UNB
FREDERICTON

Big Data Systems (CS4545/CS6545)
Winter 2021

MapReduce with Hadoop

Suprio Ray

University of New Brunswick, Fredericton

Acknowledgement

Thanks to B. Ramamurthy, Tom White (textbook), N. Venkatasubramanian, Z. Shao,
and for some of the materials in these slides. Also thanks to various articles and research papers.

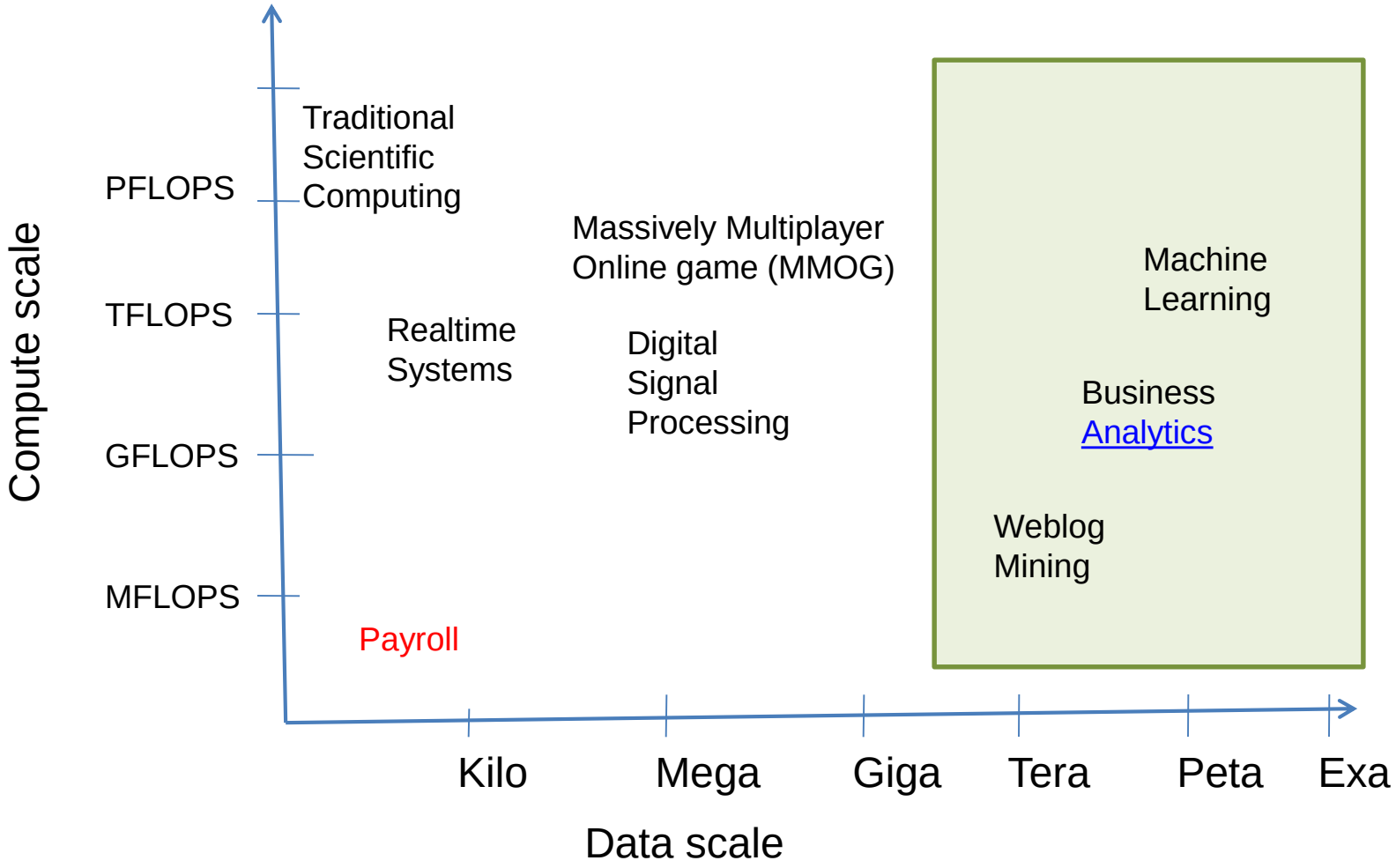
UNB
FREDERICTON

Outline

- Introduction 
- Developing a MapReduce Application
- How MapReduce Works
- YARN
- Hadoop Distributed File Systems (HDFS)
- MapReduce Algorithm Design

Problem Space

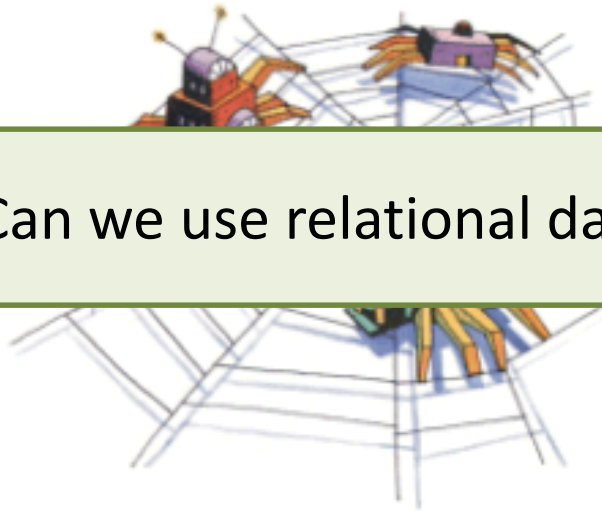
- Landscape of computational problems



Source: B. Ramamurthy

Solution space

- The question: **“How to process big data with reasonable cost and time?”**



Can we use relational databases?

RECALL: On to Google File

- Internet applications introduced a new challenge in the form web logs, web crawler's data: large scale "peta scale"

- But observe that this type of data has an uniquely different characteristic than your transactional or the "order" data on amazon.com: "write once"

- Google exploited this characteristics in its Google file system (GFS)

RECALL: Origins of "Modern" Big Data Systems

2003

The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung
Google*

→ hadoop HDFS

2004

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

→ hadoop MapReduce

2006

Bigtable: A Distributed Storage System for Structured Data

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach
Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber

{fay,jeff,sanjay,wilsonh,kerr,m3b,tushar,fikes,gruber}@google.com

Google, Inc.

Abstract

igtable is a distributed storage system for managing
tured data that is designed to scale to a very large
petabytes of data across thousands of commodity
rs. Many projects at Google store data in Bigtable,
ding web indexing, Google Earth, and Google Fi-
e. These applications place very different demands
igtable, both in terms of data size (from URLs to

achieved scalability and high performance, but Big
provides a different interface than such systems. Big
does not support a full relational data model; inste
provides clients with a simple data model that sup
dynamic control over data layout and format, an
lows clients to reason about the locality properties o
data represented in the underlying storage. Data i
dexed using row and column names that can be arbi
strings. Bigtable also treats data as uninterpreted str

→ APACHE HBASE

History: Hadoop's Developers

Doug Cutting

2005: Doug Cutting and Michael J. Cafarella developed Hadoop to support distribution for the Nutch search engine project.

The project was funded by Yahoo.

2006: Yahoo gave the project to Apache Software Foundation.

UNB FREDERICTON

What is Hadoop?

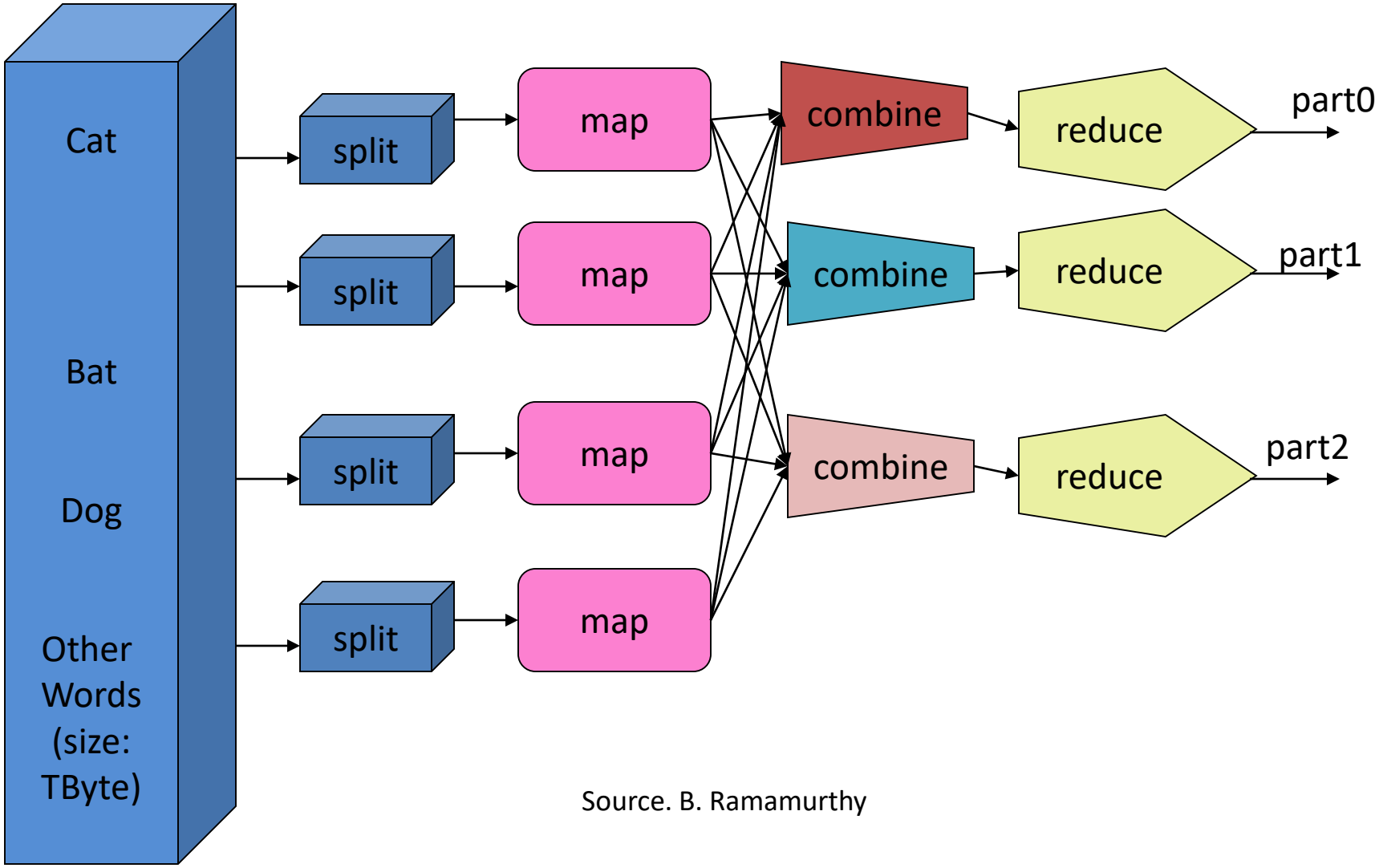
- **Hadoop:**
  - An open-source software framework that supports data-intensive distributed applications
- **Goals / Requirements:**
  - Abstract and facilitate the storage and processing of large and/or rapidly growing data sets
    - Structured and non-structured data
    - Simple programming models
  - High scalability and availability
  - Use commodity (cheap!) hardware with little redundancy
  - Fault-tolerance
  - Move computation rather than data

UNB FREDERICTON

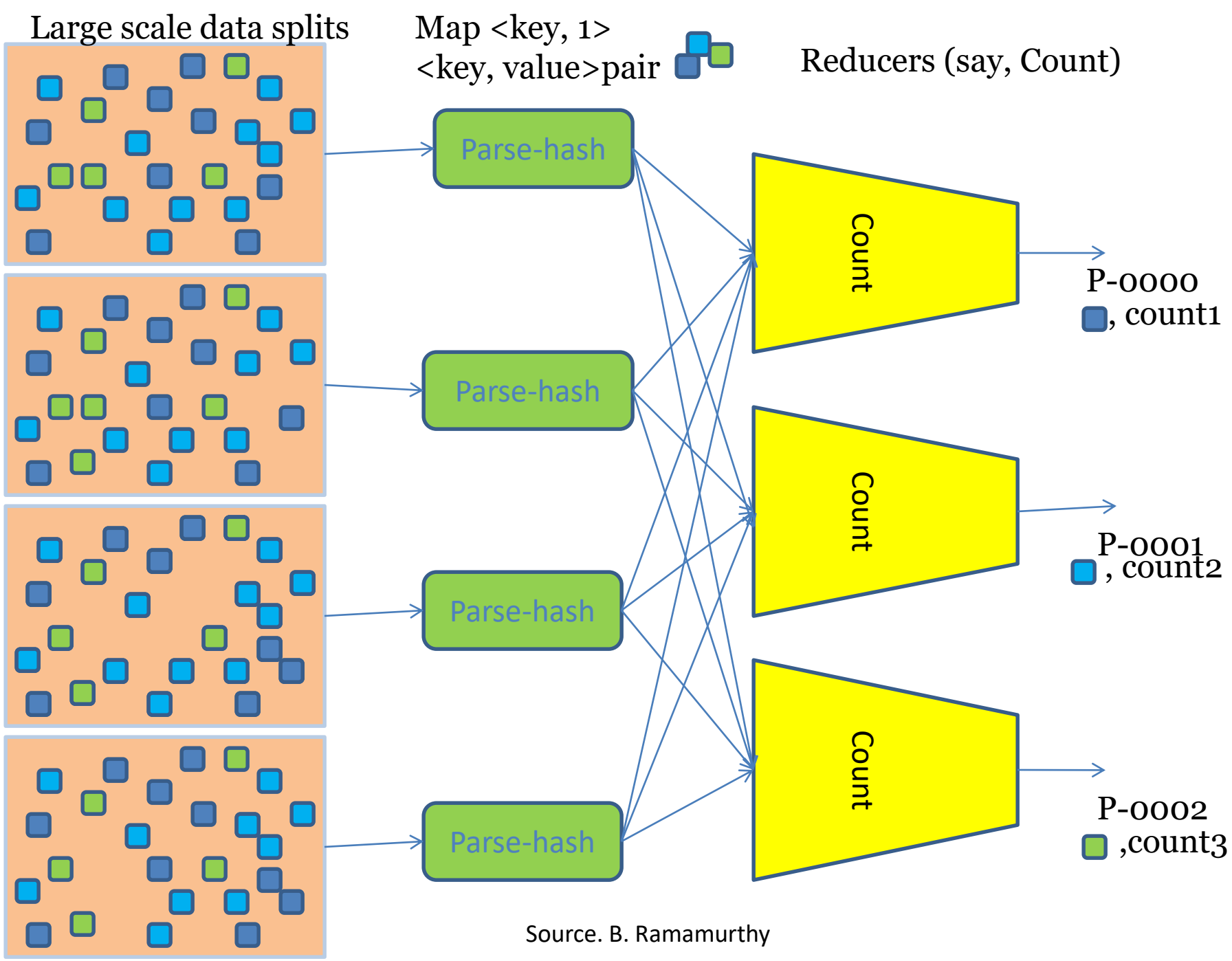
What is MapReduce?

- Programming model for distributed data processing
  - A map function extracts some intelligence from raw data.
  - A reduce function aggregates the data output by the map (according to some guides).
  - Users specify the computation in terms of a *map* and a *reduce* function,
  - Underlying runtime system automatically parallelizes the computation across large-scale clusters of machines, and
  - Underlying system also handles machine failures, efficient communications, and performance issues.

MapReduce



Source. B. Ramamurthy



Source. B. Ramamurthy

MapReduce example

Map creates key value pairs for each word and the number of times it appears in a text piece

……

Document 1

This is a cat

Cat sits on a roof

<this 1> <is 1> <a <1,1,>> <cat <1,1>> <sits 1> <on 1> <roof 1>

……

Document 2

The roof is a tin roof

There is a tin can on the roof

<the <1,1>> <roof <1,1,1>> <is <1,1>> <a <1,1>> <tin <1,1>> <then 1> <can 1> <on 1>

……

Document 3

Cat kicks the can

It rolls on the roof and falls on the next roof

<cat 1> <kicks 1> <the <1,1>> <can 1> <it 1> <roll 1> <on <1,1>> <roof <1,1>> <and 1> <falls 1> <next 1>

……

Document 4

The cat rolls too

It sits on the can

<the <1,1>> <cat 1> <rolls 1> <too 1> <it 1> <sits 1> <on 1> <cat 1>

MapReduce example

<this 1> <is 1> <a <1,1,>> <cat <1,1>> <sits 1> <on 1> <roof 1>

<the <1,1>> <roof <1,1,1>> <is <1,1>> <a <1,1>> <tin <1,1>> <then 1> <can 1> <on 1>

<cat 1> <kicks 1> <the <1,1>> <can 1> <it 1> <roll 1> <on <1,1>> <roof <1,1>> <and 1> <falls 1> <next 1>

<the <1,1>> <cat 1> <rolls 1> <too 1> <it 1> <sits 1> <on 1> <cat 1>

……

Combine the counts of all the same words:

<cat <1,1,1,1>>

<roof <1,1,1,1,1,1>>

<can <1, 1,1>>

……

Reduce (sum in this case) the counts:

<cat 4>

<can 3>

<roof 6>

UNB
FREDERICTON

Classes of problems “mapreducable”

- Benchmark for comparing: Jim Gray’s challenge on data-intensive computing. Ex: “Sort”
- Google uses it for adwords, pagerank, indexing data.
- Simple algorithms such as grep, text-indexing, reverse indexing
- Bayesian classification: data mining domain
- Facebook uses it for various operations: demographics
- Financial services use it for analytics
- Astronomy: Gaussian analysis for locating ET objects.
- Expected to play a critical role in semantic web and web3.0

UNB FREDERICTON

MapReduce Terminologies

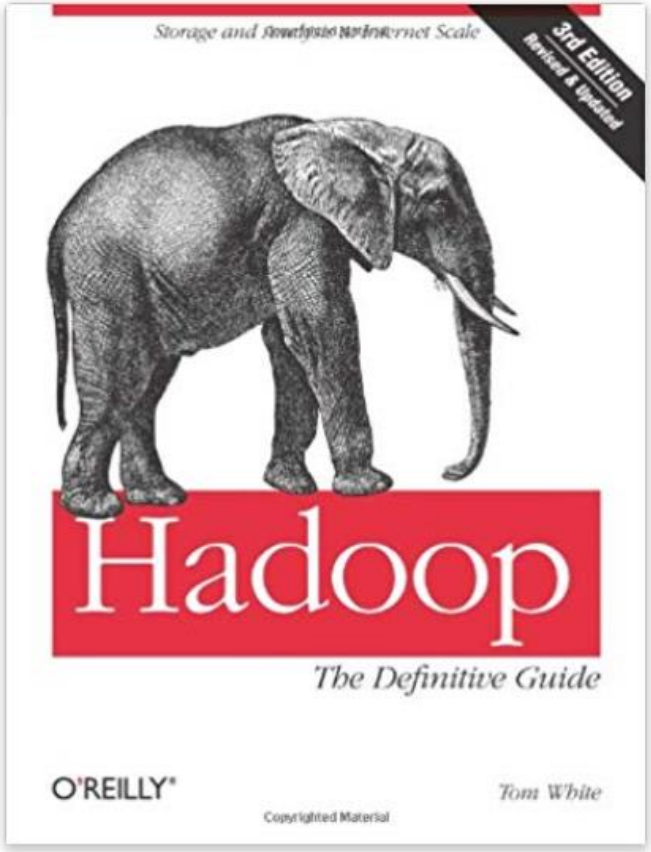
- Job: unit of work that client wants to be performed
  - Input Data
  - The MapReduce program
  - Configuration information

- Task
  - A job is divided into tasks: map task and reduce task

- Split
  - Hadoop divides the input into fixed size pieces
  - Hadoop creates one map task for each split, which runs the user-defined function for each record in the split
  - Hadoop tries to run the map task on a node where the input data resides: *data locality optimization*

Outline

- Introduction
- Developing a MapReduce Application
- How MapReduce Works
- YARN
- Hadoop Distributed File Systems
- HIVE
- File-based Data Structures



Please refer to the text for materials

UNB FREDERICTON

Writing a MapReduce Program

- Write your map and reduce functions
- Write a driver program to run a job locally
  - can run from an IDE with a small subset of the data to check that it is working
- Use the IDE's debugger to find the source of the problem (if any)
- Once everything works in local mode, run it in the cluster mode

Writing a MapReduce Program

- General form
  - The map input key and value types (K1 and V1) are different from the map output types (K2 and V2)
  - The reduce input must have the same output types as the map output
  - The reduce output type may be different (K3 and V3)

```
map: (K1, V1) → list(K2, V2)

reduce: (K2, list(V2)) → list(K3, V3)
```

  - Example:

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

    Reduce input: (1950, [0, 10, 15, 20, 25])

    Reduce output: (1950, 25)

Writing a MapReduce Program

- General form of the Mapper program

```
public class Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {
   public class Context extends MapContext<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {
      // ...
   }

   protected void map(KEYIN key, VALUEIN value, Context context) throws IOException,
            InterruptedException {
      // ...
   }
}
```

Writing a MapReduce Program

- General form of the Reducer program

```
public class Reducer<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {
   public class Context extends ReducerContext<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {
      // ...
   }

   protected void reduce(KEYIN key, Iterable<VALUEIN> values, Context context) throws
            IOException, InterruptedException {
      // ...
   }
}
```

Writing a MapReduce Program

- The context object
  - Used for emitting key-value pairs
  - Parameterized by the output types
  - The signature of the write() method is:

```
public void write(KEYOUT key, VALUEOUT value)
        throws IOException, InterruptedException
```

Installing Hadoop

- Download Hadoop package and decompress

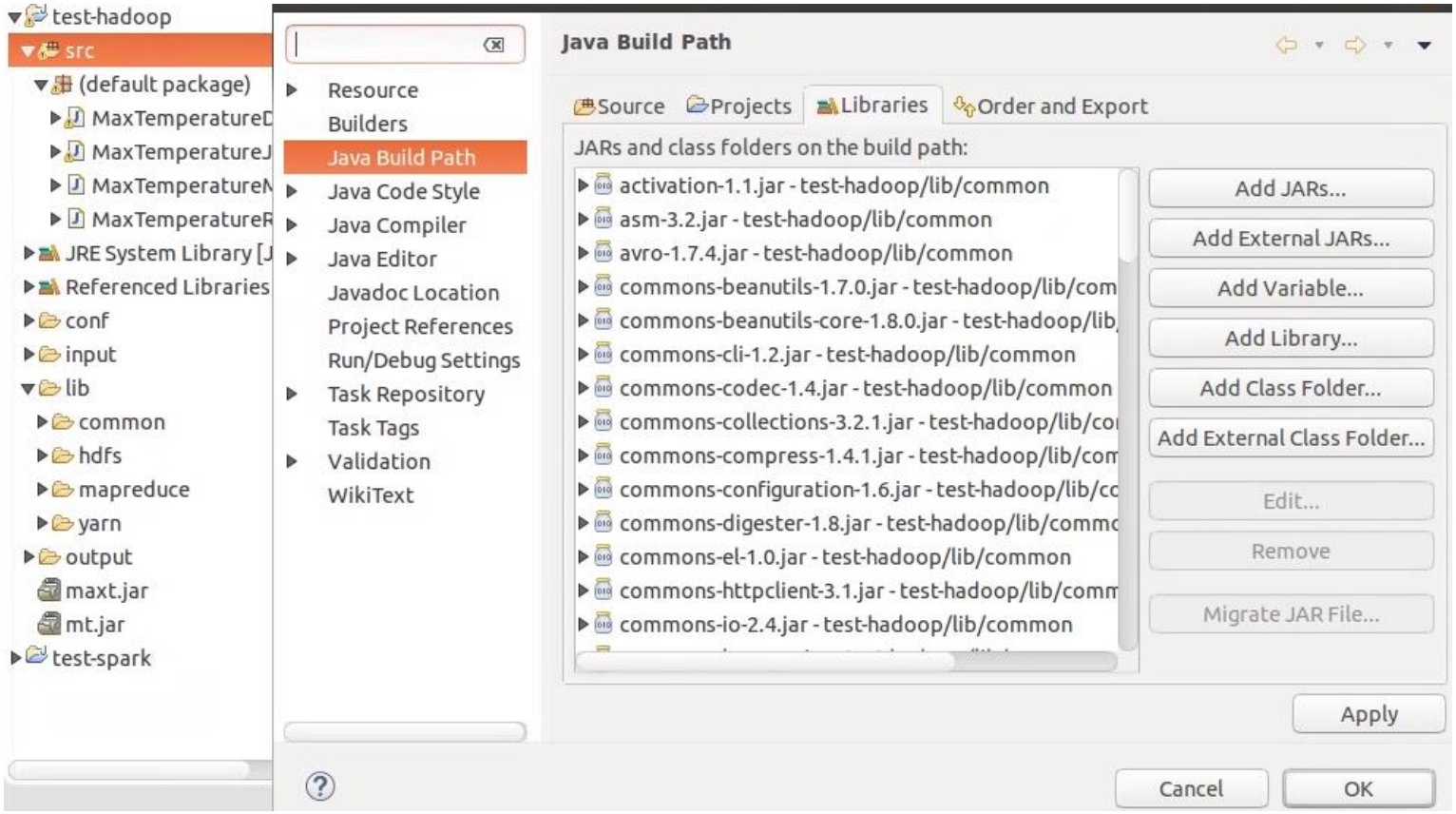
```
dbuser@srlaptop:~/hadoop$ ls
hadoop-2.3.0  hadoop-2.3.0.tar.gz
```

- Install Hadoop
  - Config files under ./etc/hadoop

```
dbuser@srlaptop:~/hadoop/hadoop-2.3.0/etc/hadoop$ ls
capacity-scheduler.xml      hdfs-site.xml              mapred-site.xml
configuration.xsl           httpfs-env.sh              slaves
container-executor.cfg      httpfs-log4j.properties    ssl-client.xml.example
core-site.xml               httpfs-signature.secret    ssl-server.xml.example
hadoop-env.cmd              httpfs-site.xml            temp
hadoop-env.sh               log4j.properties           yarn-env.cmd
hadoop-metrics2.properties  mapred-env.cmd            yarn-env.sh
hadoop-metrics.properties   mapred-env.sh             yarn-site.xml
hadoop-policy.xml           mapred-queues.xml.template
```

Setting Up the Development Environment

- Create an Eclipse project and add all .jar files from Hadoop installation



The dataset

- National Climatic Data Center, or NCDC
  - Data is stored using a line-oriented ASCII format
  - Each line is a record
  - Supports a rich set of meteorological elements, many of which are optional or with variable data
  - We focus on the basic elements, such as temperature

```
dbuser@srlaptop:~/datasets/noaa/ncdc_data$ ls
1901.gz  1903.gz  1905.gz  1907.gz  1909.gz  1911.gz  1913.gz  1915.gz  1917.gz  1919.gz
1902.gz  1904.gz  1906.gz  1908.gz  1910.gz  1912.gz  1914.gz  1916.gz  1918.gz  1920.gz
```

The dataset

- Record format

0029029070999991901010106004+64333+023450FM-12+000599999V0202701N0
15919999999N0000001N9-00781+99999102001ADDGF108991999999999999999999

- Position and description of record fields in the example:

POS: 16-23

GEOPHYSICAL-POINT-OBSERVATION date
The date of a GEOPHYSICAL-POINT-OBSERVATION.
MIN: 00000101 MAX: 99991231
DOM: A general domain comprised of integer values 0-9 in the format YYYYMMDD
YYYY can be any positive integer value; MM is restricted to values 01-12;

POS: 88-92

AIR-TEMPERATURE-OBSERVATION air temperature
The temperature of the air.
MIN: -0932 MAX: +0618 UNITS: Degrees Celsius
SCALING FACTOR: 10
DOM: A general domain comprised of the numeric characters (0-9), a plus sign (+), and a minus sign (-).
+9999 = Missing.

