DISTRIBUTED RESEARCH ON EMERGING APPLICATIONS & MACHINES
dream-lab.in | **Indian Institute of Science**, Bangalore

SE252:Lecture 13-14, Feb 24/25

ILO3:Algorithms and Programming Patterns for Cloud Applications *(Hadoop)*

Yogesh Simmhan

ILO 3

- Algorithms and Programming Patterns for Cloud Applications
  - *Examine* the design of task and data parallel distributed algorithms for Clouds and ✓
  - *use* them to construct Cloud applications. ➲Project
  - *Demonstrate* the use of task graphs and Map-Reduce programming model.
  - *Apply* Amdahl's law and data locality principles to ✓
  - *analyse* and *characterize* the potential speedup of Cloud applications. ➲Project

Patterns & Technologies

- **MapReduce** is a distributed data-parallel programming model from Google
- MapReduce works best with a distributed file system, called **Google File System (GFS)**
- **Hadoop** is the open source framework implementation from Apache that can execute the MapReduce programming model
- **Hadoop Distributed File System (HDFS)** is the open source implementation of the GFS design
- **Elastic MapReduce (EMR)** is Amazon's PaaS

MapReduce

"A simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs."

Dean and Ghermawat, "MapReduce: Simplified Data Processing on Large Clusters", OSDI, 2004

MapReduce Design Pattern

- Clean abstraction for programmers
- Automatic parallelization & distribution

- Fault-tolerance
- A batch data processing system
- Provides status and monitoring tools

MapReduce: Data-parallel Programming Model

- Process data using map & reduce functions
- **map(k_i, v_i) → List<k_m, v_m>[]**
  - ***map*** is called on every input item
  - Emits a series of intermediate key/value pairs
- All values with a given key are **grouped** together
- **reduce(k_m, List<v_m>[]) → List<k_r, v_r>[]**
  - ***reduce*** is called on every unique key & all its values
  - Emits a value that is added to the output

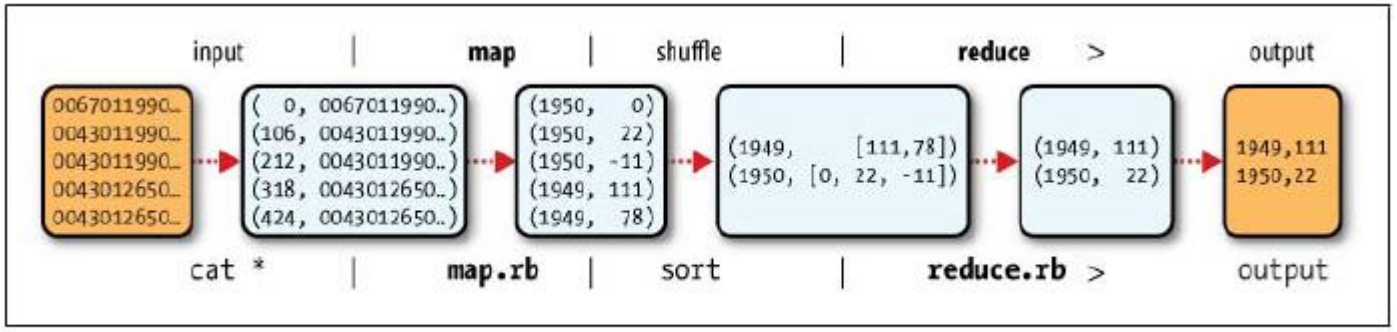


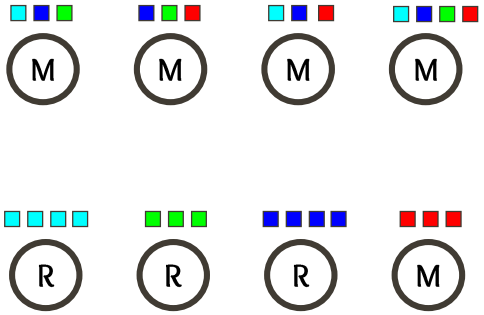
Figure 2-1. MapReduce logical data flow

MR Borrows from Functional Programming

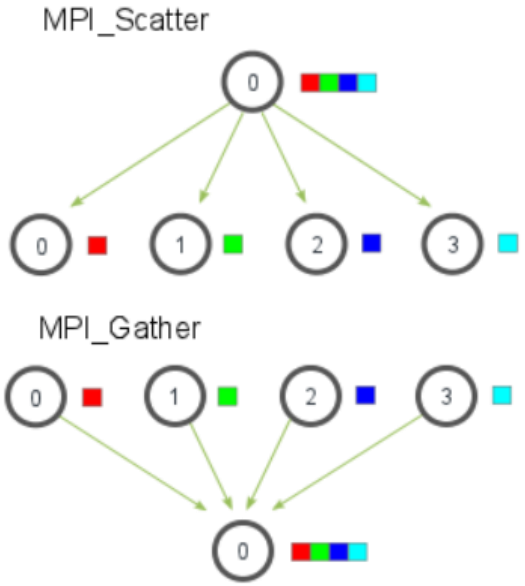
- Functional operations do not modify data structures
  - *They always create new ones*
  - Original data still exists in unmodified form (read only)
- Data flows are implicit in program design
- Order of operations does not matter
  - *Commutative*: **a ◊ b ◊ c = b ◊ a ◊ c = c ◊ b ◊ a**

- In a purely functional setting
  - Elements computed by **map** cannot see the effects of map on other elements
  - Order of applying **reduce** is *commutative*
    - » Allowing parallel/reordered execution
  - More optimizations possible if **reduce** is also *associative*
    - » **(a ◊ b) ◊ c = a ◊ (b ◊ c)**

