UNIVERSITY OF WATERLOO

Data-Intensive Distributed Computing

CS 431/631 451/651 (Fall 2019)

Part 1: MapReduce Algorithm Design (2/4)

Ali Abedi

These slides are available at https://www.student.cs.uwaterloo.ca/~cs451/

MapReduce

Source: Google

What's different?

Data-intensive vs. Compute-intensive

Focus on *data-parallel* abstractions

Coarse-grained vs. Fine-grained parallelism

Focus on *coarse-grained data-parallel* abstractions

Logical vs. Physical

Different levels of design:

"Logical" deals with abstract organizations of computing

"Physical" deals with how those abstractions are realized

Examples:

Scheduling

Operators

Data models

Network topology

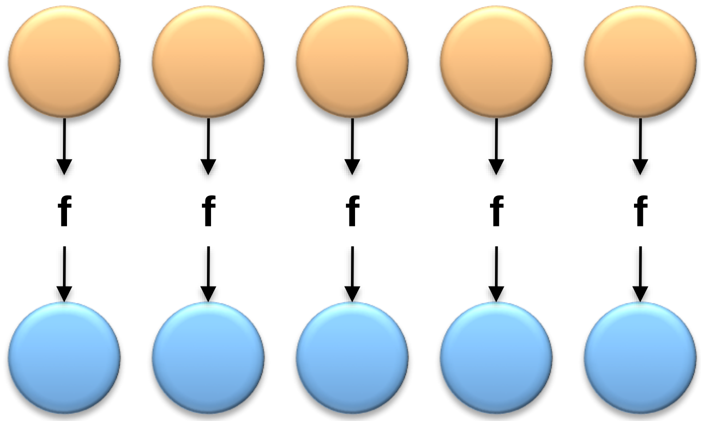
Why is this important?

Roots in Functional Programming

Simplest data-parallel abstraction

Process a large number of records: “do” something to each

Map



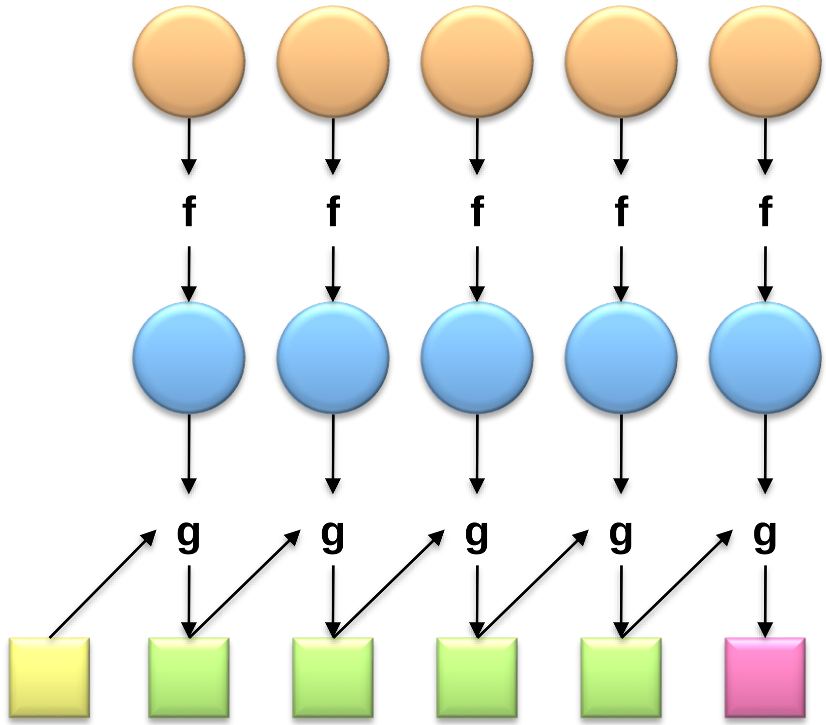
We need something more for sharing partial results across records!

Roots in Functional Programming

Let’s add in aggregation!

Map

Fold



MapReduce = Functional programming + distributed computing!

Functional Programming in Scala

```
scala> val t = Array(1, 2, 3, 4, 5)
t: Array[Int] = Array(1, 2, 3, 4, 5)

scala> t.map(n => n*n)
res0: Array[Int] = Array(1, 4, 9, 16, 25)

scala> t.map(n => n*n).foldLeft(0)((m, n) => m + n)
res1: Int = 55
```

