Hadoop

Cloudlytics

Hadoop Distributions

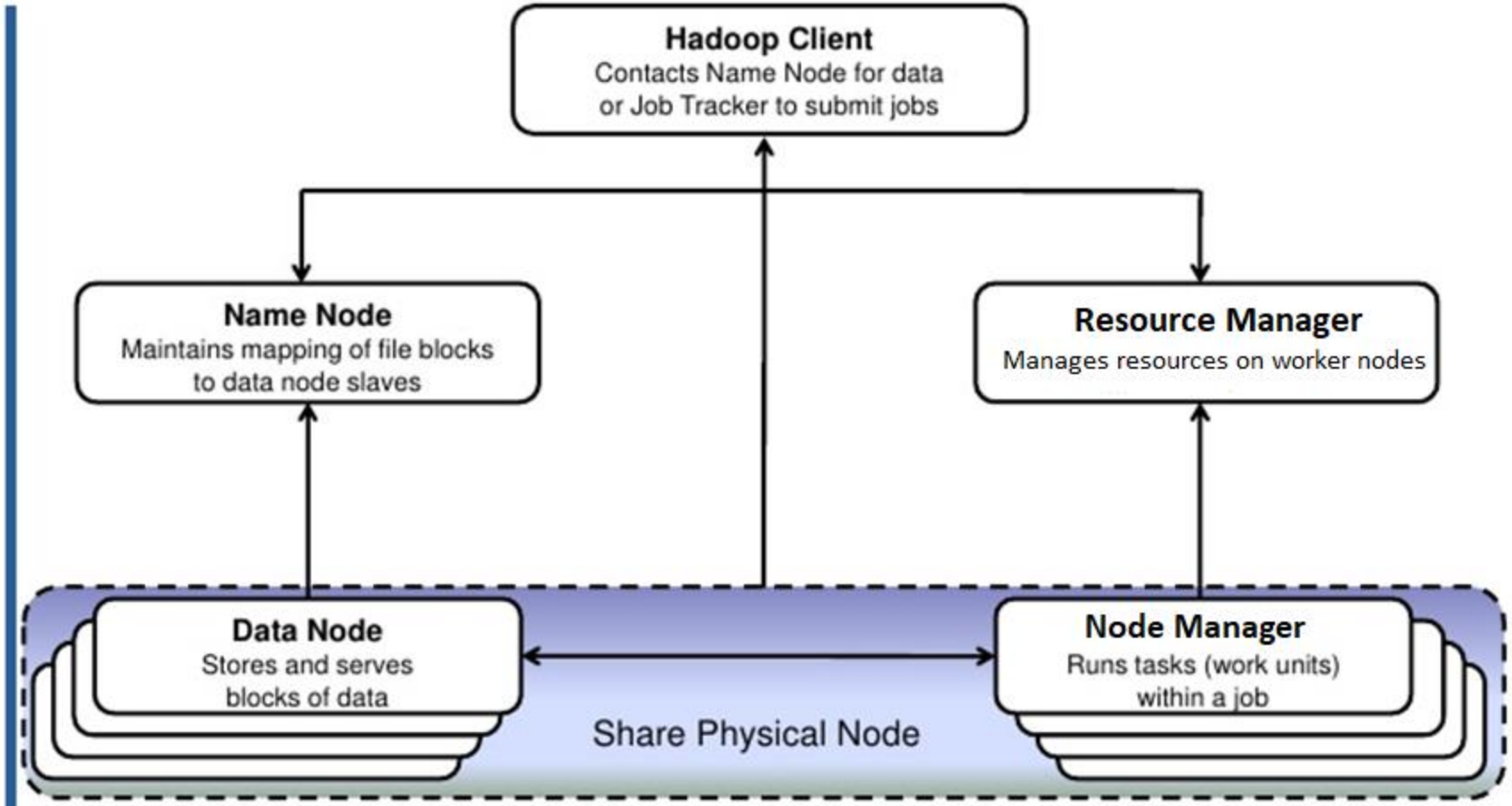
- Cloudera – CDH
- Hortonworks – HDP
- IBM – BigInsights
- Mapr – Mapr
- EMC – Greenplum
- Hp – Vertica, Haven
- Teradata – Aster
- Oracle – Exalytics
- Microsoft - HDInsights

- The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models.
- Hadoop runs on commodity hardware
- Completely written in java
- Robust, Self-healing and Resilient
- Hadoop core components : HDFS and Map Reduce

About Hadoop

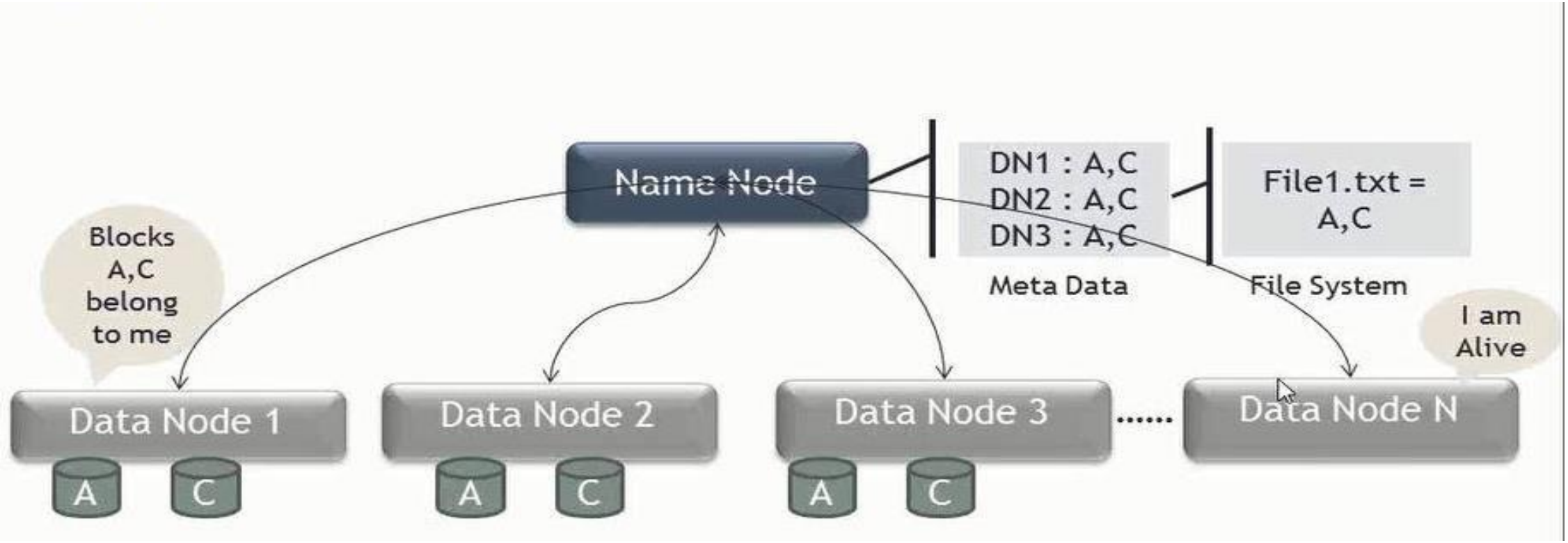
- Released in 2008 under Apache
- A well publicised feat, the New York times used Hadoop on EC2 to crunch 4 TB of scanned archives from paper, converting the to PDF for web. This processing took less than 24 hours on 100 machines
- In April 2008, Hadoop broke a world record to become fastest system to sort a terabyte of data. Running on a 910 node cluster, Hadoop sorted 1TB in 209 seconds beating previous years winner of 297 sec.
- Later in November of the same year, Google reported that itd Map-reduce implementation sorted 1TBin 68 seconds.
- In April 2009, a team at Yahoo! Used Hadoop to sort 1TB in 62 seconds

Hadoop High-Level Architecture



Cloudlytics

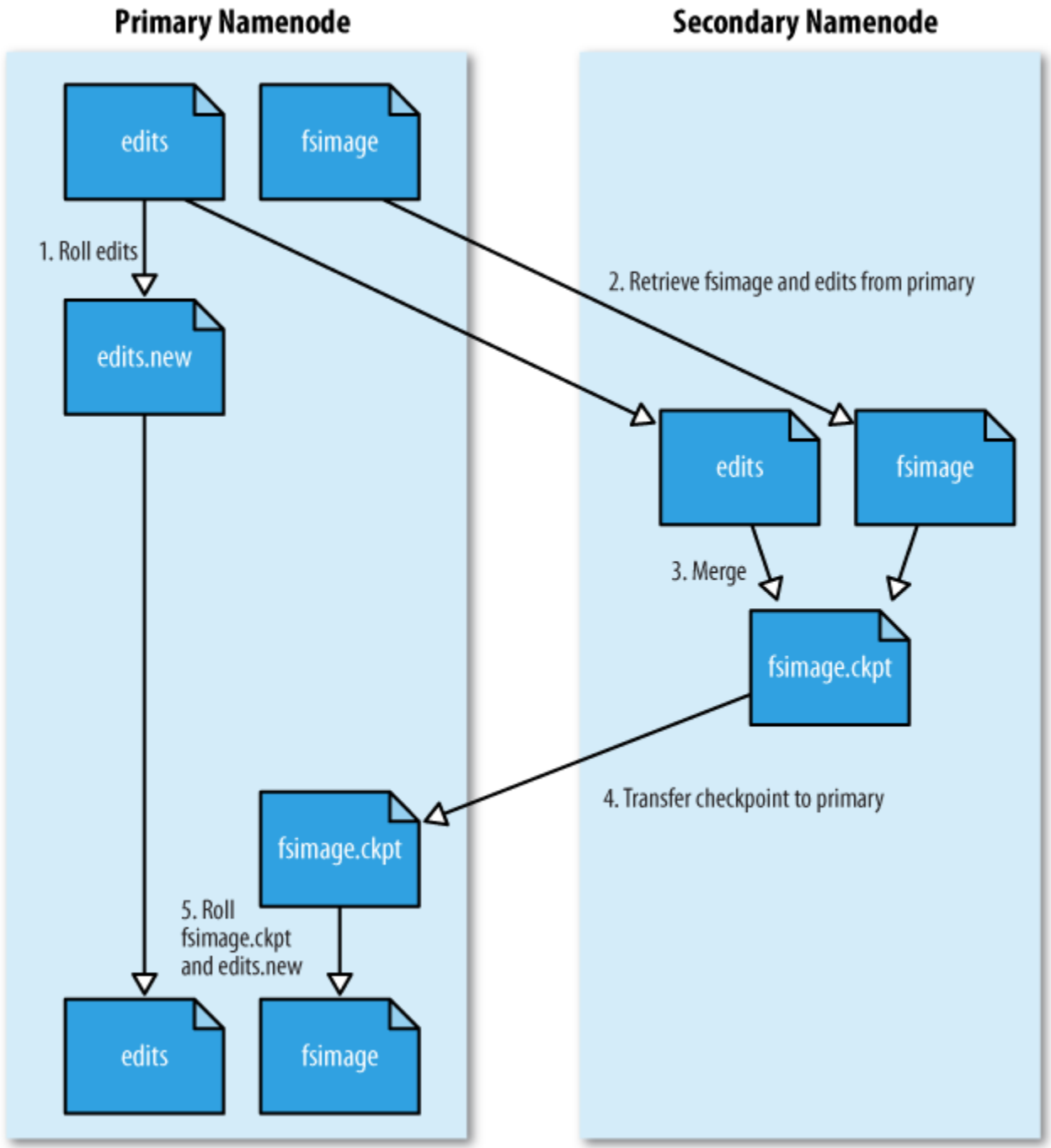
HDFS Architecture



Namenode

- The master node
- Manages the filesystem namespace
- Maintains filesystem tree and metadata for all the files and dir in the tree
- *fsimage and edits*
- *fsimage* is the complete persistent checkpoint of the filesystem metadata
- 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, because this information is reconstructed from datanodes when the system starts.
- $ hdfs fsck / -files –blocks :- command to see block for each file

