FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's thesis in Informatics

Development of an associative file system

Lars Hupel

FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's thesis in Informatics

Development of an associative file system

Entwicklung eines assoziativen Dateisystems

Author:	Lars Hupel
Supervisor:	Prof. Alfons Kemper, Ph.D.
Advisors:	Prof. Alfons Kemper, Ph.D.
	Dipl.-Inf. Florian Funke
Date:	September 15, 2011

I assure the single handed composition of this bachelor's thesis only supported by declared resources.

Ich versichere, dass ich diese Bachelorarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15. September 2011 Lars Hupel

Abstract

Organizing multimedia data, e. g. pictures, music or videos is a rather common use case for modern file systems. There are quite a number of applications which try to expose an user-friendly interface for dealing with tagging, sorting and editing these files. This becomes necessary because sets of such files do not have an intrinsic hierarchic structure. For example, pictures taken with a digital camera carry *EXIF* metadata which can be used to retrieve a picture by date, time or location instead of an artificial folder structure.

However, the major problem shared by all of those multimedia applications is that the files are actually stored in folders on a traditional file system. As such, any operation done by an user *outside* of the application leads to inconsistencies *inside* of the application. Also, metadata produced by one application cannot be consumed by another one because of proprietary formats.

In this thesis, a file system which uses the established RDF standard for storing metadata is developed which imposes only little structural requirements on the data. The system features both an API which enables high-level operations on file contents and metadata and a CLI which resembles ideas from versioning systems like *Git*.

Also, a formalization of the most important operations is given, including a concept of transactions, which has been adapted from relational database systems to fit in the environment of a file system.

Zusammenfassung

Das Verwalten von Multimediadaten, z. B. von Bildern, Musik oder Videos, ist ein vergleichsweise häufiger Anwendungsfall für moderne Dateisysteme. Es gibt eine Reihe von Anwendungen, die versuchen, dem Anwender eine benutzerfreundliche Oberfläche zum Verschlagworten, Sortieren und Bearbeiten anzubieten. Dies wird notwendig durch die Tatsache, dass einer Gruppe solcher Dateien keine hierarchische Struktur innewohnt, die es erlauben würde, eine sinnvolle Ordnerhierarchie anzuwenden. Bilder, die mit einer Digitalkamera aufgenommen wurden, sind beispielsweise mit *EXIF*-Metadaten versehen, welche geeignet sind, einzelne Bilder nach Datum, Zeit oder Ort zu filtern.

Das größte Problem aller solcher Multimediaanwendungen ist, dass Dateien in Ordnern auf einem gängigen Dateisystem abgelegt sind. Daher kann eine vom Nutzer *außerhalb* der Anwendung durchgeführte Operation grundsätzlich *innerhalb* der Anwendung nicht zuverlässig nachvollzogen werden. Erschwerend kommt hinzu, dass das Metadatenformat einer Anwendung nicht zwangsläufig kompatibel sein muss mit dem einer anderen, was die Austauschbarkeit stark einschränkt.

In dieser Arbeit wird ein Dateisystem entwickelt, welches den etablierten RDF-Standard für Metadaten nutzt und nur sehr geringe Anforderungen an die Struktur der Daten stellt. Das System stellt sowohl eine API bereit, welche Operationen auf hoher Ebene unterstützt, als auch ein CLI, welches Ideen von Versionierungssystemen wie *Git* wiederverwendet.

Außerdem wird eine Formalisierung der wichtigsten Operationen inklusive einem Transaktionskonzept, welches von relationalen Datenbanksystemen mit Hinblick auf die Anforderungen eines Dateisystemes adaptiert wurde, vorgestellt.

Contents

1 Introduction

Today's major operating systems share a number of concepts which are similar across them. One of those concepts is the file system, which is organized hierarchically in GNU/Linux, OS X, (or more general in Unix-like systems) and Windows.

Except for minor differences, it is common to all of those file system models, that there are one or more (the latter only in Windows) roots, typically denoted by `/` (or `A:`, `B:`, ..., respectively). On user level, the smallest unit of data is a *file* which is seen as a sequence of bytes which has a unique name in its *folder*. Folders may also contain other folders, forming a tree structure. Every file can be addressed by specifying the folders traversed until reaching that file.

Although similar on the surface, the different file systems flavour their notion of a "file" with other concepts: Windows' NTFS, for example, actually organizes file contents in *streams* [5; 19, § 11.8.2]. A file may contain more than one stream – known as *Alternate Data Streams*.

On the other hand, many Unix-style file systems such as XFS support *extended attributes* which serve as a key-value store for file metadata. However, depending on the concrete implementation, the size of that store may be limited to what fits into the file's *inode*[1].

Despite their limitations, all these approaches would be sufficient for adding metadata support to existing file systems. The problem is, that – contrary to the recommendations [1] – very few applications employ those capabilities.

Many media player application come with a built-in media database, though. Their usual approach is to build a database layer on top of the existing filesystem, allowing high flexibility. The major drawback is that, as mentioned in [2], "most people do not have static music collections; songs are always being renamed and moved". This extra layer has no possibility to find out which operations the user has made. However, the user expects the media collection to not degenerate, thus forcing the application to apply heuristics to detect file-level changes. This may or may not work, depending on which kinds of changes have been made.

The main insight here is that every approach to add an extra layer which is exclusive to the application using it is incomplete by design. One of the goals of this project was to solve that by storing file contents in a dedicated *target* (where user access can be restricted) which takes advantage of the speed and safety of

[1]For an explanation, see section 2.1.1.

existing file systems, and metadata in a relational DBMS which allows fast and flexible queries.

The second major feature, although somewhat unrelated to metadata organization, is the support for sessions. It occurred to the author that grouping of related operations is surprisingly a missing concept from file systems[2]. Those transactions or *sessions*, as they are called in this project, may not be useful for the user, but greatly reduce the amount of code required for writing fail-safe installation programs which have to deal with manual rollback, e. g. if a copy operation fails because of insufficient space. Here, a combined session mechanism for file content and metadata is presented and formalized. The underlying transaction layer is suitable for general use in traditional file systems but has not been decoupled from the whole application.

In conclusion, this thesis discusses the development of various aspects of Angelfish, a file system consisting of an API and a sample implementation for dealing with files based on their metadata.

[2]As described in section 3.2, there are existing approaches to that, e. g. in NTFS.

2 Data modelling

This section starts with an exploration of the hierarchic model which underlies the overwhelming majority of past and contemporary file systems. After that, a new, *associative*, folder-less model is presented.

2.1 Traditional file systems

Despite what the word "traditional" suggests, almost all current file systems for the major operating systems, like *ext4*, *FAT32* or *NTFS* count as *traditional file systems* in this elaboration. By these means, the term also denotes the model of file organization which is imposed by the file system layer of the operating system itself to distinguish it from the model introduced in the following sections.

2.1.1 Basic terminology

A *file* in the traditional sense is a name for a chunk of data, typically seen as just a sequence of bytes. Traditional file systems organize files in *folders* (or *directories*), where each file is member of exactly one folder and each folder is member of exactly one folder. At the top of the hierarchy, there is a root folder, which is represented by `/` in Unix-like operating systems, whereas Windows manages multiple roots represented by a letter and a colon, e. g. `C:`. Here, the discussion is restricted to Unix-like systems (excluding special files like sockets) for brevity.

The *path* of a file (or folder) is recursively defined as the path of the parent folder, followed by the path separator character / and the name of the file (or folder) itself. The mapping between files and paths is a bijection.

The actual data (or rather pointers to it) and corresponding metadata of files are stored in *index nodes* (short *inodes*). Those inodes carry no information about the file name or location in the hierarchy. Directories contain a mapping between a name and an inode, if the name denotes a file, or a subdirectory. Those entries pointing to an inode are called *links*[3]. An inode may be referenced more than one time which can be visualized as creating an exact copy of a file at another place, but without physically duplicating the contents. As soon as the last link to the inode is deleted, the file system deletes the inode and the corresponding data. In order to achieve that, each inode contains a counter keeping track of the number of links or *hard links* to it.

[3]Similarly, the system call to remove such an entry is called `unlink`.

Another possibility to reference a file in a different location is the use of *symbolic links*. Unlike hard links, a symbolic link references its own inode independent from the destination of the link. This inode contains a flag marking it as link and the path of the destination. Note that the term "destination" only makes sense for symbolic links, as all hard links to the same file are treated equally. Symbolic links however induce a directed graph with no guarantees of acyclicity. The destinations do not even have to exist.

A great difference between these two types of links is that hard links to a particular file have to reside in the same file system. This is caused by the fact that directory entries reference inodes by their numbers which are not unique across file system borders. Symlinks however may refer to any path, regardless of existence or file system.

For details about the structure of traditional file systems, refer to § 4.3 (files, directories, inodes, links) and § 11.8 (NT file system) of [19].

2.1.2 The Filesystem Hierarchy Standard

A good example for a hierarchy is the *Filesystem Hierarchy Standard* (short *FHS*), which specifies a common structure of the root file system of Unix-like operating systems (see table 1 for an overview[4]) [17].

This specification relies on an intrinsically hierarchic manner in which files are organized, which is suitable for many use cases. There are only very few circumstances under which a file would belong to more than one path, which is usually solved by using symbolic links, thus giving the file in question a "canonical" path and creating links to this path in all other desired places. Examples for that are executable files which belong to `/usr/bin`, but are part of larger applications under `/opt`. The use of hard links is not very wide spread in this context though due to the limitations (same file system, directories are not hard-linkeable).

2.1.3 Problems

As described above, the need to place a file under more than one path arises from time to time. The approach with symbolic links works quite well for applications which are installed and updated with a package manager, as links are taken care of by various automatisms. Assuming good maintenance of the package by the vendor, this guarantees that no dead links occur during an upgrade, e. g. when

[4]the descriptions are taken directly from the standard

Path	**Description**
`/`	Root of the hierarchy
`/bin`	Essential user command binaries (for use by all users)
`/boot`	Static files of the boot loader
`/dev`	Device files
`/etc`	Host-specific system configuration
`/home`	User home directories
`/lib`	Essential shared libraries and kernel modules
`/media`	Mount point for removeable media
`/mnt`	Mount point for a temporarily mounted filesystem
`/opt`	Add-on application software packages
`/root`	Home directory for the root user
`/sbin`	System binaries
`/srv`	Data for services provided by this system
`/tmp`	Temporary files
`/usr`	Shareable (between hosts), read-only data
`/var`	Variable data files (spool, logs, etc.)

