A Brief History of Apache Hadoop:

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.

The name Hadoop is not an acronym; it's a made-up name. The project's creator, Doug Cutting, explains how the name came about:

The name my kid gave a **yellow elephant**.

Projects in the Hadoop ecosystem also tend to have names that are unrelated to their function, often with an elephant or other animal theme ("Pig," for example).

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 one-billion-page index would cost around $500,000 in hardware, with a monthly running cost of $30,000.

Nutch was started in 2002, and a working crawler and search system quickly emerged.

However, its creators 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.

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, Nutch's developers set about writing an open source implementation, the Nutch Distributed Filesystem (NDFS).

In 2004, Google published the paper that introduced **MapReduce** to the world.

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.

At around the same time, **Doug Cutting** joined Yahoo!, which provided a dedicated team and the resources to turn Hadoop into a system that ran at web scale .

This was demonstrated in February 2008 when Yahoo! announced that its production search index was being generated by a 10,000-core Hadoop cluster.

Building Internet-scale search engines requires huge amounts of data and therefore large numbers of machines to process it.

Yahoo! Search consists of four primary components:

the ***Crawler***, which downloads pages from web servers;
the ***WebMap***, which builds a graph of the known Web;
the ***Indexer***, which builds a reverse index to the best pages;
and the ***Runtime***, which answers users' queries.

The WebMap is a graph that consists of roughly 1 trillion edges, each representing a web link, and 100 billion nodes, each representing distinct URLs.

Creating and analyzing such a large graph requires a large number of computers running for many days.

In **January 2008**, Hadoop was made its own top-level project at Apache, confirming its success and its diverse, active community.

By this time, Hadoop was being used by many other companies besides **Yahoo**!, such as **Last.fm**, **Facebook**, and the ***New York Times***.

In one well-publicized feat, the ***New York Times*** used **Amazon's EC2** compute cloud to crunch through 4 terabytes of scanned archives from the paper, converting them to PDFs for the Web. The processing took less than 24 hours to run using 100 machines,

In **April 2008**, Hadoop broke a world record to become the fastest system to sort an entire **terabyte of data**. Running on a **910-node cluster**, Hadoop sorted **1 terabyte in 209 seconds** (just under 3.5 minutes), beating the previous year's winner of **297 seconds**.

In **November of the same year**, Google reported that its MapReduce implementation sorted **1 terabyte in 68 seconds**. Then, in **April 2009**, it was announced that a team at **Yahoo!** had used Hadoop to sort **1 terabyte in 62 seconds**.

The trend since then has been to sort even larger volumes of data at ever faster rates.

In **the 2014** competition, a team from **Databricks** were joint winners of the Gray Sort benchmark. They used a **207-node Spark cluster to sort 100 terabytes** of data in **1,406 seconds,** a rate of 4.27 terabytes per minute.

Today, Hadoop is widely used in mainstream enterprises. Hadoop's role as a general purpose storage and analysis platform for big data has been recognized by the industry, and this fact is reflected in the number of products that use or incorporate Hadoop in some way.

Commercial Hadoop support is available from large, established enterprise vendors, including EMC, IBM, Microsoft, and Oracle, as well as from specialist Hadoop companies such as **Cloudera**, **Hortonworks**, and **MapR**.

A Weather Dataset:

- National Climatic Data Center, NCDC
- Weather sensors collect data every hour
- The data is stored using a line-oriented ASCII format, in which each line is a record.

Format of a National Climatic Data Center:

```
0057
332130    # USAF weather station identifier
99999     # WBAN weather station identifier
19500101  # observation date
0300      # observation time
4
+51317    # latitude (degrees x 1000)
+028783   # longitude (degrees x 1000)
FM-12
+0171     # elevation (meters)
99999
V020
320       # wind direction (degrees)
1         # quality code
N
0072
1
00450     # sky ceiling height (meters)
1         # quality code
C
N
010000    # visibility distance (meters)
1         # quality code
N
9
-0128     # air temperature (degrees Celsius x 10)
1         # quality code
-0139     # dew point temperature (degrees Celsius x 10)
1         # quality code
10268     # atmospheric pressure (hectopascals x 10)
1         # quality code
```

Analyzing the Data with Hadoop:

- MapReduce works by breaking the processing into two phases: the map phase and the reduce phase
- The programmer also specifies two functions: the map function and the reduce function
- text input format :
  - each line in the dataset as a text value
  - key is the offset of the beginning of the line from the beginning of the file

finding the maximum temperature for each year:

- ❑ Our map function is simple
- ❑ pull out the year and the air temperature
- ❑ map function merely extracts the year and the air temperature

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

These lines are presented to the map function as the key-value pairs:

```
(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...)
```

The output from the map function:

(1950, 0)

(1950, 22)

(1950, −11)

(1949, 111)

(1949, 78)

This processing sorts and groups the key-value pairs by key:

(1949, [111, 78])

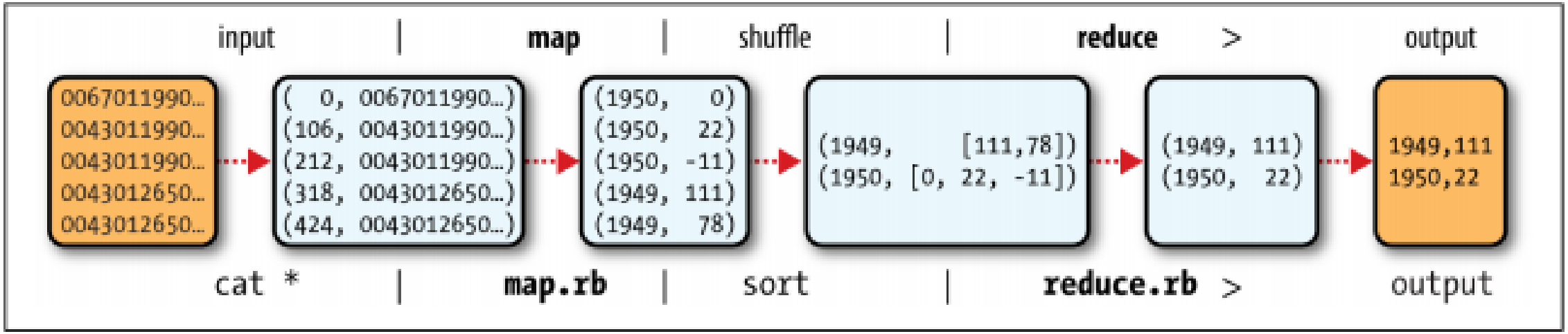
(1950, [0, 22, −11])

pick up the maximum:

(1949, 111)

(1950, 22)

MapReduce logical data flow:



Mapper for the maximum temperature example:

```
import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;

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

  private static final int MISSING = 9999;

  @Override
  public void map(LongWritable key, Text value, Context context)
      throws IOException, InterruptedException {

    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]")) {
      context.write(new Text(year), new IntWritable(airTemperature));
    }
  }
}
```

Mapper class in the example:

- four type parameters
  - input key : long integer offset
  - input value : a line of text
  - output key : year
  - output value : air temperature

Reducer for the maximum temperature example:

```
import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;

public class MaxTemperatureReducer
    extends Reducer<Text, IntWritable, Text, IntWritable> {

  @Override
  public void reduce(Text key, Iterable<IntWritable> values, Context context)
      throws IOException, InterruptedException {

    int maxValue = Integer.MIN_VALUE;
    for (IntWritable value : values) {
      maxValue = Math.max(maxValue, value.get());
    }
    context.write(key, new IntWritable(maxValue));
  }
}
```

Reducer class in the example:

- four formal type parameters
- The input types of the reduce function must match the output types of the map function (Text and IntWritable)
- Output types of the reduce function are Text and IntWritable

The third piece of code runs the MapReduce job :

```
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

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);
  }
}
```

Scale out vs Scale up

Data Flow:

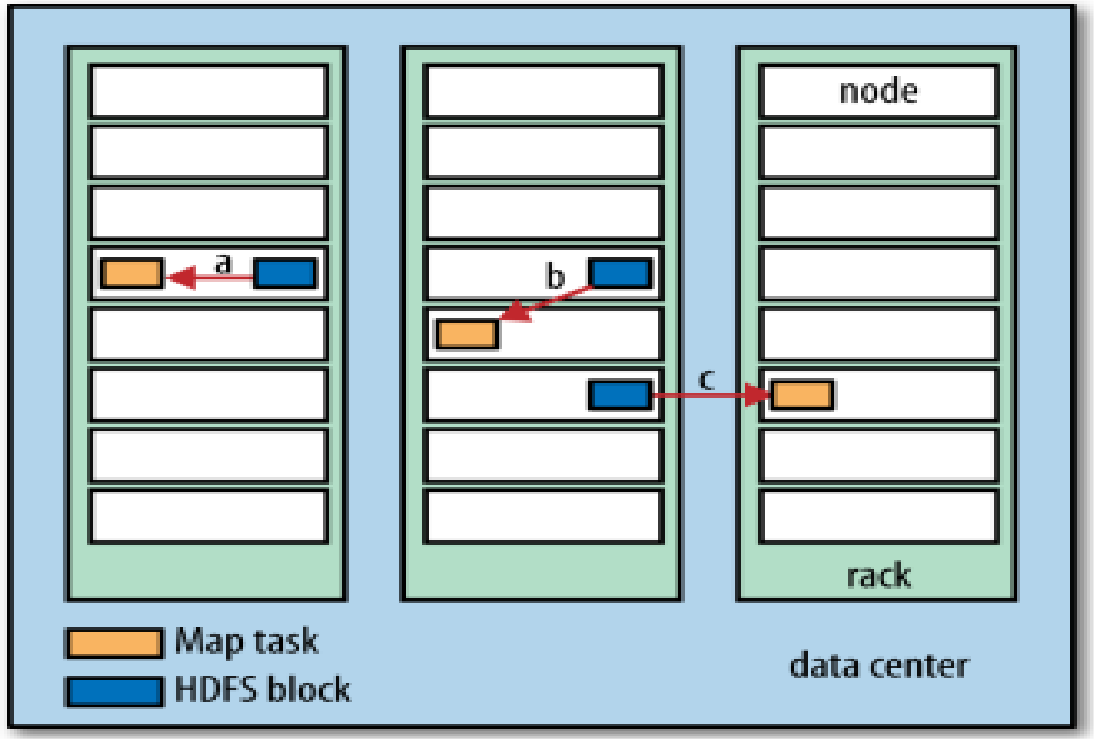
- A MapReduce job is a unit of work consists of :
  - input data
  - MapReduce program
  - configuration information
- Hadoop runs the job by dividing it into tasks, of which there are two types:
  - map tasks
  - reduce tasks

Input split:

- Hadoop divides the input to a MapReduce job into fixed-size pieces:
  - input splits or splits
- Hadoop creates one map task for each split, which runs the user-defined map function for each record in the split
- Having many splits (when the splits are small)→ better load balancing
- if splits are too small → overhead of managing the splits
- a good split size → size of an HDFS block, which is 128 MB by default

