



# Batch Processing I

(v1.00)

Week 3: September 18, 2025

Jimmy Lin  
David R. Cheriton School of Computer Science  
University of Waterloo

These slides are available at <https://lintool.github.io/cs451-2025f/>

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International  
See <https://creativecommons.org/licenses/by-nc-sa/4.0/> for details



# Key Questions

What's MapReduce and how does it work with HDFS?

What challenges do communication and skew present in scaling out?

Why is local aggregation important?

**Immutable Truth #1:** At scale, you must distribute work across multiple machines.

**Immutable Truth #2:** At scale, computing components break all the time.

How do you write a program that runs across 100 machines?

## Implications

Must create higher levels of abstraction

Must think about fault tolerance from the beginning

The essence of abstraction is preserving information that is relevant in a given context, and forgetting information that is irrelevant in that context.

– computer scientist John V. Guttag



THIS IS THE WAY

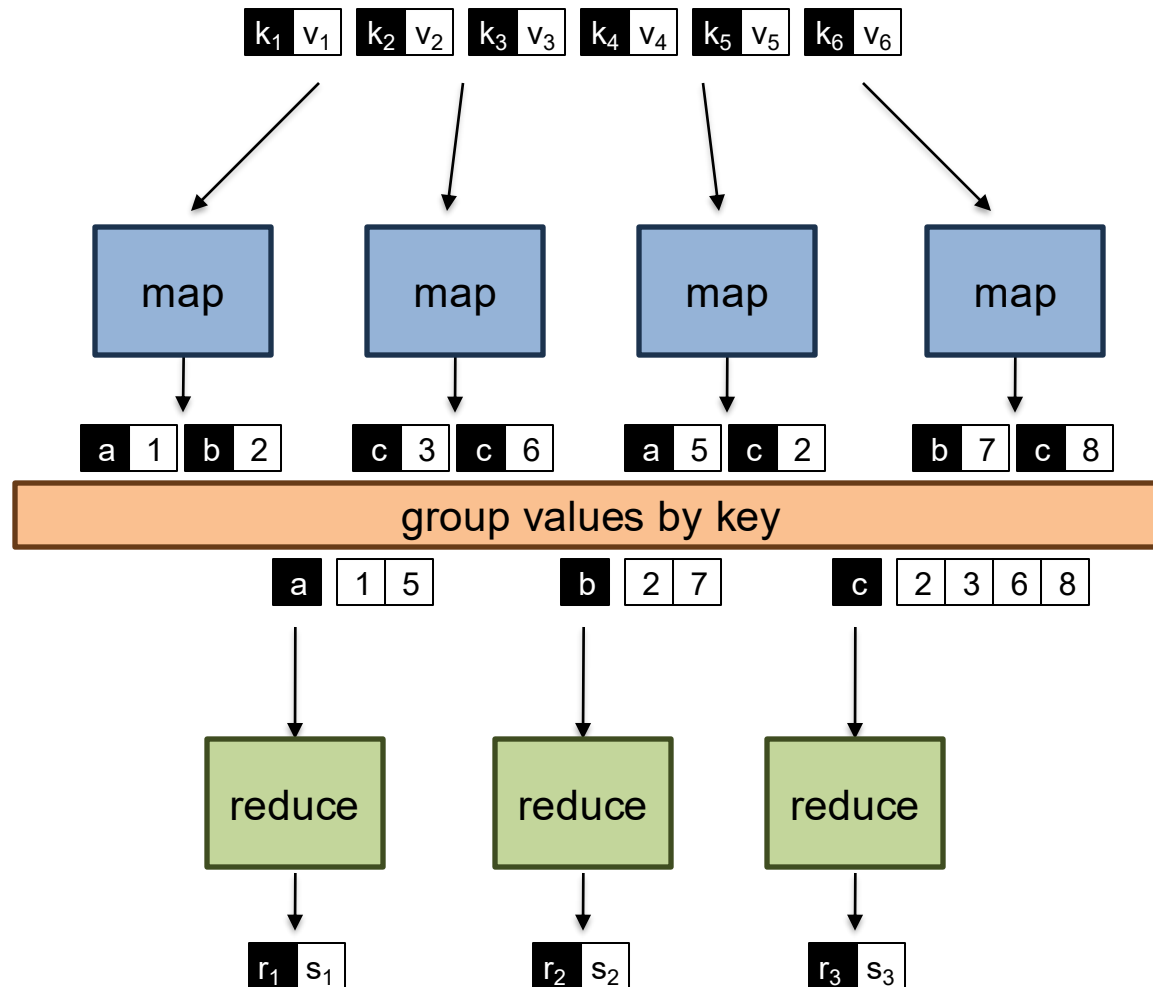
# MapReduce

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





An aerial photograph of a large datacenter facility during sunset. The facility consists of several large, white, rectangular buildings with flat roofs, arranged in a grid-like pattern. In the foreground, there is a large parking lot filled with many white semi-trailers. To the right of the parking lot, there is a large building with a complex roof structure, possibly a cooling or power plant. The facility is surrounded by green fields and some smaller buildings in the distance. The sky is a mix of orange, yellow, and blue, with the sun visible on the left side.

The datacenter *is* the computer!  
What's the instruction set?

