Parallel Data Processing with Hadoop/MapReduce

CS240A Tao Yang, 2016

Overview

- **What is MapReduce?**
  - –Example with word counting
- **Parallel data processing with MapReduce**
  - –Hadoop file system
- **More application example**

Motivations

CLUSTER COMPUTING

- **Motivations**
    - Large-scale data processing on clusters
    - Massively parallel (hundreds or thousands of CPUs)
    - Reliable execution with easy data access
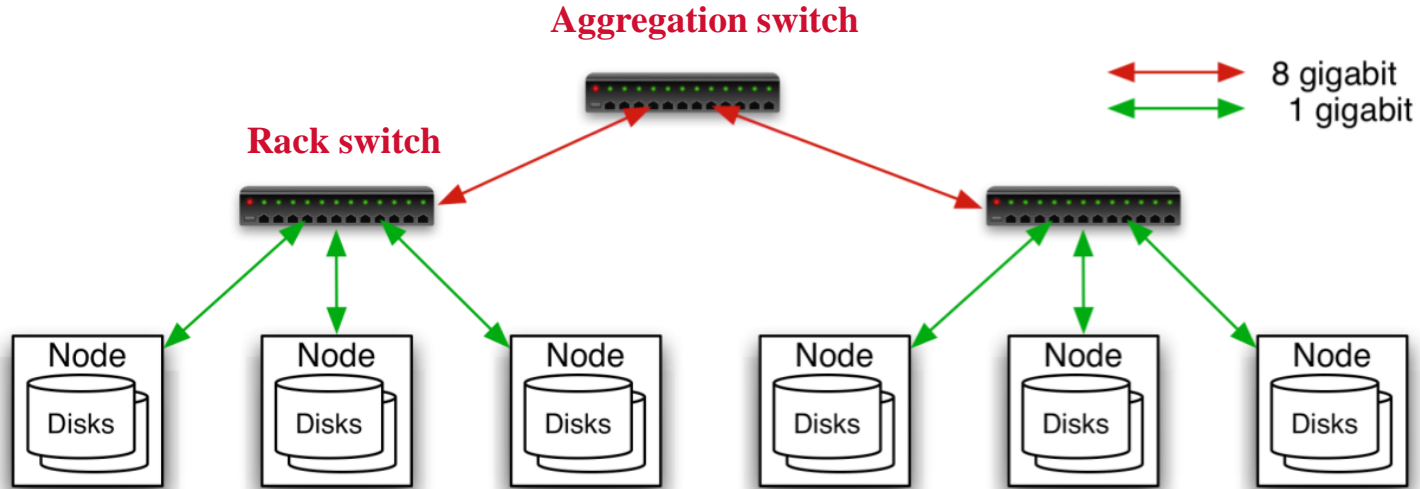- **Functions**
    - Automatic parallelization & distribution
    - Fault-tolerance
    - Status and monitoring tools
    - A clean abstraction for programmers
        - » Functional programming meets distributed computing
        - » A batch data processing system

Parallel Data Processing in a Cluster

- **Scalability to large data volumes:**
  - Scan 1000 TB on 1 node @ 100 MB/s = 24 days
  - Scan on 1000-node cluster = 35 minutes

- **Cost-efficiency:**
  - Commodity nodes /network
    - » Cheap, but not high bandwidth, sometime unreliable
  - Automatic fault-tolerance (fewer admins)
  - Easy to use (fewer programmers)

Typical Hadoop Cluster

- **40 nodes/rack, 1000-4000 nodes in cluster**
- **1 Gbps bandwidth in rack, 8 Gbps out of rack**
- **Node specs :
  8-16 cores, 32 GB RAM, 8 × 1.5 TB disks**

MapReduce Programming Model

- **Inspired from map and reduce operations commonly used in functional programming languages like Lisp.**

- **Have multiple map tasks and reduce tasks**

- **Users implement interface of two primary methods:**
    - Map: (key1, val1) → (key2, val2)
    - Reduce: (key2, [val2 list]) → [val3]

Inspired by LISP Function Programming

- **Two Lisp functions**
- **Lisp *map* function**
  - Input parameters: a function and a set of values
  - This function is applied to each of the values.

    Example:
  - (map 'length '(() (a) (ab) (abc)))

  → (length(()) length(a) length(ab) length(abc))

  → (0 1 2 3)

Lisp Reduce Function

- Lisp *reduce* function
  - given a binary function and a set of values.
  - It combines all the values together using the binary function.
- Example:
  - use the + (add) function to reduce the list (0 1 2 3)
  - (reduce   #'+   '(0 1 2 3))
  - → 6

Map/Reduce Tasks for World of Key-Value Pairs

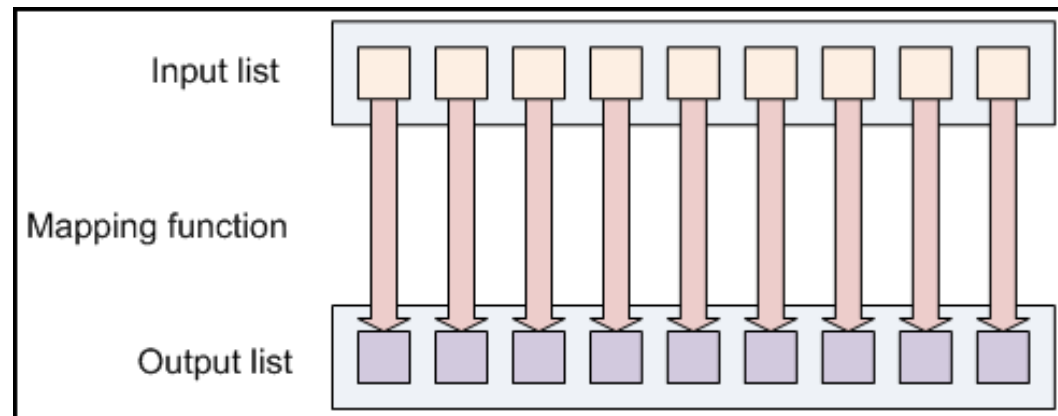|			
<satish, 26000>	<gopal, 50000>	<satish, 26000>	<satish, 26000>
<Krishna, 25000>	<Krishna, 25000>	<kiran, 45000>	<Krishna, 25000>
<Satishk, 15000>	<Satishk, 15000>	<Satishk, 15000>	<manisha, 45000>
<Raju, 10000>	<Raju, 10000>	<Raju, 10000>	<Raju, 10000>

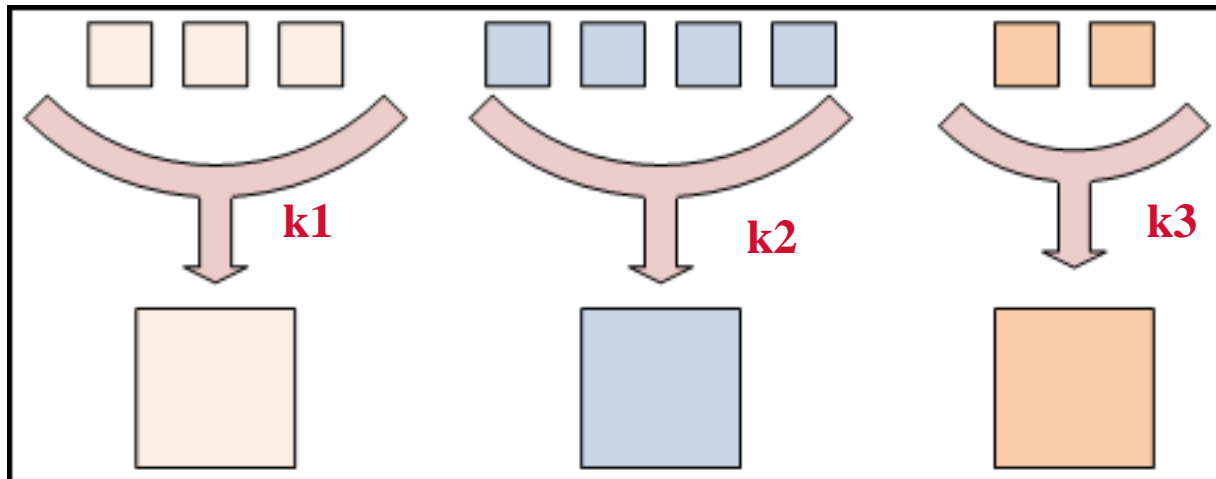**Map Tasks**   **Reduce Tasks**

Example: Map Processing in Hadoop

- **Given a file**
  - A file may be divided by the system into multiple parts (called splits or shards).

