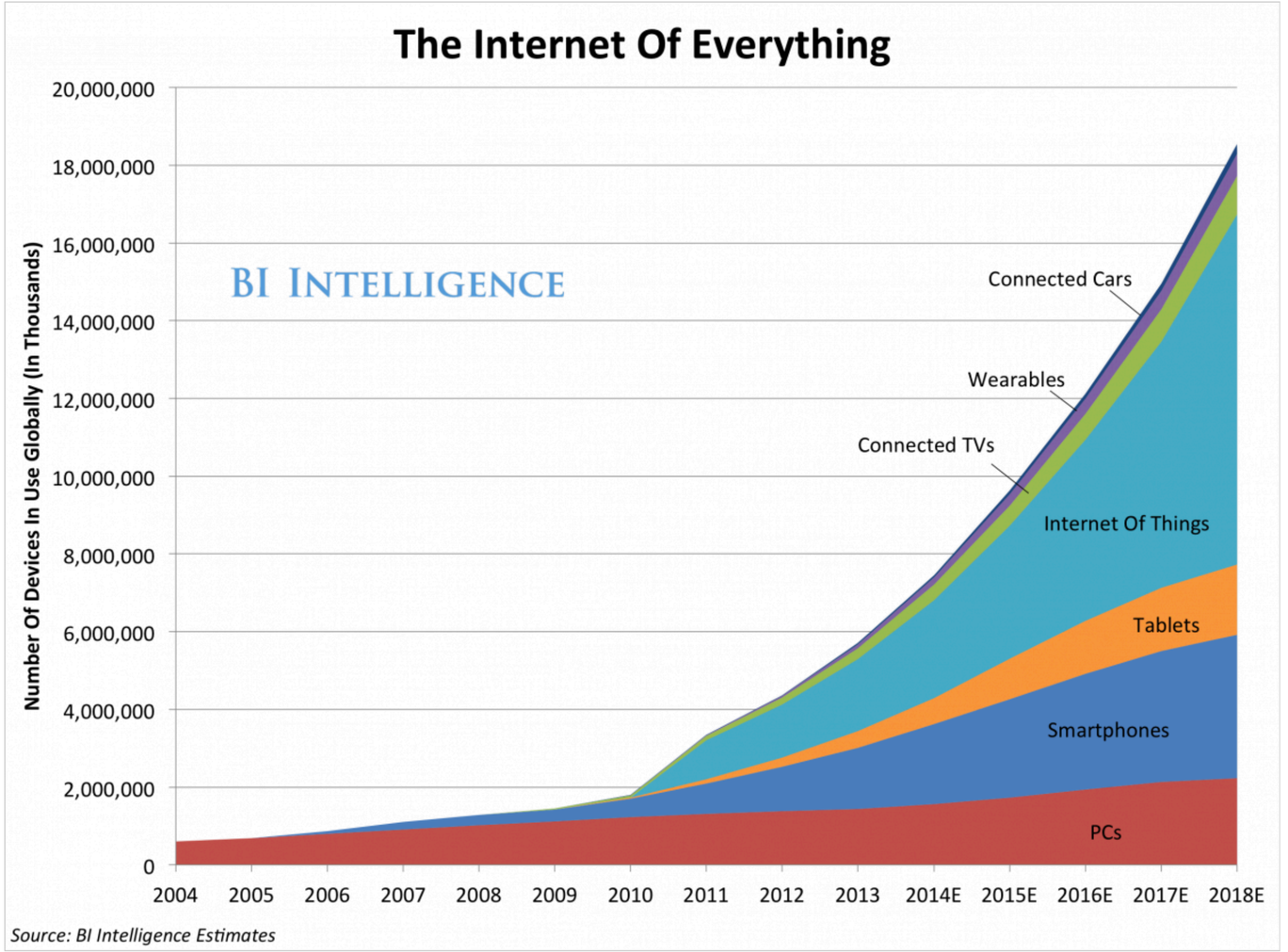
Apache Spark Crashcourse

Malyutin Eugeny

The Big Data Problem

- Data growing faster than computation speeds
- Growing data sources
  - Web, mobile, scientific, …
- Storage getting cheaper
  - Size doubling every 18 months
- But, stalling CPU speeds and storage bottlenecks



Big Data Examples

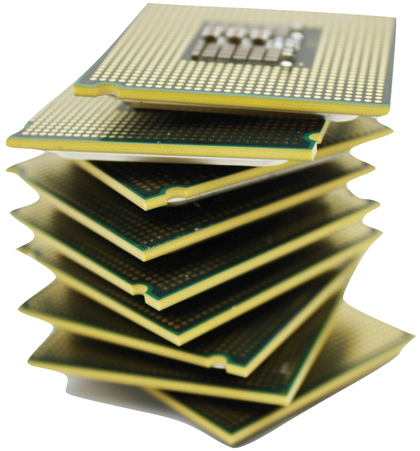
- Facebook's daily logs: **60 TB**
- 1,000 genomes project: **200 TB**
- Google web index: **10+ PB**
- Cost of 1 TB of disk: **~$35**
- Time to read 1 TB from disk: **3 hours (100 MB/s)**

Our tools:

- Pandas
- R/Python
- SQL / scikitlearn
- …
- **Runs in single machine**


Google datacenter
How can we program it?

Hardware for Big Data



Lot of hard drive

And lot of CPU’s

Hardware for Big Data

One big box?
(1990's solution)

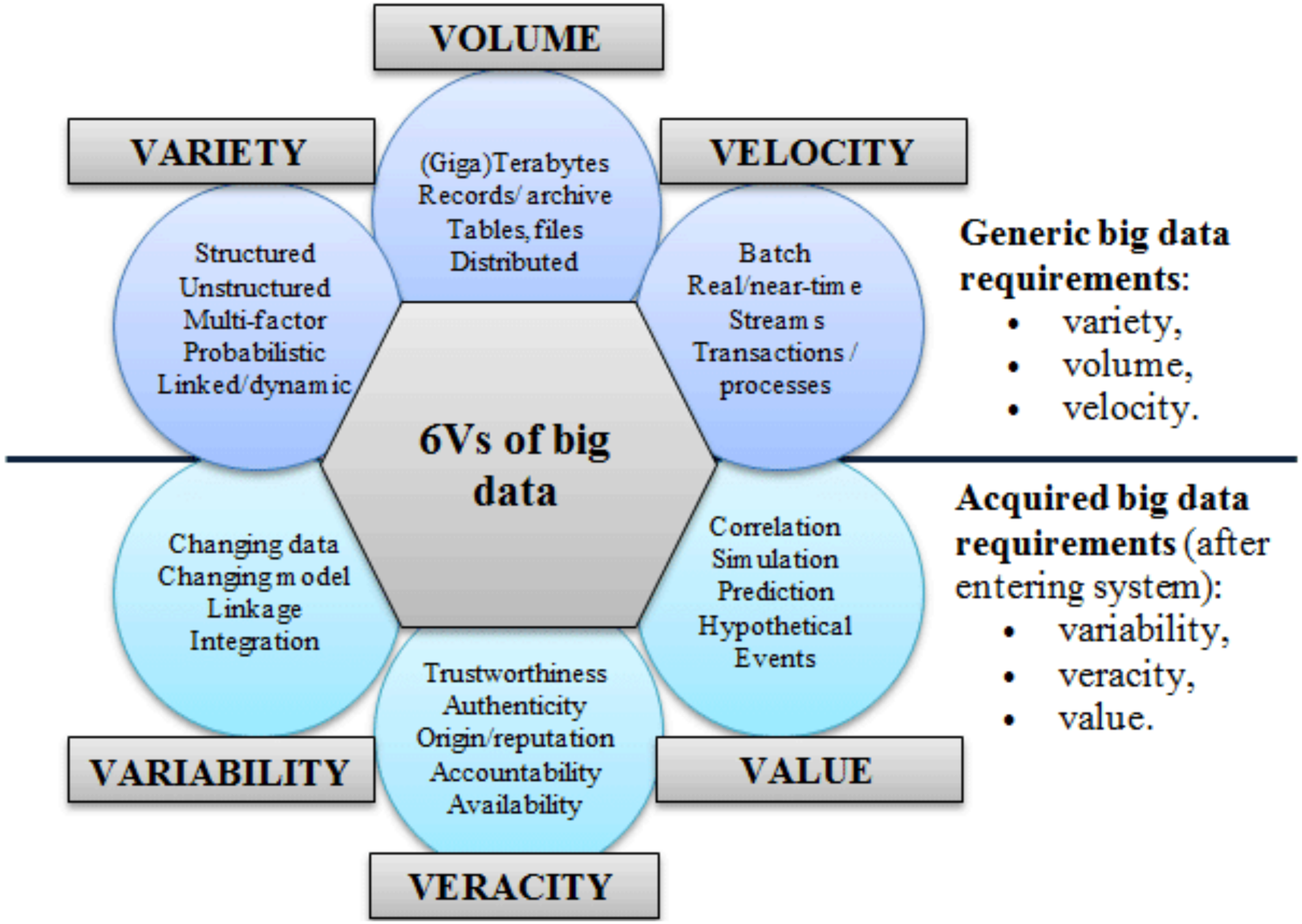
But,
expensive
Low volume
All "premium" hardware

And, still not big enough!

Hardware for Big Data

- Consumer-grade hardware
  - Not “gold plated”
- Many desktop-like servers
  - Easy to add capacity
  - Cheaper per CPU/disk
  - Complexity in software

Big data (marketing):



Problems with cheap hardware

Failures, Google's numbers:

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

Network speeds versus shared memory:

- Much more **latency**
- Network **slower than storage**
- **Uneven** performance

How do you count the number of occurrences of each word in a document?

“I am Sam
I am Sam
Sam I am
Do you like Green eggs and ham?”

