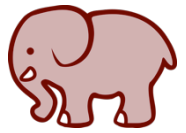
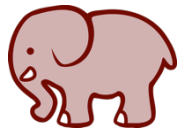
Apache Hadoop – A Developer's Course

An introduction to Big Data



What defines Big Data?

- Volume
  - Petabytes and exabytes (10^{15} bytes = 1 petabyte, 10^{18} bytes = 1 exabyte)
- Variety
  - Any type of data (text, XML, binary, structured, un-structured)
- Velocity
  - Speed at which it is collected (Social web, Satellite data)



Big Data includes all types of Data

Structured

- Pre-defined schema
- e.g. Relational schema

Semi-structured

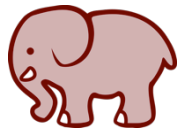
- Inconsistent structure
- Cannot be stored in rows and tables
- e.g. logs, tweets, sensor feeds

Unstructured

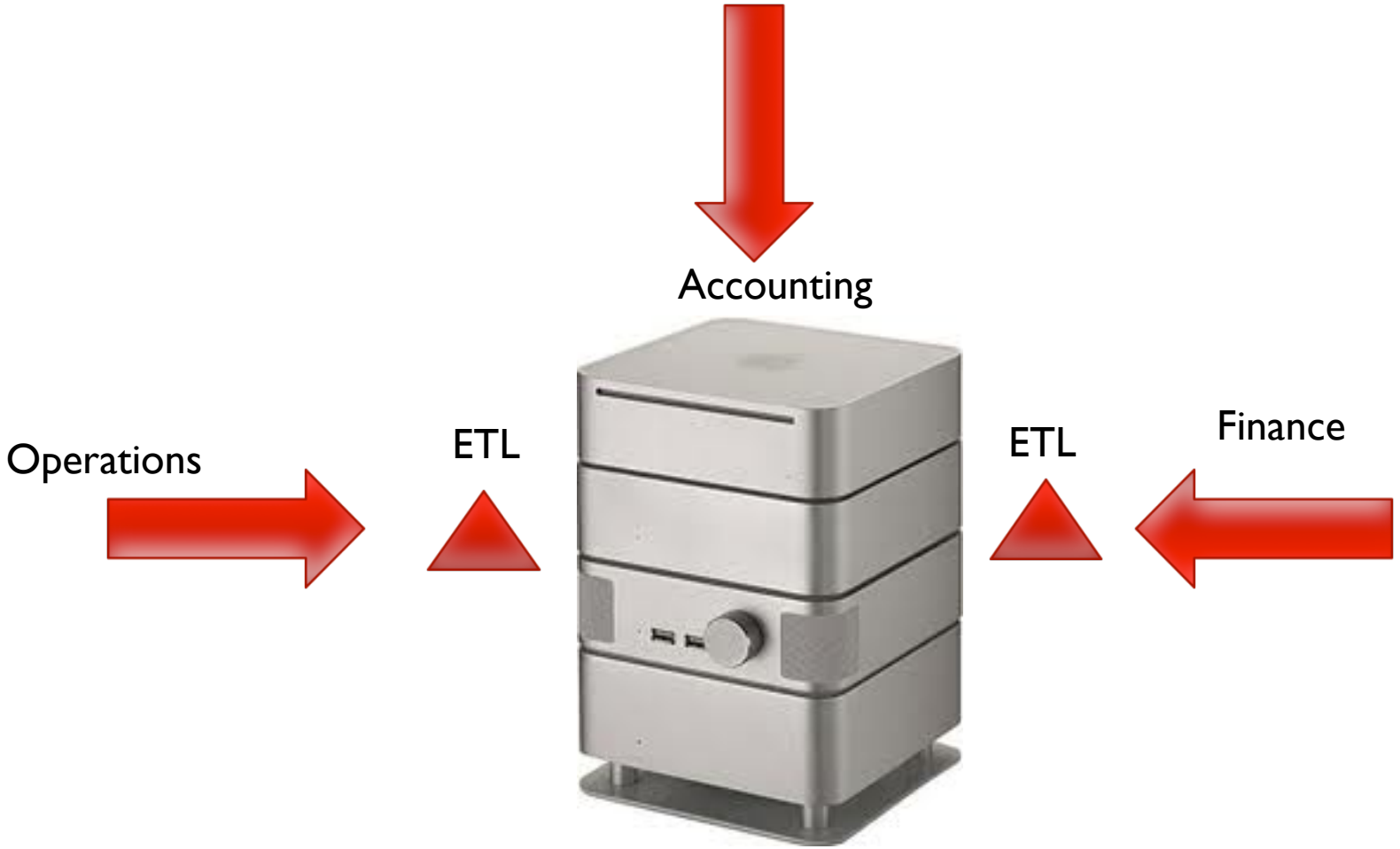
- Lacks structure (could be partly)
- e.g. free-form, text, reports, feedback forms

Time-based

Immutable

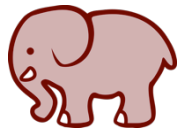


The Legacy Solution



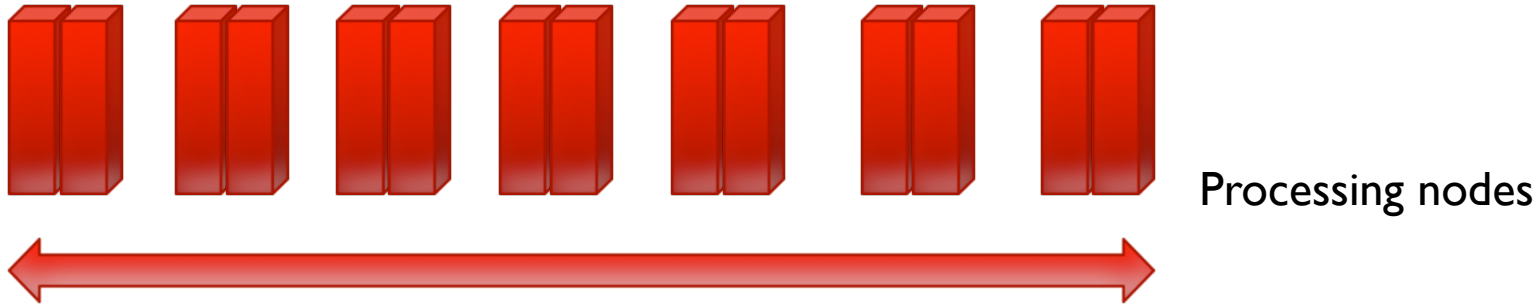
Problems with Legacy Solution

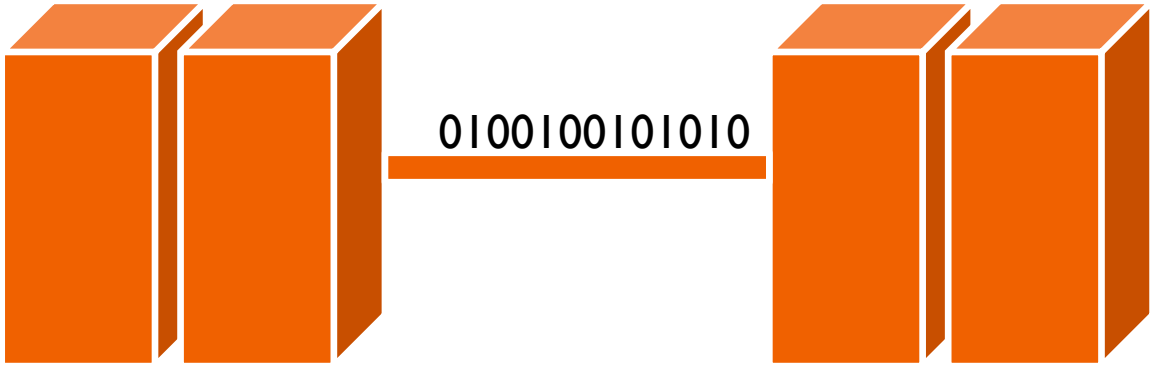
- Data collection is Expensive
- Data is Rigid
- Data generally stale when collected for processing
- Result
  - Scaling up costs lots of money



Alternate Solution

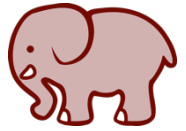
- 'Scale out' instead of 'Scale up'
- Horizontally instead of Vertically





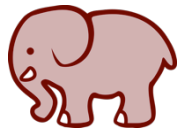
Node A

Node B



Issues with disk seek in traditional horizontal scaling

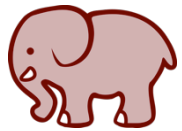
- Disk seek
  - 100 Billion records x 10 bytes = 1 TB
  - To update 10% of records would mean updating 10 billion records
- Traditional approach in classical systems:
  - Seek, read, write
  - Seek is slowest with average cost of 10ms (needs physical movement of head)
  - A seek for every record would take 2,700 hours
  - A scan read would take about 10,000 seconds (2.7 hours)
  - More seek – make the numbers go even higher



Classical systems - Assumptions

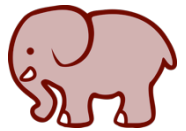
- Hardware is reliable – won't fail. System would fail if one piece of hardware fails.
- Machines have an identity
- Data sets must be centralized

NOT TRUE ANY MORE



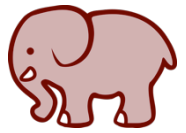
Lab 1

- Goal: Log in to remote terminal
  - ssh to your remote machine
  - go to your home directory
  - verify the contents of your home directory
  - login and password will be provided by instructor
- All labs will be performed on your remote machine
- Use an editor like `vi` or `emacs` to create or edit files in all future labs



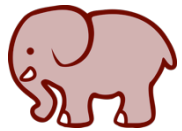
Apache Hadoop – A Developer's Course

Hadoop Basics – HDFS and MapReduce



New Paradigm

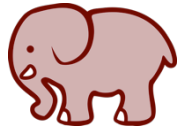
- Process data locally
- Reduce dependence on bandwidth
- Expect failure
- Handle failover elegantly
- Duplicate finite blocks of data to small groups of nodes instead of entire database
- Reduce elapsed seek time



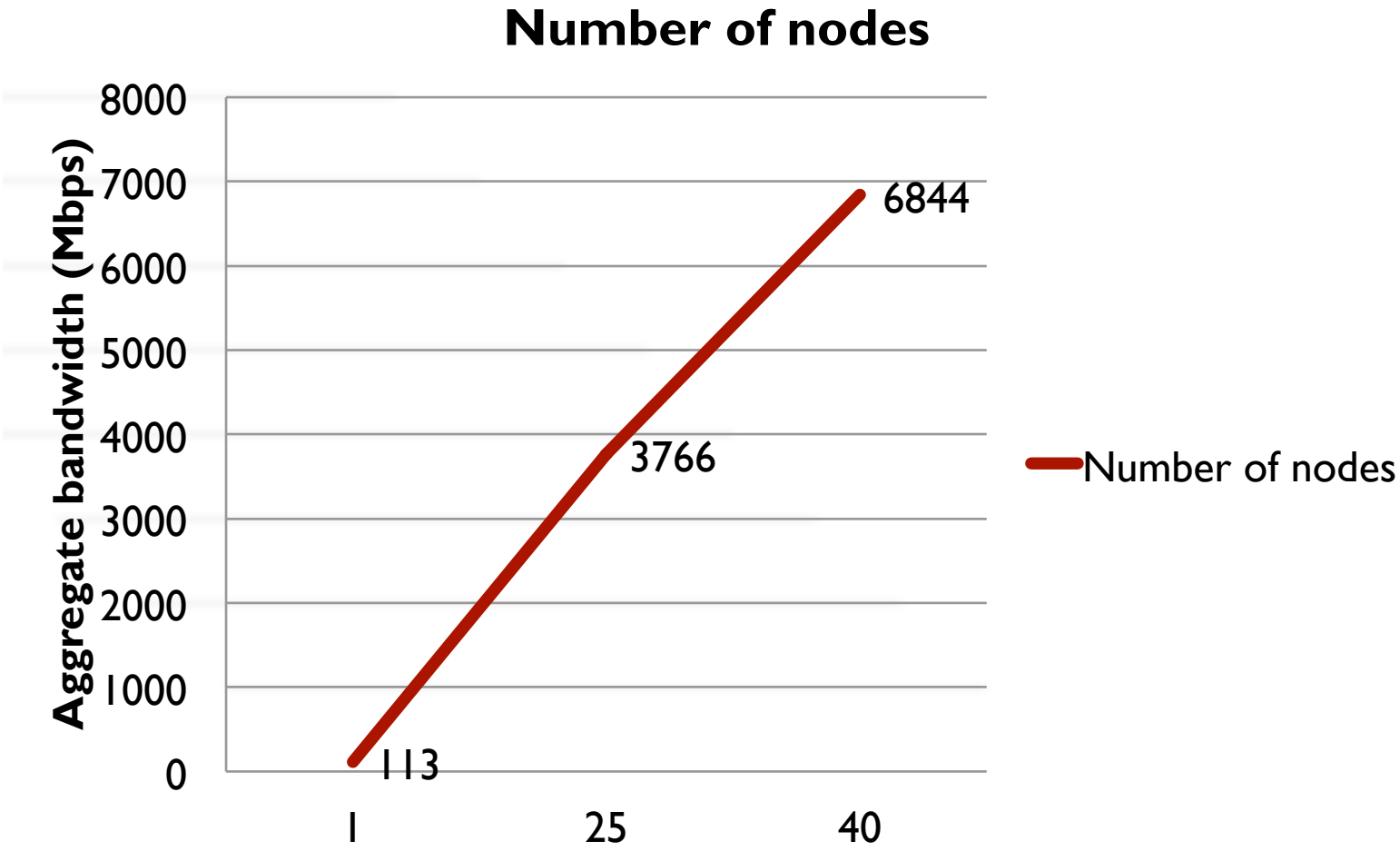
The Hadoop Approach

- HDFS – is Hadoop Distributed File System
  - HDFS provides storage
- MapReduce – a distributed analysis method
  - MapReduce provides analysis power

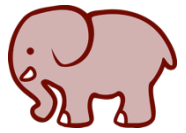
- Distribute large amounts of data across thousands of commodity hardware nodes
  - Process data in parallel
  - Replicate data across cluster for reliability
- Analysis moved to data
  - Avoids data copy
- Scanning of data
  - Avoids random seeks
  - Easiest way to process



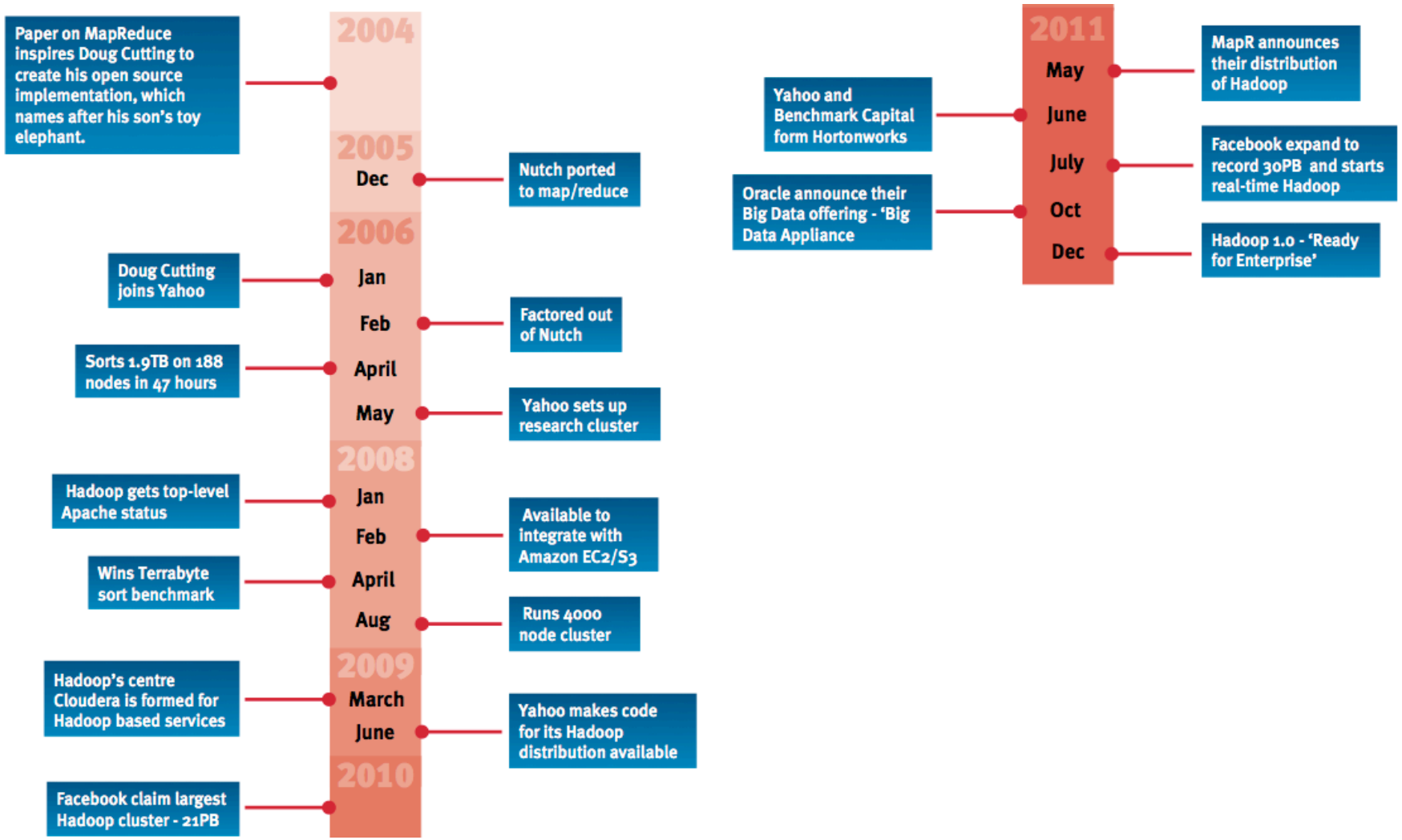
Hadoop Scalability



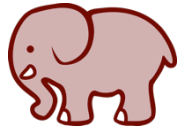
Rate: ~170 Mbps/node



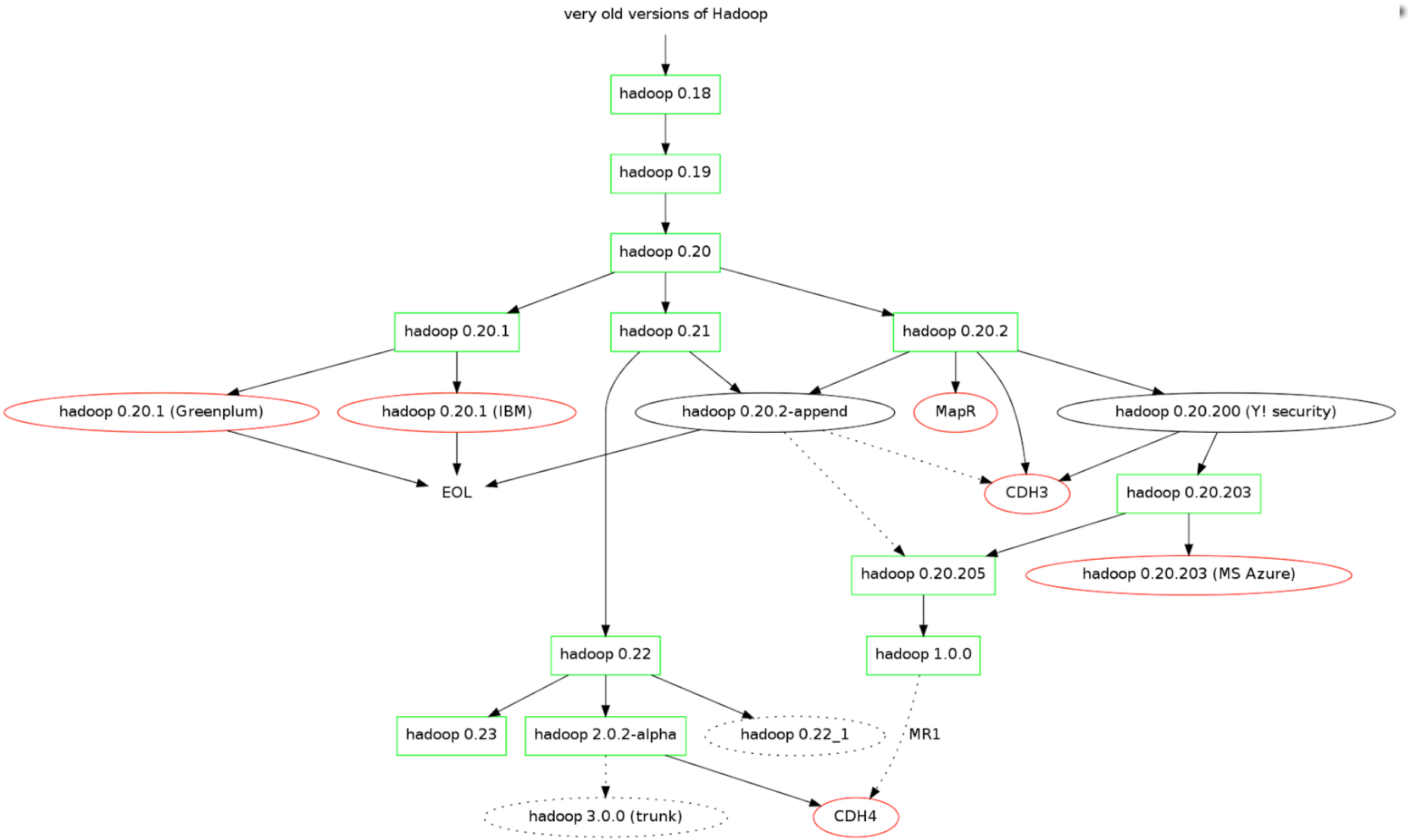
History of Hadoop



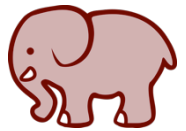
Source http://hadoopilluminated.com/hadoop_book/chapter-Hadoop_Versions.html



