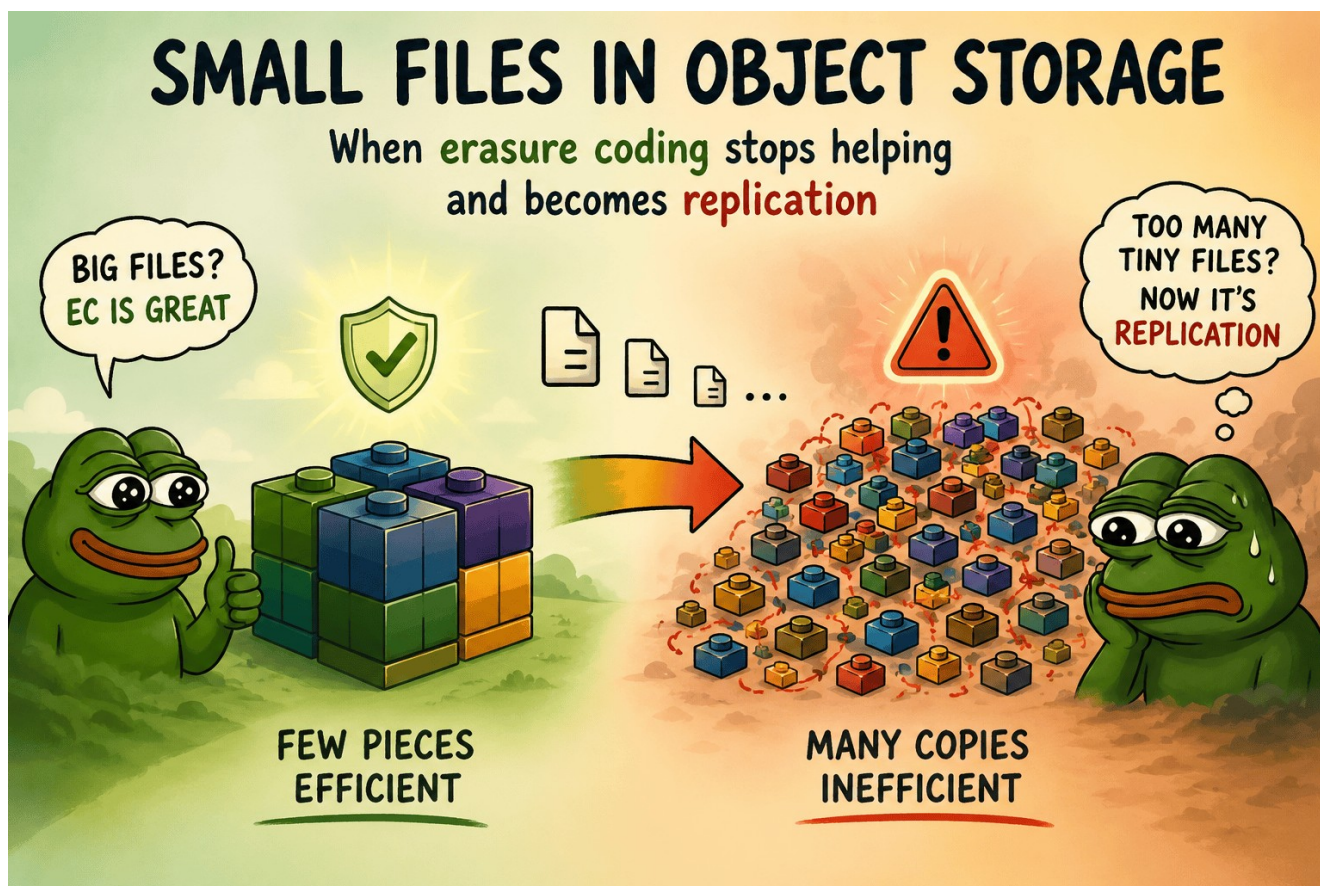


MinIO and Small Files: When Erasure Coding Becomes 15x Replication

Julien Laurenceau | Published 2026-07-23 | Updated 2026-09-30

<https://pepitedata.com/minio-small-files-erasure-coding-replication/>

PepiteData | LinkedIn: Julien Laurenceau | X: @Julien_Lau



TL;DR. MinIO published a blog, "[The Challenge of Big Data: Small Files](#)", that positions object storage as the answer to the small-file problem. On one point it is right: MinIO does not inherit HDFS's small-file problem. There is no NameNode holding the whole namespace in a single machine's RAM, and no 128 MB block charged against every tiny file. That is a real improvement, worth roughly an order of magnitude in headroom. The blog then lets the reader conclude that MinIO has no small-file problem at all. That conclusion is wrong. MinIO has no index; it stores one metadata file per object on every drive of the erasure set, and for small objects (under 128 KiB) it inlines the data into that metadata file. On small files that turns the storage-efficient erasure code into something closer to replication, with a replication factor equal to the set width. On 5 servers of 12 drives, MinIO picks a 15-wide set by default, so each 10 KB object is materialized 15 times over. This is a companion to my earlier post, [MinIO and lots of small files: why it is not a database](#).

What the MinIO blog gets right

The blog describes the classic pain of "lots of small files" (LOSF): billions of objects in the kilobyte range, where metadata operations dominate and the storage layer spends all its time opening, seeking, and

closing rather than moving bytes. That description is accurate. It then argues that a lakehouse built on MinIO is the answer, because MinIO stores metadata inline with the object and distributes small objects across drives with erasure coding, instead of leaning on an external metadata database.

That mechanism is real, and it does defeat one specific, famous failure mode: the HDFS small-file problem. To see why the improvement matters, you have to state the HDFS problem precisely.

The HDFS small-file problem, stated precisely

HDFS keeps the entire filesystem namespace in the memory of a single process, the NameNode. Every file, every block, and every directory is a live object on the NameNode's JVM heap. The common rule of thumb is about 150 bytes of heap per object.

Two consequences follow:

- **The ceiling is one machine's RAM.** A NameNode with a large heap holds on the order of a few hundred million files before it stops. Store 15 million files a day for a year and you are at 5.5 billion files. At 150 bytes each that is over 800 GB of heap on one JVM. That does not exist. HDFS falls over long before you run out of disk, and the thing that falls over is a single memory-bound process.
