Bachelor Thesis

Performance Assessment of Metadata Management with Different Databases in a FUSE File System

Martin Barthel
martin.barthel@st.ovgu.de

April 3, 2023

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

Second Reviewer:
Michael Blesel

Supervisor:
Jun.-Prof. Dr. Michael Kuhn and Michael Blesel

Abstract

File systems not only hold the information inside a file but also information describing the entry, known as metadata. As several use cases like HPC raise the desire to make use of existing technologies able to be accessed from multiple machines and capable of managing large amounts of data, databases are a likely candidate as metadata storage solution. However, many categories of databases have been implemented over the years. Examples are key value stores, relational databases, graph databases and triple stores [Meier and Kaufmann, 2019]. This prompts the question which class should be picked. To provide arguments for that decision, an existing file system implementation is picked. This is provided by julea-fuse which already uses key value stores [Kuhn, 2017]. julea-fuse is located within the JULEA-Framework. As the other metadata backend provided by it are relational databases, this is the obvious choice for the second database type to be evaluated. julea-fuse binds to the fuse library to support existing POSIX compliant applications while running in user space. To be able to support both backends via highly similar code an interface is created to hide their specifics. It is then evaluated whether julea-fuse completes common file system requests faster when using the key value store or the relational database. This is done via mdbench [kofemann et al., 2022]. There the key value store appears to be faster than the relational database across the evaluated methods. Hence the implied takeaway is that with the used file system structure, key value stores are the better storage backends for file system metadata in terms of latency.

Contents

Chapter 1.

Introduction

In this chapter, first it is pointed out why the comparison of the performance of relational databases and key value stores as metadata backend is required. Afterwards the structure of this thesis is explained.

1.1. Motivation

As processor development followed Moore's law for many years their capabilities have grown[Eeckhout, 2017]. Although some imply that it is probably slowing down. However, it has lead to increasingly capable chips being merged into supercomputers. The supercomputer with the most floating point operations per second as in November 2022, Frontier, is able to access Orion [Top500, 2022] [OLCF, 2023]. This is a file system fitted with 679 Petabytes of hard drives. However, file systems also need to keep data to manage the file content, so called metadata. Regarding that, [Alam et al., 2011] speak of a "metadata wall". They point out that in their system, metadata operations per second do not increase with more concurrent accesses as desired. Others encountered similar problems. [Luu et al., 2015] evaluated 1,080,262 runs of different applications from three supercomputers. The overall duration of the operations on file content and those on metadata were compared in their case. Metadata interactions outweighed the other type in 2 out of 5 of the runs. Before that, [Carns et al., 2011] analyzed 10 projects on the Intrepid supercomputer. During this, in 6 projects calls metadata related took longer than functions on file content. In one instance 95% of the duration of the file system calls is generated by metadata operations. As the authors note, some of the imbalance could be explained by counting a function which may issues content manipulation as metadata only. However still a lot of time is required for metadata operations. This probably is undesired as mostly the actual file content is what is of interest by the user.

Therefore improvement to the costs of metadata operations is desirable, especially on scale. Systems that handle large amounts of data entries everyday are for example databases. MySQL for instance is able to handle at least 5 billion entries in an entire instance [MySQL, 2023c]. MongoDB as NoSQL-database has a deployment with overall 200 billion entries [mongodb, 2023]. Those typically provide user space libraries which become accessible via fuse [Vangoor et al., 2017]. As referenced in Chapter 4, file systems which use databases for their metadata already exist. However this thesis is concerned with adding a second database type to an existing fuse file system. Namely the file system concerned - julea-fuse - already uses key value stores as backend for the information it saves additionally to the file content. In this thesis a way of storing this data in a relational database is added. By doing this, the different performances of both

databases in terms of latency will become clear. Hence future file system designers can pick a storage solution which suits their needs best. The database systems will be evaluated by comparing their performances at different tasks. Therefore it becomes apparent how much the time needed deviates between the different file operations. It is found that requests using key value stores consistently complete faster than those relying on relational databases. This might be influenced by the file system layout chosen

1.2. Structure of the Thesis

In order to bring this information across, first concepts of classical file systems will be discussed . Furthermore an important interface - POSIX - will be described and the fuse library which is used to export it to the user will be broken down. As there are two database types available through the JULEA-Framework, both the aims of the databases and the structure of the framework is required to be clarified. After that the design underlying the changes to the fuse library will be laid out. During this, especially the metadata collected is of interest. Before that, the general changes will be summarized. Then it is discussed how an abstraction was introduced in order to hide the specific details of the implementation. Moreover the consequences of the new metadata are discussed. Of special interest is the pathname resolution and consequently the mapping of file names to objects. After this, some file systems which also use databases for metadata storage are discussed. Consequently, operations on files and directories are compared between different databases and database types. Afterwards a suggestion on which database technology to use for file system metadata is given. Finally aspects of file system metadata management in databases which are worthwile evaluation are presented.

Summary

As metadata performance often holds back file systems in a HPC context, ways of optimizing its organization are necessary. This can be done for example by utilizing existing systems for metadata management such as relational databases and key value stores. In the following thesis a file system is described which can utilize both types and hence is able to be used to compare their performance. Its layout and the evaluation is discussed following the depicted structure.

Chapter 2.

Background

In this chapter, an overview is given on the components in a file system. Then POSIX as an interface to manipulate those is discussed. As being the way to provide a custom file system used in thesis, the internal structure of fuse is explained. This thesis compares relational databases with key value databases. Because of that information about their interfaces and some internals are given. Finally JULEA provides the framework to interact with the two types of databases

2.1. File Systems

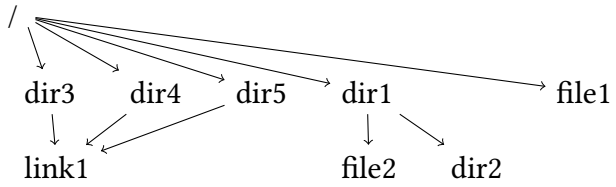


Figure 2.1.: example organization of file system entries

At some point most software uses a file system in some way for various reasons [Andrew S et al., 2015]. A File System keeps data needed beyond the execution of a program. Furthermore data contained in it exceeds usually the size allowed by the RAM. Its entries are not bound to one running program.

When using Linux, files as the entities in a file system fall under multiple distinct categories [Andrew S et al., 2015]. Files of every type are identified by a name.

Regular file refers to an entity the actual data is written to or retrieved from [Andrew S et al., 2015]. Examples for those are `file1` and `file2` in Figure 2.1. Their content needs only to be understood by the applications using the file. This is not a concern of the file system. Besides storing the data inside a file, file systems also gather information beyond that for every entry. This metadata incorporates for example the length of a file. Other possible settings found there are its visibility or the date its generation happened. As file systems usually reside on hard drives, the so called index node also features information concerning the distribution of the files content. Therefore it possesses a catalog of disk blocks containing the file content. It is also able to reference several others of such catalogs. This is deployed in case no entries are left in the first one.

Index nodes exist for every file not just for regular files [Andrew S et al., 2015].

File system entries which arrange others are called **directorie**s [Andrew S et al., 2015]. A directory might itself lay inside a higher directory like `dir2` is in `dir1` as pictured in Figure 2.1. Data about included files lays in the directories data structure. In the example `file2` and `dir4` are registered in `dir1`'s list. This is used for resolving paths. A slash splits paths into file system entry names. By choosing one entry after another the desired file is arrived at upon the last part. The path `/dir1/dir2` is processed by first getting the entry of `dir1` in `/`. After that `dir2` can be located in the entry list of `dir1`. As this is the last section of the path, the resolution has finished then.

All in all, this results in a directory tree as depicted in Figure 2.1[Andrew S et al., 2015].

Another component of file systems are links [Andrew S et al., 2015]. Through **hard links** a file can be referenced by different paths. In Figure 2.1, `link1` represents a hard link. Through that `/dir3/link1`, `/dir4/link1` and `/dir5/link1` point to the same file. In contrast a **symbolic link** stores its actual target in itself which is then resolved for file operations on the link.

Other categories like *character special files* or *block special files* exist as well [Andrew S et al., 2015].

2.2. POSIX

`POSIX` is spelled out as "Portable Operating System Interface" [IEEE and Group, 2018]. By `POSIX` method signatures and properties are laid out which can be expected by programs abiding by them. Thereby it provides a guideline for software like a file system wishing to provide compatibility with existing applications. While doing so, it is geared towards `UNIX`. `POSIX` contains the codes that are passed back by methods if a problem is encountered and which are appropriate for a method. For example they define what happens if a file does not exist when *chmod* is invoked. The implementation of the standard does not in every case relieve of recompilation. It also codifies aspects of the terminal and certain applications.

From a file system perspective, general aspects of its interface are laid out [IEEE and Group, 2018]. For instance the grammar which path names follow is set. Moreover objects in a file system are defined, namely file, link and directory. Together those should form a directed graph according to `POSIX`. The pathname resolution mechanism and its properties are also defined. It sets how symbolic links are handled and the steps when loops are known to be created by them. `POSIX` lays out the behavior of methods often applied on files. For example *open* is defined. Some methods can only be called after *open* [Andrew S et al., 2015]. That is because on running the method necessary information for e.g. *read* is cached. *creat* behaves similarly except it being used for new files. *write* puts new content into a regular file or alters it. To load this from the file, *read* is available. *chmod* and *chown* impact the permissions of user accounts on the file. If a different path than the old one should point to it, this can be accomplished by *rename*. *unlink* gets rid of no longer needed files, it could be seen as the opposite of *creat*. *fstat* provides the `metadata` as described in Table 2.1 of the file to the caller. Special for directory files are *mkdir*, *readdir* and *rmdir*. *mkdir* and *rmdir* are like *creat* and *unlink* are for other files. If a program needs to list all directory contents it can use *readdir*.

In addition, it describes the metadata expected by it [IEEE and Group, 2018]. General metadata components can be gathered from `sys/stat.h`. An overview of those can be seen in Table 2.1. In Listing 2.1 an example output of stat is visible providing possible values for stat's entries. *st_dev* defines a selector of the medium concerned. As seen in line 3, `example` lies on `0,38`.

entry
st_dev
st_ino
st_mode
st_nlink
st_uid
st_gid
st_rdev
st_size
st_atim
st_mtim
st_ctim
st_blksize
st_blocks

Table 2.1.: stat structure

To tell the entries apart the *st_ino* is used which is 933507 in Listing 2.1. This is bound to the current file system as it is not operating system wide valid. The category of the file system entry is defined in *st_mode*. For example metadata can be distinguished between directories and regular files by it. In Table 2.1 `example` is a regular file. Furthermore operations are approved or denied to happen by the operating system depending on *st_mode*. These can be given on multiple levels. The account the file belongs to as given by the number in *st_uid* might has different permissions than the group which is determined by *st_gid* is granted. `example` belongs to *uid* 1000, `martin`, and *gid* 10, `wheel`, as seen in line 4. Account `martin` is able to read, write and execute the file. Members of `wheel` can read and write it. The final level in *st_mode* defines the rights of any one else for the file. In Listing 2.1 those can view the contents. In case the metadata is assigned to a special file, *st_rdev* is required. *st_nlink* counts the references by different directory entries to this particular index node which is one as seen in line 3. *st_size* stores the amount of the data. The unit used is bytes. As seen in line 2, `example` incorporates 391 bytes of data. *st_blksize* and *st_blocks* both concern disc blocks. The first suggests an optimal length for those. By the latter the ones belonging to the file are counted. `example` covers 8 blocks. The temporal information collection at methods for files is defined. In contrast to the other values which are whole numbers the time is captured via a *timespec*. This C-struct consists of a number for nanoseconds and one for seconds. Three moments are relevant as metadata. The most recent retrieval of the file content is written to *st_atim*. If alterations occurred this is stored in *st_mtim*. *st_ctim* changes for example when *chmod* or *chown* are called. Additional to the metadata contents required by POSIX, Table 2.1 also contains the birth and the context of the file as seen in line 5 and 9.

```
1   File: example
2   Size: 391         Blocks: 8          IO Block: 4096   regular
      ↪ file
3 Device: 0,38    Inode: 933507      Links: 1
4 Access: (0764/-rwxrw-r--)  Uid: ( 1000/  martin)   Gid: (   10/
    ↪ wheel)
5 Context: unconfined_u:object_r:user_home_t:s0
6 Access: 2022-01-10 11:25:16.103340328 +0100
7 Modify: 2022-01-10 10:24:56.278402473 +0100
```

```
Change: 2022-01-10 12:26:13.475160489 +0100
 Birth: 2022-01-10 10:24:56.278402473 +0100
```

Listing 2.1: output of running `stat example` in bash

2.3. Fuse

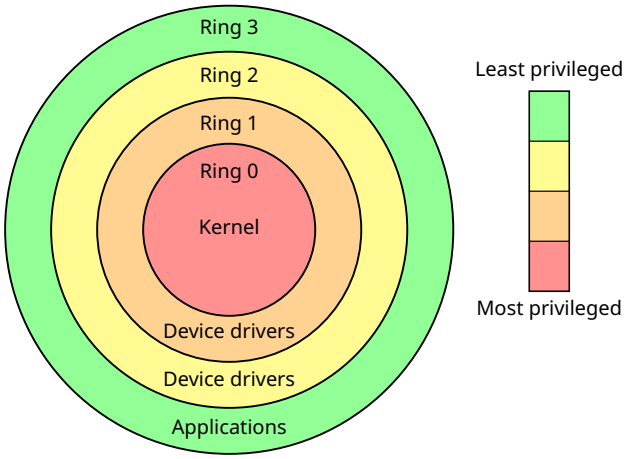


Figure 2.2.: Modes for code to run on[Hertzsprung, 2007]

Classic file systems under Linux like Ext4 bind to VFS [Andrew S et al., 2015]. In order to enable that, the VFS Interface needs to be implemented. If user space software uses a file it emits an interrupt. In the terms of Figure 2.2 it changes from the outermost layer to the innermost layer. In the kernel, the file's file system is looked up. Afterwards, the operating system executes the method registered for this type of interaction. VFS and the file system usually respond to interrupts staying in ring 0. Afterwards the caller in the outer layer in Figure 2.2 proceeds with restricted hardware access again.

