Hadoop Administration

Aakash Ahuja
aakash.x.ahuja@gmail.com
+91 7709 009634

Before we begin

Introductions

Send your email addresses so files/data can be shared

Before we begin

Generally speaking, and in principle, what is the role of an administrator?

- To ensure the system is available
- To ensure it is behaving as expected
- To ensure the system is running at required performance levels

Before we begin

What does it take to become a good administrator?

- Good understanding of the underlying system
- Command line versatility
- Basic programming skills in any language
- Problem solving skills
- Risk management
- Proactiveness
- Self learning attitude

Before we begin

Anything else?

- Keen observation
- Ability to correlate
- Ability to make mistakes

Administrator’s toolkit essentials

- Checklist of tasks
- Repeatable verification methods
- Issue log - constantly updating and shared centrally with team

Introduction to Big Data

What is Big Data
Big Deal about Big Data
Big Data Sources
Industries using Big Data
Big Data challenges

Big Data

"Big data" is a field that treats ways to analyze, systematically extract information from, or otherwise deal with data sets that are too large or complex to be dealt with by traditional data-processing application software.

Big data usually includes data sets with sizes beyond the ability of commonly used software tools to capture, curate, manage, and process data within a tolerable elapsed time. Big data philosophy encompasses unstructured, semi-structured and structured data, however the main focus is on unstructured data. Big data "size" is a constantly moving target, as of 2012 ranging from a few dozen terabytes to many exabytes of data. Big data requires a set of techniques and technologies with new forms of integration treveal insights from datasets that are diverse, complex, and of a massive scale.

Why is it important

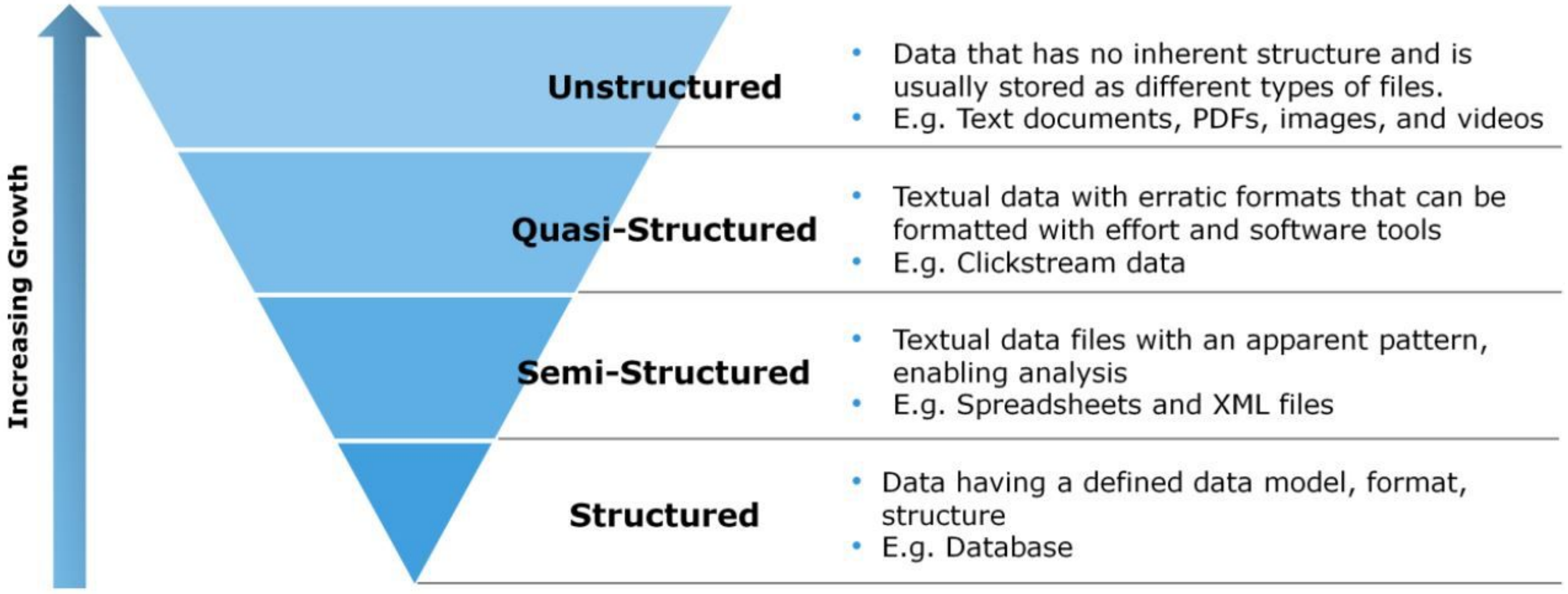
Changing technology paradigm - cheaper storage, commoditized cloud based technologies, improvement in data processing techniques, analytics

Increasing penetration of technology - Internet of things, smart devices, aerial (remote sensing), software logs, cameras, microphones, radio-frequency identification (RFID) readers and wireless sensor networks.

Ever increasing data - in part because they are increasingly gathered by cheap and numerous information- sensing. The world's technological per-capita capacity tstore information has roughly doubled every 40 months since the 1980s; as of 2012, every day 2.5 exabytes ($2.5×10^{18}$) of data are generated. Based on an IDC report prediction, the global data volume will grow exponentially from 4.4 zettabytes t44 zettabytes between 2013 and 2020.By 2025, IDC predicts there will be 163 zettabytes of data.One question for large enterprises is determining who should own big-data initiatives that affect the entire organization.

The rise of unstructured data - with internet becoming a primary medium for mankind’s needs to express itself, the value proposition has shifted from structured data to its unstructured cousin, and is likely going to remain so for some time to come

Big Data Sources



Unstructured

- Data that has no inherent structure and is usually stored as different types of files.
- E.g. Text documents, PDFs, images, and videos

Quasi-Structured

- Textual data with erratic formats that can be formatted with effort and software tools
- E.g. Clickstream data

Semi-Structured

- Textual data files with an apparent pattern, enabling analysis
- E.g. Spreadsheets and XML files

Structured

- Data having a defined data model, format, structure
- E.g. Database

Image courtesy: EMC

Industries using Big Data

Almost all industries are using Big Data now one way or another. Here are a few interesting ones:

Research	The Large Hadron Collider generates about 25 petabytes annually
Genomics	Genomic data available in public and private big data clusters have reduced the mapping time from 10 years previously t1 day now.
Aerospace	The Johns Hopkins Turbulence Databases (JHTDB) contains over 350 terabytes of spatiotemporal fields from Direct Numerical simulations of various turbulent flows
Sports	Formula One race cars are not fitted with hundreds of sensors generating terabytes of data
Retail	Massive product catalogues containing a variety of data forms are being used in combination with web logs, click streams and other user behaviour data to generate real time actions
Healthcare	Massive image datasets are being tested taid intelligent decision making in medicines
Pharma	Drug discovery timelines have greatly shortened now after the advent of big data and other associated techniques
Transit	Used to optimize user offerings by analyzing tons of transit data

Characteristics

Volume

The quantity of generated and stored data. The size of the data determines the value and potential insight, and whether it can be considered big data or not.

Variety

The type and nature of the data. This helps people who analyze it to effectively use the resulting insight. Big data draws from text, images, audio, video; plus it completes missing pieces through data fusion.

Velocity

In this context, the speed at which the data is generated and processed to meet the demands and challenges that lie in the path of growth and development. Big data is often available in real-time. Compared to small data, big data are produced more continually. Two kinds of velocity related to big data are the frequency of generation and the frequency of handling, recording, and publishing.

Veracity

It is the extended definition for big data, which refers to the data quality and the data value.The data quality of captured data can vary greatly, affecting the accurate analysis.

Challenges

Scalability, timeline and cost are the primary challenges considered before making a move to big data paradigm.

Not all data collected is useful. LHC can only used 0.001% of the data it generates - rest all is noise. It is important to be able to isolate signal from the ever increasing noise. Data with many cases (rows) offer greater statistical power, while data with higher complexity (more attributes or columns) may lead to a higher false discovery rate. Big data challenges include capturing data, data storage, data analysis, search, sharing, transfer, visualization, querying, updating, information privacy and data source.

Lack of technical talent is also emerging as a challenge in recent times.

Data security is the cornerstone of all IT infrastructure issues. The problem becomes even more prominent with the advent of HDFS like architectures.

Last but not the least, there still lies a lot of organization and human resistance when it comes to making a transition to big data.

Big Data Technologies

Solution to Big Data problems

Various Big Data Technologies

Big Data/Hadoop Platforms

Hadoop Distributions and Vendors

Big Data Suites

Popular big data platforms/suites

Cloudera

One of the first commercial Hadoop offerings and still the most popular, reportedly with more installations running than any of its competitors. Cloudera also contribute Impala, which offers real-time massively parallel processing of Big Data to Hadoop.

Amazon Web Services

Open source Big Data frameworks may not be the first thing that springs to mind when you think of Amazon, but the retailer was another one of the first to offer Hadoop in the cloud as part of its Amazon Web Services package. AWS is a hosted solution integrating Hadoop with Amazon's Elastic Cloud Compute and Simple Storage Service (S3) cloud-based data processing and storage services.

Hortonworks

Of the vendors listed here, Horton is one of the few which offer 100% open source Hadoop technology without any proprietary (non-open) modifications. They were also the first to integrate support for Apache HCatalog, which creates "metadata" – data within data – simplifying the process of sharing your data across other layers of service such as Apache Hive or Pig. This is further used internally by Microsoft and Teradata to offer their bundled Hadoop services.

MapR

Uses some differing concepts, such as native support for UNIX file systems rather than HDFS, meaning it will be more familiar to DBAs used to working in a UNIX environment. MapR technologies is also spearheading development of the Apache Drill project, which provides advanced tools for interactive real-time querying of Big Datasets.

continued...

IBM

It might be a relative newcomer to the Hadoop ecosystem, but IBM has deep roots in the computing industry, particularly in distributed computing and data management. Its BigInsights package adds its proprietary analytics and visualization algorithms to the core Hadoop infrastructure.

Intel Distribution for Apache Hadoop

Another giant of the tech world which has recently turned its attention towards Hadoop. Intel's distribution adds the company's Graph Builder and Analytics Toolkit functions to Hadoop, and claims that security updates to the infrastructure mean that their solution offers added security for your data.

Pivotal HD

Pivotal was formed as a joint venture between storage system provider EMC and virtualization specialists VMware. Pivotal HD (Hadoop Distribution) forms part of the company's Big Data Suite, which also includes database tools Greenplum and analytics platform Gemfire. Customers include China's national rail operator, China Railway – sorting out the logistics for rail journeys for 3.5 billion passengers certainly qualifies as Big Data!

Solution to big data problems

Scalability. timelines and cost are the primary challenges considered before making a move to big data paradigm. Various cloud service providers provide capacity on demand, advance monitoring and management tools to mitigate high upfront costs. Parallel processing frameworks and distributed, fault tolerant solutions like Hadoop fill in the gaps

Not all data collected is useful. LHC can only used 0.001% of the data it generates - rest all is noise. It is important to be able to isolate signal from the ever increasing noise. Data with many cases (rows) offer greater statistical power, while data with higher complexity (more attributes or columns) may lead to a higher false discovery rate. Big data challenges include capturing data, data storage, data analysis, search, sharing, transfer, visualization, querying, updating, information privacy and data source. Reusable data profiling systems must be in place for any new source of data, new business rules etc. The increasing availability of commodity solutions shall be used wherever possible

Lack of technical talent is also emerging as a challenge in recent times. While automated management available in cloud based environments helps to some extent, organizations are increasingly investing in ramping up technology skills of workforce

Data security is the cornerstone of all IT infrastructure issues. The problem becomes even more prominent with the advent of HDFS like architectures. This has been and will always remain a moving target for organizations. While native security features need to be used by Hadoop administrators, the larger onus falls on organization’s security policy makers and enforcers who need to stay ahead of the curve, ensure a reliable and feasible business continuity plan is in place, and have measures to detect and respond to security issues and threats quickly

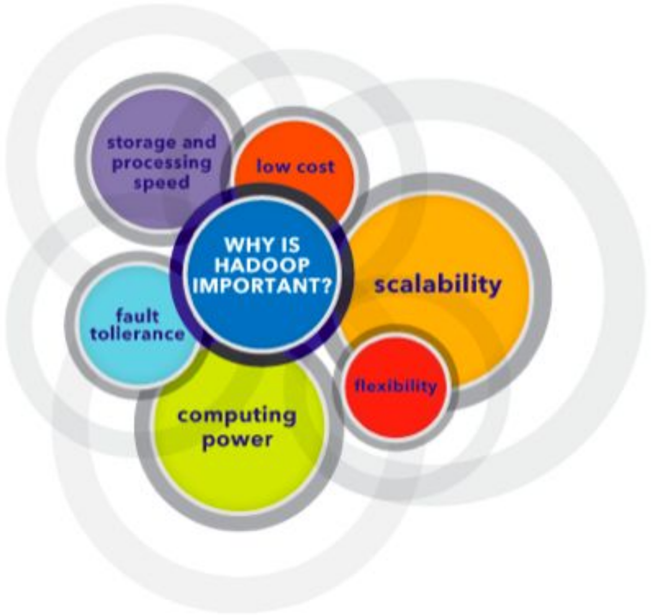
continued...

Last but not the least, there still lies a lot of organizational and human resistance when it comes to making a transition to big data. Organizational resistance always stems from not seeing the bottom line (revenue) benefits in the log terms. A detailed pros and cons report always helps. Once the decision makers are on board, it is relatively easier to rally the remaining workforce behind.

This is where Hadoop ecosystem presents itself.

So what is Hadoop and how it helps?

An **open-source software framework** for **storing massive amounts of data** and **running parallel applications** on clusters of **commodity hardware**.



Ability to store and process huge amounts of any kind of data, quickly. With data volumes and varieties constantly increasing, especially from social media and the Internet of Things (IoT), that's a key consideration.

Computing power. Hadoop's distributed computing model processes big data fast. The more computing nodes you use, the more processing power you have.

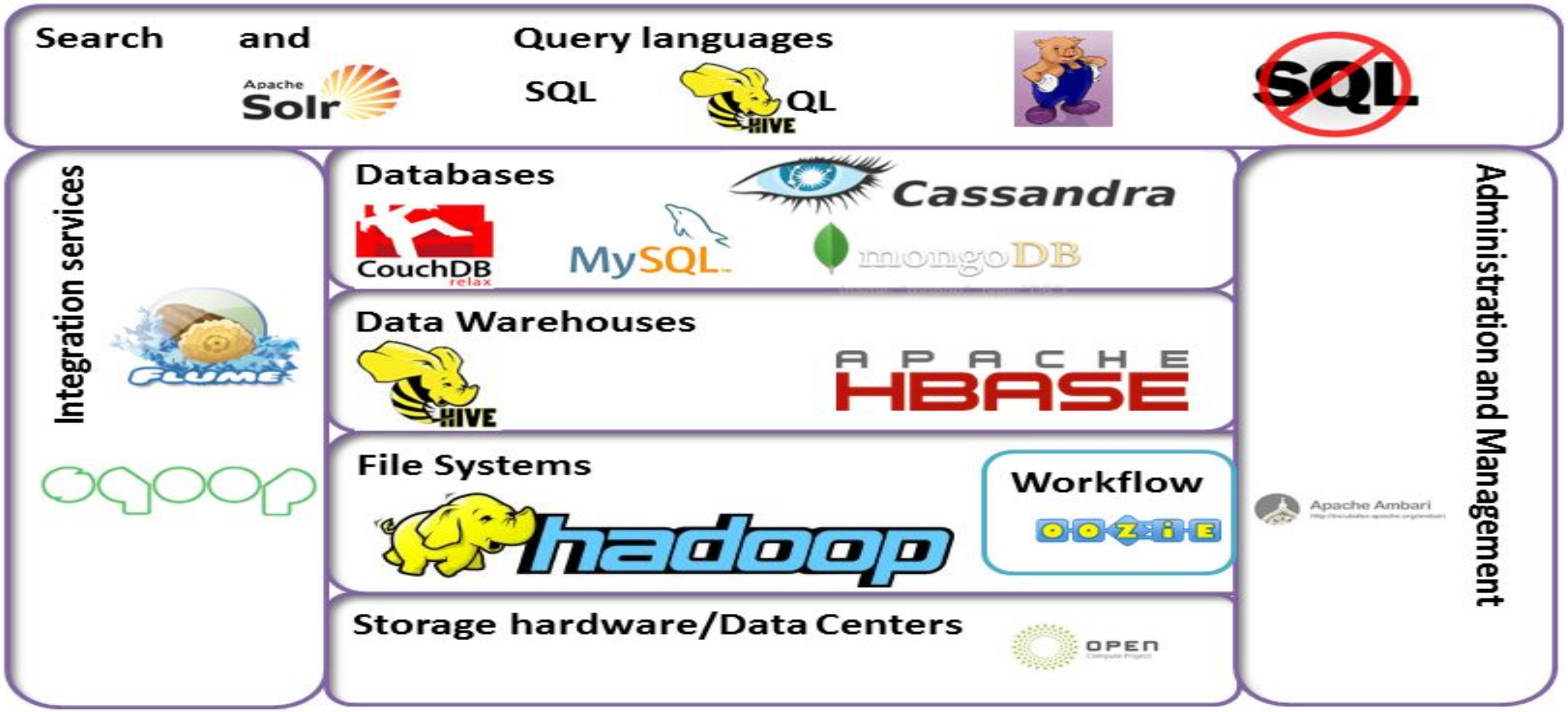
Fault tolerance. Data and application processing are protected against hardware failure. If a node goes down, jobs are automatically redirected to other nodes to make sure the distributed computing does not fail. Multiple copies of all data are stored automatically.

Flexibility. Unlike traditional relational databases, you don't have to preprocess data before storing it. You can store as much data as you want and decide how to use it later. That includes unstructured data like text, images and videos.

Low cost. The open-source framework is free and uses commodity hardware to store large quantities of data.

Scalability. You can easily grow your system to handle more data simply by adding nodes. Little administration is required.

Various big data technologies and Hadoop ecosystem



Introduction to Hadoop

A Brief History of Hadoop

Evolution of Hadoop

Comparison with Other Systems

Hadoop Releases

Hadoop - A brief history

An excerpt from Hadoop - A Definitive Guide by Tom White

Hadoop was created by Doug Cutting, the creator of Apache Lucene, the widely used text search library. Hadoop has its origins in Apache Nutch, an open source web search engine, itself a part of the Lucene project.