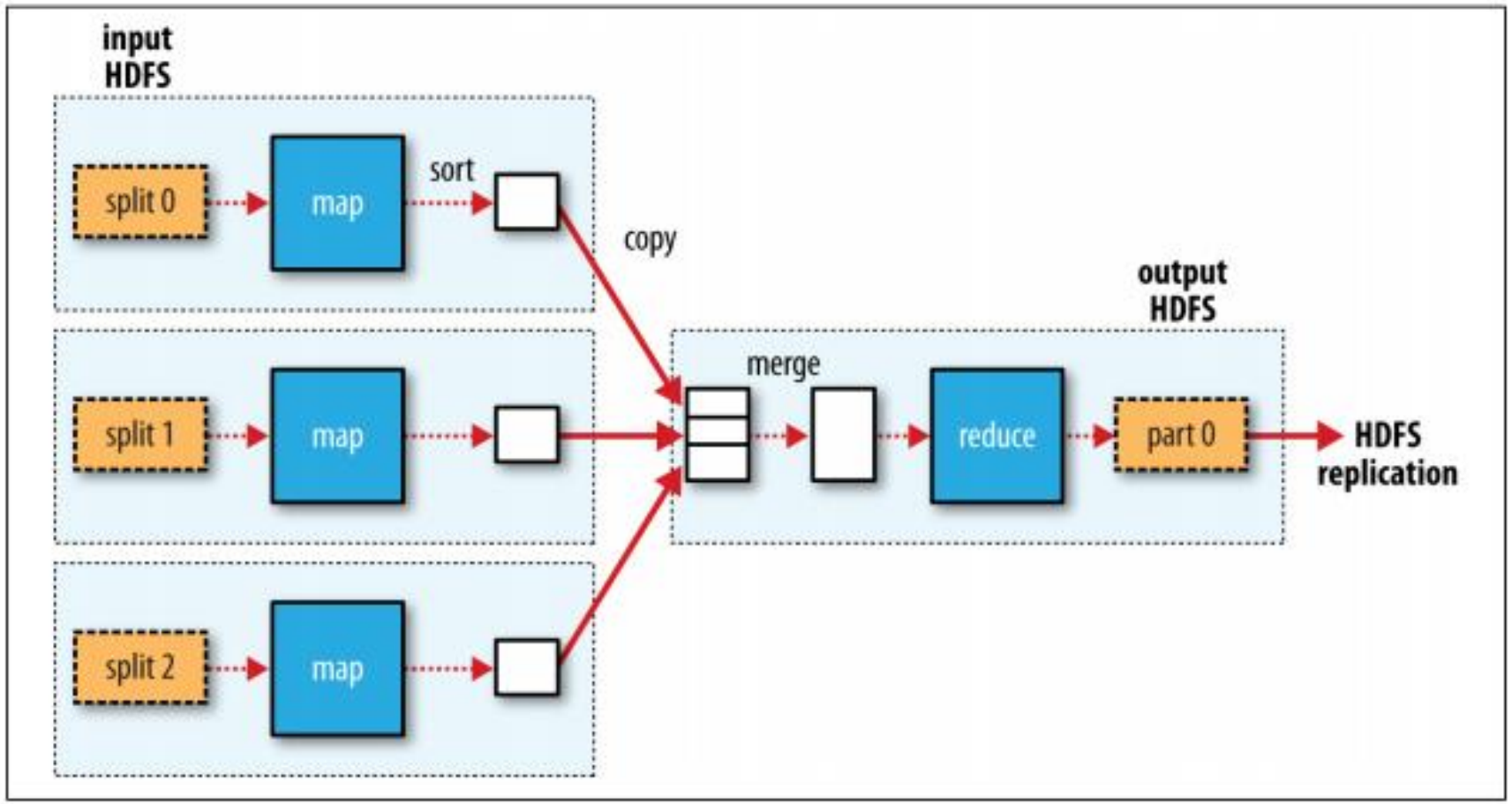
Data locality:

- Hadoop does its best to run the map task on a node where the input data resides in HDFS, because it doesn't use valuable cluster bandwidth.

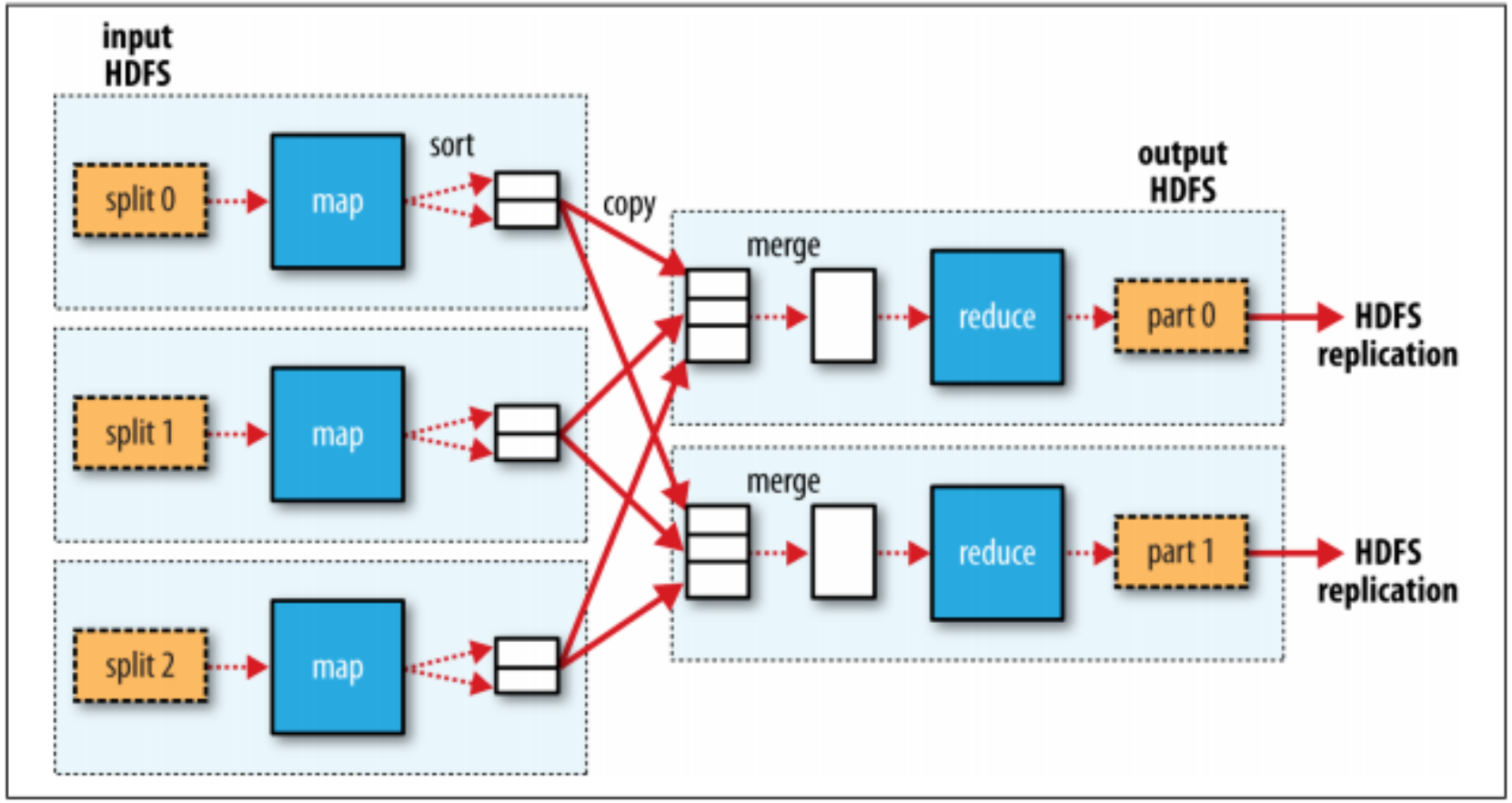


Data-local (a), rack-local (b), and off-rack (c) map tasks

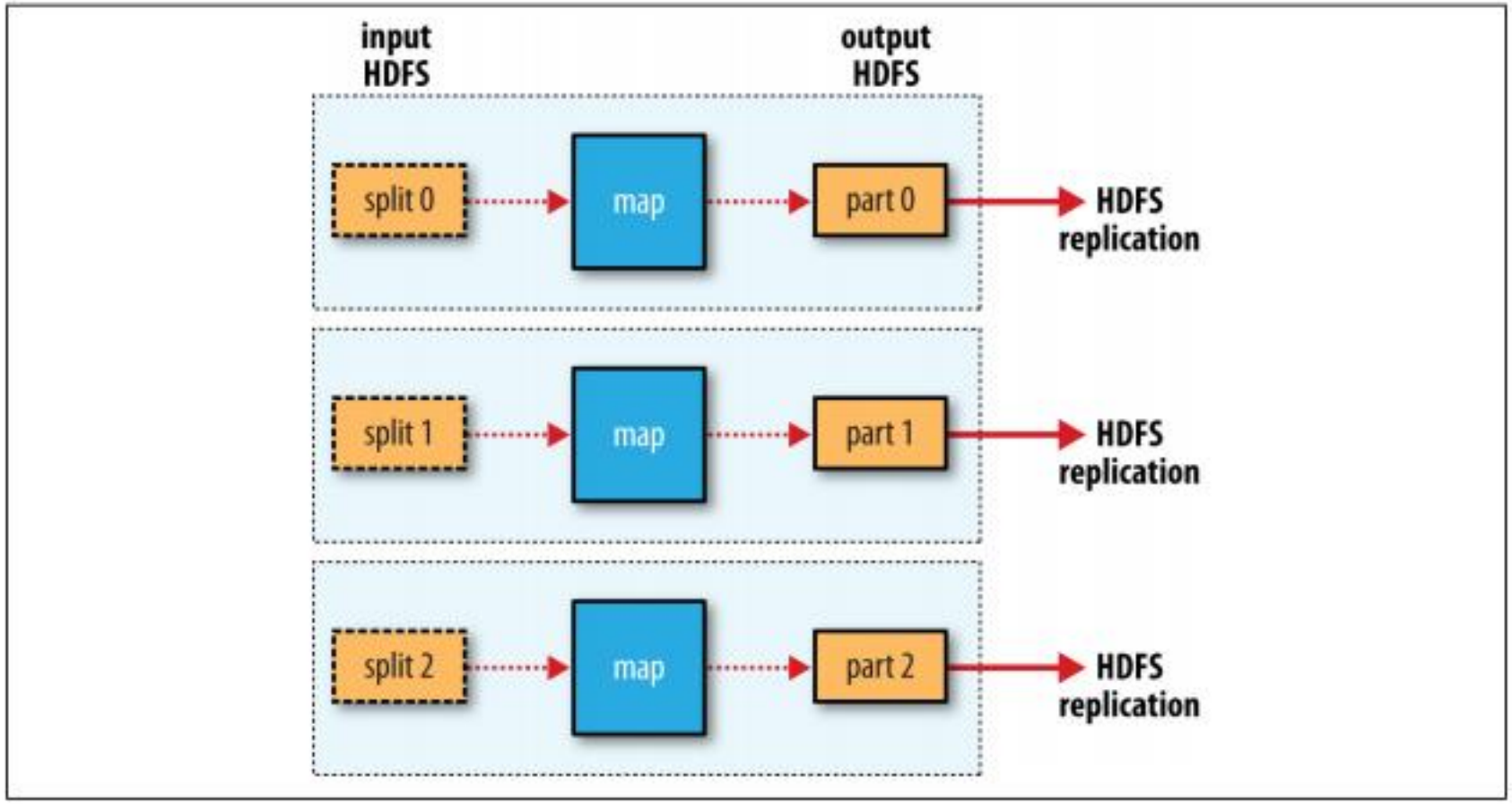
MapReduce data flow with a single reduce task:



MapReduce data flow with multiple reduce tasks:



MapReduce data flow with no reduce tasks:



Combiner Functions

- Hadoop allows the user to specify a combiner function to be run on the map output
- minimize the data transferred between map and reduce tasks
- bandwidth available on the cluster

Example of Combiner function:

- maximum temperature
- The first map output:

(1950, 0)

(1950, 20)

(1950, 10)

- The second map output:

1950, 25)

(1950, 15)

Combiner Functions

- Reduce function input:

(1950, [0, 20, 10, 25, 15])

- Reduce function output:

(1950, 25)

- Using a combiner function:

(1950, [20, 25])

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

calculating mean temperatures?

