WHITE PAPER

Disaggregated Elasticsearch on Pure FlashBlade: A Reference Architecture

A Framework for Deploying Elasticsearch® at scale on Pure Storage® FlashBlade™

Contents

Introduction

This document is intended for system administrators, storage administrators, IT managers, system architects, sales engineers, field consultants, professional services, and partners who are looking to design and deploy Elasticsearch on a FlashBlade platform. A working knowledge of Elasticsearch, Linux®, server, storage, and networks is assumed but is not a prerequisite to read this document.

This reference architecture:

- Explains the benefits of deploying modern Elasticsearch architecture on FlashBlade over traditional Elasticsearch with Distributed Direct-Attached-Storage (DDAS) and other NAS storage
- Report results of tests to demonstrate benefits including linear scalability, performance exceeding DDAS for both hot and ad-hoc queries and simple cluster management and upgrades

---

Deploying A Scalable, Disaggregated Elasticsearch with FlashBlade

The proliferation of modern apps, IoT, DevOps, and microservices have resulted in an explosive growth of machine-generated log data. There is an increasing need for enterprises to keep their infrastructure running 24x7. Monitoring all the infrastructure components (servers, storage, network, firewall, software, etc.) has become an increasingly crucial task. When things fail, the most common resolution method starts by examining the logs. Elasticsearch helps enterprises solve these complex log management and security challenges with a central repository to analyze logs originating from their distributed infrastructures.

Traditionally, an Elasticsearch cluster deployed in a Distributed Directed-Attached-Storage (DDAS) architecture provides high availability but, as the infrastructure grows, presents operational challenges. These include unpredictable search performance, infrastructure complexity and operational overhead, as well as underutilized resources due to rigid silos.

Elasticsearch and Pure Storage jointly address these challenges with traditional DDAS architecture so that enterprises can now focus on customer experience while maintaining a smaller infrastructure footprint. Pure Storage is a market leader for delivering all-flash storage arrays and eliminating storage infrastructure complexities and inefficiencies. FlashBlade provides the same ingest and hot query performance as DDAS solution, while delivering simplicity by disaggregating the compute and storage.

This document provides a reference architecture for deploying Elasticsearch with FlashBlade. With this disaggregated solution, FlashBlade delivers key advantages for Elasticsearch:

- **Faster Time to Insight for All Data:** Eliminate multiple storage layers and deliver consistent performance for all data, even “cold” tiers
- **Operational Simplicity:** Disaggregate compute and storage to deliver operational simplicity and the ability to scale them independently
- **Improved Agility for Diverse Workload Needs:** Virtualize multiple elastic workloads and deliver agility with -aaS functionality

- **Lower TCO:** Optimize infrastructure utilization and deliver lower compute and storage requirements vs. Distributed DAS deployment

To validate this reference architecture, we measured to validate two key dimensions: ingest performance by indexing documents onto FlashBlade; and search performance by fetching the log data from the FlashBlade.

We tested with 40 servers running Elasticsearch 7.2 and CENTOS 7, with each configured to run as a data node and master node. Additionally, six servers were used for load generation with Rally benchmark tool, and one server each was configured to run Kibana, Apache Bench, and Apache JMeter. On each of the Elasticsearch nodes, we configured a single mount point to FlashBlade (/mnt/esdata) to store Elasticsearch indices. On the load generator nodes, we created a single mount point to another FlashBlade (/mnt/rallydata) to read the raw data for bulk ingestion testing.

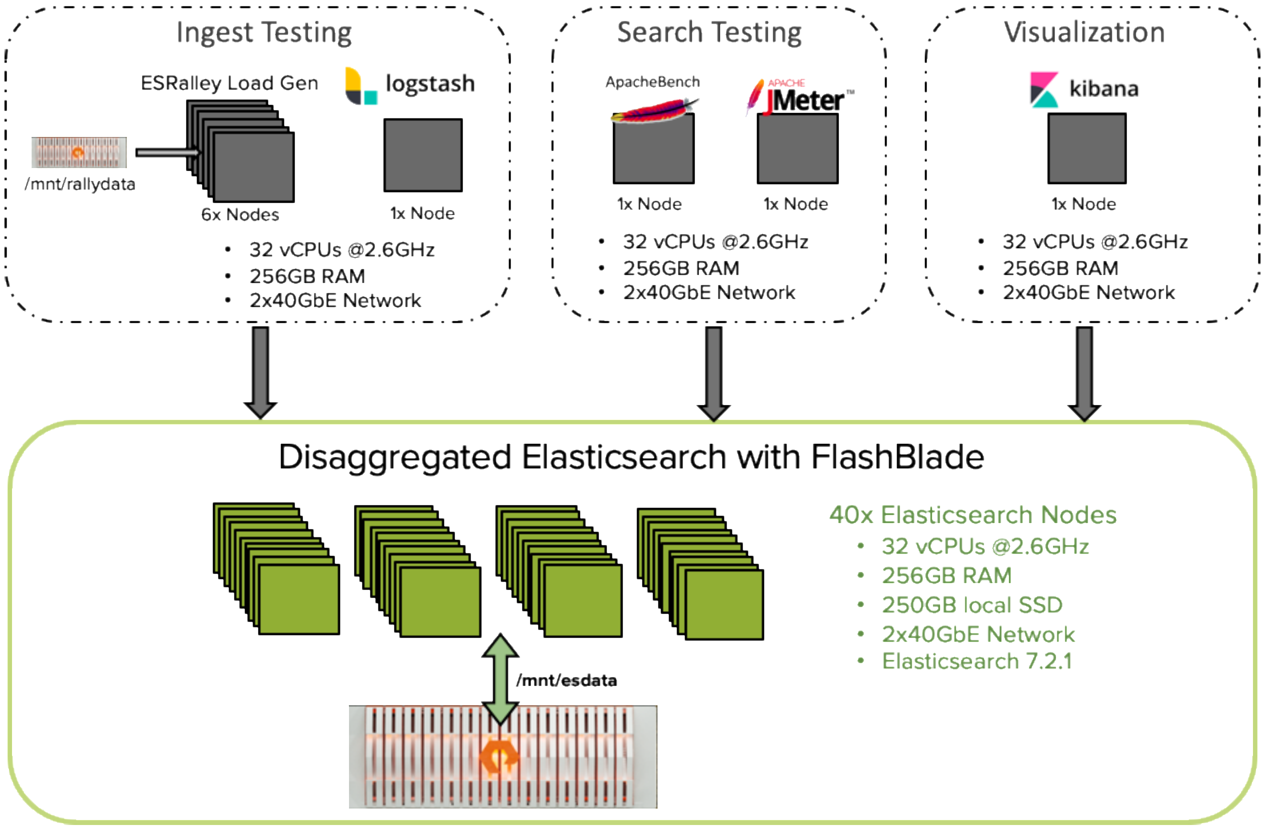
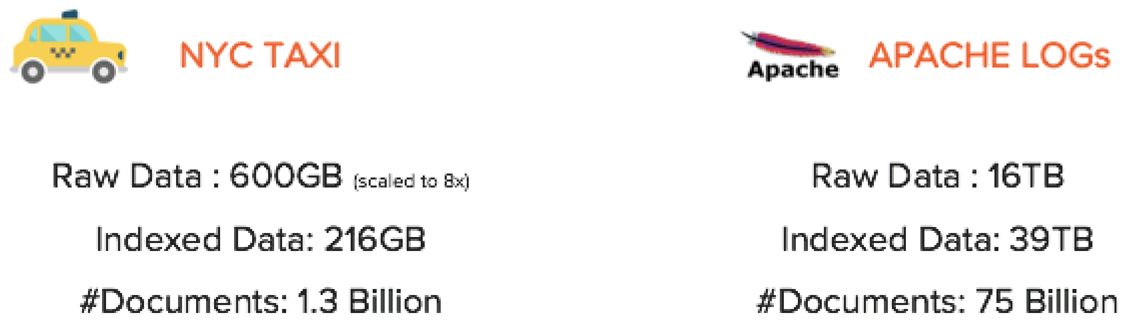


Figure 1: Test Setup

For the testing, we used two different data sets: a customized NYC taxi dataset and a dataset based on Apache Logs. The NYC taxi is a structured dataset (part of the Rally benchmark) that was customized to create 8x the data size in order to obtain more accurate ingest measurements via longer run times. The unstructured dataset based on Apache Logs was custom generated to create 16TB of raw log data.



NYC TAXI

Raw Data : 600GB (scaled to 8x)

Indexed Data: 216GB

#Documents: 1.3 Billion

APACHE LOGs

Raw Data : 16TB

Indexed Data: 39TB

#Documents: 75 Billion

Figure 2: Testing Datasets

Figure 3 below shows the ingestion performance for both DDAS and FlashBlade as the Elasticsearch nodes scale. The graph highlights two things. First, FlashBlade delivers equal or better performance as the distributed DAS cluster. Secondly, FlashBlade delivers the linear scalability for log data ingestion even as the number of Elasticsearch nodes increase. Hence FlashBlade brings simplicity due to disaggregation with no performance compromise for enterprises deploying Elasticsearch cluster.