Checkpointing using SNN

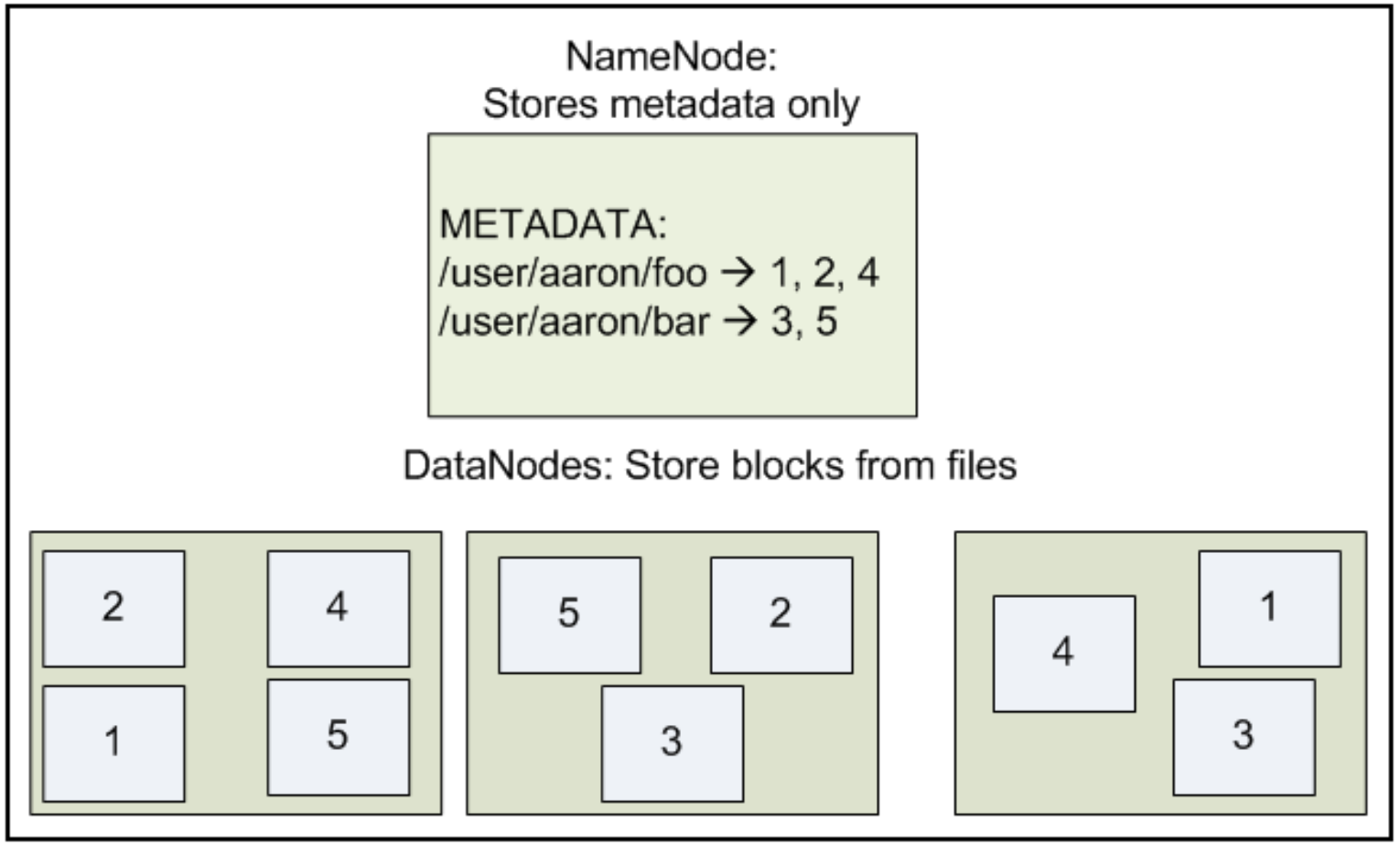


Checkpointing through SNN

- The schedule for checkpointing is controlled by two configuration parameters.

- The secondary namenode checkpoints every hour (dfs.namenode.checkpoint.period in seconds), or sooner if the edit log has reached one million transactions since the last checkpoint (dfs.namenode.checkpoint.txns), which it checks every minute (dfs.namenode.checkpoint.check.period in seconds).

Datanode

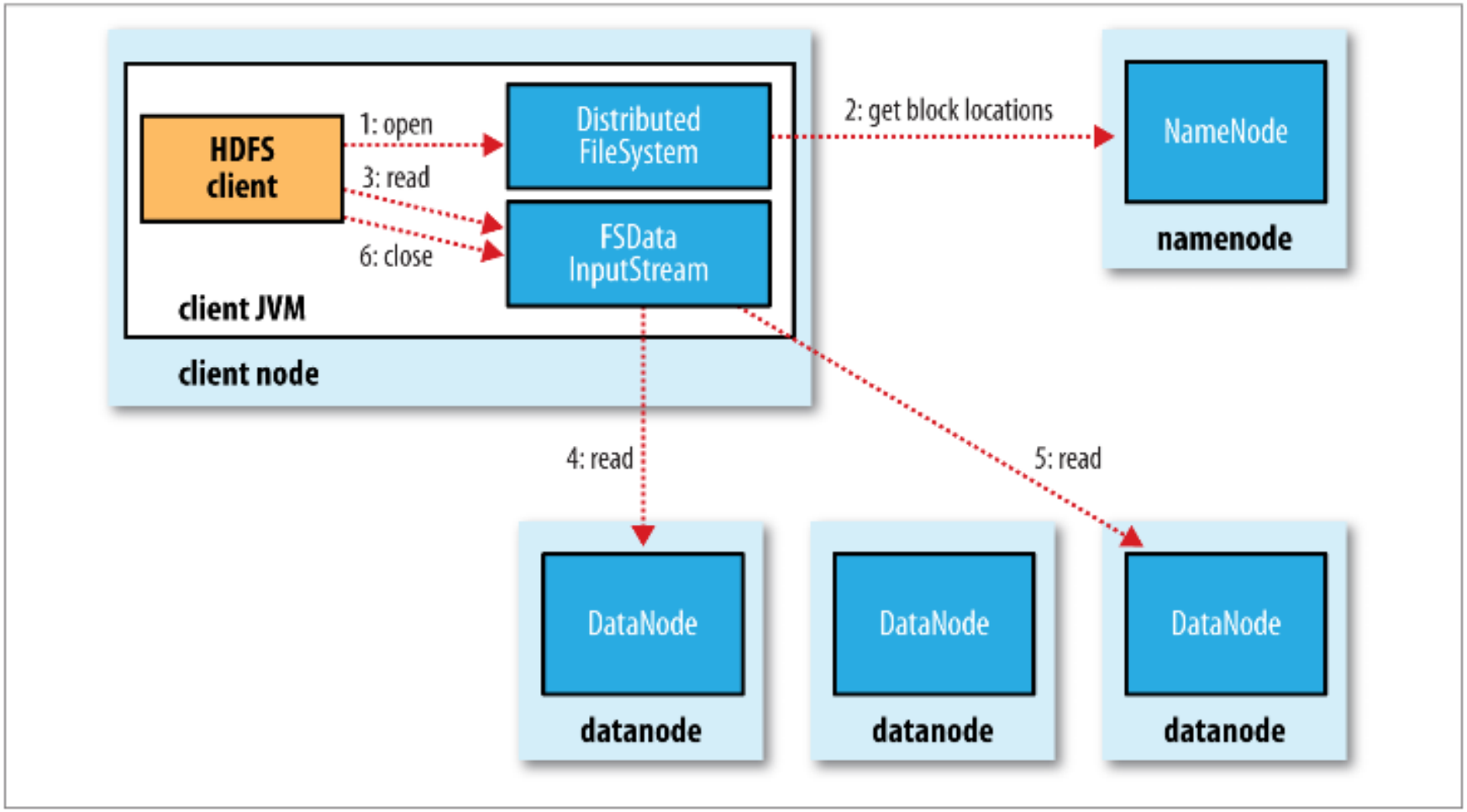


Cloudlytics

Understanding file read and write

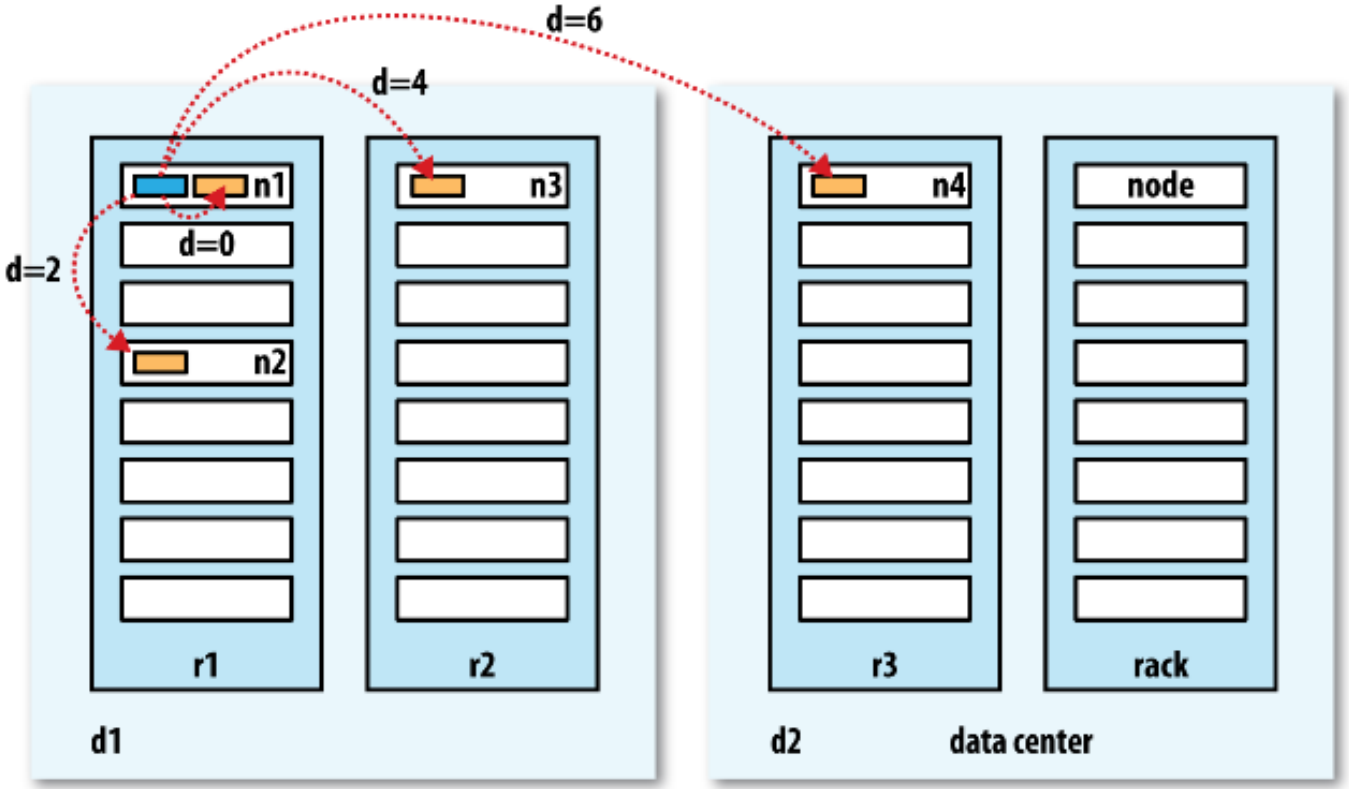
- It creates a pipeline to **write** data
- Understanding Rack awareness
- It r**eads** the data parallelly