Imagine parallelizing the map and fold across a cluster…

A Data-Parallel Abstraction

Process a large number of records

Map "Do something" to each

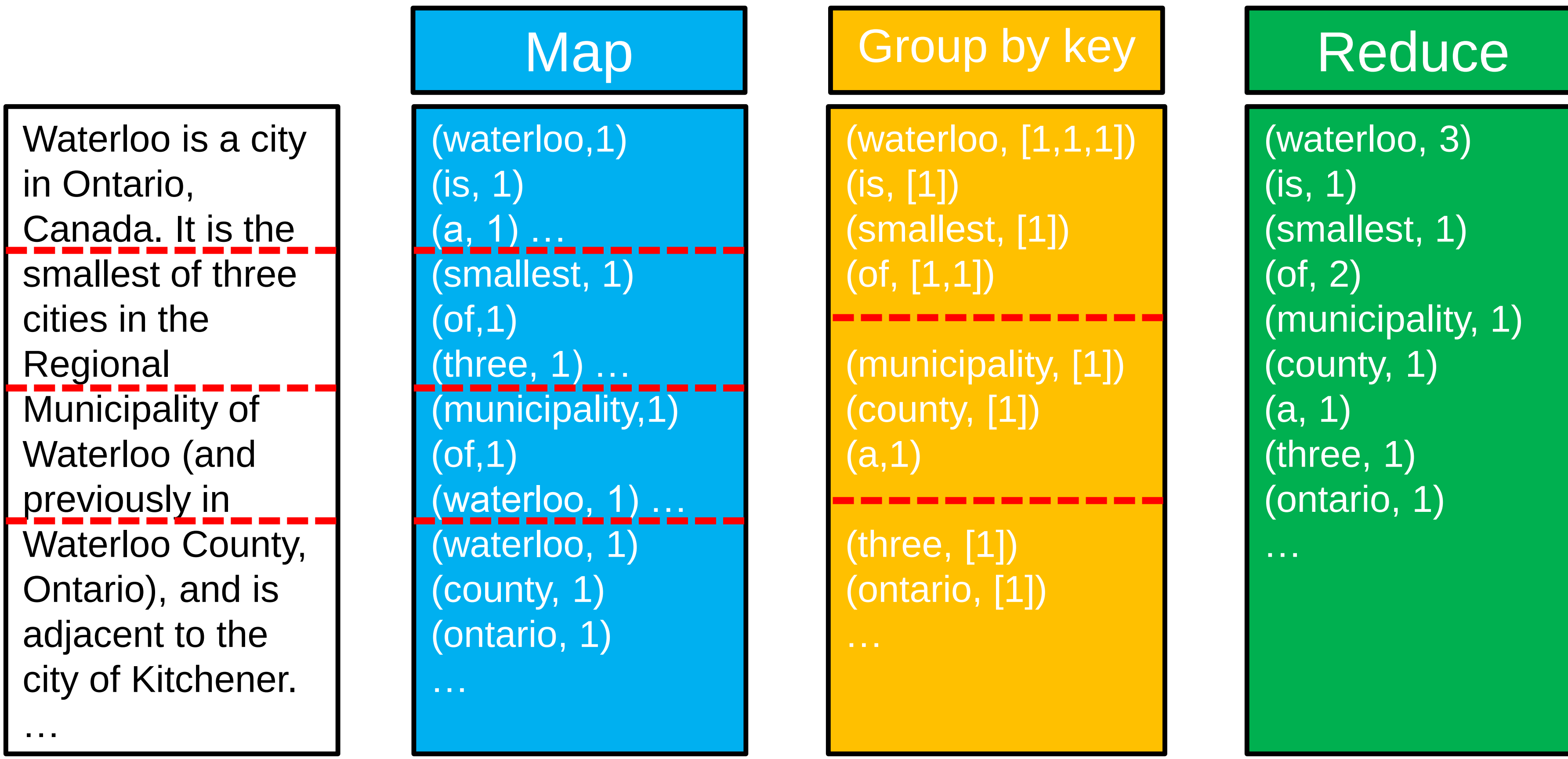
Group intermediate results

"Aggregate" intermediate results Reduce

Write final results

Key idea: provide a functional abstraction for these two operations

MapReduce “word count” example



Big document

MapReduce “word count” pseudo-code

```
def map(key: Long, value: String) = {
  for (word <- tokenize(value)) {
    emit(word, 1)
  }
}

def reduce(key: String, values: Iterable[Int]) = {
  for (value <- values) {
    sum += value
  }
  emit(key, sum)
}
```

MapReduce

Programmer specifies two functions:

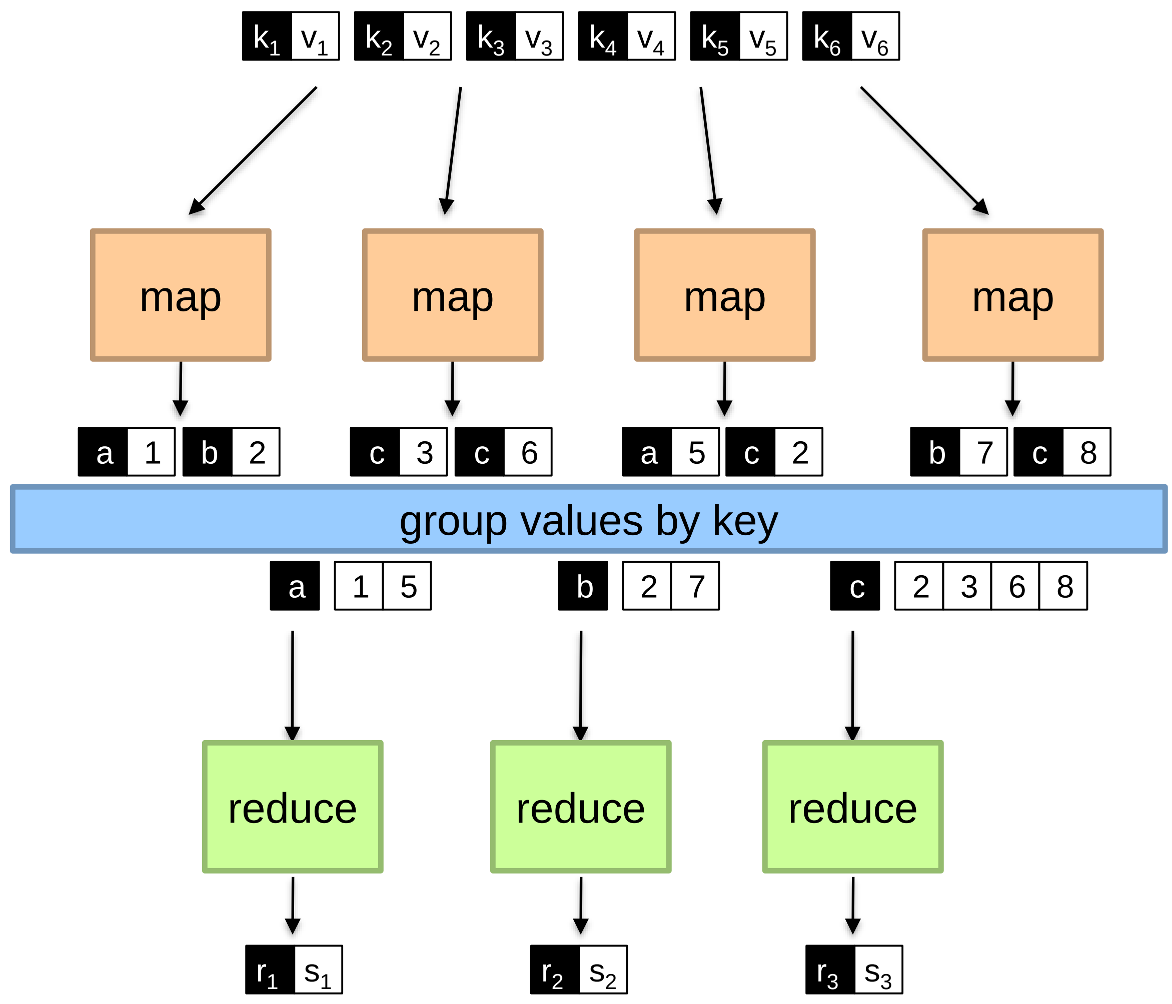
map $(k_1, v_1) \rightarrow List[(k_2, v_2)]$

reduce $(k_2, List[v_2]) \rightarrow List[(k_3, v_3)]$

All values with the same key are sent to the same reducer

What does this actually mean?

The execution framework handles everything else…


k1 v1
k2 v2
k3 v3
k4 v4
k5 v5
k6 v6
map
map
map
map
a 1 b 2
c 3 c 6
a 5 c 2
b 7 c 8
group values by key
a 1 5
b 2 7
c 2 3 6 8
reduce
reduce
reduce
r1 s1
r2 s2
r3 s3

MapReduce

Programmer specifies two functions:

map $(k_1, v_1) \rightarrow \text{List}[(k_2, v_2)]$

reduce $(k_2, \text{List}[v_2]) \rightarrow \text{List}[(k_3, v_3)]$

All values with the same key are sent to the same reducer

The execution framework handles everything else…

What's "everything else"?