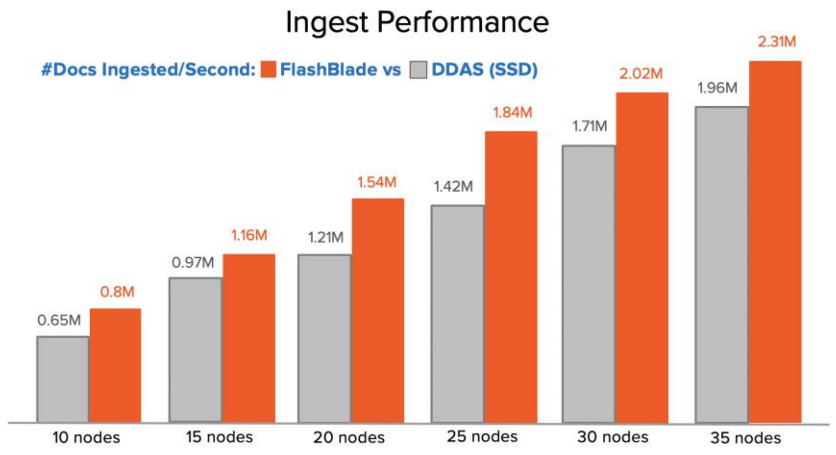


Figure 3: Ingestion Node Performance for DDAS vs. FlashBlade

Figure 4 shows the performance for hot queries (Elasticsearch's "hot" data queried from servers' DRAM) and ad-hoc queries (Elasticsearch's "cold" data queried from the FlashBlade). The top part of the image shows the performance of executing both the hot and ad-hoc queries. The bottom part of the image shows the FlashBlade storage performance while the queries are executed. Since hot queries are accessed from local servers, the storage traffic is seen only while executing the three batches of ad-hoc queries. From the graph, it is evident that running ad-hoc queries have little to no impact on the hot query performance. Furthermore, since the cold data can utilize the full read bandwidth of all-flash FlashBlade, the performance of ad-hoc queries and hot queries are similar.

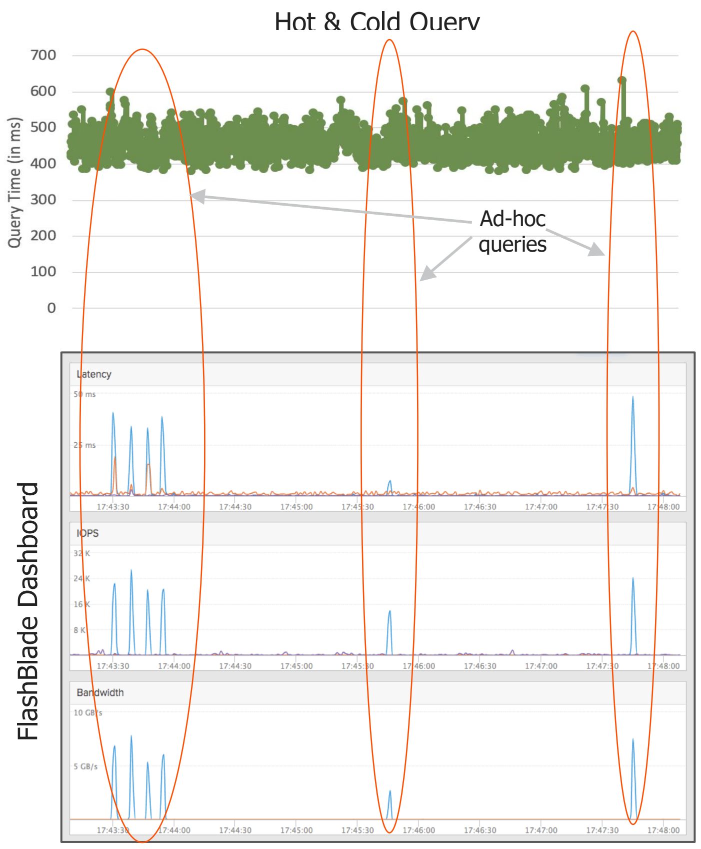


Figure 4: Hot and Ad-hoc query Performance

For more details, check Appendix-A for the reference architecture design and Appendix-B for the testing results.

Conclusion

Elasticsearch is a leading search vendor with increasing market presence and user growth. Elasticsearch deployments have been growing considerably across use cases that need faster ingestion as well as longer data retention. This stresses the underlying limitations of DDAS architecture and reduces Elasticsearch efficiency. Elasticsearch on FlashBlade provides a simple, scalable architecture for enterprises to achieve their goals faster.

The solution demonstrates that the scalability and simplicity of FlashBlade complements Elasticsearch's architecture. FlashBlade delivers faster time to insight for all data, hot or cold, by eliminating the storage tiers. Eliminating tiers also doubles the storage utilization by using 1 replica shards while delivering the same data protection. Due to the all flash read performance, even the ad-hoc queries deliver performance consistent with the hot tier.

Appendix A: Detailed Reference Architecture Design

The guiding principles for implementing this reference architecture include:

- **Simplicity** – Avoid unnecessary and/or complex configurations or techniques that make the results look better than a normal out-of-box environment.
- **Repeatability** – Create a scalable building block that can be easily replicated at any customer site. Publish the versions of firmware under test and uncover any issues in the lab before customers deploy the solution.
- **Availability** – Complement Elasticsearch high availability architecture with Pure's non-disruptive upgrade (NDU) capability.
- **Efficiency** – Take advantage of Elasticsearch expanding capabilities while leveraging Pure's low latency, high IOPS, and throughput, space, power, and cooling efficiencies.
- **Scalability** – Linear scalability of Elasticsearch deployment with FlashBlade.

What is the Problem FlashBlade and Elasticsearch are Trying to Resolve?

Elasticsearch is a distributed, RESTful search and analytics engine capable of solving a growing number of use cases. It helps small and large companies solve their complex data management and network security challenges while scaling their businesses. Accessible through an extensive and elaborate API, Elasticsearch can power extremely fast searchers that support data discovery applications.

Traditionally, Elasticsearch delivers simplicity with scale-out DAS based commodity servers. However, as the dataset grows to 100s of TBs, deployment requires specific server configurations. Enterprises using traditional DAS based architecture to address the performance challenge face rising costs as they scale. In addition, over time, the search performance degrades and becomes inconsistent.

While disaggregated compute and storage seems to be an obvious choice to address these limitations, the latency introduced impacts the Elasticsearch performance. We propose a solution with Pure Storage FlashBlade that eliminates the scalability challenges with Elasticsearch while delivering the highest performance and lowest latency for the search queries.

Technology Overview

FlashBlade

Pure Storage developed the FlashBlade architecture to meet the storage needs of data-driven businesses. FlashBlade is an all-flash system, primarily optimized for storing and processing unstructured data. A FlashBlade system can simultaneously host multiple file systems and multi-tenant object stores for thousands of clients.

A FlashBlade system's ability to scale in performance, capacity and connectivity are based on five key innovations:

- **High-performance storage device**
  FlashBlade maximizes the advantages of an all-flash architecture by storing data in storage units and ditching crippling, high-latency storage media such as traditional spinning disks and conventional solid-state drives. The integration of scalable NVRAM into each storage unit helps performance and added capacity scale proportionally when new blades are added to a system.

- **Unified network**
  A FlashBlade system consolidates high communication traffic between clients and internal administrative hosts into a single, reliable, high-performing network that supports both IPv4 and IPv6 client access over Ethernet links up to 100Gb/s.
- **Purity//FB storage operating system**
  Purity//FB symmetrical operating system running on FlashBlade fabric modules minimizes workload balancing problems by distributing all client operation requests evenly among the blades on FlashBlade.
- **Common media architectural design for files and objects**
  The FlashBlade single underlying media architecture supports concurrent access to files via a variety of protocols such as NFSv3, NFS over HTTP, and SMB (with Samba-level functionality) and objects via S3 across the entire FlashBlade configuration.
- **Simple usability**
  Purity//FB on FlashBlade alleviates system management headaches as it simplifies storage operations by performing routine administrative tasks autonomously. With a robust operating system, FlashBlade is capable of self-tuning and providing system alerts when components fail.

FlashBlade Hardware

A full FlashBlade system configuration consists of up to five self-contained rack-mounted chassis interconnected by high-speed links to two external fabric modules (XFM). At the rear of each chassis are two on-board fabric modules for interconnecting the blades, other chassis, and clients using TCP/IP over high-speed Ethernet. Each of the two fabric modules are interconnected and each contains a control processor and Ethernet switch ASIC. For reliability, each chassis is equipped with redundant power supplies and cooling fans.

The front of each chassis holds up to fifteen blades for processing data operations and storage. Each blade assembly is a self-contained compute module equipped with processors, communication interfaces, and either 17TB or 52TB of flash memory for persistent data storage.

The current FlashBlade system can support over 1.5 million NFSv3 getattrs per second, or >17 GiB/sec of 512KiB reads or >8 GiB/sec of 512KiB overwrites on a 3:1 compressible dataset in a single 4U chassis with 15 blades and can scale both compute and performance up to a 5 x 4U chassis with 75 blades.

Learn more about FlashBlade technical specifications

Elasticsearch

Elasticsearch is open-source software that indexes and stores information in a NoSQL datastore based on Apache Lucene search engine. Elasticsearch is the heart of the Elastic Stack (ELK), in which Logstash is used to process and stream data into the Elastic indexer and Kibana is used for querying and presenting results in user friendly dashboards.