Fuse however enables programmers to create a file system that runs in user mode [Vangoor et al., 2017]. Hence many hassles are taken out of the way for writing new file systems. Developers are used to the environment. In case of a fault in the implementation the operating system keeps running. It only effects the file system. Therefore it can be easily run again. Contributing to another benefit of fuse is that interactions with third party software usually occur in ring 3. Therefore their use is possible via fuse whereas for example binding to libjulea could be impossible in the kernel. Additionally on the positive side, it is possible for accounts other then the superuser to run a fuse program [fuse, 2023b][ubuntusers, 2021]. Therefore file systems become usable for them without contacting an admin. A major disadvantage of the usage of fuse is that context switches are costly in terms of time. Implementations inside the kernel can avoid that. In order to not be implemented in the kernel, fuse is split into two parts: one that runs in ring 0 and one running in ring 3. The first piece of software passes an interface to VFS. The second program handles commands it receives from the first. The communication is carried out via a block device. It is referenced as /dev/block.

In order to complete an operation in a fuse file system, the following steps are needed [Vangoor et al., 2017]: First an software interrupt is triggered from user land. In Figure 2.3 this is done by the ls program via libc. In the kernel the Virtual File System passes the call to the

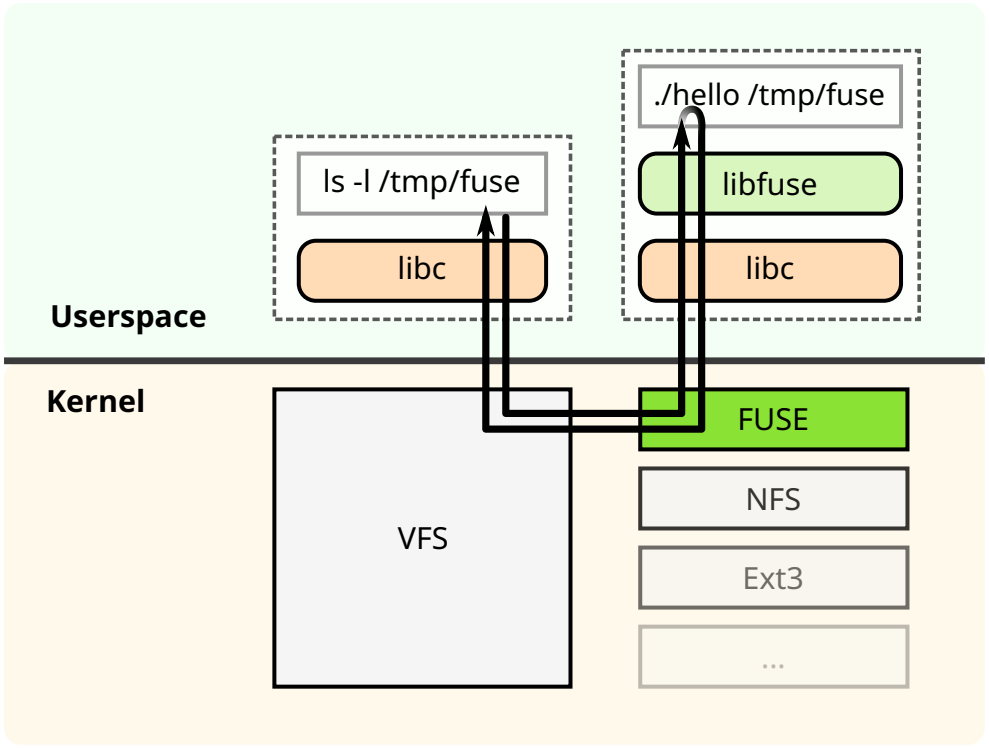


Figure 2.3.: example call in fuse[Sven and ElementW, 2019]

fuse module as this is the file system responsible for the file /tmp/fuse. Synchronous calls are enqueued in a FIFO list called pending, other calls go to background before this happens. The directory contents are read synchronously, so it is inserted in pending. After that, /dev/fuse is written to from the kernel module. It is marked in green in Figure 2.3. Other mounted file systems in this graphic appear to be NFS and Ext3. Those might be called by the VFS for other files in different locations. In /dev/fuse, sequence numbers distinguish the separate calls. In a call the node ID maps it to an index node. Furthermore the call changes into the processing list. At some point in time the instruction in fuses format is picked up by the fuse daemon. The daemon is called libfuse and marked light green in Figure 2.3. It interacts with /dev/fuse via libc. /dev/fuse can be seen as the means to pass the border between Kernel and Userspace in Figure 2.3. The fuse daemon then runs the implementation of the method the specific file system has passed to it. In the example the function to list the directory contents with their metadata. The concurrent handling of calls is possible, for example to list another directory at the same time. If enabled a single call is assigned per thread. Afterwards it sends the information of completion of the call with the specific sequence number via /dev/fuse to the kernel module as it is symbolized by the returning arrow. The fuse module can then dequeue the call from the processing list. Additionally to those lists it keeps others. From those, interrupt is relevant if a calls execution is obsolete. Finally the kernel can return to user space and the calling program can continue. In Figure 2.3 the VFS returns from the call to FUSE. In userspace the function provided by libc returns. Therefore, ls can proceed to list the contents of /tmp/fuse on the terminal interface.

Programmers may choose from two interfaces [Vangoor et al., 2017][libfuse, 2023b][libfuse, 2023a]. The first one, *fuse_operations* , provides the path as string or an integer - the file handle - to identify the file the operation should manipulate. It itself relies on *file_lowlevel_ops* . *file_-lowlevel_ops* distinguishes between files by index nodes.

2.4. Databases

2.4.1. Key Value Store

In a key value store entries are distinguished by keys [Wiese, 2015]. The other entry component is the value. Entries can be inserted into the database. Then the content can be queried. The last standard procedure is deleting the entry. The key value store is not concerned with the insides of the value. It can be anything especially a char array. It is treated as raw chunk of data. The key value store is hence referenced as schemaless.

Key value stores come very close to document stores [Wiese, 2015]. In the latter, the data attached to the key follows a grammar like given by xml, json or bson.

In contrast to the following database technology multiple machines can be organized together for a database [Meier and Kaufmann, 2019]. Even their online status can change dynamically. The allocation to computers relies on consistent hashing. The data structure on one of these is then also geared towards the hash of the identifier of the value.

key	value

"/folder/example"	<pre>{
	"st_mode": 33268,
	"st_ino": 933507,
	"st_dev": 38,
	"st_nlink": 1,
	"st_uid": 1000,
	"st_gid": 10,
	"st_size": 391
	}</pre>

"/folder"	<pre>{
	"st_mode": 16877,
	"st_ino": 287,
	"st_dev": 38,
	"st_nlink": 1,
	"st_uid": 1000,
	"st_gid": 1000,
	"entries": [933507]
	}</pre>

Table 2.2.: Two example entries in an key value store. example is referenced via "/folder/example" and folder's value can be accessed via "/folder". Json is used for encoding the values. In the graphic, `example` has a st_size entry whereas this is skipped in `folder`. In contrast, `folder` provides a list of entries in the value "entries".

2.4.2. Relational Databases

Another class of databases are relational databases [Meier and Kaufmann, 2019]. The organization of it differs form key value stores. A similar concept to the key value pair in the previous

database type is represented by the tuple [Saake, 2018]. This is assigned to a table respectively a relation. In contrast to key value stores, what makes up an entry is clearly defined beforehand. This is carried out on a per relation basis. Each of the relation's tuples exhibits the attributes given in this predefined schema. One or more relations are collected within an instance. An interaction with relations from the outside is usually carried out by formulating the desired instructions in SQL. Another key aspect of both this database class and SQL is the relational algebra. It contains unary and binary functions. Their arguments and results are collections of entries as suggested by the name of the algebra. By that a table with regular files in a specific directory can be generated from a table containing all files like *regular_files* in Table 2.3 via `"SELECT * FROM regular_files WHERE folder=287;"`. Another example for relational algebra would be to only output name, group and owner of entries via `"SELECT st_name, st_gid, st_uid FROM regular_files;"`. Moreover from relational algebra a single list might be generated from a list of directories and a list of regular files inside them with `"SELECT * FROM regular_files JOIN directories ON regular_files.folder=directories.st_ino;"`. These functions are referred to as selection, projection and join. All in all, when retrieving data from relational databases a much richer interface is provided then in key value stores. Like key value stores, relational databases provide several possibilities to change content in an database instance. In contrast to the former those instructions can be directed at more then one entry. The database can be instructed to remove rows. This can happen e.g. via `"DELETE FROM regular_files WHERE st_name='example';"`. Of course the user was able to add those rows to the database before via the sql interface provided with statements such as `"INSERT INTO directories VALUES ('folder',287,169877,1,1000,1000,0);"`. Information within rows can be altered hence not needing to rewrite the entire object. An example for that is `"UPDATE directories SET st_name='dir' WHERE st_ino=287;"`. It only sets the st_ino to 287. Relational databases assure the fulfillment of certain points to the user. Those are usually summarized as Atomicity, Consistency, Isolation and Durability. The ACID stands for this. Those are directed towards transactions. Through them, after executing the latter the instance does not show remains of a partial, failed execution and still conforms to given rules. The database hides the effects of concurrent access from the user and queries after the execution keep returning its state. B-trees usually lead to a desired tuple within relational databases. The instance generates them usually for indizes. An index could be the inode number as tuples might be often referred by it.

regular_files							
st_name	st_ino	st_mode	st_nlink	st_uid	st_gid	st_size	folder
example	933507	33268	1	1000	10	391	287

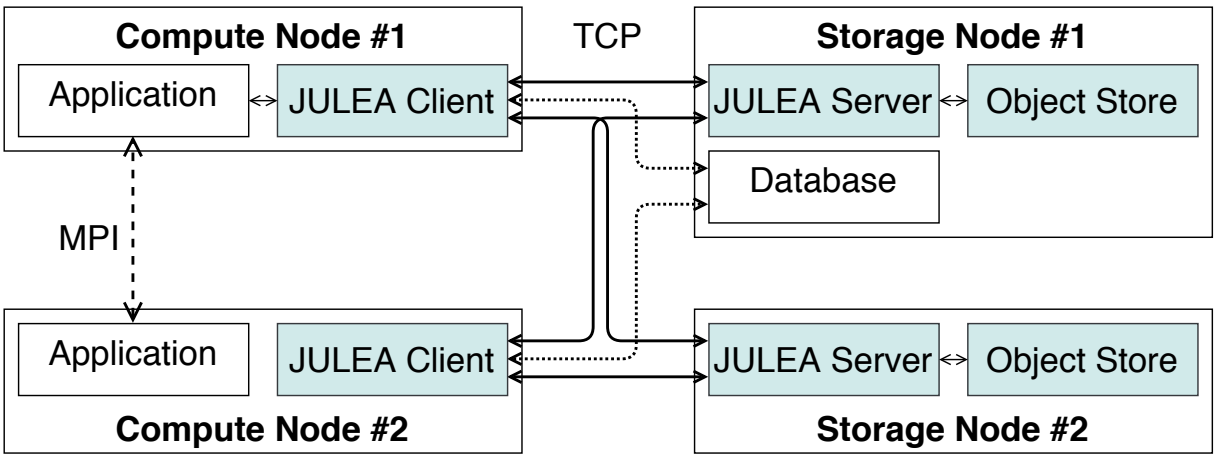
directories						
st_name	st_ino	st_mode	st_nlink	st_uid	st_gid	folder
folder	287	169877	1	1000	1000	0

Table 2.3.: example relational database

2.5. JULEA

JULEA facilitates several ways to manage persistent data while being aimed especially at HPC applications [Kuhn, 2017][Kuhn, 2015]. If new technology in this realm concerning file systems is to be assessed building everything from the ground up would consume many resources. JULEA tries to mitigate that. This is done by providing several often required parts. Those can then be used together as desired in the specific application.

JULEA ships with an application library, a server and certain targets which store the information[Kuhn, 2017][K The last part, the backend, provides an generic interface for the server to use key value stores, object stores or relational databases [Warnke, 2019]. An application using JULEA usually calls its library functions. This is pictured in Figure 2.4 on the left. There the application runs on Compute Node #1 and #2, on both the julea client offering the API for the program is present. The client connects with the JULEA server. In this case - as it is not a development environment - this is done over network using TCP. The server running on Storage Node #1 and #2 then calls functions provided by the appropriate backend. In Figure 2.4 it is the one for the object store. This then hands the query over to the actual object store. Typically on the backend side different devices either serve as key value, relational database or object store. As seen in Figure 2.4 the client connects directly to the database server on Storage Node #1, because relational databases are not accessed through the julea-server [Kira Duwe and Michael Kuhn, 2021]. In this case Storage Node #1 provides database and object store services whereas #2 only serves as object store. Its the applications choice how to pick a server to use for an entry if multiple of one kind exists. In Figure 2.4 the clients are able to select #1 or #2 for their objects. In the case of JItems, this can be done via utilizing JULEAs distributions.



[Kira Duwe and Michael Kuhn, 2021]

Figure 2.4.: JULEAs componenents

JULEA is able to alter the requirements its methods conform to while it is executed [Kuhn, 2015]. This is done by choosing the semantics parameters which are currently used. For example, the safety of the information can be set to "Network" or "Storage". By that it is assured that it has reached the given medium. JULEA ships with several semantics templates with already chosen options.

Method calls in JULEA are grouped into batches [Kuhn, 2015]. Thus the library can perform alterations on them. For example some requests might be united to one or there exists a better

sequence which decreases resource usage. Each of the batches can be executed with different semantics.

