COMP 322: Parallel and Concurrent Programming

Lecture 6: Map/Reduce

Zoran Budimlić and Mack Joyner
{zoran, mjoyner}@rice.edu

http://comp322.rice.edu

Streaming data requirements have skyrocketed

AT&T processes roughly 30 petabytes per day through its telecommunications network

Facebook, Amazon, Twitter, etc, have comparable throughputs

IBM Watson knowledge base stored roughly 4 terabytes of data when winning at Jeopardy

The amount of data in the world was estimated to be 44 zettabytes at the beginning of 2020. 175 zettabytes by 2025.

By 2025, the amount of data generated each day is expected to reach 463 exabytes globally.

Electronic Arts process roughly 50 terabytes of data every day.

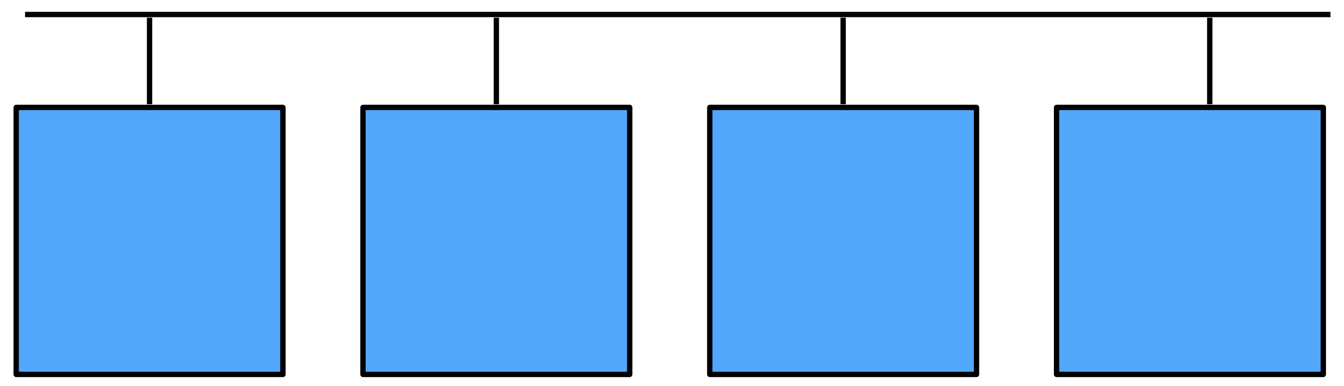
In 2019, nearly 695,000 hours of Netflix content were watched per minute across the world.

Parallelism enables processing of big data

Continuously streaming data needs to be processed at least as fast as it is accumulated, or we will never catch up

The bottleneck in processing very large data sets is dominated by the speed of disk access

More processors accessing more disks enables faster processing



But massive parallel programming is hard!

100’s of 1000’s of servers

Dozens of cores per server

Gigabytes (terabytes) of data per server

Where to begin?

“MapReduce: Simplified Data Processing on Large Clusters”. *Jeffrey Dean and Sanjay Ghemawat*. OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA (2004), pp. 137-150

"Our abstraction is inspired by the map and reduce primitives present in Lisp and many other functional languages."

Map/Reduce frameworks

- Hadoop MapReduce
  - Java based
  - All data is stored on Hadoops distributed file system (HDFS)
  - Can process enormous data sets
  - More in COMP 330!
- Apache Spark
  - Fits operations in memory
  - APIs for Java, Scala, Python and R
  - Requires a lot more computer memory
  - More in COMP 330!
- Many others
  - Phoenix++, MARISSA, MARIANE, MapReduce-MPI, SASReduce, MARLA, DRYAD, Themis, MR4C…

MapReduce Pattern

Apply Map function f to user supplied record of key-value pairs

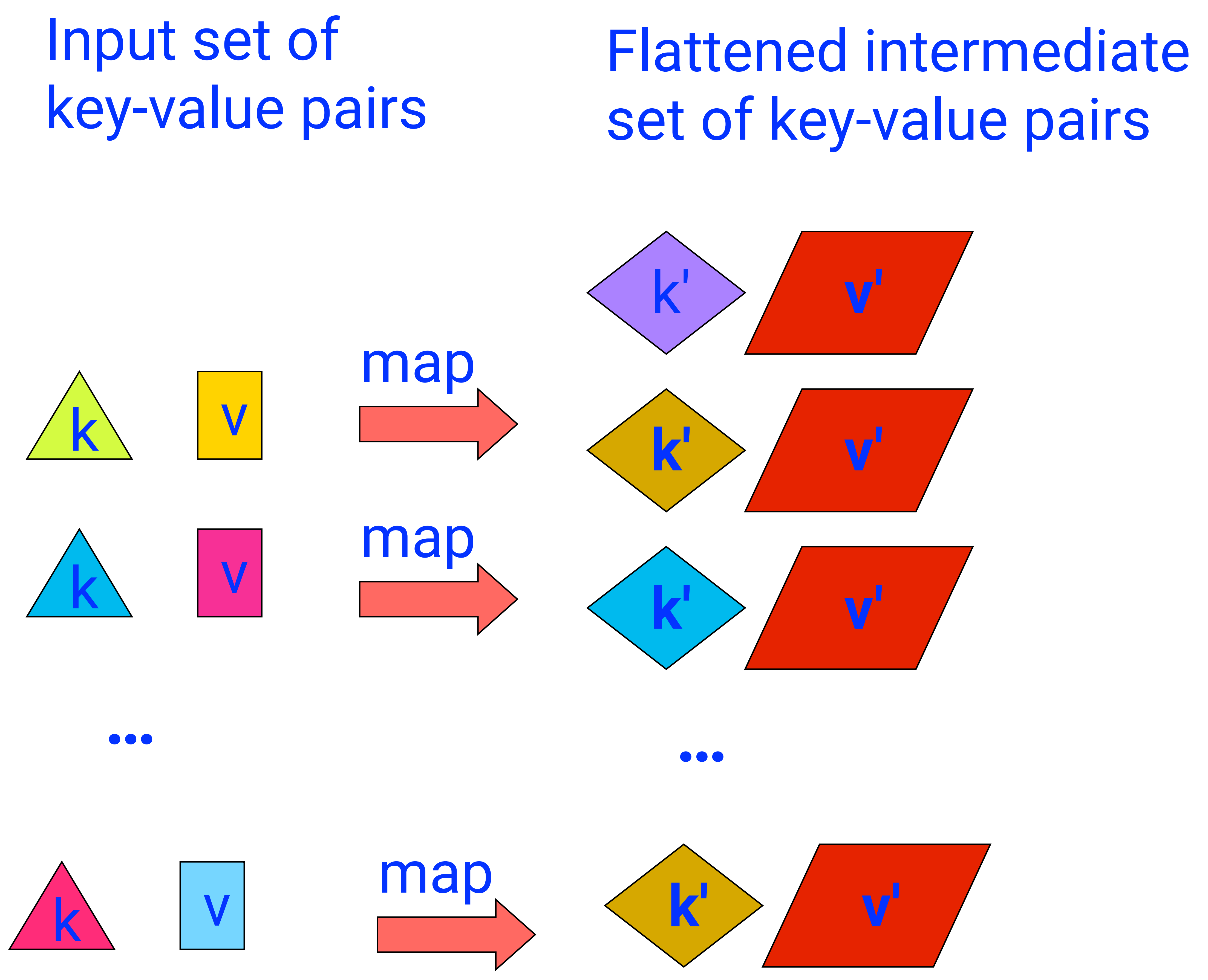
Compute set of intermediate key/value pairs

Apply Reduce operation g to all values that share the same key to combine derived data properly