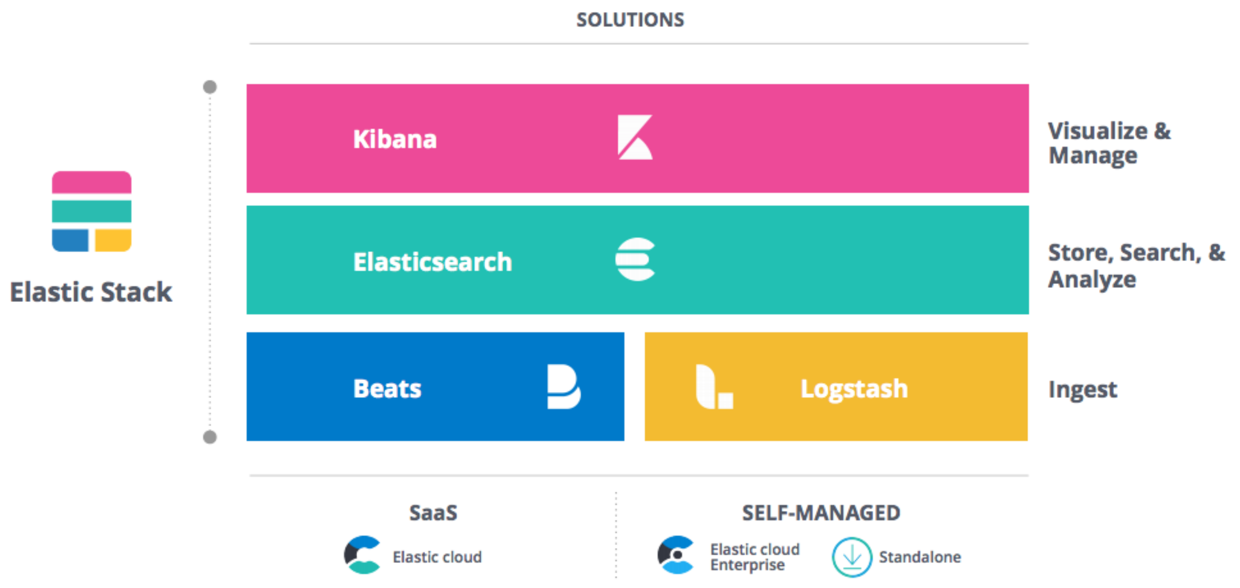


Figure 5: Components of Elastic Stack

Elasticsearch provides the following key advantages:

- **Especially suited to large and growing database environments**, the Elasticsearch distributed architecture can scale up to thousands of nodes while the cluster solution can scale to accommodate petabytes of data. Elastic Common Schema (ECS) offers support for multi-tenancy in these environments. Distributed indexes divide into shards with replica support for each shard. Routing and rebalancing are done automatically to ease the burden on IT.
- **Elasticsearch simplifies search** for a range of use-cases (e.g. classic full-text search, analytics, auto-complete, spell checker, alert engine and general-purpose data store). It extends searching capabilities using APIs and query DSLS for easy integration with numerous programming languages. Elasticsearch supports several search options including text splitting, custom stemming, faceted search, full-text search, auto completion, instant search, fuzzy search and suggestions. It's document-oriented to store complex data structures in json format and indexes all fields by default.
- **High performance** results from the use of RESTful APIs and cached search filters while translog changes can be captured to retain data integrity. Recent versions have incorporated Machine Learning modules.

Elastic Data Storage

In the Elasticsearch paradigm, documents are JSON objects stored within an index as a base unit of storage. Within each document are fields comprised of keys and values.

Mapping defines the fields for documents of a specific data type and determines how it should be indexed and stored in Elasticsearch.

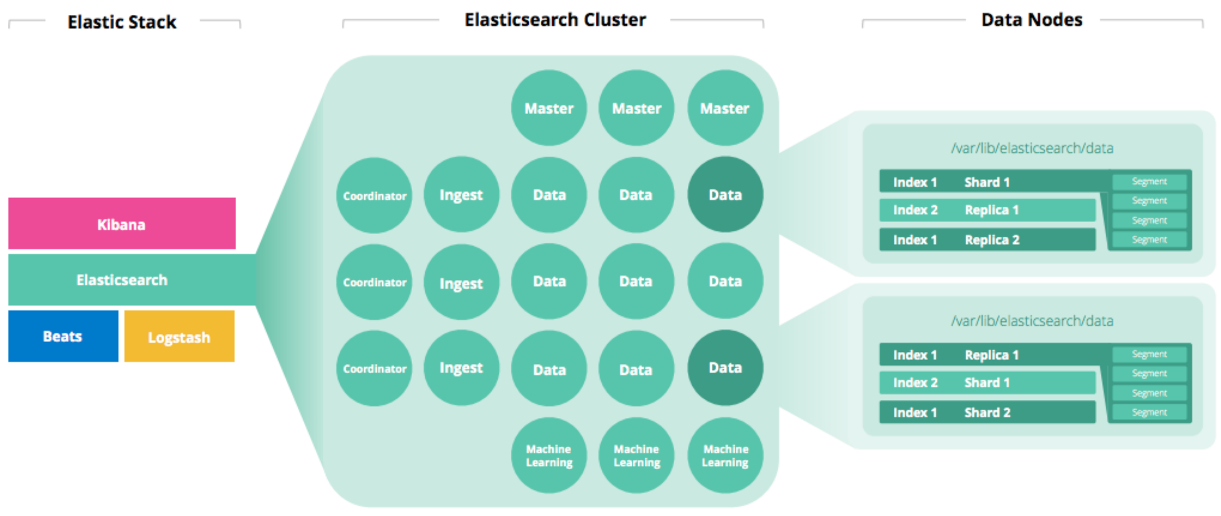


Figure 6: Elasticsearch Architecture

A shard is a single Lucene index forming the building block of Elasticsearch and facilitating its scalability. Indices are the largest unit of data in Elasticsearch. They are logical partitions of documents as shards.

While creating an index, the user defines the number of shards. Each shard is an independent Lucene index that can be hosted anywhere in the cluster. The replica shard is a copy of the index's primary shard. It's used for backup and restore should a node crash.

While Elasticsearch nodes can perform one or multiple roles, it often makes sense to assign one role per node and to optimize the hardware for each role to prevent nodes from competing for resources.

Data nodes store data and execute data-related operations such as search and aggregation.

Master nodes handle cluster-wide management and configuration actions (e.g. adding and removing nodes) and help ensure the stability of clusters by preventing other nodes from consuming any of their resources.

Client nodes forward cluster requests to the master node and data-related requests to data nodes.

Ingest nodes that run many pipelines or use many processors will demand extra compute resources.

Machine Learning that run many jobs or use many splits, buckets, or complex aggregations will demand extra memory and compute resources.

Dedicated Coordinator nodes can benefit hybrid use cases by offloading the merge phase of searches from data nodes that are constantly indexing.

Node data:

```
$ tree data
data
└── elasticsearch
    └── nodes
        └── 0
            ├── _state
            │   └── global-0.st
            └── node.lock
```

Figure 7: Empty data directory when starting Elastic Node

The node.lock file is there to ensure that only one Elasticsearch installation is reading/writing from a single data directory at a time.

Index Data:

When you create an index, data is stored in various folders.

```
$ tree -h data
data
└── [ 102]  elasticsearch
    └── [ 102]  nodes
        └── [ 170]  0
            ├── [ 102]  _state
            │   └── [ 109]  global-0.st
            ├── [ 102]  indices
            │   └── [ 136]  foo
            │       ├── [ 170]  0
            │       │   ├── .....
            │       └── [ 102]  _state
            │           └── [ 256]  state-0.st
            └── [   0]  node.lock
```

Figure 8: Index data stored in various folders

Shard Data:

The shard data directory contains a state file for the shard that includes versioning as well as information about whether the shard is considered a primary shard or a replica.

```
$ tree -h data/elasticsearch/nodes/0/indices/foo/0
data/elasticsearch/nodes/0/indices/foo/0
├── [ 102]  _state
|   └── [  81]  state-0.st
├── [ 170]  index
|   ├── [  36]  segments.gen
|   ├── [  79]  segments_1
|   └── [   0]  write.lock
└── [ 102]  translog
    └── [  17]  translog-1429697028120
```

Figure 9: Shard Data Directory Structure

The **Elasticsearch transaction log** makes sure that data can safely be indexed into Elasticsearch without having to perform a low-level Lucene commit for every document. Committing a Lucene index creates a new segment on the Lucene level which is fsync()-ed and results in a significant amount of disk I/O which affects performance.

Incoming documents are initially stored in translog and then stored in segments. Several rounds of merge could happen to segments.

Indexing needs high write throughput at consistent, low-enough latency. We recommend doing extensive testing at scale for your data.

Disaggregation enables service delivery and scaling models not possible with local drives.

Challenges for Enterprises

Collocated storage

The Elasticsearch cluster deployment is based on a distributed scale-out model that provides high data availability and fidelity but also presents significant cost challenges. Elastic nodes are configured to replicate shards, thus preventing data loss and facilitating searches in case of a node failure. This model is well suited to earlier Big Data technologies, such as Hadoop, which relied on close server-storage proximity to achieve high performance.

In a distributed scale-out model, every Elastic data node is configured-- predominantly through direct attached storage (DAS)—to have similar sized storage for Hot/Warm and Cold tiers. This model worked well in the past to process the necessary volumes of data. However, as data grows, storage and compute requirements don’t scale linearly. Adding a node of the same type with compute and storage to address the storage requirement isn’t only suboptimal but also cost-prohibitive.

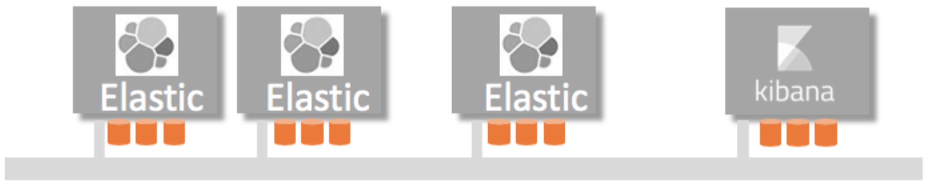


Figure 10: Classical Elasticsearch with DDAS

Data Usage, Tiering and Retention Requirements

A key requirement of a distributed scale-out model is that replica shards are stored on additional nodes in the cluster based on the replication factor. Hence the storage requirement spikes up considerably with higher replication in a cluster environment.