- mean(0, 20, 10, 25, 15) = 14
- mean(mean(0, 20, 10), mean(25, 15)) = mean(10, 20) = 15
- Not all functions possess this property
- The combiner function doesn’t replace the reduce function

Combiner **Functions**:

```
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 Jol
    job.setJarByClas
    job.setJobName("M
    FileInputFormat.a
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

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

    job.setOutputKeyClass(Text.class);

    job.setOutputValueClass(IntWritable.class);

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

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

Hadoop Streaming:

- Hadoop provides an API to MapReduce to write your map and reduce functions in languages other than Java
- Hadoop Streaming uses Unix standard streams as the interface between Hadoop and your program
- can use any language that can read standard input and write to standard output

The Hadoop Distributed Filesystem:

- Hadoop Distributed Filesystem (HDFS)
- sometimes references to “DFS” informally or in older documentation or configurations
- HDFS is a filesystem designed for storing very large files with streaming data access patterns

Advantage of HDFS:

- **Very large files :** There are Hadoop clusters running today that store petabytes of data
- **Streaming data access:** data processing pattern is a **write-once, read-many-times** pattern (time to read the whole dataset is more important than the latency in reading the first record)
- **Commodity hardware:** Hadoop doesn’t require expensive, highly reliable hardware

Disadvantage of HDFS:

- **Low-latency data access :** Applications that require low-latency access to data, in the tens of milliseconds range, will not work well with HDFS (HBase is currently a better choice for low-latency access)
- **Lots of small files :** namenode holds filesystem metadata in memory, the limit to the number of files in a filesystem(amount of memory on the namenode)
- **Multiple writers, arbitrary file modifications:** Files in HDFS may be written to by a single writer. Writes are always made at the end of the file, in append-only fashion. There is no support for multiple writers or for modifications at arbitrary offsets in the file

HDFS Concepts:

- ❑ Blocks
- ❑ Namenodes and Datanodes
- ❑ Block Caching
- ❑ HDFS Federation
- ❑ HDFS High Availability

Blocks:

- A disk has a block size, which is the minimum amount of data that it can read or write
- Filesystem blocks are typically a few kilobytes in size, whereas disk blocks are normally 512 bytes
- HDFS, too, has the concept of a block, but it is a much larger unit—128 MB by default
- files in HDFS are broken into block-sized chunks
- Why Is a Block in HDFS so Large? to minimize the cost of seeks
- blocks fit well with replication for providing fault tolerance

and availability.

Namenodes and Datanodes:

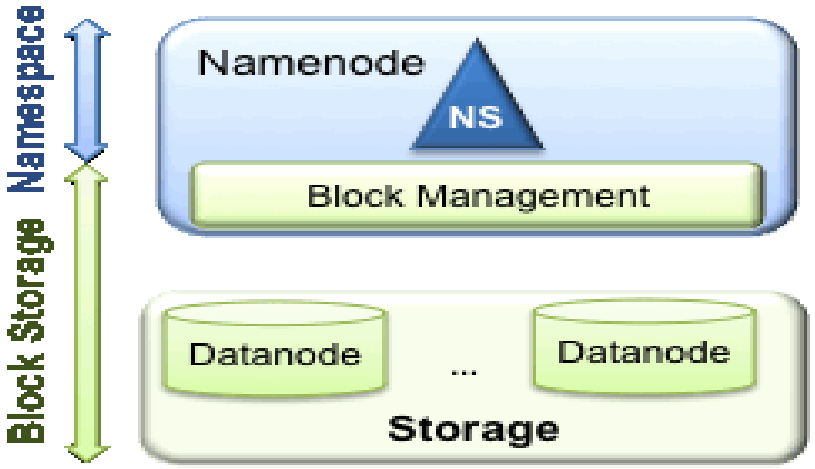
- namenode(the master) and a number of datanodes(workers)
- The namenode manages the filesystem namespace
- The namenode maintains the filesystem tree and the metadata for all the files and directories in the tree
- This information is stored persistently on the local disk in the form of two files: the namespace image and the edit log

