Masterarbeit

Integrating self-describing data formats into file systems

vorgelegt von

Benjamin Warnke

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang:	Informatik
Matrikelnummer:	6676867
Erstgutachter:	Prof. Dr. Thomas Ludwig
Zweitgutachter:	Dr. Michael Kuhn
Betreuer:	Dr. Michael Kuhn
Betreuer:	Kira Duwe

Hamburg, 2019-11-08

Abstract

The computational power of huge supercomputers is increasing exponentially every year. The use of this computational power in scientific research in various fields quickly generates large amounts of data. The amount of data, that can be stored in individual file systems, as well as their access speed, also increases exponentially. Unfortunately, the memory size and speed grow slower than the computational performance. The research needs a tool that makes it possible to quickly find the data of interest to researchers in these huge file collections.

File system interfaces were defined in the early days of computers and have not been modified since their invention. A search for certain data in huge data collections was not intended. This search could be done by custom metadata. This was not directly possible in the past because the Portable Operating System Interface (POSIX) does not contain the required functions.

Self-Describing Data Format (SDDF) were developed for the exchange and reusability of research results between research groups. These are file formats that store metadata along with their raw data, that is the file itself contains the description of the data as well as how it is encoded.

Looking for information according to certain criteria, all eligible files had to be opened.

This thesis describes the development of a new interface as well as the reference-implementation for the combined storage of data and metadata. This novel interface includes functions for storing metadata in a dedicated Structured Query Language (SQL) backend. Storing metadata in a dedicated backend enables a more efficient search process. Since metadata is now stored in a central database, the files which meet the search-criteria can be opened systematically, as many files are excluded from the beginning. This saves time. The existence of metadata in file systems also has the advantage of intelligently distributing file content across different storage nodes. For example, metadata can be written to faster storage mediums than the raw data which is needed less frequently. This saves time and money.

Contents

1 Introduction **5**
1.1 Existing work . . . 8
1.2 Goal . . . 9
1.3 Thesis outline . . . 9

2 State of the art **10**
2.1 File systems . . . 10
2.1.1 Local file systems . . . 11
2.1.2 Distributed file systems . . . 11
2.1.3 Typical file system library stack . . . 12
2.2 HDF5 . . . 12
2.2.1 Usage example . . . 15
2.3 ADIOS2 . . . 16
2.4 JULEA . . . 17

3 Design **20**
3.1 Naive approach . . . 20
3.2 JULEA DB backend . . . 21
3.3 JULEA DB client . . . 25
3.3.1 `JDBSchema` . . . 25
3.3.2 `JDBSelector` . . . 26
3.3.3 `JDBEntry` . . . 27
3.3.4 `JDBIterator` . . . 28
3.4 HDF5 VOL Plugin on top of JULEA DB client . . . 29
3.4.1 HDF5-File . . . 29
3.4.2 HDF5-Link . . . 30
3.4.3 HDF5-Datatype . . . 30
3.4.4 HDF5-Space . . . 31
3.4.5 HDF5-Dataset and HDF5-Attribute . . . 32
3.4.6 HDF5 VOL Plugin example . . . 34

4 Implementation **35**
4.1 Backend Operation . . . 35
4.2 SQL wrapper . . . 37
4.3 Usage example . . . 39
4.3.1 `JDBSchema` . . . 40
4.3.2 `JDBEntry` . . . 42

4.4 Adding new backend implementations . . . 43
4.5 JULEA debugging . . . 43
4.5.1 AFL . . . 43
4.5.2 Mockup . . . 44
4.5.3 Static code analysis . . . 44
4.5.4 Error handling . . . 44
4.5.5 Other . . . 44

5 Evaluation **45**
5.1 Synthetic HDF5 benchmarks . . . 45
5.2 Enzo benchmark - a real application using HDF5 . . . 46
5.3 Synthetic JULEA DB client benchmarks . . . 48
5.3.1 Null-Backend . . . 48
5.3.2 Impact of the used database . . . 50
5.3.3 Impact of the number of fields . . . 51
5.3.4 Impact of Indices . . . 52
5.3.5 Impact of the storage medium . . . 54
5.3.6 Impact of the Process count . . . 55
5.4 New functionality . . . 57

6 Related Work **59**
6.1 Compression . . . 59
6.2 EMPRESS . . . 60
6.3 Dynamic metadata Management for Petabyte-scale File Systems . . . 60
6.4 Semantic file system . . . 61
6.5 Parallel data analysis directly on scientific file formats . . . 61
6.6 Making Sense of File Systems Through Provenance and Rich metadata . 62
6.7 Structured metadata for the JULEA storage framework . . . 63

7 Summary, Conclusion and Future Work **64**
7.1 Summary . . . 64
7.2 Conclusion . . . 64
7.3 Future work . . . 65

Bibliography **66**

Appendices **69**

List of Figures **70**

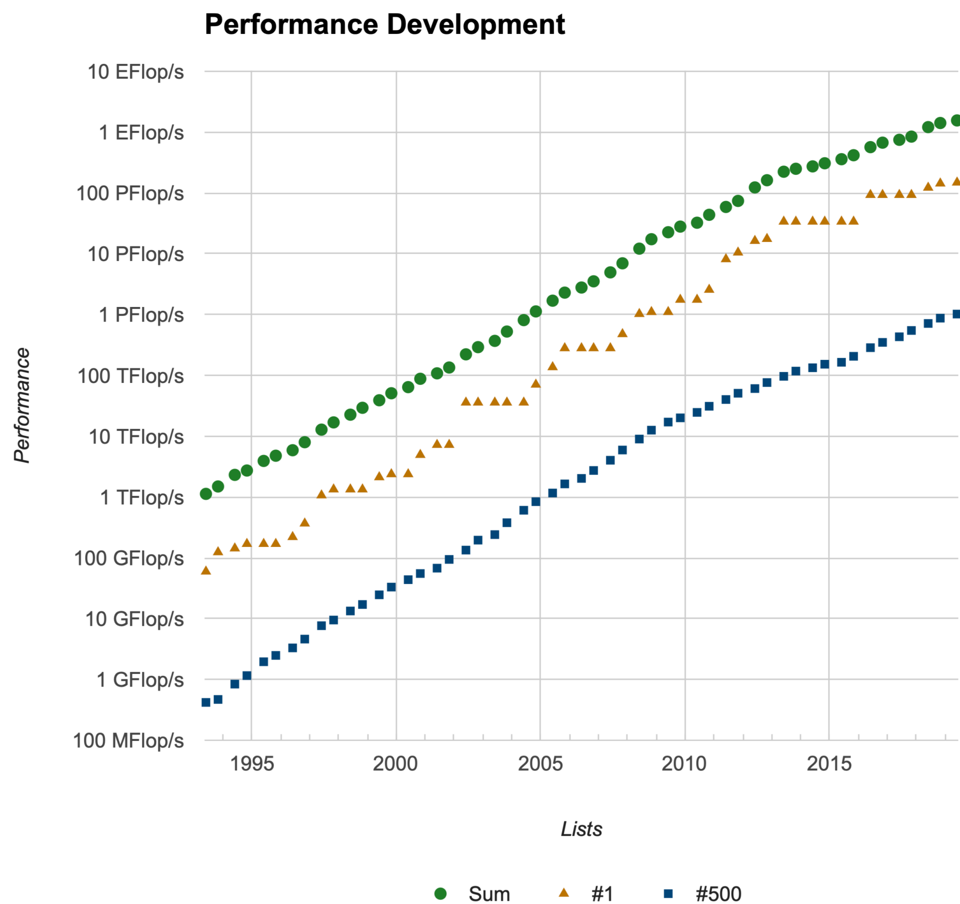
List of Listings **71**

List of Tables **72**

List of Abbreviations **73**

1 Introduction

The computational power of the largest supercomputers in the world is growing exponentially every year. Figure 1.1 shows the performance improvement of the 500 largest supercomputers in the last 25 years. As the computational power increases, so does the amount of data generated and stored.



[ht]

Figure 1.1: Performance development in the TOP500 list [Erich Strohmaier, 2019a] in the last 25 years

The first hierarchical file system for the systematic storage of data was designed in 1965 [Daley and Neumann, 1965]. The storage space of a single hard drive at that time was about one megabyte [IBM, 1965]. Then, only a single computer was connected to a storage disk. These computers were used by a few people each.

The networking of computers and their services have become more complex. Some computers are designed for high-speed computing, others for permanent storage. Today supercomputers consist of hundreds to thousands of compute nodes. Many of the nodes in a supercomputer are connected to the same storage system. Both, the programs for calculation and the file systems for storage are each distributed to different computers. Within the storage nodes there are several hard drives which also have to be coordinated.

On the supercomputers of the TOP500 list [Erich Strohmaier, 2019b] the main applications either simulate complex models or analyze a large amount of sensor or application data. In modern science, many different research areas use simulations [Arie Shoshani, 2019]. During the simulation of complex models, petabytes of data are generated.

Then the data is analyzed. Post-processing tools are used for this analysis to extract the data of interest from the total file collection [Arie Shoshani, 2019]. Often extreme values are of particular interest. The extraction of the information of interest from a growing volume of data can benefit from rich metadata.

The increase of the existing data volume and the access speed in an increasingly complex computer landscape results in novel requirements for file system interfaces.

POSIX has been developed as a file system interface to provide a unified interface for a variety of storage systems for different operating systems.

One of the biggest problems with POSIX is that it is necessary to propagate every change to a file instantly to all other nodes.

If there are only a few nodes, then this is fine. However, current servers have thousands of compute nodes and millions of connected users. This massively parallel access causes immense network traffic to synchronize all file content on all connected nodes.

POSIX does not contain transaction or batch semantics because it was not required at the time of its creation. The benefit of batch semantics is to efficiently write large volumes of data in a batch process. The benefit of transaction semantics is ensuring data consistency across multiple files and operations.

Today, there are many different attempts to define structured data formats with different properties to efficiently support as many application requirements as possible. Due to the limitations contained in the definitions, not every definition is suitable for every application. These definitions are implemented in different libraries. Libraries differ in their interfaces and are therefore incompatible with each other.

Libraries make it easier to describe raw data with metadata. As the simplest variant, small JavaScript Object Notation (JSON) or Extensible Markup Language (XML) files can be used, which are stored in a separate file in addition to the file containing the raw data.

SDDF were developed for the exchange of research results between research groups and reusability. These are file formats that store the metadata together with their raw data, that is, the file itself contains the description of the data as well as how it is encoded. Examples of SDDF are libraries such as Hierarchical Data Format (HDF5) [Group, 2019] and Adaptable Input/Output System (ADIOS2) [Laboratory, 2019].

This results in the following advantages:

- The metadata is always stored together with the raw data and remains consistent. Only the user himself may explicitly rewrite or delete the file.
- Although the architectures of the Central Processing Unit (CPU) are different, these libraries can still use the metadata to read the raw data. Without these libraries, this would not be as easy.

A disadvantage of generic data formats is that the data access speed can be lower compared to proprietary binary data formats. This is due to another library layer between the physical storage and the application.

The Network Common Data Format (NetCDF) was designed as a stand-alone file format. During development, the data format has changed. With the announcement of the fourth version of the NetCDF HDF5 is now used as a storage layer below NetCDF. The intended improvement was the combination of the simpler interface of NetCDF with the increased flexibility of HDF5.

There are also many specialized data formats for different applications. Some of these, such as HDF5 for molecular data (H5MD) [Developers., 2019] merely extend existing generic data formats to support specific types of applications. The advantage is that it may be easier to develop a new application if previously implemented dataformats can be reused. The disadvantage of this type of library is that every layer in the storage stack tries to optimize data access for itself. These additional library layers on top of the generic data format increase the complexity of the entire library stack and thus reduce the flexibility of the structured data format. Because of the complexity, some libraries in the stack can undo previous library improvements, effectively reducing the overall performance of the application.

Each application decides which data formats are supported and which are not. In climatology and geosciences, the file formats HDF5 and NetCDF are often used. This is mainly due to the fact that it was recognized early on in this field of research that the interchangeability of data is an important part of research. To ensure the interchangeability of their data, geoscientists have agreed on a common standard.

Other research areas do not define a universal file format standard. This is where the problem of many coexisting data formats occurs, so that there are many applications that compute similar things but store them in different ways. This makes it harder to compare or reuse results.

So, there are many different applications for particle and material simulation with different output formats. An example of this is EDEM, an application for simulating materials and elements. For 7 years, this application had used its own proprietary data format. For 3 years, the data is now stored in HDF5 format, so users can now more easily compare their data with the results of other applications [Rowan, 2019]. Previously, scientists had to unite the file format with others if they wanted to share and compare their results.

It would be more useful and efficient, however, if the file system itself provided such a flexible interface. Then any application could easily be shared.

1.1 Existing work

Currently, most metadata is stored next to the data within the same file. Typical everyday formats such as Joint Photographic Experts Group (JPEG) and Portable Network Graphics (PNG) for images and Moving Picture Experts Group (MPEG) for video files store their metadata directly inside the file. The file system stores only the most basic metadata, for example, the creation date of the file. There are many different photo and video formats, and each format stores its metadata differently, making it difficult to find specific metadata attributes. The metadata in some of these formats, such as JPEG, may contain duplicate information such as the creation date, which is also stored in the file system.

The same applies to websites. Each Hypertext Markup Language (HTML) page stores its metadata in its own header using special tags, which does not change the visual output of the web page. However, computers can use this metadata to better interpret the content of the page. Similar to the consumer computer, only the basic metadata such as the website Uniform Resource Locator (URL) tree can be specified globally per homepage.

Even in the big supercomputers, metadata, if available, is stored individually in each file.

CoSEMoS

The current storage stack in supercomputers has many problems with data throughput and metadata management. An important cause of this type of problem is the separation of metadata in files and metadata in the underlying file system. As a result, the information about how the file data is structured, is not known within the file system. During post-processing, many files are opened just to read the metadata, if minimum and maximum values are going to be extracted. Another problem is the static and non-configurable Input/Output (IO) semantics in POSIX file systems. The target of the Coupled Storage System for Efficient Management of Self-Describing Data Formats (CoSEMoS) project is to solve this problem. Therefore, CoSEMoS expands JULEA.

To solve the problems shown, CoSEMoS will add a new storage interface to JULEA. Since JULEA is a core dependency of this thesis, it is described in detail in Section 2.4. The interface should provide a simplified access to the files metadata within the file system. Additionally, a global metadata management system will be added. This management system will use the provided metadata to decide which backend should be used to store the data. If the application provides enough metadata, the management system can automatically decide whether to save the data to a Non-Volatile Random-Access Memory (NVRAM), a Solid-State Drive (SSD), a Hard Disk Drive (HDD) or tape storage. JULEA already defines a variety of semantics that can be set by the application depending on the needs of the application. This deep coupling between file system and file metadata allows novel interfaces to analyze large amounts of data more efficiently.

Part of the CoSEMoS project is the implementation of a structured backend where the generic metadata provided by the application can be stored. This structured backend

will be implemented as part of this master thesis.

To simplify the transition for existing applications, CoSEMoS provides plugins for self-describing data formats, such as HDF5 and ADIOS2, to automatically store their data in JULEA. The HDF5 Virtual Object Layer (VOL) Plugin is implemented as a proof-of-concept to show that the new JULEA Database (JULEA DB) interface can be used to store metadata in one central location. The HDF5 VOL Plugin is also utilized to analyze the impact of the newly introduced file system interface on performance.

The global metadata manager is not part of this thesis and will be developed in the future.

1.2 Goal

Current self-describing data formats such as HDF5 and ADIOS2 store their metadata within each file. Because of the flexibility of these data formats, it is not possible to find specific attributes at fixed offsets in the files. Instead, each file must be opened with the appropriate library to search for the requested attributes. This makes queries on large file collections inefficient.

The HDF5 VOL Plugin previously used in JULEA uses the JULEA Key Value Store (JULEA KV) backend to store its metadata. The advantage of the JULEA KV backend compared to native HDF5 is that the JULEA KV backend can identify identical metadata. The problem with this approach is that it is still not possible to efficiently query all files that contain a specific attribute set.

The goal of this thesis is to store the metadata in a dedicated metadata backend. Saving the metadata of all files, along with file system metadata, allows complex SQL-like queries against the metadata. Using complex metadata queries makes it easy to find files of interest in large file collections without the need to create individual indices next to the file system.

1.3 Thesis outline

The remainder of this thesis is structured as follows: Chapter 2 describes how currently used data formats store their metadata. Chapter 3 shows how the new proposed storage interface works, and which problems are taken into consideration. Chapter 4 shows how the proposed design is implemented. Chapter 5 compares different aspects of the implementation with similar approaches. Chapter 6 shows related work, which tries to solve similar problem definitions. Additionally, differences and problems with the shown approaches are highlighted.

2 State of the art

First, this chapter defines different kinds of file systems. Especially the metadata structure is highlighted, because this is a main component of the following chapters.

2.1 File systems

File systems are present since the early days of computers. The main task of those systems is to transform a file structure into a byte array, which then can be stored on a physical storage medium. To fulfill that task, file systems are forced to store metadata information. This metadata primarily contains the name of the file, and the location of the bytes, which belong to this file. Nowadays all major file systems are hierarchical file systems. The hierarchical component allows the definition of a directory structure. These directories are special files, which contain the locations of other files metadata instead of file data.

Most existing file systems implement the POSIX standard. This interface defines the functions, which access and modify the file structure and data. Because this interface is old, multiple aspects, which are important nowadays, are not taken into consideration in its design. Some important aspects which were not considered in the standard are:

- huge directories with thousands or millions of files
- huge files which contain terabytes of data
- copying of huge files which contain gaps, which are never initialized, between different file systems
- network synchronization overhead
- the ability to attach searchable metadata to files
- the requirement that each file access needs to be written as a time stamp to the file metadata

Some of these aspects are only relevant for server-cluster use cases, while other aspects are serious performance killers for any use case. There are multiple attempts to solve some of these problems, but as long as the interface with its definition is not changed, these are just workarounds, and not solutions.

2.1.1 Local file systems

Local file systems store their data on a physical storage medium, which is directly connected to the main board of the computer. The direct physical connection within this computer allows the file system to cache often required file system metadata in its own memory. The file system can assume, that the cached metadata is always valid, since there is no other computer which might modify anything.

