Software project WS 2008/09

DBFS - Database Filesystem

Timo Minartz

April 11, 2010

supervised by Julian Kunkel

Contents

1 Concept and problem case

The goal of this software project was to implement a lightweight filesystem with FUSE [Sou] to map filesystem sources and database tables in one namespace for a multiuser environment.
The concept of this software project is based on an existing problem case, which is discussed here first. Within the scope of a research project a program generates lots of tiff-Files in a specific hierarchical order. These tiff-Files are explicitly identified by a *collaboration*, *project*, *plate*, *replicate*, *well* and *file name*. This results in an file structure like collaboration/project/plate/replicate/well-filename.tiff. The different tiff-Files need to be evaluated by a set of different *applications* and the output data should be administered by a database system containing all data from all *applications*. The different *applications* are used at the project layer meaning a *project* can be evaluated by different *applications*. The basic goal is to realize a mapping between the database and the filesystem in one namespace to abstract the database access for the different *applications*. To realize the mapping, two additional layers need to be added to the path, one for the different *applications* and one containing the application data at *well* level. The resulting filesystem (named "Fuse filesystem" or "Dbfs") structure is like collaboration/project/application/plate/replicate/well/filename.tiff. In the *well* folder are several files containing the application data (i.e. .../well/data). This files are mapped from the database in this filesystem while the tiff-Files are mapped directly from the "base filesystem" (the tiff-Files are not stored in the database). The files mapped from the database will be named "virtual files" and similarly the *well* folders "virtual folders".
Furthermore there are some constraints to take care of which are described in the following: At first, all files existing in the "base filesystem" must be mapped in the Dbfs with their existing filesystem permissions. These files are used to store meta data information and so named "metadata files" in the following. For the "virtual files" writing and reading access must be supported. The access to the tiff-Files have to be limited to read these files. Every other operation has to be illegal, such as create/remove files, change permission/ownership etc. This means every change to the "fuse filesystem" data must be done in the "base filesystem" rather in the database. Furthermore more permissions at the application level should be implemented to allow users (meaning the operating system users) access to the underlying folders and files. Except for these described practical constraints some further requirements exists, so the resulting Dbfs is benchmarked to evaluate the filesystem performance.
The next chapter discusses the software design followed by a chapter containing the implementation details. This document ends with the benchmark results and future work for this project.

2 Software design

This chapter is a description of the software design to realize the former mentioned requirements. First the table layout is discussed followed by a section containing the feature of permissions for the Dbfs. The next section deals with managing the Dbfs directory structure and how changes to the structure can be realized. In the last section optimizations and restrictions of the design are discussed.

2.1 Table layout

The table layout is chosen because of the following requirements:

- keep the table layout simple
- simple handling for the application layers
- make it easy to use SQL functions on each application layer

For every *application*, there are two tables necessary: One for the file permissions (discussed in the next section) and one for the files. The one for the files will be created automatically based on the one with the permissions, so the user does not have to take care about the creation.
This file table is named by the following convention `collaboration-name_project-name_application-name`. Hence, for every a*application* (even in the same *collaboration*) a new table has to be created. This is also true for each of the different *projects* in the same *collaboration*. This scheme is chosen to realize the second requirement about the simple handling for the application layers: It's easy to create or delete *applications* for an existing project (see also the next section).
Each of these file tables has at least three columns for the *plate*, the *replicate* and the *well*. For every *well* a new row is added to the table, meaning for the paths plate/replicate0/000 and plate/replicate0/001 two rows are added with the values `plate,replicate0,000` and `001` respectively (see table 2.1). This database design is not even in the second normal form, but because of the requirements only one table is used to contain all the file oriented data. This results in easy dropping, creating or copying tables for different *applications*.

