The Basic Hadoop Components

* Hadoop Common – libraries and utilities
* Hadoop Distributed File System (HDFS) – a distributed file-system
* Hadoop YARN – a resource-management platform, scheduling
* Hadoop MapReduce – a programming model for large scale data processing

HDFS (redundant, reliable storage)

MapReduce (cluster resource management & data processing)

YARN (cluster resource management)

Pig (data flow), Hive (sql), RT Stream, Graph (Storm, Graph), Service (HBase)

Applications and Framework

* HBase – a scalable data warehouse with support for large tables.
* Hive – a data warehouse infrastructure that provides data summarization and ad hoc querying
* Pig – A high-level data-flow language and execution framework for parallel computation
* Spark – a fast and general compute engine for Hadoop data. Wide range of applications – ETL, Machine Learning, stream processing, and graph analytics.

HDFS Design Concept

* Scalable distributed filesystem
* Distribute data on local disks on several nodes
* Low cost commodity hardware

Original HDFS Design Goals

* Resilience
* Scalable
* Application Locality
* Portability

Original HDFS Design

* Single NameNode
* Multiple DataNodes

Manage storage – blocks of data

Serving read/write requests from clients

Block creation, deletion, replication

HDFS in Hadoop 2

* HDFS Federation
* Benefits:

Increased namespace scalability

Performance

Isolation

* How its done

Multiple Namenode servers

Multiple namespaces

Block pools

* High Availability – redundant NameNodes
* Heterogeneous Storage and Archival Storage

ARCHIEVE, DISK, SSD, RAM\_DISK

MapReduce Framework

* Software framework – for writing parallel data processing applications
* MapReduce job splits data into chunks
* Map tasks process data chunks
* Framework sorts map output
* Reduce tasks use sorted map data as input
* Typically compute and storage nodes are the same
* MapReduce tasks and HDFS running on the same nodes
* Can schedule tasks on nodes with data already present

Original MapReduce Framework

* Single master JobTracker
* JobTracker schedules, monitors, and re-executes failed tasks
* One slave TraskTracker per cluster node
* TaskTracker executes tasks per JobTracker requests

YARN: NexGen MapReduce

* Main idea – Separate resource management and job
* Global ResourceManager (RM)
* NodeManager on each node
* ApplicationMaster – one for each application

Additional YARN Features

* High Availability ResourceManager
* Timeline Server
* Use of Cgroups
* Secure Containers
* YARN – web services REST APIs

NextGen Execution Frameworks

* Enter: Execution frameworks like YARN. Tez, Spark
* Support complex directed acyclic graph (DAG) of tasks
* In memory caching of data

Hadoop Execution Environment

* Layout of new frameworks (YRAN, Tez, Spark) in Hadoop environment.
* Optimization strategies used in new frameworks
* Examples illustrating use of Tez, Spark

YARN

* MapReduce
* Open source/commercial applications
* User developed applications
* Frameworks like Tez, Spark

Tez

* Dataflow graphs
* Custom data types
* Can run complex DAG of tasks
* Dynamic DAG changes
* Resource usage efficiency

Spark

* Advanced DAG execution engine
* Supports cyclic data flow
* In-memory computing
* Java, Scala, Python, R
* Existing optimized libraries

Hadoop Resource Scheduling

* Resource management
* Different kinds of scheduling algorithms
* Types of parameters that can be controlled

Motivation for Scheduling

* Various execution engines/options
* Scheduling, Performance
* Control of resources between components

Schedulers

* Default – First in First out (FIFO)
* Fairshare
* Capacity

Capacity Scheduler

* Queues and sub-queues
* Capacity Guarantee with elasticity
* ACLs for security
* Runtime changes/draining apps
* Resource based scheduling

Fairshare Scheduler

* Balances out resource allocation among apps over time
* Can organize into queues/sub-queues
* Guarantee minimum shares
* Limits per user/app
* Weighted app priorities

Hadoop Applications

* Overview of apps, high level languages, services
* Databases/Stores
* Querying
* Machine Learning
* Graph Processing

Databases/Stores

* Avro: data structures within context of Hadoop MapReduce jobs.
* Hbase: distributed non-relational database
* Cassandra: distributed data management system

Querying

* Pig: platform for analysing large data sets in HDFS
* Hive: Query and manage large datasets
* Impala: High-performance, low-latency SQL querying of data in Hadoop file formats
* Spark: General processing engine streaming, SQL, Machine Learning and graph processing

Machine Learning, Graph Processing

* Giraph: Iterative graph processing using Hadoop framework
* Mahout: Framework for machine learning applications using Hadoop, Spark

Apache Pig

* Pig components – PigLatin, and infrastructure layer
* Typical Pig use cases
* Run Pig with Hadoop integration
* Platform for data processing
* Pig Latin: High level language
* Pig execution environment: Local, MapReduce, Tez
* In built operators and functions
* Extensible

Pig Usage Areas

* Extract, Transform, Load (ETL) operations
* Manipulating, analysing “raw” data
* Widely used, extensive list at:

Apache Hive

* Query and manage data using HiveQL
* Run interactively using beeline
* Other run mechanisms
* Data warehouse software
* HiveQL: SQL like language to structure, and query data
* Execution environment: MapReduce, Tez, Spark
* Data in HDFS, HBase
* Custom mappers/reducers

Hive Usage Areas

* Data mining, analytics
* Machine Learning
* Ad hoc analysis

Summary

* Used beeline for interactive Hive example
* Can also use

Hive command line interface (CLI)

