CS523 - BDT
Big Data Technologies

Advanced MapReduce

(Combiner, Partitioner, In-Mapper Combining)

MapReduce: Recap

- Input and output: each a set of key/value pairs.
- Two functions implemented by users:

  Map (k1, v1) -> list(k2, v2)
  - takes an input key/value pair
  - produces a set of intermediate key/value pairs

  Reduce (k2, list(v2)) -> list(k3, v3)
  - takes a set of values for an intermediate key
  - produces a set of output value
  - MapReduce framework guarantees that all values associated with the same key are brought together in the reducer.

Local Aggregation

- In Hadoop, intermediate results (i.e. map outputs) are written to local disk before being sent over the network.
- Network and disk latencies are expensive and so Hadoop uses the concept of local aggregation.
- Local aggregation of intermediate results reduces the number of key-value pairs that need to be shuffled from the mappers to the reducers.

Need for Local Aggregation

- In the canonical example of word counting, a key-value pair is emitted for every word found.
- Imagine the word count example on a text containing one million times the word "the".
- Without local aggregation, the mapper will send one million key/value pairs of the form **<the, 1>**.
- This extremely high numbers of "intermediate" key-value pairs (these are the key-value pairs being sent from the mappers to the reducer(s)) is a pain point in the speed of completing the overall M/R job.
- With local aggregation techniques, it is possible to send much less key/value pairs to the reducer of the form ***<the, N>*** with N a number potentially much bigger than 1.

Local Aggregation with Combiner

- The first technique for local aggregation is the Combiner.
- Combiners are an optimization in MapReduce that allow for local aggregation before the shuffle and sort phase.
- The Combiner is a "mini-reduce" process which operates only on data generated by one mapper.
- Reduce tasks can be used as combiner if commutative & associative.
- Combiners are run on map machines after map phase and they aaggregate map outputs with the same key.
- This results in a reduction in the number of intermediate key-value pairs that need to be shuffled across the network (less network traffic) from the order of *total number of terms* in the input split to the order of the *number of unique terms* in the input split.
- So, combiners help to reduce the number of (key, value) pairs sent to the reducers, thus saving bandwidth.

Effect of Combiners

```
Map (String lineOffset, String line):
    for each word w in line:
        Emit(w, 1);

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

What's the impact of combiners?

What is the number of records being shuffled?

- without combiners?
- with combiners?

Need for Partitioner

- Number of default reducers is 1. So usually the o/p data from the reducer is contained in 1 file.
- This is OK for small datasets, but if the output is large (more than a few tens of gigabytes, say) then it's normally better to have a partitioned file, so you take advantage of the cluster parallelism for the reducer tasks.
- `setNumReduceTask()` method of Job object is used to specify the number of reducers that you need.

Default Partitioner

- For the case of multiple reducers, if there's no Partitioner specified then MR framework uses it's default partitioner which is ***HashPartitioner***.
- It involves computing the hash value of the key and then taking the *mod* of that value with the number of reducers specified.
- This assigns approximately the same number of keys to each reducer.

