

# **XFS Filesystem Disk Structures**

---

**3rd Edition**

---

Copyright © 2006 Silicon Graphics Inc.

© Copyright 2006 Silicon Graphics Inc. All rights reserved. Permission is granted to copy, distribute, and/or modify this document under the terms of the Creative Commons Attribution-Share Alike, Version 3.0 or any later version published by the Creative Commons Corp. A copy of the license is available at <http://creativecommons.org/licenses/by-sa/3.0/us/>.

---

<b>REVISION HISTORY</b>			
NUMBER	DATE	DESCRIPTION	NAME
0.1	2006	Initial Release	Silicon Graphics, Inc
1.0	Fri Jul 03 2009	Publican Conversion	Ryan Lerch
1.1	March 2010	Community Release	Eric Sandeen
1.99	February 2014	AsciiDoc Conversion	Dave Chinner
3	October 2015	Miscellaneous fixes. Add missing field definitions. Add some missing xfs_db examples. Add an overview of XFS. Document the journal format. Document the realtime device.	Darrick Wong
3.1	October 2015	Add v5 fields. Discuss metadata integrity. Document the free inode B+tree. Create an index of magic numbers. Document sparse inodes.	Darrick Wong
3.14	January 2016	Document disk format change testing.	Darrick Wong
3.141	June 2016	Document the reverse-mapping btree. Move the b+tree info to a separate chapter. Discuss overlapping interval b+trees. Discuss new log items for atomic updates. Document the reference-count btree. Discuss block sharing, reflink, & deduplication.	Darrick Wong
3.1415	July 2016	Document the real-time reverse-mapping btree.	Darrick Wong

# Contents

<b>I</b>	<b>High Level Design</b>	<b>1</b>
1	Overview	3
2	Metadata Integrity	4
3	Sharing Data Blocks	5
4	Metadata Reconstruction	6
5	Common XFS Types	8
6	Magic Numbers	10
7	Theoretical Limits	13
8	Testing Filesystem Changes	14
<b>II</b>	<b>Global Structures</b>	<b>15</b>
9	B+trees	16
9.1	Short Format B+trees . . . . .	17
9.2	Long Format B+trees . . . . .	18
10	Allocation Groups	20
10.1	Superblocks . . . . .	21
10.1.1	xfs_db Superblock Example . . . . .	29
10.2	AG Free Space Management . . . . .	30
10.2.1	AG Free Space Block . . . . .	30
10.2.2	AG Free Space B+trees . . . . .	32

---

---

10.2.3	AG Free List	34
10.2.3.1	xfs_db AGF Example	36
10.3	AG Inode Management	38
10.3.1	Inode Numbers	38
10.3.2	Inode Information	38
10.4	Inode B+trees	40
10.4.1	xfs_db AGI Example	42
10.5	Sparse Inodes	43
10.5.1	xfs_db Sparse Inode AGI Example	44
10.6	Real-time Devices	46
10.7	Reverse-Mapping B+tree	46
10.7.1	xfs_db rmapbt Example	48
10.8	Reference Count B+tree	51
10.8.1	xfs_db refcntbt Example	52
<b>11</b>	<b>Journaling Log</b>	<b>54</b>
11.1	Log Records	54
11.2	Log Operations	56
11.3	Log Items	57
11.3.1	Transaction Headers	58
11.3.2	Intent to Free an Extent	59
11.3.3	Completion of Intent to Free an Extent	60
11.3.4	Reverse Mapping Updates Intent	61
11.3.5	Completion of Reverse Mapping Updates	63
11.3.6	Reference Count Updates Intent	63
11.3.7	Completion of Reference Count Updates	64
11.3.8	File Block Mapping Intent	65
11.3.9	Completion of File Block Mapping Updates	66
11.3.10	Inode Updates	66
11.3.11	Inode Data Log Item	68
11.3.12	Buffer Log Item	68
11.3.13	Buffer Data Log Item	69
11.3.14	Update Quota File	69
11.3.15	Quota Update Data Log Item	69
11.3.16	Disable Quota Log Item	70
11.3.17	Inode Creation Log Item	70
11.4	xfs_logprint Example	71

---

---

<b>12 Internal Inodes</b>	<b>74</b>
12.1 Quota Inodes	74
12.2 Real-time Inodes	77
12.2.1 Real-Time Bitmap Inode	77
12.2.2 Real-Time Summary Inode	77
12.2.3 Real-Time Reverse-Mapping B+tree	78
12.2.3.1 xfs_db rtrmapbt Example	79
<b>III Dynamically Allocated Structures</b>	<b>82</b>
<b>13 On-disk Inode</b>	<b>83</b>
13.1 Inode Core	84
13.2 Unlinked Pointer	89
13.3 Data Fork	90
13.3.1 Regular Files (S_IFREG)	91
13.3.2 Directories (S_IFDIR)	91
13.3.3 Symbolic Links (S_IFLNK)	91
13.3.4 Other File Types	91
13.4 Attribute Fork	92
13.4.1 Extended Attribute Versions	92
<b>14 Data Extents</b>	<b>94</b>
14.1 Extent List	95
14.1.1 xfs_db Inode Data Fork Extents Example	96
14.2 B+tree Extent List	98
14.2.1 xfs_db bmbt Example	102
<b>15 Directories</b>	<b>103</b>
15.1 Short Form Directories	104
15.1.1 xfs_db Short Form Directory Example	106
15.2 Block Directories	109
15.2.1 xfs_db Block Directory Example	114
15.3 Leaf Directories	117
15.3.1 Directory and Attribute Block Headers	119
15.3.2 xfs_db Leaf Directory Example	122
15.4 Node Directories	126
15.4.1 Directory and Attribute Internal Nodes	127
15.4.2 xfs_db Node Directory Example	131
15.5 B+tree Directories	133
15.5.1 xfs_db B+tree Directory Example	134

---

---

<b>16 Extended Attributes</b>	<b>137</b>
16.1 Short Form Attributes	137
16.1.1 xfs_db Short Form Attribute Example	139
16.2 Leaf Attributes	143
16.2.1 xfs_db Leaf Attribute Example	148
16.3 Node Attributes	150
16.3.1 xfs_db Node Attribute Example	151
16.4 B+tree Attributes	154
16.4.1 xfs_db B+tree Attribute Example	154
16.5 Remote Attribute Values	155
<b>17 Symbolic Links</b>	<b>156</b>
17.1 Short Form Symbolic Links	156
17.1.1 xfs_db Short Form Symbolic Link Example	157
17.2 Extent Symbolic Links	157
17.2.1 xfs_db Symbolic Link Extent Example	159

---

## Part I

# High Level Design

---



XFS is a high performance filesystem which was designed to maximize parallel throughput and to scale up to extremely large 64-bit storage systems. Originally developed by SGI in October 1993 for IRIX, XFS can handle large files, large filesystems, many inodes, large directories, large file attributes, and large allocations. Filesystems are optimized for parallel access by splitting the storage device into semi-autonomous allocation groups. XFS employs branching trees (B+ trees) to facilitate fast searches of large lists; it also uses delayed extent-based allocation to improve data contiguity and IO performance.

This document describes the on-disk layout of an XFS filesystem and how to use the debugging tools `xfs_db` and `xfs_logprint` to inspect the metadata structures. It also describes how on-disk metadata relates to the higher level design goals.

The information contained in this document derives from the XFS source code in the Linux kernel as of v4.3. This book's source code is available at `git://git.kernel.org/pub/scm/fs/xfs/xfs-documentation.git`. Feedback should be sent to the XFS mailing list, currently at `xfs@oss.sgi.com`.

---

**Note**

All fields in XFS metadata structures are in big-endian byte order except for log items which are formatted in host order.

---

# Chapter 1

## Overview

XFS presents to users a standard Unix filesystem interface: a rooted tree of directories, files, symbolic links, and devices. All five of those entities are represented inside the filesystem by an index node, or “inode”; each node is uniquely referenced by an inode number. Directories consist of (name, inode number) tuples and it is possible for multiple tuples to contain the same inode number. Data blocks are associated with files by means of a block map in each index node. It is also possible to attach (key, value) tuples to any index node; these are known as “extended attributes”, which extend beyond the standard Unix file attributes.

Internally, XFS filesystems are divided into a number of equally sized chunks called Allocation Groups. Each AG can almost be thought of as an individual filesystem that maintains its own space usage, index nodes, and other secondary metadata. Having multiple AGs allows XFS to handle most operations in parallel without degrading performance as the number of concurrent accesses increases. Each allocation group uses multiple B+trees to maintain bookkeeping records such as the locations of free blocks, the locations of allocated inodes, and the locations of free inodes.

Files, symbolic links, and directories can have up to two block maps, or “forks”, which associate filesystems blocks with a particular file or directory. The “attribute fork” tracks blocks used to store and index extended attributes, whereas the “data fork” tracks file data blocks, symbolic link targets, or directory blocks, depending on the type of the inode record. Both forks associate a logical offset with an extent of physical blocks, which makes sparse files and directories possible. Directory entries and extended attributes are contained inside a second-level data structure within the blocks that are mapped by the forks. This structure consists of variable-length directory or attribute records and possibly a second B+tree to index these records.

XFS employs a journalling log in which metadata changes are collected so that filesystem operations can be carried out atomically in the case of a crash. Furthermore, there is the concept of a real-time device wherein allocations are tracked more simply and in larger chunks to reduce jitter in allocation latency.

## Chapter 2

# Metadata Integrity

Prior to version 5, most XFS metadata blocks contained a magic number that could provide a minimal sanity check that a block read off the disk contained the same type of data that the code thought it was reading off the disk. However, this was insufficient – given a correct type code, it was still impossible to tell if the block was from a previous filesystem, or happened to be owned by something else, or had been written to the wrong location on disk. Furthermore, not all metadata blocks had magic numbers – remote extended attributes and extent symbolic links had no protection at all.

Therefore, the version 5 disk format introduced larger headers for all metadata types, which enable the filesystem to check information being read from the disk more rigorously. Metadata integrity fields now include:

- **Magic numbers**, to classify all types of metadata. This is unchanged from v4.
- A copy of the filesystem **UUID**, to confirm that a given disk block is connected to the superblock.
- The **owner**, to avoid accessing a piece of metadata which belongs to some other part of the filesystem.
- The filesystem **block number**, to detect misplaced writes.
- The **log serial number** of the last write to this block, to avoid replaying obsolete log entries.
- A CRC32c **checksum** of the entire block, to detect minor corruption.

Metadata integrity coverage has been extended to all metadata blocks in the filesystem, with the following notes:

- Inodes can have multiple “owners” in the directory tree; therefore the record contains the inode number instead of an owner or a block number.
- Superblocks have no owners.
- The disk quota file has no owner or block numbers.
- Metadata owned by files list the inode number as the owner.
- Per-AG data and B+tree blocks list the AG number as the owner.
- Per-AG header sectors don’t list owners or block numbers, since they have fixed locations.
- Remote attribute blocks are not logged and therefore the LSN must be -1.

This functionality enables XFS to decide that a block contents are so unexpected that it should stop immediately. Unfortunately checksums do not allow for automatic correction. Please keep regular backups, as always.

---

## Chapter 3

# Sharing Data Blocks

On a traditional filesystem, there is a 1:1 mapping between a logical block offset in a file and a physical block on disk, which is to say that physical blocks are not shared. However, there exist various use cases for being able to share blocks between files—deduplicating files saves space on archival systems; creating space-efficient clones of disk images for virtual machines and containers facilitates efficient datacenters; and deferring the payment of the allocation cost of a file system tree copy as long as possible makes regular work faster. In all of these cases, a write to one of the shared copies **must** not affect the other shared copies, which means that writes to shared blocks must employ a copy-on-write strategy. Sharing blocks in this manner is commonly referred to as “relinking”.

XFS implements block sharing in a fairly straightforward manner. All existing data fork structures remain unchanged, save for the addition of a per-allocation group [reference count B+tree](#) Section 10.8. This data structure tracks reference counts for all shared physical blocks, with a few rules to maintain compatibility with existing code: If a block is free, it will be tracked in the free space B+trees. If a block is owned by a single file, it appears in neither the free space nor the reference count B+trees. If a block is shared, it will appear in the reference count B+tree with a reference count  $\geq 2$ . The first two cases are established precedent in XFS, so the third case is the only behavioral change.

When a filesystem block is shared, the block mapping in the destination file is updated to point to that filesystem block and the reference count B+tree records are updated to reflect the increased refcount. If a shared block is written, a new block will be allocated, the dirty data written to this new block, and the file’s block mapping updated to point to the new block. If a shared block is unmapped, the reference count records are updated to reflect the decreased refcount and the block is also freed if its reference count becomes zero. This enables users to create space efficient clones of disk images and to copy filesystem subtrees quickly, using the standard Linux coreutils packages.

Deduplication employs the same mechanism to share blocks and copy them at write time. However, the kernel confirms that the contents of both files are identical before updating the destination file’s mapping. This enables XFS to be used by userspace deduplication programs such as `duperemove`.

## Chapter 4

# Metadata Reconstruction

---

**Note**

This is a theoretical discussion of how reconstruction could work; none of this is implemented as of 2015.

---

A simple UNIX filesystem can be thought of in terms of a directed acyclic graph. To a first approximation, there exists a root directory node, which points to other nodes. Those other nodes can themselves be directories or they can be files. Each file, in turn, points to data blocks.

XFS adds a few more details to this picture:

- The real root(s) of an XFS filesystem are the allocation group headers (superblock, AGF, AGI, AGFL).
- Each allocation group's headers point to various per-AG B+trees (free space, inode, free inodes, free list, etc.)
- The free space B+trees point to unused extents;
- The inode B+trees point to blocks containing inode chunks;
- All superblocks point to the root directory and the log;
- Hardlinks mean that multiple directories can point to a single file node;
- File data block pointers are indexed by file offset;
- Files and directories can have a second collection of pointers to data blocks which contain extended attributes;
- Large directories require multiple data blocks to store all the subpointers;
- Still larger directories use high-offset data blocks to store a B+tree of hashes to directory entries;
- Large extended attribute forks similarly use high-offset data blocks to store a B+tree of hashes to attribute keys; and
- Symbolic links can point to data blocks.

The beauty of this massive graph structure is that under normal circumstances, everything known to the filesystem is discoverable (access controls notwithstanding) from the root. The major weakness of this structure of course is that breaking a edge in the graph can render entire subtrees inaccessible. `xfs_repair` "recovers" from broken directories by scanning for unlinked inodes and connecting them to `/lost+found`, but this isn't sufficiently general

---

to recover from breaks in other parts of the graph structure. Wouldn't it be useful to have back pointers as a secondary data structure? The current repair strategy is to reconstruct whatever can be rebuilt, but to scrap anything that doesn't check out.

The [reverse-mapping B+tree](#) Section 10.7 fills in part of the puzzle. Since it contains copies of every entry in each inode's data and attribute forks, we can fix a corrupted block map with these records. Furthermore, if the inode B+trees become corrupt, it is possible to visit all inode chunks using the reverse-mapping data. Should XFS ever gain the ability to store parent directory information in each inode, it also becomes possible to resurrect damaged directory trees, which should reduce the complaints about inodes ending up in `/lost+found`. Everything else in the per-AG primary metadata can already be reconstructed via `xfs_repair`. Hopefully, reconstruction will not turn out to be a fool's errand.

## Chapter 5

# Common XFS Types

All the following XFS types can be found in `xfs_types.h`. NULL values are always -1 on disk (ie. all bits for the value set to one).

**xfs\_ino\_t**

Unsigned 64 bit absolute [inode number](#) [Section 10.3.1](#).

**xfs\_off\_t**

Signed 64 bit file offset.

**xfs\_daddr\_t**

Signed 64 bit disk address (sectors).

**xfs\_agnumber\_t**

Unsigned 32 bit [AG number](#) [Chapter 10](#).

**xfs\_agblock\_t**

Unsigned 32 bit AG relative block number.

**xfs\_extlen\_t**

Unsigned 32 bit [extent](#) [Chapter 14](#) length in blocks.

**xfs\_extnum\_t**

Signed 32 bit number of extents in a data fork.

**xfs\_aextnum\_t**

Signed 16 bit number of extents in an attribute fork.

**xfs\_dablk\_t**

Unsigned 32 bit block number for [directories](#) [Chapter 15](#) and [extended attributes](#) [Chapter 16](#).

**xfs\_dahash\_t**

Unsigned 32 bit hash of a directory file name or extended attribute name.

**xfs\_fsblock\_t**

Unsigned 64 bit filesystem block number combining [AG number](#) [Chapter 10](#) and block offset into the AG.

**xfs\_rfsblock\_t**

Unsigned 64 bit raw filesystem block number.

---

**xfs\_rtblock\_t**

Unsigned 64 bit extent number in the [real-time](#) Section [10.6](#) sub-volume.

**xfs\_fileoff\_t**

Unsigned 64 bit block offset into a file.

**xfs\_filblks\_t**

Unsigned 64 bit block count for a file.

**uuid\_t**

16-byte universally unique identifier (UUID).

**xfs\_fsize\_t**

Signed 64 bit byte size of a file.

---



## Chapter 6

# Magic Numbers

These are the magic numbers that are known to XFS, along with links to the relevant chapters. Magic numbers tend to have consistent locations:

- 32-bit magic numbers are always at offset zero in the block.
- 16-bit magic numbers for the directory and attribute B+tree are at offset eight.
- The quota magic number is at offset zero.
- The inode magic is at the beginning of each inode.

Flag	Hexadecimal	ASCII	Data structure
XFS_SB_MAGIC	0x58465342	XFSB	<a href="#">Superblock</a> Section 10.1
XFS_AGF_MAGIC	0x58414746	XAGF	<a href="#">Free Space</a> Section 10.2.1
XFS_AGI_MAGIC	0x58414749	XAGI	<a href="#">Inode Information</a> Section 10.3.2
XFS_AGFL_MAGIC	0x5841464c	XAFL	<a href="#">Free Space List</a> Section 10.2.3, v5 only
XFS_DINODE_MAGIC	0x494e	IN	<a href="#">Inodes</a> Section 13.1
XFS_DQUOT_MAGIC	0x4451	DQ	<a href="#">Quota Inodes</a> Section 12.1
XFS_SYMLINK_MAGIC	0x58534c4d	XSLM	<a href="#">Symbolic Links</a> Section 17.2
XFS_ABTB_MAGIC	0x41425442	ABTB	<a href="#">Free Space by Block B+tree</a> Section 10.2.2
XFS_ABTB_CRC_MAGIC	0x41423342	AB3B	<a href="#">Free Space by Block B+tree</a> Section 10.2.2, v5 only
XFS_ABTC_MAGIC	0x41425443	ABTC	<a href="#">Free Space by Size B+tree</a> Section 10.2.2
XFS_ABTC_CRC_MAGIC	0x41423343	AB3C	<a href="#">Free Space by Size B+tree</a> Section 10.2.2, v5 only
XFS_IBT_MAGIC	0x49414254	IABT	<a href="#">Inode B+tree</a> Section 10.4

Flag	Hexadecimal	ASCII	Data structure
XFS_IBT_CRC_MAGIC	0x49414233	IAB3	Inode B+tree Section 10.4, v5 only
XFS_FIBT_MAGIC	0x46494254	FIBT	Free Inode B+tree Section 10.4
XFS_FIBT_CRC_MAGIC	0x46494233	FIB3	Free Inode B+tree Section 10.4, v5 only
XFS_BMAP_MAGIC	0x424d4150	BMAP	B+Tree Extent List Section 14.2
XFS_BMAP_CRC_MAGIC	0x424d4133	BMA3	B+Tree Extent List Section 14.2, v5 only
XLOG_HEADER_MAGIC_NUM	0xfeebabe		Log Records Section 11.1
XFS_DA_NODE_MAGIC	0xfebe		Directory/Attribute Node Section 15.4.1
XFS_DA3_NODE_MAGIC	0x3ebe		Directory/Attribute Node Section 15.4.1, v5 only
XFS_DIR2_BLOCK_MAGIC	0x58443242	XD2B	Block Directory Data Section 15.2
XFS_DIR3_BLOCK_MAGIC	0x58444233	XDB3	Block Directory Data Section 15.2, v5 only
XFS_DIR2_DATA_MAGIC	0x58443244	XD2D	Leaf Directory Data Section 15.3
XFS_DIR3_DATA_MAGIC	0x58444433	XDD3	Leaf Directory Data Section 15.3, v5 only
XFS_DIR2_LEAF1_MAGIC	0xd2f1		Leaf Directory Section 15.3
XFS_DIR3_LEAF1_MAGIC	0x3df1		Leaf Directory Section 15.3, v5 only
XFS_DIR2_LEAFN_MAGIC	0xd2ff		Node Directory Section 15.4
XFS_DIR3_LEAFN_MAGIC	0x3dff		Node Directory Section 15.4, v5 only
XFS_DIR2_FREE_MAGIC	0x58443246	XD2F	Node Directory Free Space Section 15.4
XFS_DIR3_FREE_MAGIC	0x58444633	XDF3	Node Directory Free Space Section 15.4, v5 only
XFS_ATTR_LEAF_MAGIC	0xfbee		Leaf Attribute Section 16.2
XFS_ATTR3_LEAF_MAGIC	0x3bee		Leaf Attribute Section 16.2, v5 only
XFS_ATTR3_RMT_MAGIC	0x5841524d	XARM	Remote Attribute Value Section 16.5, v5 only
XFS_RMAP_CRC_MAGIC	0x524d4233	RMB3	Reverse Mapping B+tree Section 10.7, v5 only
XFS_RTRMAP_CRC_MAGIC	0x4d415052	MAPR	Real-Time Reverse Mapping B+tree Section 12.2.3, v5 only
XFS_REFC_CRC_MAGIC	0x52334643	R3FC	Reference Count B+tree Section 10.8, v5 only

The magic numbers for log items are at offset zero in each log item, but items are not aligned to blocks.

Flag	Hexadecimal	ASCII	Data structure
XFS_TRANS_HEADER_MAGIC	0x5452414e	TRAN	<a href="#">Log Transactions Section 11.3.1</a>
XFS_LI_EFI	0x1236		<a href="#">Extent Freeing Intent Log Item Section 11.3.2</a>
XFS_LI_EFD	0x1237		<a href="#">Extent Freeing Done Log Item Section 11.3.3</a>
XFS_LI_IUNLINK	0x1238		Unknown?
XFS_LI_INODE	0x123b		<a href="#">Inode Updates Log Item Section 11.3.10</a>
XFS_LI_BUF	0x123c		<a href="#">Buffer Writes Log Item Section 11.3.12</a>
XFS_LI_DQUOT	0x123d		<a href="#">Update Quota Log Item Section 11.3.14</a>
XFS_LI_QUOTAOFF	0x123e		<a href="#">Quota Off Log Item Section 11.3.16</a>
XFS_LI_ICREATE	0x123f		<a href="#">Inode Creation Log Item Section 11.3.17</a>
XFS_LI_RUI	0x1240		<a href="#">Reverse Mapping Update Intent Section 11.3.4</a>
XFS_LI_RUD	0x1241		<a href="#">Reverse Mapping Update Done Section 11.3.5</a>
XFS_LI_CUI	0x1242		<a href="#">Reference Count Update Intent Section 11.3.6</a>
XFS_LI_CUD	0x1243		<a href="#">Reference Count Update Done Section 11.3.7</a>
XFS_LI_BUI	0x1244		<a href="#">File Block Mapping Update Intent Section 11.3.8</a>
XFS_LI_BUD	0x1245		<a href="#">File Block Mapping Update Done Section 11.3.9</a>

## Chapter 7

# Theoretical Limits

XFS can create really big filesystems!

Item	1KiB blocks	4KiB blocks	64KiB blocks
Blocks	$2^{52}$	$2^{52}$	$2^{52}$
Inodes	$2^{63}$	$2^{63}$	$2^{64}$
Allocation Groups	$2^{32}$	$2^{32}$	$2^{32}$
File System Size	8EiB	8EiB	8EiB
Blocks per AG	$2^{31}$	$2^{31}$	$2^{31}$
Inodes per AG	$2^{32}$	$2^{32}$	$2^{32}$
Max AG Size	2TiB	8TiB	128TiB
Blocks Per File	$2^{54}$	$2^{54}$	$2^{54}$
File Size	8EiB	8EiB	8EiB
Max Dir Size	32GiB	32GiB	32GiB

Linux doesn't support files or devices larger than 8EiB, so the block limitations are largely ignorable.

## Chapter 8

# Testing Filesystem Changes

People put a lot of trust in filesystems to preserve their data in a reliable fashion. To that end, it is very important that users and developers have access to a suite of regression tests that can be used to prove correct operation of any given filesystem code, or to analyze failures to fix problems found in the code. The XFS regression test suite, `xfstests`, is hosted at `git://git.kernel.org/pub/scm/fs/xfs/xfstests-dev.git`. Most tests apply to filesystems in general, but the suite also contains tests for features specific to each filesystem.

When fixing bugs, it is important to provide a testcase exposing the bug so that the developers can avoid a future re-occurrence of the regression. Furthermore, if you're developing a new user-visible feature for XFS, please help the rest of the development community to sustain and maintain the whole codebase by providing generous test coverage to check its behavior.

When altering, adding, or removing an on-disk data structure, please remember to update both the in-kernel structure size checks in `xfs_ondisk.h` and to ensure that your changes are reflected in `xfstest xfs/122`. These regression tests enable us to detect compiler bugs, alignment problems, and anything else that might result in the creation of incompatible filesystem images.

## **Part II**

# **Global Structures**

---

## Chapter 9

# B+trees

XFS uses b+trees to index all metadata records. This well known data structure is used to provide efficient random and sequential access to metadata records while minimizing seek times. There are two btree formats: a short format for records pertaining to a single allocation group, since all block pointers in an AG are 32-bits in size; and a long format for records pertaining to a file, since file data can have 64-bit block offsets. Each b+tree block is either a leaf node containing records, or an internal node containing keys and pointers to other b+tree blocks. The tree consists of a root block which may point to some number of other blocks; blocks in the bottom level of the b+tree contains only records.

Leaf blocks of both types of b+trees have the same general format: a header describing the data in the block, and an array of records. The specific header formats are given in the next two sections, and the record format is provided by the b+tree client itself. The generic b+tree code does not have any specific knowledge of the record format.

```
+-----+-----+-----+
| header | record | records... |
+-----+-----+-----+
```

Internal node blocks of both types of b+trees also have the same general format: a header describing the data in the block, an array of keys, and an array of pointers. Each pointer may be associated with one or two keys. The first key uniquely identifies the first record accessible via the leftmost path down the branch of the tree.

If the records in a b+tree are indexed by an interval, then a range of keys can uniquely identify a single record. For example, if a record covers blocks 12-16, then any one of the keys 12, 13, 14, 15, or 16 return the same record. In this case, the key for the record describing "12-16" is 12. If none of the records overlap, we only need to store one key.

This is the format of a standard b+tree node:

```
+-----+-----+-----+-----+
| header | key | keys... | ptr | ptrs... |
+-----+-----+-----+-----+
```

If the b+tree records do not overlap, performing a b+tree lookup is simple. Start with the root. If it is a leaf block, perform a binary search of the records until we find the record with a lower key than our search key. If the block is a node block, perform a binary search of the keys until we find a key lower than our search key, then follow the pointer to the next block. Repeat until we find a record.

However, if b+tree records contain intervals and are allowed to overlap, the internal nodes of the b+tree become larger:

```

+-----+-----+-----+-----+-----+-----+-----+
| header | low key | high key | low key | high key... | ptr | ptrs... |
+-----+-----+-----+-----+-----+-----+-----+

```

The low keys are exactly the same as the keys in the non-overlapping b+tree. High keys, however, are a little different. Recall that a record with a key consisting of an interval can be referenced by a number of keys. Since the low key of a record indexes the low end of that key range, the high key indexes the high end of the key range. Returning to the example above, the high key for the record describing "12-16" is 16. The high key recorded in a b+tree node is the largest of the high keys of all records accessible under the subtree rooted by the pointer. For a level 1 node, this is the largest high key in the pointed-to leaf node; for any other node, this is the largest of the high keys in the pointed-to node.

Nodes and leaves use the same magic numbers.

## Short Format B+trees

Each allocation group uses a "short format" B+tree to index various information about the allocation group. The structure is called short format because all block pointers are AG block numbers. The trees use the following header:

```

struct xfs_btree_sblock {
    __be32          bb_magic;
    __be16          bb_level;
    __be16          bb_numrecs;
    __be32          bb_leftsib;
    __be32          bb_rightsib;

    /* version 5 filesystem fields start here */
    __be64          bb_blkno;
    __be64          bb_lsn;
    uuid_t          bb_uuid;
    __be32          bb_owner;
    __le32          bb_crc;
};

```

### **bb\_magic**

Specifies the magic number for the per-AG B+tree block.

### **bb\_level**

The level of the tree in which this block is found. If this value is 0, this is a leaf block and contains records; otherwise, it is a node block and contains keys and pointers.

### **bb\_numrecs**

Number of records in this block.

### **bb\_leftsib**

AG block number of the left sibling of this B+tree node.

### **bb\_rightsib**

AG block number of the right sibling of this B+tree node.

### **bb\_blkno**

FS block number of this B+tree block.



**bb\_lsn**

Log sequence number of the last write to this block.

**bb\_uuid**

The UUID of this block, which must match either `sb_uuid` or `sb_meta_uuid` depending on which features are set.

**bb\_owner**

The AG number that this B+tree block ought to be in.

**bb\_crc**

Checksum of the B+tree block.

## Long Format B+trees

Long format B+trees are similar to short format B+trees, except that their block pointers are 64-bit filesystem block numbers instead of 32-bit AG block numbers. Because of this, long format b+trees can be (and usually are) rooted in an inode's data or attribute fork. The nodes and leaves of this B+tree use the `xfstree_lblock` declaration:

```
struct xfstree_lblock {
    __be32          bb_magic;
    __be16          bb_level;
    __be16          bb_numrecs;
    __be64          bb_leftsib;
    __be64          bb_rightsib;

    /* version 5 filesystem fields start here */
    __be64          bb_blkno;
    __be64          bb_lsn;
    uuid_t          bb_uuid;
    __be64          bb_owner;
    __le32          bb_crc;
    __be32          bb_pad;
};
```

**bb\_magic**

Specifies the magic number for the btree block.

**bb\_level**

The level of the tree in which this block is found. If this value is 0, this is a leaf block and contains records; otherwise, it is a node block and contains keys and pointers.

**bb\_numrecs**

Number of records in this block.

**bb\_leftsib**

FS block number of the left sibling of this B+tree node.

**bb\_rightsib**

FS block number of the right sibling of this B+tree node.

**bb\_blkno**

FS block number of this B+tree block.

---

**bb\_lsn**

Log sequence number of the last write to this block.

**bb\_uuid**

The UUID of this block, which must match either `sb_uuid` or `sb_meta_uuid` depending on which features are set.

**bb\_owner**

The AG number that this B+tree block ought to be in.

**bb\_crc**

Checksum of the B+tree block.

**bb\_pad**

Pads the structure to 64 bytes.

---

## Chapter 10

# Allocation Groups

As mentioned earlier, XFS filesystems are divided into a number of equally sized chunks called Allocation Groups. Each AG can almost be thought of as an individual filesystem that maintains its own space usage. Each AG can be up to one terabyte in size ( $512 \text{ bytes} \times 2^{31}$ ), regardless of the underlying device's sector size.

Each AG has the following characteristics:

- A super block describing overall filesystem info
- Free space management
- Inode allocation and tracking
- Reverse block-mapping index (optional)
- Data block reference count index (optional)

Having multiple AGs allows XFS to handle most operations in parallel without degrading performance as the number of concurrent accesses increases.

The only global information maintained by the first AG (primary) is free space across the filesystem and total inode counts. If the `XFS_SB_VERSION2_LAZYSB_COUNTBIT` flag is set in the superblock, these are only updated on-disk when the filesystem is cleanly unmounted (`umount` or `shutdown`).

Immediately after a `mkfs.xfs`, the primary AG has the following disk layout; the subsequent AGs do not have any inodes allocated:

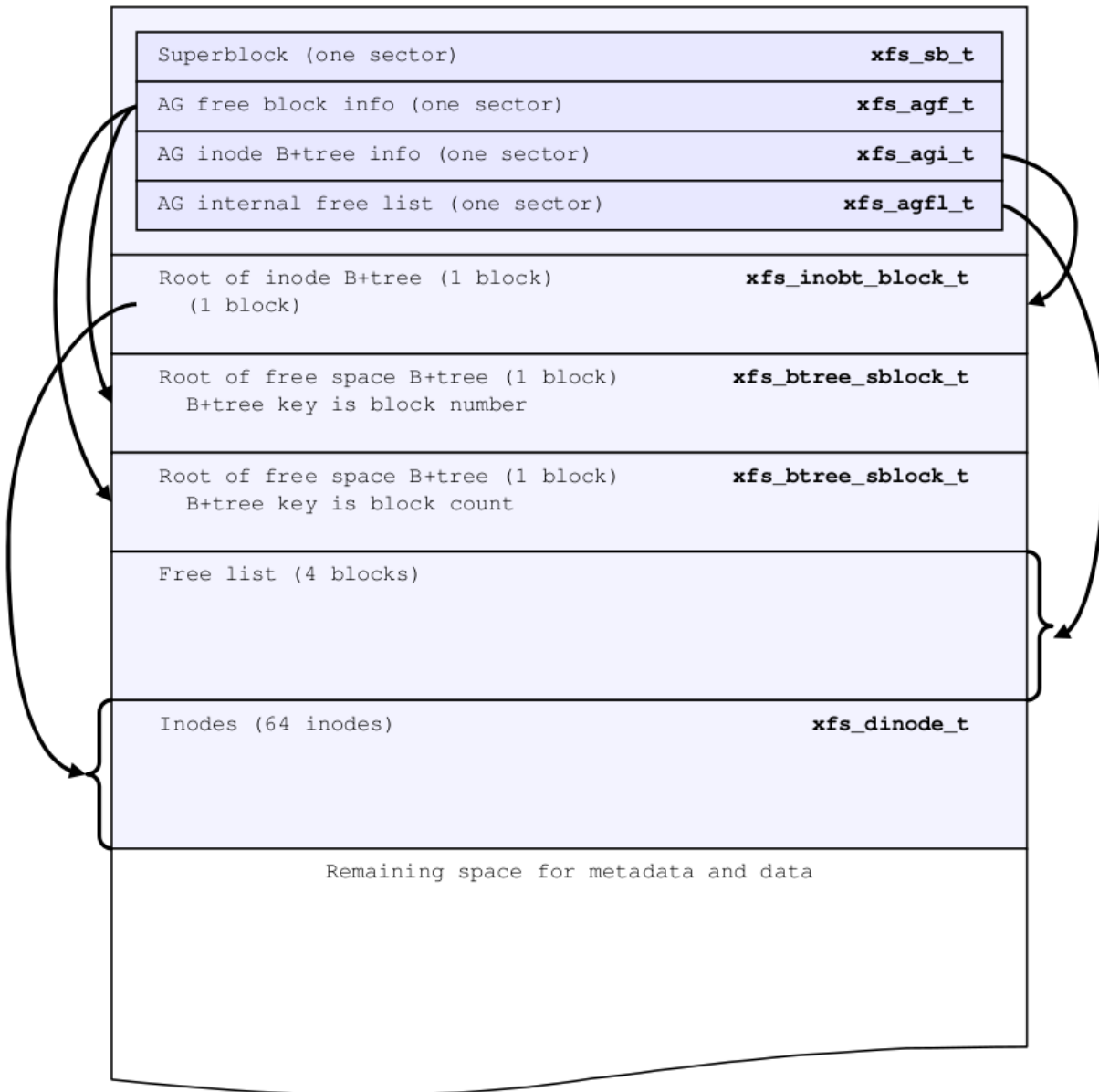


Figure 10.1: Allocation group layout

Each of these structures are expanded upon in the following sections.

## Superblocks

Each AG starts with a superblock. The first one, in AG 0, is the primary superblock which stores aggregate AG information. Secondary superblocks are only used by `xfs_repair` when the primary superblock has been corrupted. A superblock is one sector in length.

The superblock is defined by the following structure. The description of each field follows.

```

struct xfs_sb
{
    __uint32_t          sb_magicnum;
    __uint32_t          sb_blocksize;
    xfs_rfsblock_t      sb_dblocks;
    xfs_rfsblock_t      sb_rblocks;
    xfs_rtbblock_t      sb_rextents;
    uuid_t              sb_uuid;
    xfs_fsbblock_t      sb_logstart;
    xfs_ino_t            sb_rootino;
    xfs_ino_t            sb_rbmino;
    xfs_ino_t            sb_rsumino;
    xfs_agblock_t        sb_rextsize;
    xfs_agblock_t        sb_agblocks;
    xfs_agnumber_t      sb_agcount;
    xfs_extlen_t        sb_rmblocks;
    xfs_extlen_t        sb_logblocks;
    __uint16_t          sb_versionnum;
    __uint16_t          sb_sectsize;
    __uint16_t          sb_inodesize;
    __uint16_t          sb_inopblock;
    char                 sb_fname[12];
    __uint8_t           sb_blocklog;
    __uint8_t           sb_sectlog;
    __uint8_t           sb_inodelog;
    __uint8_t           sb_inopblog;
    __uint8_t           sb_agblklog;
    __uint8_t           sb_rextslog;
    __uint8_t           sb_inprogress;
    __uint8_t           sb_imax_pct;
    __uint64_t          sb_icount;
    __uint64_t          sb_ifree;
    __uint64_t          sb_fdblocks;
    __uint64_t          sb_frextents;
    xfs_ino_t            sb_uquotino;
    xfs_ino_t            sb_gquotino;
    __uint16_t          sb_qflags;
    __uint8_t           sb_flags;
    __uint8_t           sb_shared_vn;
    xfs_extlen_t        sb_inoalignmt;
    __uint32_t          sb_unit;
    __uint32_t          sb_width;
    __uint8_t           sb_dirblklog;
    __uint8_t           sb_logsectlog;
    __uint16_t          sb_logsectsize;
    __uint32_t          sb_logsunit;
    __uint32_t          sb_features2;
    __uint32_t          sb_bad_features2;

    /* version 5 superblock fields start here */
    __uint32_t          sb_features_compat;
    __uint32_t          sb_features_ro_compat;
    __uint32_t          sb_features_incompat;
    __uint32_t          sb_features_log_incompat;

```

```

        __uint32_t          sb_crc;
        xfs_extlen_t       sb_spino_align;

        xfs_ino_t          sb_pquotino;
        xfs_lsn_t          sb_lsn;
        uuid_t             sb_meta_uuid;
        xfs_ino_t          sb_rrmapino;
};

```

**sb\_magicnum**

Identifies the filesystem. Its value is XFS\_SB\_MAGIC “XFSB” (0x58465342).

**sb\_blocksize**

The size of a basic unit of space allocation in bytes. Typically, this is 4096 (4KB) but can range from 512 to 65536 bytes.

**sb\_dblocks**

Total number of blocks available for data and metadata on the filesystem.

**sb\_rblocks**

Number blocks in the real-time disk device. Refer to [real-time sub-volumes](#) Section 10.6 for more information.

**sb\_rextents**

Number of extents on the real-time device.

**sb\_uuid**

UUID (Universally Unique ID) for the filesystem. Filesystems can be mounted by the UUID instead of device name.

**sb\_logstart**

First block number for the journaling log if the log is internal (ie. not on a separate disk device). For an external log device, this will be zero (the log will also start on the first block on the log device). The identity of the log devices is not recorded in the filesystem, but the UUIDs of the filesystem and the log device are compared to prevent corruption.

**sb\_rootino**

Root inode number for the filesystem. Normally, the root inode is at the start of the first possible inode chunk in AG 0. This is 128 when using a 4KB block size.

**sb\_rbmino**

Bitmap inode for real-time extents.

**sb\_rsumino**

Summary inode for real-time bitmap.

**sb\_rextsize**

Realtime extent size in blocks.

**sb\_agblocks**

Size of each AG in blocks. For the actual size of the last AG, refer to the [free space](#) Section 10.2 `agf_length` value.

**sb\_agcount**

Number of AGs in the filesystem.

**sb\_rmblocks**

Number of real-time bitmap blocks.

**sb\_logblocks**

Number of blocks for the journaling log.