Hadoop versions and branches



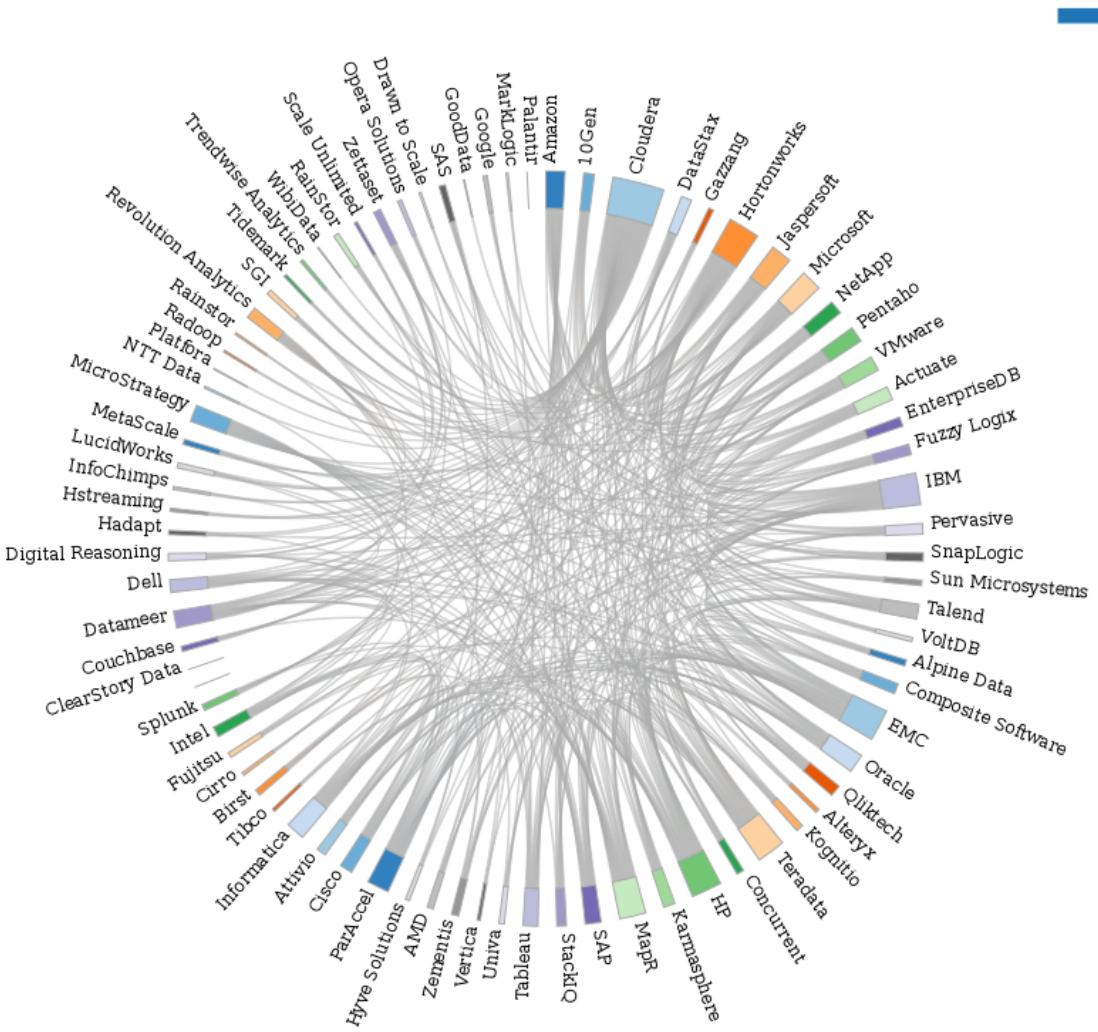
Genealogy of Elephants II: http://drcos.boudnik.org/2013/01/what-you-wanted-to-know-about-hadoop.html

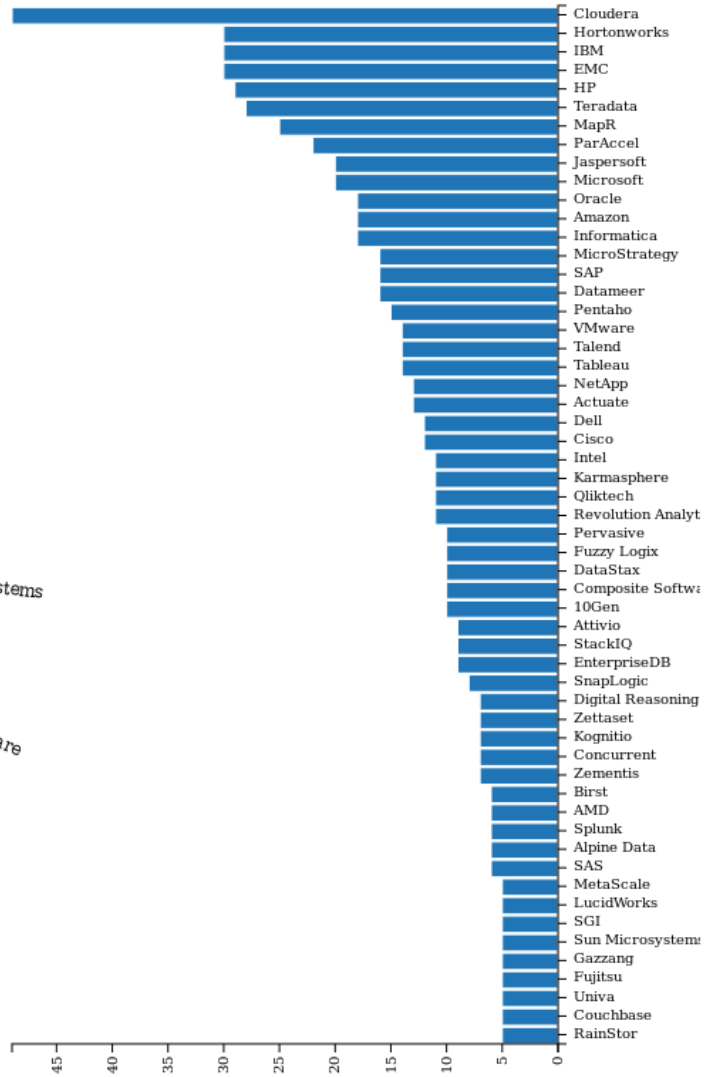


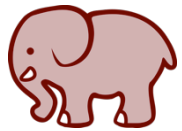
HADOOP ECOSYSTEM

Who has the most connections/partners?

data from January 2013

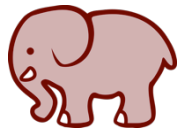






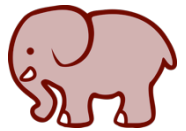
Apache Hadoop – A Developer's Course

Working with HDFS

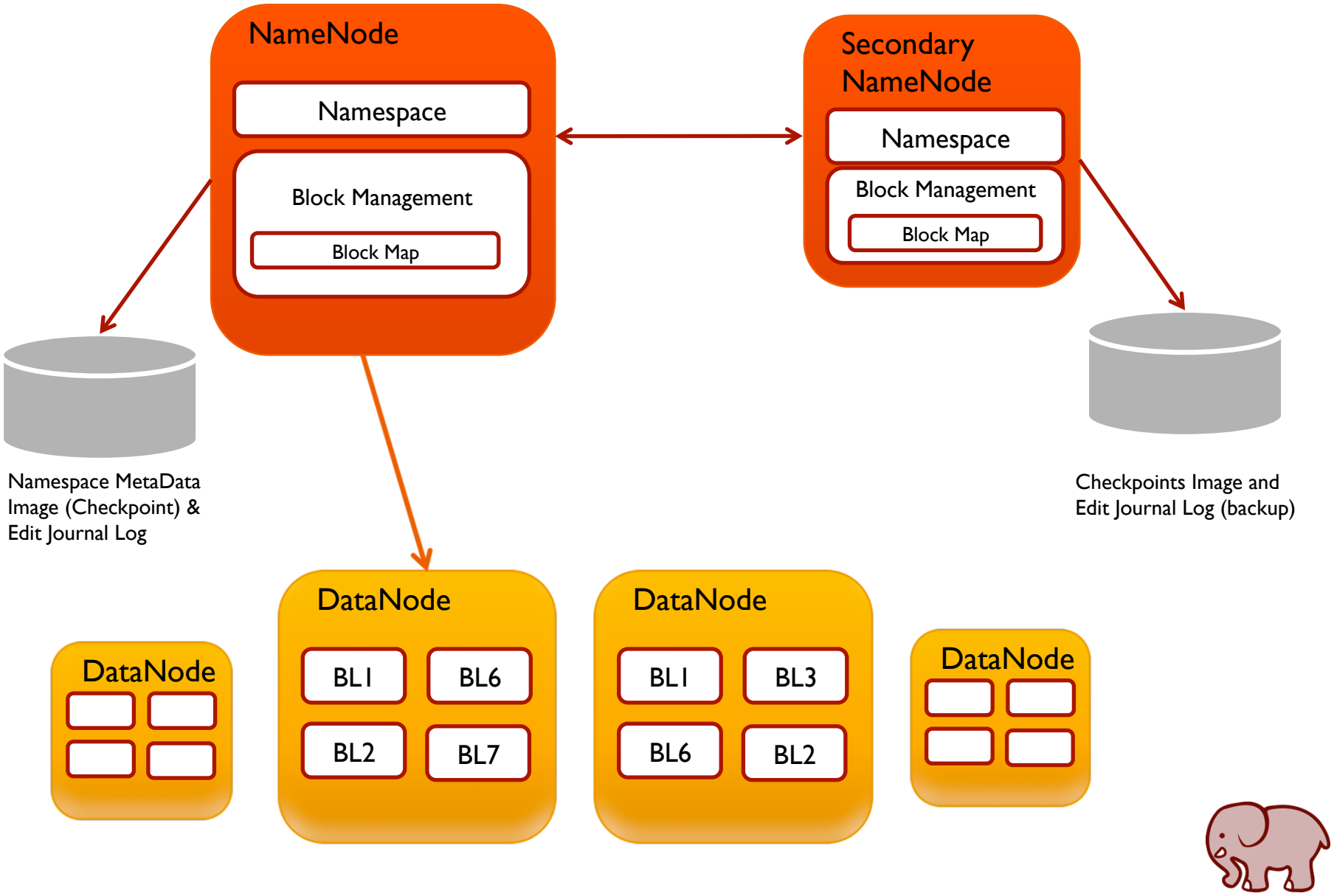


HDFS – An Introduction

- HDFS = Hadoop Distributed File System
- Primary storage for Hadoop
- Fast and reliable
- Deployed only on Linux so far (Windows in the works)
- Characteristics of HDFS
  - Persistent
  - Replicated
  - Linearly scalable
  - Applications sequentially stream reads – often from very large files
  - Optimized for read performance – avoids random disk seeks
  - Write once and read many times
  - Files only append
  - Data stored in blocks distributed over many nodes with block sizes from 128MB to 1GB

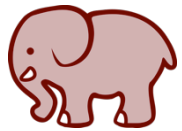


HDFS Architecture



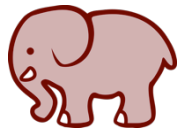
Data Organization

- Metadata
  - Organized into files and directories
  - Linux-like permissions prevent accidental deletions
- Files
  - Divided into uniform sized blocks
  - Default 64 MB
  - Distributed across clusters
- Rack-aware
- Keeps sizing checksums
  - for corruption detection



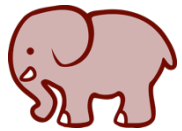
HDFS Cluster

- HDFS runs in Hadoop distributed mode
  - on a cluster
- 3 main components:
  - NameNode
    - Manages DataNodes
    - Keeps metadata for all nodes and blocks
  - DataNodes
    - Hold data blocks
    - Live on racks
  - Client
    - Talks directly to NameNode and then DataNodes



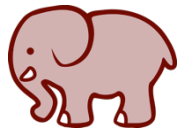
NameNode

- Server running the *Namenode* daemon
  - Responsible for coordinating datanodes
- The 'master' of the DataNodes
- Problems
  - Lots of overhead to being the Master
  - Should be special server for performance
  - Single point of failure



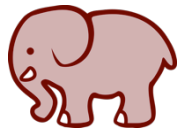
Secondary NameNode

- Not really an automatic secondary – named poorly!
- NO auto-failover
- Must be configured
- Merges and manages two files
  - Filesystem image (*fsimage*)
  - Edit log
- Sends resulting metadata back to NameNode
- Should run on a separate Node

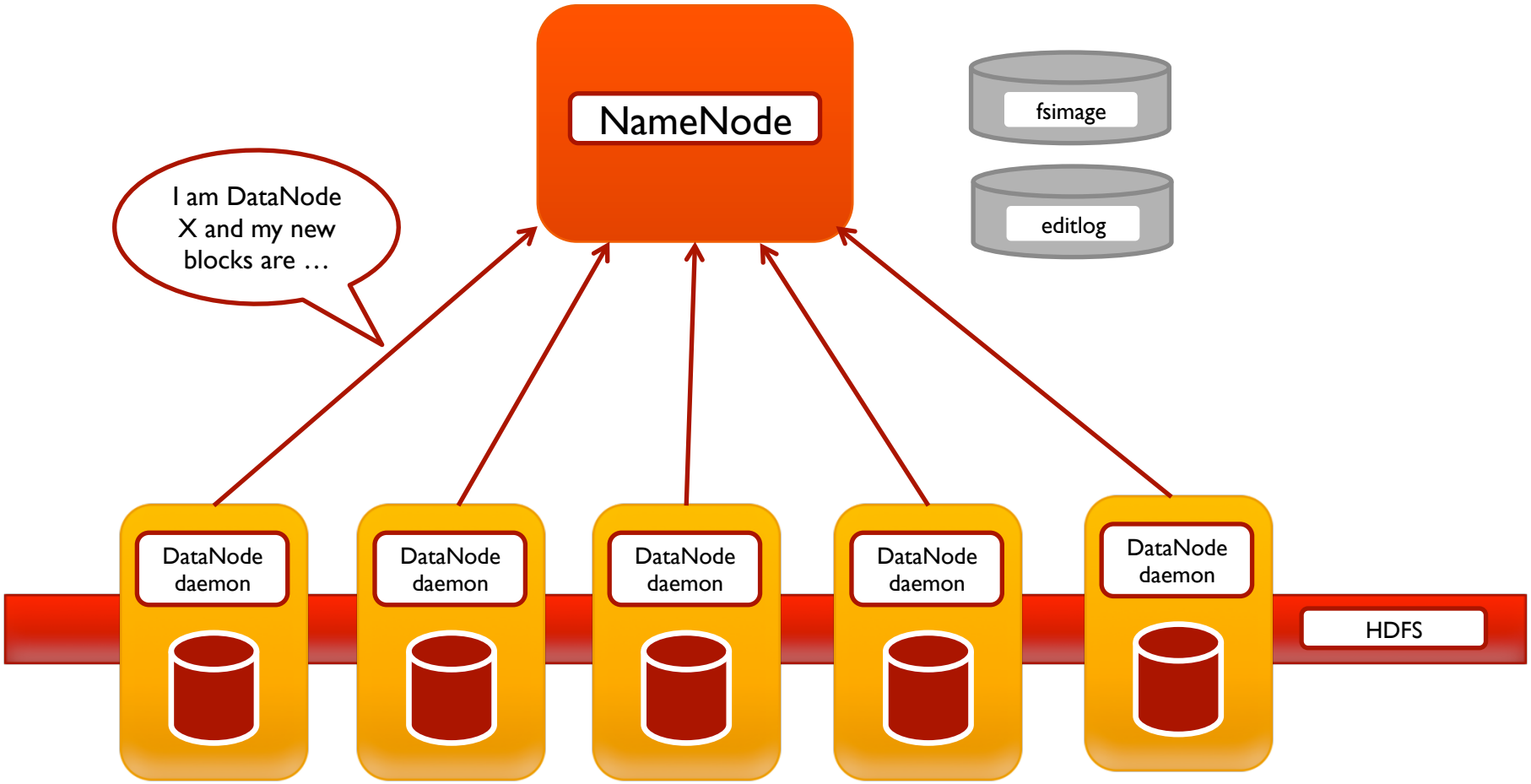


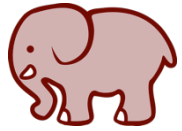
DataNode

- A common node running a *datanode* daemon
- Slave
- Manages block reads/writes for HDFS
- Manages block replication
- Pings NameNode and gets instructions back
- If heartbeat fails
  - NameNode removes from cluster
  - Replicated blocks take over

