Ideas and Observations
Foundations for a versatile filesystem

Arkadiusz Bulski

Abstract

Cameleonica is a prototype of a highly versatile filesystem. This paper describes a wide range of concepts that could but have not been incorporated into popular filesystem designs. Both potential benefits and reasons why these features were not popularized earlier are presented. Expectations of their performance are then justified.

Introduction

When reading papers that describe historical filesystem designs from the golden times of Unix development, one might come to a conclusion that evolution of filesystem designs was very incremental in nature and much effort was put into not making too many dramatical changes at any one time.

[work in progress]

New ideas

Inodes are not efficient.

Since the early days, filesystems were designed under a principle that data structures have to be broken down into fixed size blocks. This seems to have been due to the fact that hard disks require read and write operations to be carried out on 512 byte long, discrete blocks of data. In traditional designs most operations can be done by processing one block at a time. Computers of the past often had little memory [McKusi84] and caching complete sets of metadata was not feasible. Also changing metadata often ended up in writing just one block, which might have been seen as a compelling reason to use this approach. Furthermore hard disks guarantee that any given block is written atomically [Twee00]. All these reasons contributed to a decision that data structures had to be broken down into blocks.

To fully describe a single file, we need an amount of metadata that almost always takes space of more than one block. Traditionally one main block (called inode) would hold most important data, several blocks would point to where actual content is located, some blocks would keep directory entries in case of a directory, and so on. For a large 1 GB file for example, assuming 4 KB blocks and 64 bit pointers, at least 512 blocks would be needed. Certain operations such as copying or deleting file, or browsing directory would necessarily require all metadata blocks to be read.

Consider now a range of possibilities, where all metadata blocks are either allocated in one continuous range (called an extent), are divided into subsets, or are divided into individual blocks. First approach we shall call the *extent approach* and it requires that all blocks are read from an extent at once and after changes are made all blocks are stored to an extent at once. Last approach we shall call *block approach.* It is prevalent in modern filesystems. Space is allocated conservatively, one block at a time eventually leading to fragmentation of metadata.

Block approach seems efficient at first glance because changing one detail should require only one block to be written to disk. This kind of reasoning is flawed because we do not account for future operations. We should consider amortizing performance over entire lifespan of a file, not assume that any one operation must be carried out as fast as possible independently of other further operations.

It could be argued that first approach, where all metadata is always loaded and stored in one sweep is better in every usage scenario. To show that, we need to recognize that magnetic hard disks have quite skewed performance characteristics. Flash based disks are precluded due to reasons explained elsewhere. Representative hard disk is capable of ~120 MB/s of sustained throughput and ~10 ms seek time to random location on average. Quick calculation shows that reading or writing an extent smaller than 1.2 MB is faster than literally one seek, on average.

This observation may seem counter intuitive. Filesystem designs so far seem to be based on the assumption that data needs to be processed in smallest chunks possible and only way to go is to process as small amount of them. It seems that the opposite approach of processing bigger chunks never gained traction. In traditional designs [OSTEP] directories put entries into individual data blocks, files put data locations into individual indirect blocks, and so on. In modern designs [Rodeh12] [ZFS] copy on write B-trees are replacing indirect blocks but data is still being divided into small chunks. The issue being described here has not gone away.

To show that extent approach is better we will consider total time spent on all operations throughout most of the lifetime of a file. Only metadata is counted towards the time spent on read and write operations. Parent directory metadata is also excluded. At time zero we assume that all file content and metadata is already stored on disk. We let a certain number of cycles of the file being opened, accessed several times and then closed. After each opening some metadata blocks are accessed, either read or written. We also assume that cached blocks are forgotten between cycles due to assumed long time intervals between cycles. Blocks are assumed to be 4 KB size. Suffixes like K M refer to thousands and millions of bytes.