Table 2.1: *Example database table for two well folders in the same replicate and plate folder*

plate	replicate	well	testfile
plate	replicate0	000	"content for testfile located in plate/replicate0/000/testfile"
plate	replicate0	001	"content for testfile located in plate/replicate0/001/testfile"

If a file should appear in the *well* folder, a new column has to be added to the table. This file appears now in every *well* folder of every *replicate* and/or *plate* of the specified *application*, but the contents differ because of every *well* is saved in an own table row. This file can be one of the SQL field types, means `varchar` or `integer`. Other field types are also possible but so far not implemented. The use of `varchar` or `integer` leads to a simple restriction: If `integer` is chosen as file type, only one value can be stored in the file. Every editing of the file, i.e. echo 4711 > integerFile will overwrite the table value, similarly the command echo 4711 > varcharFile. Attention should also be paid to the fact that appending to a file is not supported, the call will be interpreted as a normal write call.

Every column is default defined as a `varchar` with a limit to 30 and 3 characters rather for the *well* column, but this value can easily be changed in the table definition or the program code. If a write call to a field is bigger than the max value, the rest is discarded but the writable characters are written. The three main columns are combined to a primary key for this table and therefor not `nullable`. Every added column is also not allowed to be **null**, so a default value has to be specified (an empty string in case of a `varchar` column, 0 otherwise).

2.2 Permissions

As mentioned in the previous section, the second of the two necessary tables is the one for the permissions. If a new project should appear in the Dbfs, a new table must be created by hand or by using the GUI tool (see next section for using the GUI tool). Based on this permissions table the filesystem is able to generate the file table to complete all necessary conditions. It's important that the permissions specified as mentioned used for the files stored in the table ("virtual files"). The tiff-Files linked in the *well*-Folders are write protected for everyone. The other linked files ("metadata files") inherit the "base filesystem" permissions. It's also not allowed to change the directory structure by hand (creating and deleting can only be done by changing the database tables, see next section)
The naming convention for the permission table is `permissions_collaboration-name_project-name`.
The table has a simple structure, only two columns for name (`varchar`) and owner (`integer`) have to be specified (see figure 2.2).
The permissions are on the application layer, so for every *application* a data row is added to the table which identifies the owner by his operating system user id. If no owner is specified, the default owner is user id 0 (root).
The rows in the permission table initiate the program to use (or create if not yet existing) the application layer for the filesystem.

Table 2.2: *Example permission table `permissions_collaboration0_project0` to specify two applications*

name	owner
application0	1000
application1	1001

2.3 Managing the directory structure

To manage the directory structure should attention be paid to the different types of files in the Dbfs:

1. Files linked from the "base filesystem" (tiff-Files and "metadata files")
2. "Virtual files/folders" from the database

To remove tiff-Files or metadata files, the "base filesystem" has to be changed. This results in changes to all application layers. To change or remove folders relative to the *application* folder, the corresponding entries in the file table in the database have to be changed. Also for the creation or deletion of the "virtual files" the tables have to be edited.
For abstracting the SQL language for the user, a GUI tool is implemented. This tool shows all relevant entries of the Dbfs based on the tables and makes it easy to perform the updates mentioned above (see figure 2.1). The "base filesystem" path has to specified at first, than the corresponding database tables are read and displayed. The desired *collaboration* can be selected, the corresponding *projects/applications* can be edited. For every *application* the "virtual files" can be created or deleted

(editing is not possible, so delete first and than recreate). For creating the desired file type and the length must be chosen.