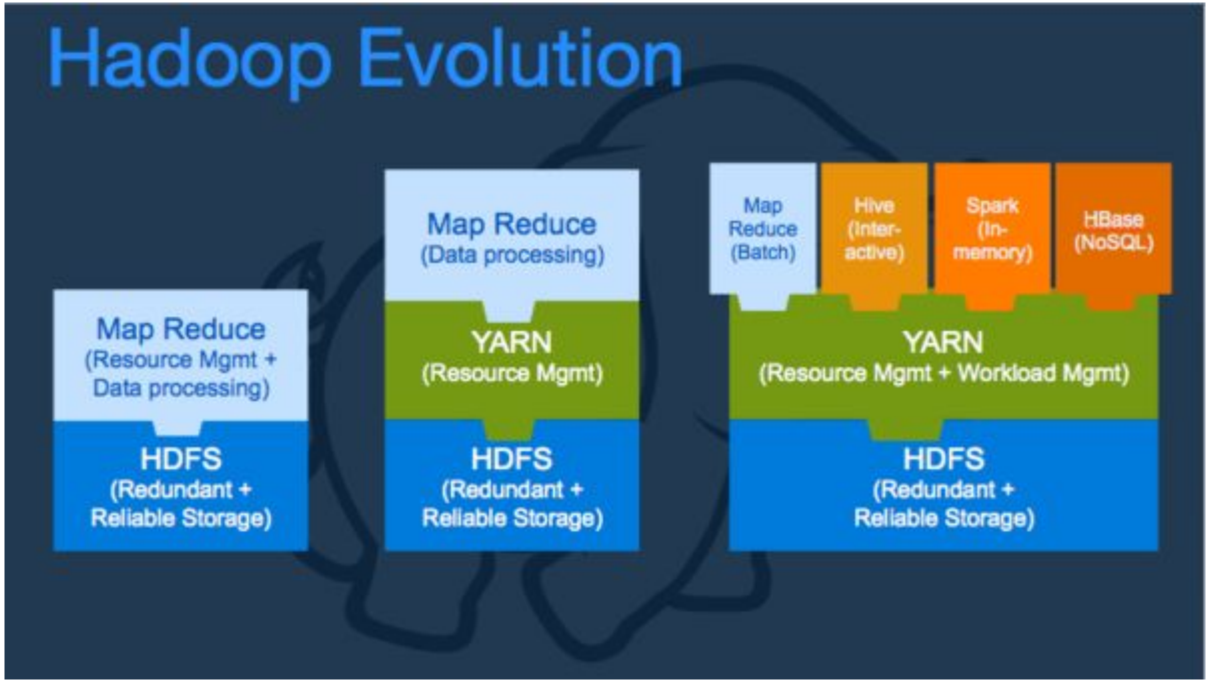
Building a web search engine from scratch was an ambitious goal, for not only is the software required to crawl and index websites complex to write, but it is also a challenge to run without a dedicated operations team, since there are so many moving parts. It's expensive, too: Mike Cafarella and Doug Cutting estimated a system supporting a 1-billion-page index would cost around half a million dollars in hardware, with a monthly running cost of $30,000.[10] Nevertheless, they believed it was a worthy goal, as it would open up and ultimately democratize search engine algorithms.

Nutch was started in 2002, and a working crawler and search system quickly emerged. However, they realized that their architecture wouldn't scale to the billions of pages on the Web. Help was at hand with the publication of a paper in 2003 that described the architecture of Google's distributed filesystem, called GFS, which was being used in production at Google.[11] GFS, or something like it, would solve their storage needs for the very large files generated as a part of the web crawl and indexing process. In particular, GFS would free up time being spent on administrative tasks such as managing storage nodes. In 2004, they set about writing an open source implementation, the Nutch Distributed Filesystem (NDFS).

In 2004, Google published the paper that introduced MapReduce to the world.[12] Early in 2005, the Nutch developers had a working MapReduce implementation in Nutch, and by the middle of that year all the major Nutch algorithms had been ported to run using MapReduce and NDFS.

NDFS and the MapReduce implementation in Nutch were applicable beyond the realm of search, and in February 2006 they moved out of Nutch to form an independent subproject of Lucene called Hadoop. Hadoop's first recorded massive scale production was by Yahoo! in 2007 on a 1,000 node cluster.

Evolution of Hadoop



Hadoop 1
Advent of distributed computing powered by a highly distributed, fault tolerant storage and data local, parallel processing

Hadoop 2
Spinning off of MR responsibilities around task management into YARN and thereby allowing better fault tolerance. Ever increasing ecosystem now allows more specialized applications and in-memory processing

Hadoop 3
As the time of writing, Hadoop 3 has been released that has enhancements around fault tolerance, optimization etc

Comparison with other systems

RDBMS

RDBMS is good for point queries or updates, where the dataset has been in- dexed to deliver low-latency retrieval and update times of a relatively small amount of data. MapReduce is a good fit for problems that need to analyze the whole dataset, in a batch fashion, particularly for ad hoc analysis.

RDBMS is good for datasets that are continually updated. MapReduce suits applications where the data is written once, and read many times.

RDBMS operates on structured data, and there is evolving support for semi-structured data. MapReduce works well on unstructured or semi-structured data.

Relational data is often normalized. MapReduce doesn't necessarily need normalized data.

RDBMS is not scalable linearly. MapReduce is a linearly scalable programming model.

HPC/Grid Computing

Uses APIs and Message Passing Interfaces (MPI) which move the data round for processing, which becomes a bottleneck with very large data. MapReduce works on data locality principle and mostly works without moving the data around the network.

MPIs allow higher flexibility to programmers but they must handle data movement themselves. MapReduce does data movement implicitly.

Fault tolerance and restartability has to handled explicitly. MapReduce handles fault tolerance and restartability implicitly and is easily configurable in this regard.

Volunteer Computing

Works with very small units of work, around 0.35 MB each, which is pushed to volunteer devices for computation. Massive datasets can be processed without necessarily having to break them into such small scale units, ie, the data locality approach.

Largely works by CPU donating, bandwidth intensive jobs cannot be handled easily. While bandwidth is a sacred resource even in the Mapreduce world, behavior can be largely customized and tuned to get optimum results in some cases..

Not always possible to run very long running jobs. MapReduce can handle jobs that runs for hours, days, weeks, and in some cases, months.

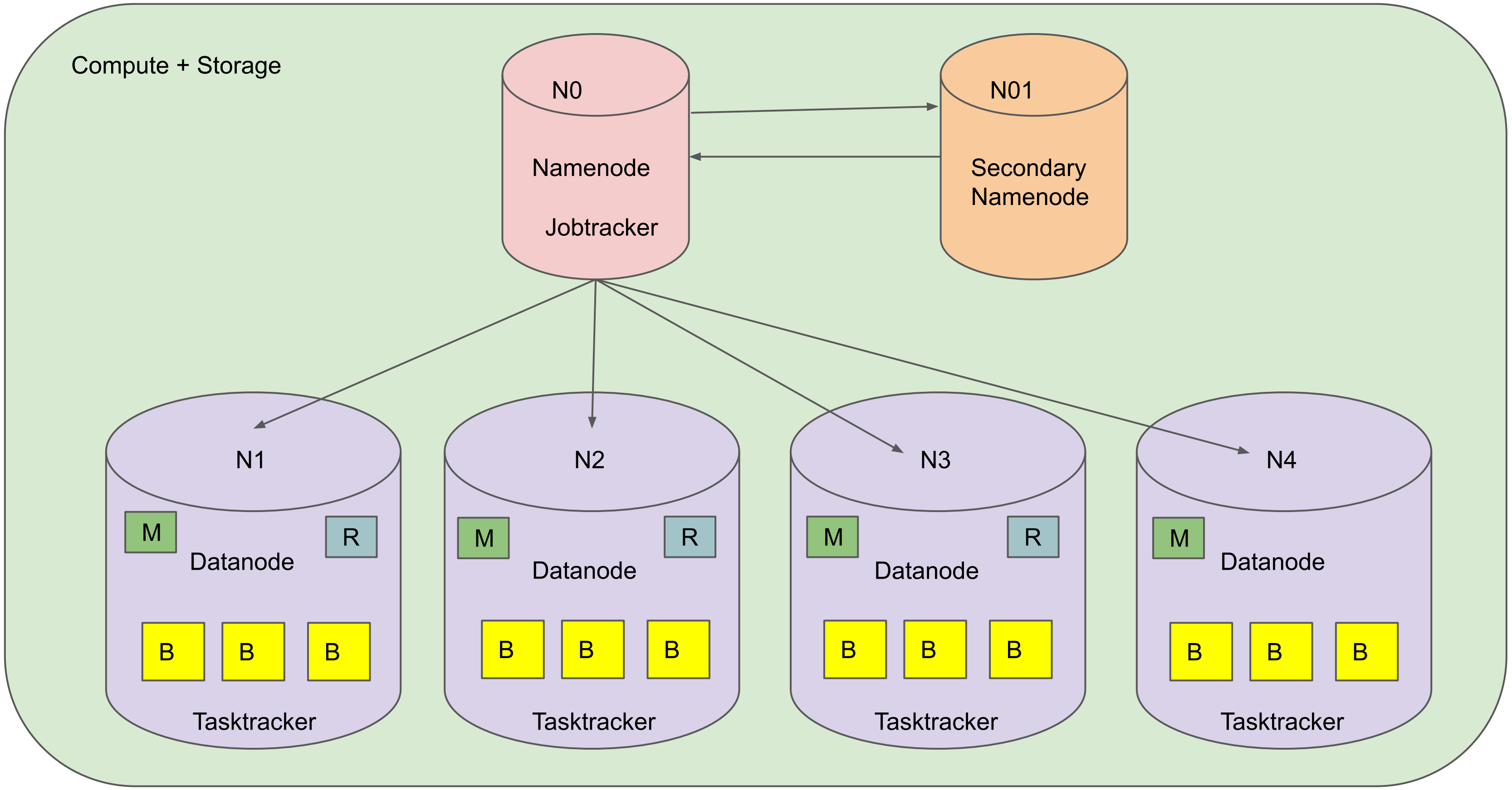
Hadoop releases

At the time of writing:

Apache Hadoop is into 3.x.x

Cloudera Distribution of Hadoop is into 5.x

Hadoop architecture



Hadoop architecture components

Namenode

- Maintains the filesystem tree & metadata. This information is stored on the local disk in the form of two files: the namespace image and the edit log. The namenode also knows the datanodes on which all the blocks for a given file are located, however, it does not store block locations persistently, since this information is reconstructed from datanodes when the system starts
- Clients connect to the namenode to perform filesystem operations; although, block data is streamed to and from datanodes directly, so bandwidth is not limited by a single node
- Datanodes regularly report their status to the namenode in a heartbeat. This means that, at any given time, the namenode has a complete view of all datanodes in the cluster, their current health, and what blocks they have available
- When a datanode initially starts up, as well as every hour thereafter, it sends what's called a *block report* to the namenode. The block report is simply a list of all blocks the datanode currently has on its disks and allows the namenode to keep track of any changes. This is also necessary because, while the file to block mapping on the namenode is stored on disk, the locations of the blocks are not written to disk. This may seem counterintuitive at first, but it means a change in IP address or hostname of any of the datanodes does not impact the underlying storage of the filesystem metadata. Another nice side effect of this is that, should a datanode experience failure of a motherboard, administrators can simply remove its hard drives, place them into a new chassis, and start up the new machine. As far as the namenode is concerned, the blocks have simply moved to a new datanode. The downside is that, when initially starting a cluster (or restarting it, for that matter), the namenode must wait to receive block reports from all datanodes to know all blocks are present
- The namenode filesystem metadata is served entirely from RAM for fast lookup and retrieval, and thus places a cap on how much metadata the namenode can handle. A rough estimate is that the metadata for 1 million blocks

Hadoop architecture components

Datanode

- The workhorses of the filesystem
- They store and retrieve blocks when they are told to (by clients or the namenode)
- They report back to the namenode periodically with lists of blocks that they are storing

Hadoop architecture components

Secondary namenode

- Without the namenode, the filesystem cannot be used. In fact, if the machine running the namenode were obliterated, all the files on the filesystem would be lost since there would be no way of knowing how to reconstruct the files from the blocks on the datanodes. For this reason, it is important to make the namenode resilient to failure. While there are methods to achieve this, this leads us to the concept of a secondary namenode
- This does not act as a namenode
- Its main role is to periodically merge the namespace image with the edit log to prevent the edit log from becoming too large
- The secondary namenode usually runs on a separate physical machine, since it requires plenty of CPU and as much memory as the namenode to perform the merge
- It keeps a copy of the merged namespace image, which can be used in the event of the namenode failing. However, the state of the secondary namenode lags that of the primary, so in the event of total failure of the primary, data loss is almost certain. The usual course of action in this case is to copy the namenode's metadata files that are on NFS to the secondary and run it as the new primary.

Hadoop architecture components

Jobtracker

- The master process
- Responsible for accepting job submissions from clients
- Responsible for scheduling tasks to run on worker nodes. This is not scheduling in the way that the cron daemon executes jobs at given times, but instead is more like the way the OS kernel schedules process CPU time
- Responsible for providing administrative functions such as worker health and task progress monitoring to the cluster
- There is one jobtracker per MapReduce cluster and it usually runs on reliable hardware since a failure of the master will result in the failure of all running jobs. Clients and tasktrackers communicate with the jobtracker by way of remote procedure calls (RPC).
- Tasktrackers inform the jobtracker as to their current health and status by way of regular heartbeats. Each heartbeat contains the total number of map and reduce *task slots* available, the number occupied, and detailed information about any currently executing tasks. After a configurable period of no heartbeats, a tasktracker is assumed dead. The jobtracker uses a thread pool to process heartbeats and client requests in parallel.
- When a job is submitted, information about each task that makes up the job is stored in memory. This task information updates with each tasktracker heartbeat while the tasks are running, providing a near real-time view of task progress and health. After the job completes, this information is retained for a configurable window of time or until a specified number of jobs have been executed

Hadoop architecture components

Tasktracker

- Accepts task assignments from the jobtracker, instantiates the user code, executes those tasks locally, and reports progress back to the jobtracker periodically . There is always a single tasktracker on each worker node
- Both tasktrackers and datanodes run on the same machines, which makes each node both a compute node and a storage node, respectively.
- Configured with a specific number of map and reduce task slots that can be run in parallel. A task slot is an allocation of available resources on a worker node to which a task may be assigned, in which case it is executed
- Upon receiving a task assignment from the jobtracker, the tasktracker executes an attempt of the task in a separate process. The distinction between a task and a task attempt is important: a task is the logical unit of work, while a task attempt is a specific, physical instance of that task being executed. Since an attempt may fail, it is possible that a task has multiple attempts, although it's common for tasks to succeed on their first attempt when everything is in proper working order. As this implies, each task in a job will always have at least one attempt, assuming the job wasn't administratively killed. Communication between the task attempt (usually called the child, or child process) and the tasktracker is maintained via an RPC connection over the loopback interface called the umbilical protocol. The task attempt itself is a small application that acts as the container in which the user's map or reduce code executes. As soon as the task completes, the child exits and the slot becomes available for assignment.
- The tasktracker uses a list of user-specified directories (each of which is assumed to be on a separate physical device) to hold the intermediate map output and reducer input during job execution. This is required because this data is usually too large to fit exclusively in memory for large jobs or when many jobs are running in parallel
- Tasktrackers, like the jobtracker, also have an embedded web server and user interface

Hadoop Distributed Filesystem

- Filesystems that manage the storage across a network of machines are called distributed filesystems
- Since they are network-based, all the complications of network programming kick in, thus making distributed filesystems more complex than regular disk filesystems. For example, one of the biggest challenges is making the filesystem tolerate node failure without suffering data loss
- Hadoop comes with a distributed filesystem called HDFS, which stands for Hadoop Distributed Filesystem
- HDFS is a filesystem designed for storing very large files with streaming data access patterns, running on clusters of commodity hardware

HDFS features

- Can process hundreds of terabytes of data. There are petabyte scale systems being used as of today
- HDFS is built around the idea that the most efficient data processing pattern is a write-once, read-many-times pattern. A dataset is typically generated or copied from source, then various analyses are performed on that dataset over time. Each analysis will involve a large proportion, if not all, of the dataset, so the time to read the whole dataset is more important than the latency in reading the first record.
- Hadoop doesn’t require expensive, highly reliable hardware to run on. It’s designed to run on clusters of commodity hardware (commonly available hardware available from multiple vendors for which the chance of node failure across the cluster is high, at least for large clusters. HDFS is designed to carry on working without a noticeable interruption to the user in the face of such failure

HDFS limitations

- Applications that require low-latency access to data, in the tens of milliseconds range, will not work well with HDFS. Remember, HDFS is optimized for delivering a high throughput of data, and this may be at the expense of latency. HBase is currently a better choice for low-latency access
- It is not suited for very small sized large number of files. Since the namenode holds filesystem metadata in memory, the limit to the number of files in a filesystem is governed by the amount of memory on the namenode. As a rule of thumb, each file, directory, and block takes about 150 bytes. So, for example, if you had one million files, each taking one block, you would need at least 300 MB of memory. While storing millions of files is feasible, billions is beyond the capability of current hardware
- Files in HDFS may be written to by a single writer. Writes are always made at the end of the file. There is no support for multiple writers, or for modifications at arbitrary offsets in the file. (These might be supported in the future, but they are likely to be relatively inefficient.)

HDFS components

- Namenode and datanode are covered in previous sections. Here we will focus on disk blocks
- But first, what is a block?
- Why use blocks at all?
- a file can be larger than any single disk in the network.
- There's nothing that requires the blocks from a file to be stored on the same disk, so they can take advantage of any of the disks
- It simplifies the storage subsystem. The storage subsystem deals with blocks, simplifying storage management (since blocks are a fixed size, it is easy to calculate how many can be stored on a given disk) and eliminating metadata concerns (blocks are just a chunk of data to be stored—file metadata such as permissions information does not need to be stored with the blocks, so another system can handle metadata separately)
- Furthermore, blocks fit well with replication for providing fault tolerance and availability
- HDFS blocks are a relatively larger size then normal disk blocks. 64 MB by default usually, some versions may have a default of 128 MB as well
- By making a block large enough, the time to transfer the data from the disk can be made to be significantly larger than the time to seek to the start of the block. Thus the time to transfer a large file made of multiple blocks operates at the disk transfer rate.
- Files in HDFS are broken into block-sized chunks, which are stored as independent units
- Unlike a filesystem for a single disk, a file in HDFS that is smaller than a single block does not occupy a full block's worth of underlying storage
- Map tasks work at a block level

Installing Hadoop

Single node cluster

Multi node cluster

steps

Requirements:

VM running Ubuntu v14

Minimum about 1 vcore, 10 GB persistence storage, 4 GB RAM

root access

Refer to Hadoop Installation excel, tab single node

Repeat for multi node install, use tab multi node

HDFS architecture

The Hadoop Distributed File System (HDFS):

- is a distributed file system
- designed to run on commodity hardware
- highly fault-tolerant
- provides high throughput access to application data and is suitable for applications that have large data sets

HDFS - Assumptions and goals

Hardware Failure

Hardware failure is the norm rather than the exception. An HDFS instance may consist of hundreds or thousands of server machines, each storing part of the file system's data. The fact that there are a huge number of components, any of which could fail at anytime. Therefore, detection of faults and quick, automatic recovery from them is a core architectural goal of HDFS.

