Design and Implementation of a Transaction-Based Filesystem on FreeBSD

Jason Evans
The Hungry Programmers
jasone@hungry.com, http://www.hungry.com/~jasone

Abstract

Transactional database management systems (DBMS's) have special data integrity requirements that standard filesystems such as the Berkeley Fast Filesystem do not address. This paper briefly describes the requirements a transactional DBMS makes of a transaction-based filesystem, then goes on to describe the design and implementation of such a filesystem, referred to as a *block repository*[1], which is part of the SQRL DBMS project.

The implementation of SQRL's *block repository* is different than most traditional filesystems in that it is purposely implemented in user-land using raw devices and threads. Its performance is more tunable to the needs of transaction processing than would be the case if it were integrated into the kernel.

1 Introduction

Transactional database management systems go to great lengths to never lose or corrupt data, even in cases of unexpected system failure. Algorithms that achieve atomic writes of data stored on disk are complex and can be very slow, depending on what support is available from the underlying filesystem. Traditional filesystems such as the Berkeley Fast Filesystem (FFS) guarantee atomic updates of filesystem metadata in order to avoid filesystem corruption caused by system failures, but no atomicity guarantees are made for file writes.[2] This means that in order to avoid possible file corruption, programmers of transaction-based applications have to do extra work to make atomic changes to files.

One of the simplest, though not most efficient, methods of implementing atomic writes on FFS is to use triple redundancy:

```
a = open("A", O_RDWR);
b = open("B", O_RDWR);
c = open("C", O_RDWR);
...
[Write changes to A.]
fsync(a);
[Write changes to B.]
fsync(b);
[Write changes to C.]
fsync(c);
...
```

If there is a system failure during writing, there are only three possible inconsistent states, all of which are fixable:

1. $A \neq B = C$. Fix: $A \leftarrow B$.
2. $A \neq B \neq C$. Fix: $B \leftarrow A$, $C \leftarrow A$.
3. $A = B \neq C$. Fix: $C \leftarrow B$.

A more practical implementation of atomic writes on FFS takes advantage of atomic metadata updates:

1. Definitions for emphasized words appear in the "Terms and Abbreviations" section toward the end of this paper.

2. FFS does not implement atomic file updates because the associated overhead is unacceptable for general use, since most applications do not need atomicity guarantees when writing to files.

```
/* "A" contains valid data. */
...
a_new = open("A_new", O_RDWR | O_CREAT,
0644);
[Write data to A_new.]
fsync(a_new);
rename("A_new", "A");
```

Recovery is as simple as deleting "A_new", if it exists.

Transactional DBMS's must be able to atomically write data to persistent storage. As shown above, atomic writes can be achieved using standard filesystems. However, the performance of such schemes is far from ideal. The filesystem discussed in this paper is specially designed to meet the specific needs of a transactional DBMS. The filesystem discussed in this paper, hereafter referred to as a *block repository* or *BR*, has many architectural similarities to journaled filesystems. It differs though from most journaled filesystems in at least the following ways:

- Provides a simple block-oriented interface, as opposed to a file-oriented interface. The *BR* provides a mechanism for implementing data storage, but almost no policy.
- Implemented in user-land for improved performance and control. Rather than reading and writing data via system calls, a library is linked into the application. The *block repository* library assumes exclusive access of all storage resources that are allocated to it. This is undesirable for typical multiprocess applications, but a useful simplification for single-process server applications.
- Data are stored on multiple devices, called backing stores. In this regard, the *BR* integrates and relies on some concepts normally found in volume management software.

2 Backing Store Creation

A *block repository* consists of four or more backing stores. A backing store is a file or raw device that consists of a header and data space, as shown in Figure 1. The backing store header is triple-redundant so that atomic header updates can be guaranteed. The three copies of backing store header data are striped so that the first portion of each copy can be compared in order to detect and repair data corruption before reading in the remainder of the three copies of the header data. If the three copies of header data were not striped, then the first copy's notion of the header size would have to be trusted, which is not safe.

