

Batch Processing I

(v1.00)

Week 3: September 16

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 the difference between scaling up and scaling out?

What are the implications of distributed processing
across many machines?

What are the challenges for a divide-and-conquer strategy?

What challenges does partitioning address?
What challenges does it exacerbate?

What challenges does replication address?
What challenges does it exacerbate?

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

What are we trying to do? tl;dr – everything!

You want

Flexible tools

Diverse data and workloads

High scalability and elasticity

Low latency, high throughput

...

Your boss wants

Cheap

Easy to manage

Small environmental footprint

...

Remember: There are no solutions, only tradeoffs!

Example

Is saving 1.27ms in latency
worth \$1 billion dollars?



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

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

Trick #1: Partition

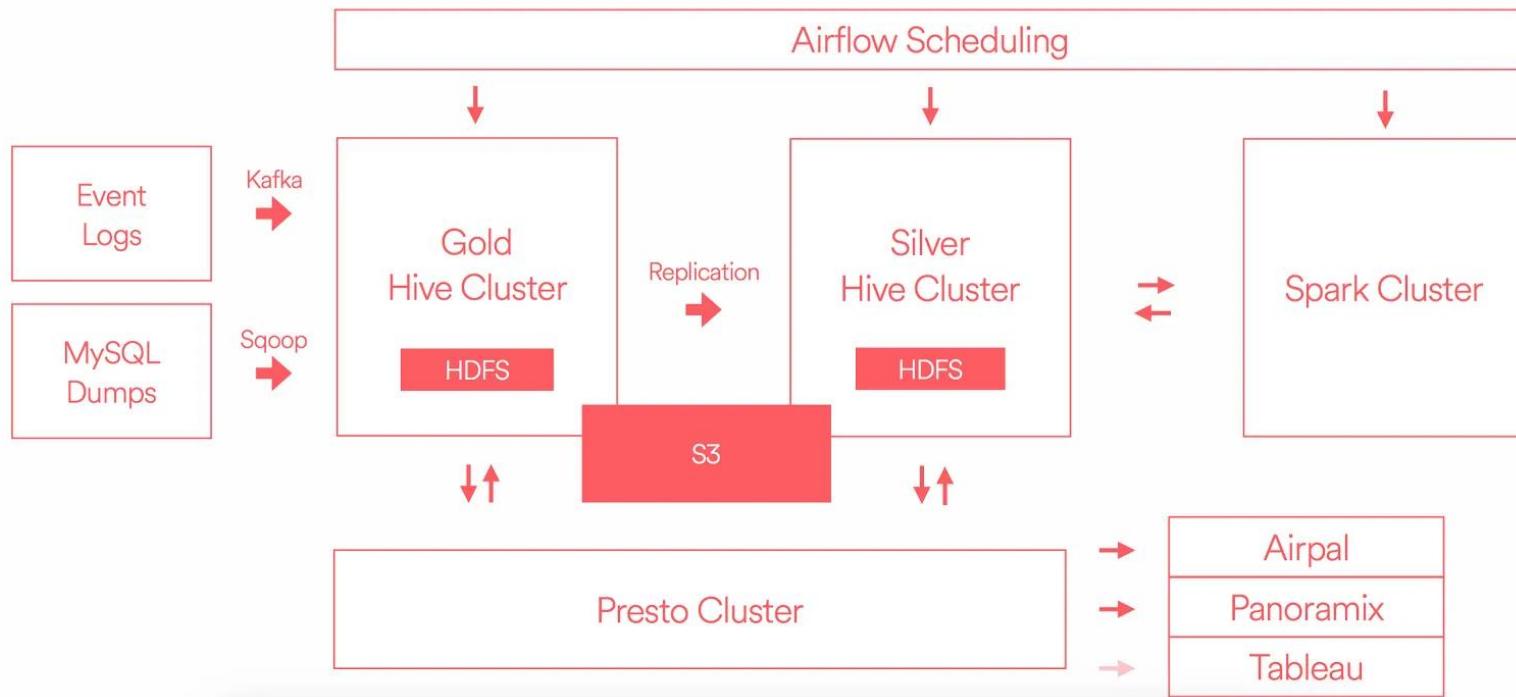
Trick #2: Replicate

Remember: There are no solutions, only tradeoffs!

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

Obvious?

AIRBNB DATA INFRA

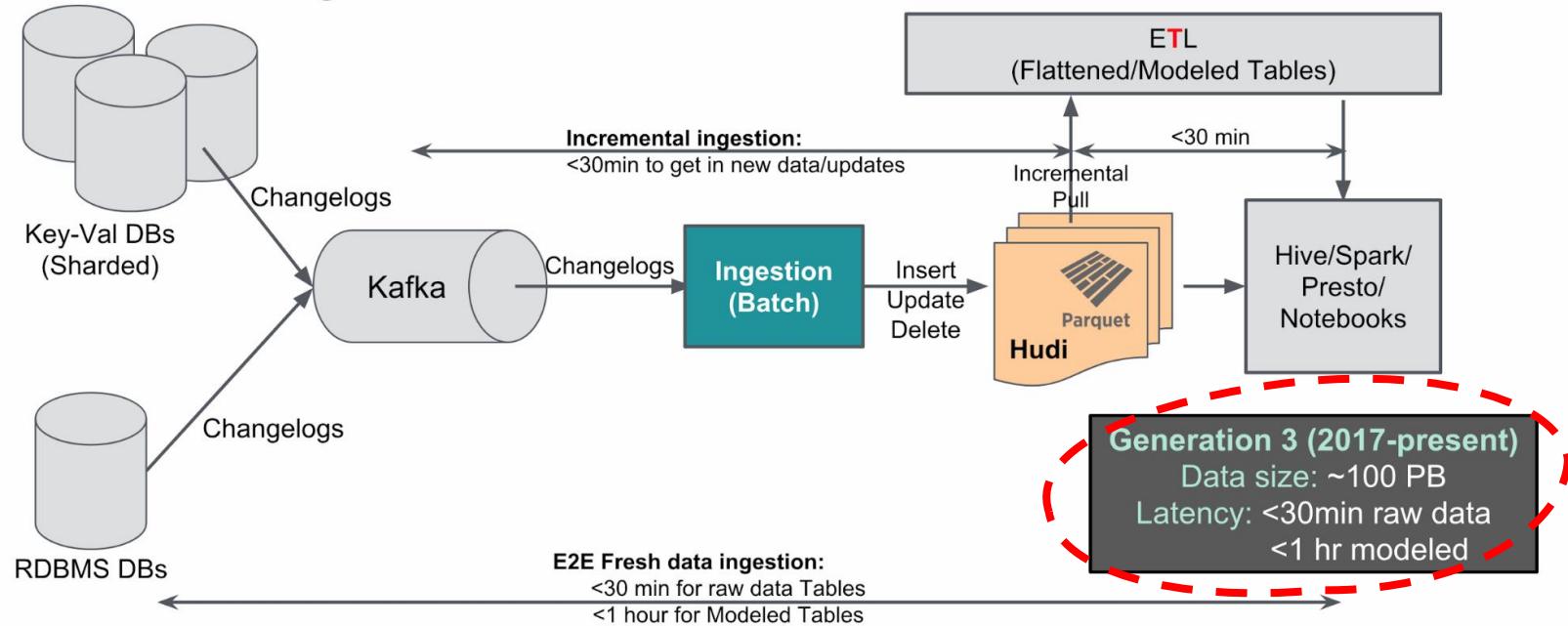


To set some context for scale, two years ago we moved from Amazon EMR onto a set of EC2 instances running HDFS with 300 terabytes of data. Today, we have two separate HDFS clusters with 11 petabytes of data and we also store multiple petabytes of data in S3 on top of that.

AirBnB's data platform (circa 2016)

Generation 3 (2017-present) - Let's rebuild for long term

Incremental ingestion:



Uber's data platform (circa 2018)

<https://www.uber.com/en-CA/blog/uber-big-data-platform/>

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

Obvious?

