Introduction to MapReduce

Bu eğitim sunumları İstanbul Kalkınma Ajansı'nın 2016 yılı Yenilikçi ve Yaratıcı İstanbul Mali Destek Programı kapsamında yürütülmekte olan TR10/16/YNY/0036 no'lu İstanbul Big Data Eğitim ve Araştırma Merkezi Projesi dahilinde gerçekleştirilmiştir. İçerik ile ilgili tek sorumluluk Bahçeşehir Üniversitesi'ne ait olup İSTKA veya Kalkınma Bakanlığı'nın görüşlerini yansıtmamaktadır.

Adopted from Jimmy Lin's slides (at UMD)

Typical Large-Data Problem

Map

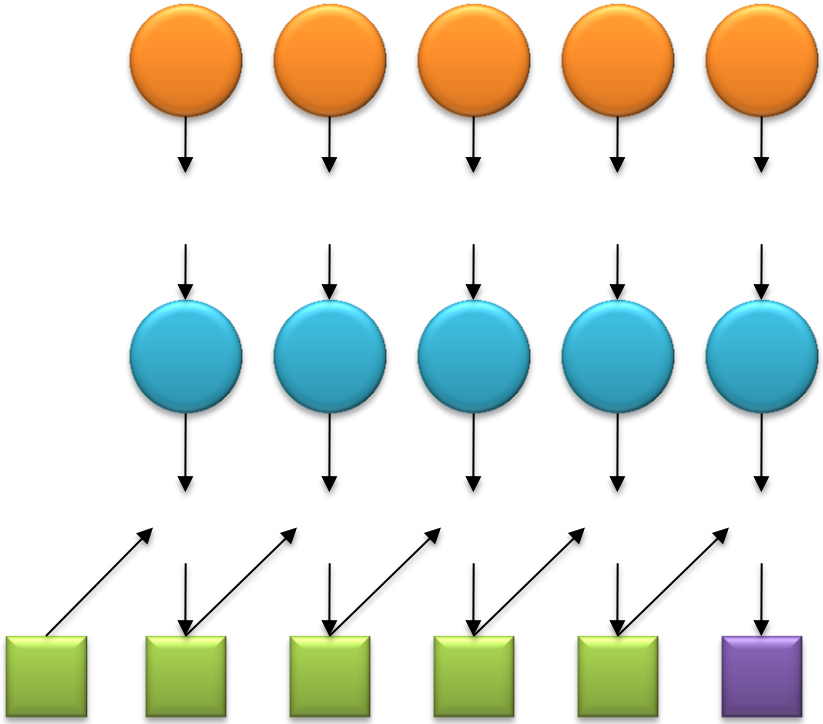
- Iterate over a large number of records
- Extract something of interest from each
- Shuffle and sort intermediate results

Reduce

- Aggregate intermediate results
- Generate final output

Key idea: provide a functional abstraction for these two operations

Roots in Functional Programming



MapReduce

- Programmers specify two functions:

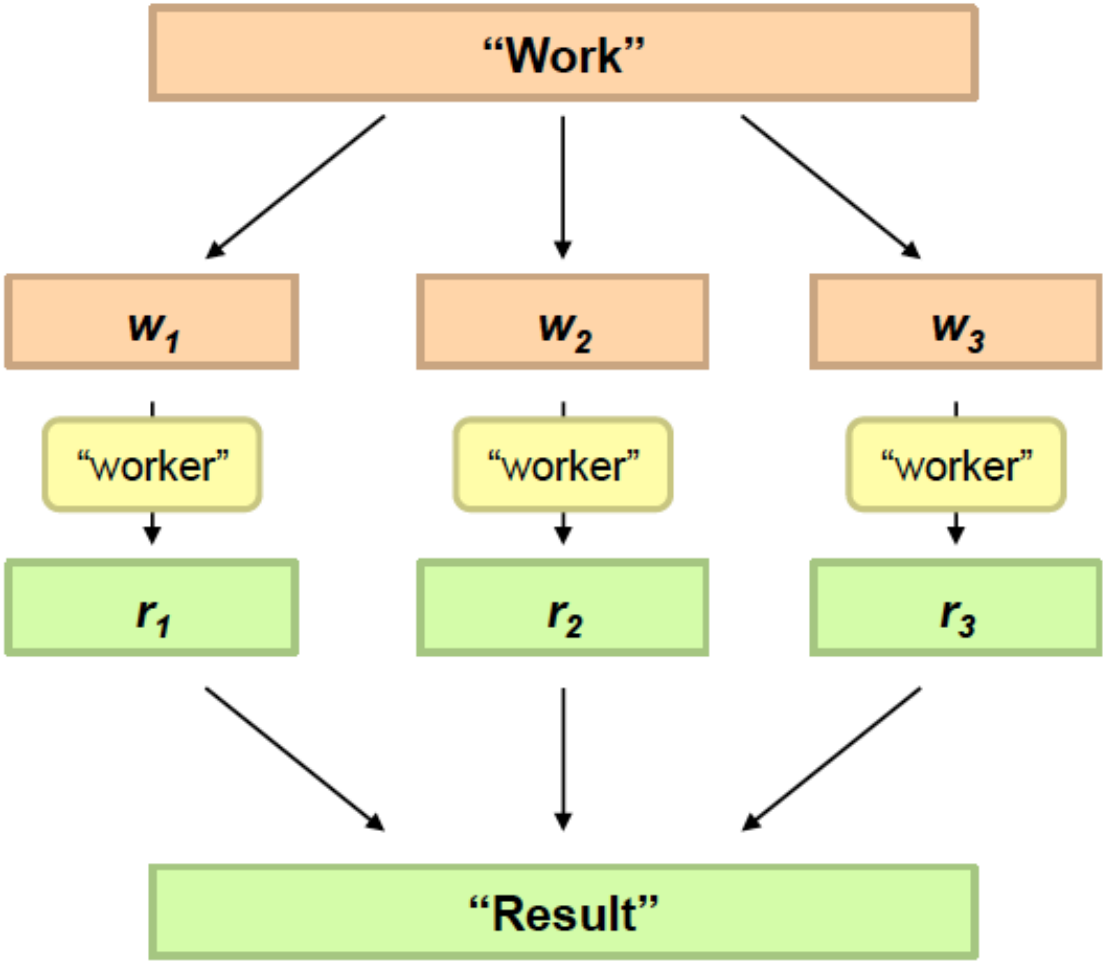
  map (k, v) → [(k', v')]