JULEA provides multiple APIs [Kuhn, 2015]. The main access mode consists of namespaces grouping collections containing items. In this concept items resemble files. They consist of a key value pair and a object store entry. The first contains the metadata and the latter is used for the actual file content. JULEA already provides a implementation of a fuse file system. This allows existing software to also store its data through JULEA without having to alter it.

Object and key value store can also both be directly accessed through the appropriate APIs [Kuhn, 2017]. Additionally to the former the interface for using relational databases is also tailored for managing metadata [Warnke, 2019].

It consists of multiple structures to interact with the database [Warnke, 2019]. Columns can be defined via JDBSchema. It manages a single chart of the relational database. The JDBSelector allows choosing a set of rows which match the given criteria. A JDBIterator can then go over each of those. To put a single row into the database, JDBEntry is used. Futhermore it is used for updates.

An object is manipulated with methods using a JObject structure [Kuhn, 2017]. It is identified by a unique string. Its contents can be accessed via specifying a position and length, mapping relatively close to POSIX's functionality [IEEE and Group, 2018].

As mentioned above, JULEA can utilize various databases and object stores as storage for these three interfaces [Kuhn, 2017]. In case of key value stores those are for example sqlite and mongodb. Posix can act as an object store. Examples of relational databases are represented by MySQL and sqlite.

Summary

As described previously file systems consist of multiple entry types. Operations on them and their metadata is standardized by POSIX. Fuse enables an easy way to implement those functions as normal application. Both key value stores and relational databases can act as metadata storage. To access them in a implementation agnostic manner, the interface provided in the JULEA framework for HPC is used.

Chapter 3.

Implementation

In this chapter, the implementation of julea-fuse is described. The context in which julea-fuse operates is pointed out.

3.1. Added Functionality

As mentioned before, an fuse implementation using JULEA already exists in the framework [Kuhn, 2017]. This previous version was written for fuse2 and was already updated to fuse3. However, besides fuse, JULEA moved forward so that it now provides an relational database Interface additionally to the API for using key value stores which was previously the only one intended for metadata. In order to react to these changes, this implementation extends the existing one in order to also support relational databases. This is done with support by the C preprocessor. *DATABASE_METADATA* is used to switch between relational database and key value store as metadata storage in fuse. If it is true, the sql database version is compiled. Otherwise the user obtains the key value version. In Table 3.1 and Table 3.2, the operation

posix functions	
VFS	
fuse module	
julea-fuse	
libjulea	
julea-server	
object store	key value store

Table 3.1.: the key value version

posix functions	
VFS	
fuse module	
julea-fuse	
libjulea	
julea-server	
object store	relational database

Table 3.2.: the relational database version

travels from top to bottom. It is first being initiated via a call to a POSIX function. It travels through the layers as described in Section 2.3 until it reaches JULEAs fuse implementation called julea-fuse and which is marked with light gray. julea-fuse then calls JULEAs library functions and hence communicates by that with one or more julea servers. The functionality to also use relational databases for metadata is added by this thesis. The relational database interface used was designed by [Warnke, 2019]. The added metadata target is highlighted gray in Table 3.2. Additional to that, julea-fuse is changed as needed in order to use two different metadata backends. In order to accomplish that, an abstraction is introduced at this layer as described in Section 3.3 to move the database specific code into separate files. Futhermore julea-fuse was extended if it seemed helpful in order to increase compatibility with POSIX. This allows for example that the performance of the implementation is measured by existing tools like mdbench which measure e.g. *chmod* that only existed as a stub in the previous version [kofemann et al., 2022]. This would lead to invalid results. By implementing these methods, the two versions become more comparable.

3.2. Metadata

key	value
path	size
	name
	file
	size
	time

Table 3.3.: metadata structure of the key value store in the version before this thesis, created after the source of [Kuhn, 2017]

schema	entries
posix_metadata	path
	size
	owner
	group
	mode
	object
	atime_n
	atime_s
	mtime_n
	mtime_s
	ctime_n
	ctime_s

key	value
path	size
	owner
	group
	mode
	object
	atime_n
	atime_s
	mtime_n
	mtime_s
	ctime_n
	ctime_s

Table 3.4.: metadata structure of the relational database and the key value database

As mentioned previously, JULEA already provides an a fuse implementation [Kuhn, 2017]. It uses key value stores for managing its metadata. The metadata is encoded as a bson document. The document contained the name as string, a boolean which is set if its a file and not a directory, the size and the time it was created. This is depicted in Table 3.3.

For the next fuse implementation, the key value store implementation was changed to make use of a struct. The direct implementation has some benefits, such as not needing to make calls to the bson library but also becomes machine dependent[Warnke, 2019]. Beneficial for bson is that it also fits to the framework as it is utilized in other components[Kuhn, 2017]. The relational database version of the implementation transmits data as bson. Thus if there is a drawback in performance due bson usage it is in any case present in this version.

As not using bson comes with drawbacks an different implementation was added, where the metadata again is converted to a bson document on writes and the values are retrieved from it afterwards. Therefore the users are able to choose at compile time whether or not they want to use structs or bsons for their key value metadata storage. Bson also add some tolerance for changes in the metadata format compared to structs which could break if an attribute is added between two values and hence their relative address changes. This can be accomplished with e.g. *bson_iter_find* which goes to the desired value. This could be helpful in file systems where there are multiple versions of a file system used. For example if the newer system has an additional value, the old system could still use data of the new system. Vice versa the new system would need to be programmed to expect key value pairs which miss attributes. However, as the julea-fuse version introduced in this thesis assumes the bson documents to contain all required variables. Hence using metadata in the format of older fuse versions is not possible.

In case of the relational database, the metadata is stored in a table *metadata*. It consists of the columns *path*, *file*, *size*, *owner*, *group*, *mode*, *atime_n*, *atime_s*, *mtime_n*, *mtime_s*, *ctime_n* and *ctime_s*. *path* is a string. All other types are integers. For every gathered time two columns are used. This is because a time consists of two variables according to [IEEE and Group, 2018]. Namely those are nanoseconds and seconds. Each time column is followed by an *_n* or an *_s*, determining which unit is used.

As the implementation uses the path to access files, it makes sense to define it as a index. By doing so, retrieval and manipulation of metadata entries becomes more optimal [Warnke, 2019]. Putting new files in the table receives some latency burden.

The key value store implementation stores the same information as the database table except for *path* as this is the key which references the values as it has been before this thesis [Kuhn, 2017]. By enlarging the collected metadata, the obtained results might become more comparable to different works as this is quite similar to the ones required by POSIX [IEEE and Group, 2018].

The current implementation already calls the necessary function to run fuse at start up[Kuhn, 2017]. However, the relational database needs to conform to a schema. Hence it needs to be retrieved from the database as described in [Warnke, 2019]. On first start the database is initialized with it. On every other occasion it is retrieved from it.

3.3. Abstraction

During the implementation of the methods, it came apparent that the functionality used is very similar. For example in order to change a metadata parameter, the value needs to be loaded from the key value store [Kuhn, 2017]. After that it is manipulated and written back. When implementing the logic for the database, similar repetitions occur. In this case, an entry and a

selector needs to be created. Then, an update needs to be performed. The application logic shared by those methods seems to be very similar.

This can be shown at the *chmod* method. In case of the key value backend a JKV named *kv* is created in line 6 using the interfaces from [Kuhn, 2017]. It lays in the posix namespace. Its key is the path passed to *chmod*. The operation to load - *j_kv_get* - it is then put in the batch in line 7, which is executed afterwards in the next line. This would fail if the file does not exist. After that, the mode is set in the struct in line 10. Then the update function for the key value pair - *j_kv_update* - is inserted in the batch at line 11 which again is executed in the next line. Afterwards, if both batches do not run into errors, the return value can be set to 0 to indicate success in the if-statement beginning with line 12.

```
1  int
2  jfs_chmod(char const* path, mode_t mode, struct fuse_file_info*
       ↪ fi)
3  {
4      //...
5      batch=j_batch_new_for_template(J_SEMANTICS_TEMPLATE_POSIX);
6      kv=j_kv_new("posix",path);
7      j_kv_get(kv,&val,&len,batch);
8      if(j_batch_execute(batch)){
9          file_metadata* metadata=(file_metadata*)val;
10         metadata->mode=mode;
11         j_kv_put(kv,val,len,g_free,batch);
12         if (j_batch_execute(batch))
13         {
14             //...
15 }
```

Listing 3.1: chmod with key value backend

In case of the relational database backend we also need a batch to perform the update on the entry. First, we need to specify which entry we want to operate on [Warnke, 2019]. This is done via a *JDBSelector* in line 5 and 6 which receives the path as the field the selection is based on via *j_db_selector_add_field*. The latter happens in line 7 to 8. This can be compared to the *JKV* structure in the case of the key value store. Contrary to the *JKV* the selector can represent multiple entries. In this implementation we can skip loading the metadata and only run the update. For an update, a *JDBEntry* is created in line 10. *j_db_entry_set_field* writes the new mode to the mode column in line 11. Finally the update assigned to the batch with *j_db_entry_update* in line 12 which is then executed in line 13.

```
1  int
2  jfs_chmod(char const* path, mode_t mode, struct fuse_file_info*
       ↪ fi)
3  {
4      //...
5      selector=j_db_selector_new(
6          db_schema,J_DB_SELECTOR_MODE_AND,NULL);
7      j_db_selector_add_field(
8          selector,"path",J_DB_SELECTOR_OPERATOR_EQ,path,-1,NULL);
```

```
9       batch=j_batch_new_for_template(J_SEMANTICS_TEMPLATE_POSIX);
10      entry=j_db_entry_new(db_schema,NULL);
11      j_db_entry_set_field(entry,"mode",&mode,-1,NULL);
12      j_db_entry_update(entry,selector,batch,NULL);
13      if (j_batch_execute(batch))
14      {
15          //...
16  }
```

Listing 3.2: chmod with relational database backend

In both cases the file the operations uses needs to be selected. In both cases, this is done by specifying the path. Also, the mode is written to a data structure. Moreover both updates require a batch. A major difference is that loading the metadata is not necessary in case of the relational database. Reading and altering values also takes place in all other methods concerning metadata. In fact, all methods inserted by julea-fuse into the *fuse_operations* struct deal with metadata [Kuhn, 2017].

In order to avoid this duplication a common interface was implemented which encapsulates the database specific parts and the always needed steps. It consists of *JFileMetadataOut*, *JFileMetadataIn*, *JFileSelector* and *JDirectoryIterator*. The *JFileSelector* represents either a *JKV* or *JDBSelector*. It is used to determine the file system entry the desired value is mapped to. The *JFileMetadataIn* and *JFileMetadataOut* store the actual metadata. The first one is loaded from the database, the second one is the entry which is used to update it. This distinction is not necessary in case of the key value store. However, the relational database returns an iterator from an executed select statement and therefor needs a *JDBEntry* to write to the database [Warnke, 2019]. In theory it might be possible to convert the bson document as it is contained in the iterator directly to an entry as can be found in the source code of [Kuhn, 2017]. However, this might break when JULEA's SQL implementation is updated as this is not part of the official interface. On the other hand, in case of key value stores the piece of data can be used both for getting and setting, so both *JFileMetadataIn* and *JFileMetadataOut* reference it. Functions can read metadata values like owner, group etc. from *JFileMetadataIn*. This structure is loaded from the database by specifying a *JDBSelector* for the file. When using the SQL interface, the batch is created internally. The value from the key value store is loaded using the batch passed to the function.

```
1  JFileMetadataIn* j_file_metadata_new(JFileSelector*, JBatch*);
2  JFileMetadataIn* j_file_metadata_new_load(JFileSelector*);
3  guint64 get_size(JFileMetadataIn*);
4  guint64 get_owner(JFileMetadataIn*);
5  guint64 get_group(JFileMetadataIn*);
6  gint32 get_mode(JFileMetadataIn*);
7  //...
```

Listing 3.3: JFileMetadataIn

On the other hand, methods which manipulate for example the creation time or other data act on *JFileMetadataOut* structures. This datatype can be initialized by converting *JFileMetadataIn* to it with *j_file_metadata_in_to_out*. As the update does not require to load the metadata in case of the relational database backend, *j_file_metadata_out_new_load_for_update* is introduced

to only do that in the key values case. Alternatively, when laying out a file it can be created from scratch. It provides appropriate methods to write it to the database. For write access both metadata interfaces use an external batch so the caller needs to provide one.

```
1  JFileMetadataOut* j_file_metadata_in_to_out(JFileMetadataIn*);
2  JFileMetadataOut* j_file_metadata_out_new_load(const char* path,
      ↪ JFileSelector* fs);
3  void j_file_metadata_write(JFileSelector*, JFileMetadataOut*,
      ↪ JBatch*);
4  void j_file_metadata_create(JFileSelector*, JFileMetadataOut*,
      ↪ JBatch*);
5  JFileMetadataOut* j_file_metadata_out_new(const char*);
6  void set_name(JFileMetadataOut*, char*);
7  void set_file(JFileMetadataOut*, gboolean);
8  void set_size(JFileMetadataOut*, guint64);
9  void set_owner(JFileMetadataOut*, guint64);
10 void set_group(JFileMetadataOut*, guint64);
11 void set_mode(JFileMetadataOut*, gint32);
12 //...
```

Listing 3.4: JFileMetadataOut

The final datatype provided is the *JDirectoryIterator*. It is needed to list the entries of directories. It uses the *JKVIterator* or the *JDBIterator* internally. The first one was already used in the original implementation provided by [Kuhn, 2017] and returns all entries with the dir name as prefix. The *JDBIterator* version works in a similar way to keep its behavior consistent with the previous one. All strings which are smaller than the first one outside the directory and the who follow alphabetically after the last string inside the directory.

```
1  JDirectoryIterator*
2  j_directory_iterator_new(const char* dirname);
3  gboolean
4  j_directory_iterator_next(JDirectoryIterator* dir_iter);
5  char*
6  j_directory_iterator_get(JDirectoryIterator* dir_iter);
```

Listing 3.5: JFileMetadataOut

With the interface introduced by this thesis *chmod* can be implemented as shown below. Again, a batch is created in line 5. Then a *JFileSelector* is created for the file given in *path* in line 6. As no metadata needs to be read *j_file_metadata_out_new_load_for_update* is run to only load data in case of a key value store is used. If this is successful, the mode can be set in line 9. Afterwards the update can be added to the batch and then run in line 10 to 11.

```
1  int
2  jfs_chmod(char const* path, mode_t mode, struct fuse_file_info*
      ↪ fi)
3  {
4      //...
5      batch=j_batch_new_for_template(J_SEMANTICS_TEMPLATE_POSIX);
6      fs=j_file_selector_new(path);
```

```
7       fe=j_file_metadata_out_new_load_for_update(path,fs);
8       if(fe){
9           set_mode(fe,mode);
10          j_file_metadata_write(fs,fe,batch);
11          if (j_batch_execute(batch))
12          {
13              //...
14  }
```

Listing 3.6: chmod with common interface

3.4. Synchronization

When a value in a key value store is altered a race condition can occur. If machine 0 alters the same value in the store as machine 1 before the update of the other arrives, e.g. machine 0 changes the mode and machine 1 the atime, one of the changes is not in the final version of the metadata value. This is also a obstacle for relational databases and is known as "lost update" [Saake, 2018]. In the previous version of julea-fuse as in [Kuhn, 2017] only the size value was modified. As in this thesis more metadata variables were added which could be changed after file creation the magnificence of the problem has grown as more variables were introduced where the described phenomenon can occur. However by adding a version for relational databases, there now exists an implementation were updates are only done on certain values and without fetching the entire value block. Relational databases are moreover designed to comply to ACID which requires them to avoid "lost updates" [Saake, 2018].