- Often produces smaller set of values

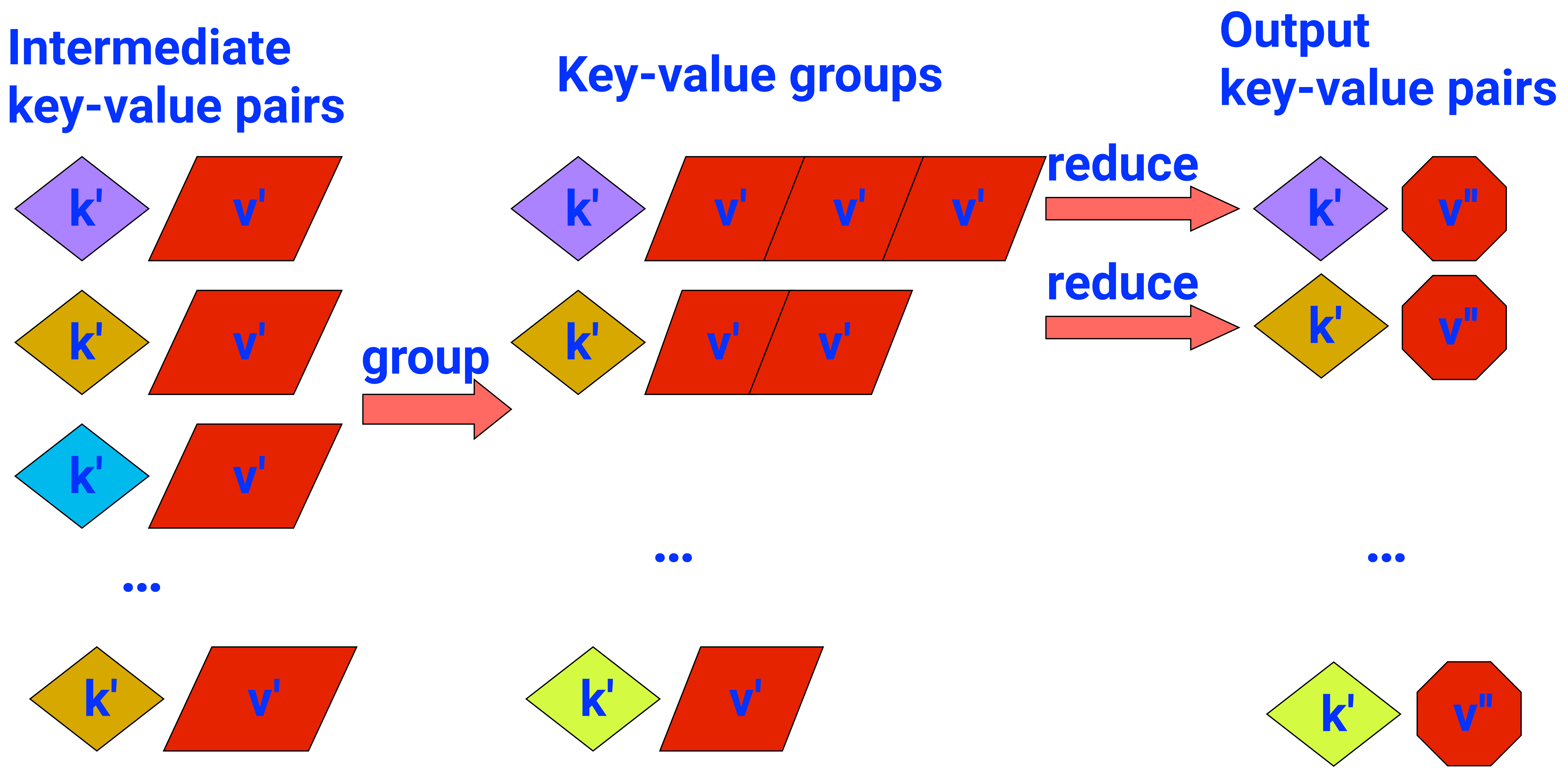
User supplies Map and Reduce operations in a functional model so that the system can parallelize them, and also re-execute them for fault tolerance

MapReduce: Map Step


Input set of key-value pairs
Flattened intermediate set of key-value pairs
k
v
map
k'
v'
…

Source: http://infolab.stanford.edu/~ullman/mining/2009/mapreduce.ppt

MapReduce: Reduce Step



Source: http://infolab.stanford.edu/~ullman/mining/2009/mapreduce.ppt

Map Reduce: Summary

Input set is of the form {(k1, v1), . . . (kn, vn)}, where (ki, vi) consists of a key, ki, and a value, vi.

- Assume key and value objects are immutable

Map function f generates sets of intermediate key-value pairs, f(ki,vi) = {(k1' ,v1'),...(km',vm')}. The km' keys can be different from ki key in the map function.

- Assume that a flatten operation is performed as a post-pass after the map operations, so as to avoid dealing with a set of sets. In other words, assume that a FlatMap operation is used.

Reduce operation groups together intermediate key-value pairs, {(k', vj')} with the same k’, and generates a reduced key-value pair, (k',v"), for each such k’, using reduce function g

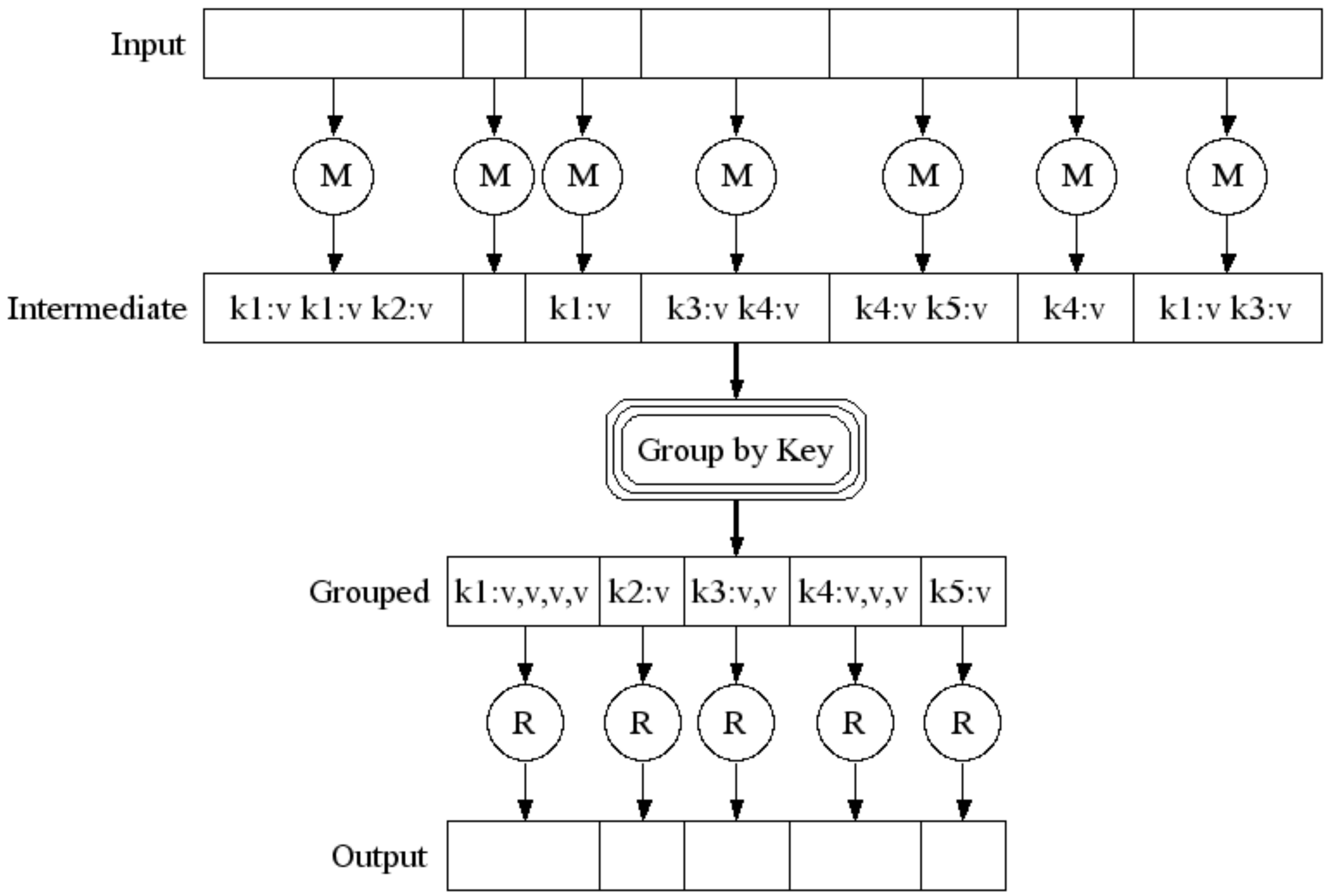
Google Uses MapReduce

Web crawl: Find outgoing links from HTML documents, aggregate by target document

Google Earth: Stitching overlapping satellite images to remove seams and to select high-quality imagery

