

MapReduce & Spark

Hanspeter Pfister
pfister@seas.harvard.edu

Big Data

Big Data

- Facebook's daily logs: 60 TB
- 1,000 genomes project: 200 TB
- Google web index: 10+ PB
- Cost of 1 TB of disk: ~\$35
- Time to read 1 TB from disk: 3 hours ! (100 MB/s)"

Big Data

1. Need to build summaries or simple predictive models from a large amount of data

Can sample data and run in memory

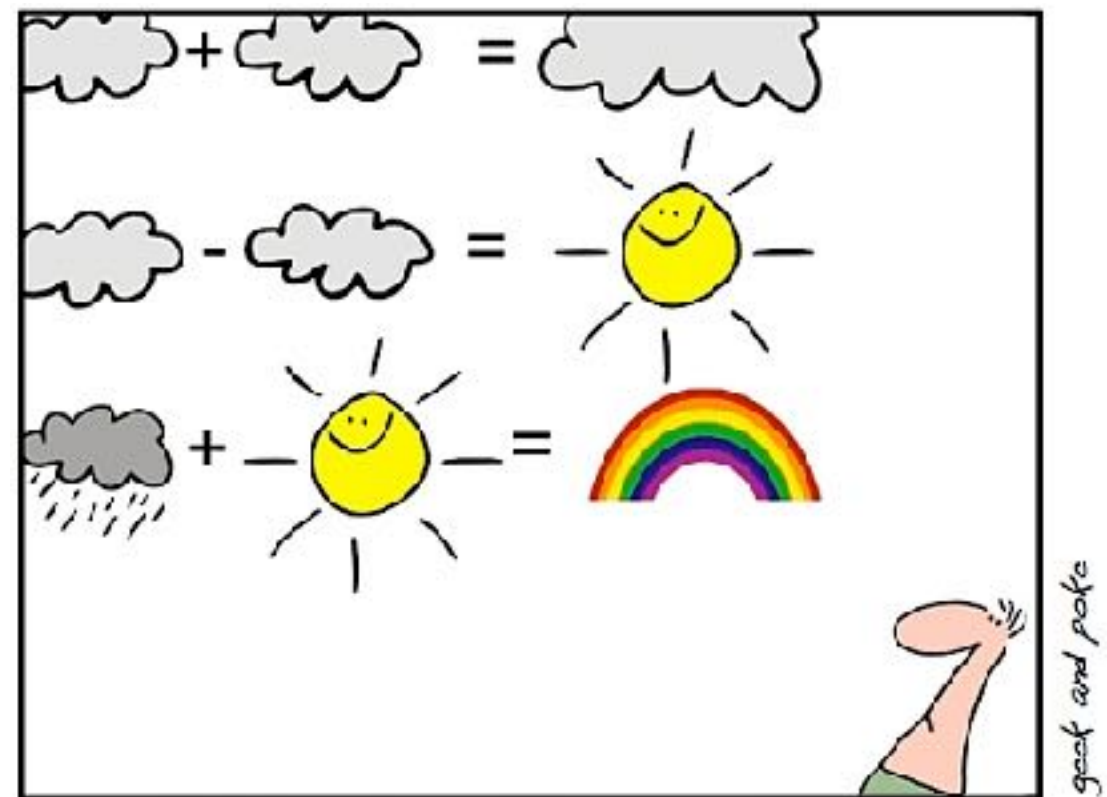
2. Need to apply a model or transformation to a lot of data

Can distribute work to a cluster

3. Need to compute something involving all the relations between the data

Need parallel message-passing algorithms (hard!)

Cloud Computing



*SIMPLY EXPLAINED - PART 17:
CLOUD COMPUTING*

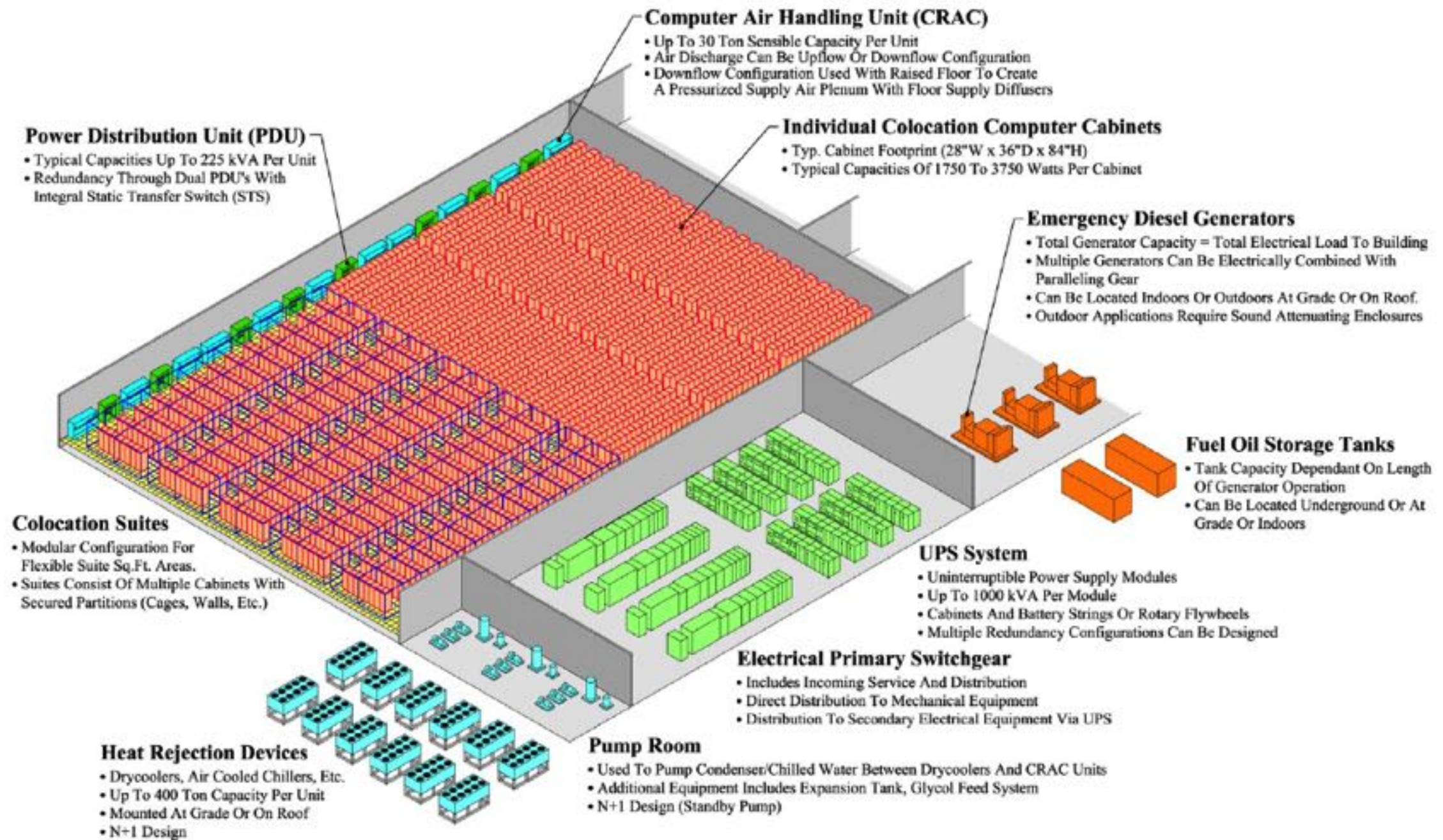
Google Data Center



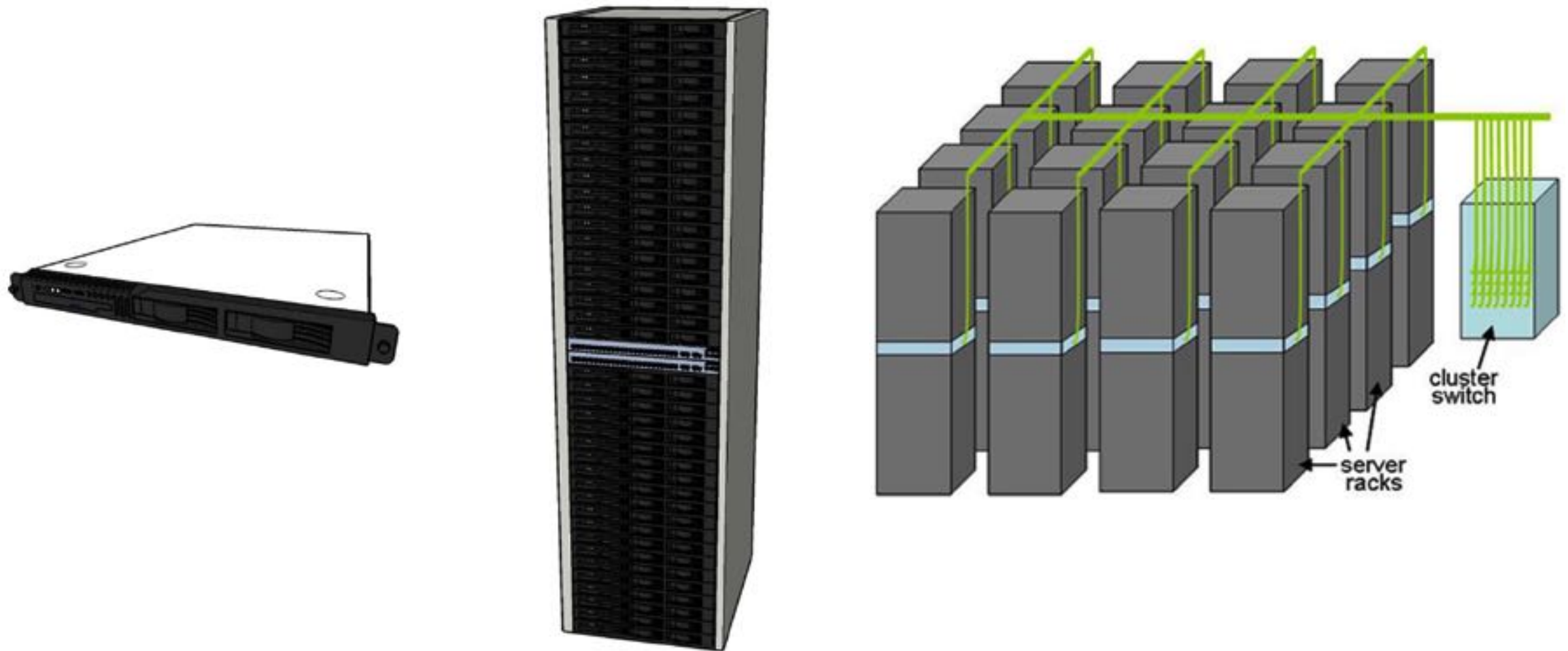


Source: Bonneville Power Administration

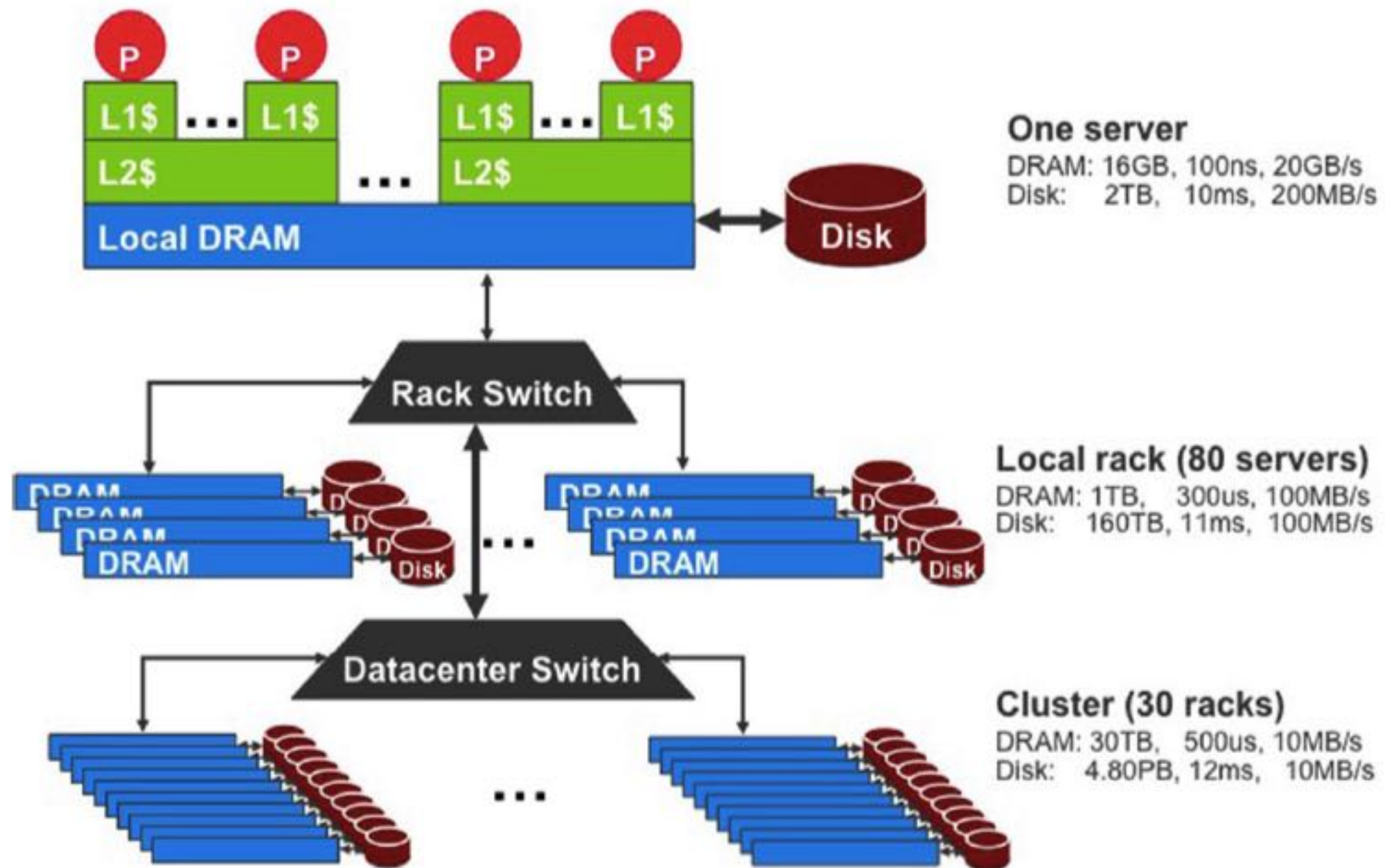
Data Center Anatomy



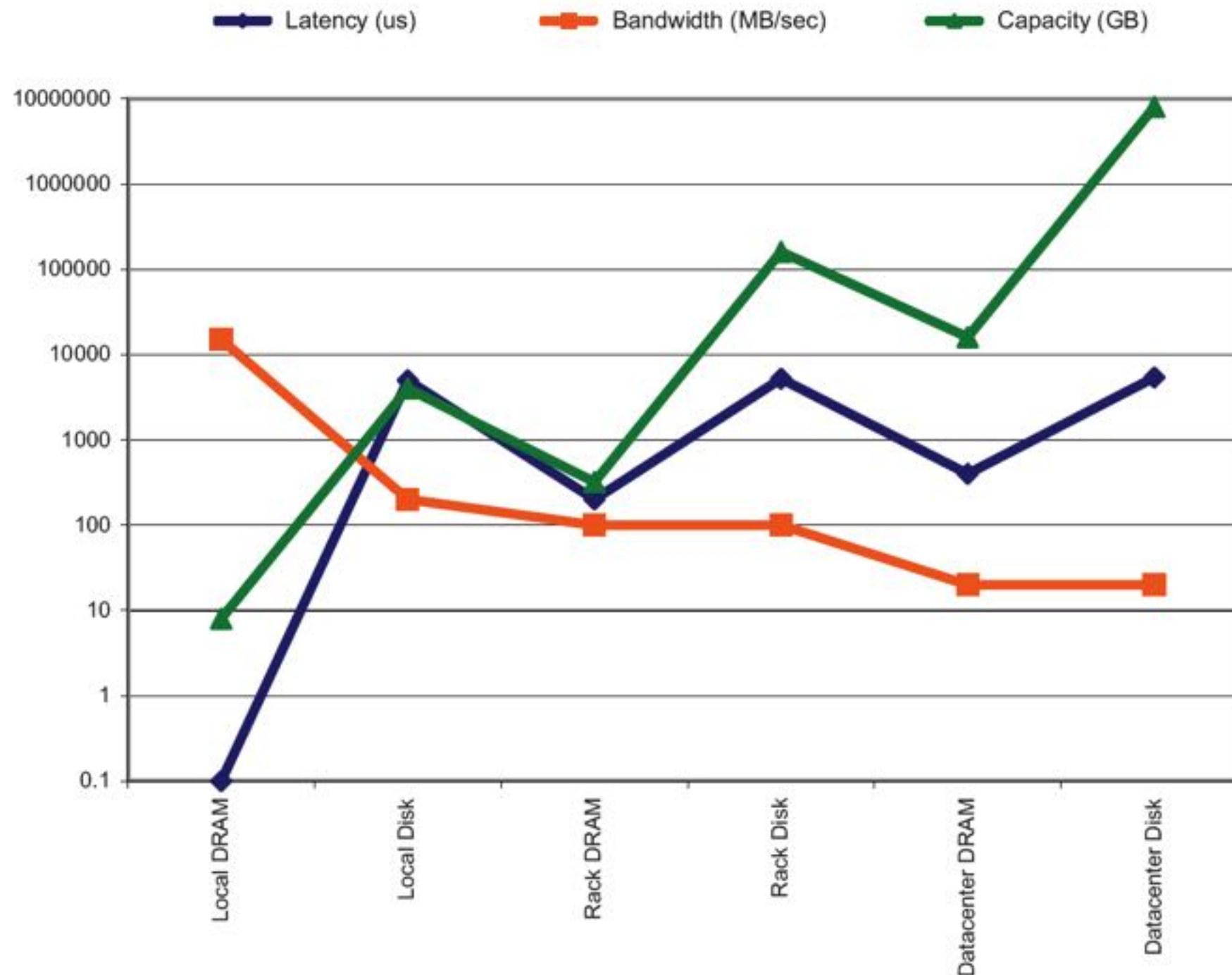
Building Blocks



Storage Hierarchy



Storage Hierarchy



Big Ideas

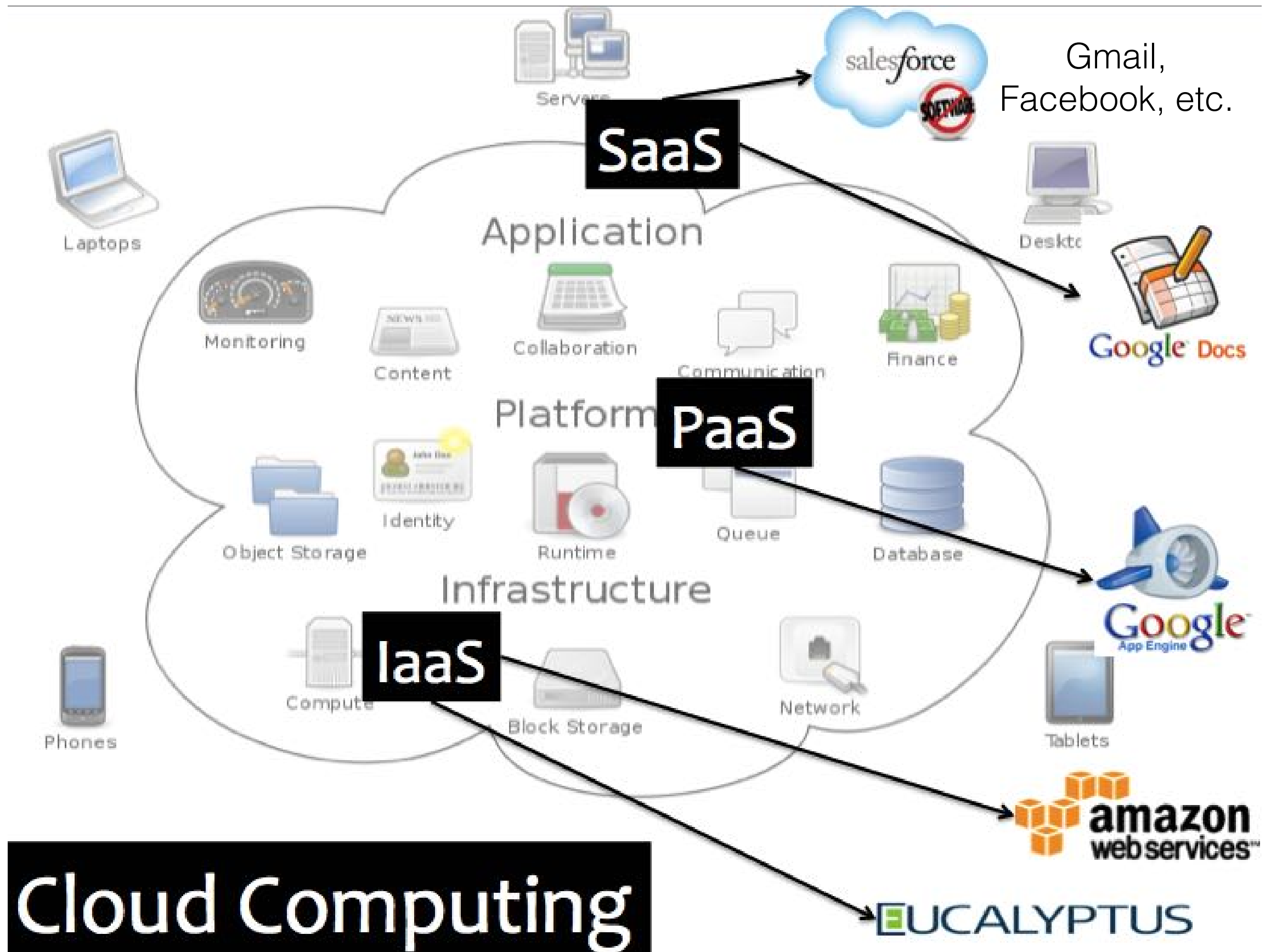
- Scale “out”, not “up”
- Move processing to the data
- Process data sequentially, avoid random access
 - Seeks are expensive, disk throughput is reasonable
- Seamless scalability

Cloud Computing

- A new business model!
- Main characteristics:
 - On-demand self-service
 - Broad network access
 - Resource pooling
 - Rapid elasticity

Everything is a Service

- Infrastructure as a Service (IaaS)
 - Why buy machines when you can rent cycles
- Platform as a Service (PaaS)
 - Give me a nice API and take care of the maintenance, upgrades, etc.
- Software as a Service (SaaS)
 - Just run it for me!





