# Facebook's Tectonic Filesystem: Efficiency from Exascale

Satadru Pan[1], Theano Stavrinos[1,2], Yunqiao Zhang[1], Atul Sikaria[1], Pavel Zakharov[1], Abhinav Sharma[1], Shiva Shankar P[1], Mike Shuey[1], Richard Wareing[1], Monika Gangapuram[1], Guanglei Cao[1], Christian Preseau[1], Pratap Singh[1], Kestutis Patiejunas[1], JR Tipton[1], Ethan Katz-Bassett[3], and Wyatt Lloyd[2]

[1]*Facebook, Inc.,* [2]*Princeton University,* [3]*Columbia University*

## Abstract

Tectonic is Facebook's exabyte-scale distributed filesystem. Tectonic consolidates large tenants that previously used service-specific systems into general multitenant filesystem instances that achieve performance comparable to the specialized systems. The exabyte-scale consolidated instances enable better resource utilization, simpler services, and less operational complexity than our previous approach. This paper describes Tectonic's design, explaining how it achieves scalability, supports multitenancy, and allows tenants to specialize operations to optimize for diverse workloads. The paper also presents insights from designing, deploying, and operating Tectonic.

## 1 Introduction

Tectonic is Facebook's distributed filesystem. It currently serves around ten tenants, including blob storage and data warehouse, both of which store exabytes of data. Prior to Tectonic, Facebook's storage infrastructure consisted of a constellation of smaller, specialized storage systems. Blob storage was spread across Haystack [11] and f4 [34]. Data warehouse was spread across many HDFS instances [15].

The constellation approach was operationally complex, requiring many different systems to be developed, optimized, and managed. It was also inefficient, stranding resources in the specialized storage systems that could have been reallocated for other parts of the storage workload.

A Tectonic cluster scales to exabytes such that a single cluster can span an entire datacenter. The multi-exabyte capacity of a Tectonic cluster makes it possible to host several large tenants like blob storage and data warehouse on the same cluster, with each supporting hundreds of applications in turn. As an exabyte-scale multitenant filesystem, Tectonic provides operational simplicity and resource efficiency compared to federation-based storage architectures [8, 17], which assemble smaller petabyte-scale clusters.

Tectonic simplifies operations because it is a single system to develop, optimize, and manage for diverse storage needs. It is resource-efficient because it allows resource sharing among all cluster tenants. For instance, Haystack was the storage system specialized for new blobs; it bottlenecked on hard disk IO per second (IOPS) but had spare disk capacity. f4, which stored older blobs, bottlenecked on disk capacity but had spare IO capacity. Tectonic requires fewer disks to support the same workloads through consolidation and resource sharing.

In building Tectonic, we confronted three high-level challenges: scaling to exabyte-scale, providing performance isolation between tenants, and enabling tenant-specific optimizations. Exabyte-scale clusters are important for operational simplicity and resource sharing. Performance isolation and tenant-specific optimizations help Tectonic match the performance of specialized storage systems.

To scale metadata, Tectonic disaggregates the filesystem metadata into independently-scalable layers, similar to ADLS [42]. Unlike ADLS, Tectonic hash-partitions each metadata layer rather than using range partitioning. Hash partitioning effectively avoids hotspots in the metadata layer. Combined with Tectonic's highly scalable chunk storage layer, disaggregated metadata allows Tectonic to scale to exabytes of storage and billions of files.

Tectonic simplifies performance isolation by solving the isolation problem for groups of applications in each tenant with similar traffic patterns and latency requirements. Instead of managing resources among hundreds of applications, Tectonic only manages resources among tens of traffic groups.

Tectonic uses tenant-specific optimizations to match the performance of specialized storage systems. These optimizations are enabled by a client-driven microservice architecture that includes a rich set of client-side configurations for controlling how tenants interact with Tectonic. Data warehouse, for instance, uses Reed-Solomon (RS)-encoded writes to improve space, IO, and networking efficiency for its large writes. Blob storage, in contrast, uses a replicated quorum append protocol to minimize latency for its small writes and later RS-encodes them for space efficiency.

Tectonic has been hosting blob storage and data warehouse in single-tenant clusters for several years, completely replacing Haystack, f4, and HDFS. Multitenant clusters are being methodically rolled out to ensure reliability and avoid performance regressions.

Adopting Tectonic has yielded many operational and efficiency improvements. Moving data warehouse from HDFS onto Tectonic reduced the number of data warehouse clusters by 10×, simplifying operations from managing fewer clusters. Consolidating blob storage and data warehouse into multitenant clusters helped data warehouse handle traffic spikes with spare blob storage IO capacity. Tectonic manages these efficiency improvements while providing comparable or better performance than the previous specialized storage systems.

## 2 Facebook's Previous Storage Infrastructure

Before Tectonic, each major storage tenant stored its data in one or more specialized storage systems. We focus here on two large tenants, blob storage and data warehouse. We discuss each tenant's performance requirements, their prior storage systems, and why they were inefficient.

### 2.1 Blob Storage

Blob storage stores and serves **b**inary **l**arge **ob**jects. These may be media from Facebook apps (photos, videos, or message attachments) or data from internal applications (core dumps, bug reports). Blobs are immutable and opaque. They vary in size from several kilobytes for small photos to several megabytes for high-definition video chunks [34]. Blob storage expects low-latency reads and writes as blobs are often on path for interactive Facebook applications [29].

**Haystack and f4.** Before Tectonic, blob storage consisted of two specialized systems, Haystack and f4. Haystack handled "hot" blobs with a high access frequency [11]. It stored data in replicated form for durability and fast reads and writes. As Haystack blobs aged and were accessed less frequently, they were moved to f4, the "warm" blob storage [34]. f4 stored data in RS-encoded form [43], which is more space-efficient but has lower throughput because each blob is directly accessible from two disks instead of three in Haystack. f4's lower throughput was acceptable because of its lower request rate.

However, separating hot and warm blobs resulted in poor resource utilization, a problem exacerbated by hardware and blob storage usage trends. Haystack's ideal effective replication factor was 3.6× (i.e., each logical byte is replicated 3×, with an additional 1.2× overhead for RAID-6 storage [19]). However, because IOPS per hard drive has remained steady as drive density has increased, IOPS per terabyte of storage capacity has declined over time.

As a result, Haystack became IOPS-bound; extra hard drives had to be provisioned to handle the high IOPS load of hot blobs. The spare disk capacity resulted in Haystack's effective replication factor increasing to 5.3×. In contrast, f4 had an effective replication factor of 2.8× (using RS(10,4) encoding in two different datacenters). Furthermore, blob storage usage shifted to more ephemeral media that was stored in Haystack but deleted before moving to f4. As a result, an increasing share of the total blob data was stored at Haystack's high effective replication factor.

Finally, since Haystack and f4 were separate systems, each *stranded* resources that could not be shared with other systems. Haystack overprovisioned storage to accommodate peak IOPS, whereas f4 had an abundance of IOPS from storing a large volume of less frequently-accessed data. Moving blob storage to Tectonic harvested these stranded resources and resulted in an effective replication factor of ~2.8×.

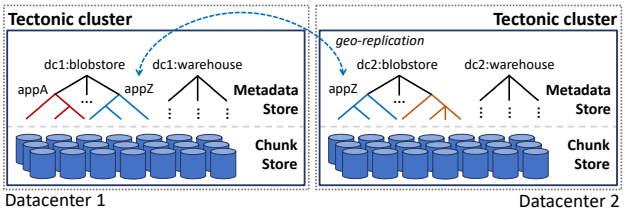



**Figure 1: Tectonic provides durable, fault-tolerant storage inside a datacenter. Each tenant has one or more separate namespaces. Tenants implement geo-replication.**

### 2.2 Data Warehouse

Data warehouse provides storage for data analytics. Data warehouse applications store objects like massive map-reduce tables, snapshots of the social graph, and AI training data and models. Multiple compute engines, including Presto [3], Spark [10], and AI training pipelines [4] access this data, process it, and store derived data. Warehouse data is partitioned into datasets that store related data for different product groups like Search, Newsfeed, and Ads.

Data warehouse storage prioritizes read and write throughput over latency, since data warehouse applications often batch-process data. Data warehouse workloads tend to issue larger reads and writes than blob storage, with reads averaging multiple megabytes and writes averaging tens of megabytes.

**HDFS for data warehouse storage.** Before Tectonic, data warehouse used the Hadoop Distributed File System (HDFS) [15, 50]. However, HDFS clusters are limited in size because they use a single machine to store and serve metadata.

As a result, we needed tens of HDFS clusters per datacenter to store analytics data. This was operationally inefficient; every service had to be aware of data placement and movement among clusters. Single data warehouse datasets are often large enough to exceed a single HDFS cluster's capacity. This complicated compute engine logic, since related data was often split among separate clusters.

