VAST

White paper

VAST AI Operating System

Table of Contents

Introduction

AI has emerged as the most transformative technology of our time. It is redefining how knowledge is produced, how decisions are made, and how entire scientific and industrial domains push the boundaries of what is possible. AI is accelerating discovery, amplifying human capability, and turning formerly intractable problems into solvable ones across every discipline. It is reshaping how we teach and learn, engineer complex systems, build and ship software, personalize services at scale, and diagnose and document care. Yet the magnitude of this transformation is not merely in what AI can do today, but in the rate at which its capabilities are improving. Every breakthrough becomes a stepping stone to the next, and the pace is outrunning all assumptions of earlier eras of technology.

Bringing these breakthroughs from the lab into the world demands more than clever algorithms; it requires an equally unprecedented rethinking of the physical, computational, and organizational infrastructure needed to make intelligence possible at scale. The world is already feeling the strain: hyperscale datacenter construction is outpacing every other category of industrial development, regional power grids are being re-architected to supply multi-gigawatt AI campuses, and policymakers are confronting the uncomfortable fact that AI demand is beginning to shape national energy strategy. These pressures dominate headlines because they reveal a structural constraint: AI has outgrown the scale of any single building, any single campus, and even the capacity of regional transmission infrastructure. The bottleneck in AI's progress is no longer the ingenuity of model designers, but our ability to build, connect, and govern the massive, distributed machinery required to run these systems reliably, economically, and responsibly.

As we narrow the scope from national infrastructure down to the operational realities within and across AI facilities, the complexity deepens. Modern AI workloads require storage, networking, and compute to operate not as independent subsystems, but as a tightly coupled, data-centric fabric. Clusters must move and process petabytes of data with global consistency; they must elastically federate across datacenters, regions, or even continents; and they must do so with automation, auditability, and fine-grained data governance that satisfies regulatory, operational, and societal demands.

At this scale, the true constraint is not raw hardware capacity, but the absence of an integrated operating system that coordinates these distributed resources as a coherent whole. This need for a unified, data-aware, federated control plane is precisely what led us to build the VAST AI Operating System.

What is the VAST AI Operating System?

The VAST AI Operating System is a software platform that simplifies and accelerates the entire AI pipeline—from raw data ingest, through context analysis, to intelligent action—as a continuous, real-time flow. It is designed to be the foundation upon which AI-driven applications and autonomous agents are built, deployed, and managed at scale. Think of it as an operating system for the datacenter itself: one that manages the underlying resources across hundreds to thousands of systems, provides essential shared services, and offers a consistent environment for AI applications to run—much as a traditional OS does for a single machine, but built for the massive distributed scale and unique demands of modern AI.

The platform's origins trace back to a fundamental rethinking of how distributed systems should be built. In 2016, VAST introduced the Disaggregated and Shared Everything (DASE) architecture, which separates compute logic from physical storage media while allowing all compute nodes to access all storage devices in parallel via high-performance NVMe-over-Fabrics. This "shared everything" approach, combined with novel transactional data structures, eliminates the data partitioning and complex inter-server coordination that plague traditional "shared-nothing" designs. The result is a system that scales linearly from terabytes to exabytes, breaks the traditional tradeoffs between performance and capacity, and simplifies data management across all-flash infrastructure. DASE provided the foundation for VAST's Universal Storage platform in 2019, its evolution into an AI Data Platform by 2023, and ultimately the VAST AI Operating System in 2025. As a software-defined system, it runs on your choice of vendor server or cloud platform, ensuring total flexibility with no hardware lock-in.

Built upon this DASE foundation, the AI Operating System is organized into two broad tiers: foundational data platform services and specialized execution engines. The data platform services—**DataStore, DataBase, and DataSpace**—address the storage, organization, and global accessibility of data.

- **DataStore** provides the high-performance, resilient file, object, and block storage where data lives, building on VAST's Universal Storage heritage to ingest, store, and deliver rapid access to the massive datasets AI demands.
- **DataBase** extends this with a purpose-built database for AI, capable of handling structured data such as enterprise data warehouse tables alongside contextual metadata, vectors for similarity search, real-time streams, catalogs, and logs—making all of this data queryable and usable for model training and inference.
- **DataSpace** provides the overarching globally distributed data computing layer, enabling organizations to manage and access data across on-premises, cloud, and edge environments through a unified namespace and access framework.

Together, these services ensure that data of any type, at any location, are continuously available to the AI pipeline without the silos and fragmentation that traditionally impede large-scale workloads.

The execution engines transform this data foundation into intelligent, autonomous action.

- **DataEngine** serves as the orchestration supervisor for the AI OS, transforming distributed computing resources into an integrated execution environment by managing serverless function pipelines driven by event triggers and Python-based microservices. By decoupling applications from data gravity—intelligently moving compute to where the data resides—it optimizes for cost, execution time, and hardware requirements including GPUs, and includes a native Kafka-compatible Event Broker that stores message topics as VAST Tables for global ordering and real-time SQL querying of live streams.
- **SyncEngine** operates within the DataEngine as a highly scalable data discovery and migration service, indexing and synchronizing data from external sources—S3 buckets, file systems, SaaS platforms—into the VAST DataSpace, maintaining a durable searchable index and triggering enrichment pipelines automatically as new data is onboarded.
- **InsightEngine** focuses on enriching data, providing services that compute, index, maintain, and serve vector embeddings for data as it evolves to enable read-time semantic search across all data for RAG-enabled applications and agents.
- **AgentEngine** provides the production-grade deployment and orchestration layer for AI agents, bridging the gap between notebook prototyping and enterprise-scale execution with a robust runtime for long-running containerized services, lifecycle management, persistent state, and secure service discovery—all fully auditable and traceable for compliance across complex multi-agent deployments.

The power of this architecture lies in how these layers compose into a seamless, end-to-end workflow. The SyncEngine discovers and onboards data from across the enterprise into the DataStore, preserving identity and continuity. The DataEngine processes and streams that data through event-driven serverless functions and the Event Broker, ensuring it is AI-ready. The InsightEngine leverages content-specific chunking and embedding models to contextualize the processed data, generating vector embeddings within the DataBase that make information immediately queryable and readable for large language models. And the AgentEngine provides the stable, secure environment at the top of the stack where AI agents execute complex, multi-step tasks by leveraging tools and data across the entire ecosystem.

By consolidating infrastructure that traditionally required separate silos—object stores, analytics clusters, Kubernetes orchestration, and message brokers—into a single pool of resources governed by a unified control plane, the VAST AI Operating System eliminates the performance bottlenecks and operational fragmentation that have historically constrained AI at scale. It remains entirely model-agnostic, runs on any preferred hardware or cloud, and imposes no vendor lock-in—acting as the orchestrator that bridges infrastructure and intelligence with the unified data layer and management framework necessary for seamless performance at any scale.

What is Possible with the VAST AI OS?

With the foundational challenges of scale, coordination, and governance now addressed, organizations can shift their focus from managing infrastructure to advancing AI itself.

For model trainers, this means training runs are no longer constrained by single-datacenter capacity. Multi-site training becomes operationally viable, because runs can access common data from a globally consistent namespace, maintain predictable performance through failures, and checkpoint at scale without destabilizing the entire data platform.

For inference operators, this means models deploy closer to data sources across on-premises, cloud, and edge environments, while requests are intelligently routed and governance policies are applied uniformly and automatically. Vector embeddings remain current as data evolves, allowing inference to be consistent regardless of scale or geography.

For agentic systems, this means agents discover, access, and reason over diverse data types—structured tables, vectors, streams, logs—through a single, scalable, and observable interface. Long-running workflows maintain persistent state across failures, compose multiple models and tools, and generate complete audit trails, all within a globally unified namespace.

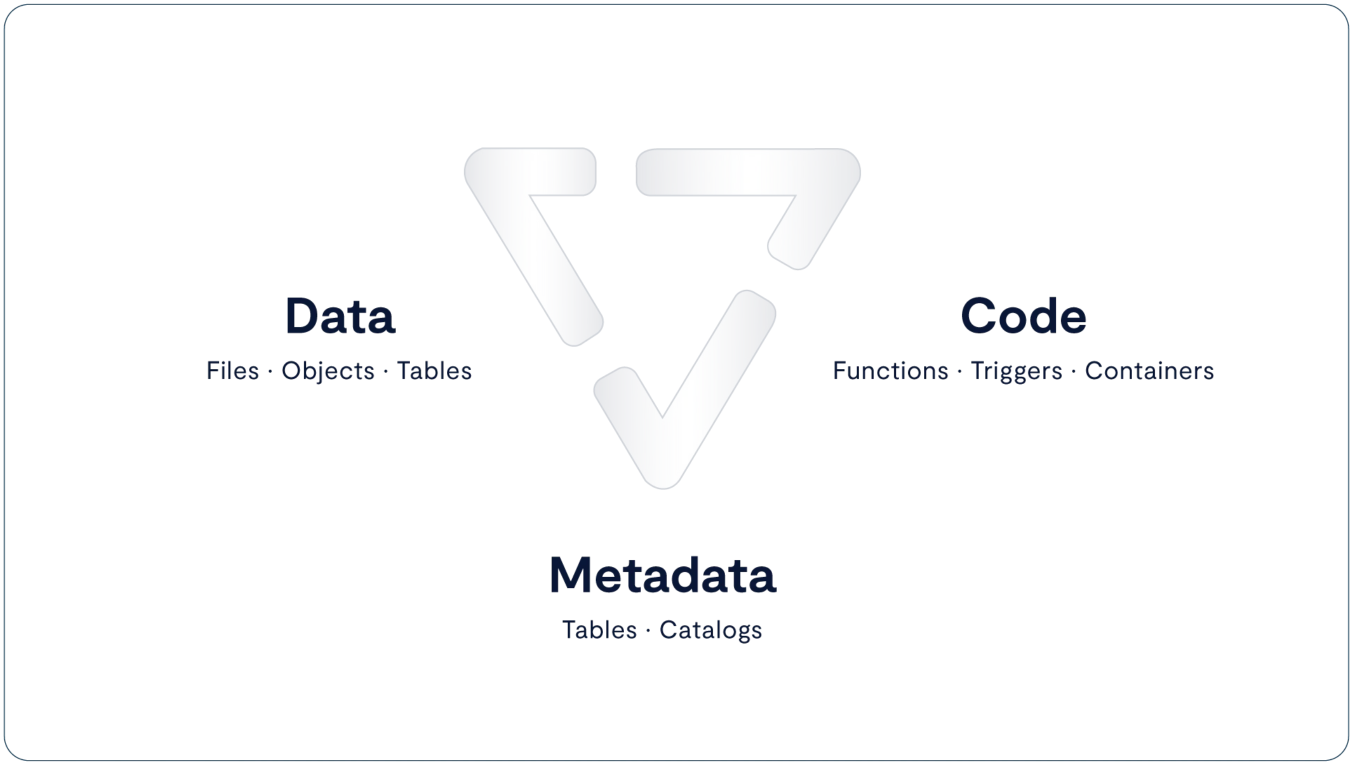
By unifying storage, networking, compute, orchestration, observability, and governance in a single, federated control plane, the VAST AI OS allows every aspect of operational AI to scale within and across datacenters without fighting the infrastructure beneath them. The system abstracts complexity, scales linearly, and enforces governance so that engineering efforts shift from integrating disparate systems to solving the problems AI was meant to address. With the VAST AI OS, you can deploy new models, onboard new data sources, and scale workloads at the speed of your ambitions rather than the pace of infrastructure expansions.

The following sections detail how the VAST AI OS makes this possible: the architecture, coordination mechanisms, and operational semantics designed specifically to transform distributed resources into a coherent operating system for AI.

The VAST Data Platform

How It Works

The VAST Data Platform is a unified and containerized software environment that brings different aspects of functionality to different application consumers.



Data

Files · Objects · Tables

Code

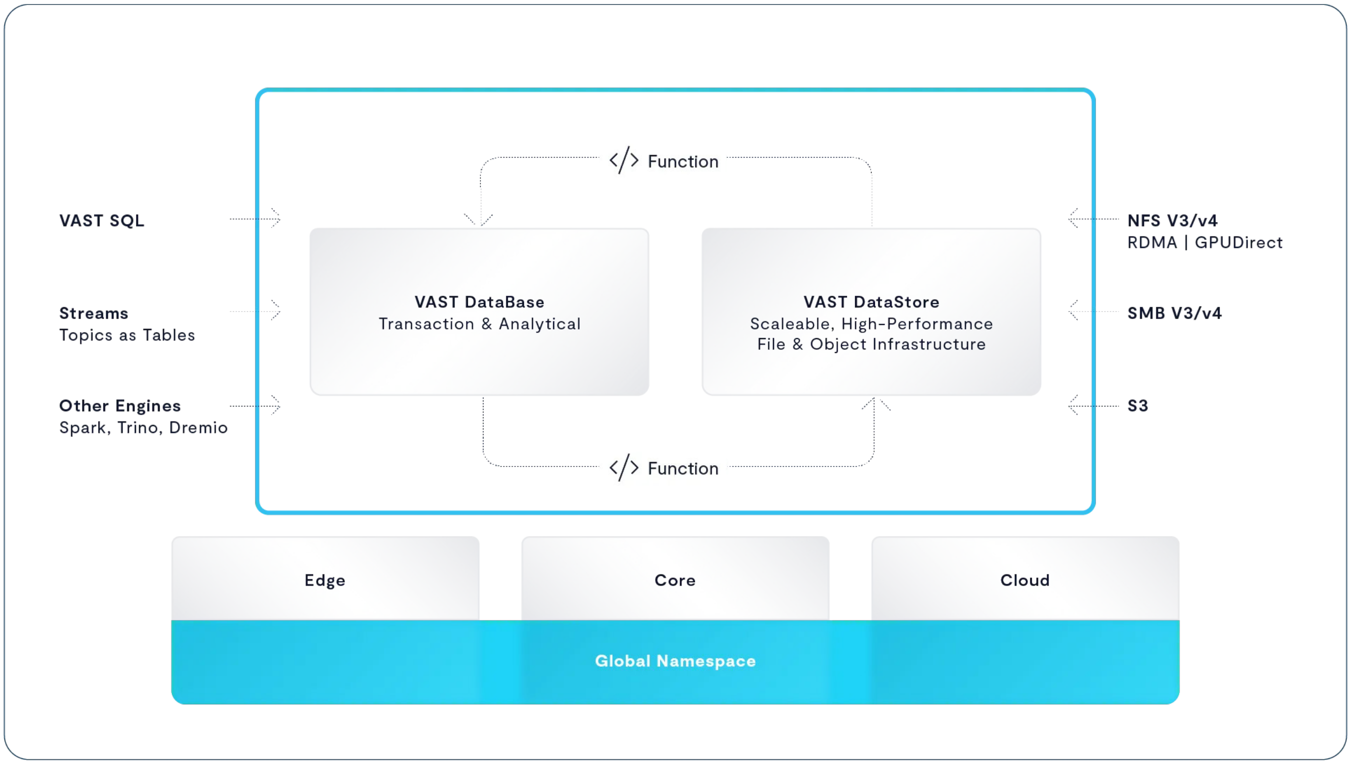
Functions · Triggers · Containers

Metadata

Tables · Catalogs

The system is the first data platform to bring together native support for tabular data (either written natively or converted from open formats such as Parquet), data streams and notifications that work with Kafka endpoints and unstructured data from high-performance enterprise file protocols such as NFS, SMB, and S3. By adding a serverless computing engine (supporting functions written in Python), the VAST Data Platform brings life to data by creating an environment for recursive AI computing. Data events become the functional application triggers that create opportunities for the system to catalog data, an event that may then trigger additional functions such as AI inference, metadata enrichment, and subsequent AI model retraining. By marrying data with code, the system can recursively compute on both new and long-term data, thus getting ever more intelligent by combining new interactions with past realizations in real-time.

Unlike batch-based computing architectures, the VAST architecture leverages a real-time write buffer to capture and manipulate data in real-time as it flows into the system. This buffer can intercept small and random write operations (such as an event stream or a database entry) as well as massively parallel write operations (such as application checkpoint file creations) all into a persistent memory space that is immediately available for retrieval and correlative analysis against the rest of the system's corpus that is largely stored in low-cost hyperscale-grade flash-based archive storage. With a focus on deep learning, the platform works to derive and catalog structure from unstructured data, to serve as the foundation of automation and discovery harnessing data that is captured from the natural world.



The functional areas of the system break down into a few components which all combine into on Platform:

- The **VAST DataStore** is the storage foundation of the Platform. Previously known as VAST's Universal Storage offering, the DataStore is responsible for persisting data and making it available via all of the different protocols that applications may write or read from. The DataStore can be scaled to exabytes within a single data center, and is renowned for breaking the fundamental tradeoff between performance and capacity so that customers can manage their files, objects and tables on a single tier of affordable flash, thus making it ready for any-scale and any-depth data computing.

- The DataStore's ultimate purpose is to capture and serve the immense amount of natural raw, unstructured and stream data that is being captured from the natural world. The **VAST DataBase** is the system's database management service that writes tables into the system and enables real-time, fine-grained queries into VAST reserves of tabular data and cataloged metadata. Unlike conventional approaches to database management systems, the VAST DataBase is transactional like a row-based OLTP database, from a columnar data structure that processes analytical queries like a flash-based data warehouse, and has the scale and affordability of a data lake.

- The DataBase's ultimate purpose is to organize the corpus of knowledge that lives in the VAST DataStore and to catalog the semantic understanding of unstructured data. The **VAST DataEngine** (shipping in 2024) is the system's declarative function execution environment that enables serverless functions much like AWS Lambda and event notifications to be deployed in standard Linux containers. With a built-in scheduler and cost-optimizer, the DataEngine can be deployed on CPU, GPU and DPU architectures to leverage scalable commodity computing to bring life to data. Unlike classic approaches to computing, the DataEngine bridges the divide between event-based and data-driven architectures that enables you to stream into your systems of insight and derive real-time insight by analyzing and inferring and training against all of your data.

- The DataEngine's ultimate purpose is to transform raw, unstructured data into information by inferring on and analyzing data's underlying characteristics.

- The **VAST DataSpace** then takes these concepts and extends them across data centers, building a unified computing fabric and storage namespace that is intended to break the classic tradeoffs of geo-distributed computing. The DataSpace provides global data access by synchronizing metadata and presenting data through remote data caches, but allows for each site to assume temporary responsibility for consistency management at extreme levels of namespace granularity (file-level, object-level, table-level). With this decentralized and fine-grained approach to consistency management, the VAST Data Platform is a global data store that ensures strict application consistency while also providing remote functions with high-performance. The DataSpace makes access easy, by eliminating islands of information while also keeping pipes full and keeping remote CPUs and GPUs busy by prefetching and pipelining data in an intelligent fashion.

  The DataEngine then layers on top of this namespace to create a flexible computational fabric that can route functions to the data (when data gravity is greater) or routes data to the function (when processing resources are scarce near the data). In this way, the DataSpace helps organizations fight against both processor and data gravity as they build global AI computing environments.

  The DataSpace's ultimate purpose is to be the Platform's interface to the natural world, extending access on a global basis and enabling federated AI training and AI inference.

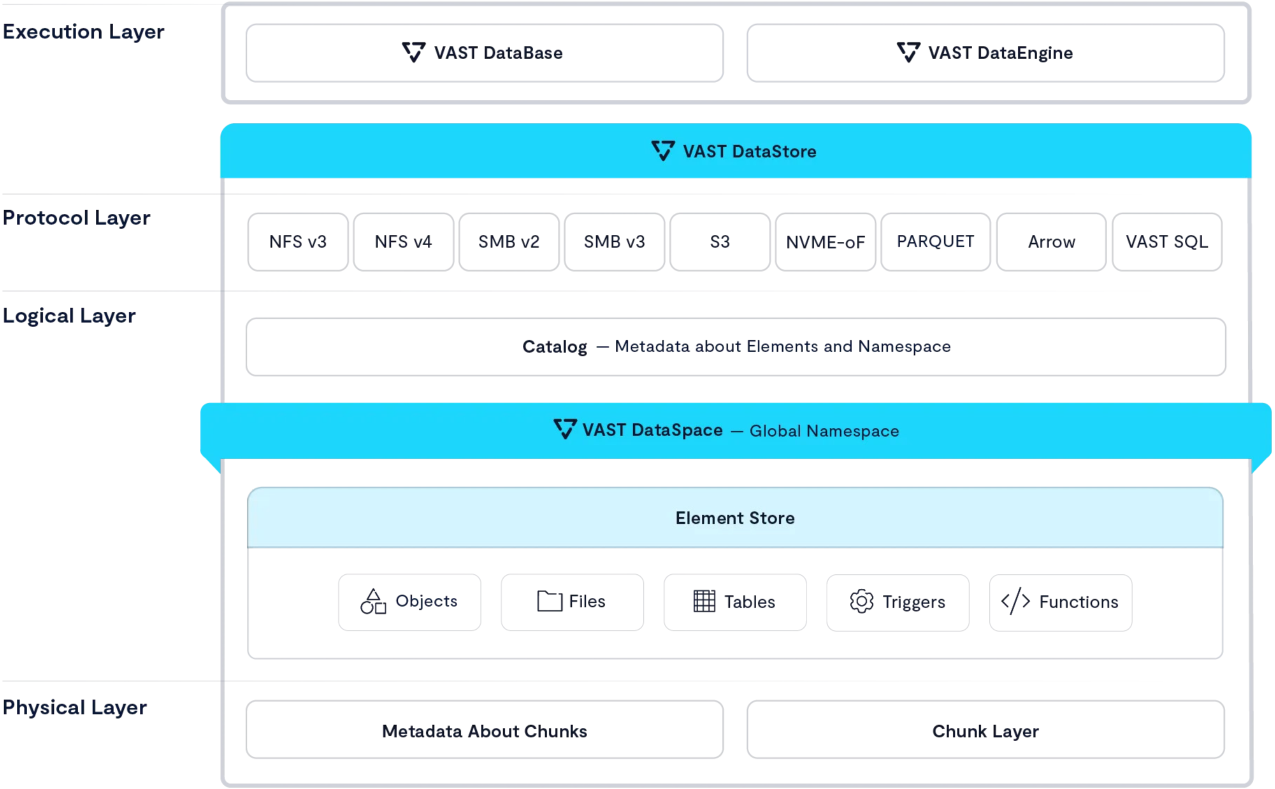
Now that you have a basic idea what The VAST Data Platform is let's see how well it fits what we identified earlier as requirements for big data and deep learning workloads:

Category	Big Data	Deep Learning	VAST Data Platform
Data Types	Structured & Semi-Structured Tables, JSON, Parquet	Unstructured Text, Video, Instruments, etc.	Structured and Unstructured
Processor Type	CPUs	GPUs, AI Processors & DPUs	Orchestrates across, manages CPU, GPU, DPU, etc.
Storage Protocols	S3	S3, RDMA file for GPUs	S3, NFSoRDMA, SMB
Dataset Size	TB-scale warehouses	TB–EB scale volumes	100 TB–EBs
Namespace	Single-Site	Globally-Federated	Globally-Federated
Processing Paradigm	Data-Driven (Batch)	Continuous (Real-Time)	Real-time and batch

Architecting the VAST Data Platform

The VAST Data Platform is architected as a collection of service processes that communicate with outside clients and each other to provide a wide range of data services. The easiest way to explain these services and how they interact is to view them as providing layered services analogous to the seven layers of the OSI networking model.

However, unlike the OSI model, which defines a strictly layered architecture with clear boundaries between protocols at different layers, the VAST Data Platform layers are a tool of explanation; they do not represent hard boundaries. Some services may provide services across what would logically be a layer boundary, and services communicate across layers using non-public interfaces.



Starting from the bottom, because every architecture has to have a solid foundation, the layers of the VAST Platform are:

- **The VAST DataStore** – The VAST DataStore is responsible for storing and protecting data across the VAST global namespace and making that data available both via traditional storage protocols and through internal protocols to The VAST DataBase and The VAST DataEngine. The VAST DataStore has three significant sub-layers:
  - The Physical or Chunk Management Layer provides basic data preservation services for the small data chunks the VAST Element Store uses as its atomic units. This layer includes services such as erasure-coding, data distribution, data reduction, encryption at rest, and device management.
  - The Logical Layer aka **The VAST Element Store** - Uses metadata to assemble the physical layer's data chunks into the Data Elements like files, objects, tables and volumes that users and applications interact with and then organizes those Elements into a single global namespace across a VAST cluster and with **The VAST DataSpace** across multiple VAST clusters worldwide.
  - Like a file system after exposure to gamma rays The VAST Element Store organizes the physical storage layer's chunks into a global namespace holding Elements such as file/objects, tables, and block volumes/LUNs.
  - The VAST Element Store provides services at the Element or path level including access control, encryption, snapshots, clones, and replication.
  - The Protocol Layer provides multiprotocol access to data Elements. All the protocol modules are peers providing full multiprotocol access to Elements as appropriate to their data type.
- The Execution Layer - The execution layer provides and orchestrates the computing logic to turn data into insight through data-driven processing. The execution layer includes two major services:

- **The VAST DataBase** – This service manages structured data. The VAST DataBase is designed to provide the transactional consistency required for OLTP (Online Transaction Processing), the data organization, and the complex query processing demanded by OLAP (Online Analytical Processing) at the scale required for today's AI applications.

- Where the logical layer stores tables and the protocol layer provides basic SQL access to those tables, The VAST DataBase turns those tables into a full-featured database management system providing advanced database functions from sort keys to foreign keys and joins.

- **The VAST DataEngine** – The VAST DataEngine adds the intelligence required to process, transform, and ultimately infer insight from raw data. The VAST DataEngine performs functions such as facial recognition, data loss prevention scanning, or transcoding on Elements based on event triggers like the arrival of an object meeting some filter or a Lambda function.

- The VAST DataEngine acts as a global task scheduler assigning functions to execute where the combination of compute resources, data accessibility, and cost best meet user requirements across a global network of public and private resources.

There's nothing revolutionary about layered architecture; IT organizations have been running relational databases on top of SAN storage with orchestration engines like Kubernetes for years. The revolutionary parts of The VAST Data Platform are the tight integration between the services across the typical layer boundaries and the Disaggregated Shared Everything (DASE) cluster architecture those services run on.

The Disaggregated Shared Everything Architecture

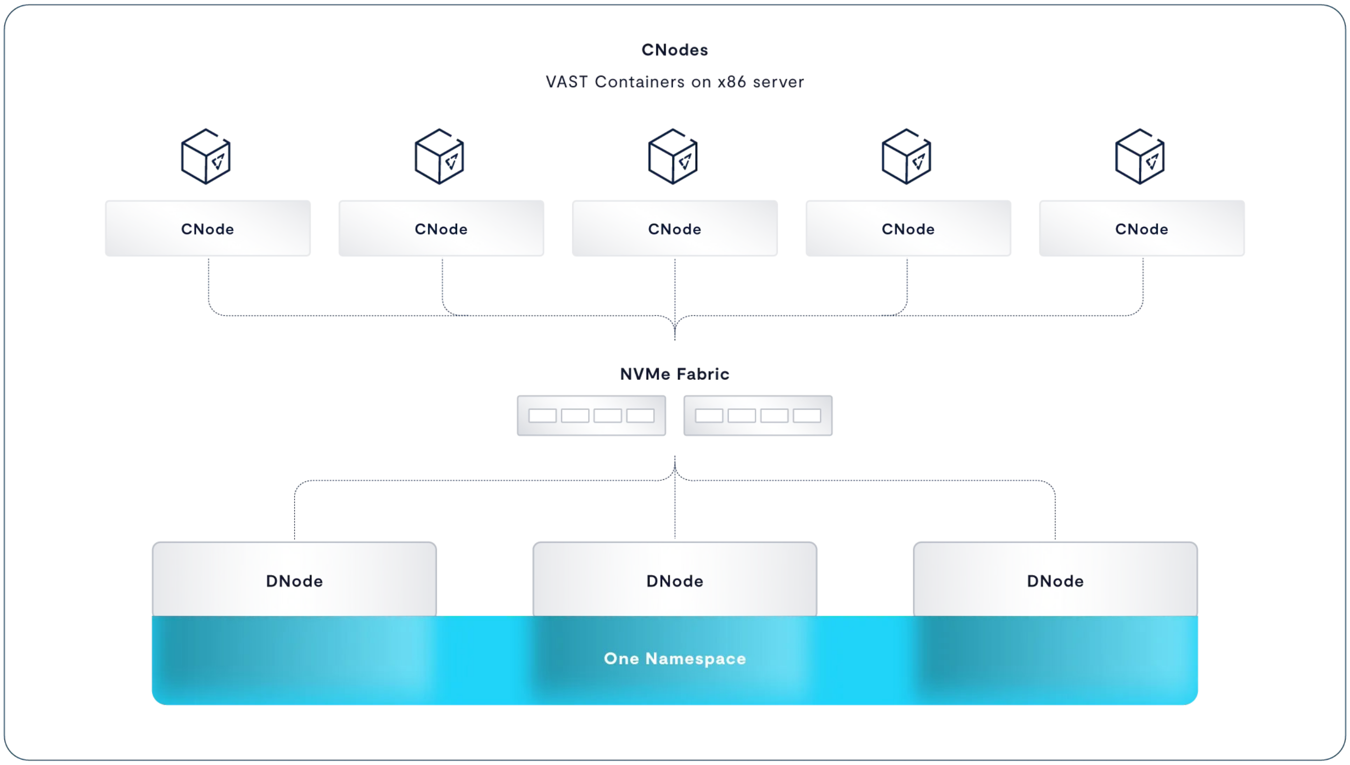
The VAST AIOS is software-defined, meaning it does all its magic in software running in containers on standard x86 and ARM CPUs. But that doesn't mean we wrote that software to run on a cluster of least-common-denominator x86 servers. By using the latest storage and networking technologies like Storage Class Memory SSDs and NVMe over Fabrics (NVMe-oF), this Disaggregated Shared Everything (DASE) architecture allows VAST clusters to scale to unprecedented size. It breaks the tradeoffs inherent in the shared-nothing and shared-media models used in the past.

The DASE architecture introduces two revolutionary new concepts to data system cluster design.

- **Disaggregating the cluster's computational resources from its persistent data and system state.** In a DASE cluster, all the computation is performed by compute nodes (VAST Servers, sometimes called CNodes) that run in stateless containers. This very much includes all the computation needed to maintain the persistent storage that is traditionally performed by storage controller CPUs. This enables the cluster compute resources to be scaled independently from storage capacity across a commodity data center network.

- **A Shared-Everything model** that provides every CNode with direct access to all the data, metadata, and other state information in the cluster. In a DASE cluster, all that system state is stored on NVMe SSDs, not just in the node's DRAM. Since those SSDs are connected to the cluster's NVMe fabric, all the overhead created by device ownership in other system architectures is eliminated.

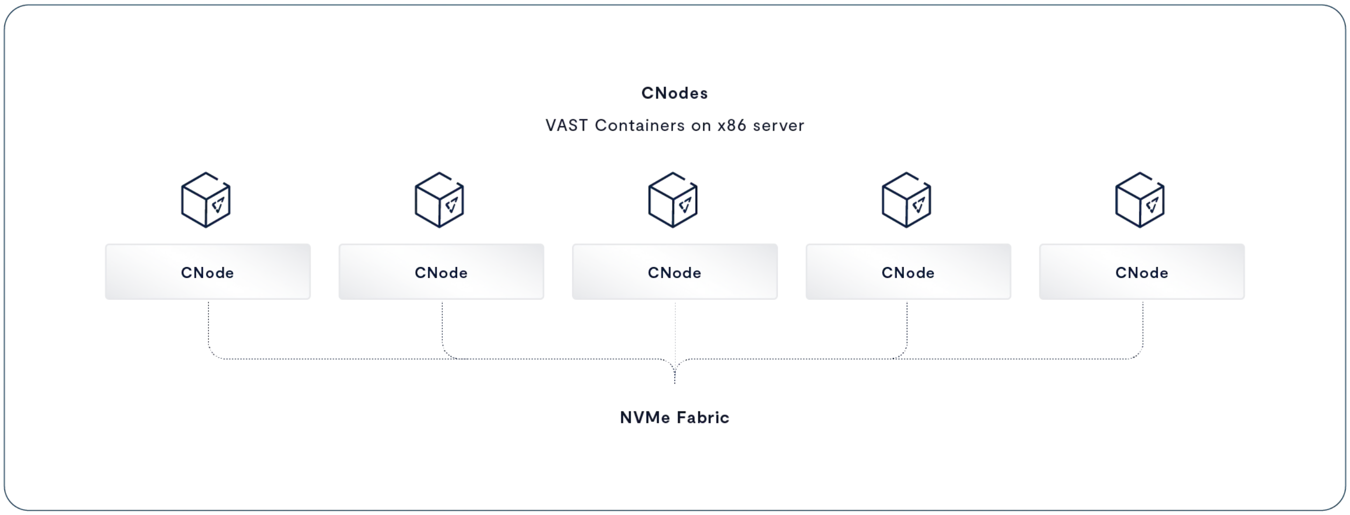
Every CNode in a cluster mounts every SSD in the DASE cluster at boot time and has direct access to the shared system state, which provides a single, unpartitioned source of truth for everything, from global data reduction to database transaction state.



From the key points and diagram above, you've probably figured out that the DASE architecture is based on two basic components: CNodes, which run all the software to perform all the logic, and DBoxes, which hold all the storage media and the system state.

VAST Servers (CNodes)

VAST Servers, or CNodes (for compute nodes), provide the intelligence to manage a VAST cluster. This includes responding to client requests, protecting data in the VAST Element Store, processing database queries, and determining the best place to transcode a sports highlight to get it into the next highlights show, and processing that stream.



The VAST AIOS runs as a group of stateless containers across a cluster of servers. The term CNode (for compute node) usually refers to the VAST server container running as a node in a VAST cluster. However, it can occasionally be used to refer to the server on which the CNode container runs. If you ever see a reference like "Each CNode can use the same 100 Gbps Ethernet card for both cluster internal traffic and connections to the client network," it's safe to assume CNode in that context means the server hardware the VAST CNode container runs on.

All the CNodes in a cluster mount all the SCM and hyperscale flash SSDs in the cluster via NVMe-oF at boot time. This means that every CNode can directly access all the data and metadata across the cluster. In the DASE architecture, everything — every storage device, every metadata structure, and the state of every transaction within the system — is shared across all the CNode servers in the cluster.

In a DASE cluster, nodes don't own storage devices or the metadata for a volume. When a CNode needs to read from a file, it accesses that file's metadata from SCM SSDs to find the location of the data on hyperscale SSDs and then reads the data directly from the hyperscale SSDs. There's no need for that CNode to ask other nodes for access to the data it needs. Each CNode can process simple storage requests, such as read or write to completion, without consulting the other CNodes in the cluster.

More complex requests like database queries and VAST DataEngine functions will be parallelized across multiple CNodes by the VAST DataEngine, but we'll talk about that when we talk about The VAST DataBase and The VAST DataEngine below.

Stateless Containers

The CNode containers that run the DASE cluster are stateless, meaning that any user request or background task that changes the system's state, from a simple write to internal processes like garbage collection or rebuilding after a device failure, is written to multiple SDDs in the cluster's DBoxes before it is acknowledged or finally committed. CNodes do NOT cache writes and metadata updates in DRAM or even power-failure-protected NVRAM.

Where NVRAM appears safe, its contents are really only protected against power failures. Even then, data could be lost if the wrong two nodes fail, and the system must run special, and therefore error-prone, recovery routines to restore the data saved in the NVRAM or NVDIMM's flash back to RAM on power failure.

The best demonstration of this occurred a few years ago at a data center in the Boston area, which houses HPC and research computing systems for many of the area's universities. The data center suffered a power failure, and of the many storage systems housed there, only the VAST clusters simply returned to operation when power was restored. The other systems required their administrators and/or vendor support to take some action.

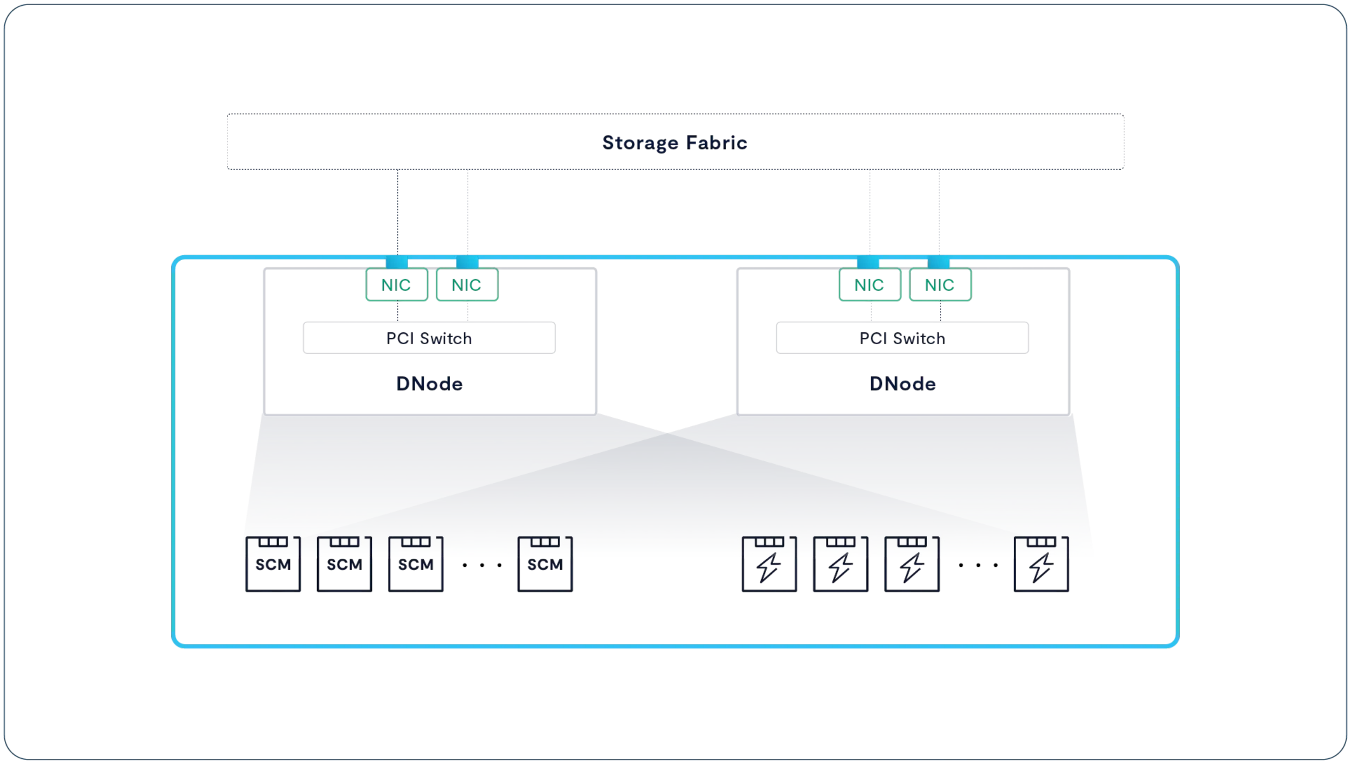
The ultra-low latency of the direct NVMe-oF connection between CNodes and the SSDs in DBoxes also relieves CNodes of the need to maintain read or metadata caches in DRAM. When a CNode needs to know where byte 3,451,098 of an Element is located, the metadata's single source of truth for that information is just microseconds away. Because CNodes don't cache, they avoid all the complexity and east-west, node-to-node traffic required to keep a cache coherent across the cluster.

Containers make it simple to deploy and scale The VAST AIOS as software-defined microservices while laying the foundation for a much more resilient architecture where container failures are non-disruptive to system operation. Legacy systems must reboot nodes to instantiate a new software version, which can take a minute or more as the node's BIOS performs a power-on self-test (POST) on the node's DRAM. The upgrade process for VASTOS instantiates a new VASTOS container without restarting the underlying OS, which reduces the time a VAST server is offline to at most a few seconds.

The combination of statelessness and rapid container updates enables VAST systems to perform all system updates, from BIOS and SSD firmware re-flashes to simple patches, non-disruptively, even for stateful protocols like SMB. In most cases, a VAST cluster updates a CNode by spinning up an instance of the new version of the VAST container and then making that container active. Containerization reduces node downtime by running both the old and new versions of the container simultaneously, allowing for a seamless switch in seconds. Traditional solutions must take the node offline to reboot the operating system or monolithic application.

HA Enclosures (DBoxes)

VAST Enclosures, also known as DBoxes (for Data bBxes), are NVMe-oF storage shelves that connect their SCM and hyperscale flash SSDs to an ultra-low latency NVMe fabric using Ethernet or InfiniBand. All DBoxes are highly redundant with no single point of failure. The DNodes that route NVMe over Fabrics requests between the NVMe fabric network and SSDs, NICs, fans, and power supplies are all fully redundant, making the clusters highly available regardless of whether they have one DBox or 1,000 HA DBoxes.



As you can see in the figure above, each DBox houses multiple DNodes that are responsible for routing NVMe-oF requests from their fabric ports to the enclosure's SSDs through a complex of PCIe switch chips on each DNode.

With no single point of failure from network port to SSD, DBoxes combine enterprise-grade resiliency with high-throughput connectivity. While at face value, the architecture of a DBox appears similar to a dual-controller storage array, there are, in reality, several fundamental differences:

- DNodes do not execute any of the storage logic of the cluster; thus, their CPUs can never become a bottleneck as new capability is added to the VAST AIOS.
- Unlike legacy controllers, the DNodes don't aggregate SSDs to LUNs or provide data services. Instead, these servers need little more than the most basic logic to present each SSD to the Fabric and route requests to/from SSDs within microseconds. One Gemini-supported enclosure, Ceres, consumes less power by using ARM DPUs as DNodes.
- The DNodes within a DBox run active-active. Under normal conditions, each DNode connects its share of the DBox's SSDs to the NVMe fabric. When a DNode goes offline, the DBox's PCI switches remap the PCIe lanes from the failed DNode to the surviving DNode(s) while retaining atomic write consistency.
- Some DBoxes implement four DNodes in two hot-swappable DTrays. Should either of the DNodes in a DTray go offline, the DBox will switch all the SSD connections to the DNodes in the surviving DTray to avoid an additional reconfiguration when the DTray is replaced.

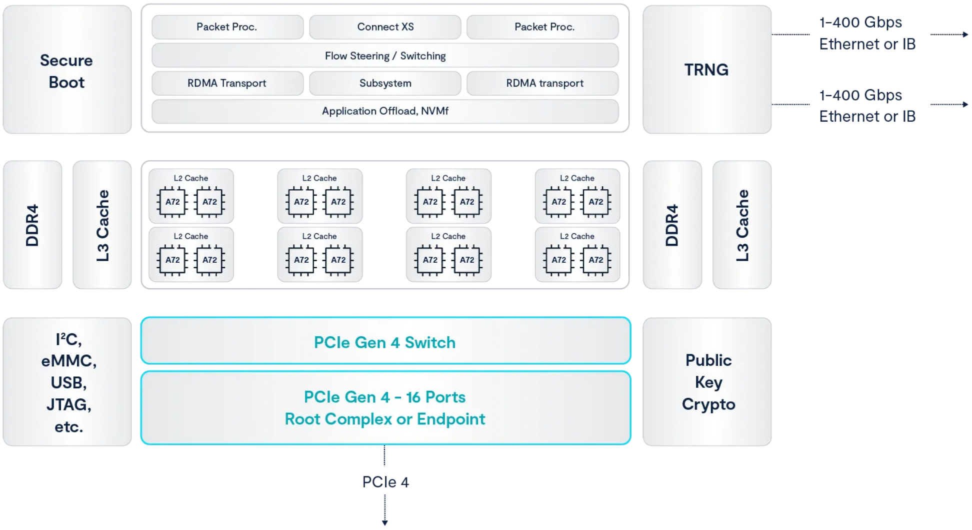
Introducing the Data Processing Unit

Data Processing Units, or DPUs, use an ASIC (Application Specific Integrated Circuit) that combines the usual functions of a network card with intelligence in two forms. DPU ASICs implement functions like RDMA, encryption, and even NVMe-oF in optimized hardware to accelerate those functions and offload those compute-intensive tasks from the server's CPU.

Individual tasks like TCP processing or encrypting data don't seem that demanding until you factor in that a server today has to do all that processing on multiple 200 or 400 Gbps ports.

Cloud providers use DPUs like Amazon's Nitro series to offload networking and security tasks, allowing their hosts to run more user instances and the cloud provider to make more money.

DPU ASICs also include a set of computing cores, almost always based on the ARM architecture, that customers, like VAST, can use to run their own software.


Secure Boot
Packet Proc.
Connect XS
Packet Proc.
Flow Steering / Switching
RDMA Transport
Subsystem
RDMA transport
Application Offload, NVMf
TRNG
1-400 Gbps Ethernet or IB
1-400 Gbps Ethernet or IB
DDR4
L3 Cache
L2 Cache
A72
A72
DDR4
L3 Cache
I²C, eMMC, USB, JTAG, etc.
PCIe Gen 4 Switch
PCIe Gen 4 - 16 Ports Root Complex or Endpoint
Public Key Crypto
PCIe 4

VAST's manufacturing partner AIC uses Nvidia Bluefield DPUs to run VAST DNode containers in the Ceres line of DBoxes. The Bluefield 3 DPU, shown in the functional diagram above, runs the VAST DNode container on its ARM cores while also taking advantage of the ASIC's encryption and other offloads. The result is a DBox that uses less power and, therefore, produces less heat than JBOFs using x86 processors as NVMe routers.

Storage Class Memory

The term Storage Class Memory (SCM) refers to a class of technologies that offer lower write latency and greater endurance than commodity NAND flash, bridging the gap between flash and DRAM in terms of performance with a new, persistent layer. While the term was once used to describe novel memory technology, including Optane, the SCM performance and endurance niche is now also filled with SSDs using SLC (Single-Level Cell) flash.

The VAST DataStore leverages SCM SSDs both as a high-performance write buffer and as a global metadata store. SCM SSDs were chosen for their low write latency and long endurance, which allows DASE clusters to extend the endurance of their hyperscale flash and provide sub-millisecond write latencies without the complexity of DRAM caching.

Each VAST cluster includes tens to hundreds of terabytes of SCM capacity, which provides the VAST DASE architecture with several architectural benefits:

- Protects Low-Endurance flash from Transient Writes: Data can live in the SCM write buffer indefinitely, and because the buffer is so comparatively large, it relieves the hyperscale SSDs from the wear of many intermediate updates.
- Data Protection Efficiency: The SCM write buffer provides the capacity to assemble many wide, deep stripes concurrently and write them in a near-perfect form to hyperscale SSDs to get 20x more longevity from these low-cost SSDs than classic enterprise storage write patterns can achieve.
- Protects Low-Endurance flash from Aggressive Data Reduction: SCM enables the VAST Cluster to perform data reduction calculations after the write has been acknowledged to the application but before the data is migrated to hyperscale flash – thus avoiding the write-amplification created by post-process approaches to data reduction.
- Data Reduction Dictionary Efficiency: SCM acts as a shared pool of metadata that stores, among other types of metadata, a global compression dictionary that is available to all of the VAST Servers. This allows the system to enrich data with much more data reduction metadata than classic storage can (to further reduce infrastructure costs) while avoiding the need to replicate a data reduction index into the RAM space of every VAST Server (a classic problem with deduplication appliances).

