Unit-II
Hadoop Architecture

What is Hadoop

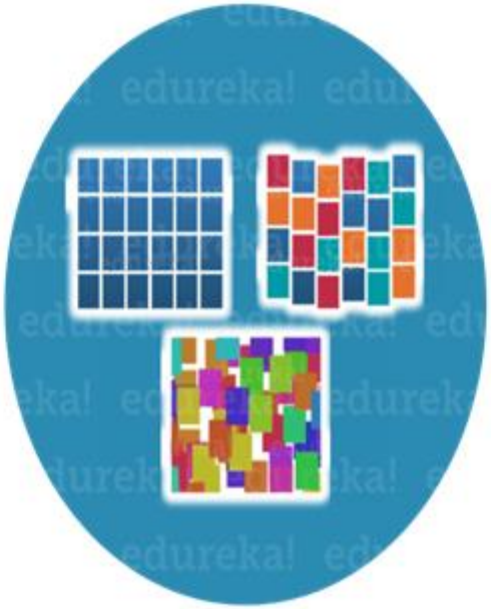
- The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models.
- It is designed to scale up from single servers to thousands of machines, each offering local computation and storage.
- Library designed to detect and handle failures at the application layer.
- Assumption that hardware failures are common and automatically handled by the framework.
- Core of a storage part, known as Hadoop Distributed File System (HDFS)

What is Hadoop

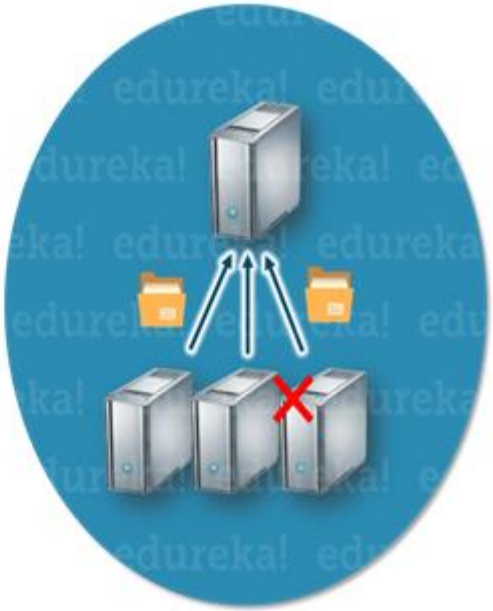
- **Apache Hadoop** is a collection of open-source software utilities that facilitates using a network of many computers to solve problems involving massive amounts of data and computation.
- It provides a software framework for distributed storage and processing of big data using the MapReduce programming model.
- Hadoop was originally designed for computer clusters built from commodity hardware.

What is Hadoop

Storing huge and exponentially growing datasets

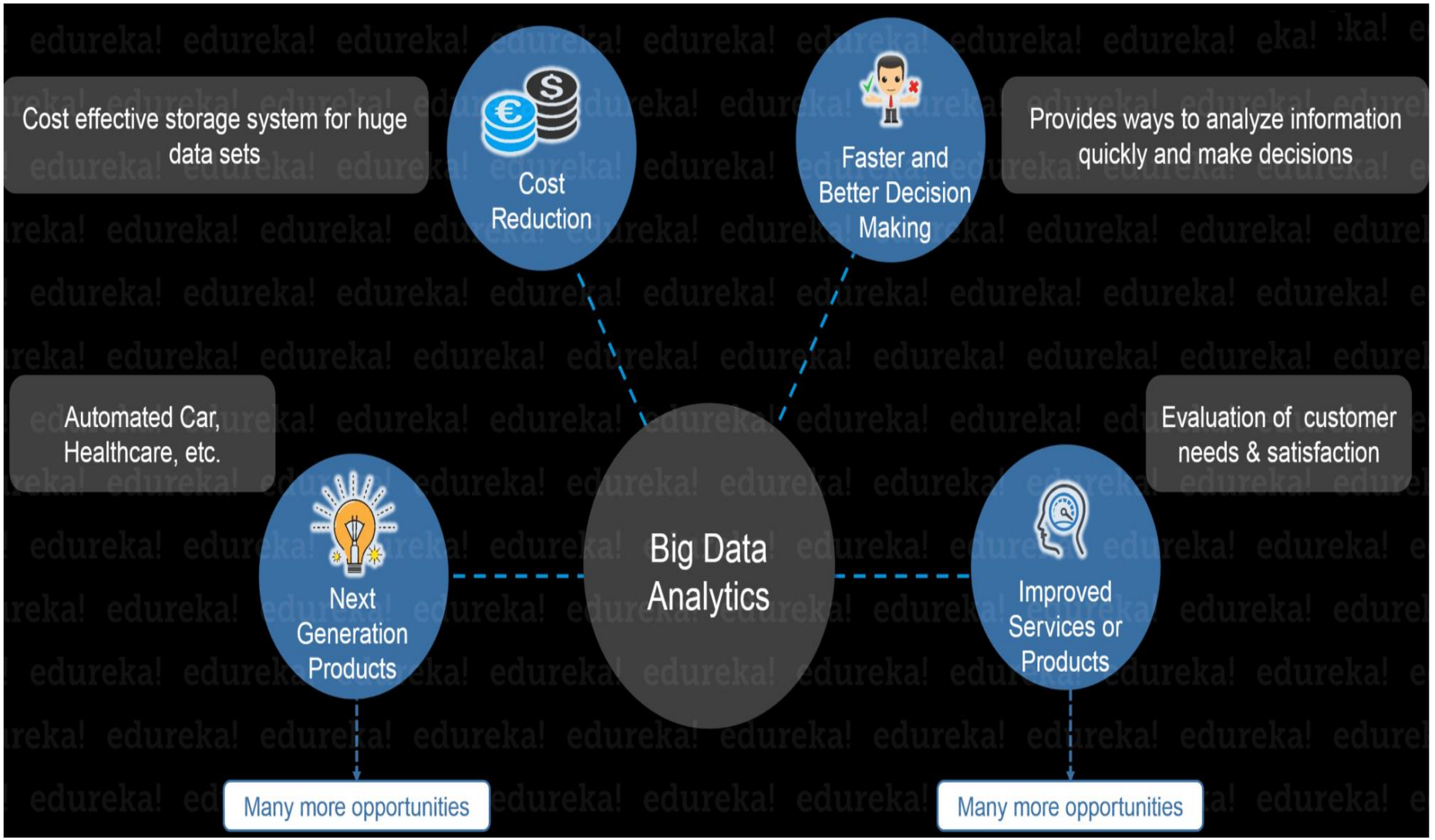


Processing data having complex structure (structured, un-structured, semi-structured)

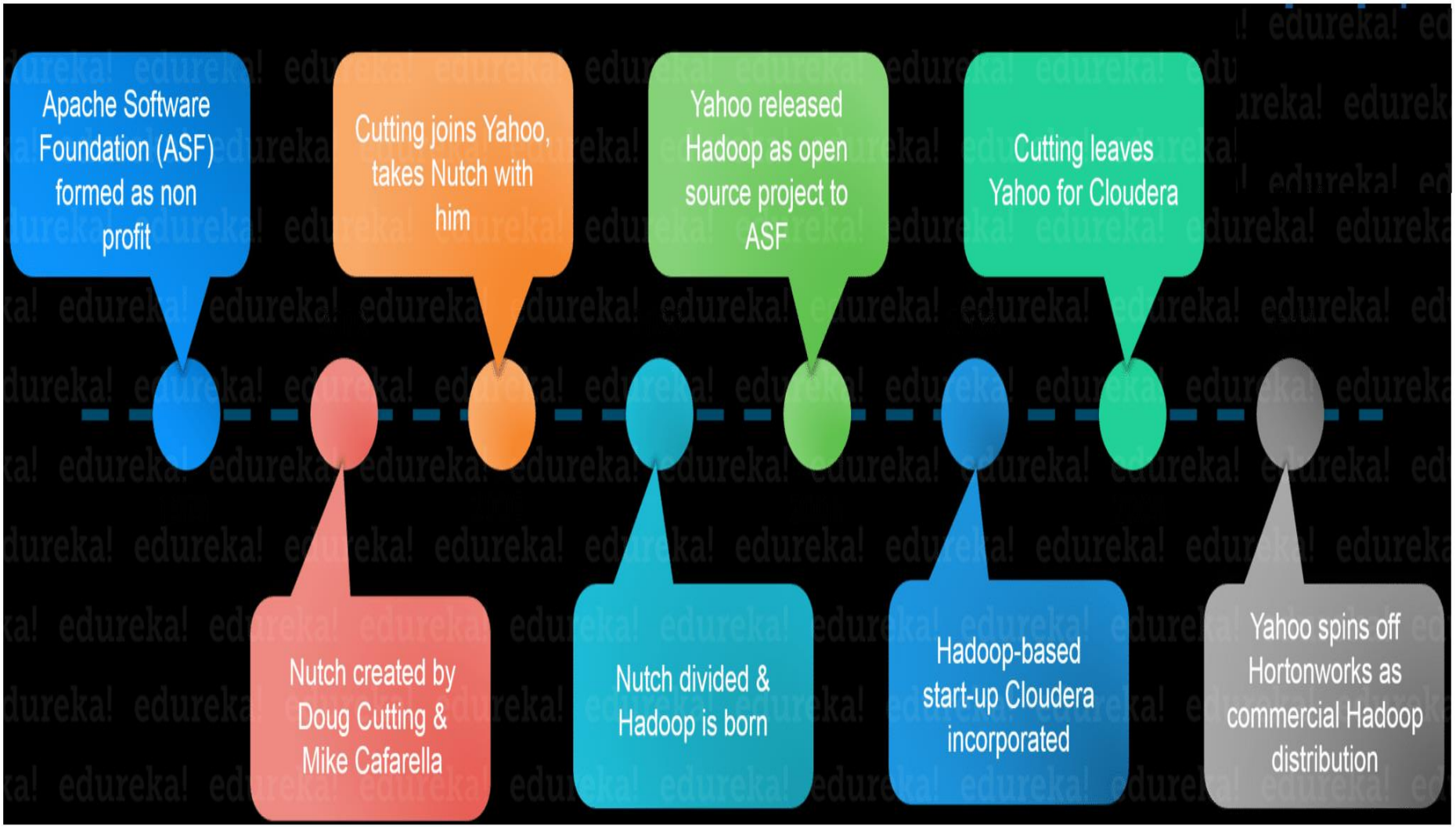


Bringing huge amount of data to computation unit becomes a bottleneck

What is Hadoop – Benefits of Big Data Analytics



Evolution of Hadoop



Hadoop Framework

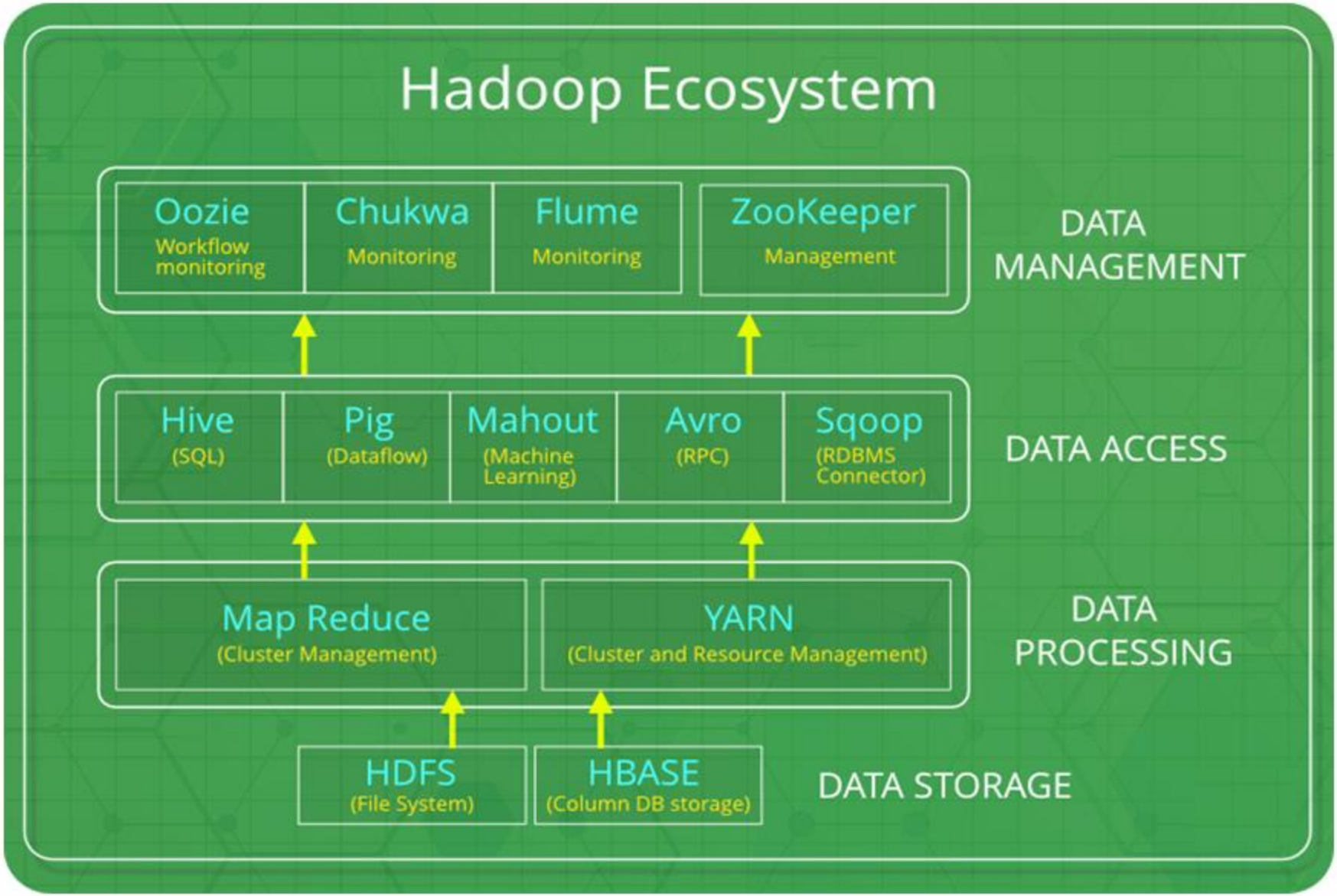
Apache Hadoop framework is composed of :

- **Hadoop Common** – contains libraries and utilities needed by other Hadoop modules;
- **Hadoop Distributed File System (HDFS)** – a distributed file-system that stores data on commodity machines, providing very high aggregate bandwidth across the cluster;
- **Hadoop YARN** – (introduced in 2012) a platform responsible for managing computing resources in clusters and using them for scheduling users' applications
- **Hadoop MapReduce** – an implementation of the MapReduce programming model for large-scale data processing.
- **Hadoop Ozone** – (introduced in 2020) An object store for Hadoop

Hadoop Framework

- The term Hadoop is often used for both base modules and sub-modules and also the ecosystem, or collection of additional software packages
- Software packages that can be installed on top of or alongside Hadoop, such as
  - Apache Pig,
  - Apache Hive,
  - Apache HBase,
  - Apache Phoenix,
  - Apache Spark,
  - Apache ZooKeeper,
  - Cloudera Impala,
  - Apache Flume,
  - Apache Sqoop,
  - Apache Oozie, and
  - Apache Storm

Hadoop Ecosystem



Hadoop Ecosystem

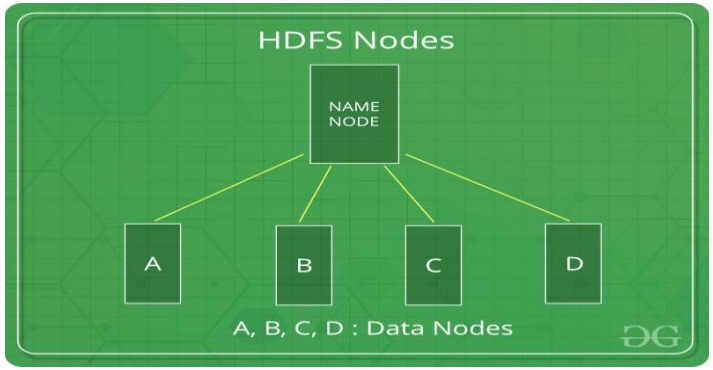
Components of Hadoop ecosystem:

- **HDFS:** Hadoop Distributed File System
- **YARN:** Yet Another Resource Negotiator
- **MapReduce:** Programming based Data Processing
- **Spark:** In-Memory data processing
- **PIG, HIVE:** Query based processing of data services
- **HBase:** NoSQL Database
- **Mahout, Spark MLLib:** Machine Learning algorithm libraries
- **Solar, Lucene:** Searching and Indexing
- **Zookeeper:** Managing cluster
- **Oozie:** Job Scheduling

Hadoop Ecosystem

HDFS:

- HDFS is the primary or major component of Hadoop ecosystem and is responsible for storing large data sets of structured or unstructured data across various nodes and thereby maintaining the metadata in the form of log files.
- HDFS consists of two core components i.e.
  - Name node
  - Data Node
- Name Node is the prime node which contains metadata (data about data) requiring comparatively fewer resources than the data nodes that stores the actual data.
- HDFS maintains all the coordination between the clusters and hardware, thus working at the heart of the system.



