# JFS layout
## How the Journaled File System handles the on-disk layout

Steve Best, Linux Technology Center, IBM
Dave Kleikamp, Linux Technology Center, IBM
May 2000

This article describes the on-disk Journaled File System (JFS) layout and the mechanisms used to achieve scalability, reliability, and performance using the on-disk layout structures. You'll learn about the policies and algorithms used to manipulate these structures and where JFS uses B+ trees throughout the file system to increase file system operations.

Contents:



The JFS architecture can be explained in the context of its disk layout characteristics. The on-disk layout is the format used by JFS to control the file system. This paper covers extent-based file geometry, the directory formats, the formats of block allocation maps, inodes, and other characteristics of the layout structures. It provides detail and examples of the B+ tree data structures used for file layout. B+ trees were selected to increase the performance of reading and writing extents, the most common operations that JFS does.

### Partitions, aggregates, allocation groups, filesets
Here is the "big picture" view of the on-disk layout.

### Partitions
A JFS file system is built on top of a partition, which is the abstraction exported to JFS by FDISK.

A partition has:

- A fixed partition block size, with legal values of 512, 1024, 2048, or 4096 bytes. The partition block size defines the smallest unit of I/O supported on the partition. It corresponds to the underlying disk sector size of the physical device making up the partition, with 512 bytes being the most common size.
- A size, PART_NBlocks, which is the number of partition disk blocks.
- An abstract address space, [ 0 .. PART_NBlocks - 1 ], of partition disk blocks.

### Aggregates
To support DCE DFS (Distributed Computing Environment Distributed File System), JFS separates the notion of a disk space allocation pool, called an aggregate, from the notion of a mountable file system sub-tree, called a fileset. The terms aggregate and fileset in this article correspond to their DFS usage. There is exactly one aggregate per partition; there may be multiple filesets per aggregate. In the first release, JFS only supports one fileset per aggregate; however, all of the meta-data has been designed for the fully general case.

Figure 1 shows the layout of an aggregate with two filesets.

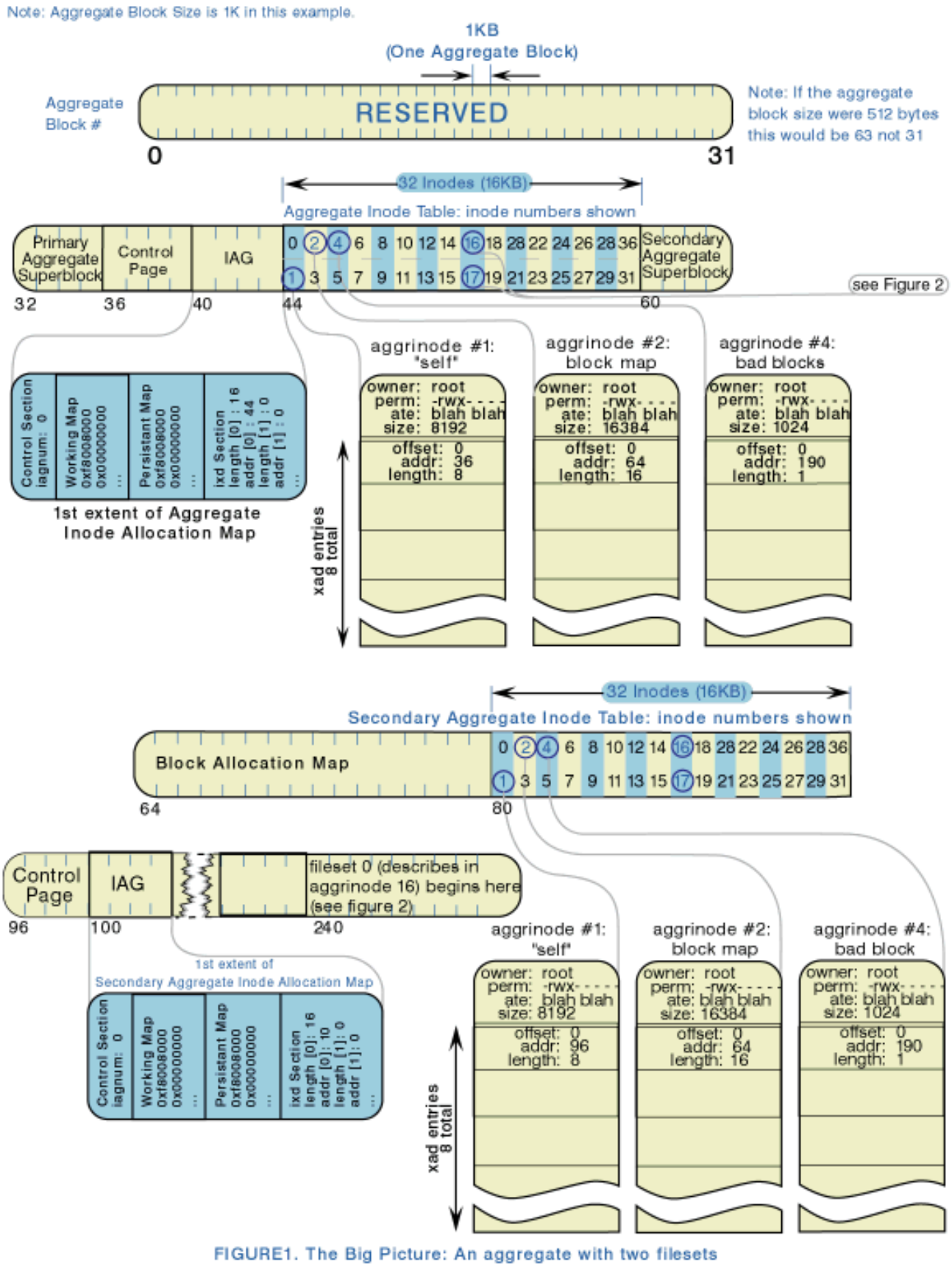



FIGURE1. The Big Picture: An aggregate with two filesets

An aggregate has:

- A 32K reserved area at the front of it.
- A fixed aggregate block size, with legal values of 512, 1024, 2048, or 4096 bytes, but no smaller than the partition block size. The aggregate block size defines the smallest unit of space allocation supported on the aggregate. Do

not confuse it with the partition block size, which defines the smallest unit of I/O.

- A Primary Aggregate Superblock and Secondary Aggregate Superblock. The superblocks contain aggregate-wide information such as the size of the aggregate, size of allocation groups, aggregate block size, etc. The secondary aggregate superblock is a direct copy of the primary aggregate superblock. The secondary superblock is used if the primary aggregate superblock is corrupted. These superblocks are at fixed locations. This allows JFS to always be able to find these without depending on any other information. The superblock structure is defined in jfs_superblock.h, `struct jfs_superblock`.
- An Aggregate Inode Table, containing inodes describing the aggregate-wide control structures. The Aggregate Inode Table logically contains an array of inodes. An aggregate has no directory structure; the aggregate inodes are not visible anywhere in the aggregate or fileset name space.
- A Secondary Aggregate Inode Table, containing replicated inodes from the Aggregate Inode Table. Since the inodes in the Aggregate Inode Table are critical for finding any file system information, they will each be replicated in the Secondary Aggregate Inode Table. The actual data for the inodes will not be repeated, just the addressing structures used to find the data and the inode itself.
- An Aggregate Inode Map, which describes the Aggregate Inode Table. The Aggregate Inode Allocation Map contains allocation state information on the aggregate inodes as well as their on-disk location.
- A Secondary Aggregate Inode Map, which describes the Secondary Aggregate Inode Table. Since the Aggregate Inode Table itself must be duplicated, the Secondary Aggregate Inode Map is actually a separate mapping structure from the Aggregate Inode Allocation Map.
- A Block Allocation Map, which describes the control structures for allocating and freeing aggregate disk blocks within the aggregate. The Block Allocation Map maps one-to-one within the aggregate disk blocks.
- A `fsck` Working Space (not shown in Figure 1), which provides space for `fsck` to track the aggregate block allocation. This space is necessary because JFS supports very large aggregates; there might not be enough memory to track this information in memory when `fsck` is run. The space is described by the superblock. One bit is needed for every aggregate block. The `fsck` working space always exists at the end of the aggregate.
- An In-line Log (not shown in Figure 1) provides space for logging of meta-data changes of the aggregate. The space is described by the superblock. The in-line log always follows the `fsck` working space.

Initially, the first inode extent is allocated when the aggregate is created. Additional inode extents are allocated and deallocated dynamically as needed. These Aggregate Inodes each describe certain aspects of the aggregate itself, as follows:

- Aggregate Inode zero is reserved.
- Aggregate Inode one, the "self" inode, describes the aggregate disk blocks comprising the Aggregate Inode Map. This is a circular representation, in that Aggregate Inode one is itself in the file that it describes. The obvious circular representation problem is handled by forcing at least the first aggregate inode extent to appear at a well-known location, namely, 4K after the Primary Aggregate Superblock. Therefore, JFS can easily find Aggregate Inode one, and from there it can find the rest of the Aggregate Inode Table by following the B+ tree in Inode one.
- To duplicate the Aggregate Inode Table, JFS will also need to find the copy of the Aggregate Inode one to find the rest of the duplicated table. The superblock will contain an extent descriptor that describes the location of the first inode extent of the Second Aggregate Inode Table. From that JFS will be able to find the Secondary Aggregate Inode one and the rest of the Secondary Aggregate Inode Table.
- Aggregate Inode two describes the Block Allocation Map.
- Aggregate Inode three describes the In-line Log when mounted. This inode is allocated, but no data is saved to disk.
- Aggregate Inode four describes the bad blocks discovered during formatting of the aggregate. These bad blocks are marked allocated in the block map. This inode is a normal file whose data is the bad blocks.
- Aggregate Inodes five through 15 are reserved for future extensions.
- Starting at Aggregate Inode 16, there is one inode per fileset, the Fileset Allocation Map Inode. This inode describes the control structures that represent filesets. As additional filesets are added to the aggregate, the Aggregate Inode Table itself may have to grow to accommodate additional fileset inodes.

**Allocation groups**

Allocation groups (AGs) divide the space in an aggregate into chunks, and allow JFS resource allocation policies to use well known methods for achieving great JFS I/O performance. First, the allocation policies try to cluster disk blocks and disk inodes for related data to achieve good locality for the disk. Files are often read and written sequentially, and the files within a directory are often accessed together. Second, the allocation policies try to distribute unrelated data throughout the aggregate in order to accommodate locality. Allocation groups within an aggregate are identified by a zero-based AG index, the AG number.

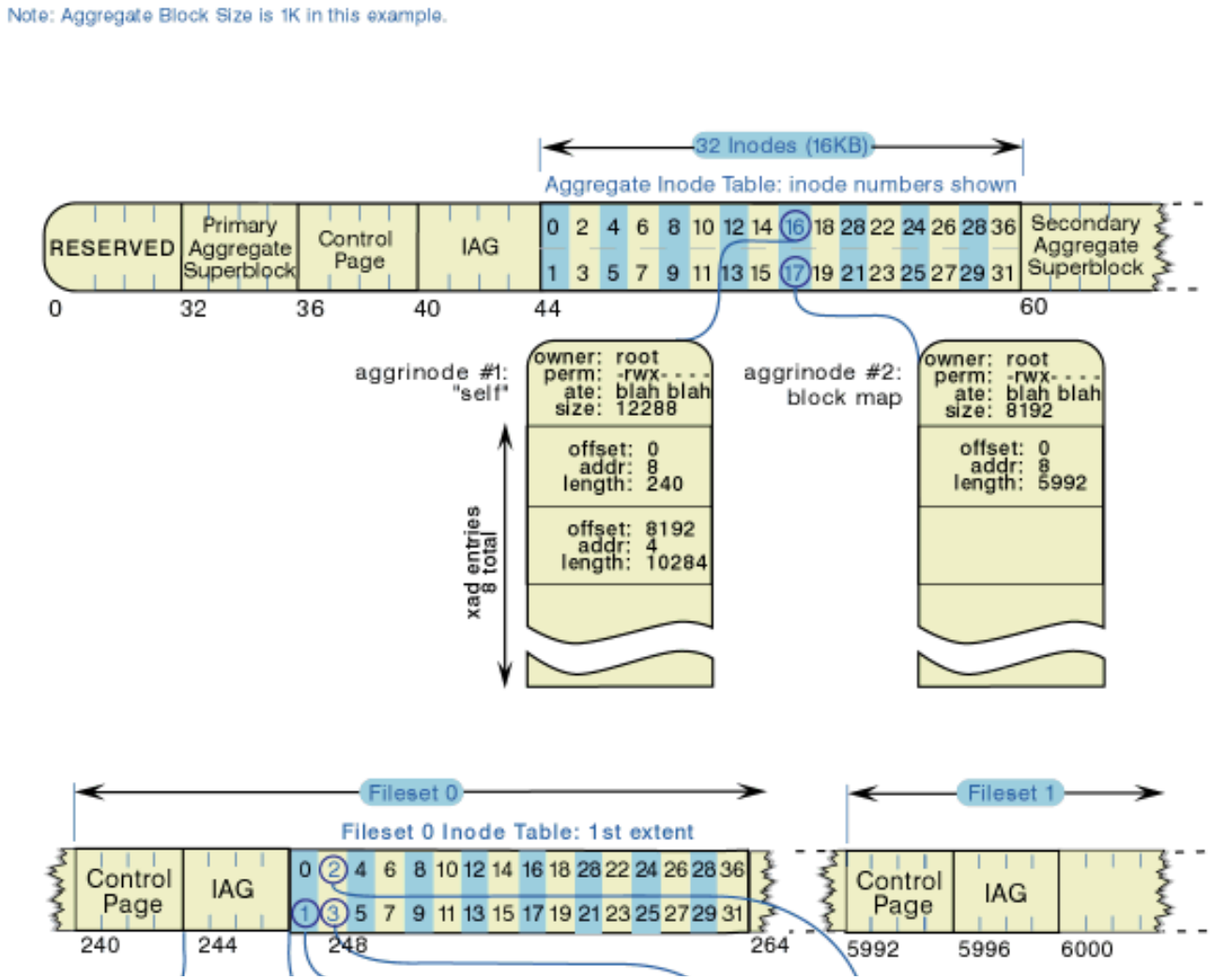
Allocation group sizes must be selected, which yield AGs that are sufficiently large to provide for contiguous resource allocation over time. To minimize the number of updates that need to be done when an aggregate is expanded or shrunk, the allocation groups need to be limited to a maximum number of groups, 128. Additionally, JFS will impose a minimum on the allocation group size of 8192 aggregate blocks. The allocation group size must always be a power of 2 multiple of the number of blocks described by one dmap page (1, 2, 4, 8, ... dmap pages). The allocation group size is stored in the aggregate superblock.

