Introduction to Scalable Data Analytics using Apache Spark

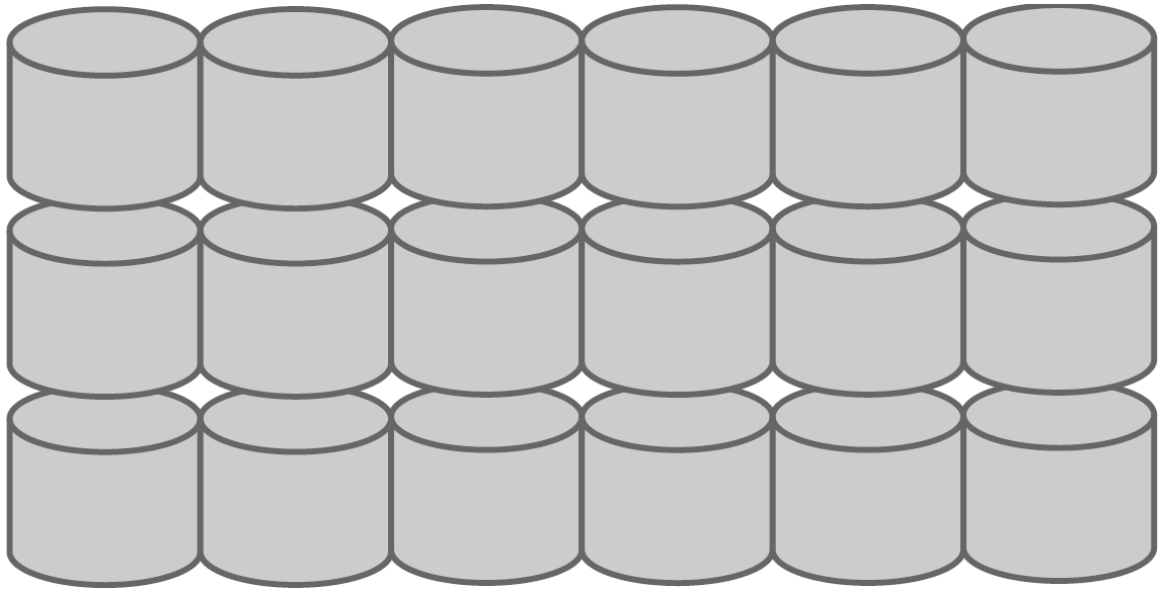
http://www.csd.uoc.gr/~hy562
University of Crete, Fall 2024

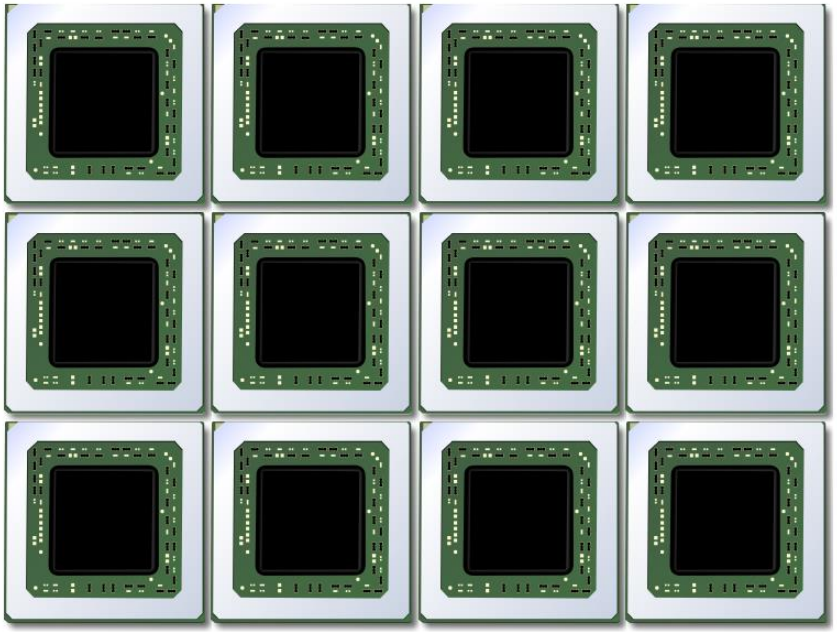
Outline

- Big Data Problems: Distributing Work, Failures, Slow Machines
- What is Apache Spark?
- Core things of Apache Spark
  - RDD
- Core Functionality of Apache Spark
- Simple tutorial

Big Data Problems: Distributing Work, Failures, Slow Machines

Hardware for Big Data





Bunch of Hard Drives

…. and CPUs

- The Big Data Problem
  - Data growing faster than CPU speeds
  - Data growing faster than per-machine storage
- Can’t process or store all data on one machine

Hardware for Big Data

- One big box ! (1990s solution)
  - All processors share memory
- Very expensive
  - Low volume
  - All “premium” HW
- Still not big enough!

Image: Wikimedia Commons / User:Tonusamuel

Hardware for Big Data

- Consumer-grade hardware
  - Not "gold plated"
- Many desktop-like servers
  - Easy to add capacity
  - Cheaper per CPU/disk
- But, implies complexity in software

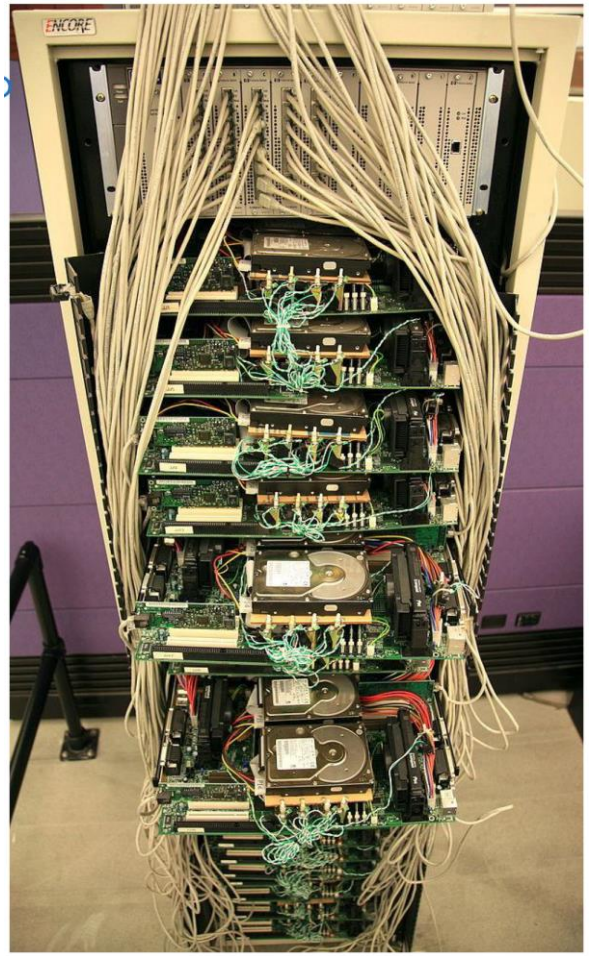
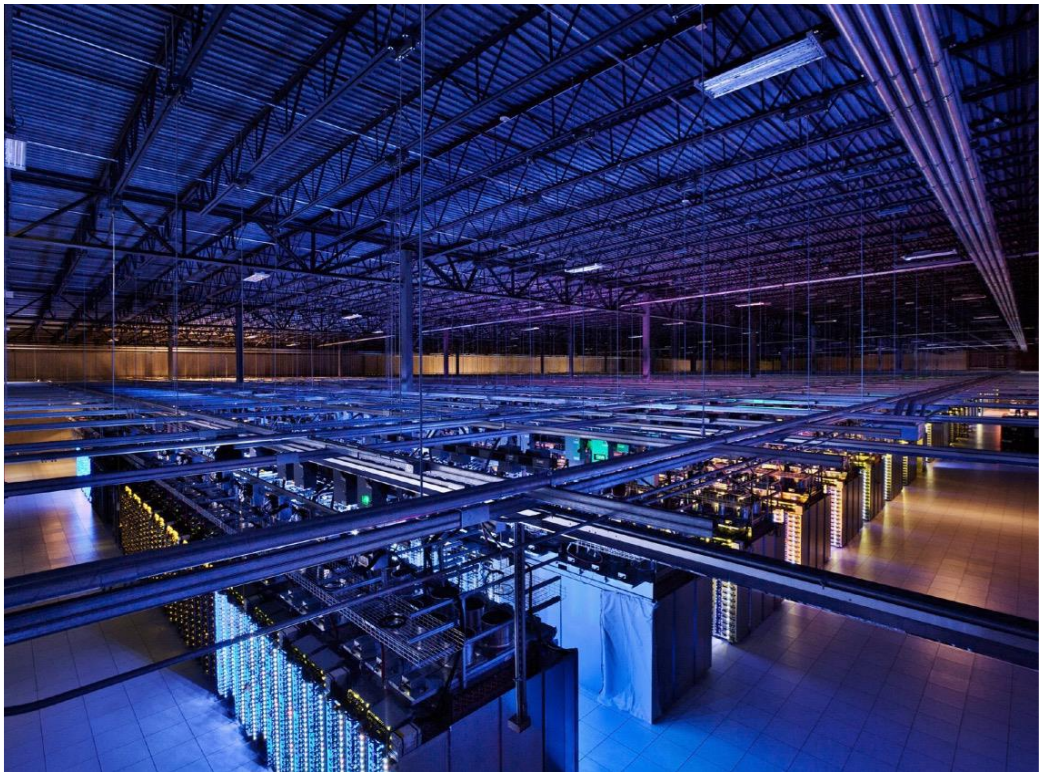


Image: Steve Jurvetson/Flickr

Problems with Cheap HW

- Failures, e.g. (Google numbers)
  - 1-5% hard drives/year
  - 0.2% DIMMs/year
- Network speeds vs. shared memory
  - Much more latency
  - Network slower than storage
- Uneven performance



Google Datacenter

The Opportunity

- Cluster computing is a game-changer!
- Provides access to low-cost computing and storage
- Costs decreasing every year
- The challenge is programming the resources
- What's hard about Cluster computing?
  - How do we split work across machines?

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
…

Centralized Approach: Use a Hash Table

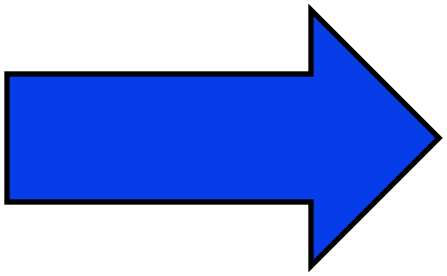
"I am Sam

I am Sam

Sam I am

Do you like

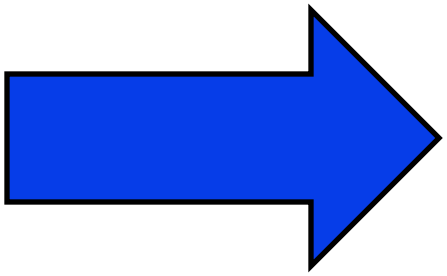
Green eggs and ham?"



{ }

Centralized Approach: Use a Hash Table

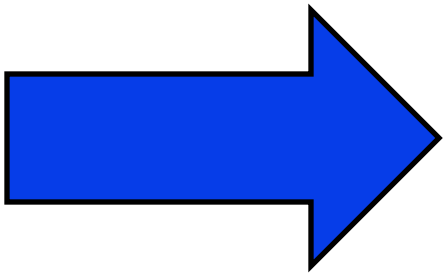
"I am Sam
I am Sam
Sam I am
Do you like
Green eggs and ham?"



{ I: **1**, }

Centralized Approach: Use a Hash Table

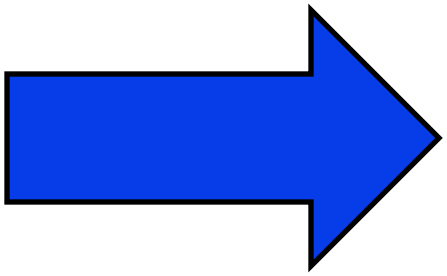
"I am Sam
I am Sam
Sam I am
Do you like
Green eggs and ham?"



{ I: 1,
  am: 1,
}

Centralized Approach: Use a Hash Table

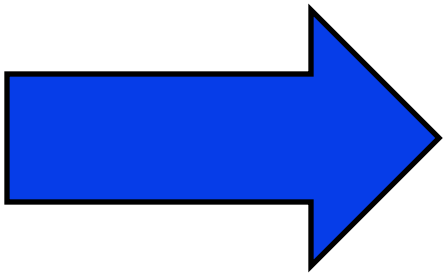
"I am Sam
I am Sam
Sam I am
Do you like
Green eggs and ham?"



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

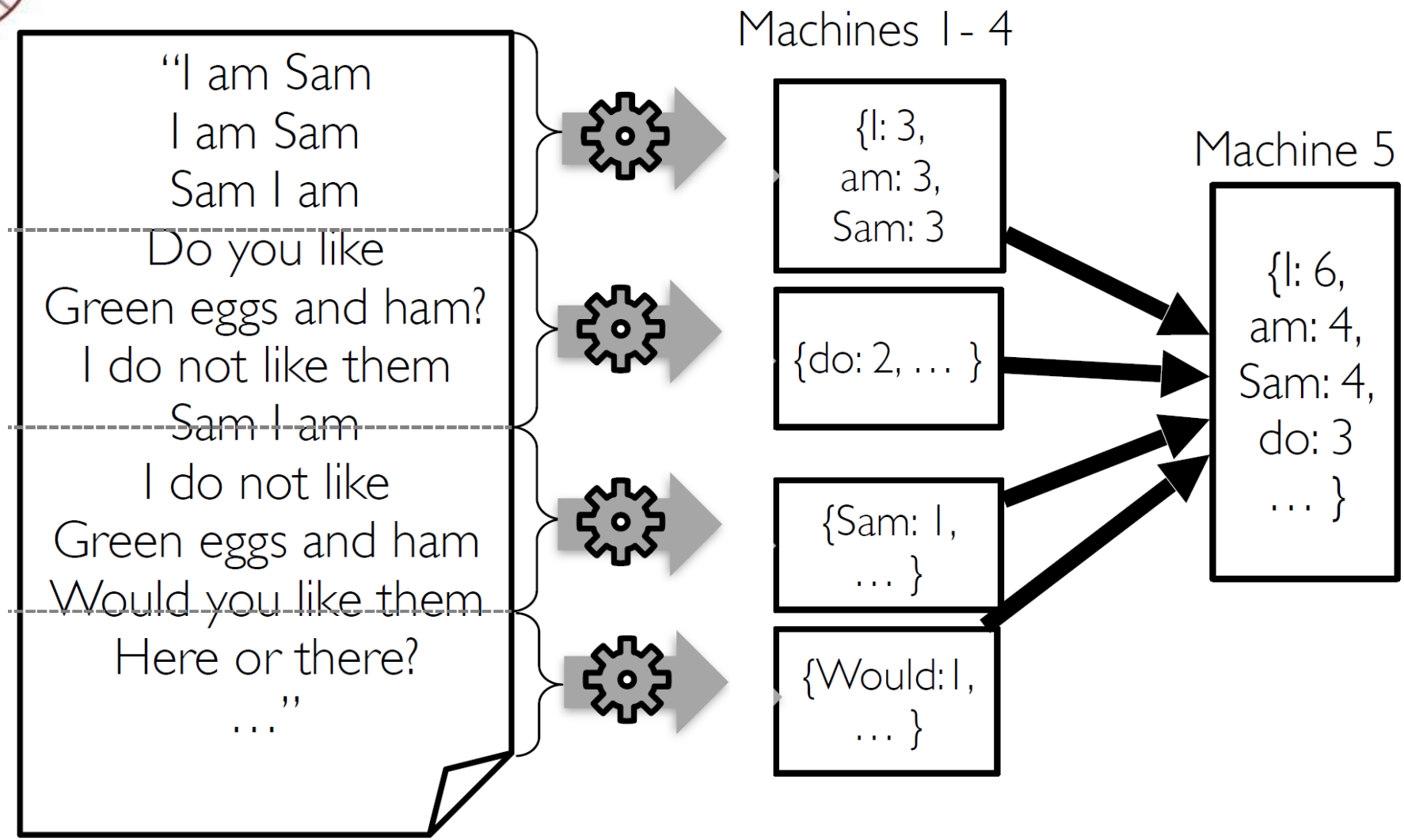
Centralized Approach: Use a Hash Table

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



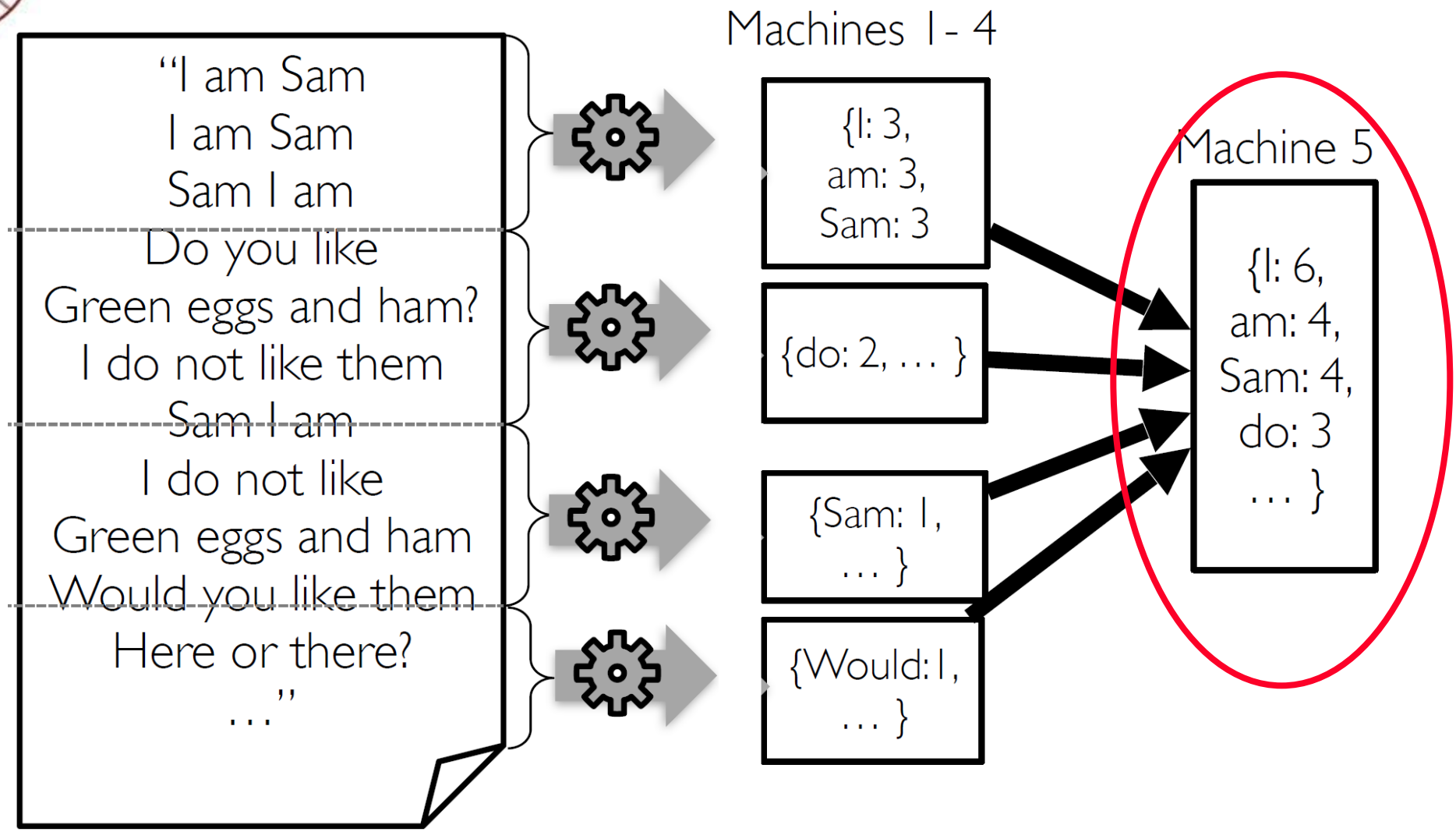
{ I: **2**,
am: **1**,
Sam: 1,
}

A Simple Parallel Approach



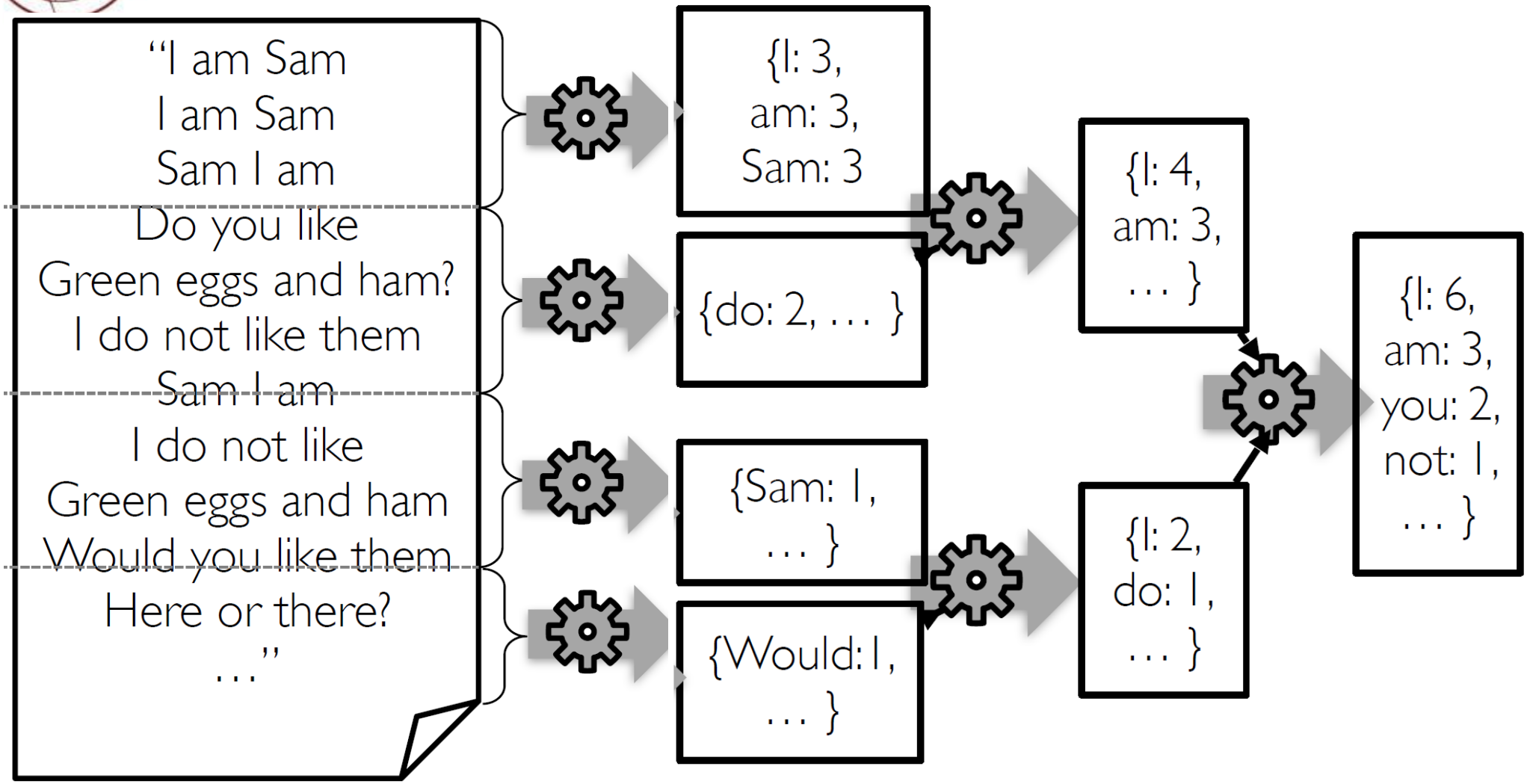
What's the problem with this approach?