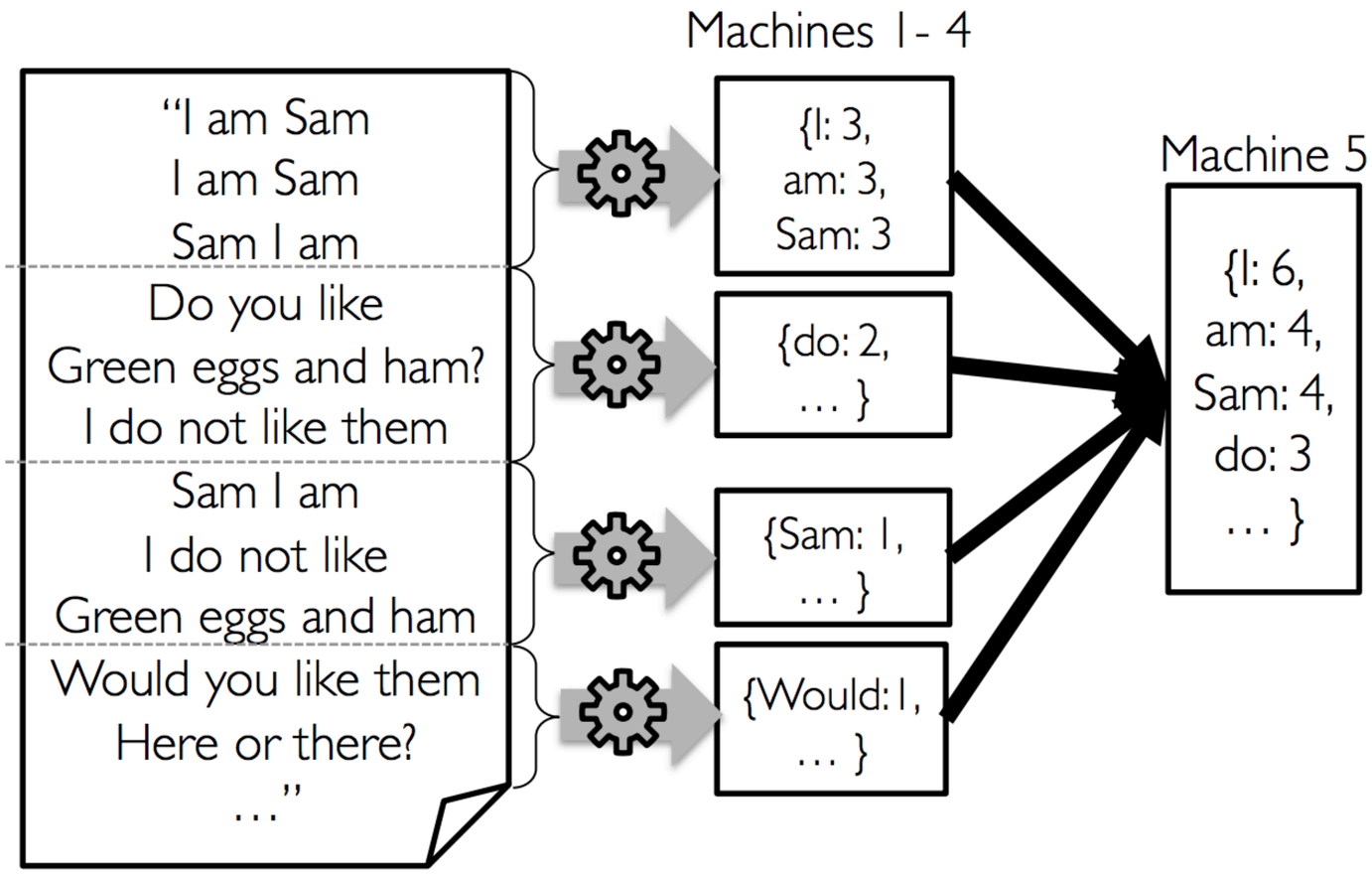
I: 3
am: 3
Sam: 3
do: 1
you: 1
like: 1
…

1st approach: hash table

"I am Sam
I am Sam
Sam I am
Do you like
Green eggs and ham?"

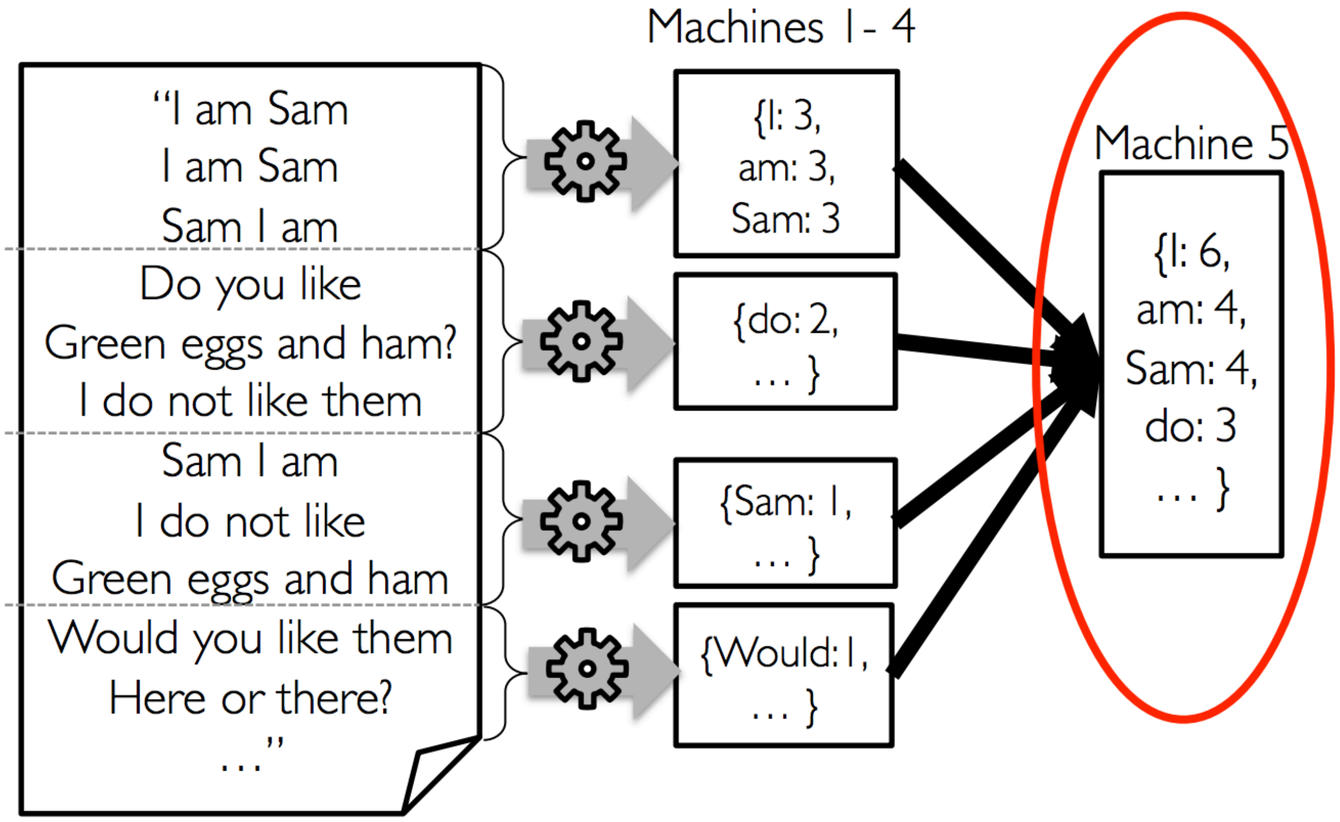
{I: 2,
am: 1,
Sam: 1}

2nd approach:
What if the Document is Really Big?



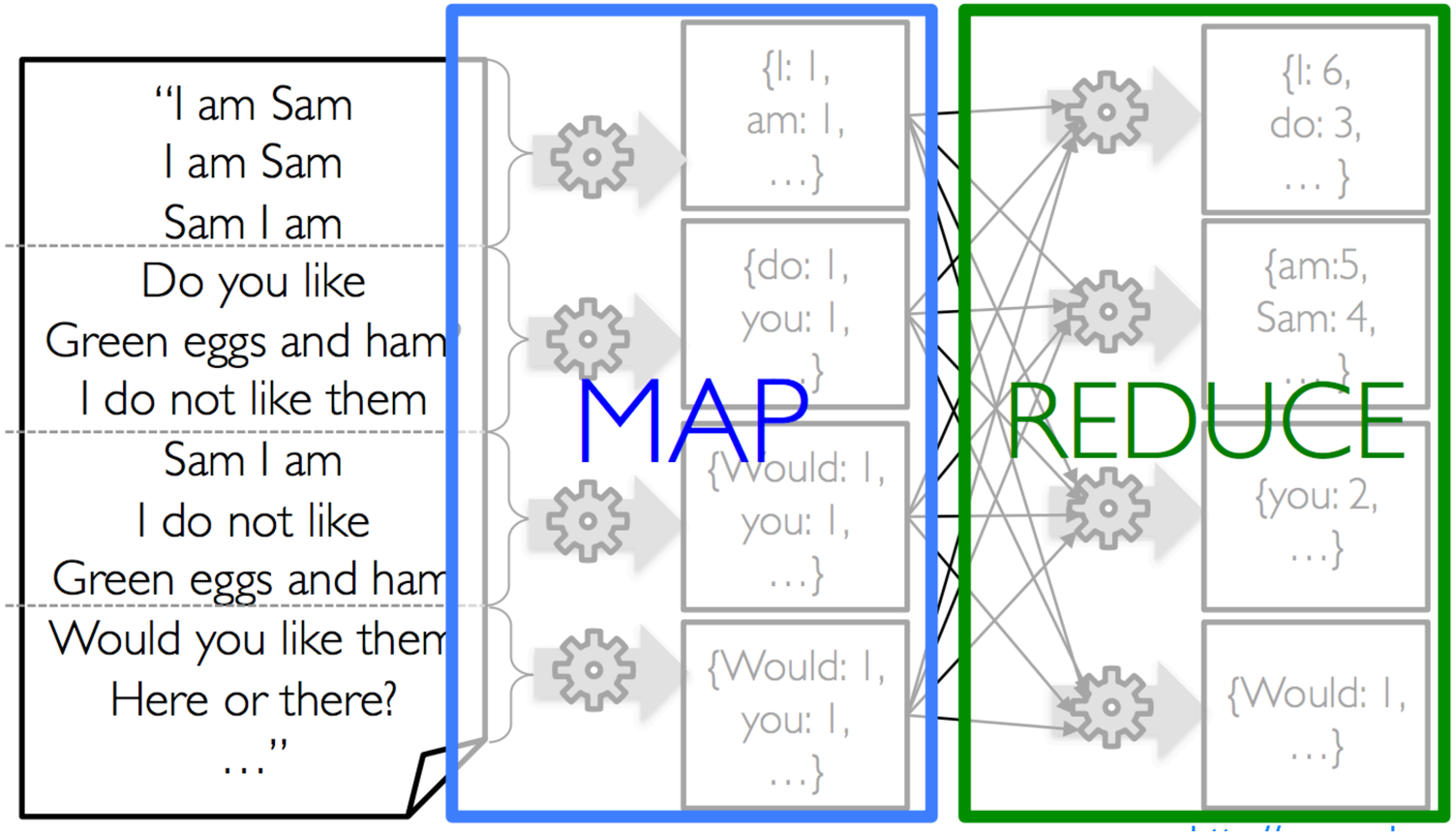
What's the problem with this approach?