Amazon Web Services

S3, EC2, EMR

- Amazon Simple Storage Service (S3)
 - Let's you put data in the cloud that is addressed by a URL
 - Can be private or public
- Amazon Elastic Cloud Compute (EC2)
 - Let's you pick the size and number of computers in the cloud
- Amazon Elastic Map Reduce (EMR)
 - Automatically builds a Hadoop cluster on top of EC2, feeds the data from S3, saves results in S3, then removes the cluster and frees up the EC2 systems

S3 Pricing

<https://aws.amazon.com/s3/pricing/>

Storage Pricing (varies by region)

Region: US East (Ohio)

	Standard Storage	Standard - Infrequent Access Storage †	Glacier Storage
First 50 TB / month	\$0.023 per GB	\$0.0125 per GB	\$0.004 per GB
Next 450 TB / month	\$0.022 per GB	\$0.0125 per GB	\$0.004 per GB
Over 500 TB / month	\$0.021 per GB	\$0.0125 per GB	\$0.004 per GB

Data Transfer OUT From Amazon S3 To Internet

First 1 GB / month	\$0.000 per GB
Up to 10 TB / month	\$0.090 per GB
Next 40 TB / month	\$0.085 per GB
Next 100 TB / month	\$0.070 per GB
Next 350 TB / month	\$0.050 per GB
Next 524 TB / month	Contact Us

EC2 Pricing

<https://aws.amazon.com/ec2/pricing/on-demand/>

Linux	RHEL	SLES	Windows	Windows with SQL Standard	Windows with SQL Web
Windows with SQL Enterprise					
Region: US West (Oregon) ▾					
	vCPU	ECU	Memory (GiB)	Instance Storage (GB)	Linux/UNIX Usage
GPU Instances - Current Generation					
p2.xlarge	4	12	61	EBS Only	\$0.9 per Hour
p2.8xlarge	32	94	488	EBS Only	\$7.2 per Hour
p2.16xlarge	64	188	732	EBS Only	\$14.4 per Hour
g2.2xlarge	8	26	15	60 SSD	\$0.65 per Hour
g2.8xlarge	32	104	60	2 x 120 SSD	\$2.6 per Hour

EC2 Instance Types

<http://aws.amazon.com/ec2/instance-types/>

Accelerated Computing Instances

P2

P2 instances are intended for general-purpose GPU compute applications.

Features:

- High Frequency Intel Xeon E5-2686v4 (Broadwell) Processors
- High-performance NVIDIA K80 GPUs, each with 2,496 parallel processing cores and 12GiB of GPU memory
- Supports GPUDirect™ (peer-to-peer GPU communication)
- Provides [Enhanced Networking](#) using the Amazon EC2 Elastic Network Adaptor with up to 20Gbps of aggregate network bandwidth within a Placement Group
- EBS-optimized by default at no additional cost

Model	GPUs	vCPU	Mem (GiB)	GPU Memory (GiB)
p2.xlarge	1	4	61	12
p2.8xlarge	8	32	488	96
p2.16xlarge	16	64	732	192

Use Cases

Machine learning, high performance databases, computational fluid dynamics, computational finance, seismic analysis, molecular modeling, genomics, rendering, and other server-side GPU compute workloads.

What is a GPU?

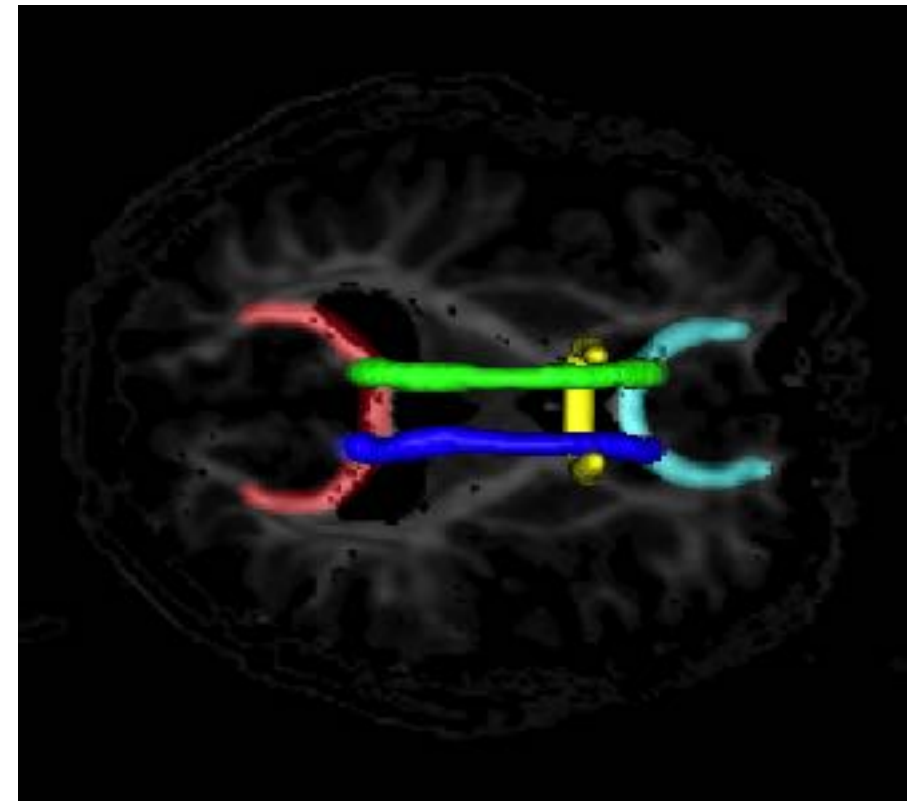
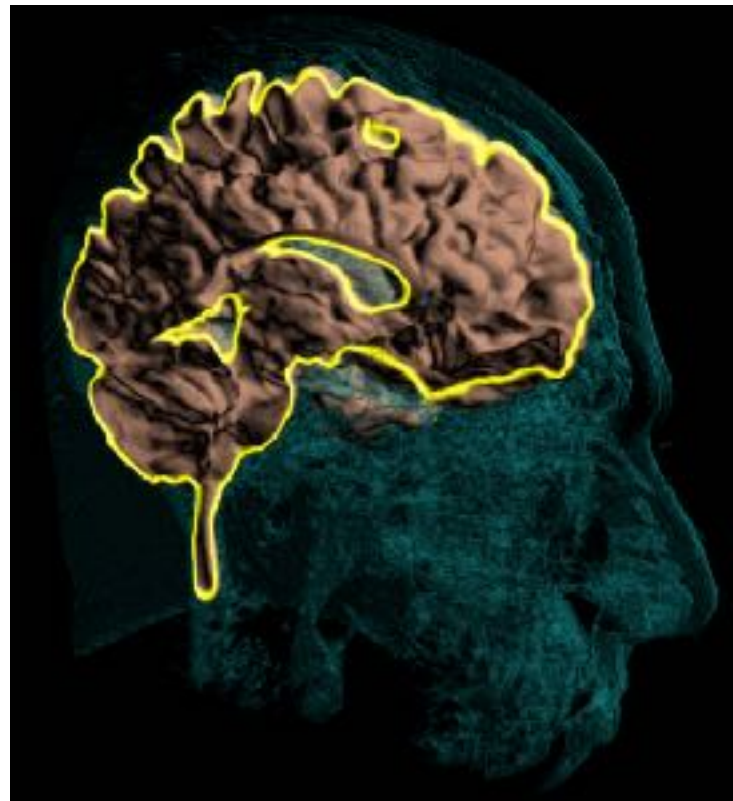
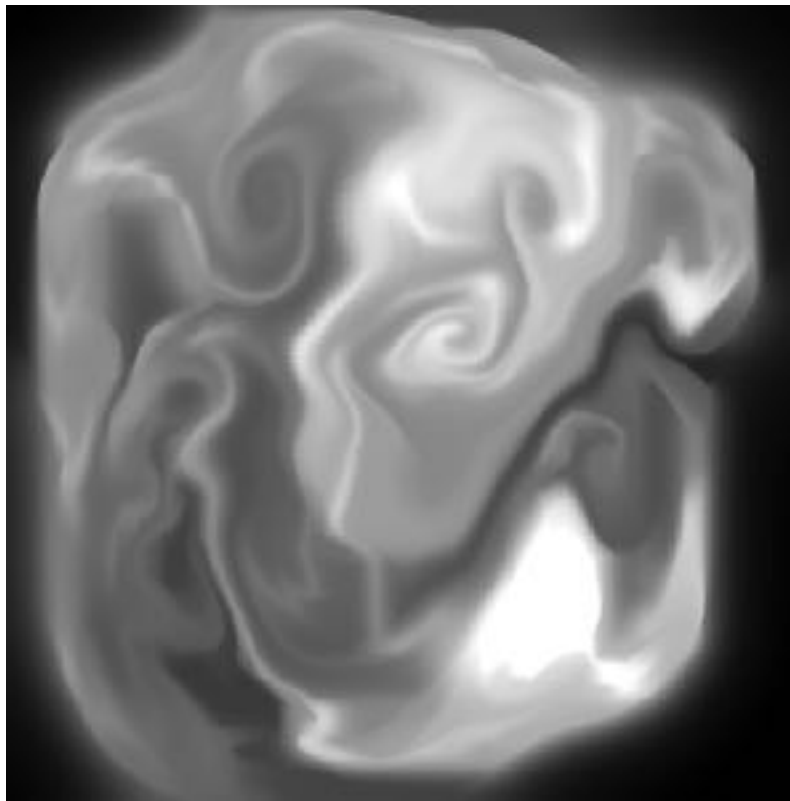
- PC hardware dedicated for 3D graphics
- Performance pushed by game industry



NVIDIA SLI System

GPGPU

- General Purpose computation on the GPU
- Mapping computational problems to graphics rendering pipeline



Why GPU for computing?

- Fast and massively parallel
- High memory bandwidth
- Programmable (NVIDIA CUDA, DirectX Compute Shader, OpenCL)
- Inexpensive supercomputers

Deep Learning Tests

- David Wihl's unscientific tests of TensorFlow on different platforms

```
python -m tensorflow.models.image.mnist.convolutional
```

	TensorFlow Version	# CPU Cores	Memory	GPU	Elapsed time (h:m:s)
David's Custom Linux box	0.10.0rc0	12	32 GB	GTX1080	0:01:00
AWS p2.xlarge	0.12.0	4	61 GB	Tesla K80	0:01:44
MacBook Pro 2016	0.12.0	4	16 GB	0	0:23:44
AWS Microinstance	Python 3 0.12.0	1	1 GB	0	4:37:24

Pitfalls

- EC2 do not persist! Once the instance shuts down everything is lost! Store persistent data on S3 or export to your local machine
- If you leave your instance running, even if it does not compute anything, it will cost you money! Make sure to SHUT DOWN all instances before you log out
- If you run over the \$100 credit, your credit cards will be charged. Do not despair! We will be able to fix things with Amazon (but it's painful!)

Amazon Data Center

How do we program this thing?



Parallel Computing

Estimating Pi

Estimate Pi by throwing darts at a square

Calculate percentage that fall in the unit circle

$$\text{Area of square} = r^2 = 1$$

$$\text{Area of circle quadrant} = \frac{1}{4} * \pi r^2 = \pi/4$$

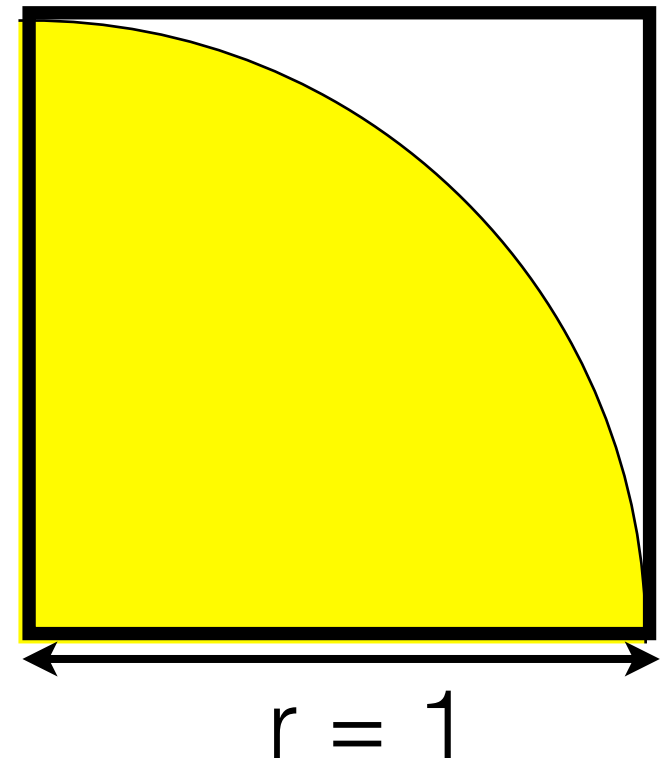
Randomly throw darts at x,y positions

If $x^2 + y^2 \leq 1$, then point is inside circle

Compute ratio:

points inside / # points total

$$\pi = 4 * \text{ratio}$$



Computing Pi in Python

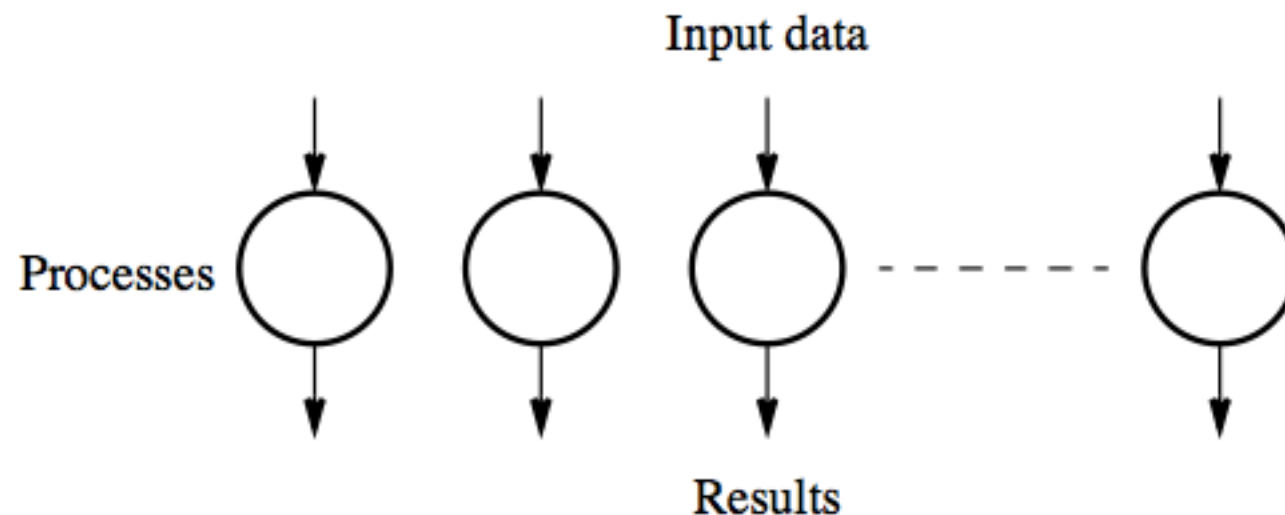
```
1  # import random number module
2  import random
3  # import the math module for comparison
4  import math
5
6  # counters (want them to be floats)
7  inPts = 0.0
8  totPts = 0.0
9
10 # 'throw' a number of darts
11 for i in xrange(100000):
12     # generate two uniform random numbers
13     # between 0 and 1
14     x=random.random()
15     y=random.random()
16
17     # increment counters if necessary
18     totPts += 1
19     if x**2 + y**2 <= 1:
20         inPts += 1
21
22 # area of quarter circle is pi/4
23 print(4 * inPts / totPts)
24 print(math.pi)
```


And now in parallel...

- How is the computation being set up?
- What is each processor doing?
- What needs to be communicated?
- How is the final result computed?

