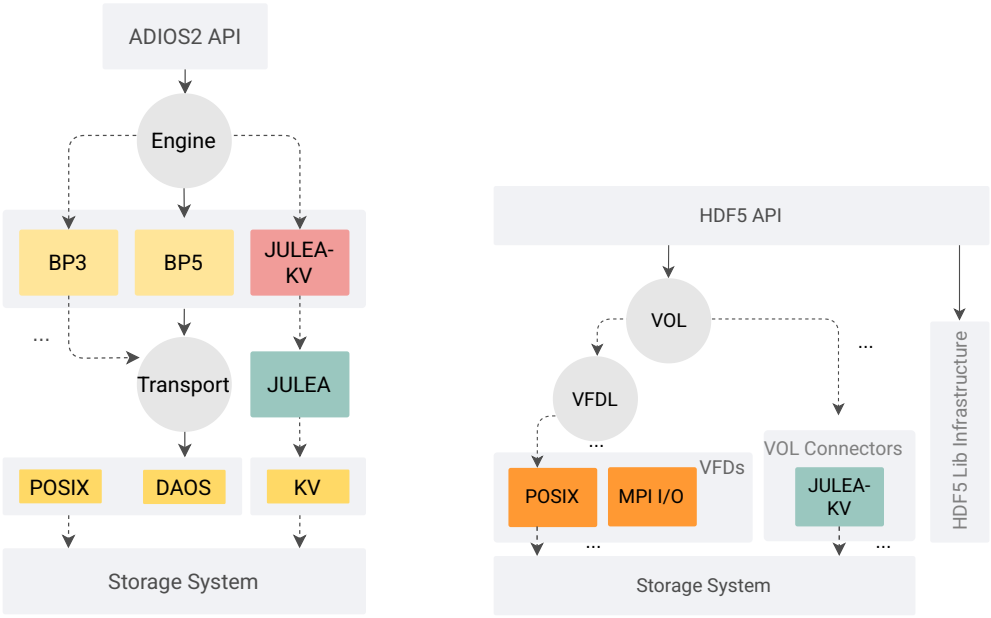
Figure 4.7.: HSM support in parallel distributed file systems compared to the current and future HSM support in JULEA. In parallel distributed file systems, the file metadata (green with a fine-grained pattern) is stored as part of the file. There are different approaches to spreading data across different storage tiers. However, when dissecting the file format, the metadata is handled by the key-value or database backends in JULEA that can be configured to run on faster hardware in comparison to the object store that manages the data (grey with coarse-grained pattern). In the future, the object store can be distributed over different storage tiers to store important and frequently accessed data on faster but smaller hardware.

4.3. Implementation

As previously discussed, the format dissection is implemented inside the I/O library. Figure 4.8 serves as a reminder of the libraries' mechanisms to change the I/O functionality and redirect the calls to JULEA. For both ADIOS2 and HDF5, solutions using key-value stores and relational databases have been developed. As the focus of this thesis will lie on the respective engines, the HDF5 VOL plugins are only discussed very briefly. Note there is no additional transport implementation for the JULEA engines. The respective functionality is implemented in the engines as well to reduce the complexity and number of leverage points.

4.3.1. ADIOS2 - Key-Value Store Engine

One metric to consider for the evaluation of the dissection approach is the implementation effort for a specific library. ADIOS2 is written in C++, HDF5 in C. The code to enable just



(a) JULEA-KV Engine **(b)** JULEA-KV VOL Plugin

Figure 4.8.: The mechanisms of ADIOS2 and HDF5 to allow different I/O handling in the form of engines or VOL connectors

writing and reading global arrays already has about 5000 lines of code for the JULEA-KV engine alone. This is in part caused by the template usage to support numerous data types. In the following, the most important functions that need to be implemented are discussed.

To ease the understanding, a small summary of the library components and their respective responsibilities is given first. The application interacts with the ADIOS library using the `adios::ADIOS` handle. This handle works as a factory object and is the only part that is owned by the user. It allows declaring `IO` objects and defining operators. The `IO` in turn is responsible for the overall management of the I/O process, and stores the used engines and their parameters as well as a map of defined variables and attributes. The actual interaction with the file like opening, writing and reading is performed by the engines.

Data Type Templates As briefly mentioned in the previous section, the different variable types in ADIOS2 are currently implemented using the template functionality of C++. While this is convenient when writing C++ code, it comes with some overhead in terms of conversion and type parsing because the JULEA interface is written in C. When serialising the data, the templates have to be resolved. Values such as the extrema are type-specific whereas the structural metadata remains the same for all variable types. As a first step, two types are supported, i.e. integers and doubles. Different types such as shorts or various other signed and unsigned integers can be mapped to the same type as they have the same or smaller size. The specific type is stored in JULEA, so that on data access the original ADIOS2 type can be restored.

In the following, the skeleton engine is presented to show which public and private functions need to be implemented for every engine regardless of the format. Alongside, relevant details for the JULEA engine implementations are given as well. The respective writer header is shown in Listing 4.3.

```
class SkeletonWriter : public Engine {
public:
  SkeletonWriter(IO &adios, const std::string &name, const Mode mode,
                 helper::Comm comm);
  ~SkeletonWriter() = default;
  StepStatus BeginStep(StepMode mode, const float timeoutSeconds = -1.0) final;
  void EndStep() final;
  size_t CurrentStep() const final;
  void PerformPuts() final;
  void Flush(const int transportIndex = -1) final;

private:
  int m_Verbosity = 0;
  int m_WriterRank;                // rank in the writers' comm
  int m_CurrentStep = -1;          // starts at 0
  bool m_NeedPerformPuts = false;  // whether EndStep must call PerformPuts

  void Init() final;
  void InitParameters() final;
  void InitTransports() final;

#define declare_type(T)                                                    \
  void DoPutSync(Variable<T> &, const T *) final;                          \
  void DoPutDeferred(Variable<T> &, const T *) final;
  ADIOS2_FOREACH_STDTYPE_1ARG(declare_type)
#undef declare_type

  void DoClose(const int transportIndex = -1) final;
  template <class T>
  void PutSyncCommon(Variable<T> &variable, const typename Variable<T>::Info
      ↪ &blockInfo);
  template <class T>
  void PutDeferredCommon(Variable<T> &variable, const T *values);
};
```

Listing 4.3: Skeleton Engine Write Header

Step functionality The step functionality provided by ADISO2 is very convenient when dissecting a self-describing data format. In contrast to HDF5, a step of any kind can be directly represented using the interface. There is no need for an extra dimension that stores time information which makes it easier to handle the parallelisation introduced by the inherent MPI support. Nonetheless, there are still a lot of cases to consider when implementing local arrays. Therefore, focusing on the global arrays because they are by far the most common variable type reduces the complexity considerably. As one process will only write one block

per step, the `blockID` equals the respective rank. In combination with the file name, variable name and the specific step the block can be uniquely addressed. The step concept is mostly realised through `BeginStep()` and `EndStep()` (Lines 7 to 8). In `EndStep()`, `PerformPuts()` (Line 9) is called when variables have been written asynchronously in this step. Furthermore, the data is flushed with `Flush()` (Line 10) when `m_FlushStepscount` was set accordingly. Lastly, additional management is performed such as increasing the step counter and resetting the `BlockID`.

Writing In the high-level, i.e. user-facing, API ADIOS2 offers `Put()` to write variables. This function is overloaded to offer greater flexibility. Internally `Put()` calls either `DoPutSync()` or `DoPutDeferred()` (lines 23 to 24). Both are implemented in the actual engine, that is, e.g. BP4 or the JULEA KV engine. While the specific behaviour of these functions differs, most will store the variable metadata and either add the variable to a buffer of deferred variables or write it. In the case of asynchronous I/O, the data cannot be reused until `EndStep()`, `Close()` or `PerformPuts()` is called. The synchronous write guarantees, that the data can be reused directly after the call.

The macro from lines 22 to 26 allows to define the functions for all variable types without explicitly stating everyone of them. `DoPutDeferred()` and `DoPutSync()` in turn call `PutDeferredCommon()` or `PutSyncCommon` respectively. Then, inside `PutDeferredCommon()`, `PerformPuts()` is called for every variable in `m_DeferredVariables`. `PerformPutCommon` is then called from `PerformPuts()`. In contrast to variables, attributes are only written when the engine is closed.

Regarding the implementation of parallelism, ADIOS2 is more convenient than HDF5 as MPI is a part of the core design of the format. For global arrays, all processes write exactly one block per step and this block can be uniquely identified using the process rank as the blockID in the specific step. This dramatically reduces the synchronisation overhead compared to the VOL plugin [Perevalova, 2017].

Note that there is no delete functionality. Also, writes do not update data. Either data is appended or the file is completely overridden. Deleting data is regarded to be the file system's responsibility. From the perspective of a developer used to providing general-purpose systems, this is unusual at first. However, most large-scale simulations in the HPC community tend to have WORM workloads, that is write-once read-many. Therefore, it makes sense to focus on supporting this very common case. Implementing a custom engine is therefore easier because updates and thereby inconsistencies cannot happen.

Reading The functions to read variables are mostly built analogue to the write functionality, e.g. the equivalent to `PerformPuts()` is `PerformGets()`. So, the high-level API offers an overloaded `Get()` which then calls `DoGetSync()` and `DoGetDeferred()`. Internally `GetSyncCommon()` and `GetDeferredCommon()` are called. However, one difference is the function `AllstepsBlocksInfo()` that retrieves all available blocks for a specific variable. As the memory scales with the size of the metadata, this can be an expensive operation.

4.3.2. HDF5 - Key-Value Store Plugin

Parts of this section have been published in [Kuhn and Duwe, 2020], while other parts are to be submitted in [Kuhn and Duwe, 2023]. In the following, it is shown how the format dissection can be performed for another library and format, not just ADIOS2 and its BP formats. HDF5 supports various objects as explained in the Section 2.1, The focus of the implementation lies on files, groups, datasets and attributes. Therefore, the most important functionality is covered without introducing immense complexity by providing every feature possible. The initial prototype of the JULEA-kV VOL plugin was implemented in 2017 by Perelova [Perevalova, 2017]. In Figure 4.9a and 4.9b all required fields for files and groups are shown.

Note the assumption that each dataset belongs to only one group. Thereby, the implementation of links can be avoided which again reduces the complexity.

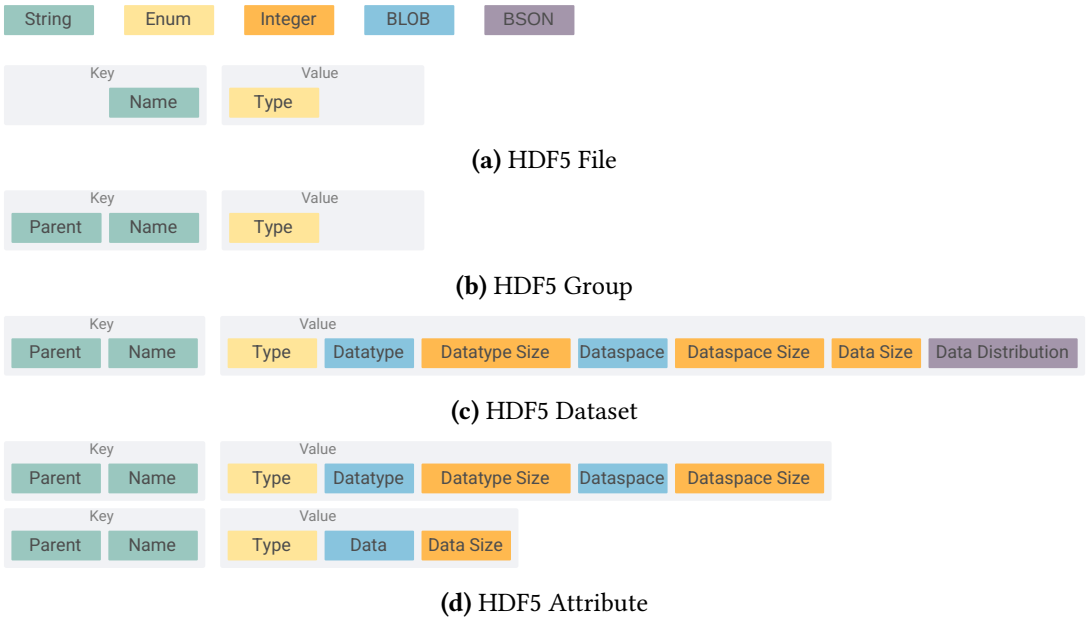
Files Files are the foundation within HDF5, storing all the other data structures. There are no higher-level data structures, so no further metadata is required as seen in Figure 4.9a. They are accessed via a *name* that traditionally refers to a POSIX path. The file name can be an arbitrary string. It is used as the key for a key-value pair in the JULEA backend.

Groups As previously mentioned, groups are used to structure an HDF5 file's namespace. Groups can contain datasets or further groups and it is, therefore, necessary to store parent/child relationships. They are accessed via a *name* and their parent group to allow groups with different parent groups to have the same name without causing conflicts. In Figure 4.9b an overview is provided of all required fields to represent an HDF5 group. The group's name as well as information about its parent is encoded within the key-value pair's key. Concretely, the key contains the full path to the group, such as `file/group1/group2`. The impact of the hierarchical structuring of the namespace on the performance will be discussed in the next chapter when relational databases are used in JULEA.

Datasets Another important data structure are datasets as they contain the actual data. Datasets are the data structure containing the actual data. Like all other data structures, datasets are accessed via a *name*. Additionally, parent/child relationship information makes it clear which group a dataset belongs to.

A complete overview of the required fields is shown in Figure 4.9c. Datasets are stored in a hybrid format: While most information is stored in a key-value pair, the actual data is stored as an object distributed across the storage system's data servers. Moreover, datasets consist of multi-dimensional arrays of a given *data type*. Therefore, information about the underlying data type has to be stored. A dataset's dimensionality is managed within the *data space*, which also has to be stored. Finally, the data size and information about the data distribution settings are stored in the key-value pair. As mentioned previously, the actual data is stored as an object distributed across multiple data servers.

Attributes The final data structure supported by the current implementation are HDF5's attributes, which allow storing arbitrary metadata attached to files, groups or datasets. Just like all other data structures, attributes are accessed via a *name* and their parent data structure.



(a) HDF5 File

(b) HDF5 Group

(c) HDF5 Dataset

(d) HDF5 Attribute

Figure 4.9.: Overview of the HDF5 metadata stored in the key-value store backend in JULEA for the different HDF5 objects. The attribute metadata is stored in two key-value pairs to allow updates to the data without changing the rest of the metadata.

All fields stored for attributes are illustrated in Figure 4.9d. Just like datasets, attributes are made up of multi-dimensional arrays of a given *data type*. Additionally, the *data space* contains information about the attribute's dimensionality. In contrast to files and groups, attributes are stored as two key-value pairs: The actual data is stored within a second key-value pair to allow updates to an attribute's content without having to modify the key-value pair containing the information about its data type and data space. This will not be necessary in the future when the attributes are stored in the database backend.

4.3.3. BlueStore Backend

Parts of this section have been published [Duwe and Kuhn, 2021b]. As the data is managed by JULEA's object store, it is time to look at the available options. Currently, the object store is a prototype built to store the data directly in POSIX. Objects are mapped to files similar to Ceph's FileStore. This approach brings along the expected limitations discussed in the introduction. Therefore, other options were investigated.

There are hundreds of key-value stores and NoSQL databases.[6] Furthermore, there exist several object store solutions, especially for cloud storage, like Amazon S3 and Google Cloud Storage or open-source variants such as OpenStack Swift[7] and MinIO[8].

[6] The list of NoSQL database management systems (DBMS) `https://hostingdata.co.uk/nosql-database/` currently contains 225 DBMS.
[7] `https://opendev.org/openstack`
[8] `https://github.com/minio/minio`

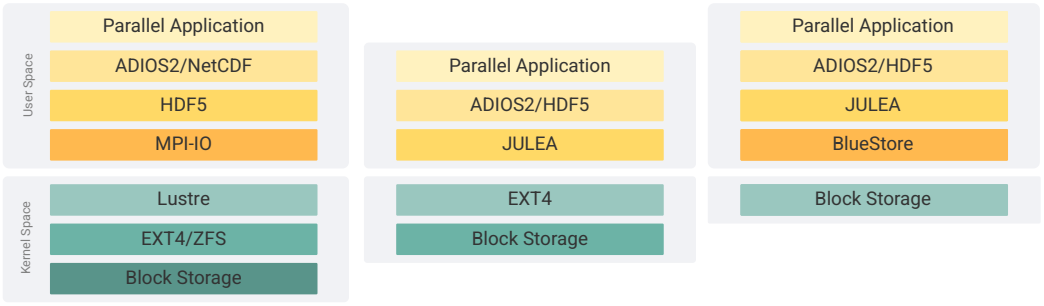


Figure 4.10.: I/O stack comparison: Legacy (left), JULEA (middle), JULEA with BlueStore (right)

Another prominent example is the Reliable Autonomic Distributed Object Store (RADOS), which is at the core of Ceph and provides three services. These are object storage similar to Amazon S3 through the RADOS Gateway, a virtual block device through the RADOS Block Device, and the distributed file system CephFS built on POSIX. However, the Ceph developers turned away from building storage backends on local file systems, as they deem them unfit for distributed storage backends [Aghayev et al., 2020]. Instead, they designed a new object store called BlueStore that works directly on raw storage devices and runs in user-space [Aghayev et al., 2019b]. This decision gives more control over the I/O stack and allows them to decrease performance variability.

Integration into JULEA As this is a very interesting concept, BlueStore was chosen for the development of another object store prototype. The first objective was, to see whether it would be possible to use BlueStore without running a full-fledged Ceph Cluster. Figure 4.10(b) and Figure 4.10(c) depict the I/O stack using JULEA and using JULEA with BlueStore. As can be seen, the stack has become simpler by moving more layers into the user-space. By using BlueStore, only the block storage remains in kernel-space.

In Figure 4.11, the layers of the JULEA BlueStore backend are shown in comparison to the Ceph backends.

To ease the future usage of BlueStore independently of Ceph, a small library was developed providing the general functionality. The following calls are supported at the moment: `init`, `mkfs`, `mount`, `create collection`, `umount`, `create`, `delete`, `write`, `read`, `status` The JULEA backend for BlueStore is in essence a thin layer wrapping the library functions. Thus, the object store client uses the BlueStore backend which in turn uses the BlueStore Library.

At the moment to use BlueStore with JULEA, the user needs to be able to either install the Ceph dependencies themselves or convince the admin of their cluster to provide them. Then, Ceph needs to be compiled to obtain the necessary BlueStore libraries.

The performance evaluation is presented in Section 7.1.2. BlueStore was run on a loop device as block storage to avoid having to repartition the existing storage devices. Even though the current state is still away from the final goal to extract BlueStore, it offers a considerable simplification compared to the only previous option of running a full-fledged Ceph cluster.

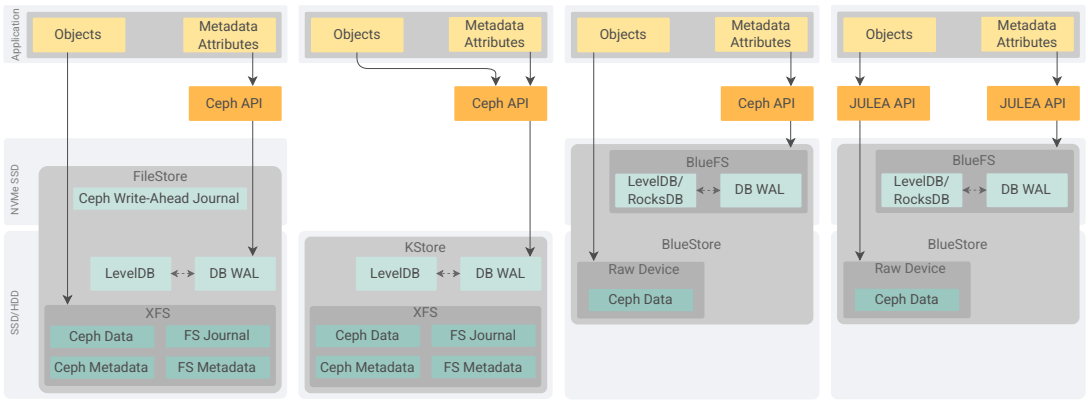


Figure 4.11.: Architecture of the Ceph storage backends FileStore, KStore, BlueStore and BlueStore using JULEA. The figure extends the work by Lee et al.[Lee et al., 2017]

In summary, the prototype unfortunately was not able to outperform JULEA's POSIX object store. Therefore, further tuning is required to make BlueStore a viable backend for JULEA.

Summary

This chapter answered research question 1.2 (Where is untapped potential, and how can it be used?). By dissecting the file format, the file metadata can be managed by suitable backends in JULEA, for example, key-value stores. This improves access to the information already provided by the format. Furthermore, the dissection enables the addition of custom metadata which will be discussed in Chapter 5 and 6. The ADIOS2 variables were examined in detail as they lay the foundation for the following chapters. Afterwards, the options are discussed on how the variable can be serialised. In order to focus on the format dissection, the JULEA storage framework is used to store the metadata and data as it offers numerous backends, which reduces the implementation efforts considerably. Both an ADIOS2 engine and an HDF5 VOL plugin were discussed to show that the format dissection works for different formats. The chapter ends with a short digression into the object storage backends of JULEA because they will, for the most part, determine the potential performance of writing data. Currently, JULEA only offers an object store where the objects are mapped to files in the underlying file system. Therefore BlueStore was studied and whether it can be run on a cluster without having to deploy a Ceph cluster.

5. Using Relational Databases for Improved Data Access

In this chapter, the format dissection using relational and NoSQL databases are discussed. As JULEA does not support relational databases, a new JULEA client and backend must be developed. These are then used to store the file metadata of HDF5. Popular candidates are examined to see which database technologies may be beneficial for the format dissection. Then, typical access patterns are studied, and the results from the first part of the user survey are shown. Combining the gathered insights enables choosing a suitable database candidate. Finally, the design and implementation of the ADIOS2 database engine are presented. The chapter is split into the following sections:

- *Section 5.1 focuses on advancing format dissection, building database support in JULEA and investigating database technology*
- *Section 5.2: contains the data access analysis of common data layout and the first part of the user survey on format usage*
- *Section 5.3: describes the design and implementation of the database engine.*

In the following, research question 2 and all its subquestions are covered. The focus lies on continuing the remaining aspects when answering research question 2.1. Afterwards, research questions 2.2 and 2.3 are discussed as well. However, the examination is kept at the general concept level and thus is not finished in this chapter. The specific custom metadata and the actual HSM support are defined and explained in Chapter 6.

- 2.1: How can metadata and data be queried efficiently?
- 2.2: What custom metadata is interesting?
- 2.3: How can the format dissection benefit HSM?

The previous chapter showed the possibility of splitting the self-describing data formats and managing them in appropriate backends. Therefore, file metadata can now be accessed separately from the data. However, storing the file metadata in key-value stores has drawbacks. The largest is the access granularity. The value can only be accessed in its entirety which hinders fine-grained access. In order to better use the potential of self-describing data formats, support for a query interface is required. So, the design and implementation of the new relational database JULEA client and backend are presented.

To answer research question 2.1, the following steps are taken in the next sections. First, possible alternatives to key-value stores like relational databases or data warehouses are studied. Their different feature sets are examined. It is vital to understand the data access patterns

to decide which option is the most suitable. Therefore, afterwards, an analysis is performed to study the access patterns that can be expected. This knowledge is then used to choose an option. The choice is motivated and weighed against the other candidates. Here, research question 2.2 is kept in mind. To allow the user to define custom metadata, the respective metadata backend needs to offer a certain degree of flexibility. Furthermore, it also has to provide an efficient interface to query additional metadata as well. Finally, it is decided what the concrete mapping could and should look like.

Next, the focus lies on research question 2.2 and the freedom that comes with giving up the format limitations. As part of the previous question, the possibility of offering custom metadata, in general, was discussed. Now the question lies in what types of additional metadata are interesting as the format limitations are no longer imposed when the format is dissected. Here, general categories and specific examples are given. The implementation of supporting custom metadata will be described as part of the DAI in the next chapter.

5.1. Advancing the Format Dissection

After using a key-value store as a backend to store the file metadata, an apparent next candidate to allow more complex querying is relational databases. The design and implementation of a new JULEA client and backend using relational databases is discussed first. As the design gradually evolved, custom metadata was not the focus at this point. Instead, finding a solution for fine-grained access was the primary concern. Thus, the strict constraints from the relational database schema definition are no problem. A relational schema works fine when only the original file format is mapped to a database.

5.1.1. Database Client and Backend for JULEA

As previously mentioned, JULEA did not have clients or backends for structured metadata based on relational databases. Therefore, the client and backend were designed and implemented to use SQL databases. Currently, MySQL/MariaDB and SQLite are supported as JULEA backends. In the following, the different objects and their most essential functions are detailed. The focus is put on the client functionality as this is what is used by the database engine. Functions like `ref` and `unref` that are defined for all objects are not shown. Warnke realised the implementation [Warnke, 2019].

Client

In order to support a wide variety of scenarios, the interface definition for the client is kept as limited and general as possible. Detailed functionality quickly becomes very specific and is therefore not applicable to every application. A set of `JDBTypes` data types was defined to abstract from the actual database technology. It is shown in Listing 5.1.

```
enum JDBType {
  J_DB_TYPE_SINT32,
  J_DB_TYPE_UINT32,
  J_DB_TYPE_FLOAT32,
  J_DB_TYPE_SINT64,
  J_DB_TYPE_UINT64,
  J_DB_TYPE_FLOAT64,
  J_DB_TYPE_STRING,
  J_DB_TYPE_BLOB,
  J_DB_TYPE_ID};
```

Listing 5.1: JDBTypes: The supported data types for the JULEA database clients and backends

Schema An undeniable functionality required for a relational database implementation is the option to define a schema. The `JDBSchema` requires the specification of a namespace as well as a schema name upon creation as shown in Listing 5.2 (s. Lines 1-2). The schema definition is done by adding the respective fields, i.e. columns. Besides the schema, the field name and the datatype are necessary (s. Line 5). There is no implicit type conversion in JULEA DB. So, the user has to use the same type for writing and reading data. This is no problem, as the interface is not used directly by the application developers but inside the I/O library. Therefore, the schema definition, as well as the used types, are encapsulated and kept transparent to the application layer. The schema can be retrieved via `j_db_schema_get` (s. Line 6). The specific field can then either be queried one at a time (s. Line 7) or using the iterator returned by `j_db_schema_get_all_fields` (s. Line 8). To enable faster access, indices can be defined and built.

```
JDBSchema* j_db_schema_new(gchar const* namespace_, gchar const* name, GError** error);
gboolean j_db_schema_create(JDBSchema* schema, JBatch* b, GError** e);
gboolean j_db_schema_delete(JDBSchema* schema, JBatch* batch, GError** error);

gboolean j_db_schema_add_field(JDBSchema* s, gchar const* name, JDBType t, GError** e);
gboolean j_db_schema_get(JDBSchema* schema, JBatch* batch, GError** error);
gboolean j_db_schema_get_field(JDBSchema* s, gchar const* name, JDBType* t, GError** e);
guint32 j_db_schema_get_all_fields(JDBSchema* s, gchar*** names, JDBType** t, GError** e);

gboolean j_db_schema_add_index(JDBSchema* s, gchar const** names, GError** e);
gboolean j_db_schema_equals(JDBSchema* s1, JDBSchema* s2, gboolean* equal, GError** e);
```

Listing 5.2: JDBSchema: The schema-related functions offered by the new database client.

Selector The selector is used to update, query and delete entry objects from the table. Creating a new selector requires the schema on which to work as well as a mode. This mode defines how the fields later added to the selector are combined in a query. The mode can be either *and* or *or*. More complex queries can be realised by adding sub-selectors as shown in Listing 5.3(Line 4).

```
JDBSelector* j_db_selector_new(JDBSchema* s, JDBSelectorMode m, GError** e);
gboolean j_db_selector_add_field(JDBSelector* s, gchar const* name, JDBSelectorOperator o,
  gconstpointer value, guint64 length, GError** e);
gboolean j_db_selector_add_selector(JDBSelector* s, JDBSelector* sub_s, GError** e);
```

Listing 5.3: JDBSelector: The selector-related functions offered by the new database client.

Entry The `JDBEntry` directly corresponds to the entry in an SQL database. The functions are shown in Listing 5.4. To create an entry, the schema has to be passed to allow data type inference. The database auto-generated the ID, as mentioned in the DB VOL plugin in Section 5.1.2. It can be used to identify an entry in the table uniquely. Specific values can be added using `j_db_entry_set_field`. When not all fields are specified, the other fields are set to NULL on inserts. For updates, no changes are made to fields that are not specified.

```
JDBEntry* j_db_entry_new(JDBSchema* schema, GError** e);
gboolean j_db_entry_get_id(JDBEntry* e, gpointer* value, guint64* length, GError** e);
gboolean j_db_entry_set_field(JDBEntry* e, gchar const* name, gconstpointer value, guint64
    ↪ length, GError** e);
gboolean j_db_entry_insert(JDBEntry* e, JBatch* b, GError** e);
gboolean j_db_entry_update(JDBEntry* e, JDBSelector* s, JBatch* b, GError** e);
gboolean j_db_entry_delete(JDBEntry* e, JDBSelector* s, JBatch* b, GError** e);
```

Listing 5.4: JDBEntry: The entry-related functions offered by the new database client.

Iterator As the name suggests, the iterator is used to iterate over database entries. The functions are shown in Listing 5.5. The entries are transferred in bulk to reduce the network traffic. On creation, the iterator takes schema and selector and retrieves matching entries. The actual metadata is then retrieved via `j_db_iterator_get_field`.

```
JDBIterator* j_db_iterator_new(JDBSchema* s, JDBSelector* s, GError** e);
gboolean j_db_iterator_next(JDBIterator* i, GError** e);
gboolean j_db_iterator_get_field(JDBIterator* i, gchar const* name, JDBType* t, gpointer*
    ↪ value, guint64* length, GError** e);
```

Listing 5.5: JDBIterator: The iterator-related functions offered by the new database client.

Backend

The backend definition is kept short to make implementing new backends as easy as possible. The I/O access is kept to `insert`, `update`, `query` and `delete`. The complete interface definition can be found in Appendix 2.3. While the definition is aimed at SQL implementations, it does not require one [Warnke, 2019]. To allow supporting different SQL backends, a generic SQL wrapper is employed. Thus, the actual backend implementation can be exchanged easily. It also reduces code duplication as common functionality is implemented in the generic

wrapper. Internally BSON (binary JSON) is used for the network communication as it was already used in JULEA, therefore not adding new dependencies. While encapsulating the metadata using BSON makes using different CPU architectures easier, it adds overhead due to packing and unpacking. Note that JULEA backends can be used either on the client or as a server. Since SQLite does not require its own server to run, it will be used on the server side in the evaluation. In contrast, having to employ an additional server when running a MySQL/MariaDB server adds unnecessary overhead. Thus, the MySQL backend is used on the client side.

Additional Benchmarks

Adding a new type of client and backend to JULEA also required respective benchmarks. The benchmarks were implemented by Michael Straßberger[1]. In Listing 7.1 a small subset of the database benchmarks is shown.[2] The benchmarks are part of the evaluation because the database client and backend are the direct foundation of the DAI. As they allow examining the database performance below the application layer is not necessary to evaluate all possible data access patterns at the DAI layer. Note that this thesis focuses on managing self-describing data formats more efficiently on HPC systems, and database systems are simply the chosen solution. Database improvements are not part of the thesis.

```
1 j_benchmark_add("/db/iterator/get-simple", benchmark_db_get_simple);
2 j_benchmark_add("/db/iterator/get-simple-index-single",
      ↪ benchmark_db_get_simple_index_single);
3 j_benchmark_add("/db/iterator/get-simple-index-all", benchmark_db_get_simple_index_all);
4 j_benchmark_add("/db/iterator/get-simple-index-mixed", benchmark_db_get_simple_index_mixed);
5 j_benchmark_add("/db/iterator/get-range", benchmark_db_get_range);
6 j_benchmark_add("/db/iterator/get-range-index-single", benchmark_db_get_range_index_single);
7 j_benchmark_add("/db/iterator/get-range-index-all", benchmark_db_get_range_index_all);
8 j_benchmark_add("/db/iterator/get-range-index-mixed", benchmark_db_get_range_index_mixed);
```

Listing 5.6: An excerpt from the new database benchmarks: For brevity's sake only the benchmarks evaluating the performance of the read operation based on the database iterator are shown. They were selected because the DAI interface will make heavy use of them.

5.1.2. Using a Relational Database for HDF5

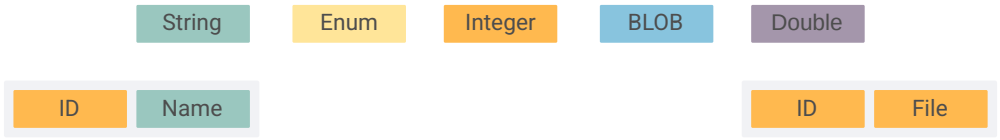
In the following, the straightforward design and implementation of mapping HDF5 to a relational database are discussed. Defining a fixed SQL schema works well for a standardised file format. Thus, frequent changes to the database schema are not to be expected. Using a relational database provides the option to perform more detailed and advanced metadata queries in comparison to the prototype using the key-value store.

[1] The pull request can be found here: `https://github.com/julea-io/julea/pull/82`

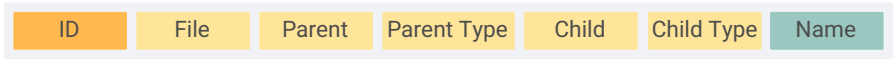
[2] The complete benchmark can be found here: `https://github.com/julea-io/julea/tree/master/benchmark/db`

One aspect that will be discussed more in the next sections is whether a format that potentially uses very deep hierarchies is suitable for a relational database. At this point, initial work suggested that wide-column stores or even graph databases would be a better option if the hierarchies were indeed deep [Lobedank, 2021]. However, it was not yet clear how the format was used. This is one of the reasons that inspired the user survey to see whether theoretical problems like these are an actual obstacle in practice. Warnke implemented the original prototype [Warnke, 2019].

File & Groups An overview of the fields stored for a file is given in Figure 5.1a. The file schema is straightforward and consists only of the file name and the auto-generated database ID. The other SQL schemata hold a reference to that file to ease identifying the related data of an HDF5 file, which allows more efficient truncating. Groups have a similarly simple schema as illustrated in Figure 5.1a. Besides containing the auto-generated ID, only the file ID is stored.

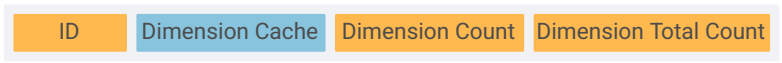


(a) Schemata for the HDF5 file table (left) and HDF5 group table (right)

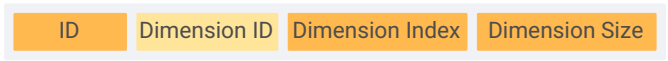


(b) Schema for the HDF5 link table

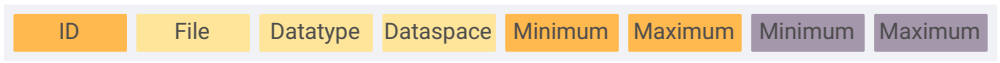
(c) Schema for the HDF5 datatype table (shortened version)



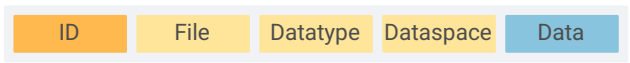
(d) Schema for the HDF5 space header table



(e) Schema for th HDF5 space table



(f) Schema for the HDF5 dataset table



(g) Schema for the HDF5 attribute table

Figure 5.1.: Database schemata to store the file metadata of different HDF5 objects.

Link In contrast to the implementation of the KV VOL plugin, here, links are supported explicitly. Each data structure of an HDF5 file needs to be stored in the link table shown in Figure 5.1b. Links are used to define any dependencies between data structures, even cyclic ones. The fields for parent and child hold the respective database IDs. The parent and child types indicate the target table of the corresponding attribute.

Dataset & Attributes Figure 5.1f shows the structure for HDF5 datasets. As for julea-kv, julea-db does not store the data inside the database backend. Instead, it is handled by JULEA's object store backends [Duwe and Kuhn, 2021b] which is designed for the large data sizes commonly encountered in HPC environments. The database ID for a dataset is used as the key in the object store. Note that the extrema are not originally part of HDF5. However, they were found very helpful for ADIOS2 and thus also incorporated in this VOL plugin.

To be able to work on the values directly instead of opaque objects like arrays, a different set of fields is necessary for each datatype, as illustrated by the two minima and maxima fields. The extrema could also be stored in a BLOB, but then queries could not work directly on them. Instead, every value would have to be read from the database and unpacked to be used in a query.

Both attributes and datasets have an additional entry within the HDF5_datatype and the HDF5_space schema. Attributes have a very similar schema to the datasets. The main difference is that the attribute's data is stored in the database as well because accessing the object store for this introduces unnecessary overhead. As the attribute's data is only small, it is no problem for the database performance or its size.

Datatype In contrast to the HDF5 space mentioned below, the datatype describes a single element, HDF5 supports data type definitions that can be very complex. At the moment, only a simple subset is enabled in this plugin. The most important fields for the basic datatypes that are currently supported are shown in Figure 5.1c. A full schema definition can be found in [Kuhn and Duwe, 2023]. Most fields are straight-forward, e.g. `size` stores the variable size in bytes. The binary HDF5 type is stored in the type `cache`. The `class` field holds the variable type. The `sign` is only required for integer types. There are a number of other fields further specifying details of the datatype such as `order`, `precision` and `offset`.

Space The HDF5 space defines the dimensions of a multidimensional array. As the space can contain a varying number of dimensions, one table is not sufficient to represent this behaviour [Warnke, 2019]. The number of dimensions in the DB VOL plugin is specified through the space header. Then, the range of each dimension is set in the space table. The dimension size of the space determines the allowed range within the dimension [Warnke, 2019].

5.1.3. Investigating Database Technologies

When splitting the format, it no longer restricts the possible metadata. The work with ADIOS2 showed that storing the extrema helps query the variable range. Therefore, the DB VOL plugin contained the minima and maxima for integers and doubles. One idea here is to pre-compute

results that are used in post-processing. Imagine analysis working predominantly on the mean values instead of the original data. Then, pre-computing the mean value, e.g. as part of the I/O process, would not only speed up post-processing but, more importantly, reduce the number of data accesses required for the analysis. Simply retrieving the mean values would be sufficient then. So, if the database and thus this custom metadata is stored on fast hardware, then the runtime of post-processing analysis can be significantly reduced.

Even though the interface and implementation of the pre-computation are discussed in the next chapter, the general idea is required to allow a sensible discussion on enhancing the current solution, i.e. the key-value engine. Therefore, custom and pre-computed metadata is a large focus in the following sections and the next chapter. For now, it is presumed that there are typical operations performed in most post-processing and analysis. That this is indeed a common scenario will be detailed in Section 6.1.3. In summary, the analysis of climate simulation output is often performed on aggregated data like the mean value.

There is a wide variety of terminology of how this approach can be called, ranging from *in-situ post-processing* over *interleaving computation and I/O* to calculating *derived metadata* or *aggregations*. Each of these terms has a slightly different notion and comes from another background or community, e.g. data science, HPC, database systems and big data. A rough overview of these perspectives and their underlying goals and motivations will be given throughout this chapter. In this thesis, the approach will be referred to as the pre-computation of results.

Custom Metadata in Relational Databases The access granularity of key-value stores was mentioned to be insufficient for operations more refined than simply reading the complete variable. An obvious candidate with support for advanced and complex queries is relational databases like MariaDB, for example. Relational databases are also very suitable for storing a clearly defined structure as defined by the self-describing data formats. As discussed in Chapter 3, they have proven useful in approaches like EMPRESS and EMPRESS2. Therefore, the HDF5 DB VOL plugin prototype was developed using relational databases that worked well for the initial task.

However, when the idea arose to add custom metadata, the structured approach of the relational databases seemed unnecessarily rigid and inflexible. Also, while relational databases are deemed to be a new approach in the eyes of many HPC people, database researchers have a different view. Therefore, data warehousing was investigated as the next step in the engine evolution to see whether it would be a suitable replacement. As the lines get more and more blurry between terms like data warehousing and big data, both are studied in the following. As a side note, the difference between HPC and big data is also increasingly diffuse, with a promised convergence of both topics, possibly drifting towards cloud computing. So, the question was what approaches exist and which technologies could be suitable as the basis for the next ADIOS2 engine and for managing custom file metadata.

OLTP and OLAP

Two vital terms when talking about data warehousing are OLTP, i.e. online transaction processing, and OLAP, i.e. online analysis processing. They were developed decades ago to optimise for a specific scenario [Chaudhuri and Dayal, 1997].

There are differing opinions on whether this separation should be kept. For example, Conn finds that one data model cannot work well for the different approaches, whereas Plattner states this separation is not necessarily still useful [Conn, 2005, Plattner, 2009].

OLTP focuses on operational database support, which can be considered traditional DBMS. The respective systems provide the infrastructure for general data management and storage, whereas OLAP systems offer the analytics perspective. OLTP is explicitly optimised for inserts and updates ranging from microseconds to milliseconds. OLTP systems are often NoSQL databases like MongoDB, Cassandra and HBase. With the focus on inserts and updates, latency is an important metric. OLTP systems have low to moderate parallelism.

OLTP systems are often the basis for OLAP processing on top. Data warehouses are designed for decision support. Thus, data from various sources is stored in the data warehouse to enable in-depth analysis. Data is curated through the ETL process, i.e. extract, transform and load. To allow continuous analysis and track progress over time, data is potentially kept for a longer time frame than in OLTP systems. Combining various data sources and storing them long-term can make data warehouses very large. OLAP systems typically target aggregations and parallel querying. Such systems are optimised for large read workloads in the range of TB or PB per operation. There exist several modifications of OLAP, namely MOLAP and ROLAP. The former works on multidimensional data, whereas the latter uses an extended relational database.

In contrast, data lakes delay this process as much as possible to reduce the overhead of the costly ETL process. The underlying assumption/fact is that most of the data will not be used. So, to decrease the overhead, data is only undergoing the ETL process when it is actually accessed. Smart data lakes take this approach one step further and renounce the data curation completely. Instead, approximate query processing (AQP) and on-the-fly creation of indices are used for efficient data access and to speed up future accesses as well [Sanca and Ailamaki, 2022, Bian et al., 2021]. Thus, they abstract from the different data formats and offer a unified interface.

OLAP Cubes The goal of OLAP is to enable complex analysis and, e.g. visualisation. Data typically has multiple dimensions of interest that can be hierarchical, e.g. time, location, and product category. Taking time as an example, the different hierarchy levels may be hours, days, weeks, months, or years. As mentioned, OLAP often works on aggregations, possibly affecting multiple dimensions. The multidimensional view of the data aggregations is often called an OLAP cube. The most common operations are *rollup* (decreasing the level of detail), *drilldown* (increasing the level of detail), *slice_and_dice* (selecting and projecting data) and *pivot* to change the data view [Chaudhuri and Dayal, 1997]. Rollups are realised through performing additional grouping operations on the dimensions. As OLAP cubes come from the world of business analytics, the typical example data to illustrate possible cube aggregations is sales, as shown in Figure 5.2.

Supporting these types of operations requires a specific data organisation. In contrast to entity relationship diagrams used as a basis for OLTP systems, data warehouses need something different to perform efficient querying and incremental loading operations, that is, denormalised schemes. Thus, they are not integrated into OLTP systems but instead built on

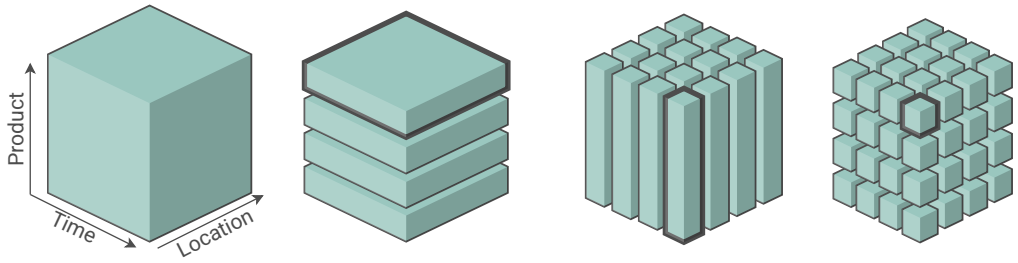


Figure 5.2.: Example cube aggregations on sales data. In the first aggregation, one slice would equal all sales of one product, e.g. 'The Silmarillion, over the entire accounting period, e.g. a year, and all locations, e.g. Europe. Then the marked column of the second aggregation can represent the sales of all Tolkien books in the fourth quarter in Hamburg. The highlighted cube in the third aggregation is the number of sales of 'The Silmarillion in the fourth quarter in Hamburg. These aggregations can be performed for all dimensions. Slices along the time or location dimension could then equal either the annual sales of Tolkien books in Hamburg or the quarterly sales of Tolkien books in Germany.

data warehouses. The most common data layouts are the star schema and the snowflake schema. The former uses one fact table and an additional table for each dimension. These dimension tables are referenced in the fact table. The snowflake schema uses denormalised dimension tables. This denormalisation means that dimensions are represented through multiple tables. Note that the example was chosen because it is the most popular and highlights the technique's origin. Nonetheless, such aggregations can be applied to scientific data just as well as demonstrated by the Python package Iris [Met Office, 2022]. Iris represents climate and weather phenomena as cubes based on NetCDF files. For example, scientists can analyse the air temperature with the longitude, latitude and height as its dimensions.

Cassandra and Other Wide Column Stores A popular technology is wide-column stores that are optimised for column access. They can be viewed as an extension of a key-value store where the value consists of nested key-value pairs [Davoudian et al., 2018]. Wide column stores have so-called column families allowing runtime modifications, i.e. creating and deleting columns [Davoudian et al., 2018]. This feature seemed to be especially useful for managing custom file metadata. Thus, Cassandra was a promising candidate for the database engine.

Column stores also allow better parallel processing because the columns can be accessed concurrently [Plattner, 2009]. Typically column stores are smaller in size because columns can be better compressed. After all, data in a column shares more similarities than that in a row. Plattner found a difference of a factor of 10 when comparing column stores with row storage [Plattner, 2009]. He also stated that enterprise applications are working on set processing rather than accessing complete tuples, making them an excellent fit for column stores. So, the question is, does this also hold for scientific data?

Data Warehousing Design Process

One of the fundamental differences between a relational database and a data warehouse lies in the design process. In classical databases, such as relational databases, it is bottom-up. The data is studied to find a logical and coherent data model capturing all the nuances. Then, the database schemas are designed and implemented to serve as a general-purpose system that is able to answer all queries. Depending on the subfield, different features and operations are considered to be more important than others, write vs update performance, and point vs range queries. In contrast, data warehouses are designed from top to bottom. Not the data is studied, but the user behaviour and what questions are most likely to be posed. Then, the data representation is chosen accordingly to work well with these most common queries. Thus, the resulting question is, again, how scientific data is accessed and what questions HPC users will ask. Therefore, an in-depth data analysis is presented in Section 5.2.

5.2. Data Access Analysis

This section continues to answer research question 2.1. (How can metadata and data be queried efficiently?) In the previous section, database technologies were studied, and their respective features are examined concerning the management of self-describing data formats. However, more information about how users access data is required to choose a suitable candidate. Therefore, common data layouts are studied first. Afterwards, the performance implications are discussed given different writing and reading access patterns. Next, examples of data layout optimisations are explained. Finally, the first part of the conducted user survey is presented, where the use of self-describing data formats is the main focus.

Many workloads in HPC are WORM (write once read many), in contrast to typical database workloads that contain frequent updates. Another typical scenario in HPC is writing checkpoints. Scientific simulations run for a long time, meaning the computation will not finish inside the possible allocation time on the cluster. Therefore, applications running on supercomputers typically write checkpoints, saving the current state of the simulation. These checkpoints can then be read on start-up and fed back into the simulation to continue the computations.

In recent work, Purandare et al. performed a large-scale analysis of CERN workloads of over 2.49 billion events [Purandare et al., 2022]. They found that only 0.22% are updated at all and only 0.09% more than once. Furthermore, the examination revealed a total write size of over 150 PB and a total read size of over 300 PB. These results are in contrast to the previous analysis, for example, by Grawinkel et al., that found read access to only 9% of the entire ECFS system, which is the file system of the ECMWF [Grawinkel et al., 2015].

5.2.1. Common Data Layouts

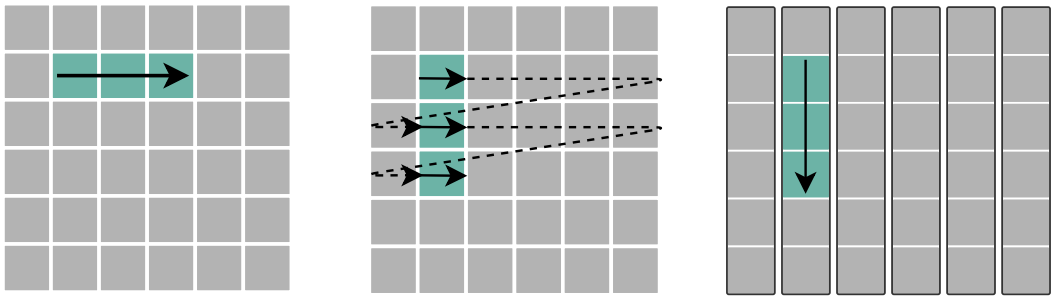
In order to improve the I/O performance, in-depth knowledge is required about the typical data layouts and how they influence the performance. Therefore, the most common data arrangement techniques are discussed in the following. These are managing a logically contiguous file, chunking datasets and sub-filing. While some of them may seem trivial, recent

work by Wan et al., studying the performance impact of storage layouts on exascale applications, shows that even supposedly simple layouts can become challenging to optimise [Wan et al., 2022]. One factor contributing to the complexity is the dynamic load balancing of applications. Subsection 5.2.1 and 5.2.2 follow the arguments made by Wan et al. and highlight some of their crucial findings.

Logically Contiguous Layout

A standard data structure is the so-called *logically contiguous* layout. Logically, in this context, this means the actual on-disk format may not be in one piece. For example, storing contiguous data in a parallel and distributed file system can lead to the striping of the data across multiple servers [Wan et al., 2022].

Data can be stored in a row-major (C) or column-major (Fortran) version. In Figure 5.3, reading a row is compared with reading a column in a row-major arrangement. To read the highlighted elements, only one access is required, as shown in Figure 5.3a which is the same for reading a data chunk as depicted in Figure 5.3c. As illustrated in Figure 5.3b, accessing three elements of a column in a row-major layout needs three accesses of one element each. One solution to tackling the latter is data sieving. When using data sieving, all elements between the first and last requested ones are read. Then, the data is sieved, essentially discarding all other elements. Depending on how much of the data will be discarded, this technique works because accessing a contiguous piece of data on traditional storage mediums like HDDs is faster than multiple smaller accesses. This is mainly due to the general concept of an HDD where a write-read head is hovering above a spinning disk. In I/O libraries like HDF5, data sieving is only possible for contiguous datasets.