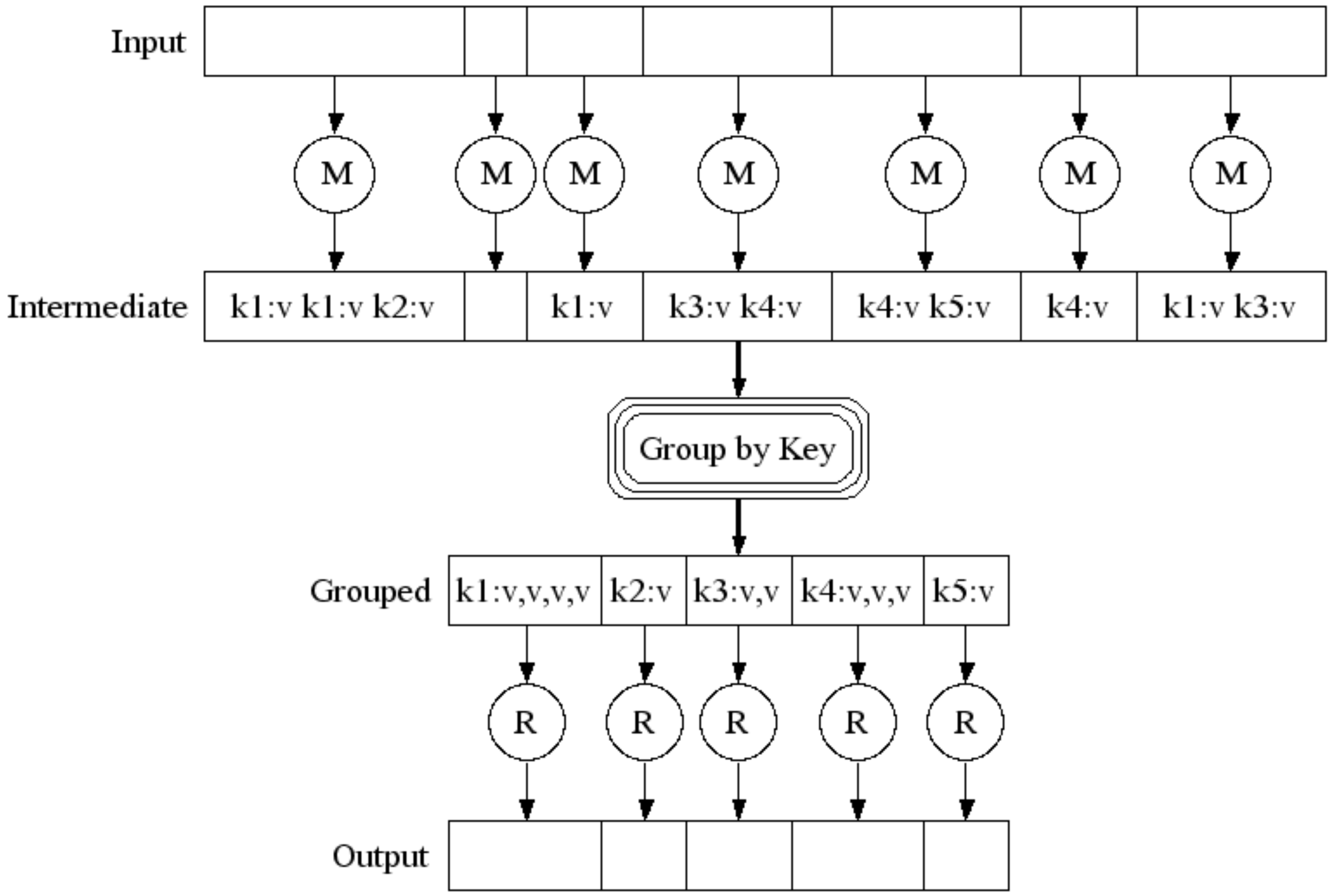
Google Maps: Processing all road segments on Earth and render map tile images that display segments

MapReduce Execution



MapReduce Execution

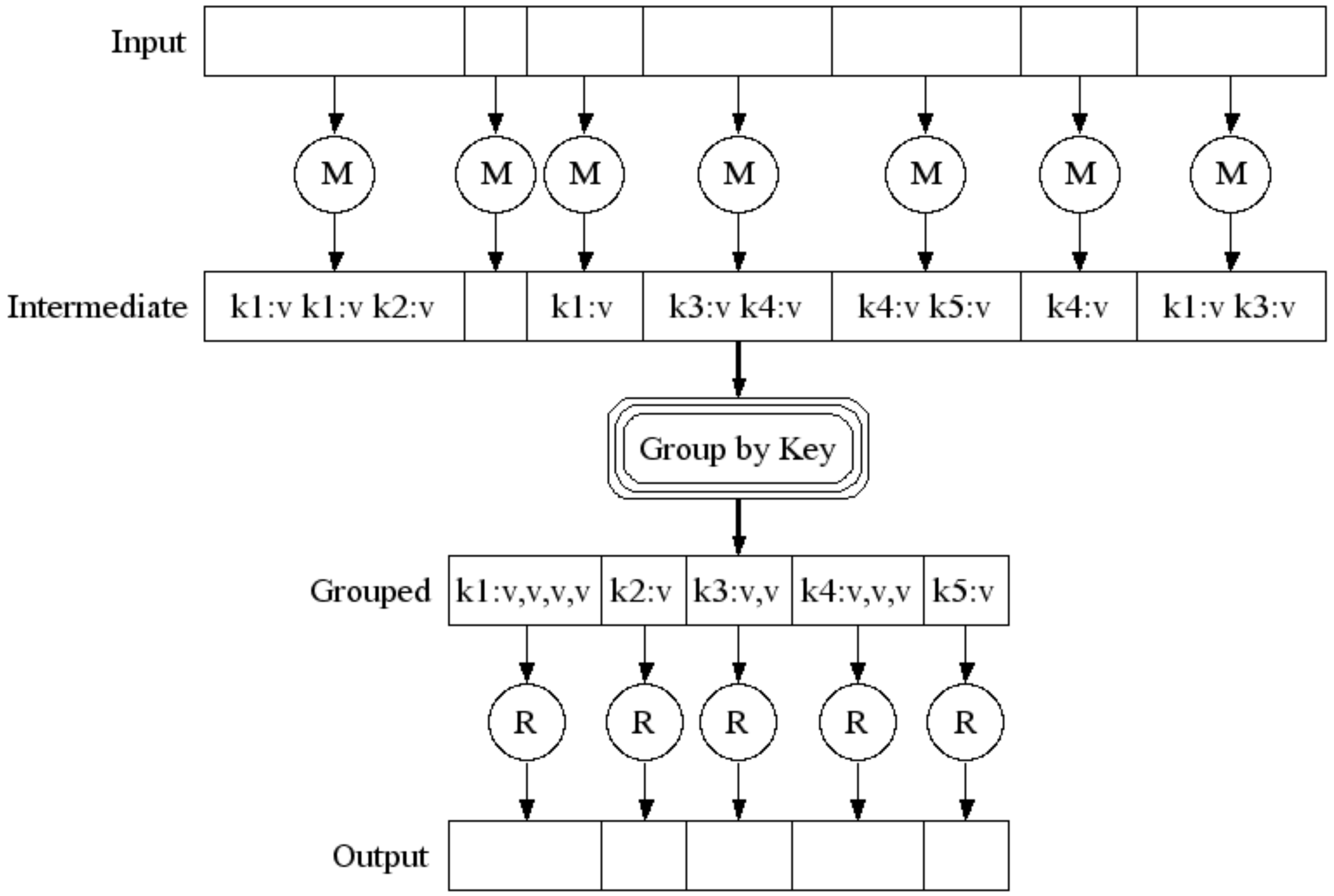
Fine granularity tasks: many more map tasks than machines