- **Each record in a split is processed by a user Map function,**
  - takes each record as an input
  - produces key/value pairs

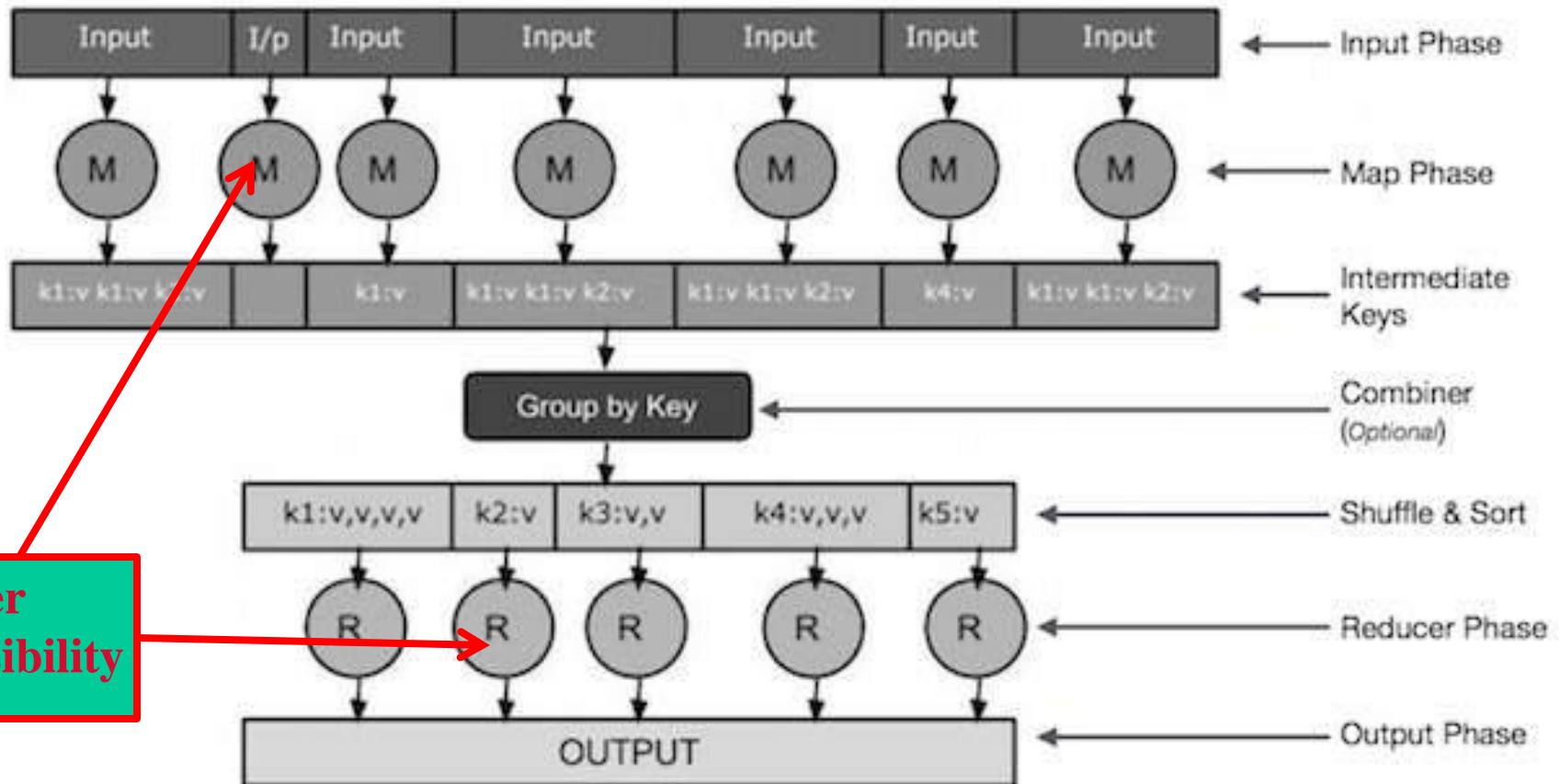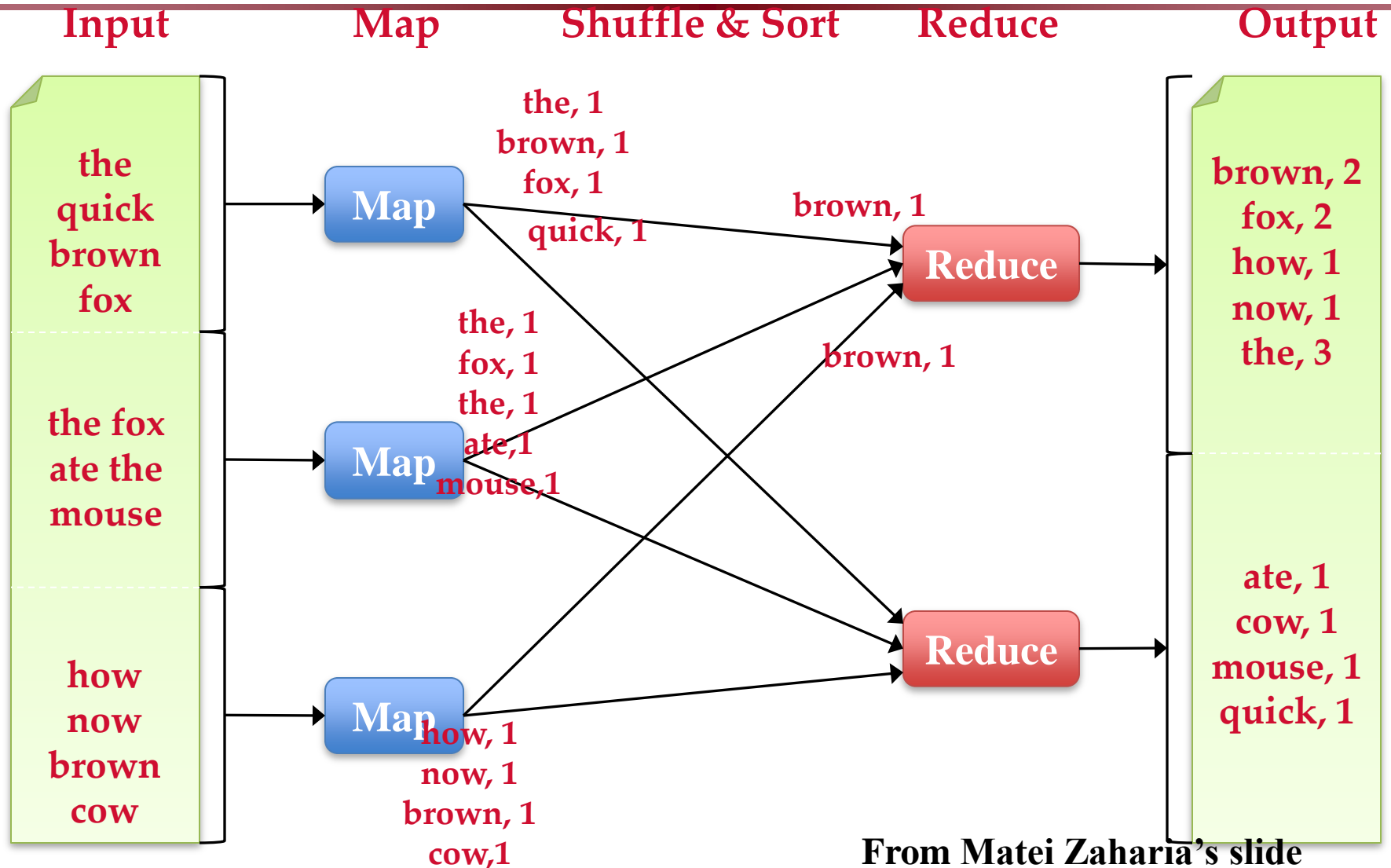| Input list	Mapping function	Output list

Processing of Reducer Tasks

- **Given a set of (key, value) records produced by map tasks.**
  - all the intermediate values for a key are combined together into a list and given to a reducer. Call it [val2]
  - A user-defined function is applied to each list [val2] and produces another value

Put Map and Reduce Tasks Together

Example of Word Count Job (WC)

Input	Map	Shuffle & Sort	Reduce	Output

Input:
- the quick brown fox
- the fox ate the mouse
- how now brown cow

Map outputs:
- the, 1 / brown, 1 / fox, 1 / quick, 1
- the, 1 / fox, 1 / the, 1 / ate, 1 / mouse, 1
- how, 1 / now, 1 / brown, 1 / cow, 1

Shuffle & Sort:
- brown, 1
- brown, 1

Output:
- brown, 2 / fox, 2 / how, 1 / now, 1 / the, 3
- ate, 1 / cow, 1 / mouse, 1 / quick, 1

From Matei Zaharia's slide

Input/output specification of the WC mapreduce job

Input : a set of (key values) stored in files

  key: document ID

  value: a list of words as content of each document

Output: a set of (key values) stored in files

  key: wordID

  value: word frequency appeared in all documents

MapReduce function specification:

  map(String input_key, String input_value):

   reduce(String output_key, Iterator intermediate_values):

Pseudo-code

map(String input_key, String input_value):
// input_key: document name
// input_value: document contents
    for each word w in input_value:
      EmitIntermediate(w, "1");

reduce(String output_key, Iterator intermediate_values):
// output_key: a word
// output_values: a list of counts
    int result = 0;
    for each v in intermediate_values:
      result = result + ParseInt(v);
    Emit(output_key, AsString(result));