Capacity planning

It's important to plan for growth and provision capacity for efficient operational performance. This means Elasticsearch clusters need to be designed to keep nodes up and running at a capacity that can accommodate spikes in search or index jobs. At the same time, one must keep memory from growing out of control and prevent unexpected actions/events from bringing down nodes.

While there is no set formula for planning capacity, specific best practices can assist:

- Simulate your actual use-case: setup nodes and fill with real dataset until the shard breaks.
- Try various combinations of shard size, bulk indexing clients, bulk sizes.
- Determine optimal numbers and configure your index with the correct shard-size and refresh-interval.

Once you define a shard's capacity, you can apply it throughout your entire index. It's important to understand the optimal resource utilization during the testing process in order to determine the proper amount of RAM for nodes and JVM heap space.

The Solution

The solution presented here is comprised of Elasticsearch on FlashBlade as the storage layer for distributed nodes.

FlashBlade is a ground-breaking scale-out all-flash file and object storage system from Pure Storage architected to consolidate complete data silos while accelerating real-time insights from machine data using applications such as Elasticsearch.

Physical machines running Centos 7.4 were used for elastic nodes.

CentOS version 7.4 is a Linux distribution derived from the sources of Red Hat Enterprise Linux (RHEL) and provides free, enterprise-class computing platform functionalities.

The current FlashBlade system can support a read rate up to 17Gbps and a write rate of 4.5Gbps, in a single 4U chassis with 15 blades and can scale all the way up to 85Gbps of read rate and 20Gbps of write rate on 5 x 4U chassis with 75 blades.

Elasticsearch accelerated by Pure Storage FlashBlade
In a traditional DAS environment, storage resources use CPU cycles which might be otherwise available to indexer nodes. Thus, there is added overhead and no data service. By contrast, in a FlashBlade environment, compression can significantly reduce the storage requirements of the Elasticsearch clusters.

The following diagram shows Elasticsearch with high performance NFS as storage components of the architecture.

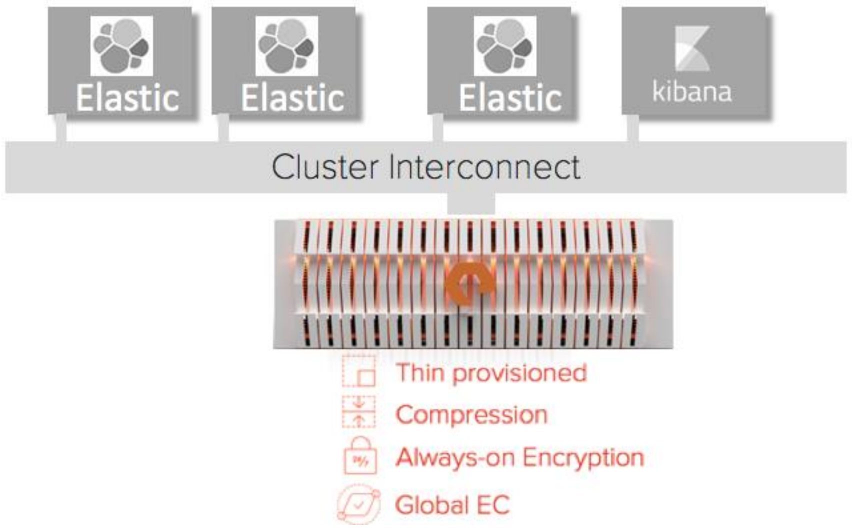


Figure 11: Elasticsearch with NFS

FlashBlade with RAID augments data protection while erasure encoding reduces the operational burden for system administrators who must determine and create RAID volumes every time a Data node is added. In addition, FlashBlade provides always-on encryption services that keep the data encrypted at rest.

Compared to DAS, FlashBlade offers better performance for the Elasticsearch core functions, data ingest and searches. Thanks to FlashBlade's technology, the performance characteristics of the Cold tier now match that of the Hot/Warm tier. Customers can search all available data across Hot/Warm and Cold tiers and get similar performance.

Index process:

During the index process, changes to a shard/Lucene are only persisted to storage during a Lucene commit, which is a relatively expensive operation and hence can't be performed after every index or delete operation.

Because Lucene commits are too expensive to perform on every individual change, each shard copy also has a *transaction log* known as its *translog* associated with it. All index and delete operations are written to the translog after being processed by the internal Lucene index but before they are acknowledged. An Elasticsearch flush is the process of performing a Lucene commit and starting a new translog. Flushes are performed automatically in the background in order to make sure the translog doesn't grow too large. The data in the translog is only persisted to disk when the translog is fsynced and committed.

A storage type commonly used in Elastic is the NIO FS type, which stores the shard index on the file system (maps to Lucene NIOFSDirectory) using NIO. It allows multiple threads to read from the same file concurrently.

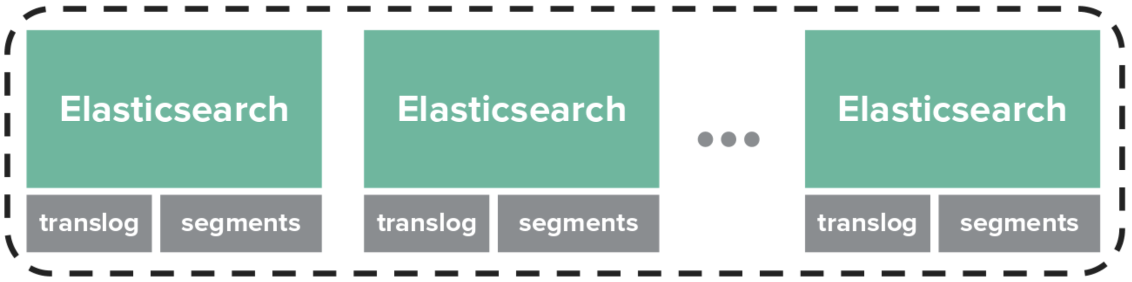


Figure 12: Indexing in Elasticsearch

Overall, Elasticsearch on Pure Storage systems

- Provides highly scalable storage solutions
- Provides all flash performance across all storage tiers
- Enables dynamic cluster scaling by adding compute and storage independently
- Further reduces the space usage through compression
- Protects Elastic index data at rest through encryption

Solution Design

Design Topology

This section describes the design topology for the Elasticsearch on Pure Storage systems that was tested in our lab.

The solution includes 40x Intel CPU based servers for hosting the Elasticsearch nodes. Six other nodes were used for Load generator and running benchmark tools. The solution includes Pure Storage FlashBlade with 15 blades to host the data storage layer.

FlashBlade Configuration

The FlashBlade hosts Elasticsearch index data, compressed and encrypted at the storage level. A shared NFS filesystem was used with subfolders for each data node, connected over ethernet and hard mounted to the Elastic nodes.

Component	Description
FlashBlade	15 x 17 TB blades
Capacity	240 TB raw
Connectivity	162.46 TB usable (with no data reduction)
Physical	4 x 40Gb/s Ethernet (data)
Software	2 x 1Gb/s redundant Ethernet (Management port)

Operating System and Software Configuration

OS and Software	Description
Linux	CentOS 7.4 (64 bit)
Elastic Search	Elastic 7.2.1
Rally Benchmark	Version 1.1

Physical Topology

The solution, Elasticsearch on Pure Storage, consists of a combined stack of hardware (compute, storage, network) and software (Elasticsearch, CentOS Linux).

Component	Description
Elastic Nodes	40x Intel based server each with: · 2 x Intel Xeon Gold 6138 @ 2GHz (20 cores) · 256GB of memory
Benchmark nodes	6 Intel based server each with: · 2 x Intel Xeon processor E5-2670 v3 CPUs (12 cores) · 256GB of memory
Logstash	6x Intel based server each with: · 2 x Intel Xeon processor E5-2609 v4 CPUs (8 cores) · 64GB of memory
Networking	TOR 2x Cisco Nexus 9372PX Ethernet Switches
Virtual Interface Cards:	40Gbps Unified I/O ports on Cisco UCS VIC 1340

Logical Topology

For enterprise-grade deployment of Elasticsearch that requires multiple terabytes of data to be ingested daily while supporting numerous concurrent searches, we recommend a distributed deployment of elastic clusters.

