# Spectrum Scale: GPFS Metadata

Yuri Volobuev

Spectrum Scale Development

Ver 1.0, 20-Oct-16

The term ‘metadata’ comes up often in various documents and discussions involving GPFS technology that underpins IBM Spectrum Scale. What does *metadata* really mean though? How much metadata space needs to be set aside during new file system provisioning? How does metadata space allocation work? What considerations go into the optimal metadata block size selection? What do metadata IO patterns look like? How does one optimize the storage layer for good metadata IO performance? This documents attempts to address these and other questions.

### What is *metadata*?

The term ‘metadata’ was coined relatively recently, and generally means something like ‘data about data’. In the context of a file system, metadata refers to various on-disk structures that are necessary to properly organize and protect user data. In short, everything in a file system that is not data is metadata. This is a very broad concept, and it encompasses a number of entities with very different properties.

## Classes of GPFS metadata

Metadata in GPFS can be broadly divided into 3 classes: descriptors, system metadata, and user metadata.

### Descriptors

The term ‘descriptor’ in GPFS is used to describe several different objects that ‘describe’ a corresponding object. Those are the lowest-level metadata structures in GPFS, and thus conform to somewhat different rules, relative to other metadata.

#### NSD descriptor

A Network Shared Disk (NSD) is a basic building block of a GPFS file system. A file system consists of one or more NSDs. An NSD is often referred to simply as ‘disk’. Each NSD has a cluster-unique name and a globally unique NSD ID. An NSD descriptor is a small (512 or 4096 bytes) structure that identifies a given block device as a GPFS NSD, and contains data representing the essential properties of the device, most importantly its NSD ID. A synonymous term that is commonly used elsewhere in the industry for similar structures is *disk label*. GPFS discovers devices that are under its purview by reading an NSD descriptor off every block device visible to GPFS.

The original version of the NSD descriptor, known as NSDv1, typically resides on a raw device that lacks any sort of a partition table or a record in MBR identifying the device as in-use by GPFS. This has led to repeated instances of various OS partitioning tools concluding that the device is free and reformatting GPFS NSDs for other uses, with predictably disastrous consequences. To address this problem, and simultaneously to make GPFS compatible with storage devices that have native 4 KiB physical block size (something referred to as "devices with 4K sector size", or "4KN devices"), GPFS v4.1.1 introduced a new NSD format, NSDv2. When this format is in use, the block device holding a GPFS NSD has a GPT partition label, and it is harder to reuse such a device accidentally. In addition, the NSD descriptor itself is 4 KiB in size and is aligned on the 4 KiB boundary.

The list of NSDs and their IDs is stored in the GPFS static configuration repository, known as *mmsdrfs*. So the loss of an NSD descriptor per se is a recoverable operation, provided some knowledge about the mapping between NSDs and underlying block devices is retained. IBM Service can provide assistance in the event of an NSD descriptor being damaged.

### Disk descriptor

When an NSD is added to a file system (at file system format time or `mmadddisk` time), a *disk descriptor* (or label) is written to the NSD. A disk descriptor is a small fixed-size data structure that basically identifies a disk as belonging to a specific file system, to prevent accidental reuse. There's one disk descriptor on every disk that belongs to a file system.

If a file system is destroyed without performing full cleanup (e.g. when some disks are unavailable), the disk descriptor may remain on some NSDs. If one attempts to add such an NSD to a file system, using `mmadddisk` or `mmcrfs`, GPFS code is going to detect that a valid disk descriptor is already present on the device, and reject the operation, necessitating the use of an override flag (`-v no`) to proceed. Overriding a safety check has clear risks, and it would be the best to avoid such situations if possible by allowing full cleanup to occur at disk or file system deletion time.