MapReduce Execution

Fine granularity tasks: many more map tasks than machines

Bucket sort to get same keys together

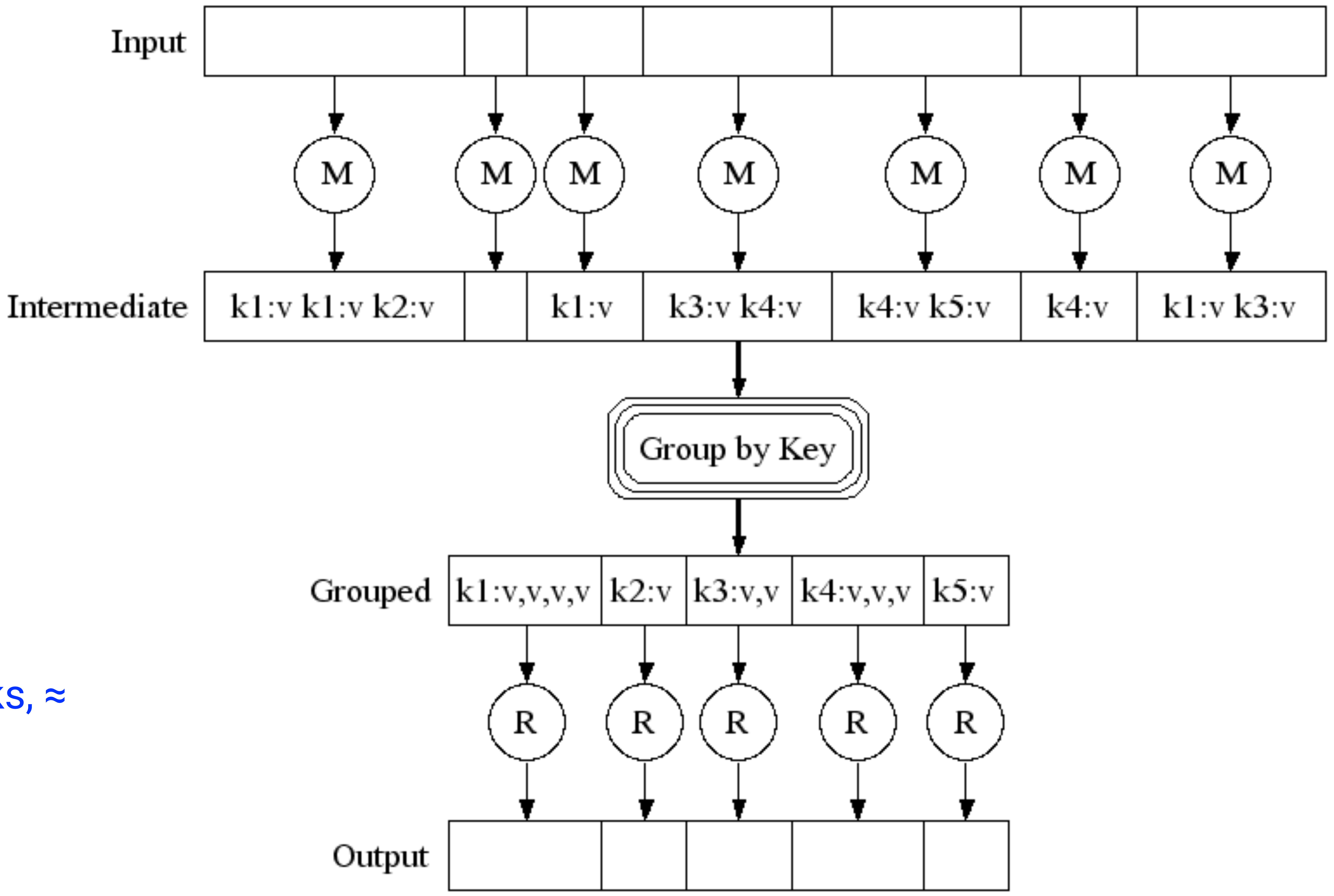


MapReduce Execution

Fine granularity tasks: many more map tasks than machines

Bucket sort to get same keys together

2000 servers => ≈ 200,000 Map Tasks, ≈ 5,000 Reduce tasks



Word Count Example

In: set of words

Out: set of (word,count) pairs

Algorithm:

1. For each in word W, emit (W, 1) as a key-value pair (map step).
2. Group together all key-value pairs with the same key (reduce step).
3. Perform a sum reduction on all values with the same key(reduce step).

All map operations in step 1 can execute in parallel with only local data accesses

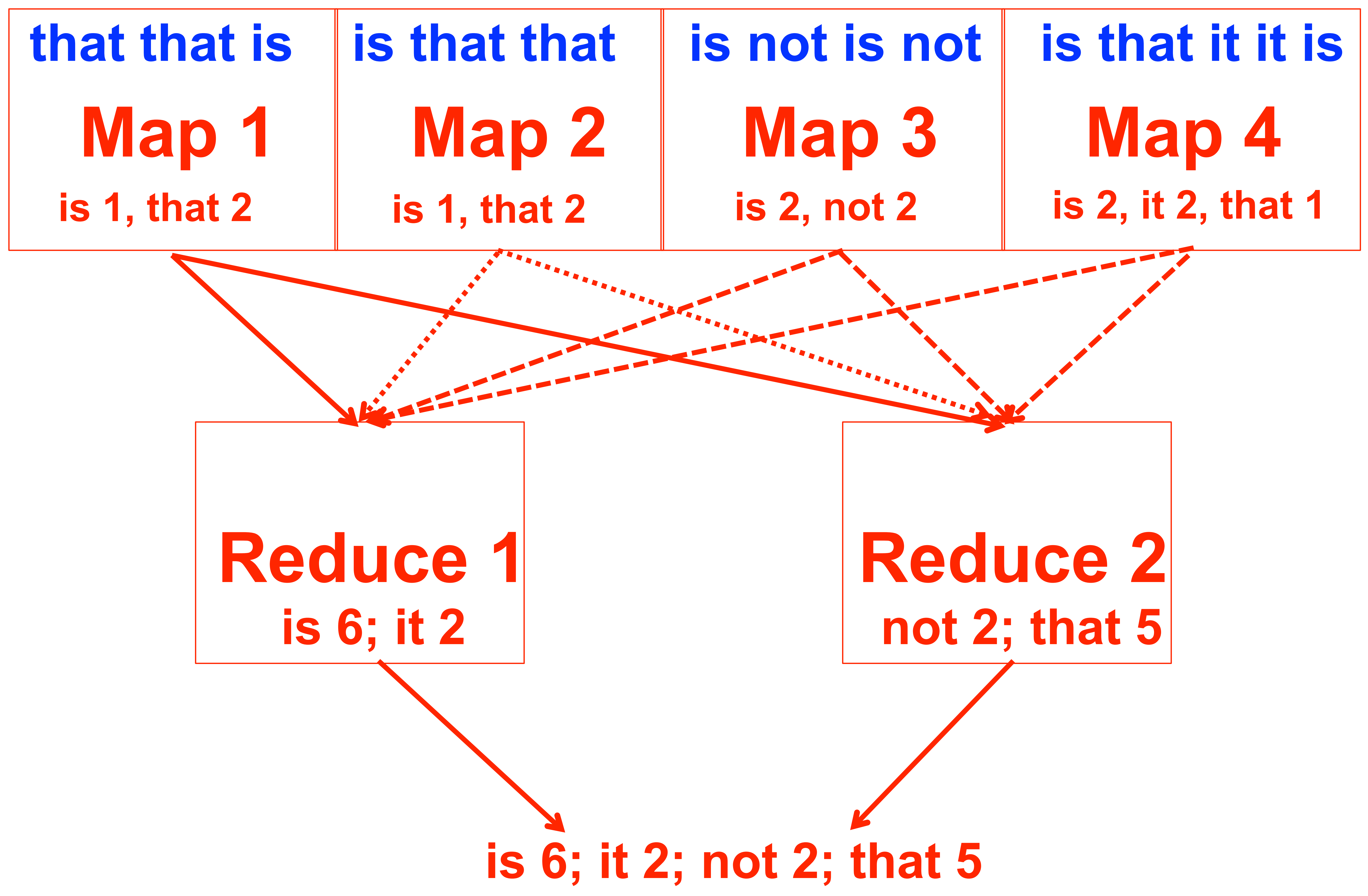
Step 2 may involve a major reshuffle of data as all key-value pairs with the same key are grouped together.