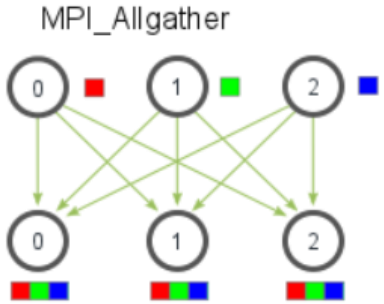
MapReduce & MPI Scatter-Gather



Routing determined by key



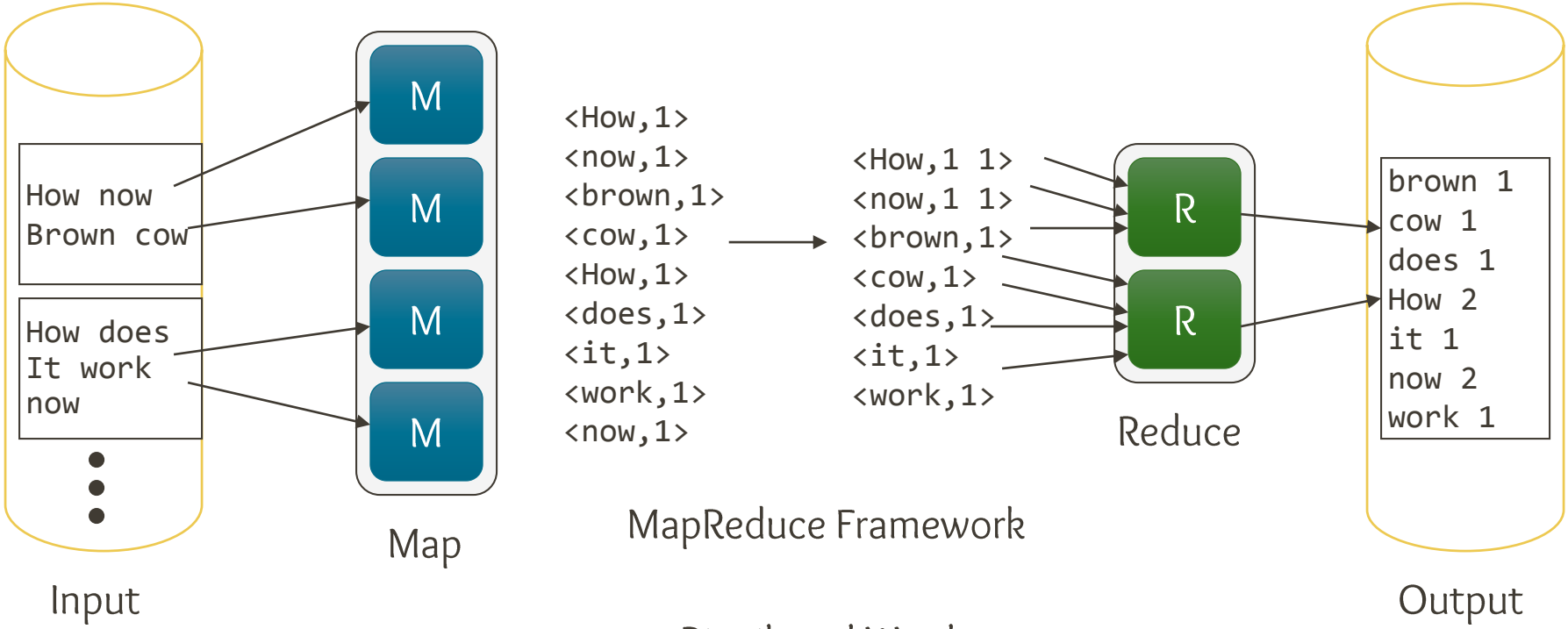
Routing determined by array index/element position



http://mpitutorial.com/mpi-scatter-gather-and-allgather/

MapReduce: Programming Model

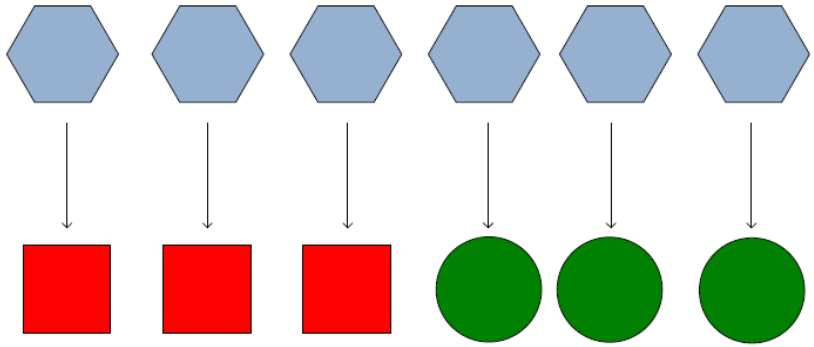
Map(k1,v1) → list(k2,v2)
Reduce(k2, list(v2)) → list(v2)



Distributed Wordcount

Map

- Input records from the data source
  - lines out of files, rows of a database, etc.
- Passed to map function as key-value pairs
  - Line number, line value
- map() produces *zero or more intermediate values*, each associated with an output key



Map

- Example Wordcount

```
map(String input_key, String input_value):
  // input_key: line number
  // input_value: line of text
  for each Word w in input_value.tokenize()
    EmitIntermediate(w, "1");
```

(0, “How now brown cow”) →
[(“How”, 1), (“now”, 1), (“brown”, 1), (“cow”, 1)]

- Example: Upper-case Mapper

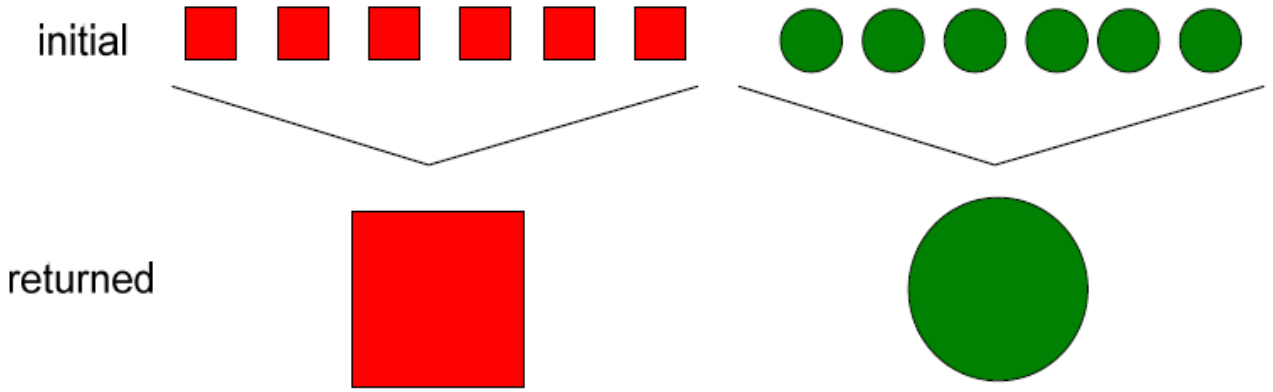
```
let map(k, v) = emit(k.toUpper(), v.toUpper())
(“foo”, “bar”) → (“FOO”, “BAR”)
(“Foo”, “other”) → (“FOO”, “OTHER”)
(“key2”, “data”) → (“KEY2”, “DATA”)
```

- Example: Filter Mapper

```
let map(k, v) =
if (isPrime(v)) then emit(k, v)
(“foo”, 7) → (“foo”, 7)
(“test”, 10) → (nothing)
```

Reduce

- All the intermediate values from map for a given output key are combined together into a list
- reduce() combines these intermediate values into one or more final values for that same output key … Usually one final value per key