What if the Document is Really Big?



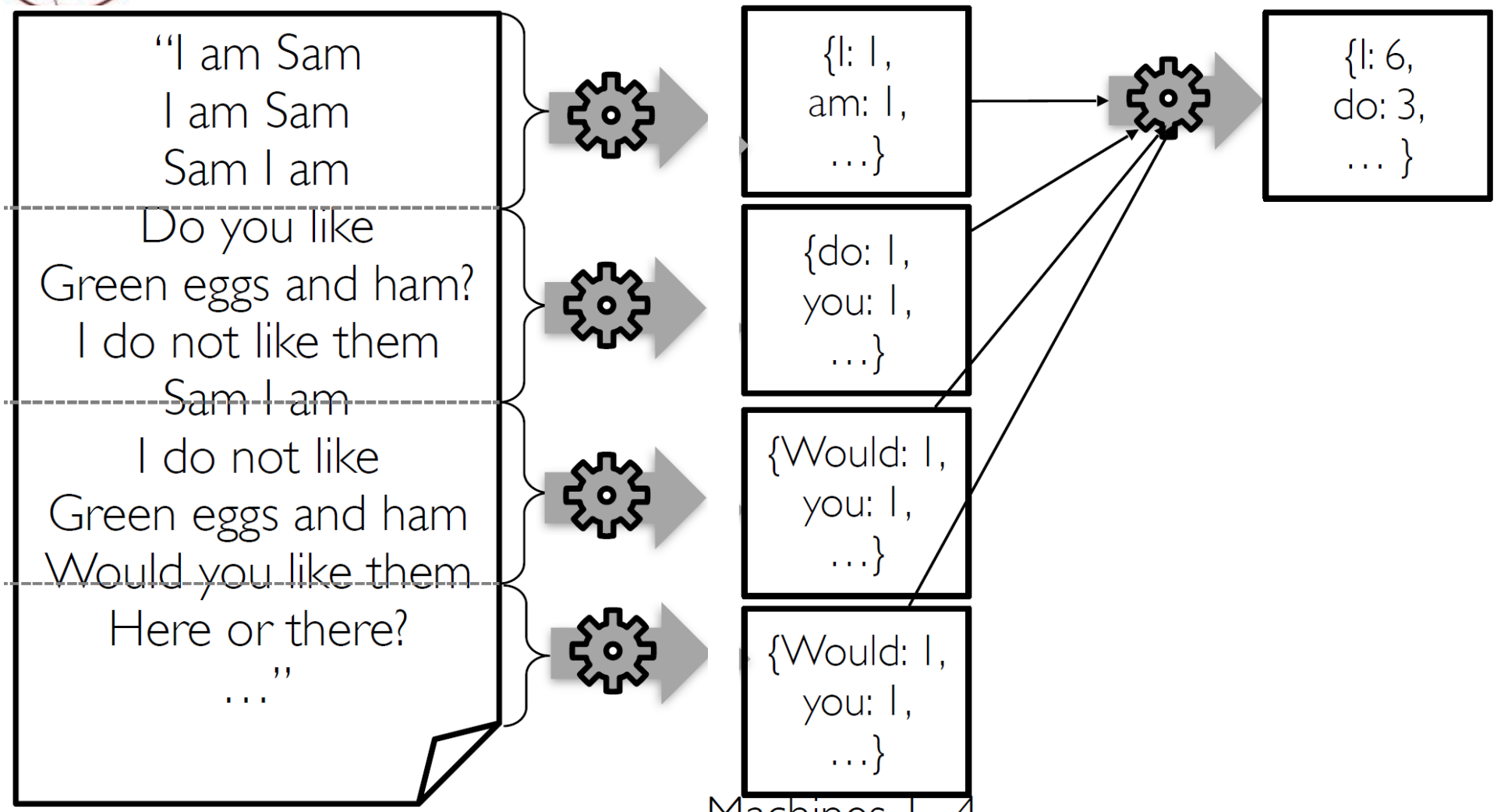
Results have to fit on one machine !

What if the Document is Really Big?



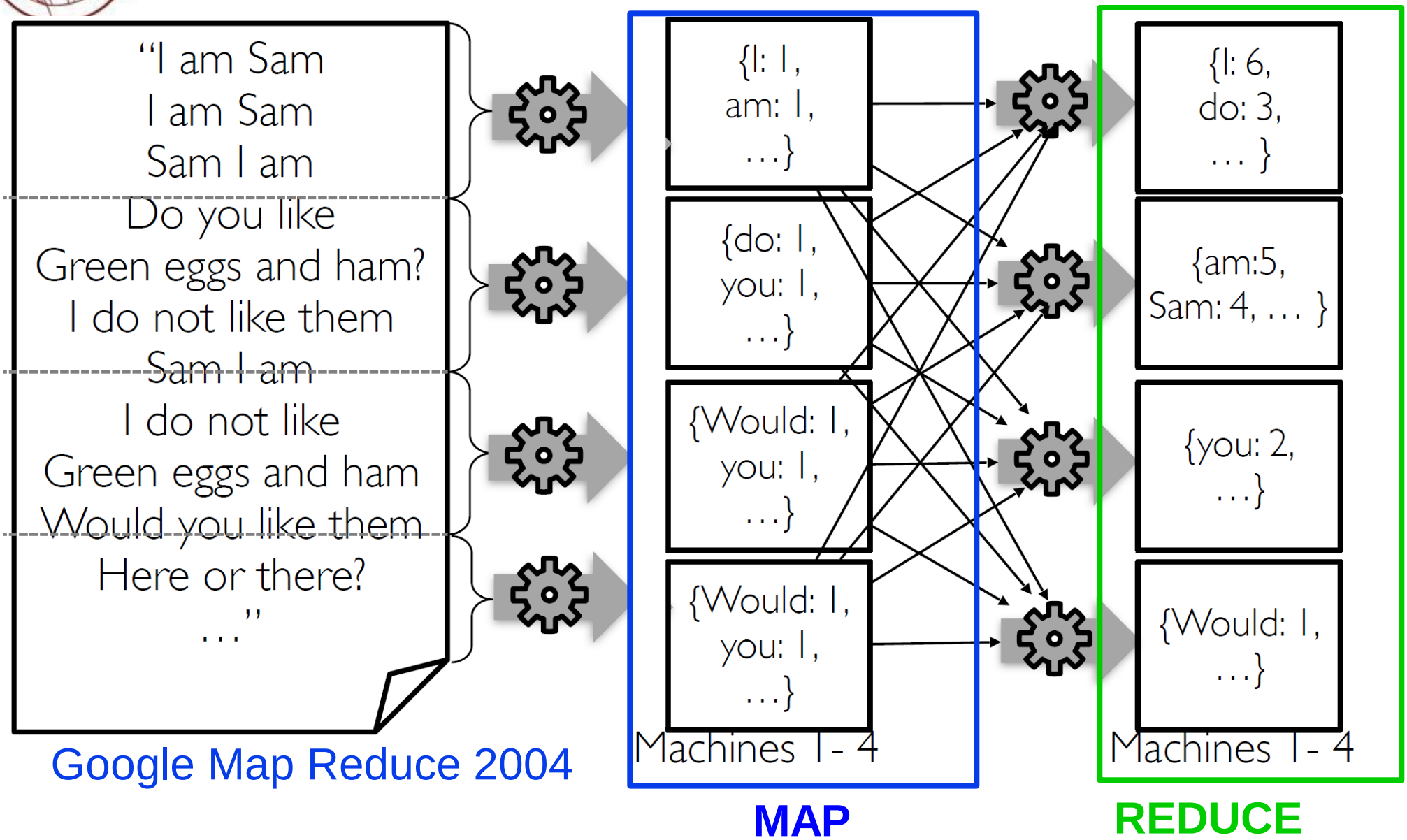
Can add aggregation layers but results must still fit on one machine

What if the Document is Really Big?



Use Divide and Conquer!!

What if the Document is Really Big?



Google Map Reduce 2004

What About the Data? HDFS!

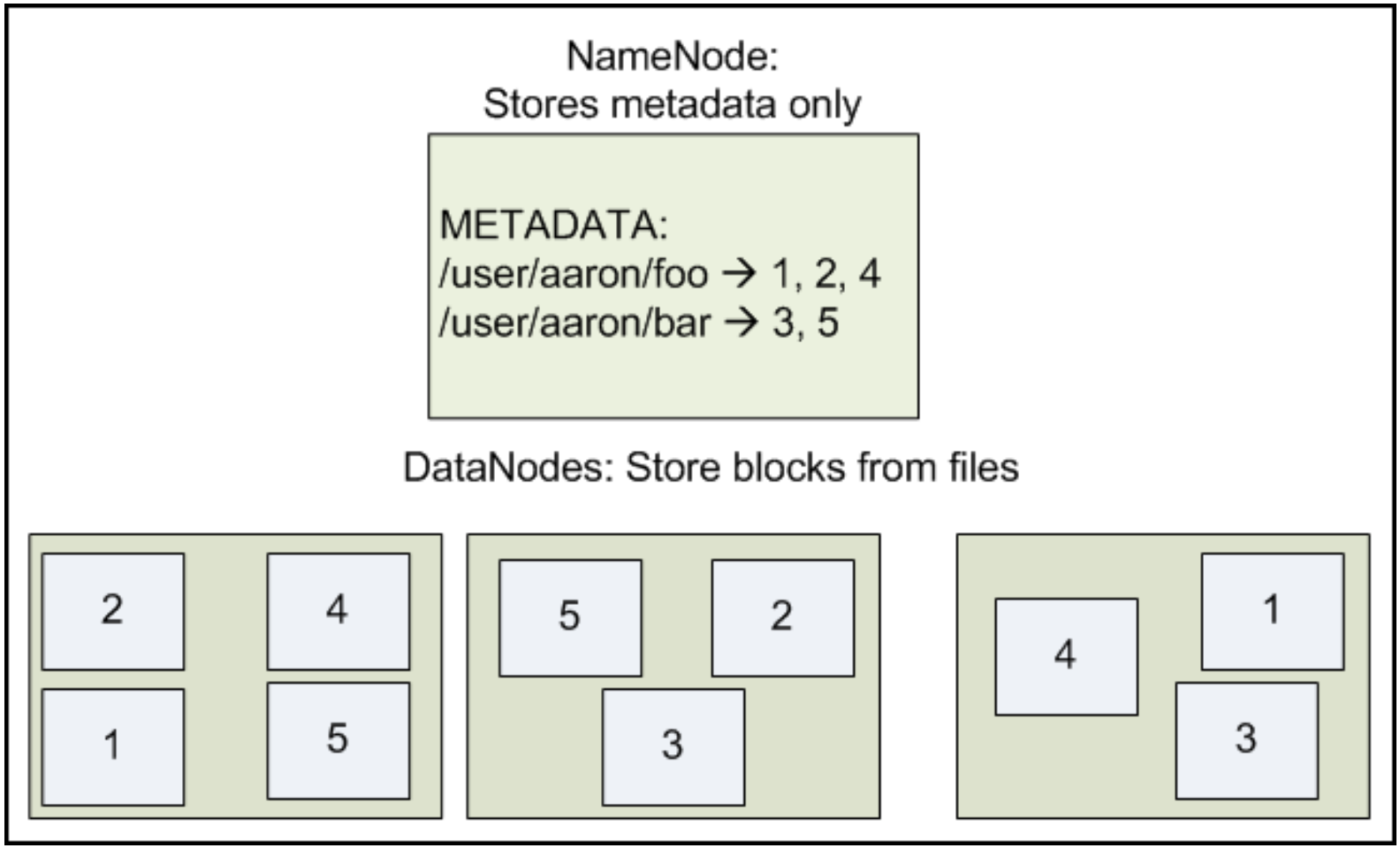
HDFS is a distributed file system designed to hold very large amounts of data (terabytes or even petabytes), and provide high-throughput access to this information

Files are stored in a redundant fashion across multiple machines to ensure their durability to failure and high availability to very parallel applications

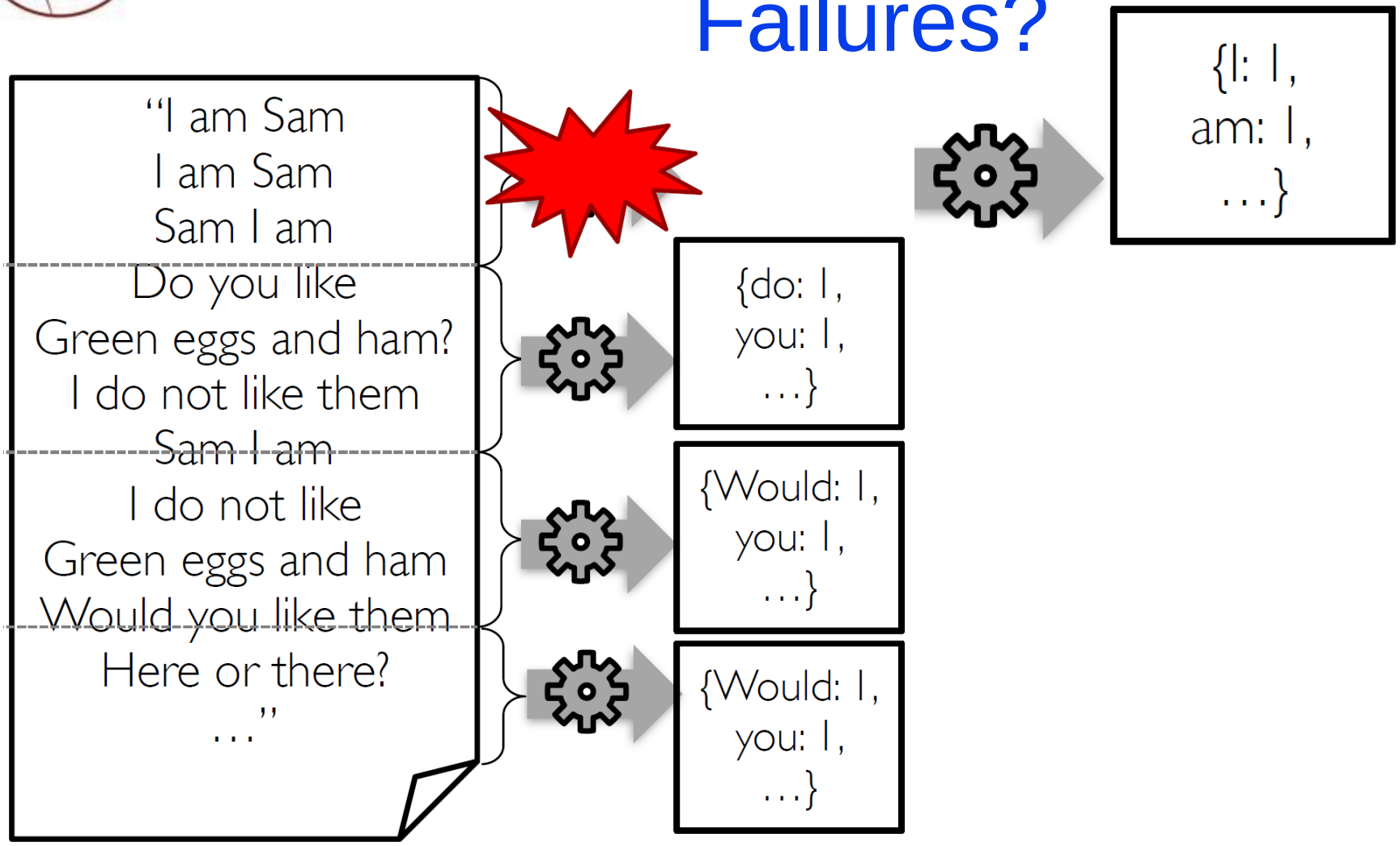
HDFS is a block-structured file system:

- individual files are broken into blocks of a fixed size (default 128MB)
- These blocks are stored across a cluster of one or more machines (DataNodes)
- The NameNode stores all the metadata for the file system

HDFS nodes

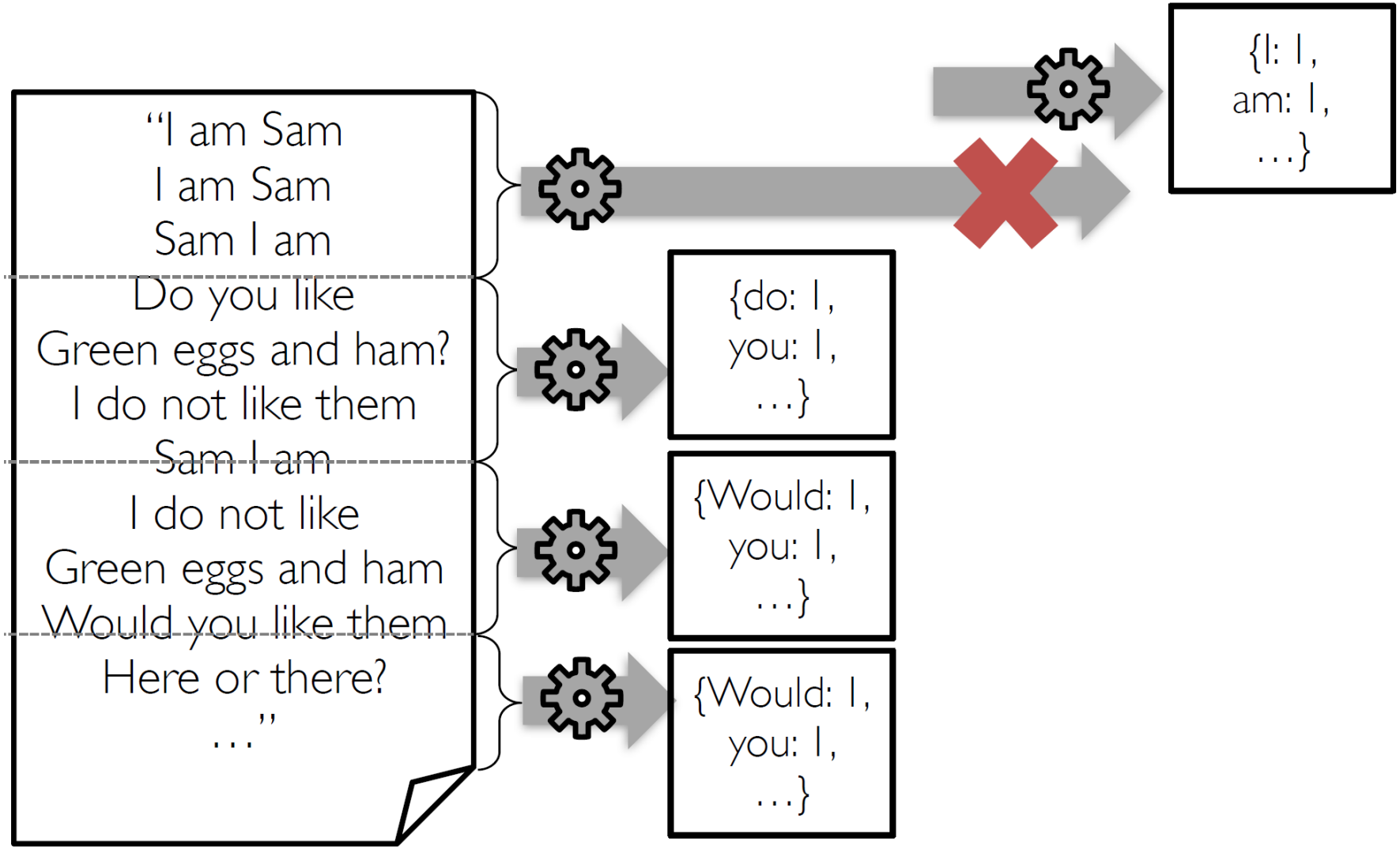

NameNode:
Stores metadata only
METADATA:
/user/aaron/foo → 1, 2, 4
/user/aaron/bar → 3, 5
DataNodes: Store blocks from files
2
4
1
5
5
2
3
4
1
3

How Do We Deal with Machine Failures?



