COMP5349 – Cloud Computing

Week 4: MapReduce and Hadoop

A/Prof. Uwe Roehm
School of Information Technologies

Outline

- **Theoretical Foundation**
- **MapReduce Framework**
- **Hadoop Basics**

Based on Jeff Dean, Sanjay Ghemawat, *MapReduce: Simplified Data Processing on Large Clusters*. In OSDI'04,
Yahoo! Hadoop Tutorial, Module 4: MapReduce http://developer.yahoo.com/hadoop/tutorial/module4.html
Tom White, Hadoop, the definitive Guide, O'reilly, 2009

Functional Programming

- "**functional programming** is a programming paradigm that treats computation as the evaluation of mathematical functions" [-- wikipedia]
  - Lisp, Erlang, F#, Scala etc.

- Most of the languages we learn and use (Java, C#, C, C++,…) belong to **imperative programming**, which is based on the von Neumann architecture
  - emphasising on telling computer step-by-step what to do

Big Data and Functional Programming

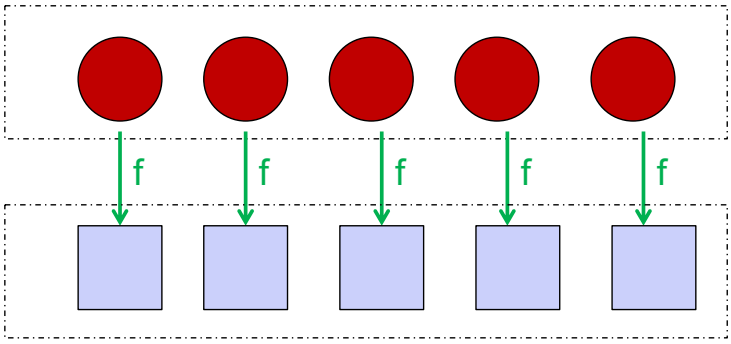
- Functional language is well suited for parallel programming
  - Industry started to adopt functional language
  - Prominent new features of Java 8
    - Functional interface and *lamda expression*.
    - Collection API has been rewritten to use lots of functional interface through a new Stream API
    - Enable parallel processing on single node with multi core processors
    - The functional interface include map, reduce and a lot more
      - Very much like what Spark provides

Features of Functional Programming

- Functional operations do not modify data structures, they just create new ones
  - No “side effects”
  - Easier to verify, optimize, and parallelize programs
- Higher-order functions, which takes another functions as parameters provide an easy way to handle collection
  - Traditional imperative programming usually relies on a loop structure, visitor pattern, etc. to traverse a collection
  - Some script language, javascript, python, ruby simulate higher-order functions using the closure concept
- Two useful higher-order functions that inspire MapReduce framework are:
  - **map** and **fold**, or **reduce**

Higher-order function: map

- The map function applies a given function to all elements in a list and returns the result as a new list
  - map f originalList



The original list with five elements

Apply function f to all element

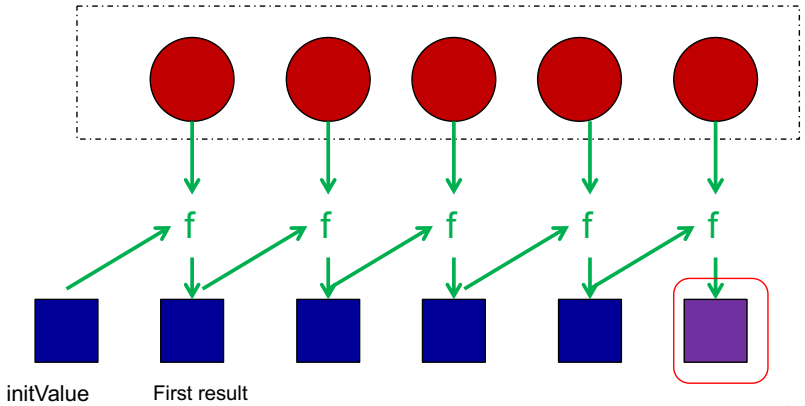
Obtain a new list of five elements

We can easily parallel the execution of function f

The diagram is based on MapReduce lecture slides used in CSE 490H in University of Washington

Higher order function: fold/reduce

- The fold function apply a given function together with an initial value iteratively on list elements; it returns the value obtained from applying the function and initial value to the last element.
  - fold f initValue originalList



The original list with five elements

The diagram is based on MapReduce lecture slides used in CSE 490H in University of Washington