Step 3 performs a standard reduction algorithm for all values with the same key, and in parallel for different keys.

Pseudocode for Word Count

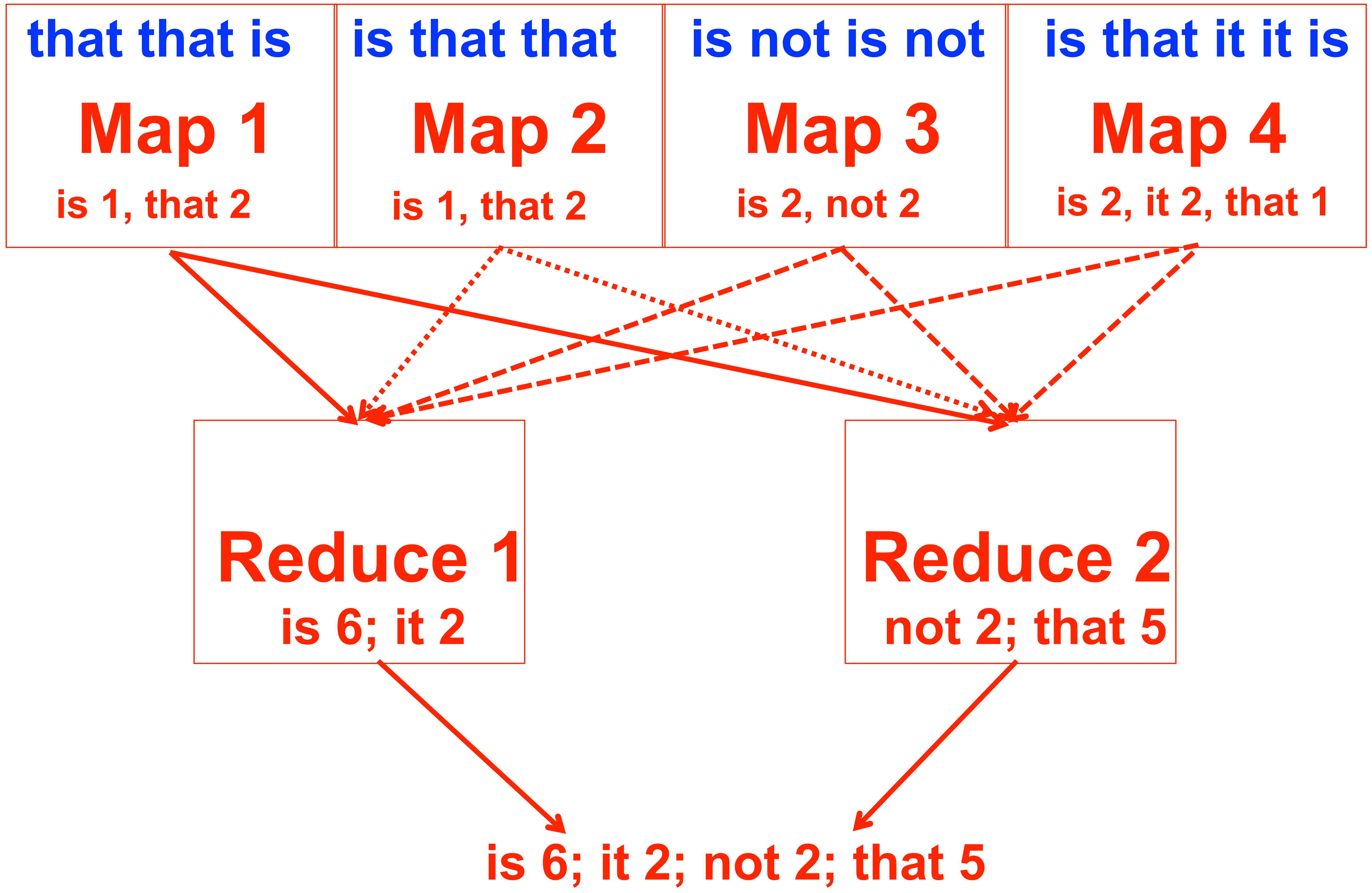
```
<String, Integer> map(String inKey, String inValue):
  // inKey: document name
  // inValue: document contents
  for each word w in inValue:
    emitIntermediate(w, 1) // Produce count of words

<Integer> reduce(String outKey, Iterator<Integer> values):
  // outKey: a word
  // values: a list of counts
   Integer result = 0
   for each v in values:
     result += v // the value from map was an integer
   emit(result)
```

