MapReduce

2015

Professor Sasu Tarkoma

MapReduce Model

Google MapReduce introduced in 2004

Jeffrey Dean et al. MapReduce: Simplified Data Processing on Large Clusters. OSDI 2004.

Apache Hadoop since 2005

http://hadoop.apache.org/

Apache Hadoop 2.0 introduced in 2012

Vinod Kumar Vavilapalli et al. Apache Hadoop YARN: Yet Another Resource Negotiator, SOCC 2013.

New cluster resource management layer (YARN)

Implicit Parallelism

The map function has implicit parallelism

This is because the order of the application of the function f to elements in a list is commutative

We can parallelize or reorder the execution

MapReduce builds on this parallelism

MapReduce

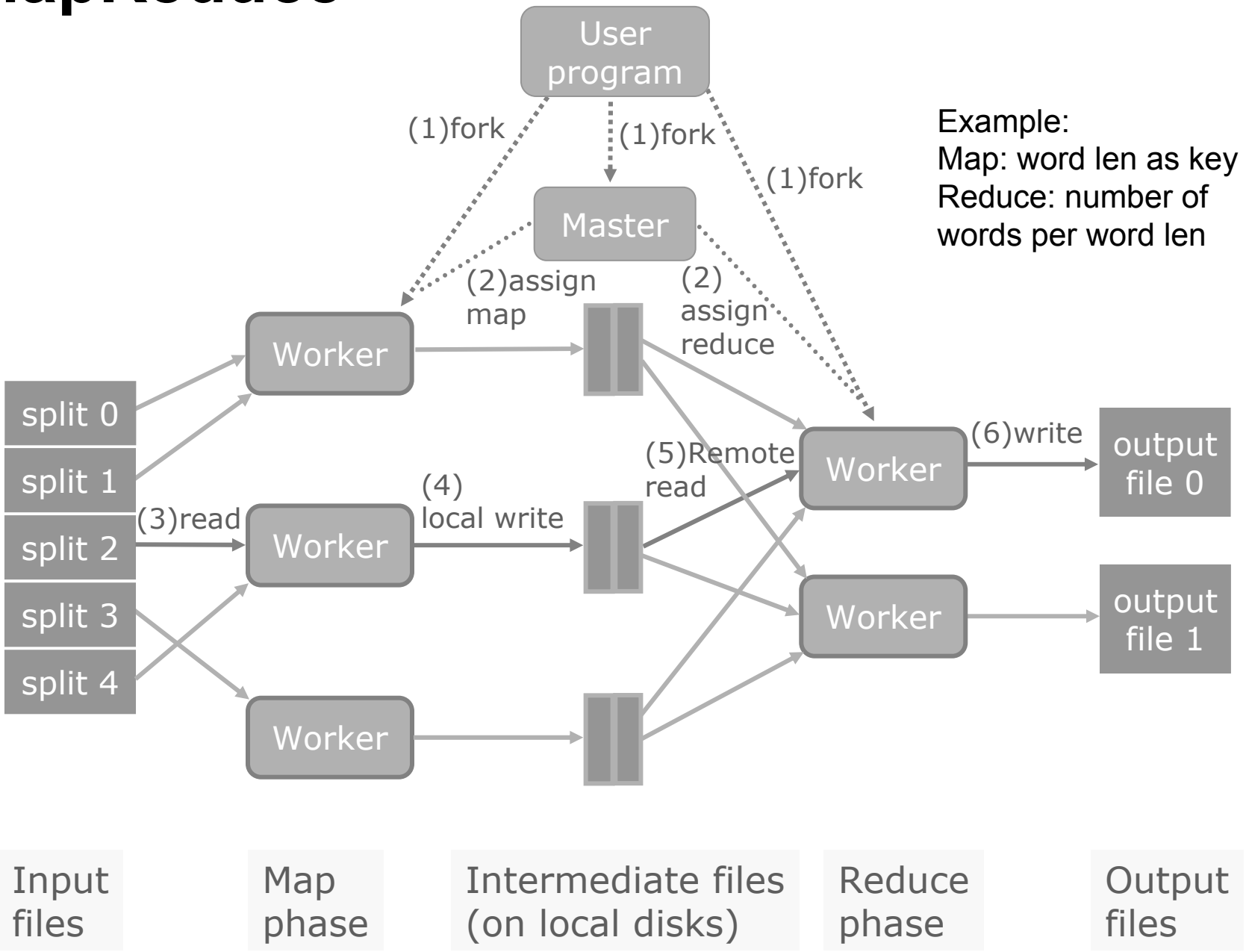
Automatic distribution and parallelization