Block Caching:

- a datanode reads blocks from disk, but for frequently accessed files the blocks may be explicitly cached in the datanode’s memory

- By default, a block is cached in only one datanode’s memory, although the number is configurable

HDFS Federation:



- The namenode keeps a reference to every file and block in the filesystem in memory
- HDFS federation, introduced in the 2.x release series, allows a cluster to scale by adding namenodes

HDFS High Availability:

- The combination of replicating namenode metadata on multiple filesystems 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)
- Hadoop 2 remedied this situation by adding support for HDFS high availability (HA) In this implementation, there are a pair of namenodes in an active-standby configuration.

Hadoop - Command Reference:

There are many more commands in **"$HADOOP_HOME/bin/hadoop fs"** than are demonstrated here, although these basic operations will get you started.

Running ./bin/hadoop dfs with no additional arguments will list all the commands that can be run with the FsShell system.

Furthermore, **$HADOOP_HOME/bin/hadoop fs -help** command Name will display a short usage summary for the operation in question, if you are stuck.

A table of all the operations is shown below. The following conventions are used for parameters −

"<path>" means any file or directory name.
"<path>..." means one or more file or directory names.
"<file>" means any filename.
"<src>" and "<dest>" are path names in a directed operation.
"<localSrc>" and "<localDest>" are paths as above, but on the local file system.

Basic Filesystem Operations:

- fs –help
- hadoop fs -copyFromLocal input/docs/quangle.txt hdfs://localhost/user/tom/quangle.txt
- the default, hdfs://localhost, as specified in core-site.xml:

hadoop fs -copyFromLocal input/docs/quangle.txt /user/tom/quangle.txt

- hadoop fs -copyToLocal quangle.txt quangle.copy.txt
- hadoop fs -mkdir books
- hadoop fs -ls

Sr.No	Command & Description
1	**-ls <path>** Lists the contents of the directory specified by path, showing the names, permissions, owner, size and modification date for each entry.
2	**-lsr <path>** Behaves like -ls, but recursively displays entries in all subdirectories of path.
3	**-du <path>** Shows disk usage, in bytes, for all the files which match path; filenames are reported with the full HDFS protocol prefix.
4	**-dus <path>** Like -du, but prints a summary of disk usage of all files/directories in the path.
5	**-mv <src><dest>** Moves the file or directory indicated by src to dest, within HDFS.
6	**-cp <src> <dest>** Copies the file or directory identified by src to dest, within HDFS.

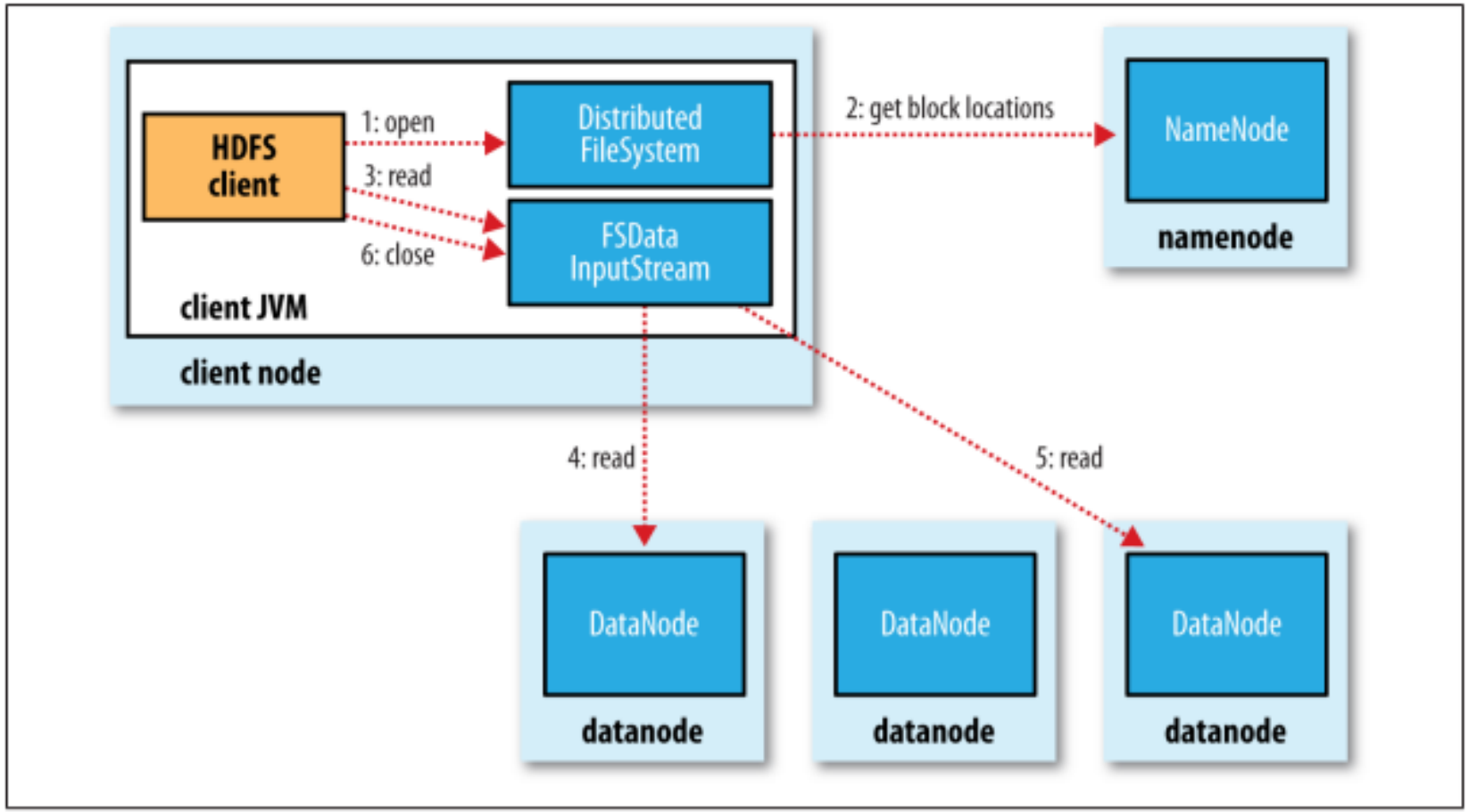
7	**-rm <path>** Removes the file or empty directory identified by path.
8	**-rmr <path>** Removes the file or directory identified by path. Recursively deletes any child entries (i.e., files or subdirectories of path).
9	**-put <localSrc> <dest>** Copies the file or directory from the local file system identified by localSrc to dest within the DFS.
10	**-copyFromLocal <localSrc> <dest>** Identical to -put
11	**-moveFromLocal <localSrc> <dest>** Copies the file or directory from the local file system identified by localSrc to dest within HDFS, and then deletes the local copy on success.