# “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)  
}
```



# 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 “runtime” handles everything else...

# MapReduce “Runtime”

Handles scheduling

Assigns workers to map and reduce tasks

Handles “data distribution”

Moves processes to data

Handles coordination

Groups and shuffles intermediate data

Handles errors and faults

Detects worker failures and restarts

**Immutable Truth #1:** At scale, you must distribute work across multiple machines.

**Immutable Truth #2:** At scale, computing components break all the time.

How do you write a program that runs across 100 machines?

### Implications

Must create higher levels of abstraction  
Must think about fault tolerance from the beginning

*tl;dr – MapReduce takes care of it!*

</end> ?

# More? Why do you care?

The essence of abstraction is preserving information that is relevant in a given context, and forgetting information that is irrelevant in that context.

– computer scientist John V. Guttag

1. All abstractions are *leaky*
2. Important to develop intuitions
3. What do you want to be?
4. Curiosity

You don't have to be an engineer to be a racing driver,  
but you do have to have mechanical sympathy

– Formula One driver Jackie Stewart

# 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

That's it?

Not quite...



**Immutable Truth #1:** At scale, you must distribute work across multiple machines.

Ideal scaling characteristics:

Twice the data, twice the running time

Twice the resources, half the running time

Why can't we achieve this?

1. Communication is unavoidable
2. Skew creates idle workers

# Ideal Scaling: Why not?

Communication is unavoidable

Workers need to share intermediate results...

Which requires communication across machines...

Which requires synchronization...

Which kills performance.

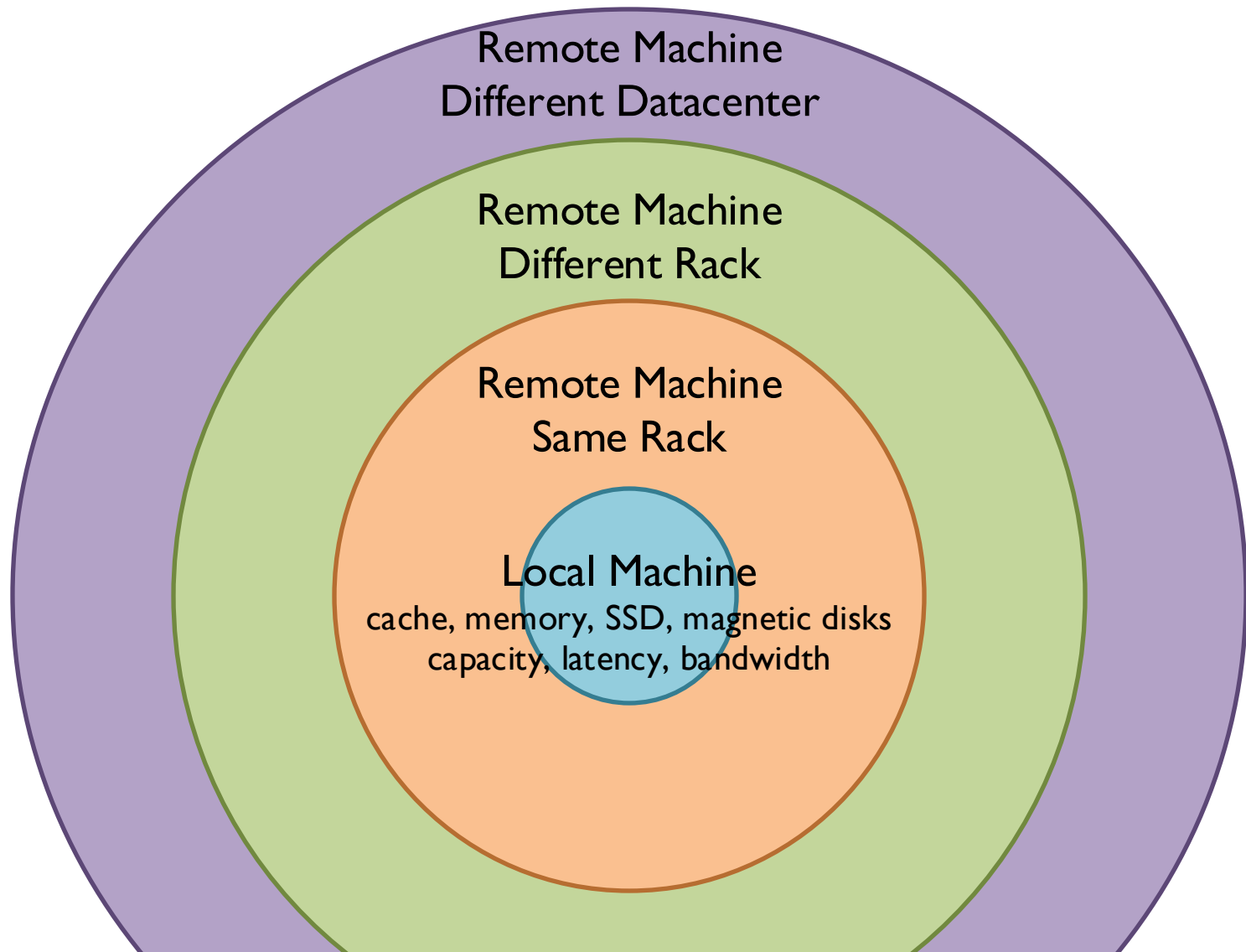
Skew creates idle workers

Tasks are never divided perfectly evenly...

And even if they are, processing times can be unpredictable...

Which leads to idle workers.

# Storage Hierarchy



tl;dr – communication is costly

So let's reduce the amount of data shuffling!  
How? Aggregate intermediate results locally!

# MapReduce

Programmer specifies ~~two~~<sup>four</sup> 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**  $(k', p) \rightarrow 0 \dots p-1$

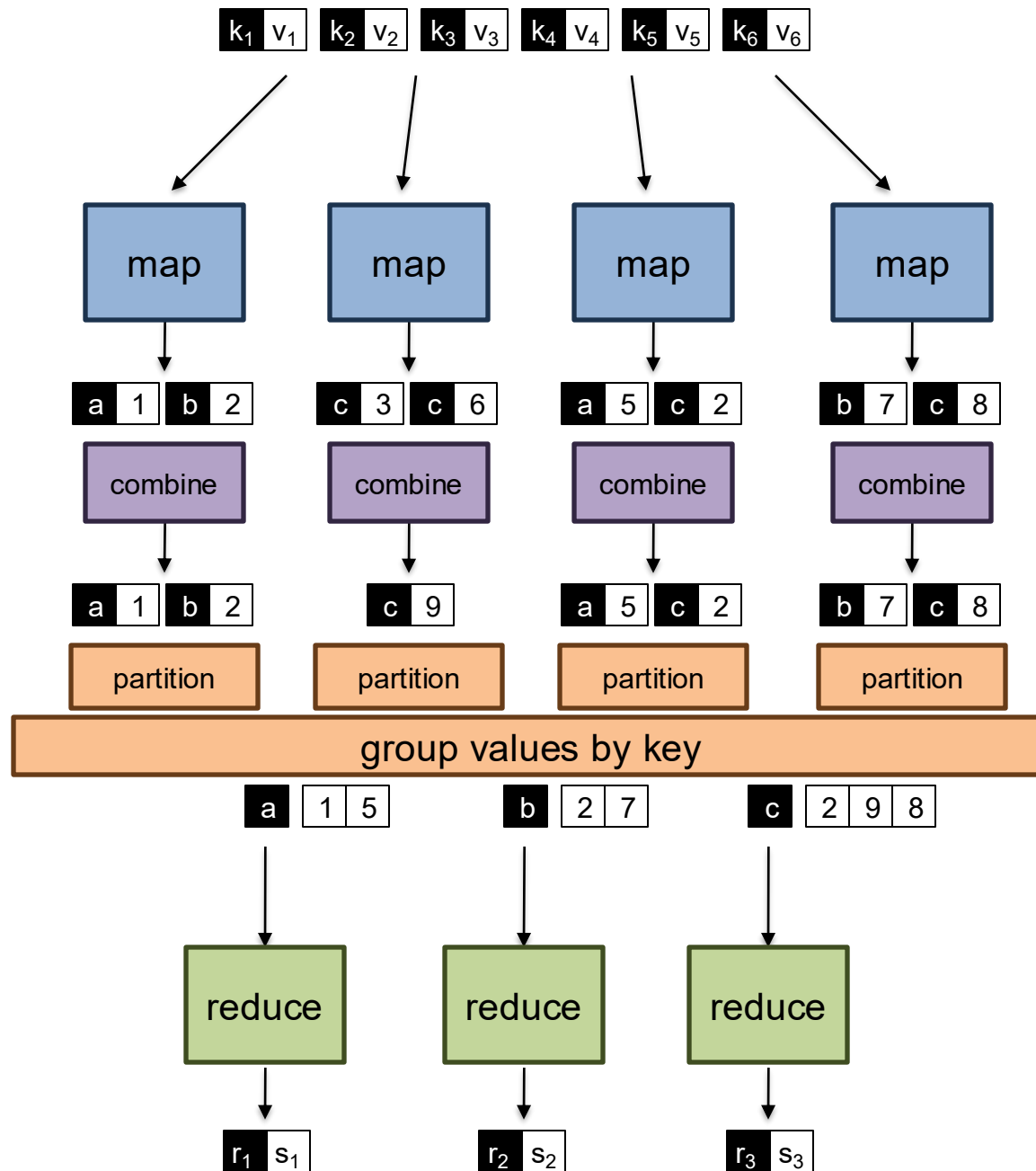
Often a simple hash of the key, e.g.,  $\text{hash}(k') \bmod n$

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 the amount of data shuffling





# Word Count: Baseline