Hyperscale Flash

Hyperscale flash refers to the types of SSDs used by hyperscalers like Facebook, Google, and Baidu to minimize flash costs. Because hyperscalers build their systems very differently from enterprise SAN arrays, hyperscale SSDs are different from both enterprise and consumer SSDs.

Enterprise SSDs are designed for enterprise SAN arrays where dual-port drives have long been connected to dual controllers and to deliver consistently low write latency. That means enterprise SSDs need expensive components like dual-port controllers, DRAM caches, and power protection circuitry, along with flash overprovisioning to manage endurance for the 4KB random-write-heavy JEDEC testing regime.

The hyperscalers build storage systems from servers that can only connect to one port on an SSD, only writing large objects, and therefore don't need dual ports, a DRAM cache, or much overprovisioning. Hyperscale SSDs make flash affordable by using the densest available flash--currently four-bit per cell QLC--and delivering that capacity directly to storage.

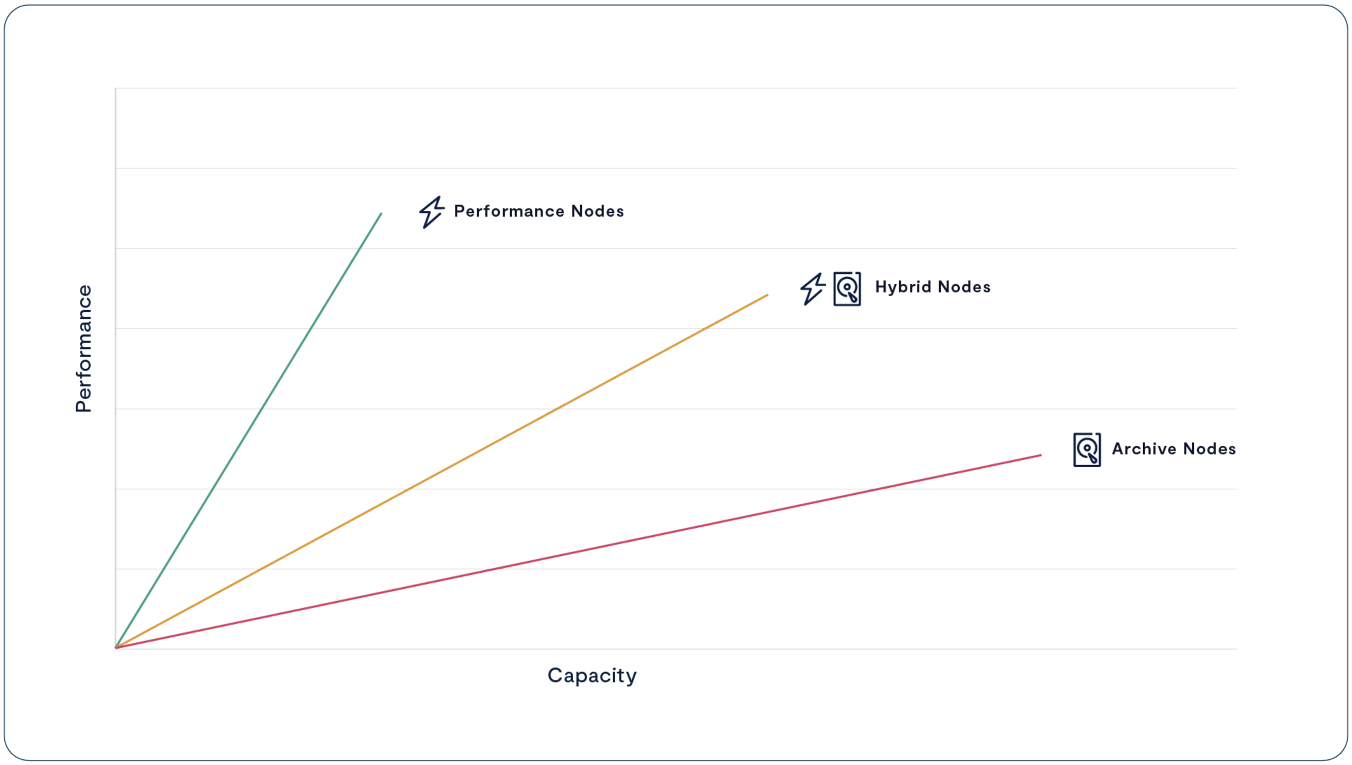
Squeezing another bit into each flash cell boosts capacity, and since it doesn't significantly increase manufacturing costs, it brings the cost per GB of flash down to unprecedentedly low levels. Unfortunately, squeezing more bits in each cell has a trade-off. As each successive generation of flash chips reduced cost by fitting more bits in a cell, each generation also had lower endurance, wearing out after fewer write/erase cycles. The differences in endurance across flash generations are huge – while the first generation of NAND (SLC) could be overwritten 100,000 times, QLC endurance is 100x lower. As flash vendors promote their upcoming PLC (Penta Level Cell) flash that holds 5 bits per cell, endurance is projected to be even lower.

Erasing flash memory requires high voltage that causes damage to the flash cell's insulating layer at a physical level. After multiple cycles, enough damage has accumulated to allow some electron leakage through the silicon's insulating layer. This insulating layer wear is the cause of high bit density flash's lower endurance. For a QLC cell, for example, to hold a four-bit value, it must hold one of 16 discrete charge/voltage levels, all between 0 and 3 volts or so. Holding that many bits as slightly different voltage levels makes QLC more sensitive to electron leakage through the insulating layers. As the number of bits that have to be represented by a voltage between 0 and 3 volts increases, the difference between one value and another shrinks, making each generation of flash more sensitive to the escape of a few electrons from a cell. This allows low bit density flash to absorb more damaging erase cycles before leakage changes a 1 to a 0 or vice versa.

VAST's Universal Storage systems were designed to minimize flash wear in two ways: first, by using innovative new data structures that align with the internal geometry of low-cost hyperscale SSDs in ways never before attempted; and second, by using a large SCM write buffer to absorb writes, providing the time, and space, to minimize flash wear. The combination allows VAST Data to support our flash systems for 10 years, which has its own impact on system ownership economics.

Asymmetric Scaling

Legacy scale-out architectures aggregate computing power and capacity into either shared-nothing nodes with a single “controller” per node or shared-media nodes with a pair of controllers and their drives. Either way, users are forced to buy computing power and capacity together across a limited range of node models while balancing the cost, performance, and data center resources needed by a small number of large nodes vs. a larger number of nodes with less capacity per node.



The DASE architecture eliminates these limitations as the simple result of disaggregating a VAST system’s computing power into CNodes that are independent of the DBoxes that provide capacity. VAST customers training their AI models (a workload that accesses a large number of small files) or processing complex queries using The VAST DataBase will use as many as a dozen or more CNodes per DBox. At the other extreme, customers using their VAST clusters to store backup repositories, archives, and other less active datasets typically run their clusters with less than one CNode per DBox.

When a VAST customer needs more capacity, they can expand their VAST cluster by adding more DBoxes without the cost (and not insignificant power consumption) of adding more compute power. This is a feature of old-school scale-up storage systems that the scale-out vendors abandoned.

Expanding capacity alone is cost-effective and old-school cool, but the only path legacy vendors ever provided to increase the amount of computing power in a cluster for a given capacity was to use a forklift to install new, faster nodes or add more nodes with smaller drives, which also boosts the CPU power per Petabyte.

When VAST customers discover their new AI engine can extract value from what they thought was a cold archive, or the new version of their key application starts performing a lot more small random I/Os than the old one, or simply that their new application is more popular across the company than the thought it would be, they can add more computing power to their cluster by adding more CNodes. The system will automatically rebalance VIPs (Virtual IP Addresses) and processing across the new CNodes when they’re added to a pool.

Asymmetric and Heterogeneous

Shared-nothing users face a difficult challenge when their vendors release a new generation of nodes. Because all the nodes in a storage pool have to be identical, a customer with a 16-node cluster of 3-year-old nodes who needs to expand capacity by 50% faces two choices:

- Buy 8 additional nodes and extended support for the current 16 nodes
    - Extended support only available for total 5 or 6 years so replacement of all 24 nodes will be required in 2-3 years
- Buy 5 new model nodes with twice the density and create a new pool
    - Performance will depend on pool data is in
    - Multiple pools add complexity
    - Lower efficiency from small cluster

This gets especially dicey when new features, like inline deduplication and compression, require the faster processor of the new model nodes.

When we say that DASE is an asymmetric architecture, we don't just mean that customers can vary the computing power per petabyte of their systems by adjusting the number of servers (CNodes) per storage enclosure (DBox), or petabyte. Asymmetric also means that DASE systems support asymmetry across the servers running the VAST CNode containers, DBoxes, and the SSDs inside those enclosures.

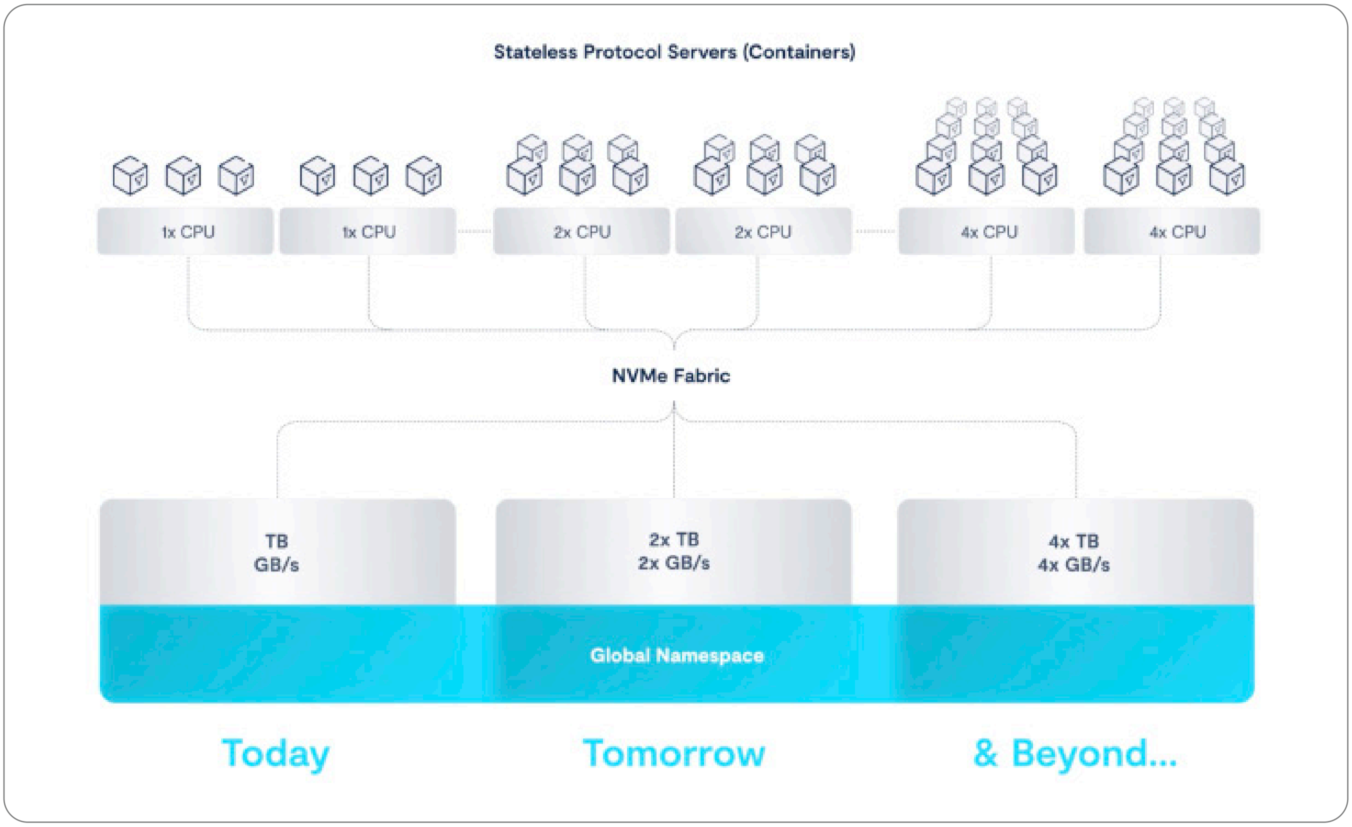
VAST systems accommodate CNodes with different numbers or speeds of cores by treating the CNodes in a cluster as a pool of computing power, scheduling tasks across the CNodes the way an operating system schedules threads across CPU cores. When the system assigns background tasks, like reconstructing data after an SSD failure or migrating data from the SCM write buffer to hyperscale flash, tasks are assigned to the servers with the lowest utilization. Faster CNodes will be capable of performing more work and will therefore be assigned more work to do.

DASE systems similarly manage the SSDs in the cluster as pools of available SCM and hyperscale flash capacity. Every CNode has direct access to every SSD in the cluster, and the DNodes provide redundant paths to those SSDs, so the system can address SSDs as independent resources and failure domains.

When a CNode needs to allocate an SCM write buffer, it selects the SCM SSDs that have the most write buffer available and are as far apart as possible, always connected to the fabric through different DNodes, in different DBoxes if possible. Similarly, when the system allocates an erasure-code stripe on the hyperscale flash SSDs, it selects the SSDs in the cluster with the most free capacity and endurance for each stripe.

Since SSDs are chosen by the amount of space and endurance they have remaining, any new SSDs or larger SSDs in a cluster will be selected for more erasure-code stripes than the older, or smaller SSDs until the wear and capacity utilization equalize across the SSDs in the cluster. See A Breakthrough Approach To Data Protection below for more details.

The result is that VAST customers are never faced with choosing between buying a little more of the old technology they're already using, knowing these nodes will have a short working life, or replacing the whole cluster, even though their current nodes have a few more years of life. When a VAST customer's cluster requires more compute power, they don't have to upgrade all their nodes to the new model with a faster processor; instead, they can just add a few CNodes.



VAST is committed to supporting clusters with as many as three generations of hardware and to supporting any VAST-branded CBox or DBox for up to 10 years under Gemini. This allows VAST customers to run VAST clusters for extended periods by adding new hardware and retiring old hardware from the cluster. In any case, data and work are balanced automatically and transparently, eliminating not just the forklift but all the drama from upgrades.

Variations on DASE

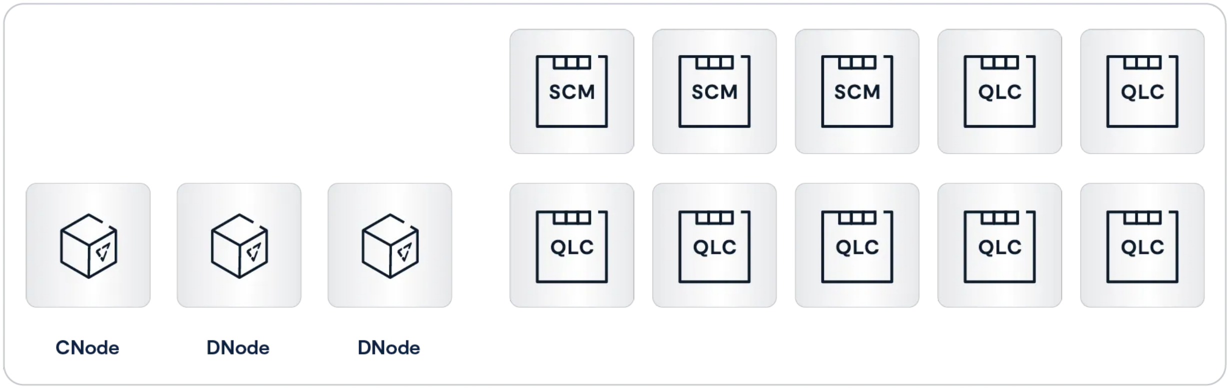
Our original model for DASE, utilizing highly available DBoxes to store persistent data and state, is optimized for efficiency. Since the DBox's redundant DNodes and power supplies ensure that no single component failure can take the DBox's SSDs offline, the VAST DataStore can treat each SSD as an independent failure domain relying on the DBox to provide fault tolerance. This, in turn, allows the VAST DataStore to write very wide erasure code stripes that protect against as many as four SSD failures with as little as 2.7% overhead.

DBoxes provide high efficiency, and resilience but one size fits all solutions rarely actually fit everyone so we've expanded the DASE architecture to include configurations that don't include highly available JBOFs like DBoxes. Cloud providers, hyperscalers and enterprises that build their data centers along the hyperscaler model wanted to run the VAST AIOS on the hardware they had very carefully engineered and on virtual machine cloud instances while server OEMs like SuperMicro, Lenovo and Cisco wanted to sell VAST clusters running on their standard servers so we've adapted the DASE architecture to support these variations.

Introducing the E for Everything Box

The first step in broadening the VAST hardware universe was adapting DASE to run on industry-standard servers functioning as both the CNodes and DBoxes. The easiest way to do that would have been for us to simply assign servers to be either CNodes or DBoxes. The DBoxes would hold all the SCM and hyperscale flash SSDs, and CNodes would access those SSDs just as if they were the HA DBoxes in a VAST Classic cluster. This solution would work; however, it would require a large number of servers, reducing system density while increasing power and data center costs.

Instead of dedicating servers to the DBox role, and since DBoxes don't do much computational work, wasting the compute power in those servers, EBoxes, the E stands for Everything, take advantage of The VAST AIOS's containerized architecture to combine the CNode and DBox functions into a single server.



Each EBox runs three VAST containers. One CNode container that, just like a CNode container running on a server dedicated to being a CNode, mounts all the SSDs in the cluster, processes requests for files via NFS, or for the next entry in a topic via the Kafka API request, and does its share of managing the VAST Element Store.

Like the DNodes in a DBox, the DNode containers connect the SCM and hyperscale flash SSDs in the EBox to the cluster's NVMe fabric, routing NVMe commands and data between the SSDs and the cluster's CNodes. Under normal conditions, each DNode connects half of the EBox's SSDs to the cluster's NVMe fabric. Should one DNode go offline, such as during an update, the other DNode will connect all the EBox's SSDs to the fabric.

While EBoxes have redundant DNode containers just like DBoxes, unlike DBoxes, they are just industry-standard servers that pass VAST's strict validation testing, which means that, unlike DBoxes, they have multiple single points of failure (motherboard, memory DIMMs, etc).

As a result, EBoxes will go offline more often than DBoxes, so EBox clusters are designed to remain available even when two EBoxes go offline simultaneously. This is accomplished by erasure coding data in the cluster's write buffer with double-parity erasure codes and 3-way mirroring the cluster's metadata.

Together, these features, along with VAST's locally decodable codes and an optimized data layout, provide n+2 protection for all the system's data and metadata so that an EBox cluster will remain fully functional through the simultaneous failure of two EBoxes. Customers seeking even higher levels of fault tolerance can implement Rack Level Resilience, which can keep the system running, with all of its contents available, through full rack failures.

EBox clusters, like all VAST clusters, are designed to allow components, individual boxes, or even full racks to fail in place, rebuilding the system to a fully protected state even after multiple consecutive failures. The only limitation on this ability to rebuild is the availability of free space on the system to rebuild onto.

Truly Non-Disruptive Upgrades

Shared-nothing clusters try to provide non-disruptive upgrades by rolling the update process across the cluster's nodes, updating the software on, and rebooting one or two nodes at a time. Since the SSDs in each can only be accessed by that node's CPU, the cluster has to operate in a degraded state when any of its nodes is rebooting, or offline because an update failed.

While we could expect that a cluster with a node down for update to deliver lower performance because the cluster is now running 19 or 39 nodes instead of 20 or 40, the problem is bigger than that. Not only will the cluster be busy rebuilding all the data on the node that is now offline, but even more significantly, the cluster will have to reconstruct any data on the offline node's drives when that data is read turning one read I/O from a client into 8 or 16 I/Os to recall the entire erasure code stripe.

EBoxes run multiple containers to avoid this problem by almost never , taking the whole EBox offline to update. When the system needs to update an EBox it:

Updates the CNode:

- A container running the new version of the CNode function is spun up on the EBox in parallel with the existing CNode
- The virtual IP addresses being serviced by the EBox's CNode container are transferred to the new CNode container so clients, even SMB clients, can continue accessing their data.
- The old CNode container is spun down.

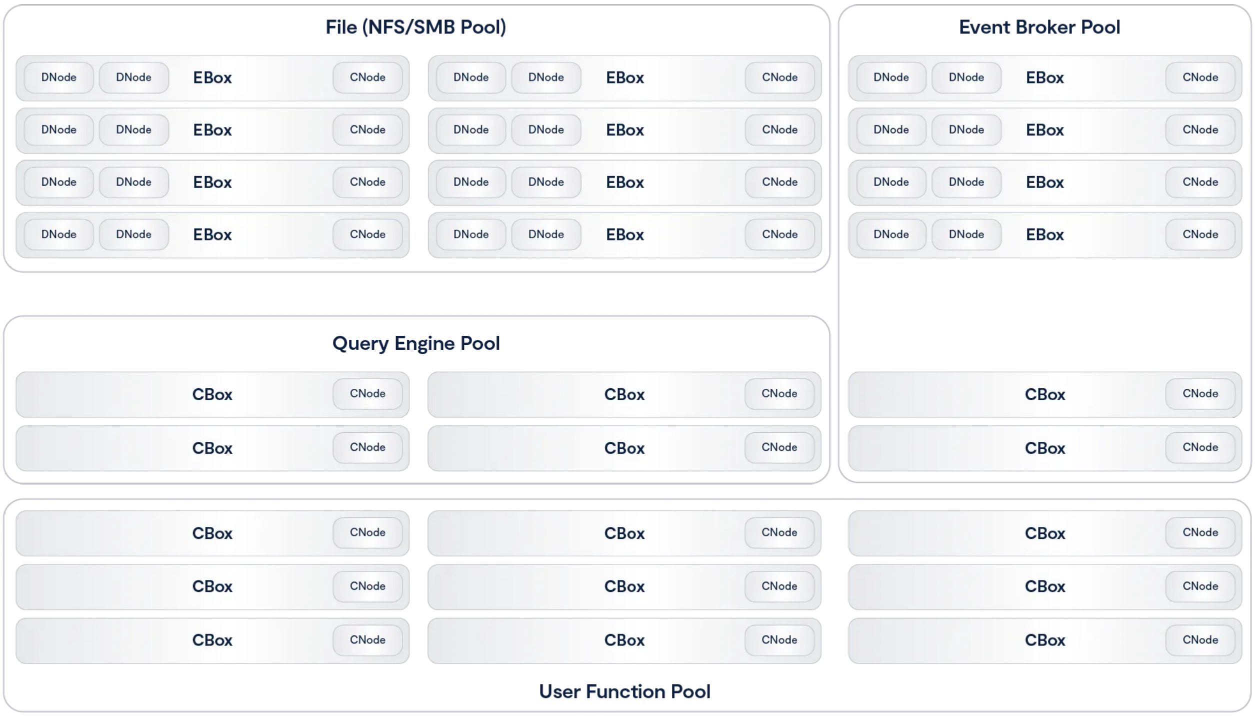
Updates the DNode containers

- One of the two DNode containers in the Ebox (DNodeA) is selected for update first
- The SSDs being connected via DNodeA are transferred to DNodeB
- A new container is spun up for DNodeA
- The SSD connections are transferred to the new version of DNodeA
- The old DNodeA container is spun down
- A new DNodeB container is spun up
- The new DNodeB container takes over connections for half the SSDs
- The old DNodeB container is spun down

All of the EBox's SSDs remain online and available through the upgrade, limiting the performance impact of the upgrade.

Asymmetric Scaling with EBoxes

Asymmetric scaling has always been one of the key advantages of the DASE architecture, allowing VAST customers to add capacity to their by adding DBoxes and compute power by adding CNodes. This ability to expand the computing power of a cluster is key to the VAST AI OS's ability to not just provide the persistent storage for your AI pipelines, but also the query engine for the pipeline's data warehouse, the pipeline's orchestration and automation engine, and a managed Kubernetes cluster to execute the pipeline's functions.



Extending Resiliance Beyond the DBox

VAST's original data layout maximizes efficiency by writing very wide erasure code stripes across a cluster's SSDs without regard to which DBox held each SSD, relying on the highly available DBox to provide resilience. Since it takes multiple device failures to take the SSDs in a DBox offline, the system can write many more strips of each erasure code stripe to each DBox in the cluster than the four erasures locally decodable codes could correct for losing.

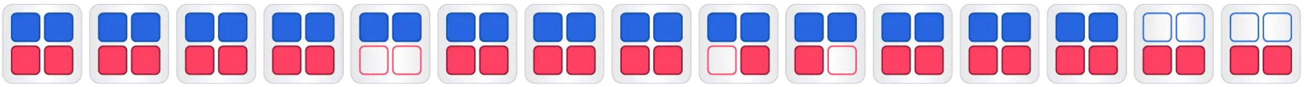
A VAST classic cluster with over 150 hyperscale SSDs will use erasure code stripes of 146 data strips plus 4 parity strips (146D+4P), providing protection from up to four SSDs going offline concurrently with just 2.7% overhead. Still, since each enclosure holds more strips of each erasure code stripe than the four losses the system can correct for, the system will go offline when any of the cluster's DBoxes goes offline.

While locally decodeable codes and DBoxes provide a higher level of resilience than classic enterprise systems, even the most resilient appliance remains vulnerable to power or network service failures in the rack they're housed in. When we started introducing VAST into hyperscalers who designed their applications to accommodate whole racks of servers going offline, and then decided to save money by building out their data centers with one power feed or top-of-rack switch per rack, we realized we needed to expand high-availability beyond the resilience provided by a DBox.

Maintaining Availability through DBox Failures with DBox-HA

Our first step in expanding resilience beyond a DBox, and addressing the age-old question "Sure, I know DBoxes are highly available with no single point of failure, but what happens when a DBox blows up?" was DBox-HA. DBox-HA uses all the same data protection methods as VAST classic, but it redefines the domains of failure it protects the system from up from individual SSDs to DBox so a VAST cluster can continue operating through a Dbox failure. The failure of a DBox takes all the SSDs in that DBox offline, so the system has to be able to rebuild any of the data from the missing DBox when users read it.

To ensure that all the data in a cluster is still available, even with a DBox offline, VAST clusters running DBox-HA stripe limit the portion of each erasure code stripe written to each DBox to two data and/or parity strips, as shown below. That limit of two strips of each erasure code stripe per DBox means that a DBox going offline causes two erasures in every erasure code stripe, which is well below the four erasures that VAST's locally decodable codes can correct for allowing VAST clusters running DBox-HA to continue to operate, and access all of their contents, even with a DBox offline.

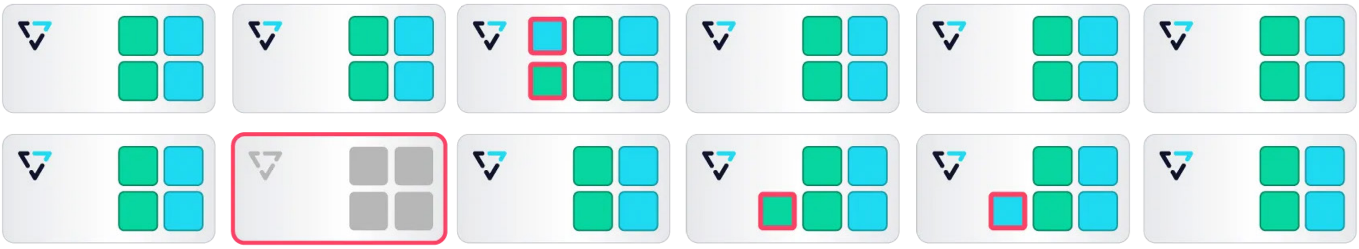


DBox HA Writes Two Data or Parity Strips Per DBox

As the great Robert A Heinlein wrote, "There Ain't No Such Thing As A Free Lunch," and DBox-HA is no exception, the two strips per DBox limit means that DBox-HA clusters write narrower, and therefore somewhat less efficient erasure code stripes than VAST classic clusters that rely on the inherent high-availability of DBoxes. A cluster of ten Ceres DBoxes with 220 hyperscale flash SSDs will write erasure code stripes of 146 data strips and four parity strips (146D+4P), which has 2.7% overhead. That same cluster running DBox-HA will write erasure code stripes with 36 data strips (36D+4P), raising the data protection overhead from 2.7% to 10%, which is still significantly less than the 20% or greater overhead of legacy storage solutions.

Even better, unlike many shared-nothing systems, VAST clusters not only reconstruct the data from the offline DBox to satisfy read requests but also restore the data to full protection. When a DBox fails in a cluster running DBox-HA, the cluster begins rebuilding the data on its SSDs shortly after the failure is detected.

The system writes 1 reconstructed strip of each erasure-code stripe to 2 of the surviving DBoxes. Since each DBox in a DBox-HA cluster already holds 2 strips of every erasure-code stripe, and these additional strips are distributed across all the surviving DBoxes, each DBox will hold 3 strips of some number of erasure-code stripes as shown outlined in red below.



Restriping After a DBox Failure

The rebuild process will continue until all the cluster's erasure code stripes are back to full +4 protection, and a cluster will be able to rebuild from multiple failures as long as there is free space to rebuild into without writing more than three strips of any strips to any DBox.

It is, however, much more common for the DBox to come back online when the circuit breaker is reset, or the DBox is installed in its new rack or data center cage. When a DBox returns to a cluster, there's no need to keep rebuilding data and so the rebuild stops.

Since VAST clusters determine the erasure code stripe width based on the number of active DBoxes when the stripe is written, any new erasure code stripes that are written while a DBox is offline will be written slightly narrower (34D+4P for our example cluster with nine active DBoxes vs 36D+4P when all 10 DBoxes were online). These new stripes are fully protected, and there's no need to rebuild them when the disconnected DBox comes back online.

When that DBox returns online, some of the existing erasure code stripes will not yet have been rebuilt. Those stripes are now once again complete with our wayward DBox back online and don't need to be rebuilt. Other stripes will have already been rebuilt when the DBox returns. Since these stripes have been rebuilt, the system marks the space used by rebuilt stripes on the returning DBox as free space.

The other cost of DBox-HA is a larger minimum cluster size. VAST classic clusters can run on a single DBox, but DBox-HA requires a minimum of eight DBoxes to manage both erasure code stripe width and, therefore, efficiency, and the impact of a DBox failure.

Write Buffer Erasure Coding

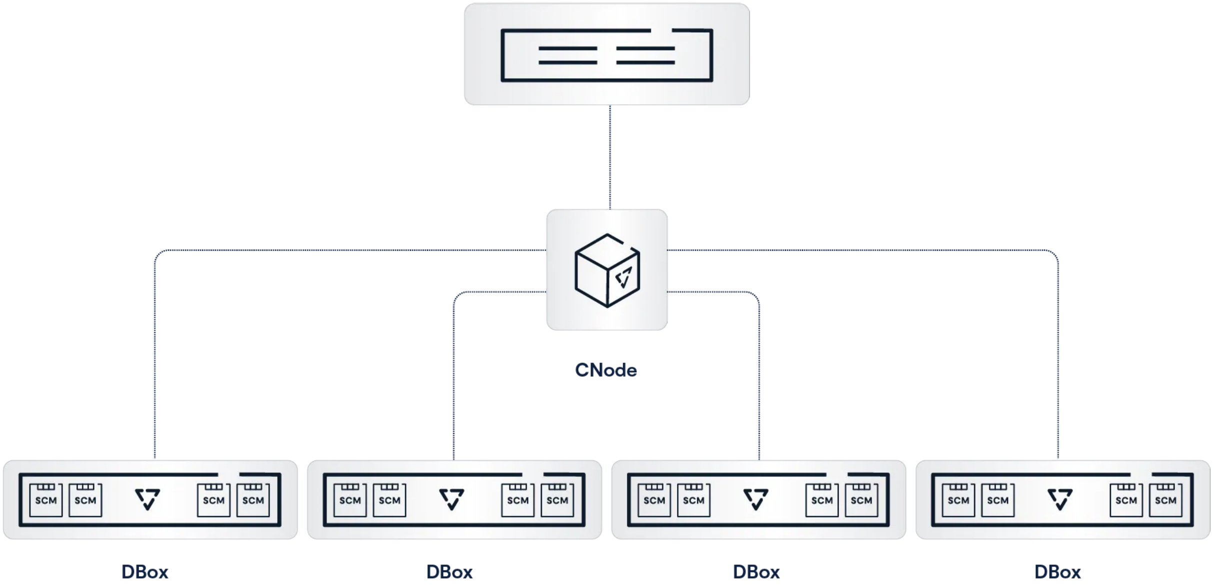
As we've already established, when a client, or AI pipeline function running on the VAST cluster itself, writes new data to a VAST cluster, the CNode that receives that new data saves that data to multiple SSDs before acknowledging the write to the client. VAST classic clusters rely on the physical redundancy of DBoxes to ensure that CNodes can access the SSDs even if a DNode, power supply, or network card in the DBox fails, mirroring the write buffer across two SCM SSDs with DBox and DNode anti-affinity, writing each of the two replicas of the data to different SSDs connected to the NVMe fabric through different DNodes and in different DBoxes if multiple DBoxes are available.

Mirroring ensures that the data in the write buffer remains available through any single SSD or DNode failure and introduces very little latency, but mirroring isn't perfect. The obvious problem is that two-way mirroring only protects the data from the loss of one of the two replicas of that data. Should both replicas go offline, that data will be unavailable until one of the replicas can be restored.

The other problem with mirroring is that fully 50% of the write buffer space, and even more significantly, half the write bandwidth to the SSDs, is consumed by overhead, reducing a cluster's ability to accept writes.

More recent releases of the VAST AI OS have transitioned from using 2-way mirroring to protect data in the write buffer to protecting that data with double parity erasure codes, striping data across as many as 12 SCM SSDs, reducing the bandwidth consumed for data protection from 50% to 16.7% significantly boosting cluster write bandwidth.

As a CNode receives write requests, it accumulates newly written data in memory until it has enough data for a full erasure code stripe, calculates parity, and writes the data to the buffer before acknowledging the write(s) to the client(s). To minimize write latency during periods of low to moderate write traffic, should a CNode not receive sufficient data to build a full erasure code stripe in 300 μsec, it will pad the data it holds in memory with zeros and write the stripe.



How the cluster distributes erasure code strips depends on the cluster's availability level:

- VAST Classic clusters stripe their write buffers across the cluster's SCM SSDs, protecting the write buffer against up to SCM SSDs going offline
- Clusters running DBox-HA stripe their write buffer across the DBoxes in the cluster, placing two strips per DBox, protecting the write buffer against one DBox or two SCM SSDs going offline
- EBox clusters, and clusters running with Rack Level Resiliance stripe data across their EBoxes or racks with no more than 1 strip per Ebox or rack, protecting the write buffer from up to two racks going offline

Write Buffer Bursting

We've discussed in this paper how VAST's DASE uses high-endurance storage-class memory to absorb writes to the system and thereby extend the more limited endurance of the low-cost hyperscale flash SSDs. Having all writes land on SCM before being migrated onto the hyperscale SSDs extends the hyperscale SSD's endurance, but it also limits the write bandwidth of a VAST cluster to the bandwidth of the relatively small number of SCM SSDs in the cluster.

This limitation on write bandwidth is problematical for some leading-edge applications, including some HPC simulations and AI model training. These applications run on leading-edge computing environments with the latest CPU, GPU, and networking technologies. Unfortunately, these leading-edge environments are frequently not the most reliable environments, with servers crashing because of overheating or just from running software developed by that somewhat disheveled post-doc who sleeps under his desk last night.

Training an AI model or running a detailed simulation of a nuclear weapon aging in its silo, or a car crashing, requires running a single compute job across hundreds or thousands of processors for days or weeks. To prevent the job from having to restart from the beginning when any of the many servers it's running across crashes, these applications write periodic checkpoints, saving the state of the application to persistent storage. Now, when a server crashes or a top-of-rack switch takes a rack offline, the application can restart its work from the last checkpoint, instead of from the beginning of the job.

While AI training models have predominantly shifted to asynchronous checkpointing, older libraries and many HPC applications will still checkpoint synchronously, holding up their computing while the checkpoint is being written, making the speed at which a system can save a checkpoint a significant consideration for many users.

The VAST AI OS accommodates this need for periodic high-performance writes by directing the large writes created by servers dumping all or part of their memory to the cluster's hyperscale flash SSDs, as well as the SCM SSDs. Since a typical VAST cluster has roughly three times as many hyperscale SSDs as SCM SSDs, this roughly doubles the write bandwidth of the cluster for checkpoints and similar applications.

A training cluster would typically be configured to take hourly checkpoints with the system's burst write bandwidth and the size of the extended write buffer designed to save a checkpoint in roughly three minutes, so that even the most basic synchronous checkpoints would take less than 5% of the system's time to save.

During a checkpoint or other write burst, CNodes receiving write I/Os smaller than ~256 KB save that data to the SCM write buffer as usual. Larger writes are also written to a limited amount of space on the cluster's hyperscale flash SSDs using the same double-parity erasure codes used to write to the SCM. VAST users can configure the size of this write buffer burst area to manage the amount of hyperscale flash to dedicate to the extended write buffer and manage the amount of burst data that can be written.

This segregation of writes, with only large writes going to the hyperscale flash SSDs, minimizes flash wear. To minimize costs, hyperscale SSDs use flash chips with a large page size. The page size is the minimum amount of data the flash chip can store at a time. When a hyperscale SSD receives writes smaller than the 128 KB size of its internal pages, each of those writes is stored individually on a page, consuming 128 KB of endurance, even for a 4 KB write. The hyperscale SSDs used in VAST systems have twenty times the endurance when data is written to them in 128 KB or larger sequential writes than for 4 KB random writes. Add that to the small amount of hyperscale flash designated for the extended write buffer, and VAST's cluster-wide wear-leveling, and the endurance impact of write buffer bursting is minimal.

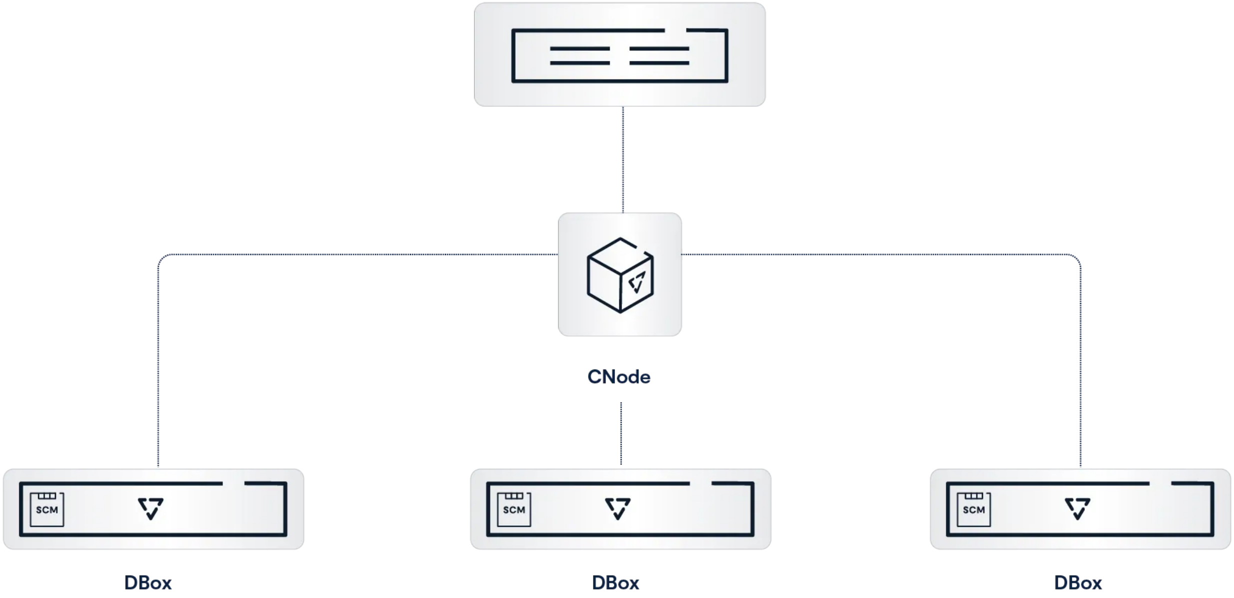
Once a checkpoint, or other write burst, ends, the system will of course, have more than the usual amount of data in the write buffer to migrate to hyperscale flash. As a result, systems will dedicate more resources to emptying the extended write buffer after a burst ends, resulting in a temporary performance dip.

Metadata Triplication

The double parity erasure codes VAST clusters use for the write buffer strike a balance between the greater efficiency of wider stripes and the impact those stripes may have on latency for typical storage write I/O patterns. Under heavy write demand, CNodes can assemble full erasure code stripes and maximize bandwidth. When demand is lower, they pad stripes with zeros, trading unused bandwidth for reduced latency. Any wasted SCM space will be quickly and inexpensively recovered when the write buffer is migrated to QLC.

As efficient as those erasure codes are for newly written data, they don't fit the I/O patterns for metadata updates. Every write or S3 PUT causes the creation of multiple new pointers, creating new 4 KB metadata blocks and new versions of existing blocks. All those small I/Os landing in the middle of erasure code stripes would trigger significant read-modify-write I/O amplification, the last thing you want for your metadata.

VAST clusters protect the system's metadata against two EBoxes, or with Rack Level Resilience, two DBoxes, going offline simultaneously by triplicating each metadata update to three SCM SSDs in different DBoxes, or EBoxes.. The CNode processing the write I/O from a client writes the new metadata block(s) to three SCM SSDs in different boxes, or racks, simultaneously. It sends an acknowledgment of the write to the client only when the NVMe over Fabrics (NVMe-oF) acknowledgment comes back from all three.



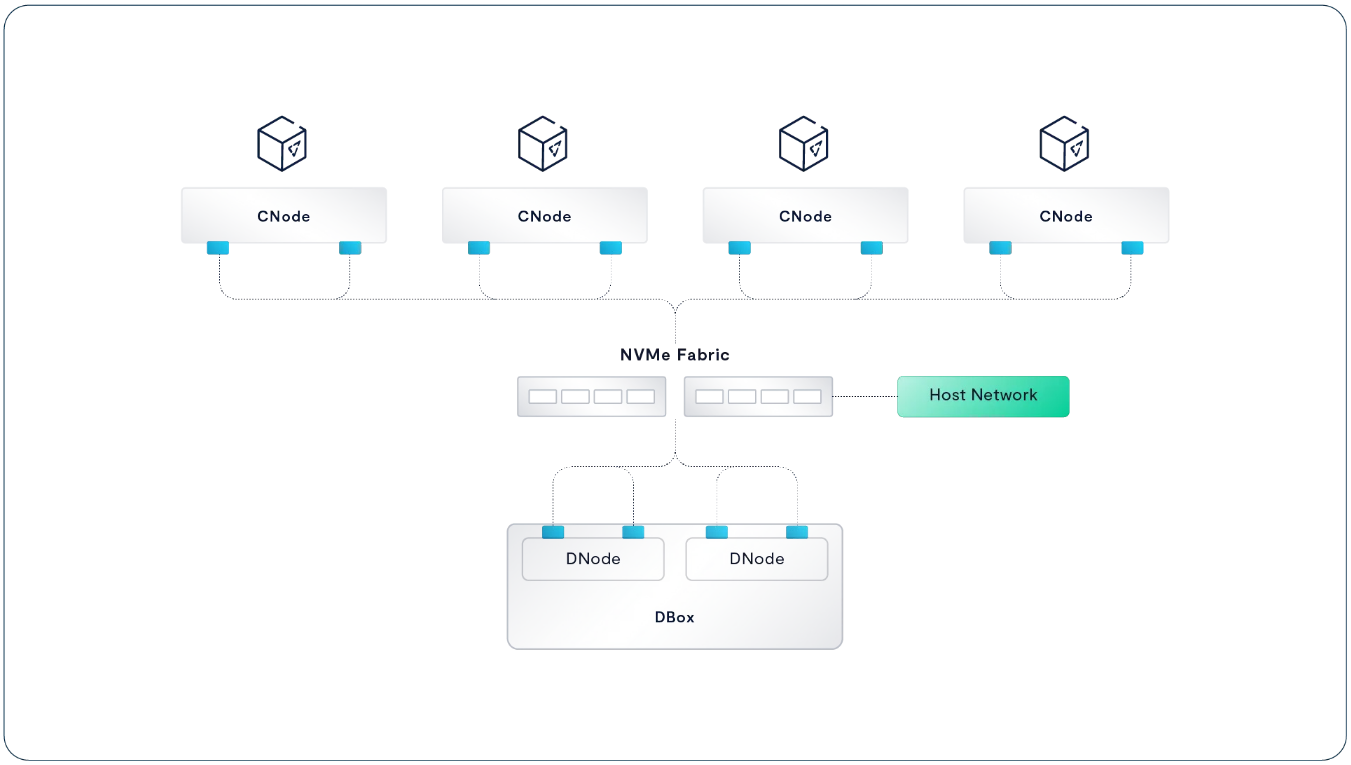
Networking in DASE

A DASE cluster includes four primary logical networks.

- The NVMe fabric, or back-end network, connects CNodes to DNodes. VAST clusters use NVMe over RDMA for CNode<->DNode communications over 100 Gbps Ethernet or InfiniBand with Ethernet as the default.
- The host network, or front-end network, that carries file, object, or database requests from client hosts to the cluster's CNodes.
- The management network that carries management traffic to the cluster, including DNS and authentication traffic.
- The IPMI network used for managing and monitoring the hardware in the cluster.

Depending on their requirements, VAST customers have several options for how to implement these logical networks using dedicated ports, and/or VLANs as their network designs and security concerns dictate. The biggest decision is how the DASE cluster is connected to the customer's data center network to provide host access.

Connect via Switch



The Connect via Switch option runs the NVMe back-end fabric and the front-end host network connections as two VLANs on the NVMe fabric switches that are included in each DASE cluster. The customer's host network is connected to the cluster through an MLAG connection from the fabric switches to the customer's core switches as shown in green above.

Each CNode has a single 100 Gbps network card. A splitter cable turns that into a pair of 50 Gbps connections, one for each of the two fabric switches. Each 50 Gbps connection carries NVMe fabric traffic on one VLAN and host data traffic on another.