Anatomy of file read

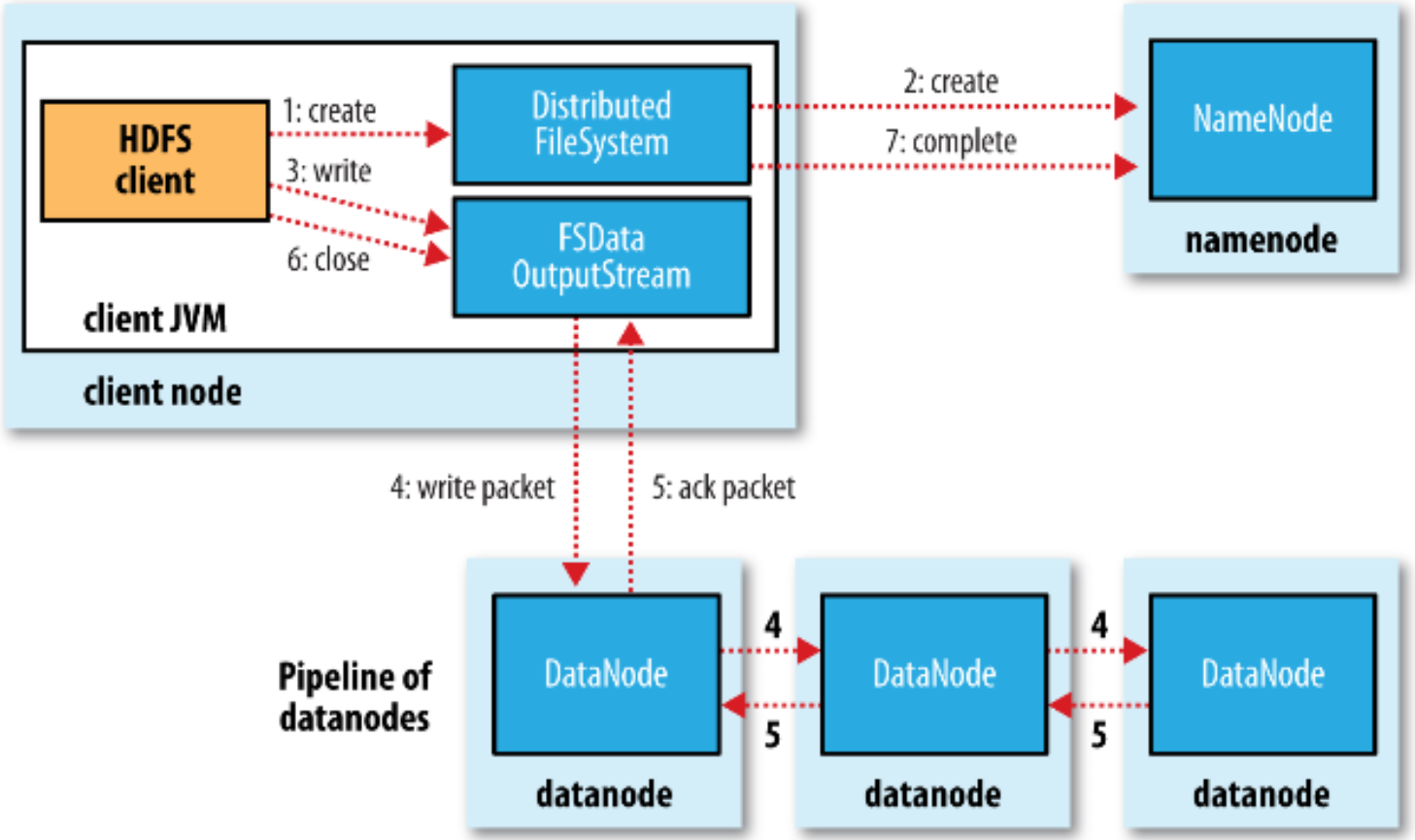


Rack Awareness

- We can create a topology.sh script for rack awareness
- The script's location is controlled by the property net.topology.script.file.name.
- The above property has to be added in core-site.xml



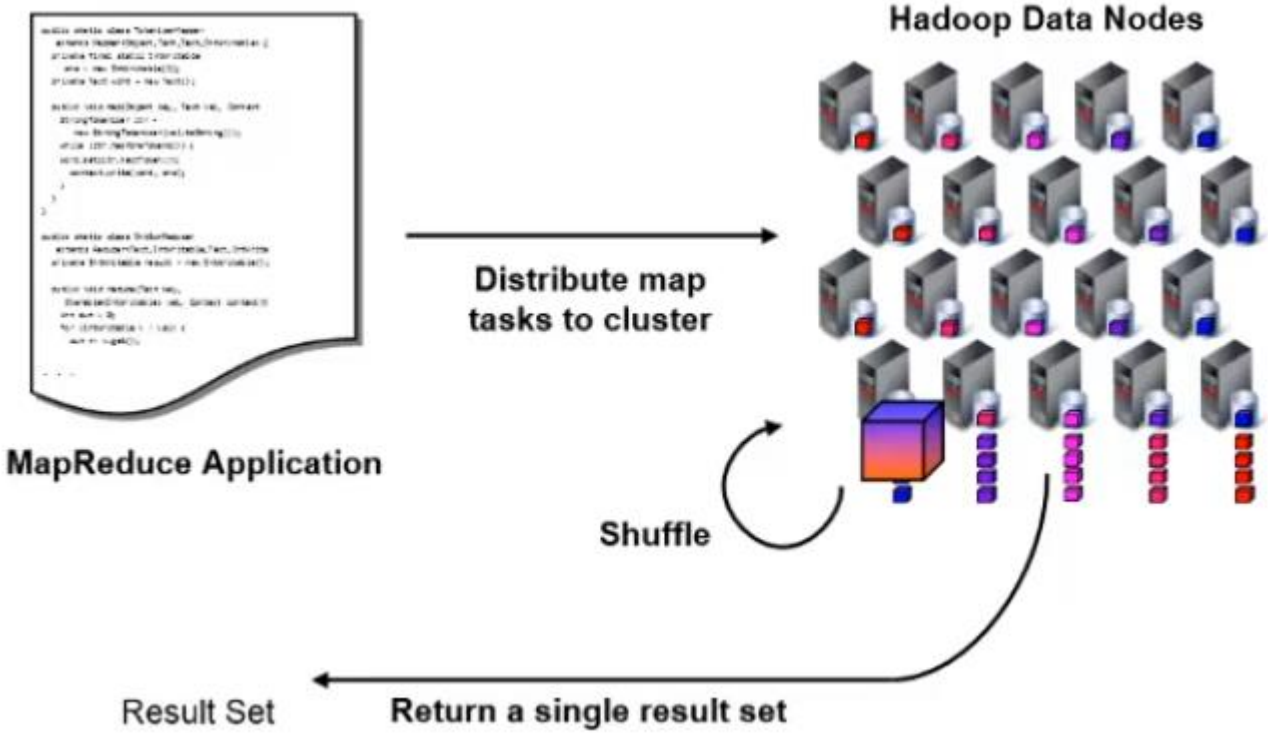
Anatomy of a file write



Map Reduce

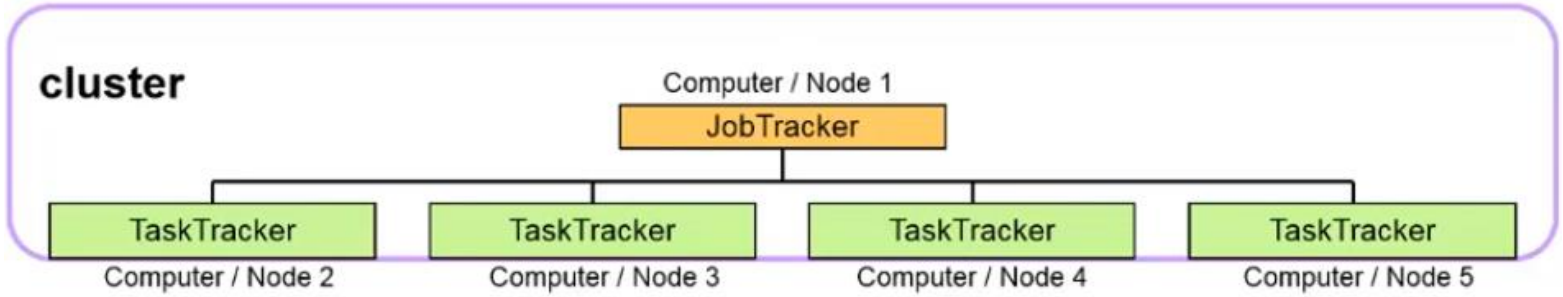
- Introduced originally by Google
- Data is stored across the cluster
- Data localisation
  - Programs are brought to the data,

    not the data to the program



MapReduce V1

- Master Slave architecture
  - Single master (JobTracker) controls job execution on multiple slaves (Tasktracker)
- Job Tracker
  - Accepts map reduce jobs submitted by clients
  - Pushes map and reduces tasks to TT
  - Keeps the work as physically close to data as possible
  - Monitor tasks and TT status
- Task Tracker
  - Runs map and reduce
  - Reports status to JT
  - Manages storage and transmission of intermediate output
  - Sends heartbeat to JT