Hadoop Ecosystem

YARN:

- Yet Another Resource Negotiator, as the name implies, YARN is the one who helps to manage the resources across the clusters.
- It performs scheduling and resource allocation for the Hadoop System.
- Consists of three major components i.e.
  - Resource Manager
  - Nodes Manager
  - Application Manager
- Resource manager has the privilege of allocating resources for the applications in a system
- Node managers work on the allocation of resources such as CPU, memory, bandwidth per machine and later on acknowledges the resource manager.
- Application manager works as an interface between the resource manager and node manager and performs negotiations as per requirement of two.

Hadoop Ecosystem

MapReduce

- By making the use of distributed and parallel algorithms, MapReduce makes it possible to carry over the processing's logic and helps to write applications which transform big data sets into a manageable one.
- MapReduce makes the use of two functions i.e. Map() and Reduce() whose task is:
  1. Map() performs sorting and filtering of data and thereby organizing them in the form of group. Map generates a key-value pair based result which is later on processed by the Reduce() method.
  2. Reduce(), as the name suggests does the summarization by aggregating the mapped data.
- In simple, Reduce() takes output generated by Map() as input and combines those tuples into smaller set of tuples.

Hadoop Ecosystem

PIG:

- Pig was basically developed by Yahoo which works on a pig Latin language, which is Query based language similar to SQL.
- It is a platform for structuring the data flow, processing and analyzing huge data sets.
- Pig does the work of executing commands and in the background, all the activities of MapReduce are taken care of.
- Pig Latin language is specially designed for this framework which runs on Pig Runtime.
- Pig helps to achieve ease of programming and optimization and hence is a major segment of the Hadoop Ecosystem.

Hadoop Ecosystem

HIVE:

- With the help of SQL methodology and interface, HIVE performs reading and writing of large data sets.
- its query language is called as HQL (Hive Query Language).
- It is highly scalable as it allows real-time processing and batch processing both.
- All the SQL datatypes are supported by Hive thus, making the query processing easier.
- Similar to the Query Processing frameworks, HIVE comes with two components: JDBC Drivers and HIVE Command Line.
- JDBC, along with ODBC drivers work on establishing the data storage permissions and connection whereas HIVE Command line helps in the processing of queries.

Hadoop Ecosystem

Mahout:

- Mahout, allows Machine Learnability to a system or application.
- It provides various libraries or functionalities such as collaborative filtering, clustering, and classification.
- It allows invoking algorithms as per our need with the help of its own libraries.

Hadoop Ecosystem

Apache Spark:

- It's a platform that handles all the process consumptive tasks like batch processing, interactive or iterative real-time processing, graph conversions, and visualization etc.
- It consumes in memory resources hence, thus being faster than the prior in terms of optimization.
- Spark is best suited for real-time data whereas Hadoop is best suited for structured data or batch processing, hence both are used in most of the companies interchangeably.

Hadoop Ecosystem

Apache HBase:

- It's a NoSQL database which supports all kinds of data and thus capable of handling anything of Hadoop Database.
- It provides capabilities of Google's BigTable, thus able to work on Big Data sets effectively.
- At times where we need to search or retrieve the occurrences of something small in a huge database, the request must be processed within a short quick span of time. At such times, HBase comes handy as it gives us a tolerant way of storing limited data.

Design of Hadoop distributed file system (HDFS)

- When a dataset outgrows the storage capacity of a single physical machine, it becomes necessary to partition it across a number of separate machines.
- Filesystems that manage the storage across a network of machines are called *distributed filesystems*.
- Network based - complications of network programming kick in
- Complex than regular disk filesystems.
- Biggest challenges is making filesystem node failure tolerate without suffering data loss.

Design of HDFS

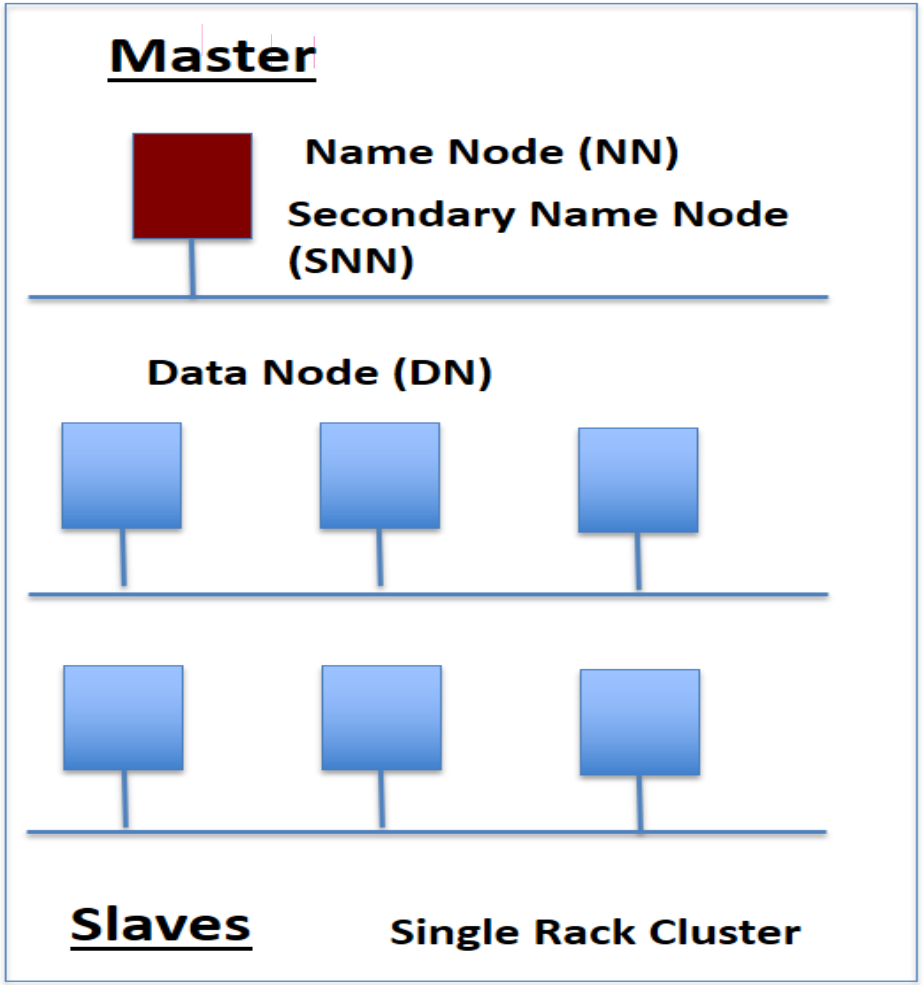
- HDFS is a filesystem designed for storing very large files with streaming data access patterns, running on clusters of commodity hardware.
  - *Very large files*
  - *Streaming data access*
  - *Commodity hardware*
- Applications for which using HDFS does not work so well
  - *Low-latency data access*
  - *Lots of small files*
  - *Multiple writers, arbitrary file modifications*

Design of HDFS - Blocks

- A disk has a block size, which is the minimum amount of data that it can read or write.
- Filesystem blocks - few kilobytes in size, disk blocks are normally 512 bytes
- HDFS has block, —128 MB by default
- Files in HDFS are broken into block-sized chunks
- Large block size
  - File can be larger than any single disk in the network
  - Simplifies the storage subsystem
  - Blocks fit well with replication for providing fault tolerance and availability.
- A block that is no longer available due to corruption or machine failure can be replicated from its alternative locations to other live machines to bring replication factor back to the normal level.

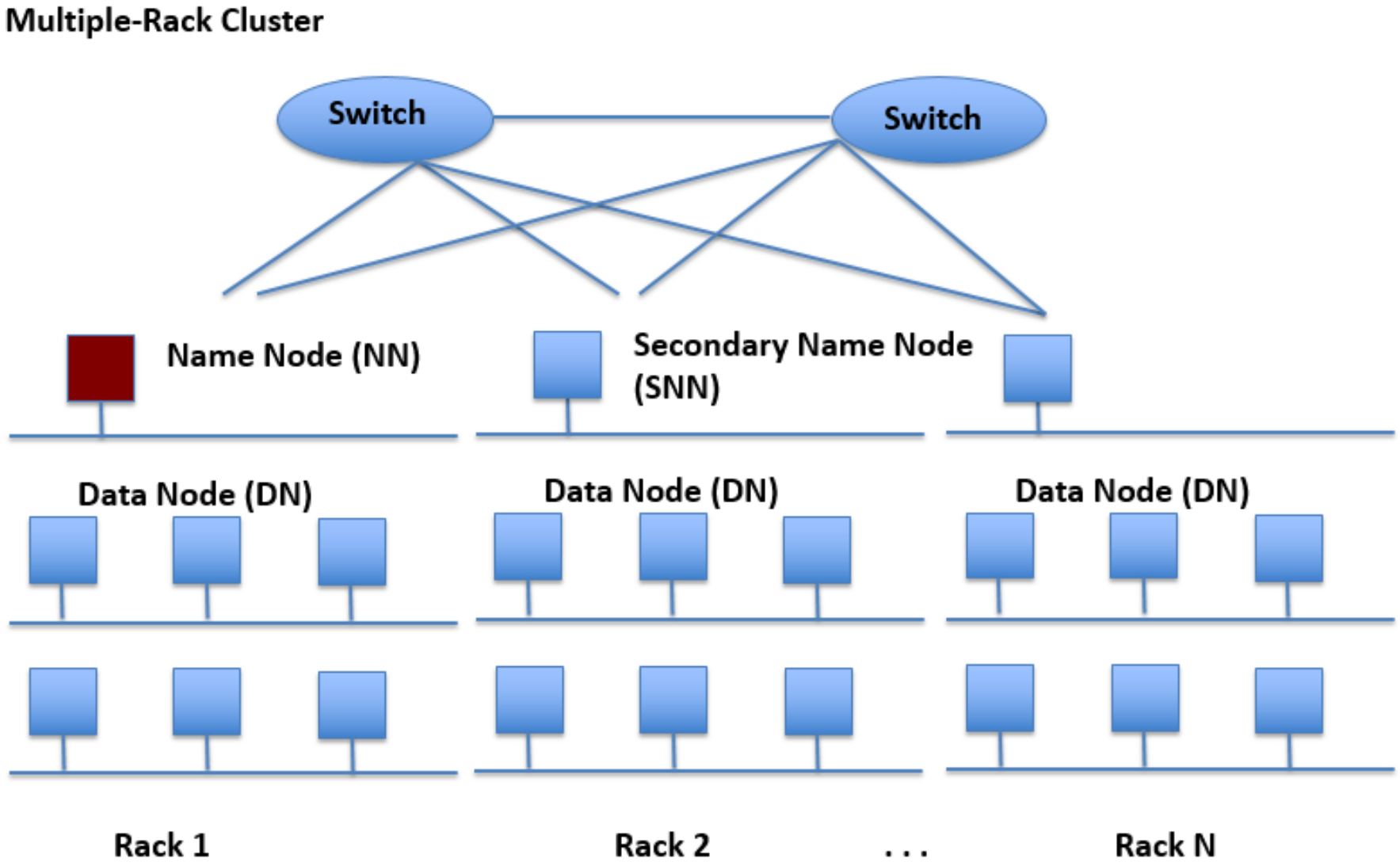
Design of HDFS - Namenodes and Datanodes

An HDFS cluster has two types of nodes operating in a master−worker pattern: a *namenode* (the master) and a number of *datanodes* (workers).



- Name Node: Controller
  - File System Name Space Management
  - Block Mappings
- Data Node: Work Horses
  - Block Operations
  - Replication
- Secondary Name Node:
  - Checkpoint node

Design of HDFS - Namenodes and Datanodes



Design of HDFS - Namenodes and Datanodes

Hadoop Cluster at Yahoo! (Credit: Yahoo)

Design of HDFS - Namenodes and Datanodes

- A *client* accesses the filesystem on behalf of the user by communicating with the namenode and datanodes.
- User code does not need to know about the namenode and datanodes to function
- Datanodes are the workhorses of the filesystem.
- Without the namenode, the filesystem cannot be used
- Mechanisms to make the namenode resilient to failure are:
  - Back up the files that make up the persistent state of the filesystem metadata
  - Run a *secondary namenode*, which despite its name does not act as a namenode
- 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.
- Possible to run a hot standby namenode instead of a secondary

Design of HDFS - Block Caching

- Blocks may be explicitly cached in the datanode's memory
- Job schedulers can take advantage of cached blocks by running tasks on the datanode where a block is cached
- Users or applications instruct the namenode which files to cache
- Cache pools are an administrative grouping for managing cache permissions and resource usage.

Design of HDFS - HDFS Federation

- namenode keeps a reference to every file and block in the filesystem
- Becomes the limiting factor for scaling
- HDFS federation, introduced in the 2.x release series, allows a cluster to scale by adding namenodes, each of which manages a portion of the filesystem namespace.
- For example, one namenode might manage all the files rooted under */user*, say, and a second namenode might handle files under */share*.
- Each namenode manages a *namespace volume*,
- *namespace volume* is made up of the metadata for the namespace, and a *block pool* containing all the blocks for the files in the namespace.
- Namespace volumes are independent of each other
- Failure of one namenode does not affect the availability of the namespaces managed by other namenodes.

Design of HDFS - HDFS High Availability

- Replicating namenode metadata and using the secondary namenode to create checkpoints protects against data loss
- But it does not provide high availability of the filesystem.
- The namenode is still a single point of failure (SPOF).
- Namenode failure - whole Hadoop system would effectively be out of service until a new namenode could be brought online.
- Administrator starts a new primary namenode with one of the filesystem metadata replicas and configures datanodes and clients to use this new namenode.
- The new namenode is not able to serve requests until it has
  I. loaded its namespace image into memory,
  II. replayed its edit log, and
  III. received enough block reports from the datanodes to leave safe mode.
- On large clusters with many files and blocks, the time it takes for a namenode to start from cold can be 30 minutes or more.
- Long recovery time is a problem

Design of HDFS - HDFS High Availability

- Hadoop 2 remedied this situation by adding support for HDFS high availability (HA).
- Pair of namenodes in an active-standby configuration.
- Failure of the active namenode - standby takes over its duties
- A few architectural changes are needed to allow this to happen:
  - The namenodes must use highly available shared storage to share the edit log
  - Datanodes must send block reports to both namenodes
  - Clients must be configured to handle namenode failover, is transparent to users.
  - The secondary namenode's takes periodic checkpoints of the active namenode's namespace.

Design of HDFS - HDFS High Availability

- There are two choices for the highly available shared storage:
  1. an NFS filer, or
  2. a *quorum journal manager* (QJM).
- The QJM is a dedicated HDFS implementation, designed for a highly available edit log,
- QJM recommended choice for most HDFS installations.
- The QJM runs as a group of *journal nodes*, and each edit must be written to a majority of the journal nodes.
- Typically, there are three journal nodes, so the system can tolerate the loss of one of them.
- This arrangement is similar to the way ZooKeeper works

Design of HDFS - Failover and fencing

- The transition from active namenode to the standby is managed by a new entity in the system called the failover controller.
- Uses ZooKeeper to ensure that only one namenode is active.
- Each namenode runs a lightweight failover controller process whose job is to monitor its namenode for failures (using a simple heartbeating mechanism) and trigger a failover.
- Impossible to be sure that the failed namenode has stopped running (slow network or a network partition)
- The HA implementation goes to great lengths to ensure that the previously active namenode is prevented from doing any damage and causing corruption—a method known as fencing
- The QJM only allows one namenode to write to the edit log at one time;
- Fencing methods
  - kill the earlier namenode's process
  - revoking the namenode's access to the shared storage directory
  - disabling its network port via a remote management command

Design of HDFS - Failover and fencing

- Client failover is handled transparently by the client library.
- The simplest implementation uses client-side configuration to control failover.
- HDFS URI uses a logical hostname that is mapped to a pair of namenode addresses (in the configuration file)
- Client library tries each namenode address until the operation succeeds.

Design of HDFS

- **Problem 1:** Data is too big to store on one machine.

- **HDFS:** Store the data on multiple machines!

Design of HDFS

- **Problem 2:** Very high end machines are too expensive

- **HDFS:** Run on commodity hardware!

Design of HDFS

- **Problem 3:** Commodity hardware will fail!

- **HDFS:** Software is intelligent enough to handle hardware failure!

Design of HDFS

- **Problem 4:** What happens to the data if the machine stores the data fails?

