DISTRIBUTED COMPUTING

Lecture 2 - Map Reduce

Marco Massenzio
http://codetrips.com
http://linkedin.com/in/mmassenzio

AGENDA

- Map Reduce basics
- Hadoop - architecture and example usage
- MapReduce example
- Hadoop File System (HDFS)
- MapReduce Jobs
- Real-life Hadoop: Clusters

MAP-REDUCE

At its most basic, a map() takes a List of objects; a function that operates on each of those objects and returns an object of another (or even the same) type; and then returns the list resulting from applying the method to each of those objects:

```
val names: List[String]= List(“foo”, “bar”, “quz”)
val upper_names = names.map(n: String => n.upper())
upper_names === List(“FOO”, “BAR”, “QUZ”)
```

This is a general pattern in functional programming, Scala is the foremost example and more recently, Java 8 introduced it too (Lambdas)

MAP-REDUCE

More formally, a map() is defined as:

```
List[A]::map(f: A => B): List[B]
```

in Python the `map(function, iterable, ...)` built-in takes an Iterable and essentially does the same thing:

```
>>> names = ['joe', 'bob', 'althea']
>>> map(lambda s: s.upper(), names)
['JOE', 'BOB', `ALTHEA']
```

You can obviously pass in a function:

```
>>> def capitalize_first(name):
        return ``.join([name[0].upper(), name[1:]])
>>> map(capitalize_first, names)
['Joe', 'Bob', 'Althea']
```

https://docs.python.org/3.4/library/functions.html#map

MAP-REDUCE

The reduce() is the "accumulation" counterpart to the map() in that it takes a List of objects; a function that operates on each of those objects and an "accumulator" and returns an object of another (or even the same) type; and then returns the result of applying the method to each of those objects:

```
val names: List[String]= List("First", "Second", "Third")
val concat = names.reduceLeft[String]{(acc: String, n: String) =>
      s"$acc, $n"}
concat === "First, Second, Third"
```

This is a less general pattern in functional programming (Scala prefers the `foldLeft/foldRight` patterns) but Java 8 has it too (Lambdas)

MAP-REDUCE

In Python the `reduce(function, iterable, [initializer])` built-in takes an Iterable and essentially does the same thing:

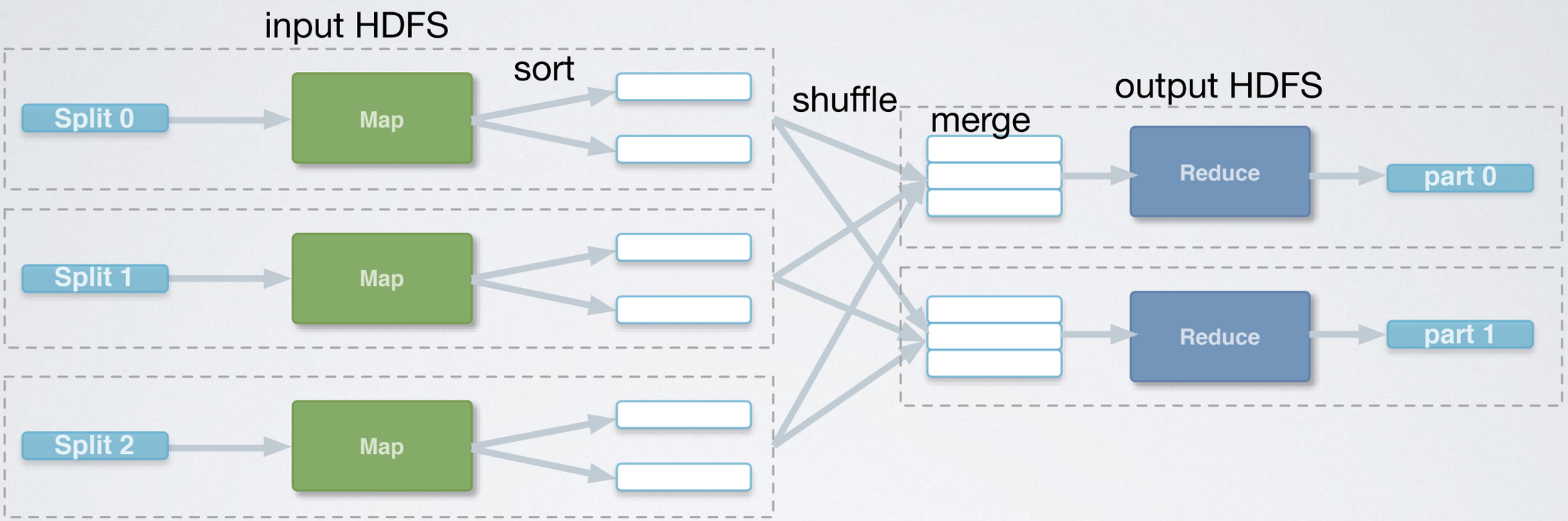
```
>>> names = ['joe', 'bob', 'althea']
>>> reduce(lambda acc, n: acc + “, “ + n, names)
'joe, bob, althea'
```