Table 1: / according to the FHS [17]

the package name has changed. However, if an application is managed by the local system administrator, some files might get orphaned because they slip the operator's attention.

The same is true for hard links, where reference leaks may occur. Deleting a file does not actually free memory if there is another link to it. Although there will be no broken links, files might get superfluous without anyone noticing it.

Both kinds of links are problematic under these circumstances as there is no possibility to list all links to a particular file without traversing the whole directory tree which might be an expensive operation.

Altogether, the FHS is a widely adopted standard. It is working quite well for operating systems, because – amongst other things – the descriptions of the specified folders have only very small overlaps and there is a package manager in modern operating systems taking care of the system files and links.

2.1.4 Multimedia usage

The problems stated above become more important with respect to user data, i. e. data which does not belong to the system and is controlled by the user. The best examples are multimedia files like pictures and movies. To illustrate that, consider a library of music files. One could possibly classify the files by artist, which would be the first hierarchy level. The second level might be the album. Now, filtering the library with the predicate "album released prior to 1980" is obviously non-trivial for the user, as the predicate applies to the second level of the hierarchy. Moving the album level up, thus obsoleting the artist level, would not help at all, because

- albums might have multiple artists,
- songs may appear on different albums,
- the user might want query by genre or length of the song or
- the library contains single songs which do not belong to an album,

to name just a few problems. Clearly, the traditional concept of hierachic storage is not suitable for this kind of library. Those files *should* be classified by their metadata, not based on artificial structural limitations imposed by the system.

Media libraries There are quite a number of applications that try to solve the "multimedia problem". As mentioned in the introduction, *Amarok* is one of them which approaches it with a technology called *AFT* (short for *Amarok File Tracking*) [2]. It is based on hashes which are calculated from several properties of a given file. If this hash is known to Amarok, it is assumed that it is already part of the media collection and the path of the file is updated accordingly. If the hash is unknown, a new entry is added to the library. It is also possible to use *MusicBrainz* identifiers which are computed by "characteristics of the audio data" [2]. However, all these approaches have drawbacks:

- storing the hash in the file involves *modifying* the file content, thus mixing content and metadata
- calculating the hash each time a new file is seen may result in the behaviour that the same file, renamed and with some of the tags updated is recognized as a different file
- MusicBrainz identifiers may recognize two different files as the same, e. g. when they contain different encodings of the same song

In conclusion, this may work reasonably well in practice, but it is obvious that media libraries which are based on adding another layer which is opaque to the user are incomplete by design.

Interoperability Libraries as described above have also two other problems: domain and interoperability. Amarok is a music player, thus its library manages only audio files. Other applications are similar in the sense that they have a restricted scope and do not accept other file types. While it is only sensible for some kinds of data to be organized in such a way, a file system should not reject contents which it is not aware of – in fact, it should not be aware of *any* particular file type by default.

The second point is the difficult interoperability. While it is technically possible to exchange library contents between applications which have a documented structure, this always requires a fair amount of tinkering to write a specific conversion routine. Here, the RDF standard is used instead (see section 2.3.2) which should help keeping the metadata reusable.

2.2 Previous work

The most notable, though cancelled previous approach to build a file system supporting both structured and unstructured data was *WinFS* (short for *Windows Future Storage*). There are not many resources about that topic, but the official blog at MSDN [6] serves as a good starting point. It also used a relational database system to store the metadata, but used a schema-based model. Using such a schema makes the file system aware of the different content types, whereas a generic model is used in this file system. Also, it provided featured a type system which resembled some object oriented concepts like inheritance, making it possible to manage entities in the file system which do not strictly qualify as "files" (e. g. contacts). In the end, Microsoft decided to not "ship WinFS as a separate, monolithic software component" [6[5]], but to incorporate parts of the technology into other products.

2.3 ANGELFISH's data model

In this data model, there is a strict distinction between "data" and "metadata" which operates on the semantic level. While the former denotes sequences of bytes which have a well-defined content type as stated by the second part of the MIME standard [12], such as `image/jpeg`, the latter denotes data about such content.

[5]Article "Update to the update", `http://blogs.msdn.com/b/winfs/archive/2006/06/26/648075.aspx`, June 26, 2006

2.3.1 Files and Blobs

A sequence of bytes (or a file in traditional file systems) is called a *blob* in the style of Git [7]. Blobs are stored on a *target*, which is simply a folder in an existing file system which must be empty prior to setting up Angelfish and must not be touched by the user afterwards (similar to Git's `.git` folder).

Zero or more blobs (usually at least one) are grouped together in *files*. On a low level, there are no restrictions on which blobs form a file. On a semantic level however a file should denote a piece of information with the blobs being different representations of it. As an example, consider a music recording as a file which has an OGG and an MP3 encoded blob. There is also no problem in adding computed data to a blob, e. g. a search index or hash sums. This concept needs to be used carefully as either the system or the user has to take care of keeping all blobs of a file in sync. Usually, each file will have one human-generated blob and zero or more system- or application-generated blobs. There may also be virtual blobs which are not stored on the target but lazily computed each time the contents are requested (cf. KDE's KIO slaves [8]). Every blob is part of exactly one file.

Objects Both files and blobs are called *objects*. Files are identified by an UUID [11] and have no "file name". The blobs contained in a file are identified within the file by a alphanumerical *name*, whereas the lexical rules of C identifiers [15, § 6.4.2.1] should be applied[6]. If hierarchic names are desired, the name parts should be separated with a full stop. It is encouraged that human-generated blobs are named `default`. In the global scope, a blob can unambiguously be addressed with a pair of the file UUID and the name of the blob.

UUID An *UUID* (short for *Universally unique identifier*) is a 128-bit number, usually written in hexadecimal notation, e. g. `180547ff-7563-4174-b42b-4cee0e0941d8`. There are different types of UUIDs depending on the way they are generated (hash-based, MAC address, time stamp or random). In this file system, only randomly generated identifiers are used to identify files.

Those UUIDs only have 122 bit of entropy, because several bits are fixed. An approximation for the collision probability is

[6]in short: an identifier begins with an underscore or a letter, followed by underscores, letters or digits

$$\begin{aligned} P(n) &= 1 - \frac{2^{122}!}{(2^{122}-n)!(2^{122})^n} \\ &\approx 1 - \exp\left(-\frac{n^2}{2 \cdot 2^{122}}\right) \end{aligned}$$

Even for 10^9 files, the probability for collisions would stay below 10^{-19}, which is reasonably small.

2.3.2 Metadata

On the other side, there is the metadata which is completely independent of the other objects. During the planning phase, a self-defined metadata structure using a key-value attributes which are validated against schemas was considered by the author. However, a design decision in favour of the *RDF* (short for *Resource Description Framework*) was made, as this is a common standard in the *Semantic Web* domain, striving, amongst other things, for adding machine-readable metadata to web pages [9].

Although RDF is used in this project without any connections to the web, it is very well suited because of its simple structure: Basically, a set of RDF *triples* define a directed graph. Every triple consists of a subject, a predicate and an object. Hence, a triple represents a labelled edge from the subject node to the object node. As usual, every component of the triple may be namespaced. Objects carry additional type information.

To apply RDF to the file system, the restriction that subjects have to be blobs has been made. This prevents adding arbitrary triples which have no relation to the data on the file system. More importantly, files cannot be used as subjects or objects for triples. This is a trade-off in the current implementation, because that greatly simplifies the session model. A workaround for that is to have an empty `rdf` blob in a file and use that as a placeholder for its file.

Namespaces for attributes and the object types are implemented as *prefixes*. A prefix consists of an unique name and an identifying *URI* (short for *Uniform Resource Identifier*). The name is just used as a shorthand for the URI and may be different across file system instances. A type references a prefix and has an identifying name. In summary, a triple consists of

- a reference to a blob

- a predicate (pair of prefix and name as character sequence)
- an object (pair of type and value as character sequence)

If the object type is set to a system-defined default, it is interpreted as a blob. In that case, the representation of the blob as character sequence depends on the implementation.

3 Transactions and Sessions

In this section, the concept of "sessions" is introduced. First, the traditional *ACID* transactional semantics of relational databases are described. After that, we will consider the author's attempt of porting those to a relational file system, including a brief comparison to database transactions.

3.1 ACID transactions

Transactions are a fundamental part of most relational database systems, that is, for a fixed sequence of operations it is the properties *atomicity*, *consistency*, *isolation* and *durability* – in short *ACID* – are guaranteed [3; 18, § 9.5].

Atomicity ensures that either every or no operation in a sequence is executed. If all operations can be applied cleanly, the final state is recorded (*commit*). If not (or the user decides to abort the sequence), a *rollback* to the state before the first operation occurs.

Consistency ensures that the system is in a consistent state after the execution of a sequence of operations if it has been in a consistent state before. This property operates on a semantic level, thus, for example, requiring that foreign key constraints hold.

Isolation avoids that multiple concurrent transactions affect each other. A common problem is the order of execution if two transactions read and write the same data.

Durability ensures that a commited transaction persists in the database, that is, neither failures nor competing transactions overwrite the results of the original transaction. To undo the transaction, a new and inverse transaction is needed.

