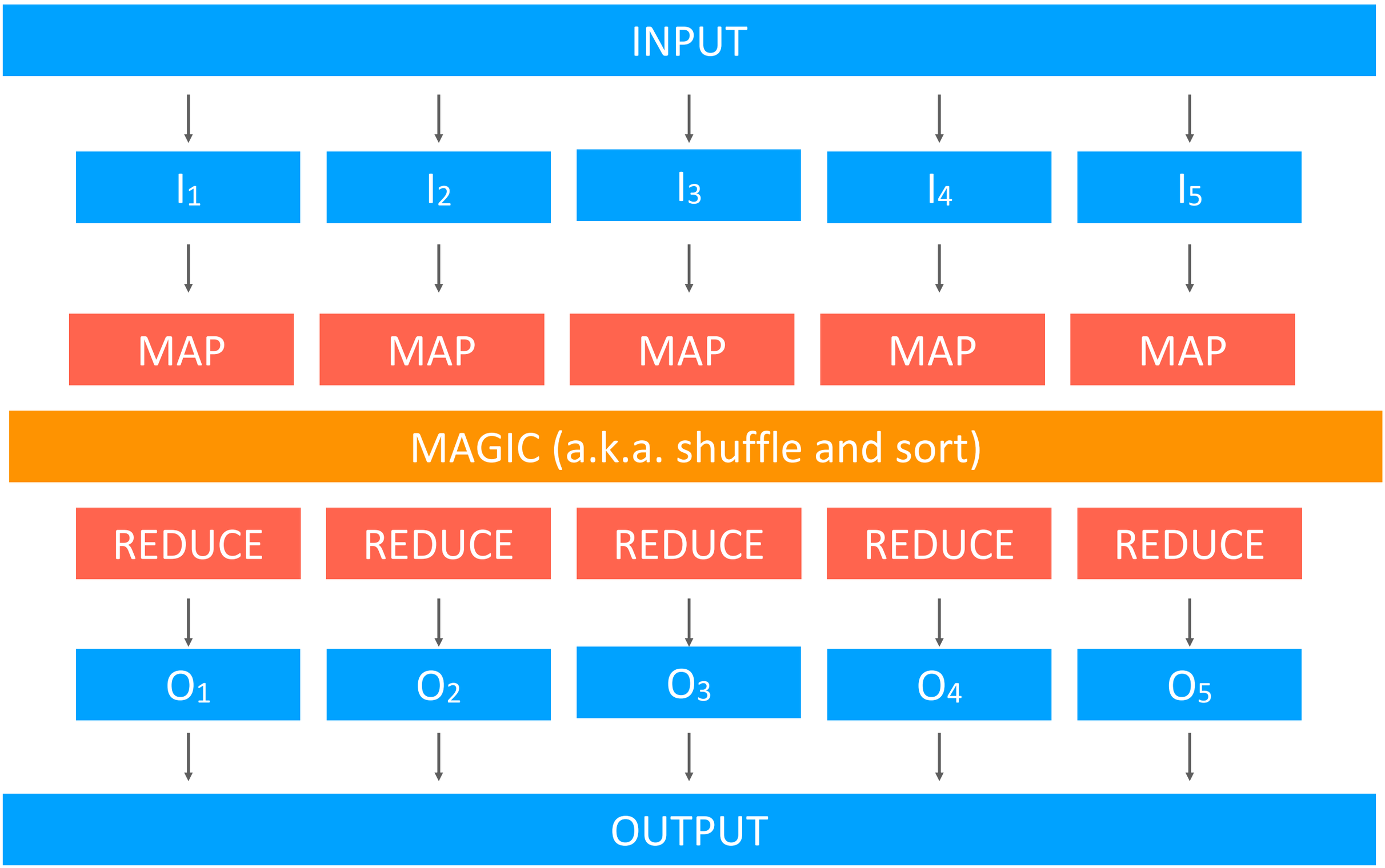
MapReduce Programming Model

- Programmers specify two functions
  - *map function*: from [key, value] (1) to [key, value] (0 or more)
  - *reduce function*: from [key, list of values] (1) to [key, value] (0 or more)
- Map function
  - Receives as input a key-value pair
  - Produces as output a list of key-value pairs (typically 1 or more per input)
  - The function is invoked by the Mapper (function)
- Reduce function
  - Receives as input a key-list of values pair
  - Produces as output a list of key-value pairs (typically none or 1 per input)
  - The function is invoked by the Reducer (function)
- Both functions are STATELESS