(a) Reading part of a row **(b)** Reading part of a column **(c)** Reading chunk

Figure 5.3.: Reading access from different data layouts. Redrawn from [HDF-Group, 2019]

Globally Contiguous Shared File In Figure 5.4 the challenge of writing a row-major array with multiple processes is shown. While the data held by each process is potentially logically contiguous, it is not adjacent from a global perspective [Wan et al., 2022]. In order to create a layout that is also globally contiguous, substantial data movement can be required. A lot of coordination in the form of MPI communication is involved in managing the rearrangement,

for example, collective operations. This process becomes very expensive for high-dimensional data when many processes perform I/O.

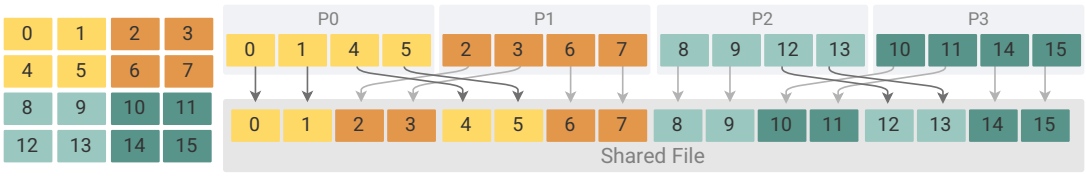


Figure 5.4.: Writing a logically contiguous file using four processes. A 2D array (here row-major) is partitioned as shown on the left. Top: The in-memory data layout for each process Bottom: Logically contiguous data arrangement in the shared file. Necessary data movement is indicated by arrows. Reprinted with permission © [2022] IEEE [Wan et al., 2022]

Chunking

As a result of the high cost of the data rearrangement, other layout approaches are used, such as chunking. Here, the application data is not linearised as contiguous data. Instead, the data is grouped into so-called chunks that provide a small contiguous data block. Therefore, they can be managed separately. When using chunking, each process reserves the required space in the file to write its data. So, as shown in Figure 5.5, the chunks are written to one file independently of another, which allows for reducing the overhead. Furthermore, chunking can also improve the read performance of data subsets as chunks can be accessed separately.

However, when not done carefully, chunking can decrease the performance and even worsen the rearrangement costs [Wan et al., 2022]. Assume, for example, that the chunks are set as columns. Then, the data of different processes is contained in one chunk resulting in a large communication overhead. Also, the data of a process may not be large enough or not have a suitable shape for chunking. Two-phase I/O can be used to mitigate the rearrangement overhead [Tessier et al., 2017, Kumar et al., 2019].

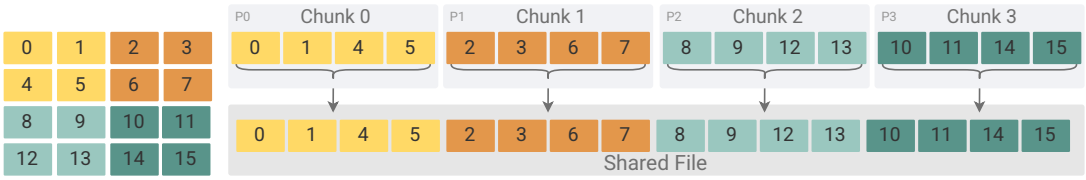


Figure 5.5.: Writing a chunked 2D array with four processes, each using one chunk. Each process allocates the appropriate size in the file allowing independent access. Reprinted with permission © [2022] IEEE [Wan et al., 2022]

In the following, HDF5 is used as an example to illustrate the intricacies introduced by these seemingly simple layout optimisations. Parallel HDF5 offers several tuning parameters related to chunking. The first is the *chunk size* itself. If the chunk size is too small, the overhead of the additional chunk management becomes unfeasible. Also, more chunks lead to more access operations as well as larger file sizes due to the extra storage required for the

chunking information. On the other hand, if the chunk size is too large, then too much data needs to be written and read, even for small selections.

To improve the chunking performance, HDF5 employs the concept of a chunk cache [HDF-Group, 2019] where each opened dataset has its own cache. Therefore, the second tuning option is the *cache size*. In general, the cache has to be big enough as well. Otherwise, simply accessing a few chunks will lead to eviction. Furthermore, the chunk size should not be larger than the size of the chunk cache. Like many other caches, the chunk cache uses a hashtable to improve the lookup speed. The *size of the hashtable* is another tunable to be kept in mind. Besides setting the tuning parameters unfavourably, the chunk cache itself can harm the performance as data that could have been contiguous on disk can now only be access in chunks.

All these decisions can greatly impact the application's performance. However, an application user or even an application developer is not necessarily familiar with all the different tuning parameters and how they should be set in relation to each other. In-depth knowledge of the application's behaviour, the storage system and the post-processing is required to choose a sensible setup. Yet, making every application user a storage expert is not feasible.

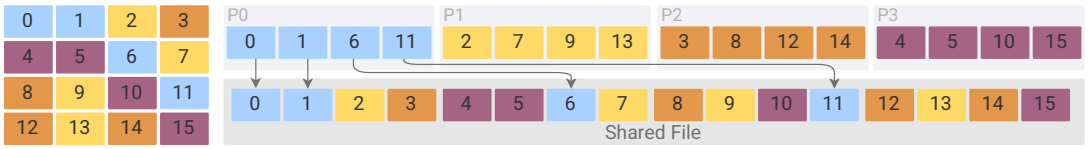
Therefore, more decisions have to be made by the I/O libraries. ADIOS2, for example, does not offer custom chunk sizes. Every block is considered a separate chunk. Thus, ADIOS2 uses a log-structured file format to avert MPI coordination for the layout management [Wan et al., 2022]. As a result, ADIOS2 needs additional metadata to note the location of the chunks concerning the global dimensions as mentioned in Section 4.1. Here, coupling the I/O library with the storage systems offers new possibilities as the different layers can exchange information about the data access.

Sub-Filing

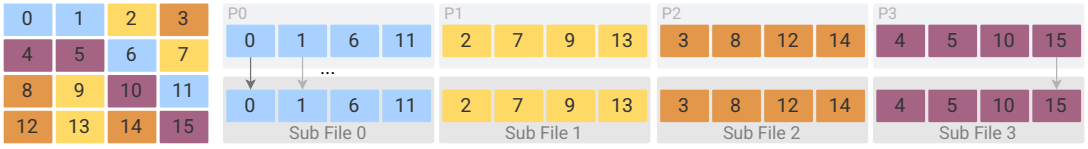
As previously discussed, chunking has less overhead to rearrange data than the logically contiguous data layout. However, parallel access can easily lead to locking problems as illustrated in Figure 5.6a [Wan et al., 2022]. Here, a 2D array is split into 16 chunks. Each process works using four chunks. The difference from the previous examples is that the chunks belonging to one process are not necessarily next to each other from a global perspective. Such patterns often occur when running applications that employ dynamic load balancing, which focuses on distributing the computation load equally [Wan et al., 2022]. This can lead to sub-optimal decomposition in terms of storage layout and thus very intricate I/O patterns [Rowan et al., 2021]. Nevertheless, load balancing is important to fully use the hardware capabilities [Huang et al., 2018]. Besides dynamic load balancing, adaptive mesh refinement (AMR) can also cause these patterns. As a result, each process has to access the file at four different offsets, as shown in Figure 5.6a. The parallel access at different offsets possibly leads to conflicts upon file locking.

.

So far, all processes have shared access to the same file. The opposing approach is called sub-filing. As illustrated in Figure 5.6b, every process uses its own sub-file. Therefore, the I/O of different processes can happen more independently from another. However, the increase in files can lead to several problems [Wan et al., 2022]. The first obstacle is the additional



(a) Without sub-filing: Resembles the contiguous layout albeit with a more complex data distribution.



(b) With sub-filing: The different chunks of a process are gathered and written into a separate sub-file.

Figure 5.6.: Writing a chunked 2D array with four processes (four chunks each) without sub-filing (a) and without sub-filing (b). Reprinted with permission © [2022] IEEE [Wan et al., 2022]

overhead of managing the chunks in different files. The second challenge arises when many processes are used to run the application. Then, an even larger number of comparatively small files is created, putting additional strain on the parallel file system, particularly the metadata management. Lastly, the read latency is increased dramatically because all sub-files need to be found and accessed.

5.2.2. Various Access Patterns

In the following, different access patterns and their potential performance implications are studied.

Writing and Reading One Variable with Various Patterns

Now that the common data layouts are described and discussed, the study performed by Wan et al. is examined [Wan et al., 2022]. They use WarpX, a physics application that models a particle-in-cell accelerator using AMR. Parallel HDF5 and ADIOS2 are the employed I/O libraries. Logically contiguous and chunked layouts are performed with HDF5, whereas ADIOS2 covers sub-filing. Parallel HDF5 is not using the collective mode as this is not usable for load-balancing codes.

The simulation is run on up to 1,024 nodes, each using six MPI processes. Note that workload-aware striping is not examined because applications cannot tune the stripe sizes on Summit, as the used GPFS does not allow this. As GPFS uses a block size of 16 MB, the minimal chunk size is never set smaller.

In Figure 5.7, the rearrangement overhead of the logically contiguous layout becomes obvious. Chunking can increase the writing performance significantly. However, both strategies are outperformed by chunking with sub-filing by an order of magnitude, especially for a larger number of nodes. While the file-per-process strategy works well for fewer nodes, the

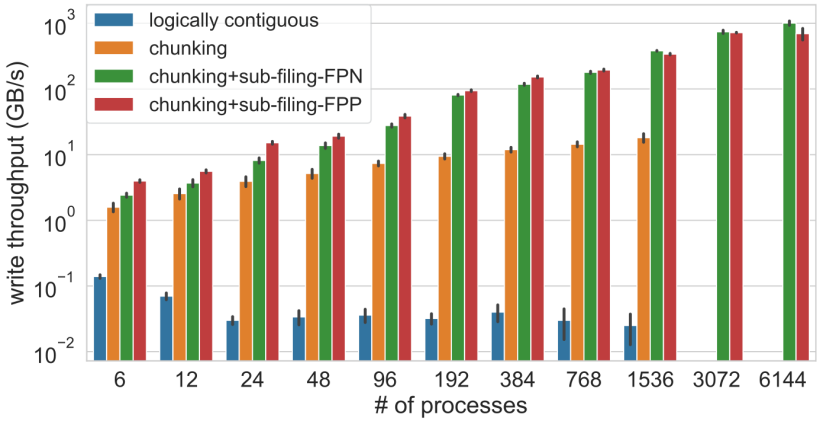


Figure 5.7.: Write performance of WarpX on Summit using Parallel HDF5 for logically contiguous (blue) and chunked data (orange) and ADIOS2 for two different sub-filing strategies: FPN means file per node (green), whereas FPP means file per process (red). Note the logarithmic scales on both axes. Reprinted with permission © [2022] IEEE [Wan et al., 2022]

file-per-node approach eventually becomes more performant. As previously mentioned, this is because of the overhead of managing a large number of small files within a parallel and distributed file system. The HDF5 results for 3,072 and 6,144 processes are missing because they did not complete within the allocated time as a result of their low performance [Wan et al., 2022].

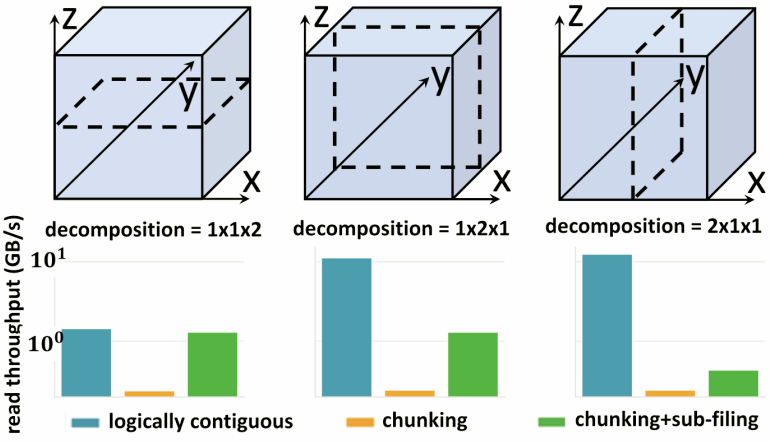


Figure 5.8.: Read performance of WarpX on Summit using different decomposition. The decomposition 1x1x2 splits the data in two along the z axis; 2x1x1 and 1x2x1 along the x and y axis respectively. Note the logarithmic scale on the y axis. Reprinted with permission © [2022] IEEE [Wan et al., 2022]

The performance of reading a 3D mesh variable of 256 GB using two nodes is shown in Figure 5.8. Here, Wan et al. studied the impact of decomposition on the read performance. As the variable is split along the different dimensions, each process read half of it. This evaluation

shows that the different data layouts are immensely important for the read performance. It also illustrates that not only the data layout is crucial but the decomposition is as well.

Overall, the evaluation clearly demonstrated the tradeoff between write and read performance. The logically contiguous layout has a very poor write performance. However, when reading data, it is as good as chunking with sub-filing or better for two decomposition schemes. Wan et al. find that none of these layouts works well for both writing and reading.

They further evaluate the read performance with a variety of different access patterns shown in Figure 5.9. These are typical access patterns for 3D mesh variables. This set of patterns was first gathered and analysed by Lofstead et al. [Lofstead et al., 2011]. It captures the challenges common to scientific data access. A single variable can be read with basically any type of access pattern. The rest of the evaluation by Wan et al. is summarised in text for brevity's sake. It showed that for a small number of processes (eight or fewer), the contiguous layout continues to perform well for five of the patterns. Only when reading the XY plane did it fall behind chunking and sub-filling. However, when the number of processes is increased, the number of parallel read accesses to different regions of the layout also increases. This essentially means random access to the storage.

The drawbacks of reading chunked data were also clearly visible as it did not perform well for any of the access patterns. This is due to the need to find, access and rearrange the chunks to form a contiguous layout in memory. The read performance can be increased using a larger chunk size, which reduces the write performance, as previously mentioned. For larger numbers of processes, chunking and sub-filing work well. However, for fewer processes, the overhead of merging the chunks is higher than the overhead of concurrent read requests to the contiguous layout.

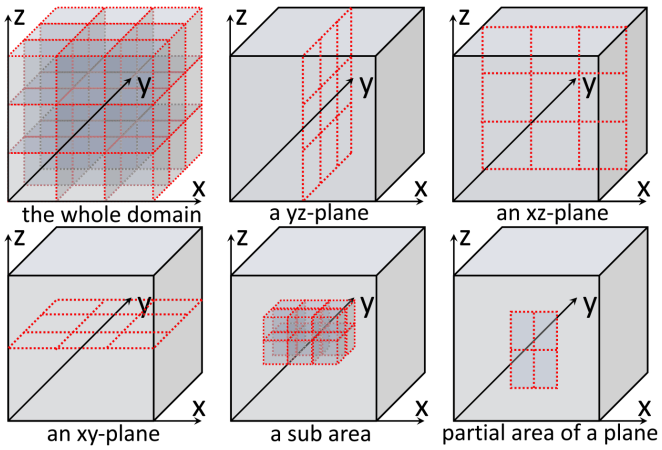
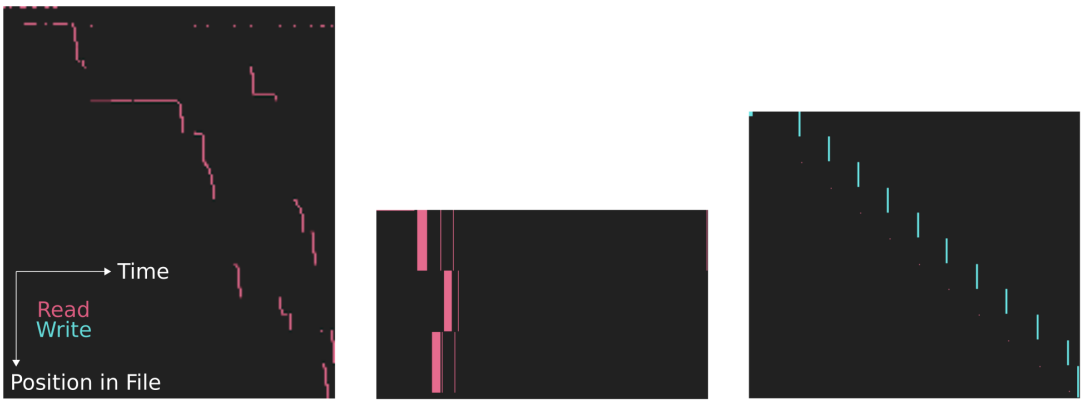


Figure 5.9.: Six typical access patterns on variables from I/O libraries like HDF5 and ADIOS2. The red lines indicate the parts that are written in each pattern. Reprinted with permission © [2022] IEEE [Wan et al., 2022]

Miscellaneous Patterns Found in the Wild

Besides the very clearly defined access patterns shown in Figure 5.9, other common patterns are shown in Figure 5.10. They were captured by Lüttgau on the DKRZ system Mistral when running the ICON (Icosahedral Nonhydrostatic Weather and Climate Model) simulation by the DWD (Germany's National Meteorological Service). ICON uses an unstructured icosahedral grid. Therefore, the coordinates have to be read to allow mapping the data previously written to the respective location in the grid. While all 196 processes need a subset of this information, only 14 processes are accessing the file. The resulting pattern is shown in Figure 5.10a. Reading the input file required at the beginning of a new modelling run is illustrated in Figure 5.10b. 192 processes are accessing the file. The repeated access to specific locations is sped up as indicated by the quick access. A probable reason is caching in HDF5. In contrast to the previous two examples, the third pattern shown in Figure 5.10c depicts writing; more specifically, the asynchronous writing of a checkpoint using just one process. As mentioned before, checkpoint files are used to preserve the state of the computation in case the simulation times are longer than the allowed allocation time on the cluster. It can then be used to continue the computation in the next job.



(a) Reading Coordinates **(b)** Reading an Input File **(c)** Writing Checkpoints

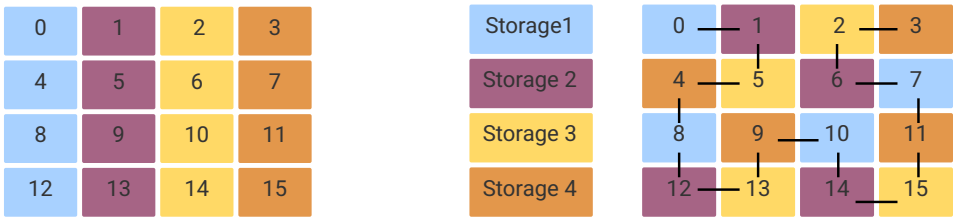
Figure 5.10.: Access patterns of the ICON (Icosahedral Nonhydrostatic Weather and Climate Model) application, observed on Mistral, DKRZ's previous supercomputer. [Lüttgau, 2021]

So what do I/O libraries use?

Given the disadvantages of all the previous approaches, the question arises whether any I/O library employs them. This will be examined in the following paragraphs.

HDF5 HDF5 offers all of the mentioned data layouts and leaves the decision to the user. The chunking and the respective cache have been discussed before. Subfiling in HDF5 is essentially implemented through the virtual datasets introduced in 1.10. They offer a unified view of data split across different files and datasets. In JULEA, the VOL plugins currently support only contiguous datasets that are striped when the distributed object client is used.

ADIOS2 ADIOS2 uses a combination of various strategies, among others chunking and sub-filing, despite their performance problems. This design decision was made early on for ADIOS following the insights by Sarawagi and Stonebraker in 1994 that the query performance depends on the dimension and not the query size [Liu et al., 2014, Sarawagi and Stonebraker, 1994] Therefore, chunking instead of a contiguous layout is supported. Aggregations are used to lessen the performance impact of sub-filing. Furthermore, intensive research went into studying the nature of parallel access and the effects of different placement strategies such as optimised chunking [Tian et al., 2012].



(a) Round-robin ordering **(b)** Hilbert curve ordering

Figure 5.11.: Linear data placement orderings. For the second distribution strategy, a Hilbert curve (here of second order) is computed that fills the entire space. The chunks are then distributed to the storage nodes according to the order determined by the Hilbert curve.

To illustrate the complexities, one short example is given in the following. Here Hilbert space-filling curves are used to enable a balanced distribution and thus to reduce the conflicts of concurrent accesses [Lofstead et al., 2011, Tian et al., 2011, Liu et al., 2014]. In Figure 5.11, round-robin and Hilbert curve ordering are illustrated. Assume four storage nodes are used. When Hilbert curve ordering is used as shown in Figure 5.11b, the first two chunks are stored on the same storage nodes as if the round-robin strategy was used. However, the third node will get chunk five and the fourth chunk four. The advantage of Hilbert curves and the reason they are used here is that they offer the possibility to fill a two-dimensional space with a continuous one-dimensional curve [Wikipedia, 2021]. This allows transforming approaches that rely on a linear order into a multi-dimensional space. So, it is used here to the balance the distribution of all dimensions of multi-dimensional data. Further chunking within the ADIOS2 process group helps level the number of seeks and the overhead of large datasets. The results show that the read performance, especially of planar read, is improved by up to a factor of 37.

5.2.3. Survey on Format Usage

In order to have a better understanding of the users'/developer's needs, a small survey was conducted. This also allowed us to include different experiences in the design process. The survey questions are summarised into larger question groups (QG), namely QG 1 and QG 2. QG1 examines the actual usage of the self-describing data formats, whereas QG2 focuses on the used interfaces and the users' preferences.

In the following, the survey setup and questions are presented. Afterwards, the results for QG1 are shown. The answers for QG2 will be illustrated and discussed in the next chapter to inform the design decisions for the DAI interface.

LimeSurvey The survey was conducted using the LimeSurvey instance of the Otto von Guericke University[3]. LimeSurvey is an open-source solution that allows creating of online surveys. Limesurvey offers various features, among them collecting and anonymising the IP addresses of the participants which were chosen to reduce the number of very blatant resubmissions. However, it was not used for any evaluation. Instead, the submission ID is used to evaluate the answers in the context of one another. Assuming users indicate using an uncommon file format, it might be especially interesting to see what post-processing operations they use.

Survey Design To get the user accustomed to questions and to gain some basic understanding of the participants, some background information was collected first. Then, the format usage is the next larger topic. Finally, post-processing and analysis are examined. The answers were multiple choice with several pre-made options and the possibility to add additional information via a text field. Choosing more than one answer per question allowed to get a more complete view of the behaviour. To reduce friction and avoid unwillingness to answer the survey altogether, there were no hard requirements or large text fields to complete. Furthermore, the question catalogue was condensed to more general statements instead of presenting a new interface prototype and asking the participants about that. Nonetheless, the answers allow conclusions to inform the design and implementation decisions of the database engine and the DAI as well.

Duration and Participants The survey was started on 2022.05.16 and essentially ran for two weeks. While it was left open for two months, all responses came in in the first two weeks. The survey was sent to scientists at several institutes, including but not limited to the German Climate Computing Center (DKRZ), Max-Planck-Institute of Meteorology (MPI-M), German Electron Synchrotron (DESY), Center for Bioinformatics (ZBH), University of Hamburg (UHH), Helmut-Schmidt University (HSU), German Aerospace Center (DLR), DDN, Sandia National Labratories (SNL), University of Tennessee, Lawrence Berkeley National Laboratory (LBNL), and McGill University. Additionally, the survey was posted in the HDF5 user forum as well as in the ADIOS discussions on GitHub. This enabled feedback from the library developers themselves. In total, 38 people participated, and 22 filled in the questions completely.

Questions The complete survey, including all images and code examples presented, can be found in Appendix C. The exact phrasing of the question is mentioned in the respective discussion of the results. The question groups and their general focus area are summarised in the following.

[3] `https://limesurvey.ovgu.de/index.php/admin/index`

Question Group 0:
- Workplace & scientific field

Question Group 1:
- Which formats, hierarchy depth
- Number of variables per file, number of attributes, number of time steps
- File per variable/thread/process/node/timestep
- Files per simulation run & file size

Question Group 2:
- Programming language for simulation & post-processing and analysis
- Typical computations in post-processing and analysis
- Used Tools and library packages
- Interface rating
- General feedback

Limitations This survey has several limitations that reduce the significance of the results. First and most important, no specific user selection was set into place. The participant selection was essentially anonymous as the survey was sent to various colleagues and known research institutions. There was no filtering or background check besides a note in the distribution email to ensure only suitable candidates answered the survey. Not everyone that the email reached is equally qualified to provide feedback. Furthermore, there was no evaluation of the experience with specific formats or post-processing. Thus, the noticed usage may not be indicative of, e.g. good format usage.

Additionally, while the number of participants is high enough to justify a detailed examination, it is too small to allow representative conclusions for the entire HPC field or even the earth system community. Another aspect is the bias possibly introduced through the provided answer options. They unintentionally steer the participants in a direction they may not have taken otherwise.

Also, the anonymised IP tracking avoids multiple submissions by the same person only at a very superficial level that can be evaded through using different devices or a VPN. However, it is assumed that motivation to intentionally sabotage such a small survey is very low as there is nothing to gain from it. Following this assumption, it is also unlikely that participants knowingly stated the untruth.

Results

In the following, the results concerning the format usage are presented. Only answers from completely filled surveys are shown, i.e. 22 from 38. As previously mentioned, the submission IDs are captured so that the answer behaviour of different participants across the survey can be observed. As ADIOS2 is the focus in terms of formats, the answers that ADIOS2 users gave, i.e. all participants that selected ADIOS2 as a used format, are highlighted in the text.

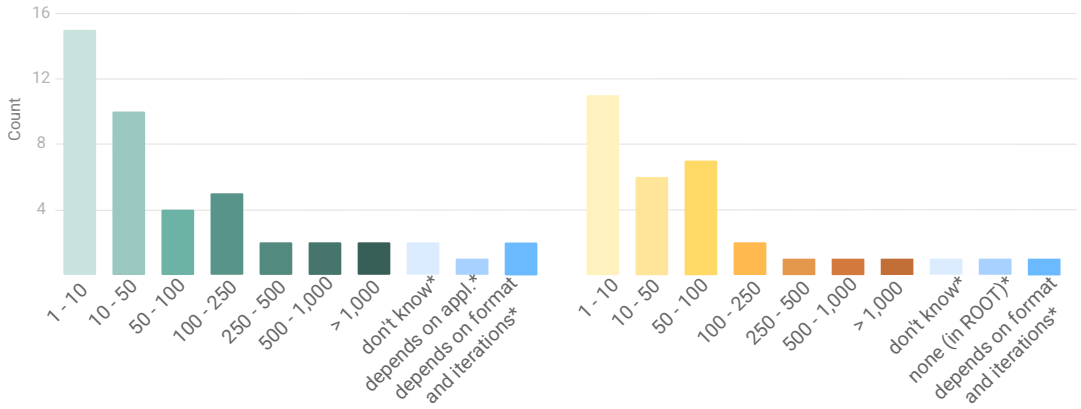


Figure 5.12.: Survey results (QG 1) for the number of variables (green) and attributes (yellow) used per file. Answers given using the free text field are coloured in blue and marked with an asterisk.

Variables *How many variables (ADIOS2)/ datasets (HDF5) do you use per file?* The results regarding the number of used variables and attributes are illustrated in Figure 5.15b. The majority of votes (25/44) indicate that most participants use between one to ten variables (15) or ten to 50 (10). There is a clear trend favouring a small number of variables per file. Studying the answers in more detail, it becomes clear that only two participants used more than 250 variables. It is also those two who chose the options '500 - 1,000' and '> 1,000'. Both have a physics background and therefore use NeXuS and FITS besides HDF5 and NetCDF. So, only a few participants used a greater number of variables. Indeed, nine participants only used one to ten variables per file, as they only chose the first option. Of the five participants that indicated they use ADIOS2, three only use one to ten variables.

Attributes *How many attributes do you use per file?* Most participants also use a similar number of attributes compared to the number of variables. Overall, they chose fewer options. There were only 32 votes, whereas there were 44 for the variable usage. This voting behaviour leads to the impression that the attribute usage is not as varied. Interestingly, the participant that uses over a thousand variables per file also uses over a thousand attributes per file. Others use 100 - 250 variables and 500 to 1,000 attributes. Eight participants only use one to ten attributes. Six thereof only use one to ten variables as well. The others use up to a hundred variables, meaning they use more variables than attributes. Thus, there are all combinations of how both concepts can be used together with a clear trend to using only a small number of variables and attributes.

The other answers from the text fields mainly state that it depends on the application and format. One participant noted to only consume HDF5 files and is, therefore, unsure how the data is generated. Therefore, the same remark is given for all the following questions except the timesteps. An insightful comment is provided by an astronomer mentioning that the number of used variables also can depend on the number of runs or iterations. This may

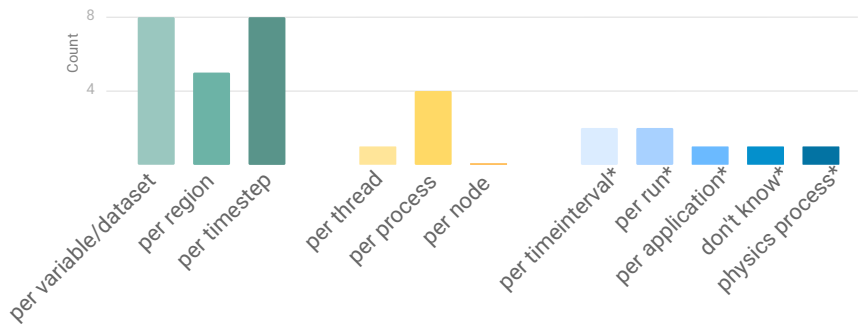


Figure 5.13.: Survey results (QG 1) on how files relate to high-level concepts of ADIOS2/HDF5 like variables and timesteps (green) and cluster concepts (yellow). Answers provided using the free text field are coloured in blue and marked with an asterisk.

be due to increased resolution or new events, e.g., when two particles collide and decay into numerous other particles.

File Usage *Which answers describe your file usage the best? One file ...* In Figure 5.13 the possible data access is examined. The question was posed to understand which concepts users typically gather in one file. Sixteen participants only chose one answer. The most common pattern is using a file per variable/dataset or timestep, with eight users each. Note that two additional participants added a remark that they use a file per specific time interval, e.g. a day or a year. They possibly wanted to distinguish their usage from the timestep concept of ADIOS2. Another popular one is the usage of one file per region, i.e. physical location. Lastly, 5 participants typically have a file per process or thread.

These answers reinforce the impression that data access is incredibly varied across different user groups. Even employing one file per timestep works entirely different than using the step concept of ADIOS2, where steps are clearly defined and recorded in one file. Also, having a file per dataset indicates that complex hierarchies are often not used. This impression is corroborated by the answers to the next question.

Hierarchy Depth *How deep are the used hierarchies, that is, how many subgroups does a group have typically?* The results regarding the hierarchy depth are presented in Figure 5.14. Fifteen participants used a hierarchy depth of one. Upon studying the multiple choice answers, it turned out that twelve of them exclusively employ one subgroup. Another one utilises one or two, and two participants one to three. That means that among the participants, deep hierarchies are rarely ever used. Interestingly, four of the five participants using ADIOS2 only used one group as well.

Number of Files & File Size *How many files are produced per simulation run?* Most participants use a small two-digit number of files per run, as shown in Figure 5.15a. However,

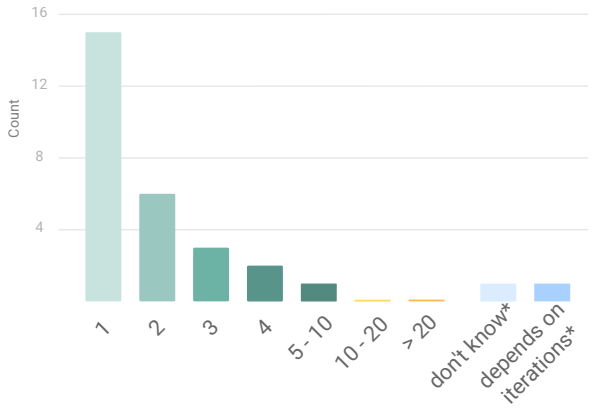


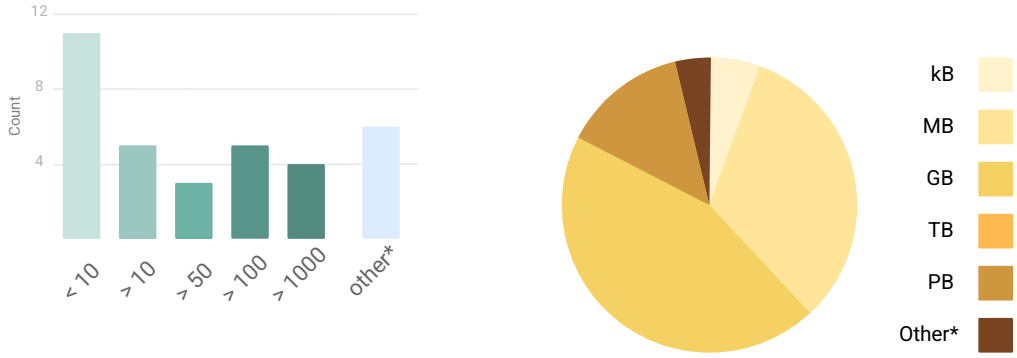
Figure 5.14.: Survey results (QG 1) for the used hierarchy depth, i.e. the number of subgroups. Results from 1 to 10 are coloured in green while the free text answers are coloured in blue and marked with an asterisk.

there are also 4 and 5 participants using over 100 or 1,000 files, respectively. Only 5 participants chose more than one answer. So, for some scenarios, a well-defined number of files is created, while the file count can vary heavily for others. For example, one participant from radio-astronomy noted this number depends on the type of observation performed. In contrast, another participant does not generate any new files but solely works on an existing one. Another added a remark that some runs are not from simulations, but training runs for machine learning leading to a large number of files. A very detailed comment described that this concept is not as clearly defined for particle physics simulations that are data parallel. There, typically less than ten files are written per job. However, often a hundred to a thousand jobs are run at once. To make them more manageable, they are merged to roughly one GB per file afterwards. In summary, having only the number of files is insufficient to derive the potential performance indications. Therefore, the typical file size is examined next.

How large is a file typically? On average, the file sizes are in the range of GB as indicated by 16 participants as illustrated in Figure 5.15b. Eleven also encounter MB-sized files, whereas only a few deal with kB or TB of data in a single file. One participant stated that it varies widely depending on the application.

How many time steps do you use per file? The number of used timesteps is shown in Figure 5.16. Unfortunately, option '100 - 500' was missing in the survey version that went online. While survey modifications in a running survey are possible to a certain degree, keeping the survey questions and, therefore, the answers consistent within was deemed more valuable than updating the premade options. This would have resulted in incomparable answers. Despite this flaw, a general trend can be observed. Most likely, the answer would have been between 10 ('20 - 100') and 5 ('500 - 1000').

Most users have some scenarios where one to twenty timesteps are sufficient. While a total of 53 answers were recorded, thirteen participants chose only one option. Seven thereof chose the first option and four the second. The other two participants used more than a thousand



(a) Estimated number of files per simulation run

(b) Estimated average file size

Figure 5.15.: Survey results (QG 1) for the typical count and size of self-describing data files. Answers provided via the freetext option are marked with an asterisk.

or more than five thousand steps. So, they have particular and continuous circumstances for their work. On the other hand, three participants use anything between one and over a hundred thousand or one, even over a million timesteps. This is an enormous variance.

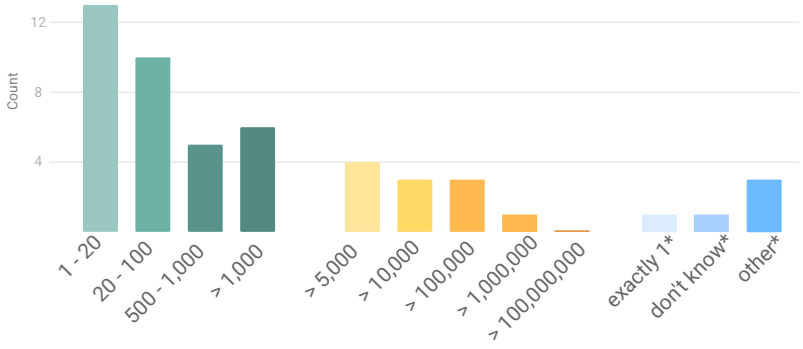


Figure 5.16.: Survey results (QG 1) for the number of used timesteps per file. Numbers between 1 and 5000 are coloured green whereas numbers ranging from 5000 to 100,000,000 are coloured yellow to orange. Answers given through the free text field are coloured in blue and marked with an asterisk.

5.3. Design and Implementation

In the following, the resulting requirements from the analysis in the last section are gathered. They are then compared to the features offered by the different database technologies. Then, the candidate to work as a basis for the database engine is chosen. The decisive criteria are named and to why other options were not suitable.

5.3.1. Chosing a Suitable Candidate

After the detailed data access analysis in the last section, here is the decision to use a relational database. In short, the access patterns are just too diverse to build a data warehouse for it. Also, a data lake is not required because the vast format support is not needed due to the focus on self-describing data formats in this thesis. As all features come with a price, it is better to use a solution that focuses on the essential aspects.

Summary of Insights and Requirements

Here all the gathered insights from the previous sections are put into requirements for the DB engine.

- **Data access**:
  - Various access patterns: The most important insight from the previous access analysis is the vast range of potential access patterns. The six access patterns to only one variable alone are, in part, the complete opposites of each other.
  - Design must be general purpose: It is not sufficient to offer only good read performance. The writing performance still has to be acceptable. Otherwise, the data cannot be written.
  - Data layouts can be non-trivial: The analysis showed that anticipating access patterns from applications employing dynamic load balancing is not easy.
- **Survey results**:
  - Support for timesteps: Most participants use some kind of step concept. These may be steps as in ADIOS or different custom time intervals.
  - File size: The typical file size lies in the range of MB and GB. No support for enormous file sizes is required.
  - Hierarchy depth: Most participants employ a very flat namespace. The typical hierarchy depth lies between 1 to 2.
  - Number of used variables: The most common number of variables is 1 to 250. The same goes for attributes.
  - File usage: The way participants use files to map concepts varies a lot. There is no clear trend. So supporting multiple approaches seems necessary. Typical ones use a file per variable/dataset or per timesteps showing that data layouts described in the data access analysis are not necessarily what users employ.

- **Other requirements**:
  - Complex queries: Post-processing will require the support of complex queries.
  - Dynamic table design: To support custom metadata, having the possibility for a dynamic table design would be great.
  - Distribution: For future work, the option to distribute the database across different nodes would be useful.
- **Software requirements**:
  - Open source: In order to promote open research, an open source license is required.
  - Language: To reduce the friction when setting up JULEA on a supercomputer, having a C/C++ API or at least the option for C/C++ bindings is vital.

Choice and Reasoning Behind

As discussed in Section 5.1.3, wide-column stores work great for columnar access. Unfortunately, the access analysis and the user survey revealed that scientific applications have an extensive range of access patterns. NoSQL models would offer support for various requirements and potentially great performance for a few scenarios. However, the changing access patterns of HPC applications essentially eliminate this type of database as an option because their database design is built to perform well only for specific scenarios. During the technology review, it also became clear that Cassandra no longer supports the dynamic schema definition. Furthermore, ScyllaDB instead of Cassandra would have been used if the choice fell on wide-column stores because it is a popular reimplementation of Cassandra in C++ instead of Java promising better performance. Yet, ScyllaDB never offered this feature. Another aspect that is not in favour of wide-column stores is the evaluation by Lofstead et al., ultimately giving up on the attempt to switch the relational databases in EMPRESS [Lofstead et al., 2019].

Besides the access patterns, another vital point is that the write performance has to be good not to worsen the I/O bottleneck further. Thus, focusing simply on great reading performance is not feasible. Nonetheless, the evaluation was not in vain. For example, aggregations similar to the OLAP cubes seem interesting for the DAI interface.

In summary, the various access patterns mean that a general-purpose system is required when the applications should not be drastically restricted in their I/O behaviour and data design. As self-describing data formats do not limit the use cases, the support has to be broadly based. Therefore, support for complex queries is also vital. Thus, the relational database was chosen for the database engine. It is a robust and well-tried approach. As a result of the technology evaluation, the schema definition will lean towards the denormalised approach of data warehouses, as this makes querying the data easier.

While data lakes are an interesting concept, they do not fit the requirements. They come with several features that are unnecessary at the moment or in the near future. The types of data sources are limited by the scope of the thesis to self-describing data formats. Therefore, combining various data forms like log files or spreadsheets is not required. However, they could work well for a more generalised approach that does not focus on a category of formats, as will be discussed in Section 8.2.

5.3.2. Mapping BP-Formats to Relational Database

In the following, the mapping of the BP formats to the relational database is described.

Data Granularity As previously discussed, the choice was made to store information at the block level. This may come with a certain overhead. On the other hand, the analysis may require this level of detail. As it is the finest level that ADIOS2 offers, it will be used for this engine as well. This also makes data storage easier. Each block can be uniquely addressed.

Using the block granularity also means that working on multi-dimensional values is avoided for most parts. Thus, the database functionality can be used directly. Then, data is not a blob, thereby avoiding the problems that database systems can have with scientific data.

Database Schema The decision to work on single statistical values and not arrays by working at the block level has some impact on the schema creation. Now, the extrema and other data characteristics cannot be represented by a single field in the schema because the data type is not a blob capable of storing any type of data. For example, storing the minimum of an integer requires a different field than storing the minimum of a double. When working at the step level or the variable arrays, i.e. blobs would have been used to store the minima regardless of the data type. As explained before, the file metadata is managed at the block level to allow fine-grained access and to reduce the data size that needs to be accessed. In Listing 5.7 the function is shown that sets the respective fields for the database schema according to the data types. Note that different data types are handled in the same field (Lines 4 to 14). This is because JULEA offers abstractions from the actual database technologies and therefore needs to provide a generalised set of types supported by most database technologies. They can be found in Listing 5.1 as part of the new JULEA database client and backend.

```
1  void JuleaInteraction::SetMinMaxValueFields(std::string *min, std::string *max,
2  std::string *value, std::string *mean, std::string *sum, const adios2::DataType varType){
3    switch (varType){
4      case adios2::DataType::Int8:
5      case adios2::DataType::UInt8:
6        ...
7      case adios2::DataType::Int32:
8        *min = "min_sint32";
9        ...
10       *sum = "sum_sint32";
11       break;
12     case adios2::DataType::Double:
13       *min = "min_float64";
14       ...
15       break;
16     case adios2::DataType::String:
17       *value = "value_sint32";
18       break;
19     case adios2::DataType::Compound:
20       std::cout << "Compound variables not supported";
21       break;
22   }
23 }
```

Listing 5.7: Setting the field names of the type-related columns for the database schema. To avoid storing opaque blobs in the database, a field for every data type is required.

Data characteristics like extrema do not make sense for every data type. For example, no minimum or maximum is stored for strings as it is unclear how this would be defined or if it would even be helpful. Another detail shown in Listing 5.7 is the value fields in Lines 11 and 21. As ADIOS2 supports single values and treats them as part of the file metadata, this approach is also continued for the JULEA engines. Therefore, fields exist to store a single value according to data type.

Variable Table In the following, the variable table is described. The schema is shown in Table5.1. It holds all file metadata that is required at the variable level. These are the same as mentioned in the key-value chapter 4. First, access patterns are managed. Afterwards, the shapeID and the dimensions follow. As previously mentioned, the ADIOS2 data types are mapped to the data types defined by the database client and backend in JULEA. Therefore, the exact data type has to be stored to allow exporting the accurate data that was stored without any loss. The block array stores the blockIDs for the respective steps. Additionally, all tables in JULEA have an ID which is automatically generated by the database itself.

File Name	**IsConstantDims**	**IsReadAsJoined**	**IsReadAsLocalValue**	**IsRandomAccess**
String	Integer	Integer	Integer	Integer
IsValue	**ShapeID**	**TypeInt**	**ShapeSize**	**Shape**
Integer	Integer	Integer	Integer	BLOB
StartSize	**Start**	**CountSize**	**Count**	**NumberSteps**
Integer	BLOB	Integer	BLOB	Integer
blockArray	**Min**	**Max**		
BLOB	Double	Double		

Table 5.1.: Table schema for the variable metadata. The field (column) names (bold) are printed above the respective field type. They are separated by the light-grey line. Due to the table's width line breaks are necessary. They are indicated by the doubled horizontal line.

Block Tables The information stored for the blocks is shown in Table 5.2. Each block is uniquely identified by the combination of the namespace, file name, variable name, step and blockID. This is not ideal, especially for a read-intensive application. Therefore, an index is built for every mentioned feature. This covers most calls from the engine to the database because they all work on a single variable in a file. Nonetheless, the index is also built for steps and blockIDs because of the functionality required by the DAI. Furthermore, the DAI will offer functions to retrieve the database's autogenerated `_id` to access a specific entry. This way, the lengthy string comparisons can be avoided. The same effect could also be achieved for the engine when the entryID would be cached in the engine. However, caching introduces the option for inconsistencies and performance bottlenecks when done inefficiently. As it is

Namespace	File Name	Var Name	Step	Block	ShapeSize
String	String	String	Integer	Integer	Integer
Shape	**StartSize**	**Start**	**CountSize**	**Count**	**MemStartSize**
BLOB	Integer	BLOB	Integer	BLOB	Integer
MemStart	**MemCountSize**	**MemCount**	**isValue**	**Min**	**Max**
BLOB	Integer	BLOB	Boolean	Double	Double

Table 5.2.: Table schema to store the variable block metadata. The field (column) names (bold) are printed above the respective field type. They are separated by the light-grey line. Due to the table's width line breaks are necessary. They are indicated by the doubled horizontal line.

mainly an optimisation technique, it has not been implemented for the engine. This could, of course, be done in the future. The actual index concept used is determined by the used database technology, e.g., SQLite or MariaDB.

The computation of the mean value is not part of ADIOS2 but is added to highlight the gained flexibility over the SDDFs. More details about the potential of derived metadata are provided in Section 5.3.3. This custom metadata can only be accessed through the JULEA client. In theory, it could be added to the BP engines. However, changes to the engines would require significant changes in the file format. When exporting data from JULEA to BP files, any custom metadata will be lost.

```
1 template <class T>
2 void PutVariableDataToJulea(core::Variable<T> &variable, const T *data, const std::string
      ↪ Namespace, const std::string file, size_t step, uint32_t ID, bool isKV) const;
3
4 template <class T>
5 void GetVariableDataFromJulea(core::Variable<T> &variable, T *data, const std::string
      ↪ project, const std::string file, size_t offset, long unsigned int dataSize, const
      ↪ size_t step, uint32_t ID, bool isKV) const;
```

Listing 5.8: JuleaInteraction function to write and read the variable data. Both engines use the same functions. The only difference is, that the key-value engine does not have entryIDs as these are the auto-generated IDs of the database. Therefore, the project namespace, file name, step number need to be passed as well because they are used to address the objects from the key-value engine.

Refactoring Both the KV engine and the DB engine store the data in the same way. So, to avoid code duplication of the complete data writing/reading process, the code was refactored. It is now structured like the code for the BP and HDF5 engines as shown in Figure 5.17.

The core engine implementation was kept in the engine directories. The data preparation and computation of additional statistics were moved into the `toolkit` namespace. In `toolkit/format` the serialiser and deserialiser for the different BP formats are managed. HDF5 and the JULEA communication are stored in `toolkit/interop` as they do not belong to

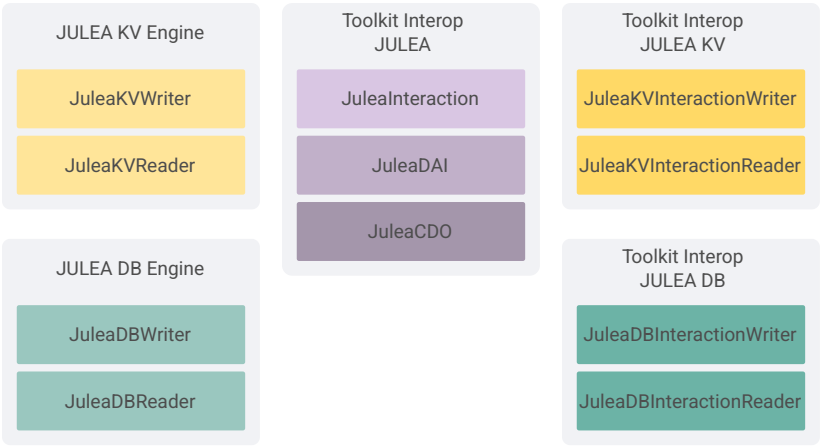


Figure 5.17.: JULEA related classes in ADIOS2: The classes belonging to the key-value engine are coloured in yellow. The classes of the database engine in teal and the classes that both engines use in lilac. The classes on the right encapsulate the interaction with the respective JULEA backends. As both engines store their data in the object store, they use the same functions.

the native BP format. The engines' communication is realised using the `JuleaInteraction` class shown in Listing 5.8.

Both engines implement their own derived classes, e.g. `JuleaDBInteractionWriter` and `JuleaDBInteractionReader` for the DB engine. In the `JuleaInteraction` class, the management of the data is handled for both engines. Since they work at the same granularity level, writing and reading data to and from the object store can be done in the same way for both engines.

5.3.3. Derived Metadata

As discussed at the beginning of this chapter, the format dissection enables leaving the format limitations behind. The data characteristics managed by ADIOS2 like local and global minima and maxima are an interesting feature. The assumption at this stage was that the computation of additional characteristics is beneficial, particularly for post-processing. This leads to the research question 2.2: What custom metadata is especially interesting? The terminology varies across different computer science communities. Whether the mean value is called additional data characteristics, derived metadata or new data is a bit of philosophical debate. For the sake of simplicity and consistency with ADIOS2, all information that is computed or derived from data will be counted as file metadata like the extrema in ADIOS2. In the following, potentially useful metadata classes will be categorised and how they can be managed in a relational database. Possible solutions to storing dynamic metadata in static schemes will be discussed as part of the DAI design and implementation in Chapter 6.

