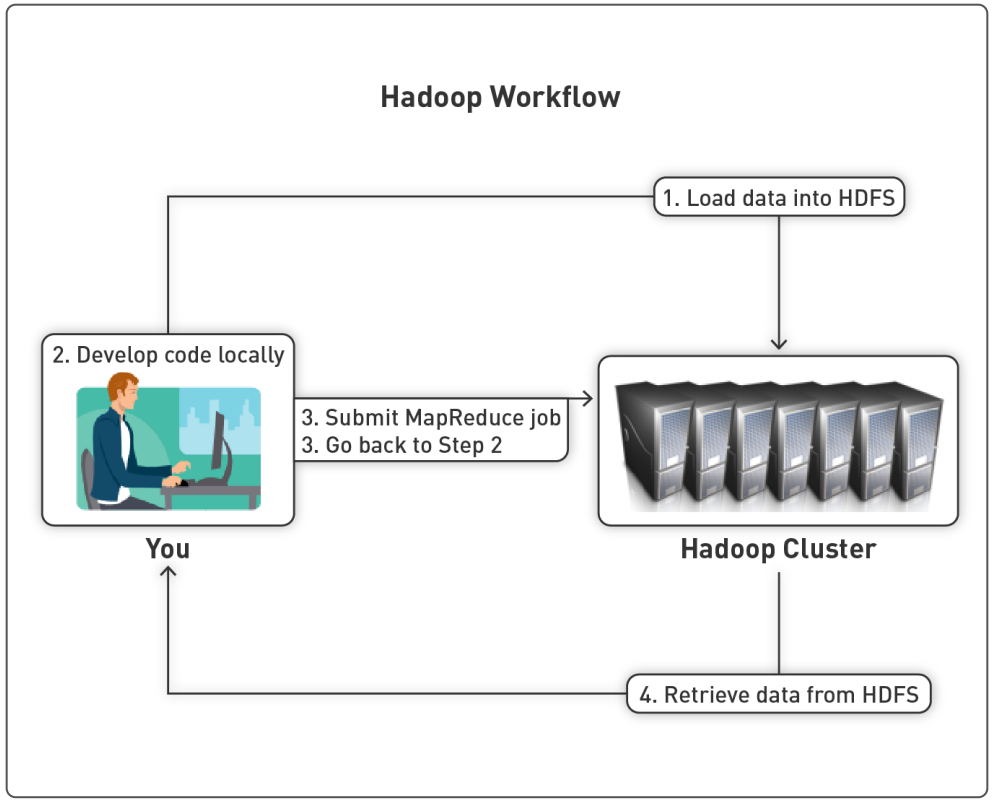
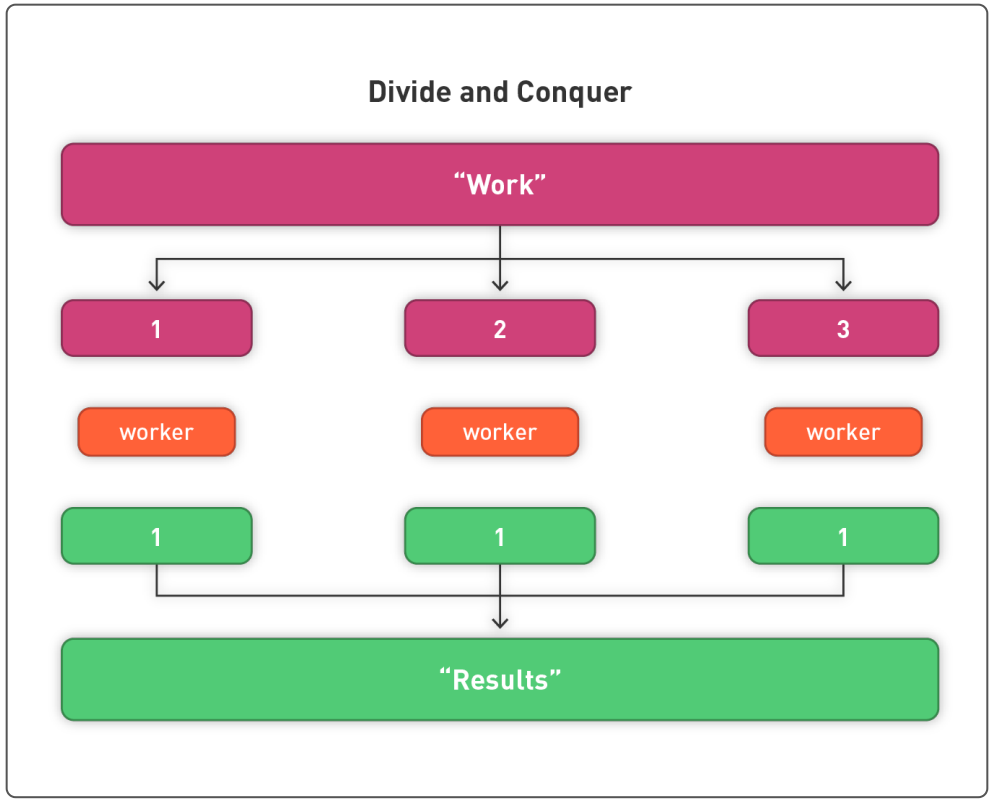

Hadoop Workflow
1. Load data into HDFS
2. Develop code locally
3. Submit MapReduce job
3. Go back to Step 2
You
Hadoop Cluster
4. Retrieve data from HDFS

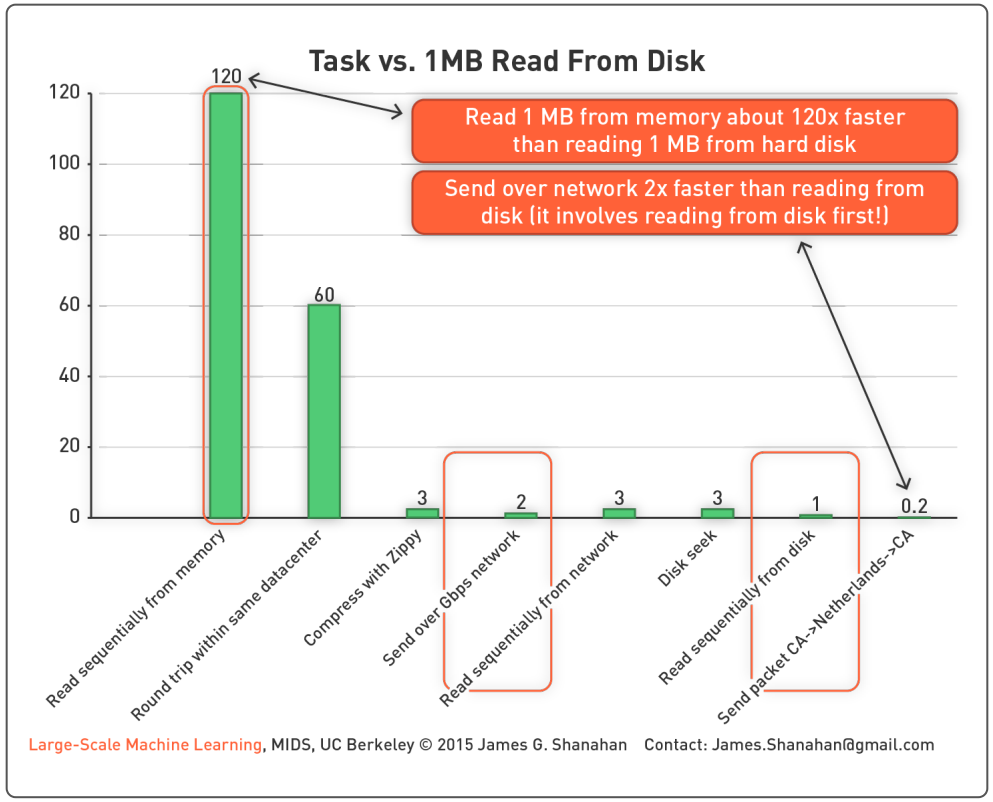

Divide and Conquer
"Work"
1
2
3
worker
worker
worker
1
1
1
"Results"