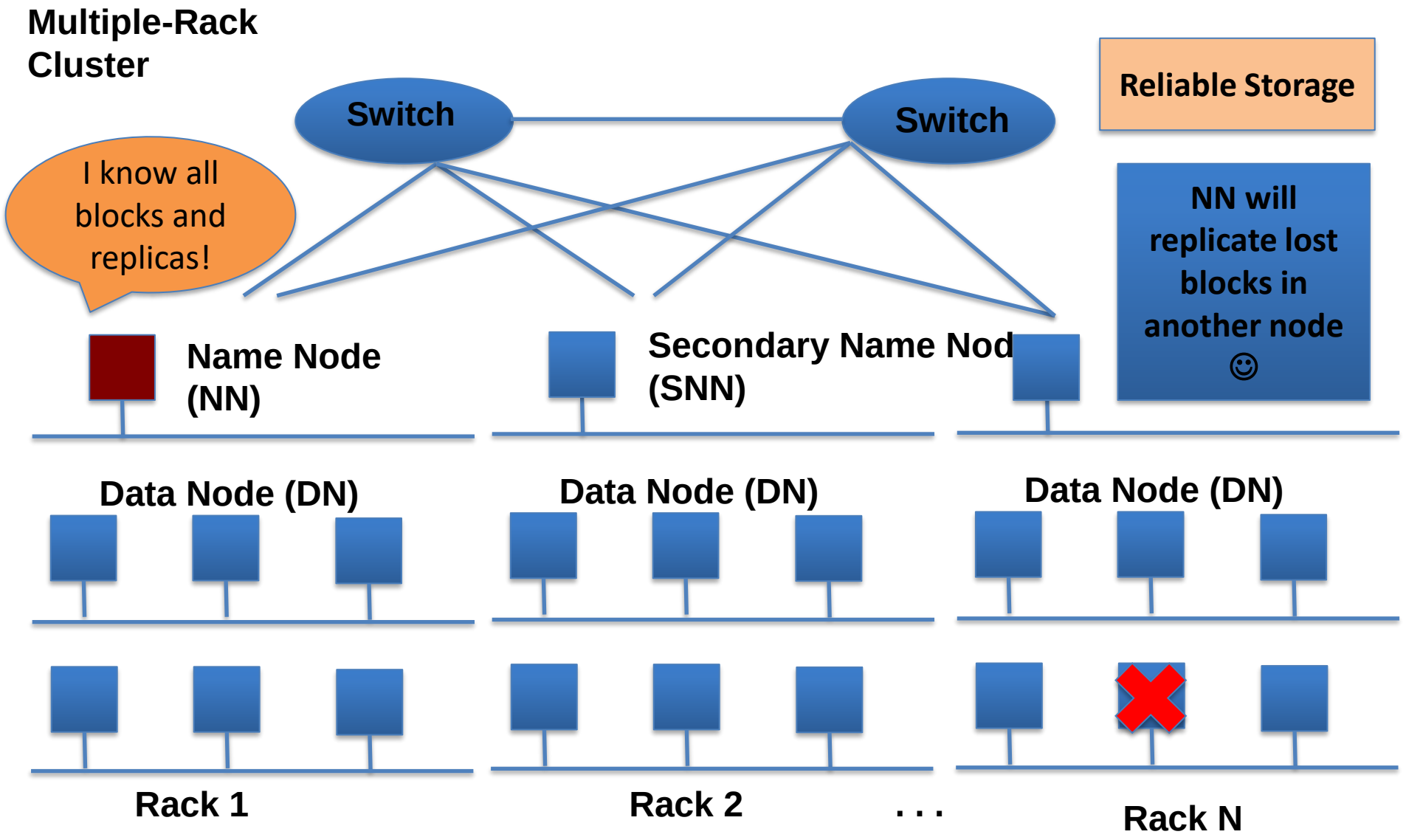
- **HDFS:** Replicate the data!

Design of HDFS

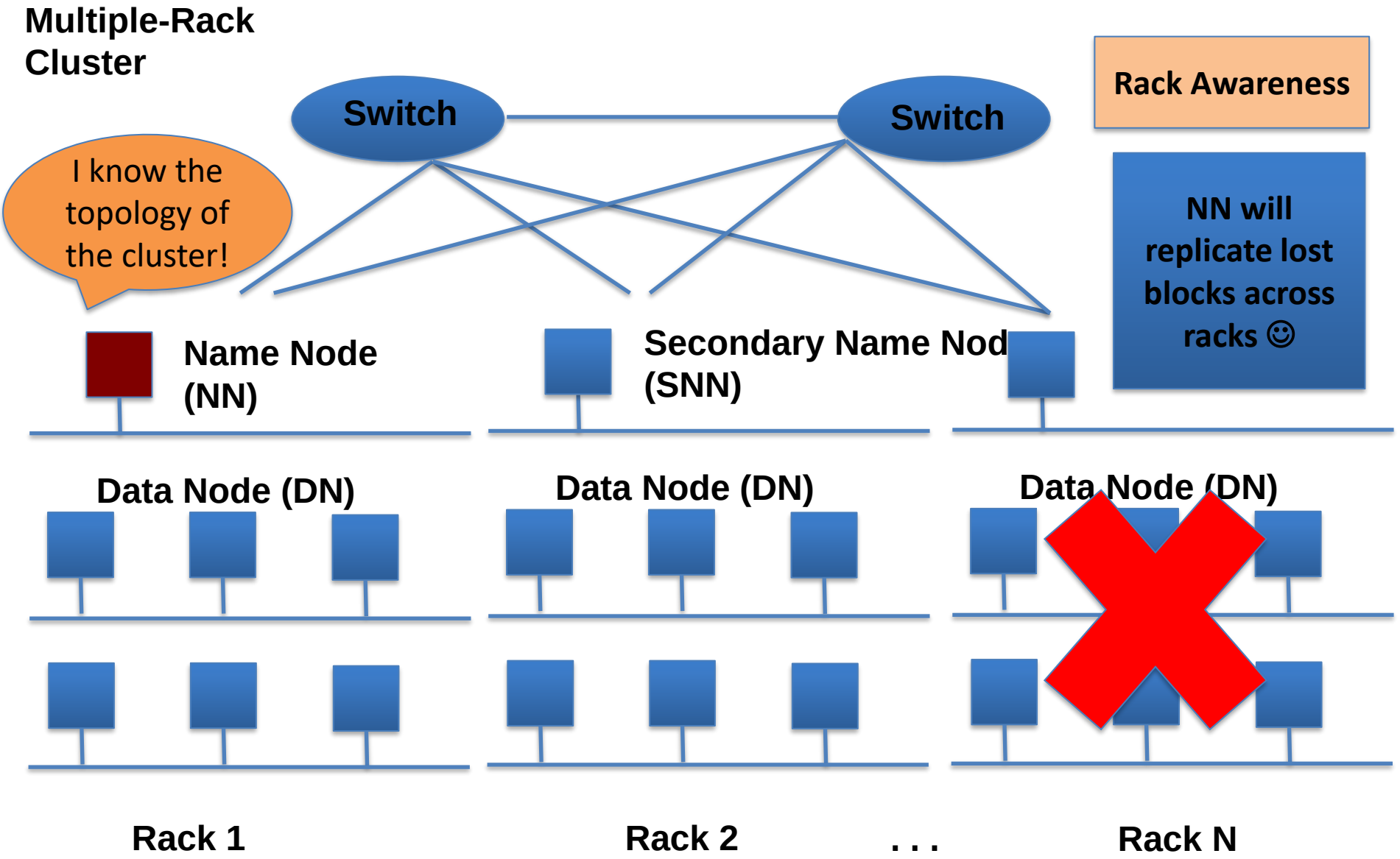
- **Problem 5:** How can distributed machines organize the data in a coordinated way?

- **HDFS:** Master-Slave Architecture!