MapReduce WordCount.java

Hadoop distribution: **src/examples/org/apache/hadoop/examples/WordCount.java**

```java
public static class TokenizerMapper
    extends Mapper<Object, Text, Text, IntWritable>{

  private final static IntWritable one = new IntWritable(1); // a mapreduce int class
  private Text word = new Text(); //a mapreduce String  class

  public void map(Object key, Text value, Context context
          ) throws IOException, InterruptedException { // key is the offset of
  current record in a file
    StringTokenizer itr = new StringTokenizer(value.toString());
    while (itr.hasMoreTokens()) { // loop for each token
      word.set(itr.nextToken());  //convert from string to token
      context.write(word, one);  // emit (key,value) pairs for reducer
    }
  }
```
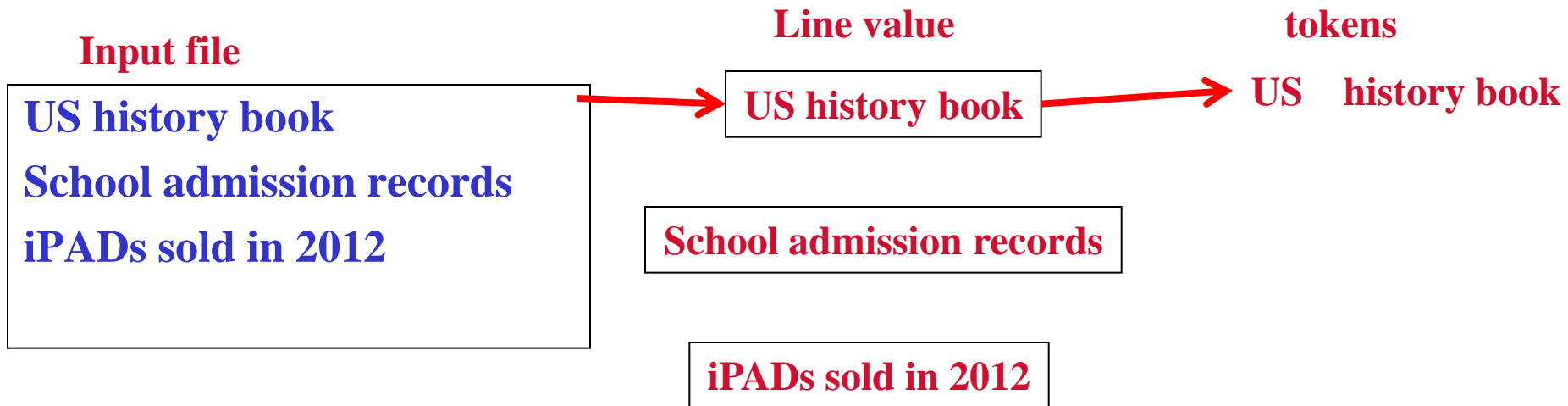
MapReduce WordCount.java

map() gets a key, value, and context
- key - "bytes from the beginning of the line?"
- value - the current line;

in the while loop, each token is a "word" from the current line

Input file

US history book
School admission records
iPADs sold in 2012

Line value

US history book

School admission records

iPADs sold in 2012

tokens

US   history book

Reduce code in WordCount.java

```java
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);  //convert  "int" to IntWritable
    context.write(key, result); //emit the final key-value result
  }
}
```

The driver to set things up and start

```
//  Usage: wordcount <in> <out>
public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
    Job job = new Job(conf, "word count"); //mapreduce job
    job.setJarByClass(WordCount.class); //set jar file
    job.setMapperClass(TokenizerMapper.class); // set mapper class
    job.setCombinerClass(IntSumReducer.class); //set combiner class
    job.setReducerClass(IntSumReducer.class);   //set reducer class
    job.setOutputKeyClass(Text.class);          // output key class
    job.setOutputValueClass(IntWritable.class);  //output value class
    FileInputFormat.addInputPath(job, new Path(otherArgs[0]));   //job input path
    FileOutputFormat.setOutputPath(job, new Path(otherArgs[1])); //job output path
    System.exit(job.waitForCompletion(true) ? 0 : 1);  //exit status
```

Systems Support for MapReduce
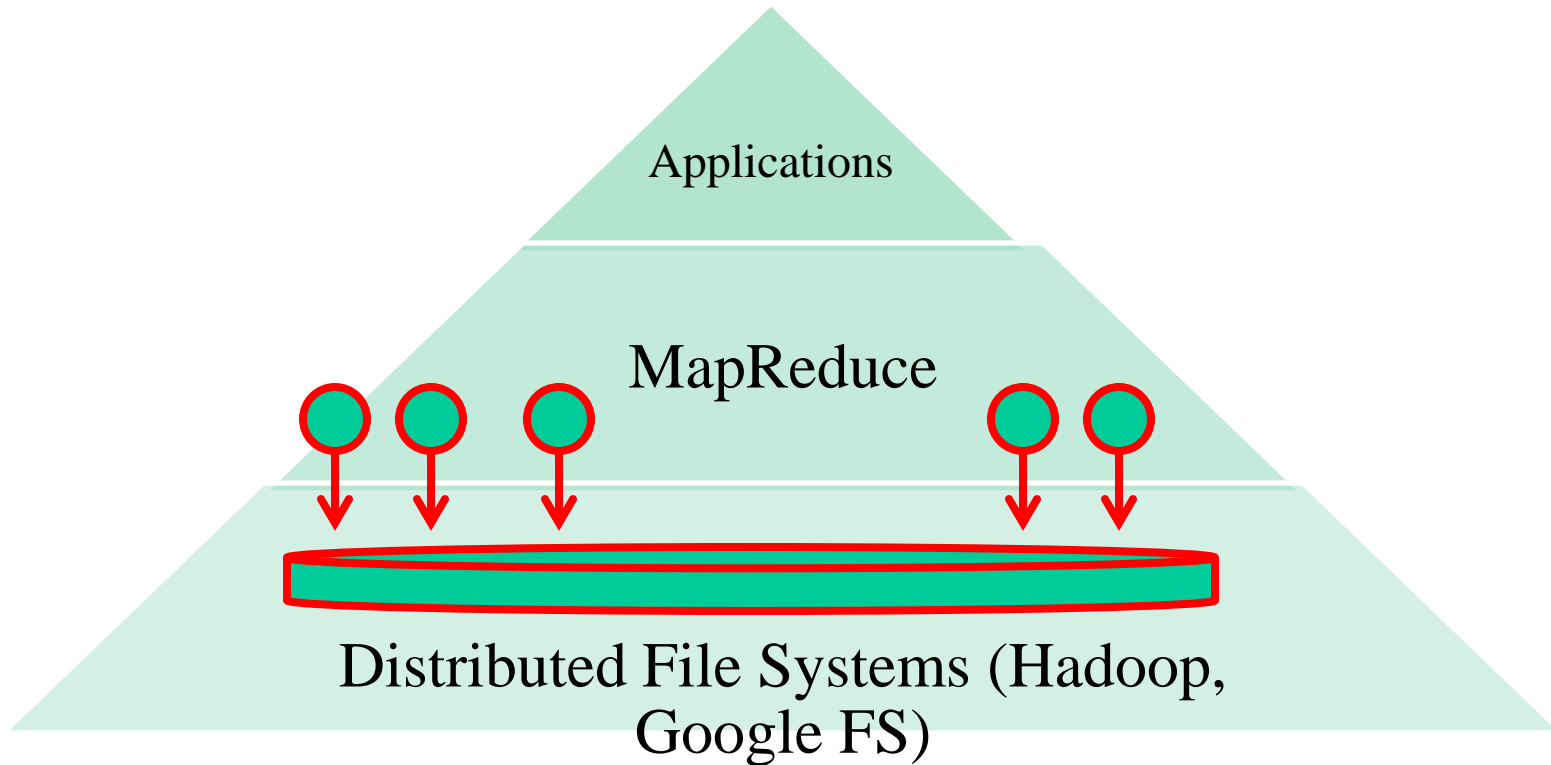
Distributed Filesystems

- **The interface is the same as a single-machine file system**
  - create(), open(), read(), write(), close()
- **Distribute file data to a number of machines (storage units).**
  - Support replication
- **Support concurrent data access**

  - Fetch content from remote servers. Local caching
- **Different implementations sit in different places on complexity/feature scale**
  - Google file system and Hadoop HDFS
    - » Highly scalable for large data-intensive applications.
    - » Provides redundant storage of massive amounts of data on cheap and unreliable computers

Assumptions of GFS/Hadoop DFS

- **High component failure rates**
  - Inexpensive commodity components fail all the time
- **"Modest" number of HUGE files**
  - Just a few million
  - Each is 100MB or larger; multi-GB files typical
- **Files are write-once, mostly appended to**
  - Perhaps concurrently
- **Large streaming reads**
- **High sustained throughput favored over low latency**

Hadoop Distributed File System

- **Files split into 64 MB blocks**

- **Blocks replicated across several datanodes (3)**

- **Namenode stores metadata (file names, locations, etc)**

- **Files are append-only. Optimized for large files, sequential reads**
  - Read: use any copy
  - Write: append to 3 replicas

