

MinIO and Lots of Small Files: Why It's Not a Database, and Why That Matters

Julien Laurenceau | Published 2026-06-29 | Updated 2026-09-30

<https://pepitedata.com/minio-lots-of-small-files-losf/>

PepiteData | LinkedIn: Julien Laurenceau | X: @Julien_Lau



Millions of small objects on MinIO is the architecture review I run more often than any other. The [MinIO, S3 and object storage consulting](#) page describes what such a review covers.

MinIO has a narrow sweet spot: serving medium-to-large objects with S3-compatible semantics on commodity hardware, using erasure coding for durability. Push it into Lots of Small Files (LOSF) territory, millions or billions of objects under a few hundred KiB each, and you are operating outside its design envelope. The failure modes are structural, not tuning problems.

I have run MinIO at scale for years. I have also debugged production data loss caused by the interaction of tiering, small files, and the scanner's repair blind spot. This article is what I wish someone had handed me before I debugged it in production.

MinIO is not a database

MinIO is not a database: it is an object store, and that architectural line is the root of every LOSF problem even though it sounds obvious. A database, Cassandra, ScyllaDB, PostgreSQL, even SQLite, maintains at least one of these: an index that finds keys without a filesystem walk, read repair that fixes replicas on read, or compaction that merges tombstones away.

- **A global index.** You ask for a key; the database finds it in $O(\log n)$ or $O(1)$ time, not an $O(n)$ filesystem walk.
- **Read repair.** On every read, the database compares replicas and fixes inconsistencies. Cassandra calls this `read_repair_chance`; ScyllaDB does it by default.
- **Compaction.** Tombstones and obsolete versions are merged away in the background by a process that understands the data model.

MinIO has none of these.

No global index

MinIO has no global index: it translates S3 prefixes into real directories on an XFS filesystem. Each S3 object is a subdirectory containing one `xl.meta` file per drive of the erasure set. An S3 LIST operation is a filesystem walk: `readdir()` on the parent, `stat()` on every object subdirectory, `open() + read()` on each `xl.meta`.

This works fine when you have a few thousand objects per prefix. At 100,000 objects per directory, XFS switches its directory format from leaf to B+tree, and `readdir()` becomes much more expensive. At 250,000 entries, you are deep in B+tree territory and every LIST on that prefix consumes the IOPS budget on the drives of one erasure set while the rest of the cluster idles.

I measured the per-object cost on cold cache at ~ 2.5 IOPS per object per drive queried. A directory of one million objects costs ~ 2.5 million aggregate IOPS to list end-to-end. On NVMe that is tens of seconds; on HDD it is minutes. During that window, concurrent PUTs on the same erasure set see latency spikes and may time out.

No read repair

MinIO has no read repair: its erasure coding provides durability within a single cluster, and with EC:4 on a 16-drive set, you can lose up to 4 drives per set and still read. But the scanner that detects and repairs inconsistencies is **sampling-based**: `healObjectSelectProb` defaults to 1024, meaning only 1 object in 1024 is checked per scan pass. Sixteen scanner cycles are needed for full namespace coverage.

If a bit rots on a drive that happens not to be sampled for weeks, the object sits degraded until a client reads it and hits the quorum shortfall. Or a second drive fails and the object is gone.

Cassandra's read repair fixes this by comparing replicas on every read (when configured). MinIO has no equivalent. The scanner is a background maintenance sweep, not an anti-entropy mechanism. On a large namespace with small files, the gap between "degraded" and "unrecoverable" can be months of scanner lag.

No compaction

MinIO does not compact: in a versioning-enabled bucket, every DELETE creates a Delete Marker, a 0-byte object that becomes the "current" version and hides everything beneath it. The data is still there, consuming inodes, scanner cycles, and replication bandwidth, and Delete Markers accumulate without bound unless you configure ILM rules to expire them.

Versioning is required for site replication, so every multi-site MinIO deployment runs with versioning on. Without aggressive non-current-version expiry policies, a bucket of 100 million versioned objects with an average of 5 versions behaves, from the scanner's perspective, like a 500-million-object bucket.

The scanner becomes the bottleneck on LOSF

MinIO's background scanner becomes the bottleneck on a LOSF namespace, and it drives usage accounting, ILM expiry, replication retry, and heal sampling, so everything behind it degrades too. On a healthy namespace with well-partitioned prefixes, it keeps up. On a LOSF namespace, full coverage takes weeks to months and expiry, replication retries and heal sampling all lag behind it.

The scanner sleeps 1 ms between folders. That is negligible. The real cost is per-object metadata work: inode lookup plus `xl.meta` read on the primary drive, roughly 2.5 IOPS per object on cold cache.