```
public class HashPartitioner<K, V> extends Partitioner<K, V>
{
  public int getPartition(K key, V value, int numReduceTasks)
  {
     return (key.hashCode() & Integer.MAX_VALUE) %
                                              numReduceTasks;
  }
}
```

Custom Partitioner

- If we want to send specific (key, value) pairs to specific reducer then the default *HashPartitioner* will not work; we need to write our own custom partitioner.

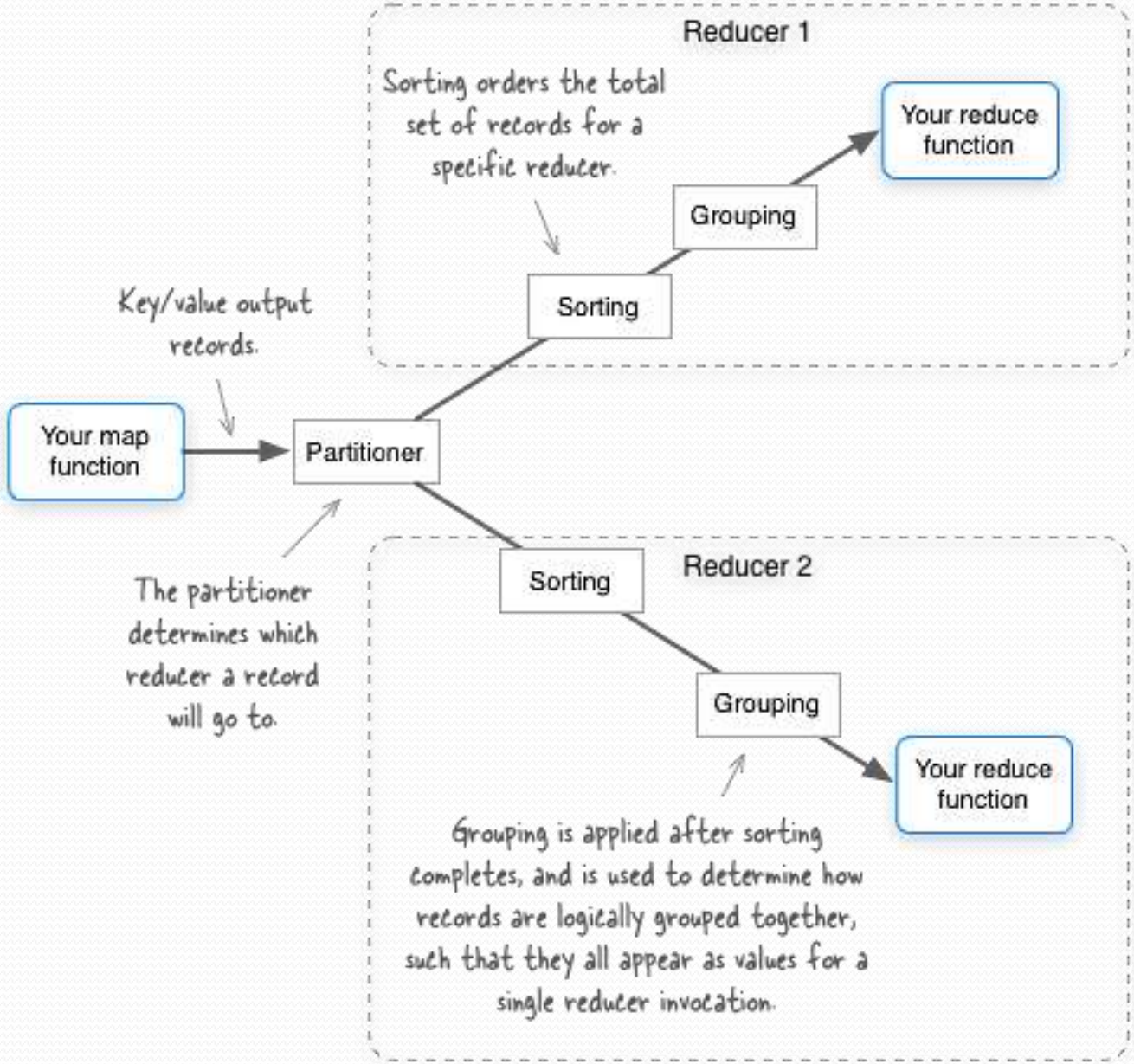
```
// K can be (k1,k2)
int getPartition(K key, V value, int numReduceTasks)
{
   if(key.ch(0) < 'd') return 0;
   if(key.ch(0) < 'g') return 1;
   if(key.ch(0) < 'm') return 2;
   else return 3;
}
```

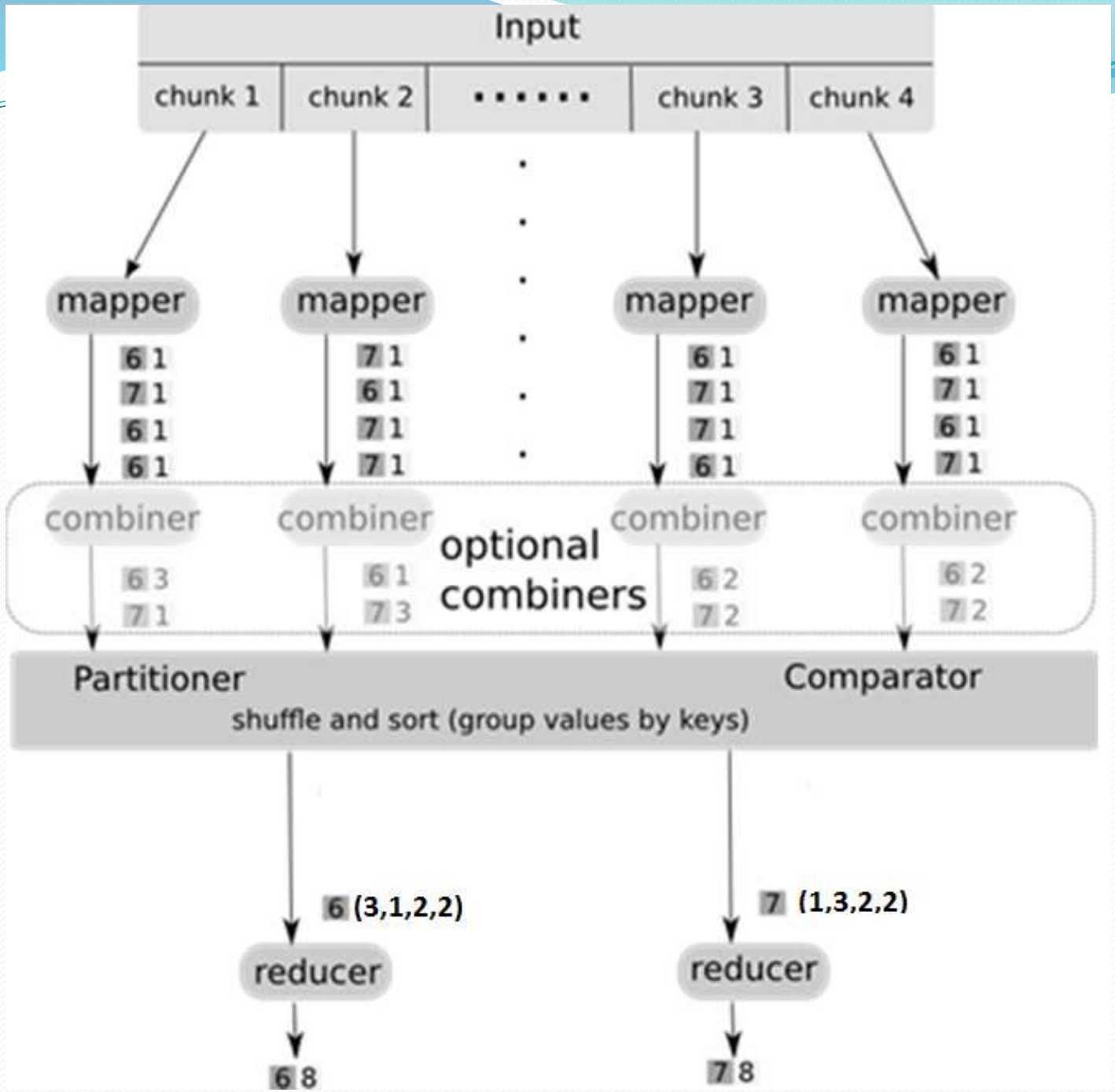
- How many maximum partitioners can there be?
- In other words, how many reducers can there be?

Partitioner

- Partitioner partitions the key space.
- Partitioners are responsible for dividing up the intermediate key space and assigning intermediate key-value pairs to the reducers.
- Partitioner specifies the reduce task to which an intermediate key-value pair must be copied.
- The total number of partitions created by the Partioner is the same as the number of reduce tasks for the job. Within each reducer, keys are processed in sorted order (which is how the “group by" is implemented).

Partitioner Functionality





Constraints on Combiners

- With or without combiner should not affect algorithm correctness.
- Combiners and reducers MUST share the same method signature
  - Combiner i/p and o/p key-value types must match reducer i/p key-value types (mapper o/p key-value type)
- Combiners can only be used on the functions that are commutative(a.b = b.a) and associative {a.(b.c) = (a.b).c} .
  That means sometimes, reducers can serve as combiners.
  Often, not…

Constraints on Combiner

- Because Combiner is an optimization, Hadoop does not guarantee how many times the combiner is applied, or that it is even applied at all!!!
- The combiner is provided as a semantics-preserving optimization to the execution framework, which has the option of using it, perhaps multiple times, or not at all.
- So a given MapReduce job should not depend on the combiner executions and should always produce the same results with or without combiner.
- In some cases such indeterminism is unacceptable, which is exactly why programmers often choose to perform their own local aggregation in the mappers.

Modified Word Count – Version 1