What if the Document is Really Big?



Results have to fit on one machine

Map Reduce for Sorting



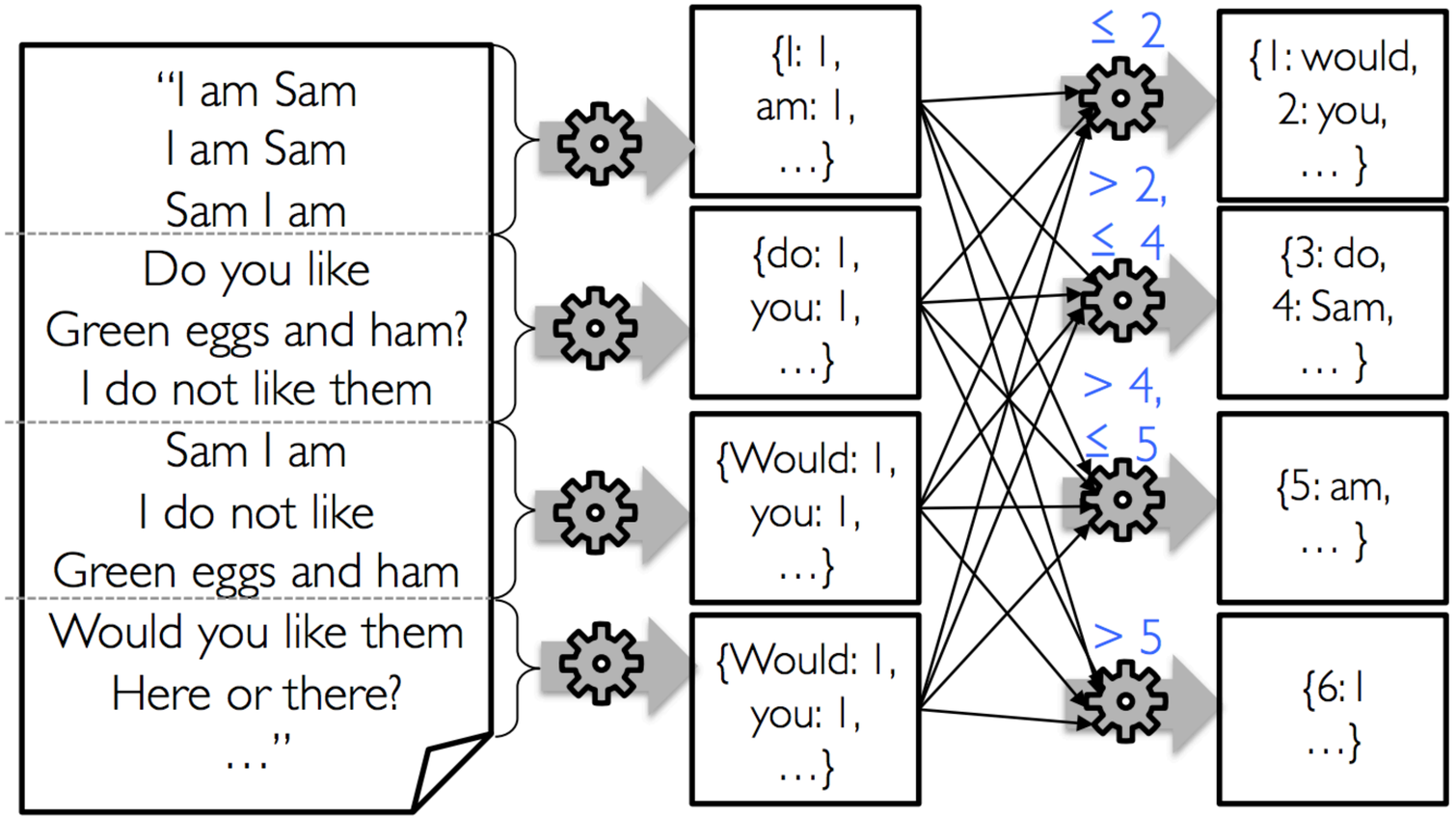
map

In the mapper, user-provided code is executed on each key/value pair from the record reader to produce zero or more new key/value pairs, called the intermediate pairs.

reduce

The reducer takes the grouped data as input and runs a `reduce`function once per key grouping. The function is passed the key and an iterator over all of the values associated with that key.

Map Reduce for Sorting

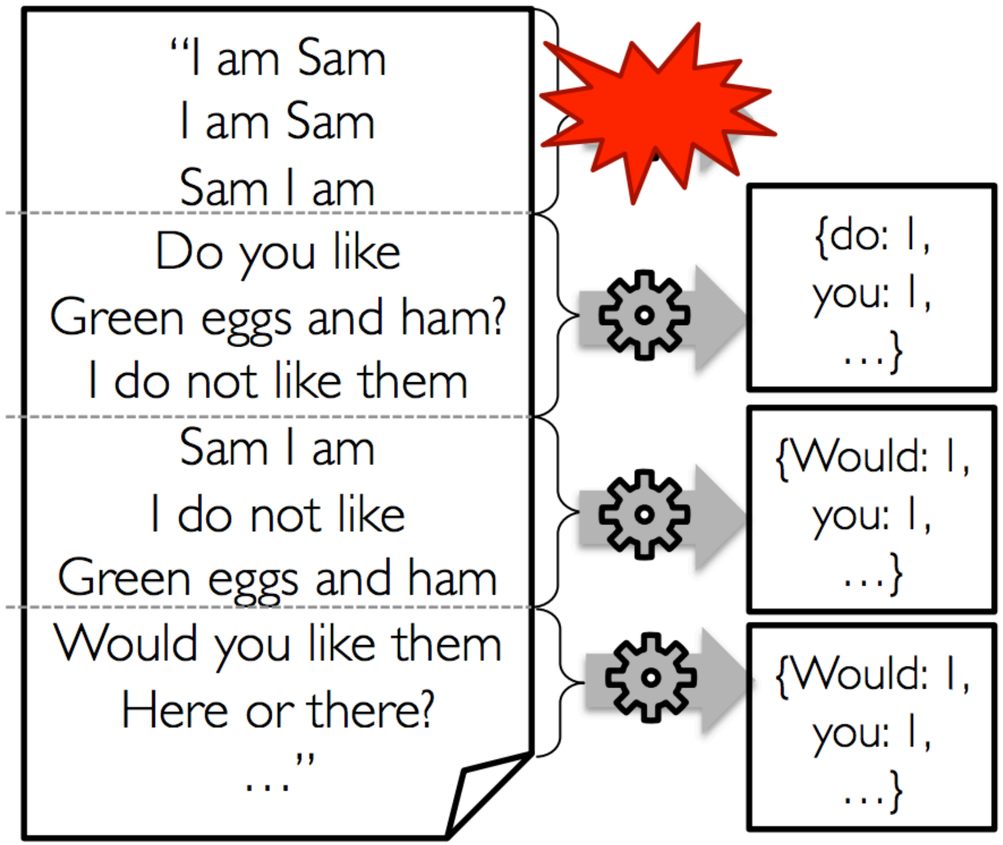


"What word is used most?"

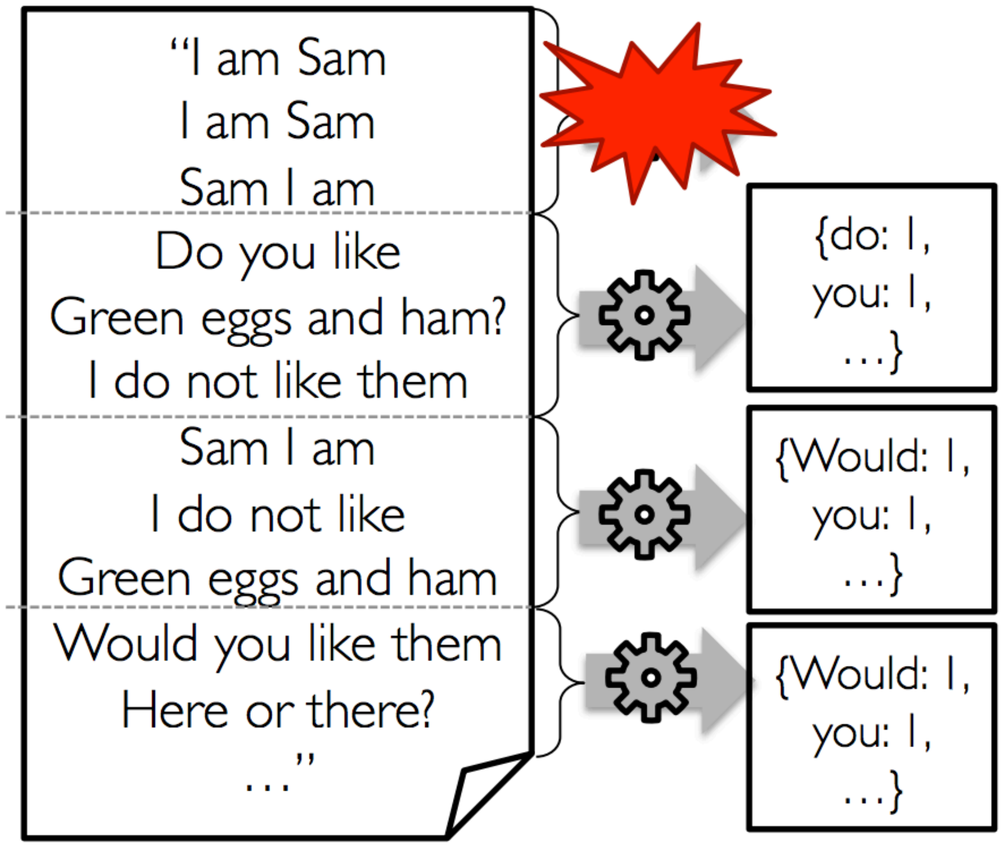
What’s Hard About Cluster Computing?

- How to divide work across machines?
  - Must consider network, data locality
  - Moving data may be very expensive
- How to deal with failures?
  - 1 server fails every 3 years = with 10,000 nodes see 10 faults/day
  - Even worse: stragglers (not failed, but slow nodes)

How Do We Deal with Failures?