Write the mapper class

```
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.Mapper;
public class MaxTemperatureMapper extends  Mapper<LongWritable, Text, Text, IntWritable> {
        private static final int MISSING = 9999;
         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
                        airTemperature = Integer.parseInt(line.substring(88, 92)); //signs
                } else {
                        airTemperature = Integer.parseInt(line.substring(87, 92));
                }
                String quality = line.substring(92, 93);
                if (airTemperature != MISSING)  {
                        context.write(new Text(year), new
                        IntWritable(airTemperature));
                }
        }
}
```

Write the reducer class

```
import java.io.IOException;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.Reducer;

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

        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));
        }
}
```

Run it

- Start Hadoop processes with start-all.sh
  - In turn runs start-dfs.sh and start-yarn.sh

```
dbuser@srlaptop:~/eclipse/workspace/test-hadoop$ start-all.sh
This script is Deprecated. Instead use start-dfs.sh and start-yarn.sh
Starting namenodes on [srlaptop]
Starting secondary namenodes [0.0.0.0]
starting yarn daemons
starting resourcemanager
starting nodemanager
starting nodemanager
```

Run it

- Check the running processes using ‘jps’ command

```
dbuser@srlaptop:~/eclipse/workspace/test-hadoop$ jps
1389 Jps
30891 NameNode
31253 SecondaryNameNode
31422 ResourceManager
31565 NodeManager
31052 DataNode
21830 RunJar
4716 org.eclipse.equinox.launcher_1.3.100.v20150511-1540.jar
```

Running on a cluster: the driver class

```
public class MaxTemperatureJobRunner {
   public int run(String[] args) throws Exception  {
      Job job = new Job();
      String userJarLocation = "/home/dbuser/eclipse/workspace/test-hadoop/maxt.jar";
      job.setJar(userJarLocation);      //alternative
      job.setJarByClass(MaxTemperatureJobRunner.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);
      boolean success = job.waitForCompletion(true);
      return success ? 0 : 1;
   }
  public static void main(String[] args) throws Exception {
      MaxTemperatureJobRunner driver = new MaxTemperatureJobRunner();
      driver.run(args);
   }
}
```

Running on a cluster

- Packaging the jar file
  - A job's classes must be packaged into a *job JAR file* to send to the cluster
  - Hadoop will find the job JAR automatically by searching for the JAR on the driver's classpath that contains the class set in the setJarByClass()

- Launching a job

```
$ hadoop jar ./maxt.jar MaxTemperatureJobRunner  ./input ./output

16/02/08 17:55:55 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1454960982536_0003
16/02/08 17:55:55 INFO impl.YarnClientImpl: Submitted application application_1454960982536_0003
16/02/08 17:55:55 INFO mapreduce.Job: The url to track the job: http://srlaptop:8088/proxy/application_1454960982536_0003/
16/02/08 17:55:55 INFO mapreduce.Job: Running job: job_1454960982536_0003
16/02/08 17:56:00 INFO mapreduce.Job: Job job_1454960982536_0003 running in uber mode : false
16/02/08 17:56:00 INFO mapreduce.Job:  map 0% reduce 0%
16/02/08 17:56:05 INFO mapreduce.Job:  map 100% reduce 0%
16/02/08 17:56:11 INFO mapreduce.Job:  map 100% reduce 100%
16/02/08 17:56:12 INFO mapreduce.Job: Job job_1454960982536_0003 completed successfully
```

Check the output

- Look what's inside the output folder

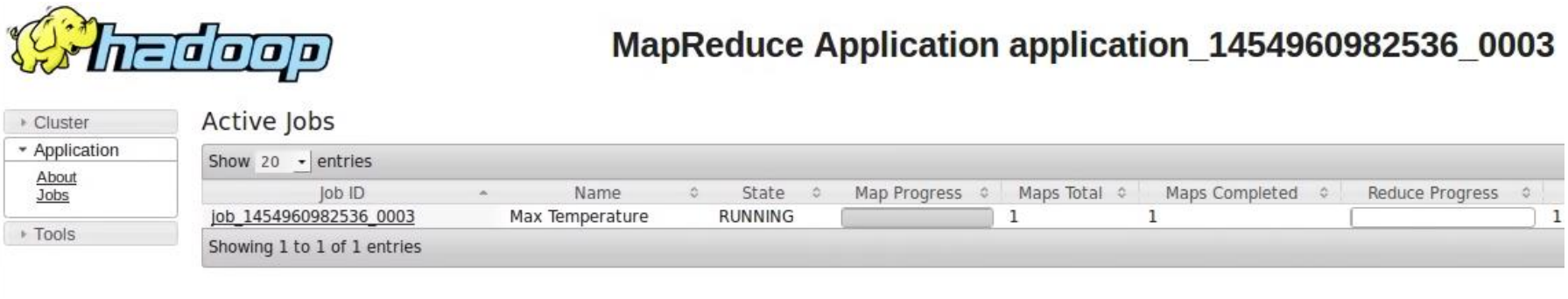
```
$ hadoop dfs -ls /user/dbuser/op
Found 2 items
-rw-r--r--   2 dbuser supergroup          0 2016-02-08 15:54 /user/dbuser/op/_SUCCESS
-rw-r--r--   2 dbuser supergroup         63 2016-02-08 15:54 /user/dbuser/op/part-r-00000
```

- The output file

```
$ hadoop dfs -cat /user/dbuser/op/part-r-00000

1901    317
1902    244
1903    289
1904    256
1905    283
1906    294
1907    283
```

The MapReduce job page

- While the job is running, we can monitor its progress on this page.
  - Shows the map progress and the reduce progress
  - "Total" shows the total number of map and reduce tasks for this job (a row for each)



Running jobs locally

- Tool and ToolRunner
  - Helper classes: make it easier to run jobs from command line
  - Extends the Tool interface
  - Run your application with ToolRunner

```
public interface Tool extends Configurable {
  int run(String [] args) throws Exception;
}
```

```
public class ConfigurationPrinter extends Configured implements Tool {

@Override
public int run(String[] args) throws Exception {
  Configuration conf = getConf();
  for (Entry<String, String> entry: conf) {
    System.out.printf("%s=%s\n", entry.getKey(), entry.getValue());
  }
  return 0;
}

public static void main(String[] args) throws Exception {
  int exitCode = ToolRunner.run(new ConfigurationPrinter(), args);
  System.exit(exitCode);
}
}
```

Writing a MapReduce program: putting it all together

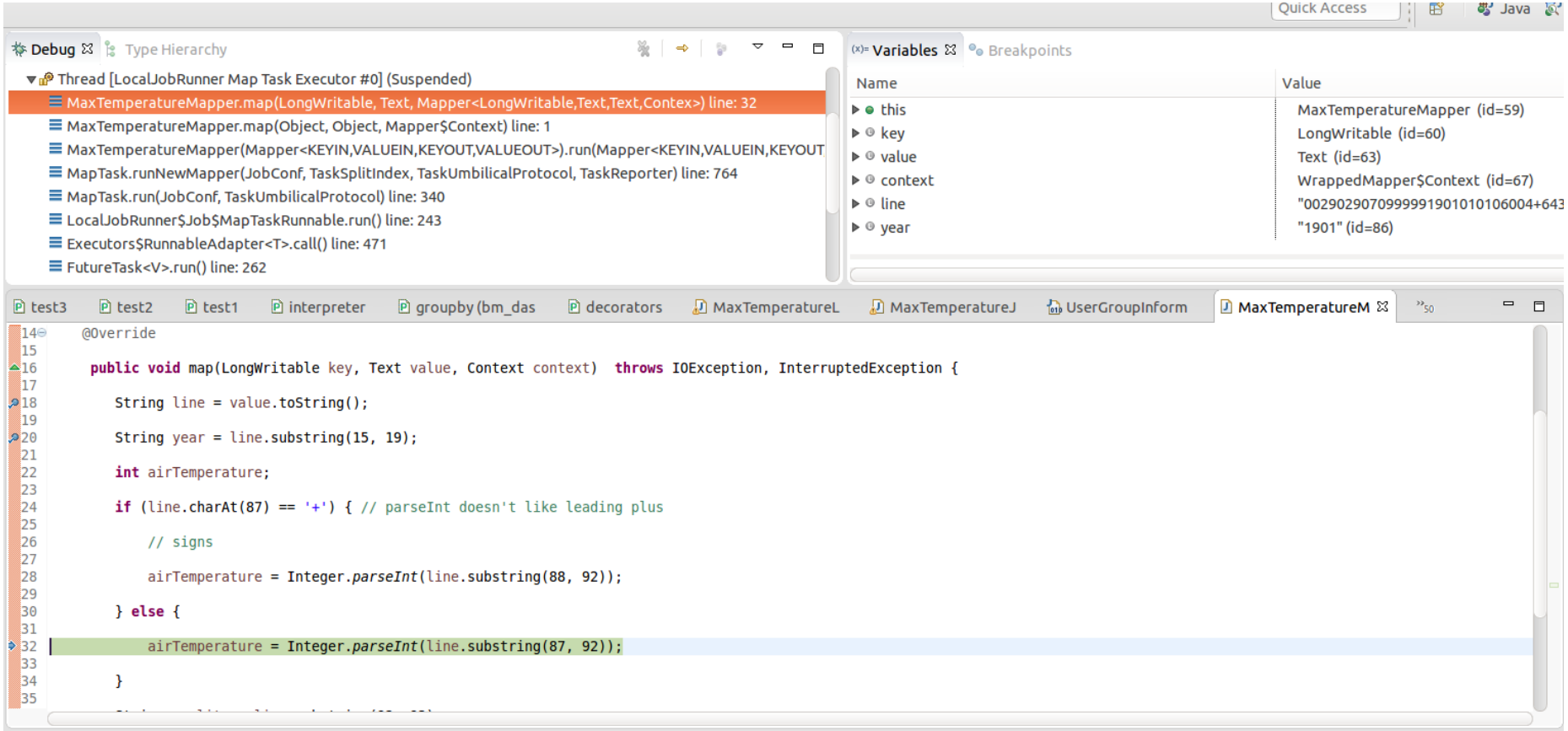
- Create a launching program for your application
- The launching program configures:
  - The *Mapper* and *Reducer* classes to use
  - The output key and value types (input types are inferred from the *InputFormat*)
  - The locations for your input and output
- The launching program then submits the job and typically waits for it to complete

Write the driver class to run/debug locally

```
public class MaxTemperatureDriver extends Configured implements Tool {
   public int run(String[] args) throws Exception  {
      Job job = new Job();
      job.setJar("/home/dbuser/eclipse/workspace/test-hadoop/maxt.jar"); //optional
      job.setJarByClass(MaxTemperatureDriver.class);
      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);
      boolean success = job.waitForCompletion(true);
      return success ? 0 : 1;
   }
   public static void main(String[] args) throws Exception {
      MaxTemperatureDriver driver = new MaxTemperatureDriver();
      int exitCode = ToolRunner.run(driver, args);
      System.exit(exitCode);
   }
}
```

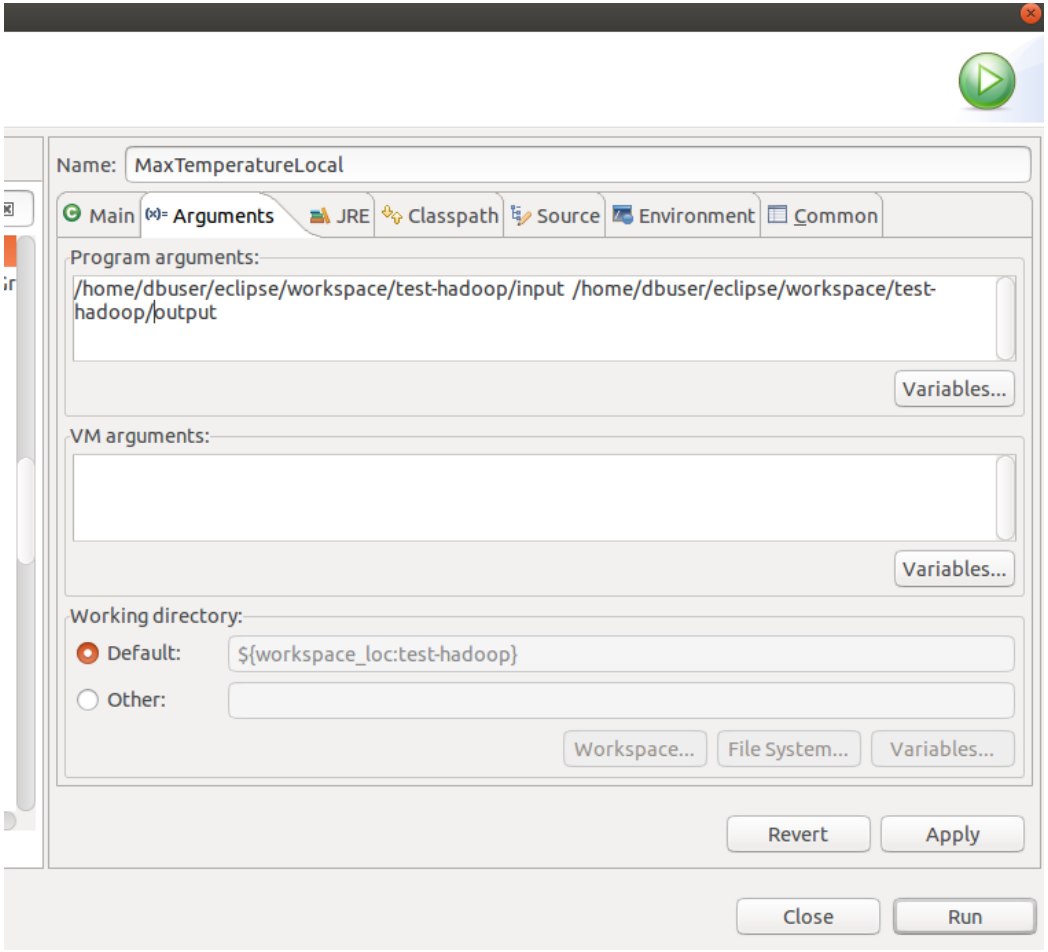
Running with Eclipse

- The program can be run from within Eclipse



Running with Eclipse

- The program can be run from within Eclipse
  - Make sure to specify the program arguments

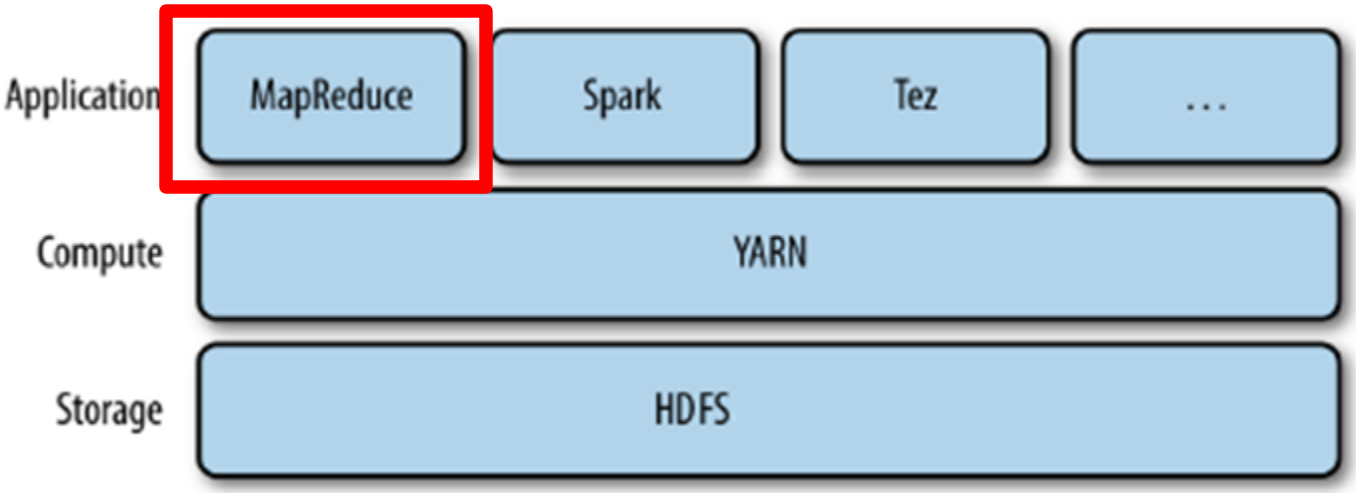


UNB FREDERICTON

Outline

- Introduction
- Developing a MapReduce Application
- How MapReduce Works 
- YARN
- Hadoop Distributed File Systems (HDFS)
- MapReduce Algorithm Design

MapReduce


Application
MapReduce
Spark
Tez
...
Compute
YARN
Storage
HDFS

Revisiting: General form of MapReduce

- General form
  - The map input key and value types (K1 and V1) are different from the map output types (K2 and V2)
  - The reduce input must have the same types as the map output
  - The reduce output type may be different (K3 and V3)

```
map: (K1, V1) → list(K2, V2)

reduce: (K2, list(V2)) → list(K3, V3)
```

  - Example:

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

    Reduce input: (1950, [0, 10, 15, 20, 25])

    Reduce output: (1950, 25)

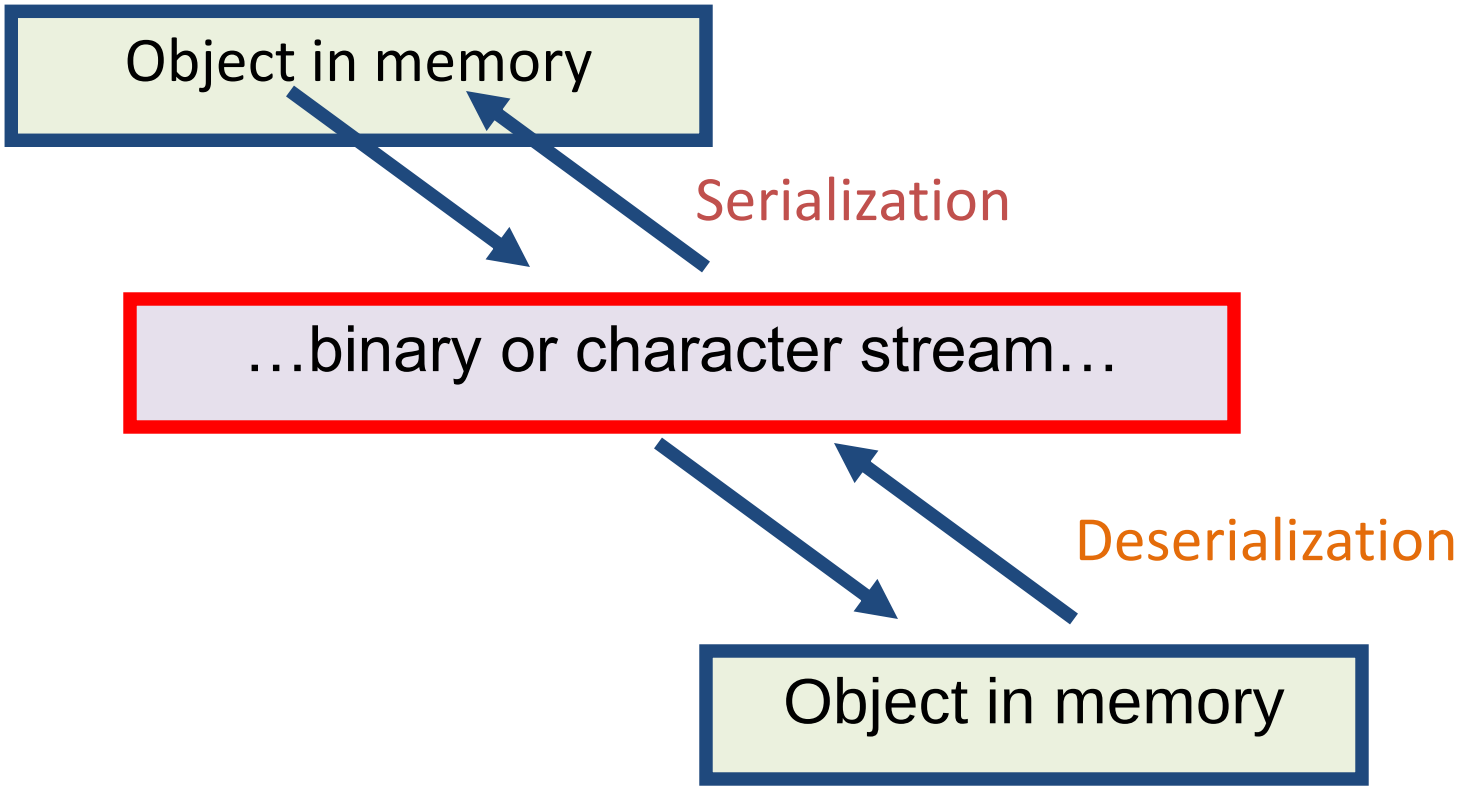
Revisiting: write the mapper class

```
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.Mapper;
public class MaxTemperatureMapper extends  Mapper<LongWritable, Text, Text, IntWritable> {
        private static final int MISSING = 9999;
         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
                        airTemperature = Integer.parseInt(line.substring(88, 92)); //signs
                } 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));
                }
        }
}
```

Revisiting: write the reducer class

```
import java.io.IOException;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.Reducer;

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

        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));
        }
}
```

Serialization/Deserialization



UNB FREDERICTON

What is Writable?

- Hadoop defines its own “box” classes for **serialization**
  - strings *(Text),* integers *(IntWritable)*, etc.

- All keys are instances of *WritableComparable*

```
IntWritable w1 = new IntWritable(163);
IntWritable w2 = new IntWritable(67);
assertThat(comparator.compare(w1, w2), greaterThan(0));
```

- All values are instances of *Writable*
  - Text is a Writable for UTF-8 sequences (equivalent of java.lang.String)

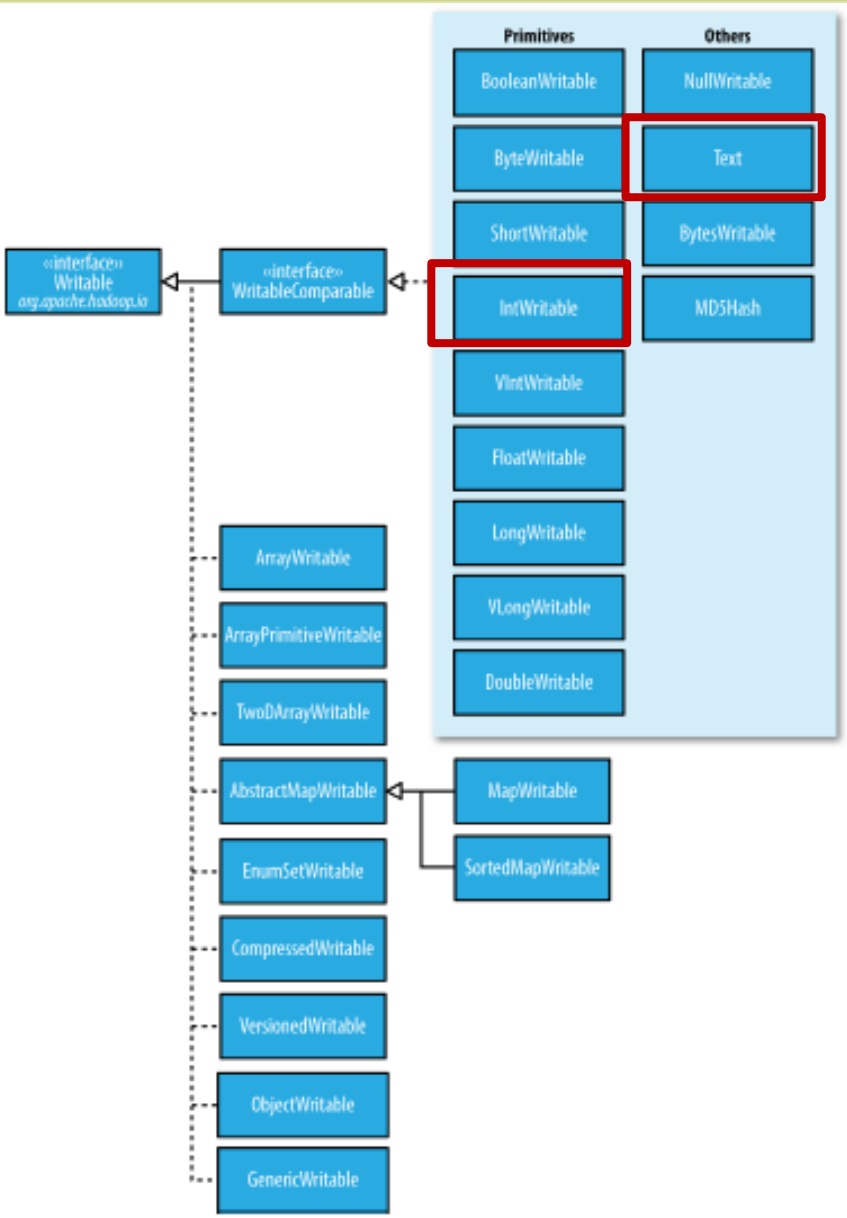
UNB FREDERICTON

Writable wrapper for Java primitives

- There are Writable wrapper for all Java primitive types

Java primitive	Writable implementation	Serialized size (bytes)
boolean	BooleanWritable	1
byte	ByteWritable	1
short	ShortWritable	2
int	IntWritable	4
	VIntWritable	1–5
float	FloatWritable	4
long	LongWritable	8
	VLongWritable	1–9
double	DoubleWritable	8

Writable class hierarchy



UNB
FREDERICTON

Javadoc

org.apache.hadoop.io

Class IntWritable

java.lang.Object
    org.apache.hadoop.io.IntWritable

All Implemented Interfaces:

Comparable<IntWritable>, Writable, WritableComparable<IntWritable>

org.apache.hadoop.io

Class Text

java.lang.Object
    org.apache.hadoop.io.BinaryComparable
        org.apache.hadoop.io.Text

All Implemented Interfaces:

Comparable<BinaryComparable>, Writable, WritableComparable<BinaryComparable>

RECALL: Writing a MapReduce program: putting it all together

- Create a launching program for your application
- The launching program configures:
  - The *Mapper* and *Reducer* classes to use
  - The output key and value types (input types are inferred from the *InputFormat*)
  - The locations for your input and output
- The launching program then submits the job and typically waits for it to complete

Revisiting: write the driver/launcher class

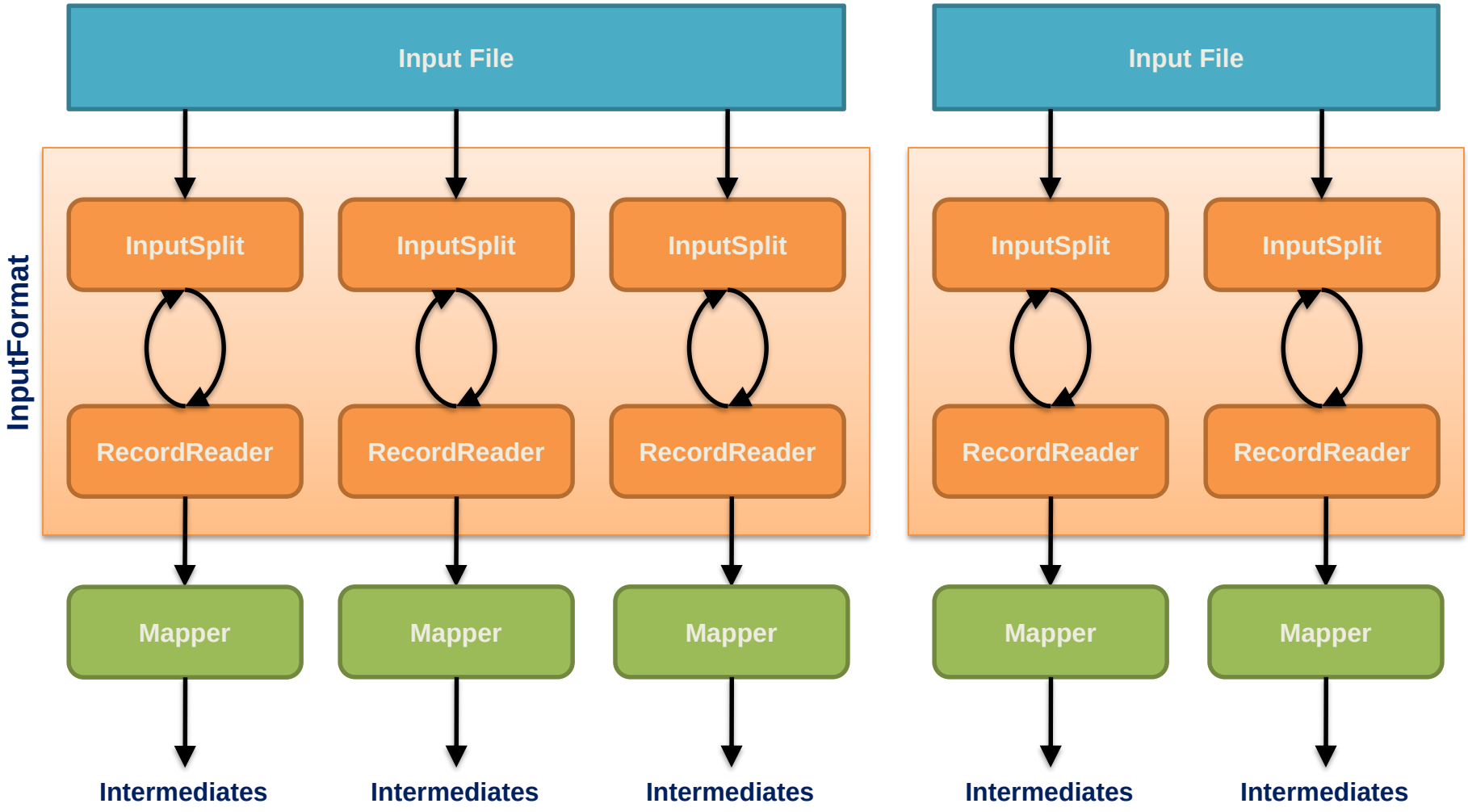
```
public class MaxTemperatureDriver extends Configured implements Tool {
   public int run(String[] args) throws Exception  {
      Job job = new Job();
      job.setJar("/home/dbuser/eclipse/workspace/test-hadoop/maxt.jar");
      job.setJarByClass(MaxTemperatureDriver.class);
      FileInputFormat.addInputPath(job, 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);
      boolean success = job.waitForCompletion(true);
      return success ? 0 : 1;
   }
   public static void main(String[] args) throws Exception {
      MaxTemperatureDriver driver = new MaxTemperatureDriver();
      int exitCode = ToolRunner.run(driver, args);
      System.exit(exitCode);
   }
}
```

UNB FREDERICTON

Reading Data

- A Map/Reduce may specify how it's input is to be read by specifying an *InputFormat* to be used
  - Defines input data (e.g., a directory)
  - Identifies partitions of the data that form an *InputSplit*
  - Factory for *RecordReader* objects to extract (k, v) records from the input source
  - **Default** to *TextInputFormat*
  - Can be set using *job.setInputFormatClass(…)*

Input to the Mapper



Source: redrawn from a slide by Cloduera, cc-licensed

UNB FREDERICTON

Specifying Input Formats

- The Hadoop framework provides a large variety of input formats.
  - KeyValueTextInputFormat: Key/value pairs, one per line.
  - TextInputFormat: The key is the byte offset of the beginning of the line within the file, and the value is the line.
  - NLineInputFormat: Similar to TextInputFormat, but the splits are based on *N* lines of input rather than *Y* bytes of input.
  - SequenceFIleInputFormat: The input file is a Hadoop sequence file, which is a binary file of (k, v) pairs with some additional metadata

UNB FREDERICTON

Example of TextInputFormat

- The TextInputFormat: the key is the byte offset within the file of the beginning of the line, and the value is the line
  - A file containing the following text:

```
On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.
```

  - Gets interpreted as:

```
(0, On the top of the Crumpetty Tree)
(33, The Quangle Wangle sat,)
(57, But his face you could not see,)
(89, On account of his Beaver Hat.)
```

Example of NLineInputFormat

- The NLineInputFormat: the key is the byte offset for each line, and the value is the number of lines specified as input

  - Code in the driver class to specify input format

```
job.setInputFormatClass(NLineInputFormat.class); NLineInputFormat.addInputPath(job, new Path(args[0])); job.getConfiguration().setInt("mapreduce.input.lineinputformat.linespermap", 2);
```

  - Input: a file containing the following text

```
2015-8-02
error2014 blahblahblahblah
2015-8-02
blahblahbalh error2014
2015-8-03
err2015 abracadabra
2015-8-03
abracadabra err2015
```

  - Output

Mapper0 output

0 2015-8-02
10 error2014 blahblahblahblah

Mapper1 output

37 2015-8-02
47 blahblahbalh error2014

. . . . .