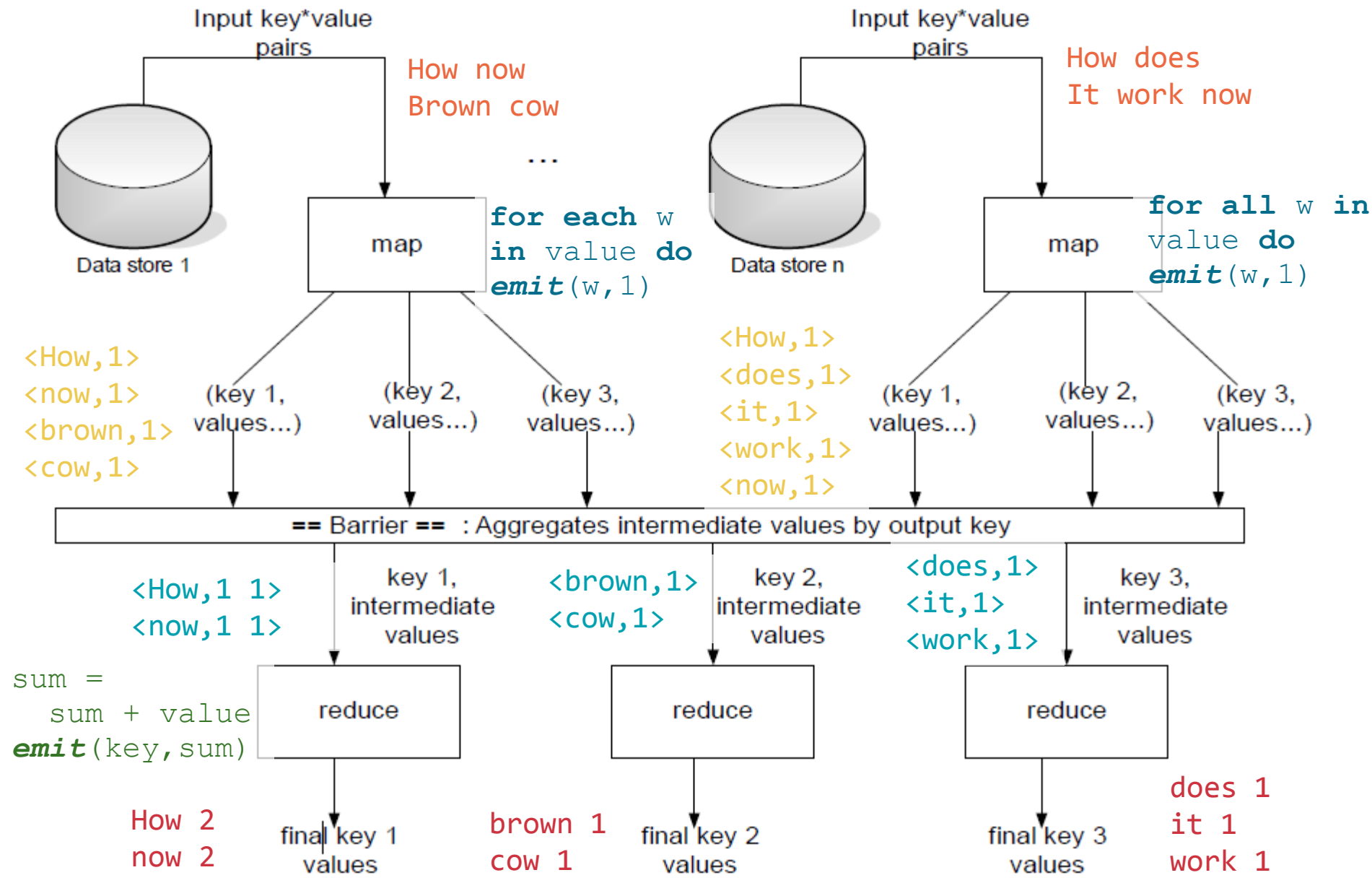


Reduce

- Example Wordcount

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

(“A”, [1, 1, 1]) → (“A”, 3)
(“B”, [1, 1]) → (“B”, 2)


Input key*value pairs
How now
Brown cow
...
Data store 1
map
for each w in value do
emit(w,1)
Input key*value pairs
How does
It work now
Data store n
map
for all w in value do
emit(w,1)
<How,1>
<now,1>
<brown,1>
<cow,1>
(key 1, values...)
(key 2, values...)
(key 3, values...)
<How,1>
<does,1>
<it,1>
<work,1>
<now,1>
(key 1, values...)
(key 2, values...)
(key 3, values...)
== Barrier == : Aggregates intermediate values by output key
<How,1 1>
<now,1 1>
key 1, intermediate values
<brown,1>
<cow,1>
key 2, intermediate values
<does,1>
<it,1>
<work,1>
key 3, intermediate values
sum =
sum + value
emit(key,sum)
reduce
reduce
reduce
How 2
now 2
final key 1 values
brown 1
cow 1
final key 2 values
final key 3 values
does 1
it 1
work 1

Anagram Example

*"An **anagram** is a type of word play, the result of rearranging the letters of a word or phrase to produce a new word or phrase, using all the original letters exactly once; for example **orchestra** can be rearranged into **carthorse**." … Wikipedia*

```
public class AnagramMapper extends MapReduceBase implements
                  Mapper<LongWritable, Text, Text, Text> {
  private Text sortedText = new Text();
  private Text orginalText = new Text();
  public void map(LongWritable key, Text value,
        OutputCollector<Text, Text> outputCollector, Reporter reporter) {

     String word = value.toString();
     char[] wordChars = word.toCharArray();
     Arrays.sort(wordChars);
     String sortedWord = new String(wordChars);
     sortedText.set(sortedWord);
     orginalText.set(word);
     // Sort word and emit <sorted word, word>
     outputCollector.collect(sortedText, orginalText);
  }
}
```

http://code.google.com/p/hadoop-map-reduce-examples/

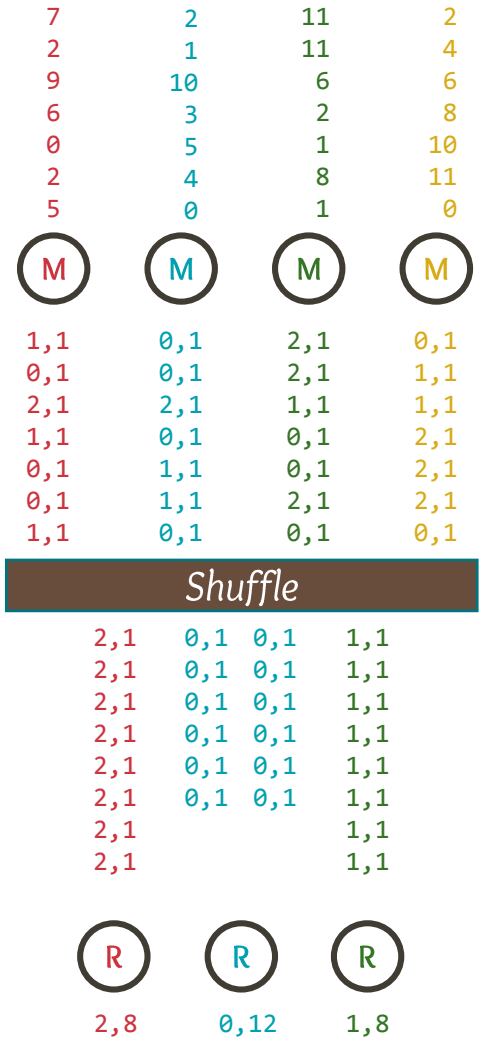
Anagram Example…

```
public void reduce(Text anagramKey, Iterator<Text> anagramValues,
       OutputCollector<Text, Text> results, Reporter reporter) {
    String output = "";
    while(anagramValues.hasNext()) {
       Text anagram = anagramValues.next();
       output = output + anagram.toString() + "~";
    }
    StringTokenizer outputTokenizer =
       new StringTokenizer(output,"~");
    // if the values contain more than one word
    // we have spotted a anagram.
    if(outputTokenizer.countTokens()>=2) {
       output = output.replace("~", ",");
       outputKey.set(anagramKey.toString());
       outputValue.set(output);
       results.collect(outputKey, outputValue);
    }
}
```