2.1.2 Distributed file systems

In contrast to local file systems, the physical storage is reachable over a network connection. This allows the physical storage to be connected to multiple different storage nodes. At the same time multiple client nodes can connect to the same physical storage.

One advantage is, that the overall storage volume and speed can be increased, because multiple storage nodes share the task to store data. At the same time a huge collection of clients can read and modify the same data at the same time. This creates the additional task for the file system to coordinate the concurrent access of the clients.

Coordination of thousands of compute nodes is a complex task. Additionally the POSIX standard defines hard requirements on the behavior of the implementation. A major problem with the POSIX standard for this use case is the requirement, that any change to any file needs to be made visible instantly to every connected user. Additionally the POSIX semantic requires, that for every read access the metadata should be updated to store the timestamp of the occurred read operation. Together this creates a huge amount of network traffic, if a single user reads only a few bytes in a small file. Additionally to the network traffic, this makes caching file system data much more complex, because there might be another user, who just in this moment modified the targeted file or directory.

To reduce the complexity of distributed file systems, local file systems can be used on the storage nodes as an additional layer. For example, Lustre uses the Zettabyte File System (ZFS) locally on its storage nodes, to focus only on the network layer. This reduces the file system complexity, but increases the amount of library and caching layers between the physical storage and the client node.

2.1.3 Typical file system library stack

In super computers environments the major application data often is not written directly to the POSIX storage. Instead these application often use libraries, which provide a more suitable storage interface. Figure 2.1 shows the typical library stack, which is used in server environments.

Application	`struct{int x; float y;}[50]z;`
NetCDF	`int[50] x, float[50] y`
HDF5	`int[50] x, float[50] y`
Distributed File System	`400 bytes`
Local File System	`400 bytes`
Physical Storage	`400 bytes`

Figure 2.1: Typical Application Storage Hierarchy. The left column shows the library layer. The right column shows the known data type.

In the first layer, the application itself does exactly know, the structure and the meaning of all of its data. The next layer is the storage library, which is used by the application.

In scientific applications, often NetCDF is used here. Because this library supports a fully structured data interface, it is possible to exactly define the whole data structure in form of metadata. NetCDF itself requires HDF5.

HDF5 allows the storage of rich metadata too. Due to the different interfaces, the metadata needs to be transformed between these two libraries. This is true, even if both of these libraries can store the exact data structure, just because the metadata is stored in a different way.

HDF5 then writes the data to a POSIX file system. To do this, HDF5 needs to convert the structural metadata into a binary array. After the conversion into a binary array, all structural information is lost. Any of the following libraries of file systems may only know, that there is a specific amount of bytes, which needs to be stored.

Every library layer potentially looses some structural information. If the structural information is lost, it can not be restored in a lower library layer. Without the structural information, the file system is not able to perform queries on parts of the binary data.

2.2 HDF5

HDF5 [Group, 2019] is a hierarchical data format. The HDF5-Group defines the binary encoding of the data and provides libraries that can read and write files of that format. HDF5 files store large amounts of structured data along with their metadata.

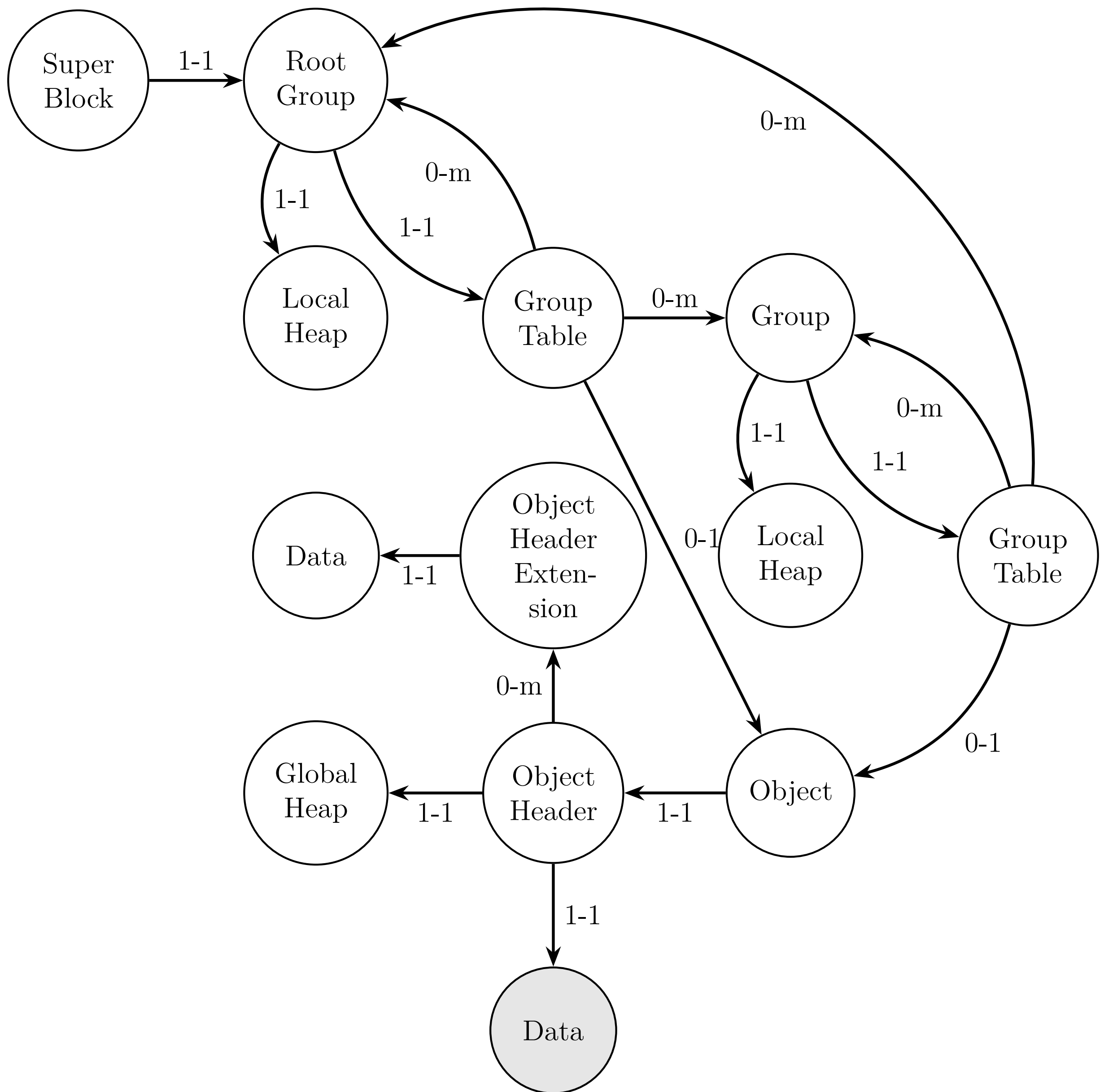


Figure 2.2: The HDF5 File Structure [Group, 2006]. The gray node shows where the actual data is stored. Everything else is metadata. The arrow captions highlight, which components are either optional or can be referenced multiple times from a single component.

Each HDF5 file is structured as shown in Figure 2.2. This file format allows a lot of different aspects of the data to be modeled in the metadata. Some components in a file are optional, other can exist multiple times. As a consequence, the HDF5 library defines a huge amount of functions in its application interface. The elements which can hold the actual data are called `dataset` or `attribute`. The `dataset` can store huge amounts of data. On the other hand, the `attribute` stores the application defined metadata, which can be attached to any `dataset`, `group` or `file`.

All element types contained in HDF5, such as `dataset` or `attribute`, are special cases of the generic type `group`. Therefore, it is sufficient to allow cyclic dependencies between `groups` to allow similar relationships between `dataset` as well. These cycles allow the definition and storage of complex dependency graphs within a single HDF5 file.

The advantage of storing metadata in each file is, that the flexibility of those files is maximized, because any file can have a different format. In real world applications like simulation applications, this feature is not used because the data format is defined once. Afterwards a huge amount of files with the same file format is going to be written to storage during the runtime of the application.

To gain any scientific results, these generated files need to be analyzed later. For a quick overview, to identify if something bad happened, it is often useful to find extreme values across all of the generated files. The current approach, to store the metadata in each file requires, that each file is individually opened, and the extreme values need to be extracted. If the metadata would have been stored in the file system, it would be much faster to perform this kind of analysis.

In contrast there are aspects, where metadata in each file is an advantage. For example, if the metadata is contained in each file, then the file can be easily send to another user, and that other user can use the metadata to learn, what the file contains. Structured HDF5 object headers contain information about the exact encoding of the binary file. This allows moving the file to another supercomputer with a different CPU architecture.

The size of the metadata itself is usually small and of little importance compared to the size of the real data.

Finally, the storage location of the metadata is a choice between portability and searchability.

HDF5 does not currently create its own metadata, such as minimum and maximum values. Instead, the application must explicitly define all metadata.

HDF5 allows Message Passing Interface (MPI) applications to access the same file. However, there is a performance issue if multiple MPI processes update the file metadata. The reason for this performance issue is, that all MPI processes which open a file must perform all operations which change the file metadata collectively. This includes the creation of a dataset as well as writing to an attribute. Even those MPI processes, which do not have to write data to those datasets, need to participate in the collectively called `H5Dcreate` function [Chaarawi, 2015].

2.2.1 Usage example

This section shows an example of how HDF5 can be used. Additionally the improvements of using HDF5 compared to direct IO are highlighted.

```
int write_hdf_data(int *data, hsize_t dims){
        hid_t file;
        hid_t dataspace;
        hid_t dataset;

        file = H5Fcreate("JULEA.h5", H5F_ACC_TRUNC, H5P_DEFAULT,
         ↪  H5P_DEFAULT);

        dataspace = H5Screate_simple(1, &dims, NULL);

        dataset = H5Dcreate2(file, "test", H5T_NATIVE_INT, dataspace,
         ↪  H5P_DEFAULT, H5P_DEFAULT, H5P_DEFAULT);
        H5Dwrite(dataset, H5T_NATIVE_INT, H5S_ALL, H5S_ALL, H5P_DEFAULT,
         ↪  data);
        H5Dclose(dataset);

        H5Sclose(dataspace);

        H5Fclose(file);
}
```

Listing 2.1: HDF5 usage example

First Listing 2.1 assumes in line 1 that there is an array of integer values, which should be written to the storage. Additionally it is known, how many integer values are going to be written.

The first HDF5 call in line 6 initializes a new HDF5 file. The truncate flag is used to define, that the file should be truncated if it exists, otherwise this file is newly created. The other parameters during file creation can be used to pass some global options to the file. For example, these parameters can be used to specify that the JULEA HDF5 VOL Plugin should be used instead of the internal binary encoding.

Afterwards a `dataspace` is defined. Within HDF5 this kind of variable is used to define the size and dimension count of an array. In this example the array has only a single dimension, which reflects the amount of data points to be written to storage.

Next, the `dataset` itself is created within the specified file. During the `dataset` creation, the variable type and the number of elements need to be defined. The variable type of a specific `dataset` can not be changed for the lifetime of an HDF5 file. In this example, the array is of type integer. It would be possible to declare the variable type to be any complex datatype.

Finally, after the definition is ready, the data is written to the `dataset` in line 11. Here the whole data is written at once. Instead of `H5S_ALL` the user could define any subset of fields within the array. This allows the partial writing of data.

Afterwards every created handle needs to be closed, to flush the data to disk and release all in-memory pointers.

If the file is opened later by another user, the metadata could tell, that there is an array of integer defined under the key `test`. The user could use that information, to read and use the data in its own application.

In contrast, if only a plain POSIX file is used, than there is no structural information within the file. The file-system could tell the user only, that there is an specific amount of bytes. Without the file structure information, the file is just binary data, which can not be interpreted on its own. To use the data, it is required to ask the file creator, how the file needs to be parsed.

In summary, HDF5 provides a way, that the file data is described by the file itself. This costs some time during data creation, but makes it much easier to read the data later.

2.3 ADIOS2

ADIOS2 is a unified IO performance framework [Laboratory, 2019]. The framework can be used to transport self-discribing data over the network. Additionally this framework can store data in a file system or in-memory. The target is to do the provided operations with a high-performance. In contrast to the performance issue of HDF5 [Chaarawi, 2015], ADIOS2 allows high-performance MPI access to the same data stream. This is possible because ADIOS2 may use a different metadata file for each process.

Time steps are also a central element of ADIOS2. These time steps allow different ADIOS2 applications to stream data through files. Even if one application is still writing new time steps, another application could start to process those files at the same time. The time step model fits perfectly to applications, which simulate something, and write their data in static time intervals. On the other hand, if the application does not write time-dependent data, this approach has its disadvantages. ADIOS2 itself does not provide an Application Programming Interface (API) to define hierarchical data formats. If hierarchical dependencies are desired, these must be added by another library on top of ADIOS2. Another difference to HDF5 is, that ADIOS2 calculates its own metadata along with the data provided by the application. The calculated properties include minimum and maximum values for the different variables.

ADIOS2 provides multiple engines. Each engine is specialized in its own task. Some engines store the data in the file system, others transmit data over the network. The following paragraph describes the binary-packed (BP4) engine, which is the most current file storage engine. This engine uses a folder structure with multiple different files, to store the data which belongs together. The variable number of files within the directory structure allows multiple processes to write at the same time into different files, without the risk of damaging data of other processes. The same applies to the metadata. The

metadata is stored within the same directory structure, but in different files than the data. This allows for more flexible handling of the metadata. Even if the data is stored in multiple files, all of these files belong together. In addition to advantages during writing, this approach allows a reader application, to process the data at the same time, as it is generated.

2.4 JULEA

JULEA [Kuhn, 2017] is designed to simplify the IO stack of High-Performance Computing (HPC) applications and quickly evaluate new storage concepts. To reach this goal JULEA defines a collection of simple interfaces. To enable rapid prototyping, JULEA has a modular structure. For example, a JULEA module provides a Filesystem in Userspace (Fuse) implementation, which supports the user with a fully POSIX compatible file system. In addition to the plain POSIX, JULEA defines several different alternative storage interfaces.

Currently, the code base of JULEA is small. Because of this property, JULEA is a beginner-friendly storage environment. The core of JULEA includes a client-server socket connection to enable distributed storage systems.

The basic structure of an application using JULEA is shown in Figure 2.3.

The JULEA KV interface defines generic functions for storing relationships between keys and small objects. Data for a specific key in JULEA KV must be completely written immediately.

The JULEA Object Store (JULEA OBJ) interface defines generic functions for storing relationships between keys and huge objects. To write objects that are larger than the available Random-Access Memory (RAM), this interface allows partial read and write access to objects. The JULEA KV and JULEA OBJ components can be used together to implement all functions required for a POSIX file system.

The new JULEA DB interface extends or replaces the JULEA KV interface, allowing for efficient querying of partial keys or partial data content.

There are multiple benefits if, for example, HDF5 objects are mapped to JULEA objects, key-value pairs and db-entries:

- There are different backend interfaces, which can be used to implement plugins for different higher storage abstraction layers.
- Filesystem metadata and file metadata is not required to be separated from each other.
- Metadata may be stored in a more structured backend, which allows more complex and faster analysis queries, even across across multiple files. With these fast metadata queries, the searching for data pieces of interest in huge file collections is supported.

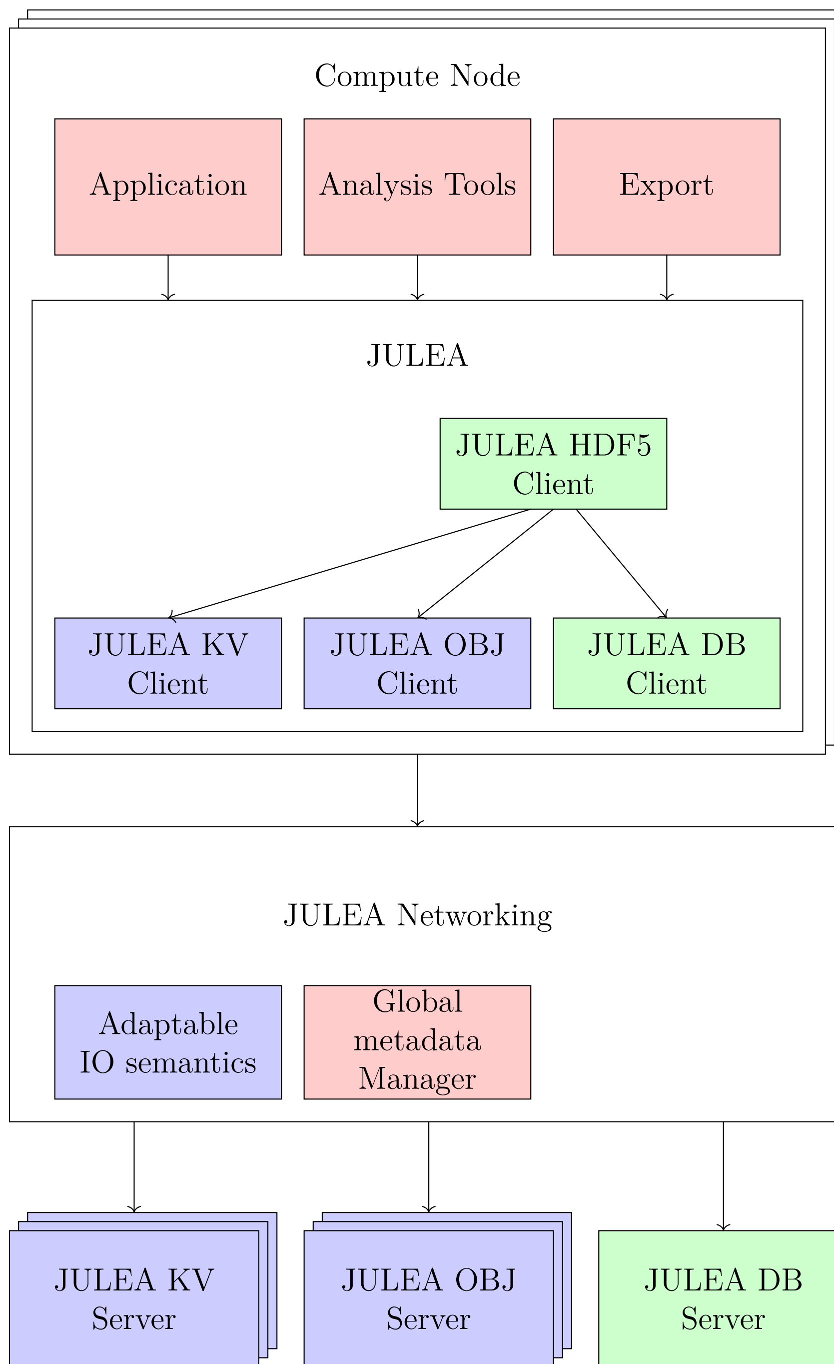


Figure 2.3: JULEA applications Structure. The blue parts in the figure existed before this thesis. The green components are added in this thesis. The red boxes are planned, but not included in this thesis.