Streaming Data Access

Applications that run on HDFS need streaming access to their data sets. They are not general purpose applications that typically run on general purpose file systems. HDFS is designed more for batch processing rather than interactive use by users. The emphasis is on high throughput of data access rather than low latency of data access.

Large Data Sets

Applications that run on HDFS have large data sets. A typical file in HDFS is gigabytes to terabytes in size. Thus, HDFS is tuned to support large files. It should provide high aggregate data bandwidth and scale to hundreds of nodes in a single cluster.

Simple Coherency Model

HDFS applications need a write-once-read-many access model for files. A file once created, written, and closed need not be changed except for appends and truncates. Appending the content to the end of the files is supported but cannot be updated at arbitrary point. This assumption simplifies data coherency issues and enables high throughput data access. A MapReduce application or a web crawler application fits perfectly with this model.

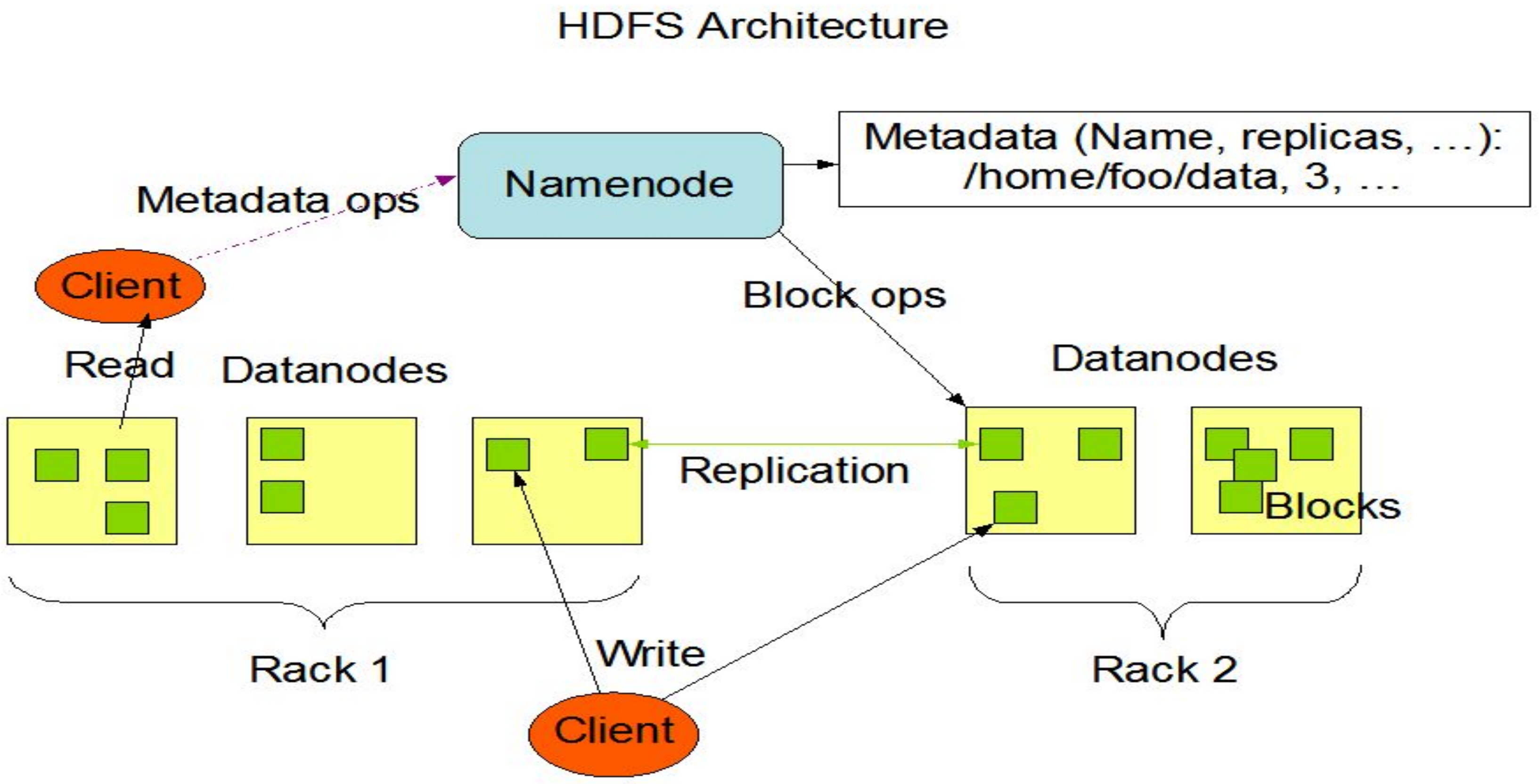
"Moving Computation is Cheaper than Moving Data"

A computation requested by an application is much more efficient if it is executed near the data it operates on. This is especially true when the size of the data set is huge. This minimizes network congestion and increases the overall throughput of the system. The assumption is that it is often better to migrate the computation closer to where the data is located rather than vice versa.

Portability Across Heterogeneous Hardware and Software Platforms

HDFS has been designed to be easily portable from one platform to another.

HDFS architecture (also shows HDFS data storage process)



NameNode and DataNodes

- HDFS is built using the Java language; any machine that supports Java can run the NameNode or the DataNode software
- HDFS has a master/slave architecture
- An HDFS cluster consists of a single NameNode, a master server that manages the file system namespace and regulates access to files by clients
- There are a number of DataNodes, usually one per node in the cluster, which manage storage attached to the nodes that they run on
- HDFS allows user data to be stored in files. Internally, a file is split into one or more blocks and these blocks are stored in a set of DataNodes.
- The NameNode executes file system namespace operations like opening, closing, and renaming files and directories. The data never flows through Namenode
- It also determines the mapping of blocks to DataNodes
- The DataNodes are responsible for serving read and write requests from the file system’s clients
- The DataNodes also perform block creation, deletion, and replication upon instruction from the NameNode

The File System Namespace

- HDFS supports a traditional hierarchical file organization
- A user or an application can create directories and store files inside these directories
- User can create and remove files, move a file from one directory to another, or rename a file
- HDFS supports user quotas and access permissions
- HDFS does not support hard links or soft links

Data Replication

- An application can specify the number of replicas of a file that should be maintained by HDFS. The number of copies of a file is called the replication factor of that file. This information is stored by the NameNode
- HDFS stores each file as a sequence of blocks. The blocks of a file are replicated for fault tolerance. The block size and replication factor are configurable per file
- An application can specify the number of replicas of a file. The replication factor can be specified at file creation time and can be changed later
- The NameNode makes all decisions regarding replication of blocks. It periodically receives a Heartbeat and a Blockreport from each of the DataNodes in the cluster. Receipt of a Heartbeat implies that the DataNode is functioning properly. A Blockreport contains a list of all blocks on a DataNode

Replica Placement

- Large HDFS instances run on a cluster of computers that commonly spread across many racks. Communication between two nodes in different racks has to go through switches. In most cases, network bandwidth between machines in the same rack is greater than network bandwidth between machines in different racks
- The NameNode determines the rack id each DataNode belongs to via the process outlined in Hadoop Rack Awareness. A simple but non-optimal policy is to place replicas on unique racks. This prevents losing data when an entire rack fails and allows use of bandwidth from multiple racks when reading data. This policy evenly distributes replicas in the cluster which makes it easy to balance load on component failure. However, this policy increases the cost of writes because a write needs to transfer blocks to multiple racks

- The placement of replicas is critical to HDFS reliability and performance and uses rack-awareness to optimize it
- The purpose of a rack-aware replica placement policy is to improve data reliability, availability, and network bandwidth utilization

- For the common case, when the replication factor is three, HDFS's placement policy is to put one replica on the local machine if the writer is on a datanode, otherwise on a random datanode, another replica on a node in a different (remote) rack, and the last on a different node in the same remote rack
- This policy does not impact data reliability and availability guarantees. However, it does reduce the aggregate network bandwidth used
- With this policy, the replicas of a file do not evenly distribute across the racks. One third of replicas are on one node, two thirds of replicas are on one rack, and the other third are evenly distributed across the remaining racks. This policy improves write performance without compromising data reliability or read performance.
- If the replication factor is greater than 3, the placement of the 4th and following replicas are determined randomly while keeping the number of replicas per rack below the upper limit (which is basically (replicas - 1) / racks + 2).
- Because the NameNode does not allow DataNodes to have multiple replicas of the same block, maximum number of replicas created is the total number of DataNodes at that time

The Persistence of File System Metadata

- The HDFS namespace is stored by the NameNode
- The NameNode uses a transaction log called the EditLog to persistently record every change that occurs to file system metadata. The NameNode uses a file in its local host OS file system to store the EditLog. The entire file system namespace, including the mapping of blocks to files and file system properties, is stored in a file called the FsImage. The FsImage is stored as a file in the NameNode's local file system too.
- The NameNode keeps an image of the entire file system namespace and file Blockmap in memory
- When the NameNode starts up, or a checkpoint is triggered by a configurable threshold, it reads the FsImage and EditLog from disk, applies all the transactions from the EditLog to the in-memory representation of the FsImage, and flushes out this new version into a new FsImage on disk. It can then truncate the old EditLog because its transactions have been applied to the persistent FsImage. This process is called a checkpoint
- Even though it is efficient to read a FsImage, it is not efficient to make incremental edits directly to a FsImage. Instead of modifying FsImage for each edit, we persist the edits in the Editlog. During the checkpoint the changes from Editlog are applied to the FsImage
- A checkpoint can be triggered at a given time interval (dfs.namenode.checkpoint.period) expressed in seconds, or after a given number of filesystem transactions have accumulated (dfs.namenode.checkpoint.txns). If both of these properties are set, the first threshold to be reached triggers a checkpoint.
- The DataNode stores HDFS data in files in its local file system. The DataNode has no knowledge about HDFS files. It stores each block of HDFS data in a separate file in its local file system
- The DataNode does not create all files in the same directory. Instead, it uses a heuristic to determine the optimal number of files per directory and creates subdirectories appropriately. It is not optimal to create all local files in the same directory because the local file system might not be able to efficiently support a huge number of files in a single directory
- When a DataNode starts up, it scans through its local file system, generates a list of all HDFS data blocks that correspond to each of these local files, and sends this report to the NameNode. The report is called the *Blockreport*

Robustness

The primary objective of HDFS is to store data reliably even in the presence of failures. The three common types of failures are NameNode failures, DataNode failures and network partitions.

Data Disk Failure, Heartbeats and Re-Replication

Each DataNode sends a Heartbeat message to the NameNode periodically. A network partition can cause a subset of DataNodes to lose connectivity with the NameNode. The NameNode detects this condition by the absence of a Heartbeat message. The NameNode marks DataNodes without recent Heartbeats as dead and does not forward any new IO requests to them. Any data that was registered to a dead DataNode is not available to HDFS any more. DataNode death may cause the replication factor of some blocks to fall below their specified value. The NameNode constantly tracks which blocks need to be replicated and initiates replication whenever necessary. The necessity for re-replication may arise due to many reasons: a DataNode may become unavailable, a replica may become corrupted, a hard disk on a DataNode may fail, or the replication factor of a file may be increased.

The time-out to mark DataNodes dead is conservatively long (over 10 minutes by default) in order to avoid replication storm caused by state flapping of DataNodes. Users can set shorter interval to mark DataNodes as stale and avoid stale nodes on reading and/or writing by configuration for performance sensitive workloads.

Data Integrity

It is possible that a block of data fetched from a DataNode arrives corrupted. This corruption can occur because of faults in a storage device, network faults, or buggy software. The HDFS client software implements checksum checking on the contents of HDFS files. When a client creates an HDFS file, it computes a checksum of each block of the file and stores these checksums in a separate hidden file in the same HDFS namespace. When a client retrieves file contents it verifies that the data it received from each DataNode matches the checksum stored in the associated checksum file. If not, then the client can opt to retrieve that block from another DataNode that has a replica of that block.

Metadata Disk Failure

The FsImage and the EditLog are central data structures of HDFS. A corruption of these files can cause the HDFS instance to be non-functional. For this reason, the NameNode can be configured to support maintaining multiple copies of the FsImage and EditLog. Any update to either the FsImage or EditLog causes each of the FsImages and EditLogs to get updated synchronously. This synchronous updating of multiple copies of the FsImage and EditLog may degrade the rate of namespace transactions per second that a NameNode can support. However, this degradation is acceptable because even though HDFS applications are very data intensive in nature, they are not metadata intensive. When a NameNode restarts, it selects the latest consistent FsImage and EditLog to use.

Snapshots

Snapshots support storing a copy of data at a particular instant of time. One usage of the snapshot feature may be to roll back a corrupted HDFS instance to a previously known good point in time.

Data Organization

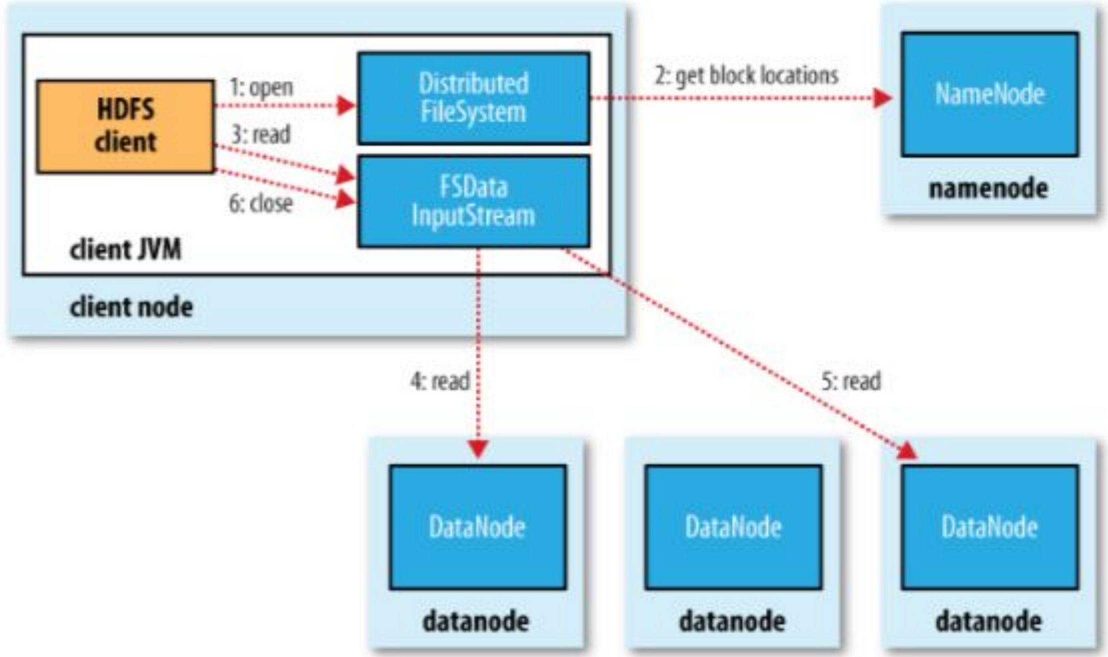
Data Blocks

HDFS is designed to support very large files. Applications that are compatible with HDFS are those that deal with large data sets. These applications write their data only once but they read it one or more times and require these reads to be satisfied at streaming speeds. HDFS supports write-once-read-many semantics on files. A typical block size used by HDFS is 128 MB. Thus, an HDFS file is chopped up into 128 MB chunks, and if possible, each chunk will reside on a different DataNode.

Replication Pipelining

When a client is writing data to an HDFS file with a replication factor of three, the NameNode retrieves a list of DataNodes using a replication target choosing algorithm. This list contains the DataNodes that will host a replica of that block. The client then writes to the first DataNode. The first DataNode starts receiving the data in portions, writes each portion to its local repository and transfers that portion to the second DataNode in the list. The second DataNode, in turn starts receiving each portion of the data block, writes that portion to its repository and then flushes that portion to the third DataNode. Finally, the third DataNode writes the data to its local repository. Thus, a DataNode can be receiving data from the previous one in the pipeline and at the same time forwarding data to the next one in the pipeline. Thus, the data is pipelined from one DataNode to the next.

Anatomy of a file read



The client opens the file it wishes to read by calling open() on the FileSystem object, which for HDFS is an instance of DFS (step 1)

DFS calls the namenode, using RPC, to determine the locations of the blocks for the first few blocks in the file (step 2).

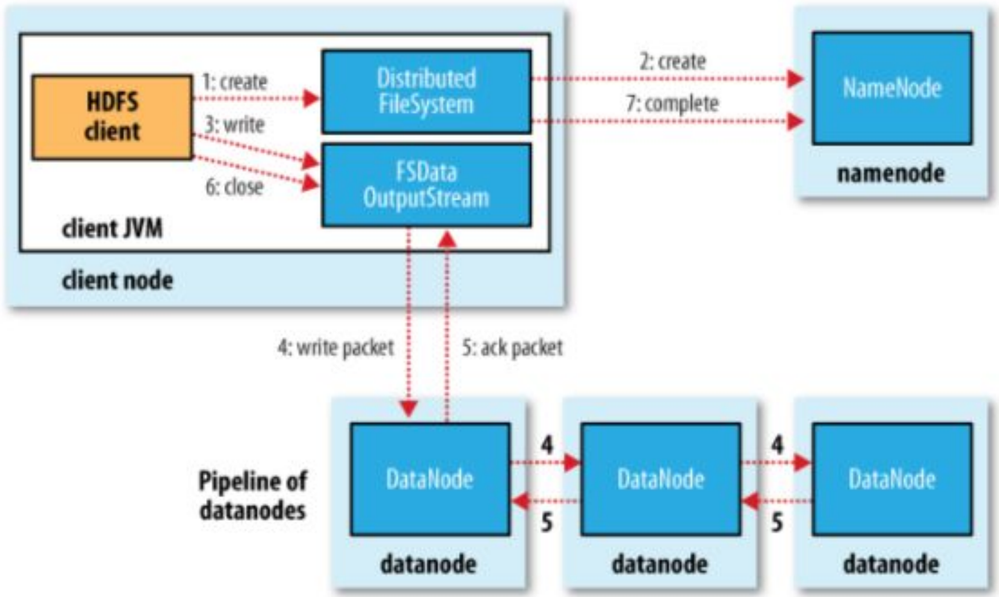
For each block, the namenode returns the addresses of the datanodes that have a copy of that block. Datanodes are sorted according to their proximity to the client. Datastreams are setup.

The client then calls read() on the stream (step 3). DFSInputStream, which has stored the datanode addresses for the first few blocks in the file, then connects to the first (closest) datanode for the first block in the file. Data is streamed from the datanode back to the client, which calls read() repeatedly on the stream (step 4).

When the end of the block is reached, DFSInputStream will close the connection to the datanode, then find the best datanode for the next block (step 5). This happens transparently to the client, which from its point of view is just reading a continuous stream. Blocks are read in order with the DFSInputStream opening new connections to datanodes as the client reads through the stream. It will also call the namenode to retrieve the datanode locations for the next batch of blocks as needed. When the client has finished reading, it calls close() on the FSDataInputStream (step 6).

