

Notes on Parallel File Systems: HDFS & GFS
15-440, Fall 2012
Carnegie Mellon University
Randal E. Bryant

References:

Ghemawat, Gobioff, Leung, "The Google File System," SOSP '03

Shvachko, Kuang, Radia, Chansler, "The Hadoop Distributed File System"
MSST '10

Chang, Dean, ... "BigTable: A Distributed Storage System for
Structured Data," OSDI '06, TOCS '08

Borthakur, et al., "Apache Hadoop goes realtime at Facebook",
SIGMOD '11

Both GFS & HDFS designed originally with similar goals:

- * High throughput (latency less important)
 - Designed for batch processing jobs, e.g., MapReduce
- * High capacity (large block size). Typical files > 1GB
- * High scalability. Handle > 10⁷ files
- * Reliability through replication. Treat failure as normal.

Although GFS was developed first, HDFS is much simpler, and so will describe it first.

HDFS

Based on "write once, read many" model:

- * Each file has single writer
- * Originally: File fully written and closed before any reader given access
- * Now: After file written, can reopen and append to it
 - Still guarantees that never mutate file

According to 2010 paper:

- * Yahoo's largest cluster has 3500 nodes
- * Yahoo supplied 80% of engineering effort on HDFS

Components:

- * Clients.
- * NameNode. Single node containing all metadata about all files
- * DataNodes. Set of nodes that store actual file contents.pp
- * CheckPointNode. Creates disk image of NameNode state
- * BackupNode. Creates shadow image of NameNode state

* File represented as sequence of blocks of fixed size (64 or 128 MB).
(Given byte offset for read, can immediately determine which block to read.)

- * Each block has unique block ID.
- * Blocks are distributed across multiple DataNodes to enable parallel access.
- * Blocks replicated (default 3X) to enable recovery when DataNode fails.
 - Thought question: What is advantage of replication to using RAID?
- * When block created, NameNode decides placements
 - Default: two within single rack, third on a different rack
 - Access time / safety tradeoff.

NameNode

- * Metadata: Information about file + plus set of block IDs +

- location of all replicas of all blocks
- * Treats each FS operation as transaction
 - Maintains all information in memory
 - Logs to EditLog file
 - Must do write to editlog before file operation considered to be completed.
 - Can group updates from independent clients.
 - (Possibly outdated) backup copy stored on disk.
 - Can bring this copy up to date by replaying entries in EditLog
- * Periodic checkpoint:
 - Done by CheckPoint Node
 - Apply transactions in editlog to disk image
 - Delete old parts of editlog
 - Can do in background while other updates occurring.
 - Does not store locations of replicas
- * Tries to satisfy given read request with nearby DataNode
 - Same node / Same rack / Same system

DataNode

- * Uses its local file system to store blocks
- * Each block has two files
 - Actual data
 - Metadata: checksum, generation stamp (to detect stale copies)
- * Has no understanding of overall FS semantics
- * Periodically (every hour) sends "block report" to NameNode, containing information about all replicas it holds.
- * More often, sends heartbeat message to NameNode. (Default: every 3 seconds)

Client

- * Buffers file as it is being written
- * Create another block only when reach threshold
- * Once time to push data to DataNodes, sets up pipeline from client, through each replica's DataNode.
- * File not committed until closed.
- * Read: Retrieve list of blocks and where replicas are available
 - Subsequent reads involve direct interaction between client & datanodes.
- * API exposes block replica locations
 - E.g., so that MapReduce can schedule a task near a copy of its data.

Interactions

- * Client & DataNodes communicate to NameNode via RPC

Failures

- * DataNode
 - Detected by NameNode when DataNode fails to send heartbeat messages (default: 10 minutes)
 - NameNode will decrement replica counts for each of its blocks
 - Will cause replication to commence
 - * NameNode
 - Point of high vulnerability
 - Requires manual intervention
 - 1-3 hours of effort.
 - Must rebuild memory image of metadata
 - Must build map of replicas from DataNodes (poll DataNodes for their block info).
 - Recently: Availability of BackupNode means that only have to get maps from datanode
- s
- (Still requires ~20 minutes for failure recovery)

Some statistics

Facebook, 2010 (Largest HDFS installation at the time)

2000 machines, 22,400 cores
24 TB / machine, (21 PB total)

Writing 12TB / day
Reading 800TB / day
25K MapReduce jobs / day
65 Million HDFS files
30K simultaneous clients.