Map Reduce Application



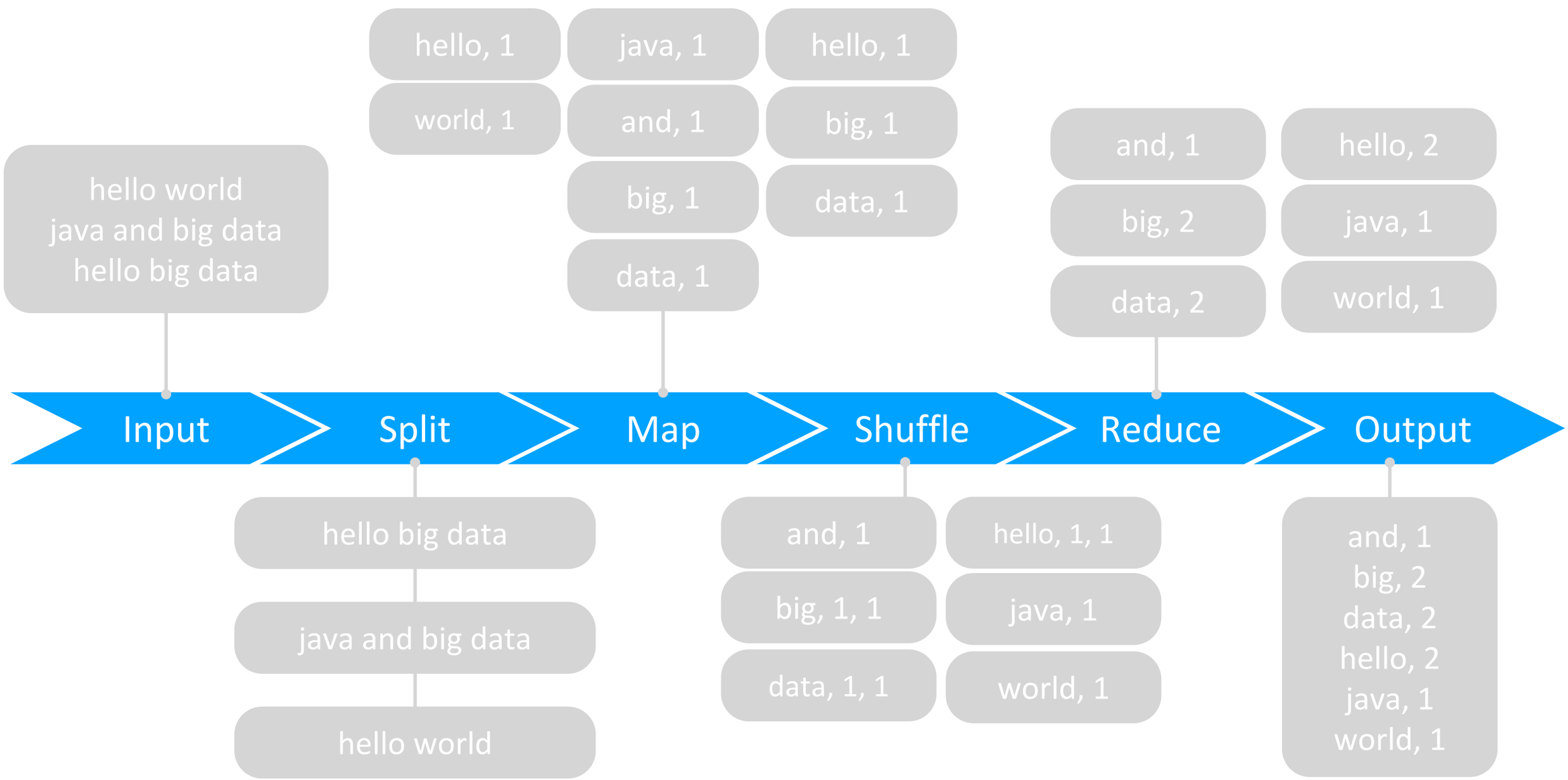
More on mappers

- Mappers should run on nodes which hold their (portion of the) data locally, to avoid network traffic
- Multiple mappers run in parallel, each processing (a portion of) the input data
- The mapper reads in the form of key/value pairs
  - These are read from the distributed file system
  - The mapper may use or completely ignore the input key
  - For example, a standard pattern is to read one line of a file at a time
    - The key is the byte offset into the file at which the line starts
    - The value is the contents of the line itself
    - Typically the key is considered irrelevant
- If the mapper writes anything out, the output must be in the form of key/value pairs

More on reducers

- After the map phase is over, all intermediate values for a given intermediate key are combined together into a list
- Each of these lists is given to a reducer
  - There may be a single reducer, or multiple reducers
  - All values associated with a particular intermediate key are guaranteed to go to the same reducer
  - The intermediate keys, and their value lists, are passed to the reducer in sorted key order
  - This step is known as the '*shuffle and sort*'
- The reducer outputs zero or more final key/value pairs
  - These are written to the distributed file system
  - In practice, the reducer usually emits a single key/value pair for each input key

Wordcount Example (I)



Wordcount Example (II)

```
class MAPPER
method MAP(docid a, doc d)
for all term t in doc d do
EMIT(term t, count 1)
```

```
class REDUCER
method REDUCE(term t, counts [c1, c2,...])
sum ← 0
for all count c in counts [c1, c2,...] do
sum ← sum + c
EMIT(term t, count sum)
```

Exercise: Wordcount Example (III)

- What if we want to compute the word frequency instead of the word count?
  - Input: large number of text documents
  - Output: the word frequency of each word across all documents
  - Note: Frequency is calculated using the total word count

- Hint 1: We know how to compute the total word count
- Hint 2: Can we use the word count output as input?