During reading, if the DFSInputStream encounters an error while communicating with a datanode, then it will try the next closest one for that block. Checksums are done on blocks.

Anatomy of a file write



Client creates the file by calling create() on DFS (step 1). DFS makes an RPC call to the namenode to create a new file in the namespace, with no blocks associated with it (step 2). Namenode makes sure the file doesn't already exist, and that the client has the right permissions to create the file. If yes, the namenode makes a record of the new file and a DFSOutputStream is returned to client; else file creation fails.

DFSOutputStream maintains an internal queue of packets that are waiting to be acknowledged by datanodes, called the *ack queue*. A packet is removed from the ack queue only when it has been acknowledged by all the datanodes in the pipeline (step 5).If a datanode fails while data is being written to it, then the following actions are taken, which are transparent to the client writing the data. First the pipeline is closed, and any packets in the ack queue are added to the front of the data queue so that datanodes that are downstream from the failed node will not miss any packets. The current block on the good datanodes is given a new identity, which is communicated to the name-node, so that the partial block on the failed datanode will be deleted if the failed datanode recovers later on.

The failed datanode is removed from the pipeline and the remainder of the block's data is written to the two good datanodes in the pipeline. The namenode notices that the block is under-replicated, and it arranges for a further replica to be created on another node. Subsequent blocks are then treated as normal.

It's possible, but unlikely, that multiple datanodes fail while a block is being written. As long as dfs.replication.min replicas are written, the write will succeed, and the block will be asynchronously replicated across the cluster until its target replication factor is reached (dfs.replication, which defaults to three).

When the client has finished writing data, it closes the stream (step 6). All the remaining packets are flushed to the datanode pipeline and DFS waits for ac-knowledgments before contacting the namenode to signal that the file is complete (step 7). The namenode already knows which blocks the file is made up of so it only has to wait for blocks to be minimally replicated before returning successfully.

Network topology and Hadoop

What does it mean for two nodes in a local network to be “close” to each other? In the context of high-volume data processing, the limiting factor is the rate at which we can transfer data between nodes—bandwidth is a scarce commodity. The idea is to use the bandwidth between two nodes as a measure of distance.

Rather than measuring bandwidth between nodes, which can be difficult to do in practice (it requires a quiet cluster, and the number of pairs of nodes in a cluster grows as the square of the number of nodes), Hadoop takes a simple approach in which the network is represented as a tree and the distance between two nodes is the sum of their distances to their closest common ancestor. Levels in the tree are not predefined, but it is common to have levels that correspond to the data center, the rack, and the node that a process is running on. The idea is that the bandwidth available for each of the following scenarios becomes progressively less:

- Processes on the same node
- Different nodes on the same rack
- Nodes on different racks in the same data center
- Nodes in different data centers
  For example, imagine a node *n1* on rack *r1* in data center *d1*. This can be represented as */d1/r1/n1*. Using this notation, here are the distances for the four scenarios:
    - *distance(/d1/r1/n1, /d1/r1/n1)* = 0 (processes on the same node)
    - *distance(/d1/r1/n1, /d1/r1/n2)* = 2 (different nodes on the same rack)
    - *distance(/d1/r1/n1, /d1/r2/n3)* = 4 (nodes on different racks in the same data center)
    - *distance(/d1/r1/n1, /d2/r3/n4)* = 6 (nodes in different data centers)

Rack awareness

By communicating topology information to Hadoop, we influence the placement of data within the cluster as well as the processing of that data.

Both the Hadoop distributed filesystem and MapReduce are aware of, and benefit from rack topology information, when it’s available. We already understand that HDFS keeps multiple copies of each block and stores them on different machines. Without topology information, a cluster that spans racks could place all replicas on a single rack, leaving us susceptible to data availability problems in the case that an entire rack failed.

Rack topology is configured in Hadoop by implementing a script that, when given a list of hostnames or IP addresses on the command line, prints the rack in which the machine is located, in order. The implementation of the topology script is entirely up to the administrator and may be as simple as a shell script that has a hardcoded list of machines and rack names, or as sophisticated as a C executable that reads data from a relational database.

Rack awareness implementation

Under $HADOOP_CONF_DIR, create a script called rackaw.sh, add following. Set permissions to 755

```
#/bin/sh
### script to test rack awareness
### needs a file ${HADOOP_CONF_DIR}/topology.csv containing IPs and corresponding racks

### needs IP as input

nf="/usr/local/hadoop/etc/hadoop/topology.csv"
expr `grep $1 $nf | cut -f 2 -d ","`
```

Under $HADOOP_CONF_DIR, create a file called topology.csv, add following. Set permissions to 744

```
<IP for pri-node>,/rack1
<IP for d-node-a>,/rack2
<IP for d-node-b>,/rack1
```

Edit core-site.xml, add the following:

```
<property>
        <name>net.topology.script.file.name</name>
        <value>etc/hadoop/rackaw.sh</value>
</property>
```

If using a multi node cluster, replicate to other nodes.

Testing using: hadoop dfsadmin -report

Each node shall be showing with a dummy rack value.

Experiment to see how it impacts data and process placement. Try with a smaller files, and replication factor set to 2 in 3 node cluster. Does that work?

Map Reduce

Anatomy of a MapReduce Job Run

In releases of Hadoop up to and including the 0.20 release series, mapred.job.tracker determines the means of execution. If this configuration property is set to local, the default, then the local job runner is used. This runner runs the whole job in a single JVM. It's designed for testing and for running MapReduce programs on small datasets.

Alternatively, if mapred.job.tracker is set to a colon-separated host and port pair, then the property is interpreted as a jobtracker address, and the runner submits the job to the jobtracker at that address.

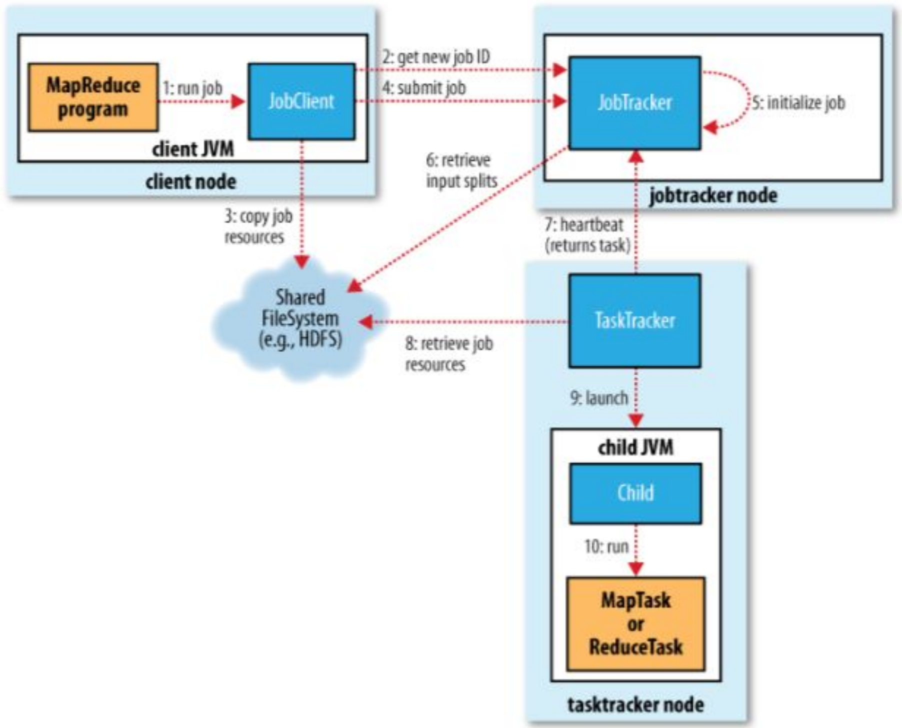
The new implementation (called MapReduce 2) is built on a system called YARN. The framework that is used for execution is set by the mapreduce.framework.name property, which takes the values local (for the local job runner), classic (for the "classic" MapReduce framework, also called MapReduce 1, which uses a jobtracker and tasktrackers), and yarn(for the new framework).

Reference: Hadoop, The Definitive Guide, Tom White

Map Reduce 1 (Classic Map Reduce)

At the highest level, there are four independent entities:

- The client, which submits the MapReduce job
- The jobtracker, which coordinates the job run. The jobtracker is a Java application whose main class is JobTracker
- The tasktrackers, which run the tasks that the job has been split into. Tasktrackers are Java applications whose main class is TaskTracker
- HDFS, which is used for sharing job files between the other entities.



The job submission process implemented by JobSummitter does the following:

- Asks the jobtracker for a new job ID (step 2)
- Checks the output specification of the job. For example, if the output directory has not been specified or it already exists, the job is not submitted and an error is thrown to the MapReduce program
- Computes the input splits for the job. If the splits cannot be computed, because the input paths don't exist, for example, then the job is not submitted and an error is thrown to the MapReduce program
- Copies the resources needed to run the job, including the job JAR file, the configuration file, and the computed input splits, to the jobtracker's filesystem in a directory named after the job ID. The job JAR is copied with a high replication factor (controlled by the mapred.submit.replication property, which defaults to 10) so that there are lots of copies across the cluster for the tasktrackers to access when they run tasks for the job (step 3).
- • Tells the jobtracker that the job is ready for execution (step 4).

MR1 execution framework - high level view

Job Initialization

When the JobTracker receives a call to its submitJob() method, it puts it into an internal queue from where the job scheduler will pick it up and initialize it. Initialization involves creating an object to represent the job being run, which encapsulates its tasks, and bookkeeping information to keep track of the tasks' status and progress (step 5).

To create the list of tasks to run, the job scheduler first retrieves the input splits computed by the client from the shared filesystem (step 6). It then creates one map task for each split. The number of reduce tasks to create is determined by the mapred.reduce.tasks property in the Job, and the scheduler simply creates this number of reduce tasks to be run.

In addition to the map and reduce tasks, two further tasks are created: a job setup task and a job cleanup task. These are run by tasktrackers and are used to run code to setup the job before any map tasks run, and to cleanup after all the reduce tasks are complete.

Task Assignment

Tasktrackers run a simple loop that periodically sends heartbeat method calls to the jobtracker. Heartbeats tell the jobtracker that a tasktracker is alive, but they also double as a channel for messages. As a part of the heartbeat, a tasktracker will indicate whether it is ready to run a new task, and if it is, the jobtracker will allocate it a task, which it communicates to the tasktracker using the heartbeat return value (step 7).

Before it can choose a task for the tasktracker, the jobtracker must choose a job to select the task from. There is a list of schedulers, but the default one simply maintains a priority list of jobs. Having chosen a job, the jobtracker now chooses a task for the job.

Tasktrackers have a fixed number of slots for map tasks and for reduce tasks: for example, a tasktracker may be able to run two map tasks and two reduce tasks simultaneously. (The precise number depends on the number of cores and the amount of memory on the tasktracker. The default scheduler fills empty map task slots before reduce task slots, so if the tasktracker has at least one empty map task slot, the jobtracker will select a map task; otherwise, it will select a reduce task.

To choose a reduce task, the jobtracker simply takes the next in its list of yet-to-be-run reduce tasks, since there are no data locality considerations. For a map task, however, it takes account of the tasktracker’s network location and picks a task whose input split is as close as possible to the tasktracker. In the optimal case, the task is *data-local*, that is, running on the same node that the split resides on. Alternatively, the task may be *rack-local*: on the same rack, but not the same node, as the split.

Task Execution

Now that the tasktracker has been assigned a task, the next step is for it to run the task. First, it localizes the job JAR by copying it from the shared filesystem to the tasktracker's filesystem. It also copies any files needed from the distributed cache by the application to the local disk. Second, it creates a local working directory for the task, and un-jars the contents of the JAR into this directory. Third, it creates an instance of TaskRunner to run the task.

TaskRunner launches a new Java Virtual Machine (step 9) to run each task in (step 10), so that any bugs in the user-defined map and reduce functions don't affect the task- tracker (by causing it to crash or hang, for example). It is, however, possible to reuse the JVM between tasks.

The child process communicates with its parent through the *umbilical* interface. This way it informs the parent of the task's progress every few seconds until the task is complete.

Job Completion

When the jobtracker receives a notification that the last task for a job is complete (this will be the special job cleanup task), it changes the status for the job to "successful." Then, when the Job polls for status, it learns that the job has completed successfully, so it prints a message to tell the user.

The jobtracker also sends an HTTP job notification if it is configured to do so. This can be configured by clients wishing to receive callbacks, via the job.end.notification.url property.

Last, the jobtracker cleans up its working state for the job and instructs tasktrackers to do the same (so intermediate output is deleted, for example).

Anatomy of a Mapreduce job

Mapper

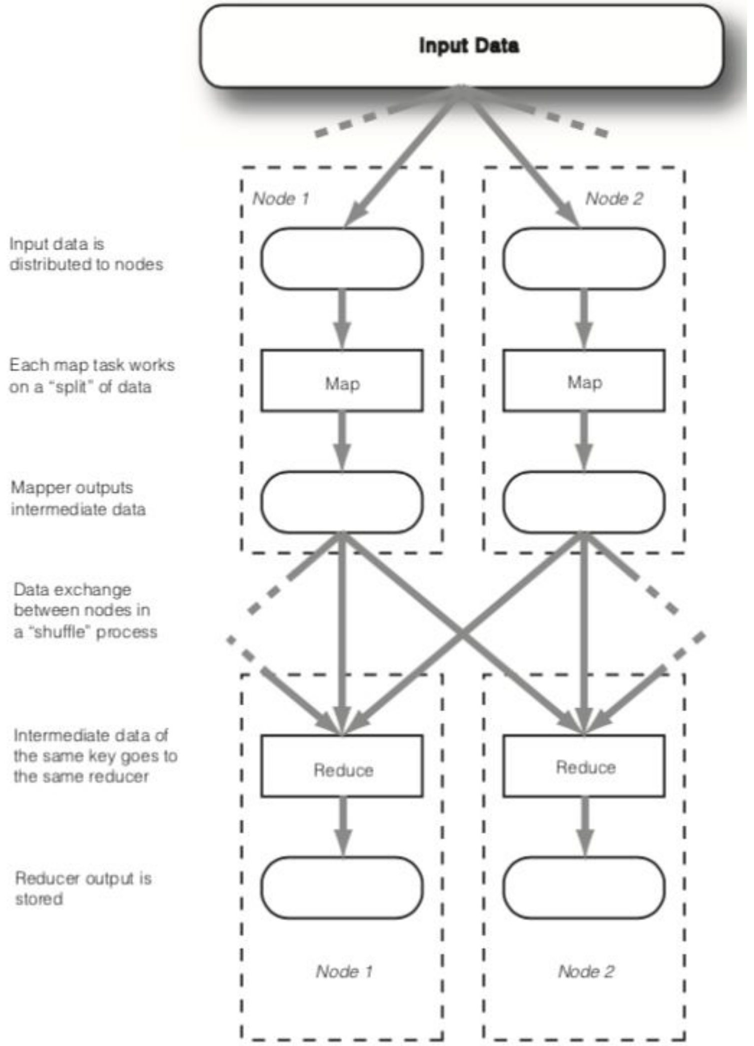
To serve as the mapper, a class implements from the Mapper interface and inherits the MapReduceBase class. The MapReduceBase class serves as the base class for both mappers and reducers.

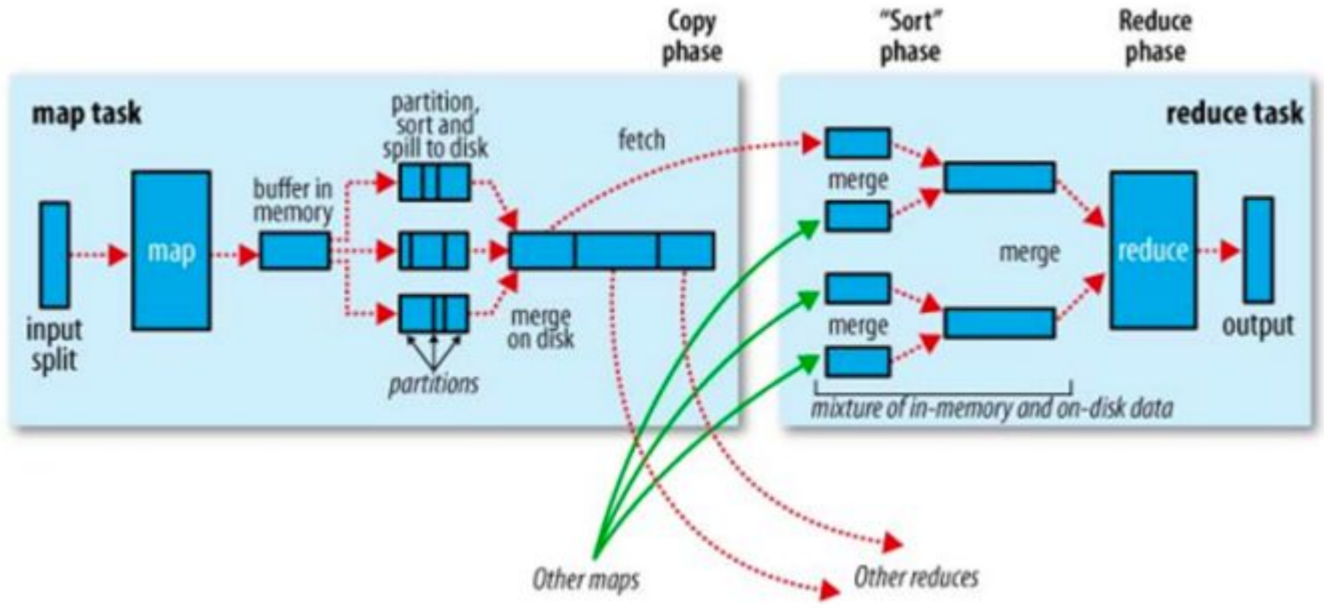
The Mapper interface is responsible for the data processing step. It utilizes Java generics of the form Mapper<K1,V1,K2,V2> where the key classes and value classes implement the WritableComparable and Writable interfaces,respectively. Its single method is to process an individual (key/value) pair.

The mapper function generates a list of (K2, V2) pairs for a given (K1, V1) input pair. An OutputCollector receives the output of the mapping process, and a Reporter provides the option to record extra information about the mapper as the task progresses.

Partitioner—redirecting output from Mapper

With multiple reducers, we need some way to determine the appropriate one to send a (key/value) pair outputted by a mapper. The default behavior is to hash the key to determine the reducer. Hadoop enforces this strategy by use of the HashPartitioner class.


Input Data
Node 1
Node 2
Input data is distributed to nodes
Each map task works on a "split" of data
Map
Map
Mapper outputs intermediate data
Data exchange between nodes in a "shuffle" process
Intermediate data of the same key goes to the same reducer
Reduce
Reduce
Reducer output is stored
Node 1
Node 2



Combiner—local reduce

In many situations with MapReduce applications, we may wish to perform a "local reduce" before we distribute the mapper results. Consider the WordCount example. If the job processes a document containing the word "the" 574 times, it's much more efficient to store and shuffle the pair ("the", 574) once instead of the pair ("the", 1) multiple times. This processing step is known as combining and is used on a need basis.

