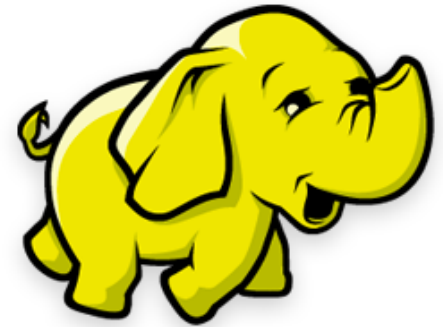


# Apache Hadoop and MapReduce

COMP9313: Big Data Management

Thanks to Dr. Xin Cao for sharing useful materials for  
the preparation of this course

# Hadoop



[source](#)

- Open-source data storage and processing platform
- Before the advent of Hadoop, storage and processing of big data was a big challenge
- Massively scalable, automatically parallelizable
  - Based on work from Google
    - Google: GFS + MapReduce + BigTable (Not open)
    - Hadoop: HDFS + Hadoop MapReduce + Hbase (opensource)
- Named by Doug Cutting in 2006 (worked at Yahoo! at that time), after his son's toy elephant

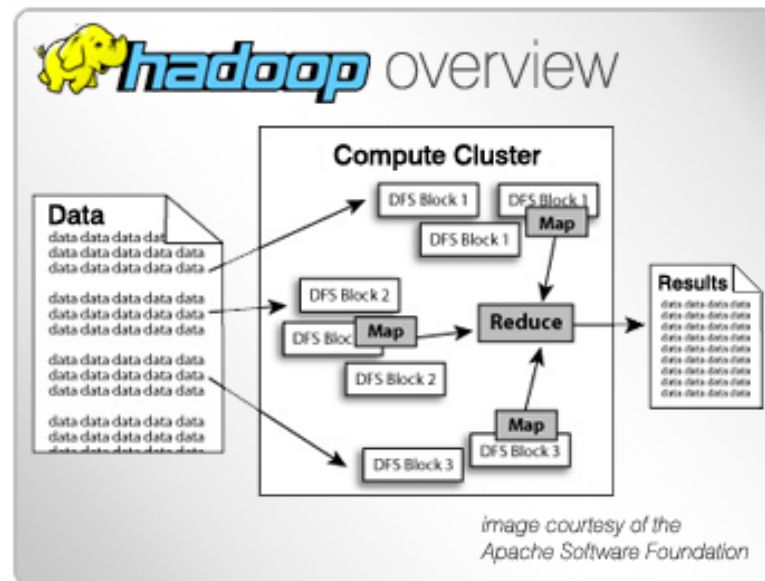
# What's offered by Hadoop?

- Redundant, fault-tolerant data storage
- Parallel computation framework
- Job coordination
- Programmers do not need to worry about:
  - Where are files located?
  - How to handle failures and data loss?
  - How to distribute computation?
  - How to program for scaling?



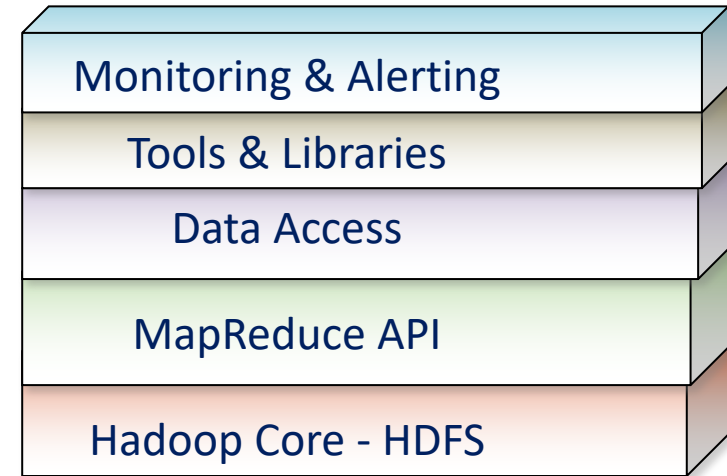
# Why use Hadoop?

- Cheap
  - Scales to Petabytes or more easily
- Fast
  - Parallel data processing
- Suitable
  - ... for particular types of big data problems

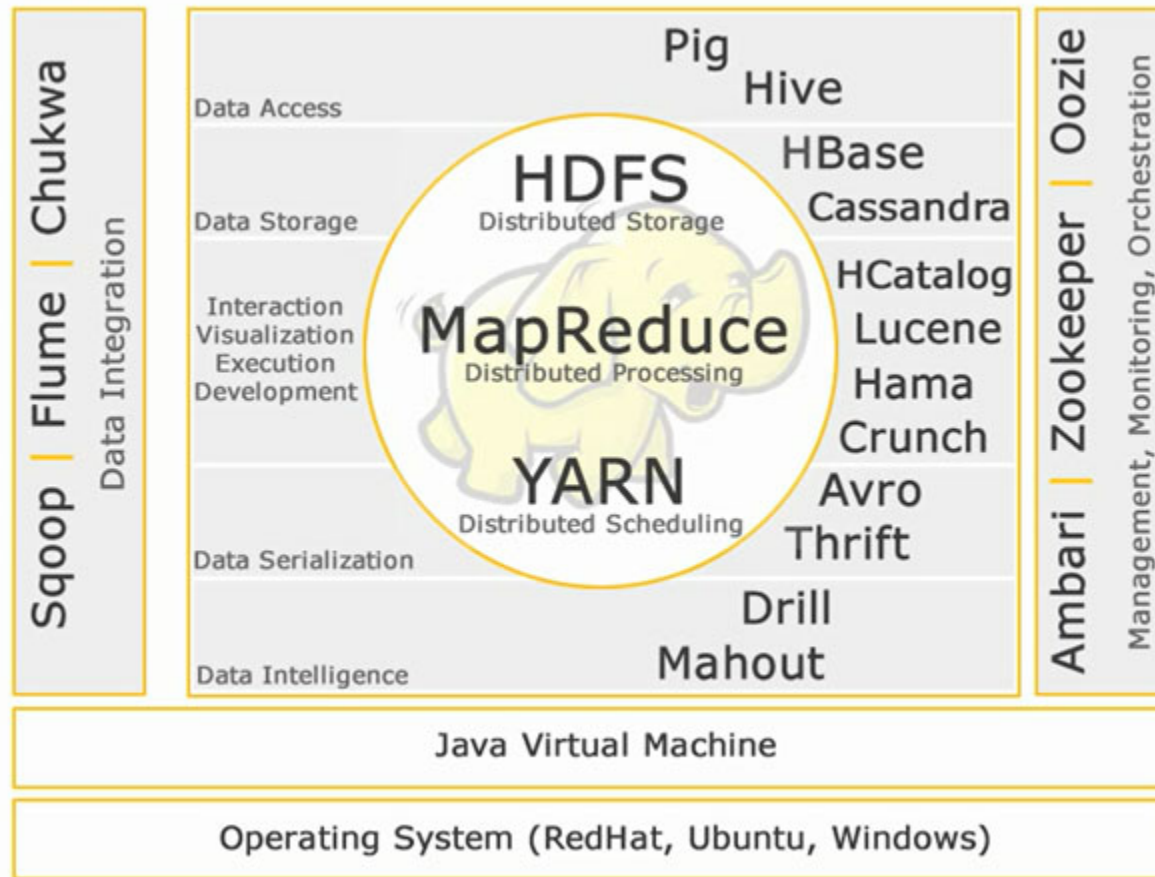


# Hadoop is a set of Apache Frameworks...

- Data storage (HDFS)
  - Runs on commodity hardware
  - Horizontally scalable
- Processing (MapReduce)
  - Parallelized (scalable) processing
  - Fault Tolerant
- Other Tools / Frameworks
  - Data Access
    - Hbase (column store), Hive (Data warehousing), Pig (high-level language on top of Hadoop), Mahout (library for ML / Data Analytics)
  - Tools
    - Hue (SQL Cloud editor), Sqoop (data transfer Hadoop / Structured stores)