- **The block tax.** HDFS uses a large default block (128 MB). A 10 KB file does not waste 128 MB of disk (that part is a myth), but it does consume a full block object in the NameNode and it wrecks job planning, because a scheduler that assigns one task per file drowns in tiny tasks.

So the HDFS small-file problem is really a **centralized, memory-bound metadata problem**. MinIO does not have a NameNode. It has no central metadata store at all. Each object's metadata (`x1.meta`) lives on disk, next to the object, spread across the drives of the erasure set. Nothing accumulates in one machine's RAM as the object count grows. On that specific axis, MinIO is genuinely better, and the blog is entitled to say so.

What MinIO actually changed

Here is the sentence the blog does not write: **MinIO did not delete the metadata cost, it relocated it from RAM to disk.**

That trade is a good one, and it buys real headroom. A memory ceiling on a single node is a hard wall you hit at hundreds of millions of files. A disk-backed, distributed namespace pushes that wall out to billions of objects. Call it a 10x improvement; the order of magnitude is the point, not the exact factor.

But relocating a cost is not the same as removing it. With metadata on disk, three new costs appear, and all three scale with the number of objects, not the number of bytes:

1. **Inodes and directory entries.** MinIO has no index. The S3 namespace *is* the XFS directory tree. Each object is a per-object subdirectory holding one `x1.meta`, and that subdirectory exists on every drive of the erasure set. Objects turn directly into inodes and directory blocks, multiplied by the set width.
2. **readdir and lookup cost.** Every LIST, every prefix walk, is a filesystem traversal. XFS directory performance degrades past tens of thousands of entries per directory and shifts to a more expensive B+tree layout past roughly 200,000. Small objects are precisely the case that fills directories fast.
3. **The background scanner.** MinIO runs one scanner that drives ILM (expiry), healing, bit-rot checks, and replication retry. It walks the namespace object by object. Its duration is linear in the object count. When it stretches from hours to weeks, everything it drives runs weeks late.

None of these is a RAM ceiling, so none of them shows up as an out-of-memory crash. They show up as slow LIST, a scanner that never finishes, healing that silently runs behind, and ILM that cannot keep up

with ingestion. The failure is quiet, and it arrives well before the disk is full. That is the difference the blog omits: HDFS fails loudly on RAM; MinIO degrades quietly on disk metadata. A quiet, distant wall is still a wall.

The wide-set trap: erasure coding that behaves like 15x replication

This is the part the sizing slide never shows, and it is the one that actually decides your capacity.

