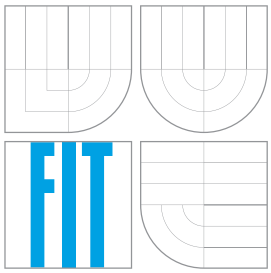
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

SQL BACKEND PRO SUBVERSION

SQL BACKEND FOR SUBVERSION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. JAN HORÁK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. TOMÁŠ KAŠPÁREK

BRNO 2010

Abstrakt

Práce analyzuje systém pro správu verzí Subversion a dostupné backendy pro ukládání dat na serveru. Tyto backendy porovnává a popisuje návrh a implementaci nového backendu, založeného na databázovém systému MySQL. Jsou analyzovány obecné přístupy ukládání stromových struktur v relační databázi, různé možnosti práce s indexy a byla provedena řada dílčích analýz, které jsou využitelné i v jiných aplikacích.
Návrh vychází z existujících backendů, jenž byly brány jako zdroj informací i při samotné implementaci. Nový backend byl implementován a zkušebně integrován do aktuální verze Subversion, nicméně zatím není implementována plná funkčnost, takže oficiální součástí systému není. Backend je v závěru porovnán s existujícími backendy BDB a FSFS a jsou navrženy další možnosti pokračování.

Abstract

The thesis analyses version control system Subversion and its available backends for storing data in a repository. It compares these backends and describes basic features of a new SQL database backend. Design and implementation of the new backend, based on MySQL database, is described and the new backend is then compared with existing backends BDB and FSFS.

Klíčová slova

Subversion, verzování, revize, SVN, SCM, Berkeley DB, FSFS, MySQL, DAG, APR

Keywords

Subversion, versioning, revision, SVN, SCM, Berkeley DB, FSFS, MySQL, DAG, APR

Citace

Jan Horák: SQL backend for Subversion, diplomová práce, Brno, FIT VUT v Brně, 2010

SQL backend for Subversion

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Tomáše Kašpárka. Další informace mi poskytli programátoři systému Subversion. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Jan Horák
May 21, 2010

Poděkování

Děkuji panu Ing. Tomáši Kašpárkovi za odborné vedení a věcné připomínky.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The work considers analysis, design and implementation of a new backend to store data in Subversion system. Subversion is an application used for management of a software source code most often with parallel access and ability to keep every change in history.

The chapter Analysis of the Subversion system presents the whole system, discuses dividing to layers and describes the lowest layer – storing data backends. The chapter contains a description of backends which are available at present time and author tries to compare advantages and disadvantages of them.

Later the analysis of requirements is discussed, considering a new backend based on SQL database. The design of the backend follows, oriented to MySQL. The last part describes implementation of the new backend and some performance and responsibility tests. The backend is compared with other existing backends and the next work is advised.

Chapter 2

Analysis of the Subversion system

This chapter describes the Subversion system as deep as we need to understand the function and integration of backends for storing data in a repository. The architecture with layered design is described shortly, the backend layer for storing data is described more detailed. Backends, that are available now, are described and their features are compared in section 2.8.

2.1 Purpose of the Subversion system

In the book *Version control with Subversion* [1] authors of Subversion describe the system as a fantastic hammer, but we cannot view every problem as a nail. When we are deciding of the deployment of Subversion, we need to think, if the project needs Subversion at all.

Generally open source Subversion system handles a management of a versioned data (most often a source code of software). The system is free to use and these are its main features:

- **Client-server architecture** – a server handles storing of a data and clients use that data. There are several alternatives, how to get the data from the server, depending on protocol, security reasons, performance, etc. The system allows using the server and the client on one computer, as well as on different computers connected using a network.

- **Keeping changes** – besides the actual version of data every change made to the data is kept. The system allows restoring any of versions from the history, even if there is a newer version of the data stored on the server. We can look at the data as an ordinary file system tree, which has one dimension more (the time). In other words the data stored on the server are sequences of snapshots, while we can see every snapshot as a file system tree.

- **Ability to work in parallel** – The Subversion system has been designed for parallel work of many clients. A client usually do the following: gets data from the server, changes the data and sends them back to the server. The system allows to do all these changes parallel by many users, without locking the data or even parts of them. Despite that, the system can keep the data in integrity and no change can be done without cognition of the user. The way how this is implemented, is called copy-change-merge solution and we can read more about this in [1].

2.1.1 Suitable deployment of Subversion

As mentioned before, the Subversion system is most often used to manage a source code of applications. It is useful utility for manage data, which are centralized on one place, but changed by many users from different places.

If a mistake is discovered, data can simply be restored to the original version. This can be done at any time in the future, even if the data are changed several times after that bad change. System stores changes in files data as well as in file properties (file property changes were not kept in CVS – Control Version System [28]).

On the other hand, Subversion is not suitable to management data, which are static or which we don't need to keep history of. There are better utilities to handle these situations.

2.1.2 History of Subversion

The history of Subversion began in 2000, when CVS had been a system used to management software source code the most often. CVS has many disadvantages, so CollabNet company (http://www.collab.net) decided to implement a new system, which will replace CVS.

CollabNet contacted author of the book *Open Source Development with CVS*, Karl Fogel, and he and others (Ben Collins-Sussman, Briean Behlendorf, Jaseon Robbins and Greg Stein) created a community of active contributors, which staied behind Subversion development untill recently. The community is now organized by The Apache Software Foundation (http://www.apache.org), which manages many other open source projects.

Note: The Subversion project has been accepted into the Apache Incubator just recently, it had been organized by Tigris.org before. The move was announced on Wednesday, November 4th, 2009. More about that on [4] or [25].

2.2 Other version control systems

Some other version control systems (or configuration management systems) are shortly described later in this chapter. More information about comparison can be found on [18] or on [49].

2.2.1 CVS

CVS (Concurrent Version System) was the most used version control system to control source code a few years back. Subversion is straight coming out of the CVS and does not want to break a good known and used way to management source code. On the other hand authors tried to get the best of the CVS and remove or minimize its disadvantages.

The CVS system composes from one central repository and every change is uploaded back to repository. It does not recognize changes in a directory tree and revision numbers are related to files, not to whole repository (like it is in Subversion). Branching and merging are not usable in the CVS, there are problems with binary files and locking is possible only explicitly by user. Despite many disadvantages the CVS has been long time the most used SCM, but now there are many alternatives, which are shortly described in the next paragraphs. More about CVS on [28].

2.2.2 Aegis

Aegis is a transaction-based software configuration management system, but it is not used very often. It concentrates integrity and testing data and uses distributed way to share data. It's not suitable to use in a network and it is relatively complicated to manage the system.

Compared to Subversion it offers disconnected commits, peer-to-peer architecture and it uses a filesystem-based storing data backend (no SQL or embedded database are integrated). More about Aegis on [38].

2.2.3 Arch

Arch system fixes some problems, which CVS had. It is similar to Subversion when a directory tree is changing or new branches are created. Thanks to support of standard protocols (FTP, SFTP and WebDAV over HTTP or HTTPS) it is easy to deploy and to use it on the Internet.

It is particularly useful for public free software projects, because of it's easy to learn and to administer. It's a distributed system. More about Arch on [27].

2.2.4 Bazaar

The system is implemented in Python, it is very simple with basic configuration, but it's very scalable using many plug-ins (new storage formats can be plugged-in too). It is well portable and able to manage even large projects. There's no need to choose between central and distributed version control tools, Bazaar directly supports many work-flows with ease.

The system supports many best practices including re-factoring, pair programming, feature branching, peer reviews and pre-commit regression testing. With true rename tracking for files and directories, merging changes from others simply works better. More about Bazaar on [33].

2.2.5 BitKeeper

BitKeeper is a very powerful, capable and reliable version control system that supports copies, moving and renaming, atomic commits, change-sets, distributed repositories and a propagation of change-sets, 3-way merging, etc. It is portable to all major UNIX flavors, and to Win32.

BitKeeper has a few drawbacks. It can only duplicate a repository and work against it and cannot work against a working copy of a snapshot. This makes large checkouts over a WAN very slow.

2.2.6 Darcs

An open source (GNU GPL) distributed version control system with a very simple repository creation and some interesting features (like spontaneous branches). An implementation in Haskel, not very supported and used language, speaks against its massive deployment. More about Darcs on [6].

2.2.7 Git

Git is a free and open source distributed version control system influenced by commercial BitKeeper. It is designed to handle everything from small to very large projects with speed and efficiency. In the beginning it was designed to use many other version control systems by, but now it is an independent project. System is often used by Linux distributions and the Linux kernel development. It supports a non-linear distributed development, branching, merging etc.

Following the UNIX tradition, Git is a collection of many small tools written in C, and a number of scripts that provide convenient wrappers. Git provides tools for both easy human usage and easy scripting to perform new clever operations. More about Git on [3].

2.2.8 Mercurial

A distributed system developed as a reaction to moving the source of the BitKeeper from open source to a commercial sphere. It was designed with the intention of being small, easy to use, and highly scalable. It is often marked as the fastest version system. Mercurial is a platform independent system written in Python and C. It offers many extensions and uses many commands known from Subversion. More about Mercurial on [41].

2.2.9 Monotone

Monotone is a distributed version control system with a different philosophy. Namely, change-sets are posted to a depot (which the communication with is done using a custom protocol called netsync), which collects change-sets from various sources. Afterwards, each developer commits the desirable change-sets into his own private repository based on their RSA certificates and possibly other parameters.

Monotone identifies the versions of files and directories using their SHA1 checksum. Thus, it can identify when a file was copied or moved, if the signature is identical and merge the two copies. It also has a command set that tries to emulate CVS as much as possible.

The Monotone architecture makes an implementation of many features easier. It is not without flaws, however. For example, Monotone is slow, and doesn't scale well to large code-bases and histories. More about Monotone on [40].

2.2.10 Perforce

Perforce is a centralized, commercial (non-free) solution for version control. It is very fast, very stable and robust. It scales very well, and has a good reputation. It requires an annual per-seat licensing, but it is also available for interested open source developers under a gratis license. More about Perforce on [43].

2.2.11 PureCM

PureCM project is a commercial, cross-platform SCM, which offers some features, which are not implemented in many other SCMs – a stream hierarchy view of the data, a command-line interface as well as a GUI interface in base, a checkpoint support for a parallel work, merging on the server, a visual folder diff, etc. More about PureCM on [34].

2.2.12 Vesta

Vesta is a mature software configuration management system that originated from an internal use by Digital Equipment Corporation (now Compaq/HP). It is a replacement for CVS and Make and provides more than a mundane revision control systems. It is an open source under the LGPL.

Vesta is a portable SCM system focused on supporting of development software systems of almost any size, from fairly small (under 10,000 source lines) to very large (10,000,000 source lines). More about Vesta on [5].

2.3 Using Subversion

This section takes a short look at an ordinary work with the Subversion version control system.

2.3.1 Repository and working copies

Subversion is a centralized (not distributed) system, where data are stored on the server, so-called *repository*. Repository is a directory tree with files and users can get the actual directory tree as well as any of the trees from history. Every change in the repository assigns a new incremental unique number to it. We call this number a revision number and the revision number 0 means an empty directory (it is a convention used by Subversion).

We need to create so-called *working copy* for working with the data from the repository. The working copy is a copy of a part of the data from the repository on the client's file system. A client can make as many changes as he wants and then the data are copied back to the repository. An operation that provides a copying the data back to the repository is called *commit* and it is designed as an atomic transaction. During the transaction data are checked if no newer data has been uploaded before (by another user). If there is a version conflict detected, the user (who is doing the commit transaction) is responsible to get all changes from both users to be applied and he is responsible to keep the data in an integrated state.

Subversion uses the copy-modify-merge way and no explicit locking is needed for a parallel access. It seems to be uncomfortable because of conflict resolutions, but the used way is much faster in the result as compare with the lock-modify-unlock solution for example.

2.3.2 Branching and Merging

Branches and *tags* are well known from many other version systems, but were not implemented well in CVS. Both are actually copies of the actual version of part of the repository. Branches are used to develop new features or new versions of the software, tags are used to store the snapshot of the some version (for example an official releases).

Subversion does not force users to keep a strict directory tree structure, nevertheless the most often organization is the following:

- /trunk – the actual version, which is under development at all the time and which is updated from branches
- /branches – development branches, which are created, updated and removed in time
- /tags – the tags are creating in time, but not changed nor deleted any more

2.4 Joining to Subversion community

Subversion development began by CollabNet Company and continues under Apache Software Foundation (ASF) now. Thanks to CollabNet a contributor's community has been created and now it is a middle-sized open source community with a specialization to development and maintenance utilities and tools to improve the collaboration on the software development. Its motto sounds *We're here to help you help us!*

Every programmer can join the community in many fields – the analysis, design or implementation of the tools, general libraries, tests or documentation. The community likes to get new suggestions for new projects or to overtake some existing ones.

Joining the Subversion development means to join the Apache Subversion mailing list and sign the Individual Contributor License Agreement. Often form of contribution is bug logging or debugging, but in the case of this thesis it is needed to join the development, to communicate and resolve problems with other contributors.

2.4.1 Working on an open source project

In the beginning of the work it was not clear how the community will adopt the intention to implement a new backend. Without the interest to cooperate, this work would be done without the community. However, it was clear from their first reply, they welcome the idea to make the system better and a help was offered to discuss the design or the implementation.

For the implementation of the new backend a new development branch should have been created, but it hasn't been done yet (the reason is described in 5.2). The new backend can be added to an official release in the future, but it is not possible right now, because of a premature state of development.

All communication on open-source projects under Tigris.org or ASF takes place only using mailing lists, so their subscription was the first step to communication. Considering that the development of new features on such a big project does not put in charge more than couple of contributors, the communication is not unnoticed using this simple way.

2.4.2 Coding style

Every new developer, who collaborates with ASF community, has to agree with conditions organized to establish an easier cooperation. Many projects are implemented in C/C++ language, which does not force programmer to follow general indent style, comments writing etc.

Reading or updating a source code in other coding style is difficult, so some rules (similar to GNU code style) had been established and every contributor should use them. The common language of the documentation is English, which should be simple and single valued. It is also the reason, why this thesis is written in English.

2.5 External libraries used by Subversion

Subversion uses some general libraries, which handle some parts of the system. These libraries are mentioned in the following paragraphs.

2.5.1 Apache Portable Runtime

The mission of the Apache Portable Runtime project (APR) is to create and maintain software C/C++ libraries that provide a predictable and consistent interface to underlying platform-specific implementations. The primary goal is to provide an API, which software developers may code in, and to be assured of predictable if not identical behavior regardless of the platform on which their software is built, relieving them of the need to code special-case conditions to work around or take advantage of platform-specific deficiencies or features.

APR was originally a part of the Apache HTTP Server, but it has been spun off into a separate project of the Apache Software Foundation, and it is used by other applications to achieve platform independence.

The range of a platform-independent functionality provided by APR includes:

- Memory allocation and memory pool functionality
- Atomic operations
- Dynamic library handling
- File I/O
- Command argument parsing
- Locking
- Hash tables and arrays
- Mmap functionality
- Network sockets and protocols
- Thread, process and mutex functionality
- Shared memory functionality
- Time routines
- User and group ID services

APR also offers an extension for working with MySQL database (APR DBD), which is a kind of wrapper for standard functions from mysql.c library, including a query preparation and transaction handling. APR only changes an interface of those functions to correspond with other APR functions.

More about APR on [23] and more about APR DBD in 4.5.1.

2.5.2 BDB (Berkeley database)

An open source, embeddable Berkeley database [22] from Oracle offers a fast, reliable local persistence library with zero administration. It is often deployed as an *edge* database and it provides very high performance, reliability, scalability, and availability for applications that do not require SQL.