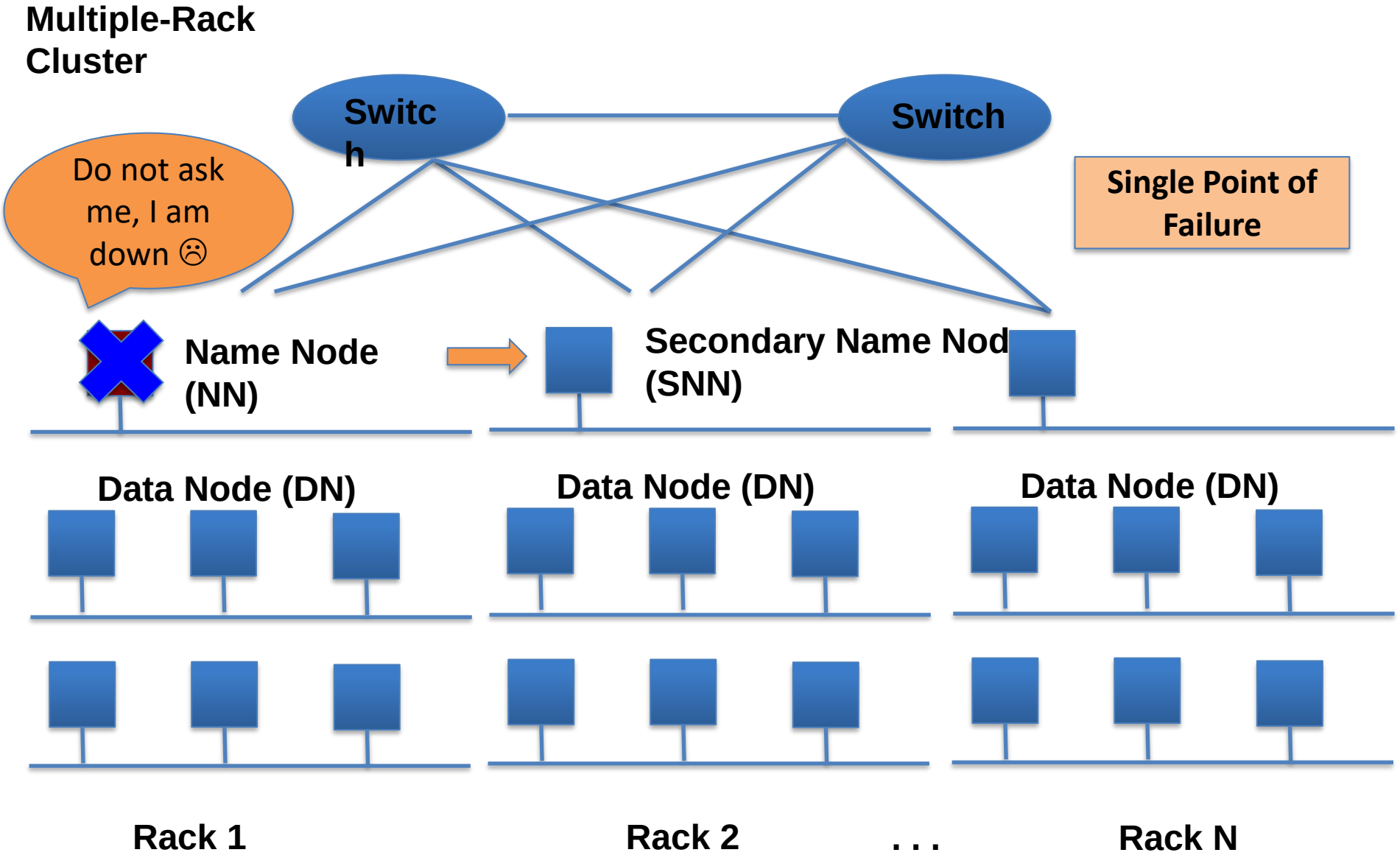
Design of HDFS



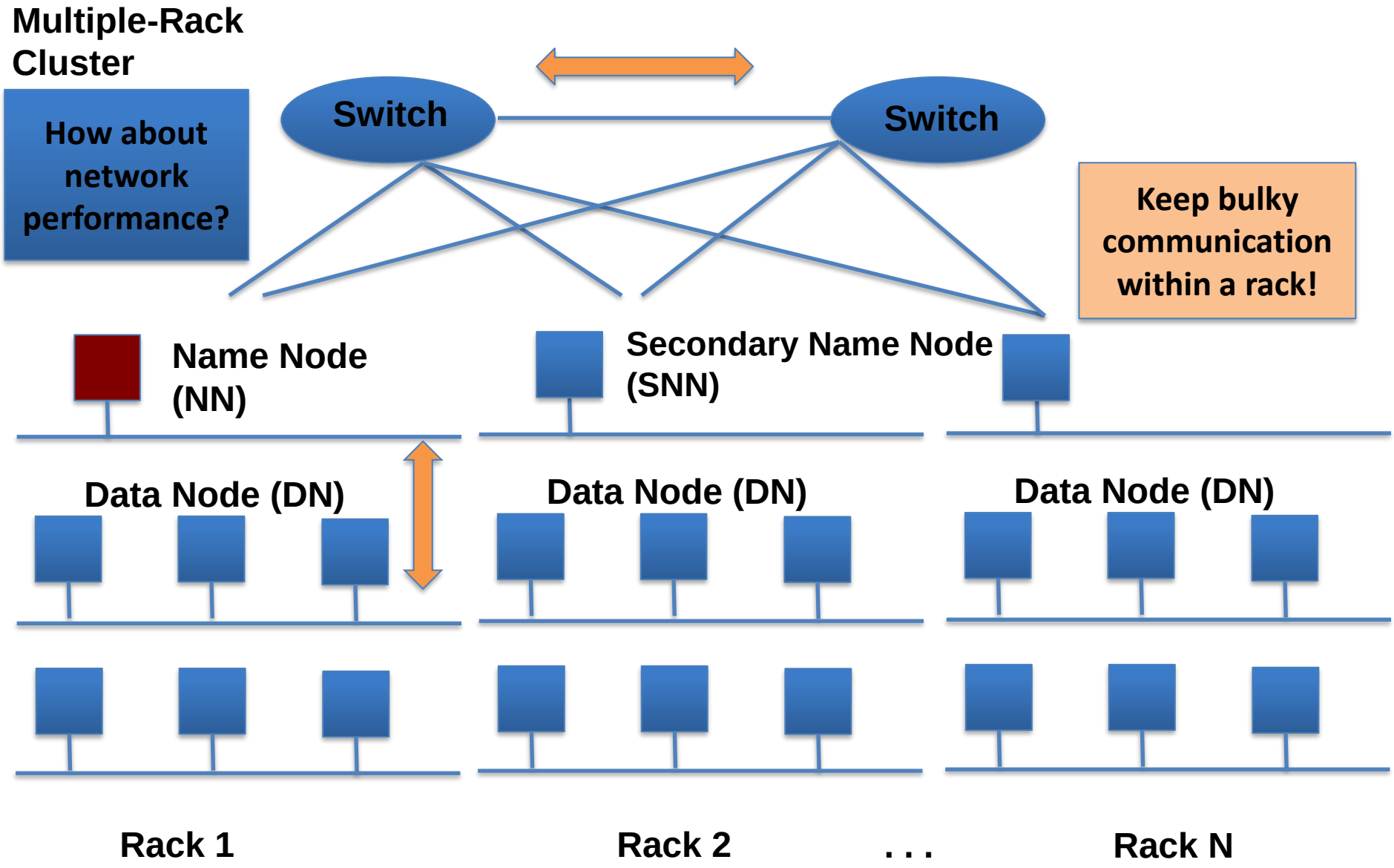
Design of HDFS



Design of HDFS



Design of HDFS



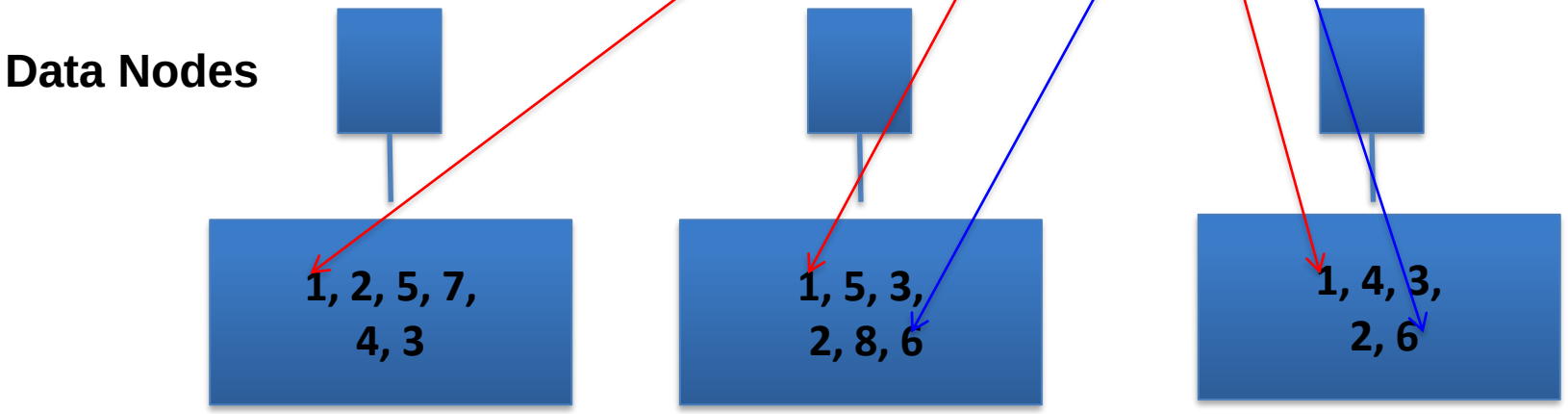
HDFS Inside: Name Node

Name Node

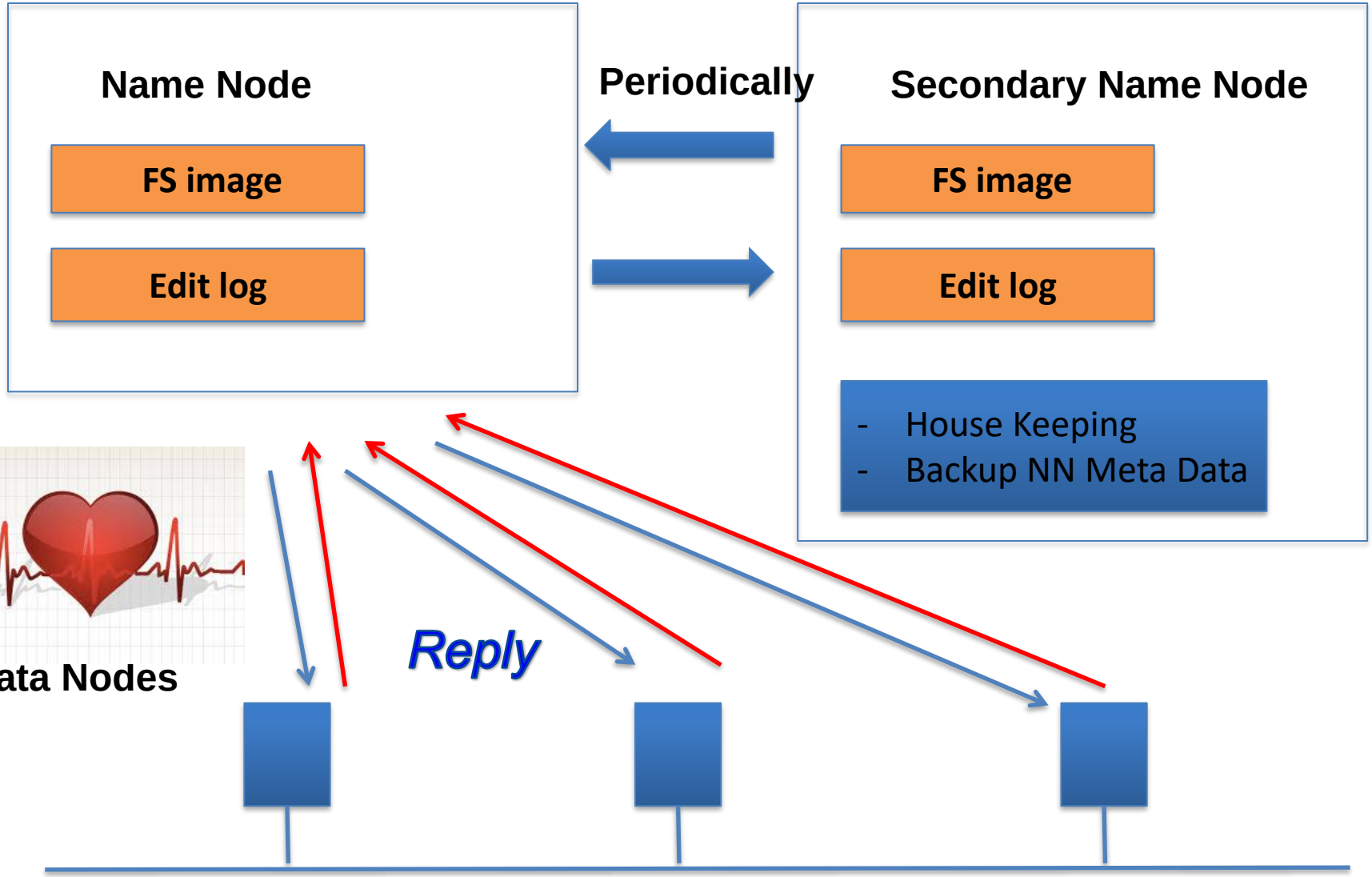
Snapshot of FS

Edit log: record changes to FS

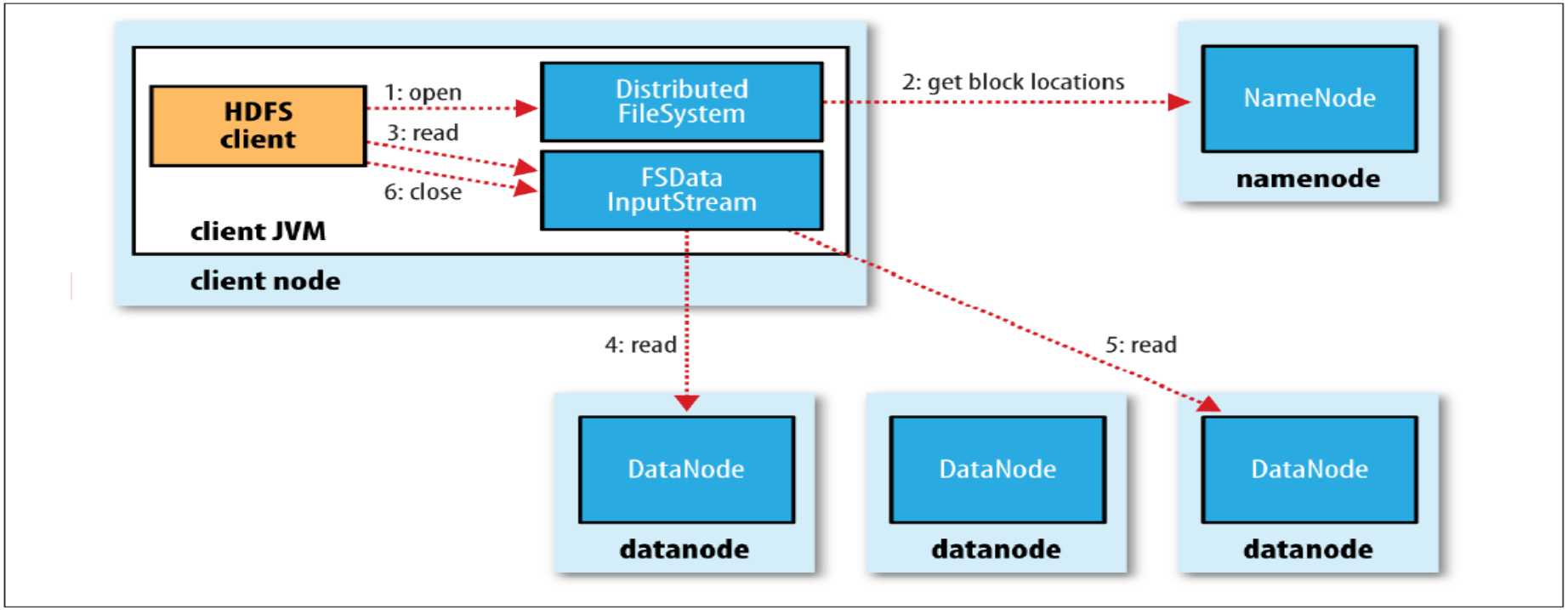
Filename	Replication factor	Block ID
File 1	3	[1, 2, 3]
File 2	2	[4, 5, 6]
File 3	1	[7,8]



HDFS Inside: Name Node



Data Flow - Anatomy of a File Read



A client reading data from HDFS

- Client connects to NN to read data
- NN tells client where to find the data blocks
- Client reads blocks directly from data nodes (without going through NN)
- In case of node failures, client connects to another node that serves the missing block

HDFS Inside: Read

- Q: Why does HDFS choose such a design for read? Why not ask client to read blocks through NN?
- Reasons:
  - Prevent NN from being the bottleneck of the cluster
  - Allow HDFS to scale to large number of concurrent clients
  - Spread the data traffic across the cluster

HDFS Inside: Network Topology

- The critical resource in HDFS is **bandwidth**, distance is defined based on that
- Measuring bandwidths between any pair of nodes is too complex and **does not scale**
- **Basic Idea:**
  - Processes on the same node
  - Different nodes on the same rack
  - Nodes on different racks in the same data center (cluster)
  - Nodes in different data centers

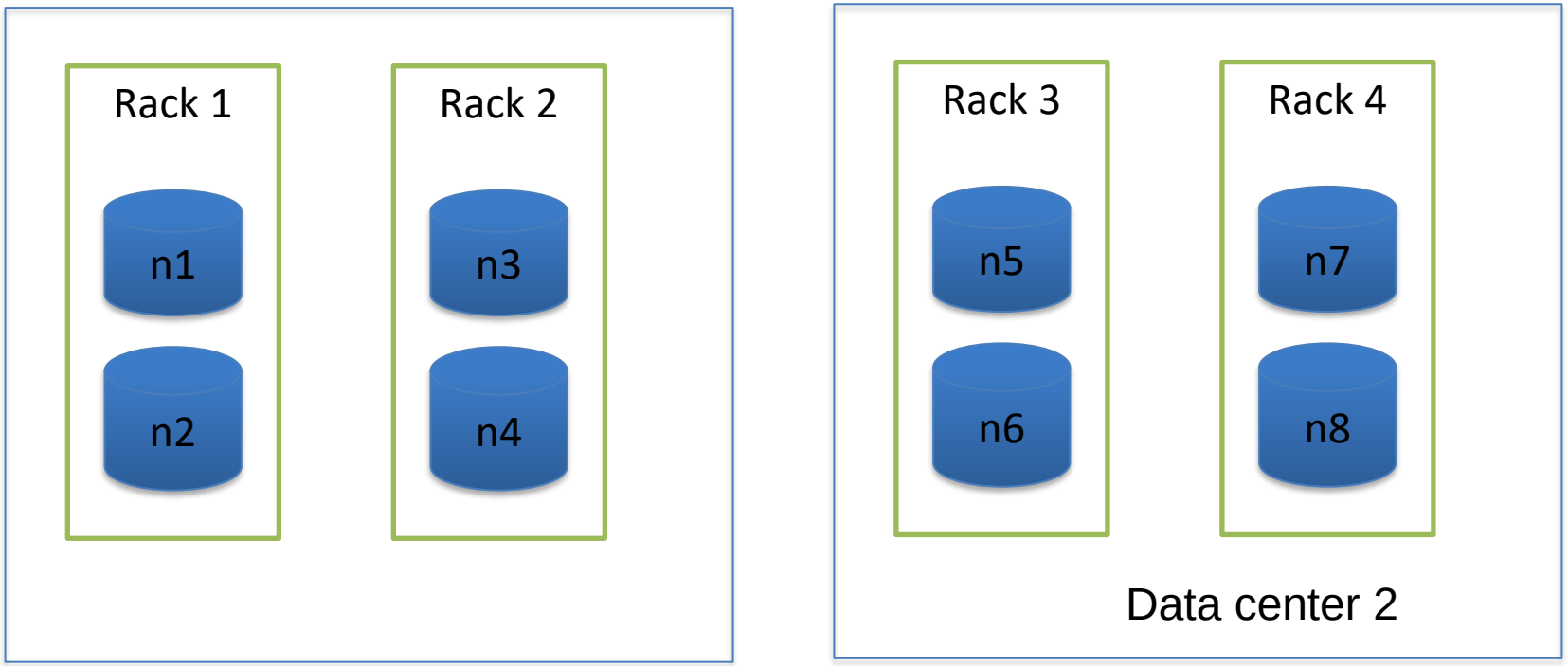
Bandwidth becomes less

HDFS Inside: Read

- Q: Given multiple replicas of the same block, how does NN decide which replica the client should read?
- HDFS Solution:
  - Rack awareness based on network topology

HDFS Inside: Network Topology

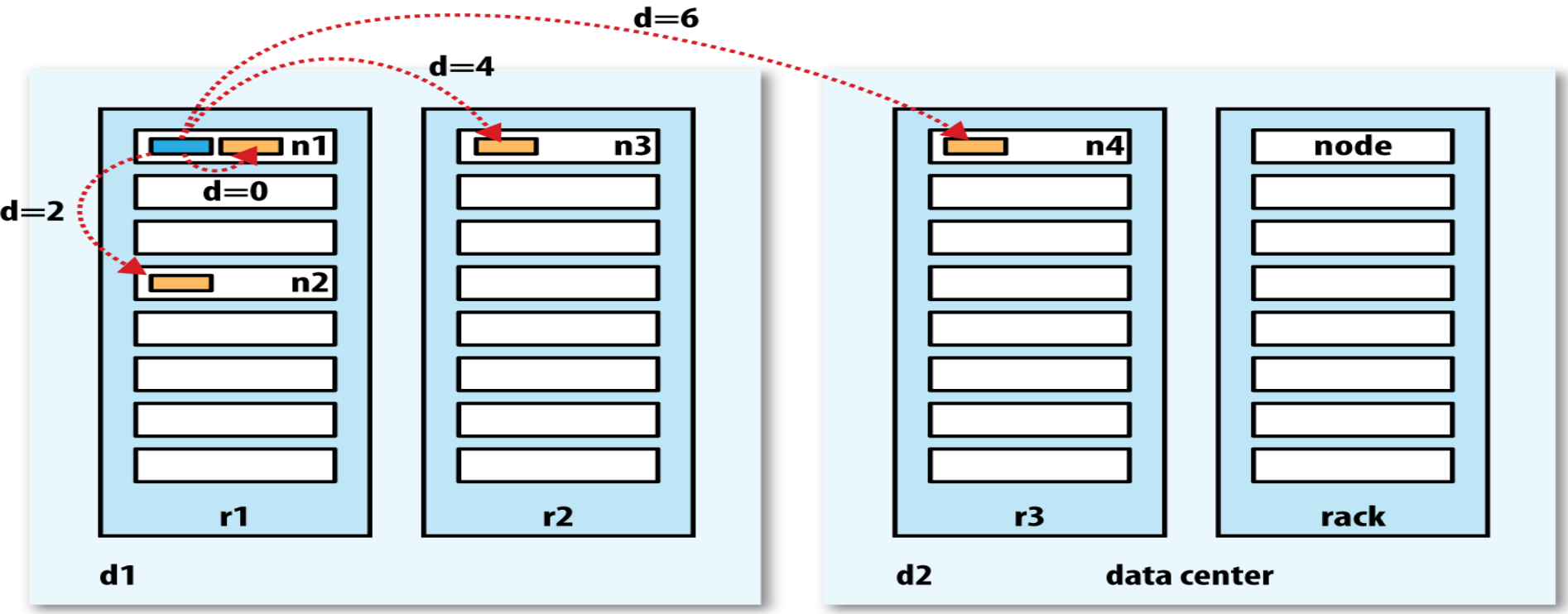
- HDFS takes a simple approach:
  - See the network as a tree
  - **Distance between two nodes is the sum of their distances to their closest common ancestor**



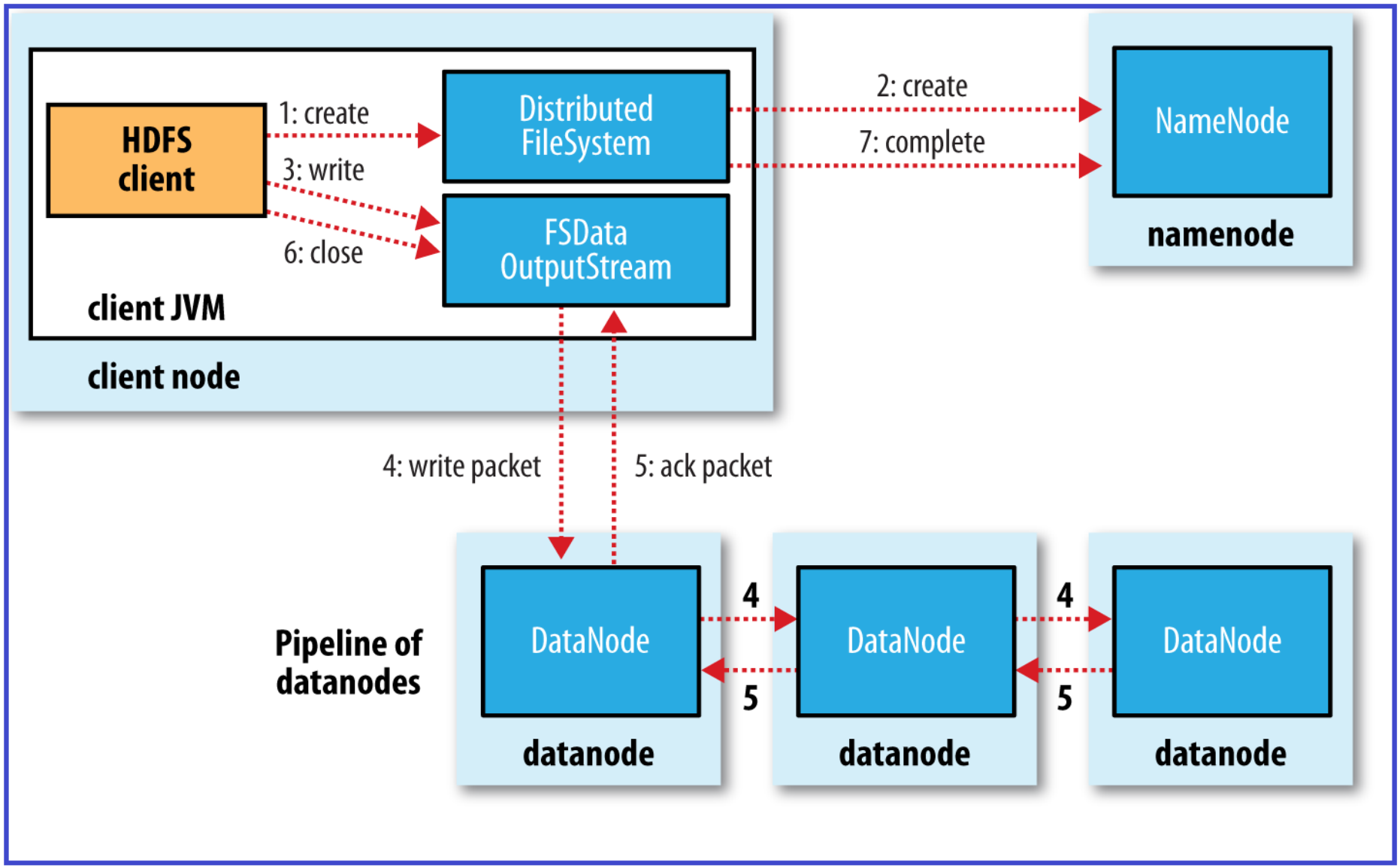
HDFS Inside: Network Topology

What are the distance of the following pairs:

- *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 diff. racks in same data center)
- *distance(/d1/r1/n1, /d2/r3/n4)* = 6 (nodes in different data centers)



Anatomy of a File Write



HDFS Inside: Write

- Q: Where should HDFS put the three replicas of a block? What tradeoffs we need to consider?
- Tradeoffs:
  - Reliability
  - Write Bandwidth
  - Read Bandwidth

Q: What are some possible strategies?