Fault-tolerance

Cluster management tools

Abstraction for programmers

MapReduce



Example:
Map: word len as key
Reduce: number of words per word len

MapReduce terminology

Job is a full program that consists of a **Mapper** and **Reducer** for a dataset

Task is an execution of a Mapper / Reducer on some data

Task-in-Progress (TIP)

Task Attempt is an instance of an attempt to run a task on a node

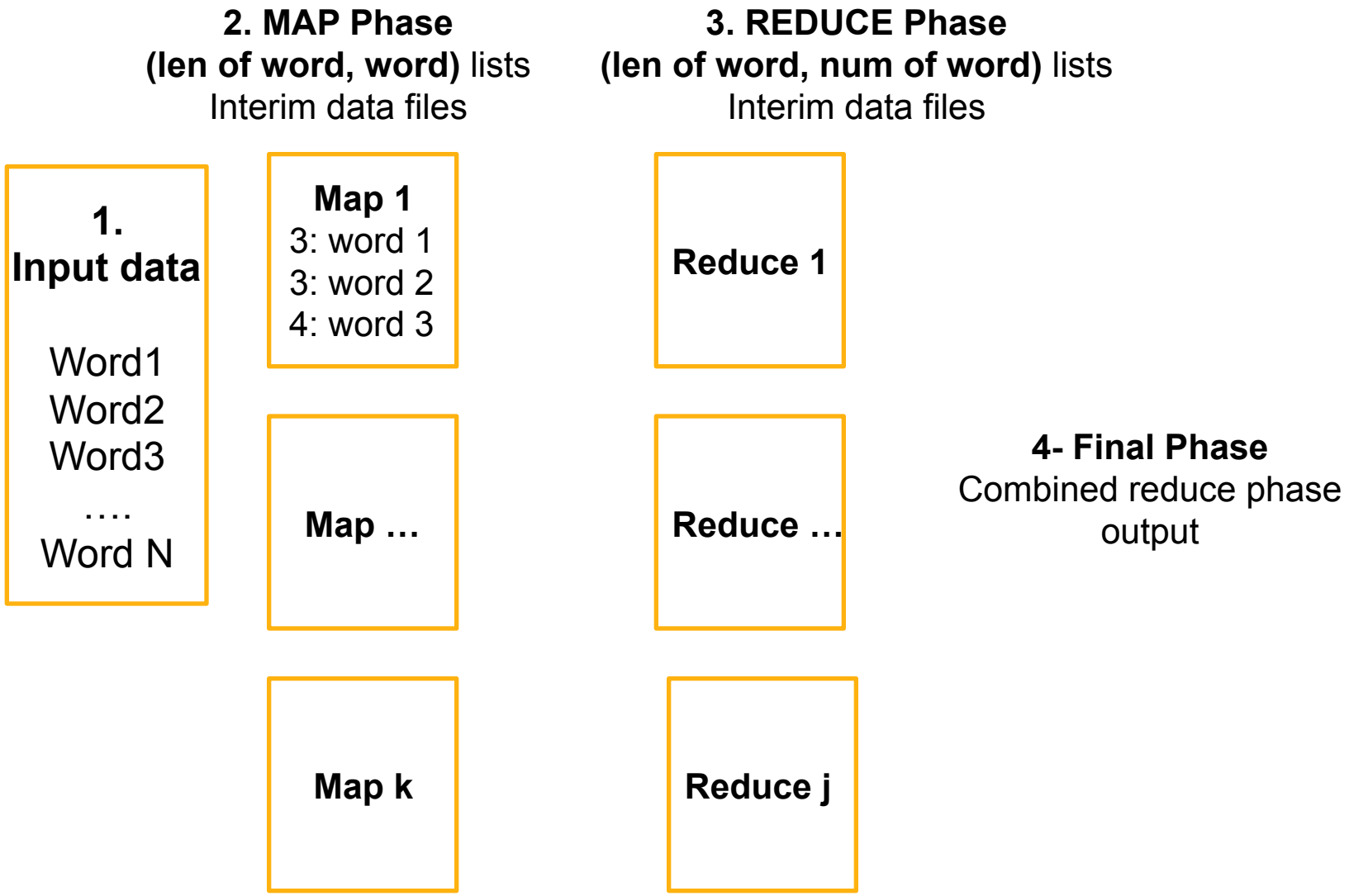
Task Attempts