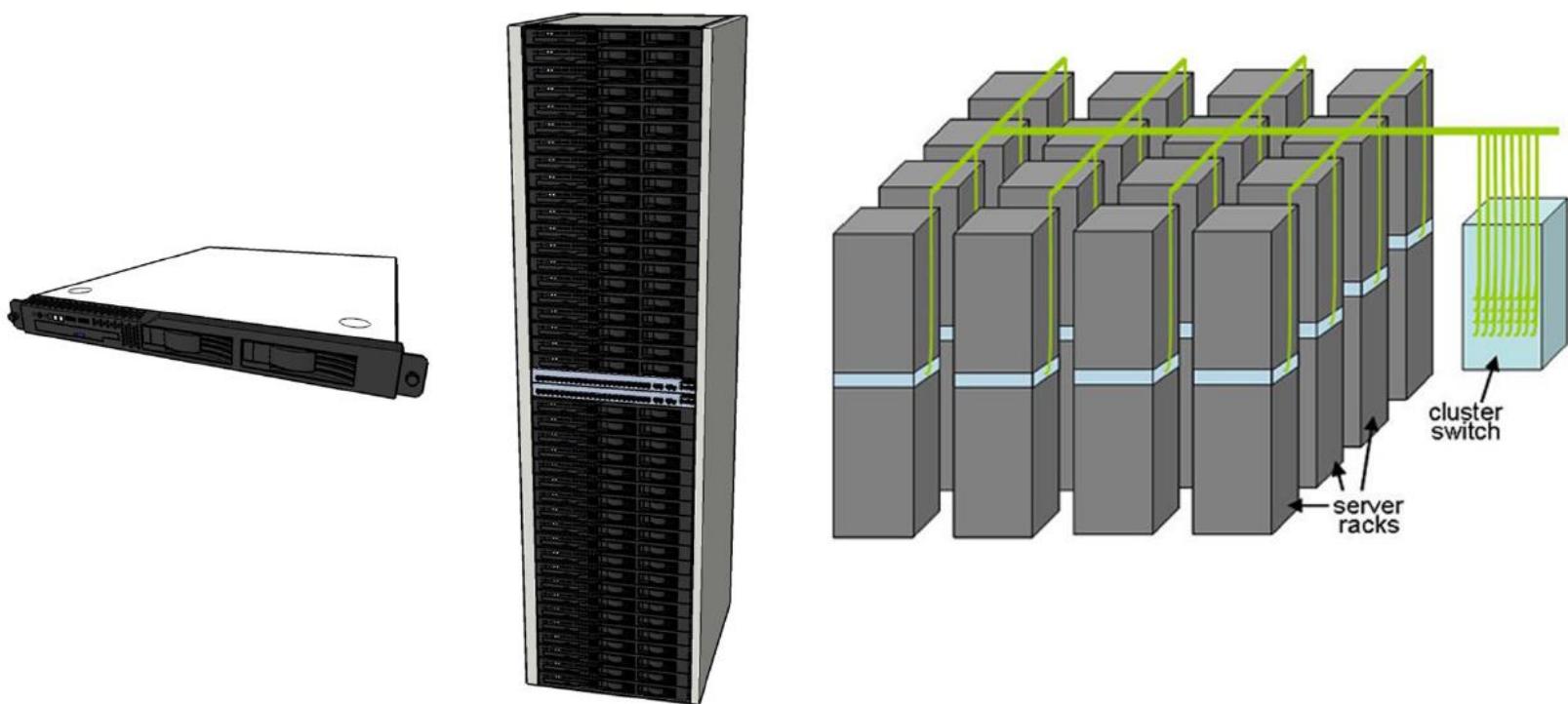
Scale-out vs. Scale-up

... but don't under-estimate the power of a single "beefy" machine

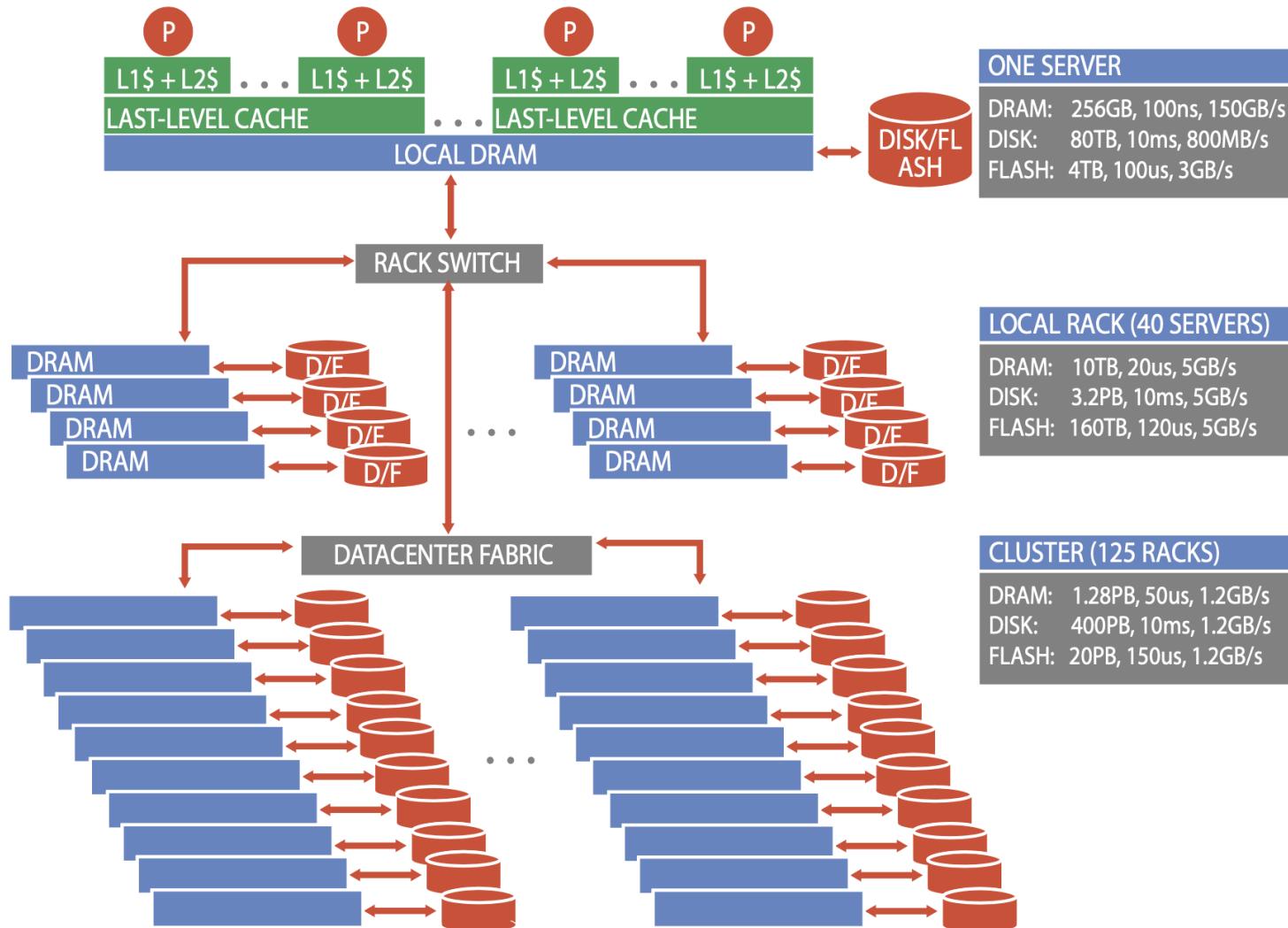
GCP x4-megamem-1920-metal instance:
1920 vCPUs, 32,768 GB RAM, 512 TiB disk