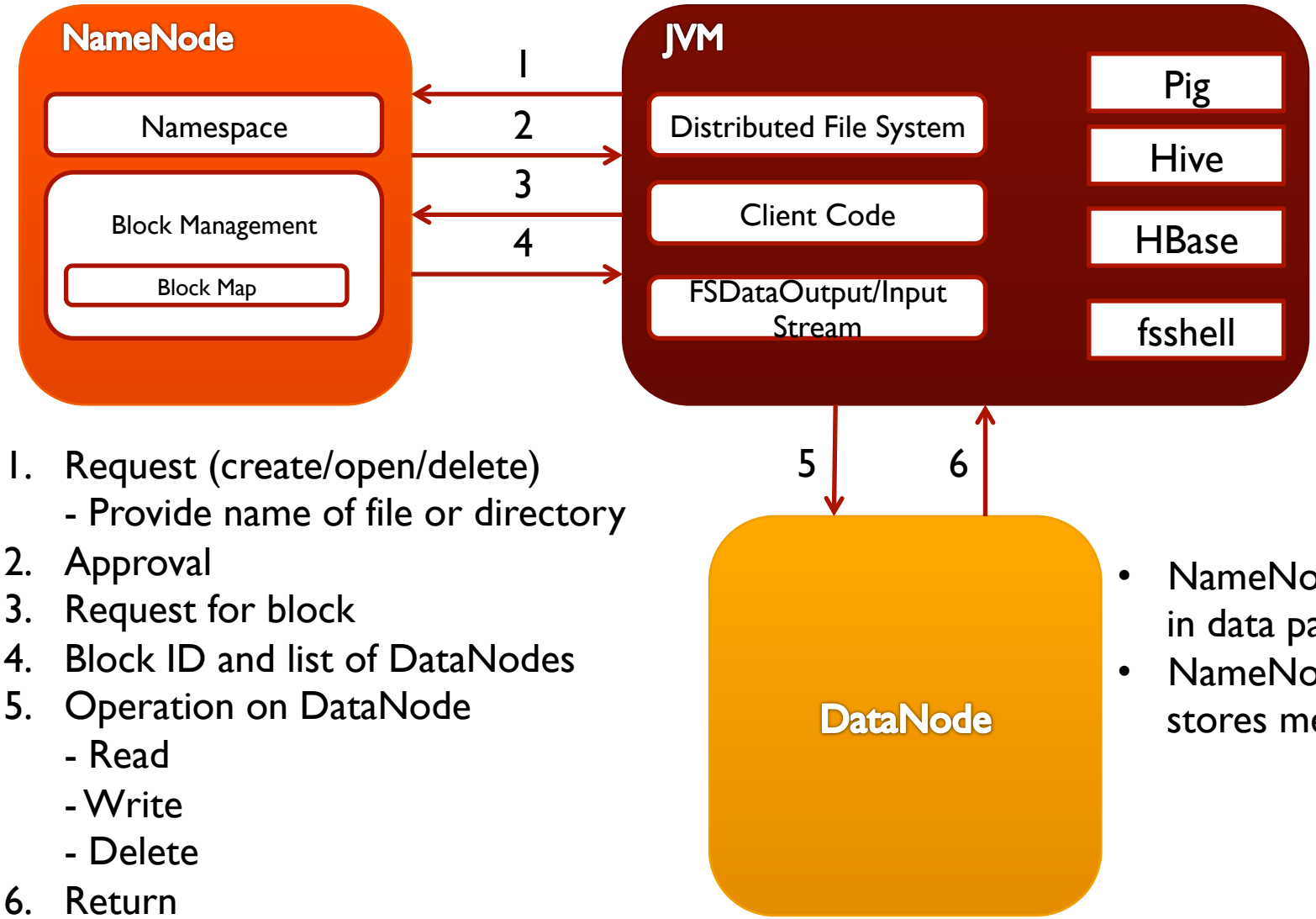


HDFS Heartbeats



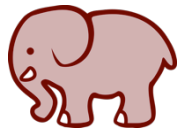


File Access - RPC



1. Request (create/open/delete)
   - Provide name of file or directory
2. Approval
3. Request for block
4. Block ID and list of DataNodes
5. Operation on DataNode
   - Read
   - Write
   - Delete
6. Return

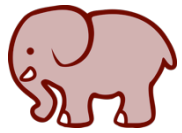
- NameNode is not in data path
- NameNode only stores metadata



HDFS Shell Commands

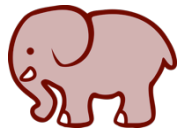
```
$ hadoop fs –command <args>
```

- HDFS command examples (around 30 commands)
  - -cat: display file contents (uncompressed)
  - -text: like cat, but works on compressed files
  - -chgrp, -chmod, -chown: just like the Unix command, changes permissions
  - -put, -get, -copyFromLocal, -copyToLocal: copies files from the local file system to HDFS and vice-versa.
  - -ls, -lsr: lists files / directories
  - -mv, -moveToLocal, -moveFromLocal: moves files
  - -stat: statistical info for any given file (block size, number of blocks, file type, etc.)



HDFS Command Examples

```
$ hadoop fs -ls /user/john
$ hadoop fs -lsr
$ hadoop fs -mkdir input
$ hadoop fs -put ~/training/commands.txt notes
$ hadoop fs -chmod 777 notes/command.txt
$ hadoop fs -cat notes/commands.txt | more
$ hadoop fs -rm notes/*.txt
```



Uploading and Retrieving files on HDFS

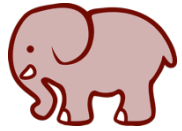
- Uploading

```
$ hadoop fs –put filenameSource filenameDestination
$ hadoop fs –put filename directoryName/fileName
```

```
$ hadoop fs –put foo bar
$ hadoop fs –put foo directoryName/fileName
$ hadoop fs –lsr directoryName
```

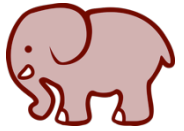
- Retrieving

```
$ hadoop fs –cat foo
$ hadoop fs –get foo LocalFoo
$ hadoop fs –copyToLocal foo LocalFoo
$ hadoop fs –rmr directory/file
```



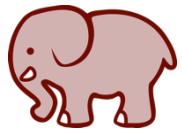
Lab 2: Get familiar with HDFS commands

- Practice basic HDFS file-system commands



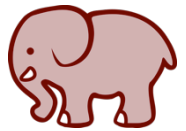
HDFS APIs

- Java APIs exists for low-level access to HDFS
  - Abstracts file-system API
  - Easy access and storage on HDFS
  - API hides low-level details about file-system
- Package containing APIs `org.apache.hadoop.fs`
  - Five interfaces provided
  - Over 20 classes, enums, exceptions provided, e.g.
    - FileSystem
    - Path
    - BlockLocation
    - FileChecksum



Core types found in .fs Package

- Path
  - Wraps a directory or file
  - Uses "/" as a path separator
- FileSystem
  - Represents either distributed HDFS of Local Linux OS file system (determined by factory method)
- FSDataInputStream and FSDataOutputStream
  - Handles File I/O
- FileStatus
  - Provides directory or file metadata
  - e.g. group, owner, directory flag etc.

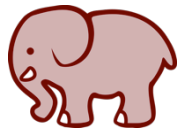


Path – allows path handling

- Path – represents and HDFS file or directory
- Used by FileSystem object at runtime
- Path has 7 constructors and over a dozen methods

```
FileSystem hadoop = FileSystem.get(new Configuration());
Path myPath = new Path(hadoop.getWorkingDirectory(), “/logdata”);
hadoop.mkdirs(myPath);  // creates with default permissions
boolean wasDeleted = hadoop.deleteOnExit(myPath);
```

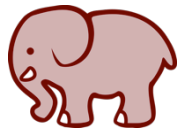
Example of how to create an HDFS directory



FileSystem – a class to represent file system

- FileSystem – represents the HDFS or local file system
- No public constructor
- Dozens of methods available
- Use public static factory methods to get an instance
  - LocalFileSystem getLocal(Configuration conf)
  - FileSystem get(Configuration conf)
  - FileSystem get(URI uri, Configuration conf)
  - FileSystem get(URI uri, Configuration conf, String user)

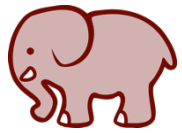
```
FileSystem hdfs = FileSystem.get(new Configuration());
```



FileStatus – a class for metadata

- FileStatus – represents metadata for HDFS or a directory
  - public Path getPath()
    - Once Path is known, the file can be read
  - Retrieve other data like length, modification time, permissions, etc.
- Returned by FileSystem methods

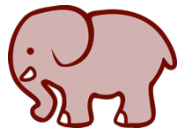
```
public FileStatus[] getFilesByDir(Path hdfsDir, FileSystem hdfs) {
        FileStatus[] allFiles = hdfs.listStatus(hdfsDir);
        return allFiles;
}
```



Using HDFS API – example to copy a file

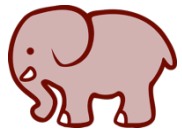
```
public void showHDFS(Path inPath, Path outPath) throws IOException {
   Configuration config = new Configuration();
   FileSystem hdfs = FileSystem.get(config); //call static factory method
   LocalFileSystem local = FileSystem.getLocal(config); // local Linux file
   FSDataInputStream inStream = local.open(inPath); // read data
   FSDataOutputStream outStream = hdfs.create(outPath); //open out stream
   byte[] bytesRead = new byte[1000]; // to hold file data in memory
   int data = 0; // counter to read input file
   while ((data = inStream.read(bytesRead)) > 0 {
      System.out.println("Reading input file: " + inPath); // report file name
      outStream.write(bytesRead); // write data to output stream
   }
   inStream.close(); // close the input stream
   outStream.close(); // close the output stream
}
```

Copies a file from local file system to HDFS



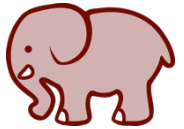
Hadoop Configuration files

- Three configuration files (Hadoop v0.2 and beyond)
  - core-site.xml for core settings
    - e.g. hostname for NameNode deamon
  - hdfs-site.xml for HDFS issues
    - e.g. how a data block replicates to DataNodes
  - mapred-site.xml for MapReduce issues
    - e.g. hostname for the JobTracker daemon
- Environment configuration stored in hadoop-env.sh
  - e.g. JAVA_HOME



Lab 3: Program HDFS

- Goal: Use Java to accomplish HDFS tasks
- Procedure:
  - Read a file from local Linux filesystem
  - Write a file to the HDFS



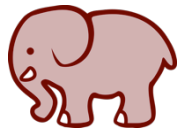
Monitoring HDFS

Web Interface

- NameNode can be accessed from port 50070
  - http://namenode:50070
  - Provides basic monitoring and file browsing operations
- DataNode can be accessed by using port 50075
  - http://datanode:50075

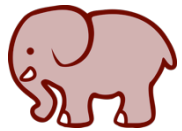
HDFS Log Files

- Hadoop in distributed modes uses log files
- The following daemons create logs
  - namenode
  - secondarynamenode
  - datanode
  - jobtracker
  - tasktracker



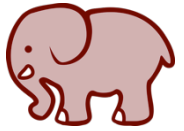
HDFS File Permissions

- Simple permission system based on POSIX model
- Does not provide strong security for HDFS
- Designed to prevent accidental corruption of data or casual misuse of information
- Users and Groups defined
- Each file or directory has 3 permissions
  - Read, write and execute
- Superuser supersedes all permission settings



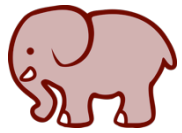
Lab 4: Monitor HDFS

- Goal: Practice using basic monitoring tasks in HDFS and explore various configurations and variables
- Procedure:
  - Explore NameNode and DataNode configuration
  - Explore JobTracker and TaskTracker configuration
  - Use the HDP web-based GUI to browse web user interfaces
  - Examine Hadoop primary daemons and environmental variables
  - Start and stop Hadoop services



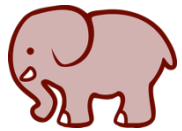
Apache Hadoop – A Developer's Course

MapReduce – A Primer

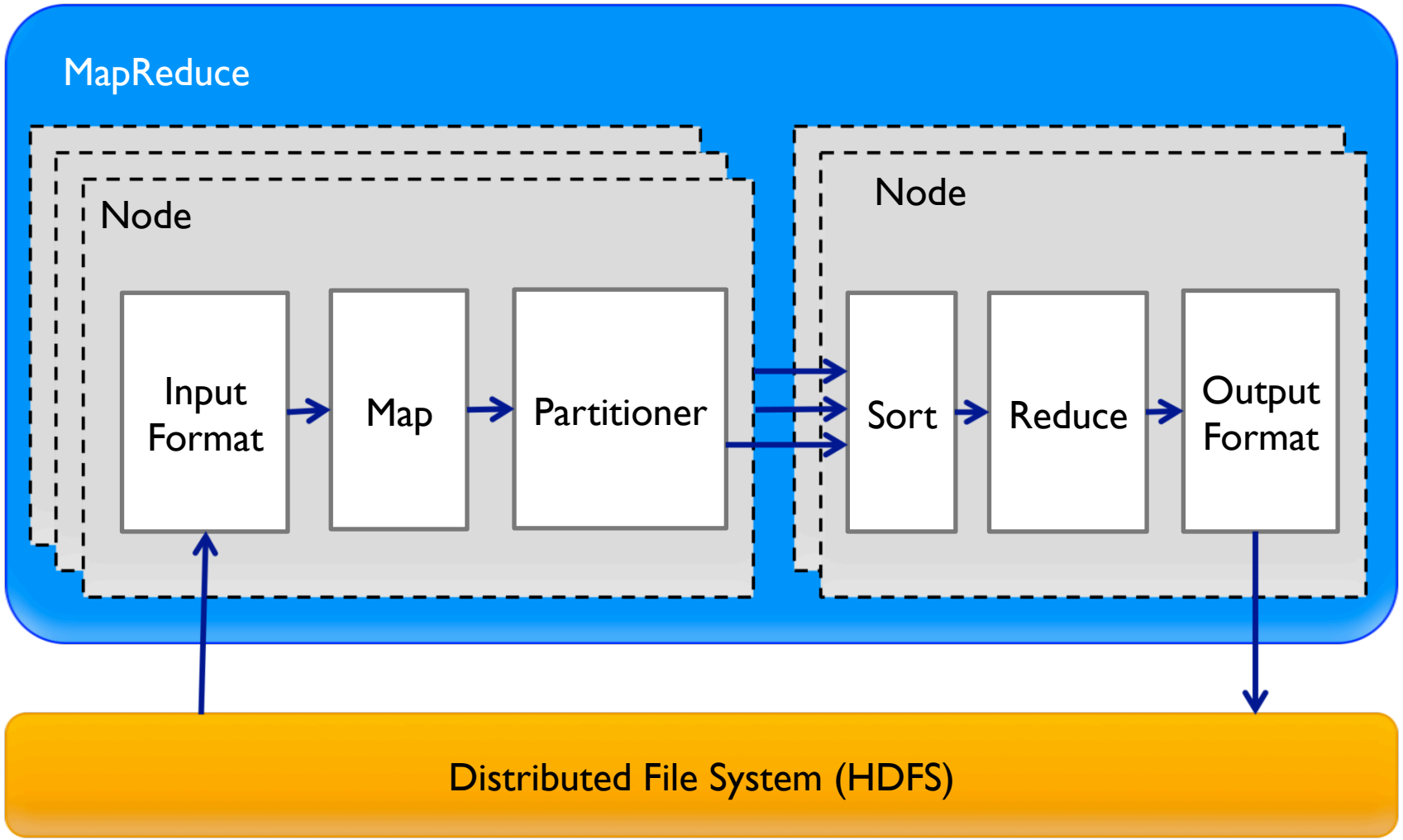


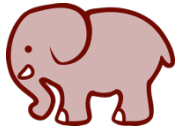
What is MapReduce?

- A framework
- Big Data analytics and processing
  - Large datasets
- Node-local computation
- Parallel processes
- Handles node fail-over
- Java
  - Other languages supported

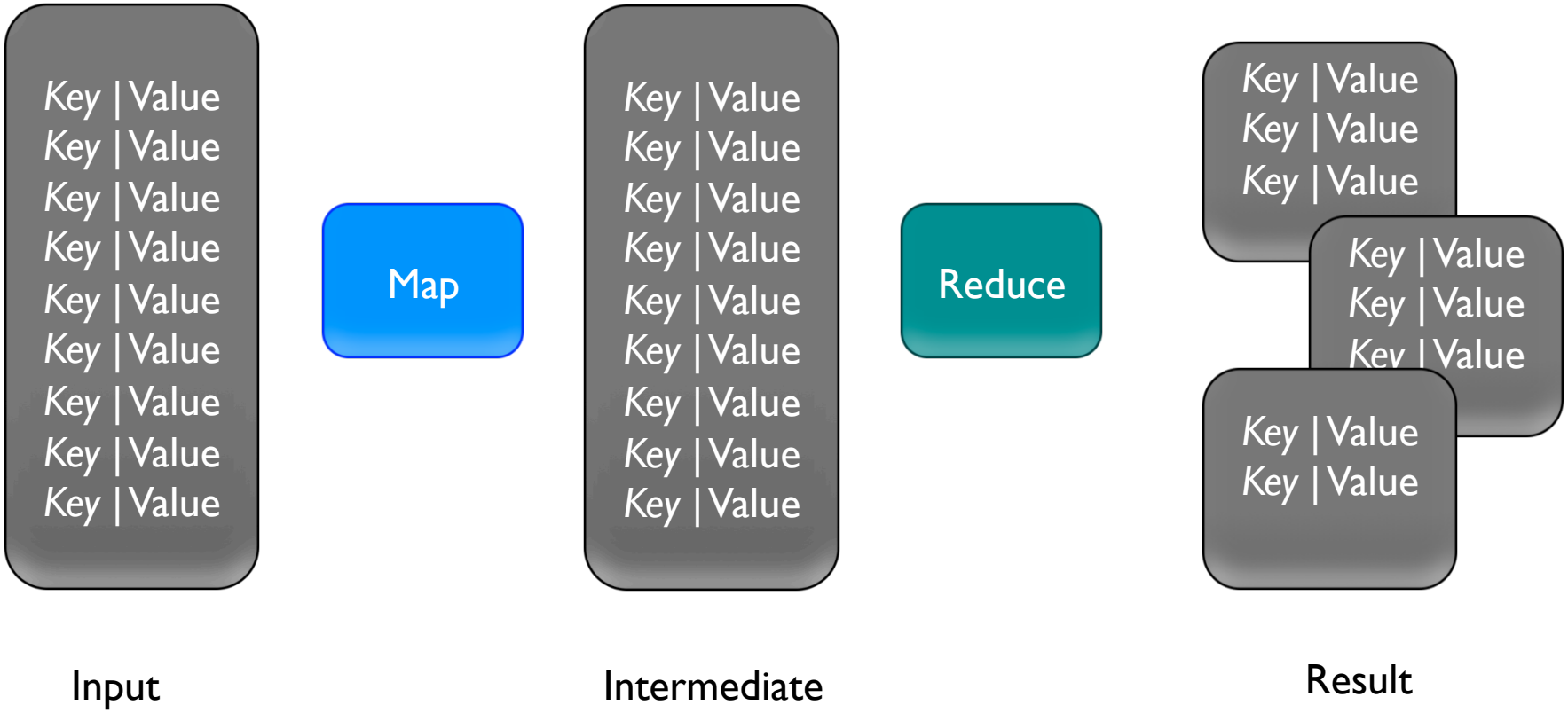


MapReduce Architecture

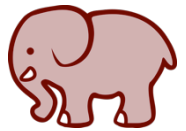




Simple MapReduce

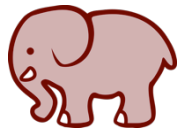


Represents some type of collection



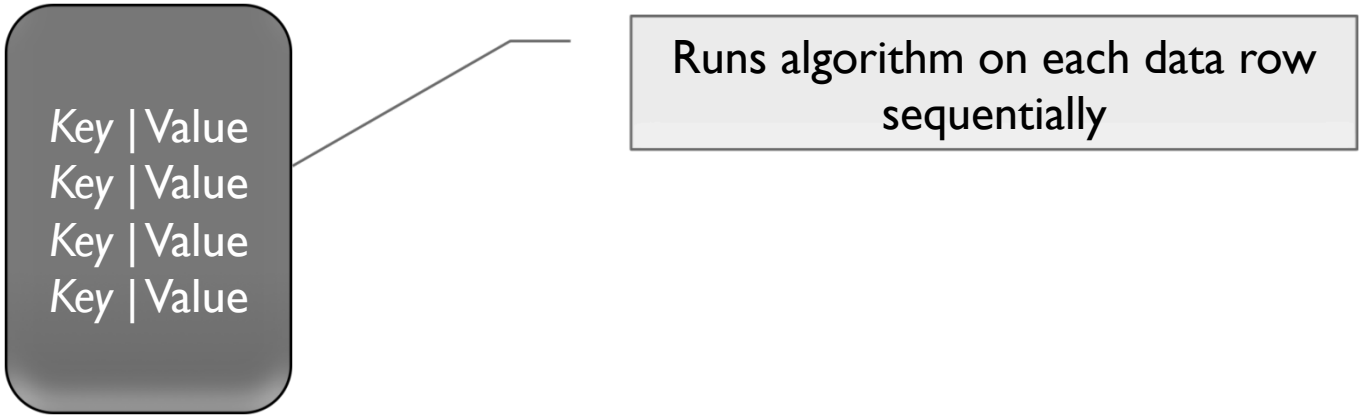
Splitting data

- InputFormat determines how the data is split up
- Creates InputSplit[] arrays
  - Each is individual map
  - Associated with a list of destination nodes
- RecordReader
  - Makes key, value pairs
  - Converts data types