- Launch another task!

How Do We Deal with Slow Tasks?



- Launch another task!

MapReduce: Distributed Execution

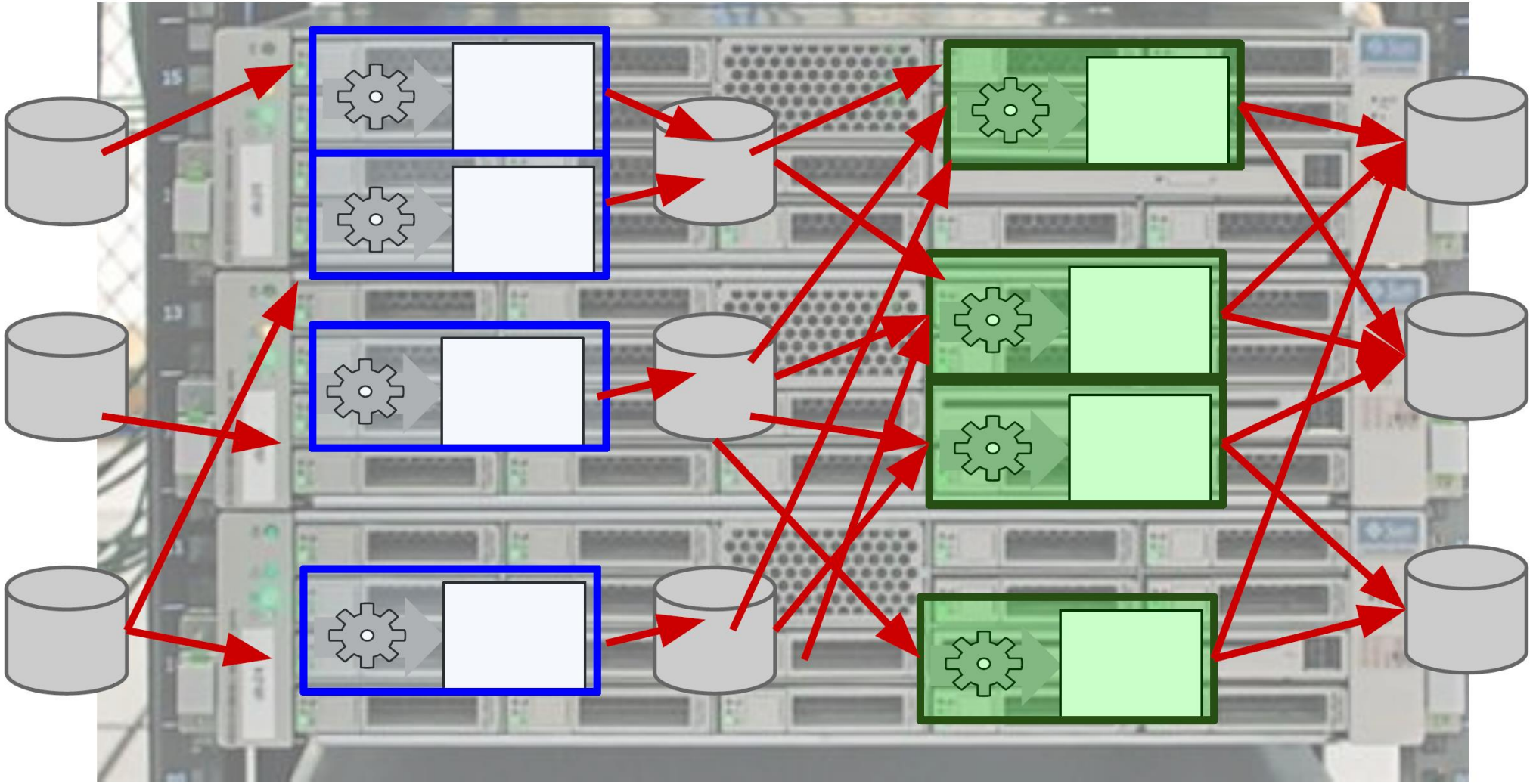
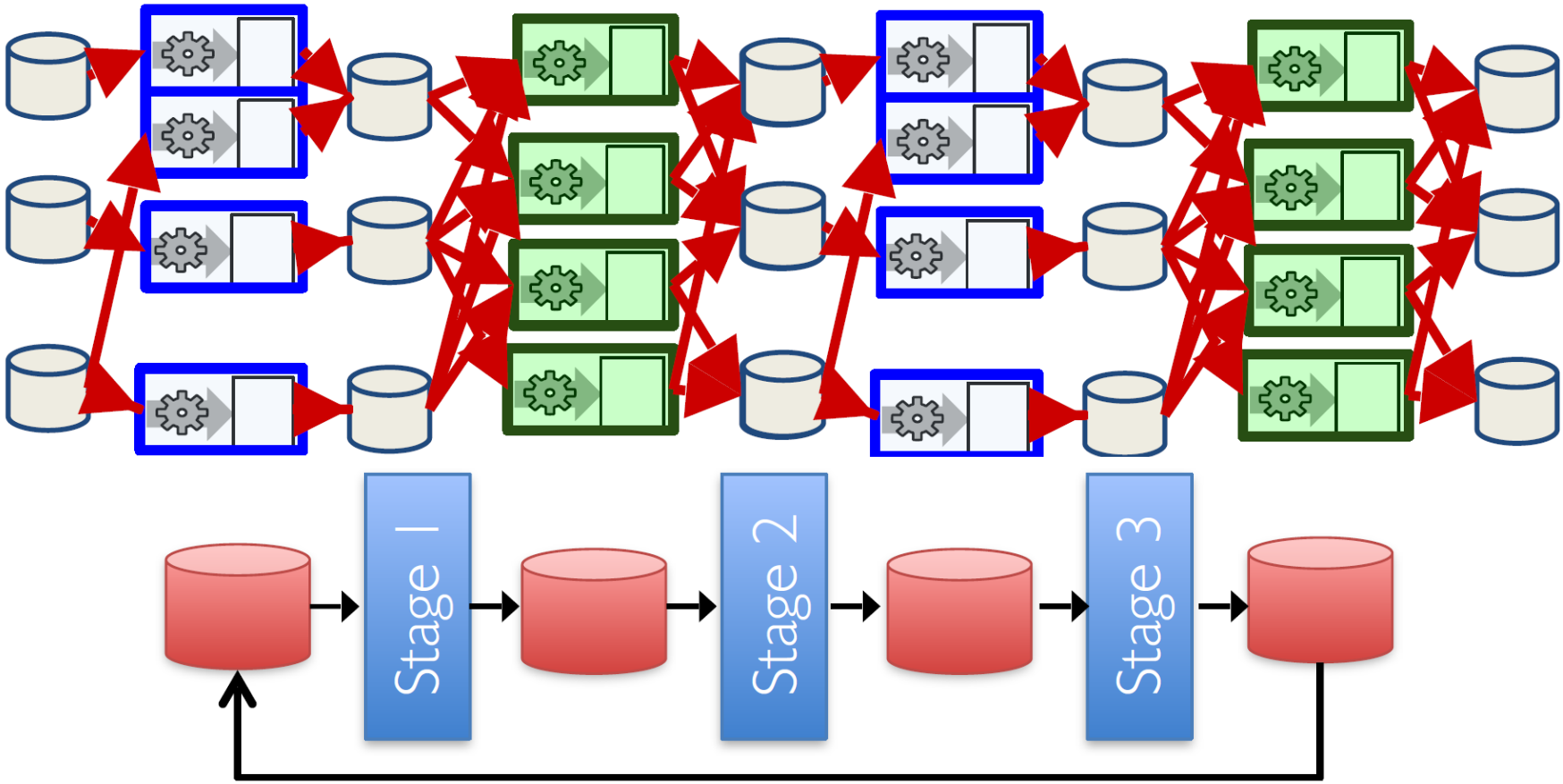


Image: Wikimedia commons (RobH/Tbayer (WMF))

- Each stage passes through the hard drives

Map Reduce: Iterative Jobs

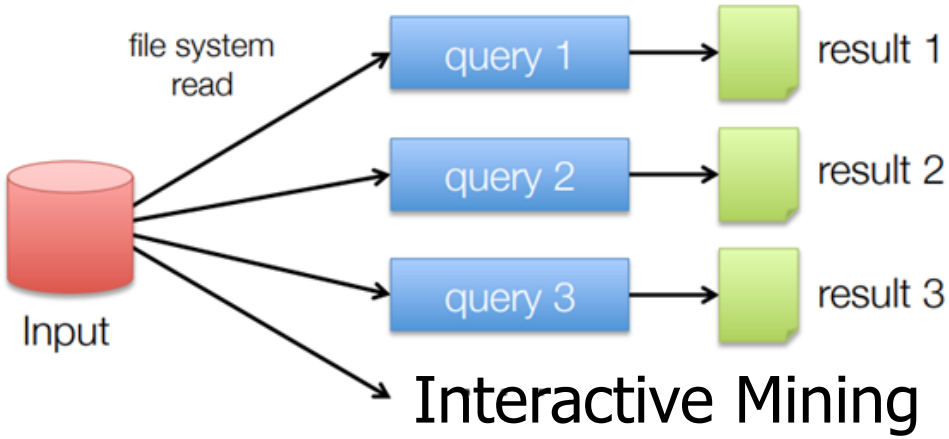
- Iterative jobs involve a lot of disk I/O for each repetition
  - Disk I/O is verv slow!

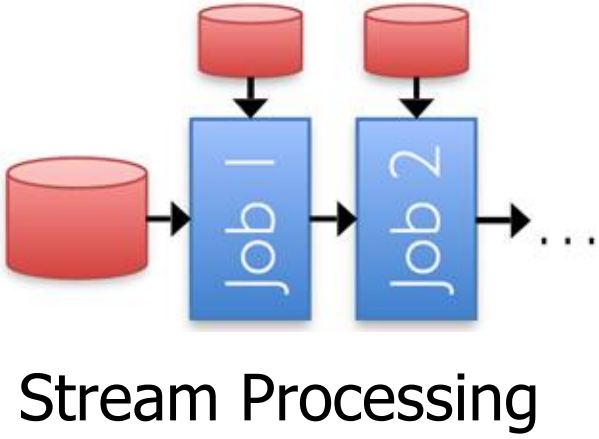


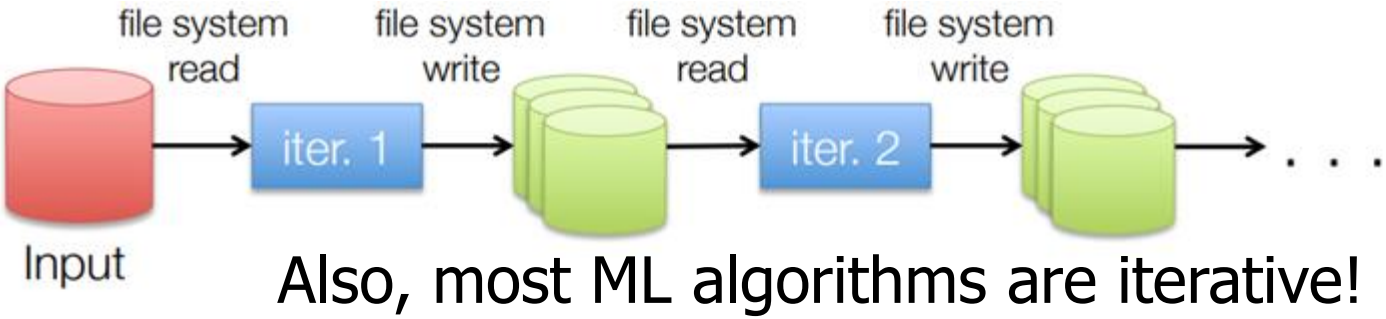
- MapReduce is great at one-pass computation, but inefficient for multi-pass algorithms

The Weakness of MapReduce

- While MapReduce is simple, it can require asymptotically lots of disk I/O for complex jobs, interactive queries and online processing

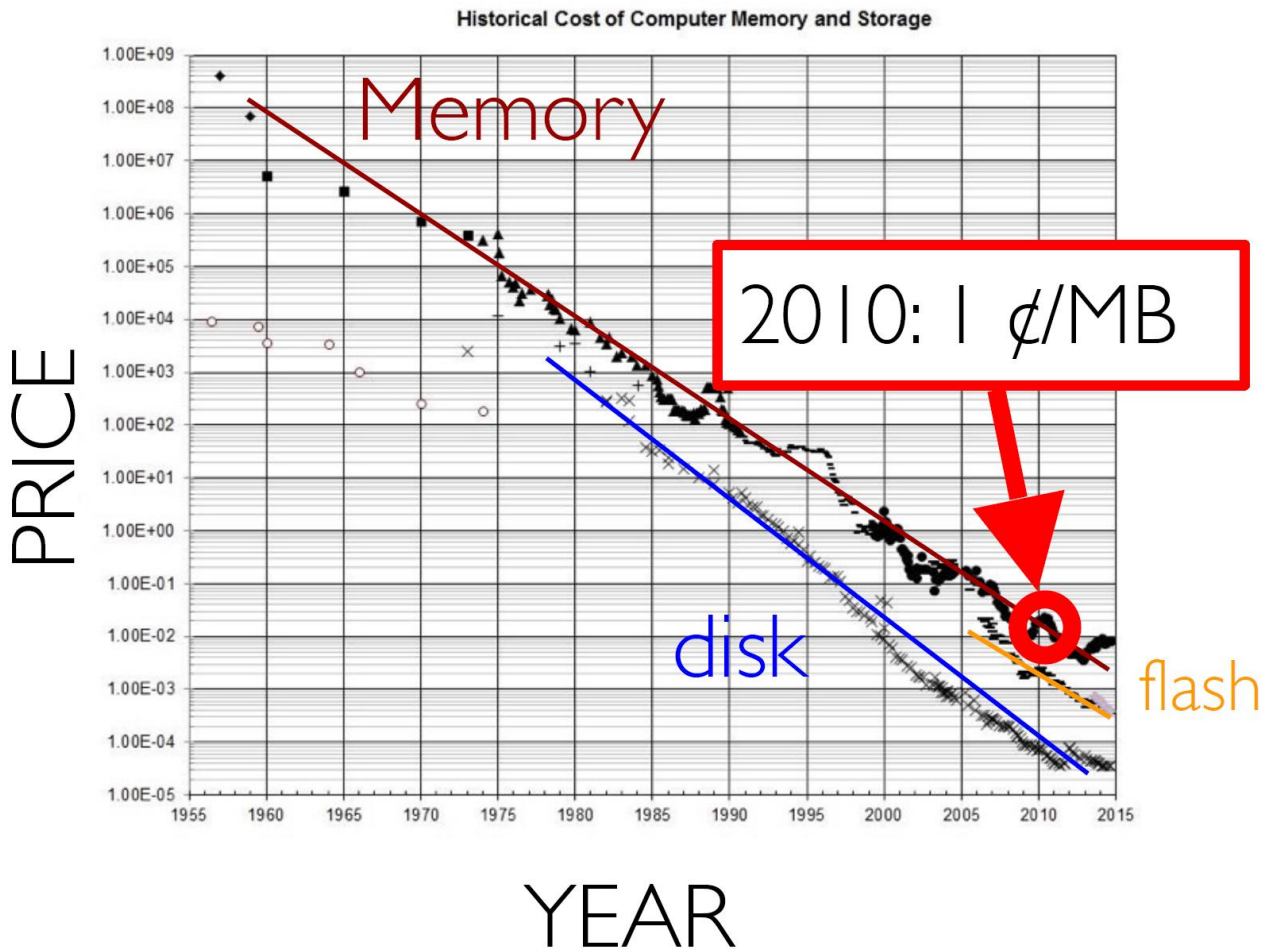






- Commonly spend 90% of time doing I/O!

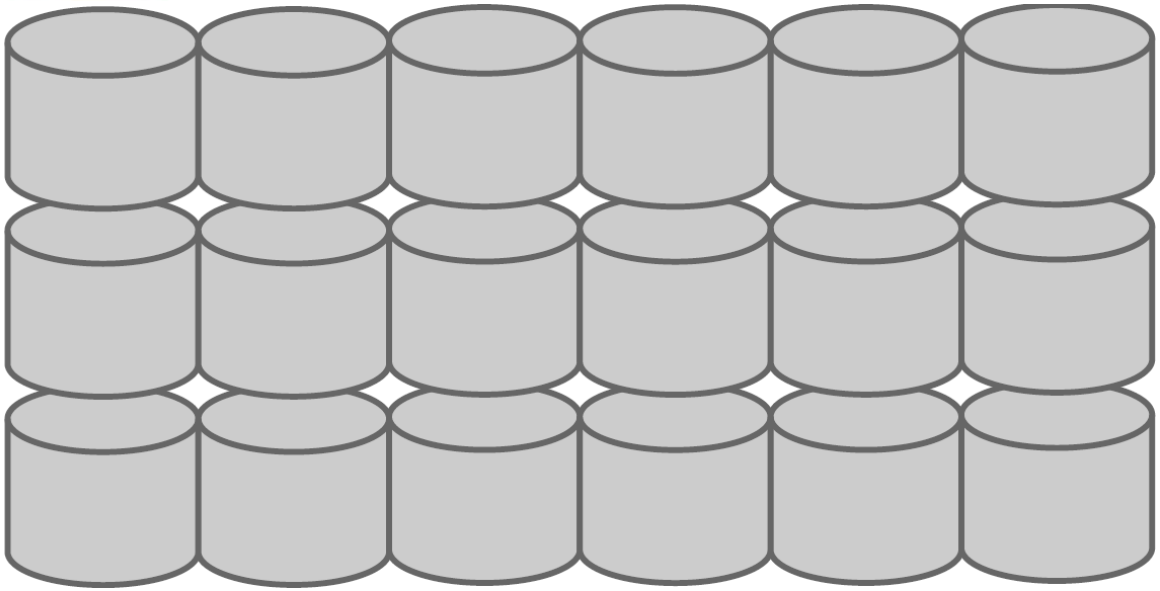
Tech Trend: Cost of Memory

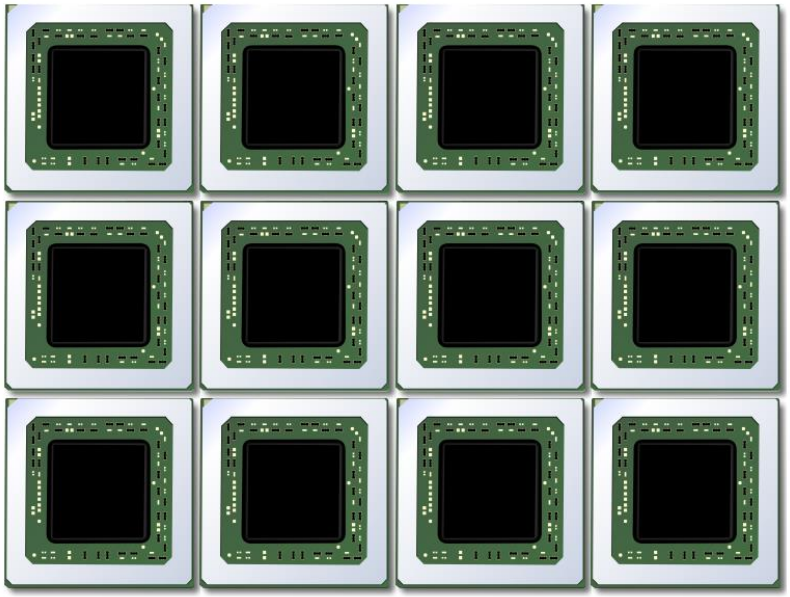


- Lower cost means can put more memory in each server

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

Modern Hardware for Big Data





Bunch of Hard Drives

…. and CPUs

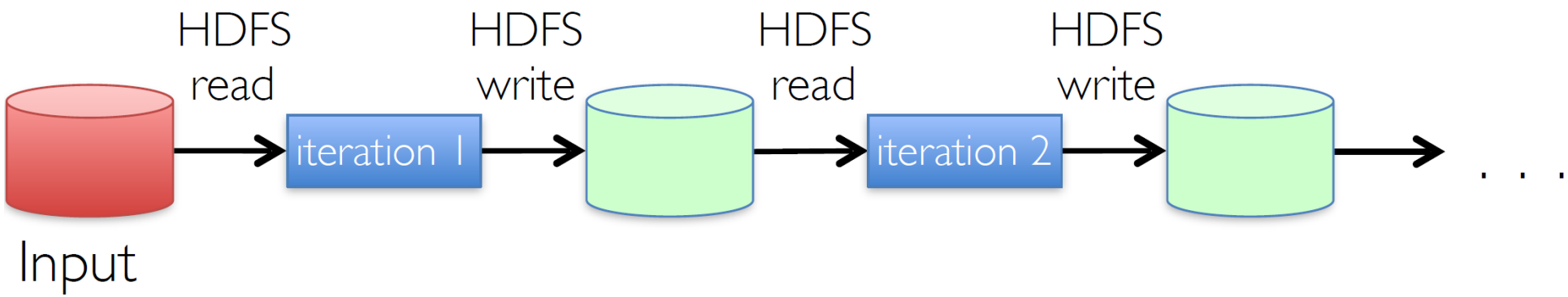
… and memory!