<https://cloud.google.com/compute/docs/memory-optimized-machines>

Building Blocks



Datacenter Organization

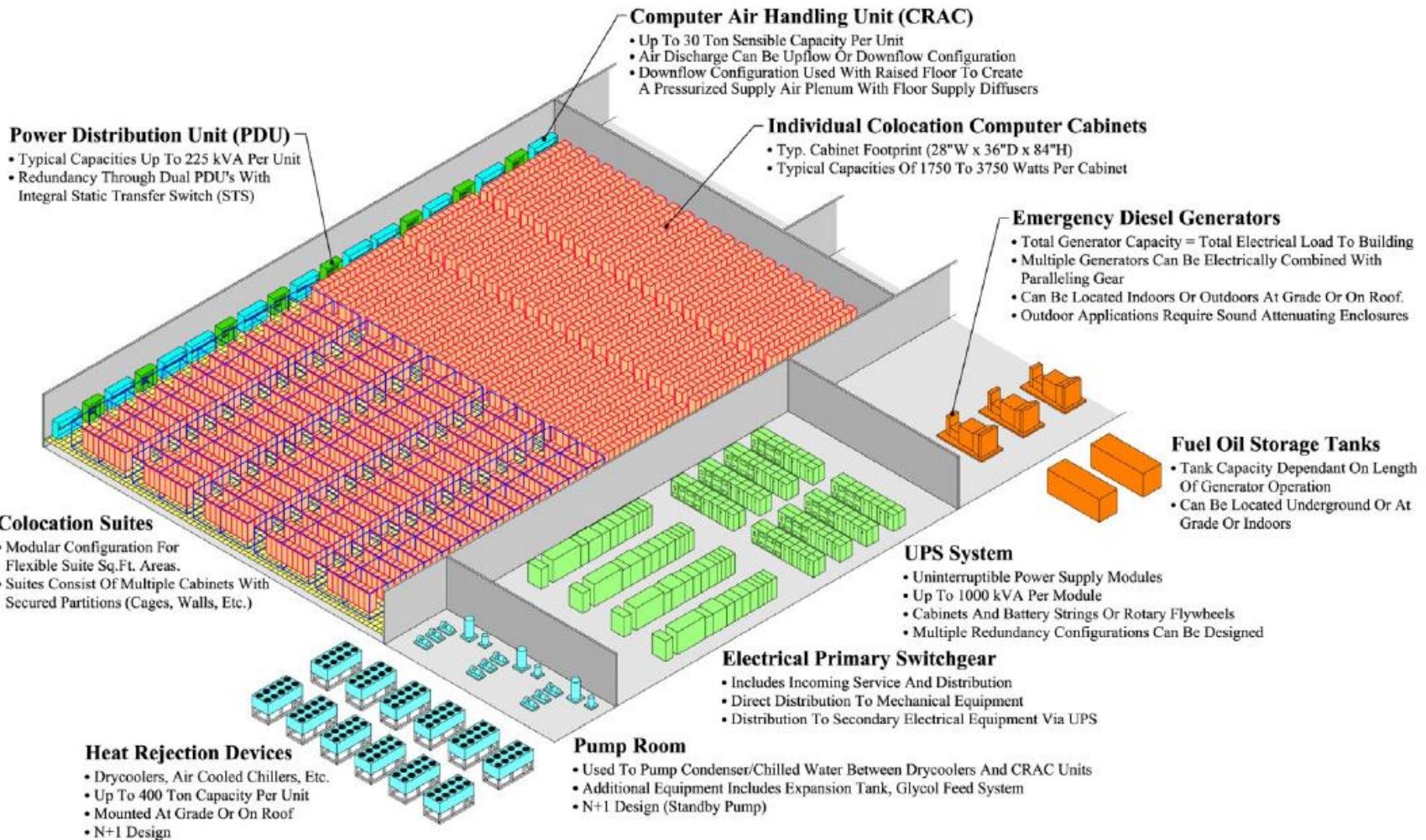






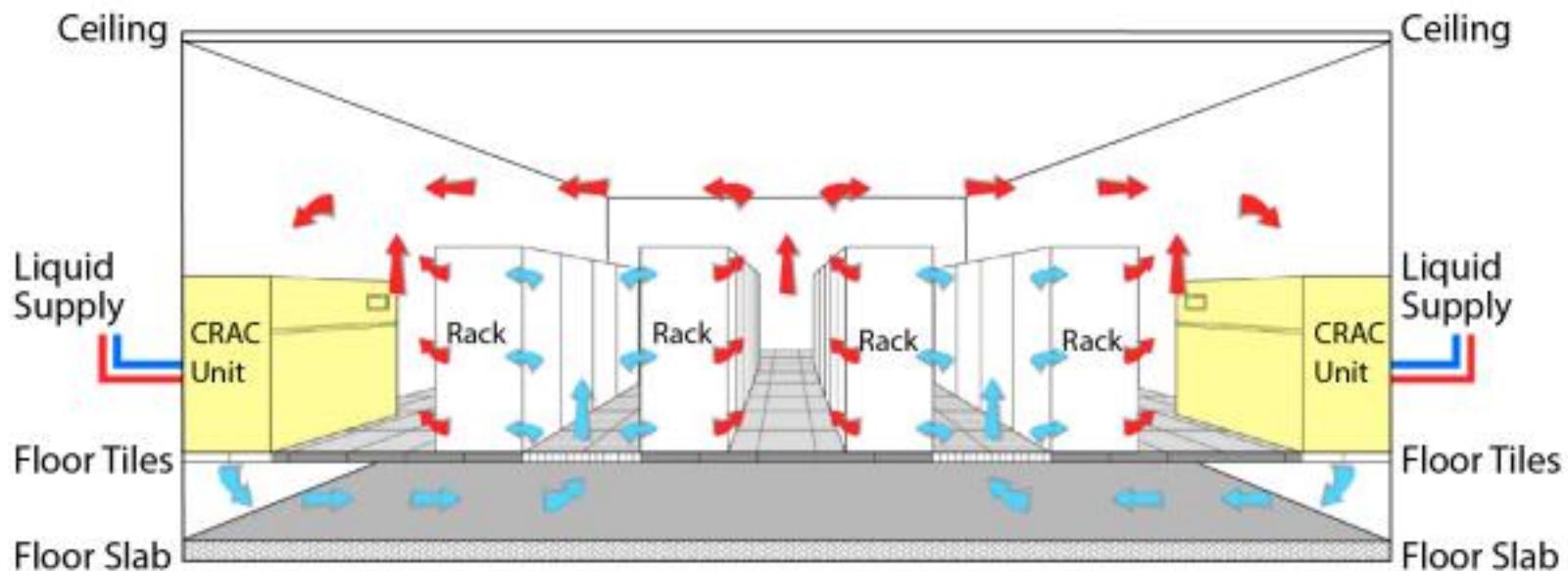


Anatomy of a Datacenter

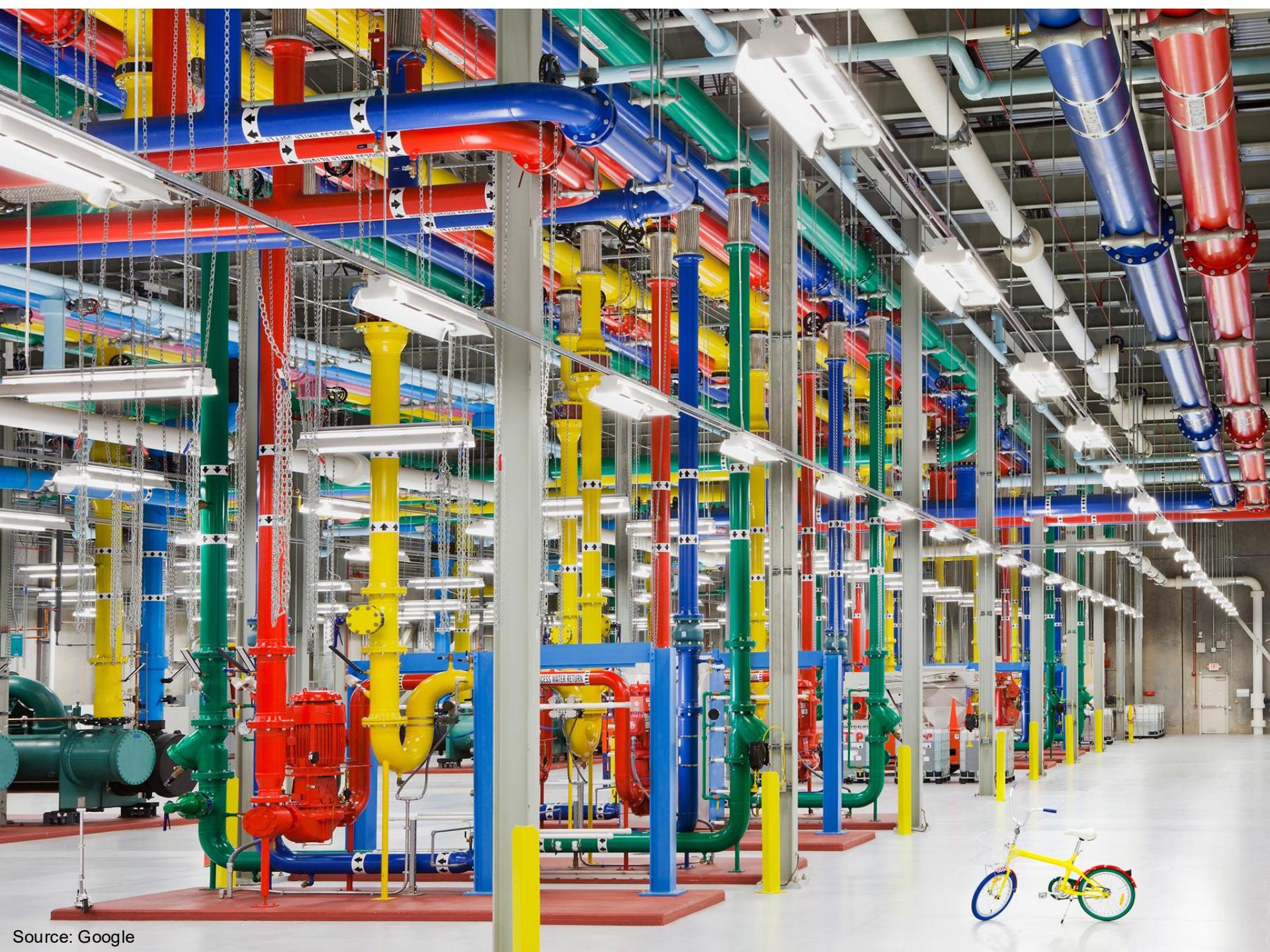


Datacenter Cooling

What's a computer?