For the lab test environment, we used:

- 40 Elastic data nodes
- 1x Benchmark coordinator node
- 5x Benchmark load generator node
- 6x for Logstash nodes on the same benchmark nodes

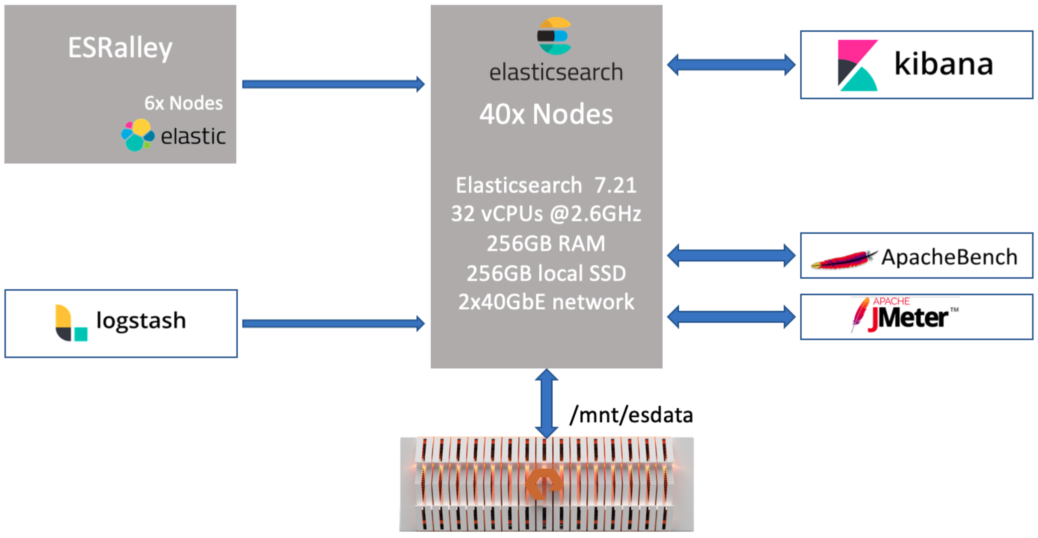


Figure 13: Benchmark Setup

Design considerations

Elasticsearch deployments are generally based on the following factors:

- **Daily ingest rate or indexing volume** - Higher index rates require more index nodes
- **Number and type of searches** - Numerous concurrent searches or resource-intensive searches (i.e. dense search) can tax the search heads and indexers.
- **Number of concurrent users** - Numerous users viewing dashboards or running searches would require more search heads, ideally a search head cluster.
- **Data fidelity** – Determine replication factor to handle failover scenarios.
- **Data Availability** – Only data nodes can search and provide access to the full set of index data.
- **Disaster recovery requirements** - A multisite elastic cluster spread between two data centers enables you to maintain identical data sets for quick recovery

Elasticsearch manages the data fidelity, availability and disaster recovery requirements at its software layer by replicating the index data by means of shards which are distributed across the cluster.

Performance
Planning system resources and bandwidth to enable search and index performance in a distributed Elastic cluster environment must factor in the total volume of data being indexed and the number of active concurrent searches (scheduled or other).

A proper sizing takes the following steps:

1. For each applicable tier – Hot, Warm, Frozen – determine the largest of the following sizes:
    a. Data volume
    b. Shard volume
    c. Indexing throughput
    d. Search throughput
2. Combine the sizes of each tier
3. Make decisions on any dedicated nodes – Master, Coordinator, Ingest, Machine Learning

Appendix B: Solution Validation and Testing

The Elasticsearch solution on FlashBlade was validated by testing the three key functions of Elastic Search namely data ingestion, search and cluster operations.

- Data Ingestion
- Search Behavior
- Operational Efficiency

Data Ingestion

While Elasticsearch provides several options to ingest data, Bulk Index API, Logstash and beats are common. The below diagram highlights a data processing flow during Index operations:

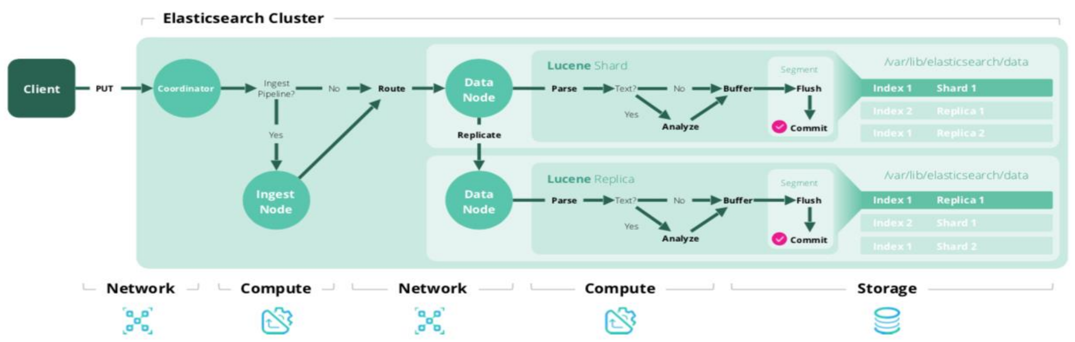


Figure 14: Data Processing Flow in Elasticsearch

Ingest test overview:

We determined the maximum performance during data ingestion of both structured and unstructured data by performing two tests:

- We used the Elastic Rally benchmark tool to ingest large datasets, making copies of the built-in dataset to increase the total. Primarily we used the NYC taxis data set.
- We created a home-grown tool to generate Apache logs of 16TB of raw data.

Ingest test setup:

Setup for indexing was based on Elastic Rally benchmark tools. We configured five load generators and one coordinator. The source data for indexing was saved in drive mounted on another FlashBlade.

Source datasets:

Structured dataset :
NYC taxis (Part of ES Rally tool)
Raw data : 600GB (scaled to 8x)
Number of documents: 1.3 Billion

Unstructured dataset:
Apache Logs - custom generated with internal tool.
Raw Data : 16TB
#Documents: 75 Billion

System mount config on Linux nodes:

Edit /etc/fstab to add mount config

```
10.21.152.245:/esdata /mnt/esdatavip nfs rw,bg,nointr,hard,tcp,vers=3 0 0
```

To verify:

```
$df -h /mnt/esdatavip
Filesystem Size Used Avail Use% Mounted on
10.21.152.245:/esdata 80T 47T 34T 58% /mnt/esdatavip1
```

Elasticsearch config for FlashBlade storage:

Edit elastic config file $ES_HOME/config/elasticsearch.yml

```
cluster.name: cks-es-40n
node.name: ${HOSTNAME}-0
path.data: /mnt/esdatavip/${node.name}/n1
path.logs: /var/log/elasticsearch/${node.name}
bootstrap.memory_lock: true
node.master: true
node.data: true
```

Ingest Test Results

Bulk size scaling test

Bulk inserting is a way to add multiple documents to Elasticsearch in a single request or API call. This method improves performance because it eliminates the need to open and close a connection for individual documents. While bulk sizing is dependent on your data, analysis, and cluster configuration, a good starting point is 1K. In our series of tests to measure the ingest performance against varying Bulk size, **good bulk size was 100K** for the given dataset.

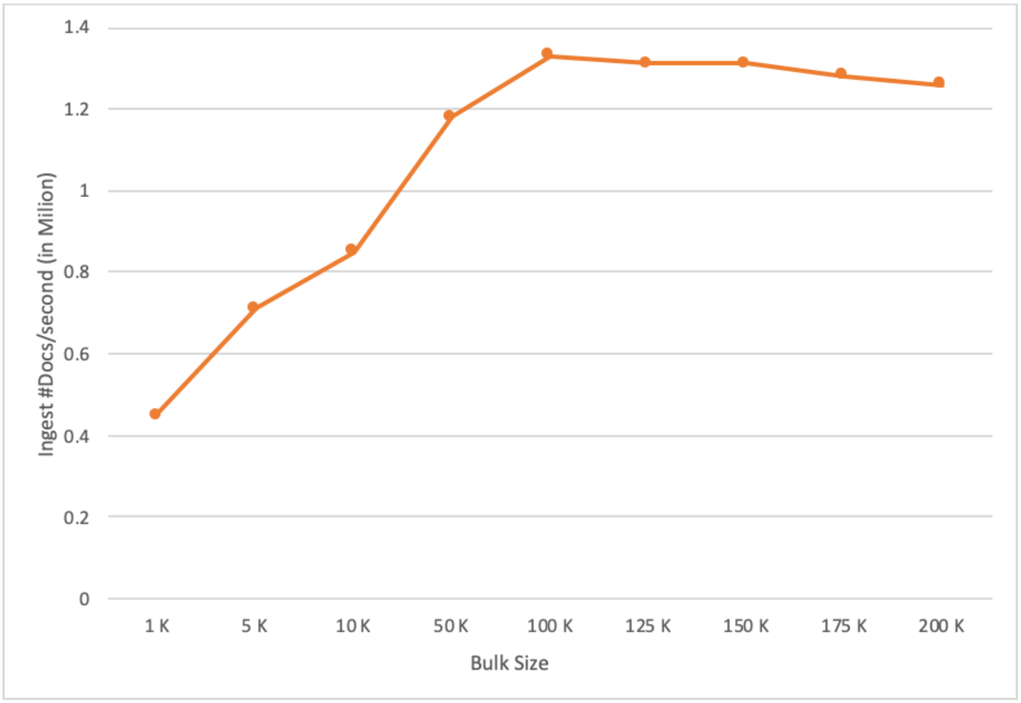


Figure 15: Bulk Document Insert Performance with FlashBlade

Shards scaling

Sharding allows scale out by partitioning data into smaller chunks that can be distributed across a cluster of nodes. The replica shard is a copy of the shard used to prevent data loss. It's designed from the ground up to be horizontally **scalable**, meaning that by adding more nodes to the cluster it's possible to grow the capacity of the cluster. In our test, we configured bulk document size to 100K and used 40 index nodes. Figure 16 shows that the for the given dataset the ingest performance was ideal with **80-100 shards per node**.