(note that in Python 3 this has been replaced with `functools.reduce()`)

MAP-REDUCE IN HADOOP

- Hadoop takes this functional paradigm and applies it to highly distributed computing, adding an "hidden" intermediate step (shuffle)
- It also uses the {key, value} concept, which we will explore in some more depth when talking about NoSQL paradigms
- An Hadoop job takes a (distributed) sequence of unsorted, unordered objects (e.g., all the lines from a CSV file)
- The Map step is then called for each of the elements of the sequence: this emits one or more {key, value} pair(s) (of the same, or entirely different types)
- The resulting pairs are then "shuffled" so that each Reduce is invoked once for each {key} (in sorted order) with the associated List(values) generated by the Map step
- Finally, the Reduce "does something" with the values and emits the result (again, indexed by a {key})

COME AGAIN?


input HDFS
sort
Split 0
Map
Split 1
Map
Split 2
Map
shuffle
output HDFS
merge
Reduce
part 0
Reduce
part 1

TEMP/CPU LOG

```
12-01-2015T21:59:45,30.0,29.0,28.0,35.0,30.0,29.0, 0.65, 0.58, 0.46
12-01-2015T22:00:15,29.0,28.0,27.0,34.0,28.0,27.0, 0.45, 0.54, 0.45
12-01-2015T22:00:45,28.0,27.0,25.0,35.0,29.0,28.0, 0.33, 0.50, 0.44
12-01-2015T22:01:15,28.0,26.0,25.0,35.0,30.0,27.0, 0.27, 0.47, 0.43
12-01-2015T22:01:45,28.0,27.0,25.0,33.0,28.0,27.0, 0.32, 0.46, 0.43
12-01-2015T22:02:15,28.0,26.0,26.0,33.0,29.0,28.0, 0.34, 0.45, 0.43
12-01-2015T22:02:45,28.0,28.0,27.0,34.0,27.0,28.0, 0.27, 0.42, 0.42
12-01-2015T22:03:15,28.0,26.0,26.0,35.0,28.0,26.0, 0.22, 0.40, 0.41
12-01-2015T22:03:45,28.0,27.0,25.0,34.0,27.0,27.0, 0.13, 0.36, 0.40
```

```
Map:

line -> (max_temp, CPU Load)
```

```
Reduce:

(temp, [CPU Load …]) -> (temp, avg Load)
```

```
35.0    1.8154256
36.0    1.4620891
37.0    1.9172298
38.0    3.088998
39.0    3.1905377
40.0    3.2241063
41.0    3.52541
42.0    1.4878695
```