  reduce (k', [v']) → [(k', v')]

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

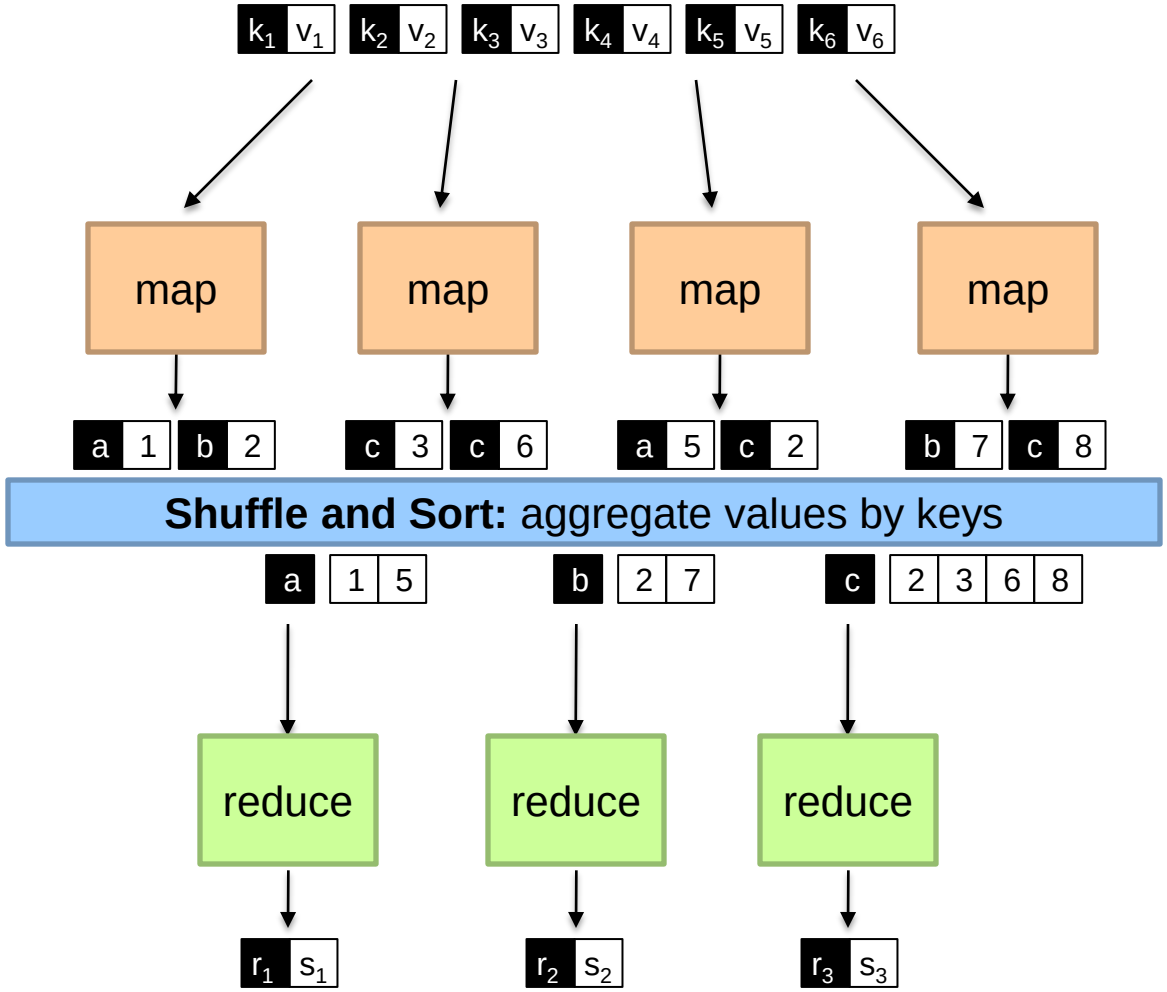
- The execution framework handles everything else…

