**Ext4**

The EXT4 filesystem primarily improves performance, reliability, and capacity. To improve reliability, metadata and journal checksums were added. To meet various mission-critical requirements, the filesystem timestamps were improved with the addition of intervals down to nanoseconds. The addition of two high-order bits in the timestamp field defers the Year 2038 problem until 2446—for EXT4 filesystems, at least.

In EXT4, data allocation was changed from fixed blocks to extents. An extent is described by its starting and ending place on the hard drive. This makes it possible to describe very long, physically contiguous files in a single inode pointer entry, which can significantly reduce the number of pointers required to describe the location of all the data in larger files. Other allocation strategies have been implemented in EXT4 to further reduce fragmentation.

EXT4 reduces fragmentation by scattering newly created files across the disk so that they are not bunched up in one location at the beginning of the disk, as many early PC filesystems did. The file-allocation algorithms attempt to spread the files as evenly as possible among the cylinder groups and, when fragmentation is necessary, to keep the discontinuous file extents as close as possible to others in the same file to minimize head seek and rotational latency as much as possible. Additional strategies are used to pre-allocate extra disk space when a new file is created or when an existing file is extended. This helps to ensure that extending the file will not automatically result in its becoming fragmented. New files are never allocated immediately after existing files, which also prevents fragmentation of the existing files.

Aside from the actual location of the data on the disk, EXT4 uses functional strategies, such as delayed allocation, to allow the filesystem to collect all the data being written to the disk before allocating space to it. This can improve the likelihood that the data space will be contiguous.

Older EXT filesystems, such as EXT2 and EXT3, can be mounted as EXT4 to make some minor performance gains. Unfortunately, this requires turning off some of the important new features of EXT4, so I recommend against this.

EXT4 has been the default filesystem for Fedora since Fedora 14. An EXT3 filesystem can be upgraded to EXT4 using the procedure described in the Fedora documentation, however its performance will still suffer due to residual EXT3 metadata structures. The best method for upgrading to EXT4 from EXT3 is to back up all the data on the target filesystem partition, use the mkfs command to write an empty EXT4 filesystem to the partition, and then restore all the data from the backup.

**Inode**

The inode, described previously, is a key component of the metadata in EXT filesystems. Figure 2 shows the relationship between the inode and the data stored on the hard drive. This diagram is the directory and inode for a single file which, in this case, may be highly fragmented. The EXT filesystems work actively to reduce fragmentation, so it is very unlikely you will ever see a file with this many indirect data blocks or extents. In fact, as you will see below, fragmentation is extremely low in EXT filesystems, so most inodes will use only one or two direct data pointers and none of the indirect pointers.

ext4 is the default file system for many Linux distributions including [Debian](https://en.wikipedia.org/wiki/Debian) and [Ubuntu](https://en.wikipedia.org/wiki/Ubuntu).[[*citation needed*](https://en.wikipedia.org/wiki/Wikipedia:Citation_needed)]

The basic disk layout of ext4 is similar to the one of ext3. Following a traditional Unix filesystem design, each file and directory in the filesystem is described by an object called inode. Inode contains information about its type (file, directory, symbolic link, etc.), file size, access rights, owner, and where data is stored. All inodes in the filesystem have the same size which is configurable when creating the filesystem. The inode size is a power of two between 128 bytes and the filesystem block size. Current default inode size is 256 bytes. Inode size above 128 bytes is needed to store inode timestamps with a sub-second precision and to allow storage of extended attributes (such as ACLs) in the inode (extended attributes which do not fit into the inode are stored in a special block).

**Inode Table**

You store your information in a file, and the operating system stores the information about a file in an inode(sometimes called as an inode number).

Whenever a user or a program needs access to a file, the operating system first searches for the exact and unique inode (inode number), in a table called as an inode table. In fact the program or the user who needs access to a file, reaches the file with the help of the inode number found from the inode table.

**Extents**

Extents replace the traditional block mapping scheme used by ext2 and ext3. An extent is a range of contiguous physical blocks, improving large-file performance and reducing fragmentation. A single extent in ext4 can map up to 128 MiB of contiguous space with a 4 KiB block size.[4] There can be four extents stored directly in the inode. When there are more than four extents to a file, the rest of the extents are indexed in a tree.

An extent is a contiguous area of storage reserved for a file in a file system, represented as a range of block numbers, or tracks on Count key data devices. A file can consist of zero or more extents; one file fragment requires one extent. The direct benefit is in storing each range compactly as two numbers, instead of canonically storing every block number in the range.[1] Also, extent allocation results in less file fragmentation.