MapReduce “Runtime”

Handles scheduling

Assigns workers to map and reduce tasks

Handles “data distribution”

Moves processes to data

Handles synchronization

Groups intermediate data

Handles errors and faults

Detects worker failures and restarts

Everything happens on top of a distributed FS (later)

MapReduce

Programmer specifies two functions:

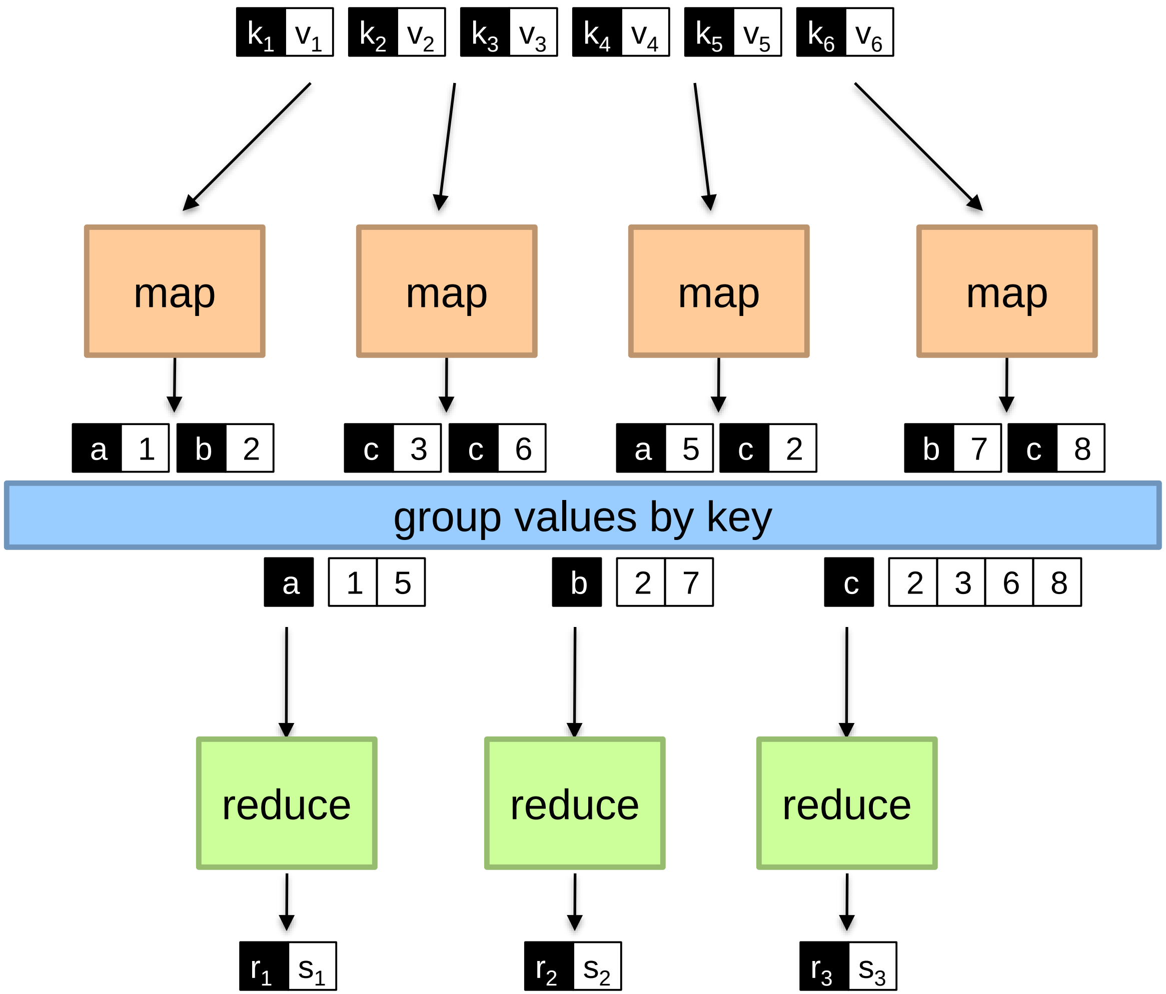
map $(k_1, v_1) \rightarrow List[(k_2, v_2)]$

reduce $(k_2, List[v_2]) \rightarrow List[(k_3, v_3)]$

All values with the same key are sent to the same reducer

The execution framework handles everything else…

Not quite…



What’s the most complex and slowest operation here?

MapReduce

four

Programmer specifies ~~two~~ functions:

map $(k_1, v_1) \rightarrow \text{List}[(k_2, v_2)]$

reduce $(k_2, \text{List}[v_2]) \rightarrow \text{List}[(k_3, v_3)]$

All values with the same key are sent to the same reducer

partition $(k', p) \rightarrow 0 \ldots p\text{-}1$

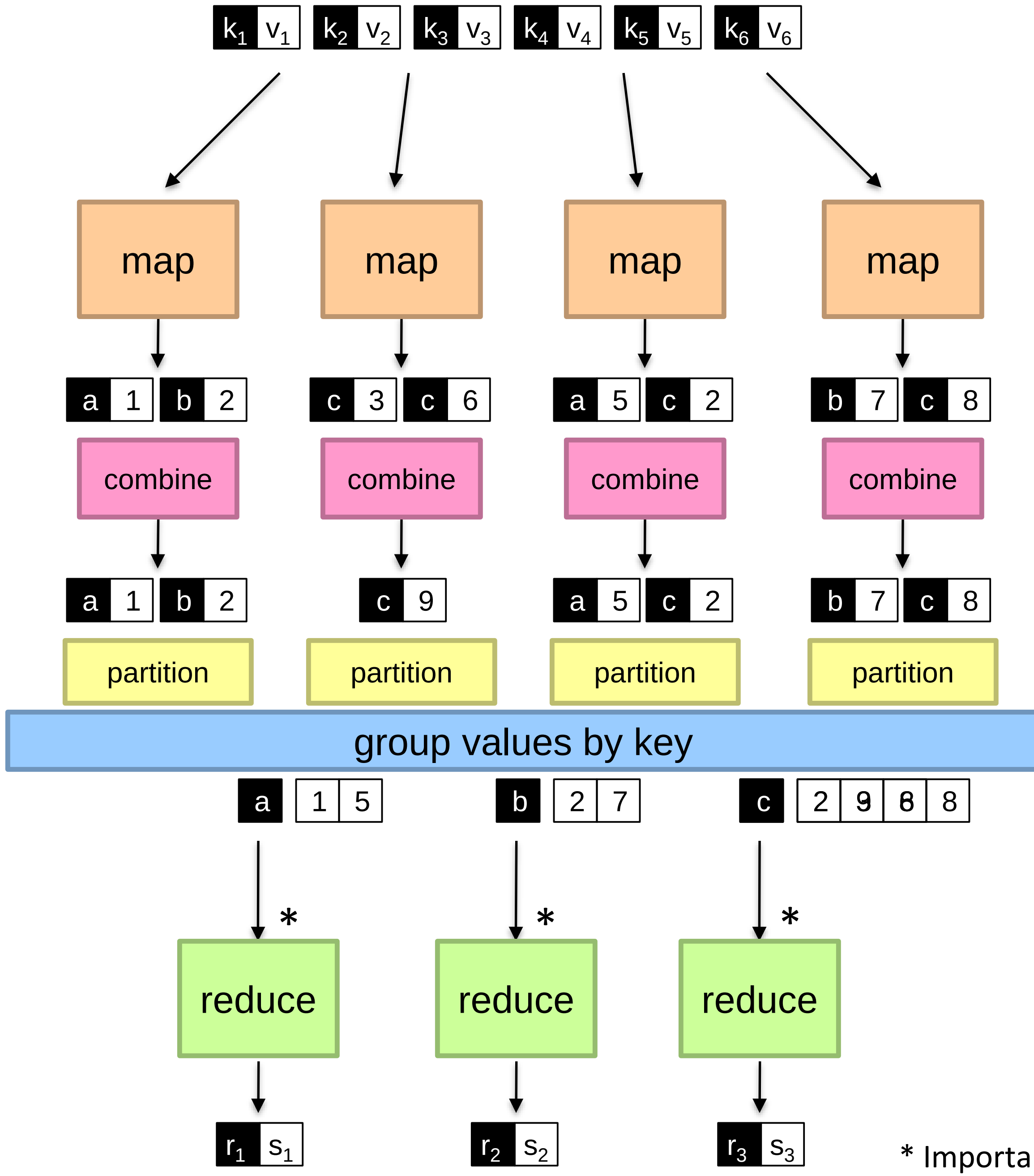
Often a simple hash of the key, e.g., hash(k') mod n