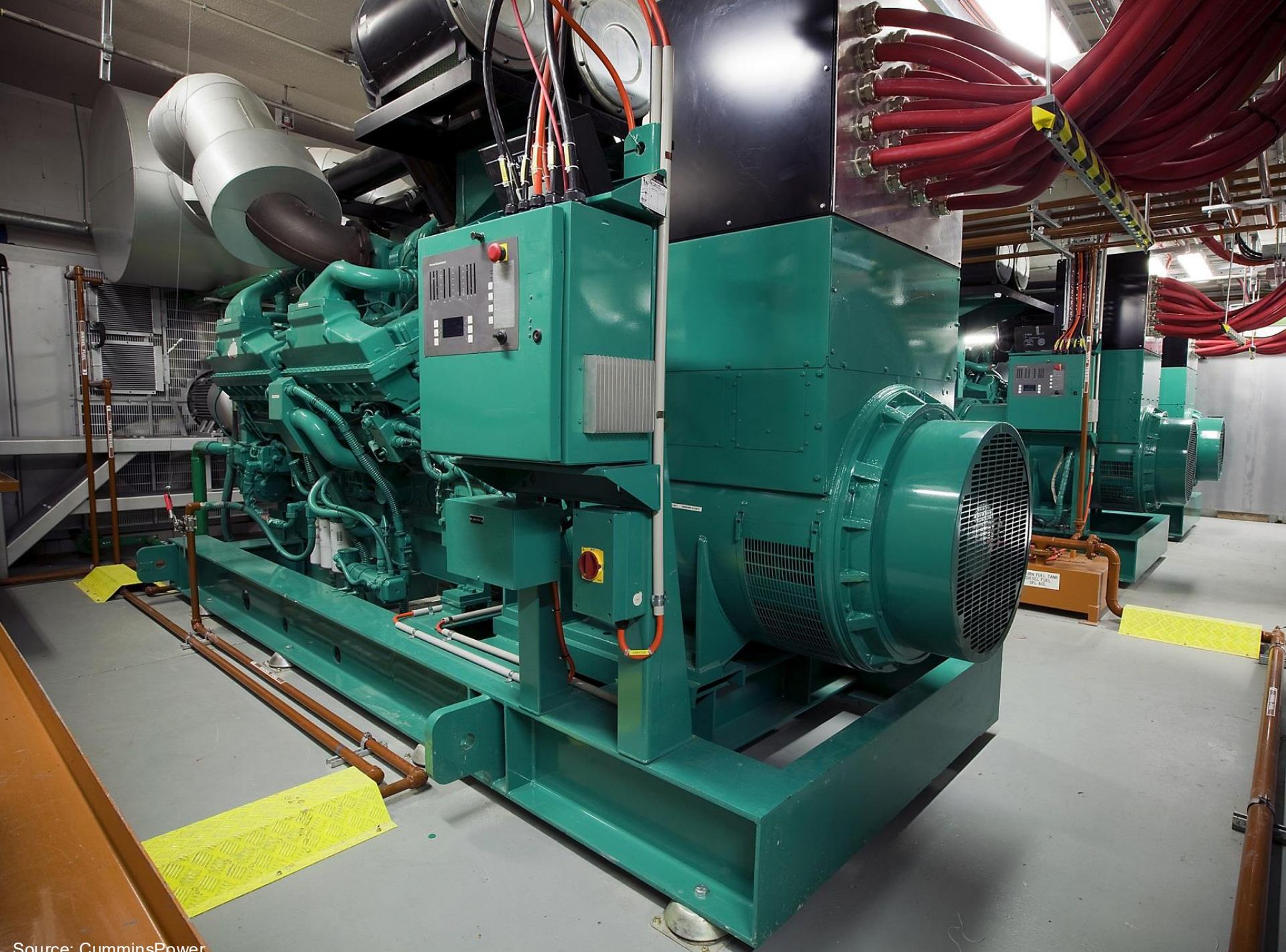




Source: Google







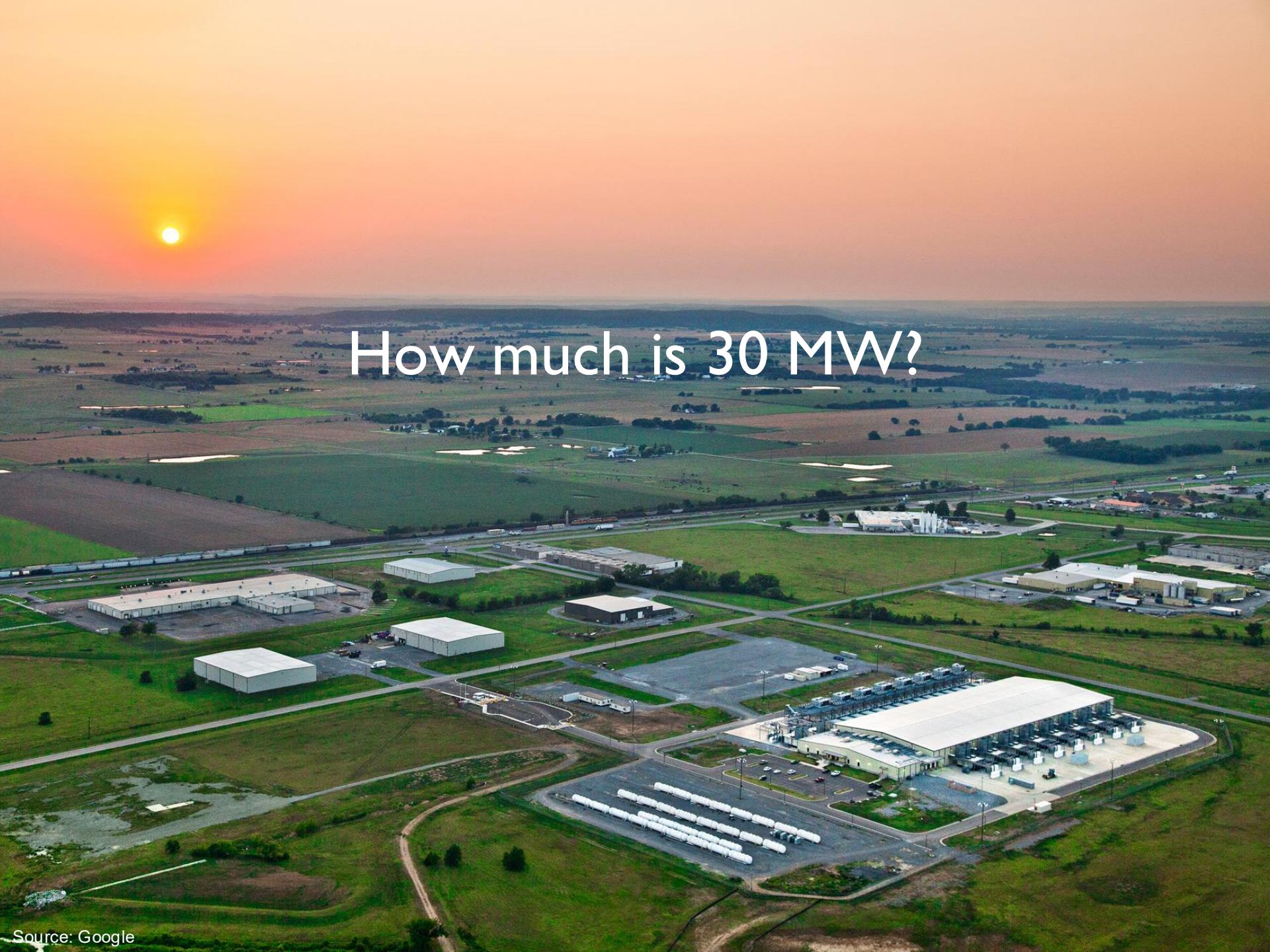




Source: Wikipedia (The Dalles, Oregon)



Source: Bonneville Power Administration

An aerial photograph of a large industrial complex during sunset. The sky is a vibrant orange and yellow. In the foreground, there's a large building with a white roof, several smaller buildings, and a parking lot with white cylindrical storage tanks. A road or highway cuts through the facility. In the background, there are more buildings, a railway line with a train, and a vast expanse of green fields stretching to a distant horizon.

How much is 30 MW?

Konstantin Pilz
@KonstantinPilz

∅ ...

After 2 years tracking GPU clusters, we're releasing our dataset of 700+ AI supercomputers.

The US clearly dominates countries, and xAI's Colossus is leading system: 200k AI chips, \$7B price tag, and power needs of a medium-sized city.

Here are my personal highlights 

Name	xAI Colossus Memphis Phase 2
First operational date	Feb. 18, 2025
Hardware	150,000 NVIDIA H100 SXM5 80GB
Secondary hardware	50,000 NVIDIA H200 SXM
H100 equivalents (8-bit)	200,000
Power capacity	280 MW
Country	United States of America

 EPOCH AI

Power capacity of AI supercomputers





Elon Musk @elonmusk

Subscribe



Colossus 2, built by [@xAI](#), will be the world's first Gigawatt+ AI training supercomputer



Michael Dell @MichaelDell · Aug 21

Great visit to [@xAI](#) with [@BrentM_SpaceX](#) [@nmswede](#) today! It's amazing to see what you guys are accomplishing and we couldn't be prouder to be part of it. I very much enjoyed it. Onward [x.com/BrentM_SpaceX/...](https://x.com/BrentM_SpaceX/)

7:01 AM · Aug 22, 2025 · 25M Views



Sir Adam Beck Hydroelectric
Generating Stations: 1,962 MW

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

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

Example: SSD Failure

Thanks to ChatGPT 😊

Assume AFR (Annual Failure Rate) of 1% (= 0.01 per year)

Assume constant hazard

The per-day failure probability for one drive is:

$$p_{day} = 0.01/365 \approx 0.000027397$$

For N independent drives, the daily failure count is well-approximated by a Poisson random variable with rate: $\lambda = N \times p_{day}$

Expected failures per day = λ

Chance of at least one failure today = $1 - e^{-\lambda}$

10,000 SSDs: ≈ 0.274 failures / day

$P(\geq 1 \text{ failure today}) \approx 24\%$

100,000 SSDs: ≈ 2.74 failures / day

$P(\geq 1 \text{ failure today}) \approx 94\%$

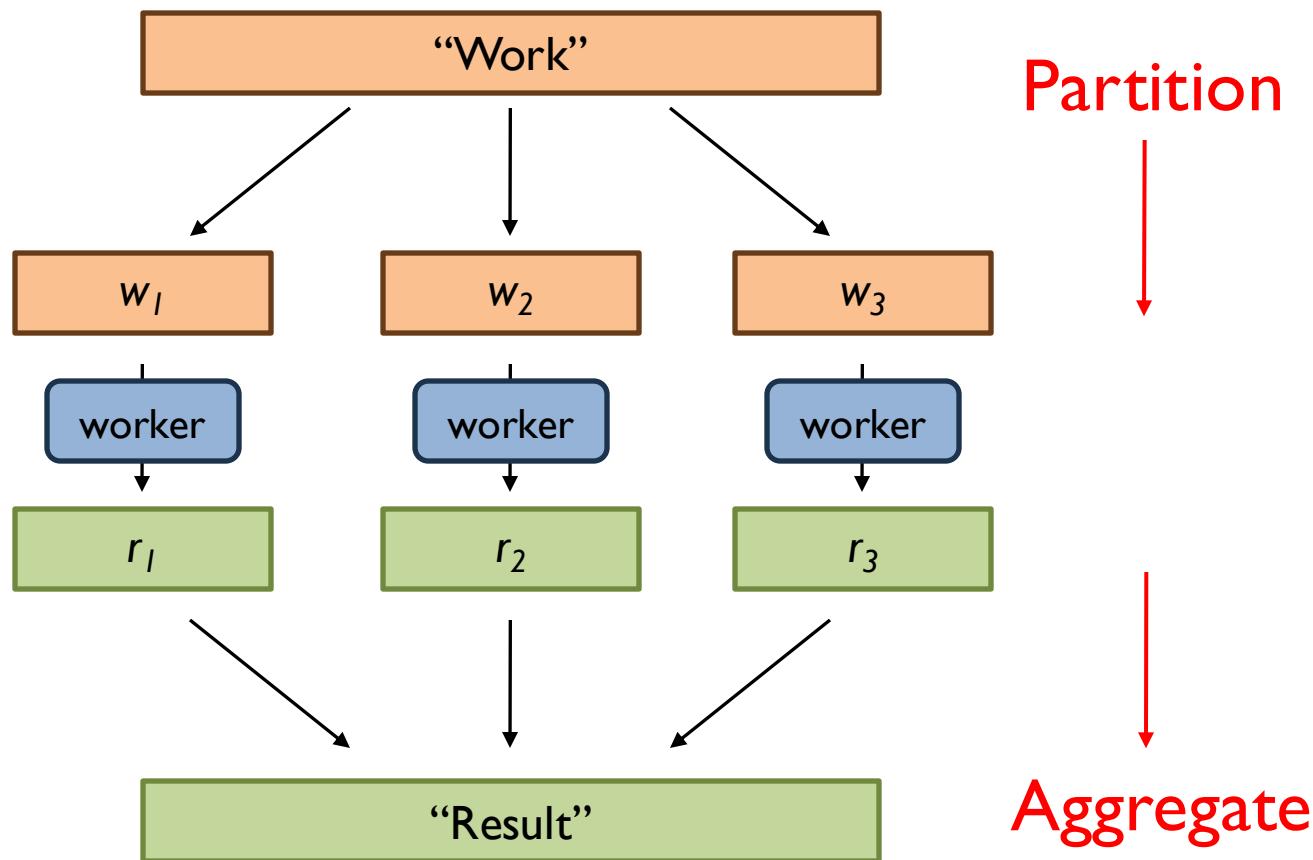
Data-Intensive Distributed Processing

Divide and Conquer!

A close-up, over-the-shoulder shot of The Mandalorian's helmet. He is looking towards the right, where a large, dark, horned animal, possibly a Tauntaun, is partially visible against a bright, hazy background.