Divide and Conquer



Divide Work

Combine Results


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
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 3 6 8
reduce
reduce
reduce
r1 s1
r2 s2
r3 s3

MapReduce

- Programmers specify two functions:

  map (k, v) → <k', v'>*

  reduce (k', v') → <k', v'>*

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

- The execution framework handles everything else…

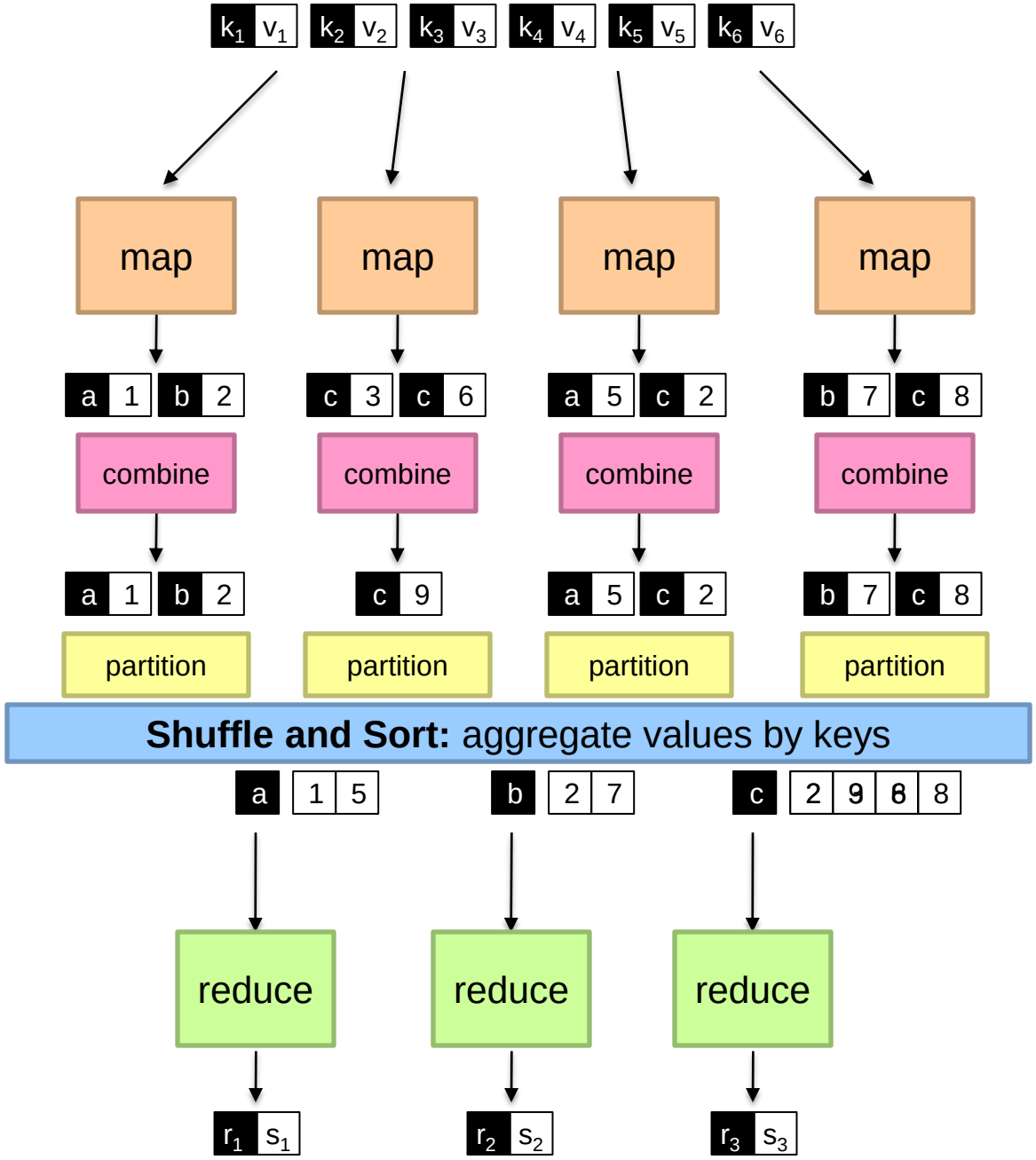
What's "everything else"?

MapReduce “Runtime”

- Handles scheduling
  - Assigns workers to map and reduce tasks
- Handles “data distribution”
  - Moves processes to data
- Handles synchronization
  - Gathers, sorts, and shuffles intermediate data
- Handles errors and faults
  - Detects worker failures and restarts
- Everything happens on top of a distributed FS (later)

MapReduce

- Programmers specify two functions:
  map (k, v) → [(k', v')]
  reduce (k', [v']) → [(k', v')]
  – All values with the same key are reduced together
- The execution framework handles everything else…
- Not quite…usually, 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


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
combine
combine
combine
combine
a 1
b 2
c 9
a 5
c 2
b 7
c 8
partition
partition
partition
partition
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 9 6 8
reduce
reduce
reduce
r1 s1
r2 s2
r3 s3

Two more details…

- Barrier between map and reduce phases
  - But we can begin copying intermediate data earlier
- Keys arrive at each reducer in sorted order
  - No enforced ordering *across* reducers

MapReduce can refer to…

- The programming model
- The execution framework (aka “runtime”)
- The specific implementation

Usage is usually clear from context!