MinIO has no index and no shared metadata store. For each object it writes one `xl.meta` file into a per-object subdirectory **on every drive of the erasure set**. Not one metadata file per object. One metadata file per object, per drive. The width of the erasure set is therefore a multiplier on everything: inodes, directory entries, filesystem blocks, scanner work.

For objects below MinIO's inline threshold, which is 128 KiB by default, there is no separate data file at all. MinIO **inlines the object's payload directly inside** `xl.meta` rather than writing a `part.1` shard file next to it. A 10 KB object is therefore fully contained in those per-object metadata files, one on each of the 15 drives. There is no `part.1` on disk, only 15 `xl.meta`, and each is a real file that carries a slice of the data plus the full metadata envelope. This is why the per-drive multiplier applies to the whole object, data included, and not just to a few bytes of bookkeeping.

Now look at what MinIO chooses for you. Erasure coding is sold as the storage-efficient alternative to replication: EC 8+7 on a 15-drive set has a raw overhead near 1.9x, versus 3x for triple replication. That holds for large objects. For small objects it inverts, because three fixed costs dominate the tiny payload, and each is paid once per drive of the set:

1. **The metadata envelope.** Each `xl.meta` carries the object's metadata, ETag, version ID, checksums, and erasure layout. On a tiny object that envelope is already comparable to the payload, and it exists on all 15 drives.
2. **Filesystem block rounding.** XFS allocates in 4 KiB blocks. A 2 KiB `xl.meta` still consumes a full 4 KiB block. Times 15 drives.
3. **The per-object subdirectory.** Each object's subdirectory costs its own inode and directory block, again on all 15 drives.

The data itself is genuinely erasure-coded into shards, not copied 15 times. But the footprint and the metadata cost are set by the per-drive envelope plus block rounding, times the set width. For an object small enough, those fixed costs dwarf the shard, and the object behaves on disk **as if it were replicated across every drive of the set**. You chose erasure coding to pay 1.9x instead of 3x. On a 10 KB object you pay closer to 12x, which is worse than triple replication by a factor of four, and the multiplier is the set width: 15.

Here is the trap inside the trap: **you did not choose 15**. Give MinIO 5 servers of 12 drives (60 drives) and it auto-selects the widest erasure set that divides the topology. That is a 15-drive set: four sets of 15. The default parity on a 15-drive set is EC:4, and for a durability or compliance posture you push it to the maximum the set allows, EC:7, giving EC 15:7. The parity choice moves the byte overhead a little. It does not move the replication-like behavior on small files, because that is fixed by the width. The wider the set MinIO picks, the worse small files get, and it picks a wide one by default. See the [MinIO erasure-coding documentation](#) for how the set size and parity are selected.

A concrete sizing that goes absurd

Numbers make this real. The following is an illustrative sizing, not a real client platform. It uses round figures on purpose.

Workload. 15 million objects per day, 10 KB each, written continuously. The write shape alone sets the ceiling; the read pattern does not change the argument.

Hardware. All-NVMe, which rules out spinning-disk excuses. 5 servers, 12 drives each, 3.5 TB per drive. That is 60 drives and 210 TB raw. MinIO auto-selects erasure sets of 15 (four sets across the five nodes), run at EC 15:7, so 8 data shards and 7 parity per set.

Step 1: the capacity view, which says "years"

This is the calculation that goes on a sizing slide.

- Usable ratio at EC 15:7: 8 data of 15 drives = 53%.
- Usable capacity: 210 TB x 0.53 = about 111 TB.
- Logical data: 15M x 10 KB = 150 GB/day.
- Runway: 111 TB / 150 GB per day = about 740 days, roughly 2 years.

Two years of retention on this box. Procurement signs off. This number is the trap.

Step 2: the honest on-disk view, which says "months"

A 10 KB object does not cost 10 KB of erasure-coded bytes. It costs one `x1.meta` on each of the 15 drives of its set. Each `x1.meta` rounds up to a 4 KiB XFS block, and each per-object subdirectory costs its own inode and directory block, on every drive. Rough per-drive cost is about 8 KiB, times 15 drives in the set:

- Effective on-disk per object: about 120 KiB for a 10 KB payload, roughly 12x inflation.
- Raw per day: 15M x 120 KiB = about 1.84 TB/day.
- Runway: 210 TB / 1.84 TB per day = about 114 days, under 4 months.