### File system descriptor

A *file system descriptor* (originally called *stripe group descriptor*) is a data structure that is conceptually similar to a *superblock* in a traditional Unix file system. A file system descriptor is a variable-length data structure that comprehensively describes the basic file system composition. It consists of a fixed-size header followed by several variable-length sections that describe other file system properties (a list of disks, a list of log files, and a list of global

snapshots). Depending on the file system configuration, a file system descriptor could be anywhere from a small number of kilobytes to a small number of megabytes in size. The content of a file system descriptor is absolutely vital: without having a good copy of a file system descriptor at hand, a file system cannot be mounted or otherwise accessed. In order to tolerate individual disk failures and increase the overall system fault tolerance, multiple copies of a file system descriptor are written, to a *write quorum* of disks. At mount time, a set of matching file system descriptors needs to be read from a *read quorum* of disks in order to proceed. This has very important implications for setting up certain Disaster Recovery configurations. In general, not every disk in a file system has a copy of a file system descriptor, only some disks do, and it is critical to have enough copies of the file system descriptor to reach the read quorum. In particular, losing exactly half of the disks results in the inability to reach a read quorum, thus an additional copy of the file system descriptor on a 3rd (tie-breaker) site is needed in "stretch cluster" synchronous disaster recovery configurations.

Together with NSD and Disk descriptors, file system descriptors represent an exception: metadata objects that can reside on NSDs outside of the *system* pool.

## System metadata

Under the covers, GPFS maintains a number of *system metadata files*. Those files have inode numbers, like regular files, but are not visible in the directory structure (with the notable exception of quota files, as discussed below), and are not directly accessible to file system users through regular interfaces.

### Inodes

An *inode* is a basic metadata structure that describes a single file (or directory). The inode size can be chosen at file system format time, and ranges from 512 to 4096 bytes (as of Spectrum Scale V4.2, supported inode sizes are 512, 1024, and 4096 bytes). The default inode size historically has been 512 bytes for all early GPFS releases, and as of V4.1.1 it defaults to 4 KiB.

An inode consists of a fixed-size (128 bytes as of V4.2) header, followed by a payload consisting of one or more of the following:

File data (when the file or directory is small enough to fit in inode, for file systems formatted with V4.1 or later)

A list of disk addresses pointing to data blocks or indirect blocks (depending on the file size)

Extended Attributes (EAs), when present

Naturally, there's competition between different types of the inode payload, and the exact details of what goes into the inode get complicated when in-inode EAs are present (more on this below).

Inodes are organized in *inode files*. An inode file is a record-based file, where each record is an inode. Inode file always uses inode number 0. Every file system contains at least one inode file, containing all inodes in the *active file system*. When a snapshot is created, a new inode file is created to represent the snapshot. Each snapshot has a (sparse) inode file. When multiple *inode spaces* are present, as the result of creation of *independent filesets*, the logical inode number space in the inode file is subdivided, and the active file system inode file may become sparse (it is dense when only a single inode space is present); snapshot inode files are always sparse.

An active file system inode file is created at file system format time. At that time, the limit on the maximum number of inodes is set, and some inodes are allocated. The inode file can then grow if the file system runs low on free inodes, as long as the limit on the number of inodes is not reached. This operation is known as *inode expansion*. One can choose to pre-allocate all inodes up front, or let inode expansion replenish the supply of free inodes as needed. The same concept applies to individual independent filesets: some quantity of inodes is allocated at fileset creation time, and an inode expansion can then kick in to allocate more when needed.

An inode file can grow, but it cannot shrink. This is the basic reason why GPFS imposes a cap on the maximum number of inodes. A runaway job that creates files on a fast file system can allocate too many inodes quickly, and that is usually not a good outcome. One can set the maximum number of inodes high if desired (the GPFS architectural limit on the maximum number of inodes is $2^{64}$, while the maximum tested size is about 9 billion, as of V4.2.1), but it should be kept in mind that the inode file represents a major source of metadata space consumption in GPFS, and a larger-than-necessary inode file negatively impacts the runtime of certain file system management operations.

