Introduction to Scalable Data Analytics using Apache Spark

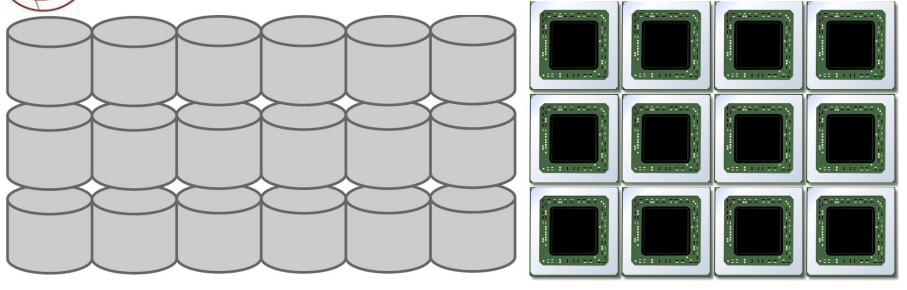
Vassilis Christophides
christop@csd.uoc.gr
http://www.csd.uoc.gr/~hy562
University of Crete, Fall 2019

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

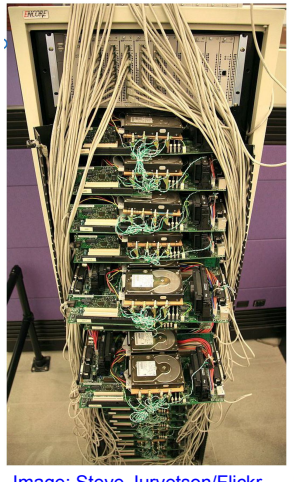
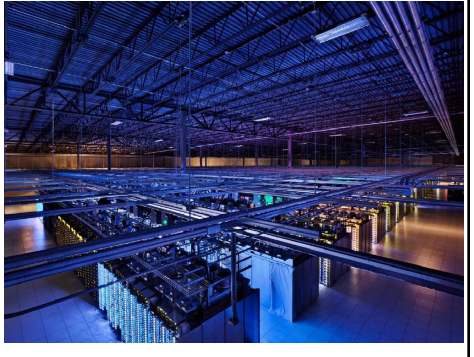


Image: Steve Jurvetson/Flickr

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



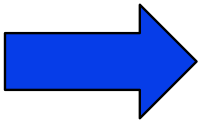
Google Datacenter

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?

How do you Count the Number of Occurrences of each Word in a Document?

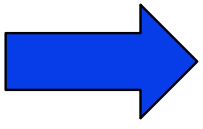
"I am Sam
I am Sam
Sam I am
Do you like
Green eggs and ham?"



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

Centrilized Approach: Use a Hash Table

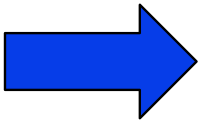
"I am Sam
I am Sam
Sam I am
Do you like
Green eggs and ham?"



{ }

Centrilized Approach: Use a Hash Table

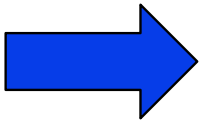
"I am Sam
I am Sam
Sam I am
Do you like
Green eggs and ham?"



{ I: **1**, }

Centrilized Approach: Use a Hash Table

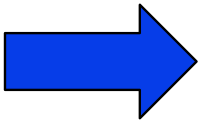
"I am Sam
I am Sam
Sam I am
Do you like
Green eggs and ham?"



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

Centrilized Approach: Use a Hash Table

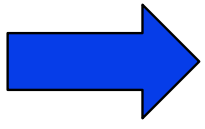
"I am Sam
I am Sam
Sam I am
Do you like
Green eggs and ham?"