THIS IS THE WAY

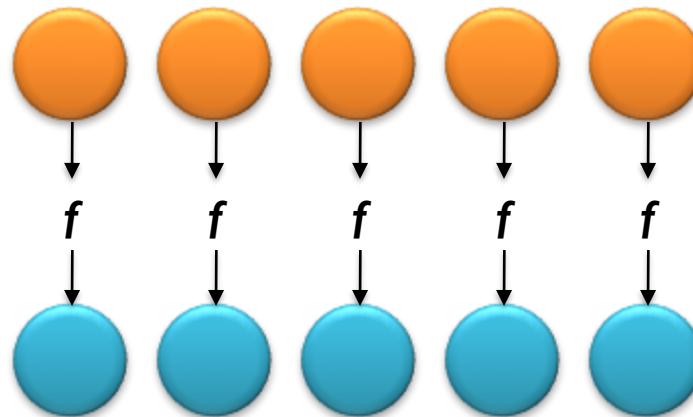
Divide and Conquer



Roots in Functional Programming

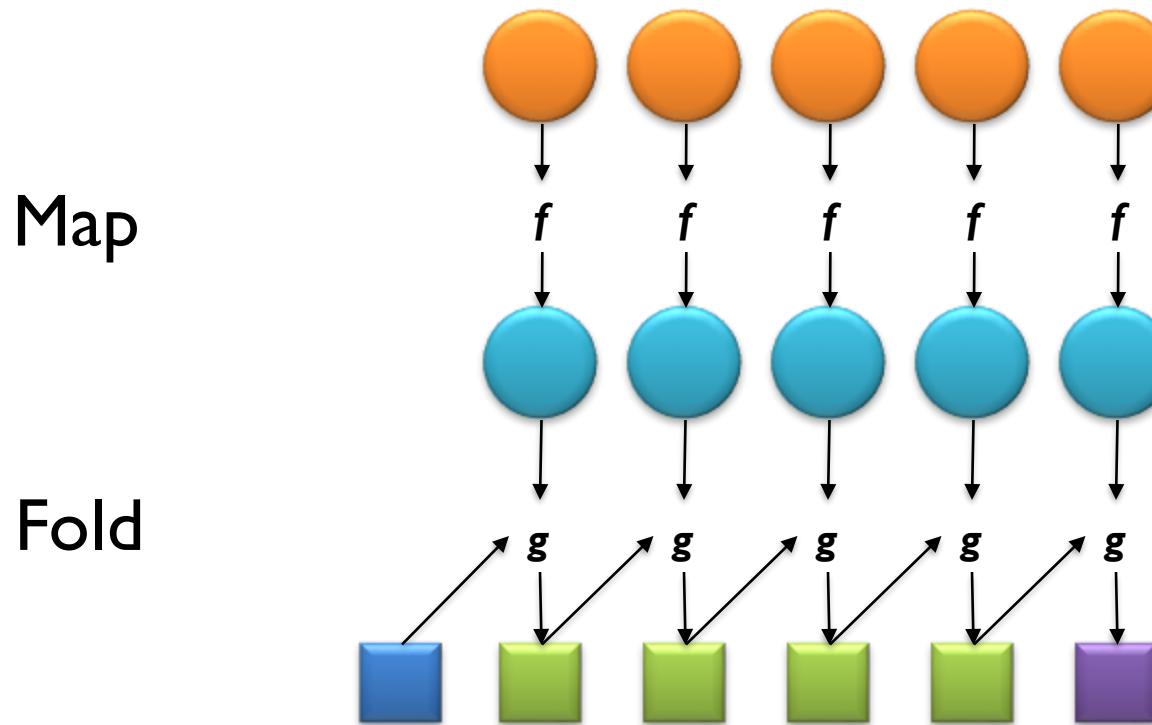
Partition: process many records by “doing” something to each (f)

Map



Roots in Functional Programming

Aggregate: combine results in a particular way (g)



Functional Programming in Scala

```
scala> val t = Array(1, 2, 3, 4, 5)
t: Array[Int] = Array(1, 2, 3, 4, 5)
```

```
scala> t.map(n => n*n)
res0: Array[Int] = Array(1, 4, 9, 16, 25)
```

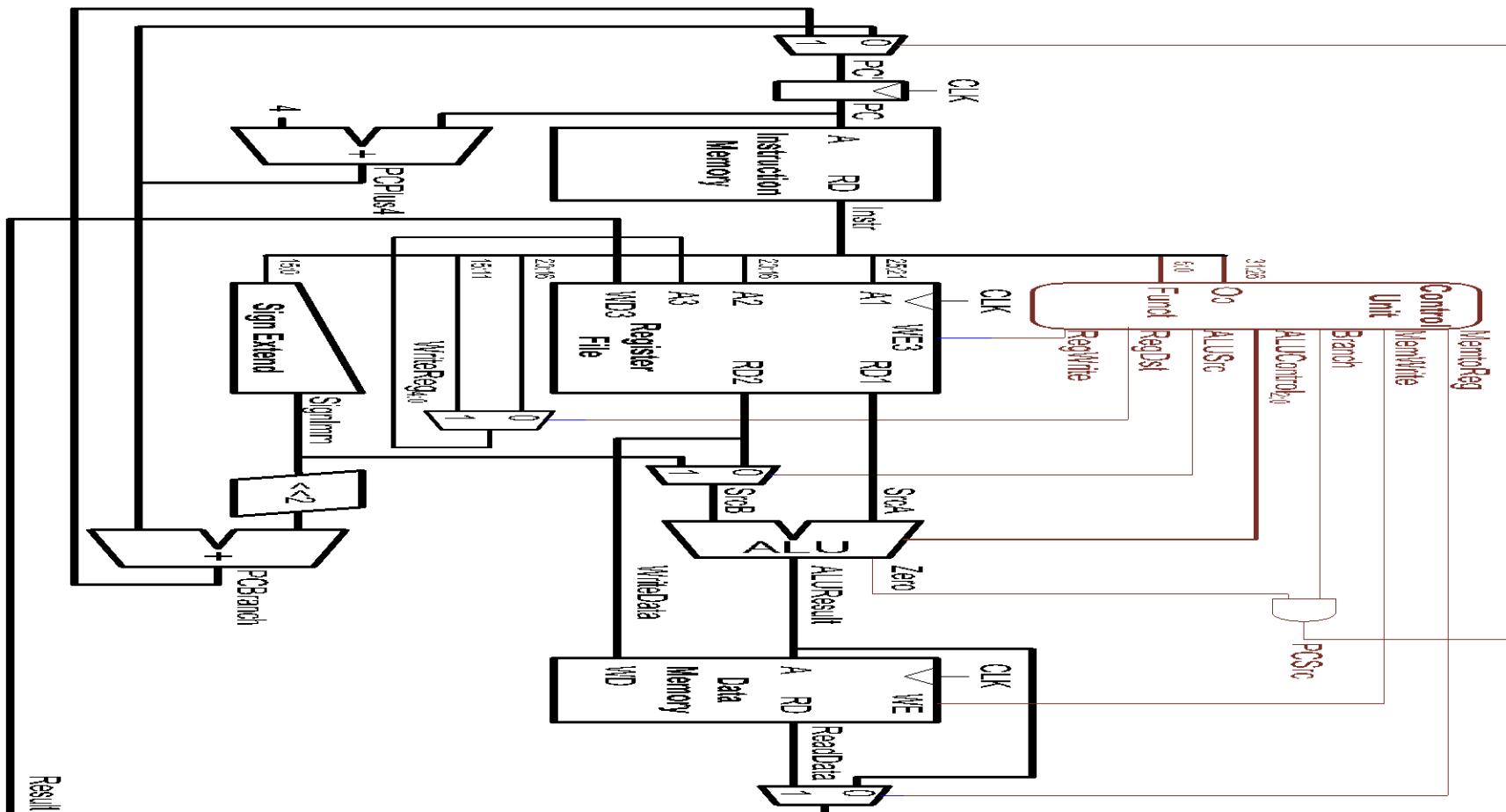
```
scala> t.map(n => n*n).foldLeft(0)((m, n) => m + n)
res1: Int = 55
```

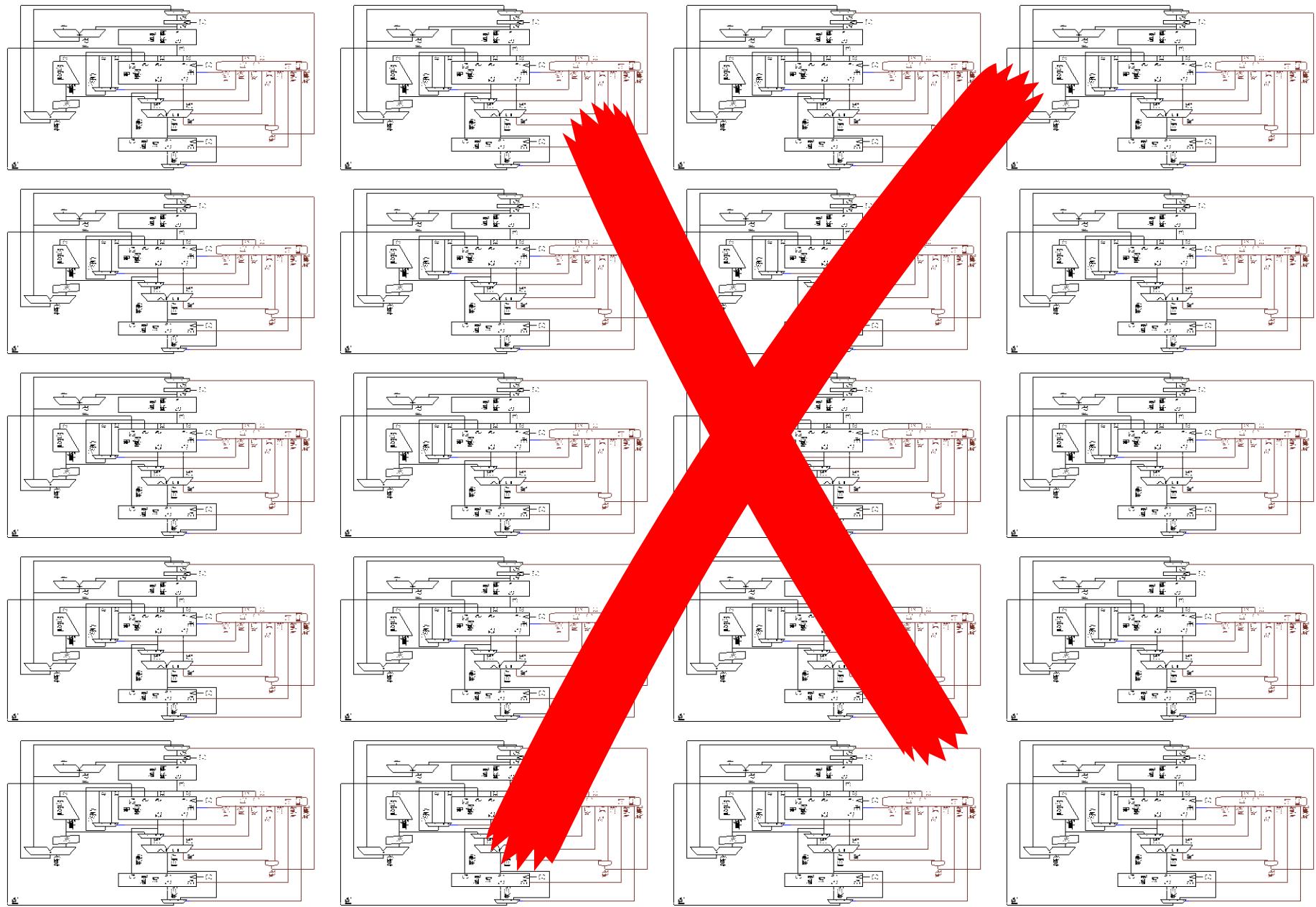
Now do this across many machines...