5-min Assignment

1) Build a histogram
2) Sort numbers

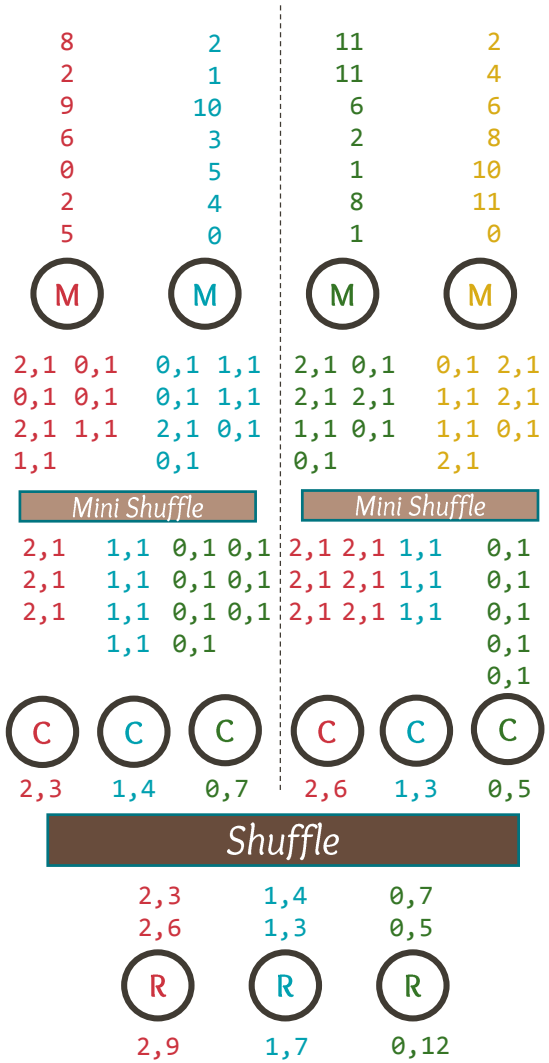
MapReduce for Histogram




```
int bucketWidth = 4
Map(k, v) {
    emit(v/bucketWidth, 1)
}
```

```
Reduce(k, v[]){
    sum=0;
    foreach(w in v[])  sum++;
    emit(k, sum)
}
```

MapReduce for Histogram




```
int bucketWidth = 4
Map(k, v) {
   emit(v/bucketWidth, 1)
}

Combine(k, v[]) {
   // same code as reducer
}

Reduce(k, v[]){
   sum=0;
   foreach(w in v[])  sum+=w;
   emit(k, sum)
}
```

If a Reducer is *commutative* and *associative*, it can be used as a Combiner

DREAM:Lab

Hadoop Execution Model

Figures are from *Hadoop: The Definitive Guide*, Tom White, O'Reilly, 2011

Hadoop MapReduce & HDFS

- HDFS offers a distributed, replicated file store for commodity hardware

- Hadoop MapReduce uses HDFS for input, intermediate & output data staging
  - Tightly coupled with HDFS, e.g. scheduling based on locality

HDFS Read/Write

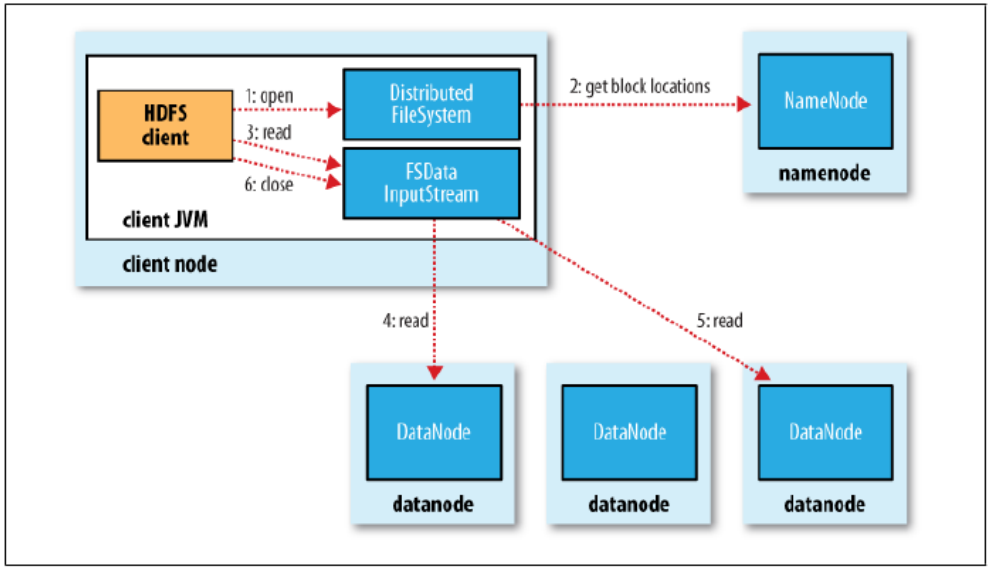
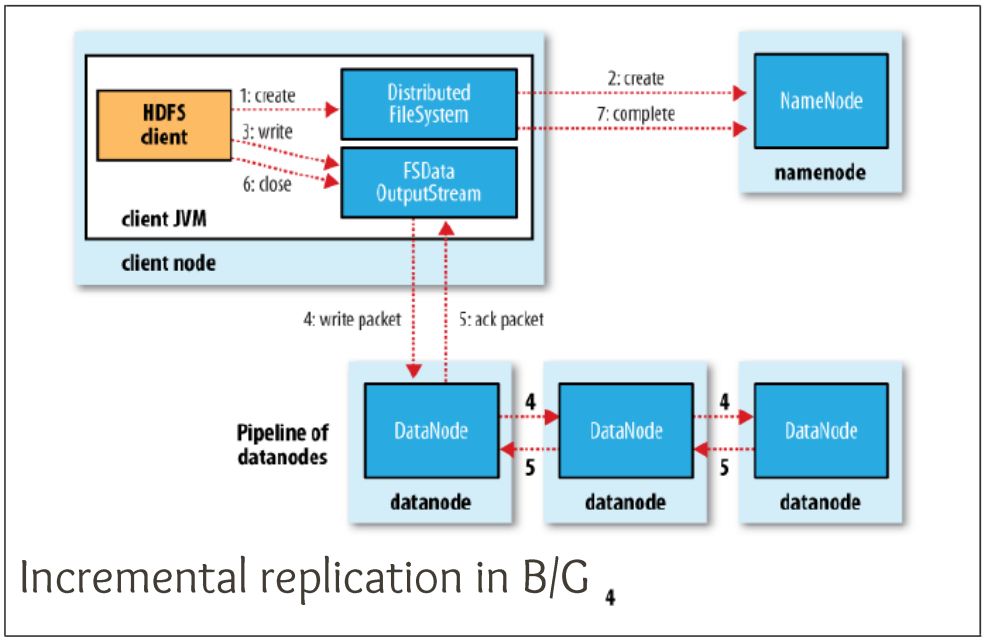


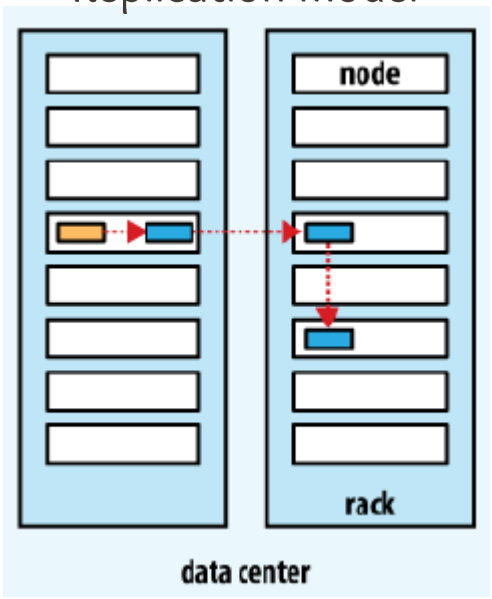
Figure 3-1. A client reading data from HDFS



Incremental replication in B/G [4]

Figure 3-3. A client writing data to HDFS

Replication Model



Avoid data loss due to node/rack failures.
Reduce N/W bandwidth usage, time taken to replicate.

