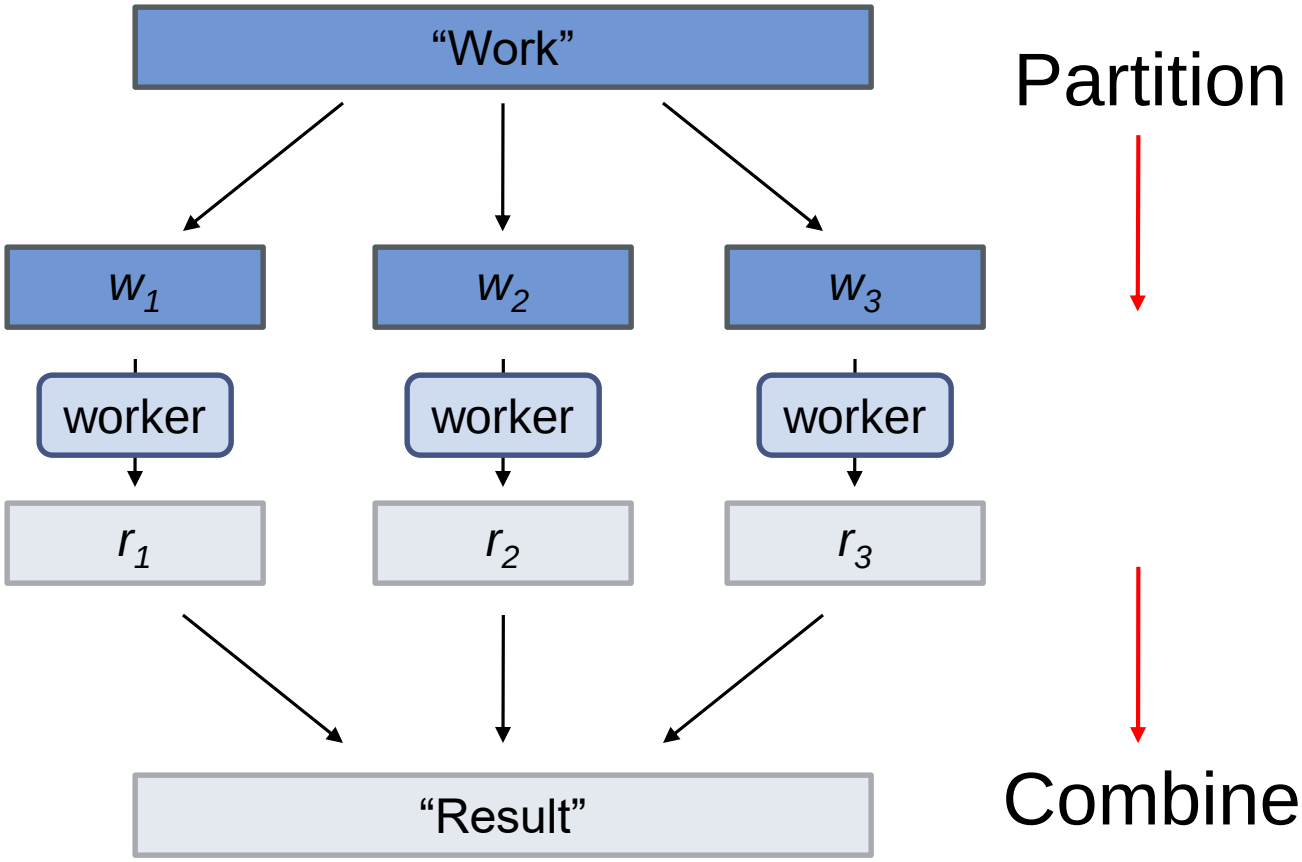
Introduction to MapReduce

Prof. Dr. Stephan Trahasch
Hochschule Offenburg

Divide and Conquer



Parallelization Challenges

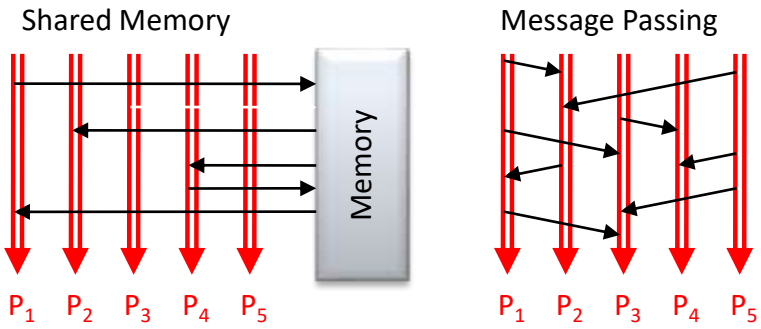
- How do we assign work units to workers?
- What if we have more work units than workers?
- What if workers need to share partial results?
- How do we aggregate partial results?
- How do we know all the workers have finished?
- What if workers die?

What's the common theme of all of these problems?

