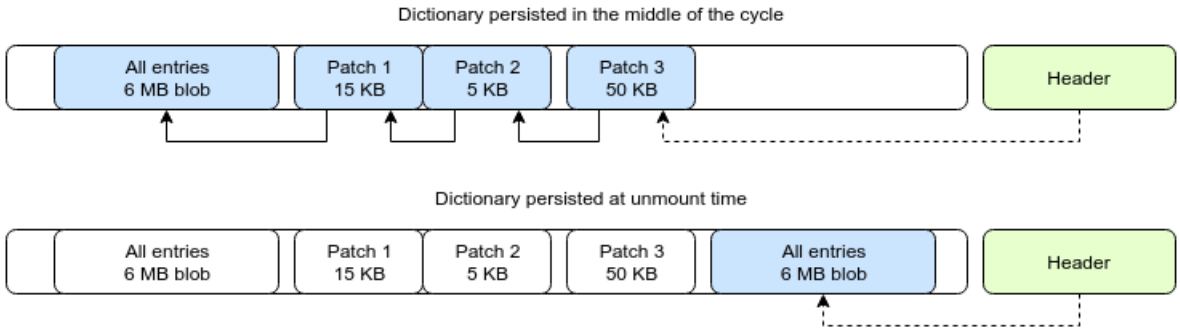
B-trees are not efficient.

There is a trend among filesystem designs towards using both modified in-place [Mathur07] and copy on write B-trees [Rodeh12] [ZFS]. This trend might be easily explained by common expectation that filesystems should be able to handle huge numbers of files. Popular filesystems even explicitly advertise their ability to scale among main points why to choose them over the competition. B-trees are a good approach if a huge amount of keys is expected. Trees scale asymptotically with logarithmic complexity which means even billions of entries can be stored with only a few disk accesses needed to reach any given entry. This asymptotic behavior seems to have mislead everyone. It would be justified to use B-trees if billions of files were expected to be stored but that is a false assumption, at least in general case. Most computers hold on the order of 100'000 files [Agraw07]. That amount of dictionary entries can be stored and accessed more efficiently.

Second reason to use B-trees in the past is that memory needed to query and update trees is small, comparable to number of nodes from root to leaf. Computers of the past often had little memory, [McKusi84] had maybe 8 MB of memory. However, modern desktop computers have several gigabytes of memory and servers have tens of gigabytes of memory. Conserving memory is not justified anymore. Memory constrained devices like smartphones constitute an exception but that does not negate benefits on desktops and servers.

Consider the three approaches described below, depending on how many entries we expect to hold at any given time. Respective capacities will be tweaked in order to make performance comparable. As in previous section, we assume hard disks can sustain ~120 MB/s throughput and ~10 ms seek time on average.

First approach, entire dictionary is stored in a single extent. At mount time, entire dictionary is loaded from disk in one sweep, kept in memory in entirety and entire time and regularly stored in entirety to disk as copy on write in one sweep. 1’000’000 entries dictionary would take 23 MB of disk space, assuming 64 bit keys and 128 bit values, and even less space if VarInt encoding was used. This amount of extents should be enough to store the files found on a typical desktop, which is 100K files of mean size 190 KB as found by [Agraw07]. Even better, small patches containing sets of changed entries can be stored to disk every few seconds and entire dictionary can be stored only every few minutes. If filesystem gets unmounted successfully, only the final chunk containing the entire dictionary is going to be loaded at mount time, otherwise some few dozen patches need to be loaded from disk and combined, however they were all stored in one disk segment anyway so one disk sweep is needed. Entire dictionary is being cached so all key lookups are instantaneous. Concurrency is a non issue since read operations are always diskless and instantaneous, and write operations are buffered. Amount of extents can be minimized by deferred allocation and regular compaction of disk segments. Following diagram shows deallocated space in white.



Second approach, B-trees as the contemporary alternative. Capacity is a function of tree height which is equivalent to a number of disk seeks. Assuming the root node is always in memory and nodes have 1480 entries (64 KB with 36 byte values and 64 bit pointers) then 1 seek gives a capacity of $2.2x10^6$ entries, 2 seeks give $3.2x10^9$ entries, 3 seeks gives $4.7x10^{12}$ entries, 4 seeks gives $7x10^{15}$ entries, and so on. Bitmaps can used to track unused space in allocated segments, leading to reusing of nodes or compaction of entire segments. This approach has its obvious benefits but one has to remember that most desktops have between 100K and 200K files. At this amount of entries, B-trees are not necessary and perform worse than a dictionary kept in memory, the first, diskless approach.

This argument is basically the same as for inode extent. If extent dictionary is considered as a metadata of some special file then same theoretical argument can be applied. Importantly, these two approaches are not mutually exclusive. When entire dictionary grows over a limit the entries can be easily migrated into a B-tree, and when the tree becomes tiny then entries can be migrated to an entire dictionary. The filesystem can keep switching between these two.

Block allocation is not efficient.

Since the times of Fast File-System and up to Ext4, disk space was divided into blocks but grouped into clusters. There are few heuristics when assigning blocks to files that can increase overall performance. For one, same file can have blocks allocated in a row continously or at least in close proximity. Large files are especially demanding of continous allocation. Secondly, files that are supposed to be accessed together, for example if created by same application, can be allocated next to each other. This approach seems to be sub-optimal. When space gets allocated at block resolution, fragmentation of both files and free space grows seemingly without bounds. At some point it becomes very difficult to find a large contiguous area while there is a lot of small free chunks. Problem is that seek time over small distances is quite significant, due to the settling phase. The right approach would be to keep actively countering free space fragmentation and keep defragmenting files soon after closing in the background.

Log-structured filesystem invented by [Rosenblum91] solves the problem by allocating disk space from one huge circular log, while regularly defragmenting files by moving files from the other end of the log and compacting them at same time. This approach has its own problems, that already continuous files, large or small, still need to be moved.

Segmented log-structure as described in this design aims to achieve the best of both worlds. Entire disk gets divided into segments which can then be differently divided into extents. After some extents were deallocated or even internally deallocated, the segment is read in one sweep, compacted internally and possibly recompressed, and then stored to disk in one sweep into a new segment. This approach has a problem of having a lot of segments that are half compact, as investigated by Rosenblum and Ousterhout. This is to be solved by redistributing extents so that there is not a 1-to-1 mapping of segments, but many partially filled segments turn into a few segments that are full. This means extents from one segment can end up split among different destination segments or even be split into separate extents.