Outline

- Theoretical Foundation
- **MapReduce Framework**
- Hadoop Basics

Motivation: Large Scale Data Processing

- Need to process large amounts of data (> 1 TB)
  - Eg. build inverted word-document index for the whole web
- Need to parallelize across hundreds/thousands of CPUs
- Want to make this easy
  - Automatic parallelization and distribution
  - Fault-tolerance
  - I/O scheduling
  - Status and monitoring

Programming Model

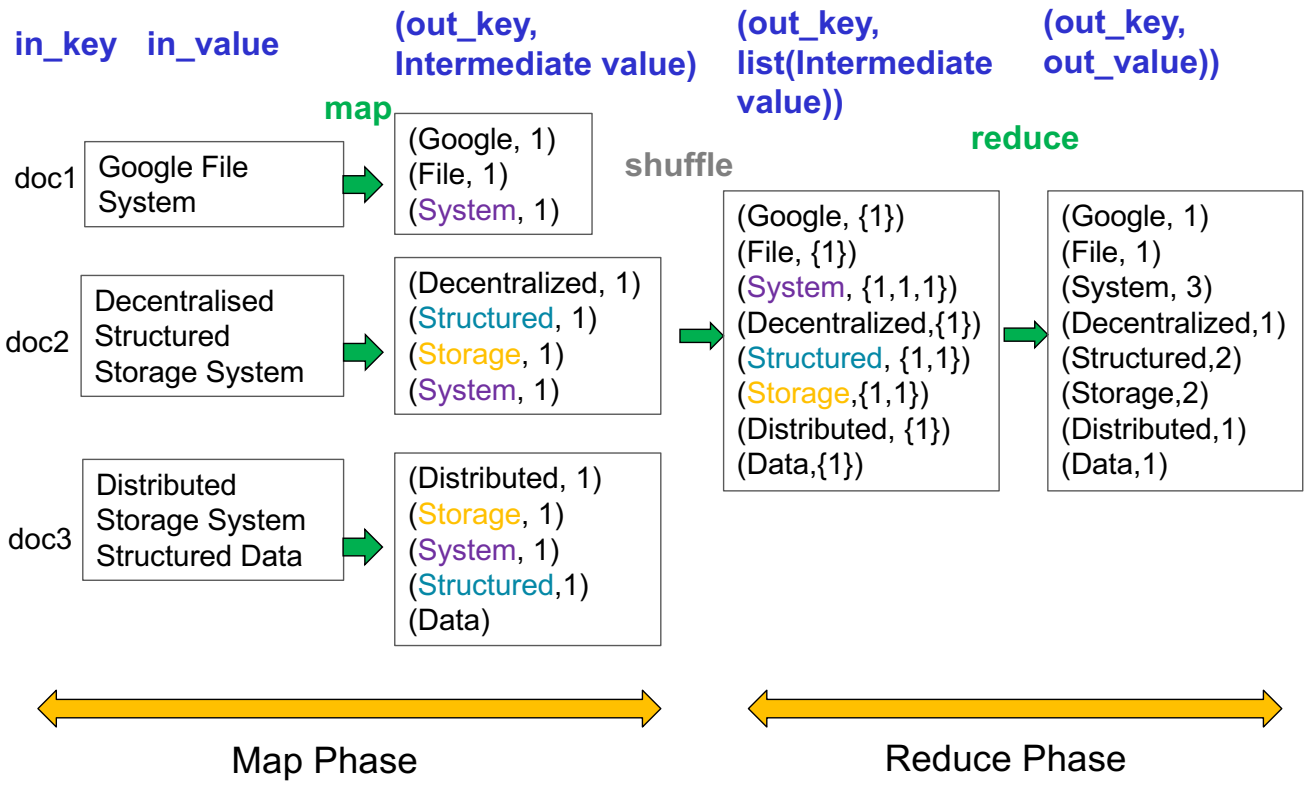
- Inspired by **map** and **fold** in FP
- Input & Output: each a set of key/value pairs
- Programmer specifies two functions:
  - map (**in_key**, in_value) -> list(**out_key**, intermediate_value)
    - Processes input key/value pair
    - Produces a list of intermediate pairs
  - reduce (**out_key**, list (intermediate_value)) -> list(**out_key**,out_value)
    - Combines all intermediate values for a particular key
  - Produces a set of merged output values for a given key (usually just one)