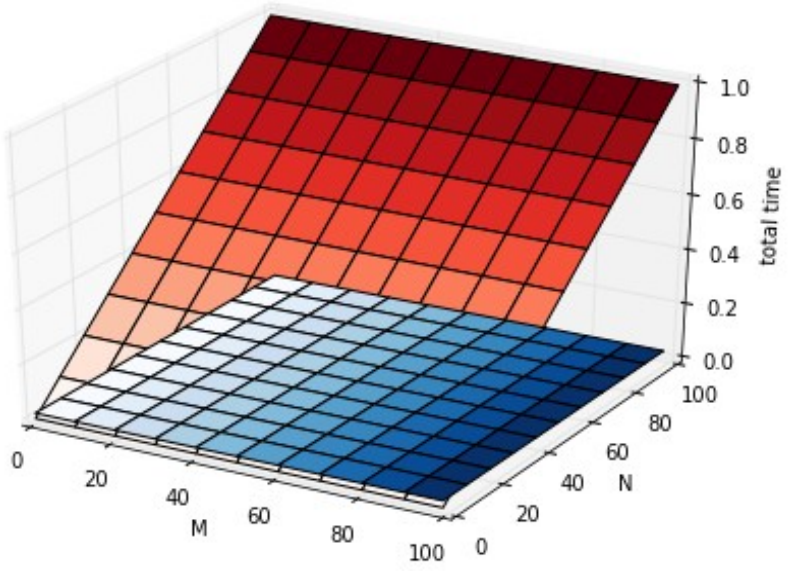
Reading or writing N blocks would take (assuming sequential and random pattern, referring to extent and block approach respectively):

$$R_1(N) = 0.010 + N \cdot 4\,K/120\,M = 0.010 + N \cdot 0.00003$$
$$R_N(N) = N \cdot (0.010 + 4\,K/120\,M) = N \cdot 0.01003$$

We conceive a simulation with three independent variables, N is number of blocks accessed in each cycle, M is number of blocks total, and B is number of cycles. Only a subset of blocks is accessed during a cycle so N<M. Therefore total time would be (extent and block approach respectively):

$$T_1(N, M, B) = \sum_{i=1}^{B} R_1(M) + R_1(M)$$
$$T_N(N, M, B) = \sum_{i=1}^{B} R_N(N)$$

Plots show total time of extent approach (blue) and block approach (red) for least and most accessed, and least and most sizable files.



Let us look at the cases where extent approach loses in comparison. Total times can be compared through an inequality. Notice that number of cycles cancels out.

$$T_N < T_1$$
$$B \cdot R_N(N) < B \cdot 2 \cdot R_1(M)$$
$$0.01003 \cdot N < 2 \cdot (0.01 + 0.00003 \cdot M)$$

$$N < 1.994 + 0.006 \cdot M$$
$$M > 166 \cdot N - 332$$

Last two inequalities show limits on how much metadata could be accessed during one cycle before block approach starts to lose advantage. Asymptotically, less than 1 in 166 metadata blocks could be accessed. For smallest files the limit is 2 blocks.

Remember when earlier it was mentioned that a 1 GB file would require at least 512 metadata blocks? According to the limits above, after opening it we could access at most 5 metadata blocks before block approach would lose advantage, and this includes the inode block. Is this really the kind of workload we are aiming to support?

When we store individual blocks we gain on transfer time related to small block size but we pay for it with seek time the next time we read said blocks. Disk characteristics make this exchange totally unfair. One seek takes as much time as transferring 1.2 MB which can for all practical purposes fit complete metadata of any imaginable file, ever.

Additionally the extent approach has the benefit that one checksum is enough to verify integrity of all metadata while the block approach would use as many checksums as

there are blocks.

Finally it should be admitted that the theoretical model above assumes most naive implementation of block approach where blocks are allocated individually from entire disk (independently from uniform distribution). It would be possible to allocate blocks close to each other whenever possible, increasing so called *spatial locality*. Orlov allocator is the mechanism meant to achieve exactly that. However it would arguably make the block approach only so much better in comparison and it is doubtful that entire argument would get trumped. Constructing a theoretical model that would include dependent allocation is not doable. Distribution parameters are not known.

B-trees are not efficient.