The authors of the framework pointed out, that this behavior also occurs at the hdf5 part as can be found in the source of [Kuhn, 2017]. According to them, it could be mitigated by using a kv-pair for each variable in the entry, which would increase the number of searches needed, or by providing a locking solution.

Inconsistent metadata however is probably still undesired by users. To circumvent this, files might only be accessed in non concurrent manner. As this is very broad there might be a desire to limit the operations so more clients can access a file. The following julea-fuse operations can be run in parallel on key value stores without missing changes to other variables: *creat*, *unlink*, *mkdir*, *rmdir*, *truncate*, *access* and *getattr*. The same is true for *read* after opening the file.

Other solutions exist as well, which could solve the problem. The user software could communicate within itself to determine who is allowed to access the file. A node could be assigned the task to determine the one allowed to access the metadata of a file exclusively [Van Steen and Tanenbaum, 2017]. Other possibilities of mutual exclusion could in theory be realized via the current storage framework. One possibility would be a token ring. In this case a token would be given to every machine in one cycle. The machine holding it could keep it if needed or put it to the next machine. This could be realized by having an entry per mount point or machine in the key value store or a separate relation for those in the relational database. There would then be a ever running process per julea-fuse mount who checks if its got the token or not. If the token is currently hold, it either enters the critical section. If done with that it could go writing the token to the entry of the next machine.

In key value stores, it would be possible to only keep track of the changes plus their sequential arrangement and not save the actual value as it is done commonly in distributed environments[Van Steen and Tanenbaum, 2017]. The client needing the data would then construct the metadata entry dynamically. This is comparable to state machine replication, but the collection of changes is stored more or less centrally and only processed by the fuse client if the metadata they alter is needed. Conflicts, like executing *chmod* as user A after a *chown* call set the owner to user B so the privileges are missing, would need to be resolved by the fuse client. This solution would also lead to many entries, which might could be compiled to actual entries from time to time by some always running process, so only this entry and updates after it would be kept in the store. Those entries could be accessed via an iterator, which would create a lot of overhead. This solution is comparable to journaling file systems [Andrew S et al., 2015].

Those options would probably increase the software's complexity tremendously, which would lead to for example difficulties for maintenance, adding new functionality or finding bugs, it was decided to evade this. It also is not necessarily a topic differentiating key value stores from relational databases when excluding the granularity discussion. It might be solved with future work.

3.5. Access Control

As mentioned above, functionality for both *chmod* and *chown* is added to julea-fuse by this thesis. Those are concerned with managing the rights of user accounts for a single file [IEEE and Group, 2018]. Additional the metadata required for them - owner, group number and the mode - is kept. If a file is copied from an existing POSIX file system it might bring these attributes which is another argument for adding them. When retrieved again from julea-fuse and put in a traditional file system, without those, the metadata is cut off or replaced with JULEAS's place holders. However, this fuse application still does not implement access control. Therefore it is only performed if fuse is instructed to let the operating system handle it [Vangoor et al., 2017]. To be checked, the os needs to retrieve the necessary information and after that the actual operation is run. Hence it could be already invalid. Therefor atomicity for those operations like in relational database as found in [Meier and Kaufmann, 2019] would be required.

JULEA as found in the source of [Kuhn, 2017] also does not check if an operation by an user is allowed or not. If requests to the JULEA server are made with an malicious intent, one could easily read or manipulate file system information. Hence, the current method of access control can only be used if there are no harmful actors on the systems. Measures which only exist on a layer above the client would need to be supported by additional measures like firewalls which limit communications to the JULEA server.

All in all, the implementation of file access data has not many benefits considering security. It increases conformity with POSIX, for example if files a transferred from another files systems this information is not lost.

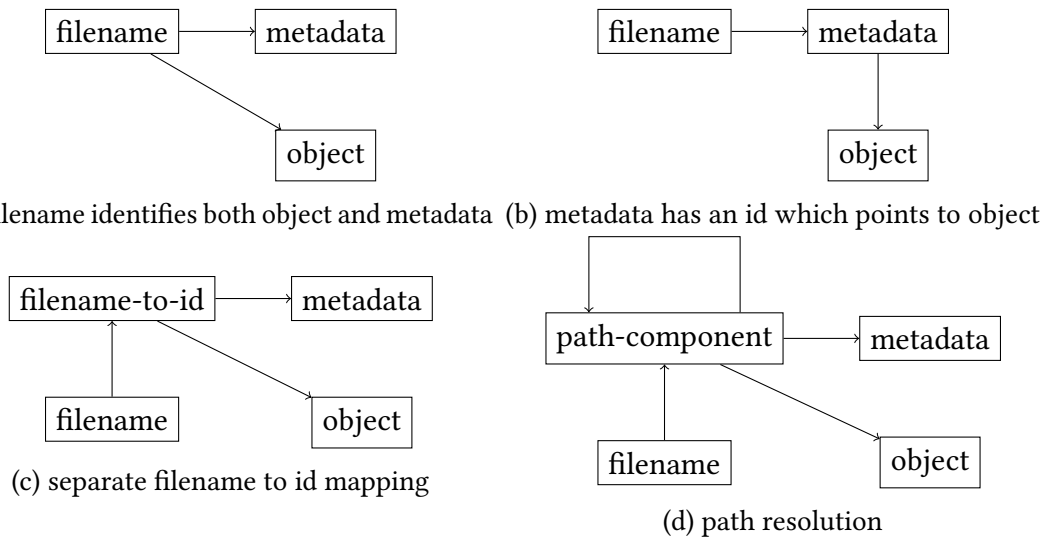
3.6. High Level Interface

As mentioned in section fuse Section 2.3 there are two levels a fuse implementation can operate on [Vangoor et al., 2017]. The fuse version of [Kuhn, 2017] chooses the high level interface. As mentioned before, the high level interface uses pathnames to select files. Given that the way of file look up remains the same as in [Kuhn, 2017], julea-fuse also still uses the high level functions. Then the node id is handled internally by fuse. This keeps the application logic from JULEA's point of view smaller. Thereby it is easier to understand and it contains less possibilities for errors. If julea-fuse is set to use 64-bit integers as identifiers, this advantage would vanish. Fuse's *fuse_lowlevel_ops* uses index node numbers as a function parameter [libfuse, 2023a]. Therefore *lookup* could retrieve the target of the file name it receives.

In contrast, simple integers might be hard to read without context. Integers could be used both as metadata and object names. The integer then would also be used as inode number returned by the lookup call in *fuse_lowlevel_ops*. As mentioned in linking, doing so comes with its own benefits and drawbacks. Additionally, the total number of files in julea-fuse would be limited to 2^{64} [fuse, 2023a].

Alternatively, the response to lookup could include a index node number unique to the client. Then the limit would only apply locally. The names would be still be human readable. It would require fuse to become stateless.

3.7. Pathname Mapping



(a) filename identifies both object and metadata (b) metadata has an id which points to object

(c) separate filename to id mapping

(d) path resolution

Figure 3.1.: lookup

In the previous version of julea-fuse in [Kuhn, 2017] metadata and objects are both referenced via their names. This is visualized in Figure 3.1a. However, common local file systems are usually organized as tree data structures [Andrew S et al., 2015] as described in Section 2.1. julea-fuse in contrast lists directory contents by selecting those with their name as prefix. This causes files to be efficiently accessible using the file name, which is passed to methods in fuse's highlevel API [Vangoor et al., 2017]. Various other authors choose a similar addressing scheme like [Tatebe et al., 2022].

If the file would need to be looked up by every part of its name as described in Section 2.1 the computational complexity would increase by a factor which is the number of parts in the file name. However some file systems with metadata databases use exactly this [Ren and Gibson, 2012]. As shown in Figure 3.1d the path components would contain the key to other path components if it were directories. If they are regular files, they would need to include the identifier of the metadata and the object store entry. This would be very inefficient as the process goes from one directory to another until the file is reached. To avoid that, the identifier of metadata and object can be saved every time a file is opened and through the lookup process could be reduced to its current complexity. On the side of benefits, that would make it possible to create hard links for both files and for directories. It would be the only version with links on directories as those can occur on every path resolution step [Andrew S et al., 2015]. In relational databases lookup over every path component might be achieved with a single request. In SQL, recursive queries can be used to get from one starting entry to the next entry according to it until the end is reached [Saake, 2018]. This is however not supported by the current interface [Warnke, 2019].

If a file is renamed, the key in the key value database can be changed easily by inserting the value elsewhere [Wiese, 2015]. This is also the case at the relational database entry by updating the path. The object store however does not provide a renaming method. An option for mitigation would be simply copying the object to the object with the new name. This would take a long time on large files, especially if there is a network connection between the machine running the copying operation and the one providing the object store. Additionally this might cause the file system to be in a inconsistent state where reads and writes could happen to both the new and old object while the operation is pending. In order to avoid that, a unique identifier for the object could be created. The SQL database interface already provides a way of generating one through its *_id* field [Warnke, 2019]. In case of the key value, this is however not possible. To mitigate this, the file name is hashed and it is looked up in the object store if there is already an object existing with the same id as name. If so a random number is added to the current id until a free object name is found. This of course adds overhead to the file creation especially if there are close to 2^{64} objects which maximum count of objects to the file handle size [fuse, 2023a] where the id is kept after *open*. However, creation only happens once per file. This new implementation causes an additional step per object access if the identifier is not cached locally in the file handle. In contrast to the previous possibility file content access would not be influenced, only the metadata access would fail if it is renamed. As renaming is so important that its part of mdbench [kofemann et al., 2022] this method was chosen. If files that are directories need to be renamed however difficulties occur as also pointed out by [Tatebe et al., 2022]. In this case an iteration over every path with it as prefix in the database occurs. Found metadata is updated to start with the new directory. Hard links for regular files could be realized in this version if there are separate metadata entries per link which would be updated every time the metadata of the file is changed, introducing a lot of overhead and increasing the chance that a value is not updated because an operation fails or the values become inconsistent due to concurrent updates by different clients. In case of the key value store a list with all other metadata entries would need to be kept in each one of them to update those to. Relational databases however also enable updating values in multiple entries [Saake, 2018] which would enable links via separate meta data copies, but also ship with overhead. Some other key value store file systems use special entries for hard links and relegate to them from the key value pair addressed with the path[Stender et al., 2010]. This however requires a conditional resolution in every method depending on whether its a hard link or not.

In order to avoid that, it would be possible to use an identifier also for metadata. In this case, there would also be an operation needed for retrieving the mapping from file name to id. The lookup would take place before every operation on metadata. This is pictured in Figure 3.1c. This would also enable hard links for regular files directly without traversing the file tree, special or multiple metadata entries. The operations for looking up the identifier are always necessary on key value stores. Relational databases provide the possibility to join tables [Saake, 2018]. Therefore two tables, one with the links and the identifier and one with the actual metadata, would enable the creation of links and reduce retrieving the metadata into one operation. As there is currently no way for joining schemas provided trough the interface it is not feasible [Kuhn, 2017]. This would clearly be an advantage on the side of relational databases, a thesis concerned with assessing the performance for metadata should honor this by implementing it if possible. Otherwise there is avoidable overhead on the side of relational databases.

To summarize this section, with this thesis julea-fuse moves from Figure 3.1a as in [Kuhn, 2017] to Figure 3.1b.

3.8. Statefulness

Fuse file systems can be either stateful or stateless [Vangoor et al., 2017] [libfuse, 2023b]. The previous form of julea-fuse from [Kuhn, 2017] realizes a stateless implementation. Stateless versions relinquish data structures for open files. As fuse runs the functions of the implementation in multiple threads access to an entry would need to be synchronized locally on a machine if the information stored changes after being loaded. For example, the file size could be updated. If this is done by multiple threads the update by one could be lost. Therefore probably mutexes would be needed per entry variable (if updates of it depend on the previous state).