Divides up key space for parallel reduce operations

combine $(k_2, \text{List}[v_2]) \rightarrow \text{List}[(k_2, v_2)]$

Mini-reducers that run in memory after the map phase

Used as an optimization to reduce network traffic



* Important detail: reducers process keys in sorted order

MapReduce can refer to…

The programming model

The execution framework (aka “runtime”)

The specific implementation

Usage is usually clear from context!

MapReduce Implementations

Google has a proprietary implementation in C++

Bindings in Java, Python

Hadoop provides an open-source implementation in Java

Development begun by Yahoo, later an Apache project

Used in production at Facebook, Twitter, LinkedIn, Netflix, …

Large and expanding software ecosystem

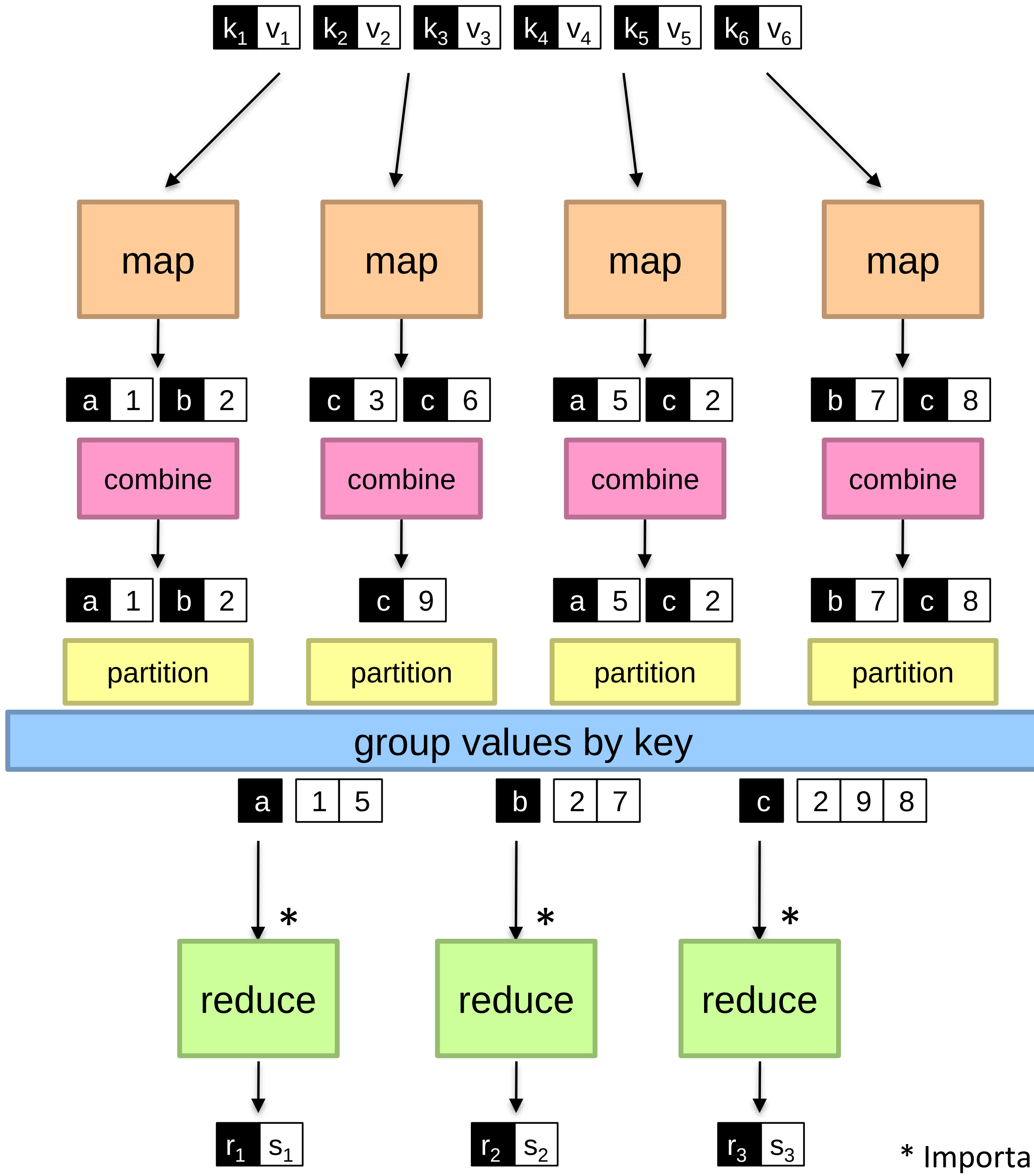
Potential point of confusion: Hadoop is more than MapReduce today