3.1.1 Problems with concurrent transactions

Assume that a database system performs no checks before a transaction is committed. In that case, a number of problems which violate isolation may occur, depending on which operations are performed concurrently [18, § 11.1; 16, § 4.28]:

Lost update a change made by one transaction is overridden or obsoleted by another transaction

Level	$\mathbf{P_0}$	$\mathbf{P_1}$	$\mathbf{P_2}$	$\mathbf{P_3}$
Read uncommitted	✗	✓	✓	✓
Read committed	✗	✗	✓	✓
Repeatable read	✗	✗	✗	✓
Serializable	✗	✗	✗	✗

P_0: lost update, P_1: dirty read, P_2: non-repeatable read, P_3: phantom read

Table 2: Isolation levels as defined by the SQL-92 standard

Dirty read a change is based on data from another transaction which has not been committed yet and might be aborted later

Non-repeatable read the result of two subsequent reads of the same object (i. e. a tuple) in one transaction differ, because another transaction modified that object

Phantom the results of two subsequent reads of a number of objects in one transaction differ, because another one added a new object between the reads

3.1.2 SQL isolation levels

Despite the rather strict ACID model, the isolation property is often relaxed. In order to cope with the problems described above, the SQL-92 standard defines the four *isolation levels* which specify which of them may occur [16, § 4.28] (see table 2 for an overview). This means that the user explicitly states that isolation violations are expected and, depending on the isolation level the user defined, the database system guarantees which problems may or will not occur.

Observe that none of the levels allow lost updates, as this is considered critical and should never occur in any application. The strongest level guaranteeing full isolation is "Serializable" which requires that the transaction may be executed in sequential order. It is also the only level which fully prevents consistency errors.

3.2 Journalling file systems

Modern file systems like *ext4* or *NTFS* have *journalling* capabilities, meaning that they are able to recover to a consistent state after failing to complete an operation, e. g. due to a power outage. This mechanism has become necessary to avoid lengthy file system checks on a reboot and to minimize the loss of data[7]. This behaviour corresponds to the durability guarantee from above. However, the journal only ensures that the file system is on a consistent state on file level, not on application level. If an application, e. g. an installation routine, started but did not succeed, it is up to the user or this particular application to detect that condition and act upon it, i. e. finish all remaining steps or roll back to the previous state. This can be painful for the developer, as complex error handling is necessary, and for the user, when multiple such transactions are to be executed concurrently.

In the past, various efforts have been made to solve these problems: e. g. *Transactional NTFS*, introduced by Microsoft as an extension to NTFS, which is available since Windows Vista [4]. However, it seems that these methods are not heavily used in practice.

3.3 Transactions in an associative file system

Databases and file systems differ in a fundamental point: Database transaction should be short lived to ensure the efficiency (and in turn, feasibility) of ACID, whereas file system transactions may last a couple of hours as the user mostly operates with the system directly. Thus, a very lightweight concept of transactions has to be introduced, leading to some restrictions with respect to the ACID model on application level.

3.3.1 Requirements

If we assume that no operation is executed outside of a transaction context, consider the following use case: The user starts an interactive transaction s_1 and operates on a couple of files. At some point he realizes that he needs some files from a zipped file, so he unpacks the contents in a different transaction s_2. s_2 now succeeds and commits its results to the file system. If full isolation would be guaranteed, there would be no way for s_1 to use the results of s_2. Now, one could argue that the unpacking process should not open a new transaction but rather use the existing transaction s_1. However, if the unpacking process fails, s_1 will be left in an inconsistent state forcing the user to either manually roll back to a "good" state or to abandon s_2, resulting in data loss.

In conclusion, transactions for a file system should fulfil the following requirements:

[7]some systems like ext4 allow users to choose between a number of throughput/safety levels

1. never lose any data the user entered unless a rollback is explicitly requested
2. from that, the ability to redo transactions up to the point before a crash without committing, allowing the user to continue working from this point
3. allow the use case stated above where isolation is not needed
4. allow long running transactions which do not prevent others from running

Of course, these requirements are based on the assumption that both the underlying database system and the target storage operate correctly.

In conclusion, it is reasonable to call transactions in Angelfish *sessions* as this illustrates the higher level and the different semantics than database transactions.

In the following, only structural changes (adding or removing blobs, files or storing content to blobs) are regarded.

3.3.2 Terminology

Any operation executed by the file system must be wrapped in a *session*. Thus, a session is a sequence of operations with a couple of attributes:

active the session has been opened and is currently in use

pending the state between the request to commit the session and the actual commit

stale a session where the corresponding process terminated unexpectedly

finished the transaction is committed or has been rolled back

As stated in section 2.3.1, files and blobs are called *objects*. Opening an object creates a *handle* which is used to determine when pending sessions are allowed to run. Changes produce a *journal* entry[8].

3.3.3 Handles and sessions

Handles are associated to each session and its open files and blobs, regardless whether the intention is to modify them. Note that opening a file does not imply allocating handles for each of its blobs. Also, two or more sessions are allowed to have a handle on a file or blob simultaneously, which is the expected behaviour for file systems.

[8]See section 3.5 for a formal specification of these operations.

In order to support lightweight sessions, opening a session and allocating handles are very cheap actions. It is immediately clear that opening a session should always succeed, unless there is a artificial limit on the number of parallel sessions. Additionally, it makes sense that allocating handles should also never fail – even in the case that some other session intends to remove the associated object – because of two reasons: In the first place, the system does not (and cannot) know whether a competing session will ever succeed, which may lead to hastily denying a handle. Secondly, it allows the system to postpone those kind of checks to the moment when a commit is requested which increases the (perceived) interactivity.

By default, handles will not be released during a session. The rationale here is, again, isolation. An object which has been opened is seen as a candidate for subsequent visits thus avoiding changes from another session to this object to be visible until it is sure that the object is not needed any more. This model eliminates the need to close handles in an application, but it also requires developers to carefully design the extent of a session.

3.3.4 Isolation levels

A key decision on the handling of isolation was to choose when the commit of a session takes place. As the underlying database system is memory-based (which will be discussed in section 4), it is very cheap to work with the database and there is (almost) no need to buffer data in the file system process. Therefore, Angelfish follows the strategy that – instead of immediately committing a session and prevent these changes to be visible in other session, which would require a significant memory overhead – it defers the commit to a point when it is sure that no other session is affected. In any case, changes in an uncommitted session are not visible in other session.

This strategy introduces a new problem: When a session is set to pending, the system has to determine when to perform the commit. This decision depends on the configured level of isolation and on the handles which are held by other sessions. The easier implementation is to not accept the commit request if other handles are blocking. In that case, the user has to repeat the request at a later time.

Similar to the isolation levels defined in SQL, there is a level with full guarantees and some weaker variants. They do not compare directly to the SQL levels, because "our" levels specify which changes can be exported from a session, whereas SQL levels specify which changes can be imported into a session.

Also, sequenced execution of session is not the expected behaviour of a file system, because only one session may be active at commit time. Therefore, the following weaker isolation levels exist:

Almost full There may be other active sessions, but none of them may hold a handle on any object. This is the default choice and does not prevent phantoms.

File exclusive In this level, new files may be added at any time. A file may also be removed if there is no handle on it and no handle on any of its blobs. Adding or removing a blob also requires that there is no handle on this file.

Blob exclusive This level relaxes some of the restrictions of "File exclusive": Adding a blob may be done at any time and removing a blob only requires that no other session is holding a handle on it.

No isolation Structural changes may occur at any time, without respect to other sessions. This is the minimal level of isolation as all changes immediately propagate to other sessions. It is also unsafe in the sense that durability is no longer guaranteed.

There is no need for a global isolation level, because every time a commit is requested, an individual level may be defined. This is especially useful when there is one long running interactive session s_1 where data from s_2 needs to be available, but not from s_3. In that case, one would specify e. g. "Blob exclusive" for s_2, but "Full" for s_3. The level for s_1 does not matter, so "No isolation" would be a sensible choice.

3.4 Examples

Before formalizing the above definitions, the following scenarios are presented to illustrate the intention behind the isolation levels. In the following examples, s_i represents a sessions, while f_i denotes a file and $(f_i, n_1), (f_i, n_2), \ldots$ the (named) blobs of the file f_i. The figures show the timelines for each session with the corresponding operation (green if it succeeded before commit time, red otherwise).

Scenario 1. *s_1 adds a file and a blob and commits. s_2 tries to access the file before s_2 committed and fails, but succeeds after s_1 committed. This behaviour can be observed using any weak isolation level for s_1 (figure 1), but not for full isolation, as s_1 would not be able to commit because s_2 is active.*

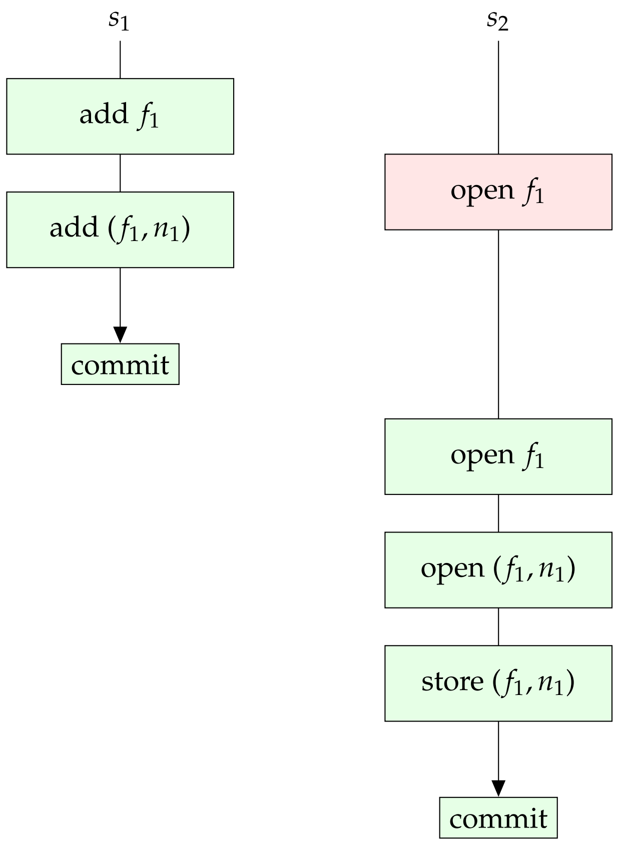

Figure 1: Simple example: s_1 adds objects, s_2 uses them

Observe that s_1 is not able to commit with full isolation, because s_2 is active. A commit is possible with almost full isolation, though, because s_2 has no open handles yet.