- Solution: Use two MapReduce computations
  - MR1: count number of all words in the documents
  - MR2: count number of each word and divide it by the total count from MR1

Image Data Example (I)

- Image data from different content providers
  - Different formats
  - Different coverages
  - Different timestamps
  - Different resolutions
  - Different exposures/tones
- Large amount of data to be processed
- Goal: produce data to serve a "satellite" view to users

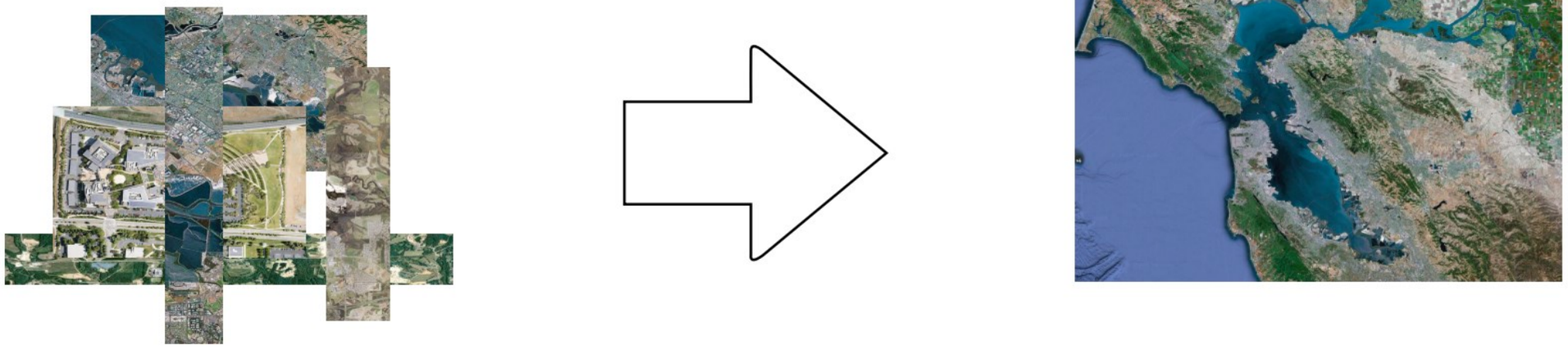
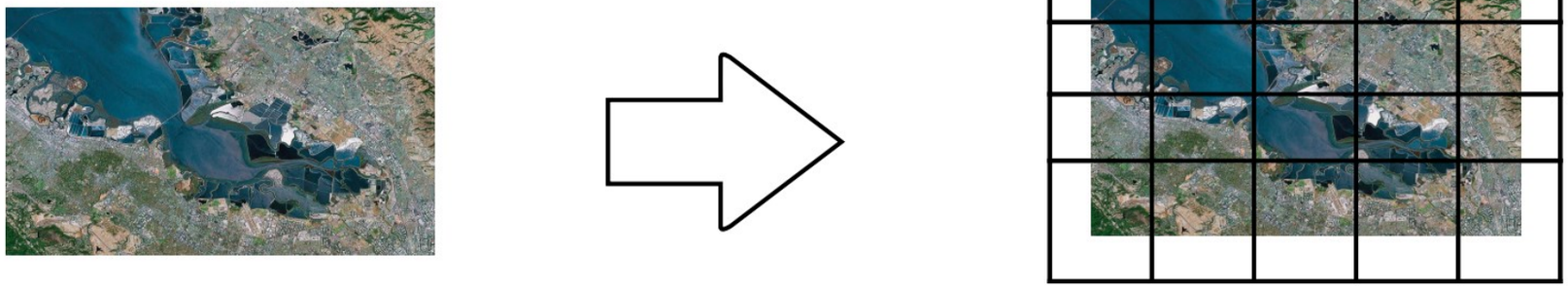
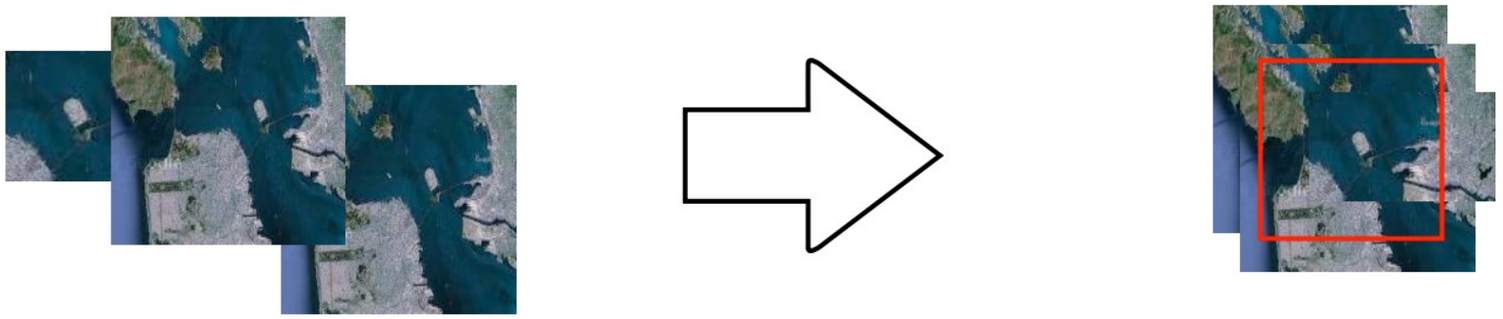