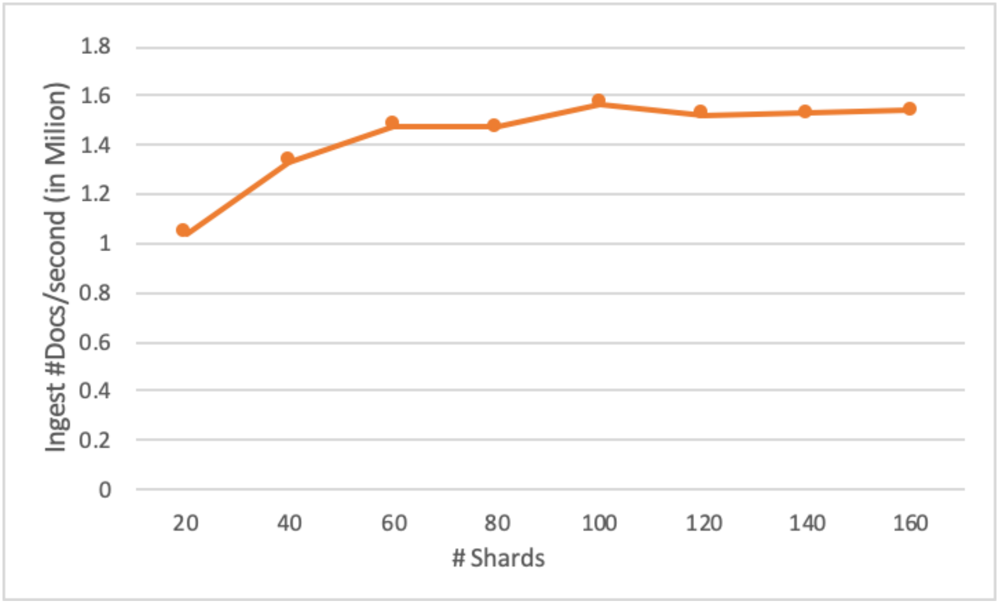


Figure 16: Ingest Performance vs #shards with FlashBlade

Bulk index clients scaling

Bulk index clients are the number of simultaneous threads/connections for bulk indexing. A good starting number is 10 clients. In our series of test, we increased the bulk index clients and found **80-100 bulk index clients** to show good performance for the given dataset configured with 40 shards/index.

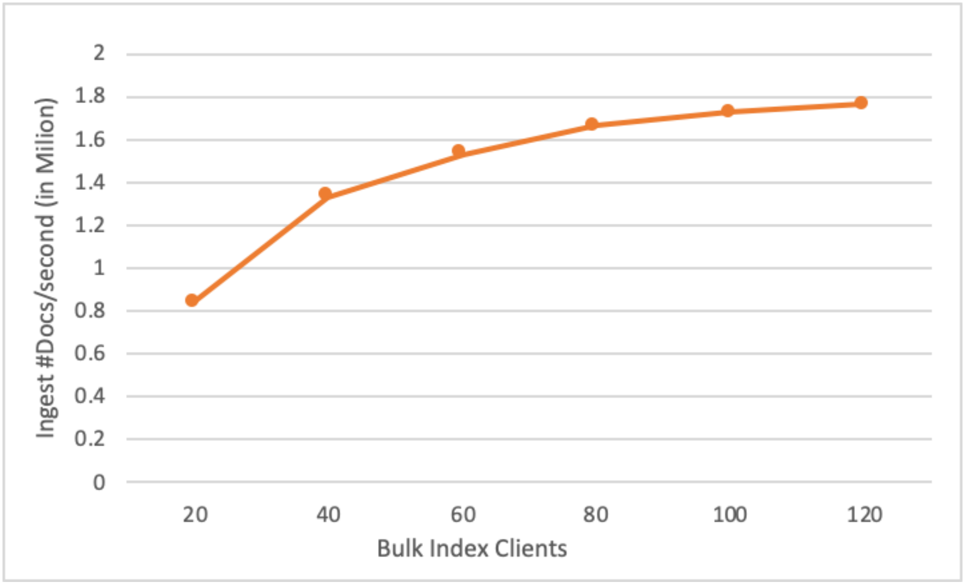


Figure 17: Ingest Performance vs #Index Clients with FlashBlade

FlashBlade Utilization during Index benchmarks:

During all the index benchmarks, the FlashBlade achieved approximately 2-3Gb/s of write. Figure 18 shows the FlashBlade performance metrics captured across all the tests.

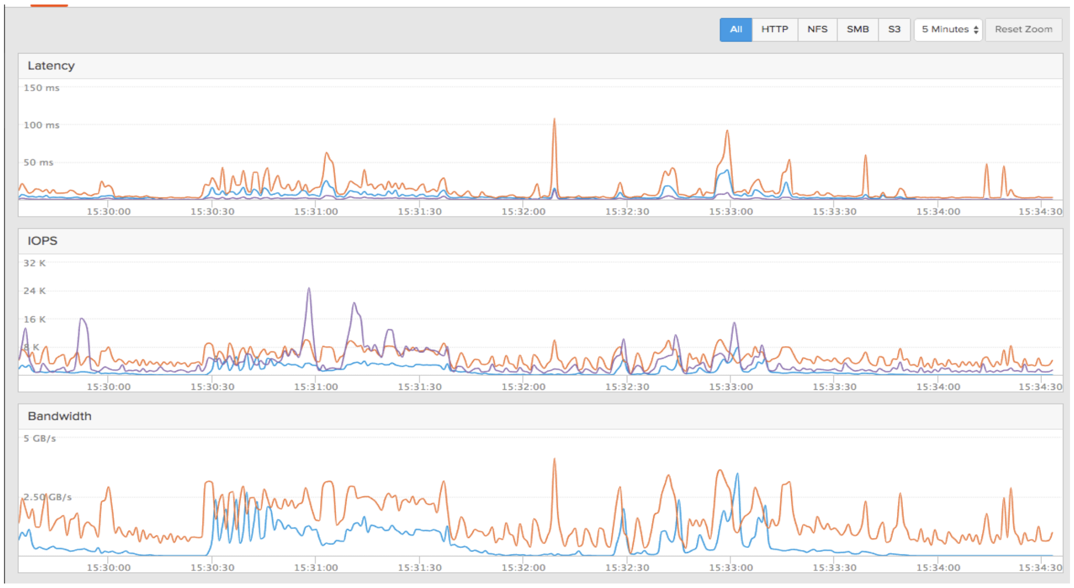


Figure 18: FlashBlade performance metrics for indexing benchmarks

Comparison with DAS SSD

We compared FlashBlade to a traditional architecture with local SSDs (DAS). While both configurations scale linearly, the FlashBlade configuration has slightly higher ingest performance than DAS. Note that at the 40-node configuration, the 15 blade FlashBlade system is periodically pushed to performance limits, indicating that additional blades are needed beyond this scale.

Testing on the same 40 nodes with DAS SSD demonstrates equal or slightly better performance with FlashBlade.

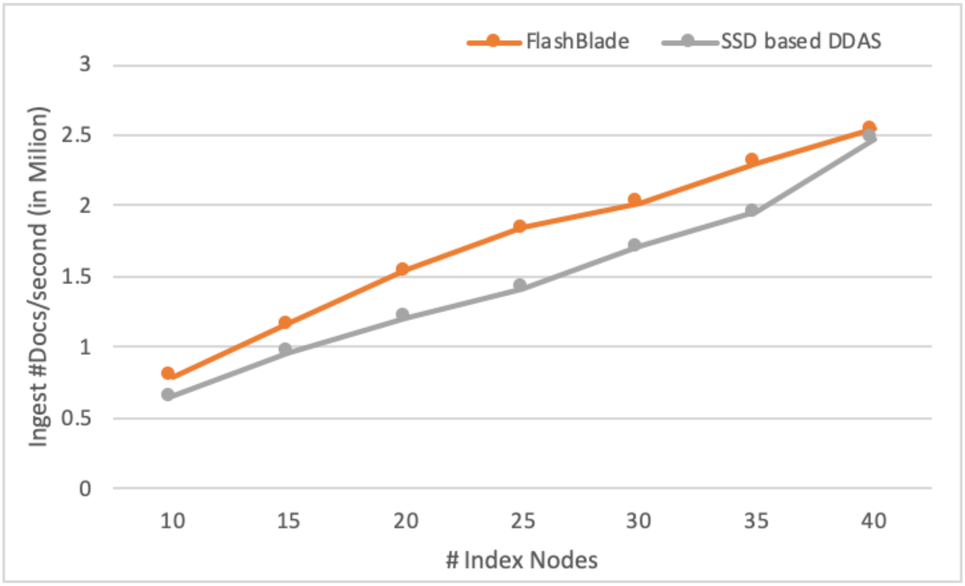


Figure 19: Node Scaling with FlashBlade vs SSD DDAS

Search Behavior

In addition to search API, Elastic provides search through a Kibana dashboard. We wanted to measure search performance during ingestion with variable hot and cold queries at random intervals to simulate typical search behavior. The below diagram is data processing flow during search operations:

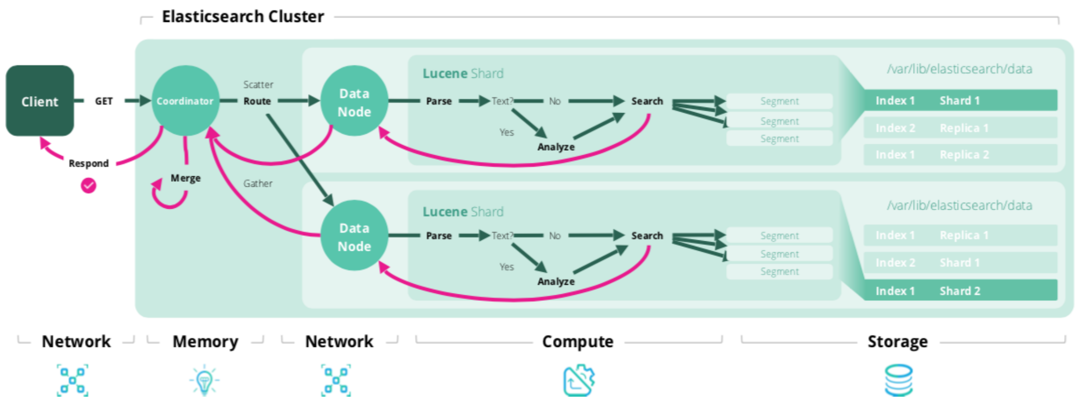


Figure 20: Elasticsearch Search Processing Flow

Search test overview:

Setup for search was based on the same 40 node Elastic clusters. We used the custom generated 16TB of Apache log data for our search queries and used the same index data as part of ingest testing.