Embarrassingly Parallel

- Computation can be divided into completely independent parts
- Each part executed by a separate processor



Embarrassingly Parallel

- Computer Graphics
- Image Processing
- Monte Carlo Estimations (MCMC)
- Cross-validation
- Grid search
- Bagging

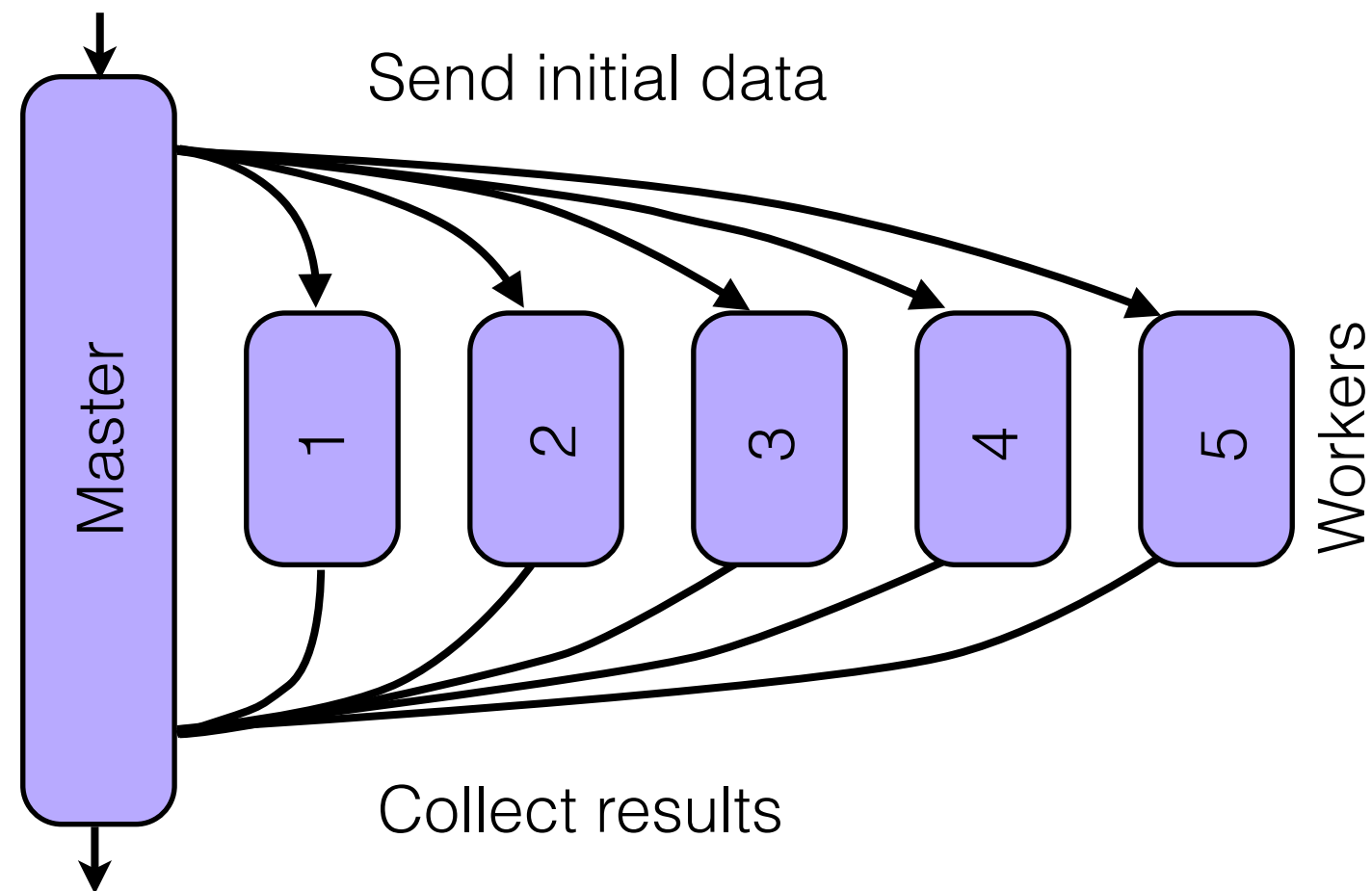


Question

- Which of these applications is NOT embarrassingly parallel?
 - a. Searching for patterns in log files
 - b. Computing recommendations for Amazon
 - c. Fluid flow simulations
 - d. Computing Google's PageRank on the web

Master-Worker Approach

Independent processing, no communication



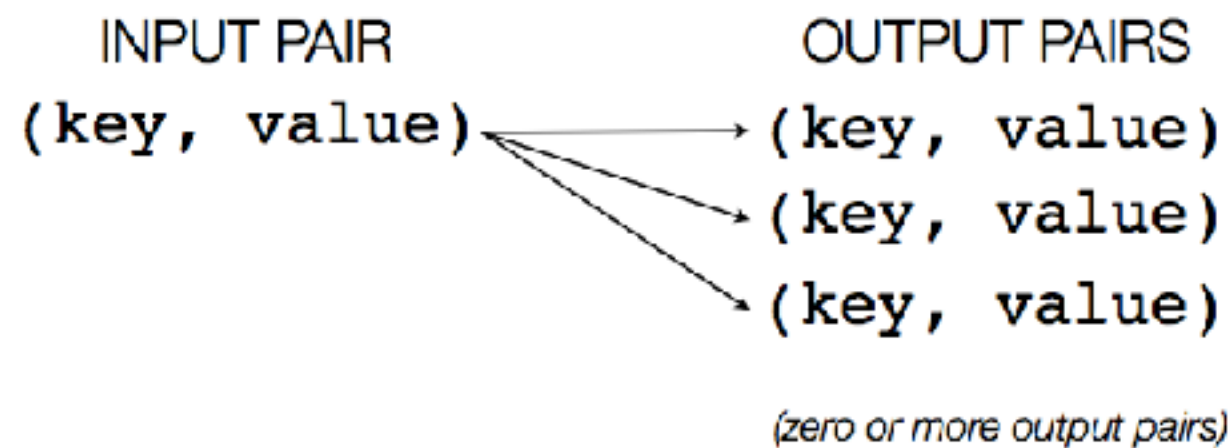
However...

- It is not trivial to run master-worker jobs across many computers
- How do you distribute the data?
- What about task assignments?
- What can go wrong?

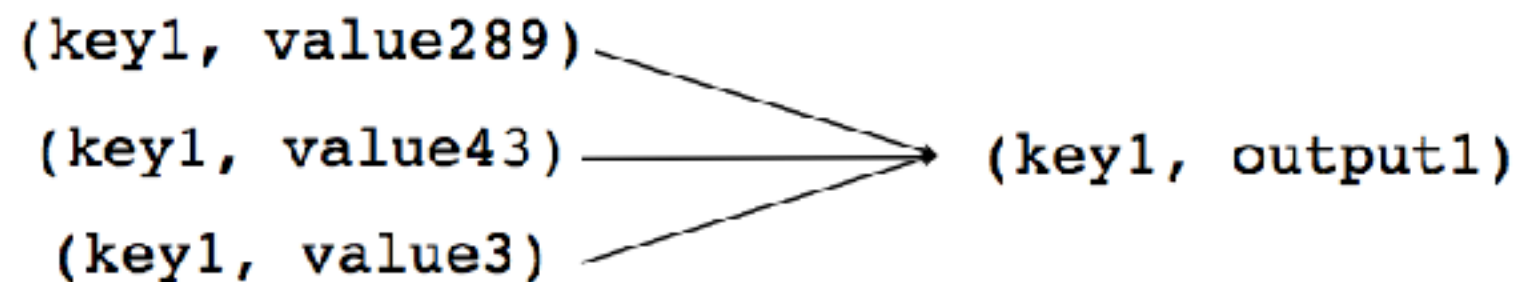
MapReduce

MapReduce

Map: specify operations on key-value pairs



Reduce: combine key-value results



MapReduce Phases

1. Map
2. Shuffle / Sort
3. Reduce

Ex: Word Count

Map:

`(line1, "Hello there.")` → `("hello", 1)`
→ `("there", 1)`

`(line2, "Why, hello.")` → `("why", 1)`
→ `("hello", 1)`

Ex: Word Count

Shuffle / Sort:

`("hello", 1)`
`("hello", 1)`

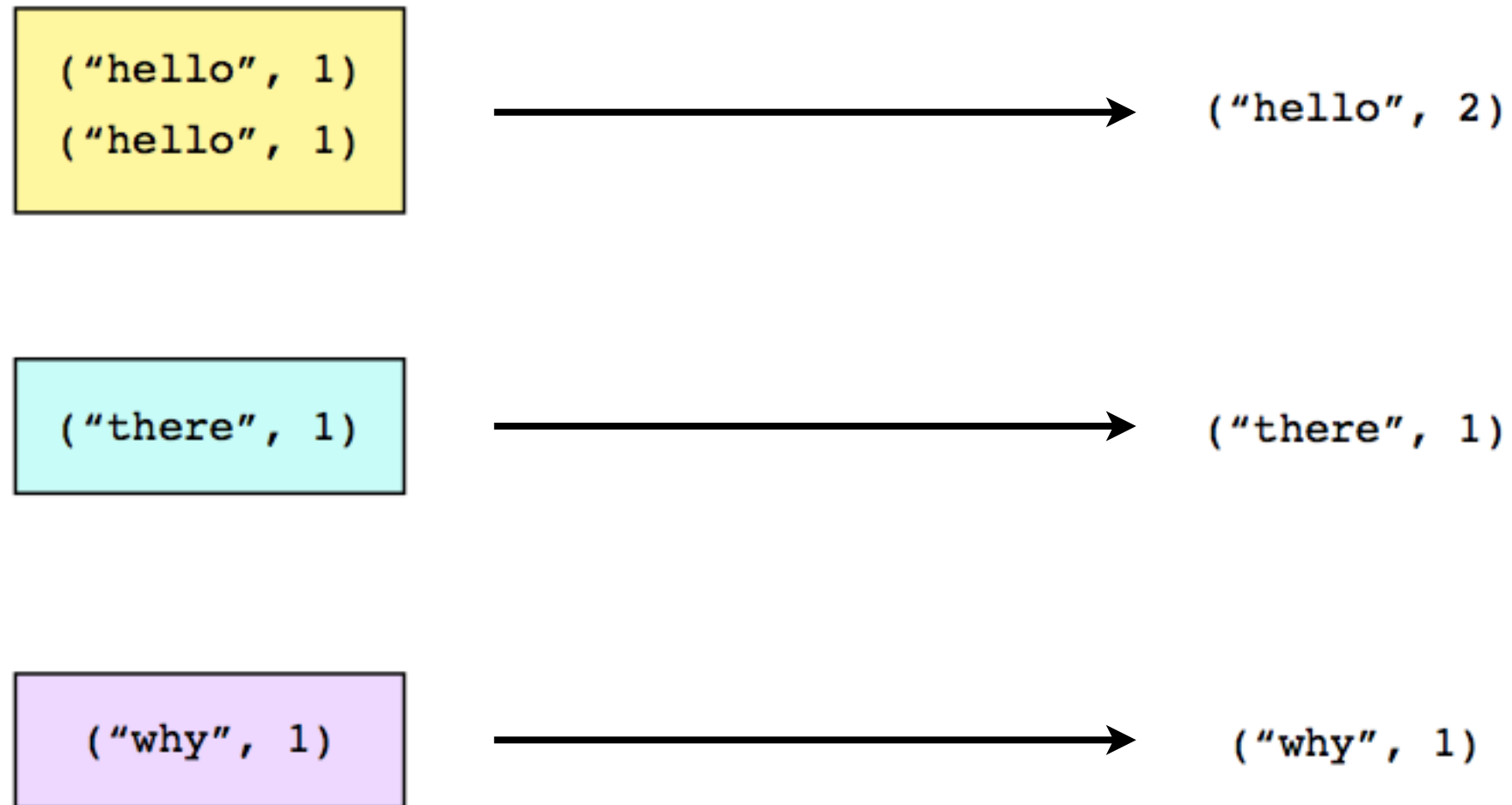
`("there", 1)`

`("why", 1)`

All values with the same key are sent to the same reducer

Ex: Word Count

Reduce:



Code (MrJob)

```
from mrjob.job import MRJob

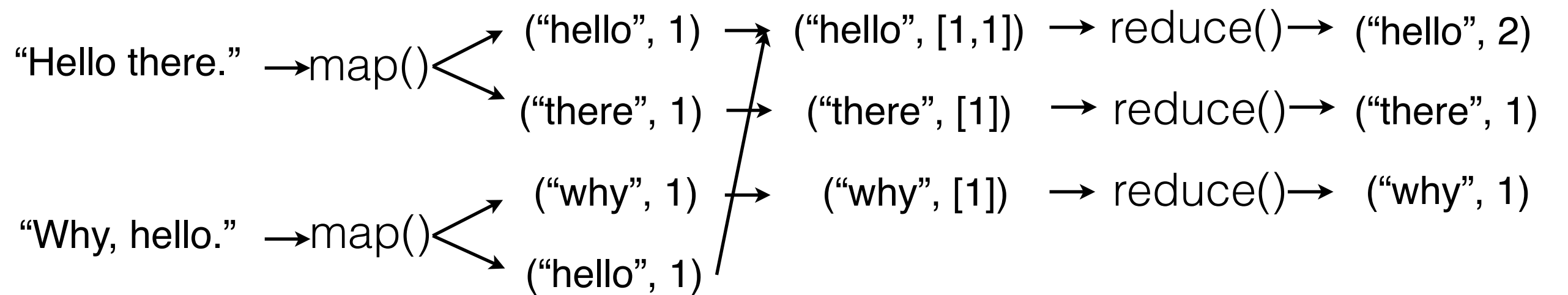
class mrWordCount(MRJob):

    def mapper(self, key, line):
        for word in line.split(' '):
            yield word.lower(), 1

    def reducer(self, word, occurrences):
        yield word, sum(occurrences)

if __name__ == '__main__':
    mrWordCount.run()
```

Logical MR



Input

Map

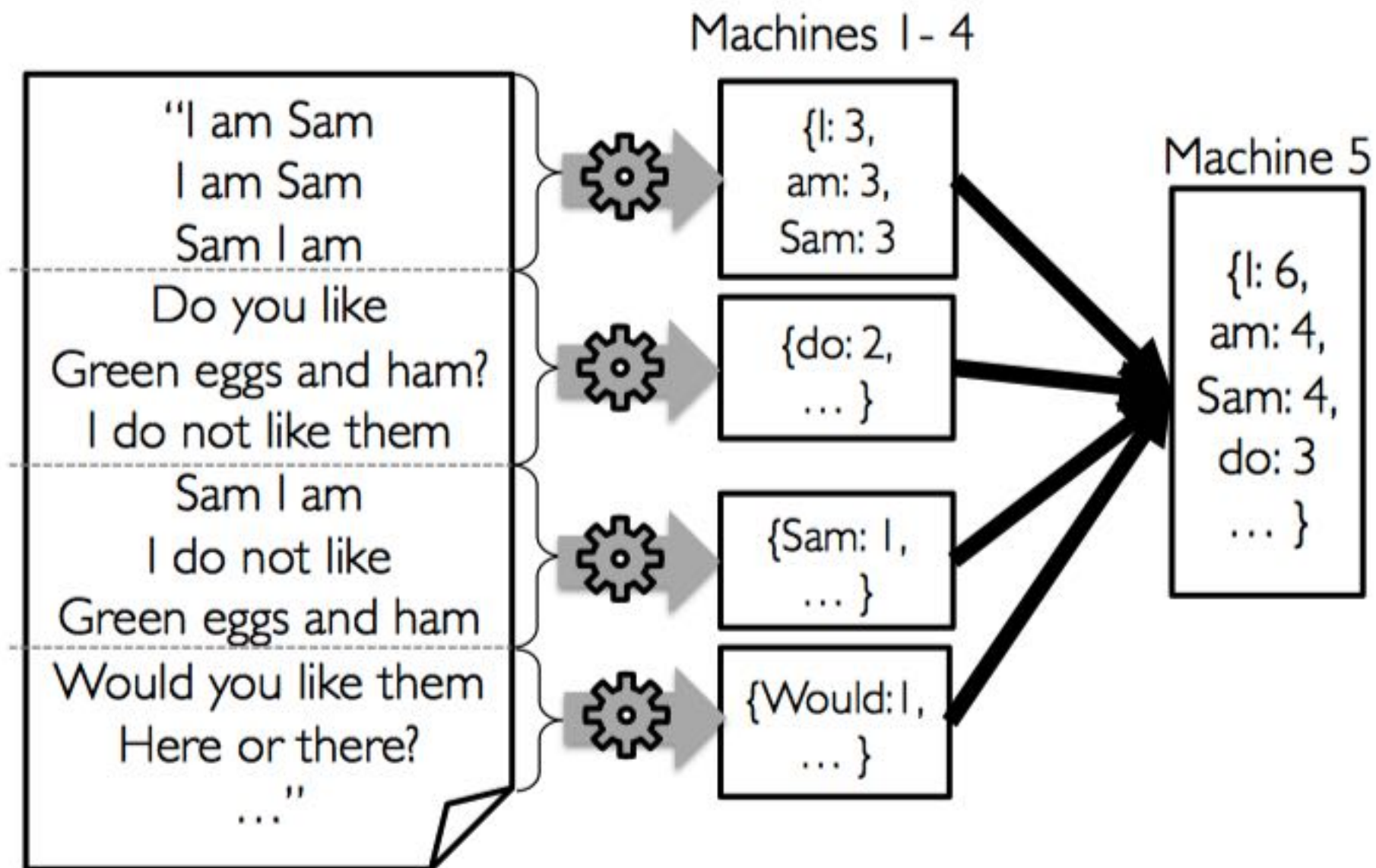
Shuffle / Sort

Reduce

Output

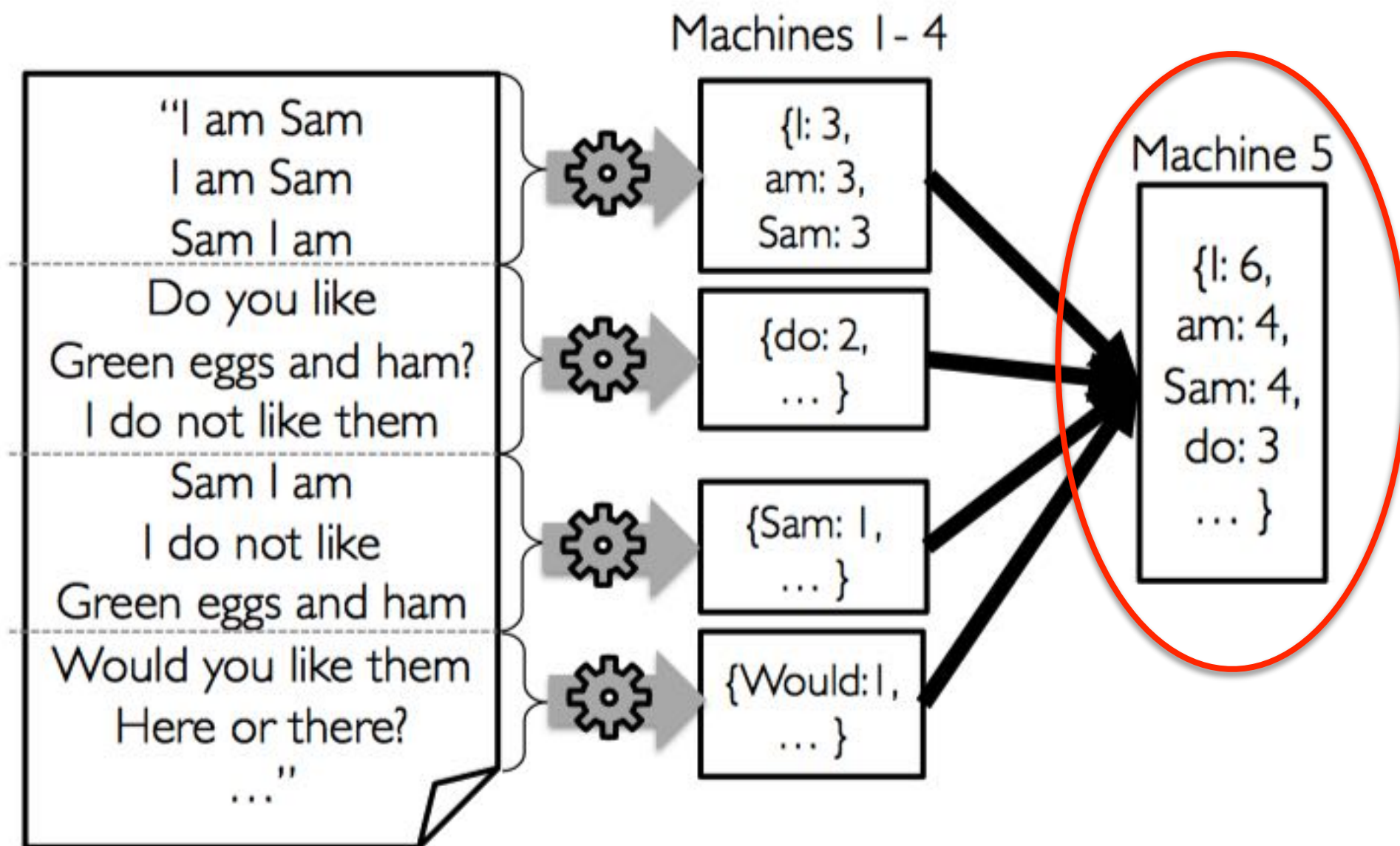
Distributed MR

What is the problem with this approach?