The Connect via Switch method has several advantages:

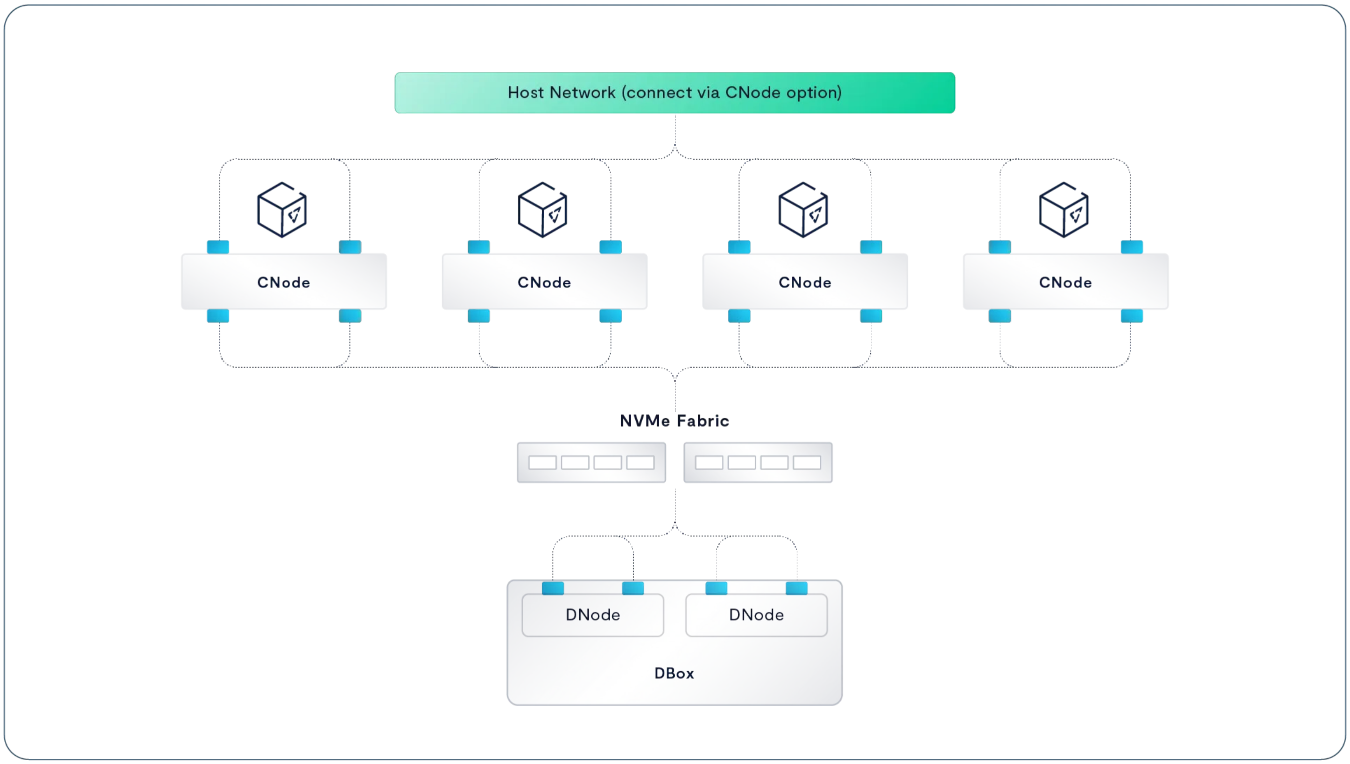
- Only one RNIC is needed in each CNode
- Network traffic is aggregated to a small number of 100 Gbps links MLAGed together minimizing the number of host network switch ports needed

If Connect via Switch was perfect we wouldn't have any other options, but there are disadvantages:

- Host network connections must be the same as the fabric
  - Infiniband fabrics only support Infiniband hosts
  - 40, 25, 10 Gbps Ethernet connections expensive in 100 Gbps fabric ports
- Only one physical host network

Connect via CNode

Connecting DASE clusters to the customer's network through the NVMe fabric switches is simple, and minimizes the number of network ports. But as we discussed in the server pooling section above, customers may need more flexibility or control over how their various clients and tenants should connect to their DASE cluster.



When VAST customers need to connect clients from multiple disparate networks with different technologies, or security considerations, they can solve that problem by connecting those networks to CNodes with a second network card that connects directly to the network that CNode will serve.

Advantages of Connect via CNode

- Connect hosts to the DASE cluster via different technologies
  - Infiniband clients served by CNodes with Infiniband NICs
  - Ethernet clients served by CNodes with Ethernet NICs
  - Or Ethernet clients connected via fabric switches
- Support for new technologies like 200 Gbps Ethernet for client connect
- Connect multiple security zones w/o routes

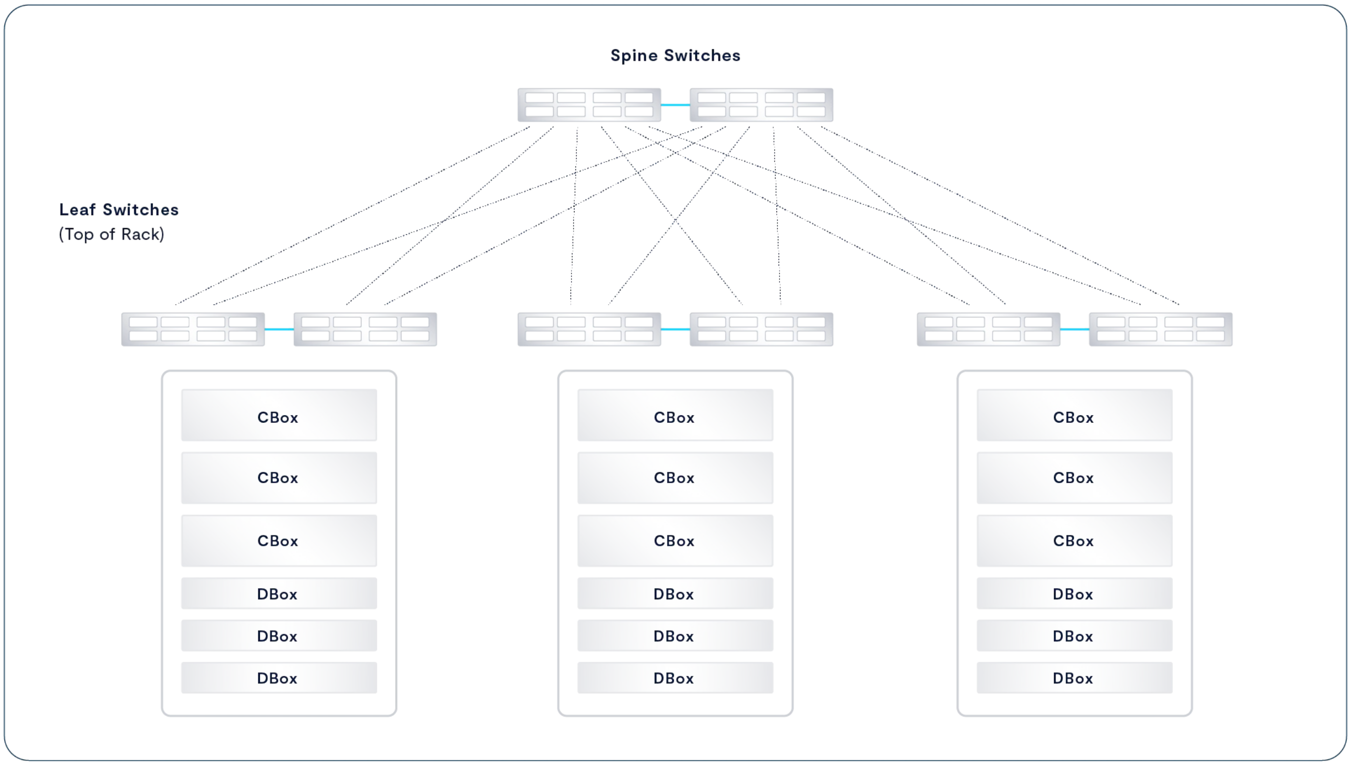
Disadvantages

- Requires more network cards, switch ports, IP Addresses, etc.

As noted in the advantages section above, VAST customers are free to mix the "connect via CNode" and "connect via switch" models. A customer with a few Infiniband hosts could install IB cards in one pool of CNodes and still connect their Ethernet clients to the DASE cluster through the Ethernet fabric switches to minimize the number of switch ports needed.

Leaf-Spine for Large Clusters

As DASE clusters grow to need more NVMe fabric connections than a pair of 64-port switches can provide, the pair of fabric switches usually shown at the core of a DASE cluster grows into a leaf-spine network. The CBoxes (multi-server chassis running multiple CNodes in a single appliance) and DBoxes are still connected to a pair of switches, but instead of that top-of-rack switch pair being the core of the cluster they become leaves redundantly connected to a pair of Spine switches, as are the leaf switches at the top of the other racks of the cluster.



Leaf-spine networking allows DASE clusters to grow to well over 100 appliances, especially when large port count director class switches are used in the spine. Planning is underway for very large clusters of 1,000 appliances or more, so the groundwork will be laid before customers reach that size.

Scale-Out Beyond Shared-Nothing

For the past decade or more, the storage industry has convinced itself that a shared-nothing storage architecture is the best approach to achieving storage scale and cost savings. Following the release of the Google File System architecture whitepaper in 2003, it became table stakes for storage architectures of almost any variety to be built from a shared-nothing model, ranging from hyper-converged storage to scale-out file storage to object storage to data warehouse systems and beyond. Ten years later, the basic principles that shared-nothing systems were founded on are much less valid for the following reasons:

- Shared-nothing systems were designed to co-locate disks with CPUs, in an era when networks were slower than local storage. However, with the advent of NVMe-oF, it's now possible to disaggregate CPUs from storage devices without compromising performance while accessing SSDs and SCM devices remotely.

- Shared-nothing systems force customers to scale compute power and capacity together, creating an inflexible infrastructure model vs. being able to scale up CPUs as the data set needs faster access performance.

- Shared-nothing systems limit storage efficiency. Since each node in a shared-nothing cluster owns some set of media, a shared-nothing cluster must accommodate node failures by erasure-coding across nodes, limiting stripe width and shard, or replicate data reduction metadata, limiting data reduction efficiencies. Shared-everything systems can build wider, more efficient RAID stripes when no one machine exclusively owns any SSDs and build more efficient global data reduction metadata structures.

- As containers become an increasingly popular choice for deploying applications, this microservices approach to deploying applications benefits from the stateless approach containers bring to the table, making it possible to easily provision and scale data services on composable infrastructure when data locality is no longer a concern.

The Advantages of a Stateless Design

When a VAST Server (CNode) receives a read request, that CNode accesses the VAST DataStore's persistent metadata from shared Storage Class Memory to find where the data being requested actually resides. It then reads that data directly from hyperscale flash (or SCM if the data has not yet been migrated from the write buffer) and forwards the data to the requesting client. For write requests, the VAST Server writes both data and metadata directly to multiple SSDs and then acknowledges the write.

This direct access to shared devices over an ultra-low latency fabric eliminates the need for VAST servers to talk with each other to service an IO request – no machine talks to any other machine in the synchronous write or read path. Shared-Everything makes it easy to linearly scale performance just by adding CPUs and thereby overcome the law of diminishing returns that is often found when shared-nothing architectures are scaled up. Clusters can be built from thousands of VAST servers to provide extreme levels of aggregate performance. The primary limiter on VAST cluster scale is the size of the network fabric that customers configure.

Storing all the system's metadata on shared, persistent SSDs across an ultra-low latency fabric eliminates the need for CNodes to cache metadata and, therefore, any need to maintain cache coherency between Servers/CNodes. Because all data is written to persistent SCM SSDs before being acknowledged to the user, not cached in DRAM, there's no need for the power failure protection hardware usually required by volatile and expensive DRAM write-back caches. VAST's DASE architecture pairs 100% nonvolatile media with transactional storage semantics to ensure that updates to the Element Store are always consistent and persistent.

The DASE architecture eliminates the need for storage devices to be owned by one, or an HA pair, of node controllers. Since all the SCM and hyperscale flash SSDs are shared by all the CNodes, each CNode can access all the system's data and metadata to process requests from beginning to end. That means a VAST Cluster will continue to operate and provide all data services even with just one VAST Server running. If, for example, a cluster consisted of 100 servers, said cluster could lose as many as 99 machines and still be 100% online.

The VAST DataStore

As its very name implies, data has always been the critical core of data processing, but for many years, organizations really only processed a small fraction of the data they had access to, or even the data they generated internally. Corporate IT departments treated the relatively small databases that ran their ERP, logistics, and other line of business applications as their crown jewels, but since the OLTP platforms that held that data could only process a limited amount of data at a reasonable cost, historical data was treated as a luxury, and unstructured data was treated as second-class.

Over the past two decades or so, that's changed. First, the so-called big data analytics platforms from Hadoop to Databricks and Snowflake made it practical to glean business insight across data from multiple applications, and database management systems, and datasets too big for classical databases. More recently, data has been central to AI model training and **RAG (Retrieval Augmented Generation).** Since AI model accuracy is directly related to the amount of data used to train and optimize the model, many organizations are facing data storage, or more broadly, a data management crisis.

The VAST DataStore provides the persistence and basic data services layers of the VAST AI Operating System. In English, that means that the VAST DataStore is responsible for storing, protecting, securing, and presenting the VAST AI OS's data. While it's not limited to the functions provided by conventional operating systems we saw in the VAST AI Operating System section above, the VAST DataStore is analogous to the logical volume manager, file system, and protocol services/daemons in traditional operating systems.

In addition to making the data it holds available across the standard storage protocols for file (NFS and SMB), object (S3 API), and block (NVMe over TCP) access the VAST DataStore also provides direct access to and manages the contents of tables for the VAST DataBase and provides data services like snapshots, and clones across its entire contents of both structured, and unstructured data.

Designing the VAST DataStore

Before VAST, the computing world assumed that the only way to efficiently store data was with multiple tiers of storage, each providing a unique price/performance proposition. Storage vendors built, and storage users bought, storage systems designed to fit one or two tiers in the storage pyramid. All-flash systems were fast, but small and expensive; shared-nothing systems scaled, but couldn't handle small files or deliver 1 ms latency.

For the VAST DataStore to truly deliver universal storage, we had to build a system that broke all the tradeoffs underlying previous solutions. It had to scale to the exabytes of data needed to train the most advanced AI models while still delivering the sub to single digit millisecond latency and five 9s reliability users expect from an all-flash array. It had to support parallel I/O from thousands of clients and deliver the strict atomic consistency to support transactional applications. Perhaps most importantly, it had to do it all at a price customers could afford when buying petabytes at a time.

The VAST DataStore is the software and metadata structures that turn the CNodes and SSDs of a DASE cluster into a coherent system that provides universal storage alongside all the AI OS's other services. It supports a wide range of applications and data types, from volumes to files and tables. The VAST DataStore takes full advantage of the low-latency direct connection from every CNode to every SSD in the cluster by optimizing metadata structures for the shared, persistent SCM SSDs.

Unlike first-generation all-flash arrays that offered low latency at limited capacity, the VAST DataStore was designed to efficiently manage petabytes to exabytes of data. This doesn't just mean efficient use of capacity through erasure codes and data reduction, though those are an important part of the VAST DataStory (Sorry, I couldn't resist the pun); it also means the efficient management of flash endurance.

The VAST DataStore manages data through three sublayers:

- The Physical or Chunk Management layer provides the basic data preservation services for the small (32 KB average) data chunks the VAST DataStore uses as its atomic units. This layer includes services such as erasure-coding, data distribution, data reduction, flash management, and encryption at rest.
- The Element Management or Namespace layer organizes the data chunks protected by the chunk management layer into Elements such as files, objects, tables, and LUNs, much the way a filesystem does in a conventional operating system. Unlike conventional file systems, the Element Store includes all the metadata that may be used by any of the access protocols and knowledge not just of the element's size and where it's stored, but for table elements also about the internal structure of the data – See the VAST DataBase section below for details. We call the closely integrated physical and namespace layers the VAST Element Store. The VAST Element Store also provides element- or path-based services such as snapshots, clones, quotas, and quality of service policy enforcement.
- The protocol access layer implements all the protocols that users and applications use to access the VAST DataStore's contents. All of the protocols in the VAST AI OS are implemented as peers with multiprotocol access to Elements. File protocols, including NFS and SMB, directly access the same elements as S3 without the limitations imposed by the object-to-file or file-to-object translations common to other storage systems.

These three layers work together to provide a next-generation datastore designed to:

- Scale to exabytes while providing all-flash performance and hard drive economics
- Provide a single namespace that natively accommodates a wide range of data elements including files, objects, block volumes, and tables, eliminating the need for gateways and protocol translations
- Provide a full set of data services including zero-write snapshots and flexible replication
- Use the SCM provided by VAST as single source of truth for system state and metadata
- Provide strict ACID consistency for data and metadata updates
- Minimize flash wear via predictive placement and write shaping
- Provide the highest level of resilience (n+4) with less than 3% overhead, breaking the traditional tradeoffs between efficiency, performance, and cost

Remember, the layered structures we describe in this whitepaper don't have the strict boundaries the layered description may imply. The physical and Element Store layers share common metadata structures, and some system tasks or data services may straddle the logical boundaries between layers. This fuzziness is most apparent when we talk about table elements that are stored by the VAST DataStore but managed and accessed through the VAST DataBase.

Defining the DataStore

A New Approach to Metadata

While it's a bit of an oversimplification, you can think of the VAST Element Store as an Über (Nietzsche not rideshare) File System that presents files, tables, volumes, event triggers, functions, and objects all with equal aplomb. The VAST Element store is, of course, tightly coupled to the physical layer using byte-oriented metadata.

While it's not often discussed, modern storage features, including common features like thin provisioning, snapshots, clones, and data deduplication, are based on metadata. We've seen many times that data layout and metadata structure decisions made during initial system design can have a big impact on the system's feature set down the road.

The most important task of any data store is maintaining a consistent view of the data so that users and applications receive the data they expect when they issue a read. Eventual consistency may be good enough for some applications, but real data processing requires strictly consistent data. Before we dig into how the VAST DataStore organizes data, let’s look at how it maintains that consistency.

The most important task of any data store is maintaining a consistent view of the data so users and applications get the data they should when they issue a read. Eventual consistency may be good enough for some applications, but real data processing requires strictly consistent data. Before we dig into how the VAST DataStore organizes data, let’s look at how it maintains that consistency.

Inherently Persistent

All the VAST DataStore’s metadata, from basic file names to multiprotocol ACLs and locks, is maintained on shared media in VAST enclosures (DBoxes). This allows the mirrored, distributed metadata to serve as a single, consistent source of truth for the Element Store's state.

Eliminating the need for server-side caching also eliminates the overhead, and once again complexity, of keeping cached data coherent across multiple storage controllers. VAST systems store all their system state in shared enclosures that are globally accessible to each server over NVMe-oF. Because each VAST Server directly accesses a single source of system state, they don’t need to create any east-west traffic between nodes that shared-nothing systems require to update each other’s caches. This has two significant advantages:

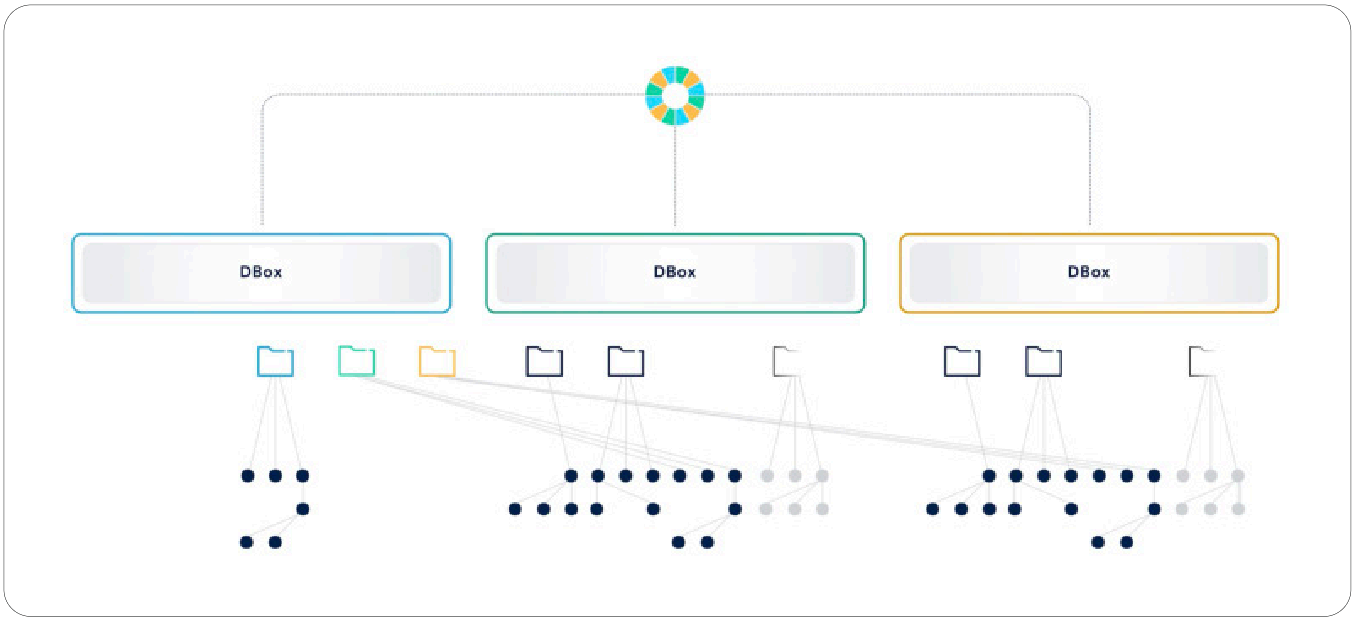
- Since the SCM and flash that the metadata is stored on is inherently persistent, there’s no data in DRAM or NVRAM cache. Destaging data from a volatile cache is often the largest contributing factor to enterprise storage data loss, as cache management and graceful destaging are simply difficult problems to solve. VAST Datastore avoids this issue altogether (even while providing support for exceptionally large and efficient write stripes). And because there’s no cache, there’s no need for power-failure protection, such as batteries (which need to be periodically replaced) or expensive ultracapacitors.

- Without the need to coordinate caches, it is much easier to scale I/O services across a scale-out namespace. The VAST Datastore architecture eliminates the need for east-west traffic in the synchronous write and read path, thus eliminating a number of operations that would otherwise need to be coordinated across the cluster (metadata updates, lock management, etc.). As server CPUs are added, the cluster benefits from a linearly scalable increase in performance – whereas other systems frequently experience the law of diminishing returns as global update operations need to be shared across an increasingly larger number of nodes.

Transactionally Consistent

Because we were designing the VAST DataStore to hold tables as well as files and objects, we knew we'd have to marry the transactional guarantees of an ACID database with the performance of a parallel file system and the scale of an object store. To achieve this goal, VAST Data needed to create a new distributed transaction model, with hybrid metadata structures that combine consistent hashing and tree- oriented metadata with a log influenced, write in free space data layout, new locking, and transaction management techniques.

At its core, the DataStore manages its metadata across a shared pool of Storage Class Memory with a V-Tree holding each element's (file, object, folder, Table, Volume, etc) metadata. CNodes locate the root of each element's V-Tree using consistent hashing of each element's unique handle. The hash space is divided into ranges, with each range assigned to two, or three, of the cluster's enclosures. Those DBoxes then hold the metadata roots for elements whose handles hash to values in ranges they are responsible for.



VAST Servers load the 1 GB consistent hash table into memory at boot. When a VAST Server wants to find data within a particular file or object, it calculates the hash of the element's unique handle and performs a high-speed memory lookup to find which enclosure(s) hold that element's metadata by hash range. The VAST Server can then read the element's V-Tree from one of the DBoxes responsible for that portion of the hash space.

By limiting the use of consistent hashing to only the root of each data element, the dataset size per each hash table is very small – the system scales while minimizing the amount of hash data that has to be recalculated when VAST clusters expand. When a new enclosure is added to a cluster, it assumes ownership of its share of the consistent hash space and only the root of the V-Trees for Elements for the hash-space segments are migrated to the new enclosure.

V-Trees for Fast Access

The VAST DataStore maintains its persistent metadata in V-Trees. VAST's V-Trees are a variation of a B-tree data structure, specifically designed to be stored in shared, persistent memory. Because VAST Servers are stateless, a new metadata structure was needed to enable them to quickly traverse the system's metadata stored on remote SCM devices. To achieve this, VAST designed a tree structure for extremely wide fan-out: each node in a V-Tree can have 100s of child elements – thus limiting the depth of an element search and the number of round-trip requests over the network to no more than seven hops.

VAST CNodes do not, themselves, maintain any local state – thereby making it easy to scale services and fail around any CNode outage. When a VAST CNode joins a cluster, it executes a consistent hashing function to locate the root of various metadata trees. As Server resources are added, the cluster leader rebalances responsibility for shared functions. Should a Server go offline, other Servers easily adopt its VIPs and the clients will connect to the new servers within standard timeout ranges upon retry.

While it isn't organized in tables, rows, and columns, the DataStore's V-Tree architecture enables the metadata store to act in many ways like a database – allowing VAST CNodes to perform queries in parallel and locate, for example, an object in an S3 bucket, or the same data as a file, as it existed when the 12.01 AM January 1, 2024 snapshot was taken.

Just as CPUs add a linearly scalable unit of capacity, DataStore metadata is distributed across all the cluster's Storage Class Memory, which enables the namespace to scale as well as enabling performance to scale as more and the VAST Cluster scales.**Database**

Database Semantics

VAST DataStore's namespace metadata can be thought of in many ways as a database or key-value store against which the system makes queries to locate pieces of data by file or object name, snapshot time, and other metadata attributes. That database metaphor also extends to how the VAST DataStore uses transactional semantics to ensure that the VAST DataStore, like a relational database, is fully ACID (Atomic, Consistent, Isolated, Durable).

Unlike eventually-consistent systems, like most object storage systems, the VAST DataStore provides a consistent namespace across all the VAST CNodes in a cluster to all the users. Changes made by a user on one node are immediately reflected and available to all other users.

To remain consistent, the VAST DataStore ensures that each transaction is atomic. To achieve this, each storage transaction is either applied to the metadata (and all of its mirrors) in its entirety, or not applied to the metadata at all (even if a single transaction updates many metadata objects). With atomic write consistency, classic file system check tools (such as the dreaded fsck) are no longer needed, and systems can be instantaneously functional upon power cycle events.

Transaction Tokens

VAST V-Tree update transactions are managed using transaction tokens. When a VAST CNode initiates a transaction, it creates a transaction token metadata object and increments the transaction token counter. The transaction token contains a globally unique identifier that is accessible from all VAST CNodes and is used to track updates across multiple metadata objects. This token is infused with the identity of the VAST CNode that owns the transaction, as well as the transaction's state (ongoing, canceled, committed), to enable the system to avoid complications from parallel operations.

As the VAST CNode writes changes to an Element's V-Tree, it creates new metadata objects with the transaction token attached. When another VAST CNode accesses that element's metadata, it checks the transaction token state and then takes the appropriate action using the latest data for committed transactions. If a VAST CNode (requesting Server) wants to update a piece of metadata that is already part of an in-flight transaction, it will poll the owning Server to ensure it is still operational and will subsequently stand down until the owning server completes the in-flight transaction. If, however, a requesting CNode finds in-flight data that is owned by a non-responsive CNode, the requesting CNode will access the consistent state of the namespace using the previous metadata and can also cancel the transaction and remove the metadata updates with that token.

Bottom-Up Updates

One key to designing a high-performance transactional namespace is to minimize the exposure to failure during any given transaction by minimizing the number of steps in a transaction that could cause the namespace to be inconsistent if the transaction didn't complete. The VAST DataStore minimizes this exposure by making its updates from the bottom of the V-Tree and working up. When a client overwrites data in an existing file, the system performs the following steps (simplified for the purposes of this document):

1. Data is written to free space on mirrored Storage Class Memory SSDs (via indirection)

2. Metadata objects and attributes are created (BlockID, life expectancy, checksum)

3. The file’s metadata is changed to link to any new metadata objects

4. This new write is then, and only then, acknowledged to the client

For operations that fail before step #4 is successfully completed, the system will force clients to retry a write and any old/stale/disconnected metadata that remains from the previous attempt will be handled via a background scrubbing process.

If, by comparison, the system updated its metadata from the top down, that is first add a new extent to the file descriptor, followed by the extent, and block metadata; a failure in the middle of the transaction would result in corrupt data from pointers to nothing. Such a system would have to treat the entire change as one complex transaction, with the file object locked the whole time. Updating from the bottom, the file only has to be locked when the new metadata is linked in (3 writes vs. 20). Shorter locks reduce contention and therefore improve performance.

Element Locking

While each read operation within the metadata store is lockless, the VAST cluster employs internal write locks to ensure parallel write While each read operation within the metadata store is lockless, the VAST cluster employs internal write locks to ensure parallel write consistency across the namespace. Element Locks differ from Transaction Tokens in that Transaction Tokens ensure consistency during Server failure; while Element Locks ensure consistency when multiple writers attempt to operate on a common range of data within the DataStore.

Metadata locks, like Transaction Tokens, are signed with the ID of the VAST CNode that reserved the lock. When a VAST CNode detects a metadata object is locked, it contacts the server associated with the lock, preventing zombie locks without relying on a central lock manager. If the owner server is unresponsive, the requesting server will ask another non-involved server to also query the owning server. This approach helps ensure that the requesting server does not get a false positive and prematurely dismiss the owning server from the VAST Cluster.

To ensure that write operations are fast, the VAST Cluster holds read-only copies of Element Locks in the DRAM of the VAST DBox where the relevant Element lives. In order to quickly verify an Element’s lock state, a VAST CNode performs an atomic RDMA operation to the memory of an Enclosure’s Fabric Module to verify and update locks.

While the above Locking semantics apply at the storage layer, the VAST Cluster also provides facilities for byte-granular file system locking, as discussed later in this document (see: VAST Datastore: Protocols).

The Physical Chunk Management Layer

The Physical or chunk management layer of the VAST DataStore performs many of the same functions that a SAN array or logical volume manager (LVM) does in conventional architectures, protecting data against device failures and managing the storage devices.

While the physical layer, like RAID, is responsible for persisting and maintaining data written to the VAST DataStore, the methods it uses, like locally decodable erasure codes, have come a long way from the days of RAID. The other big difference is that the VAST DataStore uses one integrated set of metadata to manage the namespace of elements (files/objects/Tables/etc) and the datachunks so it can make smarter decisions when placing data.

Previous storage and file systems were designed to minimize random I/O, because hard drives could only perform a few IOPS each by caching metadata in controller DRAM. This not only created complexity to keep this distributed cache coherent, but the limited amount of DRAM forced file system architects to simplify their metadata in several ways.

- They allocated space and reduced and protected data in fixed-size allocation blocks, forcing tradeoffs between the advantages of small or large blocks for different purposes and limiting the effectiveness of their data reduction if any
- They limited scale with intermediate abstractions like RAID sets, volumes, and file systems, which created fragmented namespaces
- They replicated or erasure-coded file by file instead of globally, complicating the management of small files and data reduction

The VAST DataStore was designed for the DASE architecture, and since DASE clusters have no spinning disks, there's no need to optimize I/O for sequential access. Instead, the VAST DataStore is optimized for low-cost hyperscale flash by maximizing efficiency and minimizing write amplification, which consumes flash endurance in more conventional systems.

The VAST DataStore introduces several innovative processes and data structures while leveraging the best ideas from the last half century of storage. Basic principles of the VAST DataStore's design are:

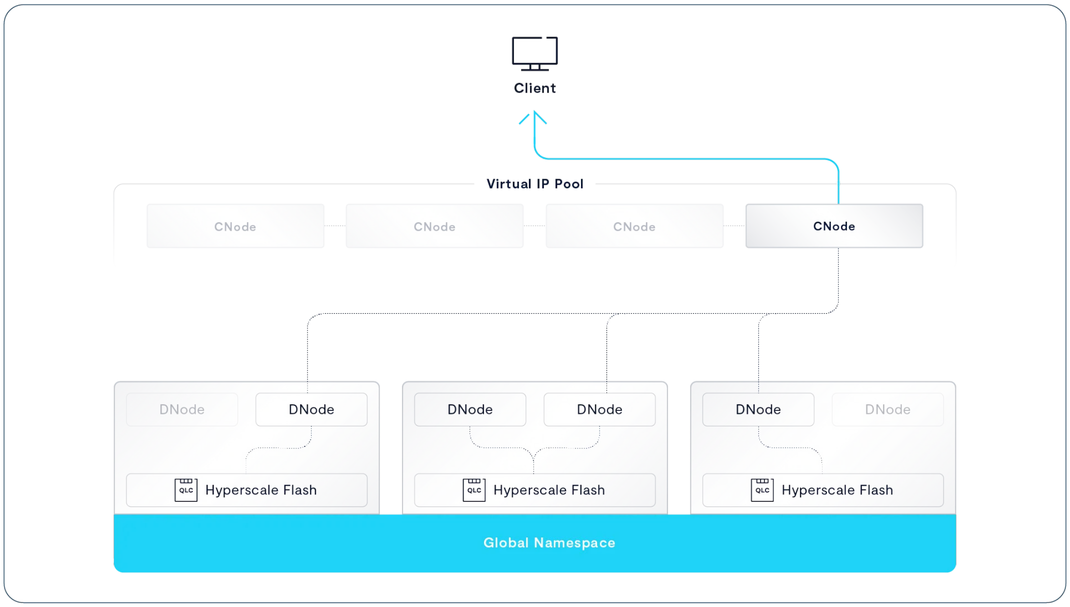
- **SCM Write Buffer:** Incoming data is written, by the receiving CNode, to write buffers on two or three SCM SSDs. Writes are acknowledged only once data is safely written to SSD.
- **Asynchronous Migration:** Data is migrated from the write buffer to hyperscale flash asynchronously. This allows time for more effective data reduction, among other functions.
- **Flash Management:** The VAST DataStore is designed to minimize flash wear through multiple techniques. Foresight minimizes the amount of data moved in garbage collection while the system writes and deletes data in blocks that minimize flash wear.
- **A Write-in-free-space data layout:** The VAST DataStore uses a write-in-free-space data layout. All new data is written to full erasure code stripes in free space on the hyperscale SSDs.
- **Byte Granularity:** There are no fixed-size allocation blocks in the VAST DataStore. The physical chunk layer manages write buffer chunks, which are the size of the write from a few bytes to a megabyte, and reduced data chunks on hyperscale flash which average 32 KB in size (See Adaptive Chunking). Pointers to both types of data chunk point to a byte range (SSD, LBA, Offset, Length), allowing chunks to be stored without any padding to allocation block, or even LBA boundaries.
- **Breakthrough Similarity Data Reduction:** VAST clusters reduce data as it is migrated from SCM to hyperscale flash using a combination of techniques guaranteed to reduce data better than any other storage solution.
- **Highly Effcient Erasure Codes:** VAST's Locally Decodable Codes provide protection from as many as four simultaneous SSD failures with as little as 2.7% overhead.

The physical layer breaks data down into variable-sized data chunks, reduces those chunks, and protects them with Locally Decodable Erasure Codes. Before we look in detail at how the VAST DataStore performs those tasks let's take a quick look at how data flows through the VAST DataStore.

Data Flows in the VAST DataStore

Before we look in detail how the VAST DataStore follows all of these principles, let's take a quick look at how the VAST DataStore handles basic reads and writes.

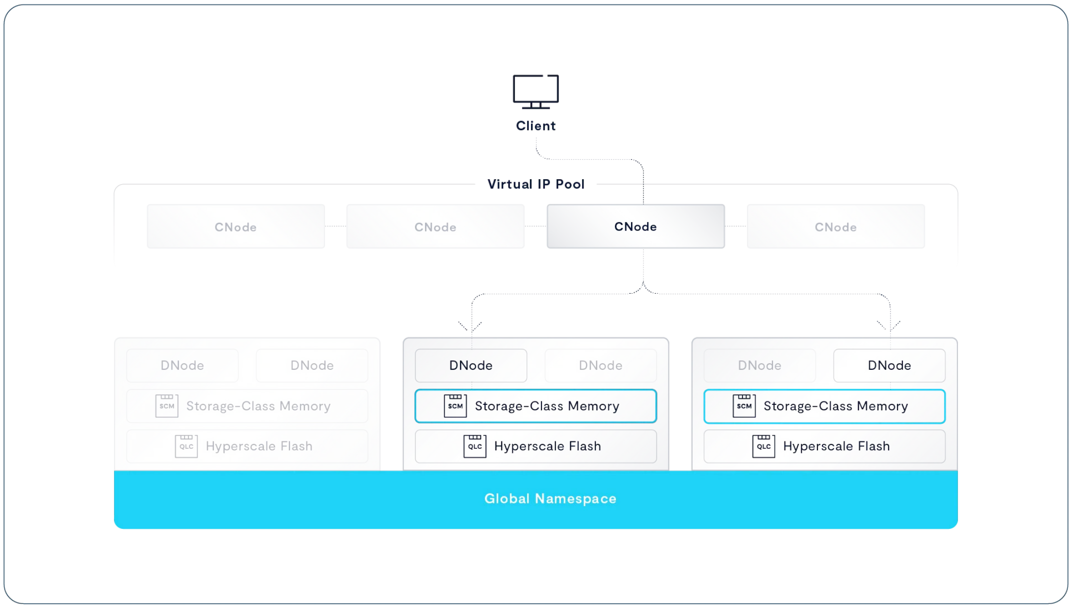
Read



When a CNode receives a read request, that CNode locates the root of the metadata V-Tree for the Element being read using the cluster's consistent hash table. It then follows the V-Tree's pointers in SCM until it finds pointers to the requested content.

The CNode then retrieves the content chunks directly from the hyperscale SSDs, assembles the requested data, and sends it to the client.

Write to SCM

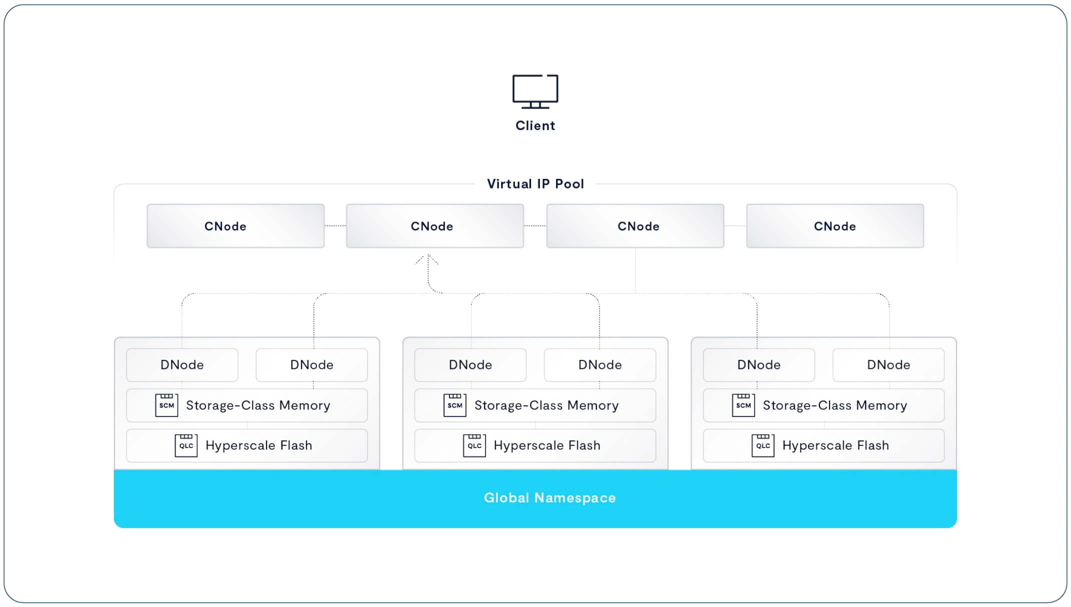


When a client writes data to an Element in the VAST DataStore, the CNode receiving that request writes the data to protected write buffers across multipleSCM SSDs. If encryption at rest is enabled, the data is encrypted by the CNode before being written

Data is written to the write buffer in chunks that are generally the same size as the write I/O, though some large writes may be divided into multiple chunks. Once the data is written to multiple SSDs, the CNode updates the metadata for the write, which is itself mirrored across two or three SCM SSDs and sends an acknowledgement to the client.

The system attempts to maximize the distance between the multiple copies of any data chunk or metadata block. Clusters with more than one DBox will write to SCM SSDs in different DBoxes. Clusters with only one DBox will write to SCM SSDs connected to the NVMe fabric through different DNodes.

Migrate to Flash



When the data in a cluster's write buffer reaches a high water mark, the system starts the process of migrating data from the write buffer to the main repository on hyperscale flash. The migration process is distributed across multiple CNodes in parallel.

Each CNode reads data from the write buffer and reduces it, breaking it into smaller, variably sized chunks that average between 16 and 64 KB. The CNode then writes the reduced data into very wide locally decodable erasure-code stripes to the hyperscale SSDs.

This asynchronous migration provides inline data reduction while also allowing the system time to reduce data more effectively. Since the migration process runs after writes are acknowledged to the client, data reduction time isn't reflected in write latency. As long as CNodes in parallel can drain the write buffer faster than new data fills it, the time it takes to process any given data chunk is irrelevant.

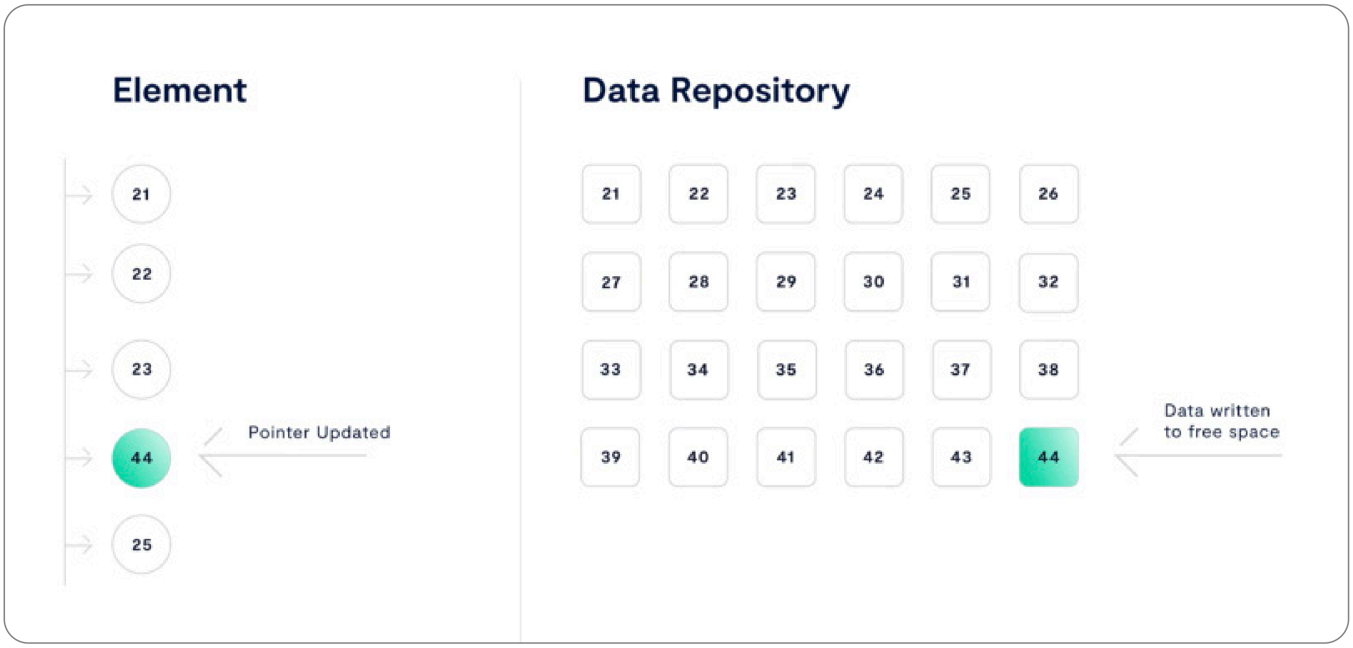
Once an erasure code stripe has been written to the hyperscale SSDs, the CNode updates the system metadata to point to chunks in the new stripe and marks the write buffers it has emptied as free for new data.

Write in Free Space Indirection

Conventional storage systems maintain static relationships between logical locations, like the 1 millionth to 4 millionth bytes of Little_ Shop_of_ Horrors.MP4, and the physical storage locations that hold that data. As such, when an application writes to that location, the system physically overwrites the old data with the new data in-place. Back in the day of hard disk arrays, static mappings like this had the advantage of keeping sequential data physically adjacent, reducing the number of hard drive head motions.

DASE systems don't have spinning disks; instead, they use low-cost hyperscale flash SSDs to provide their capacity. It therefore doesn't make any sense to optimize the data layout on a VAST cluster to maximize sequential I/O. Instead, the VAST DataStore is optimized for the efficient use of both the capacity and the endurance of those SSDs.

The VAST DataStore uses a write-in-free-space data layout that performs all writes via indirection. New data is written to hyperscale flash in full-erasure-code stripes as it's migrated from the write buffer. When data is logically overwritten, the metadata for the affected Element (file/object/table) is updated to point to the new data chunk.



The write-in-free-space method has several advantages:

- Eliminates the read-modify-write overhead of overwrites in place
- Provides pointer mechanisms for low-overhead snapshots, clones, deduplication, replication, and other services
- Empowers the system to write and delete from SSDs in patterns that minimize flash wear

Challenges with Commodity Flash

The low cost of commodity, hyperscale-grade SSDs is an important part of how a VAST DataStore cluster can achieve archive economics. There are a few barriers that present challenges for legacy systems to use this new type of flash technology:

1. **Write Performance:** As the flash foundries squeeze more bits per flash cell, the resulting SSDs take longer to write data to the flash, as they carefully tune the charge in each cell to one of 16 (QLC) or 32 (PLC) voltage levels.

2. **Endurance:** Write endurance is the single biggest challenge when considering the use of commodity flash. Some Hyperscale flash drives can only be overwritten a few hundred times before they are worn out.

As we've already seen, the VAST DataStore uses SCM SSDs as a write buffer, and since VAST systems acknowledge writes once the data is safe on SCM SSDs, high write latency from the hyperscale SSDs doesn't affect system performance. That same SSD write buffer also helps VAST systems manage endurance across the estate of hyperscale SSDs in the cluster.

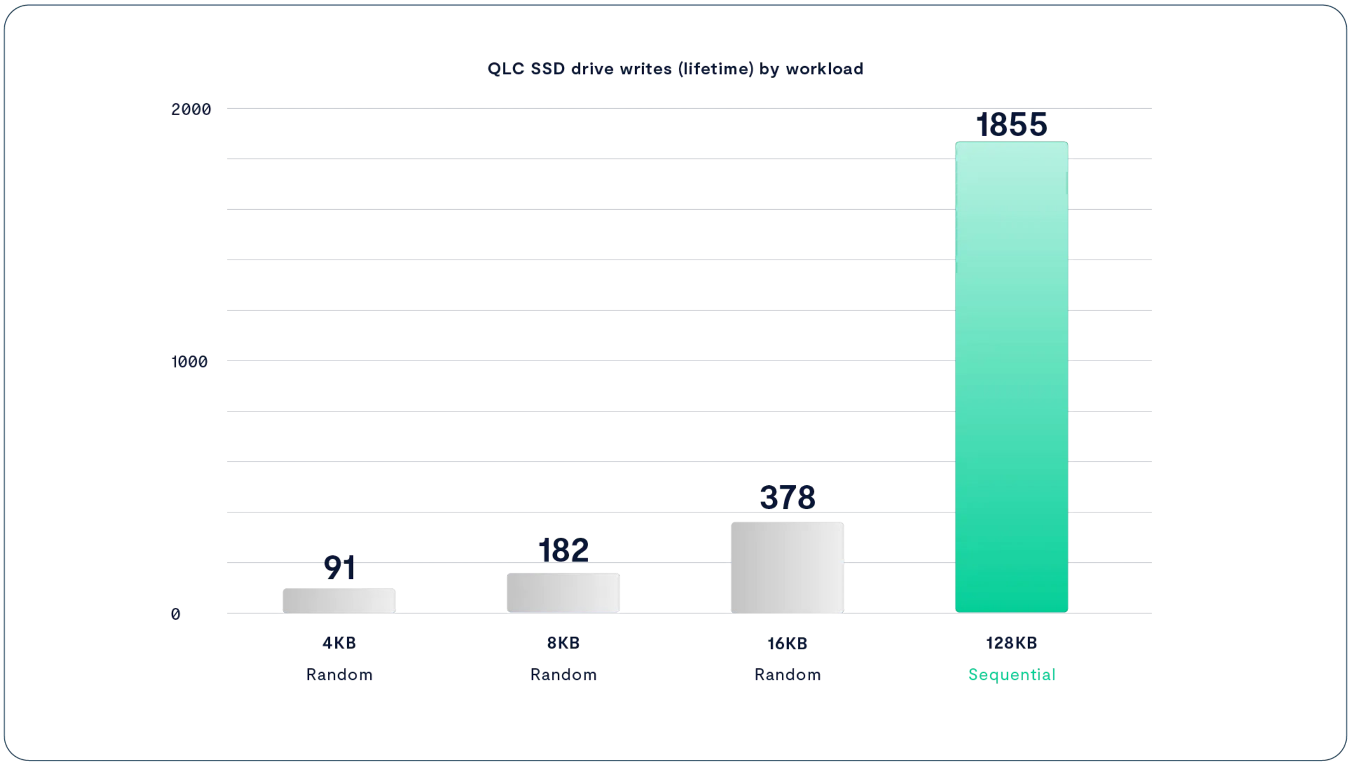
Endurance Is Write Dependent

SSD vendors specify an SSD's endurance in DWPD (Drive Writes Per Day for 5 years) or TBW (Terabytes Written) based on a JEDEC standard workload that includes 77% writes of 4 KB or smaller. This makes sense because many legacy storage products were designed to write in 4KB disk blocks and 4KB is how people tend to think about IOPS workloads – which is what people have historically been buying flash for in the first place.