The "EC 15:7 = 1.9x overhead" figure from the sizing slide is off by roughly 6x on the real footprint, because it counts bytes and ignores the per-drive envelope, block rounding, and per-object structure multiplied by the 15-wide set. Still, four months sounds like time in hand. It is not.

Step 3: the scanner view, which says "weeks"

The binding operational constraint is not space, it is the scanner walk. Take a measured anchor from a cluster of this profile: about 48 million objects took 21 hours for a full scan at default speed, and scan time is close to linear in object count. Retention R days means a steady-state population of $N = 15M \times R$ objects (more when versioning is on; site replication forces versioning, which multiplies N by the average version count).

Retention	Objects (N)	Full scan, default	Full scan, fast (~4x)
14 days	210M	~3.8 days	~1 day
30 days	450M	~8 days	~2 days
90 days	1.35B	~24 days	~6 days
180 days	2.7B	~49 days	~12 days
365 days	5.5B	~99 days	~25 days

ILM expiry is scanner-driven. To hold a retention of R, the scanner must complete well inside R. At 90 days the scan takes three weeks or more, so expiry runs three weeks late, so the population overshoots

its target, so the next scan is slower still. The loop does not settle. Healing, bit-rot detection, and replication catch-up ride the same scanner, so all of them inherit the same multi-week lag. You are operating a platform whose self-repair SLA is measured in weeks, on hardware you provisioned expecting hours.

Healing is the part of that self-repair SLA that ends the argument in an operations review, so it deserves its own line. Scanner speed is a good proxy for healing speed: when a drive or a whole server fails, MinIO rebuilds the missing shards by walking the affected objects and reconstructing them one by one, the same per-object metadata work the scanner does, at the same rate. If a full pass takes weeks at this object count, then rebuilding a replaced drive takes weeks, and rebuilding a replaced server takes longer. A platform where bringing a replacement server back to full redundancy runs into months is not one you can operate. Through that whole window you sit below your parity target, so a second failure lands much closer to real data loss, and every maintenance and capacity plan that assumed a bounded rebuild time is wrong. Storage you cannot heal on a human timescale is not durable storage, whatever the parity math on the slide says.

Step 4: inode exhaustion, the punchline

Now count inodes per drive. Objects spread across 4 sets, each object writes two inodes (subdirectory plus `x1.meta`) on each of the 15 drives of its set, so per-drive inode count is about $N/2$.

- At 90 days: $1.35\text{B} / 2 =$ about 675 million inodes per drive.
- At 180 days: $2.7\text{B} / 2 =$ about 1.35 billion inodes per drive.
- At 365 days: $5.5\text{B} / 2 =$ about 2.7 billion inodes per drive.

A 3.5 TB XFS filesystem formatted with defaults (25% inode allocation cap, 512-byte inodes) tops out near 1.7 billion inodes. You cross that ceiling around 225 days, about 7.5 months, with roughly half the drive's bytes still free. The drive returns `ENOSPC` while `df` shows terabytes available. You can raise the inode cap, but read that sentence again: you are tuning your way around inode exhaustion on an NVMe drive that is half empty. That is the absurd result the capacity slide promised you would never see, and it exists only because the wide default set writes each object across 15 drives.

And 7.5 months is optimistic, because that count ignores the directory tree you are forced to build. You cannot pour billions of objects into a flat or shallow namespace. MinIO's own thresholds forbid it: keep a leaf directory under about 10,000 objects and any parent under about 50,000 subdirectories, or `readdir` and the scanner degrade. At 15 million objects a day, staying under those limits forces a deep key scheme, at least three directory levels of time and hash, something like `{YYYY}/{MM}/{DD}/{HH}/{hash}/uuid`. Every intermediate and leaf directory in that tree is itself an inode on every drive of the set, on top of the two inodes per object. The tree is not what breaks the inode budget: at this scale it adds a few million inodes per drive against billions of per-object inodes, well under one percent. What it breaks is the scanner. The walk now has to cross millions of directories to reach the objects, on top of the per-object metadata work, so the point where the platform becomes unmanageable is set by the scanner in days, not by the inode ceiling in months. Read the 7.5-month figure as a physical ceiling you never reach, not a retention you can plan for.

What the sizing actually concludes

Put the views side by side:

- The capacity slide says 2 years.
- Real footprint says the disks fill in under 4 months.

- Inodes exhaust around 7.5 months, and the deep prefix tree the object count forces makes even that optimistic.
- The scanner makes anything past a few days operationally unsafe, and site replication makes that worse.