Example Execution of Word Count Program

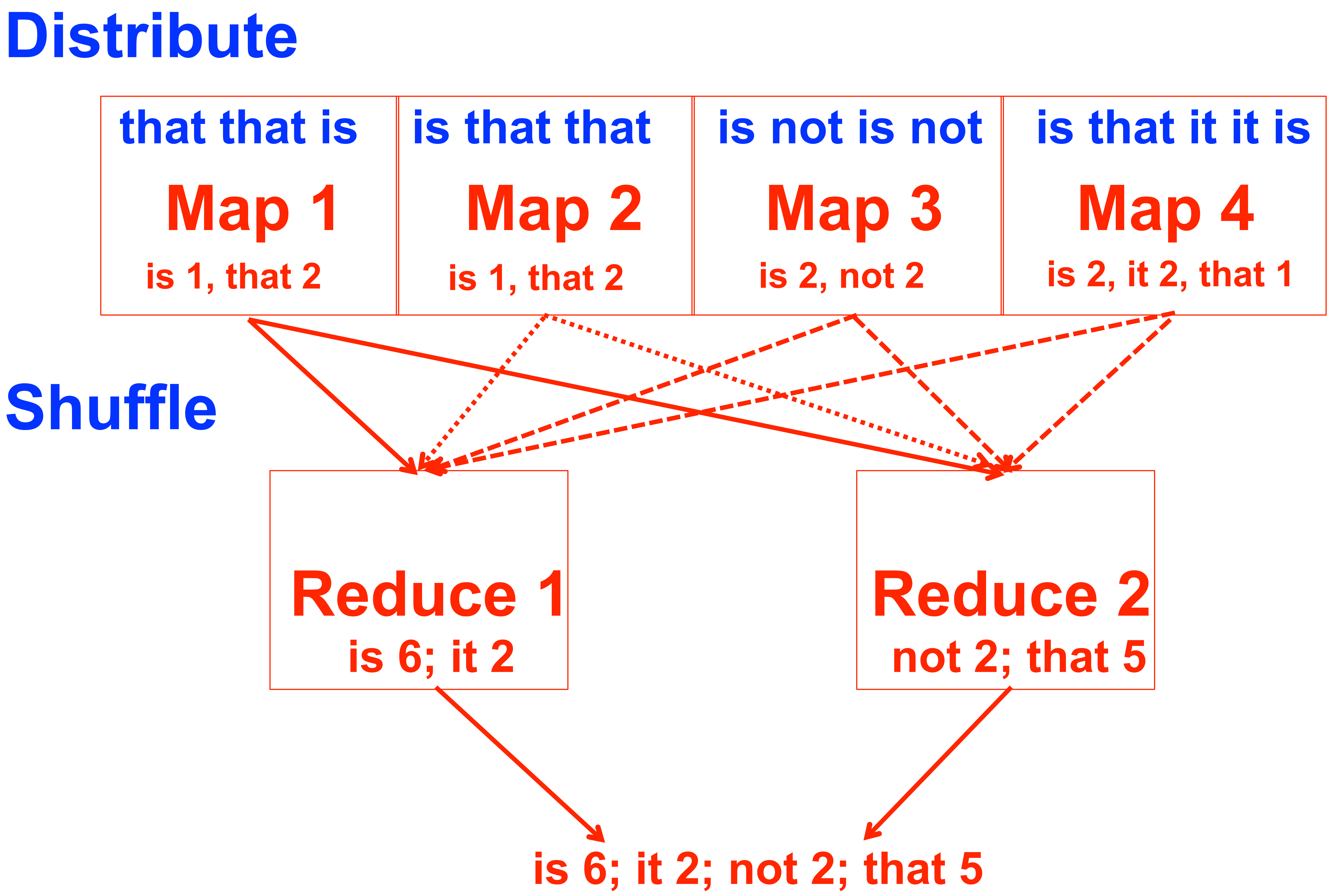


Example Execution of Word Count Program

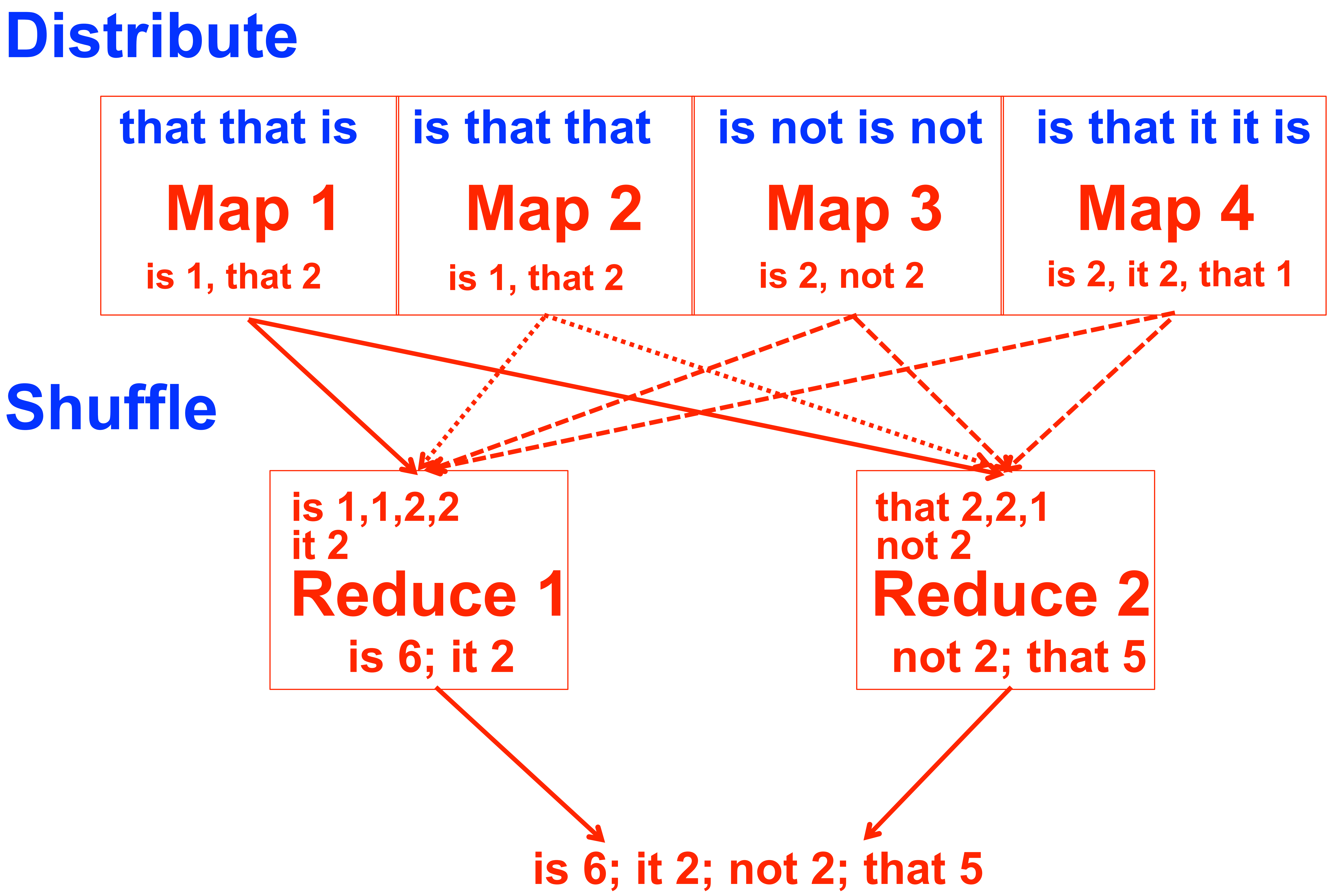
Distribute



Example Execution of Word Count Program



Example Execution of Word Count Program

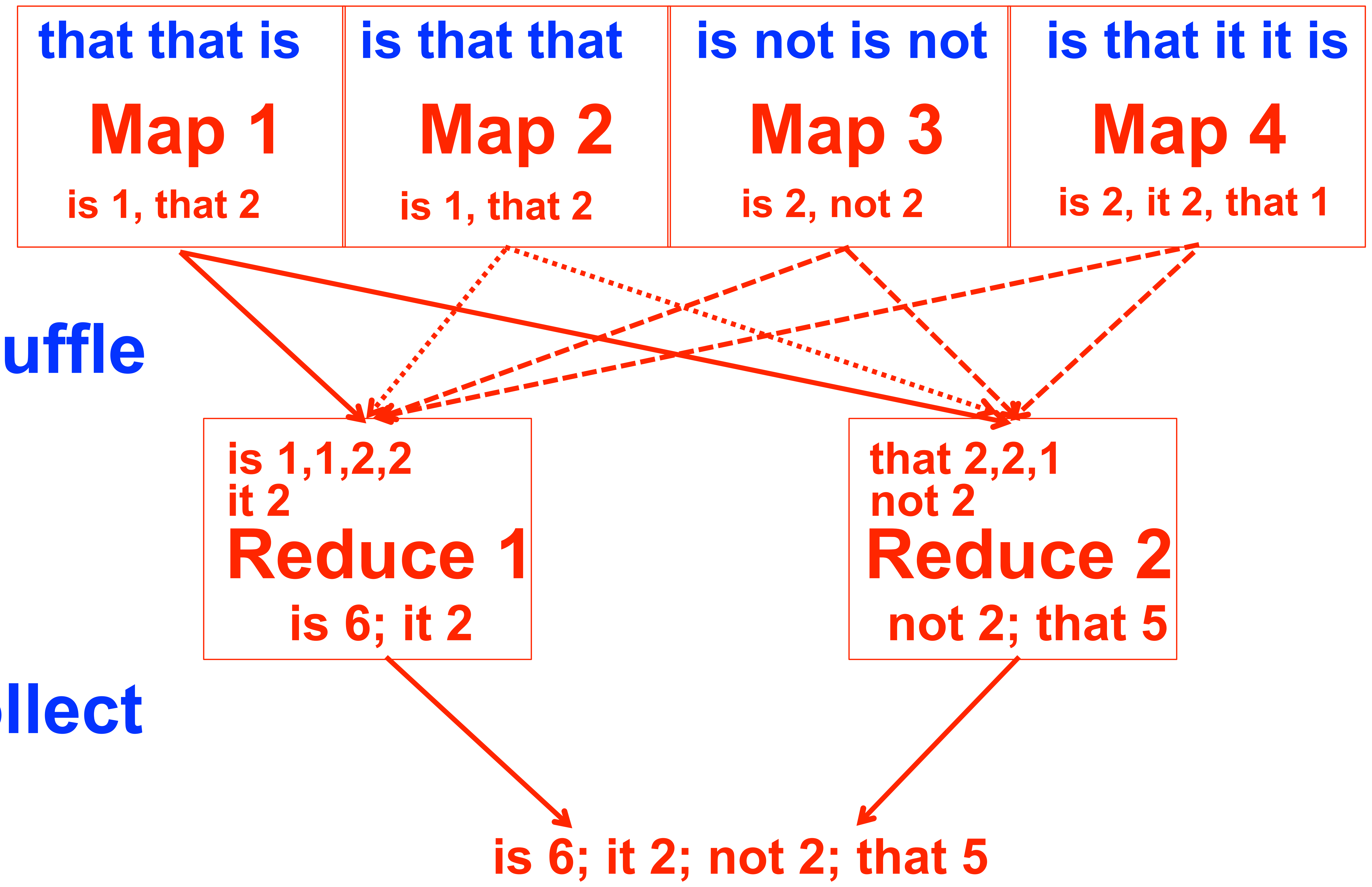


Example Execution of Word Count Program

Distribute

Shuffle

Collect



Simple parallel WordCount using Streams

```
var file = Arrays.asList("that", "that",  "is",
    "is", "that", "that", "is", "not", "is", "not", "is", "that", "it", "it", "is");

Map<String, Integer> result = file.parallelStream()
    .collect(groupingBy(Function.identity(), summingInt(e -> 1)));

result.forEach((k, v) -> System.out.println("The word \"" + k + "\" appears " + v + " times"));
```

Simple parallel WordCount using Streams

```
var file = Arrays.asList("that", "that",  "is",
    "is", "that", "that", "is", "not", "is", "not", "is", "that", "it", "it", "is");

Map<String, Integer> result = file.parallelStream()
    .collect(groupingBy(Function.identity(), summingInt(e -> 1)));

result.forEach((k, v) -> System.out.println("The word \"" + k + "\" appears " + v + " times"));
```

Map

Simple parallel WordCount using Streams

```
var file = Arrays.asList("that", "that",  "is",
    "is", "that", "that", "is", "not", "is", "not", "is", "that", "it", "it", "is");

Map<String, Integer> result = file.parallelStream()
    .collect(groupingBy(Function.identity(), summingInt(e -> 1)));

result.forEach((k, v) -> System.out.println("The word \"" + k + "\" appears " + v + " times"));
```

Reduce

Map

Simple parallel WordCount using Streams

```
var file = Arrays.asList("that", "that",  "is",
    "is", "that", "that", "is", "not", "is", "not", "is", "that", "it", "it", "is");

Map<String, Integer> result = file.parallelStream()
    .collect(groupingBy(Function.identity(), summingInt(e -> 1)));

result.forEach((k, v) -> System.out.println("The word \"" + k + "\" appears " + v + " times"));
```

Shuffle

Reduce

Map

Simple parallel WordCount using Streams

```
var file = Arrays.asList("that", "that",  "is",
    "is", "that", "that", "is", "not", "is", "not", "is", "that", "it", "it", "is");

Map<String, Integer> result = file.parallelStream()
    .collect(groupingBy(Function.identity(), summingInt(e -> 1)));

result.forEach((k, v) -> System.out.println("The word \"" + k + "\" appears " + v + " times"));
```

Shuffle

Reduce

Map

```
The word "that" appears 5 times
The word "not" appears 2 times
