The Case for On-Premises Hadoop with FlashBlade

FlashBlade is a high-performance, scale-out architecture that challenges the traditional view of shared storage and NFS as too slow or unreliable for big-data analytics.

A follow-up post explains in detail how to configure MapReduce for FlashBlade and compares the performance of FlashBlade versus traditional HDFS.

In short, a high-performance, scale-out NFS service both improves MapReduce performance and greatly simplifies operating and scaling the storage tier of a Hadoop cluster:

- When you connect to a high-performance storage device like FlashBlade that delivers 5GB/s write throughput and 15GB/s read throughput in 4RU, MapReduce on NFS can fully utilize that performance.
- MapReduce reads and writes to NFS using standard filesystem operations, leveraging the kernel NFS client, with no complicated configuration or non-standard code required.
- A representative wordcount mapreduce showed a 31% speedup versus HDFS backed by local SSDs.
- Replacing HDFS with shared storage allows separate scaling of compute and storage tiers, allowing either to grow depending on need.
- Simplify administration by reducing the number of failing components from a large number of commodity parts to a denser, more reliable networked appliance.
- Scale metadata with a general-purpose and flexible shared storage tier without the

metadata/namenode limitations of HDFS.

Disclaimer: I helped build FlashBlade.

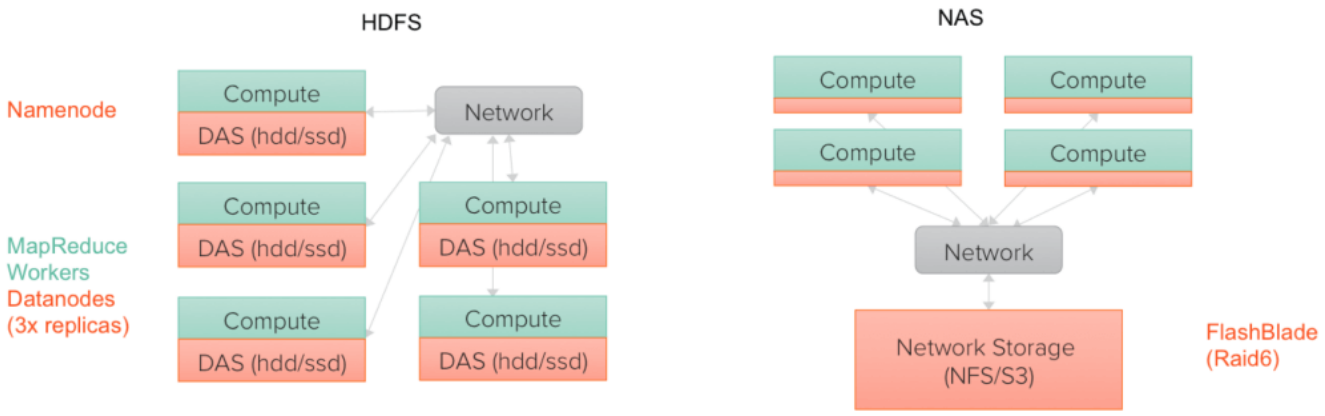
This article will dive deeper into the requirements and candidates for Hadoop's storage tier and look at why it makes sense to replace HDFS with a high-performance, scale-out storage device like FlashBlade.

There have been others who have suggested the benefits of non-HDFS alternatives, most notably AWS's S3 (more here), but also GlusterFS and proprietary solutions like MapR-FS, but none of these have the small-file performance, general flexibility, and wide adoption of NFS and the high-performance, scale-out architecture of FlashBlade.

Cluster Architecture

NFS over FlashBlade can either be used in full on-premises environments or hybrid clouds environments where EC2 compute instances access the storage device using AWS direct connect. This article focuses on MapReduce, but the limitations and inflexibility of HDFS hinder other applications, like Apache Spark, in the same way.

The figure below contrasts the architectures of HDFS and shared storage (NAS). The key difference is the unit of scaling; in an HDFS cluster, each node is both compute and storage and hence scaling the cluster means always scaling both proportionally. In contrast, the shared storage architecture allows adding additional compute nodes separately from the storage tier.



Storage Layer Requirements

The Hadoop project combines three main components: a storage tier (HDFS), a parallel computation engine (MapReduce), and a resource scheduler (YARN). MapReduce simplifies large, batch computations by allowing the programmer to write single-threaded code and it automatically parallelizes the computation to run on 1000s of machines. Though the default in Hadoop, HDFS can be replaced by a different storage tier such as a networked-filesystem (e.g., NFS) or object store (e.g., S3). Each job's storage requirement depends on data size and the computational complexity of the task, but IO-dominant workloads require reading and writing data at 1) high aggregate bandwidth, 2) predictable performance and 3) high concurrency.