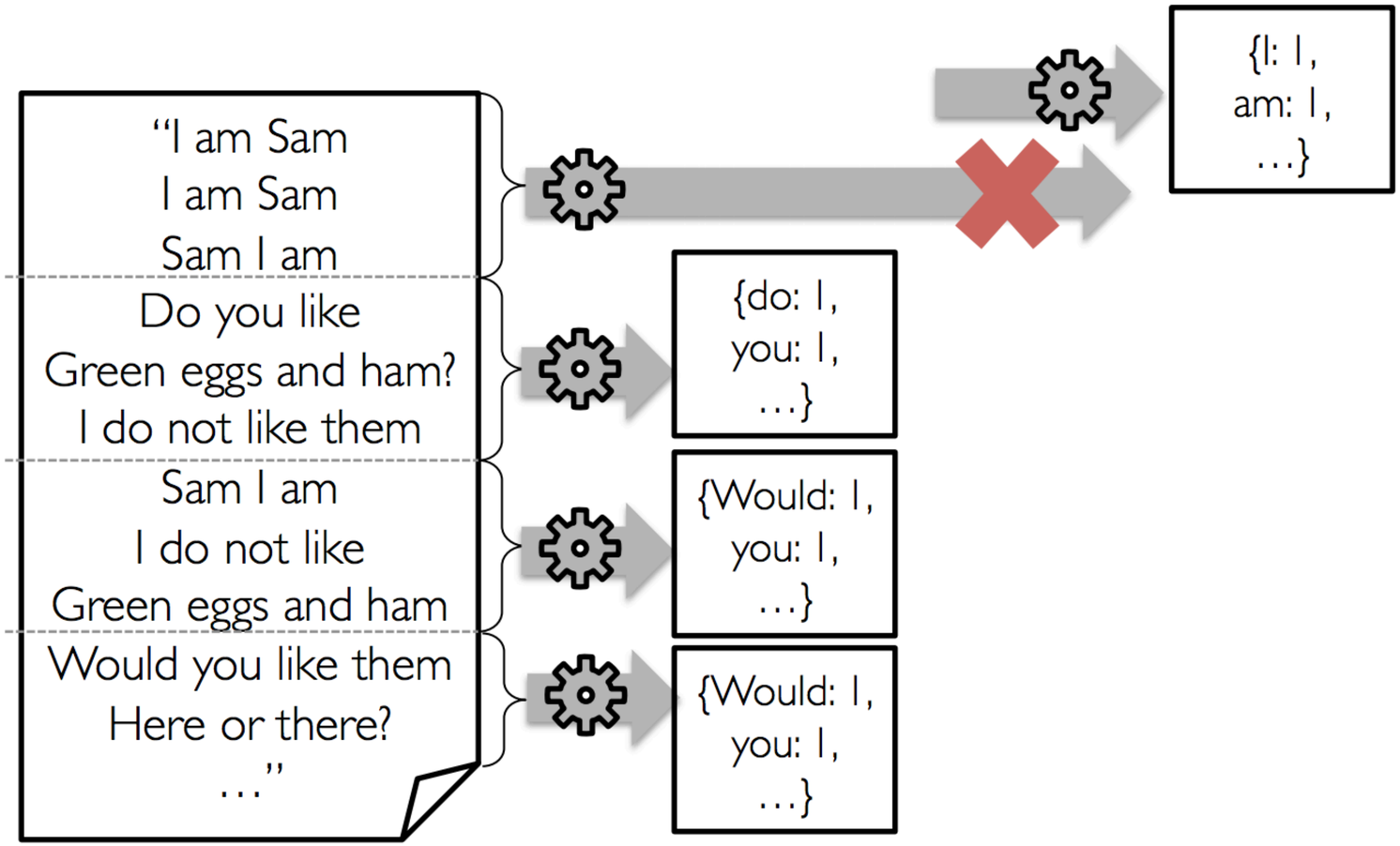


How Do We Deal with Failures?

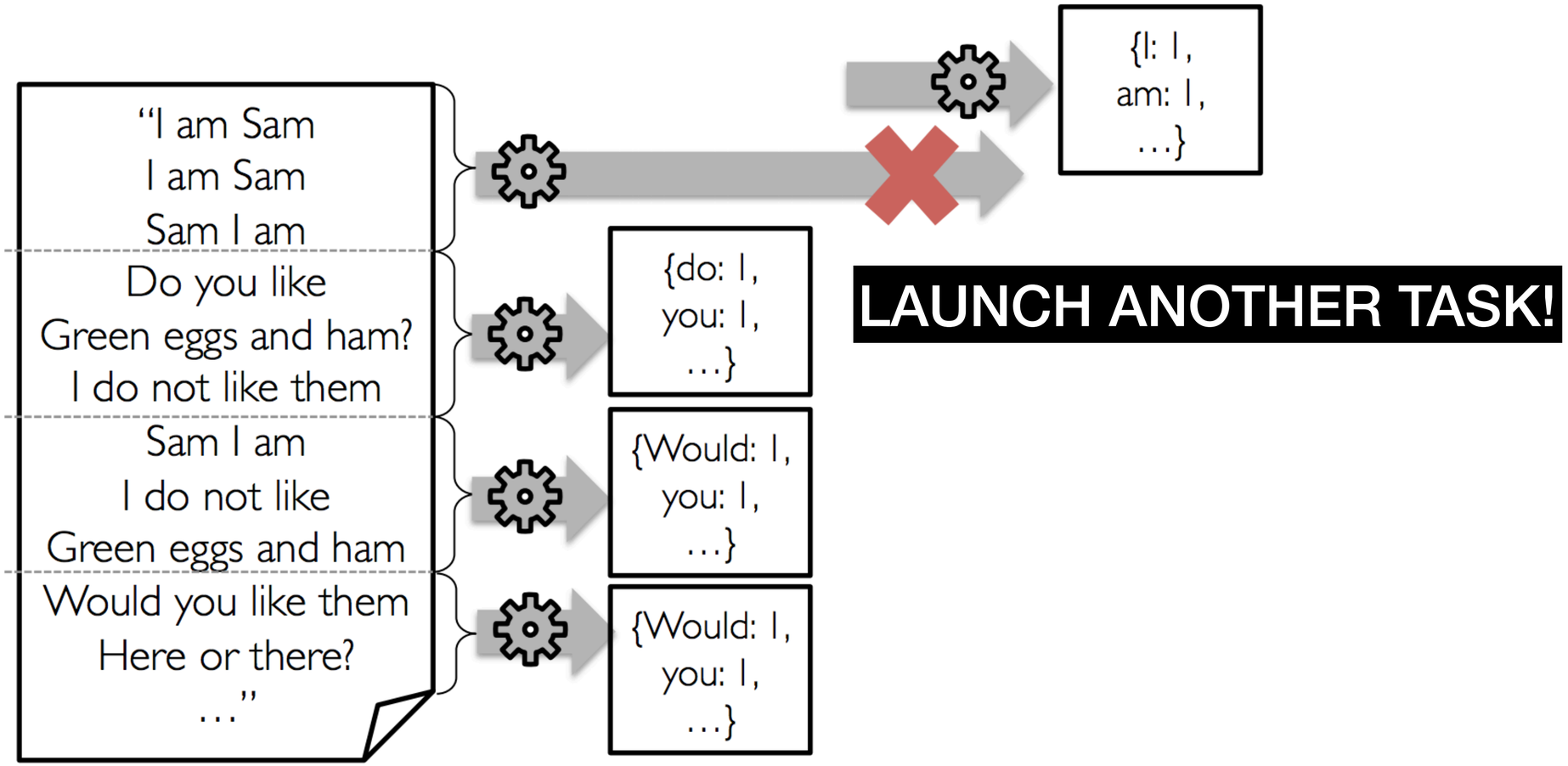


LAUNCH ANOTHER TASK!

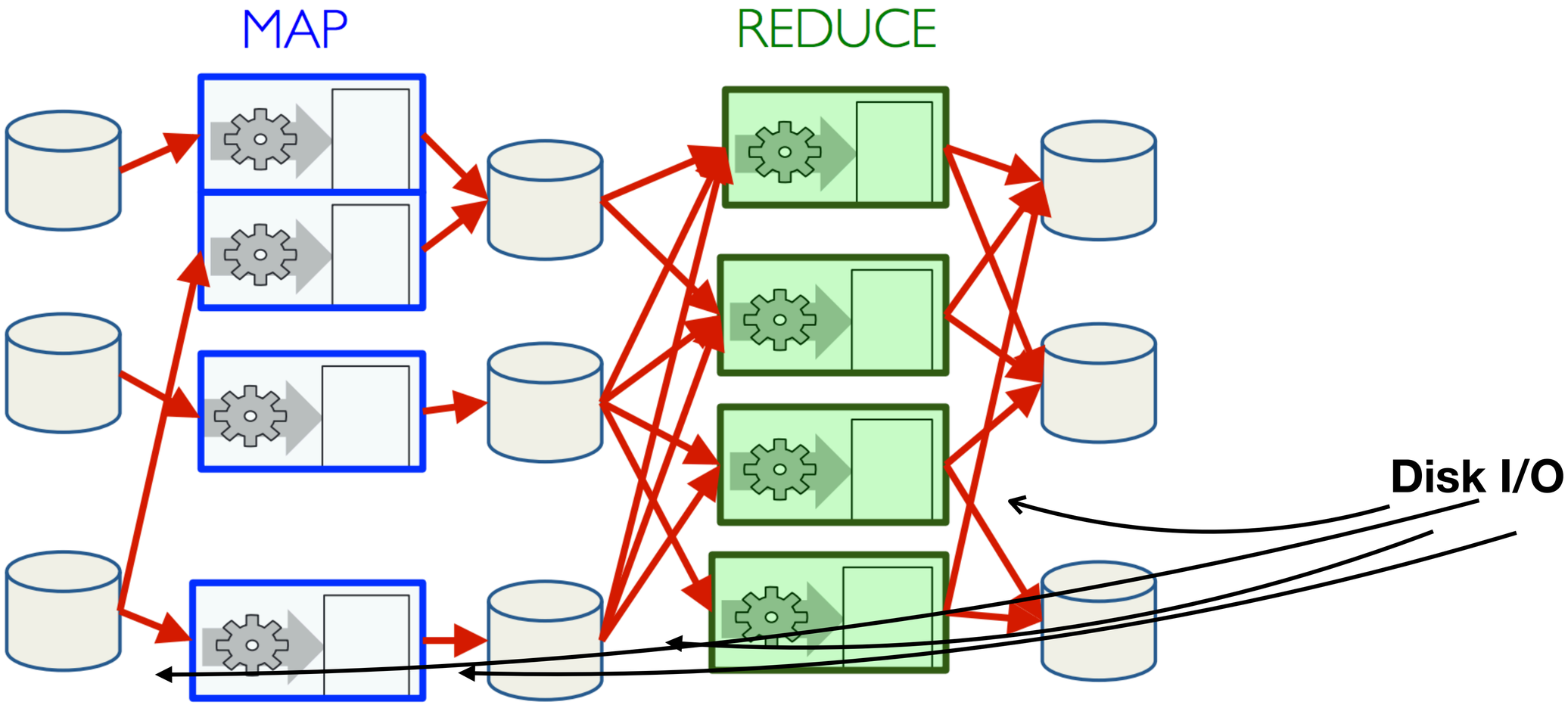
How Do We Deal with Struggles?