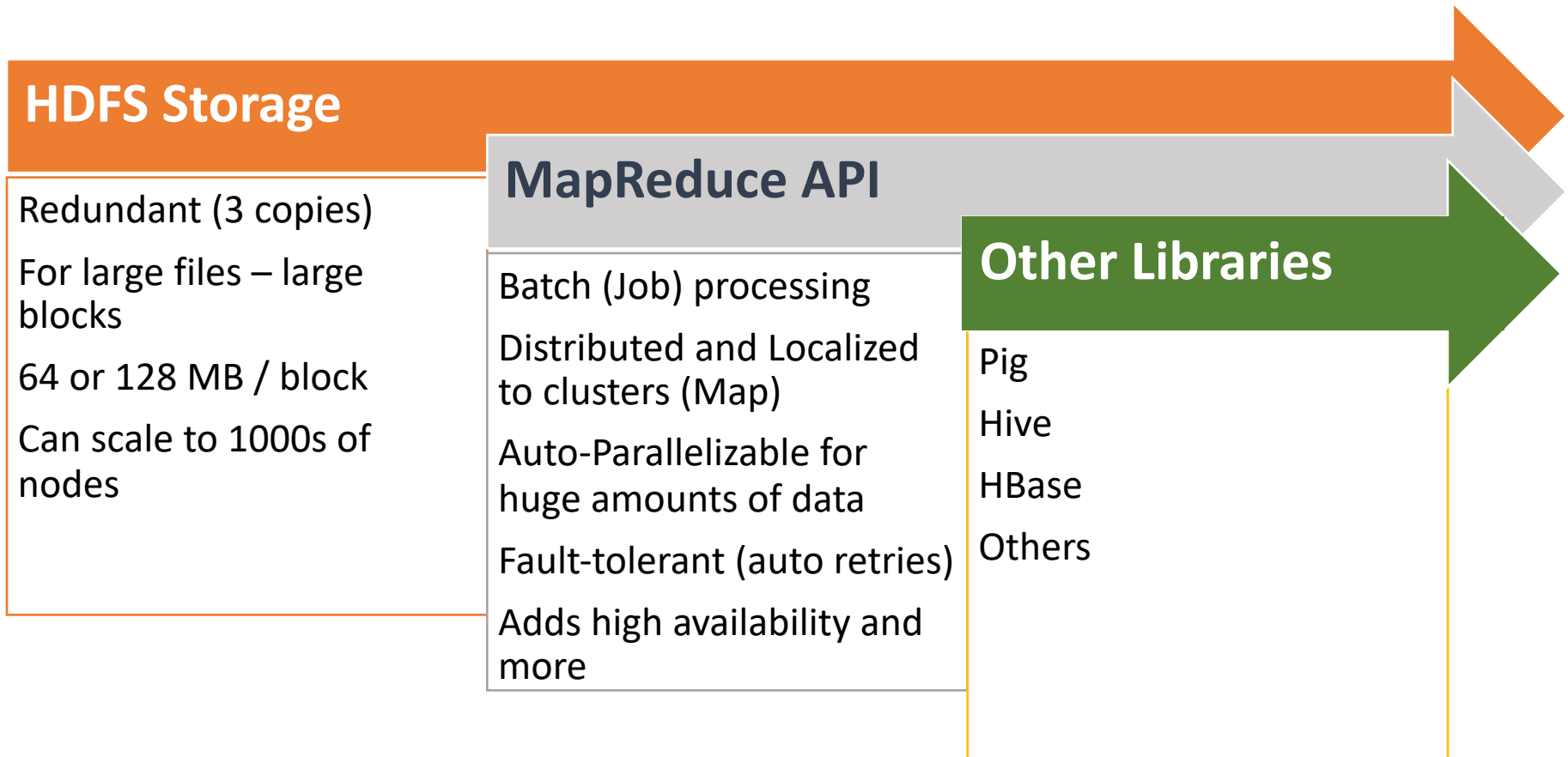


# Hadoop ecosystem



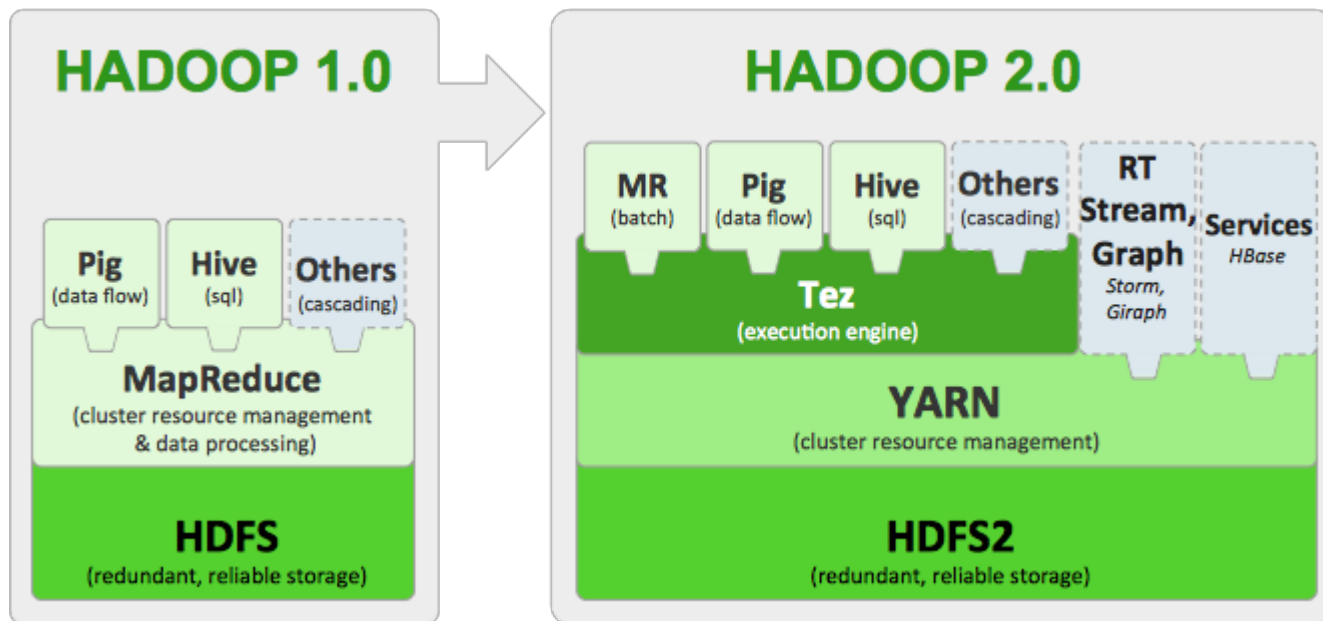
[source](#)

# Core Parts of a Hadoop Distribution



# Hadoop 1.0 vs Hadoop 2.0

- Single Use System
  - Batch apps
- Multi-Purpose Platform
  - Batch, Interactive, Online, Streaming

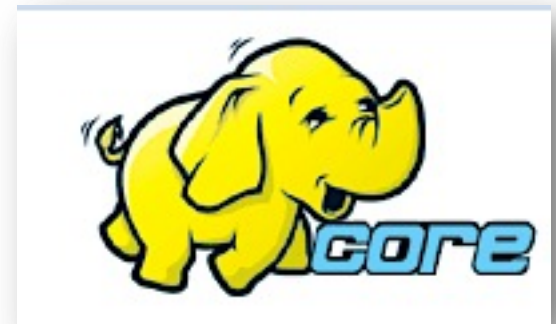


***Hadoop YARN*** (Yet Another Resource Negotiator): a resource-management platform responsible for managing computing resources in clusters and using them for scheduling of users' applications