We can see a trend among filesystem designs towards using both modified in place and copy on write B-trees [Rodeh12] [ZFS]. This trend might be easily explained by common expectation that filesystems should be able to handle billions 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 we expect huge amounts of keys to be stored. B-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 everybody. It would be justified to use B-trees if we expected billions of files to be stored but that is a false assumption at least in general case. Most computers hold something on the order of 100'000 files. See [Agraw07] for representative statistics. That amount of dictionary entries can be stored more efficiently by other means.

Second reason to use B-trees is that memory needed to query and update trees is small, comparable to number of blocks from root to leaf. Modern desktop computers have several gigabytes of memory nowadays and servers have even more. Conserving memory is not justified. Embedded devices like phones and wristwatches are precluded.

Consider a hybrid or rather a transitional approach. Initially all entries are loaded from one extent, kept in memory in entirety and the whole time, and occasionally stored to disk as one extent. If at some point the amount of entries grows over a certain threshold, we commence a transition to a B-tree representation, we relocate all entries (already in memory) to tree nodes, and store all tree nodes to disk in one sweep. The threshold can be chosen low enough so storing entire collection in one sweep is faster than analogous B-tree operation (few disk seeks). After transition operations are carried out on a B-tree representation which is the compared alternative. If changes are being accumulated over periods of time then comparison is even more favorable. Also intents (patches) can be stored to disk to persist individual operations instead of storing the entire dictionary every time.

The threshold would actually be very high. We need to recognize that magnetic hard disks have quite skewed performance characteristics. Flash based disks are precluded due to reasons explained elsewhere. Representative hard disk is capable of ~120 MB/s of sustained throughput and ~10 ms seek time to random location on average. Quick calculation shows that reading or writing an extent smaller than 1.2 MB is faster than literally one seek, on average. Therefore a B-tree operation requiring 2 seeks or more

would be slower than reading or writing an entire 1.2 MB in one sweep. A huge amount of entries can be stored in that amount of data.

Actually even more entries can be stored on disk than in memory. If for example entry keys are sorted then we can store their pairwise differences instead. Smaller numbers can be encoded more compactly using varint encoding. This topic is further explained in a later chapter on extent efficiency.

This argument is basically the same as for the inode extent. If you consider the mapping entries as metadata of some special file then the same theoretical model can be applied.

Fsync is not efficient.

Files stored on computers were important since about the time when files started being stored on computers. Programs started using techniques that would ensure reasonable state after computer was interrupted in it's work which did and still does occur quite often. POSIX standard implies an approach that is being used to this day. Windows system uses an identical approach despite not recognizing POSIX. The well established approach replaces a file using following template:

```
int fd = creat("file.new", S_IRUSR|S_IWUSR);
write(fd, buf, len);
fsync(fd);
close(fd);
rename("file.new", "file");
```

This approach has been the expected way of achieving atomic changes to files and caused a lot of pain when people were depending on different guarantees than that of fsync. For example, ext3 implemented fsync in a naive way that flushed all files to disk and not just the file of interest, which caused it to be slower than perhaps it should have been. At same time ext3 implemented non-POSIX behavior of ordering data blocks before related metadata which made the rename safe even without fsync. These two facts made fsync both slow and useless. As result people started skipping on fsync. Later ordering behavior was disabled to increase performance and non-standard code started corrupting files. This story has been described in LWN post POSIX vs Reality.

Another example is a property called *safe new file* that makes fsync on parent directory unnecessary. SQLite in particular allows to take advantage of this and few other behaviors although it is enabled only on user's request. This behavior is common in modern filesystems but it is not mandated by POSIX so applications should not rely on this property being always met. Refer to [Pillai13] for modern filesystem semantics that are common but are not part of the standard (POSIX). Notice that all of the 6 behaviors they describe are met by a filesystem where all operations are both atomic and ordered, as proposed below. Article also describes how SQLite can increase performance by assuming specific filesystem behavior although user has to enable it on his own accord. Also article shows bugs found in LevelDB, a predecessor to SQLite, that corrupted files because the same behavior was assumed but never verified. Both SQLite and LevelDB would be safe on a filesystem where all operations are atomic and ordered. How such a filesystem could be constructed is described below.