- If the file system has structure information about the file content, then often used file pieces can be stored on a fast storage medium, while archive data can be compressed and stored on slow storage devices.
- The file system may use different storage backends for different kinds of files. Some backends allow faster read access, others allow faster updating of data. Each backend has its own advantages, and each user has its own reqirements. Rich metadata within a file system allows more efficient storage access.
- Deduplication of similar data can be performed more easily, because the file system can now focus on similar file sections instead of just raw data files.
- Additionally, this allows the file system to store similar data sections of different files more efficiently, to allow faster read access later. Faster read access may be provided due to the distribution of data to the backends. A larger number of storage nodes may increase the read access to the data later.

Adding new backend types to JULEA is easy, but not the main goal of the JULEA design. Prior to this thesis, JULEA only defined two backend types. Backend types define internal interfaces used by the JULEA client library.

If the required backend type is defined then it is very easy to implement a new backend. Each backend type interface tries to be as generic as possible to accommodate a wide variety of different implementations. All existing backend types have at least two different implementations, but the number of available backends is not limited by the source code itself. Currently, each instance of JULEA can load at most one backend implementation of each type. This may change in the future from JULEA, if automated algorithms based on application requirements decide which backend leads to the best performance.

Because JULEA is a distributed storage system, backend implementations should always keep in mind that the code can be accessed by multiple threads simultaneously.

With JULEA a different semantics can be specified for each message. Semantics can change the behavior of the JULEA backend if multiple processes write their data at the same time. Currently, the semantic part of JULEA is in an early stage of development and therefore not fully supported by all backends.

3 Design

This chapter describes the design decisions before and during the development of the JULEA DB backend.

3.1 Naive approach

There are several approaches, how applications can interact with a structured backend type. The simplest approach is, that each application uses its own SQL library directly. Then each application can connect to the same database. With this approach, only one database schema would have to be defined. Afterwards, each application can write its data independently. That way, the application has full control over its metadata. The advantage is that every application has the maximum flexibility to use a particular backend. At the same time, this advantage is also a disadvantage. If the applications are not forced to a common schema, some applications do not adhere to a proposed scheme if it does not fit perfectly with their needs. If each application uses its own schema, the benefits are reduced because the results cannot be easily compared between applications. Because of the differences between the various SQL libraries, the application cannot easily change its database without requiring full refactoring.

3.2 JULEA DB backend

The core idea of the JULEA DB backend interface definition is to support many different backend implementations. To keep JULEA as simple as possible, the number of functions in the interface is kept as low as possible.

```
gboolean backend_init (gchar const*);
void backend_fini (void);

gboolean backend_batch_start (gchar const*, JSemantics*, gpointer*,
    GError**);
gboolean backend_batch_execute (gpointer, GError**);

gboolean backend_schema_create (gpointer, gchar const*, bson_t const*,
    GError**);
gboolean backend_schema_get (gpointer, gchar const*, bson_t*, GError**);
gboolean backend_schema_delete (gpointer, gchar const*, GError**);

gboolean backend_insert (gpointer, gchar const*, bson_t const*, bson_t*,
    GError**);
gboolean backend_update (gpointer, gchar const*, bson_t const*, bson_t
    const*, GError**);
gboolean backend_delete (gpointer, gchar const*, bson_t const*,
    GError**);
gboolean backend_query (gpointer, gchar const*, bson_t const*,
    gpointer*, GError**);
gboolean backend_iterate (gpointer, bson_t*, GError**);
```

Listing 3.1: JULEA backend interface

Listing 3.1 shows the API of the JULEA DB backend.

The interface includes functions to define and use a structured datatype. After the definition of the structured datatype, the interface can be used to access the data. The data access can be any of the basic data operations like `insert, update, delete and query`.

A lower abstraction layer like `write, read and truncate` would not be useful at this place, because the intended improvement is to allow structured data within the file system. A higher abstraction layer on the other hand, would increase the complexity of the backend. At the same time complex functions may not be suitable to map every desired datatype to the backend, because some libraries may have completely different assumptions.

Each backend type in JULEA provides functions to initialize and finalize any backend, which must be called at application startup and termination respectively. These functions

follow a different error handling strategy then every other function in the API. The detailed error handling is described in Section 4.5.4.

The definition of this backend type implies but does not require SQL implementations.

The proposed internal interface uses Binary JSON (BSON) [bson, 2019] as a container to send data over the network. BSON is part of the public MongoDB [MongoDB, 2019] interface. MongoDB is a non relational SQL (noSQL) database that is already used as a JULEA KV backend.

The BSON structure is similar to JSON but is binary coded. Binary coding reduces the size of data for transmission over the network compared to simple JSON. In addition, the use of BSON prevents the use of many string conversion functions. BSON already has several language bindings, which makes the development of JULEA language bindings even easier.

BSON has already been used at various points in JULEA for transfer purposes so that the JULEA DB interface can share the existing core code. The advantage of BSON is those type conversions are applied automatically when the communicating computers use different CPU architectures, such as big and little-endian. This allows easy portable code between different CPU architectures. The disadvantage of BSON is that the data has to be converted into a key-value format. Then, the receiver side has to extract the data from this key-value format. Unfortunately, benchmarks show that especially the creation of large BSON document structures takes much time Table 5.1.

Figure 3.1 shows the overhead generated by the JULEA DB design while writing metadata. The metadata must be transformed multiple times on the way from the application to the POSIX store.

To prevent problems with incompatible CPU architectures, the data is first packaged into a BSON structure. Depending on whether the client or server-side backend is loaded, the BSON is either sent as a byte stream over a GLib socket connection or passed directly to the JULEA backend API. Within the backend, the BSON needs to be parsed for the SQL API to use the data. The SQL backend then transforms the data into its own internal format and integrates the data into the corresponding tables. Finally, a confirmation signal is sent back to the application. If an error occurs it will be returned to the application instead of the acknowledgment signal, and all further processing of that message will be aborted.

Transactions are part of the JULEA DB interface as shown in Listing 3.1. To begin a transaction the function `backend_batch_start` is used. All functions which are called within a transaction are restricted to the same namespace, which is passed as the first parameter. The message semantics are also the same for every function within a transaction. The third parameter outputs an opaque pointer, which is afterwards passed to every other backend function as the first parameter. This ensures a consistent behavior within a transaction.

To finally commit a transaction, `backend_batch_execute` must be called. This function must be called in case of any error within the transaction too because this function is responsible to free any transaction-specific resources.

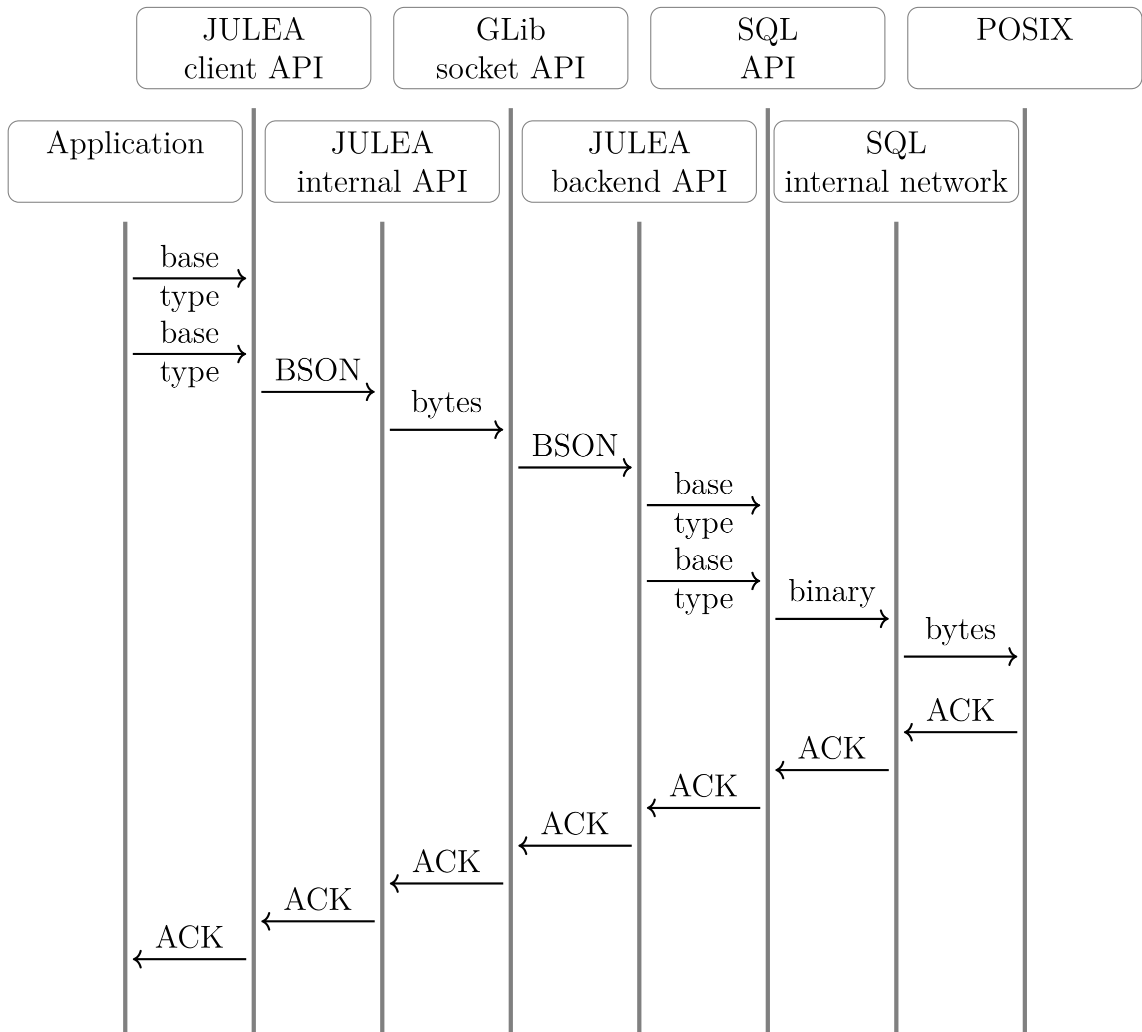


Figure 3.1: JULEA metadata Write

The existence of transactions has performance reasons too. Without transactions, a huge amount of small messages would generate a huge amount of network traffic. If all these small transactions are combined in a single transaction, then the amount of network packets is reduced. Databases are typically faster if multiple operations are combined in a single transaction. Future versions of JULEA might be able to sort and deduplicate the requests, to gain additional speed. The scope of the transactions can be defined by the message semantics. The application can determine whether each operation should be performed in a separate transaction, or whether all operations within a message should be performed together in a transaction. Later it will be possible to explicitly start and end transactions containing several messages. Transactions that span multiple messages can be useful to ensure consistent file system status, especially if the application may crash.

Not all backend implementations can handle all combinations of message semantics. Some backend libraries used, may have limitations that JULEA cannot simulate.

For example, the Data Definition Language (DDL) contains all the commands in a database system that are used to create, modify or delete database schema structures. Normally, DDL operations are not performed while an SQL transaction is not completed. For example, MySQL closes all open transactions before running a DDL operation. In contrast, SQLite fully supports DDL operations within transactions.

However, JULEA DB schema operations to create or delete a schema should very rarely be used. Applications should define their data layout once at the beginning of the program and then use it for their runtime.

To avoid inconsistent behavior, JULEA currently explicitly closes any open transactions before DDL operations are applied. As a result, JULEA does not execute all `backend_schema_create` operations within a single database transaction, even if the message semantics so specifies. The same applies to `backend_schema_delete` because both create and delete SQL schema in their backend implementations.

In addition to the special cases mentioned, the JULEA backend implementation respects any semantics as much as possible. In the future, JULEA will decide whether to issue errors or notify the application that the requested semantics is not applicable.

The JULEA DB backend interface should only be called internally in JULEA and never directly from the application. In this way, JULEA can perform extended error checking on the client-side. Currently, most errors are also explicitly caught in the backend, but some errors would only be detected by SQL compilation errors. Future versions of JULEA may eliminate this redundant error checking to improve performance. In the future, the client-side library will perform additional optimizations within the application to improve the performance and usability of the data.

3.3 JULEA DB client

The definition of the client-side functions follows the same approach as the definition of the backend-side interface. The client-side JULEA library should allow the application to use the JULEA library in very different scenarios. As a consequence, the client-side JULEA library was developed to support the implementation of various other libraries based on JULEA. For example, JULEA includes an HDF5 VOL Plugin based on its own client-side library. To make the memory management easier, each allocated object uses reference counting.

3.3.1 JDBSchema

The client-side JDBSchema object as shown in Listing 3.2 contains the structure definition of a data type which is specified by the application. The corresponding concept in SQL is the schema of a table.

```
JDBSchema* j_db_schema_new (gchar const* namespace, gchar const* name,
  ↪ GError** error);
JDBSchema* j_db_schema_ref (JDBSchema* schema);
void j_db_schema_unref (JDBSchema* schema);

gboolean j_db_schema_add_field (JDBSchema* schema, gchar const* name,
  ↪ JDBType type, GError** error);
gboolean j_db_schema_get_field (JDBSchema* schema, gchar const* name,
  ↪ JDBType* type, GError** error);
guint32 j_db_schema_get_all_fields (JDBSchema* schema, gchar*** names,
  ↪ JDBType** types, GError** error);
gboolean j_db_schema_add_index (JDBSchema* schema, gchar const** names,
  ↪ GError** error);

gboolean j_db_schema_equals(JDBSchema* schema1, JDBSchema* schema2,
  ↪ gboolean* equal, GError** error);

gboolean j_db_schema_create (JDBSchema* schema, JBatch* batch, GError**
  ↪ error);
gboolean j_db_schema_get (JDBSchema* schema, JBatch* batch, GError**
  ↪ error);
gboolean j_db_schema_delete (JDBSchema* schema, JBatch* batch, GError**
  ↪ error);
```

Listing 3.2: JULEA DB client-side JDBSchema

This JDBSchema object contains the information about which variables exist and which type should be used to access those variables. JULEA DB itself does not allow implicit

type conversions. Applications should use the same type of data to read and write variables anyway.

During the schema build phase, the application may decide to add an arbitrary amount of fields to the schema using `j_db_schema_add_field`. After the definition of a field, the application may index some of the variables by calling `j_db_schema_add_index` to improve the access speed later. Indexes initially cost additional time when the affected variables are written. However, indexes can also achieve a tremendous speed increase as shown later in Figure 5.6 if the variables are frequently read or queried. Therefore, the application should pay attention to which variables should be indexed and which should not.

After the schema is created via `j_db_schema_create`, the application can query an existing `JDBSchema` definition at any time. To query `JDBSchema` information first a new schema handle needs to be created by supplying the namespace and name to the function `j_db_schema_new`. Afterwards, `j_db_schema_get` retrieved the desired information. The queried `JDBSchema` information can later be used to check if the backend uses a compatible definition. In addition to checking the structure, the requested `JDBSchema` information may include additional automatically generated variables, including their type. The most important automatically generated variable is the `_id`, which can be used as a SQL foreign key in various related `JDBSchema` definitions. To enable more flexible backend implementations, each backend implementation can convert each variable type to another compatible type. Because of this behavior, it may be useful to query the structural information immediately after the `JDBSchema` is created.

3.3.2 JDBSelector

Depending on the purpose, the same `JDBSelector` can be used to update, query, and delete some `JDBEntry` objects. The `JDBSelector` only works for data already stored in the backend.

```
1 JDBSelector* j_db_selector_new (JDBSchema* schema, JDBSelectorMode mode,
   ↪ GError** error);
2 JDBSelector* j_db_selector_ref (JDBSelector* selector);
3 void j_db_selector_unref (JDBSelector* selector);
4
5 gboolean j_db_selector_add_field (JDBSelector* selector, gchar const*
   ↪ name, JDBSelectorOperator operator, gconstpointer value, guint64
   ↪ length, GError** error);
6 gboolean j_db_selector_add_selector (JDBSelector* selector, JDBSelector*
   ↪ sub_selector, GError** error);
```

Listing 3.3: JULEA DB client-side `JDBSelector`

Listing 3.3 shows the `JDBSelector` object type, which is used to restrict a query to the specified conditions.