Scheduling a MR Job

Mapper and Reducer *slots* available on each task tracker node.
Typically, 2 mapper slots per core, 1 reducer slot per core (reducers often "costlier" than mapper).

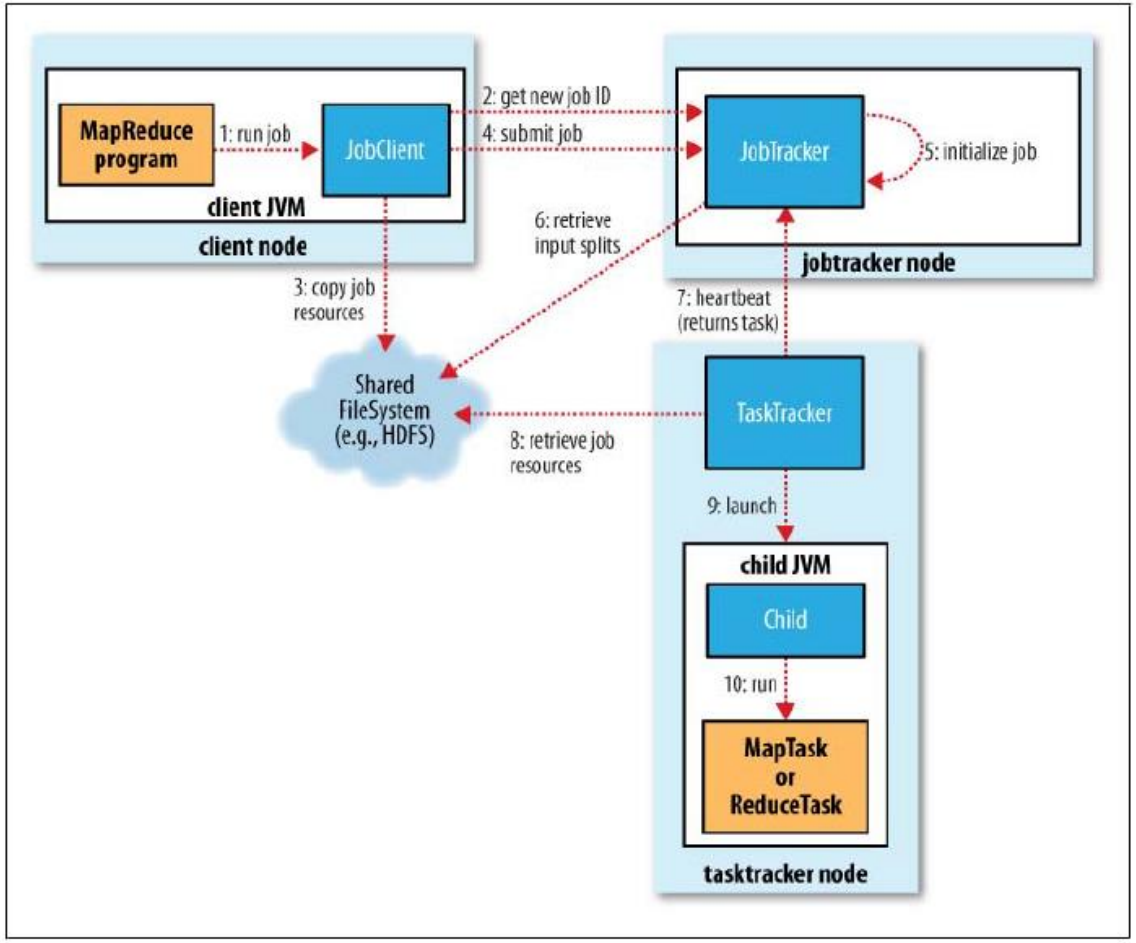


Figure 6-1. How Hadoop runs a MapReduce job

MapReduce w/ 1 & N Reducers

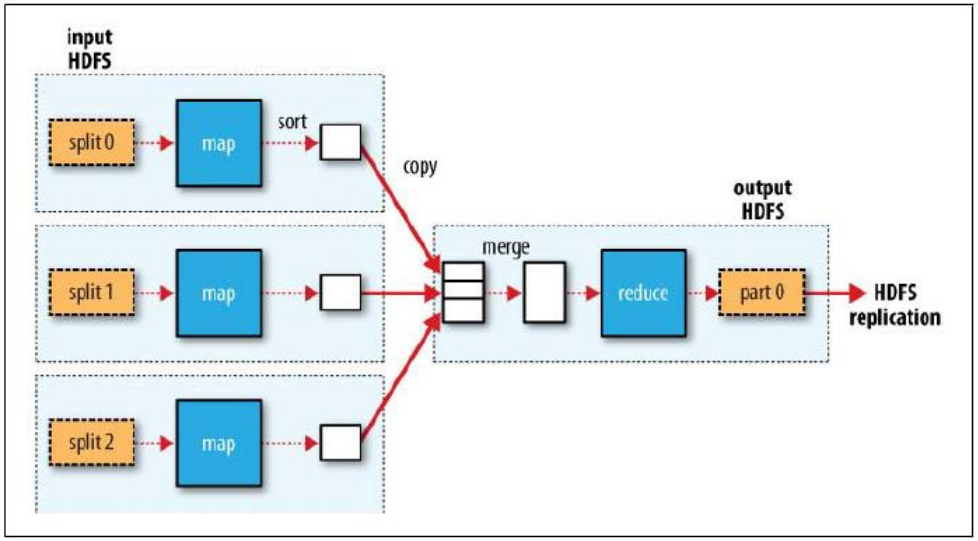


Figure 2-2. MapReduce data flow with a single reduce task

In-memory or in-disk sort, based on size of output from map for a split.
Output sent to one reducer where merge-sort happens on sorted outputs from all mappers.

One output sorted output generated by each mapper for each reducer.