MAPPER

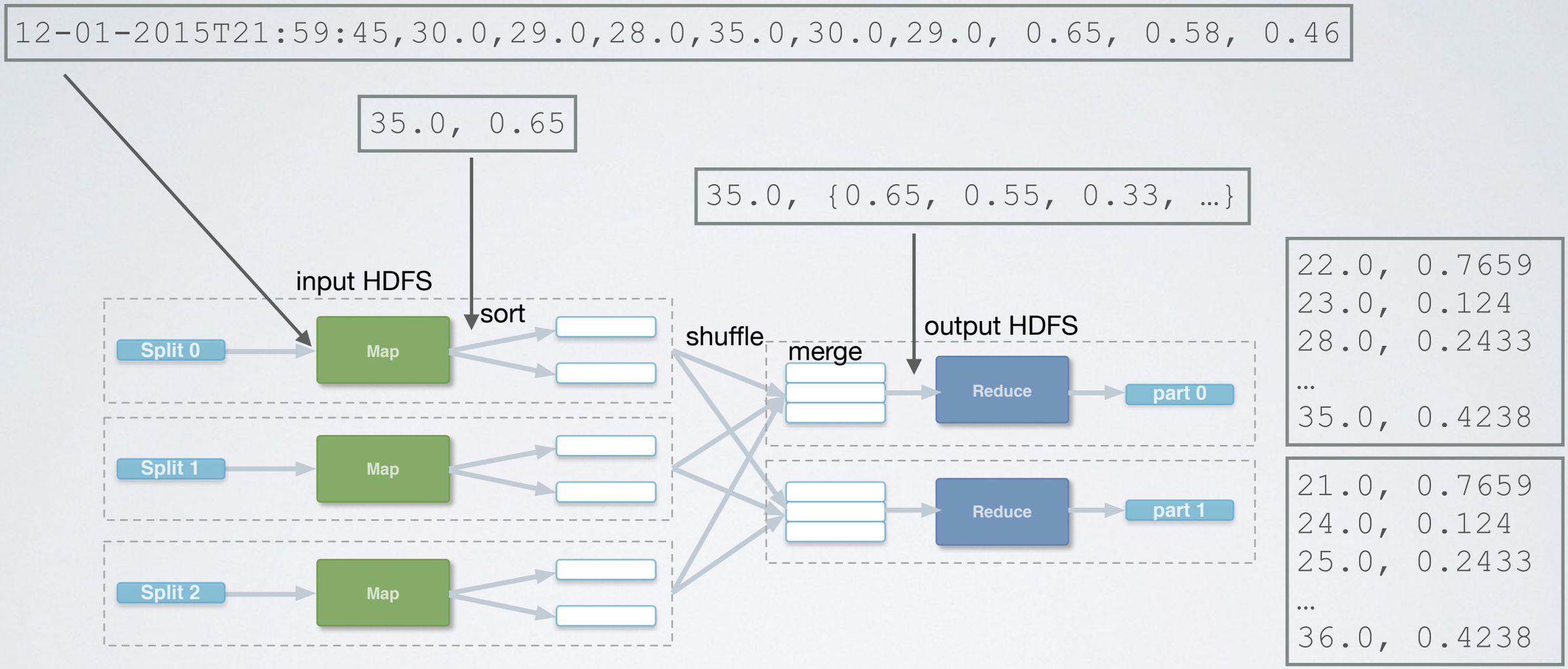
```
public void map(LongWritable ignore, Text text,
                OutputCollector<FloatWritable, FloatWritable> collector,
                Reporter reporter) {
    String line = text.toString();
    String[] tokens = line.split(",");
    if (tokens.length == 10) {
            // Find the highest CPU temperature
            Float maxTemp = 0.0F;
            for (int i = 1; i < 7; ++i) {
                Float temp = Float.parseFloat(tokens[i]);
                if (temp > maxTemp) {
                    maxTemp = temp;
                }
            }
            Float avgCpuLoad = Float.parseFloat(tokens[7]);
            outputCollector.collect(new FloatWritable(maxTemp),
                                    new FloatWritable(avgCpuLoad));
    }
}
```