During the creation, it is required, to specify a JDBSchema, on which the selection is performed. The JDBSchema is used to verify, that all requested fields exist. Additionally type checking can be performed. To allow querying multiple fields at once, the mode parameter allows choosing how all fields are connected with each other. Currently, mode can be either AND or OR. Listing 3.4 shows an example of how complex combinations of conditions can be applied to a JDBSelector.

j_db_selector_add_field can be used to define an arbitrary amount of fields together with their conditions.

```
sel1 = j_db_selector_new(..OR..);
sel2 = j_db_selector_new(..AND..);
j_db_selector_add_field(sel1, ..cond1..);
j_db_selector_add_field(sel1, ..cond2..);
j_db_selector_add_field(sel2, ..cond1..);
j_db_selector_add_field(sel2, ..cond2..);
j_db_selector_add_selector(sel1, sel2, ..);
// .. WHERE sel1cond1 OR sel1cond2 OR (sel2cond1 AND sel2cond2)
```

Listing 3.4: JULEA DB client-side complex JDBSelector query

3.3.3 JDBEntry

The client-side JDBEntry object contains the variable values for a subset of the variables defined in the corresponding JDBSchema.

```
JDBEntry* j_db_entry_new (JDBSchema* schema, GError** error);
JDBEntry* j_db_entry_ref (JDBEntry* entry);
void j_db_entry_unref (JDBEntry* entry);

gboolean j_db_entry_set_field (JDBEntry* entry, gchar const* name,
    gconstpointer value, guint64 length, GError** error);

gboolean j_db_entry_insert (JDBEntry* entry, JBatch* batch, GError**
    error);
gboolean j_db_entry_update (JDBEntry* entry, JDBSelector* selector,
    JBatch* batch, GError** error);
gboolean j_db_entry_delete (JDBEntry* entry, JDBSelector* selector,
    JBatch* batch, GError** error);

gboolean j_db_entry_get_id (JDBEntry* entry, gpointer* value, guint64*
    length, GError** error);
```

Listing 3.5: JULEA DB client-side JDBEntry

Listing 3.5 shows the interface for the JDBEntry object. During the creation, the j_db_entry_new requires the JDBSchema reference. This reference is required, to be able to automatically infer the types of the values, which are set afterwards using j_db_entry_set_field.

j_db_entry_set_field can be used to define the values for a subset of the fields defined within the schema definition. Depending on the use-case, the remaining fields are automatically filled. If the use-case is j_db_entry_insert, then the remaining values are set to NULL. When an existing entry should be updated using j_db_entry_update, then the remaining fields are not touched. In the last use-case, where an entry should be deleted, the fields do not matter anyway.

The function j_db_entry_get_id may only be called directly after a successful insert operation. In this case, the function retrieves a unique identifier, which can be used to find exactly this entry later in the backend. Entries have no methods for querying their values. Their only purpose is to add structured data to the backend.

3.3.4 JDBIterator

The last component of the client-side API is the JDBIterator. This component allows the iteration over multiple JDBEntry objects. In order to reduce the number of network requests and obtain consistent data, all selected JDBEntry objects are simultaneously placed in a buffer within the JULEA DB client to allow for fast iteration thereafter. Due to the bulk transfer of data between the backend and the client, the client and backend JDBIterator definitions are completely independent of each other.

```
1 JDBIterator* j_db_iterator_new (JDBSchema* schema, JDBSelector*
  ↪  selector, GError** error);
2 JDBIterator* j_db_iterator_ref (JDBIterator* iterator);
3 void j_db_iterator_unref (JDBIterator* iterator);
4
5 gboolean j_db_iterator_next (JDBIterator* iterator, GError** error);
6
7 gboolean j_db_iterator_get_field (JDBIterator* iterator, gchar const*
  ↪  name, JDBType* type, gpointer* value, guint64* length, GError**
  ↪  error);
```

Listing 3.6: JULEA DB client-side JDBIterator

Besides the memory management functions, only two functions are defined.

j_db_iterator_next can be used to loop over the retrieved entries. If there are no more elements then a no-more-elements-error is thrown. If the completion of the loop is indicated as an error, then the function implementation may also return additional errors.

To retrieve the values of the fields for a current entry, the function `j_db_iterator_-get_field` is used. Besides the binary data, this function returns the field type too. The application may use this type of information to verify, that the requested and retrieved data types match.

3.4 HDF5 VOL Plugin on top of JULEA DB client

The HDF5 VOL Plugin interface can override HDF5 functions directly related to storage IO. The HDF5 VOL Plugin interface allows for alternative encoding as well as file structures optimized for different file systems. In addition, this interface enables server admins to change the storage format globally without the need to change an application. There are already implementations [Olgasnezh, 2017] [Perevalova, 2017] that store all HDF5 metadata in a SQLite database. JULEA already offers an implementation of the HDF5 VOL Plugin. The current implementation of the HDF5 VOL Plugin is based on the JULEA KV and JULEA OBJ clients. The newly proposed implementation of the HDF5 VOL Plugin uses the JULEA DB client instead of the JULEA KV client. The analysis of the stored HDF5 files is thereby to be improved because now complex SQL queries can be performed across multiple files. Since some of the existing JULEA KV backends are also based on SQL databases, the performance difference should be minimal.

Since HDF5 already specifies all public functions, the design of the HDF5 VOL Plugin can define a fixed SQL schema. The JULEA DB schema definition is the main difference to the existing implementation of JULEA KV, as this new interface allows much more detailed metadata queries. The complete ER-Diagram of the SQL schema is shown in Figure 3.2. To allow a simplified representation of the SQL table structure, each SQL table is additionally defined in its own table.

3.4.1 HDF5-File

Each HDF5 file has exactly one entry in the `HDF5_file` schema. This `HDF5_file` entry is the entry point for all data and metadata belonging to the same file. The schema of a file defines only the file name as seen in Table 3.1.

column name	type	references	comment
`_id`	Integer		auto-generated by the database
`name`	String		the name of the file

Table 3.1: `HDF5_file` defined on behalf of the HDF5 VOL Plugin plugin

To quickly identify which data and metadata belong to an `HDF5_file`, every other SQL schema has a direct reference to that `HDF5_file`. This relation is needed to efficiently truncate a file. HDF5 itself does not define functions for deleting an `HDF5_attribute` nor functions for deleting the entire file. Native HDF5 also does not require a file deletion operation because the file is accurately represented in a POSIX file system. To empty

the entire file, the HDF5 VOL Plugin can be used to open the file in truncate mode. To remove only a single `HDF5_attribute`, the entire file must be repackaged, skipping the `HDF5_attribute` to be removed. HDF5 does not define such a copy function on its own. Instead, an external application must be used to perform this operation.

After the `HDF5_file` entry has been created, any other HDF5 object can be added to this file. Each object added to a file has an entry in the corresponding SQL schema.

3.4.2 HDF5-Link

In addition, each object needs at least one entry in the `HDF5_link` schema. The `HDF5_link` schema is used to define any cyclic dependencies between any HDF5 objects. The table definition can be seen in Table 3.2. Cyclic dependencies are allowed between any kind of object except files. Files are currently not allowed to belong to another file. To detect which object types are referenced, extra columns indicate the types of the referenced objects.

column name	type	references	comment
`_id`	Integer		auto-generated by the database
`file`	Integer	file→`_id`	
`parent_type`	Integer		indicating the target table of the `parent` attribute
`child_type`	Integer		indicating the target table of the `child` attribute
`parent`	Integer	{group, dataset, attr}→`_id`	
`child`	Integer	{group, dataset, attr, file}→`_id`	

Table 3.2: `HDF5_link` defined on behalf of the HDF5 VOL Plugin plugin

3.4.3 HDF5-Datatype

The `HDF5_datatype_header` entry defines the type of a single value. HDF5 allows very detailed and complex definitions of data types. Currently, only basic variable types are allowed. Later the HDF5 VOL Plugin implementation might be extended to enable the more complex data types features. Even if only very basic variables are considered, HDF5 allows a wide variety of options. Table 3.3 shows the definition of the basic datatype within JULEA DB.

column name	type	references	comment
_id	Integer		auto-generated by the database
type_cache	BLOB		caching the HDF5 binary type
type_class	Integer		the type of the variable
type_size	Integer		the amount of bytes per variable
type_sign	Integer		only for integer types
type_···	Integer		other type properties as defined by HDF5

Table 3.3: HDF5_datatype defined on behalf of the HDF5 VOL Plugin plugin

3.4.4 HDF5-Space

The HDF5_space_header entry, along with the associated HDF5_space entries, defines how many values are contained in the related HDF5_attribute or HDF5_dataset. The HDF5_space entries may define unlimited dimensions that allow any amount of data to be written to the file. Because a space consists of a variable amount of dimensions, multiple tables are required. At first, Table 3.5 defines, how many dimensions exist.

column name	type	references	comment
_id	Integer		auto-generated by the database
dim_index	Integer		the index of the dimension
dim_size	Integer		the allowed range within the dimension
dim	Integer	space_header→_id	

Table 3.4: HDF5_space defined on behalf of the HDF5 VOL Plugin plugin

Afterwards, Table 3.4 defines the ranges of each each dimension.

column name	type	references	comment
_id	Integer		auto-generated by the database
dim_cache	BLOB		caching the HDF5 binary space
dim_count	Integer		how many dimensions exist
dim_total_count	Integer		multiplicative sum of all dimensions ranges

Table 3.5: HDF5_space_header defined on behalf of the HDF5 VOL Plugin plugin

3.4.5 HDF5-Dataset and HDF5-Attribute

Currently, datasets and attributes are handled the same way within the JULEA DB HDF5 VOL Plugin. Therefore only the structure of the dataset is shown in Table 3.6. The data itself is not stored in the JULEA DB backend. Instead, the `_id` value of the corresponding `HDF5_attribute` or `HDF5_dataset` is used as the key in the JULEA OBJ store. Later the values of the attributes are going to be extracted and stored within the Database directly. The dataset data is expected to be large in size, and therefore will remain in the JULEA OBJ.

Both attributes and datasets, have an additional entry within the `HDF5_datatype` schema and the `HDF5_space` schema.

column name	type	references	comment
`_id`	Integer	JULEA OBJ→key	auto-generated by the database
`file`	Integer	file→`_id`	
`dataspace`	Integer	dataspace_header→`_id`	
`datatype`	Integer	datatype_header→`_id`	

Table 3.6: `HDF5_dataset` defined on behalf of the HDF5 VOL Plugin plugin

Currently, each `HDF5_dataset` and `HDF5_attribute` creates its own file in the JULEA OBJ store. As long as there are only a few thousand `HDF5_dataset`, this works, but every average HDF5 file defines multiple `HDF5_datasets` and `HDF5_attributes`. If the JULEA OBJ store is filled with a large number of keys, then all access times to the JULEA OBJ backend may increase. This behaviour was observed while using the POSIX object store backend, without explicitly using directories within the key names.

Within this thesis there is no aspiration to fully support all functions of the HDF5 VOL Plugin, but only a proof-of-concept benchmark comparison should be possible. The HDF5 JULEA library can be used to evaluate the performance and cost differences when storing metadata globally in the file system.

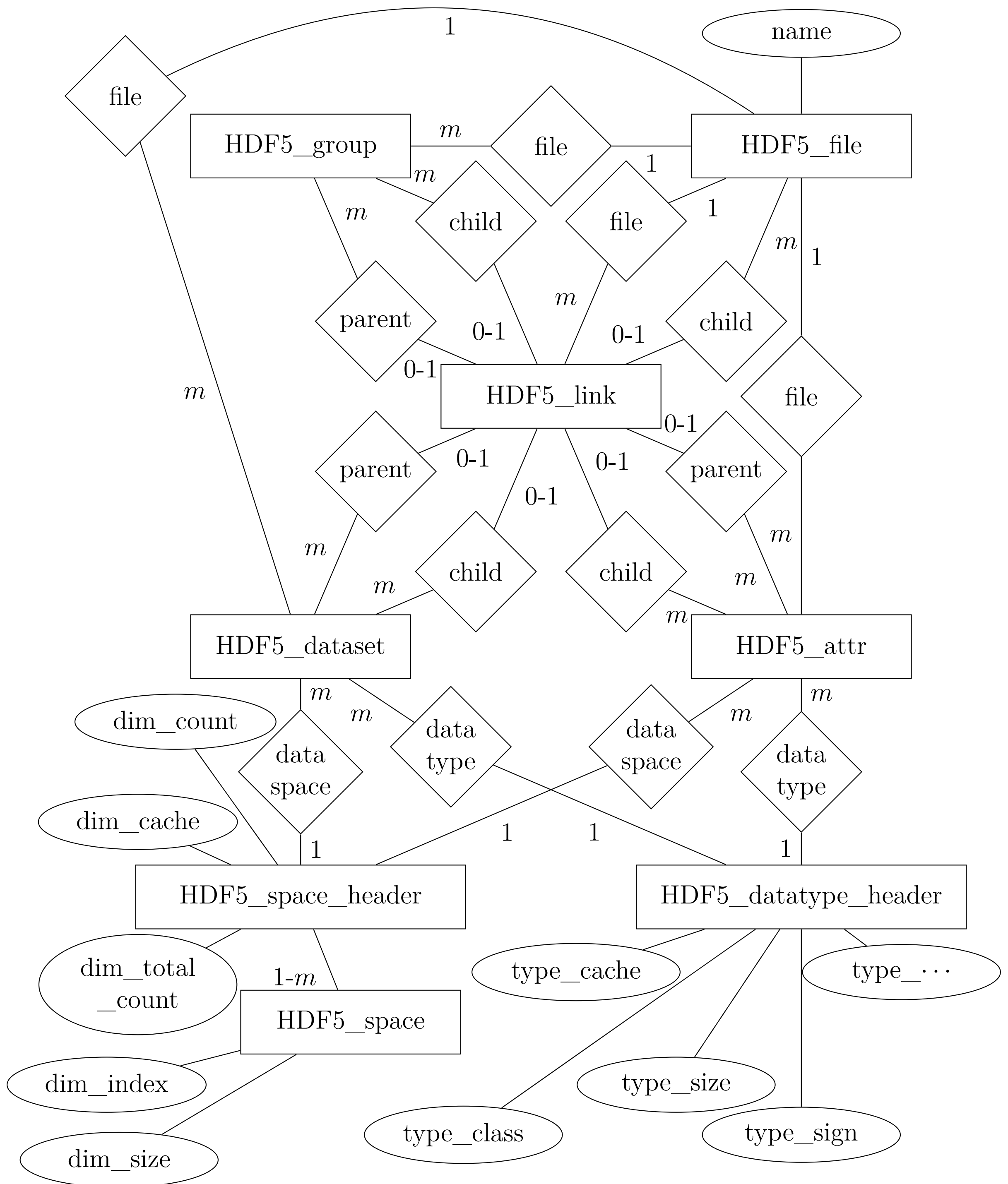


Figure 3.2: HDF5 VOL Plugin ER-Diagram

3.4.6 HDF5 VOL Plugin example

```
1  hid_t julea_vol_id;
2  hid_t fapl;
3  hid_t file;
4
5  // initialize vol-plugin
6
7  julea_vol_id = H5VLregister_connector_by_name("julea", H5P_DEFAULT);
8  H5VLinitialize(julea_vol_id, H5P_DEFAULT);
9
10 // create / open file
11
12 fapl = H5Pcreate(H5P_FILE_ACCESS);
13 H5Pset_vol(fapl, julea_vol_id, NULL);
14 file = H5Fcreate("julea.h5", H5F_ACC_TRUNC, H5P_DEFAULT, fapl);
15
16 // do sth with file
17
18 // close file
19
20 H5Fclose(file);
21 H5Pclose(fapl);
22
23 // finalize vol-plugin
24
25 H5VLterminate(julea_vol_id);
26 H5VLunregister_connector(julea_vol_id);
```

Listing 3.7: HDF5 VOL Plugin example code

Listing 3.7 shows an example of how to use the JULEA HDF5 VOL Plugin. To update existing applications to use the HDF5 VOL Plugin, the HDF5 VOL Plugin must first be loaded at program startup (see lines 7-8). After that, the functions `H5Fcreate` and `H5Fopen` are each invoked with a file access properties list (`fapl`) which activates the HDF5 VOL Plugin interface as shown in line 14. Afterwards, the file can be used as in any HDF5 application (line 16). Since these modifications are small, it is possible to path existing applications to use a Plugin.

4 Implementation

This chapter shows some implementation details. The most important parts of the written code are already merged into the master branch `https://github.com/wr-hamburg/julea`. Everything else, especially the benchmarks, are accessible at `https://github.com/Qualenritter/julea`.

4.1 Backend Operation

JULEA allows backends to be loaded either on the client or the server-side. Every function which might be called on server-side, therefore, needs to be able to transmit its parameters to the corresponding server. Before this thesis, every function needs to explicitly pack all parameters into a network packet, and parse the parameters from the binary stream on its own. During this work, a new wrapper mechanism is introduced. Now every function, which might be called on the server-side, defines a constant struct, which describes the parameters. Listing 4.1 shows the struct definition.

At first the struct `JBackendOperationParam` defines the type and the value of a single function parameter. On the server-side, the binary data input stream remains valid until the end of the function call. Therefore the temporary static storage can be used. A BSON can be statically initialized using a static buffer, without any additional memory allocations. The same applies to the `GError` data type. For every other variable type, and on the client-side, the output data needs to be written into the provided buffers for each variable. In this case the variables `ptr` and `ptr_const` are holding the data. The variable `len` always holds the length of the data behind `ptr`. This length is required, to move the data over the network.

Additionally `backend_func` defines the function, which should be called, with the extracted and parsed parameters. Every other variable within this struct defines the amount, type and value of the input and output parameters. To be able to handle errors on the client-side, the last output parameter is required to be of type `GError`. Currently, the amount of parameters has a fixed upper limit of 20. This limit is required, to gain an absolute maximum struct size, to be able to call `memcpy` on this struct. This `memcpy` includes all required parameters to call a function. Of course one needs to be careful because function pointers must not be copied and used within other program instances.

Together these data stored in these structs can be used within a single wrapper function, which is able to extract typed parameters from a binary stream. This reduces the duplicated code and allows easier testing. The testing is simplified, because now, there is a single location, where a mockup can be used instead of networking code.

```c
struct JBackendOperationParam
{
	JBackendOperationParamType type;
	union
	{	// Only for temporary static storage
		struct
		{
			gboolean bson_initialized;
			bson_t bson;
		};
		struct
		{
			const gchar* error_quark_string;
			GError error;
			GError* error_ptr;
		};
	};
	union
	{
		gconstpointer ptr_const;
		gpointer ptr;
	};
	gint len;
};
struct JBackendOperation
{
	gboolean (*backend_func)(struct JBackend*, gpointer, struct
	 ↪ JBackendOperation*);
	guint in_param_count;
	guint out_param_count;
	JBackendOperationParam in_param[20];
	JBackendOperationParam out_param[20];
};
```

Listing 4.1: JULEA `JBackendOperation` interface to simplify transfer of data over the network

4.2 SQL wrapper

Within the JULEA DB backend implementation, it is expected that there will be many different SQL powered backends. Every database client API provides different function prototypes. Therefore a generic SQL wrapper is used. This allows switching the real used SQL backend easily. The downside of a generic wrapper around SQL is the added overhead.