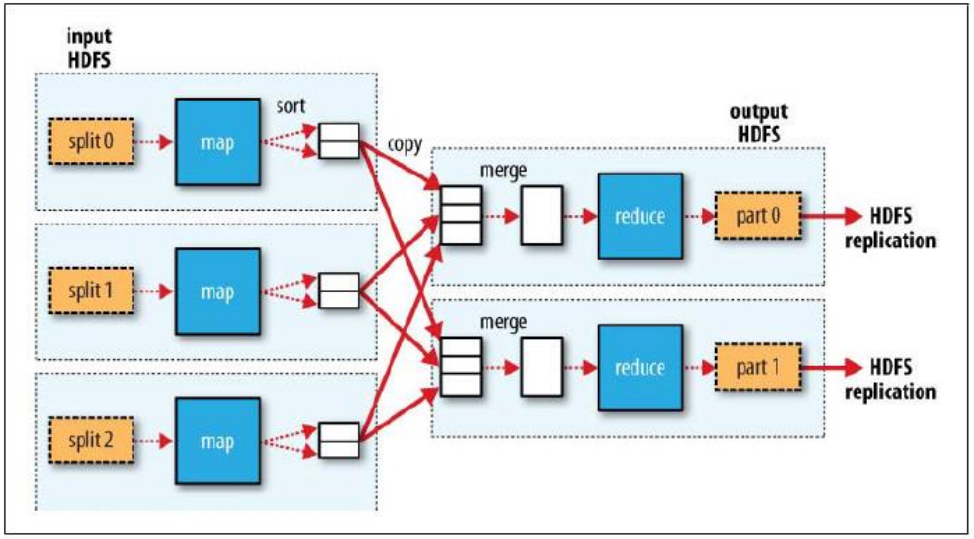


Figure 2-3. MapReduce data flow with multiple reduce tasks

Map only job

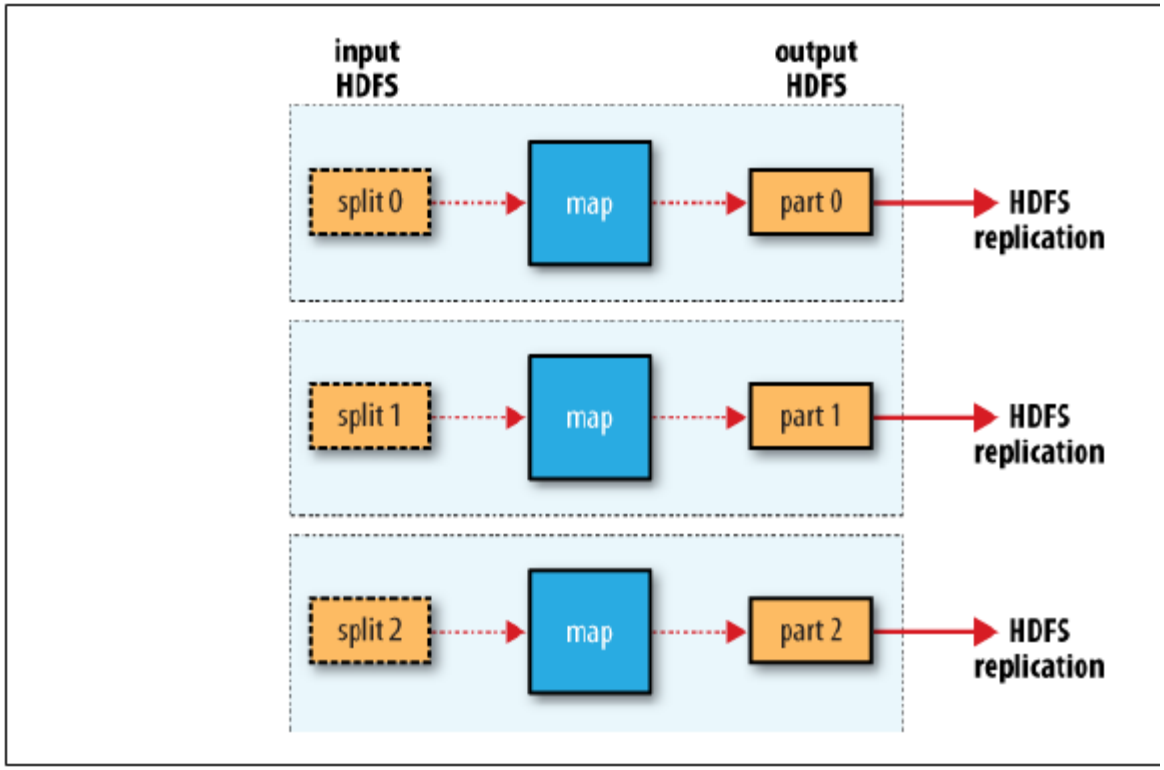


Figure 2-4. MapReduce data flow with no reduce tasks

Output parts from Mappers concatenated to get final output

Pipelining during Shuffle & Sort

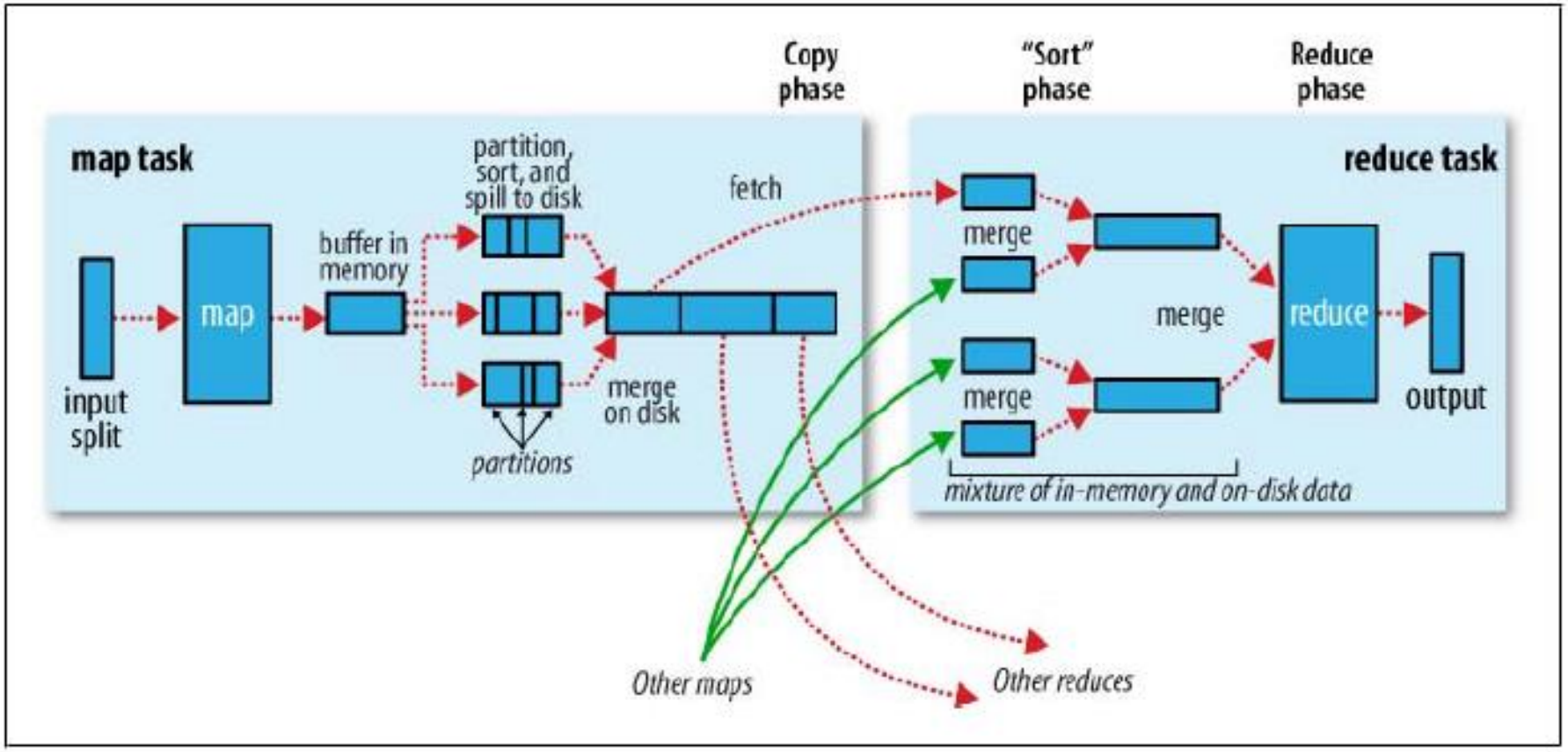


Figure 6-4. Shuffle and sort in MapReduce

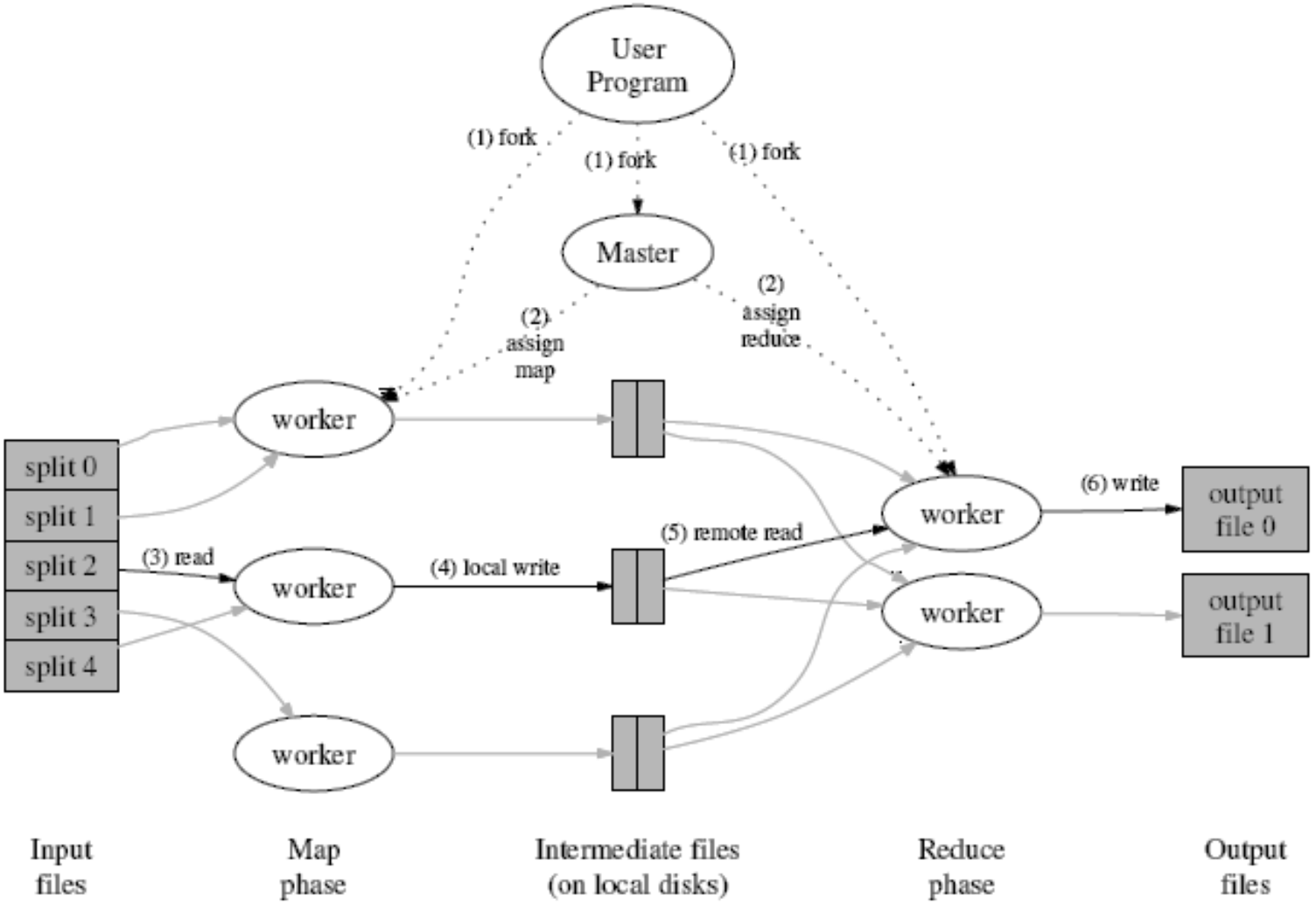
Sorting using MapReduce *(Map Only)*

- *Map* emits number as key
- *Shuffle* sorts keys within machine
- Write to disk
- Perform merge sort offline

Sorting using MapReduce

- *Map* emits number as key
- *Shuffle* sorts keys across machines