Current Concepts

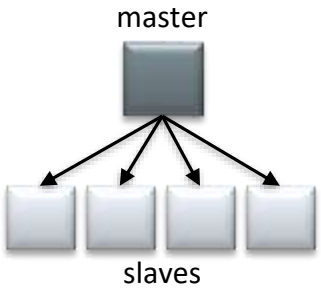
Programming models

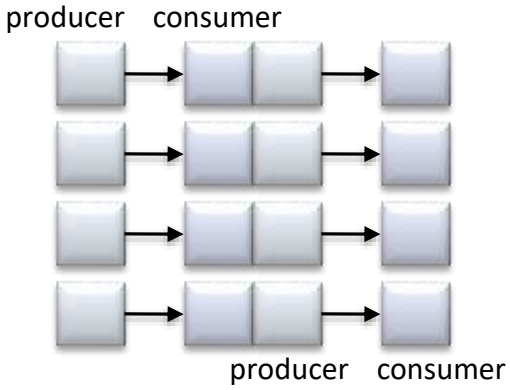
- Shared memory
- Message passing (MPI)

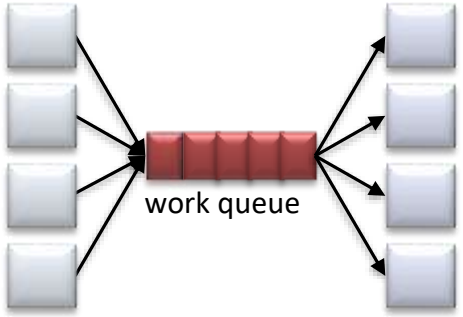


Design Patterns

- Master-slaves
- Producer-consumer flows
- Shared work queues







New Concepts

- Scale “out”, not “up”
  - Limits of SMP and large shared-memory machines
- Move processing to the data
  - Cluster have limited bandwidth
- Process data sequentially, avoid random access
  - Seeks are expensive, disk throughput is reasonable
- Seamless scalability
  - From the mythical man-month to the tradable machine-hour

MapReduce

- Programming model for processing large unstructured or semi-structured data sets
- Map and Reduce are concepts of functional programming languages such as LISP or Scheme
- MapReduce framework distributes the calculations to several processing units
  - Parallel execution on multiple computers
  - After the calculations have been completed, the framework aggregates the results
- Prerequisites
  - It must be possible to partition data.
  - Calculation can be performed on one partition independently of another.
  - Data is distributed and stored in HDFS

Scenarios where MapReduce should not be used

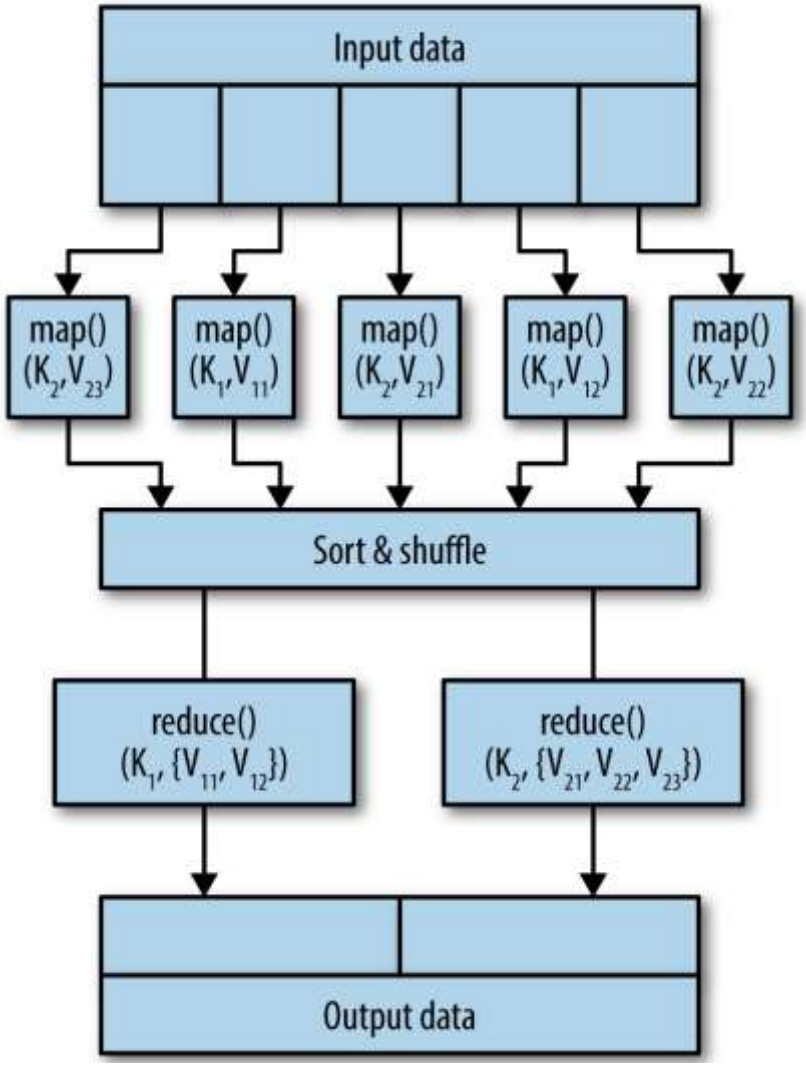
- If the computation of a value depends on previously computed values.
  Example Fibonacci series: F(k + 2) = F(k + 1) + F(k)
- If the data set is small enough to be computed on a single machine. It is better to do this as a single reduce(map(data)) operation rather than going through the entire MapReduce process.
- If synchronization is required to access shared data.
- If all of your input data fits in memory.
- If one operation depends on other operations.
- If basic computations are processor-intensive.

Map and Reduce – Key Idea

1. **Map**
   Input: (key, value)
   Output: zero or more (Key, Value) pairs
   $Map(K_1,V_1)$ -> $list(K_1',V_1')$

2. **Reduce**
   All values for a particular intermediate key go to same Reducer.
   Intermediate keys/value lists are passed in sorted key order.