Lots of custom research implementations

Tackling Big Data

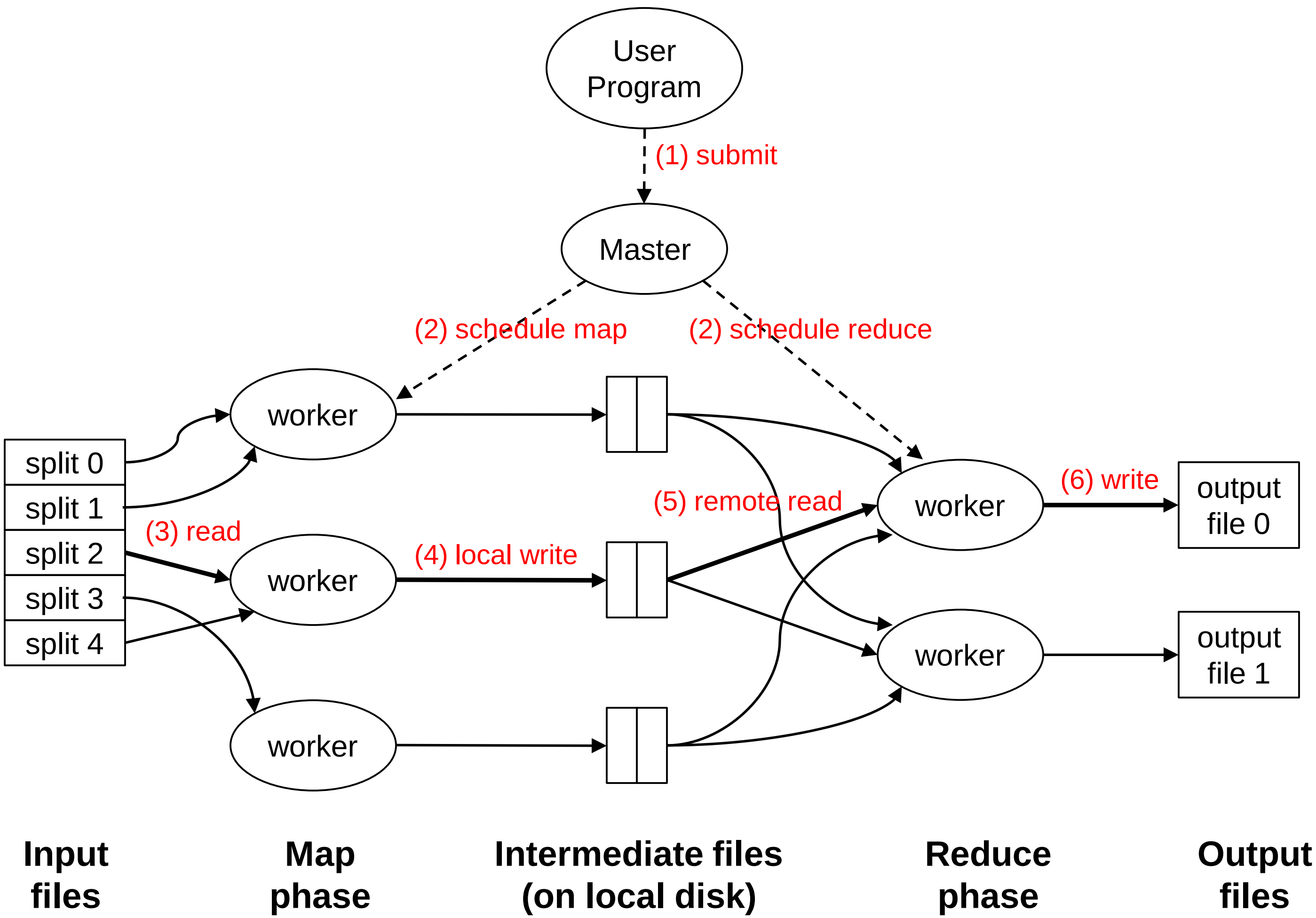
Source: Google

Logical View



* Important detail: reducers process keys in sorted order

Physical View



Adapted from (Dean and Ghemawat, OSDI 2004)


The datacenter *is* the computer!
Source: Google

The datacenter *is* the computer!

It's all about the right level of abstraction

Moving beyond the von Neumann architecture

What's the "instruction set" of the datacenter computer?

Hide system-level details from the developers

No more race conditions, lock contention, etc.

No need to explicitly worry about reliability, fault tolerance, etc.

Separating the *what* from the *how*

Developer specifies the computation that needs to be performed

Execution framework ("runtime") handles actual execution

The datacenter *is* the computer!
“Big ideas”

Scale “out”, not “up” *

Limits of SMP and large shared-memory machines

Assume that components will break

Engineer software around hardware failures

Move processing to the data *

Cluster have limited bandwidth, code is a lot smaller

Process data sequentially, avoid random access

Seeks are expensive, disk throughput is good

Seek vs. Scans

Consider a 1 TB database with 100 byte records

We want to update 1 percent of the records

Scenario 1: Mutate each record

Each update takes ~30 ms (seek, read, write)

10^8 updates = ~35 days

Scenario 2: Rewrite all records

Assume 100 MB/s throughput

Time = 5.6 hours(!)

Lesson? Random access is expensive!

Source: Ted Dunning, on Hadoop mailing list

Source: Wikipedia (Mahout)

A tale of two packages…

org.apache.hadoop.mapreduce
org.apache.hadoop.mapred

Source: Wikipedia (Budapest)

MapReduce API*

Mapper<K_{in},V_{in},K_{out},V_{out}>

void setup(Mapper.Context context)

Called once at the start of the task

void map(K_{in} key, V_{in} value, Mapper.Context context)

Called once for each key/value pair in the input split

void cleanup(Mapper.Context context)

Called once at the end of the task

Reducer<K_{in},V_{in},K_{out},V_{out}>/Combiner<K_{in},V_{in},K_{out},V_{out}>

void setup(Reducer.Context context)

Called once at the start of the task

void reduce(K_{in} key, Iterable<V_{in}> values, Reducer.Context context)

Called once for each key

void cleanup(Reducer.Context context)

Called once at the end of the task

*Note that there are two versions of the API!

MapReduce API*

Partitioner<K, V>

int getPartition(K key, V value, int numPartitions)

Returns the partition number given total number of partitions

Job

Represents a packaged Hadoop job for submission to cluster

Need to specify input and output paths

Need to specify input and output formats

Need to specify mapper, reducer, combiner, partitioner classes

Need to specify intermediate/final key/value classes

Need to specify number of reducers (but not mappers, why?)

Don’t depend on defaults!

*Note that there are two versions of the API!

Data Types in Hadoop: Keys and Values

Writable

Defines a de/serialization protocol.
Every data type in Hadoop is a Writable.

↑

WritableComprable

Defines a sort order.
All keys must be of this type (but not values).

↑

IntWritable
LongWritable
Text
…

Concrete classes for different data types.
Note that these are container objects.

SequenceFile

Binary-encoded sequence of key/value pairs.