Extent-based file systems can also eliminate most of the metadata overhead of large files that would traditionally be taken up by the block-allocation tree. But because the savings are small compared to the amount of stored data (for all file sizes in general) but make up a large portion of the metadata (for large files), the overall benefits in storage efficiency and performance are slight.

An extent is a continuous run of physical blocks carrying data for a continuous run of logical file blocks. So for example a single extent describes that data from logical blocks 3 − 10 of file foo are located in physical blocks 2101 − 2108. Such description is much more space efficient than simple block pointers used by ext3 on an unfragmented filesystem as a length of an extent can be upto 2 15 blocks in ext4.

**Fragmentation**

File fragmentation is a term that describes a group of files that are scattered throughout a hard drive platter instead of one continuous location. Fragmentation is caused when information is deleted from a hard drive and small gaps are left behind to be filled by new data. As new data is saved to the computer, it is placed in these gaps. If the gaps are too small, the remainder of what needs to be saved is stored in other available gaps.

Fragmentation causes slow access time because read/write head accessing the data must find all fragments of a file before it can be opened or executed. If the hard drive has to do this many files each time it's opened, it can decrease the computer's performance.

**B Tree**

To allow fast lookup which physical block describes a given logical block, extents are kept in a b-tree indexed by the starting logical block number of an extent. The root node of the b-tree is stored in the inode itself and can contain six extents (or indices to lower levels). If the extents do not fit into the inode, other nodes of the b-tree are stored in blocks allocated for this purpose. When a node of the b-tree gets empty, the block is released. Currently, optimizations of the b-tree such as merging of index blocks or reduction of a depth are not implemented.

**Format of directories**

Format of directories is exactly the same in ext4 as it is in ext3. So let us just quickly remind it. Following a traditional ext3 design, directory is stored as a file which contains directory entries instead of data. Each directory entry contains a name of a file, directory, symbolic link, etc. and a number of inode which contains details about it (see Figure 2.1). To speedup searching in large directories, an htree feature can be enabled. With this feature, a search tree of directory entries is maintained. The nodes of the tree are hidden in a special directory entries and thus this feature is fully backward compatible.

An HTree is a specialized tree data structure for directory indexing, similar to a B-tree. They are constant depth of either one or two levels, have a high fanout factor, use a hash of the filename, and do not require balancing.[1] The HTree algorithm is distinguished from standard B-tree methods by its treatment of hash collisions, which may overflow across multiple leaf and index blocks. HTree indexes are used in the ext3 and ext4 Linux filesystems, and were incorporated into the Linux kernel around 2.5.40.[2] HTree indexing improved the scalability of Linux ext2 based filesystems from a practical limit of a few thousand files, into the range of tens of millions of files per directory.

**Block-group**

Each block group contains a copy of the superblock and block group descriptor table, and all block groups contain a block bitmap, an inode bitmap, an inode table, and finally the actual data blocks.

The superblock contains important information that is crucial to the booting of the operating system. Thus backup copies are made in multiple block groups in the file system. However, typically only the first copy of it, which is found at the first block of the file system, is used in the booting.

**Journaling**

Journaling filesystems write metadata (i.e., data about files and directories) into the journal that is flushed to the HDD before each command returns. In the event of a system crash, a given set of updates may have either been fully committed to the filesystem (i.e., written to the HDD), in which case there is no problem, or the updates will have been marked as not yet fully committed, in which case the system will read the journal, which can be rolled up to the most recent point of data consistency.

**Allocation algorithms**

Another key difference of ext4 to ext3 is a different strategy of block allocation. Ext4 supports so called delayed allocation. The principle of this feature is that during a write system call, filesystem just estimates how many new blocks it will need to allocate. It reserves this amount of blocks but does not allocate any particular blocks yet. The real allocation of blocks is done when memory management decides to flush written data from memory to disk. This delaying of allocation allows for better decision about the final size of the file and thus for better allocation decisions: • More blocks can be allocated at once thus reducing fragmentation and CPU utilization. • Writes happening at random offsets (possibly also by different threads) are joined into one monotone writeout sequence. • In case of short-lived temporary files, filesystem can avoid allocating any blocks at all. A disadvantage of this feature is that in case of a crash, larger amount of data is lost compared to standard ordered mode without delayed allocation. With a standard system configuration, writeout of data happens once every 30 seconds and it need not write all of the written data. On the other hand in ordered mode without delayed allocation, data is guaranteed to hit the disk after 5 seconds from the write

**Sources**

<https://pdfs.semanticscholar.org/f834/a990d2e1d4f96c857e953904cf893d45b3c6.pdf>

<https://en.wikipedia.org/wiki/Ext4>