Map--Reduce Code

- `main()` method
- `Job` object –Collects up all the specs for the job
  - Where is the JAR file to distribute?
  - Type of the output pair
  - `Mapper` and `Reducer` classes
  - Input and output file formats
  - Input file(s), output directory

- Configuration object – forwarded to `map()`, `reduce()`
  - Job level parameters communicated via this object

Map--Reduce Code

- MapClass
  - Extends `Mapper`, declaring the input and output pair types for the `map()` method

- `map()` method
  - Arguments: input pair, and the `Context`
  - Output done via the context object

Map--Reduce Code

- ReduceClass
  - Extends `Reducer`, declaring the input and output pair types for the `reduce()` method

- `reduce()` method
  - Arguments: input pair, and the `Context`
  - Output done via the context object

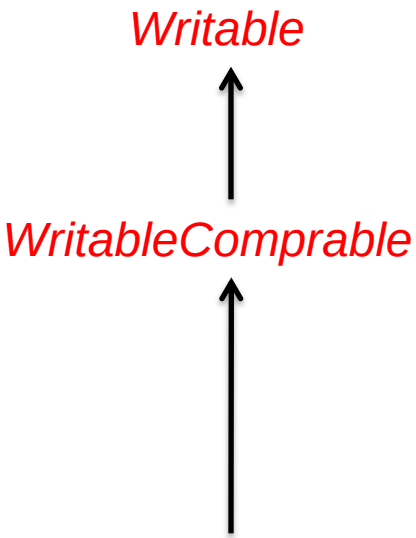
Map--Reduce Code

- `map()` and `reduce()` input pair and output pair types
- Derived from `Writable`
  - `readFields(DataInput in)`
  - `write(DataOutput out)`

- `Text`, `IntWritable`, `LongWritable` all implement `Writable`
  - As do many other types, some of which we will use

- Possible to design your own class that implements `Writable`

Data Types in Hadoop: Keys and Values



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

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

Concrete classes for different data types.

Binary encoded of a sequence of key/value pairs

“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);
```

Example Word Count (Map)

```
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);
    }
  }
}
```

Example Word Count (Reduce)

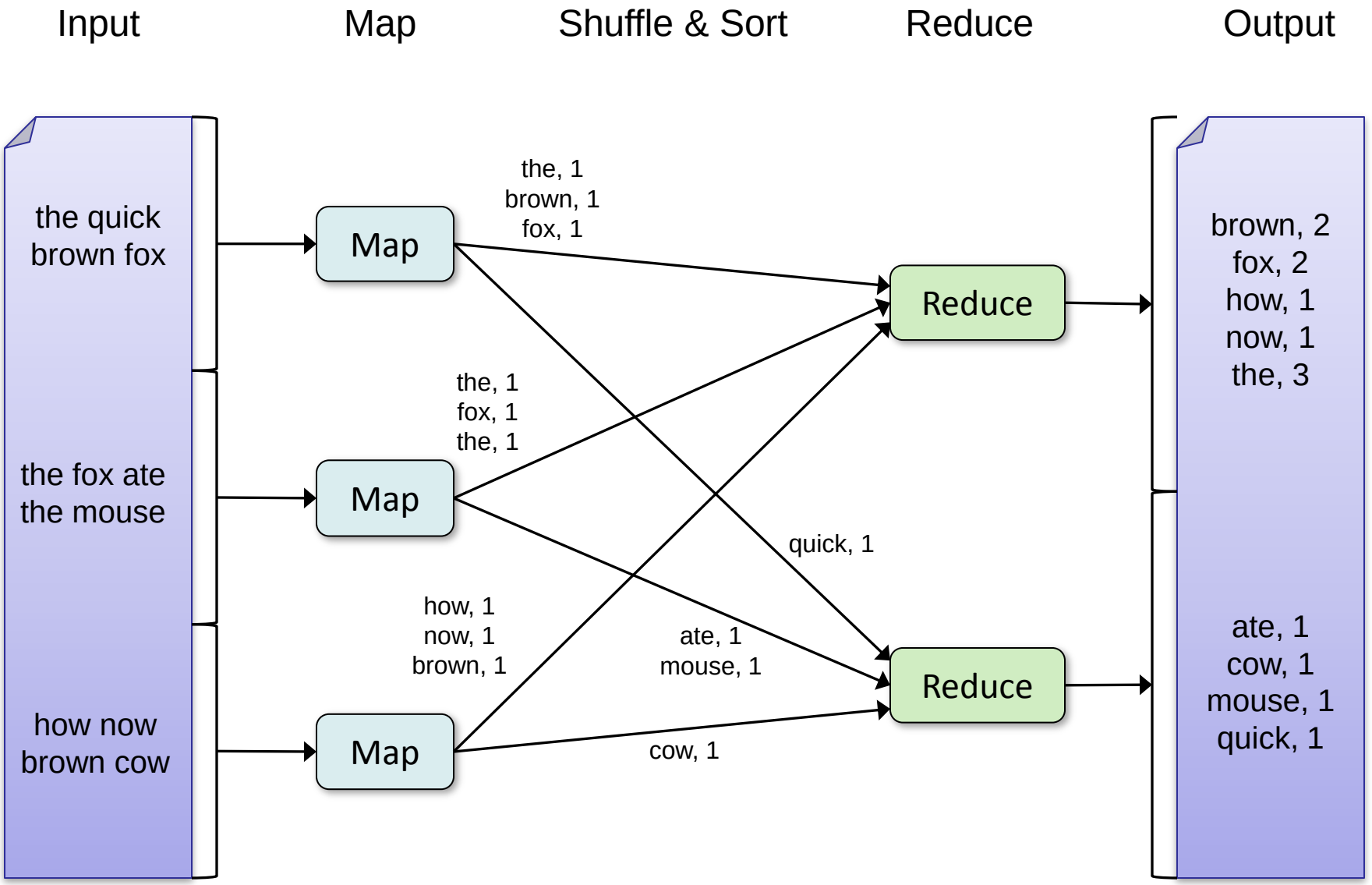
```
public static class IntSumReducer
     extends Reducer<Text,IntWritable,Text,IntWritable> {
  private IntWritable result = new IntWritable();