```
class Mapper {  
  def map(key: Long, value: String) = {  
    for (word <- tokenize(value)) {  
      emit(word, 1)  
    }  
  }  
}  
  
class Reducer {  
  def reduce(key: String, values: Iterable[Int]) = {  
    for (value <- values) {  
      sum += value  
    }  
    emit(key, sum)  
  }  
}
```

**What's the impact of combiners?**

# Word Count: Mapper Histogram

```
class Mapper {  
  def map(key: Long, value: String) = {  
    val counts = new Map()  
    for (word <- tokenize(value)) {  
      counts(word) += 1  
    }  
  
    for ((k, v) <- counts) {  
      emit(k, v)  
    }  
  }  
}
```

Are combiners still needed?

# Ideal Scaling: Why not?

Communication is unavoidable

Workers need to share intermediate results...

Which requires communication across machines...

Which requires synchronization...

Which kills performance.

*Combiners help here!*

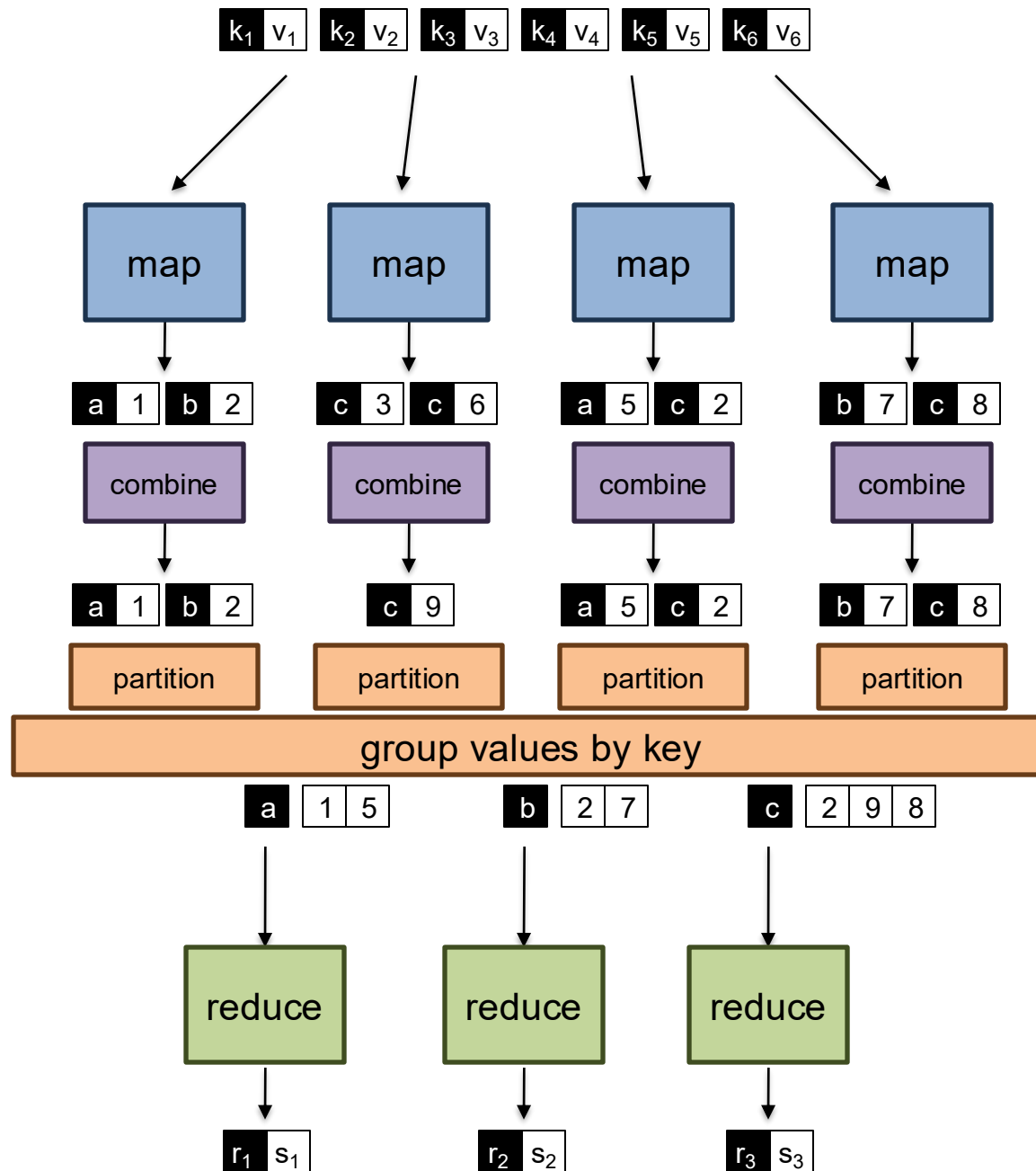
Skew creates idle workers

Tasks are never divided perfectly evenly...

And even if they are, processing times can be unpredictable...

Which leads to idle workers.

*What about partitioners?*



# MapReduce *That's it!*

Programmer specifies ~~two~~<sup>four</sup> 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**  $(k', p) \rightarrow 0 \dots p-1$

Often a simple hash of the key, e.g.,  $\text{hash}(k') \bmod n$

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 the amount of data shuffling

# Which means...

You have 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