Finally, distributing datasets among the HDFS clusters created a two-dimensional bin-packing problem. The packing of datasets into clusters had to respect each cluster's capacity constraints and available throughput. Tectonic's exabyte scale eliminated the bin-packing and dataset-splitting problems.

## 3 Architecture and Implementation

This section describes the Tectonic architecture and implementation, focusing on how Tectonic achieves exabyte-scale single clusters with its scalable chunk and metadata stores.

### 3.1 Tectonic: A Bird's-Eye View

A *cluster* is the top-level Tectonic deployment unit. Tectonic clusters are datacenter-local, providing durable storage that is resilient to host, rack, and power domain failures. Tenants can build geo-replication on top of Tectonic for protection against datacenter failures (Figure 1).

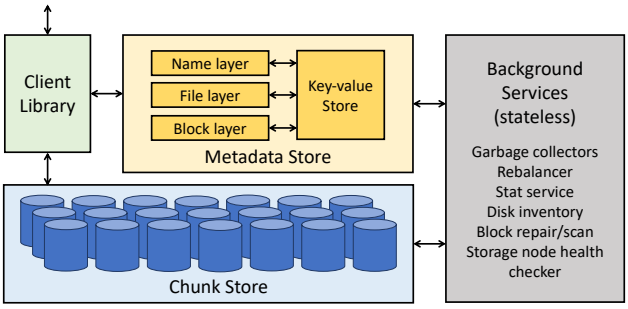



**Figure 2: Tectonic architecture. Arrows indicate network calls. Tectonic stores filesystem metadata in a key-value store. Apart from the Chunk and Metadata Stores, all components are stateless.**

A Tectonic cluster is made up of storage nodes, metadata nodes, and stateless nodes for background operations. The Client Library orchestrates remote procedure calls to the metadata and storage nodes. Tectonic clusters can be very large: a single cluster can serve the storage needs of all tenants in a single datacenter.

Tectonic clusters are *multitenant*, supporting around ten tenants on the same storage fabric (§4). *Tenants* are distributed systems that will never share data with one another; tenants include blob storage and data warehouse. These tenants in turn serve hundreds of *applications*, including Newsfeed, Search, Ads, and internal services, each with varying traffic patterns and performance requirements.

Tectonic clusters support any number of arbitrarily-sized *namespaces*, or filesystem directory hierarchies, on the same storage and metadata components. Each tenant in a cluster typically owns one namespace. Namespace sizes are limited only by the size of the cluster.

Applications interact with Tectonic through a hierarchical filesystem API with append-only semantics, similar to HDFS [15]. Unlike HDFS, Tectonic APIs are configurable at runtime, rather than being pre-configured on a per-cluster or per-tenant basis. Tectonic tenants leverage this flexibility to match the performance of specialized storage systems (§4).

**Tectonic components.** Figure 2 shows the major components of Tectonic. The foundation of a Tectonic cluster is the *Chunk Store* (§3.2), a fleet of storage nodes which store and access data chunks on hard drives.

On top of the Chunk Store is the *Metadata Store* (§3.3), which consists of a scalable key-value store and stateless metadata services that construct the filesystem logic over the key-value store. Their scalability enables Tectonic to store exabytes of data.

Tectonic is a client-driven microservices-based system, a design that enables tenant-specific optimizations. The Chunk and Metadata Stores each run independent services to handle read and write requests for data and metadata. These services are orchestrated by the *Client Library* (§3.4); the library converts clients' filesystem API calls into RPCs to Chunk and Metadata Store services.

Finally, each cluster runs stateless background services to maintain cluster consistency and fault tolerance (§3.5).

## 3.2 Chunk Store: Exabyte-Scale Storage

The Chunk Store is a flat, distributed object store for *chunks*, the unit of data storage in Tectonic. Chunks make up blocks, which in turn make up Tectonic files.

The Chunk Store has two features that contribute to Tectonic's scalability and ability to support multiple tenants. First, the Chunk Store is flat; the number of chunks stored grows linearly with the number of storage nodes. As a result, the Chunk Store can scale to store exabytes of data. Second, it is oblivious to higher-level abstractions like blocks or files; these abstractions are constructed by the Client Library using the Metadata Store. Separating data storage from filesystem abstractions simplifies the problem of supporting good performance for a diversity of tenants on one storage cluster (§5). This separation means reading to and writing from storage nodes can be specialized to tenants' performance needs without changing filesystem management.

**Storing chunks efficiently.** Individual chunks are stored as files on a cluster's storage nodes, which each run a local instance of XFS [26]. Storage nodes expose core IO APIs to get, put, append to, and delete chunks, along with APIs for listing chunks and scanning chunks. Storage nodes are responsible for ensuring that their own local resources are shared fairly among Tectonic tenants (§4).

Each storage node has 36 hard drives for storing chunks [5]. Each node also has a 1 TB SSD, used for storing XFS metadata and caching hot chunks. Storage nodes run a version of XFS that stores local XFS metadata on flash [47]. This is particularly helpful for blob storage, where new blobs are written as appends, updating the chunk size. The SSD hot chunk cache is managed by a cache library which is flash endurance-aware [13].

**Blocks as the unit of durable storage.** In Tectonic, blocks are a logical unit that hides the complexity of raw data storage and durability from the upper layers of the filesystem. To the upper layers, a block is an array of bytes. In reality, blocks are composed of chunks which together provide block durability. Tectonic provides per-block durability to allow tenants to tune the tradeoff between storage capacity, fault tolerance, and performance. Blocks are either Reed-Solomon encoded [43] or replicated for durability. For RS$(r,k)$ encoding, the block data is split into $r$ equal chunks (potentially by padding the data), and $k$ parity chunks are generated from the data chunks. For replication, data chunks are the same size as the block and multiple copies are created. Chunks in a block are stored in different fault domains (e.g., different racks) for fault tolerance. Background services repair damaged or lost chunks to maintain durability (§3.5).

| Layer | Key | Value | Sharded by | Mapping |
|---|---|---|---|---|
| Name | (`dir_id`, *subdirname*) | `subdir_info`, `subdir_id` | `dir_id` | dir → list of subdirs (expanded) |
| | (`dir_id`, *filename*) | `file_info`, `file_id` | `dir_id` | dir → list of files (expanded) |
| File | (`file_id`, `blk_id`) | `blk_info` | `file_id` | file → list of blocks (expanded) |
| Block | `blk_id` | list<`disk_id`> | `blk_id` | block → list of disks (i.e., chunks) |
| | (`disk_id`, `blk_id`) | `chunk_info` | `blk_id` | disk → list of blocks (expanded) |

**Table 1: Tectonic's layered metadata schema. *dirname* and *filename* are application-exposed strings. `dir_id`, `file_id`, and `block_id` are internal object references. Most mappings are expanded for efficient updating.**

## 3.3 Metadata Store: Naming Exabytes of Data

Tectonic's Metadata Store stores the filesystem hierarchy and the mapping of blocks to chunks. The Metadata Store uses a fine-grained partitioning of filesystem metadata for operational simplicity and scalability. Filesystem metadata is first *disaggregated*, meaning the naming, file, and block layers are logically separated. Each layer is then hash partitioned (Table 1). As we describe in this section, scalability and load balancing come for free with this design. Careful handling of metadata operations preserves filesystem consistency despite the fine-grained metadata partitioning.

**Storing metadata in a key-value store for scalability and operational simplicity.** Tectonic delegates filesystem metadata storage to ZippyDB [6], a linearizable, fault-tolerant, sharded key-value store. The key-value store manages data at the shard granularity: all operations are scoped to a shard, and shards are the unit of replication. The key-value store nodes internally run RocksDB [23], a SSD-based single-node key-value store, to store shard replicas. Shards are replicated with Paxos [30] for fault tolerance. Any replica can serve reads, though reads that must be strongly consistent are served by the primary. The key-value store does not provide cross-shard transactions, limiting certain filesystem metadata operations.

Shards are sized so that each metadata node can host several shards. This allows shards to be redistributed in parallel to new nodes in case a node fails, reducing recovery time. It also allows granular load balancing; the key-value store will transparently move shards to control load on each node.

**Filesystem metadata layers.** Table 1 shows the filesystem metadata layers, what they map, and how they are sharded. The Name layer maps each directory to its sub-directories and/or files. The File layer maps file objects to a list of blocks. The Block layer maps each block to a list of disk (i.e., chunk) locations. The Block layer also contains the reverse index of disks to the blocks whose chunks are stored on that disk, used for maintenance operations. Name, File, and Block layers are hash-partitioned by directory, file, and block IDs, respectively.