- *Partitioner* buckets keys ranges and maps them to Reducers
- *Reducer* writes keys to disk

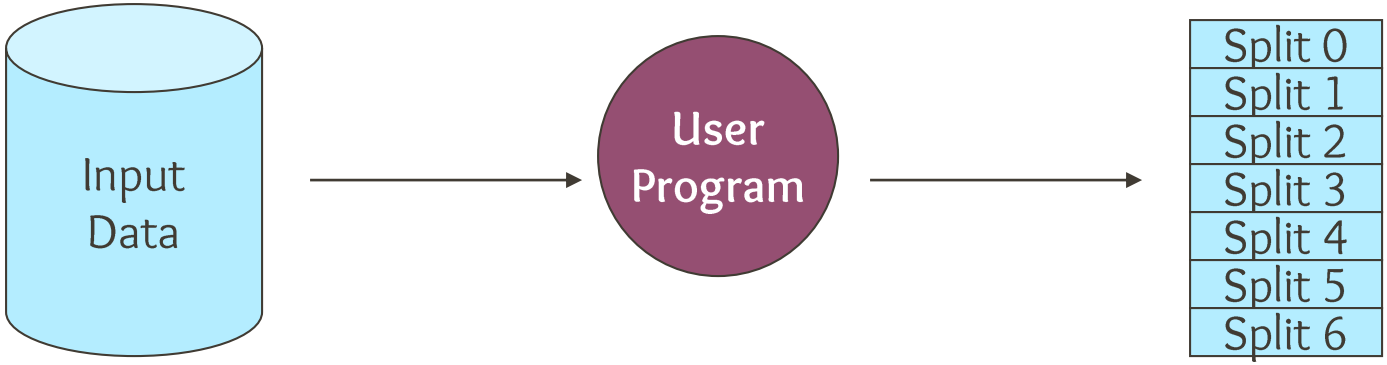
MapReduce Execution Overview



http://code.google.com/edu/parallel/mapreduce-tutorial.html

MapReduce Execution Overview

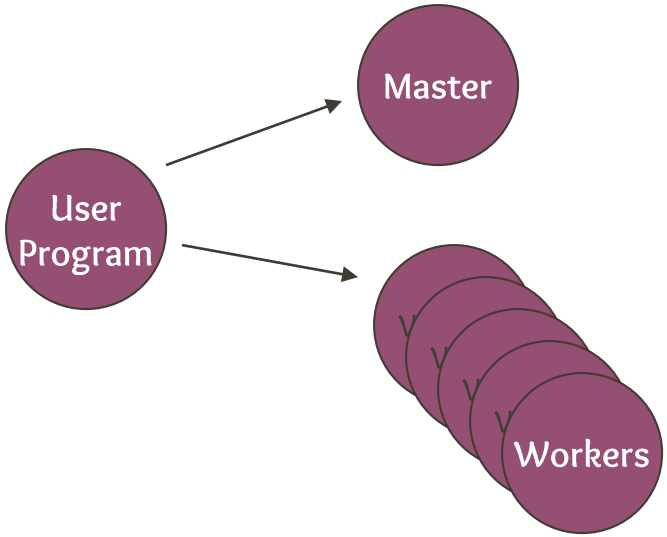
1. The user program, via the MapReduce library, splits the input data into splits



* Splits are typically 16-64mb in size

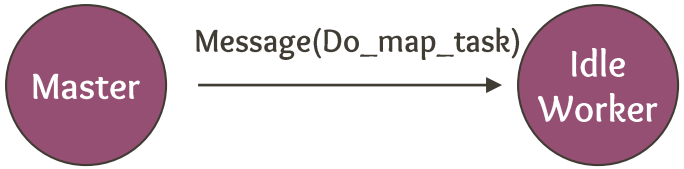
MapReduce Execution Overview

2. The user program creates process copies distributed on a machine cluster. One copy will be the “Master” and the others will be worker threads.