Most of the functions, required by the backend type interface are implemented within the generic SQL backend. This shared code base allows simplified testing. Especially the generation of SQL code from the input BSON structure is implemented here. To reuse SQL compiled code, prepared statements are heavily used. Unfortunately, the SQL string needs to be repeatedly generated. If the SQL string is found within a hash table then the corresponding prepared statement is selected and used. Otherwise, the SQL string is used to compile a new prepared statement. To allow fully multi-threaded backends, each thread has its own SQL connection, with its own set of prepared statements.

The provided set of SQL wrapper functions is shown in Listing 4.2.

```
1  gpointer j_sql_open(void);
2  gboolean j_sql_close(gpointer db);
3
4  gboolean j_sql_prepare(gpointer db, const char* sql, gpointer* stmt,
   ↪  GArray* types_in, GArray* types_out, GError** error);
5  gboolean j_sql_finalize(gpointer db, gpointer stmt, GError** error);
6
7  gboolean j_sql_bind_null(gpointer db, gpointer stmt, guint idx, GError**
   ↪  error);
8  gboolean j_sql_bind_value(gpointer db, gpointer stmt, guint idx, JDBType
   ↪  type, JDBTypeValue* value, GError** error);
9  gboolean j_sql_column(gpointer db, gpointer stmt, guint idx, JDBType
   ↪  type, JDBTypeValue* value, GError** error);
10
11 gboolean j_sql_step(gpointer db, gpointer stmt, gboolean* found,
   ↪  GError** error);
12 gboolean j_sql_reset(gpointer db, gpointer stmt, GError** error);
13
14 gboolean j_sql_start_transaction(gpointer db, GError** error);
15 gboolean j_sql_commit_transaction(gpointer db, GError** error);
16 gboolean j_sql_abort_transaction(gpointer db, GError** error);
```

Listing 4.2: JULEA backend SQL interface

The function `j_sql_open` opens a new connection to a database backend. To prevent any conflicting access, each thread must call this method on its own. The returned

pointer is used as an opaque handle to the database connection. Before a SQL enabled thread is terminating itself, `j_sql_close` must be called on the corresponding backend connection.

`j_sql_prepare` and `j_sql_finalize` are called to create and finalize a prepared statement. Different database clients provide very different prepared statement functionality. Some databases like MySQL enforce a lot of preparation and cleanup, while other databases like SQLite handle the memory management internally. To be able to support a wide variety of database interfaces, the prepare function requires all column types together with the SQL string directly at initialization time.

The passing of parameters is different between different databases as well. To provide a common interface, the memory management has to be included within the wrapper functions. Another trap is the indexing of parameters in different database interfaces. Some start the numbering of parameters at one, and some at zero. The wrapper implementation needs to make sure, that the indexing works the same way for any database. To receive a value from the query, the value type needs to be specified as well. The main reason for the requirement to provide the variable type here too is to allow error checking for consistent variable types.

The first call to `j_sql_step` executes the query, following calls iterate over the result set rows. Some databases use the same function for execution and iteration of SQL queries, while other databases separate this functionality into different functions. The wrapper function must internally verify, that the function performs consistently across multiple database types. If an iteration over the result set is finished, the `j_sql_reset` function needs to be called afterwards, to reset the prepared statement, such that it can be reused.

For the transaction management, additional functions are provided. Even if most SQL databases allow transaction management through the SQL string, this is not true for every database.

SQLite was chosen as the first database backend for the newly created JULEA DB backend type because this database was already used in another JULEA backend type. Because SQLite uses SQL as the input language, the backend implementation must issue SQL statements. To simplify the implementation of additional SQL-based backends, the SQL database-specific code is separate from the generic SQL wrapper implementation.

SQLite itself allows multiple parallel threads, but if multiple writes occur concurrently, SQLite returns only one error code indicating that the requested SQL query should be rerun later. The retry must then be performed either by the JULEA DB backend or by the application. The expected use-case of the JULEA DB client is that MPI applications with very many processes write to the same backend at the same time. Due to existing SQLite-busy errors within a multi-threaded application on a single SQLite table, the SQLite code is now protected by a global lock. This avoids large retry-loops within the implementation at the expense of performance, since all queries to the JULEA DB SQLite backend are serialized.

The SQL language itself is standardized. However, in a file system that is expected to contain several petabytes of application-specific data, the severe limitations of the database system used may become relevant. Each database system has different restric-

tions on the number of columns that can be used in a schema definition or the number of columns that can be specified in a SQL-where clause. Normally these restrictions should not be a problem, but the relevance of these restrictions depends on the implementation in an unknown collection of applications that use the JULEA library.

In a file system containing petabytes of data, only a relatively small fraction of the total data is metadata [Weil et al., 2004]. Currently, JULEA only issues errors if the SQL backend cannot execute the requested queries. Later, JULEA should tell the application which limitations exist for the scientist to debug more easily the application. In JULEA, a global metadata manager is planned, but it does not yet exist. Later, this metadata manager identifies which backend implementations can meet the application requirements and automatically selects appropriate backends. As a result, the application no longer needs to be informed about restrictions later. This automated decision could be based on a combination of the hard limitations of backend implementations and speed measurements for different use-cases. The metadata manager would need an additional interface to the backend implementation to determine the limits and expected metrics of various functions to accomplish its task.

The current JULEA DB backend implementation does not have a full static SQL schema definition. The only known table contains the association between JULEA DB schema names and the variables contained therein. All other tables are created at runtime directly on behalf of applications from the JULEA DB.

4.3 Usage example

The JULEA DB client-side interface allows many different complex usage scenarios. Therefore here only a few selected use-cases can be shown. Most function prototypes in the JULEA DB client interface have a `GError` pointer as the last function argument. To reduce the sample code included in this thesis, the error handling is omitted here. For more advanced examples you could read the JULEA unit tests. There the error handling is used excessively.

Every example requires an instance of `JBatch`. This instance is used to communicate with the backend. As everywhere else in JULEA, the batch structure can be used to issue multiple functions at the same time.

Every function defined in the JULEA DB client interface is prefixed with `j_db_*`. Additionally, every struct is prefixed with `JDB*`. These prefixes allow a fast overview, where which function is defined.

4.3.1 JDBSchema

Listing 4.3 is showing, how a small example schema can be defined. To show multiple use-cases, this figure shows how to create a new schema, as well as how to open an existing schema. Opening an existing JDBSchema is much easier than creating a new one.

```
void schema_create_open(gboolean create)
{
	g_autoptr(JDBSchema) schema = NULL;
	g_autoptr(JBatch) batch =
	↪ j_batch_new_for_template(J_SEMANTICS_TEMPLATE_DEFAULT);

	schema = j_db_schema_new("test", "variables", NULL);

	if(create)
	{
		gchar const* idx_file[] = {"file", NULL};

		j_db_schema_add_field(schema, "file", J_DB_TYPE_STRING,
		↪ NULL);
		j_db_schema_add_field(schema, "min", J_DB_TYPE_FLOAT64,
		↪ NULL);
		j_db_schema_add_field(schema, "max", J_DB_TYPE_FLOAT64,
		↪ NULL);

		j_db_schema_add_index(schema, idx_file, NULL);

		j_db_schema_create(schema, batch, NULL);
	}
	else // open
	{
		j_db_schema_get(schema, batch, NULL);
	}
	j_batch_execute(batch);
	// use the schema
}
```

Listing 4.3: JULEA DB client interface example showing the creation and loading of a JDBSchema

At first, the function j_db_schema_new is used, to allocate a new JDBSchema object with a given namespace and name. This step is required for both, creation and loading of a schema.

If the schema is going to be newly created then the next step is to define the field structure. In this example the fields `file`, `min` and `max` are added. The names and the types of the fields can be chosen by the application. It is recommended, that prefix underscores should not be used by applications, because this prefix is used by the backend to auto-generate some fields. Later some additional operations might be auto-generated too using the underscore-prefix.

After the definition of all fields, optional indices might be defined. In this example, only a single index on the `file` field is added. It is important, that the index array is NULL-terminated. The definition of the index function allows indices over multiple columns as well as multiple independent indices. These indices are created within the SQL backend.

After the structure definition is finished, the schema is finally published to the backend. Here the functions `j_db_schema_create` and `j_batch_execute` are used together.

If the `JDBSchema` is already defined then the structure only needs to be loaded. A single call to the get functions issues the request to load the structure. After the batch is executed, the schema holds the structure definition, as it is stored in the backend.

If applications are executed multiple times, it might be unknown, if the requested schema already exists or not. In this case, it is recommended, that the application first tries to load the schema. If the schema is not existing, an error is returned. In case of an error, the schema needs to be created. Afterwards, to gain a consistent full view to the schema, the application should load the schema again. This has multiple advantages. First, the application can verify, that the schema creation was successful. Second, the application can now see the auto-generated fields. Especially the `_id` column can be a very useful identifier for many applications.

4.3.2 JDBEntry

The main use-case of JULEA DB is to add new data to a backend. For this use-case, the client-side JDBEntry is implemented. Listing 4.4 shows the insertion of an entry. For simplicity reasons, the same schema definition as in Listing 4.3 is reused.

```
void entry_insert(JDBSchema *schema)
{
        g_autoptr(JDBEntry) entry = NULL;
        g_autoptr(JBatch) batch =
         ↪  j_batch_new_for_template(J_SEMANTICS_TEMPLATE_DEFAULT);

        gchar const* file = "demo.bp";
        gdouble min = 1.0;
        gdouble max = 42.0;

        entry = j_db_entry_new(schema, NULL);
        j_db_entry_set_field(entry, "file", file, strlen(file), NULL);
        j_db_entry_set_field(entry, "min", &min, sizeof(min), NULL);
        j_db_entry_set_field(entry, "max", &max, sizeof(max), NULL);
        j_db_entry_insert(entry, batch, NULL);
        j_batch_execute(batch);
}
```

Listing 4.4: JULEA DB client interface example showing the creation and insertion of a JDBEntry

At first, in line 10 the new JDBEntry to be inserted is allocated. During the entry allocation, the schema is supplied. This allows the client-side implementation, to obtain the variable types information. These variable types are required when the client sets the values for the different fields.

After the entry is allocated, all desired fields are initialized with the given values. Every field, which is not explicitly initialized, will be set to NULL within the backend.

Finally, the entry is inserted into the batch operation. After all desired entries are added to the batch, the batch is executed, and all data is transmitted to the database.

4.4 Adding new backend implementations

The main focus of JULEA is, to allow fast prototyping of exotic storage interfaces.

All backend type interfaces are defined in the file `include/core/jbackend.h`.

The first step towards a new backend implementation is to choose a matching backend type. Currently, three types of backends exist. Each backend type provides a different set of functions to the client-side API. This directly results in different requirements on the backend side.

Depending on the targeted backend, it might make sense, to allow the backend to be loaded either on the server or client-side. Allowing the backend to be loaded on both sides may allow easier testing.

Finally, all required functions in the `JBackend` struct need to be implemented.

To add a new backend implementation to an existing JULEA installation, only the backend library needs to be added to the existing installation. No modification of existing libraries is required. This ensures the high stability of applications, which use older versions of other backend implementations.

4.5 JULEA debugging

JULEA uses the GLib testing framework to verify that the library is working properly.

During developing the JULEA DB backend type, it turned out that traditional unit tests are not enough to find all sorts of bugs. Some errors occur due to race conditions, others only after complex sequences of library calls. Because of this problem, the JULEA DB backend and client code work with various debugging strategies.

4.5.1 AFL

To find and eliminate further errors in the code, the JULEA DB code is also tested with the framework American Fuzzy Lop (AFL) [lcamtuf, 2019]. The difference between AFL and traditional unit tests is that traditional unit tests reset the library to the initial state after each test case. On the other hand AFL tests reuse the library state of previous test runs. Code tested by AFL maintains its internal state and allows for many different test case combinations without the developer having to explicitly write down all test sequences. The core of AFL is a pseudo-random generator that generates binary streams. These binary streams are passed to the library under test. The library under test analyzes these binary streams and performs the functions defined by the binary data. As a result, the library calls several functions, each with different parameters. The library behaves as if there were an application that constantly writes random data to the library. After a few hours, millions of different test cases are executed. Each of the test cases consists of a series of random function calls. Together, a large part of the library is executed in various combinations. The code and branch coverage of the library under test easily reach high levels such as 95%. This greatly reduces the likelihood of large library errors.

4.5.2 Mockup

Another way to test JULEA is to use mockup functions to facilitate the testing of complex or slow code. For some test cases in the JULEA DB backend, the network socket code was replaced by a mockup. In this way, the test framework can test the construction and parsing of network messages within a single process without sending a true network packet. If the network code is simulated in a single process, it is much easier to test multiple instances of the library simultaneously.

4.5.3 Static code analysis

The clang tool suite contains the static code analyzer scan-build. This analyzer adds an extra step to the compilation process. During the analysis phase, the tool looks for potential memory access errors. Unfortunately, the `g_auto_free` and `g_auto_ptr` annotations from the GLib cause many false-positive warnings about memory leaks. The static code analysis alone cannot find complex errors in parallel applications, as only a few statistical checks are performed on large functions.

4.5.4 Error handling

Different types of errors require different types of error handling. The programming errors within JULEA should be fixed within JULEA and not be shown to any application. Recoverable errors caused by improper use of the API should instead be reported to the application for the application to respond to these errors.

GLib defines its own collection of error functions for both error classes. For the in-library errors, use `g_return_if_fail` and `g_return_val_if_fail`. These functions log a warning that their assertion is violated. These functions then either abort the current function with the specified abort values, depending on the environment variables or terminate the entire program with an error. In the debug version, both variants are acceptable, but in the release version of JULEA, it would be inconvenient if a server process is suddenly stopped because a single application misuses the API.

For recoverable runtime errors caused by applications the functions `g_set_error` and `g_set_error_literal` should be used. These functions generate an error code along with an error message and send both to the application. Thereafter, based on the error code, the application may decide to either ignore or handle the error. The advantage is that the entire system will run stable even if a JULEA client receives error messages.

4.5.5 Other

To ensure the correct behavior of the benchmarks, the code is extended to a debug mode in which each return value is explicitly tested to ensure that it matches the expected result. In the final measurements, these tests are skipped because they do not resemble the behavior of an application in practice.

5 Evaluation

In this chapter the properties of the JULEA DB implementation are evaluated. In addition, the properties of the HDF5 VOL Plugin based on the JULEA DB library are considered. Various benchmarks are performed to compare different aspects of the new implementation.

All benchmarks run on a consumer desktop PC. The used CPU model is i7-8700K at a clock rate of 4.4 GHz. The RAM has 32 GB of DDR4 memory at 2.4 GT / s. The RAM is large enough to keep all benchmark applications completely in memory. The HDD on which some benchmarks are run is a TOSHIBA DT01ACA300, which rotates at 7200 rpm.

5.1 Synthetic HDF5 benchmarks

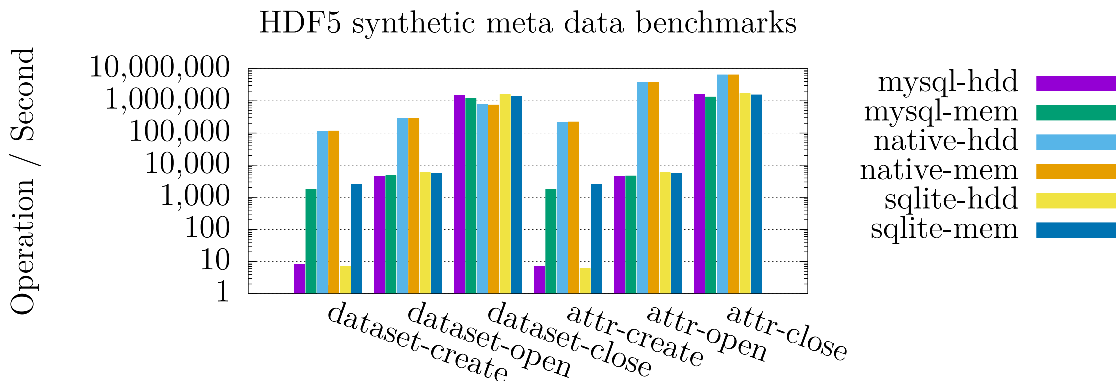


Figure 5.1: HDF5 synthetic metadata benchmarks

Figure 5.1 shows the speed differences between native HDF5 and the implementation of the HDF5 VOL Plugin in JULEA using multiple backends. The native HDF5 is much faster because it can use the file system cache. For this reason, the native HDF5 implementation has the same speed regardless of the storage medium used. In contrast, the SQL databases are very pessimistic, ensuring that all their data is synchronized with

persistent storage before a confirmation signal is returned to the application. This allows transaction semantics, but requires more time by design. In addition, JULEA assumes a server environment in which the persistent storage is accessible only through other dedicated storage nodes. JULEA must send all data to be stored in SQL at least once over a virtual network. If the SQL database needs to be stored on an HDD then the read and write speed is very low. The speed differences between the currently existing JULEA backends are minimal when only a single process accesses the backend.

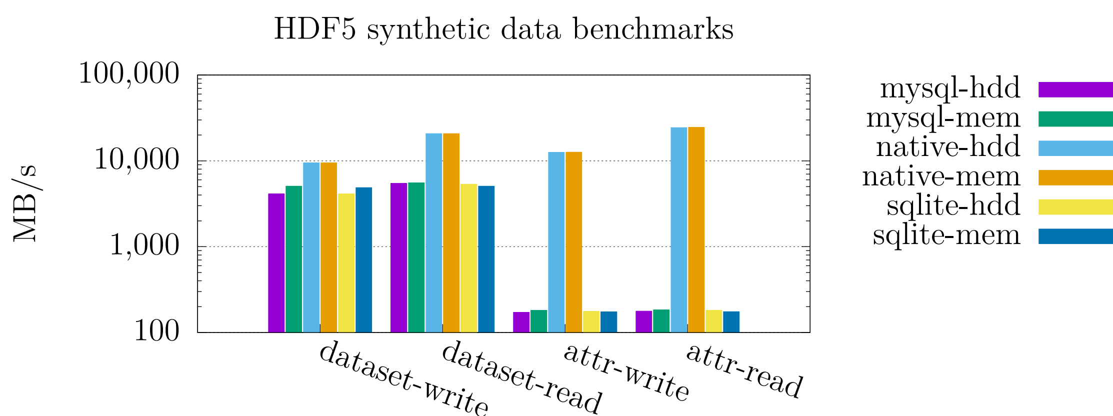


Figure 5.2: HDF5 synthetic data benchmarks

Figure 5.2 shows the speed at which data is written via the HDF5 VOL Plugin. The JULEA OBJ backends are not implemented in this thesis, but have similar speed issues. Native HDF5 can use the file system cache to write large amounts of data. JULEA DB instead assumes a distributed storage. Currently JULEA is kept simple, therefore there are no caching layers within JULEA. Internally, JULEA must repack the data and send it at least once over a virtual network connection before the data is stored on the HDD. That takes time.