  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);
  }
}
```

Example Word Count (Driver)

```
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> <out>");
    System.exit(2);
     }
  Job job = new Job(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);
  FileInputFormat.addInputPath(job, new Path(otherArgs[0]));
  FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));
  System.exit(job.waitForCompletion(true) ? 0 : 1);
 }
```

Word Count Execution



An Optimization: The Combiner

- A combiner is a local aggregation function for repeated keys produced by same map
- Combines multiple outputs from a Mapper before shuffle
- For associative ops. like sum, count, max
- Decreases size of intermediate data

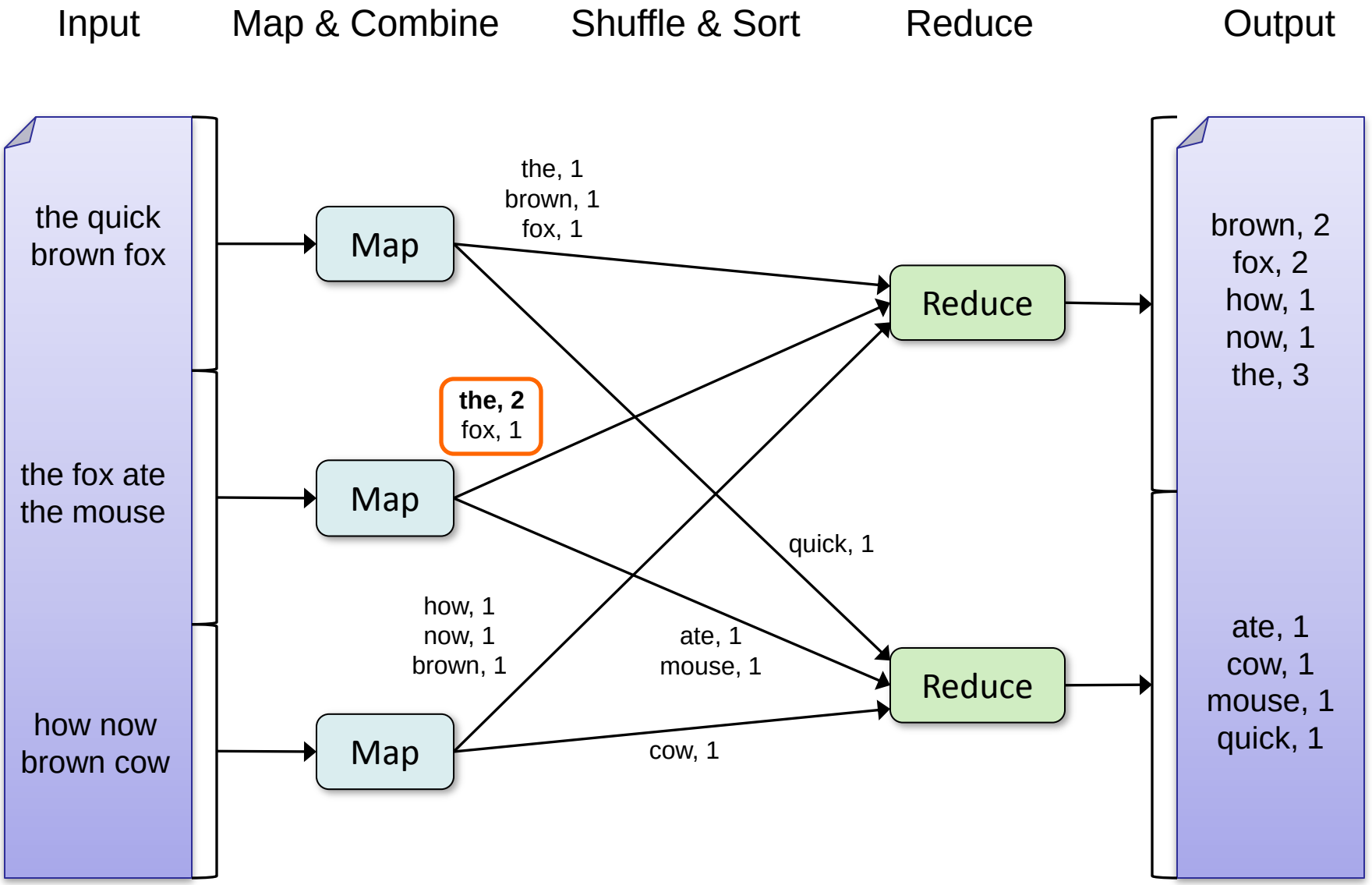
- Example: local counting for Word Count:

```
def combiner(key, values):
  output(key, sum(values))
```

Combiner

- Input and output pair types must be the same.
  - Why?
- When can a combiner be used?
  - Map output can be processed (“combined”) even through we do not see all values associated with the key
  - Combiner output can be interpreted by reducer
  - Word count, and many other counting applications can use a combiner.

Word Count with Combiner

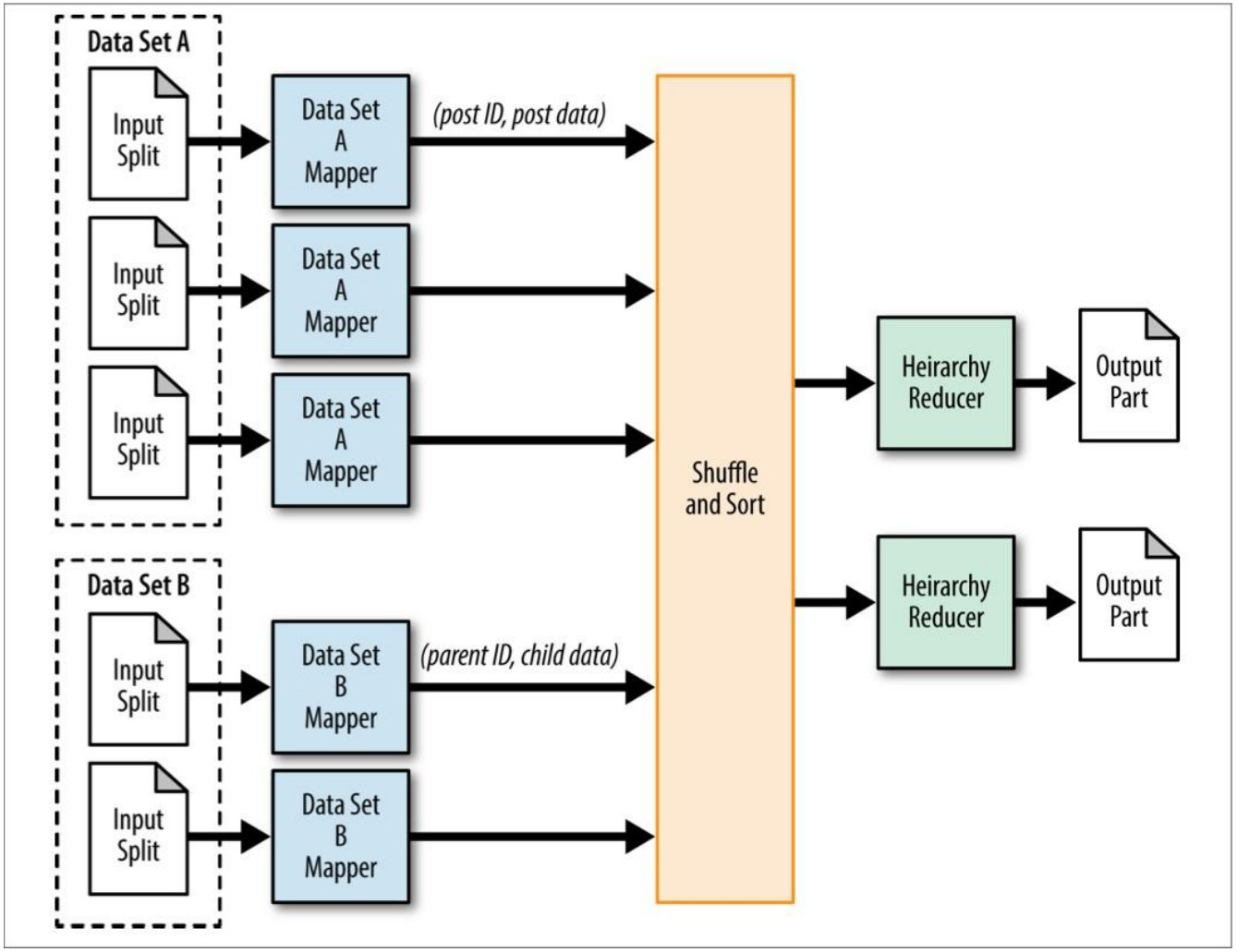


Multiple Inputs

- It is possible to define multiple mappers
- Each mapper can read a different input format
- Each mapper transforms the input data to a common format for output
  - Extracts the key
  - Puts the data into a common data structure

Data Flow

Figure 4–1 from MapReduce Design Patterns



Partitioning

- Organize “similar” records into partitions

- Why?
  - Future jobs will only focus on subsets of the data
- Partitioning schemes:
  - Time: hour, day, week, month, year
  - Geography: ZIP, DMA, state, time zone, country
  - Data source: web site
  - Data type

