Scaling Up

EMSE 6992
Benjamin Harvey

Overview

- Why Big Data? (and Big Models)
- Hadoop
- Spark
- MPI

Big Data

Lots of Data:

- Facebook's daily logs: 60 **TB**
- 1000 genomes project: 200 **TB**
- Google web index: 10+ **PB**

Lots of Questions:

- Computational Marketing
- Recommendations and Personalization
- Genetic analysis

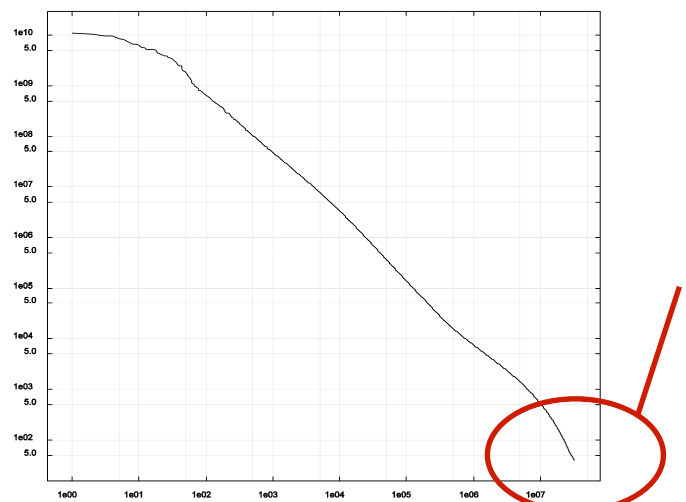
But How Much Data Do You Need?

The answer of course depends on the question but for many applications the answer is:

- **As much as you can get**

Big Data about people (text, web, social media) follow power law statistics.

Log feature frequency



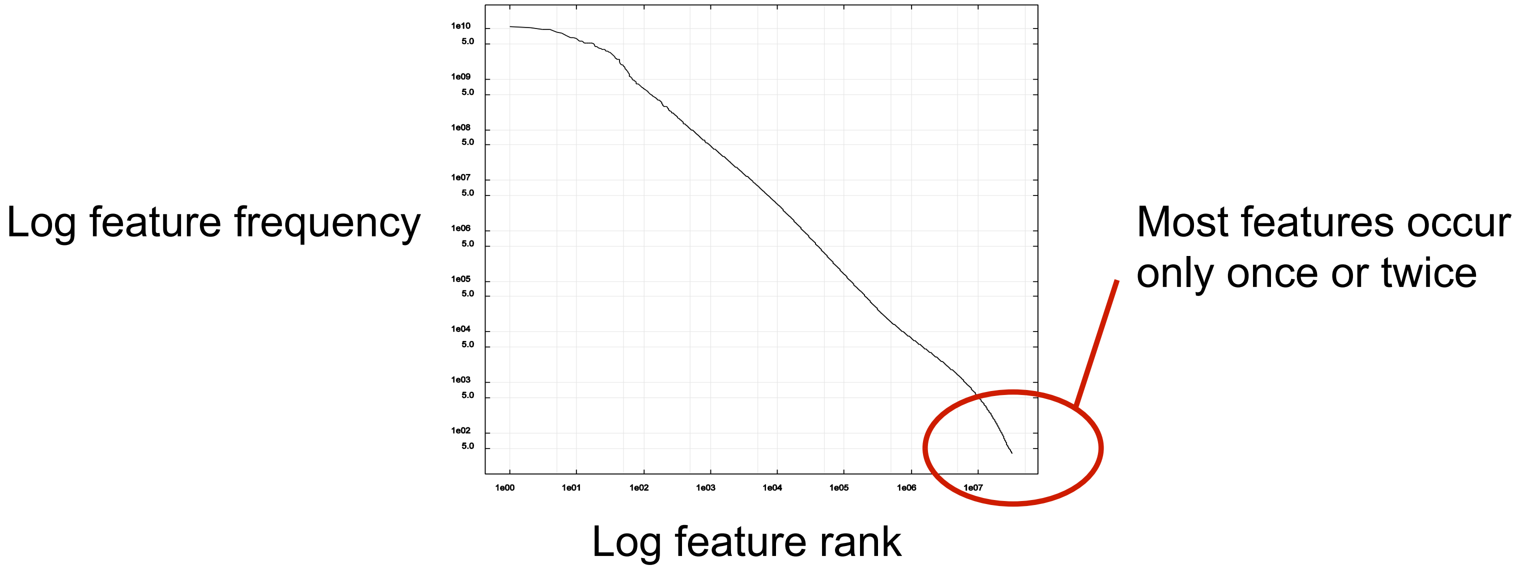
Most features occur only once or twice

Log feature rank

How Much Data Do You Need?

The number of features grows in proportion to the amount of data – doubling the dataset size roughly doubles the number of users we observe.

Even one or two observations of a user improves predictions for them, so more data (and bigger models!) ➔ more revenue.



Hardware for Big Data

Budget hardware

Not "gold plated"

Many low-end servers

Easy to add capacity

Cheaper per CPU/disk

Increased Complexity in software:

- Fault tolerance
- Virtualization

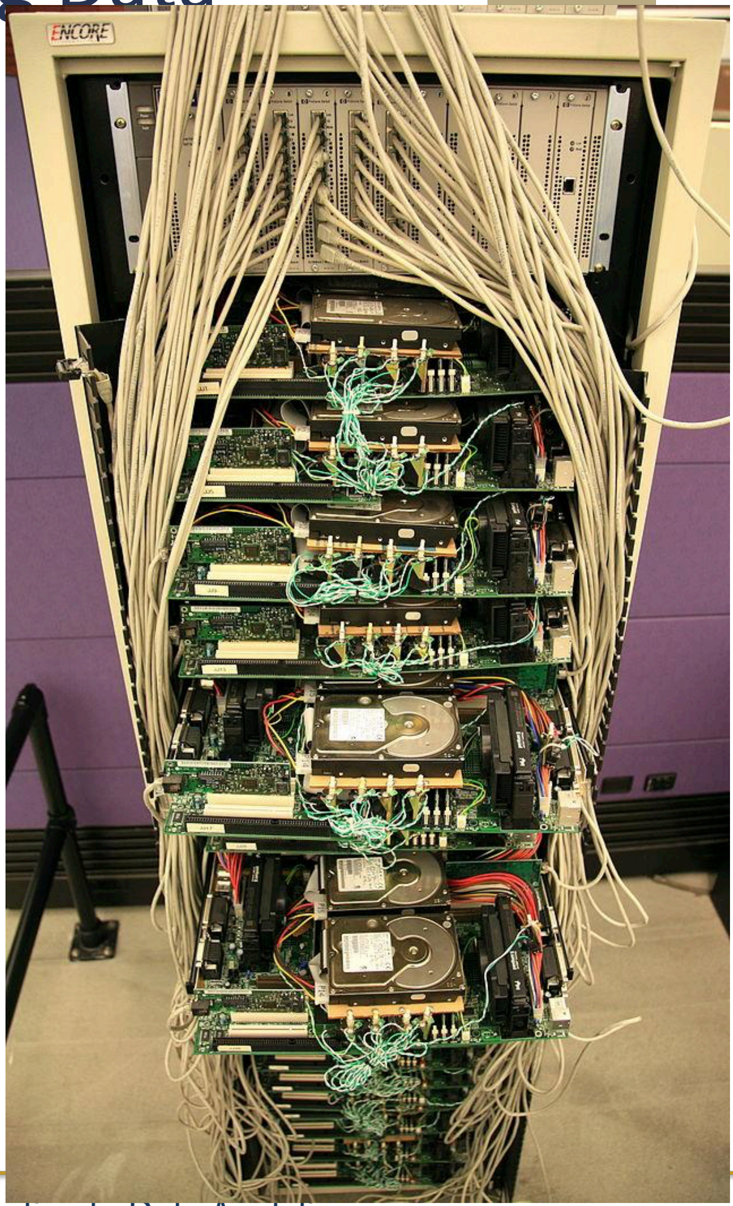


Image: Steve Jurvetson/Flickr

Problems with Cheap HW

Failures, e.g. (Google numbers)

- 1-5% hard drives/year
- 0.2% DIMMs/year

Commodity Network (1-10 Gb/s) speeds vs. RAM

- Much more latency (100x – 100,000x)
- Lower throughput (100x-1000x)