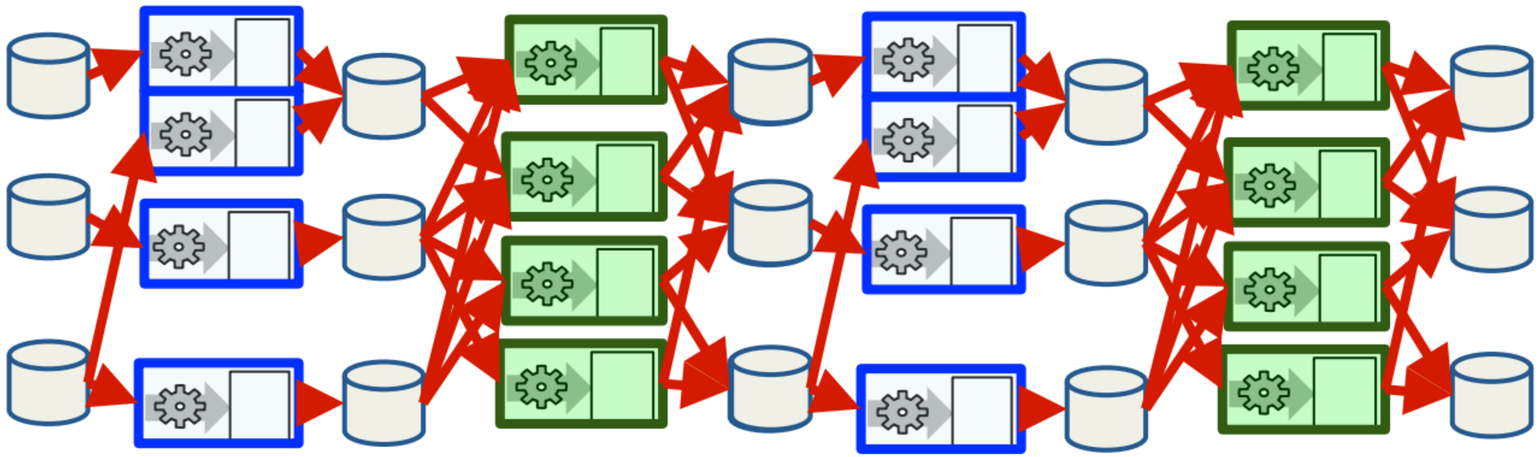
How Do We Deal with Struggles?

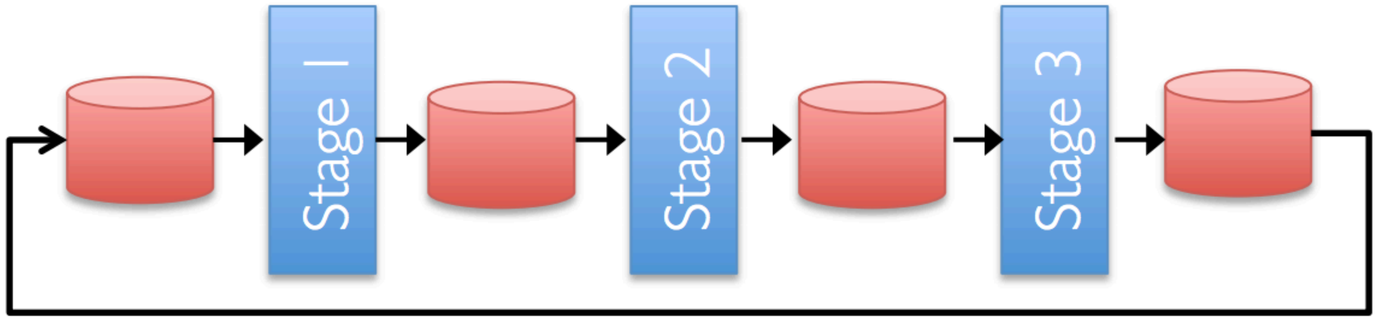


Map Reduce and Disk I/O



Map Reduce and Disk I/O



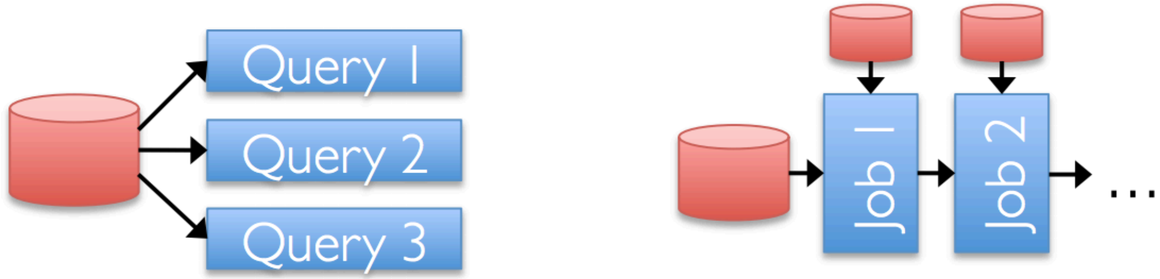


Iterative jobs run multiple stages

Disk I/O is very slow

Apache Spark Motivation

Using Map Reduce for complex jobs, interactive queries and online processing involves lots of disk I/O!