Hcatalog

WebHcat

Apache HBase

* Hbase features
* Run interactively using HBase shell
* List other access mechanisms
* Scalable data store
* Non-relational distributed database
* Runs on top of HDFS
* Compression
* In-memory operations: MemStore, BlockCache

HBase Features

* Consistency
* High Availability
* Automatic Sharding
* Replication
* Security
* SQL like access (Hive, Spark, Impala)

Summary

* We used: Apache HBase Shell
* Other options:

HBase, MapReduce

HBase API

HBase External API

HDFS Design Factors

* Hundreds/Thousands of nodes

Need to handle node/disk failures

* Portability across heterogeneous hardware/software
* Handle large data sets
* High throughput

Approach to meet HDFS design goals

* Simplified coherency model0 write once read many.
* Data Replication – helps handle hardware failures
* Move computation close to data
* Relax POSIX requirements – increase throughput

Summary of HDFS Architecture

* Single NameNode – a master server that manages the file system namespace and regulates access to files by clients
* Multiple DataNodes – typically one per node in the cluster. Functions:

Manager storage

Serving read/write requests from clients

Block creation, deletion, replication based on instructions from NameNode

Performance Envelope of HDFS

* Able to determine number of blocks for a given file size
* Key HDFS and system components impacted by block size
* Impact of small files on HDFS and system

HDFS optimized for large files

* Key takeaway – lots of small file is bad
* Solutions:

Merge/Concatenate files

Sequence files

HBase, HIVE configuration

CombineFileInputFormat

Write/Read processes on HDFS

* Able to explain write process in HDFS
* Detail the replication pipeline process
* Explain read process in HDFS

Overview of HDFS Tuning

* Tuning parameters
* DFS Block Size
* NameNode, DataNode system/dfs parameters

HDFS Block Size

* Recall from previous video: impacts NameNode memory, number of maps tasks, and hence performance.
* 64MB is the default. Can be changed based on workloads. Typically bumped up to 128MB
* dfs.blocksize, dfs.block.size

HDFS Replication

* Default replication is 3.
* Parameter: dfs.replication
* Tradeoffs:

Lower it to reduce replication cost

Less robust

Higher replication can make data local to more workers

Lower replication => more space

Mitigation of common failures

* Periodic heartbeat: from DataNode to NameNode
* DataNodes without recent heartbeat:

Marked dead, no new IO sent

Blocks below replication factor re-replicated on other nodes

* Checksum computed on file creation
* Checksums stored in HDFS namespace
* Used to check retrieved data. Re-read from alternate reply need
* Multiple copies of central meta data structures
* Failover to standby NameNode – manual by default

Performance

* Recall: changing blocksize and replication factor can improve performance

Replication trade off w.r.t robustness

* Reducing replication has a trade off w.r.t robustness:

Might lose a node or local disk during the run – cannot recover if there is no replication

If there is data corruption of a block from one of the datanodes – again cannot recover without replication

Application programming interfaces

* Native Jave API: Base class org.apache.hadoop.fs.FileSystem
* C API for HDFS: libhdfs, header file (hdfs.h)
* WebHDFS REST API: HTTPGet, Put, Post and Delete operations

HDFS NFS Gataway

* Mount HDFS as a filesystem on the client
* Browse files using regular filesystem commands
* Upload/download files from HDFS
* Stream data to HDFS

Several other options

* Apache Flume – collecting, aggregating streaming data and moving into HDFS
* Apache Sqoop – Bulk transfer between Hadoop and datastop

Applications using HDFS

* Can use APIs to interact with HDFS
* Core component of Hadoop stack – used by all applications
* HBase is a good example of an application that runs on top of HDFS with good integration
* Spark can run directly on HDFS without other Hadoop components

Make a directory: sudo -u hdfs Hadoop fs -mkdir /user/test

Sudo -u hdfs Hadoop fs -put /user/test

Native Java API for HDFS

* List main classes needed for HDFS access
* Additional classes and methods: IO, Configuration and path information

Overview

* Base class: org.apache.hadoop.fs.FileSystem
* Important classes: FSDataInputStream, FSDataOutputStream
* Methods: get, open, create

FSDataInputStream Methods

* Read: read bytes
* Readfully: read from stream to buffer
* Seek: seek tp given offset
* getPos: get current position in stream

FSDataOutputStream Methods

* getPos: get current position in stream
* hflush: flush out the data in client’s user buffer
* close: close the underlying output stream

Reading from HDFS using API

* get an instance of FileSystem

FileSystem fs = FileSystem.get(URI.create(uri),conf)

* open an input system

in = fs.open(new Path(uri))

* use IO utilities to copy from input stream

IOUtils.copyBytes(in, System.out, 4096, fals)

* close the stream

IOUtils.closeStream(in)

Writing to HDFS using API

* get an instance of FileStream

FileSystem fs = FileSystem.get(URI.create(outuri),conf)

* create a file

out = fs.create(new Path(outuri))

* write to output stream

out.write(buffer, 0, nbytes)

* close the file

out.close()

Enabling WebHDFS

* In hdfs-site.xml

Dfs.webhdfs.enabled

Dfs.web.authentication.kerberos.principal

Dfs.web.authentication.kerberos.keytab

HTTP Operations

* HTTP GET: file status, checksums, attributes
* HTTP PUT: create, change ownership, rename, permissions, snapshot
* HTTP POST: append, concat
* HTTP DELETE: Delete files, snapshot