Example of KeyValueTextInputFormat

- The KeyValueTextInputFormat: Key/value pairs, one per line, a separator needs to be specified

  - Input: a file containing the following text:

```
one,first line
two,second line
three,third line
four,fourth line
```

  - Output:

one	first line
two	second line
three	third line
four	fourth line

Example of KeyValueTextInputFormat (cont.)

- The KeyValueTextInputFormat: Key/value pairs, one per line, a separator needs to be specified
  - Code in the mapper class

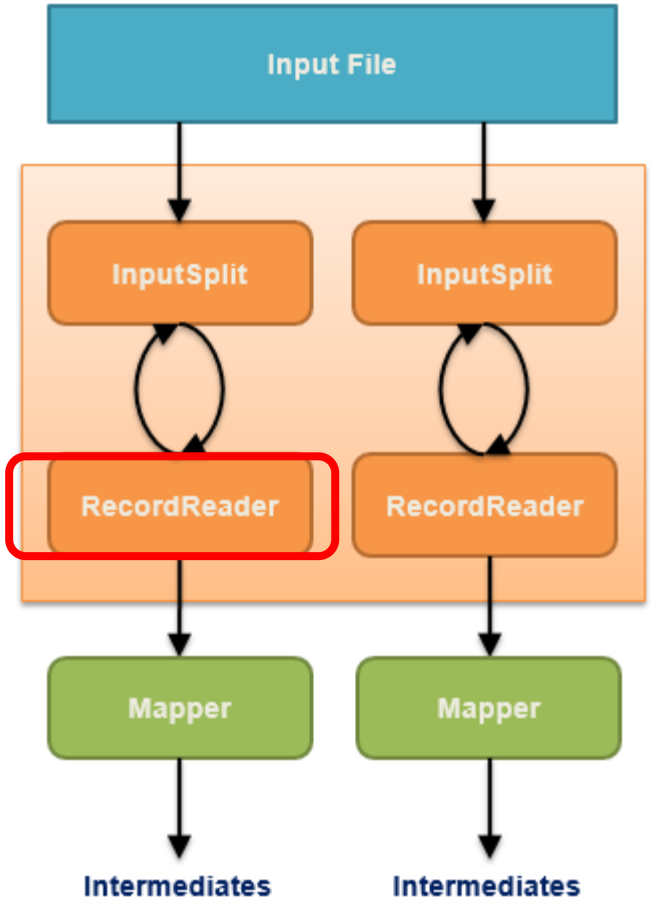
```
public class KVInputformatMapper extends Mapper<Text, Text, Text, Text> {//keyin, valuein, keyout, valueout

 @Override
   public void map(Text key, Text value, Context context) throws IOException, InterruptedException {
      context.write(key, value);
   }
}
```

  - Code in the driver class to specify input format

```
job.getConfiguration().set("mapreduce.input.keyvaluelinerecordreader.key.value.separator",",");
job.setInputFormatClass(KeyValueTextInputFormat.class);
KeyValueTextInputFormat.addInputPath(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(Text.class);
job.setMapperClass(KVInputformatMapper.class);
job.setNumReduceTasks(0);
```

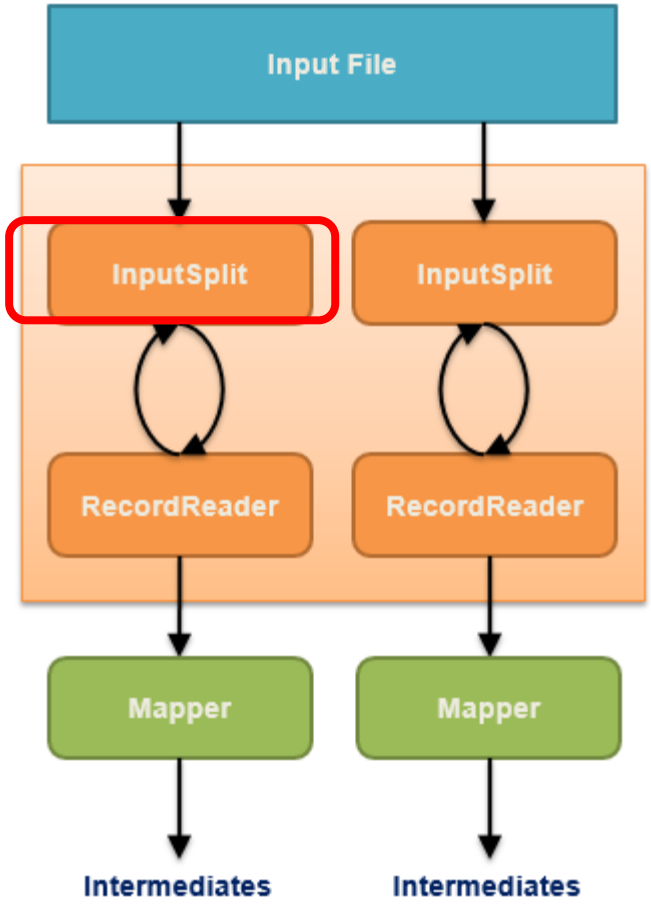
Record Readers

- Each *InputFormat* provides its own *RecordReader* implementation
- *LineRecordReader* – Reads a line from a text file
- *KeyValueRecordReader* – Used by *KeyValueTextInputFormat*



Input Split Size

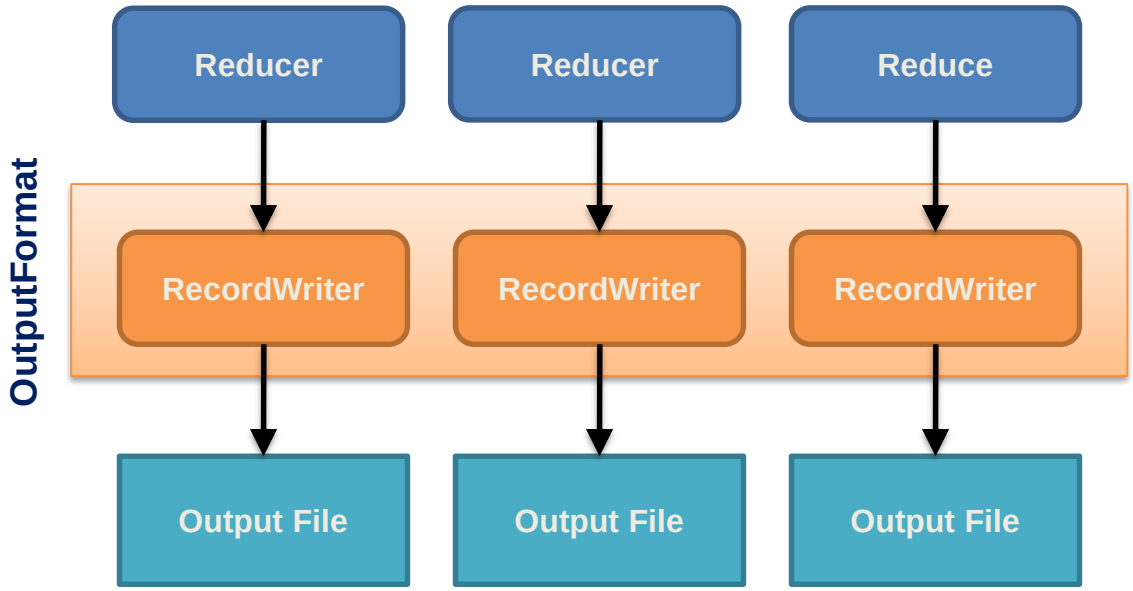
- *FileInputFormat* will divide large files into chunks
  - Exact size controlled by mapred.min.split.size
- RecordReaders receive file, offset, and length of chunk
- Custom *InputFormat* implementations may override split size – e.g., "NeverChunkFile"



Output from the Reducer

- A Map/Reduce may specify how it's output is to be written by specifying an *OutputFormat* to be used

- Defaults to *TextOutputFormat.* Other options include *SequenceFileOutputFormat*,…



Source: redrawn from a slide by Cloduera, cc-licensed

UNB FREDERICTON

Setting the Output Parameters

- The framework requires that the output parameters be configured, even if the job will not produce any output.

- The framework will collect the output from the specified tasks and place them into the configured output directory.

RECALL: MapReduce Terminologies

- Job: unit of work that client wants to be performed
  - Input Data
  - The MapReduce program
  - Configuration information

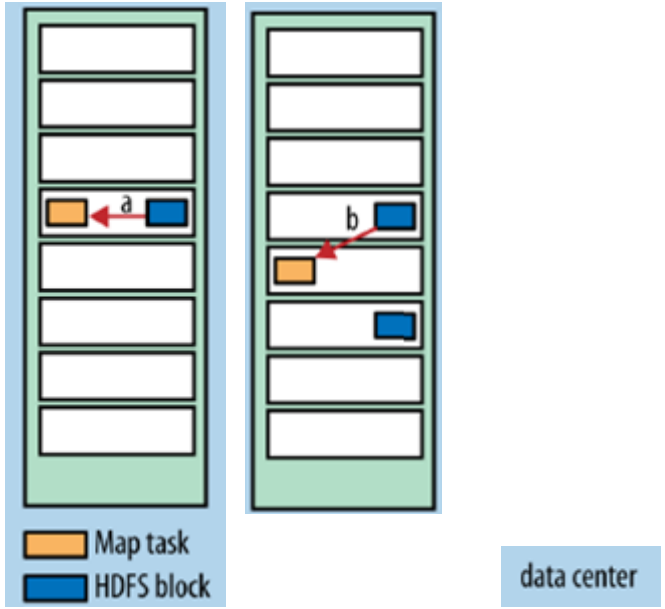
- Task
  - A job is divided into tasks: map task and reduce task

- Split
  - Hadoop divides the input into fixed size pieces
  - Hadoop creates one map task for each split, which runs the user-defined function for each record in the split
  - Hadoop attempts for *data locality optimization*

Map task placement

- Strategies for *data locality optimization*
  a) Attempts to run the map task on a node where the input data resides on the HDFS

  b) If no free map slot in the same node, look for a free slot in the same rack

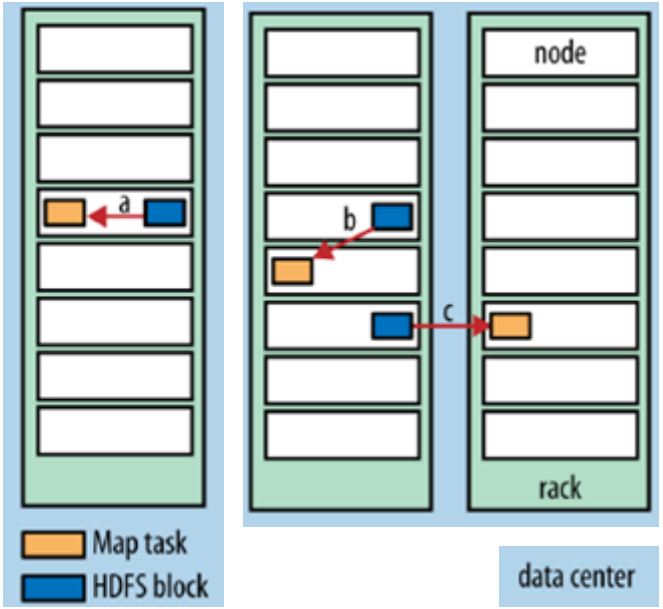


Map task placement

- Strategies for *data locality optimization*
  a) Attempts to run the map task on a node where the input data resides on the HDFS

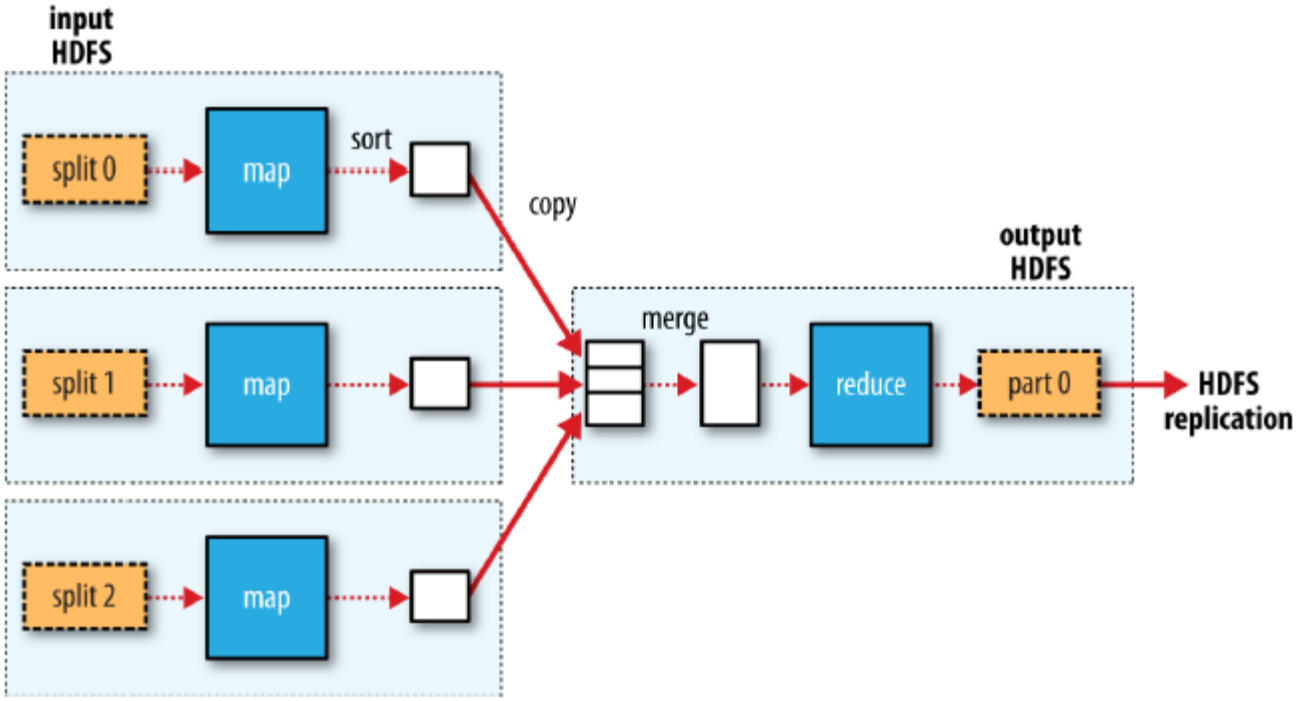
  b) If no free map slot in the same node, look for a free slot in the same rack

  c) If not found, look for a free slot in the same data center



Reduce task placement

- Reduce tasks don't have the advantage of data locality
  - Input to a reduce task is the output from all mappers
  - The sorted map outputs are transferred to the node where the reduce task is running



UNB FREDERICTON

Combiner

- Minimize the data transfer between map and reduce
  - Combiner function's output forms the input to the reduce function

- Example: find the max temp per year (without combiner)
  - First map output: (1950, 0), (1950, 20), (1950, 10)
  - Second map output: (1950, 25), (1950, 15)
  - Reduce input: (1950, [0, 10, 15, 20, 25])
  - Reduce output: (1950, 25)

UNB FREDERICTON

Combiner

- Minimize the data transfer between map and reduce
  - Combiner function's output forms the input to the reduce function

- Example: find the max temp per year (with combiner)
  - First map output: (1950, 0), (1950, 20), (1950, 10)
  - First combiner output: (1950, 20)

  - Second map output: (1950, 25), (1950, 15)
  - Second combiner output: (1950, 25)

  - Reduce input: (1950, [20, 25])
  - Reduce output: (1950, 25)

RECALL: Running on a cluster: the driver class

```
public class MaxTemperatureJobRunner {
   public int run(String[] args) throws Exception  {
      Job job = new Job();
      String userJarLocation = "/home/dbuser/eclipse/workspace/test-hadoop/maxt.jar";
      job.setJar(userJarLocation);      //alternative
      job.setJarByClass(MaxTemperatureJobRunner.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);
      boolean success = job.waitForCompletion(true);
      return success ? 0 : 1;
   }
  public static void main(String[] args) throws Exception {
      MaxTemperatureJobRunner driver = new MaxTemperatureJobRunner();
      driver.run(args);
   }
}
```

UNB FREDERICTON

Anatomy of a MapReduce job run

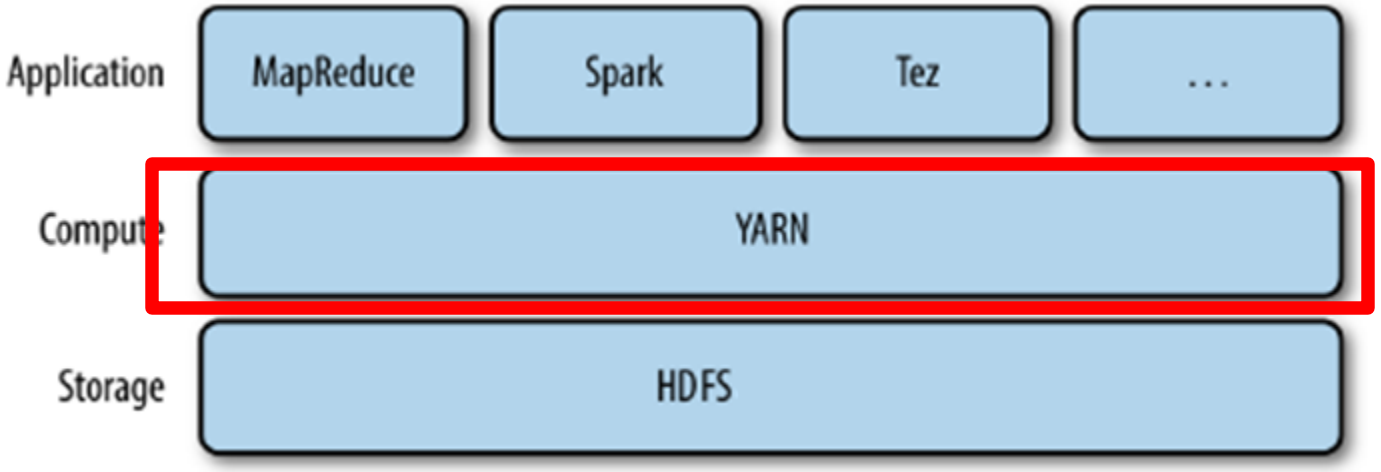
- 2 ways to run a job
  - job.submit()
  - Job.waitForCompletion(): submits the job, then waits for it to finish

- Entities involved
  - The client: submits the MapReduce job
  - YARN resource manager: coordinates the allocation of compute resources
  - YARN node managers: launch and monitor the compute containers
  - MapReduce application master: coordinates the tasks running the Map-Reduce job
  - HDFS

Outline

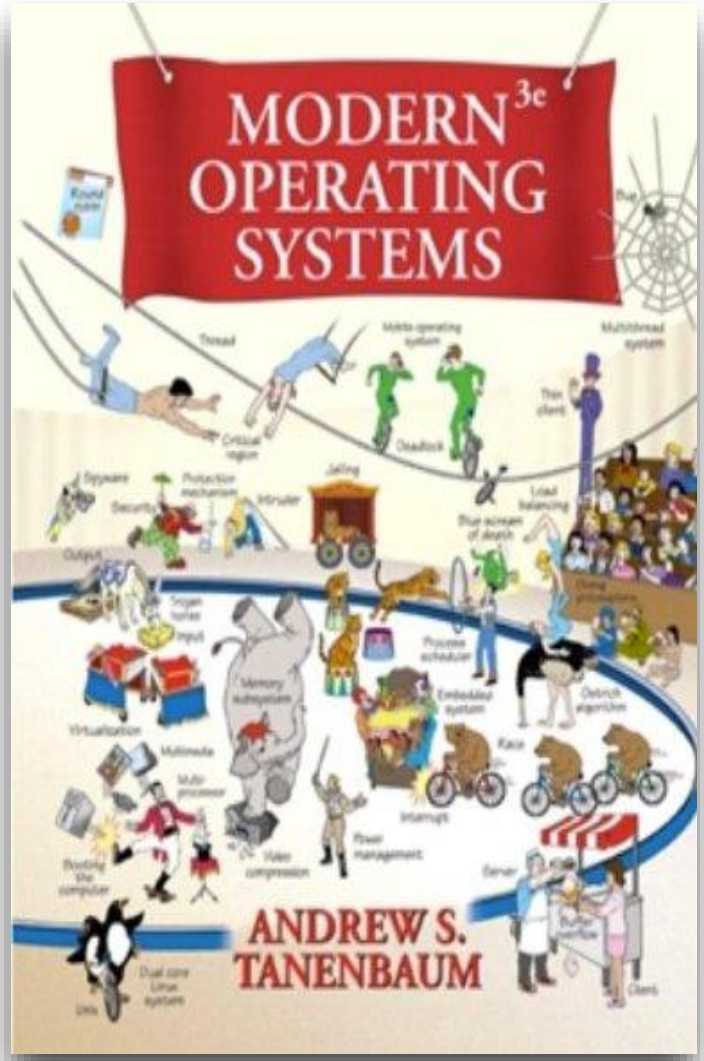
- Introduction
- Developing a MapReduce Application
- How MapReduce Works
- YARN 
- Hadoop Distributed File Systems (HDFS)
- MapReduce Algorithm Design

YARN


Application
MapReduce
Spark
Tez
...
Compute
YARN
Storage
HDFS

The role of the Operating Systems

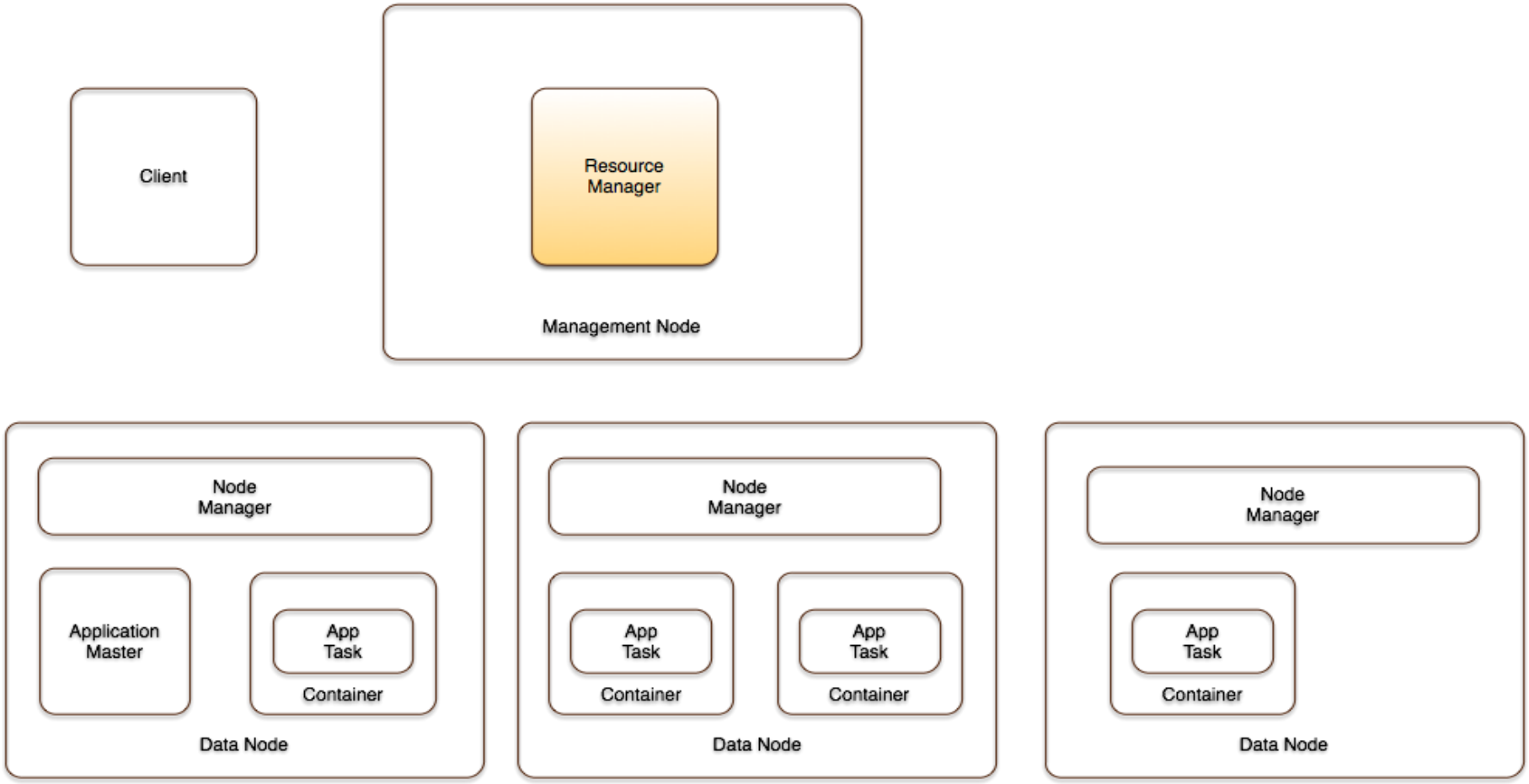
"…The Operating System as a **Resource Manager**… the job of the operating system is to provide for an orderly and controlled allocation of the processors, memories, and/or devices among the various programs competing for them"




With YARN
Hadoop becomes
a distributed "OS"