BDB stores arbitrary key/data pairs as byte arrays (no database scheme), and supports multiple data items for a single key. Key and data are specified only by byte length and no

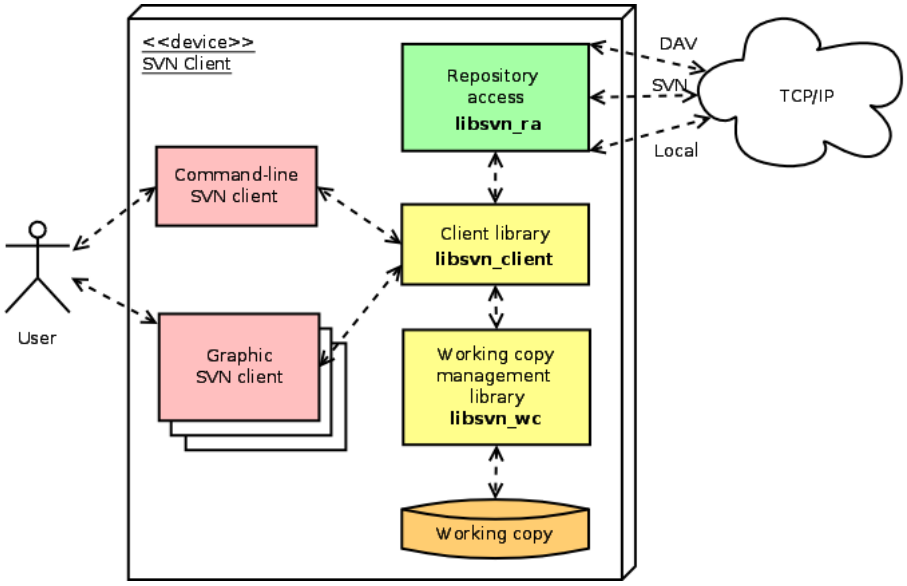


Figure 2.1: Client part of Subversion [1]

other structure is defined (it depends only on a developer). BDB has bindings in C, C++, Java, Perl, Python, Ruby, Tcl, Smalltalk, and other programming languages.

Despite very simple design, BDB is a universal embedded database, which allows to store even terabytes of data as well as kilobytes. The database has very fast indexing and a sequence access, it offers some advanced features like locking, shared memory, logging, backup and replication. Transactions in BDB follow ACID (atomicity, consistency, isolation, durability).

More about Oracle Berkeley DB on [22] or in [21].

2.6 Subversion Architecture

Generally, if we want to implement a new feature of the system, we should know the rest of the system. Figures 2.1 and 2.2 describe the Subversion architecture fastest. A reader will understand where important components are located, but detailed structure of them is not needed generally. Only important components are described in more detail later.

The server communicates with the client application using several Internet protocols, based on TCP/IP. User always works with his working copy, not with the repository directly. The repository itself can be directly accessed only by a repository administrator, using several routines, such as *svnadmin*, *svnlook*, etc.

Client applications are graphic or command-line clients using the *libsvn_client* library to process commands, *libsvn_wc library* to work with working copy and *libsvn_ra* library to access the repository. Client's application can be integrated in OS, like the TortoiseSVN does it in Windows. The entire client's side is independent to the backend used to store the data in the repository, so we don't need to describe the client's part any more.

We can recognize a layered architecture on the server too (see the Figure 2.2). Every layer has its own interface, while the most important interface for this thesis is the repository interface.

Layers communicating with a client use the *libsvn_fs* library to access the repository

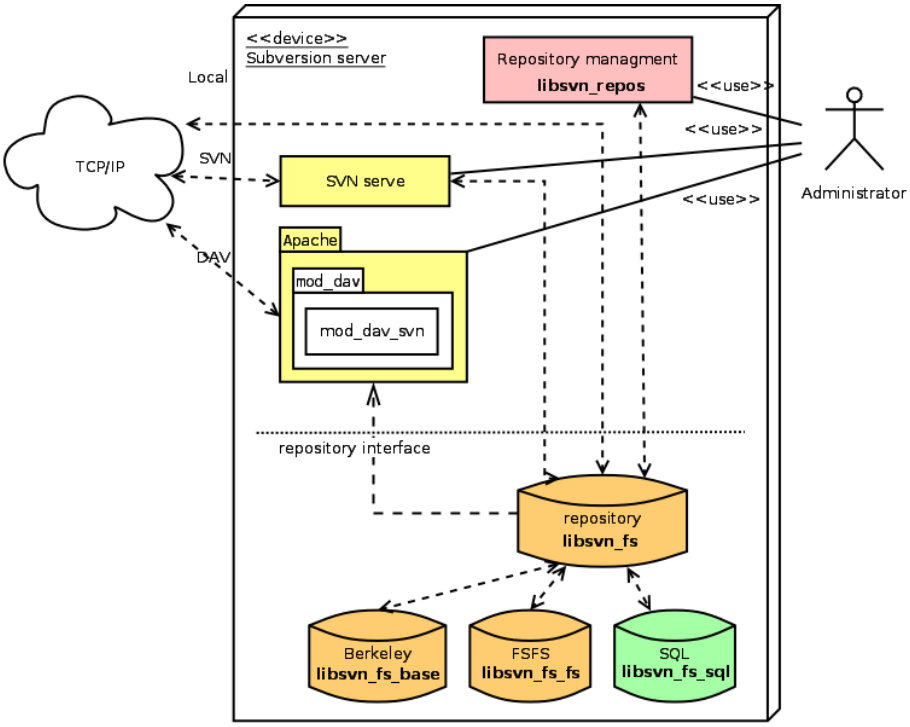


Figure 2.2: Server part of Subversion [1]

data itself. This library offers a general interface to work with the repository and uses backend-specific libraries. The backend libraries are the most important libraries for our intention.

In the present Subversion supports backends that store data in BDB and using native OS file system – backend FSFS. The aim of this thesis is to add a new backend, which will store data in SQL database (concretely in MySQL). The new backend is inserted to the Figure 2.2 and as we see, thanks to the layered libraries design, we will touch only two libraries (library *libsvn_fs* to connect the new backend and the most part of work will be done in the new backend library).

More information about other layers can be found in [1].

2.7 System layers

We have mentioned the architecture divided into layers, while every layer or library has strictly defined interface. Every library is a part of exactly one layer, which can be the following:

- Repository Layer
- Repository Access (RA) layer
- Client layer

Let's take a short look on all libraries [1]:

- **libsvn_client** – Primary interface for client programs

- **libsvn_delta** – Tree and byte-stream differencing routines
- **libsvn_diff** – Contextual differencing and merging routines
- **libsvn_fs** – Filesystem commons and module loader
- **libsvn_fs_base** – The Berkeley DB filesystem backend
- **libsvn_fs_fs** – The native filesystem (FSFS) backend
- **libsvn_ra** – Repository Access commons and module loader
- **libsvn_ra_local** – The local Repository Access module
- **libsvn_ra_neon** – The WebDAV Repository Access module
- **libsvn_ra_serf** – Another (experimental) WebDAV Repository Access module
- **libsvn_ra_svn** – The custom protocol Repository Access module
- **libsvn_repos** – Repository interface
- **libsvn_subr** – Miscellaneous helpful subroutines
- **libsvn_wc** – The working copy management library
- **mod_authz_svn** – Apache authorization module for Subversion repositories access via WebDAV
- **mod_dav_svn** – Apache module for mapping WebDAV operations to Subversion ones

The fact that the word *miscellaneous* appears only once in the previous list is a good sign. The Subversion development team is serious about making sure that functionality lives in the right layer and libraries.

2.7.1 Repository layer

Repository layer consists of libraries *libsvn_fs*, *libsvn_fs_base* and *libsvn_fs_fs*. The first one offers an interface for the higher layer, so the Repository Layer can access all backends using only one interface. The library *libsnv_fs* offers almost similar operations, as an ordinary file system – creating, changing, moving and deleting files. However, we cannot forget that all changes are stored in time forever.

We have mentioned the commit operation and we called this operation a transaction. Let's have a short look how this type of transaction differs from the transaction, as it is commonly known from database systems. The main purpose is the same – to ensure, that data will remain in consistent state. This is ensured following the ACID (atomicity, consistency, isolation, durability) set of properties.

Subversion transactions are much bigger than ordinary database transactions, while many relatively complicated operations are done during one Subversion transaction and it could last for several minutes. But the final effect is the same – the transaction will proceeded all or not at all. Transactions are always related to a sub-tree in the repository. It could be one file or even the whole repository.

2.7.2 Backend FSFS

The FSFS backend was first integrated in Subversion 1.1 and from Subversion 1.2 it is the standard backend. FSFS is versioned file system implementation, that uses the native OS file system directly, rather than via a database library or some other abstraction layer, to store the data [1].

The FSFS abbreviation can be misunderstood with Fast Secure File System, which is a user-space, client-server distributed file system, but is not used by Subversion.

The philosophy of implementing FSFS has been to make the best of the files, which are the most important parts of the operation system. A similar way is often used to store data in other version control systems.

2.8 Comparison of backends FSFS and BDB

Many features are very similar in both existing backends, however, some differences could be found. The following paragraphs take a look at some of these differences and compare FSFS and BDB backend [1].

2.8.1 Performance and reliability

BDB is very reliable and keep data in integrity, but there had to be done some recovery operations by a user in older versions from time to time, because the database had got to an inconsistent state. From version 4.4 BDB database uses an auto-recover system, so no manual recovery actions have to be done any more. FSFS is generally reliable, except some bugs in the history, which were rarely demonstrated, but were data-destroying.

BDB is more sensitive to unexpected interruptions, it can be left wedged and some recovery procedures have to be done. FSFS is not very sensitive to interruptions. BDB cannot be accessible from a read-only mount (FSFS can) and does not have a platform independent storage format (FSFS has). FSFS can be used over network filesystems, while BDB can't and FSFS has a smaller repository usage on the disk.

FSFS is faster, when there are many files in the directory, but there used to be problems with a large number of revisions (many entries in one directory were the problem, but this disadvantage has been repaired already). FSFS is generally faster (especially if we store smaller files, see section 2.8.3), but commits last sometimes very long.

2.8.2 Description and design of benchmark tests

Both of the existing backends have been tested for speed and disk usage requirements. A new benchmark in Python was implemented. It should simulate many different operations, usually done by user while working with Subversion. The tests are configure-able to simulate various behaviors.

A repository and a working copy were hosted on the same PC to avoid the net quality in-correctness. SVN protocol has been chosen to communicate to simulate the ordinary work.

Two identical tests have been launched on every backend, while pseudo-random generator has been initialized by the same seed. Every time a lot of small files (kilobytes) first, then less large files (megabytes) has been tested.

One test composed from four steps:

- creating files
- modifying files
- combination of operations (creating, modifying and deleting files)
- deleting files

Note: Tested on a computer with Intel Core2Duo T5550, 1.83GHz, 2GB RAM, HDD Hitachi HTS542525K9SA00, 5400rpm, Ubuntu 9.10, Linux Kernel 2.6.31-16.

2.8.3 Comparison backends using benchmark tests

The results of the tests are mentioned in the Table 2.1 (size of the repository) and in the Table 2.2 (speed of operations). The same results are shown in the Figure 2.1 and in the Figure 2.2.

	backend BDB	backend FSFS
Empty	1456	68
Filled	314572	206040

Table 2.1: Repository size [kB]

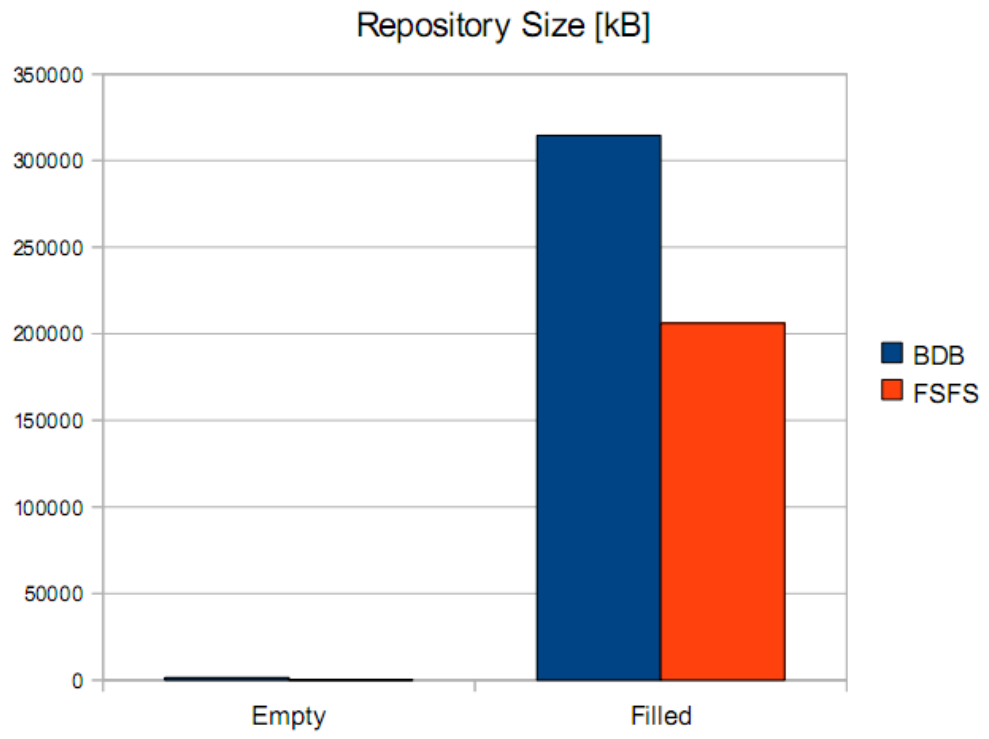


Figure 2.3: Graph of the repository size

We can see, that the size of the FSFS repository is by one third smaller, then when using the BDB backend.

	backend BDB	backend FSFS
Small files – creating	212.0	164,0
Small files – modifying	256.0	230,0
Small files – various actions	354.0	319,1
Small files – deleting	220.0	219,0
Large files – creating	4200.0	4224,0
Large files – modifying	114.3	78,3
Large files – various actions	1500.0	1544,8
Large files – deleting	100.0	104,6

Table 2.2: Operations speed [s]

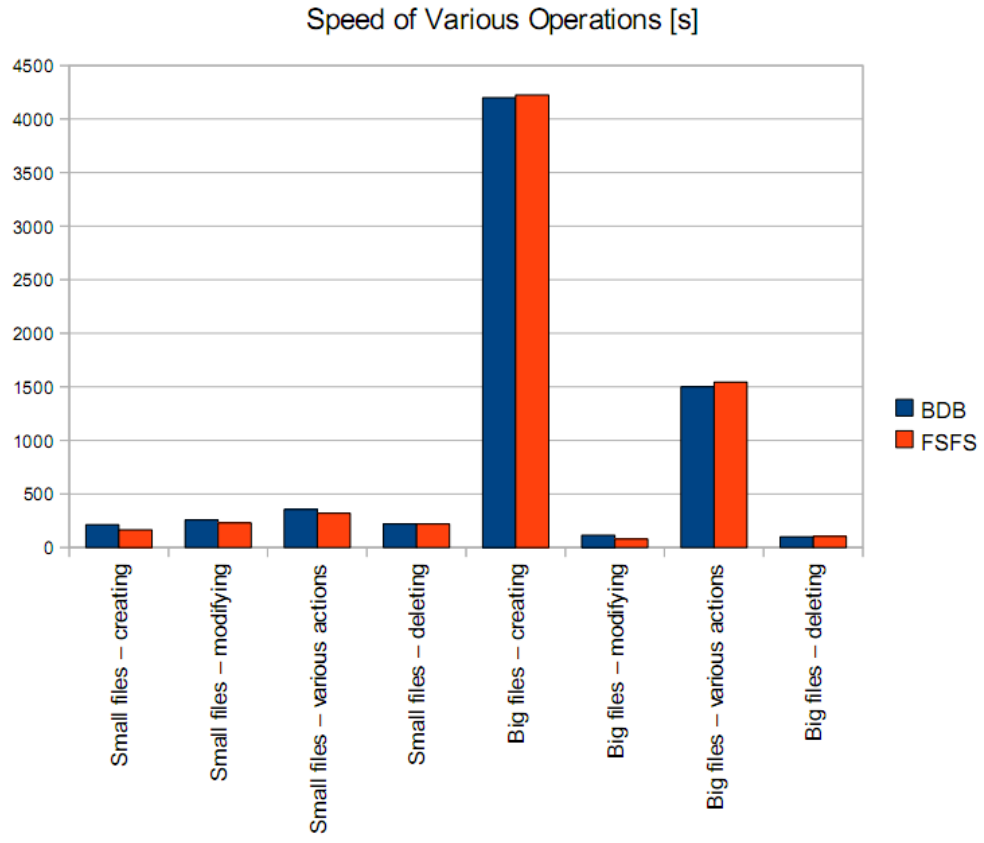


Figure 2.4: Graph of the repository commintting speed

We can see that the speed is very similar in both backends, however, FSFS is a bit faster. Thanks the good result of the FSFS, it is the default backend from Subversion 1.2.