### Inode allocation map

*Inode allocation map* (inode number 2) is a system metadata file that contains a *fat bitmap* that represents, using 2 bits per inode, the allocation state of each inode in the file system: in use, free, being allocated, being deleted. The bitmap is partitioned into *inode allocation segments*, where each segment is individually accessible. There is exactly one inode allocation file per file

system; since no inode allocation or deletion can happen in a snapshot, there's no need to have a per-snapshot inode allocation map. Just like an inode file, an inode allocation file is allocated at file system format time, and can grow (but not shrink) when inode expansion runs. A single inode allocation file is shared by all inode spaces.

### Block allocation maps

A *block allocation map* is a system metadata file that contains a simple bitmap that represents, using 1 bit per *subblock* (i.e. 1/32$^{nd}$ of a block), the allocation state of each subblock in a storage pool: free or in use. The bitmap is partitioned into *allocation segments*, where each segment is individually accessible. There is one block allocation map per storage pool. The map for the system pool always uses inode number 1, while maps for other pools use dynamically allocated inode numbers. Just like with the inode allocation map, there's no need to have a per-snapshot block allocation map.

A block allocation map is created at storage pool creation time (which means file system creation time in the case of the *system* pool). Several different considerations go into the determination of the allocation map layout. Ideally, the map should have enough segments for each node in the cluster to have a handful of segments cached, without having to fight with other nodes for access rights, but not so many as to create an undue amount of overhead and hurt IO performance. In a non-FPO GPFS file system, each allocation segment has a piece of bitmap covering every disk in the storage pool, to allow for wide striping, and this means that having segments that are too small is bad for performance: a node writing to a large file would need to procure fresh allocation segments often due to quickly running out of space in the ones if already has cached. So there needs to be a healthy balance between the number of allocation segments (not too few, not too many) and the allocation segment size (not too small). The two inputs that have the biggest role in the allocation map layout are the expected maximum number of nodes in the cluster (`-n` parameter of `mmcrfs`), and the size of the largest disk in the storage pool. Those two parameters need to be chosen judiciously, because once an allocation map is formatted, it cannot be reshaped. If an existing allocation map has a suboptimal format (for example, the default value of 32 was assumed for the maximum number of nodes, while the cluster actually contains hundreds of nodes), the only way to work around this is to create a new storage pool and migrate data to the new pool.

### Log files

GPFS uses *logging*, also known as *journaling*, to atomically commit complex transactions that simultaneously modify multiple objects. Each node mounting a file system is assigned an

individual *log file*, which acts as a circular buffer for writing a stream of *log records*. The size of an individual log file is set at file system format time (as of V4.2, the default log size is 4 MB or the metadata block size, whichever is larger). There are at least as many log files as the nodes mounting the file system. The initial set of log files is created at file system format time, and more are allocated on demand as more nodes mount the file system.

The way log files are managed changed over time. More recent versions of GPFS, starting with V3.5, use *striped* logs, where blocks in each log file are allocated from different disks, the same way blocks are allocated for regular files. Older versions of GPFS use a different approach: each log file has blocks allocated from a single disk, and a *log group* comprising of several log files is assigned to each node to provide tolerance against disk failures. Striped logs generally provide better performance due to a greater level of IO parallelism.

Log files exist only in an active file system, not in snapshots. They use dynamically allocated inode numbers, although since the initial set of log files is created early during file system format, those have low inode numbers.

### Access Control List file

*Access Control List (ACL) file* is a record-based system metadata file, where each record contains a GPFS ACL object. Files and directories that use ACLs refer to records in the ACL file rather than store ACL objects in the inode or another per-inode structure. This works well because in practice the same ACL is often used by a large number of files and directories, due to the way ACL inheritance is typically set up. So storing ACLs in a centralized location offers substantial deduplication benefits. ACL file records are allocated on demand, and the file size is proportional to the number of unique ACLs in the file system. Unused ACL records are freed by a garbage collector, but the ACL file itself never shrinks.

While inodes in snapshots can point to records in the ACL file, the ACL file itself is not copied to snapshots, and only exists in the active file system.