Source datasets:

Unstructured dataset:
Apache Logs – custom generated with internal tool.
Raw Data: 16TB
Number of Documents: 75 Billion

Concurrent search test

Apache bench provided a simple http-based search request test, with the ability to set the number of concurrent sessions. Although these queries were identical, we felt it was adequate for testing basic concurrent search.

Apache Jmeter provided a way to test with variable query data in addition to proving concurrent search results. In addition, the results were fed into another instance of Elasticsearch for query and analysis in Kibana.

Search test results:

The performance difference when comparing search results for concurrent queries with and without indexing was minimal. This shows the ability for the solution to scale.

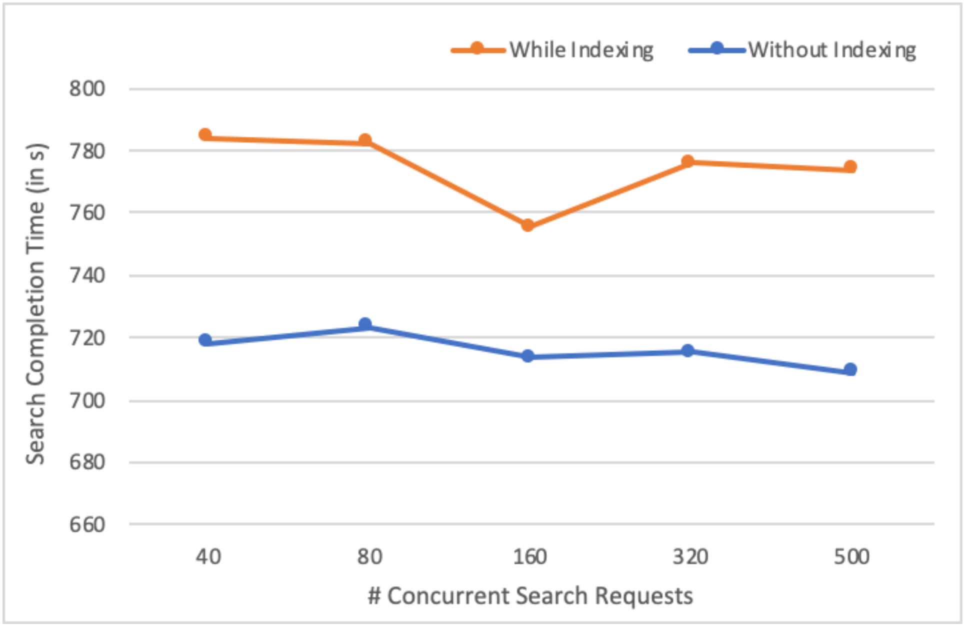


Figure 21: Search Performance with and without indexing with FlashBlade

Search with hot and cold queries:

In order to validate that FlashBlade could handle ad hoc queries requiring IO and CPU cycles, we introduced a unique test. While running hot queries in infinite loops, we introduced ad hoc, cold queries. The hot queries involved minimal IO in that they were one search term and recent date ranges. As these are frequent queries, Elasticsearch intelligently caches them in the cluster.

After a few minutes, we ran a few small batches of ad hoc queries for an older data range with a combination of one and two search terms to simulate an enterprise analytics search scenario. These queries are expected to take longer and consume higher IO and CPU cycles. The chart below demonstrates the impact of concurrent hot queries running in infinite loops when random ad hoc cold queries (4 sets) are run at intervals. Notice there is hardly a change in query performance. The bottom part of the image shows the performance metrics from the FlashBlade dashboard.

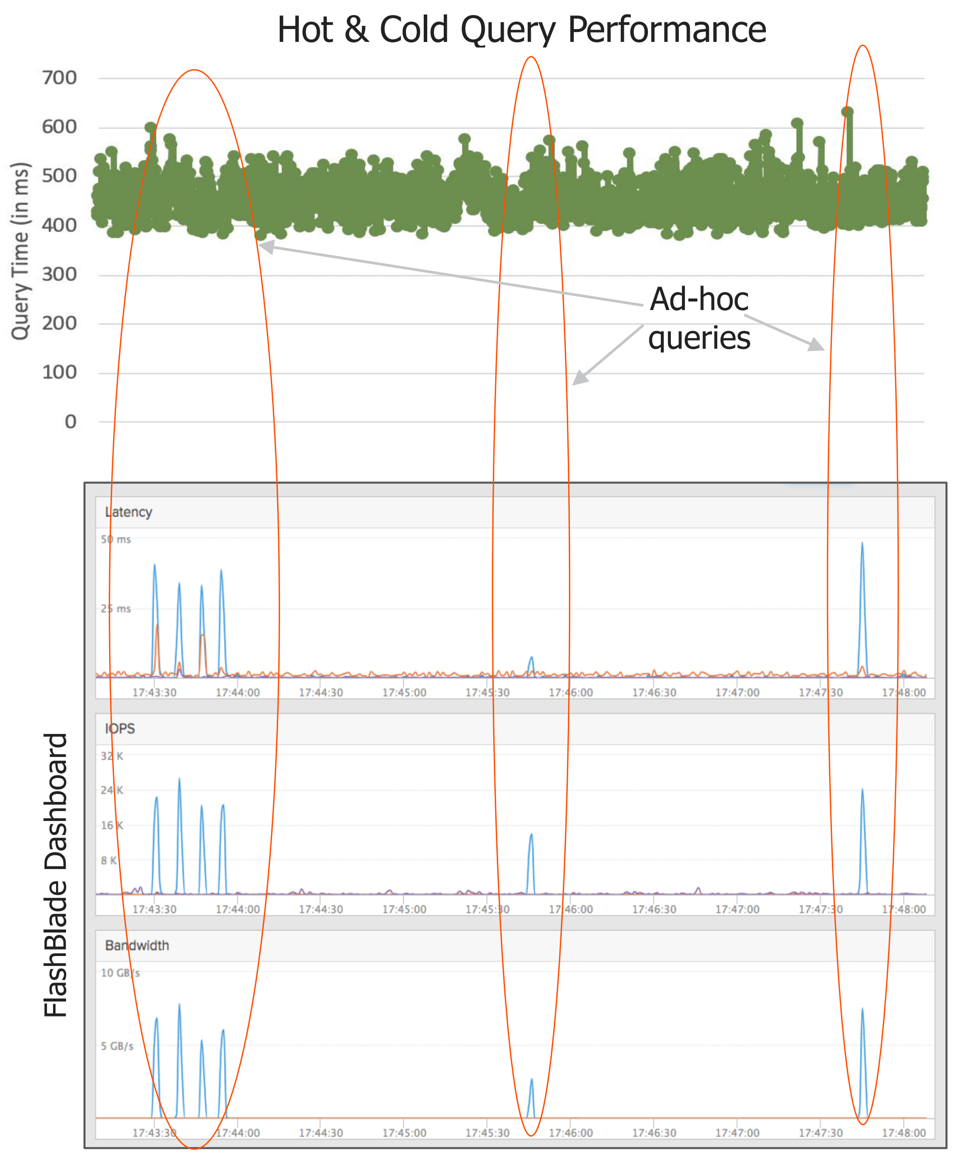


Figure 22: Hot and Cold Query Performance with FlashBlade

Operational Efficiency

To ensure that disaggregating storage and compute simplifies management, we tested failure scenarios.

Failure scenarios

- This phase tests the performance of Elastic cluster when node failures occur:
  We used the same dataset as NYC taxis dataset with 1 replica. To simulate a failure scenario, we powered off one of the Elastic nodes. Kibana dashboard immediately showed the Health of the Cluster as red. However, within a few minutes, the cluster was back to green as the replica shards recreated themselves as primary shards in another node.
- Node failure when there were no replica shards:
  In this scenario, we used the Apache logs dataset which weren't indexed with replica shards. So, when we recreated a scenario to fail a node, the data for that shard seemed to be lost. However, after we brought the node back up, it automatically picked up data from the mounted NAS folder without loss of data.

Out of storage scenario

- Storage scalability on live cluster.
  In this scenario, we used the allocated storage in FlashBlade so that only 3% was remaining. Naturally, the health status of Elastic showed first yellow and then red due to the low storage available for Elasticsearch. We were able to fix this issue in minutes by changing the allocation in the FlashBlade config screen. It was immediately reflected on the Linux nodes and the Elastic nodes showing green for indexes that were affected.

Appendix C: Best Practices and Additional Resources

Best practices

Pure FlashBlade

FlashBlade filesystems

- Create a single NFS filesystem on FlashBlade and use one unique subfolder for each data node as mount point. This helps track space and performance on a per filesystem basis in FlashBlade
- As FlashBlade file systems are always thin-provisioned, Elastic Administrators can provision a large sized file system to avoid updating the size to meet the growth
- It's not recommended to set hard limit parameters for the file system sizes as this will limit the flexibility of adding more space when needed
- Keep all the NFS file systems for all the indexers in a cluster at the same size

Linux Mount options

Use the following mount options to mount the NFS filesystem on the indexer nodes for the data nodes.
`rw,bg,nointr,hard,tcp,vers=3`

Don't specify the wsize option as the host can get the default size offered by FlashBlade (512K).