REDUCER

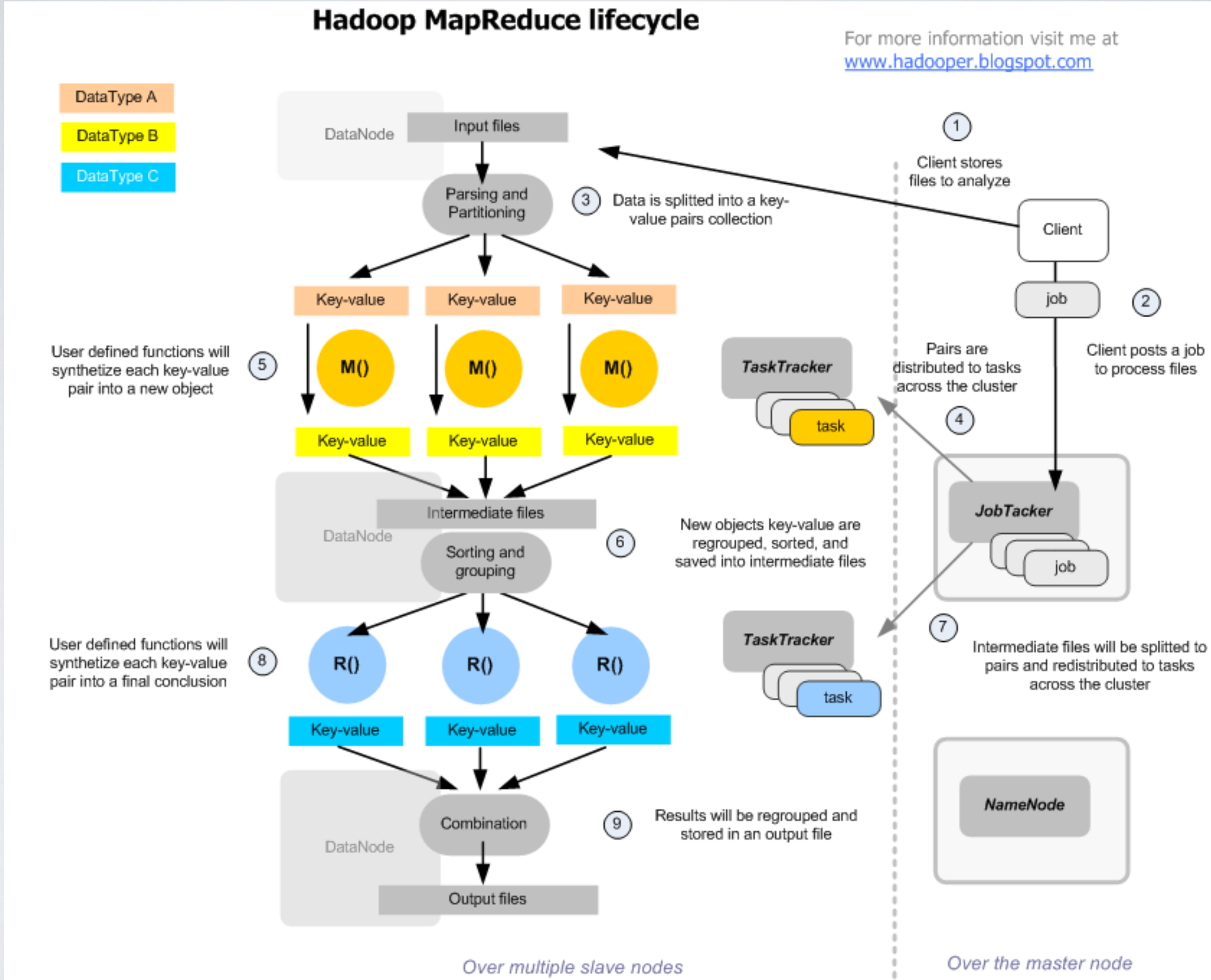
Remember from the Mapper: `OutputCollector<FloatWritable, FloatWritable>`

```
public void reduce(FloatWritable temp, Iterator<FloatWritable> cpuLoads,
                   OutputCollector<FloatWritable, FloatWritable> collector,
                   Reporter reporter) {
    int count = 0;
    Float sum = 0.0F;
    while (cpuLoads.hasNext()) {
        sum += cpuLoads.next().get();
        count++;
    }
    if (count > 0) {
        collector.collect(temp, new FloatWritable(sum / count));
    }
}
```

MAP-REDUCE (3)