### Extended Attribute file

*Extended Attributes (EA) file* is a record-based system metadata file that was used in older versions of GPFS to store EA records. Starting with GPFS V3.4, EAs are stored either in the inode or in EA overflow blocks (see below). Only file systems formatted with older versions of GPFS and not migrated to the new format contain an EA file. Whether the old EA storage format or the new format (“FASTEA”) is used can be determined using the

"`mmlsfs --fastea`" command. A file system that still uses the old format can be converted to the FASTEA format, in offline or online mode, using the `mmmigratefs` command.

Where present, the EA file typically uses inode number 5 (although in certain rare scenarios it may use a different inode number). The file is typically sparse, containing one variable-sized (at least 16 KiB, or metadata subblock, whichever is larger) record per inode. If a given inode has no EAs, the corresponding EA file record is a hole.

### Quota files

When the quota feature is enabled, one or more *quota files* are created. Prior to V4.1, quota files had the distinction of being the only type of metadata files that are visible in the directory structure (under self-explanatory names "user.quota", "group.quota", and "fileset.quota" in the file system root directory). The original intent of making the quota files visible was facilitating backup and restore operations. However, given that a GPFS file system has much other metadata that ought to be backed up, and it's impossible to backup/restore such metadata using standard POSIX interfaces, making an exception for quota files made little sense (although this created many complications for the implementation). In later versions, quota files are no longer visible, although they are still in use under the covers. Quota files contain configured quota limits and recorded quota usage for a given principal (user ID, group ID, or fileset ID). Since the number of principals rarely gets extremely large, quota files typically stay fairly small. Earlier version of GPFS copied quota files to snapshots, to facilitate a snapshot restore operation, but the current versions do not (the usage is recalculated at snapshot restore time).

### Fileset Metadata file

Starting with V3.1, every file system contains a *fileset metadata file* (FMF). This is a record-based low-level system metadata file, where each record contains information about a fileset or fileset snapshots. Auxiliary fileset information needed by AFM is also stored in an FMF. In the original implementation, the record size was fixed at 1 KiB, starting with V3.5, the record size increased to 4 KiB. The number of records in an FMF depends on the number of filesets in the file system, and if snapshot of independent filesets are present, the number of such snapshots. Given the present limits on the number of filesets, an FMF file cannot get very big. This is a rare case of a system metadata file that gets copied to snapshots.

### Policy file

If a *placement policy* is installed in a file system, using the `mmchpolicy` command, the content of the policy file (copied almost verbatim from the input file, after the content has been validated) is stored in the *policy file*. The file is correspondingly a small text file.

### Allocation Summary file

GPFS calculates the block and inode usage in a file system by running block and inode allocation recovery code when the file system is opened by a file system manager. This basically involves reading all block and inode allocation map segments, and tallying up the results. In a large file system, allocation bitmaps can similarly be fairly large, and thus is takes some time to run the allocation recovery operation. While the recovery is in progress, what to do if a `df` command (which issues a *statfs* syscall) is run? Without real usage totals at hand, the *statfs* result would have to be fudged in some fashion, and the conservative approach is to under-report (as opposed to over-report) free block and inode usage. This means that an untimely `df` command may show odd-looking numbers for a GPFS file system. To ameliorate this problem, GPFS maintains an *allocation summary file* for each storage pool. This is a small file that contains the basic data needed to respond to a *statfs* syscall. The file is updated periodically, and has approximately accurate information. The file is automatically created as needed.

## User metadata

In contrast to system metadata, *user metadata* describes objects created by users: files and directories. As such, almost all user metadata is created on demand, when the file system is in use (the root directory and indirect blocks for system metadata are created at file system format time). This stipulates different dynamics for management and allocation of metadata objects (relative to system metadata): they are allocated and freed as needed, one at a time, as opposed to being packed as records into a larger file.

### Directories

A directory is a special kind of a file. A directory consists of an inode and zero or more blocks. In file systems that support storing data in inode (V3.5 and later), small directories reside in their inodes (this is in fact an important use case for data-in-inode). Larger directories are basically sparse files with a custom block size. Older versions, up to V4.1, use up to 32 KiB directory block size. Versions V4.1 and newer allow directory block size up to 256 KiB (or metadata block size, whichever is smaller).