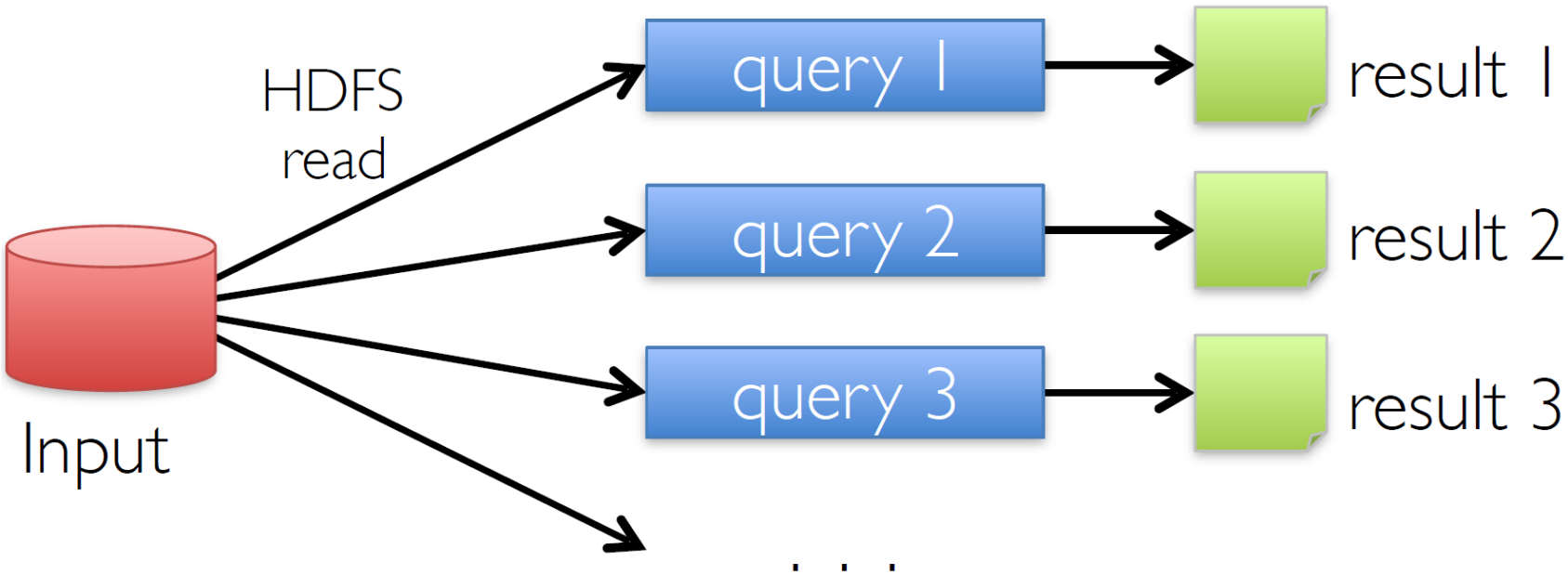
Opportunity

- Keep more data in-memory
- Create new distributed execution engine:
- One of the most efficient programming frameworks offering abstraction and parallelism for clusters
- It hides complexities of:
  - Fault Tolerance
  - Slow machines
  - Network Failures

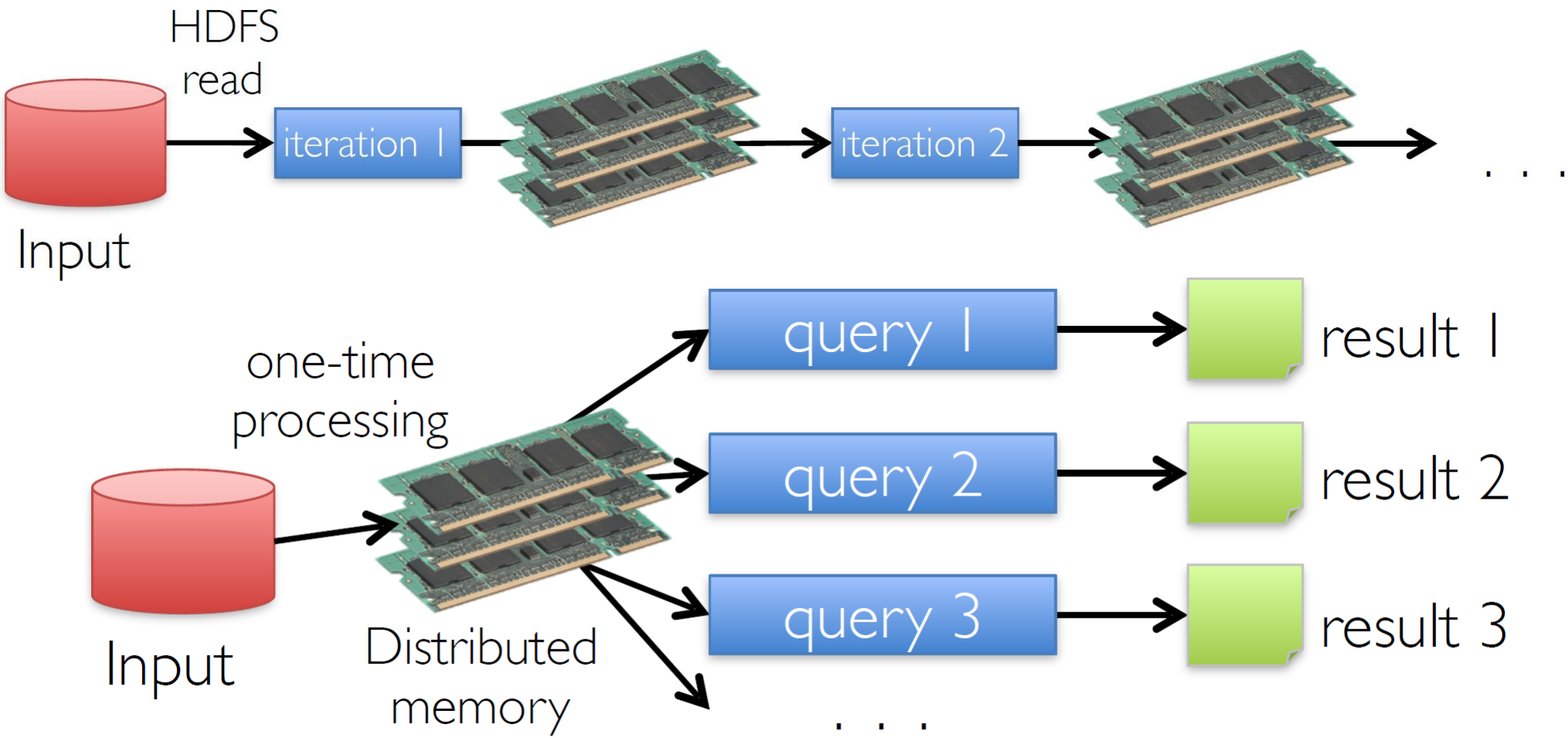
http://people.csail.mit.edu/matei/papers/2010/hotcloud_spark.pdf

Use Memory Instead of Disk





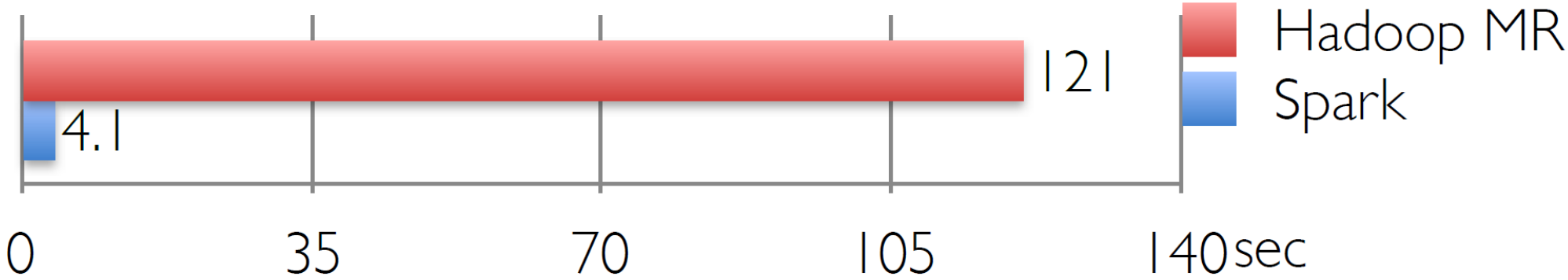
In-Memory Data Sharing



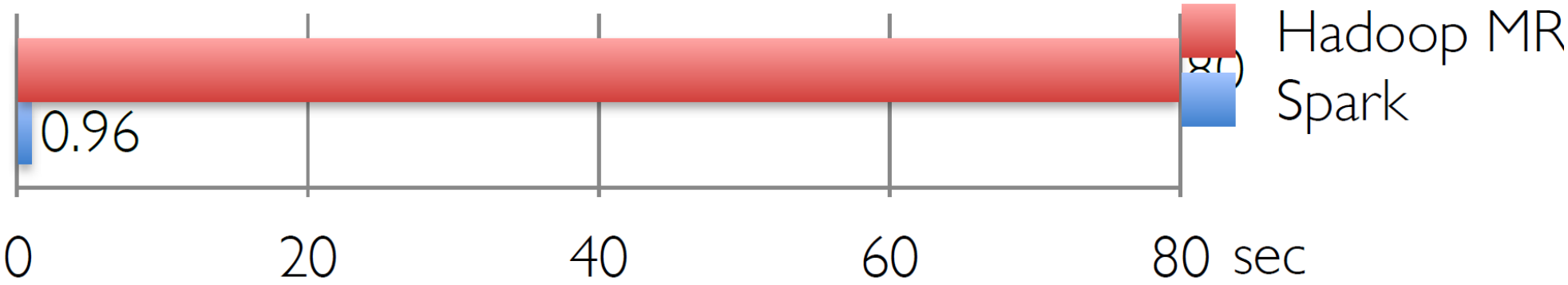
- 10-100x faster than network and disk!

In-Memory Can Make a Big Difference

- (2013) Two iterative Machine Learning algorithms:
  - K-means Clustering

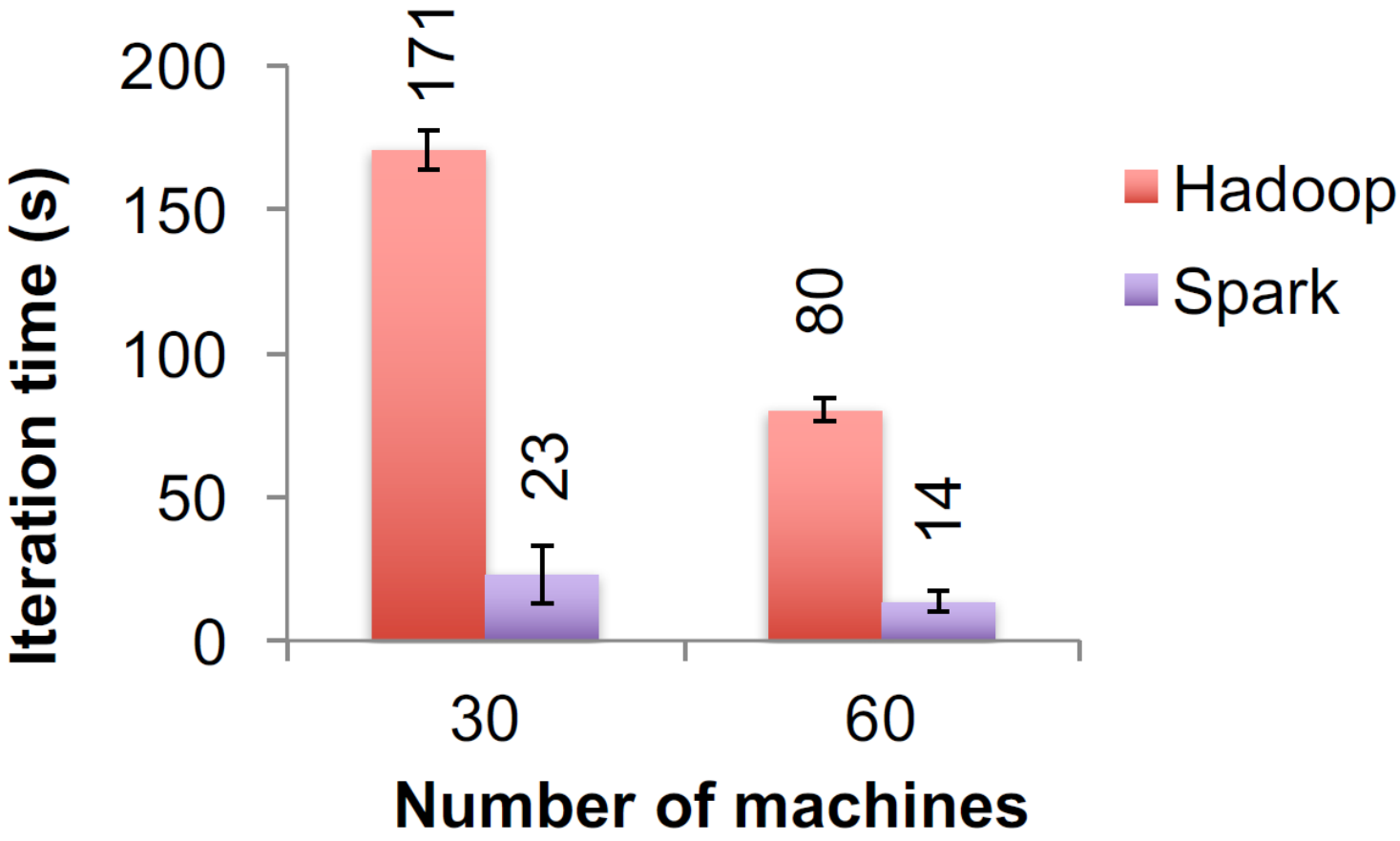


  - Logistic Regression



In-Memory Can Make a Big Difference

- PageRank



What is Apache Spark™?

RDDs
Transformations
Actions

Recall What's Hard with Big Data

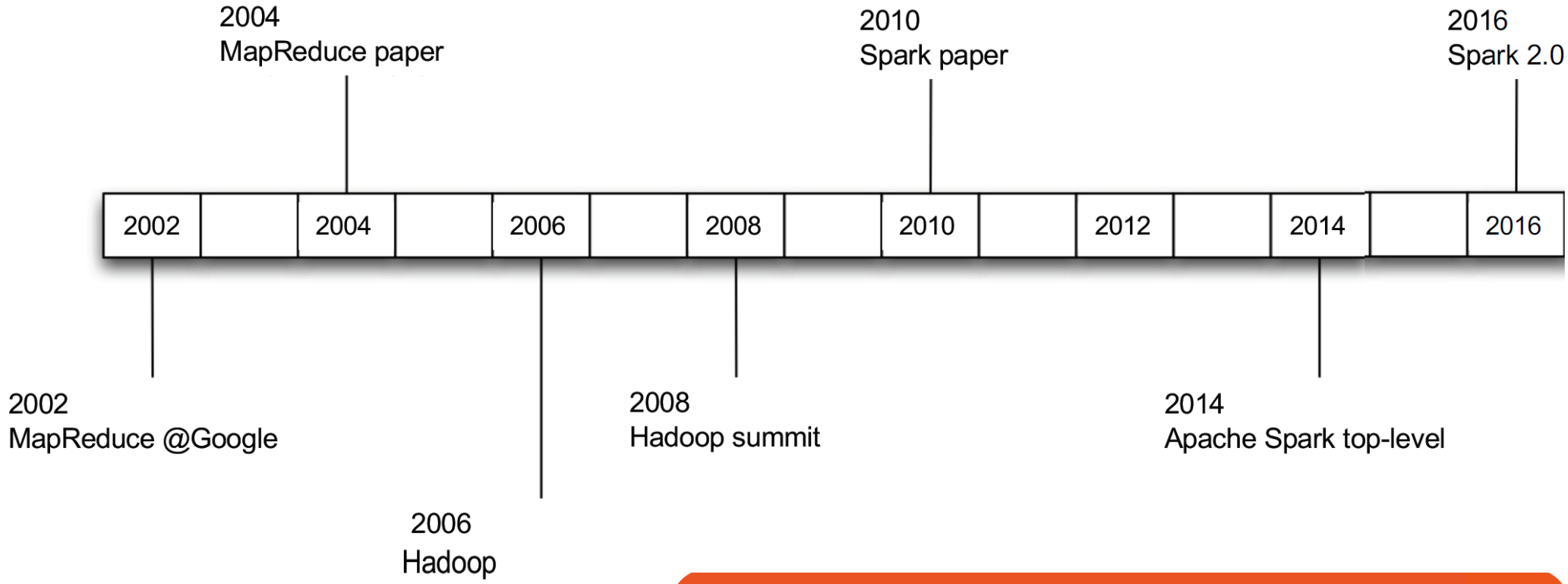
- Complex combination of processing tasks, storage, systems and modes
  - ETL, aggregation, streaming, machine learning

- Hard to get both productivity and performance!

Spark's Philosophy

- Unified Engine: Fewer Systems to Master
  - Express an entire pipeline in one API
  - Interoperate with existing libraries and storage
- Richer Programming Model: improves usability for complex analytics
  - High-level APIs (RDDs, Data Frames, Data Pipelines)
  - Scala/Java/Python/R
  - Interactive shell (repl)
  - 2-10x less code (than MapReduce)
- Memory Management: improves efficiency for complex analytics
  - Avoid materializing data on HDFS after each iteration:
    - ...up to 100x faster that Hadoop in memory
    - ...or 10x faster on disk
- New fundamental data abstraction that is
  - … easy to extend with new operators
  - … allows for a more descriptive computing model

A Brief History

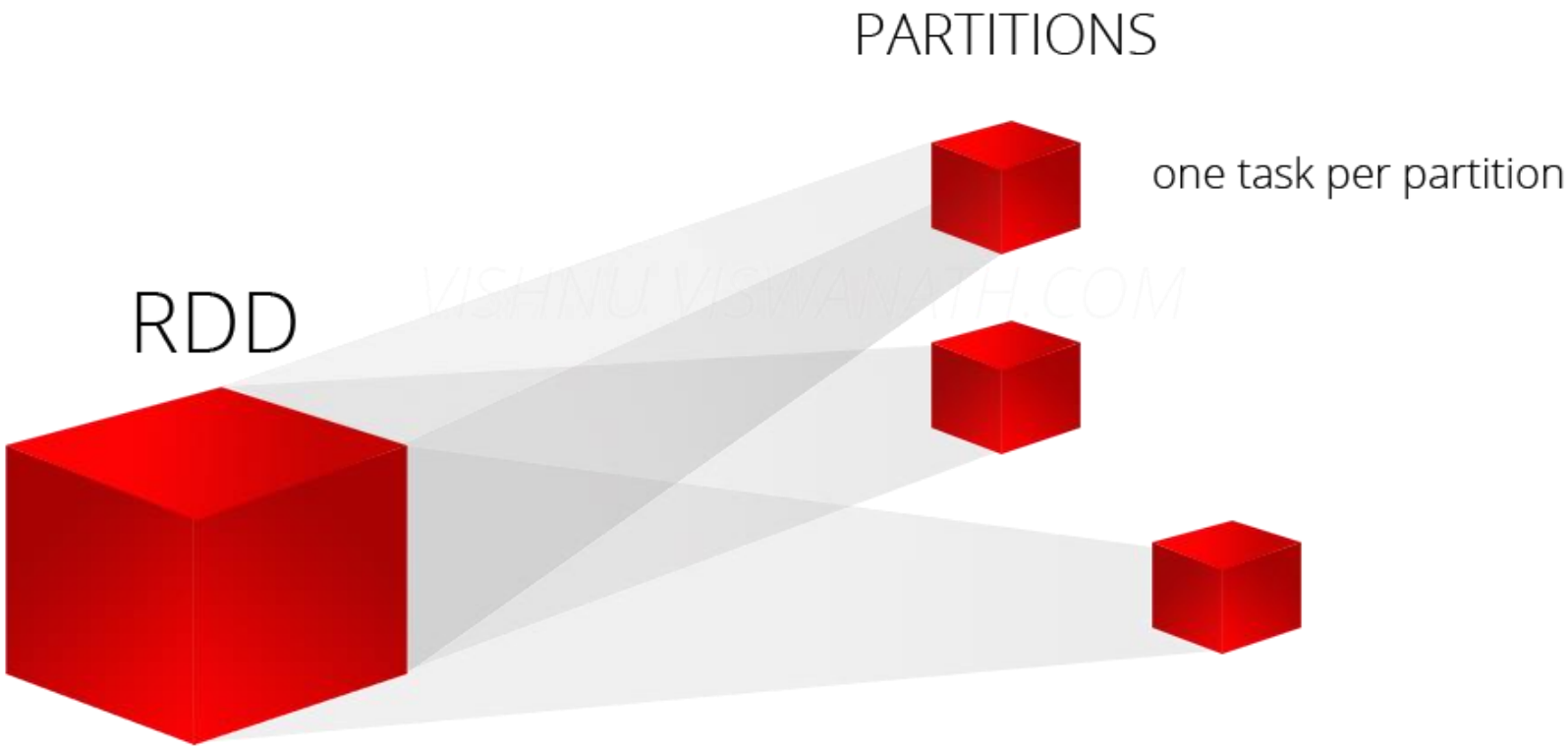


- Hadoop touched half of the world data in 2015
- Hadoop market is forecast to grow at a compound annual growth rate 58% surpassing $1 billion by 2020
- Software market rapidly shifting to Big Data 32% compound annual growth rate in EU through 2016