Scenario 2. *Assume that f_1 and (f_1, n_1) existed before s_1 and s_2 started. s_1 writes new content to (f_1, n_1) whereas f_2 deletes (f_1, n_1). Neither session is then able to commit with one of the safe weak isolation levels, as both hold handles on (f_1, b_1) and have journal entries recording changes to this blob (figure 2). Clearly, this indicates a deadlock which might be detected by the system to some extent. Note that such detection is not implemented at the moment.*

There are two possible solutions for this case: rollback or commit with no isolation. The latter solution leads to behaviour dependent on the order of the commit. If s_1 commits before s_2 (figure 3a), the new data of (s_1, n_1) is lost; otherwise (figure 3b), s_1 fails to commit because (s_1, n_1) does not exist any more, giving the user the chance to save the new data to another location. This behaviour is clearly a race condition*, which means that the unsafe isolation level should be avoided if possible.*

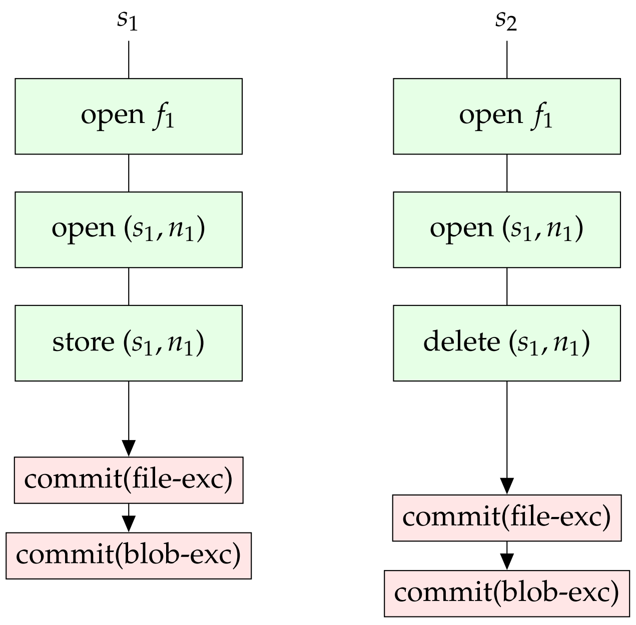

Figure 2: Competing changes: s_1 modifies a blob, s_2 deletes the same blob

Scenario 3. *Assume that f_1 existed before s_1 and s_2 started. The sessions s_1 and s_2 add the blobs (f_1, n_1) and (f_1, n_2) to f_1, respectively (where $n_1 \neq n_2$). s_1 is not able to succeed with the file exclusive level (see figure 4), because s_2 modified the same file as s_1. In that case, the blob exclusive level can be used which only requires that a blob must not have any competing changes, whereas the former attempt requires no competing changes on files.*

After s_1 committed, there are no other sessions and s_2 may choose an arbitrary level.

3.5 Formalization

In the following sections, we will introduce a formal definition of the session implementation of ANGELFISH. This definition consists of the *state* of the system and the four components of operational semantics:

1. preconditions and consequences of operations
2. conditions for isolation
3. condition for invalidation
4. preconditions and consequences of commits

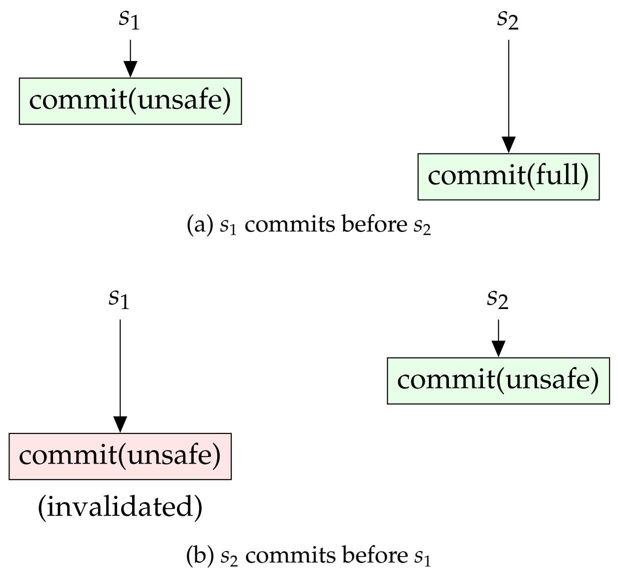

Figure 3: Race condition depending on which session commits first

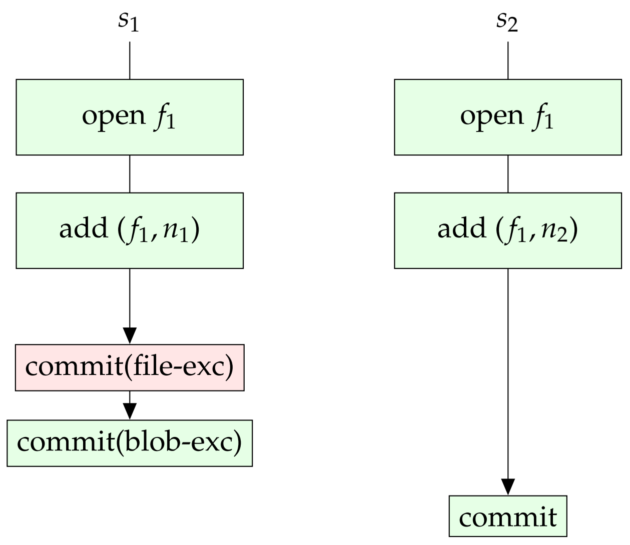

Figure 4: "Blob exclusive" example: s_1 and s_2 add blobs to the same file

3.5.1 Definitions

The state of the file system can be characterized by the following components:

- a set of sessions $\mathcal{S}$,
- sets of files $\mathcal{F} \subseteq \mathcal{F}_0$,
- a set of blobs $\mathcal{B} \subseteq \mathcal{F}_0 \times N \subseteq \mathcal{B}_0$,
- a set of RDF triples $\mathcal{R} \subseteq \mathcal{B}_0 \times (\mathcal{B}_0 \cup \{\bot\}) \subseteq \mathcal{R}_0$,
- a set of handles $\mathcal{H} \subseteq \mathcal{S} \times (\mathcal{F}_0 \cup \mathcal{B}_0)$ and
- sets of journal entries $\mathcal{J}_s \subseteq (\mathcal{F}_0 \cup \mathcal{B}_0 \cup \mathcal{R}_0) \times \{\texttt{insert}, \texttt{delete}\}$ together with total orderings $\leq_s$: $\mathcal{J}_s \times \mathcal{J}_s$ for each $s \in \mathcal{S}$

where N denotes the set of all strings according to section 2.3.1. Additionally, the sets $\mathcal{F}_0$, $\mathcal{B}_0$, $\mathcal{R}_0$, $\mathcal{S}$, $\mathcal{H}$, $\mathcal{J}_s$ and N have to be disjoint.

Rationale The sets for the objects (files, blobs and RDF triples) each have a corresponding superset (indexed with 0). Although this introduces more complexity in the definition, this is necessary to allow unsafe sessions – there might be handles on objects which have been removed in another session. So, these supersets can be seen as "universe sets" which contain every object which has been created, whereas the normal sets only contain the objects which are "active", i. e. have not been deleted. Note that it is still possible to re-create an object which has been deleted previously.

The set of RDF triples only contains pairs, as the predicate is not relevant for the further discussion. The symbol $\bot$ is used as a placeholder if the object of the triple is not a blob, where the particular object is again irrelevant.

The journal sets contain pairs of an object and an operation. It is possible for any given pair to distinguish which kind of object it is, as the universe sets are disjoint. Because of that, no further "tagging" is needed in the formalization.

Conventions The following shorthands and conventions are used throughout this section:

$\mathcal{T}$	...	the set of all possible states
M, M'	...	a set M before and after an operation, respectively
t_M	...	the set M in the state $t \in \mathcal{T}$, e. g. $t_{\mathcal{S}}$ for the set of sessions in the state t

Usually, the operations and conditions assume an implicit state t. For brevity, this t is often omitted and a set M is to be interpreted as t_M.

The operators "for all" and "there is" are used differently than their counterparts $\forall$ and $\exists$ in the following sections, because the former match on existing variables: Assuming that a is a fixed variable, the following equality holds:

$$\text{for all } (a,b) \in S : P(a,b) \Leftrightarrow \forall_{(a',b)\in S}\ a = a' \implies P(a,b)$$

and similarly for existential quantification.

3.5.2 Operations

We will now describe which state transitions are induced by the file system operations. For brevity, a free s in the second column means "in session s" in the first column. Furthermore, the following macros are used:

$$\begin{aligned} \mathit{append}(j) &= \leq_s \cup\{(j',j) | j' \in \mathcal{J}'_s\} \\ \mathit{invisible}(o) &= (s,o,\texttt{delete}) \in \mathcal{J} \vee \exists_{s'}\ s' \neq s \wedge (s',o,\texttt{insert}) \in \mathcal{J} \\ \mathit{unmodifiable}(o) &= (s,o,\texttt{delete}) \in \mathcal{J} \vee (s,o) \notin \mathcal{H} \end{aligned}$$

Table 3 specifies the transitions for all structural changes (opening, adding, deleting, storing).