The VAST DataStore was designed to run on the lowest-cost hyperscale SSDs, unlike more expensive enterprise SSDs. Hyperscale SSDs write data directly to flash, without an internal DRAM buffer where the SSD controller can accumulate data and write it to flash in larger blocks.

That's significant because small writes to flash (<128 KB) consume significantly more of the SSD's total endurance than the same data written as large I/Os.

As seen in the chart below, the endurance of the Intel P4326 SSD varies significantly based on the size and distribution of writes:



QLC SSDs provide 20 times more endurance when they're written to in large sequential stripes as compared to being written to in the 4KB random writes, as is often common in many enterprise storage systems.

The reason hyperscale SSDs deliver higher endurance for large writes than small random writes is because those large writes better align with the internal structure of the flash in the SSD. Flash memory stores data in the form of a charge (some number of electrons) trapped within a memory cell.

The chip can then measure the charge in the cell, and based on the voltage of the charge in the cell, decide what the value of the cell is. That's easy with SLC flash: there's either a charge (0) or not (1). With QLC flash the chip has to differentiate between 16 voltage levels to deliver 4 bits/ cell.

The cells in a flash chip are organized in a complex hierarchy of structures, luckily only two of which are significant for managing flash:

- The Page – The page is the smallest structure that can be written. Once any data is written to a page that data cannot be modified until the page is erased. Hyperscale SSDs today have pages 64-256 KB in size
- The Erase Block – An erase block, which contains many (256-64K) pages, is the smallest unit the flash chip can erase. Erase blocks on today's high-density flash are several MB to several GB in size

When the SSD controller receives new data, it writes that data to an empty page and updates the metadata of its internal log-structured data layout. Each write consumes one program/erase cycle of every cell in the page, so 4 KB writes consume a whole 128 KB page's endurance.

When the SSD starts running out of space, it has to garbage collect. It compacts valid data from several erase blocks into new pages, allowing those erase blocks to be erased, thereby creating more empty pages for the next batch of incoming data.

The problem is that erasing flash requires very high voltages, at least within a chip, and causes quantum-mechanical dynamic wear on the flash cell insulation. Eventually this causes electrons to leak out, and the flash cells can't reliably hold data.

VAST Datastore Data Structures

Optimized for Hyperscale Flash

The VAST Datastore is designed to minimize **write amplification**; that is, the number of times any data is moved from one place to another, usually via garbage collection, either in the write-in-free-space DataStore or internally to the SSDs.

The first step is a cluster-wide flash translation layer. The VAST DataStore manages the flash across all its hyperscale SSDs much like the SSD controller in an SSD manages its flash chips. In both cases the software takes advantage of special knowledge of the hardware's abilities and limitations to manage those resources.

With Large Data Stripes, Drives Never Need to Garbage Collect

Legacy indirection-based file systems use a single logical block size through all their data services. Using fixed data sizes of 4 KB or 32 KB blocks for data reduction, data protection, and data storage keeps things simple for traditional storage systems, especially when sizing file system metadata stores because the systems can always point to a consistently-sized data block.

The problem with this approach is that it forces the storage architect to compromise between the low-latency and fine-grained deduplication of small blocks vs. the greater compressibility and lower flash wear of larger blocks. Rather than make that compromise, the VAST Element Store manages data at each stage of its journey:

- Data is written to the SCM write buffer in the size of the write I/O to minimize latency
- It is adaptively chunked into variable size chunks (average size 16-64KB) and reduced
- Those data chunks are tightly packed into erasure code substripes each a multiple of the SSD's page size so the SSD controller can write full pages
- Each erasure code stripe is made up of many substripes, creating an erasure code stripe multiple times the size of the SSD's erase block

The VAST DataStore writes to SSDs in 1 MB I/Os, a significant multiple of the underlying flash's 64 KB-128 KB page size, thereby allowing the SSD controller to parallelize the I/O across the flash chips within an SSD. It also completely fills flash pages, preventing the write amplification that is caused by partially written pages.

The Element Store manages data in deep data strips that layer 1 MB I/Os into a larger 1 GB strip of data on each SSD. Just as writing 1 MB I/ Os to the SSDs aligns the write to a number of full flash pages, managing data in 1 GB strips aligns each erasure code stripe with a set of erase blocks inside the SSD.

When the VAST DataStore performs garbage collection, it erases the entire 1 GB erasure-code strip on each SSD, causing the SSD controller to erase the contents of several erase blocks. This leaves no data for the SSD controller to relocate, thereby preventing the SSD from performing internal garbage collection.

The 1 GB strip depth and 1 MB sub-strip depth are not fixed values in the VAST DataStore architecture; they were determined empirically by measuring the wear caused by writing and erasing multiple patterns on today's generation of capacity SSDs. As new SSDs are qualified and as PLC flash enters the market, the VAST Cluster write layout can adjust to even the larger writes and erases that future generations of flash will require to minimize wear.

The underlying architecture components make all of this intelligent placement possible:

- A large, distributed buffer, built from SCM, provides the system with the time needed to form a collection of erasure-encoded write stripes as large as 146+4 – giving the application snappy response times without the need to hold up writes before flushing data down to hyperscale flash
- NVMe-oF and NVMe drives provide the shared-everything cluster foundation that makes it possible for multiple writers to gracefully fill erase blocks with 1 MB strips

VAST Foresight

Retention-Aware Data Protection Stripes

The one big downside to a write-in-free-space data layout is that eventually the system runs out of free space, and the system has to garbage collect, relocating the still-valid data chunks mixed in with all the deleted and overwritten data ready for the proverbial bit bucket. Foresight minimizes the write amplification created by this garbage collection by writing data to erasure code stripes based on the data's life expectancy.

Before Foresight, storage systems generally wrote data to their drives in the order the data was written to the system. If one set of hosts creates the final render of "Sherlock Holmes vs Dracula" while another set of hosts writes thousands of temp files for another project, the data from those two streams is going to be interleaved as it's written.

If that storage system used a log-structure or other write-in-free-space layout it would have to garbage collect and compact the render data some time after the temp files are deleted. Over three or four years data that hasn't been logically moved may still be relocated by garbage collection several times, consuming valuable flash wear each time.

Rather than storing data in the order it was written, or with logically adjacent data physically adjacent to optimize sequential reads for hard disks, Foresight organizes data into erasure code stripes based on its life expectancy. The metadata describing each chunk in the VAST DataStore's physical layer has a life expectancy value.

When the system builds erasure code stripes it assembles data chunks with similar longevity. The temporary files and final render data will get written to different erasure code stripes. Once the temporary files are deleted those erasure stripes will be mostly empty, leaving little data to move during garbage collection.

VAST DataStore only garbage collects when free space falls below a low water mark, and garbage collects from the erasure code stripes with the most recoverable space which, combined with Foresight, allows the garbage collection process to create the most free space with the least data motion.

The data that is garbage collected has its life expectancy increased. Over time, this will cause the very long-lived data on the system to accumulate into erasure code stripes for "immortal" data stripes where they can live permanently eliminating the repeated relocation of garbage collection on legacy systems.

Endurance Is Amortized

Wear-Leveling Across Many-Petabyte Storage

The VAST DataStore also extends flash endurance by treating the petabytes of flash in a cluster as a single pool of erase blocks that can be globally managed by any VAST CNode. The VAST Cluster performs wear-leveling across that pool to allow a VAST DataStore system to amortize even very-high-churn applications across petabytes of flash in the Cluster. Because of its scale-out design, the VAST Cluster only needs to work toward the weighted overwrite average of the applications in an environment, where (for example) a database that continually writes 4 KB updates will only overwrite a fraction of even the smallest VAST cluster in a day.

The large scale of VAST clusters is also key to the system's being able to perform garbage collection using the 150 GB stripes that minimize flash wear and data protection overhead. Unlike some other write-in-free-space storage systems, the VAST Element store continues to provide full performance up to 90% full.

A Breakthrough Approach to Data Reduction

The long history of data reduction has primarily focused on refining compression and deduplication: two complementary techniques that use computing power to reduce the size of data stored. Compression, the older technique, reduces repeated copies of small bit patterns over a limited range. Deduplication techniques eliminate repeated patterns in much larger blocks over correspondingly larger sets of data.

The VAST DataStore uses these two techniques and also adds a new technique called similarity reduction. This reduces the amount of storage needed to store data chunks that are similar, but not identical, to existing chunks. We are so confident the VAST DataStore will reduce any unencrypted dataset better than any commercially available storage solution that we guarantee it.

This section provides a high-level view of VAST's data reduction methods, including similarity reduction. For a deeper dive see "All Data Reduction Is Not Equal."

Beyond Deduplication and Compression

Traditional compression operates over the limited scope of a block of data or a file of a few KB to a few MB. The compressor identifies small sets of repeating data patterns, such as the nulls padding out database fields or frequently used words. It replaces those repeating patterns with smaller symbols and creates a dictionary defining what data that symbol represents. To decompress the data when a read is issued, the data-to-symbol dictionary is used to reverse the process and replace the symbols with the corresponding, larger data strings.

Many file types, especially for media, include compression as part of the file type definition. Since these compression techniques can be specific to the type of media being stored, like H.264 MPEG storing frames as the changed pixels since the last frame, they can be more effective than more generic storage system compression algorithms. As a result, some unstructured data repositories may not see any significant compression from the storage system.

Data deduplication, by comparison, identifies much larger blocks of data that repeat globally across a storage namespace. Rather than storing multiple copies of the data, a deduplication system uses pointers to direct applications to a single physical instance of a deduplication block. Most deduplication systems break data into chunks somewhere between 4KB and 1MB in size. It then hashes the chunks and looks for duplicates. It saves duplicate chunks 2–n as pointers to the original chunk.

Because deduplicated data stores are based on pointers, deduplicating systems have to rehydrate data for sequential reads, making random read I/Os on the backend to reassemble data that's logically sequential but stored randomly across the deduplicated repository. This makes deduplicating storage systems that use hard disks on the back end much slower for reads as the hard drives thrash their positioners. SSDs perform random and sequential I/O at equal speeds, eliminating this rehydration tax.

The block hashing employed by deduplication is much more sensitive to very small differences in data and therefore can force a deduplication system to store a new, full block of data even when there is just a single byte of difference between deduplication blocks.

This has the effect of doing a lot of work to hash the blocks without substantial dataset size reduction.

To minimize this sensitivity to entropy in data, deduplication systems can deduplicate with smaller block sizes. Small deduplication blocks create more metadata, and since most deduplicating systems have to hold the table of hashes already stored in DRAM, deduping on smaller blocks requires more memory and more CPU to manage the memory, and may not pay off in lower system cost.

Similarity Reduction to the Rescue

The combination of hyperscale flash, ten years of system longevity, and breakthrough erasure code efficiency creates an economically compelling proposition for organizations that are looking to reduce the cost of all-flash or hybrid (flash+HDD) infrastructure. The VAST approach to global data reduction, known as similarity-based reduction, further redefines the economic equation of flash. It is now possible to build a system that provides a total-effective-capacity acquisition cost that rivals or betters the economics of an all-HDD infrastructure. The objective is simple: to combine the global benefits of deduplication-based approaches to data reduction with the byte-granular approach to pattern matching, which until now has only otherwise been found in localized file or block compression.

A flash system can cut up and correlate data for purposes of data reduction that results in a block size on disk that is extremely small, turning every workload into an IOPS workload. When one has a system that has, for all purposes, unlimited IOPS, this becomes an ideal workload. This same workload on HDDs would result in tremendous fragmentation and HDDs simply can't deliver enough IOPS to support the many random- access reads in fine-grained data reduction. It may seem counter-intuitive, but the only way to build a system that can beat the economics of an HDD-based storage system is using similarity-based reduction with commodity flash.

Similarity Reduction

When data is written to the VAST DataStore, it is first written to the storage-class memory (SCM) write buffer and acknowledged (ACKed). The VAST DataStore then performs data reduction as part of the process of migrating data from the SCM write buffer to the hyperscale flash capacity pool.

Since writes are acknowledged to applications when written to the SCM write buffer, VAST systems have plenty of time to perform more meticulous data reduction. As long as the system drains the write buffer as fast as new data is written to the system, the time it takes to reduce the data has no impact on system performance or write latency.

Adaptive Chunking

The VAST systems break data into variable-size chunks (average size between 16 and 64 KB) using a rolling hash function to maximize storage efficiency. It then hashes the chunks with several hash functions, including a strong hash to give the chunk a unique identity. As with conventional deduplication, when multiple chunks of identical data are written, the system stores a single copy of the data and uses metadata pointers for the rest.

The system also hashes the chunk with a series of similarity hash functions. The strong hashes used for deduplication are designed to be collision-resistant; a small change in the input data causes a large change in the output so two different blocks of data don't generate the same hash. Similarity hashes are, by comparison, weak hashes; they're designed to generate the same hash value for any inputs that are within a short cryptographic distance of each other. In plain English that means these similarity hash functions will generate the same hash for any chunks that require a small number of bits to be flipped to turn one chunk into another.

If it will only take a few bit flips to turn chunk A into Chunk B that means there must be multiple, significant strings of byte that are the same across the two chunks. That in turn means that the two chunks will compress with the same compression dictionary to reduce those strings to symbols.

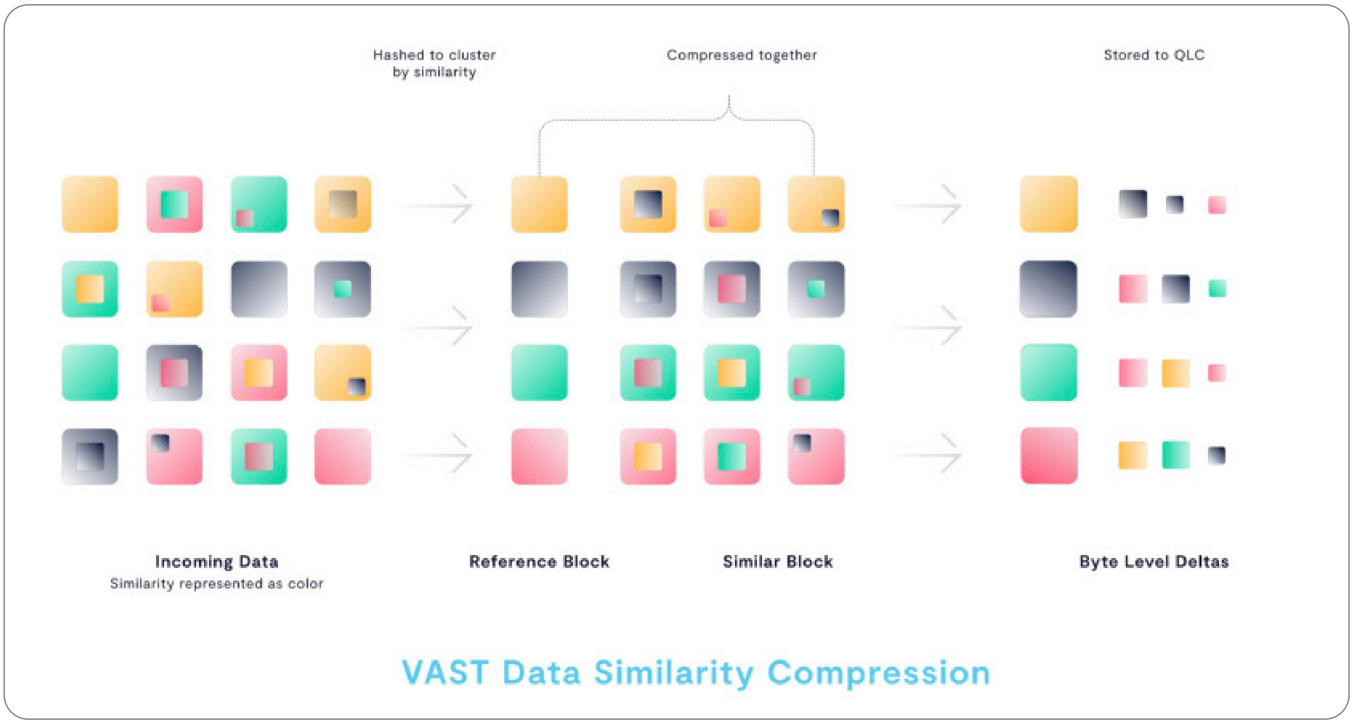
When a chunk of new data generates a unique similarity hash, that chunk is compressed with ZSTD, a new compression algorithm from Facebook optimized for decompression performance, and added to the erasure-code stripe the current CNode is assembling. When a chunk of new data generates a similarity hash that's the same as the hash generated by a previous data chunk, the system recalls the original reference chunk and compresses the new data with the compression dictionary from the reference chunk. The system then stores the resulting compressed data as a delta block without the overhead of storing the dictionary a second time.

A Single Reduction Realm

As we noted earlier, conventional deduplication systems have to store their hash tables in DRAM to process data fast enough. This limits the amount of data they can deduplicate to the amount of memory a single controller can hold, about 1 PB for the largest purpose-built backup appliances. Once a dataset grows to the point where it has to be spread across multiple appliances, each appliance forms an independent deduplication realm and any data chunks that are written to multiple appliances will be stored multiple times. Some systems limit the effectiveness of their deduplication by deduplicating within volumes, file systems, or other intermediate abstractions. This creates even more deduplication realms, and copies of really common data.

Scale-out systems face the even more challenging problem of maintaining a hash table in memory across a cluster without generating so much east-west traffic between the nodes that deduplication is no longer practical. The result is that deduplication is less common on scale-out systems and requires compromises such as having a local hash table in each node, effectively turning each node into a separate deduplication realm.

VAST systems store their hash tables and other DataStore metadata on SCM SSDs in DBoxes. Because those SCM SSDs are shared across all the CNodes, the entire cluster constitutes a single data reduction realm for deduplication and similarity, and since there's SCM in every DBox that gets added to the cluster, that single reduction realm can grow to exabytes.


Hashed to cluster by similarity
Compressed together
Stored to QLC
Incoming Data
Similarity represented as color
Reference Block
Similar Block
Byte Level Deltas
VAST Data Similarity Compression

Similarity Reduction in Practice

The efficiency that can be gained from similarity-based data reduction is of course data-dependent. While encrypted data will see almost no benefit from this approach (or any other form of data reduction), other applications will often see significant gains.

Reduction gains are, of course, relative—and where a VAST cluster may be 2x more efficient than a legacy deduplication appliance for backup data (at 15:1 reduction), a reduction of 2:1 may be as valuable in an unstructured data environment where legacy file storage systems have never been able to demonstrate any reduction.

Some examples of VAST's similarity reduction, derived from customer testing, include:

- AI/Machine learning training data: 3:1
- Enterprise backup files: up to 20:1
- Log stores (indexes and compressed data): 4:1
- Multi-tenant HPC environments: 2:1
- Pre-compressed (GZIP) financial services market data: 1.5:1
- Pre-compressed video: 1.1:1
- Seismic data: 3:1
- VFX and animation data: 3:1
- Weather data: 2.5:1

A deeper dive of VAST's similarity-base data reduction is found in "All Data Reduction Is Not Equal."

A Breakthrough Approach to Data Protection

Protecting user data is the primary goal of any enterprise computing system overhead down from 66% (using an acute-case example, triplication overhead) to as little as 2% while also increasing the resilience of a cluster beyond what classic triplication and erasure codes provide today. The result is a new class of error-correction codes that deliver higher resiliency (millions of hours mean time to data loss and over 11 9s of data durability) and lower overhead (typically under 3%).

This section describes how VAST systems protect data at a high level. For a deeper dive see "Ensuring Storage Reliability at Scale."

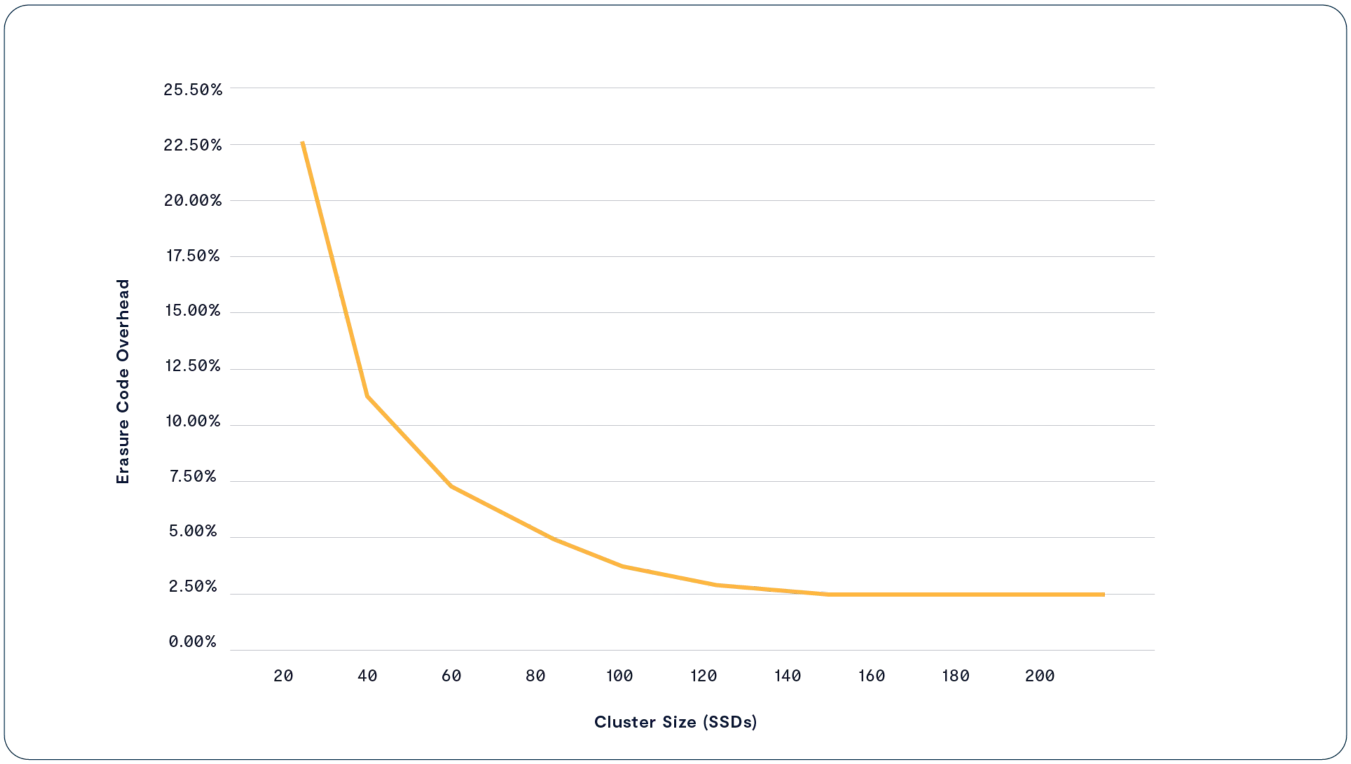
Wide Stripes for Storage Efficiency

The key to efficiency in data protection is to write wide erasure code stripes. It's obvious that an erasure code that writes 8 data strips and 2 parity strips per stripe, abbreviated 8D+2P, has more overhead (20%) than a 14D+2P stripe would (12.5%).

Since a single failure in a shared-nothing node takes all the drives in that node offline, shared-nothing systems must treat the node as a unit of failure and stripe across nodes as well as drives. VAST clusters running on highly available DBoxes, take advantage of that high availability by striping across all the hyperscale SSDs in a cluster, safe in the knowledge that no single failure will take more than one SSD offline.

Because there are many more SSDs than enclosures, DBox clusters can write erasure code stripes much wider, up to 150 total strips per stripe. This achieves much lower overhead than a shared-nothing architecture, where a node is a unit of failure. For example:

- A small collection of VAST DBoxes can provide redundant access to a collection of drives striped across the DBoxes in a 146+4 SSD write stripe
- If a shared-nothing cluster were to implement this same stripe width, a system would require 75 to 150 storage servers to achieve this same level of low overhead



Quad Parity for Higher Resilience

Wide write stripes are the means to maximize storage efficiency, but wide stripes also increase the probability that multiple devices within a stripe could fail. While flash SSDs are very reliable devices, especially when compared to HDDs, it's simply more statistically likely that two of the 148 SSDs in a 146+2 stripes will fail, than 2 stripes out of the 12 SSDs in a stripe coded 10+2.

To support wide stripes and the resilience to provide the 11 9s of data durability that exabyte data repositories demand, VAST systems always write erasure-code stripes with four parity strips. This allows a VAST system to rebuild its data protection and continue to serve data to users, with as many as four simultaneous SSD failures.

While adding additional parity strips to a write stripe helps increase the stripe's resilience, this is not enough to counterbalance the higher probability of failure that results from large write striping. The other aspect of ensuring high resilience in a wide-write-striping cluster is to minimize the time to recovery.

VAST Data Locally Decodable Codes

Storage systems have historically kept erasure-coded stripe sizes narrow. That's because the RAID and Reed-Solomon erasure codes they use require systems to read all the surviving data strips in a protection stripe, and one or more parity strips, to regenerate data from a failed drive.

To understand this problem in practice, consider a storage system using the same distribution of 146 data strips and four (4) parity strips per erasure-code stripe (146D+4P) as a large VAST Cluster with Reed-Solomon erasure codes. When an SSD on that system fails, the system has to read the contents of all the surviving data SSDs plus one of the four parity strips, for a total of 146X the failed SSD size.

To avoid rebuilds of that magnitude, which would consume a large fraction of our hypothetical system's performance for days or weeks, storage vendors limit the blast radius of device failures by keeping erasure code stripes below 16 – 24 total data and parity strips per stripe.

To get around this problem, VAST Data designed a new class of erasure code borrowing from a new algorithmic concept called locally decodable codes. The advantage of locally decodable codes is that they can reconstruct, or decode, data from a fraction of the surviving data strips within a protection stripe. That fraction is proportional to the number of parity strips in a stripe. A VAST cluster reconstructing a 146D+4P data stripe only has to read 38 x 1-data strips, just ¼th of the survivors.

How Locally Decodable Erasure Codes Work

VAST's locally decodable codes calculate each protection strip from slightly different sets of data strips across more than just one stripe of data. This allows protection strips to "stand-in" for large groups of data strips during rebuild calculations. By having three parity strips "stand-in" for ¾ of a stripe's data strips, the system only needs to read from 1/4th of the surviving data strips in a rebuild to perform a recovery within a stripe.

Storage-Class Memory Eliminates the Need for Nerd Knobs

Too many storage systems present administrators with a dizzying assortment of stripe widths and RAID protection levels. The VAST DataStore can maintain a fixed selection of n+4 data protection because the very large SCM write buffer in a VAST system decouples write latency from the backend locally decodable stripe layout. In other words, it reduces latency by acknowledging writes once they are stored in the SCM and it can handle the encoding process asynchronously.

Writes to SCM are mirrored. The low latency of SCM combines with the fact that, unlike flash, SCM devices don't have a logical block addressing (LBA), page, block hierarchy, or complications that hierarchy creates. Their prodigious endurance isn't dependent on write patterns the way flash SSDs are. This means the SCM write buffer provides consistently high write performance regardless of the mix of I/O sizes. Any data written by an application is written immediately to multiple SCM SSDs. Once data is safely written to the SCM, the write is acknowledged to the application. Data is migrated to hyperscale flash later, after the write has been acknowledged.

On the flip side, the random access of flash combines with VAST's wide write stripes and parity declustering to ensure that reads are very fast because all writes are parallelized across many flash devices. That said, locally decodable erasure codes do not require a full-stripe read on every operation. To the contrary, reads can be as little as one SSD chunk and as large as a multitude of stripes. In this way, there is only a loose correlation between the width of a stripe and the width of a file, object, directory, or bucket read.

Intelligent, Data-Only Rebuilds

The VAST DataStore knows which blocks on each SSD are used for active data, which are empty, and which are used but are holding deleted data. When an SSD fails, we only have to copy the active data and can ignore the deleted data and empty space.

Declustered Parity

Rather than clustering SSDs to hold data strips, parity strips, or spares, the VAST DataStore distributes erasure-coded stripes across all the SSDs in the system. The system selects which SSDs to write each stripe of data, based on SSD space and wear, not the locations being written. Data is then layered into the 1 GB strips via a collection of sub-stripes. which can each be written by different VAST CNodes.

Because of the DASE architecture's shared-everything nature, recovery processes are parallelized across all the VAST CNodes in the cluster. VAST's declustered approach to data protection ensures that every device has some amount of data and parity from an arbitrary collection of stripes. This enables a reconstruction event to be federated across all available SSDs and all VAST server resources in a cluster. A large VAST DataStore System will have dozens of VAST Servers sharing the load and shortening rebuild time.

A Fail-in-Place Cluster

As storage systems scale to hundreds of SSDs, failures are inevitable. VAST systems are designed to allow SSDs to fail in place. Rebuilds in large clusters are not dependent on failed devices being replaced, even if multiple SSDs fail. VAST systems always write erasure code stripes somewhat narrower than the total number of SSDs in the system, so there's space to rebuild into.

VAST clusters rebuild from multiple drive failures by writing the reconstructed data from the failed drives to free space on drives that didn't participate in the erasure code stripe being rebuilt. If, after multiple drive failures, the cluster reaches a state where there are no SSDs that aren't holding part of the damaged erasure code stripes, the cluster will restripe all of its contents in narrower erasure code stripes. Either way, the system will continue to operate and maintain protection against four additional SSD failures as long as there's enough free space available.

VAST Checksums

To protect user data from the silent data corruption that can occur within SSDs, the VAST DataStore keeps a checksum for each data and metadata chunk in the system. These checksums are CRCs calculated from the data chunk or metadata structure's contents and are stored in the metadata structure that describes the data chunk. The checksum for a folder's contents is stored as part of its parent folder, the checksum for a file extent's contents in the extent's metadata, and so on.

When data is read, the checksum is recalculated from the data chunk and compared to the checksum in the metadata. If the two values are not the same, the system will rebuild the bad data chunk using locally decodable parity data.

However, solutions that store checksums in the same block (or chunk) as the data, such as ANSI T10 Data Integrity Field standard (T10-DIF), only protect data from SSD bit rot, but they do not protect the data path from internal system data transfer errors. If, in the case of T10-DIF, there is as little as a 1-bit error when transmitting an LBA address from an SSD, and the SSD actually reads LBA 33,458, a T10-based system will return the wrong data, because it's reading the wrong LBA, but the data and checksum will match because they're both from LBA 33,458 even though we wanted the data from 33,547.

To protect data being stored on the system from experiencing an accumulation of bit errors, a background process also scrubs the entire contents of the system periodically. Unlike the CRCs that are attached to VAST's variable-length data chunks, the system creates a second set of CRCs for 1 MB flash sub-strips in order to swiftly perform background scrubs. This second level of CRCs solves the problem of customers who store millions, or billions, of 1-byte files; VAST's background scrubber can deliver consistent performance irrespective of file/object size.

Encryption at Rest

Encryption at rest is a system-wide option. When enabled, VAST systems encrypt all data using FIPS 140-3 validated libraries as it is written to the Storage Class Memory (SCM) and the hyperscale SSDs.

Even though Intel's AES-NI accelerates AES processing via microcode, encryption and decryption still require significant CPU horsepower. Conventional storage architectures, like one scale-out file system that always has 15 SSDs per node, can only scale capacity and compute power together. This leaves them without enough CPU power to both encrypt data and deliver their rated performance.

Conventional storage vendors resolve this by using self-encrypting SSDs. Self-Encrypting Drives (SEDs) offload the encryption from the storage controller's CPU to the SSD controller's CPU, but that offload literally comes at a price; that is, the premium price SSD vendors charge for enterprise SEDs.

To ensure that we can always use the lowest-cost SSDs available, VAST systems encrypt data on the hyperscale SSDs in software, avoiding the cost premium and limited selection of self-encrypting SSDs. VAST's DASE architecture allows users to simply add more computing power, in the form of additional VAST CNodes, to accommodate the additional compute load encryption may present.

Rather than being a performance limitation, or significant additional cost, VAST encryption becomes just one more factor in balancing the performance provided by VAST CNode CPUs and the capacity of VAST enclosures.

The Logical Element Store Layer—Building Elements from Data Chunks

As we've seen, the Physical Layer takes a revolutionary new approach to the device management and data protection functions traditionally performed by logical volume managers, RAID controllers, and SAN arrays, delivering unprecedented scale and efficiency. By that same measure, the VAST Element Store is an Uber-namespace generalized and abstracted to store structured as well as unstructured data.

It's important to note that the physical and logical layers of the VAST DataStore are much more tightly integrated than our comparison to a logical volume manager and file system/object namespace would imply. In those legacy architectures the RAID system presents virtual volumes and the file system can only address the media by Logical Block Addresses (LBAs) on those virtual disks.

The Element Store Layer's metadata provides the structure that organizes the data chunks stored in the Physical Layer into Elements of several types. Because the Element metadata is managing logical chunks, not just blocks on a virtual disk, the Element layer and chunk layer are much more tightly integrated than a volume manager and file system.

Each Element is defined by its metadata and the methods/protocols used to access it. Since the goal of the VAST DataStore is to provide universal storage with universal access so users can access all their data via whatever protocol is convenient for them at the time. The Element Store maintains an abstract set of metadata properties about each Element that includes metadata properties required by any access protocol.

This allows VAST system operators to apply S3 metadata tags to Elements that are never actually accessed as S3 objects, like database tables and NVMe over Fabrics volumes. A university could tag such Elements with ownership information and then, for example, quickly query The VAST Catalog to find all of Dr. Doofenshmirtz' data when he left the university due to his imprisonment.

The VAST Element Store manages several types of data elements:

- **File/object** – File/object Elements are stored by the system as strings of data chunks
    - File/object Elements are accessed through the NFS, SMB, and S3 protocols
- **Volume** – Volume Elements, like file/object Elements, are stored as strings of data chunks. Volume Elements have additional access control list and authentication metadata including CHAP handshakes.
    - VAST systems make volume objects available via NVMe over Fabrics (more specifically NVMe/TCP)
- **Table** – Table Elements hold structured, tabular data in a columnar format
    - Table Elements are accessed via SQL through the VAST DataBase
- **Event Triggers** – Event triggers define the conditions that will trigger the VAST DataEngine to execute a user function

We've long talked about files and objects as being unstructured data while database management systems held structured data. To the VAST Element Store, unstructured means that the contents of the Element are opaque to the VAST AI OS. The system can reduce and fingerprint the data, but if customers want to glean insight from the contents of those Elements it's going to have to come from outside the system. Structured data therefore means that data where the system understands the internal structure of the Element and can therefore access the contents to glean insight.

Inherently Scalable Namespace

Users have long struggled with the scaling limitations of conventional file systems for a variety of reasons, such as metadata slowdowns when accessing folders that contain more than a few thousand files. Other systems deal with challenges around having preallocated inodes that limit the number of files that can be stored in a filesystem as a whole. These limitations were one of the major drivers behind the rise of object storage systems in the early 2000s. Freed from the limitations of storing metadata in inodes and nested text files, object stores scale to billions of files over petabytes of data without being bogged down.

The VAST DataStore provides the essentially unlimited scalability promised by object stores for applications using both object and file access methods and APIs. The DataStore's V-Trees expand as enclosures are added to the VAST cluster, with consistent hashes managing the distribution of V-Trees across enclosures (all without explicit limits on object sizes, counts or distribution).

Discovering a New Element – The Table

We've always called the namespace a VAST Cluster creates and presents as an Element Store because The VAST Element Store can accommodate the hierarchical structures of file systems and the flat organization of object buckets, using the word Element to mean a file or an object and. While files and objects are accessed by different protocols, they are both opaque containers, with the VAST system ignorant of their internal structures.

The VAST DataStore uses B-Tree structures to build File/Object type Elements as a simple list of the data chunks we described in the physical layer section. Elements that hold tables are different because the VAST DataStore is aware of their internal structure. The metadata for Table type Elements still maps the table's contents to physical layer data chunks. Where the metadata for a File/Object Element sequentially maps byte offsets within the file to data chunks, the metadata for Table Elements maps the table's contents by row and column to those data chunks.

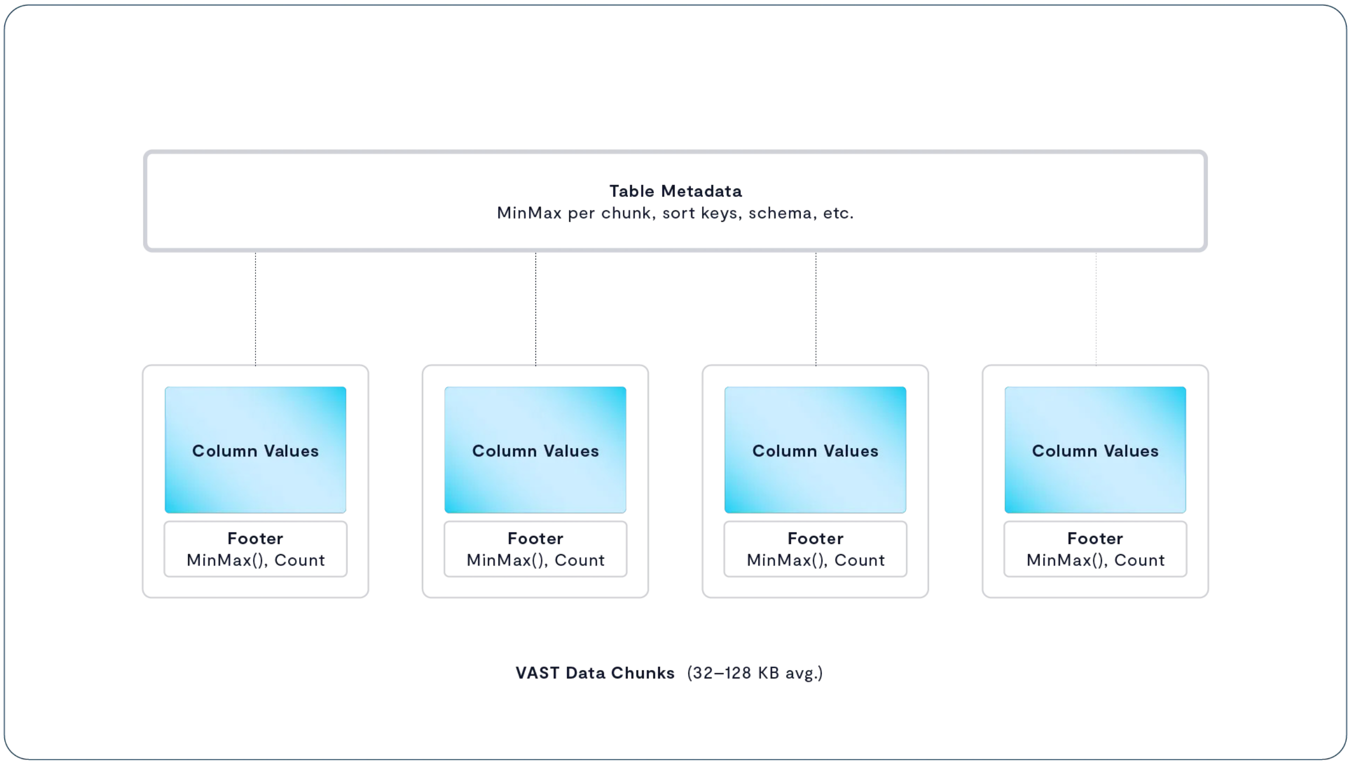
Because that metadata is stored in shared non-volatile memory in VAST DBoxes, it's available to, and shared by, all the CNodes in the VAST cluster. There's no metadata server to become a bottleneck or metadata cluster to manage, allowing a single VAST cluster to scale to exabytes. It also allows the VAST DataStore to be fully ACID so it can support the transaction consistency applications expect from relational databases as well as the query performance required to suggest a hotel, or movie, you would like in seconds.

To be more specific, the VAST DataStore stores tables into reduced data chunks with a columnar structure that is similar to the way Parquet files are organized but at a much finer granularity. Each data chunk holds the values of one or more columns for a row group in the table. Some of the statistics about those column values are stored in the table's metadata as well as the chunk footer to accelerate queries.

Because data is stored in its natural form as a table by the VAST DataStore, users are relieved of the tasks of partitioning their tables across files of the right size to maximize the large sequential reads that disk-based object stores are good at. Perhaps paradoxically, combining the storage and data layout functions into a single VAST DataStore relieves data scientists of the need to plan table layouts to accommodate storage limitations.

If the data was in Parquet files in an S3 data lake with the recommended row group size of 128MB to 1GB in size, then Spark, Trino, or some other query engine would have to read the footer from all the Parquet files in the bucket, then read an average of 512MB of data for each row group that isn't disqualified from the query because, for example, the min in the footer is greater than the range being queried.

The VAST DataBase can perform the whole footer scanning stage by reading from the metadata store alone, without reading any data chunks and since the data chunks are small (4,000x smaller than a standard Parquet row group according to Apache Iceberg best practices) will have to read much less data for each row group identified as having data of interest from footer data. Since the VAST footer is logical, it could also be extensible allowing customers to add additional statistics to the footers to accelerate their common queries.



This treatment of tables as a special class of Element is just one aspect of the unique level of integration within the VAST AI OS. Previous data management platforms were built using discrete components. In those systems structured data meant that database management engine imposed structure on the data in the files it created but none of that structure carried through down to the file system.

The VAST DataStore uniquely uses one set of integrated metadata to manage data from a specific field in a table through the Element of the table itself to a particular data chunk and ultimately to a location on an SSD. In this section we've explored how tables are stored, we'll examine how the VAST AI OS manages tables and the services it provides for tables in the section on The VAST DataBase.

Even though we've broken up the conversation here to make it easier to understand and provide simple comparisons to the traditional data platforms it's important to remember that the VAST AI OS is one integrated whole managing data from field/cell to SSD and beyond.

Element Store Data Services

Legacy storage systems have traditionally provided data services such as snapshots and replication at the volume or file-system level. This forces users to create multiple file systems to provide different service levels to their datasets.

The VAST DataStore creates a single namespace, eliminating the need for users to manage intermediate abstractions such as volumes and file systems. Instead, the VAST DataStore provides snapshots, replication, and similar services for any folder/bucket in the VAST Element store.

Low-Overhead Snapshots and Clones

A write-in-free-space storage architecture is especially suitable for high-performance snapshots because all writes are tracked as part of a global counter, and the system can easily maintain new and old/invalid state at the same time. VAST brings this new architecture to make snapshots painless by eliminating the data and/or metadata copying that often occurs with legacy snapshot approaches.

The VAST Element Store was designed to avoid several of the mistakes of previous snapshot approaches and provides many advantages for VAST DataStore users.

- VAST Snapshots leverage zero-copy mechanisms for both data and metadata to minimize the bad performance effects of storing and deleting snapshots
- VAST Snapshots do not require snapshot space/reserves. Snapshots use any free space and provide greater flexibility of media utilization as compared to more rigid approaches
- VAST Clusters experience negligible performance impact when they have an arbitrary number of snapshots active

Instead of making clones of a volume's metadata every time a snapshot is created, VAST snapshot technology is built deep into the metadata data structure itself. Every metadata object in the VAST Element Store is time-stamped with what is called snaptime. The snaptime is a global system counter that dates back to the installation of a VAST cluster and is advanced synchronously across all the VAST CNodes in a cluster approximately once a minute.

As with data, metadata in the VAST Element Store is never directly overwritten. When an application overwrites a file or object, the new data is written in free space across the SCM write buffer. The system creates a pointer to the physical location of the data and links that pointer into the V-Tree metadata that defined the object.

Following VAST's general philosophy of efficiency, VAST snapshots and clones are based on the small (average size 16-64 KB) data chunks that are the backbone of the VAST DataStore's physical layer. That's significantly finer grained than those on many conventional storage systems which may snapshot on pages of 1 MB or larger.

When you create a snapshot in the VAST Element Store, the system preserves the most recent version of the metadata pointers within the protected path, along with the data chunks that metadata references until the snapshot is deleted. The snapshot presents the data using the metadata pointers with the latest snaptimes earlier than the snapshot's time.

Because VAST snapshots are based on snaptimes, all snapshots taken at the same time are consistent across a VAST cluster. There's no need to define consistency groups; just use a single protection policy for all the paths you need to take consistent snapshots across.

When the system performs garbage collection it will delete the metadata pointers that aren't either the latest version of that pointer or used by a snapshot, and any data used exclusively by those pointers.

Users can, of course, create snapshots on demand with a call to the system's REST API, or through the GUI which will call the REST API itself, but most snapshots are managed though protection policies.

Once a protection policy is established the VAST DataStore presents a ./snapshots system folder in the root folder of each protected path. The ./snapshots folder will contain a subdirectory for each snapshot that provides read-only access to the snapshot's contents. Several VAST customers use the ./snapshot feature to provide self-service restores to their users.

Truly Independent Clones

VAST clones provide rapid read-write access to a snapshot's contents by creating an independent replica of a snapshot's contents. When a VAST administrator creates a clone of a snapshot through the RESTful API, the system performs a background server-side copy of the snapshot's contents to the specified directory.

The new folder and its contents appear and are available for both reads and writes in seconds. While the cloning process is in progress, read requests for data from the clone will be satisfied from the snapshot. Since the VAST DataStore only keeps a single copy of any data chunk,these copies occupy no space in the DataStore. This process can also be used to create clones of snapshots on a remote VAST cluster. See Global Clones below for details.

Administrators can choose to perform a full background copy of the snapshot's contents or to let the system be lazy and copy data to the clone only as it is accessed. Lazy clones minimize the amount of data copied, and therefore the impact on the system, making them perfect for when you have to mount multiple clones to locate the last known good point in time.