HDFS Inside: Write

- Replication Strategy vs Tradeoffs

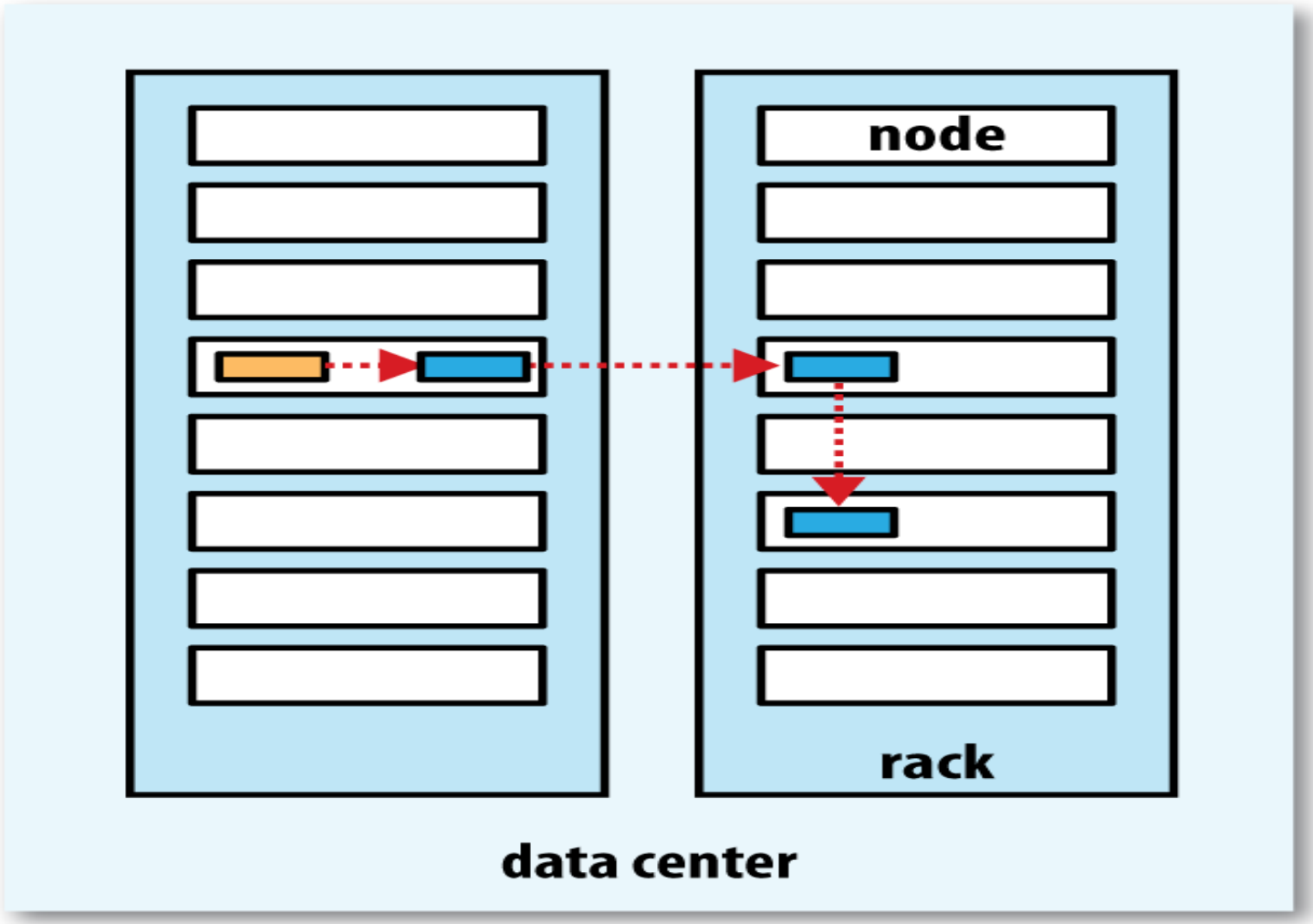
	Reliability	Write Bandwidth	Read Bandwidth
Put all replicas on one node			
Put all replicas on different racks			

HDFS Inside: Write

- **Replication Strategy vs Tradeoffs**

	Reliability	Write Bandwidth	Read Bandwidth
Put all replicas on one node			
Put all replicas on different racks			
HDFS: 1-> same node as client 2-> a node on different rack 3-> a different node on the same rack as 2			

HDFS Inside: Write



typical replica pipeline

Coherency Model

- A coherency model for a filesystem describes the data visibility of reads and writes for a file.
- After creating a file, it is visible in the filesystem namespace, as expected:

```
Path p = new Path("p");
fs.create(p);
assertThat(fs.exists(p), is(true));
```

- However, any content written to the file is not guaranteed to be visible, even if the stream is flushed. So, the file appears to have a length of zero:

```
Path p = new Path("p");
OutputStream out = fs.create(p);
out.write("content".getBytes("UTF-8"));
out.flush();
assertThat(fs.getFileStatus(p).getLen(), is(0L));
```

- Once more than a block's worth of data has been written, the first block will be visible to new readers.

Coherency Model

- HDFS provides a way to force all buffers to be flushed to the datanodes via the hflush() method on FSDataOutputStream.
- After hflush(), HDFS guarantees that data written up to that point in the file has reached all the datanodes in the write pipeline and is visible to all new readers.

```
Path p = new Path("p");
FSDataOutputStream out = fs.create(p);
out.write("content".getBytes("UTF-8"));
out.hflush();
assertThat(fs.getFileStatus(p).getLen(), is(((long) "content".length())));
```

- hflush() does not guarantee that datanodes have written the data to disk, only that it's in the datanodes' memory
- For this stronger guarantee, use hsync() instead.
- Closing a file in HDFS performs an implicit hflush()
- With no calls to hflush() or hsync(), may lose up to a block of data in the event of client or system failure
- Should call hflush() at suitable points

Parallel Copying with distcp

- Hadoop comes with a useful program called *distcp* for copying data to and from Hadoop filesystems in parallel.
- One use for *distcp* is as an efficient replacement for hadoop fs –cp
- For example, you can copy one file to another with:10

```
% hadoop distcp file1 file2
```

You can also copy directories:

```
% hadoop distcp dir1 dir2
```

- If *dir2* already exists, then *dir1* will be copied under it, creating the directory structure *dir2/dir1*
- *Can* supply the -overwrite option to keep the same directory structure
- Can use -update option to update files that have changed
- *distcp* is implemented as a MapReduce job

Parallel Copying with distcp

- Work of copying is done by the maps that run in parallel across the cluster.
- common use case for *distcp* is for transferring data between two HDFS clusters
- Following creates a backup of first cluster's */foo* directory on the second:

% hadoop distcp -update -delete -p hdfs://namenode1/foo hdfs://namenode2/foo

- -delete flag causes *distcp* to delete any files
- If the two clusters are running incompatible versions of HDFS, then you can use the webhdfs protocol to *distcp* between them:

% hadoop distcp webhdfs://namenode1:50070/foo webhdfs://namenode2:50070/foo

Keeping an HDFS Cluster Balanced

- When copying data into HDFS, it's important to consider cluster balance.
- HDFS works best when the file blocks are evenly spread across the cluster
- For example, if you specified -m 1, a single map would do the copy, which—apart from being slow and not using the cluster resources efficiently— would mean that the first replica of each block would reside on the node running the map (until the disk filled up).
- The second and third replicas would be spread across the cluster, but this one node would be unbalanced.
- By having more maps than nodes in the cluster, this problem is avoided.
- For this reason, it's best to start by running distcp with the default of 20 maps per node.
- not always possible to prevent a cluster from becoming unbalanced

MapReduce

What is MapReduce in Hadoop?

- **MapReduce** is a software framework and programming model used for processing huge amounts of data.
- **MapReduce** program work in two phases, namely, Map and Reduce.
- Map tasks deal with splitting and mapping of data while Reduce tasks shuffle and reduce the data.
- Hadoop is capable of running MapReduce programs written in various languages: Java, Ruby, Python, and C++.
- Map Reduce programs useful for performing large-scale data analysis using multiple machines in the cluster.

MapReduce Architecture

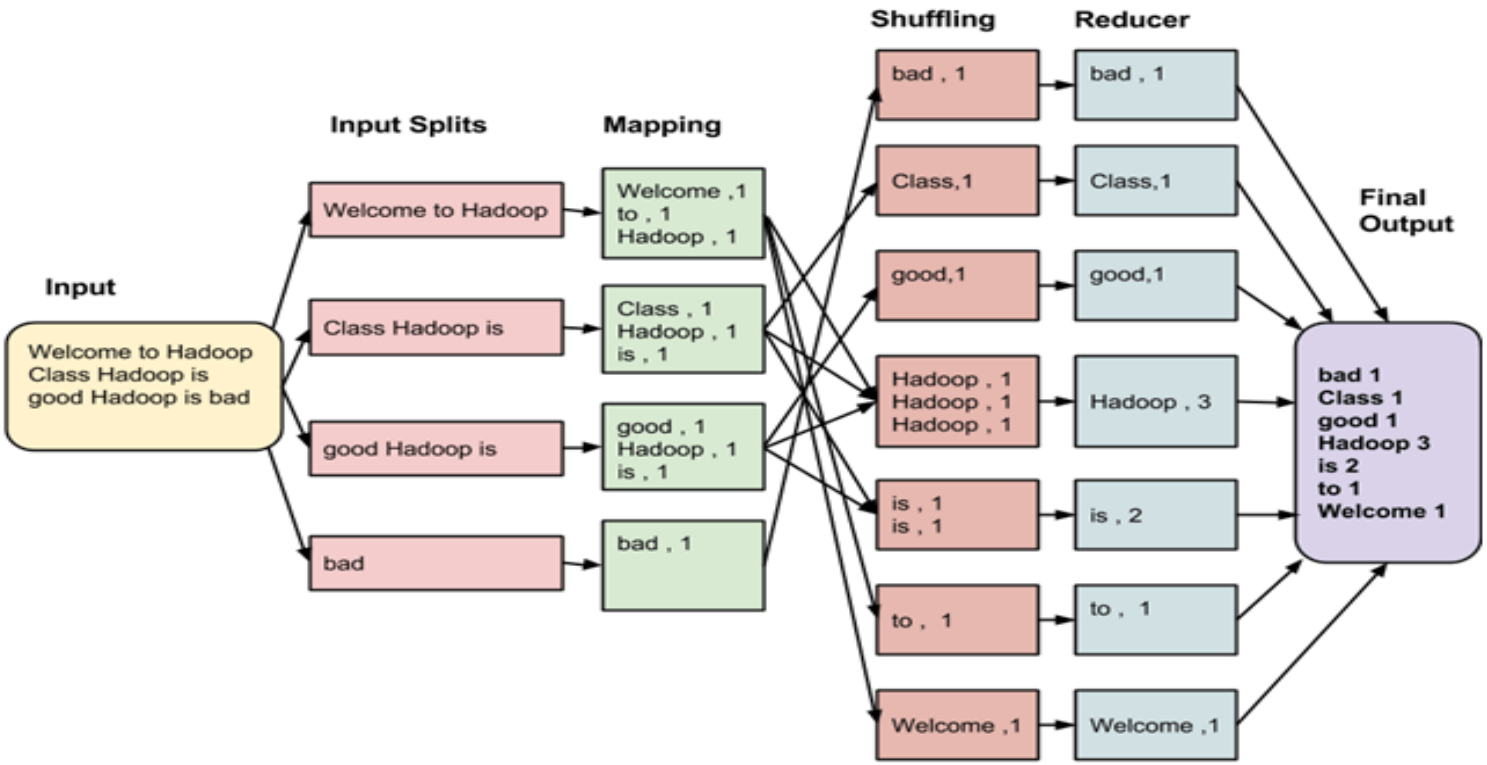
- The input to each phase is **key-value** pairs.
- Every programmer needs to specify two functions: **map function** and **reduce function**.
- The whole process goes through four phases of execution namely,

1. splitting,
2. mapping,
3. shuffling, and
4. reducing.

MapReduce Architecture

Consider you have following input data for your MapReduce in Big data Program

```
Welcome to Hadoop Class
Hadoop is good
Hadoop is bad
```



MapReduce Architecture

MapReduce

The final output of the MapReduce task is

bad	1
Class	1
good	1
Hadoop	3
is	2
to	1
Welcome	1

MapReduce Architecture explained in detail

- One map task is created for each split which then executes map function for each record in the split.
- It is always beneficial to have multiple splits.
- However, it is also not desirable to have splits too small in size.
- For most jobs, it is better to make a split size equal to the size of an HDFS block (which is 128 MB, by default).
- Execution of map tasks results into writing output to a local disk on the respective node and not to HDFS.
- Reason for choosing local disk over HDFS is,
  - to avoid replication which takes place in case of HDFS store operation.
  - Map output is intermediate output which is processed by reduce tasks to produce the final output.
  - Once job is complete, map output can be thrown away. So, storing it in HDFS with replication becomes overkill.

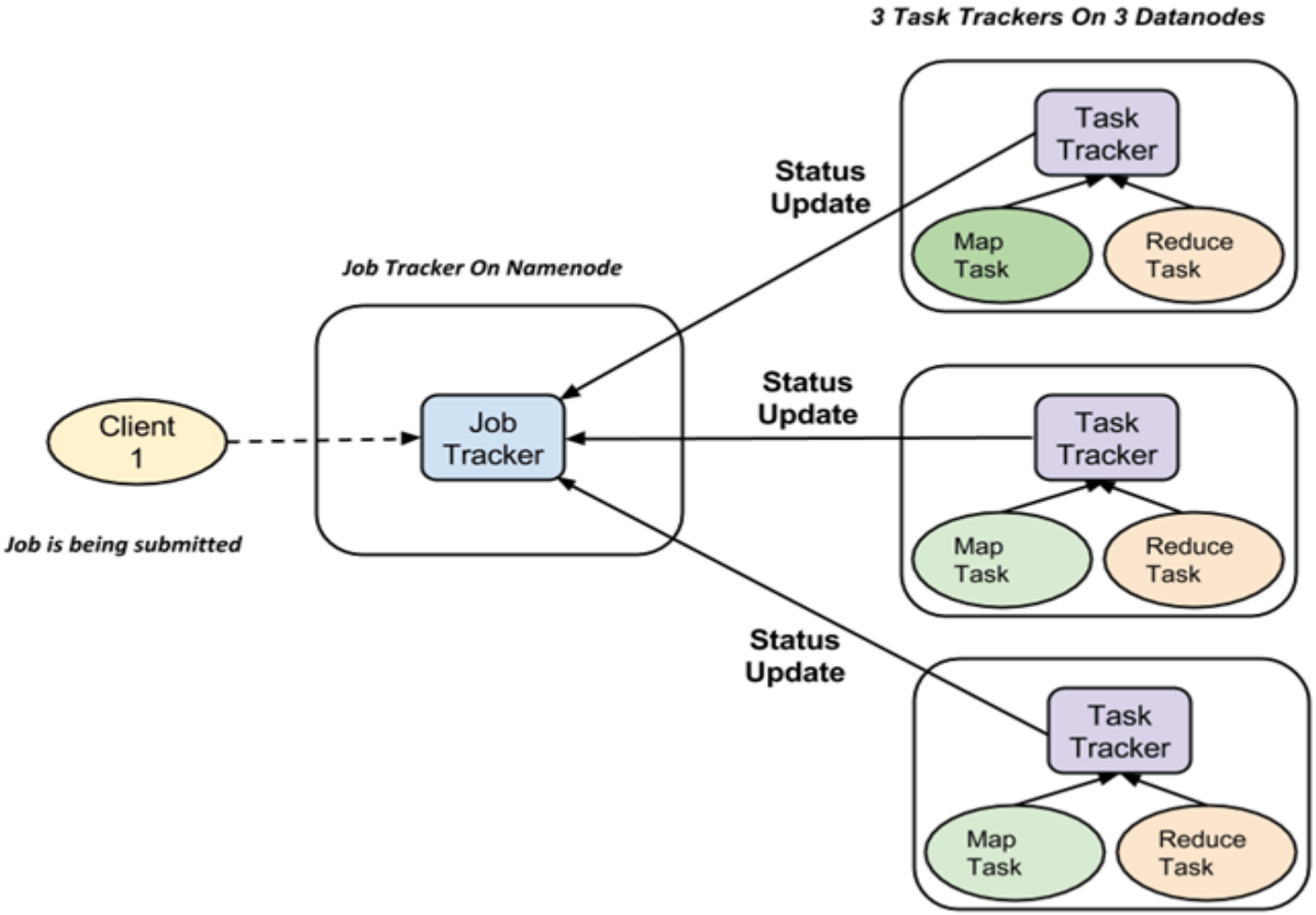
MapReduce Architecture explained in detail

- In the event of node failure, before the map output is consumed by the reduce task, Hadoop reruns the map task on another node and re-creates the map output.
- Reduce task doesn't work on the concept of data locality. An output of every map task is fed to the reduce task.
- Map output is transferred to the machine where reduce task is running.
- On this machine, output is merged and then passed to the user-defined reduce function.
- Unlike map output, reduce output is stored in HDFS (the first replica is stored on the local node and other replicas are stored on off-rack nodes).