- **Uneven Performance**
- Variable network latency
- External loads

MapReduce: Word Count

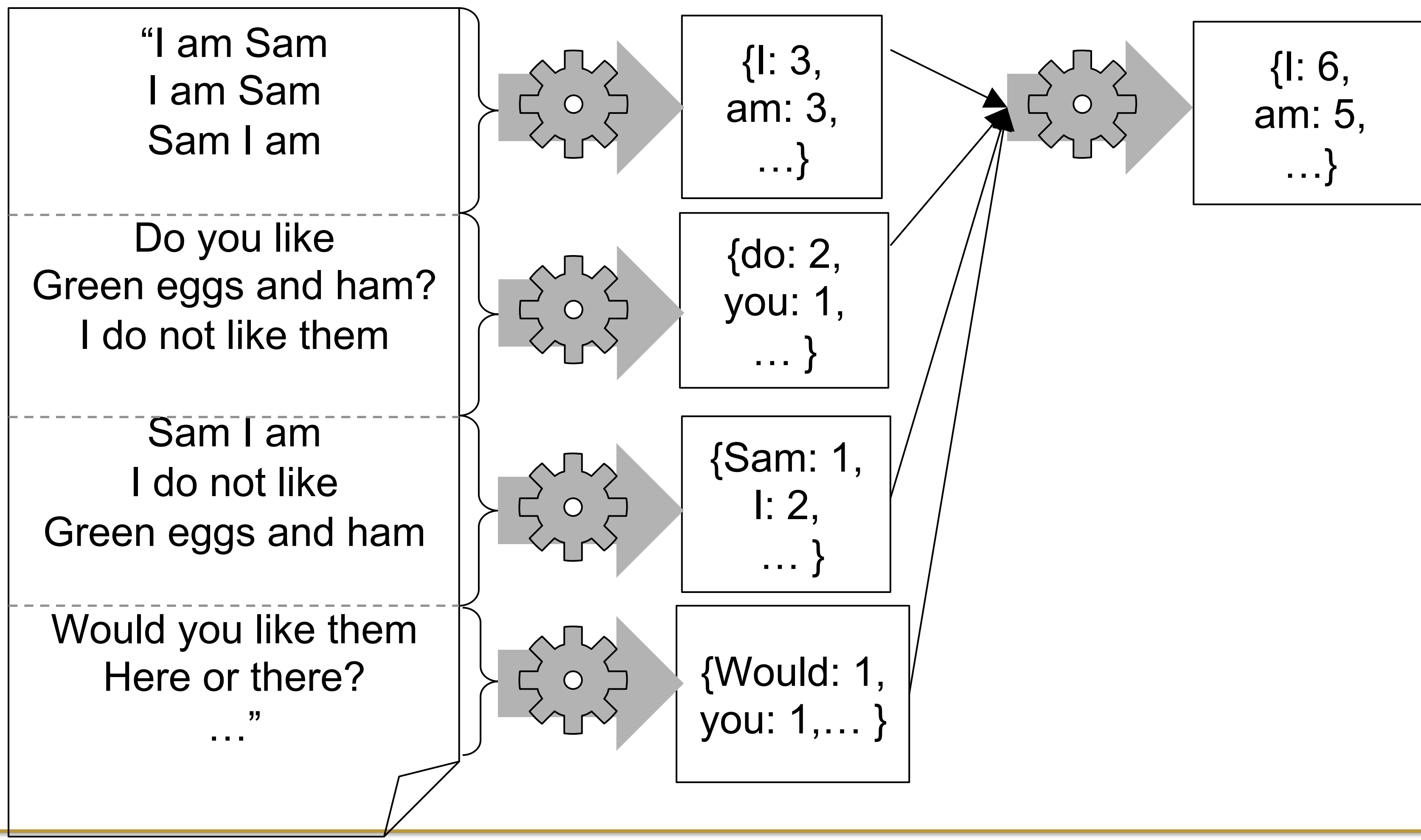
"I am Sam
I am Sam
Sam I am

Do you like
Green eggs and ham?
I do not like them

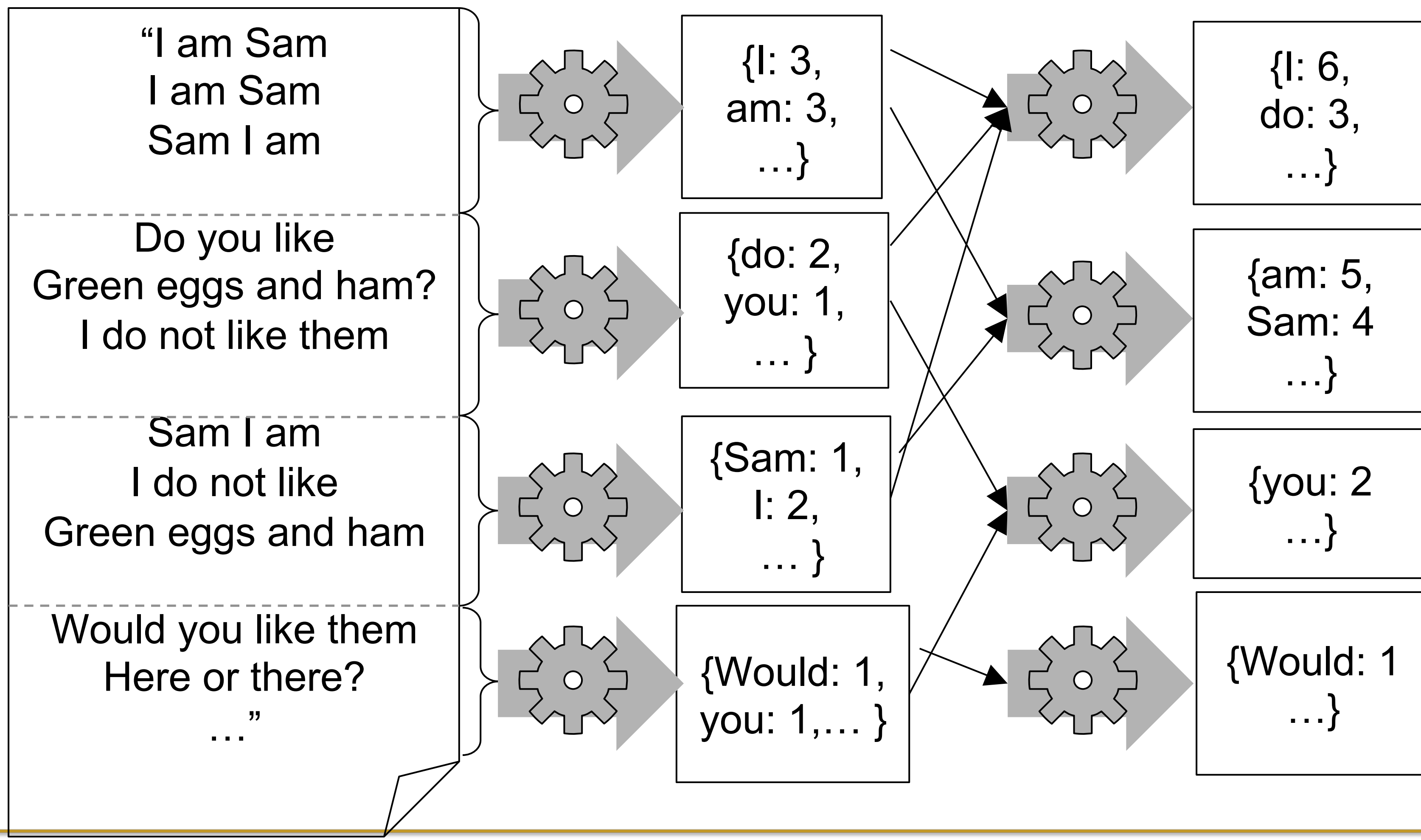
Sam I am
I do not like
Green eggs and ham

Would you like them
Here or there?
…"