12	**-get [-crc] <src> <localDest>** Copies the file or directory in HDFS identified by src to the local file system path identified by localDest.

13	**-getmerge <src> <localDest>** Retrieves all files that match the path src in HDFS, and copies them to a single, merged file in the local file system identified by localDest.
14	**-cat <filen-ame>** Displays the contents of filename on stdout.
15	**-copyToLocal <src> <localDest>** Identical to -get
16	**-moveToLocal <src> <localDest>** Works like -get, but deletes the HDFS copy on success.
17	**-mkdir <path>** Creates a directory named path in HDFS. Creates any parent directories in path that are missing (e.g., mkdir -p in Linux).
18	**-setrep [-R] [-w] rep <path>** Sets the target replication factor for files identified by path to rep. (The actual replication factor will move toward the target over time)

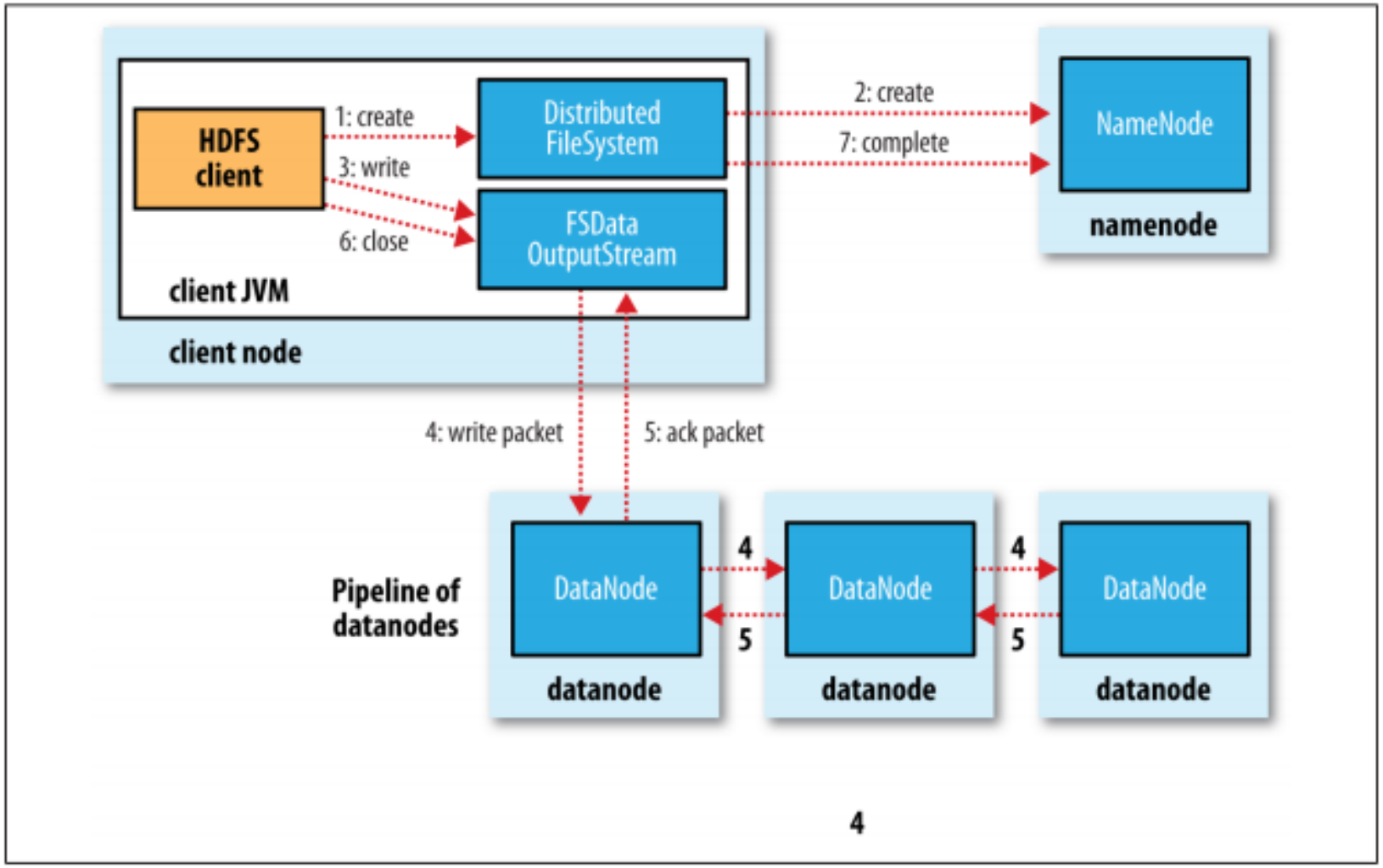
19	**-touchz <path>** Creates a file at path containing the current time as a timestamp. Fails if a file already exists at path, unless the file is already size 0.
20	**-test -[ezd] <path>** Returns 1 if path exists; has zero length; or is a directory or 0 otherwise.
21	**-stat [format] <path>** Prints information about path. Format is a string which accepts file size in blocks (%b), filename (%n), block size (%o), replication (%r), and modification date (%y, %Y).
22	**-tail [-f] <file2name>** Shows the last 1KB of file on stdout.
23	**-chmod [-R] mode,mode,... <path>...** Changes the file permissions associated with one or more objects identified by path.... Performs changes recursively with R. mode is a 3-digit octal mode, or {augo}+/-{rwxX}. Assumes if no scope is specified and does not apply an umask.
24	**-chown [-R] [owner][:[group]] <path>...** Sets the owning user and/or group for files or directories identified by path.... Sets owner recursively if -R is specified.

25	**-chgrp [-R] group <path>...** Sets the owning group for files or directories identified by path.... Sets group recursively if -R is specified.
26	**-help <cmd-name>** Returns usage information for one of the commands listed above. You must omit the leading '-' character in cmd.

A client reading data from HDFS:



A client writing data to HDFS:



Network Topology and Hadoop:

- The idea is to use the bandwidth between two nodes as a measure of distance
- measuring bandwidth between nodes → difficult
- Hadoop takes a simple approach in which the network is represented as a tree
- Distance between two nodes is the sum of their distances to their closest common ancestor
- levels : **data center**, **rack**, and the **node** that a process is running on

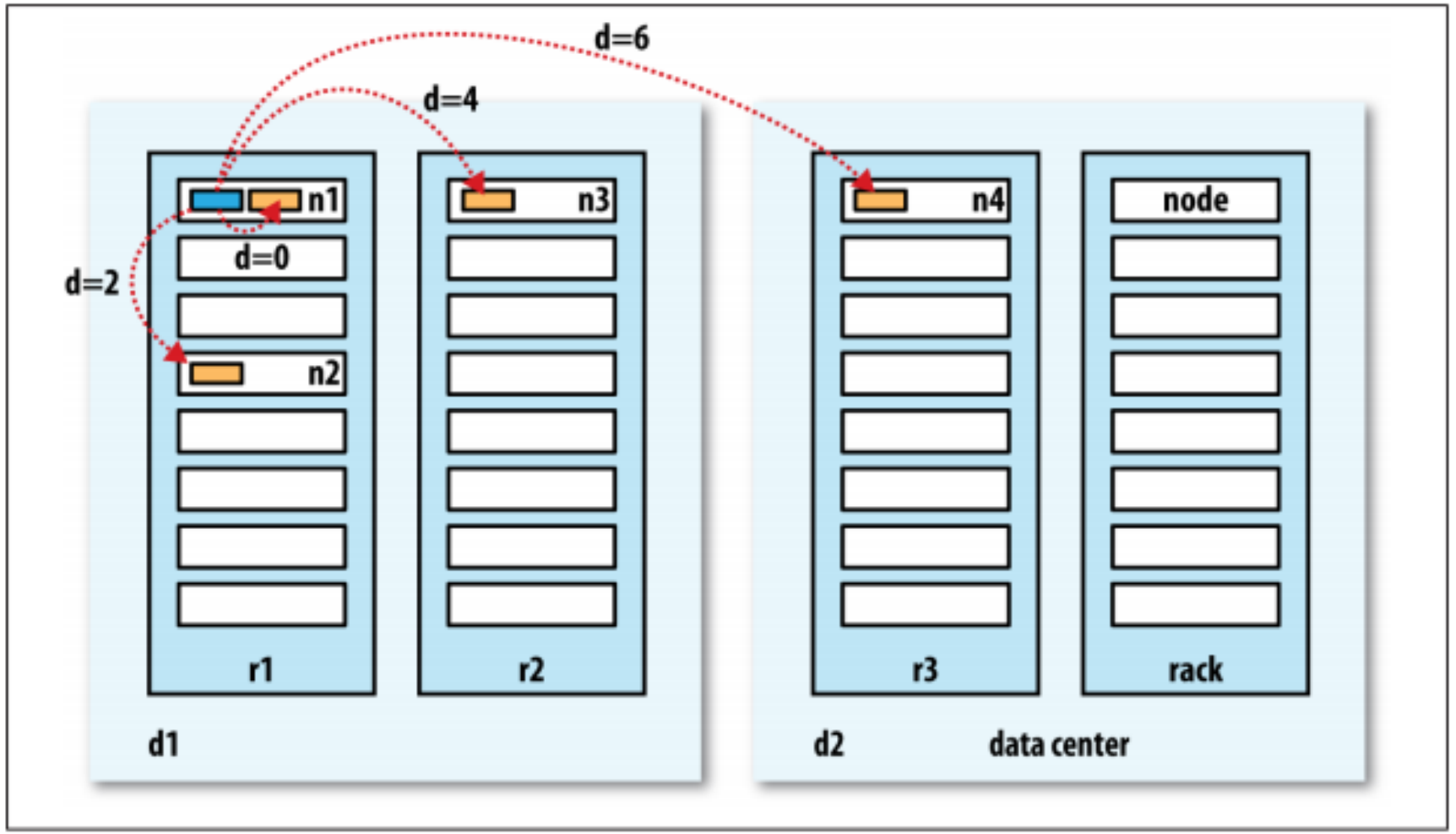
Network Topology and Hadoop:

- 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
- node n1 on rack r1 in data center d1 → /d1/r1/n1

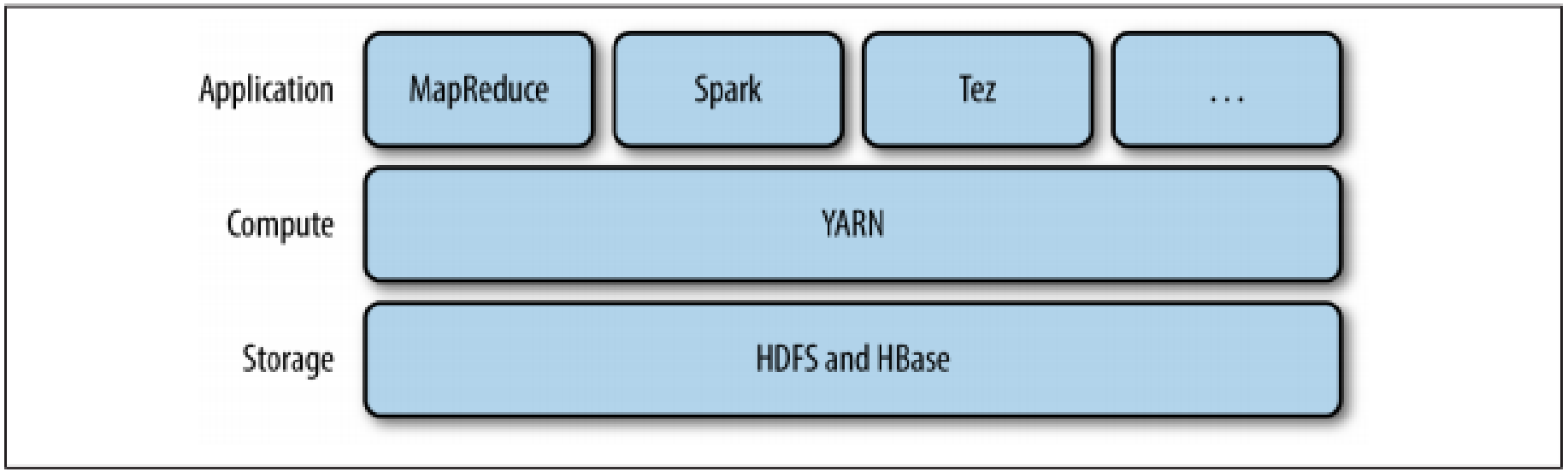
four scenarios:

- 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)

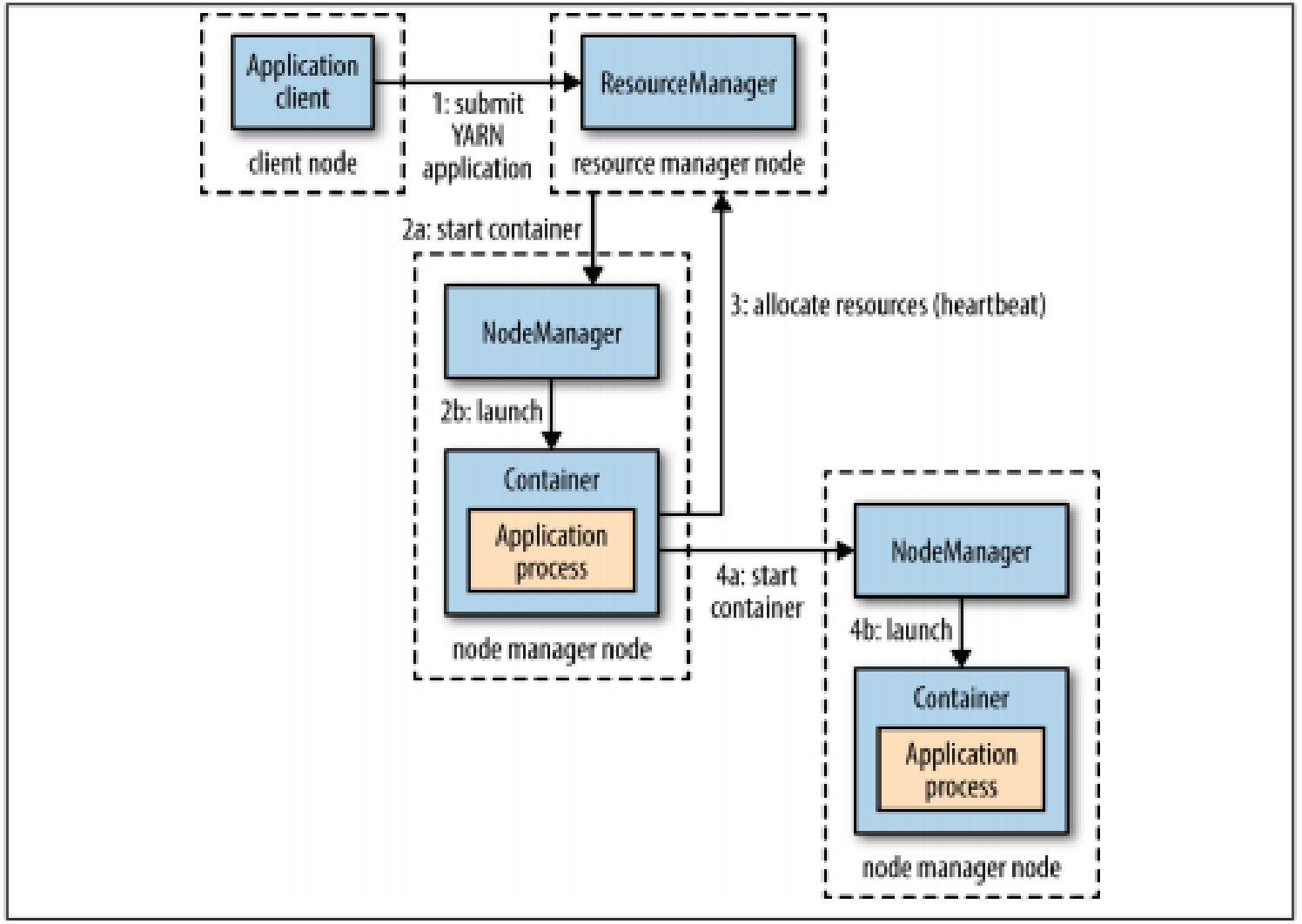
Network distance in Hadoop:



Apache YARN (Yet Another Resource Negotiator):



Anatomy of a YARN Application Run:



YARN Compared to MapReduce 1:

- In MapReduce 1, there are two types of daemon that control the job execution process: a **jobtracker** and one or more **tasktrackers**
- The jobtracker coordinates all the jobs run on the system by scheduling tasks to run on tasktrackers
- Tasktrackers run tasks and send progress reports to the jobtracker
- If a task fails, the jobtracker can reschedule it on a different tasktracker

- In MapReduce 1, the jobtracker takes care of both **job scheduling** (matching tasks with tasktrackers) and **task progress monitoring**

- in YARN these responsibilities are handled by separate entities: the **resource manager** and an **application master** (one for each MapReduce job)

- The **jobtracker** is also responsible for storing **job history** for completed jobs, although it is possible to run a job history server as a separate daemon. In **YARN**, the equivalent role is the **timeline server**, which stores application history

YARN Compared to MapReduce 1:

MapReduce 1	YARN
Jobtracker	Resource manager, application master, timeline server
Tasktracker	Node manager
Slot	Container

Advantage of YARN:

- YARN was designed to address many of the limitations in MapReduce1 Including:
- **Scalability:** MapReduce 1 hits scalability bottlenecks in the region of 4,000 nodes and 40,000 tasks YARN overcomes these limitations scale up to 10,000 nodes and 100,000 tasks
- **Availability:** jobtracker's responsibilities split between the resource manager and application master in YARN.

Divide and-conquer problem : provide HA for the resource manager, then for YARN applications

Advantage of YARN:

- **Utilization:** In MapReduce 1, each tasktracker is configured with a static allocation of fixed-size "**slots**," which are divided into **map slots** and **reduce slots** at configuration time. A map slot can only be used to run a map task, and a reduce slot can only be used for a reduce task. In YARN, a node manager manages a pool of resources, rather than a fixed number of designated slots. Furthermore, resources in YARN are fine grained, so an application can make a request for what it needs

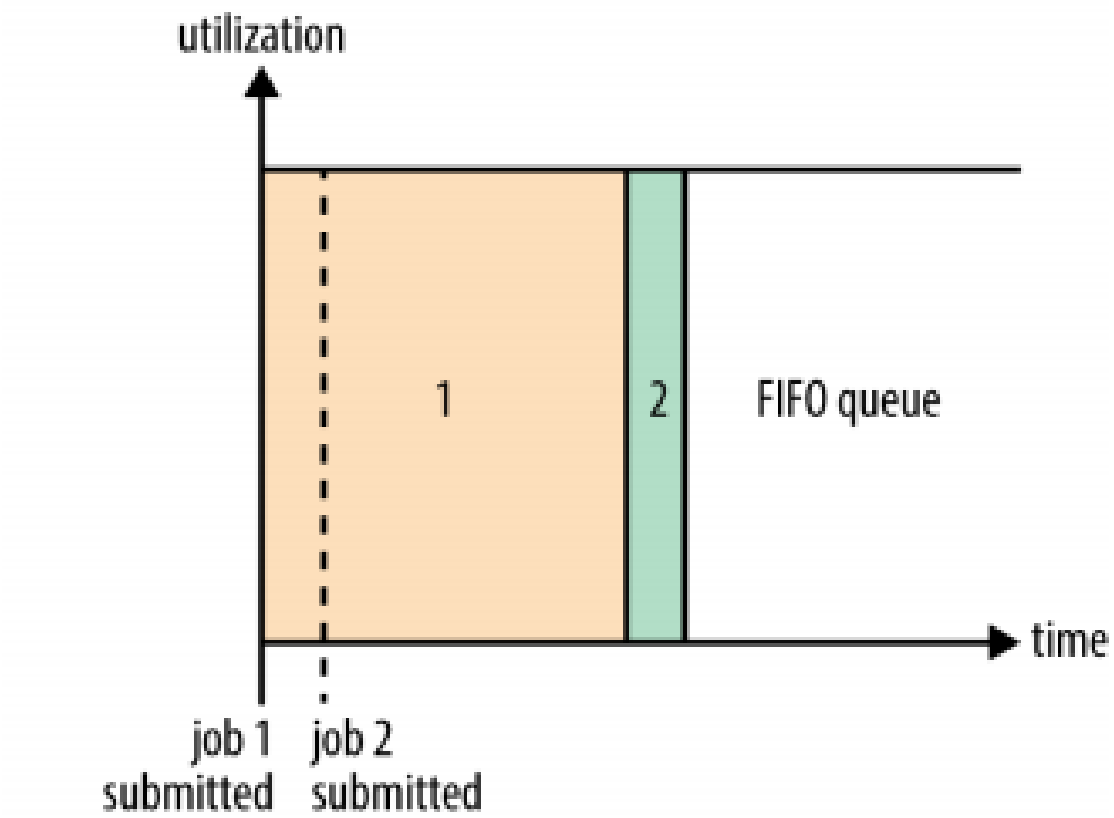
Scheduling in YARN :

- In the real world, however, resources are limited
- allocate resources to applications according to some defined policy
- Scheduling in general is a difficult problem and there is no one “best” policy
- Three schedulers are available in YARN: the **FIFO**, **Capacity**, and **Fair** Schedulers

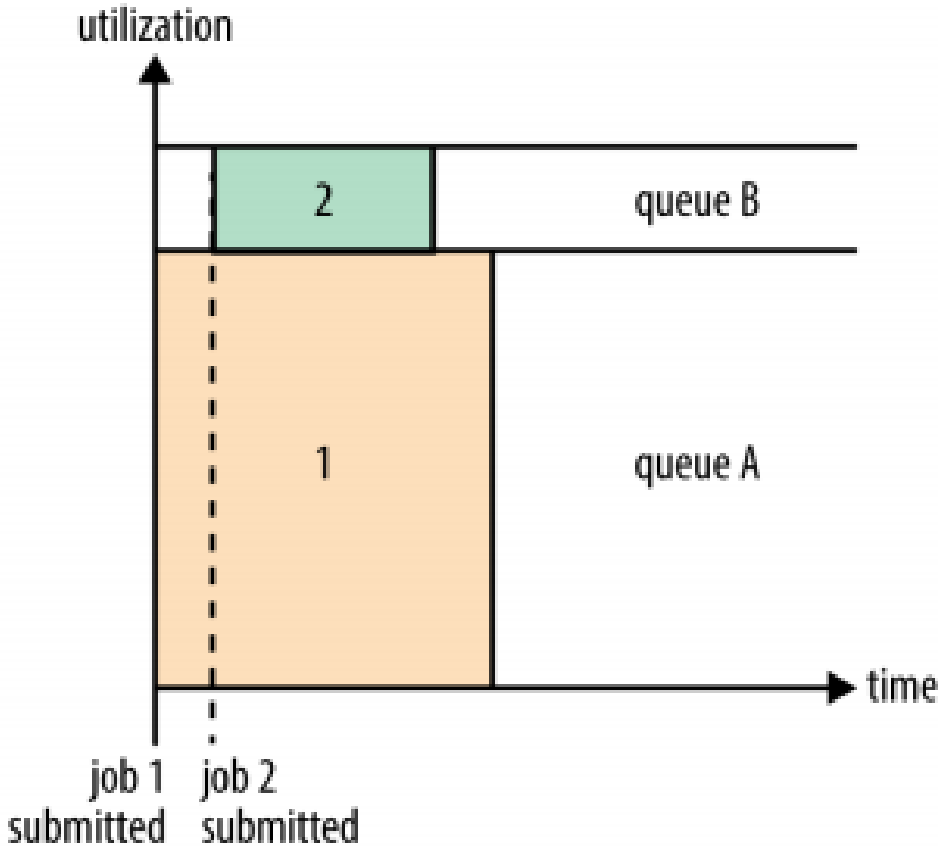
Scheduling in YARN :

- FIFO (First In First Out) : Large applications will use all the resources in a cluster, so each application has to wait its turn
- Capacity Scheduler , a separate dedicated queue allows the small job to start as soon as it is submitted
- Fair Scheduler , there is no need to reserve a set amount of capacity, since it will dynamically balance resources between all running jobs

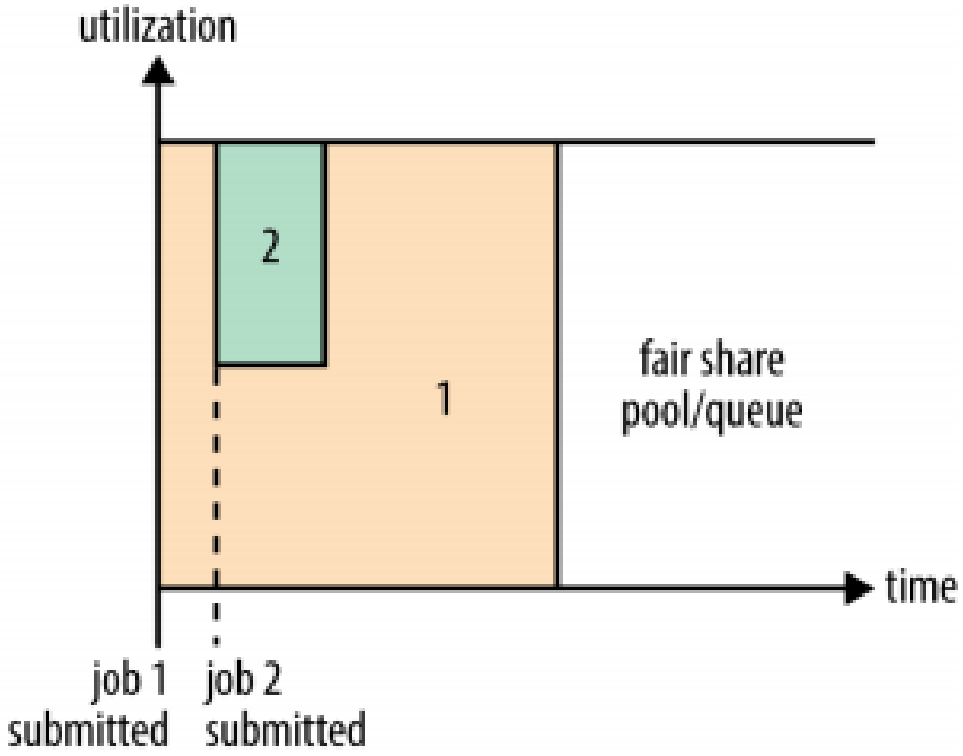
FIFO Scheduler:



Capacity Scheduler :

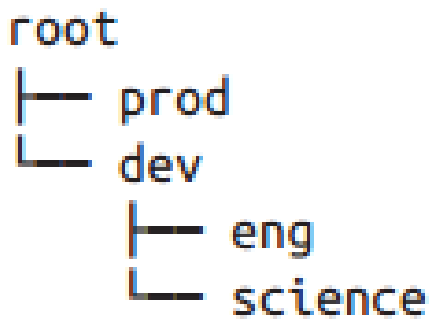


Fair Scheduler :



Capacity Scheduler Configuration :

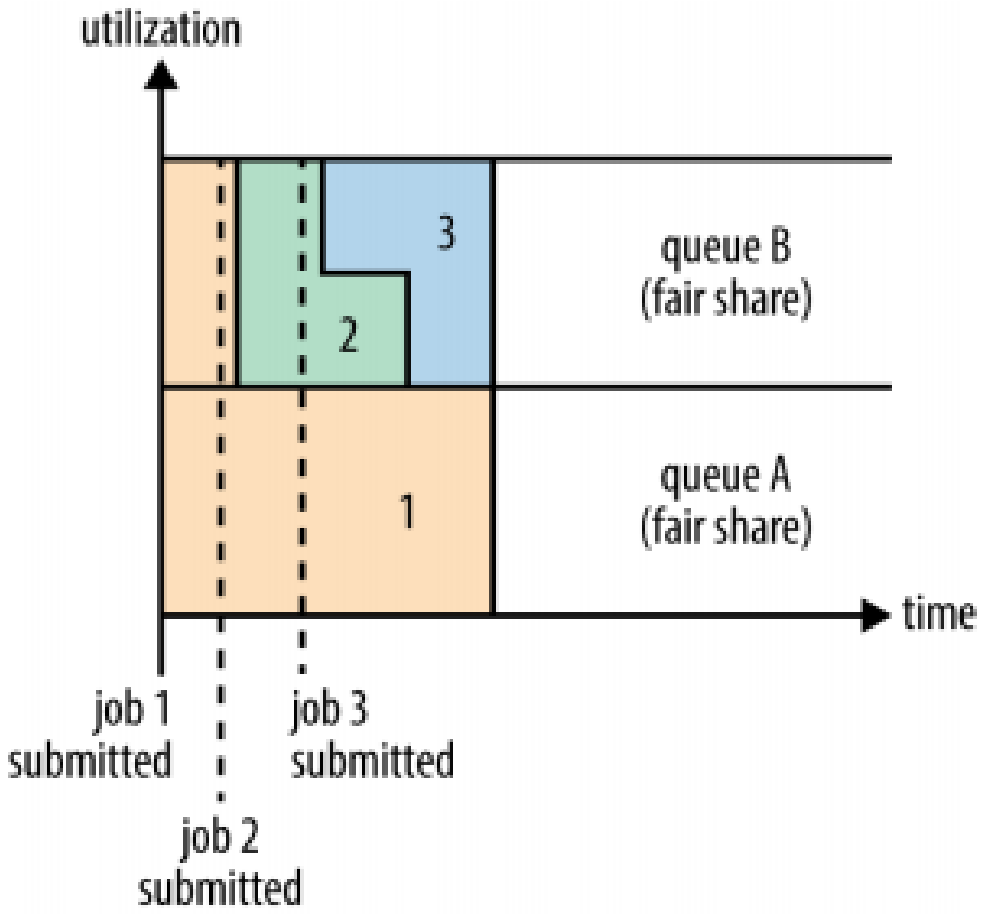
- queue hierarchy
- Capacity Scheduler configuration file : capacity-scheduler.xml



configuration file for the Capacity Scheduler :