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?







The datacenter *is* the computer!

An aerial photograph of a large datacenter complex during sunset. The sky is a vibrant orange and yellow. In the foreground, there are several large industrial buildings, parking lots, and rows of white shipping containers. A major highway runs through the middle ground. The background shows a vast, flat landscape with green fields and distant hills under the setting sun.

The datacenter *is* the computer!

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 build higher-level abstractions

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



The datacenter *is* the computer!

An aerial photograph of a massive data center complex during sunset. The sky is a vibrant orange and yellow. In the foreground, there's a large building with a white roof and several rows of white shipping containers or server racks in a parking lot. Behind it, there's a mix of smaller buildings, some green fields, and a highway with a few cars. The terrain is flat and extends to a distant horizon.

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

A Data-Parallel Abstraction

Start with a large number of records

Map “Do something” to each

Group intermediate results

“Aggregate” intermediate results

Write final results

Reduce

Key idea: provide a functional abstraction for these two operations

Dean and Ghemawat (2004)

Historical Note

Google “invented” MapReduce
Hadoop is an open-source implementation
Unless explicitly stated otherwise, we’re referring to Hadoop

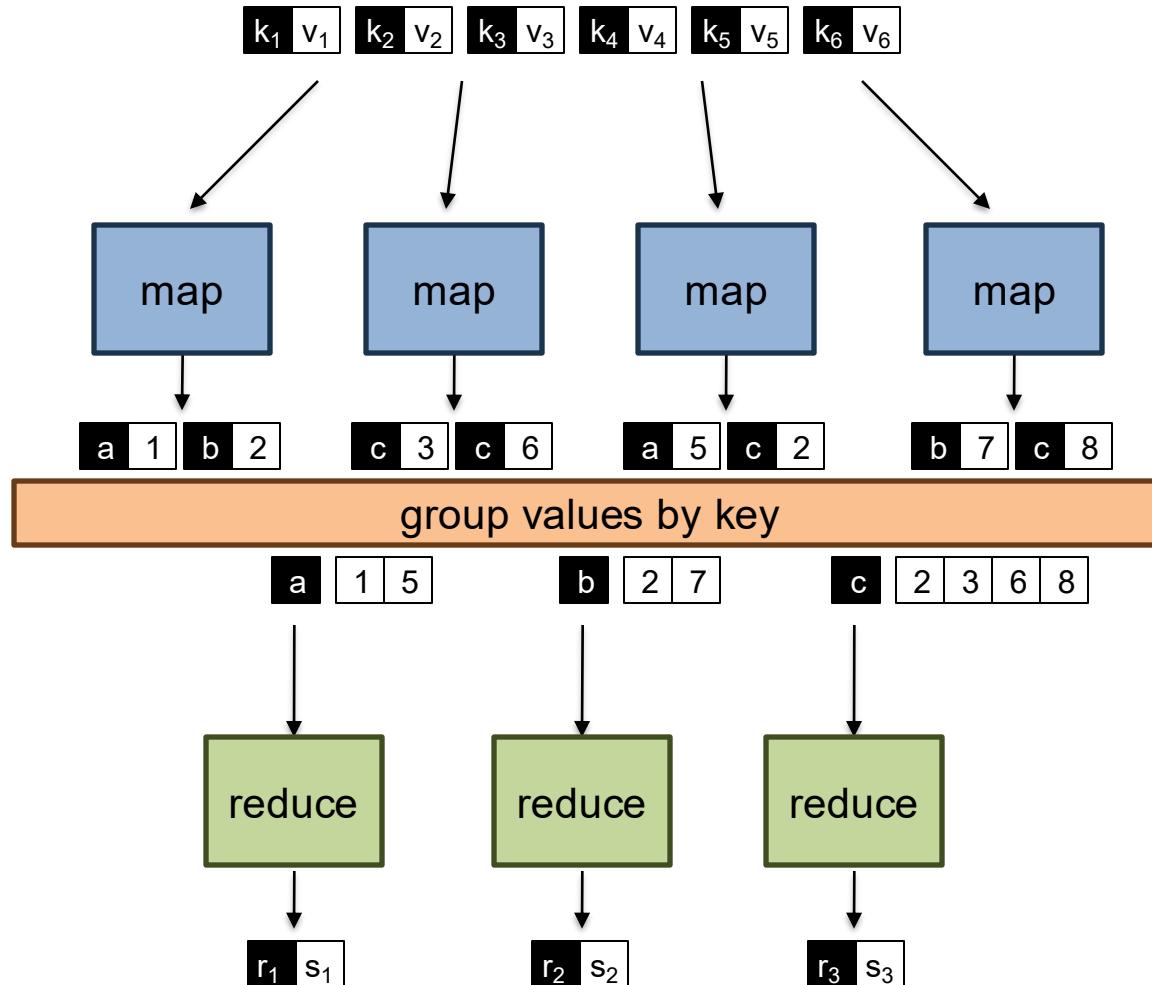


MapReduce

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



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...

What’s “everything else”?

MapReduce “Runtime”

Handles scheduling

Assigns workers to map and reduce tasks

Handles “data distribution”

Moves code to data

Handles coordination

Groups and shuffles intermediate data

Handles errors and faults

Detects failures and compensates

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...
(Not quite... but later)

“Hello World” MapReduce: Word Count

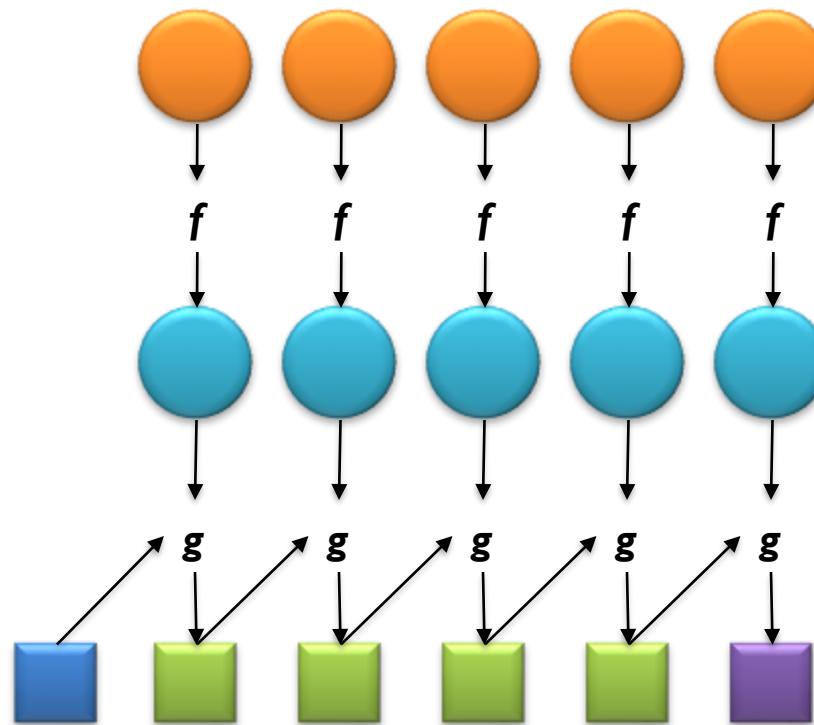
```
def map(key: Long, value: String) = {
    for (word <- tokenize(value)) {
        emit(word, 1)
    }
}

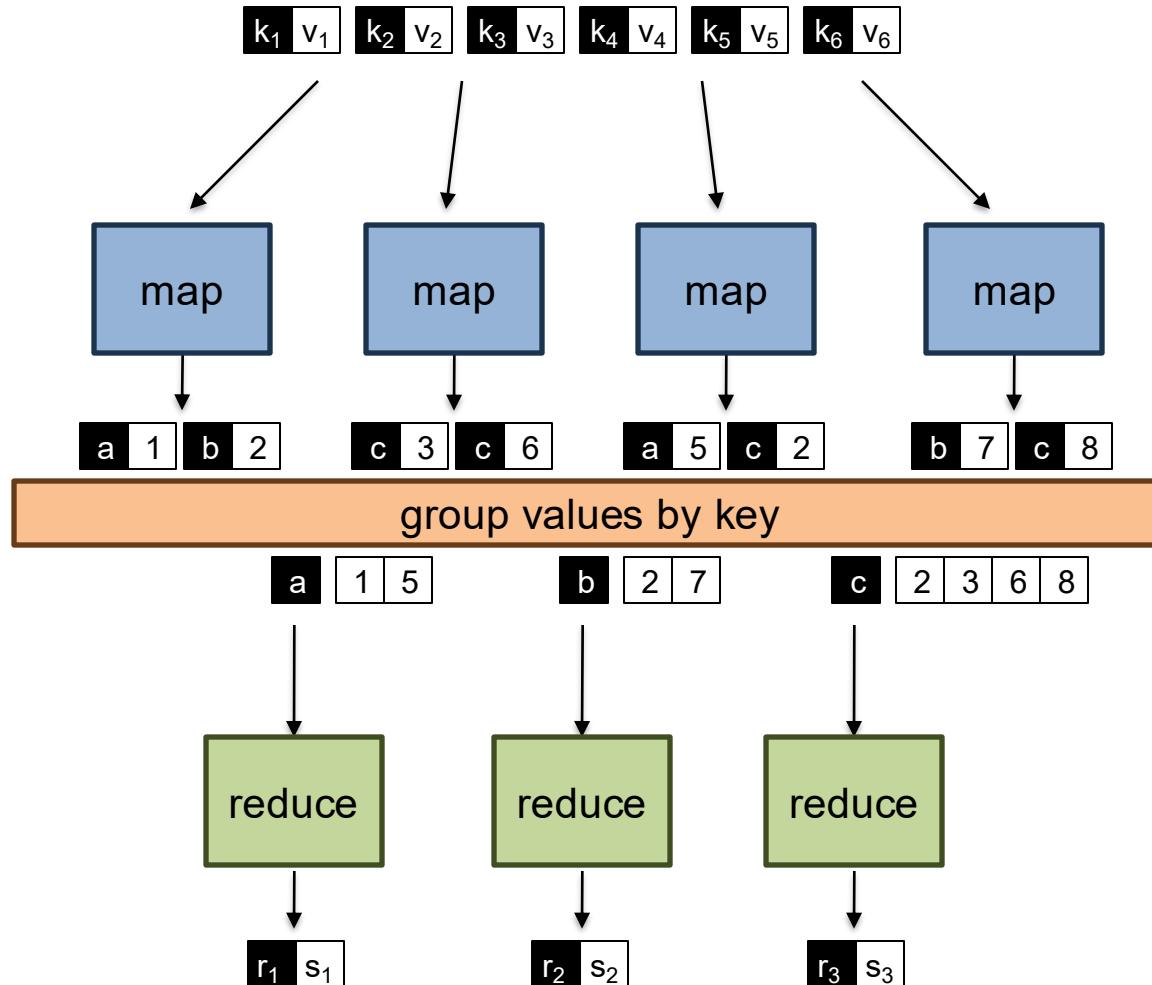
def reduce(key: String, values: Iterable[Int]) = {
    for (value <- values) {
        sum += value
    }
    emit(key, sum)
}
```

Roots in Functional Programming

Map

Fold





That's it.