“Hello World” MapReduce: Word Count

```
def map(key: Long, value: String) = {
  for (word <- tokenize(value)) {
    emit(word, 1)
  }
}


def reduce(key: String, values: Iterable[Int]) = {
  for (value <- values) {
    sum += value
  }
  emit(key, sum)
}
```

Word Count Mapper

```
private static final class MyMapper
    extends Mapper<LongWritable, Text, Text, IntWritable> {

  private final static IntWritable ONE = new IntWritable(1);
  private final static Text WORD = new Text();

  @Override
  public void map(LongWritable key, Text value, Context context)
      throws IOException, InterruptedException {
    for (String word : Tokenizer.tokenize(value.toString())) {
      WORD.set(word);
      context.write(WORD, ONE);
    }
  }
}
```

Word Count Reducer

```
private static final class MyReducer
    extends Reducer<Text, IntWritable, Text, IntWritable> {

   private final static IntWritable SUM = new IntWritable();

   @Override
   public void reduce(Text key, Iterable<IntWritable> values,
      Context context) throws IOException, InterruptedException {
     Iterator<IntWritable> iter = values.iterator();
     int sum = 0;
     while (iter.hasNext()) {
       sum += iter.next().get();
     }
     SUM.set(sum);
     context.write(key, SUM);
   }
 }
```

Getting Data to Mappers and Reducers

Configuration parameters

Pass in via Job configuration object

"Side data"

DistributedCache

Mappers/Reducers can read from HDFS in setup method

Complex Data Types in Hadoop

How do you implement complex data types?

The easiest way:

Encode it as Text, e.g., (a, b) = “a:b”

Use regular expressions to parse and extract data

Works, but janky

The hard way:

Define a custom implementation of Writable(Comprable)

Must implement: readFields, write, (compareTo)

Computationally efficient, but slow for rapid prototyping

Implement WritableComparator hook for performance

Somewhere in the middle:

Bespin (via lin.tl) offers various building blocks

Anatomy of a Job

Hadoop MapReduce program = Hadoop job

Jobs are divided into map and reduce tasks

An instance of a running task is called a task attempt

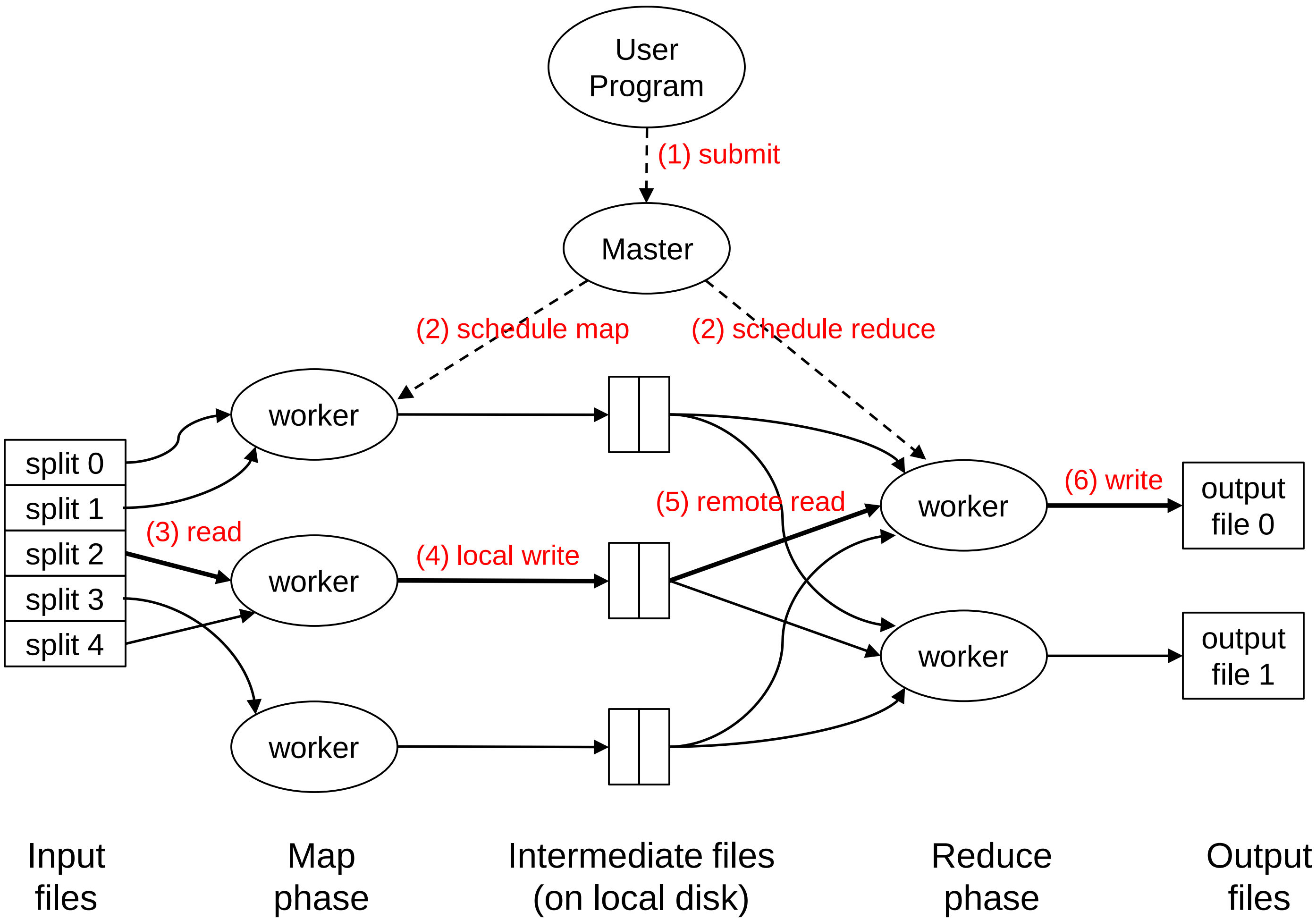
Each task occupies a slot on the tasktracker

Multiple jobs can be composed into a workflow

Job submission:

Client (i.e., driver program) creates a job,
configures it,
and submits it to jobtracker