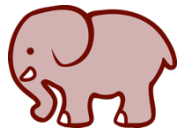


Mapping data

- Mapping operates on every data set individually
- Often used for extraction, transformation and loading (ETL) operations



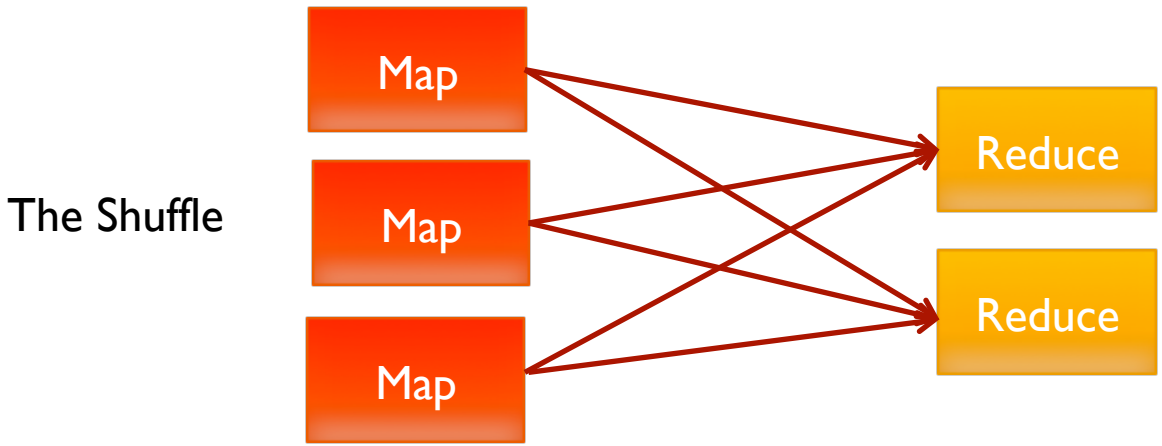

```
map(K1 key,
    V1 value,
    OutputCollector<K2, V2> output,
    Reporter reporter)
```

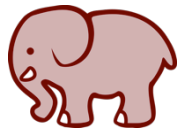


Partitioner

- Partitioner distributes [*key*, value] pairs
- Decides the target Reducer
  - Uses the key to determine
  - Uses Hash function by default
  - Can be overridden with custom function

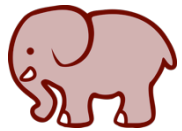
```
getPartition(K2 key,
             V2 value,
             int numPartitions);
```





Sort operation

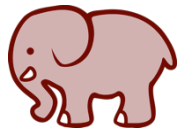
- Sort guarantees sorted inputs to Reducers
- Final step of Shuffle
- Helps to merge Reducer inputs



Reduce

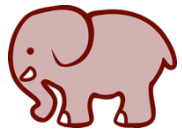
- Receives output from multiple Mappers
- Consolidates values for common intermediate keys
- Groups values by *key*

```
reduce(K2 key,
       Iterator<V2> values,
       OutputCollector<K3, V3> output,
       Reporter reporter);
```



OutputFormat

- OutputFormat is a validator
  - for output specifications
- Sets up a RecordWriter
  - Which writes out to HDFS
  - Organizes output into part-r-0000x files



Configuring Hadoop using Properties

- Hadoop has hundreds of properties
  - Some are related to HDFS

```
<property>
 <name>dfs.blocksize</name>
 <value>67108864</value>
 <description>
    The default block size for new files, in bytes.
 </description>
</property>
```

  - Some are related to MapReduce

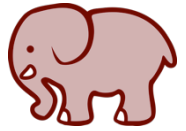
```
<property>
 <name>mapreduce.output.fileoutputformat.compress</name>
 <value>false</value>
 <description>Should the job outputs be compressed?
 </description>
</property>
```

  - Some are related to system as a whole

```
<property>
 <name>io.file.buffer.size</name>
 <value>4096</value>
 <description>The size of buffer for use in sequence files.
 The size of this buffer should probably be a multiple of hardware
 page size (4096 on Intel x86), and it determines how much data is
 buffered during read and write operations.</description>
</property>
```

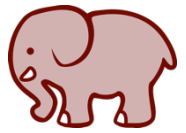
  - Source code is best source to find properties
  - One can create own properties too

See more at http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/hdfs-default.xml



Apache Hadoop – A Developer's Course

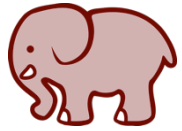
Programming MapReduce



WordCount – A Classic Example

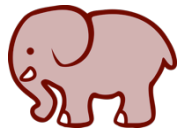
- Input file kept in folder /user/yourname/input
- Output is generated in folder /user/yourname/output

```
public class WordCount {

   /**
    * The mapper extends from the org.apache.hadoop.mapreduce.Mapper interface. When Hadoop runs,
    * it receives each new line in the input files as an input to the mapper. The "map" function
    * tokenize the line, and for each token (word) emits (word,1) as the output.
    */
   public static class TokenizerMapper
         extends Mapper<Object, Text, Text, IntWritable>{
   }

   /**
    * Reduce function receives all the values that has the same key as the input, and it output the key
    * and the number of occurrences of the key as the output.
    */
   public static class IntSumReducer
         extends Reducer<Text,IntWritable,Text,IntWritable> {
   }

   /**
    * As input this program takes any text file. Create a folder called input in HDFS (or in local directory if
you are running this locally)
    * @param args
    * @throws Exception
    */
   public static void main(String[] args) throws Exception {
   }
}
```



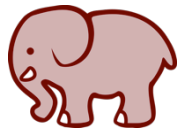
map() method of a basic MapReduce job

```
/**
   * The mapper extends from the org.apache.hadoop.mapreduce.Mapper
   * interface. When Hadoop runs, it receives each new line in the input files
   * as an input to the mapper. The "map" function tokenize the line, and for
    * each token (word) emits (word,1) as the output.
   */
  public static class TokenizerMapper
        extends Mapper<Object, Text, Text, IntWritable>{

     private final static IntWritable one = new IntWritable(1);
     private Text word = new Text();

     public void map(Object key, Text value, Context context) throws
              IOException, InterruptedException {
        StringTokenizer itr = new StringTokenizer(value.toString());
        while (itr.hasMoreTokens()) {
           word.set(itr.nextToken());
           context.write(word, one);
        }
     }
  }
```



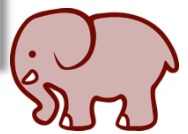
reduce() method of a basic MapReduce job

```
/**
   * Reduce function receives all the values that has the same key
   * as the input, and it output the keyand the number of occurrences
   * of the key as the output.
   */
  public static class IntSumReducer
       extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
                       Context context
    ) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }
```



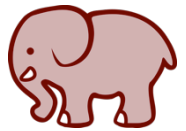
main() for a basic MapReduce job

```
/**
    * As input this program takes any text file. Create a folder called input in
    * HDFS (or in local directory if you are running this locally)
    * @param args
    * @throws Exception
    */
   public static void main(String[] args) throws Exception {
      JobConf conf = new JobConf();
      String[] otherArgs = new GenericOptionsParser(conf,
                  args).getRemainingArgs();
      if (otherArgs.length != 2) {
         System.err.println("Usage: wordcount <in> <out>");
         System.exit(2);
      }

      Job job = new Job(conf, "word count");
      job.setJarByClass(WordCount.class);
      job.setMapperClass(TokenizerMapper.class);
      //job.setCombinerClass(IntSumReducer.class); //Uncomment for combiner
      job.setReducerClass(IntSumReducer.class);
      job.setOutputKeyClass(Text.class);
      job.setOutputValueClass(IntWritable.class);
      FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
      FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
      System.exit(job.waitForCompletion(true) ? 0 : 1);
   }
```

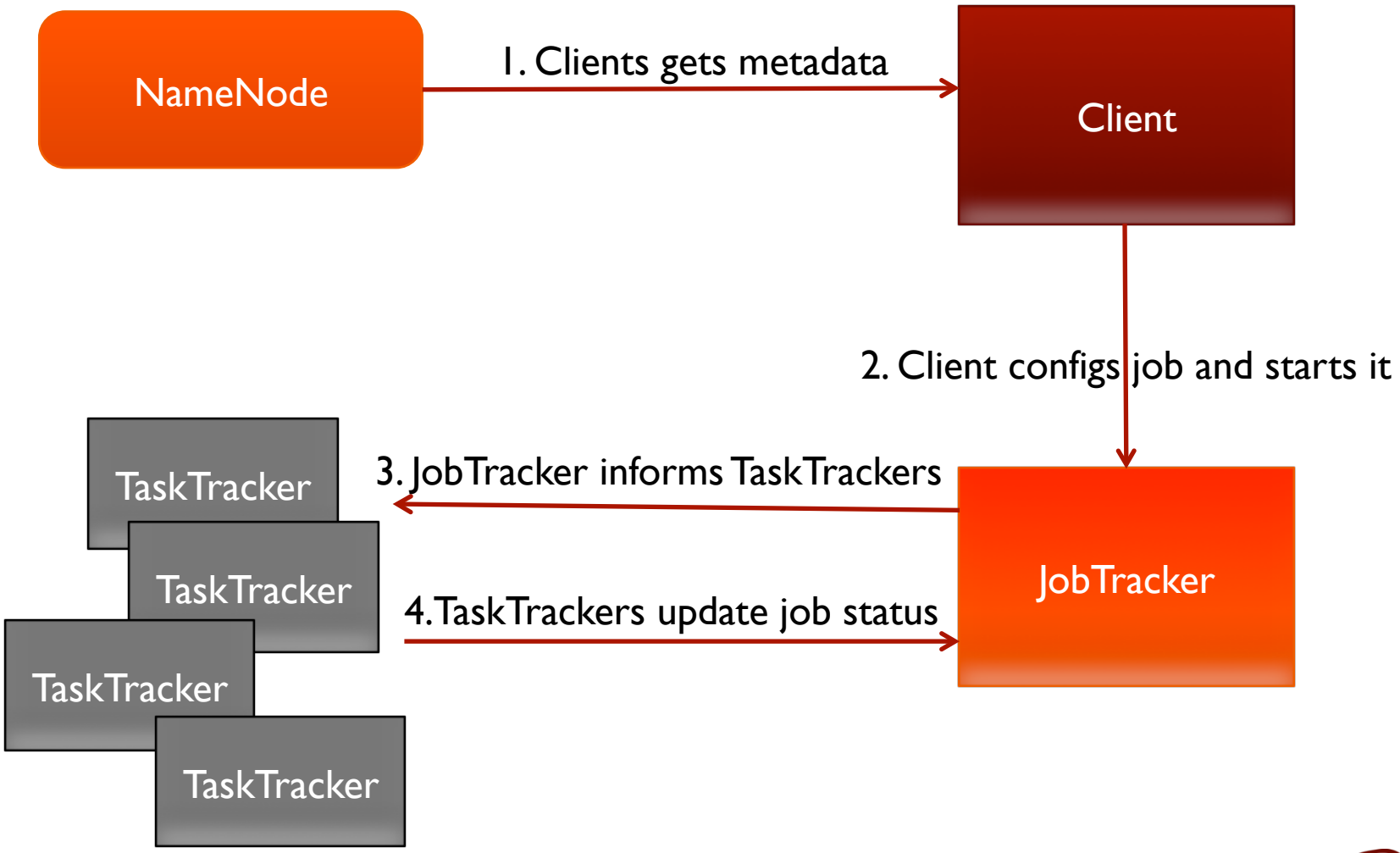


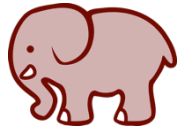
Lab 5: Run an existing MapReduce job

- Goal: Run a simple MapReduce job
- Procedure:
  - Use Hadoop built-in map() and Hadoop built-in reduce() code
  - Review the code for Job setup and configuration
    - Configuration() class, JobConf() class, and JobClient() class
  - Run the job



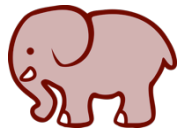
MapReduce life-cycle





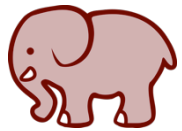
JobTracker

- A "master" daemon of MapReduce
  - along with NameNodes
- Provides resource management
  - Manages TaskTrackers at each node
  - Schedules tasks to available resources
- Manages Job lifecycle
  - Job submission
  - Task execution
  - Task error recovery
  - Run job's tasks to completion

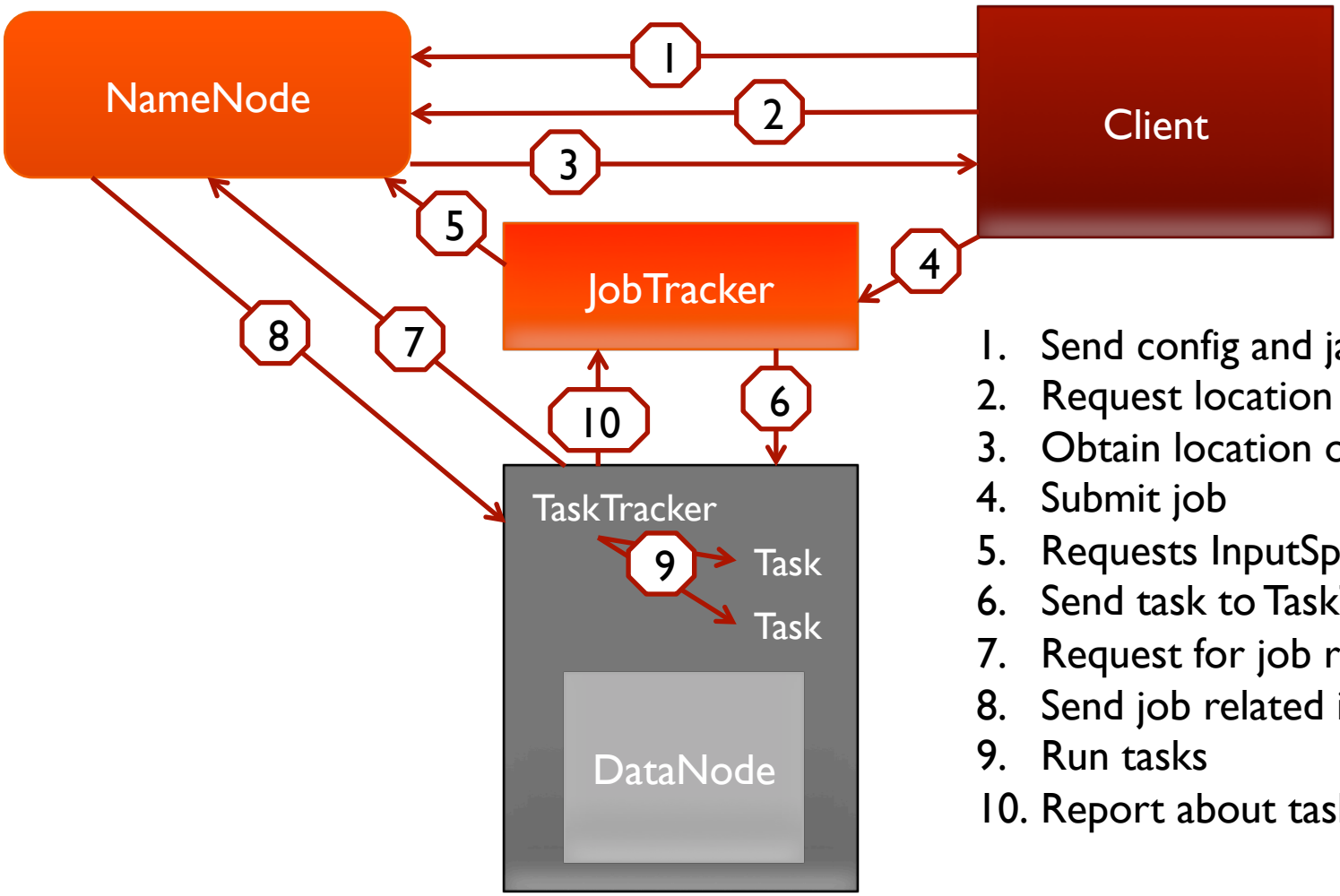


TaskTracker

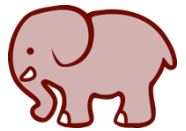
- Is a node-level daemon
- Manages tasks at the node
- Talks with `JobTracker`
  - Gets new tasks
  - Provides updates
- Typically fetches executables from HDFS
- Has *slots*
  - One task per slot



Job Submission and Running



1. Send config and jars
2. Request location of data
3. Obtain location of data
4. Submit job
5. Requests InputSplits
6. Send task to TaskTransfer
7. Request for job related info
8. Send job related info
9. Run tasks
10. Report about tasks



Hadoop Job Submission

```
$ hadoop fs myMapReduce.jar com.mycompany.hadoop.MyJob
   inputFile outputDirectory
```

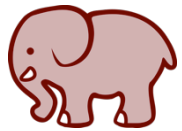
-Dproperty=value –conf path & file –fs some uri -files

Invocation options:

```
$ hadoop jar myMapreduce.jar MyJob arg1 arg2
```

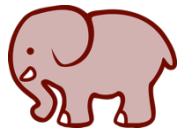
```
// config job in main
public static void main(String[] args)
```

```
// call run() from main()
public static void main(String[] args)
   ToolRunner.run(MyClass, args)
```



ToolRunner

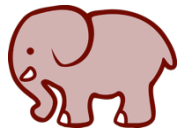
- Handles command-line arguments
- Job config is in `run()` method
- `main()` now calls ToolRunner
  - Invokes your `run()` method
  - Needs to know the class holding your run method, and the command line arguments
  - Passes parameters to `run()` via a `Configuration` object