   $Reduce(K_1',list(V_1''))->(K,V)$



K = Key, V = Value

Hello World: Word count

```
Map(String docid, String text):
      for each word w in text:
            Emit(w, 1);


Reduce(String term, Iterator<Int> values):
      int sum = 0;
      for each v in values:
            sum += v;
            Emit(term, value);
```

MapReduce

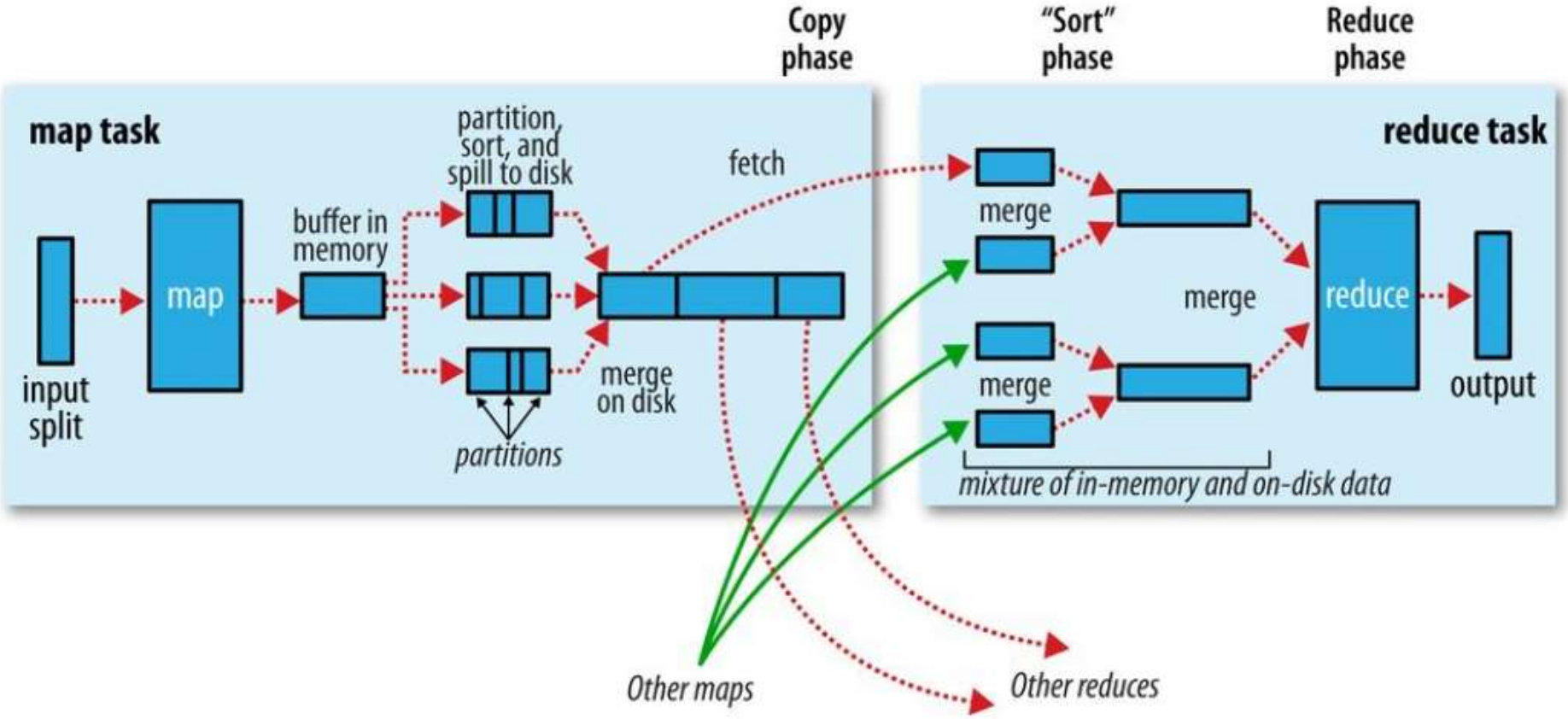
1. **Map**
   - Mappers map input data to intermediate key/value pairs
     parse, filter, or transform the data
   - Each Map task operates on a single HDFS block
   - run on the node where the block is stored

2. **Shuffle and Sort**
   - Sorts and consolidates intermediate data from all mappers
   - Happens after all Map tasks are complete and before Reduce tasks start → place for optimization

3. **Reduce**
   - Reducers process Mapper output into final key/value pairs
     aggregate data using statistical functions
   - Operates on Map task output (shuffled/sorted intermediate data)

The execution framework handles everything else…

Details zu 2. Shuffle and Sort



Quelle: Hadoop – The Definitive Guide

Map: Buffer and Combiner

- Each map task has circular memory buffer that it writes output to. Default buffer size is 100 MB.
- If buffer size reaches a threshold size (default 80%)
  → background thread starts to spill the contents to disk
- Spills are written in round-robin fashion to directories in a job-specific subdirectory
- Before it writes to disk, thread divides the data into partitions corresponding to the reducers that they will be sent to. Within each partition, the thread performs an in-memory sort by key.
- Combiner Function
  If there is a combiner, it is run on output of sort. Running the combiner function makes for a more compact map output. There is less data to write to local disk and to transfer to the reducer.