run demo

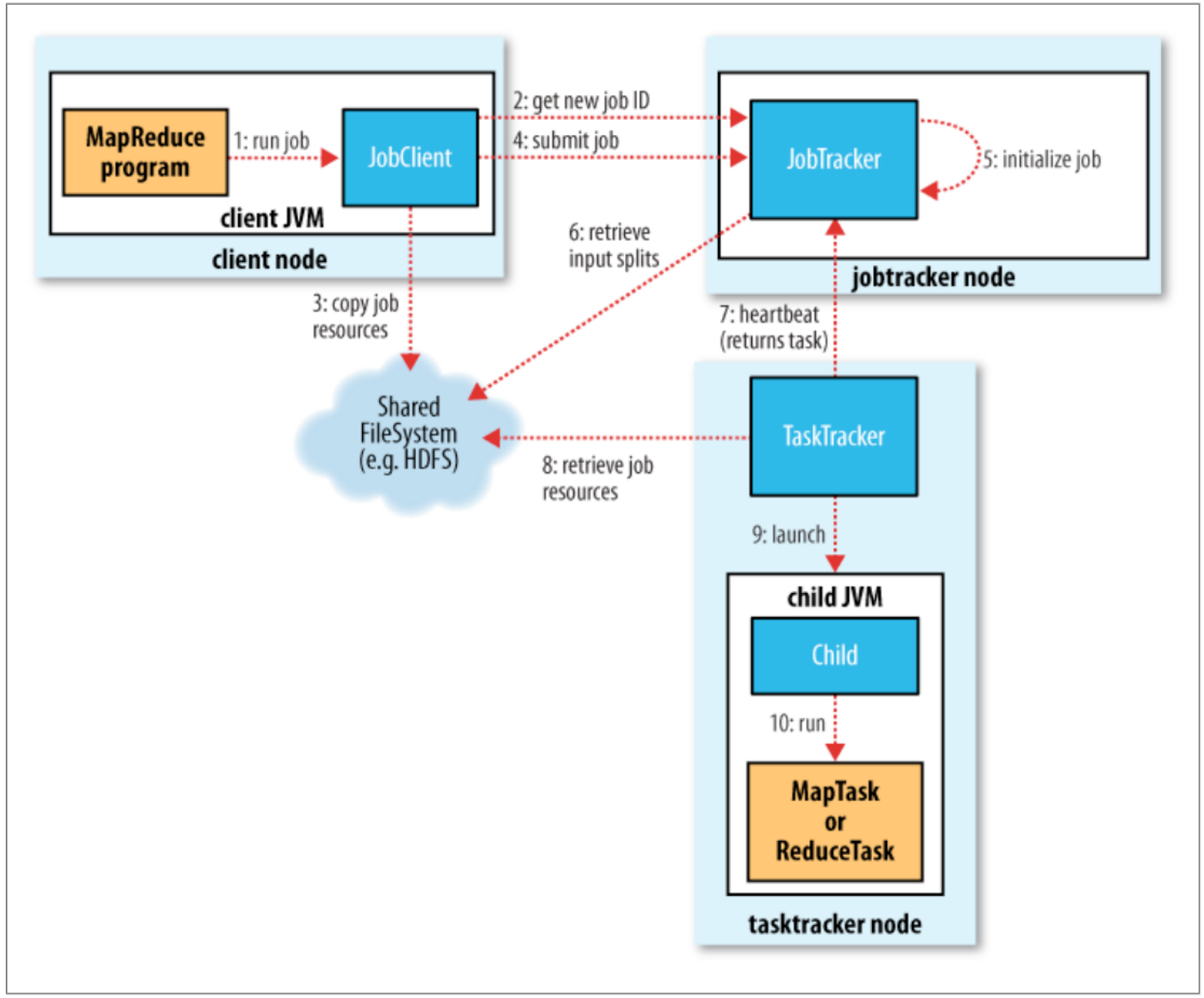


MapReduce tutorial:
http://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html

DON'T DO THIS AT HOME

- Setting up a Hadoop cluster is, like a FB status, *complicated*
- You also have to setup HDFS (the underlying filesystem) and, most likely, a NoSQL database to read data from / write results to
- Anything can fail (and, in fact, it will)
- Other concerns that are just as important: monitoring, failures recovery, alerting, and security (I'm assuming you care about your data, do you?)