Resilient Distributed Dataset (RDDs)

- Immutable collection of objects spread across a cluster (partitions)
  - Immutable once they are created
- Build through parallel transformations (map, filter)
  - Diverse set of operators that offers rich data processing functionality
- Automatically rebuilt on (partial) failure
  - They carry their lineage for fault tolerance
- Controllable persistence (e.g., cashing in RAM)

RDD: Partitions

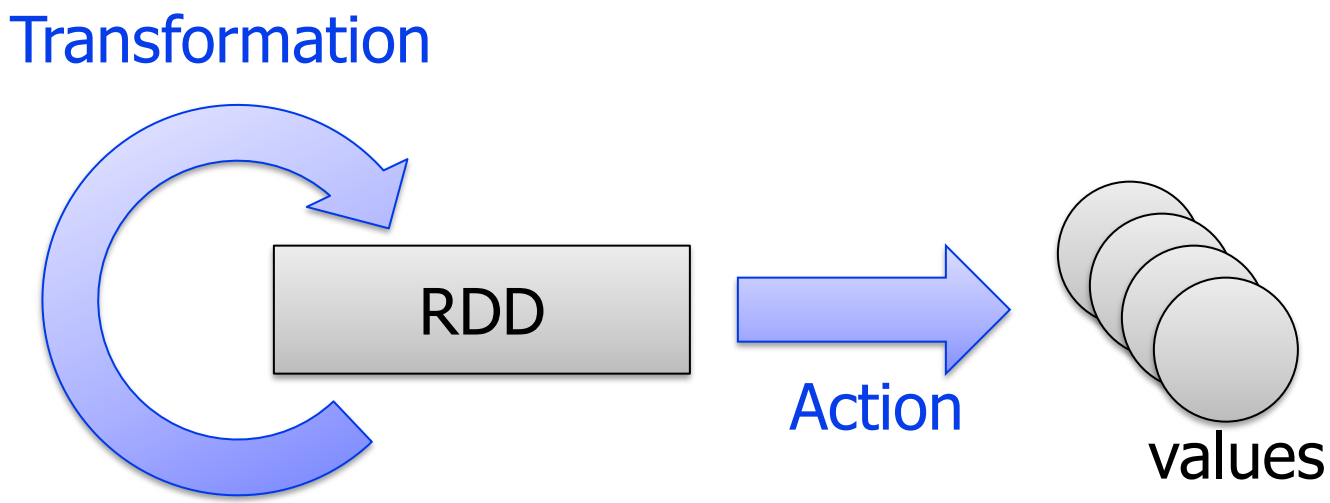


http://datalakes.com/rdds-simplified/

RDD: Partitions

- RDDs are automatically distributed across the network by means of partitions
  - A partition is a logical division of data
  - RDD data is just a collection of partitions
  - Spark automatically decides the number of partitions when creating an RDD
    - All input, intermediate and output data will be presented as partitions
  - Partitions are basic units of parallelism
  - A task is launched per each partition

Two Types of Operations on RDDs



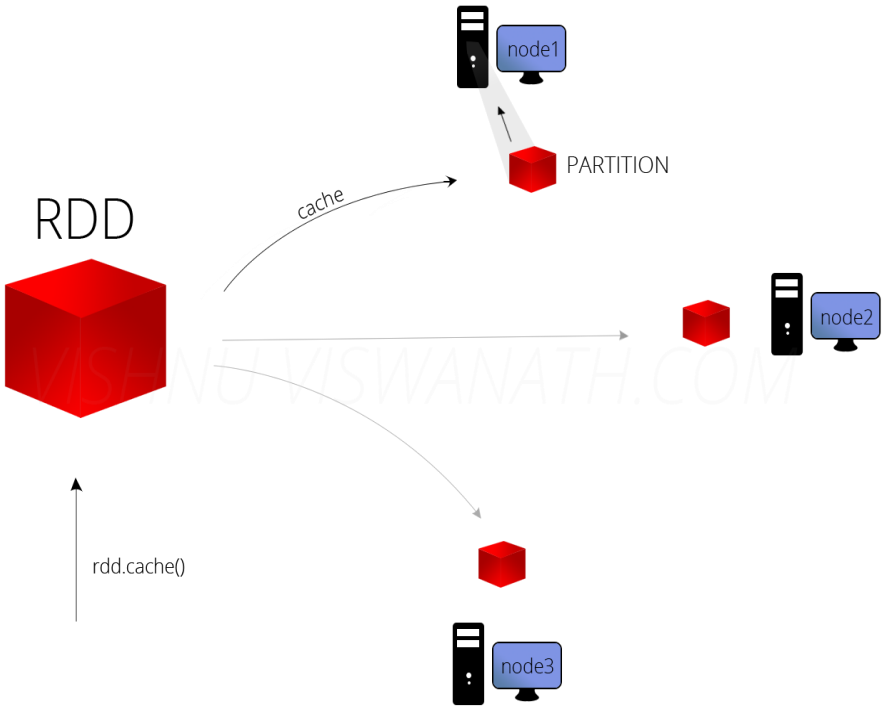
Transformations are lazy:
Framework keeps track of lineage

Actions trigger actual execution:
Transformations are executed when an action runs

- Operator cache persists distributed data in memory or disk

RDD Cache - rdd.cache()

- If we need the results of an RDD many times, it is best to cache it
  - RDD partitions are loaded into the memory of the nodes that hold it
  - avoids re-computation of the entire lineage
  - in case of node failure compute the lineage again

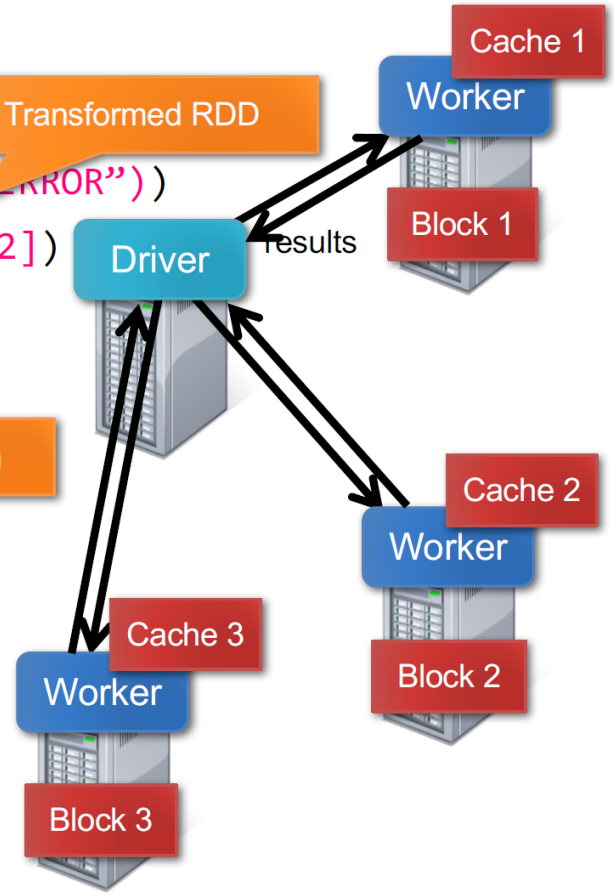


http://datalakes.com/rdds-simplified/

Example: Mining Console Logs

- Load error messages from a log into memory, then interactively search for patterns

```
logLines = spark.textFile("hdfs://...")
errorsRDD = logLines.filter(lambda s: s.startswith("ERROR"))
messagesRDD = errorsRDD.map(lambda s: s.split('\t')[2])
messagesRDD.cache()


messagesRDD.filter(lambda s: "foo" in s).count()
messagesRDD.filter(lambda s: "bar" in s).count()
. . .
```

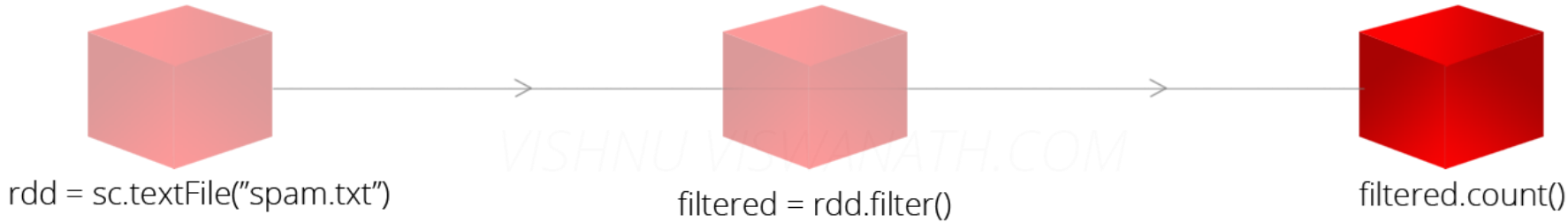


Result: full-text search of Wikipedia in < 5 sec (vs 20 sec for on-disk data)

Result: scaled to 1 TB of data in 5-7 sec (vs 170 sec for on-disk data)

RDD operations - Transformations

- As in relational algebra, the application of a transformation to an RDD yields a new RDD (immutability)
- Transformations are lazily evaluated which allow for optimizations to take place before execution
  - The lineage keeps track of all transformations that have to be applied when an action happens

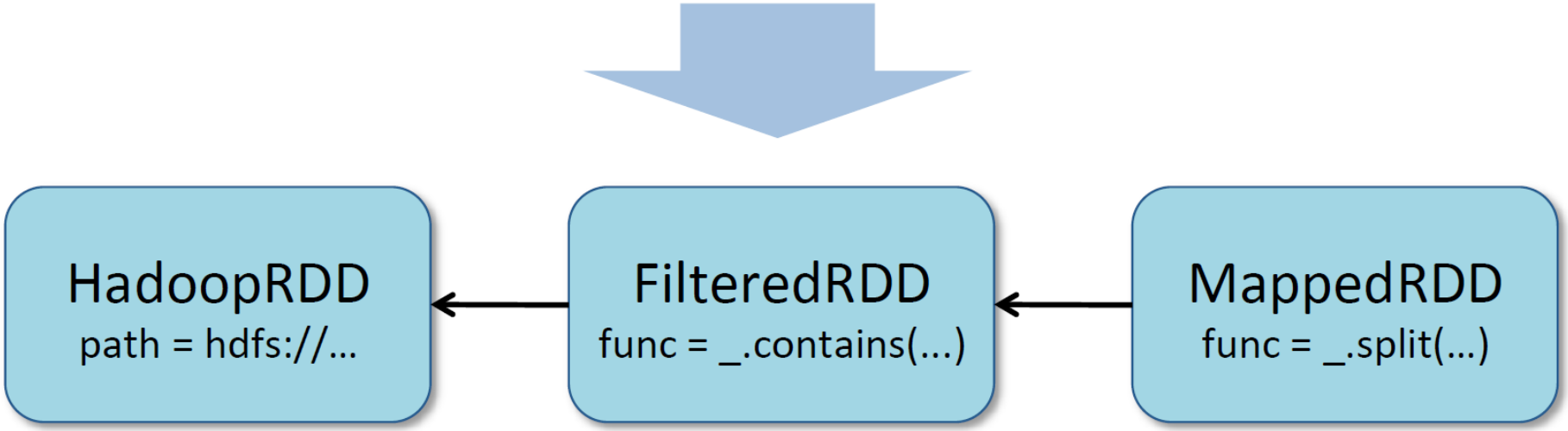


http://datalakes.com/rdds-simplified/

RDD Lineage (aka Logical Logging)

- RDDs track the transformations used to build them (their lineage) to recompute lost data

```
messages = textFile(...).filter(_.contains("error"))
                         .map(_.split('\t')(2))
```



M. Zaharia, et al, Resilient Distributed Datasets: A fault--tolerant abstraction for in--memory cluster computing, NSDI 2012

Useful Transformations on RDDs

Transformation	Description
map	Return a new distributed dataset formed by passing each element of the source through a function *func*.
mapPartitions	Similar to map, but runs separately on each partition (block) of the RDD, so *func* must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.
filter	Return a new dataset formed by selecting those elements of the source on which *func* returns true.
sample	Sample a fraction *fraction* of the data, with or without replacement, using a given random number generator seed.
repartition	Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network.

More Useful Transformations on RDDs

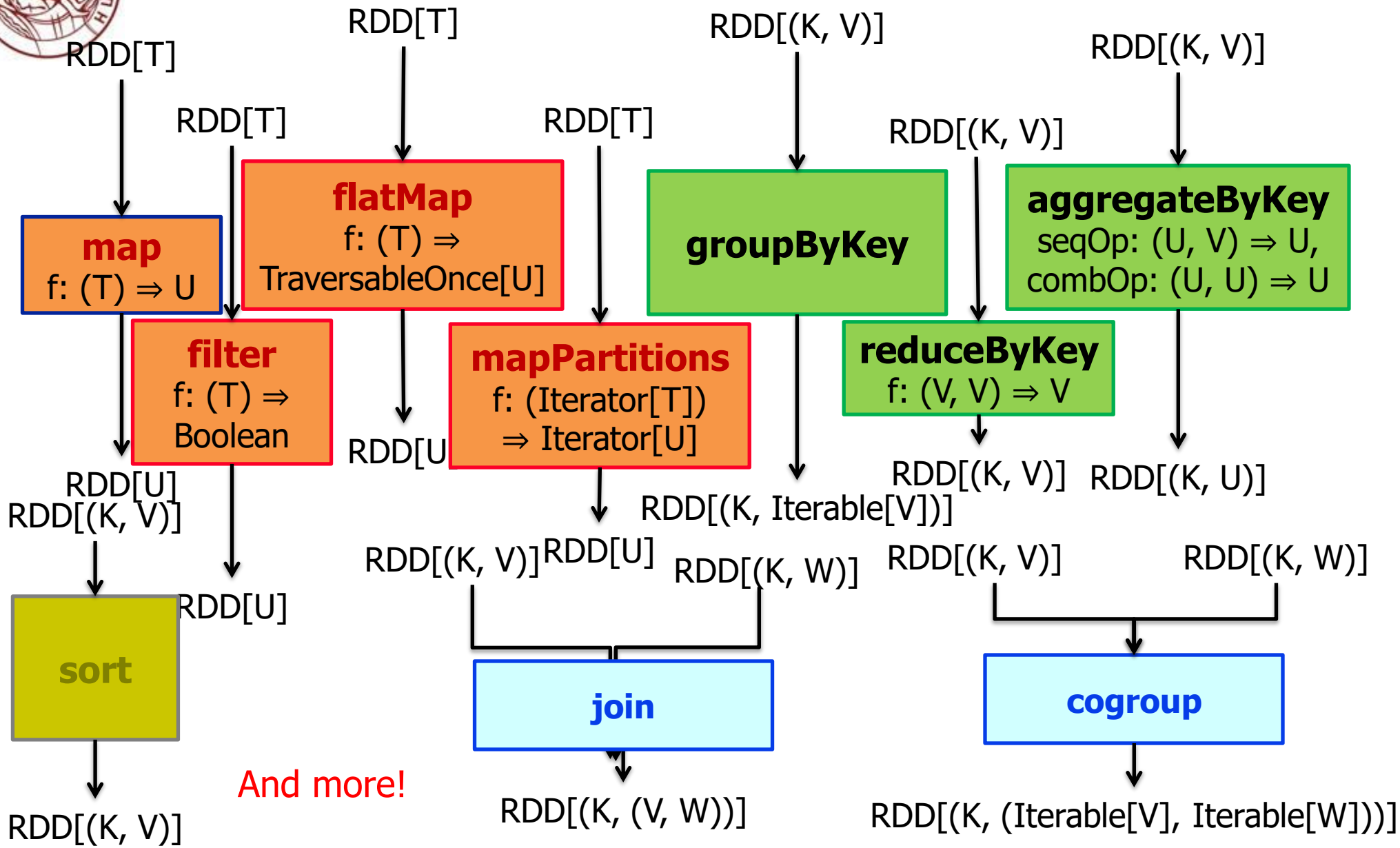
Transformation	Description
groupByKey	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs.
reduceByKey	When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) => V. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.
aggregateByKey	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.
sortByKey	When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument.
join	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through leftOuterJoin, rightOuterJoin, and fullOuterJoin.

RDD Common Transformations: Examples

Unary	RDD	Result
rdd.map(x => x * x)	{1, 2, 3, 3}	{1, 4, 9, 9}
rdd.flatMap(line => line.split(" "))	{"hello world", "hi"}	{"hello", world", "hi"}
rdd.filter(x => x != 1)	{1, 2, 3, 3}	{2, 3, 3}
rdd.distinct ()	{1, 2, 3, 3}	{1, 2, 3}

Binary	RDD1	RDD2	Result
rdd.union(other)	{1, 2, 3}	{3,4,5}	{1,2,3,3,4,5}
rdd.intersection(other)	{1, 2, 3}	{3,4,5}	{3}
rdd.subtract(other)	{1, 2, 3}	{3,4,5}	{1, 2}
rdd.cartesian(other)	{1, 2, 3}	{3,4,5}	{(1,3),(1,4), … (3,5)}

Useful Transformations on RDDs



RDD operations - Actions

- Apply transformation chains on RDDs, eventually performing some additional operations (e.g. counting)
  - i.e. trigger job execution
- Used to materialize computation results
- Some actions only store data from the RDD upon which the action is applied and convey it to the driver

RDD Actions

Action	Description
Take(n)	Return an array with the first *n* elements of the dataset.
TakeOrdered(n)	Return the first *n* elements of the RDD using either their natural order or a custom comparator.
First	Return the first element of the dataset (similar to take(1)(0)).
Collect	Return all the elements of the dataset as an array at the driver program. This is 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.
Reduce	Aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.

RDD Actions: Examples

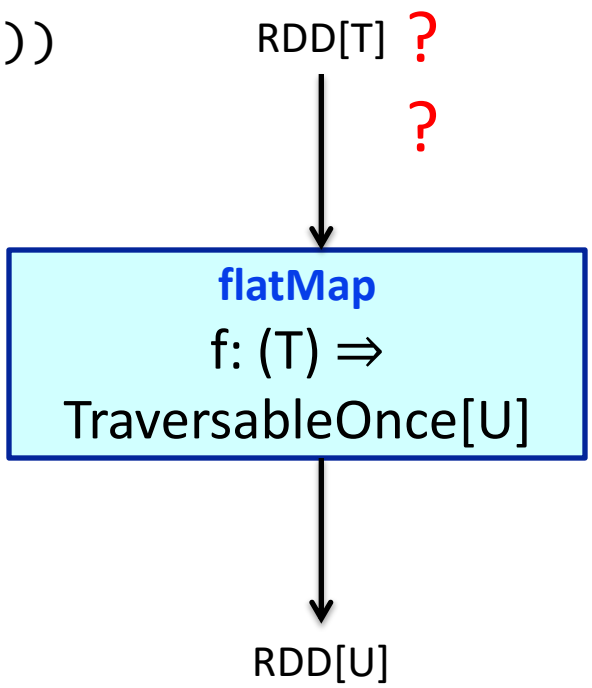
	RDD	Result
rdd.reduce((x, y) => x + y)	{1,2,3}	6
rdd.foreach(x=>println(x))	{1,2,3}	prints “1 2 3”

	RDD	Result
rdd.collect()	{1,2,3}	{1,2,3}
rdd.first()	{1,2,3,4}	1
rdd.count()	{1,2,3,3}	4
rdd.max()	{1,2,3,3}	3
rdd.top(2)	{1,2,3,3}	{3,3}

	RDD	Result
rdd.countByKey()	{(a,x),(a,y),(b,x)}	{(a,2),(b,1)}

Spark Word Count

```
val textFile = sc.textFile(args.input())

textFile
  .flatMap(line => tokenize(line))
  .map(word => (word, 1))
  .reduceByKey((x, y) => x + y)
  .saveAsTextFile(args.output())
```



Spark Word Count

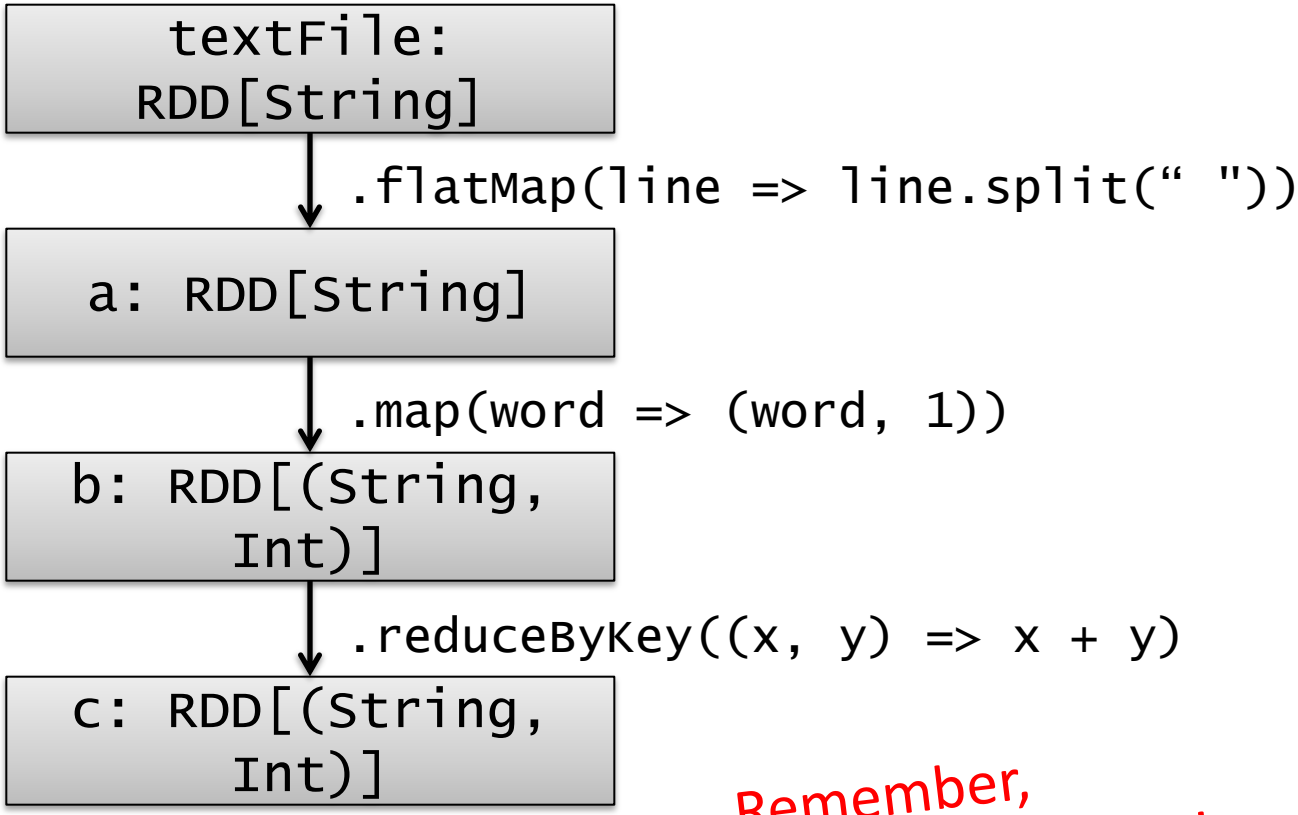
RDDs

```
val textFile = sc.textFile(args.input())

val a = textFile.flatMap(line => line.split(" "))
val b = a.map(word => (word, 1))
val c = b.reduceByKey((x, y) => x + y)

c.saveAsTextFile(args.output())
```