Tool Interface

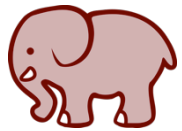
- When Tool is implemented you also get GenericOptionsParser
  - A class that interprets Hadoop command-line options
    - Sets them in a Configuration object
- run() method gets Configuration using getConf()

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

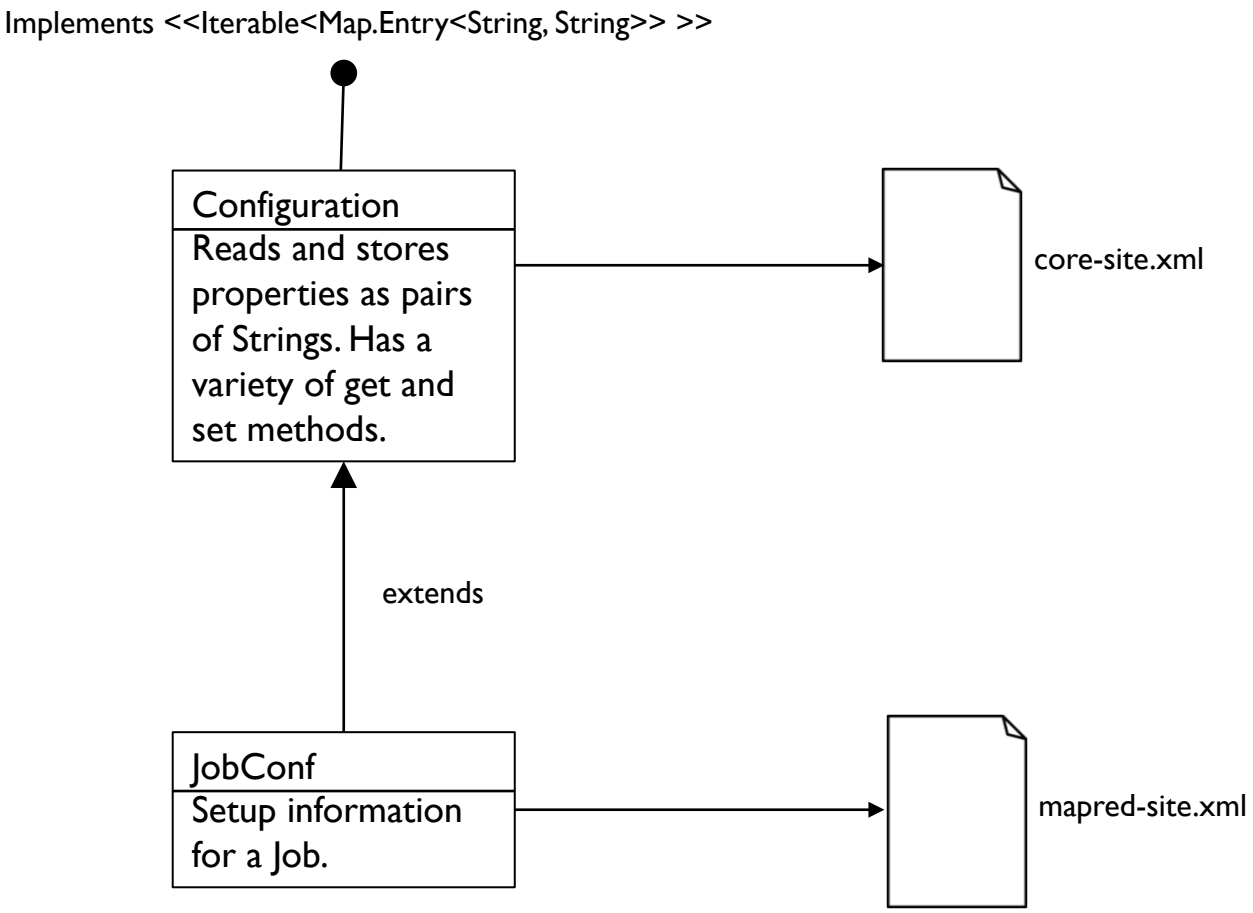


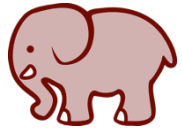
Job Configuration

- JobConf defines Hadoop job properties
  - Reads a Configuration() instance to get some of those properties
- Assigns a job name
- Specifies key/value data types for map and reduce
- Assigns map and reduce runtime classes
- Specifies HDFS input and output directories



JobConf() properties

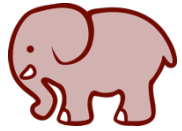




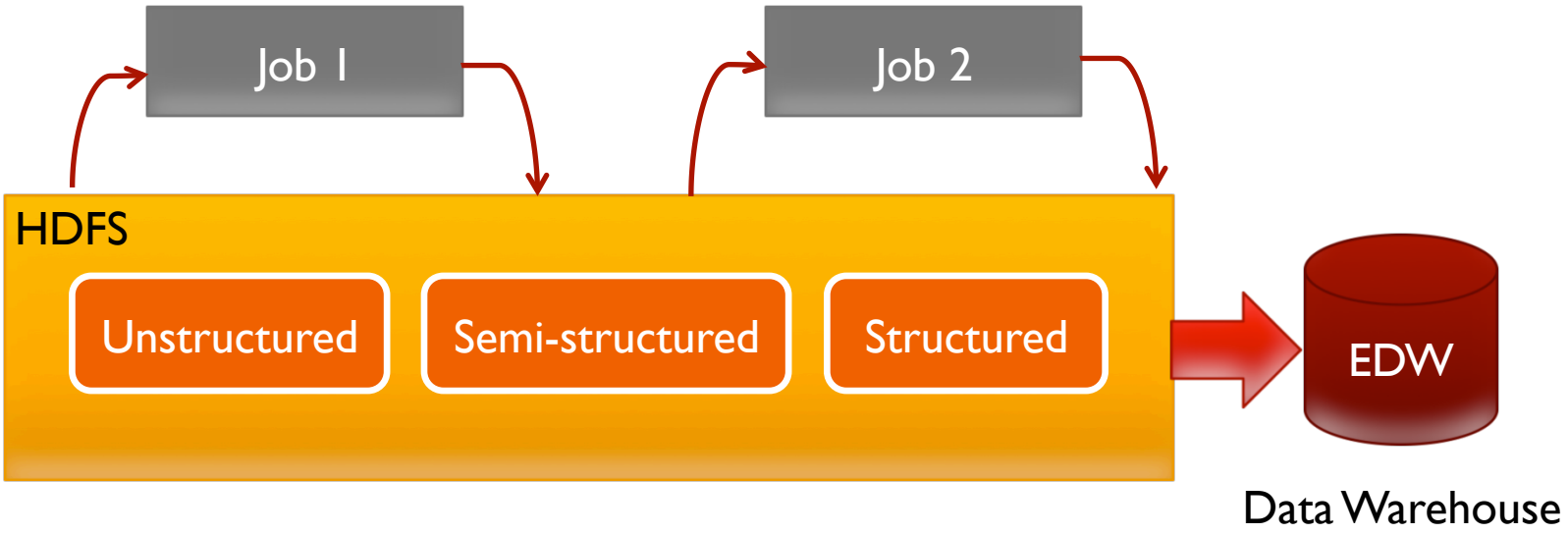
Some JobConf() Methods

```
job.setJobName("Log Processing Job");
job.setMapperClass(Mapper.class);
job.setReduceClass(Reducer.class);
job.setOutputKeyClass(Text.class); // to HDFS from reduce()
job.setOutputValueClass(Text.class);  // to HDFS from reduce()
job.setCombinerClass(Reducer.class);
job.setJarByClass(PageStat.class);
job.setMapOutputKeyClass(Text.class);  // output from map()
job.setMapOutputValueClass(IntWritable.class); // from map()
job.setNumReduceTasks(1);
job.set(String name, String value); // set arbitrary property
String jobName = job.get(String name);
```

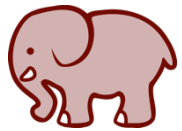
Full set may be found at http://hadoop.apache.org/docs/r1.2.0/api/org/apache/hadoop/mapred/JobConf.html



Chaining Jobs

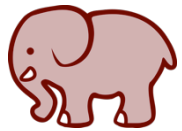


General mechanism to chain jobs for any problem solution



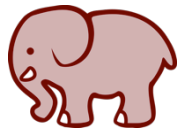
Fault Tolerance

- TaskTracker sends heartbeats to the JobTracker
- If the TaskTracker fails during Map process
  - Other TaskTrackers will be assigned
  - Re-execute all tasks
- If the TaskTracker fails during Reduce process
  - Other TaskTrackers will be assigned
  - Re-execute only incomplete tasks



Output Files

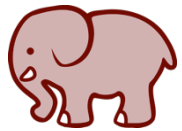
- Number of reducers is configurable
  - number of output files = number of reducers
  - One-to-one reducer to output file
  - One can specify the number of reducers in code `JobConf.setNumReduceTasks()`
- A map-only job
  - Specify number of reducers to be zero
  - number of output files = number of mappers = number of input blocks
  - Override `InputFormat.getSplits()` to control the number of map tasks



Killing a job

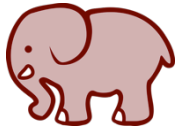
- Runaway jobs can be killed with a "job –kill" command

```
$ mapred job -list
0 jobs currently running
JobId   State   StartTime       UserName        Priority        SchedulingInfo
job_201305162210_0001   4       1368767710835   shrek  NORMAL  NA
$ mapred job -kill  job_201305162210_0001
```



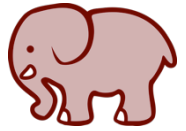
Lab 6: Write a simple MapReduce job

- Goal: Write a MapReduce program to count the occurrences of distinct words
- Procedure:
  - Create an input file and copy it to HDFS
  - Create a java file that adds the Java and Hadoop imports, the MapReduce class and sub-classes, map and reduce methods, and specify other aspects of the job
  - Compile and run the program



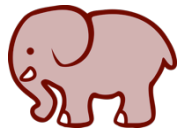
Apache Hadoop – A Developer's Course

Hadoop Data Types



Hadoop Built-in classes

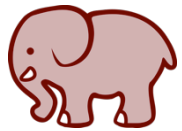
- Simple implementations of map() and reduce()
  - simple e.g. reverse K,V mapper
- map() "pass through"
  - copies K,V input as K,V output
  - still groups by key
  - sends the grouped data to a reducer
- reduce() "pass through"
  - pulls out each V from the Iterator and creates a KV pair
  - KV pairs are sent as output to HDFS



A Writable Datatype

```
public class MyWritable implements Writable {
      // Some data
      private int counter;
      private long timestamp;

      public void write(DataOutput out) throws IOException {
        out.writeInt(counter);
        out.writeLong(timestamp);
      }

      public void readFields(DataInput in) throws IOException {
        counter = in.readInt();
        timestamp = in.readLong();
      }

      public static MyWritable read(DataInput in) throws IOException {
        MyWritable w = new MyWritable();
        w.readFields(in);
        return w;
      }
    }
```

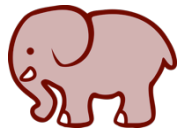
Example of a Writable class



Writable interface

- Methods
  - write(DataOutput out)
  - readFields(DataInput in)
- DataInput and DataOutput are streaming datatypes
  - methods that can be used on those streams
    - e.g. readInt(), writeInt(), readLong(), writeLong()
    - I/O on those streams is Hadoop-specific

API docs available at:
http://hadoop.apache.org/docs/r1.2.0/api/org/apache/hadoop/io/Writable.html
http://java.sun.com/javase/6/docs/api/java/io/DataInput.html
http://java.sun.com/javase/6/docs/api/java/io/DataOutput.html



WritableComparable / RawComparator

WritableComparable

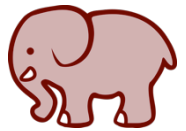
- Interface for objects that are both Writable and Comparable
  - Same methods as Writable
  - plus compareTo method
- Make your keys implement this interface

API docs: http://hadoop.apache.org/docs/r1.2.0/api/org/apache/hadoop/io/WritableComparable.html

RawComparator

- Compares keys by bytes during the sorting phase of MapReduce
- Avoids the overhead of keys being deserialized
- Improves performance greatly
- Write your own class that implements RawComparator or extends WritableComparator class

API docs: http://hadoop.apache.org/docs/r1.2.0/api/org/apache/hadoop/io/RawComparator.html



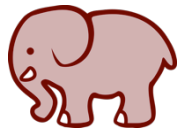
WritableComparator

- Utility class to simplify implementing RawComparator

```
public class MyComparator extends WritableComparator {
   protected MyComparator() {
      super(MyKey.class);
   }

   @Override
   public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2) {
      // Return 0 if b1 and b2 are equal
      // Return a negative value if b1 < b2
      // Return a positive value if b1 > b2
   }
}
```

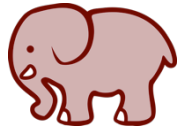
- b1 - The first byte array.
- s1 - The position index in b1. The object under comparison's starting index.
- l1 - The length of the object in b1.
- b2 - The second byte array.
- s2 - The position index in b2. The object under comparison's starting index.
- l2 - The length of the object under comparison in b2.

API docs: http://hadoop.apache.org/docs/r1.2.0/api/org/apache/hadoop/io/WritableComparator.html



WritableComparable Example

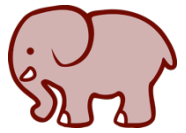
```
public class MyWritableComparable implements WritableComparable {
      // Some data
      private int counter;
      private long timestamp;

      public void write(DataOutput out) throws IOException {
        out.writeInt(counter);
        out.writeLong(timestamp);
      }

      public void readFields(DataInput in) throws IOException {
        counter = in.readInt();
        timestamp = in.readLong();
      }

      public int compareTo(MyWritableComparable w) {
        int thisValue = this.value;
        int thatValue = ((IntWritable)o).value;
        return (thisValue < thatValue ? -1 : (thisValue==thatValue ? 0 : 1));
      }
```



Basic Writable Datatypes

- Sub-classes generally support .get(), .set(), and .toString()
- Examples
  - ByteWritable
  - IntWritable
  - LongWritable
  - FloatWritable
  - DoubleWritable
  - BooleanWritable
  - NullWritable (with no .set() function)
  - IntPairWritable
  - Text

See full set in API docs: http://hadoop.apache.org/docs/r1.2.0/api/org/apache/hadoop/io/Writable.html

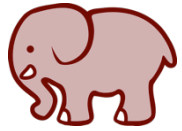


Writable Collections

- ArrayWritable for a Java Array
  - myArrayWritable.set(Writable[])
  - myArrayWritable.get() – returns a Writable object
  - myArrayWritable.toArray() – returns a Java object
- MapWritable for a Java HashMap
  - myMayWritable.put(K, V)
  - myMayWritable.get(K)
- BytesWritable for a Java array of bytes
  - myByteWritable.set(byte[], offset, len)
  - myByteWritable.get() - returns a byte array byte[]

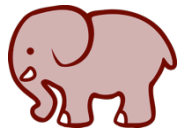
API docs:

http://hadoop.apache.org/docs/r1.2.0/api/org/apache/hadoop/io/ArrayWritable.html
http://hadoop.apache.org/docs/r1.2.0/api/org/apache/hadoop/io/MapWritable.html
http://hadoop.apache.org/docs/r1.2.0/api/org/apache/hadoop/io/BytesWritable.html



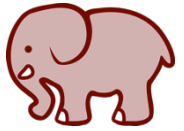
Other Serialization Options

- Apache Thrift (http://thrift.apache.org/)
- Avro (http://avro.apache.org/)
- Google Protocol Buffers (http://code.google.com/p/protobuf/)



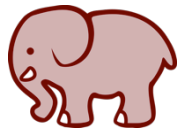
Lab 7: Create a custom Writable Type

- Goal: Create a new Hadoop datatype that can be passed as a value between your map() and reduce() code. It can be serialized to disk or transmitted across the network.



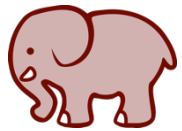
Apache Hadoop – A Developer's Course

InputFormat, OutputFormat, Combiner and Partitioner

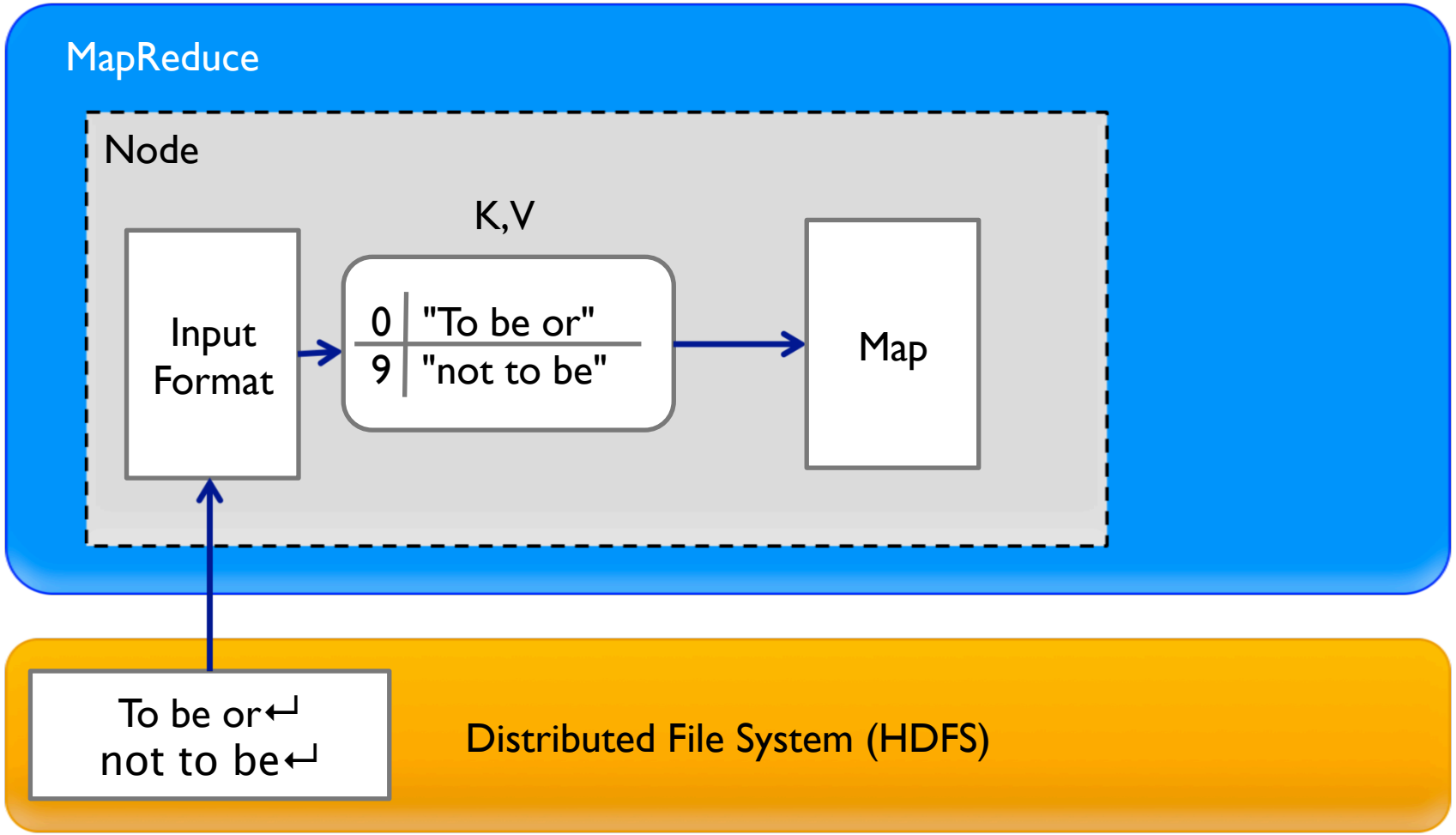


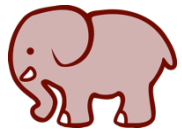
InputFormat

- Identifies input type
- Can be anything like
  - Text
  - Numbers
  - Complex types
  - DB rows
- Defines the way input files are chopped up
  - `getSplits()`
- Individual records come from splits
  - `getRecordReader()`
    - Parses input split into a set of key/value pairs