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

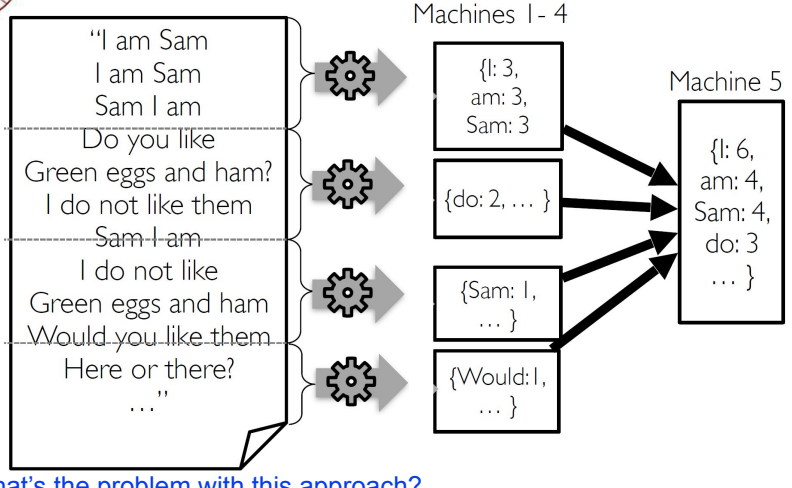
Centrilized Approach: Use a Hash Table

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



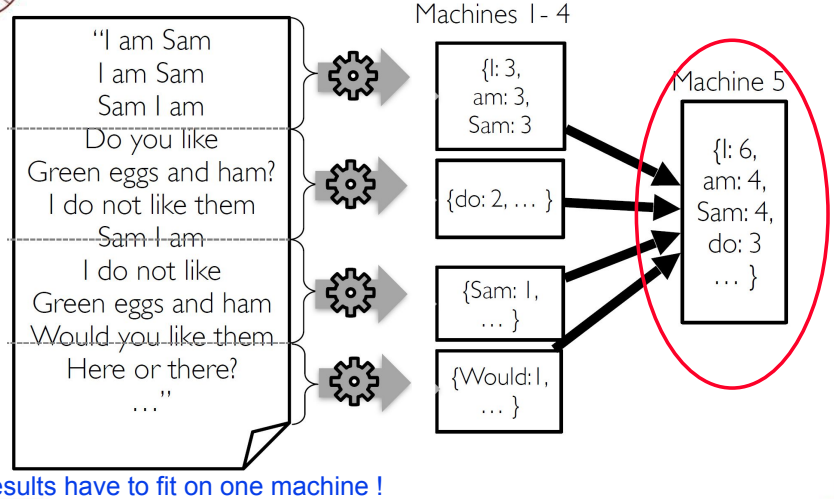
{ I: **2**,
am: **1**,
Sam: 1,
}

What if the Document is Really Big?



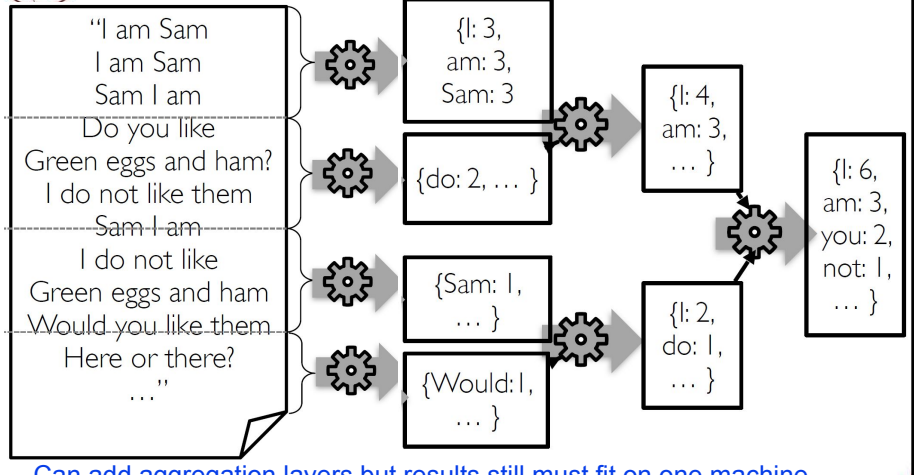
What's the problem with this approach?

What if the Document is Really Big?



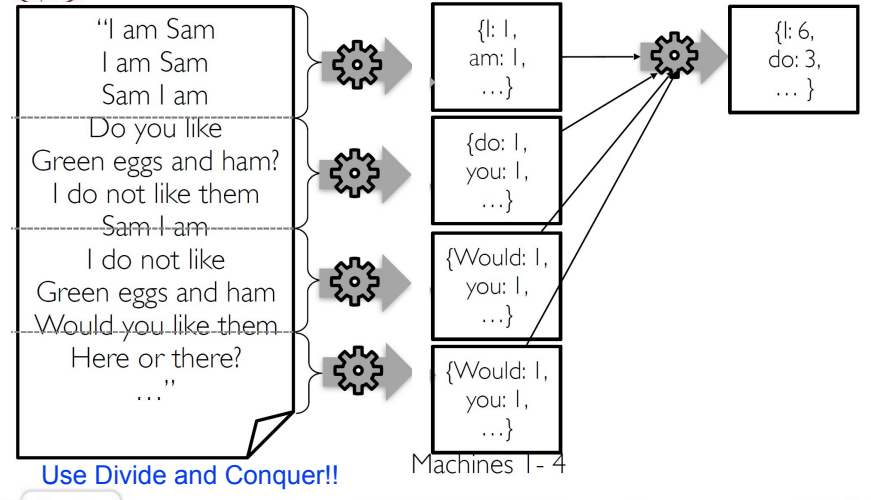
Results have to fit on one machine !