**sb\_versionnum**

Filesystem version number. This is a bitmask specifying the features enabled when creating the filesystem. Any disk checking tools or drivers that do not recognize any set bits must not operate upon the filesystem. Most of the flags indicate features introduced over time. If the value of the lower nibble is  $\geq 4$ , the higher bits indicate feature flags as follows:

Table 10.1: Version 4 Superblock version flags

Flag	Description
XFS_SB_VERSION_ATTRBIT	Set if any inode have extended attributes.
XFS_SB_VERSION_NLINKBIT	Set if any inodes use 32-bit <code>di_nlink</code> values.
XFS_SB_VERSION_QUOTABIT	Quotas are enabled on the filesystem. This also brings in the various quota fields in the superblock.
XFS_SB_VERSION_ALIGNBIT	Set if <code>sb_inoalignmt</code> is used.
XFS_SB_VERSION_DALIGNBIT	Set if <code>sb_unit</code> and <code>sb_width</code> are used.
XFS_SB_VERSION_SHAREDBIT	Set if <code>sb_shared_vn</code> is used.
XFS_SB_VERSION_LOGV2BIT	Version 2 journaling logs are used.
XFS_SB_VERSION_SECTORBIT	Set if <code>sb_sectsize</code> is not 512.
XFS_SB_VERSION_EXTFLGBIT	Unwritten extents are used. This is always set.
XFS_SB_VERSION_DIRV2BIT	Version 2 directories are used. This is always set.
XFS_SB_VERSION_MOREBITSBIT	Set if the <code>sb_features2</code> field in the superblock contains more flags.

If the lower nibble of this value is 5, then this is a v5 filesystem; the `XFS_SB_VERSION2_CRCBIT` feature must be set in `sb_features2`.

**sb\_sectsize**

Specifies the underlying disk sector size in bytes. Typically this is 512 or 4096 bytes. This determines the minimum I/O alignment, especially for direct I/O.

**sb\_inodesize**

Size of the inode in bytes. The default is 256 (2 inodes per standard sector) but can be made as large as 2048 bytes when creating the filesystem. On a v5 filesystem, the default and minimum inode size are both 512 bytes.

**sb\_inopblock**

Number of inodes per block. This is equivalent to `sb_blocksize / sb_inodesize`.

**sb\_fname[12]**

Name for the filesystem. This value can be used in the mount command.

**sb\_blocklog**

$\log_2$  value of `sb_blocksize`. In other terms, `sb_blocksize = 2sb_blocklog`.

**sb\_sectlog**

$\log_2$  value of `sb_sectsize`.

**sb\_inodelog**

$\log_2$  value of `sb_inodesize`.

- sb\_inopblog**  
log<sub>2</sub> value of sb\_inopblock.
- sb\_agblklog**  
log<sub>2</sub> value of sb\_agblocks (rounded up). This value is used to generate inode numbers and absolute block numbers defined in extent maps.
- sb\_rextslog**  
log<sub>2</sub> value of sb\_rextents.
- sb\_inprogress**  
Flag specifying that the filesystem is being created.
- sb\_imax\_pct**  
Maximum percentage of filesystem space that can be used for inodes. The default value is 5%.
- sb\_icount**  
Global count for number inodes allocated on the filesystem. This is only maintained in the first superblock.
- sb\_ifree**  
Global count of free inodes on the filesystem. This is only maintained in the first superblock.
- sb\_fdblocks**  
Global count of free data blocks on the filesystem. This is only maintained in the first superblock.
- sb\_frextents**  
Global count of free real-time extents on the filesystem. This is only maintained in the first superblock.
- sb\_uquotino**  
Inode for user quotas. This and the following two quota fields only apply if XFS\_SB\_VERSION\_QUOTABIT flag is set in sb\_versionnum. Refer to [quota inodes](#) Section 12.1 for more information
- sb\_gquotino**  
Inode for group or project quotas. Group and Project quotas cannot be used at the same time.
- sb\_qflags**  
Quota flags. It can be a combination of the following flags:

Table 10.2: Superblock quota flags

Flag	Description
XFS_UQUOTA_ACCT	User quota accounting is enabled.
XFS_UQUOTA_ENFD	User quotas are enforced.
XFS_UQUOTA_CHKD	User quotas have been checked.
XFS_PQUOTA_ACCT	Project quota accounting is enabled.
XFS_OQUOTA_ENFD	Other (group/project) quotas are enforced.
XFS_OQUOTA_CHKD	Other (group/project) quotas have been checked.
XFS_GQUOTA_ACCT	Group quota accounting is enabled.

- sb\_flags**  
Miscellaneous flags.



Table 10.3: Superblock flags

Flag	Description
XFS_SBF_READONLY	Only read-only mounts allowed.

**sb\_shared\_vn**

Reserved and must be zero (“vn” stands for version number).

**sb\_inoalignmt**

Inode chunk alignment in fsblocks. Prior to v5, the default value provided for inode chunks to have an 8KiB alignment. Starting with v5, the default value scales with the multiple of the inode size over 256 bytes. Concretely, this means an alignment of 16KiB for 512-byte inodes, 32KiB for 1024-byte inodes, etc. If sparse inodes are enabled, the `ir_startino` field of each inode B+tree record must be aligned to this block granularity, even if the inode given by `ir_startino` itself is sparse.

**sb\_unit**

Underlying stripe or raid unit in blocks.

**sb\_width**

Underlying stripe or raid width in blocks.

**sb\_dirblklog**

$\log_2$  multiplier that determines the granularity of directory block allocations in fsblocks.

**sb\_logsectlog**

$\log_2$  value of the log subvolume’s sector size. This is only used if the journaling log is on a separate disk device (i.e. not internal).

**sb\_logsectsize**

The log’s sector size in bytes if the filesystem uses an external log device.

**sb\_logsunit**

The log device’s stripe or raid unit size. This only applies to version 2 logs XFS\_SB\_VERSION\_LOGV2BIT is set in `sb_versionnum`.

**sb\_features2**

Additional version flags if XFS\_SB\_VERSION\_MOREBITSBIT is set in `sb_versionnum`. The currently defined additional features include:

Table 10.4: Extended Version 4 Superblock flags

Flag	Description
XFS_SB_VERSION2_LAZYSBCOUNTBIT	Lazy global counters. Making a filesystem with this bit set can improve performance. The global free space and inode counts are only updated in the primary superblock when the filesystem is cleanly unmounted.
XFS_SB_VERSION2_ATTR2BIT	Extended attributes version 2. Making a filesystem with this optimises the inode layout of extended attributes. See the section about <a href="#">extended attribute versions</a> Section 13.4.1 for more information.

Table 10.4: (continued)

Flag	Description
XFS_SB_VERSION2_PARENTBIT	Parent pointers. All inodes must have an extended attribute that points back to its parent inode. The primary purpose for this information is in backup systems.
XFS_SB_VERSION2_PROJID32BIT	32-bit Project ID. Inodes can be associated with a project ID number, which can be used to enforce disk space usage quotas for a particular group of directories. This flag indicates that project IDs can be 32 bits in size.
XFS_SB_VERSION2_CRCBIT	Metadata checksumming. All metadata blocks have an extended header containing the block checksum, a copy of the metadata UUID, the log sequence number of the last update to prevent stale replays, and a back pointer to the owner of the block. This feature must be and can only be set if the lowest nibble of <code>sb_versionnum</code> is set to 5.
XFS_SB_VERSION2_FTYPE	Directory file type. Each directory entry records the type of the inode to which the entry points. This speeds up directory iteration by removing the need to load every inode into memory.

**sb\_bad\_features2**

This field mirrors `sb_features2`, due to past 64-bit alignment errors.

**sb\_features\_compat**

Read-write compatible feature flags. The kernel can still read and write this FS even if it doesn't understand the flag. Currently, there are no valid flags.

**sb\_features\_ro\_compat**

Read-only compatible feature flags. The kernel can still read this FS even if it doesn't understand the flag.

Table 10.5: Extended Version 5 Superblock Read-Only compatibility flags

Flag	Description
XFS_SB_FEAT_RO_COMPAT_FINOBT	Free inode B+tree. Each allocation group contains a B+tree to track inode chunks containing free inodes. This is a performance optimization to reduce the time required to allocate inodes.
XFS_SB_FEAT_RO_COMPAT_RMAPBT	Reverse mapping B+tree. Each allocation group contains a B+tree containing records mapping AG blocks to their owners. See the section about <a href="#">reconstruction</a> Chapter 4 for more details.

Table 10.5: (continued)

Flag	Description
XFS_SB_FEAT_RO_COMPAT_REFLINK	Reference count B+tree. Each allocation group contains a B+tree to track the reference counts of AG blocks. This enables files to share data blocks safely. See the section about <a href="#">reflink and deduplication</a> Chapter 3 for more details.

**sb\_features\_incompat**

Read-write incompatible feature flags. The kernel cannot read or write this FS if it doesn't understand the flag.

Table 10.6: Extended Version 5 Superblock Read-Write incompatibility flags

Flag	Description
XFS_SB_FEAT_INCOMPAT_FTYPE	Directory file type. Each directory entry tracks the type of the inode to which the entry points. This is a performance optimization to remove the need to load every inode into memory to iterate a directory.
XFS_SB_FEAT_INCOMPAT_SPINODES	Sparse inodes. This feature relaxes the requirement to allocate inodes in chunks of 64. When the free space is heavily fragmented, there might exist plenty of free space but not enough contiguous free space to allocate a new inode chunk. With this feature, the user can continue to create files until all free space is exhausted. Unused space in the inode B+tree records are used to track which parts of the inode chunk are not inodes. See the chapter on <a href="#">Sparse Inodes</a> Section 10.5 for more information.
XFS_SB_FEAT_INCOMPAT_META_UUID	Metadata UUID. The UUID stamped into each metadata block must match the value in <code>sb_meta_uuid</code> . This enables the administrator to change <code>sb_uuid</code> at will without having to rewrite the entire filesystem.

**sb\_features\_log\_incompat**

Read-write incompatible feature flags for the log. The kernel cannot read or write this FS log if it doesn't understand the flag. Currently, no flags are defined.

**sb\_crc**

Superblock checksum.

**sb\_spino\_align**

Sparse inode alignment, in fsblocks. Each chunk of inodes referenced by a sparse inode B+tree record must be aligned to this block granularity.

**sb\_pquotino**

Project quota inode.

**sb\_lsn**

Log sequence number of the last superblock update.

**sb\_meta\_uuid**

If the XFS\_SB\_FEAT\_INCOMPAT\_META\_UUID feature is set, then the UUID field in all metadata blocks must match this UUID. If not, the block header UUID field must match `sb_uuid`.

**sb\_rrmapino**

If the XFS\_SB\_FEAT\_RO\_COMPAT\_RMAPBT feature is set and a real-time device is present (`sb_rblocks > 0`), this field points to an inode that contains the root to the [Real-Time Reverse Mapping B+tree](#) Section 12.2.3. This field is zero otherwise.

**xfs\_db Superblock Example**

A filesystem is made on a single disk with the following command:

```
# mkfs.xfs -i attr=2 -n size=16384 -f /dev/sda7
meta-data=/dev/sda7          isize=256    agcount=16, agsize=3923122 blks
      =                       sectsz=512   attr=2
data      =                   bsize=4096  blocks=62769952, imaxpct=25
      =                       sunit=0       swidth=0 blks, unwritten=1
naming    =version 2         bsize=16384
log       =internal log     bsize=4096  blocks=30649, version=1
      =                       sectsz=512   sunit=0 blks
realtime  =none              extsz=65536 blocks=0, rtextents=0
```

And in `xfs_db`, inspecting the superblock:

```
xfs_db> sb
xfs_db> p
magicnum = 0x58465342
blocksize = 4096
dblocks = 62769952
rblocks = 0
rextents = 0
uuid = 32b24036-6931-45b4-b68c-cd5e7d9a1ca5
logstart = 33554436
rootino = 128
rbmino = 129
rsumino = 130
rextsize = 16
agblocks = 3923122
agcount = 16
rbmblocks = 0
logblocks = 30649
versionnum = 0xb084
sectsize = 512
inodesize = 256
inopblock = 16
```

```

fname = "\000\000\000\000\000\000\000\000\000\000\000\000"
blocklog = 12
sectlog = 9
inodeolog = 8
inopblog = 4
agblklog = 22
rextslog = 0
inprogress = 0
imax_pct = 25
icount = 64
ifree = 61
fdblocks = 62739235
frextents = 0
uquotino = 0
gquotino = 0
qflags = 0
flags = 0
shared_vn = 0
inoalignmt = 2
unit = 0
width = 0
dirblklog = 2
logsectlog = 0
logsectsize = 0
logsunit = 0
features2 = 8

```

## AG Free Space Management

The XFS filesystem tracks free space in an allocation group using two B+trees. One B+tree tracks space by block number, the second by the size of the free space block. This scheme allows XFS to find quickly free space near a given block or of a given size.

All block numbers, indexes, and counts are AG relative.

### AG Free Space Block

The second sector in an AG contains the information about the two free space B+trees and associated free space information for the AG. The “AG Free Space Block” also known as the AGF, uses the following structure:

```

struct xfs_agf {
    __be32      agf_magicnum;
    __be32      agf_versionnum;
    __be32      agf_seqno;
    __be32      agf_length;
    __be32      agf_roots[XFS_BTNUM_AGF];
    __be32      agf_levels[XFS_BTNUM_AGF];
    __be32      agf_flfirst;
    __be32      agf_fllast;
    __be32      agf_flcount;
    __be32      agf_freeblks;
    __be32      agf_longest;
    __be32      agf_btreelks;

```

```

/* version 5 filesystem fields start here */
uuid_t          agf_uuid;
__be32          agf_rmap_blocks;
__be32          agf_refcount_blocks;
__be32          agf_refcount_root;
__be32          agf_refcount_level;
__be64          agf_spare64[14];

/* unlogged fields, written during buffer writeback. */
__be64          agf_lsn;
__be32          agf_crc;
__be32          agf_spare2;
};

```

The rest of the bytes in the sector are zeroed. `XFS_BTNUM_AGF` is set to 3: index 0 for the free space B+tree indexed by block number; index 1 for the free space B+tree indexed by extent size; and index 2 for the reverse-mapping B+tree.

#### **agf\_magicnum**

Specifies the magic number for the AGF sector: “XAGF” (0x58414746).

#### **agf\_versionnum**

Set to `XFS_AGF_VERSION` which is currently 1.

#### **agf\_seqno**

Specifies the AG number for the sector.

#### **agf\_length**

Specifies the size of the AG in filesystem blocks. For all AGs except the last, this must be equal to the superblock’s `sb_agblocks` value. For the last AG, this could be less than the `sb_agblocks` value. It is this value that should be used to determine the size of the AG.

#### **agf\_roots**

Specifies the block number for the root of the two free space B+trees and the reverse-mapping B+tree, if enabled.

#### **agf\_levels**

Specifies the level or depth of the two free space B+trees and the reverse-mapping B+tree, if enabled. For a fresh AG, this value will be one, and the “roots” will point to a single leaf of level 0.

#### **agf\_ffirst**

Specifies the index of the first “free list” block. Free lists are covered in more detail later on.

#### **agf\_flast**

Specifies the index of the last “free list” block.

#### **agf\_flcount**

Specifies the number of blocks in the “free list”.

#### **agf\_freeblks**

Specifies the current number of free blocks in the AG.

#### **agf\_longest**

Specifies the number of blocks of longest contiguous free space in the AG.

**agf\_btreeblks**

Specifies the number of blocks used for the free space B+trees. This is only used if the XFS\_SB\_VERSION2\_LAZYSBCOUNTBIT bit is set in `sb_features2`.

**agf\_uuid**

The UUID of this block, which must match either `sb_uuid` or `sb_meta_uuid` depending on which features are set.

**agf\_rmap\_blocks**

The size of the reverse mapping B+tree in this allocation group, in blocks.

**agf\_refcount\_blocks**

The size of the reference count B+tree in this allocation group, in blocks.

**agf\_refcount\_root**

Block number for the root of the reference count B+tree, if enabled.

**agf\_refcount\_root**

Depth of the reference count B+tree, if enabled.

**agf\_spare64**

Empty space in the logged part of the AGF sector, for use for future features.

**agf\_lsn**

Log sequence number of the last AGF write.

**agf\_crc**

Checksum of the AGF sector.

**agf\_spare2**

Empty space in the unlogged part of the AGF sector.

## AG Free Space B+trees

The two Free Space B+trees store a sorted array of block offset and block counts in the leaves of the B+tree. The first B+tree is sorted by the offset, the second by the count or size.

Leaf nodes contain a sorted array of offset/count pairs which are also used for node keys:

```
struct xfs_alloc_rec {
    __be32          ar_startblock;
    __be32          ar_blockcount;
};
```

**ar\_startblock**

AG block number of the start of the free space.

**ar\_blockcount**

Length of the free space.

Node pointers are an AG relative block pointer:

```
typedef __be32 xfs_alloc_ptr_t;
```

- As the free space tracking is AG relative, all the block numbers are only 32-bits.

- The `bb_magic` value depends on the B+tree: “ABTB” (0x41425442) for the block offset B+tree, “ABTC” (0x41425443) for the block count B+tree. On a v5 filesystem, these are “AB3B” (0x41423342) and “AB3C” (0x41423343), respectively.
- The `xfs_btree_sblock_t` header is used for intermediate B+tree node as well as the leaves.
- For a typical 4KB filesystem block size, the offset for the `xfs_alloc_ptr_t` array would be 0xab0 (2736 decimal).
- There are a series of macros in `xfs_btree.h` for deriving the offsets, counts, maximums, etc for the B+trees used in XFS.

The following diagram shows a single level B+tree which consists of one leaf:

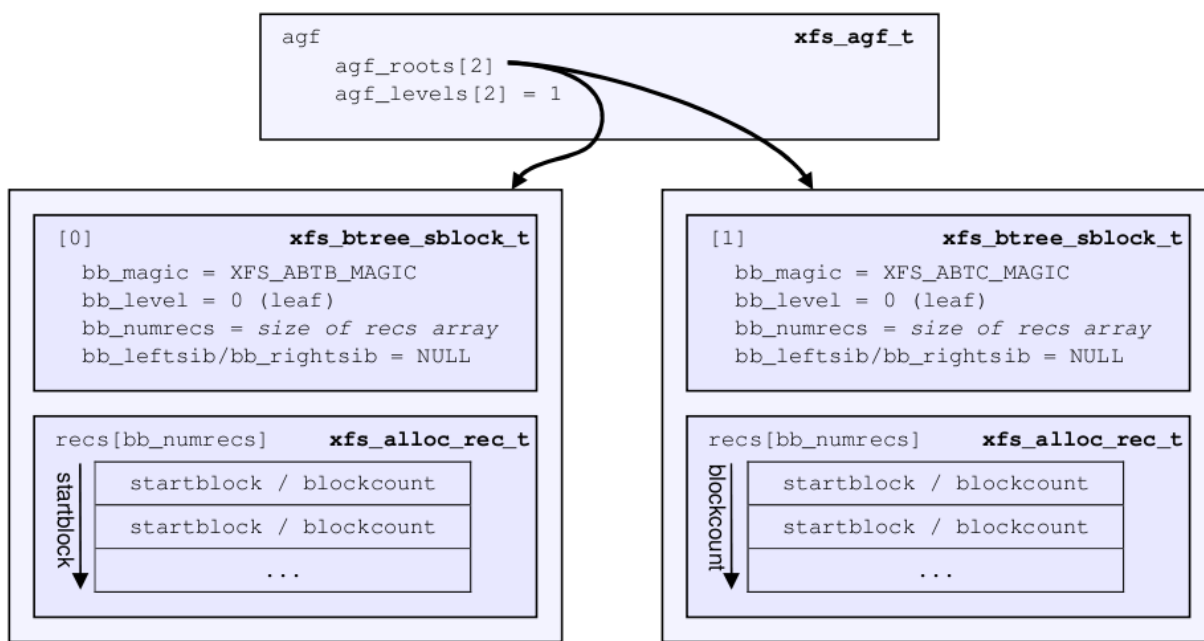


Figure 10.2: Freespace B+tree with one leaf.

With the intermediate nodes, the associated leaf pointers are stored in a separate array about two thirds into the block. The following diagram illustrates a 2-level B+tree for a free space B+tree:



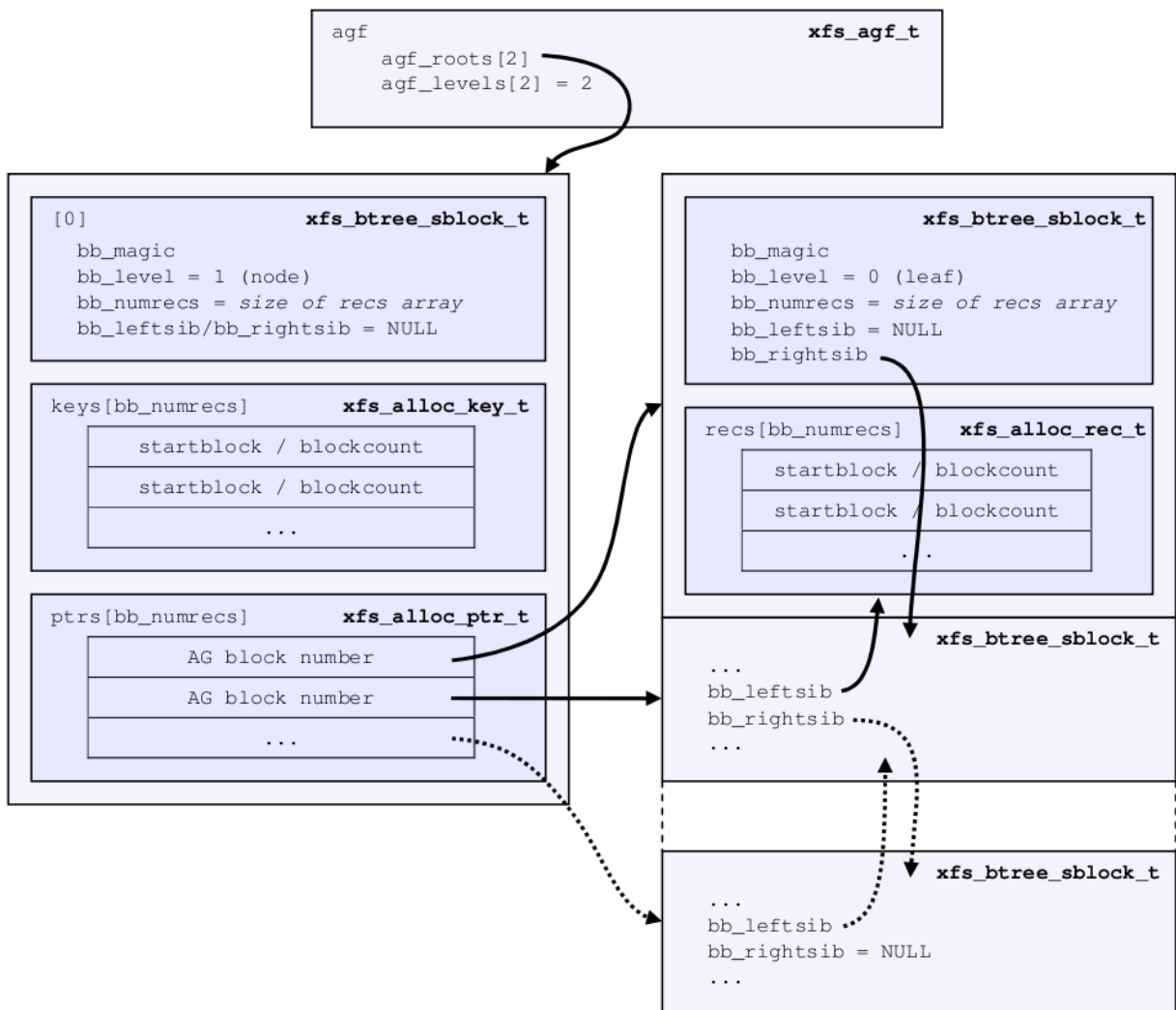


Figure 10.3: Multi-level freespace B+tree.

## AG Free List

The AG Free List is located in the 4<sup>th</sup> sector of each AG and is known as the AGFL. It is an array of AG relative block pointers for reserved space for growing the free space B+trees. This space cannot be used for general user data including inodes, data, directories and extended attributes.

With a freshly made filesystem, 4 blocks are reserved immediately after the free space B+tree root blocks (blocks 4 to 7). As they are used up as the free space fragments, additional blocks will be reserved from the AG and added to the free list array. This size may increase as features are added.

As the free list array is located within a single sector, a typical device will have space for 128 elements in the array (512 bytes per sector, 4 bytes per AG relative block pointer). The actual size can be determined by using the XFS\_A GFL\_SIZE macro.

Active elements in the array are specified by the [AGF's Section 10.2.1](#) `agf_flfirst`, `agf_fllast` and `agf_flcount` values. The array is managed as a circular list.

On a v5 filesystem, the following header precedes the free list entries:

```
struct xfs_agfl {
    __be32          agfl_magicnum;
    __be32          agfl_seqno;
    uuid_t          agfl_uuid;
    __be64          agfl_lsn;
    __be32          agfl_crc;
};
```

**agfl\_magicnum**

Specifies the magic number for the AGFL sector: "XAFL" (0x5841464c).

**agfl\_seqno**

Specifies the AG number for the sector.

**agfl\_uuid**

The UUID of this block, which must match either `sb_uuid` or `sb_meta_uuid` depending on which features are set.

**agfl\_lsn**

Log sequence number of the last AGFL write.

**agfl\_crc**

Checksum of the AGFL sector.

On a v4 filesystem there is no header; the array of free block numbers begins at the beginning of the sector.

---

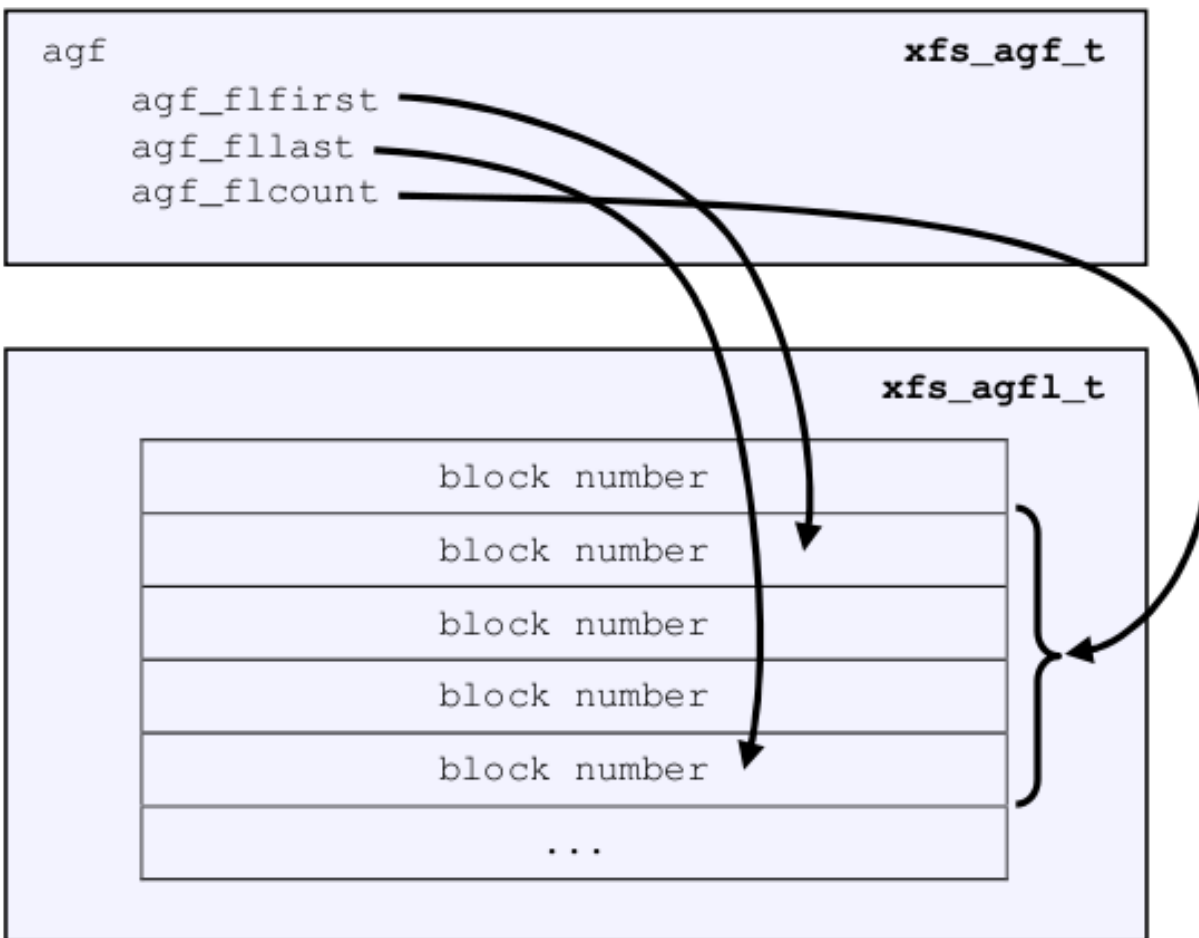


Figure 10.4: AG Free List layout

The presence of these reserved blocks guarantees that the free space B+trees can be updated if any blocks are freed by extent changes in a full AG.

### xfs\_db AGF Example

These examples are derived from an AG that has been deliberately fragmented. The AGF:

```
xfs_db> agf 0
xfs_db> p
magicnum = 0x58414746
versionnum = 1
seqno = 0
length = 3923122
bnoroot = 7
cntroot = 83343
bnolevel = 2
cntlevel = 2
flfirst = 22
```

```

fllast = 27
flcount = 6
freeblks = 3654234
longest = 3384327
btreeblks = 0

```

In the AGFL, the active elements are from 22 to 27 inclusive which are obtained from the `flfirst` and `fllast` values from the `agf` in the previous example:

```

xfs_db> agfl 0
xfs_db> p
bno[0-127] = 0:4 1:5 2:6 3:7 4:83342 5:83343 6:83344 7:83345 8:83346 9:83347
            10:4 11:5 12:80205 13:80780 14:81496 15:81766 16:83346 17:4 18:5
            19:80205 20:82449 21:81496 22:81766 23:82455 24:80780 25:5
            26:80205 27:83344

```

The root block of the free space B+tree sorted by block offset is found in the AGF's `bnoroot` value:

```

xfs_db> fsblock 7
xfs_db> type bnobt
xfs_db> p
magic = 0x41425442
level = 1
numrecs = 4
leftsib = null
rightsib = null
keys[1-4] = [startblock,blockcount]
            1:[12,16] 2:[184586,3] 3:[225579,1] 4:[511629,1]
ptrs[1-4] = 1:2 2:83347 3:6 4:4

```

Blocks 2, 83347, 6 and 4 contain the leaves for the free space B+tree by starting block. Block 2 would contain offsets 12 up to but not including 184586 while block 4 would have all offsets from 511629 to the end of the AG.

The root block of the free space B+tree sorted by block count is found in the AGF's `cntroot` value:

```

xfs_db> fsblock 83343
xfs_db> type cntbt
xfs_db> p
magic = 0x41425443
level = 1
numrecs = 4
leftsib = null
rightsib = null
keys[1-4] = [blockcount,startblock]
            1:[1,81496] 2:[1,511729] 3:[3,191875] 4:[6,184595]
ptrs[1-4] = 1:3 2:83345 3:83342 4:83346

```

The leaf in block 3, in this example, would only contain single block counts. The offsets are sorted in ascending order if the block count is the same.

Inspecting the leaf in block 83346, we can see the largest block at the end:

```

xfs_db> fsblock 83346
xfs_db> type cntbt
xfs_db> p
magic = 0x41425443
level = 0

```

```

numrecs = 344
leftsib = 83342
rightsib = null
recs[1-344] = [startblock,blockcount]
                1:[184595,6] 2:[187573,6] 3:[187776,6]
                ...
                342:[513712,755] 343:[230317,258229] 344:[538795,3384327]

```

The longest block count (3384327) must be the same as the AGF's longest value.

## AG Inode Management

### Inode Numbers

Inode numbers in XFS come in two forms: AG relative and absolute.

AG relative inode numbers always fit within 32 bits. The number of bits actually used is determined by the sum of the [superblock's](#) Section 10.1 `sb_inoplog` and `sb_agblklog` values. Relative inode numbers are found within the AG's inode structures.

Absolute inode numbers include the AG number in the high bits, above the bits used for the AG relative inode number. Absolute inode numbers are found in [directory](#) Chapter 15 entries and the superblock.

#### Relative Inode number format



#### Absolute Inode number format

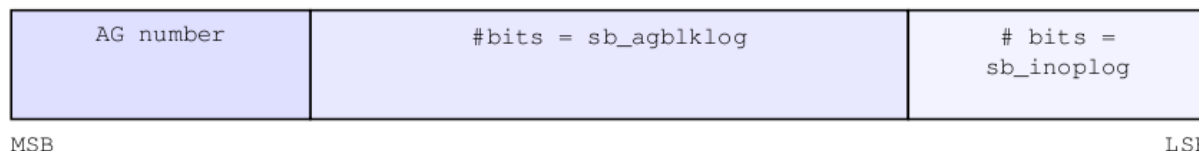


Figure 10.5: Inode number formats

### Inode Information

Each AG manages its own inodes. The third sector in the AG contains information about the AG's inodes and is known as the AGI.

The AGI uses the following structure:

```

struct xfs_agi {
    __be32    agi_magicnum;
    __be32    agi_versionnum;
    __be32    agi_seqno;
    __be32    agi_length;
}

```

```

__be32      agi_count;
__be32      agi_root;
__be32      agi_level;
__be32      agi_freecount;
__be32      agi_newino;
__be32      agi_dirino;
__be32      agi_unlinked[64];

/*
 * v5 filesystem fields start here; this marks the end of logging region 1
 * and start of logging region 2.
 */
uuid_t      agi_uuid;
__be32      agi_crc;
__be32      agi_pad32;
__be64      agi_lsn;

__be32      agi_free_root;
__be32      agi_free_level;
}

```

**agi\_magicnum**

Specifies the magic number for the AGI sector: “XAGI” (0x58414749).

**agi\_versionnum**

Set to XFS\_AGI\_VERSION which is currently 1.

**agi\_seqno**

Specifies the AG number for the sector.

**agi\_length**

Specifies the size of the AG in filesystem blocks.

**agi\_count**

Specifies the number of inodes allocated for the AG.

**agi\_root**

Specifies the block number in the AG containing the root of the inode B+tree.

**agi\_level**

Specifies the number of levels in the inode B+tree.

**agi\_freecount**

Specifies the number of free inodes in the AG.

**agi\_newino**

Specifies AG-relative inode number of the most recently allocated chunk.

**agi\_dirino**

Deprecated and not used, this is always set to NULL (-1).

**agi\_unlinked[64]**

Hash table of unlinked (deleted) inodes that are still being referenced. Refer to [unlinked list pointers](#) Section 13.2 for more information.

**agi\_uuid**

The UUID of this block, which must match either `sb_uuid` or `sb_meta_uuid` depending on which features are set.

**agi\_crc**

Checksum of the AGI sector.

**agi\_pad32**

Padding field, otherwise unused.

**agi\_lsn**

Log sequence number of the last write to this block.

**agi\_free\_root**

Specifies the block number in the AG containing the root of the free inode B+tree.

**agi\_free\_level**

Specifies the number of levels in the free inode B+tree.

## Inode B+trees

Inodes are traditionally allocated in chunks of 64, and a B+tree is used to track these chunks of inodes as they are allocated and freed. The block containing root of the B+tree is defined by the AGI's `agi_root` value. If the `XFS_SB_FEAT_RO_COMPAT_FINOBT` feature is enabled, a second B+tree is used to track the chunks containing free inodes; this is an optimization to speed up inode allocation.

The B+tree header for the nodes and leaves use the `xf_s_btree_sbblock` structure which is the same as the header used in the [AGF B+trees](#) Section 10.2.2.

The magic number of the inode B+tree is "IABT" (0x49414254). On a v5 filesystem, the magic number is "IAB3" (0x49414233).

The magic number of the free inode B+tree is "FIBT" (0x46494254). On a v5 filesystem, the magic number is "FIB3" (0x46494234).

Leaves contain an array of the following structure:

```
struct xfs_inobt_rec {
    __be32          ir_startino;
    __be32          ir_freecount;
    __be64          ir_free;
};
```

**ir\_startino**

The lowest-numbered inode in this chunk.

**ir\_freecount**

Number of free inodes in this chunk.

**ir\_free**

A 64 element bitmap showing which inodes in this chunk are free.

Nodes contain key/pointer pairs using the following types:

```
struct xfs_inobt_key {
    __be32          ir_startino;
};
typedef __be32 xfs_inobt_ptr_t;
```

The following diagram illustrates a single level inode B+tree:

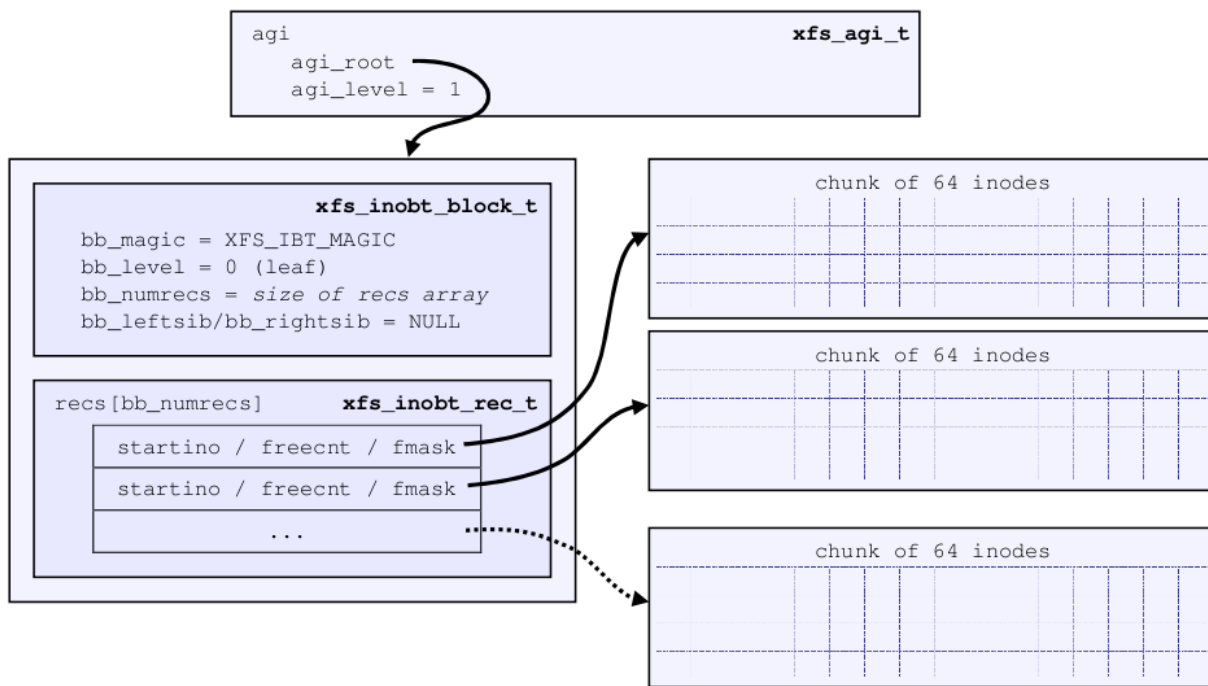


Figure 10.6: Single Level inode B+tree

And a 2-level inode B+tree:



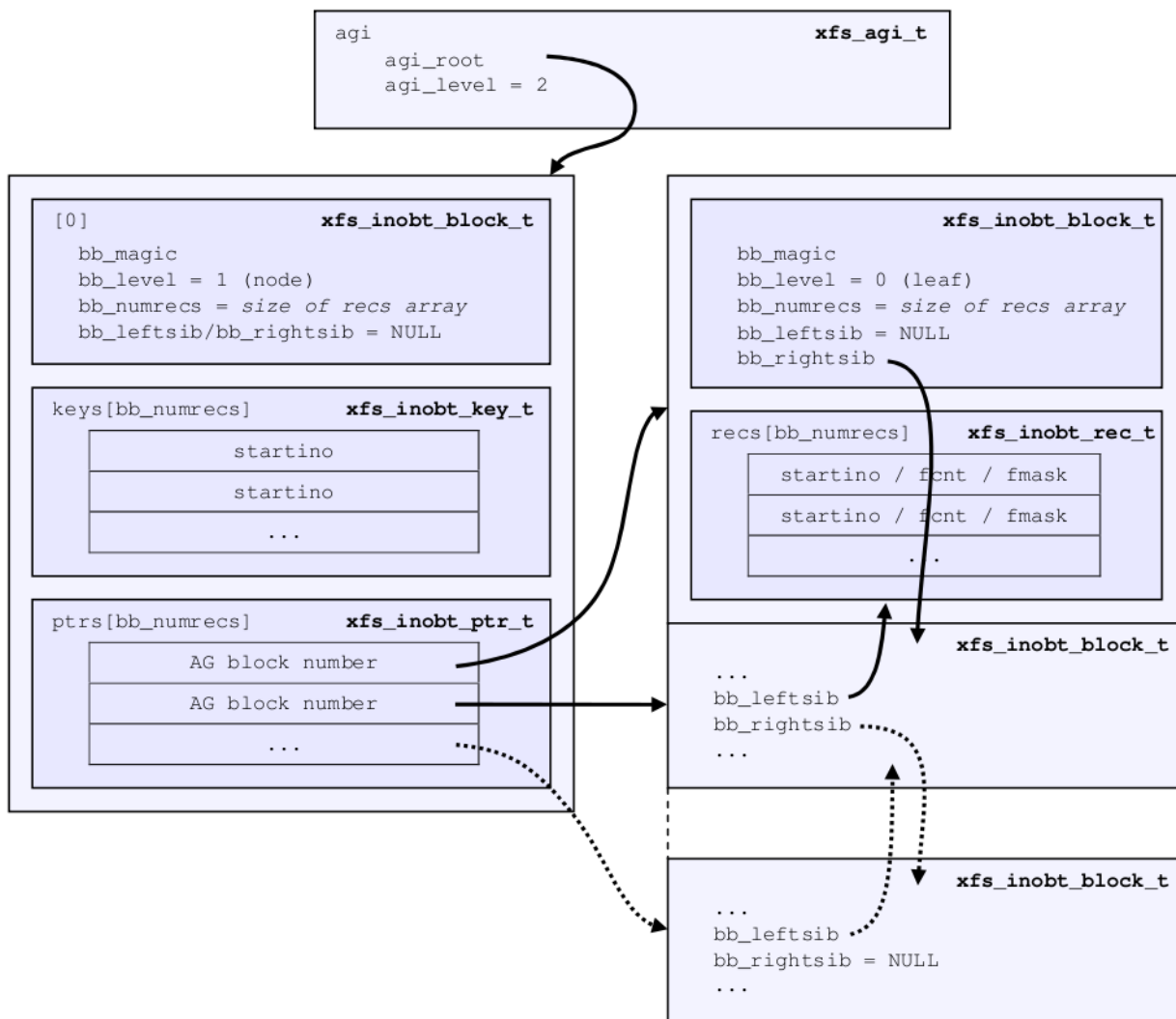


Figure 10.7: Multi-Level inode B+tree

### xfs\_db AGI Example

This is an AGI of a freshly populated filesystem:

```
xfs_db> agi 0
xfs_db> p
magicnum = 0x58414749
versionnum = 1
seqno = 0
length = 825457
count = 5440
root = 3
level = 1
freecount = 9
newino = 5792
```

```

dirino = null
unlinked[0-63] =
uuid = 3dfa1e5c-5a5f-4ca2-829a-000e453600fe
lsn = 0x1000032c2
crc = 0x14cb7e5c (correct)
free_root = 4
free_level = 1

```

From this example, we see that the inode B+tree is rooted at AG block 3 and that the free inode B+tree is rooted at AG block 4. Let's look at the inode B+tree:

```

xfs_db> addr root
xfs_db> p
magic = 0x49414233
level = 0
numrecs = 85
leftsib = null
rightsib = null
bno = 24
lsn = 0x1000032c2
uuid = 3dfa1e5c-5a5f-4ca2-829a-000e453600fe
owner = 0
crc = 0x768f9592 (correct)
recs[1-85] = [startino, freecount, free]
    1:[96,0,0] 2:[160,0,0] 3:[224,0,0] 4:[288,0,0]
    5:[352,0,0] 6:[416,0,0] 7:[480,0,0] 8:[544,0,0]
    9:[608,0,0] 10:[672,0,0] 11:[736,0,0] 12:[800,0,0]
    ...
    85:[5792,9,0xff80000000000000]

```

Most of the inode chunks on this filesystem are totally full, since the `free` value is zero. This means that we ought to expect inode 160 to be linked somewhere in the directory structure. However, notice that `0xff80000000000000` in record 85—this means that we would expect inode 5856 to be free. Moving on to the free inode B+tree, we see that this is indeed the case:

```

xfs_db> addr free_root
xfs_db> p
magic = 0x46494233
level = 0
numrecs = 1
leftsib = null
rightsib = null
bno = 32
lsn = 0x1000032c2
uuid = 3dfa1e5c-5a5f-4ca2-829a-000e453600fe
owner = 0
crc = 0x338af88a (correct)
recs[1] = [startino, freecount, free] 1:[5792,9,0xff80000000000000]

```

Observe also that the AGI's `agi_newino` points to this chunk, which has never been fully allocated.