- Aggregations:
  Aggregations are the most obvious type of custom metadata after studying ADIOS2 and OLAP cubes. Both temporal and spatial aggregations are of interest, e.g. the mean temperature for Hamburg or the total precipitation for 2022.
- Application/Domain-specific:
  Besides general operations, application or domain-specific metadata could be very useful. In studying climate research, one tool came up often, i.e. climate data operators (CDO). While they also allow for computing aggregations, the focus here will be on characteristics like the climate indices, e.g. frost days, i.e. the number of days per year where the minimum temperature is below 0°C.
- Tagging:
  Another aspect of custom metadata allows the user to specify areas of interest. This will be done through offering tagging.
- Derived information: Other metadata of interest can be things like coordinates or dates instead of the number of steps

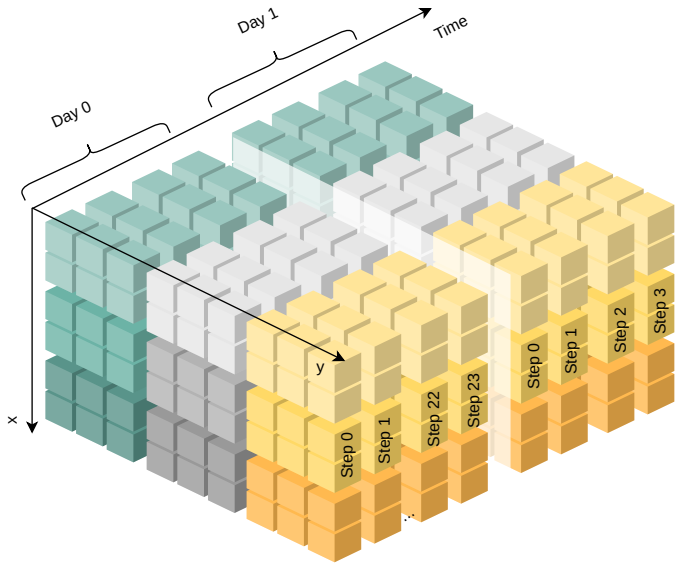


Figure 5.18.: ADIOS2 Variable cube aggregation of several time steps into a day.

Aggregations and Derived Information Aggregations over time are crucial for climate research. In Figure 5.18 a simple example is shown. Given a global 2D array with the dimensions x and y, several steps can be combined to compute statistics, e.g. the mean temperature over a day. Another option is to aggregate variables locally, e.g. compare the temperatures

of Europe to South America. Aggregations for other variables like the precipitation would possibly be to compute the sum of all precipitation.

Besides, additional convenience features can make the life of users easier, such as deriving the date from the number of steps. So, instead of having to pass '744' as the number of steps to read, they could specify that they want to retrieve the temperature for August. This would also reduce errors with leap years and changing days per month in general. While this type of information is particularly useful for climate research, particle physics could also benefit from further derived data. However, an option is required to specify what a step represents.

Chapter Summary

This chapter presented how the format dissection can be advanced. After using key-value stores in the previous chapter, the next logical step that enabled more complex queries was to use a relational database. As JULEA did not support them, a novel JULEA client and backend were developed. They were used for the second HDF5 VOL plugin that employs an entity-relationship data model. The implementation was relatively straightforward as the different objects were mapped to corresponding tables. However, the question arose whether more recent developments in database research would not be better suited. Therefore, the literature was studied. At first glance, data warehousing seemed like a promising candidate. To be able to decide whether this was accurate, more information about the data access patterns was required. Therefore, common data layouts and the results of the first part of the user survey are studied. This revealed that the access patterns vary considerably, which makes designing a sensible data warehouse data model difficult, if not impossible, as they are usually optimised for specific tasks. As a result, the ADIOS2 database engine also uses a relational database like the second VOL plugin. In contrast to the data model of the latter, the engine's data model is inspired by data warehousing. For example, the tables are not normalised. After discussing the design decisions and the implementation, custom metadata categories are derived that form the basis of the DAI pre-computation.

6. Data Analysis Interface

This chapter covers research questions 3.1 and 3.2:

- *What functionality is required for the DAI interface?*
- *How can this approach be generalised?*

To answer these questions, the scientific is specified to which the DAI is tailored. Insights into climate analysis are given to illustrate where the DAI is beneficial. In combination with the results of the second part of the user survey, requirements and use cases are derived. Afterwards, the implementation of the DAI's functionality, that is, pre-computation, tagging, reading and offering domain-specific functionality, are presented. This leads to a discussion of possible routes to generalise the developed prototype to other file formats and different scientific communities and their requirements. The challenges that arise are highlighted as well. The chapter is split into the following sections:

- *Section 6.1: covers the requirements and use cases for the interface as well as the design*
- *Section 6.2: presents the implementation of pre-computation, tagging, reading and domain-specific functionality*
- *Section 6.3 weighs different approaches to generalise the DAI.*

This chapter is dedicated to answering research question 3, that is, *How to design a file metadata interface?*. The interface is called Data Analysis Interface, short DAI. Research question 3 is split further into research question 3.1, that is, *What functionality is required for the DAI?* and 3.2, that is *How can this approach be generalised?*. First, the general approach to interface design is discussed. Climate analysis is used as a motivation and origin for specific post-processing tasks that the DAI should address. In particular, climate data operators (CDOs) are studied. Then, the second part of the user survey is presented, focusing on the users' interface preferences. The interface's requirements and functionality are derived and explained, answering research question 3.1. Afterwards, the implementation is explained in detail, starting with the general concepts used in the DAI. Following is the pre-computation of custom metadata, tagging of such, reading and querying the metadata, as well as a proposal on how domain-specific functionality can be implemented. Finally, research question 3.2 and the options to generalise this approach for different data formats but also various scientific areas are discussed. A special focus lies in the numerous ways that data can be modelled even within a single format and what challenges this brings. Then, approaches to unify the interface are discussed and balanced against each other.

6.1. Interface Design

In the following, some aspects of the design space are discussed to get a general idea of the possible shape that such a DAI could take. As mentioned throughout the previous chapters, the format dissection enables the computation and storage of new file metadata. So, there needs to be a part of the interface dedicated to declaring computations leading to new data that needs to be stored. Also, an interface to access this new information is required as well. This can be either done by adapting and extending the I/O library interface or by introducing a new and separate interface. When this interface is added to the I/O library, either the existing functionality cannot be touched to avoid API breaking, or the new interface has some restrictions in terms of design but also the actual implementation inside the core of the I/O library. All API changes must work with the existing concepts and native format requirements. The other option is to build another interface on the data. When using JULEA, this means adding a user-friendly interface on top of the database interface.

API Design Matters When tackling the design of a new interface, it seemed sensible to consult with the literature on general design recommendations. There is a wide variety of philosophies, and the focus can be on numerous aspects. However, the following steps for the design process came up often and will be considered in the next sections [Bloch, 2006, Henning, 2007].

1. Work out requirements: To provide a helpful interface to users, it is necessary to figure out the constraints and requirements. For example, when the prevalent programming languages are not taken into account, then the API might work well in theory to address the posed use cases. However, if there are no suitable language bindings, users cannot use the new interface within their existing ecosystem, ultimately making the design a failure.

2. Derive suitable use-cases: Besides defining the requirements, a crucial point of designing a novel interface is to gather possible use-cases and have a clear vision of what the interface is supposed to do. The use cases will accompany the complete process. So, getting them right is important.

3. Build application before API implementation: A very interesting recommendation is to implement the use cases, i.e. the final application, before the API implementation. This might not feel as natural as starting with the implementation but comes with many benefits. Using the interface prototype inside an application makes it much easier to put oneself in the users' shoes. The usability and intuitive interface become the main focus, forcing the designer to trace the complete thought process. As a result, missing parameters or inconvenient functions become clearer.

6.1.1. Climate Analysis

A specific problem statement is required in order to propose an interface prototype for the DAI within the scope of the thesis. Throughout the chapters, examples and problems were often

derived from the workflows observed at the German Climate Computing Center (DKRZ). As a result, climate research was an apparent choice for the topic of the case study. The following will briefly introduce the Intergovernmental Panel on Climate Change (IPCC) and, more specifically, Coupled Model Intercomparison Project (CMIP6). Typical data analysis will then be studied to gather suitable use cases and requirements for the DAI prototype.

CMIP6

The IPCC is an international committee of the united nations concerned with climate change, its causes and prognoses. The results are published as assessment reports and frequently discussed. Widely known developments often come in the form of visualisations as presented in Figure 6.1. Here, different scenarios are compared regarding the change in the global mean temperature. The scenarios range from optimistic (SSP1-2.6) to pessimistic (SSP5-8.5) prognoses. The prominent 1.5°C and 2°C goals are indicated as well [IPCC, 2022]. These are the estimated marks whereafter crossing means irreversible climate effects.

The presented data comes from the simulations run as part of CMIP6 [Eyring et al., 2016, Tebaldi et al., 2021]. Here, different climate models are evaluated in comparison to each other.

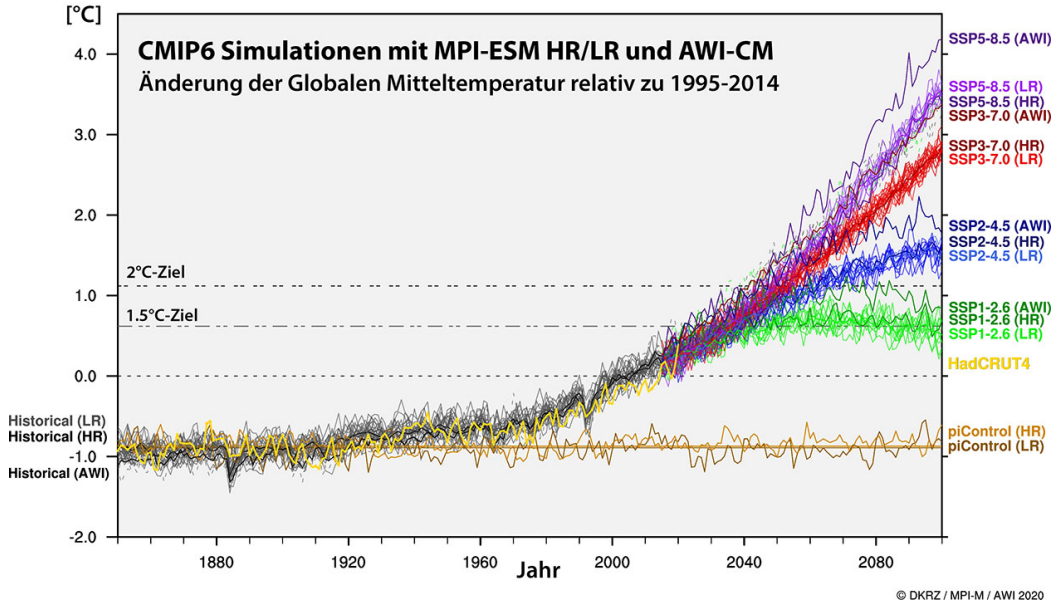


Figure 6.1.: CMPI6-Simulations - Change of the global mean temperature (relative to 1995 - 2014): Different ensemble simulations (MPI-ESM HR/LR and AWI-CM) for the air temperature (2 metres) for the past (grey curves) and for different future scenarios (purple, red, blue, green curves) are shown in comparison to the observation data (yellow) and control runs (brown). The control runs are used to examine the undisturbed climate. The different curves of the same colour are the result of running the same simulation with different starting conditions. Abbreviations: *MPI-ESM*: Max Plack Institute - Earth System Model [Mauritsen et al., 2019], *AWI-CM*: Alfred Wegener Institute for Polar and Marine Research - Climate Model [Semmler et al., 2020], *LR*: Low Resolution, *HR*: Higher Resolution, *SSP*: Shared Socioeconomic Pathways

The following discussions focus only on the most prominent examples, discussed most frequently. It is by no means a statement towards the immensely large field of climate research and meteorology. Instead, it is a subselection of typical operations that are widely known and thus easy to follow.

In terms of performed analysis, there are some aspects to note. First, it is very common for such analyses to be working on mean values, e.g. the global mean temperature. Often, very few variables, typically the temperature and precipitation are studied in great detail. Furthermore, different ensemble runs are compared with an overarching question.

Another visualisation of CMIP6 simulation results is shown in Figure 6.2. It illustrates how the global mean temperature can be broken down, both in terms of spatial and temporal dimensions. Here, the global mean temperature for a specific is computed for each country.

CDO Functionality

The climate data operators are studied next to get more insights into the detailed post-processing operations. The climate data operators (CDO) are a command line tool offering various operations on climate data [Schulzweida, 2022]. These range from file operations such as copying or splitting over arithmetic functions to the climate indices that capture relevant events for the temperature and precipitation development. Modifications such as setting the time or the grid info are also numerous. The arithmetic functions contain operations to evaluate an expression or compare simulation data to constants. Additionally, CDOs offer the computation of correlation and covariance in the grid space or over time, as well as regressions to find trends in the data. Interpolation and remapping, as well as spectral and wind transformation, are common as well.

For this thesis, a subset of operators was chosen: various selection operations, CDO statistics and the climate indices. All of them are explained in more detail in the following.

CDO Selections The CDOs support different ways to specify and retrieve data. The choice for this thesis is motivated by the typical access patterns mentioned in the data access analysis. Thus, the selection of fields, timesteps or a subset of a variable is supported. Furthermore, the CDOs offer conditional selections of fields as well as comparing variables with constants.

CDO Statistics The statistics offered by the CDOs contain various ranges, e.g. values over an entire ensemble, a field, and a time range. Thus, support for different granularity levels is added to the list of requirements. For the thesis, the following selection of statistical values was chosen:

- Extrema: like minima and maxima
- Mean: Note that the mean still works if there are missing values in contrast to the average: e.g. mean(1,2,3,miss) = 2; avg(1,2,3,miss) = miss. Also, computing the mean of several mean values is only the same as the means of all when the cardinality is the same for every submean. Here, the assumption becomes important that global arrays with a regular shape are used.

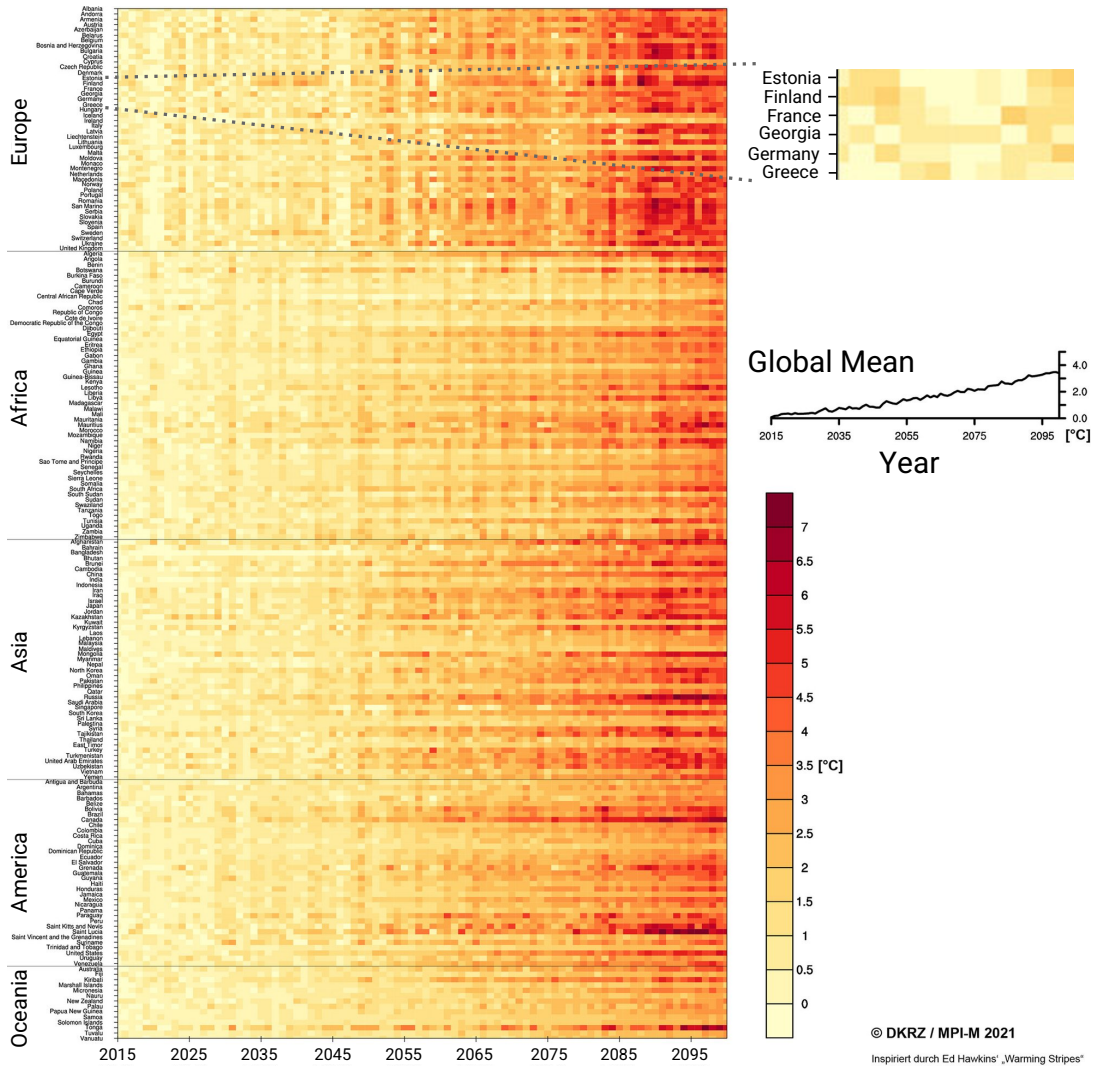


Figure 6.2.: CMIP6 Simulations with MPI-ESM HR - SSP585 (pessimistic scenario): Projected change in the mean surface air temperature relative to 1995 - 2014 for each country grouped by continents. The temperature scale ranges from red, indicating a large temperature increase by up to 7°C, to light yellow marking no temperature change. At the top of the right, the first ten years are shown for the countries Estonia to Greece. Below, the global mean is plotted over the same time period as the large plot. Abbreviations: *MPI-ESM*: Max Plack Institute - Earth System Model [Mauritsen et al., 2019], *HR*: Higher Resolution, *SSP*: Shared Socioeconomic Pathways Reprint (partially redrawn) from [DKRZ/MPI, 2021] © DKRZ / MPI [2022] (CC BY-NC-ND 4.0)

- Sum: for flux variables like the precipitation, calculating sums might make more sense than, e.g. a simple hourly maximum.
- Variance and standard deviation: to observe the data distribution. The computation of the variance of variances is required to evaluate the behaviour over, e.g. all blocks of a step. It

is shown in Equation 6.1[1]. Again the cardinality of the sub-samples has to be the same. The overall variance builds on the variance (Vj) and the mean (Ej) of each sub-sample.

$$Var(X_1, ..., X_{kg}) = \frac{k-1}{gk-1}(\sum_{j=1}^{g} V_j + \frac{k(g-1)}{k-1} Var(E_j)) \tag{6.1}$$

Climate Indices While the previous CDO functions can be used for various scientific fields, the climate indices are specific to the climate domain. They are part of the European Climate Assessment (ECA) and are used to evaluate temperature and precipitation extremes and their variability. The indices are computed for a time period, typically years. Some of the indices compare values to a reference period. For simplicity, the indices selected for the implementation in this thesis do not require a reference period. To define the temperature-related climate indices, the following variables are required:

- TN: daily minimum temperature
  Let TN_{ij} be the daily minimum temperature on day *i* in year *j*.
- TX: daily maximum temperature
- TG: daily mean temperature

Besides the presented indices, versions exist that track consecutive occurrences, e.g., days with a minimum temperature over 25°C. Thus, they capture the duration of extreme periods.

- `fd` = Frost days index per time period: Annual count of days when $TN_{ij} < 0$ °C
- `su` = Summer days index: # days when $TX_{ij} > 25$ °C
- `id` = Icing days index: # days when $TX_{ij} < 0$ °C
- `tr` = Tropical nights index: # days when $TN_{ij} > 25$ °C

Similar to temperature-related indices, precipitation indices are defined for specific thresholds. In contrast to them, precipitation indices work on the *daily precipitation sum*, not the daily maximum, which would be the highest hourly precipitation. *heavy precipitation days* (`r10mm`) have a summed daily precipitation of over 10 mm per day and 20mm for *very heavy precipitation days* (`r20mm`). There is also the generic operator *precipitation days* (`pd`) allowing to specify a custom threshold.

6.1.2. Survey on Interface Preferences

In the following research question 3.1, *What functionality is required for the DAI?*, remains the focus. In order to develop a user-friendly API, it is vital to know what the users like to use. Also, further information about the typical post-processing is required to decide on the interface placement. Thus, the user survey had a second part, Question Group 2. It focused on the used programming languages, interface preferences and common post-processing operations.

[1]The equation for unequal sub-samples and the derivation can be found at `https://stats.stackexchange.com/questions/10441/how-to-calculate-the-variance-of-a-partition-of-variables`

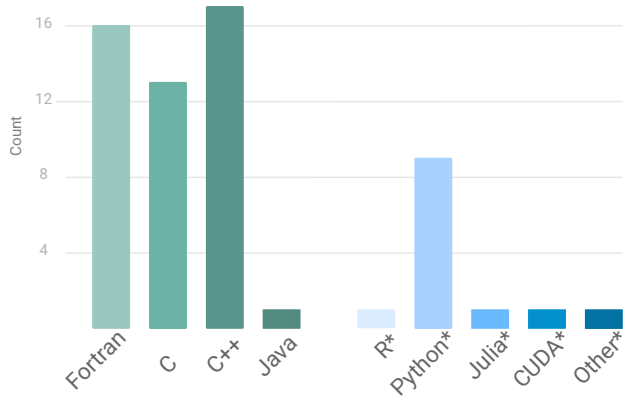


Figure 6.3.: Survey results for the programming languages used for simulations. Pre-made answers are indicated in green. Answers given using the free text fields are coloured in blue and marked with an asterisk for grey-scale prints.

Programming Languages used for Simulations The majority of simulations are written in Fortran (16) and C/C++ (13/17), as shown in Figure 6.3. All five survey participants that use ADIOS2 run C++ applications, and three of them Fortran and C as well. So, for the DAI parts facing the simulation, the interface must either be implemented in one of these languages or at least provide bindings. When offering a C++-compatible C interface, for example, Fortran and Python bindings can be generated. Such an approach would cover 55 of the 60 answers and target a large audience. The same participant uses both Java and Julia.

Programming Languages used for Post-Processing The answers presented in Figure 6.4 show that Python is the most used language for post-processing and analysis (21 of 68), followed by C/C++ (both 10). Also, all ADIOS2 users used Python for post-processing. Seven participants used Matlab and R. Besides the pre-made answers, IDL, Paraview, Bash CUDA and Julia were used by a few participants.

Most participants do not want to use another language, as shown in Figure 6.4. If anything should change, the choice is Rust(4). So, most people using Python seem to be generally satisfied. However, they could also be dissatisfied but still prefer Python over solutions they consider to be worse. Also, the results do not show whether participants could not name a specific alternative. Thus, the results are uncertain and should not be taken as proof that the current situation has no problems. Nonetheless, given the results in Figure 6.4, Python seems to be an important candidate to keep in mind when designing the DAI later.

Post-Processing Tools and Libraries The next questions examined tools and libraries used for post-processing to understand how users access their data and what they do with it. The answers in Figure 6.5 indicate that predominantly Python packages are used for post-processing. Still, this impression can be influenced by the choice of pre-made options that

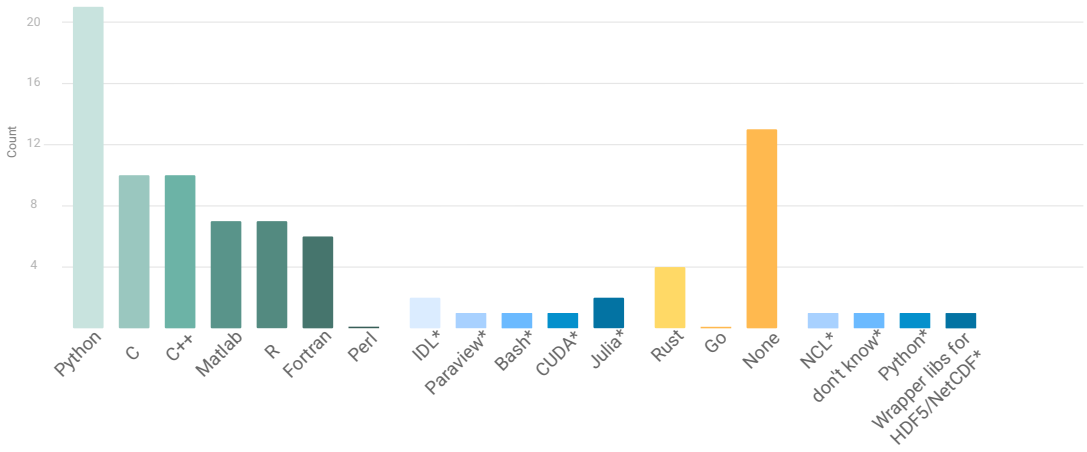


Figure 6.4.: Survey results for the programming languages used for post-processing and analysis. Pre-made answers are indicated in green. Answers given using the free text fields are coloured in blue and marked with an asterisk for grey-scale prints.

are primarily Python-based. The most used libraries or tools are Pandas (11 of 53), Dask (10) and CDOs (8). Most application users reached by the participation call for the survey are probably in the general area of earth system science. Therefore, the survey setup does not allow conclusions that CDOs are as widely used as Pandas or Dask.

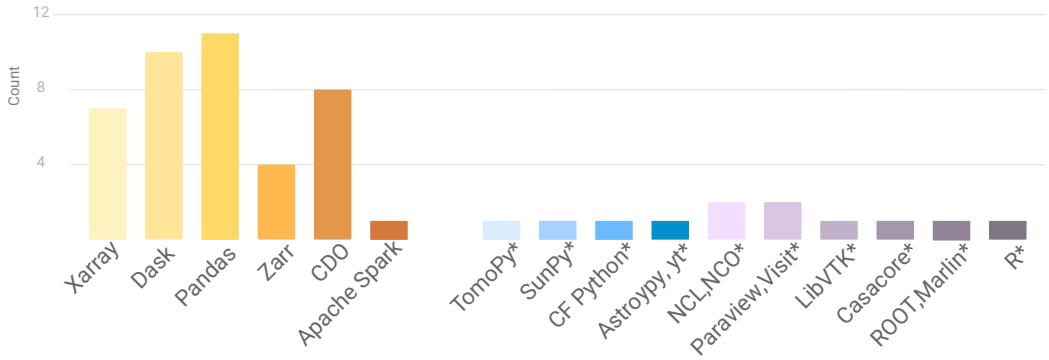


Figure 6.5.: Survey results for the tools and libraries used for post-processing and analysis. Pre-made answers are indicated by the yellow and orange colours. Answers given using the free text fields are coloured in blue and lilac and are marked with an asterix for grey-scale prints.

In Figure 6.6, the most common post-processing operations are presented. These were examined to provide further direction on what type of operation to support the DAI precomputation. The survey results illustrated in Figure 6.6 show that reductions like computing averages, extrema, sums and variances are most common.

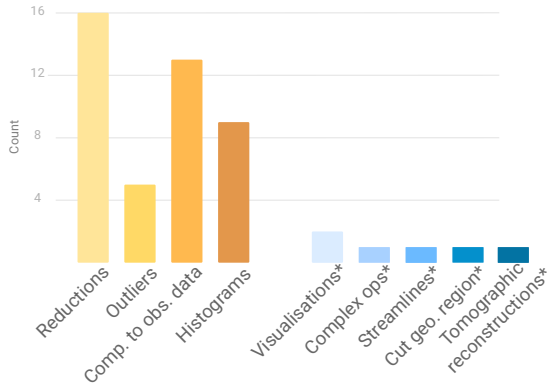


Figure 6.6.: Typical post-processing operations

Interface Preferences Lastly, more knowledge about users was required to understand which direction to take the DAI in terms of design. Therefore, the interface preferences were examined. There were two options for addressing this question; either provide different early-state DAI prototypes or evaluate common interfaces. To convey the new functionality of a prototype in the form of interface documentation is often difficult. On the other hand, large code listings were less likely to be read by the survey participants. Therefore, the second route was taken to gather preferences for established interfaces. The different interfaces are grouped to keep the number of presented candidates small.

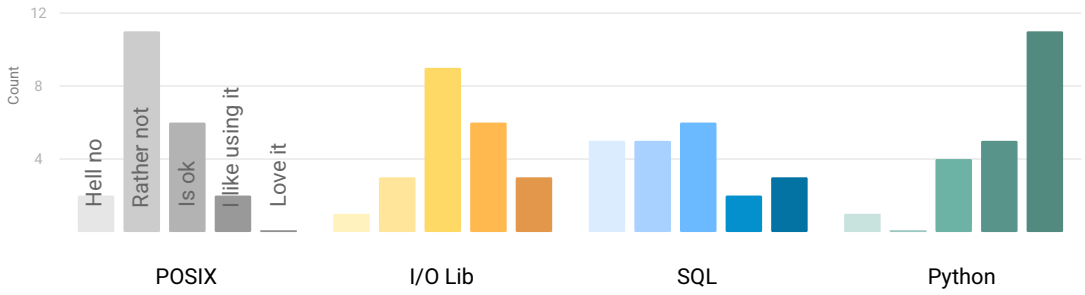


Figure 6.7.: Interface preference for low-level interfaces like POSIX and MPI I/O, I/O-libraries like HDF5 and ADIOS2, database interface using something like SQL, Python interfaces. The scale from left to right is: Hell no, Rather not, Is ok, I like using it, Love it

First, POSIX was chosen for the low-level system interfaces as it is widely known. In comparison to MPI, no notion of parallelism is required making the POSIX examples simpler. Second, I/O libraries are evaluated using ADIOS2 as a representative. Third, database interfaces were studied, focusing on SQL. Fourth, post-processing interfaces were also relevant for this survey. So, the Python package Dask is chosen. For all interfaces, short example codes were shown that enabled a direct comparison. It also allowed users to vote even on unfamiliar

interfaces. The preferences are shown in Figure 6.8. Six answer options were given. These range from apparent loathing ('Hell no') to explicit appreciation ('Love it').

To allow a conclusion, typical statistics like the mean, standard deviation, variance and median were examined. These are shown in Table 6.1. Note that one person only answered the question regarding the used I/O libraries. So, the results for the other options are computed using only 21 values. Also, given the uncertainties of the survey setup, e.g. the participant selection, these results should not be generalised to specific user groups or scientific fields. These results do not represent the preferences of computer scientists or application developers in general. Instead, they should be seen as what they are, a momentary opinion of the participants. Nonetheless, they can be used to motivate and inform the decisions made during the DAI design process.

Statistic	POSIX	I/O Lib	SQL	Python
$\bar{x}$	2.38	3.32	2.67	4.19
s	0.80	1.04	1.35	1.08
s^2	0.65	1.08	1.83	1.16

Table 6.1.: Mean ($\bar{x}$), standard deviation (s) and variance (s^2) for the interface preferences

To compute these statistics, the answer options were assigned a value from one to five from left to right. Python is the clear winner in terms of overall preference with a mean value of 4.19, while POSIX is the least liked interface reaching only a rating of 2.38. This is also interesting because POSIX has the smallest variance (0.65). So, it seems the participants agree the most when evaluating POSIX. The opinions are most diverse for SQL, with a variance of 1.8. Only the I/O libraries and Python reached an average rating of at least 'is ok', whereas POSIX and SQL have an average rating of 'rather not'.

6.1.3. Requirements and Use Cases

In the following, the requirements are gathered that can be derived from examining CMIP6, selecting the CDO subset and from the results of the user survey. Also, several use cases are defined accordingly. They are grouped into four main categories. First, the general requirements regarding the implementation are presented. Second, all requirements regarding the precomputation of custom metadata are summarised. Third, the requirements for reading the original and custom file metadata are presented. Forth, requirements concerning the tagging of data are stated. Tagging is added to the list of desirable features because of the promising results in EMPRESS2. Finally, the DAI placement in the data life cycle is discussed.

Implementation Requirements

In the following, the respective requirements derived for the implementation are explained.

- **Offer C/C++ interface**: The survey revealed in Figure 6.4 that most simulations are written in either C or C++.

- **Not require schema knowledge**: As discussed in Chapter 3 Zhang et al. point out that one of the major problems of mapping data formats to databases is that the users need to know how the database schema looks like to access the data [Zhang et al., 2019]. So, comfortable access must be offered without prior knowledge about the storage backends.
- **Not POSIX, not SQL**: The evaluation of the user preferences showed that they tend not to like POSIX or SQL. Therefore, the interface should lean more toward the higher-level APIs in the I/O libraries and Python.
- **Not replace I/O library**: While exploring the design space for the DAI, it is important to keep the initial position in mind. The goal was to limit the application changes as much as possible to keep the prior investment in their development. Therefore, the I/O library will not be replaced. Also, there is no need to reinvent the wheel and reimplement all the functionality already offered by the library. For example, complex access patterns are well supported by the native ADIOS2 interface. In Figure, an example is given to illustrate data access that is not aligned with the block borders. Reading this selection is possible with both JULEA engines, but the data is accessed through the ADIOS2 interface. So, providing access to the data itself is not the focus of the DAI; access to the file metadata is.

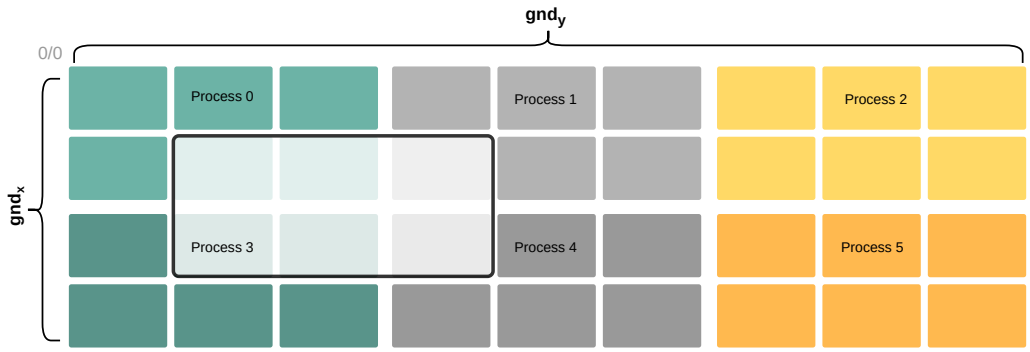


Figure 6.8.: ADIOS2 Variable: Unaligned data access to multiple but partial blocks. This global array was written by 6 processes (each indicated by a different colour). The global dimensions are gnd_x and gnd_y. Patterns like these are offered by the ADIOS2 interface already and will therefore not be the focus of the DAI.

Pre-Computation

As illustrated by the CMIP6 visualisations, post-processing and analysis often work on statistical values like means and other reductions like the sum or variance. The survey also showed that these are the most used operations in post-processing and analysis. Thus, the first requirement that can be derived is the need for *further data characteristics* besides the extrema already provided by the ADIOS2 formats. The offered additional statistics will be the subset

of selected CDOs previously mentioned because these are common operations regardless of the scientific domains. As depicted in Figures 6.2 and 6.1, these statistics can be sensible at different granularity levels like computing the global mean temperature for a year or the local mean temperature for a specific country. Therefore, another requirement is to *specify the granularity for the computation.* Note that not all statistics make sense for every physical variable, e.g. computing the sum for a temperature variable. If the interface has to guarantee that only useful statistics are computed, the physical variable characteristics must be specified. This will result in a complex interface and additional overhead for the user. In contrast, when the statistic is computed for every variable regardless of its use, the computation overhead and space are increased without offering more functionality to end users. Thus, the implementation must balance a simple interface logic and the computation of unnecessary statistics.

Use case 1: Compute additional statistics, e.g. mean temperature or daily precipitation sum.

Tagging

The second larger category of functionality is related to the concept of tagging. While EMPRESS and EMPRESS 2 suffer from different problems, the general idea is beneficial. Thus, another requirement for the DAI design is to offer *support for highlighting interesting areas.* For now, the user will have to specify what data is important explicitly. It will be future work to automate this procedure. For example, machine learning algorithms could be used to derive areas of interest and communicate these to the storage system directly without user interaction. As tagging data during the writing phase requires knowledge about analysis when the simulation is run, this approach will require a slight change in mindset. Currently, simulations and analysis are viewed as separate and independent from one another. However, taking a more workflow-oriented perspective is important to improve the overall performance from start to end. For example, knowing which data will be accessed soon can help avoid costly migration in the storage system. Tagging rare occurrences can then be used to inform data placement. Marked data can be stored on faster tiers as access is more likely. Furthermore, it could be used to adapt workflows in the case of such an event, as proposed by Foster et al. [Foster et al., 2017]. The support for the climate indices can also be realised by tagging the specific occurrences.

Use case 2: Mark specific areas of interest, e.g. tag all blocks where max. temperature > 25°C

Reading

Access extrema Querying and reading data make up the third category of functions. Following the CDO selections, accessing the extrema is important for post-processing and analysis. So, the data characteristics provided by the BP formats of ADIOs2, i.e. the minimum and maximum, need to be accessible individually. This requirement also lays the foundation for more advanced querying discussed later on. The respective use case also demonstrates that

the functionality already provided by ADIOS2 is kept and enhanced[2].

Use case 3: Access data characteristics, e.g. read minimum and maximum of a variable.

Range queries Supporting the fine-grained access to the data characteristics also enables queries to specific sub-areas by providing range queries. While this functionality directly builds upon the previous, another function will be required. So, a separate use case is defined for range queries.

Use case 4: Query the variable's data range, e.g. read all blocks where -42°C < T < 42°C.

Read custom file metadata Another requirement follows directly from the pre-computation of additional statistics. Any derived file metadata that has been precomputed must be made accessible as it cannot be accessed through the interface of the native I/O library. Thus, querying the custom file metadata makes up the next use case.

Use case 5: Query custom file metadata, e.g. read the highest mean block value.

Queries across different files Examining the CMIP6 visualisation made clear that another requirement is to perform *analysis across different files*. This is currently not very efficient because often, each file has to be opened, the file metadata has to be read, and then the respective variable needs to be read.[3] Using the database engine for ADIOS2, querying across files is a simple task as the data of different files is stored in the same tables. As a result, querying the files of different scenarios means accessing different entries in the same table. Note that the concept of a project namespace will be introduced in the actual implementation allowing more structuring. Therefore, different project namespaces will have separate variable and block tables.

Use case 5: Query across files, e.g. retrieve the mean temperature of all files

Complex queries CDOs offer the option to chain operators, which means feeding the result from one as input to another. It shows a need for combined and more complex operations, especially queries. So, another requirement that follows is to offer complex queries. While relational databases provide the option for such queries inherently, the survey has shown that the SQL interface is not incredibly popular. Therefore, it is sensible to think about a different interface. Ideally, hiding the database schema from the users makes the usage as comfortable and low-effort as possible. The actual data representation should not be a concern of the user.

Use case 6: Query more than one variable, e.g. read all blocks where T > 40° and P > 20 mm.

[2] ADIOS2 offers query functionality that is not mentioned in the documentation at all and is hardly commented. It seems to be a wrapper to read subblocks

[3] A short reminder for ADIOS2: The file metadata for a BP file is stored in a separate metadata file. Thus, not the actual BP files need to be read. Instead, the separate metadata files have to be accessed.

Domain-specific function To illustrate how domain-specific functionality can be included in the DAI, computing the selected climate indices is offered. To improve the comfort and convenience of users, dedicated functions to query them are required as well.

Use case 7: Retrieve climate indices, e.g. get the number of fd, ic, tr, su.[4]

DAI Placement in the Data Lifecycle

The data lifecycle is a common concept to visualise workflows in various settings. Though it is not clearly defined, it often relies on three major aspects, data generation, storage and analysis. As illustrated in Figure 6.9, the actual lifecycle is more complex as different workflows depend on one another. Also, there can be smaller feedback loops to tune the application parameters before the large ensemble runs.

When introducing a new interface, an important question was where it should be placed. To which area does it belong the most? Pre-computing information means that knowledge about the generation, the storage and especially the post-processing is required. So, pre-computing sits in between analysis and generation. It may be, in part, what is considered refinement. However, if the DAI functions are termed more in the direction of "interleaving computation with I/O", then the DAI is situated between generation and storage. When the focus lies on reducing the data access when analysing data, it is between storage and analysis. The respective places are highlighted in Figure 6.9.

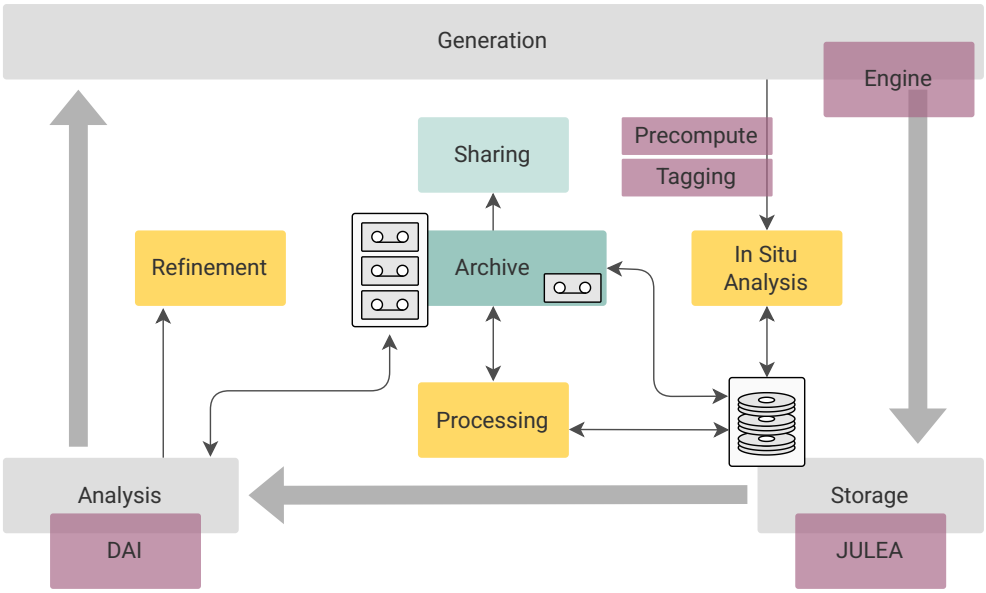


Figure 6.9.: Data Life Cycle and the leverage points (light purple) of the storage coupling including JULEA, engines and the DAI.

[4]Reminder: fd - frost days; ic - icing days; tr - tropical nights; su - summer days

In summary, the DAI may have to offer more than one interface because it can be situated at different places in the data life cycle. As previously discussed, the past efforts put into developing large-scale simulations must be kept. This means that the DAI needs mechanisms to work in different places. For example, the simulation-facing side should be kept as least invasive as possible. In the post-processing analysis, however, new concepts and a new interface can be introduced with less disruption.

6.2. Implementation

In the following, the implementation of the DAI is discussed, starting with general advice offered in the literature. The tips that stood out the most and seemed the most useful for the DAI and its estimated environment are shown below. While they are not groundbreakingly novel insights, they lay a solid foundation to reference throughout the rest of the implementation process [Bloch, 2006, Henning, 2007].

1. *You cannot please everyone. So, displease everyone equally.*
   This tip is especially important as the potential access patterns are so varied. The DAI aims to provide a general-purpose approach while demonstrating how more domain-specific functionality could be incorporated.

2. *Offer a sufficient range of functions, but…*

3. *When in doubt, leave it out.*
   The advice aims to reduce the functionality that is not necessary because adding new functions is easier than removing established functions.

4. *Keep the interface as minimal as possible.*
   Group related functions, for example, by using a suitable prefix. Therefore, all DAI functions start with `j_dai_` followed by a specific prefix. For example, all functions operating on steps have the prefix `j_dai_step`.

5. *APIs should be self-documenting. Names matter.*
   Ideally, the function and parameter names speak for themselves to avoid lengthy documentation. If it is not clear from the name what a function does, renaming should be considered.

6. *Employ the principle of least astonishment.*
   There should be no surprises about what a specific function does or what the parameters mean. Especially, hidden states of the program should be avoided. This advice influenced the realisation of the project namespace that was previously mentioned in the design section. More specifically, it helped guide the decision of whether the project namespace should be set once at the start of an application to avoid passing the parameter every time. However, it would have led to a hidden state and caused problems, for example, if users worked concurrently on more than one namespace.

7. *Consistency is key.*
   One aspect here is, for example, to provide consistent parameter ordering and use the same parameter names across functions. The naming is kept as similar as possible to make the DAI interface fit in with the other JULEA interfaces.

8. *Avoid limits on input sizes.*
   This tip may not be suitable for every environment. The size of a database entry directly depends on the length of strings, like the file and the variable name, as detailed in Equation 6.2.

9. *Do not sacrifice usability for efficiency*
   This rule is often violated by programmers in the systems community, especially beginners. Often readability is abandoned for supposed performance gains, for example, using the tertiary operator instead of a traditional if-statement. However, current compilers take care of many of these aspects that may have been real problems in the past.

While the DAI interface is not directly designed for the ADIOS2 interface and its engine concept, the following discussion of the implementation details will focus on the interaction of the DAI and the JULEA-DB engine. Regarding error handling, one sensible option is to follow the approach employed in JULEA. In JULEA, the GLib enables more convenient error handling. The last parameter of every user-facing function is of the type `GError`. Note that the error handling for the DAI has been omitted as of now. First, it makes the following code examples easier to read and provides a more straightforward interface. Second, the error handling in JULEA is planned to be overhauled. So, the error handling for the DAI can be added when it is updated throughout JULEA. Also, security is not taken into consideration for now. While it is an important aspect of a production-ready system, it is neither the focus of this thesis nor JULEA.

General Concepts

The implementation of the DAI parameters is explained in the following. To keep the number of functions as small as possible, aspects such as which statistic to compute or which comparison operator to use are passed as parameters. Another option would have been to implement dedicated functions, for example, `compute_mean`, `compute_min` or `compute_sum`. Instead only one function like `compute_statistic` is required. Working on strings in C is slightly tedious compared to other programming languages like Python. To avoid extensive string parsing whenever possible, the parameters are instead enumerations or short `enums`. In C and C++, an `enum` is a data type allowing a restricted set of integer values. The enums used in the DAI interface are explained in the following.

Statistics In Listing 6.1, the statistics that can be precomputed are shown. As discussed, the selection was made based on popular CDOs and the survey results. Assigning the specific values (base 2) to the different `JDAIStatistics` member enables to set of multiple options at once. It essentially creates a hand-made integer bitmask that makes it convenient to check which bits are set. An example is given in Line 8. Here, the computation of the mean value,

the sum and the variance are configured. This way of passing parameters or flags is common in C and often used for file system operations, for example, `open`. Therefore, it is assumed to be either a known concept to possible DAI users or easy to understand, given suitable documentation, including examples.

```
enum JDAIStatistics{
  J_DAI_STAT_MIN  = 1,         // minimum
  J_DAI_STAT_MAX  = 2,         // maximum
  J_DAI_STAT_MEAN = 4,         // mean
  J_DAI_STAT_SUM  = 8,         // sum
  J_DAI_STAT_VAR  = 16};       // variance

set_stat(..., J_DAI_STAT_MEAN | J_DAI_STAT_SUM | J_DAI_STAT_VAR);
```

Listing 6.1: DAI Interface - Statistics: The enum to specify the possible statistics that can be computed. Below an example is given, how multiple options can be passed for one parameter.

Comparison Operators The supported comparison operators are shown in Listing 6.2. The `JDAIOperator` members are selected based on the comparison operators already present in JULEA, for example, in the database client. For consistency, they are used for the DAI as well. Additionally, these are the most common operators and cover a wide variety of use cases. Given the feedback on interface preferences in the survey, the decision was made not to follow typical SQL query syntax. However, at the same time, inventing a new syntax to specify arbitrary queries is not a good idea either. Even regular expressions are not used consistently in different projects, even though they are well-defined [Davis et al., 2019]. So, the functionality required for complex tasks is split into multiple functions instead of defining a new query language. The results of one function can then be passed to the next function to solve more advanced tasks. At the moment, only one operator can be used per query. Therefore, the `JDAIOperator` members are not assigned specific values in contrast to the `JDAIStatistics` members.

```
enum JDAIOperator{
  J_DAI_OP_LT,           // < (less than)
  J_DAI_OP_EQ,           // = (equal)
  J_DAI_OP_GT,           // > (greater than)
  J_DAI_OP_NOT};         // ! (not)
```

Listing 6.2: DAI Interface - Operator: The enum to specify the possible operators that can be used in a query.

Granularity The last concept required for DAI parameters is the access granularity. The respective enum is shown in Listing 6.3. Granularity in this context means the resolution or level at which to execute specific operations. One example is the computation of statistics. Assume the users want to pre-compute the mean value automatically in the engine. This can

either be performed across a single block, across all blocks in a step, across multiple steps or globally for the entire variable. It is possible that a user may want to compute all of them. Therefore, the members are assigned values for the bitmap.

```
1  enum JDAIGranularity{
2    J_DAI_GRAN_BLOCK    = 1,
3    J_DAI_GRAN_STEP     = 2,
4    J_DAI_GRAN_VARIABLE = 4};
```

Listing 6.3: DAI Interface - Granularity: The enum to specify the granularity at which operations should be performed.

Configuration

One of the most fundamental decisions to make was how to communicate any configuration set using the DAI interface to the ADIOS2 engines. At the time of designing the interface and implementing the respective prototype, ADIOS2 did not offer any possibility to pass additional parameters to the engine directly. This route was not taken to avoid any changes to the application-facing ADIOS interface. While ADIOS offers configuring engines by adapting an XML file, there were no options to define custom parameters without introducing more extensive changes. As of release 2.8, ADIOS2 reintroduced the concept of a plugin engine. ADIOS2 already offered the plugin engine abstraction in the past, and the key-value engine was initially implemented as such. However, the sudden removal of the plugin engine concept forced a reimplementation as an engine.. With the recent major version update, slightly changed plugin engines came back. For example, they now offer the option to pass additional parameters. For the scope of this thesis, it was not sensible to migrate the key-value and database engines to the new concept. Larger changes to the core library often come with unexpected behaviour changes and, thus, possibly numerous bugs. A good example of the challenges is the attempt to compile the 2.8 release on the institute's cluster. Even after debugging various configurations, 2.8 still does not compile on this cluster at the time of writing. Thus, it was decided to stick with version 2.7. for the JULEA engines. As a result, the option to pass parameters is not yet available.

Namespace	File Name	Var Name	Granularity	Statistic	Step Number
Eval	test.bp	T	Block	Mean	4

Table 6.2.: Schema to communicate the configuration of the precomputations to the engine. Here, the configuration specifies that the mean value for the variable "T" is computed, averaging over 4 blocks.

Another option to specify the DAI configuration is by adding dedicated functions. Instead of passing additional parameters to the engine, the DAI configuration is communicated to the

engine through the database. The DAI configuration is written to a specific database table, shown in Table 6.2. When the engine is initialised, this table is read by the database engine.

The configuration consists of the project namespace, the file name, the variable name, granularity, the statistic to be computed and the number of steps across which to compute the statistic. In the example, the mean value for the temperature variable is computed over four steps. Communicating the setup to the engine through a database is not a very flexible approach, as the database schema cannot be changed easily on just one side. Both the engine and the DAI implementation have to be changed at the same time. However, changes to the engine's behaviour are not frequent.

Project Namespace The concept of the project namespace is introduced to structure the database's namespace. This makes some requirements less strict. For example, every filename can be used once per namespace, not in general. If the namespace is set to contain all measurements of a cluster, there can be conflicts between different users. On the other hand, if the namespace is set too small, the overhead of managing data across different namespaces becomes too high. The most common functions in connection to the namespace are shown in Listing 6.6. Container formats such as BSON, for example, demonstrated that the option to retrieve all members of a space is very useful. Thus, the functions allow reading all file names, variables and entryIDs belonging to a namespace.

