Google Git Sign in

kernel / pub / scm / fs / xfs / xfs-documentation / master / . / design / XFS_Filesystem_Structure / **allocation_groups.asciidoc**

blob: c0ba16a82d489fedc10bfafd90c26409938b7730 [file] [log] [blame]

```
[[Allocation_Groups]]
= 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 bytes × 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_LAZYSBCOUNTBIT+ 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:

.Allocation group layout
image::images/6.png[]

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

[[Superblocks]]
== 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.

```
[source, c]
----
struct xfs_sb
{
        __uint32_t              sb_magicnum;
        __uint32_t              sb_blocksize;
        xfs_rfsblock_t          sb_dblocks;
        xfs_rfsblock_t          sb_rblocks;
        xfs_rtblock_t           sb_rextents;
        uuid_t                  sb_uuid;
        xfs_fsblock_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_rbmblocks;
        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
xref:Real-time_Devices[real-time sub-volumes] 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
xref:AG_Free_Space_Management[free space] +agf_length+ value.

*sb_agcount*::
Number of AGs in the filesystem.

*sb_rbmblocks*::
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 >=
4, the higher bits indicate feature flags as follows:

.Version 4 Superblock version flags
[options="header"]
|=====
| Flag                          | Description
| +XFS_SB_VERSION_ATTRBIT+      |
Set if any inode have extended attributes.  If this bit is set; the
+XFS_SB_VERSION2_ATTR2BIT+ is not set; and the +attr2+ mount flag is not
specified, the +di_forkoff+ inode field will not be dynamically adjusted.
See the section about xref:Extended_Attribute_Versions[extended attribute
versions] for more information.

| +XFS_SB_VERSION_NLINKBIT+     | Set if any inodes use 32-bit di_nlink 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 sb_inoalignmt is used.
| +XFS_SB_VERSION_DALIGNBIT+    | Set if sb_unit and sb_width are used.
| +XFS_SB_VERSION_SHAREDBIT+    | Set if sb_shared_vn is used.
| +XFS_SB_VERSION_LOGV2BIT+     | Version 2 journaling logs are used.
| +XFS_SB_VERSION_SECTORBIT+    | Set if sb_sectsize 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 sb_features2 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 = 2^sb_blocklog^+.

*sb_sectlog*::
log~2~ value of +sb_sectsize+.

*sb_inodelog*::
log~2~ value of +sb_inodesize+.

*sb_inopblog*::
log~2~ value of +sb_inopblock+.

*sb_agblklog*::
log~2~ 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~2~ 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
xref:Quota_Inodes[quota inodes] for more information.

*sb_gquotino*::
Inode for group or project quotas. Group and project quotas cannot be used at
the same time on v4 filesystems.  On a v5 filesystem, this inode always stores
group quota information.

*sb_qflags*::
Quota flags. It can be a combination of the following flags:

.Superblock quota flags
[options="header"]
|=====
| 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.
```

```
| +XFS_GQUOTA_ENFD+                | Group quotas are enforced.
| +XFS_GQUOTA_CHKD+                | Group quotas have been checked.
| +XFS_PQUOTA_ENFD+                | Project quotas are enforced.
| +XFS_PQUOTA_CHKD+                | Project quotas have been checked.
|=====

*sb_flags*::
Miscellaneous flags.

.Superblock flags
[options="header"]
|=====
| 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:

.Extended Version 4 Superblock flags
[options="header"]
|=====
| 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.  If this bit is set and the +noattr2+
mount flag is not specified, the +di_forkoff+ inode field will be dynamically
adjusted.  See the section about xref:Extended_Attribute_Versions[extended
attribute versions] for more information.

| +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
+sb_versionnum+ 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.