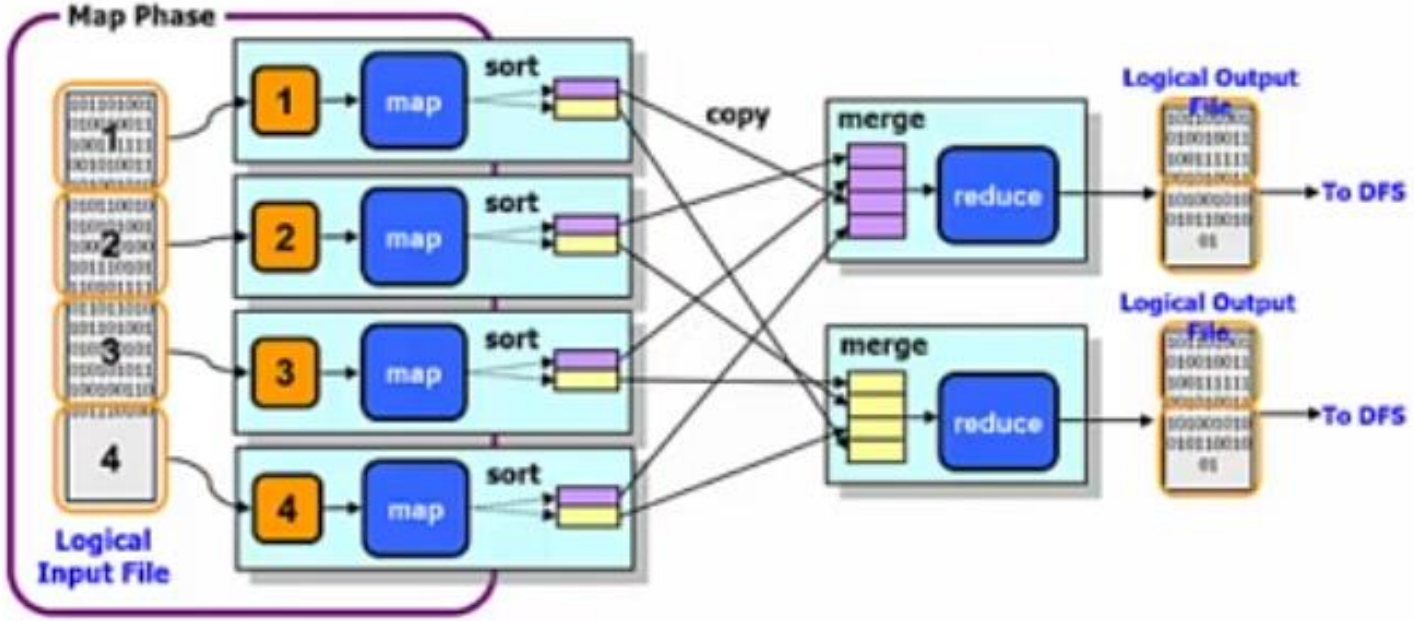
Cloudlytics

Map Reduce phases

- Map
- Shuffle
- Reduce
- Combiner

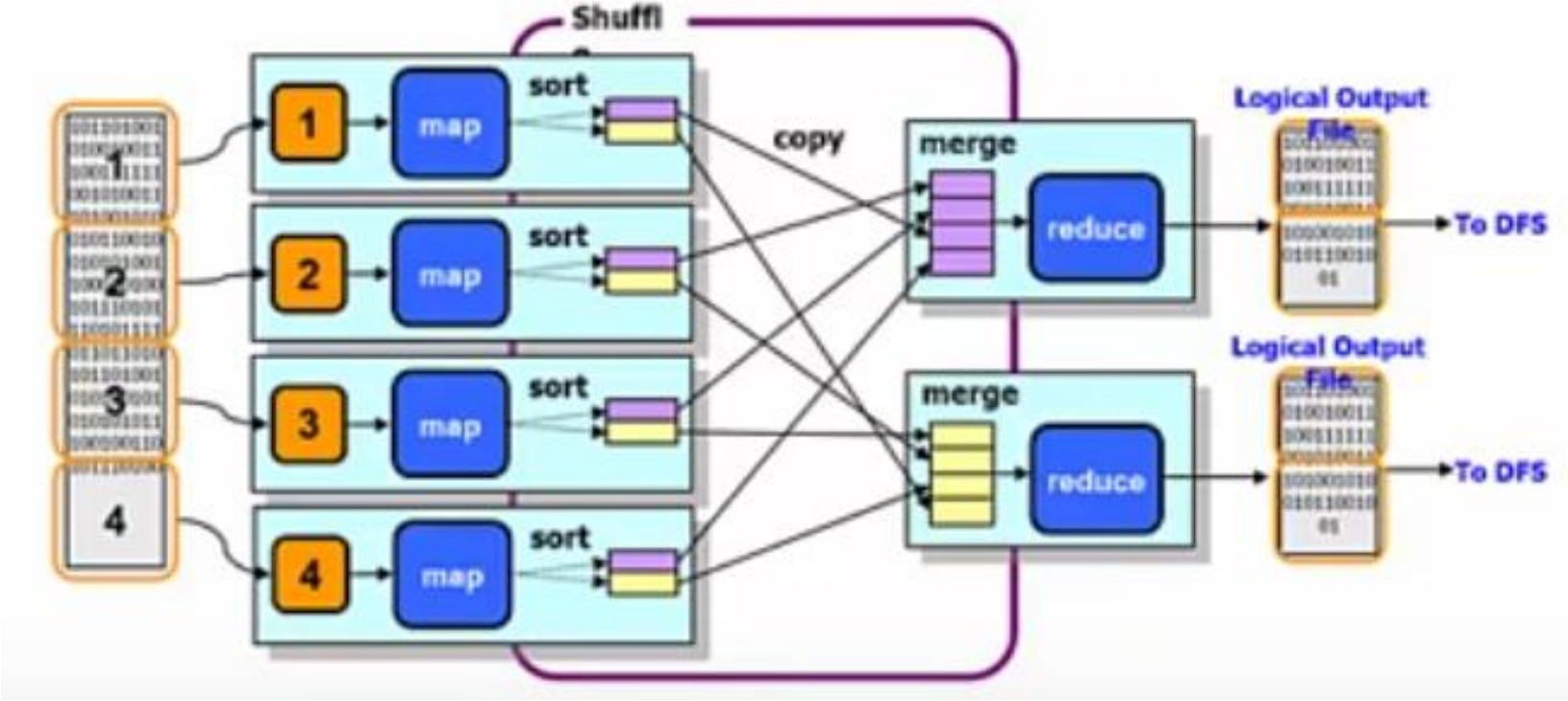
Map phase

- Mappers
  - Small programs, distributed across the cluster, local to data
  - Handed a portion of input data – called a split
  - Each mapper parses, filters or transforms its input
  - Produces <key,value> pairs



Shuffle phase

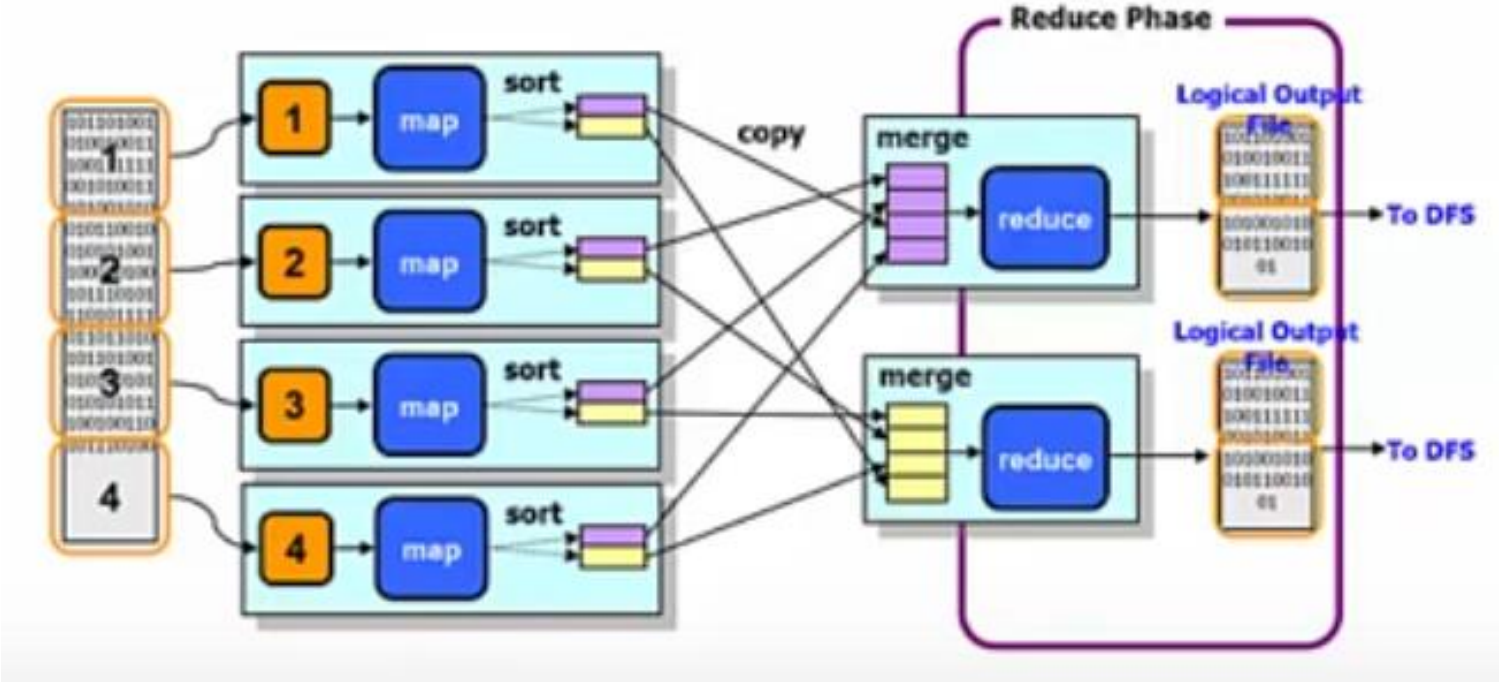
- The output of each mapper is locally grouped together by a key



Cloudlytics

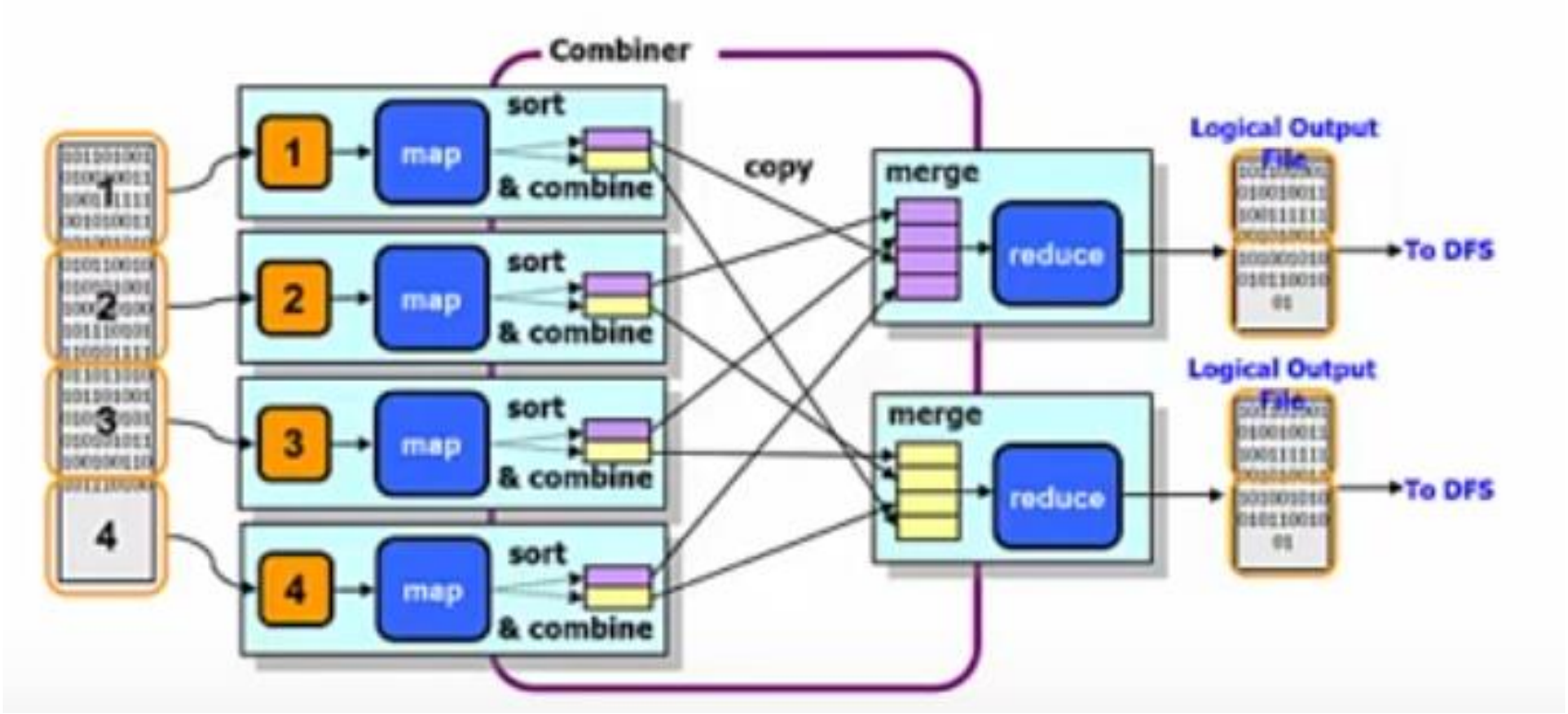
Reduce phase

- Small programs that aggregate all of the *values* for the *keys*
- Each reducer writes the output to its own file

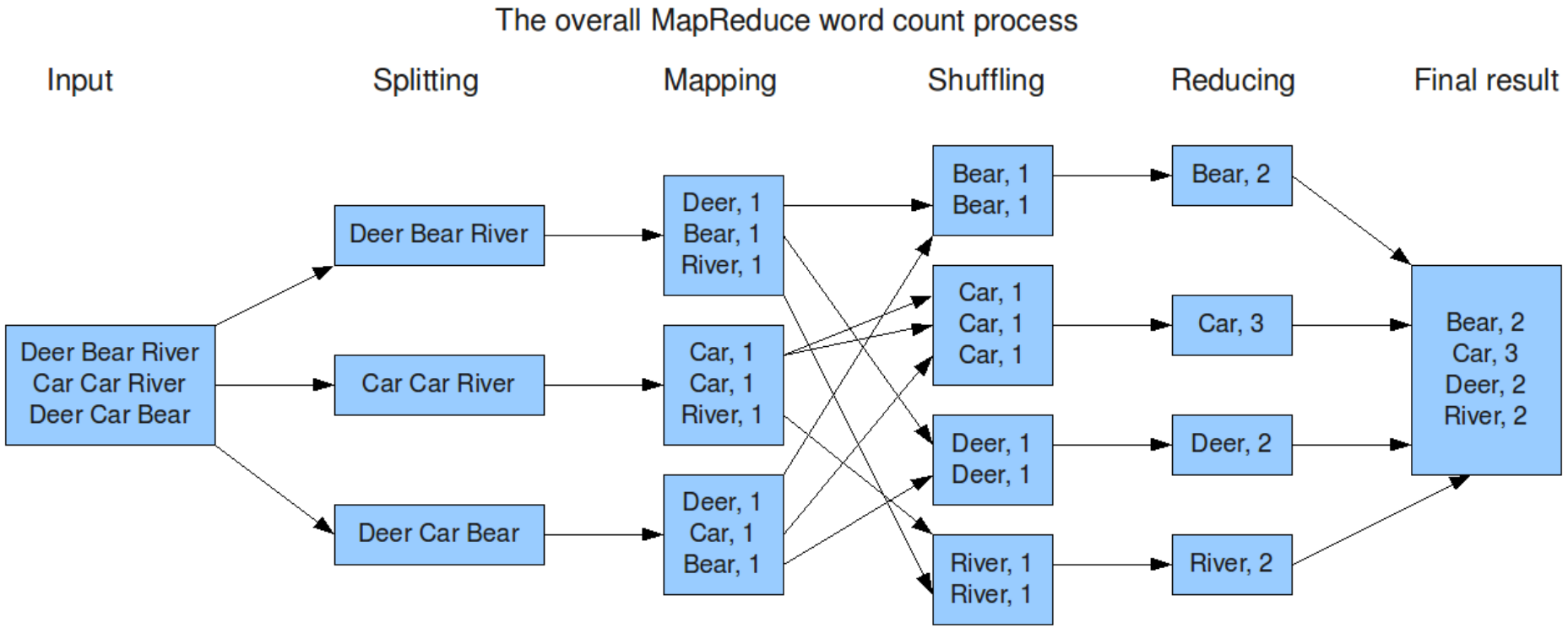


Combiner (optional)