MapReduce

```
results = records.map(...)  
          .reduce(...)  
  
results2 = results1.map(...)  
          .reduce(...)
```

Spark

```
results = rdd.foo(...)  
          .bar(...)  
          .baz(...)
```

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

One More...

In the cloud, does any of this matter?

The cloud is just another abstraction!

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).



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

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

Trick #1: Partition

Trick #2: Replicate

Remember: There are no solutions, only tradeoffs!

Partition

tl;dr – (1) divide data and store across multiple machines;
(2) divide processing across multiple machine

Challenges

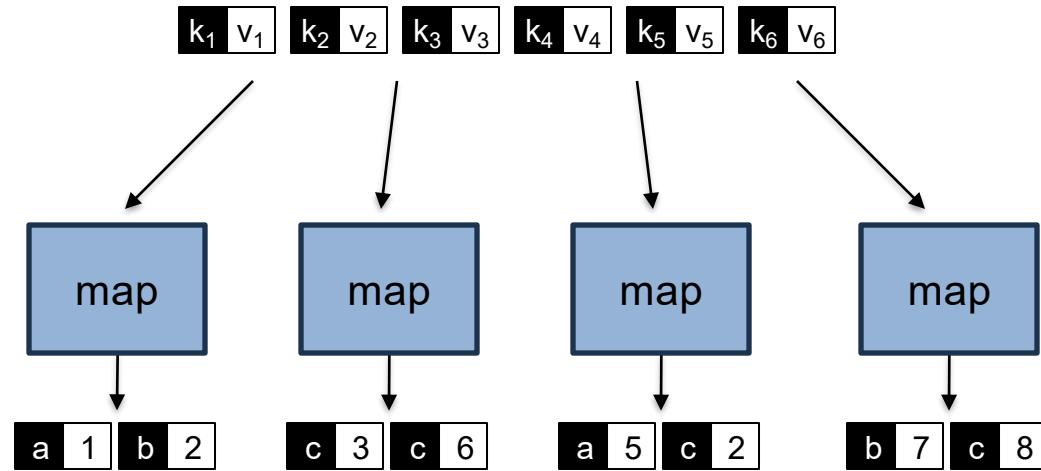
How do we divide data?

Where do we place data? (mapping data to machines)

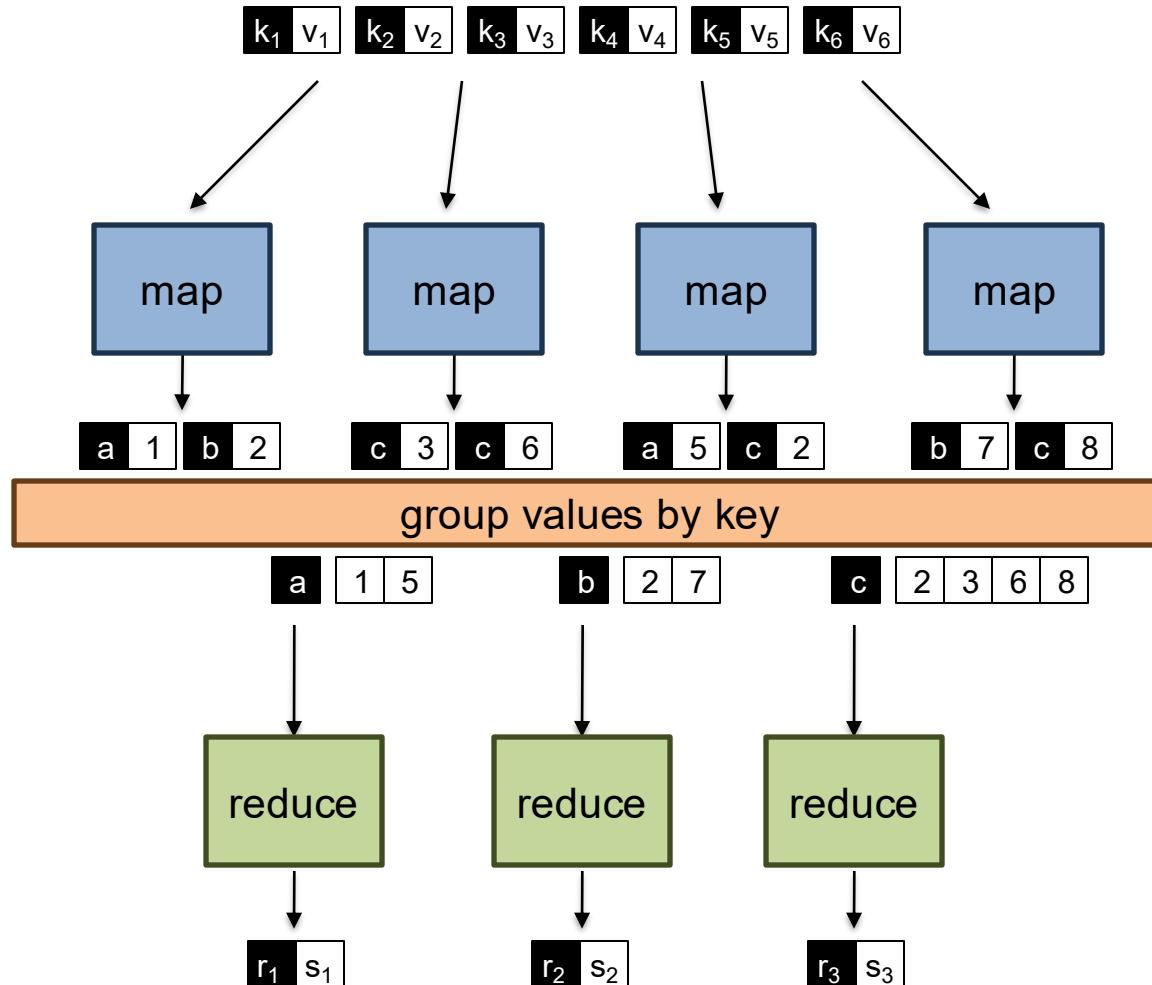
Where do we place workers? (mapping workers to machines)

How do workers access data? (mapping data to workers)

(Who's “we”, btw?)



Okay, now what?



Partition

tl;dr – (1) divide data and store across multiple machines;
(2) divide processing across multiple machine

Challenges

How do we divide data?

Where do we place data? (mapping data to machines)

Where do we place workers? (mapping workers to machines)

How do workers access data? (mapping data to workers)

How do we share intermediate results?

Replicate

tl;dr – keep multiple copies for fault tolerance

Challenges

How many copies do we keep?

Where do we keep them?

How do we keep all the copies in sync?

Which copy do we process?

(caching as a special case)

Orchestration

tl;dr – we need to coordinate *all of this*

Challenges

How do we do *all of this* in the presence of unreliable components?

How do we do *all of this* keeping every machine busy?

When workers die? *unpredictable*

When workers finish? *unpredictable*

When workers interrupt each other? *unpredictable*

When workers access resources? *unpredictable*

CAP Theorem

Consistency

Availability

Partition Tolerance

Choose two!

In practical terms, if we have a network partition (P):

What do we do?

Choose A (AP system)

What happens to C?

Choose C (CP system)

What happens to A?

Data-Intensive Distributed Processing

How do you *actually* do it?

Hadoop provides one answer...



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

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

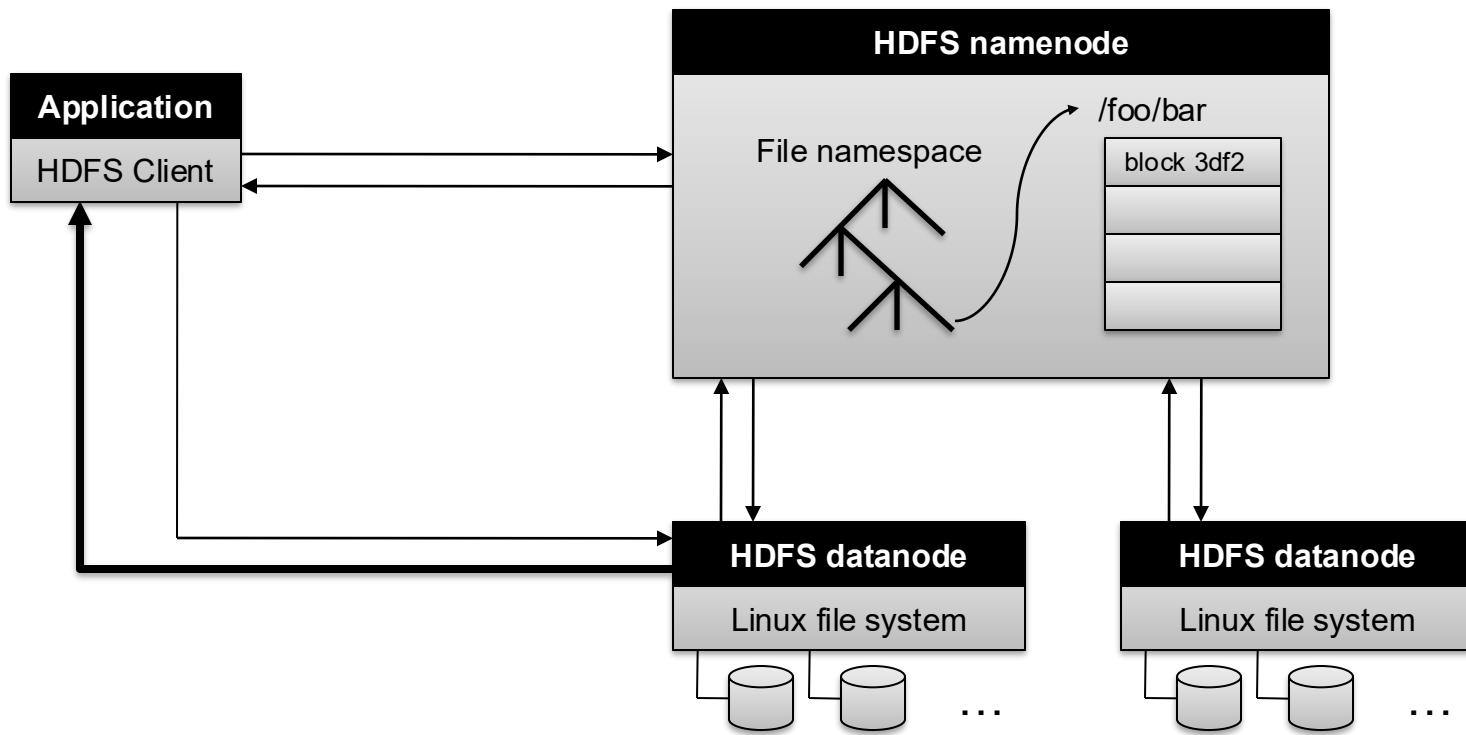
Where do we place the data?