```
<?xml version="1.0"?>
<configuration>
  <property>
    <name>yarn.scheduler.capacity.root.queues</name>
    <value>prod,dev</value>
  </property>
  <property>
    <name>yarn.scheduler.capacity.root.dev.queues</name>
    <value>eng,science</value>
  </property>
  <property>
    <name>yarn.scheduler.capacity.root.prod.capacity</name>
    <value>40</value>
  </property>
  <property>
    <name>yarn.scheduler.capacity.root.dev.capacity</name>
    <value>60</value>
  </property>
  <property>
    <name>yarn.scheduler.capacity.root.dev.maximum-capacity</name>
    <value>75</value>
  </property>
  <property>
    <name>yarn.scheduler.capacity.root.dev.eng.capacity</name>
    <value>50</value>
  </property>
  <property>
    <name>yarn.scheduler.capacity.root.dev.science.capacity</name>
    <value>50</value>
  </property>
</configuration>
```

Fair sharing between user queues:



An allocation file for the Fair Scheduler:

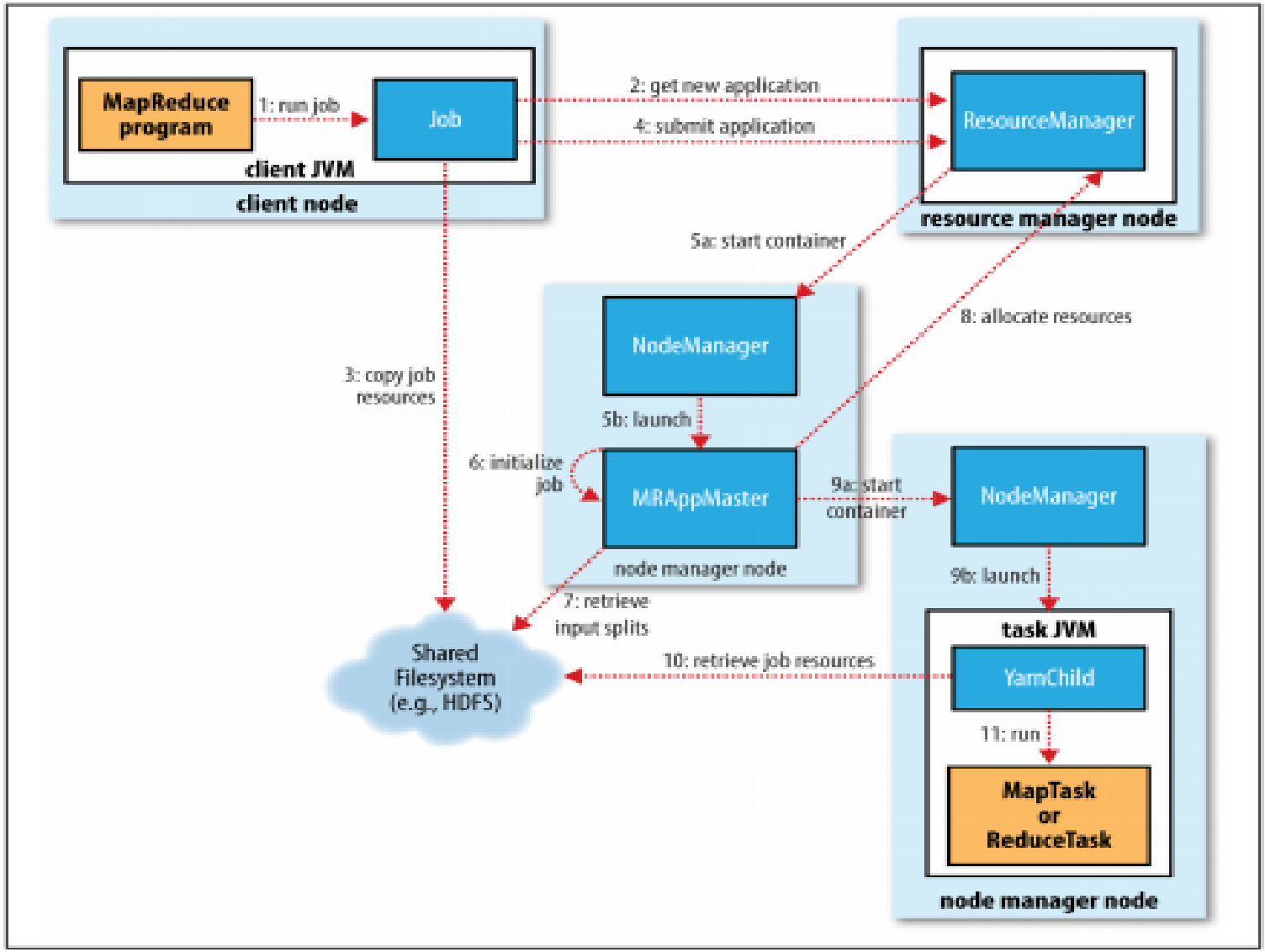
```
<?xml version="1.0"?>
<allocations>
  <defaultQueueSchedulingPolicy>fair</defaultQueueSchedulingPolicy>