Synchronization and Coordination

- Modeled after Lisp primitives:
  - *Map* (apply function to all items in a collection)
  - *Reduce* (apply function to set of items with a common key)
- Synchronization and coordination in distributed tasks is key
  - E.g., reducer is a big target of communication and synchronization (barrier between map and reduce)
  - Need to get all data for a particular key to the same location
- How to achieve synchronization in a MapReduce?
  - Keys
  - Sorting and secondary sorting
  - Partitioning
  - Framework or in-memory mappers or in-memory reducers

Communicate Through Files and Streams

- Abstraction: Instead of sending messages, different pieces of code communicate through *files*
  - Code is going to take a very "stylized" form; at each stage each machine will get input from files, send output to files
  - Files are generally persistent, nameable (in contrast to distributed hash table messages, which are transient)
  - Files consist of blocks, which are the basic unit of partitioning (in contrast to object or data item IDs)
  - Server is **stateless**, so clients must send all context (including position to read from) in each request


Task vs. 1MB Read From Disk
120
100
80
60
40
20
0
120
60
3
2
3
3
1
0.2
Read 1 MB from memory about 120x faster than reading 1 MB from hard disk
Send over network 2x faster than reading from disk (it involves reading from disk first!)
Read sequentially from memory
Round trip within same datacenter
Compress with Zippy
Send over Gbps network
Read sequentially from network
Disk seek
Read sequentially from disk
Send packet CA->Netherlands->CA
Large-Scale Machine Learning, MIDS, UC Berkeley © 2015 James G. Shanahan Contact: James.Shanahan@gmail.com

Priority Queue: Key-Value Data Structure

Key/ Priority	Value
1	$node_B$
2	$node_A$
5	$node_N$

→ Insert (4, $node_x$)

Key/ Priority	Value
1	$node_1$
2	$node_2$
4	$node_x$
5	$node_3$

→ Remove()

Key/ Priority	Value
1	$node_1$

Main methods of the priority queue abstract data type:

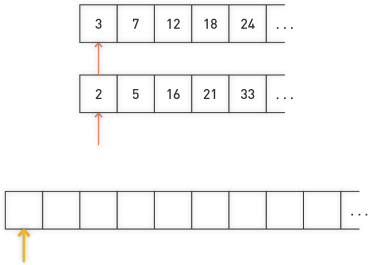
insert(k, x)
Inserts an entry with key k and value x

removeMin()
Removes and returns the entry with smallest key

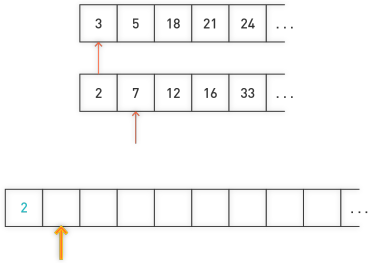
- Maintained priority queue always sorted
- A priority queue stores a collection of entries, where each entry is a pair (key, value), where the key is priority; these entries are sorted in order of decreasing priority

Merge Two Sorted Lists or Arrays

How do we **merge** two sorted subarrays? We define three references at the front of each array.



We keep picking the smallest element and move it to a temporary array, incrementing the corresponding indices.



Merge Two Sorted Lists or Arrays (cont.)

```
def mergeSortedLists(a, b):
    1 = []
    while a and b:
        if a[0] < b[0]:
            1.append(a.pop(0))
        else:
            1.append(b.pop(0))
    return 1 + a +b
```

Merge Algorithms: O(*m* log *n*) Time

- Given *n* sorted lists, let *m* = Sum of the lengths of the lists
- Merge algorithms generally run in time O(*m* log *n*)
- Merge can use a **heap**-based **priority queue**
    - Merge algorithms that operate on large numbers of lists at once will multiply the sum of the lengths of the lists by the time to figure out which of the pointers points to the lowest item, which can be accomplished with a **heap**-based **priority queue** in **O**(log *n*) time, for O(*m* log *n*) time, where *n* is the number of lists being merged and *m* is the sum of the lengths of the lists. When merging two lists of length *m*, there is a lower bound of 2*m* – 1 comparisons required in the worst case.