Operation	**Preconditions/Consequences**
create empty file system	set all sets to $\emptyset$
start new session	$\mathcal{S}' = \mathcal{S} \cup \{s\}$ where $s \notin \mathcal{S}$
add new file	$\mathcal{F}' = \mathcal{F} \cup \{f\}$ where $f \notin \mathcal{F}_0$ $\mathcal{F}'_0 = \mathcal{F}_0 \cup \{f\}$ $j = (f, \texttt{insert})$ $\mathcal{J}'_s = \mathcal{J}_s \cup \{j\}$ $\leq'_s = \mathit{append}(j)$ open file f
open file f	fails iff $\mathit{invisible}(f)$ or $f \notin \mathcal{F}$ $\mathcal{H}' = \mathcal{H} \cup \{(s,f)\}$

(continued on next page)

Operation	**Preconditions/Consequences**
delete file f	fails iff $unmodifiable(f)$ $j = (f, \texttt{delete})$ $\mathcal{J}'_s = \mathcal{J}_s \cup \{j\}$ $\leq'_s = append(j)$ for all $(s, n) \in \mathcal{B}$ where $\neg$ $invisible((s, n))$: open blob (s, n) delete blob (s, n)
add new blob (f, n)	fails iff $(f, n) \in \mathcal{B}$ or $unmodifiable(f)$ $\mathcal{B}' = \mathcal{B} \cup \{(f, n)\}$ $\mathcal{B}'_0 = \mathcal{B}_0 \cup \{(f, n)\}$ $j = ((f, n), \texttt{insert})$ $\mathcal{J}'_s = \mathcal{J}_s \cup \{j\}$ $\leq'_s = append(j)$ open blob (f, n)
open blob (f, n)	fails iff $invisible((f, n))$ or $b \notin \mathcal{B}$ or $unmodifiable(f)$ $\mathcal{H}' = \mathcal{H} \cup \{(s, b)\}$
store blob b	fails iff $unmodifiable(b)$
delete blob (f, n)	fails iff $unmodifiable((f, n))$ or $unmodifiable(f)$ $j = ((f, n), \texttt{delete})$ $\mathcal{J}'_s = \mathcal{J}_s \cup \{j\}$ $\leq'_s = append(j)$ for all $r \in \mathcal{R}$ where $r_s(r) = b \vee r_o(r) = b$ and $\neg$ $invisible(r)$: $r_s(r) \neq b \implies$ open blob $r_s(r)$ $r_o(r) \notin \{b, \perp\} \implies$ oben blob $r_o(r)$ delete triple r
add RDF triple (b_s, b_o)	fails iff $(b_s, b_o) \in \mathcal{R}$ or $unmodifiable(b_s)$ $b_o \neq \perp \implies$ fails iff $unmodifiable(b_o)$ $\mathcal{R}' = \mathcal{R} \cup \{(b_s, b_o)\}$ $\mathcal{R}'_0 = \mathcal{R}_0 \cup \{(b_s, b_o)\}$ $j = ((b_s, b_o), \texttt{insert})$ $\mathcal{J}'_s = \mathcal{J}_s \cup \{j\}$ $\leq'_s = append(t)$

(continued on next page)

Operation	Preconditions/Consequences
remove RDF triple (b_s, b_o)	fails iff $invisible((b_s, b_o))$ or $(b_s, b_o) \notin \mathcal{R}$ or $unmodifiable(b_s)$
	$b_o \neq \perp \implies$ fails iff $unmodifiable(b_o)$
	$j = ((b_s, b_o), \texttt{delete})$
	$\mathcal{J}'_s = \mathcal{J}_s \cup \{j\}$
	$\leq'_s = append(j)$

Table 3: Operations

Rationale The general rule is that, in order to modify an object, handles on that object and on the parent objects are needed and the object itself has not yet been marked for deletion in this session. For blobs, the parent objects are files and for triples, the parent objects are blobs (they may have one or two). Files have no parent objects. An exception to the rule is that there are no handles on triples. These constraints are represented by the macro *unmodifiable* which is true if they do not hold.

When opening an object, this object must be visible, meaning that it has not been marked for deletion yet and no other session marked it for insertion. Opening an object yields a handle on it in the current session.

Both “create new file” and “start new session”, as indicated by the word “new”, do not take an identifier from the user, but generate a fresh, unused one.

The operation “store blob” only lists a precondition, because the blob contents are irrelevant for this discussion.

3.5.3 Isolation levels

The second step is to formally define the five different isolation levels. Assume a state of the system $t \in \mathcal{T}$ and a session $s \in t_{\mathcal{S}}$ which is about to commit. The table lists the levels $i \in I$ and the predicates $c_i \subseteq \{(s, t) \mid t \in \mathcal{T} \wedge s \in t_{\mathcal{S}}\}$ which must be met for this commit to be completed.

i	Description	Condition
i_{F}	Full	$\mathcal{S} = \{s\}$
i_{AF}	Almost full	for all $(s', o) \in \mathcal{H}$: $s' = s$

i_{FE}	File exclusive	for all $(s', o) \in \mathcal{H}$: $s' = s$ or $o \notin \mathcal{F}_0$ or none of the following: there is a $((o, n), p) \in \mathcal{J}_s$ with $(o, n) \in \mathcal{B}_0$ there is a $((b_s, b_o), p) \in \mathcal{J}_s$ and a $n \in N$ such that $b_s = (o, n) \vee b_o = (o, n)$
i_{BE}	Blob exclusive	for all $(s', o) \in \mathcal{H}$: $s' = s$ or $o \notin \mathcal{B}_0$ or none of the following: there is a $(o, p) \in \mathcal{J}_s$ there is a $((b_s, b_o), p) \in \mathcal{J}_s$ such that $b_s = o \vee b_o = o$
i_{NI}	No isolation	`true`

3.5.4 Invalidated sessions

The third step is to define under which circumstances a session invalidated another session, i. e. a session contains journal entries which refer to objects which do not exist any more.

$$\begin{aligned}
&\mathit{invalid}_t(s) \Leftrightarrow \\
&\qquad \text{there is a } ((f, n), \texttt{update}) \in \mathcal{J}_s \text{ such that } f \in \mathcal{F}_0 \setminus \mathcal{F} \vee (f, n) \in \mathcal{B}_0 \setminus \mathcal{B} \\
&\quad \text{or } \text{there is a } ((f, n), \texttt{insert}) \in \mathcal{J}_s \text{ such that } f \in \mathcal{F}_0 \setminus \mathcal{F} \\
&\quad \text{or } \text{there is a } ((b_s, b_o), \texttt{insert}) \in \mathcal{J}_s \text{ such that } b_s \in \mathcal{B}_0 \setminus \mathcal{B} \vee b_o \in \mathcal{B}_0 \setminus \mathcal{B}
\end{aligned}$$

3.5.5 Commit

Let $\mathit{trans}_t : \mathcal{J} \to \mathcal{T}$ be the function which performs a state transformation specified by the table below, and removes j from $\mathcal{J}_s$ and $\leq_s$:

Journal entry		**Operation**
$(f, \texttt{delete})$	$f \in \mathcal{F}_0$	$\mathcal{F}' = \mathcal{F} \setminus \{f\}$
$(b, \texttt{delete})$	$b \in \mathcal{B}_0$	$\mathcal{B}' = \mathcal{B} \setminus \{b\}$
$(r, \texttt{delete})$	$r \in \mathcal{R}_0$	$\mathcal{R}' = \mathcal{R} \setminus \{r\}$

Entries carrying an `insert` operation are ignored, i. e. *trans* just removes these entries.

The following functions are used to pick the smallest (i. e. oldest) journal entry and to perform a sequence of journal entries, respectively:

$$pick_t(s) = \begin{cases} j & \text{if } \exists_{j \in \mathcal{J}_s} \forall_{j' \in \mathcal{J}_s}\, j \leq_s j' \\ \bot & \text{otherwise} \end{cases}$$

$$commit(t,s) = \begin{cases} t & \text{if } pick_t(s) = \bot \\ commit(trans_t(pick_t(s)), s) & \text{otherwise} \end{cases}$$

To sum everything up, the (partial) function C represents the whole commit process:

$$C(t,s,i) = \begin{cases} commit(t,s) & \text{if } \neg invalid_t(s) \wedge (s,t) \in c_i \\ \bot & \text{otherwise} \end{cases}$$

A result of $\bot$ indicates that s could not be committed, because another session invalidated s or the specified isolation level i is too strict. In the first case, the transaction will not be able to be committed at a future time. In the latter case, a commit is possible using a weaker level.

Obviously, after a commit the corresponding session is removed from $\mathcal{S}$. This also frees the corresponding handles from $\mathcal{H}$.

3.5.6 Soundness

We will now show selected soundness proofs, which state some of the informal guarantees described earlier. All proofs assume a non-empty state containing at least a single file $f_0 \in \mathcal{F}$, but no handles, journal entries or other sessions.

Theorem 1. *The sessions s_1 and s_2 add the blobs b_1 and b_2 to f_0, respectively (where $b_1 \neq b_2$). s_1 is not able to succeed with i_{FE} (file exclusive), but with i_{BE} (blob exclusive). This corresponds exactly to the scenario 3 (page 18). In formal terms:*

$$C(t, s_1, i_{FE}) = \bot \wedge C(t, s_1, i_{BE}) = t' \textit{ where } t' \neq \bot \wedge b_1 \in t'_{\mathcal{B}}$$

Proof. The the state transformations performed by s_1 and s_2 are traced in table 4. We can see that s_1 is not able to commit in file exclusive mode, because it had pending operations on a file which was opened by another session[9]. □

[9]Note that, although similar, "because it opened files which had pending operations by another session" would also be true here, but represents different behaviour.