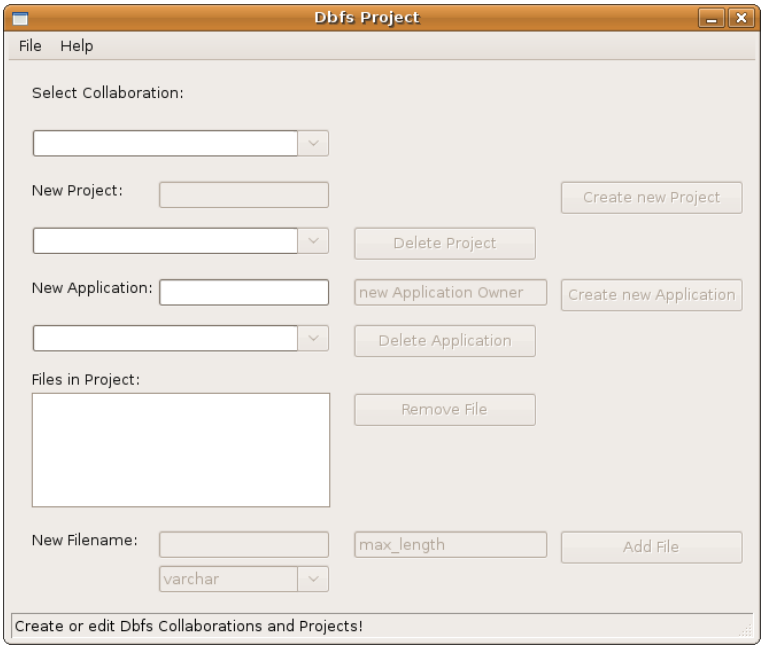

Figure 2.1: *Graphical user interface to manage the fuse filesystem directory structure*

2.4 Optimizations and restrictions

As seen in early tests of Dbfs, simple navigating in the directories leads to a lot of database queries. This behavior is caused by various checking by the fuse filesystem implementation: If the look up of a path is desired, every entry is validated. So the simple command `ls collaboration0/project0/app0/plate/replicate0` results in some database queries:

- check permissions from permission table
- get a list with all well folders
- check for each well if it is a valid virtual folder, if not repopulate database
- check for each well its permissions

Of course there is also some checking done by the "base filesystem", but this is not part of this document.
To reduce the query overhead, a buffer has been implemented which can be enabled in the source code. The main use case is to avoid querying the database for every virtual folder to exist relative to an application, so the list of wells is buffered. If there's no buffer hit, the database is queried and the buffer is rebuilded. The implementation of the buffer itself is very simple at the moment (only a vector containing database items), but the main problem about using a buffer is a consistency problem: The buffer could not recognize base filesystem changes, so the buffer has to be manually cleared after the base filesystem changed (can be done by changing the working directory in the fuse filesystem into another application). Because of this restriction, the gain of the buffer should be balanced, on the one hand the database relief, on the other the (possible) consistency problems. The main result is that caching makes only sense for more or less static systems, where the "base filesystem" isn't changing frequently.
But anyway (even with or without cache) lots of database queries are produced in a multiuser system, so the database connection must be handled

- thread safe and
- efficient.

To get this handled, a thread safe pool for database connections has been implemented. The pooling can also be disabled/enabled in the source code.

3 Implementation

This chapter deals with the implementation of the Dbfs design discussed in the previous chapter. In this chapter an overview of the implemented classes and special adaptions for the programming language is given.
The implementation of Dbfs has been done in C++. The implemented classes can be spread into four modules:

- module for handling filesystem issues
- module for database access
- module for GUI access
- helper module

The filesystem module contains the classes **Fusexx**, **Dbfs**, **Filesystem** and **Virtualization**. The class **Fusexx** plays a special role for using fuse with C++, this class is amongst others discussed in the next section. The **Dbfs** class is the entry point for the "fuse filesystem" and contains the supported filesystem operations

- getattr
- readdir
- read and
- write.

The two different virtualization layers are implemented in a static way, meaning at a path depth of 3 and 5 respective the virtual application/well layer is added to the path. These values can easily be changed at program start up. The class **Filesystem** encapsulates the "base filesystem" access and the class **Virtualization** maps the fuse system paths to real system paths and the other way around.
The database module contains the classes **Database**,**QueryResult**, **ConnectionPool**, **DbCache** and **DbManagement**. The class **DbManagement** realizes the mapping between the "fuse filesystem" calls and the database queries.
The GUI module is an wx-widgets implementation of a graphical user interface which interacts directly with the database module.
For further information the code is documented in-line with doxygen and the code documentation can be generated using make doc.