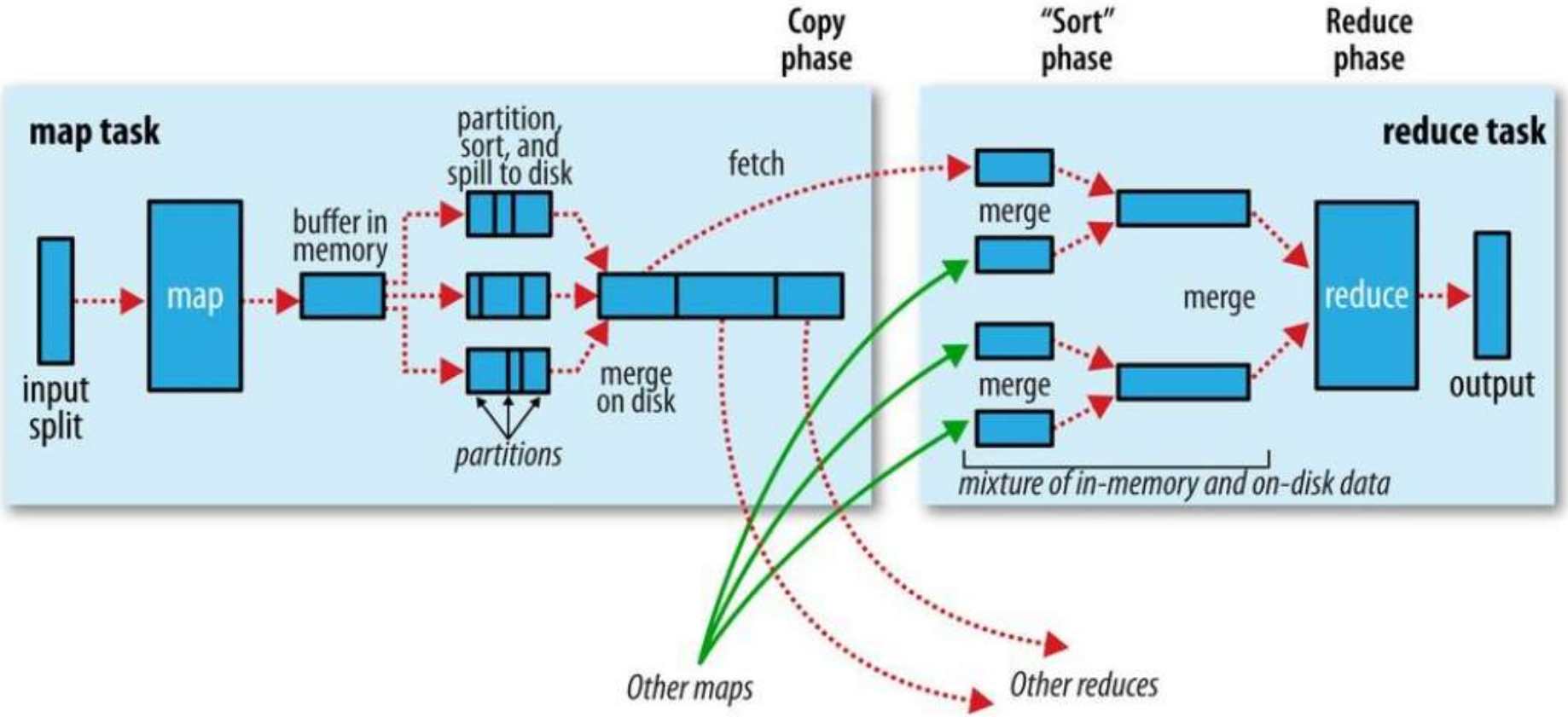
Map: Spill

- Each time the memory buffer reaches the spill threshold, a new spill file is created.
- After the map task has written its last output record, there could be several spill files.
- Before the task is finished, the spill files are merged into a single partitioned and sorted output file.
- Per default 10 files are merged at once.
- If there are at least three spill files the combiner is run again before the output file is written.
- Combiners may be run repeatedly over the input without affecting the final result.

Map: Compression

Compressing the map output as it is written to disk

→ writing is faster, saves disk space, reduces amount of data to transfer to the reducer.



Reduce Part of Shuffle & Sort

- Map output file is sitting on the local disk of the “Map Machine”
  - Map outputs always written to local disk,
  - Reduce outputs may not be.
- Data of partition is needed by the “Reduce Machine”
- Moreover, reduce task needs the map output from several map tasks across the cluster!
- Map tasks may finish at different times.
- Reduce task starts copying the outputs of Map tasks as soon as each completes: “copy phase of the reduce task”.
- Reduce task has a small number of copier threads so that it can fetch map outputs in parallel. Default 5 threads.

How do reducers know which machines to fetch map output from?