```
class Mapper
    Map (String lineOffset, String line):
        H ← new HashMap
        for each word w in line:
            H{w} ← H{w} + 1
        for all words w in H do
            Emit( w, H{w} );
```

Tally counts for the entire record(line)

Are combiners still needed?

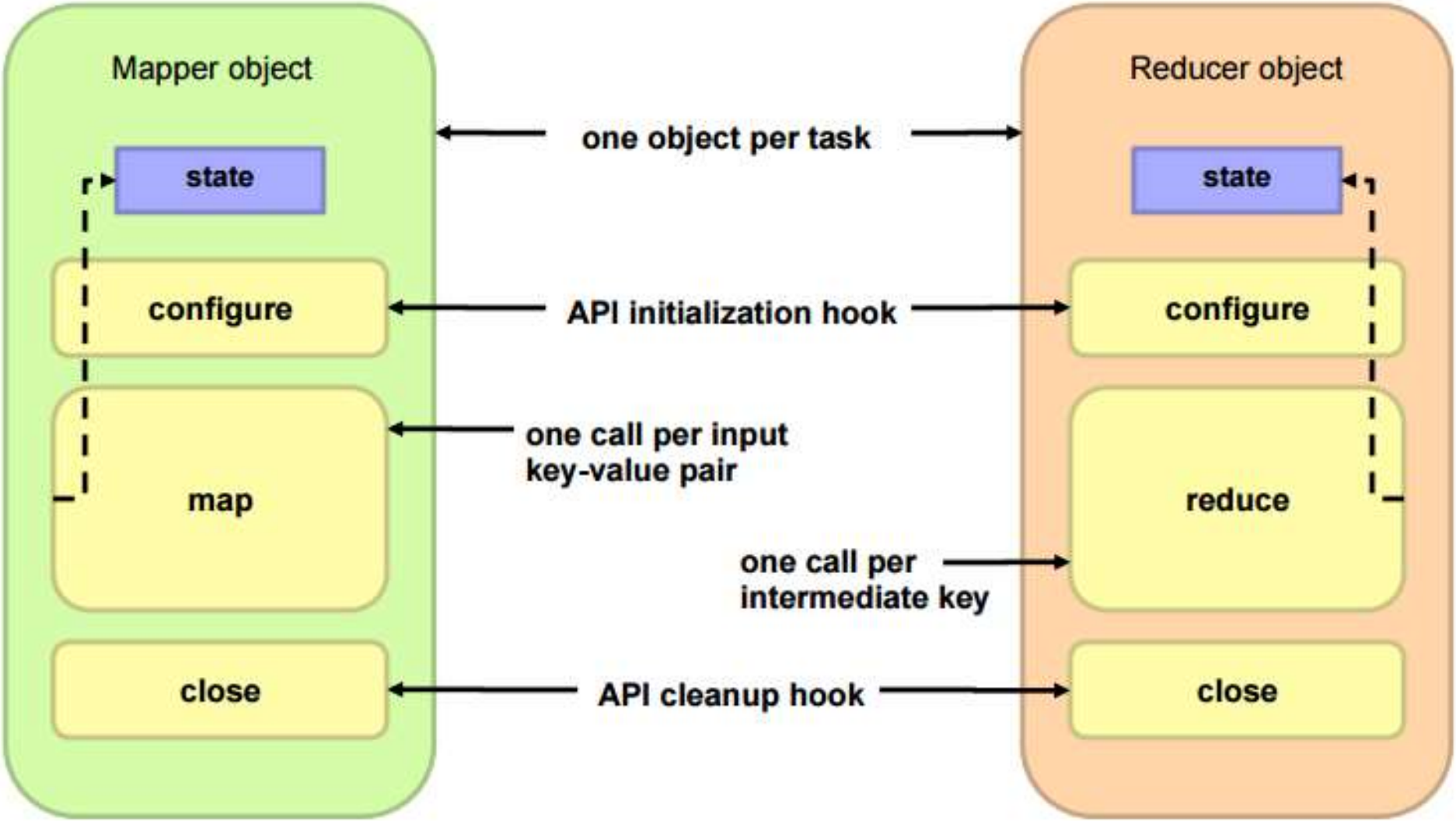
Modified Word Count – Version 1

- Only mapper is modified. An associative array (i.e., *Map* in Java) is introduced inside the mapper to tally up term counts within a single record.
- Instead of emitting a key-value pair for each term in the record, this version emits a key-value pair for each **unique** term in the record (line).
- Given that some words appear frequently within a line, this can yield substantial savings in the number of intermediate key-value pairs emitted, especially for long lines.

Modified Word Count – Version 2

- This basic idea can be taken one step further, as shown next (once again, only the mapper is modified).
- Recall, a (Java) mapper object is created for each map task, which is responsible for processing a block of input key-value pairs.
- Prior to processing any input key-value pairs, the mapper's **setup()** method is called, which is an API hook for user-specified code. In this case, we initialize a map for holding term counts.

State Preservation



Modified Word Count – Version 2

- Since it is possible to preserve state across multiple calls of the Map method (for each input key-value pair), we can continue to accumulate partial term counts in the associative array across multiple records, and emit key-value pairs only when the mapper has processed all records.
- That is, emission of intermediate data is deferred until the Close method in the pseudo-code.
- Recall that this API hook provides an opportunity to execute user-specified code after the Map method has been applied to all input key-value pairs of the input data split to which the map task was assigned.

Modified Word Count – Version 2
In-Mapper Combiner