5.2 Enzo benchmark - a real application using HDF5

Enzo [comunity, 2019] [Bryan et al., 2014] is a simulation program that can be applied to a variety of very different physical problem domains. Most examples provided by Enzo were carried out on a trial basis with JULEA and it turned out that most of the examples have very similar IO patterns. The reason for this is, that there are just a few different simulation classes within Enzo, but a huge amount of examples, which use these with different configuration parameters.

The following benchmark uses the Enzo sample parameter file *CollapseTestNonCosmological*. This test problem initializes a sphere with constant density in the middle of

the simulation area. The sphere is in a pressure equilibrium with its surroundings. The simulation dynamically collapses the sphere until it reaches its peak density. The radiant cooling is turned off so that the sphere bounces and later reaches a state of equilibrium.

Since JULEA focuses on the IO characteristics, the example has been changed to store the data at each time step. After saving the data, Enzo is restarted from the previously generated savepoint. This significantly changes the relative required time of the different functions within Enzo. In a normal production environment, IO is performed only every few hours. This extreme of saving data that often should demonstrate the efficiency differences between the storage interfaces.

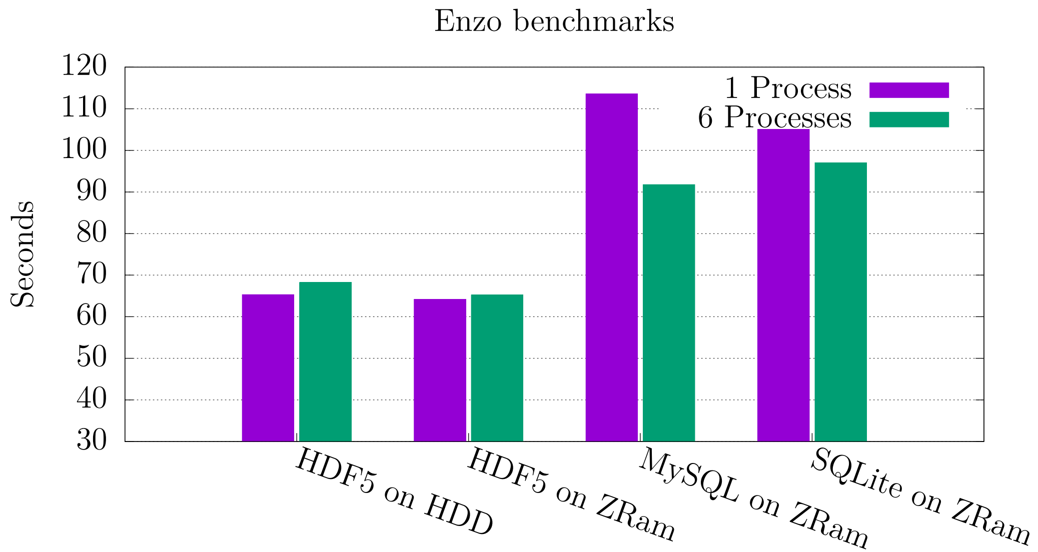


Figure 5.3: Enzo benchmarks

Figure 5.3 is showing the time required to save and restart Enzo 40 times while calculating the values described above. The HDF5 VOL Plugin implementations, which use an HDD as the storage medium, are so slow that the benchmarks are aborted due to a one hour timeout. One hour is significantly more time, than the required time by all other variants.

When JULEA is operated on ZRam, the required wall time increases by 40% to 70% compared to native HDF5. It is not easy to say how much time is needed exactly for IO, as there are side effects. An important factor in an MPI application is that all processes are waiting for each other. Even though the IO is only slightly slower, the entire application takes much longer.

It can be seen that the native HDF5 library on an HDD reaches speeds similar to those on a ZRam partition. Because the JULEA DB SQLite implementation uses a global lock to access the database, the time it takes increases as the number of processes increases. The JULEA DB MySQL implementation can instead use multi-process access to the database. This allows faster access to the metadata since multiple processes can access JULEA DB at the same time.

5.3 Synthetic JULEA DB client benchmarks

In this section, several properties of the JULEA DB implementations are evaluated. Multiple aspects are compared using different parameters or configurations.

All benchmarks insert entries into already defined `JDBSchema`. The `JDBEntry` itself are allocated initialized during the time measurement.

The amount of entries is specified by the x-axis, the resulting performance on the y-axis. All operations are performed as batch operations with a maximum size of 10000. If more than this amount of entries are added, multiple batches are performed after another. It might not be common, that an application adds thousands of metadata values at the same time, but on the other hand, an application should not add every metadata value as a different transaction.

The following benchmarks do not include explicit measurements for the selection of entries, because the performance changes are similar to the delete and update queries.

Every measurement is repeated until at least 60 seconds are used within JULEA. Some plots terminate in the middle of the figures, due to a timeout of 120 seconds. Because of the number of measurement points, and the resulting fluent plots, it is expected, that the measurements are precise enough.

Each benchmark highlights a different aspect of the performance. If an aspect is not explicitly compared in the figures, then the parameter is set to a default value.

The used default parameters are:

- entries use 50 fields
- there are indices on the queried field
- the storage medium is a ZRam partition
- 6 processes are used

5.3.1 Null-Backend

At first the theoretical maximum performance is measured with a `null-backend` implementation. This backend should be only used to evaluate the performance overhead of JULEA backends. All functions of the JULEA DB interface are implemented. Within this backend, only a single `JDBSchema` and a single `JDBEntry` are stored as an exact copy in memory as they are received. The number of inserted and deleted entries is memorized too, every other input parameter is ignored. If the entries are requested, then the last inserted entry is repeatedly returned as often as there should be entries in the backend. The reason for storing a single object of each type is, that the client side library behaves differently if the amount of received data changes. Otherwise there is no error-checking performed in the null-backend implementation.

During the benchmarks and the usual application usage, a `JDBSchema` is defined once, and than reused within every process. To perform multi-process benchmarks, this null-backend uses locks, to share the same schema definition across all processes. The

existence of a correct schema is required by the client side JULEA DB library, to perform any other operation on behalf of the user.

To be able to compare this measurements with all other benchmarks, the null-backend is loaded as a server-side component into JULEA.

	5	50	500
insert	217,647	84,069	1,722
update	235,083	80,039	1,723
delete	236,089	239,498	215,660
select one by one	204,072	89,269	1,747
select all	202,075	28,126	376

Table 5.1: Theoretical maximum performance of each operation. All values are displayed in operations per second. The columns show the impact of different field counts defined by the `JDBSchema`. The rows show the different functions which send data over the network.

Table 5.1 shows the measured maximum speed of the interface. Because this null-backend does not actually store the data, insert, update and selection of single entries reach the same speed.

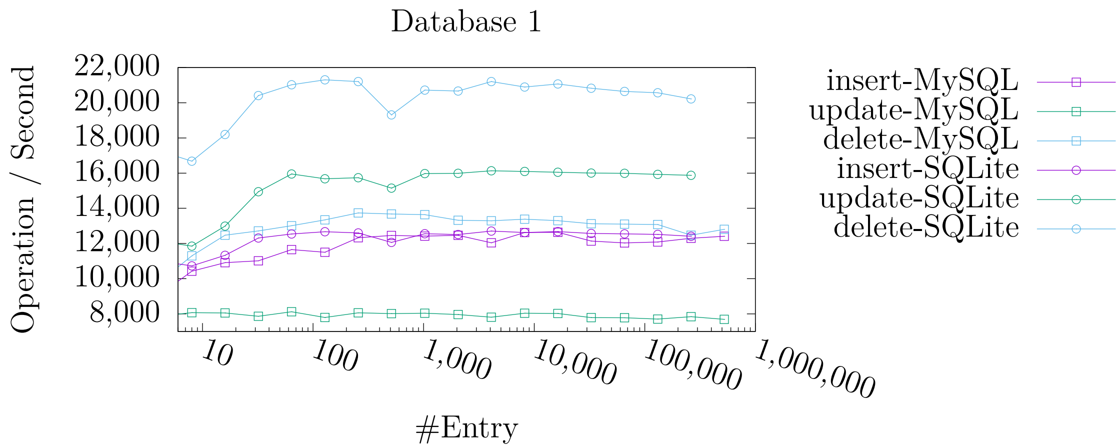
The different amount of fields in the `JDBSchema` has a huge impact on the performance. Even if the backend is replaced with a null backend, the client side library is still fully functional. Therefore the client side library needs to query the field types from the `JDBSchema`. Afterwards the fields of the BSON need to be initialized. If an BSON has 500 fields, there are many memory allocations and movements during the construction of the BSON. The more fields are added, the more memory needs to be allocated and moved.

The deletion of elements is the fastest, because this is the only operation, where only a simple `JDBSelector` needs to be sent over a network connection.

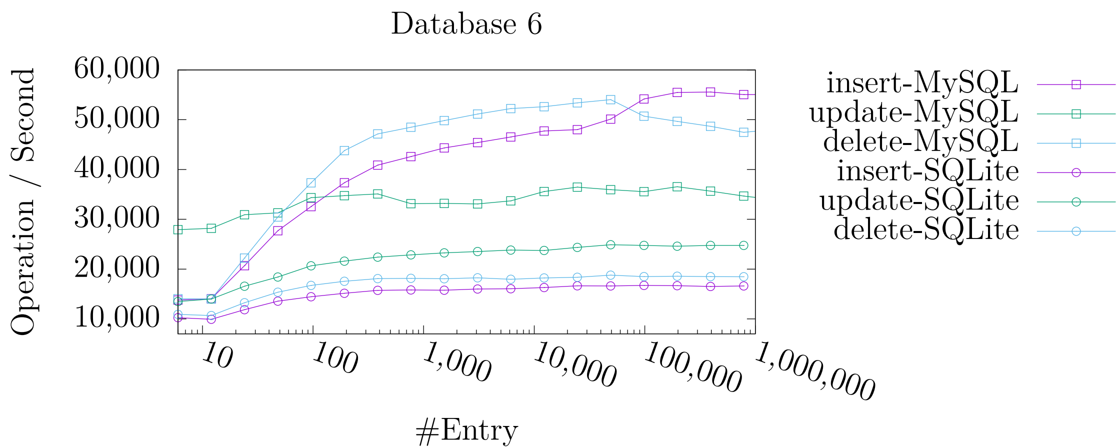
The selection performs differently depending of the amount of entries transferred at once. If only single entries are requested one by one, then the backend can just return a copy of the existing `JDBEntry` which is already stored in-memory as BSON. If all entries are requested at once, then the same speed penalty takes effect as earlier in the field count. Each entry is added one by one to a huge BSON, which in turn needs to be reallocated and moved around during construction. Finally when this huge BSON is constructed, then it is send at once over the network connection to the client.

5.3.2 Impact of the used database

Figure 5.4 compares the influence of different process counts to the overall performance of JULEA DB.



(a) JULEA synthetic database benchmarks using 1 process



(b) JULEA synthetic database benchmarks using 6 processes

Figure 5.4: Comparison of the existing JULEA DB backend databases using 1 or 6 processes which perform the same operations

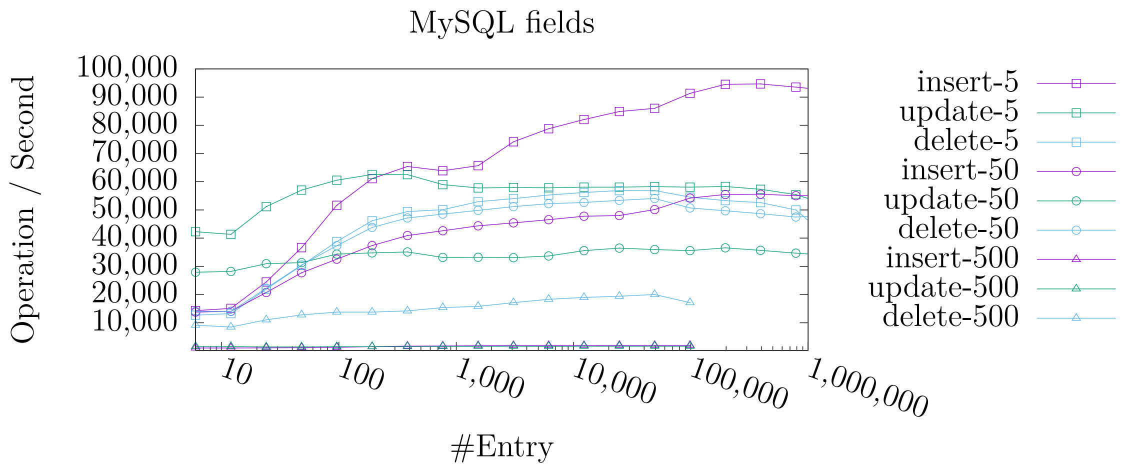
First Figure 5.4a shows the performance development if just a single process writes to JULEA DB. The amount of already stored data seems to have just a little impact on the performance. However, this figure shows, that the SQLite backend is faster than the

MySQL backend implementation.

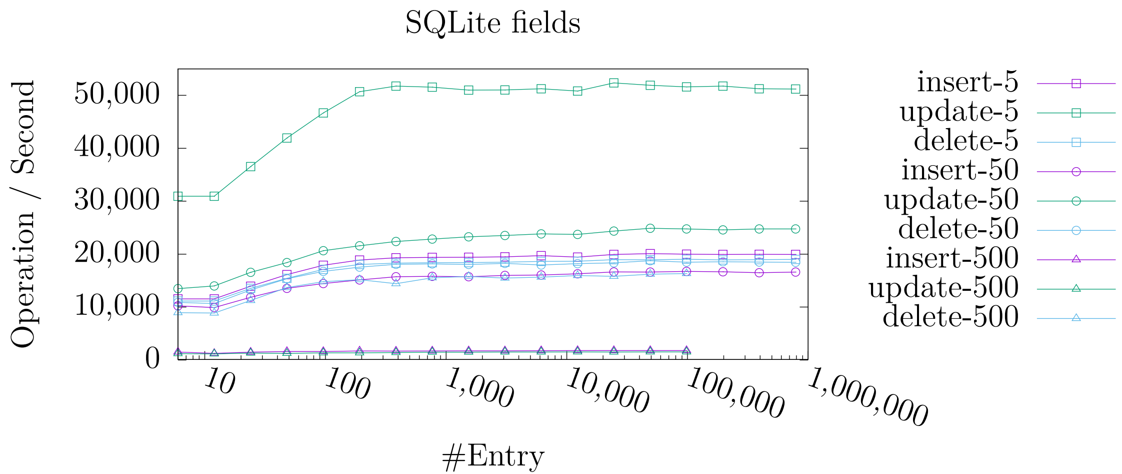
In the Second part Figure 5.4b shows the same benchmark, using 6 processes at the same time. In contrast to the first figure, now the MySQL backend is faster than the SQLite backend. The reason for this speed difference is, that the SQLite backend implementation requires a global lock around all database operations. The lock is required because all SQLite connections directly operate on the same file. Without the explicit lock, the SQLite API would throw errors.

5.3.3 Impact of the number of fields

Figure 5.5 compares the influence of different amounts of fields within the `JDBSchema` definition. The same benchmark is executed with 5, 50 and 500 integer fields. Obviously the performance is much higher, if less data needs to be inserted. The interesting part of this figure is the magnitude of the performance difference. Both implemented backends show, that `JDBSchemas` with 500 fields are very bad for the performance. It is expected, that the most common usage of JULEA DB uses less than 50 fields within a single `JDBSchema`. Most operations in both backends reach their peak performance, if around 100 entries are inserted in the database. If more than 100 entries are present in the database the performance remains the same. The exception to this observation is the insertion of elements in the MySQL backend. Here the performance increases with larger amounts of data. The reason is, that the index needs to be rebuild only once per transaction. If a huge amount of entries is inserted at once, the rebuild-insertion ratio is improved.