Linux nodes tuning

Elastic Nodes tuning:

1. Set Memory map count
For temporary change from cli: $sysctl -w vm.max_map_count=262144
Edit /etc/sysctl.conf add vm.max_map_count=262144
Verify:
To check $sysctl vm.max_map_count

2. Disable Swapping:
`$sudo swapoff -a`
==>Comment all lines with swap in `/etc/fstab`
`$sysctl vm.swappiness=1`

3. Memorylock setting:
Set `ulimit -l unlimited` as `root` before starting Elasticsearch,
or set `memlock` to `unlimited` in`/etc/security/limits.conf`

$ulimit -l unlimited

```
$ES_HOME/config/elasticsearch.yml file:
 bootstrap.memory_lock: true
```

To check

```
curl -X GET "hostname:9200/_nodes?filter_path=**.mlockall"
```

4. File Descriptors:

```
$ulimit -n 65535
```

or set nofile to 65535 in `/etc/security/limits.conf`.

To check

```
curl -X GET "10.21.236.35:9200/_nodes/stats/process?filter_path=**.max_file_descriptors"
```

5. Install nfs tools

On each node install nfs-tools

```
yum install -y nfs-utils
systemctl enable nfs
systemctl restart nfs
```

Replica nodes -rebalancing:

When a node fails, rebalance jobs start automatically but can be stopped by setting

```
Cluster.routing.enable = none
```

This is needed as data associated with the node that failed would still be available in FlashBlade. During our testing when we restored the nodes to normalcy, the data was available without rebalancing.

Elasticsearch cluster

- **Tunables**: Determine the ideal parameters (e.g. bulk size, shard count and bulk indexing clients) for your dataset.
- **Refresh interval**: Recommend refresh interval change from 1s – 30s
- **Pre indexing**: For range aggregations, reduce overhead by pre-indexing required data.
- **Thread pool setting**: Use thread pool properties for bulk indexing

Additional Resources

Important Elasticsearch configuration
https://www.elastic.co/guide/en/elasticsearch/reference/current/important-settings.html

Tuning for indexing speed
https://www.elastic.co/guide/en/elasticsearch/reference/master/tune-for-indexing-speed.html

Tuning for search speed
https://www.elastic.co/guide/en/elasticsearch/reference/master/tune-for-search-speed.html

Rally benchmark tool
https://esrally.readthedocs.io/en/stable/

Appendix D: Installing esrally benchmark tool

Rally is a public tool provided by Elastic to benchmark Elasticsearch. The goal of this setup is to run esrally to benchmark an existing cluster.

Plan to setup few nodes for esrally say 5 nodes. Nodes 10.5.5.5 - 10.5.5.10
Consider that we already have Elastic nodes running on 10.5.5.11 – 10.5.5.15. These are the benchmark candidate machines.

In the first machine we install Rally as user elastic, we will call it the benchmark coordinator.
Start the esrally daemon

```
$esrally start –node-ip=10.5.5.5 –coordinator-ip=10.5.5.5
```

The other nodes are rally driver nodes. Install rally on all the other nodes as user elastic.
On each of these nodes start the esrally daemon:

```
$esrally start -node-ip=10.5.5.6 -coordinator-ip=10.5.5.5
$esrally start -node-ip=10.5.5.10 -coordinator-ip=10.5.5.5
```

To run the benchmark:

```
esrally --track=nyc_taxis --target-hosts=10.5.5.11:9200,10.5.5.12:9200,10.5.5.13:9200,1
0.5.5.14:9200,10.5.5.15:9200 --pipeline=benchmark-only
```

Esrally daemon can be stopped using command

```
$esrallyd stop
```

Appendix E: Capacity and Sizing

Any production deployment of the Elastic Stack should be guided by capacity planning for Elasticsearch. Whether you use it for logs, metrics, traces, or search, and whether you run it yourself or in our cloud, you need to plan the infrastructure and configuration of Elasticsearch to ensure the health and performance of your deployment.

Capacity planning is about estimating the type and amount of resources required to operate an Elasticsearch deployment. Key factors are:

- Basic computing resources
- Architecture, behaviors, and resource demands of Elasticsearch
- Methodologies to estimate the requirements of an Elasticsearch deployment

The four basic computing resources are: Storage, where data persists; Memory, where data is buffered; Compute, where data is processed; Network, where data is transferred.

High performance NAS (FlashBlade) meets the need for storage.

JVM heap stores metadata about the cluster, indices, shards, segments and file data. We recommend up to 50% of available RAM with a max of 31GB RAM to avoid garbage collection. Elasticsearch will use the remainder of available memory to cache data, improve performance dramatically by avoiding disk reads during full text search, aggregations on doc values and sorts.

Elasticsearch nodes have **thread pools** and **thread queues** that utilize the available compute resources. The quantity and performance of CPU cores governs the average **speed** and peak **throughput** of data operations in Elasticsearch.

For very large deployments, the amount of data transfer for ingest, search, or replication between nodes can cause **network saturation**. In these cases, network connectivity can be upgraded to higher speeds, or the Elasticsearch deployment can be split into two or more clusters and then searched as a single logical unit using **cross-cluster search** (CCS)

There are two basic sizing methodologies that span the major use cases of Elasticsearch.

- Volume: Estimating the storage and memory resources required to store the expected amount of data and shards for each tier of the cluster.
- Throughput: Estimating the memory, compute, and network resources required to process the expected operations at the expected latencies and throughput for each operation and for each tier of the cluster.

Volume Sizing: Data Volume

Discovery questions:

- How much raw data (GB) will you index per day?
- How many days will you retain the data?
- How many replica shards will you enforce?
- How much memory will you allocate per data node?

In addition, it’s a best practice to reserve the equivalent of a data node to handle failure.

Total Data (GB) = Raw data (GB) per day * Number of days retained * Net expansion factor * (Number of replicas + 1)
Total Storage (GB) = Total Data (GB) * (1 + 0.15 Disk watermark threshold + 0.05 Margin of error)
Total Data Nodes = ROUNDUP(Total Storage (GB) / Memory per data node / Memory:data ratio) + 1 Data node for failover capacity

Compression:

Elasticsearch always compresses the data before writing to the disk. There are two modes of compression: a default option with minimal impact on the performance or *best_compression* with higher efficiency but consumes considerable CPU cycles and may impact indexing performance. For more optimal storage, it is recommended to enable *best_compression*.

Volume Sizing: Shard Volume

Discovery questions:

- How many index patterns will you create?
- How many primary and replica shards will you configure?
- At what time interval will you rotate the indices, if at all?
- How long will you retain the indices?
- How much memory will you allocate per data node?

In general, it is a good idea to not exceed 40-50GB per shard.

Tip: Collapse small daily indices into weekly or monthly indices to reduce shard count. Split large (>40GB) daily indices into hourly indices or increase the number of primary shards.

Total Shards = Number of index patterns * Number of primaries * (Number of replicas + 1) * Total intervals of retention
Total Data Nodes = ROUNDUP(Total shards / (20 * Memory per data node))

Throughput Sizing: Search Operations

Search use cases have targets for search response time and search throughput in addition to the storage capacity. These targets can demand more memory and compute resources.

Too many variables affect search response time to predict how any given capacity plan will affect it. By empirically testing search response time and estimating the expected search throughput, we can estimate the resources required for the cluster to meet the demands.

Discovery Questions

- What is your peak number of searches per second?
- What is your average search response time in milliseconds?
- How many cores and threads per core are on your data nodes?

Theory of the Approach

Rather than determining how resources will affect search speed, use search speed as a constant by measuring it on your planned hardware. Then determine how many cores are needed in the cluster to process the expected peak search throughput. Ultimately the goal is to prevent the thread pool queues from growing faster than they are consumed. With insufficient compute resources, search requests risk being dropped.

Peak Threads Size Total Data Nodes = ROUNDUP(Peak searches per second * Average search response time in milliseconds / 1000 Milliseconds)

Thread Pool = ROUNDUP((Physical cores per node * Threads per core * 3 / 2) + 1)

Total Data Nodes = ROUNDUP(Peak threads / Thread pool size)

© 2019 Pure Storage, Inc. All rights reserved. Pure Storage, Pure1, Pure1 Meta, Pure-On-The-Go, the P Logo, AIRI, the AIRI logo, CloudSnap, DirectFlash, Evergreen, FlashBlade and FlashStack, and ObjectEngine are trademarks or registered trademarks of Pure Storage, Inc. in the U.S. and other countries. All other trademarks are registered marks of their respective owners.

The Pure Storage products and programs described in this documentation are distributed under a license agreement restricting the use, copying, distribution, and decompilation/reverse engineering of the products. No part of this documentation may be reproduced in any form by any means without prior written authorization from Pure Storage, Inc. and its licensors, if any. Pure Storage may make improvements and/or changes in the Pure Storage products and/or the programs described in this documentation at any time without notice.

THIS DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID. PURE STORAGE SHALL NOT BE LIABLE FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS DOCUMENTATION. THE INFORMATION CONTAINED IN THIS DOCUMENTATION IS SUBJECT TO CHANGE WITHOUT NOTICE.

Pure Storage, Inc.
650 Castro Street, #400
Mountain View, CA 94041

PS1023-01

purestorage.com

800.379.PURE

PURESTORAGE®