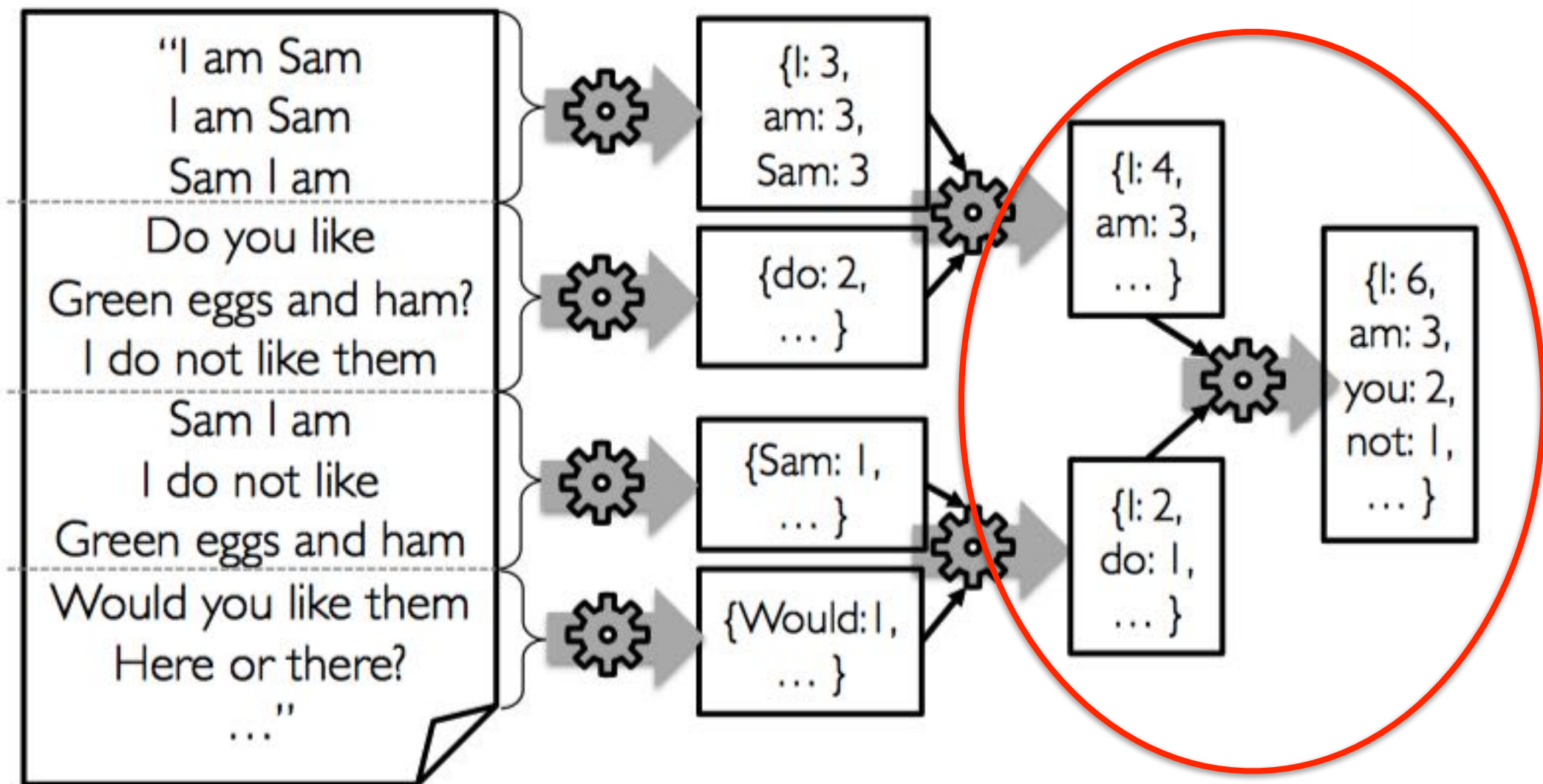
Distributed MR

Results have to fit on one machine!



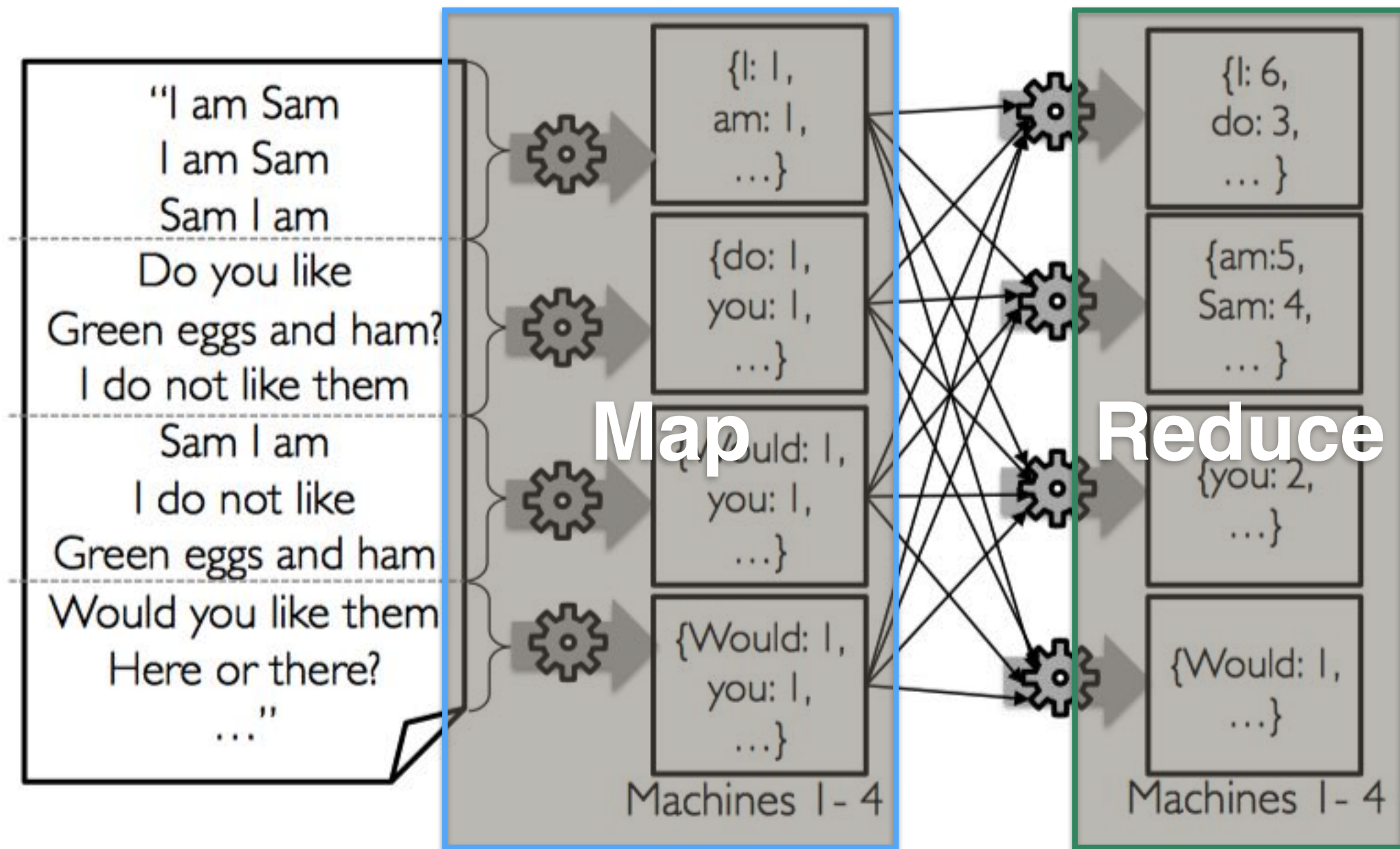
Distributed MR

Can add additional aggregation layers,
but results still need to fit on one machine



Distributed MR

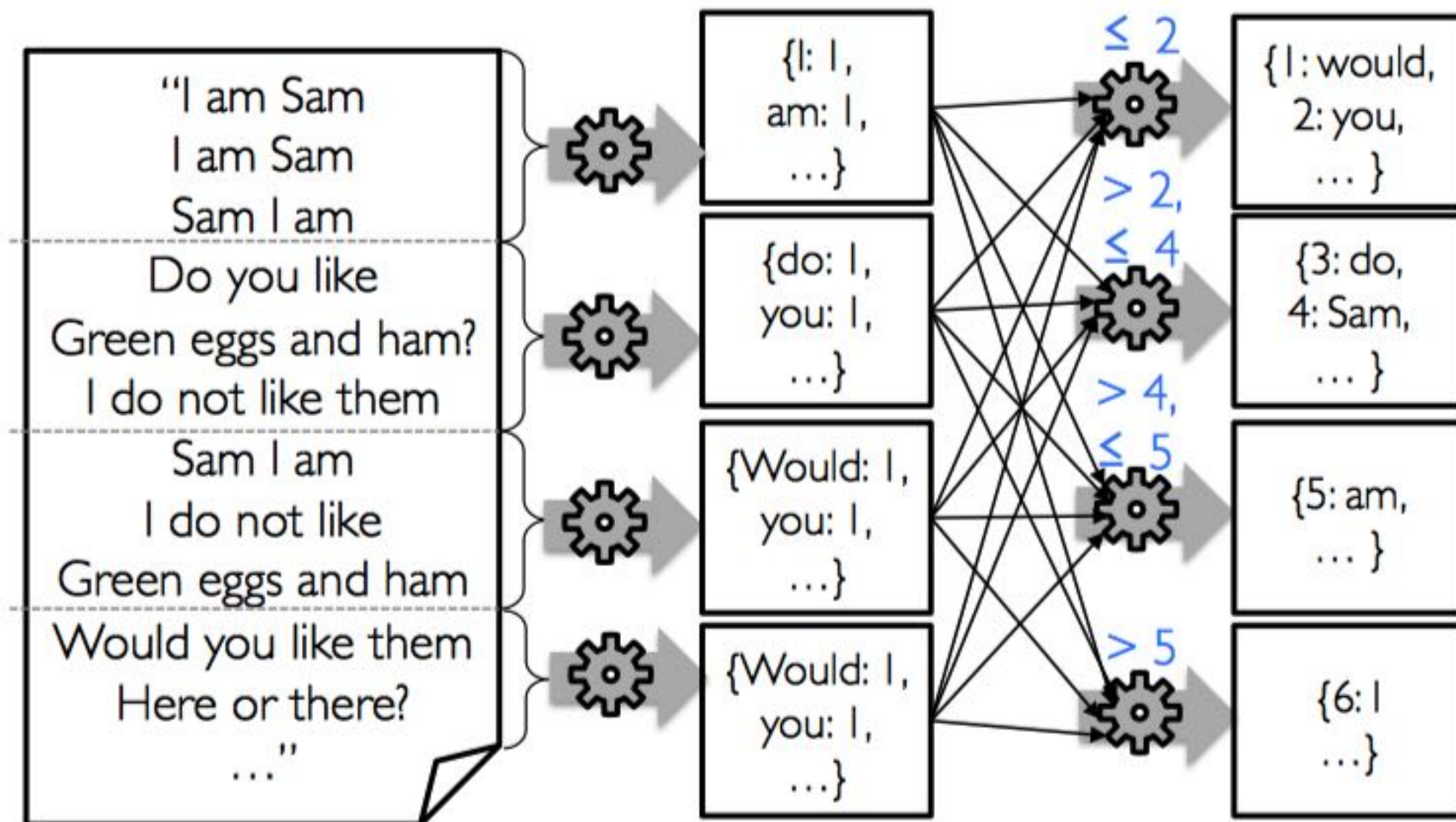
Use divide and conquer!



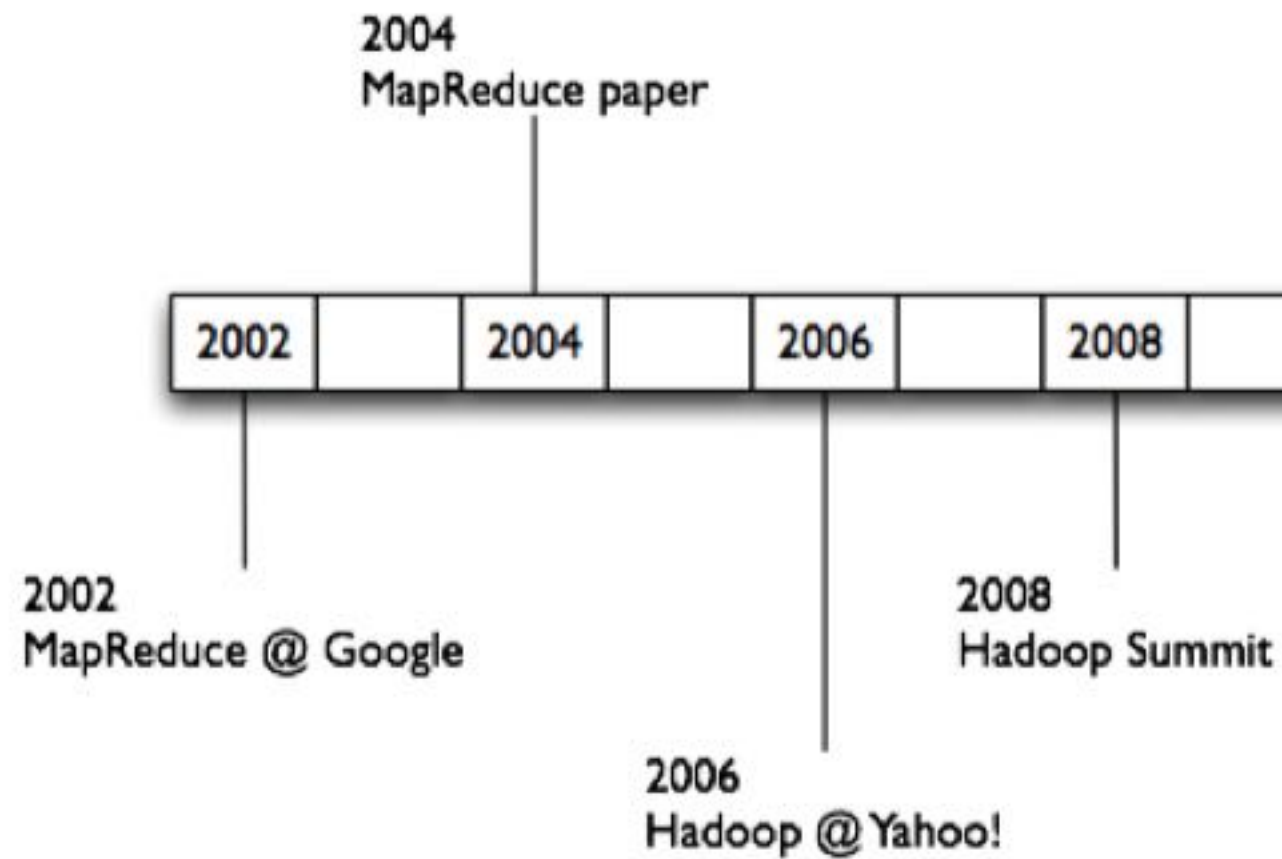
**Shuffle /
Sort**

Distributed MR





Map Reduce for sorting



History





- Most credible open-source MR toolset
- Backed by , 
- Used by ,  and many others
- Amazon's EMR based on Hadoop
- Closely matches Google's MR framework

User To Do List

- Write map() and reduce() functions
- Indicate:
 - Input / output files
 - Number of map (M) / reduce (R) tasks
 - Number of machines (W)
- Requires no knowledge of parallel computing!

You don't know...

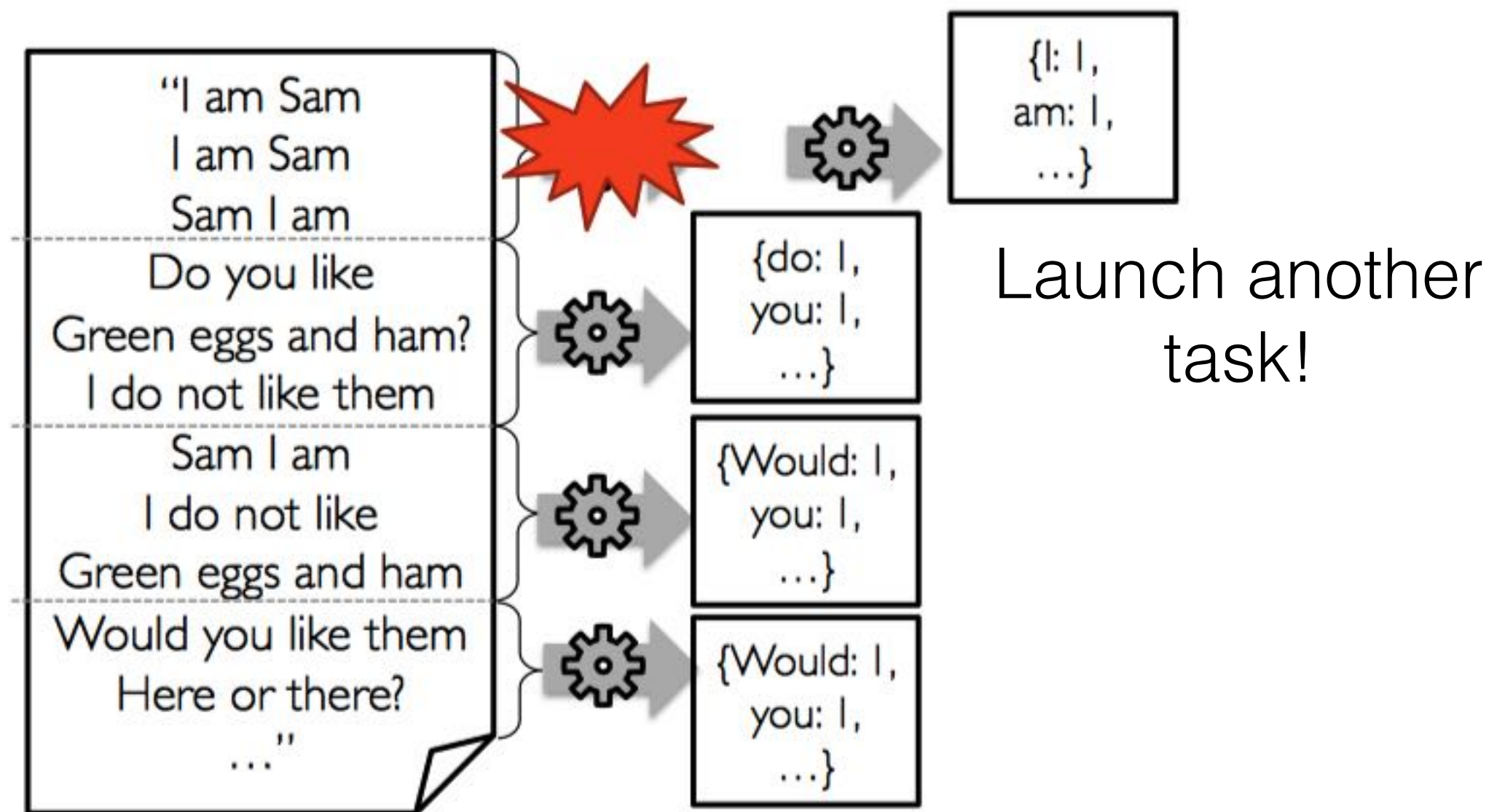
- Where mappers and reducers run
- Where the data is stored
- When a mapper or reducer begins or finishes
- Which input a particular mapper is processing
- Which intermediate key a particular reducer is processing

MR “Runtime”

- Handles “data distribution”
 - Moves processes to data
- Handles scheduling
 - Assign workers to map and reduce tasks
- Handles synchronization
 - Gathers, sorts, and shuffles intermediate data
- Handles errors and faults
 - Detects worker failures and restarts