...



# Abstractions

## Pros

You don't have to worry about it.  
You don't need to know what's going on.



## Cons

You can't worry about it (even if you wanted to).  
You don't know what's going on (even if you wanted to).



Recap, why combinators?

# Combiner Design

Combiners and reducers share same method signature

Sometimes, reducers can serve as combiners

Often, not...

Combiner are optional optimizations

Should not affect algorithm correctness

May be run 0, 1, or multiple times

**Can reducers be used as combiners?**

Example: find average of integers associated with the same key

# Computing the Mean: Version I

```
class Mapper {  
  def map(key: String, value: Int) = {  
    emit(key, value)  
  }  
}  
  
class Reducer {  
  def reduce(key: String, values: Iterable[Int]) {  
    for (value <- values) {  
      sum += value  
      cnt += 1  
    }  
    emit(key, sum/cnt)  
  }  
}
```

Why can't we use reducer as combiner?

# Computing the Mean: Version 2

```
class Mapper {
  def map(key: String, value: Int) =
    context.write(key, value)
}
class Combiner {
  def reduce(key: String, values: Iterable[Int]) = {
    for (value <- values) {
      sum += value
      cnt += 1
    }
    emit(key, (sum, cnt))
  }
}
class Reducer {
  def reduce(key: String, values: Iterable[Pair]) = {
    for ((s, c) <- values) {
      sum += s
      cnt += c
    }
    emit(key, sum/cnt)
  }
}
```

Why doesn't this work?

# Computing the Mean: Version 3