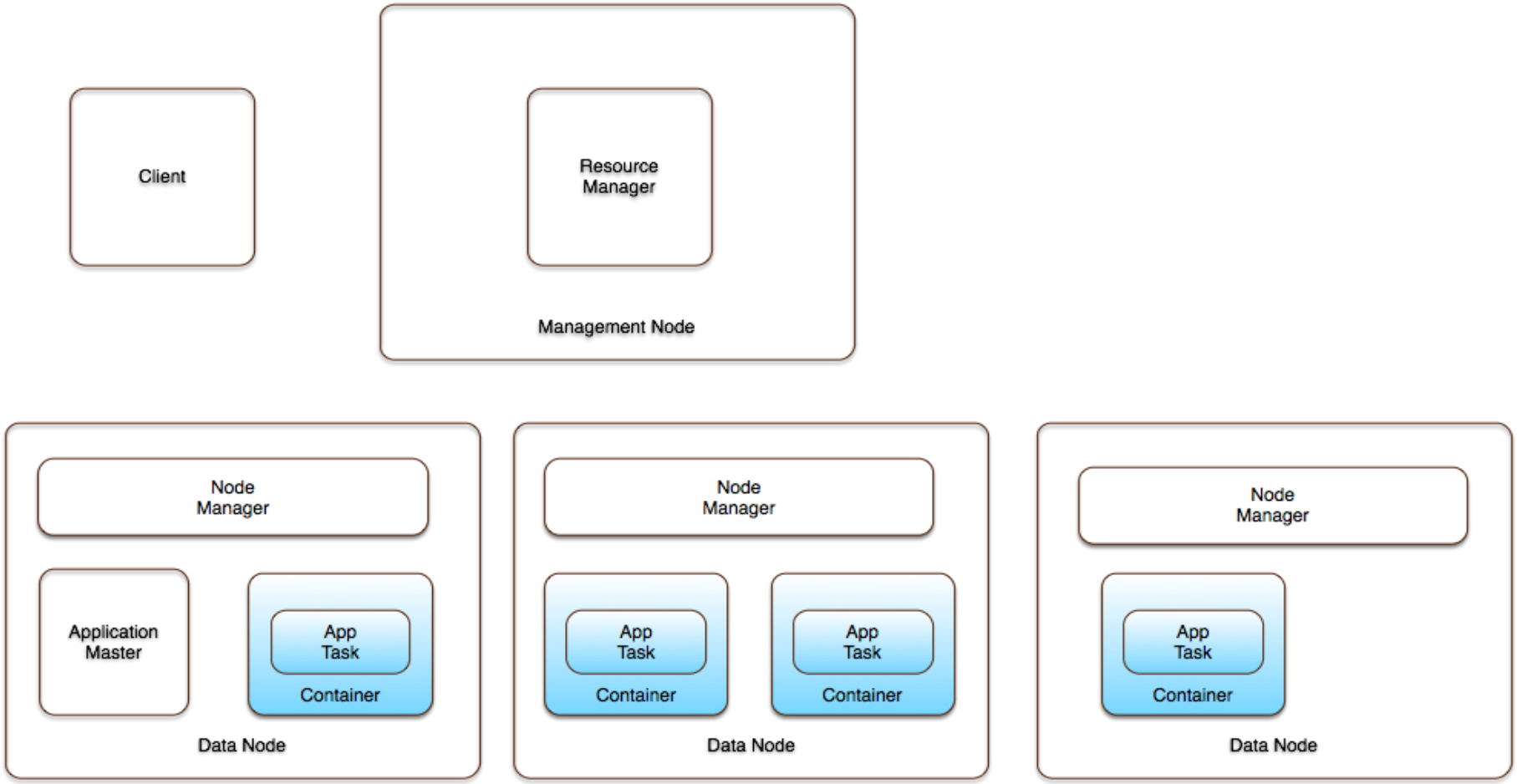
Resource Manager

- The Resource Manager is essentially a scheduler



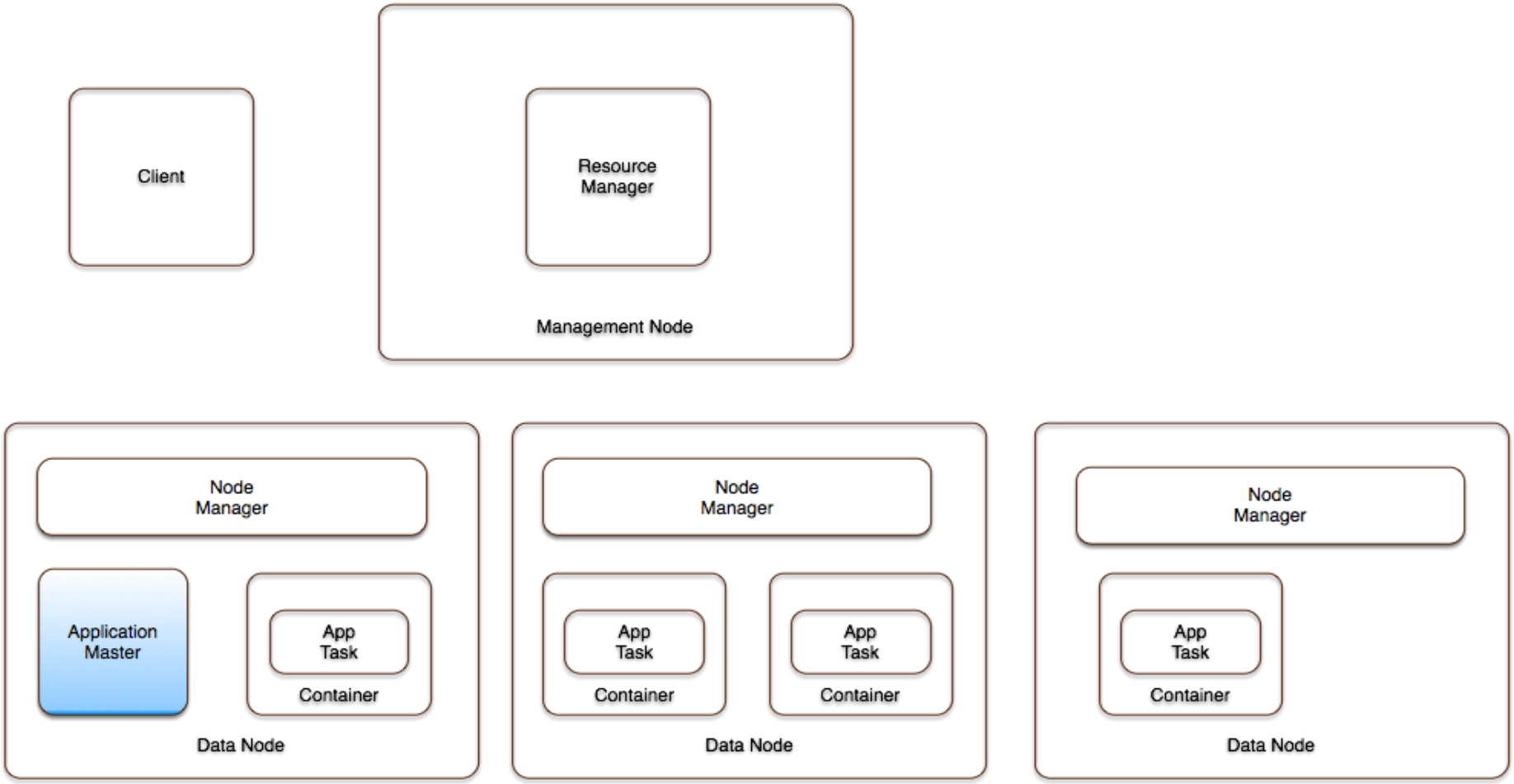
Resource Manager

- **Containers** are allocations of physical resources



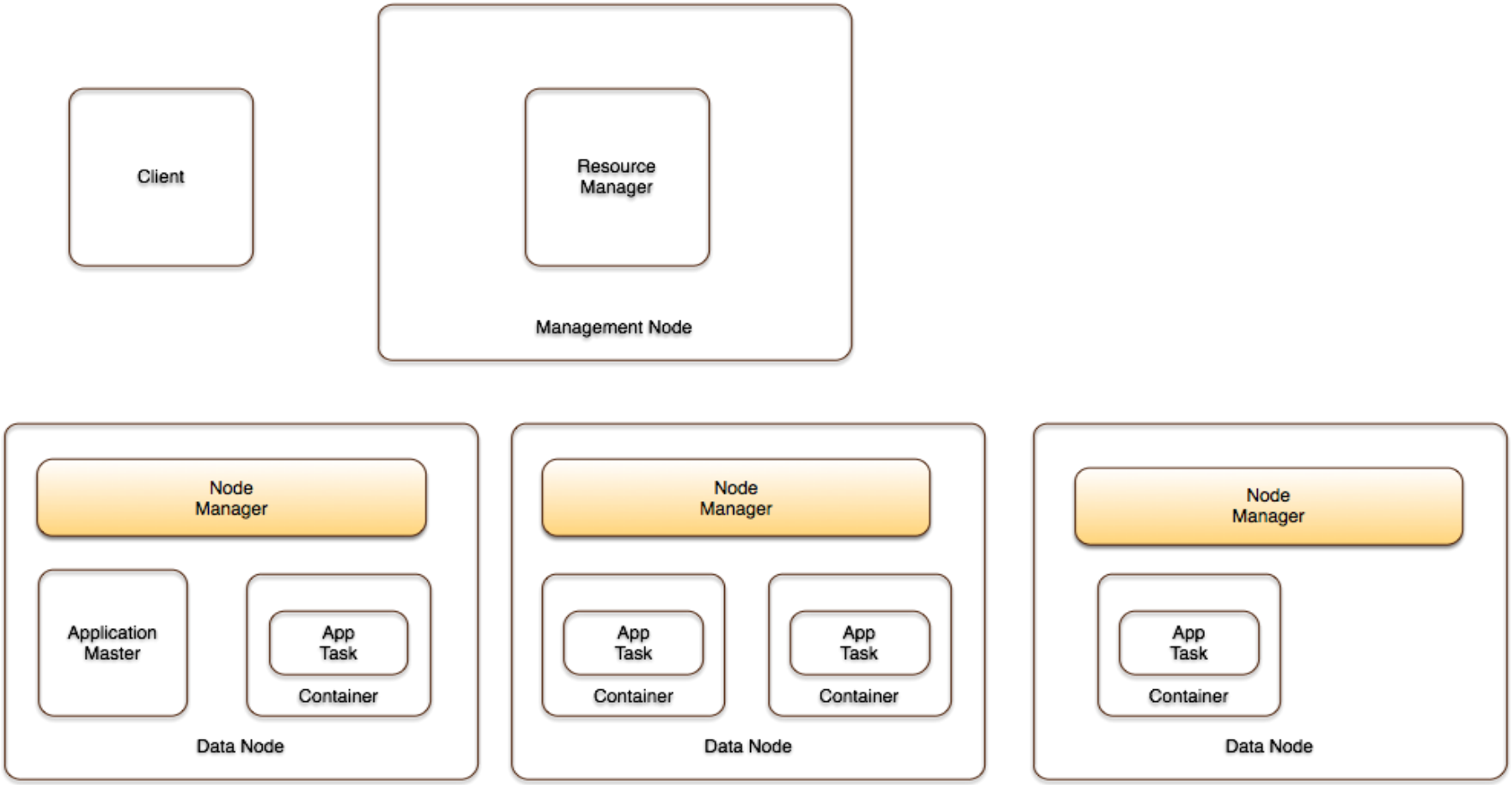
Application Manager

- Each app instance spawns an **application master** (container) - to negotiate resource and monitor app progress (tasks)