A given task is attempted at least once and possibly many times if it crashes

If a task crashes consistently, it will be abandoned eventually

Multiple attempts pertaining to a task may happen in parallel with the **speculative execution** feature

MapReduce example: counting words per word length



MapReduce Programming Model

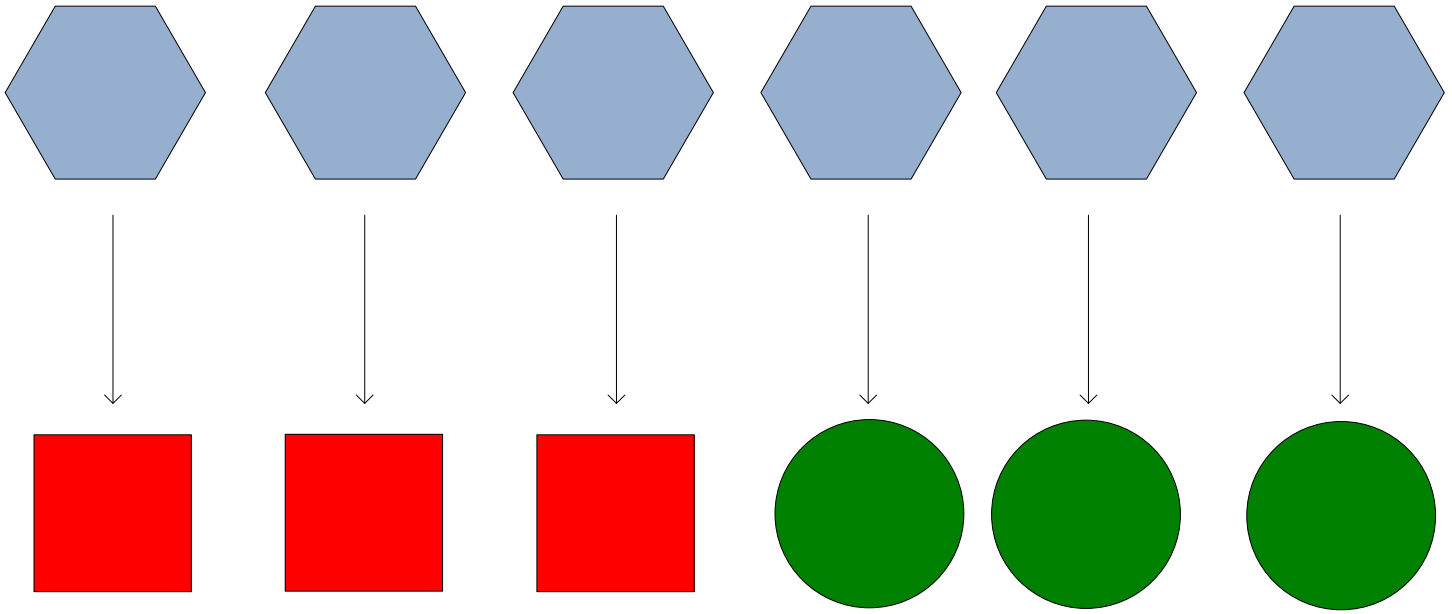
Inspired by functional programming

Two key functions that need to be implemented:

- **map** (in_key, in_value) → (out_key, intermediate_value) list
  - Data source records are fed as key,value pairs
  - map() produces one or more intermediate values with an output key
- **reduce** (out_key, intermediate_value list) → out_value list
  - Intermediate values for given key are combined into a list
  - Reduce() combines those values into one or more final values for the same output key
  - Optional and not needed by all applications

map

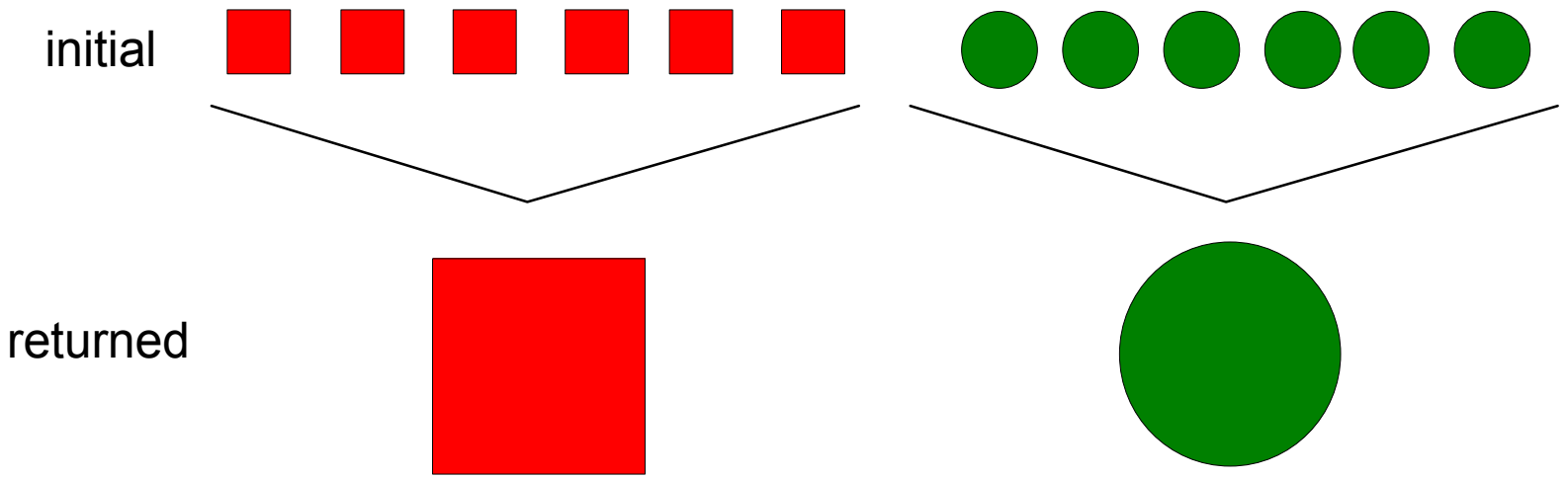
```
map  (in_key, in_value) ->
   (out_key, intermediate_value) list
```