Namenode

File1
1
2
3
4

1 2 1 3
2 1 4 2
4 3 3 4

Datanodes
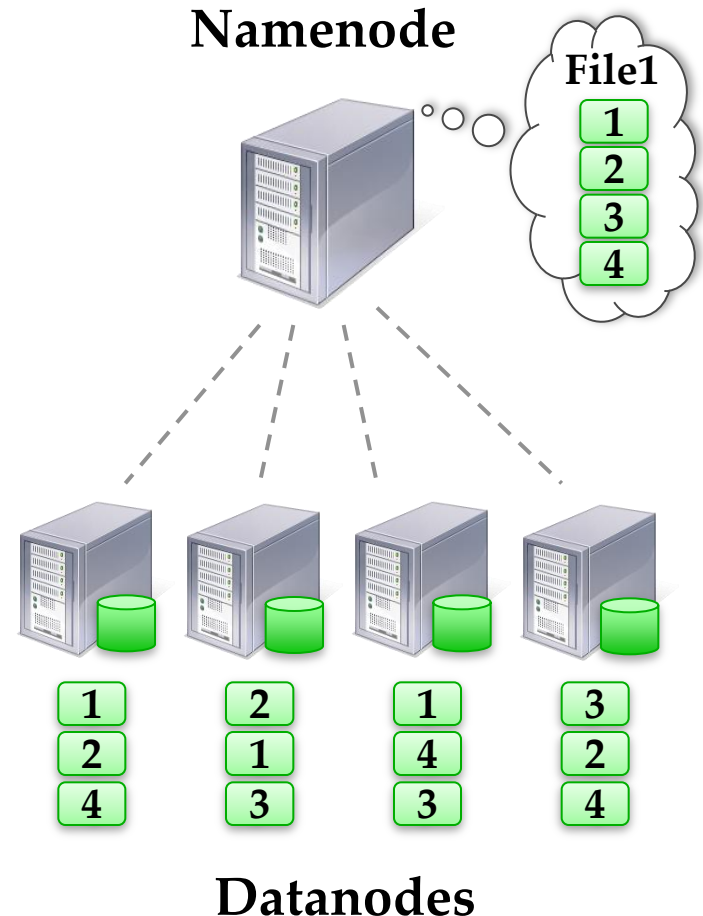
Shell Commands for Hadoop File System

Hapdoop ⟷ **Local Linux**
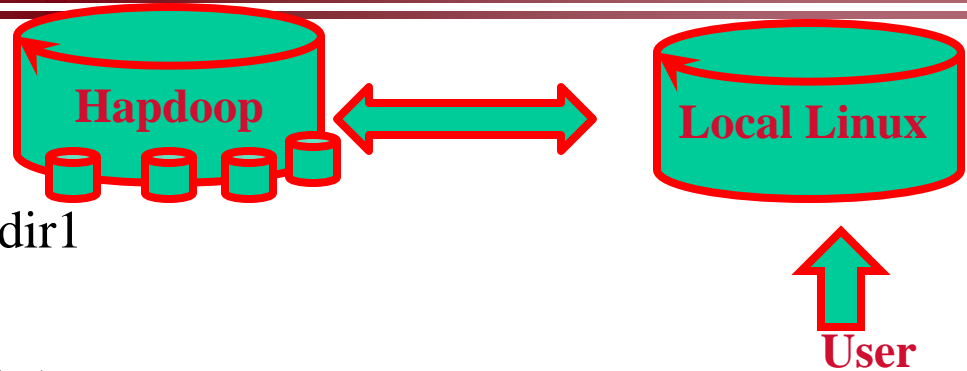
User

- **Mkdir, ls, cat, cp**
  - hadoop fs -mkdir /user/deepak/dir1
  - hadoop fs -ls /user/deepak
  - hadoop fs -cat /usr/deepak/file.txt
  - hadoop fs -cp /user/deepak/dir1/abc.txt /user/deepak/dir2
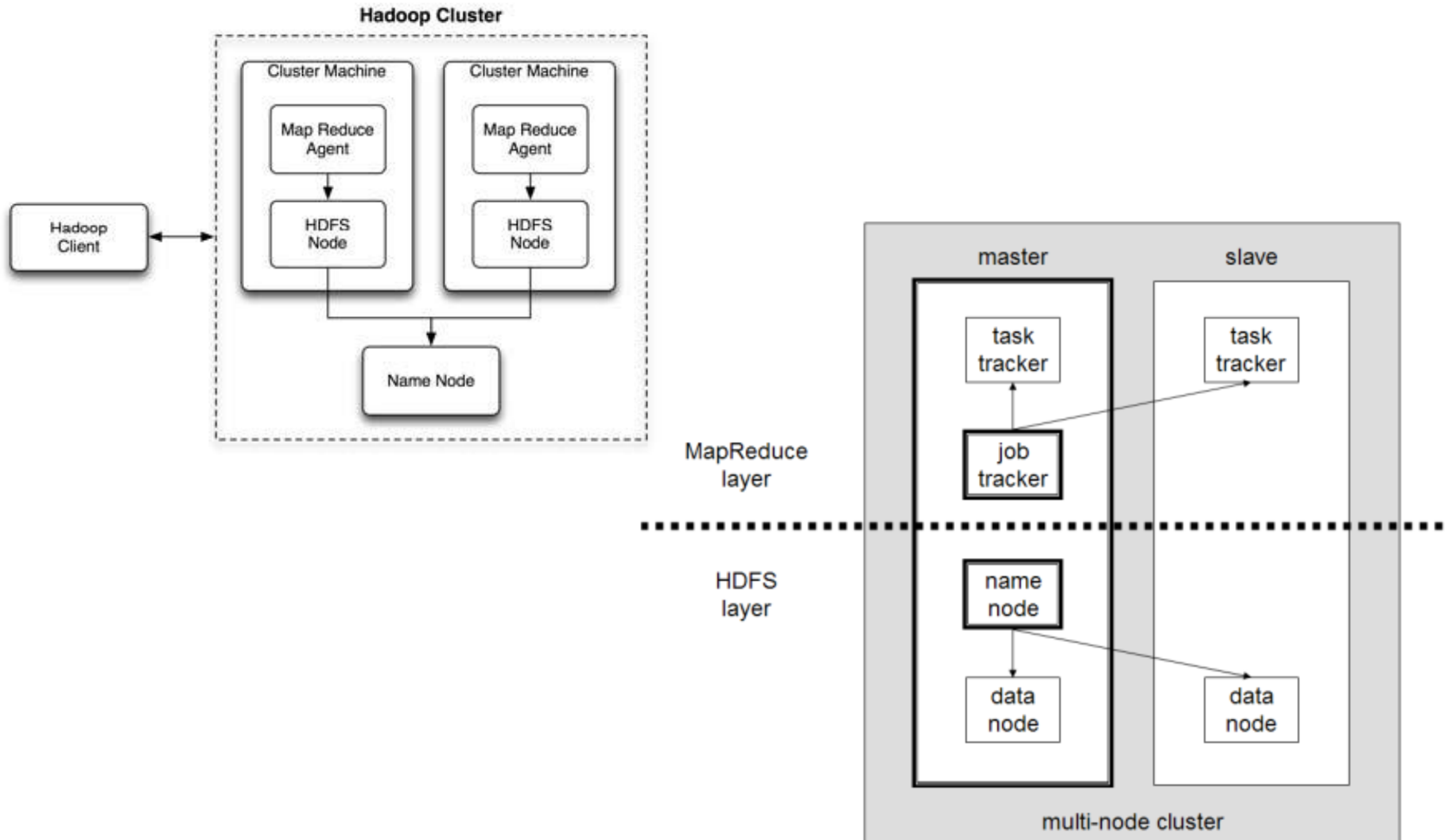- **Copy data from the local file system to HDF**
  - hadoop fs -copyFromLocal <src:localFileSystem> <dest:Hdfs>
  - Ex: hadoop fs –copyFromLocal /home/hduser/def.txt /user/deepak/dir1
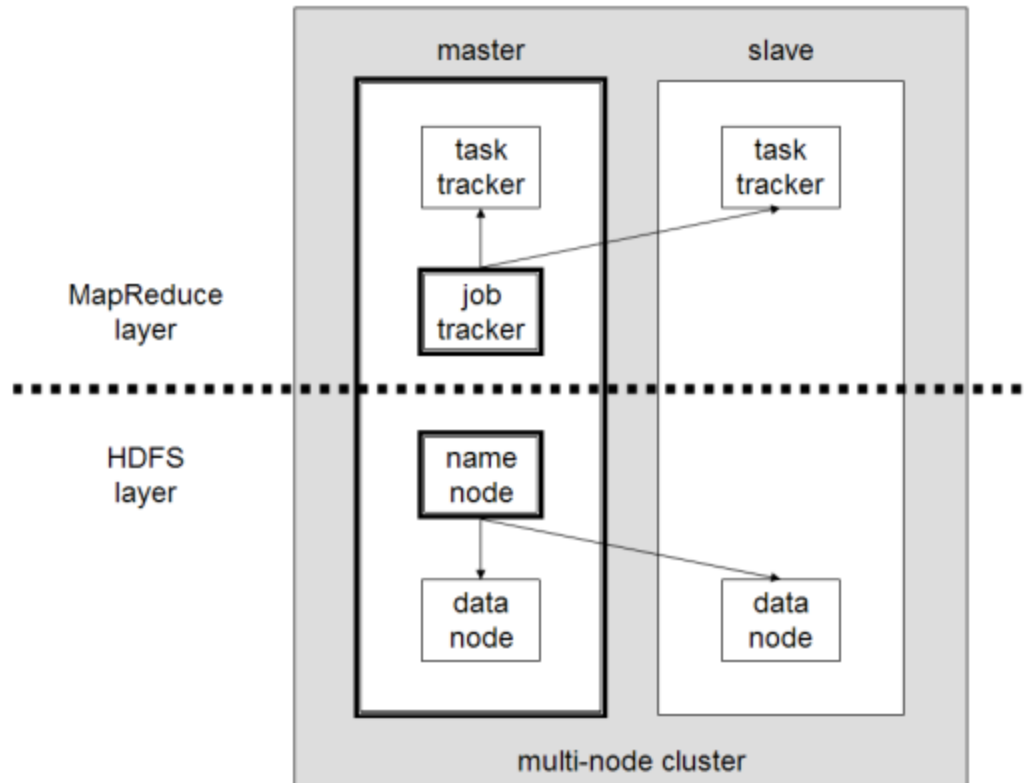- **Copy data from HDF to local**
  - hadoop fs -copyToLocal <src:Hdfs> <dest:localFileSystem>
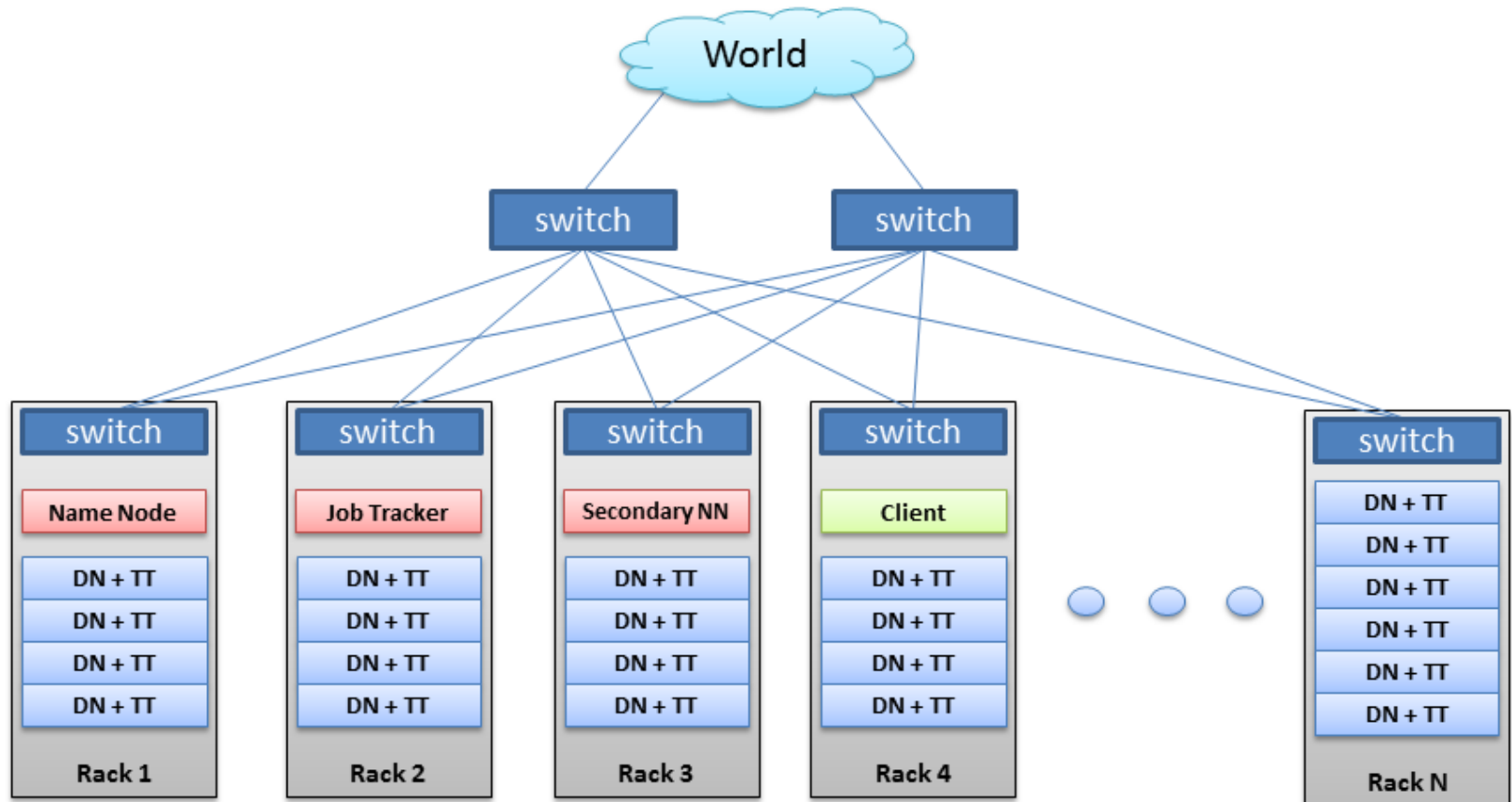
Hadoop DFS with MapReduce