Action

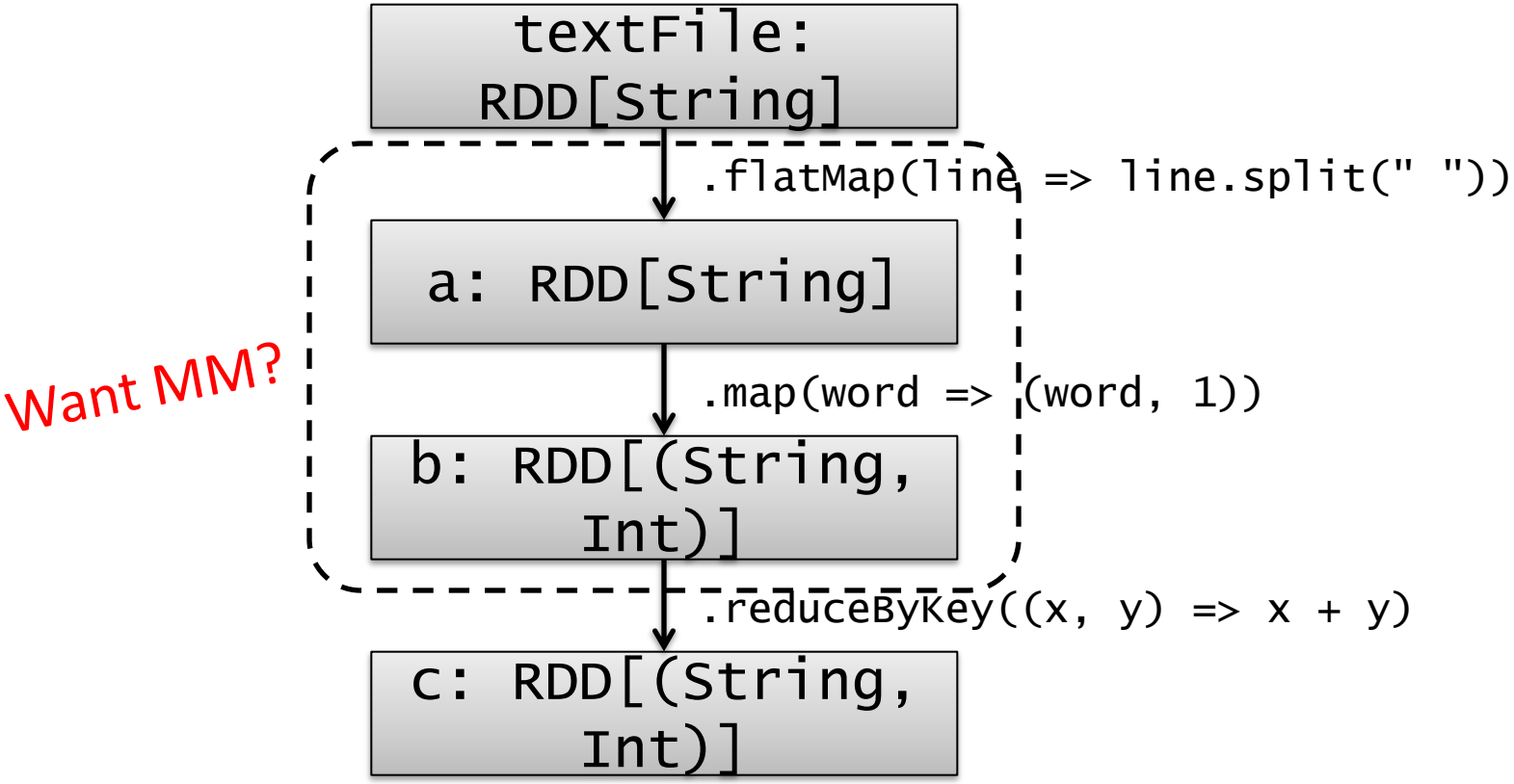
Transformations

RDDs and Lineage



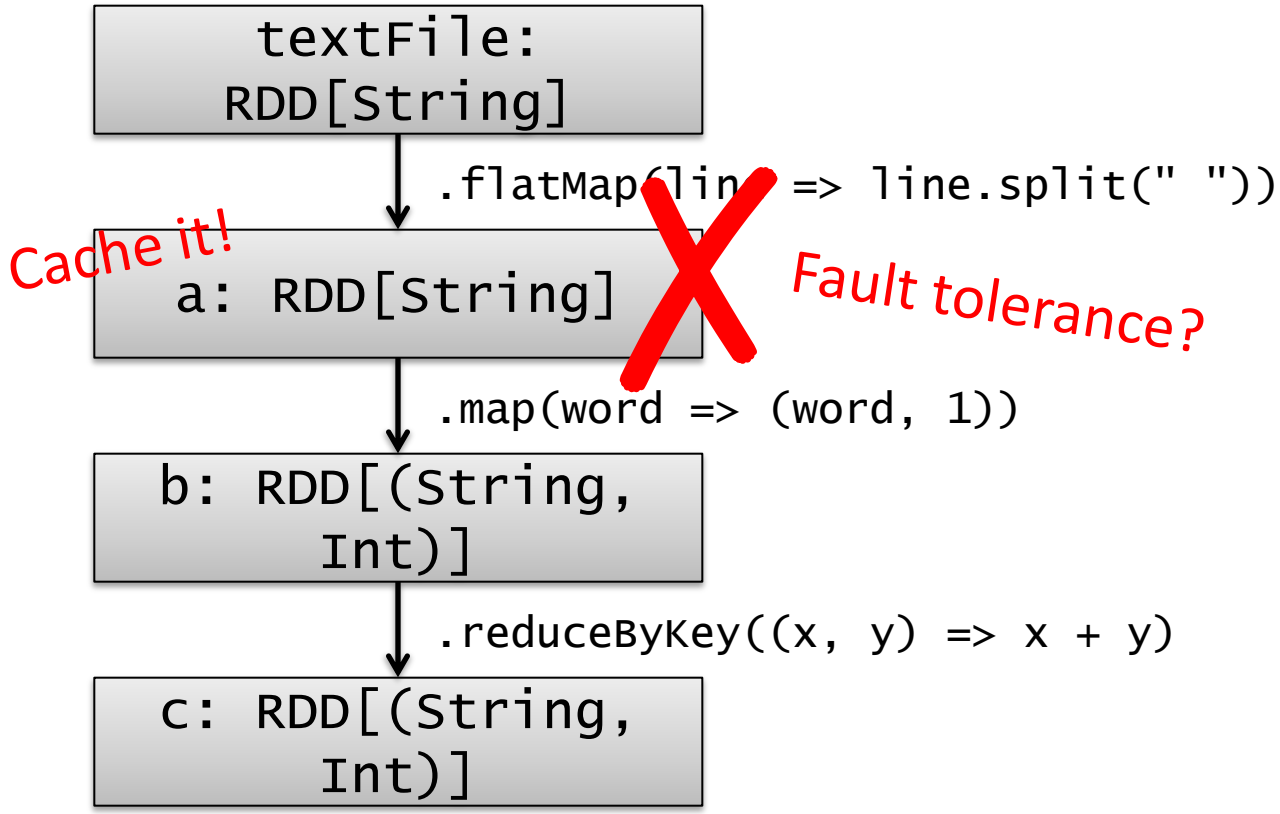
Remember,
transformations are lazy!

RDDs and Optimizations



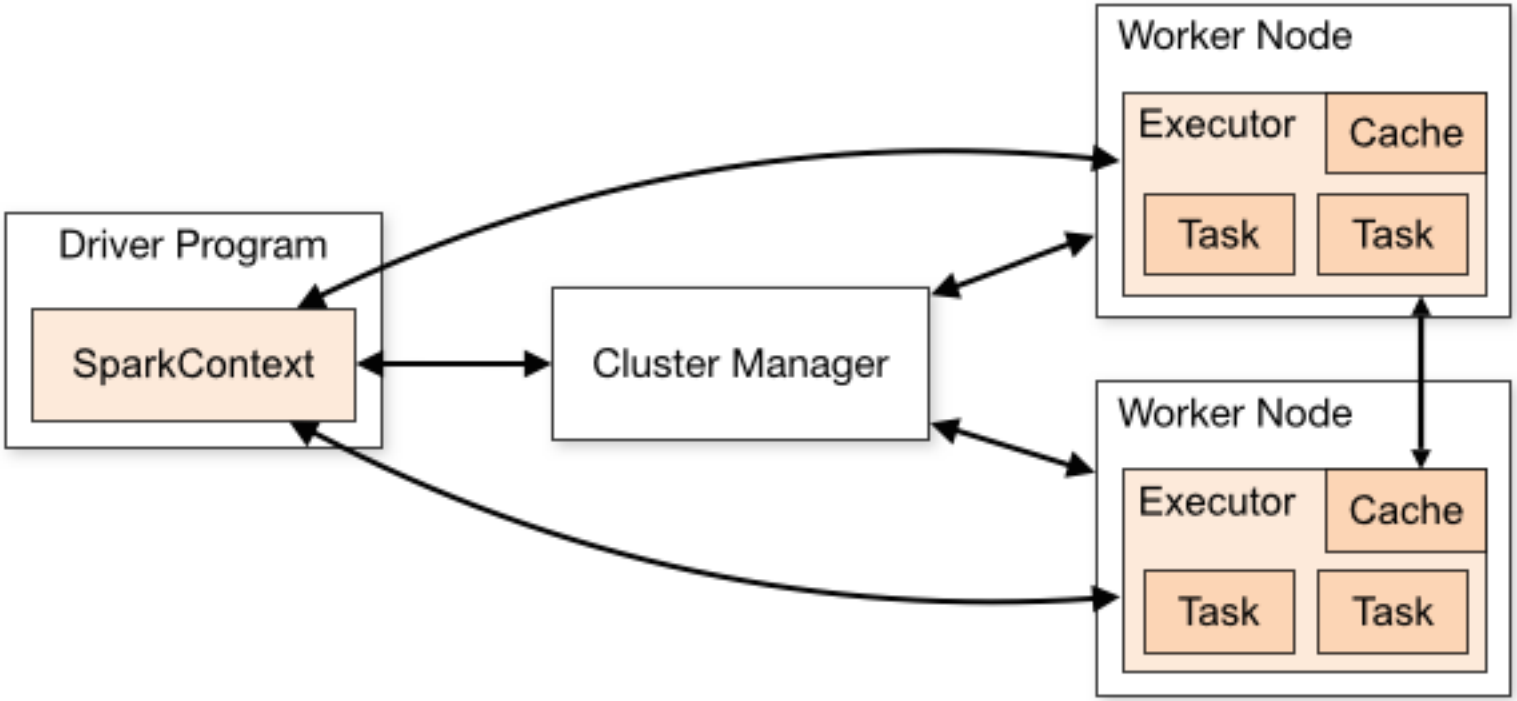
RDDs and Caching

RDDs can be materialized in memory (and on disk)!

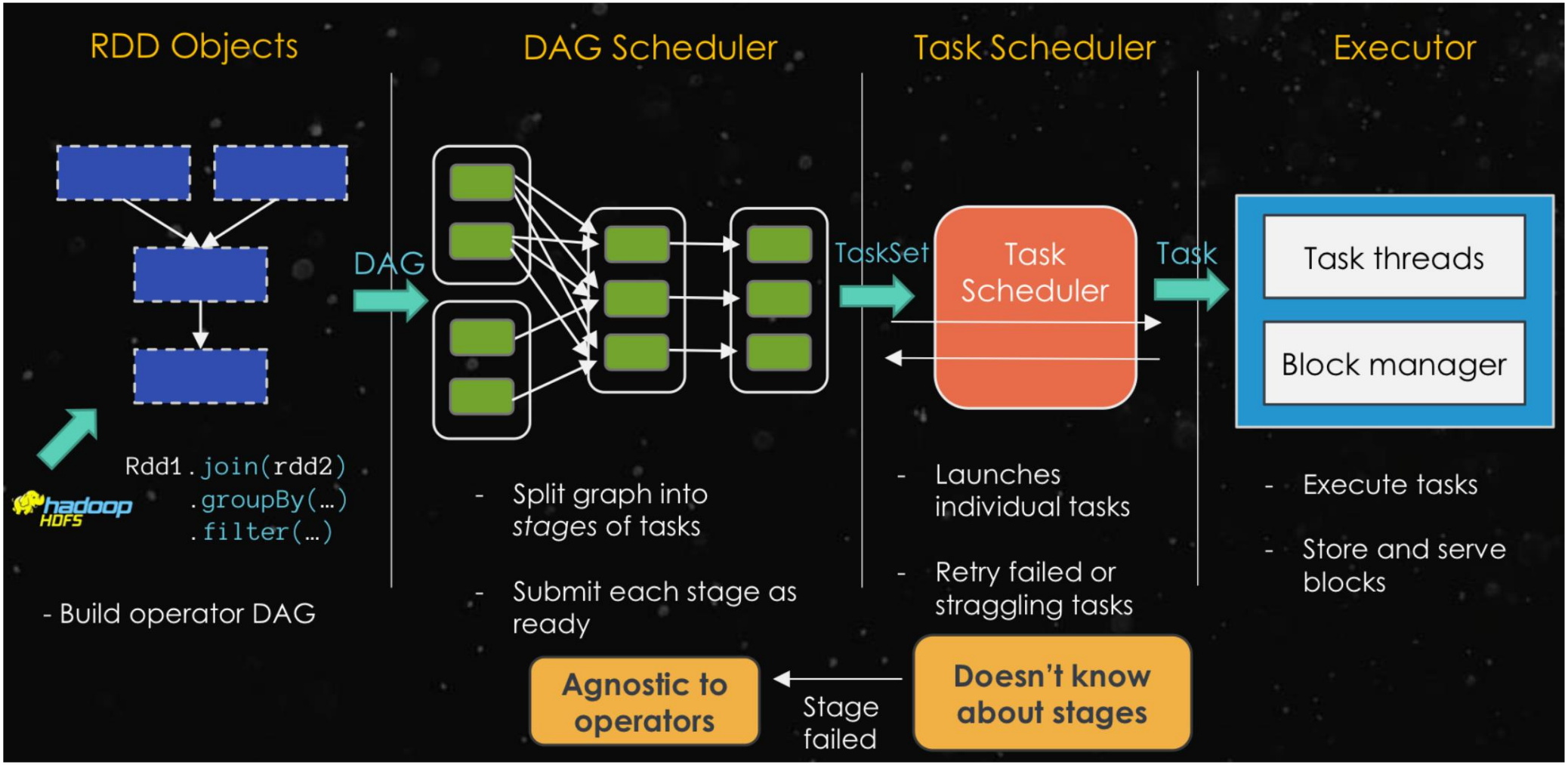


Spark works even if the RDDs are *partially* cached!

Spark Architecture



Scheduling Process

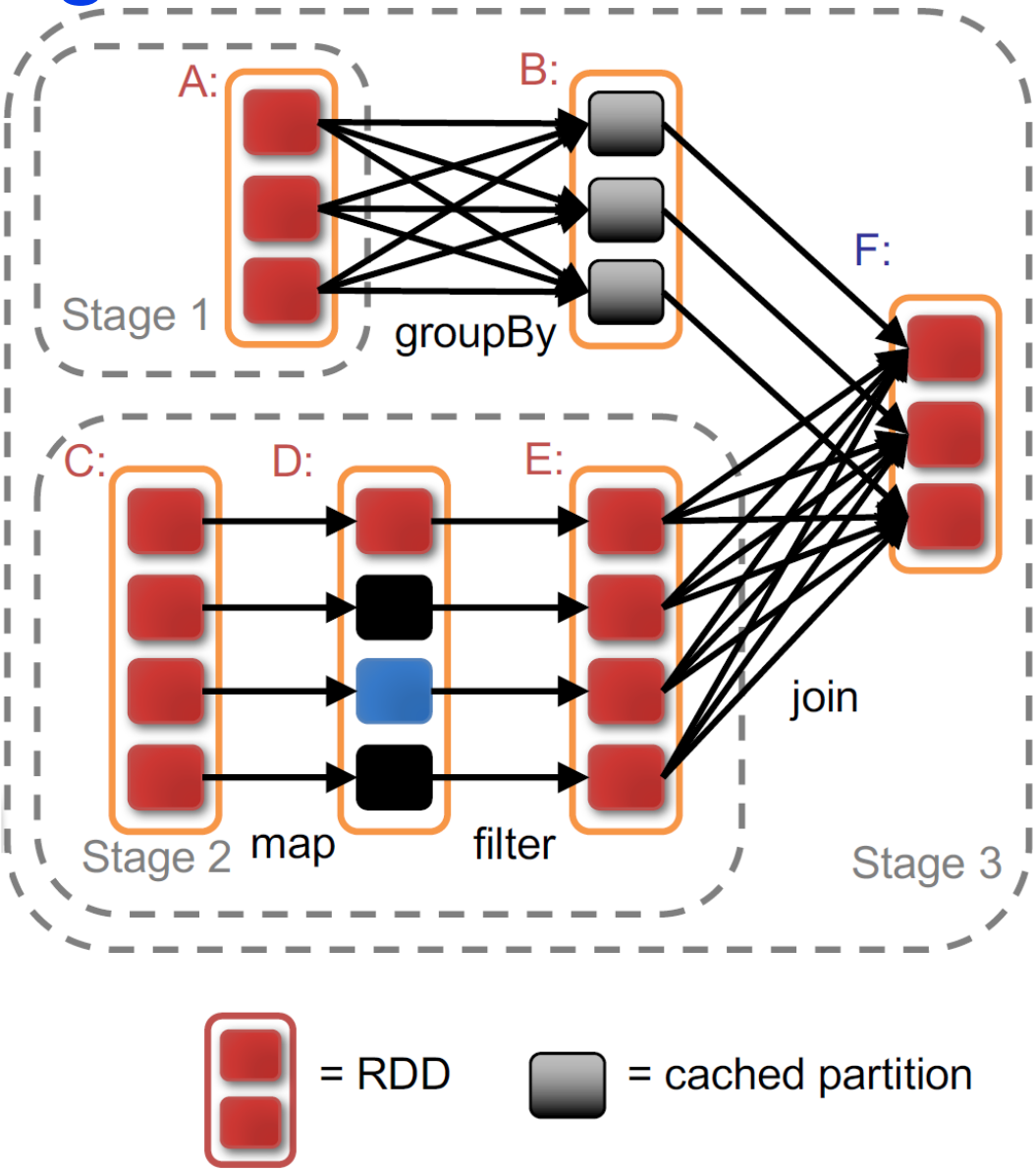


Scheduling Problems

- Supports general task graphs
- Pipelines functions where is possible
- Cache-aware data reuse and locality
- Partitioning-aware to avoid shuffles

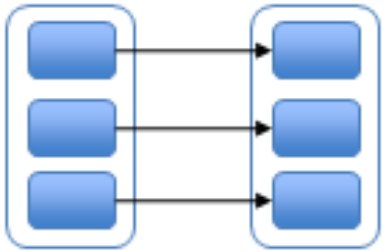
Potential bottleneck?