The option to set the namespace at the beginning of the application to avoid passing the parameter continuously was discarded because it creates a state for the storage system, which is not expected from a storage system. The project namespace also enables keeping custom metadata computation to a specific part of the data.

Pseudo-Dynamic Database Table Schemas The previous chapter studied various database technologies to find a suitable candidate for the DAI support. One feature that would have been very convenient was dynamic table schemas. While dynamic table schemas were a part of Cassandra, they no longer are. There is still the option to use collections to store new features, such as custom metadata. Nevertheless, the problem remains that queries are not directly possible on such values. The solution chosen for this thesis is to use relational databases in a pseudo-dynamic way. In the database engines, the table schemas are defined upon the engine initialisation. So, if the custom metadata requirements are set before the engine constructor is called, then the database schema can be changed accordingly. This will only work if the user specifies everything before opening the file because the `open` call of ADIOS2's `IO` object invokes the constructor of the set engine. The pre-computation and tagging details will be explained in the respective paragraphs below.

6.2.1. Pre-Computation

The first group of functions for the DAI interface is related to specifying pre-computations and respective settings. At the moment, only statistics can be precomputed that are specifically implemented in the engine. There is no option to define arbitrary mathematical operations because of various reasons. For one, the configuration of DAI settings is communicated via a database table, which hinders the usage of callback functions since it is not sensible to store

function pointers in the database. Also, a custom math syntax could be specified to avoid relying on callbacks. While the definition of an unambiguous and powerful enough syntax is not trivial itself, the requirement to store the expression in a relational database is also not ideal. This would mean either mapping the parts of the expression to database types or storing the expression as an opaque buffer, which would require serialising and deserialising the format. If the engine is reimplemented as a plugin engine in the future, the additional parameters allow passing callback functions. Then arbitrary operations can be calculated during the I/O process.

Another problem with the computation is the different range of applications and data models that can be used. For example, the mean calculation is only straightforward if the data is based on a structured and uniform grid. Otherwise, the grid information must be considered to allow sensible interpolation. This could also involve the remapping of the data. The challenges of different data models will be explained in more detail in Section 6.3.

In Listing 6.4, the central function `j_dai_pc_stat()` is shown. As previously mentioned, it takes different parameters like the statistic type, the granularity and the step number instead of defining a separate function for each statistic. As a reminder, the database engine has only two tables holding the metadata; one for the global variable information and one table for the block metadata. Custom metadata that is computed across several steps do not fit in either of them. Therefore, the resulting custom metadata is stored in a separate table to enable various usage options making the example call in Line 3 possible. The computation name is required for creating the table name and when reading the new metadata. Otherwise, the entire computation specification has to be passed every time to identify the correct database entries.

```
1 void j_dai_pc_stat(gchar const* n_space, gchar const* computation_name, gchar const* file,
      ↪ gchar const* var, JDAIGranularity g, JDAIStatistic s, size_t every_n_steps);
2
3 j_dai_pc_stat("Eval", "DailySum", "test.bp", "P", J_DAI_GRAN_STEP, JDAJ_DAI_STAT_SUM, 4);
4 j_dai_pc_stat("Eval", "LocalSum", "test.bp", "P", J_DAI_GRAN_BLOCK, JDAJ_DAI_STAT_SUM, 4);
```

Listing 6.4: DAI Interface - Statistics Precomputation: The definition of the central function and an example call to compute the sum of the precipitation variable P over four steps.

The option to specify a custom input range for the statistics computation gives the user lot more freedom than a static approach, for example, for every hour. As the step abstraction in ADIOS2 can mean various things, the goal of the DAI design was to keep this flexibility. Instead of assuming that every application will use a step to store the data accumulated in one hour, the last parameter allows different interpretations, as shown in Table 6.3 where a step equals six hours. If the granularity is set to `J_DAI_GRAN_BLOCK` as shown in Line 4 of Listing 6.4, the precipitation is summed up over four steps as well but at a block resolution. Instead of one result every four steps, there is one for every block, that is, every used process. They are cached to avoid reading the results of the previous steps, which comes with all the drawbacks of caching, most notably the increased memory consumption. The challenges of inconsistent data are no problem here, as ADIOS2 does not allow updating or overwriting data.

Namespace	File Name	Var Name	Step	BlockID	Sum
Eval	test.bp	P	4	-1	42

Table 6.3.: Statistics table "DailySum" as created by the previous example call for `j_dai_pc_stat`. The table contains the precipitation sums of every 4 steps. Assuming a step is written every 6 hours in the simulation, the custom result is the daily precipitation. The blockID is set to -1 to indicate that the used granularity is the step level, as 0 is a valid blockID.

Convenience and Defaults To make the life of application developers and users doing post-processing easier, several functions are computed automatically by default. For one, the mean value, the sum and the variance are computed for every block alongside the extrema that were already present in the BP formats. So, to enable the computation of the mentioned statistics, setting the engine type to `julea-db` is sufficient. If the pre-computation is not sensible for a specific scenario, the user can specify to use only the metadata offered by the native BP formats. When calling `j_dai_use_original_format`, no additional metadata is pre-computed.

Namespace	**File Name**	**Var Name**	**Step**	**Block**
String	String	String	Integer	Integer
ShapeSize	**Shape**	**StartSize**	**Start**	**CountSize**
Integer	BLOB	Integer	BLOB	Integer
Count	**MemStartSize**	**MemStart**	**MemCountSize**	**MemCount**
BLOB	Integer	BLOB	Integer	BLOB
isValue	**Value**	**Min**	**Max**	**Mean**
Boolean	Double	Double	Double	Double
Sum	**Variance**			
Double	Double			

Table 6.4.: Schema to store the extended block metadata. The field names (bold) are printed above the respective field type. A light-grey line separates them. Due to the table's width, line breaks are necessary which are indicated by the doubled horizontal line.

Size of Custom Metadata

One reason to opt out of the default to automatically compute the additional statistics may be the concern of the increased data size. Therefore, the size of the custom metadata will be examined in detail below. For every block, three additional variables of the same data type as the ADIOS2 variable are stored in the database. The integers noted in Table 6.4 are all of the type `size_t`. This results in the following equation:

$$\text{Entry Size} = 3 * \text{s (string)} + 7 * \text{s (size\_t)} + 5 * \text{s (BLOB)} + 1 * \text{s (bool)} + 6 * \text{s (double)} \quad (6.2)$$

where

- s() = sizeof() which is the size of the respective data type
- sizeof (string) = depends on the string length
  →1 byte per character + terminating character
- size_t: `typedef long unsigned int size_t;` [5]
  →sizeof (size_t) = 8 bytes
- sizeof (BLOB) = array of type size_t
  →8 bytes per element
- sizeof (bool) = 1 byte
- sizeof (double) = 8 bytes

Assuming each array has at least one element or character and the variable is not a single value, the minimal size is:

$$\text{Entry Size}_{min} = (3 * 2 + 7 * 8 + 5 * 8 + 1 * 1 + 5 * 8) \text{ bytes} = 143 \text{ bytes} \quad (6.3)$$

Assuming strings with a length of 9 characters and a three-dimensional dataset, the size is already:

$$\text{Entry Size} = (3 * 10 + 7 * 8 + 5 * 8 * 3 + 1 * 1 + 5 * 8) \text{ bytes} = 247 \text{ bytes} \quad (6.4)$$

This example shows that the project namespace, the file name and the variable name can contribute significantly more to the entry size than the additional statistics. Given the small storage overhead, storing three additional doubles is justifiable. Especially when considering that this means the user does not have to change anything in the application and still can benefit from the pre-computation. Due to the expected performance improvement when accessing the data, the pre-computation of the three statistics is the default. Apart from this scenario, the defaults of the DAI are more conservative and avoid adding to the storage requirements if the advantages are not widely applicable. For example, the automatic pre-computation of the climate indices makes no sense for the general HPC application.

6.2.2. Tagging

Tagging is another important feature of the DAI interface. Inspired by EMPRESS and the requirements drawn from the data access analysis and user survey, marking areas of interest seems to be a good way to mitigate the insecurities arising from the varied access patterns. When users specify which variables and even which conditions for a variable are most important to them, the storage system can accommodate for that.

[5] as defined on Manjaro 21.3.6

```
void j_dai_add_tag_d(gchar const* n_space, gchar const* tag, gchar const* file, gchar
    ↪ const* var, JDAIGranularity g, JDAIStatistic s, JDAIOperator op,  double threshold);
j_dai_add_tag_d("Eval", "minOver42", "test.bp", "T", J_DAI_GRAN_BLOCK, JDAJ_DAI_STAT_MIN,
    ↪ J_DAI_OP_GT, 42);

j_dai_pc_stat("Eval", "DailySum", "test.bp", "P", J_DAI_GRAN_STEP, JDAJ_DAI_STAT_SUM, 4);
j_dai_add_pc_tag_d("Eval", "DailySumOver42", "test.bp", "DailySum", J_DAI_OP_GT, 42);
```

Listing 6.5: DAI Interface - Adding Tags: The function to specify the tagging conditions and an example call. The JULEA engine will now add an entry whenever the mean block temperature is higher than 42°C. The suffix indicates the data type: `_d`: double

In Listing 6.5 the most important tagging-related function `j_dai_add_tag` is shown. Two versions are discussed: one that is independent of pre-computations and allows tagging all existing features, and another that allows tagging parts of the pre-computation. A common parameter is the statistic. It needs to be specified because all metadata in the database is derived data, not the data itself. For example, the database holds the minimum, maximum and mean block values. So, the user needs to explicitly state which of these characteristics should be considered. The parameter list allows the function to cover many use cases. It, therefore, reduces the API size to a single function instead of offering a function for each granularity level, statistic and comparison operator.

- Independent of Pre-Computation (`j_dai_add_tag`):
  It allows specifying the conditions when data should be marked in the database without requiring any pre-computations. As for the previous DAI functions, the project namespace, the file and variable name, the granularity and the statistic have to be passed. The other parameters are the tag name that will function as the table name and the comparison operator and threshold. With the default of automatically pre-computing the mean, sum and variance for every block, the possible characteristics are very similar to the pre-computation-based version. With the original format, the independent version could only tag the minima and maxima of blocks and variables. While this might be interesting, it is unnecessary because this metadata is easily available through the native ADIOS2 API. Note that it is impossible to tag statistics spanning multiple steps with this version. The corresponding table schema is shown in Table 6.5.

- Tagging Pre-Computation Results (`j_dai_add_pc_tag`):
  As the pre-computation of derived statistical metadata is an important aspect of the functionality of the DAI, it should also be considered for tagging. Since the conditions of the computation have been previously specified, the parameter list is shorter and the function less error-prone. A new parameter is the computation name of the pre-computation. This also dictates the order in which the functions need to be called. This version of the tagging function also requires two tables per new feature, one for the pre-computation results and one for the tagging results.

Different definitions of the functions are required for the different data types the threshold can have because templates are not available in C.[6]

In general, there are two ways how tagging could be incorporated into the existing data mapping. First, the tags are stored in the block and variable table as well. Second, the tags are managed in one or more separate tables.

For the former solution, there are different options for how this can be implemented. The tag entry can be added as a separate column to the table. Another way is to use a collection to store the tags. For example, a new column can be added to the table that holds a BSON or a similar document that collects all the tags added for a specific block or variable. As mentioned before, opaque data types have the drawback that queries cannot be performed directly on top. Instead, the BSON has to be read entirely to access specific tags. This would mean, that if a user wants to check how many tags were set, every BSON in the table, has to be read and deserialised only to find in the worst case, that no tags were added at all. Since tags are extremely likely to be accessed frequently because they mark interesting areas, this is not a sensible solution.

Namespace	File	Var	Step	Block	ID	Gran.	Stat.	Op.	Threshold
Eval	test.bp	T	0	2	4	Block	Mean	GT	42

Table 6.5.: DAI Interface - Tag Schema: The database schema to store the results for the independent tag conditions specified in Listing 6.5 where Gran. = Granularity, Stat. = Statistic, Op. = Operator. The ID is the entryID (if available) that is autogenerated by the database.

So, the decision has fallen onto the second option; storing the tags in a new table as shown in Table 6.5. This approach allows specifying new tagging scenarios after the file is opened, which is impossible when the block table schema needs to be modified. Also, the performance both for inserting and reading tags is better when only a subset of the simulation output is affected. Most database indexing mechanisms work better for smaller datasets. Additionally, sifting through fewer data makes the data exploration to gain new insights easier. Note that the ID can only be set if the granularity level is the entire variable or a single block. Steps currently have no immediate representation and thus no separate ID. It would be sufficient to store either the project name, file name, variable name, step and block or the project name and the ID. The advantage of the former is that the user can directly access the data without retrieving an ID from the database first. However, when a lot of subsequent data accesses happen, working using the ID is more convenient and also more efficient as only a single column needs to be searched instead of four. As both scenarios are plausible, both options are offered and thus stored in the tag table.

[6] There are crafty solutions in C11 like `_Generic` that can be used to accomplish this. However, `_Generic` is not valid C++ code and would require another interface implementation for C++, e.g. based on function overloading.

6.2.3. Reading

The following presents the functions to access the pre-computed and tagged data. They are grouped according to the corresponding abstraction level: project namespace, step, entry and specific query functions. At the end, a query example is shown to illustrate how the reading API can be used.

To make the interface more user-friendly, an external library, namely the GLib, was used to return the results. The reason is the memory allocation in C and the absence of a vector data structure. The user cannot allocate sufficient memory without knowing the array size, that is, the number of elements. Therefore, GArrays are used. A pointer is passed to the function, meaning the user can free the memory at the end of the code. Note that the GArray parameter is always the last parameter as it returns information.

The alternative to using an external library would be to either allocate the memory inside the DAI and thus inside JULEA or to offer two different calls whenever more than a single value needs to be retrieved. The first is to get the number of elements, and the second is to retrieve the actual result where a buffer is passed that the user allocated.

Project Namespace Listing 6.6 shows all the functions related to the project namespace. They allow to explore data when only the project namespace is known, which enables the user to examine the data without prior knowledge, an important feature of the self-describing data formats. So, the functions resemble those from the I/O library and enable the retrieval of all files, variables or entryIDs belonging to a specific namespace.

Step The definition of the step-related functions shows the problems of providing a general-purpose system. The functions are shown in Listing 6.7. For ADIOS2 variables, it is plausible to work on the entryID (Line 1) or to work on the steps and blocks (Line 3). The entryID is the unique ID auto-generated by the underlying database. It is built from the combination of the file name, the variable name, the step number and the blockID. A common scenario is retrieving all IDs where the variable meets a specific condition. At the same time, accessing and comparing different variables for the same steps and blocks is also a basic requirement. For example, the user may want to know how high the minimum temperature in Hamburg was in relation to the precipitation in Hamburg at the same time. Thus, the interface offers both.

Entry As previously mentioned, the entry concept is the foundation for many other functions. It is possible to retrieve all entries for a namespace. This is only useful when there are further options to get other metadata features. The most important functions are shown in Listing 6.8. The complete entry-related API is shown in Appendix B. As mentioned in the design, the advanced data access is left to the established I/O interfaces. For example, users may want to access data from different blocks at the same time step. This can be done well using the ADIOS2 API and intersection boxes. Therefore, the DAI focuses on reading the metadata. There is one exception. To add convenience for users, they can retrieve the data object belonging to a specific database entry whose entryID was returned in previous DAI

calls (Line 1). The functions range from retrieving the according step (Line 2) or block (Line 3) to getting a specific statistic.

```
1  void j_dai_project_get_files(gchar const* name_space, Garray* file_names);
2  void j_dai_project_get_vars(gchar const* name_space, Garray* variable_names);
3  void j_dai_project_get_entry_ids(gchar const* name_space, Garray* entry_ids);
```

Listing 6.6: DAI Interface: Project namespace related Functions

```
1  void j_dai_step_get_ids(gchar const* n_space, gchar const* file, gchar const* var, size_t
       ↪ step, Garray *ids);
2  void j_dai_step_block_get_id(gchar const* n_space, gchar const* file, gchar const* var,
       ↪ size_t step, size_t block, size_t* id);
```

Listing 6.7: DAI Interface: Step related Functions

```
1  void j_dai_entry_get_data_d(gchar const* n_space, size_t entry_id, Garray* data);
2  void j_dai_entry_get_step(gchar const* n_space, size_t entry_id, size_t* step);
3  void j_dai_entry_get_blockID(gchar const* n_space, size_t entry_id, size_t* block_id);
```

Listing 6.8: DAI Interface: Entry/Block related Functions

Query The complete query API can be found in Appendix B. The most important query-related function is used in the example in Listing 6.9 in Line 7. The example is the shortened code for the fourth query of the post-processing evaluation discussed in Section 7.5. A query that combined various functions was chosen to highlight different aspects of the DAI. The goal is to determine the maximum precipitation sum when the temperature is above 40°C at the same time (step) and place (block). The necessary steps are discussed next: First, the IDs of all blocks have to be read where the temperature exceeds the threshold (Line 7). Second, for every entryID, the respective step and block must be retrieved to ensure the time and space boundaries are met (Line 10-11). Third, the precipitation block sum is retrieved and stored for every step and block (Line 12-13). Finally, the maximum block sum is computed (Line 16).

This query shows the convenience of the DAI but also its limitations. Different mechanisms must be offered as it is unclear how a user will access the data. Thus, the interface has functions that take the entryID of the database as a parameter to avoid always passing the parameters describing a block. These are the file name, the variable name, the step number and the blockID. Their combinations give the unique entryID. This is only the case because there is only one table per metadata type in a project namespace and no ambiguity. Suppose a project namespace becomes too large and thus infeasible to manage, for example, all block metadata in a single table. In that case, introducing a separate unique identifier will probably be a good idea. It is not yet done because of the additional overhead to ensure the uniqueness across different tables. Tag tables require additionally passing the tag name to distinguish IDs belonging to different tags. More about the options to extend the interface and its capabilities are discussed in the next section.

```
1  void QueryRainTemperatureCombined(std::string project, std::string file) {
2    double result; size_t step, block; size_t *steps, *blocks;
3
4    std::vector<double> blockSums;
5    GArray *tempIDs = g_array_new(true, true, sizeof(size_t));
6
7    j_dai_query_get_ids_d(project, file, "T", J_DAI_GRAN_BLOCK, J_DAI_STAT_MAX,
         ↪ J_DAI_OP_GT, 40, tempIDs);
8
9    for (int i = 0; i < tempIDs->len; ++i) {
10     j_dai_entry_get_step(project, g_array_index(tempIDs, size_t, i), &step);
11     j_dai_entry_get_blockID(project, g_array_index(tempIDs, size_t, i), &block);
12     j_dai_block_get_stat_d(project, file, "P", step, block, J_DAI_STAT_SUM, &result);
13     blockSums.push_back(result);
14   }
15   if (blockSums.size() > 0)
16     auto maxSum = *max_element(blockSums.begin(), blockSums.end());
17 }
```

Listing 6.9: Example Query Application used in the Evaluation. The task is to find the maximum precipitation (P) sum when the temperature (T) is above 40 degrees. Initialisation and string formatting are omitted.

6.2.4. Domain-Specific Functionality

The previous discussion targeted general-purpose functionality. An example of how domain-specific functions can be offered as part of the DAI is shown in the following. The climate indices can benefit from the pre-computation, as it allows to specify over which time interval the extrema should be computed. The block and global variable extrema that are part of the ADIOS2 data characteristics are insufficient as the climate indices work on daily extrema. Besides specifying the pre-computation, the user needs to add a tag to all occurrences where the respective index condition is met. For example, the number of frost days is equal to the number of entries in the corresponding tag table. While the computation of the climate indices can be expressed through `j_dai_pc_stat` and `j_dai_add_pc_tag`, a possible implementation for a simpler interface is shown in Listing 6.10. The climate indices are clearly defined. Given a library developer invested in climate research, the knowledge of these definitions can be used in the engine without requiring the user to specify it explicitly.

One alternative to pre-computing and tagging is shown in Line 7 and 8. It makes the user's life easier as it is much shorter and more similar to the CDO syntax. Similar to the other DAI functions, an enum (Line 1 - 5) is defined to specify the computation as a parameter. As with the pre-computation, the interval has to be set, over which to compute the climate indices. While the definition states that the daily extrema are used, the user must pass the number of corresponding ADIOS2 steps. An alternative proposal to adding this as a parameter is introducing a new function like `j_dai_set_day_length`. The advantage is that if numerous functions would take the number of steps as a parameter, having a separate function which

is only called once removes room for error. At the same time, it introduces an internal state that may not be obvious to the user depending on how the application code is written.

```
1       enum JDAIClimateIndex{
2         J_DAI_CI_SU = 1,      // summer days: T_max > 25 C
3         J_DAI_CI_FD = 2,      // frost days: T_min < 0 C
4         J_DAI_CI_ID = 4,      // icing days: T_max < 0 C
5         J_DAI_CI_TR = 8};     // tropical nights: T_min > 25 C
6
7       j_dai_pc_stat("Eval", "DailyMin", "test.bp", "T", J_DAI_GRAN_STEP, J_DAI_STAT_MIN, 4);
8       j_dai_add_pc_tag_d("Eval", "FrostDays", "test.bp", "T", "DailyMin", J_DAI_OP_LT, 0);
9
10      j_dai_pc_ci("Eval", "test.bp", "T", J_DAI_CI_SU | J_DAI_CI_FD);  // Precompute
            ↪ climate indices
11      j_dai_set_day_length(4);
```

Listing 6.10: DAI Interface - Climate Indices: The possible climate indices that can be precomputed by an engine are shown. The definition of the individual indices is provided as a reminder. It can be found in detail in Section 6.1.1. In case of the climate indices, one or more can be passed to the function. In the example, both the summer days (`su`) and the frost days (`fd`) are computed.

6.3. Generalisation

In the following research question 3.2. will be discussed, that is *How can this approach be generalised?* Self-describing data formats are used in a very varied landscape. Not only do the software environments and programming languages vary, but the hardware is also very different from climate research to high-energy physics. Not all scientific fields use simulations. Often detectors are employed that generate very different data. The workflows are very different depending on the area. In physics, the raw data is stored in archives but not used directly. Typically, reconstruction software and its results are used. Also, not all areas use grids, whether structured or non-structured. This means that the requirements for an interface, storage and post-processing are highly heterogeneous, making it hard, if not impossible, to generalise an interface so everyone can use it. Next, the focus lies on the data modelling that can introduce problems for the pre-computation even if still ADIOS2 and HDF5 are used.

6.3.1. Different Ways to Model Data

One major problem when designing an interface that should work on different data formats is the numerous ways to model data. Different data formats come with different data models. This makes computation difficult because the data model influences how statistics can be computed and whether it makes sense to compute this information at all. For example, it is not very useful to compute the daily sum of temperatures. To illustrate the problems that arise from the different data models, below four versions are discussed how a variable can be stored in HDF5 [Breitenfeld et al., 2020]. This makes a big difference in the computation

of additional data statistics. The example does not consider unknown dimension sizes, data chunking or compression. It simply highlights different ways how data can be organised[7]. **T** is the number of steps, **A** is the number of arrays, **X** is the number of rows, and **Y** is the number of columns.

1. One 4D dataset: **[T,A,X,Y]** or **[A,T,X,Y]**
   All data is stored in one four-dimensional dataset. The code generating the first example layout is shown in Listing 6.11. Here, the timesteps are stored as the first dimension. However, depending on the use case, it might be advantageous to reverse the order of the for-loops. This will result in the second layout, where the timestep is the second dimension. Depending on the ordering, the computation of, for example, the mean value has different meanings. For the first ordering, an average over time is computed, and an average of one timestep in the second.

2. One 3D dataset per step: **T** * **[A,X,Y]**
   Another option to model the data from the previous example is to store one three-dimensional dataset for each timestep. This makes the computation of a mean value more complex because an average over time can no longer be computed using one dataset. For example, computing the mean value over four steps requires reading the data of four different datasets. For ADIOS2, such a data model would result in a very different implementation than for the current version. An engine deals with one variable at a time. So, it is not sufficient to simply cache results from previous steps, as the context of the variable is left. To have a different variable for each time step means the engine has to have further information on which other variables belong to the current ones. In the worst case, the engine, during writing one variable, would need to read all variables previously written. That is not a reasonable way to pre-compute statistics.

3. One 3D dataset per array: **A** * **[T,X,Y]**
   Similar to the example before, the dataset could also be modelled using a different three-dimensional dataset where the entire time series is included inside each dataset. This model has similar problems to the previous one.

4. One 2D dataset per step and array: **T*A** * **[X,Y]**
   The data could also be stored in two-dimensional datasets. This would increase the number of datasets depending on the number of arrays and timesteps. Again, computing the mean value requires reading various datasets. The problems this would bring for ADIOS2 variables will possibly be worse than for the three-dimensional datasets.

[7] According to the HDF5 developers there are more than a dozen ways to implement a 3D variable that uses timesteps [Breitenfeld et al., 2020].

```
1  for timesteps: 1...T
2    for number of arrays: 1 ... A
3      create a double array with size [X,Y]
```

Listing 6.11: A simple example of pseudo-code how data can be modelled using HDF5. **T** is the number of steps, **A** is the number of arrays, **X** is the number of rows, and **Y** is the number of columns.

LAMMPS I/O using ADIOS2 - Unideal Format Usage

Another problem with the computation of statistics is the different types of data stored in a variable. For example, the current implementation assumes that a variable will only hold data related to one physical feature, such as temperature or precipitation. However, it would also be possible to store the temperature and precipitation results in one variable. One option would be that every other value belongs to the temperature or the precipitation, respectively. This is not only a theoretical problem, as the example below demonstrates. It is from LAMMPS (Large-scale Atomic/Molecular Massively Parallel Simulator), a popular physics simulation and works well to illustrate the challenges. LAMMPS uses numerous I/O libraries, among others ADIOS2.

```
1  double    atoms      11*{32000, 8} = -3.97451 / 32000
2    step  0:
3      block 0: [    0:15999, 0:7] = -2.07314 / 16000
4    (0,0)    1 1 0        0        0        -0.145828 -0.669946 -2.01722
5    (1,0)    2 1 0.839798 0.839798 0        -1.2587    1.4644    1.58791
6    (2,0)    3 1 0.839798 0        0.839798 -1.26427  -0.661341  1.11072
7    (3,0)    4 1 0        0.839798 0.839798  1.69578   1.9159   -0.675027
8    (4,0)    5 1 1.6796   0        0        -0.875029 -0.600509 -0.226609
9    (5,0)    6 1 2.51939  0.839798 0        -0.751965 -0.560737 -1.78465
10   ...
```

Listing 6.12: The LAMMPS Data of the ADIOS2 variable `atoms` is displayed using bpls. Only the beginning of the first block of the first step is shown. The information at the start of Lines 4 to 9, for example, `(0,0)`, is part of the bpls output to indicate the position. The different columns hold the ID, a type, the position of the atom in a three-dimensional space (x,y,z) and the corresponding velocities (vx,vy,vz) as indicated by the attribute shown in Listing 6.13

LAMMPS defines an ADIOS2 variable named `atoms`. However, it does not simply hold just one type of information. Looking only at the output of `bpls` shown in Listing 6.12 it is not clear what data is stored. The variable name is not sufficient here. The first value of each line appears to be an index or counter. The last values look like typical scientific data. To understand what data is stored, one of the ADIOS2 attributes is needed. In Listing 6.13, the content of the `columns` attribute is printed. Combining the information from both listings, it becomes clear, that the `atoms` variable holds an ID, a type as well as their position in a 3D system (x,y,z) and the corresponding velocities (vx,vy,vz). This process of understanding the data representation cannot be easily automated.

```
string   columns            attr   = {"id", "type", "x", "y", "z", "vx", "vy", "vz"}
```

Listing 6.13: The `columns` ADIOS2 attribute of the LAMMPS application is displayed using bpls.

The main difference from the examples previously discussed is that this simulation of atoms does not work on a fixed grid. A climate simulation always has a temperature value for a specific grid point. It does not disappear. However, simulating moving atoms follows the individual objects, that is, updating the positions, rather than keeping a static view of the space. If all atoms moved away from a particular location, there is no need to store any information about this area. The computation of the mean value that works well for the HeatTransfer application discussed in the evaluation makes no sense for this dataset. Averaging over IDs, types, and positions does not hold any value for the application users.

Another function was designed to mitigate these challenges and compute sensible additional statistics. It is called `track_feature` and allows specifying a variable and which data parts to consider. Using the LAMMPS example, a user can pass, among others, the variable name and which elements in the data vector should be tracked, for example, the velocity in the x direction. Then, this information can be stored separately to enable sensible statistics. At first glance, this might not seem very practical. However, given the reluctance to change application codes, this function can offer a way how the user can still benefit from both the format dissection and the pre-computation without changing the I/O code. With a plugin engine, the parameters could be specified in an XML, and there is no need even to recompile the application.

```
void j_dai_track_feature(gchar const* n_space, gchar const* file, gchar const* var, gchar
    ↪ const* feature_name, size_t index_in_data_vector);

j_dai_track_feature("Eval", "atoms.bp", "atoms", "x_velocity", size_t index_in_data_vector);
```

Listing 6.14: Function to track a feature that may be hidden in the data vector if the ADIOS2 variable is not used as originally intended

6.3.2. Unified Interface

When discussing how the format dissection and the DAI interface can be generalised to other formats or scientific areas, one aspect that needs to be discussed is whether the current interface is suitable for this in general. One point to discuss here is whether or not it is sensible to unify the DAI interface with the I/O library more. One advantage would be that the user only has to utilise one interface, that of the I/O library, without additional calls to the DAI interface. In the following, specific solutions in the direction of one interface are presented first. Then, the general options regarding the implementation are discussed.

Removing DAI Calls from Main Application As previously mentioned, ADIOS2 offers the concept of a plugin engine that allows passing additional parameters to the engine. Plugin engines were only recently reintroduced after being discarded in 2018. This would make the communication from the DAI to the engine through database tables unnecessary. It would enable only using the ADIOS2 library API in the main application. The pre-computation and tagging could be adopted so that their specification can be done as a set of parameters. However, while this makes using two APIs in the application unnecessary, the parameter list will grow uncomfortably long, as each parameter must be set explicitly. A small example of how such a parameter specification can look is shown in Listing 6.15.

```
1  adios2::ADIOS adios;
2  adios2::IO io = adios.DeclareIO("writer");
3  io.SetEngine("Plugin");
4  adios2::Params params;
5
6  params["PluginName"] = "WritePlugin";
7  params["PluginLibrary"] = "PluginEngineWrite";
8
9  io.SetParameters(params);
```

Listing 6.15: Configuration of the novel plugin engine concept in ADIOS2. The interface allows to pass additional parameters that can be used to specify the DAI settings.

The corresponding XML file is shown in Listing 6.16. As mentioned previously, ADIOS2 offers to specify the I/O parameters in an XML file instead of having to call the respective functions from the application. This makes changes to the I/O behaviour easy and avoids recompilation. So, if the JULEA engines were plugin engines, all DAI calls could be changed to corresponding XMLtags , thereby avoiding any application changes.

```
1  <adios-config>
2  <io name="writer">
3      <engine type="Plugin">
4          <parameter key="PluginName" value="WritePlugin" />
5          <parameter key="PluginLibrary" value="PluginEngineWrite" />
6          <!-- any parameters needed for the plugin can be added here in the parameter tag -->
7      </engine>
8  </io>
```

Listing 6.16: Configuration of the novel plugin engine through an XML file.

Another option for the pre-computation of results is to extend and enhance the capabilities of the ADIOS2 operators. These could be used to specify which custom statistics should be computed. However, currently, the signature and functionality are really limited. Changes to the application-facing ADIOS2 interface were not a sensible option for this thesis. To add

functionality to the public API is not easily achieved, and the challenges of making it compatible with all existing engines and transports would not have been productive with respect to the overall question of how self-describing data formats can be managed efficiently. The goal was to find an option that kept changing library interfaces to a minimum. Otherwise, the approach cannot be applied to a different data format and the corresponding library. For example, extending the operator functionality in ADIOS2 does not help offer these features in HDF5 in contrast to a separate interface like the current DAI.

Implementation Placement One interesting aspect of making the interface more uniform is the location to implement the dissection and the DAI. In the following, the three options discussed throughout the thesis are summed up.

- Inside the I/O Library: Current version

  - The format dissection needs to be done again for every new format and library. While the knowledge and experience with the previous formats will help, this approach does not easily scale.
  - The pre-computation is custom as well and needs to be designed and implemented for each format.
  - The library must be compatible with C/C++; otherwise, JULEA and the DAI cannot be called, which limits the applicability. There are plans for Python bindings for JULEA, but they are not yet offered.
  - Another problem is the dependence on the library. Both HDF5 and ADIOS are under active development, which means that the API is only "fairly stable". For example, as of August 2022, the HDF5 developers advise not to use version 1.12, which is the version of the current JULEA VOL connectors. Instead, 1.13 should be targeted even though this is an experimental branch, and more interface changes could be made before 1.14 is released.
  + The dissection happens independently of the users. They do not need to do anything but still can benefit from the dissection and the flexible placement of the storage backends on faster hardware. This is possible because the dissection is completely encapsulated inside the engine and due to the default settings of the pre-computation.
  + No application changes need to be made. The used engine can be specified in the XML, and the query functionality of the DAI will not be added to the main simulation but to the post-processing scripts. These are often written and used by a single scientist, reducing the overhead of changes.
  + The metadata access is fine-grained, and the features of faster and more expensive storage hardware can be used because it is possible only to store the metadata there.
  + The pre-computation of statistics offers great performance improvements.

- Above the I/O Library: EMPRESS

  - The format itself is not split. The metadata in EMPRESS and EMPRESS2 is stored additionally, leading to data duplication.
  - The largest drawback is that the users have to set everything explicitly. For example, they need to mark where the loop for steps should begin and end. Also, every variable, tag and attribute has to be specified. This is regardless of what was specified using the original I/O library interface.
  - The flexible model above the data format leads to much reimplementation of existing functionality, especially for ADIOS2.
  + Because the user specifies everything they want to use, EMPRESS is more flexible and less dependent on the specific format as its implementation is outside the library. This means that different formats can most likely be more easily used.

- Below I/O Library: JULEA and related project (CoSEMoS)

  - Abstracting from different self-describing data formats is difficult. While their concepts may be similar, the information that must be stored to export the original format differs. Introducing a new layer below the I/O libraries requires mapping the data formats to it, which is very close to introducing a new data format.
  - This approach still requires changes to the I/O library to redirect the I/O path to JULEA. This can be done in the form of an engine, a plugin or a connector, depending on the concepts offered by the specific library.
  + This would offer a new interface to the data formats that can be used for several formats. Adopting a new format should be easy when enough formats are studied for the interface concept and implementation. This approach is most likely to scale.

Chapter Summary

This chapter addresses research questions 3.1 (What functionality is required for the DAI interface?) and 3.2 (How can this approach be generalised?). To determine the necessary functionality, the target audience has to be determined. Different post-processing tasks from the domain of climate research were studied, and a subset of the climate data operators (CDOs) was selected. Besides the computations of statistics like the sum, mean, or variance, domain-specific features like the climate indices were chosen. The evaluation of the second part of the user survey revealed that users do not like to use POSIX and SQL. They deem I/O libraries acceptable but overall favour python for data access. Combining the previous results allowed for deriving requirements and use cases. These use cases lead the implementation effort as they help to keep the end user in mind. The functionality of the DAI includes pre-computation, tagging and reading. The computation of the climate indices in the engine was presented to highlight how domain-specific features can be incorporated. The final discussion showed that the DAI prototype offers great potential for the

specific setup. In order to have other formats and scientific research areas benefit from it as well, it has to be generalised as it is currently format dependent. One challenge of offering a more general interface is that there are numerous ways to model data, even if only one library is used. For example, time can be represented as a dimension within a dataset or by creating a separate file for each time step. This could be addressed by restricting the data models or bringing the entire concept to a higher level, for example, by developing ad-hoc indices over different formats. This possible direction will be explained in more detail in future work in Chapter 8.1.

7. Evaluation

This chapter covers the evaluation of the HDF5 VOL plugins and ADIOS2 engines both for using key-value stores and relational databases.
The performance of the different JULEA backends is determined first to have a baseline for the other measurements. Also, a short evaluation of the new BlueStore backend for JULEA is presented. As the object store plays an important part in the performance of the VOL plugins and the engines, this study was done to see whether it would get better with BlueStore. Afterwards, the performance implications of the format dissection are examined by benchmarking the HDF5 VOL plugins. The performance results of an early database engine prototype evaluation are discussed next. Finally, both engines (JULEA-KV and JULEA-DB) are evaluated and compared against BP3, BP4 and BP5; first with a real-world application and then using a custom query application comparing the performance of the native ADIOS2 interface to the DAI interface.
The chapter is split into the following sections:

- *Section 7.1: contains the results of the JULEA benchmarks*
- *Section 7.2: covers the performance of both HDF5 plugins*
- *Section 7.3: provides the evaluation of an early database prototype, published in [Duwe and Kuhn, 2021a]*
- *Section 7.4: presents the performance of the final JULEA engines*
- *Section 7.5: shows the post-processing evaluation of the DAI.*

7.1. Benchmarking of JULEA

This section focuses on the performance of the different JULEA backends. As they build the foundation for the JULEA engines, it is important to get a baseline of each backend's potential performance. Afterwards, the BlueStore backend is evaluated and whether it is a suitable replacement for the current object backend in JULEA.

7.1.1. Performance of JULEA Backends

The measurements were performed using the `ants` cluster of the institute for intelligent cooperating systems (IKS) at the Otto-von-Guericke University Magdeburg. It is a joint project of different groups sharing their respective resources. As a result, there are different types of nodes. The ones used in this evaluation are larger nodes like `ant10` to `ant12` and smaller nodes like `ant13` to `ant20`. Their hardware is listed below:

- ant10:
  - CPU: 2x AMD Epyc 7543 2.8 GHz with 32 cores each
  - SSD: Intel SSDSC2KB960G8 (D3-S4510 Series), 960 GB, SATA 6GB/s
    Max. performance as per specification: 510 MB/s (write), 560 MB/s (read) [1]
  - RAM: 1024 GB
  - Network: 100GbE and 100ib Mellanox
- ant13:
  - CPU: AMD Epyc 7443, 2.85 GHz with 24 cores
  - HDD: Western Digital 500 GB WD5000AAKS
    Host to/from drive: 126 MB/s sustained [2]
  - RAM: 128 GB
  - Network: 100GbE and 100ib Mellanox

All benchmarks were run 10 times. The plots show the mean value over all runs.

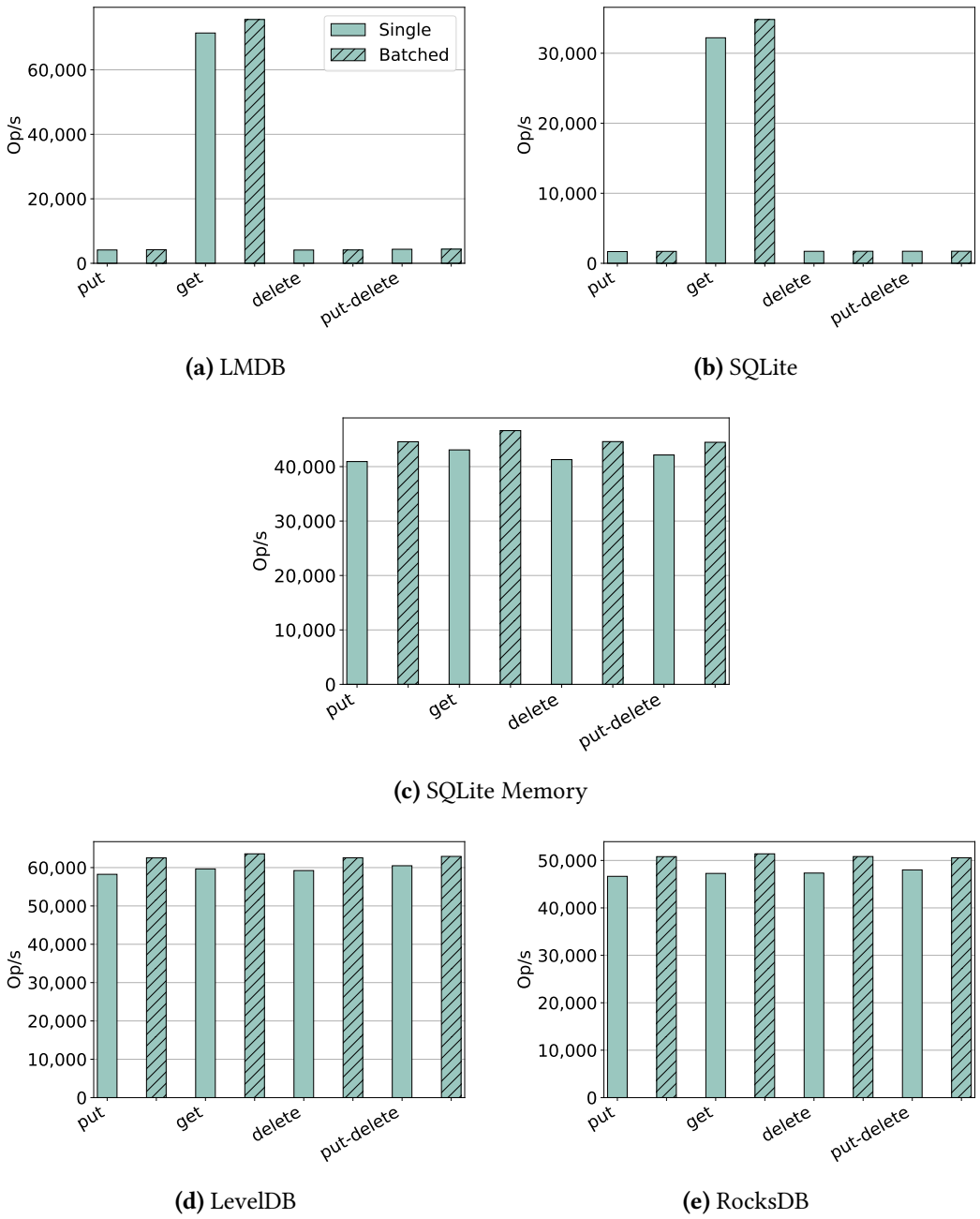
Key-Value Store Benchmarks

The used benchmark is part of JULEA[3]. It measures the operations per second that can be achieved for `put`, `get` and `delete` as well as their batched versions which group multiple operations together. Batching reduces the network overhead. The studied backends are LevelDB, LMDB, SQLite, SQLite as an in-memory database and RocksDB. The benchmark results run on `ant10` are shown in Figure 7.1.

The best overall performance has LevelDB offering both good write and read performance as illustrated in Figure 7.1d. RocksDB is very balanced as well (see Figure 7.1e). In contrast, LMDB focuses on offering a very high read throughput at the cost of other operations like writing, as highlighted in Figure 7.1a. While it achieves around 71,000 and 75,000 (batched) operations per second for reading, the write performance is very low, with 4,100 and 4,200 operations per second. The results for LMDB are not surprising, as this is what LMDB is optimised for. Figure 7.1c shows that SQLite achieves a reasonable performance when run in RAM. However, this is not suitable for all applications. Therefore, SQLite was also evaluated as a persistent option (see Figure 7.1b). The batched read performance of 35,000 operations per second is less than half of the results for LMDB. Unfortunately, the batched write reaches only about 1,600 operations per second.

The option most suitable for the JULEA-KV engine is LevelDB. While the read performance of LMDB would be very beneficial for post-processing queries, the write performance would notably slow the engine down, for example, during the writing of benchmarks. As I/O is already a bottleneck, lessening the throughput even more, is not a sensible choice.

[1] `https://ark.intel.com/content/www/us/en/ark/products/134912/intel-ssd-d3s4510-series-960gb-2-5in-sata-6gbs-3d2-tlc.html` *accessed: 09.10.2022*
[2] `https://products.wdc.com/library/SpecSheet/ENG/2879-701277.pdf` *accessed: 09.10.2022*
[3] `https://github.com/julea-io/julea/tree/master/benchmark`



(a) LMDB

(b) SQLite

(c) SQLite Memory

(d) LevelDB

(e) RocksDB

Figure 7.1.: The performance results for JULEA's key-value store backends on ant10. For *SQLite Memory*, SQLite was run as an in-memory database in the RAM. The data is therefore not persisted. The hatching indicates batched operations.

Database Benchmarks

Along with the database client and backend, corresponding database benchmarks were developed as well[4]. The benchmark functions related to inserting entries are shown in Listing 7.1[5]. For `insert`, `delete` and `update` different versions are examined. The main focus is on comparing batched to unbatched operations as well as multiple indexing strategies: no index (Lines 1-2), a single index (Lines 3-4), indexing everything (Lines 5-6) and a mixed approach (Lines 7-8).

```
1 j_benchmark_add("/db/entry/insert", benchmark_db_insert);
2 j_benchmark_add("/db/entry/insert-batch", benchmark_db_insert_batch);
3 j_benchmark_add("/db/entry/insert-index-single", benchmark_db_insert_index_single);
4 j_benchmark_add("/db/entry/insert-batch-index-single",
      ↪ benchmark_db_insert_batch_index_single);
5 j_benchmark_add("/db/entry/insert-index-all", benchmark_db_insert_index_all);
6 j_benchmark_add("/db/entry/insert-batch-index-all", benchmark_db_insert_batch_index_all);
7 j_benchmark_add("/db/entry/insert-index-mixed", benchmark_db_insert_index_mixed);
8 j_benchmark_add("/db/entry/insert-batch-index-mixed",
      ↪ benchmark_db_insert_batch_index_mixed);
```

Listing 7.1: Benchmark functions the evaluate the performance of inserting entries into the database.

JULEA currently supports two database backends, namely SQLite and MySQL/MariaDB. Given the experience from the key-value benchmarks, SQLite was only evaluated as an in-memory database.

MariaDB To ease the setup of the MariaDB server, it is run in a Singularity container. The used image is from the Docker hub[6]. The call to start the container can be found in Listing B.4 in the appendix.

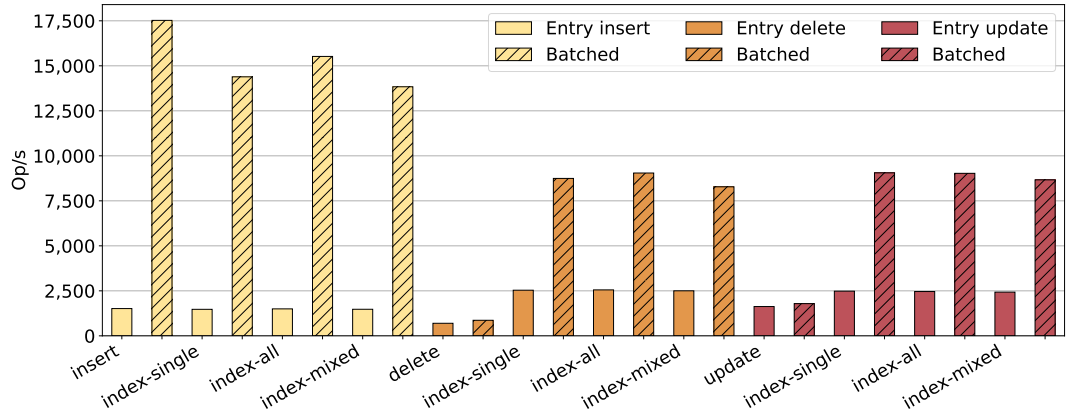
The best insert performance for MariaDB is around 17,000 operations per second when using a single index and batching, as shown in Figure 7.2a. The update performance is best when using batches and achieves around 9,000 operations per second. MariaDB achieves, at most, a read performance of 9,000 operations per second when using an iterator with a single index, as depicted in Figure 7.2c.

SQLite The results for the entry-related performance of SQLite are shown in Figure 7.2b. Like MariaDB, SQLite performs best when inserting entries using batching and a single index, achieving about 220,000 operations per second. SQLite reaches about 110,000 operations per second for updating entries with a single index and batching. While the entry-related functions are more than a factor of 10 better than MariaDB, reading with an iterator is only four times better, with about 32,000 operations per second.

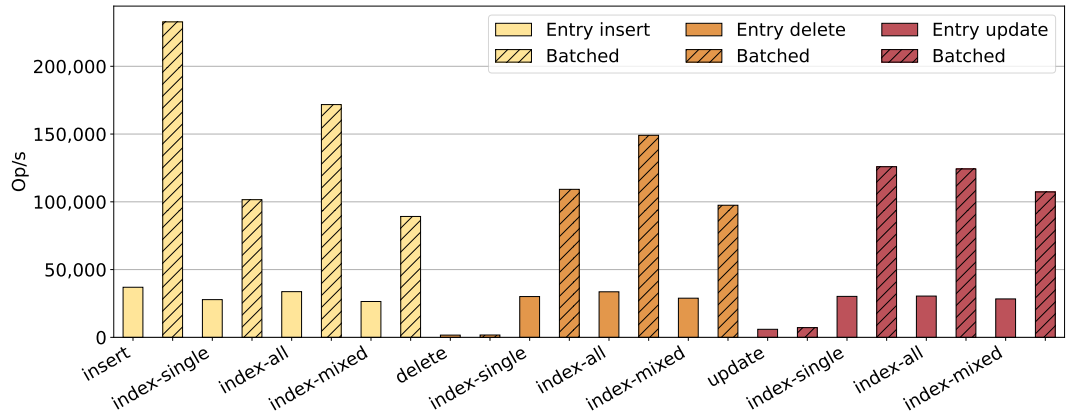
[4] The implementation was done by Michael Straßberger.

[5] The complete benchmark can be found here: `https://github.com/julea-io/julea/tree/master/benchmark/db`

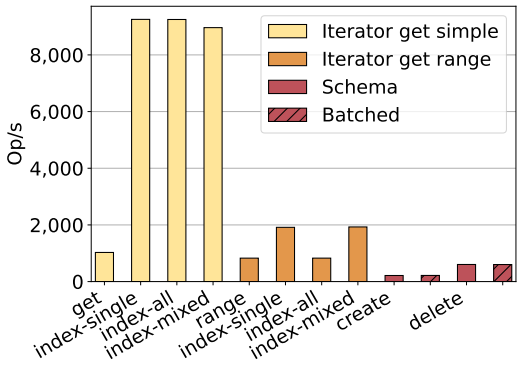
[6] `https://github.com/MariaDB/mariadb-docker`



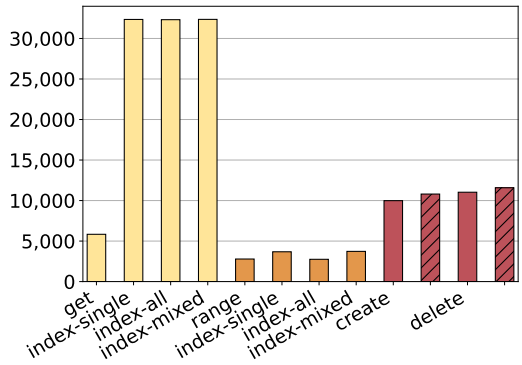
(a) MariaDB: Entry-related performance



(b) SQLite: Entry-related performance



(c) MariaDB: Iterator-related performance



(d) SQLite: Iterator-related performance

Figure 7.2.: The performance of the JULEA database backends using MariaDB (a,c) and SQLite (b,d) measured in operations per second. The results are grouped according to functions.