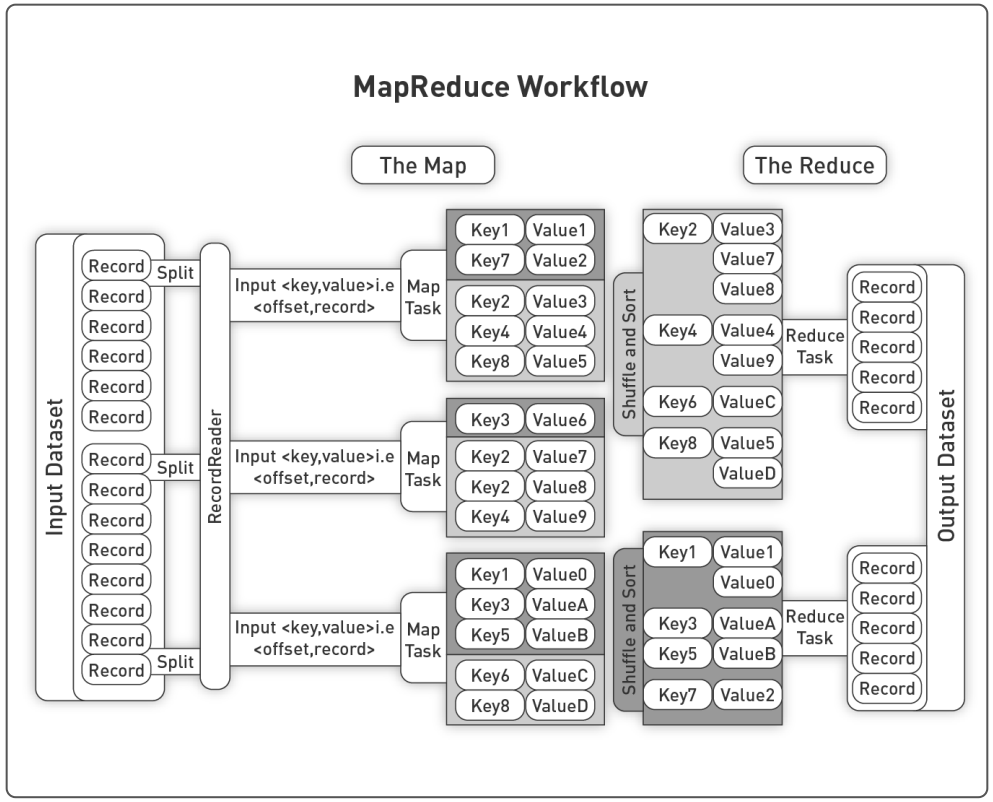
MapReduce: Recap

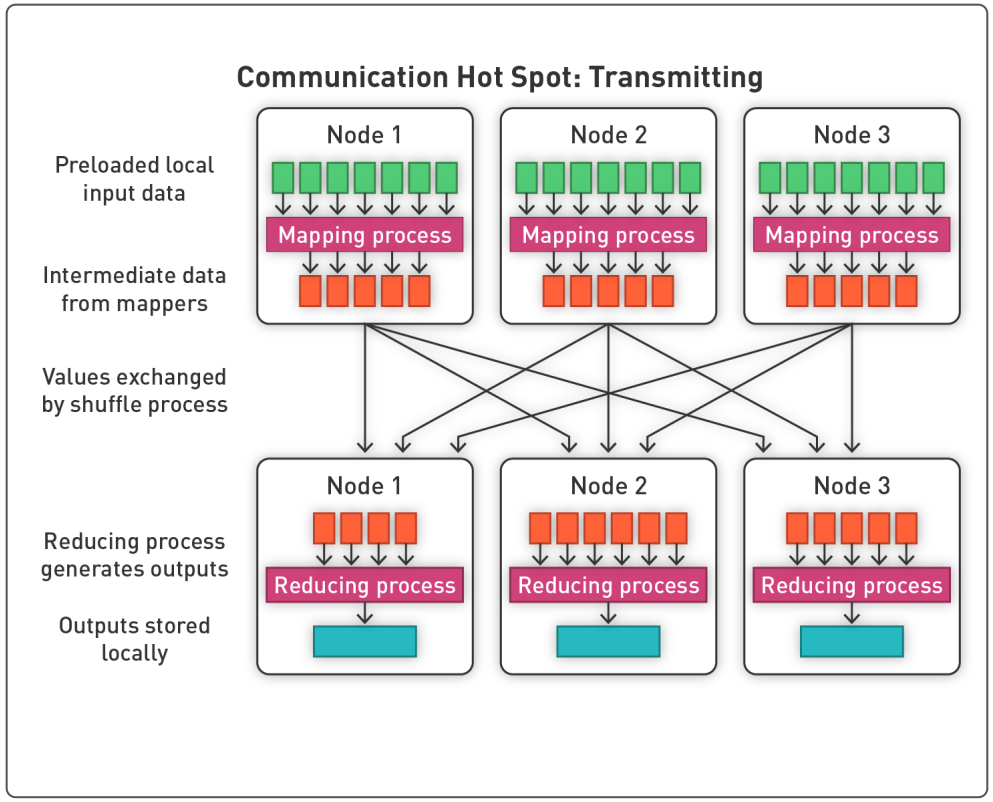
- Programmers must specify:
  - **Map** (k, v)→<k', v'>*
  - **Reduce**(k', v') → <k', v'>*
    - All values with the same key are reduced together
- Optionally also:
  - **Partition**(k', number of partitions) → partition for k'
    - Often a simple hash of the key, e.g., hash(k') mod n
    - Divides 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

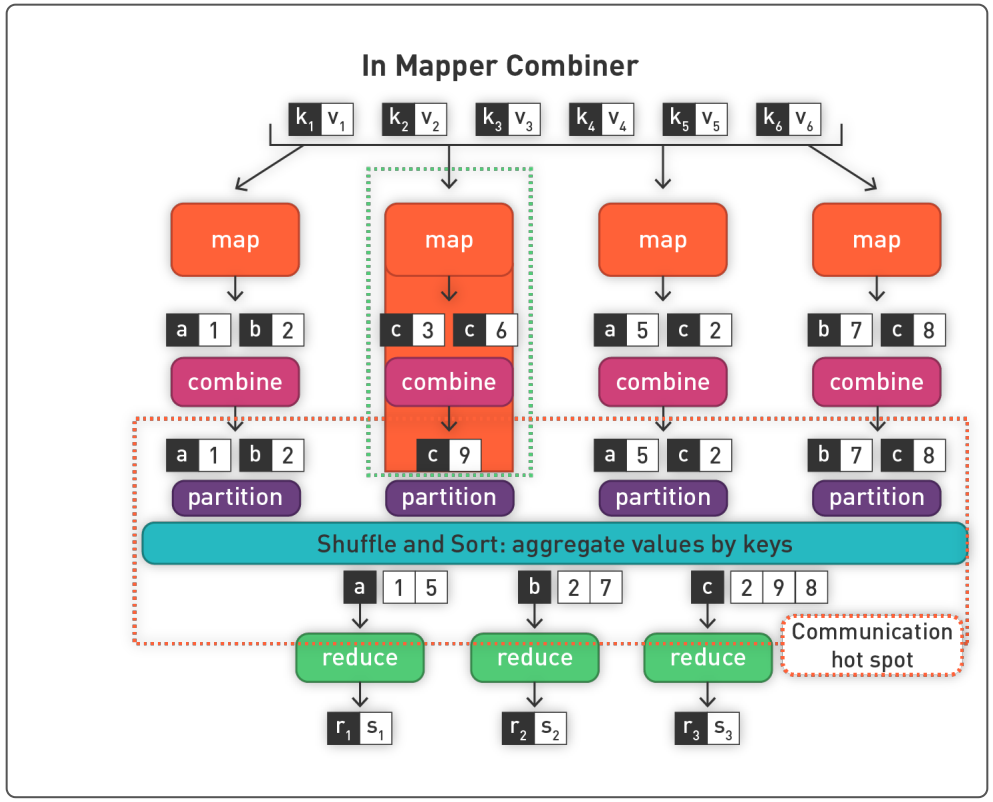
The execution framework handles everything else …

"Everything Else"

- The execution framework handles everything else …
  - Scheduling: Assigns workers to map and reduce tasks
  - "Data distribution": Moves processes to data
  - Synchronization: Gathers, sorts, and shuffles intermediate data
  - Errors and faults: Detects worker failures and restarts
- Limited control over data and execution flow
  - All algorithms must be expressed in m, r, c, p
- You don't know:
  - Where mappers and reducers run
  - When a mapper or reducer begins or finishes
  - Which input a particular mapper is processing
  - Which intermediate key a particular reducer is processing

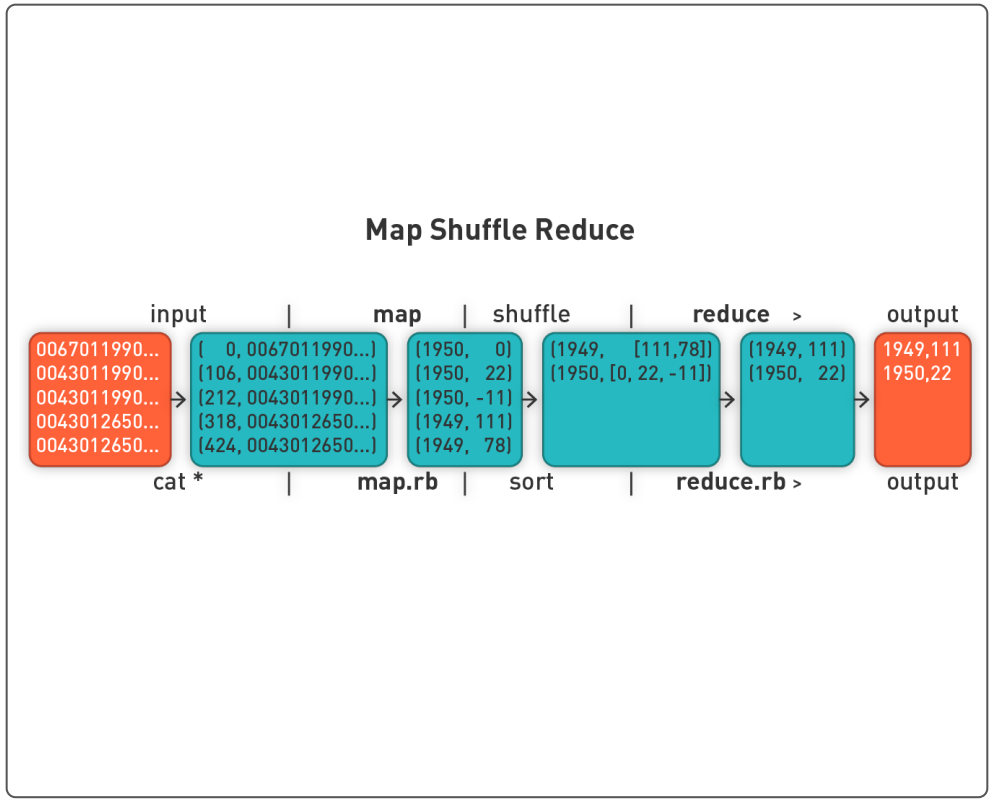

MapReduce Workflow
The Map
The Reduce
Input Dataset
Record
Split
RecordReader
Input <key,value>i.e <offset,record>
Map Task
Key1 Value1
Key7 Value2
Key2 Value3
Key4 Value4
Key8 Value5
Key3 Value6
Key2 Value7
Key2 Value8
Key4 Value9
Key1 Value0
Key3 ValueA
Key5 ValueB
Key6 ValueC
Key8 ValueD
Shuffle and Sort
Key2 Value3
Value7
Value8
Key4 Value4
Value9
Key6 ValueC
Key8 Value5
ValueD
Key1 Value1
Value0
Key3 ValueA
Key5 ValueB
Key7 Value2
Reduce Task
Output Dataset