How MapReduce Organizes Work?

- Hadoop divides the job into tasks. There are two types of tasks:
  1. **Map tasks** (Splits & Mapping)
  2. **Reduce tasks** (Shuffling, Reducing)
- The complete execution process (execution of Map and Reduce tasks, both) is controlled by two types of entities called a
  1. **Jobtracker**: Acts like a **master** (responsible for complete execution of submitted job)
  2. **Multiple Task Trackers**: Acts like **slaves,** each of them performing the job
- For every job submitted for execution in the system, there is one **Jobtracker** that resides on **Namenode** and there are **multiple tasktrackers** which reside on **Datanode**.

How MapReduce Organizes Work?



How MapReduce Organizes Work?

- A job is divided into multiple tasks which are then run onto multiple data nodes in a cluster.
- It is the responsibility of job tracker to coordinate the activity by scheduling tasks to run on different data nodes.
- Execution of individual task is then to look after by task tracker, which resides on every data node executing part of the job.
- Task tracker's responsibility is to send the progress report to the job tracker.
- Task tracker periodically sends **'heartbeat'** signal to the Jobtracker so as to notify him of the current state of the system.
- Thus job tracker keeps track of the overall progress of each job.
- In the event of task failure, job tracker can reschedule it on a different task tracker.

MapReduce

- Programming model for parallel data processing
- Hadoop can run MapReduce programs written in various languages:
  e.g. Java, Ruby, Python, C++
- MapReduce programs are inherently parallel,
- Can put very large-scale data analysis into the hands of anyone with enough machines at their disposal
- In this part of chapter
  - Introduce MapReduce programming using a simple example
  - Introduce some of MapReduce API
  - Explain data flow of MapReduce

MapReduce Example: Analysis of Weather Dataset

- Data from NCDC(National Climatic Data Center)
  - A large volume of log data collected by weather sensors: e.g. temperature
- Data format
  - Line-oriented ASCII format
  - Each record has many elements
  - We focus on the temperature element
  - Data files are organized by date and weather station
  - There is a directory for each year from 1901 to 2001, each containing a gzipped file for each weather station with its readings for that year
- **Query**
  - **What's the highest recorded global temperature for each year in the dataset?**

Year Temperature

```
006701199099999919500515070004...9999999N9+00001+99999999999...
004301199099999919500515120004...9999999N9+00221+99999999999...
004301199099999919500515180004...9999999N9-00111+99999999999...
004301265099999919490324120004...0500001N9+01111+99999999999...
004301265099999919490324180004...0500001N9+00781+99999999999...
```

Contents of data files

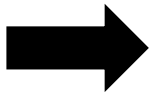
```
% ls raw/1990 | head
010010-99999-1990.gz
010014-99999-1990.gz
010015-99999-1990.gz
010016-99999-1990.gz
010017-99999-1990.gz
010030-99999-1990.gz
010040-99999-1990.gz
010080-99999-1990.gz
010100-99999-1990.gz
010150-99999-1990.gz
```

List of data files

Analyzing the Data with Unix Tools

- To provide a performance baseline
- Use *awk* for processing line-oriented data
- Complete run for the century took **42 minutes** on a single EC2 High-CPU Extra Large Instance

```
#!/usr/bin/env bash
for year in all/*
do
  echo -ne `basename $year .gz`"\t"
  gunzip -c $year | \
    awk '{ temp = substr($0, 88, 5) + 0;
           q = substr($0, 93, 1);
           if (temp !=9999 && q ~ /[01459]/ && temp > max) max = temp }
         END { print max }'
done
```



```
% ./max_temperature.sh
1901    317
1902    244
1903    289
1904    256
1905    283
...
```

How Can We Parallelize This Work?

- To speed up the processing, we need to run parts of the program in **parallel**
- **Dividing** the work
  - Process different years in different process
  - It is important to divide the work into even distribution
    - Split the input into fixed-size chunks
- **Combining** the results
  - If using the fixed-size chunks approach, the combination is more delicate
- But still we are limited by the processing capacity of a single machine
  - Some datasets grow beyond the capacity of a single machine
- To use **multiple machines**, we need to consider a variety of complex problems
  - Coordination: Who runs the overall job?
  - Reliability: How do we deal with failed processes?
- **Hadoop** can take care of these issues

Hadoop MapReduce

- To use MapReduce, we need to express out query as a MapReduce job
- MapReduce job
  - Map function
  - Reduce function
- Each function has key-value pairs as input and output
  - Types of input and output are chosen by the programmer

MapReduce Design of NCDC Example

- Map phase
  - Text input format of the dataset files
    - Key: offset of the line (unnecessary)
    - Value: each line of the files
  - Pull out the year and the temperature
    - Indeed in this example, the map phase is simply data preparation phase
    - Drop bad records(filtering)

Input File

```
0067011990999991950051507004...9999999N9+00001+99999999999...
0043011990999991950051512004...9999999N9+00221+99999999999...
0043011990999991950051518004...9999999N9-00111+99999999999...
0043012650999991949032412004...0500001N9+01111+99999999999...
0043012650999991949032418004...0500001N9+00781+99999999999...
```

Input of Map Function (key, value)

```
(0, 0067011990999991950051507004...9999999N9+00001+99999999999...)
(106, 0043011990999991950051512004...9999999N9+00221+99999999999...)
(212, 0043011990999991950051518004...9999999N9-00111+99999999999...)
(318, 0043012650999991949032412004...0500001N9+01111+99999999999...)
(424, 0043012650999991949032418004...0500001N9+00781+99999999999...)
```

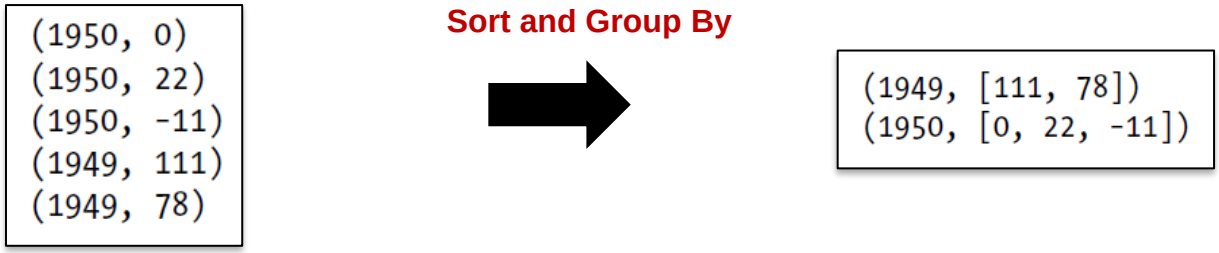
Map

Output of Map Function (key, value)

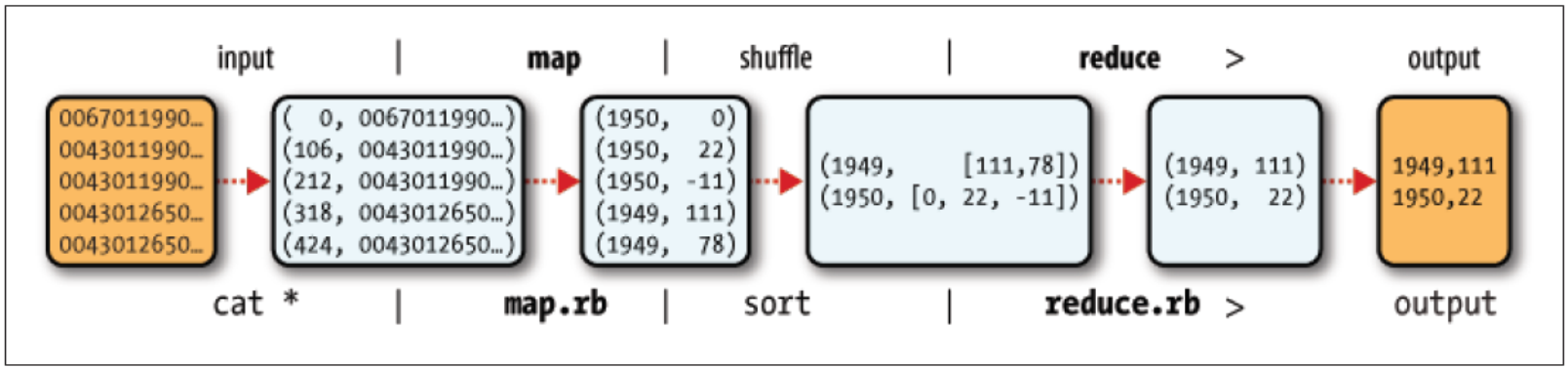
```
(1950, 0)
(1950, 22)
(1950, -11)
(1949, 111)
(1949, 78)
```

MapReduce Design of NCDC Example

- The output from the map function is processed by MapReduce framework
  - Sorts and groups the key-value pairs by key



- Reduce function iterates through the list and pick up the maximum value



Java Implementation: Map

- Map function: implementation of the Mapper interface
- Mapper interface
  - Generic type
  - Four type parameter: input key, input value, output key, output value type
- Hadoop provides its own set of basic types
  - optimized for network serialization
  - org.apache.hadoop.io package
  - e.g. LongWritable: Java Long
    Text: Java String
    IntWritable: Java Integer
- OutputCollector
  - Write the output

```
public class MaxTemperatureMapper extends MapReduceBase
  implements Mapper<LongWritable, Text, Text, IntWritable> {

  private static final int MISSING = 9999;

  public void map(LongWritable key, Text value,
      OutputCollector<Text, IntWritable> output, Reporter reporter)
      throws IOException {

    String line = value.toString();
    String year = line.substring(15, 19);
    int airTemperature;
    if (line.charAt(87) == '+') { // parseInt doesn't like leading plus signs
      airTemperature = Integer.parseInt(line.substring(88, 92));
    } else {
      airTemperature = Integer.parseInt(line.substring(87, 92));
    }
    String quality = line.substring(92, 93);
    if (airTemperature != MISSING && quality.matches("[01459]")) {
      output.collect(new Text(year), new IntWritable(airTemperature));
    }
  }
}
```

Input Type

Output Type

Java Implementation: Reduce

- Reduce function: implementation of the **Reducer** interface
- Reducer interface
  - Generic type
  - Four type parameter: input key, input value, output key, output value type
- Input types of the reduce function must match the output type of the map function

```
public class MaxTemperatureReducer extends MapReduceBase
  implements Reducer<Text, IntWritable, Text, IntWritable> {

  public void reduce(Text key, Iterator<IntWritable> values,
      OutputCollector<Text, IntWritable> output, Reporter reporter)
      throws IOException {

    int maxValue = Integer.MIN_VALUE;
    while (values.hasNext()) {
      maxValue = Math.max(maxValue, values.next().get());
    }
    output.collect(key, new IntWritable(maxValue));
  }
}
```

Input Type

Output Type

Java Implementation: Main

- **Construct JobConf object**
  - Specification of the job
  - Control how the job is run
  - Pass a class to the JobConf
    - Hadoop will locate the relevant JAR file and will distribute round the cluster
- **Specify input and output paths**
  - addInputPath(), setOutputPath()
    - If the output directory exists before running the job, Hadoop will complain and not run the job
- **Specify map and reduce types**
  - setMapperClass(), setReducerClass
- **Set output type**
  - setOutputKeyClass(), setOutputValueClass()
  - setMapOutputKeyClass(), setMapOutputValueClass()
- **Input type**
  - Here, we use the default, TextInputFormat
- **waitForCompletion()**
  - Submit the job

```
public class MaxTemperature {

  public static void main(String[] args) throws Exception {
    if (args.length != 2) {
      System.err.println("Usage: MaxTemperature <input path> <output path>");
      System.exit(-1);
    }

    Job job = new Job();
    job.setJarByClass(MaxTemperature.class);
    job.setJobName("Max temperature");

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.setMapperClass(MaxTemperatureMapper.class);
    job.setReducerClass(MaxTemperatureReducer.class);

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

Run the Job

- Install Hadoop in standalone mode (Appendix A in the book)
- Standalone mode
  - Run using the local filesystem with a local job runner
- HADOOP_CLASSPATH
  - Path of the application class

Output Log

```
% export HADOOP_CLASSPATH=build/classes
% hadoop MaxTemperature input/ncdc/sample.txt output
09/04/07 12:34:35 INFO jvm.JvmMetrics: Initializing JVM Metrics with processName=Job
Tracker, sessionId=
09/04/07 12:34:35 WARN mapred.JobClient: Use GenericOptionsParser for parsing the
arguments. Applications should implement Tool for the same.
09/04/07 12:34:35 WARN mapred.JobClient: No job jar file set.  User classes may not
be found. See JobConf(Class) or JobConf#setJar(String).
09/04/07 12:34:35 INFO mapred.FileInputFormat: Total input paths to process : 1
09/04/07 12:34:35 INFO mapred.JobClient: Running job: job_local_0001
09/04/07 12:34:35 INFO mapred.FileInputFormat: Total input paths to process : 1
09/04/07 12:34:35 INFO mapred.MapTask: numReduceTasks: 1
09/04/07 12:34:35 INFO mapred.MapTask: io.sort.mb = 100
09/04/07 12:34:35 INFO mapred.MapTask: data buffer = 79691776/99614720
09/04/07 12:34:35 INFO mapred.MapTask: record buffer = 262144/327680
09/04/07 12:34:35 INFO mapred.MapTask: Starting flush of map output
09/04/07 12:34:36 INFO mapred.MapTask: Finished spill 0
09/04/07 12:34:36 INFO mapred.TaskRunner: Task:attempt_local_0001_m_000000_0 is
done. And is in the process of commiting
09/04/07 12:34:36 INFO mapred.LocalJobRunner: file:/Users/tom/workspace/htdg/input/n
cdc/sample.txt:0+529
09/04/07 12:34:36 INFO mapred.TaskRunner: Task 'attempt_local_0001_m_000000_0' done.
```

Result

```
% cat output/part-00000
1949    111
1950    22
```

Output Log (cont.)