- The data that will go to each reduce node is sorted and merged before going to the reduce mode



WordCount example



Cloudlytics

Classes

- Three main java classes provided in Hadoop to read data in map reduce
- **InputSplitter** – divides the files into splits
- **RecordReader** – takes a split and reads the files into records
- **InputFormat** – takes each record and transforms it into a <*key,value*> pair that is then passed to map task
- Other classes are also there

When it all goes wrong

- MapReduce is designed to treat failures as common and has very well defined semantics for dealing the inevitable
- **Task failures**
  - Very common, reasons – incorrect or poorly implemented user code, unexpected data problems, temp machine failures, etc
  - Errors – It throws an uncaught exception
  - It exits with a non zero exit code
  - It fails to report progress to tasktracker for configurable amount of time
  - If a task fails, it is reported to JT in next heartbeat. JT in turn notes it and it performs additional attempts (default 4) to start failed tasks.
  - If the same task of a job fails continuously the tasktracker is added to job level blacklist. If tasks of different jobs fail on a specific tasktracker it is added to global blacklist for 24 hours

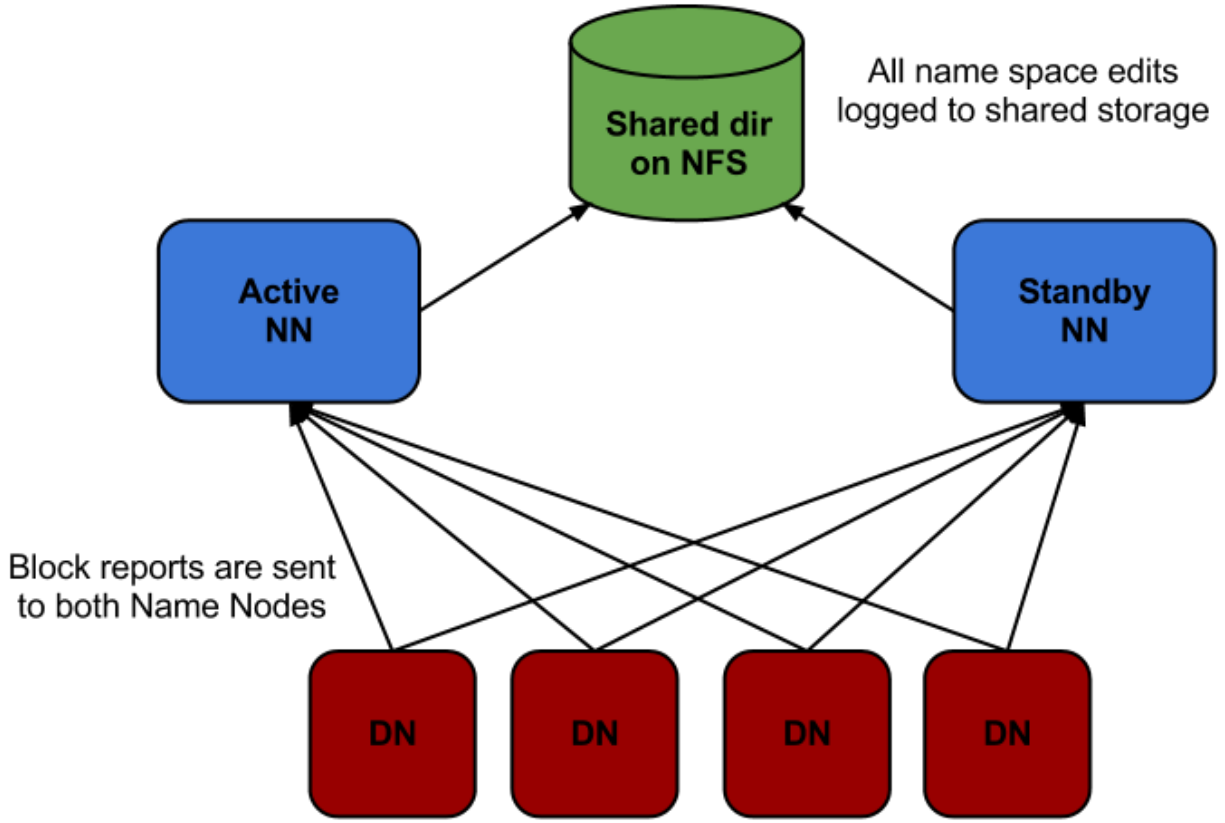
When it all goes wrong

- **TaskTracker failure**
  - If JT does not receive HB for a configurable amount of time, it treats TT as dead.
  - The tasks are then rescheduled on other TT's.
  - The client is completely shielded from internal failure. It will look to him as if job has slowed down a bit
- **JobTracker failure**
  - Very severe condition
  - The internal state of executing jobs is lost
  - Even if it immediately recovers , all running tasks eventually fail

SPOF vs HA

- SPOF – Single point of failure
- HA
  - HA using NFS
  - HA using Quorum Based Journaling

HA using NFS

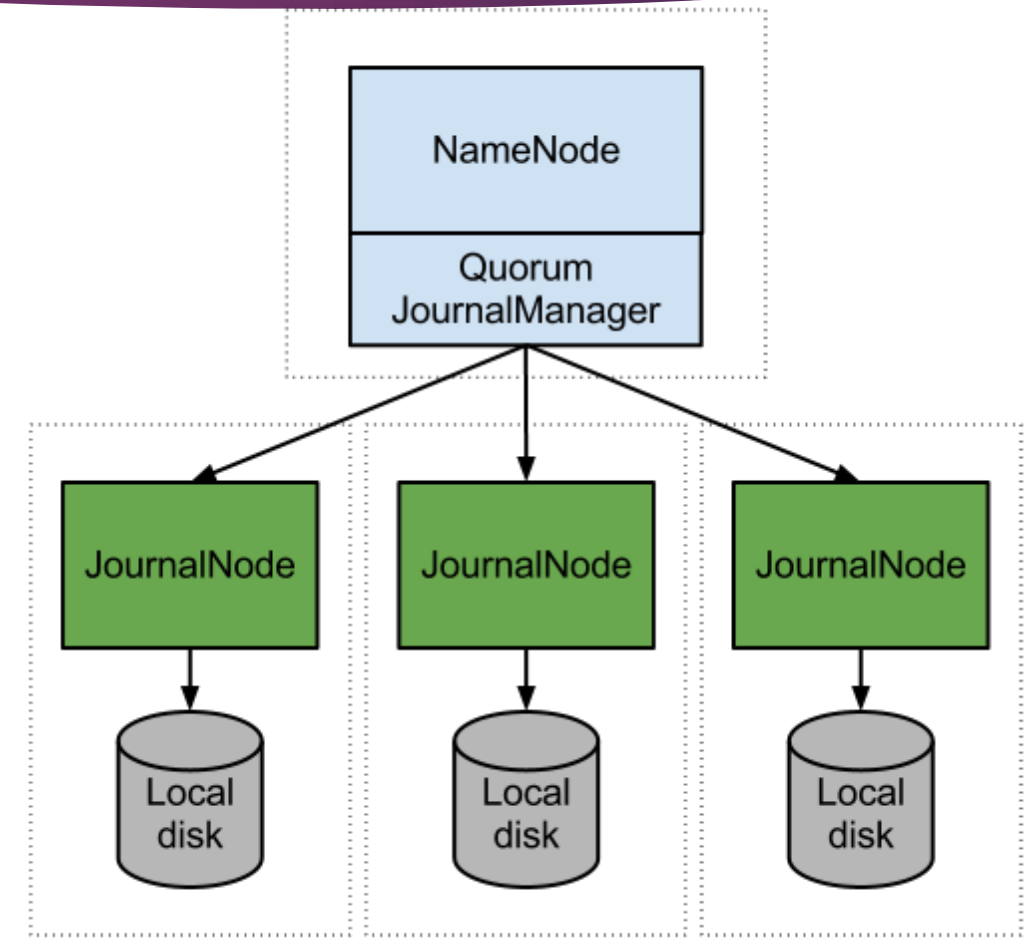

Shared dir
on NFS
All name space edits
logged to shared storage
Active
NN
Standby
NN
Block reports are sent
to both Name Nodes
DN
DN
DN
DN

Limitations of shared NFS

- The shared storage must itself be **highly available** — if it becomes inaccessible, the Active NameNode will no longer be able to continue taking namespace edits.
- **Custom hardware** – the hardware requirements of a NAS device can be expensive. Additionally, fencing configurations may require a remotely controllable Power Distribution Unit (PDU) or other specialized hardware.
- **Complex deployment** – even after HDFS is installed, the administrator must take extra steps to configure NFS mounts, custom fencing scripts, etc. This complicates HA deployment and may even cause unavailability if misconfigured.
- **Poor NFS client implementations** – Many versions of Linux include NFS client implementations that are buggy and difficult to configure.
- **External dependencies** – depending on a NAS device for storage requires that operators monitor and maintain one more piece of infrastructure. At the minimum, this involves configuring extra alerts and metrics, and in some organizations may also introduce an inter-team dependency

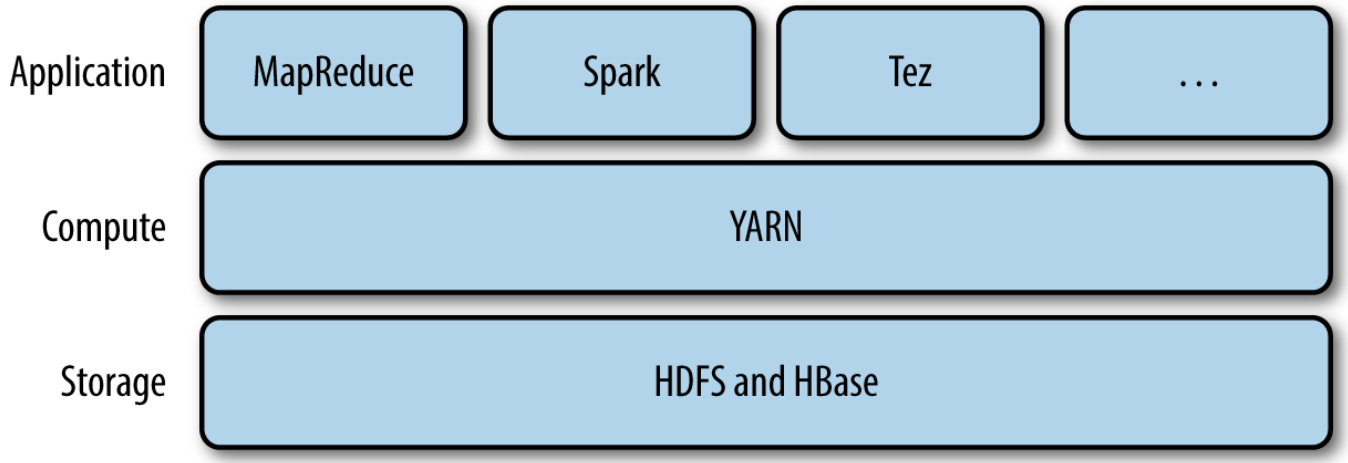
HA using QJM

Note : Quorum-based storage is the only implementation Cloudera supports in CDH 5



YARN

- YARN (Yet another resource negotiator) – Hadoop cluster resource management system