Here is what that means at scale, assuming a well-designed 2-level hash partition (256 x 256 leaf directories, 10,000 objects each = 655 million objects):

- **10,000 objects per leaf**: a few seconds of scanner I/O per leaf on cold cache.
- **65,536 leaves**: cumulatively, **days per cycle**.
- **16 cycles for full coverage** (`dataUsageUpdateDirCycles`): **weeks to months** before the scanner has touched every object.

During those weeks:

- **ILM rules lag**. An object scheduled for expiry after 30 days may sit for an extra 3 weeks before the scanner notices.
- **Replication failures sit in FAILED state**. The active retry path (3 immediate retries plus MRF ring) catches transient errors in seconds to minutes. Once an object exhausts those, it stays FAILED until the next scanner sweep, which on a LOSF namespace is weeks away.
- **Heal sampling (1 in 1024) means degradation detection takes multiple full cycles**. Silent corruption can sit undetected for months.

The OSS scanner has no abandon path. It never skips a prefix, never times out. At 50,000 subdirectories per prefix it emits a `PrefixManyFolders` alert and keeps going. At 250,000 it forces an in-memory stats tree compaction and keeps going. There is no circuit breaker.

Why tiering makes it worse

Tiering makes it worse because objects under 256 KiB are **inlined** in `xl.meta`. The data lives inside the metadata file. When tiering transitions such an object to a cold tier, it moves an empty stub. The data stays on the hot tier inside `xl.meta`, consuming hot-tier capacity without any of the cost savings tiering was supposed to deliver.

Worse: I discovered and reproduced a critical bug where the scanner **cannot heal the metadata of transitioned objects**. Remove `xl.meta` from two drives of the erasure set, and the object disappears from LIST even though the actual data is intact on the cold tier. The scanner marks all transitioned objects as "grey" (unrecoverable) by default, making it impossible to distinguish normal state from genuine data loss.

MinIO shipped a fix for this in November 2024. But the fact that it shipped at all, with healing tests broken for over 6 months and no halt on releases, tells you where LOSF plus tiering sits in the test matrix.

When MinIO can handle LOSF (and when it cannot)

MinIO can handle LOSF only with a partitioned key layout that keeps leaf directories small, and it cannot handle flat namespaces at all. A 2-level hash partition keeps leaves under 10,000 objects and reaches 655 million objects per bucket; a 3-level partition adds headroom for sustained kHz writes. What breaks is everything else: flat namespaces, versioning without ILM, tiering of small inlined objects, and bitrot

scanning.

What works

A 2-level hash partition (`{hex2}/{hex2}/{uuid}.dat`) keeps leaf directories under 10,000 objects. At 256 x 256 leaves x 10,000 objects, you get 655 million objects per bucket. That is enough for many use cases. The scanner stays manageable (days per cycle, not weeks), and LIST calls on individual leaves are fast because each leaf fits in XFS leaf format and stays in page cache.

A 3-level partition (`{YYYY}/{MM}/{DD}/{HH}/{mm-slot}/{hash-2char}/{uuid}.dat`) handles sustained kHz write rates while keeping every leaf well under the 10,000 threshold.

What breaks

- **Flat namespaces.** One prefix, millions of objects. This is the typical "I'll just use S3 as a key-value store" anti-pattern. XFS B+tree `readdir()`, serial S3 pagination, scanner saturation: everything degrades at once.
- **Versioning without ILM.** Each version doubles the on-disk file count. Five versions average = 5x the scanner work, 5x the LIST cost, 5x the replication bandwidth.
- **Tiering with small inlined objects.** You pay for the tiering infrastructure but get no capacity savings, plus you inherit the metadata repair bug if you are not on a fixed version.
- **Bitrot scanning** (`bitrotscan=on`). This makes the scanner read the actual data shards of every object, not only `xl.meta`. On LOSF, this multiplies scanner I/O by orders of magnitude. It is typically not viable past a few hundred million objects.

The sizing ceiling

The sizing ceiling for MinIO is roughly 100 million objects per bucket: below that, nominal operation with well-distributed prefixes, and beyond it LIST, scanner and ILM degrade progressively until full scans take multiple days. Above 500 million objects the cluster is in a risk zone where maintenance operations become problematic, and above 1 billion the guidance is to use a database.

Object count per bucket	Behaviour
< 100 million	Nominal if prefixes are well distributed
100-500 million	Progressive degradation of LIST, scanner, ILM. Full scan can take multiple days.
> 500 million	Risk zone. Maintenance ops become problematic.
> 1 billion	Not recommended. Use a database.

These are practical ceilings from production experience and community reports, not documented limits. MinIO does not publish a hard maximum.