As shown in Table 1, the Name and File layer and disk to block list maps are *expanded*. A key mapped to a list is expanded by storing each item in the list as a key, prefixed by the true key. For example, if directory `d1` contains files *foo* and *bar*, we store two keys (`d1`, *foo*) and (`d1`, *bar*) in `d1`'s Name shard. Expanding allows the contents of a key to be modified without reading and then writing the entire list. In a filesystem where mappings can be very large, e.g., directories may contain millions of files, expanding significantly reduces the overhead of some metadata operations such as file creation and deletion. The contents of a expanded key are listed by doing a prefix scan over keys.

**Fine-grained metadata partitioning to avoid hotspots.** In a filesystem, directory operations often cause hotspots in metadata stores. This is particularly true for data warehouse workloads where related data is grouped into directories; many files from the same directory may be read in a short time, resulting in repeated accesses to the directory.

Tectonic's layered metadata approach naturally avoids hotspots in directories and other layers by separating searching and listing directory contents (Name layer) from reading file data (File and Block layers). This is similar to ADLS's separation of metadata layers [42]. However, ADLS range-partitions metadata layers whereas Tectonic hash-partitions layers. Range partitioning tends to place related data on the same shard, e.g., subtrees of the directory hierarchy, making the metadata layer prone to hotspots if not carefully sharded.

We found that hash partitioning effectively load-balances metadata operations. For example, in the Name layer, the immediate directory listing of a single directory is always stored in a single shard. But listings of two subdirectories of the same directory will likely be on separate shards. In the Block layer, block locator information is hashed among shards, independent of the blocks' directory or file. Around two-thirds of metadata operations in Tectonic are served by the Block layer, but hash partitioning ensures this traffic is evenly distributed among Block layer shards.

**Caching sealed object metadata to reduce read load.** Metadata shards have limited available throughput, so to reduce read load, Tectonic allows blocks, files, and directories to be *sealed*. Directory sealing does not apply recursively, it only prevents adding objects in the immediate level of the directory. The contents of sealed filesystem objects cannot change; their metadata can be cached at metadata nodes and at clients without compromising consistency. The exception is the block-to-chunk mapping; chunks can migrate among disks, invalidating the Block layer cache. A stale Block layer cache can be detected during reads, triggering a cache refresh.

**Providing consistent metadata operations.** Tectonic relies on the key-value store's strongly-consistent operations and atomic read-modify-write in-shard transactions for strongly-consistent same-directory operations. More specifically, Tectonic guarantees read-after-write consistency for data operations (e.g., appends, reads), file and directory operations involving a single object (e.g., create, list), and move operations where the source and destination are in the same parent directory. Files in a directory reside in the directory's shard (Table 1), so metadata operations like file create, delete, and moves within a parent directory are consistent.

The key-value store does not support consistent cross-shard transactions, so Tectonic provides non-atomic cross-directory move operations. Moving a directory to another parent directory on a different shard is a two-phase process. First, we create a link from the new parent directory, and then delete the link from the previous parent. The moved directory keeps a backpointer to its parent directory to detect pending moves. This ensures only one move operation is active for a directory at a time. Similarly, cross directory file moves involve copying the file and deleting it from the source directory. The copy step creates a new file object with the underlying blocks of the source file, avoiding data movement.

In the absence of cross-shard transactions, multi-shard metadata operations on the same file must be carefully implemented to avoid race conditions. An example of such a race condition is when a file named *f1* in directory `d` is renamed to *f2*. Concurrently, a new file with the same name is created, where creates overwrite existing files with the same name. The metadata layer and shard lookup key (*shard*(`x`)) are listed for each step in parentheses.

A file rename has the following steps:

R1: get file ID `fid` for *f1* (Name, *shard*(`d`))

R2: add *f2* as an owner of *fid* (File, *shard*(`fid`))

R3: create the mapping *f2* $\rightarrow$ `fid` and delete *f1* $\rightarrow$ `fid` in an atomic transaction (Name, *shard*(`d`))

A file create with overwriting has the following steps:

C1: create new file ID `fid_new` (File, *shard*(`fid_new`))

C2: map *f1* $\rightarrow$ `fid_new`; delete *f1* $\rightarrow$ `fid` (Name, *shard*(`d`))

Interleaving the steps in these transactions may leave the filesystem in an inconsistent state. If steps C1 and C2 are executed after R1 but before R3, then R3 will erase the newly-created mapping from the create operation. Rename step R3 uses a within-shard transaction to ensure that the file object pointed to by *f1* has not been modified since R1.

## 3.4 Client Library

The Tectonic Client Library orchestrates the Chunk and Metadata Store services to expose a filesystem abstraction to applications, which gives applications per-operation control over how to configure reads and writes. Moreover, the Client Library executes reads and writes at the chunk granularity, the finest granularity possible in Tectonic. This gives the Client Library nearly free reign to execute operations in the most performant way possible for applications, which might have different workloads or prefer different tradeoffs (§5).

The Client Library replicates or RS-encodes data and writes chunks directly to the Chunk Store. It reads and reconstructs chunks from the Chunk Store for the application. The Client Library consults the Metadata Store to locate chunks, and updates the Metadata Store for filesystem operations.

**Single-writer semantics for simple, optimizable writes.** Tectonic simplifies the Client Library's orchestration by allowing a single writer per file. Single-writer semantics avoids the complexity of serializing writes to a file from multiple writers. The Client Library can instead write directly to storage nodes in parallel, allowing it to replicate chunks in parallel and to hedge writes (§5). Tenants needing multiple-writer semantics can build serialization semantics on top of Tectonic.

Tectonic enforces single-writer semantics with a write token for every file. Any time a writer wants to add a block to a file, it must include a matching token for the metadata write to succeed. A token is added in the file metadata when a process opens a file for appending, which subsequent writes must include to update file metadata. If a second process attempts to open the file, it will generate a new token and overwrite the first process's token, becoming the new, and only, writer for the file. The new writer's Client Library will seal any blocks opened by the previous writer in the open file call.

## 3.5 Background Services

Background services maintain consistency between metadata layers, maintain durability by repairing lost data, rebalance data across storage nodes, handle rack drains, and publish statistics about filesystem usage. Background services are layered similar to the Metadata Store, and they operate on one shard at a time. Figure 2 lists important background services.

A garbage collector between each metadata layer cleans up (acceptable) metadata inconsistencies. Metadata inconsistencies can result from failed multi-step Client Library operations. Lazy object deletion, a real-time latency optimization that marks deleted objects at delete time without actually removing them, also causes inconsistencies.

A rebalancer and a repair service work in tandem to relocate or delete chunks. The rebalancer identifies chunks that need to be moved in response to events like hardware failure, added storage capacity, and rack drains. The repair service handles the actual data movement by reconciling the chunk list to the disk-to-block map for every disk in the system. To scale horizontally, the repair service works on a per-Block layer shard, per-disk basis, enabled by the reverse index mapping disks to blocks (Table 1).

**Copysets at scale.** Copysets are combinations of disks that provide redundancy for the same block (e.g., a copyset for an RS(10,4)-encoded block consists of 14 disks) [20]. Having too many copysets risks data unavailability if there is an unexpected spike in disk failures. On the other hand, having too

few copysets results in high reconstruction load to peer disks when one disk fails, since they share many chunks.

The Block Layer and the rebalancer service together attempt to maintain a fixed copyset count that balances unavailability and reconstruction load. They each keep in memory about one hundred consistent shuffles of all the disks in the cluster. The Block Layer forms copysets from contiguous disks in a shuffle. On a write, the Block Layer gives the Client Library a copyset from the shuffle corresponding to that block ID. The rebalancer service tries to keep the block's chunks in the copyset specified by that block's shuffle. Copysets are best-effort, since disk membership in the cluster changes constantly.

## 4 Multitenancy

Providing comparable performance for tenants as they move from individual, specialized storage systems to a consolidated filesystem presents two challenges. First, tenants must share resources while giving each tenant its fair share, i.e., at least the same resources it would have in a single-tenant system. Second, tenants should be able to optimize performance as in specialized systems. This section describes how Tectonic supports resource sharing with a clean design that maintains operational simplicity. Section 5 describes how Tectonic's tenant-specific optimizations allow tenants to get performance comparable to specialized storage systems.

### 4.1 Sharing Resources Effectively

As a shared filesystem for diverse tenants across Facebook, Tectonic needs to manage resources effectively. In particular, Tectonic needs to provide approximate (weighted) fair sharing of resources among tenants and performance isolation between tenants, while elastically shifting resources among applications to maintain high resource utilization. Tectonic also needs to distinguish latency-sensitive requests to avoid blocking them behind large requests.