Shuffle

Data is moved from maps to reduers.

Reducer

When the reducer task receives the output from the various mappers, it sorts the incoming data on the key of the (key/value) pair and groups together all values of the same key and execute the required function.

MR1 execution stages

Job submission

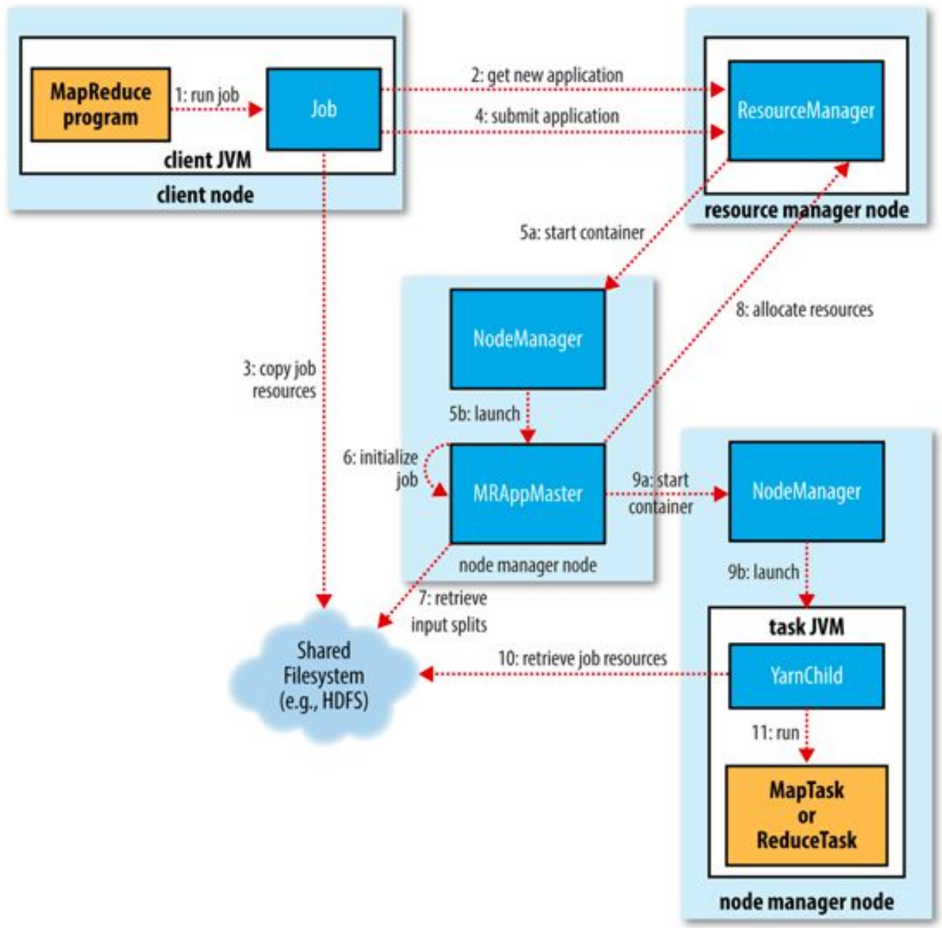
Map task execution

Partition

Shuffle and sort

Reduce task execution

Mapreduce 2 (YARN)



YARN meets the scalability shortcomings of “classic” MapReduce by splitting the responsibilities of the jobtracker into separate entities. The jobtracker takes care of both job scheduling (matching tasks with tasktrackers) and task progress monitoring (keeping track of tasks and restarting failed or slow tasks, and doing task bookkeeping such as maintaining counter totals).

YARN separates these two roles into two independent daemons: a *resource manager* to manage the use of resources across the cluster, and an *application master* to manage the lifecycle of applications running on the cluster. The idea is that an application master negotiates with the resource manager for cluster resources—described in terms of a number of *containers* each with a certain memory limit—then runs application- specific processes in those containers. The containers are overseen by *node managers* running on cluster nodes, which ensure that the application does not use more resources than it has been allocated.

In contrast to the jobtracker, each instance of a MapReduce job has a dedicated application master, which runs for the duration of the application.

MapReduce on YARN involves more entities than classic MapReduce. They are:

- The client, which submits the MapReduce job.
- The YARN resource manager, which coordinates the allocation of compute resources on the cluster.
- The YARN node managers, which launch and monitor the compute containers on machines in the cluster.
- The MapReduce application master, which coordinates the tasks running the MapReduce job. The application master and the MapReduce tasks run in containers that are scheduled by the resource manager, and managed by the node managers.
- The distributed filesystem, which is used for sharing job files between the other entities.

Job Submission

The submission process is very similar to the classic implementation. The new job ID is retrieved from the resource manager (rather than the jobtracker), although in the nomenclature of YARN it is an application ID. The job client checks the output specification of the job; computes input splits; and copies job resources (including the job JAR, configuration, and split information) to HDFS. Finally, the job is submitted.

Job Initialization

The scheduler allocates a container, and the resource manager then launches the application master's process there, under the node manager's management.

The application master for MapReduce jobs is a Java application whose main class is MRAppMaster. It initializes the job by creating a number of bookkeeping objects to keep track of the job's progress, as it will receive progress and completion reports from the tasks. Next, it retrieves the input splits computed in the client from the shared filesystem. It then creates a map task object for each split, and a number of reduce task objects determined by the mapreduce.job.reduces property.

The next thing the application master does is decide how to run the tasks that make up the MapReduce job. If the job is small, the application master may choose to run them in the same JVM as itself, since it judges the overhead of allocating new containers and running tasks in them as outweighing the gain to be had in running them in parallel, compared to running them sequentially on one node. (This is different to MapReduce 1, where small jobs are never run on a single tasktracker.) Such a job is said to be *uberized*, or run as an *uber task*.

What qualifies as a small job? By default one that has less than 10 mappers, only one reducer, and the input size is less than the size of one HDFS block.

Before any tasks can be run the job's output directory is created. In contrast to MapReduce 1, where it is called in a special task that is run by the tasktracker, in the YARN implementation the method is called directly by the application master.

Task Assignment

If the job does not qualify for running as an uber task, then the application master requests containers for all the map and reduce tasks in the job from the resource manager. Each request, which are piggybacked on heartbeat calls, includes information about each map task's data locality, in particular the hosts and corresponding racks that the input split resides on. The scheduler uses this information to make scheduling decisions (just like a jobtracker's scheduler does): it attempts to place tasks on data-local nodes in the ideal case, but if this is not possible the scheduler prefers rack-local placement to non-local placement.

Requests also specify memory requirements for tasks. By default both map and reduce tasks are allocated 1024 MB of memory, but this is configurable by setting mapreduce.map.memory.mb and mapreduce.reduce.memory.mb.

The way memory is allocated is different to MapReduce 1, where tasktrackers have a fixed number of “slots”, set at cluster configuration time, and each task runs in a single slot. Applications may request a memory capability that is anywhere between the minimum allocation and a maximum allocation, and which must be a multiple of the minimum allocation. Default memory allocations are scheduler-specific, and for the capacity scheduler the default minimum is 1024 MB (set by yarn.scheduler.capacity.minimum-allocation-mb), and the default maximum is 10240 MB (set by yarn.scheduler.capacity.maximum-allocation-mb). Thus, tasks can request any memory allocation between 1 and 10 GB (inclusive), in multiples of 1 GB (the scheduler will round to the nearest multiple if needed), by setting mapreduce.map.memory.mb and mapreduce.reduce.memory.mb appropriately.

Task Execution

Once a task has been assigned a container by the resource manager’s scheduler, the application master starts the container by contacting the node manager. The task is executed by a Java application whose main class is YarnChild. Before it can run the task it localizes the resources that the task needs, including the job configuration and JAR file, and any files from the distributed cache. Finally, it runs the map or reduce task.

The YarnChild runs in a dedicated JVM, for the same reason that tasktrackers spawn new JVMs for tasks in MapReduce 1: to isolate user code from long-running system daemons. Unlike MapReduce 1, however, YARN does not support JVM reuse so each task runs in a new JVM.

Failure handling in Classic MR

Task Failure

- The child JVM reports the error back to its parent tasktracker, before it exits. The error ultimately makes it into the user logs. The tasktracker marks the task attempt as *failed*, freeing up a slot to run another task.
- Another failure mode is the sudden exit of the child JVM—perhaps there is a JVM bug that causes the JVM to exit for a particular set of circumstances exposed by the MapReduce user code. In this case, the tasktracker notices that the process has exited and marks the attempt as failed.
- Hanging tasks are dealt with differently. The tasktracker notices that it hasn't received a progress update for a while and proceeds to mark the task as failed. The child JVM process will be automatically killed after this period. The timeout period after which tasks are considered failed is normally 10 minutes and can be configured on a per-job
- When the jobtracker is notified of a task attempt that has failed (by the tasktracker's heartbeat call), it will reschedule execution of the task.
- The jobtracker will try to avoid rescheduling the task on a tasktracker where it has previously failed. Furthermore, if a task fails four times (or more), it will not be retried further. This value is configurable: the maximum number of attempts to run a task is controlled by the mapred.map.max.attempts property for map tasks and mapred.reduce.max.attempts for reduce tasks. By default, if any task fails four times (or whatever the maximum number of attempts is configured to), the whole job fails.
- The maximum percentage of tasks that are allowed to fail without triggering job failure can be set for the job. Map tasks and reduce tasks are controlled independently, using the mapred.max.map.failures.percent and mapred.max.reduce.failures.percent properties.

Tasktracker Failure

- If a tasktracker fails by crashing, or running very slowly, it will stop sending heartbeats to the jobtracker (or send them very infrequently)
- The jobtracker will notice a tasktracker that has stopped sending heart- beats (if it hasn't received one for 10 minutes, configured via the mapred.task tracker.expiry.interval property, in milliseconds) and remove it from its pool of tasktrackers to schedule tasks on
- The jobtracker arranges for map tasks that were run and completed successfully on that tasktracker to be rerun if they belong to incomplete jobs, since their intermediate output residing on the failed tasktracker's local filesystem may not be accessible to the reduce task. Any tasks in progress are also rescheduled
- A tasktracker can also be blacklisted by the jobtracker, even if the tasktracker has not failed. If more than four tasks from the same job fail on a particular tasktracker (set by (mapred.max.tracker.failures), then the jobtracker records this as a fault. A tasktracker is blacklisted if the number of faults is over some minimum threshold (four, set by mapred.max.tracker.blacklists) and is significantly higher than the average number of faults for tasktrackers in the cluster cluster.
- Blacklisted tasktrackers are not assigned tasks, but they continue to communicate with the jobtracker. Faults expire over time (at the rate of one per day), so tasktrackers get the chance to run jobs again simply by leaving them running. Alternatively, if there is an underlying fault that can be fixed (by replacing hardware, for example), the task- tracker will be removed from the jobtracker's blacklist after it restarts and rejoins the cluster.

Jobtracker Failure

- Failure of the jobtracker is the most serious failure mode. Hadoop has no mechanism for dealing with failure of the jobtracker—it is a single point of failure—so in this case the job fails. However, this failure mode has a low chance of occurring, since the chance of a particular machine failing is low.
- After restarting a jobtracker, any jobs that were running at the time it was stopped will need to be re-submitted.

Failure handling in YARN

Task Failure

- Failure of the running task is similar to the classic case.

Application Master Failure

- Just like MapReduce tasks are given several attempts to succeed (in the face of hardware or network failures) applications in YARN are tried multiple times in the event of failure. By default, applications are marked as failed if they fail once, but this can be increased by setting the property yarn.resourcemanager.am.max-retries.
- An application master sends periodic heartbeats to the resource manager, and in the event of application master failure, the resource manager will detect the failure and start a new instance of the master running in a new container (managed by a node manager).
- In the case of the MapReduce application master, it can recover the state of the tasks that had already been run by the (failed) application so they don't have to be rerun.
- The client polls the application master for progress reports, so if its application master fails the client needs to locate the new instance. During job initialization the client asks the resource manager for the application master's address, and then caches it, so it doesn't overload the the resource manager with a request every time it needs to poll the application master. If the application master fails, however, the client will experience a timeout when it issues a status update, at which point the client will go back to the resource manager to ask for the new application master's address.

Node Manager Failure

- If a node manager fails, then it will stop sending heartbeats to the resource manager, and the node manager will be removed from the resource manager’s pool of available nodes. Any task or application master running on the failed node manager will be recovered using the mechanisms described in the previous two sections.
- Node managers may be blacklisted if the number of failures for the application is high. Blacklisting is done by the application master, and for MapReduce the application master will try to reschedule tasks on different nodes if more than three tasks fail on a node manager. The threshold may be set with mapreduce.job.maxtaskfailures.per.tracker.

Resource Manager Failure

- Failure of the resource manager is serious, since without it neither jobs nor task containers can be launched.
- After a crash, a new resource manager instance is brought up (by an administrator) and it recovers from the saved state. The state consists of the node managers in the system as well as the running applications.

Node failure management

A common design for setting up a backup NameNode server is by reusing the SNN. After all, the SNN has similar hardware specs as the NameNode, and Hadoop should've already been installed with the same directory configurations.

Network File System (NFS): Namenode data can be written to a network file system so it isn't necessarily lost if namenode is lost

To be safer, this new NameNode should *also* have a backup node set up before you start it. Otherwise you'll be in trouble if this new NameNode fails too. If you don't have a machine readily available as a backup, you should at least set up an NFS-mounted directory. This way the filesystem's state information is in more than one location.

As HDFS writes its metadata to all directories listed in dfs.name.dir, if NameNode has multiple hard drives, you can specify directories from different drives to hold replicas of the metadata. This way if one drive fails, it's easier to restart the NameNode without the bad drive than to switch over to the backup node, which involves moving the IP address, setting up a new backup node, and so on.

Recall that the SNN creates a snapshot of the filesystem's metadata in the fs.checkpoint.dir directory. As it checkpoints only periodically (once an hour under the default setup), the metadata is too stale to rely on for failover. But it's still a good idea to archive this directory periodically over to remote storage. In catastrophic situations, recovering from stale data is better than no data at all. This can be true if both the NameNode and the backup fail simultaneously (say, a power surge affecting both machines). Another unfortunate scenario is if the filesystem's metadata has been corrupted (say, by human error or a software bug) and has poisoned all the replicas.

Consider Namenode high availability

Adding a new datanode

When not using dfs.hosts.include property

1. Set up the node with required software (ensure to have the same versions as other nodes)
2. Add the new node’s hostname in the slaves file on primary node.
3. If using rack awareness, update rack files/scripts
4. Update replication factor on all nodes. Copy configurations across to new datanode

When using dfs.hosts.include property

1. If using dfs.hosts.include property, name the file dfs.include, place it under $HADOOP_CONF_DIR
2. Add property dfs.hosts.include in hdfs-site.xml, value being the name of the file above
3. add the new DNS to dfs.include file
4. Perform all steps as before
5. Execute hadoop dfsadmin -refreshNodes

Now:

1. Start datanode and nodemanager on new nodes
2. Verify: either dfsadmin -report, or from web consoles to confirm addition of a new node
3. Run the balancer utility, if cluster is not balanced: hdfs balancer

Decommissioning a node

1. Add the IP address of the datanode to the file specified by the dfs.hosts.exclude parameter. Each entry should be separated by a newline character. Name the file dfs.exclude, place it under $HADOOP_CONF_DIR
2. Execute the command hadoop dfsadmin -refreshNodes as the HDFS superuser or a user with equivalent privileges.
3. Monitor the namenode web UI and confirm the decommission process is in progress. It can take a few seconds to update.
4. Stop the datanode process on the decommissioned node.
5. If you do not plan to reintroduce the machine to the cluster, remove it from the HDFS include and exclude files as well as any rack topology database, and slaved file.
6. Execute the command hadoop dfsadmin -refreshNodes to have the namenode pick up the removal.

Similar steps need following for adding/removing jobtrackers in MR1.