The scan and everything it drives, ILM expiry, healing, bit-rot detection, and the re-push of pending replications, must complete within a few days or they fall behind ingestion. Site replication tightens the bound further: pending cross-site replications are re-pushed on the same scanner walk, so a multi-day scan means a multi-day replication lag and an RPO to match. To stay inside a few days even at fast speed, you cap the population near a couple hundred million objects, a retention of one to two weeks, not one to two years. At two weeks you are storing about 26 TB. **You bought 210 TB of NVMe and can operationally use around 12% of it, and less once replication is in the picture.** The rest is stranded, because each additional object costs scanner time and inodes, not just space, and adding more drives makes the erasure set wider, which makes every small object cost more, not less.

So the honest sizing output for "5 servers of 12 NVMe at 3.5 TB, 15 million objects/day at 10 KB, one-year retention" is not a capacity number. It is a rejection: this is not a sizing problem, it is a design problem. No amount of NVMe fixes it, because disk space was never the constraint.

Why "10x better than HDFS" still ends here

The two facts are not in tension. MinIO removed the HDFS RAM ceiling and pushed the metadata wall out by roughly an order of magnitude. That is exactly why teams reach for it, and why the blog's headline is defensible. But an order of magnitude on a wall is still a wall. HDFS stops at hundreds of millions of files because of one machine's memory. MinIO stops at low billions of objects because of inodes, `readdir`, a linear scanner, and an erasure set that, on small files, behaves like replication across its full width. The second number is bigger. It is not infinite.

The blog's real omission is the missing sentence: *object storage relocates the small-file cost, it does not delete it, and on a genuine LOSF workload you will meet the relocated cost, multiplied by the erasure-set width, long before you fill the disks.* Any sizing that prices a small-file MinIO platform in terabytes, and stops there, is measuring the wrong quantity.

What to do instead

The fix is never "tune the scanner." It is to stop making the object store hold one object per record.

- **Batch at the source.** Aggregate small writes upstream into larger objects (Parquet, Avro, tar, line-delimited JSON) before they reach MinIO. This is the single change that collapses object count by 100x to 1000x and makes the numbers above evaporate.
- **If a broker is already in the pipeline, carry the payload, not just a pointer.** When Kafka (or Pulsar, or any log-based broker) already sits in front of the store, put the actual object bytes in the message value, not only a reference to an object you also wrote to MinIO. A 10 KB file sits comfortably inside Kafka's default 1 MiB message limit, and the broker's log is built for exactly this: a high-rate stream of small records, appended sequentially, compressed, partitioned, retained, and replayable. A batching sink then compacts many messages into one large object for cold storage, so MinIO only ever sees large objects and the small-file problem never reaches it. Keep the claim-check pattern (write the blob, put only its key on the topic) for the genuinely large payloads, where inlining would strain broker memory and GC. Inline the small ones, reference the big ones.
- **Move the small-record path to a database engine.** A write-heavy, key-addressed, kilobyte-payload workload is a database workload. Cassandra or ScyllaDB fit it natively; Postgres fits it below about 100 million rows. Keep MinIO for the blobs, hold the pointer and the metadata in the

database. I go through this in detail in my earlier post, [MinIO and lots of small files: why it is not a database](#).

- **Size on objects, not bytes.** For any small-file platform, set the retention target from the scanner budget and the inode ceiling first, then check that the byte capacity is sufficient. Doing it in the other order produces the two-year slide that lasts two weeks.

MinIO earned its win over HDFS. Just do not read that win as a license to point a small-file firehose at it. The limit is real, it is quiet, and on the sizing above it is measured in days to weeks, not years.

Sizing an object store on bytes when the real constraint is object count and erasure-set width is the mistake that turns a two-year plan into a three-week incident. A [Platform Sizing review](#) catches it before the hardware is bought; a [Data Platform Performance Audit](#) explains it after it hits. [Book a 15-min intro call](#) to see which fits, or take one focused hour on the question with an [Expert Call](#).

Update 2026-09-29. This post is now part of [Object Storage for Data Platforms at Scale](#), the field guide to running object storage at scale.

Related posts

- [Object Storage for Data Platforms at Scale](#)
- [MinIO and lots of small files: why it's not a database](#)
- [MinIO on XFS: inode exhaustion and prefix design](#)
- [The hidden costs of using HDDs in on-premises MinIO deployments](#)

PDF snapshot of the public article. The online version is authoritative; links remain clickable.