```
09/04/07 12:34:36 INFO mapred.LocalJobRunner:
09/04/07 12:34:36 INFO mapred.Merger: Merging 1 sorted segments
09/04/07 12:34:36 INFO mapred.Merger: Down to the last merge-pass, with 1 segments
left of total size: 57 bytes
09/04/07 12:34:36 INFO mapred.LocalJobRunner:
09/04/07 12:34:36 INFO mapred.TaskRunner: Task:attempt_local_0001_r_000000_0 is done
. And is in the process of commiting
09/04/07 12:34:36 INFO mapred.LocalJobRunner:
09/04/07 12:34:36 INFO mapred.TaskRunner: Task attempt_local_0001_r_000000_0 is
allowed to commit now
09/04/07 12:34:36 INFO mapred.FileOutputCommitter: Saved output of task
'attempt_local_0001_r_000000_0' to file:/Users/tom/workspace/htdg/output
09/04/07 12:34:36 INFO mapred.LocalJobRunner: reduce > reduce
09/04/07 12:34:36 INFO mapred.TaskRunner: Task 'attempt_local_0001_r_000000_0' done.
09/04/07 12:34:36 INFO mapred.JobClient:  map 100% reduce 100%
09/04/07 12:34:36 INFO mapred.JobClient: Job complete: job_local_0001
09/04/07 12:34:36 INFO mapred.JobClient: Counters: 13
09/04/07 12:34:36 INFO mapred.JobClient:   FileSystemCounters
09/04/07 12:34:36 INFO mapred.JobClient:     FILE_BYTES_READ=27571
09/04/07 12:34:36 INFO mapred.JobClient:     FILE_BYTES_WRITTEN=53907
09/04/07 12:34:36 INFO mapred.JobClient:   Map-Reduce Framework
09/04/07 12:34:36 INFO mapred.JobClient:     Reduce input groups=2
09/04/07 12:34:36 INFO mapred.JobClient:     Combine output records=0
09/04/07 12:34:36 INFO mapred.JobClient:     Map input records=5
09/04/07 12:34:36 INFO mapred.JobClient:     Reduce shuffle bytes=0
09/04/07 12:34:36 INFO mapred.JobClient:     Reduce output records=2
09/04/07 12:34:36 INFO mapred.JobClient:     Spilled Records=10
09/04/07 12:34:36 INFO mapred.JobClient:     Map output bytes=45
09/04/07 12:34:36 INFO mapred.JobClient:     Map input bytes=529
09/04/07 12:34:36 INFO mapred.JobClient:     Combine input records=0
09/04/07 12:34:36 INFO mapred.JobClient:     Map output records=5
09/04/07 12:34:36 INFO mapred.JobClient:     Reduce input records=5
```

Data Flow for Large Inputs

- To scale out, we need to store the data in a distributed filesystem, HDFS
- MapReduce job is divided into map tasks and reduce tasks
- Two types of nodes
  - Jobtracker
    - Coordinates all the jobs on the system by scheduling tasks to run on tasktrackers
    - If a task fails, the jobtracker can reschedule it on a different tasktracker
  - Tasktracker
    - Run tasks and send progress reports to the jobtracker
- Divides input into fixed-size pieces***, input splits***
  - Hadoop creates one map task for each split
  - Map task runs the user-defined map function for each *record* in the spilit

Data Flow for Large Inputs

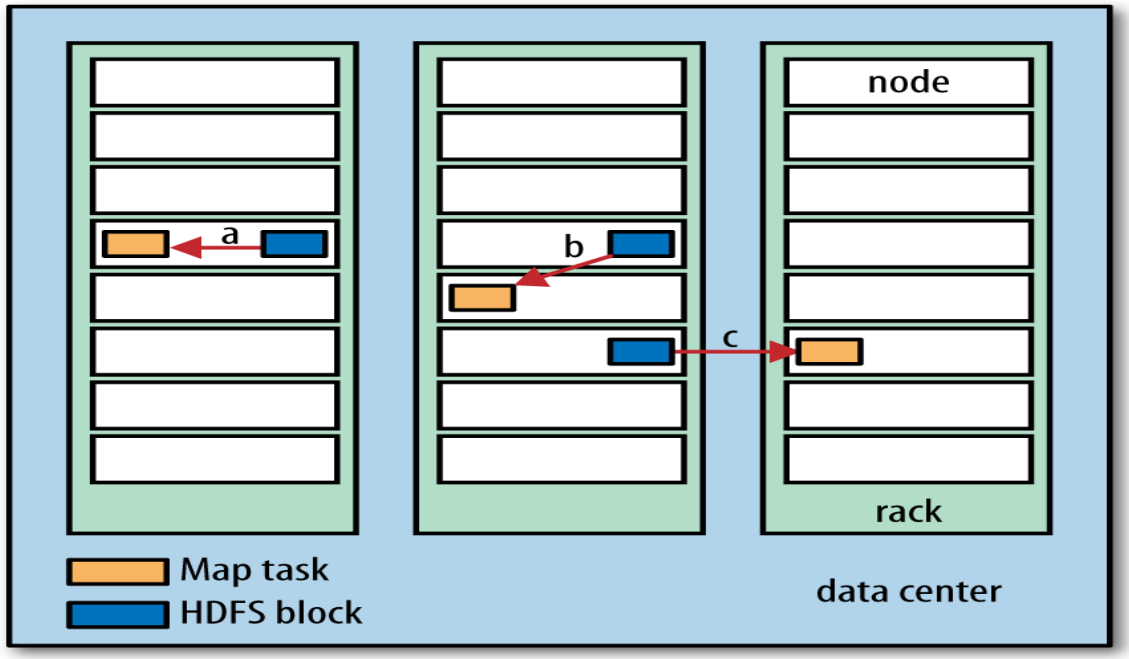
- **Size of splits**
  - Small size is better for load-balancing: faster machine will be able to process more splits
  - But if splits are too small, the overhead of managing the splits dominate the total execution time
  - For most jobs, a good split size tends to be the size of a HDFS block, 64MB(default)

- **Data locality optimization**
  - Run the map task on a node where the input data resides in HDFS
  - This is the reason why the split size is the same as the block size
    - The largest size of the input that can be guaranteed to be stored on a single node
    - If the split spanned two blocks, it would be unlikely that any HDFS node stored both blocks

- **Map tasks write their output to local disk (not to HDFS)**
  - Map output is intermediate output
  - Once the job is complete the map output can be thrown away
  - So storing it in HDFS with replication, would be overkill
  - If the node of map task fails, Hadoop will automatically rerun the map task on another node

Data Flow for Large Inputs

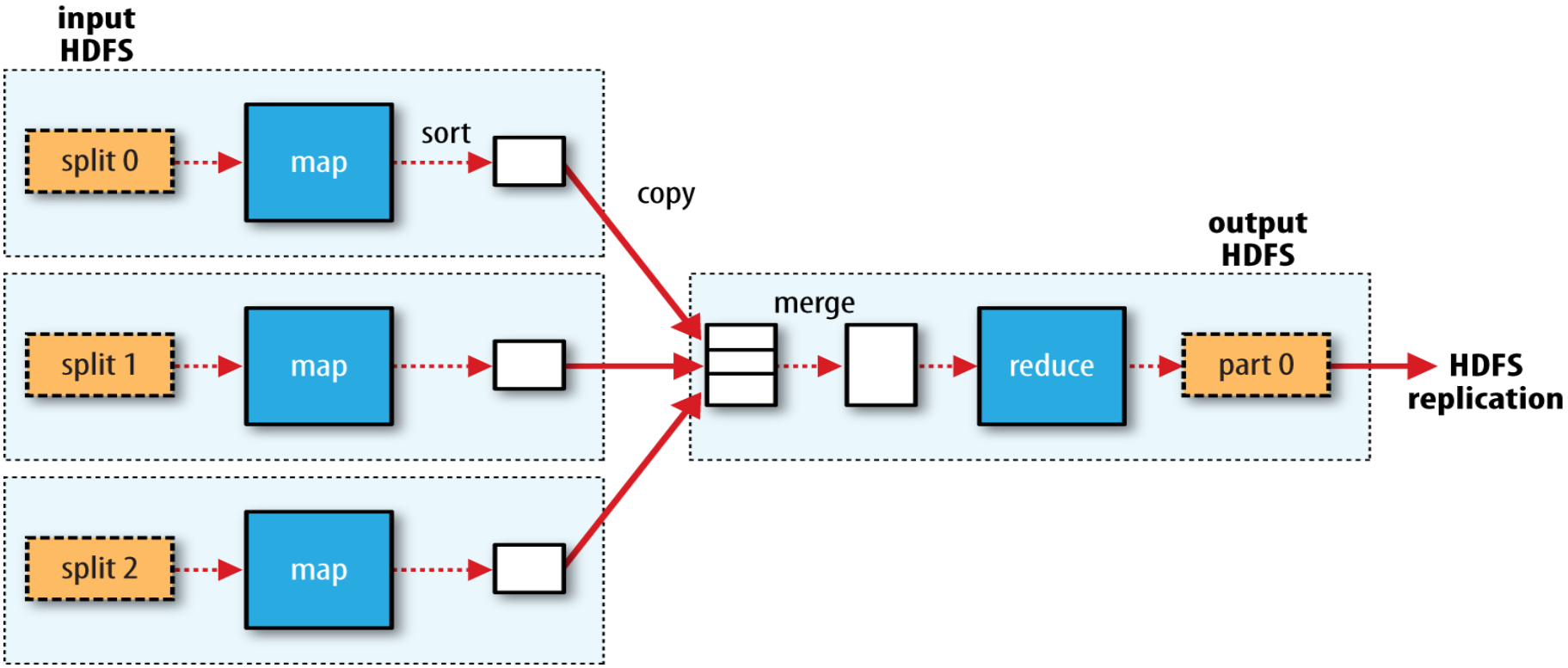
- Hadoop runs the map task on a node where the input data resides in HDFS, because it doesn't use valuable cluster bandwidth.
- This is called the ***data locality optimization***.
- Sometimes, all nodes hosting the HDFS block replicas for a map task's input split are running other map tasks - look for a free map slot on a node in same rack
- Very occasionally even this is not possible, so an off-rack node is used, which results in an inter-rack network transfer.



Data-local (a), rack-local (b), and off-rack (c) map tasks

MapReduce data flow with a single reduce task

- Reduce tasks don't have the advantage of data locality
- Input to a single reduce task is normally the output from all mappers.
- In present example, we have a single reduce task that is fed by all of map tasks.

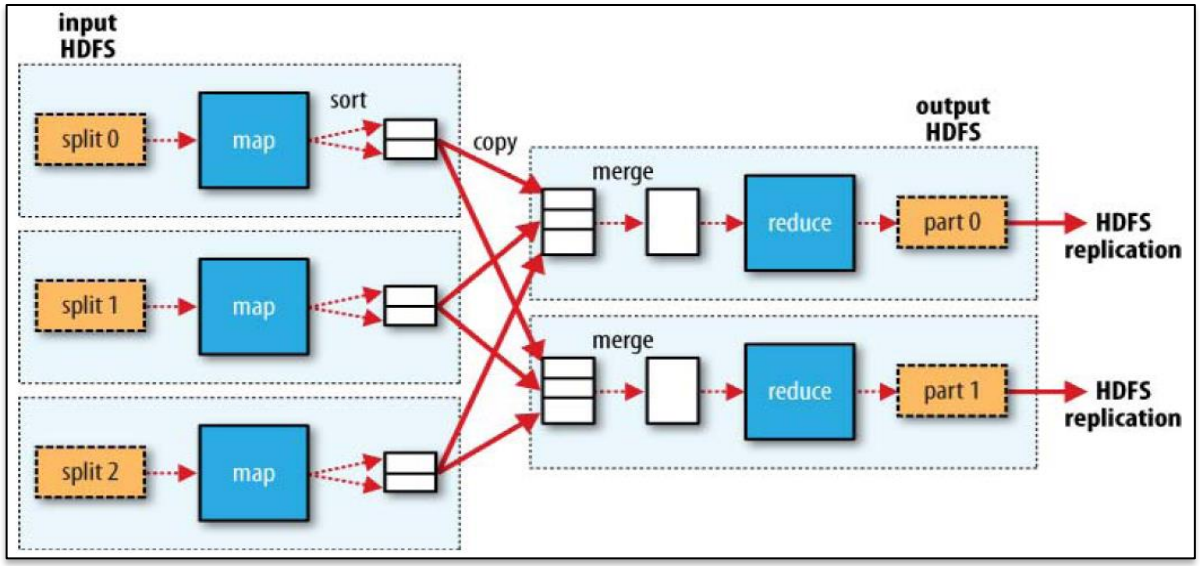


Data Flow for Large Inputs

- **Reduce tasks don't have the advantage of data locality**
  - Input to a single reduce task is normally the output from all mappers
  - Output of the reduce is stored in HDFS for reliability

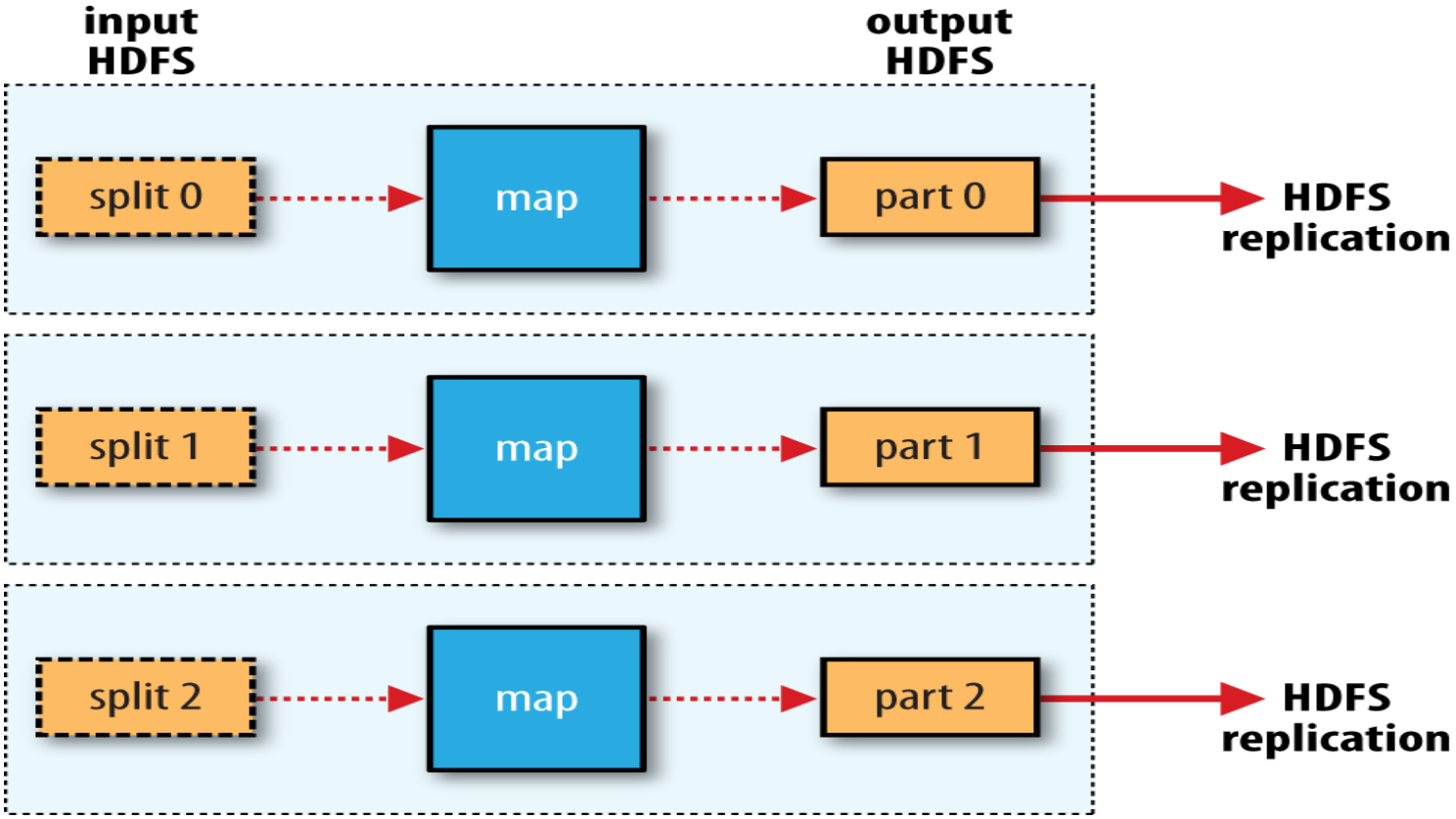
- **The number of reduce tasks is not governed by the size of the input, but is specified independently**

- **When there are multiple reducers, the map tasks partition their output:**
  - One partition for each reduce task
  - The records for every key are all in a single partition
  - Partitioning can be controlled by a user-defined partitioning function



Data Flow for Large Inputs

- Finally, it's also possible to have zero reduce tasks.
- This can be appropriate when you don't need the shuffle because the processing can be carried out entirely in parallel
- In this case, only off-node data transfer is when map tasks write to HDFS



Combiner Function

- Many MapReduce jobs are limited by the bandwidth available on cluster, so it pays to minimize the data transferred between map and reduce tasks
- Hadoop allows the user to specify a combiner function to be run on the map output, and the combiner function's output forms the input to the reduce function.
- Combiner function is run on the map output
- But Hadoop do not guarantee how many times it will call combiner function for a particular map output record
  - It is just optimization
  - The number of calling (even zero) does not affect the output of Reducers

max(0, 20, 10, 25, 15) = max(max(0, 20, 10), max(25, 15)) = max(20, 25) = 25

first map produced output:

```
(1950, 0)
(1950, 20)
(1950, 10)
```

the second produced:

```
(1950, 25)
(1950, 15)
```

The reduce function would be called with a list of all the values:

`(1950, [0, 20, 10, 25, 15])` with output: `(1950, 25)`

combiner function just like the reduce function, finds maximum temperature for each map output

Running a distributed MapReduce job 10-node EC2 cluster running High-CPU Extra Large Instances: 6 minutes

Specifying a combiner function

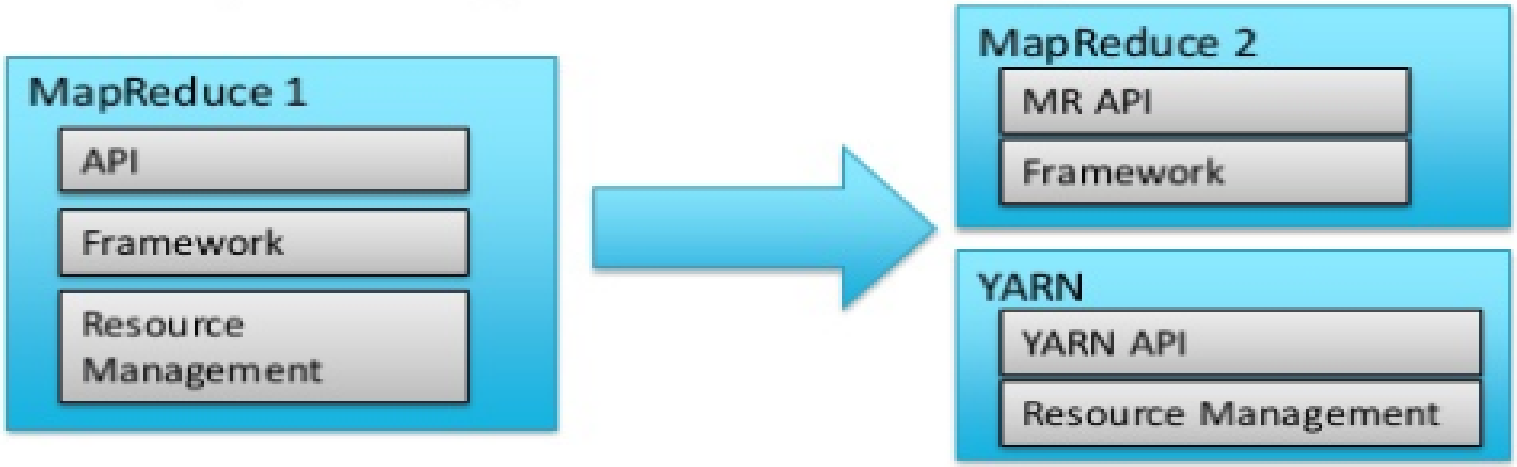
- Combiner function is defined using the Reducer class,
- For temperature application, it is same implementation as the reduce function in MaxTemperatureReducer.
- The only change we need to make is to set the combiner class on the Job