- High aggregate bandwidth is required to keep both the map and reduce workers busy. If the read and subsequent write throughput is too low, the worker resources will idle waiting for IO. By

growing clusters to 100s and 1000s of nodes, the aggregate compute resources can consume and produce 10s to 100s of GB per second.
- Predictable performance is key to reducing the impact of tail latencies. Because the batch MapReduce does not finish until all tasks have finished, consistent performance for all nodes is required to avoid a small number of slow workers slowing down the whole job.
- The high concurrency requirement arises from the greatest advantage of MapReduce: the ability to easily scale a computation by adding nodes to a cluster. But MapReduce's promise of linear performance scaling only works if both the compute and storage tiers scale linearly.

Protection against failures is also required from the storage tier. The MapReduce model helps prevent transient failures from aborting an entire, long-running job by retrying individual failed tasks. Most MapReduce input data is very expensive or impossible to reproduce if lost so the storage tier must also provide durable storage.

There are several critical requirements from general-purpose filesystem that are missing: efficient storage of small files, very large numbers of files, and overwrite support. Though not required for MapReduce, these features greatly increase the flexibility to use multiple and new applications (e.g., Apache Spark) with the same datasets; MapReduce is just one tool in a quickly growing space.

HDFS: Motivations and Relevance Today

Hadoop's HDFS design is inspired by the 2003 "Google File System" paper, which described a system for clustering large numbers of commodity servers with unreliable hard disks together into a reliable and high performance storage tier. Namespace metadata is managed by a single namenode server and data blocks are highly replicated across datanodes in the cluster to provide both aggregate bandwidth and fault tolerance.

When HDFS was originally created, its initial assumptions included:

- "Hardware failure is the norm rather than the exception."
- "A typical file in HDFS is gigabytes to terabytes in size. Thus, HDFS is tuned to support large files."
- "The emphasis is on high throughput of data access rather than low latency of data access."
- "HDFS applications need a write-once-read-many access model for files."
- "It should support tens of millions of files in a single instance."
- "The assumption is that it is often better to migrate the computation closer to where the data is located rather than moving the data to where the application is running."

Since the original design over 10 years ago, several underlying premises in the datacenter have changed or shifted:

- Network and IO speeds have increased dramatically while CPU speeds have not. This challenges the original HDFS assumption that moving the compute to the data is faster than moving the data to the compute.
- Flash storage is catching up to unreliable spinning hard-drives in price and far exceeding them in speed, density, power-efficiency, and reliability (more here). This changes fundamental characteristics of the underlying storage media.
- Frameworks for distributed processing of streaming data sources (e.g., Spark) are providing a superset of functionality of batch-processing paradigms such as MapReduce. This results in more diverse file types, sizes, and access patterns.

The largest fundamental issue with the implementation of HDFS is the serialization of metadata accesses on the single namenode. Each client read must access the namenode in order to direct a block operation (read/write) to the nearest datanode with the desired block. This means all data accesses, whether read or write, must go through the namenode. This makes the namenode a performance bottleneck in larger systems or systems with many small files. There has been effort to eliminate the namenode as a single-point-of-failure (more here), but it is still a performance chokepoint.

Additional benefits of moving to a networked storage tier like NFS:

- The first and foremost advantage of using a dedicated storage tier or appliance is decoupling compute and storage. Traditional HDFS ties together the CPU and storage, forcing an investment in both tiers even if only one is desired. A dedicated storage tier can scale independently of the number of compute nodes.
- Simplify administration by consolidating storage to dedicated storage devices instead of a large number of independent servers with unreliable hardware required for HDFS.
- Finally, a general purpose storage tier (filesystem or object) provides flexibility for future uses of new data and new applications. For example, a major hurdle is HDFS's handling of small files and metadata by a single point-of-control namenode.

When expanding a cluster, adding HDFS nodes adds both compute and storage proportionally.

By decoupling, you can add compute by the addition of new nodes with only minimal local storage. The storage tier scales either organically in a scale-out storage system, e.g. adding additional blades, or by mounting multiple storage systems.

There are other features of modern storage appliances that HDFS can or will be able to match. Many NAS systems use erasure-coding to protect user data without the inefficiencies of mirroring. In the upcoming Hadoop 3.0 release, HDFS is planned to support erasure-coding which would protect data from node failure with more efficient use of space than 3x replication. Additionally, modern storage appliances support inline compression which is also possible to enable to HDFS.

Conclusion

HDFS was originally designed 15 years ago to solve an application-specific set of storage problems.However, today's data centers must deliver faster networking and lower storage latency to accommodate the demands of mixed workloads and web-scale applications. With the introduction of scale-out, all-flash file and object storage platforms like FlashBlade, NFS-based storage is able to offer 1) similar or better performance, 2) higher densities, 3) disaggregation of compute and storage, and 4) simpler administration versus traditional HDFS.

Read the second blog post, which explores the details of how to run MapReduce on FlashBlade and presents performance comparisons against HDFS backed by local SSDs.