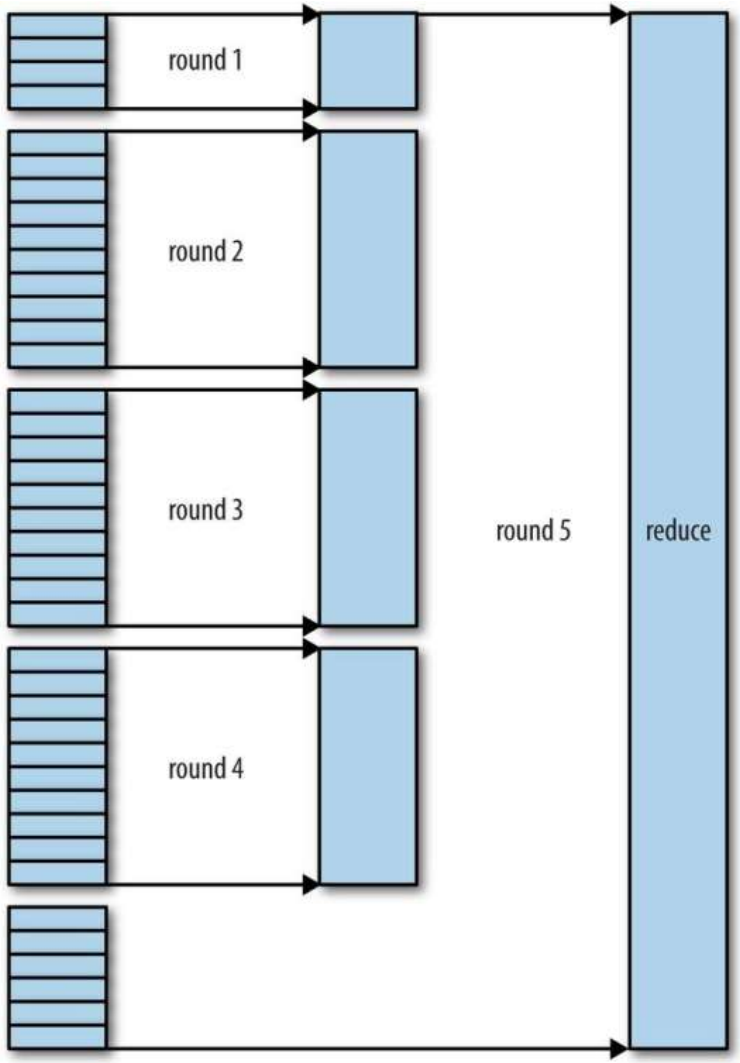
- As map tasks complete successfully, they notify their application master using the heartbeat mechanism.
- For a given job, the application master knows the mapping between map outputs and hosts.
- A thread in the reducer periodically asks master for map output hosts until it has retrieved them all.

- Hosts do not delete map outputs from disk as soon as the first reducer has retrieved them.
- Reducer may subsequently fail!
- Instead, they wait until they are told to delete them by the application master, which is after the job has completed.

Copy Phase of the Reduce Task

- Map outputs are copied to reduce task JVM's memory if they are small enough; otherwise, they are copied to disk.
- When the in-memory buffer reaches a threshold size or reaches a threshold number of map outputs
  → then it is merged and spilled to disk.
- If a combiner is specified, it will be run during the merge to reduce the amount of data written to disk.
- As the copies accumulate on disk, a background thread merges them into larger, sorted files.
- This saves some time merging later on.

Sort Phase



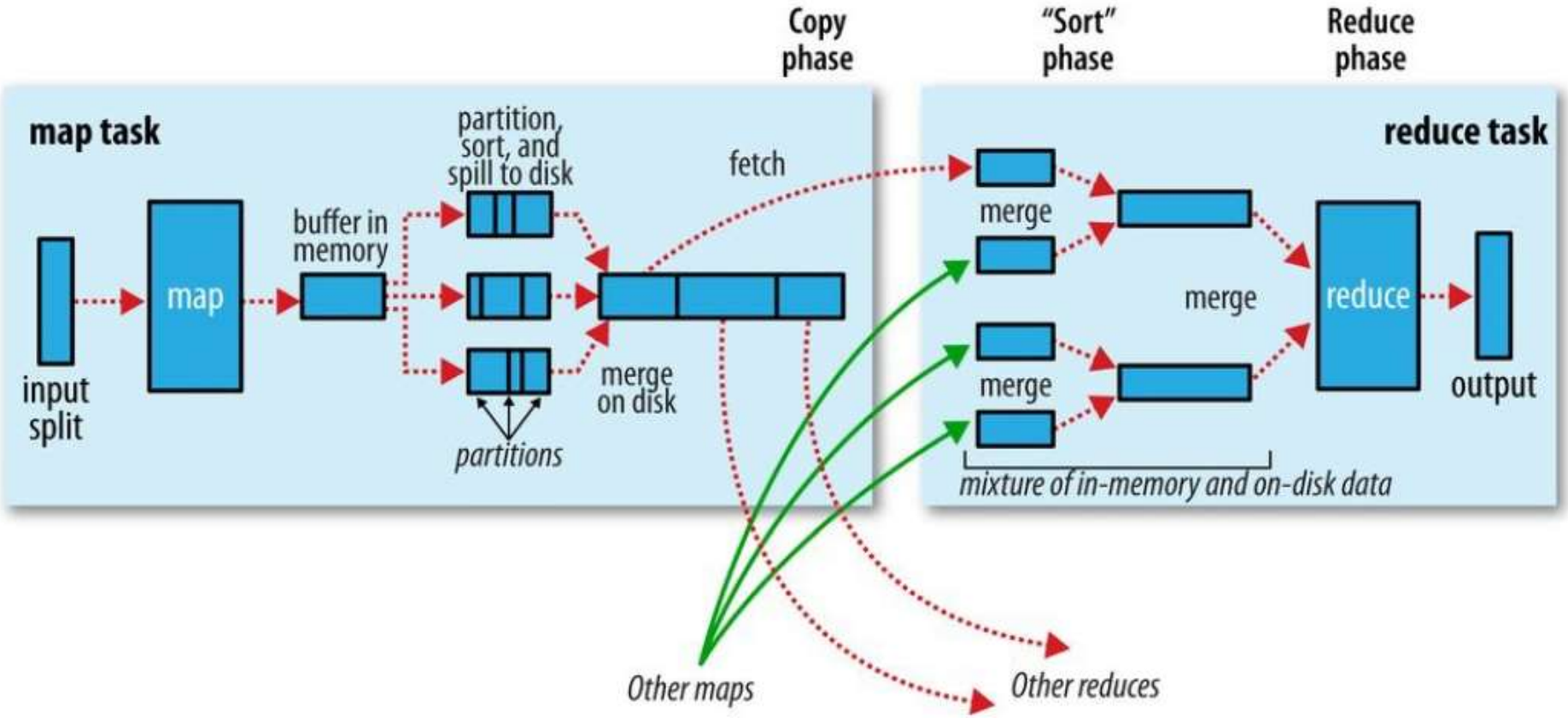
When all the map outputs have been copied, the reduce task starts the sort phase.