  <queue name="prod">
    <weight>40</weight>
    <schedulingPolicy>fifo</schedulingPolicy>
  </queue>

  <queue name="dev">
    <weight>60</weight>
    <queue name="eng" />
    <queue name="science" />
  </queue>

  <queuePlacementPolicy>
    <rule name="specified" create="false" />
    <rule name="primaryGroup" create="false" />
    <rule name="default" queue="dev.eng" />
  </queuePlacementPolicy>
</allocations>
```

Anatomy of a MapReduce Job Run:



There are five independent entities:

- **Client:** submits the MapReduce job
- **YARN resource manager :** coordinates the allocation of compute resources on the cluster
- **YARN node managers:** launch and monitor the compute containers on machines in the cluster
- **MapReduce application master:** 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
- **Distributed filesystem (HDFS):** used for sharing job files between the other entities

Failures:

- In the real world, user code is buggy, processes crash, and machines fail
- Hadoop is its ability to handle such failures
- We need to consider the failure of any of the following entities:

✓ task
✓ application master
✓ node manager
✓ resource manager

Task Failure:

- user code in the map or reduce task throws a runtime exception
- If this happens, the task JVM reports the error back to its parent **application master** before it exits
- application master marks the task attempt as failed, and frees up the **container** so its resources are available for another task
- The timeout period after which tasks are considered failed is normally **10 minutes** and can be configured on a per-job basis by setting the **mapreduce.task.timeoutproperty** to a value in milliseconds.
- The task JVM process will be killed automatically after this period

Application Master Failure:

- in the face of hardware or network failures

- 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)

Node Manager Failure:

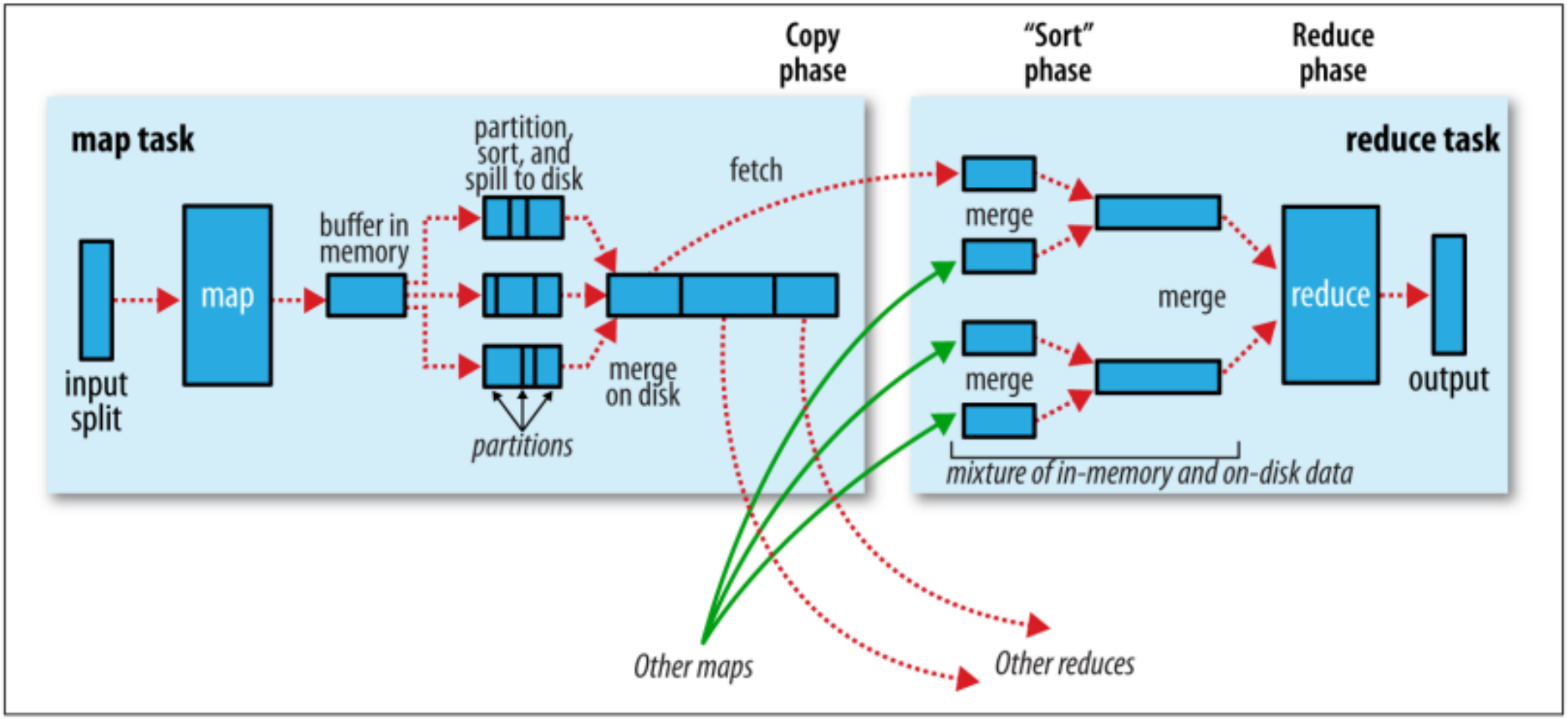
- If a node manager fails by crashing or running very slowly, it will stop sending heartbeats to the resource manager. The resource manager will notice a node manager that has stopped sending heartbeats if it hasn't received one for 10 minutes (this is configured, in milliseconds, via the yarn.resourcemanager.nm.liveness-monitor.expiry-interval-ms property) and remove it from its pool of nodes to schedule containers on.

- Node managers may be blacklisted if the number of failures for the application is high

Resource Manager Failure:

- Failure of the resource manager is serious
- To achieve high availability (HA), it is necessary to run a pair of resource managers in an active-standby configuration

Shuffle and sort in MapReduce:



Speculative Execution:

- The MapReduce model is to break jobs into tasks and run the tasks in parallel to make the overall job execution time smaller than it would be if the tasks ran sequentially

- job execution time sensitive to slow-running tasks. When a job consists of hundreds or thousands of tasks, the possibility of a few straggling tasks is very real

- Tasks may be slow for various reasons, including hardware degradation or software misconfiguration, but the causes may be hard to detect because the tasks still complete successfully, albeit after a longer time than expected. Hadoop doesn't try to diagnose and fix slow-running tasks; instead, it tries to detect when a task is running slower than expected and launches another equivalent task as a backup. This is termed speculative execution of tasks.

Speculative Execution:

- Speculative execution is turned on by default. It can be enabled or disabled independently for map tasks and reduce tasks

- Why would you ever want to turn speculative execution off? The goal of speculative execution is to reduce job execution time, but this comes at the cost of cluster efficiency

MapReduce Features:

- Counters: Counters are a useful channel for gathering statistics about the job

- Built-in Counters: Hadoop maintains some built-in counters for every job, and these report various metrics. For example, there are counters for the number of bytes and records processed

Built-in counter groups:

Group	Name/Enum
MapReduce task counters	`org.apache.hadoop.mapreduce.TaskCounter`
Filesystem counters	`org.apache.hadoop.mapreduce.FileSystemCounter`
`FileInputFormat` counters	`org.apache.hadoop.mapreduce.lib.input.FileInputFormatCounter`
`FileOutputFormat` counters	`org.apache.hadoop.mapreduce.lib.output.FileOutputFormatCounter`
Job counters	`org.apache.hadoop.mapreduce.JobCounter`

MapReduce Task counters:

Counter	Description
Map input records (MAP_INPUT_RECORDS)	The number of input records consumed by all the maps in the job. Incremented every time a record is read from a RecordReader and passed to the map's map() method by the framework.
Split raw bytes (SPLIT_RAW_BYTES)	The number of bytes of input-split objects read by maps. These objects represent the split metadata (that is, the offset and length within a file) rather than the split data itself, so the total size should be small.
Map output records (MAP_OUTPUT_RECORDS)	The number of map output records produced by all the maps in the job. Incremented every time the collect() method is called on a map's OutputCollector.
Map output bytes (MAP_OUTPUT_BYTES)	The number of bytes of uncompressed output produced by all the maps in the job. Incremented every time the collect() method is called on a map's OutputCollector.
Map output materialized bytes (MAP_OUTPUT_MATERIALIZED_BYTES)	The number of bytes of map output actually written to disk. If map output compression is enabled, this is reflected in the counter value.
Combine input records (COMBINE_INPUT_RECORDS)	The number of input records consumed by all the combiners (if any) in the job. Incremented every time a value is read from the combiner's iterator over values. Note that this count is the number of values consumed by the combiner, not the number of distinct key groups (which would not be a useful metric, since there is not necessarily one group per key for a combiner; see "Combiner Functions" on page 34, and also "Shuffle and Sort" on page 197).

MapReduce Task counters continue:

Combine output records (`COMBINE_OUTPUT_RECORDS`)	The number of output records produced by all the combiners (if any) in the job. Incremented every time the `collect()` method is called on a combiner's `OutputCollector`.
Reduce input groups (`REDUCE_INPUT_GROUPS`)	The number of distinct key groups consumed by all the reducers in the job. Incremented every time the reducer's `reduce()` method is called by the framework.
Reduce input records (`REDUCE_INPUT_RECORDS`)	The number of input records consumed by all the reducers in the job. Incremented every time a value is read from the reducer's iterator over values. If reducers consume all of their inputs, this count should be the same as the count for map output records.
Reduce output records (`REDUCE_OUTPUT_RECORDS`)	The number of reduce output records produced by all the maps in the job. Incremented every time the `collect()` method is called on a reducer's `OutputCollector`.
Reduce shuffle bytes (`REDUCE_SHUFFLE_BYTES`)	The number of bytes of map output copied by the shuffle to reducers.
Spilled records (`SPILLED_RECORDS`)	The number of records spilled to disk in all map and reduce tasks in the job.
CPU milliseconds (`CPU_MILLISECONDS`)	The cumulative CPU time for a task in milliseconds, as reported by */proc/cpuinfo*.
Physical memory bytes (`PHYSICAL_MEMORY_BYTES`)	The physical memory being used by a task in bytes, as reported by */proc/meminfo*.

MapReduce Task counters continue :

Virtual memory bytes (`VIRTUAL_MEMORY_BYTES`)	The virtual memory being used by a task in bytes, as reported by */proc/meminfo*.
Committed heap bytes (`COMMITTED_HEAP_BYTES`)	The total amount of memory available in the JVM in bytes, as reported by `Runtime.getRuntime().totalMemory()`.
GC time milliseconds (`GC_TIME_MILLIS`)	The elapsed time for garbage collection in tasks in milliseconds, as reported by `GarbageCollectorMXBean.getCollectionTime()`.
Shuffled maps (`SHUFFLED_MAPS`)	The number of map output files transferred to reducers by the shuffle (see "Shuffle and Sort" on page 197).
Failed shuffle (`FAILED_SHUFFLE`)	The number of map output copy failures during the shuffle.
Merged map outputs (`MERGED_MAP_OUTPUTS`)	The number of map outputs that have been merged on the reduce side of the shuffle.

Filesystem task counters :

Counter	Description
Filesystem bytes read (`BYTES_READ`)	The number of bytes read by the filesystem by map and reduce tasks. There is a counter for each filesystem, and *Filesystem* may be Local, HDFS, S3, etc.
Filesystem bytes written (`BYTES_WRITTEN`)	The number of bytes written by the filesystem by map and reduce tasks.
Filesystem read ops (`READ_OPS`)	The number of read operations (e.g., open, file status) by the filesystem by map and reduce tasks.
Filesystem large read ops (`LARGE_READ_OPS`)	The number of large read operations (e.g., list directory for a large directory) by the filesystem by map and reduce tasks.
Filesystem write ops (`WRITE_OPS`)	The number of write operations (e.g., create, append) by the filesystem by map and reduce tasks.

FileInputFormat task counters :

Counter	Description
Bytes read (BYTES_READ)	The number of bytes read by map tasks via the `FileInputFormat`.

FileOutputFormat task counters:

Counter	Description
Bytes written (BYTES_WRITTEN)	The number of bytes written by map tasks (for map-only jobs) or reduce tasks via the `FileOutputFormat`.

Job counters:

Counter	Description
Launched map tasks (TOTAL_LAUNCHED_MAPS)	The number of map tasks that were launched. Includes tasks that were started speculatively (see "Speculative Execution" on page 204).
Launched reduce tasks (TOTAL_LAUNCHED_REDUCES)	The number of reduce tasks that were launched. Includes tasks that were started speculatively.
Launched uber tasks (TOTAL_LAUNCHED_UBERTASKS)	The number of uber tasks (see "Anatomy of a MapReduce Job Run" on page 185) that were launched.
Maps in uber tasks (NUM_UBER_SUBMAPS)	The number of maps in uber tasks.
Reduces in uber tasks (NUM_UBER_SUBREDUCES)	The number of reduces in uber tasks.
Failed map tasks (NUM_FAILED_MAPS)	The number of map tasks that failed. See "Task Failure" on page 193 for potential causes.
Failed reduce tasks (NUM_FAILED_REDUCES)	The number of reduce tasks that failed.
Failed uber tasks (NUM_FAILED_UBERTASKS)	The number of uber tasks that failed.
Killed map tasks (NUM_KILLED_MAPS)	The number of map tasks that were killed. See "Task Failure" on page 193 for potential causes.

Job counters continue:

Killed reduce tasks (NUM_KILLED_REDUCES)	The number of reduce tasks that were killed.
Data-local map tasks (DATA_LOCAL_MAPS)	The number of map tasks that ran on the same node as their input data.
Rack-local map tasks (RACK_LOCAL_MAPS)	The number of map tasks that ran on a node in the same rack as their input data, but were not data-local.
Other local map tasks (OTHER_LOCAL_MAPS)	The number of map tasks that ran on a node in a different rack to their input data. Inter-rack bandwidth is scarce, and Hadoop tries to place map tasks close to their input data, so this count should be low. See Figure 2-2.
Total time in map tasks (MILLIS_MAPS)	The total time taken running map tasks, in milliseconds. Includes tasks that were started speculatively. See also corresponding counters for measuring core and memory usage (VCORES_MILLIS_MAPS and MB_MILLIS_MAPS).
Total time in reduce tasks (MILLIS_REDUCES)	The total time taken running reduce tasks, in milliseconds. Includes tasks that were started speculatively. See also corresponding counters for measuring core and memory usage (VCORES_MILLIS_REDUCES and MB_MILLIS_REDUCES).

User-Defined Java Counters :

Counter	Description
Filesystem bytes read (BYTES_READ)	The number of bytes read by the filesystem by map and reduce tasks. There is a counter for each filesystem, and *Filesystem* may be Local, HDFS, S3, etc.
Filesystem bytes written (BYTES_WRITTEN)	The number of bytes written by the filesystem by map and reduce tasks.
Filesystem read ops (READ_OPS)	The number of read operations (e.g., open, file status) by the filesystem by map and reduce tasks.
Filesystem large read ops (LARGE_READ_OPS)	The number of large read operations (e.g., list directory for a large directory) by the filesystem by map and reduce tasks.
Filesystem write ops (WRITE_OPS)	The number of write operations (e.g., create, append) by the filesystem by map and reduce tasks.

How Much Memory Does a Namenode Need?

- A namenode can eat up memory, since a reference to every block of every file is maintained in memory. It's difficult to give a precise formula because memory usage depends on the number of blocks per file, the filename length, and the number of directories in the filesystem; plus, it can change from one Hadoop release to another.
- The default of 1000 MB of namenode memory is normally enough for a few million files, but as a rule of thumb for sizing purposes, you can conservatively allow 1000 MB per million blocks of storage.
- For example, a 200-node cluster with 24 TB of disk space per node, a block size of 128 MB, and a replication factor of 3 has room for about 2 million blocks (or more): 200 × 24,000,000 MB ⁄ (128 MB × 3). So in this case, setting the namenode memory to 12,000 MB would be a good starting point.

How Much Memory Does a Namenode Need? Cont…

- You can increase the namenode's memory without changing the memory allocated to other Hadoop daemons by setting HADOOP_NAMENODE_OPTS in hadoop-env.shto include a JVM option for setting the memory size. HADOOP_NAMENODE_OPTS allows you to pass extra options to the namenode's JVM. So, for example, if you were using a Sun JVM, -Xmx2000m would specify that 2,000 MB of memory should be allocated to the name-node.If you change the namenode's memory allocation, don't forget to do the same for the secondary namenode (using the HADOOP_SECONDARYNAMENODE_OPTS variable), since its memory requirements are comparable to the primary namenode's.

Memory settings in YARN and MapReduce?

- YARN treats memory in a more fine-grained manner than the slot-based model used in MapReduce 1. Rather than specifying a fixed maximum number of map and reduce slots that may run on a node at once, YARN allows applications to request an arbitrary amount of memory (within limits) for a task. In the YARN model, node managers allocate memory from a pool, so the number of tasks that are running on a particular node depends on the sum of their memory requirements, and not simply on a fixed number of slots.
- The calculation for how much memory to dedicate to a node manager for running containers depends on the amount of physical memory on the machine. Each Hadoop daemon uses 1,000 MB, so for a datanode and a node manager, the total is 2,000 MB. configuration property yarn.nodemanager.resource.memory-mb to the total allocation in MB. (The default is 8,192 MB, which is normally too low for most setups.

Memory settings Cont …

❑ determine how to set memory options for individual jobs. There are two main controls: one for the size of the container allocated by YARN, and another for the heap size of the Java process run in the container.

MapReduce job memory properties (set by the client):

Property name	Type	Default value	Description
`mapreduce.map.memory.mb`	`int`	1024	The amount of memory for map containers.
`mapreduce.reduce.memory.mb`	`int`	1024	The amount of memory for reduce containers.
`mapred.child.java.opts`	`String`	`-Xmx200m`	The JVM options used to launch the container process that runs map and reduce tasks. In addition to memory settings, this property can include JVM properties for debugging, for example.
`mapreduce.map.java.opts`	`String`	`-Xmx200m`	The JVM options used for the child process that runs map tasks.
`mapreduce.reduce.java.opts`	`String`	`-Xmx200m`	The JVM options used for the child process that runs reduce tasks.

CPU settings in YARN and MapReduce?

- The number of cores that a node manager can allocate to containers is controlled by the yarn.nodemanager.resource.cpuvcoresproperty. It should be set to the total number of cores on the machine, minus a core for each daemon process running on the machine (datanode, node manager, and any other long-running processes).

- MapReduce jobs can control the number of cores allocated to map and reduce containers by setting mapreduce.map.cpu.vcores and mapreduce.reduce.cpu.vcores. Both default to 1, an appropriate setting for normal single-threaded MapReduce tasks, which can only saturate a single core.