Shuffle phase
- implemented through disk
- random I/O writes are problematic

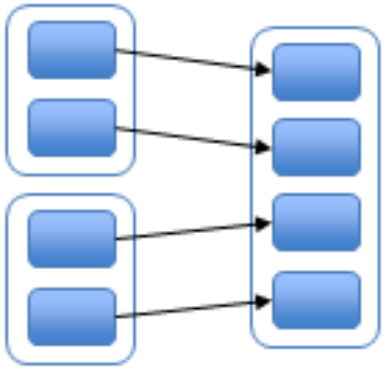


Narrow vs Wide Dependencies

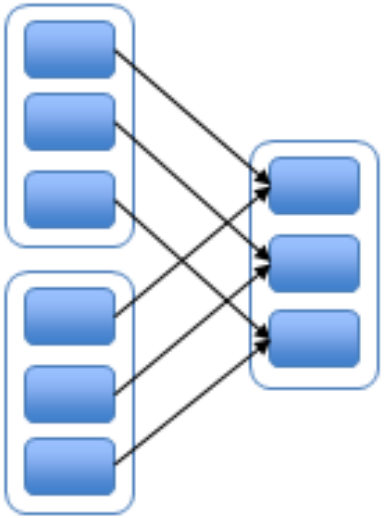
"Narrow" deps:



map, filter

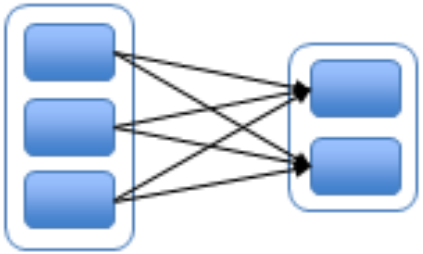


union

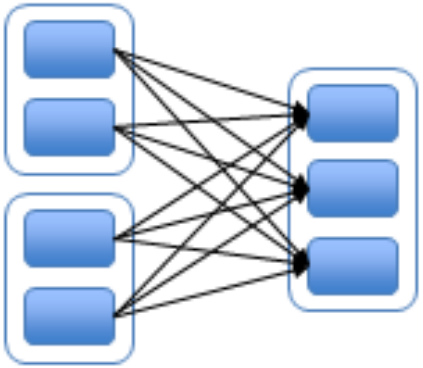


join with inputs co-partitioned

"Wide" (shuffle) deps:



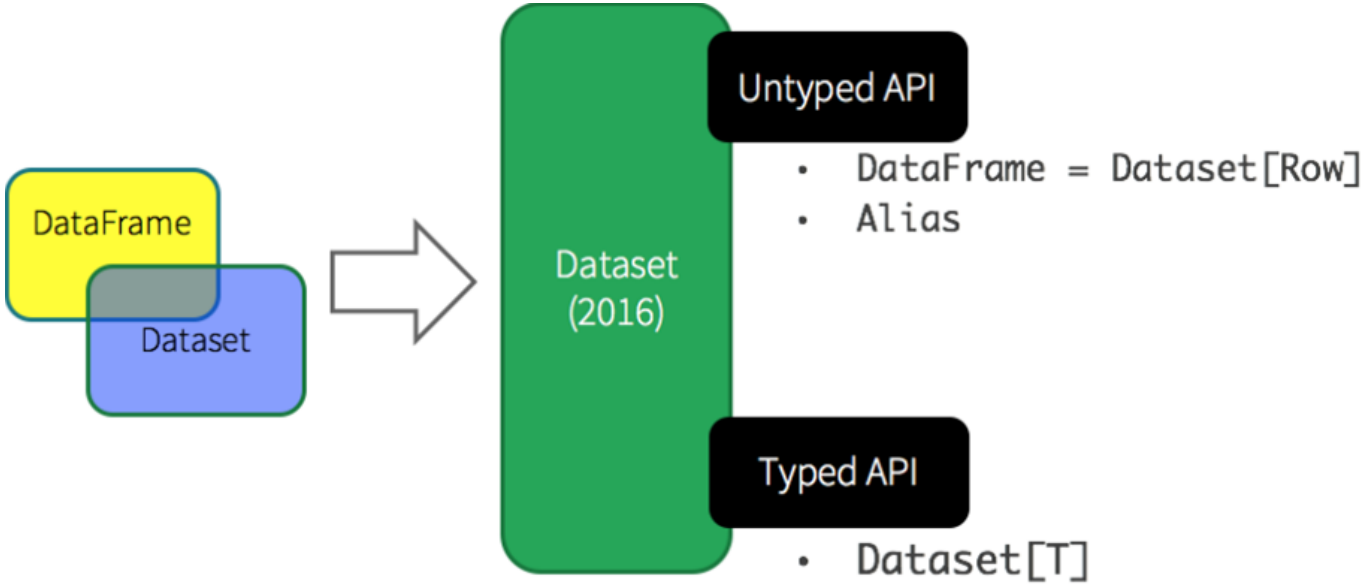
groupByKey



join with inputs not co-partitioned

https://trongkhoanguyen.com/spark/understand-rdd-operations-transformations-and-actions/

DataFrames & DataSets



- In 2015 Spark added DataFrames and Datasets as structured data APIs
- DataFrames are collections of rows with a fixed schema (table-like)
- Datasets add static types, e.g. Dataset[Person]

https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html

Static-Typing and Runtime Type-safety in Spark

	SQL	DataFrames	Datasets
Syntax Errors	Runtime	Compile Time	Compile Time
Analysis Errors	Runtime	Runtime	Compile Time

- Analysis errors reported before a distributed job starts

DataFrames: Example

```
case class User(name: String, id: Int)
case class Message(user: User, text: String)

dataframe = sqlContext.read.json("log.json")      // DataFrame, i.e. Dataset[Row]
messages = dataframe.as[Message]                  // Dataset[Message]

users = messages.filter(m => m.text.contains("Spark"))
                .map(m => m.user)                 // Dataset[User]

pipeline.train(users)                  // MLlib takes either DataFrames or Datasets
```

Where “Database Thinking” Can Get In The Way

Traditional Database Thinking

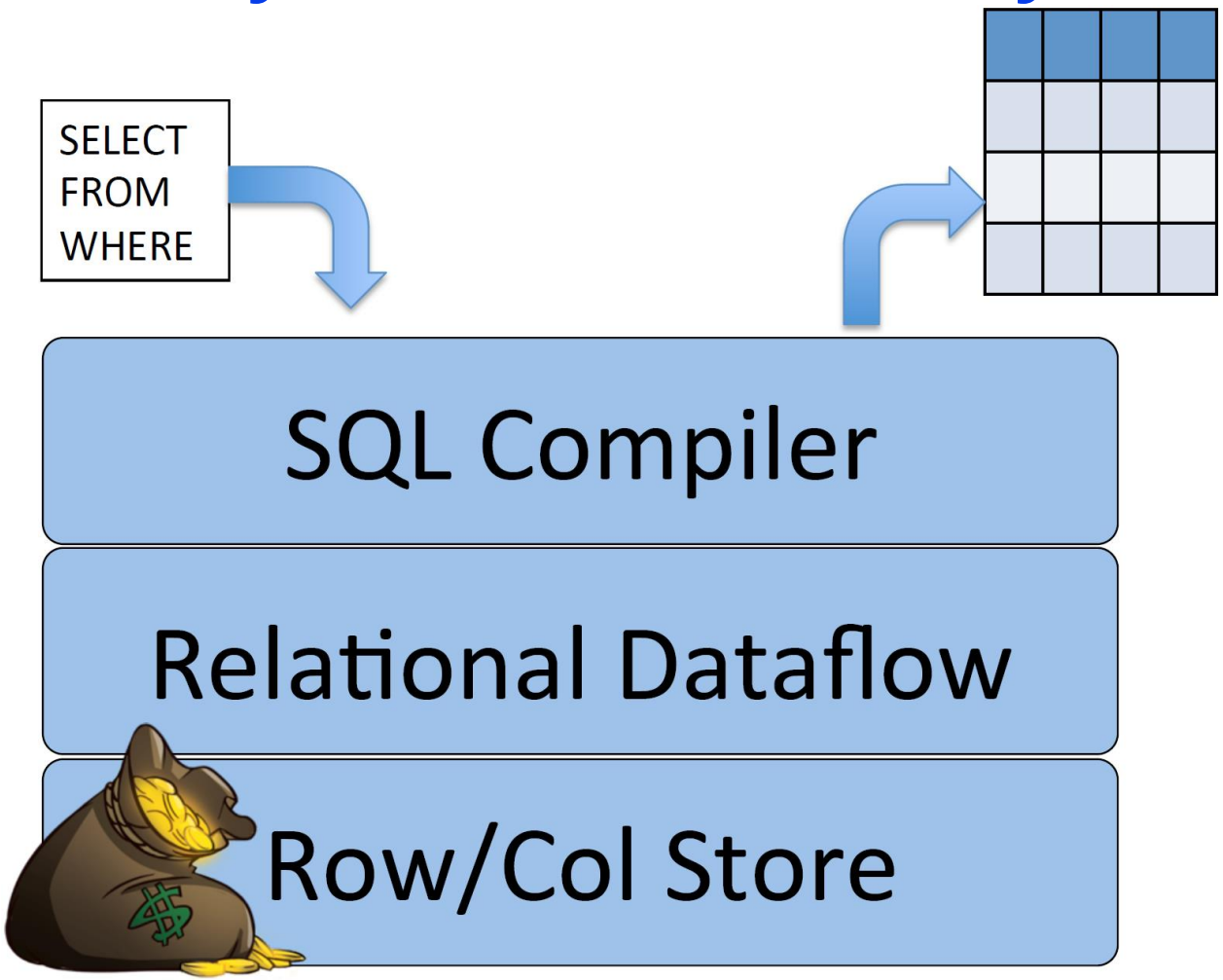
Pros

- Declarative Queries and Data Independence
  - Rich Query Operators, Plans and Optimization
  - Separation of Physical and Logical Layers
- Data existing independently of applications
  - Not as natural to most people as you'd think
- Importance of managing the storage hierarchy

Cons

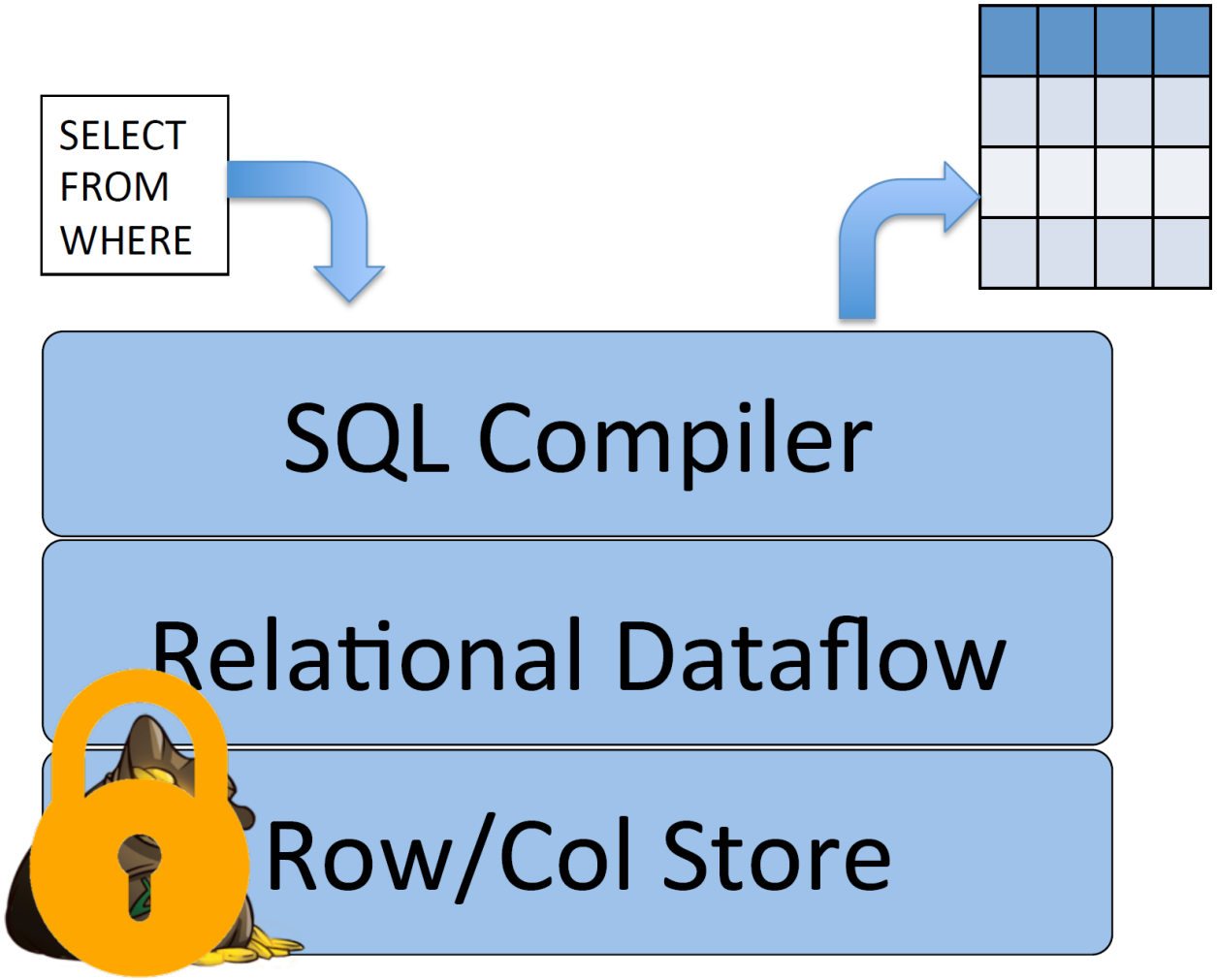
- Monolithic Systems and Control
- Schema First & High Friction
- The DB Lament: "We've seen it all before"

Database Systems: One Way In/Out



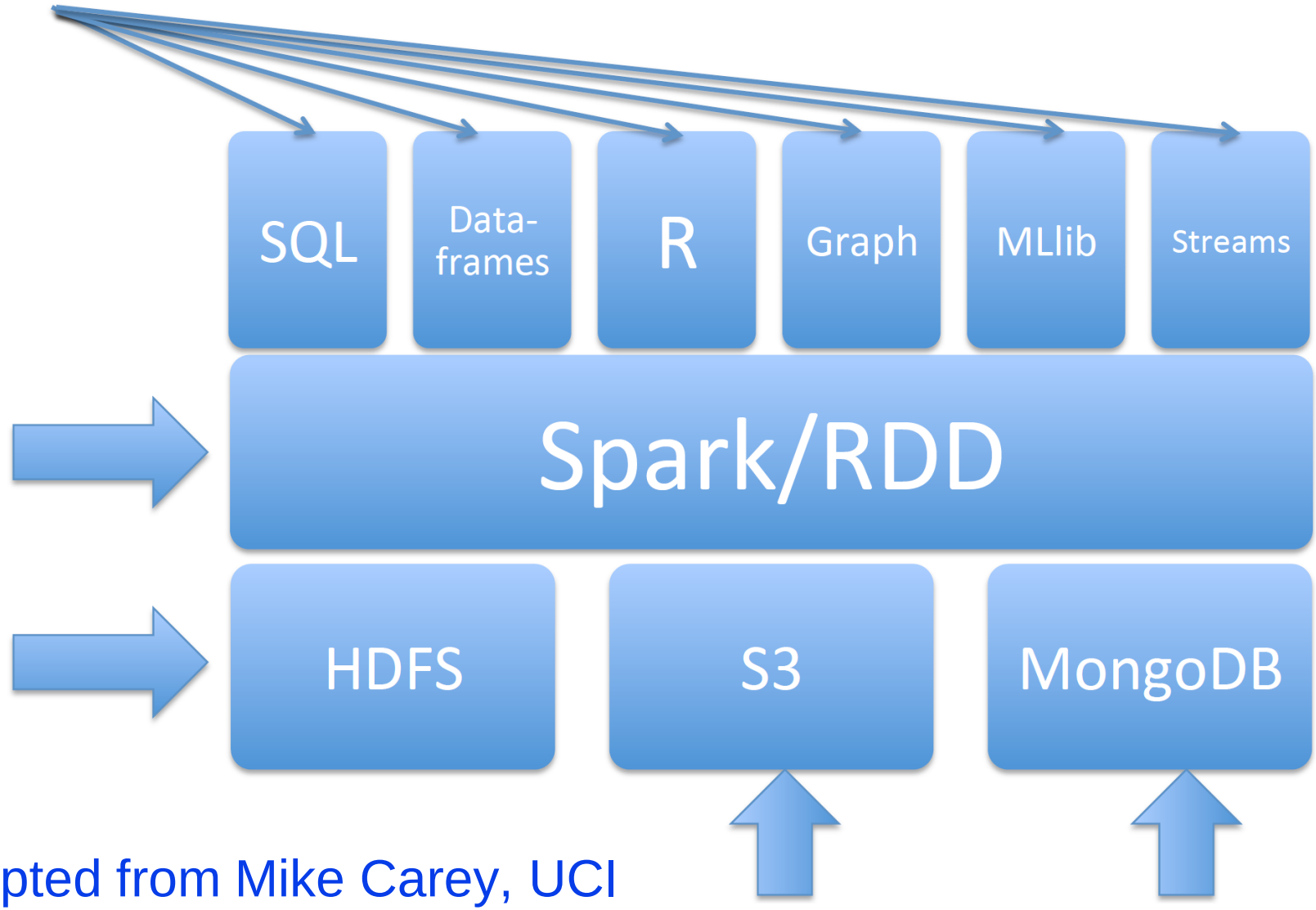
Adapted from Mike Carey, UCI

Database Systems: One Way In/Out



Adapted from Mike Carey, UCI

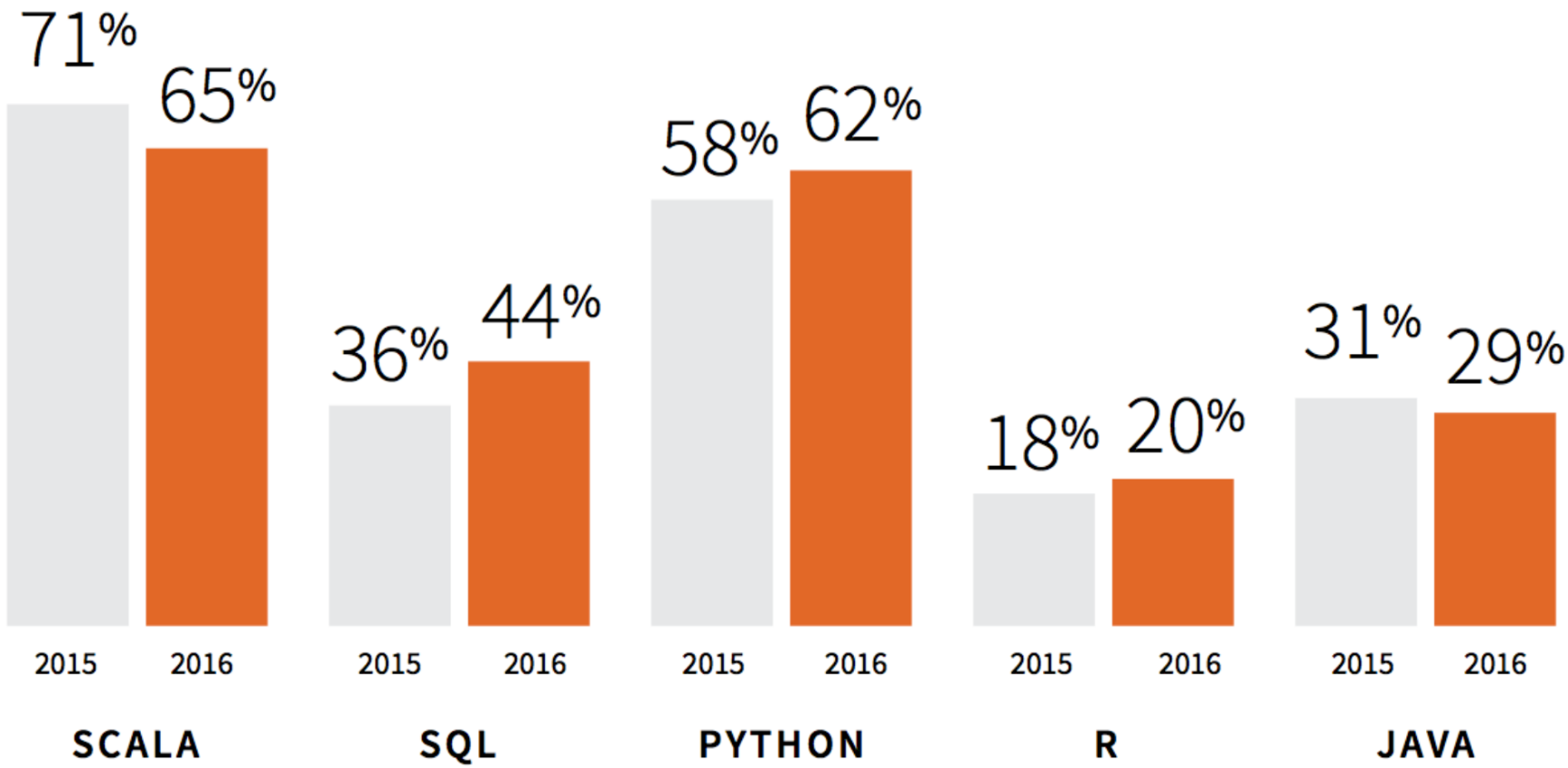
Mix and Match Data Access



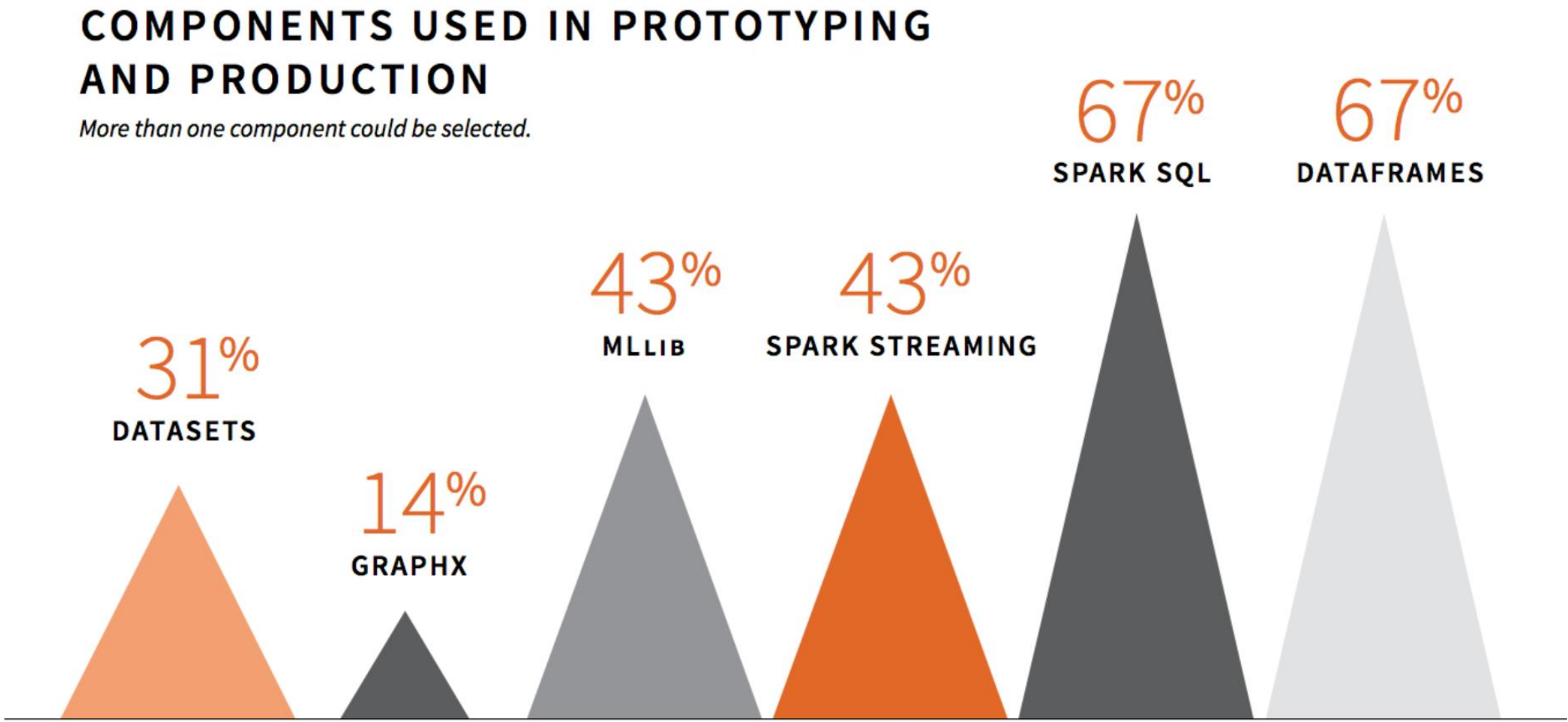
Adapted from Mike Carey, UCI

Q: WHICH LANGUAGES DO YOU USE SPARK IN?