Hadoop backup & restore

Common filesystem image backups

Distcp command

1. hadoop distcp hdfs://pri-node:9000/<source folder> hdfs://d-node-a:9000/<target folder>
2. Folders shall exist
3. Must have required permissions
4. Passwordless ssh must be set up for user between both servers
5. This runs as a mapreduce job
6. Cluster versions must be compatible. Use webhdfs for source if copying between major versions

Snapshots

HDFS Snapshots are read-only point-in-time copies of the file system. Snapshots can be taken on a subtree of the file system or the entire file system. Some common use cases of snapshots are data backup, protection against user errors and disaster recovery. The implementation of HDFS Snapshots is efficient. Blocks in datanodes are not copied: the snapshot files record the block list and the file size. To restore either copy or distcp from a snapshot to initial locations.

```
hdfs dfsadmin -allowSnapshot /userdata
hdfs dfs -createSnapshot /userdata
hdfs dfs -ls /userdata/.snapshot
```

Hadoop Configuration

Basics of configuration:

- What is required to be done
- Where to configure - configuration files
- What to configure - properties
- What values to set
- What impact will it have

Ways to check existing configuration:

curl 'http://localhost:50070/conf' > <to your local file> for future reference

hdfs getconf -confKey <key name>

Configuration basics

Identify key configuration files

Understand the purpose

Properties, values

Restart needed or not

Hadoop Configuration

Filename	*Format*	*Description*
hadoop-env.sh	Bash script	Environment variables that are used in the scripts to run Hadoop.
core-site.xml	Hadoop configuration XML	Configuration settings for Hadoop Core, such as I/O settings that are common to HDFS and MapReduce.
hdfs-site.xml	Hadoop configuration XML	Configuration settings for HDFS daemons: the namenode, the secondary namenode, and the datanodes.
mapred-site.xml	Hadoop configuration XML	Configuration settings for MapReduce daemons: the jobtracker, and the tasktrackers.
masters	Plain text	A list of machines (one per line) that each run a secondary namenode.
slaves	Plain text	A list of machines (one per line) that each run a datanode and a tasktracker.
Hadoop-metrics.properties (or metrics2)	Java Properties	Properties for controlling how metrics are published in Hadoop
log4j.properties	Java Properties	Properties for system log files, the namenode audit log, and the task log for the tasktracker child process

Some properties

Refer to sent attachments, tab Properties

Administration, Hadoop shell commands reference

Filesystem: https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HDFSCommands.html

Yarn: https://hadoop.apache.org/docs/r2.7.3/hadoop-yarn/hadoop-yarn-site/YarnCommands.html

Hadoop: https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/CommandsManual.html

Mapreduce:
https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapredCommands.html

HDFS web console:	http://localhost:50070
Resource manager web console:	http://localhost:8088

```
hdfs dfs -fsck <path>
```

perform filesystem check, view block info, view replication info etc

```
hdfs dfs
```

```
-ls, -mkdir, -cat, -put, -cp, -mv, -rm
```

The following help get configuration information

```
hdfs getconf -namenodes

hdfs getconf -secondaryNameNodes

hdfs getconf -confKey [key]
```

hdfs balancer   to balance data on a cluster (test by hdfs dfs -Ddfs.replication=1 -put <src path> <hdfs path>)

***remove the big file you stored already.

hdfs dfsadmin [-report [-live] [-dead] [-decommissioning] [-enteringmaintenance] [-inmaintenance]]

hdfs dfsadmin [-safemode enter | leave | get | wait | forceExit]

hdfs dfsadmin [-refreshNodes]

hdfs dfsadmin [-refreshUserToGroupsMappings]

hdfs dfsadmin [-refreshSuperUserGroupsConfiguration]

hdfs dfsadmin [-printTopology]

hdfs dfsadmin [-refreshNamenodes datanodehost:port]

```
hdfs dfsadmin [-deleteBlockPool datanode-host:port blockpoolId [force]]

hdfs dfsadmin [-allowSnapshot <snapshotDir>]

hdfs dfsadmin [-disallowSnapshot <snapshotDir>]

hdfs dfsadmin [-getDatanodeInfo <datanode_host:ipc_port>]

hdfs dfsadmin [-triggerBlockReport [-incremental] <datanode_host:ipc_port>]

hdfs dfsadmin [-listOpenFiles [-blockingDecommission] [-path <path>]]

hdfs dfsadmin [-help [cmd]]
hdfs namenode [-backup] |

         [-checkpoint] |

         [-format [-clusterid cid ] [-force] [-nonInteractive] ] |

         [-rollback] |

         [-bootstrapStandby [-force] [-nonInteractive] [-skipSharedEditsCheck] ] |

         [-recover [-force] ] |

         [-metadataVersion ]
```

```
mapred job | [GENERIC_OPTIONS] | [-submit <job-file>] |

[-status <job-id>] | [-counter <job-id> <group-name> <counter-name>] | [-kill <job-id>] |

[-events <job-id> <from-event-#> <#-of-events>] |

[-history [all] <jobHistoryFile|jobId> [-outfile <file>] [-format <human|json>]] | [-list [all]] |

[-kill-task <task-id>] | [-fail-task <task-id>] | [-set-priority <job-id> <priority>] | [-list-active-trackers] |

[-list-blacklisted-trackers] | [-list-attempt-ids <job-id> <task-type> <task-state>] [-logs <job-id> <task-attempt-id>] [-config
<job-id> <file>]
```

Logging Configuration

Package: log4j

File name: log4j.properties

Location: ${HADOOP_CONFIG_HOME}

Format: standard Java properties file (i.e. key value pairs)

Purpose: controls the overall log levels of both the Hadoop daemons as well as MapReduce jobs that execute on the cluster

Log structure

$$\begin{tikzcd}
\text{What to log} \arrow[r] & \text{Logger} \\
\text{Where to log} \arrow[r] & \text{Appender} \\
\text{How should it look} \arrow[r] & \text{Layout}
\end{tikzcd}$$

Logger

- A named channel for log events that has a specified minimum log level
- The supported log levels, in order of most severe to least, are FATAL, ERROR, WARN, INFO, DEBUG, and TRACE
- The minimum log level acts as a filter: log events with a log level greater than or equal to that which is specified are accepted while less severe events are simply discarded
- Loggers are hierarchical; each logger has a parent logger from which it inherits its configuration information. At the top of the inheritance tree is a root logger which is a logger with no parent
- Loggers can be specified by using the naming convention of log4j.logger.logger-name. This is often the java class name generating events
- The hierarchical relationship of a logger is defined by dotted notation with descendants having their parent's prefix. For example, the logger org.apache is the parent of org.apache.hadoop, which is the parent of org.apache.hadoop.hdfs and so on
- The value of a logger parameter is always a log level, a comma, and the name of one or more appenders. The comma and appender list is optional, in which case, the logger inherits the appender of its parent

Appender

- Loggers output their log events to an appender which is responsible to handling the event in some meaningful way
- By far, the most commonly used appenders write log events to disk, but appenders for outputting log events to the console, sending data to syslog, or even to JMS exist
- Console - outputs logs to console
- RFA - Rolls over the file at certain thresholds of size
- DRFA - Daily Rotating File Appender: roll over the file daily
- Other appends exist as well
- Reference on appenders: https://logging.apache.org/log4j/2.x/manual/appenders.html

log4j.properties structure

```
hadoop.root.logger=INFO,console
hadoop.log.dir=.
hadoop.log.file=hadoop.log

# Define the root logger tthe system property "hadoop.root.logger".
log4j.rootLogger=${hadoop.root.logger}, EventCounter

# Logging Threshold
log4j.threshold=ALL

# Null Appender
log4j.appender.NullAppender=org.apache.log4j.varia.NullAppender
```

Update logging configuration

- Log file level changes: Update log4j.properties. Set log4j.logger.org.apache.hadoop=<logger>,<appender> Restart needed for changes to take effect. Try with logger TRACE and observe differences

- System wide changes: Update hadoop-env.sh (for ex. export HADOOP_ROOT_LOGGER=WARN,DRFA). No need to restart the cluster/node

- Changing logging for your login session only: on command line, export HADOOP_ROOT_LOGGER=<logger>,<appender> No restart needed

- hadoop daemonlog -setlevel utility. No restart needed, persists only until application is up

- To do: check logging in a multi node cluster on different clusters for the same job. Is there any difference if difference logging configurations are used for different nodes?

Monitoring

Health monitoring

Performance monitoring

Health monitoring

Daemons:

If each daemon running

within normal memory consumption limits

responding to RPC requests in a defined window, and other “simple” metrics

but this doesn’t tell us whether the entirety of the service is functional (although one may infer such things).

If a certain percentage of datanodes are alive and communicating with the namenode, or what the block distribution is across the cluster

Memory

Monitor to ensure that the number of pages (or amount of data in bytes, whatever is easier) swapped in and out tdisk, per second, does not exceed zero, or some very small amount.

Health monitoring

Host monitoring

The requisite local disk capacity, free memory, and minimal amount of CPU capacity. Monitor local disk consumption of the namenode metadata (dfs.name.dir) and log data (HADOOP_LOG_DIR) directories

Track CPU metrics such as load average for performance and utilization measurement

Network bandwidth consumption.

Perform heap monitoring on the namenode, jobtracker, and secondary namenode processes using the technique described. Use simple statistical techniques to isolate real problems

Monitor the average time spent performing garbage collection for the namenode and jobtracker. Tolerance for these pauses before failure occurs will vary by application, but almost all will be negatively affected in terms of performance.

Health Monitoring

hdfs checks

- Free HDFS capacity in bytes (Free) is over an acceptable threshold
- The absolute number of active (NameDirStatuses["active"]) metadata paths is equal to those specified in dfs.name.dir, or failed (NameDirStatuses["failed"]) paths is equal tzero.
- The absolute number of missing (MissingBlocks) and corrupt blocks (Corrupt Blocks) are lower than a acceptable threshold. Both of these metrics should be zero, ideally.
- The absolute number of HDFS blocks that can still be allocated (BlockCapacity).
- The result of the current epoch time minus the last time a namenode checkpoint was performed (LastCheckpointTime) is less than accepted threshold.

Mapreduce checks

- Check whether the number of alive nodes is within a tolerance that still allows your jobs to complete within their service-level agreement. Depending on the size of your cluster and the criticality of jobs, this will vary.
- Check whether the number of blacklisted tasktrackers is below some percentage of the total number of tasktrackers in the cluster.

Monitoring Contexts

Each daemon can be configured to collect this data from its internal components at a regular interval and then handle the metrics in some way using a plug-in.

Related metrics are grouped into a named *context*, and each context can be treated independently.

Some contexts are common to all daemons, such as the information about the JVM and RPC operations performed, and others apply only to daemons of a specific service, such as HDFS metrics that come from only the namenode and datanodes.

Each context can be individually configured with a plug-in that specifies how metric data should be handled.

The primary four contexts are:

Jvm

Contains Java virtual machine information and metrics. Example data includes the maximum heap size, occupied heap, and average time spent in garbage collection. All daemons produce metrics for this context.

Dfs

Contains HDFS metrics. The metrics provided vary by daemon role. For example, the namenode provides information about total HDFS capacity, consumed capacity, missing and under-replicated blocks, and active datanodes in the cluster, and datanodes provide the number of failed disk volumes and remaining capacity on that particular worker node. Only the HDFS daemons output metrics for this context.

Mapred

Contains MapReduce metrics. The metrics provided vary by daemon role. For example, the jobtracker provides information about the total number of map and reduce slots, blacklisted tasktrackers, and failures, whereas tasktrackers provide counts of running, failed, and killed tasks at the worker node level. Only MapReduce daemons output metrics for this context

Rpc

Contains remote procedure call metrics. Example data includes the time each RPC spends in the queue before being processed, the average time it takes to process an RPC, and the number of open connections. All daemons output metrics for this context.

Although all of Hadoop is instrumented to capture this information, it’s not available to external systems by default. One must configure a plug-in for each context to handle this data in some way.

The metrics system configuration is specified by the *hadoop-metrics.properties* file within the standard Hadoop configuration directory.

org.apache.hadoop.metrics.spi.NullContext

Hadoop's default metric plug-in for all four contexts, NullContext is the */dev/null*of plug-ins. Metrics are not collected from the internal components, nor are they output tany external system. This plug-in effectively disables access to metrics

org.apache.hadoop.metrics.spi.NoEmitMetricsContext

The NoEmitMetricsContext is a slight variation on NullContext—with an important difference. Although metrics are still not output tan external system, the thread that runs within Hadoop, updating the metric values in memory *does* run. Systems such as JMX and the metrics servlet use this

org.apache.hadoop.metrics.file.FileContext

FileContext polls the internal components of Hadoop for metrics periodically and writes them out ta file on the local filesystem. FileContext is flawed and shouldn't be used in production clusters because the plug-in never rotates the specified file, leading to indefinite growth.

Monitoring interfaces

Example usage:

```
http://localhost:50070/jmx?qry=Hadoop:name=FSNamesystem,service=NameNode

curl http://localhost:50070/jmx?qry=Hadoop:name=FSNamesystem,service=NameNode
```

Others:

```
Java.lang:type=Memory
name=FSDatasetState,service=DataNode
Hadoop:service=NameNode,name=FSNamesystem
Hadoop:service=DataNode,name=DataNodeInfo
hadoop:service=NameNode,name=NameNodeActivity
hadoop:service=DataNode,name=DataNodeActivity-hostnameport
hadoop:service=JobTracker,name=JobTrackerInfo
hadoop:service=TaskTracker,name=TaskTrackerInfo
hadoop:service=ServiceName,name=RpcActivityForPort1234
hadoop:service=ServiceName,name=RpcDetailedActivityForPort1234
```

Maintenance

- Software installation, configuration, upgrades
- Resource management (Node, CPU, memory, network, disk, daemon)
- Security management
- Backup and recovery
- Balancers, and filesystem checks
- System monitoring

Reference: http://www.hadoopadmin.co.in/hadoop-administration-and-maintenance/

Benchmarking

```
hadoop jar $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-examples*.jar teragen
-Dmapreduce.job.maps=1000 10t random-data
```
***map only job to generate specified number of rows of binary data of 100 bytes each. Generate a total of 1TB of data

```
hadoop jar $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-examples*.jar terasort random-data
sorted-data
```
****sort the above data

```
hadoop jar $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-examples*.jar teravalidate sorted-data report
```
****check whether the sort is accurate. Errors in report/part-r-0000 file

Other benchmarks
TestDFSI- tests performance of hardware
MRBench - runs a small job a number of times
NNBench - load testing nodename hardware
Gridmix - model a realistic cluster workload by mimicking data access patterns
SWIM - Statistical Workload Injector for Mapreduce, a repository of real life MR workloads tbe tested on system
TPCx-HS - standardised benchmark based on TeraSort

Schedulers

- Accidentally scheduling CPU, memory, or disk IO intensive tasks on the same host can cause contention
- In Hadoop MapReduce, the scheduler— a plug-in within the jobtracker—is the component that is responsible for assigning tasks to open slots on tasktrackers
- Additionally, map tasks have a locality preference that the scheduler must take into account
- Service level agreements that must be met when defined
- The scheduler plug-in is what decides what tasks, from what queues, should be processed on what tasktrackers, in what order
- There are different scheduling algorithms, each with their own benefits and optimizations, there are multiple scheduler plug-ins an administrator can choose from when configuring a cluster. Only one scheduler at a time may be configured, however.

- Each scheduler implements data locality logic in addition to other features they may support.
- Configured by setting: mapred.jobtracker.taskScheduler

The FIFO Scheduler

- The first in, first out (FIFO) scheduler is the default scheduler in Hadoop
- It uses a simple “first come, first served” algorithm for scheduling tasks. For example, given two jobs—A and B—submitted in that order, all map tasks in job A will execute before any tasks from job B. As job A map tasks complete, job B map tasks are scheduled
- This suffers from a monopolization problem. Any job that is subsequently submitted after a big job needs to wait a considerable amount of time before any tasks will be scheduled. From the outside, the job will appear to simply make no progress
- The FIFO scheduler supports five levels of job prioritization, from lowest to highest: very low, low, normal, high, very high
- Each priority is actually implemented as a separate FIFO queue.
- Beyond prioritized task scheduling, the FIFO scheduler does not offer much in the way of additional features (compared to what we’ll see later in the Capacity and Fair Schedulers).
- For small, experimental, or development clusters, the FIFO scheduler can be adequate. Production clusters, however, should use one of the other two schedulers covered next.

- Configured by setting mapred.task.scheduler to org.apache.hadoop.mapred.JobQueueTaskScheduler

The Fair Scheduler

- The Fair Scheduler solves some of the problems that arise when using the FIFO scheduler.
- Jobs are placed into pools.
- Each pool is assigned a number of task slots based on a number of factors including the total slot capacity of the cluster, the current demand (where “demand” is the number of tasks in a pool) on other pools, minimum slot guarantees, and available slot capacity. Pools may optionally have minimum slot guarantees.
- Beyond the minimum slot guarantees, each pool gets an equal number of the remaining available slots on the cluster; this is where the “fair share” portion of the name comes from.
- All pools simply receive an equal number of slots.
- A MapReduce job property defines how the scheduler determines to which pool a job (and really, it’s tasks) should be assigned. Again, a default value is provided, which is the property user.name.

- The first step in deciding how to allocate slots is to determine the total cluster capacity; and open slots.
- When assigning tasks, the scheduler first looks at the demand for each pool.
- A pool with no demand is not given any slots, even if it has a minimum share.
- The scheduler gives each pool with demand its minimum share—if there is one configured—before going any further.
- With the minimum shares satisfied, the scheduler switches to allocating the remaining slots.
- In addition to, or in place of a minimum share, pools may also have a weight. Pools with greater weight receive more slots during fair share allocation (weight does not impact minimum share allocation). The weight of a pool simply acts as a multiplier; a weight of 2 means the pool receives two slots to every one slot the other pools receive. By default, pools have a weight of 1.
- Job priorities, like those supported in the FIFO scheduler, are also supported in the Fair Scheduler.

The important takeaways from this are:

- Minimum shares are always satisfied before fair shares.
- Pools never receive more slots than their demand, even if there’s a minimum share in place.
- During fair share assignment, slots are allocated in an attempt to “fill the water glasses evenly.”
- Pools can a have a weight that is only considered during fair share allocation.
- For multiple jobs in the same pool, resources are divided
- Fair Scheduler does not reserve slots for pools configured with minimum shares unless there is demand for those pools.

- When a job is submitted to a pool with a minimum share and those slots have been given away, there are two options: wait for the running tasks to complete and take the slots as they free up, or forcefully reclaim the necessary resources promised by the minimum share configuration. So, the scheduler simply kills a task, which then goes back into the queue for retry at a later time. Any work done by a task that is killed is thrown away. While somewhat wasteful, this does accomplish the goal of keeping the resources where they’re needed most. To look at it another way, it’s *more* wasteful to leave capacity reserved by minimum shares unused even when there’s no work in those pools.

- There are two types of preemption: minimum share preemption and fair share pre- emption. Minimum share preemption occurs when a pool is operating below its configured minimum share, whereas fair share preemption kicks in only when a pool is operating below its fair share. Minimum share preemption is the more aggressive of the two.

- Another trick in the Fair Scheduler bag is *delayed task assignment* (sometimes called delay scheduling). The goal of delayed assignment is to increase the data locality hit ratio and as a result, the performance of a job, as well as the utilization of the cluster as a whole. Delayed assignment works by letting a free slot on a tasktracker remain open for a short amount of time if there is no queued task that would prefer to run on the host in question.

- Configured by setting mapred.jobtracker.taskScheduler org.apache.hadoop.mapred.FairScheduler.

Choose the Fair Scheduler over the Capacity Scheduler if:

- You have a slow network and data locality makes a significant difference to job runtime. Features like delay scheduling can make a dramatic difference in the effective locality rate of map tasks.
- You have a lot of variability in the utilization between pools.
- You require jobs *within* a pool to make equal progress rather than running in FIFO order.

The Capacity Scheduler

- The Capacity Scheduler is a simpler and in some ways, a more deterministic scheduler than the Fair Scheduler
- An administrator configures one or more queues, each with a *capacity*—a predetermined fraction of the total cluster slot capacity. This it is reserved for the queue in question and is not given away in the absence of demand.
- Slots are given to queues (analogous to the Fair Scheduler pools, in this context), with the most starved queues receive slots first.
- Queue starvation is measured by dividing the number of running tasks in the queue by the queue's capacity or in other words, its percentage used.
- Within a queue, jobs for the same user are FIFO ordered.
- Similar to the FIFO scheduler, however, jobs can be prioritized within a queue.
- This scheduler controls allocation based on physical machine resources. The previously covered schedulers work exclusively in terms of slots, but the Capacity Scheduler additionally understands scheduling tasks based on (user defined) memory consumption of a job's tasks as well.

- When properly configured, the scheduler uses information collected by the tasktracker to aid in scheduling decisions.
- An administrator may specify a default virtual and physical memory limit on tasks that users may optionally override upon job submission.
- The scheduler then uses this information to decide on which tasktracker to place the tasks, taking into account any other tasks currently executing on the host.
- Checks exist to ensure these so-called high memory jobs are not starved for resources in the face of jobs without such a requirement.
- Set by configuring mapred.jobtracker.taskScheduler to org.apache.hadoop.mapred.CapacityTaskScheduler

Choose the Capacity Scheduler over the Fair Scheduler if:

- You know a lot about your cluster workloads and utilization and simply want to enforce resource allocation.
- You have very little fluctuation within queue utilization. The Capacity Scheduler's more rigid resource allocation makes sense when all queues are at capacity almost all the time.
- You have high variance in the memory requirements of jobs and you need the Capacity Scheduler's memory-based scheduling support.
- You demand scheduler determinism.