Once the background copy is complete, the new folder has a metadata structure that is completely independent (other than the fact that both sets of metadata point to the same data chunks in the VAST DataStore's physical layer). When a user clones their Windows Server golden image after every patch Tuesday, there's no complex dependency web amongst those clones of clones of clones of clones.

Indestructible Storage Protects from Ransomware and More

Users today face a plethora of ever-more sophisticated attacks on their sensitive data. Snapshots provide protection against ordinary users encrypting or deleting data. Today's ransomware attacks are more sophisticated, acquiring administrator/root/uberuser privileges and deleting or encrypting snapshots and backups as, or before, they attack the user's primary data.

VAST's Indestructible Snapshots protect against these sophisticated attacks and any other attacks that depend on elevated privileges, such as disgruntled administrators turning rouge, by preventing anyone, regardless of their administrative privileges on the system, from deleting Indestructible Snapshots before their expiration date.

To create Indestructible Snapshots users simply select the Indestructible option when creating the snapshot. There are three options to do this: through the GUI, through the REST API in the script that runs at the end of a backup job, or in the Protection Policy. Once an

Indestructible Snapshot is created, that snapshot is immutable—no user, not even the proverbial root user, can delete the snapshot or modify the policy.

While truly indestructible snapshots that no one could ever delete sound appealing to corporate CISOs, in the real world, there may be situations where customers need to delete snapshots they've marked as Indestructible or run out of space. In those cases, VAST support can provide a time-limited token to the user that will temporarily enable deleting Indestructible Snapshots.

Support will only release a time-limited token to a pre-authorized customer representative after meeting the conditions set in advance by that customer. Customers may require that three of seven designated people make the request, that a secret pass phrase be spoken, or any other mostly reasonable way to authenticate both the recipient and that there is an emergency.

Protection Policies Unite Snapshots and Replication

Users can of course create snapshots on demand with a call to the system's REST API, or through the GUI which will call the REST API itself, but most snapshots are managed through protection policies. A protection policy defines a multi-tiered snapshot and retention schedule for both local and replicated snapshots.

Snapshots can be taken, and replicated, as frequently as every 15 minutes, or as rarely as once every several years, with the same control over retention. A protection policy can support multiple tiers of snapshots like the example below:

- Snapshot every 15 minutes starting at 12:00:00 retain 4 hours
- Hourly at 12:00:00 retain 1 week
- Daily at 12:00:00 retain 40 days
- Monthly at 12:00:00 retain 1 year

Since all these snapshots are scheduled for the same time, they'll be congruent (the monthly snapshot is consistent with the 15-minute snapshots). One policy can be used to protect multiple paths/buckets, allowing admins to create their own zinc, tin, pewter protection levels.

The Protocol Layer Provides Multiprotocol Access

We've looked at how the VAST DataStore manages a VAST Cluster's media, data, and most importantly, metadata. But there's more to an AI Operating System or even a just storage system, than just storing data. The system also needs to make that data available to users and applications, define and enforce access security, and enable cluster administration. That access, and the associated controls, are provided by the protocol layer of the VAST DataStore's stack.

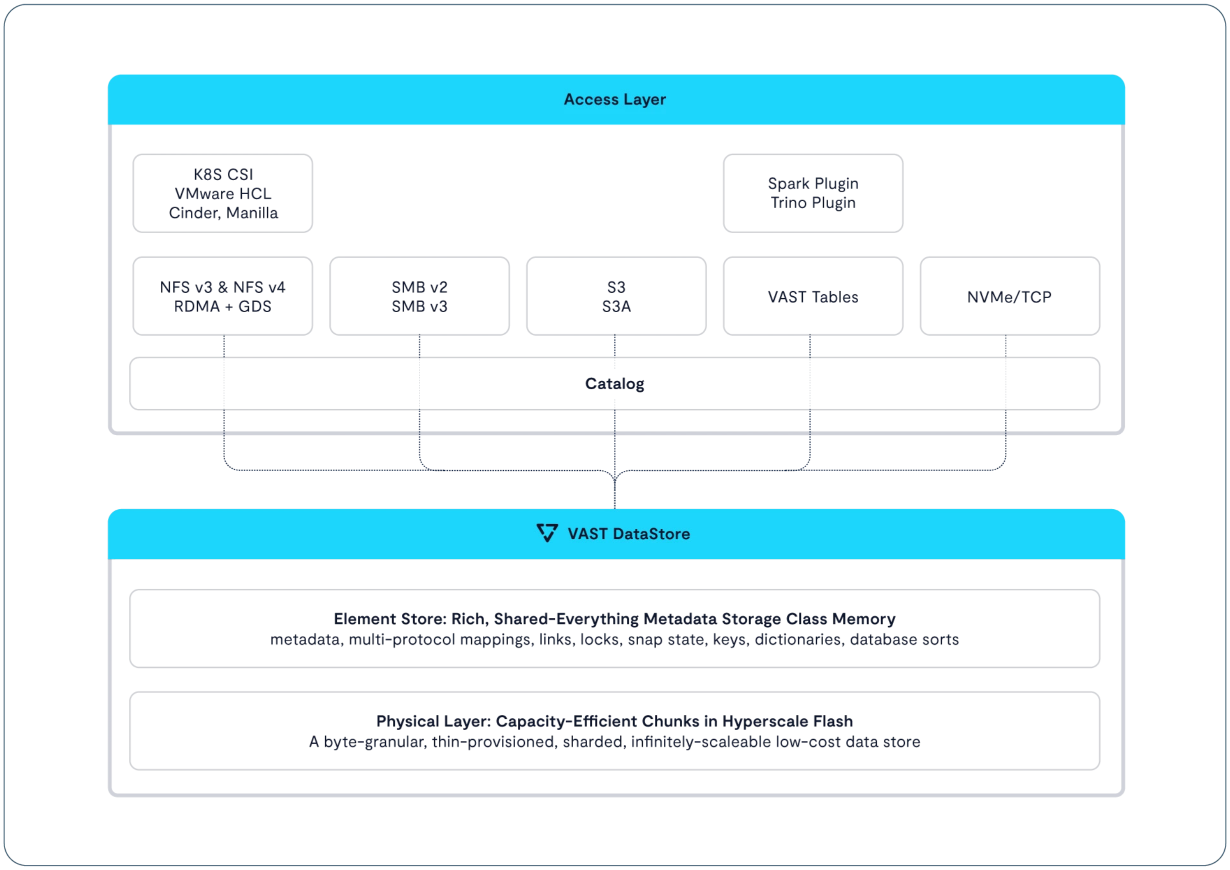
Just as the VAST Element Store is a namespace abstraction that combines the best parts of modern file systems with the scale and extended metadata of an object store (not to mention the ability to manage structured tables, and event triggers), the Protocol Layer provides a protocol-independent interface to the individual elements in the Element Store.

Some vendors support new (S3), complex (SMB), or for native object stores simply foreign (NFS, SMB) protocols by integrating open-source projects like Minio and SAMBA into their products. While this lets them check the box for multiprotocol support, it means these additional protocols are really second-class citizens.

Because the actual back-end storage is designed for a single protocol, cross-protocol ACLs are difficult, especially when the storage doesn't support an ACL concept such as deny. The other problem is that the open-source module is always running as a gateway process that, even if it doesn't translate SMB directly into S3, still has to translate requests to fit the back-end storage. What's more, most open-source, or even commercially available, protocol modules have limited scalability. Given all this, it became clear that VAST would be better off keeping protocol development in-house, which also allows VAST protocol modules to access the VAST Element Store more intimately than standard APIs would allow.

VAST develops all the protocol modules in-house as equals. The namespace abstraction provided by the Element Store enables VAST Data to add support for additional file, block, big data, and yet-to-be-invented protocols over time simply by adding additional protocol modules.

It also means that VAST systems provide similar performance through all the supported protocols.


Access Layer
K8S CSI
VMware HCL
Cinder, Manilla
Spark Plugin
Trino Plugin
NFS v3 & NFS v4
RDMA + GDS
SMB v2
SMB v3
S3
S3A
VAST Tables
NVMe/TCP
Catalog
VAST DataStore
Element Store: Rich, Shared-Everything Metadata Storage Class Memory
metadata, multi-protocol mappings, links, locks, snap state, keys, dictionaries, database sorts
Physical Layer: Capacity-Efficient Chunks in Hyperscale Flash
A byte-granular, thin-provisioned, sharded, infinitely-scaleable low-cost data store

The individual elements in the VAST Element store are also protocol independent. This means all elements, and the entire capacity of the VAST cluster, can be made accessible by any supported protocol. This lets users access the same elements over multiple protocols. Here's an example of multi-protocol access in practice: a gene sequencer stores genomic data via SMB; an application watching the folder on a Linux server via NFS notices the file, runs it through its inference engine, and writes its results as metadata tags via an S3 API call. Then this data can be made available via S3 to a pipeline build from some cloud framework.

VAST supports versions 3 and 4.1 of NFS for Linux and other open systems clients; versions 2.1 and 3.1 of SMB, the preferred file protocol for Macintosh and Windows computers; and S3, the de facto standard for object storage. Access to Table type Elements is provided via VAST SQL and associated plug-ins for Trino, Spark, and other platforms.

The VAST AI Operating System provides access to block volumes for applications like virtual machines, and Kubernetes clusters via NVMe over Fabrics.

Multiprotocol ACLs

One challenge of providing truly useful multi-protocol storage is controlling access through the very different security models expected by Linux and other open systems, NFS users, and Windows SMB users.

Classic NFS follows the Unix file security model. Each file has a nine-bit security mask that assigns read, write, and execute permissions to the file's owner, a group, and any user that's not a member of the group, known as Other. This model made aligning security groups with business organizations difficult by only granting permission to a single group.

Posix ACLs add flexibility by adding support for multiple named users and multiple named groups in the access control list for a file or folder.

The access control lists in Windows NTFS are significantly more granular, allowing administrators to control whether users can list the files in a folder, delete subfolders, or change the permissions of the folder for other users. Most significantly, Windows ACLs have a deny attribute that prevents a user or group from inheriting permissions from a folder's parents.

NFS 4.1 defines a new ACL format that's granular like Windows ACLs but just different enough to add another level of complication.

As we spoke with users, it became clear that most datasets have a dominant security point of view. Some shared folders will be used primarily by Linux servers with occasional access by Windows systems. Users wanted to manage access to these data sets with POSIX- or NFS 4-style ACLs.

The users we spoke to also had other folders primarily used by humans operating Macs and Windows PCs. Here they wanted the finer-grained access control and familiarity for the users provided by Windows ACLs.

Just as the VAST Element Store abstracts data from the file system and object store presentations, it also stores access control lists in an abstracted form. Those abstracted ACLs can be enforced as Unix Mode Bits, POSIX, Windows, or NFS 4 format ACLs as clients access the data via multiple protocols. Even with that abstraction, the SMB and NFS views of how ACLs should be processed, modified, and inherited are different enough that we let users assign any View in the system to present NFS- or SMB-flavor ACLs.

Note that a view's flavor determines which type of ACLs are dominant, not whether the view presents itself as an NFS Export, an SMB share, or both, which is controlled separately.

NFS-flavor Views act as NFS clients would expect in every way. Users can query and modify ACLs using the standard Linux tools over NFS. Those ACLs will be enforced on users accessing via both the SMB and NFS protocols.

SMB-flavor Views are managed like Windows shares and allow users to set fine-grained Windows ACLs through PowerShell scripts and the file explorer over SMB. Those ACLs, including denies, are enforced on NFS as well as SMB access

QoS Silences Noisy Neighbors

Most IT organizations have a historically well-founded fear that one or more performance-hungry applications will saturate any given storage system's ability to deliver data and cause other applications to slow down.

Those performance-hungry applications are the data center equivalent of the frat bros who rent the apartment next to yours and play death metal all night: the proverbial noisy neighbors. Fear of noisy neighbors has been a key driver of data siloization, forcing organizations to have multiple solutions to provide dedicated storage for different applications or constituencies.

VAST systems provide multiple mechanisms that VAST admins can use to keep noisy neighbors from disturbing other applications.

Performance in the DASE architecture, in particular small I/O performance, is a function of the amount of CPU power available to process user requests. Admins can therefore allocate performance to applications, or user communities, by assembling server pools that dedicate the CPU power of a set of CNodes to those applications.

Server pools dedicate the performance of CNodes to a purpose, but that control is very coarse grained; after all, a CNode is a significant amount of computing power. For finer grained control VAST clusters have a more direct method to control the Quality of Service (QoS) provided.

Admins can assign any View (mount/share) QoS limits in terms of bandwidth and/or IOPS with independent limits for reads and writes. These QoS limits can be absolute or relative to the capacity used, or provisioned, for the View, allowing service providers to have a 200 IOPS/TB class of service.

VAST NFS

Parallel File System Speed, NAS Simplicity

Let's look at each protocol in detail, starting with NFS.

NFS, the Network File System, has been the standard file sharing protocol for open systems since 1984. VAST's NFS 3 implementation includes important, standardized extensions to NFS for security (POSIX ACLS), and data management (NLM).

Traditional NFS over TCP has been performance-limited because the TCP transport protocol will only send a limited amount of data before it receives an acknowledgment from the receiver. This limit on data in flight, sometimes called the bandwidth-delay product, restricts a single NFS over TCP connection from a client to a server to approximately 2 GB/s even on 100 Gbps networks.

Users have traditionally taken one of two complex approaches to solving this problem. They use multiple mount points, and therefore multiple connections to the NAS or switch to a complex parallel file system.

Multiple mount points provide a limited performance boost at the cost of more complicated data management and do nothing to speed up a single application accessing a single mount. Parallel file systems require client-side file system drivers that limit and complicate client OS choices and updates.

While NFS is almost 40 years old, that doesn't mean it hasn't been modernized and accelerated over those 40 years. NFS v4 introduces a much higher security model than NFS v3, with more granular ACLs and encryption in flight. Meanwhile, a series of NFS enhancements allow VAST systems to deliver the performance of a parallel file system without all the complexity introduced by parallel file systems.

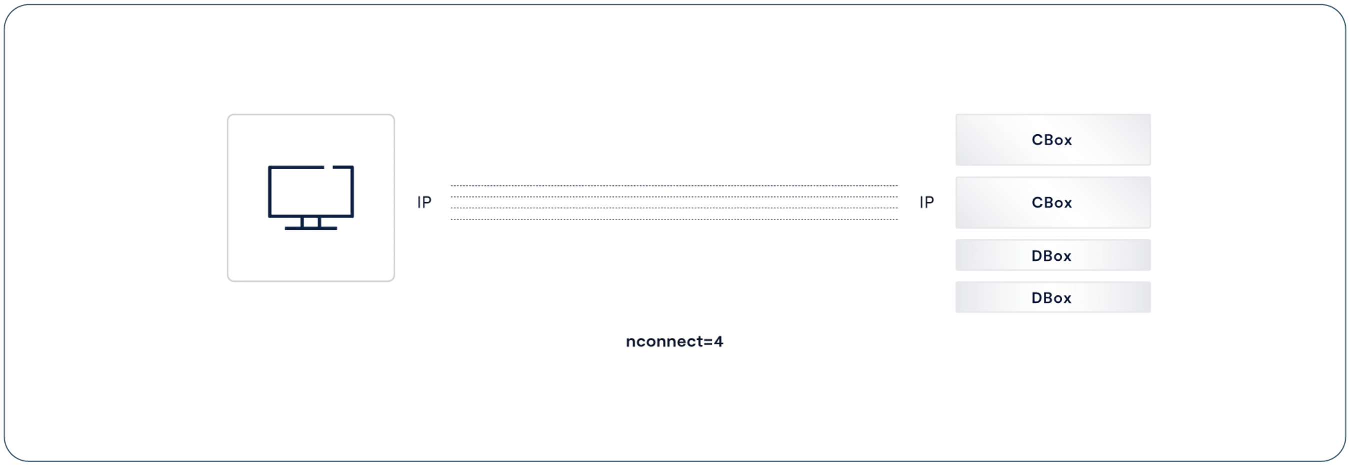
Accelerating NFS for the AI Era

VAST systems support three technologies to increase the performance of NFS-connected hosts:

- nconnect – A Linux mount option that spreads NFS over multiple TCP connections
- Multipath - Spreads multiple TCP connections from nconnect over multiple physical connections
- NFS over RDMA - Uses RDMA instead of TCP for low latency transport

nconnect for Multiple TCP Sessions

The first step to boosting NFS performance is the nconnect NFS mount option, which was added to the NFS client with the Linux 5.3 kernel in 2019. When an NFS client mounts an export with the nconnect=n option, it will load balance access to that export across n TCP sessions, not just one.



Where standard NFS over TCP maxes out at about 2 GB/s, connections with nconnect=5 or 8 can see 10 GB/s bandwidth with a 100 Gbps Ethernet connection.

VAST Adds Multipath

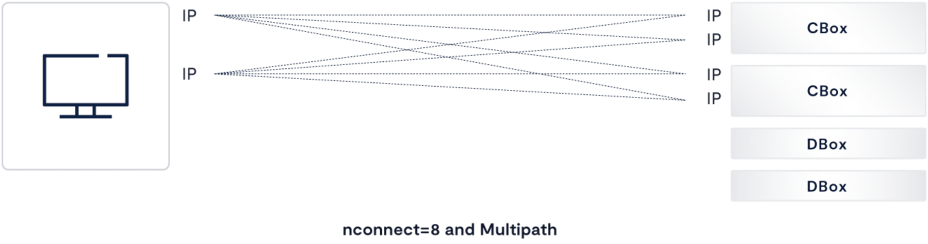
The nconnect mount option spreads the NFS traffic between an NFS client and an NFS mountpoint over multiple TCP sessions. As we've seen, this NFS connection exceeds the 2 GB/s bandwidth limit of a TCP session, but those TCP sessions are between one IP address, and therefore physical interface, on the client and one IP address on the NFS server.

That means the bandwidth boost provided by nconnect is limited to the speed of a single network connection; good for servers with 100 Gbps Ethernet cards, not so good for the video editor on a workstation with two 10 Gbps Ethernet cards.

To solve this problem VAST developed an open-source NFS driver that spreads the multiple TCP connections created by nconnect across the lists of client and server IP addresses specified. For example, if a client had mount options of:

```
Nconnect=8,ClientIP=10.253.3.17-10.253.3.18,ServerIP=10.253.4.122-10.253.4.25
```

The NFS client will set up 8 TCP sessions connecting both client IP addresses to all four server IP addresses, as shown below.



The VAST patch for both NFS 3 and NFS 4, which we've submitted upstream for inclusion in Linux distributions, not only spreads the traffic across multiple 10 Gbps links for that video edit station, but because each virtual IP address sends traffic to one CNode using multiple server addresses, it also spreads the traffic across multiple CNodes. The metadata in shared SCM on the VAST cluster serves as a single source of truth, allowing the VAST cluster to process parallel I/O across multiple CNodes and still remain strictly consistent.

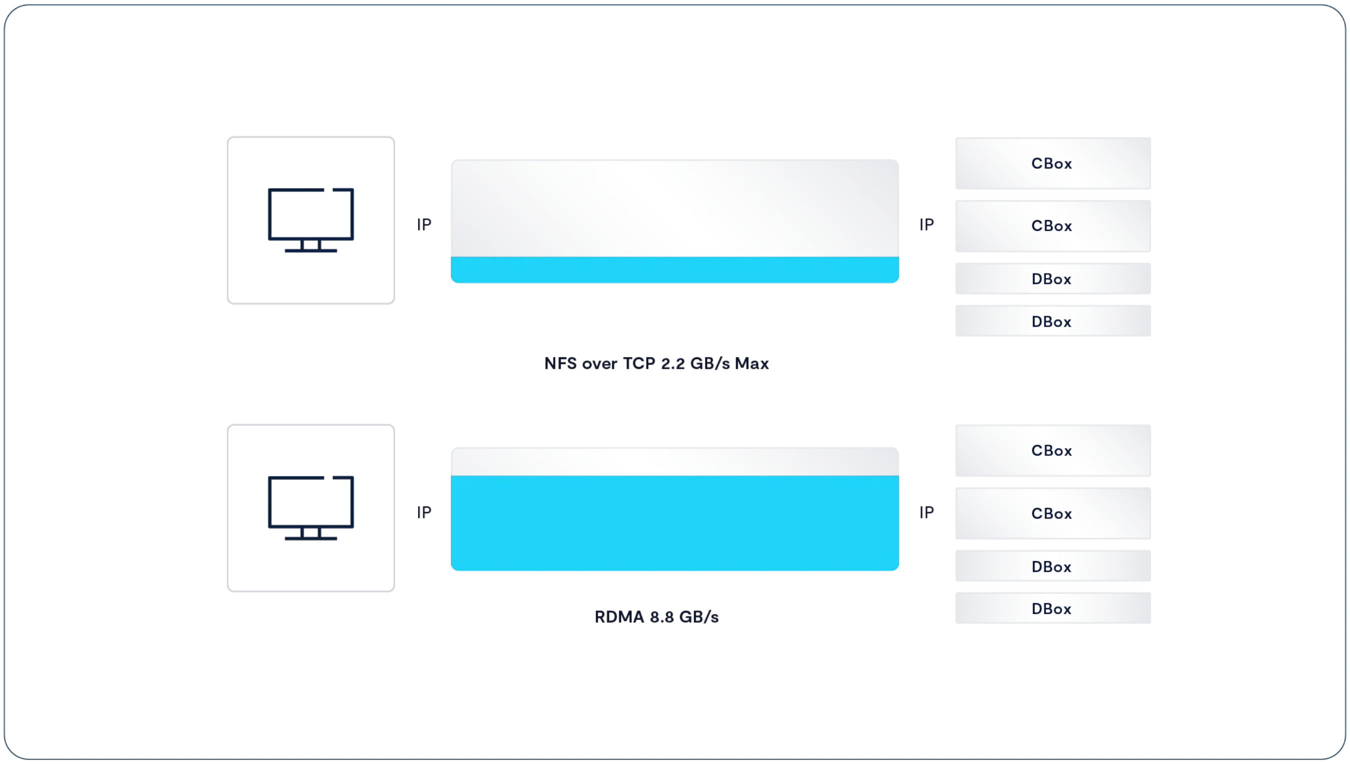
Using multiple network paths also boosts resilience for critical workloads by transparently routing traffic to the surviving link.

NFS over RDMA (NFSoRDMA)

In the mid-2010s, Oracle and other key contributors got together and updated NFS to use RDMA (Remote Direct Memory Access) to transport NFS RPCs between the client and server instead of TCP. Thanks to their work, NFSoRDMA has become part of all the major Linux distributions.

RDMA transfers data across the network directly into the memory of the remote computer, eliminating the latency of making copies of the data in TCP/IP stacks or NIC (Network Interface Card) memory. The RDMA API, known as RDMA Verbs, is implemented in the RNIC (RDMA NIC) offloading the transfer from local memory directly into memory on a remote computer and eliminating the user space/ kernel space context switches that add overhead to TCP connections.

RDMA Verbs were first developed for InfiniBand so NFSoRDMA can, of course, run over standard InfiniBand networks. More recently, RDMA Verbs have been built into Ethernet RNICs using the RoCE (RDMA over Converged Ethernet). This brings NFSoRDMA to Ethernet as well as InfiniBand data centers. RoCE v2 runs over UCP and doesn't require any special network configuration beyond enabling ECN (Explicit Congestion Notification), a standard feature of enterprise Ethernet switches since 10 Gbps networking ruled the roost.



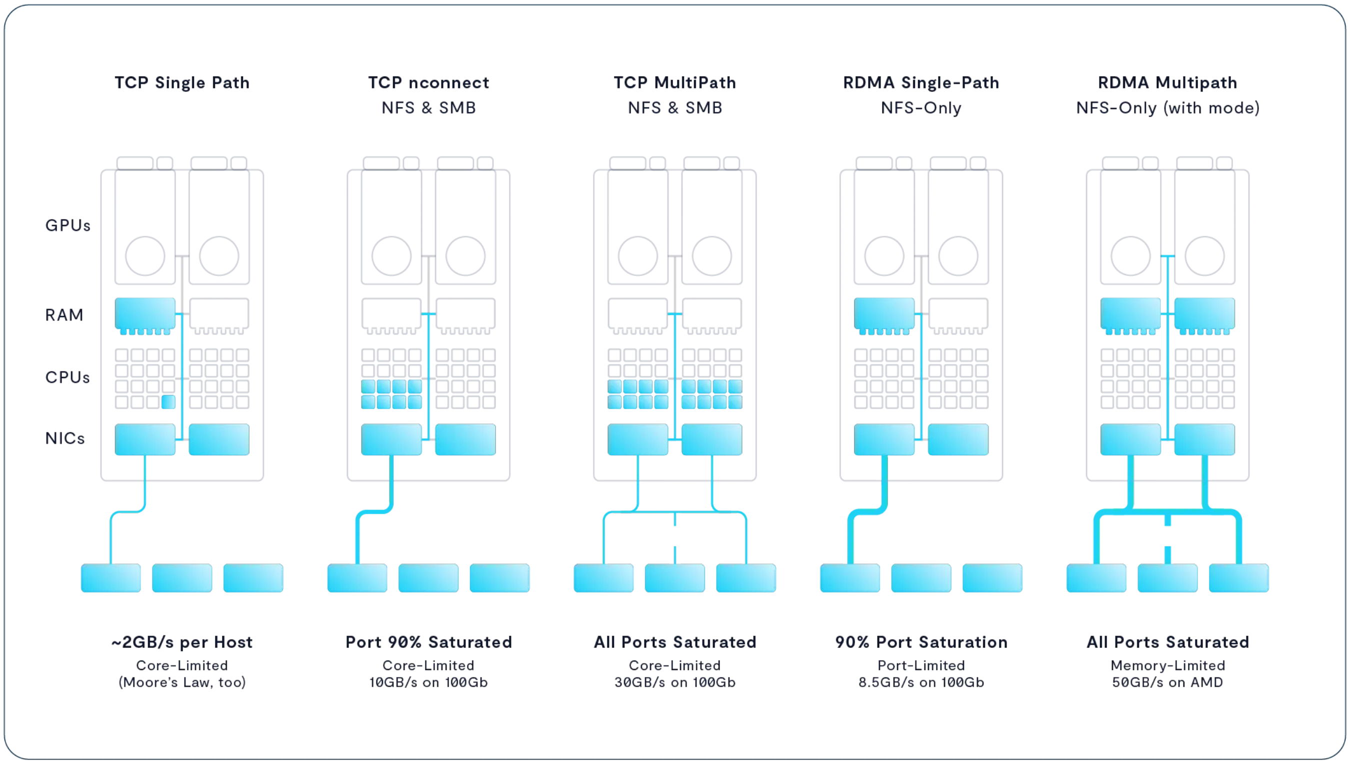
The net result is that where a single NFS over TCP session to a VAST Cluster is limited to 2 GB/s, a single NFSoRDMA session can deliver up to 70% of a 100 Gbps network's line speed, or 8.8 GB/s.

But the real beauty of NFSoRDMA lies in the simplicity of deployment: the NFSoRDMA client is standard in major Linux distributions, so it doesn't require any of the kernel patches or client agents that can make deploying and maintaining high-bandwidth parallel file system storage problematic.

With VAST NFS Means Now for Speed

In the past, users have had to use complex parallel file systems because the NFS servers of the day couldn't deliver the performance HPC, AI, and media workloads, among others, required.

Today, performance enhancements from nconnect to RDMA make NFS the protocol of choice for your most demanding workloads. The graphic below shows how each of these features involve more of the host resources to deliver greater performance. The active components in each data transfer are shown in blue.



NFS Extensions

VAST systems support several standard extensions to POSIX Access Control Lists (ACLs)

VAST is one of the few companies to implement ACLs that conform to the POSIX specification, allowing a system administrator to define broader and more detailed permissions to files and folders than is possible with the simplistic Unix/Linux model. (This model is limited to defining read/write and execute permissions to the root, a single user “owner” of the file or folder, and a single group.) Unlike Linux mode bits, POSIX ACLs are very flexible and allow administrators to assign permissions to multiple users and groups.

NLM Byte-Range Locking

VAST NFS also supports the NLM byte-range locking protocol defined in the standard Linux NFS-util. NLM, which originally stood for Network Lock Manager, defines a mechanism for NFS clients to request and release locks on NFS files and byte ranges within the files. NLM locks are advisory, so clients must test for and honor locks from other clients. NLM provides the support for shared and exclusive locks to applications and is designed for parallel applications where many byte-ranges can be locked concurrently within a single file.

VAST’s approach to NLM locking is inherently scalable because locking and lock management is fully distributed across the VAST Cluster. VAST Clusters don’t implement a centralized lock manager function or process. Instead, lock information is stored as extended file system metadata for each file in the VAST V-Tree. Since all the system metadata is available to all the VAST CNodes in the cluster, each VAST CNode can create, release, or query the lock state of each file it’s accessing without the central lock manager server that can so often become a bottleneck on other systems.

NFS Version 4

NFS version 4 was a major re-write of the NFS protocol and provides a significant improvement over NFS v3, especially where security is concerned. Unlike NFS v3 (and S3) NFS v4 is a stateful protocol that includes secure authentication via Kerberos and the detailed ACLs mentioned above.

VAST systems support the session-based NFS v4.1 over both TCP and RDMA, including support for NFS 4 byte-range locking, Kerberos authentication, Kerberos encryption in flight, and NFS v4 ACLs.

VAST SMB

File Services for Windows and Macintosh

While NFS is the native file access protocol for Linux and most other Unix derivatives, NFS isn't the only widely used file protocol. Windows and Macintosh computers use Microsoft's Server Message Block (SMB) protocol instead of NFS. The SMB protocol is also sometimes known as CIFS after an unsuccessful proposal by Microsoft to make SMB 1.0 an internet standard as Common Internet File System.

SMB is such a complex, stateful protocol that many NAS vendors choose to acquire it through the open-source SAMBA project or from one of the handful of commercial vendors that sell SMB code. Unfortunately, all the available SMB solutions have limitations, especially around scaling, that keep them from being good enough for VAST.

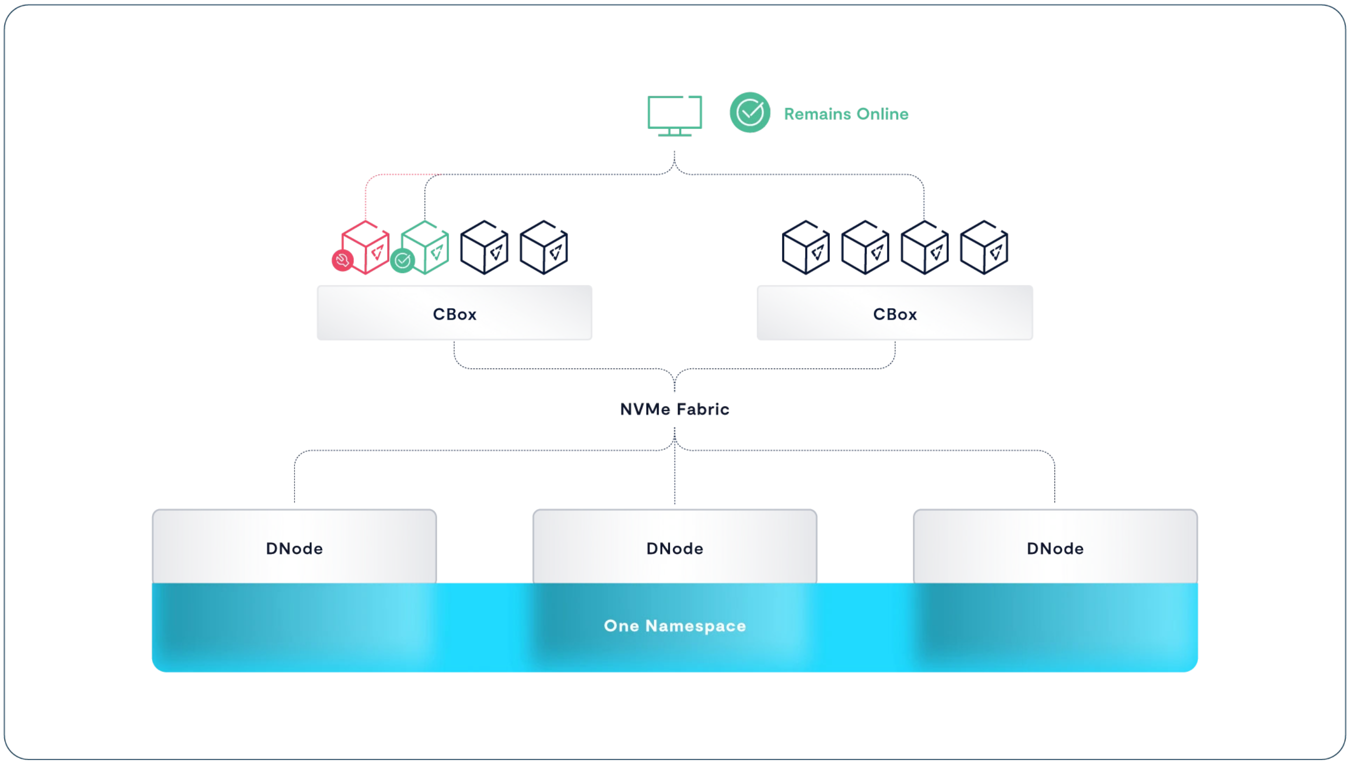
Instead, VAST developed our SMB protocol in-house to take full advantage of the DASE architecture storing session state in the cluster's shared SCM pool to allow SMB service across all the cluster's CNodes. Because the VAST DataStore is fully multi-protocol, SMB users can access the same files and folders as users accessing the system via NFS and/or S3.

File access to the VAST namespace is controlled by VAST Views. A VAST View is a protocol-independent version of an NFS Export and SMB Share. Administrators can enable NFS and/or SMB access to a view and set the access control flavor of the View to NFS or SMB.

VAST systems support SMB 2.1 and 3.1 including SMB Multichannel for enhanced throughput.

SMB Server Resilience

Every time an SMB client opens a file on an SMB server, the two systems assign an SMB handle to identify the connection. This requires both the client and SMB server to retain state information relating clients, files, open modes, and handles.



Many scale-out storage systems force users to manually retry or reconnect after a node failure, because while another node may take over the virtual IP address from the failed one, the dynamic state information, like handles, is lost. For an SMB client to automatically recover from a node failure, the state information has to be shared, and the shared-nothing model breaks down.

DASE and SMB Failover

In a VAST CNode, dynamic handles, file leases, and all the other state information that defines the relations between SMB clients and servers is stored in shared SCM (Storage Class Memory) in the cluster's VAST Enclosures. This allows the surviving VAST CNodes in a cluster to not only assume the IP addresses of a VAST CNode that goes offline, but also to resume where the offline server left off, reading the connection's state from SCM.

The SMB client sees the equivalent of a transient network glitch as the system detects a failed VAST CNode and recovers, exactly what SMB 2.1's resilient handles were designed to accommodate. No lost handles, and most importantly, users and applications continue as if the failure never happened.

VAST S3

Object Storage for Modern Applications

Amazon's S3 (Simple Storage Service) protocol, or more precisely the protocol used by Amazon's S3, has become the de facto standard for object storage. This is largely because it allows developers to support both on-premises object stores like VAST's VAST DataStore and cloud storage such as Amazon S3 and its competitors.

VAST's Protocol Manager exports S3 objects using HTTP GET and PUT semantics to transfer an entire object via a single request across a flat namespace. Each object is identified and accessed by a URI (Universal Resource Identifier). While the URIs identifying files can contain slashes (/s) like file systems, object stores don't treat the slash character as special, so that it has the ability to emulate a folder hierarchy without the complexity a hierarchy creates – slashes are just another character in an internal element identifier.

VAST Objects are stored in a similar manner to files, with the difference being that an S3 object includes not just its contents, but also user-defined metadata that allows applications to embed their metadata about objects within the objects themselves.

VAST Objects are stored in a manner similar to files, with the difference that an S3 object includes not just its contents but also user-defined metadata that allows applications to embed metadata about objects within the objects themselves.

While object storage has classically been used for archival purposes, the emergence of fast object access and, in particular, all-flash object storage, has extended the use cases for which object storage is appropriate. For example, many Massively Parallel Processing (MPP) and NoSQL Databases use object storage as their underlying data repository.

VAST DataStore systems support a subset of the S3 verbs that are offered as part of Amazon's S3 service. Whereas many of Amazon's APIs are specific to their service offering, VAST Clusters expose the S3 verbs that are required by most applications that benefit from an all-flash on- premises object store. That excludes tiering; a VAST system provides one tier, all-flash, and AWS-specific factors, like the number of availability zones each object should be replicated across.

As with VAST's NFS offering, S3 performance scales as enclosure and server infrastructure is added to a cluster. For example, a Cluster consisting of 10 enclosures can be read from at a rate of up to 230 GB/s for 1 MB GET operations, or at a rate of 730,000 random 4 KB GETs per second.

NVMe-Over TCP Block Services for the 21st Century

From their very inception, file and object storage systems were designed to provide a pool of shared storage that multiple computers could access simultaneously. Large-scale applications that use many servers to process a common data set are only practical because of this sharing.

However, some applications are somewhat more possessive about their data and insist on storing their data on disk drives, or at least virtual disk drives, that are entirely in their control. These applications generally involve some sort of tightly coupled application cluster. For example, many clusters allow only a single member to access a logical disk; they use device reservations to transfer ownership of the logical disk from one member to another when a node dies.

For the last 20 years or more, enterprises have addressed this problem with Fibre Channel SANs that connect their servers to expensive storage arrays. Those arrays slice and dice the SSDs and/or HDDs they control into virtual disk drives. Those virtual disks appear as locally attached SCSI devices to the server's operating system. Fibre Channel and iSCSI (the red-headed stepchild for Ethernet), simply transport SCSI between servers and array controllers. Each SCSI command reads or writes some number of 512-byte blocks from a starting LBA (Logical Block Address) starting at 0 on the logical disk we, for indefinable reasons, still call a LUN (Logical Unit Number – the drive's address on a SCSI bus).

As SSDs got more sophisticated, the industry developed NVMe (Non-Volatile Memory express) as a replacement for SCSI as the low-level command protocol between servers/controllers and SSDs. Because it was designed specifically for SSDs, NVMe takes advantage of the SSD's ability to process multiple requests in parallel, replacing SCSI's single command/data queue with 64K queues, reducing latency.

NVMe over Fabrics (NVMe-oF) extends the NVMe protocol across a resilient network, the fabric, just as Fibre Channel and iSCSI extended the SCSI protocol. The difference is that networks, like SSDs, have become smarter over the years since Fibre Channel was developed, and NVMe-oF can utilize these smarter networks as its fabric. VAST's DASE architecture uses NVMe over RDMA (Remote Direct Memory Access) to give the CNodes (server nodes) in the cluster ultra-low-latency access to the SSDs.

The VAST DataStore couldn't truly offer Universal Storage without supporting storage for applications, users, and admins that need block access to their data. Because we reject doing things the old way, we offer block storage with NVMe over Fabrics—specifically, NVMe over TCP (NVMe/TCP).

The block LUNs are stored as Elements in the VAST Element Store. NVMe/TCP has lower network requirements than NVMe over RDMA and provides comparable performance in many benchmarks.

All the major operating systems and hypervisors include NVMe over Fabrics initiators that will allow them to map LUNs on their VAST systems to drives on their servers using NVMe/TCP. Since any CNode in a VAST cluster can process any request for data from a LUN Element as well as any other CNode, there's no need for special ALUA multipath drivers. Users can use the standard multipath driver in their OS or hypervisor and still get optimal performance and resilience.

Ecosystem Integrations

The VAST DataStore was designed for modern applications in modern data centers. That means a data center where resources are allocated and managed by automated orchestration systems using APIs, not storage administrators clicking through GUIs. VAST DataStore has, as discussed above, an API-first design where all management functions are designed into the REST API first, and the GUI consumes the same REST APIs that our customer's platforms and scripts do.

Kubernetes CSI

Containers are the latest, most efficient way to deploy applications, which is why we deploy VAST Servers in containers ourselves. Kubernetes, originally developed at Google and now run by the Cloud Native Computing Foundation, has become the dominant orchestration engine for containers. Kubernetes automates the provisioning and management of microservices-based applications in pods of containers across a cluster of X86 server nodes.

As users deployed more complex and data-intensive applications in containers, the container community added Persistent Volumes to containers and Kubernetes to provide storage for these applications. Kubernetes supports a wide range of file, block, and cloud storage providers for Persistent Volumes using the CSI (Container Storage Interface). In addition to Kubernetes, VAST users running Apache Mesos and Cloud Foundry can also take advantage of storage automation via CSI.

VAST's CSI driver provides an interface between the Kubernetes cluster control plane and a VAST DataStore cluster. This allows the Kubernetes cluster to provision folders in an NFS Export as Permanent Volumes, define the volume's size with a quota, and publish the volume to the appropriate pods.

Following the open and open-source philosophy behind CSI, the VAST CSI driver is open source and available to anyone interested at https://hub.docker.com/r/vastdataorg/csi.

Manilla

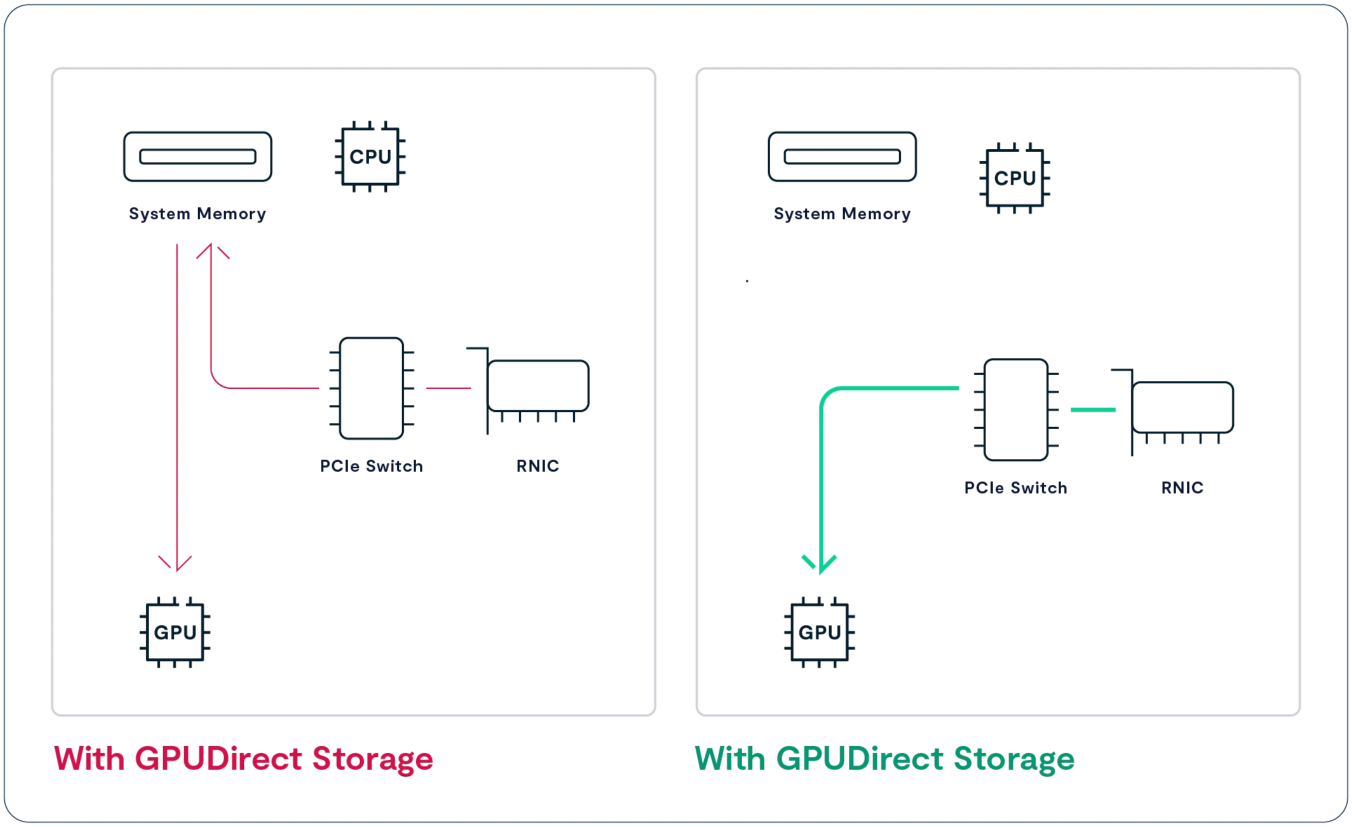
VAST's CSI driver provides a standardized interface a Kubernetes cluster can use to provision Persistent Volumes, in the form of mountable folders, for pods of containers.

VAST's Manilla Plug-in provides a tighter connection between a VAST cluster and the OpenStack open-source cloud platform. In addition to the basics of publishing volumes for VMs, as CSI does for containers, Manilla also automates the creation of NFS Exports and setting the Export's access list of IP Addresses.

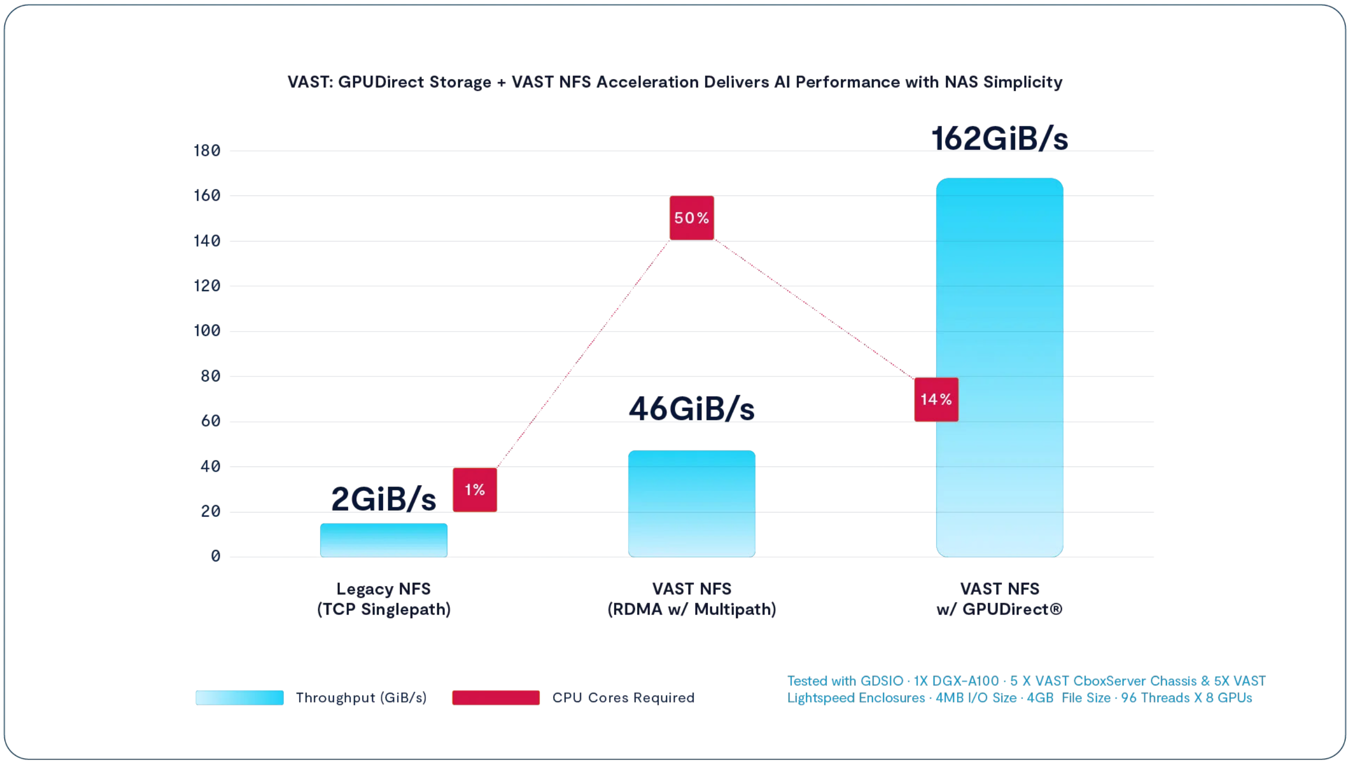
Manilla allows large operators to automate the process of provisioning Kubernetes, or OpenShift clusters, creating Exports private to each cluster, via OpenStack while provisioning volumes to container pods via CSI.