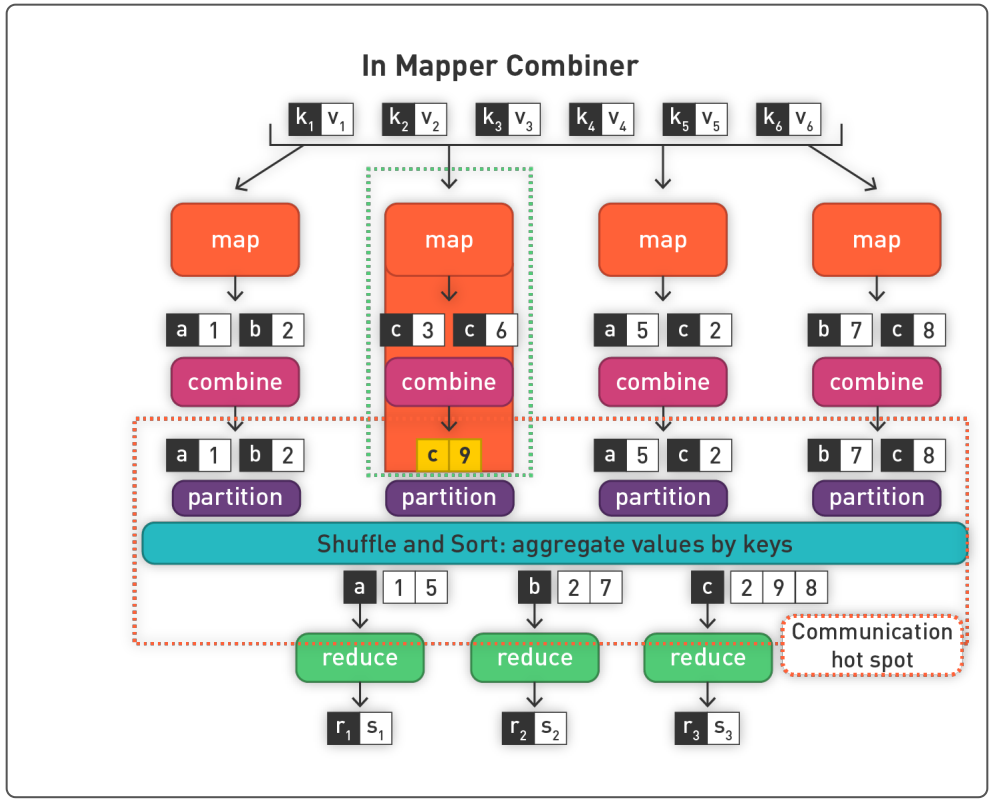

Communication Hot Spot: Transmitting
Preloaded local input data
Intermediate data from mappers
Values exchanged by shuffle process
Reducing process generates outputs
Outputs stored locally
Node 1
Node 2
Node 3
Mapping process
Mapping process
Mapping process
Node 1
Node 2
Node 3
Reducing process
Reducing process
Reducing process


In Mapper Combiner
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 8
reduce
reduce
reduce
Communication hot spot
r1 s1
r2 s2
r3 s3

Shuffle: Heart of MapReduce

- MapReduce makes the guarantee that the input to every reducer is sorted by key.
- The process by which the system performs the sort—and transfers the map outputs to the reducers as inputs—is known as the shuffle.
    - **Partition, sort, combine (in memory, on disk)**
- In this section, we look at how the shuffle works, as a basic understanding would be helpful should you need to optimize a MapReduce program.
- In many ways, the shuffle is the heart of MapReduce and is where the "magic" happens.


Map Shuffle Reduce
input
map
shuffle
reduce
output
0067011990...
0043011990...
0043011990...
0043012650...
0043012650...
(0, 0067011990...)
(106, 0043011990...)
(212, 0043011990...)
(318, 0043012650...)
(424, 0043012650...)
(1950, 0)
(1950, 22)
(1950, -11)
(1949, 111)
(1949, 78)
(1949, [111,78])
(1950, [0, 22, -11])
(1949, 111)
(1950, 22)
1949,111
1950,22
cat *
map.rb
sort
reduce.rb
output


In Mapper Combiner
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 8
reduce
reduce
reduce
Communication hot spot
r1 s1
r2 s2
r3 s3

Shuffle: Partition, Sort, Combine

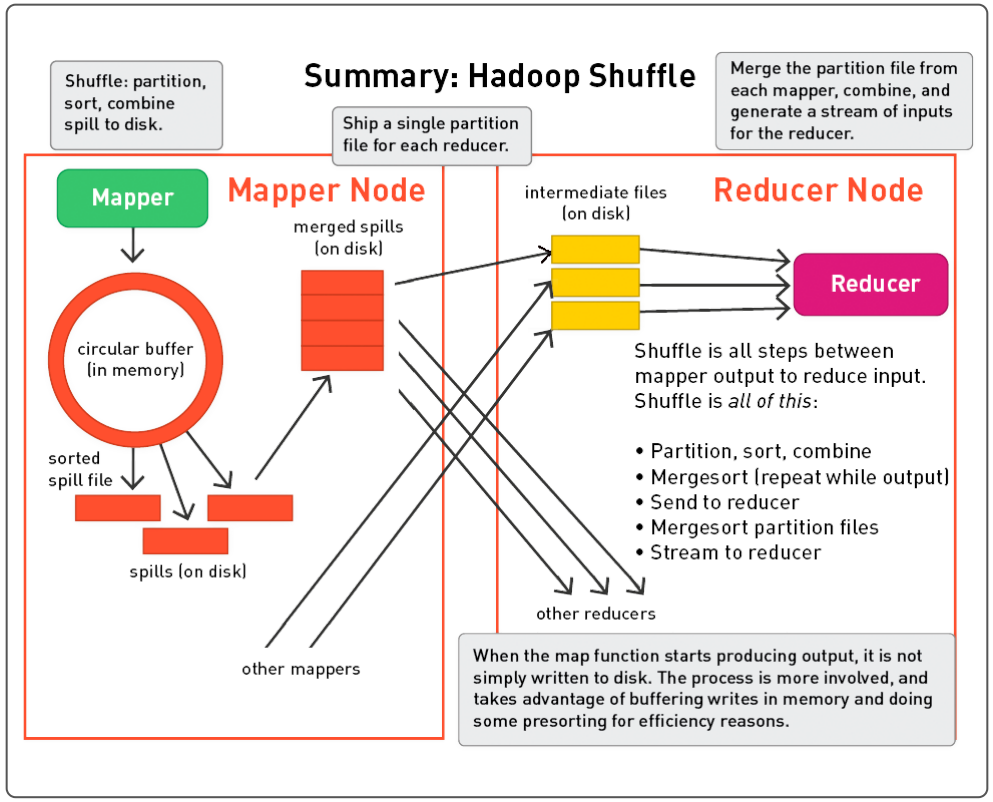
- In memory
  - Partition
  - Sort records within each partition
  - Merge sorted records
  - Combine

Shuffle in Memory: Shuffle on Disk

- Probably the most complex aspect of MapReduce!
- Map side
  - Map outputs are buffered in memory in a circular buffer (in-memory shuffle, i.e., partition, sort, combine)
  - When buffer reaches threshold, sorted contents are "spilled" to disk
  - Spills are merged in a single, partitioned file (sorted within each partition); combiner runs on the sorted partition file
- Reduce side
  - First, map outputs are copied over to reducer machine
  - "Sort" is a multipass merge (merge-sort) of map outputs (happens in memory and on disk); combiner runs here
  - Final merge pass goes directly into reducer

Mapper Memory Buffer Gets Full: Spill to Disk

- When the contents of the buffer reach a certain threshold size (mapreduce.map.sort.spill.percent, which has the default 0.80, or 80%), a background thread will start to spill the contents to disk.
- Map outputs will continue to be written to the buffer while the spill takes place, but if the buffer fills up during this time, the map will block until the spill is complete.