Cluster planning

Prerequisites

In the following simulation/scenario, while setting up the cluster, we need to know the below parameters. While this is a simulation similar exercise can be followed for various scenarios:

- What is the volume of data for which the cluster is being set? (For example, 100 TB.)
- The retention policy of the data. (For example, 2 years.)
- The kinds of workloads you have — CPU intensive, i.e. query; I/O intensive, i.e. ingestion, memory intensive, i.e. Spark processing. (For example, 30% jobs memory and CPU intensive, 70% I/O and medium CPU intensive.)
- The storage mechanism for the data — plain Text/AVRO/Parque/Jason/ORC/etc. or compresses GZIP, Snappy. (For example, 30% container storage 70% compressed.)
- Functionalities to be supported
- Hadoop production versions available and their feature support
- Operating systems, other ecosystem components
- Skill sets available

Cluster planning

Data Nodes Requirements

- With the above parameters in hand, we can plan for commodity machines required for the cluster. The nodes that will be required depends on data to be stored/analyzed.
- By default, the Hadoop ecosystem creates three replicas of data. So if we go with a default value of 3, we need storage of *100TB *3=300 TB* for storing data of one year. We have a retention policy of two years, therefore, the storage required will be *1 year data* retention period=300*2=600 TB*.
- In addition to the data, we need space for processing/computation the data plus for some other tasks.
- We need to decide how much should go to the extra space.
- We also assume that on an average day, only 10% of data is being processed and a data process creates three times temporary data. So, we need around 30% of total storage as extra storage.
- As for the data node, JBOD is recommended. We need to allocate 20% of data storage to the JBOD file system. Therefore, the data storage requirement will go up by 20%.

Number of Data Nodes Required

Now, we need to calculate the number of data nodes required for storage calculated above (let’s assume it’s around 500 TB). Suppose we have a JBOD of 12 disks, each disk worth of 4 TB. Data node capacity will be 48 TB.

The number of required data nodes is ~ *10*.

In general, the number of data nodes required is *Node= DS/(no. of disks in JBOD*disk space per disk)*.

Note: We do not need to set up the whole cluster on the first day. We can scale up the cluster as data grows from small to big. We can start with 25% of total nodes to 100% as data grows.

Now, let's discuss data nodes for batch processing (Hive, MapReduce, Pig, etc.) and for in-memory processing.

As per our assumption, 70% of data needs to be processed in batch mode with Hive, MapReduce, etc.

10.70=7* nodes are assigned for batch processing and the other 3 nodes are for in-memory processing.

CPU Cores and Tasks per Node

For batch processing nodes, while one core is counted for CPU-heavy processes, .7 core can be assumed for medium-CPU intensive processes. As we have assumption, 30% heavy processing jobs and 70% medium processing jobs, Batch processing nodes can handle [(no. of cores* %heavy processing jobs/cores required to process heavy job)+ (no. of cores* %medium processing jobs/cores required to process medium job)]. Therefore tasks performed by data nodes will be;

12.30/1+12*.70*/.7=3.6+12=15.6 ~15* tasks per node.

As hyperthreading is enabled, if the task includes two threads, we can assume 15*2~30 tasks per node.

RAM Requirement for a Data Node

Now, let's calculate RAM required per data node. RAM requirements depend on the below parameters.

> *RAM Required=DataNode process memory+DataNode TaskTracker memory+OS memory+CPU's core number *Memory per CPU core*

At the starting stage, we have allocated four GB memory for each parameter, which can be scaled up as required. Therefore, RAM required will be *RAM=4+4+4+12*4=60* GB RAM for batch data nodes and *RAM=4+4+4+16*4=76* GB for in-memory processing data nodes.

The steps defined above give us a fair understanding of resources required for setting up data nodes in Hadoop clusters, which can be further fine-tuned. In next blog, I will focus on capacity planning for name node and Yarn configuration.

Operating System selection

A significant number of production clusters run on RedHat Enterprise Linux or its freely available sister, CentOS. Ubuntu, SuSE Enterprise Linux, and Debian deployments also exist in production and work perfectly well. Your choice of operating system may be influenced by administration tools, hardware support, or commercial software support; the best choice is usually to minimize the variables and reduce risk by picking the distribution with which you’re most comfortable.

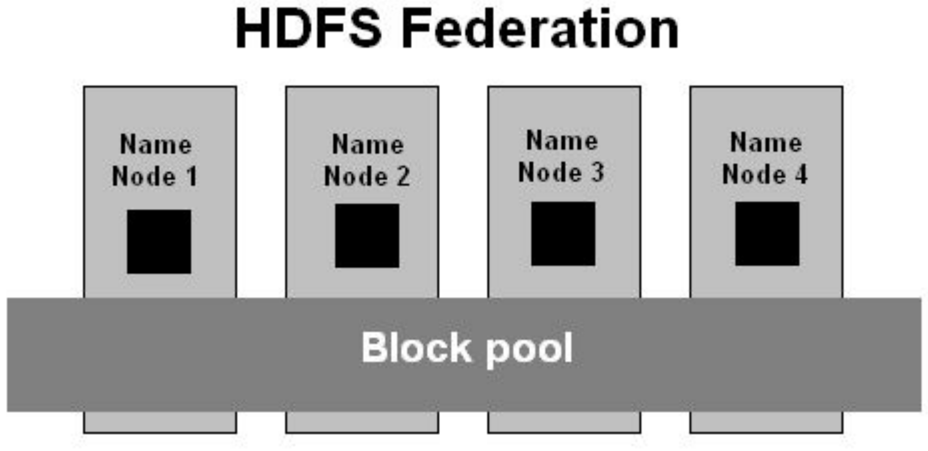
Preparing the OS for Hadoop requires a number of steps, and repeating them on a large number of machines is both time-consuming and error-prone. For this reason, it is strongly advised that a software configuration management system be used. Puppet and Chef are two open source tools that fit the bill. Chosen OS might need to be configured for following:

Network details

- Network usage happens due to data movement, heartbeats, block reports, metadata operations etc
- Primarily, it is the data moving across nodes in non-data local operations that consists of network traffic
- Data primarily moves in the east-west direction more than the north-south direction in hadoop setups
- Hadoop does not require one or the other; however, it can benefit from the additional bandwidth and lower latency of 10 Gb connectivity.
- Bandwidth requirements can be concluded by understanding the workloads. For ex, analytical jobs might not use a lot of network while data processing job might consume considerable traffic

References: cluster-planning.pdf - emailed separately, Hadoop Operations, Eric Sammers

Namenode federation



Federation is a feature new to Apache Hadoop 2.0 and CDH4, created to overcome the limitation that all filesystem metadata must fit in memory

It differs from namenode high availability in that rather than a single namespace being served from one of two possible namenodes, multiple namenodes each serve a *different* slice of a larger namespace.

It’s possible to enable either federation or high availability, or even both simultaneously. Sometimes, federation is used to provide a different level of service to a slice of a global namespace. For example, it may be necessary to tune garbage collection parameters or enable HA for the path */hbase* but not the rest of the namespace

Namenode federation

A client is unaware that it may be talking to different namenodes when it accesses different paths.

Clients of HDFS use a specialized plugin called *ViewFS* to view the logical, global namespace as a single entity. In other words,

In this way, federation is similar to Linux where there are physical devices that are mounted at a given path using */etc/fstab*.

Each datanode in a federated cluster stores blocks for each namenode.

When each namenode is formatted, it generates a *block pool* in which block data associated with that namenode is stored.

Each datanode, in turn, stores data for multiple block pools, and communicates with each namenode. This achieves better total utilization of datanode capacity.

Security in hadoop

There are 3 basic questions:

1. Who does this user claim to be?
2. Can this user prove they are who they say they are?
3. Is this user allowed to do what they're asking to do?

Hadoop has following security modes:

Simple mode - relying on OS for authentication

Secure mode - authentication via other services, ex, Kerberos, LDAP etc

Authentication must always be performed before authorization is considered, and because it is commonly the same for all services, it can be built as a separate, generic service.

Kerberos

When operating in secure mode, all clients must provide a valid Kerberos ticket that can be verified by the server.

In addition to clients being authenticated, daemons are also verified. In the case of HDFS, for instance, a datanode is not permitted to connect to the namenode unless it provides a valid ticket within each RPC.

A user in Kerberos is called a *principal*, which is made up of three distinct components: the primary, instance, and realm. The first component of the principal is called the *primary*, or sometimes the user component. The primary component is the operating system username of the user or the name of a service.

I*nstance is optional*, and is used to create principals that are used by users in special roles or to define the host on which a service runs.

aakash@HA.CDAC.COM User aakash in realm HA.CDAC.COM.

Kerberos

At its core, Kerberos provides a central, trusted service called the Key Distribution Center or *KDC*. The KDC is made up of two distinct services

The authentication server (*AS*), which is responsible for authenticating a client and providing a ticket granting ticket (*TGT*).

The ticket granting service (*TGS*), which, given a valid TGT, can grant a ticket that authenticates a user when communicating with a Kerberos-enabled (or *Kerberized*) service.

The KDC contains a database of principals and their keys, very much like */etc/passwd*.

How does it work

- Let’s say you want to run: hadoop fs -ls / userdata/input/iliad.csv
- When operating in secure mode, the HDFS namenode and datanode will not permit any communication that does not contain a valid Kerberos ticket.
- We also know that at least two (and frequently many more) services must be contacted: one is the namenode to get the file metadata and check permissions, and the rest are the datanodes to retrieve the blocks of the file.
- To obtain any tickets from the KDC, we first retrieve a TGT from the AS by providing our principal name.
- The TGT, which is only valid for an administrator-defined period of time, is encrypted with our password and sent back to the client.
- The client prompts us for our password and attempts to decrypt the TGT.
- If it works, we’re ready to request a ticket from the TGS, otherwise we’ve failed to decrypt the TGT and we’re unable to request tickets.
- It’s important to note that our password has never left the local machine; the system works because the KDC has a copy of the password, which has been shared in advance. This is a standard *shared secret* or *symmetric key* encryption model.

How does it work

- It is still not yet possible to speak to the namenode or datanode; we need to provide a valid ticket for those specific services. Now that we have a valid TGT, we can request service specific tickets from the TGS.
- To do so, using our TGT, we ask the TGS for a ticket for a specific service, identified by the service principal (such as the namenode of the cluster).
- If the TGT can be validated and it hasn’t yet expired, the TGS provides us a valid ticket for the service, which is also only valid for a finite amount of time.
- Using this ticket, we can now contact the namenode and request metadata required.
- The namenode will validate the ticket with the KDC and assuming everything checks out, then performs the operation we originally requested.

Kerberos Support in Hadoop

- There are two primary forms of authentication that occur in Hadoop with respect to Kerberos: nodes within the cluster authenticating with one another to ensure that only trusted machines are part of the cluster, and users, both human and system, that access the cluster to interact with services.
- Since many of the Hadoop daemons also have embedded web servers, they too must be secured and authenticated.
- Within each service, both users and worker nodes are verified by their Kerberos credentials.
- HDFS and MapReduce follow the same general architecture; the worker daemons are each given a unique principal that identifies each daemon.
- Since worker nodes run both a datanode as well as a tasktracker, each node requires two principals to be generated: one for the datanode and one for the task- tracker.
- Since it isn't feasible to log into each machine and execute kinit as both user hdfs andmapred and provide a password, the keys for the service principals are exported to files and placed in a well-known location. These files are referred to as key tables or just *keytabs*.
- When the daemons start up, they use this keytab to authenticate with the KDC and get a ticket so they can connect to the namenode or jobtracker, respectively. When operating in secure mode, it is not possible for a datanode or tasktracker to connect to its constituent master daemon without a valid ticket.

The high-level process for enabling security is as follows.

- Audit all services to ensure enabling security will not break anything.
  Hadoop security is all or nothing; enabling it will prevent all non-Kerberos authenticated communication. It is absolutely critical that you first take an inventory of all existing processes, both automated and otherwise, and decide how each will work once security is enabled. Don’t forget about administrative scripts and tools!
- Configure a working non-security enabled Hadoop cluster.
  Before embarking on enabling Hadoop’s security features, get a simple mode cluster up and running. You’ll want to iron out any kinks in DNS resolution, network connectivity, and simple misconfiguration early. Debugging network connectivity issues and supported encryption algorithms within the Kerberos KDC at the same time is not a position that you want to find yourself in.
- Configure a working Kerberos environment.
  Basic Kerberos operations such as authenticating and receiving a ticket-granting ticket from the KDC should work before you continue. You are strongly encouraged to use MIT Kerberos with Hadoop; it is, by far, the most widely tested. I
- Ensure host name resolution is sane.
  As discussed earlier, each Hadoop daemon has its own principal that it must know in order to authenticate. Since the hostname of the machine is part of the principal, all hostnames must be consistent and known at the time the principals are created.

- Create Hadoop Kerberos principals.
  Each daemon on each host of the cluster requires a distinct Kerberos principal when enabling security. Additionally, the Web user interfaces must also be given principals before they will function correctly. Just as the first point says, security is all or nothing.
- Export principal keys to keytabs and distribute them to the proper cluster nodes.
  With principals generated in the KDC, each key must be exported to a keytab, and copied to the proper host securely. Doing this by hand is incredibly laborious for even small clusters and, as a result, should be scripted.
- Update Hadoop configuration files.
  With all the principals generated and in their proper places, the Hadoop configuration files are then updated to enable security. The full list of configuration properties related to security are described later.
- Restart all services.
  To activate the configuration changes, all daemons must be restarted. The first time security is configured, it usually makes sense to start the first few daemons to make sure they authenticate correctly and are using the proper credentials before firing up the rest of the cluster.
- Test!

Local read short-circuiting

HDFS supports a feature called local read short-circuiting (as implemented by HDFS-2246) in which a client application running on the same machine as the datanode can completely bypass the datanode server and read block files directly from the local filesystem.

This can dramatically increase the speed of read operations, but at the cost of opening access to the underlying block data of *all blocks* to the client.

When this feature is enabled, clients must be running as the same user as the datanode *or* be in a group that has read access to block data. Both scenarios break some of the invariants assumed by the security model and can inadvertently expose data to malicious applications. Take great care when enabling this feature on a secure cluster or setting dfs.data node.data.dir.perm to anything other than 0700.

Hadoop configuration for kerberos

Reference: Hadoop Operations, Eric Sammers

Authorization

So far, we’ve discussed only how clients identify themselves and how Hadoop authenticates them. Once a client is authenticated, though, that client is still subject to authorization when it attempts to perform an action. The actions that can be performed vary from service to service. An action in the context of HDFS, for example, may be reading a file, creating a directory, or renaming a filesystem object. MapReduce actions, on the other hand, could be submitting or killing a job. Evaluating whether or not a user is permitted to perform a specific action is the process of *authorization*.

HDFS

Every filesystem operation in HDFS is subject to authorization. In an effort to exploit existing knowledge, HDFS uses the same authorization model as most POSIX filesystems. Each filesystem object (such as a file or directory) has three classes of user: an owner, a group, and “other,” which indicates anyone who isn’t in one of the two previous classes. The available permissions that can be granted to each of the three classes on an object are *read*, *write*, and *execute*, just as with Linux or other Unix-like systems. For example, it is possible to grant the owner of a file both read and write privileges. These permissions are represented by a single octal (base-8) integer that is calculated by summing permission values.

To represent the three classes of user—owner, group, and other—we use three integers, one for each class, in that order. For instance, a file that allows the owner to read and write, the group to read, and other users to read, would have the permissions 644; the 6 indicates read and write for the owner (4 + 2), whereas the subsequent fields are both 4, indicating only the read permission is available. To indicate the owner of a directory has read, write, and execute permissions, but no one else has access, the permissions 700 would be used.

In addition to the above permissions, the sticky bit is set. The sticky bit and when set on a directory, means that only the owner of a file in that directory may delete or rename the file, even if another user has access to do so (as granted by the write permission on the directory itself). The exception to this rule is that the HDFS super user and owner of the directory always have the ability to perform these actions.

MapReduce

Hadoop MapReduce, like HDFS, has a few different classes of users (four, to be exact).

Cluster owner

The cluster owner is the OS user that started the cluster. In other words, this is the user the jobtracker daemon is running as. This is normally user hadoop for Apache Hadoop and mapred for CDH. Like the HDFS superuser, the MapReduce cluster owner is granted all permissions implicitly and should be used rarely, if ever, by administrators.

Cluster administrator

One or more users may be specified as cluster administrators. These users have all of the same powers as the cluster owner, but do not need to be able to authenticate as the Linux user that started the cluster. Granting users this power allows them to perform administrative operations while still retaining the ability to audit their actions individually, rather than having them use a shared account. Use of a shared account is also discouraged as it creates the need to share authentication credentials.

Queue administrator

When a job is submitted to the jobtracker, the user specifies a *queue*. The queue has an access control list (ACL) associated with it that defines which users and groups may submit jobs, but also which users may administer the queue. Administrative actions may be changing the priority of a job, killing tasks, or even killing entire jobs.

Job owner

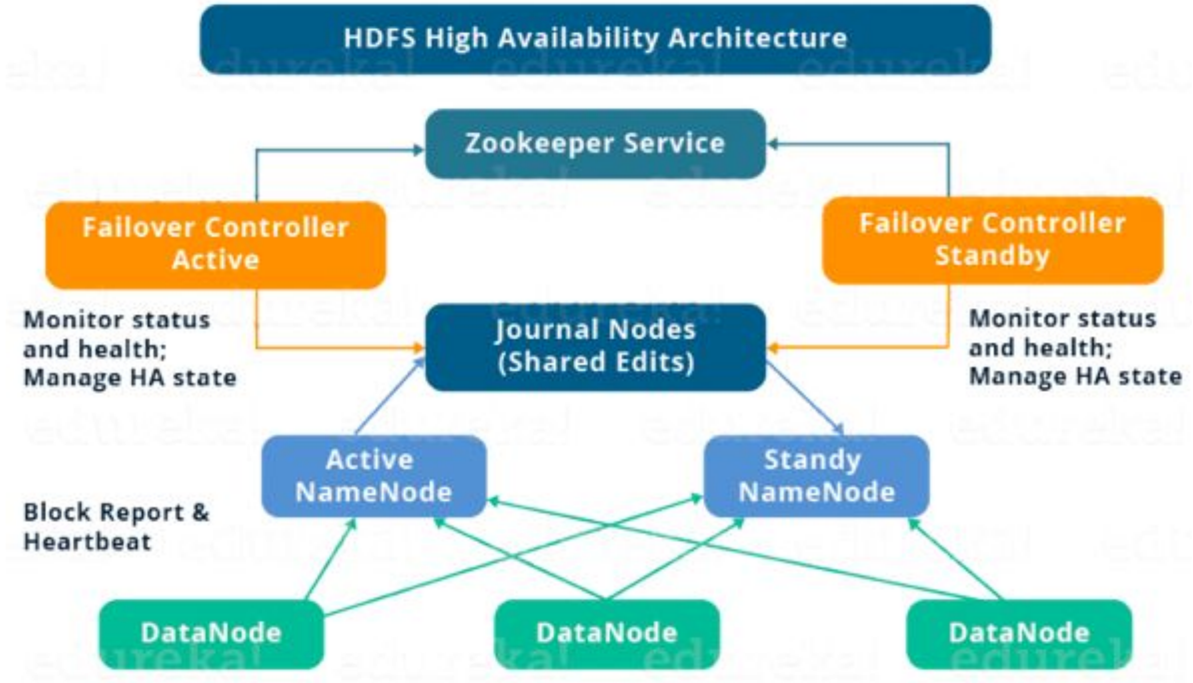
Finally, the job owner is the user who submitted a job. A job owner always has the ability to perform administrative actions on their own jobs.