Example: Count Word Occurrences

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

reduce(String out_key, Iterator intermediate_values):
    // out_key: a word
    // intermediate_values: a list of counts associated with that
    //word
     int result = 0;
     for each v in intermediate_values:
       result += ParseInt(v);
     Emit(out_key, AsString(result));
```


Word Count (WC) Example



Implementation

- Typical cluster:
  - 100s or 1000s of multi-core x86 machines, 2-4 GB of memory
  - Limited bisection bandwidth
  - Storage is on local IDE disks
  - GFS: distributed file system manages data
- Job scheduling system
  - jobs made up of tasks (map task and/or reduce task)
  - scheduler assigns tasks to machines

MapReduce Execution Overview

Data Locality

Split 0 and 1 located in the same worker machine, two map tasks are assigned to this worker. Input data is read locally!

1 GFS chunk may equal **1** or **more** splits

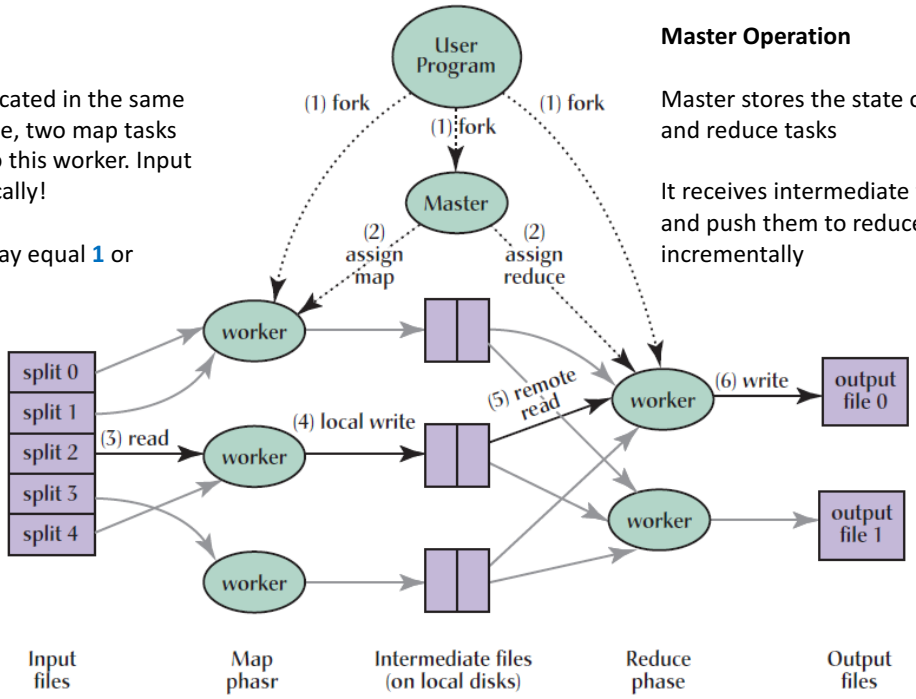


Fig. 1. Execution overview.

Master Operation

Master stores the state of each map and reduce tasks

It receives intermediate file locations and push them to reduce tasks incrementally

Diagram from the CACM version of the original MapReduce paper

Parallel Execution

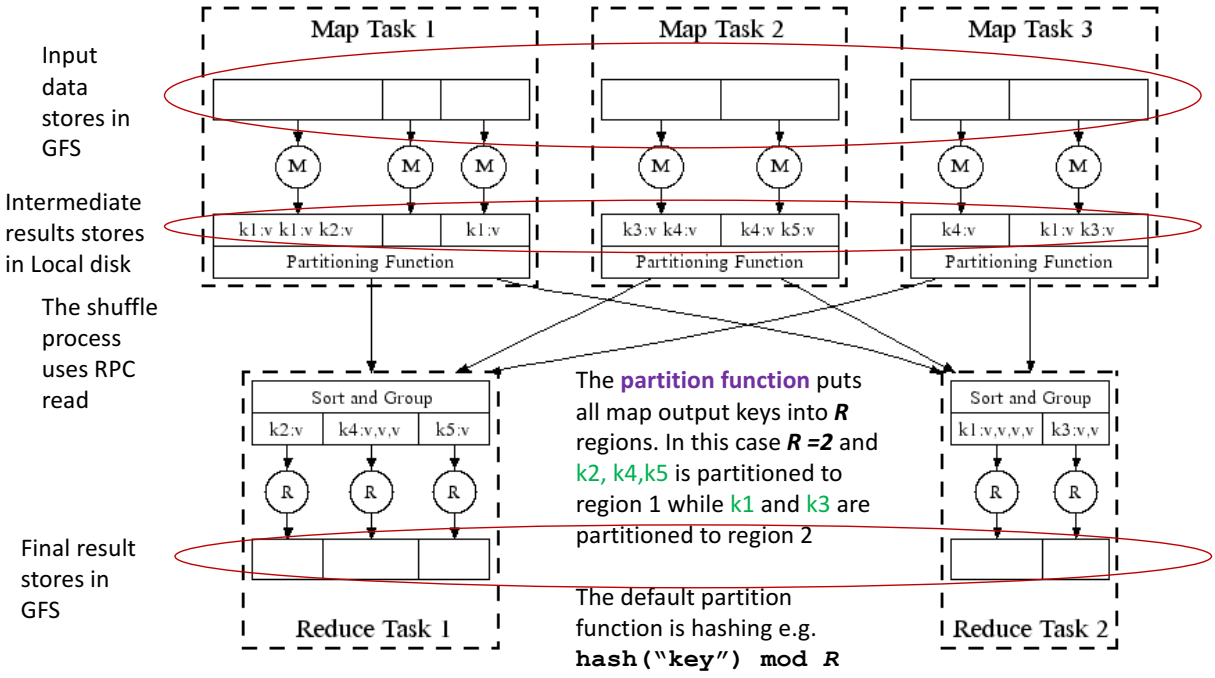


Diagram from the original slides by Jeff Dean and Sanjay Ghemawat

Questions

- Why are intermediate results saved to local disk?
- What happens when a map task fails in the middle of execution?

Outline

- Theoretical Foundation
- MapReduce Framework
- **Hadoop Basics**

Hadoop Framework (MRv1)

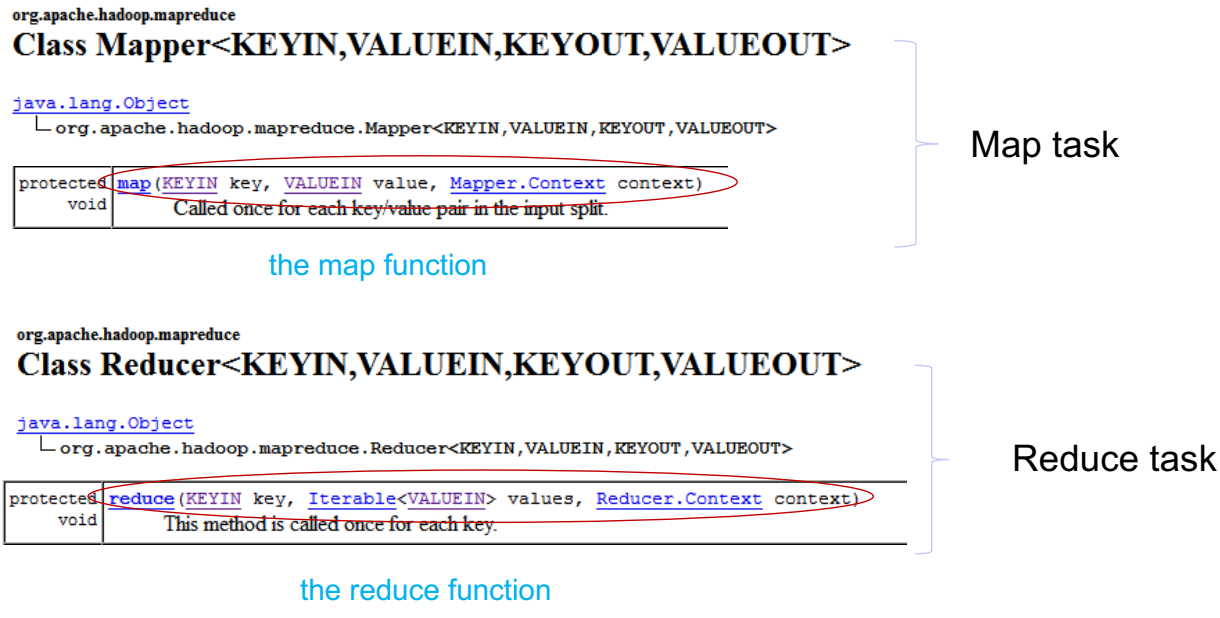
- MapReduce: the general principle as introduced by Google