The data structure either is set to a fixed size or is dynamically allocated. A fixed size structure like an array would introduce a limit for the maximum number of opened files. A dynamically allocated structure like a hash table would probably need locking if a threshold of open files is reached and a more are required. All in all, the data structures would lead to limits or overhead. In contrast, a stateless implementation is able to scale better.

In a stateless implementation, metadata needs to be loaded every time it is of interest for an operation. As the JULEA client and the backend usually communicate through a network [Kuhn, 2017] this introduces additional traffic and is time consuming. Therefore, caching already loaded variables could be helpful. Furthermore updates to metadata can be reduced. As JULEA has introduced batches aiming to accomplish this [Kuhn, 2015] it might be possible to collect all metadata operations for a file occurring wile it is open in a single batch. However this would require to make sure this data is not read in the mean time. This again and caching in general introduce problems as described in synchronization Section 3.4. Hence the data cached should not be manipulated by other clients until the file is closed.

If julea-fuse stores open files in a data structure a theoretical possibility is introduced that *close* is not run on an entry. In this case, it could fill a large chunk of memory over time. This is especially problematic as the client is likely a long running application.

As discussed above, allocating data structures when opening files introduces its own problems. However statelessness contradicts caching metadata. As the path to metadata mapping is changed as described in Section 3.7, now the object id is contained in the metadata. To avoid

loading it every time for read and write, the file handle is set to the object id in the data structure passed to those methods as described by [libfuse, 2023b] in *open*. This is a change introduced with this thesis. It can be converted to the name of the object for the operations concerning it. Time updates are avoided by altering metadata once when the *open* operation is run. This however is not done for writes as there the size needs to be updated anyway It depends on the open flags like *O_RDWR* [IEEE and Group, 2018]. This differs from POSIX as there it is suggested that this is done when e.g. *read* is actually run.

3.9. File Content

The current version of JULEA-fuse does not implement *fallocate* and trim. Some modes of *fallocate* [Chinner, 2019] would require deleting space from the middle of the file. This would be feasible if the contents following the section would be moved to the former start of the deleted part. Alternatively a table could be used to define the free spaces in a file. It could be done similar to the catalog for disk blocks in Unix inodes [Andrew S et al., 2015]. This would result in additional overhead for read operations which would need to lookup at which positions zeroes should be read. Moreover, a separate tool would be needed which shrinks the objects to their actual size because every call to *fallocate* in this would create unused space. This could then only be freed by deleting the file. Otherwise it would cause full storage medias with large chunks of unusable and invalid data. Another mode of *fallocate* provides a gap between file content at a given point. This could be realized by copies.

copy_file_range [Schumaker, 2021] is not implemented in the current version of the JULEAs fuse implementation in the source of [Kuhn, 2017]. Due to the fact that there is no interface on JULEAs server supporting it directly, the data to be copied always needs to be transferred through the network. As RAM and network package sizes are limited resources there might be multiple iterations necessary. Using *copy_file_range* reduces the internal communications between the local julea-fuse and the kernel [libfuse, 2023b]. Hence there are some very small benefits by implementing it. Other distribute file systems might be able to provide a more optimal version of this by instructing the object store to perform the copy locally.

As pointed out when describing *fallocate* still a single object is associated with a file as it is in the implementation in [Kuhn, 2017]. *read* and *write* call its appropriate methods. Additionally to manipulating the object *write* still changes the *size* as previously. As this thesis introduces new time metadata this needs to be changed according to [IEEE and Group, 2018]. Therefore *mtime_n*, *mtime_s*, *ctime_n* and *ctime_s* are set to the current time on *write*.

3.10. Directories

On classic file systems directory entries are typically stored together in a specific location [Andrew S et al., 2015]. In the implementation of JULEA as found in [Kuhn, 2017] the files of a directory a retrieved via a *JKVIterator*. The iterator selects every key with the dir name as prefix. Through this method, no separate key value pair needs to be kept per directory containing its entries. If this entry would exist solely for dir iterations it would contain duplicate information. Furthermore it would need to be updated on e.g. every file creation and deletion causing overhead and possibilities for inconsistency for example if only the value in the directory

pathname
/
/foo
/foo/bar
/foo/baz
/foo/baz/bar
/foo/baz/bar/car
/foo/car
/foo0
/foo0/bar

Table 3.5.: Database entries selected by the *JDirectoryIterator* for "/foo"

contents is updated. However it could result in faster lookup of file contents. Looking up one specific value like an entry list should be faster than looking up multiple entries. Therefore the iteration method remains the same. On the relational database side the method is realized with a *JDBSelector* and a *JDBIterator* as introduced in [Warnke, 2019]. As described before, *j_db_selector_add_field* is used to limit the items over which the iterator goes with the help of their path. Using the prefixes for the iteration in both cases, even files in sub directories are returned as is visualized in Table 3.5. Therefore, this is an inefficient implementation with room for improvement in both database types.

While not using a file list, an metadata entry is associated with each directory as was in [Kuhn, 2017]. It contains the same attributes as in Table 3.4. The object value is set to 0. The *atime** and *ctime** in the directory entry remain constant unless directly changed with *utimens*.

Because julea-fuse uses a *readdirplus* method which itself fetches all needed information and lays out data structures *opendir* and *releasedir* are not required [Kuhn, 2017]. Thus they are not implemented. If one would use a entry list it could be loaded or freed with those methods [libfuse, 2023b].

Summary

All in all, in order to support both relational and key value databases with the same methods passed to fuse, a common abstraction was used to bundle their specific code. The synchronization behavior differs as the relational database updates specific fields instead of key value pairs. Access control only happens outside julea-fuse, only the metadata needed is gathered. The path directly identifies the metadata. Hence no tree traversal happens. The object containing the file content however is addressed via an integer id. Hence on opening a file, the id is loaded and used in writes and reads making julea-fuse stateful. The high level interface of fuse is used in julea-fuse. Directories entries are collected via selecting metadata with the directory as prefix in both database implementations.

Chapter 4.

Related Work

In this chapter, contributions are listed who also use databases to store metadata or similar concepts. It is especially described how they organize their path resolution as this needs to be done for every operation depending on whether the result is cached or not and therefore it likely has a high influence on metadata performance.

There are other database types besides key value stores and relational databases [Meier and Kaufmann, 2019]. One of those are xml databases. In [Holupirek, 2012] these are used in metadata management. The framework employed to bind to the kernels file system interface is also fuse. Additionally, it too utilizes fuse_operations. It also incorporates a client-server architecture. A special database is introduced, BaseX-FS. It is split into handling file content, metadata and the directory structure. One goal is to make the metadata accessible through an interface (Xquery). In contrast to classic UNIX, the information in a file is processed by the file system. Hence the interface can be used on it. It processes paths in a classical manner in contrast to the direct access in julea-fuse. The performance of the file system is not assessed. What is of interest to the author is the performance of requests to the database introduced. In contrast to julea-fuse it is not limited to regular files and directories. It also captures POSIX metadata like julea-fuse.

[Niazi et al., 2019] used a NewSQL database. It differs from the SQL-ones in this thesis as it can comprise multiple machines. Like the previous work, **HopsFS** does not use complete paths as indices as julea-fuse does. They take responsibility to assign the file entries to the different nodes. It handles blocks, in contrast to julea-fuse, which assigns keeping the file content to the object store. HopsFS was evaluated on similar operations as julea-fuse but in regard of the behavior with more nodes for the database. Like JULEA it geared towards systems running on multiple machines.

A very early approach realizing a file system relying on a dbms is the **Inversion File System**. It differs from julea-fuse as it aims at giving the same guarantees as relational databases do for manipulations. In order to accomplish that, one relation per file is generated which then keeps its content. The organization in directories is done in a relation. Another relation holds the information for the stat structure. In julea-fuse, the metadata storage and the path resolution is conducted through one table. julea-fuse therefore does not realize all POSIX file system concepts such as hard links. Moreover its content is managed by the object store.

In [Eisl, 2019] work, the file tree is traversed as in a normal file system, not in the direct way as julea-fuse accesses files. The index node number serves as primary key. The metadata is POSIX oriented. It also stores the file content in the database but in another table.

DBFS also uses the same database type as the previous one [Kunchithapadam et al., 2011]. By doing so, the file system should be able to make similar guarantees when manipulations

happen like relational databases do. It can serve multiple machines. From within the fuse layer SQL-Statements are generated. These exert a special ContentAPI deviating from the standard database methods. It has a similar role as the fuse struct or the Virtual File System. Instances can override it with their own SQL logic. Hence one could say that DBFS is more of a family of file systems due to its adaptability. Therefor path resolution may differ. File content is split up, where as julea-fuse uses only one object. For the content SecureFiles is responsible. This is integrated into the database.

Moreover key value stores were introduced in the realm of persistent storage in the form of **tablefs** [Ren and Gibson, 2012]. Tablefs also binds to VFS via fuse. The database used is leveldb. Its purpose is to manage metadata. By doing so, it holds the file hierarchy. Moreover it contains file system entries with a content size under a certain value, namely 4kb. In contrast, the data of big regular files has another destination. It is put on the persistent storage of the computer where tablefs is deployed upon. There it is identified via the index of the metadata respectively st_ino. The first part of the metadata identifier is the serial number of the super ordinate folder. The last part is obtained via hashing the alias with which the file is referenced. In contrast to julea-fuse multiple steps are necessary to get to an entry. This is because julea-fuse uses the path as identifier. However, hard links are supported. In order to accomplish that, there is a split between reference and target. The next integer is the next file id used in contrast to the implementation of this thesis where the next free is guessed by adding a random int to the hash of the path. The related *IndexFS* implements a distributed file system [Ren et al., 2014]. However, it is also not concerned with the file content. This is the responsibility of another file system at least for files bigger than the threshold. Multiple servers can manage the metadata of an file system instance. The metadata format matches closely the one of TableFS.

LocoFS represents another distributed file system [Li et al., 2017]. The database chosen by the creators is a key value store, Kyoto Cabinet. An instance has one Directory Metadata Server in contrast to a collection of File Metadata Servers. The organization of the key value pairs differs from julea-fuse. In LocoFS, The path identifies the directory inode information. julea-fuse does this for all supported file types. The values for regular files however are referenced differently. In this case the string results from appending the last path component to a number mapped to its folder. According to this schema the metadata of `/foo/example` could have the key `10-example` if 10 is assigned to foo depending on the actual grammar used. These inodes are divided into the permissions plus creation time and the attributes altered when the information inside it changes. julea-fuse puts all metadata in one key-value pair thus involving everything in every call. LocoFS only operates on the half which is relevant. The third type of key value pairs are those made of the folder's number referencing the folders item. julea-fuse in contrast implements dir listings via the path names. However, LocoFS exploits internals of the database when changing folder names which is impossible to achieve via a generic key value interface. LocoFS does not convert to an intermediate format like bson. Its values sizes are also constant like in julea-fuse. Another similarity is that an Object Store holds the file content. In contrast to julea-fuse a cache exists in the case of folders.

BabuDB represents a key value store[Stender et al., 2010]. It is geared towards managing metadata. The inode values are distributed across several key value pairs. By having a triplet of metadata information, rewrites are smaller than in julea-fuse were the entire metadata value is exchanged. Values are referenced by the number of the containing folder followed by the last path component and ending with the number describing the metadata component. This is contrary to the approach in this thesis where the path makes up the key. Hard links are possible in BaduDB a separate key value mapping where their inode information resides. The

entry in the original mapping then points to one of the latter. This could be done in JULEA e.g. through using different name spaces, however it would also add additional overhead. BaduDB supports certain consistency requirements of classic file systems.

CHFS also relies on a key value store[Tatebe et al., 2022]. The database is called pmemkv. Similar to JULEA the backend can be spread across multiple nodes. Like julea-fuse, CHFS passes through the file system operations. Metadata is referenced in the same way as in julea-fuse except for an additional identifier at the end. This references the segment of the file's data. Therefore a file consists of multiple database entries contrary to julea-fuse with a single object attached to a metadata value. Inode information is present at every entry. Additionally, the segment of the actual file content resides in it whereas julea-fuse relies on a object store.

BetrFS puts metadata in so called B^ϵ trees [Jannen et al., 2015]. By doing so the authors aim to better the behavior on alterations or insertions of data. It binds directly to VFS unlike julea-fuse, which runs in ring 3. Both file content and inode information are managed by the database TokuDB which was transferred to ring 0 by the authors. The path name resolution happens in the same way as in julea-fuse. The content of a block is found via appending the its identifier to the path. TokuDB passes its data to ext4.

Summary

Multiple other versions of file systems using databases for metadata storage exist. However no one switches between two database types as in julea-fuse. The organization of metadata in regards of the path resolution is done in various ways. Key value stores seem to have been more favorable over time and especially in recent years. Relational databases seem to be less prevalent for this use case.

Chapter 5.

Evaluation

In this chapter, the performance of julea-fuse is measured in regard of common metadata operations. The two different database types are compared against each other. In order to do this, multiple database implementations per type are used and compared. First the operations on regular files are of interest. Afterwards directories are evaluated. Then implications are gathered.