Whatever improvements to filesystems will be made in the future it seems clear that applications will remain to use the approach established by POSIX and filesystems will have to cooperate. Inventing new APIs that ensure safety while also break existing code will not gain traction, no matter how fancy they seem. For example, Btrfs provides an interface to clone a file in zero-time. That would allow to apply sets of changes and also changes to huge files in an atomic manner. Not used anywhere. Another example, Btrfs also provides a way to manage transactions spanning across files. Would seem useful if its own documentation have not outright discouraged using it.

Below we shall consider a new solution to this problem that will keep files correct whether the POSIX compliant approach is used or not while performance is approaching no use of fsync at all.

Consider a filesystem where all operations are both atomic and ordered. Above that fsync calls are implemented as no-op.

We can see that template code remains to keep files consistent. This is due to the fact that application developers (usually) use fsync not to persist data immediately but to persist data before metadata, which is exactly what ordering means. Linux man pages define fsync as a means to persist data immediately but surprisingly this is not what POSIX strictly requires, rather it is just a mainstream interpretation. POSIX first defines fsync (in vague terms) as flushing buffers to a device but then explains in the notes, and explicitly, that if a filesystem can guarantee safety of files in a different way then that also counts as a valid implementation of fsync. Flushing buffers may be an obvious way to do it but not the only way. Excerpt from POSIX documentation:

> *(DESCRIPTION) The fsync() function shall request that all data for the open file descriptor named by fildes is to be transferred to the storage device associated with the file described by fildes. The nature of the transfer is implementation-defined. The fsync() function shall not return until the system has completed that action or until an error is detected.*
>
> *(RATIONALE) The fsync() function is intended to force a physical write of data from the buffer cache, and to assure that after a system crash or other failure that all data up to the time of the fsync() call is recorded on the disk. […] It is explicitly intended that a null implementation is permitted. This could be valid in the case where the system cannot assure non-volatile storage under any circumstances or when the system is highly fault-tolerant and the functionality is not required. In the middle ground between these extremes, fsync() might or might not actually cause data to be written where it is safe from a power failure.*

Persisting ordered operations used to be implemented though ordered persistence. Long time ago blocks were written one at a time (synchronous writes). Operating system buffered changes in memory (the buffer cache) and was the only party that buffered. When hard disks started buffering writes themselves things got broken. System was no longer in a position to tell which blocks were already persisted on the platter and which were still pending. Immediate solution was to implement *flushing buffers* through so called *draining queue*. System withholded all future writes until disk reported that queue was empty. When flushing buffers occurred quite often this approach basically defeated the purpose of installing buffers in disks in the first place.

Much later another major overhaul of hard disks happened. Up to this moment systems reordered requests themselves. Buffer cache accumulated blocks to be written and system decided, without much help from the disk, in which order should the disk write

them. It was more beneficial to process blocks in the order that minimized total time spent on jumping between locations on disk. Unfortunately, system was in no position to determine current orientation of the platter or current position of the head. So system made decisions that seemed to be good enough. Everything changed again when disks started to also reorder requests themselves. Again, system was no longer aware of what is the status of each request. After all the system no longer even knew which of them will go to the platter first. The solution to ensuring order of writes was the same.

Consider a copy on write scheme where first sector contains address of last persisted block and address of root inode, further sectors contain a continuous stream of data and metadata similar to what a log-structured filesystem would contain.

Each operation puts more blocks into the stream in a sequential but asynchronous way. Specifically blocks are ordered according to order of operations. When the filesystem is sure that blocks up to a given point were persisted it issues an atomic but also asynchronous write of the first block, moving the address of last used block. Recovery is not strictly necessary so far because first sector reliably tells us how much data was stored for sure. The window of changes that can get lost is quite significant, depending on how many changes (or for how long) are accumulated before being sent to disk. Modern filesystems however make this sacrifice on regular basis.