Directory blocks are allocated when new entries are inserted. In order to provide for an efficient entry lookup algorithm, a directory file quickly becomes sparse as entries are added, so the concept of a directory 'size' is not very meaningful; it is more instructive to look at the number of blocks allocated. When directory entries are deleted, a directory may become extremely sparse. Prior to V4.1, the code didn't attempt to compact very sparse directories or free completely empty blocks, so a directory could only grow, not shrink. Starting with V4.1, the code is enhanced to automatically compact directories that had become too sparse, and deallocate empty directory blocks.

### Indirect Blocks

GPFS addresses data blocks using 12-byte *disk addresses* (DAs). When a new file (or a directory) is created, the DAs for the first few blocks are stored in the inode. Once the file grows to be too large for all DAs to fit in the payload portion of the inode, *indirect blocks* are used to store additional DAs. An indirect block (IB) is a fixed-size metadata object, consisting of a 40-byte header followed by an array of DAs. The IB size is set at file system format time, and defaults to twice the metadata subblock size, up to 32 KiB (as of V4.2). IBs are allocated and freed on demand, as files grow and shrink (or are deleted). The number of logical blocks that a single IB can address depends on the nature of the blocks being pointed to (data vs metadata) and the corresponding data or metadata *maximum* replication factor (`-M` or `-R`). For example, a leaf IB (i.e. an IB at the bottom of the IB tree, pointing to actual data blocks rather than other IBs) for a file with the maximum data replication factor of 3 reserves 3 12-byte DA slots per one logical data block. The number of slots that contain real DAs (as opposed to the special placeholder NULL DA value) depends on the actual replication factor.

### Extended Attribute Overflow Blocks

Starting with V3.4, GPFS stores extended attributes (EAs) either in the file inode or in an *EA overflow block*. The logic used to decide where a given EA should go is involved. EAs are generally divided into two types: *system* EAs (set by GPFS code itself), and *user* EAs. Several product features rely on EAs under the covers, notably encryption, AFM, clones, and DMAPI. Different EA types have different priority levels assigned. Certain high-priority system EAs *must* reside in the inode, due to implementation considerations. Lower-priority EAs, including all user EAs, can reside either in the inode (when space permits), or in an EA overflow block. An overflow block is a fixed-size metadata object. The overflow block size is either the metadata subblock size or 64 KiB, whichever is smaller. There is at most one EA overflow block per inode, allocated on demand during EA set operations, and freed when the file is deleted. In

practice, bulk low-priority EAs are not commonly used, and thus EAs are mostly stored in inodes, and EA overflow blocks are fairly uncommon.

## Provisioning Storage for Metadata

One of the unique features of GPFS is the ability to separate data and metadata on different disks and storage pools. Starting with V3.5, different block sizes can be specified for data and metadata. All of this provides a lot of flexibility in provisioning storage in an optimal fashion, but this is also a source of additional questions.

### Data and Metadata Separation

It makes a lot of sense to place data and metadata on different devices. Metadata IO patterns are distinctly different from those of data, and in many workloads metadata performance is the bottleneck, and thus optimizing metadata performance provides an outsized benefit to the overall system performance. Furthermore, the space required for metadata storage is a small fraction of the overall file system size, which allows using smaller quantities of faster (and more expensive) media for metadata storage, in a cost-efficient manner. Similar considerations go into the block size selection: while it is often advantageous to use large (8 or 16 MiB) block sizes for data, the same is generally not true for metadata.

GPFS allows designating a given disk as *dataAndMetadata*, *dataOnly*, and *metadataOnly*. The designation is specified at `mmcrfs` or `mmadddisk` time, and can be changed at a later time using `mmchdisk` followed by `mmrestripefs`. When multiple storage pools are present, it is important to remember that metadata can only reside in the *system* pool. All other storage pools can only contain *dataOnly* disks.

It is possible to specify a different block size for data and metadata, using the `--metadata-block-size` argument of the `mmcrfs` command, but only if the *system* pool contains only *metadataOnly* disks. In other words, data and metadata must be in separate storage pools when different data and metadata block sizes are used; it's not possible to mix different block sizes in the same storage pool.

