Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Mikael Feriencik, Thomas Trachsel

Parallel distributed filesystem with metadata database

Diploma Thesis DA-2004-11
November 2003 to March 2004

Supervisor: Prof. Bernhard Plattner
Co-Supervisor: Martin Frey, Supercomputing Systems AG, Zuerich

We would like to thank Martin Frey from Supercomputing Systems AG for supervising and helping us through this thesis, Prof. Bernhard Plattner making this work possible and Supercomputing Systems AG for hosting us these 4 months.

Abstract

Lustre (www.lustre.org) is a modular, distributed filesystem designed for the use with supercomputers and high performance clusters and is developed as an Open Source project. The goal is to support up to 10’000 clients, 1000 Object Storage targets (OST) and 100 Metadata Servers (MDS). The first stage of the Lustre implementation, called Lustre Lite, is limited to the use of one MDS. If the clients use large directories, then this MDS is the bottleneck of the filesystem. One solution to eliminate this bottleneck is to use more than one MDS, but is a complex task.
The goal of this diploma thesis is to implement an efficient Metadata management system (files, directories) by using a SQL database.
This task can be separated into two parts. First a filesystem (sqlfs) that can be used like ext2 or ext3 but stores the data in a SQL database was designed and implemented. This sqlfs consist of three parts, the kernel module that implements all Virtual Filesystem (VFS) functions and communicates with the DB-Client. This userspace program acts as a proxy between the kernel module and the database, which in turn stores the data.
Then, this filesystem was integrated into Lustre. To accomplish this, a new filterdriver for the MDS had to be written.
At last we benchmarked the sqlfs filesystem within Lustre and in its native mode and compared the results with ext3 and the actual implementation of Lustre. The results were promising, as the creation of files in large directories is faster than ext3 and it scales well.

Lustre (www.lustre.org) ist ein modulares, verteiltes Dateisystem, das für den Einsatz in Supercomputer bestimmt ist und als Open Source Projekt entwickelt wird. Das Ziel ist es 10’000 Clients, 10’000 Datenserver (OST) und 100 Metadaten Server (MDS) zu unterstützen. In einer ersten Phase von Lustre, dem sogenannten Lustre Lite, wird nur ein einziger Metadatenserver eingesetzt. Dieser wird jedoch bei grossen Verzeichnissen schnell zum Engpass. Die Einführung mehrerer MDS ist eine Möglichkeit die Performance zu verbessern, ist aber schwierig zu implementieren.
Das Ziel dieser Diplomarbeit ist es ein effizientes Metadatenverwaltungsystem auf Basis einer SQL Datenbank zu implementieren. Diese Aufgabe kann in zwei Teile aufgeteilt werden.
Als Erstes wurde ein Dateisystem (sqlfs) implementiert, das wie ext2 oder ext3 verwendet werden kann. Das sqlfs Dateisystem kann in drei Teile aufgeteilt werden, das Kernel Modul implementiert alle benötigten Virtual Filsystem (VFS) Funktionen und kommuniziert mit dem Datenbank Client (DB-Client). Dieses Userspace Programm arbeitet als Proxy zwischen dem Kernel Modul und der Datenbank, die wiederum die Dateisysteminformationen abspeichert.
Anschliessend wurde das sqlfs Dateisystem ins Lustre integriert, was die Entwicklung eines neuen MDS Filtertreibers zur Folge hat.
Als letztes wurde die Geschwindigkeit des sqlfs gemessen und mit derjenigen von ext3 und Lustre verglichen. Die Resultate sind vielversprechend, das Erstellen von Dateien in sqlfs in grossen Verzeichnissen ist schneller als in ext3. Zudem skaliert das sqlfs Dateisystem sehr gut.

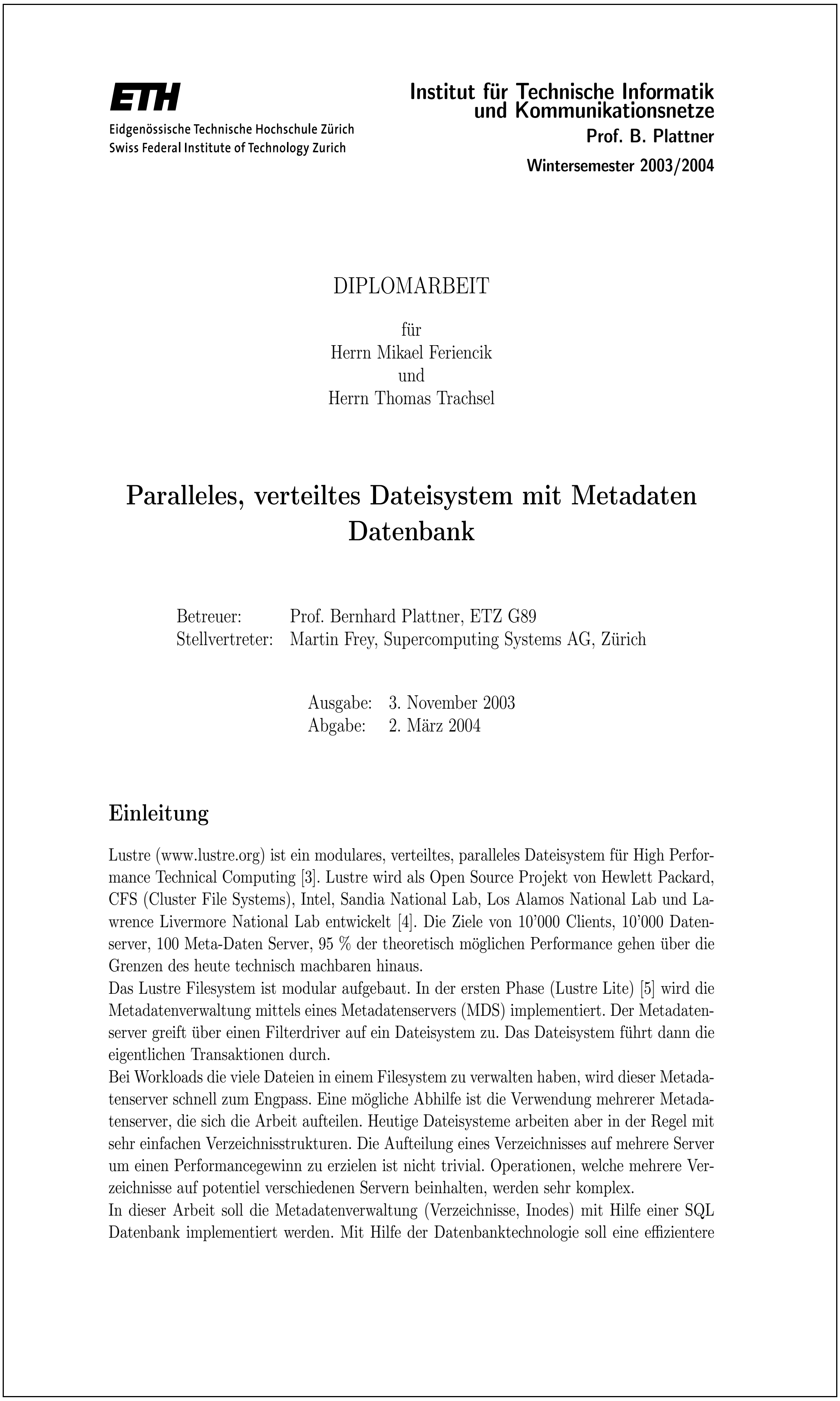
Contents

List of Figures

List of Tables

Chapter 1

Assignment by the Communication Systems Group



ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Technische Informatik und Kommunikationsnetze
Prof. B. Plattner
Wintersemester 2003/2004

DIPLOMARBEIT

für
Herrn Mikael Feriencik
und
Herrn Thomas Trachsel

Paralleles, verteiltes Dateisystem mit Metadaten Datenbank

Betreuer:	Prof. Bernhard Plattner, ETZ G89
Stellvertreter:	Martin Frey, Supercomputing Systems AG, Zürich

Ausgabe:	3. November 2003
Abgabe:	2. März 2004

Einleitung

Lustre (www.lustre.org) ist ein modulares, verteiltes, paralleles Dateisystem für High Performance Technical Computing [3]. Lustre wird als Open Source Projekt von Hewlett Packard, CFS (Cluster File Systems), Intel, Sandia National Lab, Los Alamos National Lab und Lawrence Livermore National Lab entwickelt [4]. Die Ziele von 10'000 Clients, 10'000 Datenserver, 100 Meta-Daten Server, 95 % der theoretisch möglichen Performance gehen über die Grenzen des heute technisch machbaren hinaus.
Das Lustre Filesystem ist modular aufgebaut. In der ersten Phase (Lustre Lite) [5] wird die Metadatenverwaltung mittels eines Metadatenservers (MDS) implementiert. Der Metadatenserver greift über einen Filterdriver auf ein Dateisystem zu. Das Dateisystem führt dann die eigentlichen Transaktionen durch.
Bei Workloads die viele Dateien in einem Filesystem zu verwalten haben, wird dieser Metadatenserver schnell zum Engpass. Eine mögliche Abhilfe ist die Verwendung mehrerer Metadatenserver, die sich die Arbeit aufteilen. Heutige Dateisysteme arbeiten aber in der Regel mit sehr einfachen Verzeichnisstrukturen. Die Aufteilung eines Verzeichnisses auf mehrere Server um einen Performancegewinn zu erzielen ist nicht trivial. Operationen, welche mehrere Verzeichnisse auf potentiel verschiedenen Servern beinhalten, werden sehr komplex.
In dieser Arbeit soll die Metadatenverwaltung (Verzeichnisse, Inodes) mit Hilfe einer SQL Datenbank implementiert werden. Mit Hilfe der Datenbanktechnologie soll eine effizientere

Verwaltung der Verzeichnisdaten implementiert werden. Dadurch soll eine höhere Metadatentransaktionsrate erreicht werden. Es soll ein rudimentäres Linux Dateisystem implementiert werden, dass die von Lustre benutzten Metadatenoperationen effizient implementiert.
In einer Semesterarbeit [2] wurde ein Prototyp eines Filesystems zur Metadatenverwaltung entwickelt. Die Metadatenverwaltung soll auf diesem Prototyp basierend weiterentwickelt werden.

Aufgabenstellung

1. Machen Sie sich mit der Lustre Architektur und der Postgres Datenbank vertraut.
2. Identifizieren Sie die Schnittstellen zwischen Metadatenserver (MDS), dem Filterdriver und dem unterliegenden Dateisystem.
3. Machen Sie sich mit den Ergebnissen der Semesterarbeit [2] vertraut.
4. Identifizieren Sie die fehlenden Metadatenmethoden und implementieren sie diese.
5. Entwerfen und implementieren sie eine Möglichkeit, um auch Daten in dem Dateisystem zu speichern. Lustre legt nur Konfigurationsdaten auf dem Metadatenserver ab, Zugriffe müssen deshalb nicht besonders effizient sein, aber Änderungen müssen als Transaktionen erfolgen.
6. Führen Sie einen Systemtest durch.
7. Performance Analyse:
   - Legen Sie eine Methodik für die Performance Analyse fest.
   - Führen Sie die Analyse nach dieser Methodik durch.
   - Identifizieren Sie Bottlenecks.
8. Tunen Sie das System für bessere Performance.

Durchführung der Diplomarbeit

Allgemeines

- Mit dem Betreuer sind regelmässige, zumindest wöchentliche Sitzungen zu vereinbaren. In diesen Sitzungen sollen die Studenten mündlich über den Fortgang der Arbeit und die Einhaltung des Zeitplanes berichten und anstehende Probleme diskutieren.
- Am Ende des ersten Monates muss eine Vorabversion des Inhaltsverzeichnis zur Dokumentation dem Betreuer abgegeben und mit diesem besprochen werden.
- Nach der Hälfte der Arbeitsdauer soll ein kurzer mündlicher Zwischenbericht abgegeben werden, der über den Stand der Arbeit Auskunft gibt. Dieser Zwischenbericht besteht aus einer viertelstündigen, mündlichen Darlegung der bisherigen Schritte und des weiteren Vorgehens gegenüber Prof. Plattner.
- Am Schluss der Arbeit muss eine Präsentation von 15 + 5 Minuten im Fachgruppen- oder Institutsrahmen gegeben werden. Anschliessend an die Schlusspräsentations soll die Arbeit Interessierten praktisch vorgeführt werden.

- Bereits vorhandene Software kann übernommen und gegebenenfalls angepasst werden.
- Es ist ein mit Bindespiralen gebundener Schlussbericht (am TIK vorhanden) über die geleistete Arbeit abzuliefern (4 Exemplare). Dieser Bericht besteht aus einer Zusammenfassung, einer Einleitung, einer Analyse von verwandten und verwendeten Arbeiten, sowie einer vollständigen Beschreibung der Konfiguration von den eingesetzten Programmen. Der Bericht ist in Deutsch oder Englisch zu halten. Die Zusammenfassung muss in Deutsch und Englisch verfasst werden. Reservieren Sie sich für das Erstellen des Berichtes ein bis zwei Wochen Zeit am Ende der Arbeit.
- Die Arbeit muss auf CDROM archiviert abgegeben werden. Stellen Sie sicher, dass alle Programme sowie die Dokumentation sowohl in der lauffähigne, resp. druckbaren Version als auch im Quellformat vorhanden, lesbar und verwendbar sind. Mit Hilfe der abgegebenen Dokumentation muss der entwickelte Code zu einem ausführbaren Programm erneut übersetzt werden können.
- Diese Arbeit steht unter der GNU General Public License (GNU GPL).
- Diese Arbeit wird als Diplomarbeit an der ETH Zürich durchgeführt. Es gelten die Bestimmungen hinsichtlich Kopier- und Verwertungsrechte der ETH Zürich.
- Sie verfügen über einen Arbeitsplatz in der Supercomputing Systems AG mit PC und Zugriff auf Testrechner.
- Stellen Sie Ihr Projekt zu Beginn der Diplomarbeit in einem Kurzvortrag vor.

Zürich, den 3. November 2003

Prof. B. Plattner

Literatur

[1] Wegleitung für Semester- und Diplomarbeiten am Institut für Elektronik, ETH, tha 4/97

[2] Parallel Distributed Filesystem with Metadata Database, Reinhard Bischoff, Robert Hunger, Andreas Mühlemann, Summer Term 2003, Electronics Laboratory, Departement of Information Technology and Electrical Engineering, Swiss Federal Institute of Technology, Zurich

[3] The Lustre Storage Architecture, Peter J. Braam, Cluster File Systems, Inc., http://www.lustre.org/docs.html

[4] Statement of Work: SGS File System, DOE National Nuclear Security Administration & the DOD National Security Agency, http://www.lustre.org/docs/SGSRFP.pdf

[5] Lustre Technical Project Summary, Cluster File Systems, Inc., http://www.lustre.org/docs/lustre-sow-dist.pdf

Chapter 2

Introduction

2.1 Motivation and Overview

Distributed Filesystems separate computational from storage resources, which allows the workstation to focus on their main task, the computation of complex problems. Supercomputers usually have a high demand for storage capacity and bandwith and in the future this demand will continually increase. So, it is crucial to have a filesystem that can scale with increasing demand. Another important point is consistency. An enterprise filesystem always needs to be consistent, even under heavy load. If one client changes e.g. an entry in a directory, the other clients have to see this changes immediately.
Most of the network based filesystems like NFS do not provide the needed enterprise features described above. NFS does not even guarantee that a client has always the actual view of the directory. It's just guaranteed that the filesystem on the server is in a consistent state.

Lustre [2] is a new storage and filesystem architecture designed for supercomputers and clusters and provides all the above mentioned features. The Lustre filesystem consists of three main parts, the Metadata Server (MDS), the Object Storage Target (OST) and of course the client. The developers of Lustre aim to serve 10000 clients, 1000 object storage targets and 100 metadata servers. At the moment, there exists only an implementation of the Lustre Lite filesystem, which is limited to one MDS. The MDS stores all the metadata information of Lustre, e.g. the directory hierarchy and the location of the stored data[1] If the clients work with large directories, as it is common in scientific computation, the MDS can be identified as the bottleneck of the system. The idea is now to implement a new filesystem based on a SQL database and integrate it into Lustre to improve the transaction rate, as a database is built to work on large datasets.

[1]As Lustre is a distributed fi lesystem that is able to handle striped fi les over several OSTs, the location of each fi le part has to be stored. In Lustre this information is stored in extended fi le attributres

Chapter 3

The Lustre File System

The Lustre File System is an open source, high-performance distributed file system developed by Cluster File Systems, Inc., created from scratch to become the next-generation file system for superclusters. The name "Lustre" is an amalgam of the terms "Linux" and "Clusters". Lustre is designed to meet the following performance goals:

- tens of thousands of clients
- thousands of storage servers
- hundreds of metadata servers
- petabytes of data
- more than 100GB/s I/O throughput

Lustre also supports redundancy by eliminating any single point of failure in the distributed file system environment. To achieve these goals Lustre has a highly modular design, consisting of three main components (see fig. 3.1):

- Metadata Servers (MDS)
- Object Storage Targets (OST)
- Clients

The replicated, failover metadata servers maintain the transaction-based file system changes, while the object storage targets are responsible for the actual file system I/O. Lustre takes advantage of using existing open technologies, like the Portals API from Sandia National Labs for network communication or XML and LDAP for configuration purposes.
In a first phase (Lustre Lite) the management of the metadata is done by one Metadata Server with failover, in the final version the metadata management can be clustered over many MDS.

3.1 Architecture

Lustre treats files as objects that are located through metadata servers (MDS). Metadata servers support all namespace operations, such as file lookups, file creation and file and directory attribute manipulation. The actual file I/O is directed to the object storage targets (OST), which manage the storage that is physically located on underlying object-based disks (OBD). The metadata servers keep a transactional record of file system metadata changes and the cluster status, supporting failover so that failures affecting one MDS do not affect the operation of the file system itself.
In the Lustre file system, like in many other file systems, every regular file, directory, symbolic link or special file is represented by a unique inode. The regular file inodes hold references to objects on OSTs that store the file data instead of references to the actual file data itself. In existing file systems, creating a new file causes the file system to allocate an inode. In Lustre (see fig.3.2), creating a new file causes the client to contact a MDS, which creates an inode

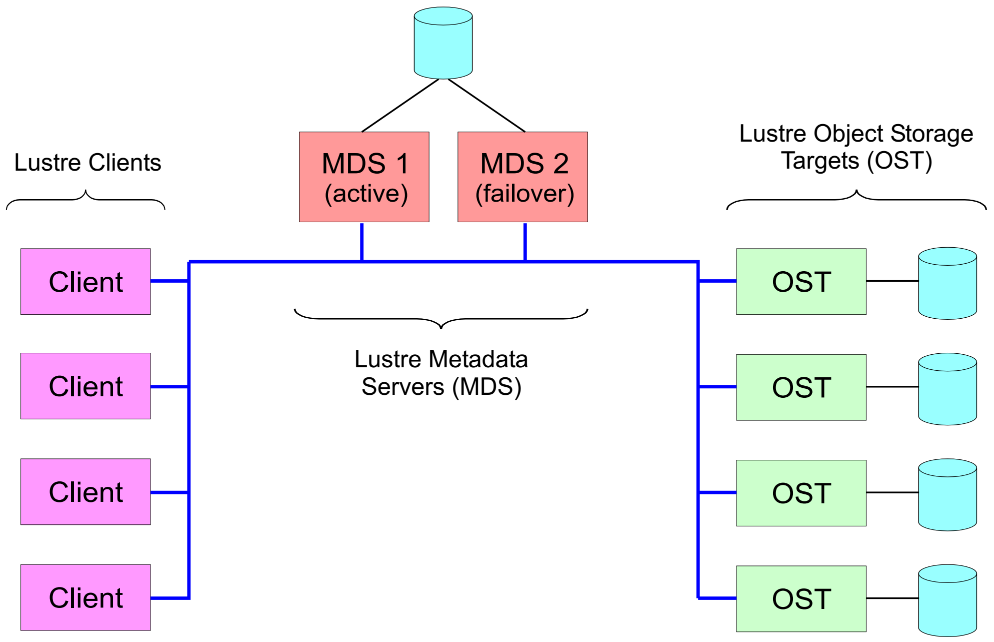


Figure 3.1: Overview of the Lustre File System

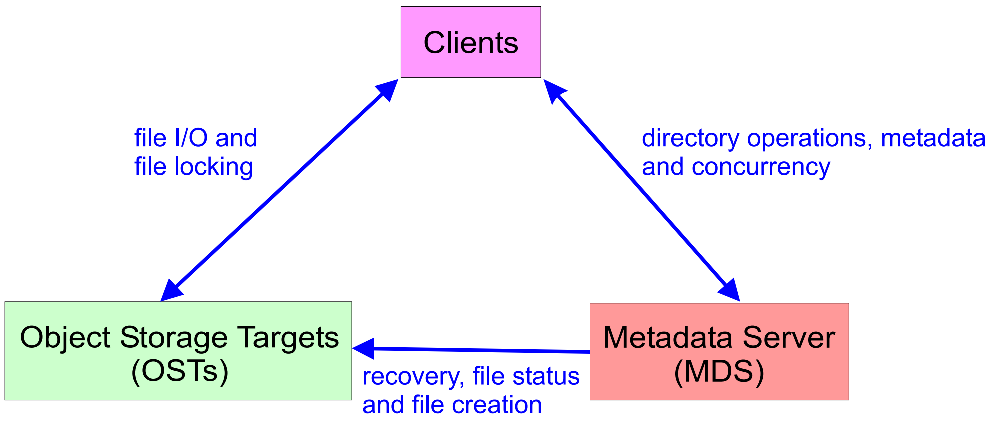

Clients
file I/O and file locking
directory operations, metadata and concurrency
Object Storage Targets (OSTs)
Metadata Server (MDS)
recovery, file status and file creation

Figure 3.2: Interaction between Lustre components

for the file and then contacts the OSTs to create objects that will actually hold the file data, which can be striped over several OSTs in a RAID pattern. Subsequent I/O to the newly created file is done directly between the client and the OST, the MDS is only updated when additional namespace changes with the new file are required.

3.2 Object Storage Target OST

Object Storage Targets handle all of the interaction between client data requests and the underlying physical storage, the Object-Based Disks (OBDs). This storage is not actually limited to disks because the interaction between the OST and the actual storage device is done through a device driver, a so-called Filter Driver[1]. This layered design provides a flexible way to add new storage to the Lustre file system. New OSTs can easily be added to the pool of OSTs a cluster's MDS is managing. Similarly, new OBDs can easily be added to the pool of OBDs associated with any OST.