- Open-Source implementation and Apache project: **Hadoop**

- A master node runs **JobTracker** (a MapReduce job)
  - In small or medium cluster (< 40 servers), it is OK to put the HDFS NameNode and MapReduce JobTracker in the same physical nodes
  - In large cluster (multiple racks), it is better to have dedicated NameNode and JobTracker
- Slave nodes (many) run **TaskTracker**
  - TaskTrackers run on DataNodes of HDFS
  - Each TaskTracker has a capacity of a number of Map or Reduce Tasks
    - Configurable, depends on the number of cores
  - Locality, moving computation to data

Hadoop Framework (MRv2)

- The resource management capabilities have been separated to Apache Hadoop YARN
  - YARN is a general purpose distributed application management framework
  - It can mange not only MR jobs but also others such as Spark jobs
  - Resource management is more flexible and efficient
    - In MRv1, each node has a fixed slots for map and reduce tasks
    - YARN does not differentiate task types, only their resource requirements matters
  - Yarn has a master node running Resource Manager and lots of slave nodes running Node manager.
  - The MRv1 and MRv2 APIs are compatible

Hadoop MapReduce Java API



Hadoop MapReduce Java API

org.apache.hadoop.mapreduce

Class Job

java.lang.Object
  └ org.apache.hadoop.mapreduce.task.JobContextImpl
    └ org.apache.hadoop.mapreduce.Job

void	setMapperClass(Class<? extends Mapper> cls) Set the Mapper for the job.
void	setReducerClass(Class<? extends Reducer> cls) Set the Reducer for the job.
void	setNumReduceTasks(int tasks) Set the number of reduce tasks for the job.

The MapReduce Job

.....

Java API: Example Mapper for WC

KeyIn ValueIn KeyOut ValueOut