What to use instead

Use an indexed store instead of MinIO: the right one depends on your access pattern, and for a genuinely LOSF workload (millions to billions of small objects, high write throughput, need for indexing and compaction), MinIO is the wrong tool. Cassandra or ScyllaDB, SeaweedFS, PostgreSQL with BLOB storage, and Ceph RADOS are the realistic candidates. The common thread is an index that decouples finding an object from walking the filesystem.

- **Cassandra / ScyllaDB.** Built for exactly this. Global index, read repair, compaction, tunable consistency per query. ScyllaDB in particular handles high write throughput with predictable latency. The trade-off is operational complexity. You are now running a distributed database.
- **SeaweedFS.** A purpose-built distributed file system that separates metadata from data, with a dedicated volume server for small files and an efficient filer store. Designed for LOSF, not retrofitted.
- **PostgreSQL with BLOB storage.** If your access pattern is key-value and you do not need horizontal scale-out, a single PostgreSQL instance with bytea columns or large-object support gives you ACID, indexes, and compaction. Extension libraries like `pg_bigm` or `pg_trgm` add fuzzy search on keys.
- **Ceph (RADOS).** Ceph's object store maintains a cluster map and uses CRUSH for data placement. Its metadata architecture is different from MinIO's, but CephFS with adequate MDS sizing handles small files better than MinIO's filesystem-walk approach.

The common thread: all of these maintain some form of **index** that decouples "find the object" from "walk the filesystem." MinIO's decision to avoid a centralized metadata store is what makes it simple to operate at medium scale and what makes it break at LOSF scale. You cannot tune your way out of an architectural choice.

There is also a strategic reason to plan an exit. MinIO has been shifting features and development focus into its commercial AIStor product, and the community edition gets less of both. If you are betting a growing small-file workload on the open-source build, weigh where the vendor is putting its effort before you scale further.

If you are stuck with MinIO

If you are stuck with MinIO, the mitigations run in order of impact: audit the prefix layout, repartition until every leaf is under 10,000 objects, ban flat LIST, and add aggressive ILM. Sometimes you inherit a MinIO cluster full of small files and migration is not an option. Then schedule the scanner, watch the LIST IOPS to PUT IOPS ratio, and keep bitrotscan off on a LOSF namespace.

1. **Audit your prefix layout.** Run `find` on a MinIO node to identify leaf directories with >10,000 object subdirectories. Anything >50,000 is an immediate hazard.
2. **Repartition.** Add a hash layer or finer time slicing to bring every leaf under 10,000. A deep schema like `{YYYY}/{MM}/{DD}/{HH}/{mm-slot}/{hash-2char}/{uuid}.dat` handles sustained write rates while keeping leaves manageable.
3. **Ban flat LIST.** Once a bucket exceeds a few hundred thousand objects, "list the whole bucket" must be forbidden. Every LIST must carry a restrictive prefix.
4. **Add aggressive ILM.** Expire non-current versions after a bounded window. Enable `--expire-delete-marker` to clean up orphaned markers. Check version count per bucket: if `Versions / Objects > 2`, you have an accumulation problem.
5. **Schedule the scanner.** Run `scanner speed=fast` during off-peak hours if your workload has a diurnal pattern; return to default during peak. At fastest, the scanner completes ~1/16 of the namespace per minute but competes aggressively for disk I/O.
6. **Monitor the LIST IOPS / PUT IOPS ratio.** A rising ratio means LISTs are eating your write budget. The typical symptom: `%util` at 100% on one erasure set's drives while the rest of the cluster idles.
7. **Do not enable bitrotscan** on a LOSF namespace unless you have measured the I/O headroom and are confident. It is a major amplifier.

Bottom line

The bottom line is that MinIO is an object store, not a database: it does not maintain a global index, it does not do read repair, it does not compact, and its scanner is a background maintenance loop, not an anti-entropy mechanism. A filesystem with an S3 API bolted on is a poor database, even with erasure coding. Keep MinIO to medium-to-large objects and simple S3 operations.

If you are storing millions of small files, you are not using an object store. You are using a filesystem with an S3 API bolted on. And filesystems, even with erasure coding, are poor databases.

If you already run MinIO, keep it to the workload it fits: medium-to-large objects and simple S3 operations. Put your small files in a system built for them.

This article reflects production experience with MinIO Community Edition and AIStor. The tiering bug was reproduced on RELEASE.2024-10-29T16-01-48Z and acknowledged by MinIO in GitHub issue #20559. Versions and behaviours described may differ in later releases.

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 small files: when erasure coding becomes 15x replication](#)
- [MinIO on XFS: inode exhaustion and prefix design](#)
- [Stop using MinIO as a NoSQL database](#)

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