[1]In the current Lustre release 1.0.2 the access to the physical storage is done through the Virtual File System (VFS) and the Filter Driver, which restricts the OST to use devices with a fi le system interface as OBD

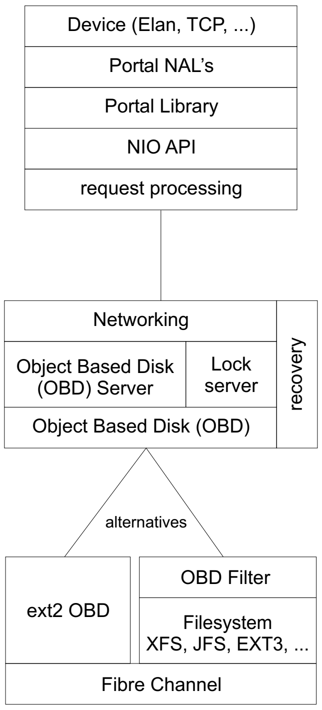


Figure 3.3: Object Storage Target (OST) Software Modules

3.3 Meta Data Server MDS

File system metadata is information about the files and directories that make up a file system. This information consists of the name of a file or directory, some status information, the targets of symbolic links, and so on. Lustre uses existing file systems (EXT3, ReiserFS,...) to store this information, which it accesses through the Linux Virtual File System (VFS) and an own Filter Driver. The Filter Driver provides functions like support for additional metadata (e.g. file striping information), transactions and callbacks.
Every file or directory in the Lustre File System is represented by a inode on the MDS. Additional metadata (like file striping information) is stored by the Filter Driver. In case of EXT3 as MDS file system, this metadata is stored in extended attributes (xattr), which can also be acessed through VFS functions. The ability to have full access to all metadata through the normal file system interface (VFS) is an ambition of Lustre ("Everything is a file system").

3.4 Client

The client is the user of the Lustre File System. It communicates directly with the MDS for metadata operations and with the OSTs for file I/O. If for example the client has to create a file (see fig. 3.6), it requests a lock from the MDS to enable a lookup operation on the parent directory and additionally tags this request with the intended operation, namely file creation. If the lock request is granted, the MDS then tries to execute the intended operation, creates the file and returns a lock on the new file instead on the directory, whereas in a conventional network file system the client would first lock the directory, do a lookup, create the file and then unlock the directory again.

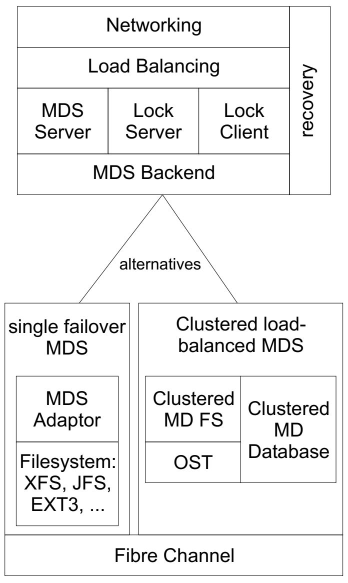


Figure 3.4: Metadata Server (MDS) Software Modules

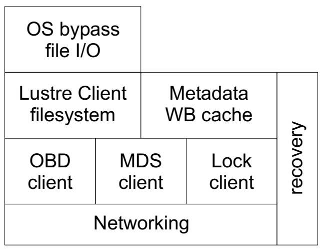


Figure 3.5: Client Software Modules

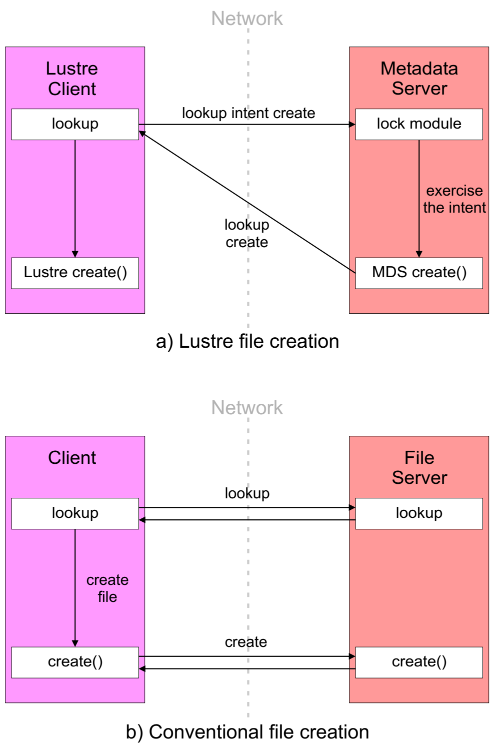


Figure 3.6: Lustre lookup intent

Chapter 4

Virtual File System (VFS)

4.1 Introduction

The Linux Virtual File System (VFS) provides an abstraction layer that hides the implementation of a specific filesystem behind a common layer. It handles all the system calls that are related to file and directory manipulation. One can transparently mount, umount and use different kinds of filesystems, the data itself can even be stored on a different machine, as it is done in the Unix Network File System (NFS). This Chapter discusses the aims, designs and properties of the VFS and focuses on the four main objects that are used: The inode, the super block, the file and the dentry.

4.2 Overview

The VFS was first introduced with BSD 4.4 (Berkeley Software Distribution) and was first implemented by Sun Microsystems which called it the `virtual-inode`, or `vnode` layer. The idea was to introduce a layer that abstracts the filesystem interface from a specific implementation, not by redesigning the whole filesystem, but by extending the model with filesystem specific parts in each object and filesystem specific functions that execute the operation. Files and directories are treated the same way, directories are just files that contain directory entries as data. This means that other filesystems like FAT which use a file allocation table to store the location of each file, have to convert their layout on the fly to the in memory structure of the VFS. The same applies to network file systems like the Network File System (NFS).
Every filesystem stores the data in a different way on a storage device, therefore it's not possible to have generic functions that create or manipulate files. Instead, the VFS uses function pointers to handle the system calls. Every filesystem needs to implement a specific subset of these functions to provide a minimal functionality. Let's look at an example.
Let's suppose that a user wants to create a directory. To accomplish this he invokes the system call `sys_mkdir`. This function call switches to the kernel context and calls the `vfs_mkdir` function provided by the VFS. The VFS does all the necessary checks that are not directly related to a specific filesystem but are necessary to fulfill all preconditions. This includes permission and consistency checks, allocation of necessary ressources, etc. After this tasks are finished, the VFS calls the `mkdir` function of the filesystem the new directory belongs to. Now, the directory is going to be created and can then be used.
One can think of the common file model as an object-oriented model. In fact, the linux kernel is not written in an object oriented language like `C++`, but in plain `C`. You can think of an object in `C` as a software construct that contains data and function pointers. Now, let's have a look at the four main object in the VFS.

4.3 Superblock

The superblock is a special object, that contains general information and statistics of the file system. This includes the blocksize of the filesystem, the magic number, etc. In addition it contains a pointer to the root inode object. For each mounted file system, there exists exactly one instance of this object in the kernel memory. Usually the superblock has an exact counterpart on the storage media. Attached to it is a structure of function pointers that provides the necessary superblock functionality like creating and deleting inodes. Let's briefly describe the most important operations:

read_inode(inode) Fills the fields of the inode object whose pointer is passed to this function with the corresponding entry on the disk. The inode number of the inode to be read is already filled in this structure. The function structures are as well filled in.

dirty_inode(inode) This function is invoked, when the inode is marked dirty. Dirty means, that the inode has changed but is not yet written back to disk.

write_inode(inode) Is called, when the system wants to write an inode back to the disk.

put_inode(inode) This is called, if the inode is no longer used and can be freed. Usually a filesystem does not need to do anything here.

delete_inode(inode) Deletes the inode on the disk, the VFS inode and any data blocks that contain data of the inode. This function has to call the VFS function `clear_inode` to indicate to the system, that this inode is no longer useful. `clear_inode` calls in turn the `clear_inode` function of the filesystem.

put_super(super) Releases the super block object. This function is called while unmounting the filesystem.

write_super(super) Updates the content of the superblock on the disk.

statfs(super, buf) Fills the structure `buf` with statistics of the mounted filesystem.

clear_inode(inode) This function does the same as `put_inode`, but also releases any allocated resources. Note that there is usually nothing to be done on the storage device.

Most of these functions are used within all of the common filesystems, but several are not. If a filesystem doesn't want to provide a specific functionality, it can simply set the function pointers to `null`. It is interesting, that no `read_super` function exist, but that makes perfectly sense, considering that the superblock object does not yet exist when this function should be invoked the first time, namely during the filesystem mount operation.

4.4 Inode

All information needed by a filesystem to handle a file is stored in a data structure called inode. You can change the name of a file, but the inode is still the same, therefore the name of a file is not stored in the inode itself, but in a dentry object (see 4.6). Each inode contains an unique number that identifies it. An inode object stores a lot of information, for example all data you get, if you execute a `stat filename`. To improve disk performance the VFS tries to cache as much data as it can. All of the cached inode objects are in one of three lists.

1. The list of unused inodes contains all inodes that have been put and are no longer of use to any process. Their usage count is set to zero and they are not marked as dirty. As a consequence, they can be reused if somebody needs an inode object, but they can also be reactivated if a process reads the inode this inode object represented. In short term: This list acts as a disk cache.

2. The list of in use inodes contains all inodes that are currently used by some process. Their inode count is above zero and they are not marked as dirty.

3. The list of dirty inodes. These inodes are not yet synced with the storage device.

To improve the speed of an inode lookup operation there exists a hash table the kernel can use if it knows the inode number and the address of the corresponding superblock object. Similar to the superblock, an inode object has an array of function pointers that can be used to execute the system calls that are mapped by the VFS to inode operations. Note that not every inode contains the same operations. For example it makes no sense that an inode attached to a file can create other files. This particular operation is reserved for inodes that are attached to a directory.

create(dir_inode, dentry, mode) Creates a new disk inode for a regular file with mode `mode` and associates it with the `dentry`.

lookup(dir_inode, dentry) Searches through the directory `dir_inode` for a particular file whose filename is included in `dentry`.

link(old_dentry, dir_inode, new_dentry) Creates a hardlink. `old_dentry` describes the old file, `new_dentry` the new file and `dir_inode` represents the parent directory of `new_dir`. Please note that hardlinks to directories are not allowed, because that could result in a recursive directory structure the kernel is unable to handle.

unlink(dir_inode, dentry) Deletes the file `dentry` that is located in the directory `dir_inode`.

mkdir(dir_inode, dentry, mode) Creates a new directory in the directory `dir_inode` with mode `mode` and attaches the `dentry` to it. The name of the directory to be created is already filled in `dentry`. Additionally the two standard entries `"."` and `".."` must be created in the new directory.

rmdir(dir_inode, dentry) Removes the directory associated with `dentry` from the directory `dir_inode`.

mknod(dir_inode, dentry, mode, rdev) Creates a new disk inode for a special file whose name is included in `dentry` in the directory `dir_inode`. The `mode` and `rdev` specify the filetype respectively the device's major number.

rename(old_dir_inode, old_dentry, new_dir_inode, new_dentry) Moves a file represented by `old_dentry` from the directory `old_dir_inode` to `new_dir_inode`. The new filename is included in `new_dentry`.

readlink(dentry, buffer, buflen) Copies into `buffer` the file pathname corresponding to the symbolic link specified by `dentry`. `buflen` contains the size of the `buffer`. Note, that `buffer` is a userspace buffer. If it is too small to keep the whole name, the filename should be truncated.

follow_link(dentry,nameidata) Gets the filename to which the symbolic link in `nameidata` points and copies the filename into the structure `nameidata`. To prevent endless loops[1], the maximal recursion depth is set to eight.

truncate(inode) This function is used to decrease or increase, if the filsystem support this, the size of the inode `inode`. The field `i_size` of `inode` is already set to the new size of the file. This function is also reponsible to free any superfluous disk pages.

getxattr(inode, name, buf, buf_size) This function fetches the extended attribute with name `name` attached to the inode `inode` and copies the attribute into `buf`. The length of `buf` is specified by `buf_size`.

listxattr(dentry, buf, buf_size) Reads all the names of the extended attributes that are attached to `dentry` and copies then into `buf`. The buffer size is specified by `buf_size`.

removexattr(dentry, name) Deletes the extended attribute with name `name` attached to `dentry`.

setxattr(inode, name, value, value_len, flags) This assigns the extended attribute `name` with the inode `inode`. The value of the extended attribute is given in `value` and has the length `value_len`. The `flags` determine if an existing attribute with the same name should be replaced or not.

[1]The symbolic link can point to an another symbolic link

Officially the extended attributes were introduced with the kernel 2.6, but because Lustre depends on them, we included the extended attribute operations in this short description. The return value of theses functions is always the size of the extended attribute or an error code. If the value of `buf_size` while calling `listxattr` or `getxattr` is set to zero, nothing will be copied into `buf`, but the size of the extended attribute will be returned.

4.5 File

A file object provides all the necessary information the kernel needs to handle file requests. Such a file object is created, when a file is opened. In fact, a userspace file handle is an identifier for a kernel file object. This structure has no counterpart on a storage device and does therefore not need a dirty flag. The most important member in this structure is the file pointer, it contains the actual reading and writing position of the file. Like the other two already described objects, the file object contains a structure of function pointers, that every filesystem has to implement. We like to describe some of the most important functions:

llseek(file, offset, origin) Sets the file position of the file `file` to `offset` relative to `origin`. `origin` can be the beginning, the end, or the actual position of the file. There exists a standard implementation for this function.

read(file, buf, count, offset) Reads `count` bytes from `file` beginning at the absolute position `*offset` (usually the current file position) and copies the data into `buf`. Afterwards, the `*offset` is incremented.

write(file, count, buf, offset) Writes `count` bytes from `buf` into the file `file` beginning at the absolute position `*offset` (usually the current file position). Afterwards, the `*offset` is incremented.

readdir(filp, dirent, filldir) This function is used to get the entries of the directory (`dirent` the file pointer `filp` is attached to. It is somehow a special function and will be dicussed in detail in chapter 4.7.

ioctl(inode, file, cmd, arg) Sends a command `cmd` to the underlying hardware. This procedure applies only to device files.

mmap(file, vma) Maps the file `file` into the memory area described by `vma`. Reading and writing into a file can then be performed by working with the memory area `vma`. Every filesystem that wants to provide support for executables has to implement this function, which is used to load the executables.

open(inode, file) Opens a new file by attaching the newly created `file` to `inode`.

release(inode, file) Releases the file object. This functon is called, if the file usage count hits zero.

fsync(file, dentry) Writes all cached data belonging to the `file` back to disk.

readv(file, vector, count, offset) This function does the same as `read` but instead of a buffer we have in this case `count` numbers of buffers arranged in `vector`.

writev(file, vector, count, offset) This function does the same as `write` but instead of a buffer we have in this case `count` numbers of buffers arranged in `vector`.

4.6 Dentry

A directory entry (dentry) primarily connects an inode with a name. The kernel creates for each directory component a new dentry object and connects it with the corresponding inode. For example when looking up the file `/etc/fstab` the kernel creates for each of the components `etc` and `fstab` a separate dentry object. To increase performance, the kernel manages a dentry cache. It tries to keep as much dentry objects as possible in the memory without harming other parts of the system. Because there's always an inode connected to a dentry[2] the dentry

[2]Except the dentry is unused or the inode attached to the dentry has already been deleted

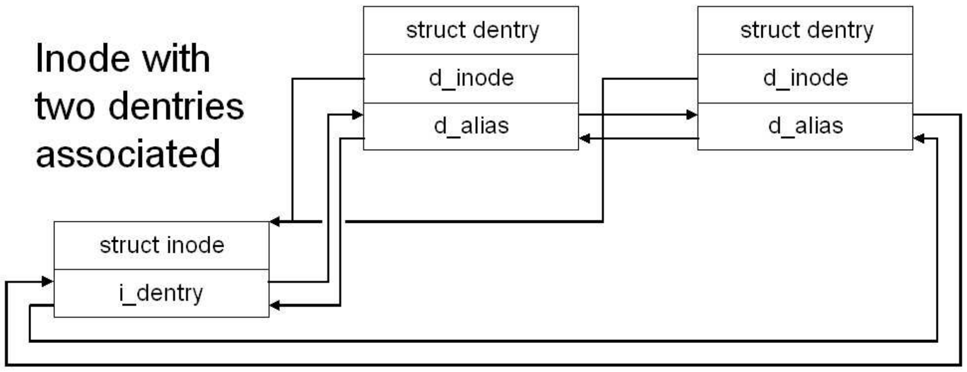


Figure 4.1: Inode with associated dentry

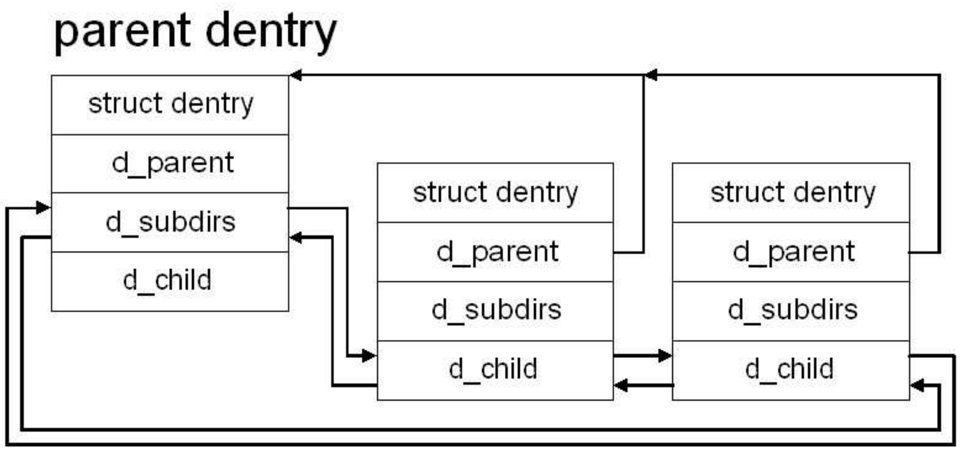


Figure 4.2: Parent dentry with children

cache controls the inode cache. To improve lookup performance, the dentries are arranged in a hash table. The size of this table depends on the amount of installed memory. The operations connected to the dentries are called dentry operations. Usually their task is not to provide filesystem functionality, but to manage the dentries, although a filesystem can exchange the default functions with its own.
In figure 4.1 the connection between an `inode` and its corresponding `dentry`s is illustrated. `d_inode` of a `dentry` points always to its associated `inode`. An `inode` can find it's associated `dentry`s by dereferencing the pointer `i_dentry` and using the `d_alias` field of the `dentry`s.
In figure 4.2 it's illustrated how a directory is linked with its directory entries on the `dentry` level. The parent `dentry` represents the directory and the two other `dentries` linked together with the `d_child` field represent the directory entries. Note, that the entry `d_subdirs` is badly named, as it is not a only a member of the linked list of all subdirectories but all entries of the directory.

4.7 Readdir system call

The purpose of the readdir system call[3] is to get the next entry of a directory. It calls the `vfs_readdir` function which in turn calls the appropriate inode `readdir` function, which is defined as follows:

```
static int
```

[3]The original readdir call is unable to handle more than one directory entry at once, therefore the system call getdents was introduced, which gets as a parameter a user supplied buffer, the system can fi ll with readdir entries.

```
        sqlfs_readdir (struct file * filp , void * dirent , filldir_t filldir);
```

The current position in the directory is determined by the file position of the file that represents the directory. To be independent of the physical layout and to provide the kernel itself the ability to get a directory listing, a callback mechanism is used for the inode `readdir` function. The VFS provides with this function the callback `filldir` that assembles a directory entry structure (`dirent`) from the given parameters. The return value of `filldir` indicates if more entries are requested or not. Such a `dirent` has the following layout:

```
struct dirent
{
        long d_ino;                     /* inode number */
        off_t d_off;                    /* offset to next dirent */
        unsigned short d_reclen;        /* length of this dirent */
        char d_name [NAME_MAX+1];       /* file name (null-terminated) */
}
```

The `offset` is used to locate the next readdir entry in the current buffer. Note, that you cannot calculate the beginning of the next `dirent` by using `d_reclen`, because there might be an empty space between two entries.

Chapter 5

Our Approach

Figure 5.1 gives a overview over the design of our approach. According to the assignment in chap. 1 the Lustre Metadata Server should use a database instead of a usual file system to store it's metadata. Unfortunately, there exists no abstraction layer for the MDS like the OBD-layer for the OST, the MDS is designed to store it's data on a usual journaling Linux File System (ext3, reiserfs, xfs, ...). For accessing the file system, the MDS uses the VFS-layer and a Lustre-specific File System Filter for functions, which the VFS does not support (e.g. transactions, additional metadata, callbacks). The file system to be implemented, namely *sqlfs*, therefore has to implement the Linux File System interface (mainly the three interfaces `superblock_operations`, `inode_operations` and `file_operations`) and the functions used by the Filter Driver *fsfilt_sqlfs*.

The access to the database can not be implemented in the sqlfs kernel module, because the libraries used for connecting to the database don't support kernel mode. The database link is therefore established through user processes, the *DB-Clients*, which act as proxy for the kernel module. Communication between the kernel module and the DB-Clients is done by a queue which is accessible by ioctls. The DB-Client enters an ioctl which is blocking, until the kernel module has placed a command in the queue which the waiting DB-Client then grabs and executes. Subsequently, the DB-Client enters the ioctl again, returns the result of the preceding and then waits for the next command.

Transactions in the file system are directly mapped to transactions on the database. Database transactions are bound to a db-connection, and as each DB-Client maintains an own db-connection the file system transactions are bound to the respective db-client. It is not supported by Linux, that file system transactions occur between different processes, and that the process which executes a transaction wants to do some file system manipulation, which is not part of the transaction. By knowing this, running transactions can be identified through the process ID. DB-Clients, which are in a open transaction, accept further commands only from the process which opened the transaction, until the transaction is commited. By this mechanism it is granted,

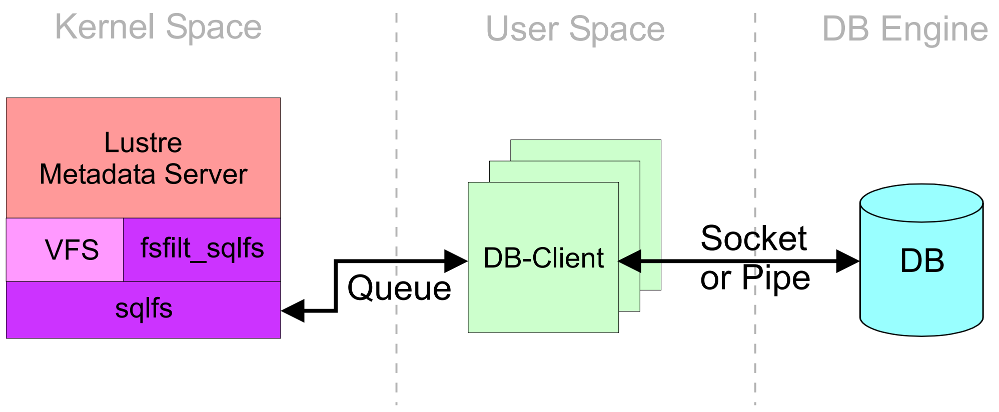


Figure 5.1: Overall design overview

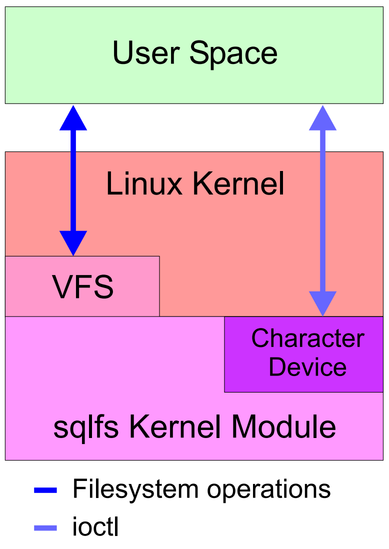


Figure 5.2: Kernel module with interfaces

that the process, which executes a transactions, gets always the same DB-Client, therefore the same db-connection and is therefore correctly mapped to a database transaction.