Demons for Hadoop/Mapreduce

- **Following demons must be running**

(use jps to show these

Java processes)

- **Hadoop**
    - Name node (master)
    - Secondary name node
    - data nodes
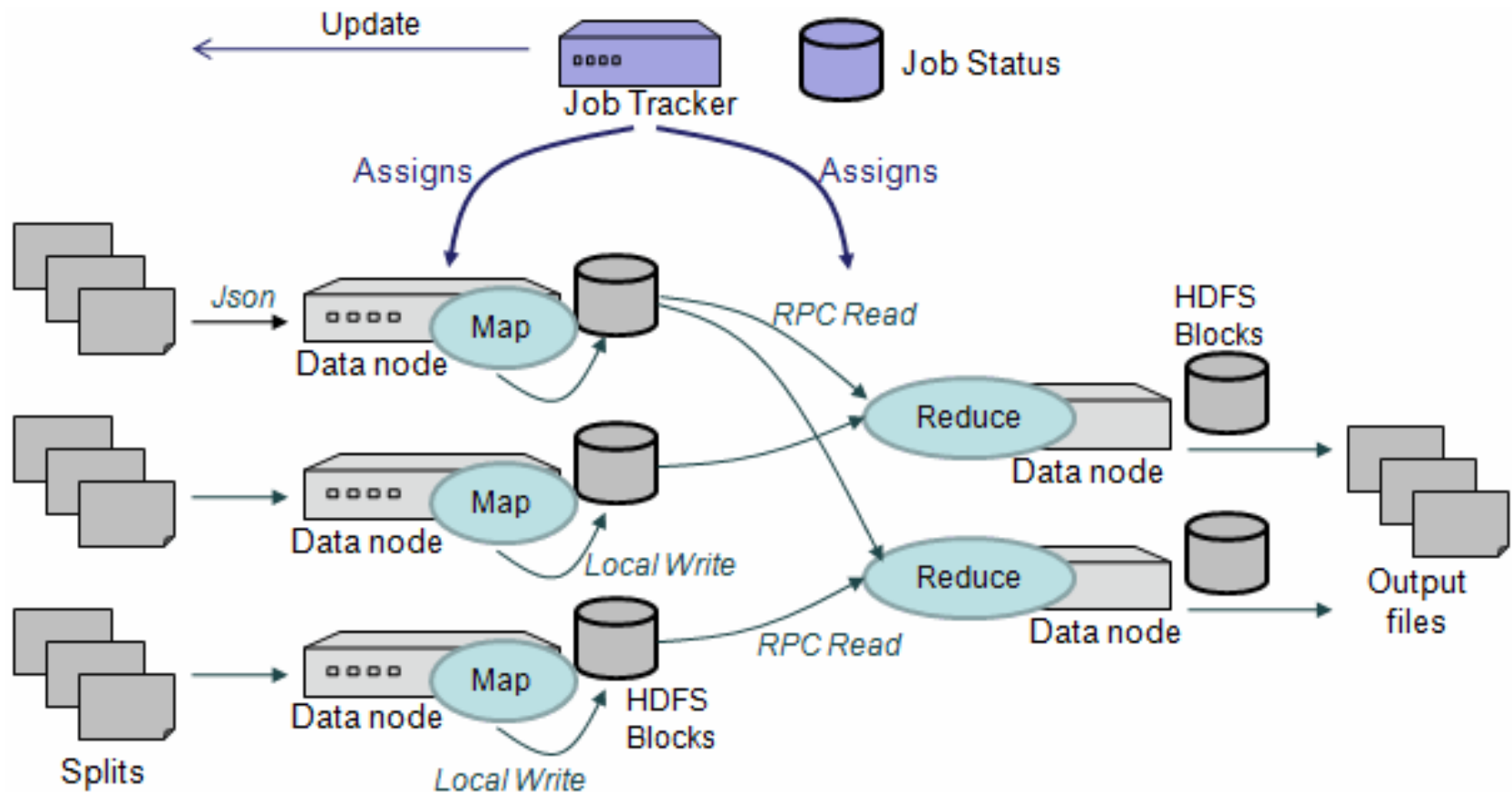- **Mapreduce**
    - Task tracker
    - Job tracker