2.8.4 Repository administration

Generally the repository is a part of the system, that does not need any maintenance, but it is good to know something about its parts, if something gets wrong. The repository has the following directory structure:

- conf – configuration files (for example repository access is set here)
- db – the data in the repository
- hooks – scripts for additional work with repository, providing extended features
- locks – files implementing the locking of some files in the repository

Note: *hook* scripts are launched similar to triggers in some database, generally before or after some actions (for example sending emails after commits).

2.8.5 Known issues of existing backends

Accessibility

BDB is generally portable, but it is not possible to move a repository to another system using a simple copying. The FSFS is more suitable for this kind of moving, so it is portable without limitations. However, we can use the *dump* function for moving BDB as well as FSFS.

If we want to use a network file system to host the repository data, we have to use FSFS backend, while BDB backend is not able to work in the network environment.

Data in repository is generally unreadable to human and we can work with them only using special applications. If we want to see the data in human-readable format, we can use the mentioned *dump* function. This function creates so-called *dump file*, which can be got back to the repository using a *load* function. Despite this feature the repository is still very hard to read by human.

What is important, the dump file is portable and compatible for both backends, so we can make a dump file from BDB repository for example and load it to other repository, which is based either on BDB or FSFS backend.

The dump file can be used to backup repository too. We can even choose what revisions should be covered up to the *dump file*. Then we can load these *dump files* back in the same order (for example if we are recovering a weakly backuped repository). After the loading we have all files in the repository even with their history up to revision 0.

FSFS generally doesn't suffer from portability issues and is able to move from one system to another.

Scalability

If we consider the scalability, it is e.g. not easy to add new indexes in FSFS, relatively much code has to be rewritten and the indexing itself is relatively complicated feature. Generally all changes in the data scheme will involved relatively many changes in the code.

Performance

In FSFS backend the last part of the commit operation can sometimes last too long and also the head revision checkout is sometimes very slow. Despite that the FSFS is still faster than BDB when we work with small files. BDB could be a bit faster if we work with large files, but it is relatively seldom situation, Subversion is used for management of many smaller files more often.

Reliability

There were serious reliability problems in FSFS in history and some recovery actions had to be done manually from time to time in BDB before version 4.4. But in the present time it seems there are generally no serious reliability problems known any more.

2.9 Expected features of the SQL backend

Generally we cannot expect much better performance from an SQL backend in comparison to FSFS and BDB backends, but it can offer a lot of new possibilities. Some of them are mentioned here.

There is one interesting point to implement the SQL backend – some potential users decided to not use Subversion because of simply absence of that type of backend. A report by user John will represents similar point: *„SVN doesn't really need an SQL backend (FSFS rocks), but it would make a lot of people feel better, if their repositories could be stored in regular SQL Server ... something they understand and feel comfortable with“* [35].

A little less serious point wrote by David Weintraub: *In large corporate environments, this can be a selling point. Typical Pointy Headed Manager's Comment: „SQL! That means it must be good“* [20].

So we can say, many users will be interested in Subversion much more, if it will have the SQL backend.

Scalabiliy

Mark Phippard, Subversion contributor, wrote (2005-03-07): *„I can picture a large hosting site like SourceForge using a clustered SQL repository that is front-ended by a large number of load-balanced Apache servers and getting very good response times. Since you would get a robust client/server architecture for free with most SQL engines, it offers a lot more possibilities for intelligently and safely splitting the workload across machines“* [20].

Adding new indexes to tables is very simple operation (much simpler than in FSFS and BDB) and indexing itself is on higher level, than in FSFS, and comparable with BDB.

We can choose the SQL database engine from very big amount of available variants, while the changes in the code to adding a support for the new database system will not be very large. Generally the SQL backend will be suitable for large projects using a well customized database, while the customization will be easy.

Performance

We should consider (like David Anderson wrote in Subversion mailing list at 2006-01-19), that *„Subversion backends are generally storage systems that store a versionned file system, which is a series of interconnected DAGs (Directed Acyclic Graph or we can call it trees) ...*

This is a problem, because SQL is great at many forms of data crunching, but representing a tree is one thing it is definitely not good at doing. There are algorithms which make it fairly easy to represent a single tree in a SQL database, with fast read access to subtrees, but slow write access ... Storing a sequence of interrelated trees in a SQL database is a Hard problem, and that is why we don't yet have a SQL backend for Subversion" [20].

So we cannot expect better or even the same speed of operations, like the FSFS or BDB backend can offer. We also can expect worse results considering the database size, while the data from tables itself will be larger comparing to FSFS and BDB and some more data will be stored for indexing.

However, SQL engines can offer better indexing and caching, than FSFS and BDB backends can offer. How much this influence the total speed, it is a question. I suppose the speed could be better only on large projects with a powerful database engine (e.g. Oracle).

We can expect faster head revision checkout and a finalization of a commit comparing to FSFS, so the SQL backend could be more suitable for installations where many readers access the repository at the same time.

Reliability/Backup/Recovery

FSFS and BDB offer backup and recovery operations, but that is available only using specific Subversion utilities. FSFS could be backed-up with standard file system utilities too, but that doesn't have to be good enough for some system administrators, because it doesn't correspond with their already using system backup work-flows/infrastructure.

Kevin Broderick wrote the following comment about potential advantages of SQL backend at 2005-07-15: *„ ... Many, if not all organizations already have databases of some sort (or of multiple sorts) in place. That usually implies that the infrastructure around the database – network, server, backups, admin tools, monitoring, etc. also in place. ... I realize that most organizations should also have filesystem backups in place, but my experience is that the databases get more attention and are a bit easier to monitor." „With that said, I think that FSFS provides a reasonable option and I'm not sure that a SQL backend would provide much in the way of feature benefits for the current user base"* [20].

Generally SQL backend will provide an easy integration to company's work-flows, a simple backup and recovery without needs of special utilities.

Accessibility

As mentioned before, SQL backend would be well suitable for larger installations and it wouldn't be problem to store big amounts of data in some power-full database (e.g. Oracle).

SQL backend will be (like FSFS) platform-independent, it would be accessible using network and it would be read-able by human without need of using *db_dump* utilities (more about *db_dump* in [1]). Dominic Anello wrote: *„it can offer more robust query interface into the repository (we can use queries like "Where were all modifications to somefile.h made„ or "What tags have been made off of the project-2.3.1 branch„ without using the log)"* [20].

The Figure 2.5 shows the possibility to create ad-hoc queries in the repository in various backends. We see that the SQL backend offers the best support for specific queries applied to the repository data.

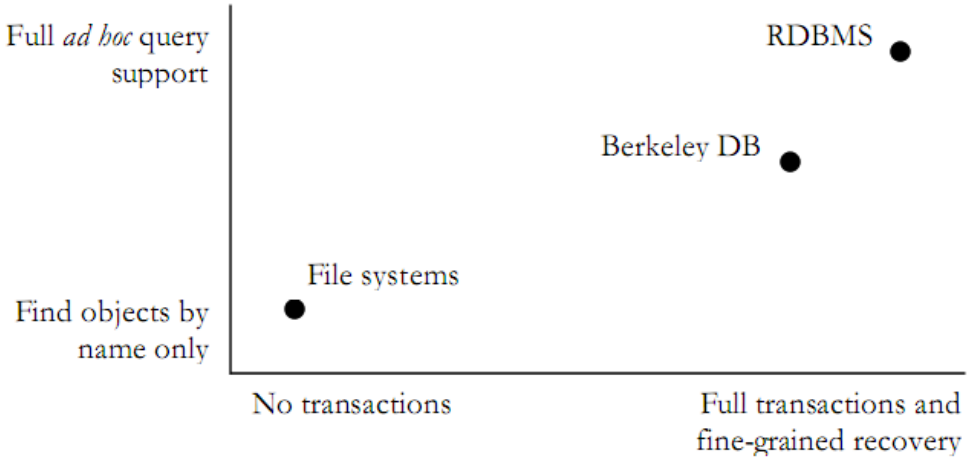


Figure 2.5: Range of Storage and Query Services [42]

2.10 Subversion analysis conclusion

Enough information needed for analyzing and designing of the new backend was the result of the semestral project. Big part of the work was to become familiar with the Subversion system as a user and the backend layer has been studied in detail. The work with existing backends continued, because it was possible to define requirements to the new SQL backend from their design and implementation. This is described in section 3.2.

The design of the SQL tables and the implementation in C/C++ language followed as a main part of the master thesis. The implementation raised from existing backends, but both of them, FSFS and BDB, are very different from SQL database; so many issues were solved individually. The result of the design is in sections 3.13.

Communication with other Subversion contributors continued during design and implementation, because they know the Subversion system much better and they had many good ideas (see 3.9, 3.10 and 3.11).

Chapter 3

Analysis and Design of the MySQL backend

Design of the MySQL backend is described in this chapter. Berkeley and MySQL databases are generally different, but if we use them for the same domain model, structures of the databases are a bit similar. Thus, MySQL backend database scheme design raised from BDB scheme. On the other side, there are many things specific for each database, such as indexing, lists storing, data types, etc.

In the beginning of this chapter (sections 3.2 and 3.3) necessary details of the existed Subversion filesystems are studied and the main features of existing backends have been taken as the source of information for design the new one. A general scheme of any backend is described in sections Subversion filesystem scheme and Base data model of the Subversion filesystem.

An example of execution of two basic commit operations is demonstrated in the section 3.4 as the result of sections 3.2 and 3.3. The purpose of this example is to describe all effects in the abstract backend, that have happened during these operations. This should help to understand how a backend works.

Next sections (3.6, 3.5 and 3.7) consider possible ways to store specific data structures in a relational database, which operations could be applied to that data and how fast the operations can be. Namely the universal approaches to store DAG structures in SQL databases are discussed to determine, if we can profit by one of the approaches. Then an access speed to large tables is tested, especially the use of different data types for indexes.

The next section 3.8 studies the performance differences of storing small and large files in relational database instead of a pure OS filesystem. After these researches the first version of MySQL database scheme of the backend was designed and offered to other Subversion contributors to comment. Some useful changes and possible issues were discussed by all interested persons and the discussions are shortly mentioned in sections 3.9, 3.10 and 3.11.

In the end of this chapter some general ideas about optimizing MySQL database are discussed in section 3.12. The last section 3.13 presents the results of the analysis and design of the MySQL backend and a MySQL database scheme of the backend is described.

3.1 Existing MySQL backend prototype

There have been some attends to develop SQL backends in the history. For example there is one very old prototype of a backend on [31], which uses MySQL database, besides BDB.

The prototype originated in early time of Subversion's existence and many things changed from that time. For example the FSFS backend was added and the whole backend interface has been redesigned.

The way how the prototype changed the database was the quickest one, only the accesses to tables were changed. No other things were designed or changed for the SQL style and that is the reason, why it is nicknamed as „Quick and Dirty" prototype. I think it wouldn't be clever to use this prototype as a source of information.

3.2 Subversion filesystem scheme

Generally a Subversion filesystem looks like an ordinary UNIX filesystem, so a node in Subversion could be either a file or a directory. The main difference from a UNIX node is that a node's content changes over time and we need to store all previous contents.

When we change the node's content or attributes (even a name of the node), it is still the same node, just with another content or attributes. So the node's identity isn't bound to a filename or content.

A *node revision* refers to a node's contents at a specific point in time. Changing a node's contents always creates a new revision of that node. Once created, a node revision's contents never change. As users make changes to the node over time, we create new revisions of that same node.

When a user commits a change that deletes a file from the filesystem, we don't delete the node, or any revision of it. Instead, we just remove the reference to the node from the directory.

3.3 Base data model of the Subversion filesystem

The purpose of this section is to describe the structure of Subversion filesystem backend using a higher level of abstraction. If anybody wants to get familiar with the Subversion backend, this section will give him the first basic information. The following scheme arises more from the BDB backend design, than from FSFS, but it tries to describe the general concept of the filesystem more than one specific implementation. It is much simpler in comparison with real structures, which are more complicated.

3.3.1 Node-revisions

If we speak about a node, we always need to know, which version we speak about, so we do not usually use a node ID, but a node revision ID instead. A *node revision ID* is composed from a *node ID*, *copy ID* and a *txn ID* („txn" is a shortcut for transaction, that is used in Subversion very often).

The *node ID* is unique to a particular node in the filesystem across all of revision history. That is, two node revisions who share revision history (e.g. because they are different revisions of the same node, or because one is a copy of the other) have the same *node ID*, whereas two node revisions who have no common revision history will not have the same *node ID*.

The *copy ID* identifies a given node revision, or one of its ancestors, resulted from a unique filesystem copy operation.

The *txn ID* is just an identifier that is unique to a single filesystem commit (it is not important if the commit succeeded or not). All *node revisions* created as part of a commit share this *txn ID*.

A *node revision* is either a file or a directory and this type cannot change over time. We store some necessary attributes by a node revision record:

- *created path* – the canonicalized absolute filesystem path at which this node revision was created
- *pred ID* – indicates the node revision which is the immediate predecessor of this node

These attributes above are the most important ones, for another attributes see [9].

3.3.2 Representations

The content of files or directories as well as file's or directory's properties are deltified (we usually don't store the whole content, but only deltas from the previous version, which saves much place on the disc). That data are stored separately from a *node revision* data and we call it representations, which are byte or text strings in *FULL* or *DELTA* format. More about representations in [9].

3.3.3 Transactions

Transactions represent exactly one commit operation and it could be in the following states: *transaction* (unfinished, active transaction), *dead* (an inactive transaction, which is not completed correctly) or *committed* (a correctly completed transaction).

A transaction contains following attributes:

- *root ID* – the node revision ID of the transaction's root directory
- *base ID* – the node revision ID of the root of the transaction's base revision (the base transaction is, of course, the transaction of the base revision)
- *transaction properties*
- *copies* – references to filesystem copies created inside of this transaction (if the transaction is aborted, these copies get removed)

3.3.4 Revisions

The revision is an integer number beginning by 0 (empty directory), whose value is the transaction that was committed to create this revision.

3.3.5 Changes

As modifications are made (files and directories added or removed, text and properties changed, etc.) on Subversion filesystem trees, the filesystem tracks basic changes made in changes records.

The main attributes of changes are:

- *path* – absolute path in the Subversion filesystem
- *node revision ID* – the node revision, that this change is related to

- *type* – this can be one of the following operations: add, delete, replace, modify
- *text-mode* – an indication, that the content of the node was modified
- *prop-mode* – an indication, that the properties of the node was modified

3.3.6 Copies

Every time a filesystem *copy* operation is performed, Subversion records meta-data about that copy.

A copy record has the following attributes:

- *type*: *copy* indicates an explicitly requested copy, and *soft-copy* indicates a node that was cloned internally as part of an explicitly requested copy of some parent directory (details in [9])
- *source path* – canonicalized absolute path of a source of the copy
- *source txn* – transaction ID of a source of the copy
- *destination node revision ID* – represents a new node revision created as a result of the copy

3.3.7 Locks

When a caller locks a file (reserving an exclusive right to modify or delete it), a lock object is created. It has the following attributes:

- *path* – an absolute filesystem path reserved by the lock
- *token* – an universally unique identifier of the lock
- *owner* – an authenticated username that *owns* the lock
- *comment* – a string describing the lock
- *XML-p* – a boolean (either 0 or 1) indicating whether the comment field is wrapped in an XML tag
- *creation-date* – date/time when the lock was created
- *expiration-date* – date/time when the lock will cease to be valid

3.4 Example of Subversion object diagram

This example originated to better understand, how a backend works inside. I also wanted to check, if I understood the backend right, so I posted this example to a Subversion mailing-list to demonstrate my thoughts to other contributors. Only one reaction without any serious suggestions came back, so I had been probably right.