3.1 Adaptions for C++ to use fuse

As mentioned previously, the class **Dbfs** is the fuse entry point to the program. This class inherits class **Fusexx** which contains definitions of the real fuse interface. In this class are all filesystem operations defined, out of this the class **Dbfs** only needs to overload the desired functions. This class is mainly a interface between the C fuse interface and C++ and calls the **fuse_main** macro. All fuse options passed to program are passed through the fuse main macro. Because of the described fuse program forks a new process, an output to stdout is with standard options not possible. To control this behavior, two solutions are possible: Logging to a file (done in class **Log** which is used by the implemented filesystem) or calling fuse with the argument fuse -f which results in fuse not forking a process. The parameter -f can easily be added to the fuse params when calling dbfs. This parameter

is especially important if you want use a tool like valgrind for debugging. By the way, to use valgrind with fuse an adjustment to the operating system is needed to get it work, see the README file in the sources for further instructions. To get the header **Fusexx** compiling, the import header **fuse.h** must be installed on the system. For Ubuntu Hardy/Intrepid Ibex this is the libfuse-dev package. To get the mysql and wx-widgets headers the packages libmysqlclient-dev and libwxgtk2.8-dev are required.

4 Benchmarks

In this chapter the performance of the implemented fuse filesystem is analyzed by a bandwidth of different benchmarks to cover almost all use cases. At first the benchmarking process with underlying file operations is described, followed by the testing (and developing) environment. In the last section the results are discussed.

4.1 The benchmark process

The benchmarking process is designed to cover different use cases:

1. reading filesystem attributes
2. reading different sized blocks from a "metadata file"and a tiff-File respectively
3. reading "virtual files"
4. writing "virtual files"

These four benchmarks are performed with a set of different options:

- Dbfs / tmpfs as benchmarked filesystem
- tmpfs filesystem for the "base filesystem"
- ext3 / tmpfs filesystem for the mysql database (see filesystem dependencies)
- clean database (only permission tables already created) or dirty database (full set of tables and rows already created)

Before all tests (excluded the dirty database test) the test folders are cleared and remounted to avoid caching from the underlying filesystem and the mysql database respectively. Also has to take care of denying other operating system processes to read/write the test data, so all daemons like the *trackerd* (used under Ubuntu for indexing files to speed up the search) have to be disabled for the duration of the tests. Please notice that all tests are local ones, so network issues like latency etc. are not measured here. Another point is that all benchmarks are performed with buffer disabled but pooling enabled.

4.2 System environment

This section includes a short description of the testing and developing system. The hardware system was composed of a dual core processor with 1.4 GHz per core and 2048 MByte RAM. The hard disk was a 120 GB SATA disk with 5400 U/min and 8MB Cache with a average seek time of 11 ms. The operating system installed was Ubuntu 8.10 (Intrepid Ibex) with kernel 2.6.27-11-generic as normal user installation using an ext3 filesystem. The mysql version for server and client is version 5.0.67 without any changes for performance issues.

4.3 Results

In this section the results of the different benchmarks are summarized and explained. The corresponding data to each benchmark is located in the test folder of the project. Each figure displays the operations per second (Ops) value for a specific operation on a different set of options for the filesystem (see the section about the benchmarking process) which are more precisely described on each specific figure. These values are based on the real Time.Real time is the elapsed real (wall clock) time used by the process (see man time).

4.3.1 Metadata

The first benchmark evaluates the performance for reading file system attributes. So the values for reading filesystem attributes for tmpfs and Dbfs are compared in figure 4.1. The first box displays the Ops when performing the command ls -lR in the directory root containing the *collaboration* folder (this folder contains 115200 folders, subfolders and files) mounted with a tmpfs filesystem (no database interaction needed). This takes about 3 seconds and results in about 38.000 Ops. The same command in the directory root mounted with Dbfs and an empty database (only containing the permission tables) takes about 200 seconds meaning about 600 Ops. This is about 65 times slower. To be able to determine the time needed for the tables to be created, a third measure has been performed. In this measure the corresponding tables are already created and filled with the data. This test takes about 20 seconds less than the one before, but this is still about 59 times slower. Out of this, the benchmark shows that read the filesystem attributes in Dbfs is the most expensive operation (compared to the other implemented operations as seen in the next sections). This results from the fact that this operation is already a expensive one in the Fuse core implementation and now the database access slows the operation extra down. The outsourcing of the database to a tmpfs filesystem doesn't result in any significant changes for the measured time (only 5 percent faster than the test run with the database on ext3 filesystem).