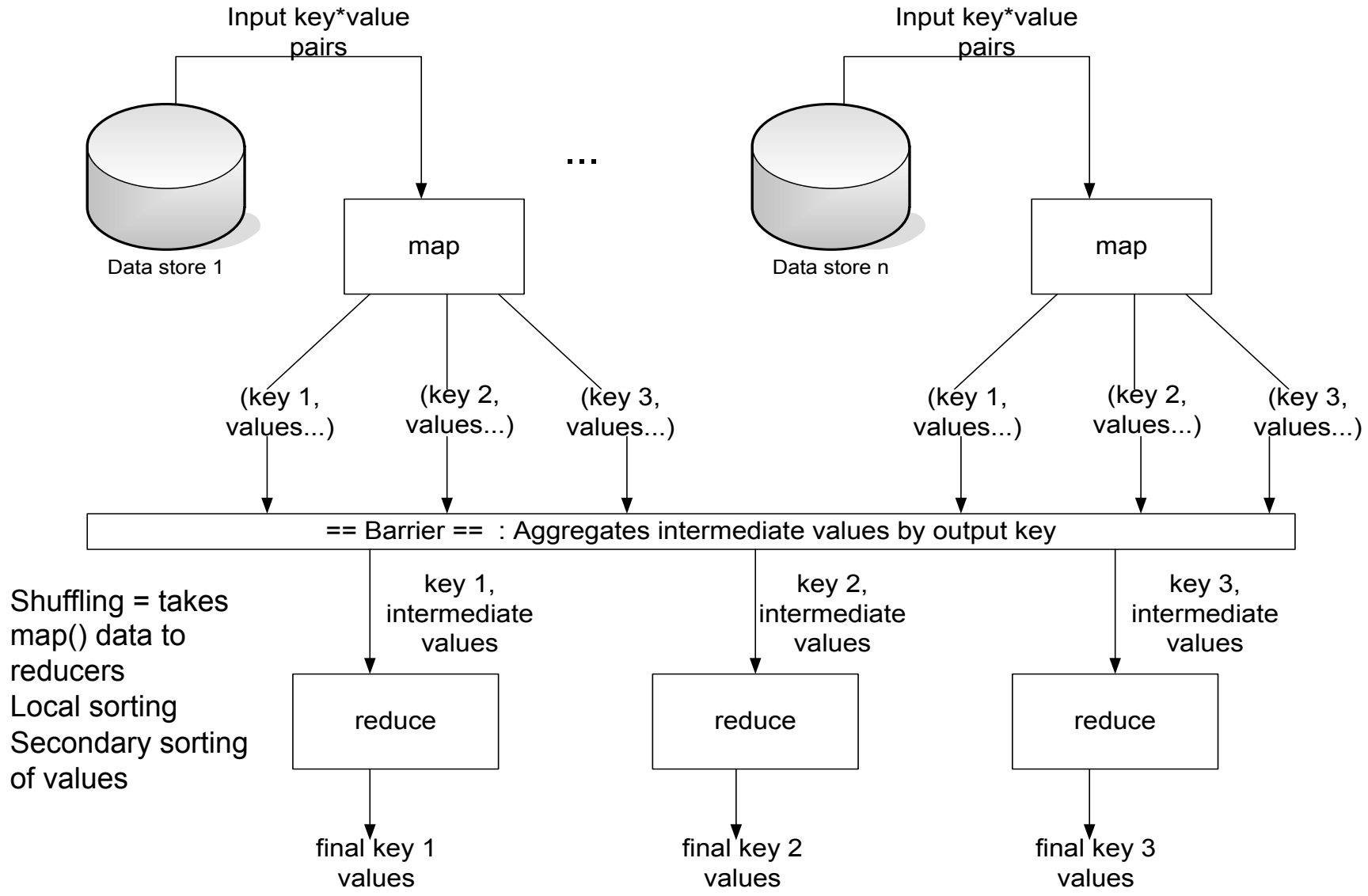
Source: https://courses.cs.washington.edu/courses/cse490h/08au/lectures/mapred.pdf

Reduce

```
reduce (out_key, intermediate_value list) ->
          out_value list
```



Source: https://courses.cs.washington.edu/courses/cse490h/08au/lectures/mapred.pdf



Source: https://courses.cs.washington.edu/courses/cse490h/08au/lectures/mapred.pdf

Reduce Details

- Three key phases
  1. Reducer copies sorted output (based on key) from each Mapper using HTTP
  2. The framework sorts Reducer inputs by keys, because different Mappers have emitted the same key. The shuffle and sort phases happen simultaneously. SecondarySort can be used to sort values.
  3. Reduce is applied for each key in the sorted inputs

Reducer output is not sorted.

Example: Count word occurrences

```
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<int>
  intermediate_values):
  // output_key: a word
  // output_values: a list of counts
  int result = 0;
  for each v in intermediate_values:
    result += v;
 Emit(result);
```

Source: https://courses.cs.washington.edu/courses/cse490h/08au/lectures/mapred.pdf

MapReduce Parallelism

- Both map() and reduce() operations are executed in parallel
- Map() creates intermediate values from input
- Reduce() functions operate on different output keys
- Values are independently processed

- Note that reduce cannot start before map is finished
- Secondary sort on values: the application can extend key with a secondary key and define a grouping comparator

Locality

The map() task input data is divided into GFS/HDFS blocks

Master program distributes tasks based on the location of data

Map() tasks should be run on the same machine as the physical data file (or the same rack)

Fault Tolerance

Fault tolerance is realized by periodic heartbeats

Master pings worker nodes to detect node failures

Master must re-execute tasks that have failed

Master can notices if particular input key/values cause problems in map and can skip those

Node failures

In the worst-case scenario the master compute node fails causing the job to be restarted