```
class Mapper
    method setup
        H ← new HashMap
    method map (String lineOffset, String line):
        for each word w in line:
            H{w} ← H{w} + 1
    method cleanup
        for all words w in H do
            Emit( w, H{w} );
```

Tally counts for the entire i/p split

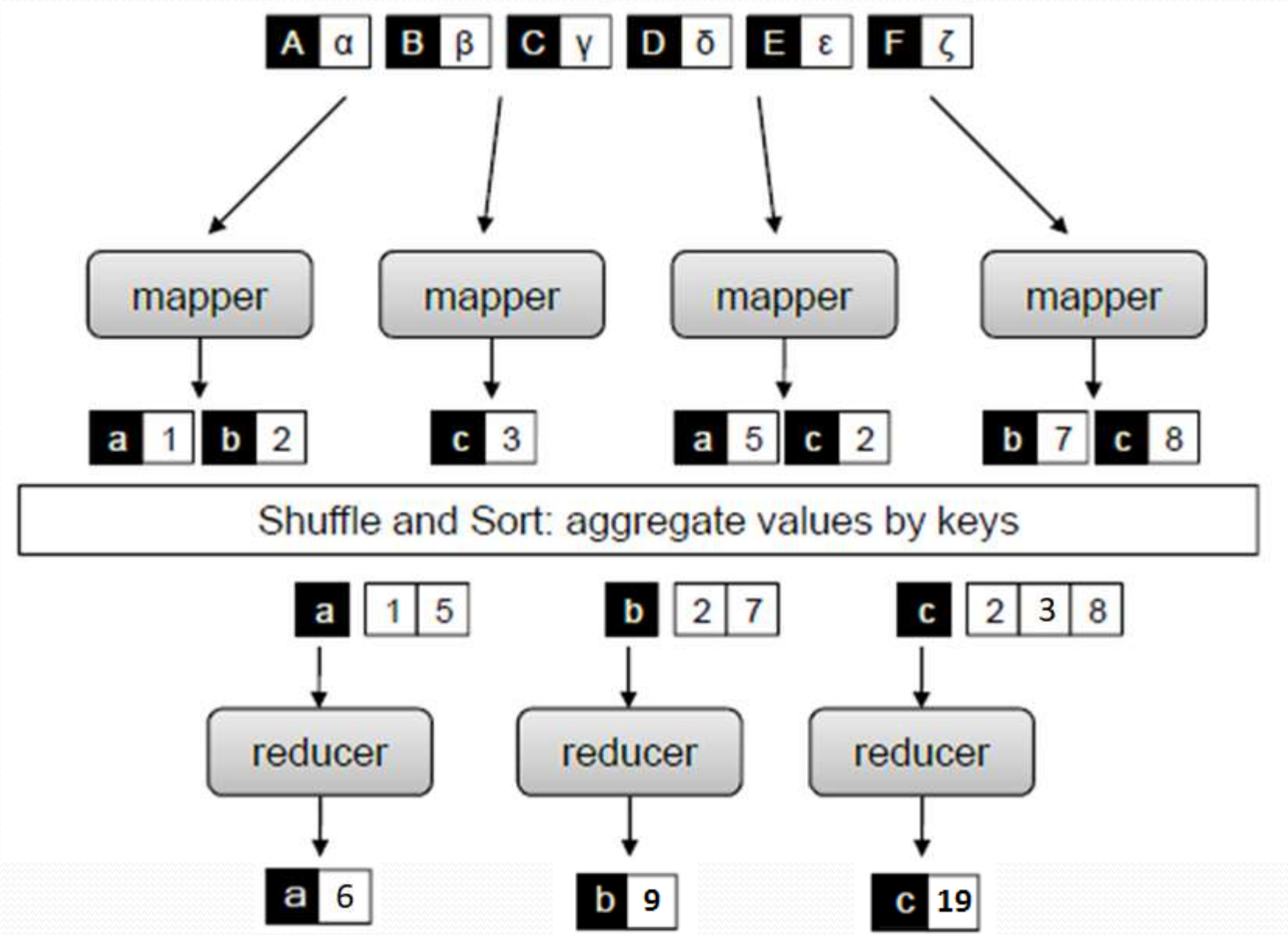
Are combiners still needed?

In-Mapper Combining

- With this technique, we are in essence incorporating combiner functionality directly inside the mapper.
- There is no need to run a separate combiner, since all opportunities for local aggregation are already exploited.
- This is a sufficiently common design pattern in MapReduce called, "**in-mapper combining**".

- "In-mapper combining"
  - Fold the functionality of the combiner into the mapper by preserving state across multiple calls to map method.

In-Mapper Combining



Advantages of the in-mapper combining pattern

- First, it provides control over when local aggregation occurs and how it exactly takes place. In contrast, semantics of default combiners are underspecified in MapReduce.
- Second, in-mapper combining will typically be more efficient than using actual combiners. (speed is more with in-mapper as it is applied inside the map code)

Disadvantages of the in-mapper combining pattern

- States are preserved within mappers
  - potentially large memory overhead
- Algorithmic behavior may depend on the order in which input key-value pairs are encountered.
  - potential for order-dependent bugs (although the correctness of in-mapper combining for word count is easy to demonstrate).

Solution for Memory Usage Problem

- One common solution to limiting memory usage is to "block" input key-value pairs and "flush" in-memory data structures periodically.
- The idea is simple: instead of emitting intermediate data only after every key-value pair has been processed, emit partial results after processing every *n* key-value pairs.
- This is straightforwardly implemented with a counter variable that keeps track of the number of input key-value pairs that have been processed.

Advantages of Local Aggregation

- Reduce network traffic.
- In our word count example, we do not filter frequently-occurring words: therefore, without local aggregation, the reducer that's responsible for computing the count of `the' will have a lot more work to do than the typical reducer, and therefore will likely be a straggler.
- Local aggregation is an effective technique for dealing with reduce stragglers that result from a highly-skewed (e.g., Zipfian) distribution of values associated with intermediate keys.
- With local aggregation we substantially reduce the number of values associated with frequently-occurring terms, which alleviates the reduce straggler problem.

Algorithmic Correctness With Local Aggregation

- In cases where the reduce computation is both commutative and associative, the reducer can also be used (unmodified) as the combiner (as is the case with the word count example).
- It results in temptation to always use combiners without ensuring whether it actually brings the expected benefits or not!
- In the general case, however, combiners and reducers are not interchangeable.
- Example: find average of all integers associated with the same key

Algorithmic Correctness With Local Aggregation

- Example: We have a large dataset where input keys are strings and input values are integers, and we wish to compute the average of all integers associated with the same key (rounded to the nearest integer).
- A real-world example might be a large user log from a popular website, where keys represent user ids and values represent some measure of activity such as elapsed time for a particular session - the task would correspond to computing the average session length on a per-user basis, which would be useful for understanding user demographics.

Computing the Average: Version 1