```
class Mapper {
  def map(key: String, value: Int) =
    context.write(key, (value, 1))
}
class Combiner {
  def reduce(key: String, values: Iterable[Pair]) = {
    for ((s, c) <- values) {
      sum += s
      cnt += c
    }
    emit(key, (sum, cnt))
  }
}
class Reducer {
  def reduce(key: String, values: Iterable[Pair]) = {
    for ((s, c) <- values) {
      sum += s
      cnt += c
    }
    emit(key, sum/cnt)
  }
}
```

Fixed? No!

How?



# Another Example

Term co-occurrence matrix for a text collection

$M = N \times N$  matrix ( $N =$  vocabulary size)

$M_{ij}$ : number of times  $i$  and  $j$  co-occur in some context  
(for concreteness, let's say context = sentence)

Why?

# Large Counting Problems

Term co-occurrence matrix for a text collection  
= specific instance of a large counting problem

A large event space (number of terms)

A large number of observations (the collection itself)

Goal: keep track of interesting statistics about the events

## Basic approach

Mappers generate partial counts

Reducers aggregate partial counts

How do we aggregate partial counts efficiently?

# First Try: “Pairs”

Each mapper takes a sentence:

Generate all co-occurring term pairs

For all pairs, emit (a, b) → count

Reducers sum up counts associated with these pairs

Use combiners!

# Pairs: Pseudo-Code

```
class Mapper {  
  def map(key: Long, value: String) = {  
    for (u <- tokenize(value)) {  
      for (v <- neighbors(u)) {  
        emit((u, v), 1)  
      }  
    }  
  }  
}
```

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

# Pairs: Pseudo-Code

One more thing...

```
class Partitioner {  
  def getPartition(key: Pair, value: Int, numTasks: Int): Int = {  
    return key.left % numTasks  
  }  
}
```

# “Pairs” Analysis

## Advantages

Easy to implement, easy to understand

## Disadvantages

Lots of pairs to sort and shuffle around (upper bound?)

Fewer opportunities for combiners to work

# Another Try: “Stripes”

Idea: group together pairs into an associative array

$(a, b) \rightarrow 1$

$(a, c) \rightarrow 2$

$(a, d) \rightarrow 5$        $a \rightarrow \{ b: 1, c: 2, d: 5, e: 3, f: 2 \}$

$(a, e) \rightarrow 3$

$(a, f) \rightarrow 2$

Each mapper takes a sentence:

Generate all co-occurring term pairs

For each term, emit  $a \rightarrow \{ b: \text{count}_b, c: \text{count}_c, d: \text{count}_d \dots \}$

Reducers perform element-wise sum of associative arrays

$$\begin{array}{r} a \rightarrow \{ b: 1, \quad d: 5, e: 3 \} \\ + \quad a \rightarrow \{ b: 1, c: 2, d: 2, \quad f: 2 \} \\ \hline a \rightarrow \{ b: 2, c: 2, d: 7, e: 3, f: 2 \} \end{array}$$

Key idea: data structure to bring together intermediate results

# Stripes: Pseudo-Code

```
class Mapper {  
  def map(key: Long, value: String) = {  
    for (u <- tokenize(value)) {  
      val map = new Map()  
      for (v <- neighbors(u)) {  
        map(v) += 1  
      }  
      emit(u, map)      a → { b: 1, c: 2, d: 5, e: 3, f: 2 }  
    }  
  }  
}
```