The following example is a demonstration of the Subversion filesystem structure during two simple commit operations. First commit will add a directory *b* and files *a.txt* and */b/c.txt* to the root in an empty repository. The next commit will change the content of

rev_num	txn
1	1

Table 3.1: Revisions table after the first commit

txt	state	root
1	committed	0.0.0

Table 3.2: Transactions table after the first commit

node ID	type	path	pred ID	rep
0.0.1	directory	/		1
1.0.1	directory	/b		2
2.0.1	file	/a.txt		3
3.0.1	file	/b/c.txt		4

Table 3.3: Node revision table after the first commit

txt	path	type	node ID
1	/	change	0.0.1
1	/b	add	1.0.1
1	/a.txt	add	2.0.1
1	/b/c.txt	add	3.0.1

Table 3.4: Changes table after the first commit

rep	type	txn	content
1	full	1	(1.0.1, 2.0.1)
2	full	1	(3.0.1)
3	full	1	'abcdef'
4	full	1	'tuvwxy'

Table 3.5: Representations table after the first commit

rev_num	txn
2	2

Table 3.6: Revisions table after the second commit

txt	state	root
2	committed	0.0.1

Table 3.7: Transactions table after the second commit

the file *a.txt*, add a file *d.txt* and rename the directory *b* to *bb*. The first commit will create a revision 1; the second will create a revision 2.

The contents of the tables after the first commit are in tables 3.1, 3.2, 3.3, 3.4 and 3.5. After the second commit the new data will be as in the tables 3.6, 3.7, 3.8, 3.9 and 3.10. The whole scheme after the 2nd commit is sketched in the Figure 3.1.

3.4.1 Conclusion of the example, operations frequency

The Subversion filesystem is sort of a directed acyclic graph (DAC). Because we need to keep every change made in that filesystem, the most often modify operation is inserting (updating or deleting objects are only occasional operations).

We have generally two options how to store deltas. We can deltify the content against the youngest revisions, this way is used in BDB backend and offers faster checkout of the youngest revision and it is also the reason, why this option was chosen for the SQL backend.

Or we can use the second way and store the content as deltas against earlier revisions. This way is used in FSFS backend and offers better simplicity and robustness, as well as the flexibility to make commits work without write access to existing revisions.

While reading the Subversion filesystem, the most often operations are retrieving all ancestors of the node and retrieving all direct descendants (e.g. all nodes in the directory).

3.5 DAG structures in SQL databases

This section summaries and describes possible ways to store Directed acyclic graphs (DAG) in a relational database. It is only general review of possible algorithms and approaches, which does not follow Subversion backend structure. The possibility of real application of any of the following ways wasn't known before studying the backend's code and it seems inapplicable after that, because the backend interface would have to change too much. The following approaches could be possibly used within the bigger changes in Subversion design or in some another project.

Let's have a general DAG structure that we want to store in an SQL database and the following operations that we want to apply to that data:

node ID	type	path	pred ID	rep
0.0.2	directory	/	0.0.1	5
1.0.2	directory	/bb	1.0.1	6
2.0.2	file	/a.txt	2.0.1	7
4.0.2	file	/d.txt		8

Table 3.8: Node revision table after the second commit

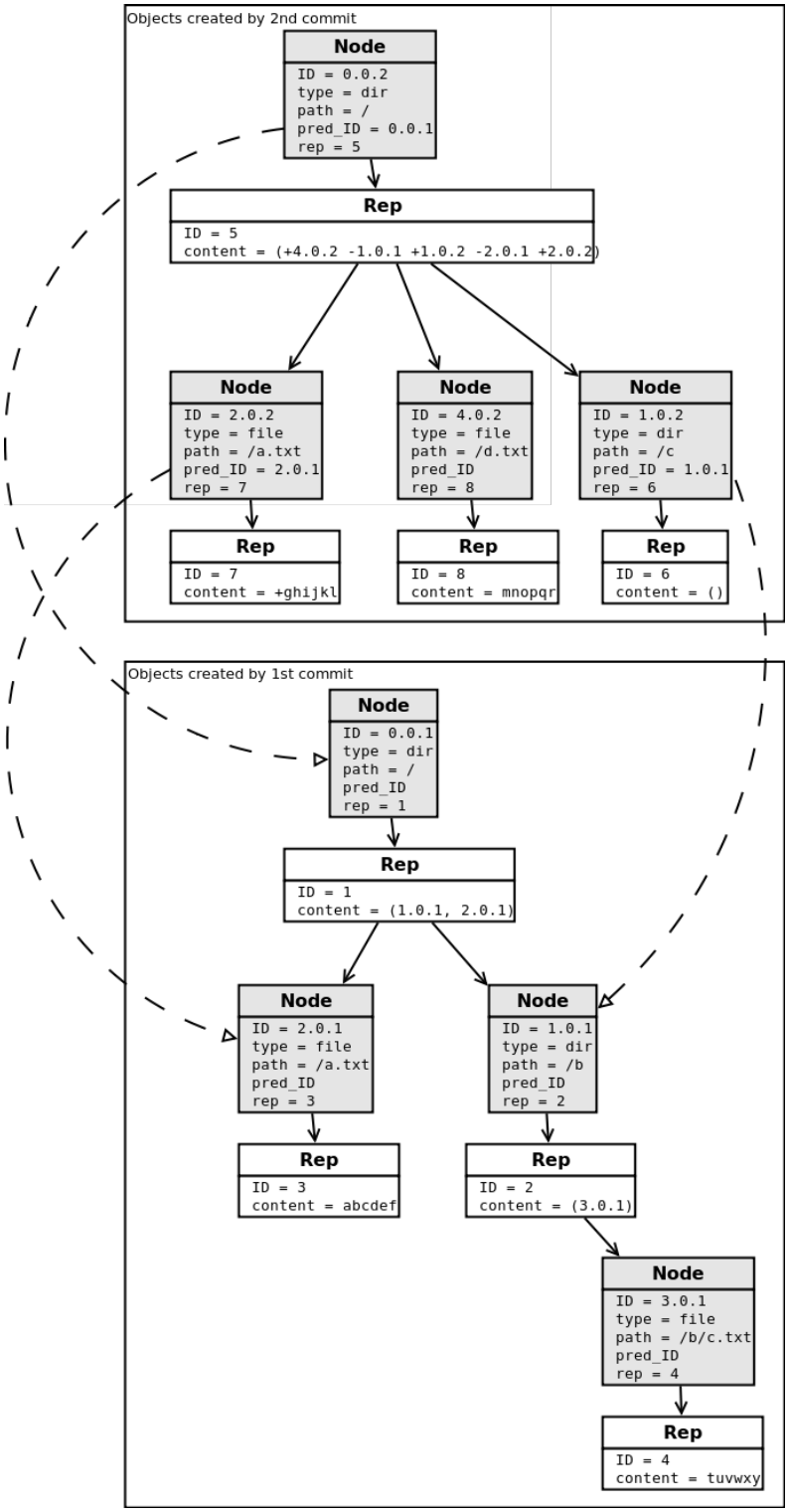


Figure 3.1: Subversion filesystem object diagram

txt	path	type	node ID
2	/	modify	0.0.2
2	/bb	rename	1.0.2
2	/a.txt	modify	2.0.2
2	/d.txt	add	4.0.2

Table 3.9: Changes table after the second commit

rep	type	txn	content
5	delta	2	(+4.0.2 -1.0.1 +1.0.2 -2.0.1 +2.0.2)
6	delta	2	()
7	delta	2	+'ghijkl'
8	full	2	'mnopqr'

Table 3.10: Representations table after the second commit

- insert a new node
- delete a node
- get direct ancestor
- get all ancestors
- get direct descendants
- get all descendants

Every request to the SQL database costs some time, so we will try to use all the operations above using as little queries as it is possible.

3.5.1 Read all to memory

We can reduce the number of database queries by loading the whole DAG structure to the memory and working with the data in the business logic. The way how to store edges between nodes could be adjacency list (see bellow), but this approach is applicable to only very small count of records, so it is definitely not applicable to Subversion filesystem.

3.5.2 Materialized path

In this approach each record stores the whole path to the root (using a character as a separator). An SQL table can look like as the table 3.11 or in Figure 3.2.

Let's look at some queries. We want to get an employee John and chain of his supervisors:

```
SELECT e1.*
FROM emp e1, emp e2
WHERE
  e2.path LIKE e1.path || '%'
  AND e2.id = 4
```

id	name	path
1	Peter	1
2	James	1.1
3	Jane	1.2
4	John	1.2.1
5	Charlie	1.2.2

Table 3.11: Materialized paths example

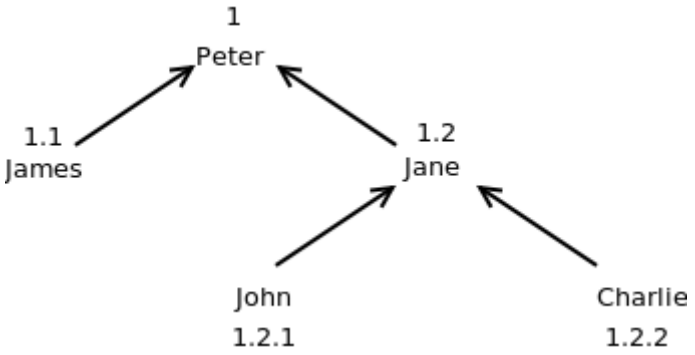


Figure 3.2: Materialized path example

Then we want to get an employee Peter and all his (indirect) subordinates:

```
SELECT e1.*
FROM emp e1, emp e2
WHERE
  e1.path LIKE e2.path || '%'
  AND e2.id = 1
```

We can see that reading one or more ancestors or descendants is very fast, but there are several problems. The adding and deleting nodes that are not leaves is very expensive, because we need to change paths of all its descendants.

Another issue is the path length, which can be very long in very complicated DAGs like a Subversion filesystem. We can reduce the size of the path using another path definition, which is describe in [32], but the disadvantage is still there.

3.5.3 Nested sets

This method is based on pre-calculated values *left* and *right* using the *preorder tree traversal algorithm*. More about this algorithm is in [16]. We will get table 3.12 for example and the structure will be like in Figure 3.3.

The advantage of this solution is easy reading the node's direct descendants as well as a whole subtree. Also reading of all leaves can be implemented using only one query. A row size is constant (only two single values are added) even if the DAG is very large. Getting the parent or all ancestors of the node is quick and it is possible to get the depth of the node in the hierarchy using `COUNT` and `GROUP BY` statements.

But there is one big disadvantage – adding or deleting nodes requires recount left and right values of many nodes in the tree, so it is not possible to use this solution in large DAGs. More information about this solution and about queries is in [16].

id	name	left	right
1	Peter	1	10
2	James	2	3
3	Jane	4	9
4	John	5	6
5	Charlie	7	8

Table 3.12: Nested sets example

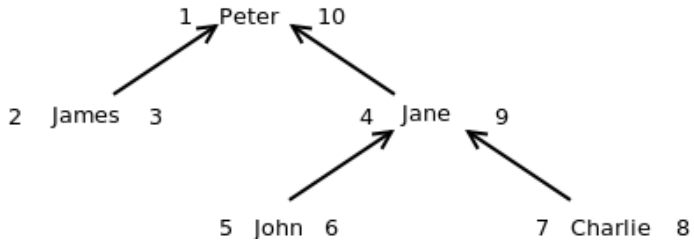


Figure 3.3: Nested sets example

3.5.4 String-Based Nested Sets

This solution, published in [48], describes using materialized path and nested sets together. It offers trigger definitions, so the nested sets are automatically maintained, but the performance is not better than ordinary nested sets and disadvantages of nested sets stay.

3.5.5 Nested intervals (Partial order)

Nested intervals are an alternative way, how to look at nested sets, but it doesn't give better performance when updating the DAG. More information about this solution in [15].

3.5.6 Adjacency nodes

The adjacency nodes solution has more implementations. The first way uses one nodes table with one special column, which references the same table and defines the parent of the node.

The other way is general definition of the edges. There are two tables – one-column table of nodes and a two-column table of edges. The edges table can be thought of as a nodes-nodes bridge table, each row containing pointers to origin and destination nodes. Other details of nodes and edges can be encoded in extra nodes and edges columns, or in child tables.

For our example the table of nodes will look like table 3.13.

The table of edges will look like table 3.14. The structure will look like as in the Figure 3.4.

If we want to get the whole subtree, we have two general options. First option is to use several joins, but we have to know the depth of the query before we build it. This is definitely not universal, so we won't use it.

The other option is to use a recursion, which means either many requests to the database or an SQL procedure with a recursion or an equivalent loop. Then we can use one of basic traversal algorithms for a tree. There are several ways how to read subtrees using MySQL procedures described in [19].

id	name
1	Peter
2	James
3	Jane
4	John
5	Charlie

Table 3.13: Adjacency nodes table example

child	parent
2	1
3	1
4	3
5	3

Table 3.14: Adjacency nodes (edges) table example

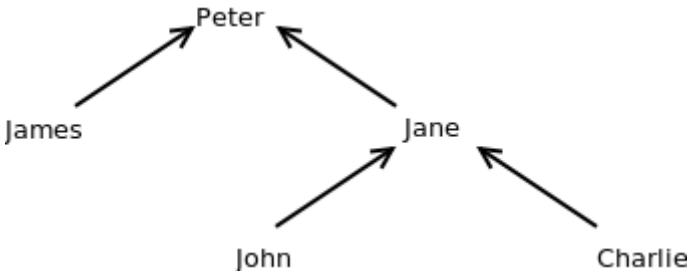


Figure 3.4: Adjacency nodes example

id	name
1	Peter
2	James
3	Jane
4	John
5	Charlie

Table 3.15: Adjacency nodes with transitive closure example

child	parent	hops
2	1	0
3	1	0
4	3	0
5	3	0
4	1	1
5	1	1

Table 3.16: Adjacency nodes with transitive closure (edges) example

The biggest advantage of this method is simple adding or deleting nodes, which is done in constant time.

We should mention another feature of adjacency nodes, even if it is not the case of the Subversion filesystem. This solution is the only one from the methods above that is available to use to represent non tree graphs.

3.5.7 Adjacency nodes with transitive closure

We have mentioned, that getting a subtree using standard adjacency list is a difficult task and we have to use recursion. However, there is another solution which requires some redundancy in the database. The purpose of that redundant information is to move recursion from the operation while selecting a subtree to tree structure itself. That means we will use a recursion just while inserting a node and we won't use it while we will retrieve the subtree.

The purpose of this method is to create a *transitive closure* of the DAG. Besides the direct edges we will store indirect edges too and we will add one extra column (hops) to the edges table, which will count number of nodes in a indirect edge. For our example the data can look like in tables 3.15 and 3.16.

This approach has many advantages. First, retrieving any subtree will be done by only one query. Addition and deletion are not so complicated as in nested sets and consumes from $O(log(N))$ up to $O(N)$ time.

On the other hand, there is one significant disadvantage too, that is number of indirect edges stored in the database. This number depends on the DAG character very much, but it can consume even $O(N^2)$ space. We can find more about this approach in [2].

A bit similar solution is used in Oracle and SQL Server implementations that support some tree operations.

3.5.8 MySQL, MSSQL, PostgreSQL and Oracle solutions

MySQL doesn't offer any DAG oriented utilities, all we can use are views, procedures or functions.

In PostgreSQL we can use a module *ltree*, which implements the Materialized path solution. More about this module in [2].

Oracle offers a `START WITH` and `CONNECT BY` statements, which are used to define a connection using the adjacency nodes. This operator can be used only for graphs that are trees. More about that operators in [2].

Microsoft SQL Server includes a `WITH` operator as a part of so-called Common Table Expressions. It can be used for processing recursive sets too. We can find a similar operator in DB2 from IBM too.