An aggregate whose size is not a multiple of the allocation group size will contain a partial allocation group; the last allocation group of the aggregate is not fully covered by disk blocks. This partial allocation group will be treated as a complete allocation group, except JFS will mark the non-existent disk blocks allocated in the Block Allocation Map.

**Filesets**

A fileset is a set of files and directories that form an independently mountable sub-tree. A fileset is completely contained within a single aggregate. Note that multiple filesets may exist within a single aggregate; in that case, all of the filesets share a common pool of free aggregate disk blocks as defined by the aggregate control structures.

Figure 2 shows the layout of two filesets contained in an aggregate.



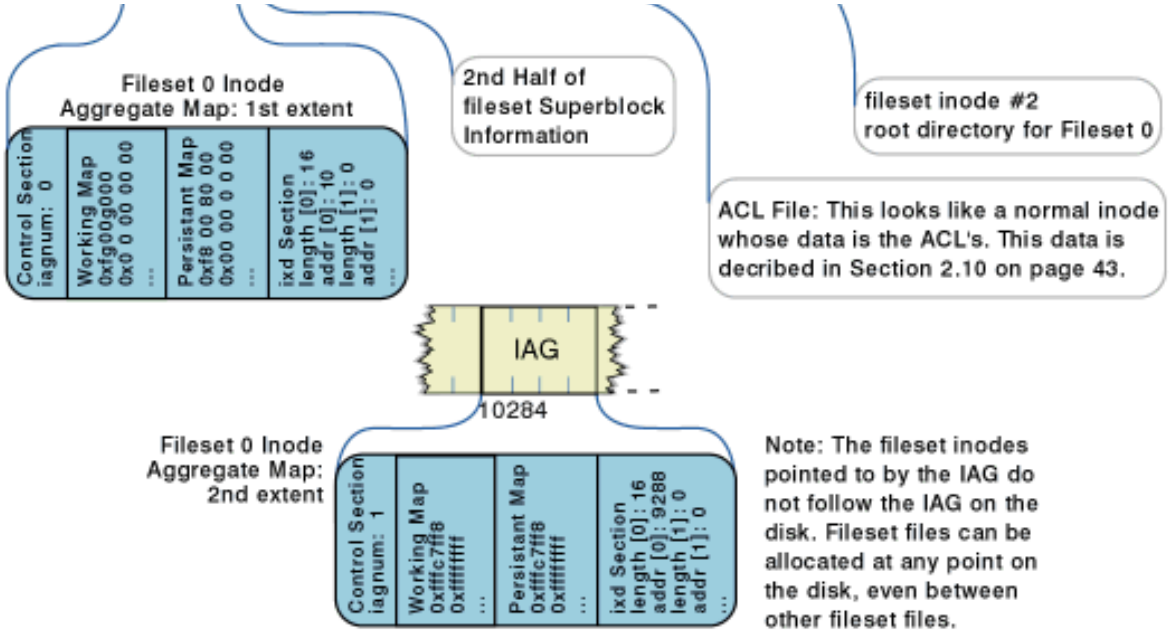



FIGURE 2. Two filesets in an aggregate

A fileset has:

- A Fileset Inode Table, containing inodes describing the fileset-wide control structures. The Fileset Inode Table logically contains an array of inodes.
- A Fileset Inode Allocation Map, which describes the Fileset Inode Table. The Fileset Inode Allocation Map contains allocation state information on the fileset inodes as well as their on-disk location. The "super-inode" describing the Fileset Allocation Map and other fileset information resides in the Aggregate Inode Table as previously described. Since the Aggregate Inode Table is replicated, there is also a secondary version of this inode, which points to the same data. The "super-inode" is itself a file. When the fileset is initially created, the first inode extent is allocated; additional inode extents are allocated and deallocated dynamically as needed.

The inodes in a fileset are allocated as follows:

- Fileset inode zero is reserved.
- Fileset inode one contains additional fileset information that would not fit in the Fileset Allocation Map Inode in the Aggregate Inode Table.
- Fileset inode two is the root directory inode for the fileset. Note that JFS preserved the common Unix convention that inode number two is the root of the file "system."
- Fileset inode three is the ACL file for the fileset.
- Fileset inodes starting with four are used by ordinary fileset objects, user files, directories, and symbolic links.

**Extents, inodes, B+ trees**

An extent is a sequence of contiguous aggregate blocks allocated to a JFS object as a unit. An extent is wholly contained within a single aggregate (and therefore a single partition); however, large extents may span multiple allocation groups.

Every JFS object is represented by an inode. Inodes contain the expected object-specific information such as time stamps and file type (regular vs. directory, etc.). They also "contain" a B+ tree to record the allocation of extents. Note specifically that all JFS meta data structures (except for the superblock) are represented as "files". By reusing the inode structure for this data, the data format (on-disk layout) becomes inherently extensible.

Details of extents, B+ trees, and inodes are in the sections that follow.

**Extents**

A "file" is allocated in sequences of extents. An extent is a contiguous variable-length sequence of aggregate blocks allocated as a unit. An extent can range in size from 1 to 2(24) - 1 aggregate blocks. An extent may span multiple Allocation Groups (AGs). These extents are indexed in a B+ tree for better performance in inserting new extents, locating particular extents, etc.

Two values are needed to define an extent, its length and its address. The length is measured in units of aggregate block size. JFS uses a 24-bit value to represent the length of an extent, so an extent can range in size from 1 to 2(24) - 1 aggregate blocks.

With a 512-byte aggregate block size (the smallest allowable), the maximum extent is 512 * (2(24) - 1) bytes long (slightly under 8G). With a 4096-byte aggregate block size (the largest allowable), the maximum extent is 4096 * (2(24) - 1) bytes long (slightly under 64G). These limits only apply to a single extent; they have no limiting effects on overall file size. The address is the address of the first block of the extent. The address is also in units of the aggregate blocks: it is the block offset from the beginning of the aggregate.

An extent-based file system combined with a user-specified aggregate block size allows JFS to need no separate support for internal fragmentation. You can configure the aggregate with a small aggregate block size (for example, 512 bytes) to minimize internal fragmentation for aggregates with a large number of small size files.

In general, the allocation policy for JFS tries to maximize contiguous allocation by allocating a minimum number of extents, with each extent as large and contiguous as possible. This allows for large I/O transfer, resulting in improved performance. However, in special cases this is not always possible. For example, copy-on-write clones of a segment will cause a contiguous extent to be partitioned into a sequence of smaller contiguous extents. Another case is restriction of extent size. For example, the extent size is restricted for compressed files since JFS must read the entire extent into memory and decompress it. JFS has a limited amount of memory available, so it must ensure that it will have enough room for the decompressed extent.

A defragmentation utility is provided to reduce external fragmentation, which occurs from dynamic allocation/deallocation of variable-size extents. This allocation and deallocation can result in disconnected variable size free extents all over the aggregate. The defragmentation utility will coalesce multiple small free extents into single larger extents.

**Inodes**

JFS on-disk inode is 512 bytes. A JFS on-disk inode contains four basic sets of information. The first set describes the POSIX attributes of the JFS object. The second set describes additional attributes for JFS object; these attributes include information necessary for the VFS support, information specific to the OS environment, and the header for the B+ tree. The third set contains either the extent allocation descriptors of the root of the B+ tree or in-line data. The fourth set contains extended attributes, more in-line data, or additional extent allocation descriptors. The definition of the on-disk inode structure is defined in jfs_dinode.h, `struct dinode`.

JFS allocates inodes dynamically, which provides the following advantages:

- Inode disk blocks may be placed at any disk address, which decouples the inode number from the location. This decoupling simplifies supporting aggregate and fileset reorganization to enable shrinking the aggregate. The inodes can be moved, and they will still have the same number. This allows JFS not to need to search the directory structure to update the inode numbers. The decoupling is also necessary for supporting DFS fileset cloning. When a fileset is cloned, just the inodes are copied. Since JFS can put the new inodes anywhere on disk, the new inodes will have the same numbers as the inodes they are copied from. This allows JFS not to have to copy the directory structures and update the inode numbers.
- It eliminates the need to allocate "ten times as many inodes as you will ever need." This is especially important with the larger inode size (512 bytes) in JFS.
- File allocation for large files can consume multiple allocation groups and still be contiguous, whereas static allocation forces a gap (for the initially allocated inodes in each allocation group).

On the other hand, dynamic inode allocation causes a number of problems, including the following:

- With static allocation the geometry of the file system implicitly describes the layout of inodes on disk; with dynamic allocation separate mapping structures are required.
- Those mapping structures are critical to JFS integrity. Due to the overhead involved in replicating these structures, JFS has decided to accept the risk of loss of these maps. However, JFS will replicate the B+ tree structures, which allows JFS to find the maps.

Inodes are allocated dynamically by allocating inode extents that are simply a contiguous chunk of inodes on the disk. By

definition, a JFS inode extent contains 32 inodes. With a 512-byte inode size, an inode extent is therefore 16KB in size on the disk.

When a new inode extent is allocated, the extent is not initialized. However, for `fsck` to be able to check if an inode is in use, JFS will need some information in the inode to check. Once an inode in an extent is marked in-use, its fileset number, inode number, inode stamp, and the inode allocation group block address must be initialized. Thereafter, the link field will be sufficient to determine if the inode is currently in use.

Notice that dynamic inode allocation implies that there is no direct relationship between an inode number and the disk address of the inode. Therefore, JFS must have a means of finding the inodes on disk. The Inode Allocation Map provides this function.

Inodes generation numbers are simply counters that get incremented each time an inode is reused.

The static-inode-allocation practice of storing a per-inode generation counter doesn't work with dynamic inode allocation, because when an inode becomes free, its disk space may literally be reused for something other than an inode (in other words, the space may be reclaimed for ordinary file data storage). Therefore, in JFS there is simply one inode generation counter that is incremented on every inode allocation, rather than one counter per inode that would be incremented when that inode is reused.

**B+ trees**

This section describes the B+ tree data structure used for file layout. B+ trees were selected to increase the performance of reading and writing extents, which are the most common operations JFS will have to do. B+ trees provide a fast search for reading a particular extent of a file. They also provide an efficient way to append or insert an extent in a file. Less commonly, JFS will need to traverse an entire B+ tree when removing a file. In order to ensure JFS will remove the blocks used for the B+ tree as well as the file data, the B+ tree is also efficient for traversal.

An extent allocation descriptor (xad structure) describes the extent and adds two more fields that are needed for representing files: an offset, describing the logical byte address the extent represents, and a flags field. The extent allocation descriptor structure is defined in jfs_xtree.h, `struct xad`.

The xad structure is:

```
struct xad {
        unsigned    flag:8;
        unsigned    rsvrd:16;
        unsigned    off1:8;
        uint32      off2;
        unsigned    len:24;
        unsigned    addr1:8;
        uint32      addr2;
        } xad_t;
```

where:

- **flag** is an 8-bit field containing miscellaneous flags. These flags can indicate copy-on-write, if the extent is allocated but not recorded, information for compression, etc.
- **rsvrd** is a 16-bit field reserved for future use. It is always zero.
- **off1,off2** is a 40-bit field, containing the logical offset of the first block in the extent. The logical offset is represented in units of the aggregate block size; in other words, to get a byte, offset must be multiplied by the aggregate block size.
- **len** is a 24-bit field, containing the length of the extent. The length is represented in units of aggregate block size.
- **addr1,addr2** is a 40-bit field, containing the address of the extent. The address is represented in units of aggregate block size.

An xad structure describes two abstract ranges:

- The physical range of disk blocks on the disk. This starts at aggregate block number xad_address and extends for xad_length aggregate blocks.

- The logical range of bytes within a file. This starts at byte number xad_offset * AGBS (aggregate block size) and extends for xad_length * AGBS bytes.

The physical range and logical range are, of course, both the same number of bytes long. Note that `xad_offset` is stored in units of aggregate block size (for example, a value of "3" in `xad_offset` means 3 aggregate blocks, not 3 bytes). It follows from this that extents within a file are always aligned on aggregate block size boundaries.

There is one generic B+ tree index structure for all index objects (except for directories) in JFS. The data being indexed will depend on the object. The B+ tree is keyed by offset of xad of data being described by the tree. The entries are sorted by the offsets of the xad structures. An xad structure is an entry in a node of a B+ tree.

Figure 3 shows a single xad structure and how it describes both the range of bytes logically within the file as well as the physical location of that range of bytes on the disk itself (in other words, with the aggregate).