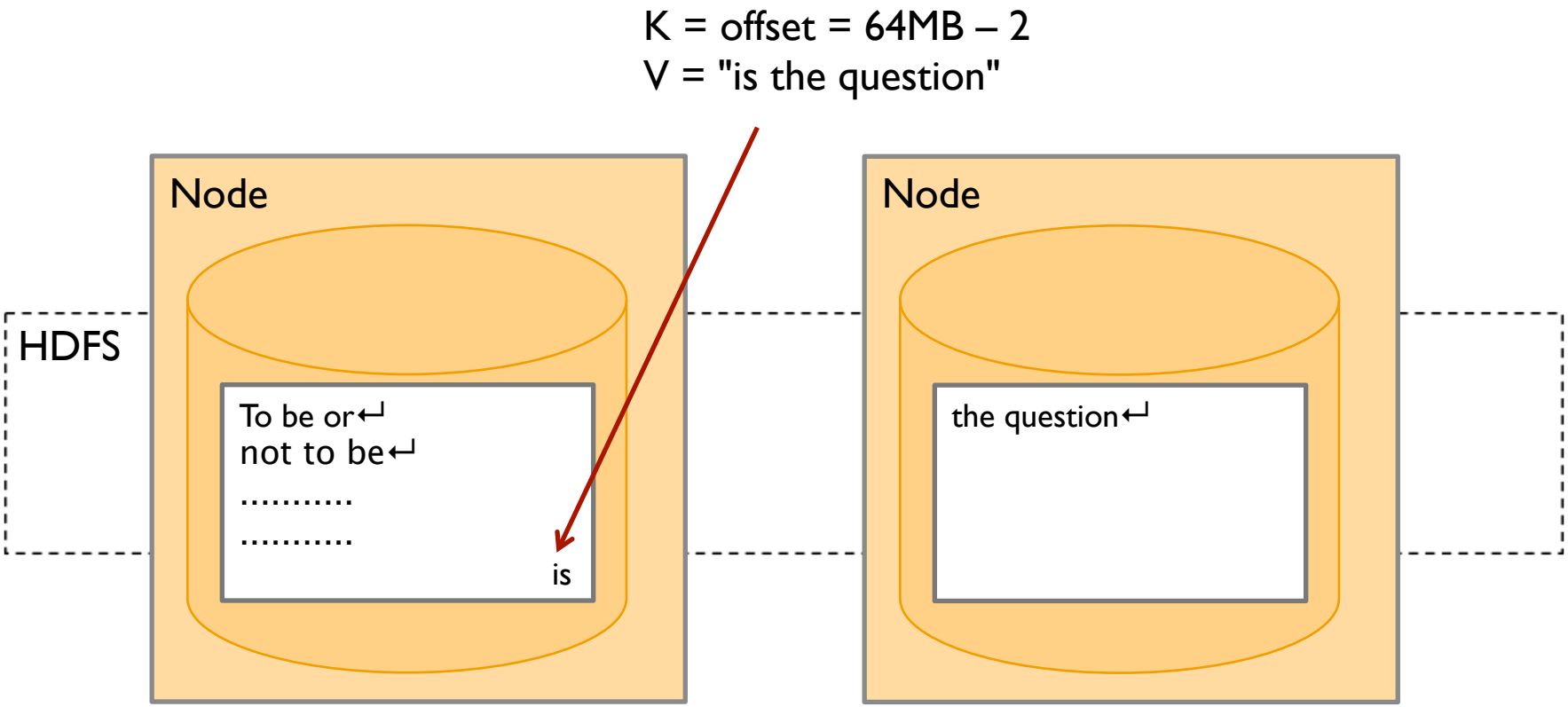


TextInputFormat

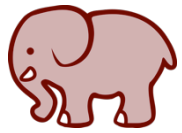




TextInputFormat

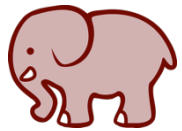


"Value" of a "key" can span over multiple blocks



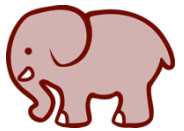
FileInputFormat

- FileInputFormat used for large files
  - Larger than HDFS block
- Split size
  - Controlled by Hadoop properties
  - Can be overridden by Application
- Hadoop is optimized by
  - decreasing the number of files and
  - making size of files larger

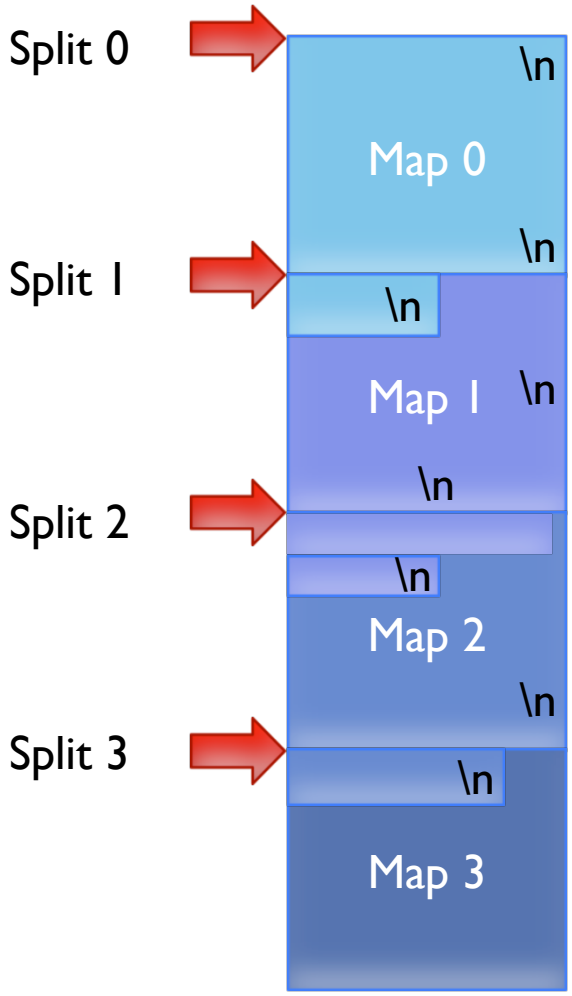


InputSplits and HDFS

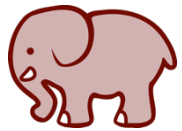
- Split is a portion of the data to be processed by a single map process in parallel
- Split = single block of data
  - records may straddle blocks
  - tasks often need to access a small portion of a record from a block on another node
  - Use larger blocks for a split to:
    - Increase performance due to more data for a mapper
    - A mapper has setup/teardown time that is constant
    - A small split means less work for the constant setup time
- `JobClient` computes input splits



InputSplit



- Usually crosses boundaries
  - of Mappers
  - of HDFS blocks

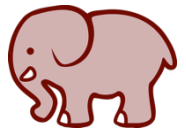


RecordReader

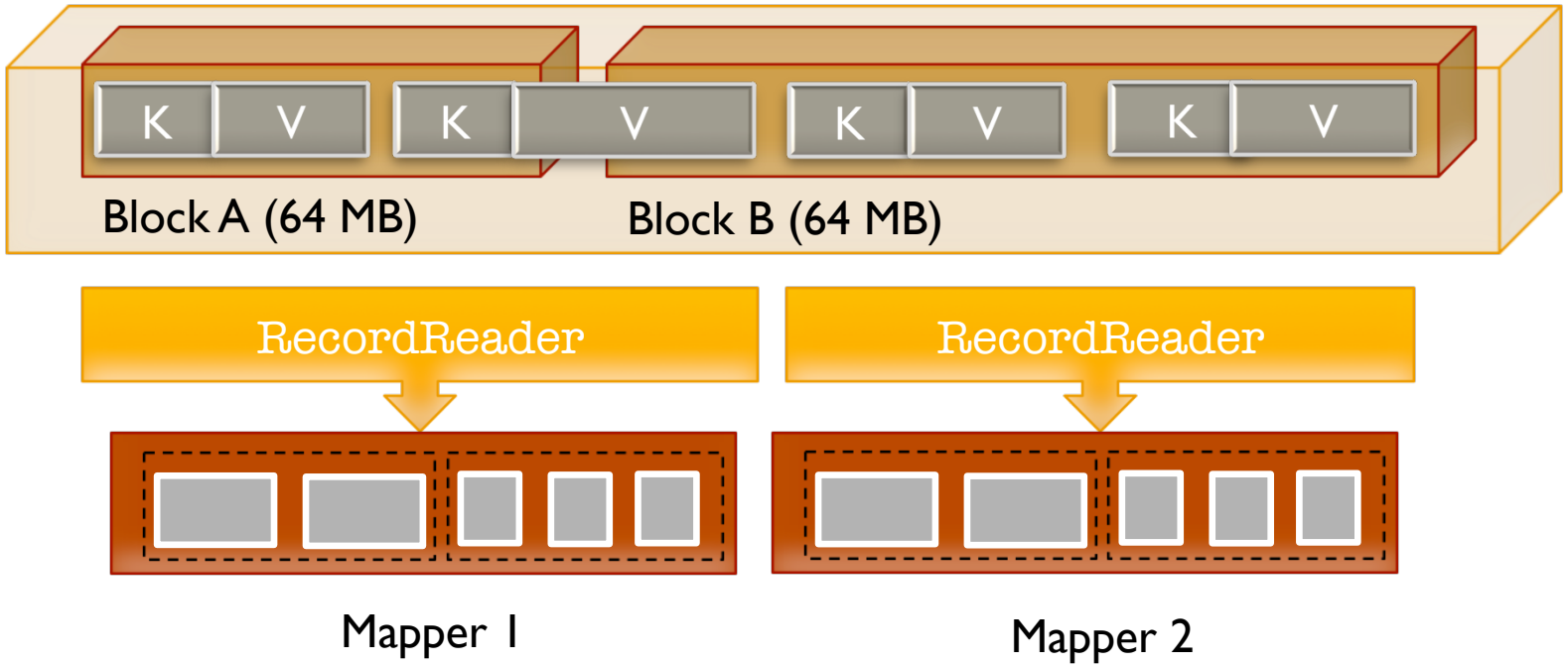
- The RecordReader breaks the data into key/value pairs for input to the Mapper

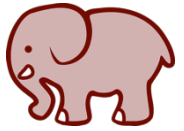
```
public interface RecordReader<K, V> {
  /* Reads the next key/value pair from the input for processing. */
  boolean next(K key, V value) throws IOException;

  /* Create an object of the appropriate type to be used as a key. */
  K createKey();

  /* Create an object of the appropriate type to be used as a value. */
  V createValue();

  /* Returns the current position in the input. */
  long getPos() throws IOException;

  /* Close this {@link InputSplit} to future operations. */
  public void close() throws IOException;

  /* How much of the input has the {@link RecordReader} consumed i.e.
     has been processed by? */
  float getProgress() throws IOException;
}
```

API docs: http://hadoop.apache.org/docs/r1.2.0/api/org/apache/hadoop/mapred/RecordReader.html



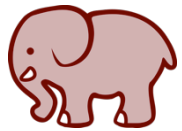
RecordReader – split vs. record





Binary Input

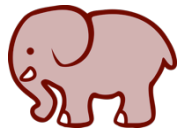
- SequenceFileInputFormat
  - Stores binary keys and values
  - Can be split
  - Can be compressed
  - Has many types
- Some variants exist
  - SequenceFileAsTextInputFormat
  - SequenceFileAsBinaryInputFormat



OutputFormat

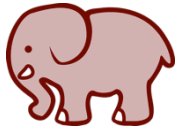
- Output files are all placed in the same directory for a job
- By default they are named `part-r` – with a five digit number
  - Numbering starts with `part-r-00000`
  - The numbering corresponds to the reducer number

```
$ hadoop fs -ls /user/shrek/output
Found 4 items
-rw-r--r--   1 shrek supergroup          0 2013-05-07 07:02 /user/shrek/output/_SUCCESS
drwxr-xr-x   - shrek supergroup          0 2013-05-07 07:00 /user/shrek/output/_logs
-rw-r--r--   1 shrek supergroup   43881658 2013-05-07 07:01 /user/shrek/output/part-r-00000
-rw-r--r--   1 shrek supergroup   42978232 2013-05-07 07:01 /user/shrek/output/part-r-00001
```



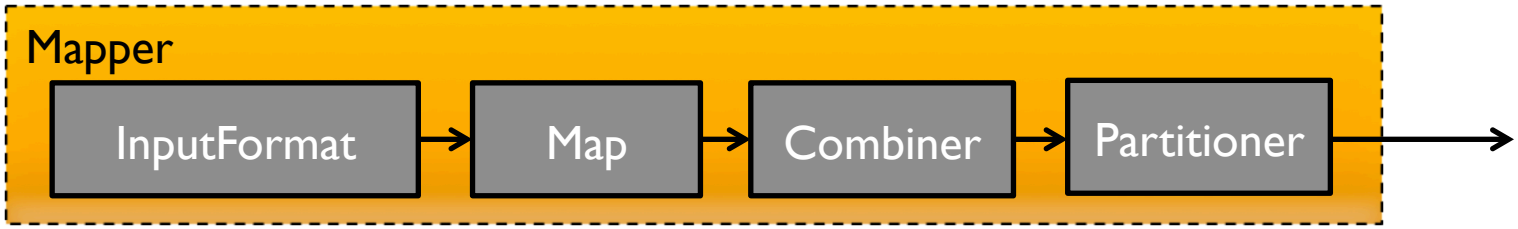
Lab 8: Create a custom InputFormat

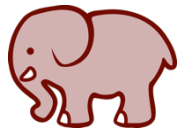
- Goal: Write an InputFormat that can read input files that are not lines of strings



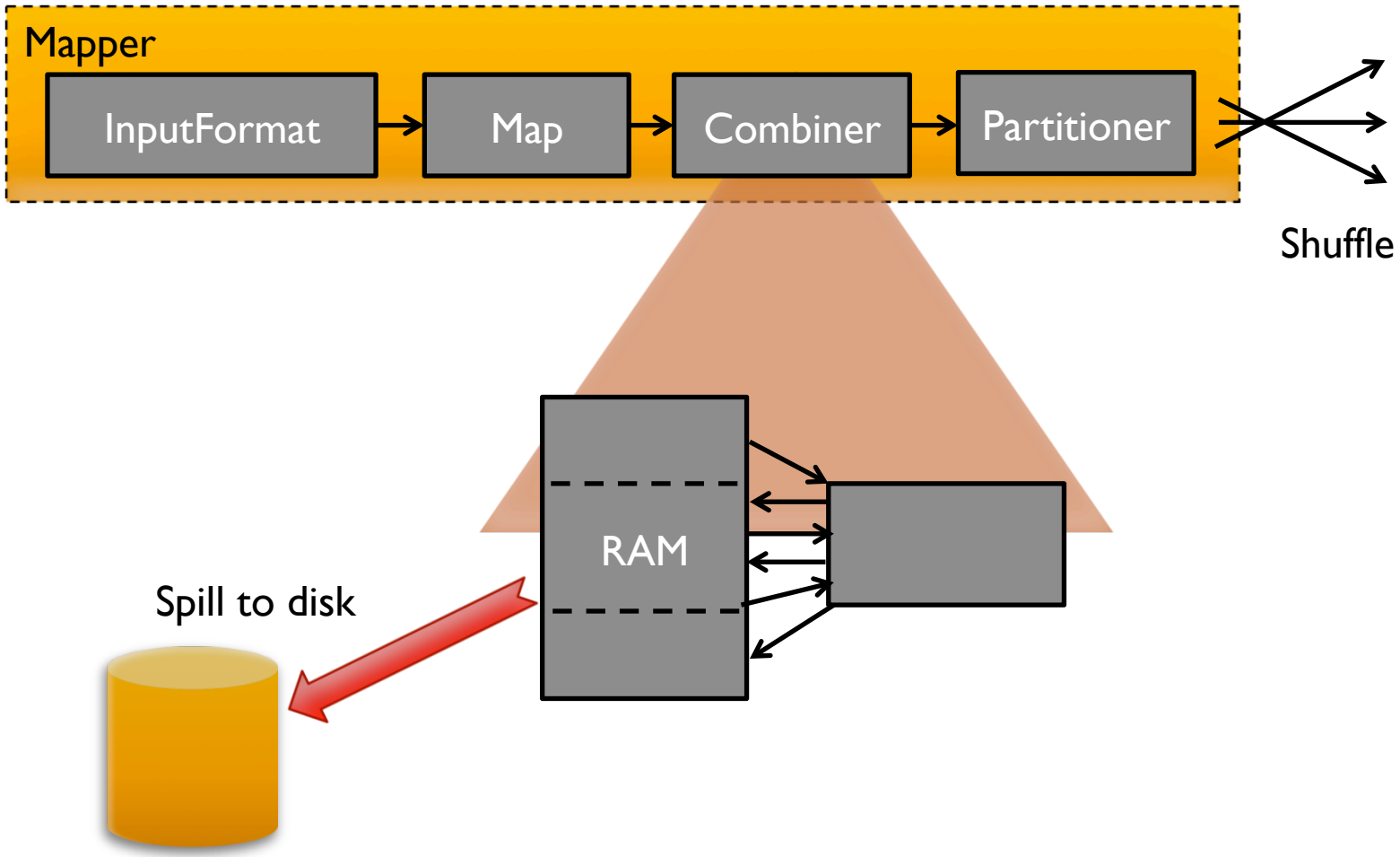
Combiner

- An optional component
- It is a map-side "mini-reducer"
  - Operates only on one map's data
- Reduces the amount of data to be shuffled
  - Across the wire
  - Should reduce the number of records
- *Caution:* Works only for commutative and associative functions