**Types of resources.** Tectonic distinguishes two types of resources: non-ephemeral and ephemeral. Storage capacity is the *non-ephemeral* resource. It changes slowly and predictably. Most importantly, once allocated to a tenant, it cannot be given to another tenant. Storage capacity is managed at the tenant granularity. Each tenant gets a predefined capacity quota with strict isolation, i.e., there is no automatic elasticity in the space allocated to different tenants. Reconfiguring storage capacity between tenants is done manually. Reconfiguration does not cause downtime, so in case of an urgent capacity crunch, it can be done immediately. Tenants are responsible for distributing and tracking storage capacity among their applications.

*Ephemeral* resources are those where demand changes from moment to moment, and allocation of these resources can change in real time. Storage IOPS capacity and metadata query capacity are two ephemeral resources. Because ephemeral resource demand changes quickly, these resources need finer-grained real-time automated management to ensure they are shared fairly, tenants are isolated from one another, and resource utilization is high. For the rest of this section, we describe how Tectonic shares ephemeral resources effectively.

**Distributing ephemeral resources among and within tenants.** Ephemeral resource sharing is challenging in Tectonic because not only are tenants diverse, but each tenant serves many applications with varied traffic patterns and performance requirements. For example, blob storage includes production traffic from Facebook users and background garbage collection traffic. Managing ephemeral resources at the tenant granularity would be too coarse to account for the varied workloads and performance requirements within a tenant. On the other hand, because Tectonic serves hundreds of applications, managing resources at the application granularity would be too complex and resource-intensive.

Ephemeral resources are therefore managed within each tenant at the granularity of groups of applications. These application groups, called *TrafficGroups*, reduce the cardinality of the resource sharing problem, reducing the overhead of managing multitenancy. Applications in the same TrafficGroup have similar resource and latency requirements. For example, one TrafficGroup may be for applications generating background traffic while another is for applications generating production traffic. Tectonic supports around 50 TrafficGroups per cluster. Each tenant may have a different number of TrafficGroups. Tenants are responsible for choosing the appropriate TrafficGroup for each of their applications. Each TrafficGroup is in turn assigned a *TrafficClass*. A TrafficGroup's TrafficClass indicates its latency requirements and decides which requests should get spare resources. The TrafficClasses are Gold, Silver, and Bronze, corresponding to latency-sensitive, normal, and background applications. Spare resources are distributed according to TrafficClass priority within a tenant.

Tectonic uses tenants and TrafficGroups along with the notion of TrafficClass to ensure isolation and high resource utilization. That is, tenants are allocated their fair share of resources; within each tenant, resources are distributed by TrafficGroup and TrafficClass. Each tenant gets a guaranteed quota of the cluster's ephemeral resources, which is subdivided between a tenant's TrafficGroups. Each TrafficGroup gets its guaranteed resource quota, which provides isolation between tenants as well as isolation between TrafficGroups.

Any ephemeral resource surplus within a tenant is shared with its own TrafficGroups by descending TrafficClass. Any remaining surplus is given to TrafficGroups in other tenants by descending TrafficClass. This ensures spare resources are used by TrafficGroups of the same tenant first before being distributed to other tenants. When one TrafficGroup uses resources from another TrafficGroup, the resulting traffic gets the minimum TrafficClass of the two TrafficGroups. This ensures the overall ratio of traffic of different classes does not change based on resource allocation, which ensures the node can meet the latency profile of the TrafficClass.

**Enforcing global resource sharing.** The Client Library uses a rate limiter to achieve the aforementioned elasticity. The rate limiter uses high-performance, near-realtime distributed counters to track the demand for each tracked resource in each tenant and TrafficGroup in the last small time window. The rate limiter implements a modified leaky bucket algorithm. An incoming request increments the demand counter for the bucket. The Client Library then checks for spare capacity in its own TrafficGroup, then other TrafficGroups in the same tenant, and finally other tenants, adhering to TrafficClass priority. If the client finds spare capacity, the request is sent to the backend. Otherwise, the request is delayed or rejected depending on the request's timeout. Throttling requests at clients puts backpressure on clients before they make a potentially wasted request.

**Enforcing local resource sharing.** The client rate limiter ensures approximate global fair sharing and isolation. Metadata and storage nodes also need to manage resources to avoid local hotspots. Nodes provide fair sharing and isolation with a weighted round-robin (WRR) scheduler that provisionally skips a TrafficGroup's turn if it will exceed its resource quota. In addition, storage nodes need to ensure that small IO requests (e.g., blob storage operations) do not see higher latency from colocation with large, spiky IO requests (e.g., data warehouse operations). Gold TrafficClass requests can miss their latency targets if they are blocked behind lower-priority requests on storage nodes.

Storage nodes use three optimizations to ensure low latency for Gold TrafficClass requests. First, the WRR scheduler provides a greedy optimization where a request from a lower TrafficClass may cede its turn to a higher TrafficClass if the request will have enough time to complete after the higher-TrafficClass request. This helps prevent higher-TrafficClass requests from getting stuck behind a lower-priority request. Second, we limit how many non-Gold IOs may be in flight for every disk. Incoming non-Gold traffic is blocked from scheduling if there are any pending Gold requests and the non-Gold in-flight limit has been reached. This ensures the disk is not busy serving large data warehouse IOs while blob storage requests are waiting. Third, the disk itself may re-arrange the IO requests, i.e., serve a non-Gold request before an earlier Gold request. To manage this, Tectonic stops scheduling non-Gold requests to a disk if a Gold request has been pending on that disk for a threshold amount of time. These three techniques combined effectively maintain the latency profile of smaller IOs, even when outnumbered by larger IOs.

## 4.2 Multitenant Access Control

Tectonic follows common security principles to ensure that all communications and dependencies are secure. Tectonic additionally provides coarse access control between tenants (to prevent one tenant from accessing another's data) and fine-grained access control within a tenant. Access control must be enforced at each layer of Tectonic, since the Client Library talks to each layer directly. Since access control is on path for every read and write, it must also be lightweight.

Tectonic uses a token-based authorization mechanism that includes which resources can be accessed with the token [31]. An authorization service authorizes top-level client requests (e.g., opening a file), generating an authorization token for the next layer in the filesystem; each subsequent layer likewise authorizes the next layer. The token's payload describes the resource to which access is given, enabling granular access control. Each layer verifies the token and the resource indicated in the payload entirely in memory; verification can be performed in tens of microseconds. Piggybacking token-passing on existing protocols reduces the access control overhead.

# 5 Tenant-Specific Optimizations

Tectonic supports around ten tenants in the same shared filesystem, each with specific performance needs and workload characteristics. Two mechanisms permit tenant-specific optimizations. First, clients have nearly full control over how to configure an application's interactions with Tectonic; the Client Library manipulates data at the chunk level, the finest possible granularity (§3.4). This Client Library-driven design enables Tectonic to execute operations according to the application's performance needs.

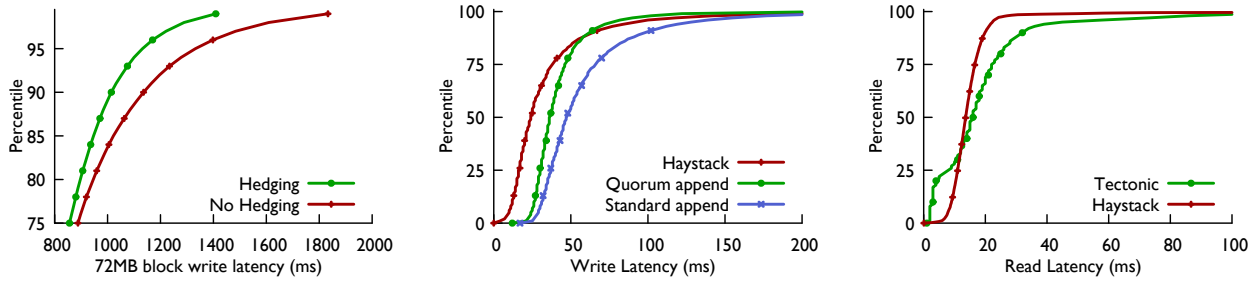
Second, clients enforce configurations on a per-call basis. Many other filesystems bake configurations into the system or apply them to entire files or namespaces. For example, HDFS configures durability per directory [7], whereas Tectonic configures durability per block write. Per-call configuration is enabled by the scalability of the Metadata Store: the Metadata Store can easily handle the increased metadata for this approach. We next describe how data warehouse and blob storage leverage per-call configurations for efficient writes.

## 5.1 Data Warehouse Write Optimizations

A common pattern in data warehouse workloads is to write data once that will be read many times later. For these workloads, the file is visible to readers only once the creator closes the file. The file is then immutable for its lifetime. Because the file is only read after it is written completely, applications prioritize lower file write time over lower append latency.