### Metadata IO Patterns and Storage Medium Selection

Characterizing metadata IO patterns is not a simple task, because there's a lot of variation in IO size and locality depending on the task. A few typical workloads can be used as examples:

i) Directory traversal (e.g. recursive `find` or `du`). This mostly involves reads of directory blocks and inodes (for stat purposes). The IO pattern on the disk level looks like a stream of random reads, with IO sizes dictated by inode and directory block sizes.

ii) Small file creation (e.g. untar'ing a source code tree). This involves lots of writes of inodes, block and inode allocation map segments, directory blocks, and log writes. The low-level IO pattern is mostly random writes, most of them single inode writes, and log file writes (a few tenths or hundreds of bytes each).

iii) Inode scan (e.g. running `mmapplypolicy`). The inode scan API reads inodes in full blocks, and then the directory traversal phase involves reading individual directory inodes and directory blocks. The low-level IO pattern is a mixture of metadata block-sized somewhat-sequential IOs and random inode and directory block reads.

iv) Deleting a large tree (`rm -r`). This involves deallocating blocks and rewriting inodes and (some) IBs, as well as deleting entries from directories, updating allocation bitmaps, and logging all of the above. The low-level IO pattern is similar to that of mass file create, but there's also a significant fraction of small random reads.

As one can observe, this isn't a simple picture, but some properties are clear. Random small read performance is very important, and so is random small write performance. On the other hand, large sequential write performance is not important at all (this only occurs at file system format time, when large system metadata files are written out for the first time). Flash and SSDs typically excel at small random reads, and constitute a good medium for storing metadata for read-mostly workloads. However, the write performance angle is more complicated. While some flash/SSD devices offer good small write IO rates, not all do. Many flash/SSD devices, especially on the lower-priced end of the spectrum, also have a problem of limited write endurance, and using those for holding GPFS log files can be problematic.

What helps tremendously is having a layer of NVRAM or reliable write cache in front of an actual storage device. Devices with NVRAM-like properties (very low small write latency, high write endurance, and no need for a large size) are particularly advantageous for holding GPFS log files. In fact, the desired properties for a storage device holding log files vs a storage device for the rest of metadata are different enough to justify separation of log files into a separate storage pool: *system.log*. Creating a dedicated *system.log* storage pool is particularly advantageous when small data write logging is enabled using the HAWC feature, but this is also important for optimizing metadata write-intensive workloads.

When traditional spinning disk RAID arrays are used for storing metadata, separating data and metadata also makes sense. While data is typically stored on RAID arrays with wide erasure codes that offer higher storage efficiency (e.g. 8+2P RAID6), using such arrays for storing metadata hurts write performance: most metadata writes are smaller than the RAID stripe size, and thus a full stripe read-modify-write operation is needed in order to commit a write. While having a substantial amount of write cache in a RAID controller helps, in addition it is advantageous to use RAID1 or RAID10 arrays for *metadataOnly* disks, to avoid the read-modify-write penalty. Since metadata is small, using less efficient RAID erasure codes for metadata storage in exchange for better performance can be a good tradeoff.

## Metadata Block Size Selection

As discussed above, when large block size is used for data, it is recommended to use a smaller block size for metadata. Why is it not a good idea to use large blocks for metadata? Since metadata writes are mostly small and random during normal operation, there's no performance benefit in using large metadata block size. However, there's a significant disadvantage: disk space utilization.