% of respondents who use each language (more than one language could be selected)



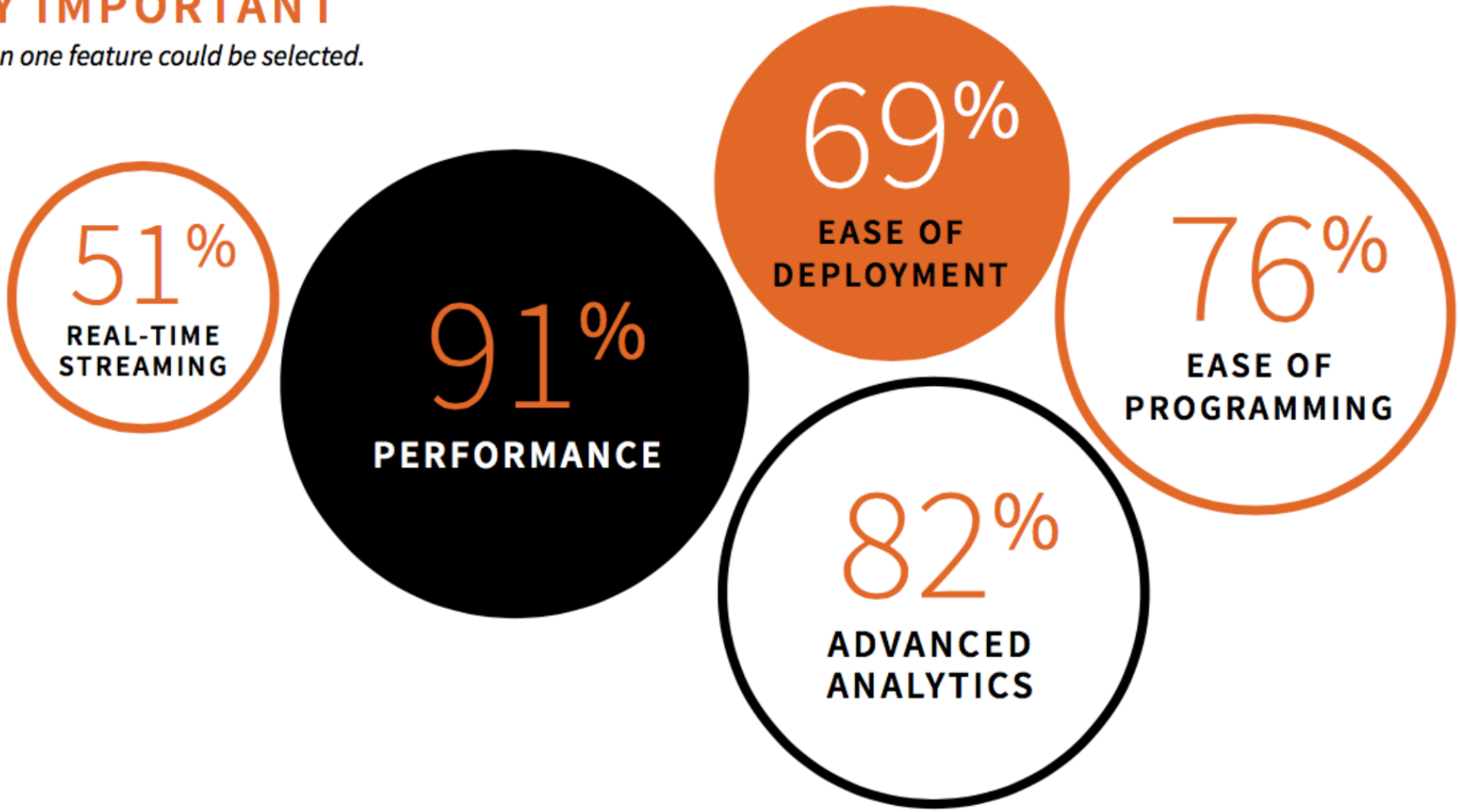
From: Spark User Survey 2016, 1615 respondents from 900 organizations
http://go.databricks.com/2016--spark--survey



From: Spark User Survey 2016, 1615 respondents from 900 organizations
http://go.databricks.com/2016--spark--survey

% OF RESPONDENTS WHO CONSIDERED THE FEATURE VERY IMPORTANT

More than one feature could be selected.



Spark Ecosystem Features

- Spark focus was initially on
  - Performance + Scalability with Fault Tolerance
- Rapid evolution of functionality kept it growing especially across multiple modalities:
  - DB,
  - Graph,
  - Stream,
  - ML,
  - etc.
- Database thinking is moving Spark and much of the Hadoop ecosystem up the disruptive technology value curve

Spark and Map Reduce Differences

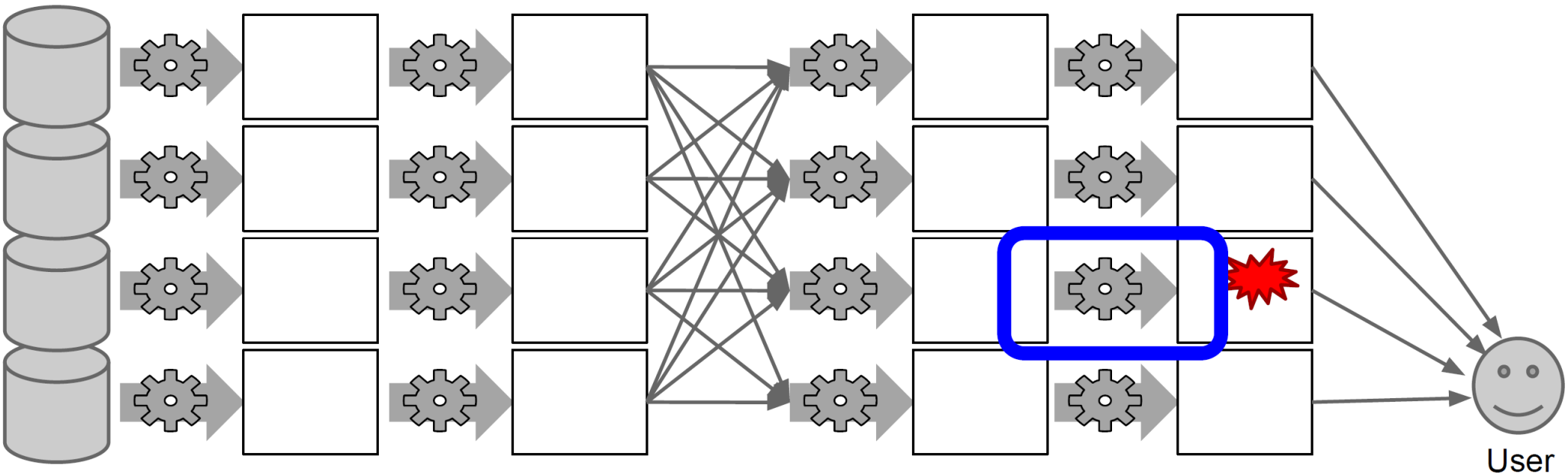
	Apache Hadoop MapReduce	Apache Spark
Storage	Disk only	In-memory or on disk
Operations	Map and Reduce	Many transformation and actions, including Map and Reduce
Execution model	Batch	Batch, interactive, streaming
Languages	Java	Scala, Java, R, and Python

Other Spark and Map Reduce Differences

- Generalized patterns for computation
  - provide unified engine for many use cases
  - require 2-5x less code
- Lazy evaluation of the lineage graph
  - can optimize, reduce wait states, pipeline better
- Lower overhead for starting jobs
- Less expensive shuffles

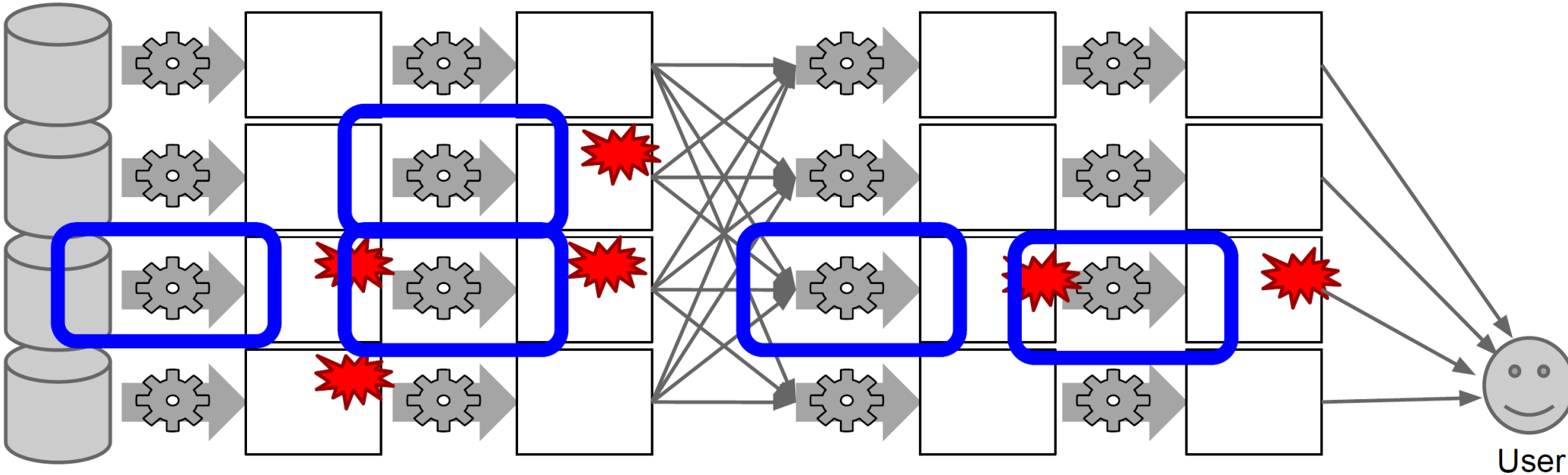
Spark: Fault Tolerance

- Hadoop: Once computed, don't lose it
- Spark: Remember how to re-compute

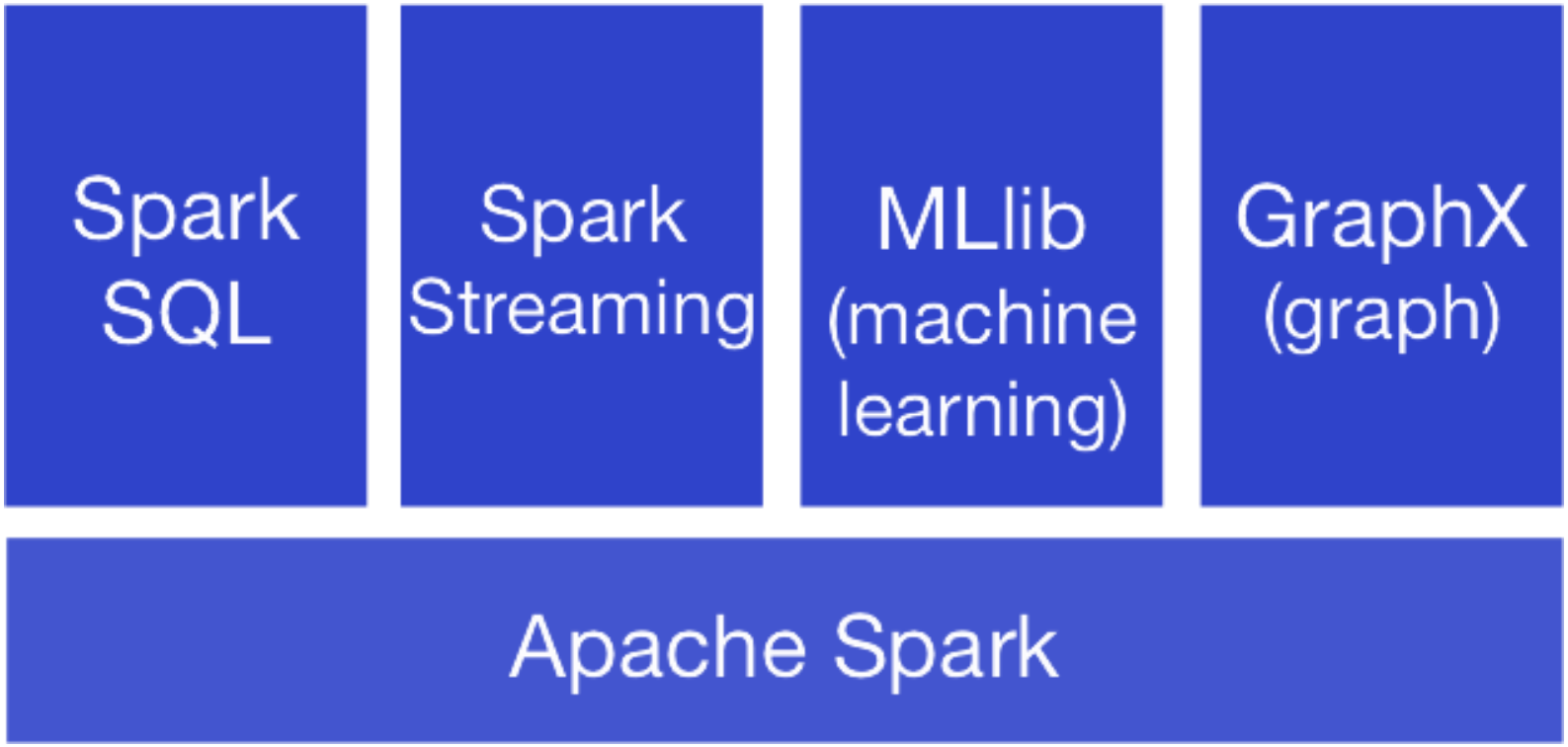


Spark: Fault Tolerance

- Hadoop: Once computed, don't lose it
- Spark: Remember how to re-compute



Apache Spark Software Stack: Unified Vision



- Spark Unified pipeline can run today's most advanced algorithms

vs Apache Hadoop

General Batching	Specialized systems			
	Streaming	Iterative	Ad-hoc / SQL	Graph
MapReduce	Storm	Mahout	Pig	Giraph
	S4		Hive	
	Samza		Drill	
			Impala	

- Sparse Modules
- Diversity of APIs
- Higher Operational Costs

Conclusions

- The Database field is seeing tremendous change from above and below
- Big Data software is a classic Disruptive Technology
- Database Thinking is key to moving up the value chain
- But we'll also have to shed some of our traditional inclinations in order to make progress

Problems Suited for Map-Reduce

Example: Host size

- **Suppose we have a large web corpus**
- Look at the metadata file
  - Lines of the form: (URL, size, date, …)
- **For each host, find the total number of bytes**
  - That is, the sum of the page sizes for all URLs from that particular host
- **Other examples:**
  - Link analysis and graph processing
  - Machine Learning algorithms

Example: Language Model

- **Statistical machine translation:**
  - Need to count number of times every 5-word sequence occurs in a large corpus of documents
- **Very easy with MapReduce:**
  - **Map:**
    - Extract (5-word sequence, count) from document
  - **Reduce:**
    - Combine the counts

Example: Join By Map-Reduce

- **Compute the natural join *R(A,B)* ⋈ *S(B,C)***
- *R* and *S* are each stored in files
- Tuples are pairs *(a,b)* or *(b,c)*

A	B
a_1	b_1
a_2	b_1
a_3	b_2
a_4	b_3

R

⋈

B	C
b_2	c_1
b_2	c_2
b_3	c_3

S

=

A	C
a_3	c_1
a_3	c_2
a_4	c_3

Map-Reduce Join

- **Use a hash function *h* from B-values to *1...k***
- **A Map process turns:**
  - Each input tuple *R(a,b)* into key-value pair *(b,(a,R))*
  - Each input tuple *S(b,c)* into *(b,(c,S))*
- **Map processes** send each key-value pair with key *b* to Reduce process *h(b)*
  - Hadoop does this automatically; just tell it what *k* is.
- Each **Reduce process** matches all the pairs *(b,(a,R))* with all *(b,(c,S))* and outputs *(a,b,c)*.

Cost Measures for Algorithms

- **In MapReduce we quantify the cost of an algorithm using**

1. *Communication cost* = total I/O of all processes
2. *Elapsed communication cost* = max of I/O along any path
3. (*Elapsed*) *computation cost* analogous, but count only running time of processes

Note that here the big-O notation is not the most useful (adding more machines is always an option)

Example: Cost Measures

- **For a map-reduce algorithm:**
  - **Communication cost =** input file size + 2 × (sum of the sizes of all files passed from Map processes to Reduce processes) + the sum of the output sizes of the Reduce processes.
  - **Elapsed communication cost** is the sum of the largest input + output for any map process, plus the same for any reduce process

What Cost Measures Mean

- Either the I/O (communication) or processing (computation) cost dominates
  - Ignore one or the other
- Total cost tells what you pay in rent from your friendly neighborhood cloud
- Elapsed cost is wall-clock time using parallelism

Cost of Map-Reduce Join

- **Total communication cost**
  $= O(|R|+|S|+|R \bowtie S|)$
- **Elapsed communication cost** = O(s)
  - We're going to pick ***k*** and the number of Map processes so that the I/O limit ***s*** is respected
  - We put a limit ***s*** on the amount of input or output that any one process can have. ***s* could be:**
    - What fits in main memory
    - What fits on local disk
- With proper indexes, computation cost is linear in the input + output size
  - So computation cost is like comm. cost

References

- John Canny Distributed Analytics CS194-16 Introduction to Data Science UC Berkeley
- Michael Franklin Big Data Software: What's Next? (and what do we have to say about it?), 43rd VLDB Conference Munich August 2017
- Intro to Apache Spark http://cdn.liber118.com/workshop/itas_workshop.pdf
- Databricks – Advanced Spark
- Pietro Michiardi - Apache Spark Internals
- Madhukara Phatak. Anatomy of RDD
- Aaron Davidson. Building a unified data pipeline in Apache Spark
- MapR. Using Apache Spark DataFrames for Processing of Tabular Data
- Jules Damji. A Tale of Three Apache Spark APIs: RDDs vs DataFrames and Dataset
- Anton Kirillov. Apache Spark in depth: Core concepts, architecture&internals
- Patrick Wendell. Tuning and Debugging in Apache Spark

Related Papers

- Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. Originally OSDI 2004. CACM Volume 51 Issue 1, January 2008. http://dl.acm.org/citation.cfm?id=1327492
- HaLoop: Efficient Iterative Data Processing on Large Clusters by Yingyi Bu et al. In VLDB'10: The 36th International Conference on Very Large Data Bases, Singapore, 24-30 September, 2010.
- Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. Matei Zaharia et al. NSDI (2012) usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf
- MLbase: A Distributed Machine-learning System. Tim Kraska et al. CIDR 2013. http://www.cs.ucla.edu/~ameet/mlbase.pdf
- Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. 2015. Making sense of performance in data analytics frameworks. In Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI'15). USENIX Association, Berkeley, CA, USA, 293-307.
- Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley,Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, Matei Zaharia Spark SQL: Relational Data Processing in Spark