```
class Reducer {  
  def reduce(key: String, values: Iterable[Map]) = {  
    val map = new Map()  
    for (value <- values) {  
      map += value      a → { b: 1,      d: 5, e: 3 }  
                        + a → { b: 1, c: 2, d: 2,      f: 2 }  
                        -----  
                        a → { b: 2, c: 2, d: 7, e: 3, f: 2 }  
    }  
    emit(key, map)  
  }  
}
```



# “Stripes” Analysis

## Advantages

- Far less sorting and shuffling of key-value pairs
- Can make better use of combiners

## Disadvantages

- More difficult to implement
- Underlying object more heavyweight
- Overhead associated with data structure manipulations
- Fundamental limitation in terms of size of event space

# Stripes >> Pairs?

## Tradeoffs

Developer code vs. framework

CPU vs. RAM vs. disk vs. network

Number of key-value pairs: sorting and shuffling data across the network

Size and complexity of each key-value pair: de/serialization overhead

Cost of manipulating data structures

Opportunities for local aggregation

# Tradeoffs

## Pairs:

- Generates a *lot* more key-value pairs
- Fewere combining opportunities
- More sorting and shuffling
- Simple aggregation at reduce

## Stripes:

- Generates fewer key-value pairs (but more complex values)
- More opportunities for combining
- Less sorting and shuffling
- More complex (slower) aggregation at reduce

(At scale, stripes are faster)

Hrm. Can we generalize?  
(Yes, but next time...)

# Deep(-er) Dive: How does this all work?

Where do we place the data?

Where do we place the compute?

**Immutable Truth #1:** At scale, you must distribute work across multiple machines.

**Immutable Truth #2:** At scale, computing components break all the time.

Where do we place the data?

Trick #1: Partition

Trick #2: Replicate

**Answer:**  
**GFS!** (circa 2003)

Remember: There are no solutions, only tradeoffs!

# GFS: Assumptions

Commodity hardware over “exotic” hardware

Scale “out”, not “up”

High component failure rates

Inexpensive commodity components fail all the time

“Modest” number of huge files

Multi-gigabyte files are common, if not encouraged

Files are write-once, mostly appended to