Trick #1: Partition

Trick #2: Replicate

Remember: There are no solutions, only tradeoffs!

HDFS Architecture



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

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

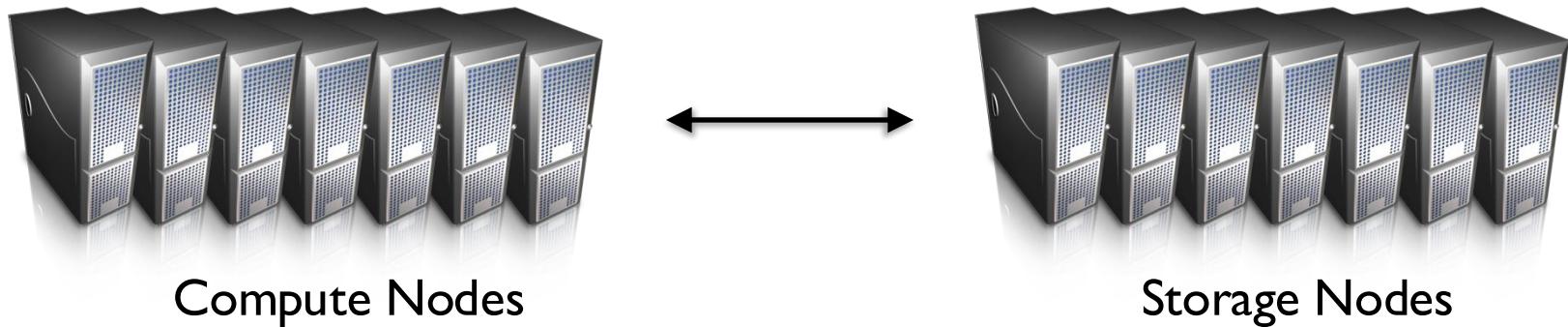
Where do we place the compute?

Trick #1: Partition

Trick #2: Replicate

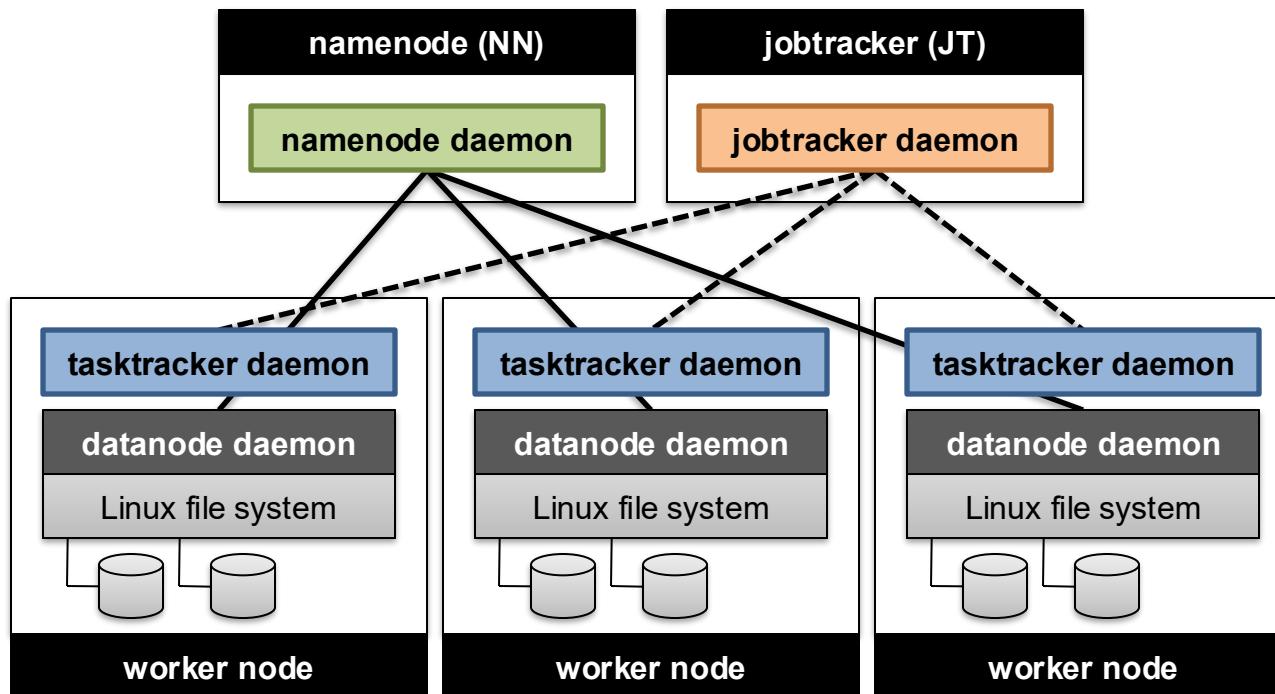
Remember: There are no solutions, only tradeoffs!

Compute meets Data!



Move data to compute?
Move compute to data?

Putting everything together...

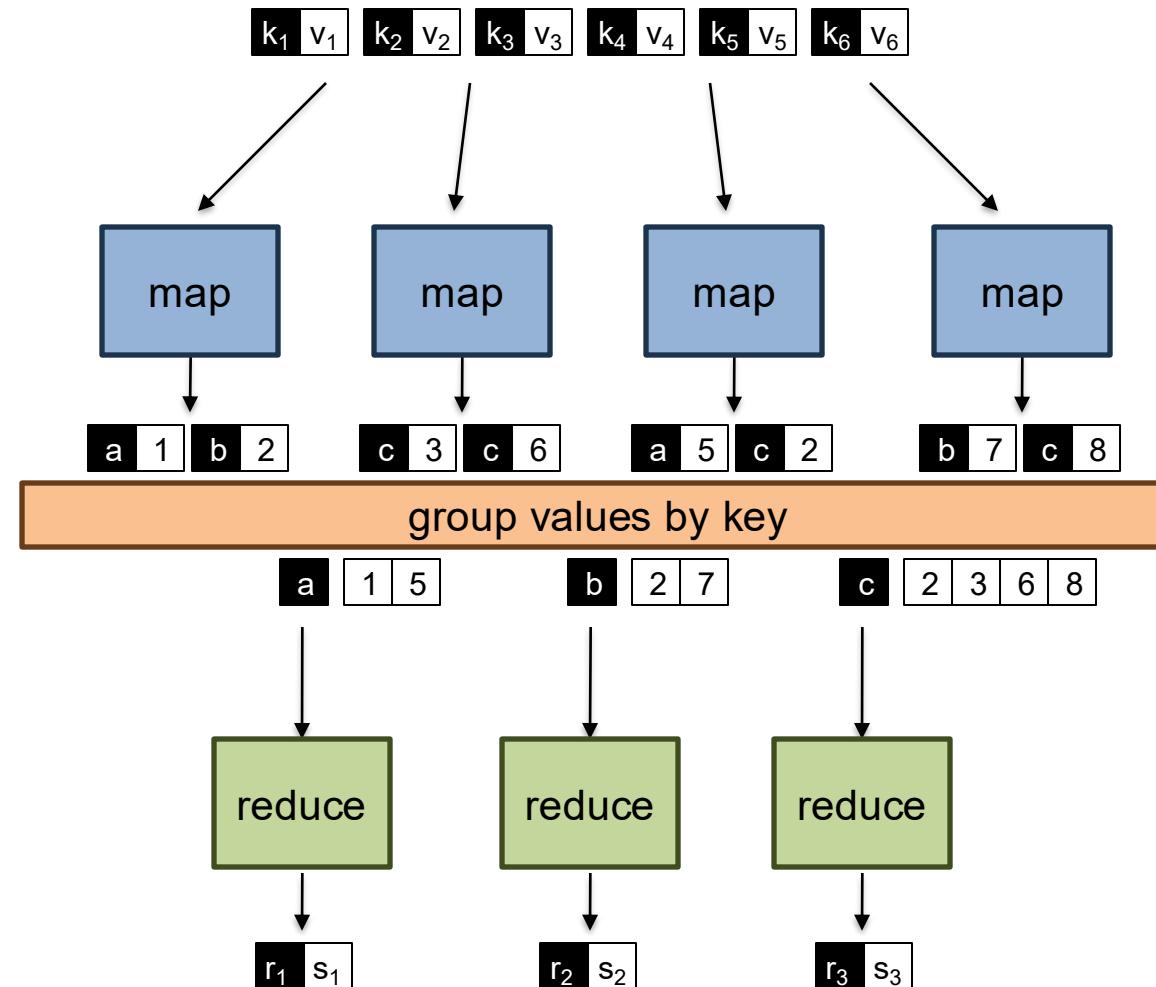


“Hello World” MapReduce: Word Count

```
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)
}
```

Logical View



Physical View