5.1 Kernel module

The kernel module, as mentioned above, provides the complete kernel part of the sqlfs filesystem. Basically, it implements all the necessary superblock, inode and file operations, except `mmap`. This procedure is basically used to support the execution of files, but as we use a character device instead of a block device, it would need an unreasonably amount of work to to implement this. Besides, the sqlfs filesystem was designed to interact with the Lustre MDS (see 3.3) which does not depend on `mmap`.

5.1.1 VFS Interface

The main part of the kernel module implements all the super block, inode and file operations explained in chapter 4. In addition, a few extensions were introduced. One special property is that the sqlfs filesystem is always synced. Every operation is physically written to disk, before the system returns back to userspace. This behaviour has the advantage, that the filesystem is always in a consistent state even if it crashes while performing some operations. However it has the disadvantage that the delete operations are quite slow compared to other filesystems.
The other approach of deleting a file, as it is performed on most disk based filesystems, is to perform the delete operation first in the in memory structures and cached disk blocks, and then sync them later.
To improve the delete performance of the sqlfs it would be possible to remove (physically) the directory entry and then delete the rest later asynchronously, but this is not implemented yet.

As a consequence of the synced operations, it's possible that the VFS tries to delete a file more than once even if it's already deleted. To catch such unnecessary operations, the flag `SQLFS_INODE_DELETED` was introduced and attached to the filsystem specific part of the `inode` structure. This flag is set, if the inode is already physically deleted, and it's therefore not necessary to perform anything on the database.

A second flag called `SQLFS_BLOB_ALLOCATED` is used to indicate that a blob (binary large object) which is used to store the content of the file, has been allocated on the datase. This flag helps to speed up the deletion of a file, because it can give the database a hint whether the blob table needs to be processed or not.

5.1.2 Database Client Interface

The kernel module provides a character device called `/dev/sqlfs` to the database client. To identify which character device belongs to which kernel module, the kernel dynamically assigns a major number to it. If a specific major number is needed, the user can pass it as an option while calling `insmod` (e.g. `insmod sqlfs sqlfs_major=254`). The corresponding device file can be created with the command `mknod c 254 0`. This has to be done just once. Although a character device appears to a user space program as an ordinary file, the `read` and `write` operations are not supported. Instead, all communication is done via the I/O controls (`ioctl`, see 4.5). As a parameter, an integer or a pointer can be passed to the kernel and vice versa. In our case we pass a pointer to a `sqlfs_comm` structure which is defined as follows.

```
struct sqlfs_comm {
  int cmd;                         /* command number */
  pid_t transaction_id;            /* transaction id */
  int err;                         /* return value from db client */

  /* inline data section */
  struct sqlfs_inode inode;        /* complete sqlfs_inode */
  __u32 parent_inode_ino;          /* inode number of parent inode */
  struct sqlfs_qstr d_name;        /* name of directory entry */

  int data_size;                   /* size in bytes of data */
  void *data;                      /* data pointer used in sqlfs_readdir */
};
```

This structure is defined in a generic way to support all the variating commands. `cmd` contains the command that should be executed. These are all the inode, superblock and file operations explained in chapter 4. The `transaction_id` is used to provide a mapping between the processes that operate on the filesystem and the DB-Clients that actually execute the commands. Each such transaction contains a return value stored in `err` the kernel module can use to take appropriate actions upon errors. Embedded in `sqlfs_comm` is a complete `sqlfs_inode` structure. Such an inode represents the physically stored subset of a VFS inode. Note, that this structure is only filled in if it's necessary to have the whole inode data available. For several operations it is essential to have the parent inode number (e.g. `mkdir`) or the name of a directory entry. These two elements are stored in `parent_inode_ino` respectively `d_name`. To support operations depending on a variable size buffer like `read` or `write` a void pointer `data` was introduced. The kernel module or the DB-Client can attach a buffer of size `data_size` to it, which is then copied into the approriate kernel or user space. Copying the whole data block is suboptimal, but it simplyfies the design.
Now, let's have a look at the used `ioctl` commands:

```
IOCTL_SQLFS_FIRST_CONNECT          /* Register this DB-Client */
IOCTL_SQLFS_COMM                   /* Return result and wait for the next
   command */
IOCTL_SQLFS_DISCONNECT             /* Kernel tells DB-Client to disconnect
   himself */
IOCTL_SQLFS_COPY_TO_USER           /* Copy data from user space to kernel space
   */
IOCTL_SQLFS_CLEAR_STAT             /* Clear statistics of kernel module */
IOCTL_SQLFS_READ_STAT              /* Get statistics of kernel module */
IOCTL_SQLFS_DISCONNECT_DB          /* Tell kernel module to disconnect all DB-
   Clients */
IOCTL_SQLFS_SET_WRITE_RECORD       /* Fake write record commands or not */
IOCTL_SQLFS_FORMAT_DB              /* Tell kernel module to format the sqlfs */
IOCTL_SQLFS_FORMAT                 /* Format sqlfs */
```

IOCTL_SQLFS_FIRST_CONNECT

The dbclinet uses this `ioctl` to register himself in the kernel module, and waits until the command queue contains an element he can fetch and process. The main difference between this call and `IOCTL_SQLFS_COMM` is, that `IOCTL_SQLFS_COMM` contains always a processed command element that is returned to the kernel, whereas `IOCTL_SQLFS_FIRST_CONNECT` has none. In addition, the kernel module maintains the number of connected DB-Clients by means of this `ioctl`.

IOCTL_SQLFS_COMM

This `ioctl` is used during the normal processing of a command. It's interesting to note that the DB-Client does not control the kernel, as it is the usual case, but the kernel controls the DB-Client. The DB-Client calls a `ioctl`, receives commands from the kernel via the command queue, copies one of them into user space and returns out of the ioctl. Afterwards, the DB-Client processes the `sqlfs_comm` structure. If the `data` pointer is set, the kernel has additional data waiting that has to be copied into user space. This has to be accomplished by the DB-Client, because a user space process has to allocate the necessary memory. The copy operation is done by executing an another `ioctl` this time with the command `IOCTL_SQLFS_COPY_TO_USER`.
After executing the received command, the DB-Client copies the results back into the `sqlfs_comm` structure and calls `IOCTL_SQLFS_COMM` to return the results. If the `data` pointer is set, the kernel allocates memory and copies the data into kernel space. The obtained result can then be put back into the queue.

IOCTL_SQLFS_COPY_TO_USER

The `ioctl` command is used to copy any data attached to the `data` pointer into user space. The user space process has already allocated enough memory to hold the entire buffer.

IOCTL_SQLFS_CLEAR_STAT

The kernel module maintains a statistic of all called inode, super block and file functions. This `ioctl` resets the statistics.

IOCTL_SQLFS_READ_STAT

This `ioctl` copies the above mentioned statistics into a user space buffer. The structure of the data is declared in the file `sqlfs_fs.h`.

IOCTL_SQLFS_DISCONNECT_DB

This `ioctl` is called by a special helper program with name `stopping` that orders the kernel to diconnects all DB-Clients. As a result of this, the `sqlfs` kernel module can be unloaded, provided that the filesystem is unmounted.

IOCTL_SQLFS_DISCONNECT

The kernel sends this `ioctl` command to a DB-Client to initiate its disconnection.

IOCTL_SQLFS_SET_WRITE_RECORD

Writing binary data into a database is usually not very fast, therefore it's possible to fake all file write operations by calling this `ioctl`.

IOCTL_SQLFS_FORMAT

Format the sqlfs filesystem, but only if it is not mounted.

5.1.3 Command queue

The command queue is the interface between the DB-Clients and the sqlfs Kernel Module. The queue-scheme allows a certain load-balancing by using more than one DB-Client. Commands are packed into queue elements which are placed in the queue, where the DB-Client can fetch them through the `IOCTL_SQLFS_COMM` ioctl. After placing the element in the queue, the initiator waits on the element. It will be woken up again after a DB-Client having fetched the command, executed it and wrote the result back in the queue element. After waking up the waiting process, the DB-Client fetches the next command or waits on the queue, until a command arrives. If the DB-Client is in a open transaction, it doesn't wait for any new command, but for a command of it's master process, the process, which started the transaction. Figure 5.3 illustrates the functionality of the queue.
A command queue element is declared as follows:

```
struct sqlfs_cmd_queue_element {
  struct sqlfs_cmd_queue_element *next; /* pointer to next element in query
                                           queue, protected by head_mutex */
  struct sqlfs_comm sqlfs_comm_request; /* data for/from dbclient */
  int done;                             /* 1 if command is processed */
  struct semaphore element_mutex;       /* semaphore protects all fields of
                                           element except 'next' after
                                           insertion in queue */
  wait_queue_head_t element_cond;       /* condition variable to wait on
                                           element */
  int reuse;                            /* indicates whether the element
                                           will be re-used, needed for
                                           transactions */
  int waiting_for_reuse;                /* if the element is being reused,
                                           this is 1 if the element is
                                           waiting */
  struct sqlfs_cb_element* cb_element;  /* pointer to first callback
                                           function */
};
```

The command queue hosts two lists, one for busy elements and one for waiting elements. Busy elements are elements which are fetched by a DB-Client and currently being processed or waiting elements which are in a running (uncommited) transaction. If the `reuse` flag is set, the element does not leave the busy-list after processing. This is used for transactions, to prevent the element (and the DB-Client) being used by other processes than the owner of the transaction. The flag `waiting_for_reuse` indicates whether the reused element is currently being processed by a DB-Client or waiting for a new command.
The following methods build the interface to the command queue:

```
int sqlfs_cmd_queue_init(struct sqlfs_cmd_queue **pqueue);
void sqlfs_cmd_queue_add(struct sqlfs_cmd_queue *queue, struct
   sqlfs_cmd_queue_element *element);
struct sqlfs_cmd_queue_element *sqlfs_cmd_queue_wait(struct sqlfs_cmd_queue *
   queue);
struct sqlfs_cmd_queue_element *sqlfs_cmd_queue_find_busy(struct
   sqlfs_cmd_queue *queue, pid_t transaction_id);
void sqlfs_cmd_queue_return_processed(struct sqlfs_cmd_queue *queue, struct
   sqlfs_cmd_queue_element *element);
int  sqlfs_cmd_queue_destroy(struct sqlfs_cmd_queue *queue);

/* sqlfs query queue element -- admin functions */
int sqlfs_get_cmd_queue_element(struct sqlfs_cmd_queue_element** elem, int*
   is_new_element, pid_t transaction_id);
int sqlfs_cmd_queue_new_element(struct sqlfs_cmd_queue_element **pelement);
void sqlfs_cmd_queue_wait_on_element(struct sqlfs_cmd_queue_element *element)
   ;
void sqlfs_cmd_queue_destroy_element(struct sqlfs_cmd_queue_element *element,
    int is_new_element);
void sqlfs_cmd_queue_wait_for_reuse(struct sqlfs_cmd_queue_element *element);
```

sqlfs_cmd_queue_init(pqueue) Initializes a new command queue.

sqlfs_cmd_queue_add(queue, element) If `element` is a reused element, the DB-Client waiting on the element is woken up and begins processing the command. If the element is a new element, it is added to `queue` and a free DB-Client is woken up to process the command.

sqlfs_cmd_queue_wait(queue) Called by the DB-Client to wait for a new command. Fetches a new command, if available, or waits until a new command queue element is added to `queue`. Moves the fetched element to the busy list.

sqlfs_cmd_queue_find_busy(queue, transaction_id) Searches for a busy element in `queue` with `transaction_id` and returns it, if found, `null` else.

sqlfs_cmd_queue_return_processed(queue, element) Called by the DB-Client after having processed the command to wake up the waiting process. If `reuse` is set on `element`, the DB-Client afterwards calls `sqlfs_cmd_queue_wait_for_reuse` to wait on the element for the next command, otherwise `sqlfs_cmd_queue_wait` to wait for a new command.

sqlfs_cmd_queue_destroy(queue) Destroys `queue`.

sqlfs_get_cmd_queue_element(elem, is_new_element, transaction_id) Used for getting a command queue element for `transaction_id` (usually the process ID). After the call, `elem` points to the new element and `is_new_element` tells whether the element was newly created or is a reused element.

sqlfs_cmd_queue_new_element(pelement) Called by `sqlfs_get_cmd_queue_element` to create a new element when no reusable one exists.

sqlfs_cmd_queue_wait_on_element(element) After adding `element` to the queue with `sqlfs_cmd_queue_add` this method is called to wait until the command is processed.

sqlfs_cmd_queue_wait_for_reuse(element) Called by the DB-Client to wait on `element` for reuse instead of waiting for a new command queue element.

sqlfs_cmd_queue_destroy_element(element, is_new_element) Destroys `element` if `is_new_element` is set. If not, just the possibly allocated memory buffer `element->sqlfs_comm_request.data` is freed. Whether the element was new or reused was returned `sqlfs_get_cmd_queue_element` when getting the element.

5.1.4 Readdir cache

Other filesystems that use block devices (e.g. harddisks) to store their data can take advantage of the block caching mechanism of the VFS. Every block that is read from the block device is cached in a similar way to the inode cache. The system tries to cache as much blocks as it can without harming the other components of the system. To achieve this, the kernel dynamically puts the cached blocks back to disk.
The sqlfs does not use a block device, hence it cannot take advantage of the kernel block caching mechanims, therefore we implemented a similar caching mechanism. Instead of using cached disk blocks, we allocate and fill memory pages and attach them to the inode that represents the wanted directory. A readdir call can now parse the memory pages instead of querying the database. An another benefit is that we can take advantage of the previously mentioned dynamic freeing mechanism. While freeing an inode, the VFS subsystem calls the inode `clear_inode` operation, we can use to free our allocated memory pages.
Instead of using a linear list of memory pages, as it would be the simplest approach, a hash table in combination with a linear list was chosen, in order to find the next entry point reasonably fast.
The basic allocated object used is a memory page (4096 bytes on the x86 system). This design decision was made to ensure a fast allocation of a used memory objects. Besides, the size of 4096 bytes appears reasonable. The allocated memory page is then casted to the needed

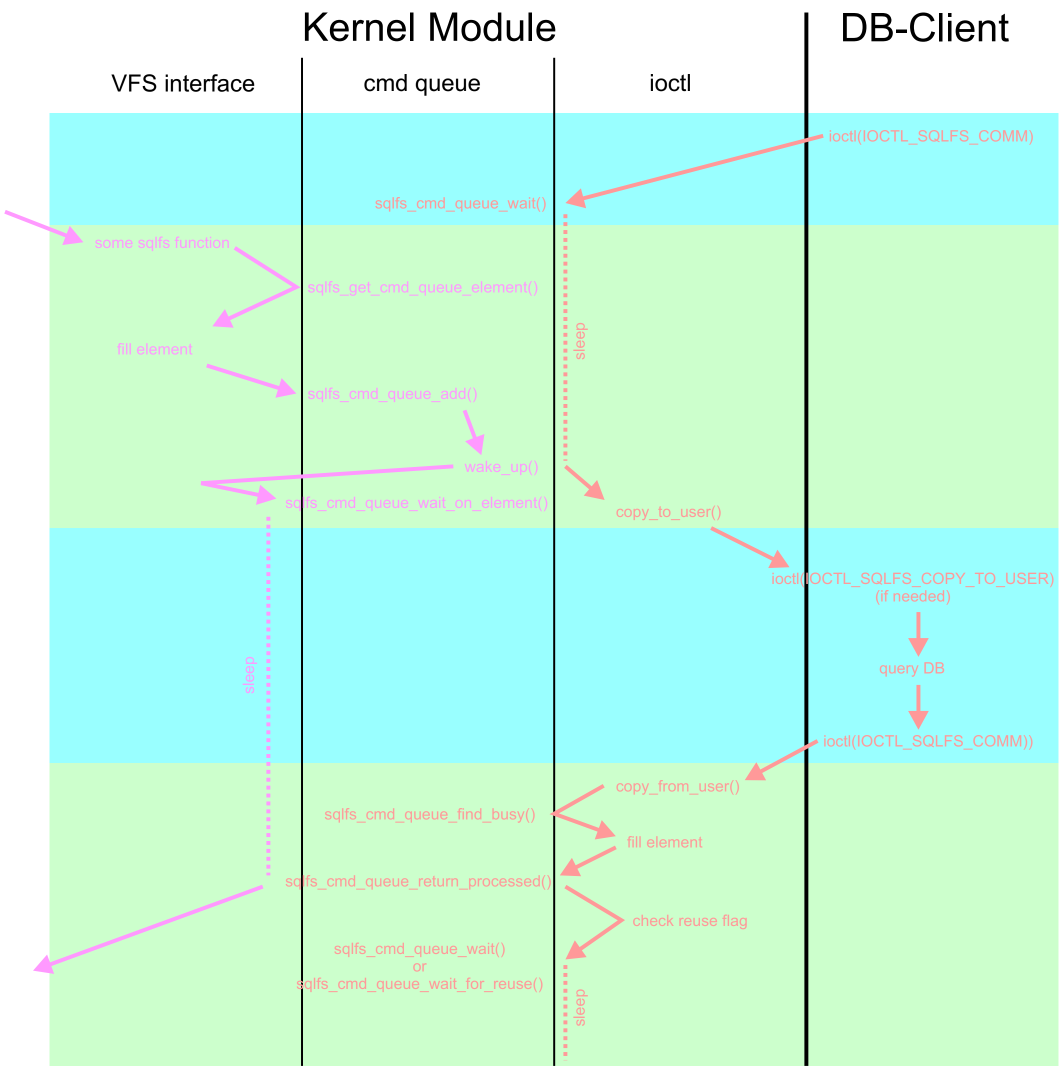


Figure 5.3: The command queue

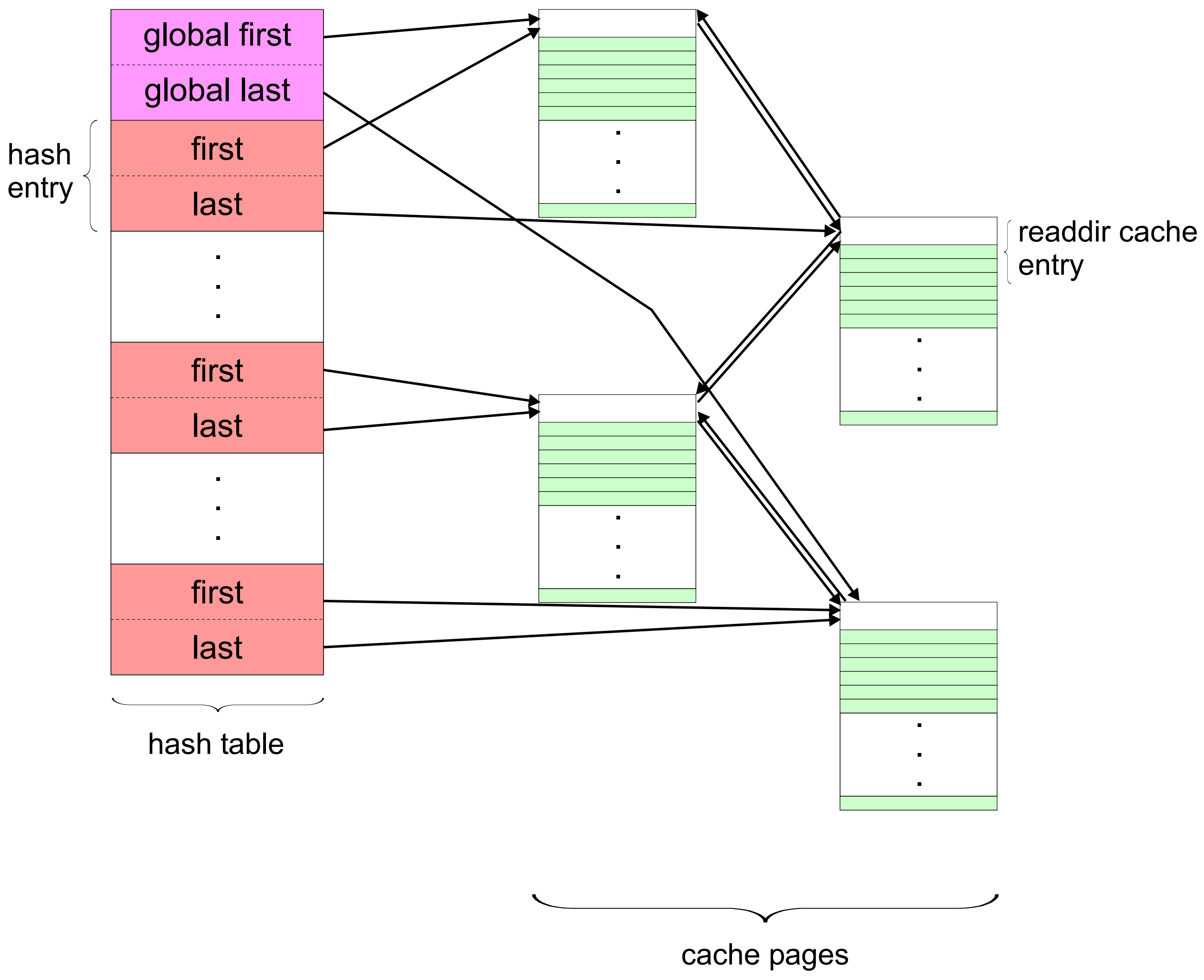


Figure 5.4: Overview of the readdir cache

structure. This can be the hash table itself or a readdir cache page that stores the directory entries.
Let's see what kind of structures are used, beginning with the smallest, a `sqlfs_readdir_cache_entry`.

```
struct sqlfs_readdir_cache_entry {
  __u32 inode;                    /* Inode number */
  __u16 rec_len;                  /* Directory entry length */
  __u8  name_len;                 /* Name length */
  __u8  file_type;                /* Type of the entry, e.g. file, directory,
     pipe, ...
  char  name[SQLFS_NAME_LEN];     /* File name */
};
```

Such a readdir entry desribes one entry of a directory, e.g. a file, and contains the `inode` number, the `name` with length `name_len` and the `type` of the described object. The type is stored to prevent unnecessary lookup operations, but it is not yet used. In order to properly support the Lustre `fs_readpage` operation (see 5.4.1 our structure corresponds exactly to the ext2 readdir structure. Please note, that the `name` element does not have the full `SQLFS_NAME_LEN` number of bytes allocated, but only as much as needed. `rec_len` is used to find the next directory entry, but it is important to know that there can be an empty space between two entries, because the deletion of an entry is done by merging the current entry with the previous one.
All these entries are stored in one or several `sqlfs_readdir_cache_page` structures.

```
/* readdir cache page, fits in one memory page */
/* Contains a variable number of "sqlfs_readdir_cache_entry" elements */
struct sqlfs_readdir_cache_page {
  struct sqlfs_readdir_cache_page *next;        /* Pointer to next cache page
      */
  struct sqlfs_readdir_cache_page *previous;    /* Pointer to previous cahe
     page */
  int num_entries; /* Number of valid sqlfs_readdir_cache_entry elements in
     this page */
  int entry_sizes;  /* Size of stored data in this page. */

  /* Pointer to the first struct sqlfs_readdir_cache_entry in this memory
     page */
  struct sqlfs_readdir_cache_entry *first_entry;
  /* Pointer to the last struct sqlfs_readdir_cache_entry in this memory page
      */
  struct sqlfs_readdir_cache_entry *last_entry;

  /* The rest of the memory page contains the readdir_cache_entries */
};
```

As mentioned above this structure is cast to a memory page. The pages are linked among themselves in a doubly linked list (`next` and `previous`), whereas the `previous` pointer of the first page and the `next` pointer of the last past are set to `null`.
The first readdir element in this structure cannot be found statically, because it can be deleted sometime. Instead the first and the last element in this cache page can be found with the pointers `first` respectively `last`. The latter is not necessary, but it improves the insertion speed of an another readdir entry. In addition, the number of valid entries in this page is stored in `num_entries` and the real size of all entries (without any gaps) is saved in `entry_sizes`. Note, that new elements are always inserted at the end of a memory page, primarily to provide a fast insert operation. If a `sqlfs_readdir_cache_page` becomes empty, because all entries were deleted, the cache page is released.
All these `sqlfs_readdir_cache_page` are attached to a hash table. This table is of type `sqlfs_readdir_hash_table`.

```
/* Hash table of a readdir table.
   We never allocate this structure directly, we allocate instead one memory
     page
   and cast the received pointer to a (sqlfs_readdir_hash_table *). The size
     argument
   in the variable "page" is therefore only a hint for the reader to
     determine the
   array size.
   The inode number is used to calculate a hash value. This value is
   used as an index into the hash table.
   */
struct sqlfs_readdir_hash_table {
  /* pointer to first and last used cache page */
  struct sqlfs_readdir_cache_page *first;
  struct sqlfs_readdir_cache_page *last;

  /* hash table entries */
  struct sqlfs_readdir_hash_table_entry entry[SQLFS_READDIR_HASHTABLE_SIZE];
};
```

Once again this structure is not directly allocated, but cast to a memory page. The a hash table is used to improve the lookup speed of a particular element. This is necessary because the readdir call is reentrant (see chapter 4.7). `first` points to the first page that is allocated in the linked list of `sqlfs_readdir_cache_pages` and `last` to the last page. This avoids a scan through the hash table to find the first allocated page.