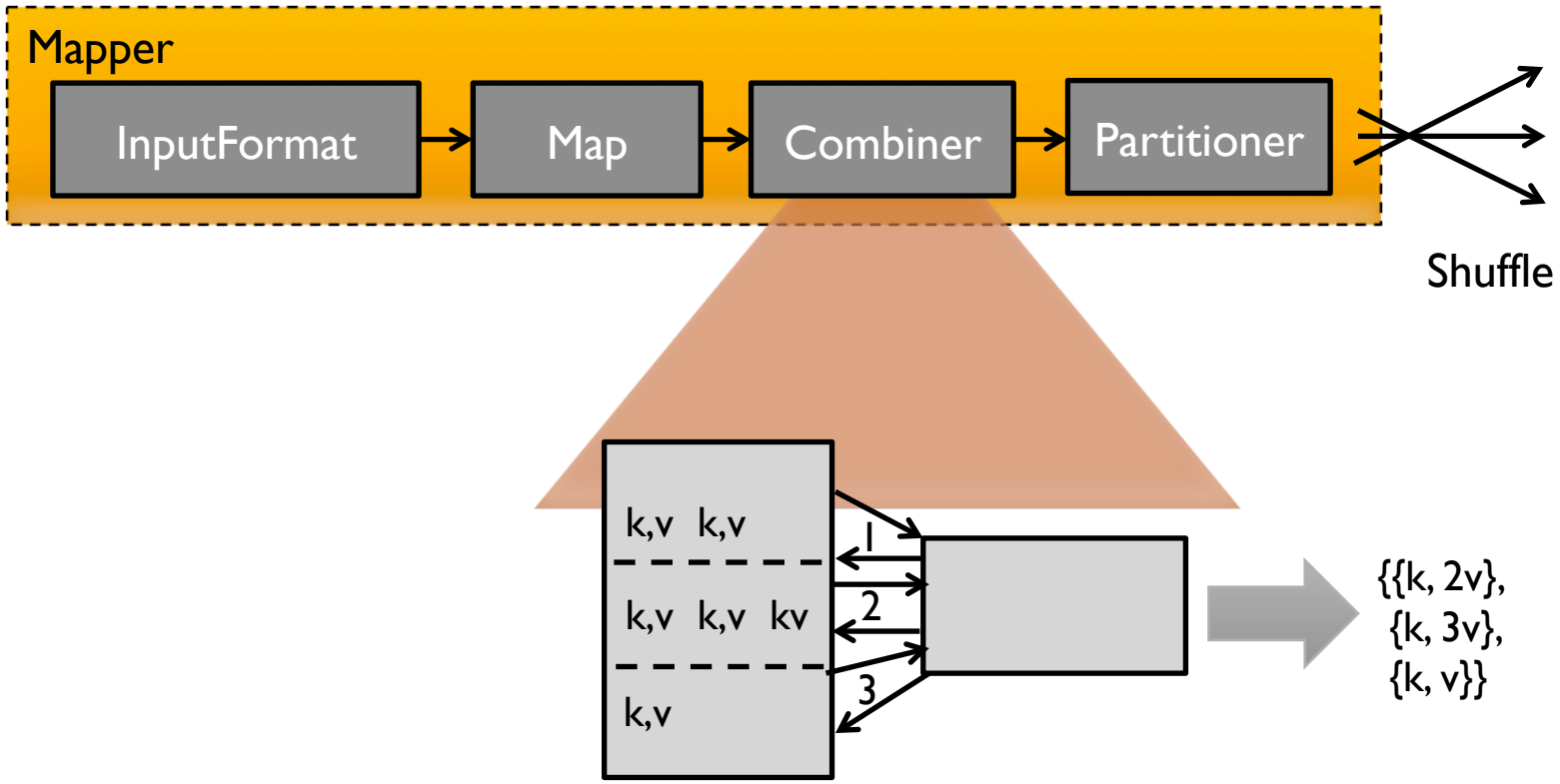


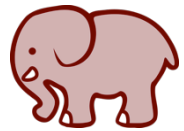


Map-side Reducer



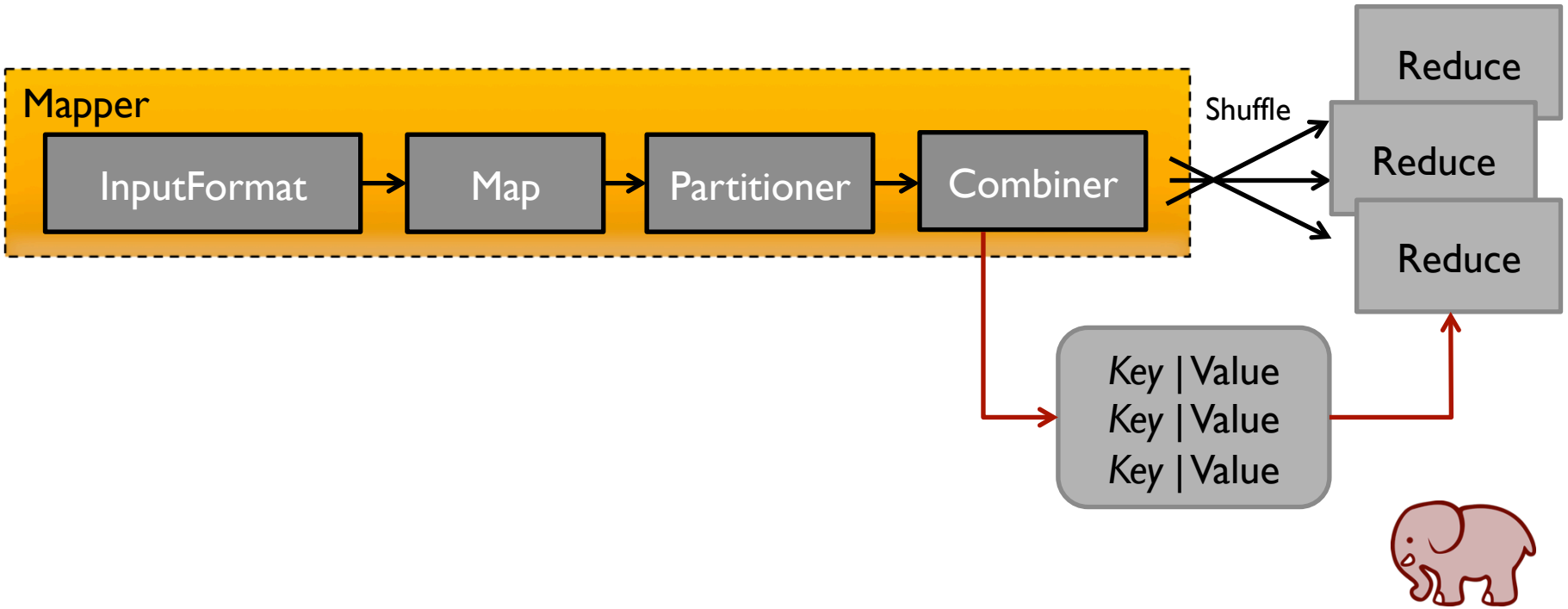
Map-Side Reducer



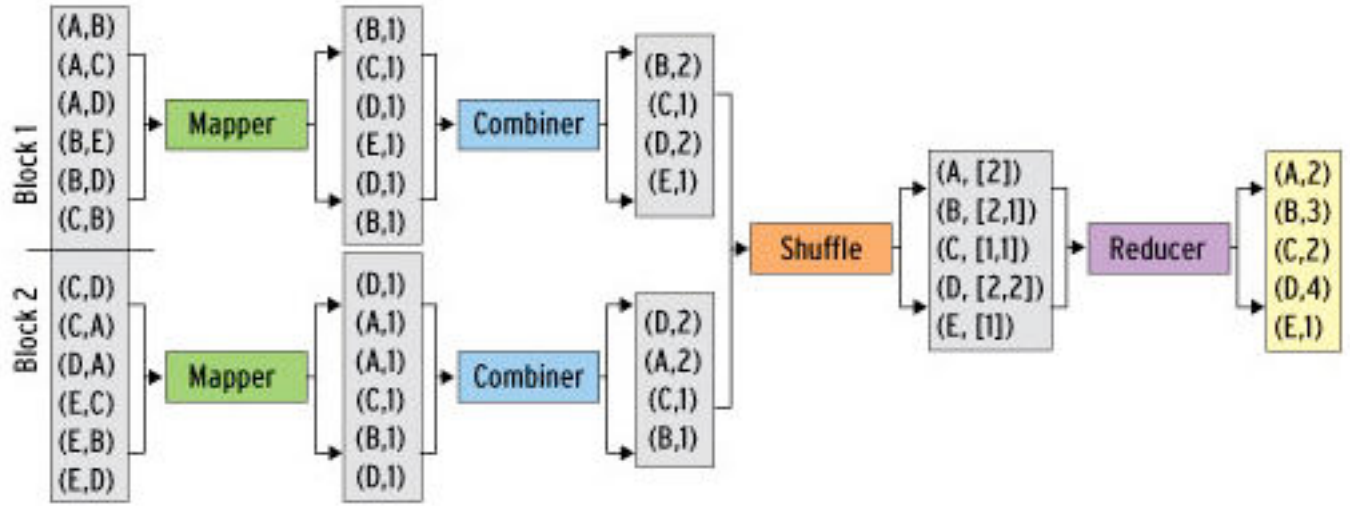


Combiner Contract

- Implements the Reducer interface
- Reducer's input pairs must be the same type as Map output
- Can run 0 to multiple iterations
- Cannot assume that incoming list is fully sorted data
- Cannot change the values of the keys

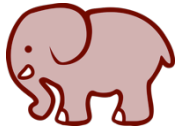


Combiner in Action



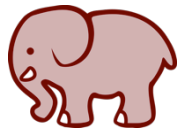
The use of a combiner makes sense for arithmetic operations in particular.

Source: http://www.admin-magazine.com/HPC/Articles/MapReduce-and-Hadoop



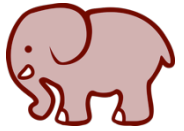
Combiner Cautions

- Should show repeatable results
  - No matter how many times it is run
- Combiners are slow
  - Compare MAP_OUTPUT_RECORDS without Combiner to COMBINER_OUTPUT_RECORDS with the Combiner
    - MAP_OUTPUT_RECORDS should be far greater than the COMBINER_OUTPUT_RECORDS
- Make sure the combiner does not cause any needed data to be dropped
- Works only for functions that satisfy Commutative and Associative property



Lab 9: Use combiner with MapReduce

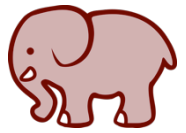
- Goal: Use the combiner to enhance performance of the job



Partitioner Contract

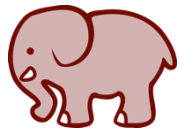
- Partitions the key space.
- Partitioner controls the partitioning of the keys of the intermediate map-outputs.
- Partitioner allocates data across the number of partitions that you specify
- Key hash value determines the reducers
  - `HashPartitioner.getPartition()`
- Can customize to ensure specific reducer targets
- The total number of partitions is the same as the number of reduce tasks for the job

API Docs: http://hadoop.apache.org/docs/r1.2.0/api/org/apache/hadoop/mapred/Partitioner.html



Default Partitioner

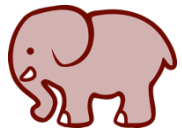
- Key's `hashcode()` is used
  - Converted to a positive value
  - Returns a number from 0 to (# of Reducers minus 1)
- Works on most keys
- May distribute unevenly
  - If keys hash to similar values
  - Customize to load-balance



HashPartitioner Class

```
/** Partition keys by their {@link Object#hashCode()}. */
public class HashPartitioner<K2, V2> implements Partitioner<K2, V2> {

  public void configure(JobConf job) {}

  /** Use {@link Object#hashCode()} to partition. */

  public int getPartition(K2 key, V2 value,
                          int numReduceTasks) {
    return (key.hashCode() & Integer.MAX_VALUE) % numReduceTasks;
  }

}
```

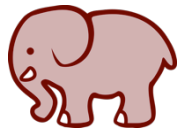
API docs: http://hadoop.apache.org/docs/r1.2.0/api/org/apache/hadoop/mapreduce/lib/partition/HashPartitioner.html



Partitioner Example

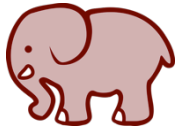
Key	Value	Partition
A	...	0
B	...	1
C	...	2
D	...	3
E	...	4
F	...	5
G	...	6
H	...	7
I	...	8
J	...	9
K	...	0
L	...	1
M	...	2
...		

- The developer needs to know the key space
- Sampling the data can help to create the understanding.
  - This could be divided up separately. For example 'S' might be broken up to go to several reducers.
    Sa to Sh 0
    Si to Sm 1
    Sn to Sz 2



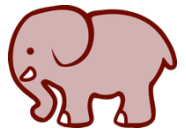
Lab 10: Create a custom Partitioner

- Goal: Write a MapReduce Java program that uses a custom partitioner
- Procedure:
  - Create AlphabeticalPartitioner class
  - Configure Hadoop to use your custom partitioner and to have 26 reducers
  - Compile and run the program



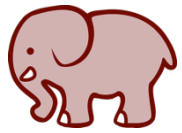
Apache Hadoop – A Developer's Course

Streaming



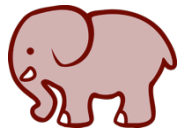
Streaming

- Hadoop streaming allows us to create and run MapReduce jobs with any executable or script
- Typically a scripting language is used
  - Supports Unix commands as mapper or reducer
    - `awk`, `sed`, `grep`
  - Supports scripting languages such as:
    - Python
    - Perl
  - Supports other languages such as:
    - C#

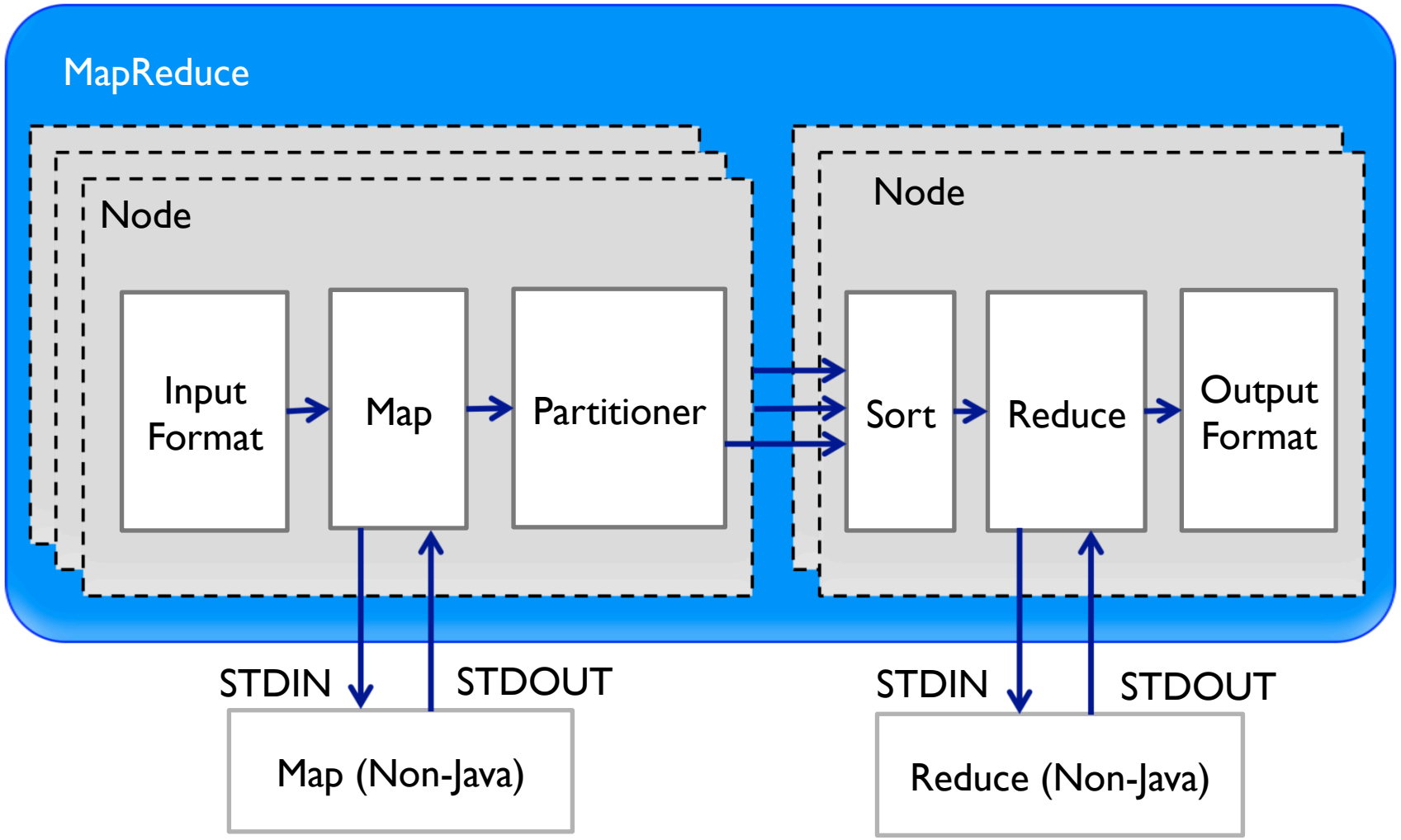


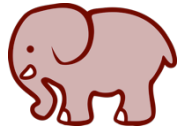
Uses of Streaming

- Useful for
  - Short MapReduce programs
    - more easily written in other languages
  - Quick answers
    - small quickly created data analysis needs
  - Data is text based
    - Each line read as a record
  - Integration with legacy languages



Streaming


MapReduce
Node
Input Format
Map
Partitioner
Node
Sort
Reduce
Output Format
STDIN
STDOUT
Map (Non-Java)
STDIN
STDOUT
Reduce (Non-Java)

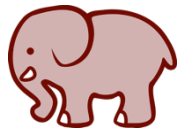


Streaming Syntax

- Look for the streaming classes in the contrib directory

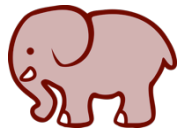
```
$ hadoop jar /usr/lib/hadoop-mapreduce/hadoop-streaming.jar
  -input inputfile
  -output outputfile
  -mapper myScript.py
  -file myScript.py
  -reducer /bin/somecommand
```

- Use built-in Unix commands to process data
  - Examples: wc, cat
  - A script using combination of above commands



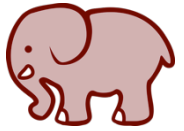
Streaming and Key-value pairs

- Streaming can operate on key/value pairs
- Uses the tab character as a separator
  - Default way to separate the key from the value in a record
  - First value is considered as a key
  - Remainder of line (excluding tab) is the value
  - Without a delimiter the entire record is read in as key



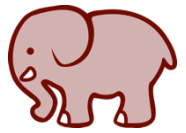
Lab 11: Use Python for MapReduce

- Goal: Demonstrate MapReduce using a non-Java language such as Python
- Procedure:
  - Examine the Python map application
  - Ensure that you have map data
  - Run the streaming job
  - Display the streaming job