**Full-block, RS-encoded asynchronous writes for space, IO, and network efficiency.** Tectonic uses the write-once-read-many pattern to improve IO and network efficiency, while minimizing total file write time. The absence of partial file reads in this pattern allows applications to buffer writes up to the block size. Applications then RS-encode blocks in memory and write the data chunks to storage nodes. Long-lived data is typically RS(9,6) encoded; short-lived data, e.g., map-reduce shuffles, is typically RS(3,3)-encoded.

Writing RS-encoded full blocks saves storage space, network bandwidth, and disk IO over replication. Storage and bandwidth are lower because less total data is written. Disk IO is lower because disks are more efficiently used. More




**(a) Data warehouse full block writes** **(b) Blob storage write latency** **(c) Blob storage read latency**

**Figure 3: Tail latency optimizations in Tectonic. (a) shows the improvement in data warehouse tail latency from hedged quorum writes (72MB blocks) in a test cluster with ~80% load. (b) and (c) show Tectonic blob storage write latency (with and without quorum appends) and read latency compared to Haystack.**

IOPS are needed to write chunks to 15 disks in RS(9,6), but each write is small and the total amount of data written is much smaller than with replication. This results in more efficient disk IO because block sizes are large enough that disk bandwidth, not IOPS, is the bottleneck for full-block writes.

The write-once-read-many pattern also allows applications to write the blocks of a file asynchronously in parallel, which decreases the file write latency significantly. Once the blocks of the file are written, the file metadata is updated all together. There is no risk of inconsistency with this strategy because a file is only visible once it is completely written.

**Hedged quorum writes to improve tail latency.** For full-block writes, Tectonic uses a variant of quorum writing which reduces tail latency without any additional IO. Instead of sending the chunk write payload to extra nodes, Tectonic first sends reservation requests ahead of the data and then writes the chunks to the first nodes to accept the reservation. The reservation step is similar to hedging [22], but it avoids data transfers to nodes that would reject the request because of lack of resources or because the requester has exceeded its resource share on that node (§4).

As an example, to write a RS(9,6)-encoded block, the Client Library sends a reservation request to 19 storage nodes in different failure domains, four more than required for the write. The Client Library writes the data and parity chunks to the first 15 storage nodes that respond to the reservation request. It acknowledges the write to the client as soon as a quorum of 14 out of 15 nodes return success. If the 15th write fails, the corresponding chunk is repaired offline.

The hedging step is more effective when the cluster is highly loaded. Figure 3a shows ~20% improvement in 99th percentile latency for RS(9,6) encoded, 72 MB full-block writes, in a test cluster with 80% throughput utilization.

## 5.2 Blob Storage Optimizations

Blob storage is challenging for filesystems because of the quantity of objects that need to be indexed. Facebook stores tens of trillions of blobs. Tectonic manages the size of blob storage metadata by storing many blobs together into log-structured files, where new blobs are appended at the end of a file. Blobs are located with a map from blob ID to the location of the blob in the file.

Blob storage is also on path for many user requests, so low latency is desirable. Blobs are usually much smaller than Tectonic blocks (§2.1). Blob storage therefore writes new blobs as small, replicated partial block appends for low latency. The partial block appends need to be read-after-write consistent so blobs can be read immediately after successful upload. However, replicated data uses more disk space than full-block RS-encoded data.

**Consistent partial block appends for low latency.** Tectonic uses *partial block quorum appends* to enable durable, low-latency, consistent blob writes. In a quorum append, the Client Library acknowledges a write after a subset of storage nodes has successfully written the data to disk, e.g., two nodes for three-way replication. The temporary decrease of durability from a quorum write is acceptable because the block will soon be reencoded and because blob storage writes a second copy to another datacenter.

The challenge with partial block quorum appends is that straggler appends could leave replica chunks at different sizes. Tectonic maintains consistency by carefully controlling who can append to a block and when appends are made visible. Blocks can only be appended to by the writer that created the block. Once an append completes, Tectonic commits the post-append block size and checksum to the block metadata before acknowledging the partial block quorum append.

This ordering of operations with a single appender provides consistency. If block metadata reports a block size of $S$, then all preceeding bytes in the block were written to at least two storage nodes. Readers will be able to access data in the block up to offset $S$. Similarly, any writes acknowledged to the application will have been updated in the block metadata and so will be visible to future reads. Figures 3b and 3c demonstrate that Tectonic's blob storage read and write latency is comparable to Haystack, validating that Tectonic's generality does

| Capacity | Used bytes | Files | Blocks | Storage Nodes |
|---|---|---|---|---|
| 1590 PB | 1250 PB | 10.7 B | 15 B | 4208 |

**Table 2: Statistics from a multitenant Tectonic production cluster. File and block counts are in billions.**

not have a significant performance cost.

**Reencoding blocks for storage efficiency.** Directly RS-encoding small partial-block appends would be IO-inefficient. Small disk writes are IOPS-bound and RS-encoding results in many more IOs (e.g, 14 IOs with RS(10, 4) instead of 3). Instead of RS-encoding after each append, the Client Library reencodes the block from replicated form to RS(10,4) encoding once the block is sealed. Reencoding is IO-efficient compared to RS-encoding at append time, requiring only a single large IO on each of the 14 target storage nodes. This optimization, enabled by Tectonic's Client Library-driven design, provides nearly the best of both worlds with fast and IO-efficient replication for small appends that are quickly transitioned to the more space-efficient RS-encoding.

## 6 Tectonic in Production

This section shows Tectonic operating at exabyte scale, demonstrates benefits of storage consolidation, and discusses how Tectonic handles metadata hotspots. It also discusses tradeoffs and lessons from designing Tectonic.

### 6.1 Exabyte-Scale Multitenant Clusters

Production Tectonic clusters run at exabyte scale. Table 2 gives statistics on a representative multitenant cluster. All results in this section are for this cluster. The 1250 PB of storage, ~70% of the cluster capacity at the time of the snapshot, consists of 10.7 billion files and 15 billion blocks.

### 6.2 Efficiency from Storage Consolidation

The cluster in Table 2 hosts two tenants, blob storage and data warehouse. Blob storage uses ~49% of the used space in this cluster and data warehouse uses ~51%. Figures 4a and 4b show the cluster handling storage load over a three-day period. Figure 4a shows the cluster's aggregate IOPS during that time, and Figure 4b shows its aggregate disk bandwidth. The data warehouse workload has large, regular load spikes triggered by very large jobs. Compared to the spiky data warehouse workload, blob storage traffic is smooth and predictable.

**Sharing surplus IOPS capacity.** The cluster handles spikes in storage load from data warehouse using the surplus IOPS capacity unlocked by consolidation with blob storage. Blob storage requests are typically small and bound by IOPS while data warehouse requests are typically large and bound by bandwidth. As a result, neither IOPS nor bandwidth can fairly account for disk IO usage. The bottleneck resource in serving storage operations is *disk time*, which measures how often a given disk is busy. Handling a storage load spike requires Tectonic to have enough free disk time to serve the spike. For example, if a disk does 10 IOs in one second with each taking 50 ms (seek and fetch), then the disk was busy for 500 out of 1000 ms. We use disk time to fairly account for usage by different types of requests.

| | Warehouse | Blob storage | Combined |
|---|---|---|---|
| **Supply** | **0.51** | **0.49** | **1.00** |
| Peak 1 | 0.60 | 0.12 | 0.72 |
| Peak 2 | 0.54 | 0.14 | 0.68 |
| Peak 3 | 0.57 | 0.11 | 0.68 |

**Table 3: Consolidating data warehouse and blob storage in Tectonic allows data warehouse to use what would otherwise be stranded surplus disk time for blob storage to handle large load spikes. This figure shows the normalized disk time demand vs. supply in three daily peaks in the representative cluster.**

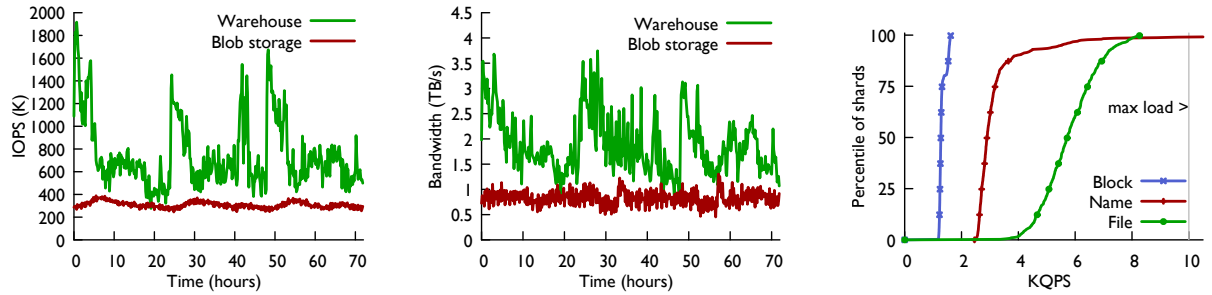
For the representative production cluster, Table 3 shows normalized disk time demand for data warehouse and blob storage for three daily peaks and the supply of disk time each would have if running on its own cluster. We normalize by total disktime corresponding to used space in the cluster. The daily peaks correspond to the same three days of traffic as in Figures 4a and 4b. Data warehouse's demand exceeds its supply in all three peaks and handling it on its own would require disk overprovisioning. To handle peak data warehouse demand over the three day period, the cluster would have needed ~17% overprovisioning. Blob storage, on the other hand, has surplus disk time that would be stranded if it ran in its own cluster. Consolidating these tenants into a single Tectonic cluster allows the blob storage's surplus disk time to be used for data warehouse's storage load spikes.