```
public static class TagMapper
  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 {

        String[] wordArray = value.toString().split(" ");

        for(String term: wordArray) {
                word.set(term);
                context.write(word, ONE);
        }

  }
}
```

Emit Intermediate result

Google File System
Decentralised Structured Storage System
Distributed Storage System for Structured Data
....

Each line of the input file is fed into the map function as a value

Java API: Example Reducer for WC

```
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);
   }
}
```

(key, list of values) passed to each reduce function

(Google, {1})
(File, {1})
(System, {1,1,1})
(Decentralized,{1})
(Structured, {1,1})
(Storage,{1,1})
(Distributed, {1})
(Data,{1})

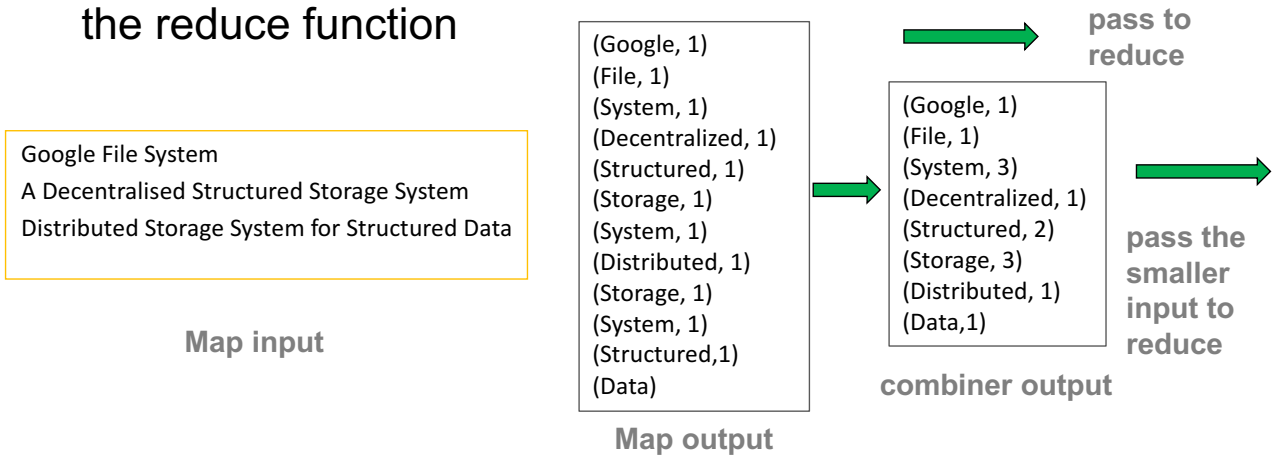
Java API: Example Driver for WC