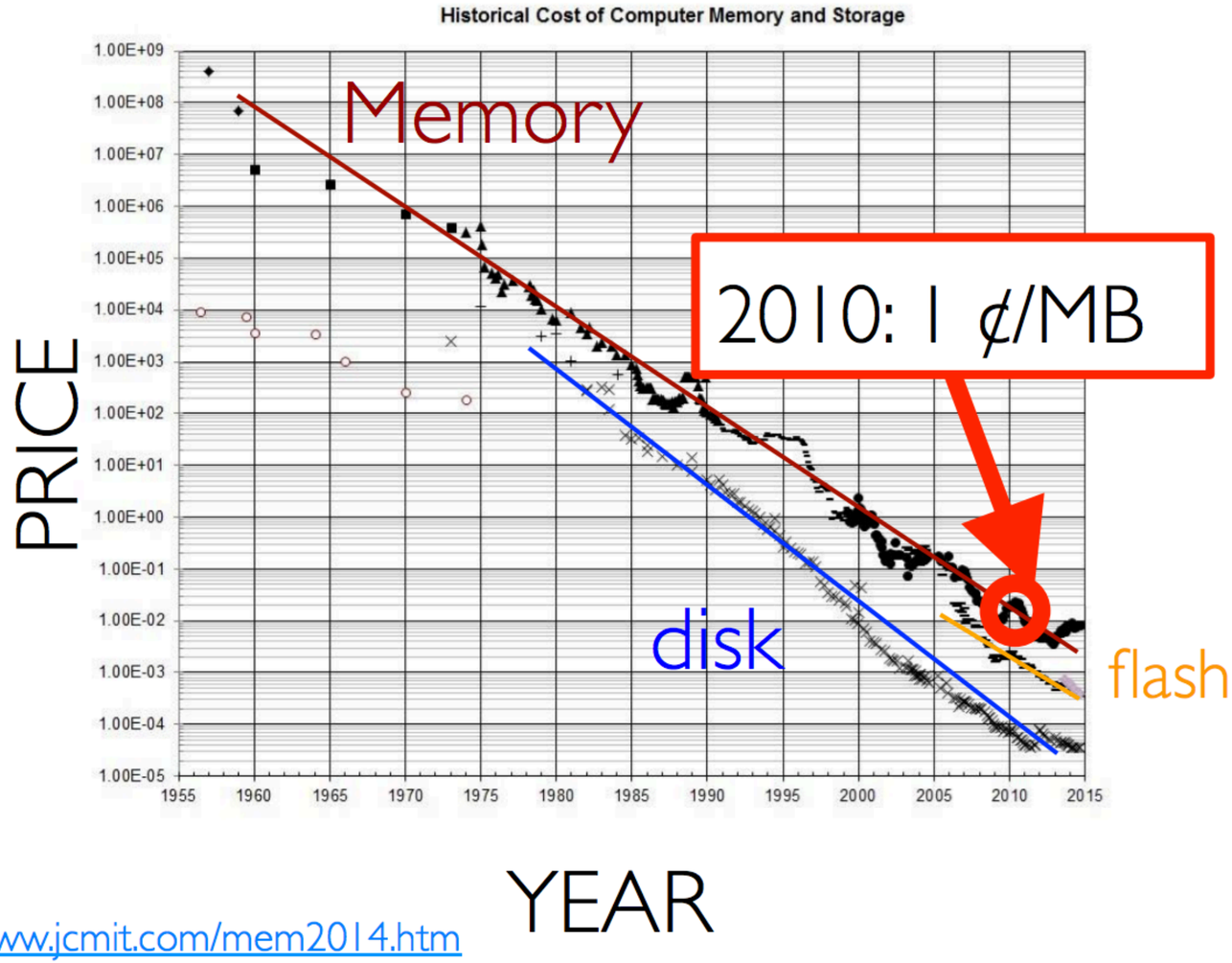


Interactive mining Stream processing

Also, iterative jobs

Disk I/O is very slow

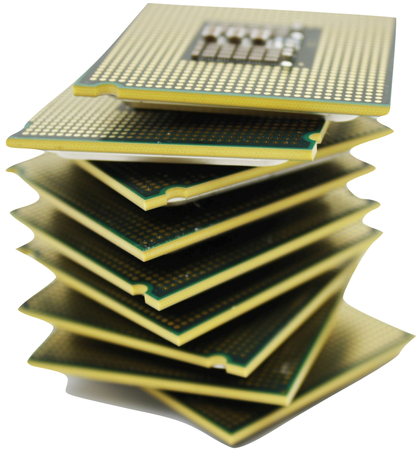
Tech Trend: Cost of Memory



Lower cost means can put more memory in each server

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

Hardware for Big Data

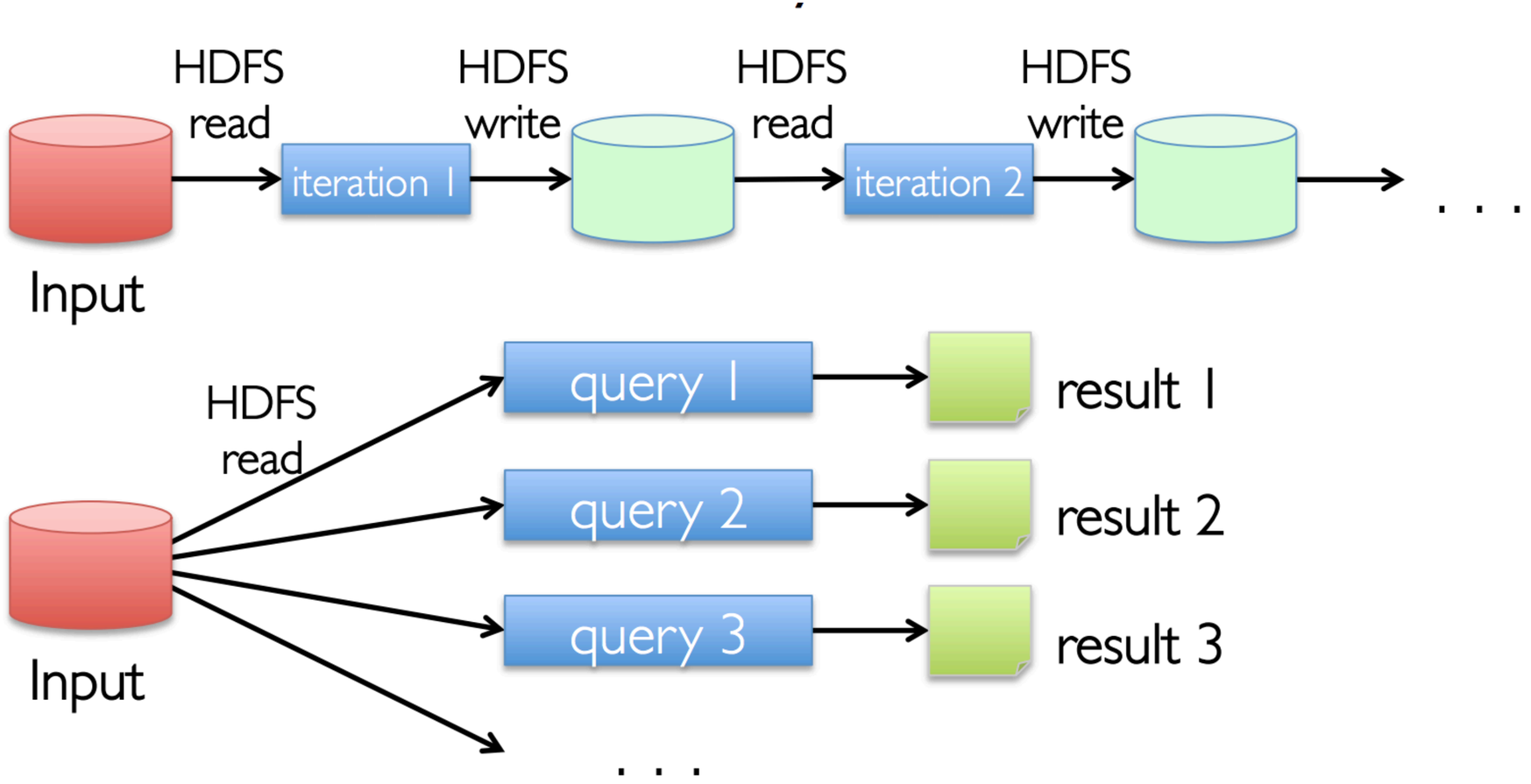


Lot of hard drive

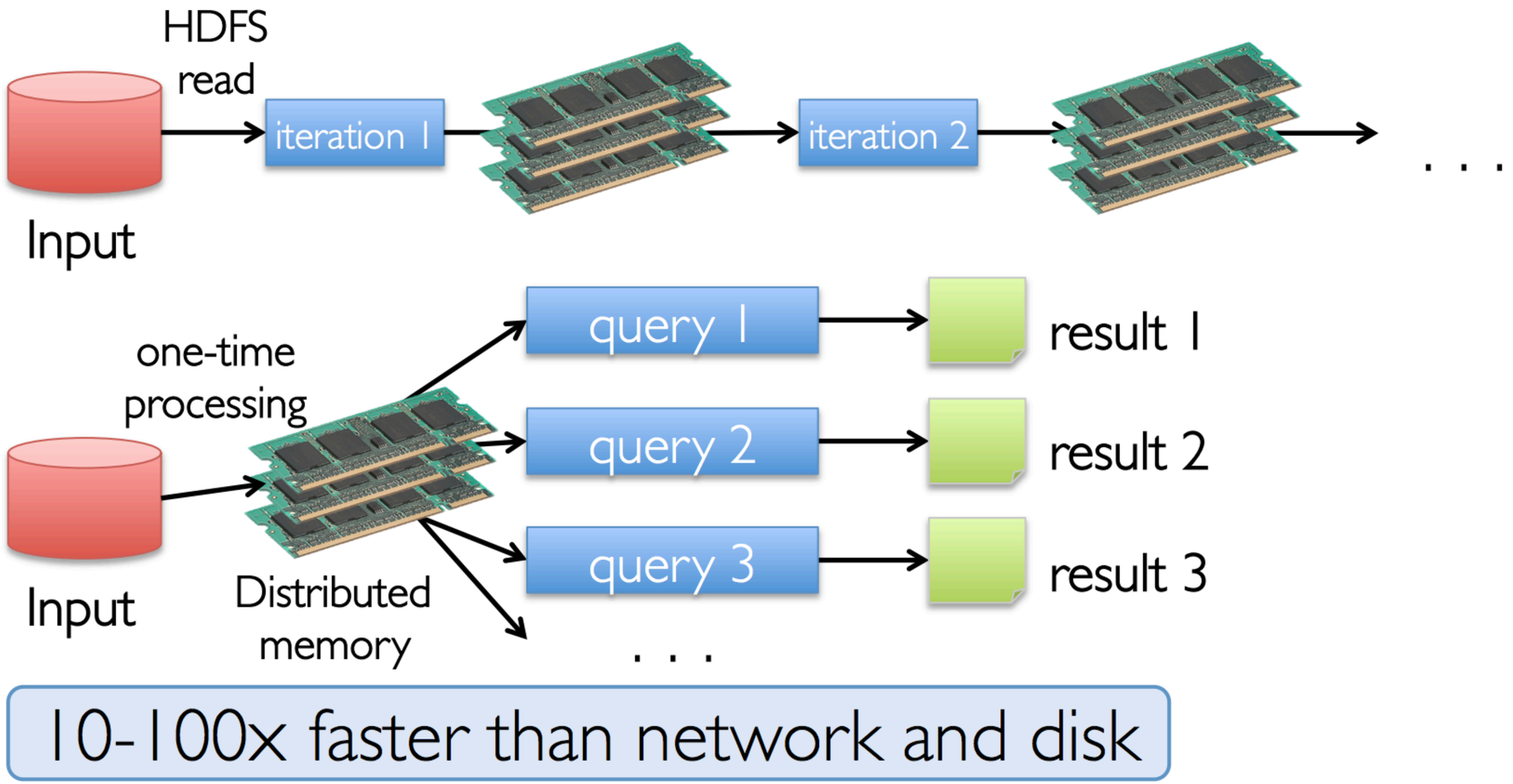
And lot of CPU’s

…and CPU

Opportunity:
Use Memory Instead of Disk

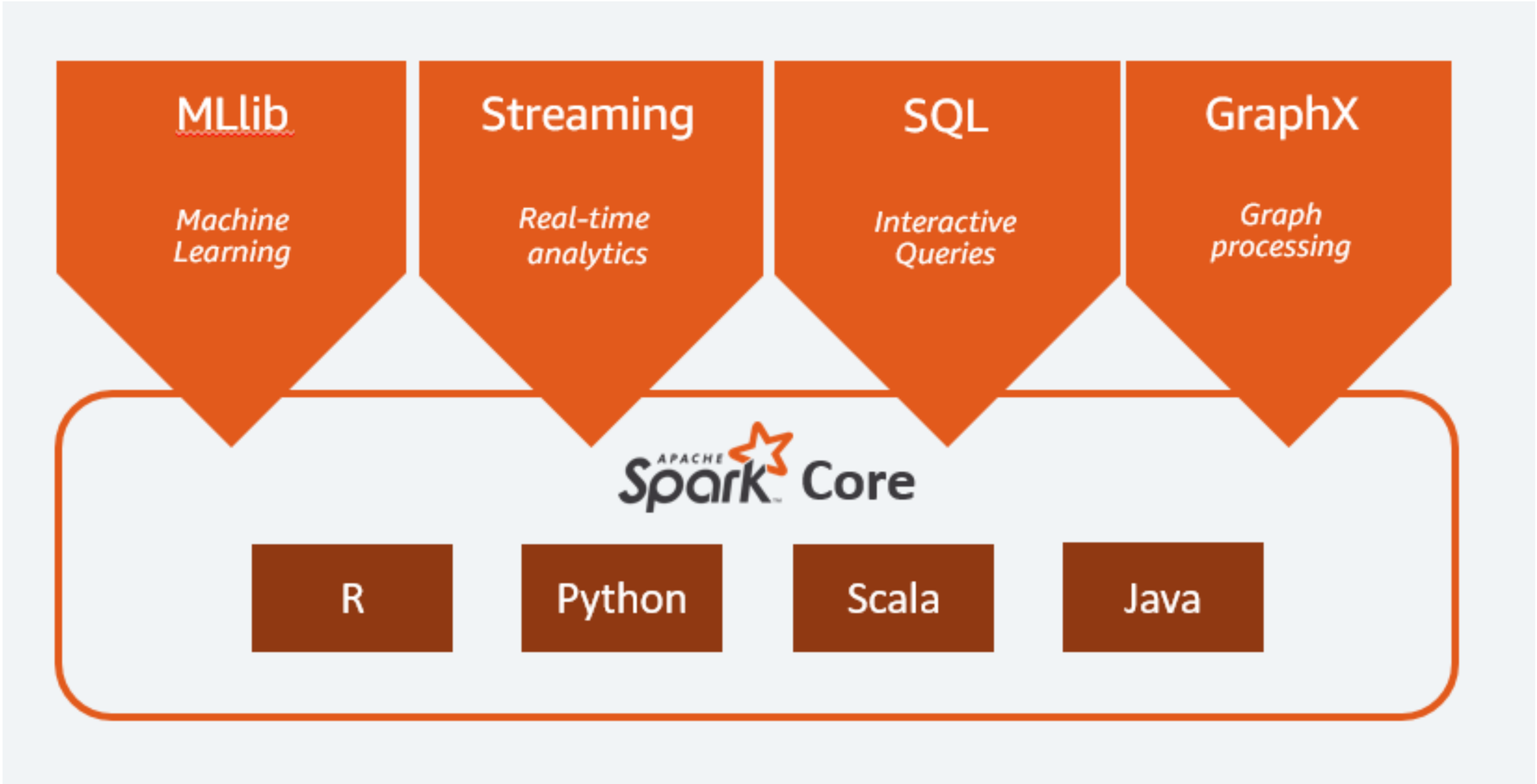


Opportunity:
Use Memory Instead of Disk

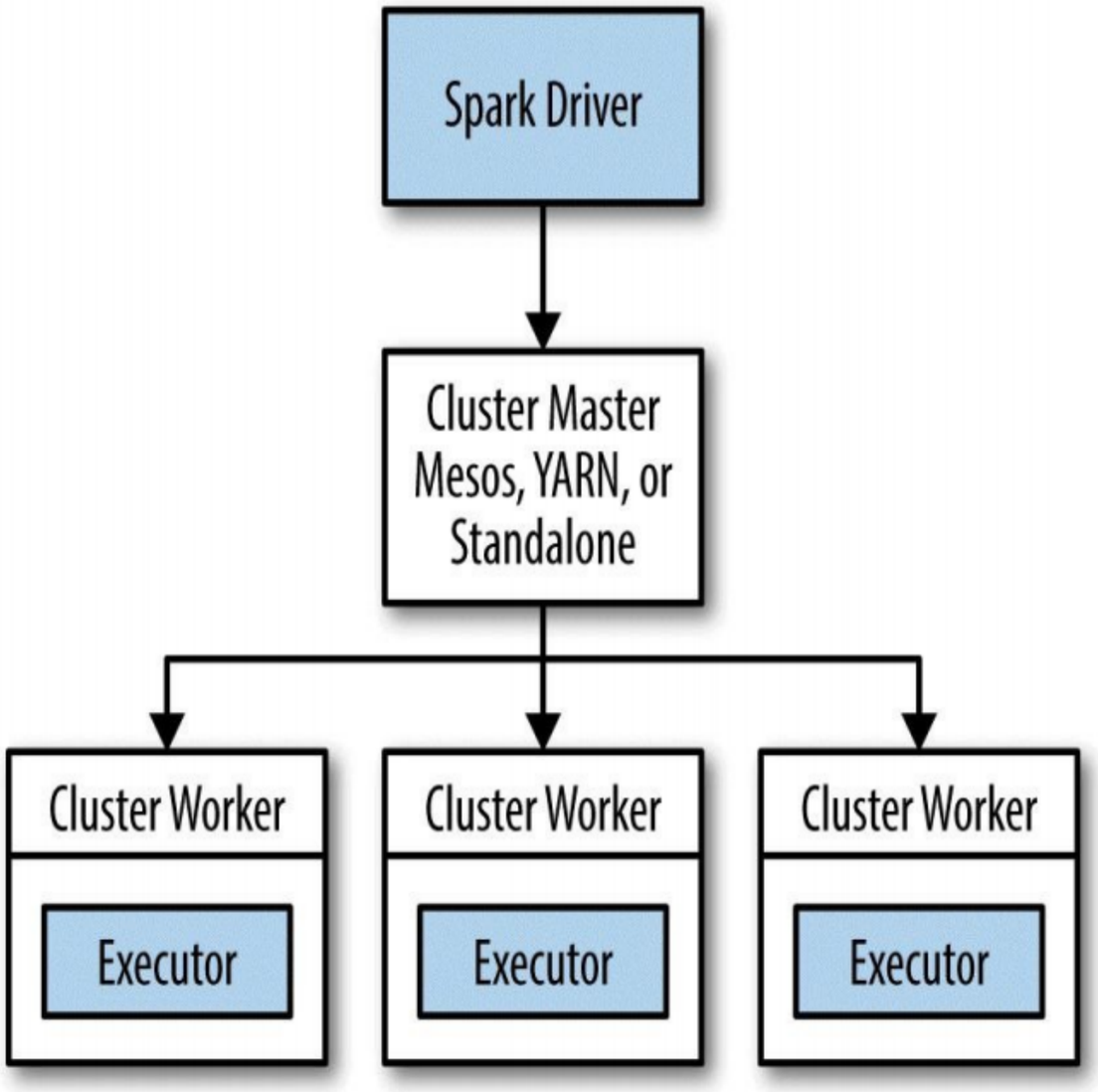


10-100x faster than network and disk

Apache Spark



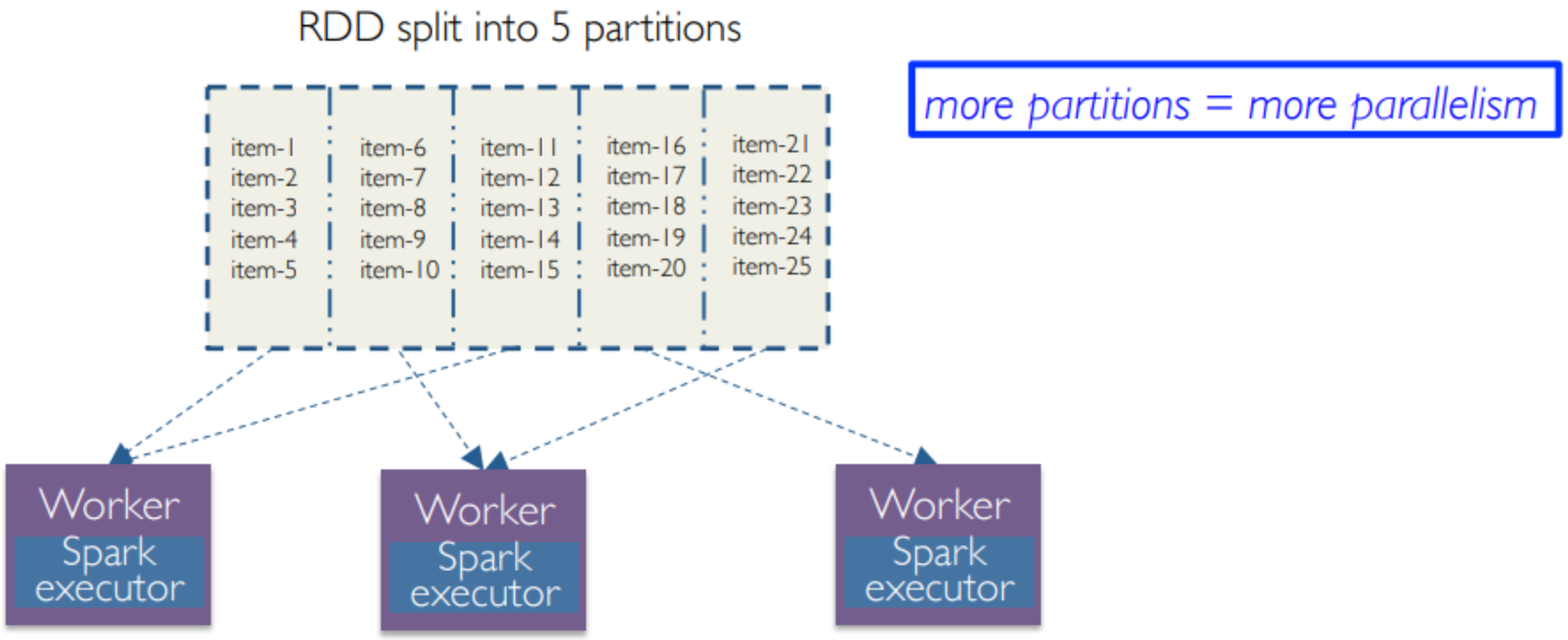
Spark Context



Spark Context

Master Parameter	Description
`local`	run Spark locally with one worker thread (no parallelism)
`local[K]`	run Spark locally with K worker threads (ideally set to number of cores)
`spark://HOST:PORT`	connect to a Spark standalone cluster; PORT depends on config (7077 by default)
`mesos://HOST:PORT`	connect to a Mesos cluster; PORT depends on config (5050 by default)