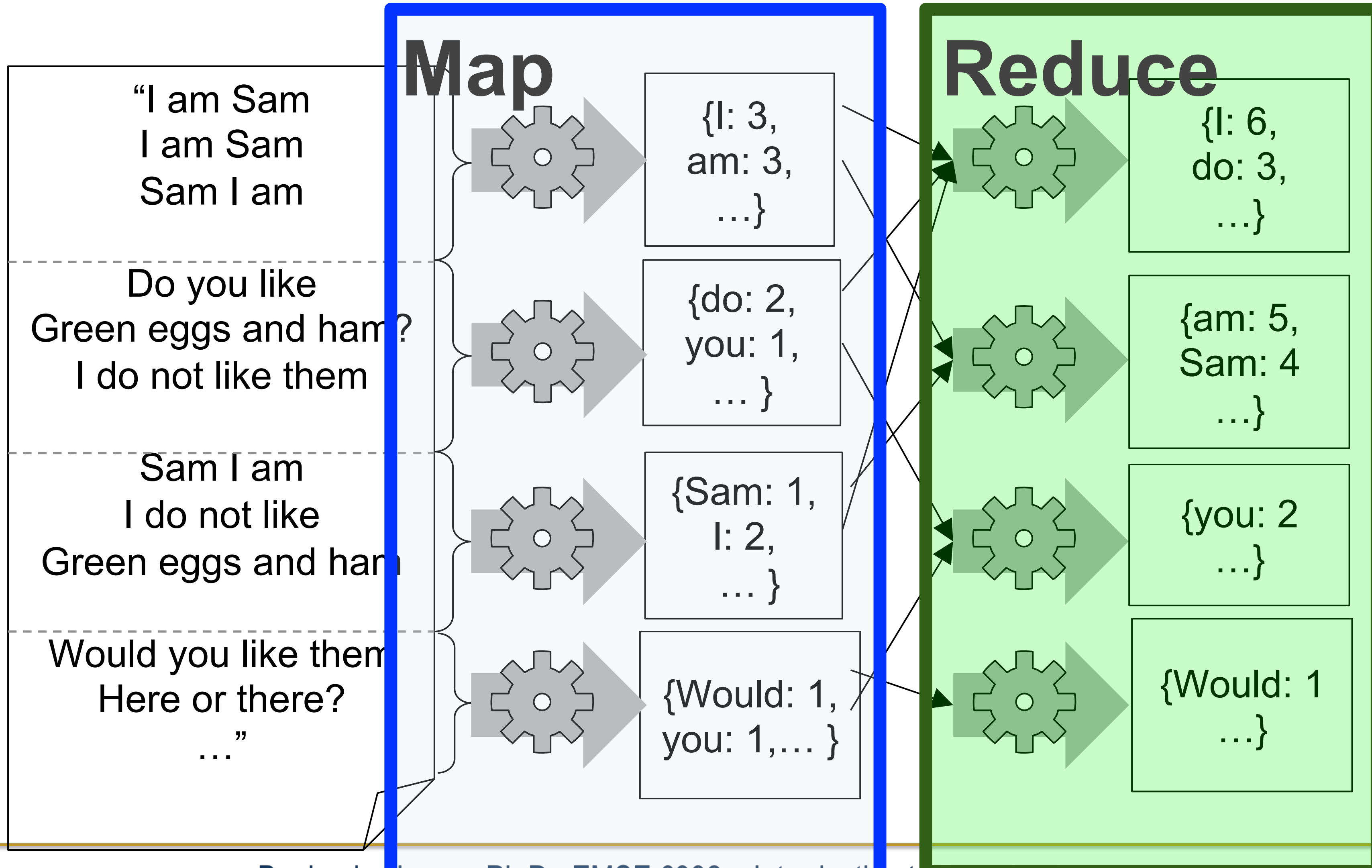
Word Count with one Reducer



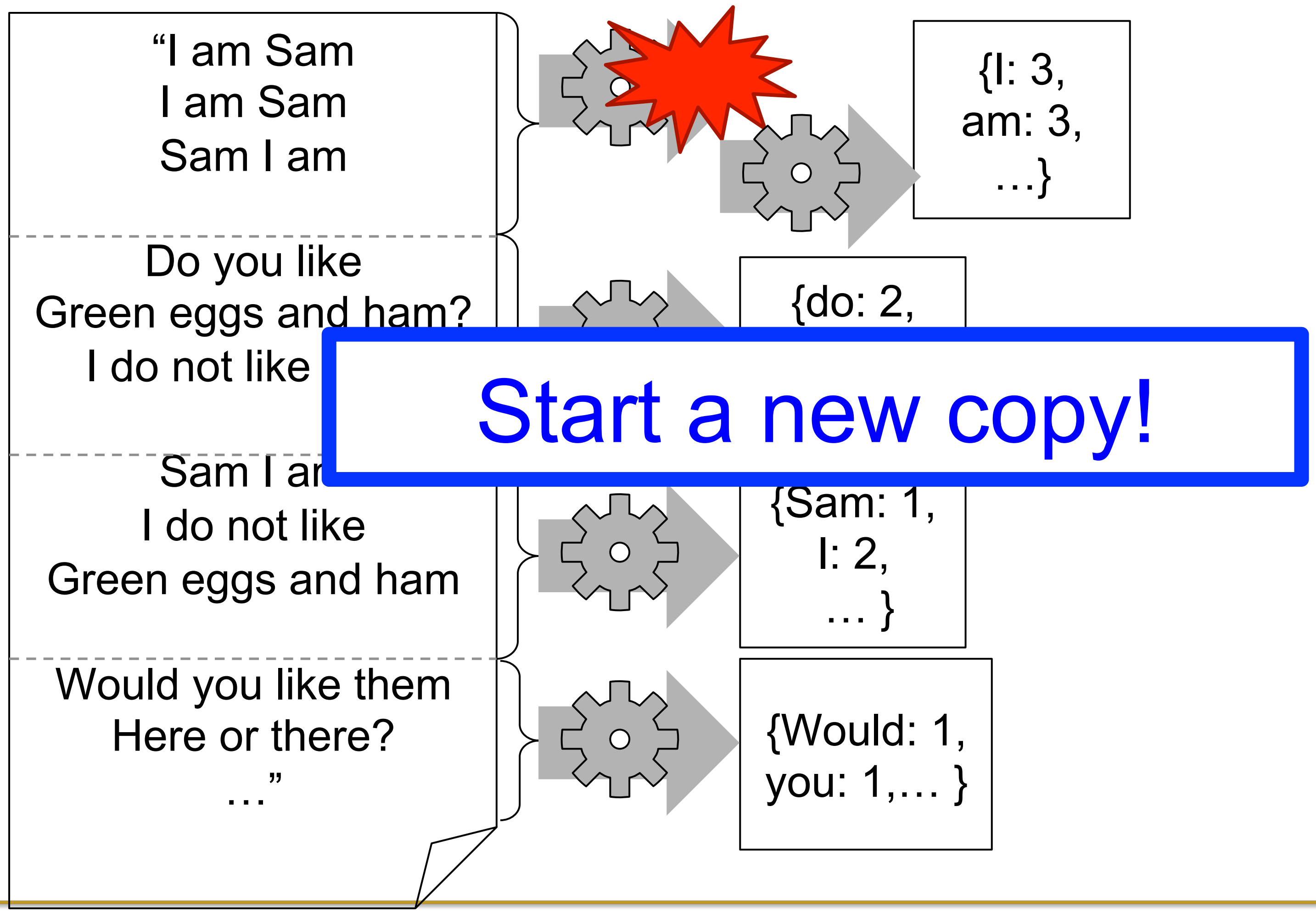
Word Count with Multiple Reducers



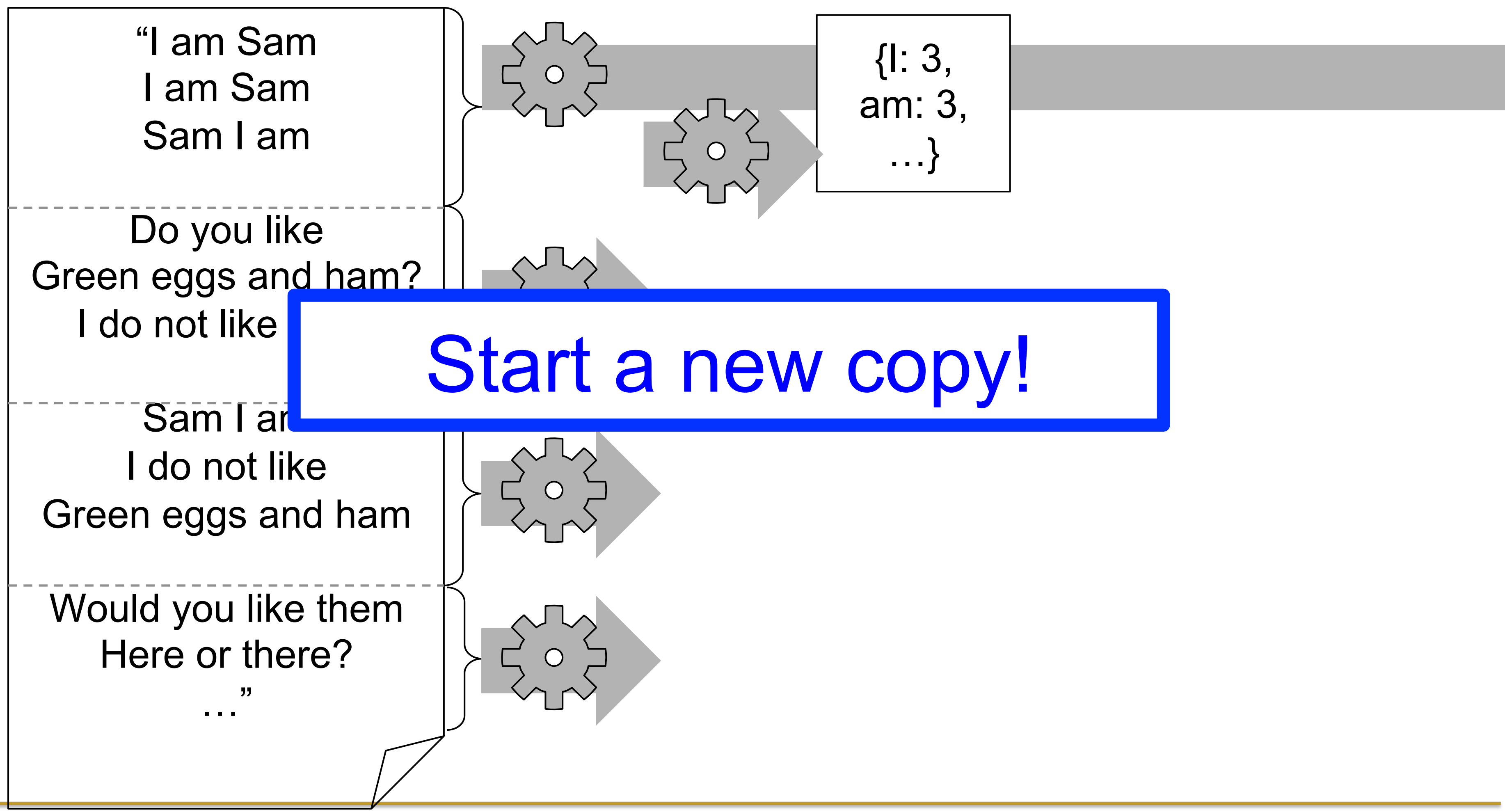
MapReduce: Word Count



MapReduce: Failures?