The rest of the memory page is filled with `sqlfs_readdir_hash_table_entries`.

```
/* This structure defines an entry of the hash table.
struct sqlfs_readdir_hash_table_entry {
  /* Points to the first sqlfs_readdir_cache_page that
  contain the sqlfs_readdir_cache_entries that hash to this value */
  struct sqlfs_readdir_cache_page *first;
  /* Points to the last sqlfs_readdir_cache_page that
  contain the sqlfs_readdir_cache_entries that hash to this value */
  struct sqlfs_readdir_cache_page *last;
};
```

This structure represents an entry of the hash table. As the names indicate, the pointers `first` and `last` point to the first and last cache page that belong to this hash table entry.
To illustrate how this hash table works, let's see how it is filled up while performing a readdir. First of all, the readdir operation fetches all entries of the queried directories from the database. For each of these entries, a hash value is generated that determines to which entry of the hash table bucket the element belongs to. Then the last readdir cache page belonging to this bucket is fetched, or if no page exists, a new one is allocated, and the element is inserted into it.
As we already know, the `readdir` call is reentrant, it fetches just a certain number of entries at once, and get's the rest by using the file position of the file representing the directory. It is now necessary that the kernel is able find a specific entry to continue a readdir call, even if someone added or removed a file. The chosen solution simply uses the file pointer to destinguish wether the actual readdir call should start at the beginning of the directory or continue the last readdir call, but the location itself is stored in a `sqlfs_readdir_info` structure attached to the file object representing this directory.

```
/* stores the last accessed element of a readdir.
   This is used to continue the last readdir call */
struct sqlfs_readdir_info {
  int inode_ino;                /* inode number */
  struct sqlfs_qstr name;       /* name of the directroy entry */
                                /* The name is needed because of hardlinks */
  int oid;                      /* oid of this element in the db */
};
```

cache_entry sqlfs_get_readdir_entry(inode, name, table) Fetches the readdir cache entry stored in `table` identified by its inode number `inode` and `name`.

cache_entry sqlfs_add_dentry_to_readdir_cache(dir, dentry) Adds a VFS `dentry` to the cache of the directory `dir`. Is used to mantain the consistency of the cache.

sqlfs_fill_readdir_cache(inode) Allocate and fill a readdir cache and attach it to `inode`.

cache_entry sqlfs_add_cache_entry_to_readdir_cache(table, cache_entry) Add a `cache_entry` to the readdir cache represented by the hash table `table`.

sqlfs_remove_dentry_from_readdir_cache(dir, dentry) Deletes the file or directory represented by `dentry` from the cache attached to `dir`.

sqlfs_remove_from_cache_page(cache_entry, previous_entry, table) Removes a `cache_entry` from the cache by merging it with the `previous_entry`.

cache_page sqlfs_add_readdir_cache_page(page, table, hash) Allocates a new cache page and inserts the new page after `page`. The `hash` is used to update the hash table `table`.

sqlfs_release_readdir_cache_page(page, table, int hash) Removes and releases the `page` of the hash table `table`.

table sqlfs_alloc_readdir_cache(void) Allocates an empty readdir cache (hash table).

sqlfs_release_readdir_cache(table) Deletes the readdir cache represented by `table`.

sqlfs_get_hash_value(inode) Calculate the hash value of the inode `inode`.

cache_entry sqlfs_get_next_readdir_cache_entry(entry, table) Fetches the follow-up element of `entry` of the cache represented by `table`.

cache_entry sqlfs_get_lustre_readdir_entry(table, offset) Does the same as `sqlfs_get_readdir_entry` but takes care of the ext2 boundary rules. This is used by the sqlfs filter of Lustre (see 5.4).

sqlfs_readdir_cache_entry_size(entry) Calculate the size a cache entry

sqlfs_readdir_entry_size(dentry) Calculates the size the cache element corresponding to `dentry`.

sqlfs_readdir_check_entry(name, table) Checks if an entry with name `name` already exists. This was used for debugging purposes.

5.1.5 Transaction and callback support

The sqlfs File System is equipped with transaction and callback support, as this is also a prerequisite for the Lustre MDS. A file system transaction is directly mapped to a transaction on the database. The transactions are identified by the process ID of the process, which initiated the transaction. This implicates that all sqlfs file system operations called by the same process between `sqlfs_start` and `sqlfs_commit` are part of the transaction.
A sqlfs callback function is a function pointer with a memory pointer for passing arguments. It is defined as follows:

```
typedef void (*sqlfs_cb_t)(void* data);
```

The following methods are used for accessing the transaction subsystem:

```
int sqlfs_start(void);
int sqlfs_commit(void);
int sqlfs_begin(int* is_new_transaction, pid_t transaction_id);
int sqlfs_end(int is_new_transaction);
int sqlfs_add_cb(sqlfs_cb_t cb_func, void* cb_data);
int sqlfs_call_cb(struct sqlfs_cmd_queue_element* element);
```

sqlfs_start() Starts explicitly a new transaction for the calling process. The command queue element used for starting the transaction on the database is marked for reuse (refer chap. 5.1.3). If this process is not already in a transaction and no other error occurs, 0 is returned, else the error code.

sqlfs_commit() Commits the running transaction for the calling process and calls all registered callbacks. If there was no open transaction or another error occurs, the respective error is returned, else 0.

sqlfs_begin(is_new_transaction, transaction_id) Starts a new transaction by calling `sqlfs_commit` if there was none yet initiated. `is_new_transaction` is set to true, if a new transaction is started or false, if there was a already a transaction running. `transaction_id` is usually set to the process ID that started the transaction. This function is used for example by the sqlfs file read/write operations to ensure they run in a transaction.

sqlfs_end(is_new_transaction) This function is just a conditional commit. `sqlfs_commit` is called when `is_new_transaction`, which was set by a preceding `sqlfs_begin`, is true.

sqlfs_add_cb(cb_func, cb_data) Adds the sqlfs callback function `cb_func` with parameters `cb_data` to the current transaction. If no open transaction exists, an error is returned. This function returns the error code if an error occured, or else 0. After the transaction is commited, `cb_func` is called with `cb_data` as argument.

sqlfs_call_cb(element) This function is used internally by `sqlfs_commit` to call all registered callback functions.

5.1.6 iopen extension

The iopen extension allows to open files (or directores) just by specifying the inode number and is required by the Lustre MDS. The access occurs through the virtual hidden directory `/__iopen__`, where for every inode a file with the inode number as it's name exists.
The iopen extension consists mainly of the following 3 methods:

```
int sqlfs_check_for_iopen(struct inode *dir, struct dentry *dentry);
int sqlfs_iopen_get_inode(struct inode *inode);
struct dentry *iopen_connect_dentry(struct dentry *de, struct inode *inode);
```

sqlfs_check_for_iopen(inode, dentry) This function is spliced into `sqlfs_lookup`. It checks, whether the name in `dentry` is `__iopen__` and `inode` is the root inode. If yes, it loads the special iopen inode by calling `iget` with `SQLFS_IOPEN_INO` as inode number and attaches the inode to the dentry. It returns 1 if the check was positive, 0 else.

sqlfs_iopen_get_inode(inode) This function is spliced into `sqlfs_read_inode`. It returns 1 if the inode number is the one for `/__iopen__`, in which case the inode is filled in appropriately. Otherwise, this fuction returns 0.

iopen_lookup(inode, dentry) This is the only file system operation attached to the special iopen inode, the `lookup` function of the `inode_operations` structure.

5.2 DB client

The database client (DB-Client) acts as proxy between the kernel module and the database. The communication with the kernel module is done via `ioctl` as it is explained in chapter 5.1.2 To improve the performance and to take advantage of the SMP capability of today's servers it is highly recommanded to start more than one DB-Client. This can be accomplished with the `-n` parameter of the DB-Client (see 5.2.2). Additional DB-Clients can always be added just by starting them. In fact, as one DB-Client executes exactly one query simultaneously, the number of started DB-Clients corresponds exactly to the number of simultaniously executed queries. Of course the database must be able to handle the amount of started clients.

5.2.1 Architecture

The architecture of the DB-Client is quite simple. After the startup procedure the DB-Client basically performs a loop in which the commands received from the kernel module are executed. To get an overview, you can take a look at figure 5.5.
During the initialisation phase, the DB-Client parses the given command line arguments, forks himself if necessary to provide concurrent access to the database, opens the `ioctl` file `/dev/sqlfs` and then connects to the database. Now, the DB-Client arrives the first time at the beginning of the main processing loop. To initiate a proper connection to the kernel module, the DB-Client performs the `ioctl` command `IOCTL_SQLFS_FIRST_CONNECT` which as well waits for the first command. When the DB-Client returns out of the `ioctl`, the received `sqlfs_comm` structure contains all necessary data to execute the query. The structure element `cmd` defines the command that should be executed. Each of this queries is encapsulated in a function that generates the SQL query by picking the needed data out of the `sqlfs_comm` structure and putting them in the appropriate SQL query string. This string is then sent to to the database which executes the query and returns the result.
Most of the queries are calls to a stored procedure on the database server, but some of them contain the whole sql query. By parsing the resultset of the query, the DB-Client can determine if everything was successfully executed and can take further actions. The result is then copied back into the `sqlfs_comm` structure and returned by executing an `ioctl`, but from this time on with the `ioctl` command `IOCTL_SQLFS_COMM`, as `IOCTL_SQLFS_FIRST_CONNECT` is only used once. The main loop has now been executed exactly one time and it will continue until the kernel module ends the loop by sending the `ioctl` command `IOCTL_SQLFS_DISCONNECT`. By receiving this command, the DB-Client ends the main loop, releases all allocated ressources and terminates.
The termination of all DB-Client can be initiated by calling the helper program `stopping`, which in turn tells the kernel to send an `IOCTL_SQLFS_DISCONNECT` to all attached DB-Clients.

5.2.2 Synopsis

```
DB-Client [-h host] [-n number of DB-Clients] [-p port] [-u user] [-a auth] db
```

option	use
`-h host`	The host that the DB-Client connects to. Default is `localhost`
`-n number of DB-Clients`	How many database client should be started.
`-p port`	Port of the postgresql server. Default is 5432
`-u user`	Specifies which user connects to the DB (normally `lustre`)
`-a auth`	Password, if needed.
`db`	Replace `db` with the name of your database (normally `lustre`)

Table 5.1: Options for DB-Client

5.3 Metadata database

The metadata database stores the data of the sqlfs filesystem and provides as well a certain amount of automatic consistency checking by using primary and foreign key constraints. Most of the superblock, inode and file operations are implemented as stored procedures. Some of this functions cannot be implemented as stored procedures because their resultsets consist of duples, therefore the whole query for this commands is sent to the database.

5.3.1 Database selection

The goals of the sqlfs filesystem are to provide a storage system, that is highly available, consistent, stable and fast. To achieve this, the used database system has to provide the same functionality.
The Lustre filesystem and mostly the sqlfs filesystem are published under the GPL General

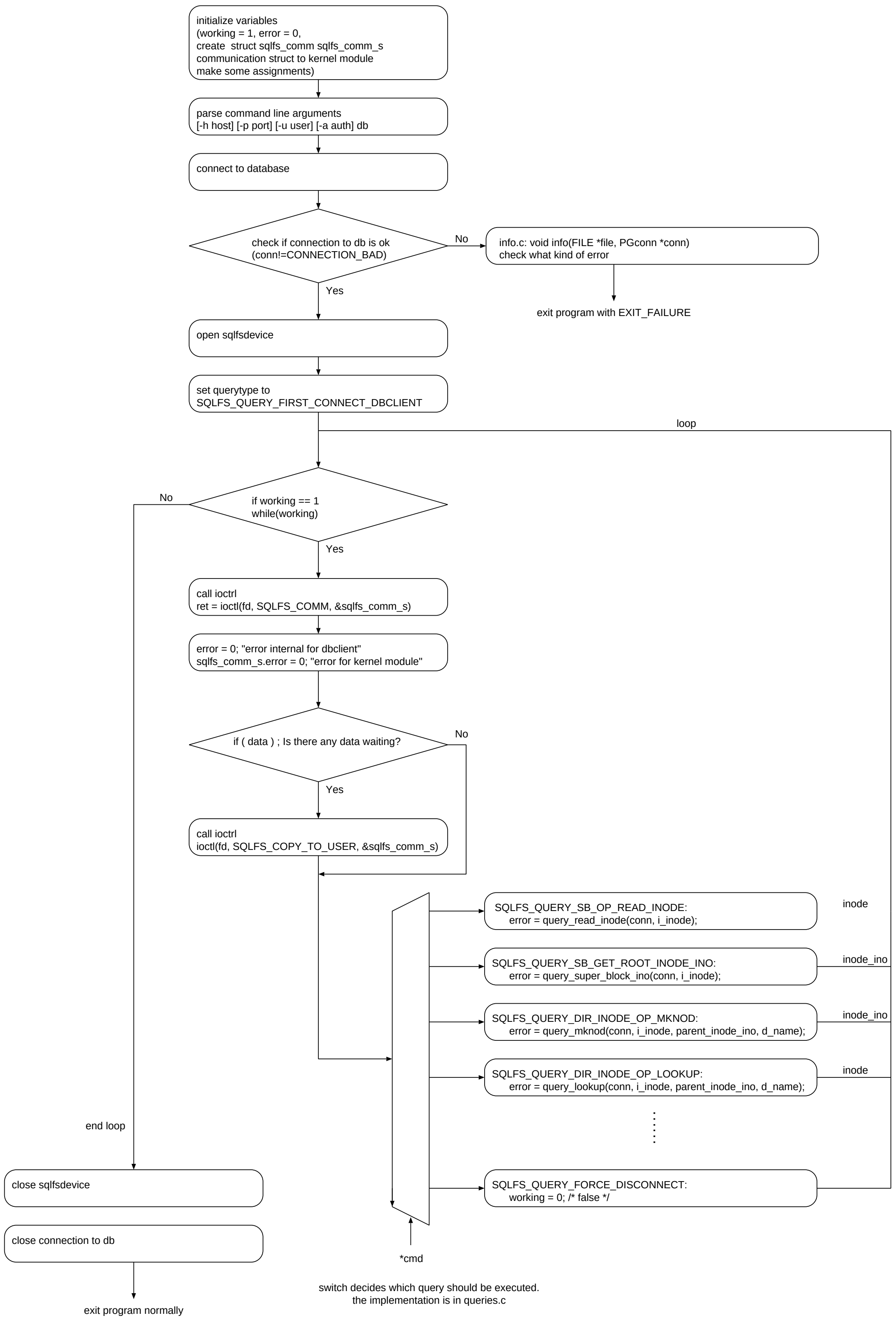

initialize variables
(working = 1, error = 0,
create struct sqlfs_comm sqlfs_comm_s
communication struct to kernel module
make some assignments)
parse command line arguments
[-h host] [-p port] [-u user] [-a auth] db
connect to database
check if connection to db is ok
(conn!=CONNECTION_BAD)
No
info.c: void info(FILE *file, PGconn *conn)
check what kind of error
exit program with EXIT_FAILURE
Yes
open sqlfsdevice
set querytype to
SQLFS_QUERY_FIRST_CONNECT_DBCLIENT
loop
No
if working == 1
while(working)
Yes
call ioctrl
ret = ioctl(fd, SQLFS_COMM, &sqlfs_comm_s)
error = 0; "error internal for dbclient"
sqlfs_comm_s.error = 0; "error for kernel module"
if (data) ; Is there any data waiting?
No
Yes
call ioctrl
ioctl(fd, SQLFS_COPY_TO_USER, &sqlfs_comm_s)
SQLFS_QUERY_SB_OP_READ_INODE:
error = query_read_inode(conn, i_inode);
inode
SQLFS_QUERY_SB_GET_ROOT_INODE_INO:
error = query_super_block_ino(conn, i_inode);
inode_ino
SQLFS_QUERY_DIR_INODE_OP_MKNOD:
error = query_mknod(conn, i_inode, parent_inode_ino, d_name);
inode_ino
SQLFS_QUERY_DIR_INODE_OP_LOOKUP:
error = query_lookup(conn, i_inode, parent_inode_ino, d_name);
inode
end loop
close sqlfsdevice
SQLFS_QUERY_FORCE_DISCONNECT:
working = 0; /* false */
close connection to db
*cmd
switch decides which query should be executed.
the implementation is in queries.c
exit program normally

Figure 5.5: Block diagram of DB-Client

Public Licence, this code can therefore be freely reused and distributed, so it is reasonable not to use a commercial database system, but a a freely available that can be used at no charge.
Three databse systems resultet as possible candidates. MySQL, SAP DB and Postgres. MySQL is fast, but the lack of stored procedures and the incomplete support of transactions disqualifies this database system. SAP DB supports all the necessary features, but it is not yet mature, as it lost all the stored data when the database was stopped. The last of our three candidates appeared to be the right choice. Postgres supports all the necessary features, is stable (we never had any problems) and with Release 7.4.x it is fast. The insert and deletion speed of version 7.3.x was not yet convincing, so everbody is encouraged to switch to the new version.

5.3.2 PostgreSQL

To guarantee that the database and the sqlfs file system is always in a consistent state, it is necessary that every operation is executed atomically. The operation has to be executed either completely or if a problem arises nothing at all should have been done. The "undoing" of actions is called "rollback". In other words, a transaction is a logical unit of work that must not be subdivided. A transaction is also used to coordinate updates made by two or more concurrent users.
Changes made by a transaction are not visible to the system until the transaction is committed and all the changes are physically written to disk.

The easiest way to take advantage of transactions is to use stored procedures, as they are always executed in a transaction. A stored procedure is a server side function that performs several operations as one unit. An even better approach would be the use of precompiled statements. This are compiled stored procedures, that need to be parsed only once. This feature was introduced with postgres 7.4.x, but as we started the sqlfs filesystem with version 7.3.x we did not use them.

Most databases use some kind of locking mechanisms to coordinate multiuser update. Postgres instead uses a model called `multi-versioning` to support concurrent accesses. The database creates a new copy of the rows that have been modified, therefore other users see the original values until the transaction is commited. This model still uses locks, but far less frequently as one might expect. The commit operation simply marks the old rows as obsolete and makes instead the new rows visible to the other users. This model is extremely useful in our case, because it we often have concurrent updates on the same table (but not the same row) and any unnecessary locking would cause a performance loss.
The multi-versioning model unfortunatly has a disadvantage. Data marked as obsolete is not automatically removed from the database, instead it is hidden. The newest version of postgres is able to reuse some of this obsolete rows, but there is nevertheless much unused space in the database after many update and delete operations. This empty spaces cause a performance loss and need to be periodocally removed from the database. This can be done by the `vacuum` command, which garbage collects the whole database and updates the access statistics to improve the optimizers planning. To accomplish this automatically, a helper program called `pg_autovacuum` can be used. This program runs as a deamon that periodically polls the database and performs a `vacuum` if it's necessary. The disadvantage is that the row level statistics of postgres must be activated which causes a slight but noticable performance loss and that the vacuum operation itself is a resource and cpu intensive task.

Postgres uses like any other database system indexes to speedup the lookup operation. Postgres automaticaly generates an index over primary and foreign keys of a table, because these are often used. More indexes provide faster lookup but slower insert, update and delete operations. It's therefore necessary to find a good trade-off between read and write speed. In our case, the default indexes were used, as we are interested in fast insert and update operations.

Database constraints (primary and foreign keys, etc) can be used to automatically verify some integrity properties of a database. For example, it is not possible to have two inodes with the same inode number stored in one filesystem. With a constraint this can be automatically guaranteed.