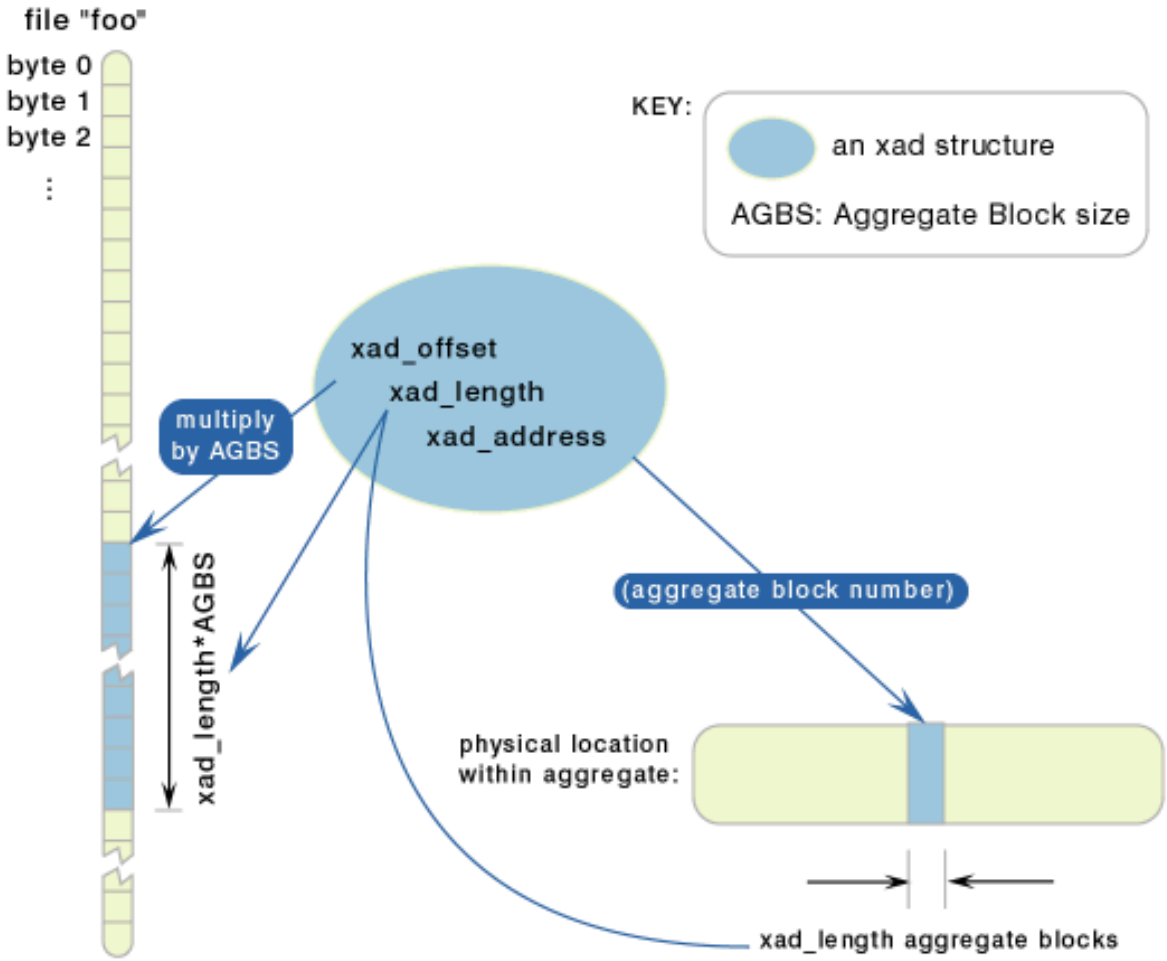
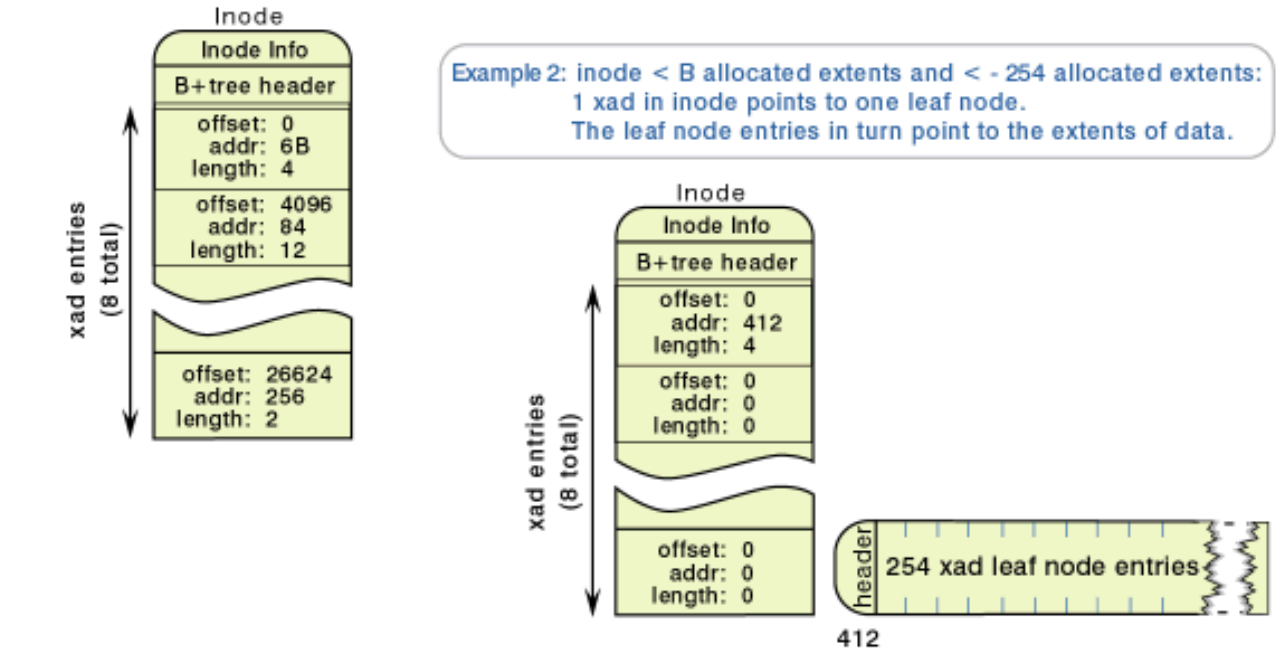



FIGURE 3. xad describes two "ranges"

The bottom of the second section of a disk inode contains a data descriptor that tells what is stored in the second half of the inode. The second half could contain in-line data for the file if it is small enough. If the file data won't fit in the in-line data space for the inode, it will be contained in extents, and the inode will contain the root node of the B+ tree. The header will indicate how many xad are in use and how many are available. Generally, the inode will contain 8 xad structures for the root of the B+ tree. If there are 8 or fewer extents for the file, then these 8 xad structures are also a leaf node of the B+ tree. They will describe the extents. (See Figure 4, example 1.) Otherwise the 8 xad structures in the inode will point to either the leaves or internal nodes of the B+ tree.



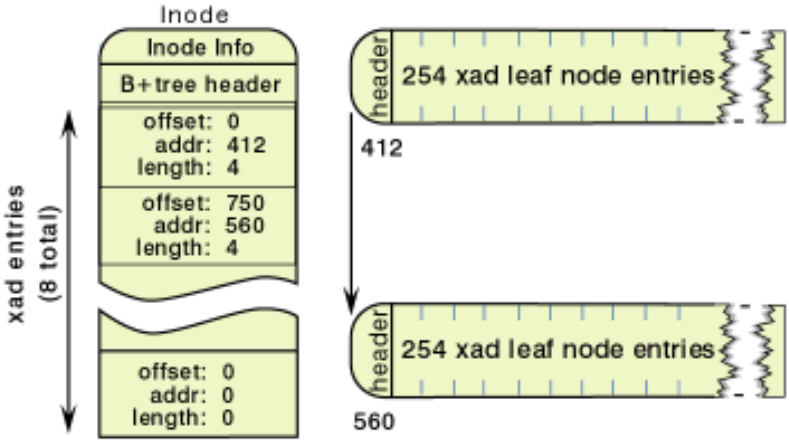
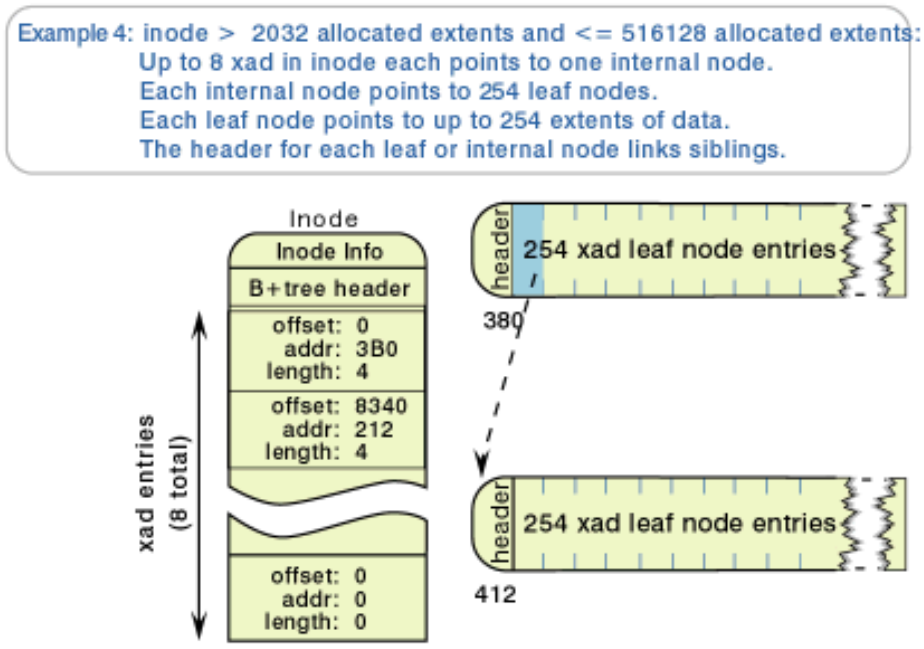




FIGURE 4. Examples of B+ -tree operation



Once the 8 xad structures in the inode are filled, an attempt will be made to use the last quadrant of the inode for more xad structures. If the INLINEEA bit is set in the `di_mode` field of the inode, then the last quadrant of the inode is available.

Once all of the available xad structures in the inodes are used, the B+ tree must be split. JFS will allocate 4K of disk space for a leaf node of the B+ tree. A leaf node is logically an array of xad entries with a header. The header points to the first free xad entry in the node, all xad entries following that one are also not allocated. The 8 xad entries are copied from the inode to the leaf node, the header is initialized to point to the 9th entry as the first free entry. Then JFS will update the root of the B+ tree into the inode's first xad structure; this xad structure will point to the newly allocated leaf node. The offset for this new xad structure will be the offset of the first entry in the leaf node. The header in the inode will be updated to indicate that now only 1 xad is being used for the B+ tree. The header in the inode also needs to be updated to indicate that the inode now contains the pure root of the B+ tree. (See Figure 4, example 2.)

As new extents are added to the file, they will continue to be added to this same leaf node in the necessary order. This will continue until this node fills. Once the node fills a new 4K of disk space will be allocated for another leaf node of the B+ tree. The second xad structure from the inode will be set to point to this newly allocated node. (See Figure 4, example 3.)

This will continue until all 8 xad structures in the inode are filled, at which time another split of the B+ tree will occur. This split will create internal inodes of the B+ tree which are used purely to route the searches of the tree. JFS will allocate 4K of disk space for an internal node of the B+ tree. An internal node looks the same as a leaf node. The 8 xad entries are copied from the inode to the internal node, the header is initialized to point to the 9th entry as the first free entry. Then JFS will update the root of the B+ tree by making the inode's first xad structure pointed to the newly allocated internal inode. The header in the inode will be updated to indicate that only 1 xad is being used for the B+ tree. (See Figure 4, example 4.)

The file jfs_xtree.h describes the header for the root of the B+ tree in `struct xtpage_t`. The file jfs_btree.h is the header for an internal node or a leaf node in `struct btpage_t`.

**Examples**
The following examples further illustrate the use of extent descriptors and xad structures:

- A 1041377 byte file, allocated contiguously.
- The same 1041377 byte file, but split into three pieces on the disk.
- A 1041377 byte file, but with a "hole" in it (a sparse file).
- A 16GB file, allocated contiguously.

In all of these examples, the aggregate block size is 1KB.

**1041377 byte file, allocated contiguously:** This file requires 1017 1KB aggregate blocks (with 31 bytes in the last aggregate block lost to internal fragmentation). Only one xad structure is required to describe this contiguous file:

```
flag        not discussed here
offset      0                   /*  the beginning of the file */
length      1017                /*  1017 1KB aggregate blocks */
address     xxxxx               /*  aggregate block #         */
```

This same xad structure could represent any contiguous file of size 1040385 (1016 * 1024 + 1) to 1041408 (1017 * 1024), because extent descriptors only represent sizes down to aggregate block size granularity. Only the inode `di_size` field records byte granularity.

**1041377 byte file, in three pieces:** Assume that the same file is split into three separate extents on the disk: one 495 aggregate blocks long, one 22, one 500. It requires three xad structures to represent this file one per physical extent:

```
xad #0:
flag        not discussed here
offset      0                   /*  the beginning of the file */
length      495                 /*  495 1KB aggregate blocks  */
address     xxxxx               /*  aggregate block #         */

xad #1:
flag        not discussed here
offset      495                 /*  the beginning of the file */
length      22                  /*  22 1KB aggregate blocks   */
address     yyyyy               /*  aggregate block #         */

xad #2:
flag        not discussed here
offset      517                 /* the beginning of the file  */
length      500                 /* 500 1KB aggregate blocks   */
address     zzzzz               /* aggregate block #          */
```

In this case, xad number 0 describes the first 495 physical aggregate blocks of the file. The `xad_offset` field contains zero, because this xad describes the bytes starting at logical offset zero. The next xad, xad number 1, describes the next 22 physical aggregate blocks of the file. The `xad_offset` field contains 495, because this xad describes the bytes starting at logical offset 506880 (495 * 1024); the previous bytes being described by xad 0. The final xad describes the last 500 blocks of the file. The `xad_offset` field here is 517. Notice that for files which are not sparse, the `xad_offset` field of a given xad is equal to the sum of the lengths of all previous xad structures (517 = 495 + 22 in this example). If this relationship were always true, the `xad_offset` fields would be redundant and could be eliminated. However, the next example shows that, for sparse files, the `xad_offset` field is not redundant.

**1041377 byte sparse file:** Consider a file created via the following POSIX style operations:

```
fd = create ("newfile", blah blah blah);
write (fd, "hi", 2);
lseek (fd, 1041374, 0);
write (fd, " bye" , 3);
```

This file has two bytes of data ("hi") starting at logical byte offset zero, and three more bytes starting at logical byte offset 1,041,374 ("bye"), and would be all zero (sparse) in between. The file is 1041377 bytes long.