MapReduce: Slow Tasks



MapReduce: Distributed Execution

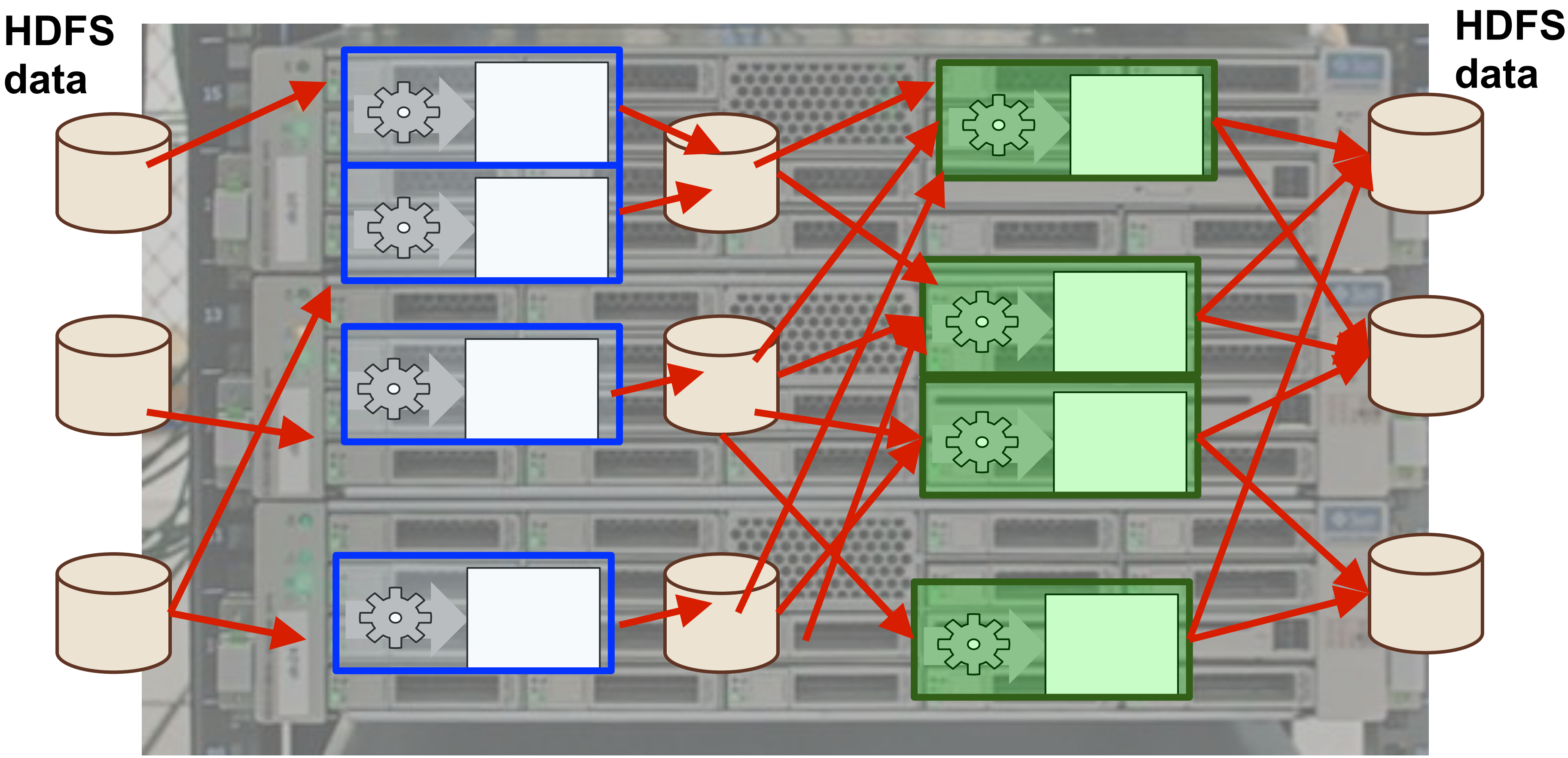


Image: Wikimedia commons (RobH/Tbayer (WMF))

MapReduce for Machine Learning

There are batch algorithms for most Machine Learning tasks:

- process entire dataset, compute gradient, update model

Two kinds of parallel ML algorithm:

1. **Data Parallel:** distributed data, shared model.
2. **Model Parallel:** data and model are distributed.

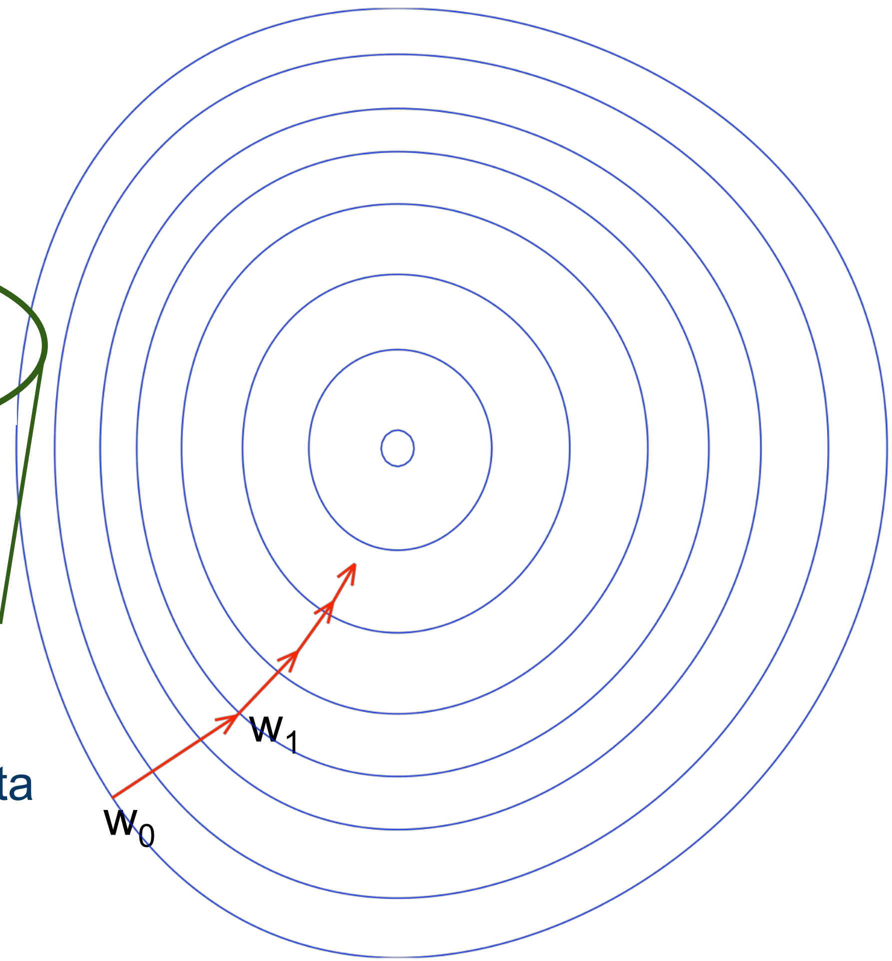
- **Data Parallel** batch algorithms can be implemented in one map-reduce step
- **Model parallel** algorithms require two reduce steps for each iteration (reduce, and then redistribute to each mapper)

Batch Gradient Descent

$$\operatorname{argmin}_{\vec{w}} f(\underbrace{\boxed{\vec{w}}}_{\text{model}}, \underbrace{\boxed{\vec{x}}}_{\text{data}})$$

$$\vec{w}_{i+1} = \vec{w}_i + \gamma \nabla f(\vec{w}, \vec{x})$$

$$\nabla f(\vec{w}, \vec{x}) = \underbrace{\boxed{\sum_{i=1}^{n}}}_{\text{Reduce}} \underbrace{\boxed{g(\vec{w}, x_i)}}_{\text{Map over data}}$$