## Sparse Inodes

As mentioned in the previous section, XFS allocates inodes in chunks of 64. If there are no free extents large enough to hold a full chunk of 64 inodes, the inode allocation fails and XFS claims to have run out of space. On a filesystem

with highly fragmented free space, this can lead to out of space errors long before the filesystem runs out of free blocks.

The sparse inode feature tracks inode chunks in the inode B+tree as if they were full chunks but uses some previously unused bits in the freecount field to track which parts of the inode chunk are not allocated for use as inodes. This allows XFS to allocate inodes one block at a time if absolutely necessary.

The inode and free inode B+trees operate in the same manner as they do without the sparse inode feature; the B+tree header for the nodes and leaves use the `xfs_btree_sblock` structure which is the same as the header used in the [AGF B+trees](#) Section 10.2.2.

It is theoretically possible for a sparse inode B+tree record to reference multiple non-contiguous inode chunks.

Leaves contain an array of the following structure:

```
struct xfs_inobt_rec {
    __be32          ir_startino;
    __be16          ir_holemask;
    __u8            ir_count;
    __u8            ir_freecount;
    __be64          ir_free;
};
```

#### **ir\_startino**

The lowest-numbered inode in this chunk, rounded down to the nearest multiple of 64, even if the start of this chunk is sparse.

#### **ir\_holemask**

A 16 element bitmap showing which parts of the chunk are not allocated to inodes. Each bit represents four inodes; if a bit is marked here, the corresponding bits in `ir_free` must also be marked.

#### **ir\_count**

Number of inodes allocated to this chunk.

#### **ir\_freecount**

Number of free inodes in this chunk.

#### **ir\_free**

A 64 element bitmap showing which inodes in this chunk are not available for allocation.

## **xfs\_db Sparse Inode AGI Example**

This example derives from an AG that has been deliberately fragmented. The inode B+tree:

```
xfs_db> agi 0
xfs_db> p
magicnum = 0x58414749
versionnum = 1
seqno = 0
length = 6400
count = 10432
root = 2381
level = 2
freecount = 0
newino = 14912
dirino = null
```

```

unlinked[0-63] =
uuid = b9b4623b-f678-4d48-8ce7-ce08950e3cd6
lsn = 0x600000ac4
crc = 0xef550dbc (correct)
free_root = 4
free_level = 1

```

This AGI was formatted on a v5 filesystem; notice the extra v5 fields. So far everything else looks much the same as always.

```

xfs_db> addr root
magic = 0x49414233
level = 1
numrecs = 2
leftsib = null
rightsib = null
bno = 19048
lsn = 0x50000192b
uuid = b9b4623b-f678-4d48-8ce7-ce08950e3cd6
owner = 0
crc = 0xd98cd2ca (correct)
keys[1-2] = [startino] 1:[128] 2:[35136]
ptrs[1-2] = 1:3 2:2380
xfs_db> addr ptrs[1]
xfs_db> p
magic = 0x49414233
level = 0
numrecs = 159
leftsib = null
rightsib = 2380
bno = 24
lsn = 0x600000ac4
uuid = b9b4623b-f678-4d48-8ce7-ce08950e3cd6
owner = 0
crc = 0x836768a6 (correct)
recs[1-159] = [startino,holemask,count,freecount,free]
  1:[128,0,64,0,0]
  2:[14912,0xff,32,0,0xffffffff]
  3:[15040,0,64,0,0]
  4:[15168,0xff00,32,0,0xffffffff00000000]
  5:[15296,0,64,0,0]
  6:[15424,0xff,32,0,0xffffffff]
  7:[15552,0,64,0,0]
  8:[15680,0xff00,32,0,0xffffffff00000000]
  9:[15808,0,64,0,0]
 10:[15936,0xff,32,0,0xffffffff]

```

Here we see the difference in the inode B+tree records. For example, in record 2, we see that the holemask has a value of 0xff. This means that the first sixteen inodes in this chunk record do not actually map to inode blocks; the first inode in this chunk is actually inode 14944:

```

xfs_db> inode 14912
Metadata corruption detected at block 0x3a40/0x2000
...
Metadata CRC error detected for ino 14912
xfs_db> p core.magic
core.magic = 0

```

```
xfs_db> inode 14944
xfs_db> p core.magic
core.magic = 0x494e
```

The chunk record also indicates that this chunk has 32 inodes, and that the missing inodes are also “free”.

## Real-time Devices

The performance of the standard XFS allocator varies depending on the internal state of the various metadata indices enabled on the filesystem. For applications which need to minimize the jitter of allocation latency, XFS supports the notion of a “real-time device”. This is a special device separate from the regular filesystem where extent allocations are tracked with a bitmap and free space is indexed with a two-dimensional array. If an inode is flagged with `XFS_DIFLAG_REALTIME`, its data will live on the real time device. The metadata for real time devices is discussed in the section about [real time inodes](#) Section 12.2.

By placing the real time device (and the journal) on separate high-performance storage devices, it is possible to reduce most of the unpredictability in I/O response times that come from metadata operations.

None of the XFS per-AG B+trees are involved with real time files. It is not possible for real time files to share data blocks.

## Reverse-Mapping B+tree

---

### Note

This data structure is under construction! Details may change.

---

If the feature is enabled, each allocation group has its own reverse block-mapping B+tree, which grows in the free space like the free space B+trees. As mentioned in the chapter about [reconstruction](#) Chapter 4, this data structure is another piece of the puzzle necessary to reconstruct the data or attribute fork of a file from reverse-mapping records; we can also use it to double-check allocations to ensure that we are not accidentally cross-linking blocks, which can cause severe damage to the filesystem.

This B+tree is only present if the `XFS_SB_FEAT_RO_COMPAT_RMAPBT` feature is enabled. The feature requires a version 5 filesystem.

Each record in the reverse-mapping B+tree has the following structure:

```
struct xfs_rmap_rec {
    __be32          rm_startblock;
    __be32          rm_blockcount;
    __be64          rm_owner;
    __be64          rm_fork:1;
    __be64          rm_bmbt:1;
    __be64          rm_unwritten:1;
    __be64          rm_unused:7;
    __be64          rm_offset:54;
};
```

### `rm_startblock`

AG block number of this record.

---

**rm\_blockcount**

The length of this extent.

**rm\_owner**

A 64-bit number describing the owner of this extent. This is typically the absolute inode number, but can also correspond to one of the following:

Table 10.7: Special owner values

Value	Description
XFS_RMAP_OWN_NULL	No owner. This should never appear on disk.
XFS_RMAP_OWN_UNKNOWN	Unknown owner; for EFI recovery. This should never appear on disk.
XFS_RMAP_OWN_FS	Allocation group headers
XFS_RMAP_OWN_LOG	XFS log blocks
XFS_RMAP_OWN_AG	Per-allocation group B+tree blocks. This means free space B+tree blocks, blocks on the freelist, and reverse-mapping B+tree blocks.
XFS_RMAP_OWN_INOBT	Per-allocation group inode B+tree blocks. This includes free inode B+tree blocks.
XFS_RMAP_OWN_INODES	Inode chunks
XFS_RMAP_OWN_REFC	Per-allocation group refcount B+tree blocks. This will be used for reflink support.
XFS_RMAP_OWN_COW	Blocks that have been reserved for a copy-on-write operation that has not completed.

**rm\_fork**

If `rm_owner` describes an inode, this can be 1 if this record is for an attribute fork.

**rm\_bmbt**

If `rm_owner` describes an inode, this can be 1 to signify that this record is for a block map B+tree block. In this case, `rm_offset` has no meaning.

**rm\_unwritten**

A flag indicating that the extent is unwritten. This corresponds to the flag in the [extent record](#) Chapter 14 format which means `XFS_EXT_UNWRITTEN`.

**rm\_offset**

The 54-bit logical file block offset, if `rm_owner` describes an inode. Meaningless otherwise.

**Note**

The single-bit flag values `rm_unwritten`, `rm_fork`, and `rm_bmbt` are packed into the larger fields in the C structure definition.

The key has the following structure:

```
struct xfs_rmap_key {
    __be32                rm_startblock;
```

```

__be64          rm_owner;
__be64          rm_fork:1;
__be64          rm_bmbt:1;
__be64          rm_reserved:1;
__be64          rm_unused:7;
__be64          rm_offset:54;
};

```

For the reverse-mapping B+tree on a filesystem that supports sharing of file data blocks, the key definition is larger than the usual AG block number. On a classic XFS filesystem, each block has only one owner, which means that `rm_startblock` is sufficient to uniquely identify each record. However, shared block support (reflink) on XFS breaks that assumption; now filesystem blocks can be linked to any logical block offset of any file inode. Therefore, the key must include the owner and offset information to preserve the 1 to 1 relation between key and record.

- As the reference counting is AG relative, all the block numbers are only 32-bits.
- The `bb_magic` value is "RMB3" (0x524d4233).
- The `xfs_btree_sblock_t` header is used for intermediate B+tree node as well as the leaves.
- Each pointer is associated with two keys. The first of these is the "low key", which is the key of the smallest record accessible through the pointer. This low key has the same meaning as the key in all other btrees. The second key is the high key, which is the maximum of the largest key that can be used to access a given record underneath the pointer. Recall that each record in the reverse mapping b+tree describes an interval of physical blocks mapped to an interval of logical file block offsets; therefore, it makes sense that a range of keys can be used to find to a record.

## xfs\_db rmapbt Example

This example shows a reverse-mapping B+tree from a freshly populated root filesystem:

```

xfs_db> agf 0
xfs_db> addr rmaproot
xfs_db> p
magic = 0x524d4233
level = 1
numrecs = 43
leftsib = null
rightsib = null
bno = 56
lsn = 0x3000004c8
uuid = 1977221d-8345-464e-b1f4-aa2ea36895f4
owner = 0
crc = 0x7cf8be6f (correct)
keys[1-43] = [startblock,owner,offset]
keys[1-43] = [startblock,owner,offset,attrfork,bmbtblock,startblock_hi,owner_hi,
              offset_hi,attrfork_hi,bmbtblock_hi]
  1:[0,-3,0,0,0,351,4418,66,0,0]
  2:[417,285,0,0,0,827,4419,2,0,0]
  3:[829,499,0,0,0,2352,573,55,0,0]
  4:[1292,710,0,0,0,32168,262923,47,0,0]
  5:[32215,-5,0,0,0,34655,2365,3411,0,0]
  6:[34083,1161,0,0,0,34895,265220,1,0,1]
  7:[34896,256191,0,0,0,36522,-9,0,0,0]
  ...
 41:[50998,326734,0,0,0,51430,-5,0,0,0]

```

```

42:[51431,327010,0,0,0,51600,325722,11,0,0]
43:[51611,327112,0,0,0,94063,23522,28375272,0,0]
ptrs[1-43] = 1:5 2:6 3:8 4:9 5:10 6:11 7:418 ... 41:46377 42:48784 43:49522

```

We arbitrarily pick pointer 17 to traverse downwards:

```

xfs_db> addr ptrs[17]
xfs_db> p
magic = 0x524d4233
level = 0
numrecs = 168
leftsib = 36284
rightsib = 37617
bno = 294760
lsn = 0x200002761
uuid = 1977221d-8345-464e-b1f4-aa2ea36895f4
owner = 0
crc = 0x2dad3fbc (correct)
recs[1-168] = [startblock,blockcount,owner,offset,extentflag,attrfork,bmbtblock]
1:[40326,1,259615,0,0,0,0] 2:[40327,1,-5,0,0,0,0]
3:[40328,2,259618,0,0,0,0] 4:[40330,1,259619,0,0,0,0]
...
127:[40540,1,324266,0,0,0,0] 128:[40541,1,324266,8388608,0,0,0]
129:[40542,2,324266,1,0,0,0] 130:[40544,32,-7,0,0,0,0]

```

Several interesting things pop out here. The first record shows that inode 259,615 has mapped AG block 40,326 at offset 0. We confirm this by looking at the block map for that inode:

```

xfs_db> inode 259615
xfs_db> bmap
data offset 0 startblock 40326 (0/40326) count 1 flag 0

```

Next, notice records 127 and 128, which describe neighboring AG blocks that are mapped to non-contiguous logical blocks in inode 324,266. Given the logical offset of 8,388,608 we surmise that this is a leaf directory, but let us confirm:

```

xfs_db> inode 324266
xfs_db> p core.mode
core.mode = 040755
xfs_db> bmap
data offset 0 startblock 40540 (0/40540) count 1 flag 0
data offset 1 startblock 40542 (0/40542) count 2 flag 0
data offset 3 startblock 40576 (0/40576) count 1 flag 0
data offset 8388608 startblock 40541 (0/40541) count 1 flag 0
xfs_db> p core.mode
core.mode = 0100644
xfs_db> dblock 0
xfs_db> p dhdr.hdr.magic
dhdr.hdr.magic = 0x58444433
xfs_db> dblock 8388608
xfs_db> p lhdr.info.hdr.magic
lhdr.info.hdr.magic = 0x3df1

```

Indeed, this inode 324,266 appears to be a leaf directory, as it has regular directory data blocks at low offsets, and a single leaf block.

Notice further the two reverse-mapping records with negative owners. An owner of -7 corresponds to XFS\_RMAP\_P\_OWN\_INODES, which is an inode chunk, and an owner code of -5 corresponds to XFS\_RMAP\_OWN\_AG, which covers free space B+trees and free space. Let's see if block 40,544 is part of an inode chunk:



```
xfs_db> blockget
xfs_db> fsblock 40544
xfs_db> blockuse
block 40544 (0/40544) type inode
xfs_db> stack
1:
    byte offset 166068224, length 4096
    buffer block 324352 (fsbno 40544), 8 bbs
    inode 324266, dir inode 324266, type data
xfs_db> type inode
xfs_db> p
core.magic = 0x494e
```

Our suspicions are confirmed. Let's also see if 40,327 is part of a free space tree:

```
xfs_db> fsblock 40327
xfs_db> blockuse
block 40327 (0/40327) type btrmap
xfs_db> type rmapbt
xfs_db> p
magic = 0x524d4233
```

As you can see, the reverse block-mapping B+tree is an important secondary metadata structure, which can be used to reconstruct damaged primary metadata. Now let's look at an extend rmap btree:

```
xfs_db> agf 0
xfs_db> addr rmaproot
xfs_db> p
magic = 0x34524d42
level = 1
numrecs = 5
leftsib = null
rightsib = null
bno = 6368
lsn = 0x100000d1b
uuid = 400f0928-6b88-4c37-af1e-cef1f8911f3f
owner = 0
crc = 0x8d4ace05 (correct)
keys[1-5] = [startblock,owner,offset,attrfork,bmbtblock,startblock_hi,owner_hi, ←
    offset_hi,attrfork_hi,bmbtblock_hi]
1: [0,-3,0,0,0,705,132,681,0,0]
2: [24,5761,0,0,0,548,5761,524,0,0]
3: [24,5929,0,0,0,380,5929,356,0,0]
4: [24,6097,0,0,0,212,6097,188,0,0]
5: [24,6277,0,0,0,807,-7,0,0,0]
ptrs[1-5] = 1:5 2:771 3:9 4:10 5:11
```

The second pointer stores both the low key [24,5761,0,0,0] and the high key [548,5761,524,0,0], which means that we can expect block 771 to contain records starting at physical block 24, inode 5761, offset zero; and that one of the records can be used to find a reverse mapping for physical block 548, inode 5761, and offset 524:

```
xfs_db> addr ptrs[2]
xfs_db> p
magic = 0x34524d42
level = 0
numrecs = 168
```

```

leftsib = 5
rightsib = 9
bno = 6168
lsn = 0x100000d1b
uuid = 400f0928-6b88-4c37-af1e-cef1f8911f3f
owner = 0
crc = 0xd58eff0e (correct)
recs[1-168] = [startblock,blockcount,owner,offset,extentflag,attrfork,bmbtblock]
1: [24,525,5761,0,0,0,0]
2: [24,524,5762,0,0,0,0]
3: [24,523,5763,0,0,0,0]
...
166: [24,360,5926,0,0,0,0]
167: [24,359,5927,0,0,0,0]
168: [24,358,5928,0,0,0,0]

```

Observe that the first record in the block starts at physical block 24, inode 5761, offset zero, just as we expected. Note that this first record is also indexed by the highest key as provided in the node block; physical block 548, inode 5761, offset 524 is the very last block mapped by this record. Furthermore, note that record 168, despite being the last record in this block, has a lower maximum key (physical block 382, inode 5928, offset 23) than the first record.

## Reference Count B+tree

---

### Note

This data structure is under construction! Details may change.

---

To support the sharing of file data blocks (reflink), each allocation group has its own reference count B+tree, which grows in the allocated space like the inode B+trees. This data could be gleaned by performing an interval query of the reverse-mapping B+tree, but doing so would come at a huge performance penalty. Therefore, this data structure is a cache of computable information.

This B+tree is only present if the `XFS_SB_FEAT_RO_COMPAT_REFLINK` feature is enabled. The feature requires a version 5 filesystem.

Each record in the reference count B+tree has the following structure:

```

struct xfs_refcount_rec {
    __be32          rc_startblock;
    __be32          rc_blockcount;
    __be32          rc_refcount;
};

```

#### `rc_startblock`

AG block number of this record.

#### `rc_blockcount`

The length of this extent.

#### `rc_refcount`

Number of mappings of this filesystem extent.

Node pointers are an AG relative block pointer:

---

```
struct xfs_refcount_key {
    __be32                rc_startblock;
};
```

- As the reference counting is AG relative, all the block numbers are only 32-bits.
- The `bb_magic` value is "R3FC" (0x52334643).
- The `xfs_btree_sblock_t` header is used for intermediate B+tree node as well as the leaves.

### xfs\_db refcntbt Example

For this example, an XFS filesystem was populated with a root filesystem and a deduplication program was run to create shared blocks:

```
xfs_db> agf 0
xfs_db> addr refcntroot
xfs_db> p
magic = 0x52334643
level = 1
numrecs = 6
leftsib = null
rightsib = null
bno = 36892
lsn = 0x200004ec2
uuid = f1f89746-e00b-49c9-96b3-ecef0f2f14ae
owner = 0
crc = 0x75f35128 (correct)
keys[1-6] = [startblock] 1:[14] 2:[65633] 3:[65780] 4:[94571] 5:[117201] 6:[152442]
ptrs[1-6] = 1:7 2:25836 3:25835 4:18447 5:18445 6:18449
xfs_db> addr ptrs[3]
xfs_db> p
magic = 0x52334643
level = 0
numrecs = 80
leftsib = 25836
rightsib = 18447
bno = 51670
lsn = 0x200004ec2
uuid = f1f89746-e00b-49c9-96b3-ecef0f2f14ae
owner = 0
crc = 0xc3962813 (correct)
recs[1-80] = [startblock,blockcount,refcount]
          1:[65780,1,2] 2:[65781,1,3] 3:[65785,2,2] 4:[66640,1,2]
          5:[69602,4,2] 6:[72256,16,2] 7:[72871,4,2] 8:[72879,20,2]
          9:[73395,4,2] 10:[75063,4,2] 11:[79093,4,2] 12:[86344,16,2]
```

Record 6 in the reference count B+tree for AG 0 indicates that the AG extent starting at block 72,256 and running for 16 blocks has a reference count of 2. This means that there are two files sharing the block:

```
xfs_db> blockget -n
xfs_db> fsblock 72256
xfs_db> blockuse
block 72256 (0/72256) type rldata inode 25169197
```

The blockuse type changes to “rldata” to indicate that the block is shared data. Unfortunately, blockuse only tells us about one block owner. If we happen to have enabled the reverse-mapping B+tree, we can use it to find all inodes that own this block:

```
xfs_db> agf 0
xfs_db> addr rmaproot
...
xfs_db> addr ptrs[3]
...
xfs_db> addr ptrs[7]
xfs_db> p
magic = 0x524d4233
level = 0
numrecs = 22
leftsib = 65057
rightsib = 65058
bno = 291478
lsn = 0x200004ec2
uuid = f1f89746-e00b-49c9-96b3-ecef0f2f14ae
owner = 0
crc = 0xed7da3f7 (correct)
recs[1-22] = [startblock,blockcount,owner,offset,extentflag,attrfork,bmbtblock]
              1:[68957,8,3201,0,0,0,0] 2:[68965,4,25260953,0,0,0,0]
              ...
              18:[72232,58,3227,0,0,0,0] 19:[72256,16,25169197,24,0,0,0]
              20:[72290,75,3228,0,0,0,0] 21:[72365,46,3229,0,0,0,0]
```

Records 18 and 19 intersect the block 72,256; they tell us that inodes 3,227 and 25,169,197 both claim ownership. Let us confirm this:

```
xfs_db> inode 25169197
xfs_db> bmap
data offset 0 startblock 12632259 (3/49347) count 24 flag 0
data offset 24 startblock 72256 (0/72256) count 16 flag 0
data offset 40 startblock 12632299 (3/49387) count 18 flag 0
xfs_db> inode 3227
xfs_db> bmap
data offset 0 startblock 72232 (0/72232) count 58 flag 0
```

Inodes 25,169,197 and 3,227 both contain mappings to block 0/72,256.

## Chapter 11

# Journaling Log

---

**Note**

Only v2 log format is covered here.

---

The XFS journal exists on disk as a reserved extent of blocks within the filesystem, or as a separate journal device. The journal itself can be thought of as a series of log records; each log record contains a part of or a whole transaction. A transaction consists of a series of log operation headers (“log items”), formatting structures, and raw data. The first operation in a transaction establishes the transaction ID and the last operation is a commit record. The operations recorded between the start and commit operations represent the metadata changes made by the transaction. If the commit operation is missing, the transaction is incomplete and cannot be recovered.

## Log Records

The XFS log is split into a series of log records. Log records seem to correspond to an in-core log buffer, which can be up to 256KiB in size. Each record has a log sequence number, which is the same LSN recorded in the v5 metadata integrity fields.

Log sequence numbers are a 64-bit quantity consisting of two 32-bit quantities. The upper 32 bits are the “cycle number”, which increments every time XFS cycles through the log. The lower 32 bits are the “block number”, which is assigned when a transaction is committed, and should correspond to the block offset within the log.

A log record begins with the following header, which occupies 512 bytes on disk:

```
typedef struct xlog_rec_header {
    __be32          h_magicno;
    __be32          h_cycle;
    __be32          h_version;
    __be32          h_len;
    __be64          h_lsn;
    __be64          h_tail_lsn;
    __le32          h_crc;
    __be32          h_prev_block;
    __be32          h_num_logops;
    __be32          h_cycle_data[XLOG_HEADER_CYCLE_SIZE / BBSIZE];
    /* new fields */
    __be32          h_fmt;
```

```

    uuid_t          h_fs_uuid;
    __be32         h_size;
} xlog_rec_header_t;

```

**h\_magicno**

The magic number of log records, 0xfeedbabe.

**h\_cycle**

Cycle number of this log record.

**h\_version**

Log record version, currently 2.

**h\_len**

Length of the log record, in bytes. Must be aligned to a 64-bit boundary.

**h\_lsn**

Log sequence number of this record.

**h\_tail\_lsn**

Log sequence number of the first log record with uncommitted buffers.

**h\_crc**

Checksum of the log record header, the cycle data, and the log records themselves.

**h\_prev\_block**

Block number of the previous log record.

**h\_num\_logops**

The number of log operations in this record.

**h\_cycle\_data**

The first u32 of each log sector must contain the cycle number. Since log item buffers are formatted without regard to this requirement, the original contents of the first four bytes of each sector in the log are copied into the corresponding element of this array. After that, the first four bytes of those sectors are stamped with the cycle number. This process is reversed at recovery time. If there are more sectors in this log record than there are slots in this array, the cycle data continues for as many sectors are needed; each sector is formatted as type `xlog_rec_ext_header`.

**h\_fmt**

Format of the log record. This is one of the following values:

Table 11.1: Log record formats

Format value	Log format
XLOG_FMT_UNKNOWN	Unknown. Perhaps this log is corrupt.
XLOG_FMT_LINUX_LE	Little-endian Linux.
XLOG_FMT_LINUX_BE	Big-endian Linux.
XLOG_FMT_IRIX_BE	Big-endian Irix.

**h\_fs\_uuid**

Filesystem UUID.

#### **h\_size**

In-core log record size. This is somewhere between 16 and 256KiB, with 32KiB being the default.

As mentioned earlier, if this log record is longer than 256 sectors, the cycle data overflows into the next sector(s) in the log. Each of those sectors is formatted as follows:

```
typedef struct xlog_rec_ext_header {
    __be32          xh_cycle;
    __be32          xh_cycle_data[XLOG_HEADER_CYCLE_SIZE / BBSIZE];
} xlog_rec_ext_header_t;
```

#### **xh\_cycle**

Cycle number of this log record. Should match `h_cycle`.

#### **xh\_cycle\_data**

Overflow cycle data.

## Log Operations

Within a log record, log operations are recorded as a series consisting of an operation header immediately followed by a data region. The operation header has the following format:

```
typedef struct xlog_op_header {
    __be32          oh_tid;
    __be32          oh_len;
    __u8           oh_clientid;
    __u8           oh_flags;
    __u16          oh_res2;
} xlog_op_header_t;
```

#### **oh\_tid**

Transaction ID of this operation.

#### **oh\_len**

Number of bytes in the data region.

#### **oh\_clientid**

The originator of this operation. This can be one of the following:

Table 11.2: Log Operation Client ID

Client ID	Originator
XFS_TRANSACTION	Operation came from a transaction.
XFS_VOLUME	???
XFS_LOG	???

**oh\_flags**

Specifies flags associated with this operation. This can be a combination of the following values (though most likely only one will be set at a time):

Table 11.3: Log Operation Flags

Flag	Description
XLOG_START_TRANS	Start a new transaction. The next operation header should describe a transaction header.
XLOG_COMMIT_TRANS	Commit this transaction.
XLOG_CONTINUE_TRANS	Continue this trans into new log record.
XLOG_WAS_CONT_TRANS	This transaction started in a previous log record.
XLOG_END_TRANS	End of a continued transaction.
XLOG_UNMOUNT_TRANS	Transaction to unmount a filesystem.

**oh\_res2**

Padding.

The data region follows immediately after the operation header and is exactly `oh_len` bytes long. These payloads are in host-endian order, which means that one cannot replay the log from an unclean XFS filesystem on a system with a different byte order.

## Log Items

Following are the types of log item payloads that can follow an `xlog_op_header`. Except for buffer data and inode cores, all log items have a magic number to distinguish themselves. Buffer data items only appear after `xfs_buf_log_format` items; and inode core items only appear after `xfs_inode_log_format` items.

Table 11.4: Log Operation Magic Numbers

Magic	Hexadecimal	Operation Type
XFS_TRANS_HEADER_MAGIC	0x5452414e	<a href="#">Log Transaction Header</a> Section 11.3.1
XFS_LI_EFI	0x1236	<a href="#">Extent Freeing Intent</a> Section 11.3.2
XFS_LI_EFD	0x1237	<a href="#">Extent Freeing Done</a> Section 11.3.3
XFS_LI_IUNLINK	0x1238	Unknown?
XFS_LI_INODE	0x123b	<a href="#">Inode Updates</a> Section 11.3.10
XFS_LI_BUF	0x123c	<a href="#">Buffer Writes</a> Section 11.3.12
XFS_LI_DQUOT	0x123d	<a href="#">Update Quota</a> Section 11.3.14
XFS_LI_QUOTAOFF	0x123e	<a href="#">Quota Off</a> Section 11.3.16
XFS_LI_ICREATE	0x123f	<a href="#">Inode Creation</a> Section 11.3.17
XFS_LI_RUI	0x1240	<a href="#">Reverse Mapping Update Intent</a> Section 11.3.4



Table 11.4: (continued)

Magic	Hexadecimal	Operation Type
XFS_LI_RUD	0x1241	<a href="#">Reverse Mapping Update Done</a> Section 11.3.5
XFS_LI_CUI	0x1242	<a href="#">Reference Count Update Intent</a> Section 11.3.6
XFS_LI_CUD	0x1243	<a href="#">Reference Count Update Done</a> Section 11.3.7
XFS_LI_BUI	0x1244	<a href="#">File Block Mapping Update Intent</a> Section 11.3.8
XFS_LI_BUD	0x1245	<a href="#">File Block Mapping Update Done</a> Section 11.3.9

## Transaction Headers

A transaction header is an operation payload that starts a transaction.

```
typedef struct xfs_trans_header {
    uint          th_magic;
    uint          th_type;
    __int32_t     th_tid;
    uint          th_num_items;
} xfs_trans_header_t;
```

### th\_magic

The signature of a transaction header, “TRAN” (0x5452414e). Note that this value is in host-endian order, not big-endian like the rest of XFS.

### th\_type

Transaction type. This is one of the following values:

Type	Description
XFS_TRANS_SETATTR_NOT_SIZE	Set an inode attribute that isn’t the inode’s size.
XFS_TRANS_SETATTR_SIZE	Setting the size attribute of an inode.
XFS_TRANS_INACTIVE	Freeing blocks from an unlinked inode.
XFS_TRANS_CREATE	Create a file.
XFS_TRANS_CREATE_TRUNC	Unused?
XFS_TRANS_TRUNCATE_FILE	Truncate a quota file.
XFS_TRANS_REMOVE	Remove a file.
XFS_TRANS_LINK	Link an inode into a directory.
XFS_TRANS_RENAME	Rename a path.
XFS_TRANS_MKDIR	Create a directory.
XFS_TRANS_RMDIR	Remove a directory.
XFS_TRANS_SYMLINK	Create a symbolic link.
XFS_TRANS_SET_DMATTRS	Set the DMAPI attributes of an inode.
XFS_TRANS_GROWFS	Expand the filesystem.
XFS_TRANS_STRAT_WRITE	Convert an unwritten extent or delayed-allocate some blocks to handle a write.

Type	Description
XFS_TRANS_DIOSTRAT	Allocate some blocks to handle a direct I/O write.
XFS_TRANS_WRITEID	Update an inode's preallocation flag.
XFS_TRANS_ADDAFORK	Add an attribute fork to an inode.
XFS_TRANS_ATTRINVAL	Erase the attribute fork of an inode.
XFS_TRANS_ATRUNCATE	Unused?
XFS_TRANS_ATTR_SET	Set an extended attribute.
XFS_TRANS_ATTR_RM	Remove an extended attribute.
XFS_TRANS_ATTR_FLAG	Unused?
XFS_TRANS_CLEAR_AGI_BUCKET	Clear a bad inode pointer in the AGI unlinked inode hash bucket.
XFS_TRANS_SB_CHANGE	Write the superblock to disk.
XFS_TRANS_QM_QUOTAOFF	Start disabling quotas.
XFS_TRANS_QM_DQALLOC	Allocate a disk quota structure.
XFS_TRANS_QM_SETQLIM	Adjust quota limits.
XFS_TRANS_QM_DQCLUSTER	Unused?
XFS_TRANS_QM_QINOCREATE	Create a (quota) inode with reference taken.
XFS_TRANS_QM_QUOTAOFF_END	Finish disabling quotas.
XFS_TRANS_FSYNC_TS	Update only inode timestamps.
XFS_TRANS_GROWFSRT_ALLOC	Grow the realtime bitmap and summary data for growfs.
XFS_TRANS_GROWFSRT_ZERO	Zero space in the realtime bitmap and summary data.
XFS_TRANS_GROWFSRT_FREE	Free space in the realtime bitmap and summary data.
XFS_TRANS_SWAPEXT	Swap data fork of two inodes.
XFS_TRANS_CHECKPOINT	Checkpoint the log.
XFS_TRANS_ICREATE	Unknown?
XFS_TRANS_CREATE_TMPFILE	Create a temporary file.

**th\_tid**

Transaction ID.

**th\_num\_items**

The number of operations appearing after this operation, not including the commit operation. In effect, this tracks the number of metadata change operations in this transaction.

**Intent to Free an Extent**

The next two operation types work together to handle the freeing of filesystem blocks. Naturally, the ranges of blocks to be freed can be expressed in terms of extents:

```
typedef struct xfs_extent_32 {
    __uint64_t      ext_start;
    __uint32_t      ext_len;
} __attribute__((packed)) xfs_extent_32_t;

typedef struct xfs_extent_64 {
    __uint64_t      ext_start;
    __uint32_t      ext_len;
    __uint32_t      ext_pad;
} xfs_extent_64_t;
```

**ext\_start**

Start block of this extent.

**ext\_len**

Length of this extent.

The “extent freeing intent” operation comes first; it tells the log that XFS wants to free some extents. This record is crucial for correct log recovery because it prevents the log from replaying blocks that are subsequently freed. If the log lacks a corresponding “extent freeing done” operation, the recovery process will free the extents.

```
typedef struct xfs_efi_log_format {
    __uint16_t      efi_type;
    __uint16_t      efi_size;
    __uint32_t      efi_nextents;
    __uint64_t      efi_id;
    xfs_extent_t    efi_extents[1];
} xfs_efi_log_format_t;
```

**efi\_type**

The signature of an EFI operation, 0x1236. This value is in host-endian order, not big-endian like the rest of XFS.

**efi\_size**

Size of this log item. Should be 1.

**efi\_nextents**

Number of extents to free.

**efi\_id**

A 64-bit number that binds the corresponding EFD log item to this EFI log item.

**efi\_extents**

Variable-length array of extents to be freed. The array length is given by `efi_nextents`. The record type will be either `xfs_extent_64_t` or `xfs_extent_32_t`; this can be determined from the log item size (`oh_len`) and the number of extents (`efi_nextents`).

## Completion of Intent to Free an Extent

The “extent freeing done” operation complements the “extent freeing intent” operation. This second operation indicates that the block freeing actually happened, so that log recovery needn’t try to free the blocks. Typically, the operations to update the free space B+trees follow immediately after the EFD.

```
typedef struct xfs_efd_log_format {
    __uint16_t      efd_type;
    __uint16_t      efd_size;
    __uint32_t      efd_nextents;
    __uint64_t      efd_efi_id;
    xfs_extent_t    efd_extents[1];
} xfs_efd_log_format_t;
```

**efd\_type**

The signature of an EFD operation, 0x1237. This value is in host-endian order, not big-endian like the rest of XFS.

**efd\_size**

Size of this log item. Should be 1.

**efd\_nextents**

Number of extents to free.

**efd\_id**

A 64-bit number that binds the corresponding EFI log item to this EFD log item.

**efd\_extents**

Variable-length array of extents to be freed. The array length is given by `efd_nextents`. The record type will be either `xfs_extent_64_t` or `xfs_extent_32_t`; this can be determined from the log item size (`oh_len`) and the number of extents (`efd_nextents`).

## Reverse Mapping Updates Intent

The next two operation types work together to handle deferred reverse mapping updates. Naturally, the mappings to be updated can be expressed in terms of mapping extents:

```
struct xfs_map_extent {
    __uint64_t      me_owner;
    __uint64_t      me_startblock;
    __uint64_t      me_startoff;
    __uint32_t      me_len;
    __uint32_t      me_flags;
};
```

**me\_owner**

Owner of this reverse mapping. See the values in the section about [reverse mapping](#) Section 10.7 for more information.

**me\_startblock**

Filesystem block of this mapping.

**me\_startoff**

Logical block offset of this mapping.

**me\_len**

The length of this mapping.

**me\_flags**

The lower byte of this field is a type code indicating what sort of reverse mapping operation we want. The upper three bytes are flag bits.

Table 11.5: Reverse mapping update log intent types

Value	Description
XFS_RMAP_EXTENT_MAP	Add a reverse mapping for file data.
XFS_RMAP_EXTENT_MAP_SHARED	Add a reverse mapping for file data for a file with shared blocks.
XFS_RMAP_EXTENT_UNMAP	Remove a reverse mapping for file data.
XFS_RMAP_EXTENT_UNMAP_SHARED	Remove a reverse mapping for file data for a file with shared blocks.

Table 11.5: (continued)

Value	Description
XFS_RMAP_EXTENT_CONVERT	Convert a reverse mapping for file data between unwritten and normal.
XFS_RMAP_EXTENT_CONVERT_SHARED	Convert a reverse mapping for file data between unwritten and normal for a file with shared blocks.
XFS_RMAP_EXTENT_ALLOC	Add a reverse mapping for non-file data.
XFS_RMAP_EXTENT_FREE	Remove a reverse mapping for non-file data.

Table 11.6: Reverse mapping update log intent flags

Value	Description
XFS_RMAP_EXTENT_ATTR_FORK	Extent is for the attribute fork.
XFS_RMAP_EXTENT_BMBT_BLOCK	Extent is for a block mapping btree block.
XFS_RMAP_EXTENT_UNWRITTEN	Extent is unwritten.