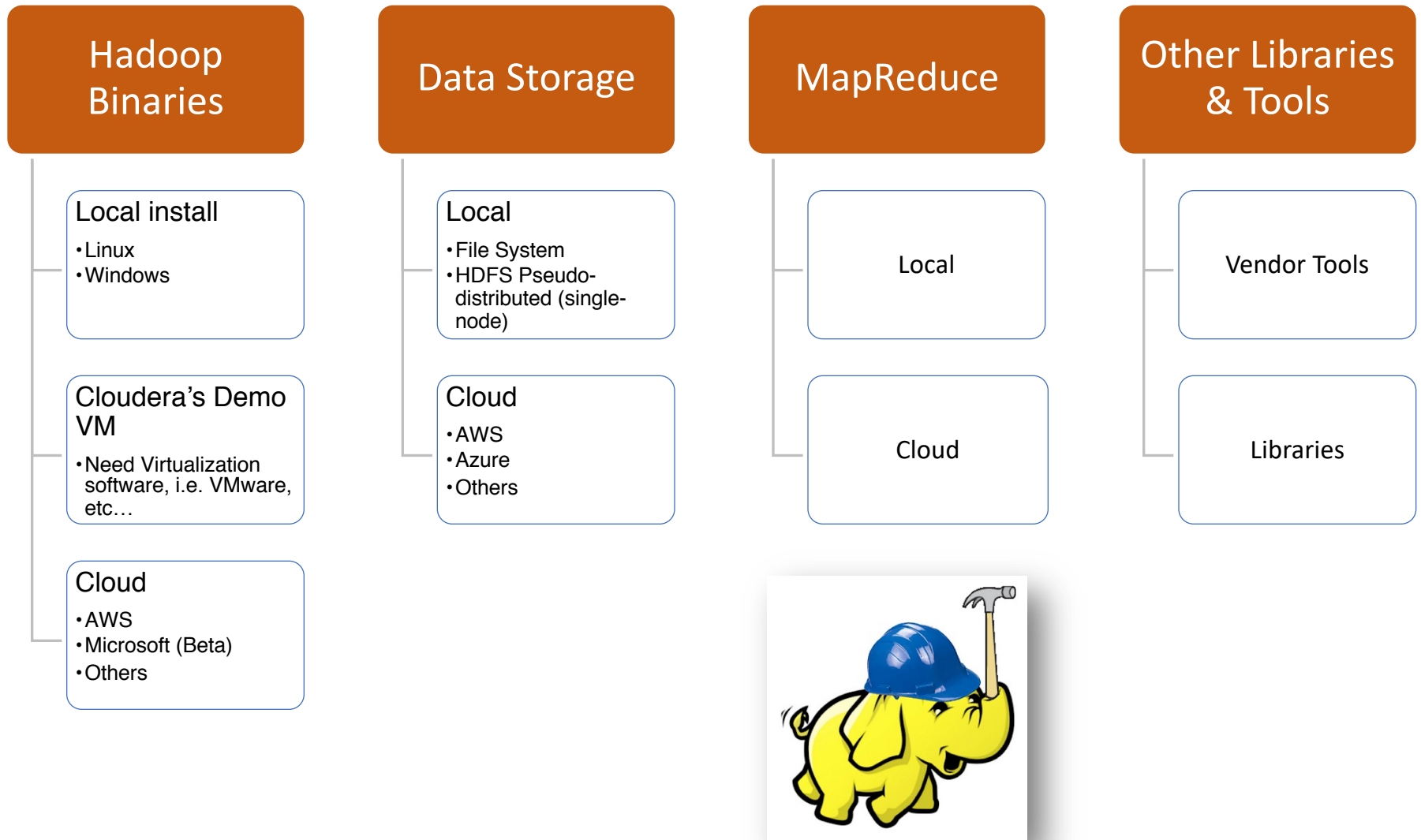


# Common Hadoop Distributions

- Open Source
  - Apache
- Commercial
  - Cloudera
  - Hortonworks
  - MapR
  - AWS MapReduce
  - Microsoft Azure HDInsight (Beta)

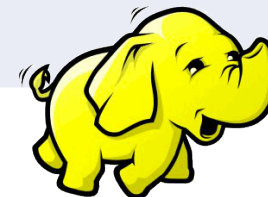
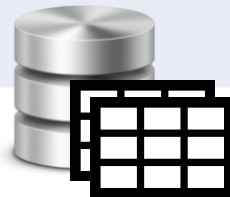


# Setting up for Hadoop Development

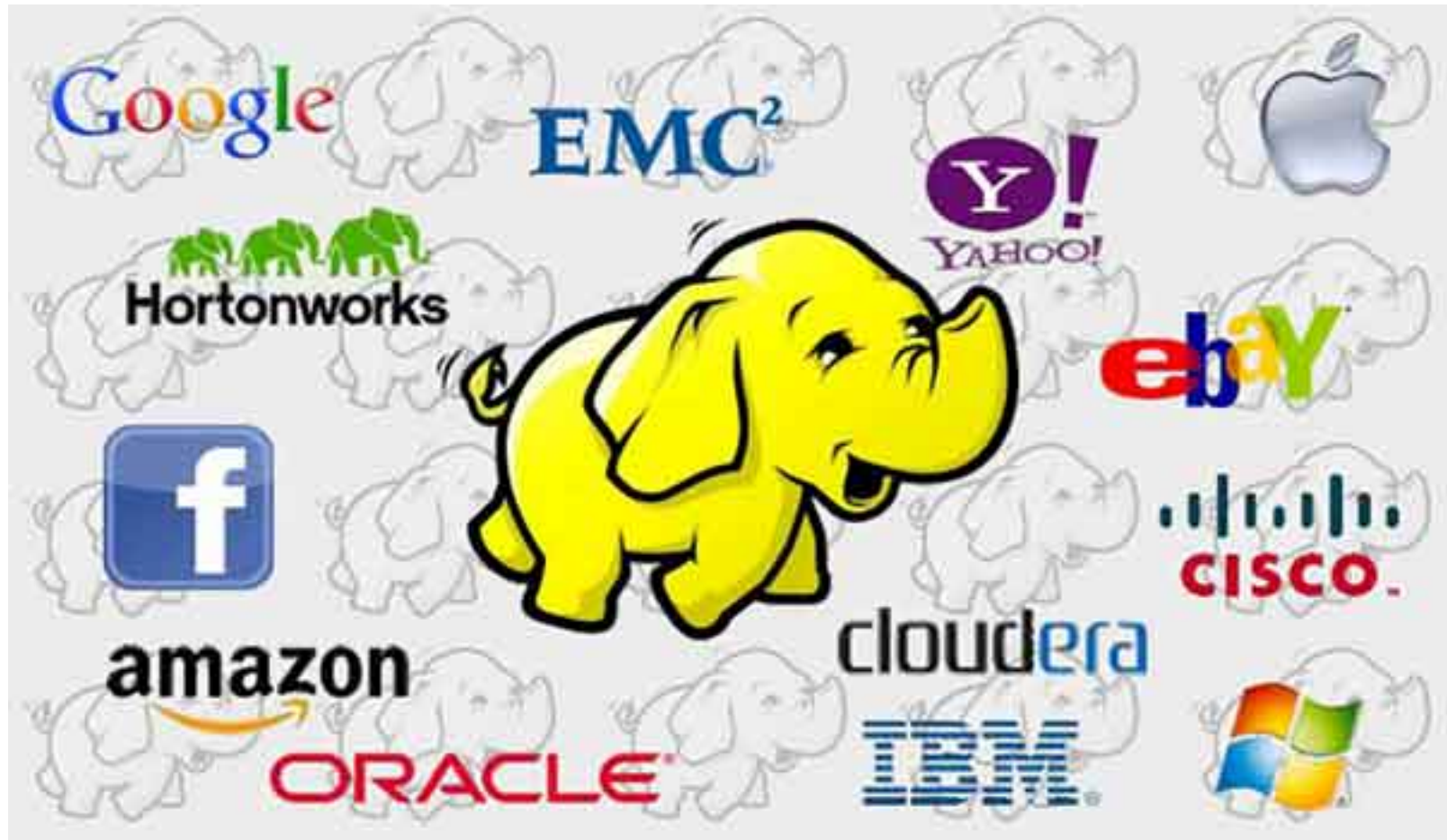


# RDMS vs Hadoop

| Feature          | RDBMS                                | Hadoop  |
|------------------|--------------------------------------|---|
| Data variety     | Mainly structured data               | Structured, semi-structured and unstructured data |
| Data storage     | Average size data (GBs)              | Used for large datasets (TBs, PTs)                |
| Querying         | SQL                                  | HQL (Hive Query Language)                         |
| Schema           | Required on write (static schema)    | Required on read (dynamic schema)                 |
| Speed            | Read are fast                        | Both read and write are fast                      |
| Use case         | OLTP (Online Transaction Processing) | Analytics (audio, video, logs), data discovery    |
| Data objects     | Works on relational tables           | Works on key/value pairs                          |
| Scalability      | Vertical                             | Horizontal  |
| Hardware profile | High-end servers                     | Commodity / Utility hardware                      |
| Integrity        | ACID                                 | Low   |



# Companies using Hadoop



[source](#)

# MapReduce

# What is MapReduce?

- Originated from Google, [OSDI'04]
  - MapReduce: Simplified Data Processing on Large Clusters
  - Jeffrey Dean and Sanjay Ghemawat ([paper](#))
- Programming model for parallel data processing
- Hadoop can run MapReduce programs written in various languages:  
e.g. Java, Ruby, Python, C++
- For large-scale data processing
  - Exploits large set of commodity computers
  - Executes process in distributed manner
  - Offers high availability

# Motivation for MapReduce (1)

- Typical big data problem challenges:
  - How do we break up a large problem into smaller tasks that can be executed in **parallel**?
  - How do we assign tasks to workers distributed across a potentially **large number** of machines?
  - How do we ensure that the workers get the **data** they need?
  - How do we coordinate **synchronization** among the different workers?
  - How do we **share** partial results from one worker that is needed by another?
  - How do we accomplish all of the above in the face of software **errors** and hardware **faults**?