Gradient Descent on MR

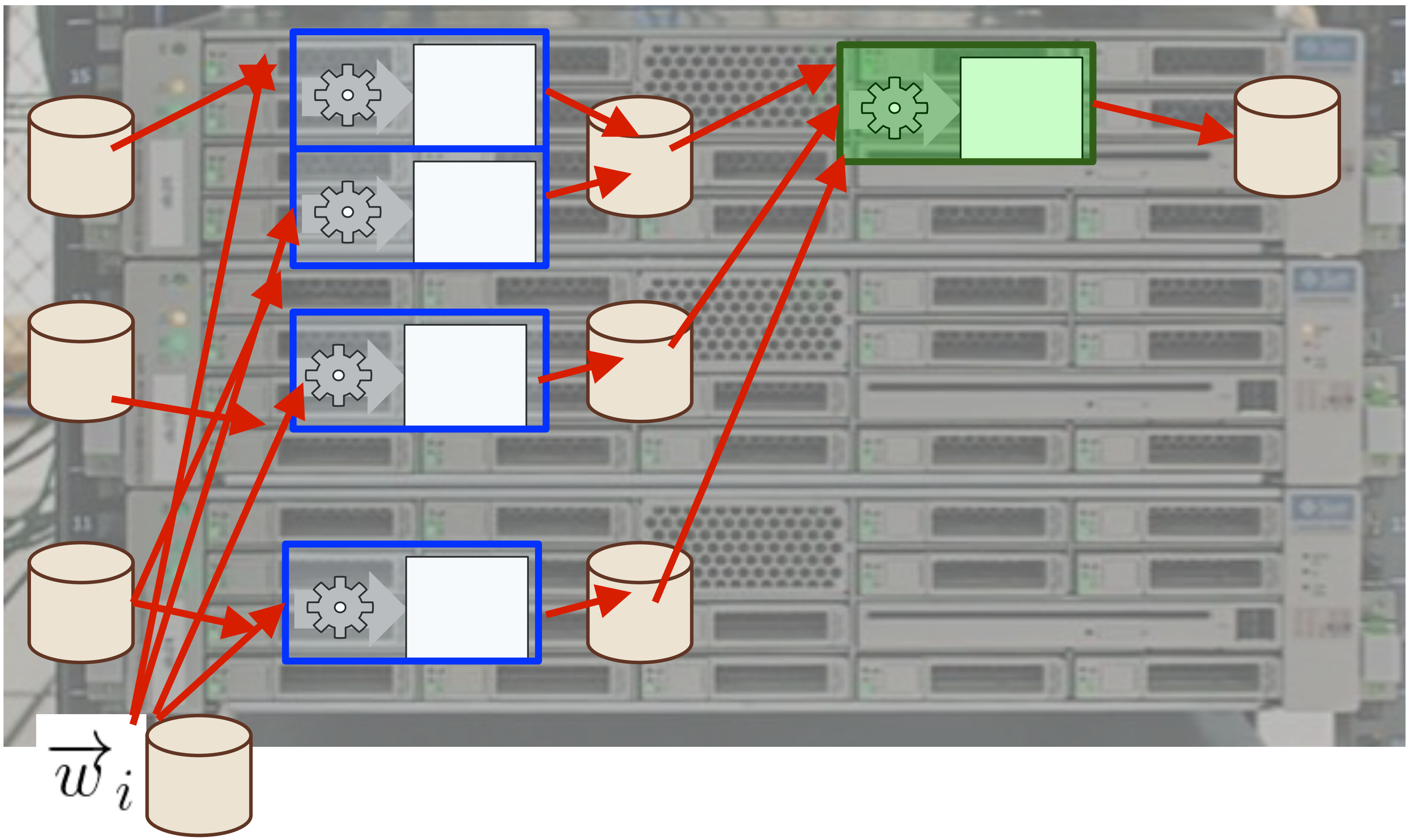


Image: Wikimedia commons (RobH/Tbayer (WMF))

Gradient Descent on MR

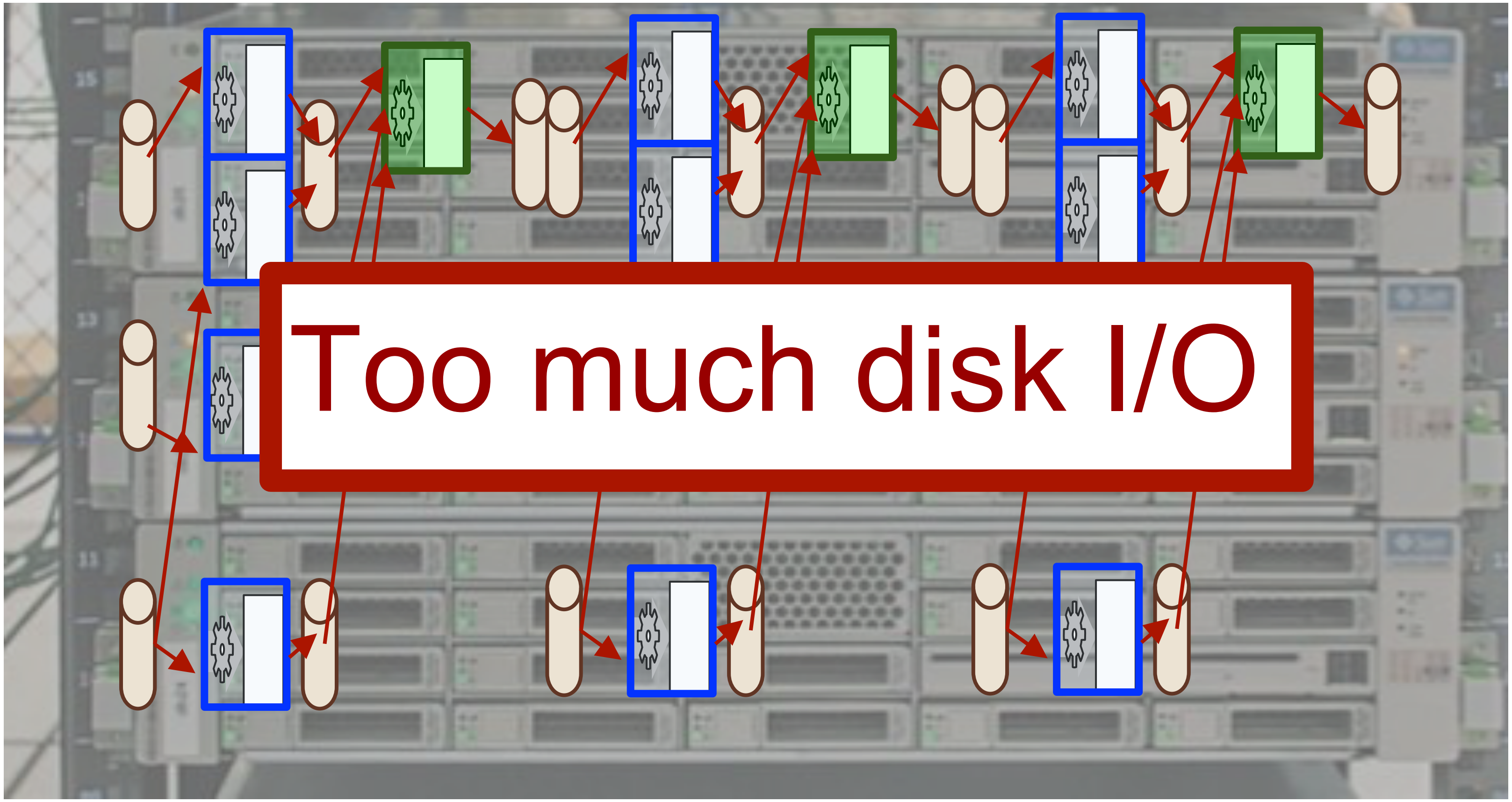
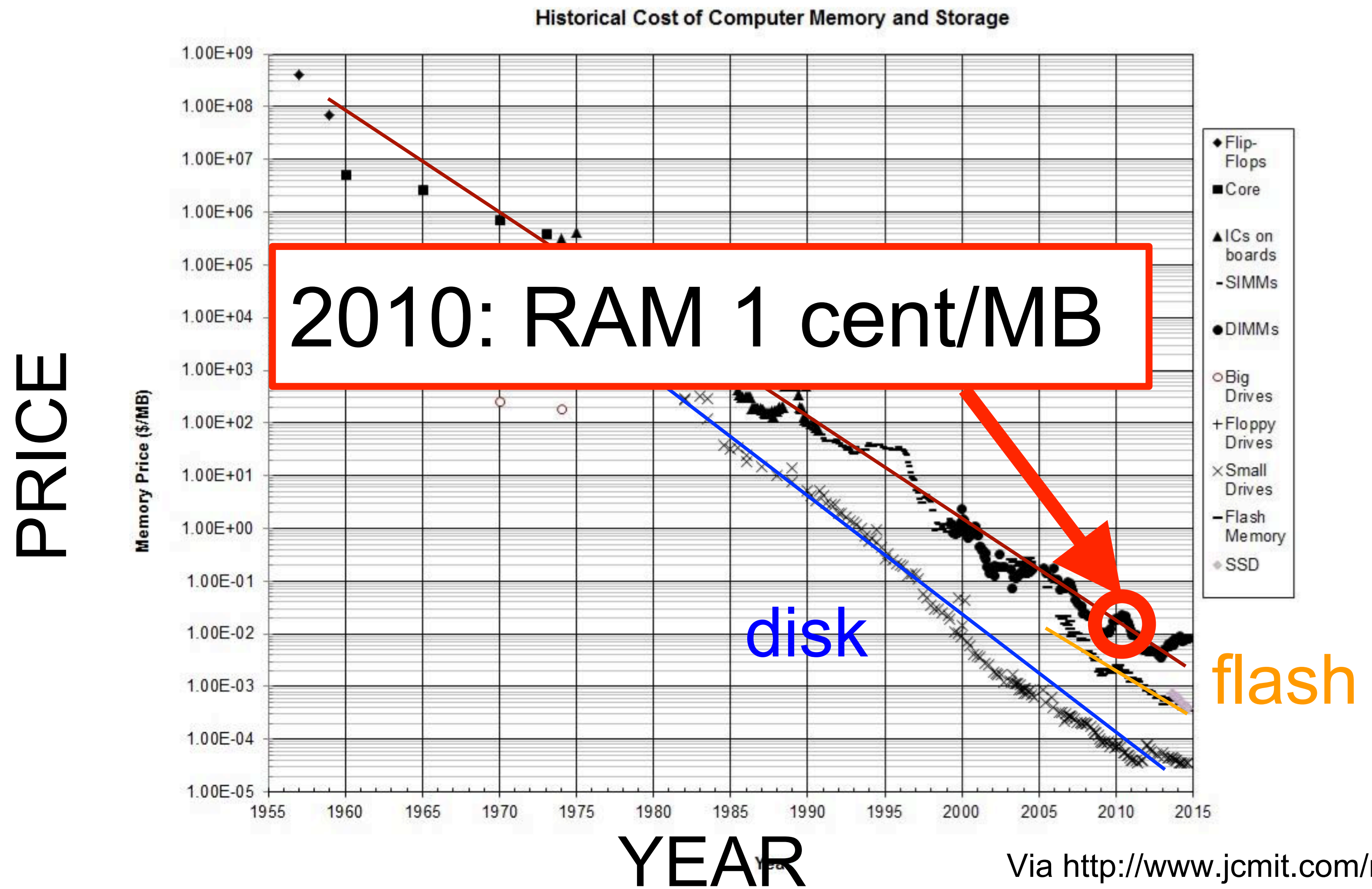