Partitioning

- No downside, as a mapReduce job can run over all partitions if needed

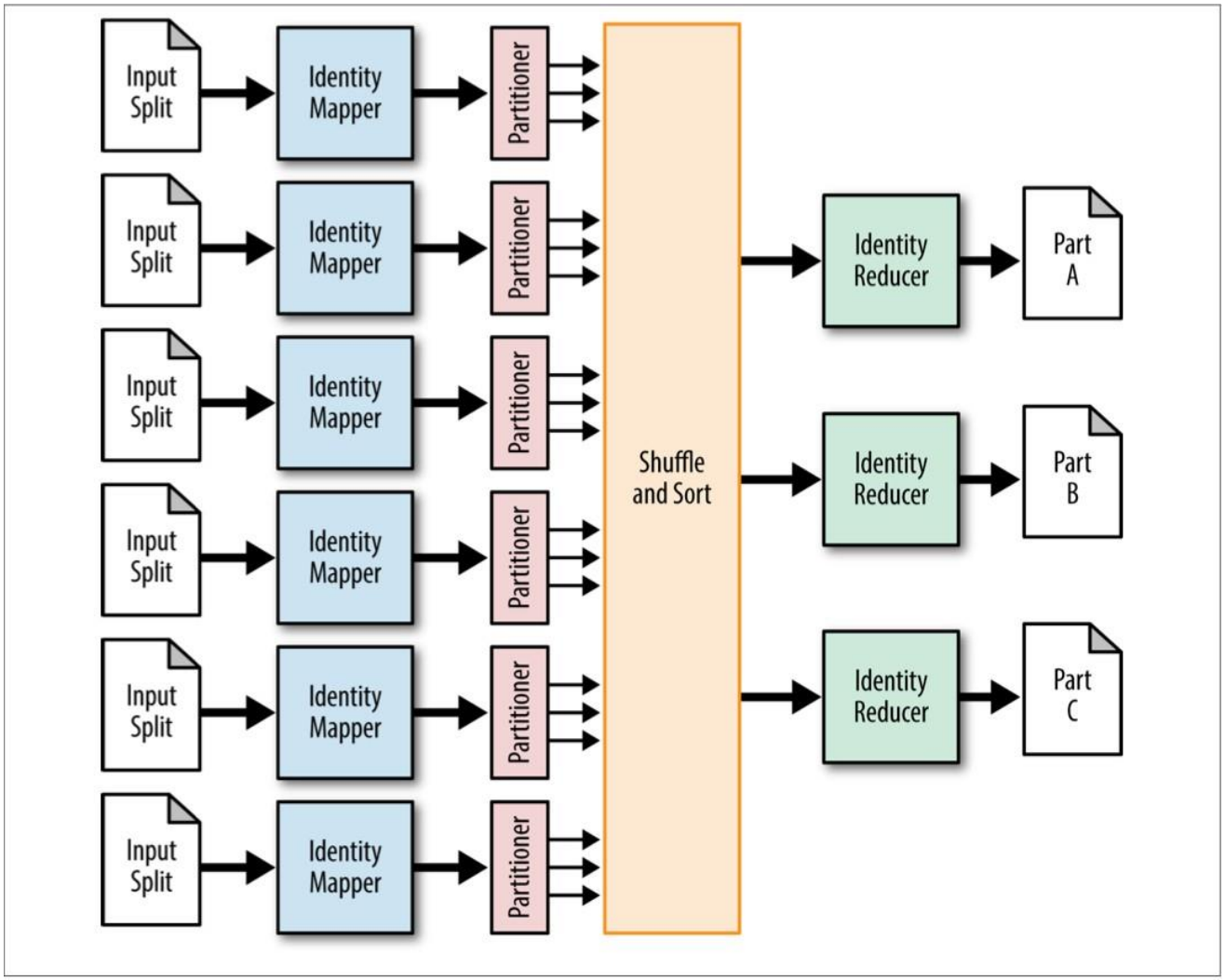
- We do need to know *a priori* how many partitions we want
  - Can run a job that scans and summarizes the data
  - Get possible values, and counts
  - Just like we did for user sessions

Partitioning

- Define a `Partitioner`
- Examines each `map()` output pair
- Computes a `partition` number

Data Flow

Figure 4-2 from MapReduce Design Patterns



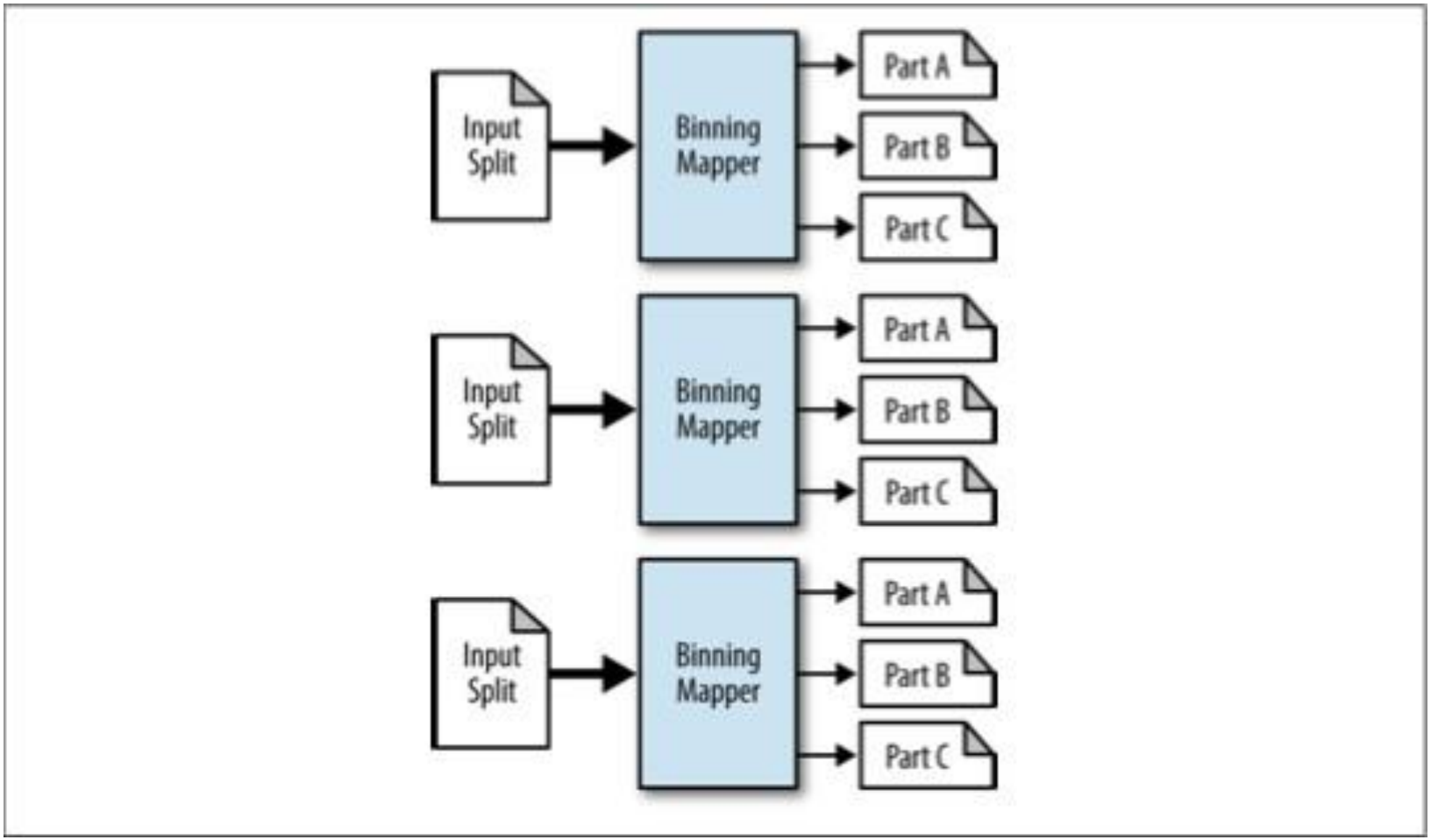
Binning

- Similar to partitioning
  - Want to organize output into categories
  - Map-only pattern (# reduce tasks set to 0)

- Mapper output written to output directories
- Uses `MultipleOutputs` class
  - Call `write()` on `MultipleOutputs`, not `Context`
  - For each category, each mapper writes a file
  - Expensive if many mappers and many categories

Binnig Data Flow

Figure 4–3 from MapReduce Design Patterns



Three Gotchas

- Avoid object creation if possible
  - Reuse Writable objects, change the payload
- Execution framework reuses value object in reducer
- Passing parameters via class statics

Getting Data to Mappers and Reducers

- Configuration parameters
  - Directly in the Job object for parameters
- “Side data”
  - DistributedCache
  - Mappers/reducers read from HDFS in setup method

Complex Data Types in Hadoop

- How do you implement complex data types?
- The easiest way:
  - Encoded it as Text, e.g., (a, b) = “a:b”
  - Use regular expressions to parse and extract data
  - Works, but pretty hack-ish
- 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:
  - JSON support and lots of useful Hadoop types

Debugging Hadoop

- First, take a deep breath
- Start small, start locally
- Build incrementally

Code Execution Environments

- Different ways to run code:
  - Plain Java
  - Local (standalone) mode
  - Pseudo-distributed mode
  - Fully-distributed mode

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
- Programming is still programming
  - 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