The word "is" appears 6 times
The word "it" appears 2 times
```

“Distributed” parallel WordCount using Streams

```java
Collection<String>[] files = new Collection[4];
files[0] = Arrays.asList("that", "that",  "is");
files[1] = Arrays.asList("is", "that", "that");
files[2] = Arrays.asList("is", "not", "is", "not");
files[3] = Arrays.asList("is", "that", "it", "it", "is");
Map<String, Integer> result = Arrays.asList(files).parallelStream()
    .map(file ->
        file.parallelStream()
            .collect(groupingBy(Function.identity(), summingInt(e -> 1)))
    )
    .collect(
        HashMap<String, Integer>::new,
        (m1, m2) -> m2.forEach((k, v) -> m1.merge(k, v, Integer::sum)),
        (m1, m2) -> m2.forEach((k, v) -> m1.merge(k, v, Integer::sum))
    );
result.forEach((k, v) -> System.out.println("The word \"" + k + "\" appears " + v + " times"));
```

“Distributed” parallel WordCount using Streams

```
Collection<String>[] files = new Collection[4];
files[0] = Arrays.asList("that", "that",  "is");
files[1] = Arrays.asList("is", "that", "that");
files[2] = Arrays.asList("is", "not", "is", "not");
files[3] = Arrays.asList("is", "that", "it", "it", "is");
Map<String, Integer> result = Arrays.asList(files).parallelStream()
    .map(file ->
        file.parallelStream()
            .collect(groupingBy(Function.identity(), summingInt(e -> 1)))
    )
    .collect(
        HashMap<String, Integer>::new,
        (m1, m2) -> m2.forEach((k, v) -> m1.merge(k, v, Integer::sum)),
        (m1, m2) -> m2.forEach((k, v) -> m1.merge(k, v, Integer::sum))
    );
result.forEach((k, v) -> System.out.println("The word \"" + k + "\" appears " + v + " times"));
```

Parallel WordCount for each file

“Distributed” parallel WordCount using Streams

```
Collection<String>[] files = new Collection[4];
files[0] = Arrays.asList("that", "that",  "is");
files[1] = Arrays.asList("is", "that", "that");
files[2] = Arrays.asList("is", "not", "is", "not");
files[3] = Arrays.asList("is", "that", "it", "it", "is");
Map<String, Integer> result = Arrays.asList(files).parallelStream()
    .map(file ->
        file.parallelStream()
            .collect(groupingBy(Function.identity(), summingInt(e -> 1)))
    )
    .collect(
        HashMap<String, Integer>::new,
        (m1, m2) -> m2.forEach((k, v) -> m1.merge(k, v, Integer::sum)),
        (m1, m2) -> m2.forEach((k, v) -> m1.merge(k, v, Integer::sum))
    );
result.forEach((k, v) -> System.out.println("The word \"" + k + "\" appears " + v + " times"));
```