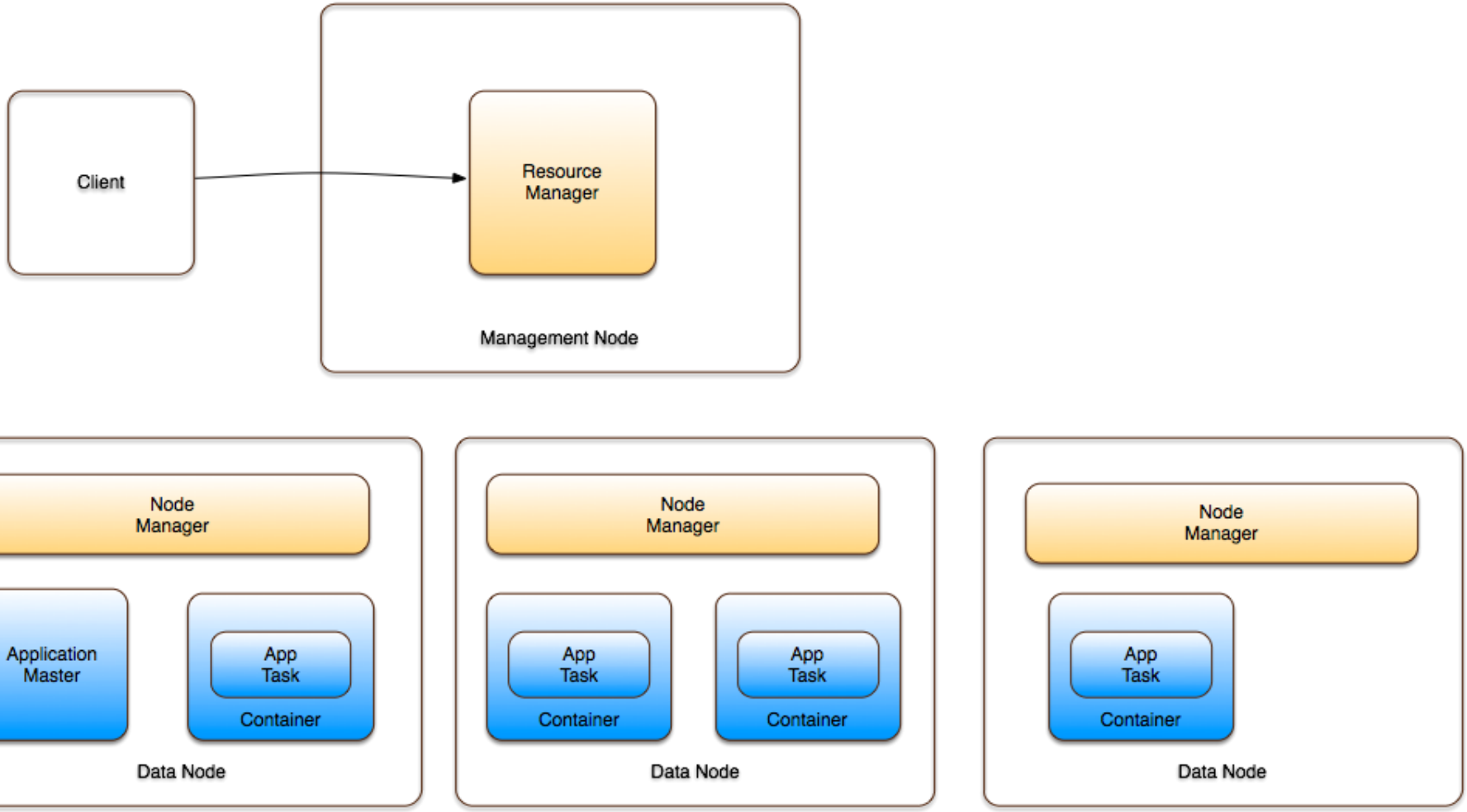
Node Manager

- **Node manager**s monitor nodes and manage containers lifecycle



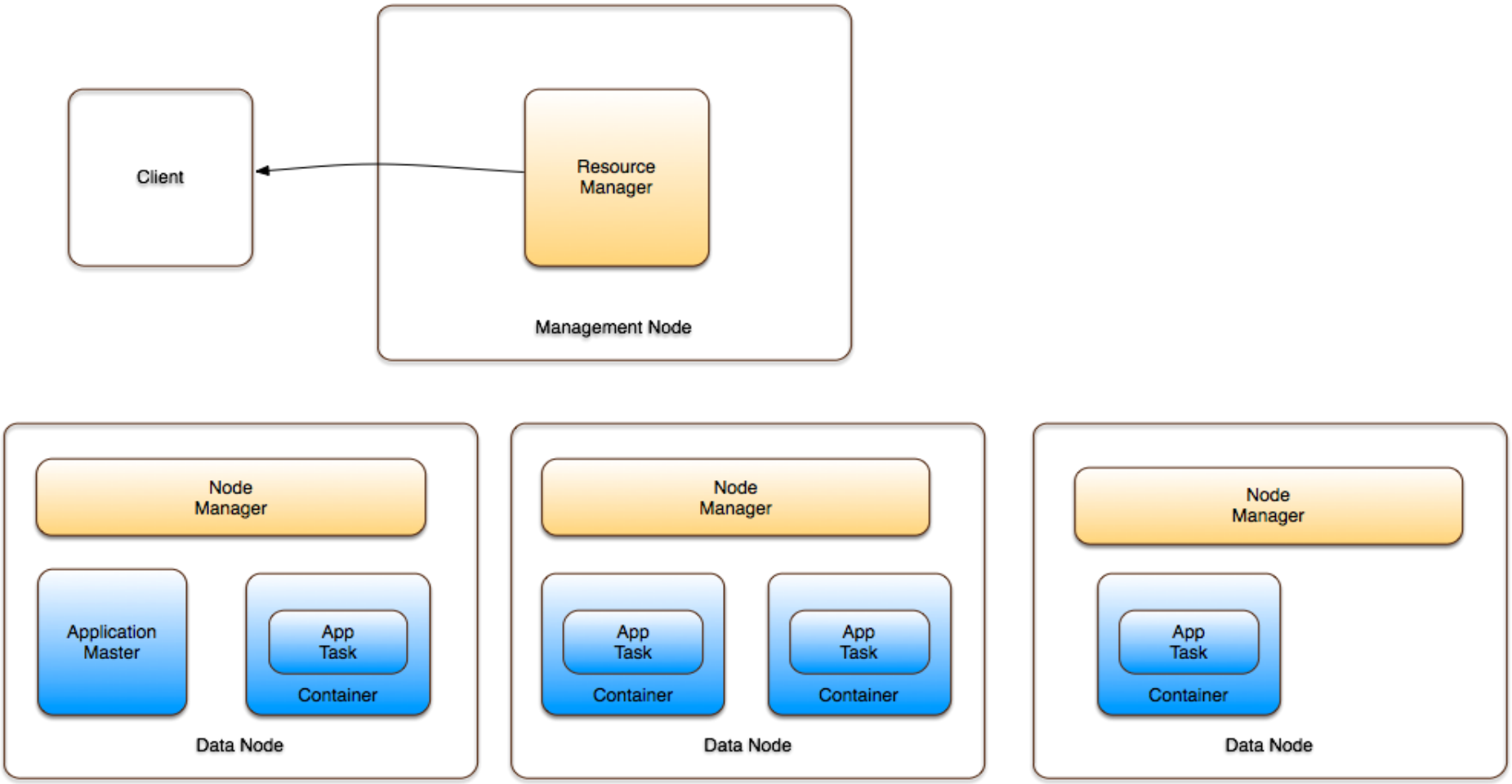
Application Initiation

1. Client submits a job/app



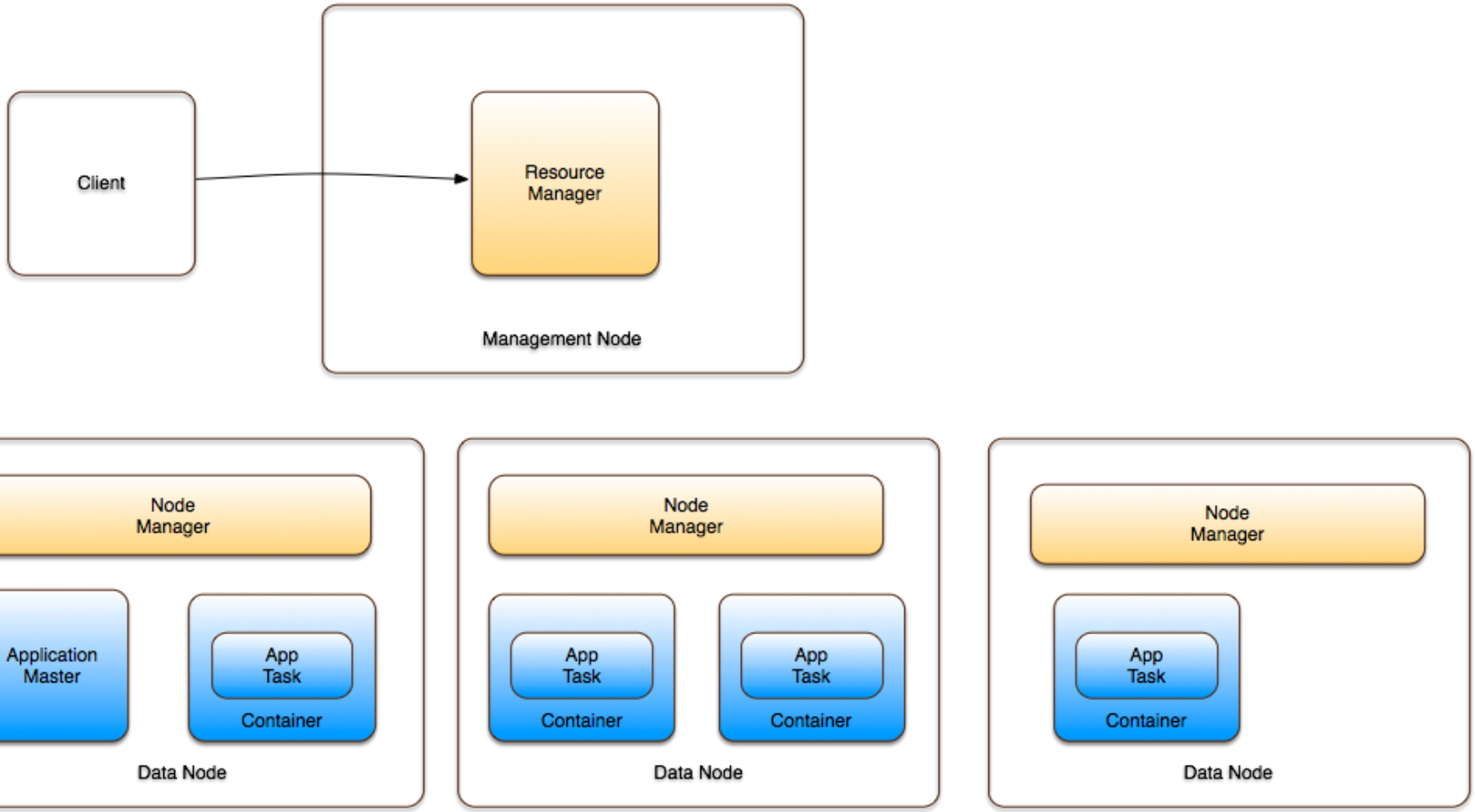
Application Initiation

2. Resource Manager (RM) provides Application Id



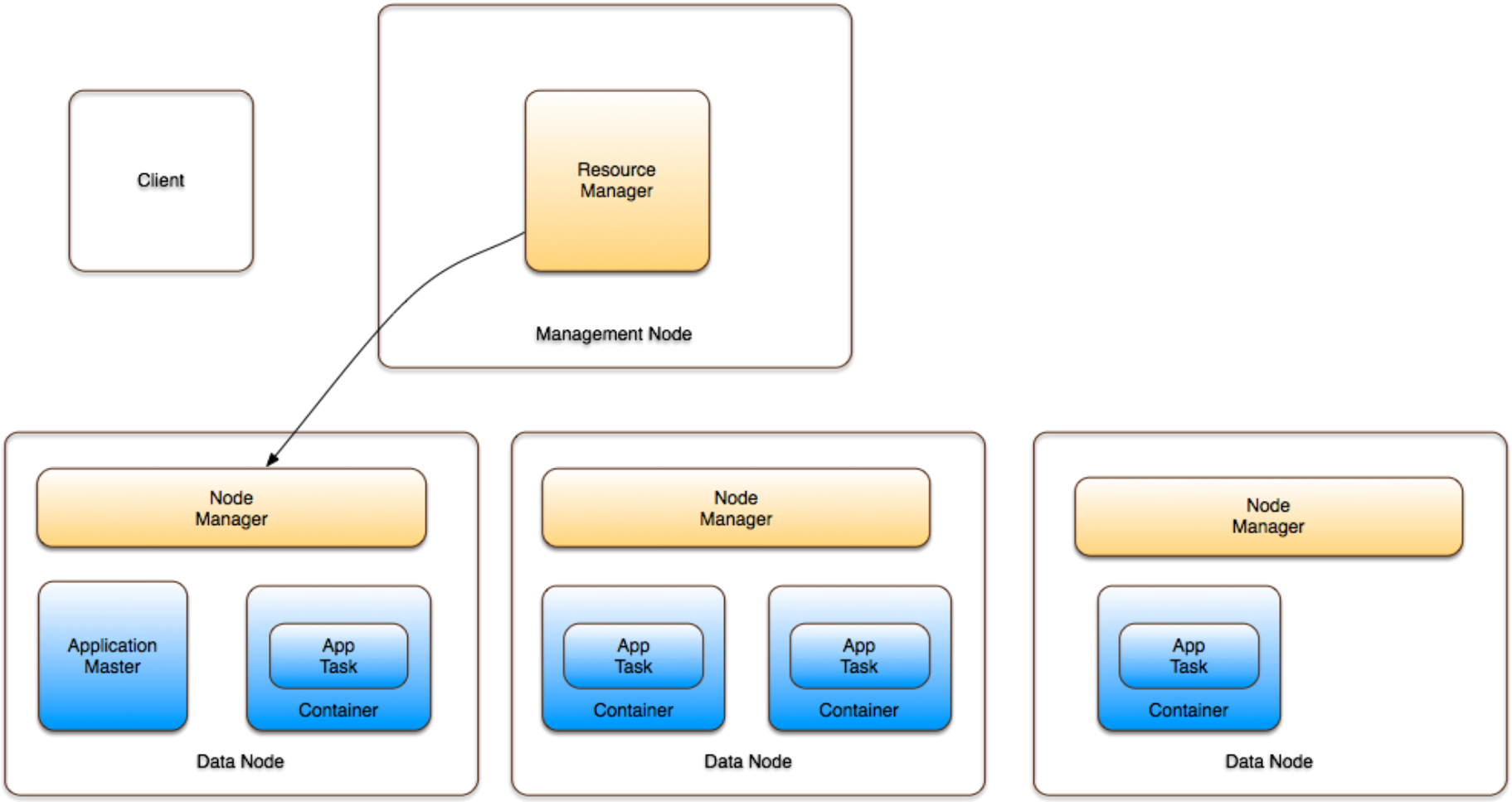
Application Initiation

3. Client provides context (queue, resource requirements, files, security tokens etc.)



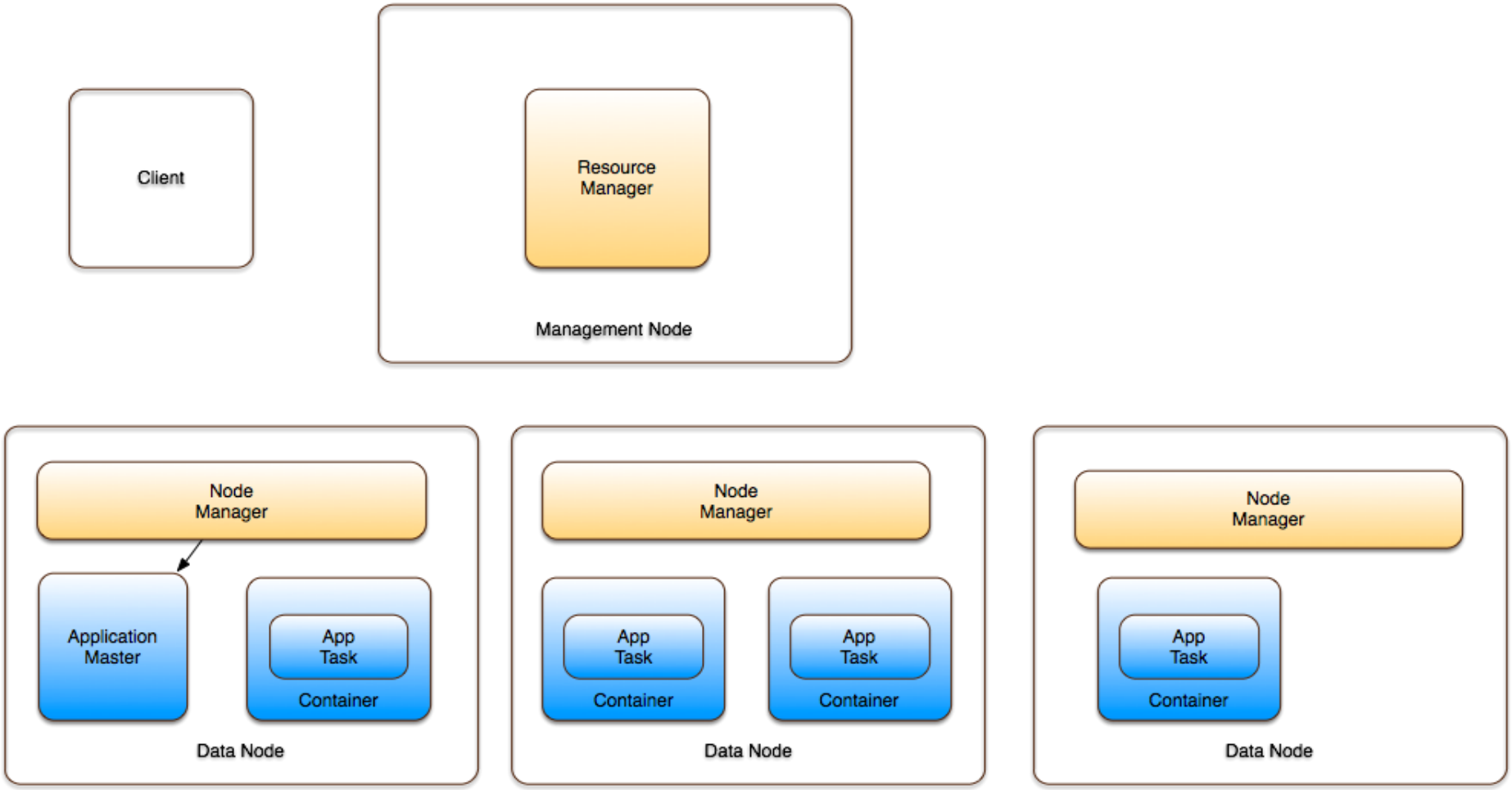
Application Initiation

4. RM asks Node Manager to launch Application Master



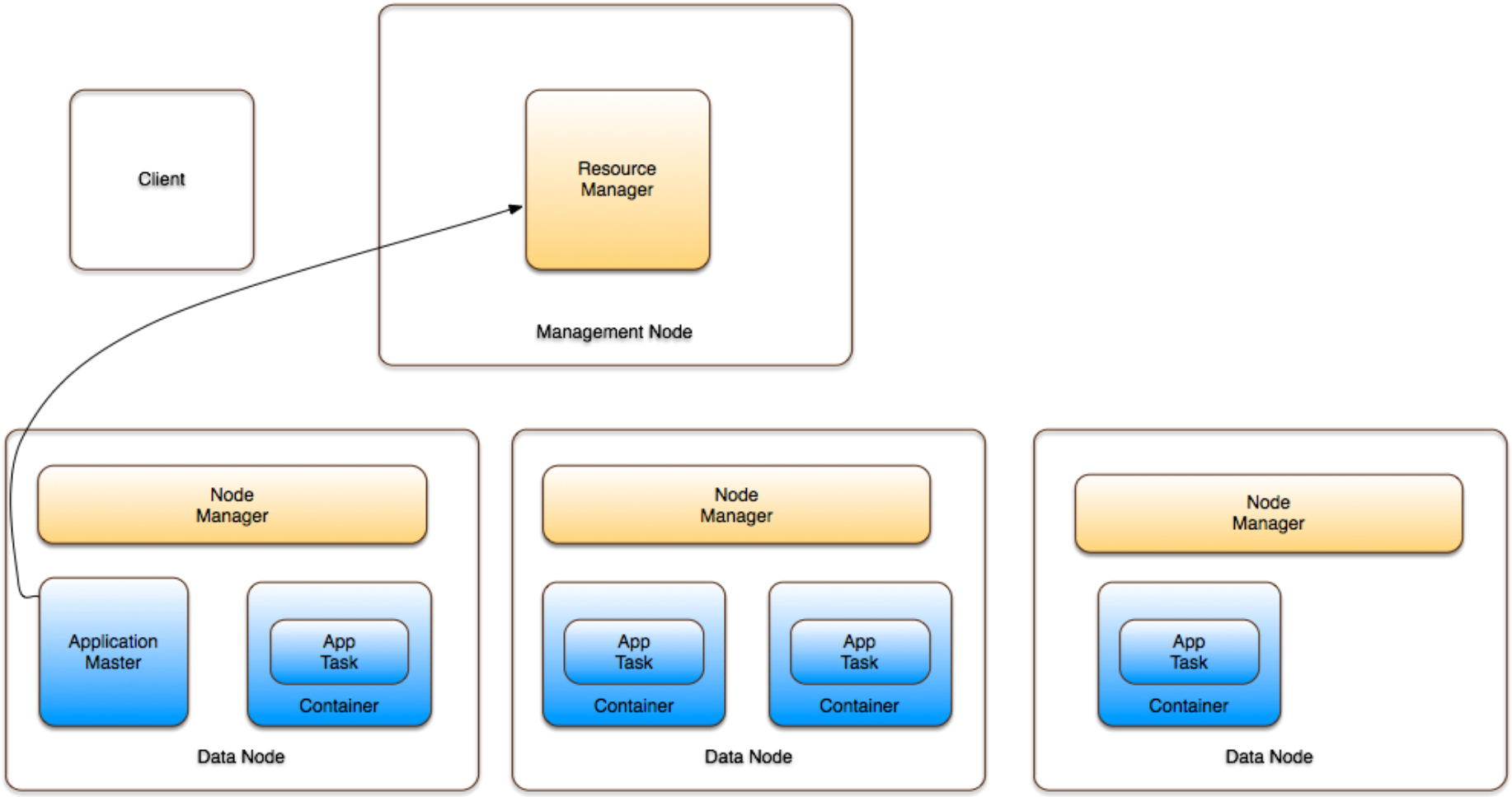
Application Initiation

5. Node Manager launches Application Master



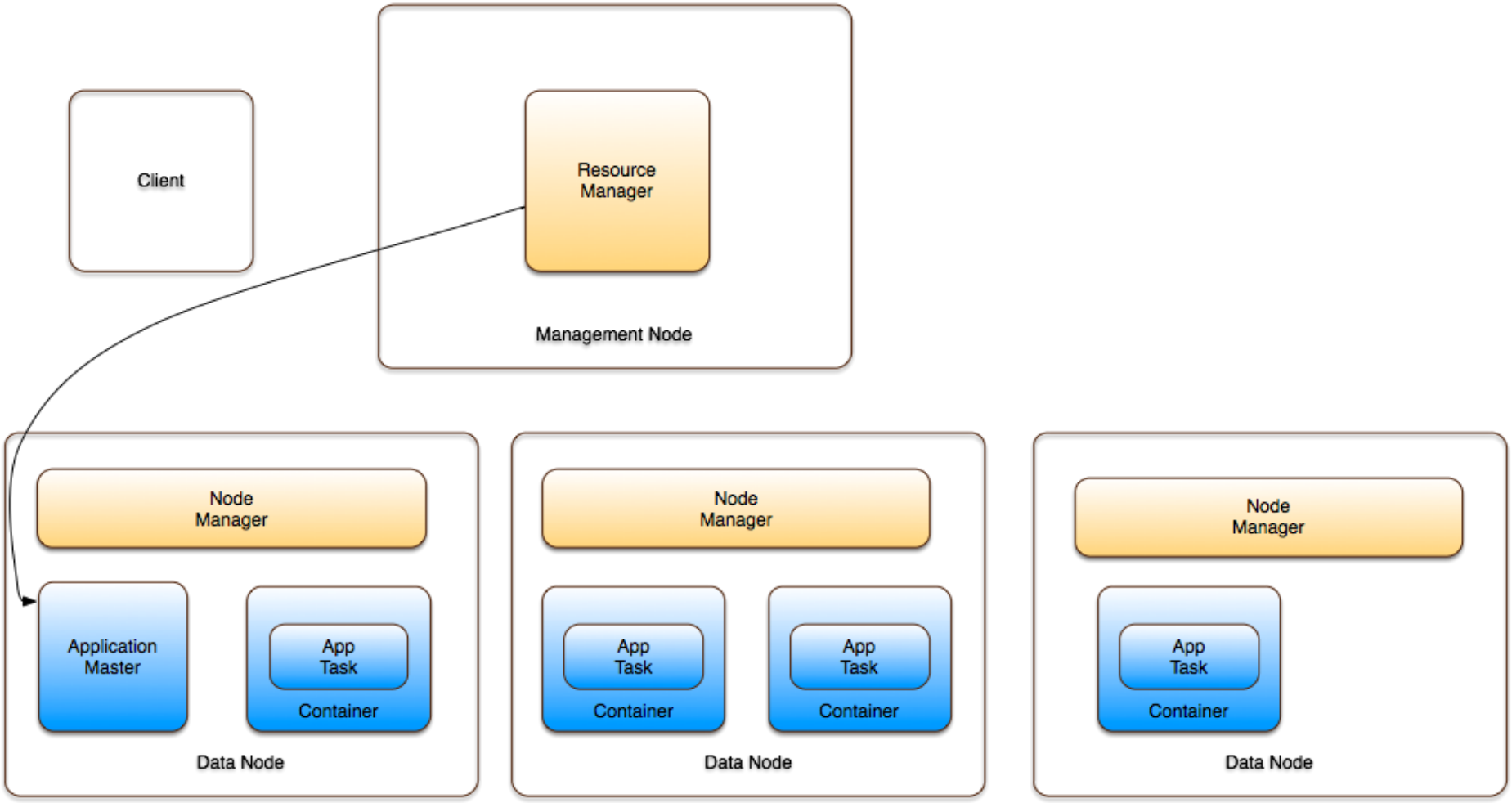
Application Initiation

6. Application Master registers with RM



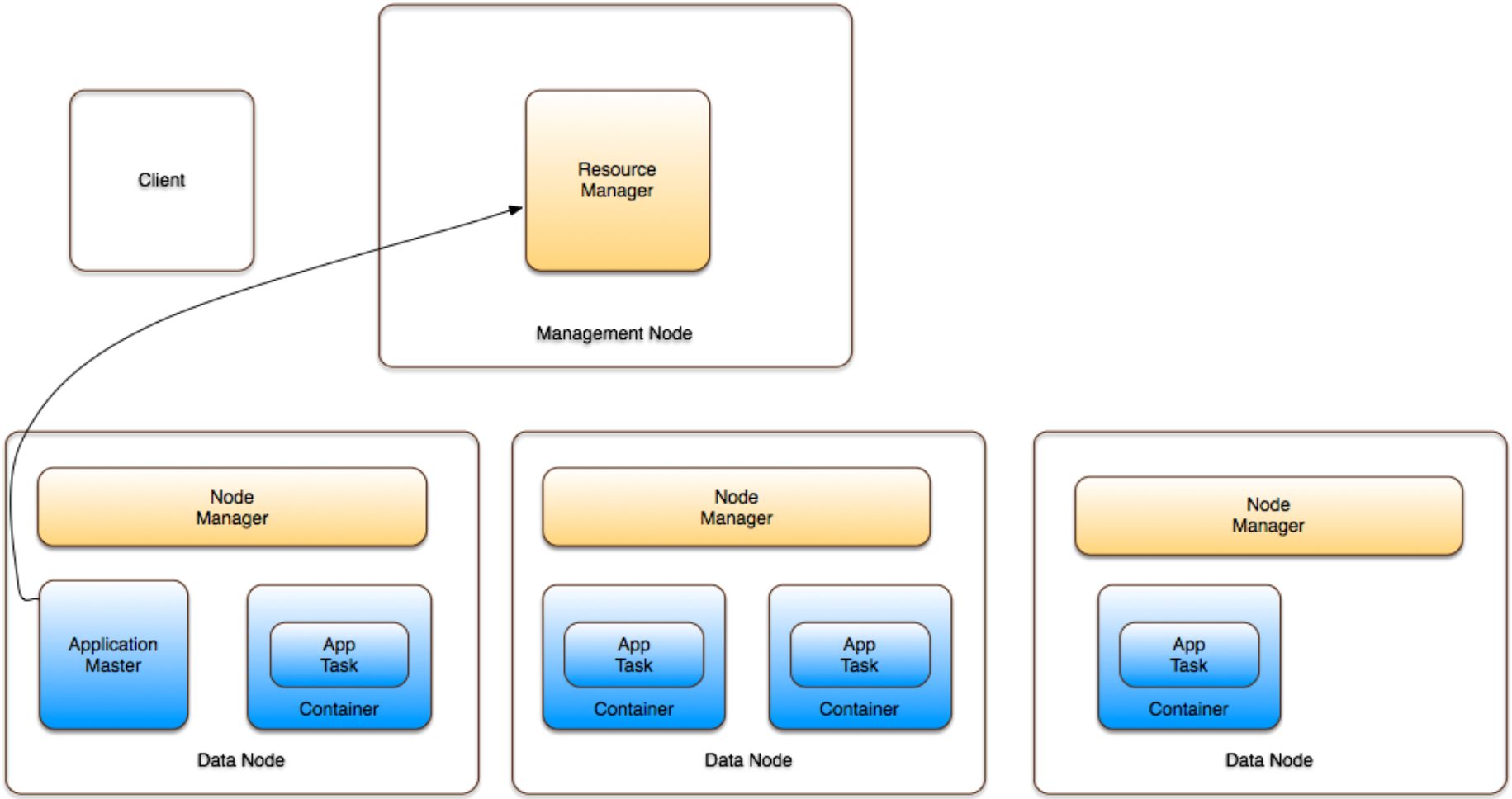
Application Initiation

7. RM shares resource capabilities with Application Master



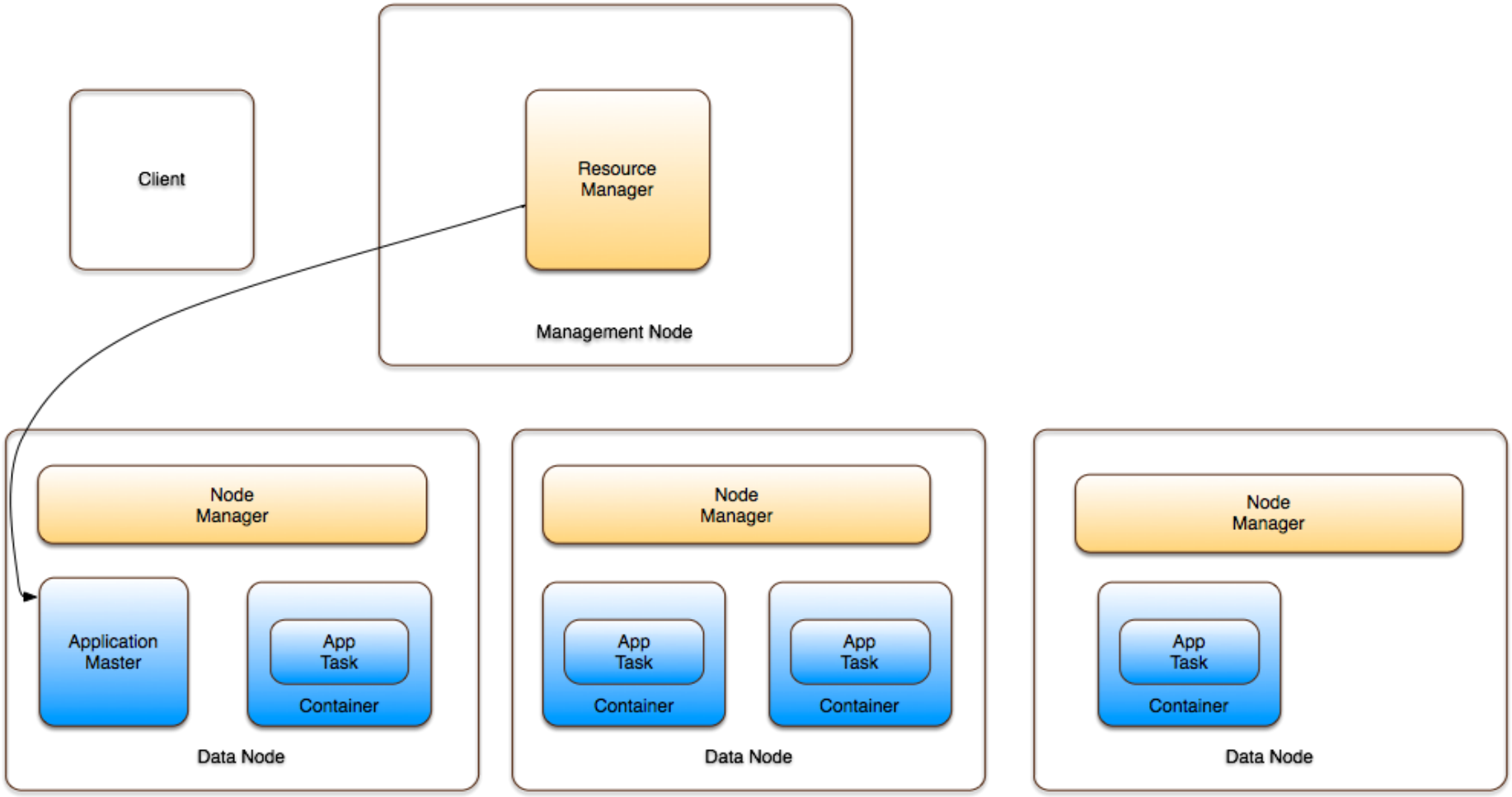
Application Initiation

8. Application Master requests containers



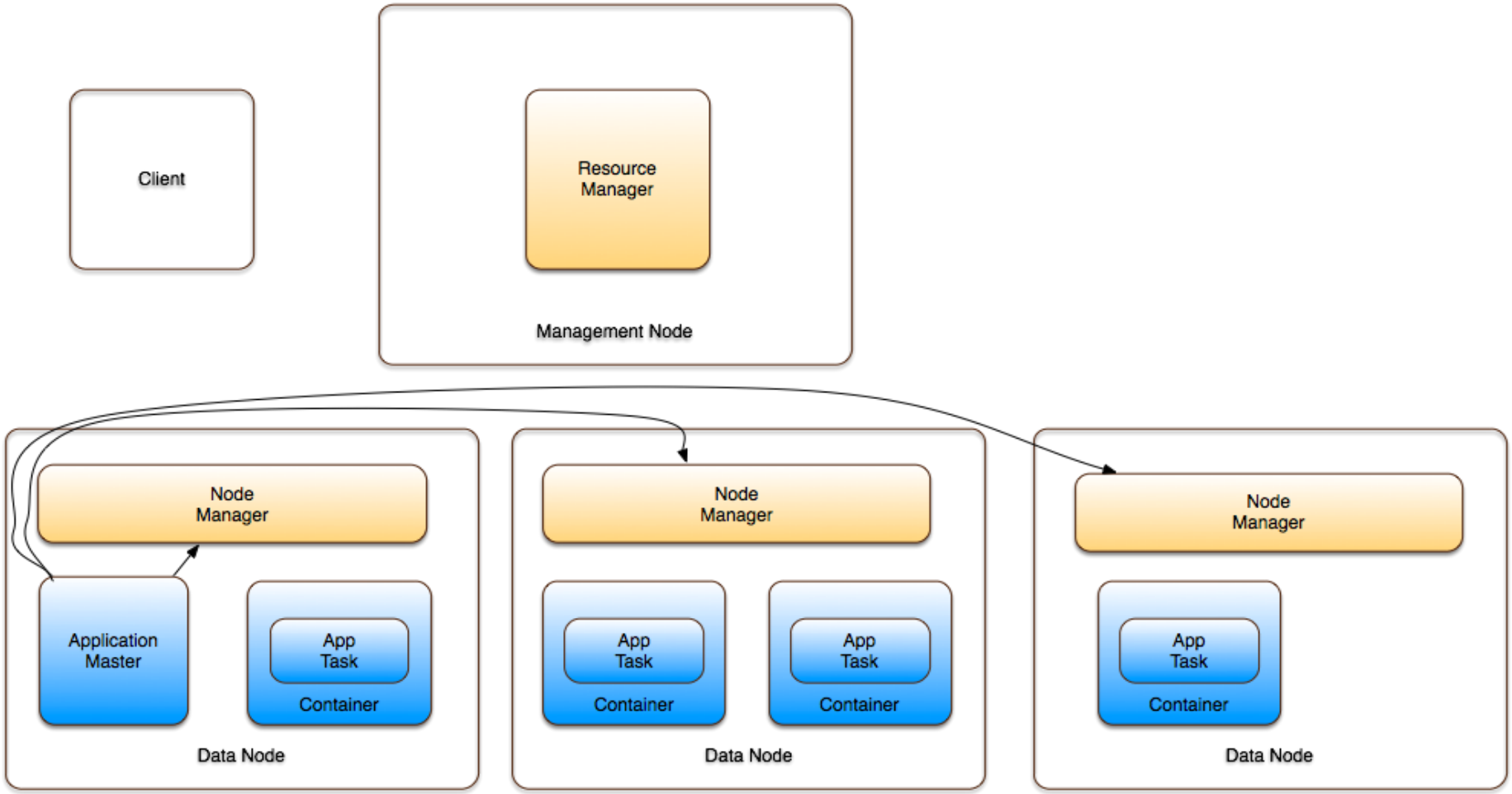
Application Initiation

9. RM assigns containers based on policies and available resources



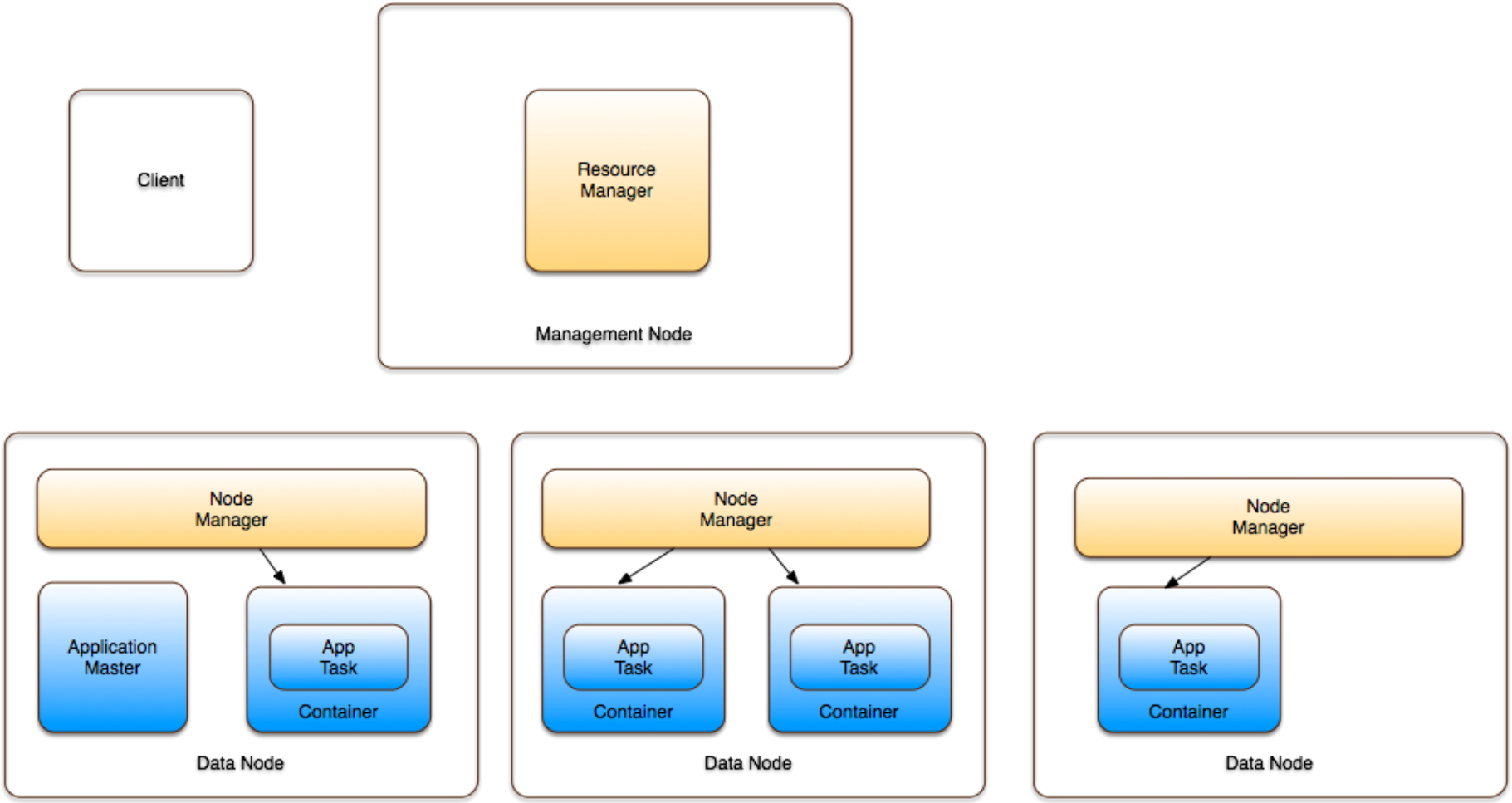
Application Initiation

10. Application Master contacts assigned node managers to instantiate containers (passing container contexts)



Application Initiation

11. Node Manager initiate container(s)




Congratulations your
Application is now running

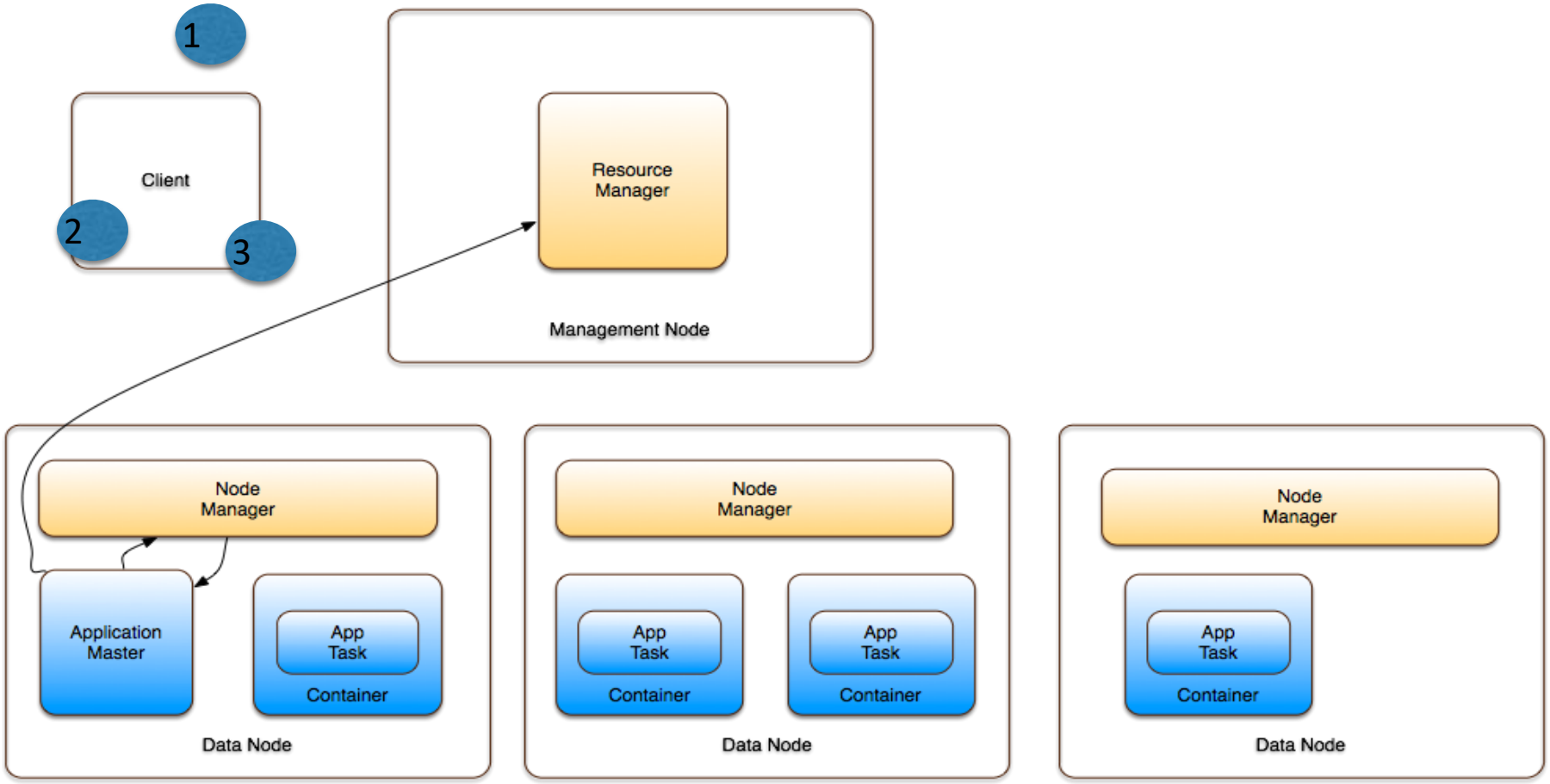
UNB FREDERICTON

Speculative Execution

- Problem
  - MapReduce model break jobs into tasks and run the tasks in parallel to make the overall job execution time smaller
  - Takes only one slow task ("straggler") to make the whole job take significantly longer
- Solution: *speculative execution*
  - Tries to detect when a task is running slower than expected and launches another equivalent task as a backup
  - Only launches speculative duplicates for tasks that are running significantly slower than the average (using **task progress** info)

Application Progress Monitoring

1. Continuous heartbeat & progress report
2. Request container status
3. Status response

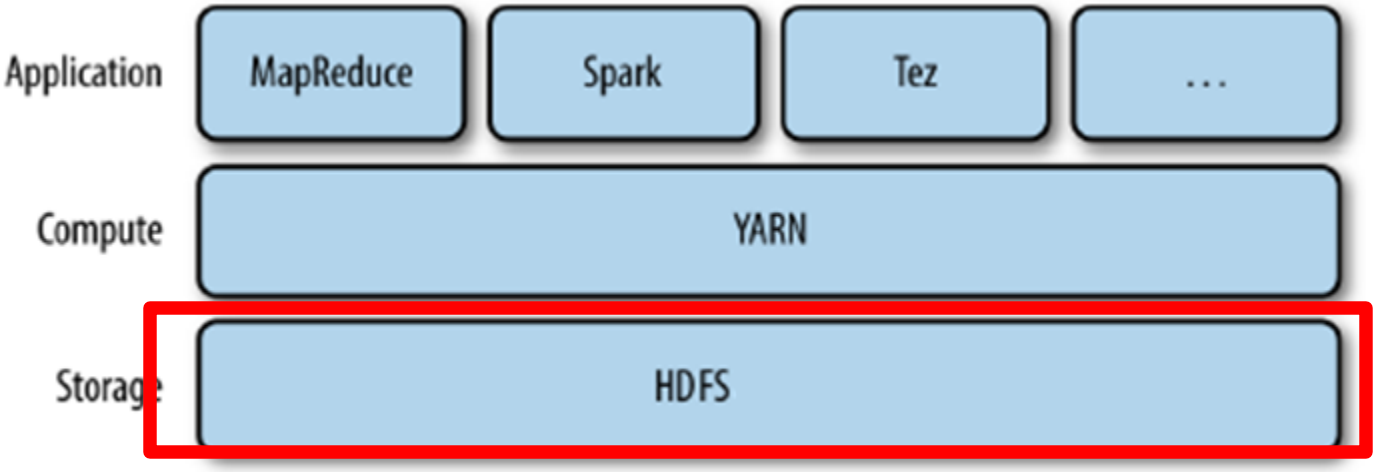


UNB FREDERICTON

Outline

- Introduction
- Developing a MapReduce Application
- How MapReduce Works
- YARN
- Hadoop Distributed File Systems (HDFS) 
- MapReduce Algorithm Design

HDFS


Application
MapReduce
Spark
Tez
...
Compute
YARN
Storage
HDFS

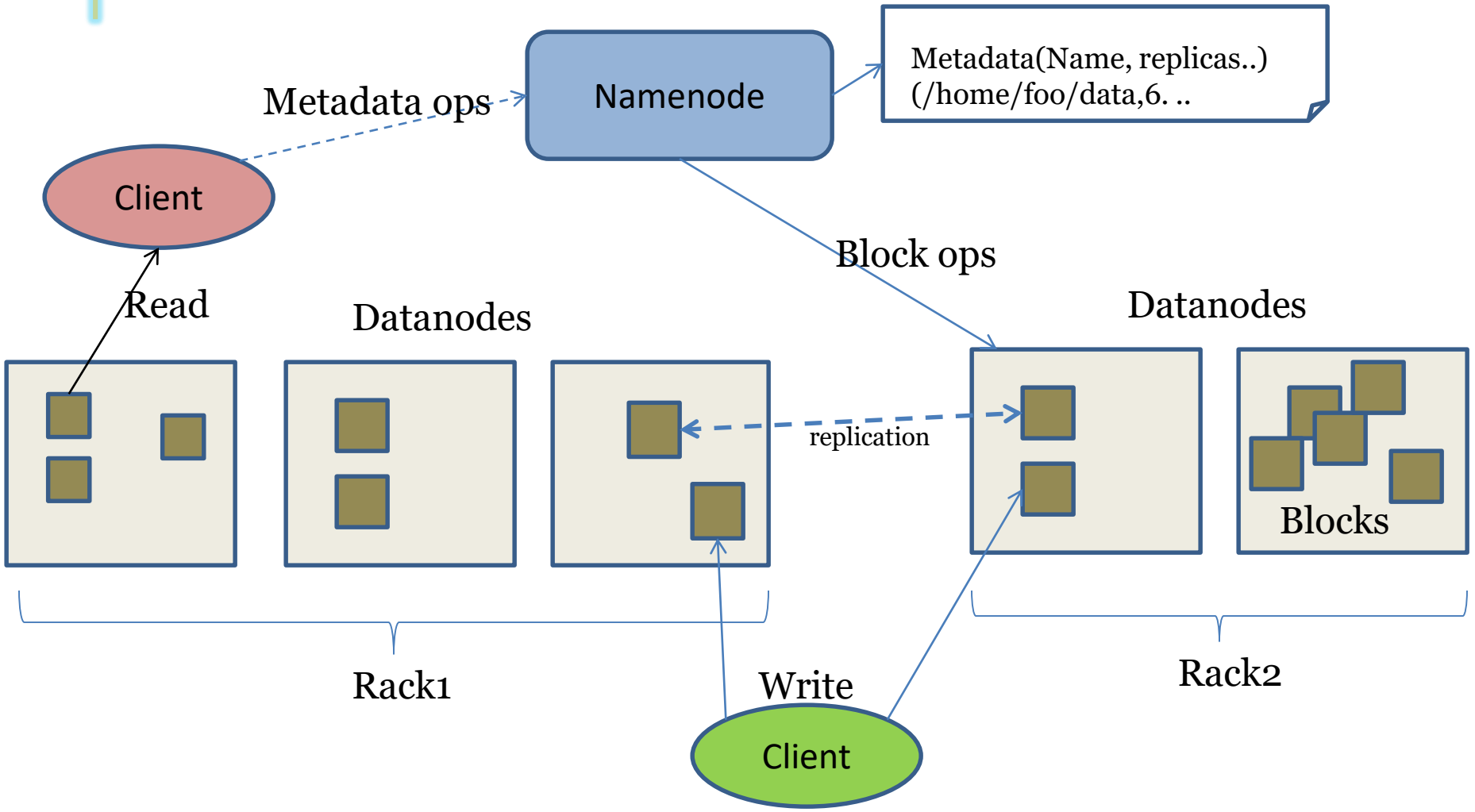
Hadoop Distributed File System (HDFS)

- HDFS is an open-source version of the GFS
- HDFS is a distributed file system for large scale data
- Advantages
  - Designed to store a very large amount of information. This requires spreading the data across a large number of machines.
  - HDFS should store data reliably. If individual machines in the cluster malfunction, data should still be available.
  - Provides fast, scalable access to this information.
  - Integrate well with Hadoop MapReduce, allowing data to be read and computed upon locally when possible.