Compactness can be achieved by storing intents instead of full structures. For example, in log structured filesystem a single file write could end up storing the data buffer, indirect blocks, file inode, parent directory data and inode, and so on up to root inode. Instead only the data buffer and an intent describing to which file this data belongs to would be enough. Inode would have to be stored eventually but not before write was persistent. Recovery would have to involve doing a so called *roll forward*, scanning the log from last known end forward, looking for intents and verifying checksums. File write intent would have to be incorporated into earlier inode during recovery. Complexity is added to recovery code but performance and utilisation gain may well be worth it.

Robustness can also be achieved in the sense that operations can become persistent as soon as they hit the disk which could be requested from anything between 5 seconds later to right away. First sector does not need to be updated for following changes to be persisted, intents and rolling forward ensure that.

Following code would be safe on atomic ordered filesystem (but not in general):

```
int fd = creat("file.new", S_IRUSR|S_IWUSR);
write(fd, buf, len);
close(fd);
rename("file.new", "file");
```

Encrypting disk blocks is not useful.

Encryption became a common feature that is not used only by geeks and criminals anymore. While in the past user would use some custom program to password-protect archives and store them away with other files, today operating systems are expected to offer encryption out of the box. In particular, systems are often installed on encrypted partitions and are supposed to boot off them. The problem is that initial code must be

loaded before any decryption can be done. Some files need to be available in the clear before rest of them can be decrypted and booting sequence can proceed.

Linux uses a separate unencrypted /boot partition to start a booting process. Then a screen pops up asking the user to provide a password which decrypts a LUKS partition. There is a reason why one partition cannot be enough to boot a system. Block devices, or partitions to be specific, are always encrypted block-wise. There is no facility to keep selected blocks in the clear. Filesystems are mounted on arbitrary block devices and are unaware of how and even whether underlying blocks are being encrypted.

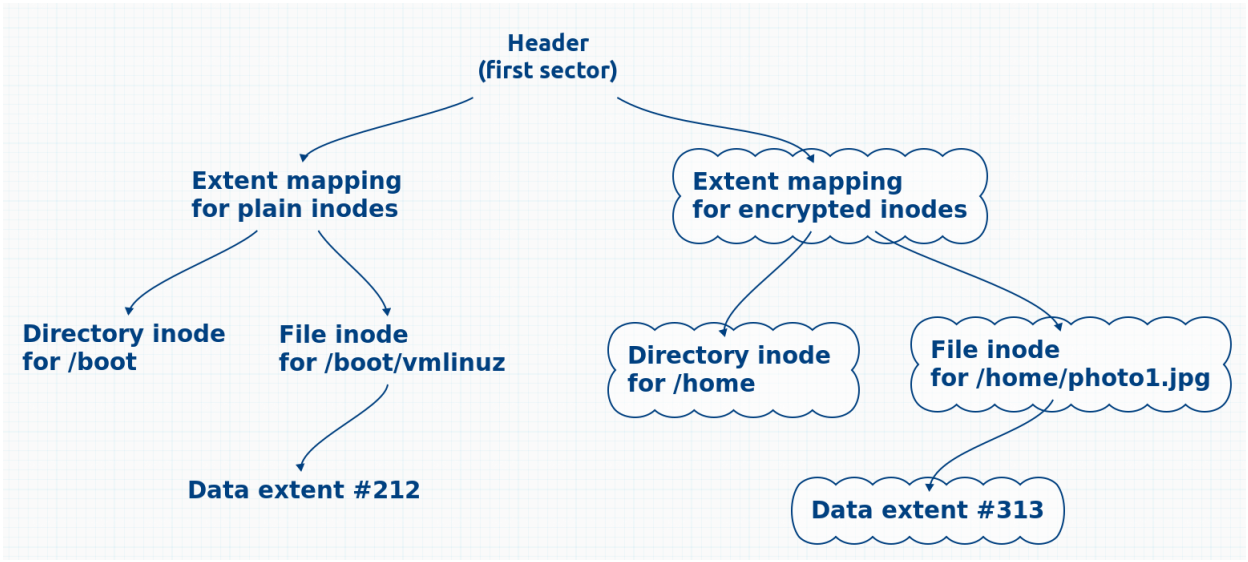
Only solution to booting from a (partially) encrypted filesystem seems to be encrypting filesystem itself and not the block storage device underneath.