Object Store Benchmarks

In the following, the performance of the object storage backend on the ants cluster is examined. As mentioned previously, JULEA does not use a traditional object store. Instead, the objects are mapped to files in the file system using POSIX. This is not ideal because several features of POSIX hinder good throughput in a distributed and parallel system. One common problem is the strict semantics. Back when POSIX was introduced, it was designed as a local file system and enforcing immediate visibility of changes, for example, was a suitable approach. However, these requirements come with many drawbacks and performance problems in cluster systems today.

In Figure 7.3, the results for the object benchmark are shown. Two different setups are compared:

- Global Ceph:
  The Ceph installation of the ants cluster uses three replicates for the home directories and is run on 4 servers and 1 workstation [7] In the first case, the path to the object backend points to a folder in the home directory, which is mounted in the global Ceph. The results show that batching helps the performance a lot. The write performance of the object when using Ceph is about 190 MB/s and 1.5 GB/s for batched writing. The read has a throughput of 190 MB/s and 965 MB/s for batched reads.

- Local storage:
  For the second case, the object backend writes to the locally attached storage on the compute node. It has a write throughput of 275 MB/s, and 2.4 GB/s for batched writes. The read throughput is 290 MB/s and 2.9 GB/s for batched reads.

It is interesting to note that the write performance for Ceph is better than the read performance, whereas the local reads are faster than the writes.

Summary of the Backend Evaluation

In the following, the most important insights from the JULEA backend evaluation are summarised:

- LevelDB is the best key-value store backend overall. If only the read performance is important, LMDB is worth considering.
- The performance of the database backends is improvable. While SQLite Memory offers sufficient performance, the volatility of the data makes it unfit as the main file metadata backend.
- The insert, delete and update performance of the databases are best when used with operation batching and a single index.

[7] Used hardware: 3 servers have 4 x 4TB NVMe SSDs, 1 server has 4TB NVMe and 8 x 3.84 TB NVMe SSDs. The workstation has 6 x 3,84 TB SATA SSDs. The servers are connected over 100GbE, and the workstation with 2 x 10GbE. This network connection to the workstation is the bottleneck, according to the system administrator (Michael Preuss).

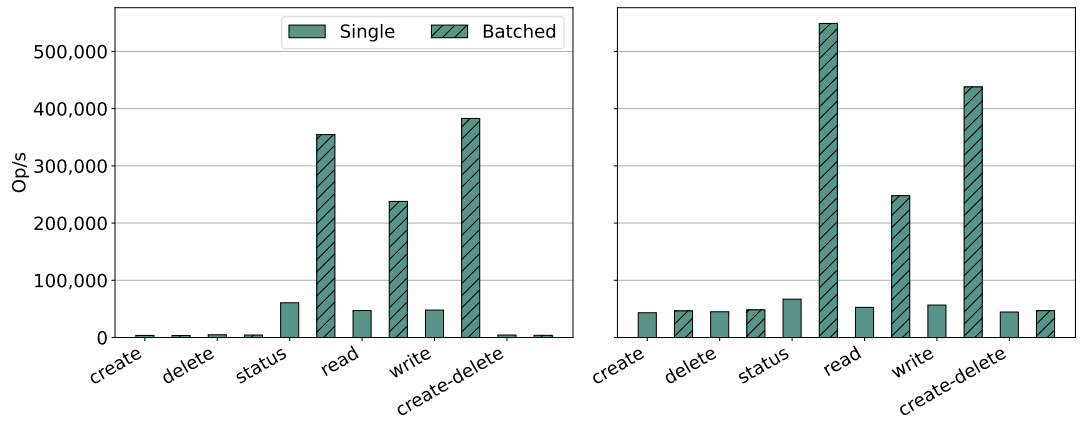


Figure 7.3.: The performance of the JULEA object backend using the global Ceph (left) and the node-local storage (right) was measured in operations per second. The hatching indicates batched operations.

- The performance of the object store is reasonable when batching is enabled. Unfortunately, this is not possible for all applications, depending on their I/O and semantics requirements.

7.1.2. Improving JULEA's Object Store - Examining BlueStore

The evaluation in this section was published in [Duwe and Kuhn, 2021b]. The drawbacks of having an object store built on top of POSIX were discussed previously. The resulting performance has been discussed in the previous section, which leads to the question of how a possible alternative could look like. Studying related work inspired the attempt to use BlueStore in JULEA.

So, the main question for the work with BlueStore was whether it would be possible to use BlueStore without a Ceph cluster. Thus, the aim was to design and implement a proof of concept prototype. Therefore, it is currently not very optimised and leaves much untapped potential but so does the current object backend.

Setup

In the following, the setup is explained first, and then the results of the BlueStore evaluation are presented. JULEA, with the new BlueStore backend, was compared against JULEA using the POSIX-based object store. Only one compute node was used to avoid conflicts between Ceph and the NFS of the test cluster.

Hardware The compute node is equipped with 4 × AMD Opteron 6344, 128 GB of main memory and a 1 TB WDC WD1003FBYZ-010FB0 HDD (with a maximum throughput of roughly 130 MB/s). For a baseline, the HDD's writing performance was measured to be 109 MB/s using `dd` with a block size of 4k and the sync flag:

```
dd if=/dev/zero of=dummy bs=4k count=1000 oflag=sync status=progress
```

Listing 7.2: dd call for HDD performance baseline

Software For the purpose of reproducibility, the versions of the used software are listed below:

- OS + Kernel: Ubuntu 4.15.0-118-generic
- Ceph: master from 2021-02-13[8]
- JULEA BlueStore: from 2021-02-23[9]
- Compiler: GCC 9.3.0

Results

As mentioned before, this evaluation is meant as a first step towards understanding BlueStore and its application areas. The results for both the BlueStore and the POSIX backend can be found in Figures 7.4 and 7.5. The measurements were performed for various durations ranging from 1 to 512 seconds to rule out variability over time. As none were found, the results are presented for a duration of 4 seconds per block size over a total of 10 runs for block sizes from 4 KiB to 4,096 KiB.

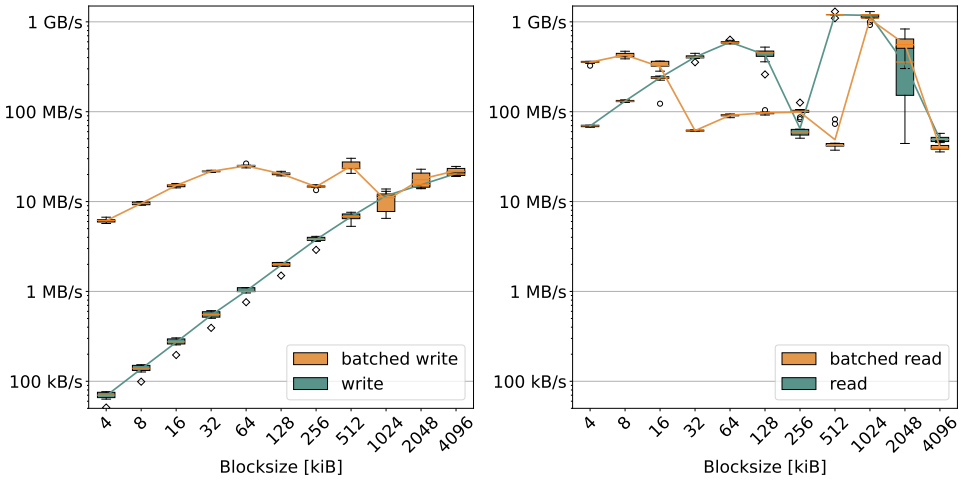
To avoid measuring only the cache, for the runs in Figure 7.4 JULEA's storage semantics parameter was set to `storage=safety`, meaning that every operation is directly synced to the HDD. However, as this is very extreme behaviour, the batched JULEA operations were also evaluated. Depending on the block size, a sync is performed after every 10.000th operation for block sizes up to 256 kiB, respectively 1.000th operation for all larger block sizes.

This difference does not impact the performance, as can be seen in all plots. As the results for synced and unsynced reading are very similar for both backends, only the synced results are shown for brevity's sake. The resemblance is no surprise, as the explicit syncing in JULEA does not considerably change the internal behaviour for reading.

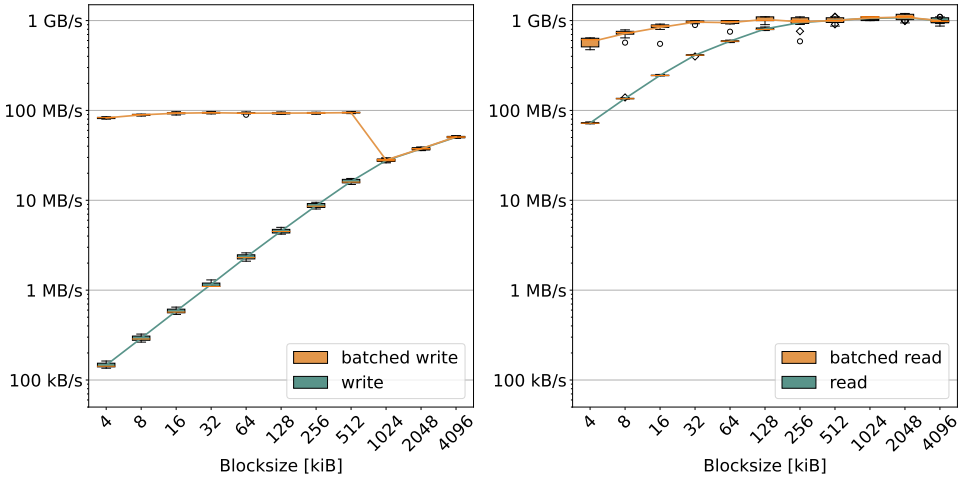
Discussion Synced writes without batching behave similarly in both POSIX and BlueStore. However, POSIX achieves about double BlueStore's performance, with POSIX reaching a peak performance of 50 MB/s for 4,096 KiB blocks, whereas BlueStore only achieves 20.5 MB/s. Batching improves the synced writes for BlueStore and POSIX but only up to a block size of 512 kiB with a peak of 24.9 MB/s and 94 MB/s, respectively. The significant drop afterwards to 10.3 MB/s for BlueStore and 28 MB/s for POSIX is unexpected and does not correspond to the change in batch size mentioned earlier. The exact reasons still have to be investigated.

The unsynced results have their peak performance at either 512 kiB or 1024 kiB for all operations for both backends. There are caching influences that could not be circumvented yet,

[8] https://github.com/ceph/ceph/commit/71c33b8466d3af08d285896f42c9f12075d44091
[9] https://github.com/Bella42/julea/commit/bec331dae4e2c7c7183e6e45047368b8c28aa971



(a) BlueStore backend with syncing

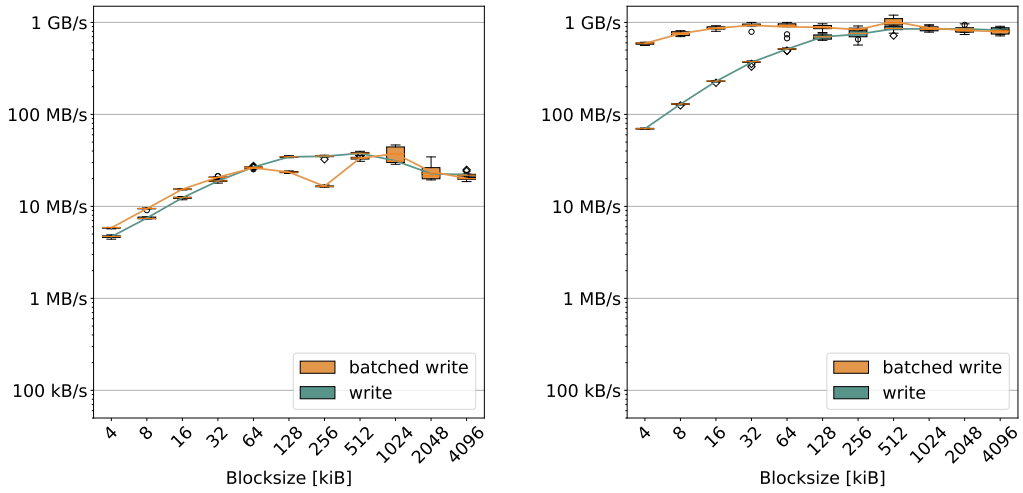


(b) POSIX backend with syncing

Figure 7.4.: Throughput for writing and reading with explicit syncing for BlueStore and POSIX. All operations are evaluated with and without batching. The line plots show the mean throughput.

as can be seen for the reading performance. The BlueStore results vary wildly, which again points to cache interference. Nevertheless, the results are still meaningful as real workloads will also encounter cache interference.

Conclusion and Future Work for Object Storage in JULEA Studying BlueStore revealed that it can be used as a semi-standalone object store. The evaluation showed that while there is still much untapped potential, the simple BlueStore backend works well. Further decoupling from Ceph will be targeted in the future so that, ideally, not all Ceph dependencies need to be installed. Also, the in-depth Ceph functionality needs to be used to optimise the be-



(a) BlueStore backend

(b) POSIX backend

Figure 7.5.: Throughput for writing without explicit syncing for BlueStore and POSIX. All operations are evaluated with and without batching. The line plots show the mean throughput.

haviour. Furthermore, it would be interesting to evaluate BlueStore's suitability to run across several nodes with JULEA.

7.2. Performance of Format Dissection using HDF5

This section shares some content, for example, the general description of the benchmark with [Kuhn and Duwe, 2020]. The measurements themselves are unpublished.

Previous performance measurements of the format dissection using HDF5 have been published [Kuhn and Duwe, 2020]. They were performed using the HDF5 JULEA benchmark. The focus was on using key-value stores to store file metadata. Also, a primitive DAI prototype was studied. The purpose of the following evaluation is to demonstrate that the format dissection does, in fact, work for other formats besides ADIOS2. Additionally, it gives a short summary of the performance of both JULEA VOL plugins for a larger number of backends. The previous measurements used only the LevelDB backend with multiple configurations using different semantics [Kuhn and Duwe, 2020]. Therefore, the performance for multiple KV backends is included.

Setup The setup is the same as previously used for the other JULEA benchmarks, using one of the larger nodes (`ant10` and `ant11`) in the ants cluster[10]. The HDF5 benchmark is part of the JULEA framework as well[11].

7.2.1. JULEA VOL Plugins: KV and DB

The results for both JULEA VOL plugins and native HDF5 are presented in Figure 7.6.

Files The performance of creating and opening empty files varies the most for different JULEA backends and HDF5. LevelDB, RocksDB and SQLite Memory each have a similar performance for creating and opening files. LevelDB achieves around 51,000 and 52,000 operations per second. LMDB reaches around 500 operations per second for creating files and roughly 65,000 for opening them. RocksDB and SQLite Memory are a bit slower. RocksDB can perform 35,000 and 38,000 operations per second, while SQLite Memory reaches around 42,000 operations per second for both. When writing to the local disk, SQLite can perform 1,600 and 30,000 operations for creating and opening files. The native HDF5 format performs worst, with about 150 operations per second for creating files and 140 operations per second for opening the files. The results for the database plugin are similar to the others, especially for SQLite, which reaches about 30,000 operations opening files again. For MariaDB, the absolute results are smaller than expected, given the benchmark evaluation. It is interesting to note that LMDB and SQLite (KV) and MariaDB and SQLite (DB) have a big performance gap between creating and opening files. This can be explained with the respective results for writing and reading data as discussed in Section 7.1. They match the differences for creating and opening files.

Groups For the group benchmark, empty groups are created and opened. While HDF5 only reached 140 operations per second for opening a file, it achieves about 32,000 operations per

[10] 2x AMD Epyc 7543 2.8 GHz with 32 cores each, Intel SSDSC2KB960G8 (D3-S4510 Series), 960 GB, SATA 6GB/s, 1024 GB RAM

[11] `https://github.com/julea-io/julea/tree/master/benchmark/hdf5`

second for opening groups. The main reason is most likely that the creation of groups does not involve the underlying file system. All other backends have a similar performance for creating and opening groups as for the previous case.

Datasets Next, the performance of writing and reading datasets is examined. Two dataset sizes are used, 4 MB (large blocks) and 4 kB (small blocks). In contrast to before, writing and reading datasets involves not only the key-value or database backend but also the object backend, which stores the data itself. As previously mentioned, JULEA currently has only one object backend, which maps the objects to files in the underlying file system.

At first glance, the results for writing and reading 4 MB datasets might seem unintuitive. Typically performing I/O with larger blocks is beneficial. That is indeed the case here as well. However, the plot shows the operations per second, not the throughput. To calculate the throughput, the results have to be multiplied by 4 MB and 4 kB, respectively.

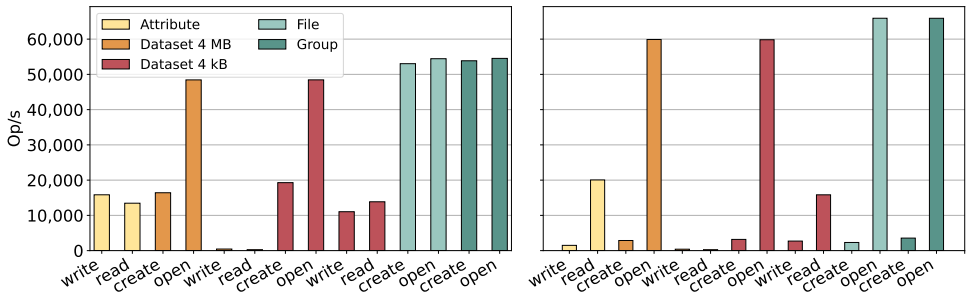
Creating and opening datasets has a similar behaviour for all candidates. Creating datasets is considerably slower than opening a dataset. Interestingly, native HDF5 has the best performance for creating and opening datasets of both sizes.

Performing these operations on the key-value and database backends requires different internal operations. For operations involving datasets, both file metadata and data have to be written, involving both the key-value store or database as well as the object backend. For example, creating a dataset requires creating an object and storing the corresponding metadata in the metadata backend. As the object backend is essentially a wrapper for the underlying file system, the performance is thereby determined by the file system's performance.

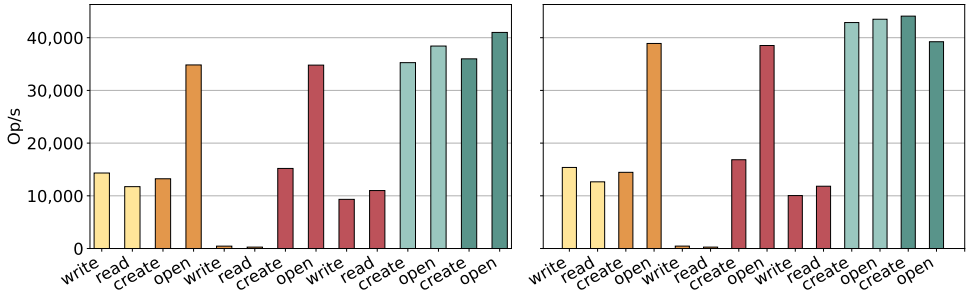
The write and throughput for the KV VOL plugins overall are very similar to the performance related to files or groups. LevelDB reaches 1,890 MB/s for writing 4 MB (large) blocks and 45 MB/s for 4 kB (small) blocks. It has a read throughput of 1,140 MB/s (large blocks) and 57 MB/s (small blocks). LMDB achieves a write throughput of 1,700 MB/s (large blocks) and 11 MB/s (small blocks). The read performance of LMDB is 1,120 MB/s (large blocks) and 64 MB/s (small blocks). RocksDB has a write performance of 1,900 MB/s (large blocks) and 40 MB/s (small blocks), while the read performance is 1,130 MB/s (large blocks) and 48 MB/s (small blocks).

The results for SQLite writing to the local storage and for the in-memory version are closer together than for the previous results. SQLite has a write throughput of 1,160 MB/s (large blocks) and 13 MB/s (small blocks), SQLite Memory of 1,900 MB/s (large blocks) and 37 MB/s (small blocks). The same applies to the read performance where SQLite reaches 1,090 MB/s (large blocks) and 18 MB/s (small blocks), while SQLite Memory achieves 1,130 MB/s (large blocks) and 45 MB/s (small blocks). They are so similar, even though SQLite Memory is used in RAM because the throughput for writing and reading is dominated by the performance of the object backend handling the actual data. The respective metadata setup is, therefore, secondary.

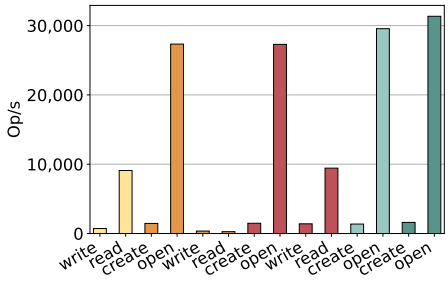
Attributes The HDF5 benchmark in JULEA only measures the write and read performance of attributes, as they are typically not used empty. The attribute creation is included in the writing, while the opening is in the reading [Kuhn and Duwe, 2020]. The performance for the



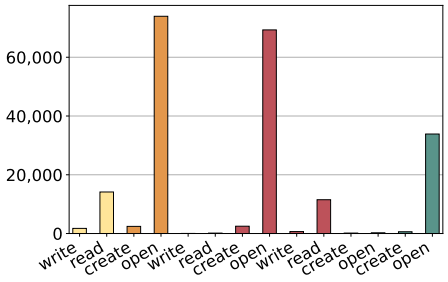
(a) JULEA-KV VOL: LevelDB (left) and LMDB (right)



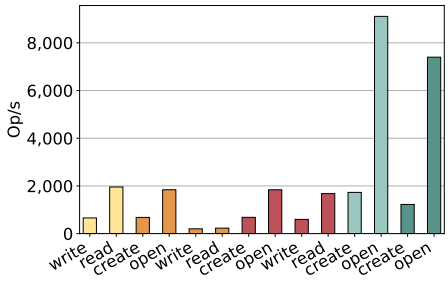
(b) JULEA-KV VOL: RocksDB (left) and SQLite Memory (right)



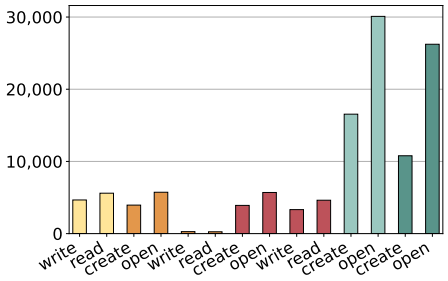
(c) JULEA-KV VOL: SQLite



(d) Native HDF5



(e) JULEA-DB VOL: MariaDB



(f) JULEA-DB VOL: SQLite

Figure 7.6.: Performance of the HDF5 VOL JULEA-KV plugin using different backend implementations.

different backends across both plugins is consistent with the write and read performance of the small blocks. Again, LMDB SQLite and native HDF5 display a larger difference between writing and reading performance.

7.2.2. Simple DAI

In order to evaluate the potential benefits of the format dissection on querying, the benchmark was extended. For this, 10,000 attributes have been created. To simulate post-processing operations, 10 HDF5 files have been written with 10,000 attributes each. The task is to read all attributes from all files, which resembles the evaluation of ensemble runs, where data is examined across files. It now contains three options to analyse the query performance:

Native HDF5 Interface The first approach to reading the attributes is using the native HDF5 interface to access the attributes. For HDF5, the attributes are stored in Ceph. The VOL plugins store the attributes in the corresponding JULEA backends. This provides no performance benefits apart from those enabled by JULEA's backends themselves because all I/O calls have to pass through the VOL plugin.

DAI-Get This option accesses the key-value store directly via JULEA's key-value client, requesting each key-value pair separately. The value, a serialized version of the HDF5 attribute, is deserialized and returned to the application.

DAI-Iterate The key-value store or the database is accessed directly through the respective clients. The JULEA iterator interface is used, which enables querying all attributes of a backend at once.

Results The following measurements used the same setup as for the other JULEA benchmarks, that is, one of the larger nodes (`ant10` and `ant11`) in the ants cluster. Some filtering is performed to identify and process only those values belonging to attributes. Once a matching value is found, it is again deserialized and returned to the application.

Figure 7.7 shows the results of reading a total of 100,000 attributes spread over 10 files, with each attribute having a size of 4 KiB.

The performance using native (the HDF5 interface) is plausible, given the results of reading attributes with HDF5 and the different VOL plugins. MariaDB has the slowest reading performance, with 1,900 operations per second, while LMDB achieves around 19,000 operations per second. DAI-get achieves between 25,000 operations per second (SQLite) and 60,000 operations per second (LMDB) for all key-value store backends, which aligns with the results of the backend benchmarks at the beginning of the chapter. Using DAI-iterate dramatically improves the performance from 1,900 to 33,000 operations per second because the I/O calls do not have to pass through the entire HDF5 library stack. The best performance for DAI-iterate is 338,000 operations per second for LMDB, which is an improvement by a factor of 5 compared to DAI-get. In comparison to the performance of the native interface, an improvement of a factor of 17 can be observed.

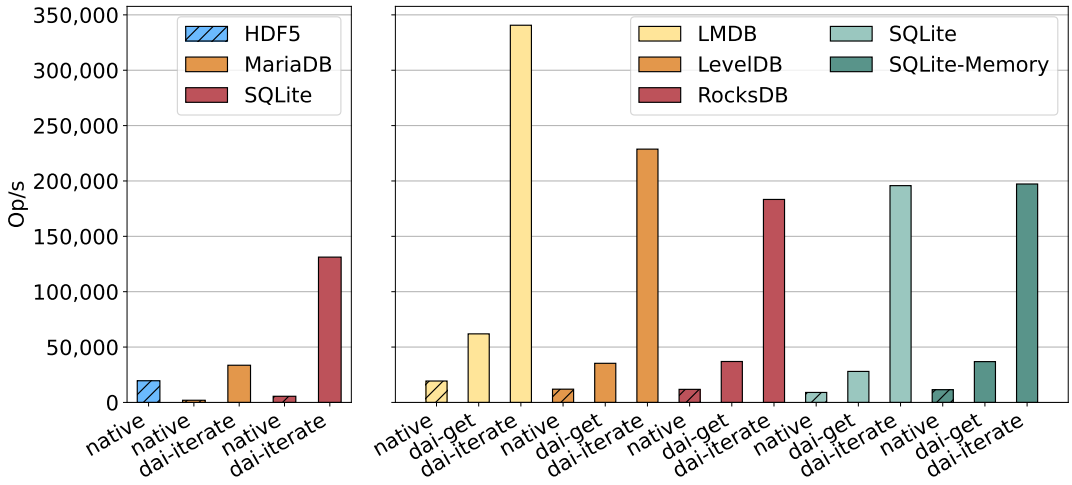


Figure 7.7.: Performance a simple post-processing scenario for HDF5 and both JULEA VOL plugins. The task is to read 100,000 attributes across 10 HDF5 files. *Native* (hatched) means that the original HDF5 interface is used to access the data. *Dai-Get* and *DAI-iterate* directly communicate with the JULEA clients. The results for HDF5 and the database VOL plugin are shown on the left, while the results for the key-value VOL plugin are displayed on the right.

Summary of the HDF5 Benchmarking

It has to be kept in mind that JULEA's key-value interface provides only basic operations such as put, get and iterate. Specifically, searching for a particular attribute or all attributes matching a certain query requires iterating over all attributes and selecting all appropriate ones. Besides the used configurations, other combinations may offer valuable insights. For example, it is possible to combine server-side backends with client-side ones, which could be used to build hybrid configurations: While file metadata can be stored on a dedicated node using a key-value backend, a client-side POSIX object backend can make use of existing infrastructure, such as an existing parallel file system [Kuhn and Duwe, 2020].

In conclusion, the VOL plugins using JULEA's key-value and database backends achieve a competitive performance compared to the native HDF5 when dealing with files, groups, datasets and attributes. Overall, LevelDB reaches the best performance of all key-value candidates. The evaluation of the simple DAI prototype demonstrated that the format dissection can be beneficial if the accesses do not need to use the I/O interface and thus do not have to pass through the entire library. If the respective JULEA client can be used instead, an improvement of a factor of 3 is possible. When the operations can be accessed at once using an iterator, the performance can increase by a factor of 17 compared to the native interface.

7.3. Evaluation of Early ADIOS2 Engine Prototype

The following evaluation was published in [Duwe and Kuhn, 2021a]. There are only minor changes to the text. The most important difference is Figure 7.9c which uses the same dataset as the subfigure above. It was included to highlight the wide variation in the BP engines' reading performance that is hidden in the published depiction.

The performance of an early JULEA-DB engine prototype using SQLite and MariaDB was compared to the BP3 and BP4 engines to validate the general approach. The main focus was not necessarily to achieve a competitive performance but to illustrate the value of the format dissection for analysing and post-processing. However, as I/O constitutes one of the most severe bottlenecks in HPC systems, the writing performance is still very relevant.

7.3.1. Setup

The evaluation has been performed on a moderately-sized research cluster at group Scientific Computing at the University of Hamburg, consisting of four compute nodes and a Lustre file system with ten nodes.

Each compute node is equipped with 4 × AMD Opteron 6344, 128–256 GB of main memory and a 1 TB HDD (with a maximum throughput of roughly 130 MB/s). The Lustre system uses 10 × 2 TB HDDs for a total capacity of 20 TB, while the metadata is stored on a single 160 GB SSD for fast access. The compute nodes are connected to the Lustre nodes via 1 Gbit/s Ethernet but also have a 40 Gbit/s InfiniBand connection with each other.

Since the compute nodes are not equipped with SSDs, SQLite databases will be put into the main memory. MariaDB was run on one of the compute nodes' HDDs, though. 1, 2 and 4 compute nodes were used for the distributed evaluation.

Application The HeatTransfer application from the ADIOS2 examples was used to evaluate the write and read performance. It solves the 2D Poisson equation using finite differences for the temperature distribution in homogeneous media. The choice fell to this application because, despite the statements by the ADIOS2 developers [Godoy et al., 2020], it is not as frequently used for the entire end-to-end workflow. For example, the popular earth system simulations SPECFEM_3D [12] and E3SM[13] either write only boundary conditions with ADIOS2 or do not use ADIOS2 for reading. Therefore, post-processing tasks cannot be evaluated using ADIOS2 then.

The parameters and the used values for the HeatTransfer application are listed below:

- **Configuration file** to set the engine. BP3, BP4, JULEA-DB MariaDB and JULEA-DB SQLite were evaluated.
- **Total number of MPI processes** to write and read: 1, 3, 6, 12, 24 and 48 processes per node (with 1–4 nodes).
- **Division of processes** in the x-dimension (N) and in the y-dimension (M). N×M must equal the total number of used MPI processes.

[12] `https://github.com/geodynamics/specfem3d_globe`
[13] `https://github.com/E3SM-Project/E3SM`

- **Local array size** (nx / ny) in x/y-dimension per processor: 1024, 2048 and 4096
- **Number of steps** to write and read: 10
- **Number of iterations** has no influence on the I/O time. It only impacts the computation time and the results.
- **Resulting data size** is: $nx * ny * number\_steps * nproc\_per\_node * number\_nodes * type\_size$.
  Largest run is $4096^2 * 10 * 48 * 4 * 4 \approx 129$ GiB

Measurement Setup The data rates shown in Figures 7.8, 7.9a and 7.9b are the mean values of the individual I/O times of each process per step over all steps. Only the time it took to write and read was measured. In detail, the difference between the time right before `Put/Get` and right after `EndStep` was computed. The initialisation overhead of the ADIOS2 library is therefore not considered in these rates.

In order to avoid only measuring the system caches, the caches were dropped in between writing, reading and querying in every configuration. Also, it was made sure to explicitly sync the data so that it is actually written in the measured time frame. In the case of JULEA, this means synchronising the writing of every block, i.e. the data of each process. For the BP engines, the POSIX file transport was adapted to call flush at the end of each step. This means that the BP engines are synced less than JULEA, which needs to be considered when comparing the results.

Lustre vs. Local HDD Measurements on one node are included to get a baseline to compare the different setups. Note that ten storage nodes and only two to four compute nodes are used. When using BP3 and BP4, the output is written to the Lustre storage nodes. However, when evaluating the new JULEA-DB engine, they cannot be used as Lustre is replaced with JULEA to simplify the I/O stack, as discussed in the introduction. In order to avoid adding both overheads by using JULEA on top of Lustre, the JULEA engine was evaluated on the compute nodes and, therefore, on different hardware. So another scenario was measured where BP3 and BP4 wrote to the local HDD on one node to provide a basic comparison between using the local HDD to using the Lustre storage nodes with the same engines.

7.3.2. Write and Read Performance

The results for writing and reading different matrix sizes on one node are shown in Figure 7.8. There are several important aspects to note. First, the read performance decreases for all evaluated configurations with an increase in data size, which is unexpected as larger data blocks usually produce lower management overhead and are preferable. In comparison, the write performance is relatively consistent for all engines over the different array sizes.

BP3 & BP4 Performance The decrease in reading performance is the most notable for BP3 when using Lustre, which starts at a data rate of roughly 110 MB/s for 48 processes with a local array size of 1024 per process. It ends at around 60 MB/s for a local array size of 4096. For BP3, writing to the local HDDs is about as fast as writing to Lustre. This is not the case for BP4. BP4 has the worst performance for all configurations on one node. Since BP4 is designed to make

heavy use of aggregation over a considerably larger number of nodes, this is not the setup from which to expect peak performance. However, as mentioned previously, this setup was included to estimate the influence of using different storage nodes for the engines. Lustre's reading performance is about twice as fast as the local ones because it distributes the data over 10 nodes which eases parallel access. Also, the enforcement of syncing impacts only the writing, not the reading, and therefore, does not prevent Lustre from making heavy use of techniques such as read-ahead.

JULEA Performance In contrast, for the JULEA configurations, the read performance is worse than the write performance for the larger array sizes. However, the data rates are still better than BP4's when using the local HDD. This suggests that the access pattern to the local HDD is not ideal and obviously limited in terms of parallel accesses in contrast to the ten storage nodes of Lustre. Both reading and writing performance for JULEA unsurprisingly benefit from running the database in the RAM in the case of SQLite compared to the MariaDB server running on the HDD.

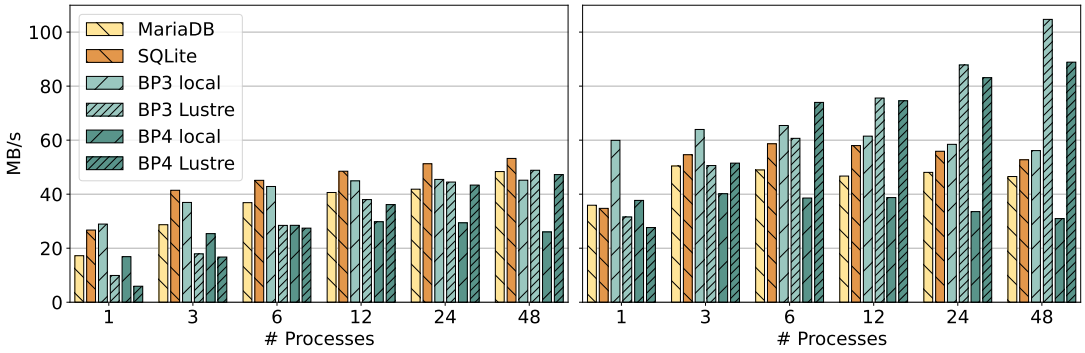
Scaling The consistent data rate over the number of processes might be irritating at first look. Intuitively, one expects an increase with the increase of used processes. However, due to the nature of the application, the problem size increases with the number of processes. In ADIOS2, the number of processes equals the number of written blocks per step. The size of each block is determined by the local array size, which is the same in the complete sub-figure. Thus, doubling the number of processes is also doubling the number of blocks and, therefore, effectively doubling the total data size. With this in mind, the data rates imply a linear or super-linear speed-up for JULEA and BP, respectively.

For 2 and 4 nodes, the behaviour for different local array sizes is comparable to that for one node. So, in the following, the focus lies on the array size of 4096 because, for HPC, larger data sizes are more prominent and, therefore, more critical to support. In Figures 7.9a and 7.9b the data rates for two and four nodes are shown. The BP data rates grow as expected by a factor of two, respectively, four. JULEA, however, only achieves an improvement by a factor of two. This is due to the current rigorous data syncing as well as the overhead of the TCP/IP network stack and the object store being emulated by POSIX.

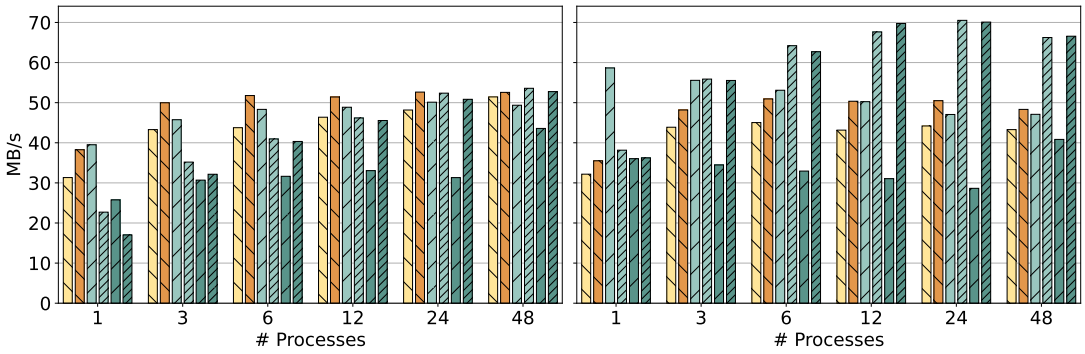
The local array size of 1024 leads to a drastically higher reading performance with Lustre, as shown in Figure 7.8a. This effect becomes more notable for more nodes, leading to a maximum data rate for BP3 with 192 processes of 370 MB/s. As mentioned, this is due to Lustre's read-ahead and the larger amounts of storage nodes.

7.3.3. Query Time

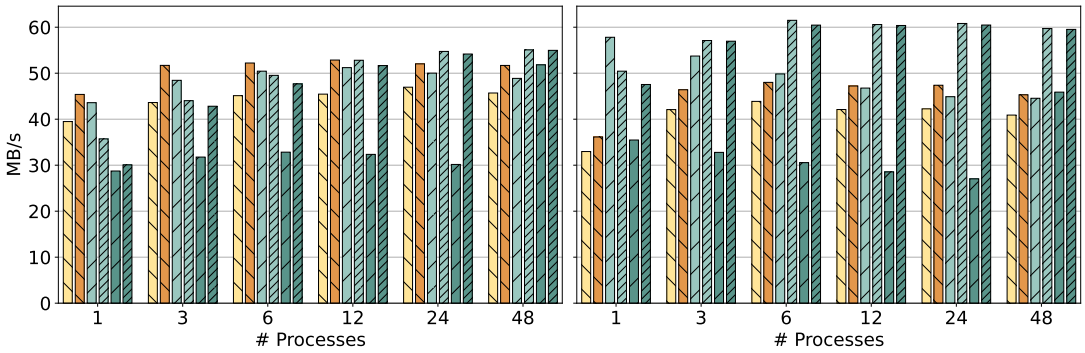
The main focus of this evaluation was to demonstrate the value of the format dissection it has for querying and post-processing. For this, two query applications were developed, ADIOS2-Query and JULEA-Query. The question posed in our example query is: *What block does have the largest difference between its mean value in step 1 and step 5?*



(a) Write and read performance for a matrixsize of 1024^2.

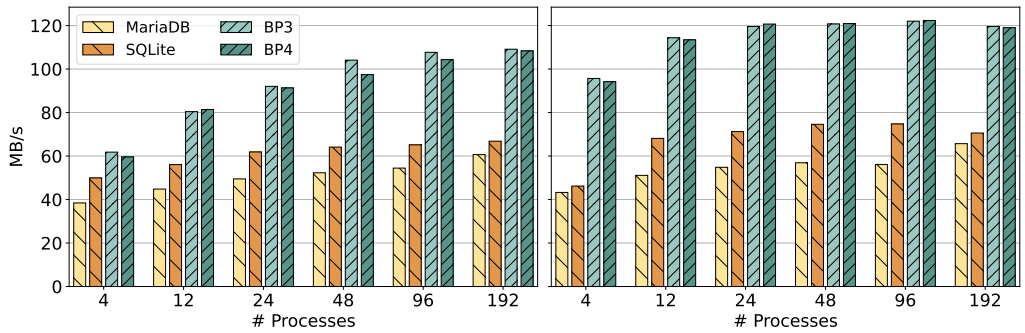


(b) Write and read performance for a matrixsize of 2048^2.

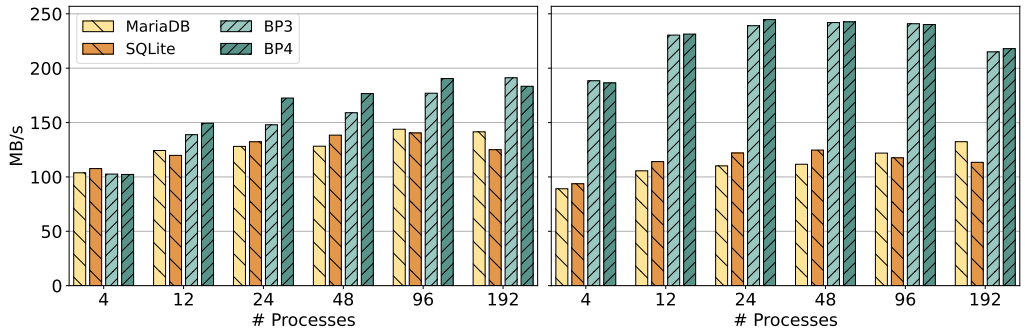


(c) Write and read performance for a matrixsize of 4096^2.

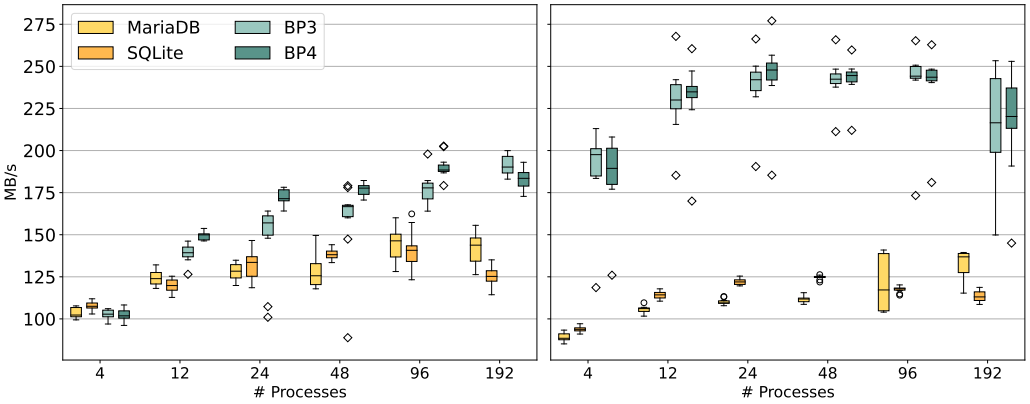
Figure 7.8.: Write and read performance for 1 node with 1 to 48 MPI processes. The matrix size is increased from 1024^2 (top) over 2048^2 (middle) to 4096^2 (bottom). For BP3 and BP4 (light and dark teal) two scenarios are measured each: 1) I/O to and from the local HDD (coarse-grained hatching) of the node and 2) to the system's Lustre (fine-grained hatching). The JULEA-DB engine is measured using a MariaDB (yellow) server running on the local HDD and also with SQLite (orange) running in RAM as the cluster does not have SSDs.



(a) Write and read performance for 2 nodes. JULEA uses 2 compute nodes and 2 separate storage nodes (s. Figure 2.11).



(b) Write and read performance for 4 nodes. JULEA uses all nodes for computing and storage.



(c) Write and read performance for 4 nodes as a boxplot. JULEA uses all nodes for computing and storage.

Figure 7.9.: Write and read performance for 4 nodes with 1 to 48 processes per node and matrix size of 4096^2. The JULEA setups differ. The results for the JULEA database engine are coloured in yellow (MariaDB) and orange (SQLite). The results for the BP engines in light teal (BP3) and darker teal (BP4). The coarse-grained hatching indicates that local storage was used (JULEA), whereas the fine-grained hatching (BP) notes the engines used the parallel file system. Subfigures (**b**) and (**c**) are derived from the same dataset. The latter has not been published in [Duwe and Kuhn, 2021a]. It is included to highlight the high deviation in the read performance of the BP engines.

ADIOS2-Query This application uses the ADIOS2 API to query the data. To answer the question, it needs to read the complete data of steps 1 and 5 and compute the mean value for each block. It then needs to determine the index of the block with the largest difference between its two mean values. The matching results for the variable steps 1 and 5 are read from the file in Lustre. The measured query time includes the reading of the data selection as well as the computation of the mean values and the difference. The computation took at most 500 microseconds and, therefore, hardly influences the total query time.

JULEA-Query To validate the usefulness of the data analysis interface (DAI) depicted in Figure 7.10, a query application was written that directly interfaces the JULEA storage system. Using the JULEA API, the mean value can be accessed that has been precomputed by the JULEA-DB engine when writing the data blocks. This query application, therefore, does not need to access the object store containing the data. It is sufficient to read the mean values from the respective database and compute the differences accordingly to ADIOS2-Query. The measured time contains the time to read the mean values and the computation time. The time to precompute the means is captured in the writing performance of the JULEA engine.

Query Times Comparison In Figure 7.10, the query time is depicted for 4 nodes and a matrix size of 4096. This configuration was chosen as it is the largest example and, thus, most fitting for an HPC environment. The number of blocks is determined by the number of MPI processes of the writing application. It is, therefore, identical to the labels of the x-axis of the previous figures. The query was only run with one process, as it is unlikely that a typical user will parallelise the post-processing and plotting. The number of used nodes is determined by the configuration used for the write measurements because the write operation has to be executed prior to the query evaluation to provide the necessary input data.

As Figure 7.10 clearly shows, JULEA-Query is incredibly fast and does, at maximum, not even take a second to finish. To allow a direct comparison, ADIOS2-Query was evaluated not only using the BP engines but also the JULEA-DB engine. Using the ADIOS2 API, there is no way to access the additional metadata stored in the database. Therefore, the JULEA-DB query time is comparable to the time it takes to read steps 1 and 5, which also applies to the BP engines. When querying 192 blocks, both SQLite and MariaDB take 0.01s, while BP3 and BP4 need 621s and 601s, respectively. This performance improvement is roughly a factor of 60000 when using the JULEA-DB engine with the DAI layer.

Discussion

In summary, it was possible to show the potential performance improvement for data querying that can be achieved by separating the BP file format into file metadata and file data. By storing it in the appropriate backends in JULEA, the flexibility to introduce new metadata was gained that can be precomputed and stored along with existing metadata, such as the minimum and maximum. However, writing file metadata into a proper database management system can incur performance overhead due to the ACID guarantees in such databases and additional overhead from serialising file metadata in the backend implementation. This is primarily an issue of the chosen HDD-based setup and can be alleviated by putting the database

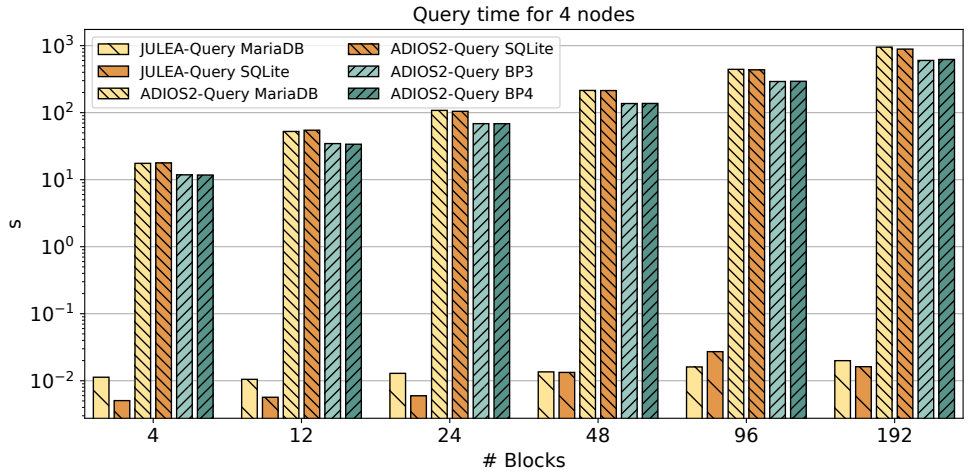


Figure 7.10.: Query time for ADIOS2-Query and JULEA-Query for 4 nodes. Only one process was used to run the query itself. The number of blocks equals the number of processes for writing and reading. The matrix size is 4096^2. Note the log scale for the time (y-axis). The JULEA engines are coloured yellow (SQLite) and orange (MariaDB). The coarse-grained hatching marks the results of the JULEA-Query application, whereas the fine-grained hatching indicates the query application using the ADIOS2 interface.

onto faster storage technologies such as SSDs or NVRAM. Moreover, the object store currently sits on top of a full-featured POSIX file system, which needs to be optimised in the future by using a proper object store backend. Since the engine implementation is a prototype and thus not production-ready, there is still much potential for performance optimisations that will be tackled as part of future work.

The general approach makes it possible to use future HSM systems that integrate different storage hardware, including HDDs, SSDs, and NVRAM. The format dissection makes it possible to map file metadata and data to appropriate storage tiers in such systems. Since file metadata represents a small percentage of the overall data volume, it can be stored on more costly storage devices with lower latencies.

7.4. Performance of JULEA Engines

In the following, both JULEA engines (JULEA-KV and JULEA-DB) are evaluated. The previous results published at the SYSTOR were measured on the cluster of the group Scientific Computing at the University of Hamburg. As this system had some problems that led to the failure of the parallel file system, another system was needed for the final evaluation. Initially, the plan was to use the new supercomputer of the DKRZ Levante instead of the WR cluster. However, currently, there is no container support which is necessary to ease running a MariaDB server. So, the evaluation was instead performed on the cluster of the IKS (Intelligent Cooperating Systems) institute of the Otto-von-Guericke University Magdeburg. Since the differences between the systems include a different file system setup, the measurements had to be repeated to gain a new performance baseline of the system and also the engines.

Setup

In the following, the used setup is described. For better readability, the details required to reproduce the results are listed in Appendix B. This includes, for example, hardware specifics but also the dependencies of the spack packages used to measure ADIOS2 2.8.3.