Other failures can be managed by the master

Failure of a worker node requires that the task is assigned to another worker

Master reschedules failed tasks when workers become available

Optimizations

Reduce tasks cannot start before the whole map phase is complete

Thus single slow machine can slow down the whole process

Master can execute many redundant map tasks and then use the results of the first task to complete

Optimizations: Combining Phase

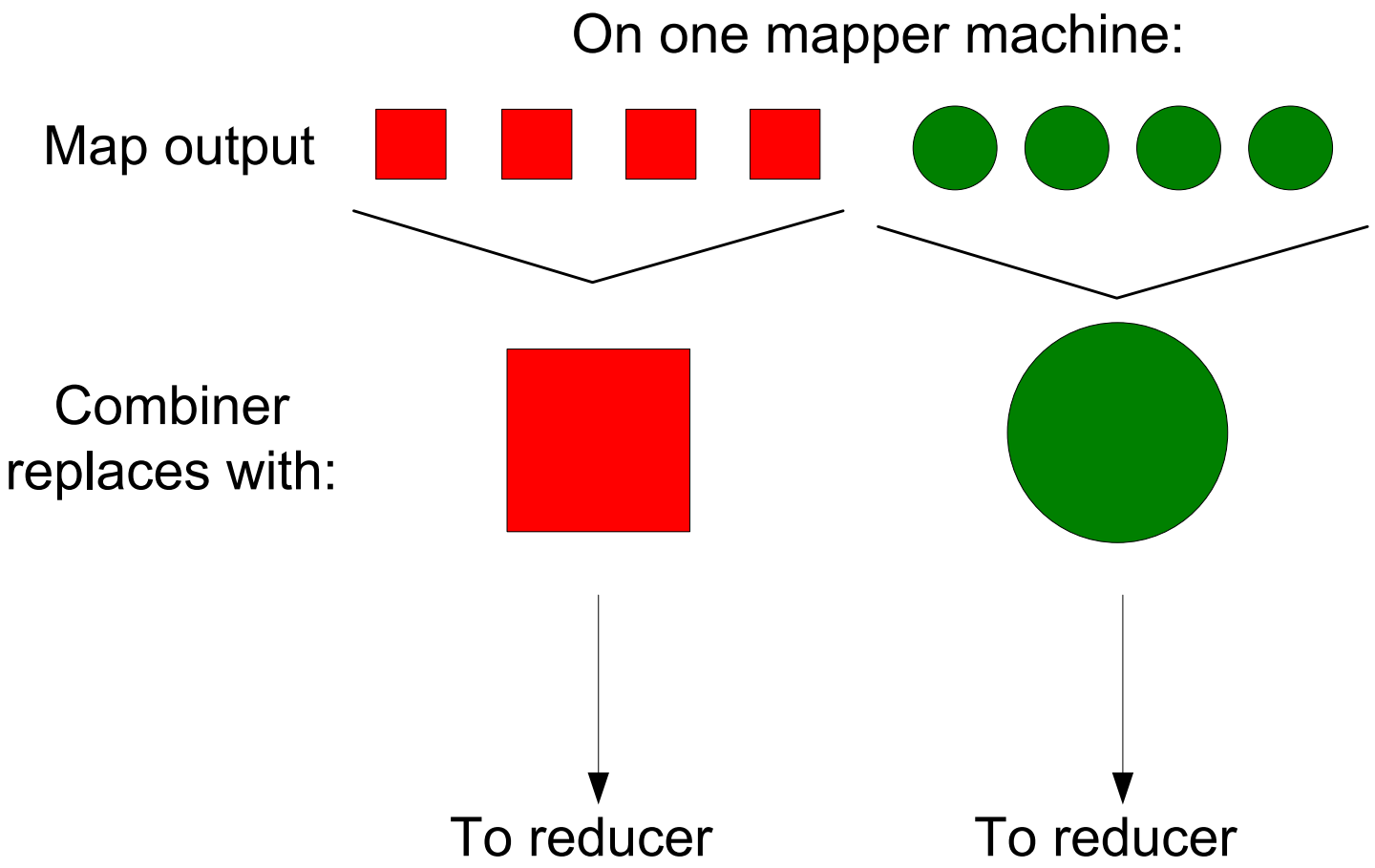
Performance can be increased by running a mini reduce phase on local map output