PostgreSQL does not support unsigned integers, but as the VFS does often use these it is necessary to introduce a mapping between the unsigend integers of the VFS filesystem and the signed integer types of PostreSQL as follows.

kernel	bytes	database	bytes
__u8	1	smallint	2
__u16	2	integer	4
__u32	4	bigint	8

Table 5.2: Numeric data types in Linux kernel and PostgreSQL

5.3.3 Database schema

The used database schema for the sqlfs filesystem is basically a mapping of the ext2 structures to a relational schema. But because we use a database to store the data, we don't have to care about the physical layout of the data, that's the job of the filesystem the database uses.
Let's explain the chosen database schema by introducing the used tables.

inodes

`oid`	`oid`	inode number
`i_mode`	`integer`	file type and access rights
`i_size`	`bigint`	file length
`i_uid`	`integer`	low 16 bits of owner uid
`i_gid`	`integer`	low 16 bits of group id
`i_links_count`	`integer`	hard links count
`i_atime`	`bigint`	time of last access
`i_ctime`	`bigint`	time of creation
`i_mtime`	`bigint`	time of modification
`i_flags`	`bigint`	file flags
`i_rdev`	`bigint`	device file information
`i_generation`	`bigint`	file version
`i_sqlfs_flags`	`bigint`	internal flags of the sqlfs
`i_real_dir_size`	`integer`	real size of the inode, not block aligned

The data of a VFS inode is mapped into the above structure. `oid` represents the inode number and has to be unique. This is guaranteed by the data type `oid` which represents a database wide unique number. The `oid` is the primary key of this table, because an inode is uniquely identified by its number. Except for the two last entries, the stored data correspond the their counterpart of the VFS inode object. The field `i_sqlfs_flags` contains the flags explained in chapter 5.1.1. `i_real_dir_size` is used by the Lustre filter driver to simulate ext2 `readdir` pages.

dir_entries

`inode`	`oid`	inode number
`name_len`	`smallint`	name length
`name`	`varchar`	name
`parent_inode`	`oid`	parent inode number

The table `dir_entries` contains all directory entries. They provide a mapping between the name of a file or directory and the inode representing it. Such a directory entry is uniquely identified by the number of its parent inode (the directory containing the file) and its name. The two entries represent together the primary key of this table.
It is possible that two files the same directory are attached to the same inode, therefore the inode number cannot be used as an identifier. `inode` and `parent_inode` are both foreign keys of the inode number (`oid`) of the inodes table.

symlinks

`inode`	`oid`	inode number
`symlink`	`text`	symlink as text

Symlinks point to a file located somewhere on all the mounted filsystems. To resolve a symbolic link, the VFS calls either the inode operation `readlink` or `follow_link` which get the `symlink` entry of the table `symlinks` by using the `inode` number. `inode` is as you might expect a foreign key of the `oid` entry of the table `inodes`.

superblock

id	integer	id for filesystem
root_inode	oid	inode number of root inode

The sqlfs does not need a real super block like ext2. Instead an ordinary inode which number is saved in this table is used to represent the root directory. This table is used for nothing else and contains exactly on entry. The primary key of this table, `id`, is used to identify the filsystem, but it is not really needed and set to 1 during the initialisation of the filesystem.

xattr

inode	oid	inode this xattr belongs to
name	varchar(256)	name of the xattr
value	bytea	data of the xattr

The table `xattr` is used to store the extended attributes, that Lustre uses. The use of these attributes is not limited to Lustre, officially they were intrudeced with the 2.6 kernel to support access control lists and other extensions of the filesystem. The primary key consists of the two entries `oid` and `name`. In addition, the `oid` has a foreign key constraint to the `oid` entry of the inode table. The element `value` contains the data of the xattr and is of type `bytea`. This type is used to store binary data of arbitrary length in escaped form.

files

inode	oid	inode number
lo_oid oid	oid	mapping to blob

This table provides a mapping between the inodes and the `blob` (binary large object) containing the filedata. This table is needed, because all the `blobs` are stored in a system table called `pg_largeobject`. The primary key of this table is the inode number `inode` which is as well a foreign key of the `oid` stored in the `inode` table.

pg_largeobject

loid	oid	blob identifier
pageno	oid	page number
data	bytea	stored data

The `pg_largeobject` table is a postgres system table that stores all `blob` objects.

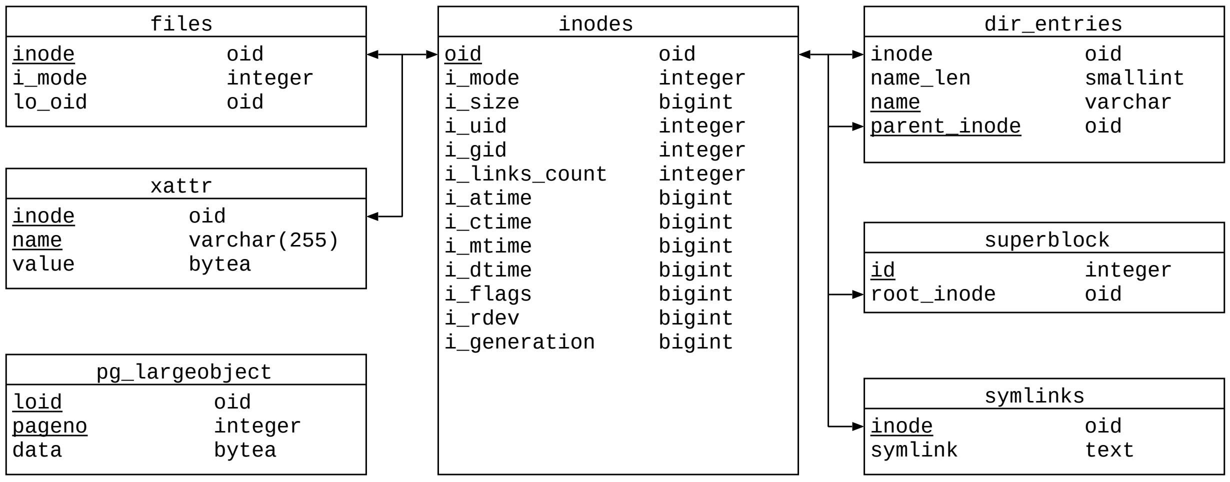


Figure 5.6: Database schema

5.4 Lustre sqlfs File System Filter

As mentioned in chap. 3.3, the Lustre Metadata Server uses, in addition to the VFS, a proprietary File System Filter to store it's data. Unfortunately, the File System Filter interface is completely undocumented, so we had to reverse-engineer existing File System Filters to figure out the specification of the interface functions.

5.4.1 Interface

The File System Filter operations are declared as follows:

```
struct fsfilt_operations {
        struct list_head fs_list;
        struct module *fs_owner;
        char   *fs_type;
        void   *(* fs_start)(struct inode *inode, int op,
                             void *desc_private);
        void   *(* fs_brw_start)(int objcount, struct fsfilt_objinfo *fso,
                                 int niocount, void *desc_private);
        int     (* fs_commit)(struct inode *inode, void *handle,
                              int force_sync);
        int     (* fs_commit_async)(struct inode *inode, void *handle,
                                    void **wait_handle);
        int     (* fs_commit_wait)(struct inode *inode, void *handle);
        int     (* fs_setattr)(struct dentry *dentry, void *handle,
                               struct iattr *iattr, int do_trunc);
        int     (* fs_iocontrol)(struct inode *inode, struct file *file,
                                 unsigned int cmd, unsigned long arg);
        int     (* fs_set_md)(struct inode *inode, void *handle, void *md,
                              int size);
        int     (* fs_get_md)(struct inode *inode, void *md, int size);
        ssize_t (* fs_readpage)(struct file *file, char *buf, size_t count,
                                loff_t *offset);
        int     (* fs_add_journal_cb)(struct obd_device *obd,
                                      __u64 last_rcvd, void *handle,
                                      fsfilt_cb_t cb_func, void *cb_data);
        int     (* fs_statfs)(struct super_block *sb,
                              struct obd_statfs *osfs);
        int     (* fs_sync)(struct super_block *sb);
        int     (* fs_map_inode_page)(struct inode *inode, struct page *page,
                                      unsigned long *blocks, int *created,
                                      int create);
        int     (* fs_prep_san_write)(struct inode *inode, long *blocks,
                                      int nblocks, loff_t newsize);
        int     (* fs_write_record)(struct file *, void *, int size,
                                    loff_t *, force_sync);
        int     (* fs_read_record)(struct file *, void *, int size,
                                   loff_t *);
        int     (* fs_setup)(struct super_block *sb);
};
```

Not all of these functions are needed for the MDS, as the File System Filter is used for both the MDS and the OST. The following explanations of the respective functions are the result of our reverse-engineering efforts. Although the MDS worked stable with these self-made specification they are provided without any guaranty of correctness.

fs_list Used by Lustre

fs_owner The module which hosts these functions

fs_type The name of the filesystem these functions work on (e.g. ext3, reiserfs)

fs_start(inode, op, desc_private) Starts a new transaction on `inode` with intended operation op. Returns a handle to the transaction.

fs_brw_start(objcount, fsfilt_objinfo, fso, niocount, desc_private) Not used by the MDS.

fs_commit(inode, handle, force_sync) Commits the transaction on `inode` represented by `handle`.

fs_commit_async(inode, handle, wait_handle) Not used by the MDS.

fs_commit_wait(inode, handle) Not used by the MDS.

fs_setattr(dentry, handle, iattr, do_trunc) Sets the attributes of an inode.

fs_iocontrol(inode, file, cmd, arg) Maps the parameters to the underlying `ioctl` file operation.

fs_set_md(inode, handle, md, size) Sets the binary metadata `md` with size `size` for `inode`. Returns the number of written bytes.

fs_get_md(inode, md, size) Reads the metadata from `inode` into the buffer `md` with size `size`. Returns the number of bytes read (0 in case of error), or the size of the metadata when `md` is `null`.

fs_readpage(file, buf, count, offset) Reads `count` bytes from `file` beginning at `offset` into `buf`, which resides in kernel space. This method is currently only called by `mds_sendpage`. The Lustre client uses this function to read the raw disk blocks of a directory on the MDS to perform a readdir, expecting a `ext2_dirent` structure!

fs_add_journal_cb(obd, last_rcvd, handle, cb_func, cb_data) Adds a callback function (called after commit) to the transaction represented by `handle`.

fs_statfs(sb, osfs) Executes a VFS `statfs` and fills `osfs` with the data.

fs_sync(sb) Flushes all data to disk.

fs_map_inode_page(inode, page, blocks, created, create) Not used by the MDS.

fs_prep_san_write(inode, blocks, nblocks, newsize) Not used by the MDS.

fs_write_record(file, buf, size, off, force_sync) A normal file write, except that `buf` resides in kernel space. Writes `size` bytes from `buf` into the file, beginning at file offset `off`, which must be incremented by the number of written bytes. Returns a negative number in case of error, else 0.

fs_read_record(file, buf, size, off) A normal file read, except that `buf` resides in kernel space. Reads `size` bytes from `buf` into the file, beginning at file offset `off`, which must be incremented by the number of read bytes. Returns a negative number in case of error, else 0.

fs_setup(sb) Used for setting up the filter driver.

5.4.2 Implementation of fsfi lt_sqlfs

Based on the gathered specification we implemented a Lustre File System Filter Driver for sqlfs. Transactions on the MDS never switch it's executing process, the MDS process which starts a transactions does also commit it and does not execute operations for other transactions in between. For that reason the transaction handle can be ignored, transaction identification is done by the process ID of the executing process[1]. The metadata is stored in `xattrs`, in the same way as in `fsfilt_ext3`.
As mentioned above, the method `fs_readpage` is used for a strange way of doing a readdir. In sqlfs, the directory information is not stored in blocks or pages as it is in disk-based filesystems like ext3, but in relational format in the database. Therefore it is necessary to build these pages or blocks dynamically, based on the current content of the directory. The readdir-call is re-entrant, and between two calls the content of the directory can change, which causes lost

[1]This is the reason, why the sqlfs transaction management identifi es transactions by the process ID.

or dublicated entries, dependent whether files were created or deleted between the re-entrant calls. For this reason, the command `rm -rf *` has to reread the directory several times, but at the end it is empty.
The functions `fs_write_record` and `fs_read_record` are implemented in the sqlfs kernel module, the filter just maps the calls to these. Additionally, the function `fs_write_record` can be enabled or disabled[2] through the respective `ioctl` (the utility `sqlfs_set_write_record` in the utils directory serves for this purpose).

[2]This was implemented to evaluate the impact of the concurrent access to the recovery fi le on the MDS performance.

Chapter 6

Benchmarks

Benchmarking is an important component of this thesis, as the goal was to improve the MDS performance. Benchmarking and the analysis of the results is very time-consuming, and although the available benchmarks were carefully chosen to show the performance of the different file systems for the dedicated operations (namely file creation and file lookup), they give just an idea of the behaviour of the different file systems.
Furthermore, a distributed file system like Lustre is a complex system with many different components, whereof the MDS (and the file system being used) is just a small part. This makes it even more difficult to gather exact information about single components by only considering the overall performance.
At the time this thesis was written, Lustre supported only ext3 as underlying filesystem, for both MDS and OST. Therefore sqlfs is only compared to ext3. First, the native performance of the file systems was measured, and in a second step the Lustre performance. The observed operations were file creation and file lookup.

6.1 Native Performance

6.1.1 Test method and Setup

The native performance was determined using the self-written utility `fsbench`. With this utility it is possible to create a specified number of files or directories with multiple simultaneously running processes, of which each works in it's own directory (process working directory - PWD) or all share the same working directory. The time needed to create these files or directories is logged in specified file intervals. Files can be empty or written with random data of specified size.
As the sqlfs is always synchronized and ext3 by default not, the ext3 file system was benched twice, once mounted with default parameters and once mounted with parameters `sync` and `dirsync` to force synchronisation.
The test setup is diagrammed in fig. 6.1, the configuration of the nodes is described in 8.1.11.

6.1.2 ext3 not synchronized

In this test, the time for creating 1'000'000 files in the ext3 file system (unsync'ed) was measured, with different parameters. The results of this test are shown in fig. 6.2. As expected, the performance is much higher, if every process has it's own working directory (PWD). If they all

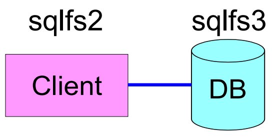


Figure 6.1: Test setup for native benchmarks

work in the same directory, only one process at a time can access the working directory (inode-lock on the directory), so the processes cannot work simultaneously, which causes poorer performance the more processes run. This is effected by increased scheduling overhead. In the second diagram (time per file of figure 6.2), we can see, that the file creation times grow linearly.

6.1.3 ext3 synchronized

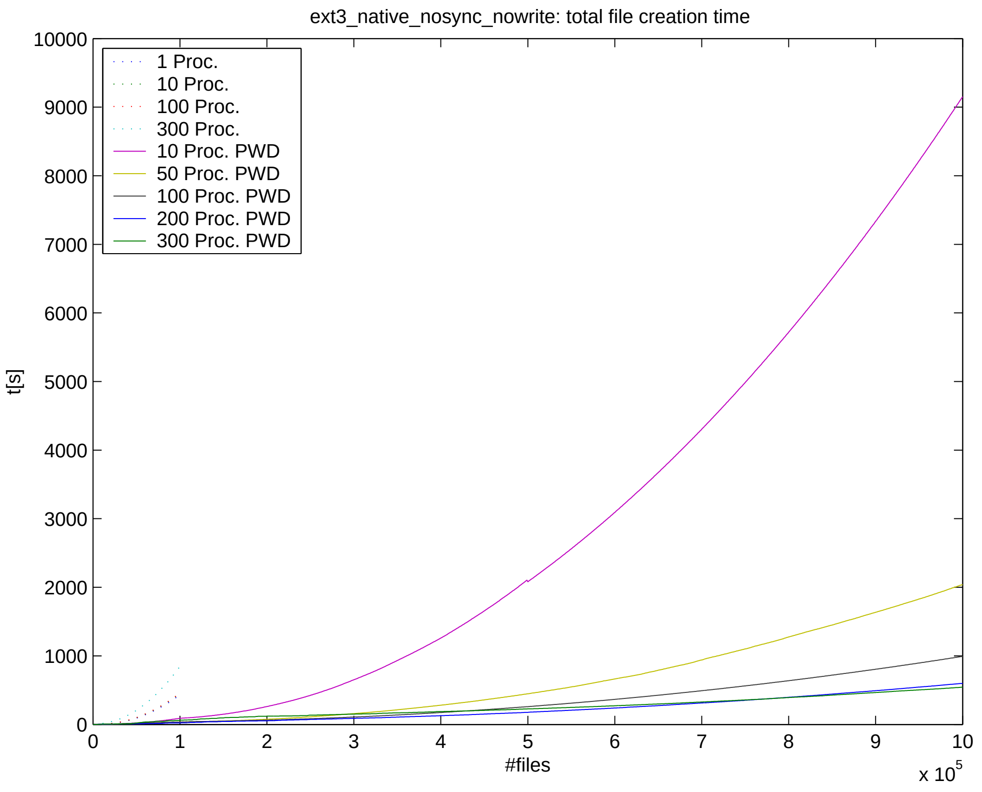
The synchronized ext3 file system was tested in the same way as the unsync'ed. The results are visualized in fig. 6.3. Again, the performance is much poorer when all processes share the same working directory, due to serialization and scheduling overhead. It is interesting, that the difference between the sync'ed and non-sync'ed tests when using the same working directory is quite small! As with the unsync'ed ext3, the file creation times grow linearly.

6.1.4 sqlfs

The sqlfs file system was benchmarked the same way as the ext3-unsync'ed. Again, the time was measured for creating 1'000'000 files in different ways. In fig. 6.4 we can see in the first graph, that the creation times are strongly linear, so the time needed to create a file remains almost constant, not being dependent on the number of files in the directory or filesystem. The peaks in the second graph are caused by the vacuum-daemon of the PostgreSQL database. In case of 300 processes, the increased scheduling overhead is responsible for the higher times.

6.1.5 Comparison

The figures 6.5, 6.6, 6.7 and 6.8 show the sqlfs file system compared to the ext3 file system. Figure 6.8 is plotted for 1'000'000 files, while all other are plotted for 100'000 files.
In sqlfs, the time to create a file is is almost constant, not dependent of the number of running processes nor of the number of files already stored in the directory. With ext3, sync'ed and unsync'ed, the file creation time grows linearly, causing a square growth of the total creation time, which implicates that at a specified amount of files in a directory, sqlfs will outperform ext3.
Compared to unsync'ed ext3, in case of one global working directory, sqlfs is faster when having more than 40'000 files the directory(see fig. 6.5). In case of per-process working directories (fig. 6.6) sqlfs is already faster when having more than 7000 files in each directory! So for 10 processes this is at 70'000 files, for 50 processes at 350'000 files and for 100 processes this would be at about 700'000 files total.
Compared to synchronized ext3, in case of one global working directory, sqlfs is faster when having more than about 27'000 files in this directory (see fig. 6.7). In case of PWDs, the turnover is at about 100'000 files total in the file system, independent from the number of processes.



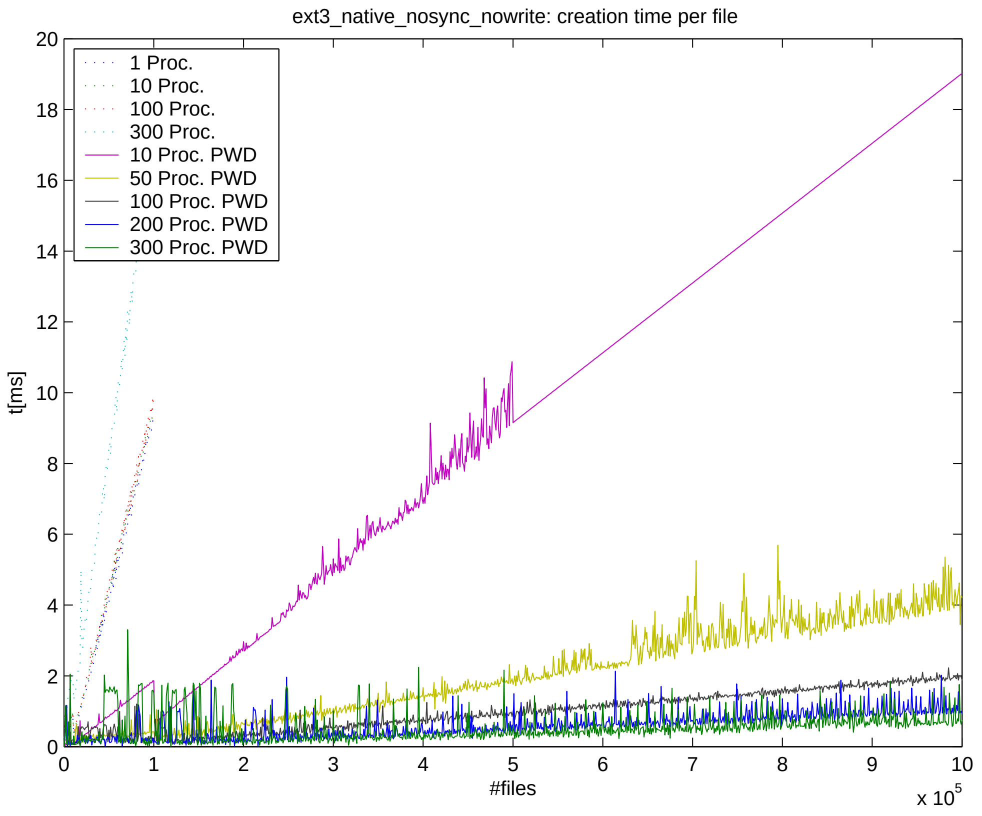
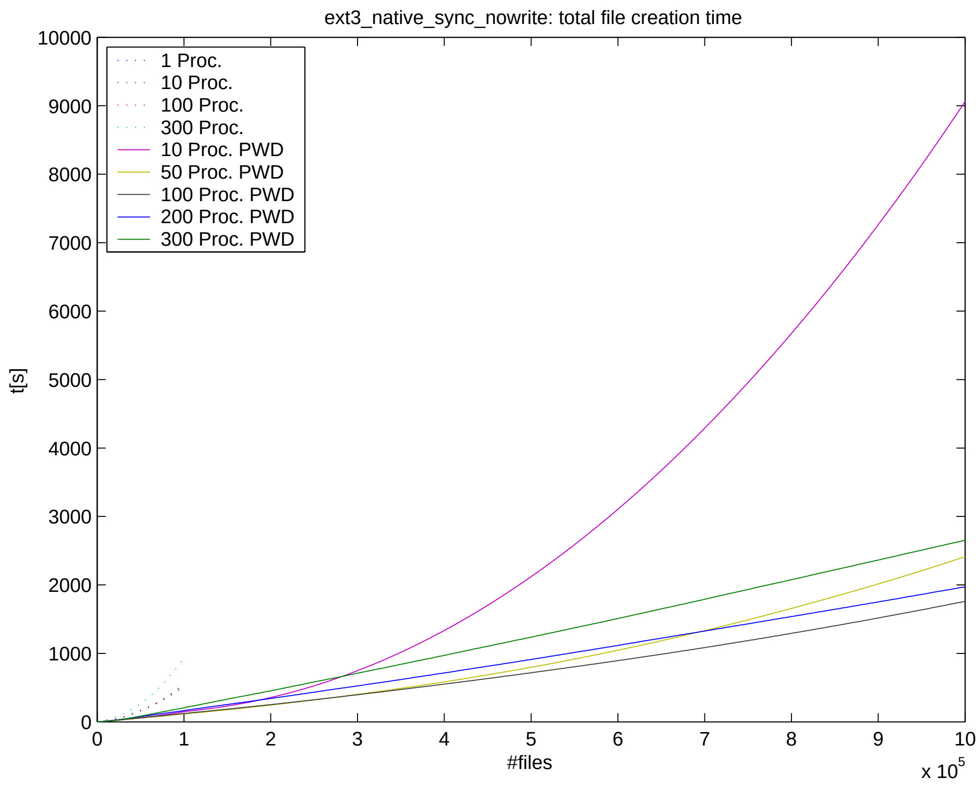