Session	**Operation**	**Consequences**
s_1	open file f_0	$\mathcal{H} \longleftarrow \mathcal{H} \cup \{(s_1, f_0)\}$
s_2	open file f_0	$\mathcal{H} \longleftarrow \mathcal{H} \cup \{(s_2, f_0)\}$
s_1	add blob $b_1 = (f_0, n_1)$	$\mathcal{B} \longleftarrow \mathcal{B} \cup \{(f_0, n_1)\}$ $\mathcal{B}_0 \longleftarrow \mathcal{B}_0 \cup \{(f_0, n_1)\}$ $j = ((f_0, n_1), \texttt{insert})$ $\mathcal{J}_{s_1} \longleftarrow \mathcal{J}_{s_1} \cup \{j\}$ $\leq_{s_1} \longleftarrow \mathit{append}(j)$
s_2	add blob $b_2 = (f_0, n_2)$	$\mathcal{B} \longleftarrow \mathcal{B} \cup \{(f_0, n_2)\}$ $\mathcal{B}_0 \longleftarrow \mathcal{B}_0 \cup \{(f_0, n_2)\}$ $j = ((f_0, n_2), \texttt{insert})$ $\mathcal{J}_{s_2} \longleftarrow \mathcal{J}_{s_2} \cup \{j\}$ $\leq_{s_2} \longleftarrow \mathit{append}(j)$
s_1	commit with i_{FE}	check whether $(s_1, t) \in i_{\text{FE}}$: $(s_1, f_0) \in \mathcal{H}$: $s_1 = s_1 \ldots$ ✓ $(s_2, f_0) \in \mathcal{H}$: $((f_0, n_1), \texttt{insert}) \in \mathcal{J}_{s_1} \ldots$ $\lightning$
s_1	commit with i_{BE}	check whether $(s_1, t) \in i_{\text{BE}}$: $(s_1, f_0) \in \mathcal{H}$: $s_1 = s_1 \ldots$ ✓ $(s_2, f_0) \in \mathcal{H}$: there is no $(f_0, p) \in \mathcal{J}_{s_1} \ldots$ ✓ there is no $((b_s, b_o), p) \in \mathcal{J}_{s_1}$ such that $b_s = f_0 \vee b_o = f_0 \ldots$ ✓ session is valid because no deletions occured $\mathcal{H} \longleftarrow \mathcal{H} \setminus \{(s_1, f_0)\}$ $\mathcal{S} \longleftarrow \mathcal{S} \setminus \{s_1\}$
s_2	commit with arbitrary i	succeeds because no other session exists $\mathcal{H} \longleftarrow \mathcal{H} \setminus \{(s_2, f_0)\}$ $\mathcal{S} \longleftarrow \mathcal{S} \setminus \{s_2\}$

Table 4: Trace of the operations of theorem 1

Theorem 2. *If all sessions are committed with i_F (full), there must not be more than one session at a time.*

Proof. By contradiction. Assume there is a state t with $|t_\mathcal{S}| > 1$. Committing any s fails because $t_\mathcal{S} \neq \{s\}$ (by definition of full isolation). $\square$

Corollary 3. *Always committing with i_F ensures ACID properties (with the durability assumption that no failures occur).*

Theorem 4. *If all sessions are committed with i_{AF} (almost full), no sessions will be invalidated.*

Proof. A session gets invalidated when a file or a blob is deleted on which this session has handles. Assume s_1 has a handle on f_0 and s_2 tries to delete f_0 using i_{AF}. This will not succeed because $(s_1, f_0) \in \mathcal{H}$, but $s_1 \neq s_2$ (by definition of almost full isolation). $\square$

4 Architecture and implementation

In this section, the architecture of Angelfish is described. First, the general concepts and terminology are discussed. In the ensuing section, the two core components of the storage backend – namely the database and the target – are presented. The last section introduces the underlying modular architecture, followed by a description of the environment classes. An user guide, including instructions to build an run the software, is provided in the appendix.

4.1 General concepts

The scope of this thesis is not to implement a disk-based associative file system, but rather to provide an API and a CLI for dealing with files in an associative manner. As such, it still qualifies as "file system", although it does not come with own mechanisms to handle disk access or other low-level concepts – the actual storage is left to the target storing the blobs and to the database storing the structure and the metadata. It can be roughly compared to FUSE [10] which allows userspace applications to provide a virtual file system. However, the major difference is that FUSE assumes hierarchic storage which is necessary to interoperate with the kernel file system layer, but complicates the applicability for non-hierarchic systems.

Operating environment Core parts of the system are platform-independent as they only use the C++ standard library and Boost libraries. A major restriction on the platforms is imposed by HyPer, but this module is optional and may be left out during compilation (see section A.1.2 for details). Also, the system uses shared memory, memory locks and file locks which have to be supported by the platform. In general, an Unix-like environment should be sufficient to compile and run Angelfish. It is known to work under GNU/Linux without any problems.

System integration An important design decision was to ignore the kernel or any other standard file system providers and to develop an independent API. A read or write request to a blob is therefore redirected to a path on the *target* (see section 4.2.2) which is simply a file on a traditional file system. Angelfish-aware Applications may use their favourite way to deal with such a file like they would for any traditional file, assuming they do not change the target directly but indicate that via appropriate API calls. Non-aware applications may still access the blobs, but have to use read-only mode and are not able to use the metadata.

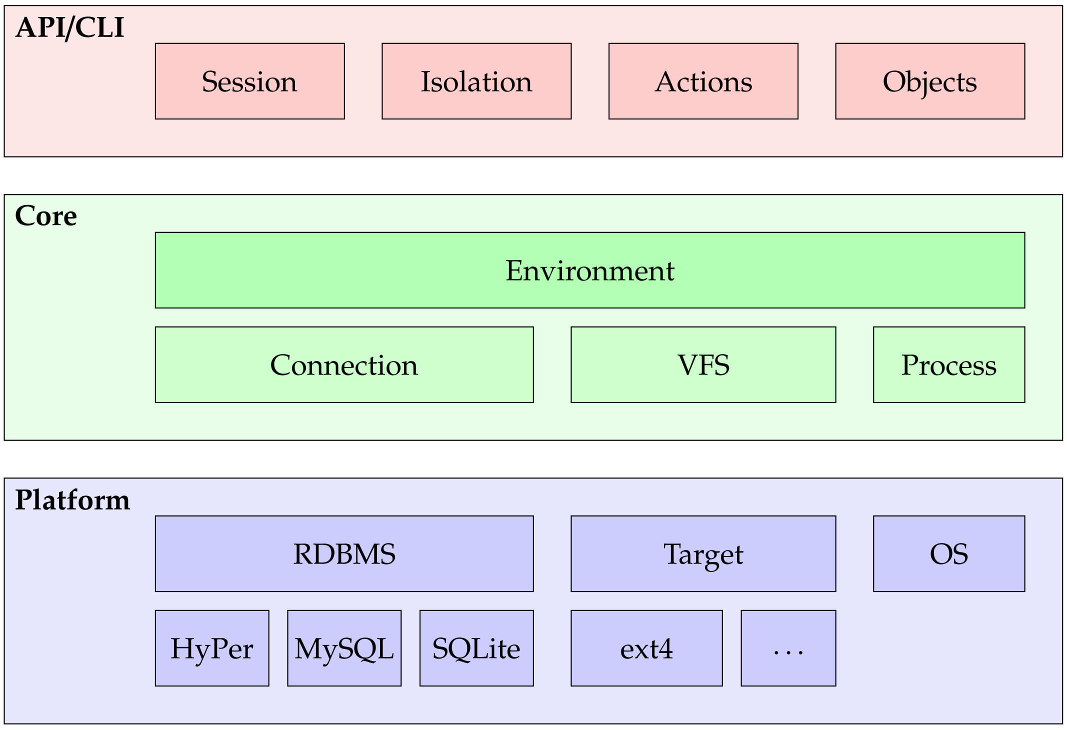

Figure 5: Overview of Angelfish's architecture

4.2 Storage backend

The storage reflects the general data model of Angelfish, which requires a strict separation between data and metadata. The former is stored on an existing file system, whereas the latter is managed in a database.

4.2.1 Database

Large parts of the infrastructure can be easily represented as relational data. Hence, it is a reasonable choice to build on top of that model. The planned system to use was *HyPer* (short for *Hybrid OLTP & OLAP High-Performance Database System*), a memory-based relational database system developed at a research group headed by Prof. Kemper and Prof. Neumann of TU München. An overview of the functionality and architecture of HyPer can be obtained from [13]. As it turned out during development of the thesis, it is sensible to support other database systems as well.

There is one drawback of that approach, because RDF data can be modelled in a relational way, but this is not the most efficient possibility. There are sophisticated efforts to manage RDF tuples in specialized *triplestores*, e. g. *RDF-3X* [14]. However, in the case of this file system, the relational model is fast enough. As such, all triples are stored in a single table. There is also no support for RDF query languages like *SPARQL*.

As one possible workflow of the file system is the repeated invocation of the command line executable, buffering of data is nearly impossible and should be left to the database system. Thus, it is assumed that the database server runs on the same machine, or has a fast connection to the machine which runs the file system to avoid delays for repeated queries[10]. Except for the embedded SQLite driver, Angelfish does not start or manage the database server in any way; this is expected to be done when the operating system boots or left for the user.

4.2.2 Target

As stated in the section above, blobs are not stored in the database, but in an existing file system. It would be possible – but not feasible – to do otherwise, because Angelfish is optimized for memory based database systems and the size of a multimedia library usually exceeds the amount of available memory by far.

Therefore, the system relies on the existence of a *target* storage – a place, where blobs are stored as regular files on a traditional file system. The target consists of multiple subdirectories[11] and files:

`blobs` This folder contains folders corresponding to the file objects, which are created as soon as content is to be written to a blob for the first time. This means that the contents of this folder are not an index of all existing file objects. For every blob to which content has already been written yet, there is a file with the same name as the blob in the folder for the file object. Example: The file `blobs/d7da770e-0066-4a20-b986-860174b49d02/default` contains the content of the blob "default" of the file object with the UUID `d7d...` Only committed changes are visible in this folder.

`temp` To ensure atomicity for operations on the target, files to be written resp. removed are copied resp. moved to this folder beforehand. In opposition to what the name "temp" suggests, it is not safe to remove files in this folder at

[10] A small amount of buffering is done, mostly to simplify some operations from an implementation point of view and to support fast rollbacks.

[11] to distinguish between "files" of the target and "files" of the associative file system, the latter ones are called "file objects"

any time as they may have to survive crashes or reboots in order to prevent data loss. Files in this folder do not follow any naming convention. It is important that this folder is located on the same physical file system as `blobs`, as otherwise atomicity of a rename is not possible.

`lock` This file is empty and is solely used for locking purposes.

The user should choose a file system as target which does not impose a limit of subfolders per folder.

4.3 Implementation

The ANGELFISH library consists of multiple subsystems which are reflected in the source folder structure:

`db` To decouple the application from a specific database systems, this subsystem provides an abstraction interface which allows database interaction through *providers*. The interface offers non-typesafe (i. e. string-based) querying, prepared queries, transactions and buffering.

`objects` This part contains data-centric classes for the different types of file system objects, e. g. files, blobs and RDF triples.

`isolation` A modular interface for isolations, and the implementations thereof.

`env` The classes abstracting from the underlying systems are placed in this subsystem. Instances of those environmental classes are needed throughout the API and are usually created right after the programme starts. The most important class is `Environment` which provides access to sessions and files.