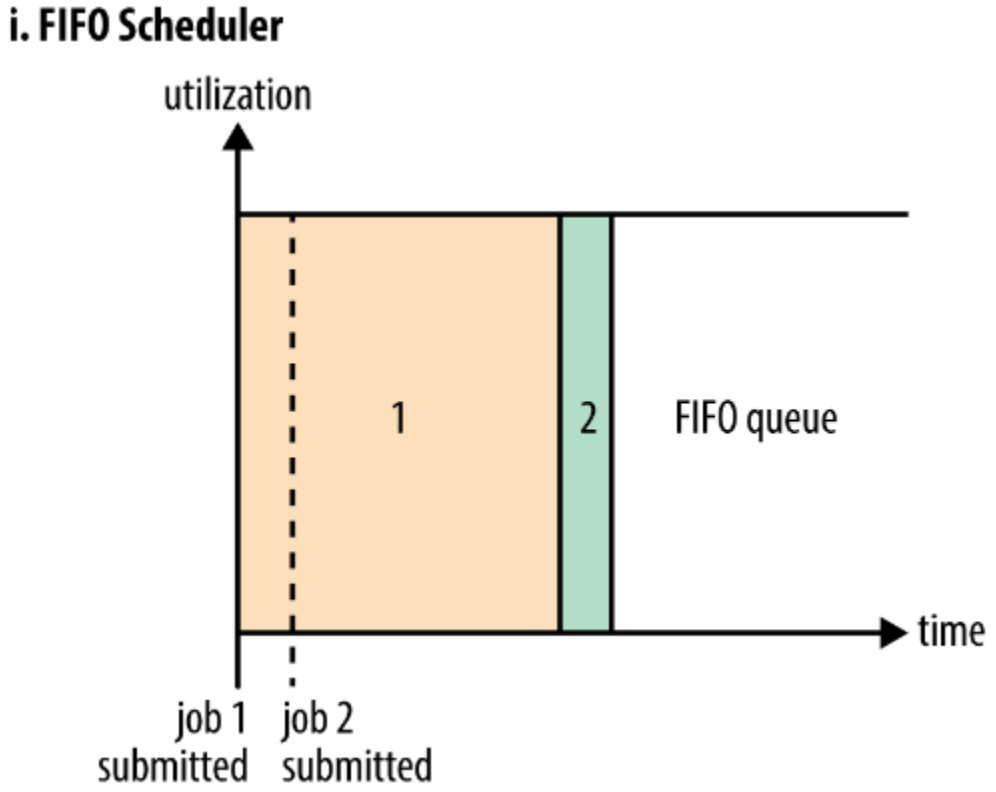


Yarn vs MR1

- In MR1, JobTracker takes care of both **scheduling** and **task progress monitoring;** In contrast, in Yarn these responsibilities are handled by two separate entities : Resource manager and Application Master
- Benefits of Yarn
  - Scalability : MR1 hits scalability bottleneck in the region of 4000 nodes and 40000 tasks. Yarn is designed to scale from 10000 and 100000 tasks
  - Availability : HA can be enabled for RM
  - Utilization
  - Multitenancy

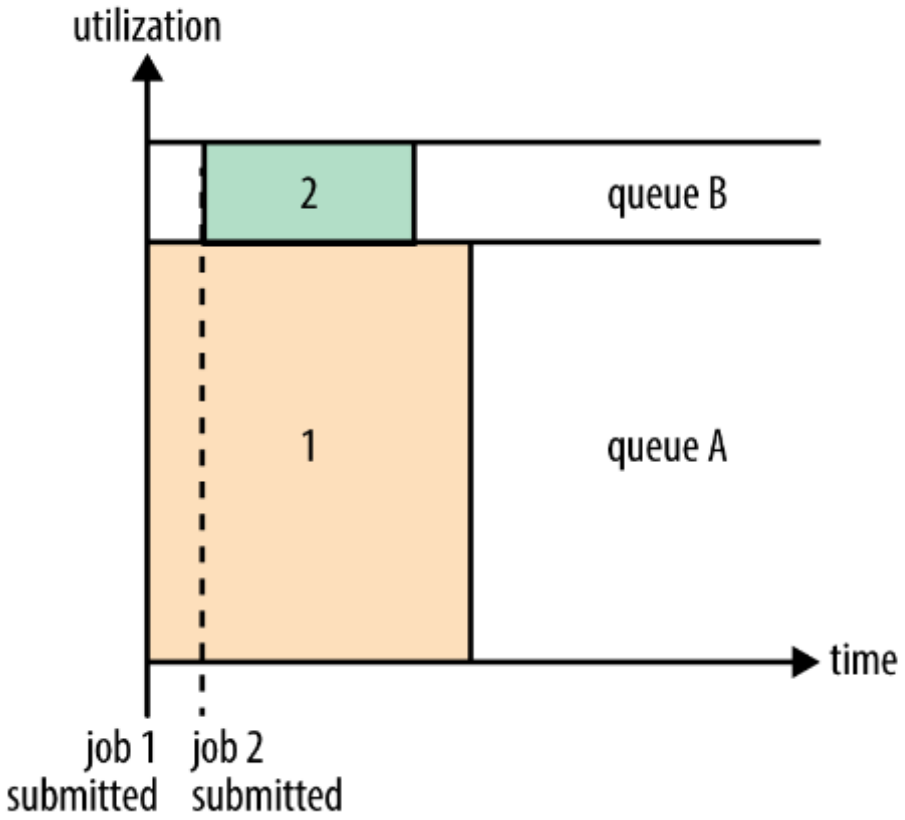
Scheduling in Yarn

- FIFO scheduler
- Simple, no configurations required
- Not good for shared clusters