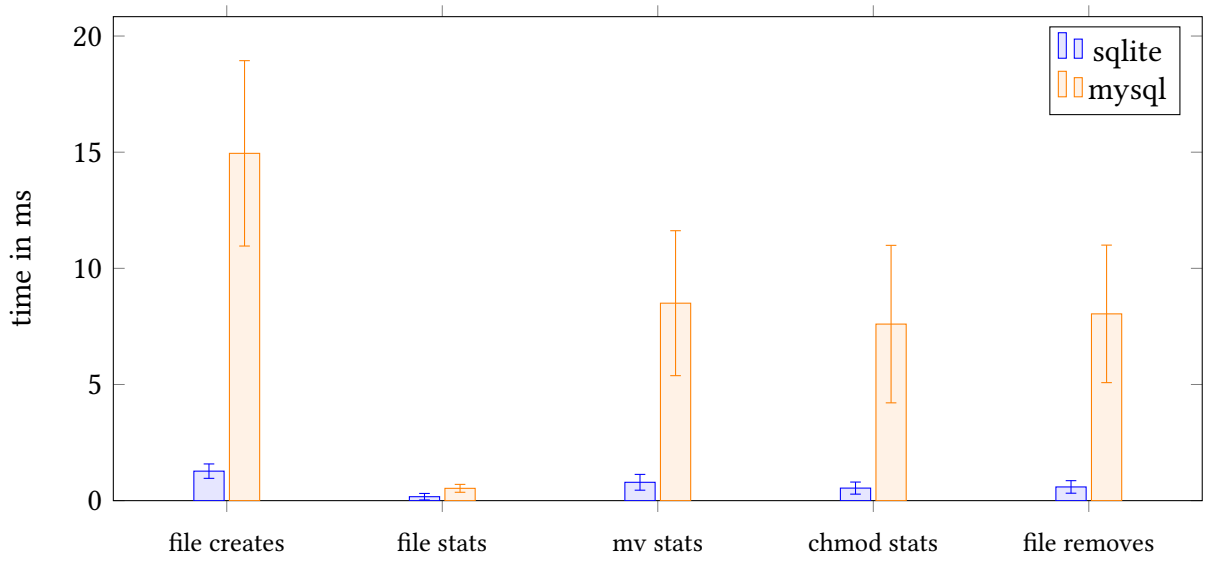
5.1. Setup

The tests where run on a 4 core Intel i5-7500 processor which could access 16 GB of RAM. The file system used was BTRFS. It resided on a WDC WDS500G2B0A Sata SSD. However this is only relevant for benchmarks using backends running as container. The other measurements were taken on componenents storing their data inside tmpfs. The benchmarks where conducted under Fedora 37 kernel version 6.1. The measurements were taken with the mdbench python script [kofemann et al., 2022]. The object store used was posix. The JULEA backends deployed for relational databases were sqlite and mysql. Mysql 8.0 was running inside a container as described inside JULEA's documentation [Kuhn, 2017]. For key value stores, measurements were taken on lmdb, leveldb, rocksdb, sqlite and mongodb. Mongodb 6.0 was also executed within a container. During all measurements the file size was one Kilobyte.

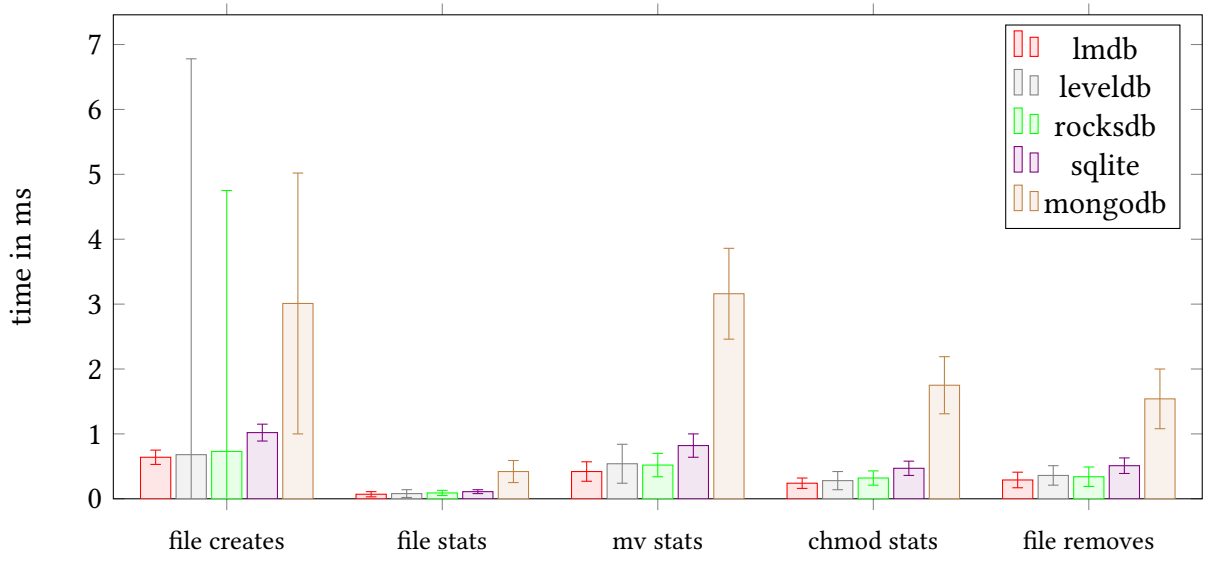
5.2. Measurements

5.2.1. Operations on Regular Files

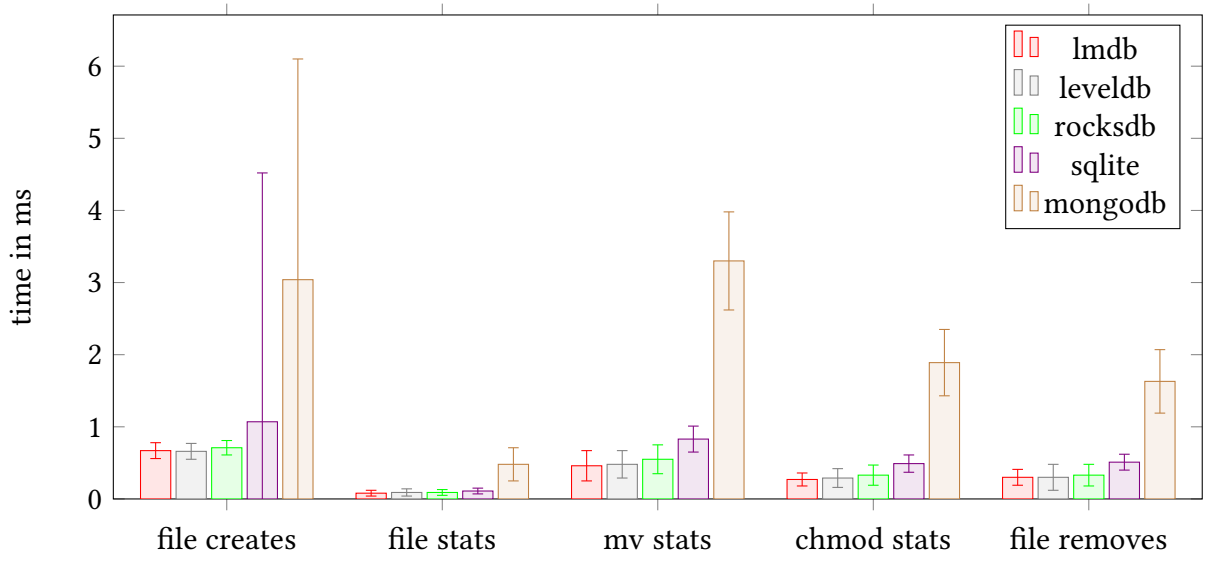
During the measurements the time of specific operations was taken. The creation, retrieval of metadata, changing the owner, changing the name and removal of a file were the actions measured. The operations took place on 32768 files which were created sequentially. The average times for file operations to complete was also measured on the fuse implementation with using bsons to store the metadata internally. By doing so, it can be concluded whether it creates an measurable amount of overhead compared to the struct version or not.



(a) Measurements of operations on regular files in Relational Databases



(b) Measurements of operations on regular files in Key Value Stores



(c) Measurements of operations on regular files in Key Value Stores with bson encoding

Comparing the times within the relational backend in Figure 5.1a it stands out that file creates take the most time. This is likely because the file object needs to be laid out, which involves trying different ids to check which one is still open. Furthermore creates are known to be costly on databases with indices [Warnke, 2019]. Retrieving values for *stat* is the least costly operation. From the remaining operations, *mv* is the lengthiest running function. Like *chmod* one value, the path, needs to be updated. The additional overhead when compared to *chmod* could be caused by the fact an index variable is concerned in this case. Therefore a new position in the internal data structure needs to be determined. In both sqlite and mysql, this is a B-Tree [SQLite, 2023b] [MySQL, 2023a]. Another reason could be that before performing *mv*, the fuse daemon might be querying both files to determine their access rights [IEEE and Group, 2018]. Additionally *rename* itself checks if the new file already exists by trying to load the metadata. *chmod* only involves a single file. File removes require the internal data structure of the relational database to be updated as entries vanish. Furthermore they cause an operation on the object store, which introduces some time overhead. These might be the reason why it needs more time than *chmod*. The latter only changes one variable in one row. Mysql is up to 14 times slower then sqlite. This occurs in *chmod*. In contrast, *stats* has only a difference of three times between the databases.

As mysql actually runs as a server, it might be more difficult to check access permissions for multiple potential users [MySQL, 2023c]. This would not explain the difference between reading (*stats*) and writes (*create*, *mv*, *chmod* and *remove*). Locking might need to be more strict when writing as it invalidates e.g. cached data. As [Saake, 2018] points out it introduces classes of problems like "lost update", "dirty read" and "nonrepeatable read". Hence both relational databases implement different lock types for information retrieval and manipulation[SQLite, 2023c][MySQL, 2023b]. sqlite enforces mutual exclusion respectively consistency on a file level. This might be beneficial in this use case because all operation happen sequentially. Hence being able to lock single entries does not benefit mysql were as sqlite only has to manage one lock and thus less overhead. One of sqlite key aspects is that manipulations are applied straightly at the database file [SQLite, 2023a].

Except for *stat*, mysql's measurements exhibit a very high deviation. The communication via server could be subject to fluctuation. Other processes might also listen at the address configured as in [Kuhn, 2017]. Moreover the operating system may cause utilization of the used SSD whereas the sqlite file on tmpfs has always the same speeds. The mysql server might run some background tasks contrary to the library calls in sqlite [MySQL, 2023c][SQLite, 2023a].

The same structure concerning time of the metadata operations arises when looking at the two key value based fuse implementations. Those can be seen in Figure 5.1b. The most time is needed when using mongodb. As this runs inside a container like mysql the the explanation for it is likely the same. In contrast to the other backends, the server within a container needs to be contacted to execute an operation. In contrast to mysql, mongodb provides only between 2.9 and 4 times slower results. The container/server configuration does not seem to add an constant overhead as the difference between lmdb - the fastest key value store backend in this comparison - and mongodb lays roughly between one 1 and 3 ms whereas in case of mysql and sqlite a gap between 7 and 14 ms occurs.

The next slowest version in the key value store is sqlite. As a relational database the operation is encoded in SQL syntax which is then interpreted [SQLite, 2023b]. Moreover, as this database type gives more guarantees about its behavior, like ACID, additional overhead is introduced especially as this needs to be met at the entire functionality of the relational database, not only

a two-value-tuple. Therefore it might be optimized for another use case than the rest of the key value stores.

Leveldb outperforms rocksdb on *creation*, *stats* and *chmod*. As rocksdb is a descended of leveldb, the similarities make sense [et al., 2023].

Lmdb accomplishes the quickest completion of requests. In contrast to other key value stores it has no own buffer[Chu, 2011]. Rather so called mapped memory serves this function. Alterations occur on duplicated pages contrary to the manipulation of existing ones. Hence the need for mutual exclusion is reduced. According to the results obtained in this thesis, these seem to be beneficial properties for serving as metadata storage. Replicating pages is probably very cost effective when those reside in tmpfs as is the case in this evaluation. It seems to be more effective than collecting changes disconnected from the affected entries as in [et al., 2023]. Rocksdb buffers needed values moreover in a so called memtable.

mongodb's results may show a high standard error for the same reasons as mysql. File creates show really high deviations. As mentioned previously, for objects ids are guessed via the name hash and random numbers. When creating a file, there was probably an id obtained which was already allocated. Therefor the same operation needed to happen again causing really long delays.

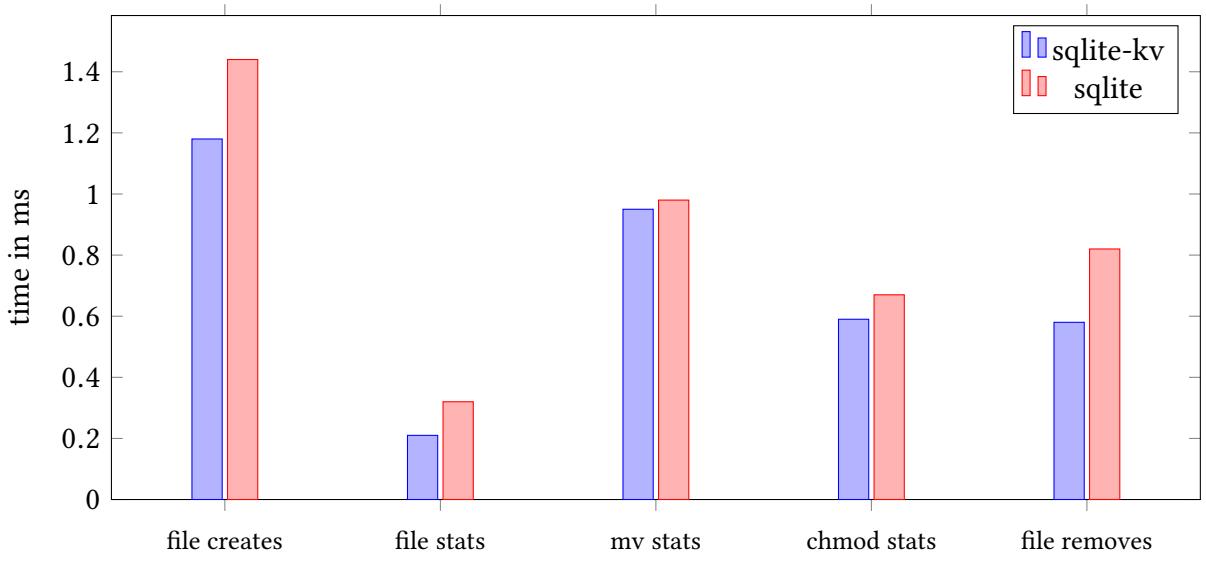


Figure 5.2.: Measurements of operations on directories for sqlite via Key Value Store and Relational Database interface