Summary: Hadoop Shuffle
Shuffle: partition, sort, combine spill to disk.
Ship a single partition file for each reducer.
Merge the partition file from each mapper, combine, and generate a stream of inputs for the reducer.
Mapper
Mapper Node
merged spills (on disk)
circular buffer (in memory)
sorted spill file
spills (on disk)
other mappers
intermediate files (on disk)
Reducer Node
Reducer
Shuffle is all steps between mapper output to reduce input.
Shuffle is all of this:
• Partition, sort, combine
• Mergesort (repeat while output)
• Send to reducer
• Mergesort partition files
• Stream to reducer
other reducers
When the map function starts producing output, it is not simply written to disk. The process is more involved, and takes advantage of buffering writes in memory and doing some presorting for efficiency reasons.

Divide and Conquer

Divide and conquer comes with massive gains and some responsibilities

- Ideal scaling characteristics (linear scaling):
  - Twice the data, twice the running time
  - Twice the resources, half the running time
- Why can't we achieve this?
  - Communication kills performance
  - Synchronization requires communication
- Thus … avoid communication?
  - Reduce intermediate data via local aggregation
  - Combiners, in-memory mappers, and reducers can help

Individual Mappers (and Reducers) Run in Isolation

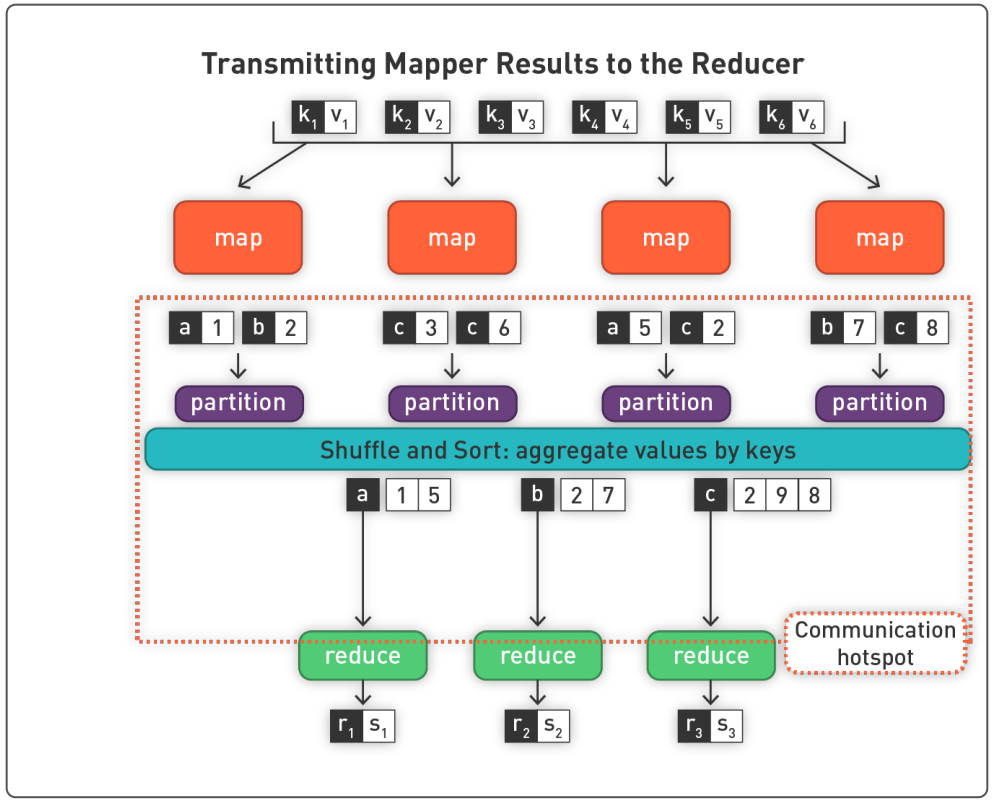
- All mappers and reducers have a common goal.
- To solve a job, they need to work as a team.
  - Teamwork requires communication and synchronization.
  - Mappers and reducers run in isolation without any mechanisms for direct communication.
  - Synchronization or merging of results can be the most tricky aspect of designing MapReduce algorithms (or, for that matter, parallel and distributed algorithms in general).
- Within a single MapReduce job, there is only one opportunity for cluster-wide synchronization—during the shuffle and sort stage where intermediate key-value pairs are copied from the mappers to the reducers and grouped by key.

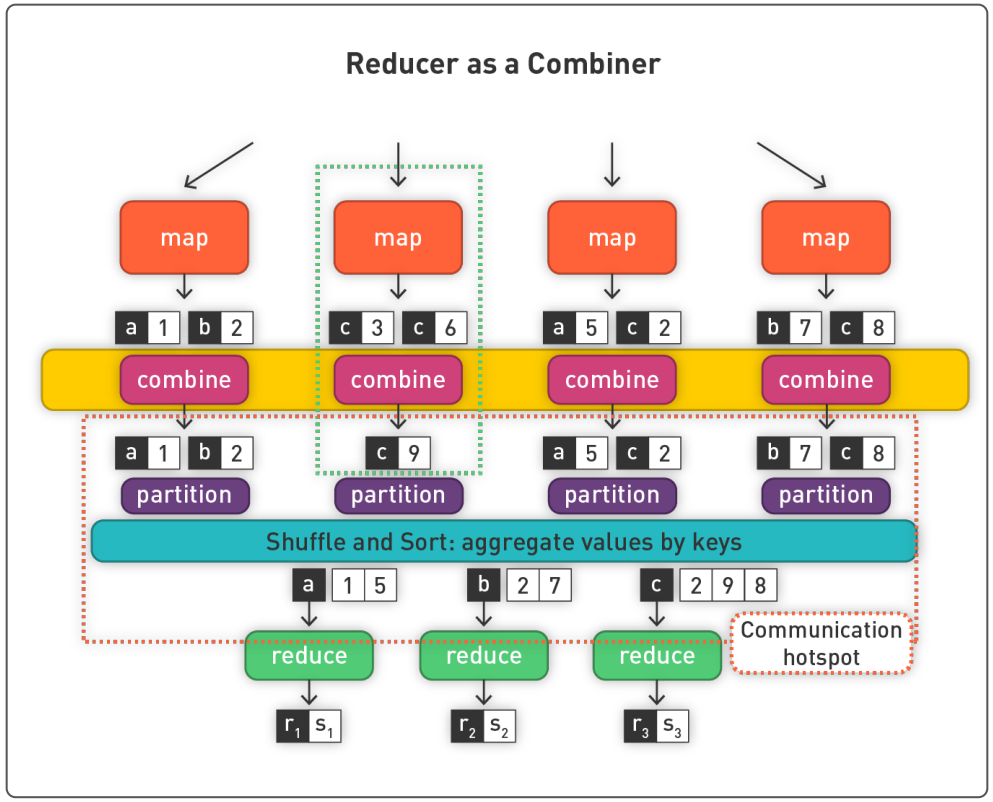
Local Aggregation

- **Transfer intermediate results: Producer→Consumer**
  - In the context of data-intensive distributed processing, the single most important aspect of synchronization is the exchange of intermediate results from the processes that produced them to the processes that will ultimately consume them.