```
public class WordCount{
   public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        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.setNumReduceTasks(2);
        job.setJarByClass(WordCount.class);
        job.setMapperClass(TagMapper.class);
        job.setCombinerClass(IntSumReducer.class);
        job.setReducerClass(IntSumReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        TextInputFormat.addInputPath(job, new Path(otherArgs[0]));
        TextOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
        System.exit(job.waitForCompletion(true) ? 0 : 1);
}
}
```

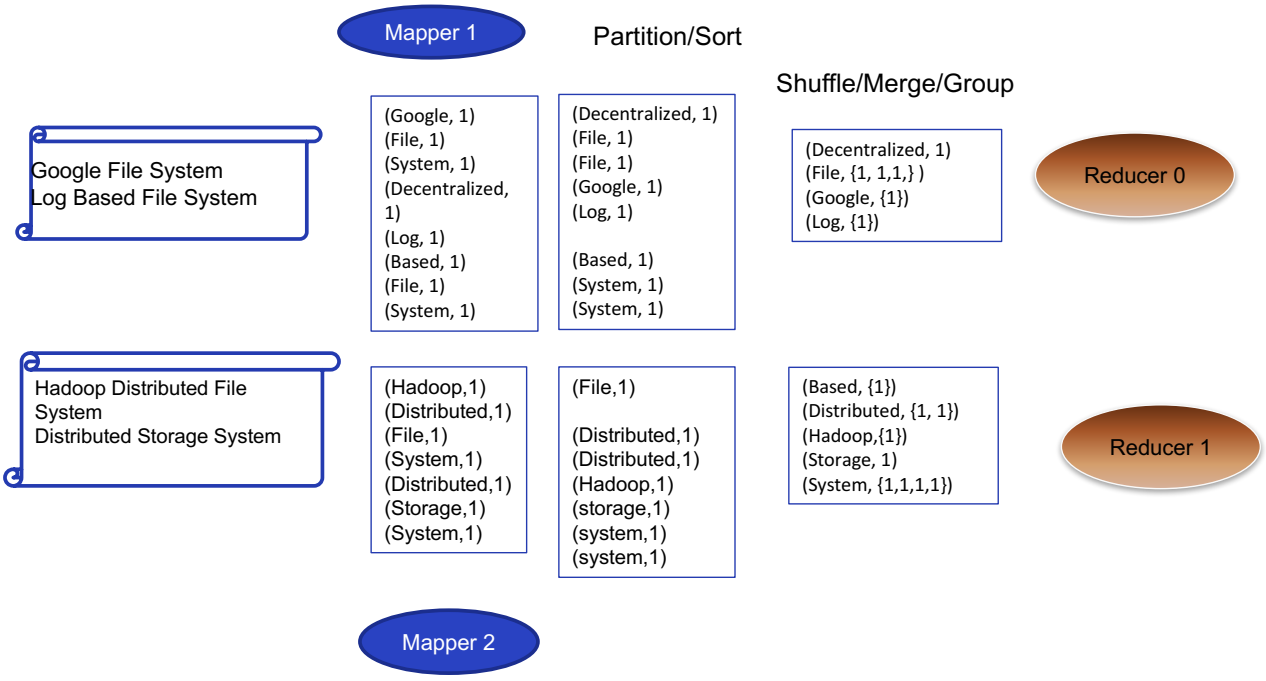
Combiner does “reduce” on local map output

The Combiner Function

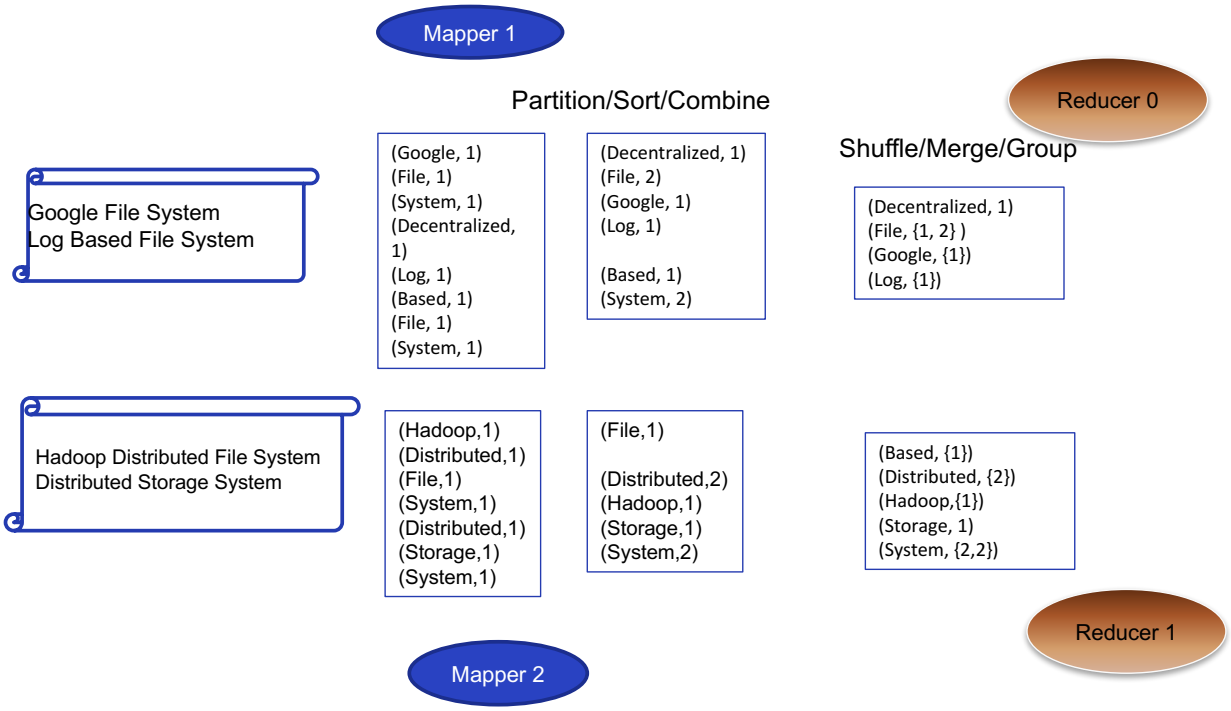
- Combiner is an optimization mechanism to minimize the data transferred between the map and reduce tasks
- Combiner function runs on the map side to merge some of the map intermediate result
  - It is like running a reduce function locally on each map task
- The output of the combiner function becomes the input of the reduce function



Word Count Without Combiner



Word Count With Combiner



MapReduce Program Design

- Involves writing **map** and **reduce** functions
- Each map and reduce **task** (mapper, reducer) will run those functions multiple times depends on the input size
- Combiner is just a **reduce** function running locally on the mapper side to aggregate results locally
  - There is a chain of keys that are related
    - Map output key is the input key of reducer if there is no combiner
    - If there is a combiner, map output key is the input key of combiner, the output key of combiner becomes the input key of the

Input Data

- **InputFormat**
  - How input files are split up and read is defined by the InputFormat. FileInputFormat is the abstract class for all file inputs.

InputFormat	Description	Key	Value
TextInputFormat	Default format for plain text files; reads lines of textfiles	Byte offset of the line	The line content
KeyValueTextInputFormat	Parse lines into key, val pairs	Everything up to the first tab character	The remainder of the line
SequenceFileInputFormat	A Hadoop-specific high-performance binary format	User defined	User defined

- **InputSplit**
  - An InputSplit describes a unit of work that comprises a single map task in a MapReduce job.
  - By default, the FileInputFormat and its descendants break a file up into 64 MB chunks (the same size as blocks in HDFS)
  - This can be modified by setting split size in configuration file or in code at run time.

Input Data

- RecordReader
  - The *RecordReader* class actually loads the data from its source and converts it into (key, value) pairs suitable for reading by the Mapper.