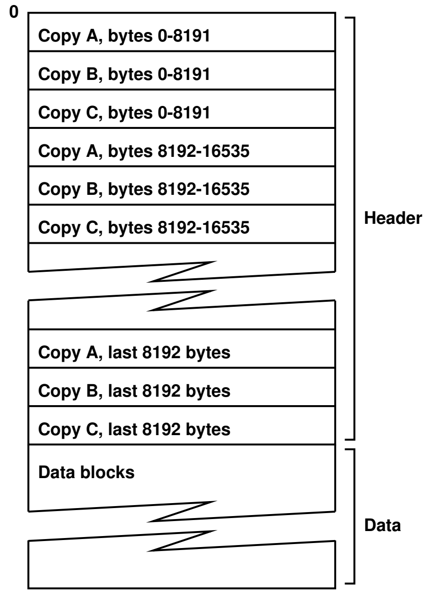

Figure 1: Backing Store Structure

The data stored in each copy of the backing store header is shown in Figure 2.

3 Block Repository Creation

Figure 3 shows the *block repository* structure. Before a *BR* can be brought online, at least a portion of each of the four logical sections of the *BR* must be backed by one or more backing stores.

Each backing store in the *BR* has a copy of the backing store list. When a change is made to the backing store list, the change is synchronously written to each backing store header, in the order that the backing stores are listed. Care is taken to write backing store header changes in this order in so as to assure the ability to recover from backing store headers that disagree

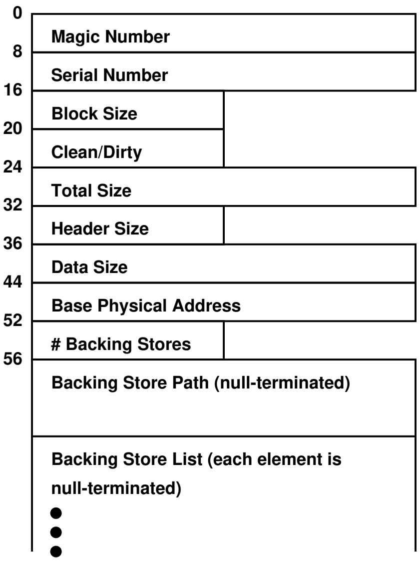

Figure 2: Backing Store Header Structure

with each other. A serial number is included in the header to aid the detection of differences between backing store headers.

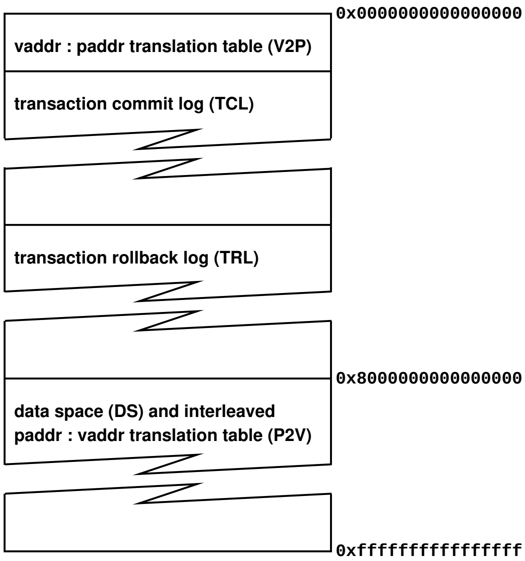

Figure 3: Block Repository Structure

4 Block Repository Startup and Recovery

As with data modifications that are made during normal *BR* operation, care must be taken during startup and recovery to never write data in such a way that could irreversibly corrupt the *BR*. The general startup and recovery algorithm is as follows:

1. Open a backing store, S, that is part of the *BR*.
2. While S is not the first backing store in S's backing store list:
   (a) Open the backing store, S', that is first in S's backing store list.
   (b) Make S' the new S.
3. Repair S if necessary.
4. Repair and roll forward all backing stores in S's list of backing stores to contain the same header data, if necessary.
5. Map all backing stores.
6. Roll forward the transaction commit log (TCL), if necessary.

5 Block Repository Operation

The *block repository* is designed to be able to stay online for long periods of time. This requires that all normal operations on the *BR* must allow uninterrupted availability of data. Below are brief descriptions of some common online *BR* operations.

5.1 Backing Store Operations

As mentioned earlier, a *BR* must be backed by at least four backing stores for it to be brought online. The *BR* design allows online insertion and deletion of backing stores, such that the *BR* can grow and shrink dramatically in size without ever having to be taken offline for maintenance or reconfiguration. Backings can overlap, but there is no performance advantage or gain in resiliency to hardware failures in doing so. The sole reason for allowing overlapping backings is to make it possible to seamlessly move data from one device to another. Possible reasons for moving data from one device to another include:

- Consolidation of multiple small backings into one larger backing.
- Migration to a different storage technology.