Image: Wikimedia commons (RobH/Tbayer (WMF))

Tech trend: cost of memory



Via http://www.jcmit.com/mem2014.htm

Approaches

- Hadoop
- Spark
- MPI

Spark

Persist data **in-memory:**

- Optimized for batch, data-parallel ML algorithms
- An efficient, general-purpose language for cluster processing of big data
- In-memory query processing (Shark)

Practical Challenges with Hadoop:

- Very **low-level** programming model (Jim Gray)
- Very **little re-use** of Map-Reduce code between applications
- **Laborious** programming: design code, build jar, deploy on cluster
- **Relies heavily on Java reflection** to communicate with to-be-defined application code.

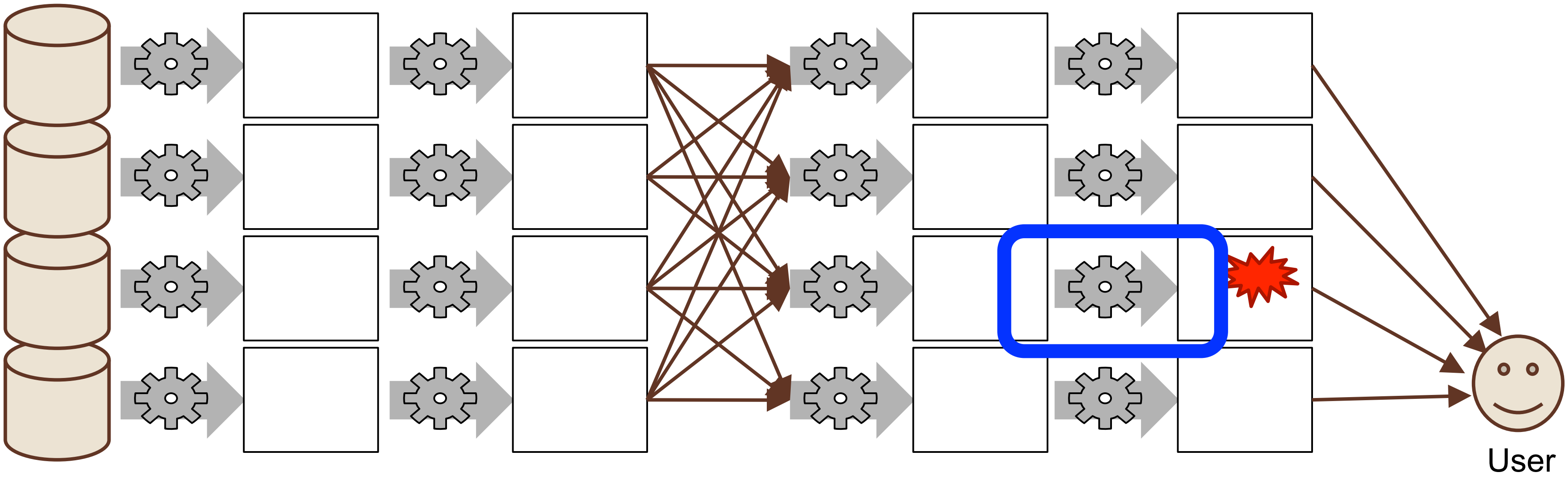
Practical Advantages of Spark:

- **High-level programming model**: can be used like SQL or like a tuple store.
- **Interactivity**.
- **Integrated UDFs** (User-Defined Functions).
- High-level model (**Scala Actors**) for distributed programming.
- **Scala generics** instead of reflection: Spark code is generic over [Key,Value] types.

Spark: Fault Tolerance

Hadoop: Once computed, don't lose it

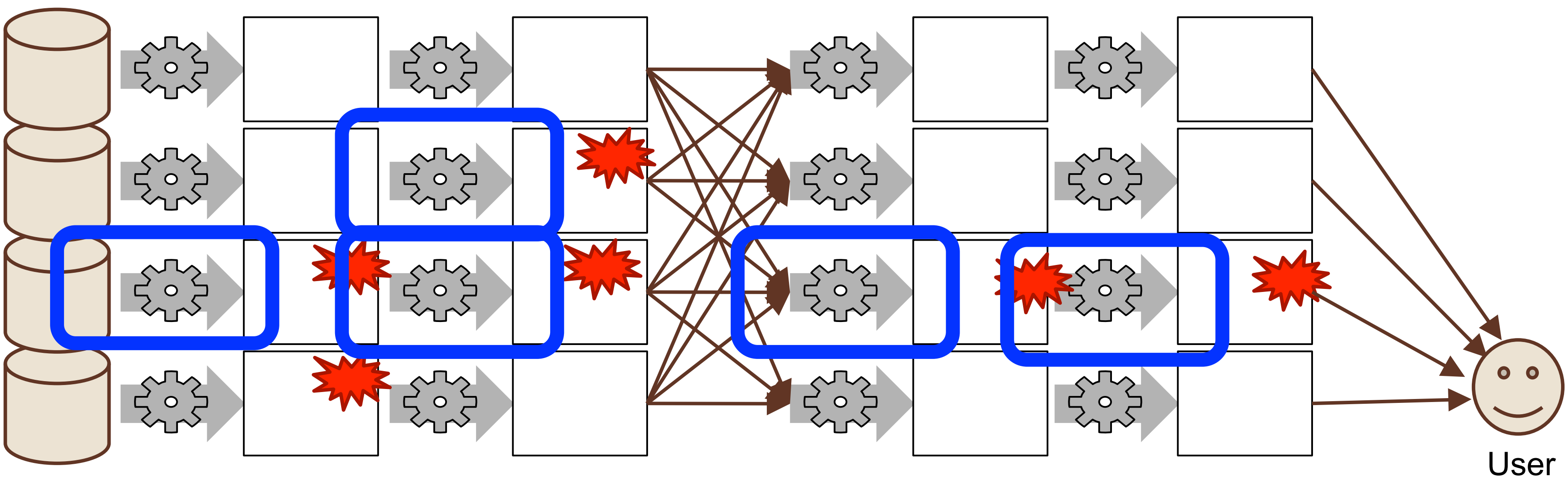
Spark: Remember ***how*** to recompute



Spark: Fault Tolerance

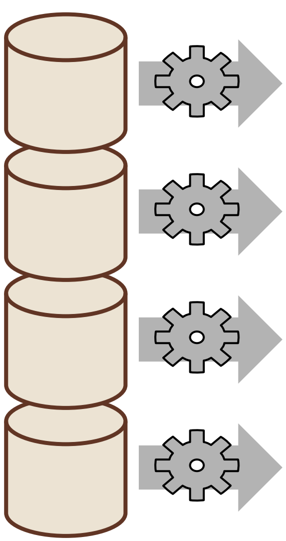
Hadoop: Once computed, don’t lose it

Spark: Remember ***how*** to recompute



Spark programming model (Python)

```
sc = pyspark.SparkContext(...)
raw_ratings = sc.textFile("...", 4)
```