3.5.9 Conclusion of the implementation the DAG structures in SQL

Out of all the ways to store a tree in a RDMS the most common are adjacency lists and nested sets. Nested sets are optimized for reads and can retrieve an entire tree in a single query. Adjacency lists are optimized for writes, reads are generally a simple query with little of data [2].

3.6 Backend operations analysis

The purpose of this analysis was to find out, which operations in which tables the BDB backend accessed most often. We can use this in the design of the database scheme, because we need to know what data we will store and what operations we will apply to that data. The data structure is described in the previous text, but the operations haven't been clear before this analysis.

3.6.1 Table access

We need to consider what tables we work with most often and what kind of work it is. In the following tables 3.17 and 3.18 there is a result of the two most important operations with repository – commit and checkout. The test tried to simulate an ordinary work with the repository.

We can see that most of accesses during commit operation, as well as during checkout operation, were done to tables *nodes*, *representations* and *transactions*. Other tables were not accessed so frequently, so we don't need to focus on optimizing those tables. We need to design a database scheme that will ensure fast read and write operations to tables *nodes*, *representations* and *transactions*.

3.6.2 Repeating operations

The second part of this analysis was aimed to discover which operations were repeated and which were used separately. The repeated operation could be possibly optimized (for example for reading by using the only one query). The analysis shows that only representations and nodes tables were accessed multiple times within one major operation.

We have already said that the Subversion filesystem is a kind of DAG, but the nodes of the DAG are generally from various tables. If we take a look at the tables separately, the tables are generally not the DAG anymore, apart from the *nodes* and *representations*

table name	access type	number of access
copies	get	3
checksum_reps	get	25
node_origins	get	30
uuids	get	42
lock_tokens	get	76
miscellaneous	get	78
revisions	get	289
transactions	get	924
representations	get	1402
nodes	get	2424
copies	put	2
checksum_reps	put	12
revisions	put	29
node_origins	put	30
changes	put	45
strings	put	203
nodes	put	221
representations	put	265
transactions	put	343
sum of all tables	both	6443

Table 3.17: Table access during a commit operation

table name	access type	number of access
uuids	get	2
revisions	get	65
transactions	get	125
nodes	get	326
representations	get	338
sum of all tables	both	856

Table 3.18: Table access during a checkout operation

tables. We can say that these two tables are a kind of DAG, so we can use one of the possible ways of storing the DAG in SQL in the future, but it will cost bigger changes in Subversion design. It is very important that there is no need to read descendants of nodes, only to read ascendants (we can imagine we read the history of the element) in both tables.

3.6.3 Discussing possible storing mechanisms

We could not use the *Nested sets* solution, because the insertion of the new nodes would cost too much. Also *Adjacency lists with transitive closure* are not suitable for this purpose, because the memory requirements would be too large.

If we consider *Materialized paths*, we need to think about how many ancestors a node could have. If a Subversion project has hundreds of thousands of revisions, we could expect that a node or a representation item could have thousands of ancestors. So the materialized path length could reach hundreds of kilobytes per item even if the item itself is very small, which is not acceptable.

Thus, the only solution to store a DAG in Subversion seems to be the *Adjacency lists without transitive closure*. It always consumes a constant memory space and the inserting and updating is very fast. The disadvantage of this solution is that it is not possible to implement the reading of all ancestors by using one query. It is possible to use procedures and functions in DML in the future, in order to delegate some primitive DAG operations from application to database logic.

3.7 Comparing numeric and character indexes

In BDB and FSFS backends there are keys in 36-base format, which are as a matter of fact strings. In MySQL it is possible to use various data types as primary keys of the tables, but speed of access using various types is different. This section should answer the question, how much faster is to use integer indexes to access data entries in a relational database, than to use 36-base character indexes.

The fastest access to data offer integer indexes, so we always should try to find numeric column to use as primary key. This is valid generally for indexes, but the most important indexes are primary keys. Some operations (especially aggregate functions) could be several orders of magnitude faster using numeric index, than using character index. More about indexes in [14].

On the other hand there is existing interface in Repository Access Layer, that use character 36-base indexes. These indexes are not changed in the existing backend layer and it would be necessary to redesign too large part of Subversion, if we want to use numeric indexes. The question is, if the numeric indexes are so much faster, that it will be worth enough to implement backend using them.

3.7.1 Simple numeric and character indexes compare test

Because of deciding if we need to use numeric indexes, a simple test was implemented. We had two tables with two columns – *primary key ID* and *DATA* column of the type `VARCHAR(255)`. In the first table a primary key was of type `CHAR(10)`, in the second table a primary key was of type `INTEGER`. Both tables were filled with 320,000 random records, but there were the same data in both tables.

	Numeric index	Character index
First query before optimize	0.230	10.700
Second query before optimize	0.010	12.400
First query after optimize	0.209	1.606
Second query after optimize	0.022	1.710

Table 3.19: Numeric and character indexes compare [s]

The first query was composed from one `SELECT` with `BETWEEN` operator and a sub-query with the second `SELECT`, also with `BETWEEN` operator. There were about 10,000 result rows for this query and the query looks as follows:

```
SELECT `key`
FROM `numeric_index_table`
WHERE
  `key` BETWEEN X1 AND Y1
  AND `key` IN (
    SELECT `key`
    FROM `numeric_index_table`
    WHERE `key` BETWEEN X2 AND Y2 )
```

The second query was very simple with only one result row and it looks as follows:

```
SELECT *
FROM `numeric_index_table`
WHERE
  `key` = X;
```

Both queries were tested twice – after inserting records and after optimalization of the tables.

How long the queries lasted is shown in table 3.19. Tests were run several times and there are average values in the table. The time is in seconds.

Note: Tested on a computer with Intel Core2Duo T5550, 1.83GHz, 2GB RAM, HDD Hitachi HTS542525K9SA00, 5400rpm, Ubuntu 9.10, Linux Kernel 2.6.31-20, MySQL 5.1.

We see, that numeric indexes are much faster then character indexes. Another interesting thing is, that using numeric indexes the less results we get, the faster the query is. But using character indexes the number of results doesn't involve the speed at all.

3.8 Database and Filesystem Access Speed Comparison

The reason of this test was to compare access speed of reading data from pure filesystem and from MySQL database running on the localhost. It was supposed that reading data from database will be slower than reading from filesystem, but the question is how much. The test was run twice with different record size.

There were same random data in binary or text form in the filesystem as well as in the MySQL database. The data were accessed by their 128bit-length md5 hash. There were totally 100,000 records of small files with average size of 3,7kB, which means totally 366MB of data and 11MB for index in database. In a test with large files there were 780MB of data, 28kB for index in 200 records.

	FS #1	MySQL #1	Comp. #1	FS #2	MySQL #2	Comp. #2
Test #1	11.29	4.81	43%	0.013	0.814	6091%
Test #2	12.84	4.74	37%	0.017	0.688	4138%
Test #3	10.69	4.56	43%	0.023	0.638	2766%
Test #4	10.67	4.32	40%	0.017	0.707	4047%
Test #5	9.92	3.98	40%	0.023	0.616	2677%
Test #6	9.86	4.07	41%	0.023	0.620	2745%
Test #7	12.37	3.78	31%	0.016	0.548	3326%
Test #8	10.19	3.25	32%	0.014	0.377	2791%
Test #9	9.43	3.50	37%	0.013	0.383	2854%
Test #10	9.34	3.00	32%	0.013	0.299	2250%

Table 3.20: Numeric and character indexes compare (small files) [s]

	FS #1	MySQL #1	Comp. #1	FS #2	MySQL #2	Comp. #2
Test #1	1.81	11.52	637%	1.07	7.29	682%
Test #2	2.45	9.71	397%	0.98	6.78	689%
Test #3	2.15	10.27	478%	1.12	7.7	687%
Test #4	1.65	7.92	480%	1.04	7.17	690%
Test #5	2.19	7.63	348%	1.1	7.63	690%
Test #6	1.55	8.38	542%	1.21	8.32	689%
Test #7	1.14	7.31	642%	1.07	7.37	691%
Test #8	1.15	8.33	723%	1.23	8.42	686%
Test #9	1.08	7.52	699%	1.07	7.39	694%
Test #10	1.04	7.39	712%	1.02	7.24	707%

Table 3.21: Numeric and character indexes compare (large files) [s]

In the filesystem files had to be placed in separate directories, because more than 1,000 files in one directory slows down the directory searching. An example of the stored files can be as follows:

- files/df/dfdbf90903852bd6bbd66c532fceb2b5
- files/c9/c9d7b037dc8dcd52fb7a972e33a0a1cf
- files/73/73f65d20dca6fc6063c6974757cab145

In the MySQL database contents of files were stored in one InnoDB table with integer primary column *ID*, column *HASH* of the type `VARCHAR(32)` for *MD5* hash of the content and column *CONTENT* of the type `LONGBLOB` for the content itself. Searching was performed using one `UNIQUE INDEX` on the *HASH* column.

Each test composed from 10 partial tests. 1,000 random records were loaded in every partial test in the case of small files and 300 records in the case of large files. Pseudo-random generator had been initialized by the same seed, so the results from MySQL and from filesystem were the same. Each test was run twice for small files and twice for large files, because repeated reading could be influenced by using a cache.

Results of the test with small files are in the table 3.20 and with large files in the table 3.21. The time is in seconds.

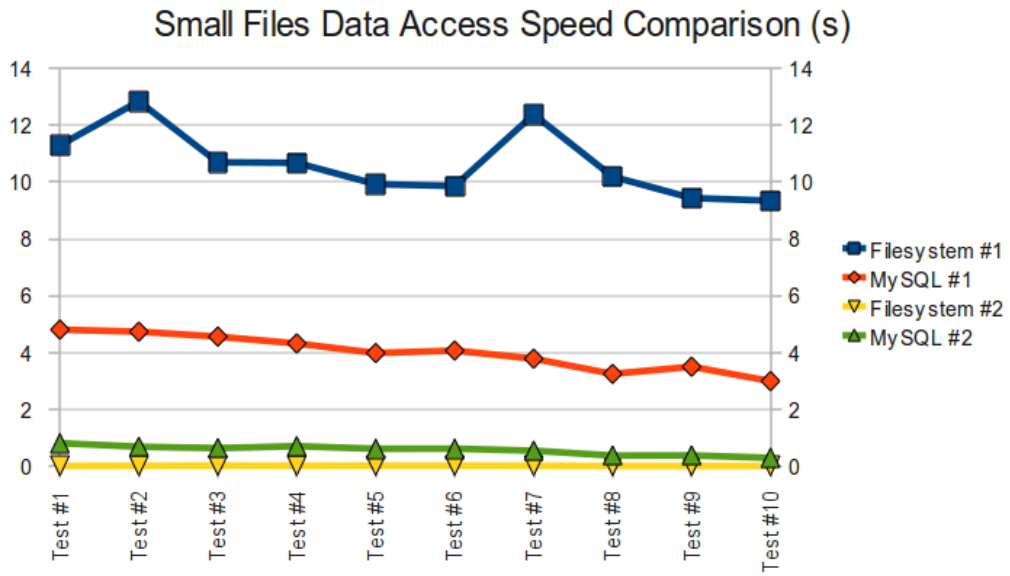


Figure 3.5: Results of access speed test with small files

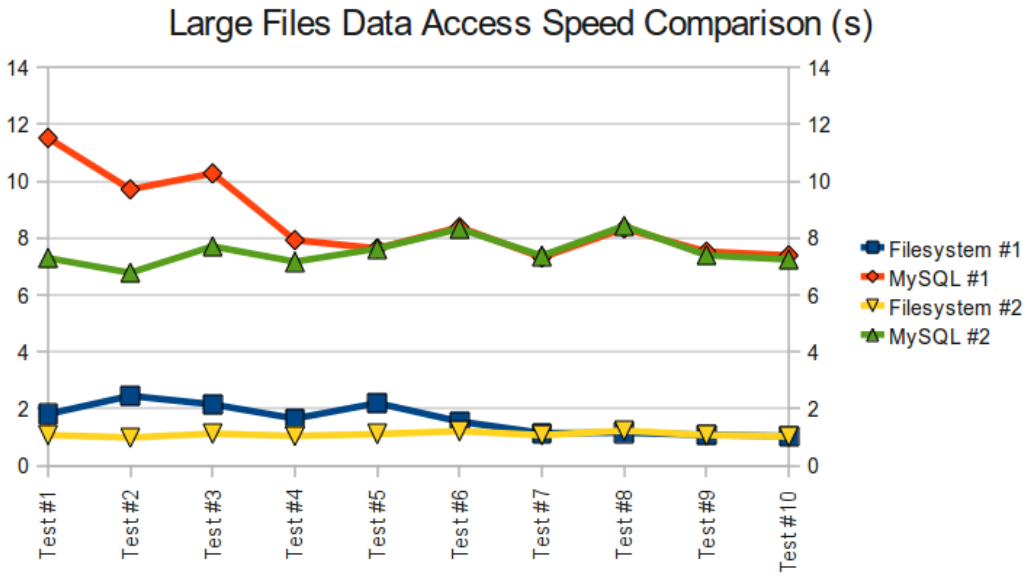


Figure 3.6: Results of access speed test with large files

Note: Tested on a computer with Intel Core2Duo T5550, 1.83GHz, 2GB RAM, HDD Hitachi HTS542525K9SA00, 5400rpm, Ubuntu 9.10, Linux Kernel 2.6.31-20, MySQL 5.1.

We can see that reading data from MySQL is surprisingly not always slower than reading data from pure filesystem. Especially in the case when small files are read first time, MySQL database is much faster than a pure filesystem. This is probably caused by caching more rows during the first query, whereas the filesystem caches each file separately.

If we want to improve the reading from pure filesystem, we could store small records in larger files so the filesystem cache could be used better. The real speed also depends on the character of the work with data, it depends on repetition of reading of the same or close data and it also depends on the size of the data.

I discussed with other Subversion developers, whether the storing data in filesystem will be faster than in database. Greg Stein (one of the most involved developers) had a very good point that I agree with:

„My gut says "not that much faster„. In most scenarios, the network bandwidth between the client/server will be the bottleneck. Reading the data off a disk (rather than from a DB) is not going to make the WAN connection any faster. On a LAN, you might have enough network bandwidth to see bottlenecks on the server's I/O channel, but really... I remain somewhat doubtful. I'd go with the "store content in the database„ until performance figures (or a DBA) demonstrates it is a problem."

Philipp Marek responded with another approach: *„How about allowing a mixed design (later, optional, when the backend is running)? Ie keep blocks smaller than N in the database, but write larger ones to the filesystem? Or provide different paths depending on the block size? Then people with SSDs could use them for the small blocks (or just keep them in the database, as before), but larger data entities could be read from disk directly. That would probably make some sense, as small blocks don't matter if they're traveling across a few pipes; but for multi-MB data blocks that should be avoided."*

This issue will probably stay opened some time, at least until any SQL prototype will be able to test and compare all approaches. Until that the content data will stay in the database. The whole thread in Subversion mailing-list can be found in [37].

3.9 Directory content storing

This section considers a problem with too many files in one directory and possible ways to avoid the speed access problems. Storing many files in one directory brings some performance issues, because files are stored in pure sequence generally. The problem comes up with about a thousand of files in one directory. Reading or writing to such directory is much slower than to smaller directory. This is general problem of many present filesystems and the same problem has arisen in Subversion filesystem too. In BDB as well as in FSFS files are stored sequentially, thus the bigger a directory is, the slower reading or writing to that directory is.

In Subversion there is another problem with big directories – very big space requirements for changes. For example, if we have a directory with 10,000 files and we delete one file, we have to store the whole directory content in the new revision. Subversion does not use the delta design for the directory content, because in most cases it doesn't bring any performance advantages, rather the opposite.

There is a user's experience in the Subversion developers mail-list [30]. Paul Holden is describing tests with *svn update* operation on two directories. The first directory contained 10,000 files in 390MB well distributed in subdirectories; the second directory contains 5,000

files in 22MB without subdirectories. Update of the first directory lasted 4 minutes, update of the second directory lasted 10 minutes. This paradox is caused by not efficient working with directory entries. We can say that disadvantages of this issue are seldom, but they can be very paint-full.