LDAP

LDAP server must be up and running and port 389 shall be opened by firewall. This is configured in hadoop.security.group.mapping.ldap.url
Changes required to core-site.xml followed by a restart.

```
<property>
        <name>hadoop.security.group.mapping</name>
        <value>org.apache.hadoop.security.LdapGroupsMapping</value>
    </property>
    <property>
        <name>hadoop.security.group.mapping.ldap.bind.user</name>
        <value>cn=admin,dc=ldap,dc=ahuja,dc=com</value> <!-- this will be pertaining to your LDAP structure setup -->
    </property>
    <property>
        <name>hadoop.security.group.mapping.ldap.bind.password</name> <!--- LDAP admin password for LDAP server->
        <value>xxxxxxxx</value>
    </property>
    <property>
        <name>hadoop.security.group.mapping.ldap.url</name>
        <value>ldap://xxxxxxxxxxxx:389</value>
    </property>
    <property>
        <name>hadoop.security.group.mapping.ldap.base</name>            <!---- pertaining to your LDAP server setup -->
        <value>dc=ldap,dc=ahuja,dc=com</value>
```

Namenode high availability


HDFS High Availability Architecture
Zookeeper Service
Failover Controller Active
Failover Controller Standby
Monitor status and health; Manage HA state
Journal Nodes (Shared Edits)
Monitor status and health; Manage HA state
Active NameNode
Standy NameNode
Block Report & Heartbeat
DataNode
DataNode
DataNode

NameNode Availability:

If you consider the standard configuration of HDFS cluster, the NameNode becomes a single point of failure. It happens because the moment the NameNode becomes unavailable, the whole cluster becomes unavailable until someone restarts the NameNode or brings a new one.

The reasons for unavailability of NameNode can be:

A planned event like maintenance work such has upgradation of software or hardware.

It may also be due to an unplanned event where the NameNode crashes because of some reasons.

In either of the above cases, we have a downtime where we are not able to use the HDFS cluster which becomes a challenge.

Solution is to have two running NameNodes at the same time in a High Availability cluster:

Active NameNode

Standby/Passive NameNode

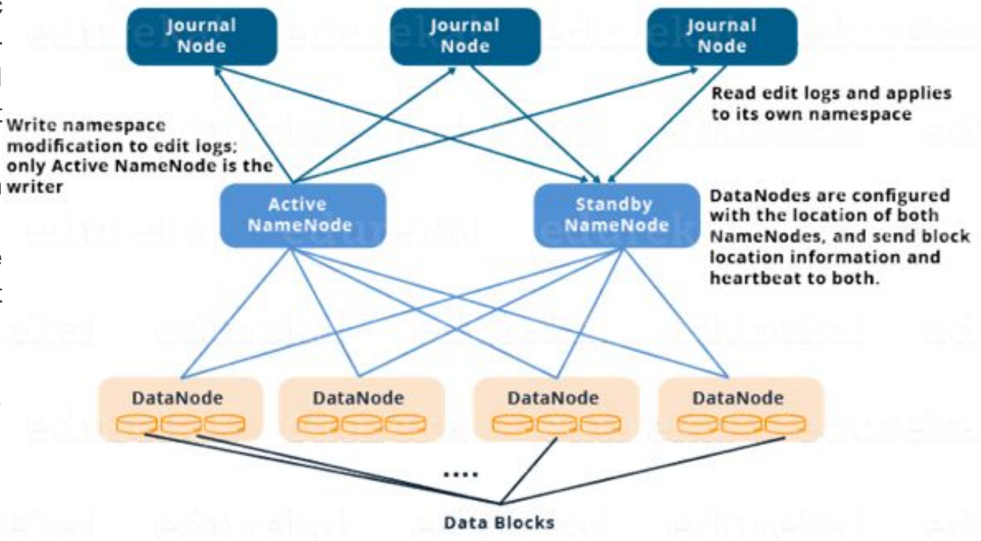
There are two issues in maintaining consistency in the HDFS High Availability cluster:

Active and Standby NameNode should always be in sync with each other, i.e. They should have the same metadata. This will allow us to restore the Hadoop cluster to the same namespace state where it got crashed and therefore, will provide us to have fast failover.

There should be only one active NameNode at a time because two active NameNode will lead to corruption of the data. This kind of scenario is termed as a split-brain scenario where a cluster gets divided into smaller cluster, each one believing that it is the only active cluster. To avoid such scenarios fencing is done. Fencing is a process of ensuring that only one NameNode remains active at a particular time.

HA using Quorum journal nodes

- The standby NameNode and the active NameNode keep in sync with each other through a separate group of nodes or daemons - JournalNodes. The JournalNode serves the request coming to it and copies the information into other Journalnodes.This provides fault tolerance in case of JournalNode failure
- Active NameNode is responsible for updating the EditLogs present in the JournalNodes.
- The StandbyNode reads the changes made to the EditLogs in the JournalNode and applies it to its own namespace in a constant manner.
- During failover, the StandbyNode makes updates its metadata information from JournalNodes, then become the new Active NameNode
- The IP Addresses of both the NameNodes are available to all the DataNodes.


Journal Node
Journal Node
Journal Node
Read edit logs and applies to its own namespace
Write namespace modification to edit logs: only Active NameNode is the writer
Active NameNode
Standby NameNode
DataNodes are configured with the location of both NameNodes, and send block location information and heartbeat to both.
DataNode
DataNode
DataNode
DataNode
....
Data Blocks

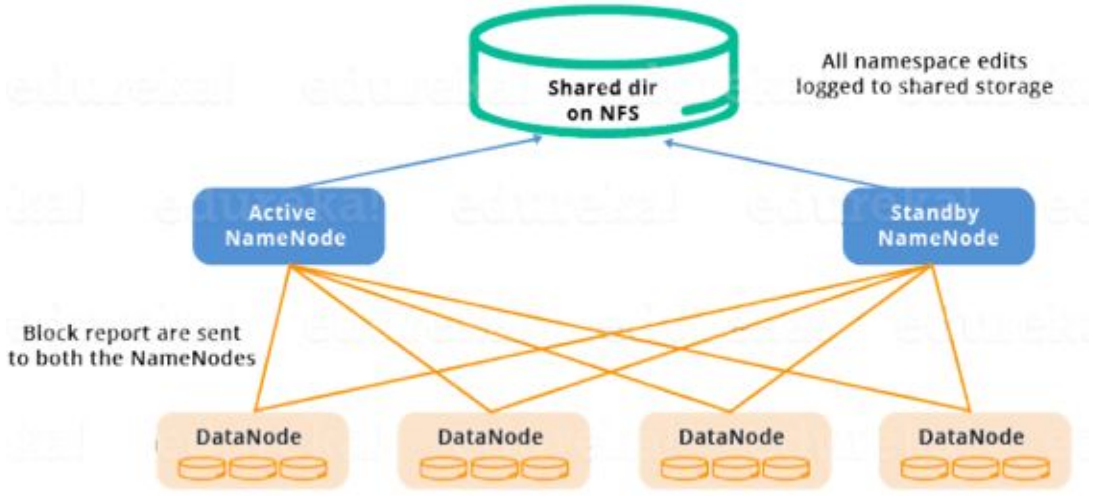
Fencing of NameNode:

Now, as discussed earlier, it is very important to ensure that there is only one Active NameNode at a time. So, fencing is a process to ensure this very property in a cluster.

- The JournalNodes performs this fencing by allowing only one NameNode to be the writer at a time.
- The Standby NameNode takes over the responsibility of writing to the JournalNodes and forbid any other NameNode to remain active.
- Finally, the new Active NameNode can perform its activities safely.

Using shared storage

- The StandbyNode and the active NameNode keep in sync with each other by using a shared storage device. The active NameNode logs the record of any modification done in its namespace to an EditLog present in this shared storage. The StandbyNode reads the changes made to the EditLogs in this shared storage and applies it to its own namespace.
- Now, in case of failover, the StandbyNode updates its metadata information using the EditLogs in the shared storage at first. Then, it takes the responsibility of the Active NameNode. This makes the current namespace state synchronized with the state before failover.
- The administrator must configure at least one fencing method to avoid a split-brain scenario.
- The system may employ a range of fencing mechanisms. It may include killing of the NameNode's process and revoking its access to the shared storage directory.
- As a last resort, we can fence the previously active NameNode with a technique known as STONITH, or "shoot the other node in the head". STONITH uses a specialized power distribution unit to forcibly power down the NameNode machine.


Shared dir on NFS
All namespace edits logged to shared storage
Active NameNode
Standby NameNode
Block report are sent to both the NameNodes
DataNode
DataNode
DataNode
DataNode

Automatic failover

Failover is a procedure by which a system automatically transfers control to secondary system when it detects a fault or failure. There are two types of failover:

- Graceful Failover: In this case, we manually initiate the failover for routine maintenance.
- Automatic Failover: In this case, the failover is initiated automatically in case of NameNode failure (unplanned event).
- Apache Zookeeper is a service that provides the automatic failover capability in HDFS High Availability cluster. It maintains small amounts of coordination data, informs clients of changes in that data, and monitors clients for failures. Zookeeper maintains a session with the NameNodes. In case of failure, the session will expire and the Zookeeper will inform other NameNodes to initiate the failover process. In case of NameNode failure, other passive NameNode can take a lock in Zookeeper stating that it wants to become the next Active NameNode.
- The ZookeerFailoverController (ZKFC) is a Zookeeper client that also monitors and manages the NameNode status. Each of the NameNode runs a ZKFC also. ZKFC is responsible for monitoring the health of the NameNodes periodically
- Now that you have understood what is High Availability in a Hadoop cluster, it's time to set it up. To set up High Availability in Hadoop cluster you have to use Zookeeper in all the nodes

  The daemons in DataNode are:

  - Zookeeper service (QuorumPeerMain)
  - JournalNode (Journalnode)
  - DataNode
  - NodeManager

  The daemons in Standby NameNode are:

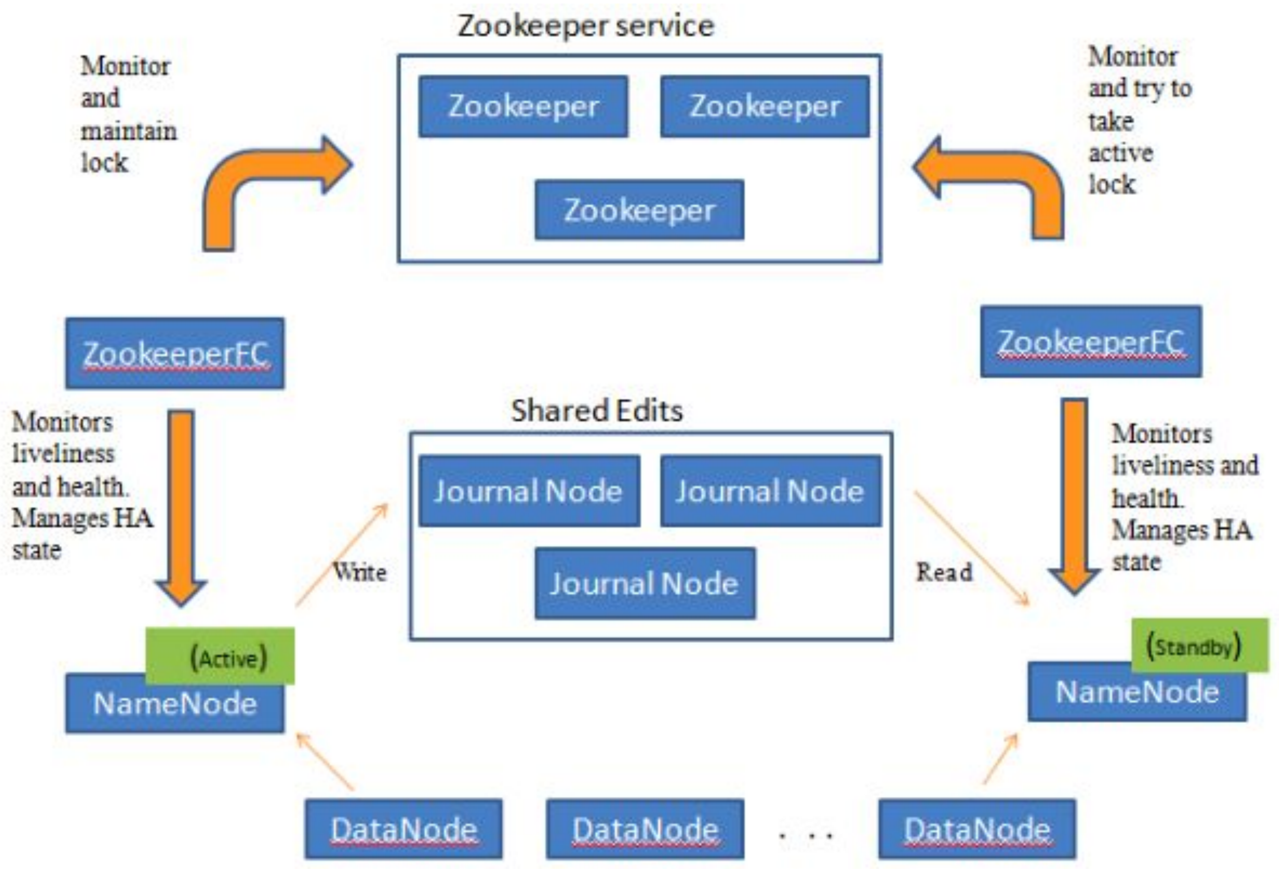
  - Zookeeper service (QuorumPeerMain)
  - Zookeeper failover controller (DFSZkFailoverController)
  - JournalNode (Journalnode)
  - NameNode

  The daemons in Active NameNode are:

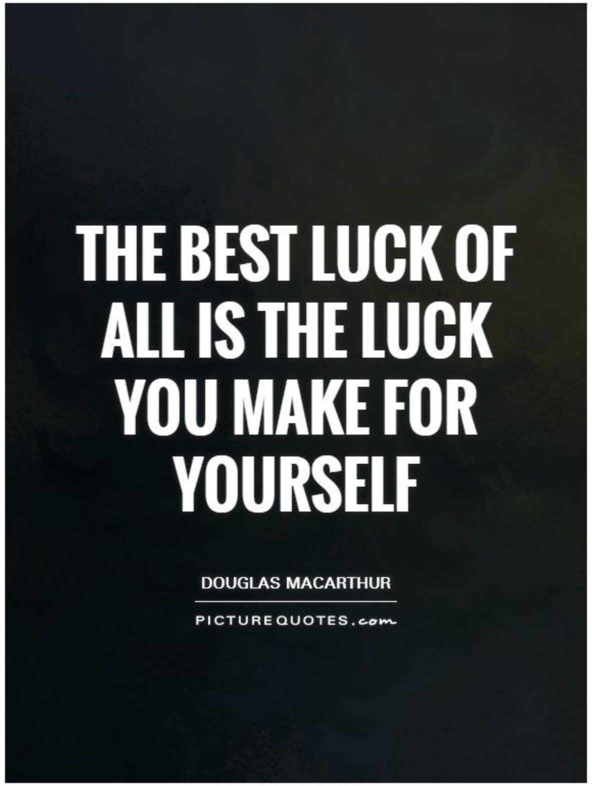
  - Zookeeper service (QuorumPeerMain)
  - Zookeeper failover controller (DFSZkFailoverController)
  - JournalNode (Journalnode)
  - NameNode
  - ResourceManager

Assumption: DataNode is planned to run on a separate node

Automatic failover architecture



Thank you & best wishes

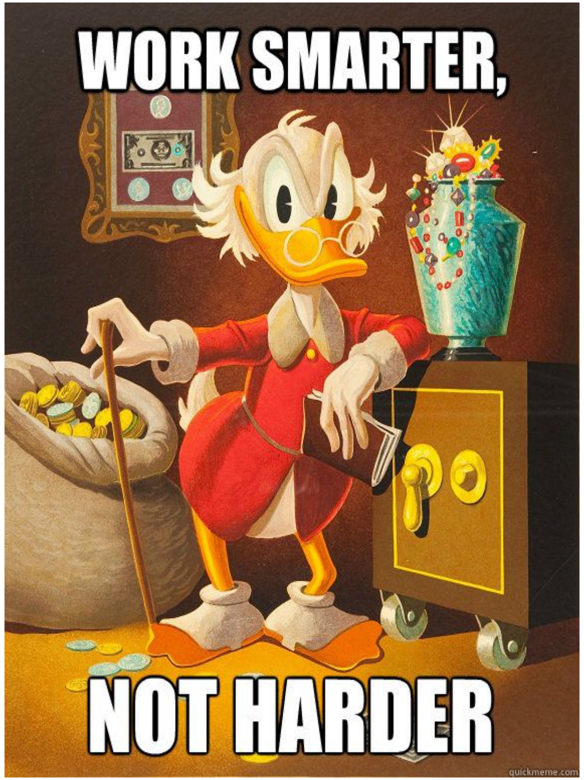


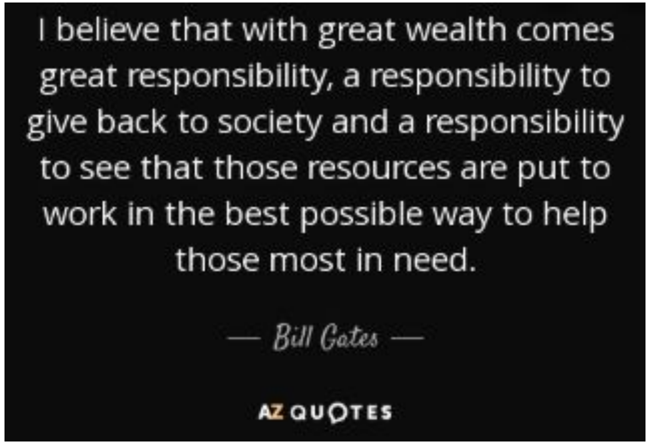
THE BEST LUCK OF ALL IS THE LUCK YOU MAKE FOR YOURSELF

DOUGLAS MACARTHUR

PICTUREQUOTES.com

PICTUREQUOTES





I believe that with great wealth comes great responsibility, a responsibility to give back to society and a responsibility to see that those resources are put to work in the best possible way to help those most in need.

— Bill Gates —

AZ QUOTES

Aakash Ahuja

AakashxAhuja