# Motivation for MapReduce (2)

- There was need for an abstraction that hides many system-level details from the programmer.
- MapReduce addresses this challenge by providing a simple abstraction for the developer, transparently handling most of the details behind the scenes in a **scalable**, **robust**, and **efficient** manner.
- MapReduce separates the **what** from the **how**



# Typical Big Data Problem

- Iterate over a large number of records
- Extract something of interest from each **Map**
- Shuffle and sort intermediate results
- Aggregate intermediate results
- Generate final output **Reduce**

Key idea:

Provide a **functional abstraction** for these two operations

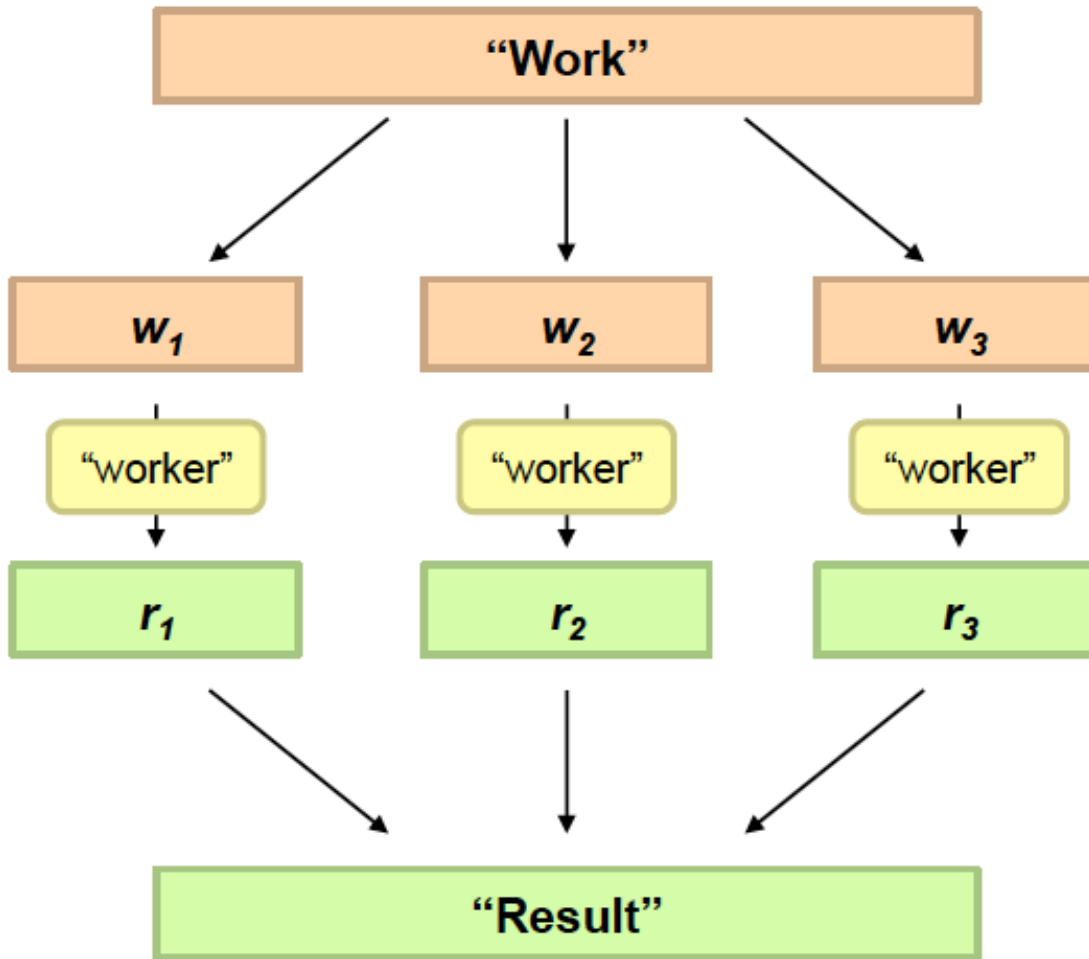
# The Idea of MapReduce (1)

- Inspired by the map and reduce functions in functional programming
- We can view map as a transformation over a dataset
  - This transformation is specified by the function  $f$
  - Each functional application happens in **isolation**
  - The application of  $f$  to each element of a dataset can be parallelized in a straightforward manner
- We can view reduce as an aggregation operation
  - The aggregation is defined by the function  $g$
  - Data locality: elements in the list must be “brought together”
  - If we can group elements of the list, also the reduce phase can proceed in parallel
- The framework coordinates the map and reduce phases:
  - Grouping intermediate results happens in parallel

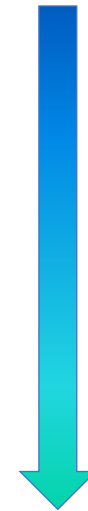
# The Idea of MapReduce (2)

- Handles scheduling
  - Assigns workers to map and reduce tasks
- Handles “data distribution”
  - Moves processes to data
- Handles synchronization
  - Gathers, sorts, and shuffles intermediate data
- Handles errors and faults
  - Detects worker failures and restarts
- Everything happens on top of a distributed file system (HDFS)
- You don’t know:
  - Where mappers and reducers run
  - When a mapper or reducer begins or finishes
  - Which input a particular mapper is processing
  - Which intermediate key a particular reducer is processing