The problem with large directories, described in the previous text, was the impulse for another discussion between Subversion contributors, including me. Philipp Marek advised to design storing diretories' contents in another way in the prepared SQL backend. The purpose of another storing algorithm is to make the storing overhead (when directory entries are changed) as little as it is possible.

It has been mentioned before; delta design is not good choice, because of unnecessary overhead in smaller directories, where no special algorithm is needed, so the delta design would probably slow down the system. Philipp Marek advised two another approaches: *„Either use a new table, with fields like parent, name (or path), valid-from-revision, valid-before-revision or something like that. Then changing an entry means only updating valid-before of the old record, and inserting a new one. Or, if you want to store directories in the same way as file data (like now in FSFS and BDB), I'd suggest to limit such blocks of directory data to a few KB, but to define an indirect-block that tells which blocks are used. A new entry could then reference all the unchanged blocks of the older revision“* .

Both of the approaches from Philipp Marek are available to design in SQL backend, which of them would be faster and if they will be better than present approach is a question that can be answered first after testing in real environment. After Phillip advised his idea, Greg Stein replied with another solution: *„Go ahead now and store megabytes for each directory, just like the other backends. And leave the solution of this problem to a future iteration of the SQL-based backend. Really... optimizing before you even get started is not advisable. Get something done. Then examine and iterate. There could be numerous other problems inherent in a SQL backend that would obviate any such solution proposed today“*.

Greg's idea is certainly right so I decided to leave this problem for the next iterations of development, when real benefits of any other approaches could be tested. The whole thread in mail-list is available in [36].

3.10 Primary key of the transaction properties table

One of the discussions between me and other Subversion contributors was about storing transactions properties. In BSB transaction properties are stored as a skel (or a list in another words) with name and value attributes, but it is not possible to store lists in an SQL database as a single value. A new table with foreign key *transaction_id* was connected to the rest of the SQL design, other columns where *name* and *value* (both strings).

It was a question, what should be the primary key of that table. In the first design there were special integer primary key and another unique key on two columns – *transaction_id* and *name*. Greg Stein suggested using that unique key as primary, because the former primary key was not referenced at all. Thus, in the next design the transaction properties table uses only one key (composed from *transaction_id* and *name*).

The whole thread in Subversion mailing-list is in [44].

3.11 Another suggested changes in SQL backend design

Some contributors or users suggested to make another changes in SQL design, that are not necessary to solve right now, but in the future, in next iterations of the SQL backend development. Martin Furter for example suggested *„to add a field repository_name to the transactions table, it would be possible to store multiple repositories in one database/schema.“*

From my sight, it would be nice to store more repositories in one database, but I find this way a bit confusing and without significant advantages (e.g. it would be not clear which rows belong to which repository). But, some table prefix could be used by all tables, it would be transparent and easy to destroy the whole repository for example (without svn routine).

The same view has Bob Archer, who wrote *„Please don't do this (adding the field repository_name) unless it is an option. Each repository should be self contained. It would also be nice if it were OS agnostic... so for example I could move a repository from one server to another (Windows to Mac) by just copying over a SQLite database. Even all the config info should be in the database rather than magic config files.“*

This thread in Subversion mailing-list can be found in [29].

3.12 Optimizing MySQL operations

This section offers some basic information how to optimize relational databases generally. We should always want to get the database design to the normalized form, so we cannot store lists in one table column for example, as it could be done in BDB database. We can also improve the database speed significantly using some basic optimization operations, some of the most important are shortly described in the following subsections.

3.12.1 MySQL engines

MySQL database offers various storing engines – MyISAM, InnoDB, Merge, Memory, Archive, CSV, Federated, NDB Cluster, ISAM, BDB, Example and MaxDB. Many of them are intended for special purposes and we will consider only two of them, which can be used for any general purpose – MyISAM and InnoDB. More information about other engines can be found in [14].

Both engines are well optimized and have a lot in common. We will discuss only differences. InnoDB have better support for parallel work of many users, offers possibility of locking rows as well as tables and includes support for transactions. It is better for storing huge amount of data, even bigger than a file size limit of the used filesystem is. But on the other side the data needs more space on disc.

Engine MyISAM doesn't use transactions, but it is a bit faster than InnoDB and offers full-text search.

We can probably use the third engine Memory, but only for some temporary data, not as a main storage engine. Data of this engine are stored in HEAP memory and can be stored only during run of the MySQL server. More about engines in [14].

3.12.2 Optimizing tables

MySQL offers many options how to optimize stored data or access performance to that data. Many of the options have its advantages and disadvantages at the same time and it depends on the particular situation which option is better.

The `ROW_FORMAT` parameter defines if a table will be dynamic or static. Dynamic tables are smaller but working with them is slower. Fixed tables are faster and their size could be optimized, but the space on the disk will be always bigger then by dynamic tables.

We can specify supposed and maximal number of rows in the table, this can be used for better optimizing. MySQL offers a special procedure *analyze* , which can be used for optimizing table schema. It is very useful to use the *OPTIMIZE TABLE* query, from time to time, which is important especially at often changed tables. More about optimizing in [14].

3.12.3 Storing the lists in SQL

Storing the whole list of more values in one column isn't possible in relational database, we have to always use another table and 1:many relations. The access speed to the joined columns depends on the indexes very much. Properly used indexes are more important in the tables with more rows and the the robustness of the index (number of different values in all rows for the specified column) should be as big as it is possible. We can also use composed indexes, if the single index has small robustness. More about storing lists in database and optimizing of joining tables in [14].

3.12.4 Using of prepared statement and multiple-lines inserts

MySQL offers prepared queries in the same way, as in other database engines. The query is prepared once and the prepared statement could be called with various parameters. We can save some processor time, because queries don't have to be analyzed every-time.

Another speed improvement is multiple-line inserts, that can be surprisingly several times faster, then the same number of simple inserts. This feature should be definitely used if a bigger number of entries will be inserted in a sequence, for example as a part of import action.

3.12.5 General acceleration of the MySQL database

There are many optimization methods how to improve the speed of MySQL database. We can for example store indexes in another server than data, to have very fast access to indexes even if the data server is busy. We can use more servers, clusters, regulate memory limits, cache sizes, page sizes or using secondary indexes to optimize database for our special purpose. It always depends on database size and operation types and we should always use a real data to test any improvements. We can find many of mentioned solutions and some others in [14].

3.13 Database scheme of the MySQL backend

This section describes the MySQL backend structure and its differences compared to BDB backend. This scheme is inspired by the BDB structure, that is accessible in [9]. The whole backend is visualized in Figure 3.7.

BDB and MySQL are both databases, but they differ quite a lot. BDB is embedded database, which stores key-value pairs, while MySQL is general purpose relation database. Both of the databases has advanced index handling, transaction support (if we use InnoDB in MySQL) and good support on many operating systems.

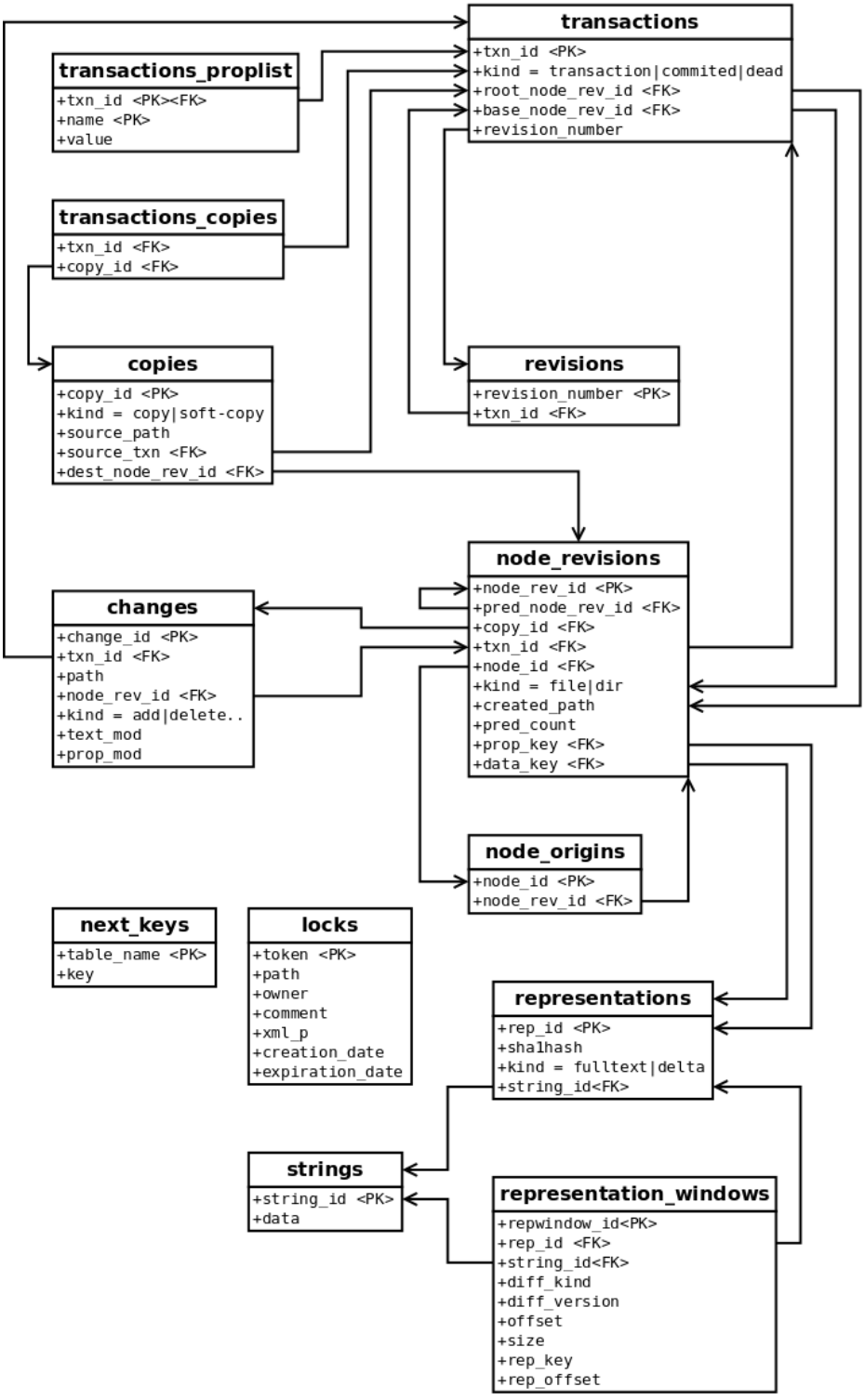


Figure 3.7: Database scheme of the MySQL backend design

The main differences between MySQL and BDB are the list, that are stored in the database entry just like other values in BDB, but a special table and the 1:many relation has to be used in MySQL. Another significant difference are indexes, that are 36-base in BDB, while there are used integer values in MySQL database. This is not necessary because of MySQL abilities, but it is much faster to use integer values, than 36-base strings (test is described in section 3.7).

3.13.1 Nodes

Nodes are abstract structures, which we can imagine like files or directories in an ordinary filesystem. So a node has its type (file or directory), content and properties. The type of the node never changes, a file will always be a file and a directory will always be a directory.

Properties and content of nodes changes in time and every change is stored in the backend. Set of changes of one or many nodes (files or directories) are done in transactions (shortly txn, we will talk about them later). Every successful transaction will make a new snapshot of the whole filesystem, and we mark the snapshot with integer value started from 0, called revision number; the snapshot is called revision.

3.13.2 Revisions

The revisions table have two columns – *revision_number* (primary key) and *txn_id*, so it is a kind of map of *revision_number* to *txn_id*. For example an item (5, 23) says that revision 5 has been made using the transaction with *txn_id* 23. That's all we can read from that table, nothing more.

The convention says, that revision 0 is an empty directory with *node_rev_id* is 0.0.0 (see bellow).

Structure of the Revisions table

- *revision_number*: `int(20)` – Number of the revision (snapshot)
- *txn_id*: `int(20)` – Transaction ID which this revision was made from

3.13.3 Node-Revisions

As we have said above, there is not a nodes table in the database, but there is a node-revisions table there. This table stores the snapshots of the nodes in time. If a node's content or properties change, a new node-revision is inserted in this table. No items are deleted, always just added. If we want to get all the actual files tree, we need to read all last node-revision entries from the table node-revisions.

Because of distinction which nodes are snapshots of the same node, there is a *node_id* column in the table. There is also a *copy_id* column, which indicate, which node-revision entries has the same history (e.g. one is a copy of another), more about copies in the text bellow.

Besides *node_id* and *copy_id* there is *txn_id*, which specifies, what snapshot the node-revision entry belongs to. So the node-revision is well defined by three values – *node_id*, *copy_id* and *txn_id*. These values are the primary key of the node-revisions table.

It is problem to reference to composed primary key from other tables using a foreign key, so a new column *node_revision_id* was added. This column correspond with the node-

revs-id in BDB design, so it is concatenation of the values *node_id* '.' *copy_id* '.' *txn_id* (primary key units, separated with '.').

Besides attributes above we store the *created_path*, which is the path, where the node has been created in. Then there are a *prop_key* and a *data_key*, that are foreign keys to the table representations, represented the properties and the content of the node respectively.

The attribute *pred_node_rev_id* is the reference to the same table which specifies the predecessor of this node. The *pred_count* attribute specifies the count of predecessors of the node_revisions.

A directory entry identifies the file or subdirectory it refers to using a node revision ID – not a node ID. This means that a change to a file far down in a directory hierarchy requires the parent directory of the changed node to be updated, to hold the new node revision ID. Now, since that parent directory has changed, its parent needs to be updated, and so on to the root. We call this process „bubble-up“.

If a particular subtree was unaffected by a given commit, the node revision ID that appears in its parent will be unchanged. When doing an update, we can notice this, and ignore that entire subtree. This makes it efficient to find localized changes in large trees.

Structure of the Node-Revisions table

- *node_rev_id*: `varchar(64)` – string representation of the primary key in the form *node_id '.' copy_id '.' txn_id*
- *pred_node_rev_id*: `varchar(64)` – reference to predecessor
- *copy_id*: `int(20)` – reference to the Copy table
- *txn_id*: `int(20)` – reference to the Transactions table
- *node_id*: `int(20)` – ID of the node
- *kind*: `int(4)` – kind of the node (either file or directory), value depends on the C-enum value
- *created_path*: `varchar(1024)` – path under which the node has been created
- *pred_count*: `int(20)` – count of all predecessors of the node-revision
- *prop_key*: `int(20)` – reference to representations table (node-revisions' properties entry)
- *data_key*: `int(20)` – reference to representations table (node-revisions' data entry)

3.13.4 Representations

This is the place, where content and properties of files and directories are stored. Content stored here differs files from directories. In directories, the content of the directory is stored always as a fulltext. No deltas (more about deltas bellow) are applied to that data, but there are possible ways to change this approach in the future, because there are some performance issues when a directory contains a lot files (see section 3.9).

The variable-length data of files is often similar from one revision to the next, so Subversion stores just the deltas between them, instead of successive fulltexts. The newest version is stored in fulltext and the older versions are stored as a difference from the previous. In

FSFS the older version is stored in fulltext, which brings greater simplicity and robustness, as well as the flexibility to make commits work without write access to existing revisions.

In SQL backend the BDB approach is followed, which brings fast checkouts of the last revisions.

Let's take a look at referencing in the representations table. There is an attribute *sha1hash*, that is (as supposed) an SHA1 hash of the entry's data. It's possible then to split blocks on manber-borders, to save space, because we don't need to store the same data again and again. We just know, we have that data already in database, so we add only a reference to that data and we're done.

Structure of the Representations table

- *rep_key*: `int(20)` – primary key
- *sha1hash*: `varchar(40)` – SHA1 hash of the data
- *kind*: `int(4)` – fulltext or delta
- *string_id*: `int(20)` – foreign key to strings table, where real data are stored (used only in fulltext mode)

3.13.5 Representation windows

This table is used to store metadata of deltified files contents. Column *rep_key* says what the window should be applied against, or none if this is a self-compressed delta. Column *size* says how much data this window reconstructs, column *version* says what version of the svndiff format is being used (currently only version 0 is supported).