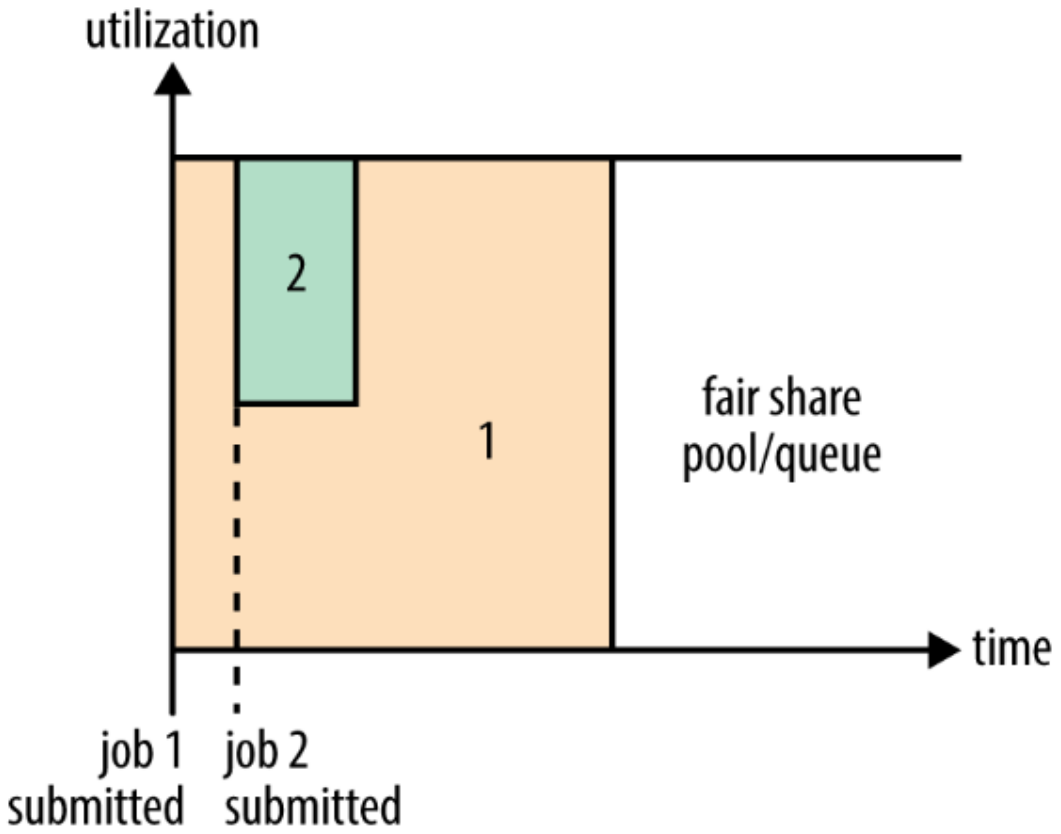
Scheduling in Yarn

ii. Capacity Scheduler

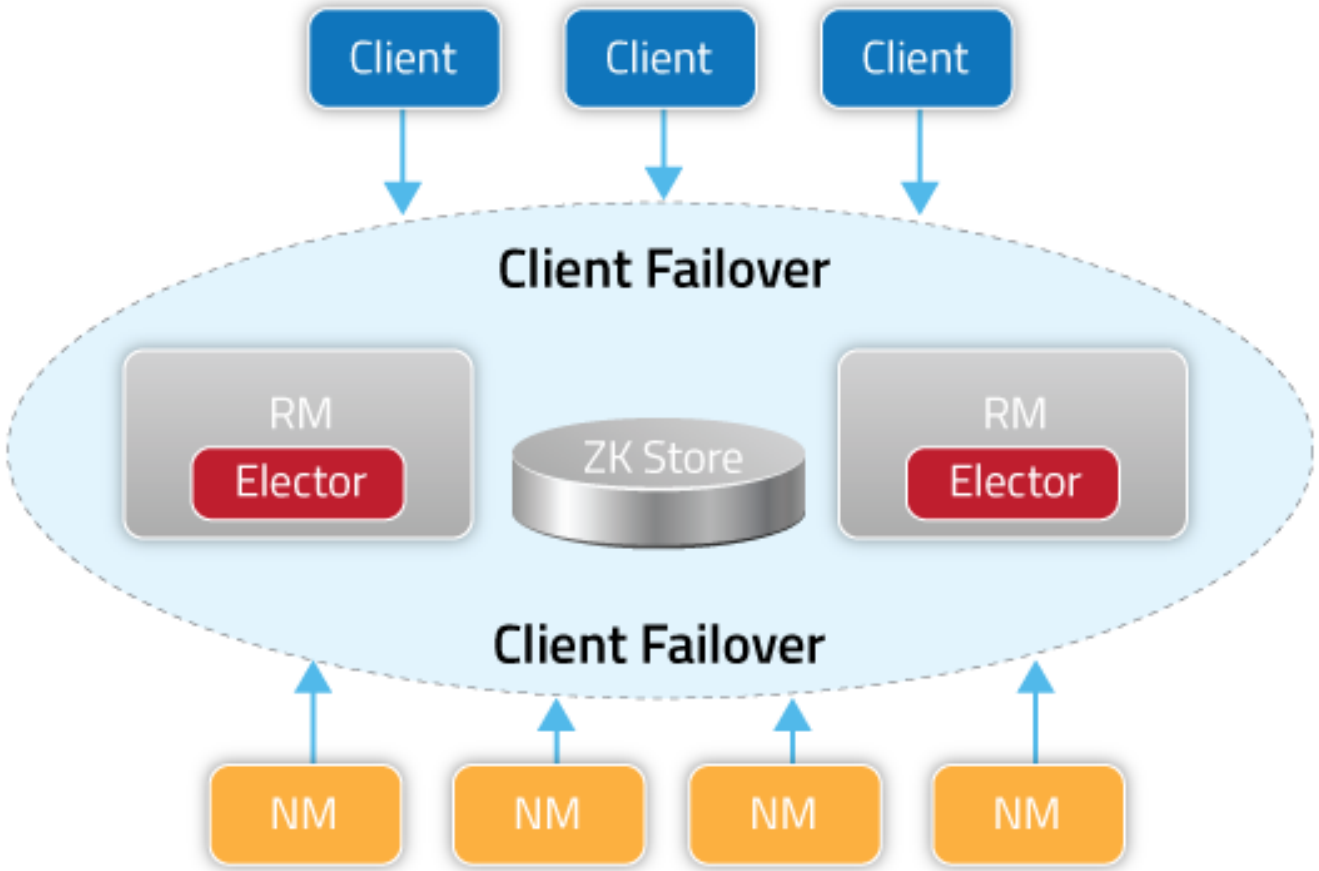


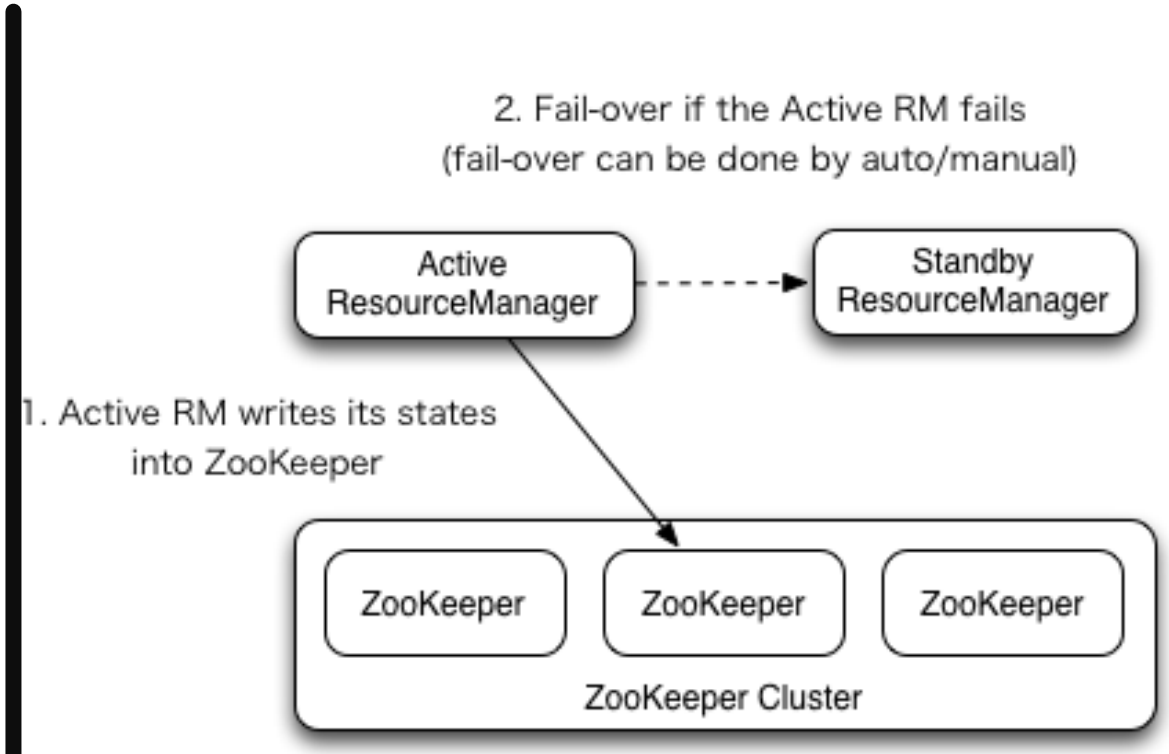
Scheduling in Yarn

iii. Fair Scheduler



Resource Manager HA





Cloudlytics

Planning a Hadoop Cluster

- **Picking a version and distribution of Hadoop**
- **Hardware selection**
  - Master h/w selection
  - Worker hardware selection
- **Cluster sizing / Capacity Planning**
  - The most common way of sizing the cluster is based on the amount of storage required

Capacity Planning

Average daily ingest rate	1 TB	
Replication factor	3 (copies of each block)	
Daily raw consumption	3 TB	Ingest × replication
Node raw storage	24 TB	12 × 2 TB SATA II HDD
MapReduce temp space reserve	25%	For intermediate MapReduce data
Node-usable raw storage	18 TB	Node raw storage – MapReduce reserve
1 year (flat growth)	61 nodes[a]	Ingest × replication × 365 / node raw storage
1 year (5% growth per month[b])	81 nodes[a]	
1 year (10% growth per month)	109 nodes[a]	

Cloudlytics

Capacity Planning

- **Blades, SAN and Virtualization**
- **OS selection**
- **Kernel tuning**
  - vm.swappiness
  - vm.overcommint_memory
    - Three possible settings : 0 , 1 , 2

Capacity Planning

- **Choosing a Filesystem**
  - ext3, ext4, xfs
  - Never use LVM for Hadoop data disks (Unfortunately LVM is default in RedHat and CentOS)

- **ext3**
  - Supports journaling
  - Most stable and highly used filesystem in production
  - Supports files upto 2TB and max filesystem size of 16TB with 4kB block size
  - mkfs -t ext3 -j -m 1 -O sparse_super,dir_index /dev/sd**XN**

Capacity Planning

- **ext4**
  - ext4 is **extent**-based ; storing contiguous blocks together
  - Supports files upto 16TB and max filesystem size of 1ExaByte with 4kB block size
  - Supports journaling with a added feature of journal checksum calculation
  - Default from RHEL6
  - mkfs -t ext4 -m 1 -O dir_index,extent,sparse_super /dev/sd**XN**

- **xfs**
  - High performance journaling filesysytem developed by SGI (Silicon Graphics Inc)
  - Similar to ext4 it supports extent
  - Allows concurrent operations in a way that ext3 and ext4 can't
  - Good for relational databases that performs many parallel, but short lived operations

Hadoop Data Formats

- **Sequence Files**
  - Problem statement
  - Sequence file is a flat file consisting of binary key/value pairs
  - It is extensively used in MapRreduce as input/output formats.
  - The temporary outputs of maps are stored using SequenceFile.
  - It is splittable and also provides compression
  - There are three diffferent sequencefile formats,
    - Uncompressed key/value records
    - Record compressed key/value records - only 'values' are compressed here.
    - Block compressed key/value records - both keys and values are collected in 'blocks' separately and compressed.

Sequence file common header

- version - A byte array: 3 bytes of magic header 'SEQ', followed by 1 byte of actual version no. (example SEQ4,SEQ6)
- keyClassName - String
- valueClassName - String
- compression - A boolean which specifies if compression is turned on for keys/values in this file.
- blockCompression - A boolean which specifies if block compression is turned on for keys/values in this file.
- compressor class - The classname of the CompressionCodec which is used to compress/decompress keys and/or values in this SequenceFile (only if compression is enabled).
- metadata - SequenceFile.Metadata for this file (key/value pairs)
- sync - A sync marker to denote end of the header.

Avro

- Apache Avro is a language-neutral data serialization system.
- Avro datafiles are similar to sequencefile but are defined for large scale data processing
- Avro datafile supports compression and is splittable.
- Avro datafiles are designed to be portable across languages (eg : you can write a file in python and read it in C)

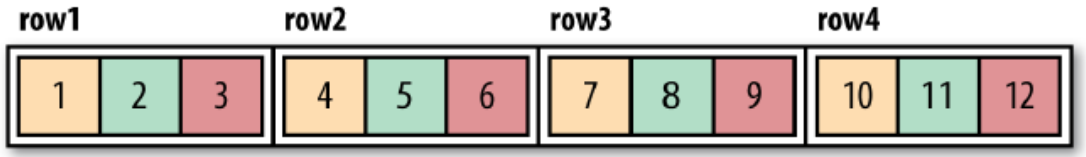
Column-Oriented Formats

- Sequence files and Avro are row-oriented i.e each roe are stored contiguously in file
- In column-oriented format, rows in a file are broken into row splits, then each split is stored in a column-oriented fashion
- Column-oriented formats work well when queries access a small number of columns. Whereas, row oriented formats are good when a large number of columns in rows are needed
- Column oriented formats need more memory while processing data
- Column-oriented not suitable for streaming data
- First column-oriented file format was hive's RCFile (Record Columnar file)
- Its now superseded by ORC (Optimised Record Columnar) and **Parquet**

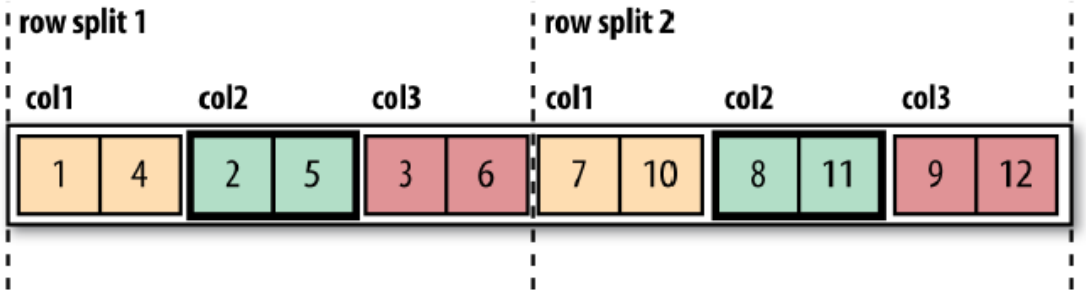
Logical table

	col1	col2	col3
row1	1	2	3
row2	4	5	6
row3	7	8	9
row4	10	11	12

Row-oriented layout (SequenceFile)



Column-oriented layout (RCFile)



“Hadoop 3”

Cloudlytics

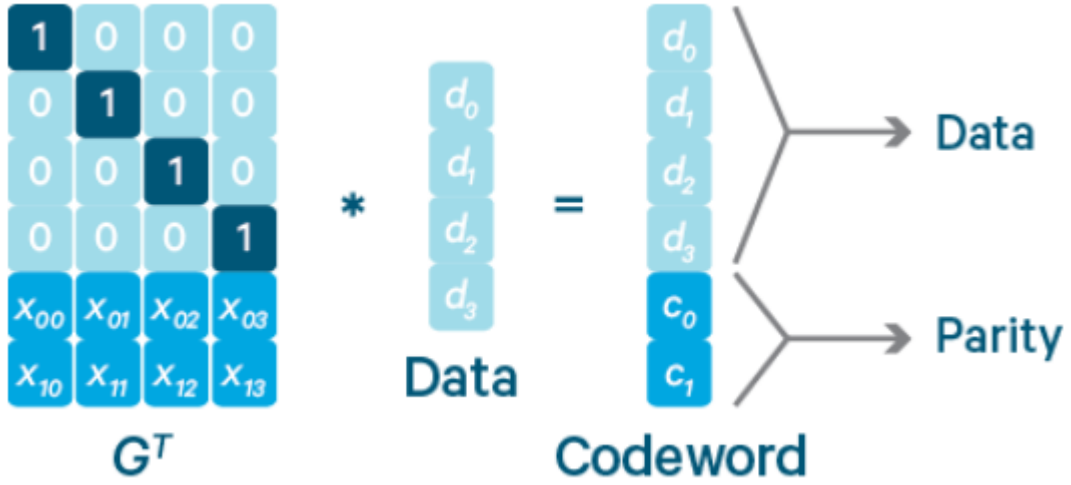
Major Changes

- **Erasure Coding**
  - Replication is expensive – 3x causes 200% overhead
  - EC has no more than 50%
  - Erasure coding (EC) is a branch of information theory which extends a message with redundant data for fault tolerance. An EC codec operates on units of uniformly-sized data termed cells. A codec can take as input a number of data cells and outputs a number of parity cells. This process is called encoding. Together, the data cells and parity cells are termed an erasure coding group. A lost cell can be reconstructed by computing over the remaining cells in the group; this process is called *decoding*.