Executed on mapper nodes after map phase

Saves bandwidth before sending data to a full reducer

Reducer can be a combiner if it is commutative and associative

Combiner, graphically



Source: https://courses.cs.washington.edu/courses/cse490h/08au/lectures/mapred.pdf

Partitioner

Partitioner divides the **intermediate key space**

Assigns intermediate key-value pairs to reducers

Thus **n partitions results in n reducers**

Between map and reduce phases:

- data is shuffled: parallel-sorted and exchanged
- data is moved to the correct shard for reducing
- partition function accepts the key and the number of reducers and then returns the index of the reducer

Supports load balancing

MapReduce Summary

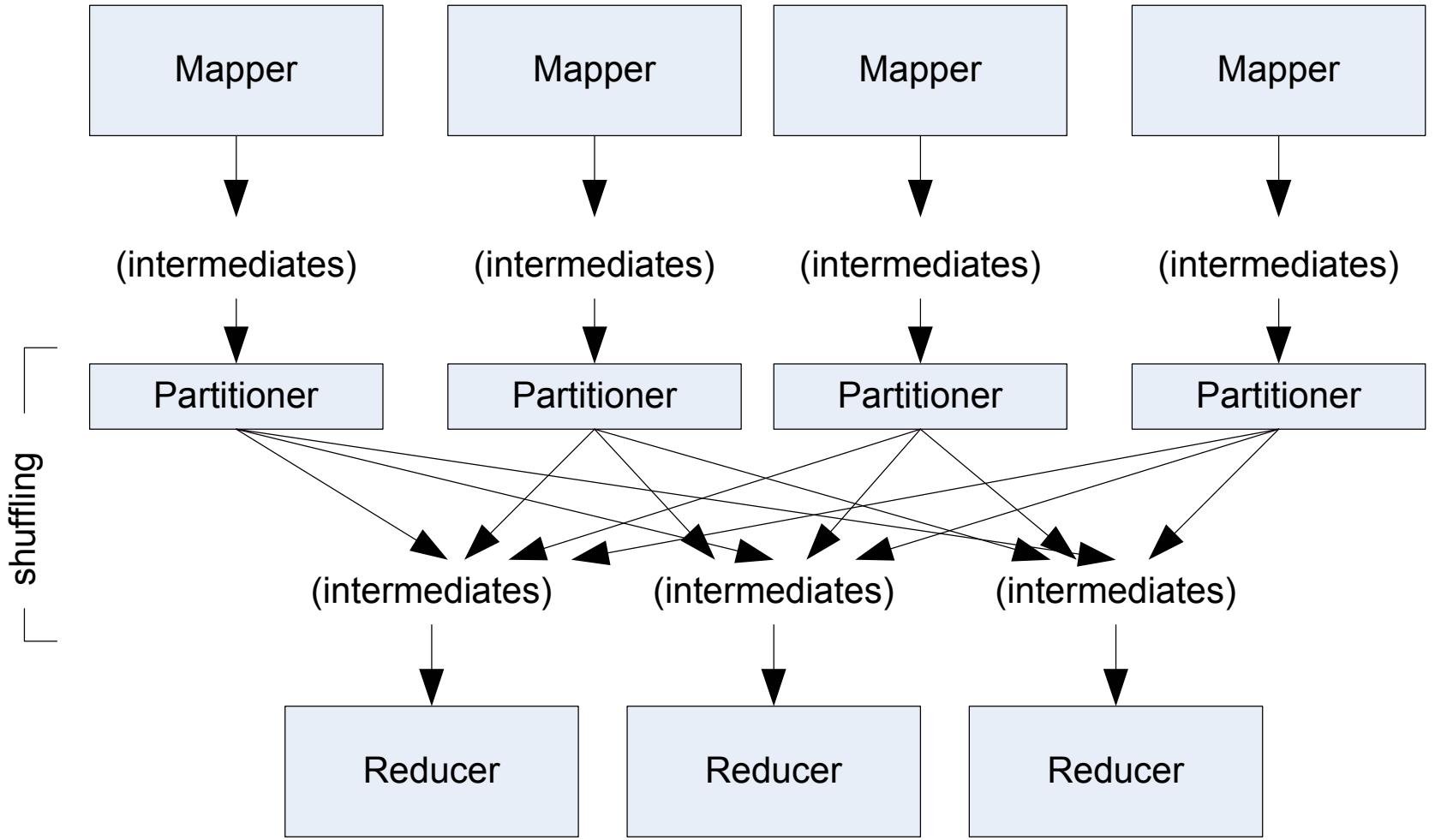
Two key functions that need to be implemented:

- **map** (in_key, in_value) → (out_key, intermediate_value) list
- **reduce** (out_key, intermediate_value list) → out_value list

With two optimizations:

- **combine** (key, intermediate_value list) → intermediate_out_value list
- **partition** (key, number of partitions) → partition for key

Partition and Shuffle



Synchronization

Intermediate key-value pairs must be grouped by key through a distributed sort

Shuffle and sort

A job with m mappers and r reduces involves up to m * r different copy operations

Each mapper may have intermediate output destined to every reducer

Data management as part of MapReduce

Purlieus and CAM

Data management as part of MapReduce

Need to know task types (map/reduce intensive)

Map intensive jobs benefit from locality awareness

Up to 80% reduction in execution time

Key algorithms for MapReduce

Inverted Index

Sorting

PageRank

Sorting

Searching

Statistics

- Average, SD, count

Advanced algorithms

Filtering Algorithms

Finding files or items with specific characteristics

Searching for patterns in web logs or files

Filtering is mostly done in the map phase

Reduce can be simply the identity

Aggregation Algorithms

Computing the minimum, maximum, sum, average, ... of the given values

Count the number of Tweets per day

Map is simple or the identity, reduce is doing most of the work

Sorting

The Mapper output is automatically sorted by keys

Reducer plays a crucial role in sorting

Terasort Benchmark

http://sortbenchmark.org/YahooHadoop.pdf

(key, value) pairs are handled in order by key and sent to a specific reducer based on hash(key)

Hash function must be chosen so that

k1 < k2 => hash(k1) < hash(k2)

If we have a single reducer, the output is sorted

If we have multiple reducers, we get partially sorted results: last-stage merge of the interim results

http://hadooptutorial.wikispaces.com/Sorting+feature+of+MapReduce

TeraSort

TeraSort is a standard map/reduce sort

A custom partitioner that uses a sorted list of N − 1 sampled keys that define the key range for each reduce.

In particular, all keys such that sample[i − 1] <= key < sample[i] are sent to reduce i.

This guarantees that the output of reduce i are all less than the output of reduce i+1.

To speed up the partitioning, the partitioner builds a two level trie that quickly indexes into the list of sample keys based on the first two bytes of the key.

TeraSort generates the sample keys by sampling the input before the job is submitted and writing the list of keys into HDFS.

K-Means Clustering Algorithm

Iterative algorithm that is run until it converges

1. K initial points (centers) are chosen at random.
2. K clusters are formed by associating every data point (observation) with the nearest center.
3. For each cluster, recompute the centers (determine centroid)
4. Repeat from 2 until convergence.

Source: Riccardo Torlone. Analytics on Big Data. Universita Roma Tre.

K-Means for MapReduce

Map phase

- Each map reads the K centroids and a block from the input dataset
- Each point is assigned to the closest centroid
- Output: <centroid, point>

Reduce phase

- Obtain all points for a given centroid
- Recompute the new centroid
- Output: <new centroid>

Iteration:

- Compare the old and new set of K centroids
- If they are similar then Stop
- Else Start another iteration unless maximum of iterations has been reached.

Optimizing K-Means for MapReduce

- Combiners can be used to optimize the distributed algorithm
  - Compute for each centroid the local sums of the points
  - Send to the reducer: <centroid, partial sums>
- Use of a single reducer
  - Data to reducers is very small
  - Single reducer can tell immediately if the computation has converged
  - Creation of a single output file

Limitation: iterative algorithms

MapReduce tasks must be written as acyclic dataflow programs

- Stateless mappers and reducers
- Batch model

Difficult to implement iterative processing of datasets

- Machine learning typically requires iterative operation: the dataset is visited multiple times by the algorithm