In general, JFS does not allocate physical disk space to hold byte ranges of a file that has never been written to. Therefore, it will take two xad structures to represent this file: one for an extent containing the "hi" data, and one for an extent containing the "bye" data:

```
xad #0 :
flag        not discussed here
offset      0                   /* the beginning of the file  */
length      1                   /* 1 1KB aggregate blocks     */
address     xxxxx               /* aggregate block #          */

xad #1:
flag        not discussed here
offset      1016                /* the beginning of the file  */
length      1                   /* 1 1KB aggregate blocks     */
address     yyyyy               /* aggregate block            */
```

In this case, the first extent (xad 0) contains the bytes "hi", followed by 1022 bytes of zero. The last extent (xad 1) contains 990 bytes of zero, followed by the 3 bytes of "bye". The remaining 31 bytes in the 1KB extent are not part of the file (they are the same 31 bytes lost to internal fragmentation as in the first example).

Notice that in this case the `xad_offset` fields are necessary; they are the only way to know that xad 1 represents a

sequence of bytes that are at an "unexpected" logical offset within the file (that is, the offset for xad 1 does not equal the offset of xad 0 + length). This is how sparse files are represented.

The `di_size` field of the inode will contain the offset value of the last byte written plus one.

**16GB file, allocated contiguously:** The length field in an xad structure is only 24 bits long: therefore, it can hold a value of up to 2(24) - 1. If the aggregate block size is 1KB ( for example), then the longest extent a single xad can represent is (2(24) - 1 ) * 2(10) = 1KB less than 16G. By implication, this is also the largest extent a single xad structure can represent. Thus, if a file is large enough, it will require multiple xad structures to represent it, even if the file is contiguous on disk. This example shows such a file: a 16G file, allocated contiguously starting at aggregate block number 12345 and going for 16777216 1KB aggregate blocks (16G).

```
xad #0:
flag        not discussed here
offset      0                   /* the beginning of the file  */
length      16777215            /* 1 1KB aggregate blocks     */
address     12345               /* aggregate block            */

xad #1:
flag        not discussed here
offset      16777215            /* the beginning of the file  */
length      1                   /* 1 1KB aggregate blocks     */
address     16789560            /* aggregate block #          */
```

In this case, whether or not the file is contiguous on the disk, it will take at least two xad structures to represent it, due to the length limitation of an individual extent.

**Block Allocation Map**

The Block Allocation Map is used to track the allocated or freed disk blocks for an entire aggregate. Since all of the filesets within an aggregate share the same pool of disk blocks, this allocation map is used by all of the filesets within an aggregate when allocating or freeing disk blocks.

The Block Allocation Map is itself a file described by aggregate inode 2. When the aggregate is initially created, the data blocks for the map to cover the aggregate space are allocated. The map may grow or shrink dynamically as the aggregate is expanded or shrunk.

The Block Allocation Map tracks if each individual aggregate block is allocated or freed.

Figure 5 shows the logical and physical structure of the map as indexed by the block map inode. Each page of the map is 4K in length. The map contains three types of pages: the bmap control page, the dmap control pages, and the dmap pages.

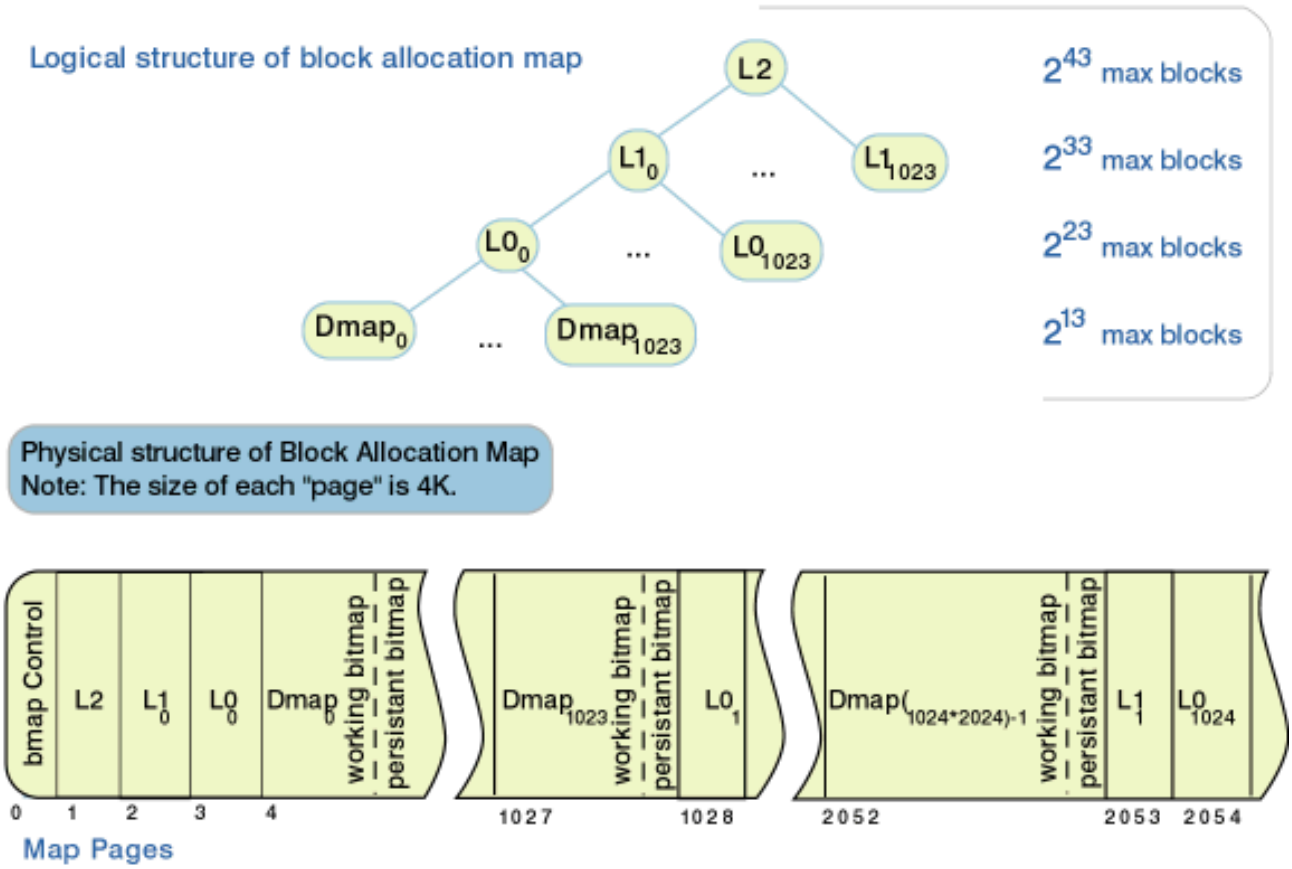



FIGURE 5. Block Allocation Map

Each dmap contains a single bit to represent each aggregate block. The i-th bit represents the allocation status of the i-th logical aggregate block. This is defined by `struct dmap_t`, in jfs_dmap.h file. Each dmap page covers 8K of aggregate blocks.

Because the Block Allocation Map may have many dmap pages they are organized by the dmap control pages, LN in Figure 5. These pages improve the performance of finding large extents of free blocks. The size of the aggregate will determine how many of these pages and how many levels are needed. At most there will be three levels, which allows a maximum size of 2(43) aggregate blocks. If not all of the levels are needed, the Block Map Inode will be a sparse file with holes for the first page of each of the unused levels.

JFS employs a commit strategy to insure that the control data is reliably updated. Reliable update means that consistent JFS structure and resource allocation state is maintained in the face of system failures. In order to ensure the Block Allocation Map is in a consistent state JFS maintains two maps in the dmap structure, the working map and the persistent map. The working map records the current allocation state. The persistent map records the committed allocation state, consisting of the allocation state as found on disk or described by records within the JFS log or committed JFS transactions. When an aggregate block is freed, the permanent map is updated first. When an aggregate block is allocated, the working map is updated first. A bit of 0 represents a free resource and a value of 1 an allocated resource.

The dmap control pages of the Block Allocation Map contain a tree similar to the tree in a dmap structure, except the leaf level contains 1024 elements. The dmap control page is defined by struct `dmapctl_t` which can be found in the jfs_dmap.h.

Figure 6 shows the detail of a tree field from one dmap structure. Note this field in the dmap structure is a flat array, but it represents a tree as shown. The tree tracks the maximum number of contiguous blocks at each level except the bottom level. The bottom level of the tree, tree[85] through tree [341], contains the binary buddy representation of the working map as described below. The other levels of the tree contain the maximum number of contiguous free blocks from four sections of the next lower level.

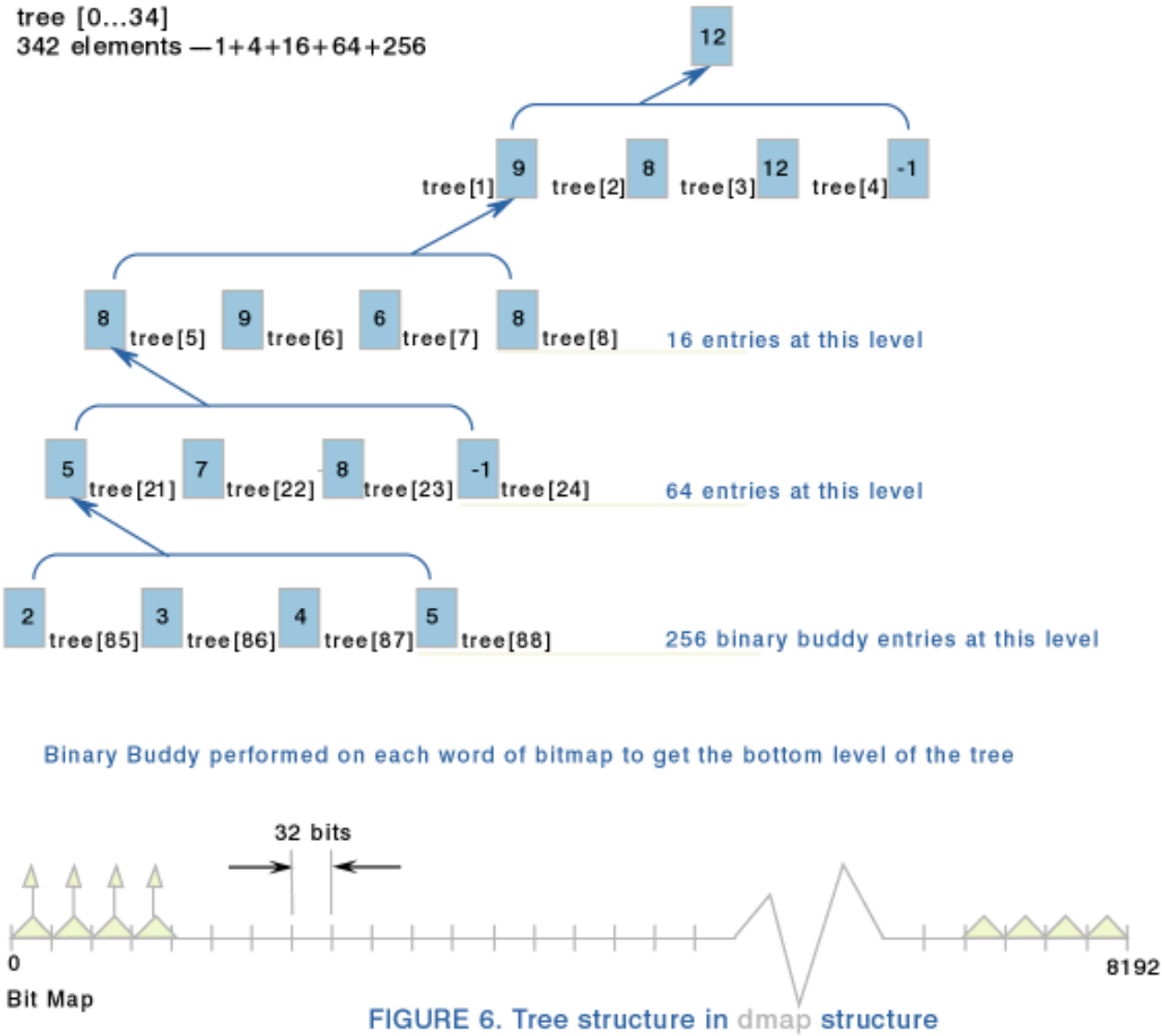
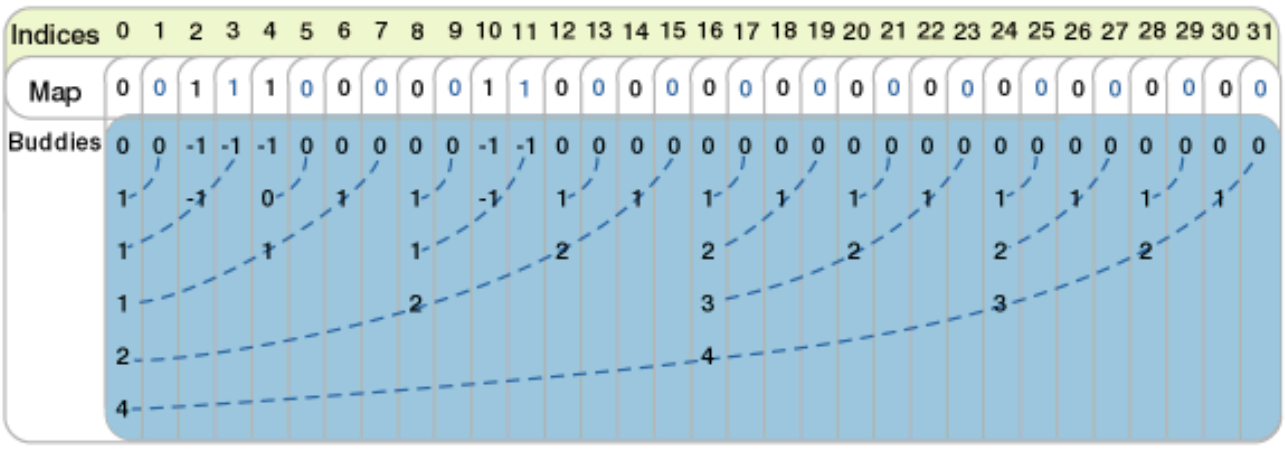



FIGURE 6. Tree structure in dmap structure

The binary buddy system is used to complete the leaf level of each of the summary trees. The tree in the dmap structure is formed by first obtaining the longest binary buddy string of free bits for each word of the bitmap. The strings are encoded as a power of 2, with -1 being used to represent all allocated.