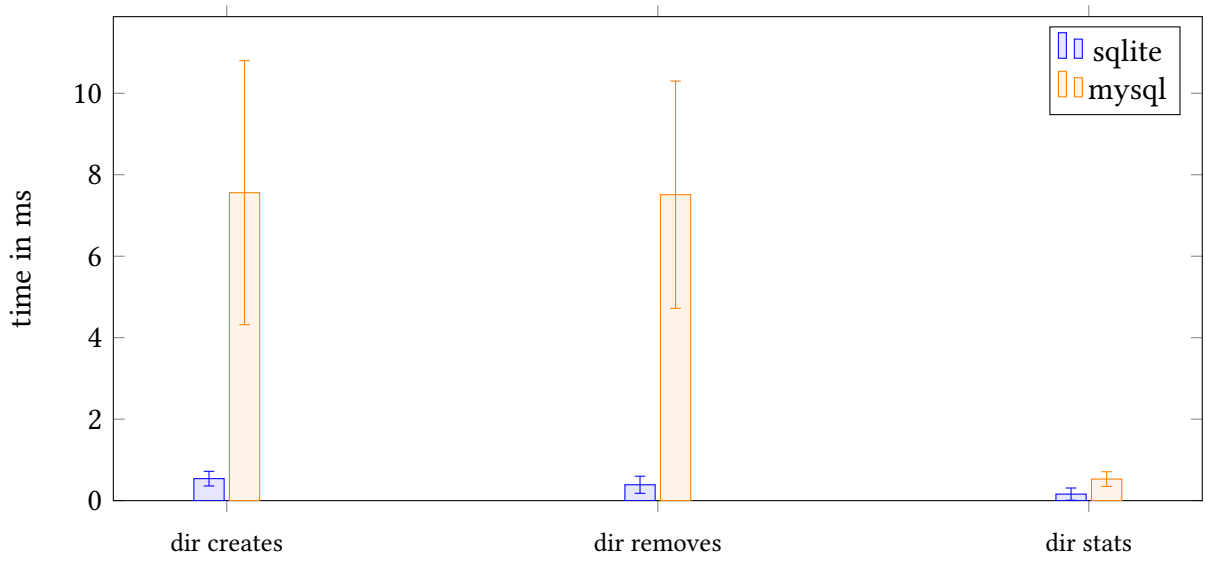
To be able to compare the two backend types together, in Figure 5.2 difference between the key value backend and the relational database backend is shown with the two implementations both using sqlite as the target. It becomes apparent that the key value store version always performs better in terms of latency. Via using the same underlying technology this part of the file system has the same overhead. Only the two interfaces and the fuse methods differ. Therefore by comparing them we can estimate the overhead stemming from those. One important difference between those to is the granularity. As mentioned before the key value store operates on values like their raw data. This is realized in SQL by using *BLOB*. By doing so sqlite is freed of managing the internal structure. For example mode does not need to be an integer. Possibly sqlite can behave more optimized if the path is the primary key and not only an index. In the key value version the current name space and key are used for the former [Kuhn, 2017]. In

the database version only an index is created on the path. The primary key is the *_id* column not directly included in the schema. Keeping those separate might be unhelpful. By knowing that a path value is unique, maybe even in combination with the name space, might provide some possible optimizations like stopping after one tuple. The other key value stores perform even better than sqlite. This is probably because they do not need use SQL as a layer between JULEA and database. Moreover they do not provide full relational database capabilities like enforcing schemas.

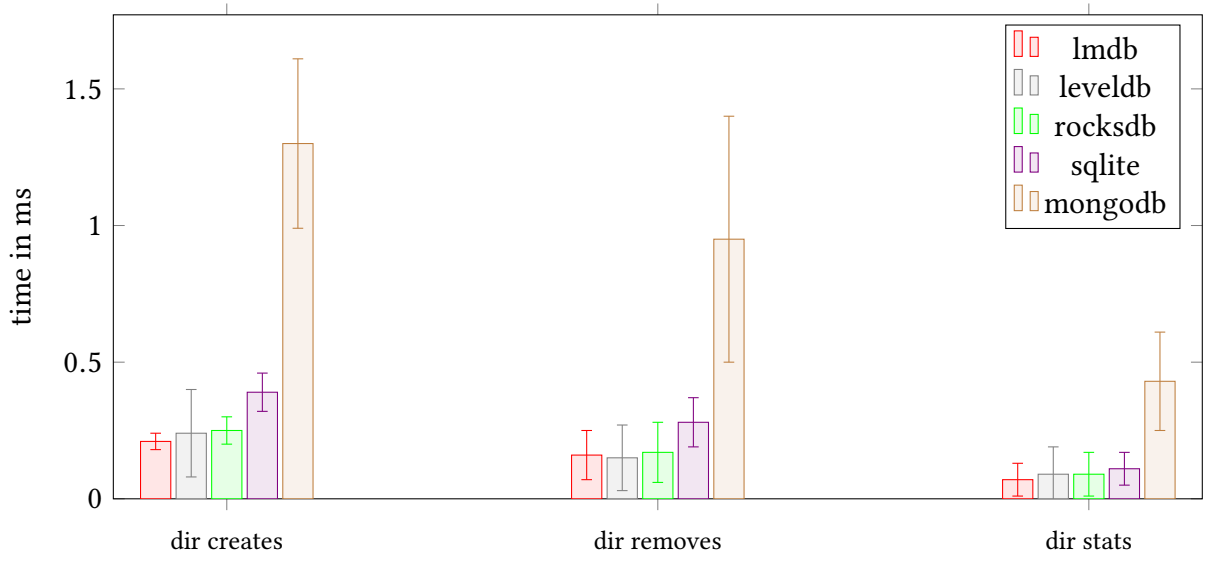
When using bson encoding as seen in Figure 5.1c the pattern remains the same as in the key value stores with structs except that leveldb on average creates files faster than lmdb. The mean of file operations via rocksdb is never faster then leveldb. The average difference between the values in the bson encoded variant and the version using structs to organize the value in the key value store are 2.86 µs. This is might as well be explained by other factors than the different structures implemented. The difference has no meaningful influence on the decision between the two variants.

5.2.2. Operations on Directories

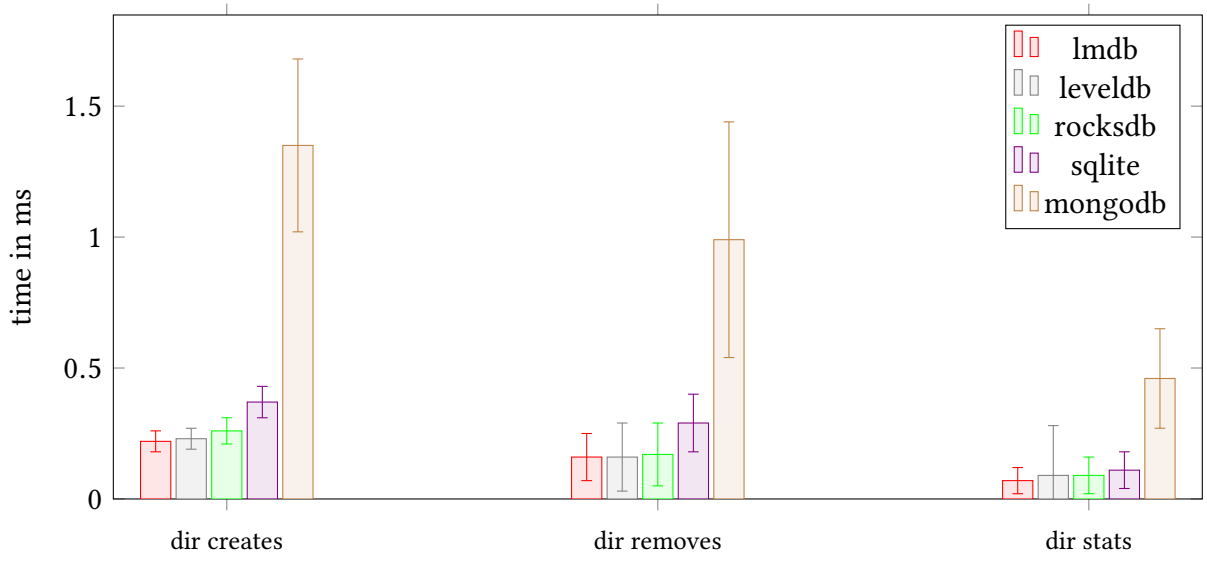
To measure the performance on directories 32768 directory files where created. Each directory contained one regular file.



(a) Measurements of operations on directories in Relational Databases



(b) Measurements of operations on directories in Key Value Stores



(c) Measurements of operations on directories in Key Value Stores with bson encoding

The directory operations show a similar structure to the actions on regular files. Querying existing data is less expensive than adding or removing entries. As the files concerned are directories they do not have an object attached to them. Therefore the overhead of object store operations is removed. Removing entries is still less costly than inserting new file metadata. Hence this is caused by the database, not the object store operations. It might be easier to remove entries than adding new ones. Only in mysql the difference is very narrow with 0.05 ms.

In terms of standard deviation it becomes apparent that directory operations do not exhibit very high values on creation compared to regular files. Therefore the fluctuation on regular file initialization was probably due to the selection of an object id. Another possible cause is overhead when writing the 1 kb of file content which is not necessary in case of directories.

5.3. Implications

The latencies of operations run on a separate server are clearly larger than those run via the library. If access to the server is not required by a third party, moving to a server less solution for file metadata improves access times. This might not hold true in other cases for example if using a server balances resource usage or lets other applications access the metadata.

For file systems with a related structure, key value stores complete file operations faster than relational databases. The latter might be able to provide better performance if path lookup is realized in a less key value oriented way. Also locking in the file system might enable more caching and collecting operations in batches therefore enabling manipulating more data at once. This is the usual mode of operation for relational databases [Saake, 2018] as mentioned before. Key values stores would then need to transmit the entire entry for every tuple in order to update it whereas databases could emit a single query consisting of multiple sql statements. However only few operations were measured. For reads in the case of *stat* for example very little time is needed on every data base backend. *chmod* also requires less time then the other operations writing metadata. As the latter by only changing a value is closely related to changing the size in writes this is a relatively cheap operation. It would be necessary to compare the write times of objects in the object store to the write times of the metadata to conclude whether 0.6 ms are relevant or not. This was not conducted in the thesis as this depends on the choice of the object store. Moreover, different storage medias like HDDs and SSDs would need to be considered.

If latency is the highest priority from the key value stores lmdb should be picked.

Bsons seem to add only very small overhead, at least if the data collected in a document remains small like in this case. As they represent a format designed for data exchange, the benefits - like readability on every machine - implicated by it outweigh the almost non existent overhead.

Summary

Key value stores were consistently able to achieve smaller latencies. The differences between structs and bsons are close to zero in regard of run time. This appears to be also caused by the used interface as provided by JULEA as became clear with sqlite. Very high deviations occurred when creating regular files.

Chapter 6.

Conclusion

In this chapter, possible limitations to the evaluation setup for metadata performance and general observations when implementing julea-fuse are described. Afterwards directions which could be assessed in fuse file systems which use databases as metadata target are pointed out.

6.1. Conclusion

All in all, it is apparent that key value databases complete queries faster than relational databases under the given workload. For file systems using a similar approach to metadata as this one, a key value store is probably the most appropriate solution. This might be controversial if other capabilities of a database are required which were not part of the evaluation. The difference between the database types is most significant on changes to the values of metadata. Especially the absolute values in file creation and removal diverge. If this is of no interest, for example if the file system to be created is to be used predominantly for read access, other metrics might be more relevant for selection than access time.

In the case of JULEA, it can be noticed that the key value store solution outperforms the relational database solution on every operation. This is true independent of the backend, as shown in the case of sqlite. Therefore JULEA's key value interface offers better key value performance on fuse file system metadata than the relational database backend does. This is somewhat surprising as the same databases should show similar performance on the same requests.

As the previous key value format from [Kuhn, 2017] is kept this may have been a cause for the outstanding performance of key value stores as it was originally geared towards them.

The overhead of using bson to obtain a intermediate format of metadata is nearly immeasurable. Thus it can be recommended if flexibility of the contained values or machine independence is needed. There was some overhead expected by this format but apparently it takes a very small amount of time compared to other operations like transfer to the server and accessing its underlying file system. This however might depend on the number and size of attributes and their type. If their length varies the metadata value might need to be resized [MongoDB, 2017].

File systems, especially those aiming at HPC use cases, apparently provide room for many different design decisions as was shown in Section 3.7 and in Chapter 4. This is especially true for path name resolution where many different approaches exist. Due to the design used, the obtained results might be more representative for files systems who use a similar path to value mapping than those who follow the tree structure. Different approaches exist for file content as

well like some discussed in Chapter 4. Those could cause slightly different results most likely in the proportion of creates and deletes when multiple segments need to be deleted or another structure referencing them is put in the inode.

6.2. Future Work

An important open aspect remained the correctness under concurrent conditions. As mentioned before, the metadata can for example be changed after it is loaded from the key value store so when putting it back it has missed some updates. If there is a solution at the level of JULEA, the concept should also be adapted for julea-fuse. By doing that, additional overhead is introduced to the key value version. Hence the comparison of key value stores and relational databases could lead to different results with those changes.

An other topic are the methods expected by fuse but which are not implemented yet. This might not concern the question of the thesis. For example locking might not have a different performance on relational databases than on key value stores. However, the fuse implementation in the state of this thesis still misses functionality from [libfuse, 2023b].