Figure 6.2: ext3 native performance (unsync'ed)



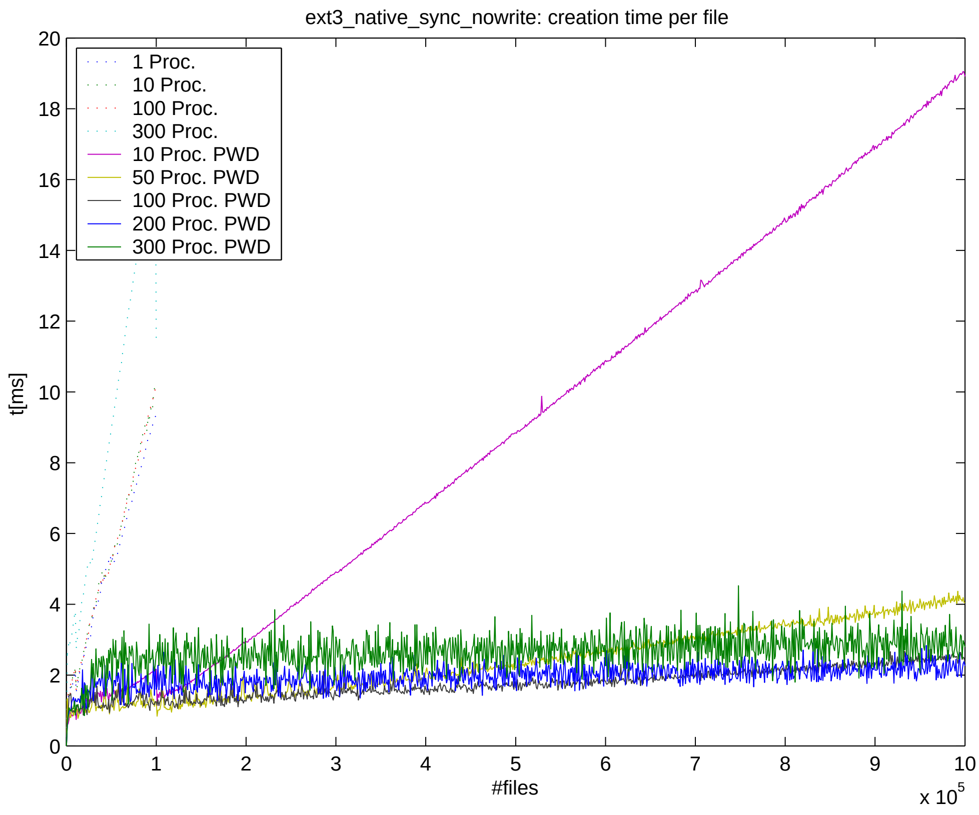
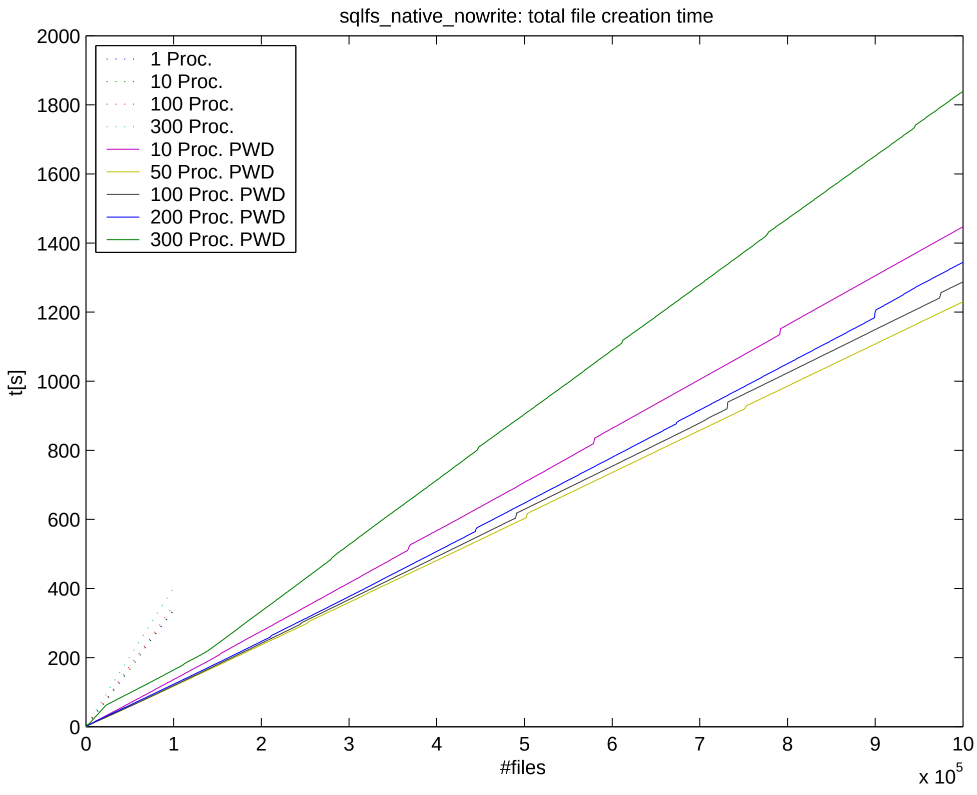


Figure 6.3: ext3 native performance (sync'ed)



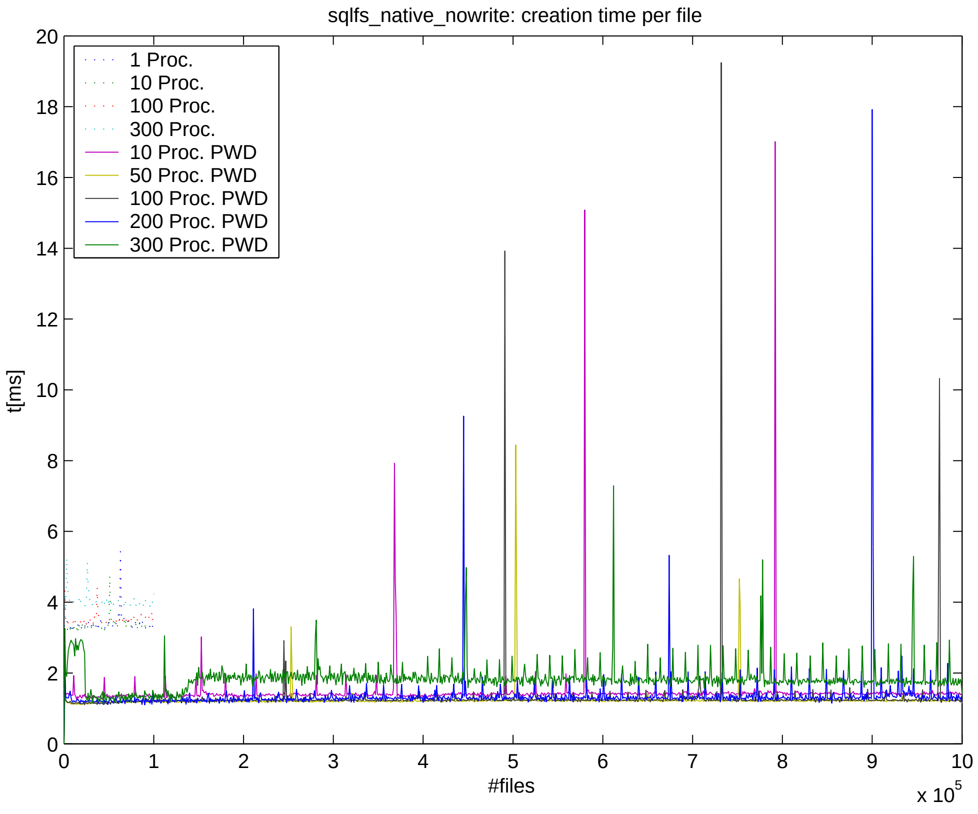
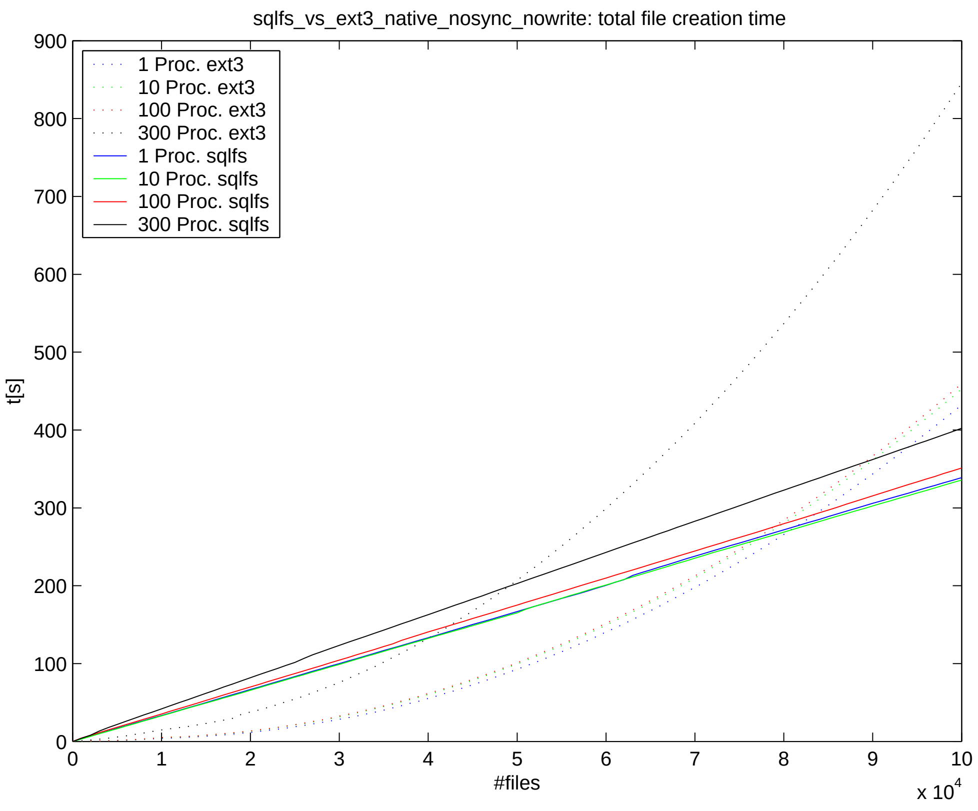


Figure 6.4: sqlfs native performance



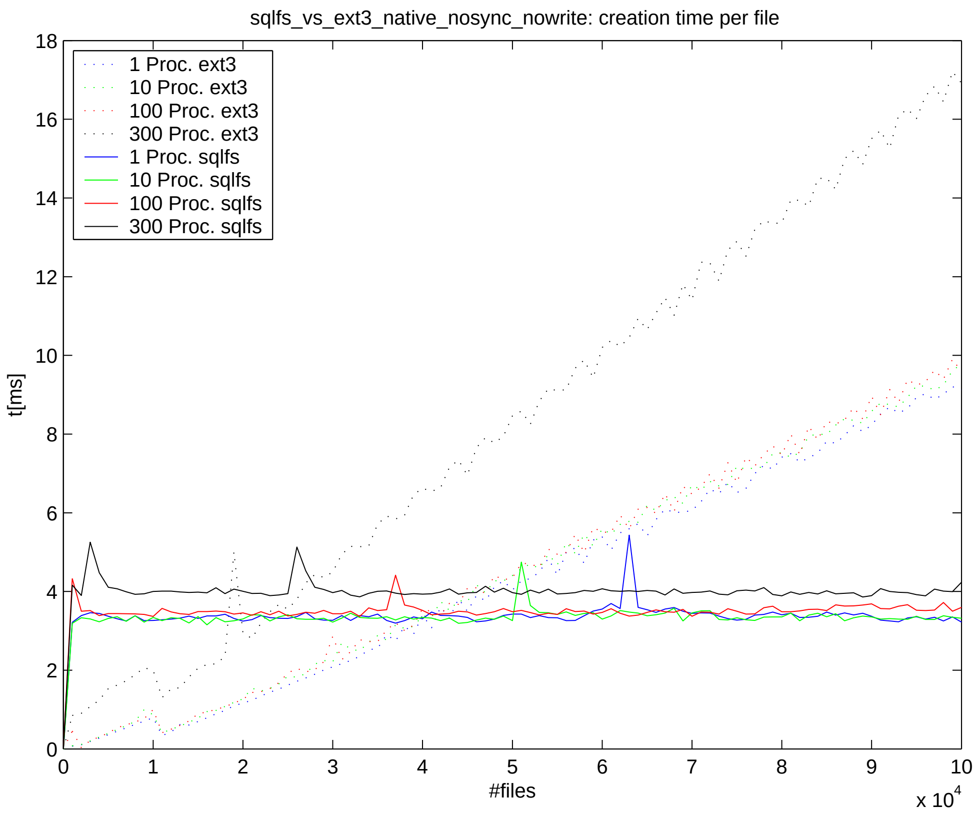
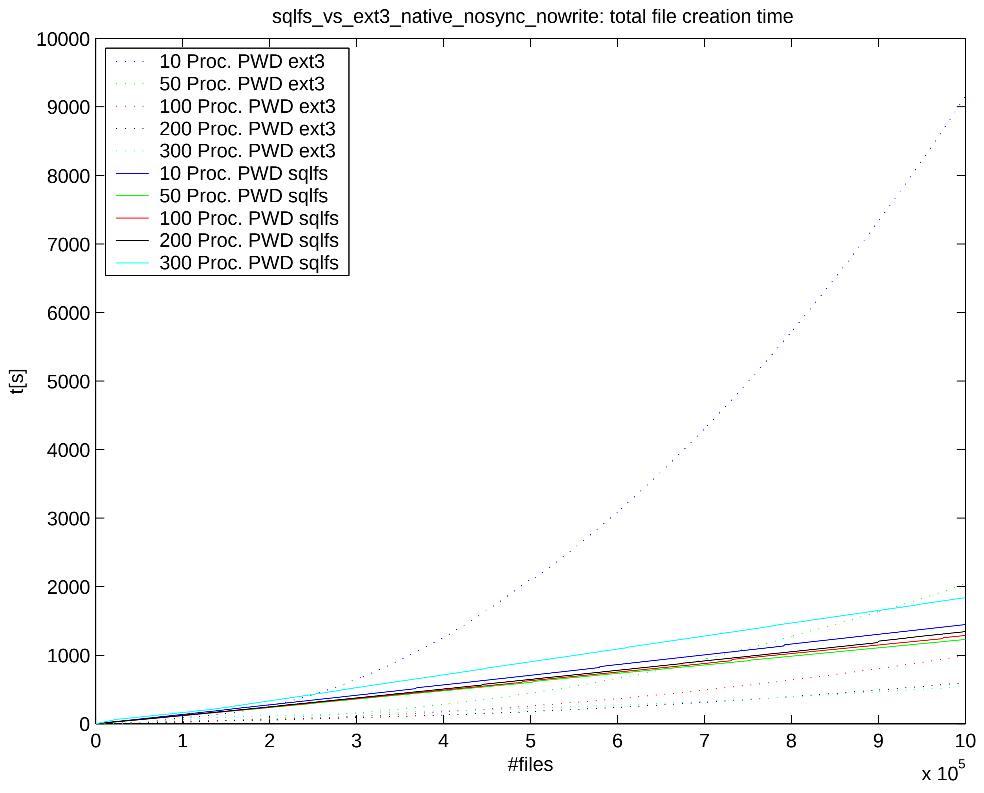


Figure 6.5: sqlfs vs. unsync'ed ext3 performance



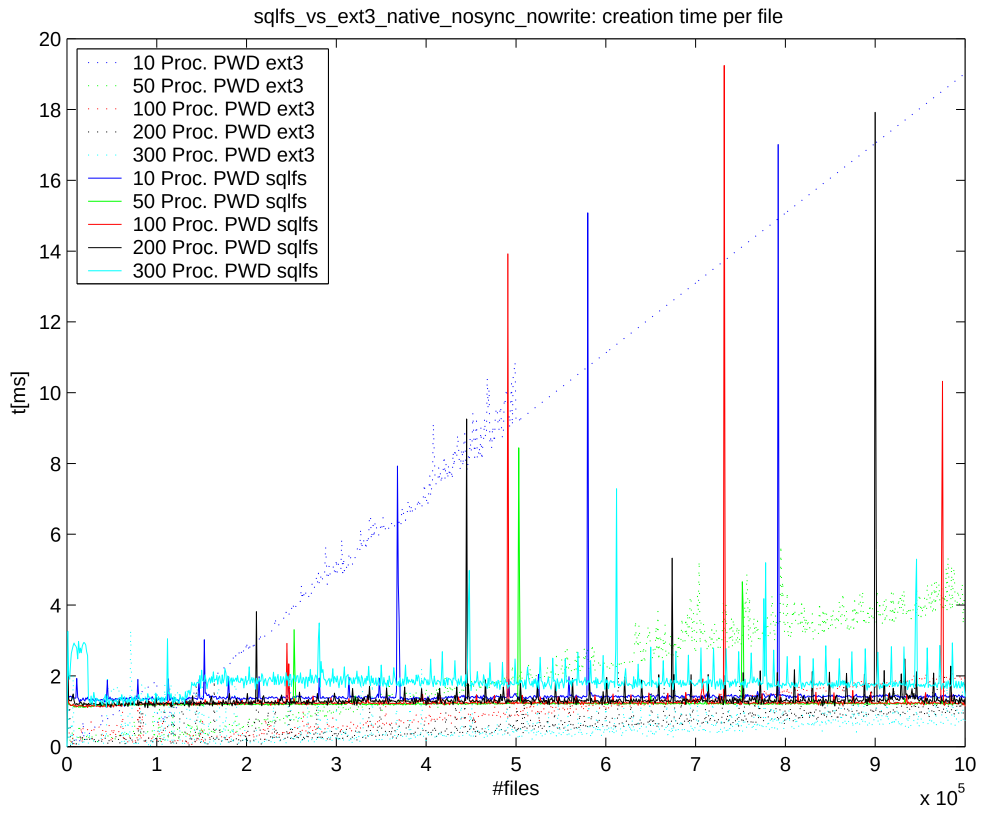
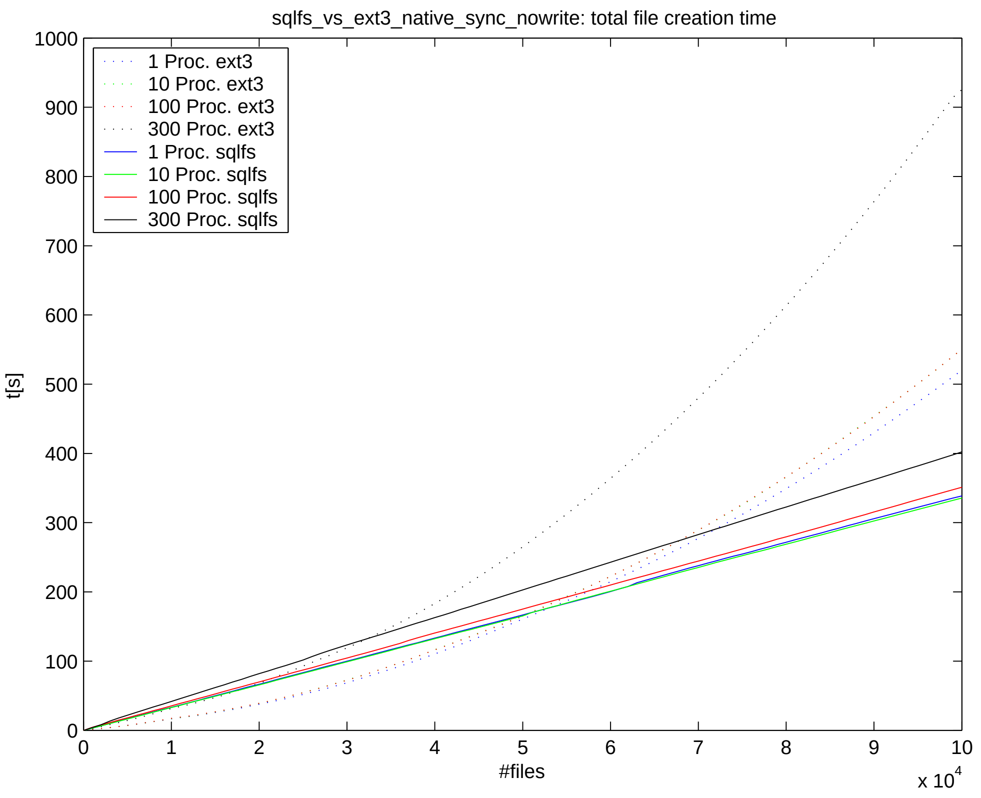


Figure 6.6: sqlfs vs. unsync'ed ext3 performance with process working directories