Column *string_id* says which string contains the actual svndiff data (there is no diff data held directly in the representations table, of course).

Note also that *rep_key* might refer to a representation that itself requires undeltification. We use a delta combiner to combine all the deltas needed to reproduce the fulltext from some stored plaintext.

Structure of the Representations table

- *repwindow_key*: `int(20)` – primary key
- *rep_key*: `int(20)` – foreign key to representation table
- *string_id*: `int(20)` – foreign key to strings table, where real data of this window are stored
- *diff_kind*: `int(4)` – kind of diff
- *diff_version*: `int(4)` – version of the diff used
- *size*: `int(20)` – size of the data stored in this window
- *offset*: `int(20)` – where the data start
- *rep_key*: `int(20)` – chunk-specific data used in data combiner
- *rep_offset*: `int(20)` – chunk-specific data prepared for possible future use, not used at present (see more in [9])

3.13.6 Strings

This table is used to store the binary data of files or directories itself.

Structure of the Representations table

- *string_id*: `int(20)` – primary key
- *data*: `blob` – binary data of various size

3.13.7 Next-keys

This table is used for generating various integer sequences, e.g new primary keys in all tables. We need to handle new keys more complexly, for example we often want to allocate a new value only, without inserting a row to the database, so it is not able to use auto-increment columns for that purpose.

The table stores the first not-used value always, so we can read a value without updating, increment the value without inserting any rows, change the next value, etc.

Structure of the table:

- *table_name*: `varchar(64)` – name of the table, generally identificator of a sequence
- *key*: `int(20)` – first unused value in a sequence

3.13.8 Transactions

Every change made in the backend is done as a part of a transaction. The transaction can be *in progress* (unfinished), *dead* (aborted) or *committed*; committed or dead transaction cannot change any more. Transactions in the backend are a bit different from the transactions in the database.

Backend transactions often last several minutes, while the database transactions seldom last a second. But the purpose of both of the transactions is the same – to run a group of changes as all of them were one atomic operation, that is either successful or it happened not at all. However, we definitely cannot use the database transactions to handle the transactions in the backend.

Transactions are always related to the specified subdirectory in the Subversion filesystem, which can be the whole repository of course. After the transaction is committed, the revision number made by this transaction is set to the transactions property.

Structure of the Transactions table

- *txn_id*: `int(20)` – primary key, ID of the txn
- *kind*: `int(4)` – status of the txn (committed, aborted, unfinished
- *root_node_rev_id*: `varchar(64)` – root of the transaction
- *base_node_rev_id*: `varchar(64)` – the node revision ID of the root of the transaction's base revision. This is of the form 0.0.BASE-TXN-ID – the base transaction is, of course, the transaction of the base revision
- *revision_number*: `int(20)` – revision number made using this txn

3.13.9 Changes

All changes made during a transaction are stored in the changes table. This table does not influence the content of the repository, it describes only changes in every transaction. These changes can be then used in the future.

Path and kind of the change (addition, modification, deletion, replace) is stored for every changed node, which is referenced by *node_revision_id*. Properties *text_mode* and *prop_mode* are boolean values that indicate what of the node has been changed, if the content or properties or both.

Structure of the Changes table

- *txn_id*: `int(20)` – reference to the txn table
- *path*: `varchar(1024)` – path where the change occurred
- *node_rev_id*: `varchar(64)` – reference to the node-revisions table; which node-revision has been changed
- *kind*: `int(4)` – type of the change (addition, modification, deletition, replace), value depends on C-enum value
- *text_mod*: `int(4)` – bit specifying that the data of the node-revision has been changed
- *prop_mod*: `int(4)` – bit specifying that properties of the node-revision has been changed

3.13.10 Copies

If we copy a node, both new nodes share the history and we need to store this fact, so we do it in copies table. A copy is defined by a *source_path*, a *source_txn* and a *destination_node*. A copy can be either soft or normal. Soft copy indicates, that a node that was cloned internally as part of an explicitly requested copy of some parent directory. Normal copy is initialized by user using a copy command.

Structure of the Copies table

- *copy_id*: `int(20)` – primary key, ID of the copy
- *kind*: `int(4)` – kind of the copy (copy, soft-copy)
- *source_path*: `varchar(1024)` – path of the source node-revision
- *source_txn*: `int(20)` – reference to the txn table, aims to the txn where the copy has been done
- *dest_node_rev_id*: `varchar(64)` – a node-revision, that has originated by this copy

3.13.11 Locks

When a caller locks a file – reserving an exclusive right to modify or delete it – an lock object is created in this table.

Locks table stores these temporary locks in the tokens. A token specifies the the path, owner, comment and dates of creation and expiration. A bit xmp_p indicates that the comment is in XML format.

Structure of the Locks table

- *token*: `int(20)` – primary ID, unique lock token
- *path*: `varchar(64)` – path which the lock belongs to
- *owner*: `varchar(64)` – user who created the lock
- *comment*: `text` – the purpose of the lock
- *xml_p*: `int(4)` – bit specifying if the lock is in the xml form
- *creation_date*: `timestamp` – timestamp when the lock has been created
- *expiration_date*: `timestamp` – timestamp when the lock can be removed (or is expirated)

The test if a path is locked is not implemented yer, but can be done the same as in the BDB backend, see [9].

Chapter 4

Implementation

4.1 Layers

Files in Subversion backend layer are arranged in levels and each file depends only on services provided by files on the previous level. A file structure is similar to structures of BDB or FSFS backends and it is described in the following list.

- *id.c*, *dbt.c*, *convert-size.c*: Low-level utility functions.
- *fs.c*: Creating and destroying filesystem objects.
- *err.c*: Error handling.
- *nodes-table.c*, *txns-table.c*, *revs-table.c*, *copies-table.c*: Create and open particular database tables. Responsible for intra-record consistency.
- *node-rev.c*: Creating, reading, and writing node revisions. Responsible for deciding what gets deltified when.
- *reps-table.c*: Retrieval and storage of represented strings. This will handle delta-based storage.
- *node-origins-table.c*, *changes-table.c*: Handle additional information during a transaction.
- *next-keys-table.c*: Generating and handling new unique identificators.
- *dag.c*: Operations on the DAG filesystem. „DAG“ because the interface exposes the filesystem's sharing structure. Enforce inter-record consistency.
- *tree.c*: Operations on the tree filesystem. This layer is built on top of dag.c, but transparently distinguishes virtual copies, making the underlying DAG look like a real tree. This makes incomplete transactions behave like ordinary mutable filesystems.
- *lock.c*: Locks handling.

4.2 Public functions analysis and implementation iterations

The MySQL backend design gives us enough knowledge to begin the implementation, but it was not easy to decide where to start, because the source code of BDB or FSFS backends reach to hundreds of kilobytes. Thus, an analysis of public interface functions from BDB backend has been done and the planning of the future work was therefore easier.

The analysis was focused on some very simple operations:

- creating repository
- adding a file and commit
- adding a directory and commit
- changing a file's content
- deleting a file
- checkout of an empty repository
- checkout of full repository

All calls of public functions from the backend's public interface were tracked and results (if they were called and how often it was) are in the Appendix A. Functions that are used the most frequently during every operation are obvious. Names of functions correspond with Subversion conventions, a prefix *svn_fs_mysql__* means that a function is from library's public interface and *mysql_* prefix means internal function in the backend, which provides some services to other level of the backend. More about conventions in [8].

In the next step operations from the analysis were taken one after the other, starting with the repository creation, adding an empty directory, etc. and functions needed by these operations were implemented with all necessary subroutines. From this analysis we can also choose functions that are used most frequently and implement them in preference.

The implementation itself proceeded in iterations, which composed of several steps. First step was to recognize which functions we need to declare. These were functions that a compiler complained about. They could be empty at this time or returning a special error NOT_IMPLEMENTED (about error handling in Subversion you can read more in section 4.3). Function's arguments and location were taken from existing BDB design.

After the compilation had proceeded without errors, we could move to the next step. The second step was to run a simple Subversion operation, which ended before being successfully finished. The error written in the console told us which function we need to implement first, so we knew where to start.

The next step was the hardest – to analyze what a similar function in BDB backend should provide and to implement the same behavior in the MySQL backend. A good point was that much code in a backend isn't database-specific, so sometimes only small modifications were done. Bigger modifications had to be done in complex operations, such as deltifying of file contents or in the database access itself.

An aim of all iterations was to make as little modifications that are able to be compiled and run as possible.

The last step of all iterations was a check if all operations do the right thing and if they do it properly. The best way how to check a backend's behavior turned out to be the use of test prints on the standard output together with check of database tables' content.

4.3 Errors handling in Subversion

Subversion is very complex application which needs to be stable along very long period. Thus, an error handling policy is very important in the whole system. The error handling is described in the following text.

Almost every function in Subversion returns a pointer to a special error structure, which contains an error code, description and some other metadata, such as where an error occurred. Only very simple functions (often containing only one line of code) return some value themselves, but other could return one or more values only by using pointers in their arguments.

Every call of a function is then wrapped with C-macro `SVN_ERR`, which catches a returned error code and checks it. If some error is recognized, the macro returns the same error to the level above. On the level above the same macro is used, so the error bubbles up until some function can proceed it or the whole process is ended and the error is printed.

Some errors, such as „an item is missing in database“ could be used to indicate some state of the database, not just the error in application itself. So errors could be used for some special messages in communication between different functions on the same level, as well as on different levels. The whole approach can evoke an exceptions handling, known from higher-level programming languages.

4.4 API interfaces using virtual tables

All interfaces in Subversion are implemented using virtual tables, that are structures of function pointers in a matter of fact. Every pointer to function has well-defined arguments as well as return value type and the collection of functions with the same orientation defines the whole interface.

Virtual tables are used between Subversion layers as well as in the layers itself. For example a distribution of backend's functions to several levels is implemented using virtual tables.

4.5 Using APR and memory pooling

Apache Portable Runtime library is used in every part of Subversion and a memory pooling is the base of the whole APR, so it was used every time when we needed to work with dynamic memory. A good description of the purpose of memory pooling and why it is better than native malloc function is described in [1]:

„A memory pool is an abstract representation of a chunk of memory allocated for use by a program. Rather than requesting memory directly from the OS using the standard malloc() and friends, programs that link against APR can simply request that a pool of memory be created (using the `apr_pool_create()` *function). APR will allocate a moderately sized chunk of memory from the OS, and that memory will be instantly available for use by the program. Any time the program needs some of the pool memory, it uses one of the APR pool API functions, like* `apr_palloc()`*, which returns a generic memory location from the pool. The program can keep requesting bits and pieces of memory from the pool, and APR will keep granting the requests. Pools will automatically grow in size to accommodate programs that request more memory than the original pool contained, until of course there is no more memory available on the system.“*

„Now, if this were the end of the pool story, it would hardly have merited special attention. Fortunately, that's not the case. Pools can not only be created; they can also be cleared and destroyed, using `apr_pool_clear()` *and* `apr_pool_destroy()` *respectively. This gives developers the flexibility to allocate several – or several thousand – things from the pool, and then clean up all of that memory with a single function call! Further, pools have hierarchy. You can make "subpools „ of any previously created pool. When you clear a pool, all of its subpools are destroyed; if you destroy a pool, it and its subpools are destroyed."*

Thus, a pointer to a pool structure is presented in most of functions in whole Subversion. The good point is that we don't need to care about memory freeing so much, because every server request is proceeded using its own pool instance, so it could be released at once in the end of the proceeding.

There is only one problem when using memory pools in iterations, if we don't know the number of iteration loops at the time of compilation. It is better to create a special memory „inter-pool" object in each loop, the reason is described in [7] and you can read more about APR itself in [23].

4.5.1 Apache SQL/Database Framework

Apache SQL/Database Framework (DBD) is incorporated in Apache APR (since version 1.2). It is used for work with MySQL database and it enables a database to applications efficiently. It offers dynamic connection pooling for scalable applications, persistent connection or database-independent framework with driver modules for different databases. APR DBD Drivers are currently available for MySQL and PostgreSQL. You can read more about DBD in [24].

InnoDB was chosen as a MySQL engine because of transactions support. SQL transactions are used every time when the database is modified. Checking the status of commit operation result allows to control, if all changes proceeded without errors, so we don't need any other checks.

Every error in the database is caught using a macro `MYSQL_ERROR`, which is similar to `SVN_ERR`) and creates a standard Subversion error structure of it.

SQL queries are prepared dynamically, so there is always a threat of SQL injection. Every SQL string argument is checked using `apr_dbd_escape` function and wrapped using apostrophes. Work with binary values is a bit more complicated and using an escape function is not good enough. There are special functions prepared for binary data in APR DBD, for example we have to use prepared statements with special modification for writing.

We also need to use a special `apr_get_datum` function to read a binary entry and then use so-called bucket and bucket brigades structures, which are integrated in APR utilities library. Brigades are a kind of abstract data types, which can be used as hash maps, heap structures or doubly-linked lists. More about Bucket Brigades could be found in public API in [26].

4.6 Modifications of other parts of Subversion application

Only very little modifications had to be done outside the new backend layer, so we can say the Subversion's layered design is very good. Some modifications were made in a backend access layer, where all backends are initialized, and some changes had to be done in *svnadmin* routine. The standard command to creation of a new repository looks as follows:

```
svnadmin create repos-bdb --fs-type=bdb}
```

If we want to use MySQL backend, we need to specify the connection parameters, such as server name, login, password and a database name. All of this information is given to the svnadmin routine while the repository is creating and it is kept in *db* subdirectory in repository's directory, specifically in *connection.inf* file. So the whole command to create a new MySQL repository looks as follows:

```
svn create repos-mysql --fs-type=mysql --sql-server=localhost
--sql-dbname=mydbname --sql-login=mylogin --sql-pass=mypass
```

Note: Besides the Subversion source some changes had to be done in *autogen.sh* routine, that is used to prepare configuration setting used before *make* itself.

4.7 Difficulties during implementation

The biggest problem during implementation was debugging. Subversion is very complex system composed of a client and a server, communicating using socket streams, while the server use threads for processing requests. Even if the system has some developer modes I couldn't use standard debugging techniques, such as gdb debugger, as much as I've planned. I found better prints to standard output while processing a request, which is very primitive way of debugging, but it brings me the best results.

Besides function names from a public API some SQL queries can be printed during proceeding, when this feature is enabled. It was useful for debugging queries itself, as well as the whole backend.

And as I've mentioned in 4.2, the implementation iterations were kept as small as possible to better mistakes recognition.

4.8 Performance test against existing backends

The new MySQL backend was compared against existing backends BDB and FSFS, specifically a speed of commit and checkout operations and a repository's size. Every operation was tested three times to minimize some random computer delays. Results while using many small files (every file had a couple of kilobytes) in one commit are in Figure 4.1. Results when a little of large files (every file had a couple of megabytes) were committed and then checkouted are in Figure 4.2.

Repository's size after committing some large files is in Figure 4.3. All tests' results are interpreted in the last chapter 5.

Note: Tested on a computer with Intel Core2Duo T5550, 1.83GHz, 2GB RAM, HDD Hitachi HTS542525K9SA00, 5400rpm, Ubuntu 9.10, Linux Kernel 2.6.31-21, MySQL 5.1.