When adding for example hard links, the functionality of relational databases could be used more extensively. As mentioned before, hard links would result in multiple entries for the same metadata resulting in multiple tables especially if normal forms [Saake, 2018] are considered. In this case, in the relational database case tables could be joined. This is in contrast to the key value store where multiple retrievals of values would be necessary. Another similar case would be *getxattr*, *listxattr* and *setxattr* from [libfuse, 2023b]. However, those might cause only a single request to the key value store as the additional properties could be stored in the metadata value. All in all, relational databases offer a much more complex interface. This could be maxed out in future work highlighting the functionality benefits in the realm of file systems.

There are other database types as well, like graph databases[Meier and Kaufmann, 2019] and are used in [Holupirek, 2012]. If other database types are introduced to JULEA, those might show a different performance profile. Future effort could be made to assess their fitting for file system metadata.

Another interesting aspect is the possibility to scale the systems. Key value stores can easily be expanded with additional nodes [Meier and Kaufmann, 2019]. This was not addressed in the setup the measurements were taken on.

Additionally, the two performances may vary whether the low level version of fuse is used or not. There the access is done via inodes [libfuse, 2023a]. the data structures inside the relational database or the hash tables might perform differently. Key value stores usually retrieve data identified with a string like described in Section 2.4.1. This might not be as optimal as relational databases. Those can construct primary keys using different data types. Hence they could offer improved performance in this use case.

Summary

All in all, identifying the changes needed on the existing implementation to add the other database backend type was a complex task. Currently key value stores outperform relational

databases. However with different implementations this deviates. Moreover, different variables than the latency might be considered in the future.

Bibliography

[Alam et al., 2011] Alam, S. R., El-Harake, H. N., Howard, K., Stringfellow, N., and Verzelloni, F. (2011). Parallel i/o and the metadata wall. In *Proceedings of the sixth workshop on Parallel Data Storage*, pages 13–18. (Cited on page 4)

[Andrew S et al., 2015] Andrew S, T., Herbert, B., et al. (2015). *Modern Operating Systems.-4th*. Pearson. (Cited on pages 6, 7, 9, 22, 23, 24, and 26)

[Carns et al., 2011] Carns, P. H., Harms, K., Allcock, W. E., Bacon, C., Lang, S., Latham, R., and Ross, R. B. (2011). Understanding and improving computational science storage access through continuous characterization. In Brinkmann, A. and Pease, D., editors, *IEEE 27th Symposium on Mass Storage Systems and Technologies, MSST 2011, Denver, Colorado, USA, May 23-27, 2011*, pages 1–14. IEEE Computer Society. (Cited on page 4)

[Chinner, 2019] Chinner, D. (2019). fallocate(2) — linux manual page. `https://man7.org/linux/man-pages/man2/fallocate.2.html`. Accessed: 2023-02-07. (Cited on page 26)

[Chu, 2011] Chu, H. (2011). Mdb: A memory-mapped database and backend for openldap. In *Proceedings of the 3rd International Conference on LDAP, Heidelberg, Germany*, volume 35. (Cited on page 34)

[Eeckhout, 2017] Eeckhout, L. (2017). Is moore's law slowing down? what's next? *IEEE Micro*, 37(4):4–5. (Cited on page 4)

[Eisl, 2019] Eisl, R. (2019). Conflict aware network file system based on a relational database. Master's thesis, University of Salzburg. (Cited on page 28)

[et al., 2023] et al., D. B. (2023). Rocksdb overview. `https://github.com/facebook/rocksdb/wiki/RocksDB-Overview`. Accessed: 2023-03-30. (Cited on page 34)

[fuse, 2023a] fuse (2023a). fuse_file_info struct reference. `https://libfuse.github.io/doxygen/structfuse__file__info.html`. Accessed: 2023-03-30. (Cited on pages 23 and 24)

[fuse, 2023b] fuse (2023b). libfuse libfuse: The reference implementation of the linux fuse (filesystem in userspace) interface. `https://github.com/libfuse/libfuse`. Accessed: 2023-02-26. (Cited on page 9)

[Hertzsprung, 2007] Hertzsprung (2007). Privilege rings for the x86 microprocessor architecture available in protected mode. operating systems determine which processes run in each mode. `https://en.wikipedia.org/wiki/Operating_system#/media/File:Priv_rings.svg`. Accessed: 2023-03-27. (Cited on page 9)

[Holupirek, 2012] Holupirek, A. (2012). *Declarative Access to Filesystem Data: New application domains for XML database management systems*. PhD thesis, University of Konstanz. (Cited on pages 28 and 39)

[IEEE and Group, 2018] IEEE and Group, T. O. (2018). Ieee standard for information technology–portable operating system interface (posix(tm)) base specifications, issue 7. *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)*, pages 1–3951. (Cited on pages 7, 14, 17, 22, 26, and 33)

[Jannen et al., 2015] Jannen, W., Yuan, J., Zhan, Y., Akshintala, A., Esmet, J., Jiao, Y., Mittal, A., Pandey, P., Reddy, P., Walsh, L., Bender, M. A., Farach-Colton, M., Johnson, R., Kuszmaul, B. C., and Porter, D. E. (2015). Betrfs: A right-optimized write-optimized file system. In Schindler, J. and Zadok, E., editors, *Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST 2015, Santa Clara, CA, USA, February 16-19, 2015*, pages 301–315. USENIX Association. (Cited on page 30)

[Kira Duwe and Michael Kuhn, 2021] Kira Duwe and Michael Kuhn (2021). Coupled storage system for efficient management of self-describing data formats (cosemos). Technical report, Otto von Guericke Universität Magdeburg. (Cited on page 13)

[kofemann et al., 2022] kofemann et al. (2022). kofemann/mdbench: simple filesystem metadata operation benchmark. `https://github.com/kofemann/mdbench`. Accessed: 2023-03-15. (Cited on pages 2, 16, 24, and 31)

[Kuhn, 2015] Kuhn, M. (2015). *Dynamically Adaptable I/O Semantics for High Performance Computing*. PhD thesis, University of Hamburg. (Cited on pages 13, 14, and 25)

[Kuhn, 2017] Kuhn, M. (2017). JULEA: A flexible storage framework for HPC. In Kunkel, J. M., Yokota, R., Taufer, M., and Shalf, J., editors, *High Performance Computing - ISC High Performance 2017 International Workshops, DRBSD, ExaComm, HCPM, HPC-IODC, IWOPH, IXPUG, P^3MA, VHPC, Visualization at Scale, WOPSSS, Frankfurt, Germany, June 18-22, 2017, Revised Selected Papers*, volume 10524 of *Lecture Notes in Computer Science*, pages 712–723. Springer. (Cited on pages 2, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 25, 26, 27, 31, 33, 34, and 38)

[Kunchithapadam et al., 2011] Kunchithapadam, K., Zhang, W., Ganesh, A., and Mukherjee, N. (2011). Oracle database filesystem. In Sellis, T. K., Miller, R. J., Kementsietsidis, A., and Velegrakis, Y., editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 1149–1160. ACM. (Cited on page 28)

[Li et al., 2017] Li, S., Lu, Y., Shu, J., Hu, Y., and Li, T. (2017). Locofs: a loosely-coupled metadata service for distributed file systems. In Mohr, B. and Raghavan, P., editors, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2017, Denver, CO, USA, November 12 - 17, 2017*, page 4. ACM. (Cited on page 29)

[libfuse, 2023a] libfuse (2023a). fuse_lowlevel_ops struct reference. `http://libfuse.github.io/doxygen/structfuse__lowlevel__ops.html`. Accessed: 2023-02-07. (Cited on pages 10, 23, and 39)

[libfuse, 2023b] libfuse (2023b). fuse_operations struct reference. `https://libfuse.github.io/doxygen/structfuse__operations.html`. Accessed: accessed 2023-02-07. (Cited on pages 10, 25, 26, 27, and 39)

[Luu et al., 2015] Luu, H., Winslett, M., Gropp, W., Ross, R., Carns, P., Harms, K., Prabhat, M., Byna, S., and Yao, Y. (2015). A multiplatform study of i/o behavior on petascale supercomputers. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 33–44. (Cited on page 4)

[Meier and Kaufmann, 2019] Meier, A. and Kaufmann, M. ([2019]). *SQL & NoSQL databases : models, languages, consistency options and architectures for Big Data Management.* Computer Science and Engineering | Springer eBook Collection. Springer Vieweg, Wiesbaden, array edition. 1 Online-Ressource (xvi, 228 Seiten), Illustrationen. (Cited on pages 2, 11, 22, 28, and 39)

[mongodb, 2023] mongodb (2023). Do things big with mongodb at scale. https://www.mongodb.com/mongodb-scale. Accessed: 2023-03-29. (Cited on page 4)

[MongoDB, 2017] MongoDB, I. (2017). bson_new_from_buffer(). http://mongoc.org/libbson/current/bson_new_from_buffer.html. Accessed: 2023-03-30. (Cited on page 38)

[MySQL, 2023a] MySQL (2023a). How mysql uses indexes. https://dev.mysql.com/doc/refman/8.0/en/mysql-indexes.html. Accessed: 2023-03-30. (Cited on page 33)

[MySQL, 2023b] MySQL (2023b). Innodb locking. https://dev.mysql.com/doc/refman/8.0/en/innodb-locking.html. Accessed: 2023-03-30. (Cited on page 33)

[MySQL, 2023c] MySQL (2023c). The main features of mysql. https://dev.mysql.com/doc/refman/8.0/en/features.html. Accessed: 2023-03-29. (Cited on pages 4 and 33)

[Niazi et al., 2019] Niazi, S., Ismail, M., Haridi, S., and Dowling, J. (2019). Hopsfs: Scaling hierarchical file system metadata using newsql databases. In Sakr, S. and Zomaya, A. Y., editors, *Encyclopedia of Big Data Technologies*. Springer. (Cited on page 28)

[OLCF, 2023] OLCF (2023). Data storage and transfers. https://docs.olcf.ornl.gov/data/index.html. Accessed: 2023-03-30. (Cited on page 4)

[Ren and Gibson, 2012] Ren, K. and Gibson, G. (2012). Tablefs: Embedding a nosql database inside the local file system. In *2012 Digest APMRC*, pages 1–6. (Cited on pages 24 and 29)

[Ren et al., 2014] Ren, K., Zheng, Q., Patil, S., and Gibson, G. A. (2014). Indexfs: Scaling file system metadata performance with stateless caching and bulk insertion. In Damkroger, T. and Dongarra, J. J., editors, *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, New Orleans, LA, USA, November 16-21, 2014*, pages 237–248. IEEE Computer Society. (Cited on page 29)

[Saake, 2018] Saake, G. ([2018]). *Datenbanken : Konzepte und Sprachen.* mitp Verlags, Frechen, array edition. 1 online resource (1 volume), illustrations. (Cited on pages 12, 21, 24, 25, 33, 37, and 39)

[Schumaker, 2021] Schumaker, A. (2021). copy_file_range(2) — linux manual page. https://man7.org/linux/man-pages/man2/copy_file_range.2.html. Accessed: 2023-02-12. (Cited on page 26)

[SQLite, 2023a] SQLite (2023a). About sqlite. https://www.sqlite.org/about.html. Accessed: 2023-03-30. (Cited on page 33)

[SQLite, 2023b] SQLite (2023b). Architecture of sqlite. https://www.sqlite.org/arch.html. Accessed: 2023-03-30. (Cited on page 33)

[SQLite, 2023c] SQLite (2023c). File locking and concurrency in sqlite version 3. https://www.sqlite.org/lockingv3.html. Accessed: 2023-03-30. (Cited on page 33)

[Stender et al., 2010] Stender, J., Kolbeck, B., Högqvist, M., and Hupfeld, F. (2010). Babudb: Fast and efficient file system metadata storage. In *2010 International Workshop on Storage Network Architecture and Parallel I/Os*, pages 51–58. IEEE. (Cited on pages 24 and 29)

[Sven and ElementW, 2019] Sven and ElementW (2019). Structural diagramm of filesystem in userspace. https://commons.wikimedia.org/wiki/File:FUSE_structure.svg. Accessed: 2023-02-28. (Cited on page 10)

[Tatebe et al., 2022] Tatebe, O., Obata, K., Hiraga, K., and Ohtsuji, H. (2022). CHFS: parallel consistent hashing file system for node-local persistent memory. In *HPC Asia 2022: International Conference on High Performance Computing in Asia-Pacific Region, Virtual Event, Japan, January 12 - 14, 2022*, pages 115–124. ACM. (Cited on pages 23, 24, and 30)

[Top500, 2022] Top500 (2022). Top500 list - november 2022. https://www.top500.org/lists/top500/list/2022/11/. Accessed: 2023-03-30. (Cited on page 4)

[ubuntusers, 2021] ubuntusers (2021). Fuse. https://wiki.ubuntuusers.de/FUSE/. Accessed: 2023-02-26. (Cited on page 9)

[Van Steen and Tanenbaum, 2017] Van Steen, M. and Tanenbaum, A. S. (2017). *Distributed systems*. Maarten van Steen Leiden, The Netherlands. (Cited on pages 21 and 22)

[Vangoor et al., 2017] Vangoor, B. K. R., Tarasov, V., and Zadok, E. (2017). To FUSE or not to FUSE: performance of user-space file systems. In Kuenning, G. and Waldspurger, C. A., editors, *15th USENIX Conference on File and Storage Technologies, FAST 2017, Santa Clara, CA, USA, February 27 - March 2, 2017*, pages 59–72. USENIX Association. (Cited on pages 4, 9, 10, 22, 23, and 25)

[Warnke, 2019] Warnke, B. (2019). Integrating self-describing data formats into file systems. Master's thesis, Universität Hamburg. (Cited on pages 13, 14, 16, 17, 18, 19, 24, 27, and 33)

[Wiese, 2015] Wiese, L. (2015). *Advanced Data Management for SQL, NoSQL, Cloud and Distributed Databases*. DeGruyter. (Cited on pages 11 and 24)

Statement of Authorship

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

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

Magdeburg, April 3, 2023

_____

Signature