Tools for Synchronization

- Cleverly constructed data structures
  - Bring partial results together
- Sort order of intermediate keys
  - Control order in which reducers process keys
- Partitioner
  - Control which reducer processes which keys
- Preserving state in mappers and reducers
  - Capture dependencies across multiple keys and values


Transmitting Mapper Results to the Reducer
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
partition
partition
partition
partition
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 9 8
reduce
reduce
reduce
Communication hotspot
r1 s1
r2 s2
r3 s3


Reducer as a Combiner
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 8
reduce
reduce
reduce
Communication hotspot
r1 s1
r2 s2
r3 s3

Word Count: Baseline

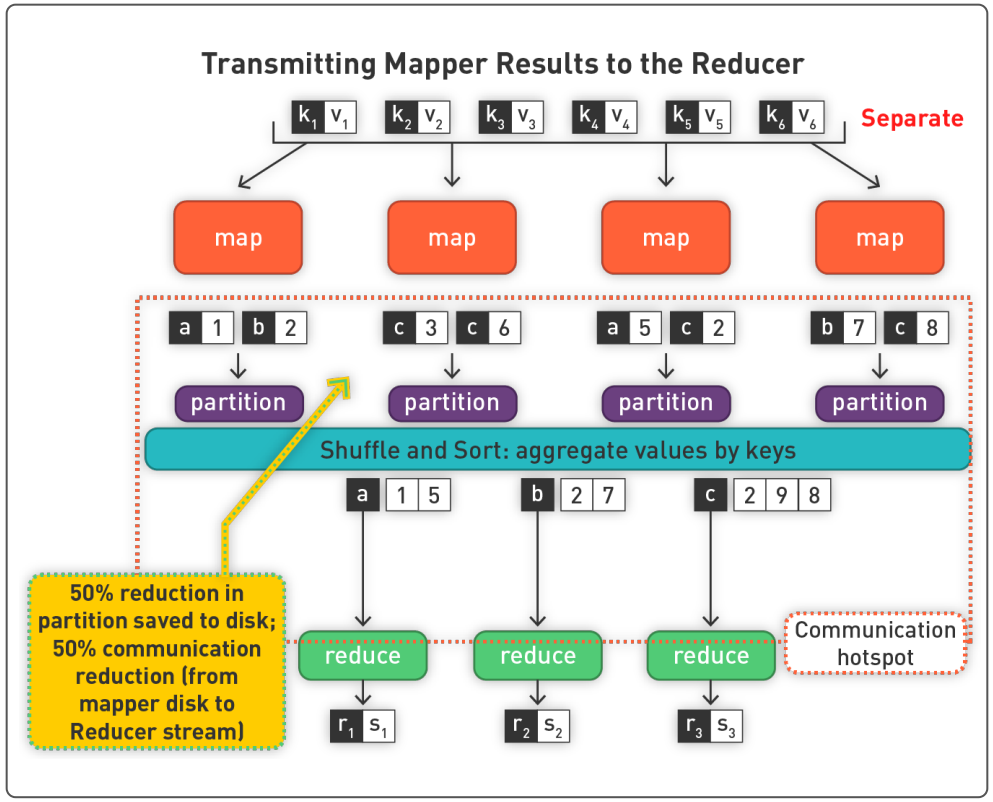
```
1:    class MAPPER
2:        method MAP(docid a, doc d)
3:            for all term t ∈ doc d do
4:                EMIT(term t, count 1)

1:    class REDUCER
2:        method REDUCE(term t, counts [c1,c2, ...])
3:            sum←0
4:            for all count c ∈ counts [c1,c2, ...] do
5:                sum←sum + c
6:            EMIT(term t, count s)
```

PIAT

1. In mem: Partition, sort ~~combine~~
2. Spill to disk
3. Merge sorted spills, ~~combine~~
4. Once mapper finishes:
    - Transfer to reducer bandwidth

Emit each word ... heavy disk storage requirements on mapper side; big sorts; big transfer from mapper to reducer

What's the impact of combiners?


Transmitting Mapper Results to the Reducer
Separate
map
map
map
map
partition
partition
partition
partition
Shuffle and Sort: aggregate values by keys
50% reduction in partition saved to disk; 50% communication reduction (from mapper disk to Reducer stream)
reduce
reduce
reduce
Communication hotspot

Other Forms of Local Optimization

- Data have gravity
- They are heavy to move around
- In-memory mappers and in-memory reducers?

Mappers Are Java Objects With Init and State

- In Hadoop, mappers are Java objects with a map method (among others).
- A mapper object is instantiated for every map task by the task tracker.
- The life cycle of this object begins with instantiation, where a hook is provided in the API to run programmer-specified code.
    - This means that mappers can read in "side data," providing an opportunity to load state, static data sources, dictionaries, etc.
    - After initialization, the map method is called (by the execution framework) on all key-value pairs in the input split.
    - Since these method calls occur in the context of the same Java object, it is possible to preserve state across multiple input key-value pairs within the same map task. This is an important property to exploit in the design of MapReduce algorithms, as we will see in the next chapter.
- After all key-value pairs in the input split have been processed, the mapper object provides an opportunity to run programmer-specified termination code. This, too, will be important in the design of MapReduce algorithms.
- The actual execution of reducers is similar to that of the mappers.

Word Count, Version 1: Record-Level Combiner

Within each document combiner

```
1:    class MAPPER
2:        method MAP(docid a, doc d)
3:            H←new ASSOCIATIVEARRAY
4:            for all term t ∈ doc d do
5:                H { t }←H{ t } + 1   →Tally counts for entire document
6:            for all term t ∈ H do
7:                EMIT(term t, count H { t })
```

Issues

1. Disk
2. CUP
3. Network

Are combiners still needed?

Scales very well but still have the same issues as the baseline mapper-reducer solution.
↓
Yes. There are some savings here, but there are still a lot of instances of the same word in the stream that can be combined.

Word Count, Version 2: In-Memory Mapper Combiner

```
1:    class MAPPER
2:        method INITIALIZE
3:            H←new ASSOCIATIVEARRAY
4:    method MAP(docid a, doc d)
5:            for all term t ∈ H do
6:                H { t }←H { t } + 1   →Tally counts across documents
7:    method CLOSE
8:            for all term t ∈ H do
9:                EMIT(term t, count H { t })
```

Resolves

1. Disk
2. CUP
3. Network

- BUT
  - Memory issues?

Are combiners still needed?

Yes. On the reduce side, combiners can be run to consolidate results before putting them into the REDUCE stream.

Design Pattern for Local Aggregation

- "In-mapper combining"
  - Fold the functionality of the combiner into the mapper by preserving state across multiple map calls
- Advantages
  - Speed
  - Why is this faster than actual combiners?
- Disadvantages
  - Explicit memory management required
  - Potential for order-dependent bugs

Associative (Grouping) Property

- Within an expression containing two or more occurrences in a row of the same associative operator, the order in which the operations are performed does not matter as long as the sequence of the operands is not changed.
- That is, rearranging the parentheses in such an expression will not change its value.