```
class Mapper
        method map (String t, int r):
                Emit (t, r)
class Reducer
        method reduce (String t, ints [r1, r2,…]):
                sum ← 0
                cnt ← 0
                for all int r in ints[r1,r2,…] do
                        sum ← sum + r
                        cnt ← cnt + 1
                avg ← sum/cnt
                Emit( t, avg );
```

Pseudo-code for the basic MapReduce algorithm that computes the average of values associated with the same key without combiners.

- Any drawback?
- Can we use reducer as combiner?

Computing the Average: Version 1

- We use an identity mapper, which simply passes all input key-value pairs to the reducers (appropriately grouped and sorted).
- The reducer keeps track of the running sum and the number of integers encountered.
- This information is used to compute the average once all values are processed.
- The average is then emitted as the output value in the reducer (with the input string as the key).

Computing the Average: Version 1

- This algorithm will indeed work, but suffers from the same drawbacks as the basic word count algorithm.
- It requires shuffling all key-value pairs from mappers to reducers across the network, which is highly inefficient.
- Unlike in the word count example, the reducer cannot be used as a combiner in this case.
- Consider what would happen if we did: the combiner would compute the average of an arbitrary subset of values associated with the same key, and the reducer would compute the average of those values.

Computing the Average: Version 1

As a concrete example, we know that:

$$\text{Avg}(1; 2; 3; 4; 5) \neq \text{Avg}(\text{Avg}(1; 2);\text{Avg}(3; 4; 5))$$

In general, the average of averages of arbitrary subsets of a set of numbers is not the same as the average of the set of numbers.

Therefore, the approach of using combiner same as reducer would not produce the correct result in this case.

Computing the Average: Version 2