# Philosophy to Scale for Big Data Processing

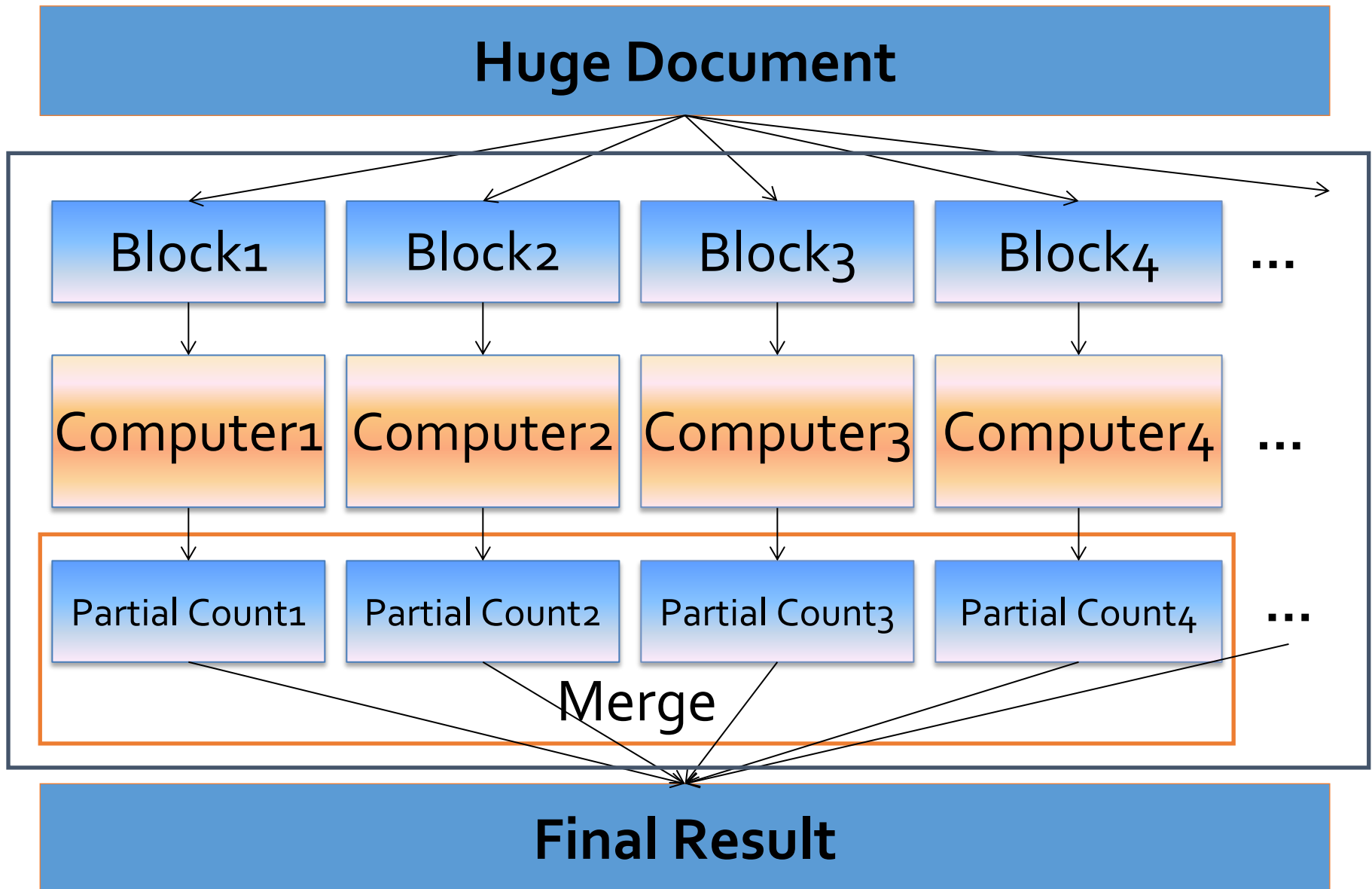


**Divide Work**



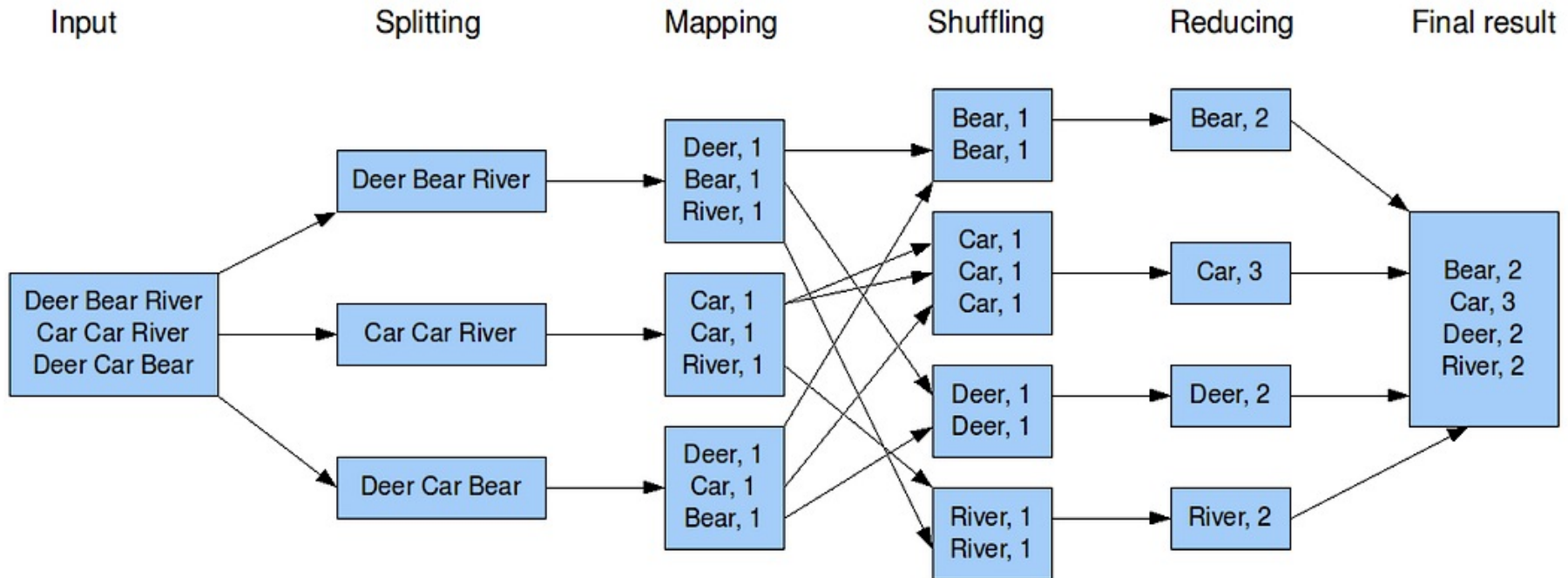
**Combine Results**

# Distributed Word Count



# MapReduce Example - WordCount

The overall MapReduce word count process



- Hadoop MapReduce is an implementation of MapReduce
  - MapReduce is a computing paradigm (Google)
  - Hadoop MapReduce is an open-source software

# Data Structures in MapReduce

- Key-value pairs are the basic data structure in MapReduce
  - Keys and values can be: integers, float, strings, raw bytes, etc.
  - They can also be arbitrary data structures
- The design of MapReduce algorithms involves:
  - Imposing the key-value structure on arbitrary datasets
    - E.g.: for a collection of Web pages, input keys may be URLs and values may be the HTML content
  - In some algorithms, input keys uniquely identify a record
  - Keys can be combined in complex ways to design various algorithms

# Map and Reduce Functions

- Programmers specify two functions:
  - **map** ( $k_1, v_1$ )  $\rightarrow$  list [ $\langle k_2, v_2 \rangle$ ]
    - Map transforms the input into key-value pairs to process
  - **reduce** ( $k_2, \text{list } [v_2]$ )  $\rightarrow$  [ $\langle k_3, v_3 \rangle$ ]
    - Reduce aggregates the list of values for each key
    - All values with the same key are sent to the same reducer
  - list [ $\langle k_2, v_2 \rangle$ ] will be grouped according to key  $k_2$  as ( $k_2, \text{list } [v_2]$ )
- The MapReduce environment takes in charge of everything else...
- A complex program can be decomposed as a succession of Map and Reduce tasks

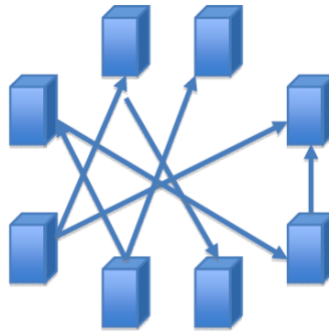


# Understanding MapReduce

## • Map>>

- $(K1, V1) \rightarrow$ 
  - Info in
  - Input Split
- $list(K2, V2)$ 
  - Key / Value out (intermediate values)
  - One list per local node
  - Can implement local Reducer (or Combiner)

## ■ Shuffle/Sort>>



## ■ Reduce

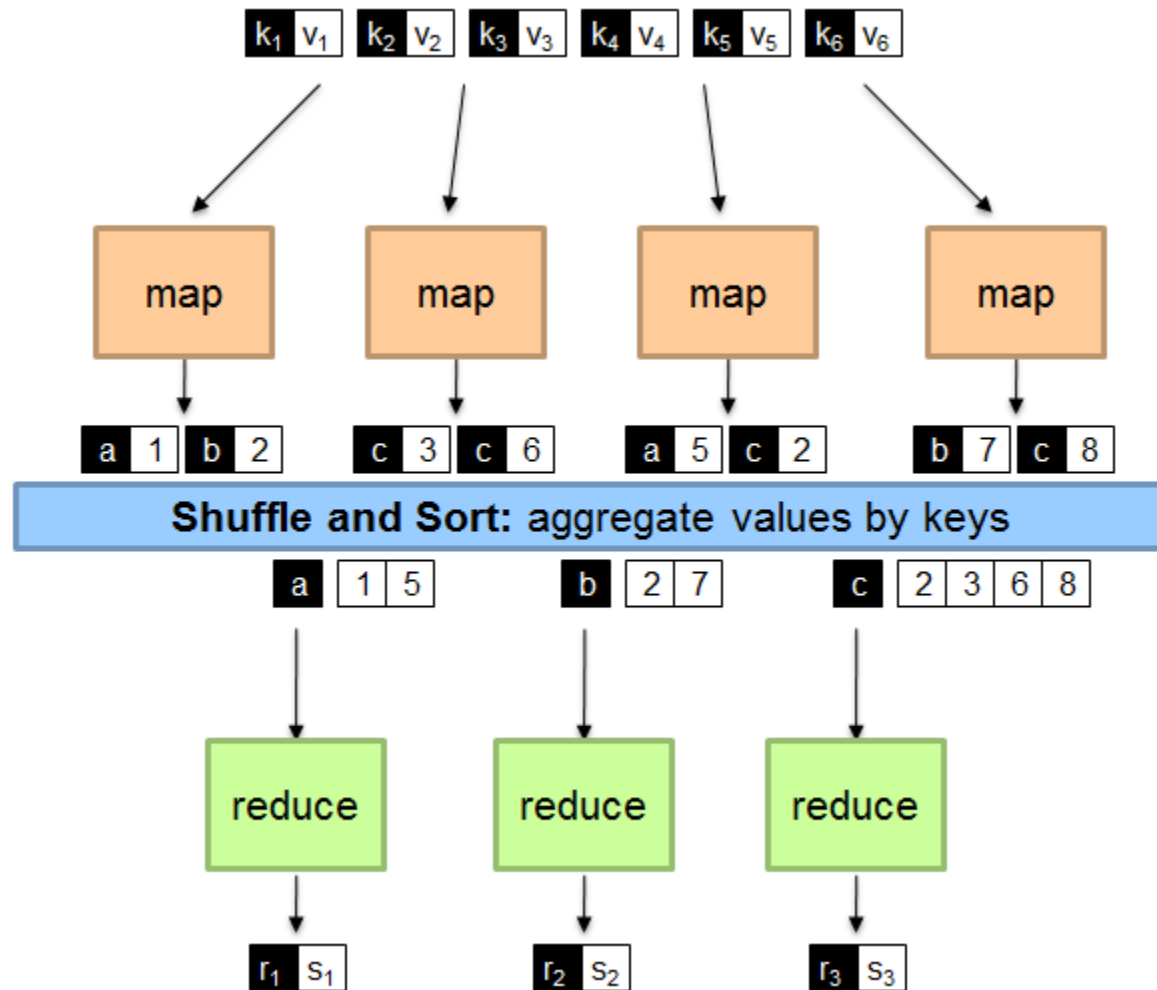
- $(K2, list(V2)) \rightarrow$ 
  - Shuffle / Sort phase precedes Reduce phase
  - Combines Map output into a list
- $list(K3, V3)$ 
  - Usually aggregates intermediate values