Image Data Example (II)

- Split the whole territory into "tiles" with fixed location IDs
- Split each source image according to the tiles it covers



- For a given tile, stitch contributions from different sources, based on its freshness and resolution, or other preference



- Serve the merged imagery data for each tile, so they can be loaded into and served from an image server farm.

Image Data Example (III)

```
class MAPPER
method MAP(filename path, image data)
tile t
switch image_type(path)
GIF: t ← convert_from_GIF(data)
PNG: t ← convert_from_PNG(data)
...
list<tile> l ← split_in_tiles(t)
for all tile t in list<tile> l do
EMIT(location(t), tile t)
```

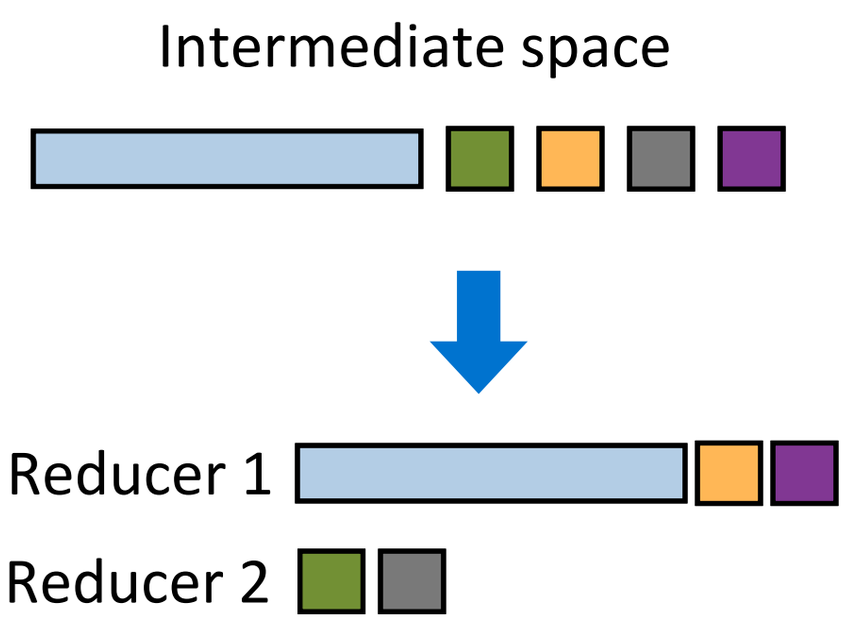
Image Data Example (IV)

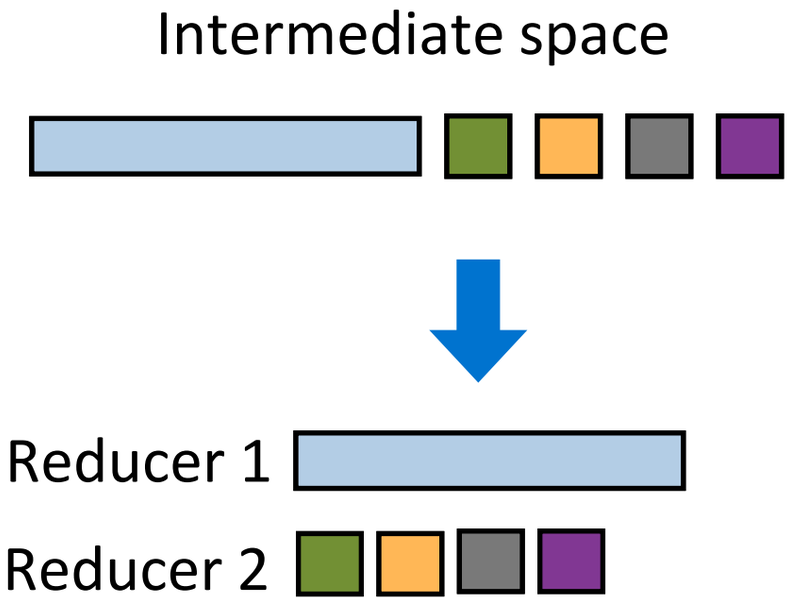
class REDUCER
method REDUCE(position *p*, tiles [t_1, t_2,...])
sort_by_timestamp(t_1, t_2,...)
tile *merged*
for all tile *t* in tiles [t_1, t_2,...] do
merged ← overlay(*merged*, *t*)
merged ← normalize(*merged*)
EMIT(position *p*, tile *merged*)