Figure 7 shows an example of finding the longest binary buddy string of free bits for a word.

FIGURE 7. Binary Buddy of a word of the bitmap



The binary buddy system is then used to complete the leaf of the tree. The tree is formed by taking the longest number of free blocks starting at the specified index, including only its buddy shown as a power of 2.

Figure 8 shows a shortened example of figuring the leaf of the dmap tree. Note that only completely free words are combined with their completely free buddy. When combined, the right-most buddy is turned into a -1 to indicate it is represented by another entry.

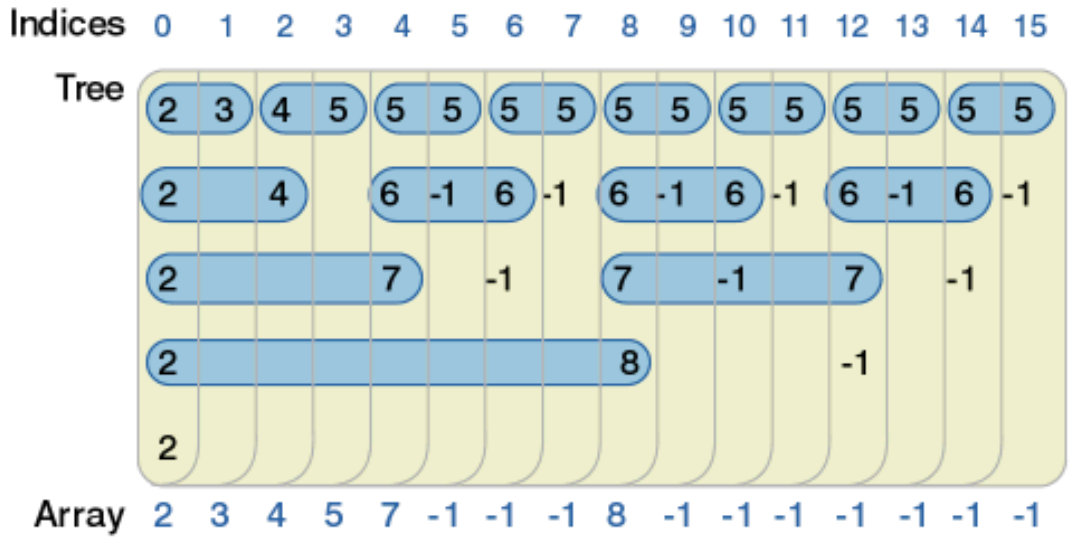



FIGURE 8. Binary Buddy

The dmap control pages of the Block Allocation Map contain a tree similar to the tree in dmap structure except the leaf level contains 1024 elements. These elements would be the binary buddy representation of tree [0] for the following 1024 map pages. For L0 pages it would be the following 1024 dmap pages, for L1 pages it would be the following 1024 L0 pages, and for L2 page it would be the following 1024 L1 pages.

At the top of the Block Allocation Map there is a map control structure, `struct dbmap_t`. This structure contains summary information which speeds up the finding of AGs which have more than average free space. The structure can be found in the jfs_dmap.h.

The Block Allocation Map is not journaled: it can be repaired during recovery time by `logredo` or reconstructed by `fsck`. Both the working and persistent maps need to be the same state after `fsck` or `logredo`.

**Extending Aggregate to increase file system size**

To increase the size of an aggregate on-line, JFS must first make sure that JFS has the Block Allocation Map pages necessary to cover the new aggregate blocks. Generally, the space for the extra map pages can come from the existing aggregate, but in the case where the aggregate is 100% full this is impossible. Therefore we need a method to handle this special case.

To solve this problem JFS always allocates more space for the Block Allocation Map than it needs to actually address the aggregate space. Each map will have an extra page of space for bitmaps, and if this page would cause another level of summary trees then the map will have extra pages for the necessary summary information. This extra space allows JFS to break the expansion of the aggregate into smaller pieces if necessary to grow the aggregate to the desired size. The following steps will be followed when an attempt is made to extend the aggregate:

1. If there is enough free space in the existing aggregate to extend the Block Allocation Map to the size needed to address all of the blocks for the new aggregate, then JFS will go ahead and extend the aggregate to the full size. Extra pages will be added as necessary to the Block Allocation Map to handle future extensions of the aggregate.
2. If there wasn't enough space for a complete extend, JFS will extend the aggregate by just the amount of blocks which can be addressed by the extra page already available in the Block Allocation Map.
3. Now JFS has some extra aggregate blocks which aren't yet being used by anything in the aggregate. JFS can use

these aggregate blocks to add space to the Block Allocation Map to continue extending the aggregate until JFS reaches the asked-for final size. JFS must remember to always keep the extra pages in the Block Allocation Map while doing this.

This interaction will be hidden from the rest of the system by having the `vfs_cntl()` call handle this completely.

**Alternative encoded binary buddy representation**
The Block Allocation Map can also be represented using an encoded binary buddy system. This representation has the same physical and logical structure as the previous representation except the leaves of the trees look different and the dmap structure looks different.

The lowest level of the Block Allocation Map is defined by `struct dmap`. Each dmap page covers 8K of aggregate blocks.

```
/*
 *      dmap summary tree
 *
 * dmaptree_t must be consistent with dmapctl_t.
 */
typedef struct {
        int32   nleafs;                 /* 4: number of tree leafs         */
        int32   l2nleafs;               /* 4: l2 number of tree leafs      */
        int32   leafidx;                /* 4: index of first tree leaf     */
        int32   height;                 /* 4: height of the tree           */
        int8    budmin;                 /* 1: min l2 tree leaf value to combine*/
        int8    stree[TREESIZE];        /* TREESIZE: tree                  */
        uint8   pad[2];                 /* 2: pad to word boundary         */
} dmaptree_t;                           /* - 360 -                         */


/*
 *      dmap page per 8K blocks bitmap
 */
typedef struct {
        int32           nblocks;          /* 4: num blks covered by this dmap */
        int32           nfree;            /* 4: num of free blks in this dmap */
        int64           start;            /* 8: starting blkno for this dmap  */
        dmaptree_t      tree;             /* 360: dmap tree                   */
        uint8           pad[1672];        /* 1672: pad to 2048 bytes          */
        uint32          wmap[LPERDMAP];   /* 1024: bits of the working map    */
        uint32          pmap[LPERDMAP];   /* 1024: bits of the persistent map */
} dmap_t;                               /* - 4096 -                         */
```

In the encoded binary buddy system each entry has three fields: `type`, `size`, and `bitmap`. The `type` field indicates whether the blocks are free, allocated, represented by the bitmap, or not represented by this field (don't care). If the type is "don't care" then those blocks are described by its left buddy and the `size` field is ignored. If the `type` is a bitmap then the `bitmap` field is used to map one-to-one with the 32 blocks to indicate whether each is free or allocated. A bit value of 0 represents a free block and a value of 1 represents an allocated block. The size is the power of 2 representation indicating how many aggregate blocks are described by the entry.

For each completely free entry, if its left-hand buddy of the same size is also completely free, then the right entry is turned into a "don't care" type. The left-hand buddy's size is incremented to include the right buddy. When allocating blocks, the buddies are only combined if they are allocated to the same extent. The "don't care" types are necessary to maintain for `logredo` to be able to correct the map.

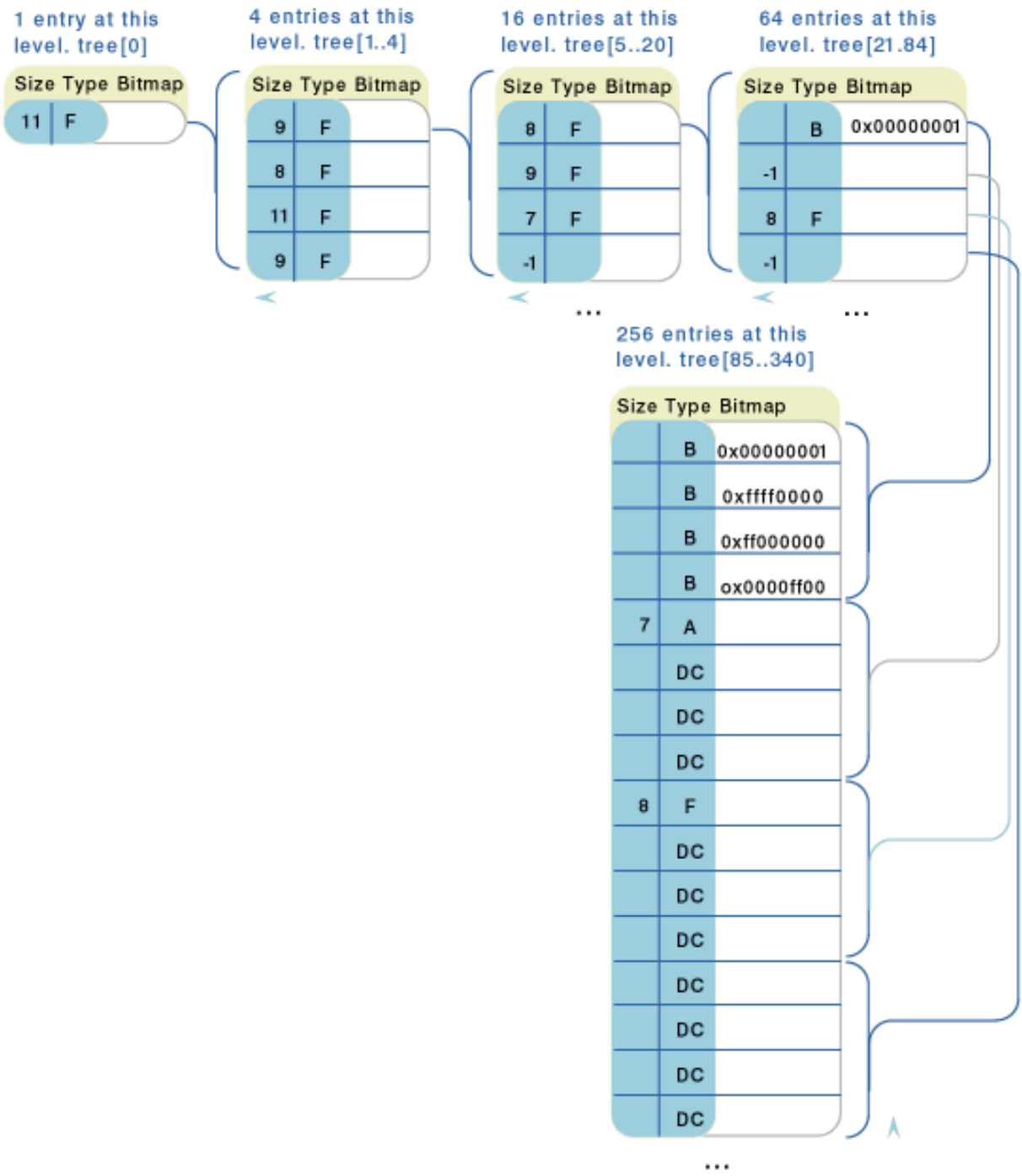
Figure 9 shows a small example of the encoded binary buddy representation for some allocations and deallocations.


FIGURE 9. Encoded Binary Buddy Example

The dmap structure contains a summary tree. Each of the other map levels also contain a summary tree. These trees improve the performance of finding large extents of free blocks. The summary information is sufficient for determining whether a dmap page has a sequence of free bits so that fruitless searches are avoided without referencing the dmap page.

Figure 10 shows the detail of a tree field from one dmap structure. Note this field in the dmap structure is a flat array, but it represents a tree as shown. The tree tracks the maximum number of contiguous blocks at each level. The bottom level of the tree, tree [21] through tree [84], maps to the encoded binary buddy representation in the working map. The other levels of the tree contain the maximum number of contiguous free blocks from four sections of the next lower level. The other levels of the Block Allocation Map would have a similar tree except the leaf level would contain 1024 elements. These elements would map to the encoded binary buddy representation of tree[0] for the following dmap pages.

FIGURE 10. Tree structure in dmap structure




If four elements to be combined are all of the "don't care" type then the combined entry is marked of size -1. The buddy entry for these items will take care of marking the correct state.

**Inode allocations**

With dynamic inode allocation, inode numbers no longer map directly to a specific logical disk block of the aggregate, therefore, data structures are needed to support three types of operations:

- Forward lookup: Given an inode number, find the on-disk inode. File lookup is a typical case.
- Reverse lookup: Given a partition disk block number (more precisely, an allocation group number), find near free inodes. This case occurs during new inode allocation, where JFS tries to locate an inode physically nearby the

chosen allocation group (so that, for example, files in the same sub directory have inodes that are all near each other).

- Free inode number lookup: To allocate a new inode extent, find the next 32 inodes which do not have a corresponding inode extent allocated for them. This case occurs when all currently allocated inodes are being used, when JFS needs inodes for an allocation group and has never allocated inodes before, or when there are no inodes free for an allocation group.

Note one subtle effect of dynamic inode allocation: inode numbers that are near each other are not necessarily near each other on disk: inode N+ 32 may be arbitrarily far from inode N. Conversely, inode numbers that are far apart may in fact can be close together on disk; it is theoretically possible for inode N+K to be next to inode N (even for the case K > 1).

**Inode Allocation Map**

The Inode Allocation Map solves the forward lookup problem. The aggregate and each fileset maintains an Inode Allocation Map, which is a dynamic array of Inode Allocation Groups(IAG). The IAG is the data for the Inode Allocation Map. For the aggregate the inodes mapped by the Inode Allocation Map are also known as the Aggregate Inode Table. For a fileset the inodes mapped by the Inode Allocation Map are also known as the File Inode Table.

