Scalable, reliable, and efficient object storage for Hadoop

Greg Dhuse
Cleversafe
Outline

- The case for object storage with Hadoop
  - Specifically, dispersed object storage
- Integration options
  - Object interface vs. drop-in replacement for HDFS
- Data-local computation and information dispersal
Popular open-source framework for distributed computation

- Commercialized by Cloudera and others
- Originally just MapReduce, but branching out to other paradigms with YARN

Flourishing ecosystem

- Open source & commercial products

Mantra: Take the computation to the data
Hadoop Architecture
Limitations of HDFS

- Master-slave architecture: NameNode
  - Point of failure: Previously a single point of failure, now a clustered point of failure with HA
  - Scalability bottleneck: In the I/O path. NameNode federation helps, but introduces administrative headaches and increases failure footprint
- Designed around replication
  - High storage overhead
What is Object Storage?

Object storage = Data payload +
(optional) Metadata + (optional) Indexing

Data payload

- Typically large (1MB++)
- Efficient streaming writes, no random offset writes
- Efficient reads
  - Streaming, range, offset
What is Object Storage?

Object storage = Data payload + (optional) Metadata + (optional) Indexing

Metadata (optional)

- Object name
  - If indexed, can be treated like a file system path at lookup time
- Typical file system metadata
- User-defined metadata
What is Object Storage?

Object storage = Data payload +

(optional) Metadata + (optional) Indexing

Indexing (optional)

- Index objects by arbitrary key
  - Name, metadata field, compound keys
- Index within objects
- Build indexes at write time or later (_optional)
- No centralized database or index controllers
Why Object Storage?

- Flexibility
  - Mix & match payload, metadata, indexing
  - Many access methods (vs. HDFS)
- Concurrency
  - Payload is write once, read many
  - Metadata & index can use lockless algorithms
- Object interfaces are simpler
  - But developers are used to POSIX
- Many applications don’t need a file system
  - May already use a database
Object Access Flexibility

Application Layer

- Application Server
  - Write by Object-ID
  - or
  - Write by Name

Access Method

- REST
- S3, CDMI, legacy fs

Access Gateways

- Object ID
  - Object data
  - Object & Metadata
  - Object data
  - Object & Metadata

Storage & compute nodes

- Site 1
- Site 2
- Site 3
- Site n
Scalability

Flat namespace

- Namespace is a mapping from node to name range
- Used by the client and servers for lookup
- Objects are randomly named and evenly distributed
- No master controllers, no databases, no NameNode
- Nodes that are physically close appear nearby in the namespace
  - Store data on opposite sides of the namespace for maximum separation
- Inspired by Distributed Hash Tables
Dispersed Object Storage

- Improved Efficiency & Reliability for Object Storage
- Instead of replication or RAID, Information Dispersal
  - An Information Dispersal Algorithm (IDA)
    - Receives $K$ inputs, produces $N$ outputs
    - Can recover input from any $K$ of the $N$ outputs
    - Where $1 \leq K \leq N$
  - In essence: an IDA is forward error correction combined with a slicing and spreading operation
IDA example: $K = 10$, $N = 16$
Storage Efficiency

- Bytes stored: N/K * object size
  - 1.0 < factor < 2.0
- Factor approaches 1.0 for fixed failure tolerance (N-K) as N increases
  - Limited memory constrains N in practice
Reliability

Annual Chance of Data Loss (Enterprise Drives)

- 20/16 Dispersal
- 8+2 RAID 6
- 4+1 RAID 5
- 3x Replication

System Size (TB):

- 1,080
- 5,400
- 10,800
- 21,600
- 27,000
- 32,400
- 37,800
- 43,200
- 48,600
- 54,000
- 59,400
- 64,800
- 70,200
- 75,600
- 81,000
- 86,400
- 91,800
- 97,200
- 102,600

Probability:

- 0%
- 10%
- 20%
- 30%
- 40%
- 50%
- 60%
- 70%
- 80%
- 90%
- 100%
Apply the IDA to Everything

- **Metadata**
  - Metadata stored as dispersed objects
  - Lockless updates

- **Indexing**
  - Index data stored as dispersed objects
  - Many concurrent readers & writers
  - Client-side caching

- For details, come to: **Scaling Metadata into the Exabytes**
## Object Storage vs. HDFS

<table>
<thead>
<tr>
<th></th>
<th>HDFS-0.23 and beyond</th>
<th>Dispersed Object Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Scalability</strong></td>
<td>Federation allows administrators to manually split namespace among a small number of nodes</td>
<td>Dispersed metadata allows all nodes to be treated equally. No special metadata nodes to spec, configure &amp; monitor</td>
</tr>
<tr>
<td><strong>Reliability / Availability</strong></td>
<td>High Availability fail-over to a backup NameNode</td>
<td>IDA offers configurable reliability &amp; availability with seamless handling of multiple simultaneous failures</td>
</tr>
<tr>
<td><strong>Cost Efficiency</strong></td>
<td>Replication (3x default)</td>
<td>IDA provides better reliability and less cost</td>
</tr>
</tbody>
</table>
HDFS at Yahoo!