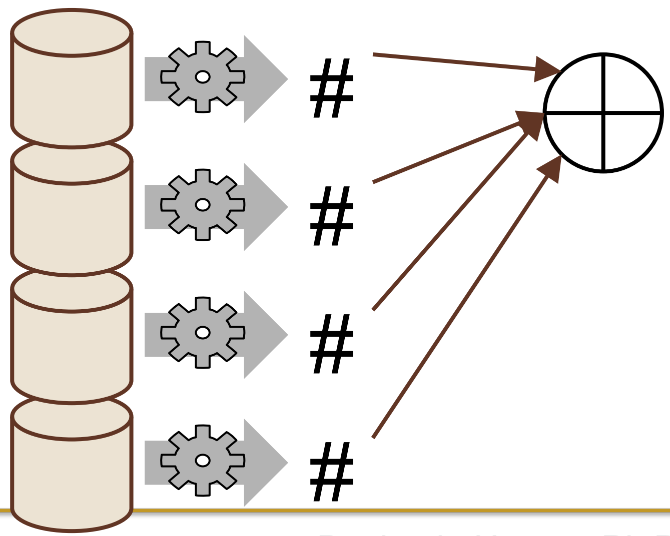


RDD (Resilient Distributed Dataset)

- Distributed array, 4 partitions
- Elements are lines of input
- Computed **on demand**
- Compute = (re)read from input

Spark programming model (Python)

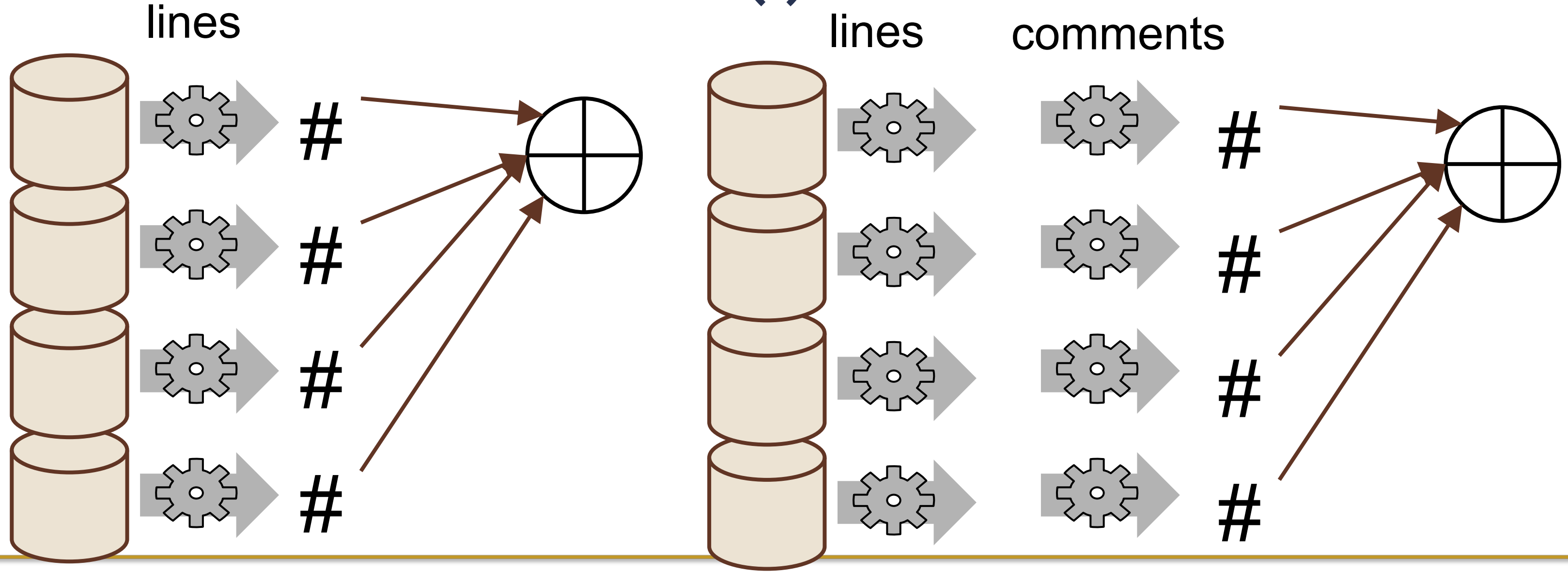
```
lines = sc.textFile("...", 4)
print lines.count()
```



Spark programming model (Python)

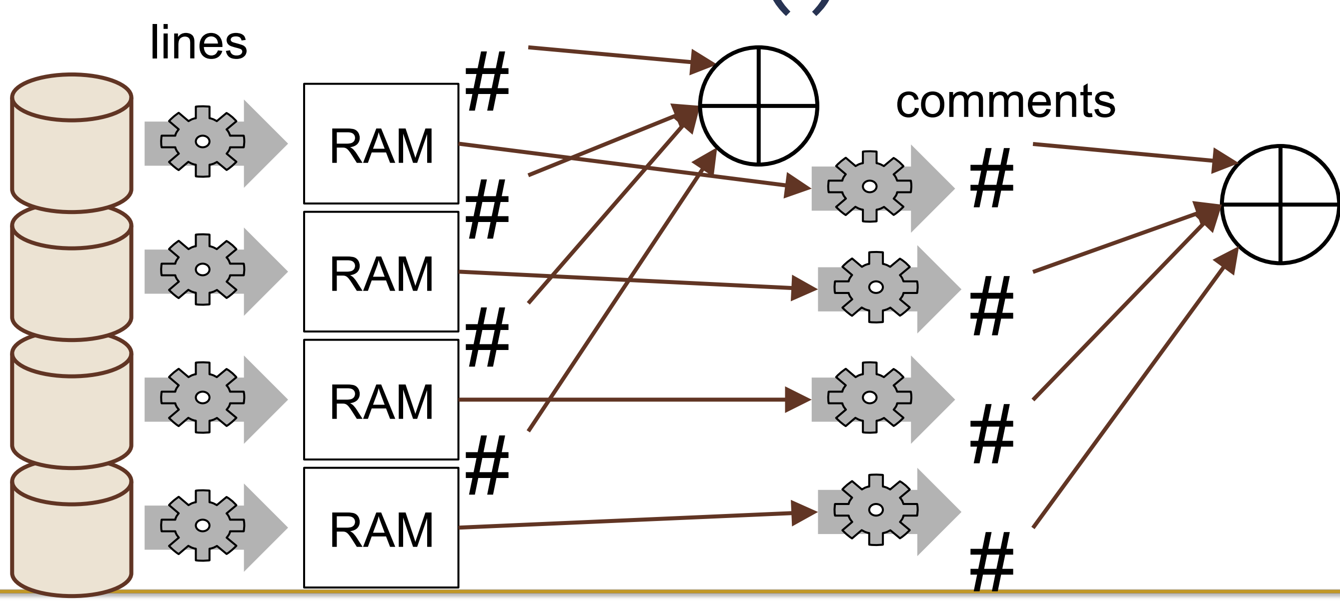
```
lines = sc.textFile("...", 4)

comments = lines.filter(isComment)
print lines.count(),
  comments.count()
```



Spark programming model (Python)

```
lines = sc.textFile("...", 4)
lines.cache() # save, don't
  recompute!
comments = lines.filter(isComment)
print lines.count(),
  comments.count()
```



Other transformations

```
rdd.filter(lambda x: x % 2 == 0)
# [1, 2, 3] → [2]

rdd.map(lambda x: x * 2)
# [1, 2, 3] → [2, 4, 6]

rdd.flatMap(lambda x: [x, x+5])
# [1, 2, 3] → [1, 6, 2, 7, 3, 8]
```

Shuffle transformations

```
rdd.groupByKey()
# [(1,'a'), (2,'c'), (1,'b')] →
#   [(1,['a','b']), (2,['c'])]


rdd.sortByKey()
# [(1,'a'), (2,'c'), (1,'b')] →
#      [(1,'a'), (1,'b'), (2,'c')]
```

Getting data out of RDDs

```
rdd.reduce(lambda a, b: a * b)
# [1,2,3] → 6
rdd.take(2)
# RDD of [1,2,3] → [1,2] # as list
rdd.collect()
# RDD of [1,2,3] → [1,2,3] # as list

rdd.saveAsTextFile(...)
```