Each IAG is 4K in size and describes 128 physical inode extents on the disk. Since each inode extent, contains 32 inodes, each IAG describes 4096 inodes. An IAG can exist anywhere in the aggregate. All of the inode extents for an IAG exist in one allocation group, the IAG is then tied to that AG until all of its inode extents are freed. At this point an inode extent could be allocated for it in any AG and then the IAG would be tied to that AG. The IAG is defined by `iag_t` structure which can be found in the jfs_imap.h .

```
/*
 *      inode allocation group page (per 4096 inodes of an AG)
 */
typedef struct {
        int64   agstart;        /* 8: starting block of ag              */
        int32   iagnum;         /* 4: inode allocation group number     */
        int32   inofreefwd;     /* 4: ag inode free list forward        */
        int32   inofreeback;    /* 4: ag inode free list back           */
        int32   extfreefwd;     /* 4: ag inode extent free list forward */
        int32   extfreeback;    /* 4: ag inode extent free list back    */
        int32   iagfree;        /* 4: iag free list                     */

        /* summary map: 1 bit per inode extent */
        int32   inosmap[SMAPSZ];/* 16: sum map of mapwords w/ free inodes;
                                 *      note: this indicates free and backed
                                 *      inodes, if the extent is not backed the
                                 *      value will be 1.  if the extent is
                                 *      backed but all inodes are being used the
                                 *      value will be 1.  if the extent is
                                 *      backed but at least one of the inodes is
                                 *      free the value will be 0.
                                 */
        int32   extsmap[SMAPSZ];/* 16: sum map of mapwords w/ free extents */
        int32   nfreeinos;      /* 4: number of free inodes             */
        int32   nfreeexts;      /* 4: number of free extents            */
                                /* (72)                                 */
        uint8   pad[1976];      /* 1976: pad to 2048 bytes              */
        /* allocation bit map: 1 bit per inode (0 - free, 1 - allocated)   */
        uint32  wmap[EXTSPERIAG];       /* 512: working allocation map     */
        uint32  pmap[EXTSPERIAG];       /* 512: persistent allocation map  */
        pxd_t   inoext[EXTSPERIAG];     /* 1024: inode extent addresses    */
} iag_t;                                /* (4096)                          */
```

The first 4K page of the Inode Allocation Map is a control page. This page contains summary information for the Inode Allocation Map. The definition for a `dinomap_t` structure can be found in the jfs_imap.h.

Abstractly, the Inode Allocation Map is a dynamically extensible array of the IAG structures:

```
struct iag inode_allocation_map [ 1.. N ];
```

Physically, the Inode Allocation Map is itself a file within the aggregate. The Aggregate Inode Allocation Map is described by the aggregate self-node. The Fileset Inode Allocation Map is described by Fileset Inode. Its pages are allocated and freed as necessary under standard B+ tree indexing. The key for the B+ tree is the byte offset of the IAG page.

JFS employs a commit strategy to insure that the control data is reliably updated. Reliable update means that consistent JFS structure and resource allocation state is maintained in the face of system failures. In order to ensure the Inode Allocation Maps are in consistent state it maintains two maps, the working map and the persistent map within each IAG. The working map records the current allocation state. The persistent map records the committed allocation state, consisting of the allocation state as found on the disk or described by records within the JFS log for committed JFS transactions.

Each bit in these maps describe whether the corresponding inode is free or allocated. A bit value of 0 represents a free inode and a value of 1 an allocated inode. Within each control section of an IAG there is a summary map which is used to improve performance of finding free inodes. The summary map maps to the working bitmap of the IAG. The summary map uses one bit to map for 32 contiguous bits of the working map. Each bit indicates either available inodes(0) or no available inodes (1) for the corresponding inodes it maps to. (If there is not an extent allocated then there are no available inodes for that inode summary map bit.)

An IAG also contains Inode Extent Descriptors which describe the corresponding inode extent. There are 128 of these per IAG. Within each control section of an IAG there is a summary map which is used to improve performance of finding free inode extents. The summary map uses one bit to map for each inode extent. A 0 indicates a free inode extent, while a 1 indicates an allocated inode extent.

Given an inode number, the Inode Allocation Map can be used to find the physical location of the inode by the following steps:

1. Find the IAG which describes this inode. Need to find the key (byte offset) to search in the B+ tree for the inode allocation map.

```
iag key = ((Inode number / Inodes per iag) * Inodes per iag) +  4096     (EQ 1)
```

2. Find which inode within the found IAG is being referenced. This can be used for indexing in the working and persistent maps of the IAG.

```
iag inode index = (Inode number) mod (Inodes per iag)                    (EQ 2)
```

3. Find the inode extent descriptor within the IAG which describes the inode extent containing the specified inode.

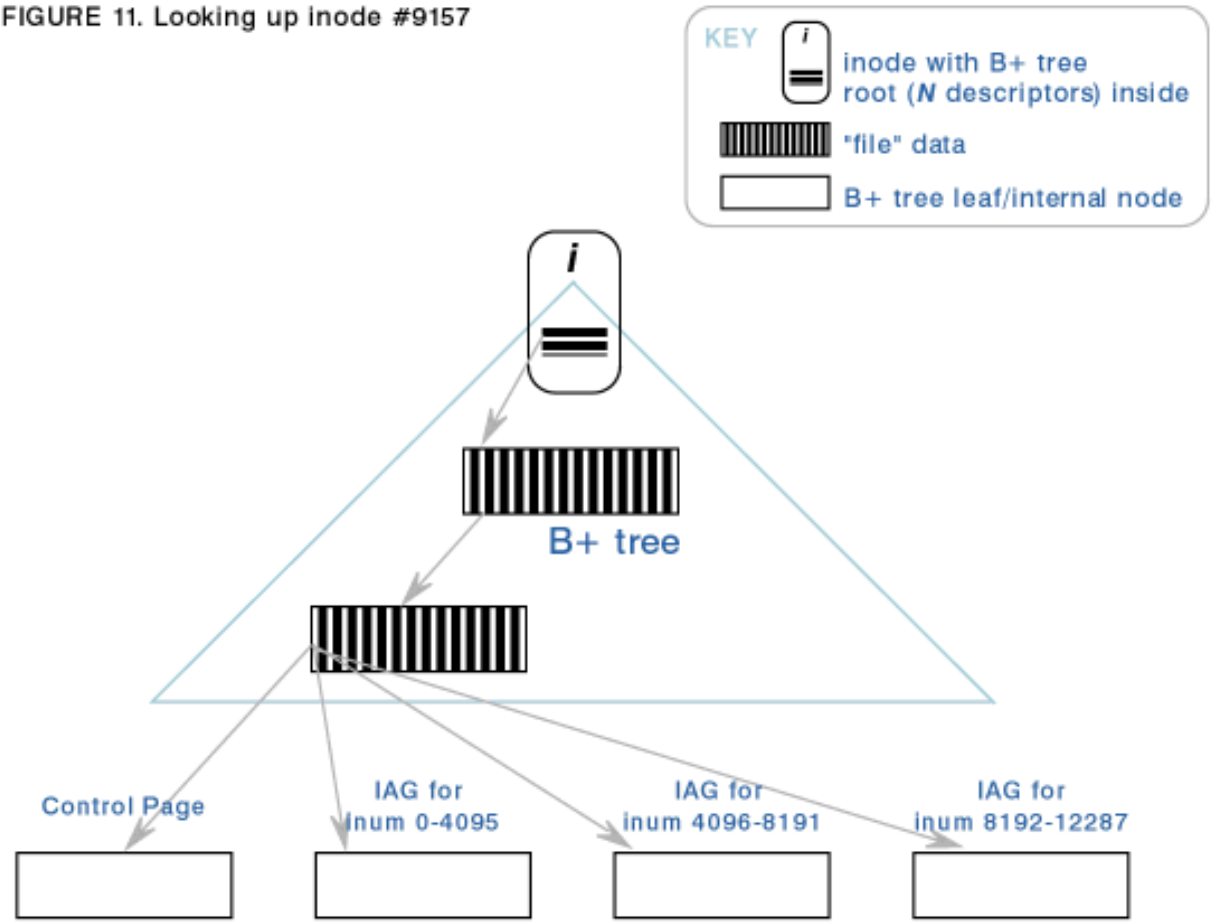
```
inode extent descriptor  = (iag inode index) / (Inode per inode extent)  (EQ 3)
```

4. The inode being sought is at the appropriate offset within the found inode extent.

```
inode offset = ((iag inode index) mod (Inodes per inode extent)
  * sizeof dinode)                                                       (EQ 4)
```

Figure 11 shows an example of looking up inode #9157. The Inode Allocation Map itself is described by the fileset's allocation map inode in the Aggregate Inode Table.

FIGURE 11. Looking up inode #9157




The inode number, # 9157, is converted to an offset by the formula shown previously:

```
iag key = ((inum / num_inodes_per_iag ) * (num_inodes_per_iag )) + 4096
                                    = ((9157 / 4096) * 4096 ) + 4096
                                    = 12288
iag inode index = inum mod num_inodes_per_iag
                                    = (9157 mod 4096 )
                                    = 965
inode extent descriptor = iag_inode_index / num_inodes_per_extent
                                    = 965 / 32
                                    = 30
inode offset     =  (iag_inode_index mod num_inodes_per_extent)
                                          * sizeof dinode
                                   = (965 mod 32) * 512
                                   = 5 * 512
                                   = 2560
```

To simplify JFS maintenance commands and to make it easier to understand the dynamics of the layout policies, the extents in a Inode Allocation Map file are always 4KB each.

When a new fileset is created one IAG must be allocated along with the first inode extent to handle the meta-data files of the fileset. (Namely the reserved inodes and the root directory inode).

**AG Free Inode List**

The AG Free Inode List solves the reverse lookup problem. In order to reduce the overhead of extending or truncating the aggregate JFS will set a maximum number of AGs allowed per aggregate. Therefore, there will be a fixed number of AG Free Inode List headers. The header for the list is in the control page of the Inode Allocation Map. The i th entry is the header for a doubly-linked list of all Inode Allocation Map entries (IAGs) with free inodes contained in the i th AG. The

IAG number is used as the index in the list. A -1 indicates the end of the list. Each IAG control section contains forward and backward pointers for the list.

Insertions for a particular AG list are done at the head of the list. An insertion can occur when a new inode extent is allocated or when an inode is deleted from an extent which was full. When all inodes extents for an IAG become full the IAG is removed from the list.

Figure 12 shows the layout of the AG Free Inode List. Notice the IAG in AG3 does not have any corresponding inode extents allocated. Therefore, these inodes do not show up in the AG Free Inode List.

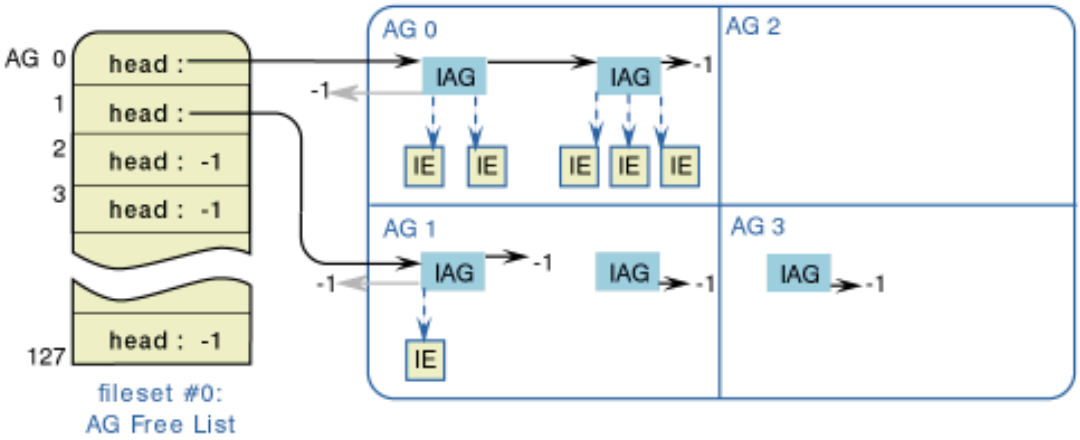



FIGURE 12. AG Free list

The table is not journaled; it can be repaired during recovery time by `logredo` or reconstructed by `fsck`. The definition for describing the AG Free list struct dinomap_t, is in jfs_imap.h file.

**AG Free Inode Extent List**

The AG Free Inode Extent List helps to solve the reverse lookup problem and the free inode number lookup problem. It allows JFS to find the next extent within an IAG for a particular AG which has not yet been backed to disk. (Which in affect tells JFS the free inode numbers.) Each fileset has its own AG Free Inode Extent List for each AG. In order to reduce the overhead of extending or truncating the aggregate JFS will set a maximum number of AGs allowed per aggregate. Therefore, there will be a fixed number of AG Free Inode Extent List headers. The header for this list is in the control page of the Inode Allocation Map. The ith entry is the header for a doubly-linked list of all Inode Allocation Map entries (IAGs) with free inode extents contained in the ith AG. The IAG number is used as the index in the list. A -1 indicates the end of the list. Each IAG control section contains forward and backward pointers for the list.

When all inodes in an extent are deleted the inode extent disk blocks are freed. When an inode extent for an IAG is deleted the IAG number is inserted as the head of the AG Free Inode Extent List. When a new IAG is created and an inode extent is allocated for it the IAG number is inserted at the head of the AG Free Inode Extent List. When all inode extents for an IAG are allocated the IAG is removed from the list. When all inode extents for an IAG are freed the IAG is removed from the list and added to the IAG Free List. When a new inode extent needs to be allocated for an AG, the first map entry is used from the head of the AG Free Extent List. Figure 12 shows the layout of the AG Free Inode Extent List. In this example the AG Free Inode Extent List looks the same as the AG Free Inode List.

The table is not journaled; it can be repaired during recovery time by `logredo` or reconstructed by `fsck`.

The current definition for the structure describing this table is in jfs_imap.h, `struct dinomap_t`.

**IAG Free List**

The IAG Free List solves the problem of the free inode number lookup. It allows JFS to find the IAG without any corresponding allocated inode extents. (This in effect tells JFS the free inode numbers). The aggregate has its own linked list and each fileset has its own linked list. This list provides the anchor for a linked list of IAGs. The IAG number is

used as the index in the list. A -1 indicates the end of the list. When all inodes in an extent are deleted, the inode extent disk blocks are freed. When all inodes are free for a particular IAG, the IAG number is inserted at the head of the IAG Free List. When a new inode extent needs to be allocated and there is not a IAG for the AG with free extents, the first map entry is removed from the head of the IAG Free List. Once allocated the inode extent allocation descriptors are never deleted. The address of the inode extent will be set to 0x0. Inodes from allocation group 3 in Figure 12 would be on this list.