(a) JULEA synthetic database benchmarks using MySQL

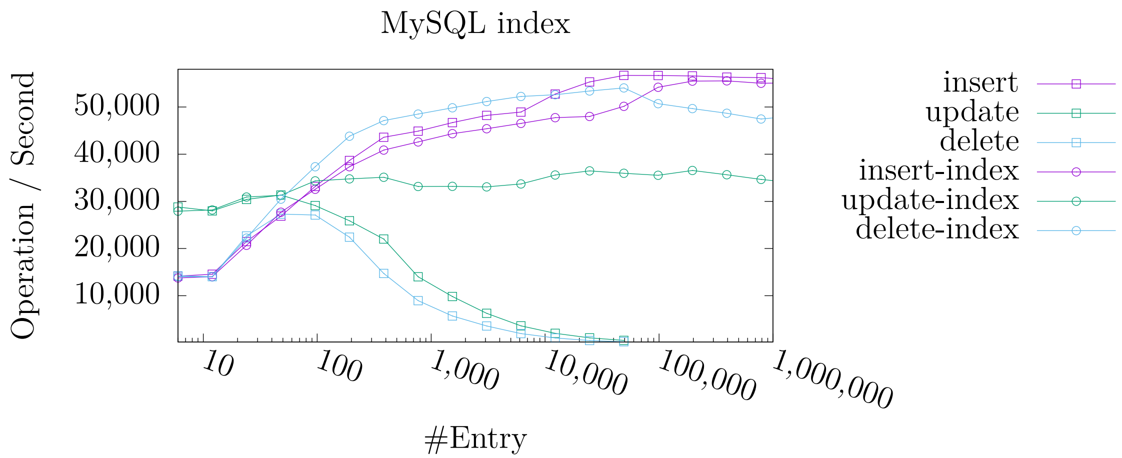


(b) JULEA synthetic database benchmarks using SQLite

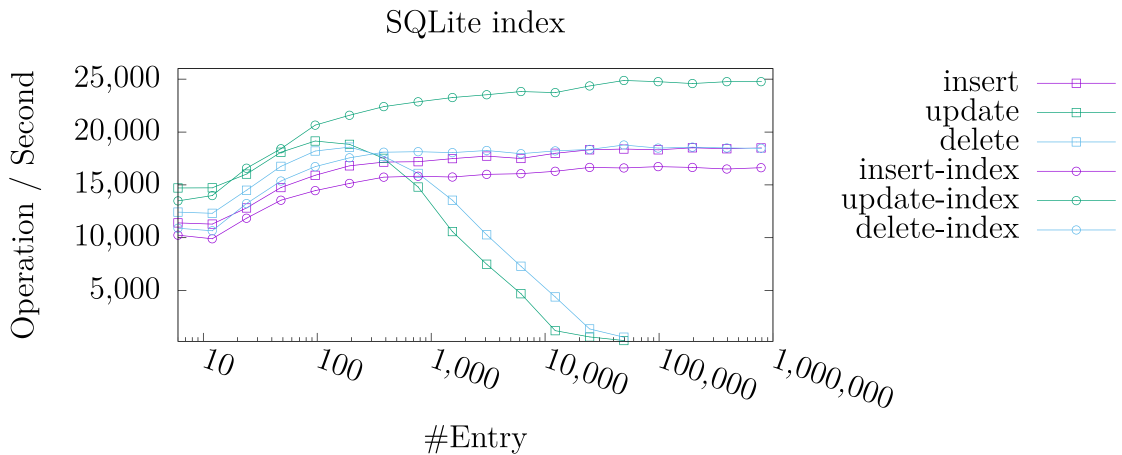
Figure 5.5: Comparison of the existing JULEA DB backend databases using a different number of fields within the schema definition

5.3.4 Impact of Indices

Figure 5.6 compares the influence of indices within the JDBSchema definition. As expected, the performance for updating values can gain huge speed improvements. To update values, the values need to be selected first. Because of the existing indices, the selection can be performed much faster, so that the operation completes faster too. The same applies to the deletion of entries.



(a) JULEA synthetic database benchmarks using MySQL



(b) JULEA synthetic database benchmarks using SQLite

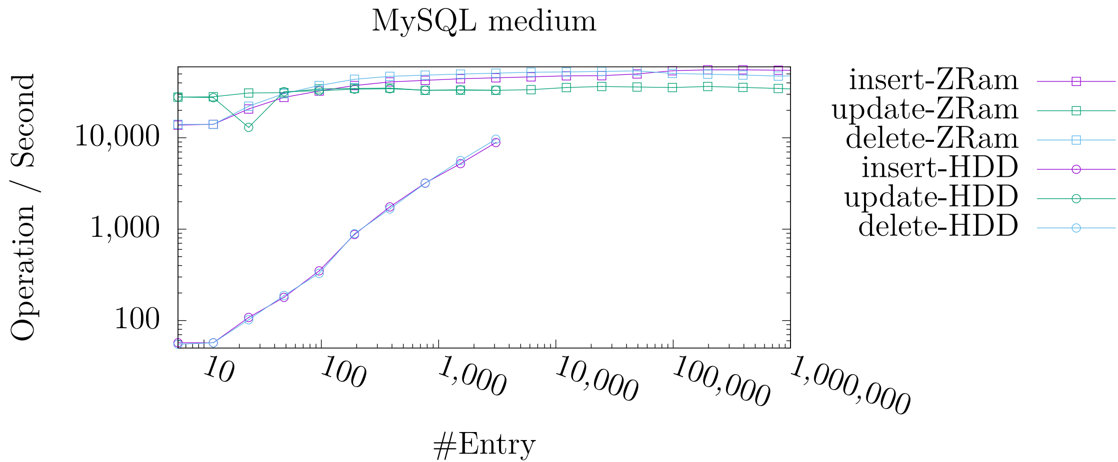
Figure 5.6: Comparison of the existing JULEA DB backend databases influence of indexes on the queried field

The performance penalty for data insertion is much smaller than expected. However both backends show, that the existence of indices costs time.

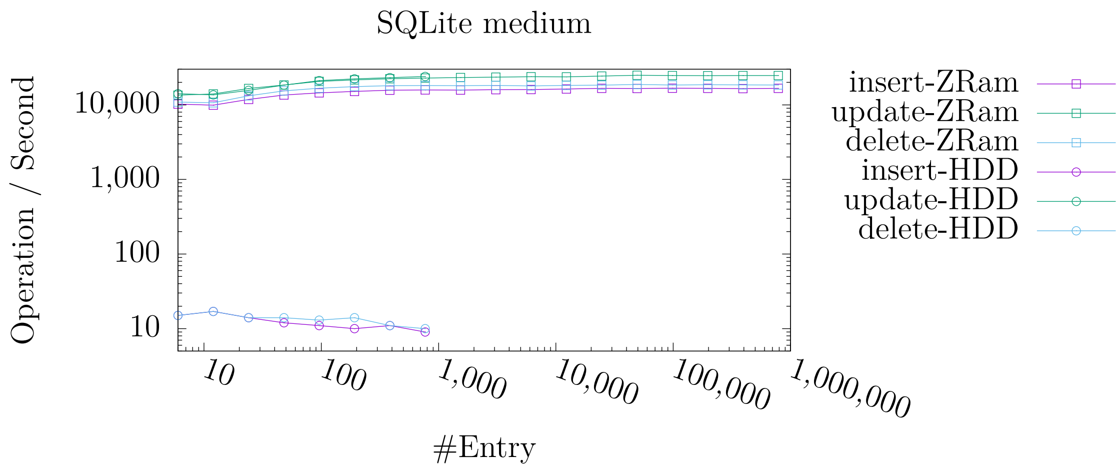
JULEA DB is intended to store huge amounts of data. Both backends show, that if entries need to be updated, indices should definitely be used, if the amount of data is large. The same applies for the querying of data, which is not shown in this figures.

5.3.5 Impact of the storage medium

Databases are stored on some kind of physical storage. To fulfill the ACID properties, databases are very pessimistic regarding the physical storage. Every operation needs to be flushed multiple times to the physical layer before any confirmation response is sent to the client. Figure 5.7 compares the performance influence of the storage medium.



(a) JULEA synthetic database benchmarks using MySQL



(b) JULEA synthetic database benchmarks using SQLite

Figure 5.7: Comparison of the existing JULEA DB backend databases influence of the storage medium below the database

The performance of SSD storage is not included in the benchmarks, but small samples

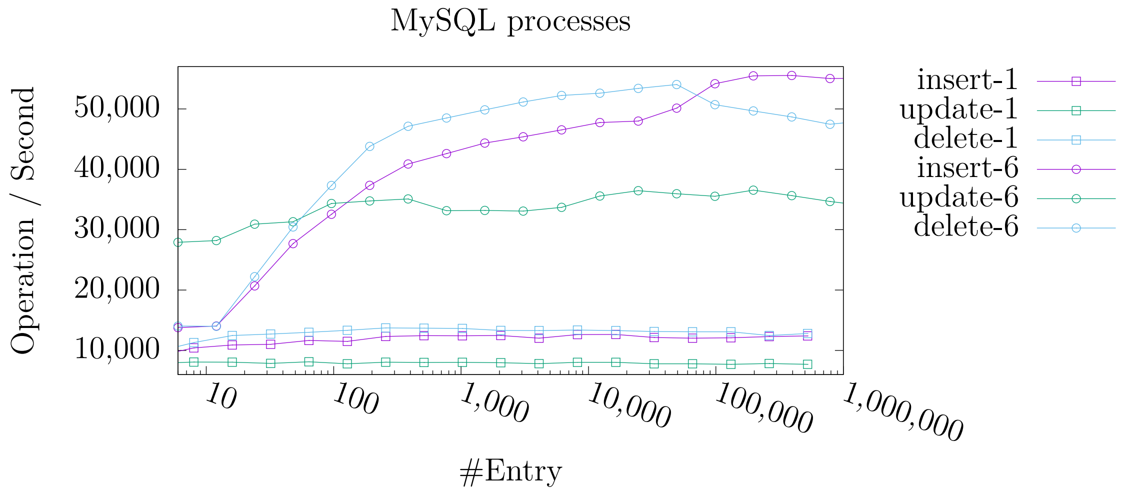
indicate, that the performance is somewhere between HDD and ZRam. As expected, HDD storage is much slower than in-memory databases.

The interesting part is in Figure 5.7a. The more entries are stored within the database, the better the performance. The reason is, that databases are only forced to flush their operations at the end of a transaction. Because of the huge transaction size, all performed measurements on the HDD are performed in a single transaction. This improves the ratio between inserted entries and the amount of flushing the storage. As a result the speed increases.

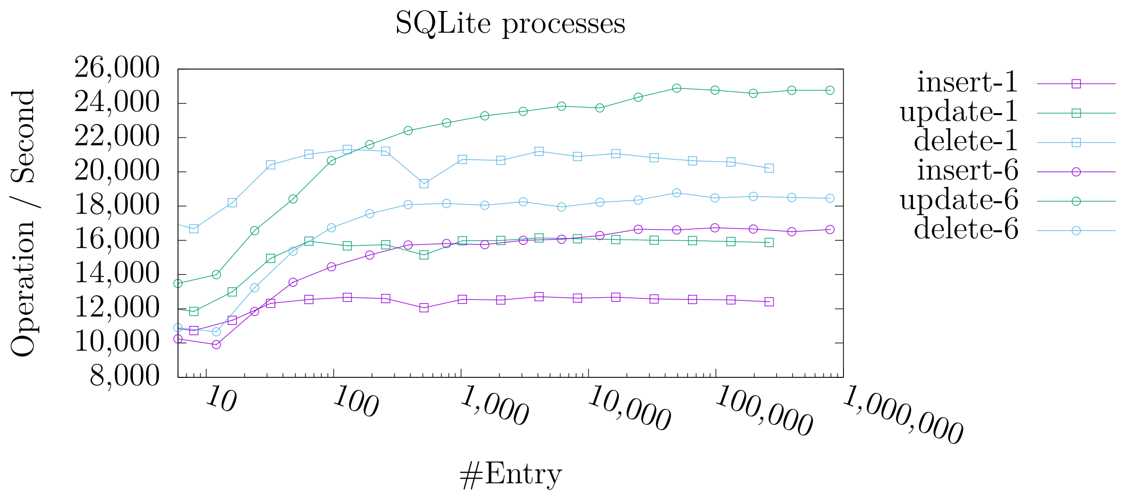
5.3.6 Impact of the Process count

Finally, the influence of different amounts of processes are directly compared in Figure 5.8.

The MySQL backend implementation reaches a nearly perfect speedup if huge amounts of data are to be written. The slowest measurement reaches an average speedup of 3. This is not perfect, but in summary, the increased number of processed increases the performance too.



(a) JULEA synthetic database benchmarks using MySQL



(b) JULEA synthetic database benchmarks using SQLite

Figure 5.8: Comparison of the existing JULEA DB backend databases influence of the process count accessing the database in parallel

On the other hand, the SQLite backend shows very poor results. As mentioned earlier, a global lock is used. The speed increases with more processes, but the effect is very small. The improvement comes from the fact, that the messages need to be sent to a central JULEA DB server, which performs the SQLite operations. The messages can be sent in parallel to the server, and are processed only in serial afterwards.

5.4 New functionality

In addition to shortening access times for application data, JULEA DB offers new features for subsequent post-processing tools. Previously, a post-processing tool had to parse each input file get an overview of the minimum and maximum values throughout the simulation. With the JULEA DB backend, this information can be retrieved easily and quickly from the JULEA DB database. Because JULEA DB metadata is stored in a database, complex data queries can be performed much more efficiently. During regular applications, the JULEA DB interface should be used to query data. During post-processing it might be more efficiently to access the database server directly through its SQL interface. To prevent the damaging of metadata, this direct SQL access must be restricted to read only operations.

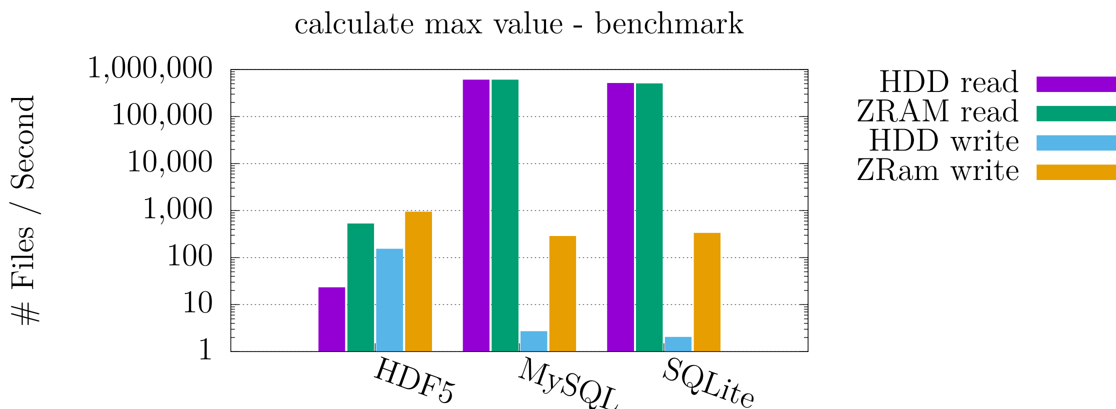


Figure 5.9: HDF5 benchmarks querying data across multiple files

Figure 5.9 shows the results of a synthetic benchmark application that writes 1,000 files, each containing 1,000,000 random integer values. After writing the data, the application calculates the maximum value across all files. Using ZRam as the backend device reduces the write speed of the application by a factor of 3. During the read phase, JULEA DB based access provides 1,000-10,000x faster access to the metadata. If the application or post-processing tool read the metadata multiple times, the speed can be significantly improved. In particular, if it is not clear which data is of interest in which files, it can be very useful to query several different properties and later decide which query returned the most helpful content.

To use JULEA in an existing HDF5 application, only a few very small changes are required because HDF5 has its own HDF5 VOL Plugin interface. With this interface, it is possible to configure HDF5 to use JULEA as the backend. An example how this plugin can be used can be seen in Listing 3.7.

In addition to an accelerated post-processing tool, JULEA DB can also identify metadata that is significantly different from the metadata in the other files created by the application. This can be useful for detecting abnormal changes in the data. This can lead to new insights into huge amounts of data. Without JULEA DB, this would require an application that indexes all the desired data and then executes the query. Without any index, the query would be even slower.

6 Related Work

This chapter shows some related work and highlights the main differences to this thesis. Furthermore, the advantages and disadvantages of some approaches are shown.

There are currently several strategies to improve the handling of huge amounts of data.

6.1 Compression

Compression can be used to improve the data throughput between the volume on which the data is stored and the application that requests the data. The cause for the higher data throughput is that the computational speed of processors increases faster than the storage or network access speed. When the data is compressed on the client-side, less data needs to be transmitted over the network, which effectively increases the amount of data sent to the storage. If the compression rate is high, more data can be cached on the client node, which also increases the observed speed. To further increase the compression rate, some files may allow the file system to compress the content with lossy compression algorithms. Even if only the data is compressed, the metadata of the file system is also affected. Now, the used compression algorithm together with its properties has to be saved alongside the data. Due to the compression, the data size is changed. The metadata must store the mapping between the actual data size and offset, as well as the stored size and offset.

Compression can be applied to different layers. One approach is to apply the compression directly within the POSIX file system Lustre [Fuchs, 2016]. The problem with compression in the file system is that, from a file system perspective, the file is just a stream of bytes. Due to missing file metadata, the compression in the file system must not be lossy.

To access the metadata during compression, another approach is to add an extra compression layer within libraries such as HDF5 before writing the data to the file system [Di and Cappello, 2016]. The metadata allows temporal and local dependencies between data to be exploited to further increase the compression rate. The disadvantage here is that the compression code must be re-implemented for each data format to be stored.

6.2 EMPRESS

EMPRESS [Lawson and Lofstead, 2018] offers a structured metadata library similar to HDF5. This library forces the application to add rich metadata `tags` to its data. At the same time, only flat metadata hierarchies up to a maximum depth of 3 are allowed. There are no automatically generated `tags`. This means that the application must align its code with this library. If the application appends more metadata, other metadata-based applications can also improve. If useful `tags` are appended to the data, the post-processing tool can request data based on the `tags`. This allows the post-processing tool to request much less data. As less data is requested and analyzed, the entire analysis phase is faster. The metadata from multiple files are stored together in an RDBMS database backend. This paper states to offer a faster metadata interface than HDF5, especially when large volumes of processes are used. This library supports importing and exporting metadata to continue to use the data in another cluster, as long as the other cluster also uses EMPRESS. To add `tags`, the application must follow a strict program structure as required by EMPRESS. This means that the application is assumed to perform a time-based simulation that has many time steps that store very similar structured data. This may be common, but this will not apply to any application. EMPRESS adds some functions that need to be called between different time steps. This forces parallel applications to synchronize between each time step to perform these EMPRESS functions.

6.3 Dynamic metadata Management for Petabyte-scale File Systems

In their paper [Weil et al., 2004], the authors analyze the importance and potential improvements of metadata servers in huge distributed file systems. In petabyte server clusters, metadata servers are critical to the overall performance of the server. The main reason for their importance is that every reading and writing of data must be recorded somewhere, which data belongs to which file and which storage server still has storage space available. In this paper, the focus is only on file system metadata and currently contains no file metadata. For this reason, the POSIX standard can still be used. To achieve their improvements, the metadata is split across multiple metadata servers. Each server and even the clients store so much metadata in their caches that fast access to all stored file data is possible. To reduce the load on the dedicated metadata servers, clients can calculate the object store IDs themselves from a small seed.

In this work, there are several layers of caches inserted between the metadata servers and the application. In summary, the improvement of this approach is entirely dependent on the existence of a lot of free RAM that can be used for caching. Because there is no file metadata in the file system metadata, queries across multiple files are not speed up because any files that may be part of the query must be opened on the client computer. Even if the files could be located more quickly on the object storage servers, much of the file content still needs to be sent to the application.