`util` A rich set of utility classes, heavily depending on the Boost C++ library, for exception handling, high-level access to compile time configuration options, logging, module loading, I/O, general runtime parameters and resource abstractions.

`action` This subsystem is only relevant for command line access as it provides the different utilities to interact with the user, e. g. a shell. Builds as a separate executable.

`test` A set of test cases and related infrastructure built upon the Google Test library. Builds as a separate executable.

4.3.1 Module infrastructure

Rationale The development started without the prospects of a modular architecture in mind. The original design goals were to use HyPer as the only database provider, thus not requiring a generic interface. Code querying the database directly used the dbcore API. However, the need for such a generic interface and therefore a module system arose quickly, for a couple of reasons:

- Different database providers increase portability. HyPer only runs on Unix-like systems, which makes it impossible to port the whole application to other operating systems.
- Making it easy to add other database providers is a trade-off between optimized, database-specific code and generic, independent code. Usually, the latter one is preferred due to greatly increased maintenance. This will also allow to stay independent from major changes in HyPer.
- During development, it turned out that the SQL interface of HyPer is not stable enough for the type of queries used. In particular, prepared queries cause HyPer to crash. Thus, a generic interface increases the stability of the whole system.
- The issue of allowing multiple database providers can be generalized to providing multiple implementations of an interface which may be selected during runtime.

In summary, such an interface increases the flexibility as the particular DBMS can be chosen at installation or even replaced at a later point of time. The side effect of adding a module infrastructure is that other subsystems may also be modularized.

At the moment, there are three interfaces employing that infrastructure: database providers, command line actions and isolation levels.

Concepts As mentioned above, the main concept is an interface, which may be an arbitrary C++ class deriving from `Module`, usually with some virtual methods declared. A *module* is an instance of a class deriving from that interface. By convention, there is only one module per deriving class and no inheritance relation between modules (except that they have a common ancestor). To work without having to define own routines for creating modules from the corresponding classes, those classes should declare a constructor without parameters. Each module has to specify a name which is unique among modules implementing the same interface and does not change during execution.

In order to actually gain access to all modules of a certain interface, the interface should declare a static instance of the `ModuleManager` class, which takes care of initialization. It has a method called `modules` which lazily generates modules out of classes and returns them.

Implementation The implementation details are in the source files `modules.cpp` and `modules.cpp` and in the build scripts located in the `cmake` folder. Despite what the name "module" might suggest, all modules must be compiled together with the interface. At the moment, no dynamic module loading is in place – it was no requirement – but that should not be too hard to implement.

Basically, there are two preprocessor macros, `MODULE_DECL` and `MODULE_DEF`. The first is used by the declaration of a module class and creates a private static field used for initialization purposes. The latter one should be put outside of the class, but still inside the same compilation unit. It is responsible for registering a factory method with the module manager of the parent interface. It also defines a function referencing the generated static field, which is necessary because the compiler or linker might otherwise elide the field together with the initialization code.

During build configuration, a shell script scans for the usage of the `MODULE_DEF` macro in `*.cpp` files. All those occurrences are gathered into another macro, e. g. `DB_INIT`. This macro is written to a separate header file.

To glue all pieces together, the interface should declare a public static method which is called by users of that interface to retrieve a module by name. The implementation of this method may simply use the macro `DB_INIT` as the first line, without any parameters. This calls the generated functions, which forces the compiler to do static initialization, but has no effect at runtime.

A short introduction on how to define new modules is given in section A.2.

Discussion Obviously, this system heavily relies on code generation and preprocessor macros, which makes it easy to extend existing structures without much boilerplate code. Other options without code generation have been considered, such as requiring the interface to list all of its modules, which would also solve the static initialization problem but violates the software engineering principle "Don't Repeat Yourself". Clearly, the solution which imposes a minimal amount of redundancy should be preferred, although it requires a sophisticated interaction between code generated by scripts and by the preprocessor. From an implementer's point of view, these details are mostly hidden.

Another advantage is the high flexibility. It is quite easy to retrieve a module by it's name, but this cannot be checked by the compiler: Assume that a piece of code wants to pick a specific module. Usually, it would call a static method on the interface, providing the name as a string. This may fail at runtime, if the string is misspelt. A superior approach would be to pick a module using a compile time constant, such that mistakes can be checked by the compiler.

Such a mechanism is actually in place and used by the isolation level interface. It is just a matter of generating an additional class which holds references to all modules as static fields. The test cases which check fixed isolation levels retrieve them via those fields, resulting in code which is less likely to fail at runtime.

4.3.2 Database providers

At the moment, the user can (effectively) choose between two different database systems: SQLite and MySQL. The third one, HyPer, should be mostly functional but is untested as its SQL interface was not stable enough at the time of development.

Most database interaction will take place inside of transactions. Therefore, the interface requires the two methods `startTransaction()` and `endTransaction(bool)`, where the argument denotes whether a commit or a rollback should be performed.

On a higher level, there is the `transaction()` method which follows the *Resource Acquisition Is Initialization* (short: *RAII*) idiom introduced by the C++ designer Bjarne Stroustrup. In short, it returns an object representing an unfinished transaction. A typical interaction would be:

```
auto t = conn->transaction();
auto result = conn->executeQuery("select * from relation");
// ...
conn->executeStatement("insert into relation values (0)");
t->commit();
```

The advantage is that if one of the operations between starting and ending the transaction throws an exception, the destructor of `t` will be called. In that case, the transaction will be rolled back automatically.

4.3.3 Environment

The environment subsystem provides high-level wrappers for many aspects of the platform.

The central class is `Environment` which is the main access point for application developers. The provided operations are:

- start a session (only one concurrent session per object allowed)
- add new file
- retrieve existing file
- commit or roll back the active session

Multiple sessions are supported by creating new instances of `Environment` as needed.

To construct an object of this class, additionally to the database provider one object of each of the following classes is needed.

Process The `Process` class is responsible for managing locks and communication between different instances of ANGELFISH sharing the same data source. Objects may request memory and file locks to synchronize non-atomic operations.

VFS This class manages the details of the target file system which is abstracted to a *VFS* (short for *virtual file system*) layer. It implements transactions which adds an additional stage after a commit which allows the system to roll back all changes. This becomes necessary because a commit of a session is actually a procedure with three stages:

1. precondition checks (isolation, invalidation)
2. mutate target
3. mutate database

If the last of those steps fails, the second-to-last step has to be undone in order to leave a consistent state. The problem faced during implementation was that, once committed, a database transaction cannot be undone. However, it is possible to implement the operations on the target in such a way that removed or overwritten blobs are first moved to a temporary area from which they can be restored if necessary. In conclusion, the revised order of the commit procedure can be defined as follows:

1. precondition checks (isolation, invalidation)
2. mutate target
3. non-final commit target
4. mutate database
5. commit database; if failed: undo target and exit
6. finish target

The implementation of this behaviour is straightforward. Every time a blob is updated, added or deleted, the corresponding objects adds a journal entry (and asks the VFS for a temporary file, if necessary), rather than mutating the target store directly. Hence, during a session, the target does not change at all (excluding the temporary area which contains all modifications). As soon as the user requests a commit, the VFS gathers the necessary information from the journal and performs the actual operations *without* overwriting or deleting any existing data which is moved to the temporary area prior to that. Now, the low-level database transaction is allowed to run. If that succeeds, the temporary are can be cleared safely. Otherwise, the inverse operations are applied, leaving the target in the same state as before the commit.

5 Evaluation

In this section, the progress and the results of the work are discussed. After that, prospects for future work are presented briefly.

5.1 Results

The result of the thesis is a system which allows to manage files folder-less, based on the metadata. The implementation is in the prototype stage, which means that it is working, reasonably well tested, but not feature complete. There is support for sessions which has been implemented on top of existing database transactions and a newly written mechanism to support undoable low-level file system transactions. The semantics of these high-level sessions have been formalized and it was shown that it is possible to prove soundness theorems using that calculus.

The software is mostly stable: due to the operating environment, many operations may fail, but in such a case, no data is destroyed. Blob contents are never deleted or overwritten unless the user explicitly requested a commit and all underlying operations succeeded. Granted, the user interface may need some polishing with respect to error tolerance and verbosity.

It has been made possible to extend the application by plugging in other database providers, actions or isolation levels. This proved to be useful during development and should be one of the building blocks of future reusability.

The database schema has been kept simple which eases future modifications. Also, the RDF model ensures that data is not "locked in" and can be ported to other applications if desired.

In summary, the goals of the projects have mostly been reached, although session support turned out to be a second, non-trivial task besides metadata modelling.

5.2 Future work

There are great opportunities to improve the prototype with additional features, in increasing order of difficulty:

- plugins which allow "virtual" blobs which are computed on request; enabling e. g. on-the-fly conversion from one audio format to another one
- plugins which automatically extract metadata from existing formats such as *ID3* or *EXIF*

- support for a specialized query language (or SPARQL) to filter blobs based on custom predicates
- define a mapping to the hierarchic model to allow third party applications to access the data
- extend the scope of the file system to include an actual disk layout

A Appendix

A.1 User guide

During the development of ANGELFISH, an user interface had to be developed which instruments the API in the most generic manner possible. Therefore, it comes with a command line interface which uses an established concept from version control systems: One single executable providing multiple sub-commands. In the following, let `fs` be the name of this executable.

A.1.1 General usage

An invocation of `fs` consists of the environment parameters (data source, target directory and session isolation level) and a sub-command (*action*)[12]. Example:

```
fs --data-source=sqlite3:db.sqlite --target=/media/fs
   --isolation-level=full create
```

In this case, the `create` action creates a new file with a random UUID. Of course, some actions may take parameters itself (global parameters omitted for brevity):

```
fs ls --uuid=a1740f38-cd1b-430a-8f2a-923ba1ada2ef
```

which would list all blobs in the specified file.

One single invocation of `fs` corresponds to exactly one session on file system level. Thus, a new session is opened – regardless of other open sessions – before the execution of the desired sub-command, and committed afterwards. Except when the commit fails, the user is not able to rollback an operation.