Partitioners

Balance the key assignments to reducers

- By default, intermediate keys are hashed to reducers
- Partitioner specifies the node to which an intermediate key-value pair must be copied
- Partitions key space for parallel reduce operations
- Partitioner only considers the key and ignores the value

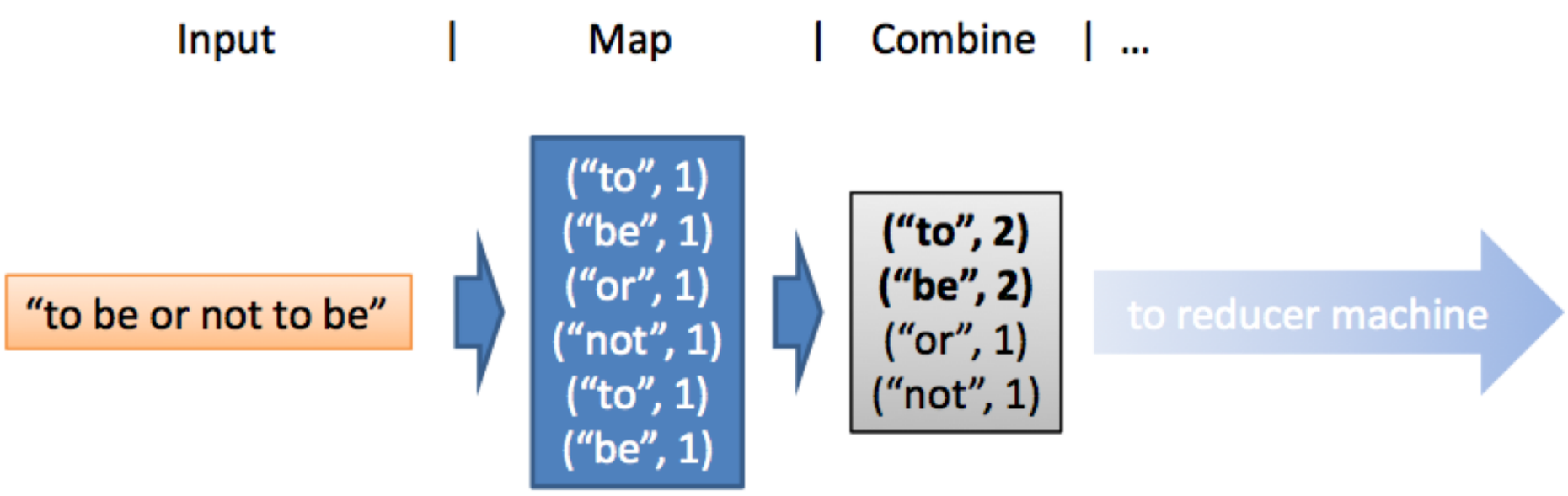




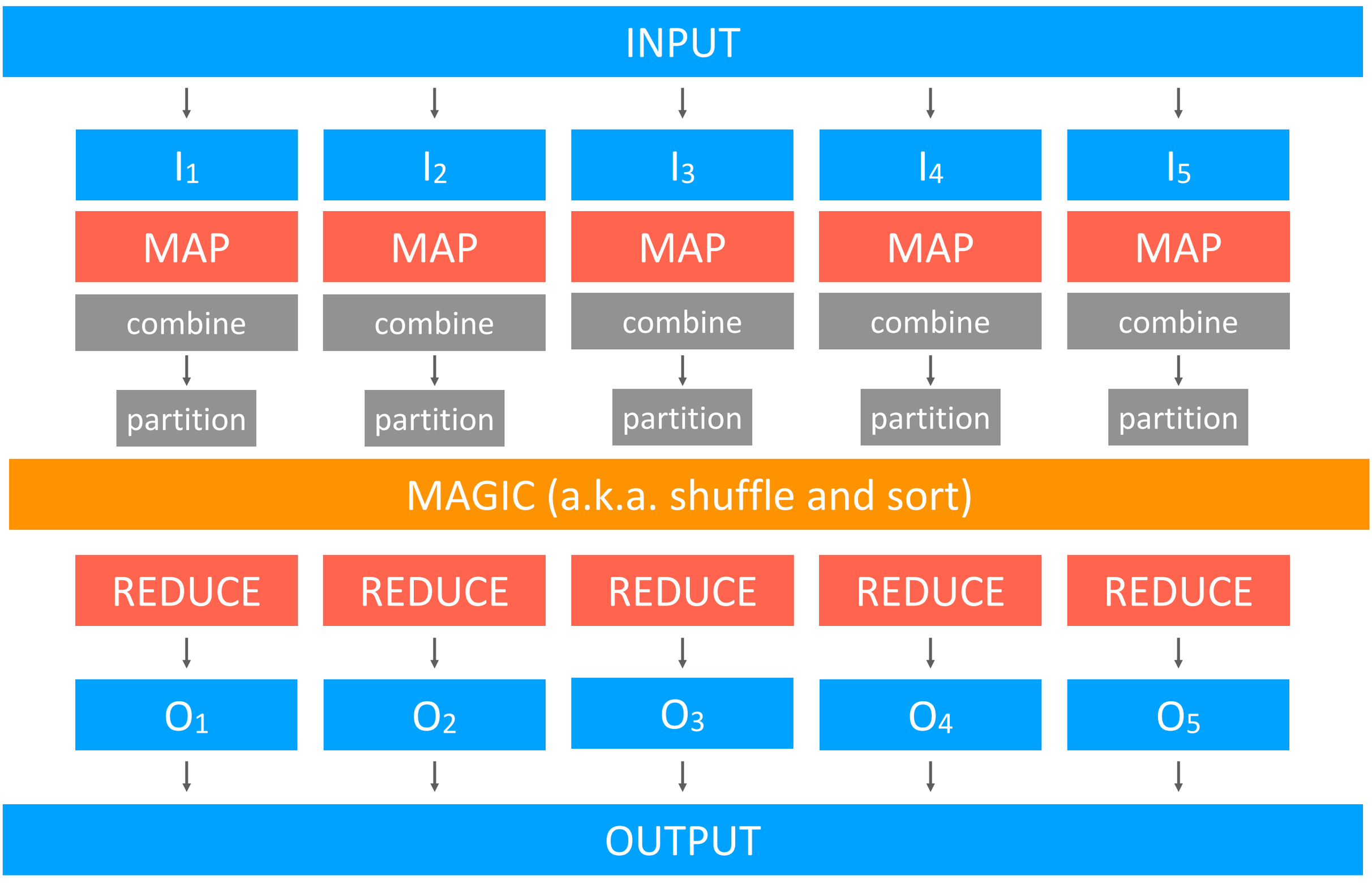
Combiners

Local aggregation before the shuffle

- Combiners are an optimization in Map Reduce
- All the key-value pairs from mappers need to be copied across the network
- The amount of intermediate data may be larger than the input collection itself
- Perform local aggregation on the output of each mapper (same machine)
- Typically, a combiner is a (local) copy of the reducer, a «mini-reducer» that takes place on the output of a mapper.
- To use combiners, the reduce function must be associative and commutative.