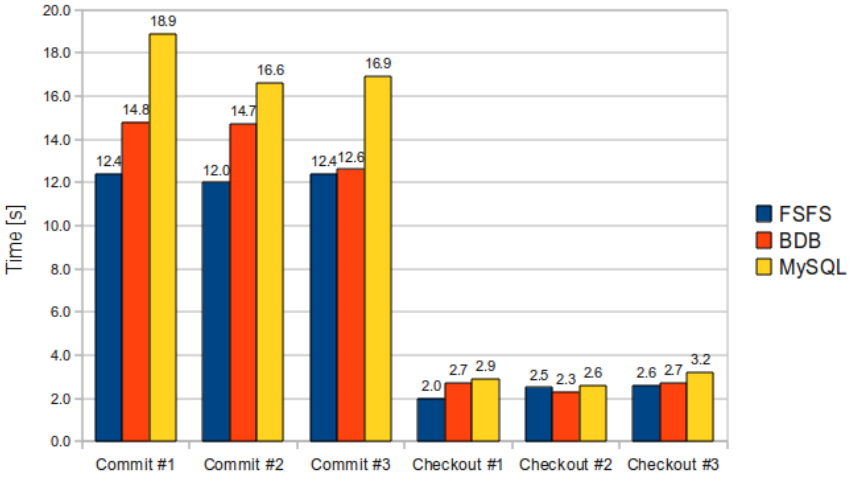


Figure 4.1: MySQL backend comparison; small files [s]

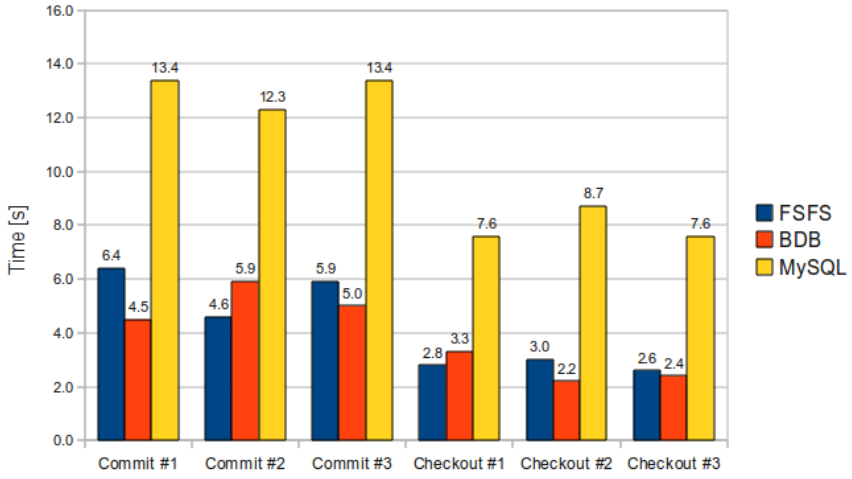


Figure 4.2: MySQL backend comparison; large files [s]

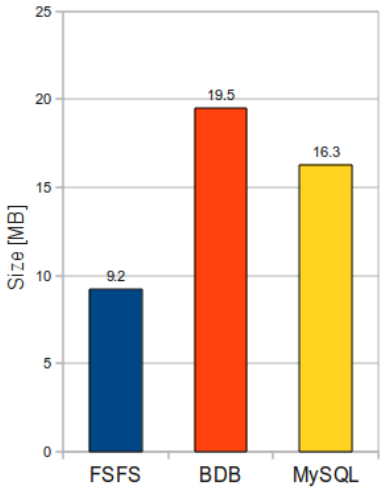


Figure 4.3: MySQL backend comparison; repository size [MB]

Chapter 5

Conclusion

Performance tests from the previous chapter are interpreted in section 5.1 and reliability of the new backend is discussed in section 5.2. Some plans for future work are advised and benefits of this work are mentioned in the last section too.

5.1 Performance conclusion

As we can see, the MySQL backend is slower than both existing backends in all operations, but it could have been expected. It is probably caused by many SQL queries that are slowed down by a connection between application and database.

We can reduce the number of SQL queries by using a special cache in the application, but the memory requirements will be a bit larger. If we used a smart cache limit, it could bring some performance improvement. The other way how to decrease the number of queries to a database is to enlarge a chunk size transmitted in one query (a chunk is one block of data read in one query).

If this backend would be used for management of large data in many thousand of revisions, tables will have several million of rows. Thus, table indexes will be very important, as well as an optimalization of tables from time to time. Index search speed could be increased by using distributed server design, where indexes could be stored on a separate server.

MySQL backend has better results only in case of repository's size in comparison to other existing backends. The size of the MySQL database is smaller than the BDB database, but it is larger than FSFS repository at the same time.

5.2 Reliability and plans for future work

The MySQL backend is not completely reliable at present and it crashes with unknown cause from time to time, concretely if delta design is chosen to store data. Also not all functions are implemented yet, so the following work should be aimed to increase the reliability, stable implementation and functionality improvement. The performance cannot be improved until the previous is done.

The whole result will be presented to Apache Subversion contributor's team and the possible continuation of work will be considered. A new development branch wasn't created in Subversion repository, in spite of planning to do so, because the community is in the middle of moving from Tigris.org to Apache Software Foundation and the development

stagnates a bit. Another complication was an Individual Contributor License Agreement, which has to be signed before any contributor involves in the project and there were not enough time to handle all these details.

I hope this work will be worth enough, even if the SQL backend wouldn't be included in Subversion official release at all. Some notice-able tests and analyses were done during the designing. As a result, I believe that some of topics (e.g. DAG structures in SQL databases in section 3.5, MySQL index performance test in section 3.7, etc.) can be used in many other applications, not only in Subversion.

Bibliography

[1] Brian W. Fitzpatrick C. Michael Pilato, Ben Collins-Sussman. *Version Control with Subversion*. O'Reilly Media, Inc., 2008. ISBN 978-0-596-51033-6.

[2] Joe Celko. Hierarchical sql [online]. `http://onlamp.com/pub/a/onlamp/2004/08/05/hierarchical_sql.html`, 2010-05-16.

[3] Scott Chacon. Git project homepage [online]. `http://git-scm.com/`, 2010-05-16.

[4] CollabNet.net. Tigris.org homepage [online]. `http://www.tigris.org`, 2010-05-16.

[5] Compaq. Vesta project homepage [online]. `http://www.vestasys.org/`, 2010-05-16.

[6] Software Freedom Conservancy. Darc project homepage [online]. `http://darcs.net/`, 2010-05-16.

[7] Subversion contributors. Apr pool usage conventions [online]. `http://subversion.apache.org/docs/community-guide/conventions.html`, 2010-05-16.

[8] Subversion contributors. Hacker's guide [online]. `http://svn.apache.org/repos/asf/subversion/tags/1.3.0-rc3/www/hacking.html`, 2010-05-16.

[9] Subversion contributors. Structure of bdb backend [online]. `http://svn.apache.org/repos/asf/subversion/trunk/subversion/libsvn_fs_base/notes/structure`, 2010-05-16.

[10] Subversion contributors. Structure of fsfs backend [online]. `http://svn.apache.org/repos/asf/subversion/trunk/subversion/libsvn_fs_fs/structure`, 2010-05-16.

[11] Subversion contributors. Subversion design [online]. `http://svn.apache.org/repos/asf/subversion/trunk/notes/subversion-design.html`, 2010-05-16.

[12] Subversion contributors. Subversion faq [online]. `http://subversion.apache.org/faq.html`, 2010-05-16.

[13] CollabNet Corporation. Collab.net [online]. `http://www.collab.net`, 2010-05-16.

[14] Schneider Robert D. *MySQL - Oficiální průvodce tvorbou, správou a laděním databází*. Grada Publishing, a.s., 2006. ISBN 80-2471-516-3.

[15] Adi Malinaru Daniel Aioanei. General trees persisted in relational databases [online]. http://www.codeproject.com/KB/database/persisting_trees.aspx, 2010-05-16.

[16] Dhananjay. Effective way to expand data tree in sql server [online]. http://dgoyani.blogspot.com/2005/09/effective-way-to-expand-data-tree-in.html, 2010-05-16.

[17] Kemal Erdogan. A model to represent directed acyclic graphs (dag) on sql databases [online]. http://www.codeproject.com/KB/database/Modeling_DAGs_on_SQL_DBs.aspx, 2010-05-16.

[18] Shlomi Fish. Available cvs alternatives [online]. http://better-scm.berlios.de/alternatives/, 2010-05-16.

[19] Dennis W. Forbes. Versatile high performance hierarchies in sql server [online]. http://www.yafla.com/papers/sqlhierarchies/sqlhierarchies.htm, 2010-05-16.

[20] Apache Software Foundation. Subversion mailing list [online]. http://subversion.apache.org/mailing-lists.html, 2010-05-16.

[21] Oracle Education Foundation. Berkeley db architecture [online]. http://oukc.oracle.com/static05/opn/oracle9i_database/34313/050306_34313/index.htm, 2010-05-16.

[22] Oracle Education Foundation. Oracle berkeley db products [online]. http://www.oracle.com/technology/products/berkeley-db/index.html, 2010-05-16.

[23] The Apache Software Foundation. Apache portable runtime project [online]. http://apr.apache.org, 2010-05-16.

[24] The Apache Software Foundation. Apache sql/database framework homepage [online]. http://apache.webthing.com/database/, 2010-05-16.

[25] The Apache Software Foundation. Apache.org homepage [online]. http://apache.org, 2010-05-16.

[26] The Apache Software Foundation. Bucket brigades api [online]. http://apr.apache.org/docs/apr-util/0.9/group__APR__Util__Bucket__Brigades.html, 2010-05-16.

[27] Inc. Free Software Foundation. Arch project homepage [online]. http://www.gnu.org/software/gnu-arch/, 2010-05-16.

[28] Inc. Free Software Foundation. Cvs project homepage [online]. http://www.nongnu.org/cvs/, 2010-05-16.

[29] Martin Furter. More repositories in one database [online]. http://svn.haxx.se/dev/archive-2010-04/0374.shtml, 2010-05-16.

[30] Paul Holden. Severe performance issues with large directories [online]. http://svn.haxx.se/dev/archive-2010-04/0180.shtml, 2010-05-16.

[31] Edmund Horner. A quick and dirty mysql backend [online]. http://homepages.paradise.net.nz/ ejrh/subversion/mysql/, 2010-05-16.

[32] Eugene Lepekhin. Trees in sql databases [online]. http://www.codeproject.com/KB/database/Trees_in_SQL_databases.aspx, 2010-05-16.

[33] Canonical Ltd. Bazaar project homepage [online]. http://bazaar.canonical.com/en/, 2010-05-16.

[34] PureCM.com Ltd. Purecm homepage [online]. http://www.purecm.com, 2010-05-16.

[35] Subversion mailing list. Subversion newbie thoughts [online]. http://svn.haxx.se/users/archive-2005-07/0862.shtml, 2010-05-16.

[36] Philipp Marek. Another performance issues with large directories [online]. http://svn.haxx.se/dev/archive-2010-04/0259.shtml, 2010-05-16.

[37] Philipp Marek. Storing content data [online]. http://svn.haxx.se/dev/archive-2010-04/0002.shtml, 2010-05-16.

[38] Peter Miller. Aegis project homepage [online]. http://aegis.sourceforge.net, 2010-05-16.

[39] MIT. How fsfs is better [online]. http://web.mit.edu/ghudson/info/fsfs, 2010-05-16.

[40] Graydon Hoare Nathaniel Smith. Monotone project homepage [online]. http://www.monotone.ca/, 2010-05-16.

[41] Jesper Noehr. Mercurial project homepage [online]. http://mercurial.selenic.com/, 2010-05-16.

[42] Oracle. A comparison of oracle berkeley db and relational database management systems [online]. http://www.oracle.com/database/docs/Berkeley-DB-v-Relational.pdf, 2010-05-16.

[43] Perforce. Perforce project homepage [online]. http://www.perforce.com, 2010-05-16.

[44] Greg Stein. Transactions properties table [online]. http://svn.haxx.se/dev/archive-2010-04/0022.shtml, 2010-05-16.

[45] Pavel Szalbot. Stromy v sql [online]. http://www.abclinuxu.cz/clanky/navody/stromy-v-sql, 2010-05-16.

[46] Tigris.org. Subversion documentation [online]. http://svn.collab.net/svn-doxygen/, 2010-05-16.

[47] Ventanazul. Why i decided to use fsfs over berkeley db with subversion [online]. http://www.ventanazul.com/webzine/articles/subversion-changing-from-bdb-to-fsfs, 2010-05-16.

[48] Rob Volk. More trees and hierarchies in sql [online].
http://www.sqlteam.com/article/more-trees-hierarchies-in-sql, 2010-05-16.

[49] David A. Wheeler. Comments on open source software [online].
http://www.dwheeler.com/essays/scm.html, 2010-05-16.

Appendix A

Public Functions frequencies per various operations

Functions from the public backend interface were traced to get the frequencies of their calling inside various operations. Count of calling of each of the functions are in the table A.1. Operations are specified a bit in the legend.

Legend:

- cre: creation of a repository
- coe: checkout of an empty repository
- caf: commit after a file was added
- cad: commit after a directory added
- upf: update of the repository with a file and a directory
- ccf: commit after a file was changed
- cdf: commit after a file was deleted

	Operations (see legend above)							
Function name	*cre*	*coe*	*caf*	*cad*	*upf*	*ccf*	*cdf*	*total*
mysql_apply_text								0
mysql_apply_textdelta			1			1		2
mysql_bdb_logfiles								0
mysql_bdb_pack								0
mysql_bdb_recover								0
mysql_bdb_set_errcall								0
mysql_closest_copy								0
mysql_contents_changed								0
mysql_copied_from								0
mysql_copy								0
mysql_create								0
mysql_delete_fs								0
mysql_delete_node							1	1
mysql_dir_entries		2	1	2	2	1	1	9
mysql_file_contents								0
mysql_file_checksum			1			1		2
mysql_file_length								0
mysql_get_description								0
mysql_get_file_delta_stream								0
mysql_get_mergeinfo		2	1	1	1	1	1	7
mysql_history_location								0
mysql_history_prev								0
mysql_hotcopy								0
mysql_change_node_prop								0
mysql_check_path		3			4		1	8
mysql_make_dir				1				1
mysql_make_file			1					1
mysql_merge								0
mysql_node_created_path								0
mysql_node_created_rev		1			1	1	1	4
mysql_node_history								0
mysql_node_id		5	1	1	8	1	2	18
mysql_node_origin_rev								0
mysql_node_prop								0
mysql_node_proplist								0
mysql_open		2	1	1	1	1	1	7
mysql_open_for_recovery								0
mysql_paths_changed								0
mysql_props_changed		1			1			2
mysql_revision_link								0
mysql_upgrade								0

Table A.1: Functions frequencies in various operations (part 1)

	Operations (see legend above)							
Function name	*cre*	*coe*	*caf*	*cad*	*upf*	*ccf*	*cdf*	*total*
svn_fs__mysql_version	3	4	2	2	2	2	2	17
svn_fs_mysql__abort_txn								0
svn_fs_mysql__begin_obliteration_txn								0
svn_fs_mysql__begin_txn			1	1		1	1	4
svn_fs_mysql__commit_obliteration_txn								0
svn_fs_mysql__commit_txn			1	1		1	1	4
svn_fs_mysql__deltify			1	1		1	1	4
svn_fs_mysql__generate_lock_token								0
svn_fs_mysql__get_lock								0
svn_fs_mysql__get_locks								0
svn_fs_mysql__get_uuid		5	2	2	3	2	2	16
svn_fs_mysql__change_rev_prop								0
svn_fs_mysql__change_txn_prop			2	2		2	2	8
svn_fs_mysql__change_txn_props			1	1		1	1	4
svn_fs_mysql__id_compare					2			2
svn_fs_mysql__id_parse	3	7	65	69	24	98	87	353
svn_fs_mysql__id_unparse	6	14	69	62	20	84	63	318
svn_fs_mysql__list_transactions								0
svn_fs_mysql__lock								0
svn_fs_mysql__open_txn								0
svn_fs_mysql__purge_txn								0
svn_fs_mysql__revision_prop			2	2		2	2	8
svn_fs_mysql__revision_proplist		1			1			2
svn_fs_mysql__revision_root		5	3	3	5	3	3	22
svn_fs_mysql__set_uuid								0
svn_fs_mysql__txn_prop								0
svn_fs_mysql__txn_proplist								0
svn_fs_mysql__txn_root			1	1		1	1	4
svn_fs_mysql__unlock								0
svn_fs_mysql__youngest_rev		2	2	2	1	2	2	11

Table A.2: Functions frequencies in various operations (part 2)