### 6.3 Metadata Hotspots

Load spikes to the Metadata Store may result in hotspots in metadata shards. The bottleneck resource in serving metadata operations is queries per second (QPS). Handling load spikes requires the Metadata Store to keep up with the QPS demand on every shard. In production, each shard can serve a maximum of 10 KQPS. This limit is imposed by the current isolation mechanism on the resources of the metadata nodes. Figure 4c shows the QPS across metadata shards in the cluster for the Name, File, and Block layers. All shards in the File and Block layers are below this limit.

Over this three-day period, around 1% of Name layer shards hit the QPS limit because they hold very hot directories. The small unhandled fraction of metadata requests are retried after a backoff. The backoff allows the metadata nodes to clear most of the initial spike and successfully serve retried requests. This mechanism, combined with all other shards being below their maximum, enables Tectonic to successfully handle the large spikes in metadata load from data warehouse.

The distribution of load across shards varies between the Name, File, and Block layers. Each higher layer has a larger distribution of QPS per shard because it colocates more of a




**(a) Aggregate cluster IOPS** **(b) Aggregate cluster bandwidth** **(c) Peak metadata load (CDF)**

**Figure 4: IO and metadata load on the representative production cluster over three days. (a) and (b) show the difference in blob storage and data warehouse traffic patterns and show Tectonic successfully handling spikes in storage IOPS and bandwidth over 3 days. Both tenants occupy nearly the same space in this cluster. (c) is a CDF of peak metadata load over three days on this cluster's metadata shards. The maximum load each shard can handle is 10 KQPS (grey line). Tectonic can handle all metadata operations at the File and Block layers. It can immediately handle almost all Name layer operations; the remaining operations are handled on retry.**

tenant's operations. For instance, all directory-to-file lookups for a given directory are handled by one shard. An alternative design that used range partitioning like ADLS [42] would colocate many more of a tenant's operations together and result in much larger load spikes. Data warehouse jobs often read many similarly-named directories, which would lead to extreme load spikes if the directories were range-partitioned. Data warehouse jobs also read many files in a directory, which causes load spikes in the Name layer. Range-partitioning the File layer would colocate files in a directory on the same shard, resulting in a much larger load spike because each job does many more File layer operations than Name layer operations. Tectonic's hash partitioning reduces this colocation, allowing Tectonic to handle metadata load spikes using fewer nodes than would be necessary with range partitioning.

Tectonic also codesigns with data warehouse to reduce metadata hotspots. For example, compute engines commonly use an orchestrator to list files in a directory and distribute the files to workers. The workers open and process the files in parallel. In Tectonic, this pattern sends a large number of nearly-simultaneous file open requests to a single directory shard (§3.3), causing a hotspot. To avoid this anti-pattern, Tectonic's list-files API returns the file IDs along with the file names in a directory. The compute engine orchestrator sends the file IDs and names to its workers, which can open the files directly by file ID without querying the directory shard again.

## 6.4 The simplicity-performance tradeoffs

Tectonic's design generally prioritizes simplicity over efficiency. We discuss two instances where we opted for additional complexity in exchange for performance gains.

**Managing reconstruction load.** RS-encoded data may be stored *contiguously*, where a data block is divided into chunks that are each written contiguously to storage nodes, or *striped*, where a data block is divided into much smaller chunks that are distributed round-robin across storage nodes [51]. Because Tectonic uses contiguous RS encoding and the majority of reads are smaller than a chunk size, reads are usually *direct*: they do not require RS reconstruction and so consist of a single disk IO. Reconstruction reads require 10× more IOs than direct reads (for RS(10,4) encoding). Though common, it is difficult to predict the fraction of reads that will be reconstructed, since reconstruction is triggered by hardware failures as well as node overload. We learned that such a wide variability in resource requirements, if not controlled, can cause cascading failures that affect system availability and performance.

If some storage nodes are overloaded, direct reads fail and trigger reconstructed reads. This increases load to the rest of the system and triggers yet more reconstructed reads, and so forth. The cascade of reconstructions is called a *reconstruction storm*. A simple solution would be to use striped RS encoding where all reads are reconstructed. This avoids reconstruction storms because the number of IOs for reads does not change when there are failures. However, it makes normal-case reads much more expensive. We instead prevent reconstruction storms by restricting reconstructed reads to 10% of all reads. This fraction of reconstructed reads is typically enough to handle disk, host, and rack failures in our production clusters. In exchange for some tuning complexity, we avoid over-provisioning disk resources.

**Efficiently accessing data within and across datacenters.** Tectonic allows clients to directly access storage nodes; an alternative design might use front-end proxies to mediate all client access to storage. Making the Client Library accessible to clients introduces complexity because bugs in the library become bugs in the application binary. However, direct client access to storage nodes is vastly more network- and hardware resource efficient than a proxy design, avoiding an extra network hop for terabytes of data per second.

Unfortunately, direct storage node access is a poor fit for remote requests, where the client is geographically distant from the Tectonic cluster. The additional network overhead makes the orchestration round trips prohibitively inefficient. To solve this problem, Tectonic handles remote data access differently from local data access: remote requests get forwarded to a stateless proxy in the same datacenter as the storage nodes.

## 6.5 Tradeoffs and Compromises

Migrating to Tectonic was not without tradeoffs and compromises. This subsection describes a few areas where Tectonic is either less flexible or less performant than Facebook's previous infrastructure. We also describe the impact of using a hash-partitioned metadata store.

**The impact of higher metadata latency.** Migrating to Tectonic meant data warehouse applications saw higher metadata latency. HDFS metadata operations are in-memory and all metadata for a namespace is stored on a single node. In contrast, Tectonic stores its metadata in a sharded key-value store instance and disaggregates metadata layers (§3.3). This means Tectonic metadata operations may require one or more network calls (e.g., a file open operation will interact with the Name and File layers). Data warehouse had to adjust how it handled certain metadata operations given the additional metadata latency. For instance, compute engines rename a set of files one by one, in sequence, after computation is done. In HDFS each rename was fast, but with Tectonic, compute engines parallelize this step to hide the extra latency of individual Tectonic rename operations.

**Working around hash-partitioned metadata.** Because Tectonic directories are hash sharded, listing directories recursively involves querying many shards. In fact, Tectonic does not provide a recursive list API; tenants need to build it as a client-side wrapper over individual *list* calls. As a result, unlike HDFS, Tectonic does not have *du* (directory utilization) functionality to query aggregate space usage of a directory. Instead, Tectonic periodically aggregates per-directory usage statistics, which can be stale.

## 6.6 Design and Deployment Lessons

**Achieving high scalability is an iterative process enabled by a microservice architecture.** Several Tectonic components have been through multiple iterations to meet increasing scalability requirements. For example, the first version of the Chunk Store grouped blocks to reduce metadata. A number of blocks with the same redundancy scheme were grouped and RS-encoded as one unit to store their chunks together. Each block group mapped to a set of storage nodes. This is a common technique since it significantly reduces metadata [37, 53], but it was too inflexible for our production environment. For example, with only 5% of storage nodes unavailable, 80% of the block groups became unavailable for writes. This design also precluded optimizations like hedged quorum writes and quorum appends (§5).

Additionally, our initial Metadata Store architecture did not separate the Name and File layers; clients consulted the same shards for directory lookups and for listing blocks in a file. This design resulted in unavailability from metadata hotspots, prompting us to further disaggregate metadata.

Tectonic's evolution shows the importance of trying new designs to get closer to performance goals. Our development experience also shows the value of a microservices-based architecture for experimentation: we could iterate on components transparently to the rest of the system.

**Memory corruption is common at scale.** At Tectonic's scale, with thousands of machines reading and writing a large amount of data every day, in-memory data corruption is a regular occurrence, a phenomenon observed in other large-scale systems [12, 27]. We address this by enforcing checksum checks within and between process boundaries.