What if the Document is Really Big?

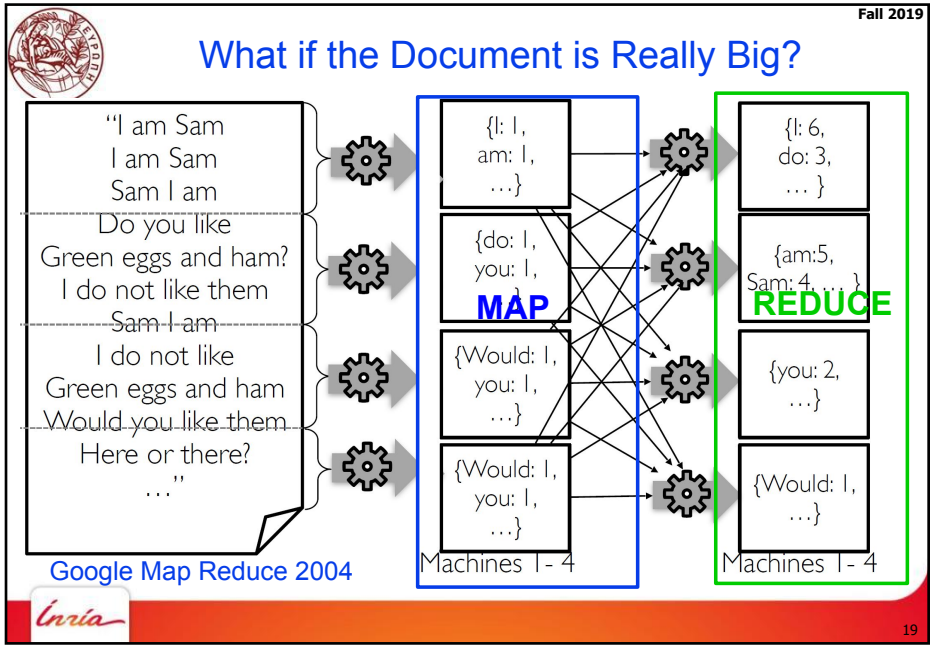


Can add aggregation layers but results still must fit on one machine

What if the Document is Really Big?



Use Divide and Conquer!!



What if the Document is Really Big?

"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?
…"