$$(2 + 3) + 4 = 2 + (3+4) = 9$$

$$2 \times (3 \times 4) = (2 \times 3) \times 4 = 24$$

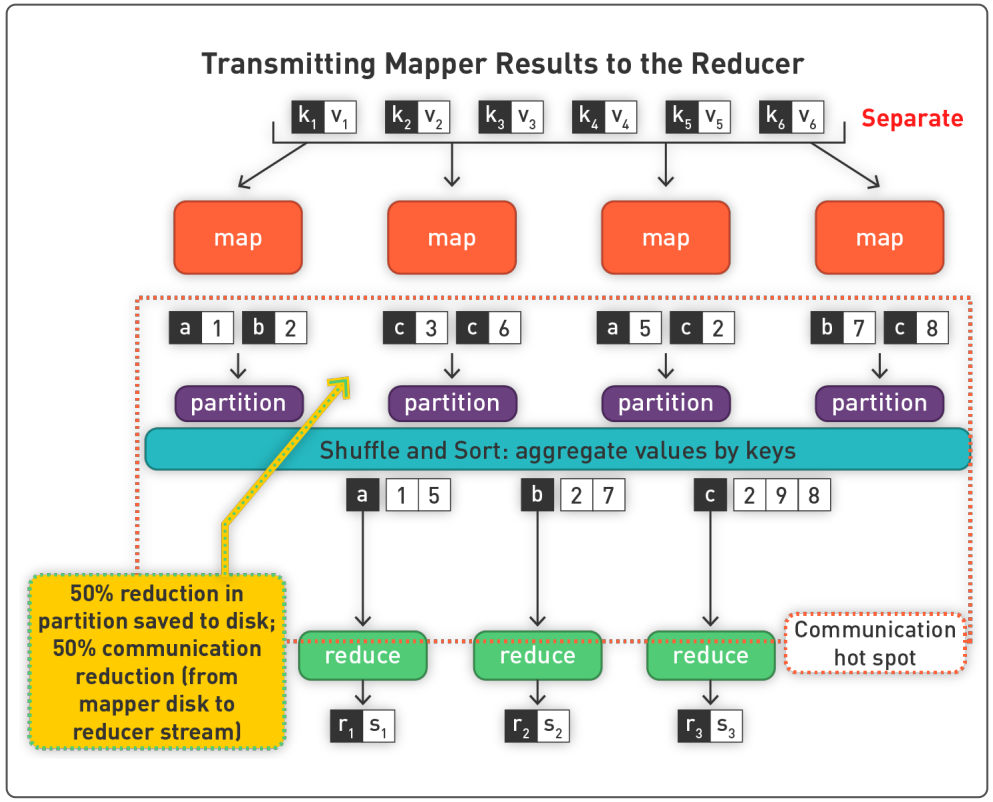
Commutative Property

In mathematics, a binary operation is commutative if changing the order of the operands does not change the result. The term *commutative* is used in several related senses.

1. A binary operation (×) on a set *S* is called *commutative*if:
    - x × y = y × x (for all x, y ∈ *S*)
      An operation that does not satisfy the above property is called *noncommutative*.
2. One says that *x commutes* with *y*under × if:
    - x × y = y × x

Combiner Design

- Combiners and reducers share the same method signature.
    - Sometimes, reducers can serve as combiners
    - Often, not ...
- Remember: Combiners are optional optimizations
    - Should not affect algorithm correctness
    - Hadoop makes no guarantees; could be run 0, 1, or multiple times
- Combiners need to be associative and commutative
- Example: Find average of all integers associated with the same key


Transmitting Mapper Results to the Reducer
Separate
map
map
map
map
partition
partition
partition
partition
Shuffle and Sort: aggregate values by keys
reduce
reduce
reduce
50% reduction in partition saved to disk; 50% communication reduction (from mapper disk to reducer stream)
Communication hot spot

Simple Problem!

- We have a large dataset where input keys are strings and input values are numeric, and we wish to compute the mean of all numerics associated with the same key.
- Customer log file (e.g., purchase transactions)
    - 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 mean session length on a per-user basis, which would be useful for understanding user demographics.

Log File of Customer Transactions

Compute the average transaction amount per customer.

SELECT UserID, Avg (Cash Spent)
FROM TransactionLogDB
GROUP BY UserID

	Log File		**Log File**
Key	Value	Value	Value
TransactionID	UserID	Date:Time	Cash Spent
2323323232	20	2/1/2015:11:05	$20.00
2323323233	44444	2/9/2015:11:06	$5.00
2323323234	20	2/9/2015:11:07	$2.00
2323323235	4444	2/9/2015:11:08	$20.00
2323323236	33333	2/10/2015:11:09	$1.00
2323323237	20	2/12/2015:11:10	$5.00
2323323238	20	2/20/2015:11:11	$10.00
2323323239	...		

Computing the Mean: Version 1, No Combiner

```
1:    class MAPPER
2:        method MAP(string t, integer r)
3:        EMIT(string t, integer r)

1:    class REDUCER
2:        method REDUCE(string t, integers [r_1, r_2, ... ])
3:            sum ← 0
4:            cnt ← 0
5:            for all integer r ∈ integers [r_1, r_2, ... ] do
6:                sum ← sum + r
7:                cnt ← cnt + 1
8:            r_avg ← sum/cnt
9:            EMIT(string t, integer r_avg)
```

Log	File
KEY	Value
USERID	**Cash Spent**
44442	$20.00
44444	$5.00
44445	$2.00
44439	$20.00
44445	$1.00
44445	$5.00
44440	$10.00
...	...
...	

- Why can't we use the reducer as a combiner?
- Mean is not an associative operation:
  - MEAN(1, 2, 3, 4, 5) ≠ MEAN(MEAN(1, 2), MEAN(3,4,5))

Challenge

- How can we transform a nonassociative operation (mean of numbers) into an associative operation?
- (Element-wise, sum of a pair of numbers, and a count, with an additional division at the very end)

Computing the Mean: Version 2, Combiner

```
1:   class MAPPER
2:       method MAP(string t, integer r)
3:           EMIT(string t, integer r)
1:   class COMBINER
2:       method COMBINE(string t, integers [r_1, r_2, … ])
3:           sum ← 0
4:           cnt ← 0
5:           for all integer r ∈ integers [r_1, r_2, … ] do
6:               sum ← sum + r
7:               cnt ← cnt + 1
8:       EMIT(string t, pair (sum, cnt))
1:   class REDUCER
2:       method REDUCE(string t, pair [(s_1, c_1), (s_2, c_2) … ])
3:           sum ← 0
4:           cnt ← 0
5:           for all pair (s, c) ∈ pairs [(s_1, c_1), (s_2, c_2) … ] do
6:               sum ← sum + s
7:               cnt ← cnt + c
8:           r_avg ← sum/cnt
9:           EMIT(string t, integer r_avg)
```

Why doesn't this work?

Log	File
KEY	Value
USERID	**Cash Spent**
44442	$20.00
44444	$5.00
44445	$2.00
44439	$20.00
44445	$1.00
44445	$5.00
44440	$10.00
…	…
…	

Element-Wise Sum of Pair of Numbers With Additional Division at End

Let us verify the correctness of this algorithm by repeating the previous exercise:
What would happen if no combiners were run? With no combiners, the mappers would send a single value directly to the reducers (cash spent). There would be as many intermediate values as there were input key-value pairs, and each of those would consist of a single value.

The reducer is expecting a subtotal and count.
Mismatch.

Computing the Mean: Version 3

```
1:   class MAPPER
2:       method MAP(string t, integer r)
3:           EMIT(string t, pair (r, 1))
1:   class COMBINER
2:       method COMBINE(string t, pairs [(s_1, c_1), (s_2, c_2) … ])
3:           sum ← 0
4:           cnt ← 0
5:           for all pair (r, c ∈ pairs [(s_1, c_1), (s_2, c_2) … ]) do
6:               sum ← sum + s
7:               cnt ← cnt + c
8:       EMIT(string t, pair (sum, cnt))
1:   class REDUCER
2:       method REDUCE(string t, pairs [(s_1, c_1), (s_2, c_2) … ])
3:           sum ← 0
4:           cnt ← 0
5:           for all pair (s, c) ∈ pairs [(s_1, c_1), (s_2, c_2) … ] do
6:               sum ← sum + s
7:               cnt ← cnt + c
8:           r_avg ← sum/cnt
9:           EMIT(string t, pair (r_avg, cnt))
```

Fixed?

Element-Wise Sum of Pair of Numbers With Additional Division at End (cont.)

Let us verify the correctness of this algorithm by repeating the previous exercise:

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 1.

The reducer would still arrive at the correct sum and count, and hence the mean would be correct. 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.