Hadoop Cluster with MapReduce

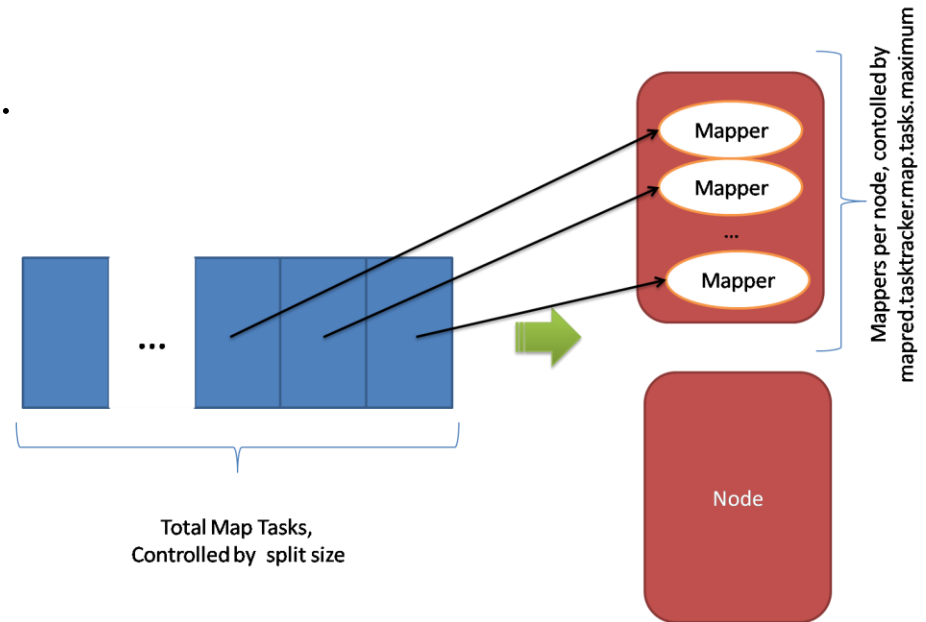# Execute MapReduce on a cluster of machines with Hadoop DFS

MapReduce: Execution Details

- **Input reader**
  - Divide input into <u>splits</u>, assign each split to a Map task
- **Map task for data parallelism**
  - Apply the Map function to each record in the split
  - Each Map function returns a list of (key, value) pairs
- **Shuffle/Partition and Sort**
  - Shuffle distributes sorting & aggregation to many reducers
  - All records for key k are directed to the same reduce processor
  - Sort groups the same keys together, and prepares for aggregation
- **Reduce task for data parallelism**
  - Apply the Reduce function to each key
  - The result of the Reduce function is a list of (key, value) pairs
- **Performance consideration in mappers/reducers: Too many key-value pairs? Not enough pairs?**
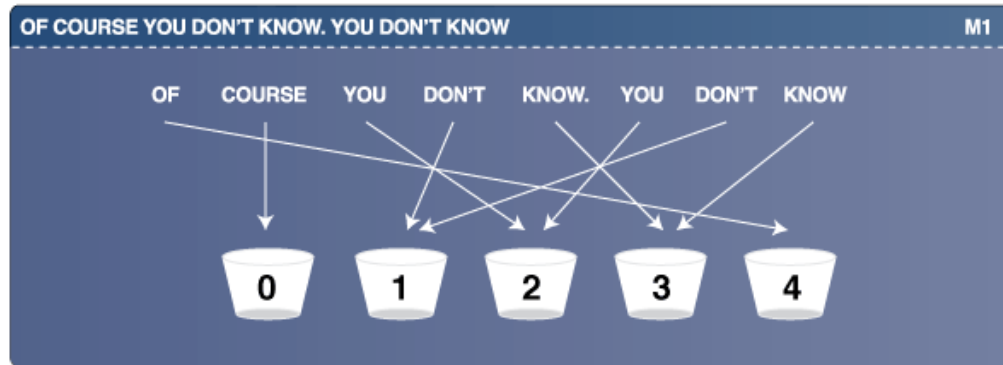
29

How to create and execute map tasks?

- **The system spawns a number of mapper processes and reducer processes**
  - A typical/default setting 2 mappers and 1 reducer per core.
  - User can specify/change setting
- **Input reader**
  - Input is typically a directory of files.
  - Divide each input file into <u>splits</u>,
  - Assign each split to a Map task
- **Map task**
  - Executed by a mapper process
  - Apply the user-defined map function to each record in the split
  - Each Map function returns a list of (key, value) pairs

Total Map Tasks, Controlled by split size

Mappers per node, contolled by mapred.tasktracker.map.tasks.maximum

Mapper

Mapper

...

Mapper

Node

How to create and execute reduce tasks?

- **Partition (key, value) output pairs of map tasks**

Partitioning is based on hashing, and can be modified.

| Key	Hash	Hash % 4
of	-1463488791	4
course	2334184425	0
you	1116843962	2
don't	-482782459	1
know	326123353	3

How to create and execute reduce tasks?

- **Shuffle/partition outputs of map tasks**
  - Sort keys and group values of the same key together.
  - Direct (key, values) pairs to the partitions, and then distribute to the right destinations.
- **Reduce task**
  - Apply the Reduce function to the list of each key
- **Multiple map tasks -> one reduce**
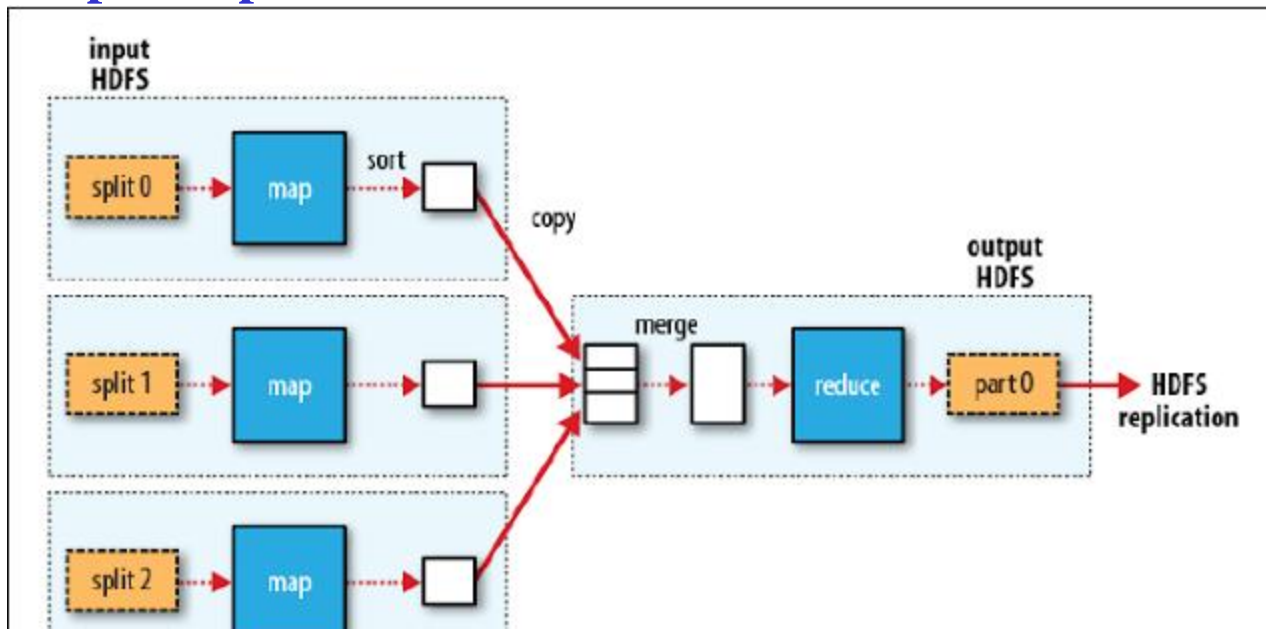
Multiple map tasks and multiple reduce tasks