$(input) \langle k1, v1 \rangle \rightarrow \text{map} \rightarrow \langle k2, v2 \rangle \rightarrow \text{shuffle/sort} \rightarrow \langle k2, list(V2) \rangle \rightarrow \text{reduce} \rightarrow \langle k3, v3 \rangle (output)$

# WordCount - Mapper

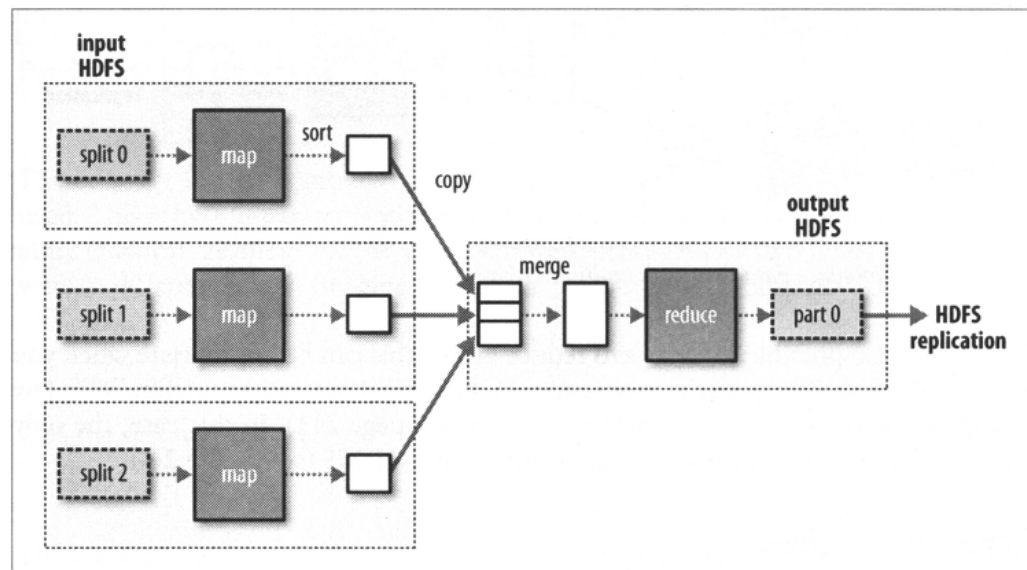
- Let's count number of each word in documents (e.g., Tweets/Blogs)
  - Reads input pair  $\langle k1, v1 \rangle$ 
    - The input to the mapper is in format of  $\langle docID, docText \rangle$ :  
 $\langle D1, "Hello World" \rangle, \langle D2, "Hello Hadoop Bye Hadoop" \rangle$
  - Outputs pairs  $\langle k2, v2 \rangle$ 
    - The output of the mapper is in format of  $\langle term, 1 \rangle$ :  
 $\langle Hello, 1 \rangle \langle World, 1 \rangle \langle Hello, 1 \rangle \langle Hadoop, 1 \rangle \langle Bye, 1 \rangle \langle Hadoop, 1 \rangle$
  - After shuffling and sort, reducer receives  $\langle k2, list(v2) \rangle$   
 $\langle Hello, \{1, 1\} \rangle \langle World, \{1\} \rangle \langle Hadoop, \{1, 1\} \rangle \langle Bye, \{1\} \rangle$
  - The output is in format of  $\langle k3, v3 \rangle$ :  
 $\langle Hello, 2 \rangle \langle World, 1 \rangle \langle Hadoop, 2 \rangle \langle Bye, 1 \rangle$

# WordCount - Mapper



# Shuffle and Sort

- Shuffle
  - Input to the Reducer is the sorted output of the mappers. In this phase the framework fetches the relevant partition of the output of all the mappers, via HTTP.
- Sort
  - The framework groups Reducer inputs by keys (since different Mappers may have output the same key) in this stage.
- Hadoop framework handles the Shuffle and Sort step .



# Word Count MapReduce in Java

# MapReduce Program

- A MapReduce program consists of the following 3 parts:
  - Driver → main (would trigger the map and reduce methods)
  - Mapper
  - Reducer
  - Include the map reduce and main methods in 3 different classes
- Check detailed information of all classes at:  
<https://hadoop.apache.org/docs/r2.7.2/api/allclasses-noframe.html>

# Mapper

```
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);
            }
        }
    }
```

# Mapper Explanation

- Maps input key/value pairs to a set of intermediate key/value pairs.

//Map class header

**public static class** TokenizerMapper

**extends** Mapper<Object, Text, Text, IntWritable>{

➤ Class Mapper<KEYIN,VALUEIN,KEYOUT,VALUEOUT>

- KEYIN,VALUEIN -> (k1, v1) -> (docid, doc)

- KEYOUT,VALUEOUT ->(k2, v2) -> (word, 1)

// IntWritable: A serializable and comparable object for integer

**private final static** IntWritable one = new IntWritable(1);

/\* Text: stores text using standard UTF8 encoding. It provides methods to serialize, deserialize, and compare texts at byte level\*/

**private** Text word = new Text();

/\* hadoop supported data types for the key/value pairs, in package org.apache.Hadoop \*/



# What's Writable?

- Hadoop defines its own “box” classes for strings (Text), integers (IntWritable), etc.
- All values must implement interface Writable
- All keys must implement interface WritableComparable
- Writable is a serializable object which implements a simple, efficient, serialization protocol

# Writable Wrappers

| Java primitive | Writable implementation       |
|----------------|-------------------------------|
| boolean        | BooleanWritable               |
| byte           | ByteWritable                  |
| short          | ShortWritable                 |
| int            | IntWritable<br>VIntWritable   |
| float          | FloatWritable                 |
| long           | LongWritable<br>VLongWritable |
| double         | DoubleWritable                |

| Java class  | Writable implementation |
|-------------|-------------------------|
| String      | Text                    |
| byte[]      | BytesWritable           |
| Object      | ObjectWritable          |
| <i>null</i> | NullWritable            |

| Java collection | Writable implementation                                      |
|-----------------|--|
| <i>array</i>    | ArrayWritable<br>ArrayPrimitiveWritable<br>TwoDArrayWritable |
| Map             | MapWritable  |
| SortedMap       | SortedMapWritable  |
| <i>enum</i>     | EnumSetWritable  |

# Mapper Explanation (Cont'd)

## //Map method header

public void map(Object key, Text value, Context context) throws IOException, InterruptedException

- Object key/Text value: Data type of the input Key and value to the mapper
- Context: An inner class of Mapper, used to store the context of a running task. Here it is used to collect data output by either the Mapper or the Reducer, i.e. intermediate outputs or the output of the job
- Exceptions: IOException, InterruptedException
- This function is called once for each key/value pair in the input split. Your application should override this to do your job.