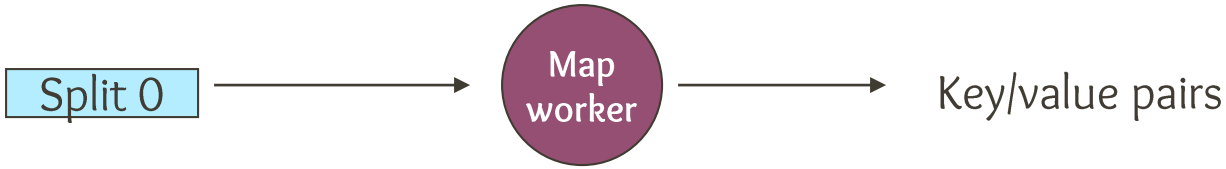
MapReduce Resources

3. The master distributes M map and R reduce tasks to idle workers.
   - M == number of splits
   - R == the intermediate key space is divided into R parts



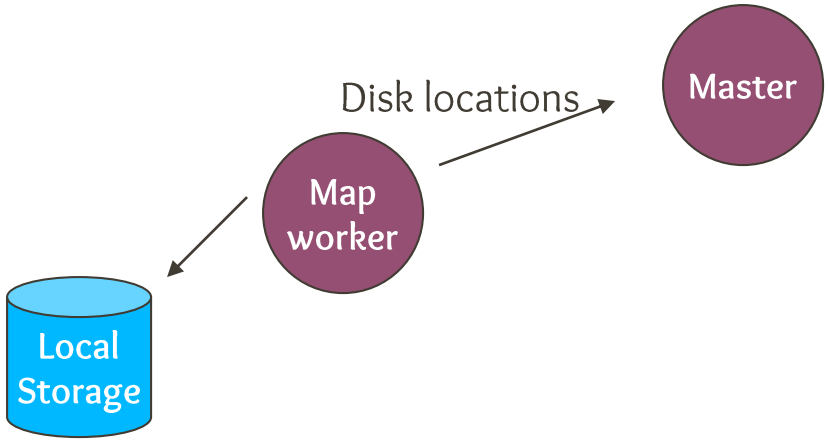
MapReduce Resources

4. Each map-task worker reads assigned input split and outputs intermediate key/value pairs.
   - Output buffered in RAM.



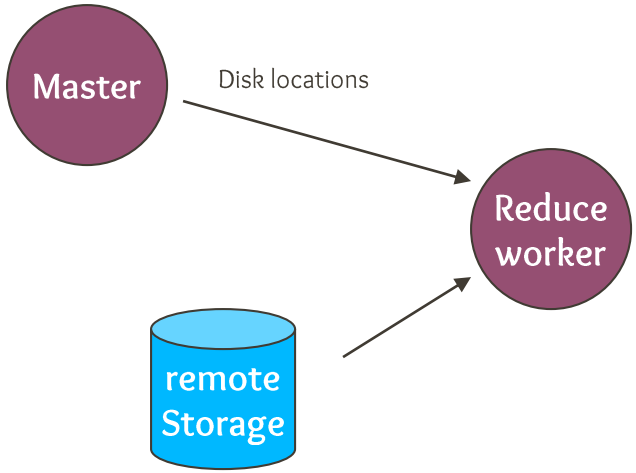
MapReduce Execution Overview

5. Each worker flushes intermediate values, partitioned into R regions, to disk and notifies the Master process.



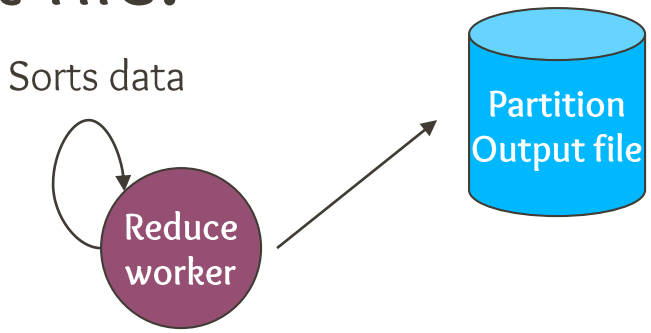
MapReduce Execution Overview

6. Master process gives disk locations to an available reduce-task worker who reads all associated intermediate data.



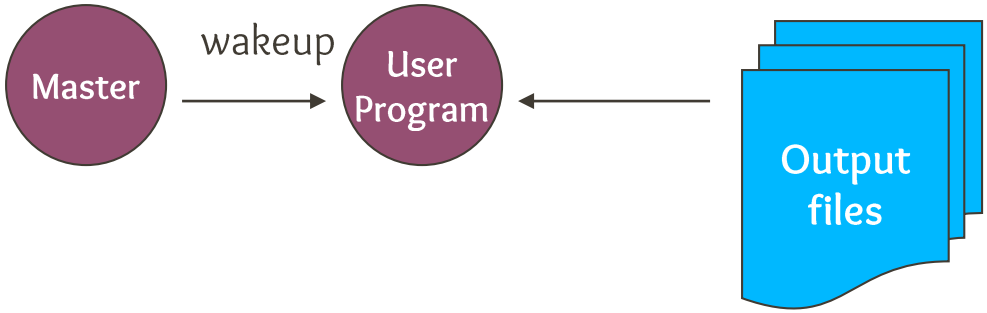
MapReduce Execution Overview

7. Each reduce-task worker sorts its intermediate data. Calls the reduce function, passing in unique keys and associated key values. Reduce function output appended to reduce-task's partition output file.



MapReduce Execution Overview

8. Master process wakes up user process when all tasks have completed. Output contained in R output files.



Locality

- Master program distributed tasks based on location of data
  - Tries to have map() tasks on same machine as physical file data, or at least same rack
- map() task inputs are divided into ~64 MB splits by default…same as HDFS block size

Fault Tolerance

- Master detects worker failures
  - Re-executes completed & in-progress map() tasks
  - Re-executes in-progress reduce() tasks
- Master notices particular input key/values cause crashes in map(), and skips those values on re-execution.
  - Effect: Can work around bugs in third-party libraries!
- NOTE: Job Tracker and Name Nodes are single points of failure

Optimizations

- No reduce can start until map is complete:
- A single slow disk controller can rate-limit the whole process
- Master redundantly executes “slow moving” map tasks; uses results of first copy to finish

Reminder

- Midterm Exam on Thu, Mar 5 during class hours (2-330PM) [10% weightage]
- Midterm paper review due on Tue 10 Mar.
- Midterm project report due on Thu 12 Mar. Demos on Fri 13 Mar. [10% weightage]