NameNode biggest impediment to scaling

- * Performance bottleneck
- * Holds all data structures in memory
- * Takes long time to rebuild metadata
- * Must vulnerable point for reliability

Current workaround

- * Support multiple name spaces
 - Not ideal from application perspective
- * Each has separate NameNode
- * Share DataNodes
 - Each data block labeled by a group ID

HDFS reliability at Yahoo 2009

Created 329M blocks on 10 clusters with total of 20K data nodes

650 lost blocks:

- * 533 Orphans from dead clients
- * 98 where user had specified that should only have 1 replica.
- * 19 lost due to software bugs (these are the more serious ones.)

HDFS availability

22 NameNode failures over 25 clusters in 18 mos.
Givens MTBF \approx 600 days
1-3 hours to recover

Assuming 3 hours to recover, this gives 0.9998 availability.
(OK, but being out of commission for 3 hours is not good.)

GFS

Supports mutable files:

- * Writes to arbitrary position
 - Special case: single writer append
- * Record append
 - Multiple writers
 - Atomic, concurrent append
 - Each record will appear in file at least once
 - May have duplicate records
 - File may also contain padding & record fragments
 - Useful for implementing log files
- * Snapshots
 - Can quickly make copy of any file
 - Uses copy-on-write, similar to AFS

Same general idea as HDFS (because Hadoop developers read papers about GFS):

- * Data divided into "chunks" of 64MB each
- * Single master node, many chunk servers

Interesting features

- * Clients get cached copy of metadata via leases (reduces load on master)
- * Replicas migrate
- * Log file from master replicated on remote machine
- * Automatic failover of master ("10s of seconds")

Supporting arbitrary writes

- * One replica designated "primary" via lease
- * It determines the serialization of writes to a file

Supporting record appends

- * If not enough room within chunk, then pad rest of chunk and retry with new chunk
- * Possible to create duplicate or fragment of record if failure occurs while writing
- * May have different versions on different replicas, but they will all have at least one copy of each record, in a unique order.

Limitations of GFS

- * Single master is serious performance bottleneck
 - MapReduce: Create many files at once
 - Have systems with multiple master nodes, all sharing set of chunk servers. Not a uniform name space.
- * Large chunk size. Can't afford to make smaller, since this would create more work for master.
 - Mitigated by move to BigTable

Building on GFS: BigTable

GFS originally designed to support high-throughput, batch operations, e.g., MapReduce jobs

Later added BigTable. A "database"

- * Information stored as records (Rows) each containing set of fields (Columns).
 - Also support for maintaining multiple entries, each identified by time stamp
 - Each row or column identified by string key
- * Does not support relational operations
- * Provides record-level atomicity (not general transactions)

Implementation

- * On top of GFS
- * Basic data unit: "tablet"
 - 100MB - 200MB
 - Stores contiguous (by key) subset of rows in a table
 - Also used to build high-radix trees
- * Multiple "tablet servers"
- * Single master

- * Tablet represented in different ways:
 - Base level via "string to string table" SSTable
 - Immutable key/value storage
 - Sorted by key
 - Updates accumulated in log file
 - + Periodically perform "minor compaction"
 - + Generate SSTable from current log file
 - + Describes updates (including deletions) to set of existing SSTables
 - + Periodically perform "major compaction"
 - + Compress entire tablet into single SSTable
 - See that only uses immutable files (SSTable's) and append-only files (log files)

- * Table represented by 3-level hierarchy of tablets
 - High radix tree structure
 - Maximum capacity = 2^{62} bytes (~1 hexabyte)

Modifying file systems to support real-time applications

Both GFS & HDFS originally conceived to support background tasks. E.g.,

- * Generate search index for set of web pages (using MapReduce) every few hours/days
- * Analyzing log/click data

Underlying assumptions

- * Large file sizes
- * Throughput more important than latency
- * File system outage of 2+ hours acceptable

Now companies have applications that require immediate response

- * Real-time updating of search data
- * Personalized searching
- * Email, messages

HDFS

- * Driven by Facebook. Want to use HDFS to store Facebook messages
- * Created real-time failover for NameNode
 - Operate two "Avatar" NameNode
 - Primary operates as master
 - Standby kept up to date
 - * Receives duplicate messages from DataNodes
 - * Continuously reads copy of Primary's Editlog to keep own state up to date
 - Can transfer control from primary to standby in a few seconds

GFS

- * Replaced by "Colossus" ~2010. Sketchy information (Wired Magazine)
 - Eliminate master node as single point of failure
 - Reduce block size to 1MB