- When there are multiple reducers, the map tasks partition their output, each creating one partition for each reduce task. There can be many keys (and their associated values) in each partition, but the records for any given key are all in a single partition
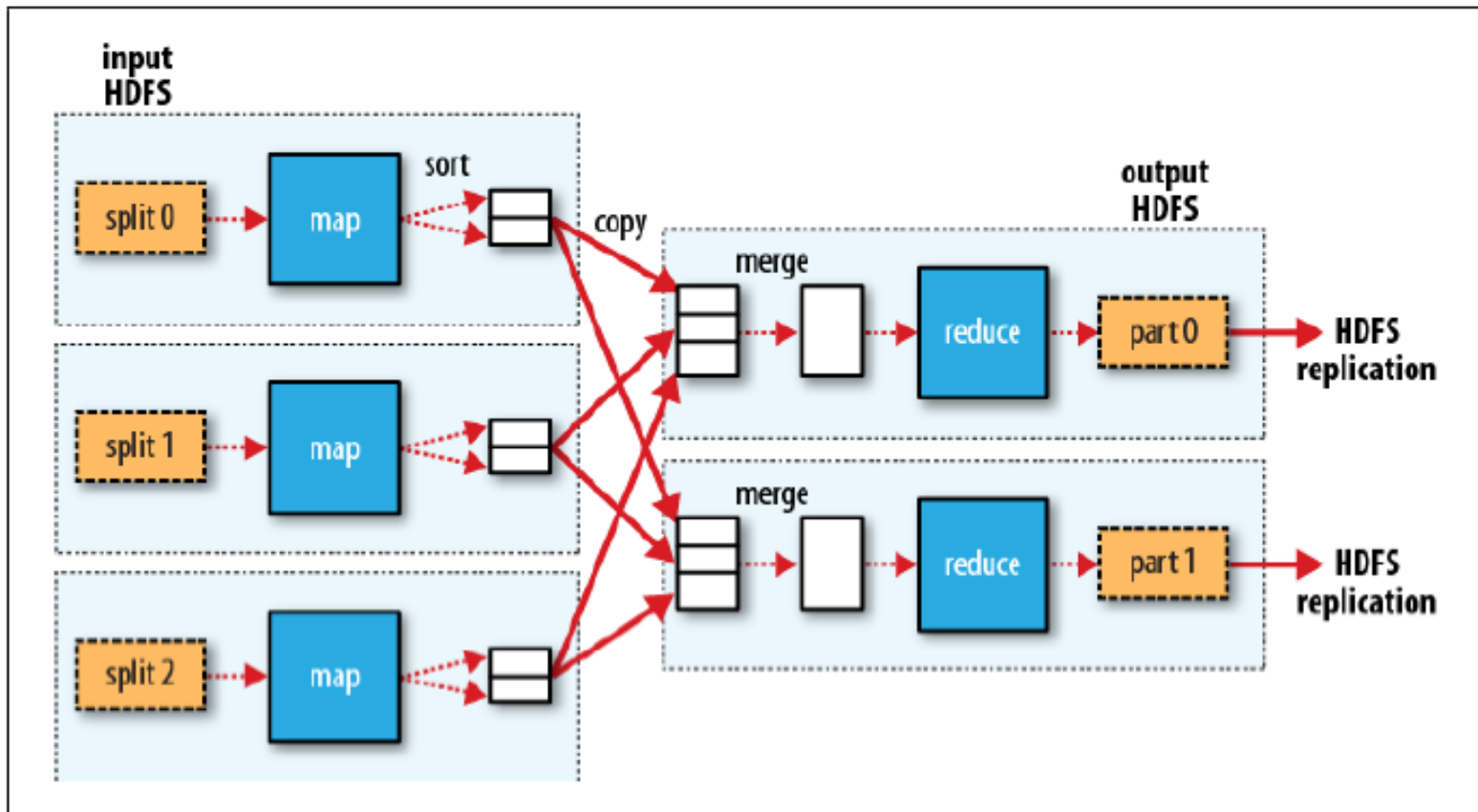
MapReduce: Fault Tolerance

- **Handled via re-execution of tasks.**
  - Task completion committed through master
- **Mappers save outputs to local disk before serving to reducers**
  - Allows recovery if a reducer crashes
  - Allows running more reducers than # of nodes
- **If a task crashes:**
  - Retry on another node
    » OK for a map because it had no dependencies
    » OK for reduce because map outputs are on disk
  - If the same task repeatedly fails, fail the job or ignore that input block
  - : For the fault tolerance to work, *user tasks must be deterministic and side-effect-free*

2. If a node crashes:
  - Relaunch its current tasks on other nodes
  - Relaunch any maps the node previously ran
   » Necessary because their output files were lost along with the crashed node

MapReduce: Redundant Execution

- Slow workers are source of bottleneck, may delay completion time.

- spawn backup tasks, one to finish first wins.

- Effectively utilizes computing power, reducing job completion time by a factor.
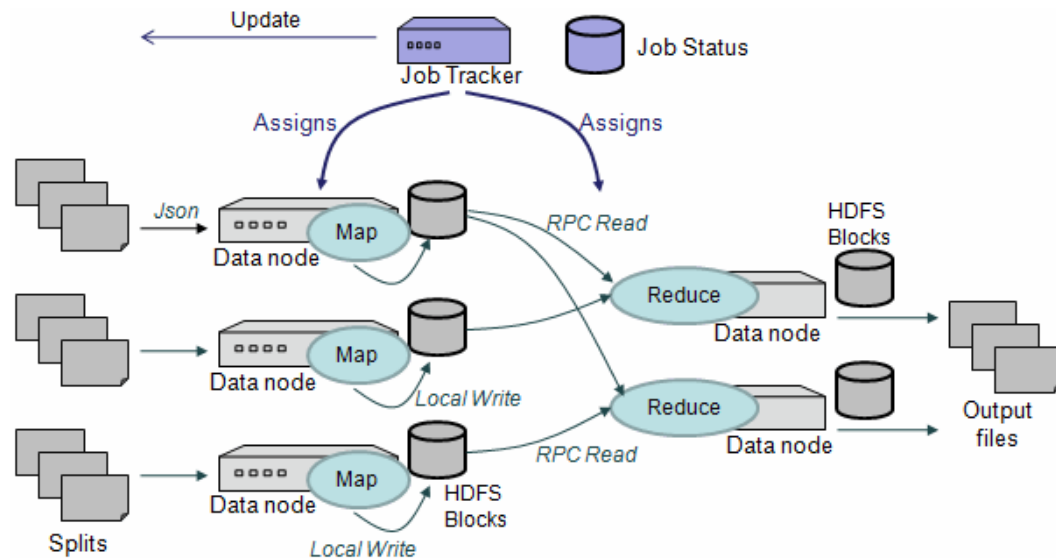
User Code Optimization: Combining Phase

- Run on map machines after map phase
  - "Mini-reduce," only on local map output
  - E.g. job.setCombinerClass(Reduce.class);
- save bandwidth before sending data to full reduce tasks
- Requirement: commutative & associative

On one mapper machine:

Map output

Combiner replaces with:

To reducer   To reducer
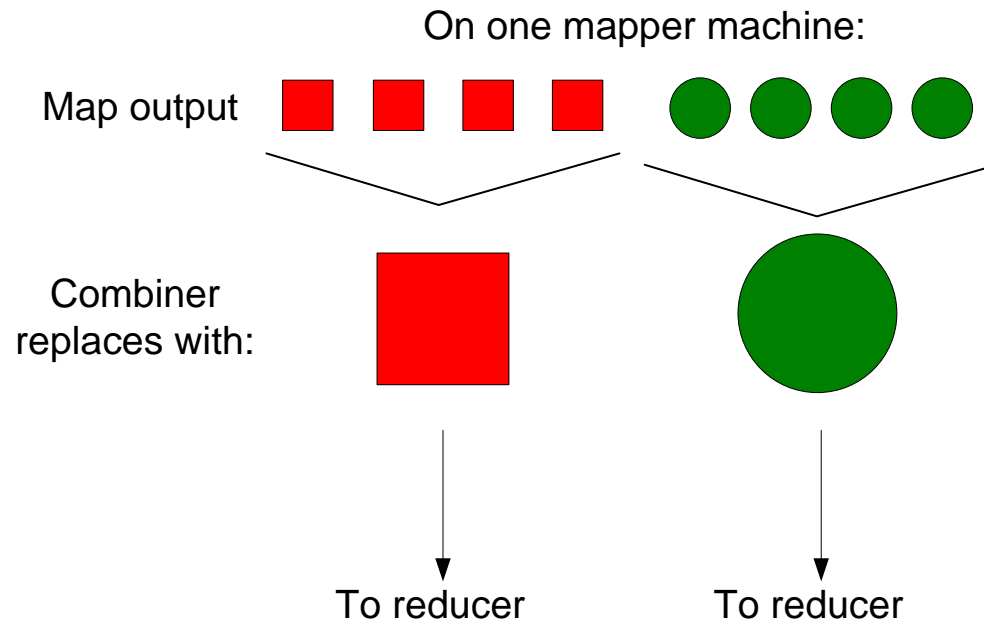
MapReduce Applications (I)

- **Distributed grep (search for words)**
  - *Map*: emit a line if it matches a given pattern

- **URL access frequency**
  - *Map*: process logs of web page access; output
  - *Reduce*: add all values for the same URL

MapReduce Applications (II)

- **Reverse web-link graph**
  - *Map*: Input is node-outgoing links. Output each link with the target as a key.
  - *Reduce*: Concatenate the list of all source nodes associated with a target.
- **Inverted index**
  - *Map*: Input is words for a document. Emit word-document pairs
  - *Reduce*: for the same word, sort the document IDs that contain this word; emits a pair.