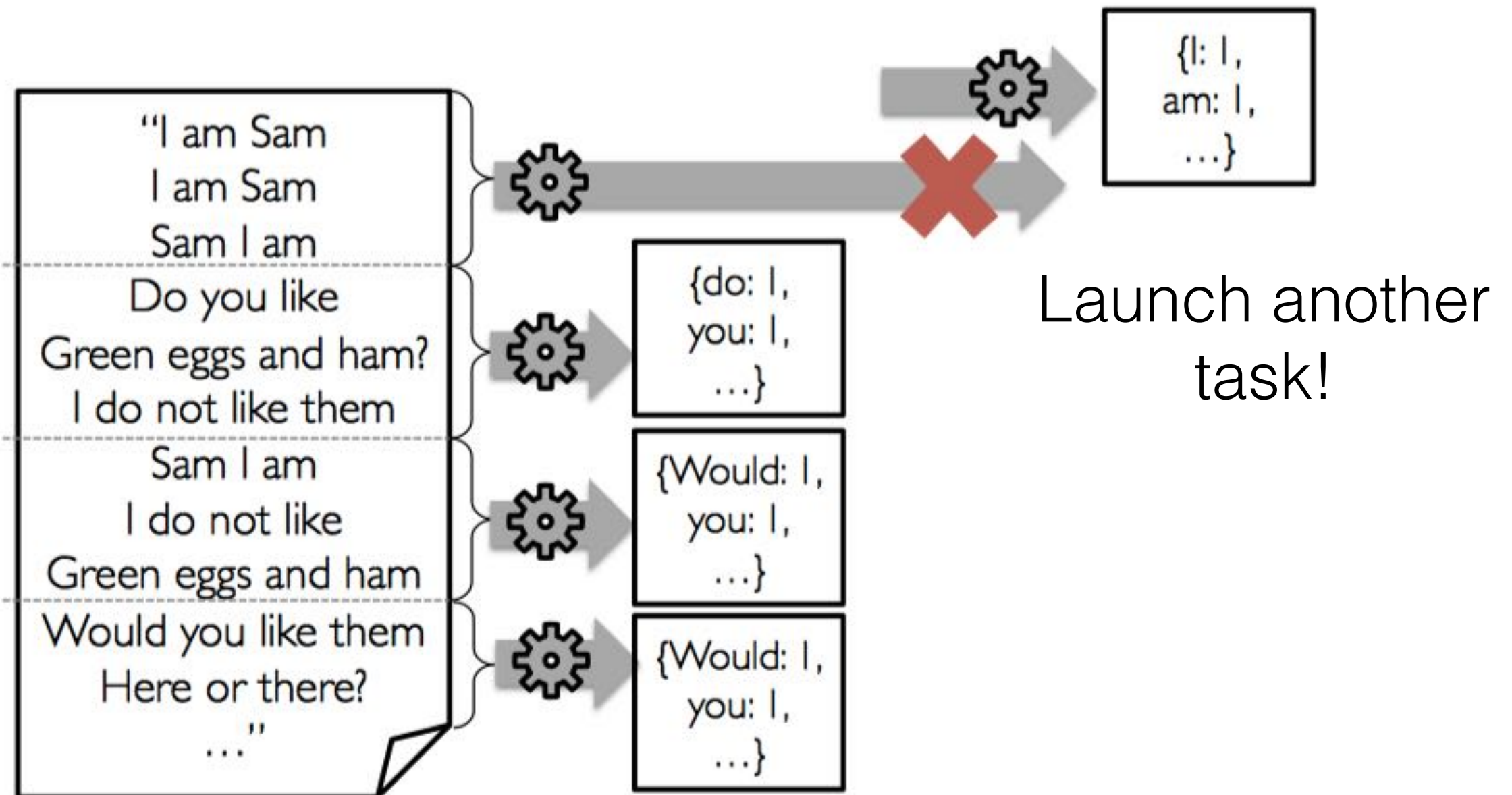
Distributed MR

Dealing with machine failures



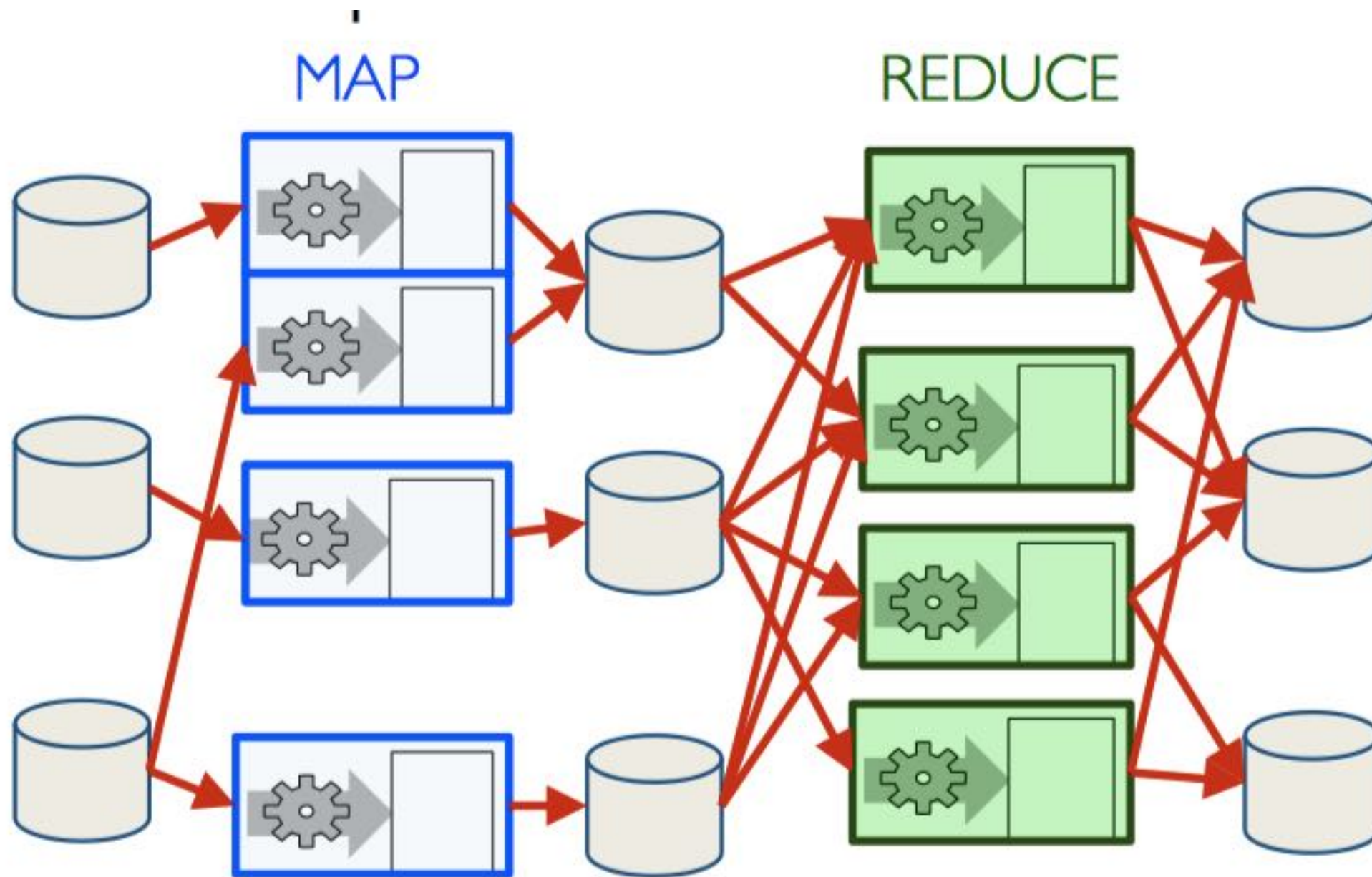
Distributed MR

Dealing with slow tasks



Data Distribution

Each stage passes through hard drives



Data Distribution

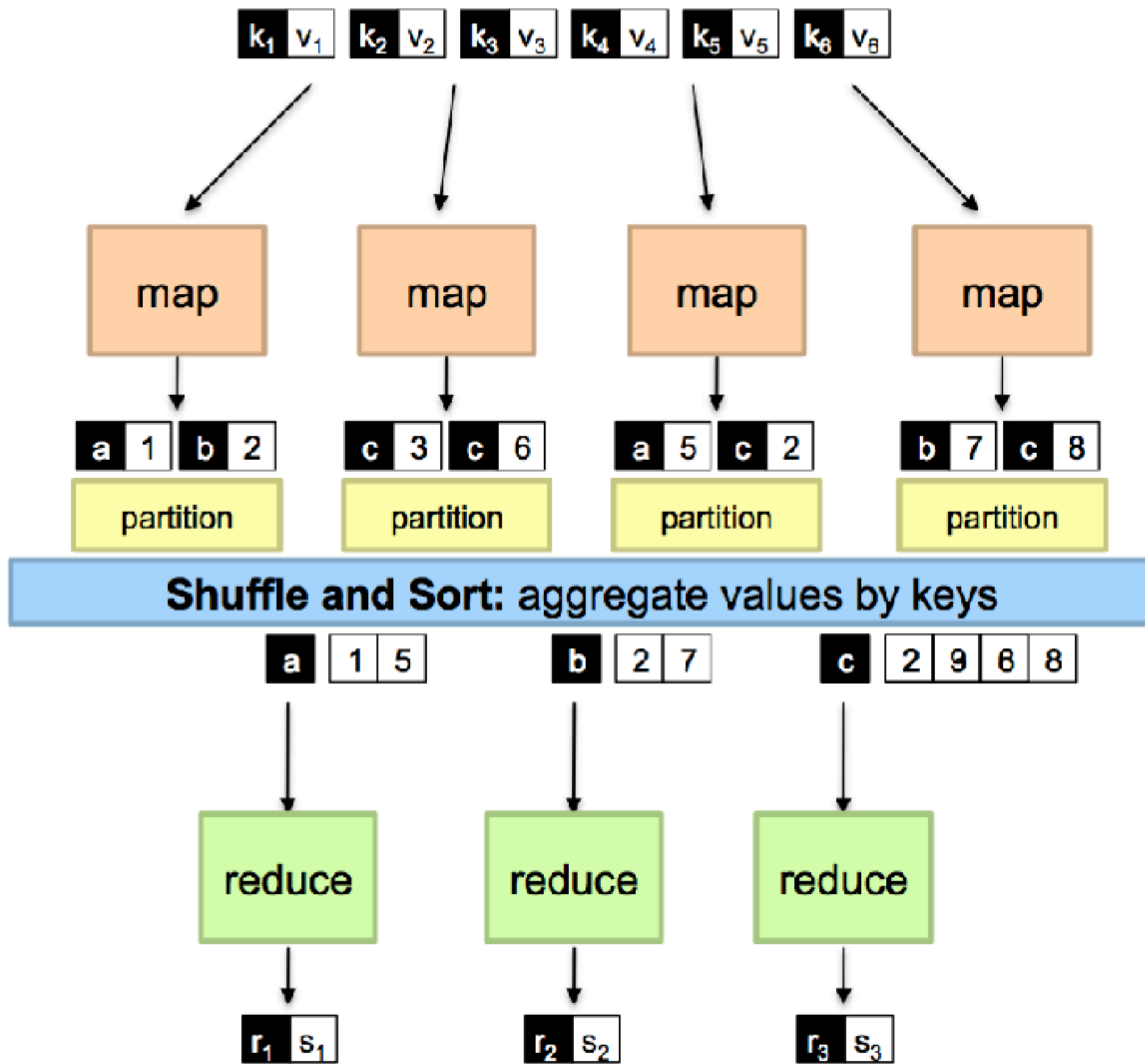
- Input files are split into “chunks” and replicated on a distributed file system
 - e.g., 16 to 64 MB chunks (user defined)
- Intermediate files from map tasks are written to local disk
- Output files are written to distributed a file system

Tasks Scheduling

- Many copies of user program (Master/Workers) are started
- Master finds idle machines and assigns them tasks
- Tries to utilize data localization by running map tasks on machines with data
 - Move the computation to the data

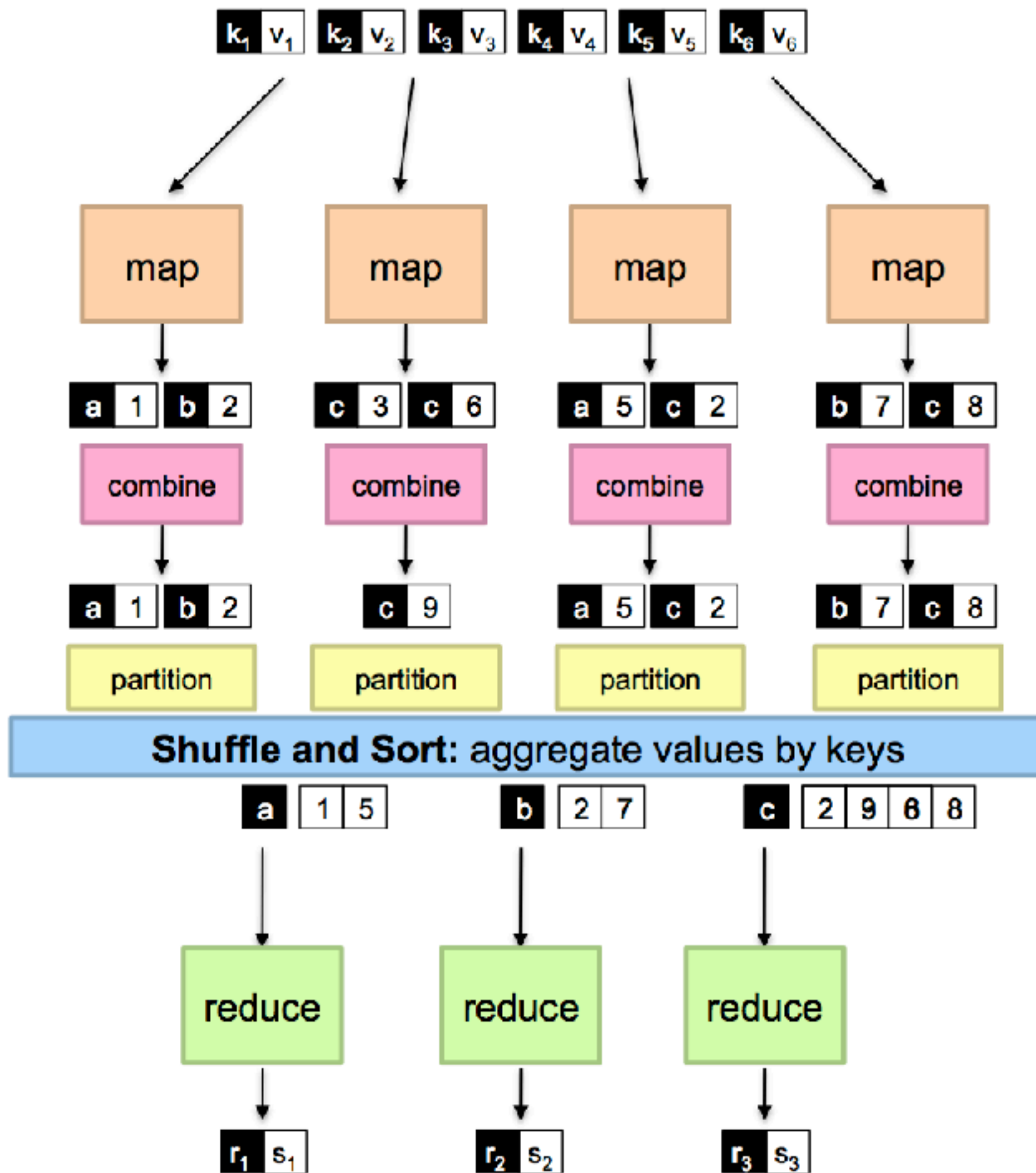
Partitions

- In addition to map and reduce functions, you may specify Partition (k, number of reducers) -- choice of reducer for k'
- Implemented by the invisible shuffle- and-sort stage
- Default partition: $\text{hash}(k) \bmod R$ (#reducers)
- Each reducer sees the keys in its partition in sorted order



Combiners

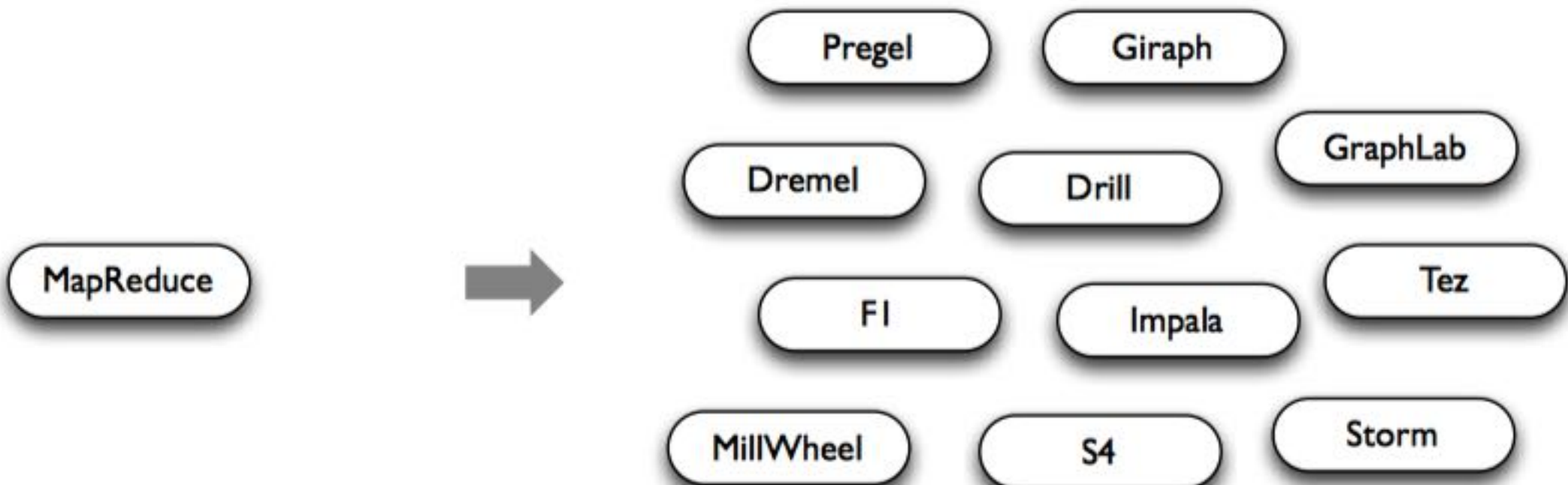
- “Combiner” functions can run on same machine as a mapper
- Causes a mini-reduce phase to occur before the real reduce phase, to save bandwidth



What are the major
drawbacks of Map Reduce?

MR Drawbacks

- Might be hard to express some problem in MapReduce
 - E.g., database schemas, complex data structures, etc.
- Inefficient for *multi-pass* algorithms



General Batch Processing

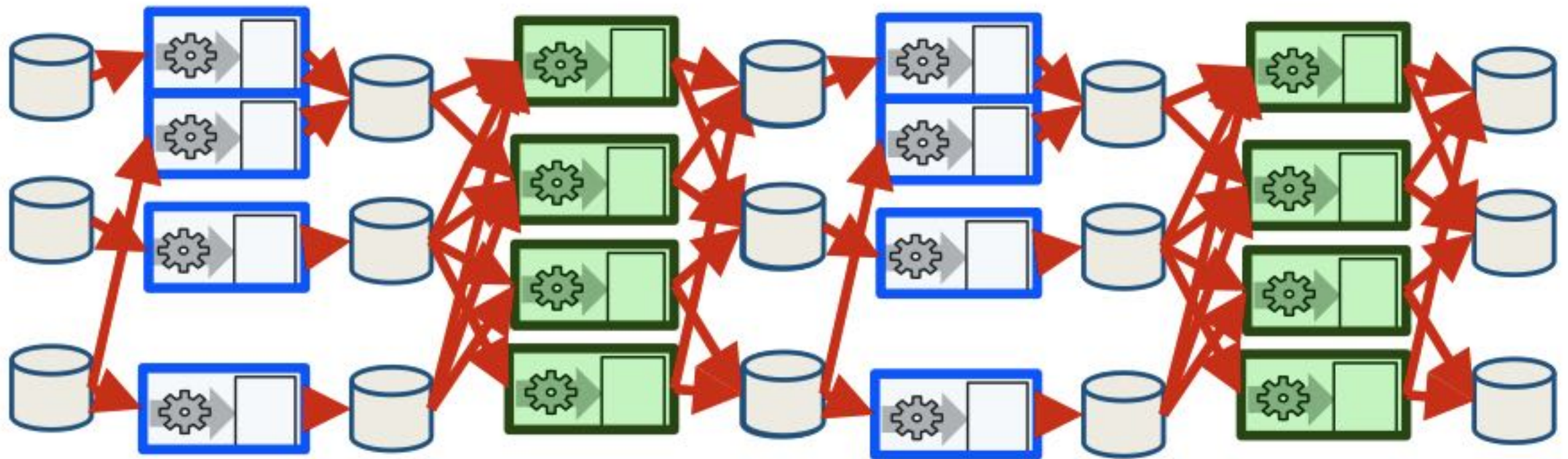
Specialized Systems:

iterative, interactive, streaming, graph, etc.