GPU Direct Storage

The GPU servers running artificial intelligence applications process orders of magnitude more data than general-purpose CPUs possibly could. Even with multiple 100 Gbps NICs, the GPUs would spend too much time waiting for data to be copied from a NIC data buffer into CPU memory and only then into the GPU's memory where it could be analyzed.



NVIDIA's GPUDirect Storage provides a direct RDMA path for NFS data from an RNIC into the memory of a GPU, thus bypassing the CPU and the CPU's memory. GPU Direct Storage (GDS) vastly increases the amount of data GPUs can access by eliminating the main memory-to-GPU bandwidth bottleneck of 50 GB/s. Using GDS the server can transfer data into the GPU from multiple RNICs in parallel, boosting total bandwidth to as much as 200 GB/s.



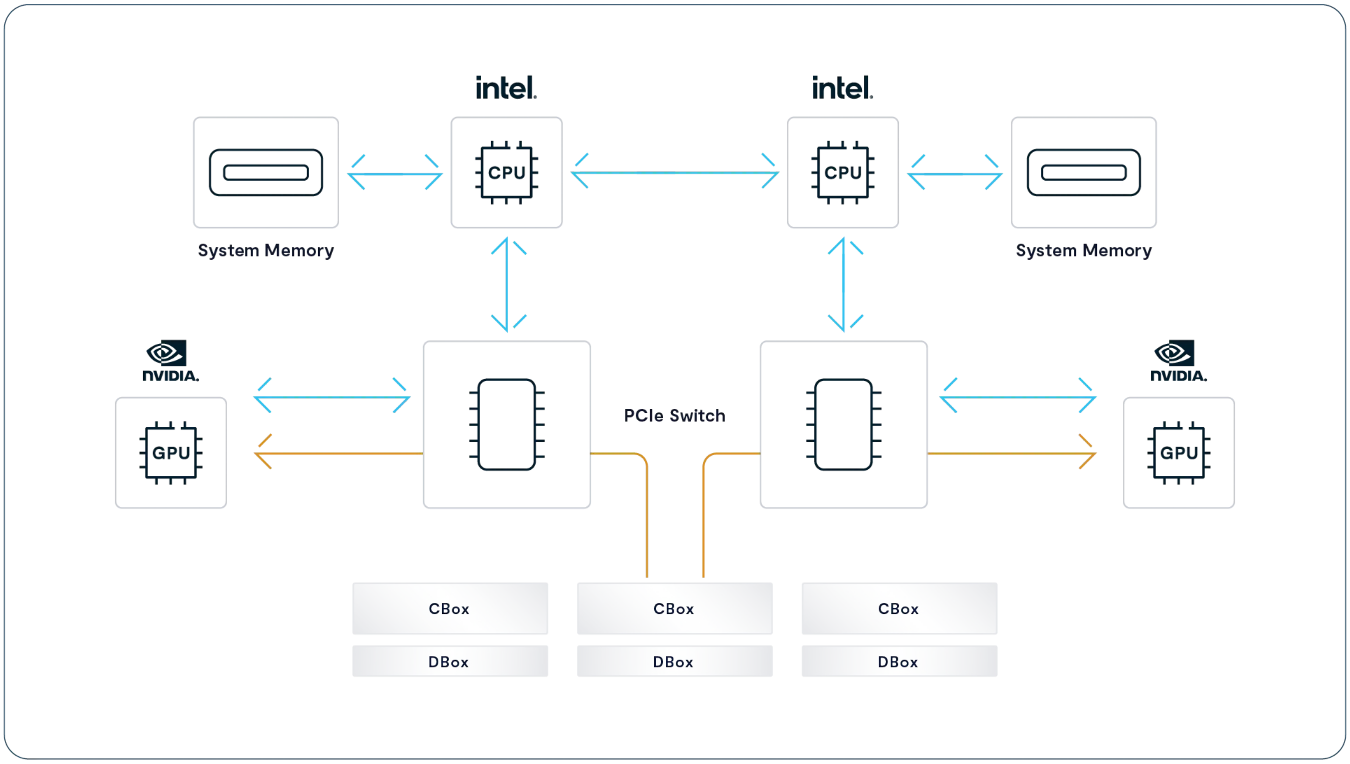
GPU Direct Storage also reduces the overhead of accessing storage. When we tested a VAST Cluster with an NVIDIA DGX-A100 GPU server, as shown in the chart above, the combination of NFS over RDMA and multipath load balancing across the DGX-A100's eight 200Gbps HDR InfiniBand boosted total bandwidth from the NFSoTCP's 2 GiB/s to 46 GiB/s. However, moving all that data through main memory to the GPUs eats up half the DGX-A100's 128 AMD ROME cores.

Switching to GPU Direct Storage not only raised throughput to 162GiB/s but also reduced the amount of CPU required from 50% to 14%.

NUMA-Aware Multipathing

The PCIe slots, like the memory sockets, in a modern server are directly connected to one of the two CPUs. The result is NUMA (Non-Uniform Memory Access) where a process running on CPU-A can access memory directly connected to CPU-A faster than memory connected to CPU-B. Access to remote memory has to cross the bridge between the two CPU sockets and the bandwidth of that channel can become a bottleneck.

GPU Direct Storage accesses perform RDMA transfers from the RNIC in one PCIe slot to the GPU in another slot. The NFS multipath driver is NUMA-aware and will direct data from the VAST cluster to an RNIC on the client that's connected to the same CPU/NUMA node as the GPU that will process the data. This keeps GDS traffic off the CPU-CPU bridge and away from that bottleneck.



Ecosystem Validations

Keeping things simple has always been one of the key design tenets of the VAST DataStore. One affordable tier of flash is simpler than a complex tiered architecture, and standard NAS protocols are simpler than SANs and parallel file systems.

Supporting standard protocols also makes it simple to use VAST as the datastore for a wide range of applications. Even though they use standard protocols, some software vendors certify or validate datastores and VAST has many such validations including:

- VMware vSphere – Validated as an NFS Datastore for vSphere
- Commvault
- Veritas
- Veeam
- Etc., etc.

More to Come

As the abilities of the VAST Data Platform grow with each new release, so do the opportunities for deeper ecosystem integrations. We're working with vendors to use the VAST Catalog to replace the time- and resource-consuming file system walks they now perform. This will allow backup applications, workflow managers, and other data movers to quickly find the files that have changed between snapshots.

NVIDIA DGX H100 SuperPod Validation

NVIDIA's DGX SuperPod is an engineered AI supercomputer using NVIDIA's GPU-optimized DGX servers. The latest generation of SuperPod uses 127 DGX H100 servers each with eight H100 GPUs and eight 400 Gbps InfiniBand ports.

VAST was the first, and as of this writing is the only, NAS solution certified by NVIDIA as a datastore for SuperPod implementations. All the other certified solutions are based on much more complicated parallel file systems. See https://vastdata.com/press-releases/vast-data-achieves-nvidia-dgx-superpod-certification

The VAST DataSpace

Within a single VAST cluster, the VAST AI OS provides a comprehensive platform for storing, processing, analyzing, and, through AI techniques, gleaning new insight from all of an organization's, or if we can be a bit bold, civilization's data. While the DASE architecture allows VAST clusters to scale to thousands of compute nodes and provides the highest levels of reliability, the fact that a single VAST cluster is local to a single data center limits both durability and availability.

The VAST DataSpace extends the AI OS's scope from managing the resources of a single cluster to managing data across multiple clusters in core, edge, and cloud environments worldwide and, if recent press releases are to be believed, into low Earth orbit. The VAST DataSpace protects your data against data center-level failures (they're not really disasters if you're properly prepared) and makes it available to computing resources across your IT estate which empowers the VAST DataEngine to manage your workflows running CPU or storage intensive stages in your data center where the bulk of your data is and GPU stages in the neocloud, where the latest GPUs are available.

When it comes to durability, the only way to make your data more durable than your data center is to spread copies of that data across multiple data centers. Once you have copies of your data in multiple data centers, not even the little Deep Impact/Armageddon scenario of a small meteorite striking your data center can cause you to lose data. The VAST DataSpace enhances durability with a flexible replication engine that efficiently manages copies and retention.

Beyond durability, the VAST DataSpace redefines availability to mean more than the percentage of a year a given system can actually serve data (Over 99.999% for the installed base of VAST clusters BTW) to making the system's data available for different uses, in different locations.

The VAST DataSpace's primary availability tool is the VAST global namespace, and we at VAST have to apologize for redefining the global in global namespace. We've long used global to mean "cluster-wide", describing the single namespace that addresses the petabytes, or exabytes, of data in a VAST cluster as that cluster's global namespace. Given the limitations of physics (Data can't travel faster than the speed of light) and networking technology, the globe that global namespace can effectively serve is practically limited to a single data center or cloud availability zone.

With the VAST DataSpace, we redefine the global in global namespace to mean worldwide, making data, and over time, additional VAST AI OS services, available across multiple VAST clusters, including VAST clusters in the cloud. The VAST global namespace allows each cluster to present a shared folder/bucket, giving servers and users in their local data center full read-write access to the shared folder with all-flash performance while maintaining strict consistency across multiple sites and multiple writers.

Just as the VAST AI OS is more than the software-defined storage solution it evolved from, the VAST DataSpace is more than just a fancy name we attached to standard, or stand-alone features like replication and a global namespace. With the VAST DataSpace, an administrator can define a single set of policies defining the number and location of replicas for a folder, the retention period for snapshots for each, and the additional clusters that should present the folder as if it were local.

VAST Replication

While we call our snapshots immutable, even Indestructible Snapshots (TM applied for, patent application considered, etc.) are vulnerable to events that damage a whole VAST cluster. For protection against fire, flood, earthquake, tornado, hurricane, typhoon, sharknado, or a steam tunnel explosion, you must get data out of the data center. That means you need to replicate that data.

The VAST AI OS supports three replication methods to cover just about any application requirement:

- Synchronous replication sends each write to a protected path to the remote VAST cluster(s) as they are received by the local cluster, only acknowledging writes to the client once data is safely written to both clusters' SSDs.
- Asynchronous replication periodically replicates the changes to a protected path from a source cluster to target clusters, leveraging the VAST DataStore's snapshot mechanisms.

- Write Leases Ensure Consistency
- Backup-to-Object, also leverages VAST snapshots, packaging the changed data from each snapshot into objects and writing those objects to an S3-compatible object store

Like most VAST data services, VAST replication operates on folders in the VAST element store that we call protected paths. Those protected paths can in turn hold any type of element from basic files and objects to block volumes or database tables

Synchronous Replication

When you absolutely, positively need to make sure your applications never lose data, even if there's a smoking crater where one of your data centers used to be, the only solution is synchronous replication. Since synchronous replication solutions ensure that data is written to both clusters before the write is acknowledged to the client, they can guarantee that customers can read back every byte that any of their applications wrote to the system.

From a disaster recovery perspective, synchronous replication can meet a recovery point objective RPO of zero seconds, allowing applications to fail-over to a secondary cluster without losing data.

The unavoidable downside of synchronous replication is the increased latency incurred when writing to multiple clusters. We can minimize that latency to a few hundred microseconds when writing to two clusters that are both physically close to each other and have sufficient network between them, like adjacent data centers in northern Virginia, the New Jersey triangle or cloud provider availability zones.

Spread the two sites further apart and the laws of physics, more specifically that speed limit we call the speed of light, start to rain on our parade. Light travels through fiber optic cables at ~200 KM/ms, a little slower than the 3*10^8 m/s speed of light in a vacuum we all learned in school. That means ~ 2.5ms of latency sending every write from your primary data center in New York to the disaster recovery site in Philadelphia.

Latency isn't the only cost of synchronous replication, there's also the cost of the high bandwidth, low-latency connection between your sites, that 2.5ms between New York and Philly is over a dedicated dark fiber link. Latency will skyrocket if the network gets saturated and your routers start buffering data or worse you try routing your synchronous traffic over the public internet with, or without, a VPN. These bandwidth and latency costs limit synchronous replication to the highest value applications and short (typically < 250 miles) distances.

Synchronous replication has historically been implemented as active-passive or primary-secondary pairs, with all write requests directed to the active/primary member of the pair, which then replicated the writes to the secondary. The primary/secondary relationship could vary by the bucket, folder or block volume being presented with each array active for some targets and the passive secondary partner for others, but for any given resource, there would be a primary and secondary system. In the event of an array failure or other disaster, the system can fail over, promoting the secondary resources to become active for new writes.

Active-Active Synchronous Replication

VAST systems, as shown in the figure above, implement active-active synchronous replication. Both clusters present the same VAST View, and clients can direct writes to that view on either cluster. When a VAST cluster receives a write request to a folder protected by synchronous replication, it forwards the request to the other cluster holding a replica of the folder in real time and only acknowledges the write back to the application once the other clusters have sent their acknowledgements. By the time a client receives the acknowledgement, your data has been safely written to SSDs on both clusters, even better than being written to the mostly safe battery-backed-up DRAM of traditional arrays.

As of this writing (January 2026), the VAST AI OS supports two-way synchronous replication for select file/object protocols. Current plans include three-way replication, block support for NVMe/TCP, automatic failure detection, and transparent failover with synchronous transactions in the VAST DataBase and distributed event brokerage further down the line.

Asynchronous Replication

As we discussed in the section above, when it comes to data protection, synchronous replication is the gold standard, protecting all your data in two locations as soon as it is written, but, as we also saw, that gold-standard protection comes at a significant cost. When you need to protect data beyond the limited distances or with lower network requirements, the solution is to remove synchronicity, allowing the remote replicas to fall behind the primary and be updated asynchronously.

VAST's asynchronous replication leverages the VAST DataStore's snapshot functionality, periodically replicating changed data from snapshots of a folder on the source cluster to one or more target clusters, where the changes are automatically applied. Since the replication of data to the remote targets is performed asynchronously with the incoming write requests, async replication doesn't affect write latency or write performance in general at all.

Async replication also places significantly less demand on the network connection between the source and target(s). First, there's the write consolidation that comes from periodically sending the net changes captured by each snapshot instead of every write as it occurs. In the very simple case of a law firm's working folders, Microsoft Word autosaves every document any of the attorneys are working on to a file every 15 minutes, which would cause a synchronous replication system to send a full copy of every file every 15 minutes. If we instead protect that path with an hourly replication policy, only 1/4th as much data will have to be sent over the network, and we've eliminated bursts that would cause latency spikes.

VAST asynchronous replication supports both one-to-many and many-to-one models. A VAST cluster can replicate any protected path to up to three other clusters, and any cluster can serve as a target for multiple protected paths. Data is transmitted between clusters in compressed form, but each cluster performs data deduplication and similarity reduction independently.

Customers can set replication QoS to prioritize traffic for specific protected paths or targets, aligning replication priority with the criticality of each folder. Additionally, traffic between VAST clusters can be encrypted with mTLS (mutual TLS), which encrypts data and provides mutual authentication for both clusters during the connection.

Conventional asynchronous replication solutions require administrators to either fail over the source and target relationships or clone the replicated folder to make it available on their target arrays. VAST clusters receiving replication of a protected path present the protected path's last fully replicated state, read-only, to prevent data integrity issues. Since the target folder is on an all-flash VAST system, it is immediately available for analysis.

Administrators can perform a graceful failover when both systems can still communicate, and a graceless failovers for when they can't. Graceful failovers turn the protected path on the source read-only, perform a final replication, make the former target read-write, and reverse the replication direction.

Graceless failovers promote a replication target to become the new source for the protected path. Should the source for a protected path go offline, as it would in a disaster, the system administrators can perform a graceless failover and continue operating, with new data written to the cluster that was promoted to be the new source. When the original source cluster is restored to operations and reconnected, the new source cluster will replicate the changes to the protected path while the original source was offline back to the original source cluster. Once the original source cluster is up to date, a graceful failover will return the replication relationships back to their original state.

Minimizing Recovery Point Objective

Most snapshot-based asynchronous replication systems allow users to take and replicate a snapshot as frequently as every 15 minutes. While storage vendors like to call their snapshot frequency the system's Recovery Point Objective (RPO), that logic ignores the time it takes to take and transfer each snapshot. More realistically, you should plan on an RPO of twice the replication frequency. After all, the source system has to transmit all the changed data to each target, and that takes time too. Realistically, a 15-minute replication frequency would allow a user to expect to lose no more than 30 minutes of data, a 30-minute RPO.

While a 30-minute RPO is sufficient for many use cases, other applications require shorter RPOs. The low cost of VAST's snapshots allows a VAST cluster to replicate data every 15 seconds, reducing the minimum RPO to under a minute.

VAST systems also monitor protected paths for RPO compliance, generating alerts when a replication pair on a protected path fails to synchronize within the allowed time across two consecutive replication periods. Administrators can then determine whether the delay was caused by a large influx of new data, insufficient bandwidth, or another factor.

As long as the protection policy for a protected path meets the requirements for both synchronous and asynchronous replication, VAST customers can switch the replication method for a protected path from synchronous to asynchronous, or vice versa.

Simple, Powerful Policies

The VAST AI Operating System uses unified data protection policies to manage both local snapshots of a folder and the replication of its contents. These protection policies allow VAST administrators to define the schedule and retention policies for local snapshots, along with the replication schedule for asynchronous replication, and the retention periods for snapshots on the replication targets as shown in the screenshot below.

Defining a Protection Policy

A typical protection policy might take a local snapshot every 5 minutes, allowing fine-grained local recovery from user errors and other data errors that are detected within the 24 hours all those snapshots are retained. That same policy would replicate the snapshots taken at the top of the hour to the customer's DR site, preserving hourly snapshots at the DR site for 48 hours and daily snapshots there for 30 days.

Backup to Object

While it doesn't rise to the level of philosophers' debates between determinism and free will, or Nils Bohr and Albert Einstein's debates about quantum mechanics, the debate over the distinctions between snapshots and backups has filled thousands of web pages and blog posts. Clearly, both snapshots and backups preserve data against future modification, deletion, and corruption, but snapshots, even replicated snapshots, and backups differ primarily in how the data they hold is most directly accessed.

While you can, and people often do, restore a single file from a VAST snapshot through the ./snapshots folder, snapshots are designed primarily to make recovering, or cloning, the entire folder they hold, quickly presenting the recovered data, or clone, as a shared folder in basically the same place, or at least on the same storage system as the snapshot was stored. Backups, on the other hand, are typically copies of the data placed on lower-cost, lower-performing storage and have to be restored to the original or alternative high-performance storage before the data can be used effectively.

VAST's Backup-to-S3 provides an automated backup mechanism for The VAST DataStore. Like asynchronous replication, Backup-to-S3 extends the data protection provided by VAST snapshots to include replicating snapshot data off-site, in this case to an S3-compatible object store in another of the customer's data centers or the public cloud. Users can control the frequency of replication, down to once every 30 minutes, by time of day.

The system writes the snapshot data to an S3 bucket, compressed, in large objects for efficiency. Once snapshot data is in the cloud, the snapshot's contents are available on the source VAST cluster through the read-only /.vastremote/ policy/snapshot_time/ folder. All a user has to do to restore a file, or a thousand files, is copy the preserved copy from /.vastremote. It's simple and safe enough for end users to restore their own files.

Backup-to-S3 and the /.vastremote folder protect the files on a VAST cluster and make it easy to restore them when the VAST system that created the snapshots is still available. The large objects in the S3 bucket are self-describing. If the source cluster is unavailable, VAST customers can mount the snapshots from any other VAST cluster, including a VAST-on-Cloud cluster.

	Synchronous	Asynchronous	Backup to Object
Minimum Frequency	Real-Time	15 sec	30 Min
Target Type	VAST Cluster	VAST Cluster	Object Store
Minimum RPO	Zero	~30 sec	Depends on data sizeRecovery Type
Recovery Type	Fail-over or restore	Fail-over or restore	Restore
Support	–	–	–
Network Bandwidth Requirement	High	Low to Moderate, depends on frequency	Very Low to moderate, depends on frequency
Application performance impact	Depends on network bandwidth and latency	None	None

Clone Snapshots Anywhere with Global Clones

Like many other modern storage systems, the VAST AI OS can create a clone from any VAST Snapshot. The difference is that, unlike the clones on those other storage systems, VAST Global Clones aren't limited to the same cluster as their source folder or even their source snapshot.

The VAST Element Store's timestamp-based snapshots and write-in-free-space data layout allow VAST clusters to create clones with essentially no overhead. However, local clones aren't the right solution for every problem. Most commonly, users don't want to allow developers write access to their production systems at all, or want to use computing resources in another data center, perhaps closer to the developers.

Global Clones allow VAST customers to create a clone on one VAST cluster from a snapshot on a different, remote VAST cluster. Once the global snapshot is created, the remote cluster can publish its content through one or more VAST Views, providing access to the clone's contents for users and servers in the remote site with all the performance of their VAST cluster.

Let's take Mammoth Pictures as an example: they run their primary data center on their lot in Los Angeles, a smaller data center at their New York office, and rent space in a colocation center in Hollywood. When compute resources become scarce in their LA data center, Mammoth can create a clone of their data in the Hollywood CoLo, to test their rendering algorithms on the latest GPUs in the CoLo, or just to stress-test the website before the release of next summer's blockbuster.

Like VAST's asynchronous replication, Global Clones leverage VAST's snapshot mechanisms move data between a snapshot on a source VAST cluster and a target cluster where the data will be used. The difference between Global Clones and clones made from replicated snapshots is what data is sent between the source and target clusters, and when it's sent.

When the Mammoth initially replicates a daily snapshot to the New York office, the LA cluster sends a full copy of the data in the protected path to the New York cluster. Once that transfer completes, the LA cluster will take daily snapshots and send the differences between each snapshot and its predecessor to New York. The New York cluster holds and protects a full copy of the protected path along with the deltas for any previous snapshots retained in New York.

The HollywoodCoLo is configured to provide Global Clones for the same protected path. Each time the LA cluster creates or deletes a snapshot of the protected path, it sends metadata updates about those snapshots to the HollywoodCoLo cluster. When the studio wants to run a test render in Hollywood, they can select any of the snapshots on the LA cluster and create a clone on the Hollywood cluster, then publish it as a VAST View (Share/Export/Bucket) in Hollywood.

Global clones present the administrators at Mammoth with two options:

- Full or Eager clones – Eager clones are basically a one-time replication of the protected path. As with the initial async replication, the source system will send the full contents of the protected path, as of the selected snapshot, to the destination.
- Cached clones – Where Eager clones transfer the entire contents of the protected path from the source to the destination cluster, cached clones only transfer data from source to destination as the data is accessed at the destination, so the clone on the destination cluster acts as a cache of the original snapshot.

Both Cached and Eager clones are available to users and applications on the remote cluster seconds after creation. The VAST system will prefetch metadata and data using heuristics that learn the application's read patterns to maximize performance as clones populate. As the term clone implies, new writes are stored locally.

By creating a global clone, Mammoth can ensure that the character definitions and backgrounds used by the render testing are available to the GPUs in the remote CoLo without the overhead of a full replication.

Global Access

Taking the VAST Namespace Global

The VAST DataStore provides a fast, reliable repository for structured and unstructured data, presenting a single namespace for the petabytes or exabytes of data in a cluster. We at VAST have referred to this as a global namespace to differentiate our single namespace per cluster model from more conventional solutions that sliced and diced their contents across multiple volumes and file systems, creating many namespaces to manage.

Given the limitations of physics (Data can't, as we've already discussed, travel faster than the speed of light) and networking technology, the globe that global namespace was able to serve was limited to a single data center or data center neighborhood, like all the CoLos and cloud providers clustered in northern Virginia.

Global Access takes the single namespace provided by a VAST cluster and makes it truly global, making an organization's data available with all-flash performance across its full estate of VAST systems worldwide and potentially into low Earth orbit. Like many VAST data services, Global Access operates on folders in the VAST namespace, turning local folders on an organization's VAST clusters into global folders that are available across multiple clusters.

Global Access makes your data available where you need it, whether that's sharing edge-generated data with core servers or shifting inference to the latest GPUs in the cloud.

Once a group of VAST clusters are interconnected as replication peers, any cluster within the group can serve as the origin for a global folder. The origin of a global folder maintains a complete copy of the folder's contents and makes that folder available in real-time to any number of satellite clusters, which then present the folder's contents to their local clients for full read-write access.

Those satellite clusters cache the global folder in their local DataStores, fetching data from the originating cluster on first access and serving subsequent reads from the local cache. All folders containing a Global Path are fully read/write across all clusters that host it. In all cases, strict data consistency is ensured by read and write leasing mechanisms we'll discuss below.

In the diagram above, the customer has created a Global Folder /path) on their origin cluster in London, then made it available to users and, of course, applications on the satellite clusters in Paris and Vienna. Since the VAST global name space is strictly consistent, users and applications at all three locations will see exactly the same data.

Eventually Consistent Isn't Consistent Enough

Most global namespace solutions periodically synchronize data from satellite locations to a central copy, which is frequently stored in a cloud object store. The result of this periodic synchronization is that these solutions are only eventually consistent, which is a less scary way of saying that they are inconsistent. Multiple users accessing the same file simultaneously may see different data. If a customer synchronizes each of their clusters only once an hour, users in one location may see data overwritten by a user in another location several minutes ago.

Even worse, if a user in New York edits the same file as a user in LA within the one-hour replication period, the user in LA's changes would be lost when the updated version from New York is replicated the next time. Eventually consistent namespaces resolve these change collisions with a combination of Last-Writer-Wins semantics and file versioning.

When there are versions of a file created in multiple locations, the system saves the version with the latest time stamp, holds the version(s) from other locations as previous versions of the file, and, in the best case, sends a message to all the users who wrote to that file so they can manually merge their changes.

Some global file solutions, like NAS appliances, address this with file locks; when a user in NY opens a file for writing via a stateful protocol like SMB, the system locks that file. Any other user who tries to open that file with an SMB-aware application like Excel will see, "Howard has that file open – Do you want to open a read-only copy?" That works pretty well for applications like Excel, where users can access a file sequentially (don't start about Google apps and collaboration here that's an application-level thing, Microsoft 365 has it for Excel, too but this is a storage thing that works for applications that are less smart about collaboration than that but Excel is a good example anyway because everyone in corporate America has seen the message above so stop being so pedantic and get off my back) under SMB, but it doesn't work for stateless protocols like NFS 3 and S3, or for applications where users need to read consistent data.

Read and Write Leases Ensure Consistency

Unlike eventually consistent solutions that provide Last Writer Wins consistency, VAST Global Access uses read and write leases to ensure strict consistency. Write leases ensure that only one VAST cluster has permission to write to a given Element (File/object/path) at any given time, ensuring elements are truly consistent and preventing the manual reconciliation of multiple versions, or even worse, lost edits, created by last-writer-wins semantics. While write leases ensure that elements are written coherently, read leases ensure that the data cached at satellite clusters is up to date before it is delivered to clients.

The Mammoth Studios Global Namespace

To better understand how read and write leases work, let's return to Mammoth Pictures, now in production of **IronGiant vs Mechagodzilla** (IGvMG),the highly anticipated crossover event.

The VAST cluster on the Mammoth lot in Hollywood holds the master copies of all of Mammoth's digital assets, including the working folders for IGvMG and assets from previous installments in the series, including CGI characters and settings. Those assets need to be available to the Mammoth render farm at its LA colocation site, and to GPUs that Mammoth rents in the cloud for specialized processing. The executives in New York also need access not to the raw assets, but instead to the "dailies" from live-action and motion capture, and to the nightly output of the render farm.

The admins at Mammoth connect their four clusters (Lot, CoLo, NY, and Cloud) as replication partners, establishing the mTLS authenticated and encrypted channels over which VAST Global Access will transfer data. They then publish the working asset folders, including the dailies folders, using the cluster on the studio lot as the origin for these folders.

The origin of a global folder holds a full copy of the global folder's contents. Mammoth makes the studio lot cluster the origin for all their assets, in part, to facilitate spooling archival copies of their assets to tapes that are sent to "The SaltMine" for apocalyptic protection. In addition to holding the full copy of the global folder, the origin cluster also serves as the lease broker issuing, tracking, and revoking read and write leases.

Each of the other three clusters acts as a satellite for some of the global folders with the cloud and CoLo clusters caching the working and asset folders, and the New York cluster caching dailies for the executives.

When a satellite cluster receives a read request for an element in a global folder, it acquires a read lease for the element and fetches the requested data from the origin. It then sends the data to the client that requested it and writes the data to the local cluster's cache.

Unlike many legacy global namespaces, VAST clusters don't fetch the entire element on first read; VAST clusters analyze the patterns of requests to elements intelligently prefetching when they detect sequential access or seemingly random accesses to hot-spots within an element. That means that the VAST cluster in the cloud will only fetch the assets that are actually used by the specialized render processes running on their cloud GPU servers.

Read leases are valid for some number of hours, and as long as a satellite holds a valid read lease, it serves subsequent requests from the local cache. Multiple satellite clusters can hold read leases on the same elements simultaneously, and clusters will renew read leases when the element they address is read after expiration. Read leases manage cache validity across the distributed cache in the global namespace.

Write leases manage where, and to some extent, how writes are processed on elements in global folders. When any cluster, including the origin cluster, receives a request to write to a global folder, it requests a write lease from the origin cluster.If no other cluster holds a write lease to the element, the write lease will be issued to the requesting cluster.

When the write lease is approved, messages are sent to all clusters holding read leases on the element, revoking those leases and informing those clusters which of their peers now holds the write lease and will therefore manage writes for that element. Once a write lease is issued, the cluster can process writes to the leased element.

When clients on a satellite cluster send requests to re-read data that's cached, but not covered by a read lease, the satellite cluster will validate the data in its cache with the write lease holder, and refresh the cache for data that has changed, before returning the data to the client.

How writes are processed has evolved since the VAST global namespace was first released. Through VAST AI OS 5.4, the origin cluster held all the active write leases. When a client wrote to a global folder on a satellite cluster, the satellite synchronously proxied the request to the origin, waiting for an acknowledgement from the origin before acknowledging the write to the client.

Since the writes are proxied to the origin, if a client at the satellite reads the newly written data, the satellite will have to fetch the data from the origin, with the second and subsequent reads satisfied from the local cache.

Since writes are processed synchronously at the origin, there can be multiple simultaneous writers to a single file; write order integrity is preserved, so the resulting file will be just as coherent as if the multiple writers were writing to a file in a local file system.

Write Leases Go Portable

More recent versions of the VAST AI OS have made write leases portable across the clusters, providing global access to a global folder. Rather than issuing a write lease to the origin cluster, and synchronously proxying all writes to the origin the VAST DataSpace now issues the write lease to the cluster a client is writing to.

When a cluster receives a write request, it acquires a write lease from the origin, invalidating all outstanding read leases, just as it always has. The difference is that when a client writes to the cluster that now proudly holds the write lease, rather than proxying the write to the origin, the cluster writes the data to its local cache, acknowledges the write to the client, and asynchronously sends the writes back to the origin.

Managing Global Access

In addition to choosing which clusters in your network are the origins and satellites for the various global folders you're sharing, the VAST DataSpace gives you additional control over how your global namespace behaves and uses system resources.

Each satellite cluster maintains a single local cache that serves all the global folders that the cluster presents. When the cluster holds the write lease on the elements being written, the cache operates as a write-back cache, storing newly written data locally and asynchronously sending it to the origin. When the cluster processing a write request doesn't have the write lease, as in the original version of the VAST global namespace, it operates as a write-around cache, with writes going directly to the origin and not updating the cache.

Administrators can set limits on cache size and/or the number of cached files or allow the cache to grow to consume all available space on the cluster. Either way, when the cache fills, stale data is evicted to make room for new data, using a combination of least-recently-used and least-frequently-use techniques.

Administrators can also control the read lease duration and prefetch behavior for each global folder, setting read leases to be longer for folders where applications create new output folders, such as image rendering, to minimize overhead, and shorter lease durations for multi-writer use cases.

By default, a satellite cluster fetches data or metadata only when a client requests it, which can introduce significant delays in some workflows. At Mammoth Pictures, the live-action footage is uploaded from cameras at the end of each day's shooting, a process that takes 2-4 hours. If the servers in their LA CoLo didn't start transcoding and assembling the raw footage into dailies for the executives until all the transfers were complete, those servers would sit idle for that whole period.

To eliminate that delay, Mammoth has configured the VAST satellite in their CoLo to prefetch the data for the camera footage folder, allowing the servers to start their nightly processing as the first files are replicated from the studio lot. Since the executives in New York only view a small fraction of the dailies, but have to be able to choose what they want to watch quickly, the New York cluster is configured to prefetch metadata and only fetch the actual video files when someone wants to view it.

Combining Replication and Global Access

Replication and Global Access address the two primary data management problems for hybrid and distributed workflows. The most important function of any IT system is to ensure the durability of the data it stores. Replication provides cross-site durability by distributing copies of data across multiple VAST clusters.

Coming in a close second in importance to durability is availability. You can sleep well at night knowing your data is safely stored in a vault somewhere, but you can't turn data in the vault into insight, business value, or, let's get right down to it, money. To do that, your data has to be available to whatever computing resources can slice, dice, and julienne value from your data, and the global namespace that Global Access does just that.

Even better, when an organization designates a VAST cluster to be both a satellite in the global namespace and a replication target for one or more folders, replication and Global Access work together to optimize capacity and network usage. When a client reads data from a combined cluster, and the latest version of the requested data has already been replicated, the cluster will use the replicated data.

When the element being read has changed since the last replication of the protected path, the combination cluster will fetch the latest version. This fetch forces the replication source or write lease holder to put the fetched data at the head of the replication queue. When data is fetched this way, the combination cluster's metadata is updated to include the data in both the replication target snapshot and the Global Access cache. Since the VAST DataStore supports deduplication, these logical inclusions are simply metadata structure updates.

Compare this to legacy systems, where replication and the global namespace were developed over many years by different development teams. These systems might require data to all be synced to a central object store or store replicated data and global namespace contents in different data structures without any synchronization, transferring, and even storing data multiple times for a single access.

DataSpace Futures

Today's VAST DataSpace addresses many of the durability and availability requirements for object and file oriented workflows, but there's more to come. The DataSpace vision goes well beyond a protected global namespace for files and objects. We'll also be taking the VAST DataSpace up the application stack, along with the rest of the VAST AI OS, allowing the VAST DataEngine to run workflow stages on compute resources close to the data or to make the data available on VAST clusters close to the right compute resources.

The first set of enhancements come to the DataSpace services that support storage level access and disaster recovery. These include enhanced support for applications that write to the same file from multiple locations with read and write leases on byte ranges within an element and support for access protocol byte range locking (NFS v4, SMB, NLM). The VAST DataSpace will add additional support for block volumes including synchronous replication eventually including an external witness, and metro-area automated failover.

Ultimately we'll integrate data durability and availability through a single policy. This policy defines the number of full copies, the clusters that must, the clusters that can hold those copies with required RPOs, and the satellites that will make the folder available. The system will then place data where it can most cost effectively meet the policy's RPO. Satellites will fetch data from the "closest" copy of the current version of each element ensuring data is always available even when whole clusters go offline.

Even further up the stack are a distributed event broker allowing producers to write into a topic locally and have that entry read by a consumer in another data center where the VAST Data Engine wants action taken. We've also started thinking about how the VAST DataSpace would support a user submitting an SQL query to one VAST cluster and getting results from across all of their organization's VAST clusters.

We're not making any promises about when any of these enhancements will appear, that depends on customer demand.

VAST on Cloud

VAST on Cloud extends the EBox model from clusters of industry-standard servers running in someone's data center to VAST clusters in your favorite public cloud. VAST on Cloud clusters are full-fledged VAST clusters that can provide the full range of VAST AI OS services in the cloud, from high-performance file services to the VAST DataEngine executing specialized functions on cloud provider GPU servers, accessing data in the VAST DataBase across the VAST DataSpace's global namespace.

The first challenge in adapting the VAST AI OS to the cloud was adapting to the different hardware environment. VAST AI OS is a software solution, but as we've discussed elsewhere in this whitepaper, VAST AI OS was also designed to take full advantage of everything the latest hardware can do. Unfortunately, public cloud providers don't supply SCM SSDs and don't support RDMA transfers between instances, so we had to adapt the VAST AI OS software to eke out the best each cloud provider's environment can provide.

The other big change required for VAST on Cloud was automating cluster deployment and integrating that automation into the cloud provider's management stack and APIs. This allows VAST customers to install VAST clusters into their VPCs (Virtual Private Clouds) from their cloud provider's marketplace.

Building Cloud Clusters

VAST on Cloud instances use the local NVMe SSDs provided with the cloud instance to store both data and metadata. Since VoC instances are virtual EBoxes, data is striped across the instances in a cluster, just as it is in an EBox cluster.

The remaining problem is that cloud instances fail and reboot more often than EBoxes. When a cloud instance reboots, it does so on a different host and doesn't have access to the NVMe SSDs it was just using. When the instance is restarted, it will load on a different host with blank SSDs.

While a VAST on Cloud cluster stripes data across its members so the cluster can rebuild from one or even two VAST on Cloud instances going offline, VAST on Cloud clusters can't provide the 11 9s of data durability that on-premises VAST clusters do with NVMe SSDs alone.

VAST customers have several ways to address this limited durability. The simplest is to simply not write irreplaceable data to a limited durability cluster. Make the cloud cluster a satellite in the global namespace in synchronous write-proxy mode, where all data is written directly to the origin, or have the cloud servers direct their output to a more persistent location.

Making VAST on Cloud Persistent

Standard VoV instances allow cloud computing resources access to the VAST AI OS 's services and data across the VAST DataSpace. This makes them a good solution for read-intensive applications. When your favorite cloud provider touts a new computer vision algorithm, you can use a VAST cloud

While VAST on Cloud clusters provide the scalability and flexibility that all VAST clusters do, that scalability only goes in one direction: the minimum cluster size is eight VoC instances, just like the minimum cluster size for EBox clusters.

While VoC instances can be much smaller than physical EBoxes (~9TB vs 50TB+ for EBoxes), an eight-node cluster might be excessive for developers testing their new workflows in the VAST Data Engine or for a proof of concept. These developers can run a single-node VAST-on-Cloud cluster in the public cloud of their choice and verify that their new workflows work before scaling up to a cluster with the capacity and compute power needed for production.

Single instance clusters have no protection against instance failures; if the cloud instance crashes, the contents of the cluster are lost. That's fine for a developer who saves their work periodically, but not so good for production workloads. VAST only recommends single-instance clusters for development and testing use cases where the cluster's data can be replaced.

Instance to give it access to your photo repository for testing. Still, many applications need a higher level of persistence.

These include both the usual applications that generate data with unique values and those that reuse datasets periodically, with limited updates. When running the latter type of application, the VAST Data Engine can spin down the cloud instance for hours or days and spin it up, complete with data, when the application needs to run again. No waiting for the cache to warm; only updates that the application actually uses get fetched.

The VAST on Cloud solution for data persistence through multiple instances or even zonal failures is for the VoC cluster to write data to the cloud provider's object store as the data is migrated from the cluster's write buffer to the main data repository on the local NVMe SSDs. If a cluster running in this persistence mode reboots due to a zone failure, it can resume from the last persistent checkpoint.

Customers looking for the ultimate in in-cloud durability can run two VAST clusters in different cloud provider zones and replicate them, potentially even synchronously.

VAST on Cloud Futures

We at VAST have never been ones to rest on our laurels, and like any software, VAST on Cloud is never finished. Over the coming months, we have many enhancements planned for VAST on Cloud, some of the most interesting are:

- VAST on Cloud object caching clusters will use the local NVMe SSDs for metadata and as a cache, storing the bulk of data in the cloud provider's object store.
- Cluster elasticity will allow VAST clusters to shrink and grow on demand to manage cloud compute instance costs

A customer with periodic cloud bursting requirements could spin up a VAST on a Cloud cluster, have it populate with the nightly or monthly job's data, and then spin the cluster down to a minimal level for ad hoc access. When the burst job repeats, more nodes spin up, expand the cache, and load data from the object store only to spin down again when the job completes.

The VAST DataBase

Given the vertically integrated nature of The VAST AI Operating system, The VAST DataBase is more a convenient way to discuss the features of the VAST AI OS that access and manage structured data stored in tables than a software module in the traditional sense. Architecturally, one could think of VAST SQL, the VAST dialect of the Structured Query Language, as just another protocol like SMB or S3, but in this section, we'll take a more conventional view and talk about the VAST DataBase like it were an actual layer of compute resources and abstraction as in more conventional data platforms.

The VAST database uniquely combines an exabyte-scale namespace for the natural data types (Images, video, LIDAR, genomes, and other rich, real-world data sources) and a tabular database to hold the catalog of expanding metadata about those objects that's generated as data works its way through the deep learning pipeline, and that information is inferred.

Storing Data in Rows or Columns

Maintaining tabular data organized row by row is efficient for OLTP (Online Transaction Processing) systems, whose primary goal is to process transactions as quickly as possible. A typical transaction involves locating a handful of records, updating some of them, and inserting, or appending, a new record or two. A row-organized database can perform these actions with a relatively small number of highly random I/Os.

Row-organized databases are less efficient at processing queries that collect data from many rows in a table. A query like "Show the average selling price for Blue Widgets by month for 2022" would have to read the entire sales transaction record (row) for every Blue Widget sale in 2022. If the table didn't have indexes on the date and item number columns, the query would have to read the entire table.

Since most queries are interested in data from only a small number of fields, processing them requires reading much less data when stored column by column than when stored row by row. Most current data analytics platforms transform data from a row-based organization to a columnar organization, where data is stored column by column rather than row by row.

The open-source Apache ORC and Apache Parquet file formats used by many Data Lakes are columnar data formats that store column values across the members of a row group. Tables in these Data Lakes are stored as hierarchical folders of Parquet files, partitioned by one or two "key fields," most often by date, to create monthly folders. Today's Data Lakes were designed by cloud companies like Google and Facebook, partly because they were among the first to collect petabytes of data they knew had aggregate value. Cloud companies design their solutions to run in the cloud, and the only affordable storage in the cloud is object storage. As a result, file formats like ORC and Parquet, and the higher-level table formats like Apache Iceberg and Databricks' Delta Lake were designed around the limitations of cloud object storage services like Amazon S3.

Since object storage services like S3 hold immutable objects, and there's relatively high latency (10s to 100s of ms), the designers of today's analytics systems designed those systems to minimize the number of I/Os they perform and to make those I/Os as sequential as possible. This begins with storing data column by column in large Parquet files and partitioning those files by column values. In addition, Parquet files also store the minimum and maximum values in each file's footer.

When the system processes a query for all orders with more than 500 Blue Widgets, it first reads the footers of all Parquet files that contain the Quantity column. Since 500 is a large order, the maximum value in the footer of many files will be less than 500, which allows the query/ database engine to avoid reading the actual data in those files.

Improved Reducibility

Column stores have one other significant advantage: the data in column stores compresses better than data stored row-by-row. The explanation is very simple: they store similar data together. The values for any column in a table are constrained by that column's type and real-world constraints. A simple example is that all the values of the zip code column in any table will be five-digit integers.

This data similarity within a column, which is somewhat different from the similarity between data chunks used by similarity reduction, and both are different from the vector similarity that powers AI agents, makes it easier to compress with both conventional dictionary and Huffman encoding techniques. Some file formats, including Parquet, include compression via Snappy or GZIP. VAST systems have the additional advantage of data-aware compression algorithms that further optimize this type of data, using techniques such as run-length and delta encoding to store consecutive column values as differences.

VAST's Columnar DataStore

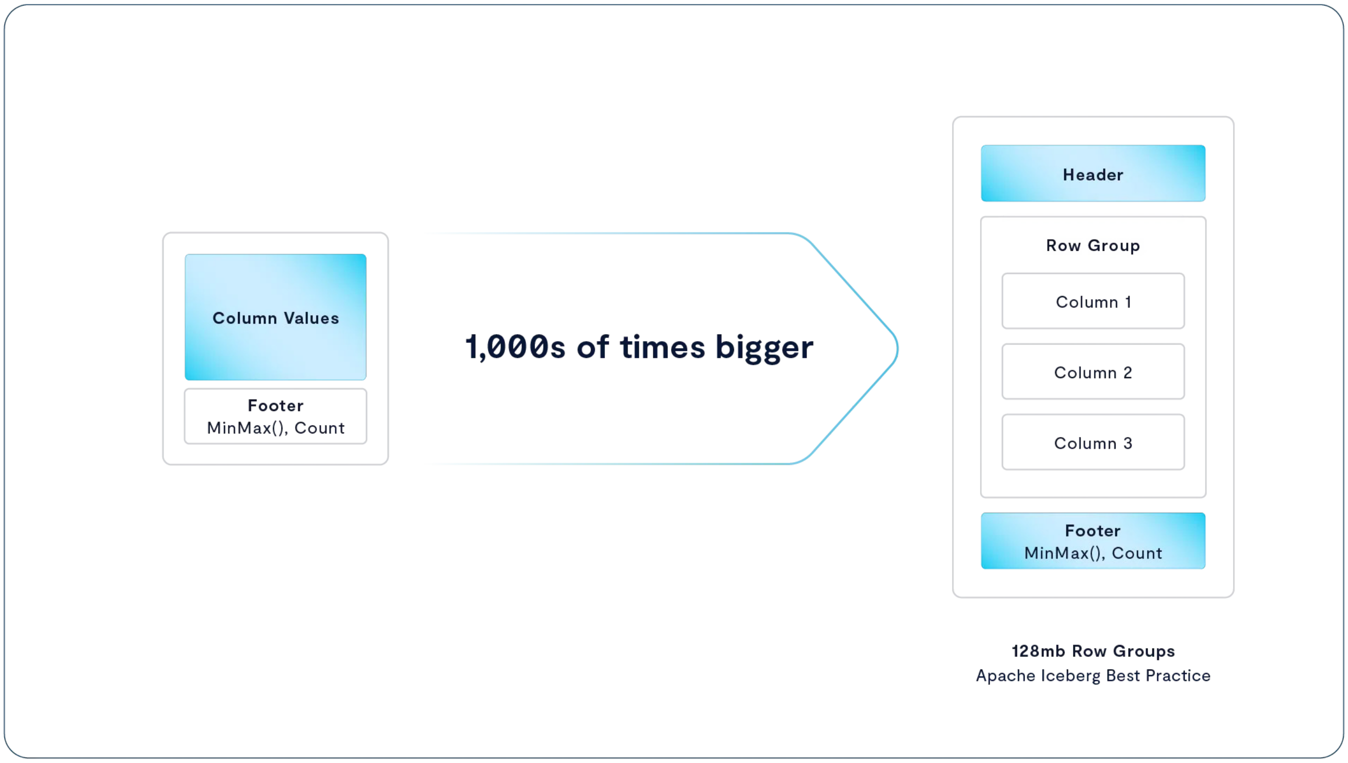
As discussed in the introduction to VAST Table Elements above, the VAST DataStore stores tables as a class of data Elements, the hierarchical equivalent to the Elements that hold files/objects. Table Elements are stored in folders within the VAST cluster's namespace and appear as files to client tools such as ls or Windows Explorer.