Basic Features: HDFS

- Can be built out of a cluster of commodity hardware -> high failure probability
- Highly fault-tolerant
- High throughput
- Suitable for applications with large data sets
- Streaming access to file system data

HDFS Architecture



Src: C B. Ramamurthy

HDFS: Namenode and Datanodes

- Master/slave architecture. HDFS exposes a file system namespace and allows user data to be stored in files.

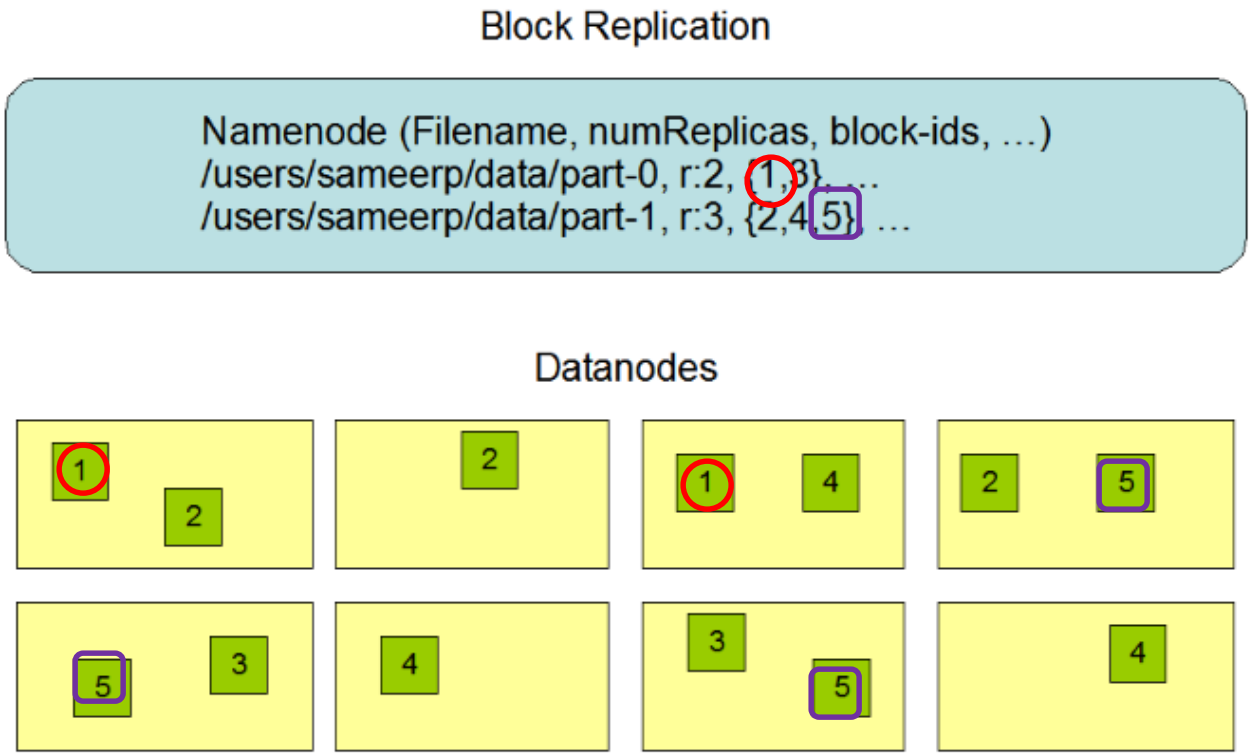
- HDFS cluster consists of a single **Namenode**, a master server that manages the file system namespace and regulates access to files by clients.

- There are a number of **DataNodes** usually one per node in a cluster.
  - Manage storage attached to the nodes that they run on.
  - Serves read, write requests, performs block creation, deletion, and replication upon instruction from Namenode.

- A file is split into one or more blocks and set of blocks are stored in DataNodes.

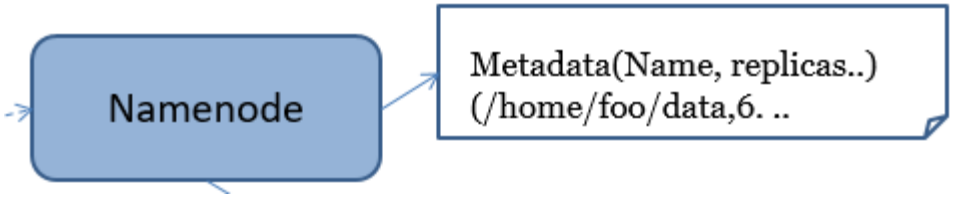
How HDFS stores files

- Each file is stored as a sequence of blocks
  - All blocks in a file except the last block are the same size
- The blocks of a file are replicated for fault tolerance
  - The block size and replication factor are configurable per file



Namenode

- Keeps image of entire file system namespace and file *Blockmap* in memory.

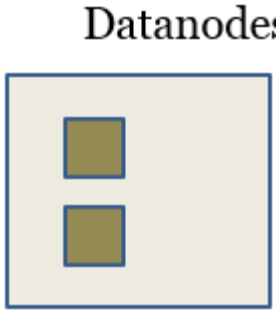


- 4GB of local RAM is sufficient to support the above data structures that represent the huge number of files and directories.

- Periodic checkpointing is done. So that the system can recover back to the last check-pointed state in case of a crash.

Datanode

- A Datanode stores data in files in its local file system.



- Datanode has “no knowledge” about HDFS filesystem
- It stores each block of HDFS data in a separate file.
- Datanode does not create all files in the same directory.
- When the filesystem starts up it generates a list of all HDFS blocks and send this report to Namenode: *Blockreport*.

HDFS command line interface

- 'POSIX-like' commands

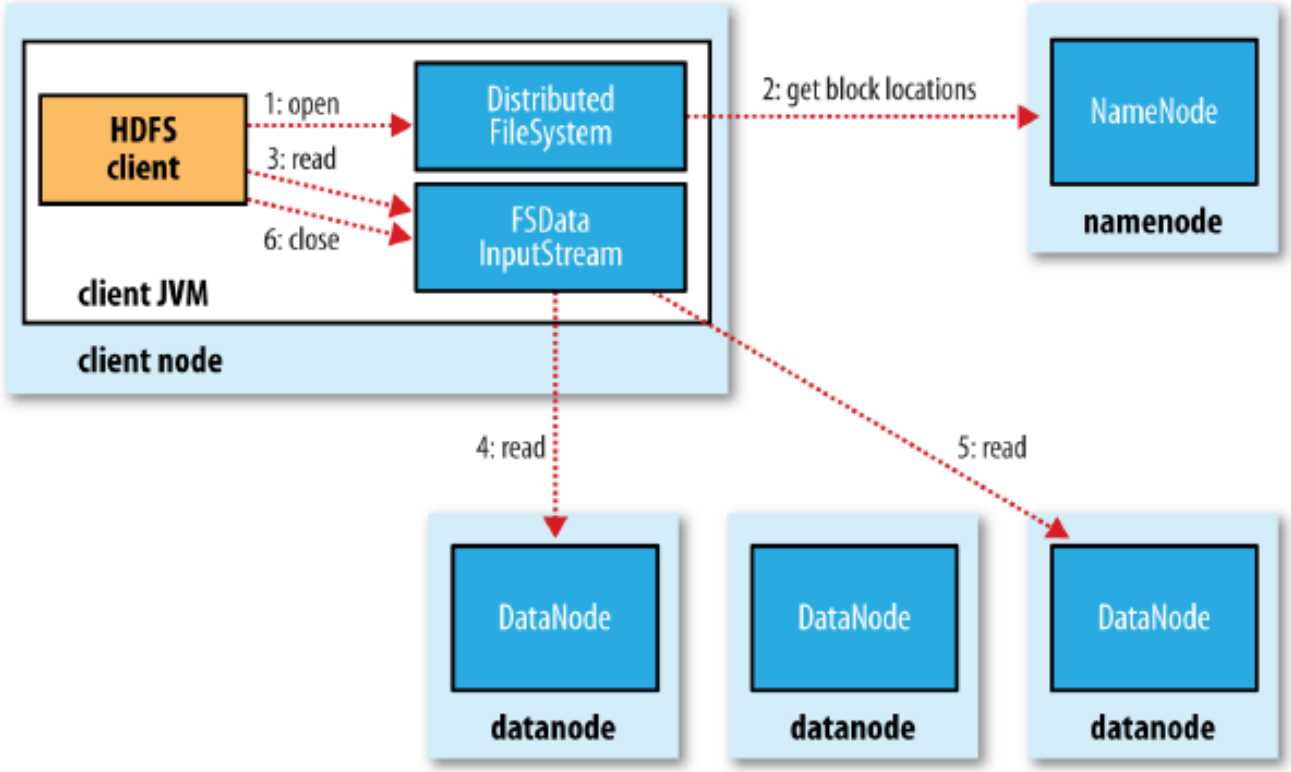
```
$ hadoop dfs  -ls /
 Found 3 items
 -rw-r--r--   2 dbuser supergroup      21055 2016-01-27 11:07 /derby.log
  drwx------   - dbuser supergroup          0 2016-02-05 15:46 /tmp
 drwxr-xr-x   - dbuser supergroup          0 2015-12-04 17:26 /user
```

```
$ hadoop dfs -cat /user/dbuser/output/part-r-00000
1901   317
```

```
$ hadoop dfs -rmr /user/dbuser/output
Deleted /user/dbuser/output
```

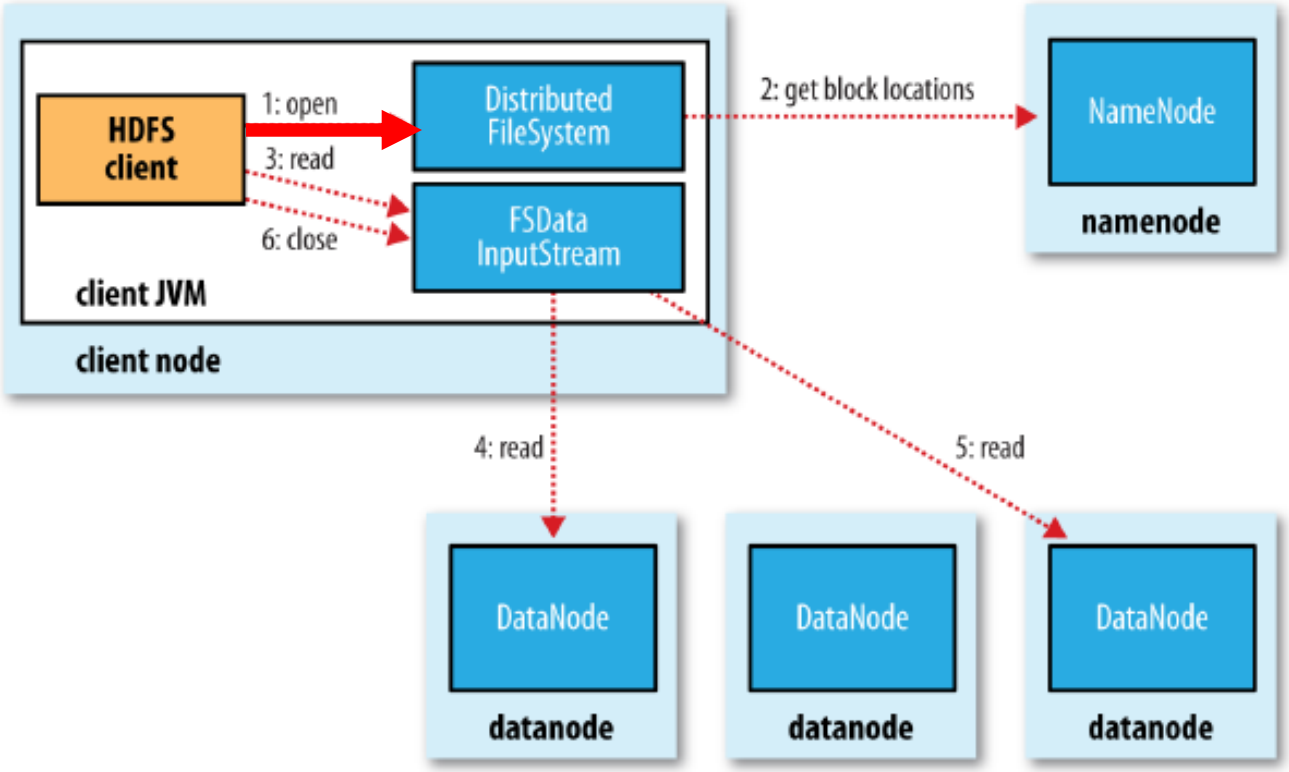
Anatomy of a file read

Read this → /usr/sameerp/data/part-0



Anatomy of a file read

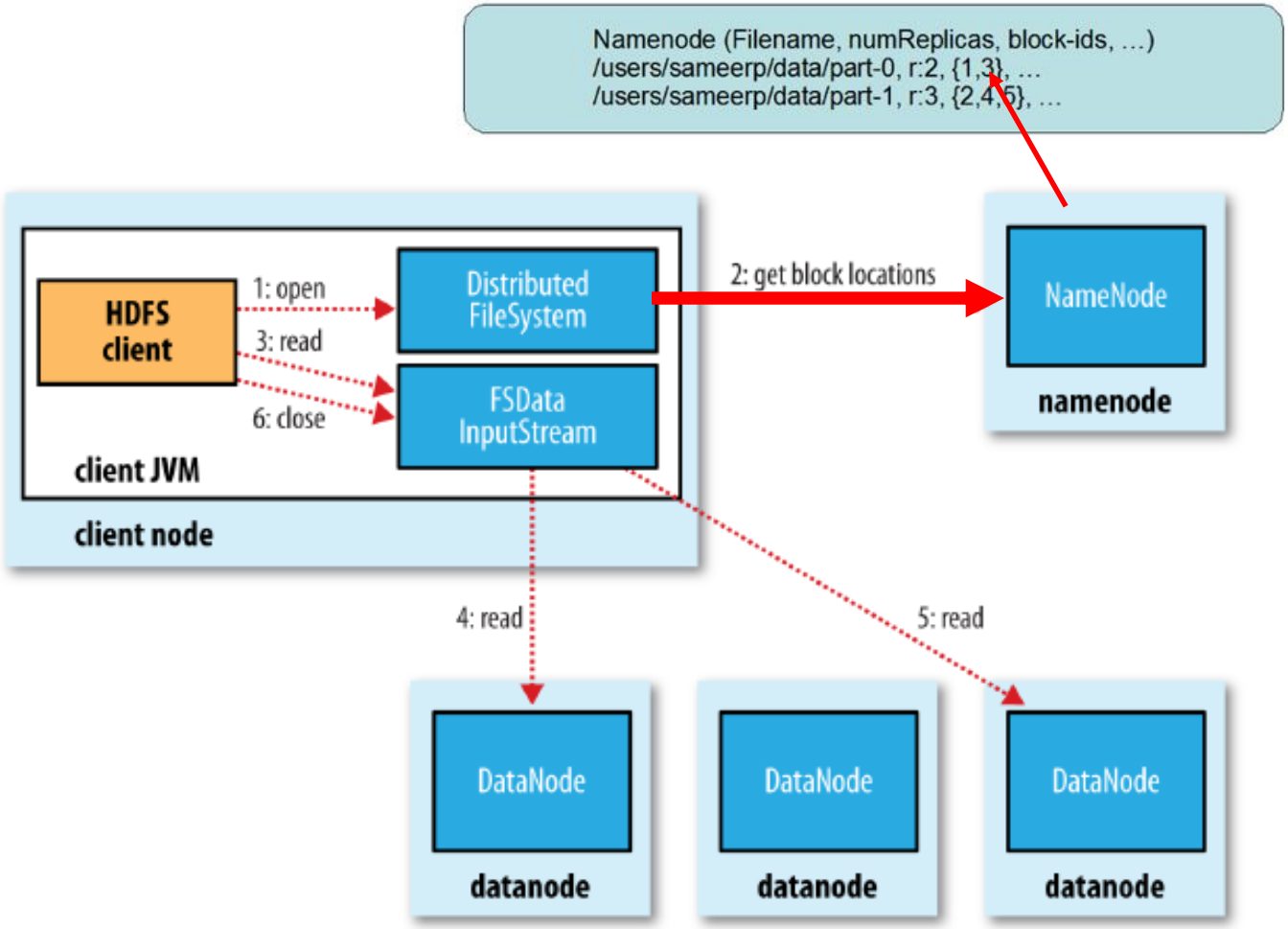
1. The client opens the file by calling ***open()*** on the ***FileSystem*** object, which is an instance of ***DistributedFileSystem***



Anatomy of a file read

2. The ***DistributedFileSystem*** object calls the namenode, using RPC to determine the first few blocks of the file

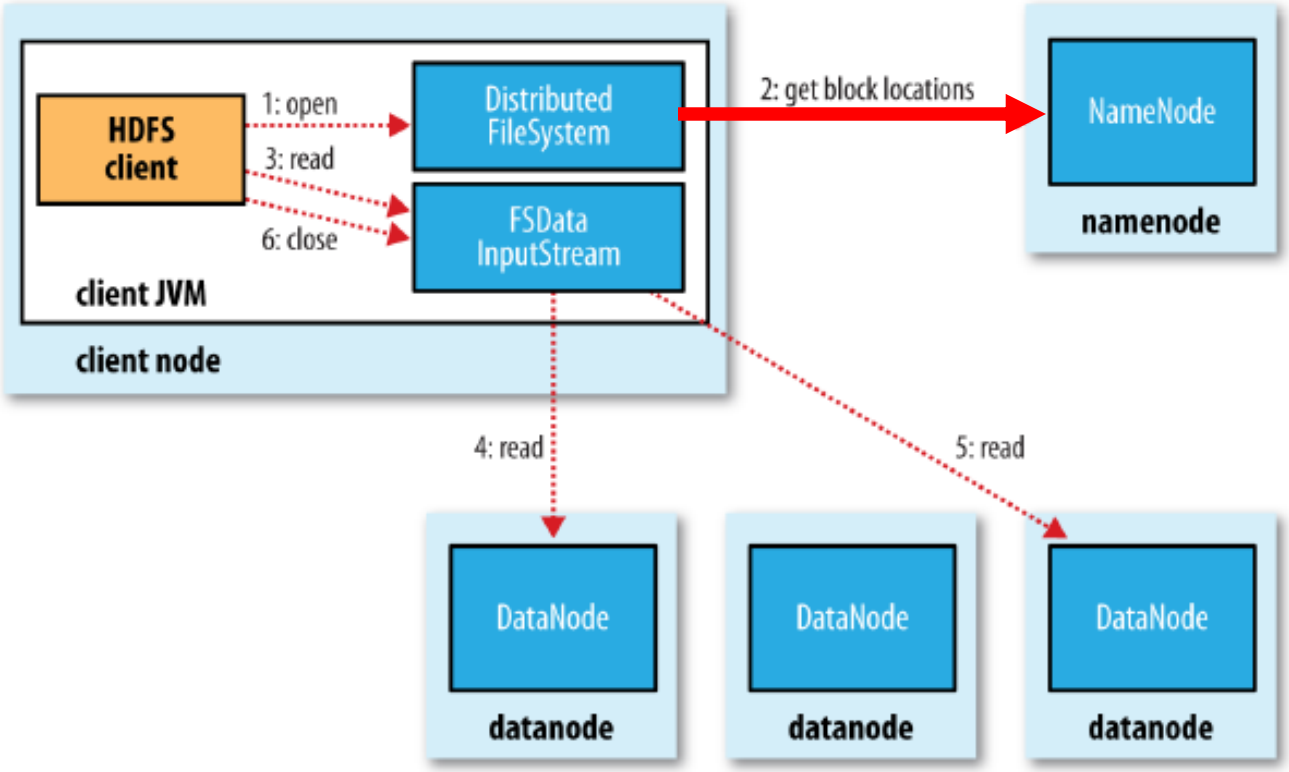
- For each block the namenode returns the addresses of the datanodes with copy of that block



Anatomy of a file read

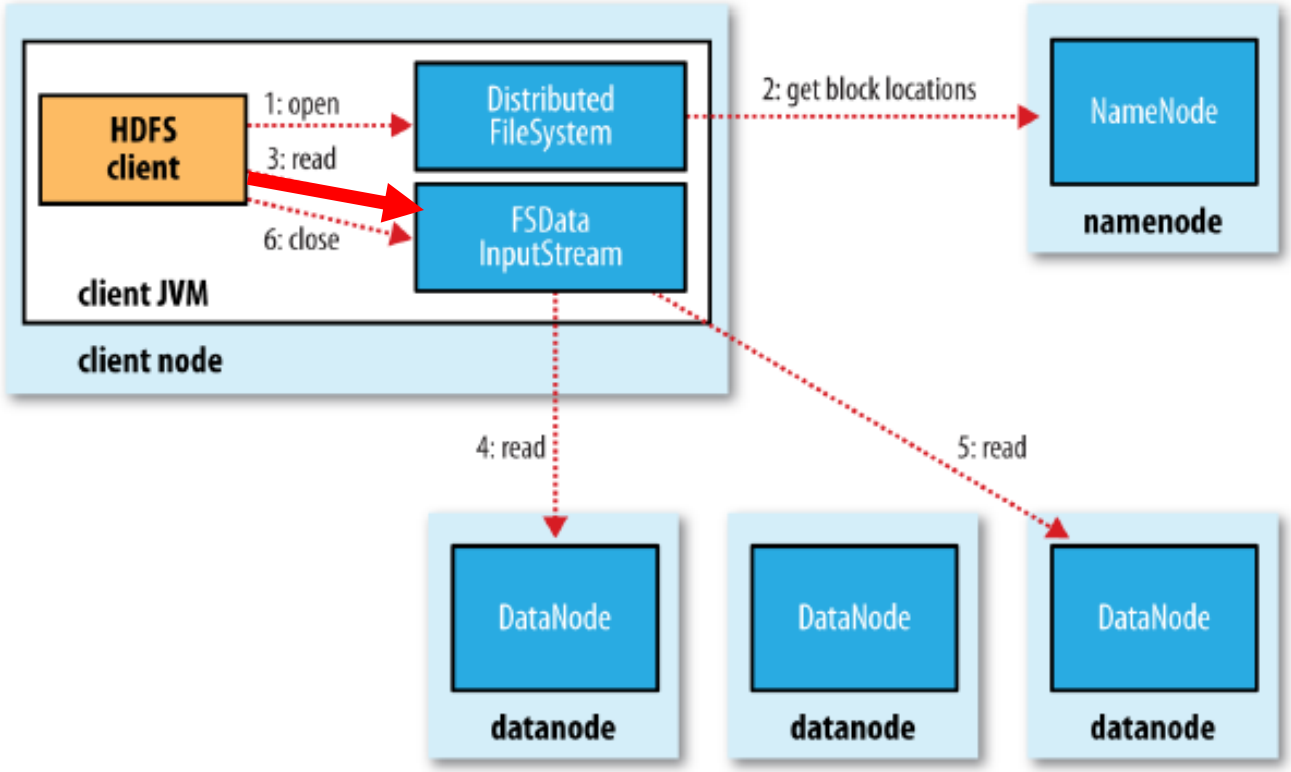
2. The ***DistributedFileSystem*** object calls the namenode, using RPC to determine the first few blocks of the file

- For each block the namenode returns the addresses of the datanodes with copy of that block
- The ***DistributedFileSystem*** object returns ***DFSInputStream*** which manages the datanode and namenode I/O



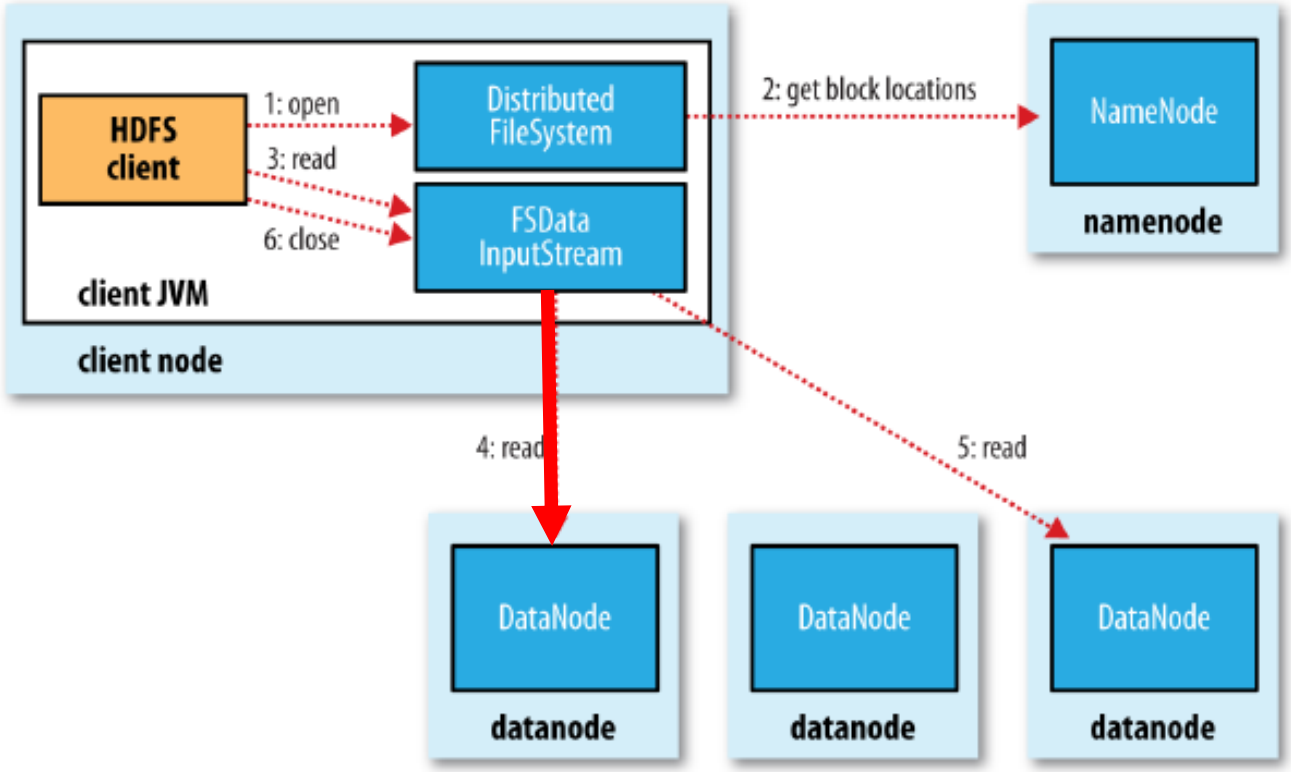
Anatomy of a file read

3. The client calls read() on ***DFSInputStream***, which then connects to the closest datanode for the first few block in the file