**MR doesn't compose well for large applications,
and so *specialized systems* emerged as workarounds**

Iterative Jobs

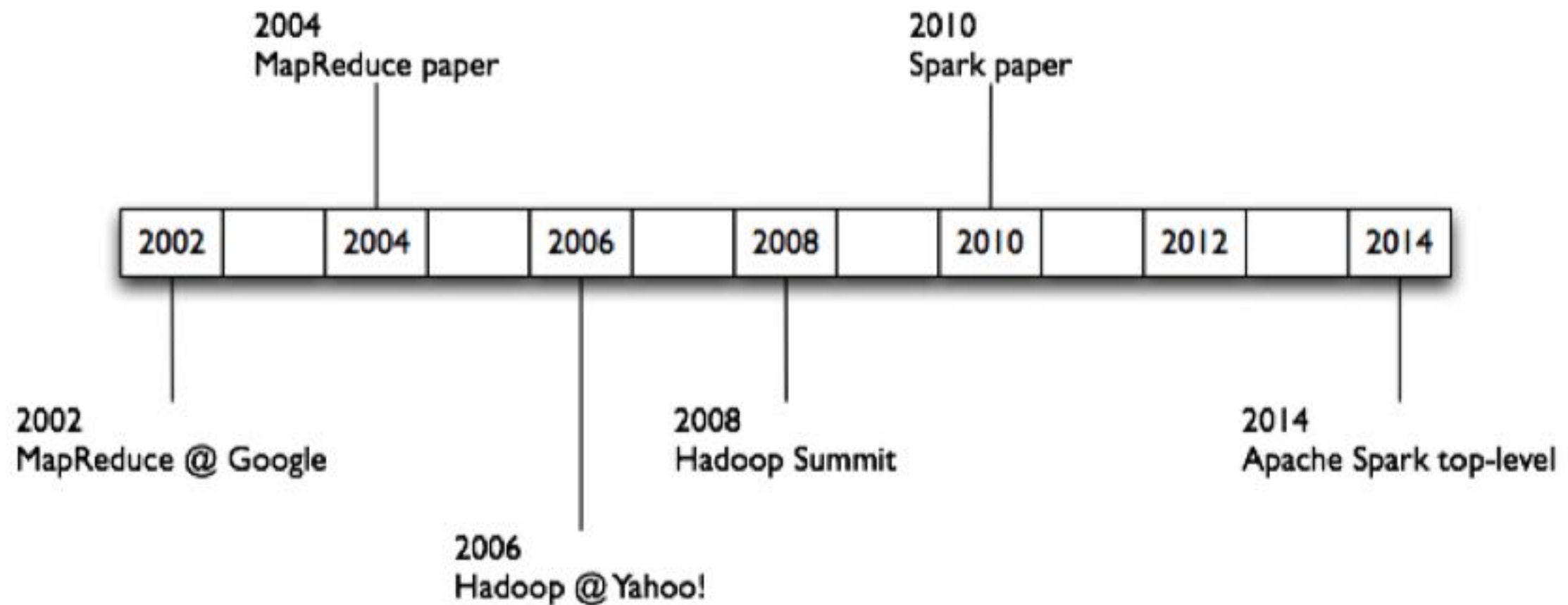
Involve a lot of disk I/O



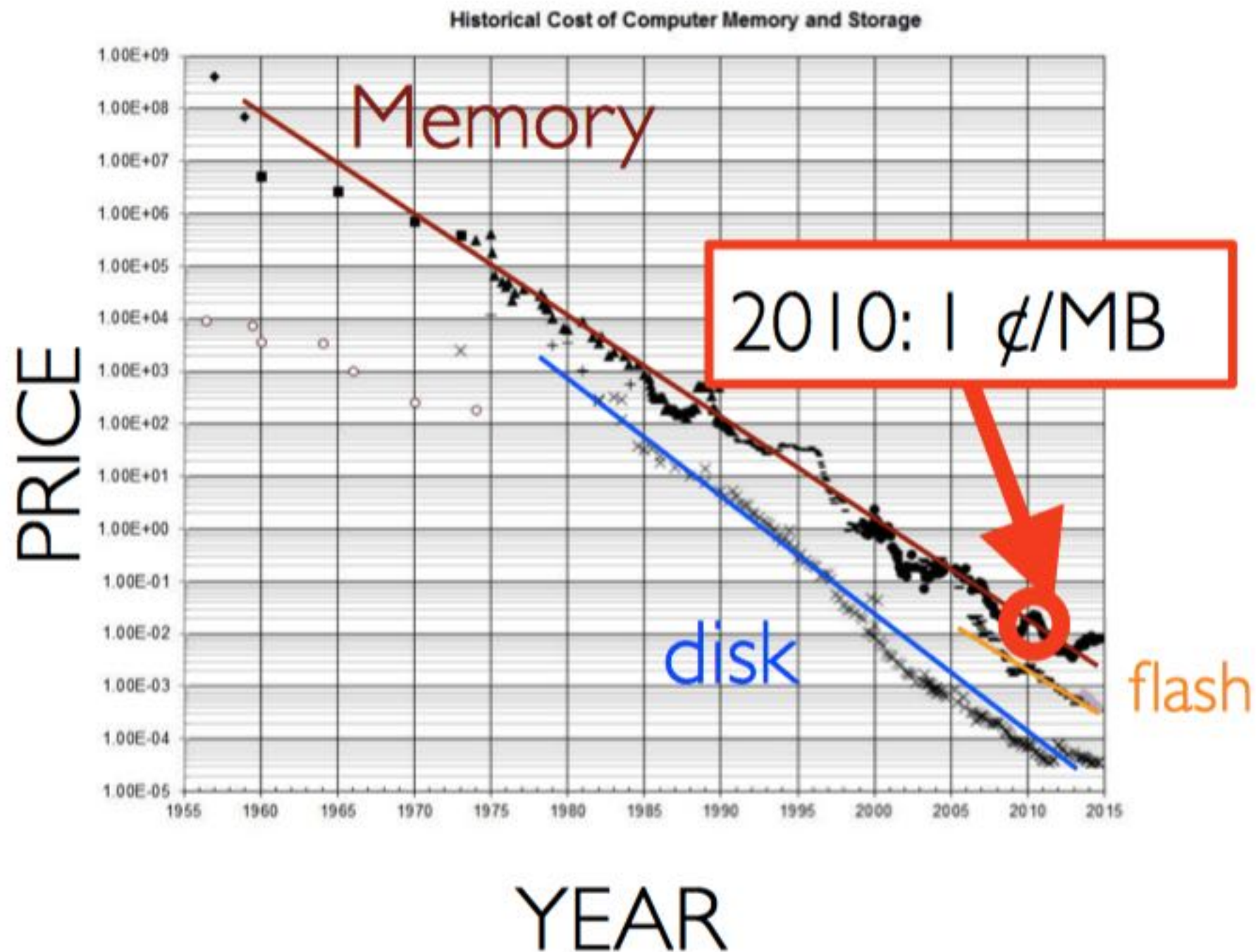
Disk I/O is very slow!



History

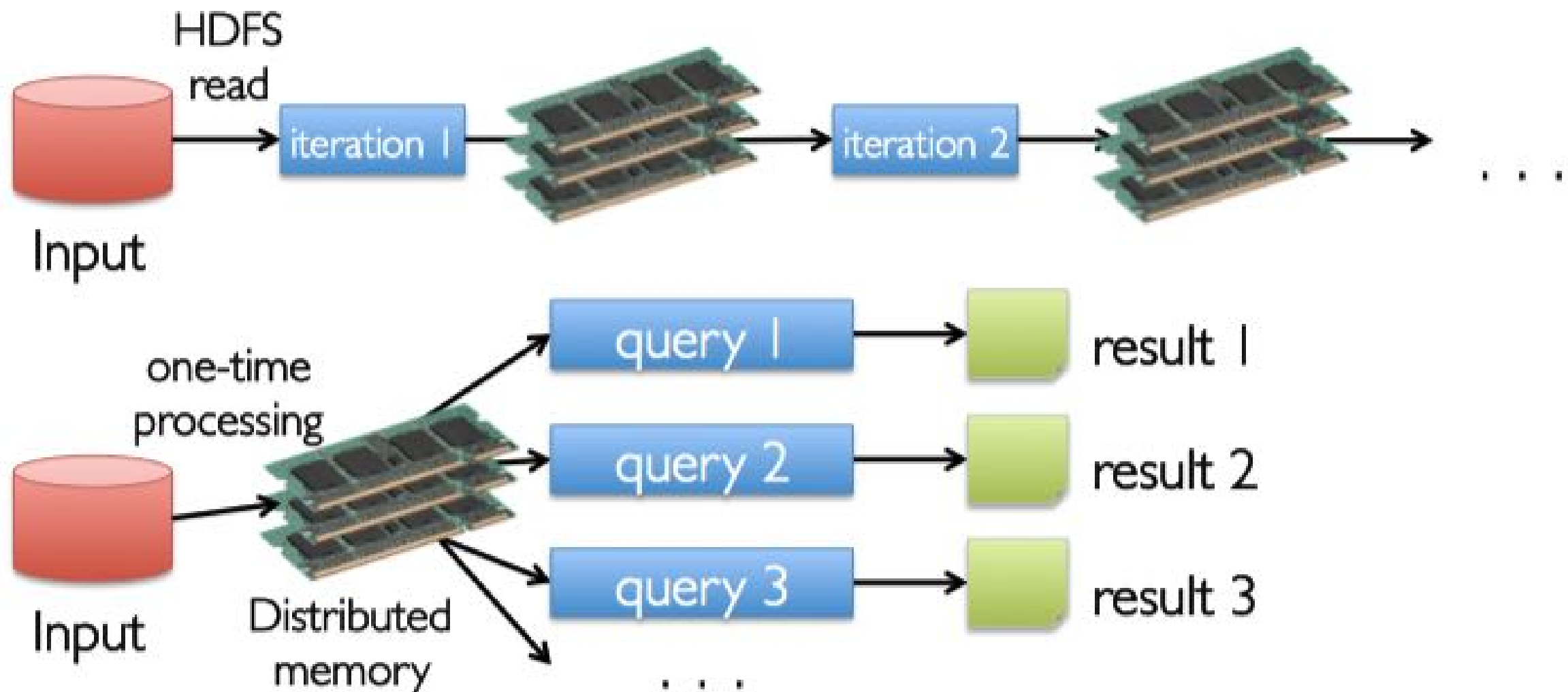


Cost of Memory



In-Memory Data Sharing

10-100x faster than network and disks



Resilient Distributed Datasets (RDDs)

- A fault-tolerant collection of elements (a list) that can be operated on in parallel
- Partitioned collections of objects spread across a cluster, stored in memory or on disk
- RDDs built and manipulated through a diverse set of parallel **transformations** (map, filter, join) and **actions** (count, collect, save)
- RDDs automatically rebuilt on machine failure

Transformations & Actions

- Transformations are *lazy* (not computed immediately)
- The transformed RDD gets *recomputed* when an action is run on it (default)
- However, an RDD can be *persisted* into storage in memory or disk

Word Count

```
wordsList = ['cat', 'elephant', 'rat', 'rat', 'cat']
wordsRDD = sc.parallelize(wordsList, 3)
wordCountsCollected = (wordsRDD
                        .map(lambda w: (w, 1))
                        .reduceByKey(lambda x,y: x+y)
                        .collect())

[('rat', 2), ('elephant', 1), ('cat', 2)]
```

Transformations

<i>transformation</i>	<i>description</i>
map (<i>func</i>)	return a new distributed dataset formed by passing each element of the source through a function <i>func</i>
filter (<i>func</i>)	return a new dataset formed by selecting those elements of the source on which <i>func</i> returns true
flatMap (<i>func</i>)	similar to map, but each input item can be mapped to 0 or more output items (so <i>func</i> should return a Seq rather than a single item)
sample (<i>withReplacement</i> , <i>fraction</i> , <i>seed</i>)	sample a fraction <i>fraction</i> of the data, with or without replacement, using a given random number generator <i>seed</i>
union (<i>otherDataset</i>)	return a new dataset that contains the union of the elements in the source dataset and the argument
distinct ([<i>numTasks</i>])	return a new dataset that contains the distinct elements of the source dataset

etc.

Actions

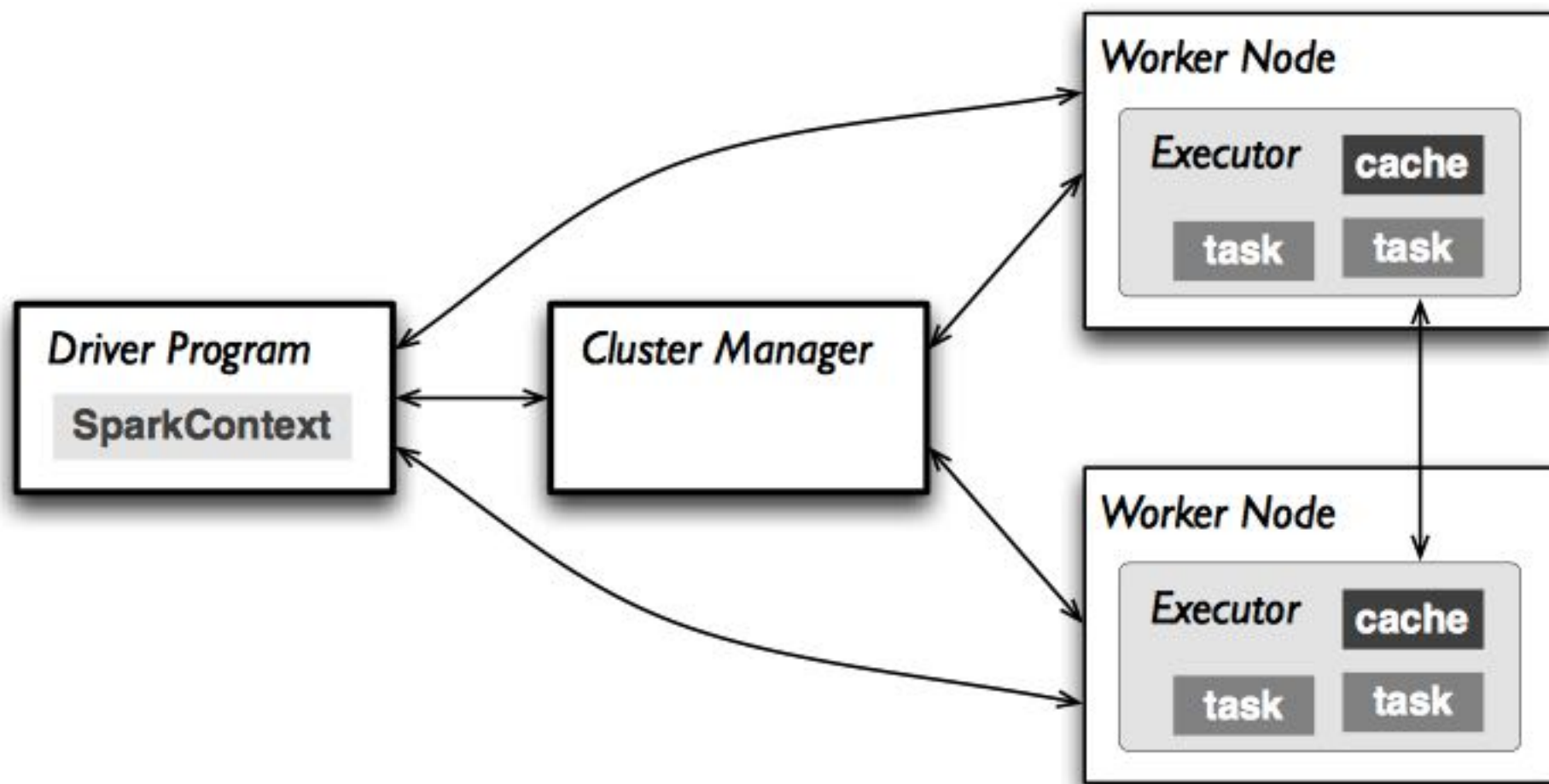
<i>action</i>	<i>description</i>
reduce (<i>func</i>)	aggregate the elements of the dataset using a function <i>func</i> (which takes two arguments and returns one), and should also be commutative and associative so that it can be computed correctly in parallel
collect ()	return all the elements of the dataset as an array at the driver program – usually useful after a filter or other operation that returns a sufficiently small subset of the data
count ()	return the number of elements in the dataset
first ()	return the first element of the dataset – similar to <i>take(1)</i>
take (<i>n</i>)	return an array with the first <i>n</i> elements of the dataset – currently not executed in parallel, instead the driver program computes all the elements
takeSample (<i>withReplacement</i> , <i>fraction</i> , <i>seed</i>)	return an array with a random sample of <i>num</i> elements of the dataset, with or without replacement, using the given random number generator seed

etc.

The Spark Framework

- Provides programming abstraction and parallel runtime to hide complexities of fault-tolerance and slow machines
- “Here’s an operation, run it on all of the data”
- “I don’t care where it runs (you schedule that)”
- “In fact, feel free to run it twice on different nodes”

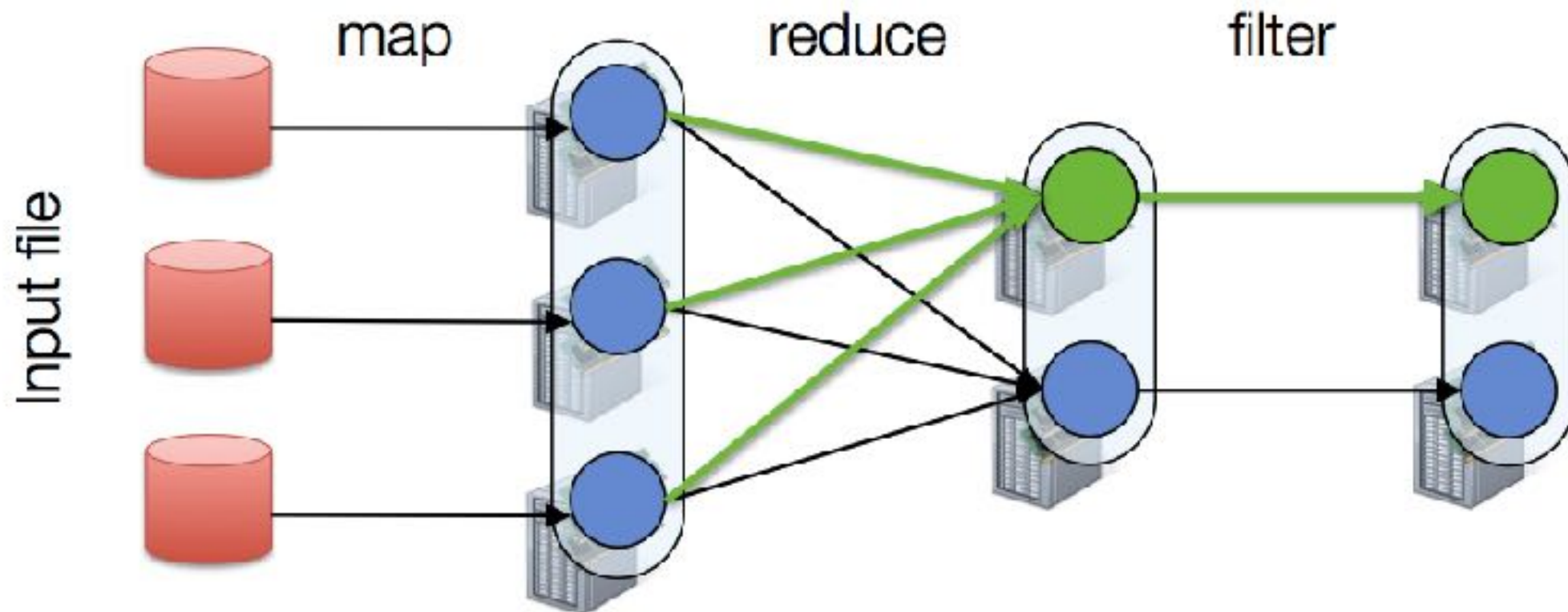
The Spark Framework



Fault Tolerance

RDDs track lineage to rebuild lost data

```
file.map(lambda rec: (rec.type, 1))  
    .reduceByKey(lambda x, y: x + y)  
    .filter(lambda (type, count): count > 10)
```