The data in those Table Elements is stored in a tabular form inspired by the Parquet table format, but is much finer-grained and managed by the same metadata that stores the Table within The VAST Element Store.

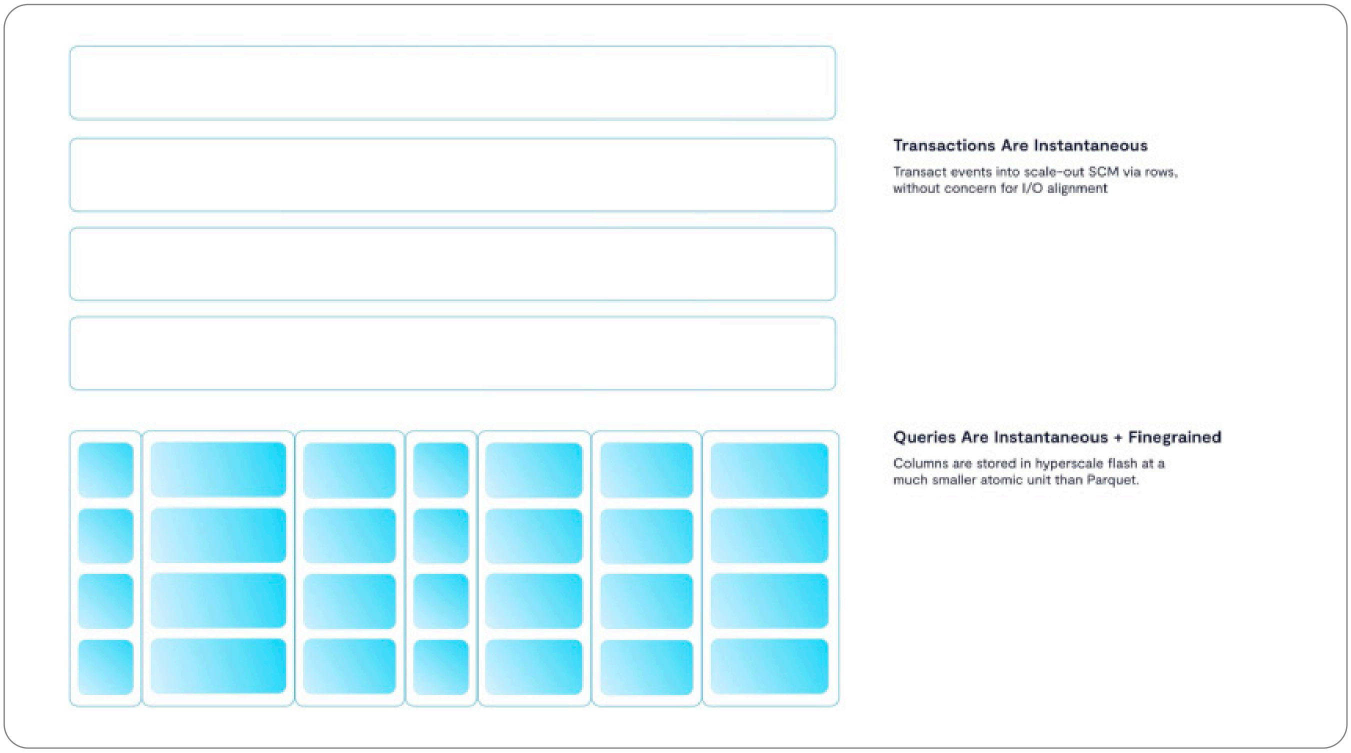
When data is written to the VAST DataBase, primarily through VAST SQL INSERT operation, that data is written to the VAST cluster row-by- row. The CNode receiving that INSERT writes the data into the SCM write buffers on two SCM SSDs just as it would for an NFS write or S3 PUT. This operation also makes the appropriate updates to the DataStore metadata.

As we saw in Transactionally Consistent, those metadata updates are fully ACI. Since the metadata updates are made after the data is written to persistent SCM SSDs, those same transactional semantics ensure that updates to VAST Tables are also fully consistent and ACID, like a relational database management system.

As we saw in the section on Table Elements, the VAST Data Platform stores tables in a tabular form inspired by the Parquet table format but much finer-grained and managed by the same metadata that stores the Table within The VAST Element Store. Each data chunk in the physical layer of The VAST DataStore holds the values for a column for a relatively small set of rows in the table.



SQL was designed before columnar data stores; this isn't an issue on queries since they specify columns, but INSERT and UPDATE operations will write data to the datastore row by row. In a VAST system, that data is immediately written to the SCM write buffer, where it is stored essentially as written, row by row. When the write buffer accumulates enough data to form a row group, those write buffers are flagged for migration to hyperscale flash., The same data migration process that organizes files into the 16-128 KB data chunks managed by the VAST DataStore's physical layer transforms that data into columnar data chunks before writing to QLC.



Transactions Are Instantaneous

Transact events into scale-out SCM via rows, without concern for I/O alignment

Queries Are Instantaneous + Finegrained

Columns are stored in hyperscale flash at a much smaller atomic unit than Parquet.

Unlike Data Lakes and other datastores with immutable backends, the VAST Database allows users to INSERT and UPDATE just like a relational database should. The VAST Datastore is, as it has always been, a write-in-free space datastore, so the replacements created by an UPDATE will create new columnar data chunks with metadata linking to the new chunks in the background. Since VAST tables can contain billions of rows, each table can replace 100s of files across a complex partitioned hierarchy.

Even though the VAST Database stores data in blocks by columns, it can deliver the writeability usually provided by row databases with small column chunks and a storage back end that performs small random I/O and large sequential reads. Using small column blocks also allows queries to run against a table via multiple dimensions like date and location without partitioning the data by each, creating multiple copies. The penalty for reading a 32 KB column block, because it contains 1 value in range, is much smaller than when 1 value between a footer's min and max meant reading a 1GB Parquet file.

The VAST Database isn't just a replacement for the Data Lakes full of Parquet and ORC files. It's a full database supporting relational/transactional applications with indexes, UPDATES, INSERTs, and other functions alien to immutable data stores. While the latest table formats allow users to rename columns or save a point-in-time snapshot on a table stored in an immutable object store, they perform their magic with layers of metadata redirection that interconnect multiple files and add both I/O and delays as queries follow complex chains to locate the data they need through multiple layers of abstraction.

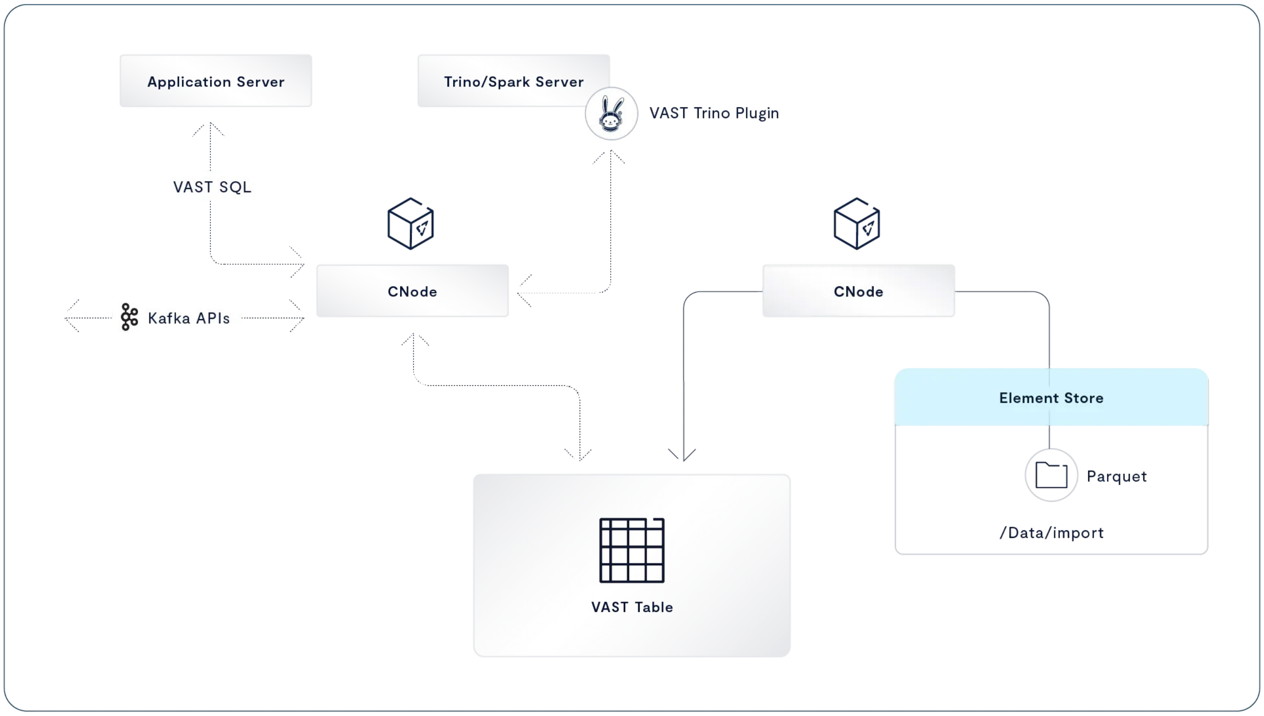
Since the VAST Database has a single, integrated set of metadata for its tables, from schema directly to data blocks, there's no need for additional external metadata layers in additional files or metabase databases. Table snapshots are handled with the same zero-footprint snaptime mechanism that provides file system snapshots, and since the VAST DataStore is a fully writable, ACID, repository operations like renames, column add, and even column deletions are actually performed, not simulated by layers of metadata manifests.

Accessing the VAST DataBase

The short answer to how you access VAST tables is that you do so just like any other tabular database using SQL (Structured Query Language), the lingua franca of tabular data. The most direct way to access VAST DataBase tables is through the VAST native query engine. The VAST native query engine is fully integrated into the DASE architecture, running across a pool of CNodes and directly accessing the shared metadata of the VAST Element Store. It includes powerful features like vector search in VAST SQL, which is VAST's dialect of Structured Query Language.

While the native query engine provides the most direct, and therefore usually fastest way to access VAST tables via SQL, it isn't the only way. SQL is a complex language, and unfortunately, ANSI SQL has always been one of those standards where even compliant solutions are different enough that many applications must be dialect-specific, which means that VAST customers will have queries and complex workflows that leverage every quirk of their existing query engine. Customers may also want to use their favorite query engine to combine data from the VAST DataBase and other sources.

Our solution is to provide plug-ins for popular SQL query engines and data platforms that enable them to read and write to VAST tables and to push functions, such as filtering, down to the VAST DataBase. This allows the VAST Data Platform to return the results of the pushed-down predicate, relieving the query engine cluster of the work. Today, we offer plugins for Apache Spark, including the DataBricks platform, Vertica, Dremio, and Trino, including Snowflake



Database File Ingest

Users can, of course, create new tables directly via SQL, and many applications will simply use SQL as a data definition language to create their databases; but running several PB of analytics data through your compute cluster is a very expensive way to copy a petabyte or two of existing data from Parquet files into VAST Database tables, especially if those Parquet files are already stored in a VAST S3 bucket holding a data lake.

The easy way to ingest that data is to simply define a folder/bucket as containing a specified table through the VAST API or GUI. The system will then ingest any Parquet table, or Iceberg data warehouse stored in that folder into VAST Database tables. Once all the data is stored as a single table element, you can delete the original files and folders along with all the work of managing file and folder sizes to accommodate storage limitations.

Both Analytical and Transactional

While some smaller-scale database solutions have provided decent OLAP (Online Analytics Processing) query processing performance while maintaining the ACID consistency required to reliably process transactions, those solutions can't handle more than a few TB of data. Large-scale analytics solutions either can't process transactions at all or can only process them at a modest rate against a single table.

The VAST DataBase is structured to support high performance for both transactional updates and analytical queries. We've already looked at how the VAST DataBase stores newly written database rows to SCM SSDs and transfigures that data into small, columnar (16-128 KB average) data chunks to optimize query performance.

Those small column chunks are how the VAST DataBase can also process transactions without inordinate overhead. When an UPDATE modifies some existing data in a table, that data can be in one of two places. If the data being updated is the quantity on hand for the latest Taylor Swift album, the day it drops, that UPDATE is updating a record that's still in the SCM write buffer. In that case, the system records the new data and updates the table metadata to point to the latest version.

If the data being updated has already been written to column chunks on hyperscale flash SSDs, the system reads the chunk that contains the data and writes a new version of that chunk to the write buffer. A 32 KB column chunk is roughly the same size as a database page in a relational database system, allowing the VAST platform to run applications that require OLTP-level transactions, even across multiple tables.

Merging Content and Context with the VAST Catalog

Having a relational database for context (Metadata, indexes, Etc) and an organized file/object store for content is a pattern that appears across a wide range of applications, not just Data Lakes and Artificial Intelligence Feature Stores. The dual datastore model drives PACS systems for medical imaging, Media Access Management systems (MAMs), enterprise document management systems, and countless other data management systems that track some portion of the context of media from documents and spreadsheets to JPEG images and video of Hank Aaron's 715th home run in relational databases.

Frankly, the dual datastore model works decently for Feature Stores and Data Lakes because the users of those systems only access their contents through tools that can also manage, or at least access, the metadata store. When we move into use cases where users have more direct access to the contents, the fragility of having two loosely coupled data stores raises its ugly head.

Suppose users mount the content repository, which they must do to use desktop applications such as video or photo editors. In that case, those users can delete, rename, move, or edit files without updating the metadata store. At best, the system scans the content repository overnight. It sends a report listing broken links and files with more recent modification dates, so an administrator can fix the broken links and make the two datasets consistent again. In the worst case, no one discovers broken links until years later, when no one remembers where the files were moved to.

Enter The Catalog

At its core, the VAST Catalog is simply the most obvious use of the VAST Database. The VAST Catalog makes the VAST Element Store metadata accessible not just as a nested series of files (folders) holding metadata about other files, accessed folder by folder via NFS or SMB, but also as a VAST Table. With the catalog, finding all files that have changed since the last midnight scan is a simple SQL query.

```
SELECT path,name, size for MTime>midnight
```

The VAST system will process the query and, a few seconds later, return a list of files that have changed since midnight and their sizes.

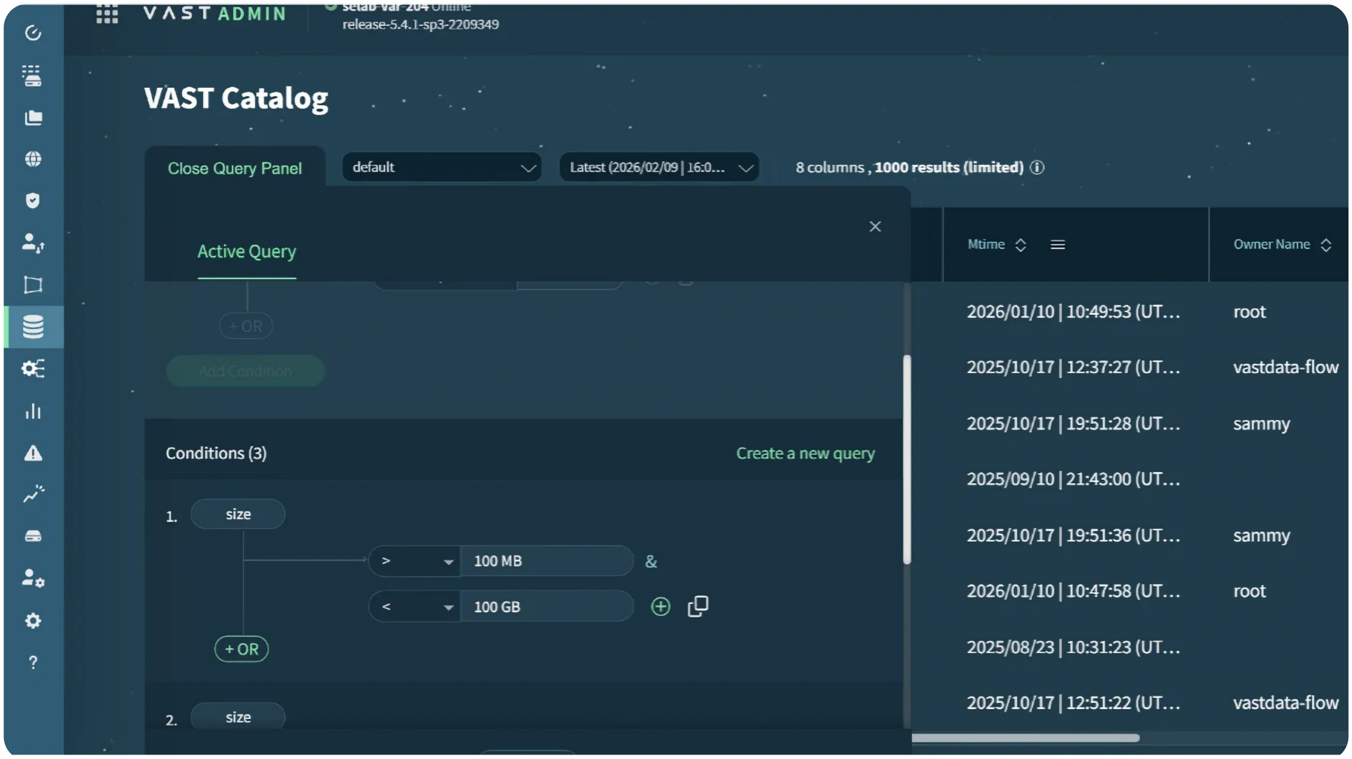
Without the catalog, our administrator friend, or more accurately, their workstation, will have to search the filesystem, loading each folder's contents to both see if there are any new files and, more significantly, if there are any additional folders to search. These file system walks are notoriously slow, taking days, if not weeks, on file systems with billions of files because file system metadata is organized to provide fast access to a file, rather than for search.

This problem first became critical when users tried to run their nightly backups, and the file system scans started taking so long that incremental backups never completed in their window because they never finished searching the file system.

The VAST Element Store generates the catalog table from a periodic snapshot of the cluster's namespace; like all VAST snapshots, the minimum granularity is 15 seconds, but generating a new catalog table every 15 seconds for a namespace of billions of elements is probably more work than it's worth. More typically, the Catalog is updated every 15 minutes to an hour.

```
Insert .CSV segment of 5-8 files and selected fields
Handle,ParentPath,Name,Size,Ctime,MTime,ACL
```

Since the Catalog is a VAST Table, like any other VAST Table, users can access it through VAST SQL or the VAST Spark and Trino plug-ins. However, since not everyone who needs access to the Catalog is a data scientist, we've also provided tools that are a little more accessible to system administrators.



VAST Catalog

8 columns , **1000 results (limited)**

Conditions (3)

1. size
   - > 100 MB &
   - < 100 GB

2. size

Mtime	Owner Name
2026/01/10 | 10:49:53 (UT...	root
2025/10/17 | 12:37:27 (UT...	vastdata-flow
2025/10/17 | 19:51:28 (UT...	sammy
2025/09/10 | 21:43:00 (UT...	
2025/10/17 | 19:51:36 (UT...	sammy
2026/01/10 | 10:47:58 (UT...	root
2025/08/23 | 10:31:23 (UT...	
2025/10/17 | 12:51:22 (UT...	vastdata-flow

As shown in the screenshot above, administrators can build queries in the VAST GUI and receive the results directly or in JSON.

With the Element Store metadata available as a table, VAST customers can easily query that table for everything from reporting capacity utilization by group or project to the "Find me the big files no one's accessed in a year 'cause we're running out of space" query.

Extensive and Extendable

As nice as having the Element Store metadata accessible as a table is, the VAST catalog is more than that because the Catalog metadata is more extensive than the basic metadata POSIX file systems provide and almost infinitely extensible. In the first case, since The VAST Element Store is both a file system and an object store, the VAST Catalog includes all the metadata defined by NFS, SMB, or S3, including S3 object metadata and extended tags.

Customers can use simple queries against the Catalog table to verify that their secure data's ACLs limit access as intended and to report consumption for bill-back or simply to shame users into cleaning up their scratch space. But things get really interesting when VAST customers start combining the basic metadata provided by the VAST Element Store with other context sources about their files.

Other context sources include metadata embedded in media and document files, and metadata created at each stage of your workflow, from columns for actors and directors attached to video files to full-text indexes and DLP system sensitivity evaluations.

Before VAST, customers would collect this data in a relational database through a Media Asset Management or Enterprise Document Management system. That database would connect the context it held to the content in the associated file system, using links, typically URLs or UNC paths, to the file's location. If files are renamed, copied, or moved outside the MAM or EDM, those become broken links.

Inseparable Context

The problem is that old-school content management systems were really two independent systems loosely coupled, not by some immutable attribute of each file it manages, but by the file's location as a URL or UNC. When VAST customers add metadata columns or tables, to the VAST Catalog, they store data in tables keyed off the unique Element handle assigned to each Element in the VAST Element store. Keying off this unique handle means that any extended metadata is tightly coupled to the Element itself, not to the Element's location, and will remain attached to the Element even when the file is moved from one folder to another or cloned via a server-side copy method like ODX.

Catalog Snapshots Add The Dimension of Time

The VAST Datastore uses snapshots to create a consistent point-in-time view of the VAST Element Store to protect the VAST Catalog's tables from, but that's not where the connection between the VAST Catalog and snapshots ends. Catalog tables, like snapshots, have a retention period, and a query can specify which snapshot(s) the query should run against.

Since each snapshot captures the state of the system's metadata at a specific point in time, VAST customers can run queries that compare tables across multiple snapshots to reveal how the system's data has changed over time. The most obvious of these queries is the change-list of new, modified, and deleted files that would feed a backup application, eliminating the backup app's file system walk to discover the data that's changed since the last backup.

The VAST DataEngine

The VAST DataEngine is where computation really joins the VAST AI Operating System. This section presents an overview of the capabilities the VAST DataEngine will deliver.

The VAST DataEngine provides the execution and orchestration intelligence to manage and execute the function pipelines that let data scientists and deep learning practitioners scrape, transform, train, infer, and otherwise derive value from the files/objects and tables the VAST AI OS holds without worrying about where, how, or possibly when, those functions are executed. The VAST DataEngine automatically optimizes these pipelines to minimize cost, execution time, and/or system utilization to deliver a serverless execution environment across multiple on-premises and cloud locations.

At the most basic level, the VAST DataEngine is analogous to an operating system's supervisor or the supervisor/state machine at the core of every storage system, including the VAST AI OS. Storage systems are constantly reacting to event triggers like SSDs failing and the arrival of user requests. These events trigger functions like erasure-code rebuilds and data reduction. The supervisor assigns the threads of the rebuild or a query across its available resources.

The big difference is that the VAST DataEngine takes a much broader view, managing not only the resources of a single server or even a single cluster in a single location but all the resources you make available to it. The VAST DataEngine turns all the computing resources across all of your data centers, and those public cloud resources you give it access to, into an integrated thinking machine that takes data in and delivers valuable insights. Let's take a look at how.

Event Triggers and Functions

The heart of the VAST DataEngine are two new concepts so central to the DataEngine's operation that we've defined new Element types in the VAST Element Store for them: event triggers and functions. Triggers define the conditions that require an action of some kind, and functions define the actions to be taken.

Since triggers and functions are Elements, they can be created and edited as files in the appropriate formats. The VAST DataEngine includes a Python SDK for building functions, further demonstrating the plasticity of data, metadata, and code in the VAST AI OS.

Event Triggers

As we've stated, event triggers define the conditions that should cause the system to act. Those conditions are based not just on the event itself but can also include filters and rules that define which objects should trigger functions, how those functions should be executed, and where they should output their results. Some of the basic conditions VAST customers can define as event triggers are:

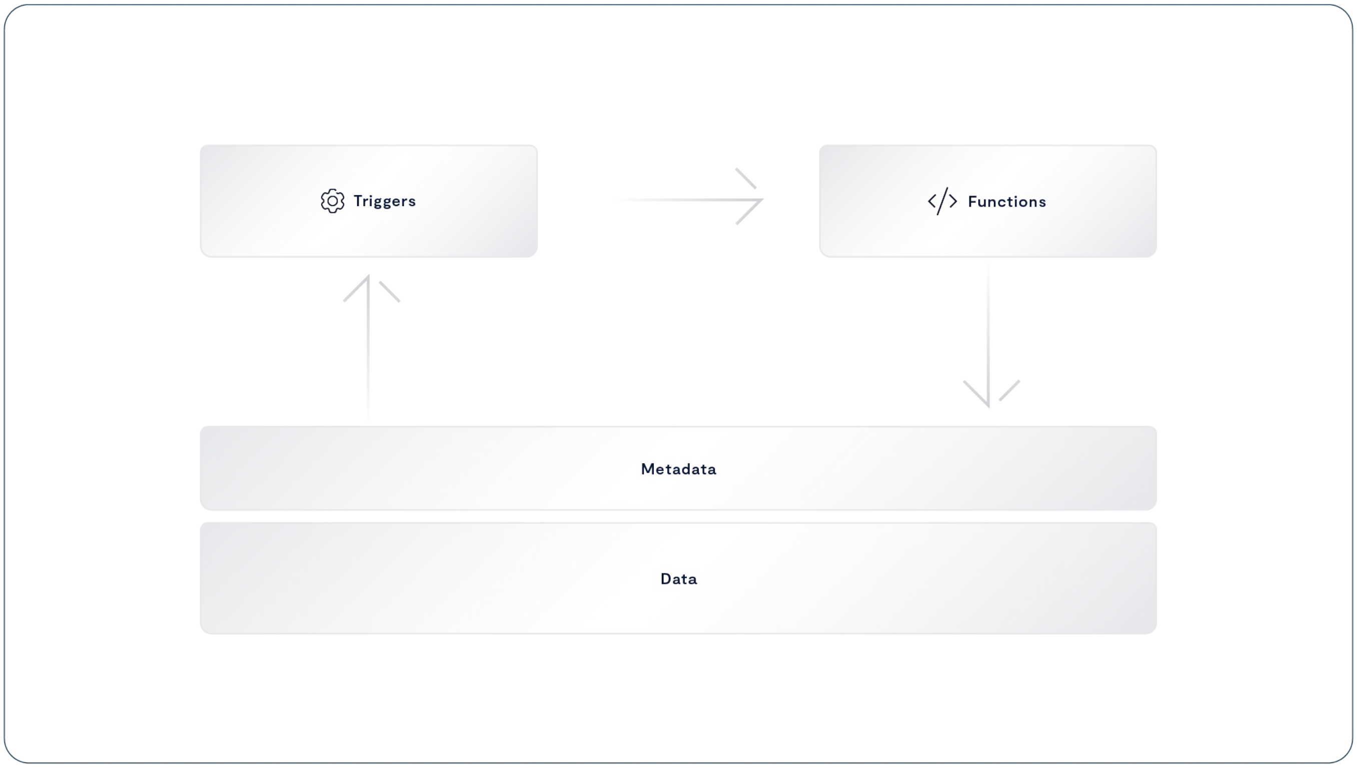
- CRUD (Create, Read, Update, Delete) events in the VAST Element Store. For example, any file of the type JPG created in the /Photos folder will trigger the built-in metadata scraping function.
- Row CRUD in VAST Tables. A new row in the specified table. Since the VAST DataEngine stores Kafka topics as VAST Tables, new entries to a Kafka topic can also serve as a trigger
- Any of the 1000s of analytics counters in the VAST cluster
- The completion of a previous function
- A periodic schedule

Event trigger rules provide fine-grained control over whether the trigger activates per event, based on folders in the namespace, catalog metadata, rates, and other filters. The event trigger rules also allow users to set execution preferences (run this function fastest vs. cheapest). When conditions fit all the event trigger parameters, the system will execute the specified function(s).

Functions

Functions are the definitions of the software functions and microservices that the system can perform when the conditions of an event trigger are met. Each Function Element defines the execution requirements of a given function, say a GPU-powered inference engine performing facial recognition. This would include hardware and location dependencies for functions orchestrated by the VAST DataEngine, or simply the execution method for functions performed by the cloud or other services outside the VAST DataEngine's compute environment.

Since the global workflow optimization engine, described below, will choose which function to perform a task based on cost, the VAST DataEngine also keeps information on the cost, CPU resources, and execution time of each function each time it's run, and uses these factors to run functions in the optimal location each time.

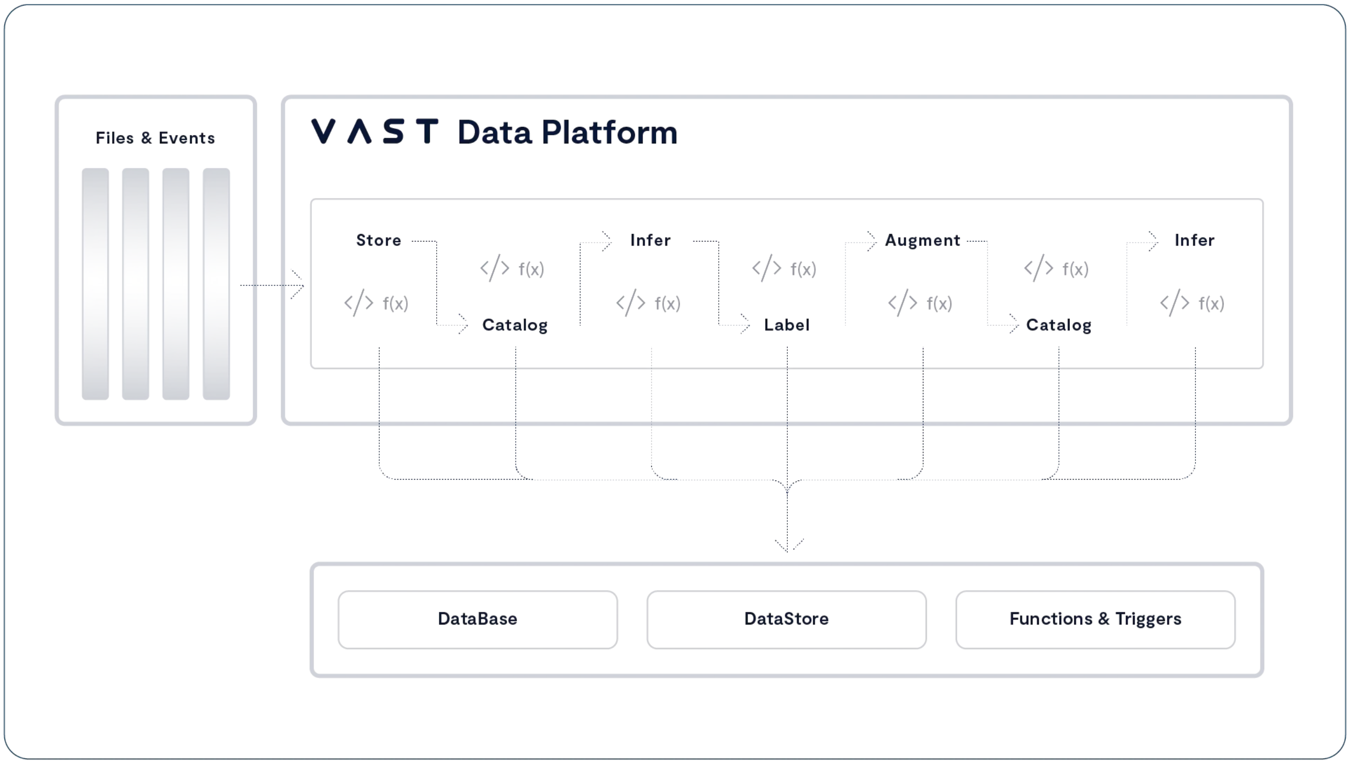


The VAST Computing Environment

The VAST Computing Environment on which the VAST DataEngine operates consists of two major components. First is a global workload optimizer that responds to event triggers, figures out the best place to run each instance of each function, and calls the services that perform those functions, from scraping metadata to matching millions of genome segments. The second is the execution and orchestration engine that manages the containers that deliver the function's services.

The Global Workflow Optimization Engine

The global workflow optimization engine goes beyond a mere task or job scheduler, providing toolkits that let you assemble the event triggers and functions we discussed above into declarative workflows with rules that define the causes, effects, and interactions between all those system Elements.



A workflow may start by scraping the embedded metadata out of the incoming files into the VAST Catalog. An event trigger may then detect that a subset of those files are, based on that photographs taken in Las Vegas over the New Year's weekend and cause an inexpensive inference function to run on those files and identify a smaller subset of images that include the image of a red car, those files are subjected to more extensive processing to ultimately return the license plate number for a hit-and-run.

Since Event Triggers can include rates, counts, and time, the system can process both event-driven stages that trigger functions immediately on each event and data-driven stages that allow data to batch to some degree before spinning up the function to process them.

Workloads can define multiple options for any stage in the workflow. A workflow could, for example, provide three options for transcoding video, one function that uses on-premises GPU resources, a second function that uses CPU resources but consumes more resources and runs slower, and a third that uses a public cloud transcoding service, The optimization engine will choose which function to run, and where to run it based on the conditions at the time and the service optimizations, primarily execution time SLA and costs, you selected.

Process Functions Worldwide

The VAST DataEngine maintains an internal database of the resources it manages from the various VAST clusters in the VAST DataSpace, and the folders they share to the computing resources (CPUs, GPUs, Memory, Etc) available to run functions in each data center, the networks connecting the various locations, and much more. When planning a workflow, the optimization engine will orchestrate functions to optimize each workflow stage for performance or cost as the user dictates.

Circumventing Data Gravity

Dave McCrory coined the term data gravity to describe how massive datasets develop an attraction for applications the same way massive objects have a gravitational attraction for each other. Once an organization develops a massive dataset, that dataset attracts applications that create additional data, which in turn attract additional applications. The organization becomes trapped in its headquarters data center or selected public cloud region.

Just as the pull of real-world gravity is a function of the mass of the objects attracting each other, data gravity is a function of the mass, or more properly, the size of the overall dataset. The more data and the more types of data stored in one place, the more attractive that place becomes as a home to run the applications that will process that data.

The VAST DataEngine circumvents data gravity by shielding applications from the mass of the entire collected dataset, leaving each function only attracted by the data it accesses. The concept of data gravity presupposes that, since which data each application will access is unpredictable, even applications that only access small subsets of the entire dataset are drawn in by the gravity of the dataset as a whole.

Since the VAST DataSpace only transfers the data that's being accessed and keeps that data local in the very large cache of a VAST cluster, the VAST DataEngine learns where data is and how to move the execution of functions to where the data they access has been accessed before.

The VAST DataEngine performs a best fit against the rules defined in a workflow to:

- Perform functions close to a replica of the data they access when sufficient resources are available to meet the workflow SLA.
- Move only active data to compute using the VAST DataSpace
  - When specialized compute like GPUs is required/fastest/cheapest
  - When compute resources are unbalanced, moving functions to locations with idle resources can reduce the total execution time and increase the total work
  - Cloud bursting to provide elastic computing capacity
  - To leverage low spot pricing for long-lived workflows

The VAST Execution Environment

The VAST Execution Environment provides the computing infrastructure for the services provided by the VAST AI OS and user workloads running within the VAST DataEngine. Rather than deploying separate infrastructure silos for each workload type, VAST AI OS consolidates compute resources into a unified pool managed by the VAST DataEngine.

Traditional data platform architectures typically require organizations to deploy and maintain multiple, independent systems:

- Object stores for Parquet files and Iceberg table metadata
- Dedicated compute clusters for database management and query engines (Spark, Trino, etc.), often with specialized hardware requirements like local SSDs for query cache

- Container orchestration platforms for training, inference, transformation, and other user tasks
- Workflow orchestration systems, including schedulers, message brokers, and other "glue" functions

Furthermore, each system must be sized for its own peak utilization and maintained separately. Since peak loads across these systems rarely overlap, this leads to underutilization across the datacenter, and the specialized requirements of some systems (for example, Trino nodes with local SSDs) prevent flexible resource sharing even when excess capacity exists.

The VAST Execution Environment takes a different approach. DataEngine acts as the orchestration layer for containers, managing both VAST-provided services and user-provided containers within the same shared infrastructure. This includes VAST containers that implement VAST AI OS capabilities (such as SQL query processing, rebuilding erasure codes after SSD failures, and running DataEngine's optimization engine itself) alongside user workloads (model training, inference, and custom applications).

This unified orchestration allows the VAST Execution Environment to optimize a broader range of workload types across a single pool of resources. However, this single pool of resources does not need to be homogeneous; workloads may require high memory, specific GPU models and libraries, or direct access to specialized networks. The VAST Execution Environment accounts for these constraints when orchestrating containers and workloads.

By consolidating infrastructure that would traditionally require separate clusters for storage services, query engines, container orchestration, and workflow management, the VAST Execution Environment reduces operational overhead while improving resource utilization across diverse workload types. In turn, this provides a lower overall cost per unit of execution.

A Kafka-Compatible Event Broker

Event-driven workflows that respond to internal operations like updates or file writes require a message broker that can keep up with the speed at which data can be changed. Apache Kafka has become the de facto standard for high-throughput event streaming, but using Kafka in a conventional data platform would require yet another dedicated cluster and the added operational burdens that the VAST DataEngine aims to eliminate. To address this, the VAST DataEngine provides the VAST Event Broker: a Kafka-compatible messaging interface as a native capability, consolidating event streaming into the same unified infrastructure that handles storage, compute, and orchestration.

The VAST Event Broker is a Kafka-compatible API endpoint that allows applications and functions to produce and consume messages using standard Kafka clients. Rather than deploying separate message broker infrastructure though, the VAST Event Broker uses the VAST DataBase to store topics, making event streams immediately queryable alongside other data without requiring the adoption of a new streaming API or deployment of additional infrastructure.

Unified Stream and Tabular Storage

In the VAST Event Broker, each Kafka topic is implemented as a table in the VAST DataBase. As Kafka producers send messages to the Event Broker through its Kafka API, they are transactionally inserted into the topic's DataBase table as rows. This table representation preserves the Kafka message structure: each row contains the message's key and value without interpretation, allowing consumers to process messages according to their own schema expectations.

Because topics are native DataBase tables, they can be queried using the VAST DataBase's SQL interfaces immediately after they are ingested, eliminating the need to extract event data into a separate analytics system or maintain separate storage systems for messages and queries. In addition, data for a single topic is stored in a single DataBase table, allowing queries to operate across all partitions with a single simple select. Partition keys are accessible as a column in each topic's table if partition information is useful, but partitions can simply be ignored when performing analytics on data streams as well.

Shared-Everything Partitions

Traditional Kafka achieves scale by partitioning topics across multiple brokers. Each partition maintains its own message order independently, and resilience is provided by replicating messages across multiple brokers. This triples the amount of storage required to store messages and results in ongoing operational overhead from managing replica health, rebalancing partitions, and handling broker lifecycle events such as failures and upgrades. Because the VAST Event Broker is built on VAST's disaggregated shared-everything (DASE) architecture, it has none of these pitfalls.

The VAST Event Broker decouples partition storage from physical servers and allows brokers to be stateless. While partition leaders are mapped to individual CNodes to maintain ordering guarantees, the underlying DataBase table is accessible from any CNode. This architecture enables seamless partition leader migration via floating VIPs without requiring log replication during failover, enabling seamless partition leader migration. Additionally, because the Event Broker leverages VAST's shared storage rather than replicating data across brokers, topics can achieve strong ingest performance even with few partitions. Consumers using the Kafka Consumer API still parallelize by partition as expected, so topics should still be partitioned appropriately when applications will consume through the standard Kafka protocol. However, applications using DataBase's SQL API can query across all partitions simultaneously, enabling different consumption patterns beyond Kafka's partition-bound model.

As a proof point of Event Broker's scalability, testing of an early release of the VAST Event Broker demonstrated 99% scaling efficiency when scaling from 3 brokers to 88 brokers and from 128 to thousands of partitions. Not only did the VAST Event Broker demonstrate 6x higher message throughput per broker when compared to Apache Kafka running on identical hardware, but the underlying topic consumed a fraction of the storage footprint. Because topics are stored as DataBase tables, Kafka replicas are simply ignored; data protection is inherited by the VAST Element Store's efficient locally decodable erasure codes, reducing the overheads from 66% to less than 3% at scale. Furthermore, streaming data also benefits from VAST's similarity-based data reduction when streams contain significant amounts of degenerate data.

Other features of the VAST DataStore are also available to the VAST Event Broker. For example, Kafka performance can be by enforcing QoS limits on the VAST DataBase table that makes up the topic. By modeling topics as DataBase tables, the Event Broker topics can be managed procedurally just as any other element within the VAST Element Store.

Large Payloads and Reference Integrity

Kafka is designed for small messages that are typically 1 MiB or less, so when events contain rich media or other large payloads, convention approaches require either chunking data across multiple messages or storing the payload in a separate system and including a reference pointer (such as a URI) to that payload in the message. This reference-based design introduces consistency challenges though; if the message broker and external storage systems diverge (e.g., a bucket endpoint changes), references break and consumers are unable to access the data they need.

Because the VAST AI OS unifies event streaming and persistent storage in a single system, messages can safely reference objects in the DataStore. Producers can write large payloads into DataStore through the same S3 endpoint that exposes the Event Broker API. Since the Event Broker endpoint and S3 endpoint are identical, it is safe to embed payload URIs into Kafka messages because both the message and the referenced object are managed within the same namespace. This eliminates broken reference problems while keeping message sizes small, broker throughput high, and payloads durably stored.

Developing Pipelines

Triggers and functions combine into pipelines, enabling complex event-driven workflows. These pipelines encode the logic that turns events into actions, and the VAST DataEngine provides two ways for users to develop and deploy these pipelines: code-first and low-code.

The low-code approach uses a visual builder where users drag triggers and functions onto a canvas and connect them with edges to define event flow. When a function completes, the DataEngine runtime automatically routes its output to downstream functions based on the pipeline's connectivity. Functions simply return structured data, and DataEngine uses the pipeline configuration to determine what happens next.

The code-first approach allows developers to create pipelines programmatically using the VAST DataEngine Python SDK. Both approaches produce the same underlying pipeline representation which defines triggers, functions, and their connectivity using a YAML-based custom resource definition. This allows teams to mix visual design for rapid prototyping with configuration-as-code for production deployment.

Function Development

Functions are standard Python modules with two entry points: An initialization function that runs when the function is deployed, and a handler that is executed whenever an event triggers the function. The handler is given two objects by the DataEngine runtime environment when it runs: a context object that provides logging, tracing, and access to secrets, and an event object that contains metadata describing the triggered event and any payload data.

Because DataEngine provides context to functions, functions themselves do not need to implement boilerplate configuration. Loggers, distributed tracing, and secret stores are all accessible within the function, and logs and traces are automatically correlated across the pipeline to provide end-to-end visibility without requiring developers to explicitly instrument each function. Functions can enrich this telemetry though; the context provided by DataEngine allows functions to add custom attributes to trace spans as execution progresses so that, for example, model names or processing stages can be tagged in the relevant pipeline stages.

Functions communicate through events, which are managed by the DataEngine runtime. When a function returns, DataEngine generates an event and routes it to the next stage of the pipeline. This event-driven model means functions do not need to know what stage comes next; they process their inputs, generate outputs, and rely on the DataEngine to handle delivery, retries, and error handling. The same function can be triggered by S3 object arrivals, scheduled timers, external message queues, or upstream function completions without changing its implementation, simplifying the process of composing data processing pipelines from simple, reusable components.

Observability and Operations

Because the DataEngine runtime automatically instruments every function call, administrators receive detailed timing data and execution traces for every pipeline run. Trace spans show which functions ran as well as the chain of events that triggered them, and developers can add augment functions with custom span attributes to simplify the auditability of pipelines and their resulting data. These traces are stored in the VAST DataBase, making them queryable through any of the DataBase's APIs as well, allowing anyone with the correct privileges to answer questions such as “which embedding model was used in this pipeline run?” or “how many video segments were processed?”

These observability capabilities extend to error handling as well; when a function fails within a pipeline, the runtime routes the failure to a dedicated dead letter topic. As described previously, this topic is also a VAST DataBase table, allowing errors to be both viewed in the DataEngine UI or procedurally queried offline. By storing traces, metrics, and errors as queryable data rather than unstructured logs, DataEngine makes operational intelligence accessible through the same tools and interfaces used for application data.

Connecting Data and Context to Action

The VAST DataEngine provides all the components necessary to make data an active participant in the orchestration of AI workflows. Action can be taken in response to changes in data, and just as DataStore and DataBase allow data to scale without limits, DataEngine allows compute to scale without limits. Because both computation and data are orchestrated under the same roof in the VAST AI OS, the interoperation of both always maintain a mutually consistent, interpretable, and auditable view of identity, access, and locality that does not compromise on performance or scale. This is the foundation of the VAST AI Operating System.

Broadly, the VAST AI Operating System has been designed to combine data with context to take action, delivering on the promise that AI will transform the way we work. Although “transforming data into action” can take innumerable forms, they all generally follow the same flow:

1. **New data is observed and stored.** These observations could come from users interacting with a web service or from complex sensor networks deployed across smart cities, and the frequency can range from sub-millisecond samples to extremely rare events.

2. **New data is contextualized with respect to past data.** The scope of this contextual data could be limited to the very recent past, or it could span millennia of simulated data, and it could be anywhere on the spectrum from fully structured to fully unstructured.

3. **Action is taken after reasoning over the contextualized data.** This could be as simple as immediately notifying a human to performing a complex, agentic workflow that decides and executes best next steps.

While VAST is not in the business of developing domain-specific solutions for users of DataEngine, we have built several frameworks atop DataEngine that address the three foundational steps in AI-driven transformation.

1. **VAST SyncEngine** ingests and synchronizes data from external repositories into VAST, providing scale, resiliency, and end-to-end observability so datasets stay current without bespoke scripts or scheduled copy jobs.

2. **VAST InsightEngine** enables real-time enrichment of data, automatically extracting structure and meaning as new data lands and maintaining a queryable index so content is immediately usable for AI applications.

3. **VAST AgentEngine** operationalizes AI agents by providing an integrated deployment and runtime layer, governed access to tools and data, and deep observability so agentic applications can run reliably at scale and resume long-running tasks without restarting from scratch.

These Engines are all independent and free-standing capabilities within the VAST AI Operating System, but they also demonstrate different ways in which DataStore, DataBase, and DataEngine can work together to enable AI applications that are deployable and maintainable at scale, because data, indices, and execution share consistent governance and observability. In the following sections, we will explain what these Engines do and how they build on the capabilities of the VAST AI OS.

SyncEngine

Infrastructure deployments are rarely fresh starts. Data already exists, it is already in use, and it carries identity, permissions, and operational expectations with it. Migrating to a new platform is less about copying bytes and more about preserving access and continuity, validating integrity, and executing cutovers with minimal disruption.