Merges map outputs, maintaining their sort ordering.

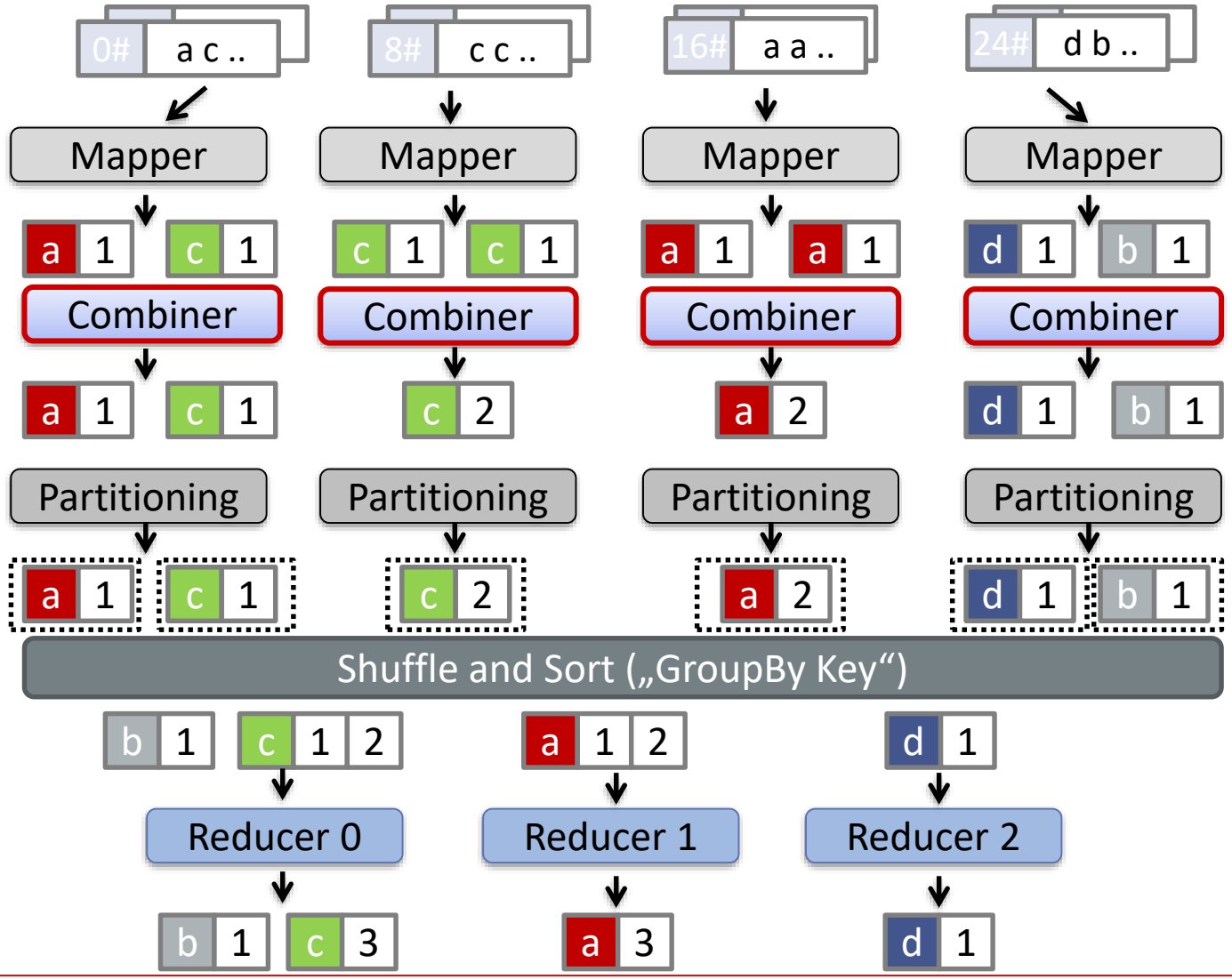
This is done in rounds.

Example, if there were 50 map outputs and the merge factor was 10, there would be five rounds. Each round would merge 10 files into 1, so at the end there would be 5 intermediate files.