Hardware In the following, the hardware of the ants cluster is summarised. The operating system used is CentOS8. Two different node types were used: larger nodes like `ant10` to `ant12` and smaller nodes like `ant13` to `ant20`:

- `ant10`:
  - CPU: 2x AMD Epyc 7543 2.8 GHz with 32 cores each
  - SSD: Intel SSDSC2KB960G8 (D3-S4510 Series), 960 GB, SATA 6GB/s
    Max. performance as per specification: 510 MB/s (write), 560 MB/s (read) [14]
  - RAM: 1024 GB
  - Network: 100GbE and 100ib Mellanox
- `ant13`:
  - CPU: AMD Epyc 7443, 2.85 GHz with 24 cores
  - HDD: Western Digital 500 GB WD5000AAKS
    Host to/from drive: 126 MB/s sustained [15]
  - RAM: 128 GB
  - Network: 100GbE and 100ib Mellanox

[14] `https://ark.intel.com/content/www/us/en/ark/products/134912/intel-ssd-d3s4510-series-960gb-2-5in-sata-6gbs-3d2-tlc.html` *accessed: 09.10.2022*

[15] `https://products.wdc.com/library/SpecSheet/ENG/2879-701277.pdf` *accessed: 09.10.2022*

Node Configuration

- Compute Nodes: 1 - 6
  The nodes `ant13` to `ant20` are used as compute notes for the measurements. Due to the cluster utilisation, it was not possible to use all 8 nodes at the same time. Therefore, the largest runs were executed on 6 nodes.

- JULEA Nodes: 1 - 2
  `ant10` and `ant11` are used to run the JULEA servers. While they have more RAM and cores than the other nodes, the important aspect is the local storage they are equipped with. This is because of the file system configuration; users can accidentally put the nodes into a drain stage because of completely filling the local file system. Therefore, having more storage attached locally helps reduce this risk and ensures that a sensible amount of data can be written.

- Allocation: Exclusive
  To minimise the influence of other jobs, the nodes were allocated exclusively, guaranteeing that the node itself is not shared with other users. However, exclusive allocation does not help against interference regarding the network throughput, latency, or the file system's performance.

Software

- File System: Ext4 and Ceph
  The local file system on the cluster is ext4 with a block size of 4096. Ceph is used as the distributed file system holding, for example, the home directories. The BP engines write to Ceph, whereas the JULEA engines write to the local storage of the nodes running the JULEA servers. This setup is chosen to avoid the overhead of another distributed file systems when using JULEA.

- HeatTransfer Application:
  The application remained largely the same as when used for the evaluation of the engine prototype. The small modifications will be explained below.

- ADIOS2 2.7.1:
  The application code is publicly available[16]. Note that the BP engines no longer require fixing, as `flush` is implemented now.

- ADIOS2 2.8.3:
  Unfortunately, neither the master from the official ADIOS2 repo nor the fork of the ParCIO group compiles in 2.8.x on the ants cluster. Since the ADIOS2 spack package can be built for version 2.8.x, it is used. However, some changes had to be made to use it in the evaluation. First, the code required changes to build and install the examples. Otherwise, the HeatTransfer application cannot be used. Second, the source code needs

[16] `https://github.com/julea-io/adios2/tree/master/examples/thesis/heatTransfer` The used commit is `https://github.com/julea-io/adios2/commit/2c6b0aa13f1b36329746d607a816441b7521e4e5`

patching to include the benchmark-related output. Third, the spack package has to be built with `keep-stage` to retain the example binaries. It is also necessary to copy the built spack package to a different directory, as `/tmp` will not be available on the compute nodes.

- JULEA:
  The JULEA version used in the evaluation contains the addition of the DAI to allow a comparison between the HeatTransfer and the query results presented in the next section [17].

- Caching:
  To get a good understanding of a system's performance, it is always desirable to eliminate as many factors as possible that could interfere with the evaluation. Caching is one common feature that can skew measurements. Unfortunately, it is impossible for a normal user to drop the caches on the ants cluster instead of the abu nodes of the WR cluster. However, one might argue that this is no major drawback because the end-user will use the caching and that the results resemble a real run more closely. So, caching effects must be considered when examining the results.

JULEA

- Configuration:
  In contrast to the SYSTOR evaluation presented earlier, the JULEA server is not run on the compute nodes. Instead, one to two separate nodes are used to run JULEA on. There are two main reasons for this. First, the performance of running the JULEA server on the compute nodes has been evaluated already. Also, this configuration illustrates how JULEA can be used if the compute nodes do not have storage attached locally. Second, the current cluster setup is not ideal for running JULEA in a distributed way. Therefore, using two separate nodes for JULEA was chosen to ensure stable measurements.

- Key-Value Backend: LevelDB
  As the JULEA benchmark revealed, LevelDB has the best overall performance for different operations, including writing and reading.

- Database Backend: MariaDB
  MariaDB is the best available option in JULEA to store data in a database. While the performance of SQLite running in RAM is better, the data is not persisted. Therefore, MariaDB was chosen. Furthermore, suppose the JULEA-DB engine and the DAI performance are competitive to the BP engines when using MariaDB. In that case, the approach proves more powerful than if the more performant but volatile solution was selected. MariaDB is run in a `Singularity` container that uses the current docker image.[18]

[17] The code can be found at `https://github.com/Bella42/julea/tree/dai`.
[18] `https://github.com/MariaDB/mariadb-docker`

Application The heat transfer application was used again to evaluate the write and read performance. For details, see the SYSTOR evaluation.

- Constant matrix size: 1024^2
  In contrast to the measurements performed for the early prototype, the matrix size of the HeatTransfer application was not changed to allow focus on other aspects, such as the number of steps and variables. As previously mentioned, the ants cluster has very limited storage space. Therefore, the smallest evaluated matrix size was used to reduce the risk of accidentally forcing nodes into a drained state. Even if the HeatTransfer measurements are guaranteed not to exceed the available space, other users might have also stored data either locally or in the global Ceph file system, which holds the home directories.

- Second variable:
  The original HeatTransfer application only writes and reads one variable holding the temperature. A second variable, namely the precipitation, was added to allow more complex queries in the DAI evaluation. The precipitation results are not computed but pseudo-randomised, ensuring that all relevant criteria stay the same over different benchmark runs. This is necessary to allow a fair comparison for the query evaluation in the next section. For example, if all blocks over a specific threshold need to be read, the number of blocks that fulfil the query has to remain the same regardless of how often the application is run. Otherwise, one engine might need to read 50 blocks and another none.

- Step number: 100
  The step number is increased from 10 for the previous evaluation to 100 now. If, instead, the matrix size is increased, only the object size, not the number of objects and corresponding metadata entries, is changed. The decision to use a larger step number was made to make the setup more unfavourable for the JULEA engines, as a larger number of steps results in more metadata entries. For a fair comparison, configurations cannot be cherry-picked to prove the best performance for the proposed solution.

- Engines: BP3, BP4, BP5, JULEA-KV, JULEA-DB
  This time both JULEA engines are studied and compared to one another but also to all ADIOS2 engines that use the BP format, namely BP3, BP4 and BP5.

Measurements As a result of the vast performance variation observed in the previous HeatTransfer evaluation, the final results are presented using boxplots whenever the data allows a readable depiction. A boxplot shows the minimum, maximum, median and the first (Q1) and third quartiles (Q3), also called 25th and 75th percentiles. The *interquartile range (IQR)* is defined as Q3 minus Q1 and is a common basis for the whisker computation.

There are two main features that the whiskers can represent: First, the minimum and maximum of the data or second, the two data points that fall into the range of 1.5 IQR above Q3 or below Q1. The latter is used in this thesis. The whiskers can have an uneven length as they

end at the last data points in the range. All data points outside the 1.5 IQR range are indicated as outliers.

The results shown in the following figures are the mean write and read data rates averaged over all processes performing I/O in a step for a total of 100 steps. So, each boxplot depicts 100 mean data rates. As for the previous measurements, the initialisation overhead of the ADIOS2 library and JULEA are not included in these rates. The variables are written and read using the deferred mode, which means asynchronous I/O. Therefore, the data is only available after `EndStep` has been called. Deferred I/O is the default and recommended option, allowing ADIOS2 to group calls before the data transport. This option was used not to hinder ADIOS2 mechanisms that would usually be in play.

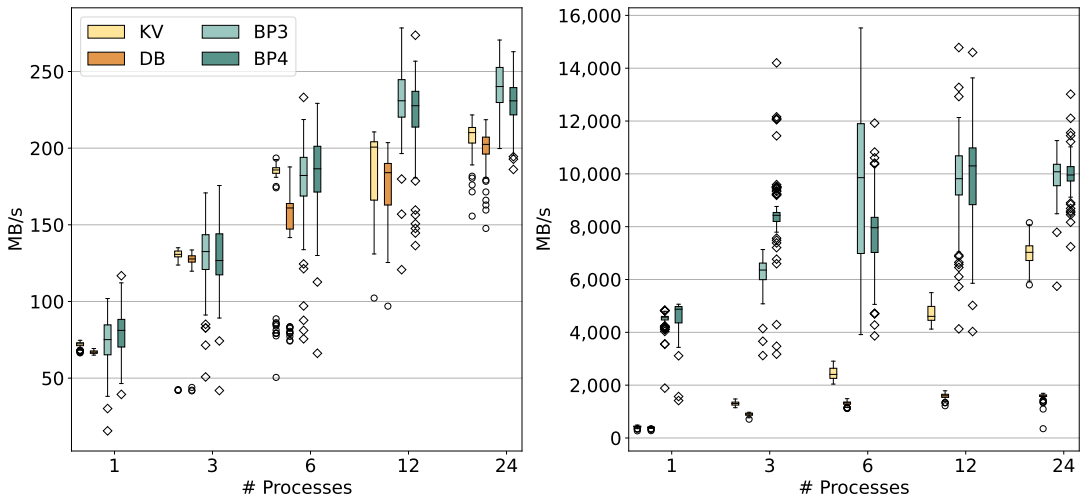
7.4.1. Write and Read Performance

In the following, the I/O performance of the three ADIOS2 and the two JULEA engines is measured. First, the general behaviour using 1, 2 and 6 nodes is shown and discussed. The results for 4 nodes will be examined in more detail in the following sections, examining two JULEA configurations and comparing the performance of two ADIOS2 versions.

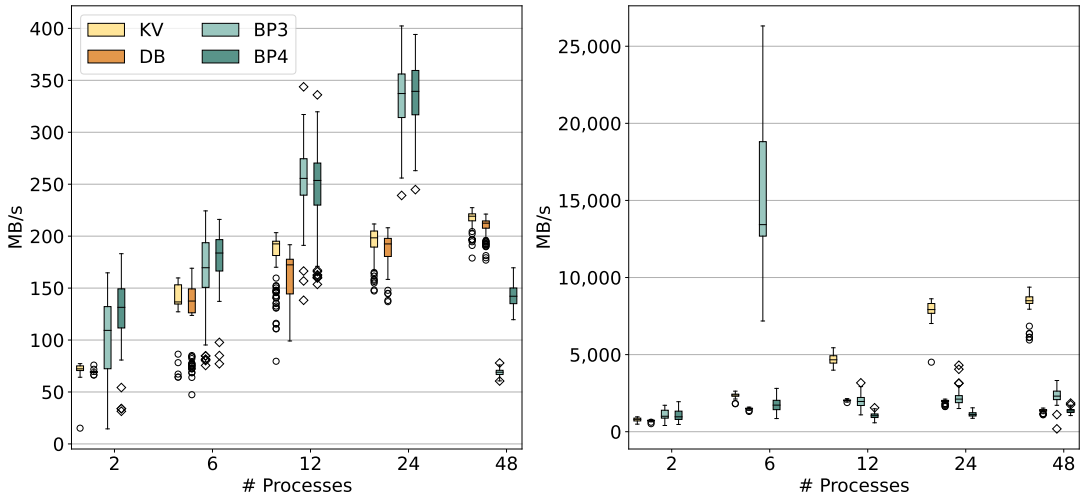
The write and read throughput for one and two nodes are shown in Figure 7.11 for BP3, BP4 and both JULEA engines. If not indicated otherwise 2 JULEA servers are used running on `ant10` and `ant11`. The write throughput of the BP engines scales up to around 24 processes per node for 1 and 2 nodes. Afterwards, the performance gain of doubling the number of processes is either negligible or even harmful in the case of 48 processes on 2 nodes. The results of using 6 nodes support this impression. For both JULEA engines, the performance gain grows less with a higher number of processes but is not decreasing. In most cases, the engine using the key-value store is only slightly better than the one using the relational database. Given the performance differences observed benchmarking the specific backends, this is surprising. While LevelDB reached up to 65,000 operations per second, MariaDB only achieved between 1,000 to 9,000 operations per second, as shown in Section 7.1.

It is also interesting to note the wide range, especially for the BP engines. For example, writing with BP4 using 6 processes can be around 70 MB/s or up to 230 MB/s, which is roughly a factor of 3. However, this variation is slight in comparison to the one observed for the reading performance, which ranges from 4 GB/s to about 15 GB/s when using BP3 on 6 processes on one node as shown in Figure 7.11a.

While BP4 shows a similar behaviour on one node, there is a drastic difference between BP3 and BP4 when using 6 processes in total on 2 nodes. BP4 ranges from 2 GB/s to 3 GB/s, whereas BP3 can reach a throughput anywhere between 7 GB/s and 26 GB/s. The read performance of BP4 on two nodes is worse than on one node in general, which cannot be observed for the other engines. Caching effects likely contribute to the wide variation. Nevertheless, it is unclear why it only occurs for BP3 and this specific number of processes.



(a) Write (left) and read (right) performance for 1 node



(b) Write (left) and read (right) performance for 2 nodes

Figure 7.11.: Write and read performance of the JULEA-KV (yellow), JULEA-DB (orange), BP3 (light teal) and BP4 (teal) engines for the HeatTransfer application using 1 (top) and 2 (bottom) nodes with 1 to 24 processes per node. The outliers of the JULEA engines are marked using a circle, whereas those of the BP engines are indicated using a small diamond shape.

7.4.2. Two JULEA Configurations

In the following, the write and read performance of the HeatTransfer application using four nodes is discussed. The results are shown in Figure 7.13. To get a better understanding and to

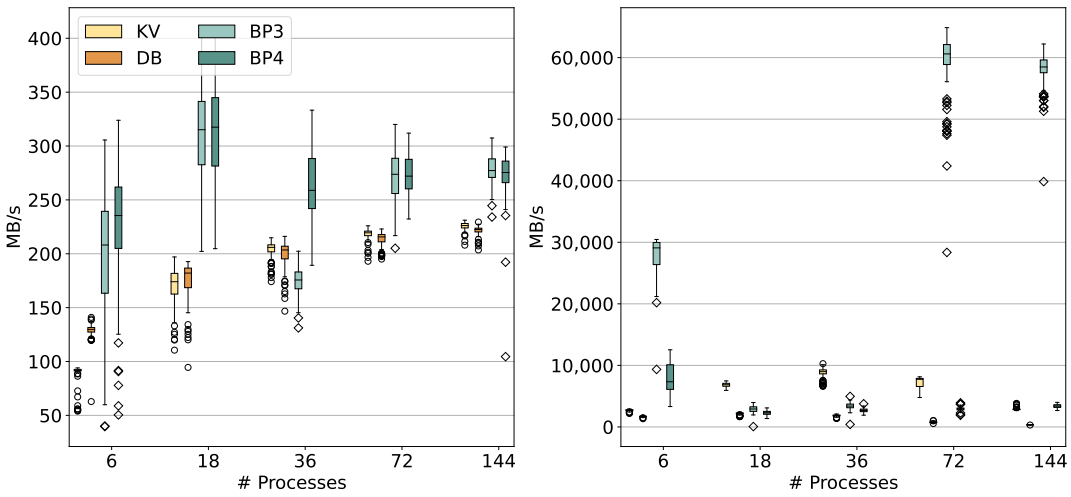


Figure 7.12.: Write and read performance of the JULEA-KV (yellow), JULEA-DB (orange), BP3 (light teal) and BP4 (teal) engines for the HeatTransfer application using 6 nodes with 1 to 24 processes per node. The outliers of the JULEA engines are marked using a circle, whereas those of the BP engines are indicated using a small diamond shape.

assess the influence of the configuration on the performance of the JULEA engine, two setups are compared:

- 1 Server:
  All backends (MariaDB and LevelDB) are run on one node (`ant10`).

- 2 Servers:
  The object backend is distributed across two node (`ant10` and `ant11`). There MariaDB server is running on `ant11`, LevelDB on `ant10`.

The difference between the results using 1 server and 2 servers is about a factor of 1,5 to 2. It is plausible that by doubling the number of hard drives, the throughput improvement will be up to a factor of 2. It appears that the used hardware is the limiting aspect, not necessarily the implementation of the engines or JULEA. The write performance of the JULEA engines does not decrease but approaches a threshold of around 240 MB/s when two servers are used. For a single server, the maximum write performance is about 120 MB/s. Overall, both engines have a similar write performance for the different configurations. However, JULEA-KV achieves a read throughput that is up to a factor of 4 higher than for JULEA-DB, even if only one JULEA server is used. As mentioned before, this is because of the

As for the previous results, the write throughput of the BP engines decreases when too many processes are used. The maximum write performance of 470 MB/s for BP3 and 450 MB/s for BP4 is achieved using 24 processes over 4 nodes. The read performance continues to vary for the engines, the number of processes and the number of nodes. In fact, the results are not shown in Figure 7.13b because the scale on the y-axis is set such that the other results

are readable. However, in Figure 7.14 they appear. While the read throughput for BP3 for 12 processes on 4 nodes ranges from 13 GB/s to 40 GB/s, the performance using 12 processes on just two nodes lies between 1 GB/s and 3 GB/s. As mentioned before, caching effects likely contribute to these large differences. Besides, the Ceph file system may be used irregularly by the other users on the cluster. This influence cannot be avoided or reduced. Still, as each variable is written for 100 steps, the results in the plots encompass 100 mean I/O rates, which should be able to mitigate a short burst of I/O activity from other users.

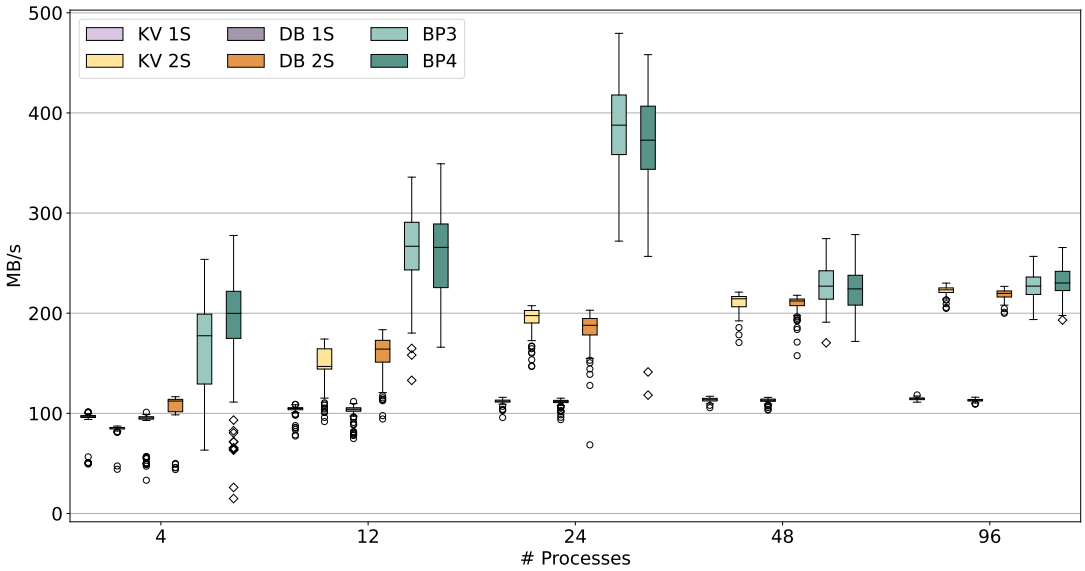
7.4.3. Comparing ADIOS2.7.1 and ADIOS2.8.3

Two ADIOS versions are compared in the following, namely 2.7.1 and 2.8.3. The results are shown in Figure 7.14. The former is the version containing the JULEA engines. The latter is used to evaluate the evolution of the ADIOS2 library as well as the new BP format, BP5. The results for the JULEA engines using 2 servers are also included to allow an easier comparison.

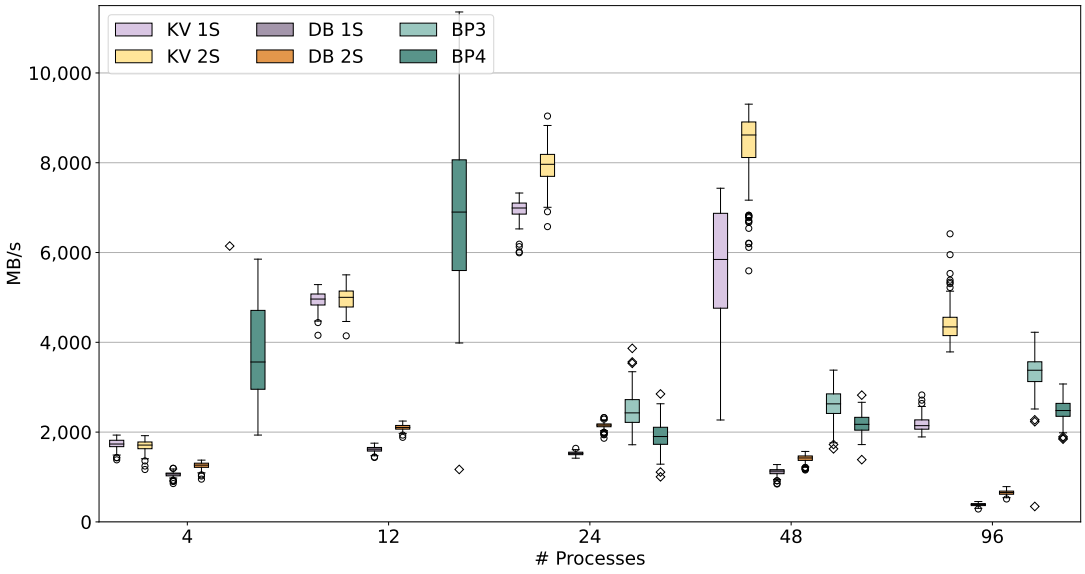
When comparing Figure 7.14 to the previous results, it becomes clear that there is a large performance gap between the different library versions. Version 2.8.3 outperforms 2.7.1 for most configurations. For 24 processes, the write performance of BP4 is about 400 MB/s for 2.7.1 but reaches close to 4 GB/s for 2.8.3. Given that the hardware and the filesystem setup are the same for both library versions, the dramatic difference has to be caused by something else.

Two potential reasons are discussed in the following. First, it is possible that the library was optimised further. When BP5 was released, the corresponding notes pointed out that the memory buffers are managed differently and that there were efforts to reduce the number of copies inside the RAM. So, some of these optimisations could potentially be employed for the other engines as well in order to lessen their memory footprint. Second, version 2.8.3 was not measured using the master branch of the official repository or the fork of the ParCIO group. Instead, as mentioned earlier, the spack package had to be used due to the compilation errors that occurred. While the build options were studied to ensure there are no major differences, it is still possible that a specific compiler flag may be set in only one of the versions. The variation in the performance of version 2.8.3 for among the engines but also for one engine across several configurations is even larger than for 2.7.1. Such a high variability points again to potential caching influences, which is plausible as it mainly concerns the read performance.

Discussion The results highlight why it is important to keep as many aspects the same across separate measurements as possible. If the JULEA engines were only compared to the results of 2.8.3, their performance would appear extremely poor. The same problems arise when only evaluating BP5 with version 2.8.3. Then, the new format specification and the changed metadata management would be the apparent difference between the engines, giving the impression that they cause the differences. The results for 2.7.1 reveal that BP3 and BP4 perform significantly better in version 2.8.3 as well. Overall, the performance of the JULEA engines is comparable to those of the BP engines in version 2.7.1. Using a different library version has more impact on the results than the choice of engine.

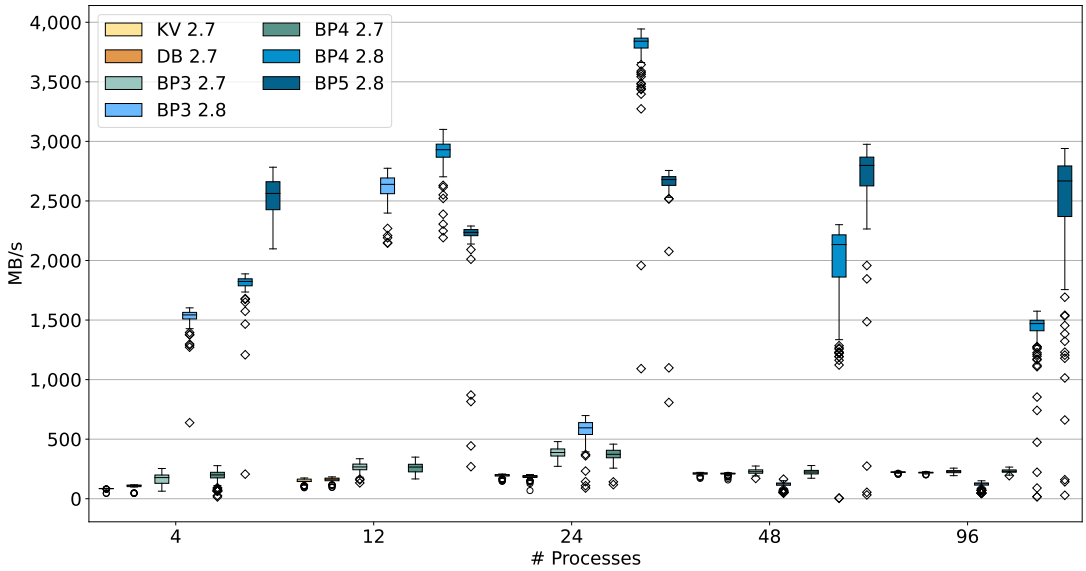


(a) Write performance for 4 nodes

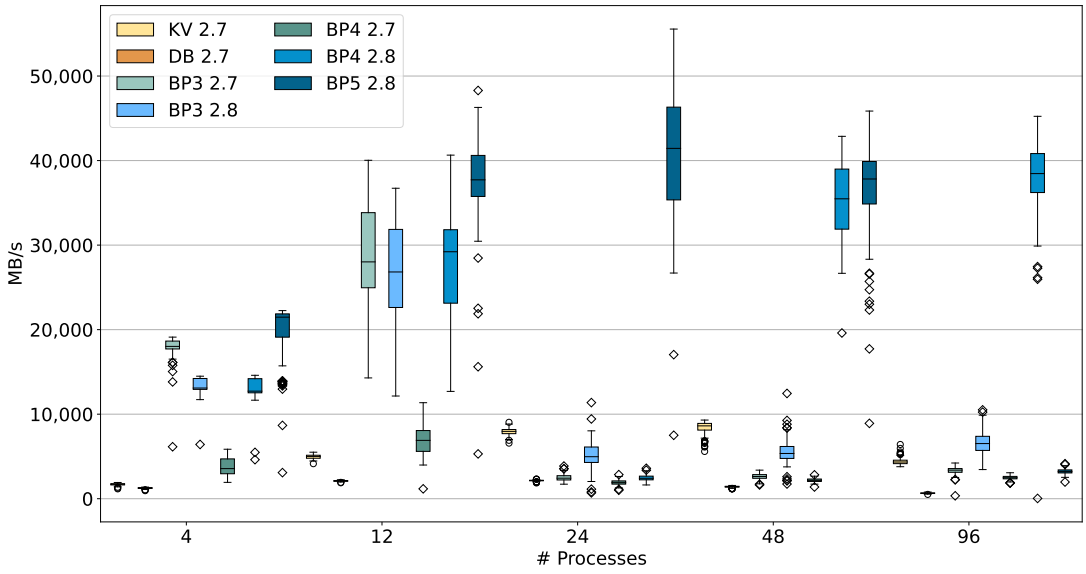


(b) Read performance for 4 nodes

Figure 7.13.: Write and read performance for the HeatTransfer application using 4 nodes with 1 to 24 processes per node for BP3, BP4, JULEA-KV and JULEA-DB. The outliers of the JULEA engines are marked using a circle, whereas those of the BP engines are indicated using a small diamond shape. For both JULEA engines, a second configuration (light lilac and lilac) uses only one node to run the server on. Note that the read performance for BP3 for 12 and 24 processes lies outside the shown range. It is included in Figure 7.14b.



(a) Write Performance for 4 Nodes



(b) Read Performance for 4 Nodes

Figure 7.14.: Write and read performance for the HeatTransfer application using 4 nodes with 1 to 24 processes per node for BP3, BP4, JULEA-KV and JULEA-DB. In addition, two ADIOS2 versions (2.7.1 and 2.8.3) are compared which allows examining the performance of BP5 as well as it was only introduced in version 2.8. The results of ADIOS2.8.3 are coloured from light to dark blue. The outliers of the JULEA engines are marked using a circle, whereas those of the BP engines are indicated using a small diamond shape.

7.5. DAI Performance for Post-Processing Queries

The previous evaluation of the query performance only included the computation of the mean value, as an early prototype was measured. Also, there was no DAI yet. Therefore, the performance of typical post-processing queries will be evaluated in the following.

Setup

For the most part, the setup used to evaluate the query performance is the same as for the write and read evaluation in Section 7.4. The query application reads part of the data that was written during previous measurements. Therefore, parameters like the matrix size, as well as the number of blocks are already determined.

Hardware

The query application is run on one of the smaller nodes, that is `ant13` to `ant20`. The JULEA servers run on the larger nodes `ant10` and `ant11`.

- Larger node - `ant10`:
  - CPU: 2x AMD Epyc 7543 2.8 GHz with 32 cores each
  - SSD: Intel SSDSC2KB960G8 (D3-S4510 Series), 960 GB, SATA 6GB/s
    Max. performance as per specification: 510 MB/s (write), 560 MB/s (read) [19]
  - RAM: 1024 GB
  - Network: 100GbE and 100ib Mellanox
- Smaller node - `ant13`:
  - CPU: AMD Epyc 7443, 2.85 GHz with 24 cores
  - HDD: Western Digital 500 GB WD5000AAKS
    Host to/from drive: 126 MB/s sustained [20]
  - RAM: 128 GB
  - Network: 100GbE and 100ib Mellanox

Software

The software used for the post-processing evaluation remains largely the same as for the HeatTransfer measurements. For more details see Section 7.4:

- ADIOS2 2.7.1:
  The used commit is [21].

[19] `https://ark.intel.com/content/www/us/en/ark/products/134912/intel-ssd-d3s4510-series-960gb-2-5in-sata-6gbs-3d2-tlc.html` *accessed: 09.10.2022*
[20] `https://products.wdc.com/library/SpecSheet/ENG/2879-701277.pdf` *accessed: 09.10.2022*
[21] `https://github.com/julea-io/adios2/commit/2c6b0aa13f1b36329746d607a816441b7521e4e5`

- ADIOS2 2.8.3:
  As the master does not currently compile for version 2.8.x on the ants cluster, the ADIOS2 spack package is used for the evaluation of 2.8.3.

- JULEA:
  The JULEA version used in the evaluation contains the addition of the DAI. The code can be found at `https://github.com/Bella42/julea/tree/dai`

Applications

Two applications have been implemented for the DAI evaluation. Both are run using only a single process as post-processing applications, especially visualisation or plotting scripts, are often not parallelised. So, in the following, the benefits are examined that users can get without requiring parallelism in their post-processing.

- ADIOS2 Query Application: ADIOS2-Query
  This first application uses the ADIOS2 interface to access the data. So, it shows how the native library would be used to answer the posed queries. In contrast to the DAI application, more computations have to be performed in this application because the respective computation is part of the DAI implementation itself.

- DAI Query Application: DAI-Query
  The second application uses the new DAI interface to query the database. The overhead of specifying the DAI pre-computations and tags is not evaluated separately, as it is already captured in the JULEA database benchmarks. This is because the overhead equals the time it takes to create a new table and enter a specific number of entries. Furthermore, the time for the pre-computations is not accounted for separately, as it was already measured in the write performance of the JULEA engines.

7.5.1. Query Time

Four queries have been derived from the use cases gathered during the specification of the requirements for the DAI in Section 6.1.3. They have been phrased such that they represent a real-world question. The corresponding steps for the implementation are shown as well.

- Query 1:
  Find all locations where the temperature is between -42°C and 42°C. →Compute which blocks have a minimum temperature above -42°C and a maximum temperature below 42°C and return the blockIDs.

- Query 2:
  What is the highest mean value of a location? →Compute the mean for all blocks and find and return the overall maximum.

- Query 3:
  Which location had the largest increase in maximum temperature between steps 1 and

100? →Retrieve the maximum temperature of all blocks for steps 1 and 100 and compute the local difference. Find the maximum and return the blockID.

- Query 4:
  What was the highest precipitation sum for a location when the maximum temperature was above 40°C? →Retrieve all blocks with a maximum temperature above 40, read the precipitation data for the same time and location (step and block), compute the sum and find and return the maximum sum.

Query 1

Find all locations where the temperature is between -42°C and 42°C.
The first query does not use new metadata but works on file metadata that is in the original BP formats. It retrieves the blocks where the conditions are met. Another way of phrasing this task is to get all city or region names where the temperature is inside this specific range. In the case of the JULEA engine, this information is stored in the database instead of the BP file.

This query was selected because it highlights the benefits of data characteristics like the minimum and maximum values that are already present in ADIOS2. Also, this reduces the advantage of the DAI as the information to query does not need to be computed for the BP engines. When evaluating novel designs, it is important to not only focus on scenarios that have a high probability of performing exceptionally well. The expectation for this query is that the performance of the JULEA engines using the ADIOS2 interface will, at best, be as good as that of the BP engines.

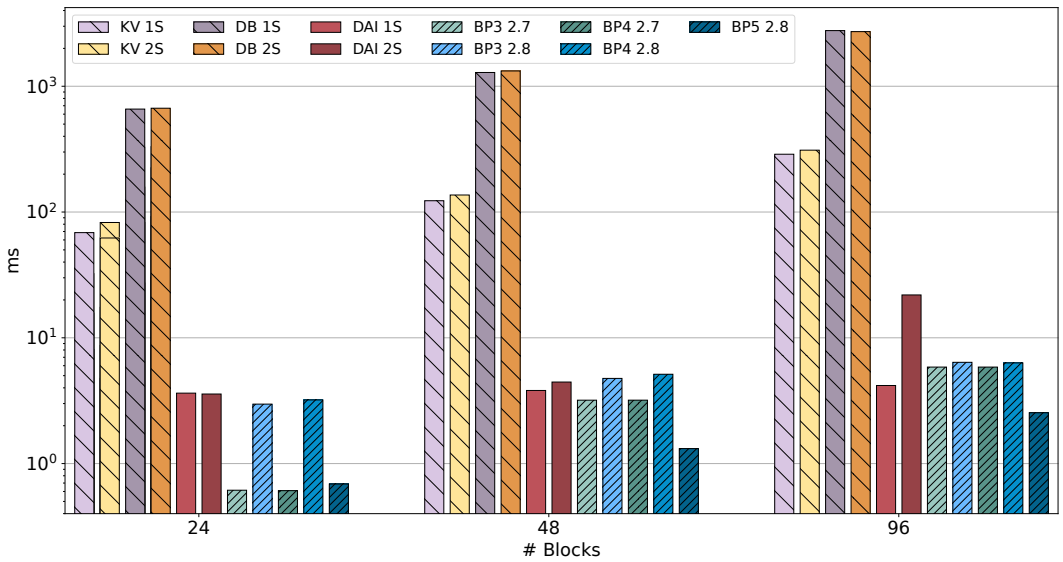
The results are shown in Figure 7.15a. As the runtime of this query for the JULEA engines is mainly determined by the performance of the specific backend technology, it is not surprising that the results using MariaDB have the longest runtime with 2,770 ms for 96 blocks. The approach to mitigate the performance problems is to run the database server on faster hardware, ideally in NVRAM. Unfortunately, such hardware is not part of the used cluster.

Note that the performance using 2 JULEA servers is worse for this query than when a single server is used. A probable cause is an overhead of distributing the data across nodes.

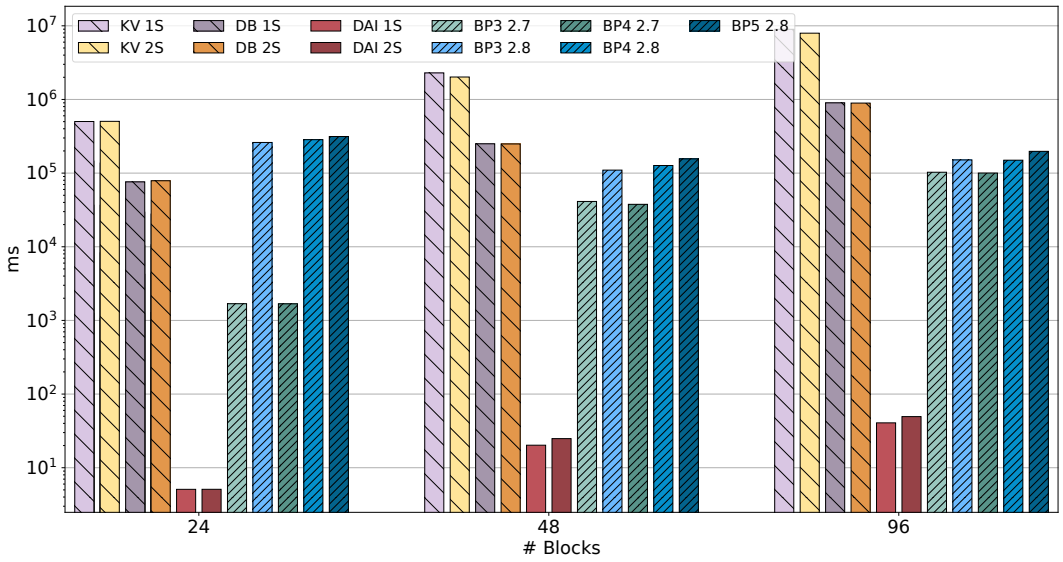
When the DAI (DAI-Query) is used instead of the ADIOS2 interface (ADIOS2-Query), the performance is improved significantly from 2,770 ms (1S) and 2,720 ms (2S) to 4 ms (1S) and 21 ms (2S). This shows the advantage when the interface has knowledge about the backend technology. Overall, ADIOS2 version 2.8.3. does not perform better than 2.7.1.

Query 2

What is the highest mean value of a location?
This query was selected to demonstrate the advantage of custom metadata when it is pre-computed. In ADIOS2-Query, the variable data needs to be read in order to compute the mean value. For DAI-Query, this is not necessary, as the mean value was already computed during the writing of the variable. Therefore, the pre-computation time is not included in query results but in the I/O throughput shown in the previous section.



(a) Query time for Query1



(b) Query time for Query2

Figure 7.15.: The runtime for Query1 (top) and Query2 (bottom). The results were written with 2, 4 and 6 nodes with 24 processes per node which equal writing 24, 48 and 96 blocks. The results using the ADIOS2 interface are hatched (coarse: JULEA engines, fine: BP engines); those using the DAI interface are not. The JULEA engines are again tested using 1 and 2 servers (1S and 2S). For the BP engines, versions 2.7.1 and 2.8.3 are compared. BP5 was first introduced in version 2.8. Note the logarithmic scale on the y-axis.

To reduce this time, the computation of additional custom metadata, which is added by default, is gathered and combined as much as possible. For example, the sum is reused for the computation of the mean and the standard deviation. However, this requires that the different metadata options are always used in combination. Furthermore, adding more computations will get more complicated this way.

As expected, the DAI-Query results are better than those of ADIOS2-Query for JULEA-DB, as depicted in Figure 7.15b. The former are 41 ms and 49 ms for 96 blocks in contrast to 899,400 ms and 893,200, which is a factor of 21,940 respectively 18,230. The performance of the DB engine is restricted by the used technology again. Therefore, the DAI interface results are impressive as they still rely on MariaDB. JULEA-KV has a runtime of 8,971,400 ms and 7,941,570 ms for 96 blocks. The BP engines have a runtime from 102,780 ms for BP3 (2.7.1) to 197,530 ms for BP5.

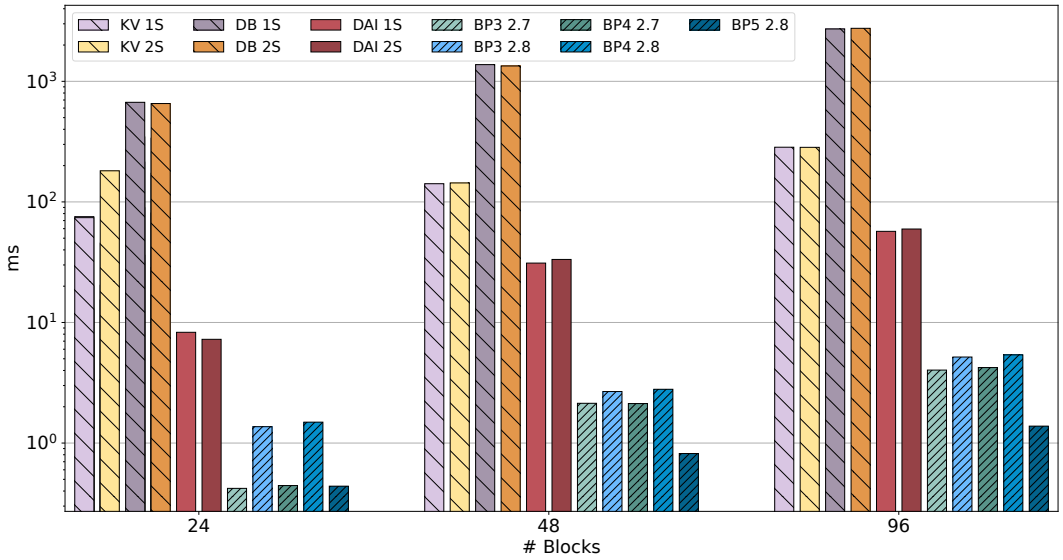
The larger the data and the more blocks it has, the better the BP engines perform compared to the JULEA engines when ADIOS2-Query is used. As all data needs to be accessed to compute the mean value, each corresponding entry in the key-value store and the database has to be accessed. Note that the number of blocks equals the number of entries per step.

Query 3

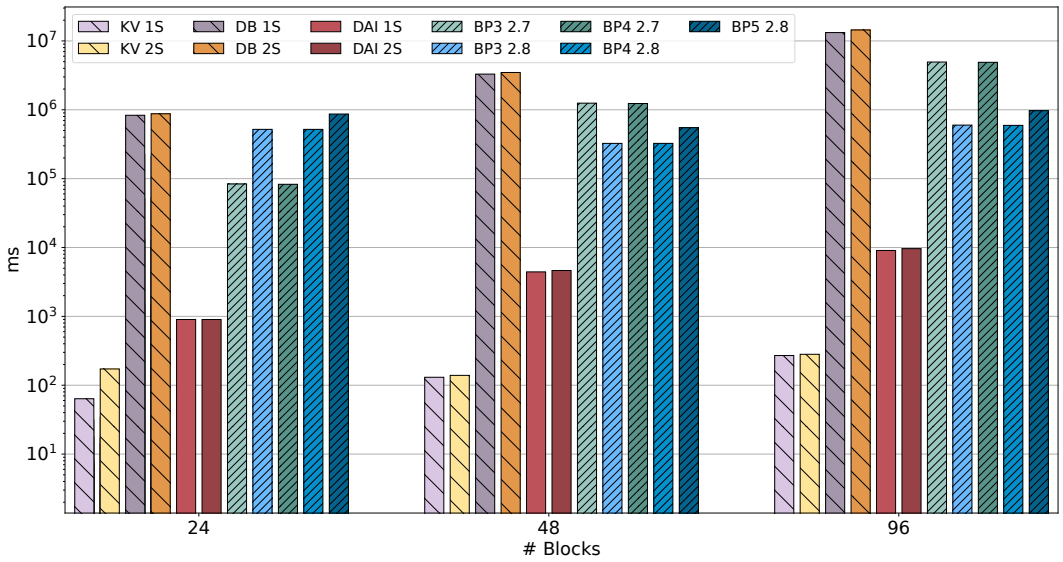
Which location had the largest increase in maximum temperature between step 1 and 100?

The minimum and maximum used in query 1 as well as the mean value used in query 2 are computed across all blocks in one step. Query 3 is included to examine the performance when the metadata is queried across multiple steps. Also, it is another example that was selected specifically to not rely completely on convenient features like the pre-computation. Instead, the maximum temperature for all blocks in step 1 and step 100 needs to be read and compared against each other.

The results are shown in Figure 7.16a. The runtime of the JULEA engines when 2 servers are used is nearly the same as if one server is used. The JULEA-DB engine has the longest runtime which is in part because of the lower performance of MariaDB as mentioned before. Therefore, it is interesting, that using the DAI which directly interfaces with the JULEA clients achieves a much smaller runtime of around 31 ms and 33 ms for 96 blocks in contrast to 2,720 ms and 2,750 ms. So, it is not only faster than using MariaDB through the ADIOS2 interface but also faster than using LevelDB even though the DAI works on MariaDB as well. The BP engines have a small runtime for example of 5 ms for BP3 2.7.1 and 1 ms for BP5. A possible reason is the number of small reads that have to be performed using the JULEA engines. Currently, there is no way to read all file metadata for a specific variable at once. In contrast, the BP engines only need to read the metadata file of their formats and all blocks maxima are available to the application. One option to mitigate the drawbacks would be to offer an iterator to the application to allow reading multiple entries more efficiently. However, this would expose more database-related functionality to the users which might not be the best way to go considering they dislike SQL.



(a) Query time for Query3



(b) Query time for Query4

Figure 7.16.: The runtime for Query3 (top) and Query4 (bottom). The results were written with 2, 4 and 6 nodes with 24 processes per node which equal writing 24, 48 and 96 blocks. The results using the ADIOS2 interface are hatched (coarse: JULEA engines, fine: BP engines); those using the DAI interface are not. The JULEA engines are again tested using 1 and 2 servers (1S and 2S). For the BP engines, versions 2.7.1 and 2.8.3 are compared. BP5 was first introduced in version 2.8. Note the logarithmic scale on the y-axis.

Query 4

What was the highest precipitation sum for a location when the maximum temperature was above 40°C?
This query was selected for the final evaluation because it queries more than one variable and is more complex than the previous ones. In Figure 7.16b the results are shown. Here, the JULEA-KV engine performs best with a runtime of around 270 ms, while JULEA-DB has a runtime of 9060 ms (1S) and 9630 ms (2S) for 96 blocks. This is a factor of about 35. There is only a slight difference for most configurations when the runtime of having one server is compared to using 2 servers. However, for 24 blocks, the DAI has a runtime of 0.5 ms and 900 ms. This is also the configuration where it makes the most difference for query 2 and query 3 as well. Therefore, it is unlikely that this gap is a coincidence. So, it seems that having a single JULEA server for smaller file sizes is beneficial for the DAI runtime.

It is interesting to note the runtimes of the two ADIOS2 versions. In contrast to the previous examples, there is no clear winner here. Instead, version 2.7.1. performs better for smaller file sizes (24 blocks per step), whereas version 2.8.3 is faster for 48 and 96 blocks per step.

Discussion

In order to draw a conclusion from the large number of results presented in this section, they are summarised in the following. In Table 7.1, the results of all measured configurations for all engines are assessed in larger groups and graded according to their runtime.

	BP Engines	JULEA-KV	JULEA-DB	DAI
Query 1	1	2	3	1
Query 2	2	4	3	1
Query 3	1	3	4	2
Query 4	3	1	4	2

Table 7.1.: Assessment of the query runtimes from best (1) to worst (4) with the corresponding colour-coding. The different configurations that are presented in the previous plots are summarised into five larger groups: BP engines, JULEA-KV, JULEA-DB, and DAI. The colour coding

Some interesting aspects become more evident in the condensed query results in Table 7.1:

- Query selection:
  The query selection was varied and focused on different capabilities.
- No clear winner:
  No candidate is best for each scenario. This is not surprising since the analysis of data access patterns indicated that they could be very different.
- Format dissection:
  The query evaluation showed that the format dissection using a relational database is not itself beneficial when a slow backend is used. JULEA-DB has the worst ratings. The format dissection using the key-value store does better than the one using the relational

database. This can be mainly ascribed to the underlying performance of LevelDB and MariaDB.

- Introducing the DAI:
  The DAI is a great addition and helps improve the performance significantly. The data and metadata that can be accessed through the DAI are written and managed by the JULEA-DB engine and therefore stored in MariaDB as well.

When bringing the format dissection, the file metadata management in a relational database and the DAI together, they can compete with the BP engines when writing data. They even outperform the native engines significantly when the custom metadata is relevant to the posed question.

Not all use cases that were derived from the requirements in Chapter 6 have been evaluated. This is because they have overlapping functionality. For example, evaluating how fast tagged data can be retrieved using the DAI is essentially reading from the used database, which is covered by the JULEA benchmarks at the beginning of this chapter. Another feature that was not evaluated is the option to query across files. The reason to omit it was mainly due to the hardware restrictions of the cluster. As the user survey showed, users typically do not use many small files because of the associated overhead for the file system. However, the limited storage space of the cluster did not allow for writing several decent-sized files. Still, it makes sense that having the file metadata inside one table allows faster querying across files. For example, if the variable blocks of different files have to be read, numerous files have to be opened.

Chapter Summary

This chapter evaluated the proposal developed in the previous chapters to dissect self-describing data format and store the file metadata in either a key-value store or a relational database while an object store manages the data. First, the performance of the different JULEA backends is measured using the built-in benchmark suite of JULEA. LevelDB is JULEA's best-performing key-value store backend overall. For applications that focus only on the read performance, LMDB is a good candidate. The performance of the relational database backends is improvable. While SQLite Memory offers sufficient performance, the volatility of the data makes it unfit as the main file metadata backend. The database backends perform best when operation batching and a single index is used. Batching also helps the object store to reach a reasonable throughput. Unfortunately, BlueStore could not be used independently of Ceph easily. The requirements to use BlueStore are reduced to installing the entire Ceph dependencies, whereas before, a Ceph instance was necessary. Aside from overhead setting BlueStore up, the performance was not sufficient. Therefore, the rest of the evaluation continued to use JULEA's POSIX object store.
The HDF5 benchmarks showed that the JULEA VOL plugins could achieve competitive performance compared to the native HDF5. The evaluation of the simple DAI prototype demonstrated that the format dissection could be beneficial if the accesses do not need to use the I/O interface and thus do not have to pass through the entire library. If the respective JULEA client can be used instead, an improvement of a factor of 3 was possible. The performance can be improved further, up to a factor of 17, if an iterator is used to access all entries at once.

The evaluation of the early database engine prototype with the HeatTransfer application revealed that the format dissection also works with ADIOS2. Writing file metadata into the database can incur performance overhead due to the ACID guarantees and the additional overhead from serialising file metadata in the backend implementation. This was primarily an issue of the chosen HDD-based setup and can be alleviated by putting the database onto faster storage technologies such as SSDs or NVRAM. However, the format dissection began to shine when the I/O path through the library was avoided in post-processing by using the DAI. The query evaluation showed that a performance improvement of a factor of 60,000 could be reached when using the JULEA-DB engine with the DAI layer.
The final evaluation results illustrated the importance of comparing the correct configurations. It showed that using a different library version can have a larger impact on the results than the choice of engine. Overall, the performance of the JULEA engines was comparable to the BP engines in version 2.7.1.