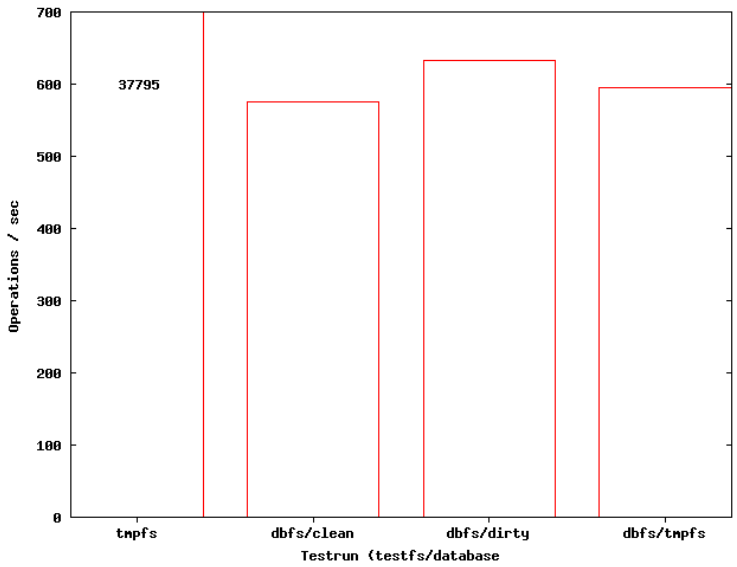

Figure 4.1: *Reading filesystem attributes from ext3/fuse filesystem and a different set of filesystems for the base filesystem and the mysql database respectively. Performed is a ls -lR command for a collaboration folder with 115200 sub-folders/files*

4.3.2 Physical files

The next benchmark (see figure 4.2) displays the Ops while reading data from "metadata files" and tiff-Files (both linked in Dbfs) with different block sizes. Each Operation performed 10.000.000 times on the same file in the same folder. A write benchmark doesn't make sense here because write access

to the tiff-Files is not allowed to the user. The three boxes (the first for reading blocks from tmpfs, the second one for Dbfs "metadata files", the third one for Dbfs tiff-Files) doesn't differ significantly. As expected, the tmpfs value is the fastest one (independent on the blocksize) followed by the metadata value. Reading tiff-Files is once again a bit slower. This results from the fact, that reading from tiff-Files is associated with a few database accesses to check if its a valid access.

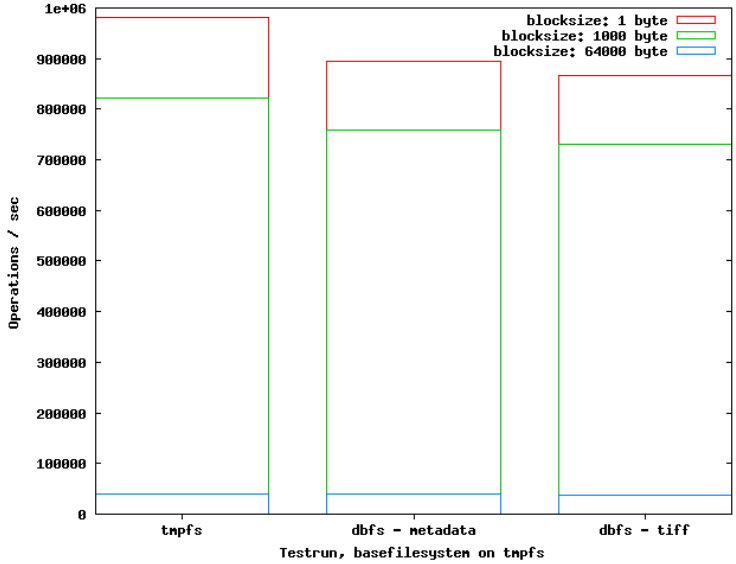

Figure 4.2: *Read test for the physical files in the fuse filesystem with different block sizes and files (metadata and tiff-Files). For this test, the same block from the same file is read 10.000.000 times*