For the aggregate the IAG Free List header is a field in the Aggregate self Inode. For each fileset the IAG Free List header is a field in the Fileset Allocation Map Inode. The list is not journaled; it can be repaired during recovery time by `logredo` or reconstructed by `fsck`.

The definition for describing the IAG Free list struct `inomap_t` is in the jfs_dinode.h file.

**IAG Free Next**
The IAG Free Next counter helps to solve the problem of the free inode number lookup. It allows JFS to find the iagnum for the next IAG which should be allocated. (Which in effects tells JFS the free inode number). The aggregate has its own counter and each fileset has its own counter. These counters are in the control page for the Inode Allocation Map. Once allocated, an IAG is never deleted.

**Fileset allocation inodes**
The Fileset Allocation Map Inodes in the Aggregate Inode Table are a special type of inode. Since they represent the fileset they are the "super-inode" for the fileset. They contain some fileset specific information in the top half of the inode instead of the normal inode data. It also tracks the location of the Fileset Inode Allocation Map in its B+ tree. The structure is defined by struct dinode, jfs_dinode.h file.

**File**
A file is represented by an inode containing the root of a B+ tree which describes the extents containing user data. The B+ tree is indexed by the offset of the extents.

**Symbolic link**
A symbolic link is represented by an inode with the `di_mode` field set to indicate a symbolic link. (S_IFLNK) The full path-name of the file being linked to is stored in-line in the inode if there is space. Otherwise, it will be stored as the data for this inode in an extent indexed by the B+ tree for the inode.

**Directory**
A Directory is a journaled meta-data file in JFS. A directory is composed of directory entries which indicate the objects contained in the directory. A directory entry links a name to an inode number. The specified inode describes the object with the specified name. In order to improve performance of locating a specific directory entry, a B+ tree sorted by name is used.

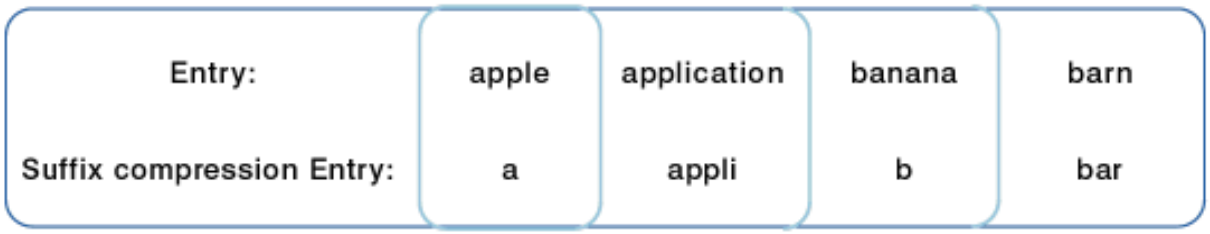
The directory inodes' `di_size` field represents just the leaf pages of the directory B+ tree. When the leaf node of the directory is contained within the inode, the `di_size` field is 256.

A directory will not contain specific entries for self(".") and parent (".."). Instead these will be represented in the inode itself. Self is the directory's own inode number. The parent will be a special field in the inode, `idotdot , struct dtroot_t`, jfs_dtree.h file.

The directory inode will contain the root of its B+ tree in a similar manner to a normal file. However this B+ tree will be keyed by name. The leaf nodes of a directory B+ tree will contain the directory entry and will be keyed from the complete name of the entry. The directory B+ tree will use suffix compression for the last internal nodes of the B+ tree. The rest of the internal nodes will use the same compressed suffix. Suffix compression truncates the name to just enough characters to distinguish the current entry from the previous entry.

Figure 13 shows an example of suffix compression.

FIGURE 13. Suffix compression




Since a B+ tree entry can be of varying size JFS needs a scheme to handle these entries. JFS wanted to avoid having to shift the entries in the page when deleting an entry, on average this would be 2K of data.

Figure 14 shows the elements of a directory B+ tree node:

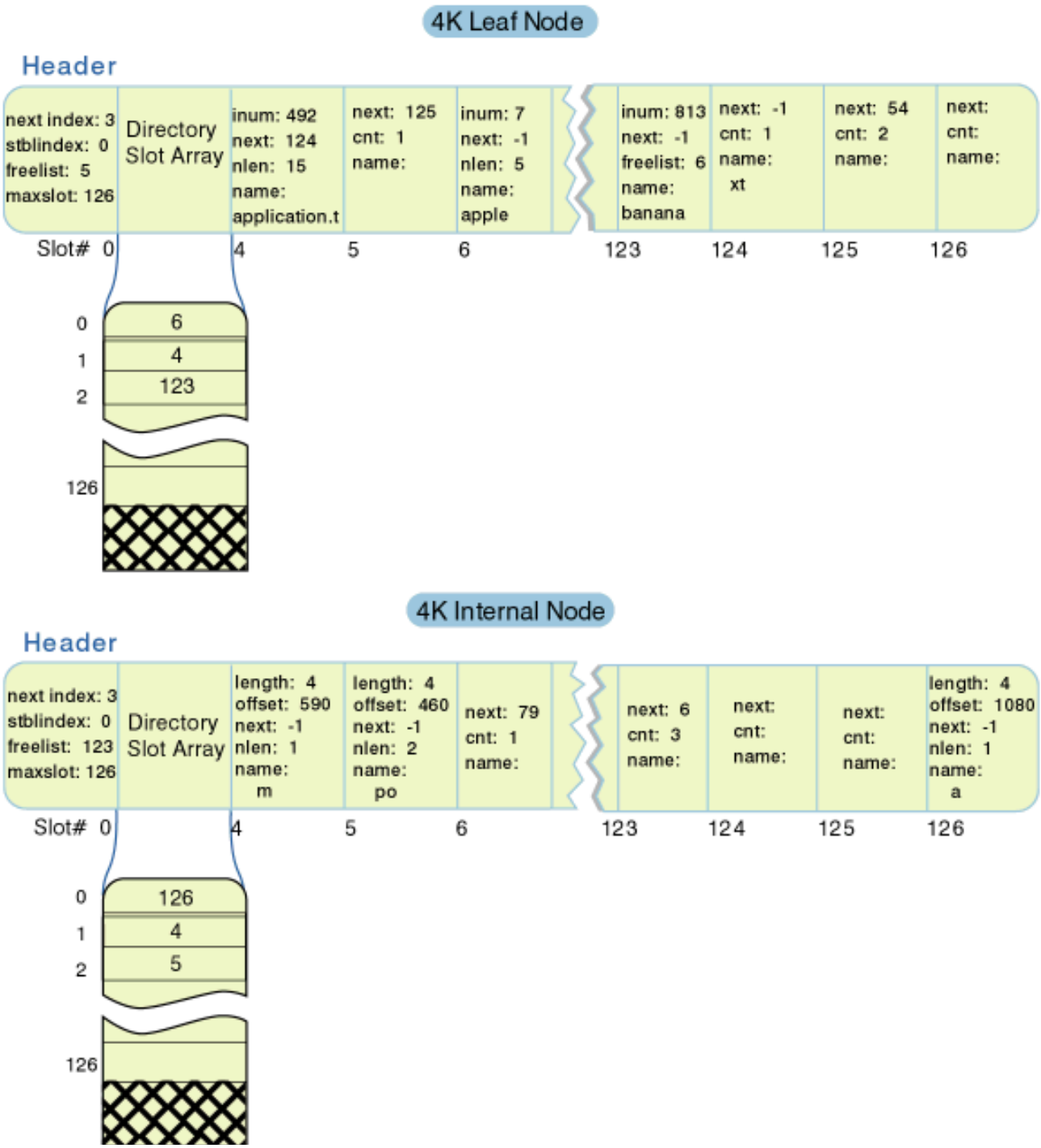


FIGURE 14. Directory B+-tree

- Fixed number of Directory Slots depending on the size of the node. These are the slots to be used for storing the directory slot array and the directory entries or router entries. A directory slot is always 32 bytes. The fixed size directory slots save JFS from having to shift when a directory entry is removed, thus avoiding internal fragmentation.
- A Directory B+ tree Header which describes the inode of the B+ tree. It contains a flag indicating whether the node is an internal or leaf node, and whether it is the root of the B+ tree. It also contains the block address of itself. The field `nextindex` tells which is the last entry used in the directory slot array. The field `stblindex` tells where the directory slot array starts. The field `freelist` contains the header to the free list of unused slots in the node.
- A Directory Slot Array which is a sorted array of indices to the directory slots that are currently in use. This array limits the amount of shifting necessary when directory entries are added or deleted. Since the array is much smaller than the entries themselves the array is shifted instead of having to shift the entries. A binary search can be used on this array to search for particular directory entries.
- A Directory B+ tree Slot Free List which helps to minimize internal fragmentation. The directory B+ tree header contains the header for the list and each free directory slot points to the next free slot in the list. The first free slot in a contiguous series of free slots will contain a count indicating how long the series is. This allows quick initialization of newly created directory B+ tree node.
- A Directory Entry which links a name to an inode number. A directory entry is contained in a directory slot of a leaf node. A directory entry can continue to additional slots if needed to contain the entire name. The field `next` in the directory entry will indicate if the entry continues to another entry. The majority of directory entries should fit into a single slot.
- A Router Entry used for routing the search of the directory B+ tree. A router entry is contained in a directory slot of an internal node. A router entry maps a suffix compressed router key to an extent containing an internal or leaf node of the next level of the directory B+ tree. A router entry can continue to additional slots if needed to contain the entire router key. The field next in the router entry will indicate if the entry continues to another entry. The majority of router entries should fit into a single slot.

An internal or a leaf node in the directory B+ tree is a 4K page. Since many directories are not very large this could result in wasted disk space for most directories. Therefore the initial leaf node for a directory will have the following allocation scheme:

1. Initial directory entries are stored in directory in-line data area.
2. When the in-line data area of the directory inode becomes full JFS allocates a leaf node the same size as the aggregate block size.
3. When that initial leaf node becomes full and the leaf node is not yet 4K double the current size. First attempt to double the extent in place, if there is not room to do this a new extent must be allocated and the data from the old extent must be copied to the new extent. The directory slot array will only have been big enough to reference enough slots for the smaller page so a new slot array will have to be created. Use the slots from the beginning of the newly allocated space for the larger array and copy the old array data to the new location. Update the header to point to this array and add the slots of the old array to the free list.
   Figure 15 depicts one level of growth of the directory.

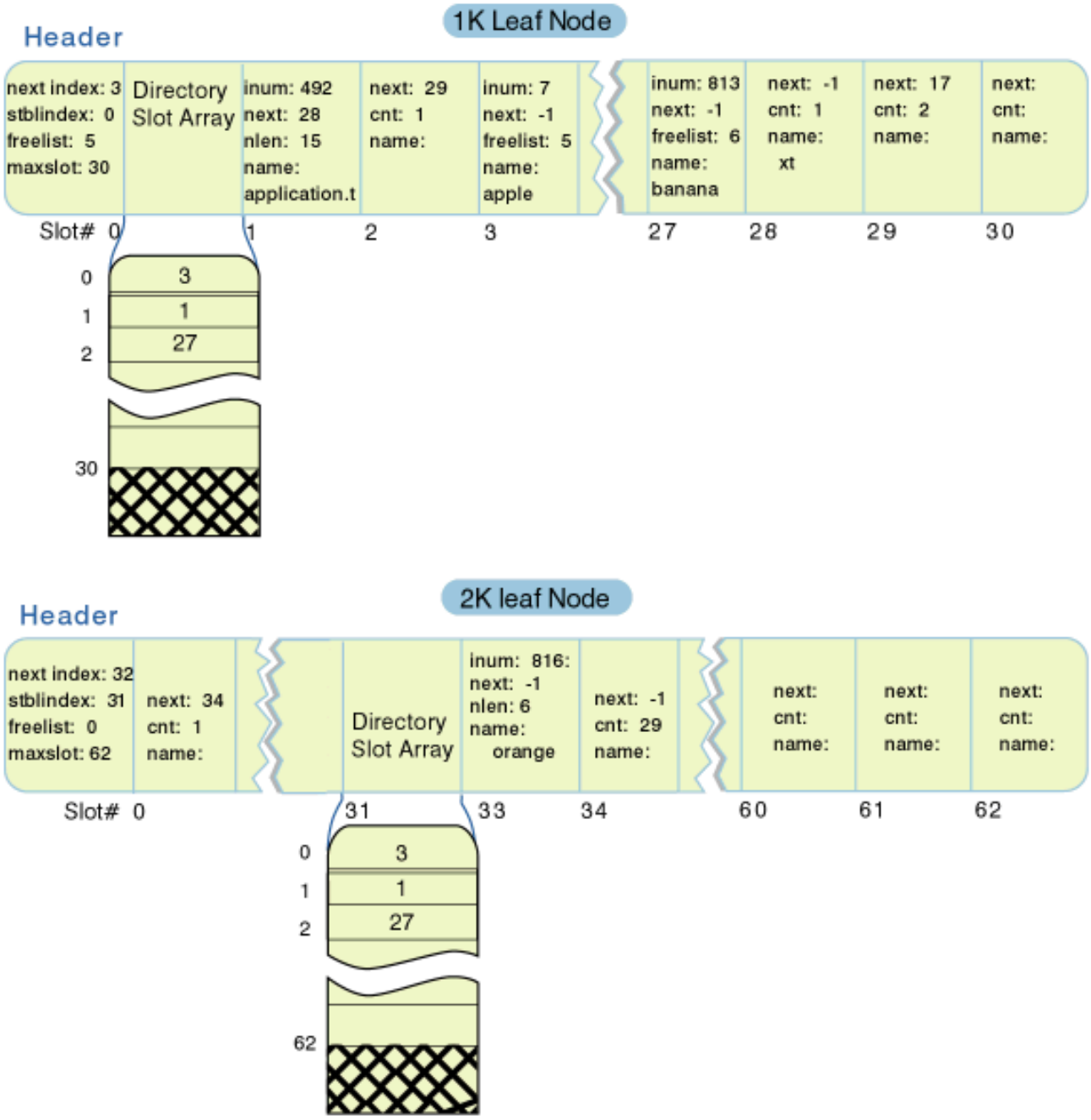



FIGURE 15. Growth of directory page

4. If the leaf node again becomes full and is still not 4K repeat step 3. Once the leaf node reaches 4K allocate a new leaf node. Every leaf node after the initial one will be allocated as 4K to start.
5. When all entries are free in a leaf page, the page will be removed from the B+ tree. The directory will shrink back into the inode only if all of the directory entries are deleted.