RUNNING A MAP-REDUCE JOB



source: Hadoop - the Definitive Guide, O'Reilly

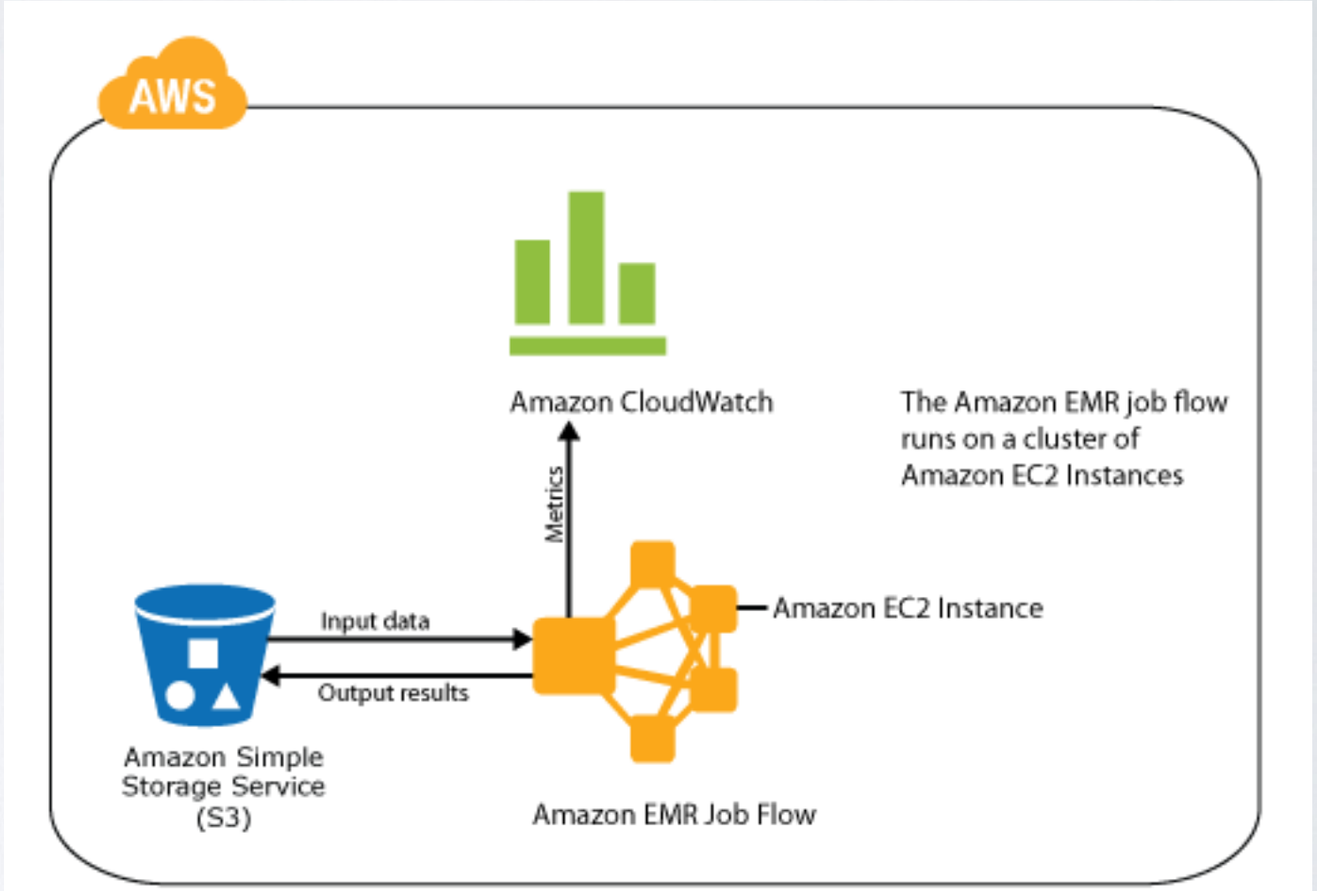
AWS ELASTIC MAPREDUCE

- "Amazon EMR simplifies running Hadoop and related big-data applications on AWS. You can use it to manage and analyze vast amounts of data. For example, a cluster can be configured to process petabytes of data."

- Put another way, they take the pain (and your money - but that's only for the compute resources, which you need anyway);

- It also integrates with Hive, Pig, HBase (and other AWS services)

- And a bunch of other commercial services (Cloudera, MapR) OR your in-house DevOps friendly folks will have automated it for you



http://docs.aws.amazon.com/ElasticMapReduce/latest/DeveloperGuide/emr-get-started.html

HDFS - BASICS

- Emphasis on Distributed - durability and availability are the key design concerns;
- Allows running large clusters, serving large data files on "commodity H/W" (note: *commodity* != *cheap*)
- Write-once, read-many-times the primary access pattern
- Not so good for: low-latency; many small files; frequent updates, especially random-access
- Blocks of 64MB (as compared to, eg, ext4: typically 4kB and 64kB max) to minimize the ratio of seek-to-transfer time: for example, at 100 MB/s transfer rate and 10ms seek time, at 64MB block size, we have seek time approx 1% of transfer time
- Files are broken in block-sized *chunks* and this allows us to store files that are larger than any of the disks in our cluster (by spreading the *chunks* around)
- Greatly simplifies the storage subsystem and management of metadata (a file is a sequence of blocks, and a block has a unique ID - metadata, if any, can be stored someplace else: this is similar to MongoDB's GridFS approach)

HDFS - READING / WRITING

Generally speaking, you can't care less that it's HDFS the underlying FS:

```
InputStream in = null;
try {
    in = new URL("hdfs://localhost/data/foo.dat").openStream();
    // do something with the InputStream
} finally {
    IOUtils.closeStream(in);
}
```

needs some work to make the JVM understand *"hdfs://"* but this is either done one-off or even taken care of by some third-party library.

HDFS - DEMO

Demo Usage of HDFS on localhost

HDFS - FILESYSTEM API

```
public static FileSystem get(Configuration conf)
public FSDataInputStream open(Path f)

FSDataInputStream extends DataInputStream implements Seekable, PositionedReadable
```

This means that, unlike your granmother's InputStreams, it supports random access and can be queried for the current position (`seek()` can move to an arbitrary position in the file) and parts of the file can be read at arbitrary offsets.