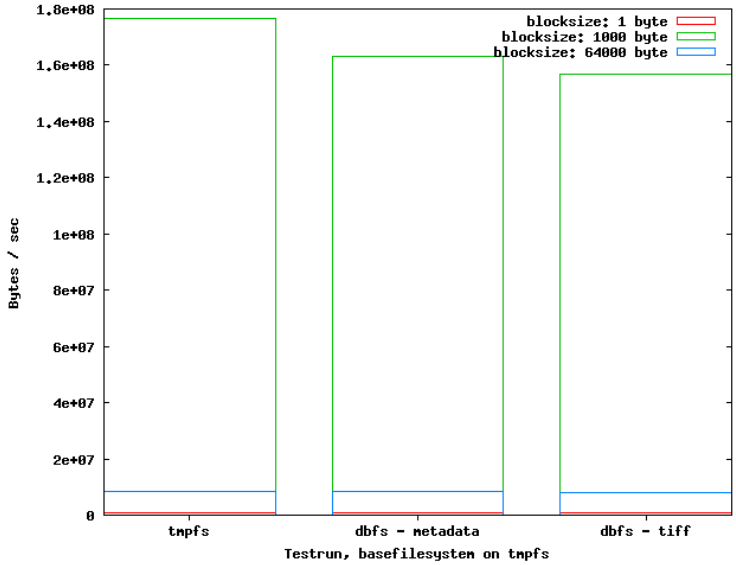

Figure 4.3: *Read test for the physical files in Dbfs and tmpfs with different block sizes, but the values are calculated to see the time needed for reading 1 byte on the different filesystems based on figure 4.2*

But figure 4.2 doesn't show the real time needed to read one byte depending on the block size. So the next two figures (see figures 4.3 and 4.4) displays these values (based on the same measured data). The characters of the underlying filesystem (first box) and the Dbfs (second box and third box) doesn't differ significantly even from this point of view. Reading with a block size of 1000 Bytes is the fastest way, while reading with a block size of 64 kBytes takes longer. The read procedure with a block size of 1 byte works as slowly as expected with the factor 8 and 9 respectively in comparison with the block size of 64 kByte.Out of these values, the simple pass-through to the underlying filesystem seems to work quite well irrespective of the block size.

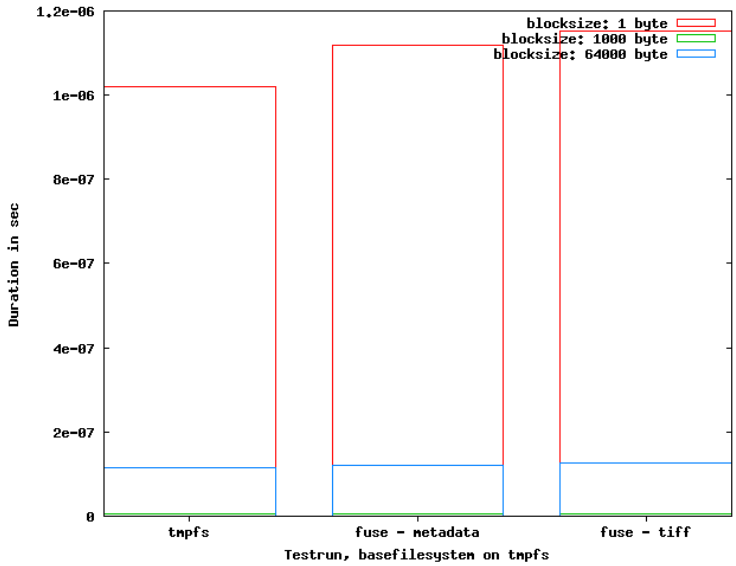

Figure 4.4: *Read test for the physical files in Dbfs and tmpfs with different block sizes, but the values are calculated to see the bytes per sec written based on figure 4.2*