  - The RecordReader instance is defined by the InputFormat. The default InputFormat, *TextInputFormat*, provides a *LineRecordReader*, which treats each line of the input file as a new value. The key associated with each line is its byte offset in the file.
  - Developers can write their own RecordReader

OutputFormat

- *Serialization is the process of turning structured objects into a byte stream for transmission* over a network or for writing to persistent storage.
- *Deserialization is the* process of turning a byte stream back into a series of structured objects.
- Serialization is important for interprocess communication and for persistent storage in Hadoop
  - The Reducer use Remote Procedure Calls(RPC) to get intermediate data stored locally in Mapper nodes.
  - The RPC protocol uses serialization to render the message into a binary stream to be sent to the remote node, which then deserializes the binary stream into the original message.
  - The Reducer also write final results to HDFS
- Both interprocess communication and persistent storage requires serialization to be compact, fast, extensible and interoperable
- Hadoop uses its own serialization format, **`Writable`**
  - **`Text`**, **`IntWritable`** are all subclass of **`Writable`**.

Output Collection

- **`Context`** is used to collect and write the ouput into intermediate as well as final files. The method
  context.write() takes (Key,Value)
- Partition and Shuffle
  - This process of moving map outputs to the reducers is known as *shuffling*.
  - The **`Partitioner`** class determines which partition a given (key, value) pair emitted by mapper will go to. The default one use hashing
- **Sort:** Each reduce task is responsible for reducing the values associated with several intermediate keys. The set of intermediate keys on a single node is automatically sorted by Hadoop before they are presented to the Reducer.
- **OutputFormat:** The (key, value) pairs collected by **`Context`** are then written to output files. The way they are written is governed by the *OutputFormat*. The default **`TextOutputFormat`** writes lines in "key \t value" form

Communication Between Mappers and Reducers

If there is a combiner function, it runs after sort and before disk spilling

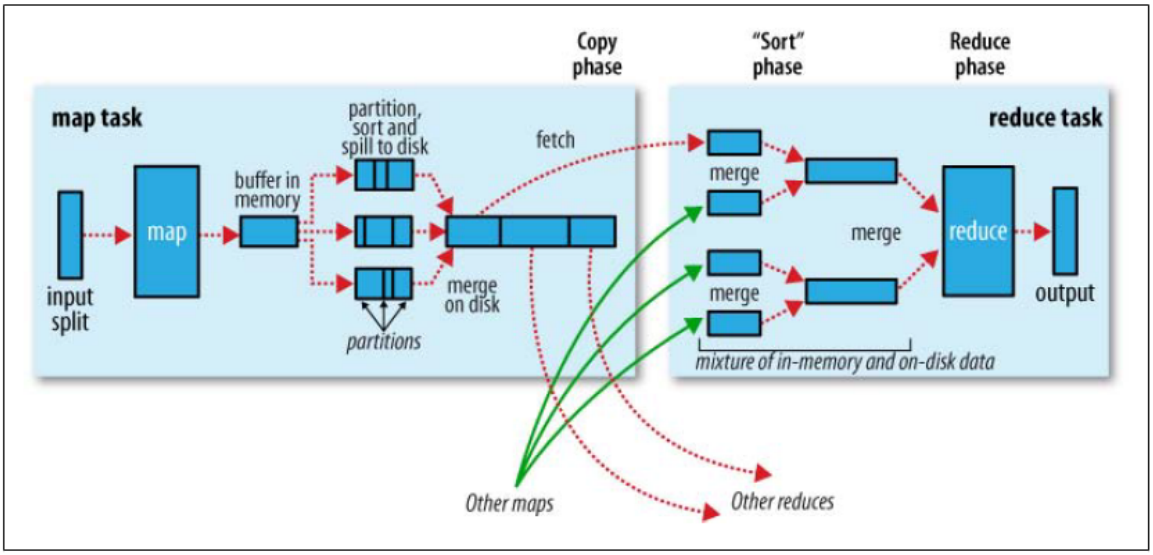


Figure 6-4. Shuffle and sort in MapReduce

Diagram from Tom White, Hadoop, the definitive Guide, O'reilly, 2009, page 163

Hadoop Streaming

- Hadoop streaming is a utility to enable writing MapReduce programs in languages other than Java
  - The utility itself is packed as a jar file
  - We can specify any **executable** or **script** as mapper/combiner/reducer
- Eg.