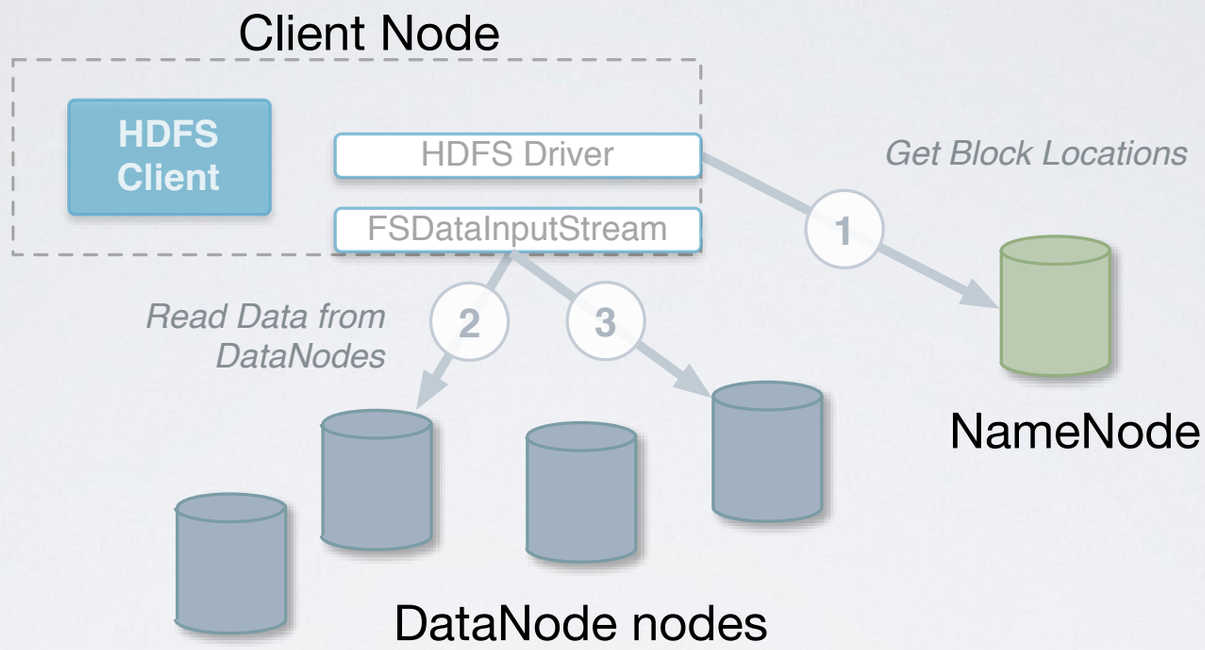
Example:

```
String filepath = args[0];
URI uri = URI.create(filepath);
Configuration conf = new Configuration();
FileSystem fs = FileSystem.get(uri, conf);
InputStream in = null;
try {
    in = fs.open(new Path(filepath));
    IOUtils.copyBytes(in, System.out, 4096, false);
} finally {
    IOUtils.closeStream(in);
}

// must be run via hadoop:

$ hadoop catFs hdfs://localhost/data/foo.txt
```

Seeking in a file is expensive!