The “rmap update intent” operation comes first; it tells the log that XFS wants to update some reverse mappings. This record is crucial for correct log recovery because it enables us to spread a complex metadata update across multiple transactions while ensuring that a crash midway through the complex update will be replayed fully during log recovery.

```
struct xfs_rui_log_format {
    __uint16_t    rui_type;
    __uint16_t    rui_size;
    __uint32_t    rui_nextents;
    __uint64_t    rui_id;
    struct xfs_map_extent    rui_extents[1];
};
```

**rui\_type**

The signature of an RUI operation, 0x1240. This value is in host-endian order, not big-endian like the rest of XFS.

**rui\_size**

Size of this log item. Should be 1.

**rui\_nextents**

Number of reverse mappings.

**rui\_id**

A 64-bit number that binds the corresponding RUD log item to this RUI log item.

**rui\_extents**

Variable-length array of reverse mappings to update.

## Completion of Reverse Mapping Updates

The “reverse mapping update done” operation complements the “reverse mapping update intent” operation. This second operation indicates that the update actually happened, so that log recovery needn’t replay the update. The RUD and the actual updates are typically found in a new transaction following the transaction in which the RUI was logged.

```
struct xfs_rud_log_format {
    __uint16_t      rud_type;
    __uint16_t      rud_size;
    __uint32_t      __pad;
    __uint64_t      rud_rui_id;
};
```

### **rud\_type**

The signature of an RUD operation, 0x1241. This value is in host-endian order, not big-endian like the rest of XFS.

### **rud\_size**

Size of this log item. Should be 1.

### **rud\_rui\_id**

A 64-bit number that binds the corresponding RUI log item to this RUD log item.

## Reference Count Updates Intent

The next two operation types work together to handle reference count updates. Naturally, the ranges of extents having reference count updates can be expressed in terms of physical extents:

```
struct xfs_phys_extent {
    __uint64_t      pe_startblock;
    __uint32_t      pe_len;
    __uint32_t      pe_flags;
};
```

### **pe\_startblock**

Filesystem block of this extent.

### **pe\_len**

The length of this extent.

### **pe\_flags**

The lower byte of this field is a type code indicating what sort of reverse mapping operation we want. The upper three bytes are flag bits.

Table 11.7: Reference count update log intent types

Value	Description
XFS_REFCOUNT_EXTENT_INCREASE	Increase the reference count for this extent.
XFS_REFCOUNT_EXTENT_DECREASE	Decrease the reference count for this extent.
XFS_REFCOUNT_EXTENT_ALLOC_COW	Reserve an extent for staging copy on write.
XFS_REFCOUNT_EXTENT_FREE_COW	Unreserve an extent for staging copy on write.

The “reference count update intent” operation comes first; it tells the log that XFS wants to update some reference counts. This record is crucial for correct log recovery because it enables us to spread a complex metadata update across multiple transactions while ensuring that a crash midway through the complex update will be replayed fully during log recovery.

```
struct xfs_cui_log_format {
    __uint16_t      cui_type;
    __uint16_t      cui_size;
    __uint32_t      cui_nextents;
    __uint64_t      cui_id;
    struct xfs_map_extent  cui_extents[1];
};
```

**cui\_type**

The signature of an CUI operation, 0x1242. This value is in host-endian order, not big-endian like the rest of XFS.

**cui\_size**

Size of this log item. Should be 1.

**cui\_nextents**

Number of reference count updates.

**cui\_id**

A 64-bit number that binds the corresponding RUD log item to this RUI log item.

**cui\_extents**

Variable-length array of reference count update information.

## Completion of Reference Count Updates

The “reference count update done” operation complements the “reference count update intent” operation. This second operation indicates that the update actually happened, so that log recovery needn’t replay the update. The CUD and the actual updates are typically found in a new transaction following the transaction in which the CUI was logged.

```
struct xfs_cud_log_format {
    __uint16_t      cud_type;
    __uint16_t      cud_size;
    __uint32_t      __pad;
    __uint64_t      cud_cui_id;
};
```

**cud\_type**

The signature of an RUD operation, 0x1243. This value is in host-endian order, not big-endian like the rest of XFS.

**cud\_size**

Size of this log item. Should be 1.

**cud\_cui\_id**

A 64-bit number that binds the corresponding CUI log item to this CUD log item.

---

## File Block Mapping Intent

The next two operation types work together to handle deferred file block mapping updates. The extents to be mapped are expressed via the `xfs_map_extent` structure discussed in the section about [reverse mapping intents](#) Section 11.3.4.

The lower byte of the `me_flags` field is a type code indicating what sort of file block mapping operation we want. The upper three bytes are flag bits.

Table 11.8: File block mapping update log intent types

Value	Description
<code>XFS_BMAP_EXTENT_MAP</code>	Add a mapping for file data.
<code>XFS_BMAP_EXTENT_UNMAP</code>	Remove a mapping for file data.

Table 11.9: File block mapping update log intent flags

Value	Description
<code>XFS_BMAP_EXTENT_ATTR_FORK</code>	Extent is for the attribute fork.
<code>XFS_BMAP_EXTENT_UNWRITTEN</code>	Extent is unwritten.

The “file block mapping update intent” operation comes first; it tells the log that XFS wants to map or unmap some extents in a file. This record is crucial for correct log recovery because it enables us to spread a complex metadata update across multiple transactions while ensuring that a crash midway through the complex update will be replayed fully during log recovery.

```
struct xfs_bui_log_format {
    __uint16_t      bui_type;
    __uint16_t      bui_size;
    __uint32_t      bui_nextents;
    __uint64_t      bui_id;
    struct xfs_map_extent bui_extents[1];
};
```

### **bui\_type**

The signature of an BUI operation, 0x1244. This value is in host-endian order, not big-endian like the rest of XFS.

### **bui\_size**

Size of this log item. Should be 1.

### **bui\_nextents**

Number of file mappings. Should be 1.

### **bui\_id**

A 64-bit number that binds the corresponding BUD log item to this BUI log item.

### **bui\_extents**

Variable-length array of file block mappings to update. There should only be one mapping present.



## Completion of File Block Mapping Updates

The “file block mapping update done” operation complements the “file block mapping update intent” operation. This second operation indicates that the update actually happened, so that log recovery needn’t replay the update. The BUD and the actual updates are typically found in a new transaction following the transaction in which the BUI was logged.

```
struct xfs_bud_log_format {
    __uint16_t    bud_type;
    __uint16_t    bud_size;
    __uint32_t    __pad;
    __uint64_t    bud_bui_id;
};
```

### **bud\_type**

The signature of an BUD operation, 0x1245. This value is in host-endian order, not big-endian like the rest of XFS.

### **bud\_size**

Size of this log item. Should be 1.

### **bud\_bui\_id**

A 64-bit number that binds the corresponding BUI log item to this BUD log item.

## Inode Updates

This operation records changes to an inode record. There are several types of inode updates, each corresponding to different parts of the inode record. Allowing updates to proceed at a sub-inode granularity reduces contention for the inode, since different parts of the inode can be updated simultaneously.

The actual buffer data are stored in subsequent log items.

The inode log format header is as follows:

```
typedef struct xfs_inode_log_format_64 {
    __uint16_t    ilf_type;
    __uint16_t    ilf_size;
    __uint32_t    ilf_fields;
    __uint16_t    ilf_asize;
    __uint16_t    ilf_dsize;
    __uint32_t    ilf_pad;
    __uint64_t    ilf_ino;
    union {
        __uint32_t    ilfu_rdev;
        uuid_t        ilfu_uuid;
    } ilf_u;
    __int64_t     ilf_blkno;
    __int32_t     ilf_len;
    __int32_t     ilf_boffset;
} xfs_inode_log_format_64_t;
```

### **ilf\_type**

The signature of an inode update operation, 0x123b. This value is in host-endian order, not big-endian like the rest of XFS.

**ilf\_size**

Number of operations involved in this update, including this format operation.

**ilf\_fields**

Specifies which parts of the inode are being updated. This can be certain combinations of the following:

Flag	Inode changes to log include:
XFS_ILOG_CORE	The standard inode fields.
XFS_ILOG_DDATA	Data fork's local data.
XFS_ILOG_DEXT	Data fork's extent list.
XFS_ILOG_DBROOT	Data fork's B+tree root.
XFS_ILOG_DEV	Data fork's device number.
XFS_ILOG_UUID	Data fork's UUID contents.
XFS_ILOG_ADATA	Attribute fork's local data.
XFS_ILOG_AEXT	Attribute fork's extent list.
XFS_ILOG_ABROOT	Attribute fork's B+tree root.
XFS_ILOG_DOWNER	Change the data fork owner on replay.
XFS_ILOG_AOWNER	Change the attr fork owner on replay.
XFS_ILOG_TIMESTAMP	Timestamps are dirty, but not necessarily anything else. Should never appear on disk.
XFS_ILOG_NONCORE	( XFS_ILOG_DDATA   XFS_ILOG_DEXT   XFS_ILOG_DBROOT   XFS_ILOG_DEV   XFS_ILOG_UUID   XFS_ILOG_ADATA   XFS_ILOG_AEXT   XFS_ILOG_ABROOT   XFS_ILOG_DOWNER   XFS_ILOG_AOWNER )
XFS_ILOG_DFORK	( XFS_ILOG_DDATA   XFS_ILOG_DEXT   XFS_ILOG_DBROOT )
XFS_ILOG_AFORK	( XFS_ILOG_ADATA   XFS_ILOG_AEXT   XFS_ILOG_ABROOT )
XFS_ILOG_ALL	( XFS_ILOG_CORE   XFS_ILOG_DDATA   XFS_ILOG_DEXT   XFS_ILOG_DBROOT   XFS_ILOG_DEV   XFS_ILOG_UUID   XFS_ILOG_ADATA   XFS_ILOG_AEXT   XFS_ILOG_ABROOT   XFS_ILOG_TIMESTAMP   XFS_ILOG_DOWNER   XFS_ILOG_AOWNER )

**ilf\_asize**

Size of the attribute fork, in bytes.

**ilf\_dsize**

Size of the data fork, in bytes.

**ilf\_ino**

Absolute node number.

**ilfu\_rdev**

Device number information, for a device file update.

**ilfu\_uuid**

UUID, for a UUID update?

**ilf\_blkno**

Block number of the inode buffer, in sectors.

**ilf\_len**

Length of inode buffer, in sectors.

**ilf\_boffset**

Byte offset of the inode in the buffer.

Be aware that there is a nearly identical `xf_s_inode_log_format_32` which may appear on disk. It is the same as `xf_s_inode_log_format_64`, except that it is missing the `ilf_pad` field and is 52 bytes long as opposed to 56 bytes.

**Inode Data Log Item**

This region contains the new contents of a part of an inode, as described in the [previous section](#) Section 11.3.10. There are no magic numbers.

If `XFS_ILOG_CORE` is set in `ilf_fields`, the corresponding data buffer must be in the format `struct xf_s_icdiinode`, which has the same format as the first 96 bytes of an [inode](#) Chapter 13, but is recorded in host byte order.

**Buffer Log Item**

This operation writes parts of a buffer to disk. The regions to write are tracked in the data map; the actual buffer data are stored in subsequent log items.

```
typedef struct xf_s_buf_log_format {
    unsigned short    blf_type;
    unsigned short    blf_size;
    ushort           blf_flags;
    ushort           blf_len;
    __int64_t        blf_blkno;
    unsigned int      blf_map_size;
    unsigned int      blf_data_map[XFS_BLF_DATAMAP_SIZE];
} xf_s_buf_log_format_t;
```

**blf\_type**

Magic number to specify a buffer log item, 0x123c.

**blf\_size**

Number of buffer data items following this item.

**blf\_flags**

Specifies flags associated with the buffer item. This can be any of the following:

Flag	Description
<code>XFS_BLF_INODE_BUF</code>	Inode buffer. These must be recovered before replaying items that change this buffer.
<code>XFS_BLF_CANCEL</code>	Don't recover this buffer, blocks are being freed.
<code>XFS_BLF_UDQUOT_BUF</code>	User quota buffer, don't recover if there's a subsequent quotaoff.
<code>XFS_BLF_PDQUOT_BUF</code>	Project quota buffer, don't recover if there's a subsequent quotaoff.
<code>XFS_BLF_GDQUOT_BUF</code>	Group quota buffer, don't recover if there's a subsequent quotaoff.

**blf\_len**

Number of sectors affected by this buffer.

**blf\_blkno**

Block number to write, in sectors.

**blf\_map\_size**

The size of `blf_data_map`, in 32-bit words.

**blf\_data\_map**

This variable-sized array acts as a dirty bitmap for the logged buffer. Each 1 bit represents a dirty region in the buffer, and each run of 1 bits corresponds to a subsequent log item containing the new contents of the buffer area. Each bit represents  $(\text{blf\_len} * 512) / (\text{blf\_map\_size} * \text{NBBY})$  bytes.

## Buffer Data Log Item

This region contains the new contents of a part of a buffer, as described in the [previous section](#) Section 11.3.12. There are no magic numbers.

## Update Quota File

This updates a block in a quota file. The buffer data must be in the next log item.

```
typedef struct xfs_dq_logformat {
    __uint16_t      qlf_type;
    __uint16_t      qlf_size;
    xfs_dqid_t      qlf_id;
    __int64_t       qlf_blkno;
    __int32_t       qlf_len;
    __uint32_t      qlf_boffset;
} xfs_dq_logformat_t;
```

**qlf\_type**

The signature of an inode create operation, 0x123e. This value is in host-endian order, not big-endian like the rest of XFS.

**qlf\_size**

Size of this log item. Should be 2.

**qlf\_id**

The user/group/project ID to alter.

**qlf\_blkno**

Block number of the quota buffer, in sectors.

**qlf\_len**

Length of the quota buffer, in sectors.

**qlf\_boffset**

Buffer offset of the quota data to update, in bytes.

## Quota Update Data Log Item

This region contains the new contents of a part of a buffer, as described in the [previous section](#) Section 11.3.14. There are no magic numbers.

## Disable Quota Log Item

A request to disable quota controls has the following format:

```
typedef struct xfs_qoff_logformat {
    unsigned short    qf_type;
    unsigned short    qf_size;
    unsigned int      qf_flags;
    char              qf_pad[12];
} xfs_qoff_logformat_t;
```

### qf\_type

The signature of an inode create operation, 0x123d. This value is in host-endian order, not big-endian like the rest of XFS.

### qf\_size

Size of this log item. Should be 1.

### qf\_flags

Specifies which quotas are being turned off. Can be a combination of the following:

Flag	Quota type to disable
XFS_UQUOTA_ACCT	User quotas.
XFS_PQUOTA_ACCT	Project quotas.
XFS_GQUOTA_ACCT	Group quotas.

## Inode Creation Log Item

This log item is created when inodes are allocated in-core. When replaying this item, the specified inode records will be zeroed and some of the inode fields populated with default values.

```
struct xfs_icreate_log {
    __uint16_t        icl_type;
    __uint16_t        icl_size;
    __be32            icl_ag;
    __be32            icl_agbno;
    __be32            icl_count;
    __be32            icl_isize;
    __be32            icl_length;
    __be32            icl_gen;
};
```

### icl\_type

The signature of an inode create operation, 0x123f. This value is in host-endian order, not big-endian like the rest of XFS.

### icl\_size

Size of this log item. Should be 1.

### icl\_ag

AG number of the inode chunk to create.

**icl\_agbno**

AG block number of the inode chunk.

**icl\_count**

Number of inodes to initialize.

**icl\_isize**

Size of each inode, in bytes.

**icl\_length**

Length of the extent being initialized, in blocks.

**icl\_gen**

Inode generation number to write into the new inodes.

## xfs\_logprint Example

Here's an example of dumping the XFS log contents with `xfs_logprint`:

```
# xfs_logprint /dev/sda
xfs_logprint: /dev/sda contains a mounted and writable filesystem
xfs_logprint:
  data device: 0xfc03
  log device: 0xfc03 daddr: 900931640 length: 879816

cycle: 48      version: 2      lsn: 48,0      tail_lsn: 47,879760
length of Log Record: 19968    prev offset: 879808    num ops: 53
uuid: 24afeec2-f418-46a2-a573-10091f5e200e  format: little endian linux
h_size: 32768
```

This is the log record header.

```
Oper (0): tid: 30483aec len: 0 clientid: TRANS flags: START
```

This operation indicates that we're starting a transaction, so the next operation should record the transaction header.

```
Oper (1): tid: 30483aec len: 16 clientid: TRANS flags: none
TRAN:   type: CHECKPOINT      tid: 30483aec      num_items: 50
```

This operation records a transaction header. There should be fifty operations in this transaction and the transaction ID is 0x30483aec.

```
Oper (2): tid: 30483aec len: 24 clientid: TRANS flags: none
BUF: #regs: 2 start blkno: 145400496 (0x8aaa2b0) len: 8 bmap size: 1 flags: 0 ←
    x2000
Oper (3): tid: 30483aec len: 3712 clientid: TRANS flags: none
BUF DATA
...
Oper (4): tid: 30483aec len: 24 clientid: TRANS flags: none
BUF: #regs: 3 start blkno: 59116912 (0x3860d70) len: 8 bmap size: 1 flags: 0 ←
    x2000
Oper (5): tid: 30483aec len: 128 clientid: TRANS flags: none
BUF DATA
 0 43544241 49010000 fa347000 2c357000 3a40b200 13000000 2343c200 13000000
 8 3296d700 13000000 375deb00 13000000 8a551501 13000000 56be1601 13000000
```

```

10 af081901 13000000 ec741c01 13000000 9e911c01 13000000 69073501 13000000
18 4e539501 13000000 6549501 13000000 5d0e7f00 14000000 c6908200 14000000

Oper (6): tid: 30483aec len: 640 clientid: TRANS flags: none
BUF DATA
 0 7f47c800 21000000 23c0e400 21000000 2d0df00 21000000 e7060c01 21000000
 8 34b91801 21000000 9cca9100 22000000 26e69800 22000000 4c969900 22000000
...
90 1cf69900 27000000 42f79c00 27000000 6a99e00 27000000 6a99e00 27000000
98 6a99e00 27000000 6a99e00 27000000 6a99e00 27000000 6a99e00 27000000

```

Operations 4-6 describe two updates to a single dirty buffer at disk address 59,116,912. The first chunk of dirty data is 128 bytes long. Notice how the first four bytes of the first chunk is 0x43544241? Remembering that log items are in host byte order, reverse that to 0x41425443, which is the magic number for the free space B+tree ordered by size.

The second chunk is 640 bytes. There are more buffer changes, so we'll skip ahead a few operations:

```

Oper (19): tid: 30483aec len: 56 clientid: TRANS flags: none
INODE: #regs: 2 ino: 0x63a73b4e flags: 0x1 dsize: 40
      blkno: 1412688704 len: 16 boff: 7168
Oper (20): tid: 30483aec len: 96 clientid: TRANS flags: none
INODE CORE
magic 0x494e mode 0100600 version 2 format 3
nlink 1 uid 1000 gid 1000
atime 0x5633d58d mtime 0x563a391b ctime 0x563a391b
size 0x109dc8 nblocks 0x111 extsize 0x0 nextents 0x1b
naextents 0x0 forkoff 0 dmevmask 0x0 dmstate 0x0
flags 0x0 gen 0x389071be

```

This is an update to the core of inode 0x63a73b4e. There were similar inode core updates after this, so we'll skip ahead a bit:

```

Oper (32): tid: 30483aec len: 56 clientid: TRANS flags: none
INODE: #regs: 3 ino: 0x4bde428 flags: 0x5 dsize: 16
      blkno: 79553568 len: 16 boff: 4096
Oper (33): tid: 30483aec len: 96 clientid: TRANS flags: none
INODE CORE
magic 0x494e mode 0100644 version 2 format 2
nlink 1 uid 1000 gid 1000
atime 0x563a3924 mtime 0x563a3931 ctime 0x563a3931
size 0x1210 nblocks 0x2 extsize 0x0 nextents 0x1
naextents 0x0 forkoff 0 dmevmask 0x0 dmstate 0x0
flags 0x0 gen 0x2829c6f9
Oper (34): tid: 30483aec len: 16 clientid: TRANS flags: none
EXTENTS inode data

```

This inode update changes both the core and also the data fork. Since we're changing the block map, it's unsurprising that one of the subsequent operations is an EFI:

```

Oper (37): tid: 30483aec len: 32 clientid: TRANS flags: none
EFI: #regs: 1 num_extents: 1 id: 0xffff8801147b5c20
(s: 0x720daf, l: 1)
\-----
Oper (38): tid: 30483aec len: 32 clientid: TRANS flags: none
EFD: #regs: 1 num_extents: 1 id: 0xffff8801147b5c20
\-----
Oper (39): tid: 30483aec len: 24 clientid: TRANS flags: none

```

```

BUF: #regs: 2  start blkno: 8 (0x8)  len: 8  bmap size: 1  flags: 0x2800
Oper (40): tid: 30483aec  len: 128  clientid: TRANS  flags: none
AGF Buffer: XAGF
ver: 1  seq#: 0  len: 56308224
root BNO: 18174905  CNT: 18175030
level BNO: 2  CNT: 2
1st: 41  last: 46  cnt: 6  freeblks: 35790503  longest: 19343245
\-----
Oper (41): tid: 30483aec  len: 24  clientid: TRANS  flags: none
BUF: #regs: 3  start blkno: 145398760 (0x8aa9be8)  len: 8  bmap size: 1  flags: 0 ←
    x2000
Oper (42): tid: 30483aec  len: 128  clientid: TRANS  flags: none
BUF DATA
Oper (43): tid: 30483aec  len: 128  clientid: TRANS  flags: none
BUF DATA
\-----
Oper (44): tid: 30483aec  len: 24  clientid: TRANS  flags: none
BUF: #regs: 3  start blkno: 145400224 (0x8aaa1a0)  len: 8  bmap size: 1  flags: 0 ←
    x2000
Oper (45): tid: 30483aec  len: 128  clientid: TRANS  flags: none
BUF DATA
Oper (46): tid: 30483aec  len: 3584  clientid: TRANS  flags: none
BUF DATA
\-----
Oper (47): tid: 30483aec  len: 24  clientid: TRANS  flags: none
BUF: #regs: 3  start blkno: 59066216 (0x3854768)  len: 8  bmap size: 1  flags: 0 ←
    x2000
Oper (48): tid: 30483aec  len: 128  clientid: TRANS  flags: none
BUF DATA
Oper (49): tid: 30483aec  len: 768  clientid: TRANS  flags: none
BUF DATA

```

Here we see an EFI, followed by an EFD, followed by updates to the AGF and the free space B+trees. Most probably, we just unmapped a few blocks from a file.

```

Oper (50): tid: 30483aec  len: 56  clientid: TRANS  flags: none
INODE: #regs: 2  ino: 0x3906f20  flags: 0x1  dsize: 16
        blkno: 59797280  len: 16  boff: 0
Oper (51): tid: 30483aec  len: 96  clientid: TRANS  flags: none
INODE CORE
magic 0x494e mode 0100644 version 2 format 2
nlink 1 uid 1000 gid 1000
atime 0x563a3938 mtime 0x563a3938 ctime 0x563a3938
size 0x0 nblocks 0x0 extsize 0x0 nextents 0x0
naextents 0x0 forkoff 0 dmevmask 0x0 dmstate 0x0
flags 0x0 gen 0x35ed661
\-----
Oper (52): tid: 30483aec  len: 0  clientid: TRANS  flags: COMMIT

```

One more inode core update and this transaction commits.



## Chapter 12

# Internal Inodes

XFS allocates several inodes when a filesystem is created. These are internal and not accessible from the standard directory structure. These inodes are only accessible from the superblock.

## Quota Inodes

If quotas are used, two inodes are allocated for user and group quota management. If project quotas are used, these replace the group quota management and therefore uses the group quota inode.

- Project quota's primary purpose is to track and monitor disk usage for directories. For this to occur, the directory inode must have the `XFS_DIFLAG_PROJINHERIT` flag set so all inodes created underneath the directory inherit the project ID.
  - Inodes and blocks owned by ID zero do not have enforced quotas, but only quota accounting.
  - Extended attributes do not contribute towards the ID's quota.
  - To access each ID's quota information in the file, seek to the ID offset multiplied by the size of `xf_s_dqblk_t` (136 bytes).
-

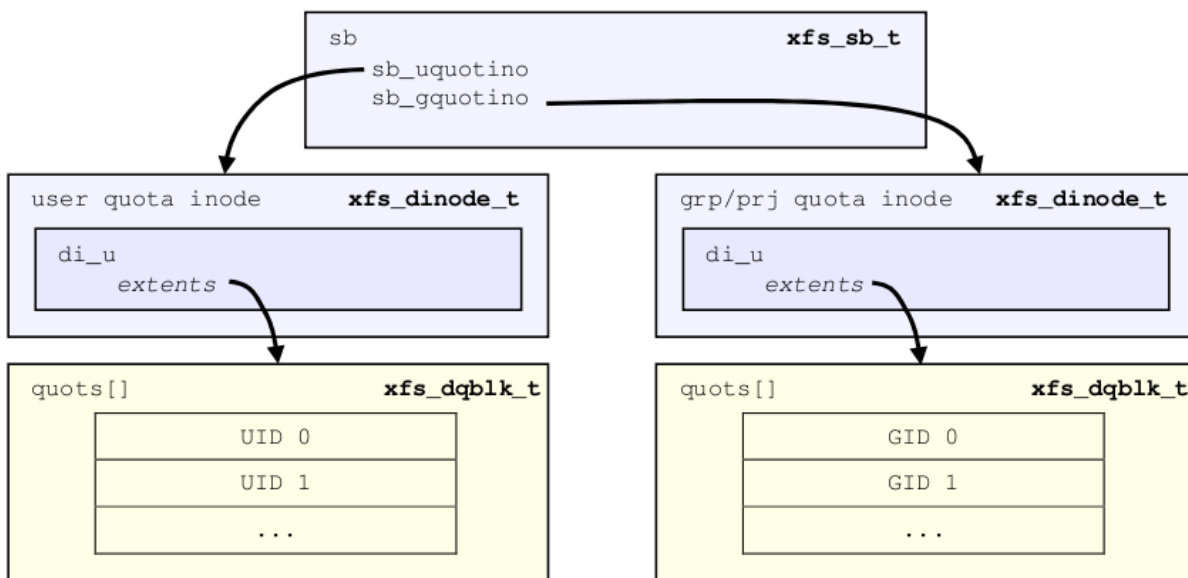


Figure 12.1: Quota inode layout

Quota information is stored in the data extents of the two reserved quota inodes as an array of the `xfs_dqblk` structures, where there is one array element for each ID in the system:

```

struct xfs_disk_dquot {
    __be16    d_magic;
    __u8     d_version;
    __u8     d_flags;
    __be32    d_id;
    __be64    d_blk_hardlimit;
    __be64    d_blk_softlimit;
    __be64    d_ino_hardlimit;
    __be64    d_ino_softlimit;
    __be64    d_bcount;
    __be64    d_icount;
    __be32    d_itimer;
    __be32    d_btimer;
    __be16    d_iwarns;
    __be16    d_bwarns;
    __be32    d_pad0;
    __be64    d_rtb_hardlimit;
    __be64    d_rtb_softlimit;
    __be64    d_rtbcount;
    __be32    d_rtbtimer;
    __be16    d_rtbwarns;
    __be16    d_pad;
};
struct xfs_dqblk {
    struct xfs_disk_dquot dd_diskdq;
    char                 dd_fill[32];

    /* version 5 filesystem fields begin here */
    __be32               dd_crc;

```

```

    __be64          dd_lsn;
    uuid_t          dd_uuid;
};

```

**d\_magic**

Specifies the signature where these two bytes are 0x4451 (XFS\_DQUOT\_MAGIC), or “DQ” in ASCII.

**d\_version**

The structure version, currently this is 1 (XFS\_DQUOT\_VERSION).

**d\_flags**

Specifies which type of ID the structure applies to:

```

#define XFS_DQ_USER   0x0001
#define XFS_DQ_PROJ   0x0002
#define XFS_DQ_GROUP  0x0004

```

**d\_id**

The ID for the quota structure. This will be a uid, gid or projid based on the value of `d_flags`.

**d\_blk\_hardlimit**

The hard limit for the number of filesystem blocks the ID can own. The ID will not be able to use more space than this limit. If it is attempted, ENOSPC will be returned.

**d\_blk\_softlimit**

The soft limit for the number of filesystem blocks the ID can own. The ID can temporarily use more space than by `d_blk_softlimit` up to `d_blk_hardlimit`. If the space is not freed by the time limit specified by ID zero's `d_btimer` value, the ID will be denied more space until the total blocks owned goes below `d_blk_softlimit`.

**d\_ino\_hardlimit**

The hard limit for the number of inodes the ID can own. The ID will not be able to create or own any more inodes if `d_icoount` reaches this value.

**d\_ino\_softlimit**

The soft limit for the number of inodes the ID can own. The ID can temporarily create or own more inodes than specified by `d_ino_softlimit` up to `d_ino_hardlimit`. If the inode count is not reduced by the time limit specified by ID zero's `d_itimer` value, the ID will be denied from creating or owning more inodes until the count goes below `d_ino_softlimit`.

**d\_bcount**

How many filesystem blocks are actually owned by the ID.

**d\_icoount**

How many inodes are actually owned by the ID.

**d\_itimer**

Specifies the time when the ID's `d_icoount` exceeded `d_ino_softlimit`. The soft limit will turn into a hard limit after the elapsed time exceeds ID zero's `d_itimer` value. When `d_icoount` goes back below `d_ino_softlimit`, `d_itimer` is reset back to zero.

**d\_btimer**

Specifies the time when the ID's `d_bcount` exceeded `d_blk_softlimit`. The soft limit will turn into a hard limit after the elapsed time exceeds ID zero's `d_btimer` value. When `d_bcount` goes back below `d_blk_softlimit`, `d_btimer` is reset back to zero.

**d\_iwarns , d\_bwarns , d\_rtbwarns**

Specifies how many times a warning has been issued. Currently not used.

**d\_rtbf\_hardlimit**

The hard limit for the number of real-time blocks the ID can own. The ID cannot own more space on the real-time subvolume beyond this limit.

**d\_rtbf\_softlimit**

The soft limit for the number of real-time blocks the ID can own. The ID can temporarily own more space than specified by `d_rtbf_softlimit` up to `d_rtbf_hardlimit`. If `d_rtbfcount` is not reduced by the time limit specified by ID zero's `d_rtbf_timer` value, the ID will be denied from owning more space until the count goes below `d_rtbf_softlimit`.

**d\_rtbfcount**

How many real-time blocks are currently owned by the ID.

**d\_rtbf\_timer**

Specifies the time when the ID's `d_rtbfcount` exceeded `d_rtbf_softlimit`. The soft limit will turn into a hard limit after the elapsed time exceeds ID zero's `d_rtbf_timer` value. When `d_rtbfcount` goes back below `d_rtbf_softlimit`, `d_rtbf_timer` is reset back to zero.

**dd\_uuid**

The UUID of this block, which must match either `sb_uuid` or `sb_meta_uuid` depending on which features are set.

**dd\_lsn**

Log sequence number of the last DQ block write.

**dd\_crc**

Checksum of the DQ block.

## Real-time Inodes

There are two inodes allocated to managing the real-time device's space, the Bitmap Inode and the Summary Inode.

### Real-Time Bitmap Inode

The real time bitmap inode, `sb_rbmno`, tracks the used/free space in the real-time device using an old-style bitmap. One bit is allocated per real-time extent. The size of an extent is specified by the superblock's `sb_rextsize` value.

The number of blocks used by the bitmap inode is equal to the number of real-time extents (`sb_rextents`) divided by the block size (`sb_blocksize`) and bits per byte. This value is stored in `sb_rmblocks`. The `nblocks` and `extent` array for the inode should match this. Each real time block gets its own bit in the bitmap.

### Real-Time Summary Inode

The real time summary inode, `sb_rsumno`, tracks the used and free space accounting information for the real-time device. This file indexes the approximate location of each free extent on the real-time device first by  $\log_2(\text{extent size})$  and then by the real-time bitmap block number. The size of the summary inode file is equal to  $\text{sb\_rmblocks} \times \log_2(\text{realtime device size}) \times \text{sizeof}(\text{xfs\_suminfo\_t})$ . The entry for a given  $\log_2(\text{extent size})$  and `rtbitmap` block number is 0 if there is no free extents of that size at that `rtbitmap` location, and positive if there are any.

This data structure is not particularly space efficient, however it is a very fast way to provide the same data as the two free space B+trees for regular files since the space is preallocated and metadata maintenance is minimal.

---

## Real-Time Reverse-Mapping B+tree

---

### Note

This data structure is under construction! Details may change.

---

If the reverse-mapping B+tree and real-time storage device features are enabled, the real-time device has its own reverse block-mapping B+tree.

As mentioned in the chapter about [reconstruction](#) Chapter 4, this data structure is another piece of the puzzle necessary to reconstruct the data or attribute fork of a file from reverse-mapping records; we can also use it to double-check allocations to ensure that we are not accidentally cross-linking blocks, which can cause severe damage to the filesystem.

This B+tree is only present if the `XFS_SB_FEAT_RO_COMPAT_RMAPBT` feature is enabled and a real time device is present. The feature requires a version 5 filesystem.

The real-time reverse mapping B+tree is rooted in an inode's data fork; the inode number is given by the `sb_rmapino` field in the superblock. The B+tree blocks themselves are stored in the regular filesystem. The structures used for an inode's B+tree root are:

```
struct xfs_rtrmap_root {
    __be16          bb_level;
    __be16          bb_numrecs;
};
```

- On disk, the B+tree node starts with the `xfs_rtrmap_root` header followed by an array of `xfs_rtrmap_key` values and then an array of `xfs_rtrmap_ptr_t` values. The size of both arrays is specified by the header's `bb_numrecs` value.
- The root node in the inode can only contain up to 10 key/pointer pairs for a standard 512 byte inode before a new level of nodes is added between the root and the leaves. `di_forkoff` should always be zero, because there are no extended attributes.

Each record in the real-time reverse-mapping B+tree has the following structure:

```
struct xfs_rtrmap_rec {
    __be64          rm_startblock;
    __be64          rm_blockcount;
    __be64          rm_owner;
    __be64          rm_fork:1;
    __be64          rm_bmbt:1;
    __be64          rm_unwritten:1;
    __be64          rm_unused:7;
    __be64          rm_offset:54;
};
```

### `rm_startblock`

Real-time device block number of this record.

### `rm_blockcount`

The length of this extent, in real-time blocks.

---

**rm\_owner**

A 64-bit number describing the owner of this extent. This must be an inode number, because the real-time device is for file data only.

**rm\_fork**

If `rm_owner` describes an inode, this can be 1 if this record is for an attribute fork. This value will always be zero for real-time extents.

**rm\_bmbt**

If `rm_owner` describes an inode, this can be 1 to signify that this record is for a block map B+tree block. In this case, `rm_offset` has no meaning. This value will always be zero for real-time extents.

**rm\_unwritten**

A flag indicating that the extent is unwritten. This corresponds to the flag in the [extent record](#) Chapter 14 format which means `XFS_EXT_UNWRITTEN`.

**rm\_offset**

The 54-bit logical file block offset, if `rm_owner` describes an inode.

**Note**

The single-bit flag values `rm_unwritten`, `rm_fork`, and `rm_bmbt` are packed into the larger fields in the C structure definition.

The key has the following structure:

```
struct xfs_rtrmap_key {
    __be64          rm_startblock;
    __be64          rm_owner;
    __be64          rm_fork:1;
    __be64          rm_bmbt:1;
    __be64          rm_reserved:1;
    __be64          rm_unused:7;
    __be64          rm_offset:54;
};
```

- All block numbers are 64-bit real-time device block numbers.
- The `bb_magic` value is “MAPR” (0x4d415052).
- The `xfs_btree_lblock_t` header is used for intermediate B+tree node as well as the leaves.
- Each pointer is associated with two keys. The first of these is the “low key”, which is the key of the smallest record accessible through the pointer. This low key has the same meaning as the key in all other btrees. The second key is the high key, which is the maximum of the largest key that can be used to access a given record underneath the pointer. Recall that each record in the real-time reverse mapping b+tree describes an interval of physical blocks mapped to an interval of logical file block offsets; therefore, it makes sense that a range of keys can be used to find to a record.

**xfs\_db rtrmapbt Example**

This example shows a real-time reverse-mapping B+tree from a freshly populated root filesystem:

```

xfs_db> sb 0
xfs_db> addr rrmmapino
xfs_db> p
core.magic = 0x494e
core.mode = 0100000
core.version = 3
core.format = 5 (rtrmapbt)
...
u3.rtrmapbt.level = 3
u3.rtrmapbt.numrecs = 1
u3.rtrmapbt.keys[1] = [startblock,owner,offset,attrfork,bmbtblock,startblock_hi,
                      owner_hi,offset_hi,attrfork_hi,bmbtblock_hi]
                      1:[1,132,1,0,0,1705337,133,54431,0,0]
u3.rtrmapbt.ptrs[1] = 1:671
xfs_db> addr u3.rtrmapbt.ptrs[1]
xfs_db> p
magic = 0x4d415052
level = 2
numrecs = 8
leftsib = null
rightsib = null
bno = 5368
lsn = 0x400000000
uuid = 98bbde42-67e7-46a5-a73e-d64a76b1b5ce
owner = 131
crc = 0x2560d199 (correct)
keys[1-8] = [startblock,owner,offset,attrfork,bmbtblock,startblock_hi,owner_hi,
            offset_hi,attrfork_hi,bmbtblock_hi]
            1:[1,132,1,0,0,17749,132,17749,0,0]
            2:[17751,132,17751,0,0,35499,132,35499,0,0]
            3:[35501,132,35501,0,0,53249,132,53249,0,0]
            4:[53251,132,53251,0,0,1658473,133,7567,0,0]
            5:[1658475,133,7569,0,0,1667473,133,16567,0,0]
            6:[1667475,133,16569,0,0,1685223,133,34317,0,0]
            7:[1685225,133,34319,0,0,1694223,133,43317,0,0]
            8:[1694225,133,43319,0,0,1705337,133,54431,0,0]
ptrs[1-8] = 1:134 2:238 3:345 4:453 5:795 6:563 7:670 8:780

```

We arbitrarily pick pointer 7 (twice) to traverse downwards:

```

xfs_db> addr ptrs[7]
xfs_db> p
magic = 0x4d415052
level = 1
numrecs = 36
leftsib = 563
rightsib = 780
bno = 5360
lsn = 0
uuid = 98bbde42-67e7-46a5-a73e-d64a76b1b5ce
owner = 131
crc = 0x6807761d (correct)
keys[1-36] = [startblock,owner,offset,attrfork,bmbtblock,startblock_hi,owner_hi,
             offset_hi,attrfork_hi,bmbtblock_hi]
             1:[1685225,133,34319,0,0,1685473,133,34567,0,0]
             2:[1685475,133,34569,0,0,1685723,133,34817,0,0]

```

```

    3:[1685725,133,34819,0,0,1685973,133,35067,0,0]
    ...
    34:[1693475,133,42569,0,0,1693723,133,42817,0,0]
    35:[1693725,133,42819,0,0,1693973,133,43067,0,0]
    36:[1693975,133,43069,0,0,1694223,133,43317,0,0]
ptrs[1-36] = 1:669 2:672 3:674...34:722 35:723 36:725
xfs_db> addr ptrs[7]
xfs_db> p
magic = 0x4d415052
level = 0
numrecs = 125
leftsib = 678
rightsib = 681
bno = 5440
lsn = 0
uuid = 98bbde42-67e7-46a5-a73e-d64a76b1b5ce
owner = 131
crc = 0xefce34d4 (correct)
recs[1-125] = [startblock,blockcount,owner,offset,extentflag,attrfork,bmbtblock]
    1:[1686725,1,133,35819,0,0,0]
    2:[1686727,1,133,35821,0,0,0]
    3:[1686729,1,133,35823,0,0,0]
    ...
   123:[1686969,1,133,36063,0,0,0]
   124:[1686971,1,133,36065,0,0,0]
   125:[1686973,1,133,36067,0,0,0]

```

Several interesting things pop out here. The first record shows that inode 133 has mapped real-time block 1,686,725 at offset 35,819. We confirm this by looking at the block map for that inode:

```

xfs_db> inode 133
xfs_db> p core.realtime
core.realtime = 1
xfs_db> bmap
data offset 35817 startblock 1686723 (1/638147) count 1 flag 0
data offset 35819 startblock 1686725 (1/638149) count 1 flag 0
data offset 35821 startblock 1686727 (1/638151) count 1 flag 0

```

Notice that inode 133 has the real-time flag set, which means that its data blocks are all allocated from the real-time device.



## Part III

# Dynamically Allocated Structures

---

## Chapter 13

# On-disk Inode

All files, directories, and links are stored on disk with inodes and descend from the root inode with its number defined in the [superblock](#) Section 10.1. The previous section on [AG Inode Management](#) Section 10.3 describes the allocation and management of inodes on disk. This section describes the contents of inodes themselves.