```
class Mapper
        method map (String t, int r):
                Emit (t, r)
class Combiner
        method combine (String t, ints [r1, r2,…]):
                sum ← 0
                cnt ← 0
                for all int r in ints[r1,r2,…] do
                sum ← sum + r
                        cnt ← cnt + 1
                Emit( t, pair (sum, cnt) );
class Reducer
        method reduce (String t, pairs [(s1,c1), (s2,c2),…]):
                sum ← 0
                cnt ← 0
                for all pair (s, c) in pairs [(s1,c1), (s2,c2),…] d
                        sum ← sum + s
                        cnt ← cnt + c
                avg ← sum/cnt
                Emit( t, avg );
```

Does it work?

Why or why not?

Computing the Average: Version 2

- The mapper remains the same.
- Added a combiner that partially aggregates results by computing the numeric components necessary to arrive at the average.
- The combiner receives each string and the associated list of integer values, from which it computes the sum and count.
- The sum and count are packaged into a pair, and emitted as the output of the combiner, with the same string as the key.
- In the reducer, pairs of partial sums and counts can be aggregated to arrive at the average.

Computing the Average: Version 2

- Unfortunately, this algorithm will not work.
- Recall that combiners must have the same input and output key-value types, which also must be the same as the mapper output type and the reducer input type.
- This is clearly not the case.
- To understand why this restriction is necessary in the programming model, remember that combiners are optimizations that cannot change the correctness of the algorithm.
- So if combiner doesn't run at all, then also your program should work and the o/p should remain the same.

Computing the Average: Version 2

- So let us remove the combiner and see what happens: the output value type of the mapper is integer, so the reducer expects to receive a list of integers as values. But the reducer actually expects a list of pairs!
- The correctness of the algorithm is dependent on the combiner running on the output of the mappers, and more specifically, that the combiner is run exactly once.
- Recall from our previous discussion that Hadoop makes no guarantees on how many times combiners are called; it could be zero, one, or multiple times.

Computing the Average: Version 3