Parallel WordCount for each file

Accumulator, combine two hash maps

“Distributed” parallel WordCount using Streams

```
Collection<String>[] files = new Collection[4];
files[0] = Arrays.asList("that", "that",  "is");
files[1] = Arrays.asList("is", "that", "that");
files[2] = Arrays.asList("is", "not", "is", "not");
files[3] = Arrays.asList("is", "that", "it", "it", "is");
Map<String, Integer> result = Arrays.asList(files).parallelStream()
    .map(file ->
        file.parallelStream()
            .collect(groupingBy(Function.identity(), summingInt(e -> 1)))
    )
    .collect(
        HashMap<String, Integer>::new,
        (m1, m2) -> m2.forEach((k, v) -> m1.merge(k, v, Integer::sum)),
        (m1, m2) -> m2.forEach((k, v) -> m1.merge(k, v, Integer::sum))
    );
result.forEach((k, v) -> System.out.println("The word \"" + k + "\" appears " + v + " times"));
```

Parallel WordCount for each file

Accumulator, combine two hash maps

Combiner, combine two hash maps

“Distributed” parallel WordCount using Streams

```java
Collection<String>[] files = new Collection[4];
files[0] = Arrays.asList("that", "that",  "is");
files[1] = Arrays.asList("is", "that", "that");
files[2] = Arrays.asList("is", "not", "is", "not");
files[3] = Arrays.asList("is", "that", "it", "it", "is");
Map<String, Integer> result = Arrays.asList(files).parallelStream()
    .map(file ->
        file.parallelStream()
            .collect(groupingBy(Function.identity(), summingInt(e -> 1)))
    )
    .collect(
        HashMap<String, Integer>::new,
        (m1, m2) -> m2.forEach((k, v) -> m1.merge(k, v, Integer::sum)),
        (m1, m2) -> m2.forEach((k, v) -> m1.merge(k, v, Integer::sum))
    );
result.forEach((k, v) -> System.out.println("The word \"" + k + "\" appears " + v + " times"));
```

Parallel WordCount for each file

Accumulator, combine two hash maps

Combiner, combine two hash maps

```
The word "that" appears 5 times
The word "not" appears 2 times
The word "is" appears 6 times
The word "it" appears 2 times
```

Parallelized Map merges

```
Collection<String>[] files = new Collection[4];
files[0] = Arrays.asList("that", "that",  "is");
files[1] = Arrays.asList("is", "that", "that");
files[2] = Arrays.asList("is", "not", "is", "not");
files[3] = Arrays.asList("is", "that", "it", "it", "is");
Map<String, Integer> result = Arrays.asList(files).parallelStream()
    .map(file -> file.parallelStream()
        .collect(groupingBy(Function.identity(), summingInt(e -> 1)))
    )
    .collect(
        ConcurrentHashMap<String, Integer>::new,
        (m1, m2) ->
            m2.entrySet().parallelStream()
              .forEach(e -> m1.merge(e.getKey(), e.getValue(), Integer::sum)),
        (m1, m2) -> m2.entrySet().parallelStream()
            .forEach(e -> m1.merge(e.getKey(), e.getValue(), Integer::sum))
    );
result.forEach((k, v) -> System.out.println("The word \"" + k + "\" appears " + v + " times"));
```

Parallelized Map merges

```
Collection<String>[] files = new Collection[4];
files[0] = Arrays.asList("that", "that",  "is");
files[1] = Arrays.asList("is", "that", "that");
files[2] = Arrays.asList("is", "not", "is", "not");
files[3] = Arrays.asList("is", "that", "it", "it", "is");
Map<String, Integer> result = Arrays.asList(files).parallelStream()
    .map(file -> file.parallelStream()
        .collect(groupingBy(Function.identity(), summingInt(e -> 1)))
    )
    .collect(
        ConcurrentHashMap<String, Integer>::new,
        (m1, m2) ->
            m2.entrySet().parallelStream()
              .forEach(e -> m1.merge(e.getKey(), e.getValue(), Integer::sum)),
        (m1, m2) -> m2.entrySet().parallelStream()
            .forEach(e -> m1.merge(e.getKey(), e.getValue(), Integer::sum))
    );
result.forEach((k, v) -> System.out.println("The word \"" + k + "\" appears " + v + " times"));
```

Have to use a ConcurrentHashMap since we are doing the merge in parallel

Parallelized Map merges

```
Collection<String>[] files = new Collection[4];
files[0] = Arrays.asList("that", "that",  "is");
files[1] = Arrays.asList("is", "that", "that");
files[2] = Arrays.asList("is", "not", "is", "not");
files[3] = Arrays.asList("is", "that", "it", "it", "is");
Map<String, Integer> result = Arrays.asList(files).parallelStream()
     .map(file -> file.parallelStream()
         .collect(groupingBy(Function.identity(), summingInt(e -> 1)))
     )
     .collect(
         ConcurrentHashMap<String, Integer>::new,
         (m1, m2) ->
             m2.entrySet().parallelStream()
               .forEach(e -> m1.merge(e.getKey(), e.getValue(), Integer::sum)),
         (m1, m2) -> m2.entrySet().parallelStream()
             .forEach(e -> m1.merge(e.getKey(), e.getValue(), Integer::sum))
     );
result.forEach((k, v) -> System.out.println("The word \"" + k + "\" appears " + v + " times"));
```

Have to use a ConcurrentHashMap since we are doing the merge in parallel

Accumulator, combine two hash maps in parallel

Parallelized Map merges

```
Collection<String>[] files = new Collection[4];
files[0] = Arrays.asList("that", "that",  "is");
files[1] = Arrays.asList("is", "that", "that");
files[2] = Arrays.asList("is", "not", "is", "not");
files[3] = Arrays.asList("is", "that", "it", "it", "is");
Map<String, Integer> result = Arrays.asList(files).parallelStream()
    .map(file -> file.parallelStream()
        .collect(groupingBy(Function.identity(), summingInt(e -> 1)))
    )
    .collect(
        ConcurrentHashMap<String, Integer>::new,
        (m1, m2) ->
            m2.entrySet().parallelStream()
              .forEach(e -> m1.merge(e.getKey(), e.getValue(), Integer::sum)),
        (m1, m2) -> m2.entrySet().parallelStream()
            .forEach(e -> m1.merge(e.getKey(), e.getValue(), Integer::sum))
    );
result.forEach((k, v) -> System.out.println("The word \"" + k + "\" appears " + v + " times"));
```

Have to use a ConcurrentHashMap since we are doing the merge in parallel

Accumulator, combine two hash maps in parallel

Combiner, combine two hash maps in parallel

Parallelized Map merges

```
Collection<String>[] files = new Collection[4];
files[0] = Arrays.asList("that", "that",  "is");
files[1] = Arrays.asList("is", "that", "that");
files[2] = Arrays.asList("is", "not", "is", "not");
files[3] = Arrays.asList("is", "that", "it", "it", "is");
Map<String, Integer> result = Arrays.asList(files).parallelStream()
    .map(file -> file.parallelStream()
        .collect(groupingBy(Function.identity(), summingInt(e -> 1)))
    )
    .collect(
        ConcurrentHashMap<String, Integer>::new,
        (m1, m2) ->
            m2.entrySet().parallelStream()
              .forEach(e -> m1.merge(e.getKey(), e.getValue(), Integer::sum)),
        (m1, m2) -> m2.entrySet().parallelStream()
            .forEach(e -> m1.merge(e.getKey(), e.getValue(), Integer::sum))
    );
result.forEach((k, v) -> System.out.println("The word \"" + k + "\" appears " + v + " times"));
```

Have to use a ConcurrentHashMap since we are doing the merge in parallel

Accumulator, combine two hash maps in parallel

Combiner, combine two hash maps in parallel

```
The word "that" appears 5 times
The word "not" appears 2 times
The word "is" appears 6 times
The word "it" appears 2 times
```

Summary

- Map/Reduce is a programming model
  - Limited expressiveness
  - If you can solve your problem using the model, and your problem is very large, go for it
- Heavily used in industry for processing large data
  - Hadoop, Spark, many others…
- Heavily based on functional programming ideas
  - map, filter, fold, lambdas
- Employs very similar ideas to Java Streams
- Easy to parallelize across large number of servers