AI amplifies this operational burden by bringing larger working sets, continuous ingest of both new and intermediate data, and greater demand to maintain availability. By the time a migration to a platform like the VAST AI OS is required, ad-hoc manual copies, rsync scripts, and hand-maintained manifests often become brittle, hard to validate, and expensive to operate at scale.

To address this, VAST developed the VAST SyncEngine, a service that deploys within the VAST DataEngine to index and synchronize data between external sources and a VAST cluster.

At its core, the VAST SyncEngine contains two parts:

1. **SyncEngine Control Plane:** a highly scalable, parallel, fault-tolerant tasking engine

2. **SyncEngine Workers:** processes that are distributed across physical servers dispatched by the control plane

SyncEngine orchestrates tasks that either discover data or copy data from external storage systems. Each task is scoped to a unit of work, such as examining a single directory or prefix, and tasks recursively generate more tasks to traverse namespaces.

When indexing, SyncEngine uses this capability to build and maintain durable indices of external file systems as a single VAST DataBase view. As data on external sources are modified, SyncEngine can periodically retrieve these changes and re-synchronize only those objects that have changed as well. As with any table in the VAST DataBase, this SyncEngine index scales to hundreds of trillions of objects, and query performance scales linearly with CNodes. Combined with its ability to connect to multiple external sources at once, SyncEngine is a powerful tool for cataloging and finding files, objects, and their associated metadata across an entire organization regardless of whether those data are resident in a VAST system or not.

SyncEngine also uses this scalable search capability to orchestrate tasks that continuously copy data from any of the external sources into a VAST DataStore view, establishing a replica of the external system within a VAST cluster. As data is synchronized into the VAST DataStore by SyncEngine, it emits all of the same triggers that a manual write or data copy would. As a result, DataEngine can then trigger enrichment pipelines, allowing SyncEngine to serve as the first step in a fully automated data ingestion pipeline upstream of RAG, ETL, or other workflows within the AI model lifecycle.

How it works

Being built on the VAST DataEngine, SyncEngine inherits the scalability, robustness, and fault tolerance of the underlying VAST AI OS: its index can scale out to hundreds of trillions of objects, and its query performance scales linearly. This allows SyncEngine's functionality to remain relatively simple in the two functions it performs: indexing external namespaces and orchestrating data copy tasks.

To index external systems, SyncEngine performs a breadth-first parallel tree walk that is parallel and distributed. A task is assigned to index the root of the external system's namespace, and it builds batches of files, objects, directories, and prefixes that are to be inserted into the SyncEngine index. As subdirectories and object prefixes delineators are encountered, subtasks are enqueued to descend into those directories and repeat the indexing process.

To copy data from external systems into a DataStore view, SyncEngine uses this same indexing process to create batches of data to copy and enqueues them as data copy tasks. Batch size can be configured to not exceed a specific number of files or number of bytes to accommodate the performance of different network and external storage systems.

Both indexing and copy tasks are executed by SyncEngine workers, which are containers running on either externally managed servers or within DataEngine's runtime environment. The role of SyncEngine workers is to simply pull tasks off of the work queue and execute them:

- Indexing tasks access an assigned directory or prefix within the external system's namespace and create batches of data to be copied. These batches are then queued as copy tasks. As it encounters subdirectories, it also recursively queues additional indexing tasks for those subdirectories.

- Copy tasks receive a batch of files, objects, directories, and prefixes and either copies them into the configured destination in VAST DataStore or inserts them into the SyncEngine index DataBase endpoint.

Adding more workers increases the concurrency of task execution, and the overall speed of this process is limited only by the resources available to workers or the limits of the external data source or destination. Fortunately, the QOS capabilities available in DataStore can also be applied to the destination view when copying data, allowing the performance of a SyncEngine migration to be controlled.

How it really works

SyncEngine consists of the following components:

- **The Control Plane**, a service that presents the SyncEngine REST API.

- **Workers** are containers that pull work from the control plane and execute tasks.

- **Connectors** specify source and destination endpoints.

- **Migrations** define a plan that maps connectors to specific source and destination paths within those connectors.
- **Sync Jobs** are individual, stateful executions of a migration.
- **Labels** are a unique, arbitrary name that map one or more workers to a single migration. Workers cannot have more than one label.

SyncEngine's connector interface is extensible and allows it to build an index across file systems, object buckets, and a growing list of cloud services such as Confluence and Google Drive. In addition, connectors define the types of migrations that SyncEngine can orchestrate:

- **File-to-file synchronization** uses the SyncEngine file connector, and copy tasks transform batches into individual invocations of rsync.
- **Object-to-object synchronization** uses the S3 connector or Azure Blob connector as the source and the S3 connector as the destination. Copy tasks use a copy tool that is aware of object versions, ACLs, and system metadata.
- **Cloud service-to-S3 synchronization** uses the Confluence connector, Google Drive connector, and others to copy data into VAST DataStore object buckets. This type of migration is typically used as the first step of a VAST DataEngine pipeline such as that implemented by InsightEngine.
- **SyncEngine Indexing** uses the VAST DataBase connector as the destination.

The process of synchronizing data with SyncEngine amounts to creating a migration and defining all of its components, then executing a job of that migration. SyncEngine jobs can be executed manually, or they can be scheduled to automatically execute periodically.

While a sync job is running, it can be observed either through a built-in dashboard or directly through SyncEngine's REST API. In both cases, SyncEngine provides complete visibility into jobs' progress and history, errors and traces, worker utilization, and the throughput of workers and the entire job.

InsightEngine

One of the most significant opportunities afforded by AI is accelerating the transformation of data into actionable insight. However, realizing this opportunity requires more than a well-trained model. Models must be able to query and interpret data that may be changing rapidly and reflects the current state of operations, not an old snapshot from when the model was last trained.

Retrieval-Augmented Generation (RAG) has emerged as the primary technique for ensuring AI models operate on current data with the broadest possible context. It works by maintaining a semantic index of an organization's data that allows models to retrieve relevant information at query time. When a user asks a question or an agent needs context to complete a task, the system searches this semantic index for the most relevant data, retrieves it from the underlying storage system, and includes that context along with the query that is sent to the model.

Implementing RAG at production scale requires several components: the AI model, a data repository, embedding models that transform content into searchable vectors, a vector database for similarity search, orchestration logic to keep the index synchronized with changing data, and security enforcement to control access. These components are often independent systems connected by custom integration code. Data must be copied between systems, permissions synchronized across disparate access control models, and any failure at the integration interfaces can cause the pipeline to return stale results or exhibit strange behavior. In addition, RAG systems are constantly evolving: each component will have different scaling limits and operational overheads, and the logic, embeddings, models, and functionality of the RAG system will evolve as the state of the art advances.

The VAST InsightEngine was developed to reduce this complexity by implementing RAG capabilities as a native service within the VAST AI OS. Rather than integrating independent systems, InsightEngine leverages existing VAST primitives:

- The VAST DataStore detects when data lands in a bucket and triggers serverless functions in the VAST DataEngine to chunk and vectorize content
- The resulting embeddings are stored as vector-indexed columns in the VAST DataBase alongside non-vector columns containing chunk data, source references, and other metadata

Because transformation logic, storage, query execution, and access control operate within the same system that manages the source data, the architecture eliminates data copying, permission synchronization, and other points of friction that bring fragility to production RAG pipelines.

With the VAST InsightEngine, RAG applications query embeddings for relevant context, and the system uses its unified understanding of identity across the query, the vector database, and the underlying data objects to return only data the end user is authorized to access. Data and embeddings exist within the same storage domain and identity domain, enabling InsightEngine to always serve RAG requests with an up-to-date view of all the data stored within the VAST AI OS.

Architecture and Data Flow

The VAST InsightEngine is a service that automates the RAG process, starting from the moment data is written into the VAST AI OS and ending with an application, chatbot, or AI agent processing a query that is enriched with all relevant data stored within the VAST DataStore. The easiest way to explain how InsightEngine operates is to walk through this process and illustrate how InsightEngine uses different components within the VAST AI OS.

Event Detection

When data is written to the VAST DataStore, DataStore can publish a structured, self-describing event notification to the VAST Event Broker. The Event Broker is a Kafka-compatible streaming event platform that uses the VAST DataBase as its underlying storage layer, and by virtue of this fact, it inherits the benefits of linear scalability, extreme resilience, and low latency design of the underlying DASE architecture. This allows it to process millions of events per CNode, and as more CNodes are added to support higher object creation rates, the Event Broker's performance also scales to deliver proportionally higher event processing throughput.

InsightEngine combines these binary, self-describing events with the Event Broker's scalability to route events to the relevant functions at extreme rates and process changes in data in real-time. Using DataEngine's triggers and functions, as data is created or modified, InsightEngine immediately executes serverless functions in parallel that kick off instances of the chunking and embedding pipeline. This pipeline splits each new object into chunks, processes each chunk through an embedding model, and generates embeddings for each chunk. By building on DataEngine's event-driven architecture, InsightEngine calculates embeddings in near-real time. This ensures that RAG queries are always able to reference the newest data available and avoids the delays associated with conventional batch processing to generate embeddings.

Orchestration of Chunking and Embedding

The serverless function that is executed on every new object launched by InsightEngine is implemented just as any other DataEngine function would be: it is written in Python, its logic is stored in an external container registry, and its lifecycle is fully managed by DataEngine. This function's purpose is to initiate a pipeline that enriches the new object in a content-specific way. For text documents, the pipeline may simply parse the PDF structure, extract multimodal data (text, tables, and infographics), and compute embeddings across modalities. Video files may be split into segments and labeled using a vision language model before embedding, and other common file formats (such formatted documents and spreadsheets) use different context-specific pipelines.

In all cases though, the pipeline chunks the incoming data into segments appropriate for embedding, and the chunk size and overlap are configurable parameters that affect embedding quality. These chunks are then passed to an embedding model which can be either external to the VAST AI OS (e.g., any managed inference service such as those provided by OpenAI, Anthropic, or Google) or running as a locally hosted microservice (e.g., NVIDIA Inferencing Microservices (NIMs)).

InsightEngine is not prescriptive in choice of models; it simply orchestrates their use and relies on DataEngine to manage container lifecycles, scale, and allocate resources for functions. In addition, DataEngine can manage the microservices that provide embedding and inferencing capabilities for the pipelines that InsightEngine executes on new data. This flexibility allows InsightEngine to route different data types to a variety of content-specific embedding pipelines, each of which may rely on any combination of open-source, proprietary, and custom models to extract the most meaningful context from all data that is written to the VAST DataStore.

Because these chunking and embedding pipelines are triggered by events rather than centrally orchestrated, the rate at which new embeddings can be generated scales linearly with the computing resources available to execution functions and calculate embeddings.

Storing Chunked and Embedded Data

Once incoming data has been chunked and embedded, InsightEngine inserts them into a VAST DataBase table as separate columns:

1. Chunks are raw data extracted from the source object and are stored in a column that is optimized for large-scale storage and structured queries. Although these chunks are also present in the source data stored in the VAST DataStore, they are deduplicated by the Element Store.

2. Embedding vectors for each chunk are stored in a second column that uses a vector index to optimize similarity search using configurable approximate nearest neighbor algorithms.

3. Other metadata, such as S3 object metadata or content-specific attributes, are stored as additional columns that enable filtered vector search and metadata queries.

The VAST DataBase's row insert performance and the VAST DataStore's object ingest performance both scale linearly with CNodes. Because InsightEngine stores vector embeddings in DataBase and source documents in DataStore, its ability to insert new embeddings inherits this linear scalability. Additionally, the VAST DataBase's columnar storage format optimizes query performance for vector similarity searches, chunk retrieval, and metadata filtering, while its transactional semantics enable InsightEngine to recompute embeddings with updated models without re-ingesting source data or losing embedding lineage.

Vector Search

When a RAG query is issued to InsightEngine, it builds a distributed query that leverages the chunks and vector columns to enable searching and filtering that accounts for semantic similarity (through the vector table) and the metadata attached to all objects. InsightEngine enforces the access controls of source data by implicitly filtering query results based on the requesting user's permissions. As a result, when a user queries InsightEngine, they will not be given vectors or chunks belonging to source data that they do not have permissions to see.

The query execution flow is much more than simple filtering, though.RAG queries to InsightEngine are executed according to the following flow:

1. When a user, application, or agent initiates a vector search, InsightEngine captures and validates the requesting identity. That identity's access policies are applied during query execution.

2. The search query itself is vectorized using the same embedding model used to generate the stored embeddings. As with the initial embedding, InsightEngine uses a configurable embedding service endpoint to perform this step.

3. InsightEngine performs a vector similarity search to identify candidate chunks, applying any optional metadata filters included with the query. If reranking is requested, these candidates are reranked to refine relevance scoring.

4. InsightEngine then returns the set of chunks and their metadata (such as source and S3 location) that are both semantically relevant to the query and that the requesting user is allowed to access.

InsightEngine exposes two query interfaces at different abstraction levels:

- **RAG retrieval API:** Accepts natural-language queries, retrieves relevant chunks, and generates an LLM response grounded in those chunks. It returns the generated response along with sources and retrieval telemetry (embedding time, search time, accuracy scores). This is the highest-level interface for RAG applications.

- **Vector search API:** Accepts pre-computed embedding vectors, retrieves relevant chunks, and returns them without LLM generation. This enables integration with custom embedding pipelines, multimodal workflows, or applications that handle LLM prompting and response generation separately.

Both of these InsightEngine APIs rely on the SQL API provided by the VAST DataBase to perform similarity searches. InsightEngine provides the connection between vectors, chunks, other metadata attributes, and the source data and object permissions in DataStore.

Consistency and Cleanup

In addition to computing embeddings on ingest, InsightEngine continually manages embeddings as data changes within the VAST DataStore. Just as is the case when an object is created, object deletion generates an event in the VAST Event Broker. InsightEngine triggers a cleanup function in response, identifying all vector and chunk table rows that refer to the deleted object and removing them. This pruning prevents subsequent vector searches from returning vectors and chunks that belong to data that no longer exists.

InsightEngine is designed to work with object semantics, where data is created or deleted but never modified. However, because the DataStore supports expressing data as files and objects equivalently, InsightEngine can also respond when new files are created through an NFSv4 mount. In this case, InsightEngine launches the embedding pipeline when a new file is first closed. Changes to files do not trigger recomputation of embeddings, as this might cause extreme computational churn on active file systems. Instead, the embedding pipeline relies on DataEngine's scheduled execution trigger, and InsightEngine uses the VAST Catalog to compare last-modified dates to embedding timestamps to determine which embeddings are stale.

As a result of this, InsightEngine cannot maintain real-time embeddings for data being actively modified in-place. However, permissions are always synchronously enforced regardless of which protocol was used to last modify them. InsightEngine also maintains namespace consistency by updating chunk metadata in response to events such as renames and hard links as well.

Architectural Advantages

The tight integration of the VAST InsightEngine with the rest of the VAST AI Operating System offers unique advantages over AI data platforms that are assembled from multiple independent components. In addition to the obvious reduction in operational burden and vendor sprawl, InsightEngine's fit within the VAST AI OS offers the architectural advantages of a single, unified security domain and real-time availability of AI-ready data.

Simplified Data Governance for RAG

The metadata and data stored in the VAST AI OS is a single source of truth that spans all protocols. As a result, InsightEngine does not have to synchronize copies of source data or embeddings in multiple places; it stores references to parent objects and their permissions alongside vectors and chunks. T

The metadata and data stored in the VAST AI OS is a single source of truth that spans all protocols. As a result, InsightEngine does not have to synchronize copies of source data or embeddings in multiple places; it stores references to parent objects and their permissions alongside vectors and chunks. This unified integration of every component required by RAG simplifies data governance in several ways:

- **Live permission enforcement:** When an object's access controls or ownership changes, the next vector search reflects the new permissions immediately. Permission changes do not require waiting for reindexing or metadata synchronization.

- **Consistent audit trail:** The VAST AI OS logs access and query activity, providing visibility into who accessed what data and what queries were executed. Because audit logs are generated by the same platform that enforces permissions, there is no risk of audit data becoming inconsistent with actual access patterns.

- **Unified data lifecycle:** Vectors and chunks effectively inherit the retention and deletion policies of source data through InsightEngine. When an object is deleted or expires, its associated embeddings are immediately pruned, preventing orphaned vectors that reference data that should no longer be accessible.

- **Single security perimeter:** InsightEngine, the VAST DataBase, and source data in the VAST DataStore share a common authentication and authorization framework, reducing the number of credential systems to manage.

- **Centralized encryption:** Data is encrypted uniformly across all components. Source files, embeddings, and metadata all use a single tenant key, simplifying compliance validation and key rotation.

Together, these capabilities reduce the operational overhead of maintaining RAG systems in regulated environments, allowing security and compliance teams to focus on policy rather than coordinating across multiple infrastructure components.

Real-time Ingestion and Indexing

InsightEngine inherits the linear scalability and low latency of the underlying DASE architecture, allowing it to keep pace with continuous, high-velocity data streams. Its event-driven architecture ensures that every object uploaded triggers embedding generation immediately, eliminating the delays intrinsic to batch-scheduled ETL workflows and centralized orchestration. This real-time embedding is critical for applications where data is continuously streaming and the value of data recency is high. For example, waiting 10-20 minutes between when a video is captured and when it becomes searchable is unacceptable when real-time incident detection is tasked with identifying accidents, fires, or traffic congestion as they occur.

InsightEngine is designed to handle these real-time workloads without compromising on throughput. The VAST Event Broker can process hundreds of millions of events per second, potentially generating an order of magnitude more database transactions. To reduce per-event overheads and minimize latency under load, InsightEngine uses micro-batching; functions can be executed on arrays of events and process embeddings in batches, amortizing both database transaction costs and the latency of embedding generation. This enables sustained ingestion at production scales while maintaining the consistent latency required for applications to query newly ingested data in near-real time rather than waiting for batch processing windows.

Because each component of InsightEngine scales linearly, its latency and throughput can be scaled out by adding additional CNodes. The ultimate limit to the gap between when an object is created and its contents being indexed by InsightEngine is how quickly the embedding pipeline can be executed. This, in turn, is governed by the models used to process each chunk. A multi-stage embedding pipeline or large embedding model will increase this embedding latency. In addition, saturating all GPUs with embedding requests will drive up this latency. However, InsightEngine performs embedding asynchronous to the actual object creation to ensure that data ingestion is not blocked waiting for the embedding to complete. The VAST Event Broker durably queues these events, ensuring that embeddings are eventually computed even when GPU capacity temporarily lags behind ingestion rates.

Managing The Platform

Operating legacy data platforms at scale has required extensive engineering in multiple disciplines. A multipetabyte object store needs to be deployed and managed to hold the data lake of Parquet, image, and other files, including periodic migrations between hardware/software generations. The platform's data management layers typically run a metabase on a relational database like MySQL, which needs another cluster for the database server processes and a durable storage system to hold the database. Finally, the query execution engine, like Vertica, Spark, or Trino, runs on another compute cluster, and the servers in that cluster have local SSD caches, and all of that is just to manage the data lake.

Managing performance, or scaling a platform like that, requires careful planning of the storage, compute, memory, and network requirements at each layer, and balancing the demands of each. Customers who believe the myth that the only old-fashioned HPC parallel file systems can provide the performance GPU computing demands (see NFS Now For Speed to disprove this myth) have to maintain expertise in yet another complex system.

Now add in AI workflows that require a Kubernetes cluster to run user functions, an event broker to manage inter-process communications that requires yet another compute cluster, the very expensive GPU server farm that runs and/or trains the AI model, and a separate vector database to hold the vector embeddings the model uses to understand your data.

Compare that to the VAST AI Operating System where instead of three, four, or more separate clusters (Object Storage, metabase, query engine, vector database, Kubernetes, Kafka, Etc.) to manage, and keep in sync, there's a single cluster of CNodes providing access to exabytes of structured and unstructured data, and all the services that manage them. The VAST DataEngine provides a single environment that stores, catalogs, indexes, and runs your inference functions across all your structured and unstructured data on a single pool of resources.

API-First Design

Twenty-first century data centers should be managed not by a high priesthood of CLI bashers who are charged with maintaining farms of data storage silos, but by orchestration platforms that manage individual devices not through a CLI but through more structured, consistent, and accessible application program interfaces (APIs).

VMS is built with an API-first design ethos. All VMS management functions on the Cluster, from creating NFS exports to expanding the cluster, are exposed through a RESTful API. To take the complexity out of learning to work with APIs and writing code across programming languages, the VMS publishes APIs using Swagger. For the uninitiated, Swagger is an open-source API abstraction that provides tools for building, documenting, and consuming RESTful web services.

While the VMS also provides a GUI (details below) and a traditional CLI, VMS's API-first design means the GUI and CLI consume the RESTful API rather than controlling the system directly. This approach ensures that API calls are sufficiently tested and that all system functions remain available through the RESTful API. Systems with CLI- or GUI-first design philosophies can often treat their RESTful APIs as second-class citizens.

A Modern GUI

While a RESTful API simplifies automating common tasks, GUIs remain the management interface of choice for customers who want a quick and simple view of system operations.

A good dashboard lets users quickly grasp their storage system's health in seconds and offers a comprehensive GUI that makes it easy to perform basic tasks while also enabling more curious users to explore the data behind the dashboard.

The VAST web GUI is implemented entirely in HTML5 and does not require Java, Adobe flash, browser plug-ins, or any other client software. Administrators can manage their VAST Clusters from any modern browser.

Viewing Analytics Data

The VAST GUI's main dashboard offers a system administrator or other authorized user a quick overview of the system's health. It also provides detailed analytics to understand events at both the system and application levels.

VMS's User Activity screen enables administrators to analyze application performance issues by revealing details about the users, exports, and clients interacting with the system, along with the traffic levels they generate. User Activity allows monitoring of any user in the system in real time, not just the top 10 "bad actors."

For historical data analysis, VAST Analytics dashboard provides administrators with a rich set of metrics to monitor everything from latency, IOPS, throughput, capacity and more – all the way to the component level.

Administrators can also create and save personalized dashboards by combining various metrics they find useful to identify event correlations.

VAST Cluster Management

Just as VAST Datastore was designed to redefine flash economics, VAST Clusters have been equally designed to minimize the cost of scale-out system operation by simplifying the system's full lifecycle of operation – ranging from day-to-day management tasks such as creating NFS Exports and quotas… all the way to performing automated, non-destructive updates and expanding the cluster online.

The VAST Management Service

The VAST Management Service (VMS) is responsible for all the system and cluster management operations across a VAST Cluster. VMS is a highly available service that runs as a Linux container in a VAST Cluster. VMS functions include:

- Exposing the VAST Cluster's RESTful API
- Serving the cluster GUI and CLI
- Collecting metrics from the servers and enclosures in the cluster
    - Reporting metrics to user
    - Reporting metrics to VAST Uplink
- Automating and managing cluster functions
    - Software updates
    - System expansion

VMS Resilience

VMS runs in a container that is independent from the container that runs the CNode and Protocol Manager processes. The VMS container only runs management processes that take instructions from the users, and exchanges data with the CNodes and Enclosures in the cluster. All VMS operations are out-of-band from the VAST Protocol Manager to ensure consistent I/O performance.

While VMS today runs on a single server, it is also designed to be highly available. Should a server running VMS go off-line, the surviving servers will detect that it isn't responding and first verify it is really dead before holding an election to assign a new host for the VMS container – at which point the new host will then spawn a new VMS process. Since all the management service state, just like Element Store state, is stored in the persistent Storage Class Memory, VMS takes over right where it left off when it ran on the failed host.

VMS Analytics

VMS polls all the systems in VAST Cluster every 10 seconds, collecting hundreds of performance, capacity, and system health metrics at each interval. These metrics are stored in the VMS database where they're used as the source for the VAST GUI's dashboard and other in-depth analytics which we'll examine in more detail below.

The system allows an administrator to examine analytics data over the past year without consuming an unreasonable amount of space, by consolidating samples as they age reducing granularity from a sample every 10 seconds to a sample once an hour for data over 30 days old.

VAST customers can consume VMS analytics directly through the VAST GUI, using the VAST REST API in scripts or through VASTs Prometheus exporter.

VAST Uplink

VAST's Remote Call Home Service

In addition to saving those hundreds of different types of system metrics to their local databases, VAST clusters also send encrypted and anonymized analytics data to VAST Uplink, a proactive remote monitoring service managed by the VAST Customer Support team. VAST Uplink is available on an opt-in basis. Customers with strong security mandates are not required to run Uplink to receive VAST support.

VAST's support and engineering teams use this platform to analyze and support many aspects of its global installed base, including:

- System health – in many cases the VAST Support team will know of system events before customer administrators do
- Monitor performance to ensure that VAST systems deliver the appropriate levels of service quality and expected throughput, IOPS, latency
- Publish non-disruptive SW updates to VAST's global installed base
- Track the benefits of Similarity-Based Data Reduction across different customers to develop trend lines around use cases
- and more.

Uplink also provides VAST's R&D team invaluable insight into how customers actually use VAST Datastore Clusters, so we can concentrate our engineering efforts on those that will have the greatest positive impact on customers.

Non-Disruptive Cluster Upgrades

Too many data management technologies today still have to be shut down to update their software. As a result, system administrators (who perform these outages during off-hours and weekends) will optimize their work/life schedule by architecting storage estates around the downtime events their systems impose upon them. The result is the use of many smaller systems to limit the scale of the outages they have to endure. Another way administrators avoid the perils of upgrade-driven storage downtime is to delay system updates, which has the unfortunate side effect of potentially exposing systems to vulnerabilities and fixes that their vendors have addressed in more recent branches of code.

VAST Data believes that disruptive updates are antithetical to the whole concept of an AI operating system. A multi-purpose, highly scalable, multi-tenant AI system must be universally and continually available even during routine maintenance events, such as system updates.

The Cluster upgrade process is completely automated: when a user specifies the update package to install, VMS does the rest. The statelessness of the CNodes also plays an outsized role in making cluster updates simple, as the system state does not need to be taken offline to update any one computer. To perform the upgrade, VMS selects a VAST Server (CNode) in the cluster, fails over the Server's

VIPs (Virtual IP Addresses) to other VAST Servers in the appropriate pools, and then updates the VAST Server container and the VASTOS Linux image (in the case of an OS update) on the host. VMS then repeats this process, transferring VIPs back to the updated servers as they reboot until all the VAST Servers in the cluster are updated.

Updating the VAST Enclosure (DBox) follows a similar process. VMS instructs the VAST Enclosure to reprogram an enclosure's PCIe switches to connect all the SSDs to one DNode, and all the VAST Servers then connect to those SSDs through the still-online DNode. Once failover is complete, VMS will update the DNode's software and reset the enclosure to bring it back online as part of an HA pair.

Since an EBox runs CNode and DNode containers, the EBox the upgrade process is usually the same as for discrete CNodes and DBoxes. The system transfers the work the CNode is performing, and its VIPs to other CNodes in the cluster and spins up a new version of the CNode container, just as it would on a physical CNode. Similarly, to update one of the DNode containers, the system transfers control of all of the EBoxes SSDs to the other DNode container, spins up a new DNode container, and transfers control of all the SSDs to the new DNode before repeating the process so both DNodes are updated.

The only difference is when the underlying Rocky Linux on the EBox has to be rebooted, in those cases, the system performs an orderly reassignment of the CNode's functions to other CNodes in the cluster and notifies the cluster's leader that the SSDs are going offline before restarting the box. While the box reboots, the system will migrate data in slightly narrower erasure code stripes and reconstruct any data requested by a client from the rebooting EBox's SSDs as it would if the EBox were offline for any other reason. EBox clusters are designed to remain available even when any two EBoxes are offline, accommodating rolling updates and failures simultaneously.

Expansion

The ability to scale storage performance via CNodes independently of DBoxes or EBoxes for capacity is one of the key advantages of the DASE architecture. Users can add CNodes and/or VAST Enclosures to a cluster at any time.

As we saw in the Asymmetric Scaling section, VAST clusters can be composed of heterogeneous infrastructure as the system scales and evolves over time. CNodes from different generations (CPU makes, speeds, and core counts), with different DBoxes with varying network connections, numbers, and sizes of SSDs, can all be members of the same cluster without imposing boundaries on datasets or introducing performance variance.

When CNodes are added to a cluster or Server Pool, they're assigned a subset of the cluster's VIPs and immediately start processing client requests on those VIPs, thereby boosting system performance. Users can orchestrate the process of adding CNodes to their cluster to accommodate expected short-term demand, such as during a periodic data load, and then release the hosts for other use at the end of that peak-demand period, thanks to the containerized packaging of VAST software.

When DBoxes or EBoxes are added to a VAST Cluster, the system immediately starts using the new Storage Class Memory and hyperscale flash to store new user data and metadata, providing a linear boost to the cluster's performance.

I/O and pre-existing data are rebalanced across the expanded capacity of the system:

- Writes are immediately spread across the entirety of the expanded pool of Storage Class Memory, so write performance automatically scales with the new resource
- Newly written data will be striped more widely across the expanded pool of hyperscale flash because of VAST's write-in-free-space approach to data placement, making subsequent reads to this data equally well-balanced
- Pre-existing data will remain in the SSDs it was originally written to and will, over time, be progressively restriped when the system performs routine garbage collection. VAST Systems don't aggressively rebalance data at rest on the senior enclosures because that would cause write amplification and degrade performance without any real benefit. VAST customers can force a restripe through the VAST management system.

Batch-Delete: The .Trash Folder

Many user workflows include cleanup stages that delete a working directory and its contents. VAST's batch-delete feature offloads this process from the NFS client to the VAST system, allowing workflows to proceed without waiting and reducing load on both the customer's compute servers and the VAST system.

Deleting large numbers of files via RM -rf /foo/bar or an equivalent can take considerable time, as the RM command walks the directory tree in a single thread, looking up and deleting files one by one.

To delete a folder and its contents, a user moves the folder to be deleted to a special .trash folder in the root of the folder's NFS export. Once the folder has been moved, the VAST system deletes its contents in the background.

Moving a folder is a single NFS rename call, relieving the client from walking the directory tree and deleting the files one by one.

Since the actual deletion occurs in the background, the system's free space won't reflect the space occupied by the deleted files for a few minutes after they're moved to the .trash folder.

Details:

- The .trash folder can be enabled or disabled by policy at the export and tenant levels
- .trash is a hidden folder bit-bucket black hole. Users cannot change its ACLs via CHMOD or access its contents in any way.
- Managers can set the GID to limit moving folders to the .trash to members of a group through the GUI/Rest API

Notes:

- Other vendors have implemented similar functions:
  - MS/PC-DOS's DELTREE command
  - Isilon Tree-Delete command

VAST Data Shield - Securing the VAST AIOS

Role-Based Access Controls

Just as large storage systems need quotas to manage capacity across a large number of users, VAST Datastore features role-based access control (RBAC) to enable multiple system tenants to work without gaining access to information that is beyond their purview.

A global administrator can assign read, edit, create and delete permissions to multiple areas of VAST system management, and establish customized sets of permissions as pre-defined roles that can be applied to classes of resources and users.

Add Role:

Realm	Create	View	Edit	Delete
Events	–	👁️	–	–
Hardware	–	–	–	–
Logical	🌟	👁️	✏️	❌
Monitoring	–	👁️	–	–
Security	–	👁️	–	–
Settings	–	–	–	–
Support	–	–	–	–

Options for defining RBAC controls in a VAST Cluster

Encryption at Rest

When encryption at rest, a system-wide option, is enabled, VAST systems encrypt all data using FIPS 140-3 validated libraries as it is written to the Storage Class Memory and hyperscale SSDs.

Even though Intel's AES-NI accelerates AES processing in microcode, encryption and decryption still require a significant amount of CPU horsepower. Conventional storage architectures, like one scale-out file system that always has 15 SSDs per node, can only scale capacity and compute power together. This leaves them without enough CPU power to both encrypt data and deliver their rated performance.

Conventional storage vendors resolve this by using self-encrypting SSDs. Self-Encrypting Drives (SEDs) offload encryption from the storage controller's CPU to the SSD controller's CPU, but that offload comes at a price: the premium SSD vendors charge for enterprise SEDs.

To ensure that we can always use the lowest cost SSDs available VAST systems encrypt data on the hyperscale SSDs in software, avoiding the cost premium and limited selection of self-encrypting SSDs. VAST's DASE architecture allows users to simply add more computing power, in the form of additional CNodes, to accommodate the additional compute load encryption may present.

Enterprise Key Management

VAST clusters can use external enterprise key managers to store the encryption keys used to protect data at rest, such as those from Thales, HashiCorp, and Entrust, or store the keys internally. In addition to the obvious security advantages of centralized key control, using an external EKM also allows much finer control over how data is encrypted, allowing VAST customers to use different encryption keys on a per-folder or per-tenant basis.

Encryption in flight

Using industry-standard protocols is one of the key design principles behind the VAST AI Operating System. Since those standard protocols define how clients send data on the wire those standards also define how VAST systems provide encryption in flight:

- NFS 3 – VAST supports NFS 3 encryption over TLS

- NFS 4.1 – VAST supports Kerberos crypto-authentication and encryption via Kerberos and/or TLS. TLS has somewhat lower overhead and therefore performance impact than Kerberos encryption

- SMB – VAST supports SMB 3.0 encryption and Scrypto-authentication
- S3, RESTapi, VAST DataBase -S3 and these other protocols run over HTTP so encryption in flight is provided via HTTPS which is HTTP over TLS

Attribute Based Access Control

The traditional security model for storage, and much of the rest of IT infrastructure, has been for each system or device to have a root or administrator account with godlike powers. File system ACLs theoretically allow users with sensitive data to deny root access to their data. The problem with that theory is that root's godlike powers allow them to change those ACLs, access the sensitive data, and reset the ACLs to cover their tracks.

Role-Based Access Controls (RBAC) let users delegate administrative functions to accounts that aren't omniscient or omnipotent, such as root. In theory, users would create delegated "demigod" accounts and leave root's password written down in a safe somewhere for emergencies. In reality, admins use the root account because they have to or as a shortcut, and all the system's data becomes vulnerable to a rogue admin or a bad actor with a compromised root password.

Following the principles of Zero Trust, Attribute-Based Access Control adds a layer of control limiting users, even root users, the ability to access or modify data, not just to who they are and what groups they belong to, but also to whether their Active Directory or other LDAP accounts include attributes that grant access. With ABAC, organizations can ensure that root users can't grant themselves access to data they shouldn't see without the directory service administrator also tagging their account with the required attribute.

VAST admins can add ABAC protection to a View by assigning one or more arbitrary directory attributes to the View as ABAC controls. When an administrator adds the "secret" tag to a view, the "secret" tag on each user's directory record determines that user's access to the view and its contents. Users who have the value of "secret" set to RW will have read/write access to the View, users where "secret"=RO will have read-only access to the View, and users with "secret" set to any other value, or that don't have a "secret" attribute in their directory account, will be denied access all-together.

When administrators assign more than one attribute to a folder, users will receive the most restrictive access, in line with Zero-Trust's principle of least privilege. ABAC works like SMB Share-level ACLs, filtering user access and applying before the file's ACLs.

Once ABAC is assigned to a View, every Element in that View is tagged with the ABAC attributes for the View, so ABAC is enforced even through any other Views that might grant access to that folder. Views with ABAC enabled can only be deleted when the underlying folder is empty, and that data can only be deleted by a user with rw in their ABAC attribute, preventing admins from deleting data they shouldn't.

Creating the Audit Trail

Strong authentication and discretionary access controls are your data's first line of defense, preventing users from accessing data they're not authorized to read or write. Unfortunately, many security breaches aren't completely untrusted people breaking down the proverbial front door to your data, but instead, someone who has access to data using that data in ways they shouldn't. When, for example, someone leaks the animatics from The Iron Giant II and posts them on YouTube, the question isn't who had access to those files, but who accessed all of the files that were posted on YouTube because your leaker certainly did.

To answer that question and more, VAST systems can keep an audit log of who accessed files, objects, or other elements, when they accessed them, and from which IP address across selected folders. Any time a user performs a CRUD (Create, Read, Update, Delete) operation on a VAST element or that element's metadata, where auditing is enabled, the action is logged.

The administrative audit log provides similar details for operations performed through the REST API. Since the VAST GUI and CLI consume the VAST API, administrative changes through those channels are logged as API calls.

While it’s initially appealing to log everything the system knows about every I/O, storage systems that log 100 bytes or more to describe every 4 KB read quickly end up with audit logs that take up more space than the data they’re supposed to manage.

To address this issue, VAST systems use several techniques to minimize audit overhead, including selective auditing of sensitive folders and operations, auto-pruning of audit logs by age or size, and, most significantly, deduplication of audit log entries. That deduplication means there’s an audit log entry when a user/client reads or writes to an Element, but there will be only one entry in the log, regardless of how many reads or writes the client performs during a default 5-second audit cycle.

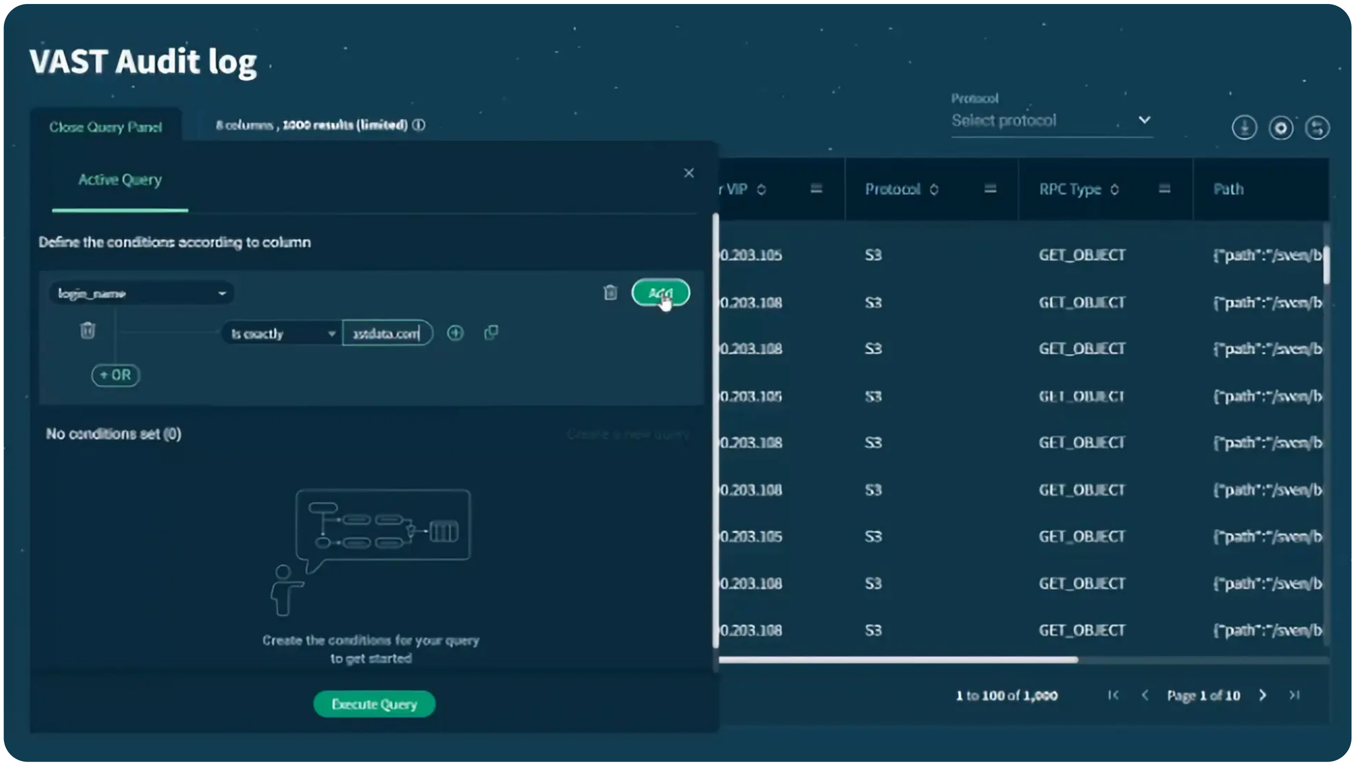
VAST customers can consume this audit data in two ways:as JSON files or as a table in the VAST database (more on that below). JSON files are a standardized, self-describing data transfer format that makes it easy to feed VAST audit data into a global SEIM (Security Information Event Management) system like Splunk or a more specialized cybersecurity tool like VAST Partner Superna’s Data Security Edition to rapidly detect ransomware attacks.

Auditing with The VAST Database

JSON is a great format for transferring data between applications, and while in a pinch, you could just open the JSON audit files in your favorite text editor; they are human-readable, after all. Actually finding who accessed the Smith file before it was leaked to the press, using raw JSON files, can be a bit tedious.

n a VAST cluster, each silo (the processes in a CNode that handle protocol requests) creates a JSON audit file every few seconds; finding who did some dirty deed with your data last week with a text editor wouldn’t just be tedious, it would be positively Sisyphean. For customers who don’t have a SIEM system to digest their audit data, and for those who want to use audit data for more than the obvious forensic security analysis, VAST clusters can write their audit data to a VAST Database table instead of, or in addition to, JSON files.

Once the audit data is in a database table, ad hoc questions like “Who accessed the Jones file last Tuesday?” become simple queries that an admin can process through the VAST GUI.



VAST Audit log

Close Query Panel   8 columns , 1000 results (limited)

Protocol: Select protocol

Active Query

Define the conditions according to column

login_name

Is exactly   astdata.com

No conditions set (0)

Create the conditions for your query to get started

r VIP	Protocol	RPC Type	Path
0.203.105	S3	GET_OBJECT	{"path":"/sven/b
0.203.108	S3	GET_OBJECT	{"path":"/sven/b
0.203.108	S3	GET_OBJECT	{"path":"/sven/b
0.203.105	S3	GET_OBJECT	{"path":"/sven/b
0.203.108	S3	GET_OBJECT	{"path":"/sven/b
0.203.108	S3	GET_OBJECT	{"path":"/sven/b
0.203.105	S3	GET_OBJECT	{"path":"/sven/b
0.203.108	S3	GET_OBJECT	{"path":"/sven/b
0.203.108	S3	GET_OBJECT	{"path":"/sven/b

1 to 100 of 1,000   Page 1 of 10

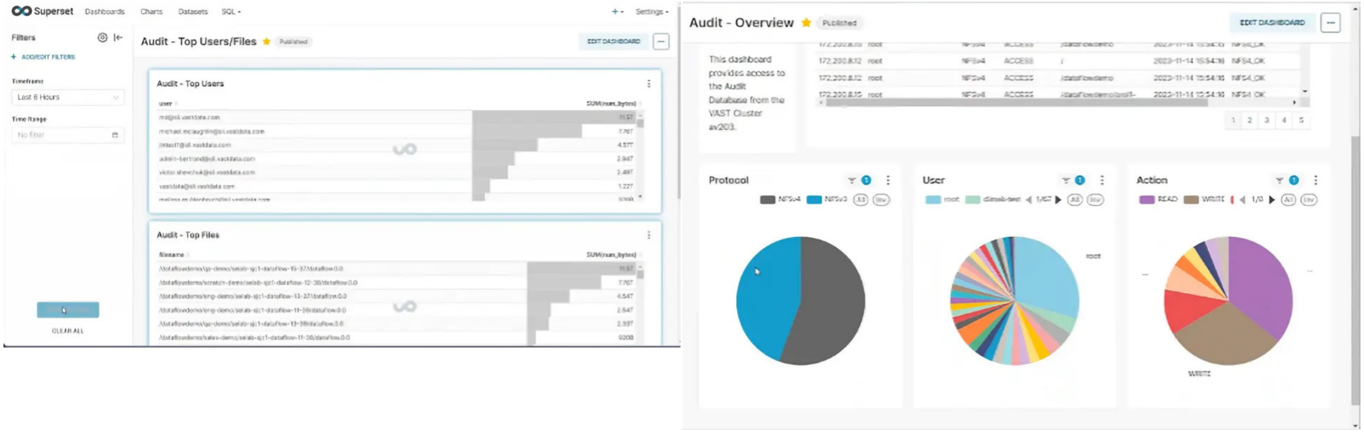
Accessing the Audit Table Through The VAST GUI

Even better, since the audit table is just another VAST database table, customers can slice and dice their audit data with standard SQL tools like Apache Superset, as shown in the examples below. The JSON files only contain data for fields that make sense for the I/Os, and each entry describes the audit table as having a column for every attribute tracked for access via any protocol. The contents of incongruous columns, like S3_Bucket_Name for an SMB read, are null.

Using the Audit Table for Usage and Performance Management

The VAST audit table has one other major advantage over the JSON version, and most other systems' similar logs are designed for forensic security analysis. The audit table includes two columns with additional data: the number of requests consolidated per row and the total bytes for those requests.

That additional information means the audit table isn't just the source of truth for forensic security but also a source of insight into how the system is used and the demands workloads place on it. Since that data is in a SQL table database, visualization tools like Apache Superset make it easy to create reports for the top users and busiest files or activity dashboards, as shown below.



Timeframe: Last 6 Hours

Time Range: No filter

Audit - Top Users

user	SUM(num_bytes)
mi@sl.vastdata.com	11.5T
michael.mclaughlin@sl.vastdata.com	7.76T
jmtest1@sl.vastdata.com	4.57T
admin-bertrand@sl.vastdata.com	2.94T
victor.shevchuk@sl.vastdata.com	2.49T
vastdata@sl.vastdata.com	1.22T
[illegible]@sl.vastdata.com	920G

Audit - Top Files

filename	SUM(num_bytes)
/dataflowdemo/qa-demo/selab-sjc1-dataflow-15-37/dataflow.0.0	11.5T
/dataflowdemo/scratch-demo/selab-sjc1-dataflow-12-38/dataflow.0.0	7.76T
/dataflowdemo/eng-demo/selab-sjc1-dataflow-13-37/dataflow.0.0	4.54T
/dataflowdemo/eng-demo/selab-sjc1-dataflow-11-39/dataflow.0.0	2.54T
/dataflowdemo/qa-demo/selab-sjc1-dataflow-13-39/dataflow.0.0	2.33T
/dataflowdemo/sales-demo/selab-sjc1-dataflow-11-38/dataflow.0.0	920G

This dashboard provides access to the Audit Database from the VAST Cluster av203.

172.200.8.12	root	NFSv4	ACCESS	/	2023-11-14 15:54:16	NFS4_OK
172.200.8.12	root	NFSv4	ACCESS	/dataflowdemo	2023-11-14 15:54:16	NFS4_OK
172.200.8.15	root	NFSv4	ACCESS	/dataflowdemo/[illegible]	2023-11-14 15:54:16	NFS4_OK

1 2 3 4 5

Visualizations from the Audit Table Using Apache Superset

The Audit Table is a companion to The VAST Catalog, which provides a view of what a system held at any point in time; the Audit Table provides a view of what the system was doing. Together, they can provide powerful insights.

Finally, the Audit Table can be used to characterize workloads and applications. Each audit entry describes a set of reads or writes, including the number of I/Os and total bytes, and is timestamped every few seconds. This makes it easy to calculate:

- The average size of reads and write I/Os
- The I/O rate (in IOPS) for reads and writes
- The ratio of reads to writes

Once you've done this for the pilot application, you can create a workload definition for a benchmark like FIO or ElBencho that generates the same I/O pattern as the application, so you can stress-test your infrastructure before putting the pilot in production, where it will perform 1000 times the I/O the pilot did.