6.4 Semantic file system

The 1991 paper [Gifford et al., 1991] proposes semantic file systems for the first time. In such a file system, there are special virtual folders that are interpreted as queries. The virtual folder paths allow such a file system to be easily integrated into existing POSIX file systems. These virtual path queries allow attribute-based access. Because the virtual paths are just an extension, this type of file system is fully compatible with traditional file systems. The paper states that this storage abstraction is more effective than the standard file systems because it is now possible to query part of the file content without the application knowing the exact file paths.

The attributes, which can be used in queries, are extracted by file type-specific transducers. These transducers parse each file and create an index that can later be used by the queries. In order to keep the index up-to-date, the file system must update the index each time the file is created and updated.

Overloading of file systems is used on UNIX systems in multiple locations. For example, each connected device appears as a special device file. Unix systems define sockets, which can be used by processes to find each other. In some operating systems, processes are also displayed as special files.

During their evaluation, the authors state that the indexing of all files is IO-bound. With the hardware development over the past 28 years, the exact bottleneck may now be in a different location, but it will still be slow to do multiple writes to the index after any file change, just to keep the indexes up-to-date.

6.5 Parallel data analysis directly on scientific file formats

The authors [Blanas et al., 2014] attempt to solve the problem of finding specific data parts inside the output of scientific applications. For this purpose, it is suggested to create indexes per file alongside the existing data files.

The authors state that much of the times used by the post-processing tool is used to convert the data from the application readable format to a format that can be read by the post-processing tool tools.

Since HDF5 and NetCDF are often used as a high-level interface for writing data, the HDF5 VOL Plugin is extended to create these index files. The authors say that HDF5 itself is finely tuned to achieve maximum peak performance on parallel file systems in large supercomputers.

The real benefit of simulations is achieved during post-processing tool, where interesting application results are highlighted. Currently, many scientists are converting their data into an auxiliary format that can be used by a query language. This conversion takes a lot of time. If indexes could be used then complex queries could be performed directly on the data without loading the entire data file. The authors' benchmarks show that their interface to HDF5 files using indexes is faster than the currently used PostgreSQL (PostgreSQL) databases alongside the HDF5 files.

6.6 Making Sense of File Systems Through Provenance and Rich metadata

In their paper [Parker-Wood et al., 2012], the authors state that the 40-year-old file-finding tools can no longer perform efficient querying on current cluster file systems. As they were created, these old tools did not have to handle as many files as they do today. It also criticizes the basic file system concept, where the application has to define its own filenames. If the application specifies enough metadata, then the file system may automatically generate file names based on the file content. This would relocate the file name invention task from the application to the file system designer. In addition, the automatically generated file names are shorter on average, because scientists often use very long file names.

Using automatically generated file structures, the computer can find data faster based on its attributes. However, if the application wants to open a particular file, the application must query the file using metadata, forcing the application to find out which file in the provided result set is what it is looking for. When scientists name their files manually, they will embed metadata in the filenames, which might become inconsistent as the application evolves.

The article states that scientists often search their data files with `grep`, `find`, `awk` and `du` to find their files and directories.

These brute-force search tools are based on inefficient linear-time functions in the file system and the access tools.

Modern scientists do not have the knowledge of database administrators. Often they are not even computer scientists. As a result these users are likely to issue vague requests, which return too many results. At first this is not a resource friendly usage of file system. On the other hand these users do not find, what they search for. Current server clusters have a similar amount of files to the Internet about 40 years ago. One could say the situation is comparable to the internet. In order to find something on the Internet, the various homepages are linked. File systems, on the other hand, are still limited to POSIX semantics.

To solve the problems with current file systems, the authors suggest adding access counters to the `inode` structure of the file system. This allows probabilistic searches based on how often a file is used. Finally, these counters can be used to find frequently used files faster. One can also use these counters to determine which files are commonly used together. This advantage is also a security issue because attackers or other applications may be able to see the information about which files are often used together.

On the Internet, the page-rank algorithm is generally regarded as the gold standard for general purpose ranking of web pages. On the Internet, the search algorithms can provide links to web pages as well as small sections of text to find the information you are looking for faster. In order to perform efficient searches in huge file systems, a comparable approach is required. The problem with binary data is that the application has no advantage when displaying random samples of binary data. The application must specify explicit metadata sections that will be displayed during a search.

The work is continued in the paper [Parker-Wood et al., 2014]. While writing their second paper, the authors had a working prototype. The authors call their prototype file system `TrueNames`.

6.7 Structured metadata for the JULEA storage framework

There already exists a previous attempt to store structured metadata within JULEA [Michael Straßberger, 2019].

Within his work, the author had defined an interface for storing and accessing metadata in the JULEA file system. The basic definition of different client side components is reused in this thesis. The client side components defined by the author are:

- The `JSMD_Type` defines an atomar data type like `Integer` or `String`.
- `JSMD_Scheme` defines a complex data type, which consists of multiple `JSMD_Type`.
- The `JSMD` object is used to write to and intended to read data from the backend.
- `JSMD_Search` should define the selection, which data should be retrieved, and provides additional functions to loop over the retrieved data. This component was not included in his reference implementation.

In comparison to this thesis, the public client side functions itself are different. One reason for the difference is, that the public function definitions of this author did not match the other JULEA functions schemes. Additionally, because the author had not finished his reference implementation, the defined interface required some functional changes too. Especially, the querying of data was not included in his reference implementation. Since the querying of data is one of the most important components of any file system, this changed different aspects of the public and internal interfaces.

7 Summary, Conclusion and Future Work

7.1 Summary

The ever-increasing computing power of supercomputers allows simulating more and more detailed models. This generates a large amount of data. In order to efficiently store these data collections as well as to quickly evaluate and locate relevant information, new tools are needed to accomplish these tasks.

In the past, there have been several approaches to improve the handling of huge amounts of files and file data. These approaches focus on finding data faster but neglect the space-efficient storage of data. Most of these alternative strategies are based on a kind of index stored either within the file system or next to the structured file. Alternatively, caches are used in different places. Both strategies require a lot of the available RAM, which the application can not use to calculate new data.

To solve these tasks, a novel file system interface was defined and a proof of concept was implemented. This new interface makes it possible to store the file metadata together with the file system metadata in a database backend. This provides the advantage of enabling complex content-based searches across multiple files. Since it is no longer necessary to open all files individually to analyze the metadata across multiple files, much time is saved. Furthermore, this allows the file system to store the data more efficiently. There are several reasons for this: On the one hand, the file system can quickly recognize right from the first writing what data is probably needed more frequently than others and store it in a preferred location. On the other hand, selective file parts can be lossily compressed if the metadata shows that higher precision is not necessary. As a disadvantage, it turned out that the writing of new data into the file system requires more time, sometimes considerably longer, depending on the application. This is because a database backend is used, which often synchronizes the data with the disk to ensure the ACID properties.

7.2 Conclusion

For this master thesis, a reference implementation of the newly defined memory interface was created. During the evaluation, it has been found that the writing of the data sometimes becomes much slower, depending on the application. This can be attributed to a more complicated storage in databases. Measurement results showed that finding files based on metadata can be extremely accelerated. The goal of storing data according

to their importance on storage media of different speeds has not yet been processed. Since the metadata already exists, this task can easily be performed without further changes to user applications later.

7.3 Future work

There are many structured file formats. Each of these structured file formats has its advantages and disadvantages. Some of the file formats are based on each other or take advantage of useful features from other structured formats such as HDF5 and NetCDF.

To further simplify the evaluation of scientific data, more of these file formats would need to be merged to unify the interfaces. In addition to merging file formats, the metadata should be more tightly integrated with file system metadata to simplify the processing of large file volumes. As a side effect, this also allows to automatically decide, if data could be compressed with lossy algorithms to further improve the compression ratio.

The strict and fixed POSIX semantics are outdated and should be mitigated or replaced by newer interface definitions.

There have been attempts in the past to define interfaes for file systems which can perform searches based on metadata. The biggest problem with this type of file system, however, is extracting metadata from any file format.

Many utilities that are currently based on the POSIX standard could be integrated directly into the file systems, because these functions are very common. With more meta data, these functions can be executed much faster.

Bibliography

[Arie Shoshani, 2019] Arie Shoshani, D. R. (2019). *Scientific Data Management: Challenges, Technology, and Deployment*. Chapman and Hall/CRC.

[Blanas et al., 2014] Blanas, S., Wu, K., Byna, S., Dong, B., and Shoshani, A. (2014). Parallel data analysis directly on scientific file formats. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data - SIGMOD '14*. ACM Press.

[Bryan et al., 2014] Bryan, G. L., Norman, M. L., O'Shea, B. W., Abel, T., Wise, J. H., Turk, M. J., Reynolds, D. R., Collins, D. C., Wang, P., Skillman, S. W., Smith, B., Harkness, R. P., Bordner, J., Kim, J.-h., Kuhlen, M., Xu, H., Goldbaum, N., Hummels, C., Kritsuk, A. G., Tasker, E., Skory, S., Simpson, C. M., Hahn, O., Oishi, J. S., So, G. C., Zhao, F., Cen, R., Li, Y., and The Enzo Collaboration (2014). ENZO: An Adaptive Mesh Refinement Code for Astrophysics. *apjs*, 211:19.

[bson, 2019] bson (2019). BSON. http://mongoc.org/libbson/current/bson_t.html. [Online; accessed August 16, 2019].

[Chaarawi, 2015] Chaarawi, M. (2015). Parallel I/O with HDF5. https://www.hdfgroup.org/2015/08/parallel-io-with-hdf5/. [Online; accessed October 27, 2019].

[comunity, 2019] comunity (2019). The Enzo Project. https://enzo-project.org. [Online; accessed September 18, 2019].

[Daley and Neumann, 1965] Daley, R. C. and Neumann, P. G. (1965). A general-purpose file system for secondary storage. In *Proceedings of the November 30–December 1, 1965, fall joint computer conference, part I on XX - AFIPS '65 (Fall, part I)*. ACM Press.

[Developers., 2019] Developers., T. H. (2019). H5MD. https://nongnu.org/h5md/. [Online; accessed August 16, 2019].

[Di and Cappello, 2016] Di, S. and Cappello, F. (2016). Fast Error-Bounded Lossy HPC Data Compression with SZ. In *IPDPS*, pages 730–739. IEEE Computer Society.

[Erich Strohmaier, 2019a] Erich Strohmaier, Jack Dongarra, H. S. M. M. (2019a). Performance development. https://www.top500.org/statistics/perfdevel/. [Online; accessed August 16, 2019].

[Erich Strohmaier, 2019b] Erich Strohmaier, Jack Dongarra, H. S. M. M. (2019b). TOP500 List - June 2019. https://www.top500.org/list/2019/06/. [Online; accessed August 16, 2019].

[Fuchs, 2016] Fuchs, A. (2016). Client-Side Data Transformation in Lustre. Master's thesis, Universität Hamburg.

[Gifford et al., 1991] Gifford, D. K., Jouvelot, P., Sheldon, M. A., and O'Toole, J. W. (1991). Semantic file systems. *ACM SIGOPS Operating Systems Review*, 25(5):16–25.

[Group, 2006] Group, T. H. (2006). The File as Written to Media. https://support.hdfgroup.org/HDF5/doc/H5.intro.html. [Online; accessed August 16, 2019].

[Group, 2019] Group, T. H. (2019). HDF5. https://portal.hdfgroup.org. [Online; accessed August 16, 2019].

[IBM, 1965] IBM (1965). Technical press release. https://www.ibm.com/ibm/history/exhibits/1130/1130_ttechnical.html. [Online; accessed September 11, 2019].

[Kuhn, 2017] Kuhn, M. (2017). JULEA: A Flexible Storage Framework for HPC. In *Lecture Notes in Computer Science*, pages 712–723. Springer International Publishing.

[Laboratory, 2019] Laboratory, O. R. N. (2019). ADIOS2. https://adios2.readthedocs.io. [Online; accessed August 16, 2019].

[Lawson and Lofstead, 2018] Lawson, M. and Lofstead, J. (2018). Using a Robust Metadata Management System to Accelerate Scientific Discovery at Extreme Scales. In *2018 IEEE/ACM 3rd International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS)*. IEEE.

[lcamtuf, 2019] lcamtuf (2019). AFL. http://lcamtuf.coredump.cx/afl/. [Online; accessed August 16, 2019].

[Michael Straßberger, 2019] Michael Straßberger (2019). Structured metadata for the JULEA storage framework. Online `https://wr.informatik.uni-hamburg.de/_media/research:theses:michael_strassberger_structured_metadata_for_the_julea_storage_framework.pdf`.

[MongoDB, 2019] MongoDB, I. (2019). MongoDB. https://www.mongodb.com. [Online; accessed September 13, 2019].

[Olgasnezh, 2017] Olgasnezh (2017). hdf5-vol-sqlite-plugin. https://github.com/Olgasnezh/hdf5-vol-sqlite-plugin. [Online; accessed September 13, 2019].

[Parker-Wood et al., 2014] Parker-Wood, A., Long, D. D. E., Miller, E., Rigaux, P., and Isaacson, A. (2014). A file by any other name. In *Proceedings of International Conference on Systems and Storage - SYSTOR 2014*. ACM Press.

[Parker-Wood et al., 2012] Parker-Wood, A., Tunkelang, D., Seltzer, M., Miller, E. L., and Long, D. D. E. (2012). Making Sense of File Systems Through Provenance and Rich Metadata. Technical Report UCSC-SSRC-12-01, University of California, Santa Cruz.

[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`.

[Rowan, 2019] Rowan, R. (2019). Adopting HDF5 for Simulation Data in EDEM Software. https://www.hdfgroup.org/2019/08/adopting-hdf5-for-simulation-data-in-edem-software/. [Online; accessed September 11, 2019].

[Weil et al., 2004] Weil, S. A., Pollack, K. T., Brandt, S. A., and Miller, E. L. (2004). Dynamic Metadata Management for Petabyte-Scale File Systems. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, SC '04, pages 4–, Washington, DC, USA. IEEE Computer Society.

Appendices

List of Figures

1.1 Performance development in the TOP500 list 5

2.1 Typical Application Storage Hierarchy 12
2.2 The HDF5 File Structure 13
2.3 JULEA applications Structure 18

3.1 JULEA metadata Write 23
3.2 HDF5 VOL Plugin ER-Diagram 33

5.1 HDF5 synthetic metadata benchmarks 45
5.2 HDF5 synthetic data benchmarks 46
5.3 Enzo benchmarks 47
5.4 Benchmarks of JULEA DB backends 50
    a JULEA synthetic database benchmarks using 1 process 50
    b JULEA synthetic database benchmarks using 6 processes 50
5.5 Benchmarks of JULEA DB backends 52
    a JULEA synthetic database benchmarks using MySQL 52
    b JULEA synthetic database benchmarks using SQLite 52
5.6 Benchmarks of JULEA DB backends 53
    a JULEA synthetic database benchmarks using MySQL 53
    b JULEA synthetic database benchmarks using SQLite 53
5.7 Benchmarks of JULEA DB backends 54
    a JULEA synthetic database benchmarks using MySQL 54
    b JULEA synthetic database benchmarks using SQLite 54
5.8 Benchmarks of JULEA DB backends 56
    a JULEA synthetic database benchmarks using MySQL 56
    b JULEA synthetic database benchmarks using SQLite 56
5.9 HDF5 benchmarks querying data across multiple files 57

List of Listings

2.1 HDF5 usage example 15

3.1 JULEA backend interface 21
3.2 JULEA DB client-side `JDBSchema` 25
3.3 JULEA DB client-side `JDBSelector` 26
3.4 JULEA DB client-side complex `JDBSelector` query 27
3.5 JULEA DB client-side `JDBEntry` 27
3.6 JULEA DB client-side `JDBIterator` 28
3.7 HDF5 VOL Plugin example code 34

4.1 JULEA `JBackendOperation` interface to simplify transfer of data over the network 36
4.2 JULEA backend SQL interface 37
4.3 JULEA DB client interface example showing the creation and loading of a `JDBSchema` 40
4.4 JULEA DB client interface example showing the creation and insertion of a `JDBEntry` 42

List of Tables

3.1 HDF5_file defined on behalf of the HDF5 VOL Plugin plugin 29
3.2 HDF5_link defined on behalf of the HDF5 VOL Plugin plugin 30
3.3 HDF5_datatype defined on behalf of the HDF5 VOL Plugin plugin 31
3.4 HDF5_space defined on behalf of the HDF5 VOL Plugin plugin 31
3.5 HDF5_space_header defined on behalf of the HDF5 VOL Plugin plugin . 31
3.6 HDF5_dataset defined on behalf of the HDF5 VOL Plugin plugin 32

5.1 Theoretical maximum performance of each operation. All values are displayed in operations per second. The columns show the impact of different field counts defined by the JDBSchema. The rows show the different functions which send data over the network. 49

List of Abbreviations

ACID Atomicity, Consistency, Isolation und Durability

ADIOS2 Adaptable Input/Output System

AFL American Fuzzy Lop

API Application Programming Interface

AVI Audio Video Interleave

BP4 binary-packed

BSON Binary JSON

CoSEMoS Coupled Storage System for Efficient Management of Self-Describing Data Formats

CPU Central Processing Unit

DDL Data Definition Language

Fuse Filesystem in Userspace

HDD Hard Disk Drive

HDF5 Hierarchical Data Format

H5MD HDF5 for molecular data

HPC High-Performance Computing

HTML Hypertext Markup Language

IO Input/Output

JPEG Joint Photographic Experts Group

JSON JavaScript Object Notation

JULEA DB JULEA Database

JULEA KV JULEA Key Value Store

JULEA OBJ JULEA Object Store

MPEG Moving Picture Experts Group

MPI Message Passing Interface

NetCDF Network Common Data Format

noSQL non relational SQL

NVRAM Non-Volatile Random-Access Memory

PNG Portable Network Graphics

POSIX Portable Operating System Interface

PostgreSQL PostgreSQL

RAM Random-Access Memory

SDDF Self-Describing Data Format

SQL Structured Query Language

SSD Solid-State Drive

URL Uniform Resource Locator

VOL Virtual Object Layer

XML Extensible Markup Language

ZFS Zettabyte File System

Eidesstattliche Versicherung

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

Ort, Datum | Unterschrift

Veröffentlichung

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

Ort, Datum | Unterschrift