An inode is divided into 3 parts:

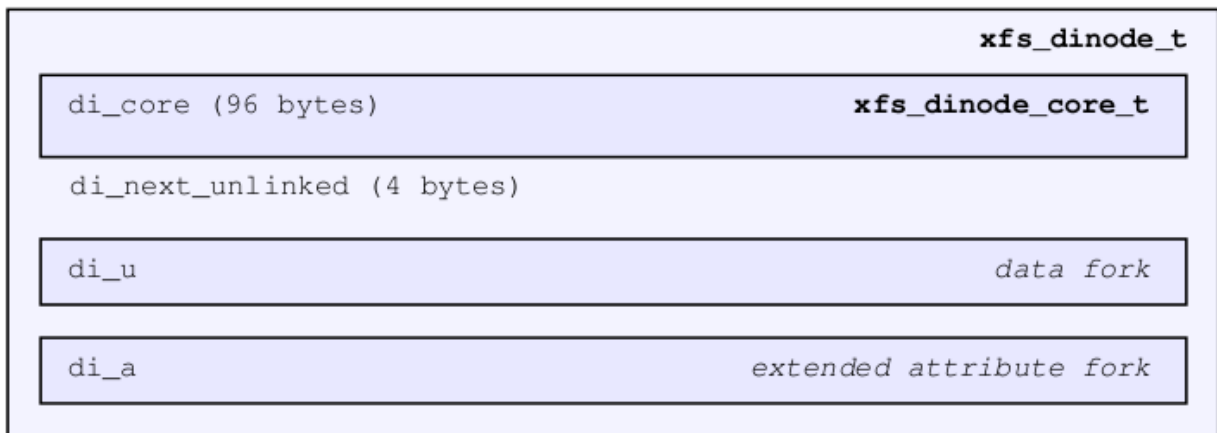


Figure 13.1: On-disk inode sections

- The core contains what the inode represents, stat data, and information describing the data and attribute forks.
- The `di_u` “data fork” contains normal data related to the inode. Its contents depends on the file type specified by `di_core.di_mode` (eg. regular file, directory, link, etc) and how much information is contained in the file which determined by `di_core.di_format`. The following union to represent this data is declared as follows:

```

union {
    xfs_bmdr_block_t di_bmbt;
    xfs_bmbt_rec_t   di_bmx[1];
    xfs_dir2_sf_t    di_dir2sf;
    char             di_c[1];
    xfs_dev_t        di_dev;
}
  
```

```

    uuid_t      di_muuid;
    char        di_symlink[1];
} di_u;

```

- The `di_a` “attribute fork” contains extended attributes. Its layout is determined by the `di_core.di_aformat` value. Its representation is declared as follows:

```

union {
    xfs_bmdr_block_t    di_abmbt;
    xfs_bmbt_rec_t     di_abmx[1];
    xfs_attr_shortform_t di_attrsf;
} di_a;

```

---

### Note

The above two unions are rarely used in the XFS code, but the structures within the union are directly cast depending on the `di_mode/di_format` and `di_aformat` values. They are referenced in this document to make it easier to explain the various structures in use within the inode.

---

The remaining space in the inode after `di_next_unlinked` where the two forks are located is called the inode’s “literal area”. This starts at offset 100 (0x64) in a version 1 or 2 inode, and offset 176 (0xb0) in a version 3 inode.

The space for each of the two forks in the literal area is determined by the inode size, and `di_core.di_forkoff`. The data fork is located between the start of the literal area and `di_forkoff`. The attribute fork is located between `di_forkoff` and the end of the inode.

## Inode Core

The inode’s core is 96 bytes on a V4 filesystem and 176 bytes on a V5 filesystem. It contains information about the file itself including most stat data information about data and attribute forks after the core within the inode. It uses the following structure:

```

struct xfs_dinode_core {
    __uint16_t    di_magic;
    __uint16_t    di_mode;
    __int8_t     di_version;
    __int8_t     di_format;
    __uint16_t    di_onlink;
    __uint32_t    di_uid;
    __uint32_t    di_gid;
    __uint32_t    di_nlink;
    __uint16_t    di_projid;
    __uint16_t    di_projid_hi;
    __uint8_t     di_pad[6];
    __uint16_t    di_flushiter;
    xfs_timestamp_t di_atime;
    xfs_timestamp_t di_mtime;
    xfs_timestamp_t di_ctime;
    xfs_fsize_t    di_size;
    xfs_rfsblock_t di_nblocks;
    xfs_extlen_t   di_extsize;
}

```

---

```

xfs_extnum_t      di_nextents;
xfs_aextnum_t     di_anextents;
__uint8_t        di_forkoff;
__int8_t         di_aformat;
__uint32_t       di_dmevmask;
__uint16_t       di_dmstate;
__uint16_t       di_flags;
__uint32_t       di_gen;

/* di_next_unlinked is the only non-core field in the old dinode */
__be32          di_next_unlinked;

/* version 5 filesystem (inode version 3) fields start here */
__le32          di_crc;
__be64          di_changecount;
__be64          di_lsn;
__be64          di_flags2;
__be32          di_cowextsize;
__u8            di_pad2[12];
xfs_timestamp_t di_crtime;
__be64          di_ino;
uuid_t          di_uuid;
};

```

**di\_magic**

The inode signature; these two bytes are “IN” (0x494e).

**di\_mode**

Specifies the mode access bits and type of file using the standard S\_Ixxx values defined in stat.h.

**di\_version**

Specifies the inode version which currently can only be 1, 2, or 3. The inode version specifies the usage of the `di_onlink`, `di_nlink` and `di_projid` values in the inode core. Initially, inodes are created as v1 but can be converted on the fly to v2 when required. v3 inodes are created only for v5 filesystems.

**di\_format**

Specifies the format of the data fork in conjunction with the `di_mode` type. This can be one of several values. For directories and links, it can be “local” where all metadata associated with the file is within the inode; “extents” where the inode contains an array of extents to other filesystem blocks which contain the associated metadata or data; or “btree” where the inode contains a B+tree root node which points to filesystem blocks containing the metadata or data. Migration between the formats depends on the amount of metadata associated with the inode. “dev” is used for character and block devices while “uuid” is currently not used. “rmap” indicates that a reverse-mapping B+tree is rooted in the fork.

```

typedef enum xfs_dinode_fmt {
    XFS_DINODE_FMT_DEV,
    XFS_DINODE_FMT_LOCAL,
    XFS_DINODE_FMT_EXTENTS,
    XFS_DINODE_FMT_BTREE,
    XFS_DINODE_FMT_UUID,
    XFS_DINODE_FMT_RMAP,
} xfs_dinode_fmt_t;

```

**di\_onlink**

In v1 inodes, this specifies the number of links to the inode from directories. When the number exceeds 65535, the inode is converted to v2 and the link count is stored in `di_nlink`.

**di\_uid**

Specifies the owner's UID of the inode.

**di\_gid**

Specifies the owner's GID of the inode.

**di\_nlink**

Specifies the number of links to the inode from directories. This is maintained for both inode versions for current versions of XFS. Prior to v2 inodes, this field was part of `di_pad`.

**di\_projid**

Specifies the owner's project ID in v2 inodes. An inode is converted to v2 if the project ID is set. This value must be zero for v1 inodes.

**di\_projid\_hi**

Specifies the high 16 bits of the owner's project ID in v2 inodes, if the `XFS_SB_VERSION2_PROJID32BIT` feature is set; and zero otherwise.

**di\_pad[6]**

Reserved, must be zero.

**di\_flushiter**

Incremented on flush.

**di\_atime**

Specifies the last access time of the files using UNIX time conventions the following structure. This value may be undefined if the filesystem is mounted with the "noatime" option. XFS supports timestamps with nanosecond resolution:

```
struct xfs_timestamp {
    __int32_t      t_sec;
    __int32_t      t_nsec;
};
```

**di\_mtime**

Specifies the last time the file was modified.

**di\_ctime**

Specifies when the inode's status was last changed.

**di\_size**

Specifies the EOF of the inode in bytes. This can be larger or smaller than the extent space (therefore actual disk space) used for the inode. For regular files, this is the filesize in bytes, directories, the space taken by directory entries and for links, the length of the symlink.

**di\_nblocks**

Specifies the number of filesystem blocks used to store the inode's data including relevant metadata like B+trees. This does not include blocks used for extended attributes.

---

**di\_extsize**

Specifies the extent size for filesystems with real-time devices or an extent size hint for standard filesystems. For normal filesystems, and with directories, the XFS\_DIFLAG\_EXTSZINHERIT flag must be set in `di_flags` if this field is used. Inodes created in these directories will inherit the `di_extsize` value and have XFS\_DIFLAG\_EXTSIZE set in their `di_flags`. When a file is written to beyond allocated space, XFS will attempt to allocate additional disk space based on this value.

**di\_nextents**

Specifies the number of data extents associated with this inode.

**di\_anextents**

Specifies the number of extended attribute extents associated with this inode.

**di\_forkoff**

Specifies the offset into the inode's literal area where the extended attribute fork starts. This is an 8-bit value that is multiplied by 8 to determine the actual offset in bytes (ie. attribute data is 64-bit aligned). This also limits the maximum size of the inode to 2048 bytes. This value is initially zero until an extended attribute is created. When an attribute is added, the nature of `di_forkoff` depends on the XFS\_SB\_VERSION2\_ATTR2BIT flag in the superblock. Refer to [Extended Attribute Versions](#) Section 13.4.1 for more details.

**di\_aformat**

Specifies the format of the attribute fork. This uses the same values as `di_format`, but restricted to "local", "extents" and "btree" formats for extended attribute data.

**di\_dmevmask**

DMAPI event mask.

**di\_dmstate**

DMAPI state.

**di\_flags**

Specifies flags associated with the inode. This can be a combination of the following values:

Table 13.1: Version 2 Inode flags

Flag	Description
XFS_DIFLAG_REALTIME	The inode's data is located on the real-time device.
XFS_DIFLAG_PREALLOC	The inode's extents have been preallocated.
XFS_DIFLAG_NEWRTBM	Specifies the <code>sb_rbmno</code> uses the new real-time bitmap format
XFS_DIFLAG_IMMUTABLE	Specifies the inode cannot be modified.
XFS_DIFLAG_APPEND	The inode is in append only mode.
XFS_DIFLAG_SYNC	The inode is written synchronously.
XFS_DIFLAG_NOATIME	The inode's <code>di_atime</code> is not updated.
XFS_DIFLAG_NODUMP	Specifies the inode is to be ignored by <code>xfsdump</code> .
XFS_DIFLAG_RTINHERIT	For directory inodes, new inodes inherit the XFS_DIFLAG_REALTIME bit.
XFS_DIFLAG_PROJINHERIT	For directory inodes, new inodes inherit the <code>di_projid</code> value.
XFS_DIFLAG_NOSYMLINKS	For directory inodes, symlinks cannot be created.
XFS_DIFLAG_EXTSIZE	Specifies the extent size for real-time files or an extent size hint for regular files.
XFS_DIFLAG_EXTSZINHERIT	For directory inodes, new inodes inherit the <code>di_extsize</code> value.

Table 13.1: (continued)

Flag	Description
XFS_DIFLAG_NODEFRAG	Specifies the inode is to be ignored when defragmenting the filesystem.
XFS_DIFLAG_FILESTREAMS	Use the filestream allocator. The filestreams allocator allows a directory to reserve an entire allocation group for exclusive use by files created in that directory. Files in other directories cannot use AGs reserved by other directories.

**di\_gen**

A generation number used for inode identification. This is used by tools that do inode scanning such as backup tools and xfsdump. An inode's generation number can change by unlinking and creating a new file that reuses the inode.

**di\_next\_unlinked**

See the section on [unlinked inode pointers](#) Section 13.2 for more information.

**di\_crc**

Checksum of the inode.

**di\_changecount**

Counts the number of changes made to the attributes in this inode.

**di\_lsn**

Log sequence number of the last inode write.

**di\_flags2**

Specifies extended flags associated with a v3 inode.

Table 13.2: Version 3 Inode flags

Flag	Description
XFS_DIFLAG2_DAX	For a file, enable DAX to increase performance on persistent-memory storage. If set on a directory, files created in the directory will inherit this flag.
XFS_DIFLAG2_REFLINK	This inode shares (or has shared) data blocks with another inode.
XFS_DIFLAG2_COWEXTSIZE	For files, this is the extent size hint for copy on write operations; see <code>di_cowextsize</code> for details. For directories, the value in <code>di_cowextsize</code> will be copied to all newly created files and directories.

**di\_cowextsize**

Specifies the extent size hint for copy on write operations. When allocating extents for a copy on write operation, the allocator will be asked to align its allocations to either `di_cowextsize` blocks or `di_extsize`

blocks, whichever is greater. The `XFS_DIFLAG2_COWEXTSIZE` flag must be set if this field is used. If this field and its flag are set on a directory file, the value will be copied into any files or directories created within this directory. During a block sharing operation, this value will be copied from the source file to the destination file if the sharing operation completely overwrites the destination file's contents and the destination file does not already have `di_cowextsize` set.

**di\_pad2**

Padding for future expansion of the inode.

**di\_crtime**

Specifies the time when this inode was created.

**di\_ino**

The full inode number of this inode.

**di\_uuid**

The UUID of this inode, which must match either `sb_uuid` or `sb_meta_uuid` depending on which features are set.

## Unlinked Pointer

The `di_next_unlinked` value in the inode is used to track inodes that have been unlinked (deleted) but are still open by a program. When an inode is in this state, the inode is added to one of the [AGI's Section 10.3](#) `agi_unlinked` hash buckets. The AGI unlinked bucket points to an inode and the `di_next_unlinked` value points to the next inode in the chain. The last inode in the chain has `di_next_unlinked` set to NULL (-1).

Once the last reference is released, the inode is removed from the unlinked hash chain and `di_next_unlinked` is set to NULL. In the case of a system crash, XFS recovery will complete the unlink process for any inodes found in these lists.

The only time the unlinked fields can be seen to be used on disk is either on an active filesystem or a crashed system. A cleanly unmounted or recovered filesystem will not have any inodes in these unlink hash chains.



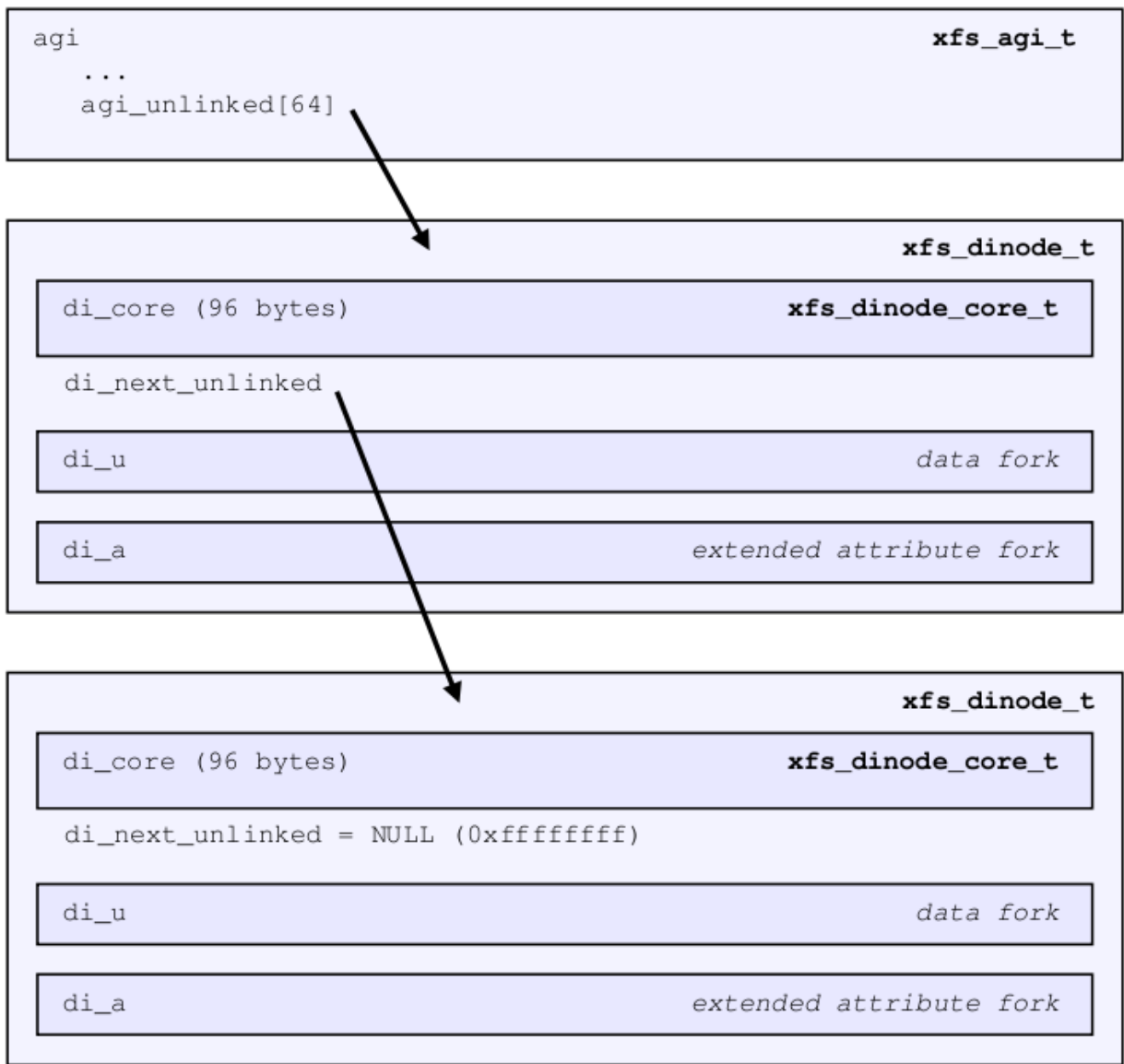


Figure 13.2: Unlinked inode pointer

## Data Fork

The structure of the inode’s data fork based is on the inode’s type and `di_format`. The data fork begins at the start of the inode’s “literal area”. This area starts at offset 100 (0x64), or offset 176 (0xb0) in a v3 inode. The size of the data fork is determined by the type and format. The maximum size is determined by the inode size and `di_forkoff`. In code, use the `XFS_DFORK_PTR` macro specifying `XFS_DATA_FORK` for the “which” parameter. Alternatively, the `XFS_DFORK_DPTR` macro can be used.

Each of the following sub-sections summarises the contents of the data fork based on the inode type.

## Regular Files (S\_IFREG)

The data fork specifies the file's data extents. The extents specify where the file's actual data is located within the filesystem. Extents can have 2 formats which is defined by the `di_format` value:

- `XFS_DINODE_FMT_EXTENTS`: The extent data is fully contained within the inode which contains an array of extents to the filesystem blocks for the file's data. To access the extents, cast the return value from `XFS_DFORK_DPTR` to `xf_s_bmbt_rec_t*`.
- `XFS_DINODE_FMT_BTREE`: The extent data is contained in the leaves of a B+tree. The inode contains the root node of the tree and is accessed by casting the return value from `XFS_DFORK_DPTR` to `xf_s_bmdr_block_t*`.

Details for each of these data extent formats are covered in the [Data Extents](#) Chapter 14 later on.

## Directories (S\_IFDIR)

The data fork contains the directory's entries and associated data. The format of the entries is also determined by the `di_format` value and can be one of 3 formats:

- `XFS_DINODE_FMT_LOCAL`: The directory entries are fully contained within the inode. This is accessed by casting the value from `XFS_DFORK_DPTR` to `xf_s_dir2_sf_t*`.
- `XFS_DINODE_FMT_EXTENTS`: The actual directory entries are located in another filesystem block, the inode contains an array of extents to these filesystem blocks (`xf_s_bmbt_rec_t*`).
- `XFS_DINODE_FMT_BTREE`: The directory entries are contained in the leaves of a B+tree. The inode contains the root node (`xf_s_bmdr_block_t*`).

Details for each of these directory formats are covered in the [Directories](#) Chapter 15 later on.

## Symbolic Links (S\_IFLNK)

The data fork contains the contents of the symbolic link. The format of the link is determined by the `di_format` value and can be one of 2 formats:

- `XFS_DINODE_FMT_LOCAL`: The symbolic link is fully contained within the inode. This is accessed by casting the return value from `XFS_DFORK_DPTR` to `char*`.
- `XFS_DINODE_FMT_EXTENTS`: The actual symlink is located in another filesystem block, the inode contains the extents to these filesystem blocks (`xf_s_bmbt_rec_t*`).

Details for symbolic links is covered in the section about [Symbolic Links](#) Chapter 17.

## Other File Types

For character and block devices (`S_IFCHR` and `S_IFBLK`), cast the value from `XFS_DFORK_DPTR` to `xf_s_dev_t*`.

## Attribute Fork

The attribute fork in the inode always contains the location of the extended attributes associated with the inode.

The location of the attribute fork in the inode's literal area is specified by the `di_forkoff` value in the inode's core. If this value is zero, the inode does not contain any extended attributes. If non-zero, the attribute fork's byte offset into the literal area can be computed from `di_forkoff × 8`. Attributes must be allocated on a 64-bit boundary on the disk. To access the extended attributes in code, use the `XFS_DFORK_PTR` macro specifying `XFS_ATTR_FORK` for the "which" parameter. Alternatively, the `XFS_DFORK_APTR` macro can be used.

The structure of the attribute fork depends on the `di_aformat` value in the inode. It can be one of the following values:

- `XFS_DINODE_FMT_LOCAL`: The extended attributes are contained entirely within the inode. This is accessed by casting the value from `XFS_DFORK_APTR` to `xfst_attr_shortform_t*`.
- `XFS_DINODE_FMT_EXTENTS`: The attributes are located in another filesystem block, the inode contains an array of pointers to these filesystem blocks. They are accessed by casting the value from `XFS_DFORK_APTR` to `xfst_bmbt_rec_t*`.
- `XFS_DINODE_FMT_BTREE`: The extents for the attributes are contained in the leaves of a B+tree. The inode contains the root node of the tree and is accessed by casting the value from `XFS_DFORK_APTR` to `xfst_bmdr_block_t*`.

Detailed information on the layouts of extended attributes are covered in the [Extended Attributes](#) Chapter 16 in this document.

## Extended Attribute Versions

Extended attributes come in two versions: "attr1" or "attr2". The attribute version is specified by the `XFS_SB_VERSION2_ATTR2BIT` flag in the `sb_features2` field in the superblock. It determines how the inode's extra space is split between `di_u` and `di_a` forks which also determines how the `di_forkoff` value is maintained in the inode's core.

With "attr1" attributes, the `di_forkoff` is set to somewhere in the middle of the space between the core and end of the inode and never changes (which has the effect of artificially limiting the space for data information). As the data fork grows, when it gets to `di_forkoff`, it will move the data to the next format level (ie. local < extent < btree). If very little space is used for either attributes or data, then a good portion of the available inode space is wasted with this version.

"attr2" was introduced to maximum the utilisation of the inode's literal area. The `di_forkoff` starts at the end of the inode and works its way to the data fork as attributes are added. Attr2 is highly recommended if extended attributes are used.

The following diagram compares the two versions:

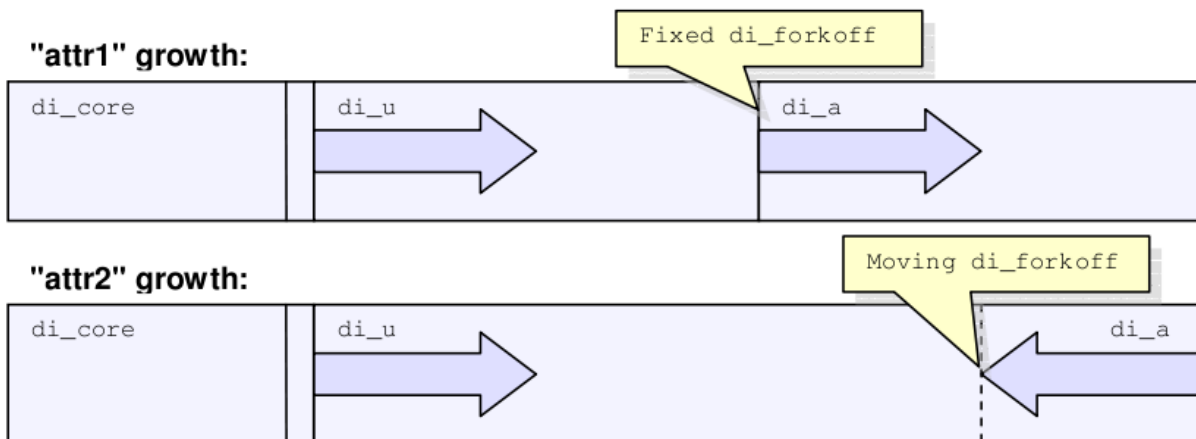


Figure 13.3: Extended attribute layouts

Note that because `di_forkoff` is an 8-bit value measuring units of 8 bytes, the maximum size of an inode is  $2^8 \times 2^3 = 2^{11} = 2048$  bytes.

## Chapter 14

# Data Extents

XFS manages space using extents, which are defined as a starting location and length. A fork in an XFS inode maps a logical offset to a space extent. This enables a file's extent map to support sparse files (i.e. "holes" in the file). A flag is also used to specify if the extent has been preallocated but has not yet been written (unwritten extent).

A file can have more than one extent if one chunk of contiguous disk space is not available for the file. As a file grows, the XFS space allocator will attempt to keep space contiguous and to merge extents. If more than one file is being allocated space in the same AG at the same time, multiple extents for the files will occur as the extent allocations interleave. The effect of this can vary depending on the extent allocator used in the XFS driver.

An extent is 128 bits in size and uses the following packed layout:

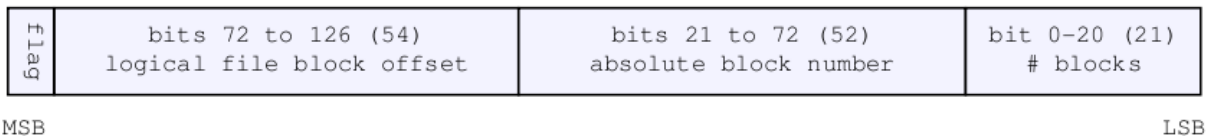


Figure 14.1: Extent record format

The extent is represented by the `xfst_bmbt_rec` structure which uses a big endian format on-disk. In-core management of extents use the `xfst_bmbt_irec` structure which is the unpacked version of `xfst_bmbt_rec`:

```
struct xfst_bmbt_irec {
    xfst_fileoff_t      br_startoff;
    xfst_fsblock_t     br_startblock;
    xfst_filblks_t     br_blockcount;
    xfst_extst_t       br_state;
};
```

### `br_startoff`

Logical block offset of this mapping.

### `br_startblock`

Filesystem block of this mapping.

**br\_blockcount**

The length of this mapping.

**br\_state**

The extent `br_state` field uses the following enum declaration:

```
typedef enum {
    XFS_EXT_NORM,
    XFS_EXT_UNWRITTEN,
    XFS_EXT_INVALID
} xfs_extst_t;
```

Some other points about extents:

- The `xfs_bmbt_rec_32_t` and `xfs_bmbt_rec_64_t` structures were effectively the same as `xfs_bmbt_rec_t`, just different representations of the same 128 bits in on-disk big endian format. `xfs_bmbt_rec_32_t` was removed and `xfs_bmbt_rec_64_t` renamed to `xfs_bmbt_rec_t` some time ago.
- When a file is created and written to, XFS will endeavour to keep the extents within the same AG as the inode. It may use a different AG if the AG is busy or there is no space left in it.
- If a file is zero bytes long, it will have no extents and `di_nblocks` and `di_nextents` will be zero. Any file with data will have at least one extent, and each extent can use from 1 to over 2 million blocks ( $2^{21}$ ) on the filesystem. For a default 4KB block size filesystem, a single extent can be up to 8GB in length.

The following two subsections cover the two methods of storing extent information for a file. The first is the fastest and simplest where the inode completely contains an extent array to the file's data. The second is slower and more complex B+tree which can handle thousands to millions of extents efficiently.

## Extent List

If the entire extent list is short enough to fit within the inode's fork region, we say that the fork is in "extent list" format. This is the most optimal in terms of speed and resource consumption. The trade-off is the file can only have a few extents before the inode runs out of space.

The data fork of the inode contains an array of extents; the size of the array is determined by the inode's `di_nextents` value.

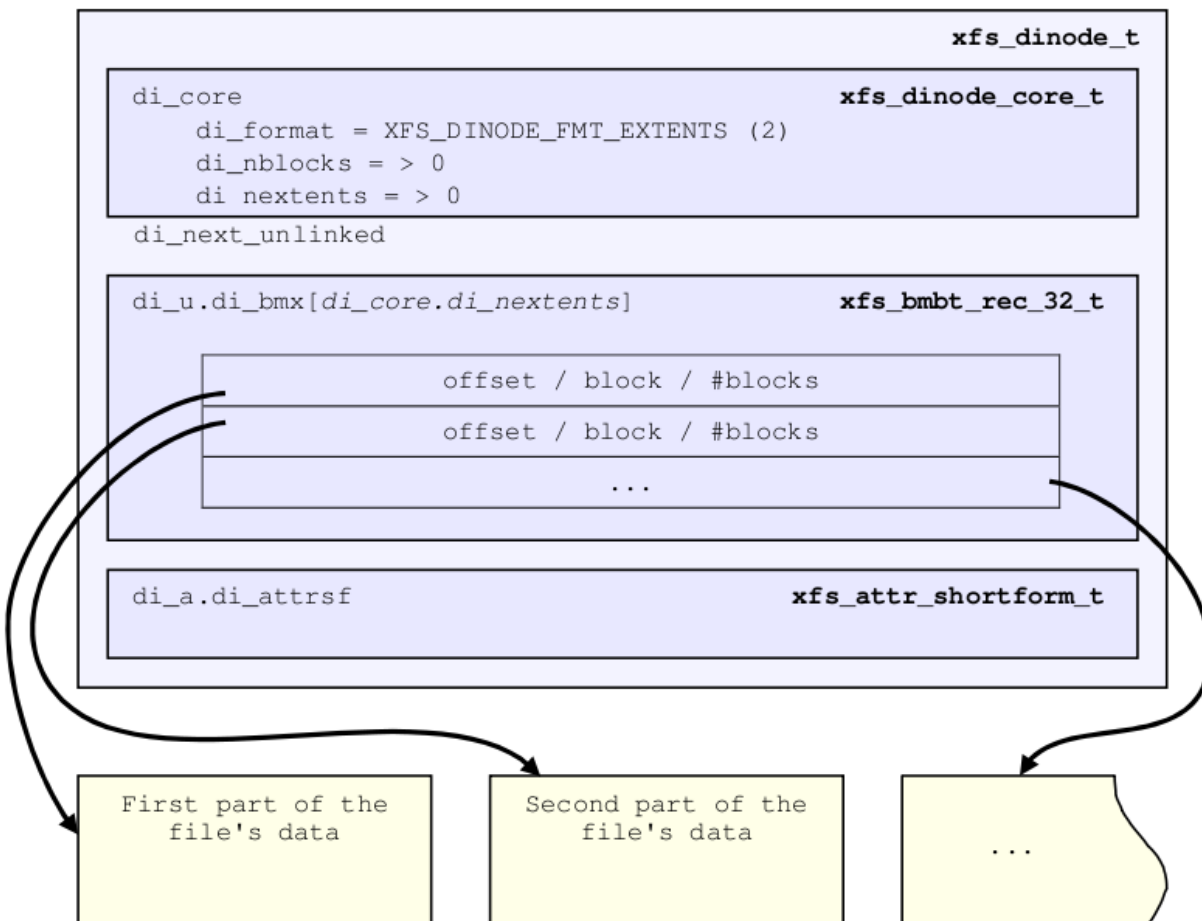


Figure 14.2: Inode data fork extent layout

The number of extents that can fit in the inode depends on the inode size and `di_forkoff`. For a default 256 byte inode with no extended attributes, a file can have up to 9 extents with this format. On a default v5 filesystem with 512 byte inodes, a file can have up to 21 extents with this format. Beyond that, extents have to use the B+tree format.

### xfs\_db Inode Data Fork Extents Example

An 8MB file with one extent:

```

xfs_db> inode <inode#>
xfs_db> p
core.magic = 0x494e
core.mode = 0100644
core.version = 1
core.format = 2 (extents)
...
core.size = 8294400
core.nblocks = 2025
core.extsize = 0

```

```

core.nextents = 1
core.naextents = 0
core.forkoff = 0
...
u.bmx[0] = [startoff,startblock,blockcount,extentflag]
           0:[0,25356,2025,0]

```

A 24MB file with three extents:

```

xfs_db> inode <inode#>
xfs_db> p
...
core.format = 2 (extents)
...
core.size = 24883200
core.nblocks = 6075
core.nextents = 3
...
u.bmx[0-2] = [startoff,startblock,blockcount,extentflag]
             0:[0,27381,2025,0]
             1:[2025,31431,2025,0]
             2:[4050,35481,2025,0]

```

Raw disk version of the inode with the third extent highlighted (d<sub>i</sub>\_u starts at offset 0x64):

```

xfs_db> type text
xfs_db> p
00: 49 4e 81 a4 01 02 00 01 00 00 00 00 00 00 00 00 IN.....
10: 00 00 00 01 00 00 00 00 00 00 00 00 00 00 01 .....
20: 44 b6 88 dd 2f 8a ed d0 44 b6 88 f7 10 8c 5b de D.....D.....
30: 44 b6 88 f7 10 8c 5b d0 00 00 00 00 01 7b b0 00 D.....
40: 00 00 00 00 00 00 17 bb 00 00 00 00 00 00 03 .....
50: 00 00 00 02 00 00 00 00 00 00 00 00 00 00 00 .....
60: ff ff ff ff 00 00 00 00 00 00 00 00 00 00 0d .....
70: 5e a0 07 e9 00 00 00 00 00 0f d2 00 00 00 0f .....
80: 58 e0 07 e9 00 00 00 00 00 1f a4 00 00 00 11 X.....
90: 53 20 07 e9 00 00 00 00 00 00 00 00 00 00 00 S.....
a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
be: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
co: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
do: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
fo: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....

```

We can expand the highlighted section into the following bit array from MSB to LSB with the file offset and the block count highlighted:

```

127-96: 0000 0000 0000 0000 0000 0000 0000 0000
95-64:  0000 0000 0001 1111 1010 0100 0000 0000
63-32: 0000 0000 0000 0000 0000 0000 0000 1111
31-0 : 0101 1000 1110 0000 0000 0111 1110 1001

```

Grouping by highlights we get:

```

file offset = 0x0fd2 (4050)
start block = 0x7ac7 (31431)
block count = 0x07e9 (2025)

```



A 4MB file with two extents and a hole in the middle, the first extent containing 64KB of data, the second about 4MB in containing 32KB (write 64KB, lseek 4MB, write 32KB operations):

```
xfs_db> inode <inode#>
xfs_db> p
...
core.format = 2 (extents)
...
core.size = 4063232
core.nblocks = 24
core.nextents = 2
...
u.bmx[0-1] = [startoff, startblock, blockcount, extentflag]
             0: [0, 37506, 16, 0]
             1: [984, 37522, 8, 0]
```

## B+tree Extent List

To manage extent maps that cannot fit in the inode fork area, XFS uses [long format B+trees](#) Section 9.2. The root node of the B+tree is stored in the inode's data fork. All block pointers for extent B+trees are 64-bit filesystem block numbers.

For a single level B+tree, the root node points to the B+tree's leaves. Each leaf occupies one filesystem block and contains a header and an array of extents sorted by the file's offset. Each leaf has left and right (or backward and forward) block pointers to adjacent leaves. For a standard 4KB filesystem block, a leaf can contain up to 254 extents before a B+tree rebalance is triggered.

For a multi-level B+tree, the root node points to other B+tree nodes which eventually point to the extent leaves. B+tree keys are based on the file's offset and have pointers to the next level down. Nodes at each level in the B+tree also have pointers to the adjacent nodes.

The base B+tree node is used for extents, directories and extended attributes. The structures used for an inode's B+tree root are:

```
struct xfs_bmdr_block {
    __be16          bb_level;
    __be16          bb_numrecs;
};
struct xfs_bmbt_key {
    xfs_fileoff_t   br_startoff;
};
typedef xfs_fsblock_t xfs_bmbt_ptr_t, xfs_bmdr_ptr_t;
```

- On disk, the B+tree node starts with the `xfs_bmdr_block_t` header followed by an array of `xfs_bmbt_key_t` values and then an array of `xfs_bmbt_ptr_t` values. The size of both arrays is specified by the header's `bb_numrecs` value.
- The root node in the inode can only contain up to 9 key/pointer pairs for a standard 256 byte inode before a new level of nodes is added between the root and the leaves. This will be less if `di_forkoff` is not zero (i.e. attributes are in use on the inode).
- The magic number for a BMBT block is "BMAP" (0x424d4150). On a v5 filesystem, this is "BMA3" (0x424d4133).