.Extended Version 5 Superblock Read-Only compatibility flags
[options="header"]
|=====
| 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
xref:Reconstruction[reconstruction] for more details.

| +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 xref:Reflink_Deduplication[reflink and deduplication] for
more details.

| +XFS_SB_FEAT_RO_COMPAT_INOBTCNT+ |
Inode B+tree block counters.  Each allocation group's inode (AGI) header
tracks the number of blocks in each of the inode B+trees.  This allows us
to have a slightly higher level of redundancy over the shape of the inode
btrees, and decreases the amount of time to compute the metadata B+tree
preallocations at mount time.

|=====

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

.Extended Version 5 Superblock Read-Write incompatibility flags
[options="header"]
|=====
| 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 xref:Sparse_Inodes[Sparse Inodes] for more information.

| +XFS_SB_FEAT_INCOMPAT_META_UUID+ |
Metadata UUID.  The UUID stamped into each metadata block must match the value
in +sb_meta_uuid+.  This enables the administrator to change +sb_uuid+ at will
without having to rewrite the entire filesystem.

| +XFS_SB_FEAT_INCOMPAT_BIGTIME+ |
Large timestamps.  Inode timestamps and quota expiration timers are extended to
support times through the year 2486.  See the section on
xref:Timestamps[timestamps] for more information.

| +XFS_SB_FEAT_INCOMPAT_NEEDSREPAIR+ |
The filesystem is not in operable condition, and must be run through
xfs_repair before it can be mounted.

| +XFS_SB_FEAT_INCOMPAT_NREXT64+ |
Large file fork extent counts.  This greatly expands the maximum number of
space mappings allowed in data and extended attribute file forks.

|=====

*sb_features_log_incompat*::
Read-write incompatible feature flags for the log.  The kernel cannot recover
the FS log if it doesn't understand the flag.

.Extended Version 5 Superblock Log incompatibility flags
[options="header"]
|=====
| Flag                                    | Description
| +XFS_SB_FEAT_INCOMPAT_LOG_XATTRS+     |
Extended attribute updates have been committed to the ondisk log.

|=====

*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
xref:Real_time_Reverse_Mapping_Btree[Real-Time Reverse Mapping B+tree].
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
inodelog = 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]]
== 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]]
=== 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 knows as the +AGF+, uses the following structure:

[source, c]
----
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_btreeblks;

     /* 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_flfirst*::
Specifies the index of the first ``free list'' block. Free lists are covered in
more detail later on.

*agf_fllast*::
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_level*::
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_Btrees]]
=== 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:

[source, c]
----
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:

[source, c]
----
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:

.Freespace B+tree with one leaf.
image::images/15a.png[]

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:

.Multi-level freespace B+tree.
image::images/15b.png[]

[[AG_Free_List]]
=== AG Free List

The AG Free List is located in the 4^th^ 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_AGFL_SIZE+ macro.

Active elements in the array are specified by the
xref:AG_Free_Space_Block[AGF's] +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:

[source, c]
```

```
----
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.

.AG Free List layout
image::images/16.png[]

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]]
== AG Inode Management

[[Inode_Numbers]]
=== 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 xref:Superblocks[superblock's] +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
xref:Directories[directory] entries and the superblock.
```

```
.Inode number formats
image::images/18.png[]

[[Inode_Information]]
=== 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:

[source, c]
----
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;

     __be32              agi_iblocks;
     __be32              agi_fblocks;

}
----
*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 xref:Unlinked_Pointer[unlinked list pointers] 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.

```

```
*agi_iblocks*::
The number of blocks in the inode B+tree, including the root.
This field is zero if the +XFS_SB_FEAT_RO_COMPAT_INOBTCNT+ feature is not
enabled.

*agi_fblocks*::
The number of blocks in the free inode B+tree, including the root.
This field is zero if the +XFS_SB_FEAT_RO_COMPAT_INOBTCNT+ feature is not
enabled.

[[Inode_Btrees]]
== 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 +xfs_btree_sblock+ structure
which is the same as the header used in the xref:AG_Free_Space_Btrees[AGF
B+trees].

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'' (0x46494254).

Leaves contain an array of the following structure:

[source,c]
----
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:

[source,c]
----
struct xfs_inobt_key {
     __be32                    ir_startino;
};
typedef __be32 xfs_inobt_ptr_t;
----

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

.Single Level inode B+tree
image::images/20a.png[]


And a 2-level inode B+tree:

.Multi-Level inode B+tree
image::images/20b.png[]


=== 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 5847 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]]
== 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 xref:AG_Free_Space_Btrees[AGF B+trees].

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:

[source,c]
```
----
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]]
== 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 xref:Real-time_Inodes[real time inodes].

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.