4.3.3 Virtual files

The last benchmark (see figures 4.5, 4.6 and 4.7) evaluates the write/read performance for files stored in the database. For this benchmark 11520 (virtual) files are created (virtual files in tmpfs means small files). From this different vitual files are a few byte read/written. The first box on figures 4.5 and 4.6 displays the time for read/write operation on a tmpfs filesystem for comparison. The next two boxes on each figure are the results of Dbfs. One of each with the database on ext3 and tmpfs filesystem respectively. Reading "virtual files" can be performed with about 85 Ops, this is nearly 3 times slower than reading from tmpfs. Outsourcingthe database to a tmpfs filesystem also doesn't gain any performance in this case. Writing "virtual files" (see figure 4.6) is about 63 times slower than writing to tmpfs. This is only slightly less than the overhead for reading the filesystem attributes. The big overhead results from the fact that one the one hand writing to tmpfs is very fast and on the other hand writing to the database takes a lot of time. If the database is located on a tmpfs filesystem the overhead can be reduced to about 58 times.
In the last figure (figure 4.7) the read and write performance are compared, based on the same measured data as in the previous figures. The "base filesystem" is for both operations a tmpfs filesystem with the database on ext3. The first box displays the read-Operations per second, the second one the write-Operations per sec. As seen in the last two figures, with a relative view to the operations read seems to be the faster operation, but with a absolute view the write operation is about 1.5 times faster. This results from the effect (as seen in the last figures) that the read operation from tmpfs takes a lot more time than the write operation.

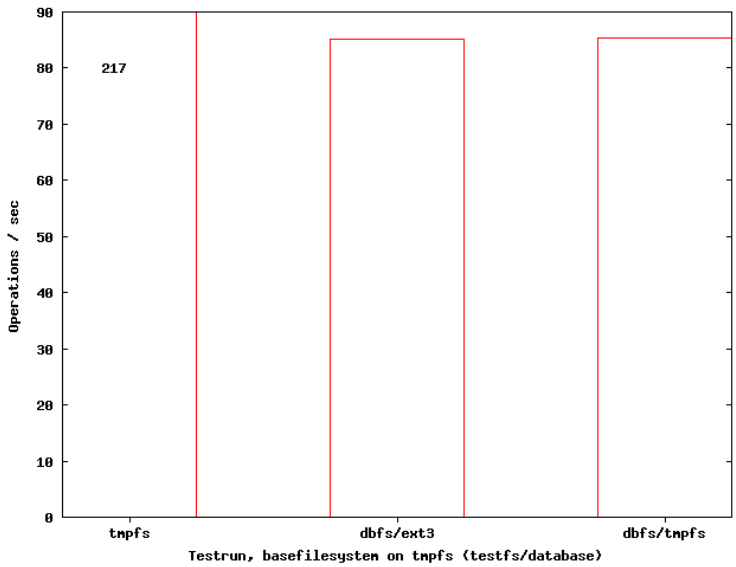

Figure 4.5: *Read test for 11520 (virtual) files with mysql database on tmpfs/ext3 filesystem*

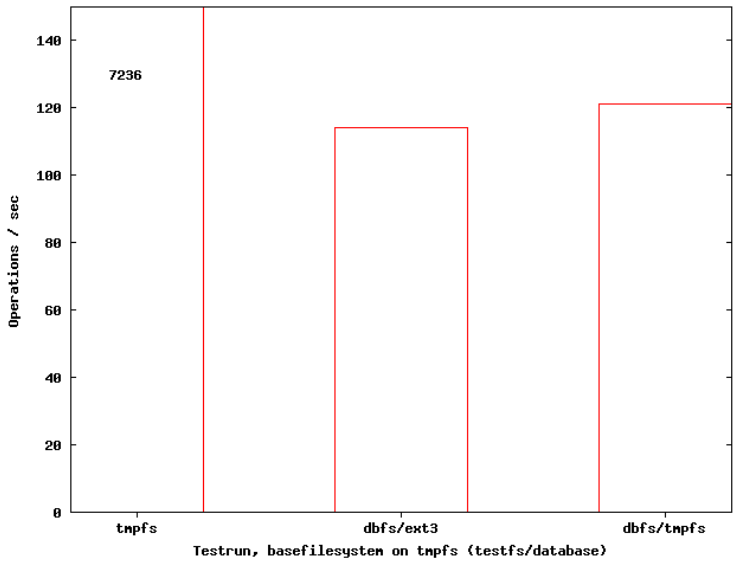

Figure 4.6: *Write test for 11520 (virtual) files with mysql database on tmpfs/ext3 filesystem*