```
public class MaxTemperatureWithCombiner {

  public static void main(String[] args) throws Exception {
    if (args.length != 2) {
      System.err.println("Usage: MaxTemperatureWithCombiner <input path> " +
          "<output path>");
      System.exit(-1);
    }

    Job job = new Job();
    job.setJarByClass(MaxTemperatureWithCombiner.class);
    job.setJobName("Max temperature");

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.setMapperClass(MaxTemperatureMapper.class);
    job.setCombinerClass(MaxTemperatureReducer.class);
    job.setReducerClass(MaxTemperatureReducer.class);

    job.setOutputKeyClass(Text.class);
```

Mapreduce 1 vs Mapreduce 2

MRv1 and MRv2

- **MapReduce 1 ("Classic") has three main components**
  - API – for user-level programming of MR applications
  - Framework – runtime services for running Map and Reduce processes, shuffling and sorting, etc.
  - Resource management – infrastructure to monitor nodes, allocate resources, and schedule jobs
- **MapReduce 2 ("NextGen") moves Resource Management into YARN**



Mapreduce 1 vs Mapreduce 2

MapReduce2 History

- **Originally architected at Yahoo in 2008**
- **"Alpha" in Hadoop 2 pre-GA**
  - Included in CDH 4
- **YARN promoted to Apache Hadoop sub-project**
  - summer 2013
- **"Production ready" in Hadoop 2 GA**
  - Included in CDH5 (Beta in Oct 2013)

Mapreduce 1 vs Mapreduce 2

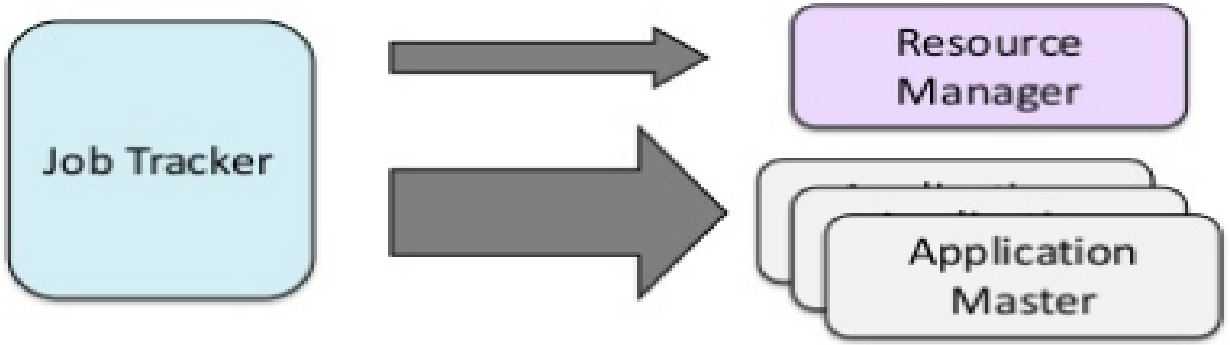
Why is YARN needed?

- **MRv1 resource management issues**
  - Inflexible "slots" configured on nodes – Map or Reduce, not both
    - Underutilization of cluster when more map or reduce tasks are running
  - Can't share resources with non-MR applications running on Hadoop cluster (e.g. Impala, Giraph)
  - Scalability – one JobTracker per cluster – limit of about 4000 nodes per cluster

Mapreduce 1 vs Mapreduce 2

Why is YARN needed?

- **YARN solutions**
  - No slots
    - Nodes have "resources" – memory and CPU cores – which are allocated to applications when requested
  - Supports MR and non-MR applications running on the same cluster
  - Most Job Tracker functions moved to Application Master – one cluster can have many Application Masters



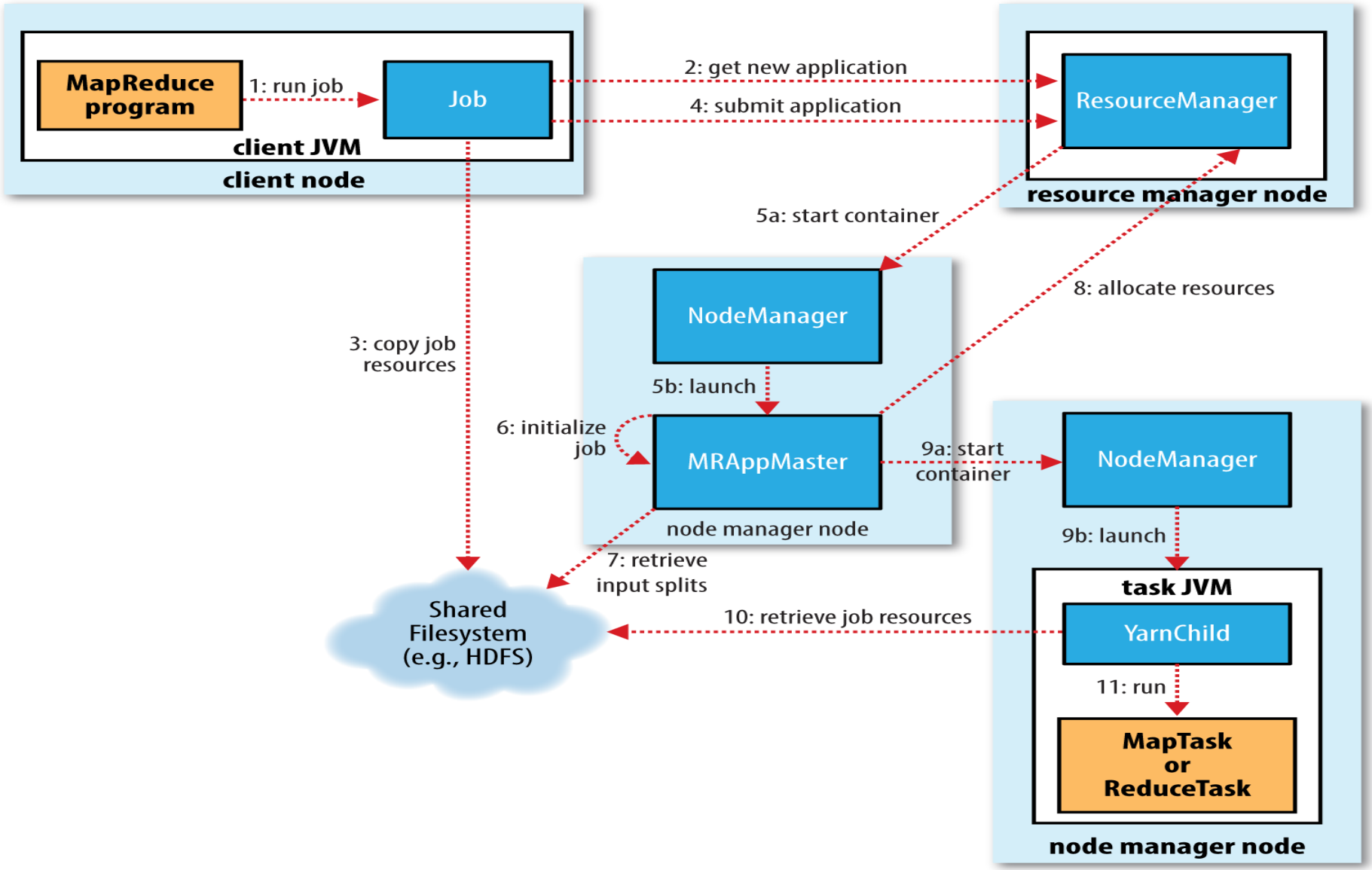
Anatomy of a MapReduce Job Run

- Can run a MapReduce job with a single method call: submit()
- Can also call waitForCompletion(),
- Conceals a great deal of processing behind the scenes
- At the highest level, there are five independent entities:
  - 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 Map-Reduce job.
  - The distributed filesystem (normally HDFS), which is used for sharing job files between the other entities.

Anatomy of a MapReduce Job Run

- YARN provides its core services via two types of long-running daemon:
  - a *resource manager* (one per cluster) to manage the use of resources across the cluster, and
  - *Node managers* running on all the nodes in the cluster to launch and monitor *containers*.
- A *container* executes an application-specific process with a constrained set of resources (memory, CPU, and so on).

Anatomy of a MapReduce Job Run



Anatomy of a MapReduce Job Run

Job Submission

- The submit() method on Job creates an internal JobSubmitter instance and calls submitJobInternal() on it.
- Having submitted the job, waitForCompletion() polls the job's progress once per second.
- When the job completes successfully, the job counters are displayed
- The job submission process implemented by JobSubmitter does the following
  - Asks the resource manager for a new application ID, used for the MapReduce job ID (step 2).
  - Checks the output specification of the job.
  - Computes the input splits for the job.
  - Copies the resources needed to run the job, including the job JAR file, the configuration file, and the computed input splits, to the shared filesystem in a directory named after the job ID
  - Submits job by calling submitApplication() on resource manager (step 4).

Anatomy of a MapReduce Job Run

Job Initialization

- When the resource manager receives a call to its submitApplication() method, it hands off the request to the YARN scheduler.
- The scheduler allocates a container, and the resource manager then launches the application master's process there, under the node manager's management (steps 5a and 5b).
- 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 (step 6).
- Next, it retrieves the input splits computed in the client from the shared filesystem (step 7).
- It then creates a map task object for each split, as well as a number of reduce task objects determined by the mapreduce.job.reduces property (set by the setNumReduceTasks() method on Job).
- Tasks are given IDs at this point.
- Application master calls the setupJob() method on the OutputCommitter

Anatomy of a MapReduce Job Run

Task Assignment

- Application master requests containers for all the map and reduce tasks in the job from the resource manager (step 8).
- Requests for map tasks are made first and with a higher priority than those for reduce tasks
- Requests for reduce tasks are not made until 5% of map tasks have completed
- Reduce tasks can run anywhere in the cluster, but requests for map tasks have data locality constraints that the scheduler tries to honor
- Requests also specify memory requirements and CPUs for tasks.
- By default, each map and reduce task is allocated 1,024 MB of memory and one virtual core

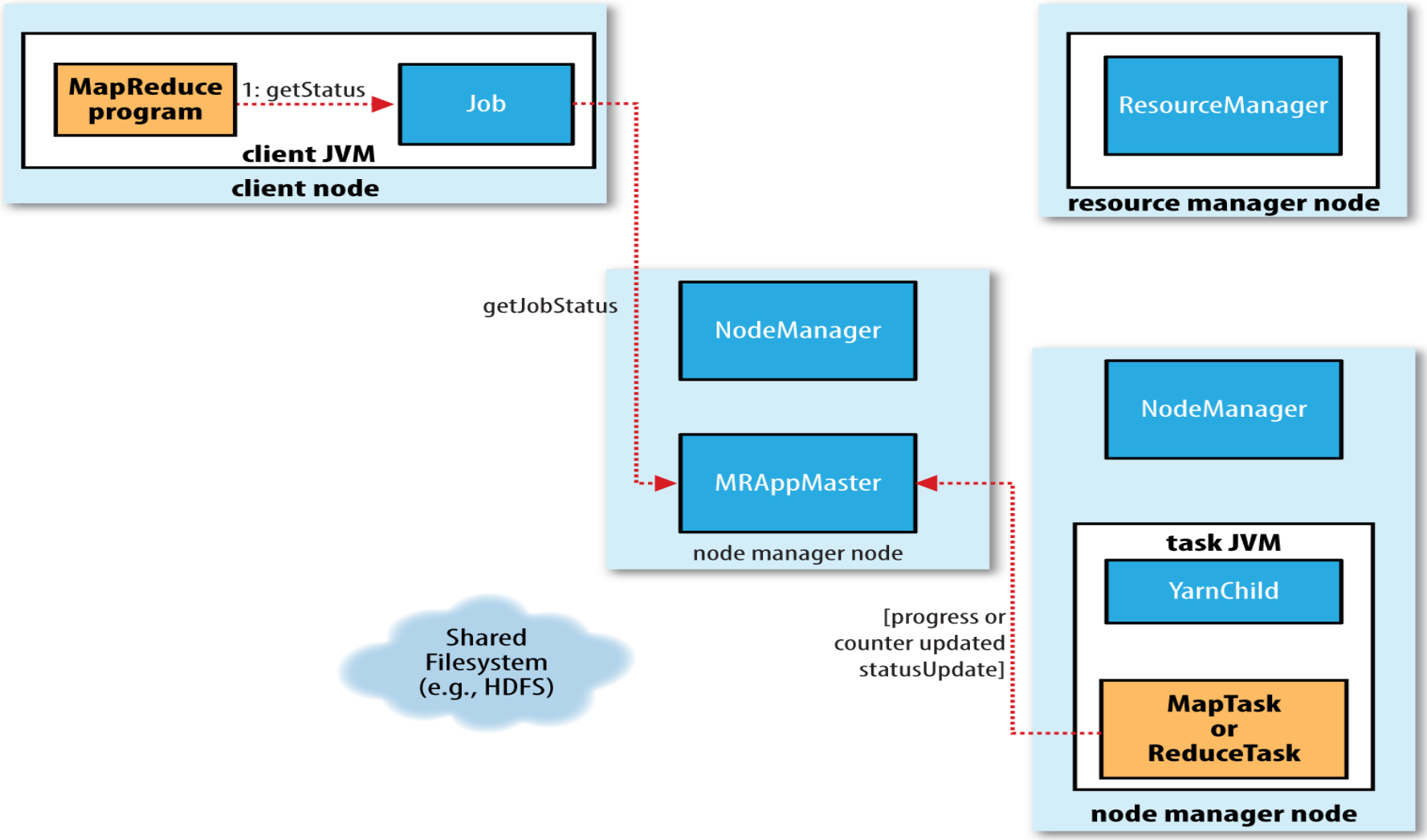
Anatomy of a MapReduce Job Run

Task Execution

- Once a task has been assigned resources for a container on a particular node by the resource manager's scheduler, application master starts container by contacting the node manager (steps 9a and 9b).
- 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 (step 10).
- Finally, it runs the map or reduce task (step 11).
- The YarnChild runs in a dedicated JVM, so that any bugs in the user-defined map and reduce functions (or even in YarnChild) don't affect the node manager—by causing it to crash or hang, for example.

Anatomy of a MapReduce Job Run

How status updates are propagated through MapReduce system



Failures

- Task Failure
- Application Master Failure
- Node Manager Failure
- Resource Manager Failure

Failures - Task Failure

- User code in the map or reduce task throws a runtime exception.
- If this happens, task JVM reports error back to its parent application master
- Application master marks task attempt as *failed*, and frees up container so its resources are available for another task.
- Sudden exit of the task JVM— due to JVM bug that causes the JVM to exit for a particular set of circumstances exposed by MapReduce user code
- Hanging tasks – application master notices that it hasn't received a progress update for a while and proceeds to mark task as failed
- When the application master is notified of a task attempt that has failed, it will reschedule execution of the task.
- For some applications, it is undesirable to abort the job if a few tasks fail
- A task attempt may also be *killed*
  - speculative duplicate
  - the node manager it was running on Failed
  - application master marked all the task attempts running on it as killed
- Users may also kill or fail task

Failures - Application Master Failure

- Like MapReduce tasks, applications in YARN are retried in the event of failure.
- The maximum number of attempts to run a MapReduce application master is controlled by the mapreduce.am.max-attempts property.
- The default value is 2
- YARN imposes a limit for the maximum number of attempts for any YARN application master running on the cluster, and individual applications may not exceed this limit.
- The way recovery works is as follows.
  - An application master sends periodic heartbeats to the resource manager
  - 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).
- If the application master fails, however, 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. This process is transparent to the user.

Failures - Node Manager Failure

- If a node manager fails by crashing or running very slowly, it will stop sending heartbeats to the resource manager
- The resource manager will notice a node manager that has stopped sending heartbeats if it hasn't received one for 10 minutes
- Any task or application master running on the failed node manager will be recovered
- In addition, application master arranges for map tasks that were run and completed successfully on the failed node manager to be rerun if they belong to incomplete jobs

Failures - Resource Manager Failure

- Failure of the resource manager is serious, because without it, neither jobs nor task containers can be launched.
- To achieve high availability (HA), it is necessary to run a pair of resource managers in an active-standby configuration.
- Information about all the running applications is stored in a highly available state store
- When the new resource manager starts, it reads the application information from the state store, then restarts the application masters for all the applications running on the cluster.
- The transition of a resource manager from standby to active is handled by a failover controller.
- The default failover controller is an automatic one, which uses ZooKeeper leader election to ensure that there is only a single active resource manager at one time.
- Clients and node managers must be configured to handle resource manager failover.

Thank You !!!

MapReduce Program – Weather Data Analysis For Analyzing Hot And Cold Days - GOOD

https://www.geeksforgeeks.org/mapreduce-program-weather-data-analysis-for-analyzing-hot-and-cold-days/

https://www.geeksforgeeks.org/hadoop-yarn-architecture/

https://www.geeksforgeeks.org/hadoop-ecosystem/

Hadoop offers

- Redundant, Fault-tolerant data storage
- Parallel computation framework
- Job coordination

Hadoop offers

- Redundant, Fault-tolerant data storage
- Parallel computation framework
- Job coordination

Programmers

No longer need to worry about

Q: Where file is located?

Q: How to handle failures & data lost?

Q: How to divide computation?

Q: How to program for scaling?