Log Mining Example

```
# load error messages from a log into memory
# then interactively search for patterns

# base RDD
lines = sc.textFile("/mnt/paco/intro/error_log.txt") \
    .map(lambda x: x.split("\t"))

# transformed RDDs
errors = lines.filter(lambda x: x[0] == "ERROR")
messages = errors.map(lambda x: x[1])

# persistence
messages.cache()

# action 1
messages.filter(lambda x: x.find("mysql") > -1).count()

# action 2
messages.filter(lambda x: x.find("php") > -1).count()
```

```
# base RDD
lines = sc.textFile("/mnt/paco/intro/error_log.txt") \
    .map(lambda x: x.split("\t"))

# transformed RDDs
errors = lines.filter(lambda x: x[0] == "ERROR")
messages = errors.map(lambda x: x[1])

# persistence
messages.cache()
```

```
# action 1
messages.filter(lambda x: x.find("mysql") > -1).count()

# action 2
messages.filter(lambda x: x.find("php") > -1).count()
```

discussing the other part

Worker

Driver

Worker

Worker

```
# base RDD
lines = sc.textFile("/mnt/paco/intro/error_log.txt") \
    .map(lambda x: x.split("\t"))

# transformed RDDs
errors = lines.filter(lambda x: x[0] == "ERROR")
messages = errors.map(lambda x: x[1])

# persistence
messages.cache()
```

```
# action 1
messages.filter(lambda x: x.find("mysql") > -1).count()

# action 2
messages.filter(lambda x: x.find("php") > -1).count()
```

discussing the other part

Worker

block 1

Driver

Worker

block 2

Worker

block 3


```
# base RDD
lines = sc.textFile("/mnt/paco/intro/error_log.txt") \
    .map(lambda x: x.split("\t"))

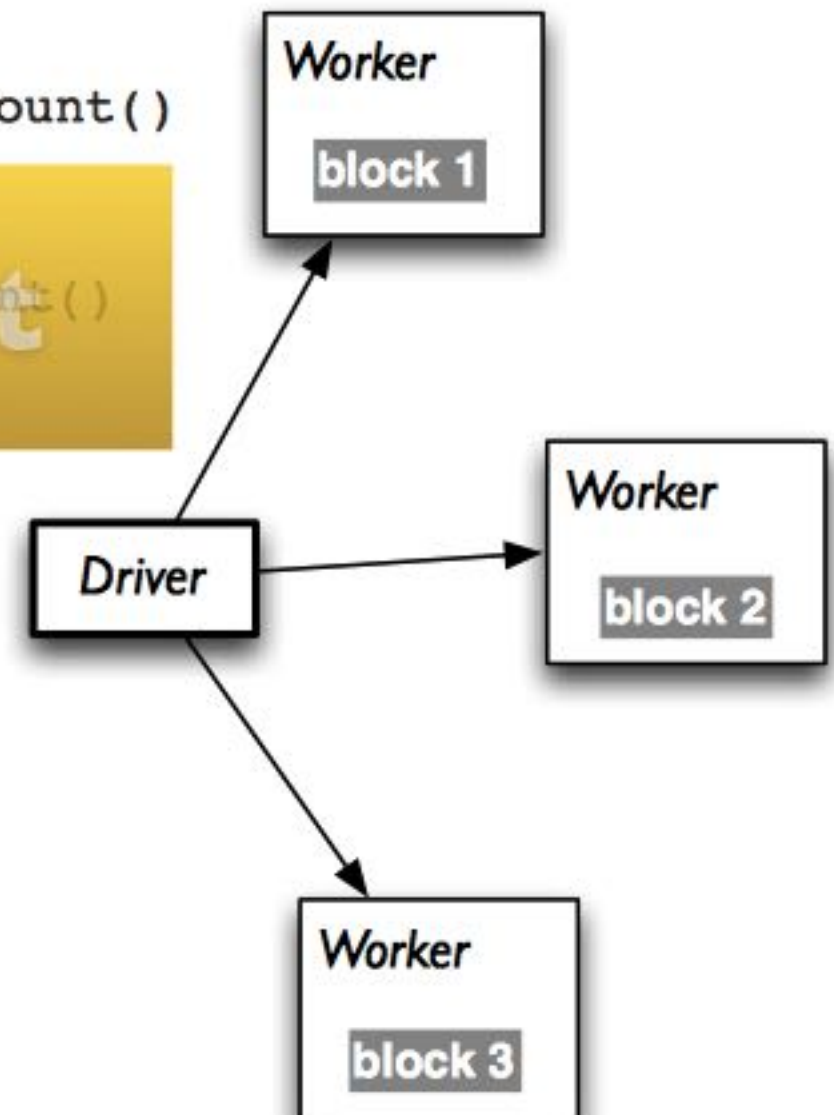
# transformed RDDs
errors = lines.filter(lambda x: x[0] == "ERROR")
messages = errors.map(lambda x: x[1])

# persistence
messages.cache()

# action 1
messages.filter(lambda x: x.find("mysql") > -1).count()
```

```
# action 2
messages.filter(lambda x: x.find("mysql") > -1).count()
```

discussing the other part



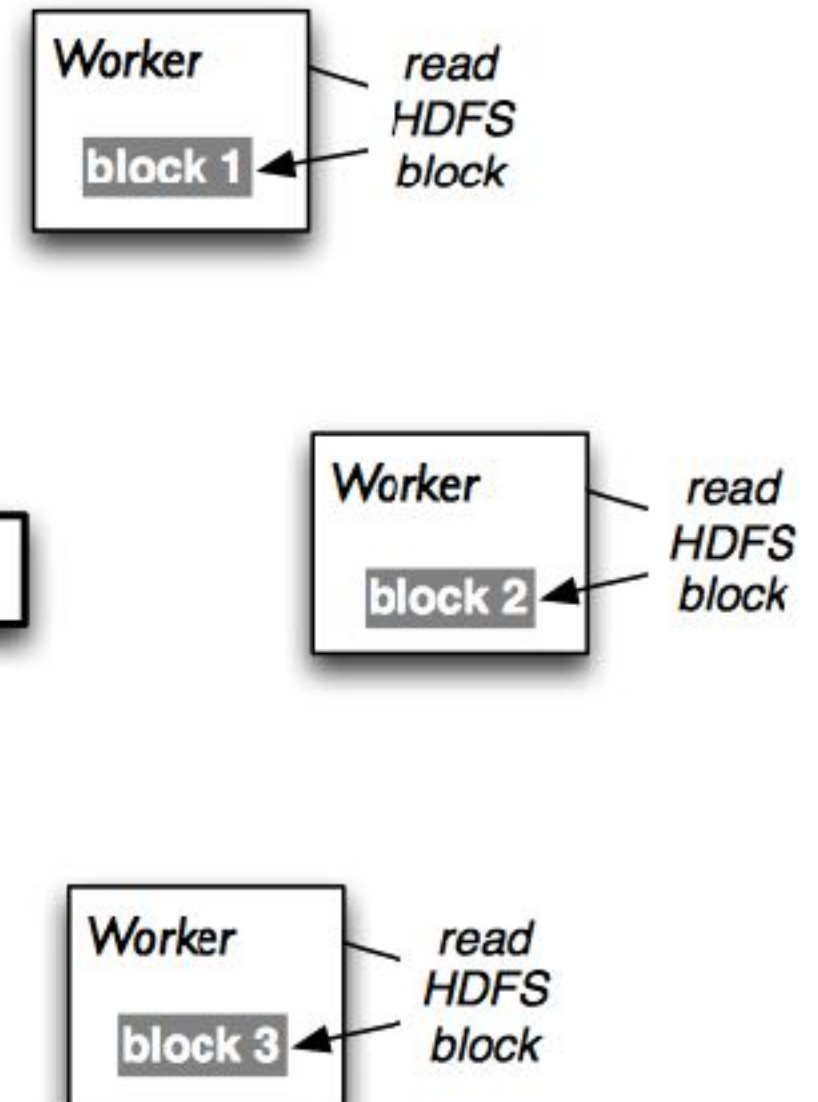
```
# base RDD
lines = sc.textFile("/mnt/paco/intro/error_log.txt") \
    .map(lambda x: x.split("\t"))

# transformed RDDs
errors = lines.filter(lambda x: x[0] == "ERROR")
messages = errors.map(lambda x: x[1])

# persistence
messages.cache()

# action 1
messages.filter(lambda x: x.find("mysql") > -1).count()
```

```
# action 2
messages.filter(lambda x: x.find("log") > -1).count()
```



```
# base RDD
lines = sc.textFile("/mnt/paco/intro/error_log.txt") \
    .map(lambda x: x.split("\t"))

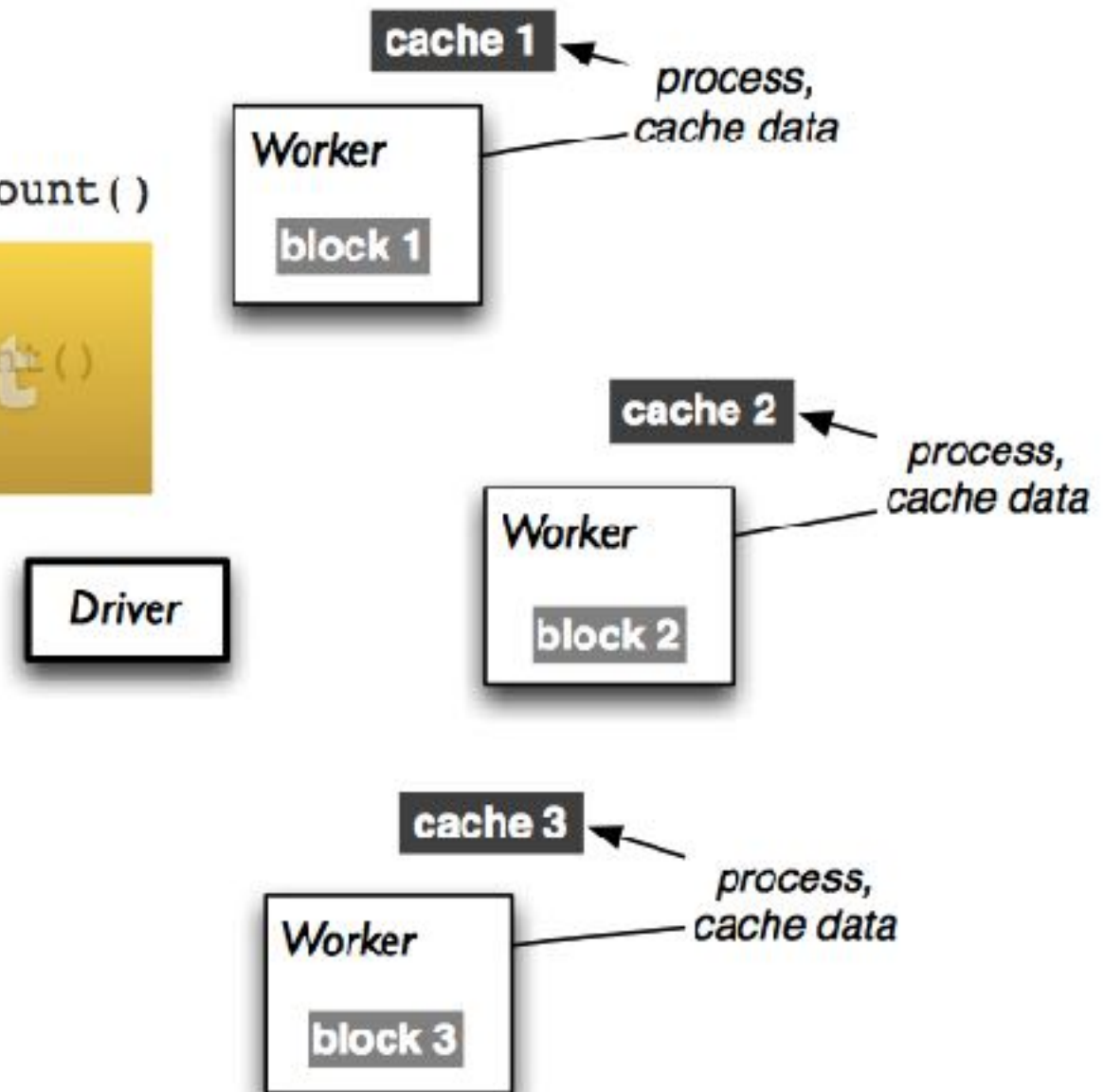
# transformed RDDs
errors = lines.filter(lambda x: x[0] == "ERROR")
messages = errors.map(lambda x: x[1])

# persistence
messages.cache()

# action 1
messages.filter(lambda x: x.find("mysql") > -1).count()
```

```
# action 2
messages.filter(lambda x: x.find("php") > -1).count()
```

discussing the other part




```
# base RDD
lines = sc.textFile("/mnt/paco/intro/error_log.txt") \
    .map(lambda x: x.split("\t"))

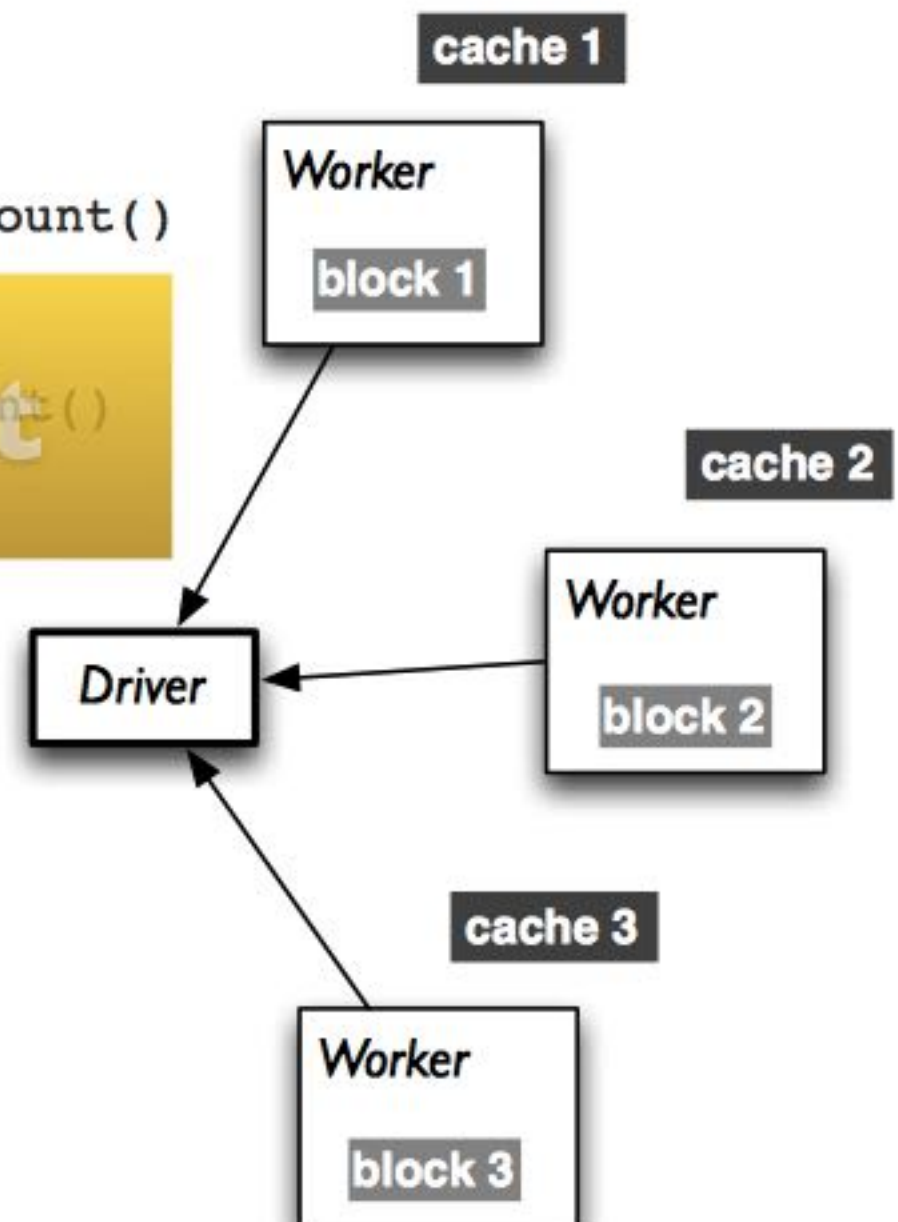
# transformed RDDs
errors = lines.filter(lambda x: x[0] == "ERROR")
messages = errors.map(lambda x: x[1])

# persistence
messages.cache()

# action 1
messages.filter(lambda x: x.find("mysql") > -1).count()
```

```
# action 2
messages.filter(lambda x: x.find("php") > -1).count()
```

discussing the other part



```
# base RDD
lines = sc.textFile("/mnt/paco/intro/error_log.txt") \
    .map(lambda x: x.split("\t"))

# transformed RDDs
errors = lines.filter(lambda x: x[0] == "ERROR")
messages = errors.map(lambda x: x[1])

# persistence
messages.cache()

# action 1
messages.filter(lambda x: x.find("mysql") > -1).count()

# action 2
messages.filter(lambda x: x.find("php") > -1).count()
```

cache 1

Worker

block 1

cache 2

Worker

block 2

Driver

cache 3

Worker

block 3

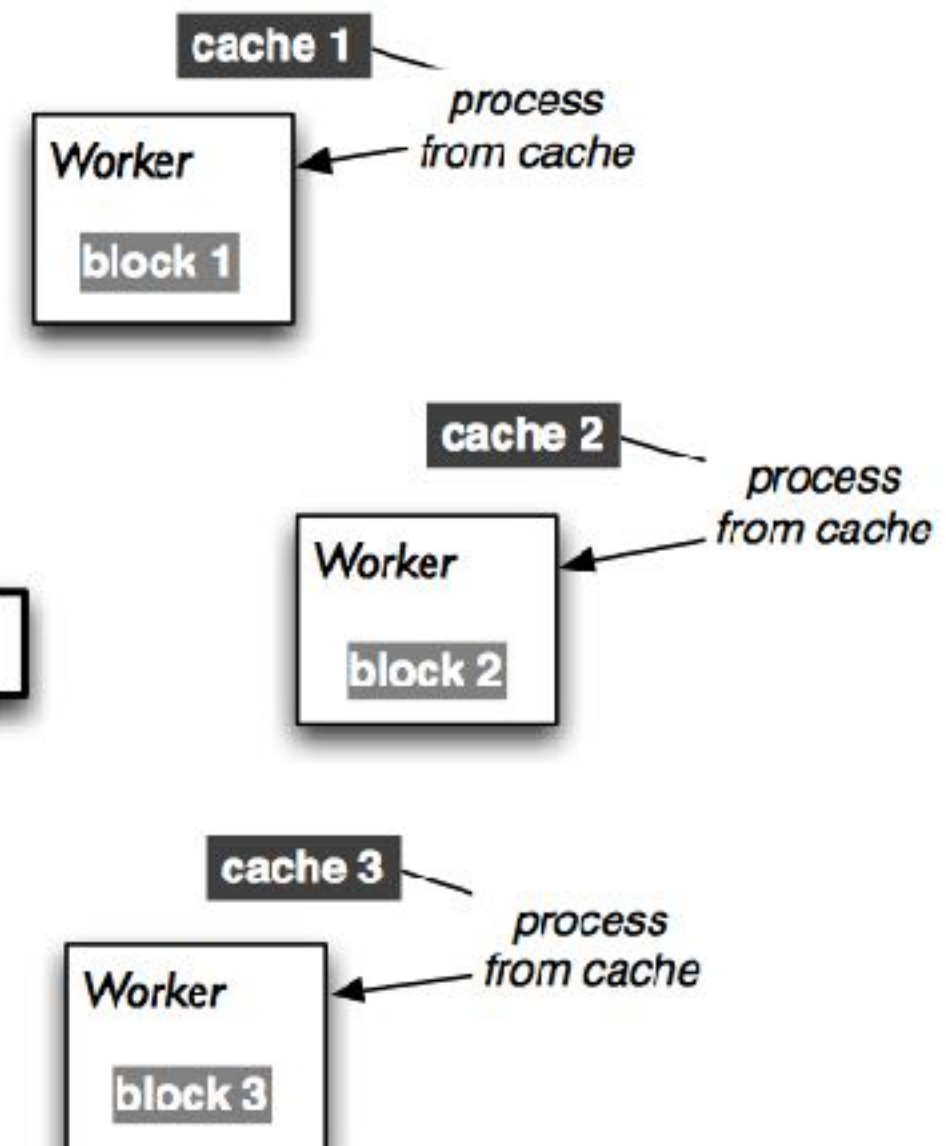
```
# base RDD
lines = sc.textFile("/mnt/paco/intro/error_log.txt") \
    .map(lambda x: x.split("\t"))

# transformed RDDs
errors = lines.filter(lambda x: x[0] == "ERROR")
messages = errors.map(lambda x: x[1])

# persistence
messages.cache()

# action 1
messages.filter(lambda x: x.find("mysql") > -1).count()

# action 2
messages.filter(lambda x: x.find("php") > -1).count()
```