For data $D$ and checksum $C_D$, if we want to perform an in-memory transformation $F$ such that $D' = F(D)$, we generate checksum $C_{D'}$ for $D'$. To check $D'$, we must convert $D'$ back to $D$ with $G$, the inverse function of $F$, and compare $C_{G(D')}$ with $C_D$. The inverse function, $G$, may be expensive to compute (e.g., for RS encoding or encryption), but it is an acceptable cost for Tectonic to preserve data integrity.

All API boundaries involving moving, copying, or transforming data had to be retrofitted to include checksum information. Clients pass a checksum with data to the Client Library when writing, and Tectonic needs to pass the checksum not just across process boundaries (e.g., between the client library and the storage node) but also within the process (e.g., after transformations). Checking the integrity of transformations prevents corruptions from propagating to reconstructed chunks after storage node failure.

## 6.7 Services that do not use Tectonic

Some services within Facebook do not use Tectonic for storage. Bootstrap services, e.g., the software binary package deployment system, which must have no dependencies, cannot use Tectonic because it depends on many other services (e.g., the key-value store, configuration management system, deployment management system). Graph storage [16] also does not use Tectonic, as Tectonic is not yet optimized for key-value store workloads which often need the low latencies provided by SSD storage.

Many other services do not use Tectonic directly. They instead use Tectonic through a major tenant like blob storage or data warehouse. This is because a core design philosophy of Tectonic is separation of concerns. Internally, Tectonic aims for independent software layers which each focus on a narrow set of a storage system's core responsibilities (e.g., storage nodes only know about chunks but not blocks or files). This philosophy extends to how Tectonic fits in with the rest of the storage infrastructure. For example, Tectonic focuses on providing fault tolerance within a datacenter; it does not pro-

tect against datacenter failures. Geo-replication is a separate problem that Tectonic delegates to its large tenants, who solve it to provide transparent and easy-to-use shared storage for applications. Tenants are also expected to know details of capacity management and storage deployments and rebalancing across different datacenters. For smaller applications, the complexity and implementation needed to interface directly with Tectonic in a way that meets their storage needs would amount to re-implementing features that tenants have already implemented. Individual applications therefore use Tectonic via tenants.

## 7 Related Work

Tectonic adapts techniques from existing systems and the literature, demonstrating how they can be combined into a novel system that realizes exabyte-scale single clusters which support a diversity of workloads on a shared storage fabric.

**Distributed filesystems with a single metadata node.** HDFS [15], GFS [24], and others [38, 40, 44] are limited by the metadata node to tens of petabytes of storage per instance or cluster, compared to Tectonic's exabytes per cluster.

**Federating namespaces for increased capacity.** Federated HDFS [8] and Windows Azure Storage (WAS) [17] combine multiple smaller storage clusters (with a single metadata node) into larger clusters. For instance, a federated HDFS [8] cluster has multiple independent single-namenode namespaces, even though the storage nodes are shared between namespaces. Federated systems still have the operational complexity of bin-packing datasets (§2). Also, migrating or sharing data between instances, e.g., to load-balance or add storage capacity, requires resource-heavy data copying among namespaces [33, 46, 54]

**Hash-based data location for metadata scalability.** Ceph [53] and FDS [36] eliminate centralized metadata, instead locating data by hashing on object ID. Handling failures in such systems is a scalability bottleneck. Failures are more frequent with larger clusters, requiring frequent updates to the hash-to-location map that must propagate to all nodes. Yahoo's Cloud Object Store [41] federates Ceph instances to isolate the effects of failures. Furthermore, adding hardware and draining is complicated, as Ceph lacks support for controlled data migration [52]. Tectonic explicitly maps chunks to storage nodes, allowing controlled migration.

**Disaggregated or sharded metadata for scalability.** Like Tectonic, ADLS [42] and HopsFS [35] increase filesystem capacity by disaggregating metadata into layers in separate sharded data stores. Tectonic hash-partitions directories, while ADLS and HopsFS store some related directory metadata on the same shards, causing metadata for related parts of the directory tree to be colocated. Hash partitioning helps Tectonic avoid hotspots local to part of the directory tree. ADLS uses WAS's federated architecture [17] for block storage. In contrast, Tectonic's block storage is flat.

Like Tectonic, Colossus [28, 32] provides cluster-wide multi-exabyte storage where client libraries directly access storage nodes. Colossus uses Spanner [21], a globally consistent database to store filesystem metadata. Tectonic metadata is built on a sharded key-value store, which only provides within-shard strong consistency and no cross-shard operations. These limitations have not been a problem in practice.

**Blob and object stores.** Compared to distributed filesystems, blob and object stores [14, 18, 36, 37] are easier to scale, as they do not have a hierarchical directory tree or namespace to keep consistent. Hierarchical namespaces are required for most warehouse workloads.

**Other large-scale storage systems.** Lustre [1] and GPFS [45] are tuned for high-throughput parallel access. Lustre limits the number of metadata nodes, limiting scalability. GPFS is POSIX-compliant, introducing unnecessary metadata management overhead for our setting. HBase [9] is a key-value store based on HDFS, but its HDFS clusters are not shared with a warehouse workload. We could not compare with AWS [2] as its design is not public.

**Multitenancy techniques.** Tectonic's multitenancy techniques were co-designed with the filesystem as well as the tenants, and does not aim to achieve optimal fair sharing. It is thus easier to provide performance isolation compared to other systems in the literature. Other systems use more complex resource management techniques to accommodate changes in tenancy and resource use policies, or to provide optimal fair resource sharing among tenants [25, 48, 49].

Some details of Tectonic have previously been described in talks [39, 47] where the system is called Warm Storage.

## 8 Conclusion

This paper presents Tectonic, Facebook's distributed filesystem. A single Tectonic instance can support all Facebook's major storage tenants in a datacenter, enabling better resource utilization and less operational complexity. Tectonic's hash-sharded disaggregated metadata and flat data chunk storage allow it to address and store exabytes. Its cardinality-reduced resource management allows it to efficiently and fairly share resources and distribute surplus resources for high utilization. Tectonic's client-driven tenant-specific optimizations allow it to match or exceed the performance of the previous specialized storage systems.

**Acknowledgements.** We are grateful to our shepherd, Peter Macko, and the anonymous reviewers of the FAST program committee whose extensive comments substantially improved this work. We are also grateful to Nar Ganapathy, Mihir Gorecha, Morteza Ghandehari, Bertan Ari, John Doty, and other colleagues at Facebook who contributed to the project. We also thank Jason Flinn and Qi Huang for suggestions for improving the paper. Theano Stavrinos was supported by the National Science Foundation grant CNS-1910390 while at Princeton University.

## References


[1] Lustre Wiki. https://wiki.lustre.org/images/6/64/LustreArchitecture-v4.pdf, 2017.

[2] AWS Documentation. https://docs.aws.amazon.com/, 2020.

[3] Presto. https://prestodb.io/, 2020.

[4] Aditya Kalro. Facebook's FBLearner Platform with Aditya Kalro. https://twimlai.com/twiml-talk-197-facebooks-fblearner-platform-with-aditya-kalro/, 2018.

[5] J. Adrian. Introducing Bryce Canyon: Our next-generation storage platform. https://tinyurl.com/yccx2x7v, 2017.

[6] M. Annamalai. ZippyDB - A Distributed key value store. https://www.youtube.com/embed/ZRP7z0HnClc, 2015.

[7] Apache Software Foundation. HDFS Erasure Coding. https://hadoop.apache.org/docs/r3.1.1/hadoop-project-dist/hadoop-hdfs/HDFSErasureCoding.html, 2018.

[8] Apache Software Foundation. HDFS Federation. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/Federation.html, 2019.

[9] Apache Software Foundation. Apache HBase. https://hbase.apache.org/, 2020.

[10] Apache Software Foundation. Apache Spark. https://spark.apache.org/, 2020.

[11] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a Needle in Haystack: Facebook's Photo Storage. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*, Vancouver, BC, Canada, 2010. USENIX Association.

[12] D. Behrens, M. Serafini, F. P. Junqueira, S. Arnautov, and C. Fetzer. Scalable error isolation for distributed systems. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*, Oakland, CA, USA, 2015. USENIX Association.

[13] B. Berg, D. S. Berger, S. McAllister, I. Grosof, J. Gunasekar, Sathya Lu, M. Uhlar, J. Carrig, N. Beckmann, M. Harchol-Balter, and G. R. Ganger. The CacheLib Caching Engine: Design and Experiences at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, Online, 2020. USENIX Association.

[14] A. Bigian. Blobstore: Twitter's in-house photo storage system. https://blog.twitter.com/engineering/en_us/a/2012/blobstore-twitter-s-in-house-photo-storage-system.html, 2012.