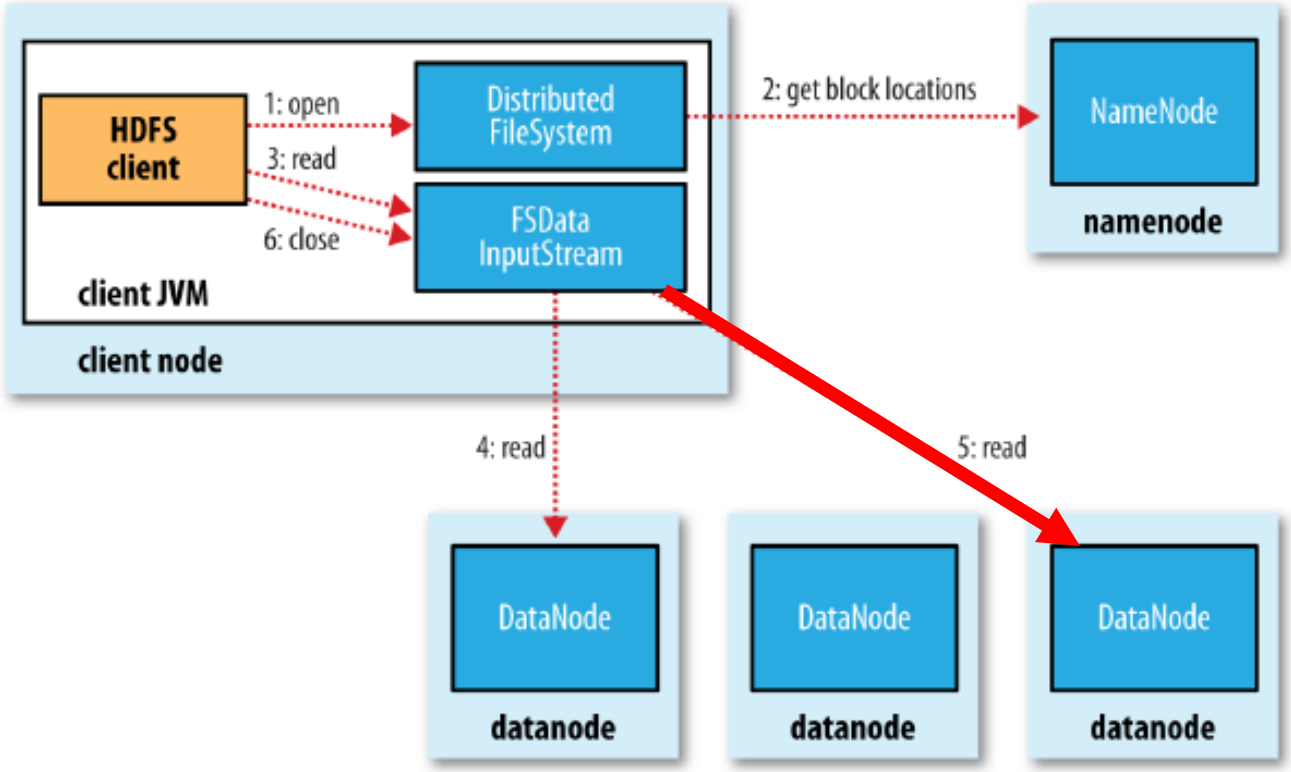
Anatomy of a file read

4. Data is streamed from the datanode back to the client which calls ***read()*** repeatedly



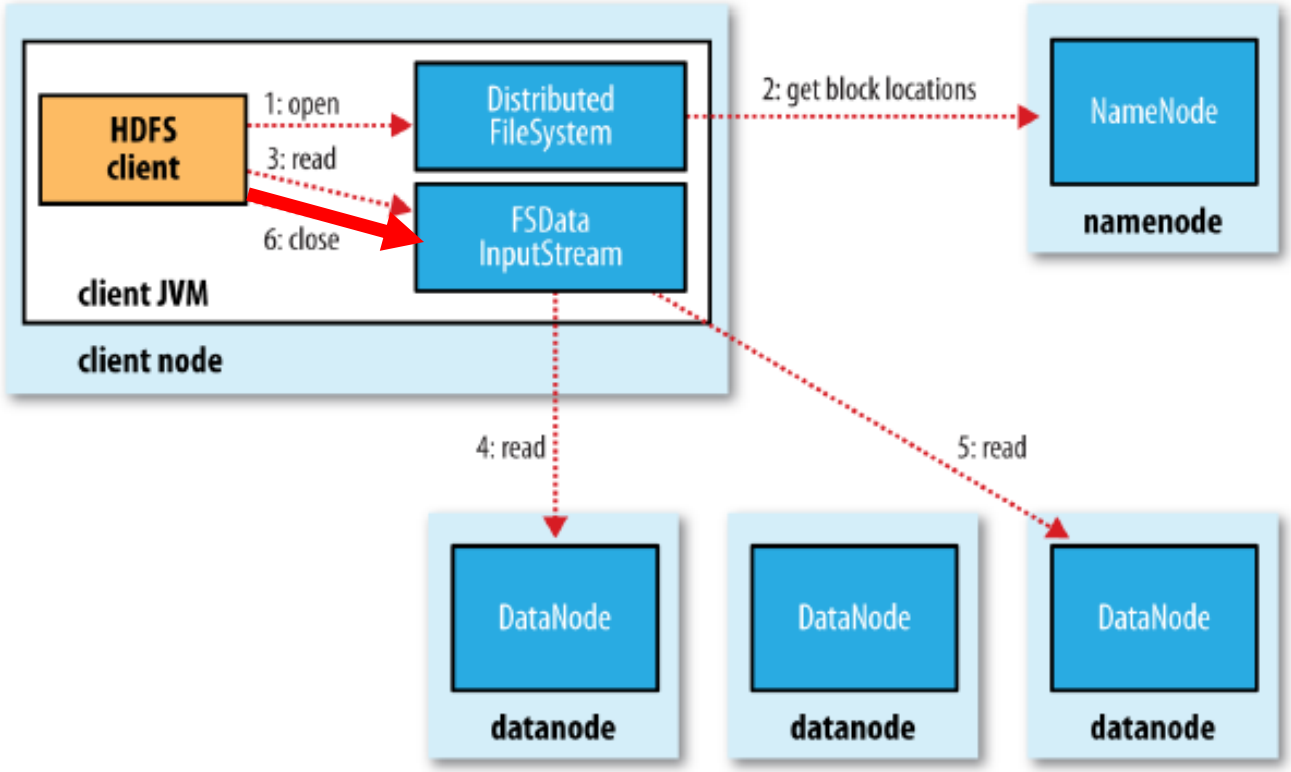
Anatomy of a file read

5. When the end of the block is reached, ***DFSInputStream*** will close the connection to the datanode and find the best node for the next block

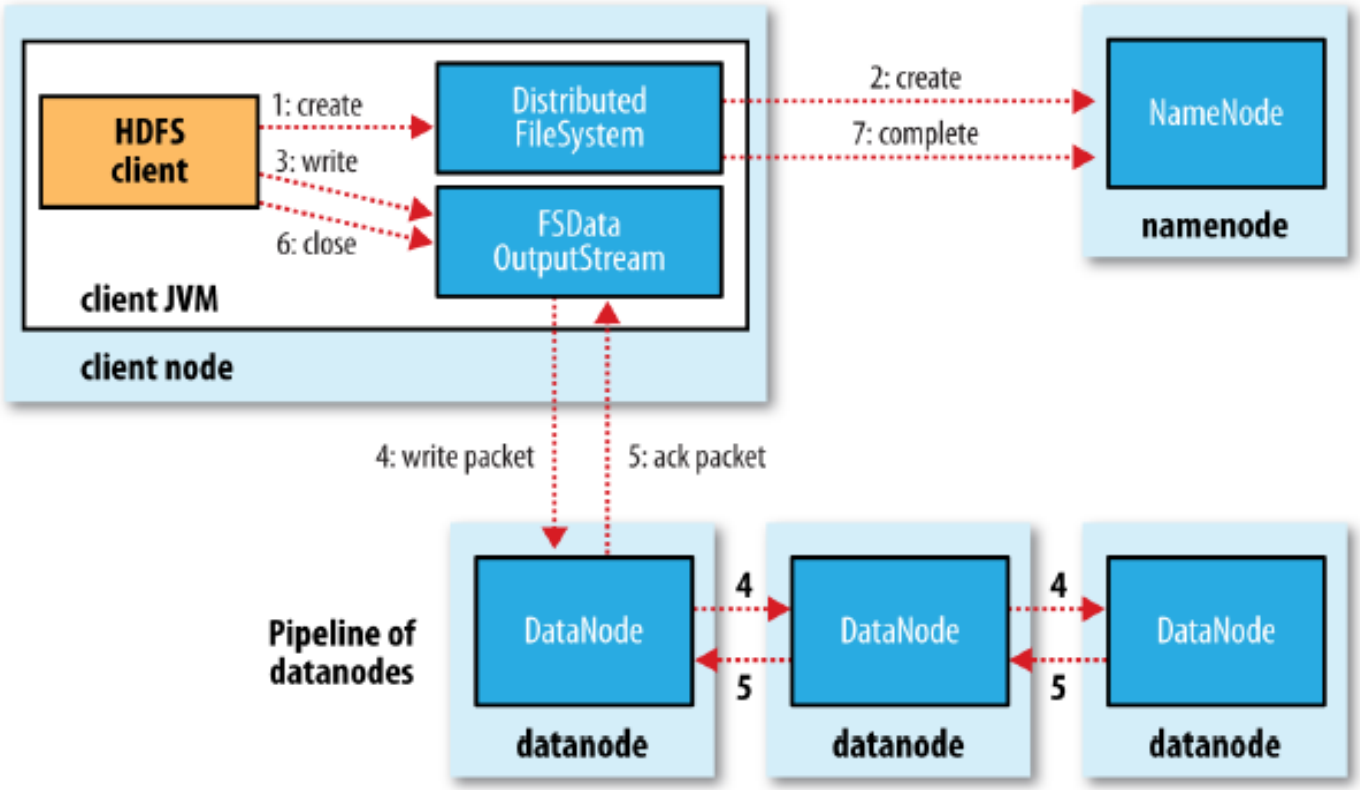


Anatomy of a file read

6. The client calls ***close()*** on ***DFSInputStream*** after finished reading

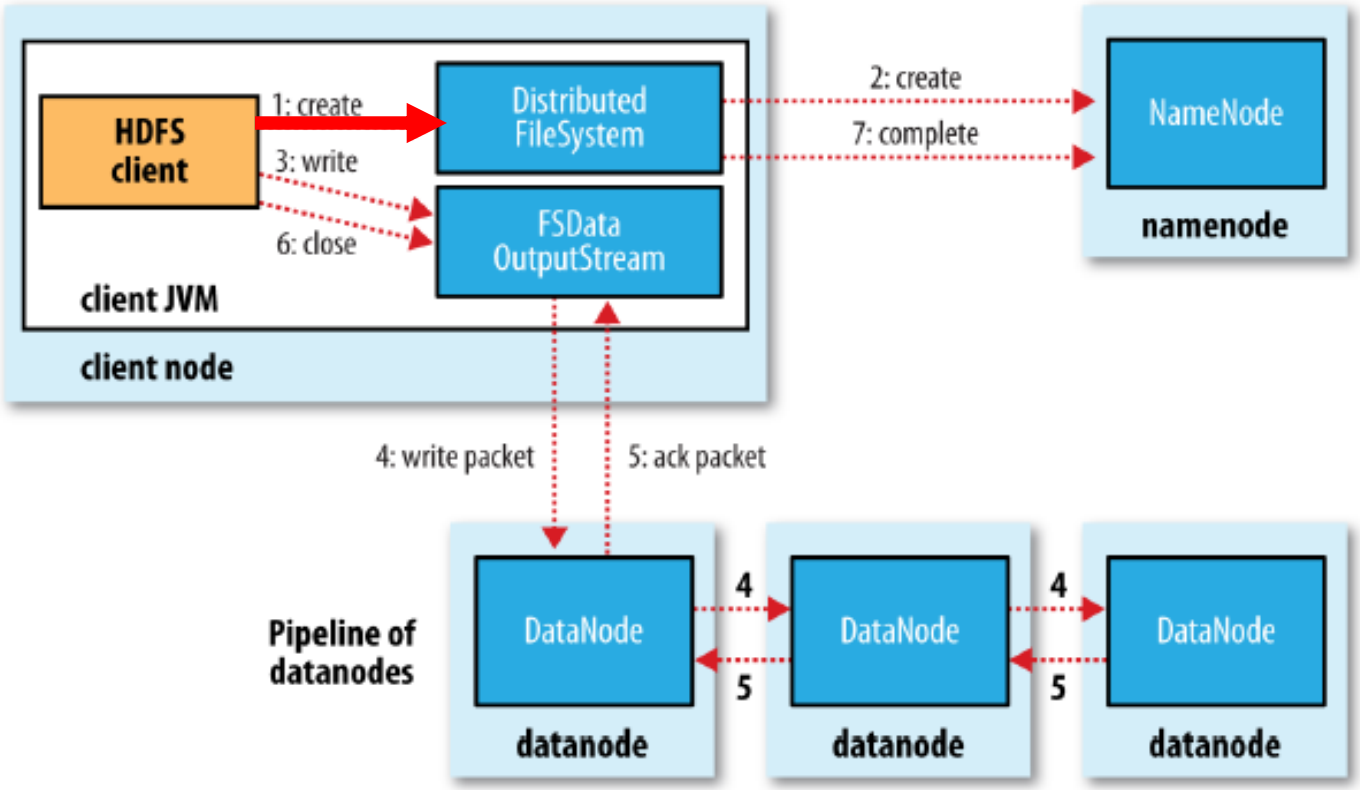


Anatomy of a file write



Anatomy of a file write

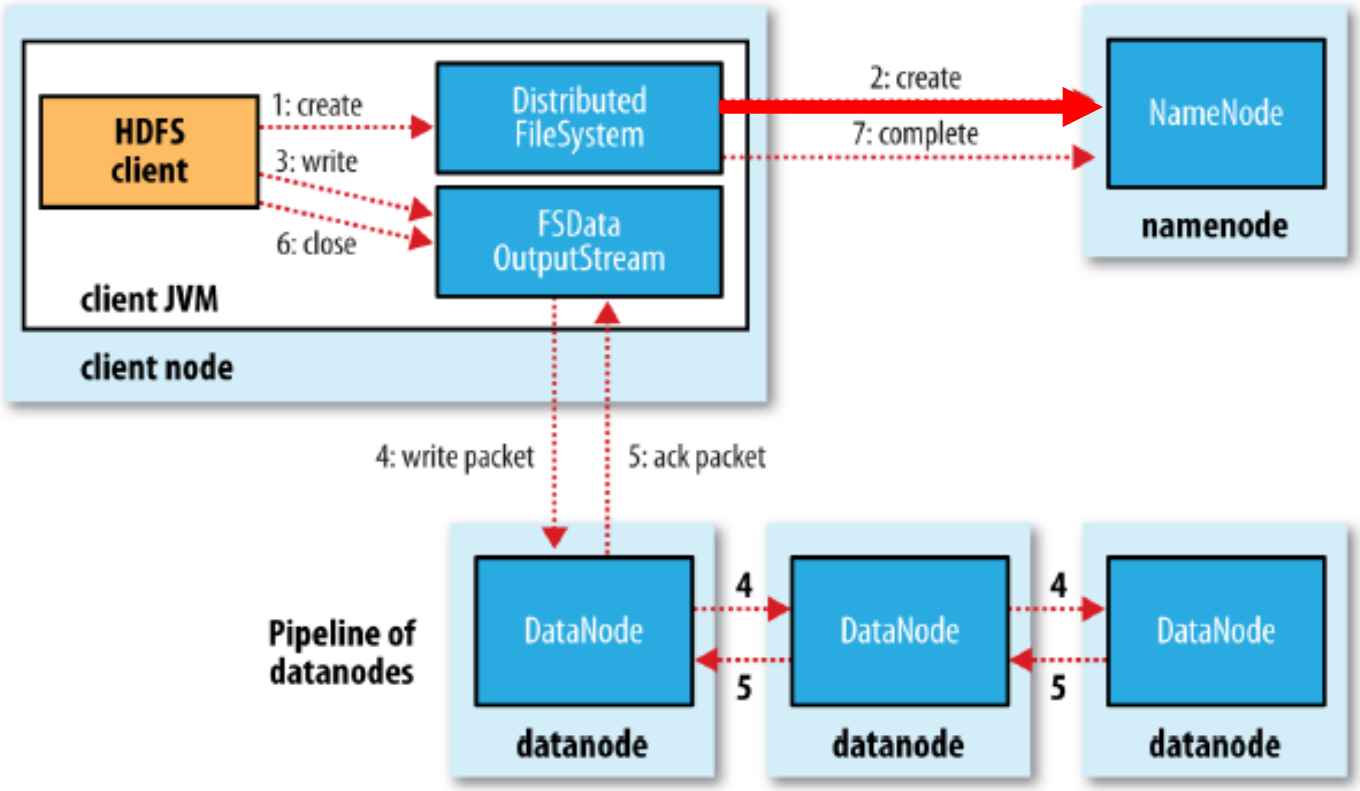
1. The client creates the file by calling ***create()*** on the ***DistributedFileSystem***



Anatomy of a file write

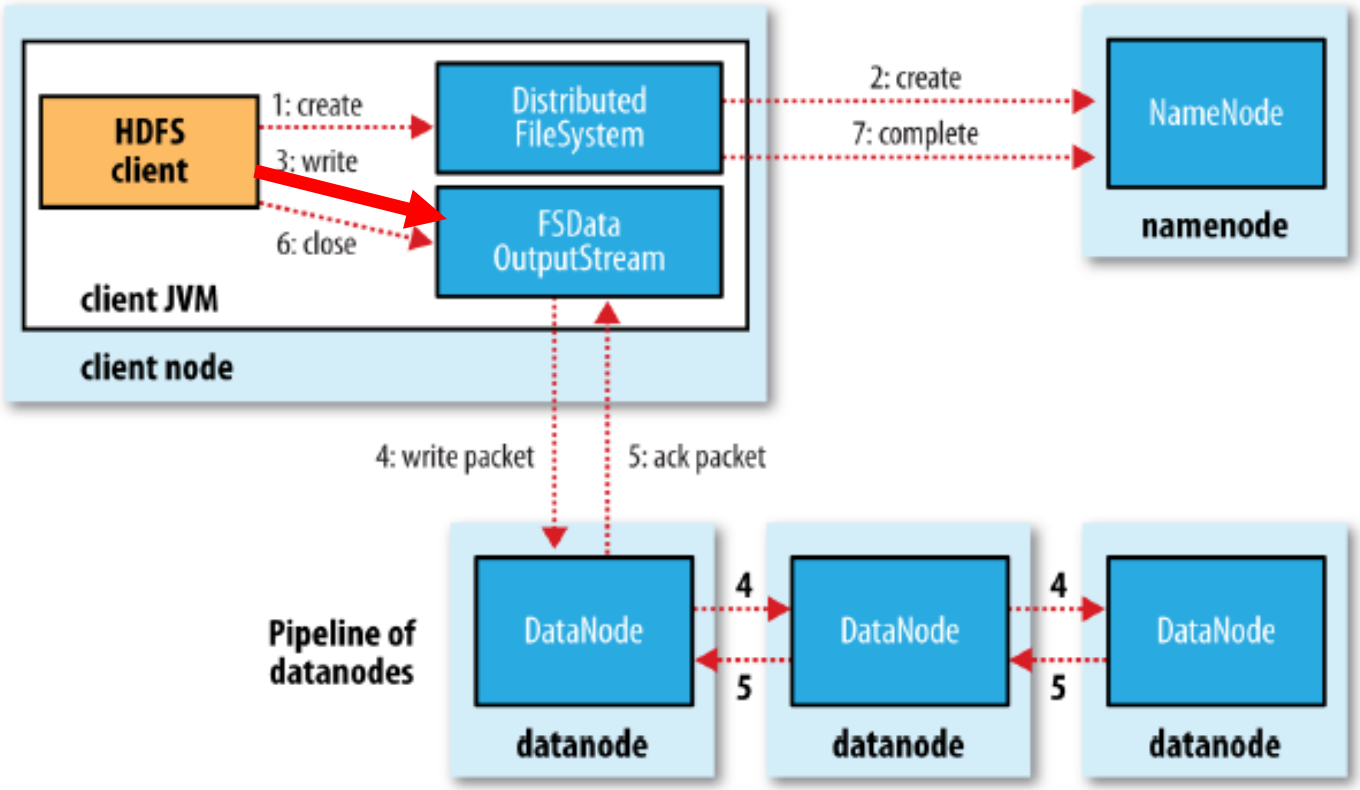
2. The ***DistributedFileSystem object*** makes an RPC call to the namenode to create a new file in the filesystem's namespace.

The ***DistributedFileSystem*** returns ***DFSOutputStream***



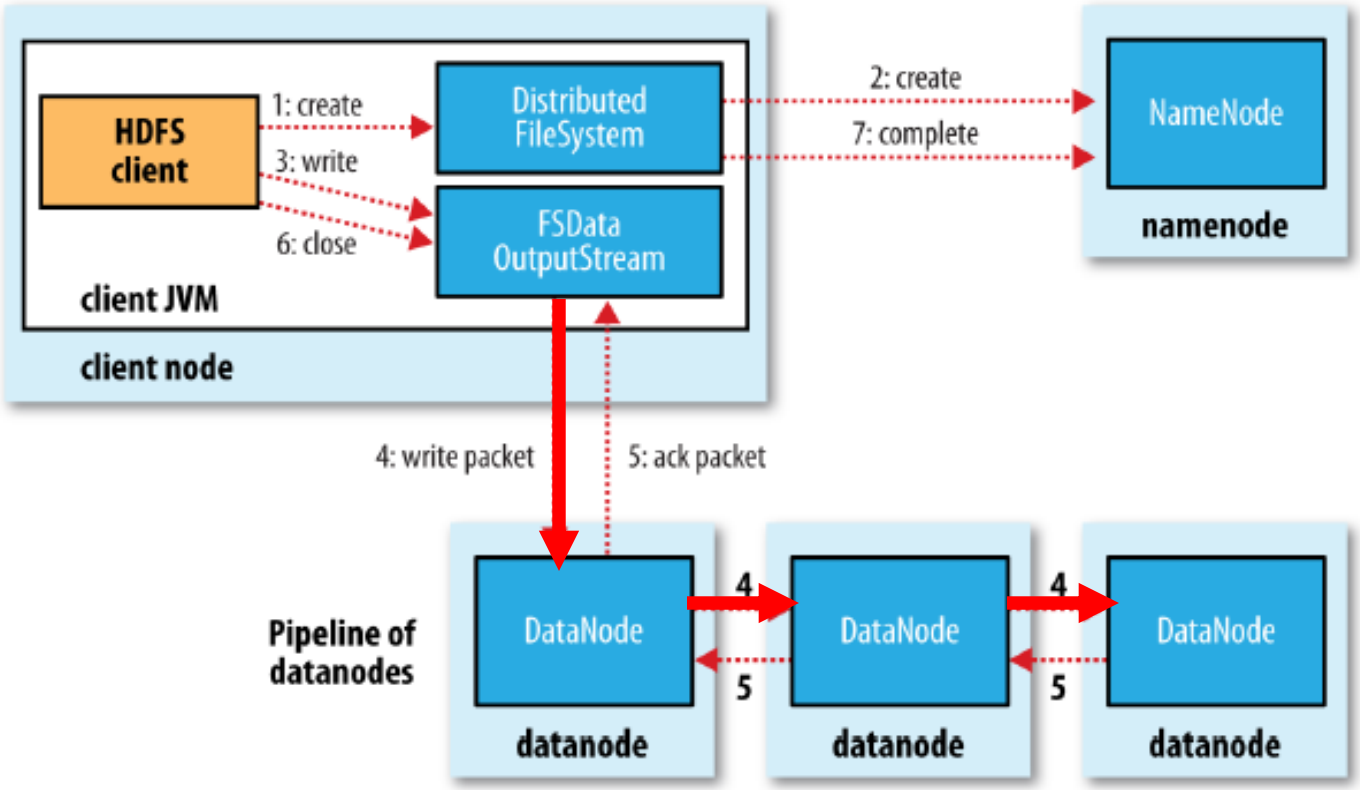
Anatomy of a file write

3. As client writes data, it is split in into packets, which are written into internal queue. The dataqueue is consumed by ***DataStreamer***



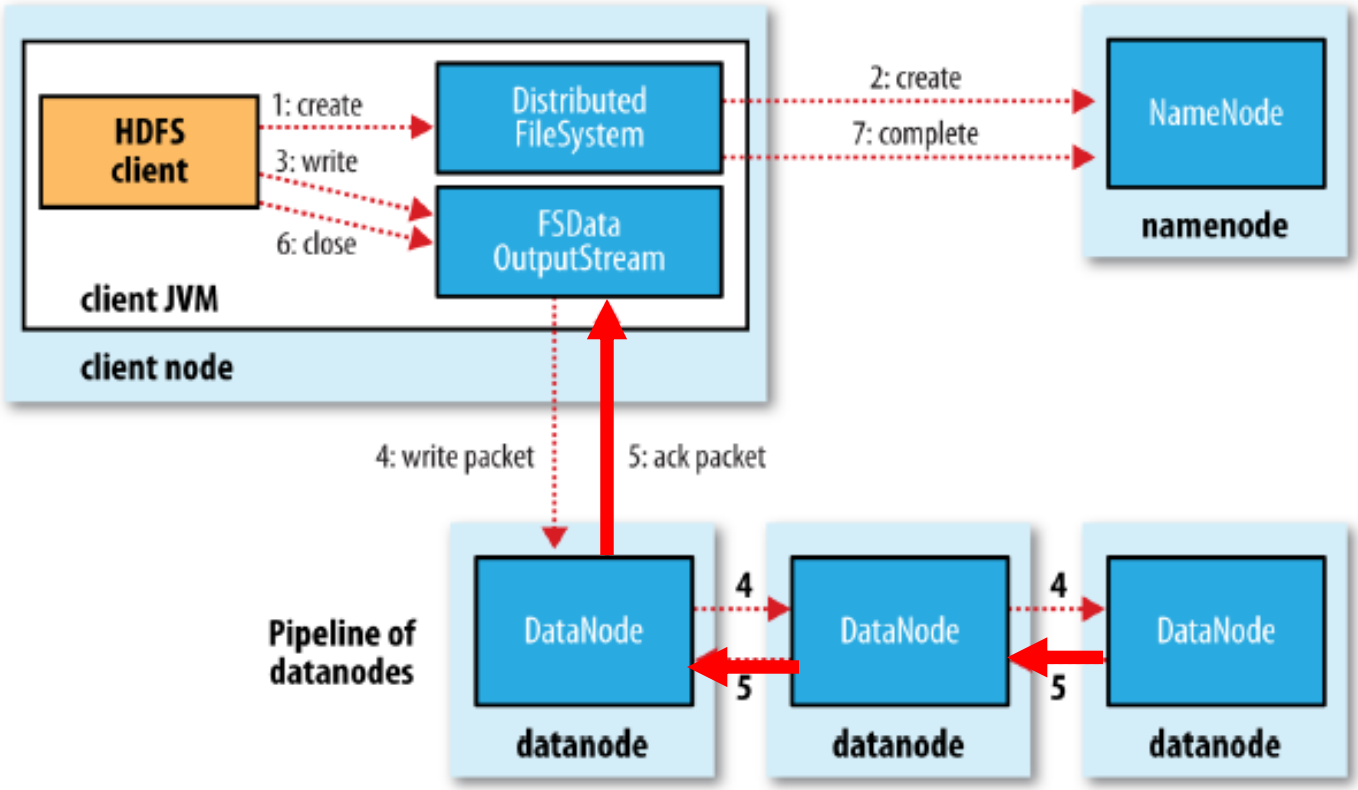
Anatomy of a file write

4. The ***DataStreamer*** streams the packets into first datanode in the pipeline, which stores each packet and forwards it into the second and the second forwards it into the third and so on.



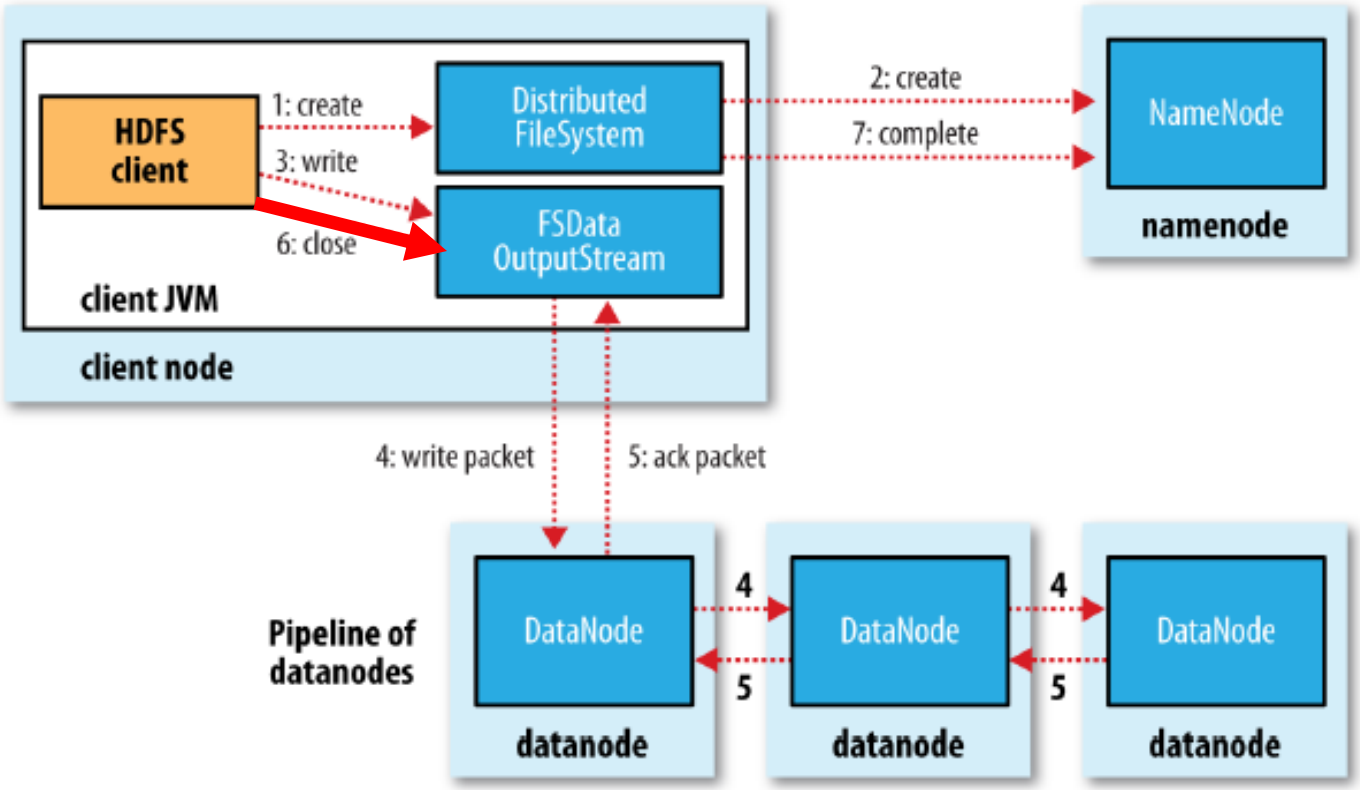
Anatomy of a file write

5. The ***DFSOutputStream*** maintains an internal queue of packets that are waiting to be acknowledged by the datanodes. A packet is removed from the ack queue when it has been acknowledged by all datanodes in the pipeline



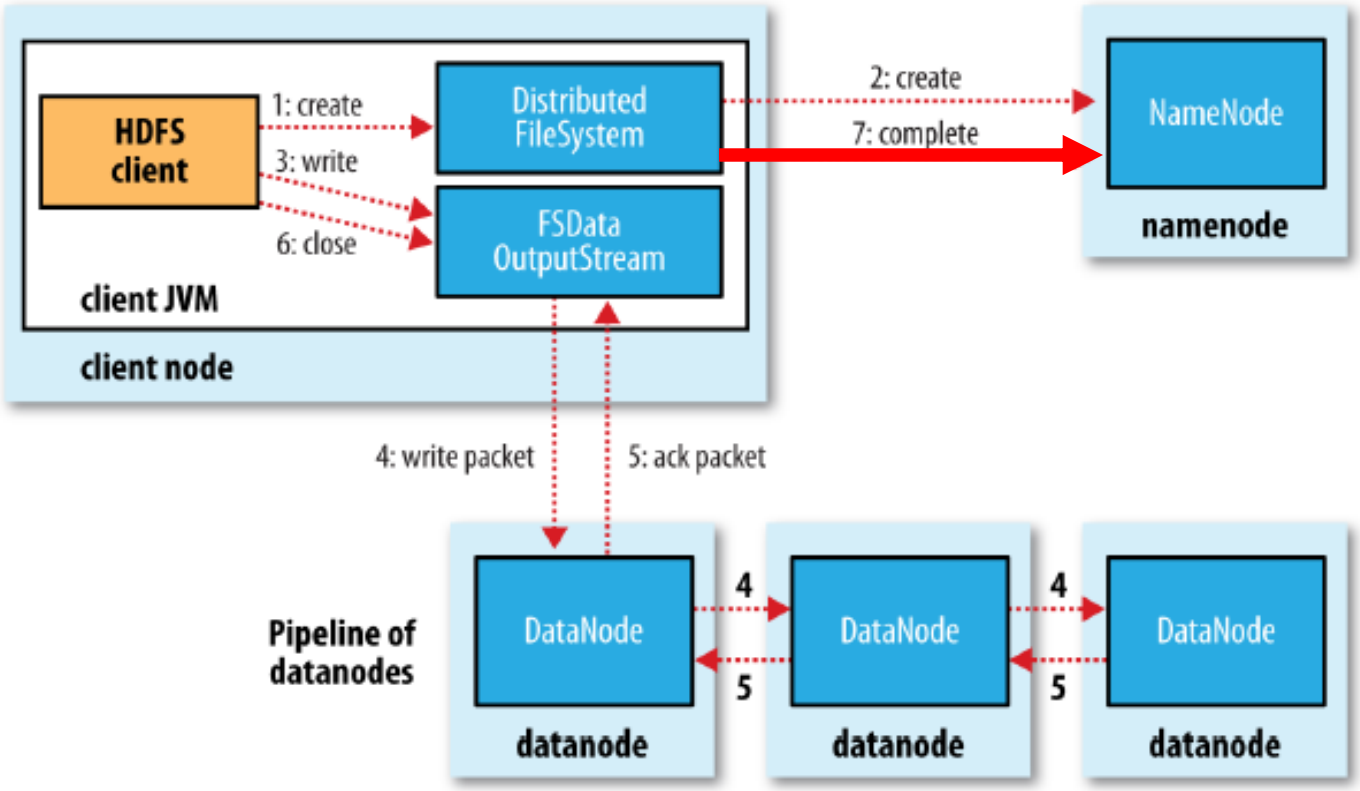
Anatomy of a file write

6. When the client finishes, it calls close() on the stream



Anatomy of a file write

7. Then all the remaining packets are flushed to the datanode pipeline and waits for acknowledgement before contacting the namenode to signal that file is complete

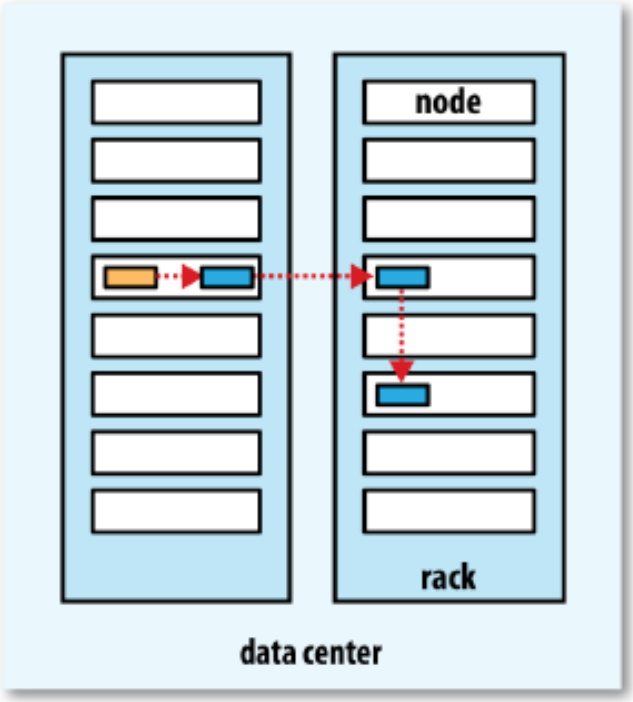


Replica Placement

- Many racks, communication between racks are through switches.
- Network bandwidth between machines on the same rack is greater than those in different racks.
- Optimizing replica placement distinguishes HDFS from other distributed file systems.
- Rack-aware replica placement:
  - Goal: improve reliability, availability and network bandwidth utilization
- Namenode determines the rack id for each DataNode.