Figure 4 shows an in-progress example of consolidating multiple backings (A, B, and C) into one backing (D). The algorithm for adding a new backing store is as follows:

1. Insert a backing store, S, into the *BR*, but mark S as invalid.
2. Begin writing all data writes that are in the range to be backed by S to S, in addition to any other backing stores that back any particular block.
3. Make a single sweep through the entire range of blocks to be backed by S, read the values of the blocks, and write them to S.
4. Mark S as valid.
5. Append S to the backing store list in each backing store header.

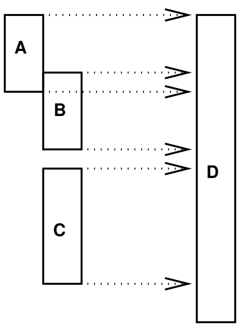

Figure 4: Consolidation of Multiple Backings

A backing store, S, can only be removed if there are one or more backing stores that also back the same blocks as S does, or if the blocks backed by S are not allocated, or a combination of both. The definition of “allocated” varies, depending on the logical section of the *BR*:

V2P: Once a *V2P* entry has been allocated, it cannot be freed without the user explicitly deallocating it. This limits the *BR*'s ability to compact and relocate entries in the *V2P* to such an extent that little effort is made to keep the *V2P* compact. The *V2P* must be contiguously backed from the beginning through the last allocated *V2P* entry. In practice, this means that the backed portion of the *V2P* can grow, but rarely shrinks substantially without the explicit aid of the user.

TCL, TRL: The *TCL* and *TRL* are conceptually rings of buffer space. During normal operation, part of the buffer ring contains valid data, and the other part is empty. At any point in time, the portion that contains valid data is considered to to be allocated. A backing store, S, can only be removed from the *TCL* or *TRL* when no portion of S is the sole backing of an allocated region.

DS/P2V: The *DS* and *P2V* are subdivided into *extent group*s, which are discussed in more detail later. An *extent group* is the unit of allocation from the perspective of backing stores. If any data block within an *extent group* is allocated, the entire *extent group* must be backed. The *P2V* makes it possible to efficiently move data blocks without affecting the user's ability to access the data blocks via *vaddr*. This means that it is practical to compact data block use, and even empty entire ranges of the *DS*/*P2V* of valid data, so that backing stores can be easily removed.

5.2 Block Caching

The *BR* implements an LRU caching policy. Blocks that have no lockers age, and are flushed from the cache when there is demand for block cache space due to reading of non-cached data. Naturally, this implies the constraint that there must be a large enough block cache to accommodate all concurrently locked blocks. In practice, for a reasonably sized block cache, this limitation is only an issue if faulty programming causes an accumulation of stale locks.

s	t	d	q	r	w	x	
√		√	√	√	√	√	s
	q		√	√	√	√	t
		√	√	√	√	√	d
			√	√			q
				√	√		r
							w
							x

Figure 5: Block Lock Compatibility Matrix

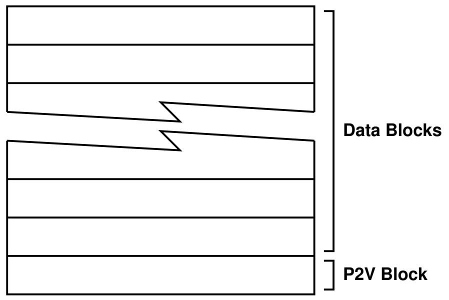

Figure 6: An Extent Group

5.3 Block Locking

Figure 5 shows the seven types of locks on data blocks, and their compatibility with each other. Following is a short description of each type of lock:

s, t : Non-serialized (*s*) and serialized (*t*) place holder locks. In order to obtain one of the other lock types, a user must first obtain an *s* or *t* lock.

d : Potential deletion lock. This is an advisory lock that supports a deletion algorithm for a form of B-trees.[3]

q : Non-exclusive read lock. This form of read lock is compatible with *w* locks, and should only be used when the reader can tolerate dirty reads of any sort that may be introduced by writers that hold *w* locks. The interactions between *q* locks and *w* locks offer a mechanism for allowing dirty reads, but no policy for how to deal with the effects of dirty reads.

r : Non-exclusive read lock.

w : Write lock that allows simultaneous *q* locks.

x : Exclusive write lock.