[15] D. Borthakur. HDFS Architecture Guide. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html, 2019.

[16] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the 2013 USENIX Annual Technical Conference*. USENIX, 2013.

[17] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*, Cascais, Portugal, 2011. Association for Computing Machinery (ACM).

[18] J. Chen, C. Douglas, M. Mutsuzaki, P. Quaid, R. Ramakrishnan, S. Rao, and R. Sears. Walnut: a unified cloud object store. 2012.

[19] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys (CSUR)*, 26(2):145–185, 1994.

[20] A. Cidon, S. Rumble, R. Stutsman, S. Katti, J. Ousterhout, and M. Rosenblum. Copysets: Reducing the Frequency of Data Loss in Cloud Storage. In *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC'13)*, San Jose, CA, USA, 2013. USENIX Association.

[21] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3), Aug. 2013. ISSN 0734-2071. doi: 10.1145/2491245. URL https://doi.org/10.1145/2491245.

[22] J. Dean and L. A. Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, Feb. 2013. ISSN 0001-

0782. doi: 10.1145/2408776.2408794. URL http://doi.acm.org/10.1145/2408776.2408794.

[23] Facebook Open Source. RocksDB. https://rocksdb.org/, 2020.

[24] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, Bolton Landing, NY, USA, 2003. Association for Computing Machinery (ACM).

[25] R. Gracia-Tinedo, J. Sampé, E. Zamora, M. Sánchez-Artigas, P. García-López, Y. Moatti, and E. Rom. Crystal: Software-defined storage for multi-tenant object stores. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*, Santa Clara, CA, USA, 2017. USENIX Association.

[26] X. F. Group. The XFS Linux wiki. https://xfs.wiki.kernel.org/, 2018.

[27] A. Gupta, F. Yang, J. Govig, A. Kirsch, K. Chan, K. Lai, S. Wu, S. Dhoot, A. Kumar, A. Agiwal, S. Bhansali, M. Hong, J. Cameron, M. Siddiqi, D. Jones, J. Shute, A. Gubarev, S. Venkataraman, and D. Agrawal. Mesa: Geo-replicated, near real-time, scalable data warehousing. In *Proceedings of the 40th International Conference on Very Large Data Bases (VLDB'14)*, Hangzhou, China, 2014. VLDB Endowment.

[28] D. Hildebrand and D. Serenyi. A peek behind the VM at the Google Storage infrastructure. https://www.youtube.com/watch?v=q4WC_6SzBz4, 2020.

[29] Q. Huang, P. Ang, P. Knowles, T. Nykiel, I. Tverdokhlib, A. Yajurvedi, P. Dapolito IV, X. Yan, M. Bykov, C. Liang, M. Talwar, A. Mathur, S. Kulkarni, M. Burke, and W. Lloyd. SVE: Distributed video processing at Facebook scale. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP'17)*, Shanghai, China, 2017. Association for Computing Machinery (ACM).

[30] L. Leslie. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[31] K. Lewi, C. Rain, S. A. Weis, Y. Lee, H. Xiong, and B. Yang. Scaling backend authentication at facebook. *IACR Cryptol. ePrint Arch.*, 2018:413, 2018. URL https://eprint.iacr.org/2018/413.

[32] M. K. McKusick and S. Quinlan. GFS: Evolution on Fast-forward. *Queue*, 7(7):10:10–10:20, Aug. 2009. ISSN 1542-7730. doi: 10.1145/1594204.1594206. URL http://doi.acm.org/10.1145/1594204.1594206.

[33] P. A. Misra, I. n. Goiri, J. Kace, and R. Bianchini. Scaling Distributed File Systems in Resource-Harvesting Datacenters. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC'17)*, Santa Clara, CA, USA, 2017. USENIX Association.

[34] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, and S. Kumar. f4: Facebook's Warm BLOB Storage System. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, Broomfield, CO, USA, 2014. USENIX Association.

[35] S. Niazi, M. Ismail, S. Haridi, J. Dowling, S. Grohsschmiedt, and M. Ronström. HopsFS: Scaling hierarchical file system metadata using NewSQL databases. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*, Santa Clara, CA, USA, 2017. USENIX Association.

[36] E. B. Nightingale, J. Elson, J. Fan, O. Hofmann, J. Howell, and Y. Suzue. Flat Datacenter Storage. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*, Hollywood, CA, USA, 2012. USENIX Association.

[37] S. A. Noghabi, S. Subramanian, P. Narayanan, S. Narayanan, G. Holla, M. Zadeh, T. Li, I. Gupta, and R. H. Campbell. Ambry: Linkedin's scalable geo-distributed object store. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD'16)*, San Francisco, California, USA, 2016. Association for Computing Machinery (ACM).

[38] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly. The Quantcast File System. In *Proceedings of the 39th International Conference on Very Large Data Bases (VLDB'13)*, Riva del Garda, Italy, 2013. VLDB Endowment.

[39] K. Patiejunas and A. Jaiswal. Facebook's disaggregated storage and compute for Map/Reduce. https://atscaleconference.com/videos/facebooks-disaggregated-storage-and-compute-for-mapreduce/, 2016.

[40] A. J. Peters and L. Janyst. Exabyte scale storage at CERN. *Journal of Physics: Conference Series*, 331 (5):052015, dec 2011. doi: 10.1088/1742-6596/331/5/052015. URL https://doi.org/10.1088/1742-6596/331/5/052015.

[41] N. P.P.S, S. Samal, and S. Nanniyur. Yahoo Cloud Object Store - Object Storage at Exabyte Scale. https://yahooeng.tumblr.com/post/116391291701/yahoo-cloud-object-store-object-storage-at, 2015.

[42] R. Ramakrishnan, B. Sridharan, J. R. Douceur, P. Kasturi, B. Krishnamachari-Sampath, K. Krishnamoorthy, P. Li, M. Manu, S. Michaylov, R. Ramos, N. Sharman, Z. Xu, Y. Barakat, C. Douglas, R. Draves, S. S. Naidu, S. Shastry, A. Sikaria, S. Sun, and R. Venkatesan. Azure Data Lake Store: a hyperscale distributed file service for big data analytics. In *Proceedings of the 2017 International Conference on Management of Data (SIGMOD'17)*, Chicago, IL, USA, 2017. Association for Computing Machinery (ACM).

[43] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[44] Rousseau, Hervé, Chan Kwok Cheong, Belinda, Contescu, Cristian, Espinal Curull, Xavier, Iven, Jan, Gonzalez Labrador, Hugo, Lamanna, Massimo, Lo Presti, Giuseppe, Mascetti, Luca, Moscicki, Jakub, and van der Ster, Dan. Providing large-scale disk storage at cern. *EPJ Web Conf.*, 214:04033, 2019. doi: 10.1051/epjconf/201921404033. URL https://doi.org/10.1051/epjconf/201921404033.

[45] F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST'02)*, Monterey, CA, USA, 2002. USENIX Association.

[46] R. Shah. Enabling HDFS Federation Having 1B File System Objects. https://tech.ebayinc.com/engineering/enabling-hdfs-federation-having-1b-file-system-objects/, 2020.

[47] S. Shamasunder. Hybrid XFS—Using SSDs to Supercharge HDDs at Facebook. https://www.usenix.org/conference/srecon19asia/presentation/shamasunder, 2019.

[48] D. Shue, M. J. Freedman, and A. Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*, Hollywood, CA, USA, 2012. USENIX Association.

[49] A. K. Singh, X. Cui, B. Cassell, B. Wong, and K. Daudjee. Microfuge: A middleware approach to providing performance isolation in cloud storage systems. In *Proceedings of the 34th IEEE International Conference on Distributed Computing Systems (ICDCS'14)*, Madrid, Spain, 2014. IEEE Computer Society.

[50] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sarma, R. Murthy, and H. Liu. Data warehousing and analytics infrastructure at facebook. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD'10)*, Indianapolis, IN, USA, 2010. Association for Computing Machinery (ACM).

[51] A. Wang. Introduction to HDFS Erasure Coding in Apache Hadoop. https://blog.cloudera.com/introduction-to-hdfs-erasure-coding-in-apache-hadoop/, 2015.

[52] L. Wang, Y. Zhang, J. Xu, and G. Xue. MAPX: Controlled Data Migration in the Expansion of Decentralized Object-Based Storage Systems. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20)*, Santa Clara, CA, USA, 2020. USENIX Association.

[53] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*, Seattle, WA, USA, 2006. USENIX Association.

[54] A. Zhang and W. Yan. Scaling Uber's Apache Hadoop Distributed File System for Growth. https://eng.uber.com/scaling-hdfs/, 2018.