Erasure codes

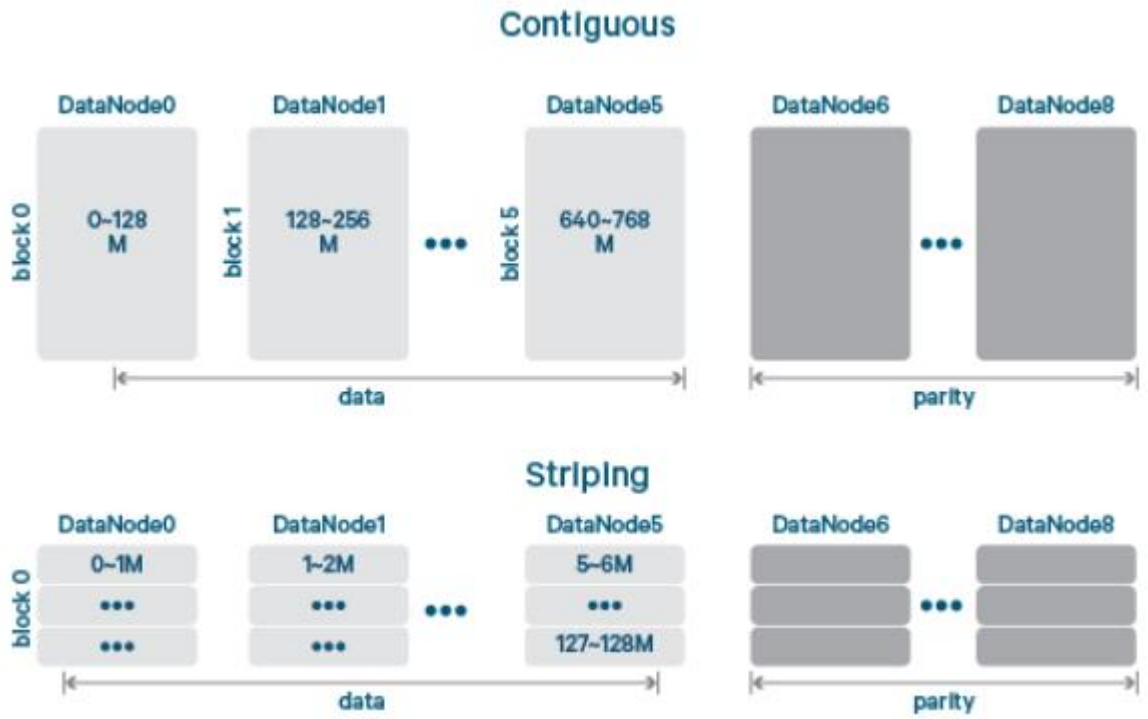
X	Y	X ⊕ Y
0	0	0
0	1	1
1	0	1
1	1	0

XOR encoding

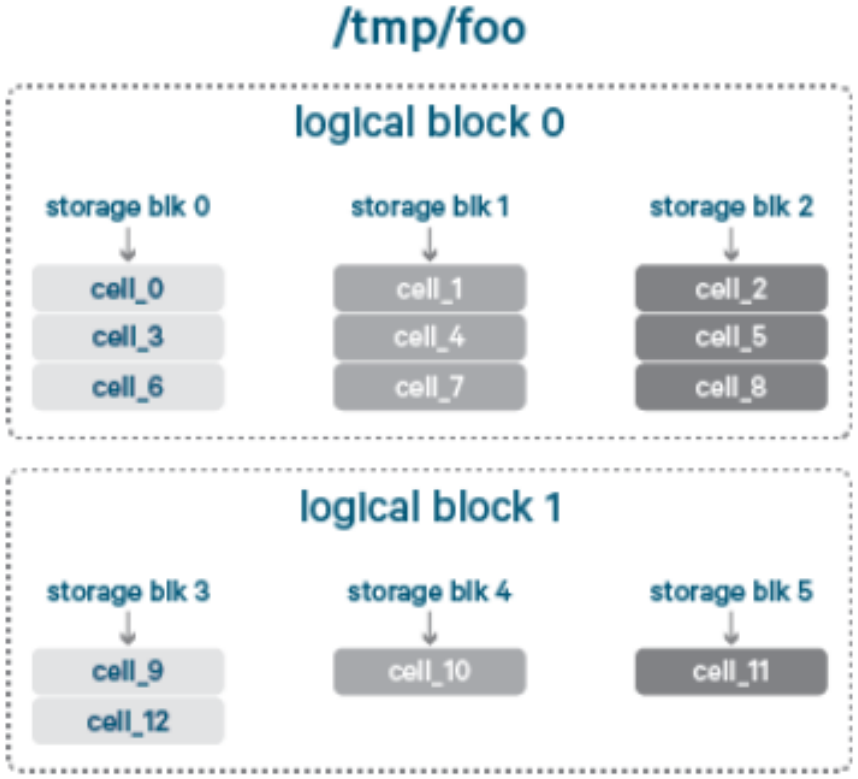


Reed-Solomon Encoding

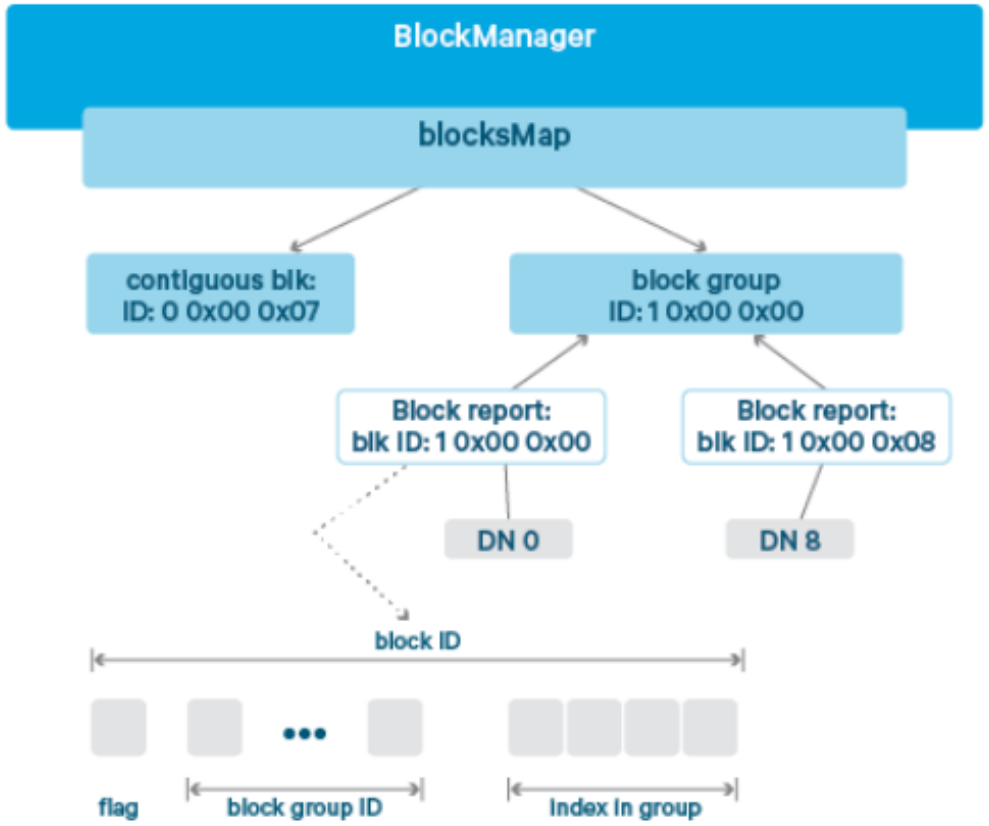
Erasure codes – Distributed systems



Block concept on Namenode



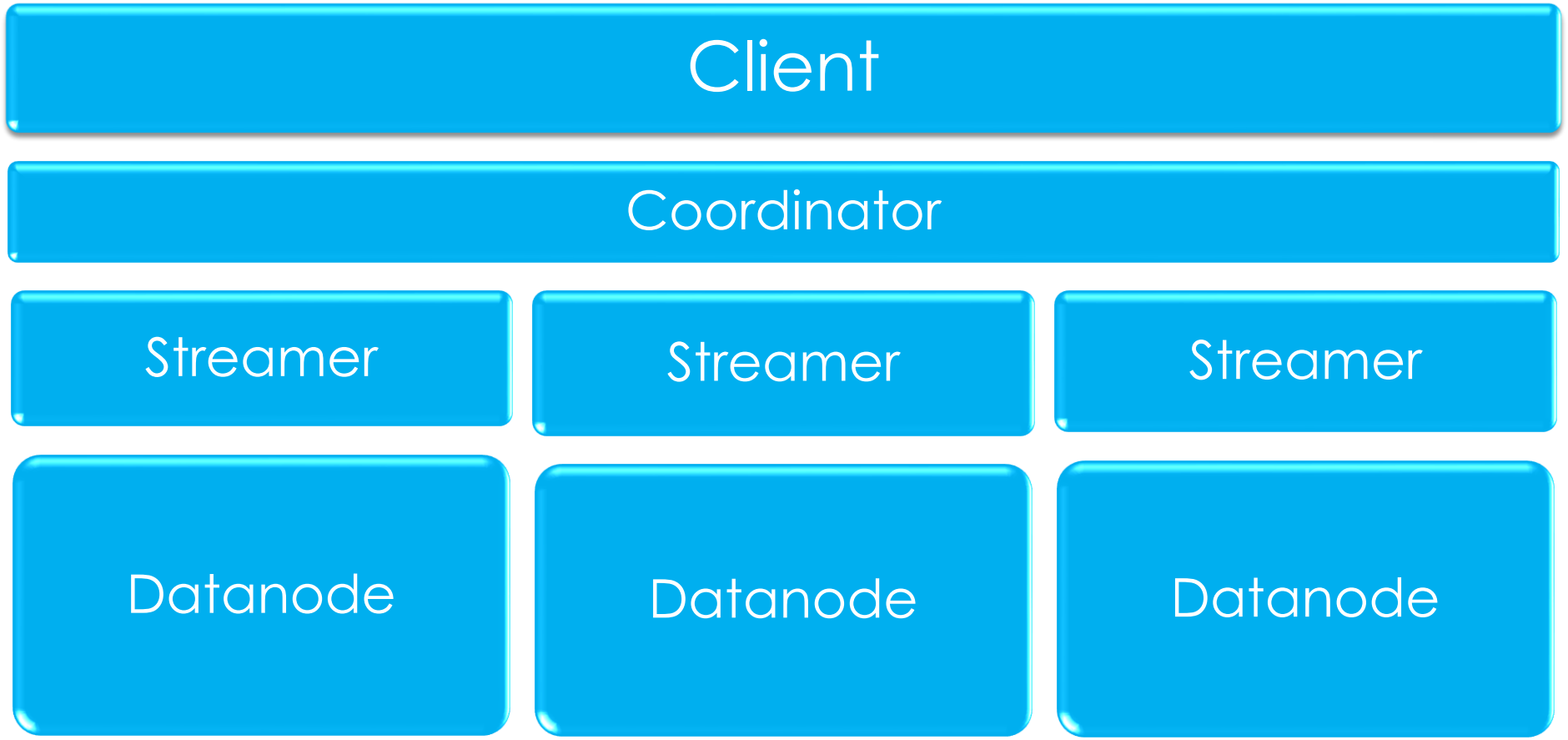
Block concept on Namenode



Block concept on Namenode

Term	Meaning
Logical block	Logical byte range of a file. Consists of a group of storage blocks internally.
Storage block	Chunk of data stored by DataNode
Striping cell	Basic unit of dividing file data for round-robin striped distribution
Data block	Storage with original file data
Parity block	Storage with encoded parity data

Client Extensions



Datanode Extension

- The DataNode runs an additional **ErasureCodingWorker (ECWorker)** task for background recovery of failed erasure coded blocks.
- Recovery task passed as a heartbeat.
- Recovery Process is similar to replication.
- Reconstruction has 3 main tasks
  - Read the data from source nodes
  - Decode the data and generate the output data
  - Transfer the generated data blocks to target nodes

Erasure coding

- Erasure Coding policies
  - Currently, six built-in policies are supported: RS-3-2-1024k, RS-6-3-1024k, RS-10-4-1024k, RS-LEGACY-6-3-1024k, XOR-2-1-1024k and REPLICATION.
- Codec Calculation Framework
  - Data encoding/decoding is very CPU intensive and can be a major overhead when using erasure coding
  - For this we use Intel's ISA-L (Intelligent Storage Acceleration Library)
- Only good for **Cold data.**

Major Changes

- Minimum required Java version increased from Java 7 to Java 8
- New Yarn timeline service v.2
- Opportunistic Containers
  - Unlike existing YARN containers that are scheduled in a node only if there are unallocated resources, opportunistic containers can be dispatched to an NM, even if their execution at that node cannot start immediately. In such a case, opportunistic containers will be queued at that NM until resources become available. The main goal of opportunistic container execution is to improve cluster resource utilization
- Support for more than 2 namenodes in a HA cluster

Major Changes

- Default port number changed
  - Namenode ports: 50470 --> 9871, 50070 --> 9870, 8020 --> 9820
    Secondary NN ports: 50091 --> 9869, 50090 --> 9868
    Datanode ports: 50020 --> 9867, 50010 --> 9866, 50475 --> 9865, 50075 --> 9864
- Intra-datanode balancer
- YARN Resource Types
  - Hadoop 3 enables scheduling of additional resources, such as disks and GPUs for better integration with containers, deep learning & machine learning.

Thank you

Cloudlytics