That's it! The Hadoop cluster takes over…

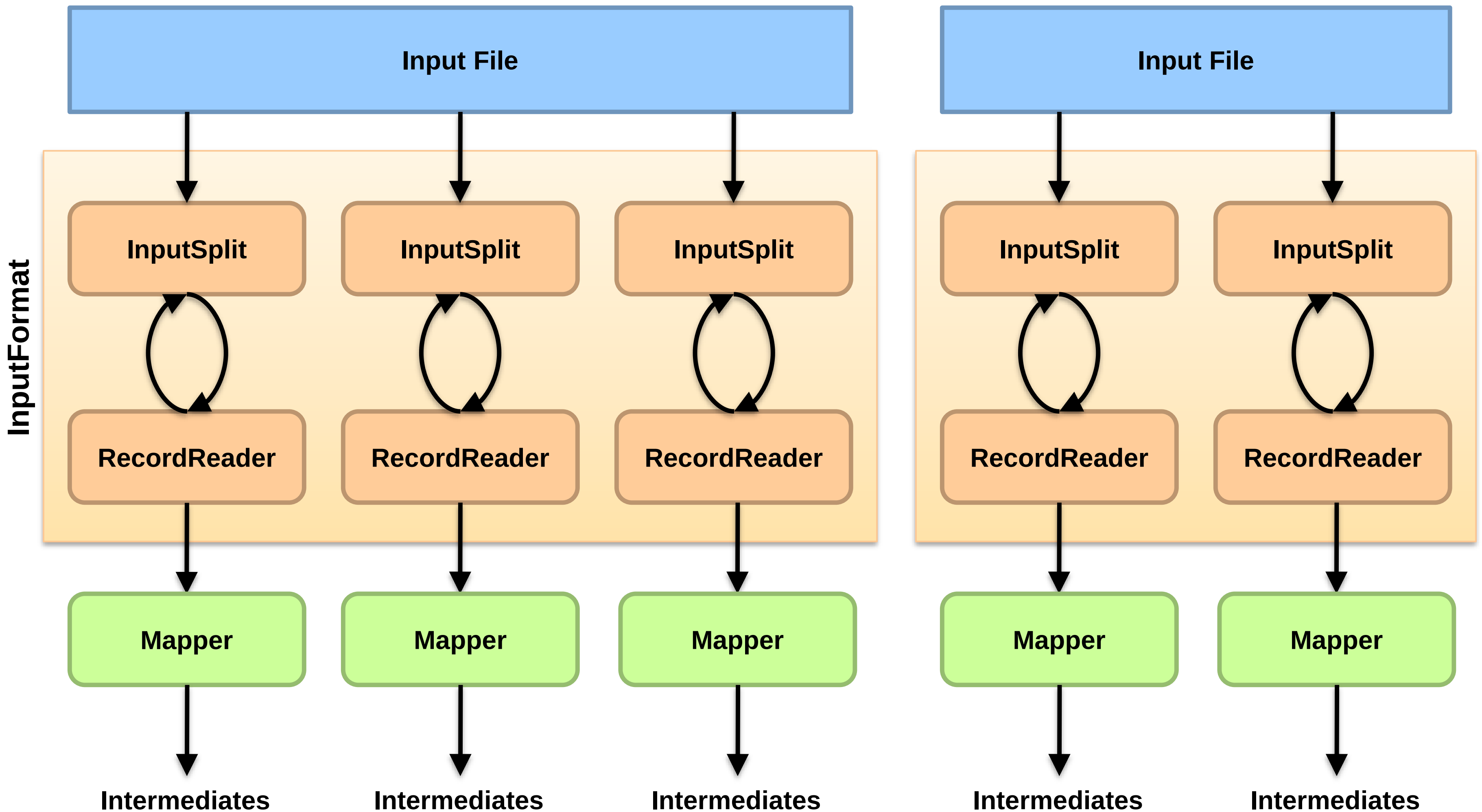


Adapted from (Dean and Ghemawat, OSDI 2004)

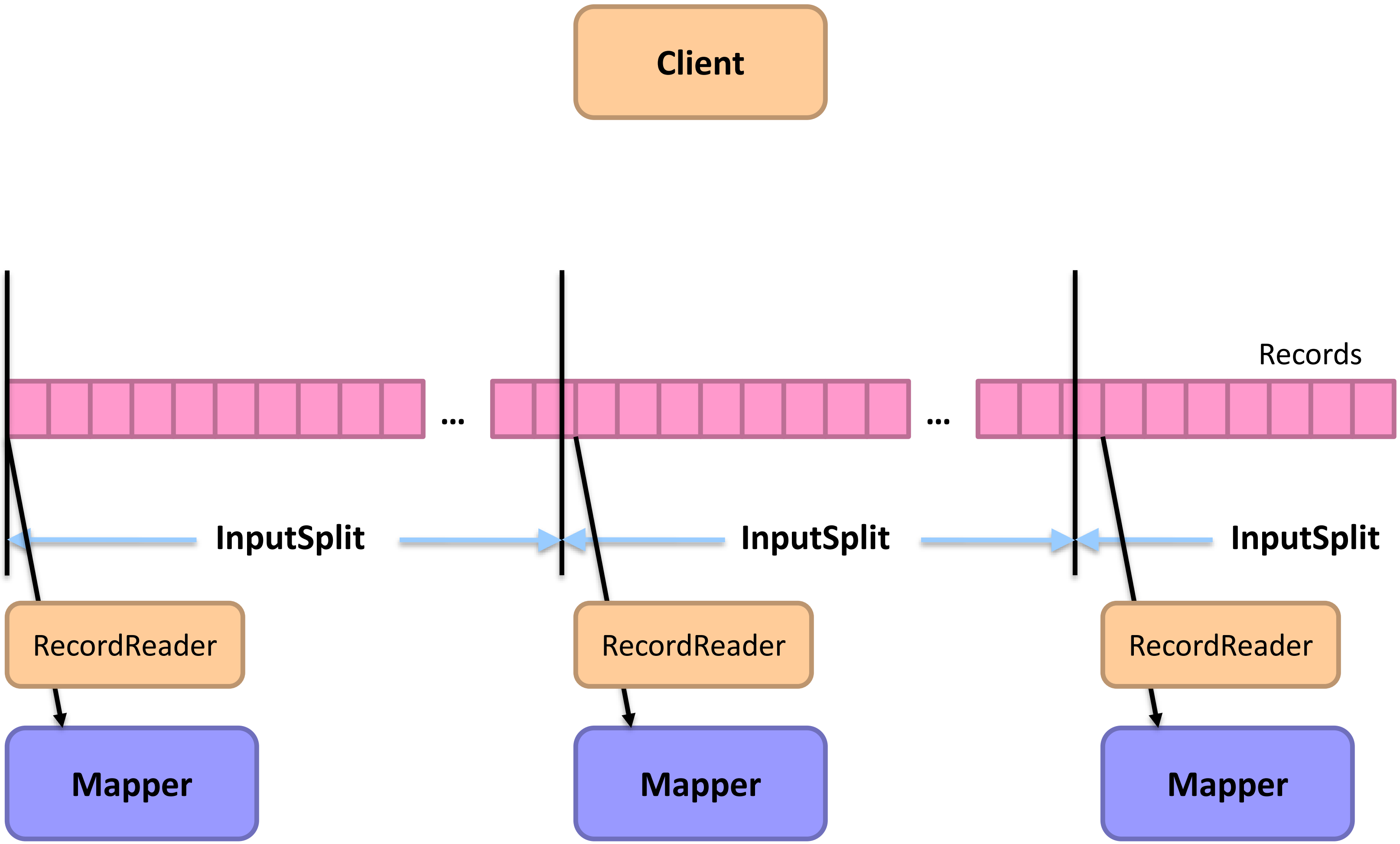
Anatomy of a Job

Behind the scenes:

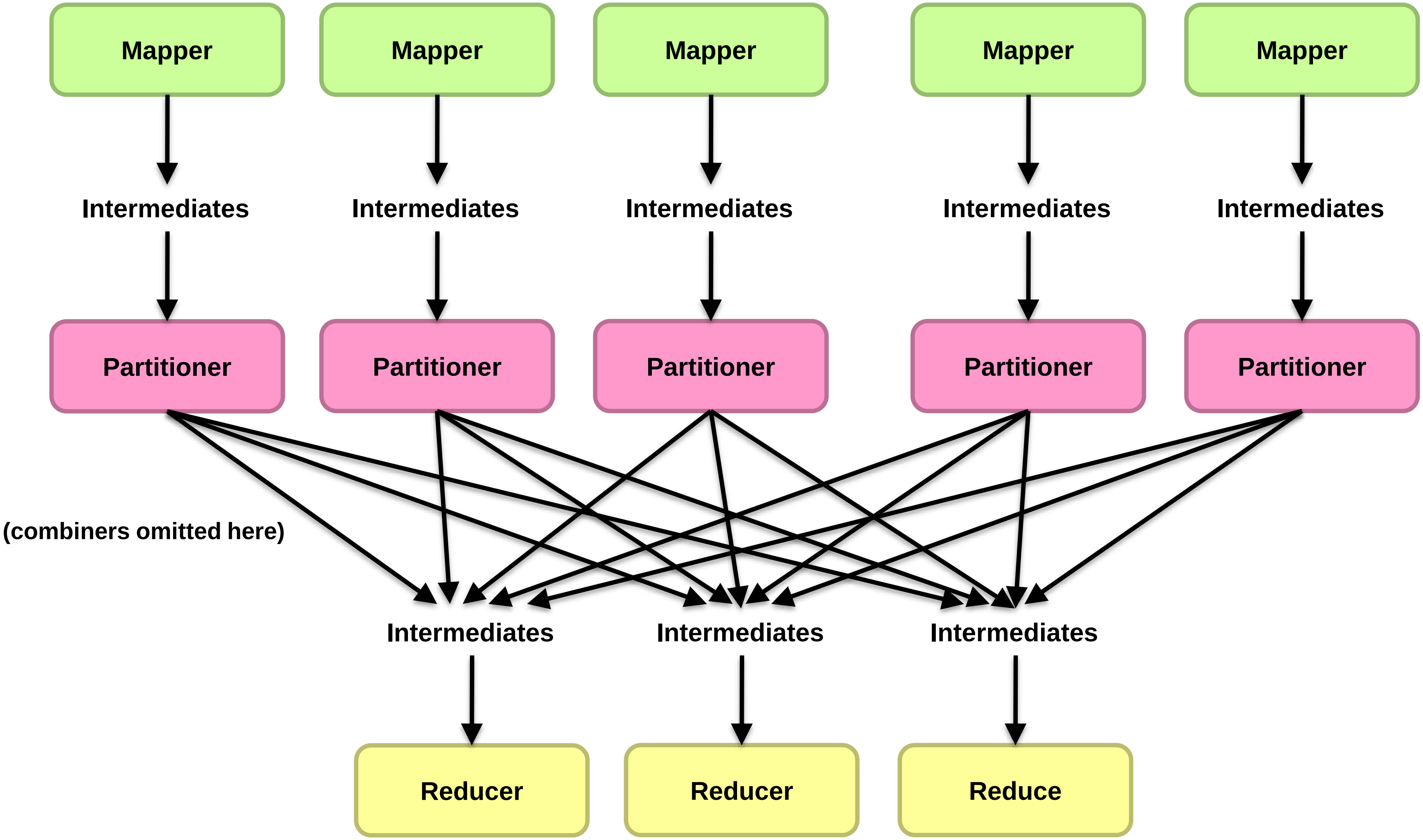
Input splits are computed (on client end)
Job data (jar, configuration XML) are sent to jobtracker
Jobtracker puts job data in shared location, enqueues tasks
Tasktrackers poll for tasks
Off to the races…



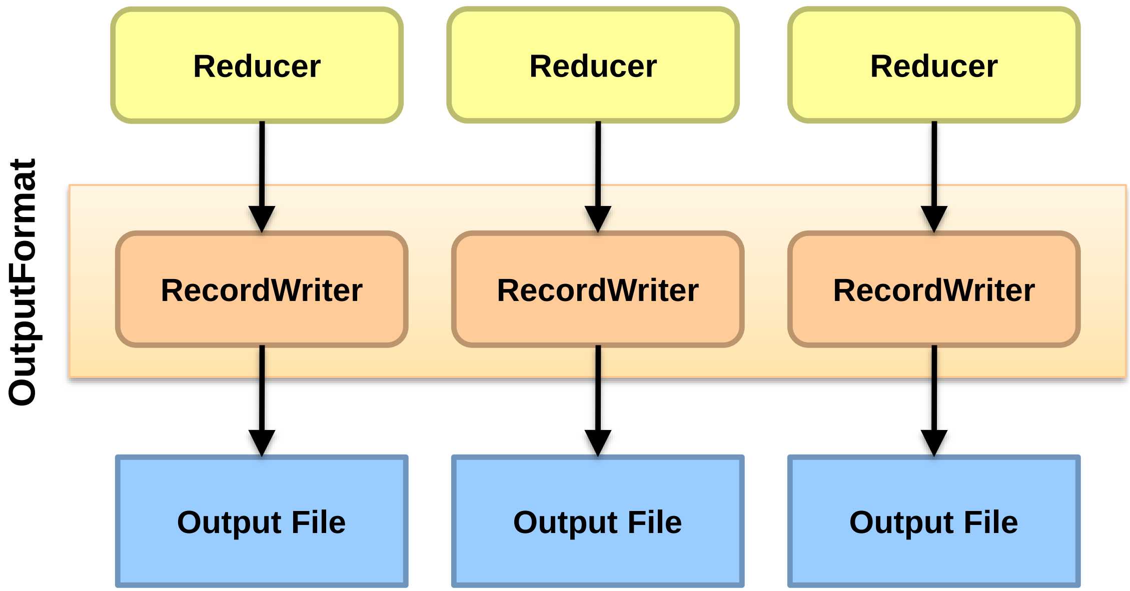
Source: redrawn from a slide by Cloduera, cc-licensed



Where’s the data actually coming from?



Source: redrawn from a slide by Cloduera, cc-licensed



Source: redrawn from a slide by Cloduera, cc-licensed

Input and Output

InputFormat

TextInputFormat
KeyValueTextInputFormat
SequenceFileInputFormat
…

OutputFormat

TextOutputFormat
SequenceFileOutputFormat
…

Spark also uses these abstractions for reading and writing data!

Hadoop Workflow

You

Submit node
(datasci)

Hadoop Cluster

Getting data in?
Writing code?
Getting data out?

Where's the actual
data stored?

Debugging Hadoop

First, take a deep breath

Start small, start locally

Build incrementally

Code Execution Environments

Different ways to run code:

Local (standalone) mode

Pseudo-distributed mode

Fully-distributed mode

Learn what's good for what

Hadoop Debugging Strategies

Good ol' System.out.println

Learn to use the webapp to access logs

Logging preferred over System.out.println

Be careful how much you log!

Fail on success

Throw RuntimeExceptions and capture state

Use Hadoop as the "glue"

Implement core functionality outside mappers and reducers

Independently test (e.g., unit testing)

Compose (tested) components in mappers and reducers

Source: Wikipedia (Japanese rock garden)