- For intermediate nodes, the data following `xf_btree_lblock` is the same as the root node: array of `xf_bmbt_key` value followed by an array of `xf_bmbt_ptr_t` values that starts halfway through the block (offset 0x808 for a 4096 byte filesystem block).
  - For leaves, an array of `xf_bmbt_rec` extents follow the `xf_btree_lblock` header.
  - Nodes and leaves use the same value for `bb_magic`.
  - The `bb_level` value determines if the node is an intermediate node or a leaf. Leaves have a `bb_level` of zero, nodes are one or greater.
  - Intermediate nodes, like leaves, can contain up to 254 pointers to leaf blocks for a standard 4KB filesystem block size as both the keys and pointers are 64 bits in size.
-

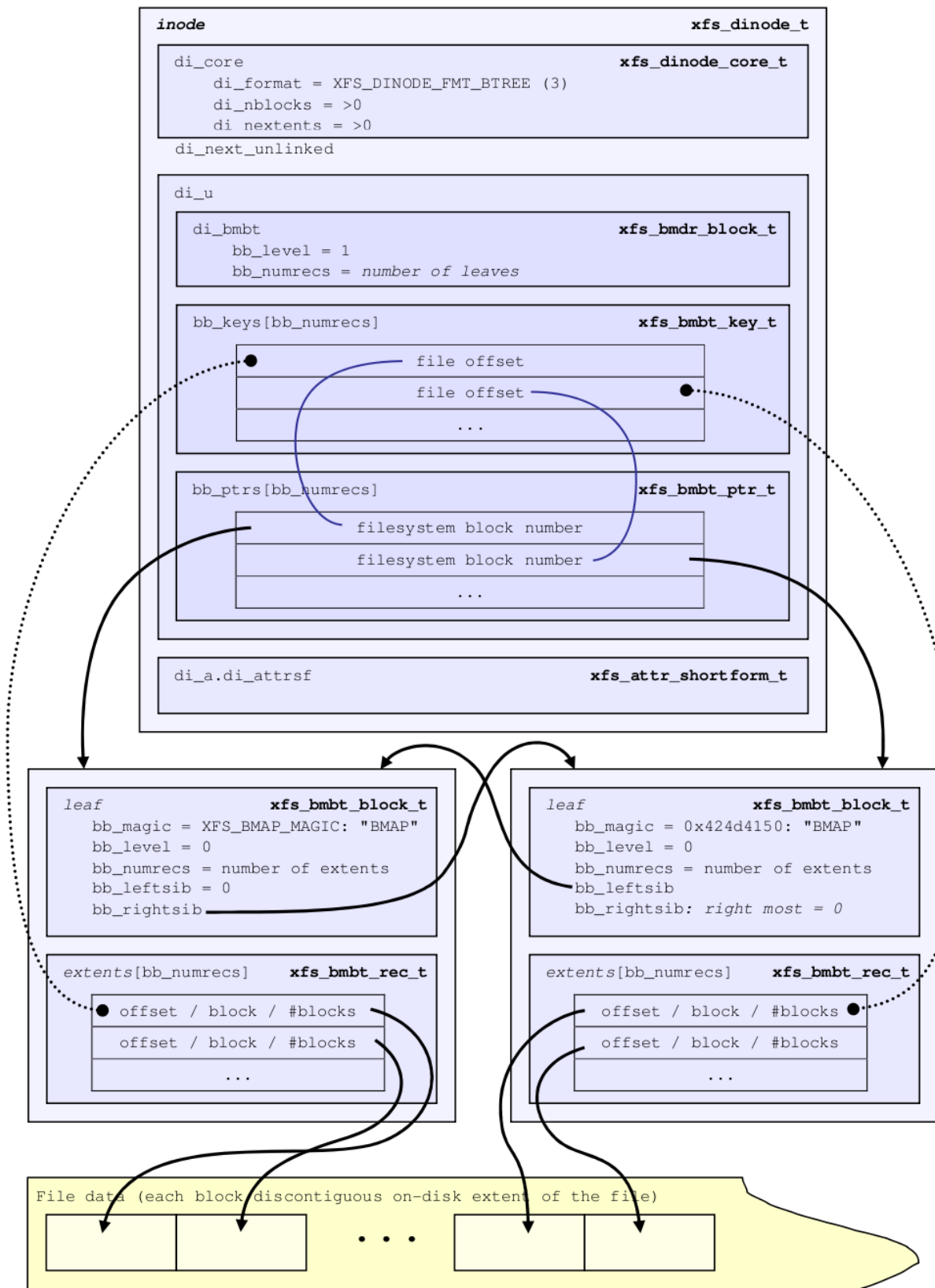


Figure 14.3: Single level extent B+tree

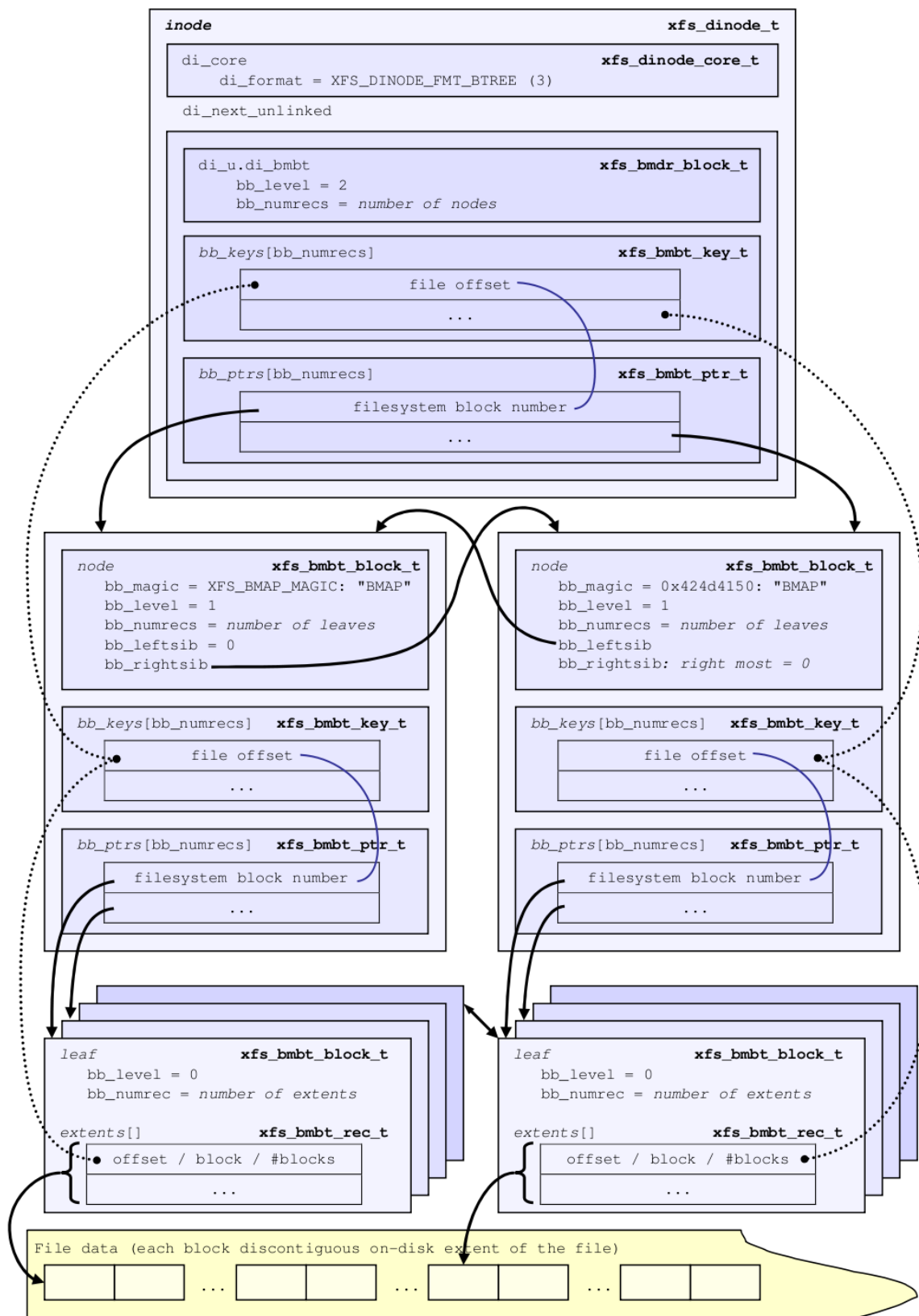


Figure 14.4: Multiple level extent B+tree

## xfs\_db bmbt Example

In this example, we dissect the data fork of a VM image that is sufficiently sparse and interleaved to have become a B+tree.

```
xfs_db> inode 132
xfs_db> p
core.magic = 0x494e
core.mode = 0100600
core.version = 3
core.format = 3 (btree)
...
u3.bmbt.level = 1
u3.bmbt.numrecs = 3
u3.bmbt.keys[1-3] = [startoff] 1:[0] 2:[9072] 3:[13136]
u3.bmbt.ptrs[1-3] = 1:8568 2:8569 3:8570
```

As you can see, the block map B+tree is rooted in the inode. This tree has two levels, so let's go down a level to look at the records:

```
xfs_db> addr u3.bmbt.ptrs[1]
xfs_db> p
magic = 0x424d4133
level = 0
numrecs = 251
leftsib = null
rightsib = 8569
bno = 68544
lsn = 0x100000006
uuid = 9579903c-333f-4673-a7d4-3254c05816ea
owner = 132
crc = 0xc61513dc (correct)
recs[1-251] = [startoff,startblock,blockcount,extentflag]
  1:[0,8520,48,0] 2:[48,4421,16,0] 3:[80,9136,16,0] 4:[96,8569,16,0]
  5:[144,8601,32,0] 6:[192,8637,16,0] 7:[240,8680,16,0] 8:[288,9870,16,0]
  9:[320,9920,16,0] 10:[336,9950,16,0] 11:[384,4004,32,0]
  12:[432,6771,16,0] 13:[480,2702,16,0] 14:[528,8420,16,0]
  ...
```

## Chapter 15

# Directories

---

**Note**

Only v2 directories covered here. v1 directories are obsolete.

---

**Note**

The term “block” in this section will refer to directory blocks, not filesystem blocks unless otherwise specified.

---

The size of a “directory block” is defined by the [superblock’s](#) Section 10.1 `sb_dirblklog` value. The size in bytes = `sb_blocksize × 2sb_dirblklog`. For example, if `sb_blocksize = 4096` and `sb_dirblklog = 2`, the directory block size is 16384 bytes. Directory blocks are always allocated in multiples based on `sb_dirblklog`. Directory blocks cannot be more than 65536 bytes in size.

All directory entries contain the following “data”:

- The entry’s name (counted string consisting of a single byte `nameLen` followed by `name` consisting of an array of 8-bit chars without a NULL terminator).
- The entry’s absolute [inode number](#) Section 10.3.1, which are always 64 bits (8 bytes) in size except a special case for shortform directories.
- An `offset` or `tag` used for iterative `readdir` calls.

All non-shortform directories also contain two additional structures: “leaves” and “freespace indexes”.

- Leaves contain the sorted hashed name value (`xf_s_da_hashname()` in `xf_s_da_btree.c`) and associated “address” which points to the effective offset into the directory’s data structures. Leaves are used to optimise lookup operations.
- Freespace indexes contain free space/empty entry tracking for quickly finding an appropriately sized location for new entries. They maintain the largest free space for each “data” block.

A few common types are used for the directory structures:

```
typedef __uint16_t xfs_dir2_data_off_t;
typedef __uint32_t xfs_dir2_dataptr_t;
```

---

## Short Form Directories

- Directory entries are stored within the inode.
- The only data stored is the name, inode number, and offset. No “leaf” or “freespace index” information is required as an inode can only store a few entries.
- “.” is not stored (as it’s in the inode itself), and “..” is a dedicated parent field in the header.
- The number of directories that can be stored in an inode depends on the [inode](#) Chapter 13 size, the number of entries, the length of the entry names, and extended attribute data.
- Once the number of entries exceeds the space available in the inode, the format is converted to a [block directory](#) Section 15.2.
- Shortform directory data is packed as tightly as possible on the disk with the remaining space zeroed:

```
typedef struct xfs_dir2_sf {
    xfs_dir2_sf_hdr_t      hdr;
    xfs_dir2_sf_entry_t    list[1];
} xfs_dir2_sf_t;
```

### hdr

Short form directory header.

### list

An array of variable-length directory entry records.

```
typedef struct xfs_dir2_sf_hdr {
    __uint8_t      count;
    __uint8_t      i8count;
    xfs_dir2_inou_t parent;
} xfs_dir2_sf_hdr_t;
```

### count

Number of directory entries.

### i8count

Number of directory entries requiring 64-bit entries, if any inode numbers require 64-bits. Zero otherwise.

### parent

The absolute inode number of this directory’s parent.

```
typedef struct xfs_dir2_sf_entry {
    __uint8_t      namelen;
    xfs_dir2_sf_off_t offset;
    __uint8_t      name[1];
    __uint8_t      ftype;
    xfs_dir2_inou_t inumber;
} xfs_dir2_sf_entry_t;
```

### namelen

Length of the name, in bytes.

---

**offset**

Offset tag used to assist with directory iteration.

**name**

The name of the directory entry. The entry is not NULL-terminated.

**ftype**

The type of the inode. This is used to avoid reading the inode while iterating a directory. The XFS\_SB\_VERSION2\_FTYPE feature must be set, or this field will not be present.

**inumber**

The inode number that this entry points to. The length is either 32 or 64 bits, depending on whether `icount` or `iscount`, respectively, are set in the header.

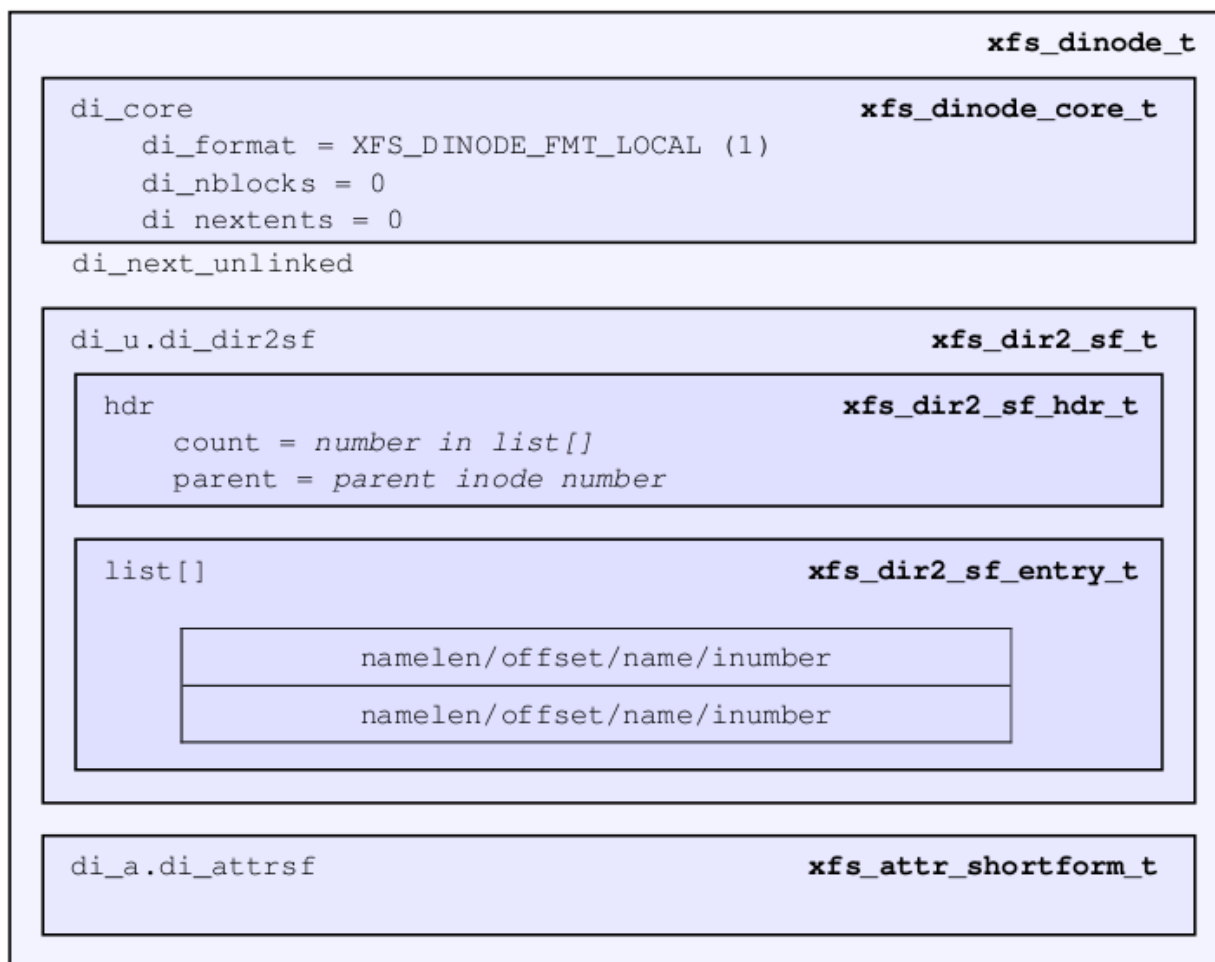


Figure 15.1: Short form directory layout

- Inode numbers are stored using 4 or 8 bytes depending on whether all the inode numbers for the directory fit in 4 bytes (32 bits) or not. If all inode numbers fit in 4 bytes, the header's `count` value specifies the number of entries in the directory and `iscount` will be zero. If any inode number exceeds 4 bytes, all inode numbers will be 8 bytes



in size and the header's `i8count` value specifies the number of entries requiring larger inodes. `i4count` is still the number of entries. The following union covers the shortform inode number structure:

```
typedef struct { __uint8_t i[8]; } xfs_dir2_ino8_t;
typedef struct { __uint8_t i[4]; } xfs_dir2_ino4_t;
typedef union {
    xfs_dir2_ino8_t    i8;
    xfs_dir2_ino4_t    i4;
} xfs_dir2_inou_t;
```

## xfs\_db Short Form Directory Example

A directory is created with 4 files, all inode numbers fitting within 4 bytes:

```
xfs_db> inode <inode#>
xfs_db> p
core.magic = 0x494e
core.mode = 040755
core.version = 1
core.format = 1 (local)
core.nlinkv1 = 2
...
core.size = 94
core.nblocks = 0
core.extsize = 0
core.nextents = 0
...
u.sfdir2.hdr.count = 4
u.sfdir2.hdr.i8count = 0
u.sfdir2.hdr.parent.i4 = 128                /* parent = root inode */
u.sfdir2.list[0].namelen = 15
u.sfdir2.list[0].offset = 0x30
u.sfdir2.list[0].name = "frame000000.tst"
u.sfdir2.list[0].inumber.i4 = 25165953
u.sfdir2.list[1].namelen = 15
u.sfdir2.list[1].offset = 0x50
u.sfdir2.list[1].name = "frame000001.tst"
u.sfdir2.list[1].inumber.i4 = 25165954
u.sfdir2.list[2].namelen = 15
u.sfdir2.list[2].offset = 0x70
u.sfdir2.list[2].name = "frame000002.tst"
u.sfdir2.list[2].inumber.i4 = 25165955
u.sfdir2.list[3].namelen = 15
u.sfdir2.list[3].offset = 0x90
u.sfdir2.list[3].name = "frame000003.tst"
u.sfdir2.list[3].inumber.i4 = 25165956
```

The raw data on disk with the first entry highlighted. The six byte header precedes the first entry:

```
xfs_db> type text
xfs_db> p
00: 49 4e 41 ed 01 01 00 02 00 00 00 00 00 00 00 INA.....
10: 00 00 00 02 00 00 00 00 00 00 00 00 00 00 02 .....
20: 44 ad 3a 83 1d a9 4a d0 44 ad 3a ab 0b c7 a7 d0 D.....J.D.....
30: 44 ad 3a ab 0b c7 a7 d0 00 00 00 00 00 00 00 5e D.....
```

```

40: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
50: 00 00 00 02 00 00 00 00 00 00 00 00 00 00 00 .....
60: ff ff ff ff 04 00 00 00 00 80 0f 00 30 66 72 61 .....0fra
70: 6d 65 30 30 30 30 30 30 2e 74 73 74 01 80 00 81 me000000.tst...
80: 0f 00 50 66 72 61 6d 65 30 30 30 30 30 31 2e 74 ..Pframe000001.t
90: 73 74 01 80 00 82 0f 00 70 66 72 61 6d 65 30 30 st.....pframe00
a0: 30 30 30 32 2e 74 73 74 01 80 00 83 0f 00 90 66 0002.tst.....
b0: 72 61 6d 65 30 30 30 30 30 33 2e 74 73 74 01 80 rame000003.tst..
c0: 00 84 00 00 00 00 00 00 00 00 00 00 00 00 00 .....

```

Next, an entry is deleted (frame000001.tst), and any entries after the deleted entry are moved or compacted to “cover” the hole:

```

xfs_db> inode <inode#>
xfs_db> p
core.magic = 0x494e
core.mode = 040755
core.version = 1
core.format = 1 (local)
core.nlinkv1 = 2
...
core.size = 72
core.nblocks = 0
core.extsize = 0
core.nextents = 0
...
u.sfdir2.hdr.count = 3
u.sfdir2.hdr.i8count = 0
u.sfdir2.hdr.parent.i4 = 128
u.sfdir2.list[0].namelen = 15
u.sfdir2.list[0].offset = 0x30
u.sfdir2.list[0].name = "frame000000.tst"
u.sfdir2.list[0].inumber.i4 = 25165953
u.sfdir2.list[1].namelen = 15
u.sfdir2.list[1].offset = 0x70
u.sfdir2.list[1].name = "frame000002.tst"
u.sfdir2.list[1].inumber.i4 = 25165955
u.sfdir2.list[2].namelen = 15
u.sfdir2.list[2].offset = 0x90
u.sfdir2.list[2].name = "frame000003.tst"
u.sfdir2.list[2].inumber.i4 = 25165956

```

Raw disk data, the space beyond the shortform entries is invalid and could be non-zero:

```

xfs_db> type text
xfs_db> p
00: 49 4e 41 ed 01 01 00 02 00 00 00 00 00 00 00 INA.....
10: 00 00 00 02 00 00 00 00 00 00 00 00 00 00 03 .....
20: 44 b2 45 a2 09 fd e4 50 44 b2 45 a3 12 ee b5 d0 D.E....PD.E....
30: 44 b2 45 a3 12 ee b5 d0 00 00 00 00 00 00 00 48 D.E.....H
40: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
50: 00 00 00 02 00 00 00 00 00 00 00 00 00 00 00 .....
60: ff ff ff ff 03 00 00 00 00 80 0f 00 30 66 72 61 .....0fra
70: 6d 65 30 30 30 30 30 30 2e 74 73 74 01 80 00 81 me000000.tst...
80: 0f 00 70 66 72 61 6d 65 30 30 30 30 30 32 2e 74 ..pframe000002.t
90: 73 74 01 80 00 83 0f 00 90 66 72 61 6d 65 30 30 st.....frame00
a0: 30 30 30 33 2e 74 73 74 01 80 00 84 0f 00 90 66 0003.tst.....f

```

```
b0: 72 61 6d 65 30 30 30 30 30 33 2e 74 73 74 01 80 rame000003.tst..  
c0: 00 84 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
```

This is an example of mixed 4-byte and 8-byte inodes in a directory:

```
xfs_db> inode 1024  
xfs_db> p  
core.magic = 0x494e  
core.mode = 040755  
core.version = 3  
core.format = 1 (local)  
core.nlinkv2 = 9  
...  
core.size = 125  
core.nblocks = 0  
core.extsize = 0  
core.nextents = 0  
...  
u3.sfd3.hdr.count = 7  
u3.sfd3.hdr.i8count = 4  
u3.sfd3.hdr.parent.i8 = 1024  
u3.sfd3.list[0].namelen = 3  
u3.sfd3.list[0].offset = 0x60  
u3.sfd3.list[0].name = "git"  
u3.sfd3.list[0].inumber.i8 = 1027  
u3.sfd3.list[0].filetype = 2  
u3.sfd3.list[1].namelen = 4  
u3.sfd3.list[1].offset = 0x70  
u3.sfd3.list[1].name = "home"  
u3.sfd3.list[1].inumber.i8 = 13422826546  
u3.sfd3.list[1].filetype = 2  
u3.sfd3.list[2].namelen = 10  
u3.sfd3.list[2].offset = 0x80  
u3.sfd3.list[2].name = "mike"  
u3.sfd3.list[2].inumber.i8 = 4299308032  
u3.sfd3.list[2].filetype = 2  
u3.sfd3.list[3].namelen = 3  
u3.sfd3.list[3].offset = 0x98  
u3.sfd3.list[3].name = "mtr"  
u3.sfd3.list[3].inumber.i8 = 13433252916  
u3.sfd3.list[3].filetype = 2  
u3.sfd3.list[4].namelen = 3  
u3.sfd3.list[4].offset = 0xa8  
u3.sfd3.list[4].name = "vms"  
u3.sfd3.list[4].inumber.i8 = 16647516355  
u3.sfd3.list[4].filetype = 2  
u3.sfd3.list[5].namelen = 5  
u3.sfd3.list[5].offset = 0xb8  
u3.sfd3.list[5].name = "rsync"  
u3.sfd3.list[5].inumber.i8 = 3494912  
u3.sfd3.list[5].filetype = 2  
u3.sfd3.list[6].namelen = 3  
u3.sfd3.list[6].offset = 0xd0  
u3.sfd3.list[6].name = "tmp"  
u3.sfd3.list[6].inumber.i8 = 1593379  
u3.sfd3.list[6].filetype = 2
```

## Block Directories

When the shortform directory space exceeds the space in an inode, the directory data is moved into a new single directory block outside the inode. The inode's format is changed from "local" to "extent" Following is a list of points about block directories.

- All directory data is stored within the one directory block, including "." and ".." entries which are mandatory.
- The block also contains "leaf" and "freespace index" information.
- The location of the block is defined by the inode's in-core [extent list](#) Section 14.1: the `di_u.u_bmx[0]` value. The file offset in the extent must always be zero and the `length = (directory block size / filesystem block size)`. The block number points to the filesystem block containing the directory data.
- Block directory data is stored in the following structures:

```
#define XFS_DIR2_DATA_FD_COUNT 3
typedef struct xfs_dir2_block {
    xfs_dir2_data_hdr_t      hdr;
    xfs_dir2_data_union_t    u[1];
    xfs_dir2_leaf_entry_t    leaf[1];
    xfs_dir2_block_tail_t    tail;
} xfs_dir2_block_t;
```

### hdr

Directory block header. On a v5 filesystem this is `xfs_dir3_data_hdr_t`.

### u

Union of directory and unused entries.

### leaf

Hash values of the entries in this block.

### tail

Bookkeeping for the leaf entries.

```
typedef struct xfs_dir2_data_hdr {
    __uint32_t      magic;
    xfs_dir2_data_free_t    bestfree[XFS_DIR2_DATA_FD_COUNT];
} xfs_dir2_data_hdr_t;
```

### magic

Magic number for this directory block.

### bestfree

An array pointing to free regions in the directory block.

On a v5 filesystem, directory and attribute blocks are formatted with v3 headers, which contain extra data:

```
struct xfs_dir3_blk_hdr {
    __be32      magic;
    __be32      crc;
    __be64      blkno;
    __be64      lsn;
    uuid_t      uuid;
    __be64      owner;
};
```

**magic**

Magic number for this directory block.

**crc**

Checksum of the directory block.

**blkno**

Block number of this directory block.

**lsn**

Log sequence number of the last write to this block.

**uuid**

The UUID of this block, which must match either `sb_uuid` or `sb_meta_uuid` depending on which features are set.

**owner**

The inode number that this directory block belongs to.

```
struct xfs_dir3_data_hdr {
    struct xfs_dir3_blk_hdr    hdr;
    xfs_dir2_data_free_t      best_free[XFS_DIR2_DATA_FD_COUNT];
    __be32                    pad;
};
```

**hdr**

The v5 directory/attribute block header.

**best\_free**

An array pointing to free regions in the directory block.

**pad**

Padding to maintain a 64-bit alignment.

Within the block, data structures are as follows:

```
typedef struct xfs_dir2_data_free {
    xfs_dir2_data_off_t    offset;
    xfs_dir2_data_off_t    length;
} xfs_dir2_data_free_t;
```

**offset**

Block offset of a free block, in bytes.

**length**

Length of the free block, in bytes.

Space inside the directory block can be used for directory entries or unused entries. This is signified via a union of the two types:

```
typedef union {
    xfs_dir2_data_entry_t    entry;
    xfs_dir2_data_unused_t   unused;
} xfs_dir2_data_union_t;
```

**entry**

A directory entry.

**unused**

An unused entry.

```
typedef struct xfs_dir2_data_entry {
    xfs_ino_t          inumber;
    __uint8_t         namelen;
    __uint8_t         name[1];
    __uint8_t         ftype;
    xfs_dir2_data_off_t tag;
} xfs_dir2_data_entry_t;
```

**inumber**

The inode number that this entry points to.

**namelen**

Length of the name, in bytes.

**name**

The name associated with this entry.

**ftype**

The type of the inode. This is used to avoid reading the inode while iterating a directory. The XFS\_SB\_VER\_SION2\_FTYPE feature must be set, or this field will not be present.

**tag**

Starting offset of the entry, in bytes. This is used for directory iteration.

```
typedef struct xfs_dir2_data_unused {
    __uint16_t        freetag; /* 0xffff */
    xfs_dir2_data_off_t length;
    xfs_dir2_data_off_t tag;
} xfs_dir2_data_unused_t;
```

**freetag**

Magic number signifying that this is an unused entry. Must be 0xFFFF.

**length**

Length of this unused entry, in bytes.

**tag**

Starting offset of the entry, in bytes.

```
typedef struct xfs_dir2_leaf_entry {
    xfs_dahash_t      hashval;
    xfs_dir2_dataptr_t address;
} xfs_dir2_leaf_entry_t;
```

**hashval**

Hash value of the name of the directory entry. This is used to speed up entry lookups.

---

**address**

Block offset of the entry, in eight byte units.

```
typedef struct xfs_dir2_block_tail {  
    __uint32_t      count;  
    __uint32_t      stale;  
} xfs_dir2_block_tail_t;
```

**count**

Number of leaf entries.

**stale**

Number of free leaf entries.

Following is a diagram of how these pieces fit together for a block directory.

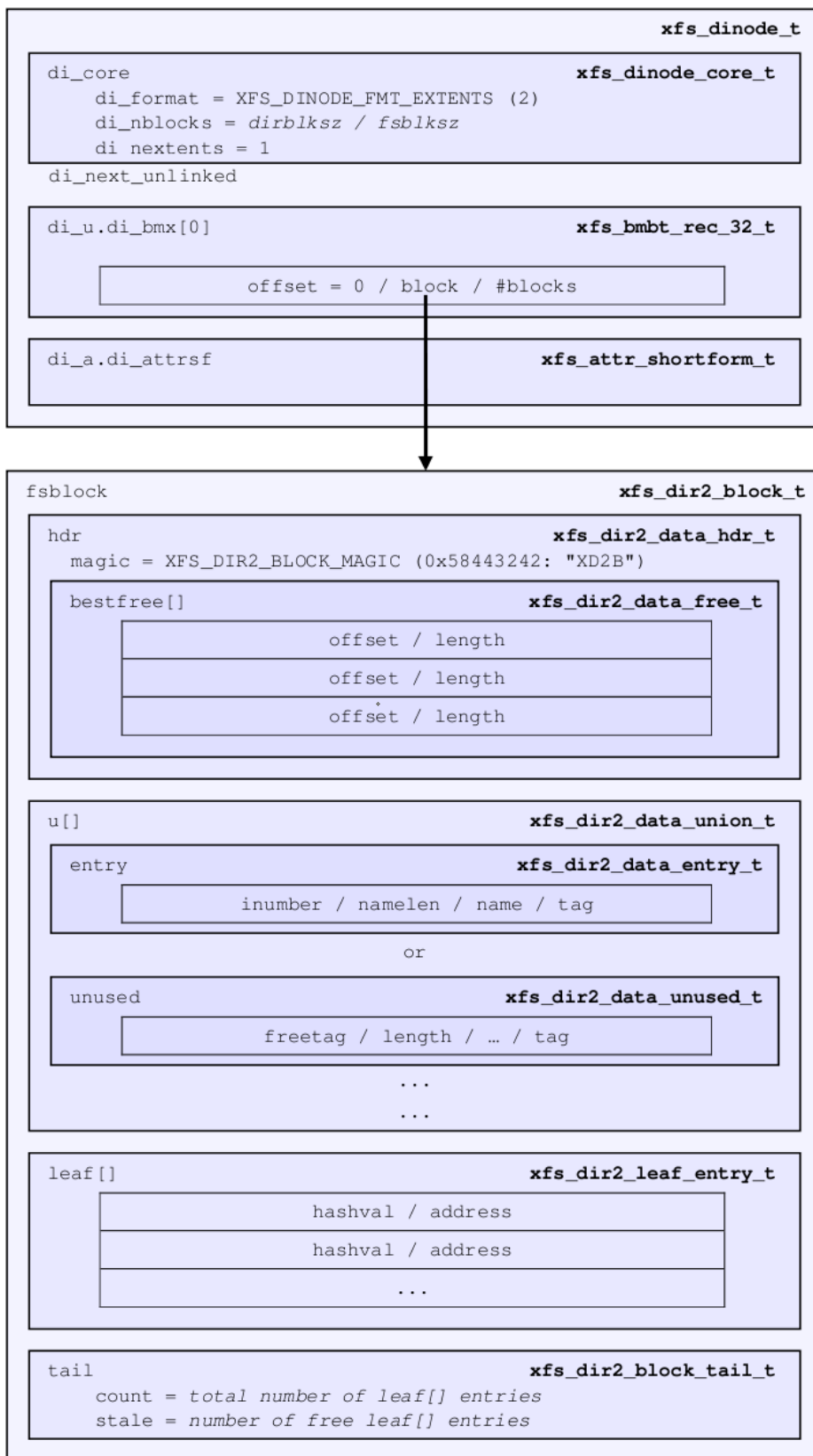


Figure 15.2: Block directory layout



- The magic number in the header is “XD2B” (0x58443242), or “XDB3” (0x58444233) on a v5 filesystem.
- The `tag` in the `xfs_dir2_data_entry_t` structure stores its offset from the start of the block.
- The start of a free space region is marked with the `xfs_dir2_data_unused_t` structure where the `freetag` is `0xffff`. The `freetag` and `length` overwrites the `inumber` for an entry. The `tag` is located at `length - sizeof(tag)` from the start of the unused entry on-disk.
- The `bestfree` array in the header points to as many as three of the largest spaces of free space within the block for storing new entries sorted by largest to third largest. If there are less than 3 empty regions, the remaining `bestfree` elements are zeroed. The `offset` specifies the offset from the start of the block in bytes, and the `length` specifies the size of the free space in bytes. The location each points to must contain the above `xfs_dir2_data_unused_t` structure. As a block cannot exceed 64KB in size, each is a 16-bit value. `bestfree` is used to optimise the time required to locate space to create an entry. It saves scanning through the block to find a location suitable for every entry created.
- The `tail` structure specifies the number of elements in the `leaf` array and the number of `stale` entries in the array. The `tail` is always located at the end of the block. The `leaf` data immediately precedes the `tail` structure.
- The `leaf` array, which grows from the end of the block just before the `tail` structure, contains an array of hash/address pairs for quickly looking up a name by a hash value. Hash values are covered by the introduction to directories. The `address` on-disk is the offset into the block divided by 8 (`XFS_DIR2_DATA_ALIGN`). Hash/address pairs are stored on disk to optimise lookup speed for large directories. If they were not stored, the hashes would have to be calculated for all entries each time a lookup occurs in a directory.

## xfs\_db Block Directory Example

A directory is created with 8 entries, directory block size = filesystem block size:

```
xfs_db> sb 0
xfs_db> p
magicnum = 0x58465342
blocksize = 4096
...
dirblklog = 0
...
xfs_db> inode <inode#>
xfs_db> p
core.magic = 0x494e
core.mode = 040755
core.version = 1
core.format = 2 (extents)
core.nlinkv1 = 2
...
core.size = 4096
core.nblocks = 1
core.extsize = 0
core.nextents = 1
...
u.bmx[0] = [startoff,startblock,blockcount,extentflag] 0:[0,2097164,1,0]
```

Go to the “startblock” and show the raw disk data:

```

xfs_db> dblock 0
xfs_db> type text
xfs_db> p
000: 58 44 32 42 01 30 0e 78 00 00 00 00 00 00 00 00 XD2B.0.x.....
010: 00 00 00 00 02 00 00 80 01 2e 00 00 00 00 00 10 .....
020: 00 00 00 00 00 00 00 80 02 2e 2e 00 00 00 00 20 .....
030: 00 00 00 00 02 00 00 81 0f 66 72 61 6d 65 30 30 .....frame00
040: 30 30 30 30 2e 74 73 74 80 8e 59 00 00 00 00 30 0000.tst..Y...0
050: 00 00 00 00 02 00 00 82 0f 66 72 61 6d 65 30 30 .....frame00
060: 30 30 30 31 2e 74 73 74 d0 ca 5c 00 00 00 00 50 0001.tst.....P
070: 00 00 00 00 02 00 00 83 0f 66 72 61 6d 65 30 30 .....frame00
080: 30 30 30 32 2e 74 73 74 00 00 00 00 00 00 70 0002.tst.....p
090: 00 00 00 00 02 00 00 84 0f 66 72 61 6d 65 30 30 .....frame00
0a0: 30 30 30 33 2e 74 73 74 00 00 00 00 00 00 90 0003.tst.....
0b0: 00 00 00 00 02 00 00 85 0f 66 72 61 6d 65 30 30 .....frame00
0c0: 30 30 30 34 2e 74 73 74 00 00 00 00 00 00 b0 0004.tst.....
0d0: 00 00 00 00 02 00 00 86 0f 66 72 61 6d 65 30 30 .....frame00
0e0: 30 30 30 35 2e 74 73 74 00 00 00 00 00 00 d0 0005.tst.....
0f0: 00 00 00 00 02 00 00 87 0f 66 72 61 6d 65 30 30 .....frame00
100: 30 30 30 36 2e 74 73 74 00 00 00 00 00 00 f0 0006.tst.....
110: 00 00 00 00 02 00 00 88 0f 66 72 61 6d 65 30 30 .....frame00
120: 30 30 30 37 2e 74 73 74 00 00 00 00 00 01 10 0007.tst.....
130: ff ff 0e 78 00 00 00 00 00 00 00 00 00 00 00 ...x.....

```

The “leaf” and “tail” structures are stored at the end of the block, so as the directory grows, the middle is filled in:

```

fa0: 00 00 00 00 00 00 01 30 00 00 00 2e 00 00 00 02 .....0.....
fb0: 00 00 17 2e 00 00 00 04 83 a0 40 b4 00 00 00 0e .....
fc0: 93 a0 40 b4 00 00 00 12 a3 a0 40 b4 00 00 00 06 .....
fd0: b3 a0 40 b4 00 00 00 0a c3 a0 40 b4 00 00 00 1e .....
fe0: d3 a0 40 b4 00 00 00 22 e3 a0 40 b4 00 00 00 16 .....
ff0: f3 a0 40 b4 00 00 00 1a 00 00 00 0a 00 00 00 00 .....

```

In a readable format:

```

xfs_db> type dir2
xfs_db> p
bhdr.magic = 0x58443242
bhdr.bestfree[0].offset = 0x130
bhdr.bestfree[0].length = 0xe78
bhdr.bestfree[1].offset = 0
bhdr.bestfree[1].length = 0
bhdr.bestfree[2].offset = 0
bhdr.bestfree[2].length = 0
bu[0].inumber = 33554560
bu[0].namelen = 1
bu[0].name = "."
bu[0].tag = 0x10
bu[1].inumber = 128
bu[1].namelen = 2
bu[1].name = ".."
bu[1].tag = 0x20
bu[2].inumber = 33554561
bu[2].namelen = 15
bu[2].name = "frame000000.tst"
bu[2].tag = 0x30

```

```
bu[3].inumber = 33554562
bu[3].namelen = 15
bu[3].name = "frame000001.tst"
bu[3].tag = 0x50
...
bu[8].inumber = 33554567
bu[8].namelen = 15
bu[8].name = "frame000006.tst"
bu[8].tag = 0xf0
bu[9].inumber = 33554568
bu[9].namelen = 15
bu[9].name = "frame000007.tst"
bu[9].tag = 0x110
bu[10].freetag = 0xffff
bu[10].length = 0xe78
bu[10].tag = 0x130
bleaf[0].hashval = 0x2e
bleaf[0].address = 0x2
bleaf[1].hashval = 0x172e
bleaf[1].address = 0x4
bleaf[2].hashval = 0x83a040b4
bleaf[2].address = 0xe
...
bleaf[8].hashval = 0xe3a040b4
bleaf[8].address = 0x16
bleaf[9].hashval = 0xf3a040b4
bleaf[9].address = 0x1a
btail.count = 10
btail.stale = 0
```

---

**Note**

For block directories, all `xfs_db` fields are preceded with "b".

---

For a simple lookup example, the hash of `frame000000.tst` is `0xb3a040b4`. Looking up that value, we get an address of `0x6`. Multiply that by 8, it becomes offset `0x30` and the inode at that point is `33554561`.

When we remove an entry from the middle (`frame000004.tst`), we can see how the freespace details are adjusted:

```
bhdr.magic = 0x58443242
bhdr.bestfree[0].offset = 0x130
bhdr.bestfree[0].length = 0xe78
bhdr.bestfree[1].offset = 0xb0
bhdr.bestfree[1].length = 0x20
bhdr.bestfree[2].offset = 0
bhdr.bestfree[2].length = 0
...
bu[5].inumber = 33554564
bu[5].namelen = 15
bu[5].name = "frame000003.tst"
bu[5].tag = 0x90
bu[6].freetag = 0xffff
bu[6].length = 0x20
bu[6].tag = 0xb0
bu[7].inumber = 33554566
```

```

bu[7].namelen = 15
bu[7].name = "frame000005.tst"
bu[7].tag = 0xd0
...
bleaf[7].hashval = 0xd3a040b4
bleaf[7].address = 0x22
bleaf[8].hashval = 0xe3a040b4
bleaf[8].address = 0
bleaf[9].hashval = 0xf3a040b4
bleaf[9].address = 0x1a
btail.count = 10
btail.stale = 1

```

A new “bestfree” value is added for the entry, the start of the entry is marked as unused with 0xffff (which overwrites the inode number for an actual entry), and the length of the space. The tag remains intact at the `offset+length-sizeof(tag)`. The address for the hash is also cleared. The affected areas are highlighted below:

```

090: 00 00 00 00 02 00 00 84 0f 66 72 61 6d 65 30 30 .....frame00
0a0: 30 30 30 33 2e 74 73 74 00 00 00 00 00 00 90 0003.tst.....
0b0: ff ff 00 20 02 00 00 85 0f 66 72 61 6d 65 30 30 .....frame00
0c0: 30 30 30 34 2e 74 73 74 00 00 00 00 00 00 00 b0 0004.tst.....
0d0: 00 00 00 00 02 00 00 86 0f 66 72 61 6d 65 30 30 .....frame00
0e0: 30 30 30 35 2e 74 73 74 00 00 00 00 00 00 0d 0005.tst.....
...
fb0: 00 00 17 2e 00 00 00 04 83 a0 40 b4 00 00 00 0e .....
fc0: 93 a0 40 b4 00 00 00 12 a3 a0 40 b4 00 00 00 06 .....
fd0: b3 a0 40 b4 00 00 00 0a c3 a0 40 b4 00 00 00 1e .....
fe0: d3 a0 40 b4 00 00 00 22 e3 a0 40 b4 00 00 00 00 .....
ff0: f3 a0 40 b4 00 00 00 1a 00 00 00 0a 00 00 00 01 .....

```

## Leaf Directories

Once a Block Directory has filled the block, the directory data is changed into a new format. It still uses [extents](#) Chapter 14 and the same basic structures, but the “data” and “leaf” are split up into their own extents. The “leaf” information only occupies one extent. As “leaf” information is more compact than “data” information, more than one “data” extent is common.