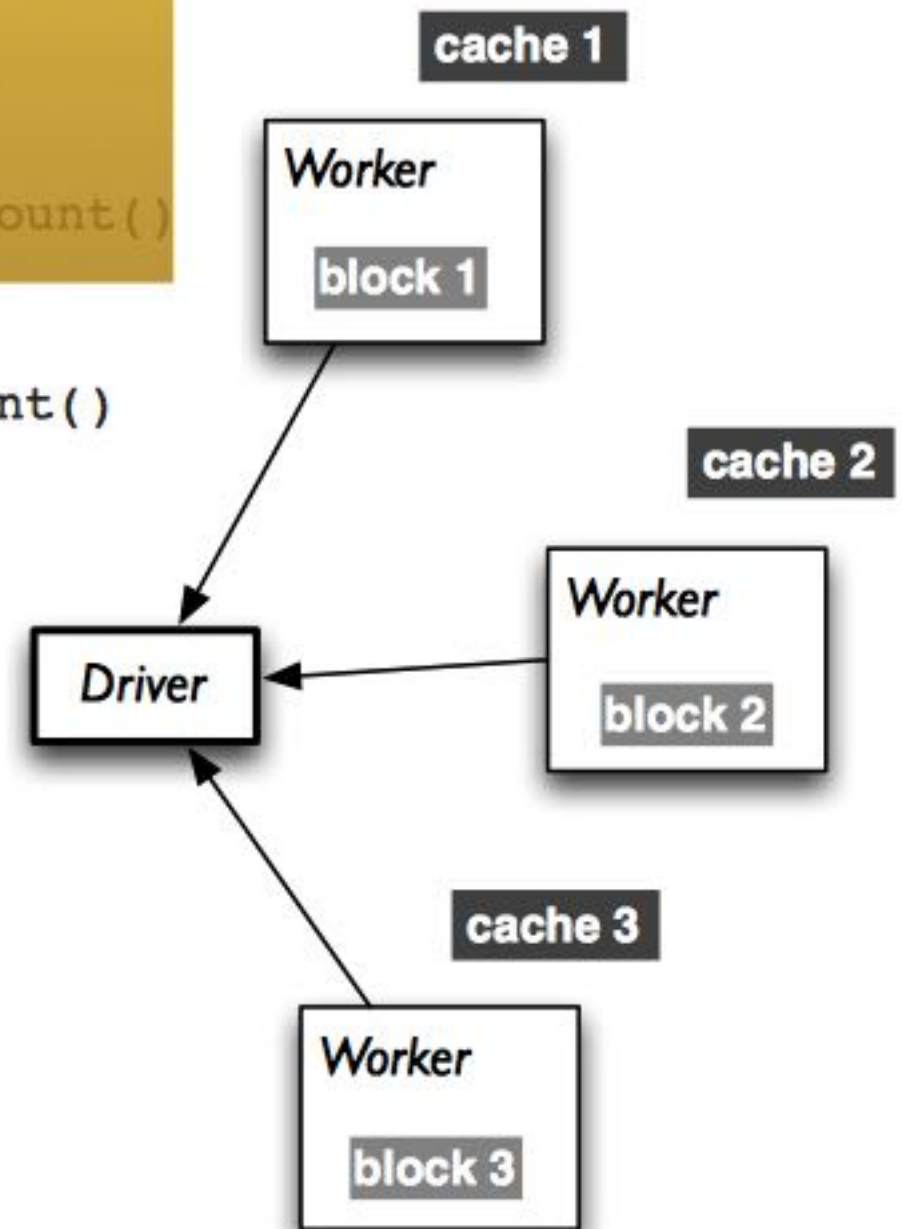
```
# base RDD
lines = sc.textFile("/mnt/paco/intro/error_log.txt") \
    .map(lambda x: x.split("\t"))

# transformed RDDs
errors = lines.filter(lambda x: x[0] == "ERROR")
messages = errors.map(lambda x: x[1])

# persistence
messages.cache()

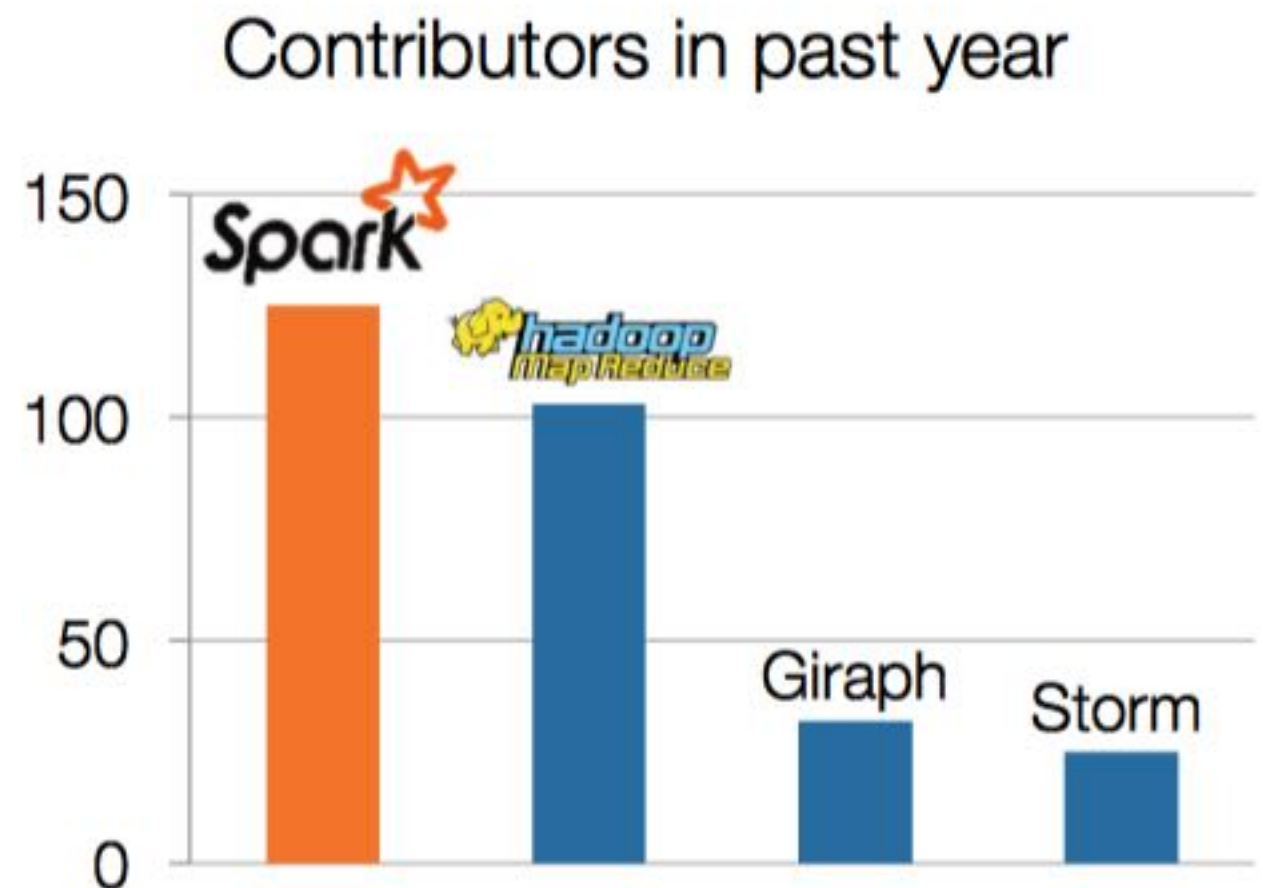
# action 1
messages.filter(lambda x: x.find("mysql") > -1).count()

# action 2
messages.filter(lambda x: x.find("php") > -1).count()
```



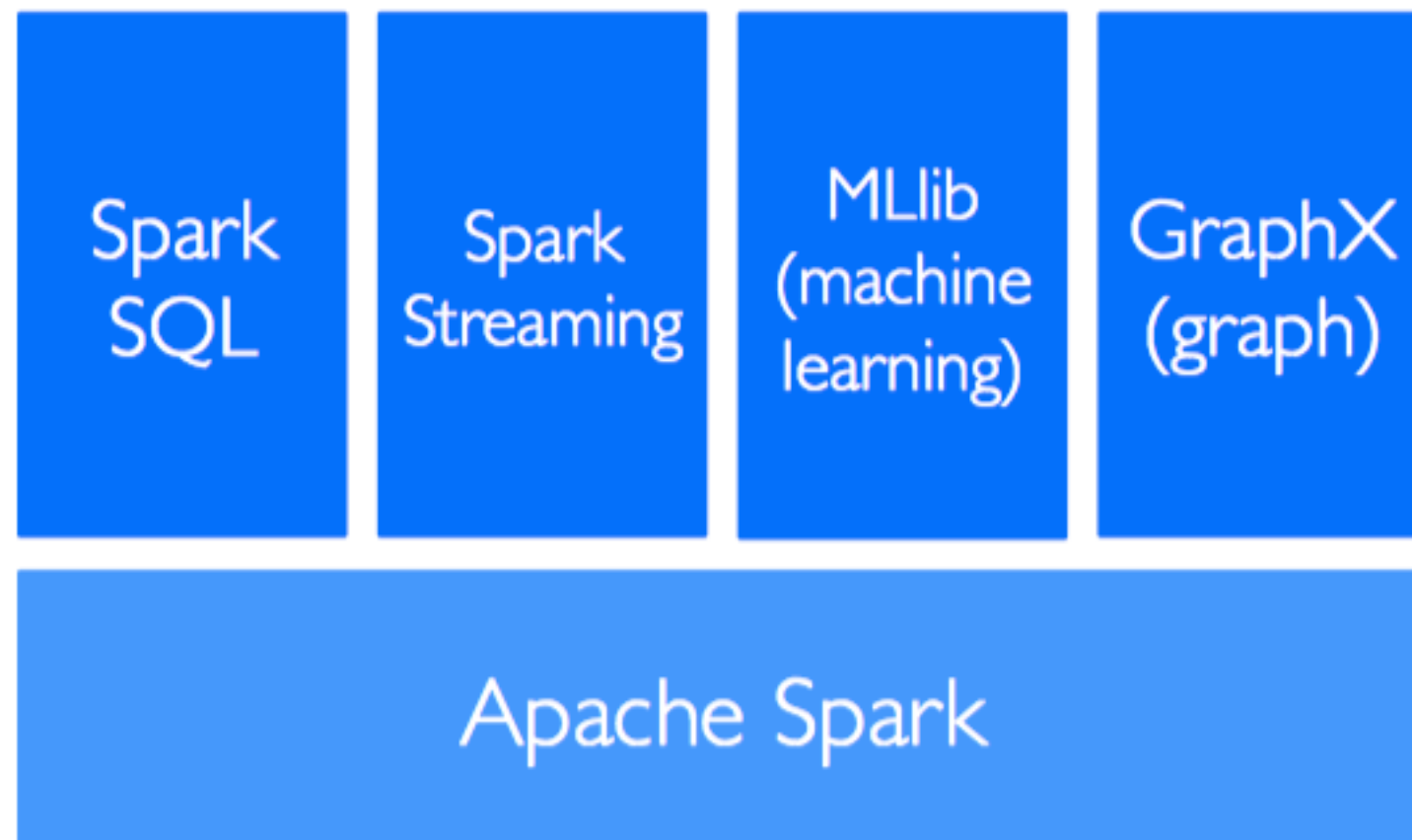
Spark Community

- Most active open source community in big data
- 200+ developers, 50+ companies contributing



Spark Tools

Standard libraries included with Spark



Spark SQL

Enables loading & querying structured data in Spark

From Hive:

```
c = HiveContext(sc)
rows = c.sql("select text, year from hivetable")
rows.filter(lambda r: r.year > 2013).collect()
```

From JSON:

```
c.jsonFile("tweets.json").registerAsTable("tweets")
c.sql("select text, user.name from tweets")
```

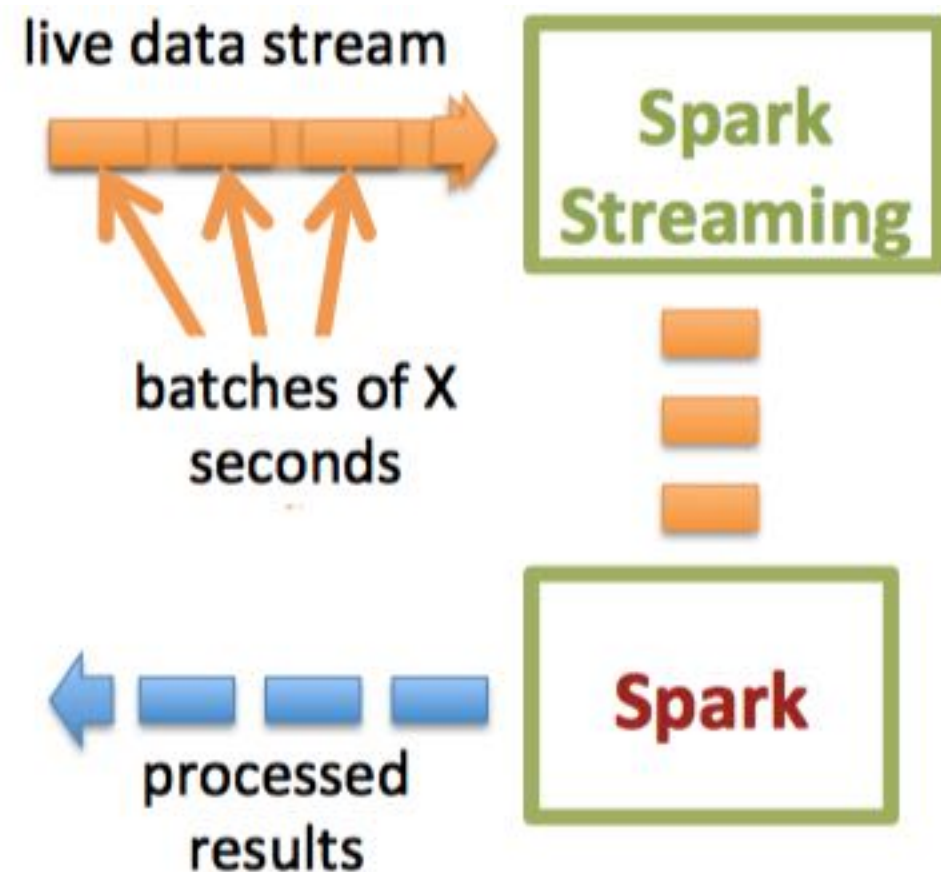
tweets.json

```
{
  "text": "hi",
  "user": {
    "name": "matei",
    "id": 123
  }
}
```

Spark Streaming

Run a streaming computation as a series of very small, deterministic batch jobs

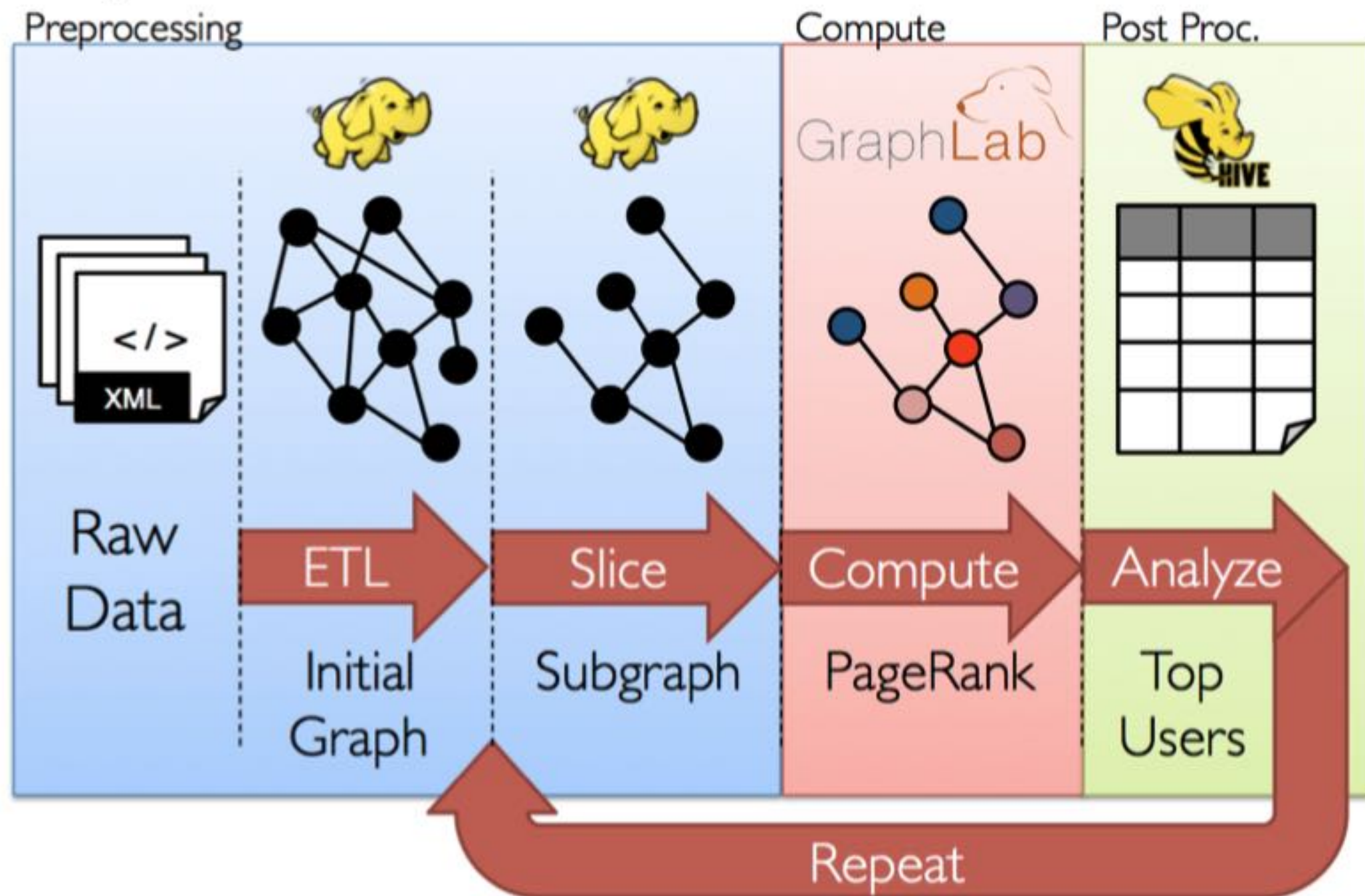
- Chop up the live stream into batches of X seconds
- Spark treats each batch of data as RDDs and processes them using RDD operations
- Finally, the processed results of the RDD operations are returned in batches



MLib Algorithms

- Regression: generalized linear models (GLMs), regression tree
- Classification: logistic regression, linear SVM, naïve Bayes, classification tree
- Decomposition: SVD, PCA
- Clustering: k-means
- Collaborative filtering: alternating least squares (ALS), non-negative matrix factorization (NMF)
- Optimization: stochastic gradient descent

GraphX



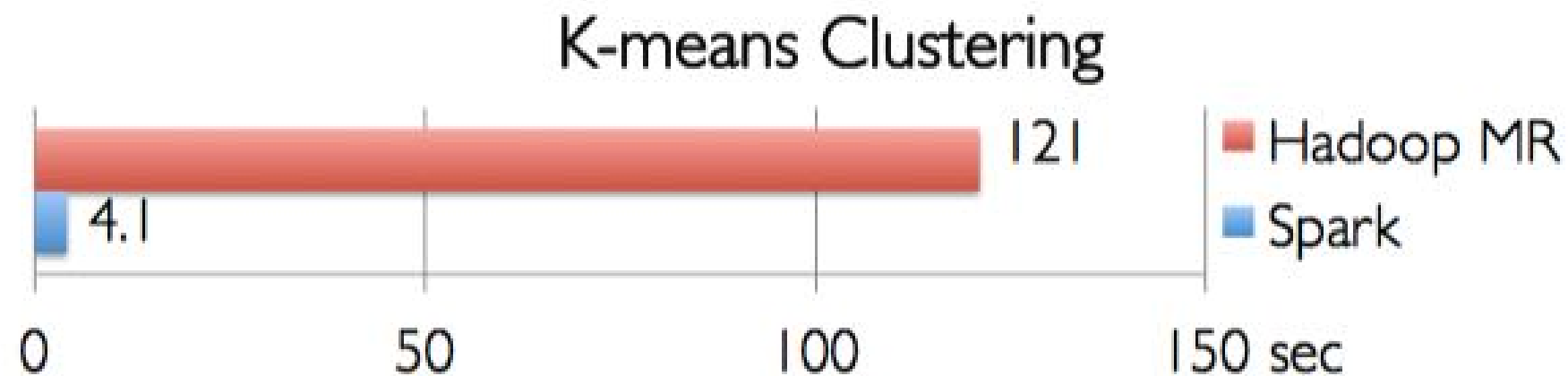
GraphX Algorithms

- Collaborative Filtering: Alternating Least Squares, Stochastic Gradient Descent, Tensor Factorization
- Community Detection: Triangle-Counting, K-core Decomposition, K-Truss
- Structured Prediction: Loopy Belief Propagation, Max-Product Linear Programs, Gibbs Sampling
- Graph Analytics; PageRank, Personalized PageRank, Shortest Path, Graph Coloring
- Classification: Neural Networks

MR / Spark Differences

	Hadoop Map Reduce	Spark
Storage	Disk only	In-memory or on disk
Operations	Map and Reduce	Map, Reduce, Join, Sample, etc...
Execution model	Batch	Batch, interactive, streaming
Programming environments	Java	Scala, Java, R, and Python

Performance Differences



Cloud Petabyte Sort

	Hadoop MR Record	Spark Record	Spark 1 PB
Data Size	102.5 TB	100 TB	1000 TB
Elapsed Time	72 mins $\xrightarrow{3x}$	23 mins	234 mins
# Nodes	2100 $\xrightarrow{10x}$	206	190
# Cores	50400 physical	6592 virtualized	6080 virtualized
Cluster disk throughput	3150 GB/s (est.)	618 GB/s	570 GB/s
Sort Benchmark Daytona Rules	Yes	Yes	No
Network	dedicated data center, 10Gbps	virtualized (EC2) 10Gbps network	virtualized (EC2) 10Gbps network
Sort rate	1.42 TB/min	4.27 TB/min	4.27 TB/min
Sort rate/node	0.67 GB/min	20.7 GB/min	22.5 GB/min