**Access Control List (ACL)**

Associated with every inode of JFS are various Access Control Lists (ACLs). ACLs can represent different items such as permissions, user identifiers, or group identifiers. The ACL fields are ignored for aggregate inodes.

Although there are no requirements on the ACL representation on disk and in memory, the "external" representation as seen outside DFS is fixed. The only limit on ACL size is that its external representation must fit within an 8192-byte `dfs_acl` structure.

Any JFS object can be associated with an ACL which governs the discretionary access to that object; this ACL is referred to as the regular ACL. Directory objects may additionally have two associated optional ACLs that are used at object

creation time; the initial directory ACL, and the initial file ACL. If present, the initial ACL is applied to any files created in that directory.

The ACL architecture does not specify how ACLs should be stored. However, it suggests the ACL fields somehow identify or name their auxiliary object such that sharing within a fileset can be detected via a simple equality check. To handle this JFS will have a file (the ACL file) in each fileset to store ACLs of the fileset; fileset inode 1 will represent this file. Each inode in the fileset will store an index into the ACL file.

The ACL file needs to have a bitmap to locate the free regions to store the ACLs. The ACL file will have a 4K bitmap followed by 8M of ACL entries, repeated as necessary. One bit in the bitmap will represent 256 bytes of contiguous disk space; the bitmap does not describe itself.

Figure 16 shows the layout of the ACL file.

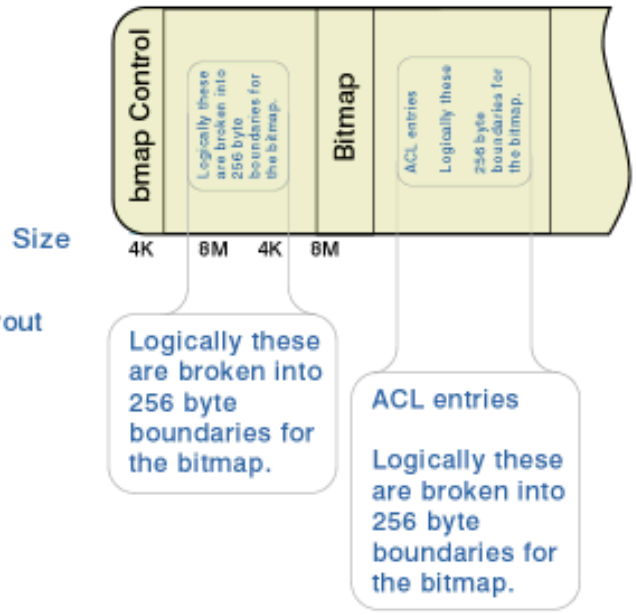



FIGURE 16. ACL File Layout

The ACL file data is journaled.

**Extended Attribute (EA)**
An Extended Attribute is a generic storage and access mechanism for data attached to JFS objects. EAs are stored contiguously in an Extended Attribute Space (EAS) as defined by the EA descriptor in the inode for the JFS object. The EA descriptor is simply an extent descriptor as defined in jfs_types.h, `struct dxd_t`.

An EA can be stored either in the inode or in a separate extent. The flags field of the EA descriptor will indicate which way it is stored. Since this space could also be used for additional xad entries for the xtree for the file, the di_mode field of the inode will indicate if this space is available. If the INLINEEA bit is set the space is available.

If the EA is stored in the inode the offset and length fields of the EA descriptor will be ignored. The size of the EA descriptor will indicate the number of bytes of the data.

If the EA is stored in an extent the EA descriptor will describe this extent. JFS does not expect EA data to be very large so JFS will not support more than one extent of EA data per inode.

An EA entry will contain both the name of the EA and its value. To access an individual EA, JFS will simply search the EA data linearly for the item.

The EA data is not journaled, however it is written synchronously (meaning it will always either be the old data or the new data, but never a partially written data). JFS will journal where the EA data is located. The in-line EA data is journaled.

**Streams**

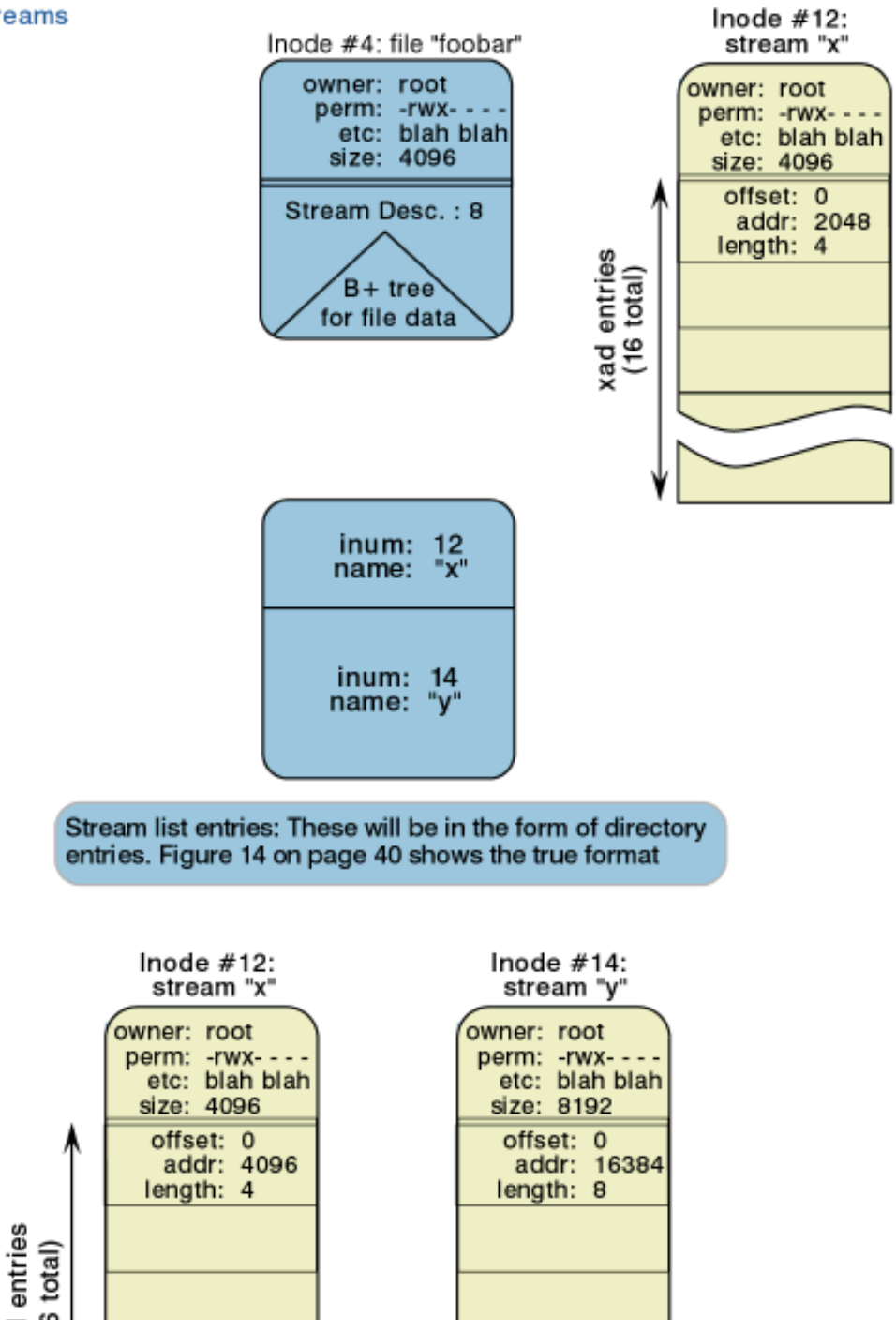
A stream is used to attach data to a file or directory. This additional data is similar to directory data since it can be referred to by name. Streams will not be supported in the first release but are discussed here to demonstrate completeness of the meta-data structures.

The second quadrant of the disk inode will have a field for the stream descriptor. Since the number of streams attached to an object can vary, the stream descriptor will be an inode number which will allow the streams to grow or shrink. The data pointed to by the stream descriptor inode will be referred to as the stream list.

Streams do not have extended attributes associated with them, therefore the inodes needed for streams will never use the last quadrant of the extended attributes. Instead it will be used for additional stream entries. The data for the B+ tree looks just like directory entries. Each stream will in turn have its own inode which addresses the data blocks where the stream data will be stored.

Figure 17 shows streams; they will not be journaled.

FIGURE 17. Streams



### Aggregate with a fileset

Figure 18 shows an aggregate containing a fileset.

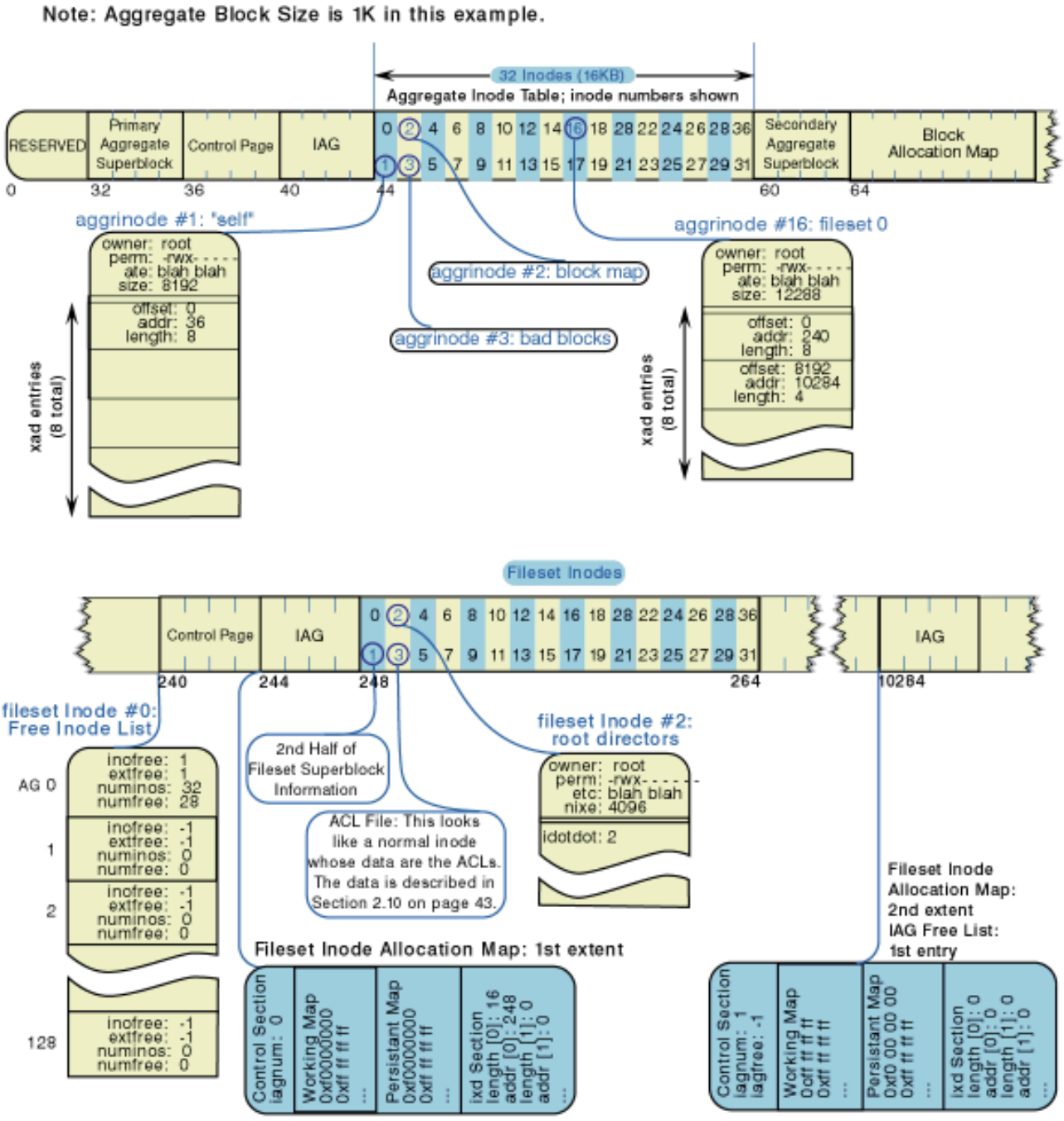



FIGURE 18. Aggregate containing a fileset

### Summary

The JFS team's most important goal was to create a reliable, high-performance file system. This article discussed the mechanisms that were used for scalability, reliability, and performance using the on-disk layout structures of JFS. Also discussed in detail was how JFS uses B+ trees throughout the file system to increase file system operations.

The JFS team is making great progress in moving JFS to Linux. To get involved, visit our JFS project page on developerWorks.

**Resources**

- JFS source code
- JFS Overview
- IBM makes JFS technology available for Linux

**About the authors**

Steve Best works in the Software Solutions & Strategy Division of IBM in Austin, Texas, as a member of the Linux Technology Center. Steve has worked on operating system development in the areas of the file system, internationalization, and security. Steve is currently working on the port of JFS to Linux. He can be reached at sbest@us.ibm.com.

Dave Kleikamp is a member of the Linux Technology Center in the Software Solutions & Strategy Division of IBM in Austin, Texas. Dave has previously worked as the technical lead on the JFS filesystem for OS/2 and as a debugging specialist for AIX. Dave is currently working on the open-source port of JFS to Linux. He can be reached at shaggy@us.ibm.com.