- Block to Leaf conversions retain the existing block for the data entries and allocate a new block for the leaf and freespace index information.
- As with all directories, data blocks must start at logical offset zero.
- The “leaf” block has a special offset defined by `XFS_DIR2_LEAF_OFFSET`. Currently, this is 32GB and in the extent view, a block offset of `32GB / sb_blocksize`. On a 4KB block filesystem, this is `0x800000` (8388608 decimal).
- Blocks with directory entries (“data” extents) have the magic number “X2D2” (`0x58443244`), or “XDD3” (`0x58444433`) on a v5 filesystem.
- The “data” extents have a new header (no “leaf” data):

```

typedef struct xfs_dir2_data {
    xfs_dir2_data_hdr_t    hdr;
    xfs_dir2_data_union_t  u[1];
} xfs_dir2_data_t;

```

**hdr**

Data block header. On a v5 filesystem, this field is struct `xfst_dir3_data_hdr`.

**u**

Union of directory and unused entries, exactly the same as in a block directory.

- The “leaf” extent uses the following structures:

```
typedef struct xfst_dir2_leaf {
    xfst_dir2_leaf_hdr_t    hdr;
    xfst_dir2_leaf_entry_t  ents[1];
    xfst_dir2_data_off_t    bests[1];
    xfst_dir2_leaf_tail_t   tail;
} xfst_dir2_leaf_t;
```

**hdr**

Directory leaf header. On a v5 filesystem this is struct `xfst_dir3_leaf_hdr_t`.

**ents**

Hash values of the entries in this block.

**bests**

An array pointing to free regions in the directory block.

**tail**

Bookkeeping for the leaf entries.

```
typedef struct xfst_dir2_leaf_hdr {
    xfst_da_blkinfo_t    info;
    __uint16_t           count;
    __uint16_t           stale;
} xfst_dir2_leaf_hdr_t;
```

**info**

Leaf btree block header.

**count**

Number of leaf entries.

**stale**

Number of stale/zeroed leaf entries.

```
struct xfst_dir3_leaf_hdr {
    struct xfst_da3_blkinfo  info;
    __uint16_t              count;
    __uint16_t              stale;
    __be32                  pad;
};
```

**info**

Leaf B+tree block header.

**count**

Number of leaf entries.

**stale**

Number of stale/zeroed leaf entries.

**pad**

Padding to maintain alignment rules.

```
typedef struct xfs_dir2_leaf_tail {
    __uint32_t      bestcount;
} xfs_dir2_leaf_tail_t;
```

**bestcount**

Number of best free entries.

## Directory and Attribute Block Headers

- Leaf nodes in directories and [extended attributes](#) Chapter 16 use the `xfs_da_blkinfo_t` filesystem block header. The structure appears as follows:

```
typedef struct xfs_da_blkinfo {
    __be32          forw;
    __be32          back;
    __be16          magic;
    __be16          pad;
} xfs_da_blkinfo_t;
```

**forw**

Logical block offset of the previous B+tree block at this level.

**back**

Logical block offset of the next B+tree block at this level.

**magic**

Magic number for this directory/attribute block.

**pad**

Padding to maintain alignment.

- On a v5 filesystem, the leaves use the `struct xfs_da3_blkinfo_t` filesystem block header. This header is used in the same place as `xfs_da_blkinfo_t`:

```
struct xfs_da3_blkinfo {
    /* these values are inside xfs_da_blkinfo */
    __be32          forw;
    __be32          back;
    __be16          magic;
    __be16          pad;

    __be32          crc;
    __be64          blkno;
};
```

```
    __be64          lsn;  
    uuid_t          uuid;  
    __be64          owner;  
};
```

**forw**

Logical block offset of the previous B+tree block at this level.

**back**

Logical block offset of the next B+tree block at this level.

**magic**

Magic number for this directory/attribute block.

**pad**

Padding to maintain alignment.

**crc**

Checksum of the directory/attribute block.

**blkno**

Block number of this directory/attribute block.

**lsn**

Log sequence number of the last write to this block.

**uuid**

The UUID of this block, which must match either `sb_uuid` or `sb_meta_uuid` depending on which features are set.

**owner**

The inode number that this directory/attribute block belongs to.

- The magic number of the leaf block is `XFS_DIR2_LEAF1_MAGIC` (0xd2f1); on a v5 filesystem it is `XFS_DIR3_LEAF1_MAGIC` (0x3df1).
- The size of the `ents` array is specified by `hdr.count`.
- The size of the `bests` array is specified by the `tail.bestcount`, which is also the number of “data” blocks for the directory. The `bests` array maintains each data block’s `bestfree[0].length` value.

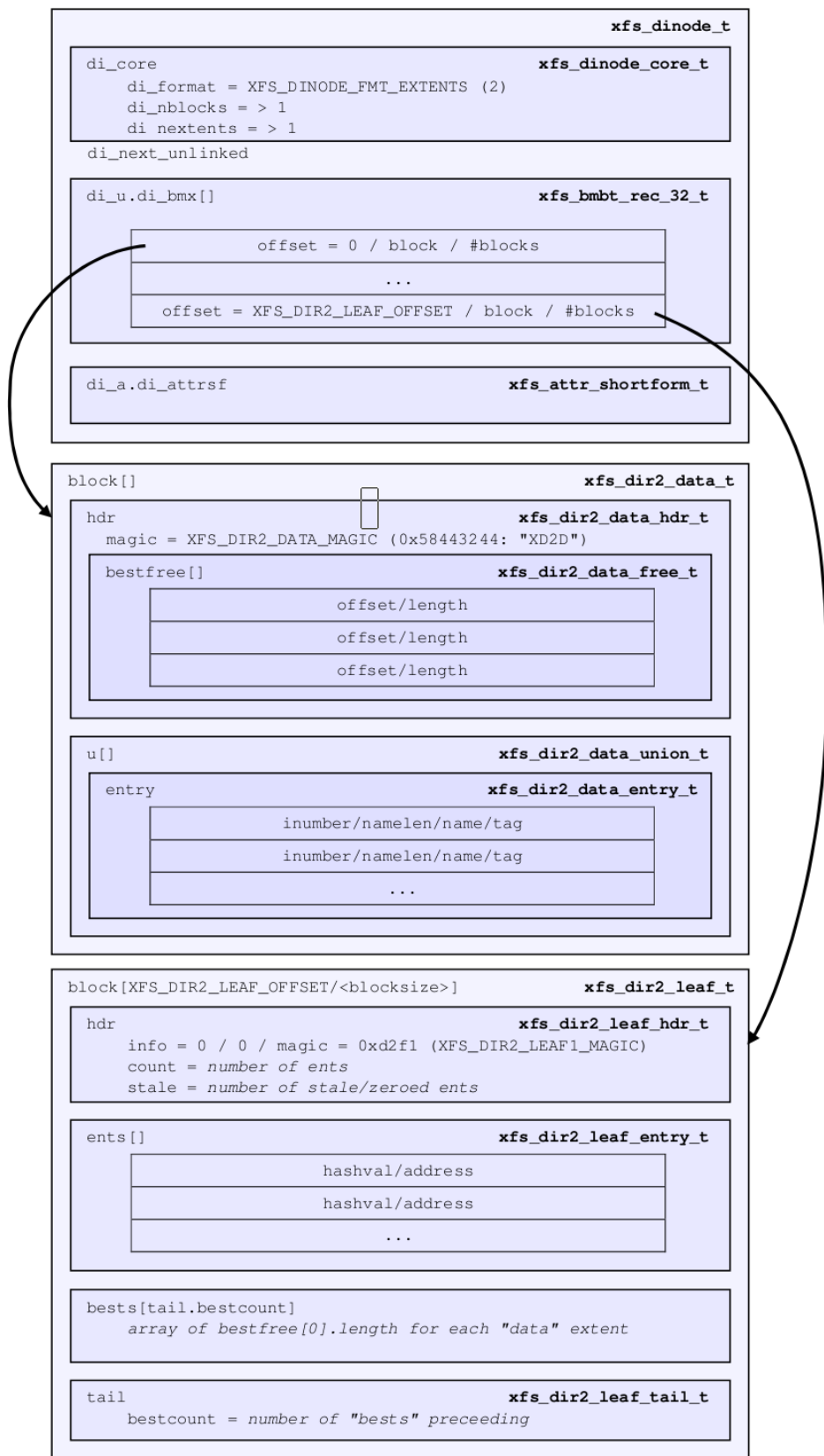


Figure 15.3: Leaf directory free entry detail



## xfs\_db Leaf Directory Example

For this example, a directory was created with 256 entries (frame000000.tst to frame000255.tst). Some files were deleted (frame00005\*, frame00018\* and frame000240.tst) to show free list characteristics.

```
xfs_db> inode <inode#>
xfs_db> p
core.magic = 0x494e
core.mode = 040755
core.version = 1
core.format = 2 (extents)
core.nlinkv1 = 2
...
core.size = 12288
core.nblocks = 4
core.extsize = 0
core.nextents = 3
...
u.bmx[0-2] = [startoff,startblock,blockcount,extentflag]
           0:[0,4718604,1,0]
           1:[1,4718610,2,0]
           2:[8388608,4718605,1,0]
```

As can be seen in this example, three blocks are used for “data” in two extents, and the “leaf” extent has a logical offset of 8388608 blocks (32GB).

Examining the first block:

```
xfs_db> dblock 0
xfs_db> type dir2
xfs_db> p
dhdr.magic = 0x58443244
dhdr.bestfree[0].offset = 0x670
dhdr.bestfree[0].length = 0x140
dhdr.bestfree[1].offset = 0xff0
dhdr.bestfree[1].length = 0x10
dhdr.bestfree[2].offset = 0
dhdr.bestfree[2].length = 0
du[0].inumber = 75497600
du[0].namelen = 1
du[0].name = "."
du[0].tag = 0x10
du[1].inumber = 128
du[1].namelen = 2
du[1].name = ".."
du[1].tag = 0x20
du[2].inumber = 75497601
du[2].namelen = 15
du[2].name = "frame000000.tst"
du[2].tag = 0x30
du[3].inumber = 75497602
du[3].namelen = 15
du[3].name = "frame000001.tst"
du[3].tag = 0x50
...
du[51].inumber = 75497650
du[51].namelen = 15
```

```
du[51].name = "frame000049.tst"
du[51].tag = 0x650
du[52].freetag = 0xffff
du[52].length = 0x140
du[52].tag = 0x670
du[53].inumber = 75497661
du[53].namelen = 15
du[53].name = "frame000060.tst"
du[53].tag = 0x7b0
...
du[118].inumber = 75497758
du[118].namelen = 15
du[118].name = "frame000125.tst"
du[118].tag = 0xfd0
du[119].freetag = 0xffff
du[119].length = 0x10
du[119].tag = 0xff0
```

---

**Note**

The xfs\_db field output is preceded by a “d” for “data”.

---

The next “data” block:

```
xfs_db> dblock 1
xfs_db> type dir2
xfs_db> p
dhdr.magic = 0x58443244
dhdr.bestfree[0].offset = 0x6d0
dhdr.bestfree[0].length = 0x140
dhdr.bestfree[1].offset = 0xe50
dhdr.bestfree[1].length = 0x20
dhdr.bestfree[2].offset = 0xff0
dhdr.bestfree[2].length = 0x10
du[0].inumber = 75497759
du[0].namelen = 15
du[0].name = "frame000126.tst"
du[0].tag = 0x10
...
du[53].inumber = 75497844
du[53].namelen = 15
du[53].name = "frame000179.tst"
du[53].tag = 0x6b0
du[54].freetag = 0xffff
du[54].length = 0x140
du[54].tag = 0x6d0
du[55].inumber = 75497855
du[55].namelen = 15
du[55].name = "frame000190.tst"
du[55].tag = 0x810
...
du[104].inumber = 75497904
du[104].namelen = 15
du[104].name = "frame000239.tst"
du[104].tag = 0xe30
```

```
du[105].freetag = 0xffff
du[105].length = 0x20
du[105].tag = 0xe50
du[106].inumber = 75497906
du[106].namelen = 15
du[106].name = "frame000241.tst"
du[106].tag = 0xe70
...
du[117].inumber = 75497917
du[117].namelen = 15
du[117].name = "frame000252.tst"
du[117].tag = 0xfd0
du[118].freetag = 0xffff
du[118].length = 0x10
du[118].tag = 0xff0
```

And the last data block:

```
xfstool> dblock 2
xfstool> type dir2
xfstool> p
dhdr.magic = 0x58443244
dhdr.bestfree[0].offset = 0x70
dhdr.bestfree[0].length = 0xf90
dhdr.bestfree[1].offset = 0
dhdr.bestfree[1].length = 0
dhdr.bestfree[2].offset = 0
dhdr.bestfree[2].length = 0
du[0].inumber = 75497918
du[0].namelen = 15
du[0].name = "frame000253.tst"
du[0].tag = 0x10
du[1].inumber = 75497919
du[1].namelen = 15
du[1].name = "frame000254.tst"
du[1].tag = 0x30
du[2].inumber = 75497920
du[2].namelen = 15
du[2].name = "frame000255.tst"
du[2].tag = 0x50
du[3].freetag = 0xffff
du[3].length = 0xf90
du[3].tag = 0x70
```

Examining the “leaf” block (with the fields preceded by an “l” for “leaf”):

```
xfstool> dblock 8388608
xfstool> type dir2
xfstool> p
lhdr.info.forw = 0
lhdr.info.back = 0
lhdr.info.magic = 0xd2f1
lhdr.count = 258
lhdr.stale = 0
lbests[0-2] = 0:0x10 1:0x10 2:0xf90
lents[0].hashval = 0x2e
lents[0].address = 0x2
```

```

lents[1].hashval = 0x172e
lents[1].address = 0x4
lents[2].hashval = 0x23a04084
lents[2].address = 0x116
...
lents[257].hashval = 0xf3a048bc
lents[257].address = 0x366
ltail.bestcount = 3

```

Note how the `lbests` array correspond with the `bestfree[0].length` values in the “data” blocks:

```

xfs_db> dblock 0
xfs_db> type dir2
xfs_db> p
dhdr.magic = 0x58443244
dhdr.bestfree[0].offset = 0xff0
dhdr.bestfree[0].length = 0x10
...
xfs_db> dblock 1
xfs_db> type dir2
xfs_db> p
dhdr.magic = 0x58443244
dhdr.bestfree[0].offset = 0xff0
dhdr.bestfree[0].length = 0x10
...
xfs_db> dblock 2
xfs_db> type dir2
xfs_db> p
dhdr.magic = 0x58443244
dhdr.bestfree[0].offset = 0x70
dhdr.bestfree[0].length = 0xf90

```

Now after the entries have been deleted:

```

xfs_db> dblock 8388608
xfs_db> type dir2
xfs_db> p
lhdr.info.forw = 0
lhdr.info.back = 0
lhdr.info.magic = 0xd2f1
lhdr.count = 258
lhdr.stale = 21
lbests[0-2] = 0:0x140 1:0x140 2:0xf90
lents[0].hashval = 0x2e
lents[0].address = 0x2
lents[1].hashval = 0x172e
lents[1].address = 0x4
lents[2].hashval = 0x23a04084
lents[2].address = 0x116
...

```

As can be seen, the `lbests` values have been update to contain each `hdr.bestfree[0].length` values. The leaf’s `hdr.stale` value has also been updated to specify the number of stale entries in the array. The stale entries have an address of zero.

TODO: Need an example for where new entries get inserted with several large free spaces.

## Node Directories

When the “leaf” information fills a block, the extents undergo another separation. All “freeindex” information moves into its own extent. Like Leaf Directories, the “leaf” block maintained the best free space information for each “data” block. This is not possible with more than one leaf.

- The “data” blocks stay the same as leaf directories.
- After the “freeindex” data moves to its own block, it is possible for the leaf data to fit within a single leaf block. This single leaf block has a magic number of XFS\_DIR2\_LEAFN\_MAGIC (0xd2ff) or on a v5 filesystem, XFS\_DIR3\_LEAFN\_MAGIC (0x3dff).
- The “leaf” blocks eventually change into a B+tree with the generic B+tree header pointing to directory “leaves” as described in [Leaf Directories](#) Section 15.3. Blocks with leaf data still have the LEAFN\_MAGIC magic number as outlined above. The top-level tree blocks are called “nodes” and have a magic number of XFS\_DA\_NODE\_MAGIC (0xfebe), or on a v5 filesystem, XFS\_DA3\_NODE\_MAGIC (0x3ebe).
- Distinguishing between a combined leaf/freeindex block (LEAF1\_MAGIC), a leaf-only block (LEAFN\_MAGIC), and a btree node block (NODE\_MAGIC) can only be done by examining the magic number.
- The new “freeindex” block(s) only contains the bests for each data block.
- The freeindex block uses the following structures:

```
typedef struct xfs_dir2_free_hdr {
    __uint32_t      magic;
    __int32_t       firstdb;
    __int32_t       nvalid;
    __int32_t       nused;
} xfs_dir2_free_hdr_t;
```

### magic

The magic number of the free block, “XD2F” (0x0x58443246).

### firstdb

The starting directory block number for the bests array.

### nvalid

Number of elements in the bests array.

### nused

Number of valid elements in the bests array.

```
typedef struct xfs_dir2_free {
    xfs_dir2_free_hdr_t      hdr;
    xfs_dir2_data_off_t      bests[1];
} xfs_dir2_free_t;
```

### hdr

Free block header.

### bests

An array specifying the best free counts in each directory data block.

- On a v5 filesystem, the freeindex block uses the following structures:

```
struct xfs_dir3_free_hdr {
    struct xfs_dir3_blk_hdr  hdr;
    __int32_t                firstdb;
    __int32_t                nvalid;
    __int32_t                nused;
    __int32_t                pad;
};
```

**hdr**

v3 directory block header. The magic number is "XDF3" (0x0x58444633).

**firstdb**

The starting directory block number for the bests array.

**nvalid**

Number of elements in the bests array.

**nused**

Number of valid elements in the bests array.

**pad**

Padding to maintain alignment.

```
struct xfs_dir3_free {
    xfs_dir3_free_hdr_t      hdr;
    __be16                  bests[1];
};
```

**hdr**

Free block header.

**bests**

An array specifying the best free counts in each directory data block.

- The location of the leaf blocks can be in any order, the only way to determine the appropriate is by the node block hash/before values. Given a hash to look up, you read the node's `btree` array and first `hashval` in the array that exceeds the given hash and it can then be found in the block pointed to by the `before` value.

## Directory and Attribute Internal Nodes

The hashing B+tree of a directory or an extended attribute fork uses nodes with the following format:

```
typedef struct xfs_da_intnode {
    struct xfs_da_node_hdr {
        xfs_da_blkinfo_t    info;
        __uint16_t          count;
        __uint16_t          level;
    } hdr;
    struct xfs_da_node_entry {
        xfs_dahash_t        hashval;
        xfs_dablk_t         before;
    } btree[1];
} xfs_da_intnode_t;
```

**info**

Directory/attribute block info. The magic number is XFS\_DA\_NODE\_MAGIC (0xfebe).

**count**

Number of node entries in this block.

**level**

The level of this block in the B+tree.

**hashval**

The hash value of a particular record.

**before**

The directory/attribute logical block containing all entries up to the corresponding hash value.

- On a v5 filesystem, the directory/attribute node blocks have the following structure:

```
struct xfs_da3_intnode {
    struct xfs_da3_node_hdr {
        struct xfs_da3_blkinfo    info;
        __uint16_t                count;
        __uint16_t                level;
        __uint32_t                pad32;
    } hdr;
    struct xfs_da_node_entry {
        xfs_dahash_t              hashval;
        xfs_dablk_t               before;
    } btree[1];
};
```

**info**

Directory/attribute block info. The magic number is XFS\_DA3\_NODE\_MAGIC (0x3ebe).

**count**

Number of node entries in this block.

**level**

The level of this block in the B+tree.

**pad32**

Padding to maintain alignment.

**hashval**

The hash value of a particular record.

**before**

The directory/attribute logical block containing all entries up to the corresponding hash value.

- The freeindex's `bests` array starts from the end of the block and grows to the start of the block.
- When an data block becomes unused (ie. all entries in it have been deleted), the block is freed, the data extents contain a hole, and the freeindex's `hdr.nused` value is decremented and the associated `bests[]` entry is set to `0xffff`.
- As the first data block always contains "." and "..", it's invalid for the directory to have a hole at the start.

- The freeindex's `hdr.nvalid` should always be the same as the number of allocated data directory blocks containing name/inode data and will always be less than or equal to `hdr.nused`. The value of `hdr.nused` should be the same as the index of the last data directory block plus one (i.e. when the last data block is freed, `nused` and `nvalid` are decremented).
-



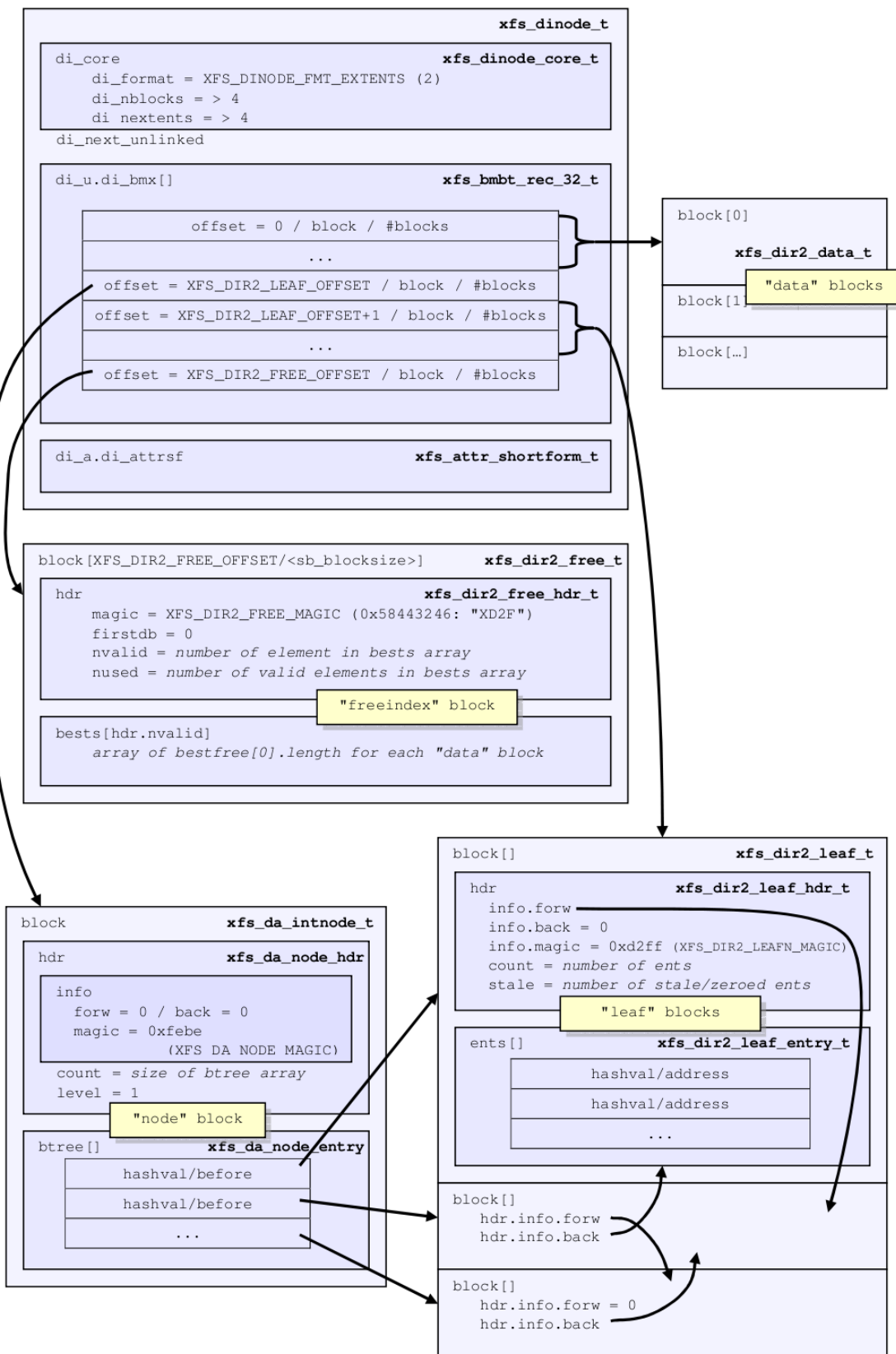


Figure 15.4: Node directory layout

## xfs\_db Node Directory Example

With the node directory examples, we are using a filesystems with 4KB block size, and a 16KB directory size. The directory has over 2000 entries:

```
xfs_db> sb 0
xfs_db> p
magicnum = 0x58465342
blocksize = 4096
...
dirblklog = 2
...
xfs_db> inode <inode#>
xfs_db> p
core.magic = 0x494e
core.mode = 040755
core.version = 1
core.format = 2 (extents)
...
core.size = 81920
core.nblocks = 36
core.extsize = 0
core.nextents = 8
...
u.bmx[0-7] = [startoff,startblock,blockcount,extentflag] 0:[0,7368,4,0]
1:[4,7408,4,0] 2:[8,7444,4,0] 3:[12,7480,4,0] 4:[16,7520,4,0]
5:[8388608,7396,4,0] 6:[8388612,7524,8,0] 7:[16777216,7516,4,0]
```

As can already be observed, all extents are allocated is multiples of 4 blocks.

Blocks 0 to 19 (16+4-1) are used for directory data blocks. Looking at blocks 16-19, we can seen that it's the same as the single-leaf format, except the length values are a lot larger to accommodate the increased directory block size:

```
xfs_db> dblock 16
xfs_db> type dir2
xfs_db> p
dhdr.magic = 0x58443244
dhdr.bestfree[0].offset = 0xb0
dhdr.bestfree[0].length = 0x3f50
dhdr.bestfree[1].offset = 0
dhdr.bestfree[1].length = 0
dhdr.bestfree[2].offset = 0
dhdr.bestfree[2].length = 0
du[0].inumber = 120224
du[0].namelen = 15
du[0].name = "frame002043.tst"
du[0].tag = 0x10
du[1].inumber = 120225
du[1].namelen = 15
du[1].name = "frame002044.tst"
du[1].tag = 0x30
du[2].inumber = 120226
du[2].namelen = 15
du[2].name = "frame002045.tst"
du[2].tag = 0x50
du[3].inumber = 120227
du[3].namelen = 15
```

```

du[3].name = "frame002046.tst"
du[3].tag = 0x70
du[4].inumber = 120228
du[4].namelen = 15
du[4].name = "frame002047.tst"
du[4].tag = 0x90
du[5].freetag = 0xffff
du[5].length = 0x3f50
du[5].tag = 0

```

Next, the “node” block, the fields are preceded with *n* for node blocks:

```

xfs_db> dblock 8388608
xfs_db> type dir2
xfs_db> p
nhdr.info.forw = 0
nhdr.info.back = 0
nhdr.info.magic = 0xfebe
nhdr.count = 2
nhdr.level = 1
nbtrees[0-1] = [hashval,before] 0:[0xa3a440ac,8388616] 1:[0xf3a440bc,8388612]

```

The two following leaf blocks were allocated as part of the directory’s conversion to node format. All hashes less than 0xa3a440ac are located at directory offset 8,388,616, and hashes less than 0xf3a440bc are located at directory offset 8,388,612. Hashes greater or equal to 0xf3a440bc don’t exist in this directory.

```

xfs_db> dblock 8388616
xfs_db> type dir2
xfs_db> p
lhdr.info.forw = 8388612
lhdr.info.back = 0
lhdr.info.magic = 0xd2ff
lhdr.count = 1023
lhdr.stale = 0
lents[0].hashval = 0x2e
lents[0].address = 0x2
lents[1].hashval = 0x172e
lents[1].address = 0x4
lents[2].hashval = 0x23a04084
lents[2].address = 0x116
...
lents[1021].hashval = 0xa3a440a4
lents[1021].address = 0x1fa2
lents[1022].hashval = 0xa3a440ac
lents[1022].address = 0x1fca
xfs_db> dblock 8388612
xfs_db> type dir2
xfs_db> p
lhdr.info.forw = 0
lhdr.info.back = 8388616
lhdr.info.magic = 0xd2ff
lhdr.count = 1027
lhdr.stale = 0
lents[0].hashval = 0xa3a440b4
lents[0].address = 0x1f52
lents[1].hashval = 0xa3a440bc
lents[1].address = 0x1f7a

```

```
...
lents[1025].hashval = 0xf3a440b4
lents[1025].address = 0x1f66
lents[1026].hashval = 0xf3a440bc
lents[1026].address = 0x1f8e
```

An example lookup using `xfs_db`:

```
xfs_db> hash frame001845.tst
0xf3a26094
```

Doing a binary search through the array, we get address `0x1ce6`, which is offset `0xe730`. Each fsblock is 4KB in size (`0x1000`), so it will be offset `0x730` into directory offset 14. From the extent map, this will be fsblock 7482:

```
xfs_db> fsblock 7482
xfs_db> type text
xfs_db> p
...
730: 00 00 00 00 00 01 d4 da 0f 66 72 61 6d 65 30 30 .....frame00
740: 31 38 34 35 2e 74 73 74 00 00 00 00 00 00 27 30 1845.tst.....0
```

Looking at the freeindex information (fields with an *f* tag):

```
xfs_db> fsblock 7516
xfs_db> type dir2
xfs_db> p
fhdr.magic = 0x58443246
fhdr.firstdb = 0
fhdr.nvalid = 5
fhdr.nused = 5
fbests[0-4] = 0:0x10 1:0x10 2:0x10 3:0x10 4:0x3f50
```

Like the Leaf Directory, each of the `fbests` values correspond to each data block's `bestfree[0].length` value.

The `fbests` array is highlighted in a raw block dump:

```
xfs_db> type text
xfs_db> p
000: 58 44 32 46 00 00 00 00 00 00 00 05 00 00 00 05 XD2F.....
010: 00 10 00 10 00 10 00 10 3f 50 00 00 1f 01 ff ff .....P.....
```

TODO: Example with a hole in the middle

## B+tree Directories

When the extent map in an inode grows beyond the inode's space, the inode format is changed to a "btree". The inode contains a filesystem block point to the B+tree extent map for the directory's blocks. The B+tree extents contain the extent map for the "data", "node", "leaf", and "freeindex" information as described in Node Directories.

Refer to the previous section on B+tree [Data Extents](#) Section 14.2 for more information on XFS B+tree extents.

The following properties apply to both node and B+tree directories:

- The node/leaf trees can be more than one level deep.
- More than one freeindex block may exist, but this will be quite rare. It would required hundreds of thousand files with quite long file names (or millions with shorter names) to get a second freeindex block.

## xfs\_db B+tree Directory Example

A directory has been created with 200,000 entries with each entry being 100 characters long. The filesystem block size and directory block size are 4KB:

```
xfs_db> inode <inode#>
xfs_db> p
core.magic = 0x494e
core.mode = 040755
core.version = 1
core.format = 3 (btree)
...
core.size = 22757376
core.nblocks = 6145
core.extsize = 0
core.nextents = 234
core.naextents = 0
core.forkoff = 0
...
u.bmbt.level = 1
u.bmbt.numrecs = 1
u.bmbt.keys[1] = [startoff] 1:[0]
u.bmbt.ptrs[1] = 1:89
xfs_db> fsblock 89
xfs_db> type bmapbtd
xfs_db> p
magic = 0x424d4150
level = 0
numrecs = 234
leftsib = null
rightsib = null
recs[1-234] = [startoff,startblock,blockcount,extentflag]
  1:[0,53,1,0] 2:[1,55,13,0] 3:[14,69,1,0] 4:[15,72,13,0]
  5:[28,86,2,0] 6:[30,90,21,0] 7:[51,112,1,0] 8:[52,114,11,0]
  ...
 125:[5177,902,15,0] 126:[5192,918,6,0] 127:[5198,524786,358,0]
 128:[8388608,54,1,0] 129:[8388609,70,2,0] 130:[8388611,85,1,0]
  ...
 229:[8389164,917,1,0] 230:[8389165,924,19,0] 231:[8389184,944,9,0]
 232:[16777216,68,1,0] 233:[16777217,7340114,1,0] 234:[16777218,5767362,1,0]
```

We have 128 extents and a total of 5555 blocks being used to store name/inode pairs. With only about 2000 values that can be stored in the freeindex block, 3 blocks have been allocated for this information. The `firstdb` field specifies the starting directory block number for each array:

```
xfs_db> dblock 16777216
xfs_db> type dir2
xfs_db> p
fhdr.magic = 0x58443246
fhdr.firstdb = 0
fhdr.nvalid = 2040
fhdr.nused = 2040
fbests[0-2039] = ...
xfs_db> dblock 16777217
xfs_db> type dir2
xfs_db> p
fhdr.magic = 0x58443246
```

```
fhdr.firstdb = 2040
fhdr.nvalid = 2040
fhdr.nused = 2040
fbests[0-2039] = ...
xfs_db> dblock 16777218
xfs_db> type dir2
xfs_db> p
fhdr.magic = 0x58443246
fhdr.firstdb = 4080
fhdr.nvalid = 1476
fhdr.nused = 1476
fbests[0-1475] = ...
```

Looking at the root node in the node block, it's a pretty deep tree:

```
xfs_db> dblock 8388608
xfs_db> type dir2
xfs_db> p
nhdr.info.forw = 0
nhdr.info.back = 0
nhdr.info.magic = 0xfebe
nhdr.count = 2
nhdr.level = 2
nbtree[0-1] = [hashval,before] 0:[0x6bbf6f39,8389121] 1:[0xfbbf7f79,8389120]
xfs_db> dblock 8389121
xfs_db> type dir2
xfs_db> p
nhdr.info.forw = 8389120
nhdr.info.back = 0
nhdr.info.magic = 0xfebe
nhdr.count = 263
nhdr.level = 1
nbtree[0-262] = ... 262:[0x6bbf6f39,8388928]
xfs_db> dblock 8389120
xfs_db> type dir2
xfs_db> p
nhdr.info.forw = 0
nhdr.info.back = 8389121
nhdr.info.magic = 0xfebe
nhdr.count = 319
nhdr.level = 1
nbtree[0-318] = [hashval,before] 0:[0x70b14711,8388919] ...
```

The leaves at each the end of a node always point to the end leaves in adjacent nodes. Directory block 8388928 has a forward pointer to block 8388919 and block 8388919 has a previous pointer to block 8388928, as highlighted in the following example:

```
xfs_db> dblock 8388928
xfs_db> type dir2
xfs_db> p
lhdr.info.forw = 8388919
lhdr.info.back = 8388937
lhdr.info.magic = 0xd2ff
...

xfs_db> dblock 8388919
xfs_db> type dir2
```

```
xfs_db> p
lhdr.info.forw = 8388706
lhdr.info.back = 8388928
lhdr.info.magic = 0xd2ff
...
```

## Chapter 16

# Extended Attributes

Extended attributes enable users and administrators to attach (name: value) pairs to inodes within the XFS filesystem. They could be used to store meta-information about the file.

Attribute names can be up to 256 bytes in length, terminated by the first 0 byte. The intent is that they be printable ASCII (or other character set) names for the attribute. The values can contain up to 64KB of arbitrary binary data. Some XFS internal attributes (eg. parent pointers) use non-printable names for the attribute.

Access Control Lists (ACLs) and Data Migration Facility (DMF) use extended attributes to store their associated metadata with an inode.

XFS uses two disjoint attribute name spaces associated with every inode. These are the root and user address spaces. The root address space is accessible only to the superuser, and then only by specifying a flag argument to the function call. Other users will not see or be able to modify attributes in the root address space. The user address space is protected by the normal file permissions mechanism, so the owner of the file can decide who is able to see and/or modify the value of attributes on any particular file.

To view extended attributes from the command line, use the `getfattr` command. To set or delete extended attributes, use the `setfattr` command. ACLs control should use the `getfacl` and `setfacl` commands.

XFS attributes supports three namespaces: “user”, “trusted” (or “root” using IRIX terminology), and “secure”.

See the section about [extended attributes](#) Section 13.4.1 in the inode for instructions on how to calculate the location of the attributes.

The following four sections describe each of the on-disk formats.

### Short Form Attributes

When the all extended attributes can fit within the inode’s attribute fork, the inode’s `di_aformat` is set to “local” and the attributes are stored in the inode’s literal area starting at offset `di_forkoff × 8`.

Shortform attributes use the following structures:

```
typedef struct xfs_attr_shortform {
    struct xfs_attr_sf_hdr {
        __be16      totsize;
        __u8        count;
    } hdr;
    struct xfs_attr_sf_entry {
```



```

        __uint8_t      namelen;
        __uint8_t      valuelen;
        __uint8_t      flags;
        __uint8_t      nameval[1];
    } list[1];
} xfs_attr_shortform_t;
typedef struct xfs_attr_sf_hdr xfs_attr_sf_hdr_t;
typedef struct xfs_attr_sf_entry xfs_attr_sf_entry_t;

```

**totsize**

Total size of the attribute structure in bytes.

**count**

The number of entries that can be found in this structure.

**namelen and valuelen**

These values specify the size of the two byte arrays containing the name and value pairs. `valuelen` is zero for extended attributes with no value.

**nameval[]**

A single array whose size is the sum of `namelen` and `valuelen`. The names and values are not null terminated on-disk. The value immediately follows the name in the array.

**flags**

A combination of the following:

Table 16.1: Attribute Namespaces

Flag	Description
0	The attribute's namespace is "user".
XFS_ATTR_ROOT	The attribute's namespace is "trusted".
XFS_ATTR_SECURE	The attribute's namespace is "secure".
XFS_ATTR_INCOMPLETE	This attribute is being modified.
XFS_ATTR_LOCAL	The attribute value is contained within this block.

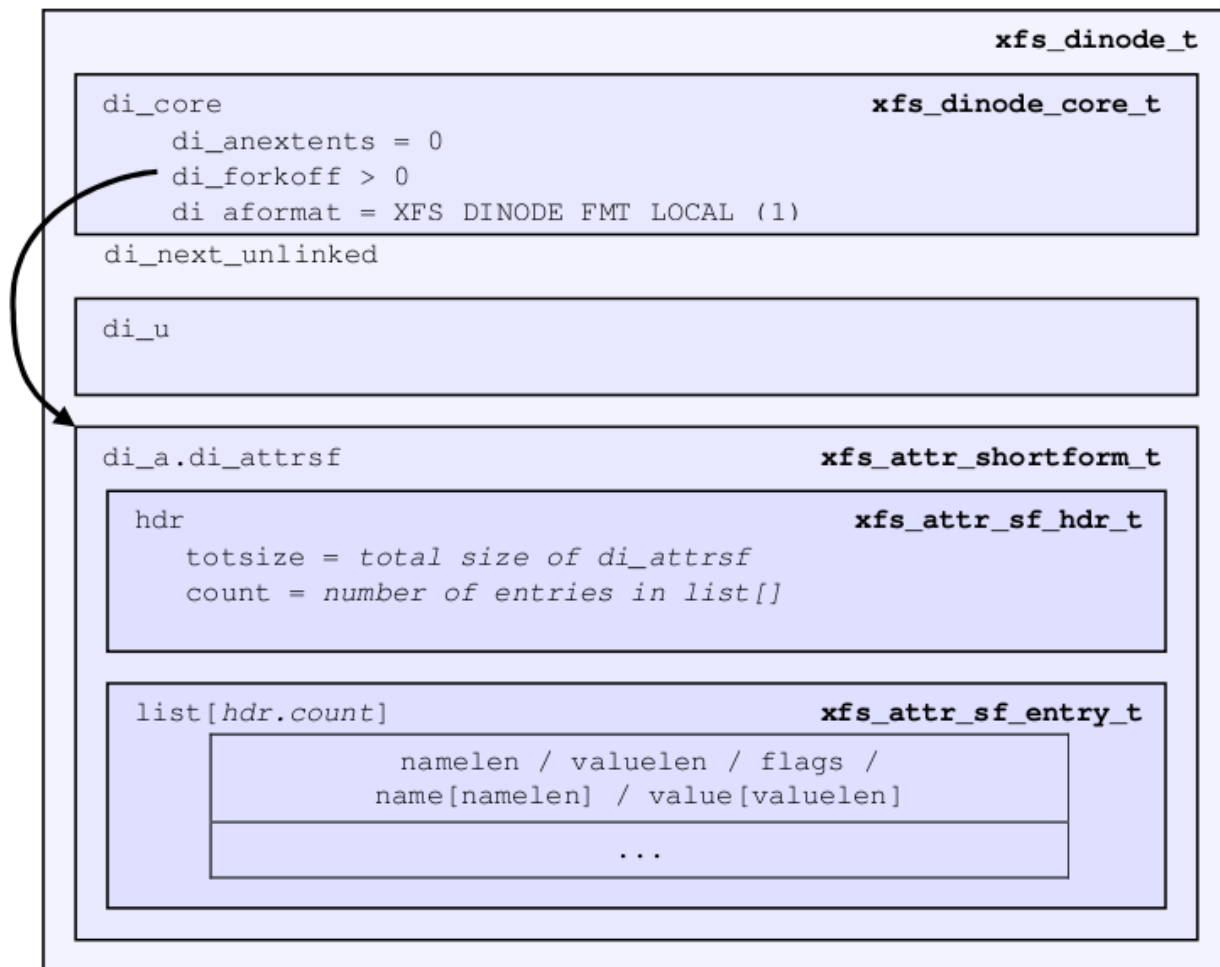


Figure 16.1: Short form attribute layout

### xfs\_db Short Form Attribute Example

A file is created and two attributes are set:

```
# setfattr -n user.empty few_attr
# setfattr -n trusted.trust -v val1 few_attr
```

Using xfs\_db, we dump the inode:

```
xfs_db> inode <inode#>
xfs_db> p
core.magic = 0x494e
core.mode = 0100644
...
core.naextents = 0
core.forkoff = 15
core.aformat = 1 (local)
```

```

...
a.sfattr.hdr.totsize = 24
a.sfattr.hdr.count = 2
a.sfattr.list[0].namelen = 5
a.sfattr.list[0].valuelen = 0
a.sfattr.list[0].root = 0
a.sfattr.list[0].secure = 0
a.sfattr.list[0].name = "empty"
a.sfattr.list[1].namelen = 5
a.sfattr.list[1].valuelen = 4
a.sfattr.list[1].root = 1
a.sfattr.list[1].secure = 0
a.sfattr.list[1].name = "trust"
a.sfattr.list[1].value = "val1"

```

We can determine the actual inode offset to be 220 (15 x 8 + 100) or 0xdc. Examining the raw dump, the second attribute is highlighted:

```

xfs_db> type text
xfs_db> p
09: 49 4e 81 a4 01 02 00 01 00 00 00 00 00 00 00 00 IN.....
10: 00 00 00 01 00 00 00 00 00 00 00 00 00 00 02 .....
20: 44 be 19 be 38 d1 26 98 44 be 1a be 38 d1 26 98 D...8...D...8...
30: 44 be 1a e1 3a 9a ea 18 00 00 00 00 00 00 00 04 D.....
40: 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 01 .....
50: 00 00 0f 01 00 00 00 00 00 00 00 00 00 00 00 .....
60: ff ff ff ff 00 00 00 00 00 00 00 00 00 00 00 12 .....
70: 53 a0 00 01 00 00 00 00 00 00 00 00 00 00 00 .....
80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
90: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..... <-- hdr. ←
    totsize = 0x18
e0: 05 00 00 65 6d 70 74 79 05 04 02 74 72 75 73 74 ...empty...trust
f0: 76 61 6c 31 00 00 00 00 00 00 00 00 00 00 00 val1.....