MapReduce Application

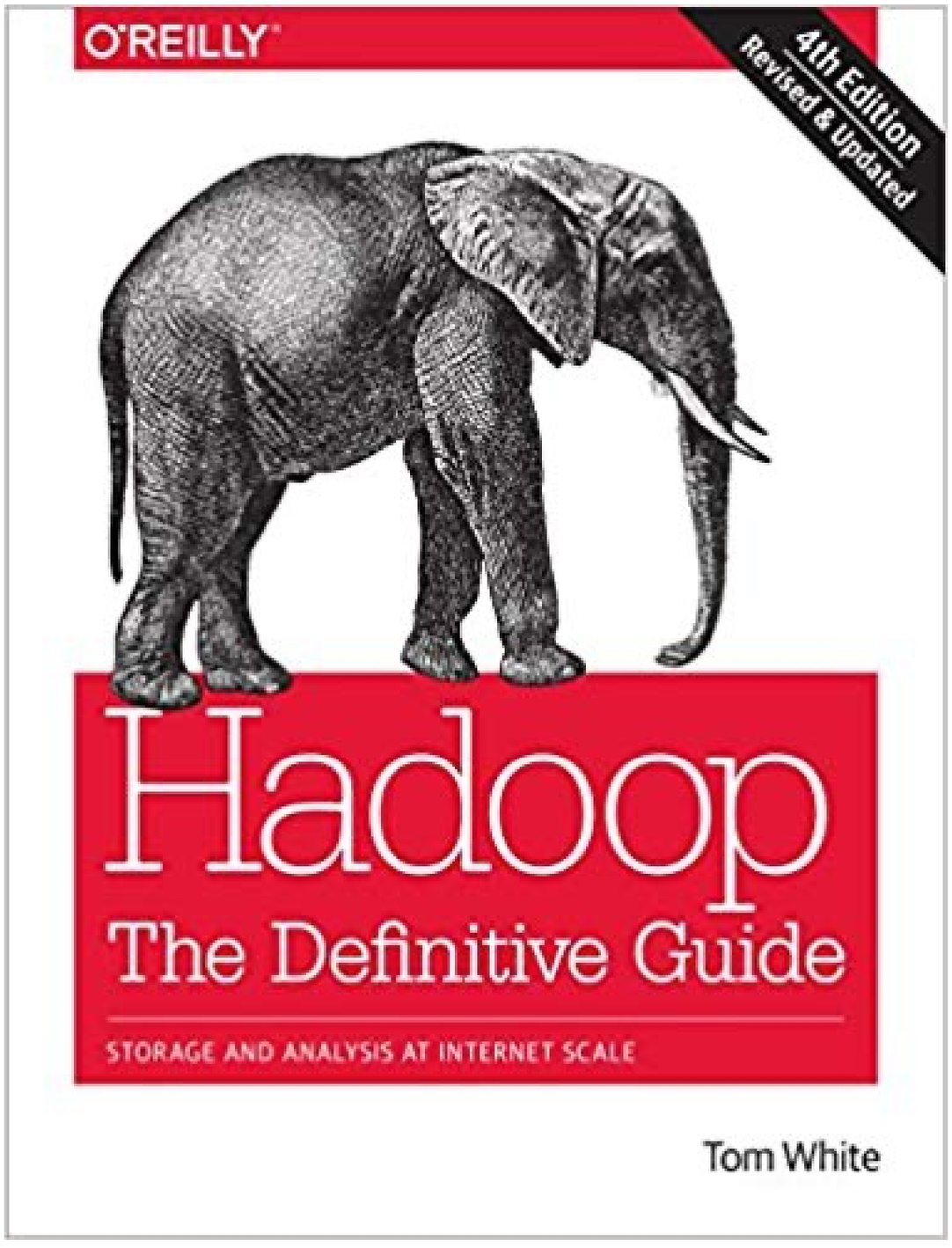


MapReduce Frameworks

- In 2003 Google presented its distributed file system for storing large data sets
  - Google File System
- In 2004 Google presented its distributed platform for processing large data sets
  - MapReduce
- In 2007, at Yahoo, Doug Cutting formed the new project Hadoop
  - Open-source implementation of Google's MapReduce software
  - Include the open-source implementation of Google's GFS software (Hadoop Distributed File System, HDFS)
  - Tested on 1000 nodes
- In 2008, Yahoo released Hadoop as an open-source project to Apache Software Foundation
  - Tested on 4000 nodes

Hadoop

Suggested Textbook



Read the Javadocs!!!

Hadoop Terminology

- A **MapReduce job** is a unit of work that the client wants to be performed. It consists of:
  - Input data
  - MapReduce program
  - Configuration information
- Hadoop runs the MapReduce job by dividing it into tasks
  - There are two types of tasks: map tasks and reduce tasks
  - The tasks are scheduled using YARN (Yet Another Resource Negotiation) and run on nodes in the cluster.
  - If a task fails, it will be automatically rescheduled to run on a different node.
- Hadoop divides the input to a MapReduce job into fixed-size pieces called **Input Splits**
- Hadoop spawns one map task for each split, which runs the user-defined map function for each record in the split

Hadoop Program Execution

- Hadoop programs are developed with a text editors or an IDE
- The actual debugging and execution must be performed on Hadoop without IDE
- Once compiled, an Hadoop program is packaged in a JAR file and submitted to a Hadoop cluster
- We need to specify where HDFS and Hadoop daemons are running
- For simplicity, we will:
  - Develop programs locally using Maven to manage dependencies
  - Package the Hadoop program in a JAR file
  - Copy the file into the Hadoop cluster using SSH
  - Perform debugging and execution on the Hadoop cluster
- Be prepared...