8. Conclusion and Future Work

This chapter recalls the initial situation and the requirements derived after reviewing state-of-the-art solutions and related work. After presenting the individual achievements, topics to analyse further are proposed.

- *Section 8.1: provides a summary the thesis*
- *Section 8.2: gathers point of attack for future work and gives an outlook.*

8.1. Summary and Conclusion

In times of continuously growing data sizes, performing insightful analysis is increasingly difficult. Therefore, it is vital to sift through the data volumes most efficiently. However, the hardware hierarchy of high-performance computing (HPC) systems complicates the data analysis. Another factor increasing the complexity is the software stack on top. The stack splits into several mostly isolated layers. While this allows exchanging them, the isolation leads to performance and management issues, e.g., how and where to optimise the data access.

To make filtering raw data easier, I/O libraries such as HDF5 (Hierarchical Data Format) and ADIOS2 (Adaptable IO System) and the corresponding self-describing data formats are used. The goal is to provide self-explanatory data that can be exchanged between researchers easily through annotating the data, for example, with information about the experiment. However, queries on this metadata are difficult as it is currently stored inside the corresponding data formats on data servers.

The approach developed in this thesis is to dissect the self-describing format inside the I/O library and forward the file metadata and data to the JULEA storage framework. File metadata will be stored in the key-value store backend or the database backend, while the data will be stored in the object storage backend. This opened the door for further extensions, such as pre-computing statistics like the variable mean. To query the custom metadata efficiently, a novel data analysis interface (DAI) was designed that works directly on the JULEA clients.

Chapter 2 introduced the background information. Popular I/O interfaces and the corresponding data formats were presented. A particular focus was set on the file formats of ADIOS2, BP3, BP4 and BP5 as they form the basis for the format dissection. An overview of database systems was given, including their underlying data models and relational databases. A taxonomy of NoSQL storage technologies was presented, which structured the details about several candidates, for example, key-value stores, wide-column stores and document stores. The chapter was completed by describing file systems and the JULEA architecture.

Chapter 3 examined the current state of related work and answered research question 1.1 (What optimisations exist for managing self-describing data formats and storage systems?). It offered an overview of possible alternatives and discussed their advantages and disadvantages in the context of the goals outlined in the introduction. The project most closely related to the thesis topic is EMPRESS 2, which was developed for a similar goal. However, a different route was taken how to achieve it. The biggest difference is that the approach developed in this thesis aims to be as transparent to the application layer as possible. In contrast, EMPRESS 2 requires that users specify the metadata they need explicitly, like defining custom time steps. Aside from EMPRESS 2 projects were discussed that involve extensive changes to the current ecosystem in order to advance the storage landscape.

Chapter 4 addressed research question 1.2 (Where is untapped potential, and how can it be used?). By dissecting the file format, the file metadata can be managed by suitable backends in JULEA; key-value stores in this chapter. The dissection improves access to the information already provided by the format. Furthermore, it enables the addition of custom metadata. The ADIOS2 variables were examined in detail as they lay the foundation for the following chapters. Afterwards, the options are discussed on how the variable can be serialised. Both an ADIOS2 engine and an HDF5 VOL plugin were discussed to show that the format dissection works for different formats.

Chapter 5 presented how the format dissection can be advanced. After using key-value stores for the first engine, the next logical step that enabled more complex queries was to use a relational database. As JULEA did not support them, a novel JULEA client and backend were developed. They were used for the second HDF5 VOL plugin that employs an entity-relationship data model.

With the plan to compute custom metadata, other database technologies were studied to find options that would offer the necessary flexibility while still maintaining performance. At first glance, data warehousing seemed like a promising candidate. In order to make the decision, more information about the data access patterns was required. Therefore, common data layouts and the results of the first part of the user survey are studied. This revealed that the access patterns vary considerably, which makes designing a sensible data warehouse data model difficult, if not impossible, as they are usually optimised for specific tasks. As a result, the ADIOS2 database engine also uses a relational database like the second VOL plugin, but the schema design was influenced by data warehousing. For example, the tables are not normalised. As the restrictions by the file format no longer apply, custom metadata can be stored along with the original file metadata. Therefore, metadata categories are derived that form the basis of the DAI pre-computation.

Chapter 6 addressed research questions 3.1 (What functionality is required for the DAI interface?) and 3.2 (How can this approach be generalised?). To determine the necessary functionality, the target audience has to be determined. Different post-processing tasks from the domain of climate research were studied, and a subset of the climate data operators (CDOs) was selected. Besides the computations of statistics like the sum, mean, or variance, domain-specific features like the climate indices were chosen. The evaluation of the second part of the user survey revealed that users do not like to use POSIX and SQL. They deem I/O libraries acceptable but overall favour python for data access. Combining the previous results allowed for

deriving requirements and use cases. These use cases lead the implementation effort as they help to keep the end user in mind. The functionality of the DAI includes pre-computation, tagging and reading. The computation of the climate indices was added to the engine and the DAI to show how domain-specific features can be incorporated. The DAI prototype is tailored to the ADIOS2 interface and data model. To have other formats and scientific research areas benefit from it as well, it has to be generalised as it is currently format dependent. One challenge of offering a more general interface is that there are numerous ways to model data, even if only one library is used. This could be addressed by restricting the data models or working on a higher abstraction level.

Chapter 7 evaluates the performance of the HDF5 VOL plugins using JULEA benchmarks and the ADIOS2 engines using the HeatTransfer and a query application. The query evaluation showed that the format dissection alone is not beneficial when a slow backend is used. However, when bringing the format dissection, the file metadata management in a relational database and the DAI together, the performance can compete with the BP engines when writing data even if the backend itself is not changed. The combination outperforms the native engines significantly when the custom metadata is queried. In conclusion, the proposed approach provides acceptable performance for the I/O typically performed in a large simulation while offering significant improvements for the post-processing. All this can be achieved without changing the application code at all because of the default pre-computation of statistics like the mean, thereby preserving the efforts that went into the development. The post-processing application has to use the DAI in order to benefit from the custom metadata.

8.1.1. Conclusion

The research questions that were selected in the introduction guided the entire design and implementation process. In the following, the insights gathered throughout the thesis are summarised briefly to provide answers:

- **RQ 1**: How to solve problems with state of the art?
  - Make the information better accessible that is already provided in the format
  - Dissect the format into file metadata and data
  - Map them to suitable storage backends
- **RQ 2**: What optimisations become possible when the format is split?
  - Extend the data characteristics already offered in ADIOS2
  - Pre-compute results that are often used in post-processing
  - Tag areas of interest to reduce data access
  - Improved placement strategies for HSM
- **RQ 3**: How to design an analysis interface for file metadata?
  - Work out requirements and use cases
  - Study access patterns and data layouts

  - Automate or hide as much as possible from the application layer
  - Build the interface directly on top of the storage system

8.2. Future Work

In the following, some aspects that would benefit from further work are explained:

- Plugin engines:
  This aspect follows from the discussion of how the current solution can be generalised. Using the plugin engine concept, the DAI configuration, for example, no longer needs to be communicated over database tables.

- Distributed Databases:
  The current solution to employ only a single database instance was sufficient for the size of the used clusters. However, to make the approach more scalable and useful for larger systems, distributed databases should be used, for example, to enable parallel access.

- Advance HSM through format dissection and tagging:
  Coupling the storage system with the I/O libraries and formats also enables exploiting structural information and thus making informed decisions about data placement. A noticeable improvement made possible by the separation is to store the database holding the file metadata on a different hardware tier. As the file metadata is small compared to the data sizes, the metadata backend can fit on faster and more expensive hardware. The option to specify tags can improve the placement decisions. Marking interesting data that is especially relevant for analysis means that this data can be stored on a faster tier.

- Automate tagging:
  Having to specify tags by hand is inconvenient for the user. Here is a lot of room for improvement. For example, the information which tags create could be automated, letting the system learn what the user is interested in.

- Export functionality:
  To support more use cases, it would be useful to have the option of exporting the variable and block metadata from the JULEA backends. A simple option would be to write the database content to a tsv (tab-separated values) file. To maintain at least a simple form of the self-describing feature, the column names and types have to be exported as well. There are advantages and disadvantages to exporting ADIOS2 internal data as well. However, it seems like a good idea to offer the option of at least exporting a subset of the variable or block metadata, for example, only the mean values and all information required to understand the data itself.

Outlook

In the future, it will become increasingly important to continue establishing a workflow perspective that considers end-to-end processing instead of the still prominent mindset that all steps in the data life cycle are separate and should continue to do so. With the complexity of today's system, the different systems need to work together to enable sufficient performance and facilitate scientific insights.

The discussion about how the proposed approach can be generalised leads to the question of what will come next. While examining the current database technologies to find a suitable candidate for the ADIOS2 database engine, one approach stood out: smart datalakes [Bian et al., 2021]. Based on the observation that a large portion of the data that is passing through the extract-transform-load (ETL) process is never accessed, data lakes skip the ETL process and work on raw data in various forms. Works like NoDB and H20 show how queries can be executed on raw data files and determine on-the-fly which data access patterns are most suitable [Alagiannis et al., 2012, Alagiannis et al., 2014, Sanca and Ailamaki, 2022]. When the HPC and the database communities overcome their differences, there is much to learn. While the initial problem phrasing as well as the hardware and software environment might be different, many challenges turn out to be very similar.

Bibliography

[Abramson et al., 2020] Abramson, D., Jin, C., Luong, J., and Carroll, J. (2020). A BeeGFS-Based Caching File System for Data-Intensive Parallel Computing. In Panda, D. K., editor, *Supercomputing Frontiers - 6th Asian Conference, SCFA 2020, Singapore, February 24-27, 2020, Proceedings*, volume 12082 of *Lecture Notes in Computer Science*, pages 3–22. Springer. (Cited on page 32)

[Aghayev et al., 2019a] Aghayev, A., Weil, S., Ganger, G., and Amvrosiadis, G. (2019a). Reconciling lsm-trees with modern hard drives using bluefs. Technical report, Technical Report CMU-PDL–, CMU Parallel Data Laboratory, April. (Cited on page 33)

[Aghayev et al., 2019b] Aghayev, A., Weil, S. A., Kuchnik, M., Nelson, M., Ganger, G. R., and Amvrosiadis, G. (2019b). File systems unfit as distributed storage backends: lessons from 10 years of Ceph evolution. In Brecht, T. and Williamson, C., editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 353–369. ACM. (Cited on pages 30, 32, 33, and 65)

[Aghayev et al., 2020] Aghayev, A., Weil, S. A., Kuchnik, M., Nelson, M., Ganger, G. R., and Amvrosiadis, G. (2020). The Case for Custom Storage Backends in Distributed Storage Systems. *ACM Trans. Storage*, 16(2):9:1–9:31. (Cited on pages 32, 33, and 65)

[Alagiannis et al., 2012] Alagiannis, I., Borovica, R., Branco, M., Idreos, S., and Ailamaki, A. (2012). Nodb: efficient query execution on raw data files. In Candan, K. S., Chen, Y., Snodgrass, R. T., Gravano, L., and Fuxman, A., editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 241–252. ACM. (Cited on page 183)

[Alagiannis et al., 2014] Alagiannis, I., Idreos, S., and Ailamaki, A. (2014). H2O: a hands-free adaptive store. In Dyreson, C. E., Li, F., and Özsu, M. T., editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 1103–1114. ACM. (Cited on page 183)

[Bez et al., 2021] Bez, J. L., Tang, H., Xie, B., Williams-Young, D. B., Latham, R., Ross, R. B., Oral, S., and Byna, S. (2021). I/O bottleneck detection and tuning: Connecting the dots using interactive log analysis. In *6th IEEE/ACM International Parallel Data Systems Workshop, PDSW@SC 2021, St. Louis, MO, USA, November 15, 2021*, pages 15–22. IEEE. (Cited on page 9)

[Bian et al., 2021] Bian, H., Chandra, B., Mytilinis, I., and Ailamaki, A. (2021). Storage management in smart data lake. In Costa, C. and Pitoura, E., editors, *Proceedings of the Workshops of the EDBT/ICDT 2021 Joint Conference, Nicosia, Cyprus, March 23, 2021*, volume 2841 of *CEUR Workshop Proceedings*. CEUR-WS.org. (Cited on pages 75 and 183)

[Bloch, 2006] Bloch, J. J. (2006). How to design a good API and why it matters. In Tarr, P. L. and Cook, W. R., editors, *Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, pages 506–507. ACM. (Cited on pages 102 and 115)

[Blockhaus et al., 2020] Blockhaus, P., Broneske, D., Schäler, M., Köppen, V., and Saake, G. (2020). Combining Two Worlds: MonetDB with Multi-Dimensional Index Structure Support to Efficiently Query Scientific Data. In Pourabbas, E., Sacharidis, D., Stockinger, K., and Vergoulis, T., editors, *SSDBM 2020: 32nd International Conference on Scientific and Statistical Database Management, Vienna, Austria, July 7-9, 2020*, pages 29:1–29:4. ACM. (Cited on page 27)

[Boncz et al., 2008] Boncz, P. A., Kersten, M. L., and Manegold, S. (2008). Breaking the memory wall in MonetDB. *Commun. ACM*, 51(12):77–85. (Cited on page 27)

[Bovet and Cesati, 2005] Bovet, D. P. and Cesati, M. (2005). *Understanding the Linux Kernel - from I/O ports to process management: covers version 2.6 (3. ed.).* O'Reilly. (Cited on page 3)

[Braam, 2019] Braam, P. (2019). The Lustre Storage Architecture. *CoRR*, abs/1903.01955. (Cited on page 31)

[Brandt et al., 2009] Brandt, S., Maltzahn, C., Polyzotis, N., and Tan, W.-C. (2009). Fusing Data Management Services with File Systems. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, PDSW '09, pages 42–46, New York, NY, USA. ACM. (Cited on page 44)

[Breitenfeld et al., 2017] Breitenfeld, M. S., Fortner, N., Henderson, J., Soumagne, J., Chaarawi, M., Lombardi, J., and Koziol, Q. (2017). DAOS for Extreme-scale Systems in Scientific Applications. *CoRR*, abs/1712.00423. (Cited on page 39)

[Breitenfeld et al., 2020] Breitenfeld, M. S., Pourmal, E., Byna, S., and Koziol, Q. (2020). Achieving High Performance I/O with HDF5. `https://www.hdfgroup.org/wp-content/uploads/2020/02/20200206_ECPTutorial-final.pdf`. last accessed: 28.08.2022. (Cited on pages 19, 128, and 129)

[Brinkmann et al., 2020] Brinkmann, A., Mohror, K., Yu, W., Carns, P. H., Cortes, T., Klasky, S., Miranda, A., Pfreundt, F., Ross, R. B., and Vef, M. (2020). Ad Hoc File Systems for High-Performance Computing. *J. Comput. Sci. Technol.*, 35(1):4–26. (Cited on page 34)

[Byna et al., 2020] Byna, S., Breitenfeld, M. S., Dong, B., Koziol, Q., Pourmal, E., Robinson, D., Soumagne, J., Tang, H., Vishwanath, V., and Warren, R. (2020). Exahdf5: Delivering efficient parallel I/O on exascale computing systems. *J. Comput. Sci. Technol.*, 35(1):145–160. (Cited on pages 10 and 41)

[Cao et al., 2020] Cao, Z., Dong, S., Vemuri, S., and Du, D. H. C. (2020). Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In Noh, S. H. and Welch, B., editors, *18th USENIX Conference on File and Storage Technologies, FAST*

2020, Santa Clara, CA, USA, February 24-27, 2020, pages 209–223. USENIX Association. (Cited on page 28)

[Chamberlin and Boyce, 1974] Chamberlin, D. D. and Boyce, R. F. (1974). SEQUEL: A Structured English Query Language. In Rustin, R., editor, *Proceedings of 1974 ACM-SIGMOD Workshop on Data Description, Access and Control, Ann Arbor, Michigan, USA, May 1-3, 1974, 2 Volumes*, pages 249–264. ACM. (Cited on page 25)

[Chang et al., 2008] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. (2008). Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26. (Cited on page 29)

[Chaudhuri and Dayal, 1997] Chaudhuri, S. and Dayal, U. (1997). An overview of data warehousing and OLAP technology. *SIGMOD Rec.*, 26(1):65–74. (Cited on pages 74 and 75)

[Chen et al., 2013] Chen, J., Wei, Q., Chen, C., and Wu, L. (2013). FSMAC: A file system metadata accelerator with non-volatile memory. In *IEEE 29th Symposium on Mass Storage Systems and Technologies, MSST 2013, May 6-10, 2013, Long Beach, CA, USA*, pages 1–11. DOI: 10.1109/MSST.2013.6558440. (Cited on page 45)

[Chu et al., 2020] Chu, X., LeFevre, J., Montana, A., Robinson, D., Koziol, Q., Alvaro, P., and Maltzahn, C. (2020). Mapping Datasets to Object Storage System. *CoRR*, abs/2007.01789. (Cited on pages 30 and 43)

[Cohen et al., 2006] Cohen, S., Hurley, P., Schulz, K. W., Barth, W. L., and Benton, B. (2006). Scientific formats for object-relational database systems: a study of suitability and performance. *SIGMOD Rec.*, 35(2):10–15. (Cited on page 44)

[Confais et al., 2017] Confais, B., Lebre, A., and Parrein, B. (2017). Performance Analysis of Object Store Systems in a Fog and Edge Computing Infrastructure. *Trans. Large Scale Data Knowl. Centered Syst.*, 33:40–79. (Cited on page 30)

[Conn, 2005] Conn, S. S. (2005). Oltp and olap data integration: a review of feasible implementation methods and architectures for real time data analysis. In *Proceedings. IEEE SoutheastCon, 2005.*, pages 515–520. IEEE. (Cited on page 75)

[Curry et al., 2016] Curry, M. L., Ward, H. L., Danielson, G., and Lofstead, J. F. (2016). An Overview of the Sirocco Parallel Storage System. In Taufer, M., Mohr, B., and Kunkel, J. M., editors, *High Performance Computing - ISC High Performance 2016 International Workshops, ExaComm, E-MuCoCoS, HPC-IODC, IXPUG, IWOPH, P^3MA, VHPC, WOPSSS, Frankfurt, Germany, June 19-23, 2016, Revised Selected Papers*, volume 9945 of *Lecture Notes in Computer Science*, pages 121–129. (Cited on page 34)

[Davis et al., 2019] Davis, J. C., IV, L. G. M., Coghlan, C. A., Servant, F., and Lee, D. (2019). Why aren't regular expressions a lingua franca? an empirical study on the re-use and portability of regular expressions. In Dumas, M., Pfahl, D., Apel, S., and Russo, A., editors, *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and*

Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019, pages 443–454. ACM. (Cited on page 117)

[Davoudian et al., 2018] Davoudian, A., Chen, L., and Liu, M. (2018). A Survey on NoSQL Stores. *ACM Comput. Surv.*, 51(2):40:1–40:43. (Cited on pages 25, 28, 29, and 76)

[Dayarathna et al., 2016] Dayarathna, M., Wen, Y., and Fan, R. (2016). Data center energy consumption modeling: A survey. *IEEE Commun. Surv. Tutorials*, 18(1):732–794. (Cited on page 3)

[Depardon et al., 2013] Depardon, B., Le Mahec, G., and Séguin, C. (2013). Analysis of Six Distributed File Systems. Research report. (Cited on page 33)

[DKRZ/MPI, 2021] DKRZ/MPI (2021). CMIP6 Results. `https://www.dkrz.de/de/kommunikation/klimasimulationen/cmip6-de/ergebnisse/2m-temperatur`. last accessed: 07.08.2022. (Cited on pages 103 and 105)

[Dong et al., 2016] Dong, B., Byna, S., Wu, K., Prabhat, Johansen, H., Johnson, J. N., and Keen, N. (2016). Data Elevator: Low-Contention Data Movement in Hierarchical Storage System. In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*, pages 152–161. (Cited on page 5)

[Duwe and Kuhn, 2021a] Duwe, K. and Kuhn, M. (2021a). Dissecting self-describing data formats to enable advanced querying of file metadata. In Wassermann, B., Malka, M., Chidambaram, V., and Raz, D., editors, *SYSTOR '21: The 14th ACM International Systems and Storage Conference, Haifa, Israel, June 14-16, 2021*, pages 12:1–12:7. ACM. DOI: 10.1145/3456727.3463778. (Cited on pages 19, 39, 137, 152, and 156)

[Duwe and Kuhn, 2021b] Duwe, K. and Kuhn, M. (2021b). Using ceph's bluestore as object storage in HPC storage framework. In Kuhn, M., Duwe, K., Acquaviva, J., Chasapis, K., and Boukhobza, J., editors, *CHEOPS '21: Proceedings of the Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems, In Conjunction with EuroSys 2021, Online Event, United Kingdom, April, 2021*, pages 3:1–3:6. ACM. DOI: 10.1145/3439839.3458734. (Cited on pages 39, 64, 73, and 143)

[Duwe et al., 2020] Duwe, K., Lüttgau, J., Mania, G., Squar, J., Fuchs, A., Kuhn, M., Betke, E., and Ludwig, T. (2020). State of the Art and Future Trends in Data Reduction for High-Performance Computing. *Supercomput. Front. Innov.*, 7(1):4–36. DOI: 10.14529/jsfi200101. (Cited on page 9)

[Erxleben et al., 2022] Erxleben, T. L., Duwe, K., Saak, J., Köhler, M., and Kuhn, M. (2022). Energy efficiency of parallel file systems on an ARM cluster. In *In Twelfth International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies (ENERGY 2022)*, pages 42–48. `https://www.thinkmind.org/index.php?view=article&articleid=energy_2022_1_90_30049`. (Cited on page 3)

[Eyring et al., 2016] Eyring, V., Bony, S., Meehl, G. A., Senior, C. A., Stevens, B., Stouffer, R. J., and Taylor, K. E. (2016). Overview of the coupled model intercomparison project phase 6 (cmip6) experimental design and organization. *Geoscientific Model Development*, 9(5):1937–1958. (Cited on page 103)

[Factor et al., 2005] Factor, M., Meth, K., Naor, D., Rodeh, O., and Satran, J. (2005). Object storage: The future building block for storage systems. In *2005 IEEE International Symposium on Mass Storage Systems and Technology*, pages 119–123. IEEE. (Cited on page 29)

[Feldman, 2019] Feldman, M. (2019). Will HPC Centers Say ADIOS To POSIX I/O? `https://www.nextplatform.com/2019/09/04/will-hpc-centers-say-adios-to-posix-i-o/`. Accessed: 2020-05-18. (Cited on page 24)

[Feser et al., 2020] Feser, J. K., Madden, S., Tang, N., and Solar-Lezama, A. (2020). Deductive optimization of relational data storage. *Proc. ACM Program. Lang.*, 4(OOPSLA):170:1–170:30. (Cited on page 27)

[Foster et al., 2017] Foster, I. T., Ainsworth, M., Allen, B., Bessac, J., Cappello, F., Choi, J. Y., Constantinescu, E. M., Davis, P. E., Di, S., Di, Z. W., Guo, H., Klasky, S., van Dam, K. K., Kurç, T. M., Liu, Q., Malik, A., Mehta, K., Mueller, K., Munson, T. S., Ostrouchov, G., Parashar, M., Peterka, T., Pouchard, L., Tao, D., Tugluk, O., Wild, S. M., Wolf, M., Wozniak, J. M., Xu, W., and Yoo, S. (2017). Computing just what you need: Online data analysis and reduction at extreme scales. In Rivera, F. F., Pena, T. F., and Cabaleiro, J. C., editors, *Euro-Par 2017: Parallel Processing - 23rd International Conference on Parallel and Distributed Computing, Santiago de Compostela, Spain, August 28 - September 1, 2017, Proceedings*, volume 10417 of *Lecture Notes in Computer Science*, pages 3–19. Springer. (Cited on page 112)

[Ghemawat and Dean, 2020] Ghemawat, S. and Dean, J. (2020). LevelDB, A fast and lightweight key/value database library by Google. `https://github.com/google/leveldb`. Accessed: 2020-12-08. (Cited on page 28)

[Godoy et al., 2020] Godoy, W. F., Podhorszki, N., Wang, R., Atkins, C., Eisenhauer, G., Gu, J., Davis, P. E., Choi, J., Germaschewski, K., Huck, K. A., Huebl, A., Kim, M., Kress, J., Kurç, T. M., Liu, Q., Logan, J., Mehta, K., Ostrouchov, G., Parashar, M., Poeschel, F., Pugmire, D., Suchyta, E., Takahashi, K., Thompson, N., Tsutsumi, S., Wan, L., Wolf, M., Wu, K., and Klasky, S. (2020). ADIOS 2: The adaptable input output system. A framework for high-performance data management. *SoftwareX*, 12:100561. (Cited on pages 22 and 152)

[Grawinkel et al., 2015] Grawinkel, M., Nagel, L., Mäsker, M., Padua, F., Brinkmann, A., and Sorth, L. (2015). Analysis of the ECMWF Storage Landscape. In Schindler, J. and Zadok, E., editors, *Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST 2015, Santa Clara, CA, USA, February 16-19, 2015*, pages 15–27. USENIX Association. (Cited on pages 10, 40, and 77)

[Gray et al., 2005] Gray, J., Liu, D. T., Nieto-Santisteban, M. A., Szalay, A. S., DeWitt, D. J., and Heber, G. (2005). Scientific data management in the coming decade. *SIGMOD Rec.*, 34(4):34–41. (Cited on pages 6, 9, and 13)

[Gu et al., 2018] Gu, J., Klasky, S., Podhorszki, N., Qiang, J., and Wu, K. (2018). Querying large scientific data sets with adaptable IO system ADIOS. In Yokota, R. and Wu, W., editors, *Supercomputing Frontiers - 4th Asian Conference, SCFA 2018, Singapore, March 26-29, 2018, Proceedings*, volume 10776 of *Lecture Notes in Computer Science*, pages 51–69. Springer. (Cited on page 8)

[Hartman and Ousterhout, 1995] Hartman, J. H. and Ousterhout, J. K. (1995). The Zebra Striped Network File System. *ACM Trans. Comput. Syst.*, 13(3):274–310. (Cited on page 34)

[HDF-Group, 2019] HDF-Group, T. (2019). Chunking in HDF5. `https://portal.hdfgroup.org/display/HDF5/Chunking+in+HDF5`. last accessed: 27.07.2022. (Cited on pages 78, 80, and 227)

[Henning, 2007] Henning, M. (2007). API design matters. *ACM Queue*, 5(4):24–36. (Cited on pages 102 and 115)

[Huang et al., 2018] Huang, D., Han, D., Wang, J., Yin, J., Chen, X., Zhang, X., Zhou, J., and Ye, M. (2018). Achieving load balance for parallel data access on distributed file systems. *IEEE Trans. Computers*, 67(3):388–402. (Cited on page 80)

[IBM, 2016] IBM (2016). General Parallel File System. `https://www.ibm.com/docs/en/gpfs`. Accessed: 2020-12-08. (Cited on page 31)

[IBM, 2022] IBM (2022). IBM Spectrum Scale Performance and simplicity for the hybrid cloud. `https://www.ibm.com/downloads/cas/GQN4XN15`. last accessed: 17.06.2022. (Cited on page 31)

[Idreos et al., 2012] Idreos, S., Groffen, F., Nes, N., Manegold, S., Mullender, K. S., and Kersten, M. L. (2012). MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Eng. Bull.*, 35(1):40–45. (Cited on page 27)

[II et al., 2014] II, D. A. B., Lakshminarasimhan, S., Zou, X., Gong, Z., Jenkins, J., Schendel, E. R., Podhorszki, N., Liu, Q., Klasky, S., and Samatova, N. F. (2014). Transparent in situ data transformations in ADIOS. In *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2014, Chicago, IL, USA, May 26-29, 2014*, pages 256–266. DOI: 10.1109/CCGrid.2014.73. (Cited on page 20)

[IPCC, 2022] IPCC (2022). *Summary for Policymakers*, page 1–24. Cambridge University Press. (Cited on page 103)

[Isakov et al., 2020] Isakov, M., Rosario, E. D., Madireddy, S., Balaprakash, P., Carns, P. H., Ross, R. B., and Kinsy, M. A. (2020). HPC I/O throughput bottleneck analysis with explainable local models. In Cuicchi, C., Qualters, I., and Kramer, W. T., editors, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, page 33. IEEE/ACM. (Cited on page 9)

[Ismail et al., 2020] Ismail, M., Niazi, S., Berthou, G., Ronström, M., Haridi, S., and Dowling, J. (2020). Hopsfs-s3: Extending object stores with posix-like semantics and more (industry track). In *Middleware '20: Proceedings of the 21st International Middleware Conference Industrial Track, Delft, The Netherlands, December 7-11, 2020*, pages 23–30. ACM. (Cited on page 45)

[Jacob et al., 2008] Jacob, B. L., Ng, S. W., and Wang, D. T. (2008). *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann. (Cited on page 5)

[Kang et al., 2021] Kang, Q., Breitenfeld, M. S., Hou, K., Liao, W., Ross, R. B., and Byna, S. (2021). Optimizing performance of parallel I/O accesses to non-contiguous blocks in multiple array variables. In Chen, Y., Ludwig, H., Tu, Y., Fayyad, U. M., Zhu, X., Hu, X., Byna, S., Liu, X., Zhang, J., Pan, S., Papalexakis, V., Wang, J., Cuzzocrea, A., and Ordonez, C., editors, *2021 IEEE International Conference on Big Data (Big Data), Orlando, FL, USA, December 15-18, 2021*, pages 98–108. IEEE. (Cited on page 12)

[Kim et al., 2020] Kim, H., So, B., Han, W., and Lee, H. (2020). Natural language to SQL: Where are we today? *Proc. VLDB Endow.*, 13(10):1737–1750. (Cited on page 25)

[Klasky et al., 2010] Klasky, S., Schwan, K., Oldfield, R. A., and Lofstead, G. F. (2010). Advanced I/O for large-scale scientific applications. Technical report, Sandia National Laboratories (United States). Funding organisation: US Department of Energy (United States). (Cited on page 24)

[Klasky et al., 2017] Klasky, S., Suchyta, E., Ainsworth, M., Liu, Q., Whitney, B., Wolf, M., Choi, J. Y., Foster, I. T., Kim, M., Logan, J., Mehta, K., Munson, T. S., Ostrouchov, G., Parashar, M., Podhorszki, N., Pugmire, D., and Wan, L. (2017). Exacution: Enhancing scientific data management for exascale. In Lee, K. and Liu, L., editors, *37th IEEE International Conference on Distributed Computing Systems, ICDCS 2017, Atlanta, GA, USA, June 5-8, 2017*, pages 1927–1937. IEEE Computer Society. (Cited on pages 2 and 40)

[Klasky et al., 2018] Klasky, S., Wolf, M., Ainsworth, M., Atkins, C., Choi, J., Eisenhauer, G., Geveci, B., Godoy, W. F., Kim, M., Kress, J., Kurç, T. M., Liu, Q., Logan, J., Maccabe, A. B., Mehta, K., Ostrouchov, G., Parashar, M., Podhorszki, N., Pugmire, D., Suchyta, E., Wan, L., and Wang, R. (2018). A view from ORNL: scientific data research opportunities in the big data age. In *38th IEEE International Conference on Distributed Computing Systems, ICDCS 2018, Vienna, Austria, July 2-6, 2018*, pages 1357–1368. IEEE Computer Society. (Cited on page 40)

[Korenblum et al., 2011] Korenblum, D., Rubin, D. L., Napel, S., Rodriguez, C., and Beaulieu, C. F. (2011). Managing biomedical image metadata for search and retrieval of similar images. *J. Digit. Imaging*, 24(4):739–748. (Cited on page 41)

[Kuhn, 2015] Kuhn, M. (2015). *Dynamically Adaptable I/O Semantics for High Performance Computing*. PhD thesis, University of Hamburg. (Cited on page 4)

[Kuhn, 2017] Kuhn, M. (2017). JULEA: A flexible storage framework for HPC. In Kunkel, J. M., Yokota, R., Taufer, M., and Shalf, J., editors, *High Performance Computing - ISC High Performance 2017 International Workshops, DRBSD, ExaComm, HCPM, HPC-IODC, IWOPH, IXPUG, P^3MA, VHPC, Visualization at Scale, WOPSSS, Frankfurt, Germany, June 18-22, 2017, Revised Selected Papers*, volume 10524 of *Lecture Notes in Computer Science*, pages 712–723. Springer. (Cited on pages 34 and 35)

[Kuhn and Duwe, 2020] Kuhn, M. and Duwe, K. (2020). Coupling Storage Systems and Self-Describing Data Formats for Global Metadata Management. In *2020 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 1224–1230. DOI: 10.1109/CSCI51800.2020.00229. (Cited on pages 39, 63, 147, 148, and 151)

[Kuhn and Duwe, 2023] Kuhn, M. and Duwe, K. (2023). Improving HDF5 Metadata Management Using Key-Value Stores and Database Systems. to be submitted. (Cited on pages 63 and 73)

[Kumar et al., 2019] Kumar, S., Petruzza, S., Usher, W., and Pascucci, V. (2019). Spatially-aware parallel I/O for particle data. In *Proceedings of the 48th International Conference on Parallel Processing, ICPP 2019, Kyoto, Japan, August 05-08, 2019*, pages 84:1–84:10. ACM. (Cited on page 79)

[Lakshman and Malik, 2010] Lakshman, A. and Malik, P. (2010). Cassandra: a decentralized structured storage system. *ACM SIGOPS Oper. Syst. Rev.*, 44(2):35–40. (Cited on page 29)

[Lawson et al., 2022] Lawson, M., Gropp, W., and Lofstead, J. (2022). Empress: Accelerating scientific discovery through descriptive metadata management. *ACM Trans. Storage*. Just Accepted. (Cited on page 41)

[Lawson and Lofstead, 2018] Lawson, M. and Lofstead, J. F. (2018). Using a robust metadata management system to accelerate scientific discovery at extreme scales. In *3rd IEEE/ACM International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems, PDSW-DISCS@SC 2018, Dallas, TX, USA, November 12, 2018*, pages 13–23. IEEE. (Cited on pages 10 and 41)

[Lawson et al., 2017] Lawson, M., Ulmer, C. D., Mukherjee, S., Templet, G., Lofstead, J. F., Levy, S., Widener, P. M., and Kordenbrock, T. (2017). Empress: extensible metadata provider for extreme-scale scientific simulations. In Mohror, K. and Welch, B., editors, *Proceedings of the 2nd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems, PDSW-DISCS@SC 2017, Denver, CO, USA, November 13, 2017*, pages 19–24. ACM. (Cited on pages 10 and 41)

[Lee et al., 2017] Lee, D.-Y., Jeong, K., Han, S.-H., Kim, J.-S., Hwang, J.-Y., and Cho, S. (2017). Understanding write behaviors of storage backends in ceph object store. In *Proceedings of the 2017 IEEE International Conference on Massive Storage Systems and Technology*, volume 10. (Cited on pages 32, 33, and 66)

[Lee et al., 2018] Lee, E., Han, Y., Yang, S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. (2018). How to Teach an Old File System Dog New Object Store Tricks. In Goel, A. and Talagala, N., editors, *10th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2018, Boston, MA, USA, July 9-10, 2018*. USENIX Association. (Cited on pages 30 and 43)

[Lee et al., 2022] Lee, S., Hou, K., Wang, K., Sehrish, S., Paterno, M. F., Kowalkowski, J. B., Koziol, Q., Ross, R. B., Agrawal, A., Choudhary, A. N., and Liao, W. (2022). A case study on parallel HDF5 dataset concatenation for high energy physics data analysis. *Parallel Comput.*, 110:102877. (Cited on page 44)

[Liang et al., 2020] Liang, Z., Lombardi, J., Chaarawi, M., and Hennecke, M. (2020). DAOS: A Scale-Out High Performance Storage Stack for Storage Class Memory. In Panda, D. K., editor, *Supercomputing Frontiers - 6th Asian Conference, SCFA 2020, Singapore, February 24-27, 2020, Proceedings*, volume 12082 of *Lecture Notes in Computer Science*, pages 40–54. Springer. (Cited on page 40)

[Lillaney et al., 2019a] Lillaney, K., Tarasov, V., Pease, D., and Burns, R. C. (2019a). Agni: An efficient dual-access file system over object storage. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019*, pages 390–402. ACM. (Cited on page 43)

[Lillaney et al., 2019b] Lillaney, K., Tarasov, V., Pease, D., and Burns, R. C. (2019b). The case for dual-access file systems over object storage. In Peek, D. and Yadgar, G., editors, *11th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2019, Renton, WA, USA, July 8-9, 2019*. USENIX Association. (Cited on page 43)

[Liu et al., 2018] Liu, H., van Oosterom, P., Tijssen, T., Commandeur, T., and Wang, W. (2018). Managing large multidimensional hydrologic datasets: A case study comparing NetCDF and SciDB. *Journal of Hydroinformatics*, 20(5):1058–1070. (Cited on page 44)

[Liu et al., 2014] Liu, Q., Logan, J., Tian, Y., Abbasi, H., Podhorszki, N., Choi, J. Y., Klasky, S., Tchoua, R., Lofstead, J. F., Oldfield, R., Parashar, M., Samatova, N. F., Schwan, K., Shoshani, A., Wolf, M., Wu, K., and Yu, W. (2014). Hello ADIOS: the challenges and lessons of developing leadership class I/O frameworks. *Concurrency and Computation: Practice and Experience*, 26(7):1453–1473. (Cited on pages 20 and 85)

[Lobedank, 2021] Lobedank, K. (2021). Hdf5-zugriffsmusteranalyse zur datenbankabstrahierung. (Cited on page 72)

[Lofstead, 2010] Lofstead, G. F. (2010). *Extreme scale data management in high performance computing*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, USA. (Cited on page 6)

[Lofstead et al., 2016] Lofstead, J. F., Jimenez, I., Maltzahn, C., Koziol, Q., Bent, J., and Barton, E. (2016). DAOS and friends: a proposal for an exascale storage system. In West, J. and Pancake, C. M., editors, *Proceedings of the International Conference for High Performance*

Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016, pages 585–596. IEEE Computer Society. (Cited on page 39)

[Lofstead et al., 2008] Lofstead, J. F., Klasky, S., Schwan, K., Podhorszki, N., and Jin, C. (2008). Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS). In Kim, Y. and Li, X., editors, *6th International Workshop on Challenges of Large Applications in Distributed Environments, CLADE@HPDC 2008, Boston, MA, USA, June 23, 2008*, pages 15–24. ACM. (Cited on page 20)

[Lofstead et al., 2011] Lofstead, J. F., Polte, M., Gibson, G. A., Klasky, S., Schwan, K., Oldfield, R. A., Wolf, M., and Liu, Q. (2011). Six degrees of scientific data: reading patterns for extreme scale science IO. In Maccabe, A. B. and Thain, D., editors, *Proceedings of the 20th ACM International Symposium on High Performance Distributed Computing, HPDC 2011, San Jose, CA, USA, June 8-11, 2011*, pages 49–60. ACM. (Cited on pages 83 and 85)

[Lofstead et al., 2019] Lofstead, J. F., Ryan, A., and Lawson, M. (2019). Adventures in nosql for metadata management. In Weiland, M., Juckeland, G., Alam, S. R., and Jagode, H., editors, *High Performance Computing - ISC High Performance 2019 International Workshops, Frankfurt, Germany, June 16-20, 2019, Revised Selected Papers*, volume 11887 of *Lecture Notes in Computer Science*, pages 227–239. Springer. (Cited on page 93)

[Lofstead et al., 2009] Lofstead, J. F., Zheng, F., Klasky, S., and Schwan, K. (2009). Adaptable, metadata rich IO methods for portable high performance IO. In *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009*, pages 1–10. IEEE. (Cited on pages 20 and 21)

[Lüttgau et al., 2018] Lüttgau, J., Kuhn, M., Duwe, K., Alforov, Y., Betke, E., Kunkel, J. M., and Ludwig, T. (2018). Survey of Storage Systems for High-Performance Computing. *Supercomput. Front. Innov.*, 5(1):31–58. DOI: 10.14529/jsfi180103. (Cited on pages 19, 30, and 39)

[Lüttgau, 2021] Lüttgau, J. (2021). Decision Support for Workflow-Aware High-Performance Storage Systems. (Cited on page 84)

[Mahgoub et al., 2017] Mahgoub, A., Ganesh, S., Meyer, F., Grama, A., and Chaterji, S. (2017). Suitability of NoSQL systems - Cassandra and ScyllaDB - For IoT workloads. In *9th International Conference on Communication Systems and Networks, COMSNETS 2017, Bengaluru, India, January 4-8, 2017*, pages 476–479. IEEE. (Cited on page 29)

[Makris et al., 2019] Makris, A., Tserpes, K., Spiliopoulos, G., and Anagnostopoulos, D. (2019). Performance Evaluation of MongoDB and PostgreSQL for Spatio-temporal Data. In Papotti, P., editor, *Proceedings of the Workshops of the EDBT/ICDT 2019 Joint Conference, EDBT/ICDT 2019, Lisbon, Portugal, March 26, 2019*, volume 2322 of *CEUR Workshop Proceedings*. CEUR-WS.org. (Cited on page 27)

[Mansouri et al., 2018] Mansouri, Y., Toosi, A. N., and Buyya, R. (2018). Data Storage Management in Cloud Environments: Taxonomy, Survey, and Future Directions. *ACM Comput. Surv.*, 50(6):91:1–91:51. (Cited on pages 26 and 30)

[Mauritsen et al., 2019] Mauritsen, T., Bader, J., Becker, T., Behrens, J., Bittner, M., Brokopf, R., Brovkin, V., Claussen, M., Crueger, T., Esch, M., Fast, I., Fiedler, S., Fläschner, D., Gayler, V., Giorgetta, M., Goll, D. S., Haak, H., Hagemann, S., Hedemann, C., Hohenegger, C., Ilyina, T., Jahns, T., Jimenéz-de-la Cuesta, D., Jungclaus, J., Kleinen, T., Kloster, S., Kracher, D., Kinne, S., Kleberg, D., Lasslop, G., Kornblueh, L., Marotzke, J., Matei, D., Meraner, K., Mikolajewicz, U., Modali, K., Möbis, B., Müller, W. A., Nabel, J. E. M. S., Nam, C. C. W., Notz, D., Nyawira, S.-S., Paulsen, H., Peters, K., Pincus, R., Pohlmann, H., Pongratz, J., Popp, M., Raddatz, T. J., Rast, S., Redler, R., Reick, C. H., Rohrschneider, T., Schemann, V., Schmidt, H., Schnur, R., Schulzweida, U., Six, K. D., Stein, L., Stemmler, I., Stevens, B., von Storch, J.-S., Tian, F., Voigt, A., Vrese, P., Wieners, K.-H., Wilkenskjeld, S., Winkler, A., and Roeckner, E. (2019). Developments in the mpi-m earth system model version 1.2 (mpi-esm1.2) and its response to increasing co2. *Journal of Advances in Modeling Earth Systems*, 11(4):998–1038. (Cited on pages 103 and 105)

[Mazumdar et al., 2019] Mazumdar, S., Seybold, D., Kritikos, K., and Verginadis, Y. (2019). A survey on data storage and placement methodologies for Cloud-Big Data ecosystem. *J. Big Data*, 6:15. (Cited on pages 28 and 29)

[Met Office, 2022] Met Office (2010 - 2022). *Iris: A Python package for analysing and visualising meteorological and oceanographic data sets*. Exeter, Devon, v3.2 edition. last accessed: 02.08.2022. (Cited on page 76)

[Mu et al., 2020] Mu, J., Soumagne, J., Byna, S., Koziol, Q., Tang, H., and Warren, R. (2020). Interfacing HDF5 with a scalable object-centric storage system on hierarchical storage. *Concurr. Comput. Pract. Exp.*, 32(20). (Cited on page 43)

[Mu et al., 2018] Mu, J., Soumagne, J., Tang, H., Byna, S., Koziol, Q., and Warren, R. (2018). A transparent server-managed object storage system for HPC. In *IEEE International Conference on Cluster Computing, CLUSTER 2018, Belfast, UK, September 10-13, 2018*, pages 477–481. IEEE Computer Society. (Cited on page 43)

[Niazi et al., 2017] Niazi, S., Ismail, M., Haridi, S., Dowling, J., Grohsschmiedt, S., and Ronström, M. (2017). Hopsfs: Scaling hierarchical file system metadata using newsql databases. In Kuenning, G. and Waldspurger, C. A., editors, *15th USENIX Conference on File and Storage Technologies, FAST 2017, Santa Clara, CA, USA, February 27 - March 2, 2017*, pages 89–104. USENIX Association. (Cited on page 45)

[Oh et al., 2016] Oh, M., Eom, J., Yoon, J., Yun, J. Y., Kim, S., and Yeom, H. Y. (2016). Performance Optimization for All Flash Scale-Out Storage. In *2016 IEEE International Conference on Cluster Computing, CLUSTER 2016, Taipei, Taiwan, September 12-16, 2016*, pages 316–325. DOI: 10.1109/CLUSTER.2016.11. (Cited on page 32)

[ORNL, 2022] ORNL, O. R. N. L. (2022). ADIOS 2: The Adaptable Input/Output System version 2. `https://adios2.readthedocs.io/en/latest/`. last accessed: 17.06.2022. (Cited on pages 22, 51, and 53)

[Ousterhout and Douglis, 1989] Ousterhout, J. K. and Douglis, F. (1989). Beating the I/O bottleneck: A case for log-structured file systems. *ACM SIGOPS Oper. Syst. Rev.*, 23(1):11–28. (Cited on page 2)

[Pallickara et al., 2012] Pallickara, S. L., Pallickara, S., and Zupanski, M. (2012). Towards efficient data search and subsetting of large-scale atmospheric datasets. *Future Generation Comp. Syst.*, 28(1):112–118. (Cited on page 10)

[Perevalova, 2017] Perevalova, O. (2017). Database VOL-plugin for HDF5. Online `https://wr.informatik.uni-hamburg.de/_media/research:theses:olga_perevalova_database_vol_plugin_for_hdf5.pdf`. (Cited on pages 62 and 63)

[Plattner, 2009] Plattner, H. (2009). A common database approach for OLTP and OLAP using an in-memory column database. In Çetintemel, U., Zdonik, S. B., Kossmann, D., and Tatbul, N., editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 1–2. ACM. (Cited on pages 75 and 76)

[Podhorszki et al., 2017] Podhorszki, N., Liu, Q., Logan, J., Mu, J., Abbasi, H., Choi, J.-Y., and Klasky, S. A. (2017). ADIOS 1.13 User's Manual. (Cited on page 224)

[Podhorszki et al., 2018] Podhorszki, N., Liu, Q., Logan, J., Mu, J., Abbasi, H., Choi, J.-Y., and Klasky, S. A. (2018). ADIOS 1.13.1 USER'S MANUAL. `https://users.nccs.gov/~pnorbert/ADIOS-UsersManual-1.13.1.pdf`. Accessed: 2020-05-15. (Cited on page 8)

[Prabhat and Quincey Koziol, 2014] Prabhat and Quincey Koziol, editors (2014). *High performance parallel I/O*. Chapman and Hall/CRC. (Cited on pages 5, 17, and 18)

[Purandare et al., 2022] Purandare, D. R., Bittman, D., and Miller, E. L. (2022). Analysis and workload characterization of the CERN EOS storage system. In Kuhn, M., Duwe, K., Acquaviva, J., Chasapis, K., and Boukhobza, J., editors, *CHEOPS@EuroSys 2022: Proceedings of the Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems, Rennes, France, 5 April 2022*, pages 1–7. ACM. (Cited on page 77)

[Qian et al., 2019] Qian, Y., Li, X., Ihara, S., Dilger, A., Thomaz, C., Wang, S., Cheng, W., Li, C., Zeng, L., Wang, F., Feng, D., Süß, T., and Brinkmann, A. (2019). LPCC: hierarchical persistent client caching for lustre. In Taufer, M., Balaji, P., and Peña, A. J., editors, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2019, Denver, Colorado, USA, November 17-19, 2019*, pages 88:1–88:14. ACM. (Cited on page 31)

[Raasveldt and Mühleisen, 2018] Raasveldt, M. and Mühleisen, H. (2018). MonetDBLite: An Embedded Analytical Database. *CoRR*, abs/1805.08520. (Cited on page 27)

[Rew et al., 1996] Rew, R., Davis, G., Emmerson, S., and Davies, H. (1996). NetCDF User's Guide - An Interface for Data Access Version 2.4. `http://www-c4.ucsd.edu/netCDF/netcdf-guide/guide_toc.html`. last accessed: 15.07.2022. (Cited on page 43)

[Robinson and Lee, 2021] Robinson, D. and Lee, J. (2021). HDF5 1.13.0 - VFD and VOL Changes. `https://www.hdfgroup.org/wp-content/uploads/2021/09/New-Features-in-HDF5-1.13.0-VFD-and-VOL-Changes-Webinar-9-24-21.pdf`. last accessed: 28.08.2022. (Cited on page 19)

[Rowan et al., 2021] Rowan, M. E., Gott, K. N., Deslippe, J., Huebl, A., Thévenet, M., Lehe, R., and Vay, J. (2021). In-situ assessment of device-side compute work for dynamic load balancing in a gpu-accelerated PIC code. In Robinson, T., editor, *PASC '21: Platform for Advanced Scientific Computing Conference, Geneva, Switzerland, July 5-9, 2021*, pages 10:1–10:11. ACM. (Cited on page 80)

[Sanca and Ailamaki, 2022] Sanca, V. and Ailamaki, A. (2022). Sampling-based AQP in modern analytical engines. In Blanas, S. and May, N., editors, *International Conference on Management of Data, DaMoN 2022, Philadelphia, PA, USA, 13 June 2022*, pages 4:1–4:8. ACM. (Cited on pages 75 and 183)

[Santamaria et al., 2019] Santamaria, P., Oden, L., Gil, E., Becerra, Y., Sirvent, R., Glock, P., and Torres, J. (2019). Evaluating the benefits of key-value databases for scientific applications. In *International Conference on Computational Science*, pages 412–426. Springer. (Cited on page 29)

[Sarawagi and Stonebraker, 1994] Sarawagi, S. and Stonebraker, M. (1994). Efficient organization of large multidimensional arrays. In *Proceedings of the Tenth International Conference on Data Engineering, February 14-18, 1994, Houston, Texas, USA*, pages 328–336. IEEE Computer Society. (Cited on page 85)

[Schultz et al., 2019] Schultz, W., Avitabile, T., and Cabral, A. (2019). Tunable Consistency in MongoDB. *Proc. VLDB Endow.*, 12(12):2071–2081. (Cited on page 29)

[Schulzweida, 2022] Schulzweida, U. (2022). Cdo user guide. (Cited on page 104)

[Selvaganesan and Liazudeen, 2016] Selvaganesan, M. and Liazudeen, M. A. (2016). An Insight about GlusterFS and Its Enforcement Techniques. In *International Conference on Cloud Computing Research and Innovations, ICCCRI 2016, Singapore, Singapore, May 4-5, 2016*, pages 120–127. IEEE Computer Society. (Cited on page 33)