```

Adding another attribute with attr1, the format is converted to extents and `di_forkoff` remains unchanged (and all those zeros in the dump above remain unused):

```

xfs_db> inode <inode#>
xfs_db> p
...
core.naextents = 1
core.forkoff = 15
core.aformat = 2 (extents)
...
a.bmx[0] = [startoff,startblock,blockcount,extentflag] 0:[0,37534,1,0]

```

Performing the same steps with attr2, adding one attribute at a time, you can see `di_forkoff` change as attributes are added:

```

xfs_db> inode <inode#>
xfs_db> p
...
core.naextents = 0

```

```
core.forkoff = 15
core.aformat = 1 (local)
...
a.sfattr.hdr.totsize = 17
a.sfattr.hdr.count = 1
a.sfattr.list[0].namelen = 10
a.sfattr.list[0].valuelen = 0
a.sfattr.list[0].root = 0
a.sfattr.list[0].secure = 0
a.sfattr.list[0].name = "empty_attr"
```

Attribute added:

```
xfstool> p
...
core.naextents = 0
core.forkoff = 15
core.aformat = 1 (local)
...
a.sfattr.hdr.totsize = 31
a.sfattr.hdr.count = 2
a.sfattr.list[0].namelen = 10
a.sfattr.list[0].valuelen = 0
a.sfattr.list[0].root = 0
a.sfattr.list[0].secure = 0
a.sfattr.list[0].name = "empty_attr"
a.sfattr.list[1].namelen = 7
a.sfattr.list[1].valuelen = 4
a.sfattr.list[1].root = 1
a.sfattr.list[1].secure = 0
a.sfattr.list[1].name = "trust_a"
a.sfattr.list[1].value = "val1"
```

Another attribute is added:

```
xfstool> p
...
core.naextents = 0
core.forkoff = 13
core.aformat = 1 (local)
...
a.sfattr.hdr.totsize = 52
a.sfattr.hdr.count = 3
a.sfattr.list[0].namelen = 10
a.sfattr.list[0].valuelen = 0
a.sfattr.list[0].root = 0
a.sfattr.list[0].secure = 0
a.sfattr.list[0].name = "empty_attr"
a.sfattr.list[1].namelen = 7
a.sfattr.list[1].valuelen = 4
a.sfattr.list[1].root = 1
a.sfattr.list[1].secure = 0
a.sfattr.list[1].name = "trust_a"
a.sfattr.list[1].value = "val1"
a.sfattr.list[2].namelen = 6
a.sfattr.list[2].valuelen = 12
a.sfattr.list[2].root = 0
```

```
a.sfattr.list[2].secure = 0
a.sfattr.list[2].name = "second"
a.sfattr.list[2].value = "second_value"
```

One more is added:

```
xfstool> p
core.naextents = 0
core.forkoff = 10
core.aformat = 1 (local)
...
a.sfattr.hdr.totsize = 69
a.sfattr.hdr.count = 4
a.sfattr.list[0].namelen = 10
a.sfattr.list[0].valuelen = 0
a.sfattr.list[0].root = 0
a.sfattr.list[0].secure = 0
a.sfattr.list[0].name = "empty_attr"
a.sfattr.list[1].namelen = 7
a.sfattr.list[1].valuelen = 4
a.sfattr.list[1].root = 1
a.sfattr.list[1].secure = 0
a.sfattr.list[1].name = "trust_a"
a.sfattr.list[1].value = "val1"
a.sfattr.list[2].namelen = 6
a.sfattr.list[2].valuelen = 12
a.sfattr.list[2].root = 0
a.sfattr.list[2].secure = 0
a.sfattr.list[2].name = "second"
a.sfattr.list[2].value = "second_value"
a.sfattr.list[3].namelen = 6
a.sfattr.list[3].valuelen = 8
a.sfattr.list[3].root = 0
a.sfattr.list[3].secure = 1
a.sfattr.list[3].name = "policy"
a.sfattr.list[3].value = "contents"
```

A raw dump is shown to compare with the attr1 dump on a prior page, the header is highlighted:

```
xfstool> type text
xfstool> p
00: 49 4e 81 a4 01 02 00 01 00 00 00 00 00 00 00 00 IN.....
10: 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00 05 .....
20: 44 be 24 cd 0f b0 96 18 44 be 24 cd 0f b0 96 18 D.....D.....
30: 44 be 2d f5 01 62 7a 18 00 00 00 00 00 00 00 04 D....bz.....
40: 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 01 .....
50: 00 00 0a 01 00 00 00 00 00 00 00 00 00 00 00 .....
60: ff ff ff ff 00 00 00 00 00 00 00 00 00 00 00 01 .....
70: 41 c0 00 01 00 00 00 00 00 00 00 00 00 00 00 A.....
80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
90: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
b0: 00 00 00 00 00 45 04 00 0a 00 00 65 6d 70 74 79 .....E....empty
c0: 5f 61 74 74 72 07 04 02 74 72 75 73 74 5f 61 76 .attr...trust.av
d0: 61 6c 31 06 0c 00 73 65 63 6f 6e 64 73 65 63 6f all...secondseco
e0: 6e 64 5f 76 61 6c 75 65 06 08 04 70 6f 6c 69 63 nd.value...polic
f0: 79 63 6f 6e 74 65 6e 74 73 64 5f 76 61 6c 75 65 ycontentsd.value
```

It can be clearly seen that attr2 allows many more attributes to be stored in an inode before they are moved to another filesystem block.

## Leaf Attributes

When an inode's attribute fork space is used up with shortform attributes and more are added, the attribute format is migrated to "extents".

Extent based attributes use hash/index pairs to speed up an attribute lookup. The first part of the "leaf" contains an array of fixed size hash/index pairs with the flags stored as well. The remaining part of the leaf block contains the array name/value pairs, where each element varies in length.

Each leaf is based on the `xf_s_da_blkinfo_t` block header declared in the section about [directories](#) Section 15.3.1. On a v5 filesystem, the block header is `xf_s_da3_blkinfo_t`. The structure encapsulating all other structures in the attribute block is `xf_s_attr_leafblock_t`.

The structures involved are:

```
typedef struct xf_s_attr_leaf_map {
    __be16          base;
    __be16          size;
} xf_s_attr_leaf_map_t;
```

### base

Block offset of the free area, in bytes.

### size

Size of the free area, in bytes.

```
typedef struct xf_s_attr_leaf_hdr {
    xf_s_da_blkinfo_t  info;
    __be16             count;
    __be16             usedbytes;
    __be16             firstused;
    __u8               holes;
    __u8               pad1;
    xf_s_attr_leaf_map_t  freemap[3];
} xf_s_attr_leaf_hdr_t;
```

### info

Directory/attribute block header.

### count

Number of entries.

### usedbytes

Number of bytes used in the leaf block.

### firstused

Block offset of the first entry in use, in bytes.

### holes

Set to 1 if block compaction is necessary.

---

**pad1**

Padding to maintain alignment to 64-bit boundaries.

```
typedef struct xfs_attr_leaf_entry {
    __be32      hashval;
    __be16      nameidx;
    __u8        flags;
    __u8        pad2;
} xfs_attr_leaf_entry_t;
```

**hashval**

Hash value of the attribute name.

**nameidx**

Block offset of the name entry, in bytes.

**flags**

Attribute flags, as specified [above](#) Table 16.1.

**pad2**

Pads the structure to 64-bit boundaries.

```
typedef struct xfs_attr_leaf_name_local {
    __be16      valuelen;
    __u8        namelen;
    __u8        nameval[1];
} xfs_attr_leaf_name_local_t;
```

**valuelen**

Length of the value, in bytes.

**namelen**

Length of the name, in bytes.

**nameval**

The name and the value. String values are not zero-terminated.

```
typedef struct xfs_attr_leaf_name_remote {
    __be32      valueblk;
    __be32      valuelen;
    __u8        namelen;
    __u8        name[1];
} xfs_attr_leaf_name_remote_t;
```

**valueblk**

The logical block in the attribute map where the value is located.

**valuelen**

Length of the value, in bytes.

**namelen**

Length of the name, in bytes.

**nameval**

The name. String values are not zero-terminated.

```
typedef struct xfs_attr_leafblock {
    xfs_attr_leaf_hdr_t      hdr;
    xfs_attr_leaf_entry_t   entries[1];
    xfs_attr_leaf_name_local_t  namelist;
    xfs_attr_leaf_name_remote_t valuelist;
} xfs_attr_leafblock_t;
```

**hdr**

Attribute block header.

**entries**

A variable-length array of attribute entries.

**namelist**

A variable-length array of descriptors of local attributes. The location and size of these entries is determined dynamically.

**valuelist**

A variable-length array of descriptors of remote attributes. The location and size of these entries is determined dynamically.

On a v5 filesystem, the header becomes `xfs_da3_blkinfo_t` to accommodate the extra metadata integrity fields:

```
typedef struct xfs_attr3_leaf_hdr {
    xfs_da3_blkinfo_t      info;
    __be16                 count;
    __be16                 usedbytes;
    __be16                 firstused;
    __u8                   holes;
    __u8                   pad1;
    xfs_attr_leaf_map_t    freemap[3];
} xfs_attr3_leaf_hdr_t;

typedef struct xfs_attr3_leafblock {
    xfs_attr3_leaf_hdr_t   hdr;
    xfs_attr_leaf_entry_t  entries[1];
    xfs_attr_leaf_name_local_t  namelist;
    xfs_attr_leaf_name_remote_t valuelist;
} xfs_attr3_leafblock_t;
```

Each leaf header uses the magic number `XFS_ATTR_LEAF_MAGIC` (0xfbee). On a v5 filesystem, the magic number is `XFS_ATTR3_LEAF_MAGIC` (0x3bee).

The hash/index elements in the `entries[]` array are packed from the top of the block. Name/values grow from the bottom but are not packed. The freemap contains run-length-encoded entries for the free bytes after the `entries[]` array, but only the three largest runs are stored (smaller runs are dropped). When the `freemap` doesn't show enough space for an allocation, the name/value area is compacted and allocation is tried again. If there still isn't enough space, then the block is split. The name/value structures (both local and remote versions) must be 32-bit aligned.

For attributes with small values (ie. the value can be stored within the leaf), the `XFS_ATTR_LOCAL` flag is set for the attribute. The entry details are stored using the `xfs_attr_leaf_name_local_t` structure. For large attribute



---

values that cannot be stored within the leaf, separate filesystem blocks are allocated to store the value. They use the `xfattr_leaf_name_remote_t` structure. See [Remote Values](#) Section 16.5 for more information.

---

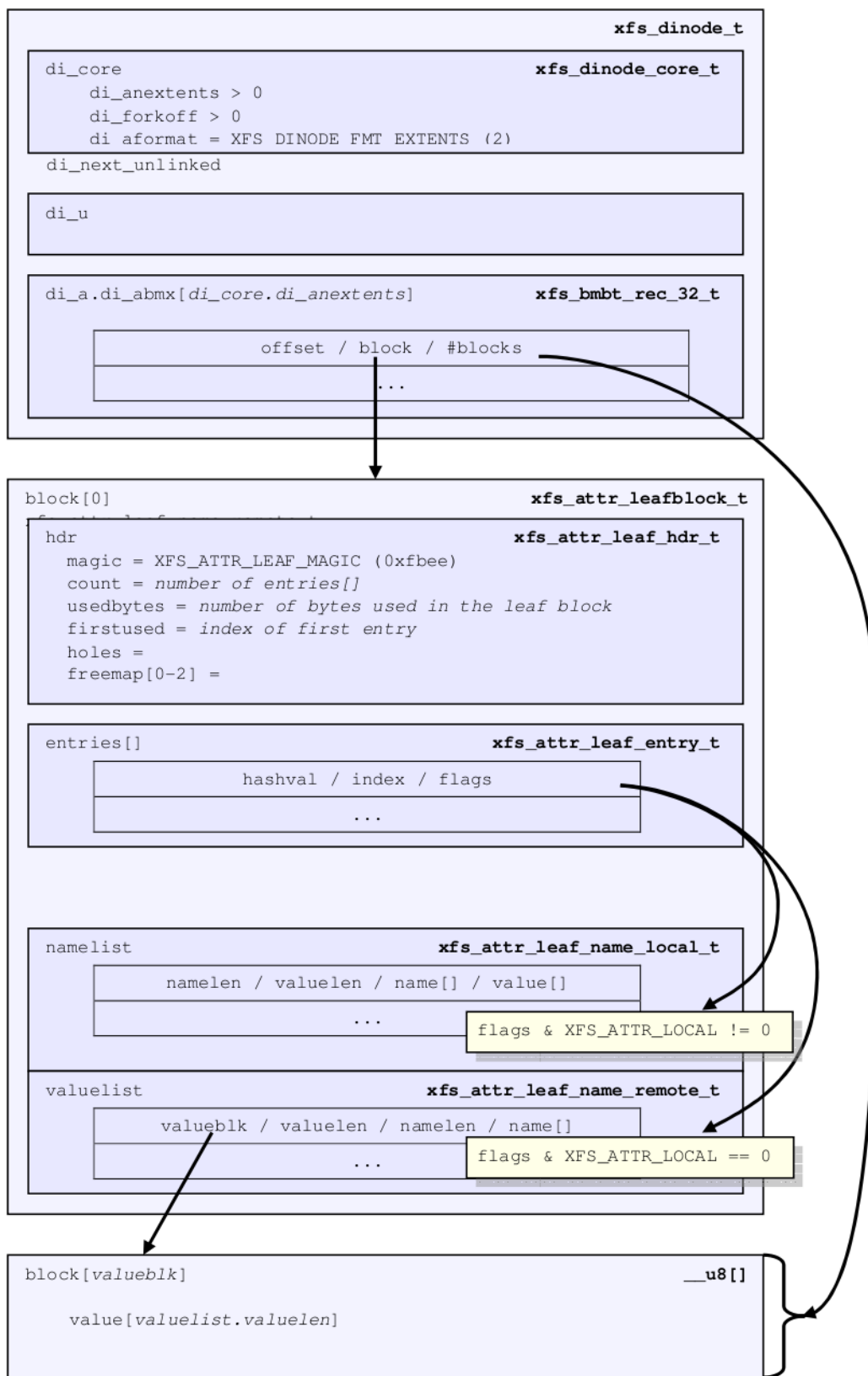


Figure 16.2: Leaf attribute layout

Both local and remote entries can be interleaved as they are only addressed by the hash/index entries. The flag is stored with the hash/index pairs so the appropriate structure can be used.

Since duplicate hash keys are possible, for each hash that matches during a lookup, the actual name string must be compared.

An “incomplete” bit is also used for attribute flags. It shows that an attribute is in the middle of being created and should not be shown to the user if we crash during the time that the bit is set. The bit is cleared when attribute has finished being set up. This is done because some large attributes cannot be created inside a single transaction.

## xfs\_db Leaf Attribute Example

A single 30KB extended attribute is added to an inode:

```
xfs_db> inode <inode#>
xfs_db> p
...
core.nblocks = 9
core.nextents = 0
core.naextents = 1
core.forkoff = 15
core.aformat = 2 (extents)
...
a.bmx[0] = [startoff,startblock,blockcount,extentflag]
          0:[0,37535,9,0]
xfs_db> ablock 0
xfs_db> p
hdr.info.forw = 0
hdr.info.back = 0
hdr.info.magic = 0xfbee
hdr.count = 1
hdr.usedbytes = 20
hdr.firstused = 4076
hdr.holes = 0
hdr.freemap[0-2] = [base,size] 0:[40,4036] 1:[0,0] 2:[0,0]
entries[0] = [hashval,nameidx,incomplete,root,secure,local]
            0:[0xfcfc89d4f,4076,0,0,0,0]
nvlst[0].valueblk = 0x1
nvlst[0].valueLen = 30692
nvlst[0].namelen = 8
nvlst[0].name = "big_attr"
```

Attribute blocks 1 to 8 (filesystem blocks 37536 to 37543) contain the raw binary value data for the attribute.

Index 4076 (0xfec) is the offset into the block where the name/value information is. As can be seen by the value, it's at the end of the block:

```
xfs_db> type text
xfs_db> p

000: 00 00 00 00 00 00 00 00 fb ee 00 00 00 01 00 14 .....
010: 0f ec 00 00 00 28 0f c4 00 00 00 00 00 00 00 .....
020: fc f8 9d 4f 0f ec 00 00 00 00 00 00 00 00 00 ...0.....
030: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
...
fe0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01 .....
ff0: 00 00 77 e4 08 62 69 67 5f 61 74 74 72 00 00 ..w..big.attr...
```

A 30KB attribute and a couple of small attributes are added to a file:

```
xfs_db> inode <inode#>
xfs_db> p
...
core.nblocks = 10
core.extsize = 0
core.nextents = 1
core.naextents = 2
core.forkoff = 15
core.aformat = 2 (extents)
...
u.bmx[0] = [startoff,startblock,blockcount,extentflag]
          0:[0,81857,1,0]
a.bmx[0-1] = [startoff,startblock,blockcount,extentflag]
             0:[0,81858,1,0]
             1:[1,182398,8,0]
xfs_db> ablock 0
xfs_db> p
hdr.info.forw = 0
hdr.info.back = 0
hdr.info.magic = 0xfbee
hdr.count = 3
hdr.usedbytes = 52
hdr.firstused = 4044
hdr.holes = 0
hdr.freemap[0-2] = [base,size] 0:[56,3988] 1:[0,0] 2:[0,0]
entries[0-2] = [hashval,nameidx,incomplete,root,secure,local]
              0:[0x1e9d3934,4044,0,0,0,1]
              1:[0x1e9d3937,4060,0,0,0,1]
              2:[0xfcf89d4f,4076,0,0,0,0]
nvlst[0].valuelen = 6
nvlst[0].namelen = 5
nvlst[0].name = "attr2"
nvlst[0].value = "value2"
nvlst[1].valuelen = 6
nvlst[1].namelen = 5
nvlst[1].name = "attr1"
nvlst[1].value = "value1"
nvlst[2].valueblk = 0x1
nvlst[2].valuelen = 30692
nvlst[2].namelen = 8
nvlst[2].name = "big_attr"
```

As can be seen in the entries array, the two small attributes have the local flag set and the values are printed.

A raw disk dump shows the attributes. The last attribute added is highlighted (offset 4044 or 0xfcc):

```
000: 00 00 00 00 00 00 00 00 00 fb ee 00 00 00 03 00 34 .....4
010: 0f cc 00 00 00 38 0f 94 00 00 00 00 00 00 00 .....8.....
020: 1e 9d 39 34 0f cc 01 00 1e 9d 39 37 0f dc 01 00 ..94.....97....
030: fc f8 9d 4f 0f ec 00 00 00 00 00 00 00 00 00 ...0.....
040: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
...
fc0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 06 05 61 .....a
fd0: 74 74 72 32 76 61 6c 75 65 32 00 00 00 06 05 61 ttr2value2.....a
fe0: 74 74 72 31 76 61 6c 75 65 31 00 00 00 00 01 ttr1value1.....
ff0: 00 00 77 e4 08 62 69 67 5f 61 74 74 72 00 00 00 ..w..big.attr...
```

## Node Attributes

When the number of attributes exceeds the space that can fit in one filesystem block (ie. hash, flag, name and local values), the first attribute block becomes the root of a B+tree where the leaves contain the hash/name/value information that was stored in a single leaf block. The inode's attribute format itself remains extent based. The nodes use the `xfst_da_intnode_t` or `xfst_da3_intnode_t` structures introduced in the section about [directories](#) Section 15.4.1.

The location of the attribute leaf blocks can be in any order. The only way to find an attribute is by walking the node block hash/before values. Given a hash to look up, search the node's btree array for the first `hashval` in the array that exceeds the given hash. The entry is in the block pointed to by the `before` value.

Each attribute node block has a magic number of `XFS_DA_NODE_MAGIC` (0xfebe). On a v5 filesystem this is `XFS_DA3_NODE_MAGIC` (0x3ebe).

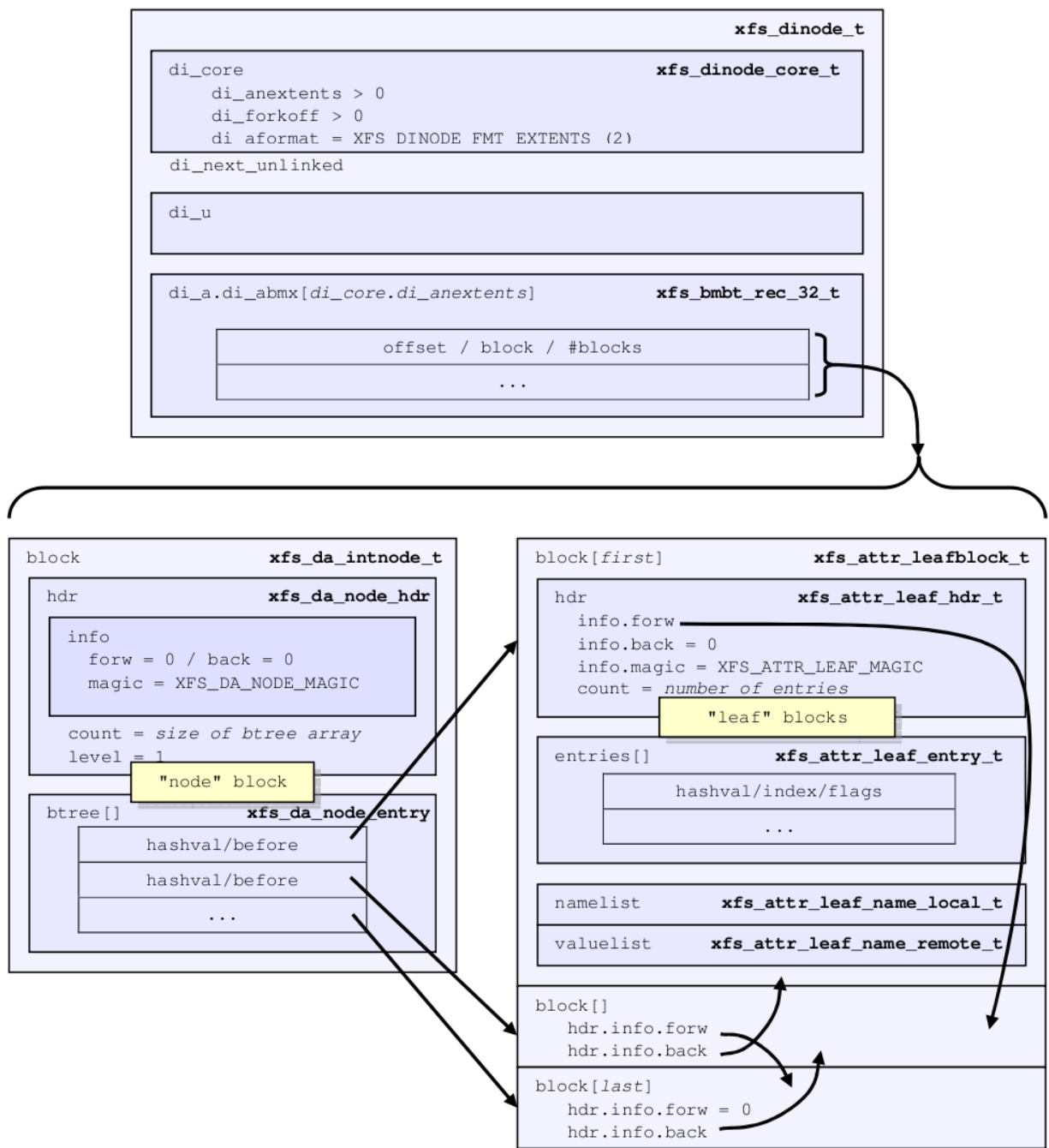


Figure 16.3: Node attribute layout

### xfs\_db Node Attribute Example

An inode with 1000 small attributes with the naming "attribute\_n" where n is a number:

```
xfs_db> inode <inode#>
```

```

xfs_db> p
...
core.nblocks = 15
core.nextents = 0
core.naextents = 1
core.forkoff = 15
core.aformat = 2 (extents)
...
a.bmx[0] = [startoff,startblock,blockcount,extentflag] 0:[0,525144,15,0]
xfs_db> ablock 0
xfs_db> p
hdr.info.forw = 0
hdr.info.back = 0
hdr.info.magic = 0xfebe
hdr.count = 14
hdr.level = 1
btree[0-13] = [hashval,before]
    0:[0x3435122d,1]
    1:[0x343550a9,14]
    2:[0x343553a6,13]
    3:[0x3436122d,12]
    4:[0x343650a9,8]
    5:[0x343653a6,7]
    6:[0x343691af,6]
    7:[0x3436d0ab,11]
    8:[0x3436d3a7,10]
    9:[0x3437122d,9]
   10:[0x3437922e,3]
   11:[0x3437d22a,5]
   12:[0x3e686c25,4]
   13:[0x3e686fad,2]

```

The hashes are in ascending order in the btree array, and if the hash for the attribute we are looking up is before the entry, we go to the addressed attribute block.

For example, to lookup attribute “attribute\_267”:

```

xfs_db> hash attribute_267
0x3437d1a8

```

In the root btree node, this falls between 0x3437922e and 0x3437d22a, therefore leaf 11 or attribute block 5 will contain the entry.

```

xfs_db> ablock 5
xfs_db> p
hdr.info.forw = 4
hdr.info.back = 3
hdr.info.magic = 0xfbee
hdr.count = 96
hdr.usedbytes = 2688
hdr.firstused = 1408
hdr.holes = 0
hdr.freemap[0-2] = [base,size] 0:[800,608] 1:[0,0] 2:[0,0]
entries[0.95] = [hashval,nameidx,incomplete,root,secure,local]
    0:[0x3437922f,4068,0,0,0,1]
    1:[0x343792a6,4040,0,0,0,1]
    2:[0x343792a7,4012,0,0,0,1]

```

```

    3:[0x343792a8,3984,0,0,0,1]
    ...
    82:[0x3437d1a7,2892,0,0,0,1]
    83:[0x3437d1a8,2864,0,0,0,1]
    84:[0x3437d1a9,2836,0,0,0,1]
    ...
    95:[0x3437d22a,2528,0,0,0,1]
nvlist[0].valueLen = 10
nvlist[0].nameLen = 13
nvlist[0].name = "attribute_310"
nvlist[0].value = "value_316\d"
nvlist[1].valueLen = 16
nvlist[1].nameLen = 13
nvlist[1].name = "attribute_309"
nvlist[1].value = "value_309\d"
nvlist[2].valueLen = 10
nvlist[2].nameLen = 13
nvlist[2].name = "attribute_308"
nvlist[2].value = "value_308\d"
nvlist[3].valueLen = 10
nvlist[3].nameLen = 13
nvlist[3].name = "attribute_307"
nvlist[3].value = "value_307\d"
...
nvlist[82].valueLen = 10
nvlist[82].nameLen = 13
nvlist[82].name = "attribute_268"
nvlist[82].value = "value_268\d"
nvlist[83].valueLen = 10
nvlist[83].nameLen = 13
nvlist[83].name = "attribute_267"
nvlist[83].value = "value_267\d"
nvlist[84].valueLen = 10
nvlist[84].nameLen = 13
nvlist[84].name = "attribute_266"
nvlist[84].value = "value_266\d"
...

```

Each of the hash entries has XFS\_ATTR\_LOCAL flag set (1), which means the attribute's value follows immediately after the name. Raw disk of the name/value pair at offset 2864 (0xb30), highlighted with "value\_267" following immediately after the name:

```

b00: 62 75 74 65 5f 32 36 35 76 61 6c 75 65 5f 32 36 bute.265value.26
b10: 35 0a 00 00 00 0a 0d 61 74 74 72 69 62 75 74 65 5.....attribute
b20: 51 32 36 36 76 61 6c 75 65 5f 32 36 36 0a 00 00 .266value.266...
b30: 00 0a 0d 61 74 74 72 69 62 75 74 65 5f 32 36 37 ...attribute.267
b40: 76 61 6c 75 65 5f 32 36 37 0a 00 00 00 0a 0d 61 value.267.....a
b50: 74 74 72 69 62 75 74 65 5f 32 36 38 76 61 6c 75 ttribute.268valu
b60: 65 5f 32 36 38 0a 00 00 00 0a 0d 61 74 74 72 69 e.268.....attri
b70: 62 75 74 65 5f 32 36 39 76 61 6c 75 65 5f 32 36 bute.269value.26

```

Each entry starts on a 32-bit (4 byte) boundary, therefore the highlighted entry has 2 unused bytes after it.



## B+tree Attributes

When the attribute's extent map in an inode grows beyond the available space, the inode's attribute format is changed to a "btree". The inode contains root node of the extent B+tree which then address the leaves that contains the extent arrays for the attribute data. The attribute data itself in the allocated filesystem blocks use the same layout and structures as described in [Node Attributes](#) Section 16.3.

Refer to the previous section on [B+tree Data Extents](#) Section 14.2 for more information on XFS B+tree extents.

### xfs\_db B+tree Attribute Example

Added 2000 attributes with 729 byte values to a file:

```
xfs_db> inode <inode#>
xfs_db> p
...
core.nblocks = 640
core.extsize = 0
core.nextents = 1
core.naextents = 274
core.forkoff = 15
core.aformat = 3 (btree)
...
a.bmbt.level = 1
a.bmbt.numrecs = 2
a.bmbt.keys[1-2] = [startoff] 1:[0] 2:[219]
a.bmbt.pters[1-2] = 1:83162 2:109968
xfs_db> fsblock 83162
xfs_db> type bmapbtd
xfs_db> p
magic = 0x424d4150
level = 0
numrecs = 127
leftsib = null
rightsib = 109968
recs[1-127] = [startoff,startblock,blockcount,extentflag]
                1:[0,81870,1,0]
...
xfs_db> fsblock 109968
xfs_db> type bmapbtd
xfs_db> p
magic = 0x424d4150
level = 0
numrecs = 147
leftsib = 83162
rightsib = null
recs[1-147] = [startoff,startblock,blockcount,extentflag]
                ...
                (which is fsblock 81870)
xfs_db> ablock 0
xfs_db> p
hdr.info.forw = 0
hdr.info.back = 0
hdr.info.magic = 0xfebe
hdr.count = 2
```

```
hdr.level = 2
btree[0-1] = [hashval,before] 0:[0x343612a6,513] 1:[0x3e686fad,512]
```

The extent B+tree has two leaves that specify the 274 extents used for the attributes. Looking at the first block, it can be seen that the attribute B+tree is two levels deep. The two blocks at offset 513 and 512 (ie. access using the `ablock` command) are intermediate `xfs_da_intnode_t` nodes that index all the attribute leaves.

## Remote Attribute Values

On a v5 filesystem, all remote value blocks start with this header:

```
struct xfs_attr3_rmt_hdr {
    __be32  rm_magic;
    __be32  rm_offset;
    __be32  rm_bytes;
    __be32  rm_crc;
    uuid_t  rm_uuid;
    __be64  rm_owner;
    __be64  rm_blkno;
    __be64  rm_lsn;
};
```

### **rm\_magic**

Specifies the magic number for the remote value block: "XARM" (0x5841524d).

### **rm\_offset**

Offset of the remote value data, in bytes.

### **rm\_bytes**

Number of bytes used to contain the remote value data.

### **rm\_crc**

Checksum of the remote value block.

### **rm\_uuid**

The UUID of this block, which must match either `sb_uuid` or `sb_meta_uuid` depending on which features are set.

### **rm\_owner**

The inode number that this remote value block belongs to.

### **rm\_blkno**

Disk block number of this remote value block.

### **rm\_lsn**

Log sequence number of the last write to this block.

Filesystems formatted prior to v5 do not have this header in the remote block. Value data begins immediately at offset zero.



## xfs\_db Short Form Symbolic Link Example

A short symbolic link to a file is created:

```
xfs_db> inode <inode#>
xfs_db> p
core.magic = 0x494e
core.mode = 0120777
core.version = 1
core.format = 1 (local)
...
core.size = 12
core.nblocks = 0
core.extsize = 0
core.nextents = 0
...
u.symlink = "small_target"
```

Raw on-disk data with the link contents highlighted:

```
xfs_db> type text
xfs_db> p
00: 49 4e a1 ff 01 01 00 01 00 00 00 00 00 00 00 00 IN.....
10: 00 00 00 01 00 00 00 00 00 00 00 00 00 00 01 .....
20: 44 be e1 c7 03 c4 d4 18 44 be e1 c7 03 c4 d4 18 D.....D.....
30: 44 be e1 c7 03 c4 d4 18 00 00 00 00 00 00 00 0c D.....
40: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
50: 00 00 00 02 00 00 00 00 00 00 00 00 00 00 00 .....
60: ff ff ff ff 73 6d 61 6c 6c 5f 74 61 72 67 65 74 ....small.target
70: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
```

## Extent Symbolic Links

If the length of the symbolic link exceeds the space available in the inode's data fork, the link is moved to a new filesystem block and the inode's `di_format` is changed to "extents". The location of the block(s) is specified by the data fork's `di_bmx[]` array. In the significant majority of cases, this will be in one filesystem block as a symlink cannot be longer than 1024 characters.

On a v5 filesystem, the first block of each extent starts with the following header structure:

```
struct xfs_dsylink_hdr {
    __be32          sl_magic;
    __be32          sl_offset;
    __be32          sl_bytes;
    __be32          sl_crc;
    uuid_t          sl_uuid;
    __be64          sl_owner;
    __be64          sl_blkno;
    __be64          sl_lsn;
};
```

### sl\_magic

Specifies the magic number for the symlink block: "XSLM" (0x58534c4d).

**sl\_offset**

Offset of the symbolic link target data, in bytes.

**sl\_bytes**

Number of bytes used to contain the link target data.

**sl\_crc**

Checksum of the symlink block.

**sl\_uuid**

The UUID of this block, which must match either `sb_uuid` or `sb_meta_uuid` depending on which features are set.

**sl\_owner**

The inode number that this symlink block belongs to.

**sl\_blkno**

Disk block number of this symlink.

**sl\_lsn**

Log sequence number of the last write to this block.

Filesystems formatted prior to v5 do not have this header in the remote block. Symlink data begins immediately at offset zero.

---

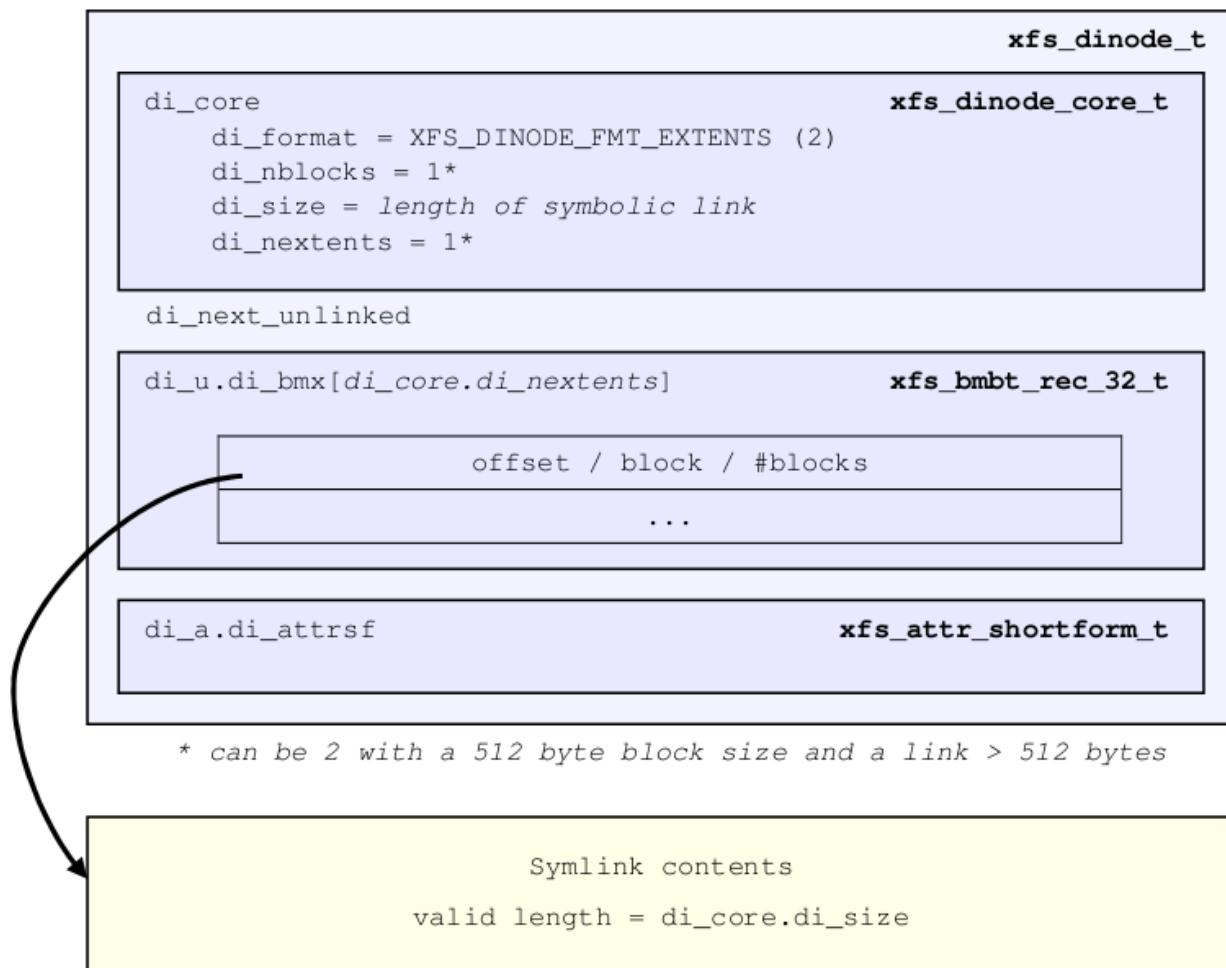


Figure 17.2: Symbolic link extent layout

### xfs\_db Symbolic Link Extent Example

A longer link is created (greater than 156 bytes):

```

xfs_db> inode <inode#>
xfs_db> p
core.magic = 0x494e
core.mode = 0120777
core.version = 1
core.format = 2 (extents)
...
core.size = 182
core.nblocks = 1
core.extsize = 0
core.nextents = 1
...
u.bmx[0] = [startoff,startblock,blockcount,extentflag] 0:[0,37530,1,0]

```

```
xfs_db> dblock 0
xfs_db> type symlink
xfs_db> p
"symlink contents..."
```