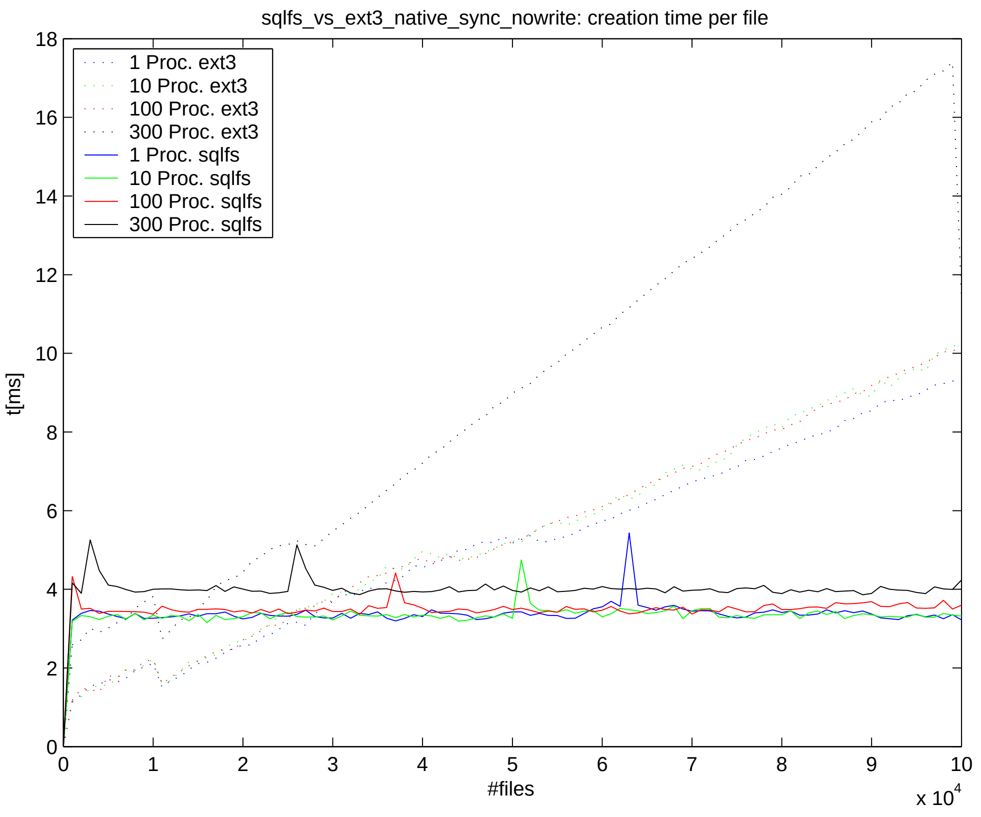
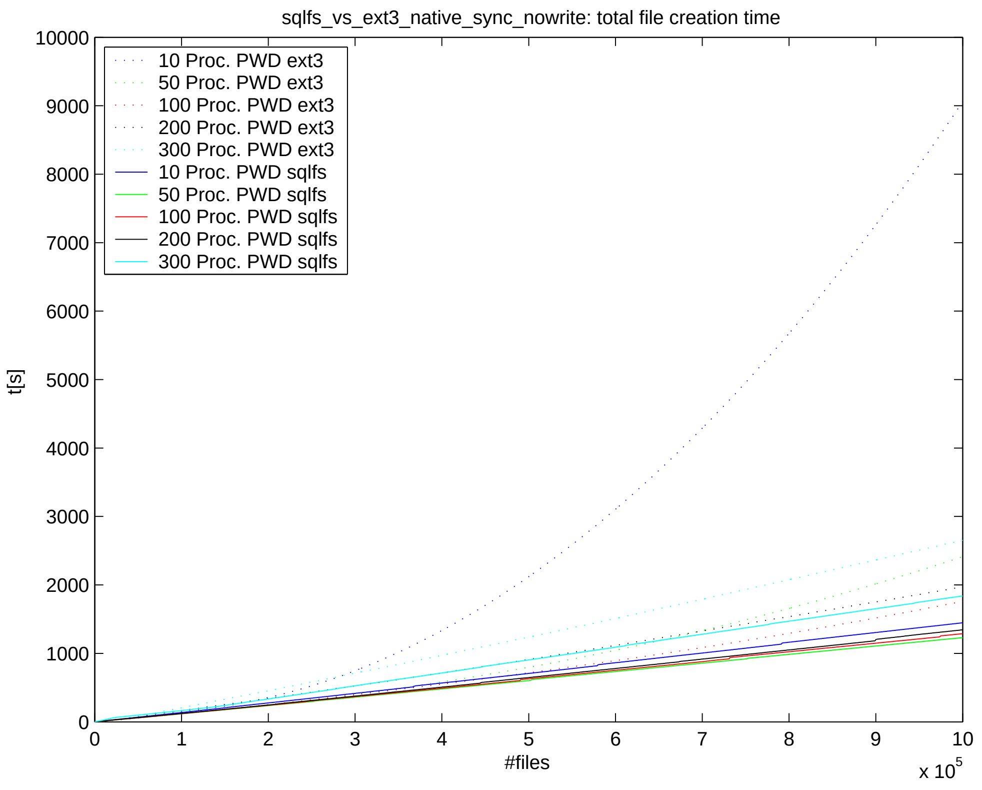


Figure 6.7: sqlfs vs. sync'ed ext3 performance



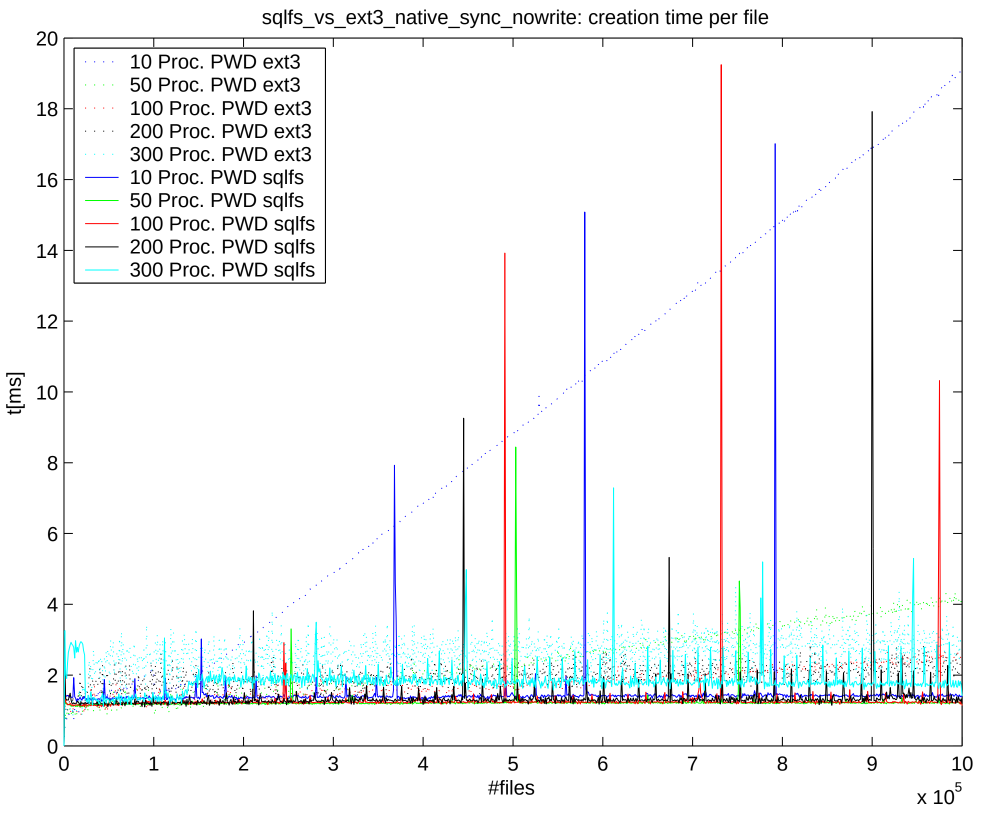


Figure 6.8: sqlfs vs. sync'ed ext3 performance with process working directories

6.2 Lustre Performance

6.2.1 Test method and Setup

The Lustre performance was determined in a similar way as the native file system benchmarks. As here not many processes on the same node, but many nodes are required, another benchmark utility had to be used, `fsbench_mpi`, which is the MPI-version of `fsbench`. In comparison to `fsbench`, `fsbench_mpi` provides three possible working directories: a global directory for all processes, a work directory for each node or a work directory for each process (if more than one process runs on a node).
For this benchmark, 800’000 files were created by different ways. If more processes than clients (nodes) were used, they were distributed uniformly on the used nodes. Each node had it’s own working directory. If the tests were marked with PWD this means that not every node, but every process had it’s own working directory. As the graphs for creation-time-per-file are quite divergent, a square approximated version is added each time to improve the legibility.
The test setup is diagrammed in fig. 6.9, the configuration of the nodes is described in 8.1.11.

6.2.2 ext3

Figures 6.10 and 6.11 show the file creation performance with Lustre running with the MDS using ext3.

6.2.3 sqlfs

Figures 6.12 and 6.13 show the file creation performance with Lustre running with the MDS using ext3.

6.2.4 Comparison

Figures 6.14 shows the Lustre performance with sqlfs and ext3 as MDS-filesystem compared to each other. The total time using ext3 grows quadratic, when using sqlfs it grows linearly. As can be seen in the second graph, the per-file creation time is for sqlfs as MDS-filesystem again more or less constant, while it grows with ext3. If both keep growing as they do until 800’000 files, sqlfs will eventually outperform ext3 also when used by Lustre. If the total file creation times are compared to the native ones, it is outstanding, that the creation times are almost the same, so Lustre is almost as fast as the native file systems (for file creation).

6.3 Analysis

Regarding the time-per-file graphs in fig. 6.5 and fig. 6.7 it is apparent, that sqlfs outperforms ext3 if the directories have a sufficient size, even for the case of unsync’ed ext3. This is due to the different scaling-behaviours of the two filesystems. While in sqlfs the time-per-file is constant (best possible scaling), in ext3 it grows linearly, which is not bad, but worse. Therefore it is just a matter of time or directory-size until the point is reached, where sqlfs is faster.
The directory size where the turnover point is situated at can be decreased by improving the time to create a file in sqlfs. There are several optimization possibilities:

- improving the data base schema
- using stored procedures written in C instead of plpgsql
- using prepared queries
- using binary mode for transmitting data to and from the DB instead of escaped text
- using a faster database engine

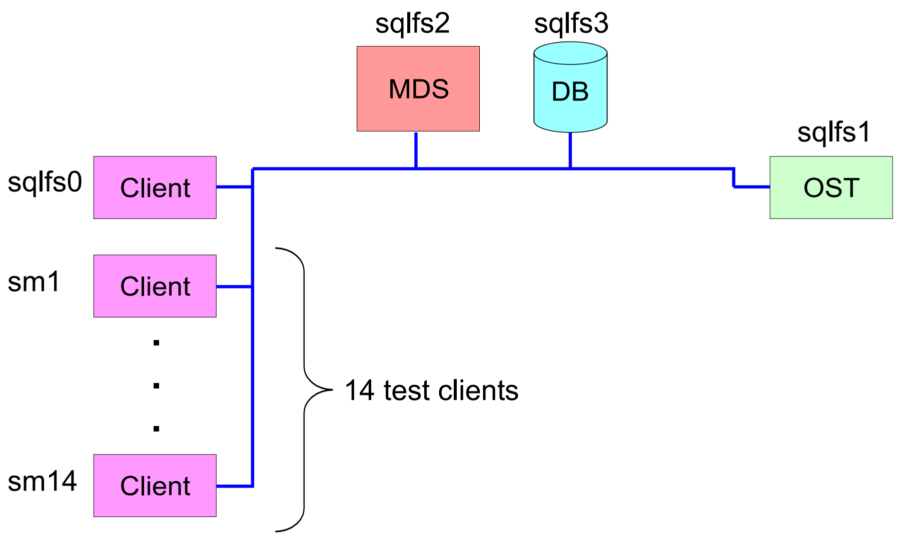


Figure 6.9: Test setup for Lustre benchmarks

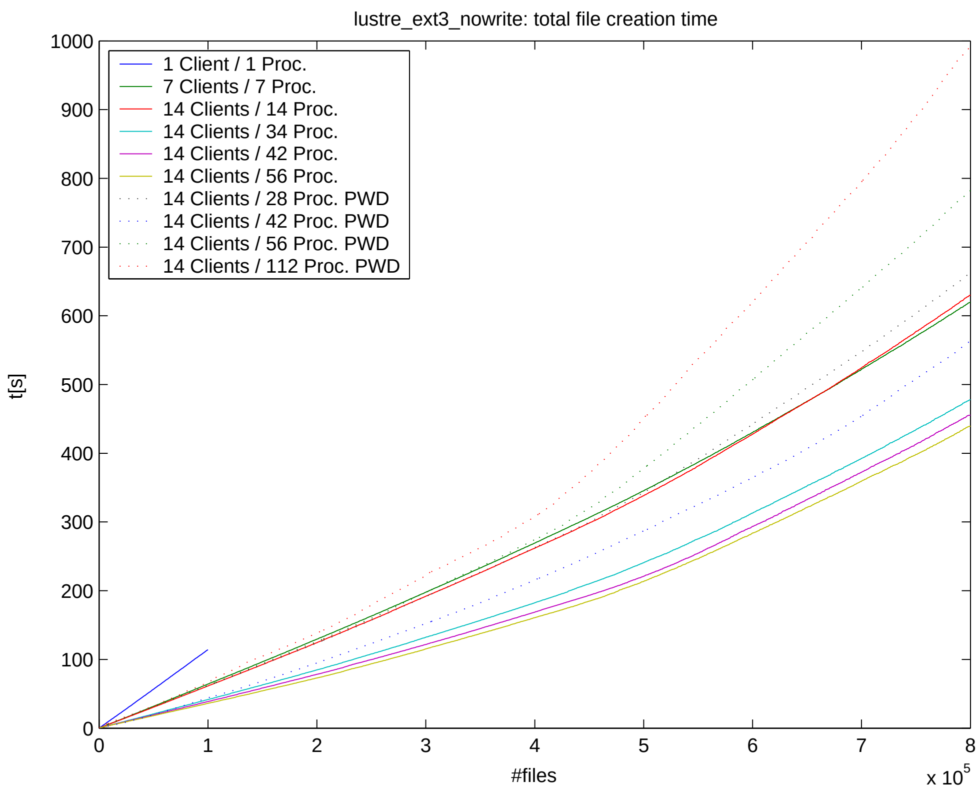
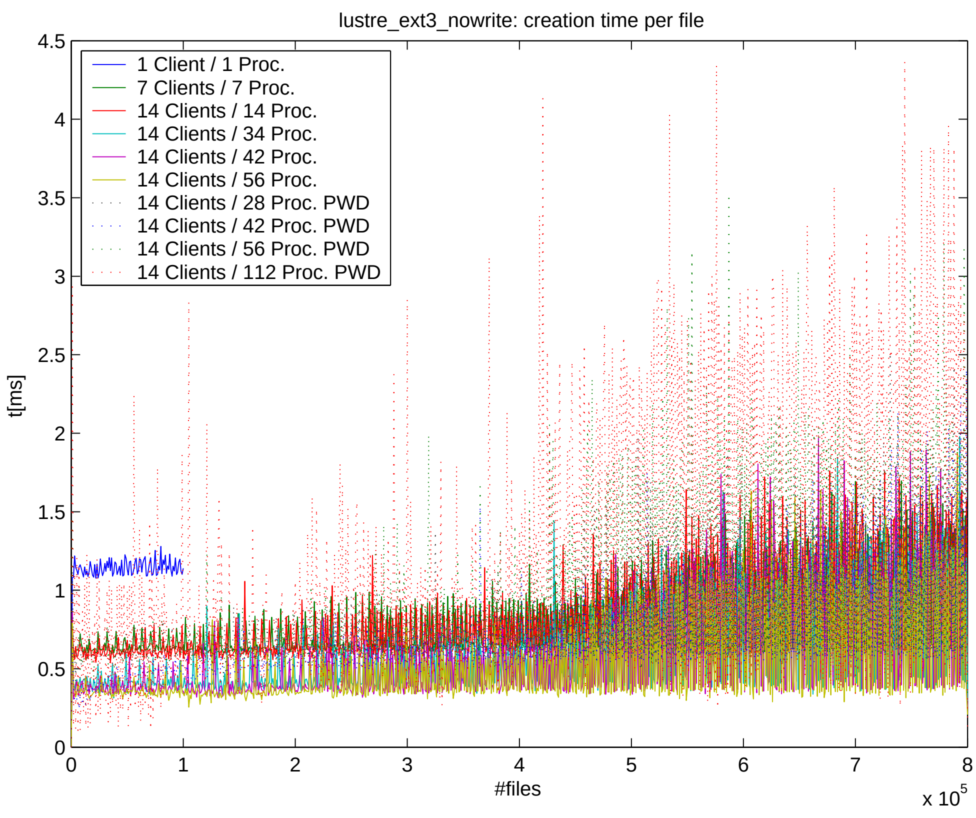


Figure 6.10: Lustre with MDS on ext3, total time



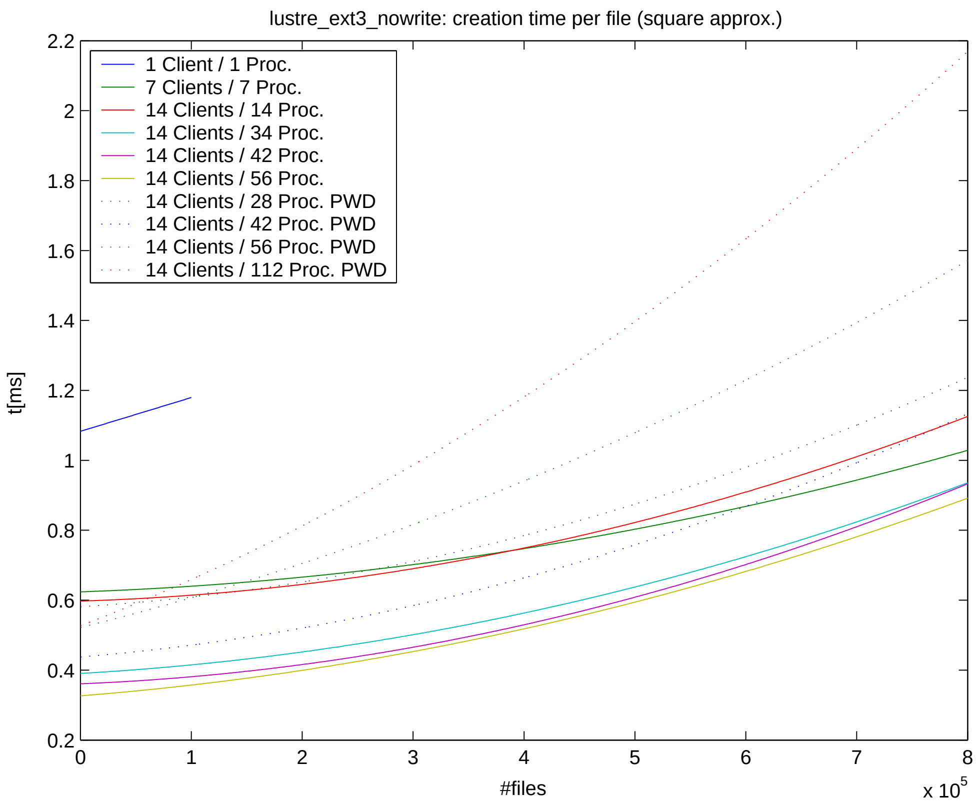


Figure 6.11: Lustre with MDS on ext3, time per file

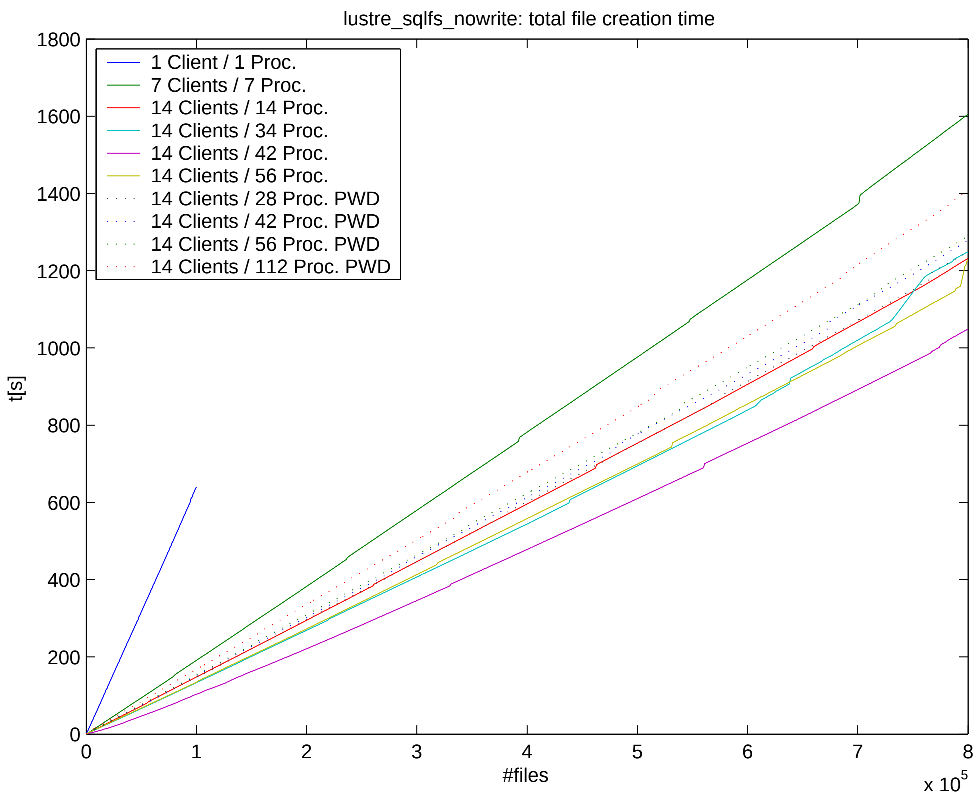
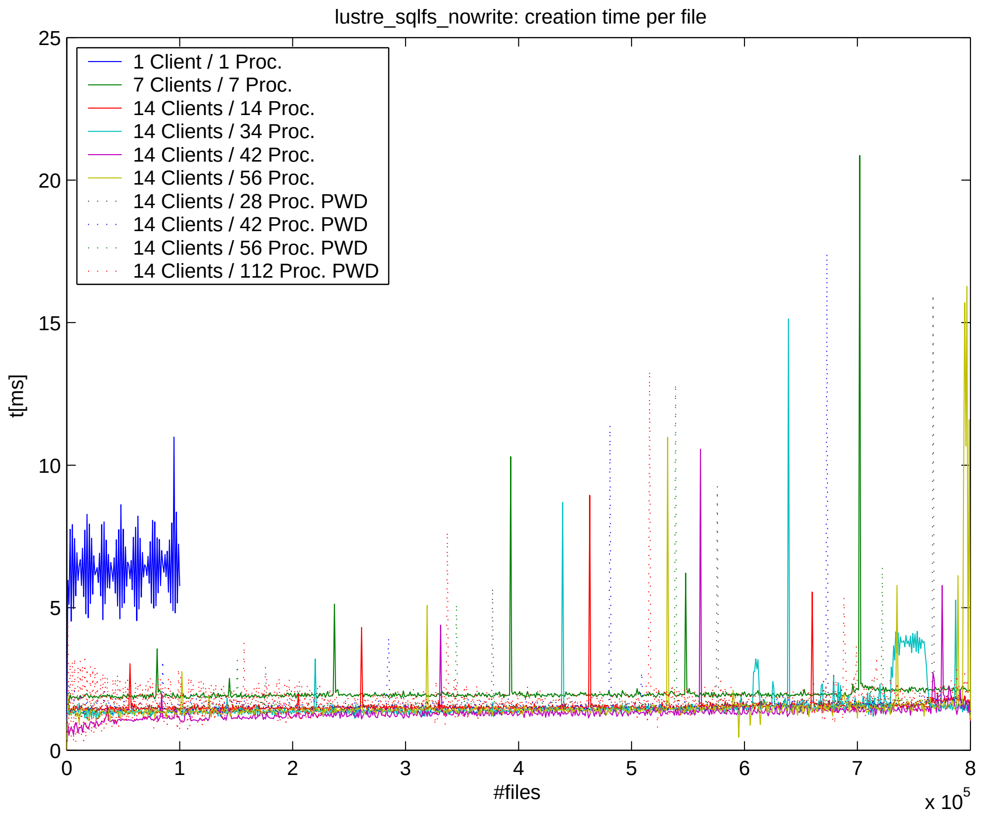


Figure 6.12: Lustre with MDS on sqlfs, total time

Although sqlfs can outperform ext3 natively, Lustre with the MDS using sqlfs is slower than when using ext3 (see fig. 6.14). Although sqlfs again shows a constant time-per-file while ext3 grows linearly, it grows too slow to be beaten by sqlfs for a reasonable directory size.
The reason why sqlfs used by the MDS is not as competetive as when used native is, that a simple file creation on the Lustre Client causes the following operations on the MDS:

1. sqlfs_lookup
2. sqlfs_start
3. sqlfs_create
4. sqlfs_add_cb
5. sqlfs_write_record
6. sqlfs_write_record
7. sqlfs_commit
8. sqlfs_write_inode
9. sqlfs_write_inode

For creating one file on the Lustre File System, the MDS performs 9 file system operations. When using sqlfs, this means, that 8 queries (sqlfs_add_cb goes not to the DB) are sent to the database, packed in 4 transactions (functions not packed between start and commit are single-instruction transactions). This is an unecessary amount of transactions and queries. On the Lustre layer for example, lookup intents are introduced. Unfortunately, this information does not arrive at the sqlfs, because it is neither implemented in the VFS interface nor in the File System Filter.
The following optimization possibilities exist:



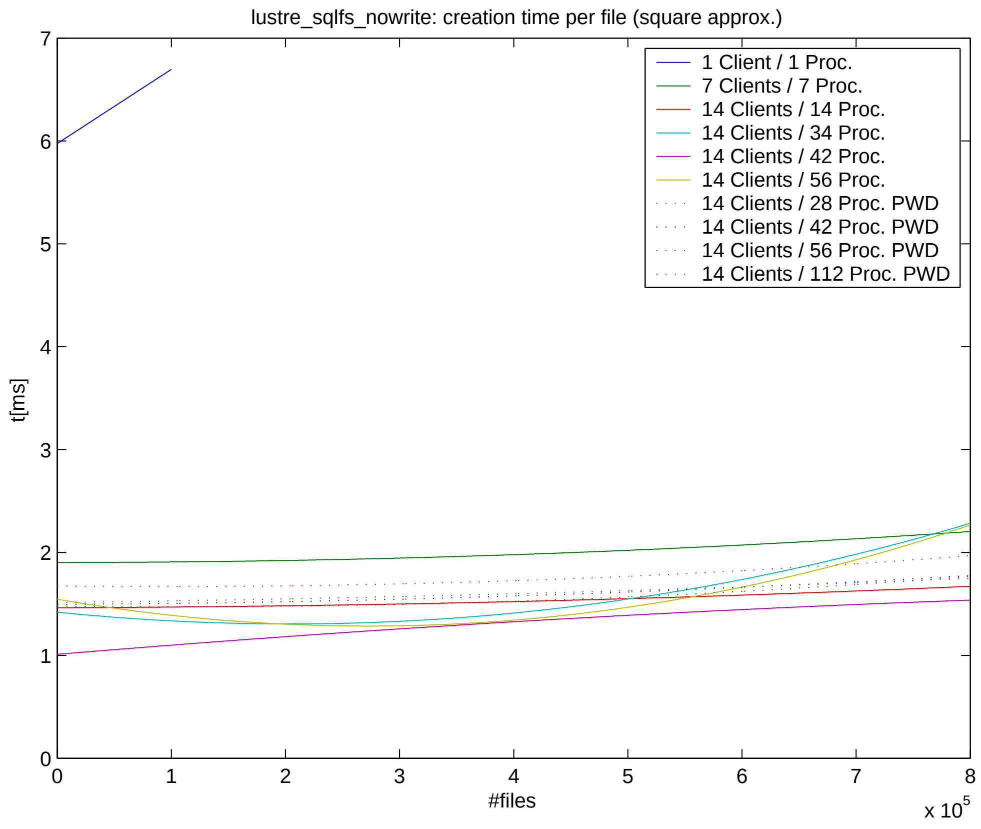
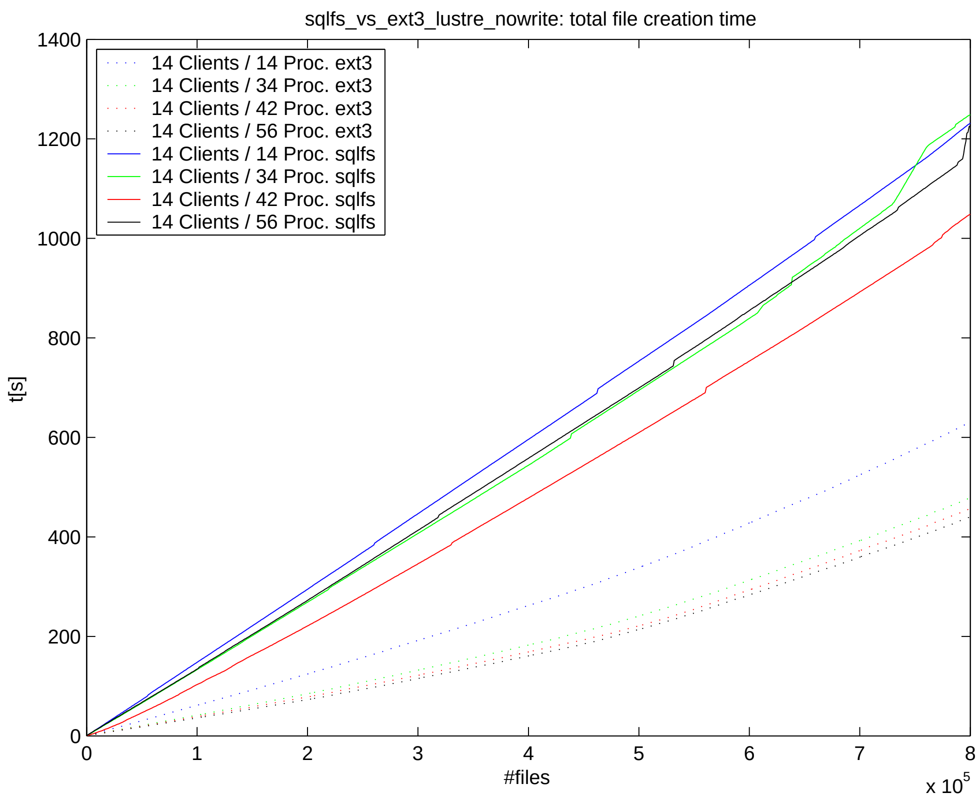


Figure 6.13: Lustre with MDS on sqlfs, time per file



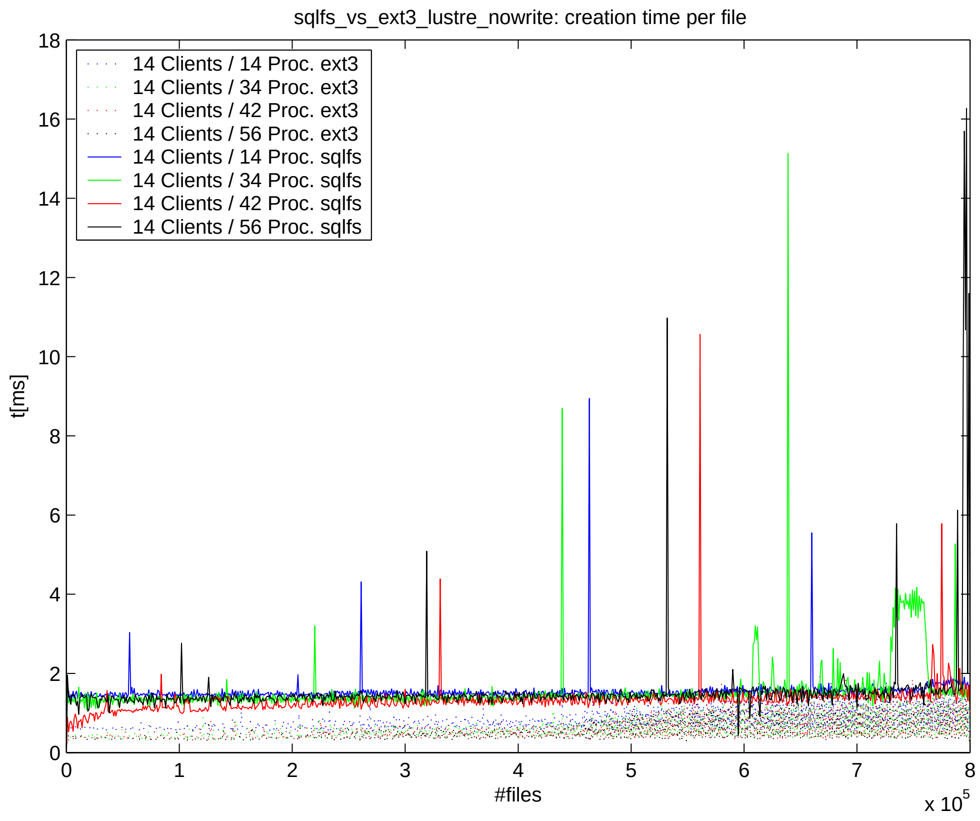


Figure 6.14: Lustre with MDS on ext3 resp. sqlfs

- Like in ext3, in sqlfs the Lustre metadata is stored in xattrs. This metadata could be stored in the sqlfs inode.
- Integrating the database in a higher level than the VFS. By that, more complex operations can be handled by the database, which reduces the number of queries and transactions. In the best case, for a file creation just one transaction and one query is needed.
- Add intent lock capability to sqlfs and modify MDS to make use of it, which shortens the 9 transactions to a single sqlfs lookup call.

Chapter 7

Conclusion and future work

For desktop computers and file servers there exist a couple of different file systems, each with it's own advantages and disadvantages. Network filesystems like NFS are also already available, and global file systems are nothing new (e.g. AFS). So why invent a new file system? Lustre - yet another file system? Certainly not. Although distributed network filesystems, also with global namespace and task sharing, exist already, none of them is open-source. As more and more of the fastest supercomputers and enterprise servers are Liniux clusters, open-source becomes also in the field of supercomputing more and more important. Lustre - with it's ambitious goals - is designed exactly for this purpose: high-performance, redundant, distributed file storage.

With this diploma thesis, a new approach of storing metadata in a file system was made. The purpose of this thesis was not to actually to improve the performance of the metadata server, but to figure out, if it can be improved with a database. Regarding the benchmarks in chap. 6, we did not only show, that a database could be faster than existing file systems, but in some cases we showed that it is faster.
During this thesis, we implemented a fully functional file system (sqlfs), which run's completely in a database system and outperforms classic file systems like ext3 if the directories are big enough. Aditionally, sqlfs supports transactions to meet the requirements as Lustre MDS file system.
This approach is not yet exhausted. As explained in 6.3 there are still many potential improvements, not only to sqlfs but also to the Lustre MDS, as the VFS is not the most optimal interface to a database. Also, different DB-Engines could be evaluated, as PostgreSQL is certainly not the fastest DB-Engine, but perhaps the fastest open-source DB-Engine meeting our requirements.
Due to it's modular design, the sqlfs is not limited to be just the Lustre MDS-filesystem, but can easily be extended to be used in completely different environments.
Possible applications could be:

Hierarchical Storage Management In a Hierarchical Storage Management (HSM) file system, many different storage medias can be used (disk arrays, solid state disks, tapes, ...). On a HSM, policies can be defined. For example it can be defined, that all files greater than 1MB, which have not been used for 10 days are stored on tapes, and all config-files are stored on the solid state disks. Gaining this metainformation in a usual file system is hard work, while in sqlfs it's just one query.

Media File System The idea of this Media File System (MFS) is to store media-dependent information of the file as metadata in the file system. For example, if a picture is copied to the MFS, it's header is parsed and information like size and colour-depth is automatically stored in the file system. For an mp3-file the extracted meta-information could be the artist, the name of the song, release year etc.

Chapter 8

Appendix

8.1 How-To

This chapter describes how to setup and mount the sqlfs filesystem.

8.1.1 Requirements

Linux Kernel 2.4.x
PostgreSQL 7.3.x, 7.4.x recommanded for performance reasons.
libpq

For compilation:
gcc, g++
make
Linux Kernel 2.4.x sources

8.1.2 Obtaining the sqlfs sources

The sources for the sqlfs filesystem can be found on the CD-ROM. Copy them to your target directory.

```
cp -r /{path to cdrom}/db /{path to target}
cp -r /{path to cdrom}/dbclient /{path to target}
cp -r /{path to cdrom}/kernel_module /{path to target}
```

8.1.3 Compiling

Change to your target directory. Note, that you eventually have to change some path settings in the Makefile.

```
cd {path to target}

cd kernel_module
make

cd ../dbclient
make

cd ..
```

Create the sqlfs device (you need to be `root`).

```
mknod /dev/sqlfs c 254 0
```

The first time the module `sqlfs` is loaded, the kernel assigns a major number to the character device. The number 254 has to be substituted with this number. Another possibility is to pass a specific major number as an option to the kernel module with insmod `insmod sqlfs sqlfs_major=....`

8.1.4 Database setup

Install the PostgreSQL 7.3.x or preferably 7.4.x client and server.
(use rpm, apt-get, tgz, ... depending on your Linux distribution)

As user `postgres` do the following (`su - postgres`):

create a user `lustre`

```
createuser -D -A -P -E lustre
```

options:
- -D user is not allowed to create databases
- -A user is not allowed to create users
- -P user needs a password to connect
- -E encrypts the user password stored in the database

Choose which option you need or want.

Create a database `lustre`. (this can be done under any user account)

```
createdb lustre
```

Set the language for stored procedures

```
createlang plpgsql lustre
```

The database is now ready to use.

Connect to the database (use the user account which created the database).

```
cd db/postgres
psql postgres
```

Execute the install.sql script

```
\i ./install.sql
```

8.1.5 Inserting the kernel module and mounting the filesystem

For the following actions you should be root. Change to the directory `sources` in your target directory.

Load the kernel module into the kernel

```
insmod ./kernel_module/sqlfs.o
```

Check if the module was successfully loaded. For this use `lsmod`. It should have an entry called `sqlfs`.

Start the database client

If the database is located on localhost with trusted users:

```
./db_client/dbclient -u lustre -n num_of_dbclients lustre
```

If the database is installed on a remote system with password authentication:

```
./db_client/dbclient [-h host] [-p port] [-u user] [-a auth] [-n num_of_dbclients] db
```

To improve the performance you can start more than one dbclient by adding the argument `-n num_of_dbclients` to the command line.

Mount the filesystem

```
mount -t sqlfs sqlfs /mnt/sqlfs
```

The filesystem is now mounted on `/mnt/sqlfs` and can be used like any other filesystem.

8.1.6 Unmounting the filesystem and removing the kernel module

Unmount the filesystem by executing the following command.

```
umount /mnt/sqlfs/
```

The sql filesystem is now unmounted, but the dbclients are still connected, thus the `sqlfs` kernel module cannot be unloaded. To disconnect them, you have to call the helper program `stopping` which you can find on the CD-ROM.

```
./dbclient/stopping
```

Now you can remove the kernel module from the running linux kernel by calling

```
rmmod sqlfs
```

8.1.7 Obtaining the Lustre Sources

The Lustre Sources can either be downloaded from the Lustre homepage (http://www.lustre.org/downloads.html) or our used Version (1.0.2) can be found on the included CD-ROM.

8.1.8 Setting up & mounting Lustre

An installation manual describing the building and installation process of the Lustre kernel and tools can be found in the Lustre How-To [3].
To include the Lustre sqlfs filter (`fsfilt_sqlfs.c`) driver in your Lustre project, it is necessary to copy the `fsfilt_sqlfs/fsfilt_sqlfs.c` and the `kernel_module/sqlfs_fs.h` files into the `lvfs` directory of your lustre source. To include the `fsfilt_sqlfs.c` in the build process, two lines of the file `lvfs/Makefile.am` of the lustre source have to be modified. These two lines should look similar to the following two:

```
line 25: modulefs_DATA = lvfs.o $(FSMOD).o fsfilt_reiserfs.o fsfilt_sqlfs.o
line 28: EXTRA_PROGRAMS = lvfs $(FSMOD) fsfilt_reiserfs fsfilt_sqlfs
```

Alternatively, the patch `fsfilt_sqlfs/make_patch` can be applied to the source.
Unfortunately it's as well necessary to patch the Lustre `utils/lconf` file. If the file that store the MDS data is a block device, this command initialises the block device, else it assumes that the specified file is an ordinary file that should be used as a loopback filesystem. As a consequence the sqlfs device file `/dev/sqlfs` is treated like an ordinary file, and therefore the initialisation failes. After applying the patch `fsfilt_sqlfs/lconf_patch`, the lconf utility treats the sqlfs device file like a block device.
Now, you can execute the `./configure --with-linux=/your/patched/kernel/sources --enable-zero` command, as it's described in the Lustre How-To and compile the sources.
Now you can create a Lustre config file that suits you, and mount Lustre as it is described in the Lustre How-To. You can find sample config files in chapter 8.1.10. Note, that the file `/utils/mkfs.sqlfs` of the sqlfs file system has to be in your path as it is needed to format Lustre. Don't forget to insert the sqlfs kernel module into the running kernel and to start as much DB-Clients as you like.
The umounting mechanism of Lustre can be found in the the Lustre How-To. To terminate the mounted dbclients you have to to call the `kernel_module/stopping` tool.

8.1.9 How to get debug output

During normal operation the debug output of the kernel module and the DB-Client is disabled. To enable it, the C preprocessor variable __KERNEL__ has to be defined at compile time of these two programs. That can be achieved by uncommenting in their Makefiles the CFLAGS definition that contains that contains the definition of the variable __KERNEL__.
All the debug output is written to the kernel log. Please consult your Linux Users Guide on how to get the kernel log.
All the debug output of the sqlfs MDS filterdriver is written to the Lustre log with debug level D_INFO. You can find a complete Lustre Debugging How-To here [4].

8.1.10 Lustre config files

sqlfs.sh

```
#!/bin/bash

#
# Mikael Feriencik, Thomas Trachsel, , 24-10-2003
LMC=../utils/lmc
FILE=‘basename $0 | cut -d. -f 1‘
FILE=$FILE.xml
#FILE=$1

#set -x
rm -f $FILE

# create nodes
$LMC -o $FILE --add node --node sqlfs2
$LMC -m $FILE --add net --node sqlfs2 --nid sqlfs2-g --nettype tcp

$LMC -m $FILE --add node --node sqlfs1
$LMC -m $FILE --add net --node sqlfs1 --nid sqlfs1-g --nettype tcp

$LMC -m $FILE --add node --node sqlfs0
$LMC -m $FILE --add net --node sqlfs0 --nid sqlfs0-g --nettype tcp

# mds
$LMC -m $FILE --add mds --node sqlfs2 --mds mds1 --fstype sqlfs --dev sqlfs

#lov
$LMC -m $FILE --add lov --lov  lov1  --mds mds1 --stripe_sz 65536 --stripe_cnt 0 --s

# ost
$LMC -m $FILE --add ost --node sqlfs1 --ost ost1 --lov lov1 --fstype ext3 --dev /dev

# clients
$LMC -m $FILE --add mtpt --node sqlfs0 --path /mnt/lustre --mds mds1 --lov lov1
#$LMC -m $FILE --add mtpt --node sqlfs1 --path /mnt/lustre --mds mds1 --lov lov1
#$LMC -m $FILE --add mtpt --node sqlfs2 --path /mnt/lustre --mds mds1 --lov lov1
```

sqlfs2.sh

```
#!/bin/bash

#
# Mikael Feriencik, Thomas Trachsel, 24-10-2003
LMC=../utils/lmc
FILE=‘basename $0 | cut -d. -f 1‘
```

```
FILE=$FILE.xml
#FILE=$1

#set -x
rm -f $FILE

# create nodes
$LMC -o $FILE --add node --node sqlfs2
$LMC -m $FILE --add net --node sqlfs2 --nid sqlfs2-g --nettype tcp

# mds
$LMC -m $FILE --add mds --node sqlfs2 --mds mds1 --fstype sqlfs --dev sqlfs

#lov
$LMC -m $FILE --add lov --lov  lov1  --mds mds1 --stripe_sz 65536 --stripe_cnt 0 --str

# ost
$LMC -m $FILE --add ost --node sqlfs2 --ost ost1 --lov lov1 --fstype ext3 --dev /dev/h

# clients
$LMC -m $FILE --add mtpt --node sqlfs2 --path /mnt/lustre --mds mds1 --lov lov1
```

sqlfs2_ext3.sh

```
#!/bin/bash

#
# Mikael Feriencik, Thomas Trachsel, 24-10-2003
LMC=../utils/lmc
FILE=‘basename $0 | cut -d. -f 1‘
FILE=$FILE.xml
#FILE=$1

#set -x
rm -f $FILE

# create nodes
$LMC -o $FILE --add node --node sqlfs2
$LMC -m $FILE --add net --node sqlfs2 --nid sqlfs2-g --nettype tcp

# mds
$LMC -m $FILE --add mds --node sqlfs2 --mds mds1 --fstype ext3 --dev /dev/hda5

#lov
$LMC -m $FILE --add lov --lov  lov1  --mds mds1 --stripe_sz 65536 --stripe_cnt 0 --str

# ost
$LMC -m $FILE --add ost --node sqlfs2 --ost ost1 --lov lov1 --fstype ext3 --dev /dev/h

# clients
$LMC -m $FILE --add mtpt --node sqlfs2 --path /mnt/lustre --mds mds1 --lov lov1
~
```

8.1.11 Lustre and SQLfs Test Machines

All Test were made with RedHat 9, Postgres 7.4.1, Kernel RH 2.4.20-28.9-lustre1_0_2, Lustre Release Version 1.0.2

Testmachines

sqlfs0 - sqlfs2	Dual 1Ghz Pentium III Xeon, 512 Mb RAM, 40 Gb
sqlfs3, sm1 - sm14	Dual 2 Ghz Pentium 4 Xeon, 1Gb Ram, 40 Gb

Bibliography

[1] R. Bischoff, R. Hunger, A. Muehlemann: *Parallel Distributed Filesystem with Metadata Database*, Student Thesis, Electronics Laboratory, ETH Zuerich, 2003.

[2] Peter J. Braam: *The Lustre Storage Architecture*, Cluster File Systems, Inc., December 2003. `http://www.lustre.org/docs.html`

[3] *Lustre - A HowTo Guide*, Cluster File Systems, Inc., December 2003. `http://wiki.clusterfs.com/lustre/LustreHowto`

[4] *Lustre - A HowTo Guide*, Cluster File Systems, Inc., December 2003. `https://wiki.clusterfs.com/lustre/LustreDebugging`

[5] Daniel P. Bovet, Marco Cesati: *Understanding the Linux Kernel*, 2nd Edition, O'Reilly, December 2002.

[6] A. Rubini, J. Corbet: *Linux Device Drivers*, 2nd Edition, O'Reilly, June 2001.

[7] M. Beck, H. Boehme, M. Dziadzka, U. Kunitz, R. Magnus, C. Schroeter, D. Verworner: *Linux-Kernelprogramierung*, 6th Edition, Addison-Wesley, April 2001.

[8] Donald A. Lewine: *POSIX Programmer's Guide*, O'Reilly, March 1994.

[9] Maurice J. Bach: *The Design of the UNIX Operating System*, 2nd Edition, O'Reilly, May 1986.

[10] Moshe Bar: *Linux Filesystems*, Osborne / McGraw-Hill, 2001.

[11] K. Douglas, S. Douglas: *PostgreSQL*, 1st Edition, Sams Publishing, February 2003.