Because this "no user rollback" behaviour is not always desired, there is a "wrapper" sub-command called `shell`. This also opens a new session, but allows the user to issue multiple sub-commands with a fine-grained control over when a commit or rollback happens. In the shell, an action invocation behaves exactly like in the top-level[13], such that the sequence

```
fs shell
> ls --uuid=a1740f38-cd1b-430a-8f2a-923ba1ada2ef
```

has the same consequences as the direct invocation above.

[12] `list-actions` will list all available actions

[13] with the minor difference that the environment parameters cannot be changed during one shell session

A.1.2 Build

Preliminaries Before building, make sure to have the following libraries installed (including headers):

- Boost `http://www.boost.org/`
- SQLite 3 `http://sqlite.org/` (for build with SQLite support only)
- MySQL `http://www.mysql.com/` (for build with MySQL support only)
- MySQL Connector/C++ `http://www.mysql.com/downloads/connector/cpp/` (for build with MySQL support only)
- Log4cpp `http://log4cpp.sourceforge.net/`
- Google Test `http://code.google.com/p/googletest/` (for test build only)
- LLVM `http://llvm.org/` (for build with HyPer support only)

It is expected that the headers and the libraries are in the respective search paths of compiler and linker. For configuration, CMake in version 2.6.4 (or higher) is used. For compiling, a current gcc (version 4.6.0 or higher) is needed. SQLite and MySQL support are enabled by default, whereas HyPer support is disabled.

Compile Obtain the current snapshot by cloning the author's Git repository from `http://home.in.tum.de/~hupel/git/associative.git`.

Now, switch to the folder `fs` and issue the following commands:

```
mkdir build
cd build
cmake ../src
make fs-main
```

If desired, the tests can be built using the `fs-test` target. If no target is specified, both the main executable and the tests are built.

HyPer support Right after cloning the repository, run `git submodule update` in the working directory of your clone[14]. Invoke `make` in the subfolders `fs/lib/dbcore` and `fs/lib/hyper` (in this order). Proceed as described above, but additionally pass the parameter `-DASSOCIATIVE_WITH_HYPER=1` to `cmake`.

[14]requires shell access to the `dbkemper` servers

Further configuration options The CMake build script accepts a number of parameters which are defined in the file `fs/src/cmake/config.cmake`. To change one of those, add a command line argument of the form `-D<param>=<value>` to the invocation of `cmake` above. Note that it is possible to re-run `cmake` in the `build` directory, using `.` instead of `../src` as argument. In most cases, a full rebuild is required for changes to take effect.

All available parameters are listed in table 5.

Parameter	**Effect**
`CMAKE_BUILD_TYPE`	Changes compiler flags and verbosity with respect to exception messages and other internals. **Values:** `Debug` or `Release` **Default:** `Release`
`ASSOCIATIVE_MAX_LOCK_TIME`	Specifies the timeout in seconds all locking operations use. This behaviour cannot be switched off (not even by using 0 seconds or negative values as timeout) in order to prevent deadlocks. **Values:** any whole number greater than 0 **Default:** 10
`ASSOCIATIVE_DEFAULT_ISOLEVEL`	If no isolation level is specified on command line, use that instead. **Values:** `unsafe`, `blob-exclusive`, `file-exclusive`, `almost-full`, `full` **Default:** `almost-full`
`ASSOCIATIVE_DEFAULT_LOG`	If no log file is specified on command line, use that instead. **Values:** valid paths on existing file systems **Default:** `/var/local/log/associative-fs`
`ASSOCIATIVE_WITH_SQLITE` `ASSOCIATIVE_WITH_MYSQL`	Enable SQLite or MySQL support, respectively. If enabled, the database providers `sqlite3` and `mysql` are available. **Values:** `ON` and `OFF` **Default:** `ON`
`ASSOCIATIVE_WITH_HYPER`	Enable HyPer support. If enabled, the database provider `hyper` is available. **Values:** `ON` and `OFF` **Default:** `OFF`

Table 5: Available configuration options

A.2 Interface and module definition

Creating a new interface and modules for it is straightforward. If, at a later time, customizable hooks should be added to the library, this may be realized in these steps:

1. In the file `hooks/hook.hpp`, declare a new class which derives from `Module`: `class Hook : public Module`.
2. Add a static `ModuleManager` instance to the class, parametrized on the class itself: `static ModuleManager<Hook> manager`.
3. Also add a public static method for clients to retrieve the hooks. Make sure that the implementation of this method uses the macro `HOOK_INIT`.
4. Create a new folder for the module classes, e. g. `hooks/impl`. Add a corresponding line to the build script `cmake/src.cmake`.
5. Implement modules using the following skeleton:

```
class CreateHook : public Hook
{
    MODULE_DECL;
}

MODULE_DEF(Create, Hook, Hook);
```

Note that these steps are just meant as an overview and might be slightly adapted for the particular use case.

A.3 SQL schema

A.3.1 Schema

```
create table pool (
  id integer not null,
  name varchar(64) not null, -- unique
  primary key (id)
);

create table file (
  id integer not null,
  uuid varchar(36) not null, -- unique
  pool_id integer not null, -- references pool (id)
  visible smallint not null,
  primary key (id, pool_id)
);

create table content_type (
  id integer not null,
  mime varchar(64) not null, -- unique
  primary key (id)
);

create table "blob" (
  id integer not null,
  file_id integer not null, -- references file (id)
  name varchar(256) not null,
  content_type_id integer not null, -- references
     content_type (id)
  visible smallint not null,
  primary key (id, file_id)
  -- unique (file_id, name)
);

create table prefix (
  id integer not null,
  name varchar(64) not null, -- unique
  uri varchar(256) not null,
  primary key (id)
);

create table type (
  id integer not null,
  prefix_id integer not null,
  name varchar(64) not null
  -- unique (prefix_id, name)
);

create table metadata (
  id integer not null,
  blob_id integer not null, -- references blob (id)
  predicate_prefix_id integer not null, -- references prefix
      (id)
  predicate varchar(256) not null,
  object_type_id integer, -- references type (id)
  object varchar(256) not null,
  visible smallint not null,
  primary key (id)
);

create table handle (
  id integer not null,
  relation integer not null,
  relation_id integer not null,
  session_id integer not null -- references session (id)
);

create table "session" (
  id integer not null,
  ready smallint not null,
  process integer not null
);
```

```
create table journal (
  id integer not null,
  session_id integer not null, -- references session (id)
  relation integer not null,
  relation_id integer not null,
  operation integer not null,
  target varchar(256),
  executed smallint not null,
  primary key (id)
);

create table ids (
  id integer not null,
  table_name varchar(32) not null, -- unique
  next_id integer not null,
  primary key (id)
);
```

A.3.2 Required initial data

```
insert into prefix values (0, 'system', '#');
insert into type values (0, 0, 'blob');

insert into ids values (0, 'prefix', 1);
insert into ids values (1, 'type', 1);
```

References

[1] freedesktop.org: "Guidelines for extended attributes", http://www.freedesktop.org/wiki/CommonExtendedAttributes?action=recall&rev=12, last revision from August 21, 2009, date retrieved: September 12, 2011

[2] Amarok Wiki: "Amarok File Tracking", http://amarok.kde.org/amarokwiki/index.php?title=Amarok_File_Tracking&oldid=16054, last revision from October 11, 2010, date retrieved: September 12, 2011

[3] Theo Haerder, Andreas Reuter: "Principles of Transaction-Oriented Database Recovery", 1983, http://web.archive.org/web/20041227230914/http://www.minet.uni-jena.de/dbis/lehre/ss2004/dbs2/HaerderReuter83.pdf

[4] MSDN library: "Transactional NTFS (TxF)", http://msdn.microsoft.com/en-us/library/bb968806(VS.85).aspx, date retrieved: May 21, 2011

[5] MSDN library: "File Streams", http://msdn.microsoft.com/en-us/library/aa364404(VS.85).aspx, date retrieved: September 12, 2011

[6] MSDN: "What's in Store – WinFS Team Blog", http://blogs.msdn.com/b/winfs/, last update June 2006, date retrieved: September 13, 2011

[7] Git man pages: "gitglossary(7)", http://www.kernel.org/pub/software/scm/git/docs/gitglossary.html, date retrieved: May 27, 2011

[8] KDE 4.7 API Reference: "KIO: Network-enabled File Management", http://api.kde.org/4.x-api/kdelibs-apidocs/kio/html/index.html, date retrieved: May 27, 2011

[9] World Wide Web Consortium: "RDF – Semantic Web Standards", http://www.w3.org/RDF/, date retrieved: May 27, 2011 (permanent link: http://www.w3.org/2001/sw/wiki/index.php?title=RDF&oldid=1517)

[10] Filesystem in Userspace, http://fuse.sourceforge.net/, date retrieved: May 27, 2011

[11] RFC 4122: "A Universally Unique IDentifier (UUID) URN Namespace", http://tools.ietf.org/html/rfc4122, date retrieved: May 27, 2011

[12] RFC 2046: "Multipurpose Internet Mail Extensions (MIME), Part Two: Media Types", `http://tools.ietf.org/html/rfc2046`, date retrieved: September 12, 2011

[13] Alfons Kemper, Thomas Neumann: "HyPer – Hybrid OLTP & OLAP High Performance Database System", Technical Report, TUM-I1010, May 19, 2010, `http://www-db.in.tum.de/research/projects/HyPer/HyperTechReport.pdf`

[14] Thomas Neumann, Gerhard Weikum: "The RDF-3X Engine for Scalable Management of RDF Data", MPI-I-2009-5-003, March 2009

[15] ISO/IEC 9899:1999 (C99), `http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1256.pdf`, date retrieved: May 27, 2011

[16] ISO/IEC 9075:1992 (SQL92), `http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt`, date retrieved: September 13, 2011

[17] Filesystem Hierarchy Standard, Version 2.3, released January 29, 2004, `http://refspecs.linuxfoundation.org/FHS_2.3/fhs-2.3.html`, date retrieved: August 24, 2011

[18] Alfons Kemper, André Eickler: "Datenbanksysteme", 2006, 6. Auflage, Oldenbourg Wissenschaftsverlag

[19] Andrew S. Tanenbaum: "Modern Operating Systems", 2009, Third Edition, Pearson Education