5.4 Data Block Management

Externally, data blocks are always accessed via *vaddr*. Internally, the *V2P* maps *vaddr*'s to *paddr*'s, and the *P2V* maps *paddr*'s to *vaddr*'s. Data blocks reside in the *DS*. The *DS* and *P2V* are interspersed such that one *P2V* block and a number of *DS* blocks are grouped into an *extent group*, as shown in Figure 6.

Extents provide a mechanism for allocating multiple data blocks that are physically near each other. This generally improves the locality of data access, assuming that data blocks that are allocated in groups tend to be accessed in groups.

The *block repository*'s support for extents is in most ways simple. Extents always consist of a number of blocks that is a power of two, so that extents can be split and collated in logarithmic time. Once an extent is allocated, there is no longer a deterministic way to know the boundaries of the extent. In other words, it begins to be treated as some number of data blocks that have no explicit link to each other. This means that blocks that were allocated as an extent can be deallocated one at a time, and the worst thing that will happen is some extent fragmentation. Even this fragmentation is of limited concern though, since data blocks can be physically moved without external effects. Thus, by compacting allocated data blocks during light system load, extent fragmentation can be kept to a minimum.

Figure 7 shows a fully collated *extent group*. Since each *extent group* is followed by one *P2V* block, *extent group*s always consist of $(2^n - 1)$ data blocks, where $(6 \leq n \leq 16)$. Figure 8 shows what a partially allocated *extent group* could look like. Note that neither of these figures distinguishes between allocated and free extents, though there is bookkeeping information in the *P2V* block that keeps track of allocation, so that extents can be

3. The original motivation for the block repository discussed in this paper is to support research on a highly concurrent B-tree algorithm.

collated with their neighbors as they are freed.

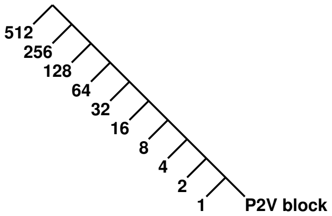

Figure 7: Fully Collated Extent Tree

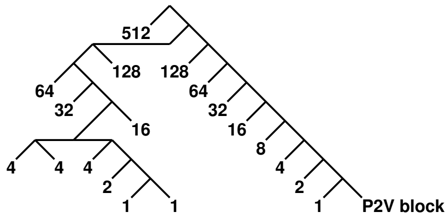

Figure 8: Partially Allocated Extent Tree

5.5 Transaction Commit Log Processing

All writes to the *V2P*, *DS*, and *P2V* sections of the *BR* are first recorded in the transaction commit log (*TCL*). At some later time, the log data are consumed by another thread of execution, here referred to as the cleaner. The cleaner reads the tail end of the *TCL*, and writes blocks to their permanent locations on disk. After each group of blocks in the *TCL* is written out, the cleaner marks the group as invalid, so that in the case of crash recovery, there is no need to process portions of the *TCL* that have already been taken care of.

Due to the interactions between the block cache and the *TCL*, there is a significant performance advantage to having a large amount of valid data in the *TCL* during normal operation. Suppose that a particular data block, *D*, is being written to quite regularly. Over time, the *TCL* will have recorded many different versions of *D*. When the cleaner comes across a block in the *TCL* that records a version of *D*, it first looks to see if *D* resides in the block cache. If so, the most recent version of *D* is written to its permanent location. If not, the cleaner can correctly assume that when *D* was flushed from the block cache, it was first written to the *TCL*, then written to its permanent location on disk. This means that the cleaner, along with how the block cache flushes aged blocks, is in many cases able to write *D* to its permanent location only once, even if *D* was modified many times. Naturally, after a system fault, the entire *TCL* must be processed before the *BR* can be brought back online.

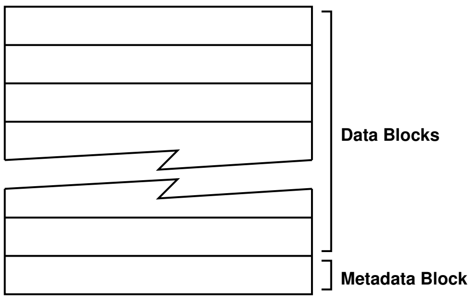

Figure 9: Structure of Transaction Commit and Rollback Logs

Each block of a log transaction has a metadata record associated with it, as shown in Figure 10. A log transaction consists of one or more data blocks. If a log transaction consists of four blocks, the transaction sequence numbers count from four down to one. The transaction sequence numbers also serve the purpose of “flip-flops”. If the “flip” and the “flop” are different, this indicates there was an incomplete write as a failure occurred. The flip-flops are necessary since the log metadata are not redundant, and in the case of log writes, redundancy would be more expensive from a performance perspective than the flip-flops.