Overview


Copy phase
"Sort" phase
Reduce phase
map task
reduce task
input split
map
buffer in memory
partition, sort, and spill to disk
partitions
merge on disk
fetch
merge
merge
merge
reduce
output
mixture of in-memory and on-disk data
Other maps
Other reduces

MapReduce Optimization: Combiner



Key: Byteoffset
Value: Textzeile

Key: Wort
Value: Anzahl

Local Aggregation

hash(key) mod R
(example R=3)

Barrier

Aggregation

Key: Wort
Value: Count

Mapper

```
public static class Map extends
Mapper<LongWritable, Text, Text, IntWritable> {
  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();

  public void map(LongWritable key, Text value, Context
context) throws IOException, InterruptedException {
    String line = value.toString();
    StringTokenizer tokenizer = new StringTokenizer(line);
    while (tokenizer.hasMoreTokens()) {
       word.set(tokenizer.nextToken());
       context.write(word, one);
    }
  }}
```

Reducer

```
public static class Reduce extends
  Reducer<Text, IntWritable, Text, IntWritable> {

public void reduce(Text key, Iterable<IntWritable> values,
  Context context) throws IOException, InterruptedException{
    int sum = 0;
    for (IntWritable val : values) {
       sum += val.get();
    }
    context.write(key, new IntWritable(sum));
  }
}
```

Ensure Types Match

Mappers and Reducers declare input and output type parameters. These must match the types used in the class. Output types must also match those set in the driver

```
public class WordMapper extends Mapper<LongWritable, Text,
Text, IntWritable>
{
     @Override
       public void map(LongWritable key, Text value,
          Context context)  throws IOException,
          InterruptedException {
    …
          context.write(new Text(word), new IntWritable(1));
    …
  }
}
```

Input key and value types

Output key and value types

Driver runs on the client machine
It configures the job, then submits it to the cluster

```
public class WordCount {
  public static class Map … {}
  public static class Reduce … {}
  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = new Job(conf, "wordcount");
    job.setJarByClass(WordCount.class);

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.setMapperClass(Map.class);
    job.setReducerClass(Reduce.class);
```

Driver runs on the client machine
It configures the job, then submits it to the cluster

```
    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(IntWritable.class);

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);

    boolean success = job.waitForCompletion(true);
    System.exit(success ? 0 : 1);

}
}
```

`job.waitForCompletion()`
Blocks (waits for the job to complete before continuing)

`job.submit()`
Does not block (driver code continues as the job is running)

Creating a New Job Object

- Job class allows you to set configuration options for your MapReduce job
  - The classes to be used for your Mapper and Reducer
  - The input and output directories
  - Many other options
- Any options not explicitly set in your driver code will be read from your Hadoop configuration files
  - Usually located in /etc/hadoop/conf
- Any options not specified in your configuration files will use Hadoop's default values
- You can also use the Job object to submit the job, control its execution, and query its state

Confguring the Job: Determining Which Files To Read

- By default, `FileInputFormat.setInputPaths()` will read all files from a specified directory and send them to Mappers
- Exceptions: items whose names begin with a period (.) or underscore (_)
- Globs can be specified to restrict input
  For example, /2017/*/01/*
- Alternatively, FileInputFormat.addInputPath() can be called multiple times, specifying a single file or directory each time

Configuring the Job: Specifying Final Output With OutputFormat

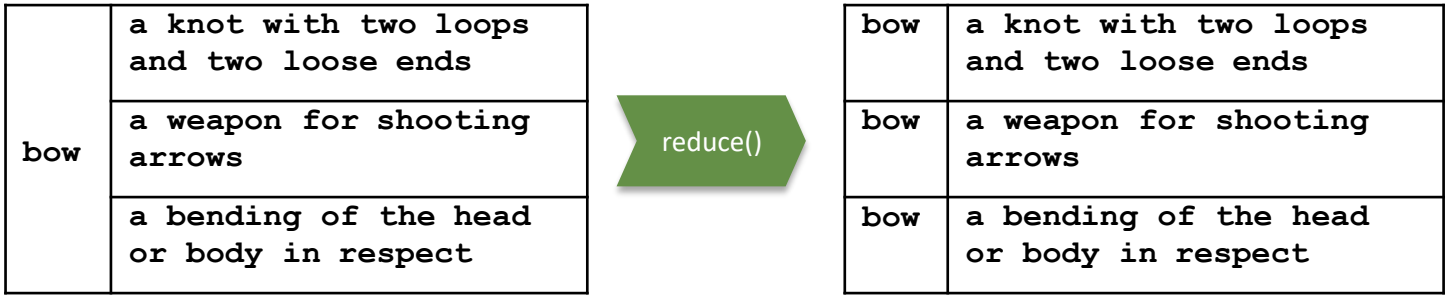
- `FileOutputFormat.setOutputPath()` specifies the directory to which the Reducers will write their final output
- The driver can also specify the format of the output data
- Default is a plain text file
- Could be explicitly written as `job.setOutputFormatClass(TextOutputFormat.class)`

Default Mapper and Reducer Classes

- Setting the Mapper and Reducer classes is optional
- If not set in your driver code, Hadoop uses its defaults

IdentityMapper

IdentityReducer



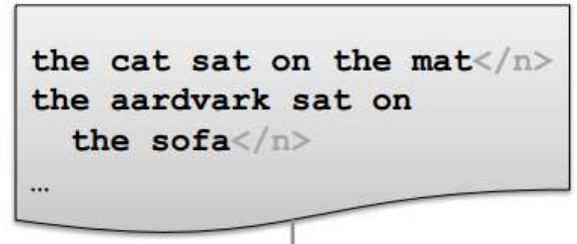
Getting Data to the Mapper

Data passed to the Mapper is specified by an `InputFormat` specified in the driver code

- defines the location of the input data
  Typically a file or directory
- determines how to split the input data into input splits
  each Mapper deals with a single input split
- Creates a RecordReader object
  RecordReader parses the input data into key/value pairs to pass to the Mapper

Example: TextInputFormat (default)

- Creates LineRecordReader objects
- Treats each \n-terminated line of a file as a value
- Key is the byte offset of that Line within the file



key	value
0	the cat sat on the mat
23	the aardvark sat on the sofa
52	…
…	…

Quelle: Cloudera

Further InputFormats

- `FileInputFormat`
  Abstract base class used for all file-based InputFormats

- `KeyValueTextInputFormat`
  Maps \n-terminated lines as ‘key [separator] value’
  By default, [separator] is a tab