Hadoop Installation

Generally Hadoop can run in three modes:

1. **Local** (or standalone) mode – this is the default mode
   - Hadoop runs as a single Java process
   - Mainly used for debugging
   - There are no daemons used in this mode.
   - Hadoop uses the local file system as a substitute for HDFS file system.
   - The jobs will run as if there is 1 mapper and 1 reducer
2. **Pseudo-distributed** mode (also known as single-node cluster)
   - All the daemons run on a single machine
   - A separate JVM for every Hadoop component
   - This setting mimics the behaviour of a cluster
   - All the daemons run on your machine locally using the HDFS protocol.
   - There can be multiple mappers and reducers (1 reducer by default)
3. **Fully-distributed** mode
   - This is how Hadoop runs on a real cluster
   - All the daemons run on (a subset of) the cluster's machines using the HDFS protocol
   - There are multiple mappers and reducers (1 reducer by default)

Word Count in Hadoop (I)

```
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);
       }
    }
}
```

Word Count in Hadoop (II)

```
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);
   }
}
```

Word Count in Hadoop (III)

```
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> [<in>...] <out>");
     System.exit(2);
   }
   Job job = Job.getInstance(conf, "word count");
   job.setJarByClass(WordCount.class);
   job.setMapperClass(TokenizerMapper.class);
   job.setCombinerClass(IntSumReducer.class);
   job.setReducerClass(IntSumReducer.class);
   job.setOutputKeyClass(Text.class);
   job.setOutputValueClass(IntWritable.class);
   for (int i = 0; i < otherArgs.length - 1; ++i)
      FileInputFormat.addInputPath(job, new Path(otherArgs[i]));
   FileOutputFormat.setOutputPath(job, new Path(otherArgs[otherArgs.length - 1]));
   System.exit(job.waitForCompletion(true) ? 0 : 1);
}
```

Hadoop Types

- *map function*: from [key (K1), value (V1)] pair to list of [key (K2), value (V2)] pairs
- *combiner function*: from [key (K2), list of values (V2)] pair to list of [key (K2), values (V2)] pairs
- *reduce function*: from [key (K2), list of values (V2)] pair to list of [key (K3), values (V3)] pairs
- *partitioner function*: from [key (K2), value (V2)] pair to integer

- If a combiner function is used, then it has the same form as the reduce function
- The combiner function is an implementation of Reducer, except its output types are the intermediate key and value types (K2 and V2)
- Often the combiner and reduce functions are the same, in which case K3 is the same as K2, and V3 is the same as V2
- The partition function operates on the intermediate key and value types (K2 and V2) and returns the partition index. In practice, the partition is determined solely by the key (the value is ignored)

Hadoop Job Configuration (I)

Property	Job setter method	Input types		Intermediate types		Output types	
		K1	V1	K2	V2	K3	V3
Properties for configuring types:							
mapreduce.job.inputformat.class	setInputFormatClass()	•	•				
mapreduce.map.output.key.class	setMapOutputKeyClass()			•			
mapreduce.map.output.value.class	setMapOutputValueClass()				•		
mapreduce.job.output.key.class	setOutputKeyClass()					•	
mapreduce.job.output.value.class	setOutputValueClass()						•
Properties that must be consistent with the types:							
mapreduce.job.map.class	setMapperClass()	•	•	•	•		
mapreduce.job.combine.class	setCombinerClass()			•	•		
mapreduce.job.partitioner.class	setPartitionerClass()			•	•		
mapreduce.job.output.key.comparator.class	setSortComparatorClass()			•			
mapreduce.job.output.group.comparator.class	setGroupingComparatorClass()			•			
mapreduce.job.reduce.class	setReducerClass()			•	•	•	•
mapreduce.job.outputformat.class	setOutputFormatClass()					•	•

Source: Tom White, "*Hadoop - The Definitive Guide (4 ed)*", O'Reilly, 2015

Hadoop Job Configuration (II)

- Input types are set by the input format.
  - For instance, a TextInputFormat (which is the default) generates keys of type LongWritable and values of type Text
- The other types are set explicitly by calling the methods on the Job
- If not set explicitly, the intermediate types default to the (final) output types, which default to LongWritable and Text.
  - For instance, if K2 and K3 are the same, you don't need to call setMapOutputKey Class()
  - Similarly, if V2 and V3 are the same, you only need to use setOutputValueClass()
- It is possible to configure an Hadoop job with incompatible types, because the configuration isn't checked at compile time.

Hadoop Configuration Defaults

- The only configuration that we set is an input path and an output path.
- The default input format is TextInputFormat
  - The key is of type LongWritable, and the value is of type Text
- The default mapper is the Mapper class, which writes the input key and value unchanged to the output
  - The output key is of type LongWritable, and the output value is of type Text
- The default reducer is the Reducer class, which simply writes all its input to its output
  - The output key is of type LongWritable, and the output value is of type Text
- The default partitioner is HashPartitioner, which hashes an intermediate key to determine which partition the key belongs to
  - Each partition is processed by a reduce task
  - So the number of partitions is equal to the number of reduce tasks for the job
  - By default there is a single reducer, and therefore a single partition
- We did not set the number of map tasks
  - The number is equal to the number of splits that the input is turned into
  - It depends on the size of the input and the file's block size (if the file is in HDFS)