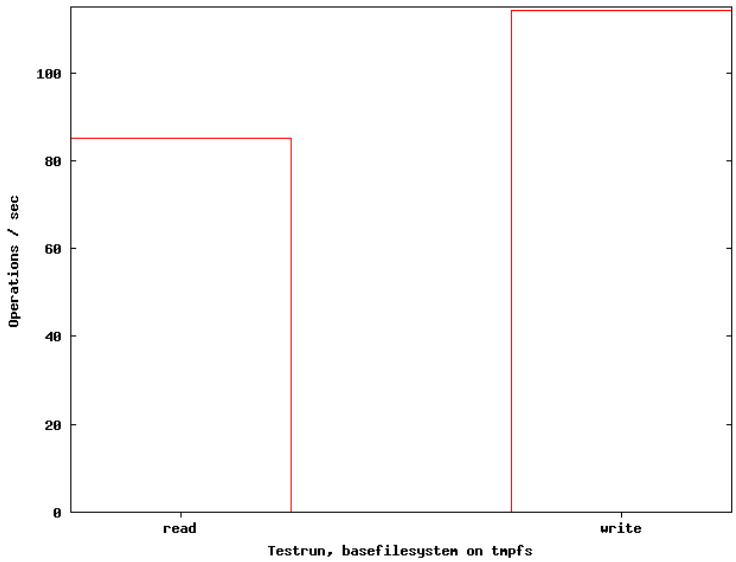

Figure 4.7: *Comparison between read and write test for virtual files with mysql database on ext3 filesystem based on the data of figures 4.5 and 4.6*

5 Conclusion and future work

The evaluation of the software project shows the described problem case of mapping filesystem source and database tables in one namespace can be solved by a fuse implementation. For this specific case the read performance for the physical files stored on the underlying filesystem (especially the tiff-Files) is interesting and this performance doesn't differ significantly from the underlying filesystem owing to the pass-through mechanism. The performance to access the "virtual files" stored in the database is not so good. But the bottleneck for this access doesn't seems to be the database access itself, because changing the filesystem for the database doesn't give any significant performance gain. Reading the filesystem attributes is the operation with highest overhead (about 65 times slower) followed by the write operation for "virtual files" (about 63 times slower). The read operation (for "virtual files") generates an overhead about 150 percent (means 1.5 times slower) while reading the physical data files produces almost no overhead. Because of this behavior, the concrete use case must determine about using this implementation.
Because of the main goal was to evaluate the feasibility, some implementation factors aren't included. So the database access statements are manually constructed in the source code, so the program is vulnerable to SQL injection attacks. To minimize this vulnerability the code has to be customized to use mysql prepared statements. Another fact is that implementing the different virtualization layers in a static way is no flexible solution if further layers should be added. A more dynamic solution could minimize the the future work for this additions.

Bibliography

[exa] *ROFS, the Read-Only Filesystem for FUSE.* http://mattwork.potsdam.edu/projects/wiki/index.php/Rofs

[IG] IEEE, The ; Group, The O.: *The Open Group Base Specifications Issue 6.* http://www.opengroup.org/onlinepubs/009695399/functions/contents.html

[Mic] Microsystems, Sun: *MySQL 6.0 Reference Manual.* http://dev.mysql.com/doc/refman/6.0/en/index.html

[Sou] Sourceforge.net: *Main Page - fuse.* http://apps.sourceforge.net/mediawiki/fuse/index.php?title=Main_Page