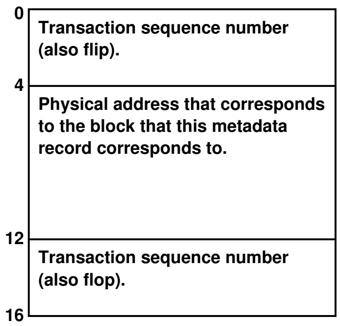

Figure 10: Log Metadata Record

6 Backup and Restore

The *block repository* supports both full and incremental online backup. A full backup is made using the following algorithm:

1. Note the current head of the *TRL*, $H = head(TRL)$.
2. Do a block-wise copy to backup of the *V2P*.
3. Do a block-wise copy of all valid *DS* blocks, in such a way that during restore, each block can be associated with its *vaddr*.
4. Note the current head of the *TRL*, $H' = head(TRL')$.
5. Do a block-wise copy of all *TRL* blocks between H and H'.

An incremental backup can be made by copying the portion of the *TCL* written since the last full backup was made.

BR restoration is accomplished by restoring a full backup, then, in chronological order, restoring any incremental backups that were made.

7 Summary

Traditional filesystems do not provide the combination of atomic data writes and a simple block-oriented interface that transactional DBMS's require. By integrating parts of existing filesystem technologies, a simple streamlined data storage mechanism can be created that meets the needs of transactional systems without the complexity and non-portability of explicit operating system kernel support.

Due to the *block repository*'s user-land nature and its use of multiple devices for storage, performance can be highly tuned to the needs of the individual application.

Terms and Abbreviations

BR: Block repository. For the purposes of discussion in this paper, a block repository is similar in many ways to a conventional filesystem, but operations are on data blocks with a flat 64 bit numerical namespace rather than on files in a hierarchical textual namespace.

vaddr: Virtual address. From the user's perspective, all data blocks are accessed by specifying a vaddr.

paddr: Physical address. Internally, the BR consists of a 64 bit "physical" address space.

V2P: vaddr to paddr translation table. The V2P makes it possible to find the paddr of any data block, given the vaddr.

backing store: Encapsulation of a device or file that provides non-volatile storage for a portion of the block repository's paddr space.

TCL: Transaction commit log. All data writes are first written to the TCL in such a way that once an entire transaction has been written to the TCL, there is enough information to be able to sometime later write the data to their permanent locations, even if there is a crash in between.

TRL: Transaction rollback log. Pristine copies of all modified data blocks are written to the TRL before modified data are written to the TCL. This provides a reliable method for restoring the state of the BR to a previous state, as well as supporting online snapshot backups.

DS: Data space. Data blocks are stored here.

P2V: paddr to vaddr translation table. The P2V is used for various internal algorithms to move data blocks around without causing any externally visible changes. The P2V also contains part of the information that is used to implement extents.

extent group: An extent group consists of a P2V block and a set of DS blocks. The P2V block stores metadata that correspond to the DS blocks in the extent group.

Availability

The block repository described in this paper is part of SQRL, which is an ongoing project sponsored by the Hungry Programmers (http://www.hungry.com) to create a free SQL-92 DBMS. Information and current source code for SQRL can be found at http://www.sqrl.org/sqrl.

All software that is part of SQRL is released under a very agreeable BSD-like license.

References

[Bernstein] Phillip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley Publishing Company, Inc. (1987).

[Elmasri] Ramez Elmasri and Shamkant B. Navathe, *Fundamentals of Database Systems, Second Edition*, Addison-Wesley Publishing Company, Inc. (1994).

[Folk] Michael J. Folk and Bill Zoellick, *File Structures, Second Edition*, Addison-Wesley Publishing Company, Inc. (1992).

[Gray] Jim Gray and Andreas Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann Publishers, Inc. (1993).

[McKusick] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman, *The Design and Implementation of the 4.4BSD Operating System*, Addison-Wesley Publishing Company, Inc. (1996).