Logs are a common case

Large streaming reads over random access

Design for high sustained throughput over low latency

# GFS: Design Decisions

Files stored as chunks

Fixed size (64MB)

Reliability through replication

Each chunk replicated across 3+ chunkservers

Single master to coordinate access and hold metadata

Simple centralized management

No data caching

Little benefit for streaming reads over large datasets

Simplify the API: not POSIX!

Push many issues onto the client (e.g., data layout)

**HDFS = GFS clone (same basic ideas)**



# From GFS to HDFS

## Terminology differences:

GFS master = Hadoop namenode

GFS chunkservers = Hadoop datanodes

## Implementation differences:

Different consistency model for file appends

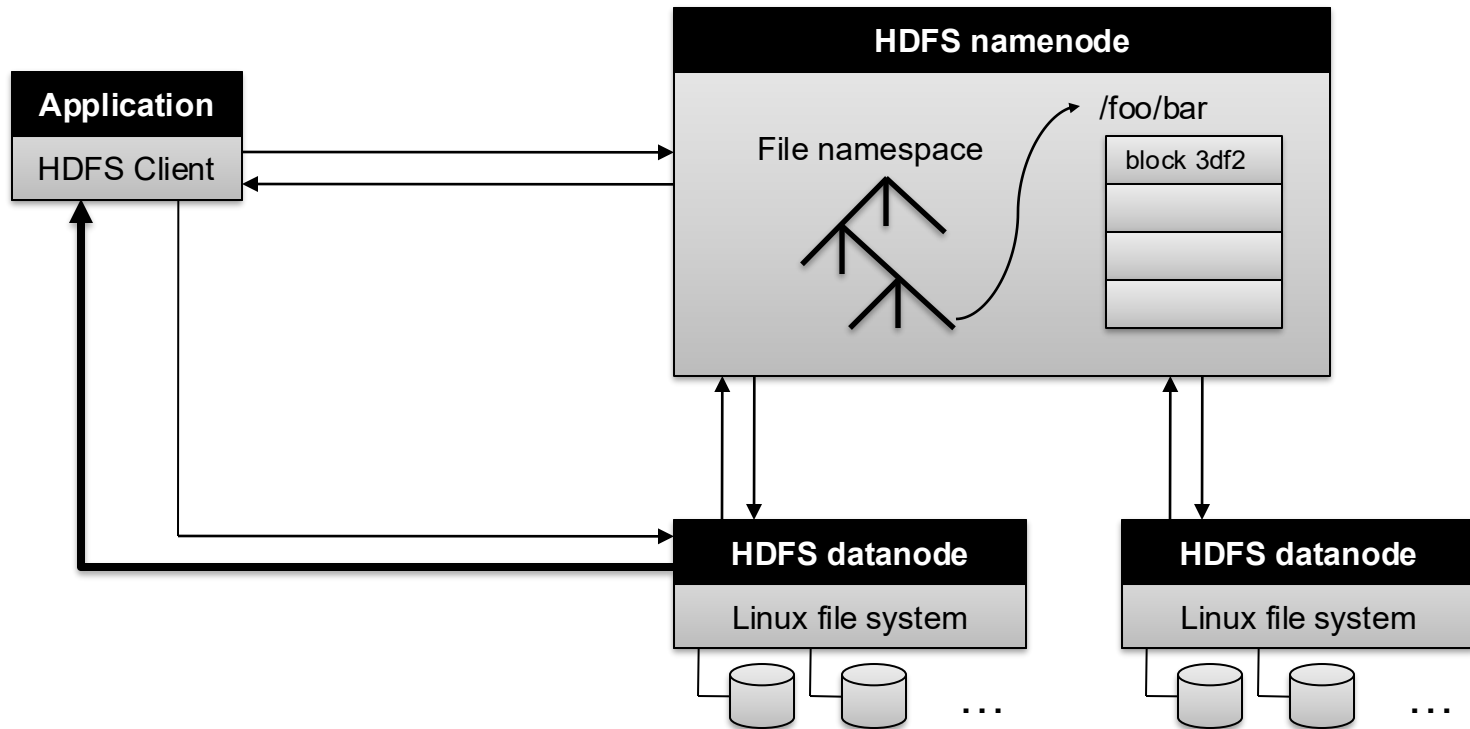
Implementation language

Performance

For the most part, we'll use Hadoop terminology...

# HDFS Architecture

(Yes, SPOF!)



# Namenode Responsibilities

Managing the file system namespace

Holds file/directory structure, file-to-block mapping,  
metadata (ownership, access permissions, etc.)

Coordinating file operations

Directs clients to datanodes for reads and writes  
No data is moved through the namenode

Maintaining overall health

Periodic communication with the datanodes  
Block re-replication and rebalancing  
Garbage collection

# Deep(-er) Dive: How does this all work?

Where do we place the data?

Where do we place the compute?

Immutable Truth #1: At scale, you must distribute work across multiple machines.

Immutable Truth #2: At scale, computing components break all the time.

Where do we place the compute?

Trick #1: Partition

Trick #2: Replicate

Remember: There are no solutions, only tradeoffs!

# Compute meets Data!



Compute Nodes



Storage Nodes

Move data to compute?  
Move compute to data?

# Basic Cluster Components

Namenode (NN)

Holds HDFS metadata

Jobtracker (JT)

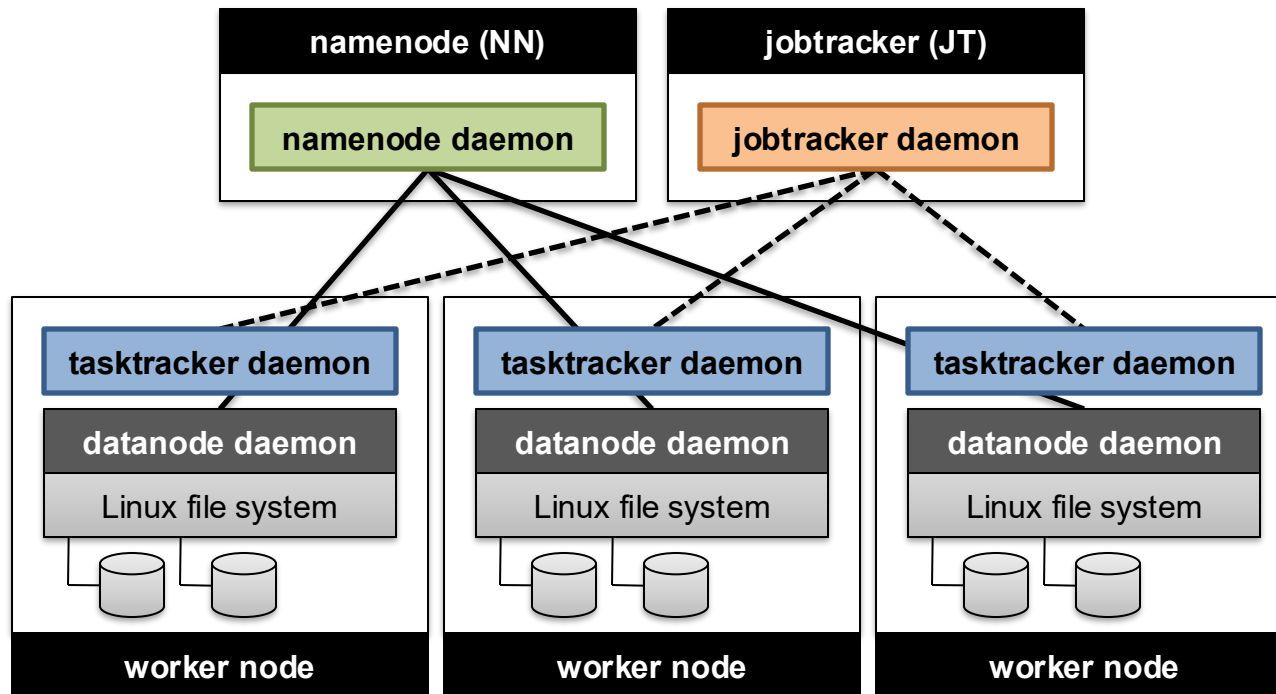
Coordinator for MapReduce jobs

On *each* of the worker machines:

Tasktracker (TT): contains multiple task slots

Datanode (DN): serves HDFS data blocks

# Putting everything together...



Key idea: align map tasks with HDFS blocks