Computing the Mean: Version 3

```
1:  class MAPPER
2:      method MAP(string t, integer r)
3:          EMIT(string t, pair (r, 1))
1:  class COMBINER
2:      method COMBINE(string t, pairs [(s_1, c_1), (s_2, c_2) … ])
3:          sum ← 0
4:          cnt ← 0
5:          for all pair (r, c ∈ pairs [(s_1, c_1), (s_2, c_2) … ]) do
6:              sum ← sum + s
7:              cnt ← cnt + c
8:      EMIT(string t, pair (sum, cnt))
1:  class REDUCER
2:      method REDUCE(string t, pairs [(s_1, c_1), (s_2, c_2) … ])
3:          sum ← 0
4:          cnt ← 0
5:          for all pair (s, c) ∈ pairs [(s_1, c_1), (s_2, c_2) … ] do
6:              sum ← sum + s
7:              cnt ← cnt + c
8:          r_avg ← sum/cnt
9:          EMIT(string t, pair (r_avg, cnt))
```

Fixed?

1. Combiners and reducers share the same method signature.
2. Remember: Hadoop makes no guarantees; could be run zero, one, or multiple times.
3. Combiners need to be associative and commutative.

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.

How Many Maps and Reduces?

- Picking the appropriate size for the tasks for your job can radically change the performance of Hadoop.
- Increasing the number of tasks increases the framework overhead but increases load balancing and lowers the cost of failures.
    - At one extreme is the one map/one reduce case, where nothing is distributed. The other extreme is to have 1,000,000 maps/1,000,000 reduces, where the framework runs out of resources for the overhead.
- The number of reduce tasks can also be increased in the same way as the map tasks, via JobConf's conf.setNumMapTasks(int num).

How Many Maps and Reduces? (cont.)

- The number of maps is usually driven by the number of DFS blocks in the input files.
- Although that causes people to adjust their DFS block size to adjust the number of maps.
- The right level of parallelism for maps seems to be around 10–100 maps/node, although we have taken it up to 300 or so for very CPU-light map tasks. Task setup takes a while, so it is best if the maps take at least a minute to execute.
- E.g., if you expect 10TB of input data and have 128MB DFS blocks, you'll end up with 82k maps, unless your mapred.map.tasks is even larger.
- 128MB block size for 10TB job → 82k maps 10–100 maps per node; each map takes one minute

Back-of-the-Envelope Estimates: 10TB

- 128MB block size for 10TB job → 82k maps (10^5)
- Recommended to have 10–100 maps per node
- $\frac{10^5 \text{tasks}}{10^2} = 1{,}000$ nodes
- Each map takes 1 minute, so about 100 minutes to run the mappers

Number of Reducers

- The ideal reducers should be the optimal value that gets them closest to:
  - A multiple of the number of blocks
  - A task time between 5 and 15 minutes
  - Creating the fewest files possible
- The number of reduce tasks can also be increased in the same way as the map tasks, via JobConf's conf.setNumReduceTasks(int num).

How Many Maps and Reduces?

- Empirical question best solved by taking this guidance and experimenting with some small jobs, and establishing the length of map and reduce tasks at different block sizes, etc.
- Recommended block sizes of 128MB–1,000MB (1GB)

Computing the Mean: Version 4, In-Memory Mapper

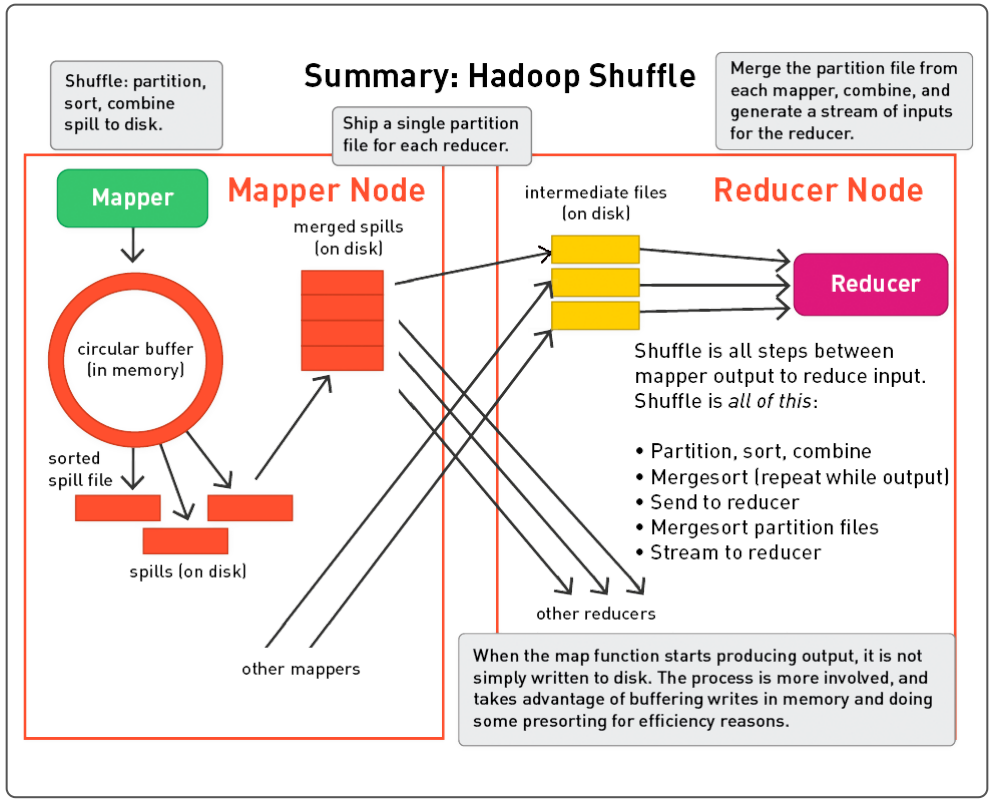
```
1:    class MAPPER
2:        method INITIALIZE
3:            S ← new ASSOCIATIVEARRAY
4:            C ← new ASSOCIATIVEARRAY
5:        method MAP(string t, integer r)
6:            S{ t } ← S{ t } + r
7:            C{ t } ← C{ t } + 1
8:        method CLOSE
9:            for all term t ∈ S do
10:               EMIT(term t, pair (S{ t }, C{t}))
```

Super-efficient: no shuffling!

- 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 above.
- Are combiners still needed?

First Part of This Lecture

- RAM vs. disk vs. bandwidth
- Priority queues and merge sort
- The internals of the Hadoop Shuffle


Summary: Hadoop Shuffle
Shuffle: partition, sort, combine spill to disk.
Ship a single partition file for each reducer.
Merge the partition file from each mapper, combine, and generate a stream of inputs for the reducer.
Mapper
Mapper Node
merged spills (on disk)
circular buffer (in memory)
sorted spill file
spills (on disk)
other mappers
intermediate files (on disk)
Reducer Node
Reducer
Shuffle is all steps between mapper output to reduce input.
Shuffle is all of this:
• Partition, sort, combine
• Mergesort (repeat while output)
• Send to reducer
• Mergesort partition files
• Stream to reducer
other reducers
When the map function starts producing output, it is not simply written to disk. The process is more involved, and takes advantage of buffering writes in memory and doing some presorting for efficiency reasons.

Summary Lecture 3

Patterns

- Local aggregation
  - Combiners and in-mapper combining
  - Algorithmic correctness with local aggregation
  - Number of tasks and in-memory mapper
- Pairs and stripes
- Computing relative frequencies
- Order inversion pattern in the stream
- Secondary sorts

Synchronization and Communication

- Constructing complex keys and values that bring together data necessary for a computation. This is used in all of the above design patterns.
- Controlling the partition of the intermediate key space. This is used in order inversion and value-to-key conversion.
- Controlling the sort order of intermediate keys. This is used in order inversion and secondary sorting.

Conclusion

- This concludes our overview of MapReduce algorithm design.
- It should be clear by now that although the programming model forces one to express algorithms in terms of a small set of rigidly defined components ...
- ... there are many tools at one's disposal to shape the flow of computation.
- These patterns apply not *just* to Hadoop MapReduce but also to MRJob and Spark, as we shall see during the rest of this class.