Replica Placement Pipeline

- Replicas are typically placed on unique racks
  - Simple but non-optimal
  - Writes are expensive
  - Replication factor is 3



- In HDFS replicas are placed:
  - First replica on the same node as the client
  - Second replica on a different rack
  - Third replica on the same rack as second, but on a different node chosen randomly

- Good balance among reliability, write bandwidth, read performance and block distribution

HDFS Limitations

- HDFS is optimized to provide streaming read performance; this comes at the expense of random seek times to arbitrary positions in files.

- Data will be written to the HDFS once and then read several times; updates to files after they have already been closed are not supported.

- Due to the large size of files, and the sequential nature of reads, the system does not provide a mechanism for local caching of data.

- Individual machines are assumed to fail on a frequent basis, both permanently and intermittently. The cluster must be able to withstand the complete failure of several machines, possibly many happening at the same time.

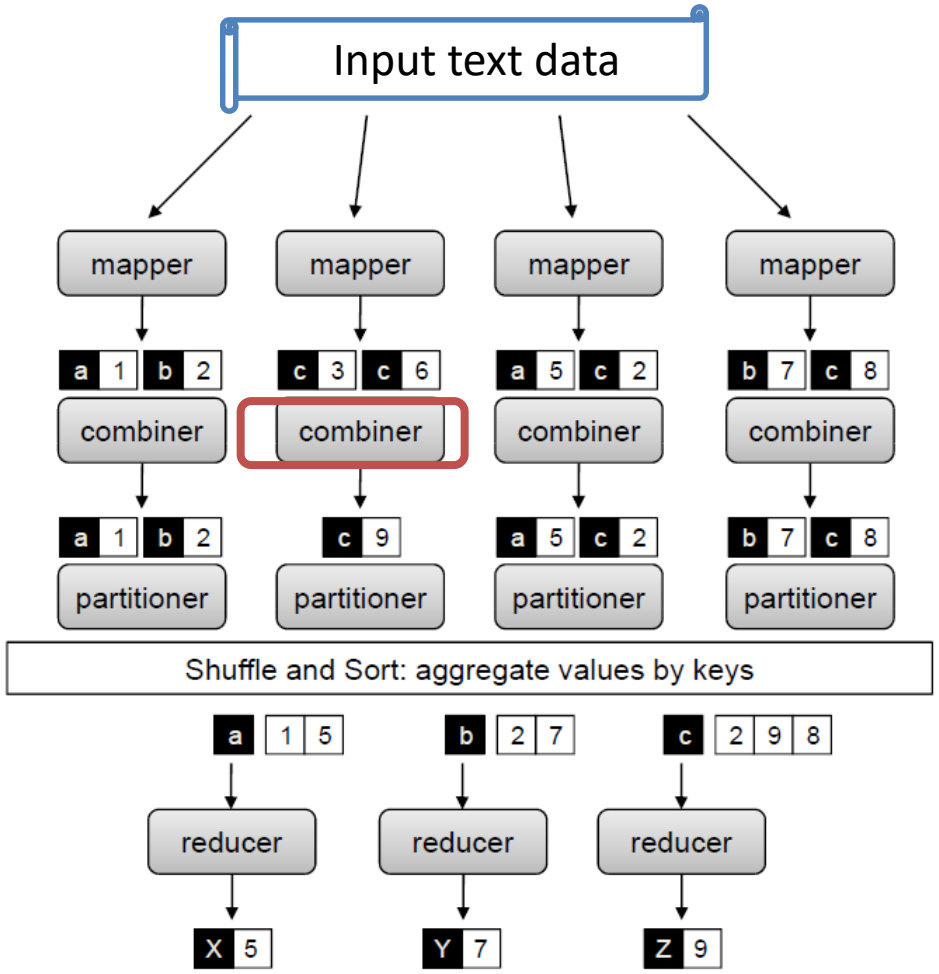
Outline

- Introduction
- Developing a MapReduce Application
- How MapReduce Works
- YARN
- Hadoop Distributed File Systems (HDFS)
- MapReduce Algorithm Design 
  - Local aggregation with Combiners
  - Secondary sorting
  - Join

MapReduce components: combiner

- Output of the mappers are processed by the combiners

- Combiner performs local aggregation to cut down on the number of intermediate key-value pair

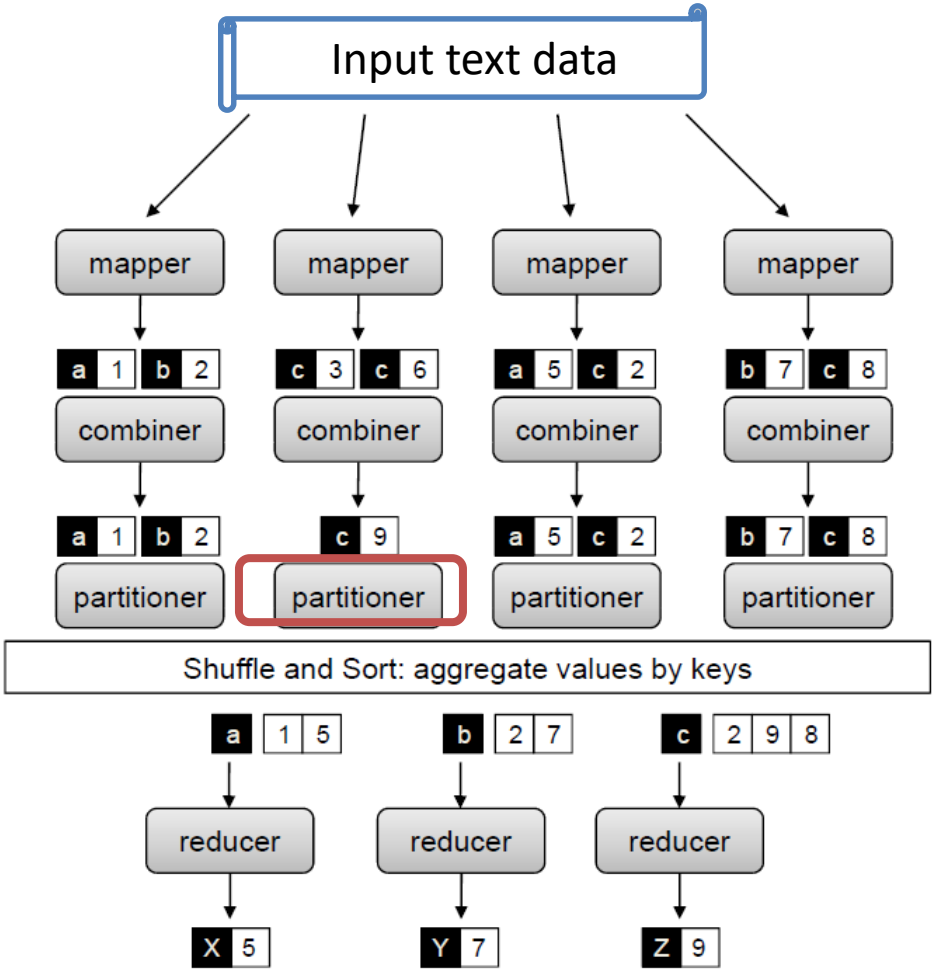


Src: ?

MapReduce components: partitioner

- The partitioner determines which reducer will be responsible for processing a particular key

- Execution framework uses this information to copy the data to the right location during the shuffle and sort phase



Outline

- Introduction
- Developing a MapReduce Application
- How MapReduce Works
- YARN
- Hadoop Distributed File Systems (HDFS)
- MapReduce Algorithm Design
  - Local aggregation with Combiners 
  - Secondary sorting
  - Join

Local aggregation

- A crucial aspect of MapReduce performance is the exchange of intermediate results
- In a cluster environment, this involves transferring data over the network.
  - In Hadoop, intermediate results from Maps are written to local disk before being sent over the network.
  - Since network and disk latencies are relatively expensive, reductions in the amount of intermediate data translate into increases in algorithmic efficiency.
- Local aggregation of intermediate results can improve performance

UNB FREDERICTON

RECALL: Combiner example

- Minimize the data transfer between map and reduce
  - The user can specify a combiner function to be run on the map output
  - Combiner function's output forms the input to the reduce function

- Example: find the max temp per year (with combiner)
  - First map output: (1950, 0), (1950, 20), (1950, 10)
  - First combiner output: (1950, 20)
  - Second map output: (1950, 25), (1950, 15)
  - Second combiner output: (1950, 25)
  - Reduce input: (1950, [20, 25])
  - Reduce output: (1950, 25)

A word count MapReduce program

- Without combiner

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term t ∈ doc d do
4:             Emit(term t, count 1)

1: class Reducer
2:     method Reduce(term t, counts [c1, c2, ...])
3:         sum ← 0
4:         for all count c ∈ counts [c1, c2, ...] do
5:             sum ← sum + c
6:         Emit(term t, count sum)
```

Pseudo-code for the basic word count algorithm in MapReduce

A word count MapReduce program

- With an "in-mapper combiner"

```
1: class Mapper
2:         method Setup
3:                 H ← new AssociativeArray
4:         method Map(docid a; doc d)
5:                 for all term t Є doc d do
6:                 H{t} ← H{t} + 1 . // Tally counts across documents
7:         method Cleanup
8:                 for all term t Є H do
9:                 Emit(term t; count H{t})
```

Pseudo-code to demonstrate in-mapper combining. Reducer is the same

- Before processing any input, the mapper's *Setup* is called
- It preserves state across multiple calls of the *Map* method and allows accumulating partial term counts in the associative array across multiple documents
- The key-value pairs are emitted only when the mapper has processed all documents and is done in the *Cleanup* method in the pseudo-code

Outline

- Introduction
- Developing a MapReduce Application
- How MapReduce Works
- YARN
- Hadoop Distributed File Systems (HDFS)
- MapReduce Algorithm Design
  - Local aggregation with Combiners
  - Secondary sorting 
  - Join

Why secondary sorting?

- MapReduce sorts intermediate key-value pairs by the keys during the shuffle and sort phase
  - The order in which the values appear is not stable, because they come from different map tasks, which may vary from run to run
  - What if in addition to sorting by key, we also need to sort by value?

- Revisiting the code to find max temperature for each year
  - If we arranged for the temperatures to be sorted in descending order, we wouldn't have to iterate to find the maximum
  - Instead, we could take the first for each year and ignore the rest.

```
1900 35°C
1900 34°C
1900 34°C
...
1901 36°C
1901 35°C
```

"Value-to-key" conversion

- In many applications we wish to first group together data one way (e.g., by year), and then sort within the groupings another way (e.g., by temperature)

- Idea: move part of the value into the intermediate key to form a composite key, and let the MapReduce execution framework handle the sorting.

- In the example, instead of emitting the *year* as the key, we can emit the *year* and the *temperature* as a composite key:

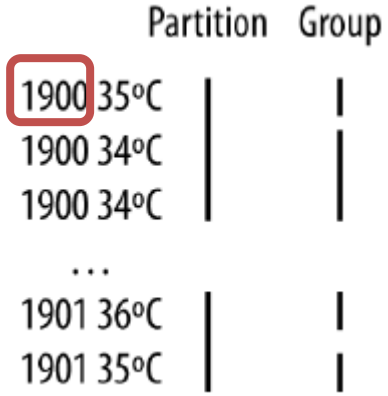
  (year, temperature)

- We want the sort order for keys to be by *year* (ascending) and then by *temperature* (descending)

Partitioner

- With a composite key (year, temperature), records for the same year would have different keys and therefore would not go to the same reducer.
  - For example, (1900, 35°C) and (1900, 34°C) could go to different reducers.

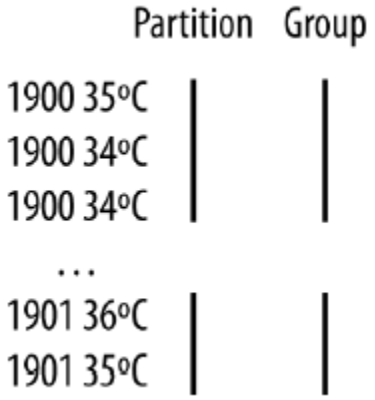
- By setting a partitioner to partition by the year part of the key, we guarantee that records for the same year go to the same reducer.



Grouping

- A partitioner ensures only that one reducer receives all the records for a year; it doesn't change the fact that the reducer groups by key within the partition.

- If we group values in the reducer by the year part of the key, we will see all the records for the same year in one reduce group.
  - Because they are sorted by temperature in descending order, the first is the maximum temperature:



Summary of secondary sorting with Hadoop

- Make the key a composite of the natural key and the natural value.
- The sort comparator should order by the composite key (i.e., the natural key and natural value).
- The partitioner and grouping comparator for the composite key should consider only the natural key for partitioning and grouping.

Outline

- Introduction
- Developing a MapReduce Application
- How MapReduce Works
- YARN
- Hadoop Distributed File Systems (HDFS)
- MapReduce Algorithm Design
  - Local aggregation with Combiners
  - Secondary sorting
  - Join 

Join

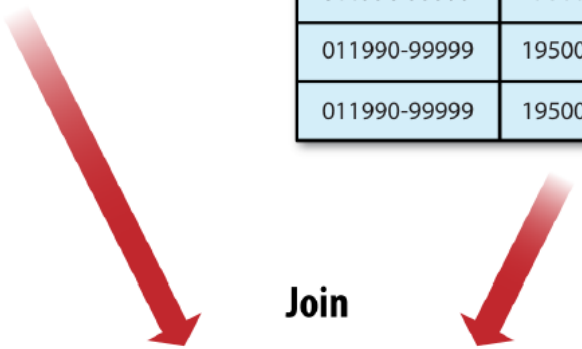
- Combining two datasets on a common key

Stations

Station ID	Station Name
011990-99999	SIHCCAJAVRI
012650-99999	TYNSET-HANSMOEN

Records

Station ID	Timestamp	Temperature
012650-99999	194903241200	111
012650-99999	194903241800	78
011990-99999	195005150700	0
011990-99999	195005151200	22
011990-99999	195005151800	-11



Join

Station ID	Station Name	Timestamp	Temperature
011990-99999	SIHCCAJAVRI	195005150700	0
011990-99999	SIHCCAJAVRI	195005151200	22
011990-99999	SIHCCAJAVRI	195005151800	-11
012650-99999	TYNSET-HANSMOEN	194903241200	111
012650-99999	TYNSET-HANSMOEN	194903241800	78

Outline

- Introduction
- Developing a MapReduce Application
- How MapReduce Works
- YARN
- Hadoop Distributed File Systems (HDFS)
- MapReduce Algorithm Design
  - Local aggregation with Combiners
  - Secondary sorting
  - Join
    - Map-side join 

Map-side join

- A map-side join between large inputs works by preprocessing data before the data reaches the map function

- The inputs to each map must be partitioned and sorted in a particular way
  - Each input dataset must be divided into the same number of partitions, and it must be sorted by the same key (the join key) in each source.
  - All the records for a particular key must reside in the same partition.

Outline

- Introduction
- Developing a MapReduce Application
- How MapReduce Works
- YARN
- Hadoop Distributed File Systems (HDFS)
- MapReduce Algorithm Design
  - Local aggregation with Combiners
  - Secondary sorting
  - Join
    - Map-side join
    - Reduce-side join 

Reduce-side join

- A reduce-side join is more general than a map-side join, but it is less efficient

- **Idea:** the mapper tags each record with its source and uses the join key as the map output key, so that the records with the same key are brought together in the reducer.

Multiple inputs

- We may have data sources that provide the same type of data but in different formats. This also arises in the case of performing joins of different datasets

- *MultipleInputs* class allows us to specify which *InputFormat* and *Mapper* to use on a per-path basis.
  - For example, if we had weather data from the UK Met Office that we wanted to combine with the NCDC data for our maximum temperature analysis, we might set up the input as follows:

```
MultipleInputs.addInputPath(job, ncdcInputPath,
    TextInputFormat.class, MaxTemperatureMapper.class);
MultipleInputs.addInputPath(job, metOfficeInputPath,
    TextInputFormat.class, MetOfficeMaxTemperatureMapper.class);
```

  - This code replaces the usual calls to *FileInputFormat.addInputPath()* and *job.setMapperClass().*

Secondary sorting

- The reducer will see the records from both sources that have the same key, but they are not guaranteed to be in any particular order.
- To perform the join, it is important to have the data from one source before that from the other.
- For the weather data join, the Stations record must be the first of the values seen for each key, so the reducer can fill in the weather records with the station name and emit them straightaway
- Then the Records record is processed by the reducer

Stations

Station ID	Station Name
011990-99999	SIHCCAJAVRI

Records

Station ID	Timestamp	Temperature
012650-99999	194903241200	111

Tagging records

- The only requirement for the tag values is that they sort in such a way that station records come before the weather records.
  - This is done by tagging station records as 0 and weather records as 1
- Mapper for tagging station records for a reduce-side join

```
public class JoinStationMapper
    extends Mapper<LongWritable, Text, TextPair, Text> {
  private NcdcStationMetadataParser parser = new NcdcStationMetadataParser();

  @Override
  protected void map(LongWritable key, Text value, Context context)
      throws IOException, InterruptedException {
    if (parser.parse(value)) {
      context.write(new TextPair(parser.getStationId(), "0"),
          new Text(parser.getStationName()));
    }
  } }
```

- Mapper for tagging weather records for a reduce-side join

```
public class JoinRecordMapper
    extends Mapper<LongWritable, Text, TextPair, Text> {
  private NcdcRecordParser parser = new NcdcRecordParser();

  @Override
  protected void map(LongWritable key, Text value, Context context)
      throws IOException, InterruptedException {
    parser.parse(value);
    context.write(new TextPair(parser.getStationId(), "1"), value);
  } }
```

Join reducer

- The reducer knows that it will receive the station record first, so it extracts its name from the value and writes it out as part of output
  - Reducer for joining tagged station records with tagged weather records

```
public class JoinReducer extends Reducer<TextPair, Text, Text, Text> {

  @Override
  protected void reduce(TextPair key, Iterable<Text> values, Context context)

      throws IOException, InterruptedException {
    Iterator<Text> iter = values.iterator();
    Text stationName = new Text(iter.next());
    while (iter.hasNext()) {
      Text record = iter.next();
      Text outValue = new Text(stationName.toString() + "\t" + record.toString());
      context.write(key.getFirst(), outValue);
    }
  } }
```

- The code assumes that every station ID in the weather records has exactly one matching record in the station dataset.
  - If this were not the case, we would need to generalize the code to put the tag into the value objects, by using another *TextPair*. The reduce() method would then be able to tell which entries were station names and detect (and handle) missing or duplicate entries before processing the weather records.

The join driver

- We partition and group on the first part of the key, the station ID, which is done with a custom Partitioner (*KeyPartitioner*) and a custom group comparator, *FirstComparator* (from *TextPair*).

```
public class JoinRecordWithStationName extends Configured implements Tool {

  public static class KeyPartitioner extends Partitioner<TextPair, Text> {
    @Override
    public int getPartition(TextPair key, Text value, int numPartitions) {
      return (key.getFirst().hashCode() & Integer.MAX_VALUE) % numPartitions;
    }
  }

  @Override
  public int run(String[] args) throws Exception {
    if (args.length != 3) {
      JobBuilder.printUsage(this, "<ncdc input> <station input> <output>");
      return -1;
    }

    Job job = new Job(getConf(), "Join weather records with station names");
```

The join driver (contd.)

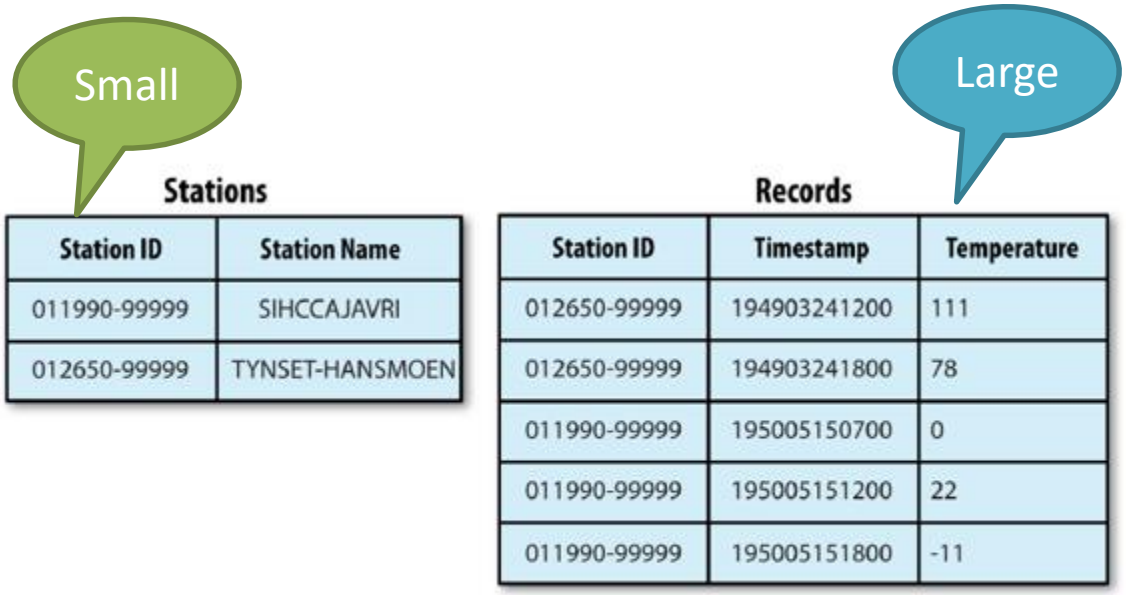
```
    job.setJarByClass(getClass());

    Path ncdcInputPath = new Path(args[0]);
    Path stationInputPath = new Path(args[1]);
    Path outputPath = new Path(args[2]);

    MultipleInputs.addInputPath(job, ncdcInputPath,
        TextInputFormat.class, JoinRecordMapper.class);
    MultipleInputs.addInputPath(job, stationInputPath,
        TextInputFormat.class, JoinStationMapper.class);
    FileOutputFormat.setOutputPath(job, outputPath);

    job.setPartitionerClass(KeyPartitioner.class);
    job.setGroupingComparatorClass(TextPair.FirstComparator.class);

    job.setMapOutputKeyClass(TextPair.class);

    job.setReducerClass(JoinReducer.class);

    job.setOutputKeyClass(Text.class);

    return job.waitForCompletion(true) ? 0 : 1;
  }

  public static void main(String[] args) throws Exception {
    int exitCode = ToolRunner.run(new JoinRecordWithStationName(), args);
    System.exit(exitCode);
  }
}
```

Outline

- Introduction
- Developing a MapReduce Application
- How MapReduce Works
- YARN
- Hadoop Distributed File Systems (HDFS)
- MapReduce Algorithm Design
  - Local aggregation with Combiners
  - Secondary sorting
  - Join
    - Map-side join
    - Reduce-side join
    - Side data distribution 

Side Data Distribution

- If one dataset is large (the weather records) but the other one is small enough to be distributed to each node in the cluster (as the station metadata is), the join can be effected by a MapReduce job that brings the records for each small dataset record together



Side Data Distribution

- If one dataset is large (the weather records) but the other one is small enough to be distributed to each node in the cluster (as the station metadata is), the join can be effected by a MapReduce job that brings the records for each small dataset record together

- The mapper or reducer uses the smaller dataset to look up the station metadata for a station ID, so it can be written out with each record.

- *Side data* can be defined as extra read-only data needed by a job to process the main dataset

Distributed cache

- We can specify the files to be distributed as a comma-separated list of URIs as the argument to the *-files* option.
  - Files can be on the local filesystem, on HDFS, or on another Hadoop-readable filesystem (such as S3)
  - Example of using distributed cache to share a metadata file for station names

```
$ hadoop jar hadoop-examples.jar \
MaxTemperatureByStationNameUsingDistributedCacheFile \
-files input/ncdc/metadata/stations-fixed-width.txt input/ncdc/all output
```

Example application using distributed cache

- Application to find the maximum temperature by station, showing station names from a lookup table passed as a distributed cache file

```
public class MaxTemperatureByStationNameUsingDistributedCacheFile
    extends Configured implements Tool {

  static class StationTemperatureMapper
      extends Mapper<LongWritable, Text, Text, IntWritable> {

    private NcdcRecordParser parser = new NcdcRecordParser();

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

      parser.parse(value);
      if (parser.isValidTemperature()) {
        context.write(new Text(parser.getStationId()),
            new IntWritable(parser.getAirTemperature()));
      }
    }
  }
```

  - The mapper (*StationTemperatureMapper*) simply emits (station ID, temperature) pairs.

Example application using distributed cache (cont.)

– We use the reducer's *setup()* method to retrieve the cache file using its original name, relative to the working directory of the task.

```
static class MaxTemperatureReducerWithStationLookup
    extends Reducer<Text, IntWritable, Text, IntWritable> {

  private NcdcStationMetadata metadata;

  @Override
  protected void setup(Context context)
      throws IOException, InterruptedException {
    metadata = new NcdcStationMetadata();
    metadata.initialize(new File("stations-fixed-width.txt"));
  }

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

    String stationName = metadata.getStationName(key.toString());

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

Example application using distributed cache (cont.)

- For the combiner, we reuse MaxTemperatureReducer (described earlier) to pick the maximum temperature for any given group of map outputs on the map side.

- The reducer (MaxTemperatureReducerWithStationLookup) in addition to finding the maximum temperature, uses the cache file to look up the station name.

```
  }

  @Override
  public int run(String[] args) throws Exception {
    Job job = JobBuilder.parseInputAndOutput(this, getConf(), args);
    if (job == null) {
      return -1;
    }

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

    job.setMapperClass(StationTemperatureMapper.class);
    job.setCombinerClass(MaxTemperatureReducer.class);
    job.setReducerClass(MaxTemperatureReducerWithStationLookup.class);

    return job.waitForCompletion(true) ? 0 : 1;
  }

  public static void main(String[] args) throws Exception {
    int exitCode = ToolRunner.run(
        new MaxTemperatureByStationNameUsingDistributedCacheFile(), args);
    System.exit(exitCode);
  }
}
```