- `SequenceFileInputFormat`
  Binary file of (key, value) pairs with some additional metadata

- `SequenceFileAsTextInputFormat`
  Similar, but maps (key.toString(), value.toString())

Keys and Values are Objects

- Keys and values in Hadoop are Java Objects. Not primitives.
- Values are objects which implement Writable
- Keys are objects which implement WritableComparable
- The Writable interface makes serialization quick and easy
- Any value's type must implement the Writable interface
- Hadoop defines its own 'box classes' for strings, integers, etc.
  - `IntWritable` for ints
  - `LongWritable` for longs
  - `FloatWritable` for floats
  - `DoubleWritable` for doubles
  - `Text` for strings

What is WritableComparable?

- A WritableComparable is a Writable which is also Comparable
- Two WritableComparables can be compared against each other to determine their 'order'
- Keys must be WritableComparables because they are passed to the Reducer in sorted order
- Note that despite their names, all Hadoop box classes implement both: Writable and WritableComparable
  For example, `IntWritable` is actually a `WritableComparable`

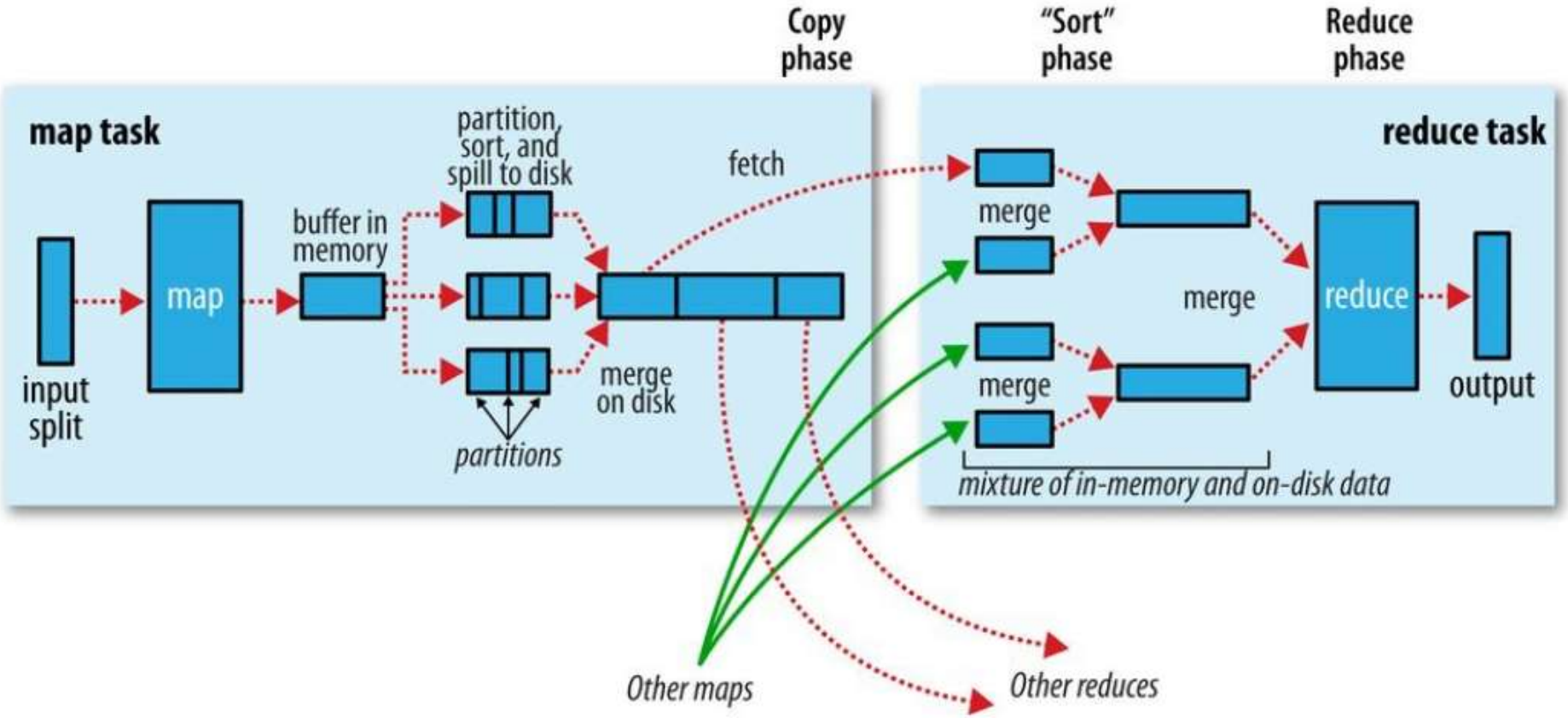
The execution framework handles everything else… What is everything else?

- 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 FS (later)
- Programmers also specify:
  partition (k', number of partitions) → partition for k'
  - Often a simple hash of the key, e.g., hash(k') mod n
  - Divides up key space for parallel reduce operations

  combine (k', v') → <k', v'>*
  - Mini-reducers that run in memory after the map phase
  - Used as an optimization to reduce network traffic

Summary

- MapReduce is not a new concept
- Divide and Conquer



Example

```
hadoop jar hadoop-*-wordcount.jar wordcount <in-dir> <out-dir>
```

Siehe Demo.