For the purposes of this discussion, it is important to understand the concepts of the *minimum allocation unit size* and the *minimum IO size*. GPFS manages disk space using *blocks*, and the minimum unit of space that can be allocated is a *subblock*, which (as of V4.2) is a 1/32$^{nd}$ of a block. It is not possible to allocate less than a subblock worth of disk space. This becomes an issue when the subblock size is larger than the maximum size of a user metadata structure that is allocated one at a time, specifically a directory block or an IB. For example, if the metadata block size is 16 MiB, meaning 512 KiB subblock size, when a new 32 KiB IB is allocated, one subblock is allocated; the first 32 KiB in the subblock contain IB data, the rest is unused. This is clearly suboptimal. Directory blocks suffer from the same predicament, although the increase in the maximum directory block size in V4.1 mitigates the problem somewhat. Note that the problem is the disk space utilization efficiency, not performance. The *minimum IO size* in GPFS is 512 bytes, and it is not directly tied to the subblock size. A 32 KiB IB is read and written using 32 KiB IO requests, even if a larger subblock has been allocated to hold it. Also, the space utilization issue doesn't affect system metadata files (e.g. an inode file), because those files have multiple records packed into a single block; regardless of what the metadata block size is and what the inode size is, an inode file block has no wasted space.

It is therefore not a good idea to use very large blocks for metadata: there are no real benefits, but there are clear downsides. What is a reasonable metadata block size then? The default of

256 KiB is a sound choice, but 512 KiB and 1 MiB blocks should work about as well. Using metadata blocks sizes larger than 1 MiB and smaller than 256 KiB is not recommended.

## Metadata Space Requirements Estimation

If placing metadata on a separate set of disks is desirable, how does one estimate how much space needs to be provisioned? Unfortunately, this is not a simple question.

Long-time users of GPFS have probably encountered various "rules of thumb" that stipulate something like "metadata is 1/2/3/4/5% of data". The reality is more complicated though, and an unfortunate result of using such "rules of thumb" is a severe over-provisioning of metadata space in many installs. It is not easy to improve upon such simple rules though.

To answer the easy question first: there's no way to put a *limit* of what fraction of the file system overall space can be occupied by metadata. It is easy to construct a pathological corner case where 100% of used space in a file system is metadata. A file system containing nothing but directories or nothing but tiny files (where 100% of data fits into inodes) would be an example of a 100% metadata file system. Such extreme cases are not practically important, of course, but they illustrate why having a hard limit on the fraction of metadata space is infeasible.

While it is pointless to try to predict a "guaranteed not to exceed to maximum" for metadata space requirements, it is possible to estimate the practical metadata space needs with considerable accuracy if some aspects of the file system usage are known.

Metadata replication is one basic factor. The metadata object size estimation discussion below covers individual metadata object sizes. If metadata replication is enabled, the individual object size needs to be multiplied by the metadata replication factor (`-m`).

### System metadata

Most system metadata objects are small enough to be ignored. On a file system of a reasonable size (tens of gigabytes and larger), only the inode file consumes a large enough amount of space to be concerned about. The active file system inode size is easy to calculate: it's (inode size × number of allocated inodes). When snapshots are present, things are more complicated. A snapshot inode file is typically very sparse, and the number of blocks allocated in the most recent snapshot depends on the rate of change in the active file system: only those inode file blocks that are modified in the active file system and required a copy-on-write are copied to the most recent snapshot. For example, a daily rate of change of 5% may result in over 5% of inodes being copied to snapshots created during a one-day interval. As a

performance optimization, in GPFS entire blocks on inodes are copied to the previous snapshot when the first inode in the block is modified after a snapshot creation. This means that the amount of space occupied by inodes in snapshots is somewhat larger than (number of modified inodes × inode size). Depending on the metadata block size and the locality of inode changes (many changes within the same block vs random changes in different blocks) the amount of space consumed by inodes in snapshots may vary. However, since in practice most changes involve an "active files" subset, typically inode files in snapshots remain very sparse.

Beyond the inode file, block and inode allocation maps are distant second and third, since the very nature of a bitmap (32 bit per block and 2 bits per inode) means that allocation maps are much smaller than the objects that they describe. A notable exception is small (hundreds of megabytes to small number of gigabytes) file systems that are sometimes created for test purposes. In a small file system, a block allocation map may represent a substantial fraction of the total, due to the block allocation map structure that's optimized for large file systems. For larger file systems, block allocation maps are small enough to ignore.