# Mapper Explanation (Cont'd)

//Use a string tokenizer to split the document into words

```
StringTokenizer itr = new StringTokenizer(value.toString());
```

//Iterate through each word and a form key value pairs

```
while (itr.hasMoreTokens()) {
```

//Assign each word from the tokenizer(of String type) to a Text 'word'

```
    word.set(itr.nextToken());
```

//Form key value pairs for each word as <word, one> using context

```
    context.write(word, one);
```

```
}
```

- Map function produces Map.Context object
  - Map.context() takes  $(k, v)$  elements
- Any (WritableComparable, Writable) can be used

# Reducer

```
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);
        }
    }
```

# Reducer Explanation

//Reduce Header similar to the one in map with different key/value data type

```
public static class IntSumReducer
```

```
    extends Reducer<Text, IntWritable, Text, IntWritable>
```

//data from map will be <"word",{1,1,..}>, so we get it with an Iterator and thus we can go through the sets of values

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

//Initialize a variable 'sum' as 0

```
    int sum = 0;
```

//Iterate through all the values with respect to a key and sum up all of them

```
    for (IntWritable val : values) {
```

```
        sum += val.get();
```

```
    }
```

// Form the final key/value pairs results for each word using context

```
    result.set(sum);
```

```
    context.write(key, result);
```

# Main (Driver)

```
public static void main(String[] args) throws Exception {  
    Configuration conf = new Configuration();  
    Job job = Job.getInstance(conf, "word count");  
    job.setJarByClass(WordCount.class);  
    job.setMapperClass(TokenizerMapper.class);  
    job.setReducerClass(IntSumReducer.class);  
    job.setOutputKeyClass(Text.class);  
    job.setOutputValueClass(IntWritable.class);  
    FileInputFormat.addInputPath(job, new Path(args[0]));  
    FileOutputFormat.setOutputPath(job, new Path(args[1]));  
    System.exit(job.waitForCompletion(true) ? 0 : 1);  
}
```

# Main (Driver)

- Given the Mapper and Reducer code, the main() method starts the MapReduction running
- The Hadoop system picks up a bunch of values from the command line on its own
- Then the main() also specifies a few key parameters of the problem in the Job object
- Job is the primary interface for a user to describe a map-reduce job to the Hadoop framework for execution (such as what Map and Reduce classes to use and the format of the input and output files)
- Other parameters, i.e. the number of machines to use, are optional and the system will determine good values for them if not specified
- Then the framework tries to faithfully execute the job as described by the job object



# Main Explanation

//Creating a Configuration object and a Job object, assigning a job name for identification purposes

```
Configuration conf = new Configuration();
```

```
Job job = Job.getInstance(conf, "word count");
```

- Job Class: It allows the user to configure the job, submit it, control its execution, and query the state. Normally the user creates the application, describes various facets of the job via [Job](#) and then submits the job and monitor its progress.

//Setting the job's jar file by finding the provided class location

```
job.setJarByClass(WordCount.class);
```

//Providing the mapper and reducer class names

```
job.setMapperClass(TokenizerMapper.class);
```

```
job.setReducerClass(IntSumReducer.class);
```

//Setting configuration object with the Data Type of output Key and Value for map and reduce

```
job.setOutputKeyClass(Text.class);
```

```
job.setOutputValueClass(IntWritable.class);
```

## Main Explanation (Cont'd)

```
/* The HDFS input and output directory to be  
fetched from the command line */
```

```
FileInputFormat.addInputPath(job, new  
Path(args[0]));
```

```
FileOutputFormat.setOutputPath(job, new  
Path(args[1]));
```

```
/* Submit the job to the cluster and wait for it to  
finish */
```

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

# Running the Code

- Configure environment variables

```
export JAVA_HOME=...
```

```
export PATH=${JAVA_HOME}/bin:${PATH}
```

```
export HADOOP_CLASSPATH=${JAVA_HOME}/lib/tools.jar
```

- Compile WordCount.java and create a jar:

```
$ hadoop com.sun.tools.javac.Main WordCount.java
```

```
$ jar cf wc.jar WordCount*.class
```

- Put files to HDFS

```
$ hdfs dfs -put YOURFILES input
```

- Run the application

```
$ hadoop jar wc.jar WordCount input output
```

- Check the results

```
$ hdfs dfs -cat output/*
```

```
import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {

    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);
            }
        }
    }

    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);
        }
    }

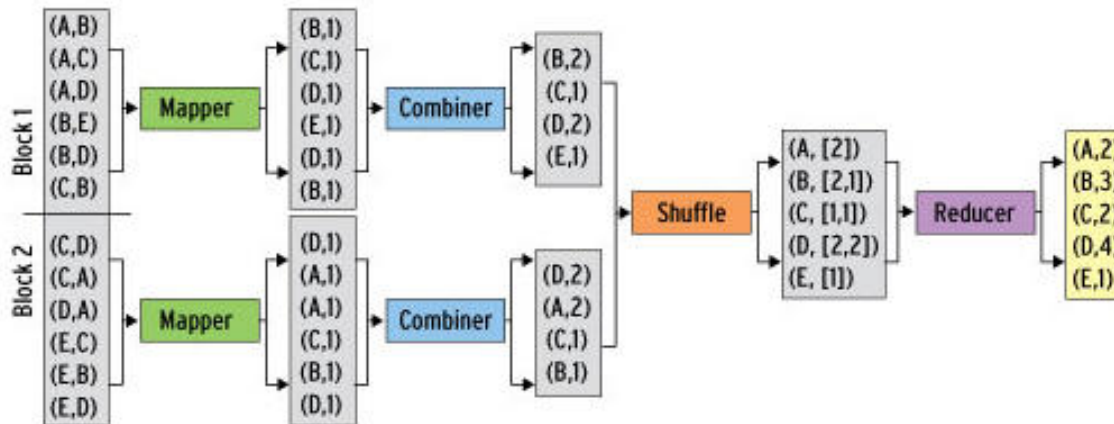
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "word count");
        job.setJarByClass(WordCount.class);
        job.setMapperClass(TokenizerMapper.class);
        job.setReducerClass(IntSumReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

# Combiners

- Often a Map task will produce many pairs of the form  $(k, v_1), (k, v_2), \dots$  for the same key  $k$ 
  - E.g., popular words in the word count example
- Combiners are a general mechanism to reduce the amount of intermediate data, thus saving network time
  - They could be thought of as “mini-reducers”
- Warning!
  - The use of combiners must be thought carefully
    - Optional in Hadoop: the correctness of the algorithm **cannot depend on** computation (or even execution) of the combiners
    - A combiner operates on each map output key. It must have the same output key-value types as the Mapper class.
    - A combiner can produce summary information from a large dataset because it replaces the original Map output
  - Works only if reduce function is commutative and associative
    - In general, reducer and combiner **are not interchangeable**

# Combiners in WordCount

- Combiner combines the values of all keys of **a single mapper node** (single machine):



- Much less data needs to be copied and shuffled!
- If combiners take advantage of all opportunities for local aggregation we have at most  $m \times V$  intermediate key-value pairs
  - $m$ : number of mappers
  - $V$ : number of unique terms in the collection
- Note: not all mappers will see all terms

# Combiners in WordCount

- In WordCount.java, you only need to add the follow line to Main:

```
job.setCombinerClass(IntSumReducer.class);
```

- This is because in this example, Reducer and Combiner do the same thing
- **Note: In most cases, this is not true!**
- You need to write an extra combiner class
- Given two files:
  - file1: Hello World Bye World
  - file2: Hello Hadoop Bye Hadoop
- The first map emits:
  - < Hello, 1> < World, 2> < Bye, 1>
- The second map emits:
  - < Hello, 1> < Hadoop, 2> < Bye, 1>

# Partitioner

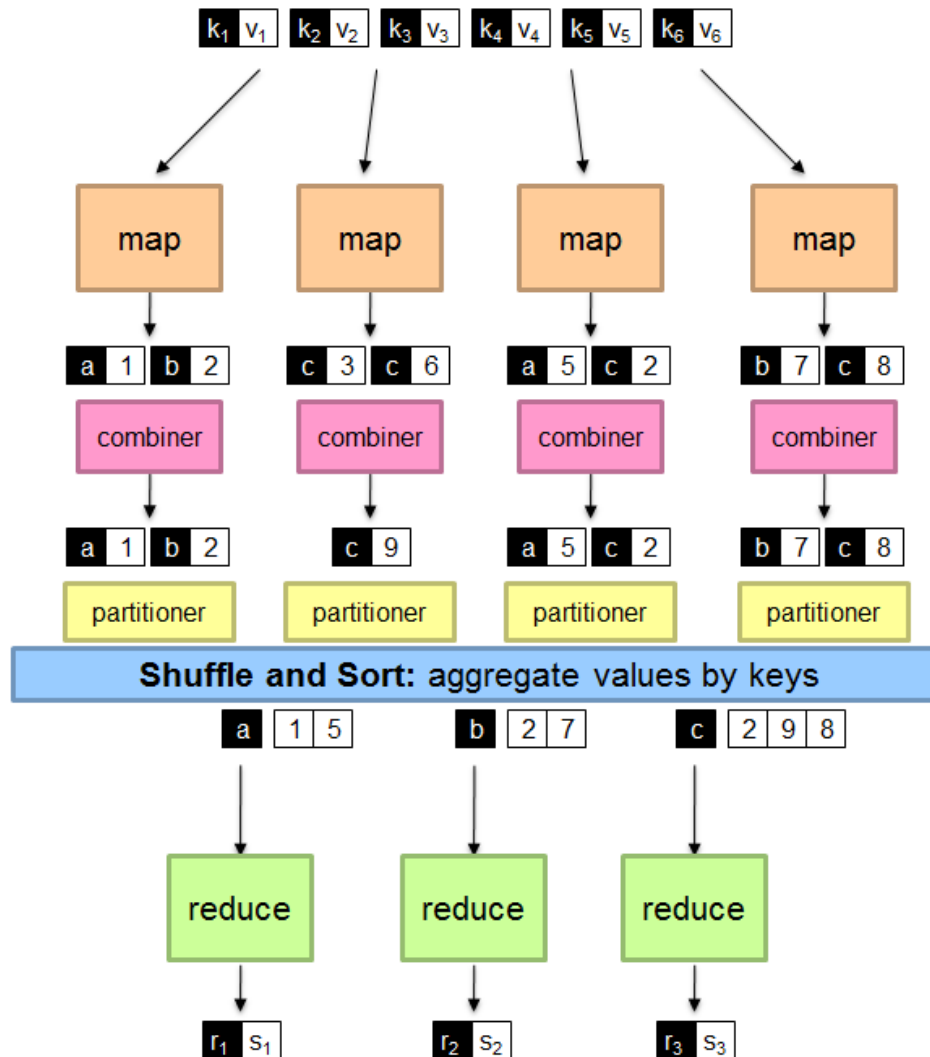
- Partitioner controls the partitioning of the keys of the intermediate map-outputs.
  - The key (or a subset of the key) is used to derive the partition, typically by a *hash function*.
  - The total number of partitions is the same as the number of reduce tasks for the job.
    - This controls which of the  $m$  reduce tasks the intermediate key (and hence the record) is sent to for reduction.
- System uses HashPartitioner by default:
  - $\text{hash}(\text{key}) \bmod R$
- Sometimes useful to override the hash function:
  - E.g.,  $\text{hash}(\text{hostname}(\text{URL})) \bmod R$  ensures URLs from a host end up in the same output file
    - <https://www.unsw.edu.au/faculties> and <https://www.unsw.edu.au/about-us> will be stored in one file
- Job sets Partitioner implementation (in Main)

# MapReduce Recap

- Programmers must specify:
  - $\text{map}(k_1, v_1) \rightarrow [(k_2, v_2)]$
  - $\text{reduce}(k_2, [v_2]) \rightarrow [k_3, v_3]$
  - All values with the same key are reduced together
- Optionally, also:
  - $\text{combine}(k_2, [v_2]) \rightarrow [k_3, v_3]$ 
    - Mini-reducers that run in memory after the map phase
    - Used as an optimization to reduce network traffic
  - $\text{partition}(k_2, \text{number of partitions}) \rightarrow \text{partition for } k_2$ 
    - Often a simple hash of the key, e.g.,  $\text{hash}(k_2) \bmod n$
    - Divides up key space for parallel reduce operations
- The execution framework handles everything else...



# MapReduce Recap



# For Large Datasets

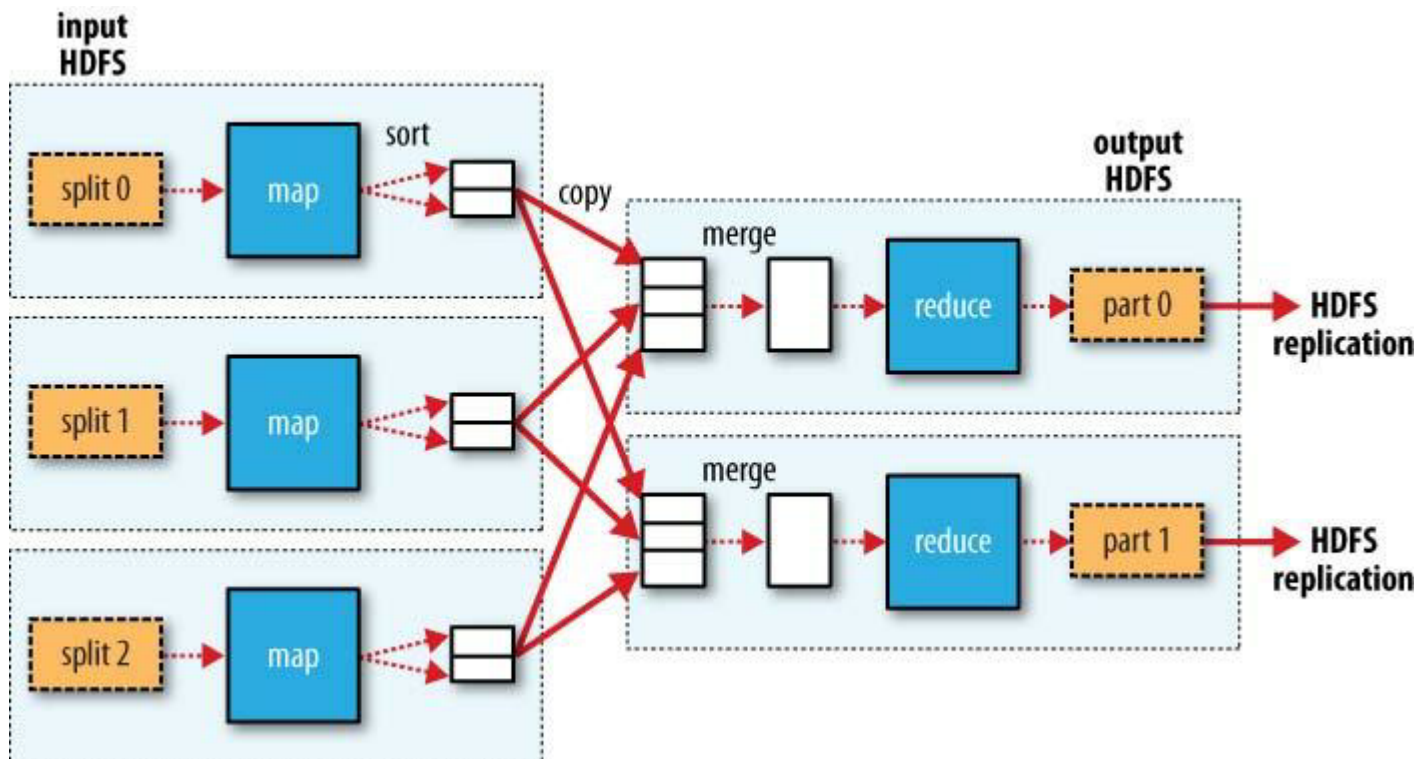
- Data stored in HDFS (organized as blocks)
- Hadoop MapReduce Divides input into fixed-size pieces, *input splits*
  - Hadoop creates one map task for each split
  - Map task runs the user-defined map function for each *record* in the split
  - Size of a split is normally the size of a HDFS block (e.g., 64Mb)
- Data locality optimization
  - Run the map task on a node where the input data resides in HDFS
  - This is the reason why the split size is the same as the block size
    - The largest size of the input that can be guaranteed to be stored on a single node
    - If the split spanned two blocks, it would be unlikely that any HDFS node stored both blocks

# For Large Datasets

- Map tasks write their output to local disk (not to HDFS)
  - Map output is intermediate output
  - Once the job is complete the map output can be thrown away
  - Storing it in HDFS with replication, would be overkill
  - If the node of map task fails, Hadoop will automatically rerun the map task on another node
- Reduce tasks don't have the advantage of data locality
  - Input to a single reduce task is normally the output from all mappers
  - Output of the reduce is stored in HDFS for reliability
  - The number of reduce tasks is not governed by the size of the input, but is specified independently

# More Detailed MapReduce Dataflow

- When there are multiple reducers, the map tasks partition their output:
  - One partition for each reduce task
  - The records for every key are all in a single partition
  - Partitioning can be controlled by a user-defined partitioning function



Thanks