Consider a filesystem where encryption is applied to data structures like inodes and not raw blocks. Each data structure is encrypted individually and initially free space is filled with random bytes. Header is stored at a known location and keeps two addresses: one pointing to unencrypted data structure and one pointing to encrypted data structure. Following one address would allow discovery of all files stored in the clear, while second address would be useful only after password was provided.

Encrypted structures would be indistinguishable from free space so there is no way to tell how much data is stored aside of files stored in the clear. When (encrypted) blocks are freed they just become part of the garbage in the background. Freeing blocks from files stored in the clear should be followed by overwrite with random garbage, although it does not need to happen right away. Biggest disadvantage is that entire block device should to be filled with random bytes before it can be used, although that also can be postponed. Filesystem creation would then happen instantaneously but several hours would pass before entire device is filled and full security is guaranteed. Until then an examination would allow to determine that a progressively shrinking fraction of blocks are not in use. Content would not be revealed, only an upper bound on usage quota, and only for first few hours of operation.

Writing files in the clear before unlocking would be impossible since without password there is no way to distinguish free space to allocate from, unless space was reserved in advance. Then files like boot logs could be stored in the clear even before unlocking.

Following diagram shows an example of data structures on disk (clouds mean extents are encrypted, locations on disk are not represented):

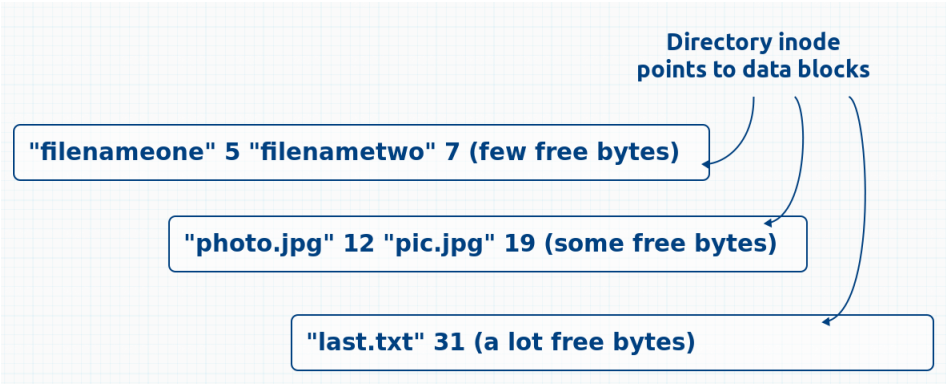


Blocks can store less than extents.

Earlier chapters discussed the issue of storing data structures like dictionaries and inodes in single extents. While previous chapters were considering performance of disk devices this chapter is focused on compactness. Less bits are needed to store data in extent form than when individual blocks are used. Tho performance increase may not be noticeable, space utilisation may increase significantly letting the user store more files and whatever else.

The main reason why extents are more efficient is that blocks impose rigid boundaries between data items stored inside. Items of variable size cannot be compactly laid out next to one another because boundaries between sets of items often not align with boundaries between blocks. When we divide data into blocks we usually do that to make further operations read or write as few blocks as possible, so dividing items between blocks should not take place.

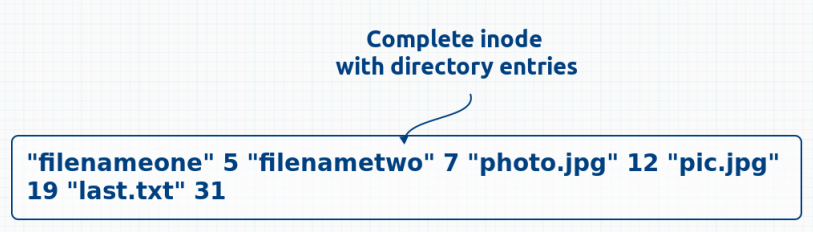
For example, directory entries used to be stored in data blocks like on a diagram below. File names are much shorter than a single block so they are being grouped to occupy as much space in a block as possible. However since file names are variable length, each block has some unpredictable amount of space left unused.