Types of MapReduce Applications

- **Map only parallel processing**
  - Count word usage for each document
- **Map-reduce two-stage processing**
  - Count word usage for the entire document collection
- **Multiple map-reduce stages**

    1. Count word usage in a document set

    2. Identify most frequent words in each document, but exclude those most popular words in the entire document set

MapReduce Job Chaining

- Run a sequence of map-reduce jobs
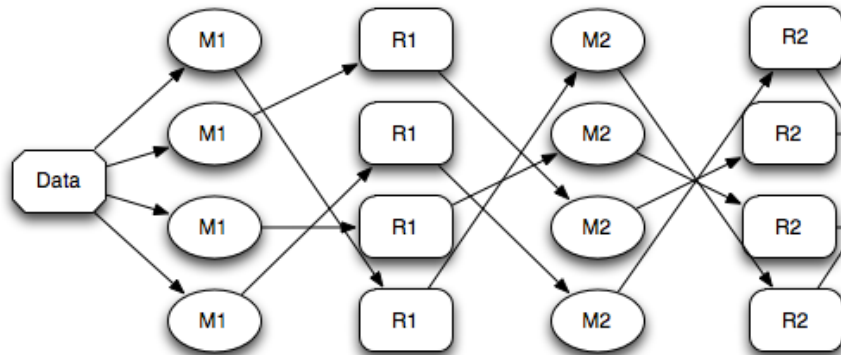
- Use job.waitForComplete()
  - Define the first job including input/output directories, and map/combiner/reduce classes.
    - » Run the first job with job.waitForComplete()
  - Define the second job
    - » Run the second job with job.waitForComplete()
- Use JobClient.runJob(job)

Example

```
Job job = new Job(conf, "word count"); //mapreduce job
    job.setJarByClass(WordCount.class); //set jar file
    job.setMapperClass(TokenizerMapper.class); // set mapper class
    ...
    FileInputFormat.addInputPath(job, new Path(otherArgs[0]));   // input path
    FileOutputFormat.setOutputPath(job, new Path(otherArgs[1])); // output path
    job.waitForCompletion(true) ;
Job job1 = new Job(conf, "word count"); //mapreduce job
    job1.setJarByClass(WordCount.class); //set jar file
    job1. setMapperClass(TokenizerMapper.class); // set mapper class
    ...
    FileInputFormat.addInputPath(job1, new Path(otherArgs[1]));   // input path
    FileOutputFormat.setOutputPath(job1, new Path(otherArgs[2])); // output path
     System.exit(job1.waitForCompletion(true) ? 0 : 1);  //exit status
}
```

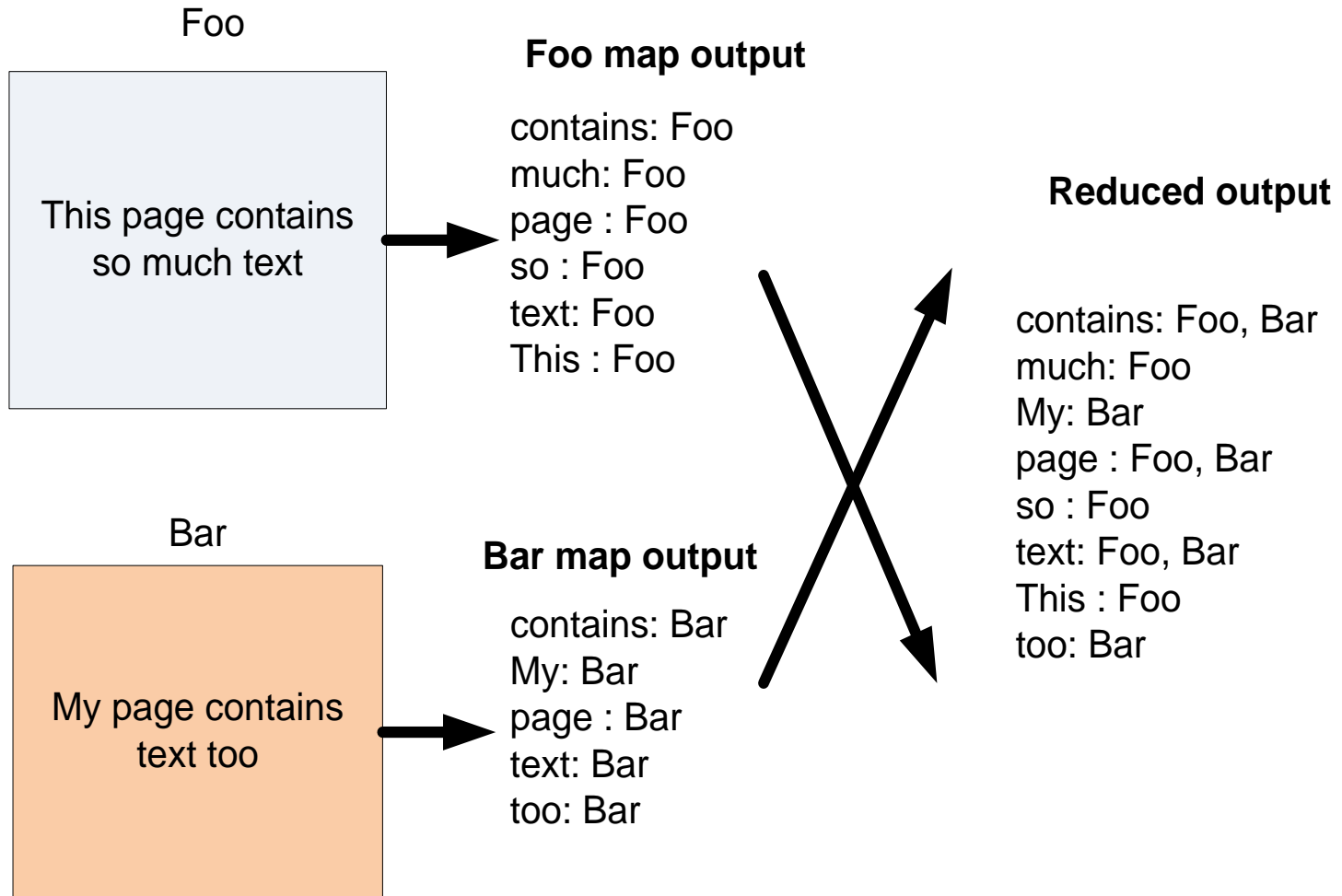# MapReduce Use Case: Inverted Indexing Preliminaries

Construction of inverted lists for document search

- Input: documents: (docid, [term, term..]), (docid, [term, ..]), ..

- Output: (term, [docid, docid, …])

  - E.g., (apple, [1, 23, 49, 127, …])

A document id is an <u>internal document id</u>, e.g., a unique integer

- <u>Not</u> an external document id such as a url

Inverted Indexing: Data flow

Foo

```
This page contains
so much text
```

Foo map output

contains: Foo
much: Foo
page : Foo
so : Foo
text: Foo
This : Foo

Reduced output

contains: Foo, Bar
much: Foo
My: Bar
page : Foo, Bar
so : Foo
text: Foo, Bar
This : Foo
too: Bar

Bar

```
My page contains
text too
```

Bar map output

contains: Bar
My: Bar
page : Bar
text: Bar
too: Bar

Using MapReduce to Construct Inverted Indexes

- **Each Map task is a document parser**
  - Input: A stream of documents
  - Output: A stream of (term, docid) tuples
    - » (long, 1) (ago, 1) (and, 1) … (once, 2) (upon, 2) …
    - » We may create internal IDs for words.
- **Shuffle sorts tuples by key and routes tuples to Reducers**
- **Reducers convert streams of keys into streams of inverted lists**
  - Input: (long, 1) (long, 127) (long, 49) (long, 23) …
  - The reducer sorts the values for a key and builds an inverted list
  - Output: (long, [frequency:492, docids:1, 23, 49, 127, …])

Using Combiner () to Reduce Communication

- **Map:** $(docid_1, content_1) \rightarrow (t_1, ilist_{1,1}) (t_2, ilist_{2,1}) (t_3, ilist_{3,1})$ …
    – Each output inverted list covers just <u>one document</u>

- **Combine locally**

    Sort by t

    Combiner: $(t_1 [ilist_{1,2}\ ilist_{1,3}\ ilist_{1,1}\ …]) \rightarrow (t_1, ilist_{1,27})$
    – Each output inverted list covers a <u>sequence of documents</u>

- **Shuffle and sort** by t

    $(t_4, ilist_{4,1}) (t_5, ilist_{5,3})\ … \rightarrow (t_4, ilist_{4,2}) (t_4, ilist_{4,4}) (t_4, ilist_{4,1})$ …

- **Reduce:** $(t_7, [ilist_{7,2}, ilist_{3,1}, ilist_{7,4}, …]) \rightarrow (t_7, ilist_{final})$

$ilist_{i,j}$:   the j'th inverted list fragment for term i

Hadoop and Tools

- **Various Linux Hadoop clusters**
  - Cluster +Hadoop: http://hadoop.apache.org
  - Amazon EC2
- **Windows and other platforms**
  - The NetBeans plugin simulates Hadoop
  - The workflow view works on Windows
- **Hadoop-based tools**
  - For Developing in Java, NetBeans plugin
- **Pig Latin,** a SQL-like high level data processing script language
- **Hive,** Data warehouse, SQL
- **HBase,** Distributed data store as a large table

New Hadoop Develpment