[Semmler et al., 2020] Semmler, T., Danilov, S., Gierz, P., Goessling, H. F., Hegewald, J., Hinrichs, C., Koldunov, N., Khosravi, N., Mu, L., Rackow, T., Sein, D. V., Sidorenko, D., Wang, Q., and Jung, T. (2020). Simulations for cmip6 with the awi climate model awi-cm-1-1. *Journal of Advances in Modeling Earth Systems*, 12(9):e2019MS002009. e2019MS002009 2019MS002009. (Cited on page 103)

[Settlemyer et al., 2021] Settlemyer, B. W., Amvrosiadis, G., Carns, P. H., Ross, R. B., Mohror, K., and Shalf, J. M. (2021). It's time to talk about HPC storage: Perspectives on the past and future. *Comput. Sci. Eng.*, 23(6):63–68. (Cited on pages 2 and 9)

[Siddiqa et al., 2017] Siddiqa, A., Karim, A., and Gani, A. (2017). Big data storage technologies: a survey. *Frontiers Inf. Technol. Electron. Eng.*, 18(8):1040–1070. (Cited on pages 27, 28, and 227)

[Singh and Bawa, 2018] Singh, H. J. and Bawa, S. (2018). Scalable metadata management techniques for ultra-large distributed storage systems - A systematic review. *ACM Comput. Surv.*, 51(4):82:1–82:37. (Cited on page 31)

[Smart et al., 2017] Smart, S. D., Quintino, T., and Raoult, B. (2017). A Scalable Object Store for Meteorological and Climate Data. In *Proceedings of the Platform for Advanced Scientific Computing Conference, PASC 2017, Lugano, Switzerland, June 26 - 28, 2017*, pages 13:1–13:8. ACM. (Cited on pages 30 and 40)

[Smart et al., 2019] Smart, S. D., Quintino, T., and Raoult, B. (2019). A high-performance distributed object-store for exascale numerical weather prediction and climate. In *Proceedings of the Platform for Advanced Scientific Computing Conference, PASC 2019, Zurich, Switzerland, June 12-14, 2019*, pages 16:1–16:11. ACM. (Cited on page 40)

[solid IT, 2021] solid IT (2021). DB-Engines Ranking. `https://db-engines.com/en/ranking`. Accessed: 2021-07-08. (Cited on pages 27 and 29)

[Stender et al., 2008] Stender, J., Kolbeck, B., Hupfeld, F., Cesario, E., Focht, E., Hess, M., Malo, J., and Martí, J. (2008). Striping without Sacrifices: Maintaining POSIX Semantics in a Parallel File System. In Cortes, T., editor, *First USENIX Workshop on Large-Scale Computing, LASCO 2008, June 23, 2008, Boston, MA, USA, Proceedings*. USENIX Association. (Cited on page 6)

[Suneja, 2019] Suneja, N. (2019). Scylladb optimizes database architecture to maximize hardware performance. *IEEE Softw.*, 36(4):96–100. (Cited on page 29)

[Tang et al., 2019] Tang, H., Byna, S., Bailey, S., Lukic, Z., Liu, J., Koziol, Q., and Dong, B. (2019). Tuning object-centric data management systems for large scale scientific applications. In *26th IEEE International Conference on High Performance Computing, Data, and Analytics, HiPC 2019, Hyderabad, India, December 17-20, 2019*, pages 103–112. IEEE. (Cited on page 43)

[Tang et al., 2017] Tang, H., Byna, S., Dong, B., Liu, J., and Koziol, Q. (2017). Someta: Scalable object-centric metadata management for high performance computing. In *2017 IEEE International Conference on Cluster Computing, CLUSTER 2017, Honolulu, HI, USA, September 5-8, 2017*, pages 359–369. IEEE Computer Society. (Cited on page 44)

[Tang et al., 2021] Tang, H., Byna, S., Petersson, N. A., and McCallen, D. (2021). Tuning parallel data compression and I/O for large-scale earthquake simulation. In Chen, Y., Ludwig, H., Tu, Y., Fayyad, U. M., Zhu, X., Hu, X., Byna, S., Liu, X., Zhang, J., Pan, S., Papalexakis, V., Wang, J., Cuzzocrea, A., and Ordonez, C., editors, *2021 IEEE International Conference on Big Data (Big Data), Orlando, FL, USA, December 15-18, 2021*, pages 2992–2997. IEEE. (Cited on pages 3 and 9)

[Tebaldi et al., 2021] Tebaldi, C., Debeire, K., Eyring, V., Fischer, E., Fyfe, J., Friedlingstein, P., Knutti, R., Lowe, J., O'Neill, B., Sanderson, B., van Vuuren, D., Riahi, K., Meinshausen, M., Nicholls, Z., Tokarska, K. B., Hurtt, G., Kriegler, E., Lamarque, J.-F., Meehl, G., Moss, R., Bauer, S. E., Boucher, O., Brovkin, V., Byun, Y.-H., Dix, M., Gualdi, S., Guo, H., John, J. G., Kharin, S., Kim, Y., Koshiro, T., Ma, L., Olivié, D., Panickal, S., Qiao, F., Rong, X., Rosenbloom, N., Schupfner, M., Séférian, R., Sellar, A., Semmler, T., Shi, X., Song, Z., Steger, C., Stouffer, R., Swart, N., Tachiiri, K., Tang, Q., Tatebe, H., Voldoire, A., Volodin, E., Wyser, K., Xin, X., Yang, S., Yu, Y., and Ziehn, T. (2021). Climate model projections from the scenario model intercomparison project (scenariomip) of cmip6. *Earth System Dynamics*, 12(1):253–293. (Cited on page 103)

[Tessier et al., 2017] Tessier, F., Vishwanath, V., and Jeannot, E. (2017). TAPIOCA: an I/O library for optimized topology-aware data aggregation on large-scale supercomputers. In *2017 IEEE International Conference on Cluster Computing, CLUSTER 2017, Honolulu, HI, USA, September 5-8, 2017*, pages 70–80. IEEE Computer Society. (Cited on page 79)

[Tian et al., 2011] Tian, Y., Klasky, S., Abbasi, H., Lofstead, J. F., Grout, R. W., Podhorszki, N., Liu, Q., Wang, Y., and Yu, W. (2011). EDO: improving read performance for scientific applications through elastic data organization. In *2011 IEEE International Conference on Cluster Computing (CLUSTER), Austin, TX, USA, September 26-30, 2011*, pages 93–102. IEEE Computer Society. (Cited on page 85)

[Tian et al., 2012] Tian, Y., Klasky, S., Yu, W., Abbasi, H., Wang, B., Podhorszki, N., Grout, R. W., and Wolf, M. (2012). SMART-IO: system-aware two-level data organization for efficient scientific analytics. In *20th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS 2012, Washington, DC, USA, August 7-9, 2012*, pages 181–188. IEEE Computer Society. (Cited on page 85)

[Top500, 2022] Top500 (2022). 59th edition of Top 500. `https://www.top500.org/`. Accessed: 2022-06-20. (Cited on pages 2 and 3)

[Tull et al., 2013] Tull, C. E., Essiari, A., Gunter, D., Li, X. S., Patton, S. J., and Ramakrishnan, L. (2013). The SPOT Suite project. `http://spot.nersc.gov/`. Accessed: 2020-10-09. (Cited on page 41)

[Uselton et al., 2010] Uselton, A., Howison, M., Wright, N. J., Skinner, D., Keen, N., Shalf, J., Karavanic, K. L., and Oliker, L. (2010). Parallel I/O performance: From events to ensembles. In *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Conference Proceedings*, pages 1–11. IEEE. (Cited on page 9)

[van Renen et al., 2019] van Renen, A., Vogel, L., Leis, V., Neumann, T., and Kemper, A. (2019). Persistent memory I/O primitives. In Neumann, T. and Salem, K., editors, *Proceedings of the 15th International Workshop on Data Management on New Hardware, DaMoN 2019, Amsterdam, The Netherlands, 1 July 2019*, pages 12:1–12:7. ACM. (Cited on page 2)

[Vef et al., 2020a] Vef, M., Moti, N., Süß, T., Tacke, M., Tocci, T., Nou, R., Miranda, A., Cortes, T., and Brinkmann, A. (2020a). GekkoFS - A Temporary Burst Buffer File System for HPC Applications. *J. Comput. Sci. Technol.*, 35(1):72–91. (Cited on page 34)

[Vef et al., 2020b] Vef, M., Steiner, R., Salkhordeh, R., Steinkamp, J., Vennetier, F., Smigielski, J., and Brinkmann, A. (2020b). Delvefs - an event-driven semantic file system for object stores. In *IEEE International Conference on Cluster Computing, CLUSTER 2020, Kobe, Japan, September 14-17, 2020*, pages 35–46. IEEE. (Cited on page 43)

[Vef et al., 2018] Vef, M., Tarasov, V., Hildebrand, D., and Brinkmann, A. (2018). Challenges and Solutions for Tracing Storage Systems: A Case Study with Spectrum Scale. *ACM Trans. Storage*, 14(2):18:1–18:24. (Cited on page 32)

[Vilayannur et al., 2004] Vilayannur, M., Ross, R. B., Carns, P. H., Thakur, R., Sivasubramaniam, A., and Kandemir, M. T. (2004). On the Performance of the POSIX I/O Interface to PVFS. In *12th Euromicro Workshop on Parallel, Distributed and Network-Based Processing (PDP 2004), 11-13 February 2004, A Coruna, Spain*, pages 332–339. DOI: 10.1109/EMPDP.2004.1271463. (Cited on page 33)

[Vora, 2011] Vora, M. N. (2011). Hadoop-HBase for large-scale data. In *Proceedings of 2011 International Conference on Computer Science and Network Technology*, volume 1, pages 601–605. (Cited on page 29)

[Wan et al., 2022] Wan, L., Huebl, A., Gu, J., Poeschel, F., Gainaru, A., Wang, R., Chen, J., Liang, X., Ganyushin, D., Munson, T. S., Foster, I. T., Vay, J., Podhorszki, N., Wu, K., and Klasky, S. (2022). Improving I/O performance for exascale applications through online data layout reorganization. *IEEE Trans. Parallel Distributed Syst.*, 33(4):878–890. (Cited on pages 78, 79, 80, 81, 82, and 83)

[Wang et al., 2021] Wang, C., Mohror, K., and Snir, M. (2021). File system semantics requirements of HPC applications. In Laure, E., Markidis, S., Verbanescu, A. L., and Lofstead, J. F., editors, *HPDC '21: The 30th International Symposium on High-Performance Parallel and Distributed Computing, Virtual Event, Sweden, June 21-25, 2021*, pages 19–30. ACM. (Cited on pages 6 and 34)

[Warnke, 2019] Warnke, B. (2019). Integrating self-describing data formats into file systems. Master's thesis, Universität Hamburg. (Cited on pages 68, 70, 72, and 73)

[Wei et al., 2015] Wei, Q., Chen, J., and Chen, C. (2015). Accelerating file system metadata access with byte-addressable nonvolatile memory. *ACM Trans. Storage*, 11(3):12:1–12:28. (Cited on page 45)

[Weil et al., 2006] Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D. E., and Maltzahn, C. (2006). Ceph: A Scalable, High-Performance Distributed File System. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, pages 307–320. (Cited on page 32)

[Weil et al., 2007] Weil, S. A., Leung, A. W., Brandt, S. A., and Maltzahn, C. (2007). RADOS: a scalable, reliable storage service for petabyte-scale storage clusters. In *Proceedings of the 2nd International Petascale Data Storage Workshop (PDSW '07), November 11, 2007, Reno, Nevada, USA*, pages 35–44. DOI: 10.1145/1374596.1374606. (Cited on page 32)

[Wikipedia, 2021] Wikipedia (2021). Hilbert Curve. `https://de.wikipedia.org/wiki/Hilbert-Kurve`. last accessed: 01.08.2022. (Cited on page 85)

[Wu et al., 2017] Wu, T., Chou, J. C., Podhorszki, N., Gu, J., Tian, Y., Klasky, S., and Wu, K. (2017). Apply block index technique to scientific data analysis and I/O systems. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2017, Madrid, Spain, May 14-17, 2017*, pages 865–871. IEEE Computer Society / ACM. (Cited on page 8)

[Yang et al., 2015] Yang, F., Dou, K., Chen, S., Hou, M., Kang, J., and Cho, S. (2015). Optimizing NoSQL DB on Flash: A Case Study of RocksDB. In *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom), Beijing, China, August 10-14, 2015*, pages 1062–1069. IEEE Computer Society. (Cited on page 28)

[Yang and Lilja, 2018] Yang, J. and Lilja, D. J. (2018). Reducing Relational Database Performance Bottlenecks Using 3D XPoint Storage Technology. In *17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications / 12th IEEE International Conference On Big Data Science And Engineering, TrustCom/BigDataSE 2018, New York, NY, USA, August 1-3, 2018*, pages 1804–1808. IEEE. (Cited on page 27)

[Zhang et al., 2019] Zhang, W., Byna, S., Tang, H., Williams, B., and Chen, Y. (2019). MIQS: metadata indexing and querying service for self-describing file formats. In Taufer, M., Balaji, P., and Peña, A. J., editors, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2019, Denver, Colorado, USA, November 17-19, 2019*, pages 5:1–5:24. ACM. (Cited on pages 8, 10, 41, and 111)

Appendices

A. List of Publications

- Erxleben, T. L.; Duwe, K.; Saak, J.; Köhler, M.; Kuhn, M. (2022). Green Storage: Parallel File Systems on ARM. *International Journal on Advances in Software* , ISSN: 1942-2628,https://www.iariajournals.org/software/tocv15n34.html

- Erxleben, T. L.; Duwe, K.; Saak, J.; Köhler, M.; Kuhn, M. (2022). Energy efficiency of parallel file systems on an ARM cluster. *In Twelfth International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies, ENERGY 2022, Venice, Italy, 22. - 26. May 2022, ENERGY 2022*, pages 42-48, IARIA. https://www.thinkmind.org/index.php?view=article&articleid=energy_2022_1_90_30049

- Duwe, K. and Kuhn, M. (2021). Dissecting Self-Describing Data Formats to Enable Advanced Querying of File Metadata. In Wassermann, B., Malka, M., Chidambaram, V., and Raz, D., editors, *SYSTOR '21: The 14th ACM International Systems and Storage Conference, Haifa, Israel, June 14-16, 2021*, pages 12:1–12:7. ACM. https://doi.org/10.1145/3456727.3463778

- Duwe, K. and Kuhn, M. (2021). Using Ceph's Bluestore as Object Storage in HPC Storage Framework. In Kuhn, M., Duwe, K., Acquaviva, J., Chasapis, K., and Boukhobza, J., editors, *CHEOPS '21: Proceedings of the Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems, In Conjunction with EuroSys 2021, Online Event, United Kingdom, April, 2021*, pages 3:1–3:6. ACM. https://doi.org/10.1145/3439839.3458734

- Duwe, K., Lüttgau, J., Mania, G., Squar, J., Fuchs, A., Kuhn, M., Betke, E., and Ludwig, T. (2020). State of the Art and Future Trends in Data Reduction for High-Performance Computing. *Supercomput. Front. Innov.*, 7(1):4–36. DOI: 10.14529/jsfi200101. https://doi.org/10.14529/jsfi200101

- Kuhn, M. and Duwe, K. (2020). Coupling Storage Systems and Self-Describing Data Formats for Global Metadata Management. *In 2020 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 1224–1230. https://doi.org/10.1109/CSCI51800.2020.00229

- Lüttgau, J., Kuhn, M., Duwe, K., Alforov, Y., Betke, E., Kunkel, J. M., and Ludwig, T. (2018). Survey of Storage Systems for High-Performance Computing. *Supercomput. Front. Innov.*, 5(1):31–58. https://doi.org/10.14529/jsfi180103

B. Reproducibility Artifacts

All results and code will be openly available at:

- Measurement results, reproducibility artefacts, for example, a detailed list of the used hardware can be found at: `https://github.com/Bella42/Thesis-results`
- JULEA Engines: `https://github.com/julea-io/adios2`
- JULEA DAI: `https://github.com/Bella42/julea/tree/dai`

ADIOS2.8.3 Unfortunately, version 2.8.3 did not compile on the ants cluster. In order to evaluate BP5, a different route had to be taken: using the ADIOS2.8.3 from spack. However, there were some tweaks required to run the heatTransfer application.

- Use spack adios2.8.3 m̂pich gcc 11.2.0; Details can be found in Listing B.3
- Create patch from commit in ADIOS2 fork to include benchmark output (`https://github.com/Bella42/ADIOS2/commit/1cea8e5e3cda53e5bac09634c8543e6891f5f018`)
- Change build parameters: set BuildExamples to true, otherwise, the heatTransfer application is not built
- Patch spack adios2 `spack patch adios2 m̂pich`
- Keep stage, so that binaries are still there; `spack install –keep-stage adios2 m̂pich`
- Copy build spack from /tmp to home directory, so that it can be used from the compute nodes (has no access to login nodes /tmp)

Cluster Hardware

```
dd if=/dev/zero of=dummy bs=4k count=10000 oflag=sync status=progress
36290560 bytes (36 MB, 35 MiB) copied, 9 s, 4.0 MB/s
40960000 bytes (41 MB, 39 MiB) copied, 9.93784 s, 4.1 MB/s

dd if=/dev/zero of=dummy bs=4M count=1000 oflag=sync status=progress
4139778048 bytes (4.1 GB, 3.9 GiB) copied, 26 s, 159 MB/s
4194304000 bytes (4.2 GB, 3.9 GiB) copied, 26.4298 s, 159 MB/s
```

Listing B.1: Hardware performance when writing with sync (block size 4k 4M)

```
dd if=/dev/zero of=dummy bs=4k count=10000 status=progress
40960000 bytes (41 MB, 39 MiB) copied, 0.0237758 s, 1.7 GB/s

dd if=/dev/zero of=dummy bs=4M count=10000 status=progress
38096863232 bytes (38 GB, 35 GiB) copied, 10 s, 3.8 GB/s
41943040000 bytes (42 GB, 39 GiB) copied, 10.9253 s, 3.8 GB/s
```

Listing B.2: Hardware performance when writing without sync (block size 4k 4M)

Spack ADIO2 to Measure BP5 In the following, the specifications for the ADIOS2 spack package are listed.

```
spack spec adios2 ^mpich
Input spec
--------------------------------
adios2
    ^mpich

Concretized
--------------------------------
adios2@2.8.3%gcc@11.2.0+blosc+bzip2~cuda~dataman~dataspaces+fortran~hdf5~ipo+mpi+pic+png
    ↪ ~python+shared+ssc+sst+sz+zfp build_type=Release patches=5272cee
    ↪ arch=linux-centos8-x86_64_v3
    ^bzip2@1.0.8%gcc@11.2.0~debug~pic+shared arch=linux-centos8-x86_64_v3
        ^diffutils@3.8%gcc@11.2.0 arch=linux-centos8-x86_64_v3
            ^libiconv@1.16%gcc@11.2.0 libs=shared,static arch=linux-centos8-x86_64_v3
    ^c-blosc@1.21.1%gcc@11.2.0+avx2~ipo build_type=RelWithDebInfo
        ↪ arch=linux-centos8-x86_64_v3
        ^cmake@3.23.2%gcc@11.2.0~doc+ncurses+ownlibs~qt build_type=Release
            ↪ arch=linux-centos8-x86_64_v3
            ^ncurses@6.2%gcc@11.2.0~symlinks+termlib abi=none arch=linux-centos8-x86_64_v3
                ^pkgconf@1.8.0%gcc@11.2.0 arch=linux-centos8-x86_64_v3
            ^openssl@1.1.1q%gcc@11.2.0~docs~shared certs=mozilla patches=3fdcf2d
                ↪ arch=linux-centos8-x86_64_v3
                ^ca-certificates-mozilla@2022-07-19%gcc@11.2.0 arch=linux-centos8-x86_64_v3
                ^perl@5.34.1%gcc@11.2.0+cpanm+shared+threads arch=linux-centos8-x86_64_v3
                    ^berkeley-db@18.1.40%gcc@11.2.0+cxx~docs+stl patches=b231fcc
                        ↪ arch=linux-centos8-x86_64_v3
                    ^gdbm@1.19%gcc@11.2.0 arch=linux-centos8-x86_64_v3
                        ^readline@8.1.2%gcc@11.2.0 arch=linux-centos8-x86_64_v3
                    ^zlib@1.2.12%gcc@11.2.0+optimize+pic+shared patches=0d38234
                        ↪ arch=linux-centos8-x86_64_v3
        ^lz4@1.9.3%gcc@11.2.0 libs=shared,static arch=linux-centos8-x86_64_v3
        ^snappy@1.1.9%gcc@11.2.0~ipo+pic+shared build_type=RelWithDebInfo
            ↪ arch=linux-centos8-x86_64_v3
        ^zstd@1.5.2%gcc@11.2.0+programs compression=none libs=shared,static
            ↪ arch=linux-centos8-x86_64_v3
    ^libfabric@1.14.1%gcc@11.2.0~debug~kdreg fabrics=sockets,tcp,udp
        ↪ arch=linux-centos8-x86_64_v3
    ^libffi@3.4.2%gcc@11.2.0 arch=linux-centos8-x86_64_v3
    ^libpng@1.6.37%gcc@11.2.0 arch=linux-centos8-x86_64_v3
```

```
    ^mpich@4.0.2%gcc@11.2.0~argobots~cuda+fortran+hwloc+hydra+libxml2+pci~rocm+romio~slurm
        ↪ ~two_level_namespace~vci~verbs+wrapperrpath datatype-engine=auto device=ch4
        ↪ netmod=ofi pmi=pmi arch=linux-centos8-x86_64_v3
        ^findutils@4.9.0%gcc@11.2.0 patches=440b954 arch=linux-centos8-x86_64_v3
        ^hwloc@2.7.1%gcc@11.2.0~cairo~cuda~gl~libudev+libxml2~netloc~nvml~oneapi-level-zero
            ↪ ~opencl+pci~rocm+shared arch=linux-centos8-x86_64_v3
            ^libpciaccess@0.16%gcc@11.2.0 arch=linux-centos8-x86_64_v3
                ^libtool@2.4.7%gcc@11.2.0 arch=linux-centos8-x86_64_v3
                    ^m4@1.4.19%gcc@11.2.0+sigsegv patches=9dc5fbd,bfdffa7
                        ↪ arch=linux-centos8-x86_64_v3
                        ^libsigsegv@2.13%gcc@11.2.0 arch=linux-centos8-x86_64_v3
                ^util-macros@1.19.3%gcc@11.2.0 arch=linux-centos8-x86_64_v3
            ^libxml2@2.9.13%gcc@11.2.0~python arch=linux-centos8-x86_64_v3
                ^xz@5.2.5%gcc@11.2.0~pic libs=shared,static arch=linux-centos8-x86_64_v3
        ^yaksa@0.2%gcc@11.2.0~cuda~rocm arch=linux-centos8-x86_64_v3
            ^autoconf@2.69%gcc@11.2.0 patches=35c4492,7793209,a49dd5b
                ↪ arch=linux-centos8-x86_64_v3
            ^automake@1.16.5%gcc@11.2.0 arch=linux-centos8-x86_64_v3
            ^python@3.9.13%gcc@11.2.0+bz2+ctypes+dbm~debug+ensurepip+libxml2+lzma~nis
                ↪ ~optimizations+pic+pyexpat+pythoncmd+readline+shared+sqlite3
                ↪ +ssl~tix~tkinter~ucs4+uuid+zlib patches=0d98e93,4c24573,f2fd060
                ↪ arch=linux-centos8-x86_64_v3
                ^expat@2.4.8%gcc@11.2.0+libbsd arch=linux-centos8-x86_64_v3
                    ^libbsd@0.11.5%gcc@11.2.0 arch=linux-centos8-x86_64_v3
                        ^libmd@1.0.4%gcc@11.2.0 arch=linux-centos8-x86_64_v3
                ^gettext@0.21%gcc@11.2.0+bzip2+curses+git~libunistring+libxml2+tar+xz
                    ↪ arch=linux-centos8-x86_64_v3
                    ^tar@1.34%gcc@11.2.0 zip=pigz arch=linux-centos8-x86_64_v3
                        ^pigz@2.7%gcc@11.2.0 arch=linux-centos8-x86_64_v3
                ^sqlite@3.38.5%gcc@11.2.0+column_metadata+dynamic_extensions+fts~functions
                    ↪ +rtree arch=linux-centos8-x86_64_v3
                ^util-linux-uuid@2.37.4%gcc@11.2.0 arch=linux-centos8-x86_64_v3
    ^sz@2.1.12.2%gcc@11.2.0~fortran~hdf5~ipo~netcdf~pastri~python~random_access+shared
        ↪ ~stats~time_compression build_type=RelWithDebInfo arch=linux-centos8-x86_64_v3
    ^zfp@0.5.5%gcc@11.2.0~aligned~c~cuda~fasthash~fortran~ipo~openmp~profile~python
        ↪ +shared~strided~twoway bsws=64 build_type=RelWithDebInfo
        ↪ arch=linux-centos8-x86_64_v3
```

Listing B.3: Spack ADIOS2

MariaDB using Singularity MariaDB is run in a `Singularity` container that uses the current docker image.[1]

```
singularity run --writable-tmpfs -B /tmp/kduwe/localdb:/var/lib/mysql --env
    ↪ MARIADB_RANDOM_ROOT_PASSWORD=yes --env MARIADB_DATABASE=julea_db --env
    ↪ MARIADB_USER=julea_user --env MARIADB_PASSWORD=julea_pw  mariadb.sif &
```

Listing B.4: Starting a MariaDB server using Singularity

[1] `https://github.com/MariaDB/mariadb-docker`

DAI - API

Entry Related Functions

```
1    void j_dai_entry_get_data_d(gchar const* n_space, size_t entry_id, Garray* data);
2
3    void j_dai_entry_get_step(gchar const* n_space, size_t entry_id, size_t* step);
4    void j_dai_entry_get_blockID(gchar const* n_space, size_t entry_id, size_t* block_id);
5
6    void j_dai_entry_meets_condition_d(gchar const* n_space, gchar const* file, gchar
         ↪ const* var, size_t step, size_t block, JDAIStatistic stat, double threshold,
         ↪ JDAIOperator op, bool* result);
7
8    void j_dai_block_get_stat_d(gchar const* n_space,gchar const* file, gchar const* var,
         ↪ size_t step, size_t block, JDAIStatistic stat, double* result);
```

Listing B.5: DAI Interface: Entry/Block related Functions

Entry Related Functions

```
1    void j_dai_query_get_ids_d(gchar const* n_space, gchar const* file, gchar const* var,
         ↪ JDAIGranularity g, JDAIStatistic s, JDAIOperator op, double threshold, Garray*
         ↪ ids);
2
3    void j_dai_range_query_get_ids_d(gchar const* n_space, gchar const* file, gchar const*
         ↪ var, JDAIGranularity g, JDAIStatistic s, double min, double max, Garray* ids);
4
5    void j_dai_query_get_global_stat_d(gchar const* n_space, gchar const* file, gchar
         ↪ const* var, JDAIGranularity g, JDAIStatistic global_s, JDAIStatistic s, double*
         ↪ result);
```

Listing B.6: DAI Interface: Query related Functions

C. User Survey

In the following, the complete survey, i.e. intro all questions and answer options as well as the outro. Alls answers can be found at `https://github.com/Bella42/Thesis-results`

Intro Welcome! Thank you very much for taking the time to complete this short survey. The answers will help me with the interface design in my PhD thesis.

All the best, Kira

For further information visit our project website: `https://parcio.ovgu.de/Research/CoSEMoS.html` There are 18 questions in this survey.

Background Information

Some information about the research area you are working in

Question 0 Example Question. All questions in this survey will be multiple choice. So, please select all answers that apply to you :-)

There will also be the possibility to add additional information. However, there is no need to do so. Note you do not have to tick each box with your mouse. You can also use:

- TAB to switch between boxes (and answers)
- SPACE to select an answer

Please choose all that apply:

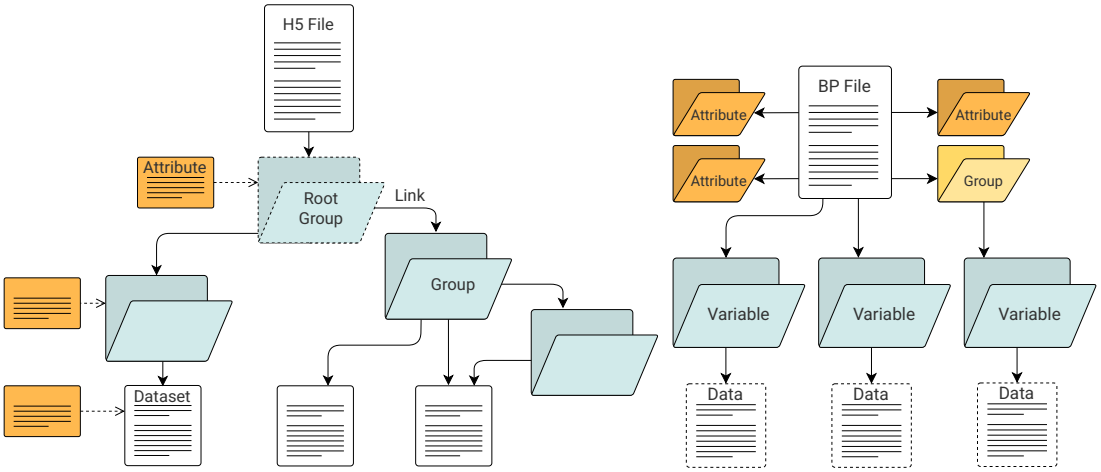
- Got it!
- Other:

Question 1.1 Where do you work?

- Academia
- Industry
- Other:

Question 1.2 In which scientific fields do you work?

- Computer science
- Physics
- Chemistry
- Biology
- Other:



(a) HDF5 Data Model **(b)** ADIOS2 Data Model

Figure C.1.: Creating subfigures in LaTeX.

Questions about SDDF usage

The following questions use the terminology of HDF5/ADIOS2 as these are widely used. For clarity their data models are depicted below. If you are not familiar with either of them, please still answer them as best as you can for the formats you use :-)

Question 2.1 Which self-describing data formats do you use?

- HDF5 (Hierarchical Data Format)
- NetCDF (Network Common Data Form)
- pNetCDF (parallel NetCDF)
- ADIOS2 BP3/BP4/BP5 (Adaptable IO System)
- ROOT
- Zarr
- FITS (Flexible Image Transport System)
- NeXuS (used for neutrons, x-ray, muons, ...)
- GRIB (Grid in Binary)
- FMF (Full-Metadata Format)
- Silo
- SDF (Simple Data Format)
- GeoTIFF (Tag Image File Format)
- NEXUS (used in bioinformatics)
- FASTA (also called FNA (Fasta Nucelic Acid) or FAA)
- Other:

Question 2.2 How many variables (ADIOS2)/ datasets (HDF5) do you use per file?

- 1 - 10
- 10 - 50
- 50 - 100

- 100 - 250
- 250 - 500
- 500 - 1000
- > 1000
- Other:

Question 2.3 How deep are the used hierarchies, that is how many sub groups does a group have typically?

- 1
- 2
- 3
- 4
- 5 - 10
- 10 - 20
- > 20
- Other:

Question 2.4 How many attributes do you use per file?

- 1 - 10
- 10 - 50
- 50 - 100
- 100 - 250
- 250 - 500
- 500 - 1000
- > 1000
- Other:

Question 2.5 How many time steps do you use per file?

- 1 - 20
- 20 - 100
- 500 - 1000
- > 1000
- > 5000
- > 10,000
- > 50,000
- > 100,000
- > 1,000,000
- > 100,000,000
- Other:

Question 2.6 Which answers describe your file usage the best? One file ...

- per timestep
- per variable/dataset
- per thread
- per process
- per node
- per region
- Other:

Question 2.7 How many files are produced per simulation run?

- < 10
- > 10
- > 50
- > 100
- > 1,000
- Other:

Question 2.8 How large is a file typically?

- kB
- MB
- GB
- TB
- PB
- Other:

Post-processing and analysis

The following questions are about the programming languages used in your field and your preferences regarding interfaces to access data.

Question 3.1 Which programming languages are typically used for simulations in your field?

- Fortran
- C
- C++
- Java
- Other:

Question 3.2 Which programming languages are typically used for post-processing and analysis in your field?

- Python
- C
- C++
- Matlab
- R
- Fortran
- Perl
- Other:

Question 3.3 Which programming languages would you rather use and for which tasks?

- Rust
- Go
- None
- Other:

Question 3.4 What are typical operations computed often in post-processing and analysis in your field?

- Reductions like min, max, mean, sum, variance, ...
- outliers...
- comparisons to observation data
- histograms
- Other:

Question 3.5 Which libraries and tools are used for post-processing/analysis?

- Xarray
- Dask
- Pandas
- Zarr
- CDO (Climate Data Operators)
- Apache Spark
- Other:

Question 3.6 What type of interface do you prefer to access data? If you do not like any of them, please add your favourite in the next and final question. For everyone not familiar with all these interfaces, small read examples are given below, so that you can still add your opinion.

POSIX API example[1]:

```
ssize_t pread(int fd, void *buf, size_t count, off_t offset);

int file_descriptor;
off_t offset = 5;
char buf[]="Test text";

file_descriptor = open("test.output", O_RDONLY);
pread(file_descriptor, buf, ((sizeof(buf)-1)-offset), offset);
close(file_descriptor);
```

ADIOS2 example[2]:

```

C++ template<class T> void Get(Variable<T> variable, T *data, const Mode launch =
    ↪ Mode::Deferred);
C   adios2_error adios2_get(adios2_engine *engine, adios2_variable *variable, void *data,
    ↪ const adios2_mode launch);

adios2::ADIOS adios(settings.configfile, mpiReaderComm);
std::vector myFloats;
adios2::IO io = adios.DeclareIO("ReadBP");
adios2::Engine reader = io.Open("exampleFile.bp", adios2::Mode::Read);
```

[1] example from https://www.ibm.com/docs/en/i/7.2?topic=ssw_ibm_i_72/apis/pread.htm

[2] based on https://github.com/ornladios/ADIOS2/blob/master/examples/heatTransfer/read/heatRead.cpp

```

  reader.BeginStep(adios2::StepMode::Read);
  adios2::Variable<float> bpFloats = io.InquireVariable<float>("bpFloats");
  reader.Get<float>(bpFloats, myFloats.data(), adios2::Mode::Sync);
  reader.EndStep();
  reader.Close();
```

SQL example:

```
  SELECT mean_temperature
  FROM variable_table
  WHERE variable_name = "temperature"
  GROUP BY coordinates;

```

Python example: `https://examples.dask.org/dataframes/01-data-access.html`

```
  fileObject = open("sample.txt", "r")
  data = fileObject.read()

  using dask
  import dask.dataframe as dd
  df = dd.read_csv('data/2000-*-*.csv', parse_dates=['timestamp'])
  df.groupby('name').x.mean().compute()
```

Please choose the appropriate response for each item:

- Low level APIs like POSIX or MPI-I/O
- I/O libraries like HDF5, NetCDF or ADIOS2
- Database interfaces using SQL
- Interface like those offered by Python

Response options were: "Hell no", "Rather not", "Is ok", "I like using this", "Love it"

Question 3.7 Any feedback you want to leave? Did I forget your favourite candidates? Please write your answer here:

Outro Thank you so much for finishing the survey. Have a great day! All the best, Kira For further information visit our project website: `https://parcio.ovgu.de/Research/CoSEMoS.html`

Used Formats

The survey and the its conditions will be explained in detail in Chapter 5. In Figure C.2, the answers are shown which formats are used. The most used format is HDF5, followed by NetCDF and pNetCDF. The figure also groups the format according to their typical area of application.

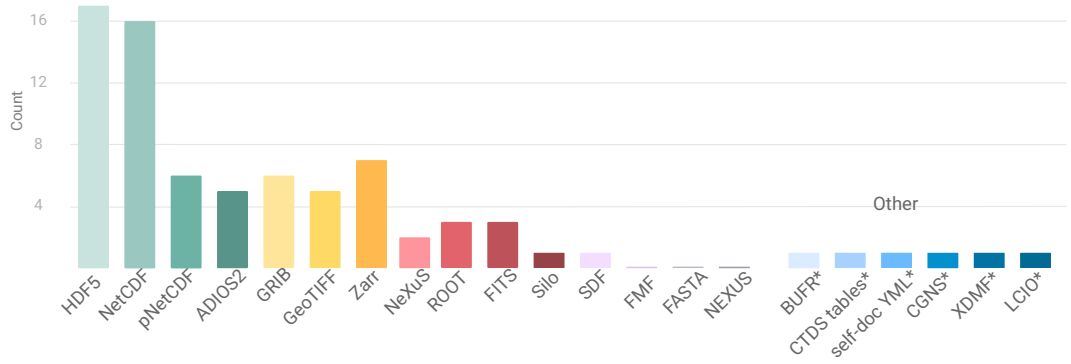


Figure C.2.: Survey: Used self-describing data formats categorised roughly by application fields: Multi-domain formats (green), often used in climate/earth system science (yellow), physics (red), miscellaneous e.g. bio-informatics (lilac), additional formats added by participants (blue)

D. Code Examples

ADIOS2

BP4 File Structure

```
name.bp/      // directory containing data subfiles and metadata files
  data.0
  data.1
  ...
  data.M
  md.0
  md.idx
  profiling.json
```

Listing D.1: BP4 File structure: BP4 files are always a directory containing data subfiles (data.0 - data.M), a metadata file (md.0) and a metadata index (md.idx). The subfile names are static and do not contai the variable name. The number of subfiles (M) by default equals the number of nodes that are used during writing. It can be configured using `NumAggregators`. The profiling option is set by default and a profiling.json is written as well. An example profiling.json can be found in Listing D.9.

BP5 File Structure

```
  name.bp/      % data and metadata files
    data.0
    data.1
    ...
    data.M
    md.0
    md.idx
    mmd.0
```

Listing D.2: BP5 File structure: BP5 files are always a directory containing data subfiles (data.0 - data.M), a metadata file (md.0) and a metadata index (md.idx) like BP4. BP5 has an additional metametadata file (mmd.0). The subfile names are static and do not contai the variable name. The number of subfiles (M) by default equals the number of nodes that are used during writing. It can be configured using `NumAggregators`. The profiling option is set by default and a profiling.json is written as well. An example profiling.json can be found in Listing D.9.

Index Table File for BP4

```
1  ========================================================
2      Index Table File: globalArray-AdiosExample.bp/md.idx
3  ========================================================
4  ------------------------------------------------------------------------------
5  | Version string  | Major | Minor | Patch | X   | Endian | BP v. | Active |  X  |
6  |   32 bytes      |   1B  |   1B  |   1B  | 1B  |   1B   |   1B  |   1B   | 25B |
7  +------------------------------------------------------------------------------+
8  | ADIOS-BP v2.7.1 |   2   |   7   |   1   |     |  yes   |   4   |   no   |     |
9  ------------------------------------------------------------------------------
10 ---------------------------------------------------------------
11 | Step | Rank | PGPtr | VarPtr | AttPtr | EndPtr | Timestamp  | X |
12 +---------------------------------------------------------------+
13 |    1 |    0 |    64 |    110 |    260 |    272 | 1655548298 | 0 |
14 |    2 |    0 |   272 |    318 |    468 |    480 | 1655548298 | 0 |
15 |    3 |    0 |   480 |    526 |    676 |    688 | 1655548298 | 0 |
16 |    4 |    0 |   688 |    734 |    884 |    896 | 1655548298 | 0 |
17 |    5 |    0 |   896 |    942 |   1092 |   1104 | 1655548298 | 0 |
```

Listing D.3: BP4 Index table file containing the metadata index information.The table at the top describes the used library version, the endianness as well as the format version. The first two columns of the second table in Line 11 show that the file, to which this metadata index belongs, was written with one MPI process and a total of 5 steps. The following columns store the respective pointer as well as the time stamps.
Abbreviations: X: unused bytes, BP v.: BP file format version, PGPtr: Process Group Pointer, VarPtr: Variable Pointer, AttPtr: Attribute Pointer, EndPtr: End Pointer.

BP5 Format Details

```
1  MetadataIndex file (md.idx)
2    BP5 header for "Index Table" (64 bytes)
3      for each Writer, what aggregator writes its data
4        uint16_t [ WriterCount]
5      for each timestep:   (size (WriterCount + 2 ) 64-bit ints
6        uint64_t 0 :  CombinedMetaDataPos
7        uint64_t 1 :  CombinedMetaDataSize
8        uint64_t 2 :  FlushCount
9        for each Writer:
10         for each flush before the last:
11           uint64_t  DataPos (in the file above)
12           uint64_t  DataSize
13         for the final flush:
14             uint64_t  DataPos (in the file above)
15   Notes - Each timestep takes this size:
16   sizeof(uint64_t)* (3 + ((FlushCount-1)*2 + 1) * WriterCount) bytes
17
18 MetaMetadata file (mmd.0) contains FFS format information
19   for each meta metadata item:
20     uint64_t  MetaMetaIDLen
```

```
      uint64_t  MetaMetaInfoLen
      char[MeatMetaIDLen] MetaMetaID
      char[MetaMetaInfoLen] MetaMetanfo
    Notes: This file should be quite small,
    with size dependent upon the number of different "formats" written by any rank.

  MetaData file (md.0) contains encoded metadata/attribute data
    BP5 header for "Metadata" (64 bytes)
    for each timestep:
      uint64_t  : TotalSize of this metadata block
      uint64_t[WriterCount]  : Length of each rank's metadata
      uint64_t[WriterCount]  : Length of each rank's attribute
      FFS-encoded metadata block of the length above
      FFS-encoded attribute data block of the length above

  Data file (data.x) contains a block of data for each timestep, for each rank
```

Listing D.4: BP5 Format Files: md.idx, mmd.0, md.0 and data.x

ADIOS2: Range queries

In the following, an example is shown to illustrate the functionality that ADIOS2 offers. A range query can be performed using ADIOS2 without any additions, as shown in Listing D.5. It is possible to retrieve the metadata of a variable using `AllStepsBlocksInfo` and check the minimum, maximum or both. There might be good reasons to define a new interface, like improved performance or the option to tailor the interface to a specific use case. Still, it seemed sensible to highlight the options that provide the same or similar functionality in an established interface to allow a better comparison.

```
auto tempVar = inIO.InquireVariable<double>("T");
auto blocksInfoAllSteps = tempVar.AllStepsBlocksInfo();
std::vector<size_t> stepIDs;

for (auto blocksInfoPerStep : blocksInfoAllSteps){
  for (auto block : blocksInfoPerStep){
      if ((block.Max < 42) && (block.Min > -42)){
        std::cout << "Is in range";
      }
  }
}
```

Listing D.5: Example implementation of a range query using ADIOS2

ADIOS2: Writing local arrays

```
const size_t Nglobal = 6;               // v0: same size for each rank
const unsigned int Nx = rand() % 6 + 5; // v1: different size for each rank
unsigned int NumberElements;            // v2: also different each step

std::vector<double> v0(Nglobal);
std::vector<double> v1(Nx);
std::vector<double> v2;

adios2::IO io = adios.DeclareIO("Output");
io.SetEngine("JULEA-KV");

adios2::Variable<double> varV0 =
  io.DefineVariable<double>("v0", {}, {}, {Nglobal});

adios2::Variable<double> varV1 =
  io.DefineVariable<double>("v1", {}, {}, {Nx});

adios2::Variable<double> varV2 =
  io.DefineVariable<double>("v2", {}, {}, {adios2::UnknownDim});

for (int step = 0; step < NSTEPS; step++){
    writer.BeginStep();

    NumberElements = rand() % 6 + 5;
    v2.reserve(NumberElements);

    // Set array size because it was unknown at the time of definition
    varV2.SetSelection(adios2::Box<adios2::Dims>({}, {NumberElements}));
    writer.Put<double>(varV0, v0.data());   // the same for all variables
    writer.EndStep();
  }
writer.Close();
```

Listing D.6: Local Arrays

ADIOS2: Compound Variables

```
  const void *m_Data;
  struct Element
  {
      const std::string Name;
      const adios2::DataType Type; // from GetDataType<T>
      const size_t Offset;         // element offset in struct
  };
  std::vector<Element> m_Elements; //primitve element types defining compound struct
```

Listing D.7: Class Members of VariableCompound

ADIOS2: Block Level Information

```
struct Info {
 // Contains seek information for available [step][blockID]
 std::map<size_t, std::vector<SubStreamBoxInfo>>  StepBlockSubStreamsInfo;
 Dims Shape;
 Dims Start;
 Dims Count;
 Dims MemoryStart;
 Dims MemoryCount;
 std::vector<Operation> Operations;
 size_t Step;
 size_t StepsStart;
 size_t StepsCount;
 size_t BlockID;
 T *Data;
 T Min;
 T Max;
 T Value;
 std::vector<T> MinMaxs;             // sub-block level min-max
 struct helper::BlockDivisionInfo SubBlockInfo;
 T *BufferP;
 std::vector<T> BufferV;
 int WriterID;
 SelectionType Selection;
 bool IsValue;
 bool IsReverseDims;
};
```

Listing D.8: Info Struct

Category	Key	Engine
Streaming through file	OpenTimeoutSecs	BP4, BP5
	BeginStepPollingFrequencySecs	BP4, BP5
	StreamReader	BP4
Aggregation	AggregationType	BP5
	NumAggregators	BP3, BP4, BP5
	AggregatorRatio	BP3, BP4, BP5
	NumSubFiles	BP5
	StripeSize	BP5
	MaxShmSize	BP5
Buffering	BufferVType	BP5
	BufferChunkSize	BP5
	MinDeferredSize	BP5
	InitialBufferSize	BP3, BP4, BP5
	(Buffer)GrowthFactor	BP3, BP4, BP5
	MaxBufferSize	BP3, BP4
	BurstBufferPath	BP4
	BurstBufferDrain	BP4
	BurstBufferVerbose	BP4
Managing Steps	AppendAfterSteps	BP5
	SelectSteps	BP5
	FlushStepsCount	BP3, BP4
Asynchronous Writing	AsyncOpen	BP5
	AsyncWrite	BP5
Direct I/O	DirectIO	BP5
	DirectIOAlignOffset	BP5
	DirectIOAlignBuffer	BP5
Miscellaneous	StatsLevel	BP4
	StatsBlockSize	BP4
	NodeLocal	BP3, BP4
	Node-Local	BP4
	MaxOpenFilesAtOnce	BP5
	Profile	BP3, BP4
	ProfileUnits	BP3, BP4
	Threads	BP3, BP4

Table D.1.: BP Engine Parameters: The native ADIOS2 engines support numerous parameters to help tune the I/O operations and the resulting performance [Podhorszki et al., 2017]. Not all parameters are available for all engines.

Engine Parameters

Profiling JSON

```
1  { "rank":0, "start":"Mon_Aug_29_10:43:55_2022", "bytes":0, "AWD":{"mus":379230,
       ↪ "nCalls":11}, "close_ts":{"mus":282, "nCalls":11}, "meta_lvl1":{"mus":514,
       ↪ "nCalls":11}, "meta_lvl2":{"mus":581, "nCalls":11}, "meta_gather1":{"mus":431,
       ↪ "nCalls":11}, "endstep":{"mus":380643, "nCalls":11},
       ↪ "transport_0":{"type":"File_POSIX", "close":{"mus":5, "nCalls":1},
       ↪ "write":{"mus":237229, "nCalls":33}, "open":{"mus":407, "nCalls":1}},
       ↪ "transport_1":{"type":"File_POSIX", "close":{"mus":1, "nCalls":1},
       ↪ "write":{"mus":200, "nCalls":55}, "open":{"mus":354, "nCalls":1}} },
2  { "rank":1, "start":"Mon_Aug_29_10:43:55_2022", "bytes":0, "AWD":{"mus":362292,
       ↪ "nCalls":11}, "close_ts":{"mus":293, "nCalls":11}, "meta_lvl1":{"mus":115,
       ↪ "nCalls":11}, "meta_lvl2":{"mus":0, "nCalls":11}, "meta_gather1":{"mus":77,
       ↪ "nCalls":11}, "endstep":{"mus":362740, "nCalls":11} }
```

Listing D.9: BP3 File structure: A BP3 file is not a file in the traditional sense as it consists of a file (x.bp) that stores the collective metadata and directory (x.bp.dir/) that contains the data subfiles (.bp.0 - .bp.M). The number of subfiles (M) by default equals the number of nodes that are used during writing. It can be configured using `NumAggregators`. When using a debug build, the profiling option is set by default and a profiling.json is written as well.

JULEA Engines

Serialised BP Format in JULEA-KV

```
1  sizeof(T) min;
2  sizeof(T) max;
3  sizeof(T) value;
4
5  size_t shapeSize = 0;
6  size_t *shape = nullptr;
7  size_t startSize = 0;
8  size_t *start = nullptr;
9  size_t countSize = 0;
10 size_t *count = nullptr;
11 size_t memoryStartSize = 0;
12 size_t *memoryStart = nullptr;
13 size_t memoryCountSize = 0;
14 size_t *memoryCount = nullptr;
15
16 size_t StepsStart = 0;
17 size_t StepsCount = 0;
18 size_t BlockID = 0;
19 size_t CurrentStep = 0;
20 size_t BlockNumber = 0;
21 bool IsValue = false;
```

Listing D.10: Serialised Format

The catalog timestep entry struct from EMPRESS 2

There are more than 70 structs in the EMPRESS 2 interface[1].

```
struct md_catalog_timestep_entry
{
  uint64_t timestep_id;
  uint64_t run_id;
  std::string path;
  uint32_t active;
  uint64_t txn_id;

  template <typename Archive>
  void serialize(Archive& ar, const unsigned int ver)
  {
  ar & timestep_id;
  ar & run_id;
  ar & path;
  ar & active;
  ar & txn_id;
  }
};
```

Listing D.11: The catalog timestep entry struct from EMPRESS 2

[1] https://github.com/mlawsonca/empress/blob/master/include/common/my_metadata_args.h

List of Figures

List of Tables

List of Listings

Ehrenerklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; verwendete fremde und eigene Quellen sind als solche kenntlich gemacht. Insbesondere habe ich nicht die Hilfe eines kommerziellen Promotionsberaters in Anspruch genommen. Dritte haben von mir weder unmittelbar noch mittelbar geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Dissertation stehen.

Ich habe insbesondere nicht wissentlich:

- Ergebnisse erfunden oder widersprüchliche Ergebnisse verschwiegen,
- statistische Verfahren absichtlich missbraucht, um Daten in ungerechtfertigter Weise zu interpretieren,
- fremde Ergebnisse oder Veröffentlichungen plagiiert,
- fremde Forschungsergebnisse verzerrt wiedergegeben.

Mir ist bekannt, dass Verstöße gegen das Urheberrecht, Unterlassungs- und Schadensersatzansprüche des Urhebers sowie eine strafrechtliche Ahndung durch die Strafverfolgungsbehörden begründen kann. Die Arbeit wurde bisher weder im Inland noch im Ausland in gleicher oder ähnlicher Form als Dissertation eingereicht und ist als Ganzes auch noch nicht veröffentlicht.

Kira Isabel Duwe Magdeburg, 21.02.2023