Example:
Logistic Regression in PySpark

```
points = sc.textFile(...).map(parsePoint).cache()
w = numpy.random.ranf(size = D) # model vector
for i in range(ITERATIONS):
  gradient = points.map(
    lambda p: (1/(1+exp(-p.y*(w.dot(p.x))))-1)*p.y*p.x
  ).reduce(lambda a, b: a + b)
  w -= gradient
print "Final model: %s" % w
```

Spark's Machine Learning Toolkit

MLLib: Algorithms

Classification

- SVM, Logistic Regression, Decision Trees, Naive Bayes

Regression

- Linear (with L1 or L2 regularization)

Unsupervised:

- Alternating Least Squares
- K-Means
- SVD

- Optimizers
  - Optimization primitives (SGD, L-BGFS)

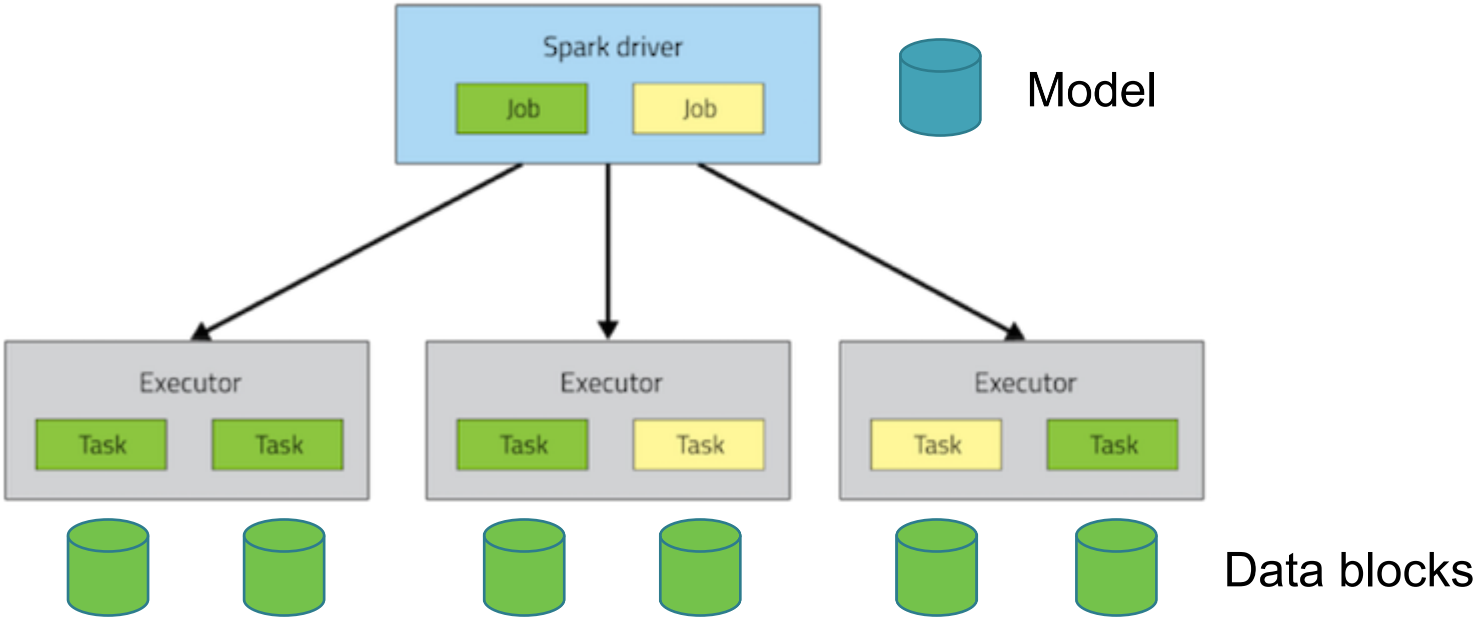
Example: Logistic Regression with MLLib

```
from pyspark.mllib.classification \
    import LogisticRegressionWithSGD

  trainData = sc.textFile("...").map(parsePoint)
  testData = sc.textFile("...").map(...)
model = \
    LogisticRegressionWithSGD.train(trainData)
predictions = model.predict(testData)
```

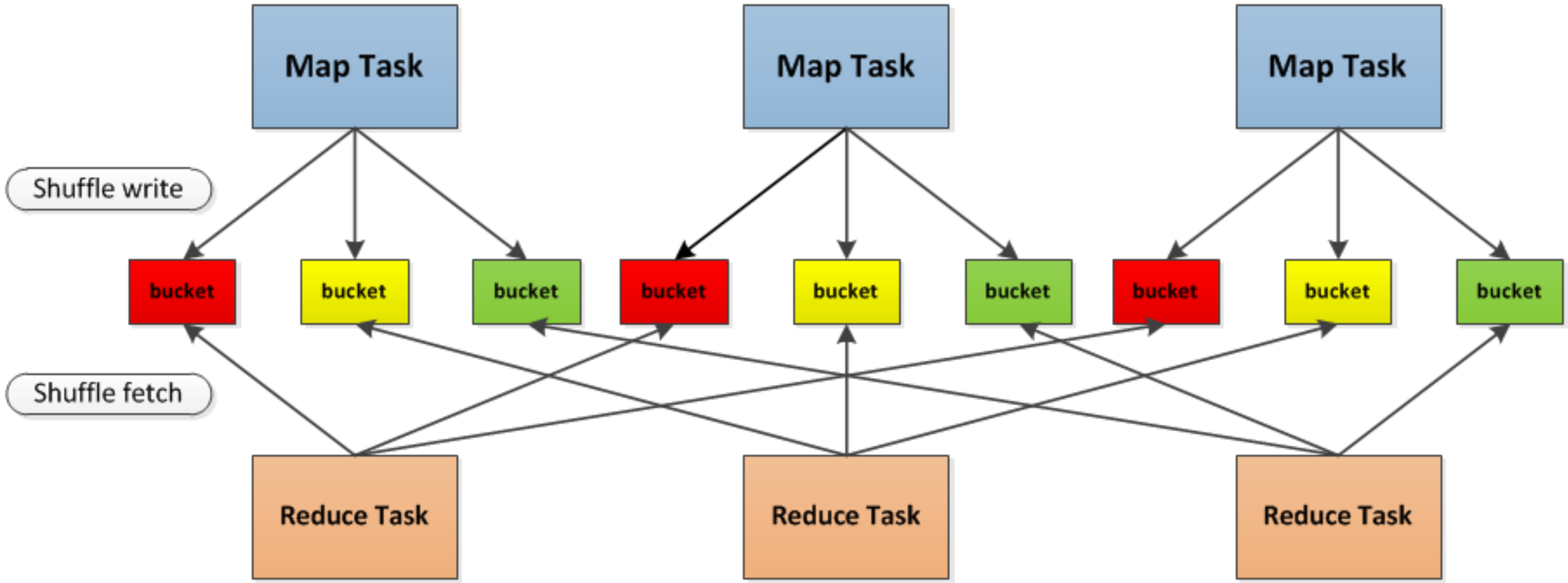
Spark Driver and Executors

- Driver runs user interaction, acts as master for batch jobs
- Driver hosts machine learning models
- Executors hold data partitions
- Tasks process data blocks
- Typically tasks/executor = number of hardware threads



Spark Shuffle

- Used for GroupByKey, Sort, Join operations
- Map and Reduce tasks run on executors
- Data is partitioned by key on by each mapper, saved to buckets, then forwarded to appropriate reducer using a key => reducer mapping function.



Architectural Consequences

- **Simple programming**: Centralized model on driver, broadcast to other nodes.
- Models must fit in single-machine memory, i.e. Spark supports **data parallelism** but not model **parallelism**.
- Heavy load on the driver. Model **update time grows** with number of nodes.
- Cluster performance on most ML tasks on par with single-node system with GPU.
- Shuffle performance is similar to Hadoop, but still improving.

Other uses for MapReduce/Spark

Non-ML applications:

Data processing:

- Select columns
- Map functions over datasets
- Joins
- GroupBy and Aggregates

- Spark admits 3 usage modes:
- Type queries interactively, use :replay
- Run (uncompiled) scripts
- Compile Scala Spark code, use interactively or in batch

Other notable Spark tools

SQL-like query support (Shark, Spark SQL)

BlinkDB (approximate statistical queries)

Graph operations (GraphX)

Stream processing (Spark streaming)

KeystoneML (Data Pipelines)

5 Min Break

Approaches

- Hadoop
- Spark
- MPI

Optimizing Parameter Servers → MPI

- Drop **client-data-push** in favor of **server pull**:
- No need for synch or locking on server.
- Use relaxed synchronization instead.

- The result is a version of MPI (**Message Passing Interface**), a protocol used in scientific computing.

- For cluster computing MPI needs to be modified to:
- Support pull/push of a subset of model data.
- Allow loose synchronization of clients.
- Some dropped data and timeouts.

- Good current research topic!

Summary

- Why Big Data (and Big Models)?
- Hadoop
- Spark
- Parameter Server and MPI