```
class Mapper
        method map (String t, int r):
                Emit (t, pair (r, 1))
class Combiner
        method combine (String t, pairs [(s1,c1), (s2,c2),…]):
                sum ← 0
                cnt ← 0
                for all pair (s, c) in pairs [(s1,c1), (s2,c2),…] do
                        sum ← sum + s
                        cnt ← cnt + c
                Emit( t, pair (sum, cnt) );
class Reducer
        method reduce (String t, pairs [(s1,c1), (s2,c2),…]):
                sum ← 0
                cnt ← 0
                for all pair (s, c) in pairs [(s1,c1), (s2,c2),…] do
                        sum ← sum + s
                        cnt ← cnt + c
                avg ← sum/cnt
                Emit( t, avg );
```

Fixed?

Computing the Average: Version 3
The corrected version

- In the mapper we emit as the value a pair consisting of the integer and one - this corresponds to a partial count over one instance.
- The combiner separately aggregates the partial sums and the partial counts (as before), and emits pairs with updated sums and counts.
- The reducer is similar to the combiner, except that the average is computed at the end.
- In essence, this algorithm transforms a non-associative operation (average of numbers) into an associative operation (element-wise sum of a pair of numbers, with an additional division at the very end).

Computing the Average: Version 3 The corrected version

What would happen if no combiners were run?

- With no combiners, the mappers would send pairs (as values) directly to the reducers.
- There would be as many intermediate pairs as there were input key-value pairs, and each of those would consist of an integer and one.
- The reducer would still arrive at the correct sum and count, and hence the average would be correct.

Computing the Average: Version 3 The corrected version

Now add in the combiners.

The algorithm would remain correct, no matter how many times they run, since the combiners merely aggregate partial sums and counts to pass along to the reducers.

- Note that although the output key-value type of the combiner must be the same as the input key-value type of the reducer, the reducer can emit final key-value pairs of a different type.

Computing the Average: With in-mapper combining

- You can design an even more efficient algorithm that exploits the in-mapper combining pattern. Inside the mapper, the partial sums and counts associated with each string are held in memory across input key-value pairs.
- Intermediate key-value pairs are emitted only after the entire input split has been processed; similar to before, the value is a pair consisting of the sum and count.
- The reducer is exactly the same as in version 3.
- Moving partial aggregation from the combiner directly into the mapper is subjected to all the tradeoffs and caveats discussed earlier this section, but in this case the memory footprint of the data structures for holding intermediate data is likely to be modest, making this variant algorithm an attractive option.

Speculative Execution

- Speculative: Something which is not based on facts or investigation.
- Jobs are broken down into tasks and tasks are run in parallel on different DataNodes.
- Only a single slow running job is enough to make the whole job take a much longer time than it would've done otherwise.
- Hadoop doesn't try to diagnose and fix slow-running tasks; instead, it tries to detect when a task is running slower than expected and launches another equivalent task as a backup.
- The scheduler tracks the progress of all tasks of the same type (map and reduce) in a job, and only launches speculative duplicates for the small proportion that are running significantly slower than the average.
- When a task completes successfully, any duplicate tasks that are running are killed since they are no longer needed.
- Speculative execution is enabled by default. You can disable speculative execution for the mappers and reducers by setting the *mapred.map.tasks.speculative.execution* and *mapred.reduce.tasks.speculative.execution* JobConf options to false, respectively.

When NOT to use MapReduce?

- There are few scenarios where MapReduce programming model cannot be employed.
- If the computation of a value depends on previously computed values, then MapReduce cannot be used. One good example is the Fibonacci series where each value is summation of the previous two values. i.e., f(k+2) = f(k+1) + f(k). Also, if the data set is small enough to be computed on a single machine, then it is better to do it as a single reduce(map(data)) operation rather than going through the entire map reduce process.
- MapReduce is not a database system and its main purpose is to process large amounts of unstructured data (e.g., web crawling) and generate meaningful structured data as its output.