RDD



more partitions = more parallelism

Operation with RDD

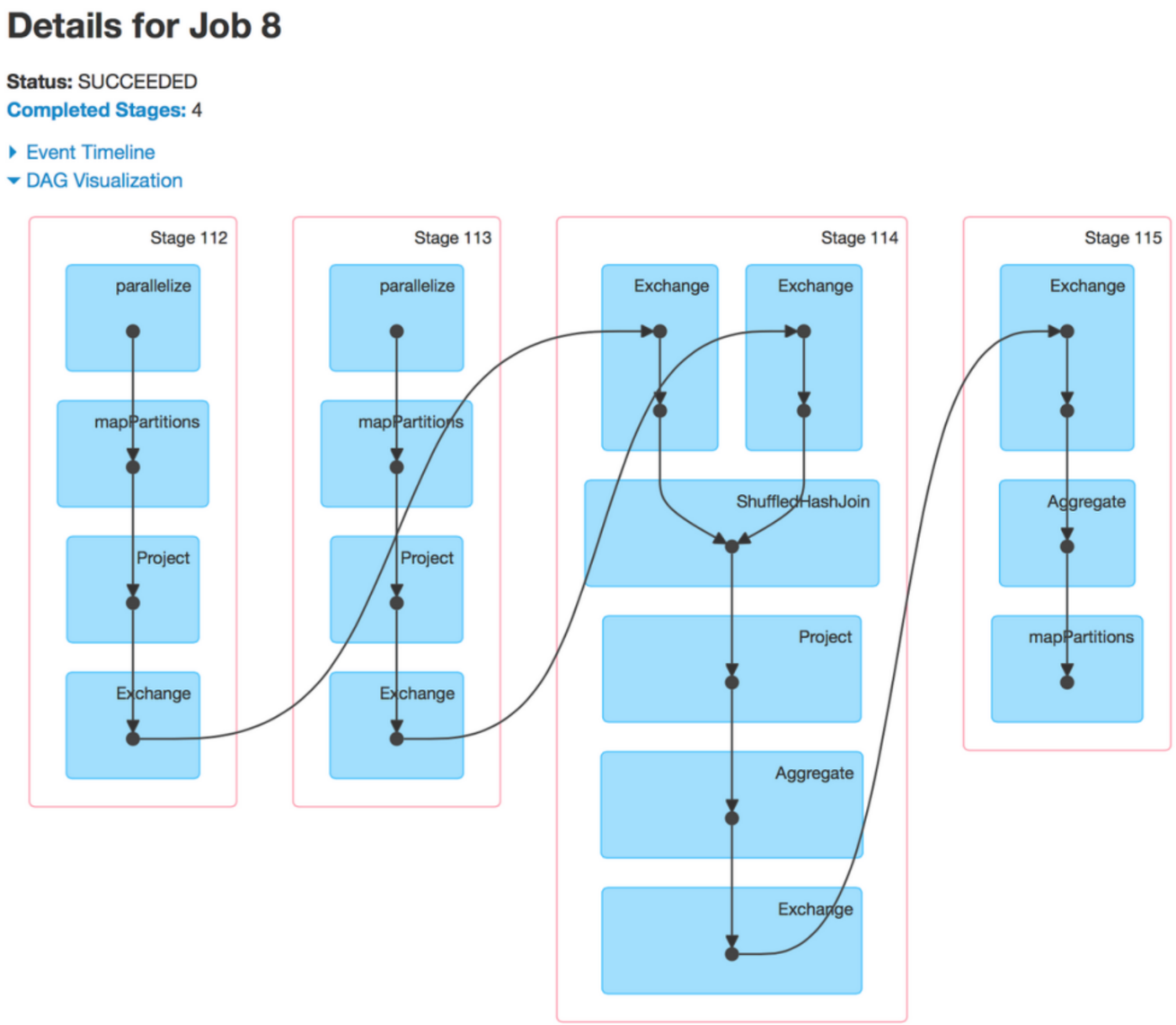
1. Transformation (lazy)

2. Action

```
sc.parallelize(Seq(1, 2, 3, 4, 5, 6, 7))

res2: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[15] at parallelize at <console>:30
```

Transformation



Filter

```
val example = sc.parallelize(Seq(1, 2, 3, 4, 5, 6, 7))
example.filter(_ % 2 == 0)
example.filter(_ % 2 == 0).collect()
```

```
example: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[30] at parallelize at <console>:29
res11: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[31] at filter at <console>:30
res12: Array[Int] = Array(2, 4, 6)
```

Map

```
val example = sc.parallelize(Seq(1, 2, 3, 4, 5, 6, 7))
example.map(x => x + 1).collect()
```

```
example: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[23] at parallelize at <console>:29
res7: Array[Int] = Array(2, 3, 4, 5, 6, 7, 8)
```

flatMap

```
val example = sc.parallelize(Seq(Seq(1, 2), Seq(3, 4), Seq(5, 6), Seq(7)))
example.flatMap(x => x.map(_ + 1)).collect()
```

```
example: org.apache.spark.rdd.RDD[Seq[Int]] = ParallelCollectionRDD[26] at parallelize at <console>:29
res9: Array[Int] = Array(2, 3, 4, 5, 6, 7, 8)
```

Action

```
val example = sc.parallelize(Seq(1, 2, 3, 4, 5, 6, 7))
example.collect()
example.take(2)
example.count()
example.reduce((x, y) => x + y)
```

```
example: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[33] at parallelize at <console>:29
res13: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7)
res14: Array[Int] = Array(1, 2)
res15: Long = 7
res16: Int = 28
```

Prevent double computing

```
val example = sc.parallelize(Seq(1, 2, 3, 4, 5, 6, 7)).map(x => x * x)
example.collect()
example.reduce((x, y) => x + y)
```

```
example: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[35] at map at <console>:29
res17: Array[Int] = Array(1, 4, 9, 16, 25, 36, 49)
res18: Int = 140
```

Cache

```
val example = sc.parallelize(Seq(1, 2, 3, 4, 5, 6, 7)).map(x => x * x).cache()
example.collect()
example.reduce((x, y) => x + y)
```

```
example: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[1] at map at <console>:29
res0: Array[Int] = Array(1, 4, 9, 16, 25, 36, 49)
res1: Int = 140
```

Persist

```
val example = sc.parallelize(Seq(1, 2, 3, 4, 5, 6, 7)).map(x => x * x).persist(StorageLevel.MEMORY_ONLY_SER)
example.collect()
example.reduce((x, y) => x + y)
```

```
example: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[3] at map at <console>:35
res2: Array[Int] = Array(1, 4, 9, 16, 25, 36, 49)
res3: Int = 140
```

Level	Space used	CPU time	In memory	On disk	Comments
MEMORY_ONLY	High	Low	Y	N	
MEMORY_ONLY_SER	Low	High	Y	N	
MEMORY_AND_DISK	High	Medium	Some	Some	Spills to disk if there is too much data to fit in memory.
MEMORY_AND_DISK_SER	Low	High	Some	Some	Spills to disk if there is too much data to fit in memory. Stores serialized representation in memory.
DISK_ONLY	Low	High	N	Y	

Key-value

```
val example = sc.parallelize(Seq(("A", 1), ("A", 1), ("B", 1), ("C", 1)))
example.reduceByKey((l, r) => l + r).collect
example.groupByKey().collect
```

```
example: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[675] at parallelize at <console>:29
res67: Array[(String, Int)] = Array((A,2), (B,1), (C,1))
res68: Array[(String, Iterable[Int])] = Array((A,CompactBuffer(1, 1)), (B,CompactBuffer(1)), (C,CompactBuffer(1)))
```

Shared variables

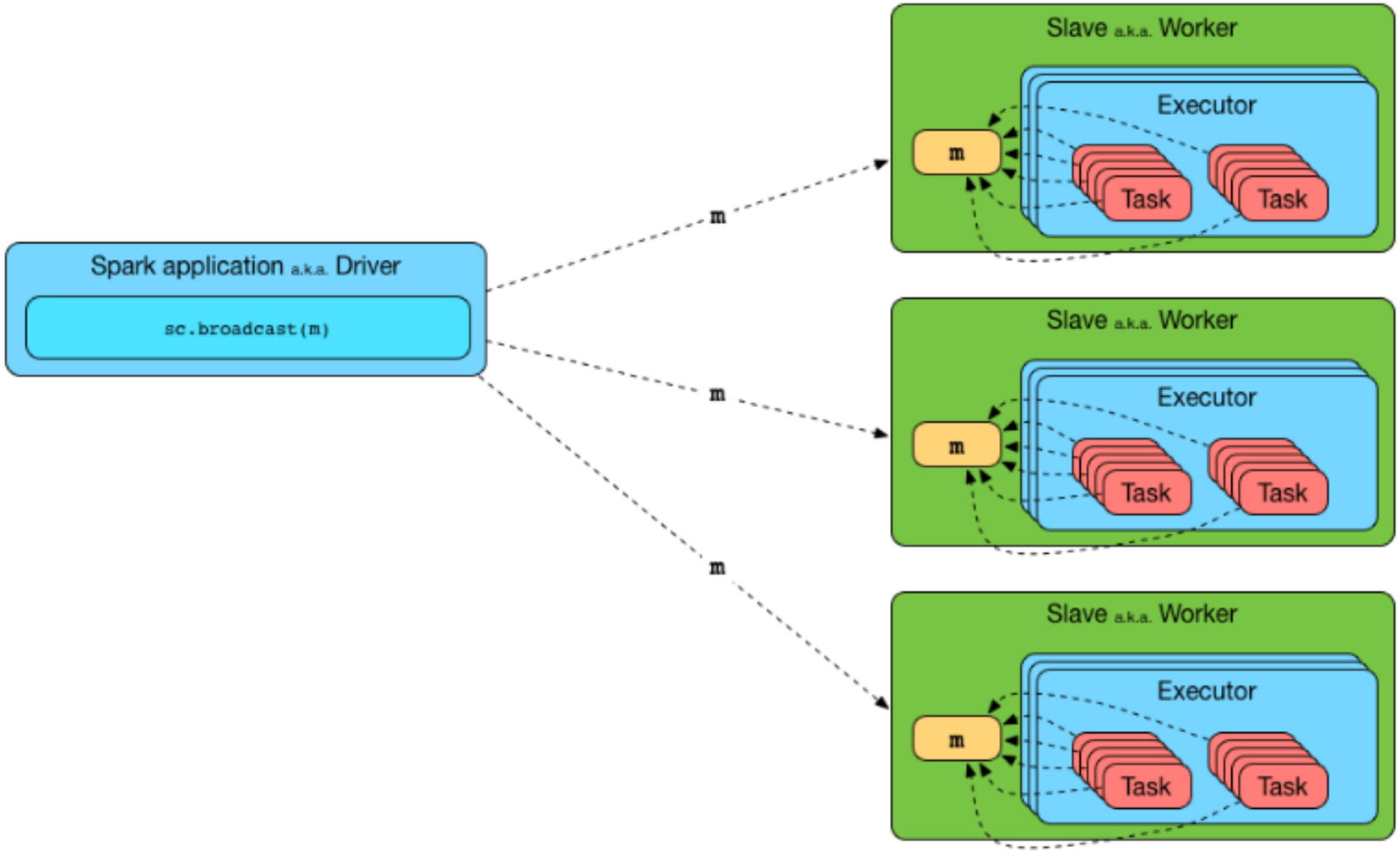


Figure 1. Broadcasting a value to executors

Accumulators

```
val counter = sc.longAccumulator("counter")
sc.parallelize(1 to 9).foreach(x => counter.add(x))
counter.value
```

```
counter: org.apache.spark.util.LongAccumulator = LongAccumulator(id: 126, name: Some(counter), value: 0)
res2: Long = 45
```

Guarantiees:

- Tasks at workers cannot access accumulator's values
- Tasks see accumulators as write-only variables
- Actions: each task's update to accumulator is applied only once
- Transformations: no guarantees (use only for debugging)
- Types: integers, double, long, float

Homework:

- Set-up Java/IDEA/Scala…
- Set-up Apache Spark locally
- Implement traditional WordCount
- Implement wordCount returning only words from specific (a-k-a query) text file;