```
hadoop jar /usr/local/hadoop/share/hadoop/tools/lib/hadoop-streaming-2.7.2.jar \
  -input myInputDirs \
  -output myOutputDirs \
  -mapper  mapper.py \
   -reducer  reducer.py \
  -D mapreduce.job.reduces=2 \
  -D mapreduce.job.name= 'word count'\
  -file mapper.py \
  -file reducer.py
```

How does Streaming work

- The Hadoop framework assigns map and reduce tasks to slave nodes as usual
- Each **map** task
  - starts the executable or script in separate **process**,
  - converts the input key value pairs into lines and feed the lines to the stdin of the **process**
    - The **process** read the input line, does map work, and write output line by line to standard out
  - collects output from the stdout of the process and convert each line to key/value pair as map output

How does Streaming work (cont'd)

- The framework does partition, shuffle and sort (but not grouping!) to prepare the reduce task input
  - The reduce task input is sorted map output
- Each reduce task
  - Starts the executable or script in separate **process**
  - converts the input key value pairs into lines and feed the lines to the stdin of the **process**
    - The **process** read the input line, does reduce work, and write output line by line to standard out
      - The input line has the format (key, value)
      - Script code needs to identify the boundary of keys (see example in lab code!)
  - collects output from the stdout of the process and convert each line to key/value pair as map output

Summary

- **Theoretical Foundation of MapReduce**
  - The higher order function map and fold in functional programming
- **MapReduce Framework**
  - Master/Slave architecture
  - Data locality
- **Hadoop**
  - Basics Architecture
    - MRv1 vs. MRv2 (YARN)
    - JobTracker vs. Resource Manager
  - Basic programming
    - Mapper, Reducer, Driver, Combiner, Input & Output format