### User metadata

User metadata, specifically directories and IBs, can be a significant source of metadata space consumption, depending on the file system composition. While it may be difficult or impossible to know at file system provisioning time what the file system composition is going to be, it is reasonable to use a fairly large body of published data describing the composition of typical file systems: the ratio of files to directories and the file size distribution. Published data hints strongly at a general rule: most files are small, but most disk space is consumed by a few large files. Of course, this rule doesn't apply in those situations where the majority of files are created by an application with a specific file usage pattern (e.g. a file system populated solely with 2-4 MB image files).

When estimating user metadata space requirements, it is also important to keep in mind disk space utilization considerations discussed above, especially when larger metadata block sizes are used.

### Indirect Blocks

While small files do not use IBs, larger files do, and if significant numbers of larger files are present, the space occupied by IBs can be a significant consideration. For estimation purposes, the following formula can be used:

$$IB\ space = 12 \times R\ \times N_B\ bytes$$

where *R* is the maximum data replication factor (`-R`) and $N_B$ is the number of blocks in large files. The threshold for a file becoming "large" and requiring IBs can be calculated using this formula:

$$Tr = \frac{I_s - 128 - S_{EA}}{R \times NB}$$

where $I_s$ is inode size, *R* is the maximum data replication factor (`-R`), $S_{EA}$ is the size of in-inode extended attributes, and $NB = Filesize/Blocksize$ is the number of blocks. If the size of a file is larger than $Tr$, the file requires one or more IBs. This formula is simple enough to use, except when EAs are present. Unfortunately, there is no simple technique for accurately calculating $S_{EA}$, as of V4.2.2; for a specific file the value can be estimated using `mmlsattr` output.

Since each IB is assigned to one inode, there is no sharing of space in a single IB by multiple inodes. So if a file is just big enough to require an IB, the unused space within the IB is in fact unused. This can be an important consideration when large numbers of medium-size files are present.

### Directories

Accurately estimating the footprint of directories is difficult, because several of the relevant input parameters are typically not known with much precision. However, it is still a useful exercise to go through.

Like regular files, small directories can fit in inodes (provided the file system format is new enough to support data-in-inode, i.e. V4.1 or later). This can be a key consideration in the selection of the inode size for a new file system, since in many file systems most directories are small. Storing directory data in inode is beneficial for metadata disk space utilization *and* performance. The published file system usage data suggests that most directories are small, and if most directories fit into inode, the amount of disk space consumed by large directories may be small enough to ignore. Of course, each file system is different.

How can the size of a directory be estimated? A GPFS directory consists of a 32 byte header, a sparse array of directory entries (where each entry may have a different size), and a footer. When a directory is stored in the inode, the footer is omitted. Thus the most important factor is the average directory entry size. A directory entry contains a file name and 12 bytes of other information (on file systems formatted with V2.2 and later). As needed, the entry size is rounded up to the next 32 byte boundary (or 16 byte boundary if the directory block size is 8 KiB, which only happens with very small metadata block sizes). Naturally, the average file

name length can vary, and the directory entry size can vary accordingly, from 32 bytes to 1056 bytes. However, in most file systems very long file names are uncommon, and it is reasonable to assume that an average directory entry is 32-64 bytes.

To optimize the lookup operation performance, GPFS directories are managed as an extensible hash table. Once a directory block fills up, it is split into more blocks, and the entries are moved to the newly allocated blocks. Thus for directories with multiple blocks a typical block is sparse. In other words, more blocks are required to store all entries than what the (total size of the entries / directory block size) number suggest. Again, a reasonable assumption is to expect an average directory block to be half-full (or half-empty, as the case may be). As discussed above, newer versions of GPFS automatically perform directory block compaction and free unused blocks after entry deletion, while older versions do not.

## Summary

Going through a metadata sizing exercise on real-world file systems demonstrates that typically the amount of space used by metadata is small, <1% of the total file system space. Excessive over-provisioning of metadata space to the levels of 5% or more of the total space is very rarely warranted. However, significant variations are possible in specific scenarios, so it is not always practical to use any simple “rules of thumb” to arrive at a reasonably accurate answer.