Apache Hadoop – A Developer's Course

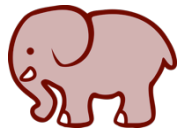
Distributed Cache



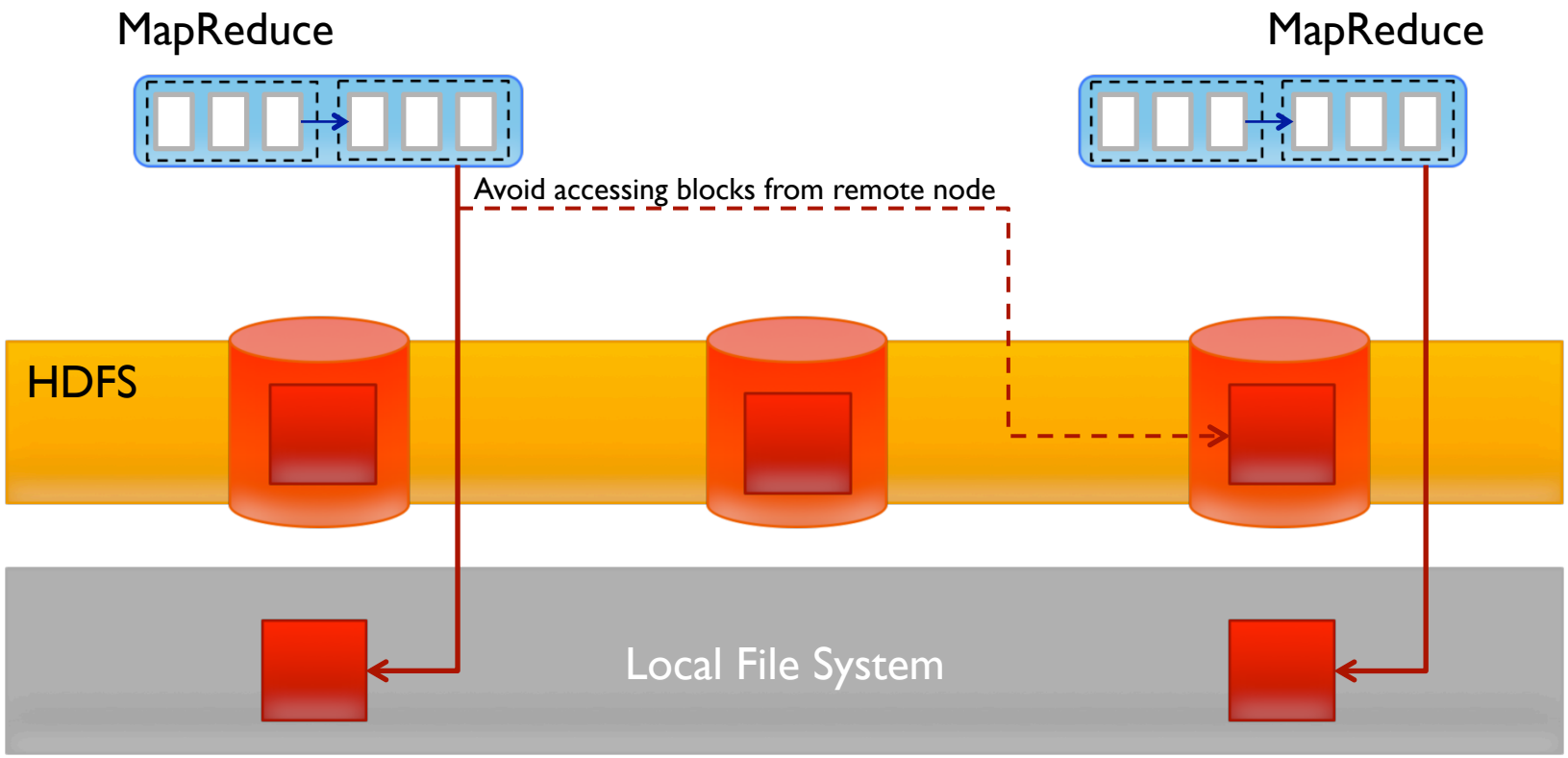
Distributed Cache

- Distributes application-specific large, read-only files efficiently
- Facility provided by Map-Reduce to cache files (text, archives, jars) needed by applications
- Specify files via URLs (hdfs:// or http://) to be cached via `JobConf`
- Framework copies files to task nodes
  - Before the tasks are launched
  - Files are read-only
  - Archives are un-archived on the slave
- Good for supplemental data
  - Used *locally* by mappers and reducers
- Often stored on node's Linux file system
  - Optionally upload from the local file system of the client
- Efficient
  - Once per job (i.e. node) [not once per task]
  - Many tasks from the same job may run on a single node

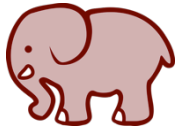
API docs: http://hadoop.apache.org/docs/stable/api/org/apache/hadoop/filecache/DistributedCache.html



Shared file access

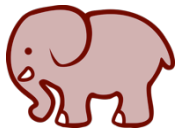


Avoid Network I/O by copying file once and keeping on local file system



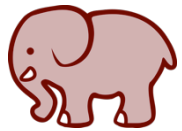
Distributed Cache Uses

- As a software distribution mechanism
  - For example to distribute shared objects (.so files)
- -files option
  - allows to provide a comma-separated list of paths
- -archive option
  - allows to distribute .jar, .zip, .tar or .tar.gz files
  - Decompression of archives is automatic
- -libjars option
  - modifies classpath of the mapper and reducer tasks
- Default cache size
  - is 10GB
  - can be set with the config local.cache.size ???????



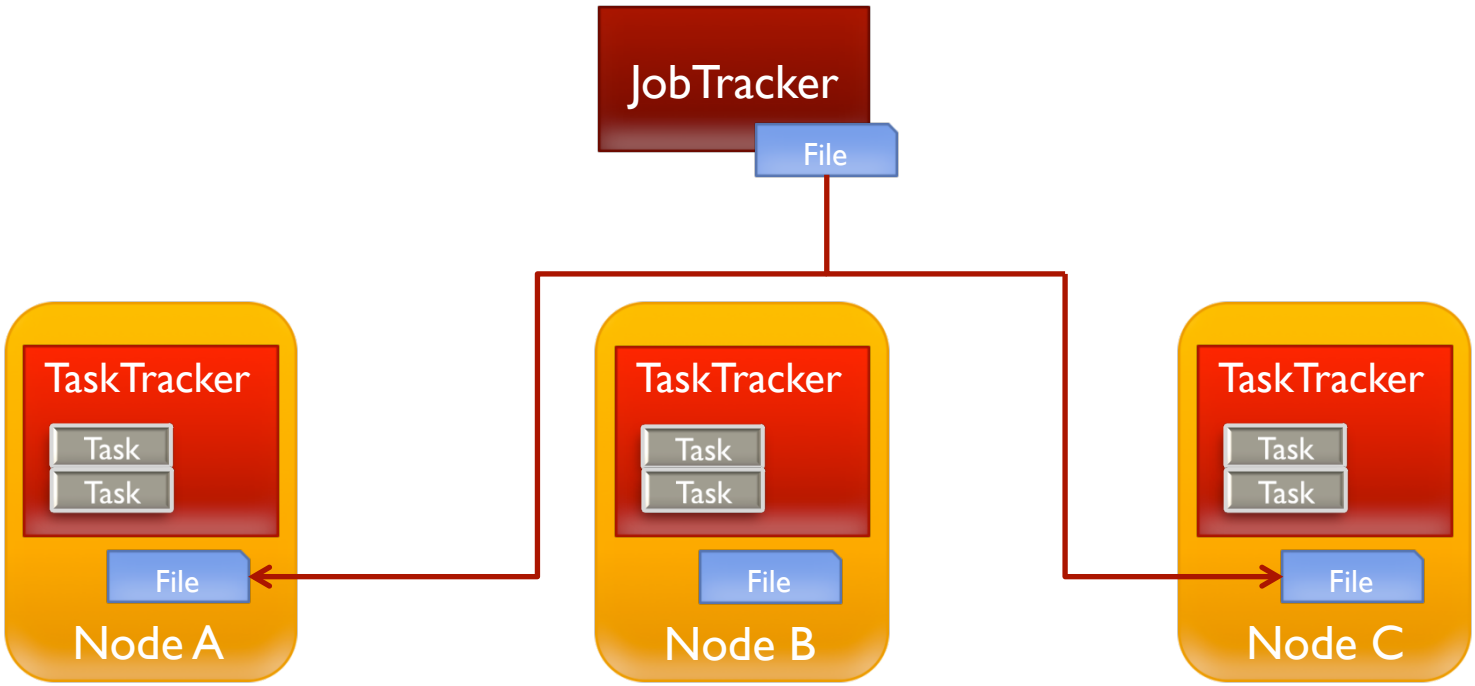
Distributed Cache Uses

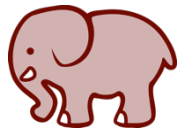
- Classification
  - Keywords for documents
  - City name for zip codes
- Distributed data
  - for Map-side Joins



Cache Deployment

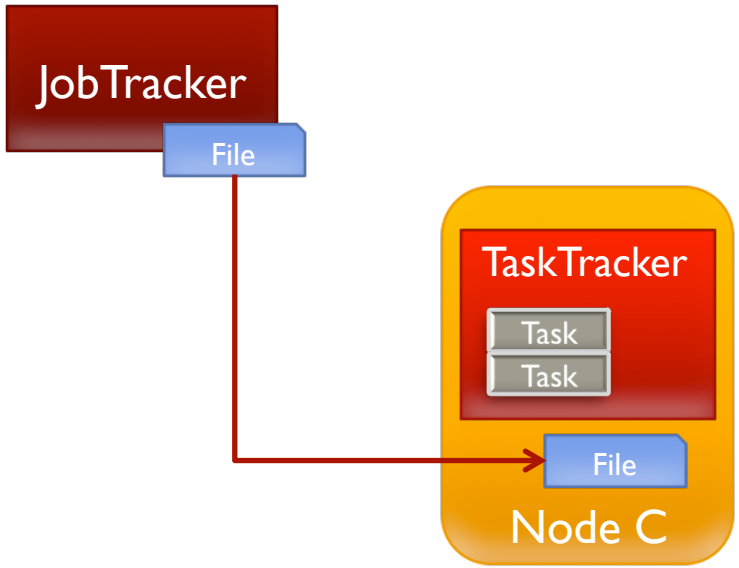
- TaskTracker deploys files
  - From JobTracker's file system to local disks
  - Before the job is run
- Reference count is maintained
  - For each task that is using each file in cache

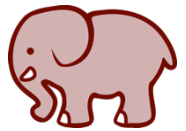




Using Distributed Cache

- Place the file on HDFS
- Call the static method `DistributedCache.addCacheFile()` or `DistributedCache.addCacheArchive()` to specify the files
- Mappers on each individual TaskTracker call method `getLocalCacheFiles()` to get an array of local file Paths where the local copy is located
- The mapper uses Java file I/O techniques to read the local copy

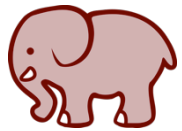




Configure Method

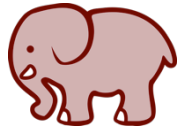
- Used to load the cache data into memory when the mapper is first initialized
- It is overwritten each time the mapper is initialized
- Provides an array of file paths to local copy of files pushed by DistributedCache
- Cache file is retrieved using its original name placed relative to the working directory of the task

```
public static class MyMapClass extends MapReduceBase
                implements Mapper<K, V, K, V> {
    private Path[] localArchives;
    private Path[] localFiles;

    public void configure(JobConf job) {
        localArchives  = DistributedCache.getLocalCacheArchives(job);
        localFiles = DistributedCachc.getLocalCacheFiles(job);
    }

    public void map(K key, V value, OutputCollector<K, V> output,
            Reporter reporter) throws IO Exception {
        // Use data from the cached archives and files here
    }
}
```



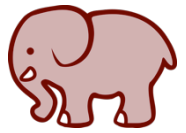
Lab 12: MapReduce with Distributed Cache

- Goal: Write a MapReduce program that uses the Distributed Cache
- Procedure:
  - Configure your job to get the local cache file for your mapper
  - In your Map class, open the File from the Distributed Cache
  - Tell conf to add the cache file
  - Compile and run your code



Apache Hadoop – A Developer's Course

Monitoring and Debugging MapReduce



Monitoring a Hadoop job

http://localhost:50030/jobtracker.jsp

localhost Hadoop Map/Reduce Administration

Quick Links

State: RUNNING
Started: Fri May 17 23:17:38 PDT 2013
Version: 2.0.0-mr1-cdh4.1.2, Unknown
Compiled: Thu Nov 1 18:05:52 PDT 2012 by jenkins from Unknown
Identifier: 201305172317

Cluster Summary (Heap Size is 29.5 MB/888.94 MB)

Running Map Tasks	Running Reduce Tasks	Total Submissions	Nodes	Occupied Map Slots	Occupied Reduce Slots	Reserved Map Slots	Reserved Reduce Slots	Map Task Capacity	Reduce Task Capacity	Avg. Tasks/Node	Blacklisted Nodes	Excluded Nodes
0	1	1	1	0	1	0	0	2	2	4.00	0	0

Scheduling Information

Queue Name	State	Scheduling Information
default	running	N/A

Filter (Jobid, Priority, User, Name)

Example: 'user:smith 3200' will filter by 'smith' only in the user field and '3200' in all fields

Running Jobs

Jobid	Priority	User	Name	Map % Complete	Map Total	Maps Completed	Reduce % Complete	Reduce Total	Reduces Completed	Job Scheduling Information	Diagnostic Info
job_201305172317_0001	NORMAL	anupam	word count	100.00%	1	1	0.00%	1	0	NA	NA

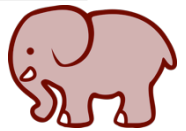
Retired Jobs

none

Local Logs

Log directory, Job Tracker History

Hadoop, 2013.



TaskTracker

http://localhost:50030/jobdetails.jsp?jobid=job_201305172317_0001&refresh=30

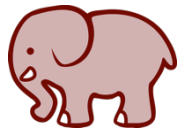
Hadoop job_201305172317_0001 on localhost

User: anupam
Job Name: word count
Job File: hdfs://localhost:8020/users/anupam/.staging/job_201305172317_0001/job.xml
Submit Host: anupam-centos
Submit Host Address: 127.0.0.1
Job-ACLs: All users are allowed
Job Setup: Successful
Status: Succeeded
Started at: Mon May 20 23:11:56 PDT 2013
Finished at: Mon May 20 23:12:45 PDT 2013
Finished in: 49sec
Job Cleanup: Successful

Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	100.00%	1	0	0	1	0	0 / 0
reduce	100.00%	1	0	0	1	0	0 / 0

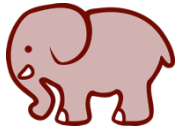
	Counter	Map	Reduce	Total
File System Counters	FILE: Number of bytes read	0	0	140
	FILE: Number of bytes written	0	0	369,122
	FILE: Number of read operations	0	0	0
	FILE: Number of large read operations	0	0	0
	FILE: Number of write operations	0	0	0
	HDFS: Number of bytes read	0	0	177
	HDFS: Number of bytes written	0	0	90
	HDFS: Number of read operations	0	0	2
	HDFS: Number of large read operations	0	0	0
	HDFS: Number of write operations	0	0	1
	Launched map tasks	0	0	1
	Launched reduce tasks	0	0	1

- Drill-down from JobTracker screen



Lab 13: Monitoring a Job with TaskTracker

- Goal: Running a MapReduce job and analyze statistics in JobTracker
- Procedure:
  - Run a Java MapReduce program
  - Use JobTracker GUI to analyze the job



Debugging MapReduce Jobs

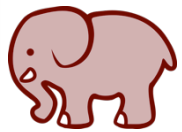
- Use an IDE to step through
- Find the log files under /var/log/hadoop*

```
$ pwd
/var/log
$ ls -al | grep hadoop
drwxrwxr-x. 4 root      hadoop      12288 May 20 23:11 hadoop-0.20-mapreduce
drwxrwxr-x. 2 hdfs      hadoop       4096 May 17 23:17 hadoop-hdfs
drwxrwxr-x  2 httpfs    httpfs       4096 Apr 22 10:36 hadoop-httpfs
drwxrwxr-x  2 mapred    hadoop       4096 Apr 22 10:36 hadoop-mapreduce
drwxrwxr-x  2 yarn      hadoop       4096 Apr 22 10:36 hadoop-yarn
```

  - Typically hadoop-*username-service-hostname*.log
  - Service could be 'jobtracker', 'namenode', 'tasktracker', etc.
  - For MapReduce jobs, 'tasktracker' is most relevant

- TaskTracker logs go to $HADOOP_LOG_DIR/userlogs
  - Start at JobTracker UI and drill down from there

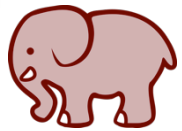
```
$ pwd
/var/log/hadoop-0.20-mapreduce/userlogs/job_201305172317_0001
$ ls  -al
total 20
lrwxrwxrwx 1 mapred mapred 114 May 20 23:12 attempt_201305172317_0001_m_000000_0 -> /
var/lib/hadoop-hdfs/cache/mapred/mapred/local/userlogs/job_201305172317_0001/
attempt_201305172317_0001_m_000000_0
```



Logging

- All Hadoop daemons and jobs use log4j
  - /etc/hadoop/conf/log4j.properties configures cluster-wide
  - One should use log4j, but with a different configuration file, or a programmable configuration for logging level
- Hadoop logs output from System.out.println() and System.err.println()
  - Stored in files named stdout, stderr and syslog
  - It is spread around the cluster
    - Co-located on nodes with TaskTracker daemons
    - $(HADOOP_LOG_DIR)/userlogs/<subdirs>

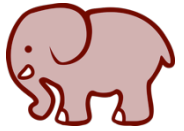
```
$ pwd
/var/log/hadoop-0.20-mapreduce
$ ls -al | grep ^d
drwxrwxr-x.  4 root   hadoop    12288 May 21 07:40 .
drwxr-xr-x. 25 root   root       4096 May 19 19:01 ..
drwxr-xr-x.  3 mapred mapred     4096 May 20 23:12 history
drwxr-xr-x.  3 mapred mapred     4096 May 20 23:12 userlogs
```



Finding location of log files

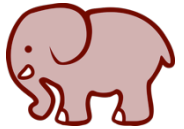
- Two methods used
  - Use the JobTracker GUI and follow the links to see your stdout, stderr and syslog generated by log4j
  - Write a script to collect logs from the cluster
    - Links to logs are stored at `$HADOOP_LOG_DIR/userlogs`
    - Other `userlogs` directories kept by each TaskTracker
    - May cause confusion when you only see a subset of your logs in a given `userlogs` directory

```
$ pwd
/var/log/hadoop-0.20-mapreduce/userlogs/job_201305172317_0001/attempt_201305172317_0001_m_000000_0
$ ls -al
total 16
drwx------ 2 mapred mapred 4096 May 20 23:12 .
drwxr-xr-x 6 mapred mapred 4096 May 20 23:12 ..
-rw-r--r-- 1 mapred mapred  143 May 20 23:12 log.index
-rw-r--r-- 1 mapred mapred    0 May 20 23:12 stderr
-rw-r--r-- 1 mapred mapred    0 May 20 23:12 stdout
-rw-r--r-- 1 mapred mapred 1514 May 20 23:12 syslog
```



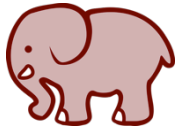
Lab 14: Debug MapReduce code

- Goal: Write a simple MapReduce program that uses log4j and standard error output (stderr) to debug code



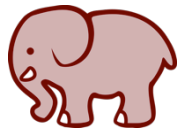
Lab 15: Practice MapReduce use-case

- Weather statistics
- Stock market statistics



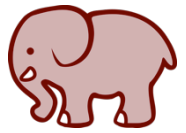
Apache Hadoop – A Developer's Course

Fundamentals of PIG



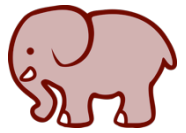
Apache Hadoop – A Developer's Course

Grunt



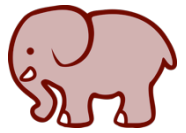
Apache Hadoop – A Developer's Course

PIG Latin



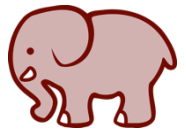
Apache Hadoop – A Developer's Course

Introduction to Hive



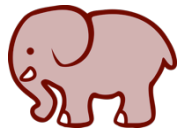
Apache Hadoop – A Developer's Course

Hive Data



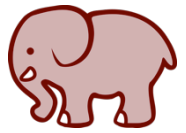
Apache Hadoop – A Developer's Course

Hive Operators and Functions



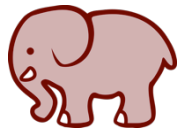
Apache Hadoop – A Developer's Course

HCatalog



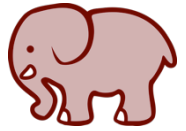
Apache Hadoop – A Developer's Course

HBase



Apache Hadoop – A Developer's Course

HBase Architecture and API



Apache Hadoop – A Developer's Course

Hadoop Web Services