https://gist.github.com/hellerbarde/2843375

HDFS - ARCHITECTURE

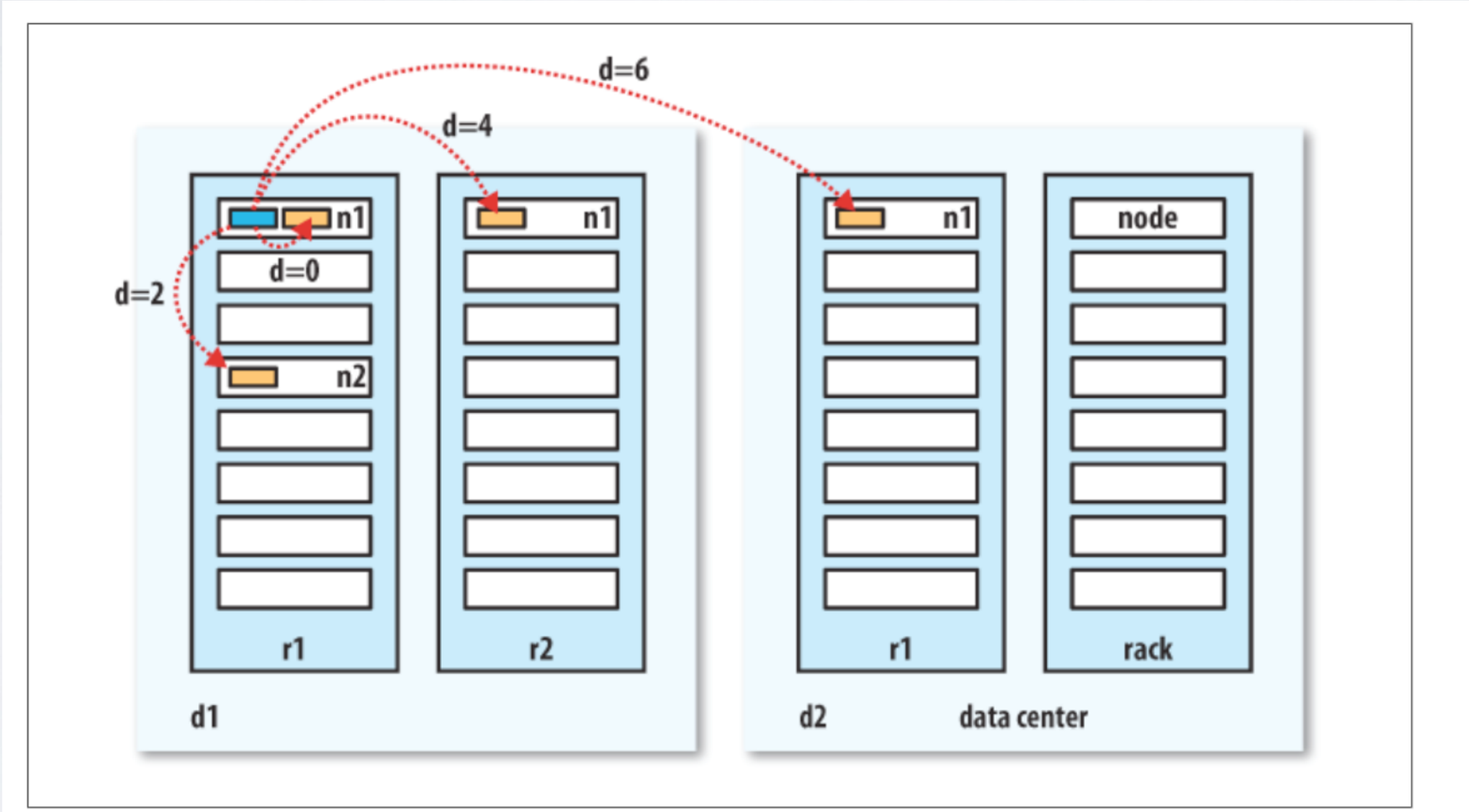


The NameNode and the DataNodes can, and usually will, live on different hosts and/or storage systems - all this is transparent to the Client that relies on the NameNode to locate the various *chunks* that make up a file, then read in (or write out) to the file via an `Input/OutputStream`.

However, each block lives on several DataNodes and their location will be returned in order of *proximity* to the client.

Writes go to the "closest" DataNode, which in turn replicates to other DataNodes, up until the required number of replicas.

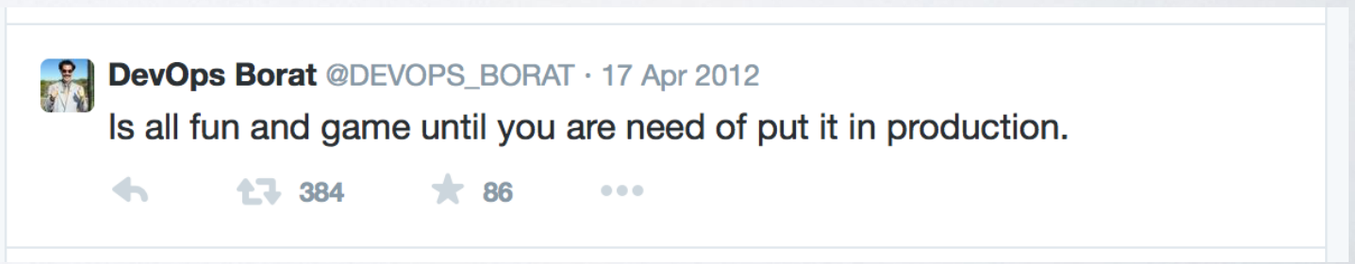
NETWORK PROXIMITY



source: Hadoop - the Definitive Guide, O'Reilly

WHAT COULD POSSIBLY GO WRONG?

- Any node can fail at any given time;
- Data may be corrupted (in transit and in storage);
- Networks may become "partitioned";
- Metadata (HDFS NameNode) may be lost or corrupted;
- TaskTrackers and JobTrackers can fail too;
- … and a lot of other fun stuff!



https://twitter.com/devops_borat

HADOOP COPES WITH A LOT OF THAT

- data chunks are replicated;
- **map** intrinsic high parallelism lends itself well to re-running failed ones (Spark's RDD take the same approach)
- *JobTracker* nodes can be replicated (and deployed in HA fashion)
- *NameNodes* can be replicated (or can have 'hot' standby nodes)
- Network "partitions" can be (to an extent) routed around, or dynamically re-configured (SDN helps a great deal here; but it's still very early days)
- **reduce**s are a lot less "resilient" (especially if they write to permanent storage and writes cannot be made idempotent) but there are ideally a lot less of them - however, the use of HDFS protects them somewhat from node failures
- ultimately, provided one call "roll-back" a failed MapReduce, and if all else fails, the whole process can be re-run - far from ideal, though.