No doubt this approach was beneficial when memory was heavily constrained. Listing

and querying entries requires processing only one block at a time. Removing or changing an entry requires writing only one block on disk. New entries can be added to an existing block in place of already removed entries, or can allocate a new block. All of these operations can be carried out with only one block in memory at a time, although keeping them in the buffer cache would also be beneficial.

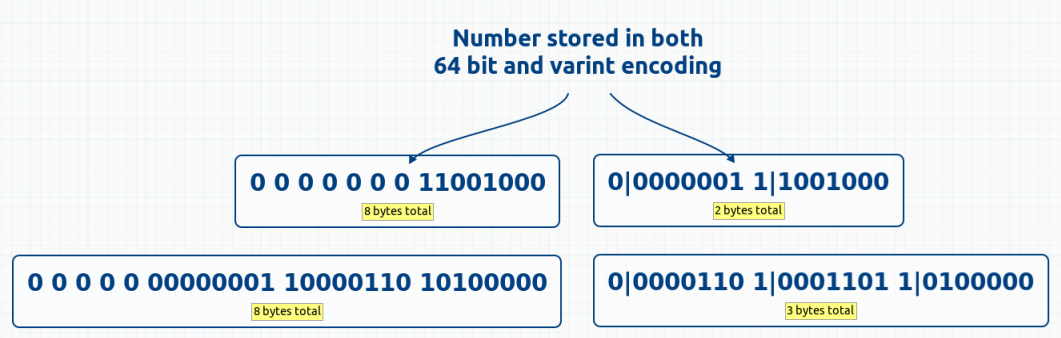
Consider using an extent instead. Items can be laid out next to each other with not even a byte wasted. At most extent could be aligned to block boundaries wasting only some fragment of the last block, often called a *tail*. However if filesystem would allow extents to start and end at byte-aligned addresses then literally no byte would go to waste. This is the case with Btrfs filesystem.



Another example, dictionary that maps integers into some values can also benefit from having all keys available when dictionary is being serialized into a binary blob (an extent). When keys are kept in separate blocks only few keys may be processed at once. Loading and storing entire dictionary each time becomes necessary when dictionary is supposed to always occupy a single extent.

Let us consider how a dictionary could be serialized into a blob. Keys can be arbitrarily reordered within the blob because deserialization process does not care. Let us sort keys in increasing order. Instead of the keys themselves only their pairwise differences are needed. Differences are smaller than entire keys, on average and however else we choose to look at them. Small numbers can be encoded more efficiently using varint encoding which is explained right below. Values may be stored after each corresponding key and be also encoded.

Normally a 64 bit number occupies exactly that much space, 8 bytes. Varint encoding divides a number into 7 bit chunks starting from least significant bits. 8th bit is used to signal whether more chunks follow. For example, numbers 200 and 100'000 are encoded as below. Each byte is separated with a space and bits are concatenated with a pipe. Chunks are shown in binary system.



Varint encoding was developed by Google and used in their serialization framework Protocol Buffers. See their website for a communicable explanation.

Bibliography

McKusi84: Marshall Kirk Mckusick, William N. Joy, Samuel J. Leffler, Robert S. Fabry, A Fast File System for UNIX, 1984
Twee00: Steven Tweedie, EXT3, Journaling Filesystem, 2000, http://olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html
OSTEP: Remzi H. Arpaci-Dusseau, Andrea C. Arpaci-Dusseau, Operating Systems: Three Easy Pieces,
Rodeh12: Ohad Rodeh, BTRFS: The Linux B-tree Filesystem, 2012
ZFS: Jeff Bonwick, Bill Moore, ZFS: The Last Word in File Systems, ,
Agraw07: Nitin Agrawal, William J. Bolosky, John R. Douceur, Jacob R. Lorch, A Five-Year Study of File-System Metadata, 2007
Pillai13: Thanumalayan Sankaranarayana Pillai, et al, Towards Efficient, Portable Application-Level Consistency, 2013

Please know that all cited documents are saved in project's repository and you do not have to search the web to read them. This includes cited web pages.