Based on a case study presented at Hadoop World 2011 by Suresh Srinivas of Hortonworks

IDA Configuration: 26/18, 1.44x expansion factor

Reliability

- 10 clusters with a total of 20K data nodes in 2009 (19.6 PB usable)
- Lost 19/329M blocks due to NameNode failures

  Yahoo! lost over 1 GB of data in 1 year, even with replication

Mean Time to Data Loss (MTTDL): 390,000 years

Availability

- 25 clusters with 22 NameNode outages in 2009, only 8 of which would be avoided with HA (2 9’s of availability)

Availability: 9 9’s of availability

Efficiency

- Assuming default 3x replication and 64MB block size: 58.8 PB raw storage

Efficiency: 1.44x expansion factor = 28.2 PB raw

Reduction of cluster to 9600 nodes from 20K
Object Storage & Hadoop

- Option 1: Object-based access
  - Benefit: Allows full flexibility of data, metadata, and indexing
  - Drawback: Interface is not compatible with HDFS
    - Requires changes to existing MapReduce jobs
  - Accomplished using custom InputFormat & OutputFormat
Replacing HDFS

- Option 2: Drop-in replacement for HDFS
  - Benefit: Fully compatible with HDFS, so existing jobs can run without modification
  - Drawback: File system paradigm is less flexible & powerful

- Accomplished by implementing Hadoop’s FileSystem interface
  - File system semantics are emulated using object storage, metadata, and indexing
  - For more details come to: Bridging POSIX-like APIs and Cloud Storage
One problem: How can we do data-local computation on dispersed objects?

- Each object is split into N slices and can be restored from any K
- Need to adhere to the mantra: Map tasks should read locally, not from the network

Solution: At write time, project input so that after dispersal raw data falls in contiguous chunks on K storage nodes
Data Local Computation & the IDA

- At read time: Hadoop tasks read locally from raw slices, bypassing IDA reconstruction
  - Fall back to full IDA reconstruction on error

- Best of both worlds
  - On a healthy system, reads for each MapReduce task can be satisfied locally, almost identically to HDFS
  - Full reliability/availability of dispersal in case of failure
Dispersal Pipeline for Hadoop

Data projection → Segmentation → IDA

Raw data stream → Compute optimized data chunks → Segmentation metadata & 1MB+ segments

Storage nodes

Storage nodes

Useful slices
HDFS Data Layout

Source data:
The quick brown fox jumped over the lazy dog.

Store 1

Store 2

Store 3

Chunk 1
Read for Task 1 (64MB)

Chunk 1
Write 1 (64MB * 3x)
Dispersed Data Projection

Source data:

The_quick_brown_fox_jumped_over_the_lazy_dog.

ChunkSet 1

Chunk 1

Chunk 2

ChunkSet 1 (128MB - 2GB+)

Store 1

Store 2

Store 3

<coded slices>

<coded slices>

Chunk I

Read for Task I (64MB)

Jump

ed_over_

Segment I

Write I (1MB)
Storage Node Architecture

- Hadoop MapReduce computation runs directly on storage nodes
- Jobs are assigned to nodes for local data access
  - Calculate custom InputSplits based on object namespace and data projection
Dispersed Object Storage for Hadoop

- What changes
  - With Option 1: Object storage interfaces used by MapReduce jobs

- What stays the same
  - With Option 2: Same FileSystem interface as HDFS. Dispersed Object Storage used transparently from MapReduce jobs
  - Compatible with existing Hadoop distributions and many existing tools & projects
  - Compute administration done using existing tools
IDR reduces the effective chunk size for objects that are smaller than one full chunk set: $K \times \text{chunk size}$

- Example: $K=10$, 64MB chunk size, 640MB chunk set size. A 500MB object would only have 50MB chunks
- Also applies to the last chunk set of any object

- Write projection requires substantial memory or an on-disk cache, which may hurt performance
Questions & Answers

Questions?
One bonus feature: Build & use Object Storage indexes from Hadoop jobs

Build indexes on data using Indexing APIs from MapReduce jobs

- Analyze and index data in parallel using index APIs
- Search and query your indexed data

Use indexes in MapReduce jobs to efficiently find the data you need to process

- Index data and metadata at ingest or later using MapReduce
- Query the index directly from MapReduce jobs to find the data you need to analyze
- Perform targeted analysis on only the relevant data