{I: 1, am: 1, …}
{do: 1, you: 1, …}
{Would: 1, you: 1, …}
{Would: 1, you: 1, …}

MAP

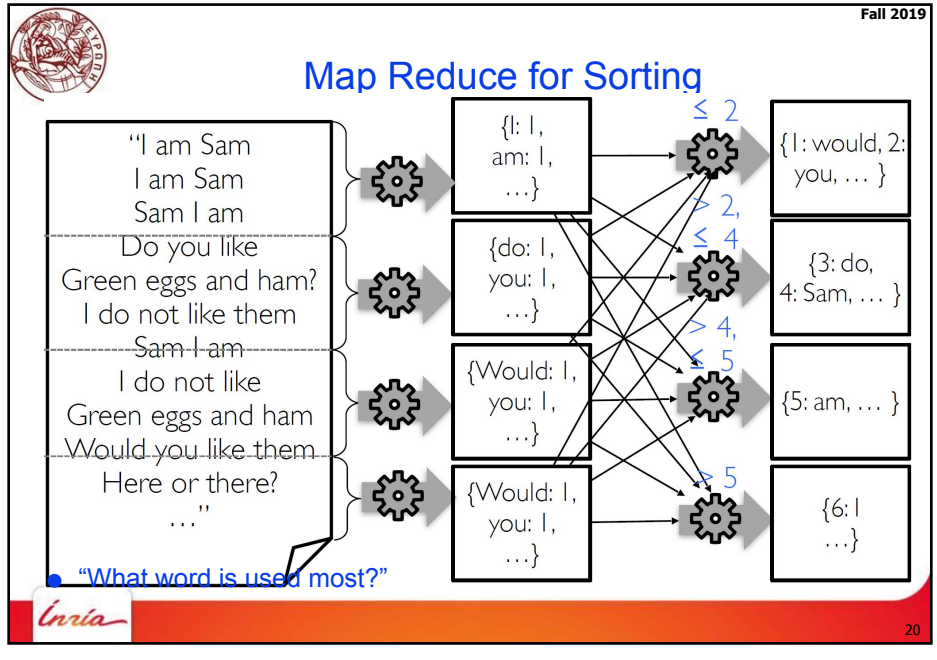
Machines 1- 4

{I: 6, do: 3, … }
{am:5, Sam: 4, … }
{you: 2, …}
{Would: 1, …}

REDUCE

Machines 1- 4

Google Map Reduce 2004



Map Reduce for Sorting

"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?
…"

{I: 1, am: 1, …}
{do: 1, you: 1, …}
{Would: 1, you: 1, …}
{Would: 1, you: 1, …}

≤ 2
> 2, ≤ 4
> 4, ≤ 5
> 5

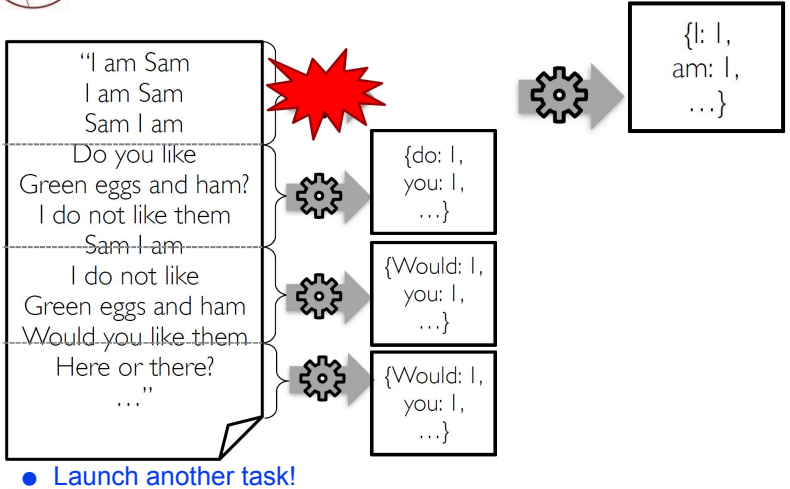
{1: would, 2: you, … }
{3: do, 4: Sam, … }
{5: am, … }
{6: I …}

- "What word is used most?"

What's Hard About Cluster Computing?

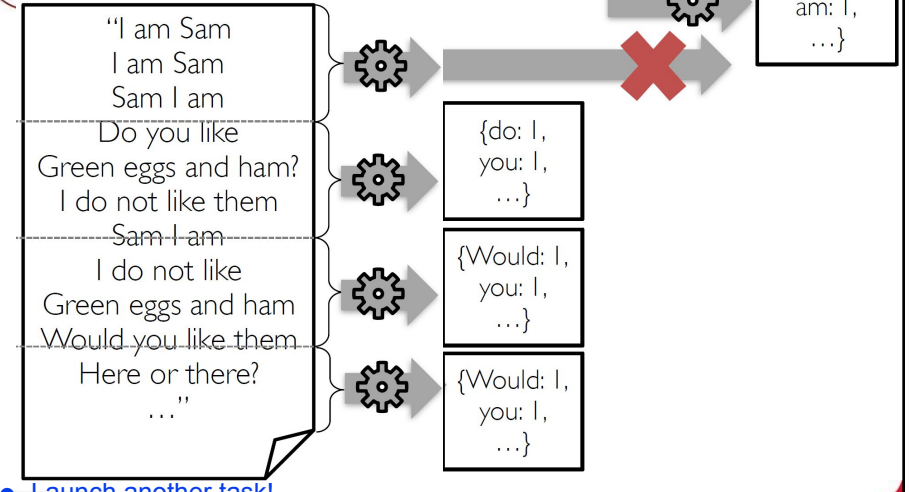
- How to divide work across machines?
  - Must consider network, data locality
  - Moving data may be very expensive
- How to deal with failures?
  - 1 server fails every 3 years => with 10,000 nodes see 10 faults/day
- Even worse: stragglers (not failed, but slow nodes)

How Do We Deal with Machine Failures?



- Launch another task!

How Do We Deal with Slow Tasks?



- Launch another task!

MapReduce: Distributed Execution

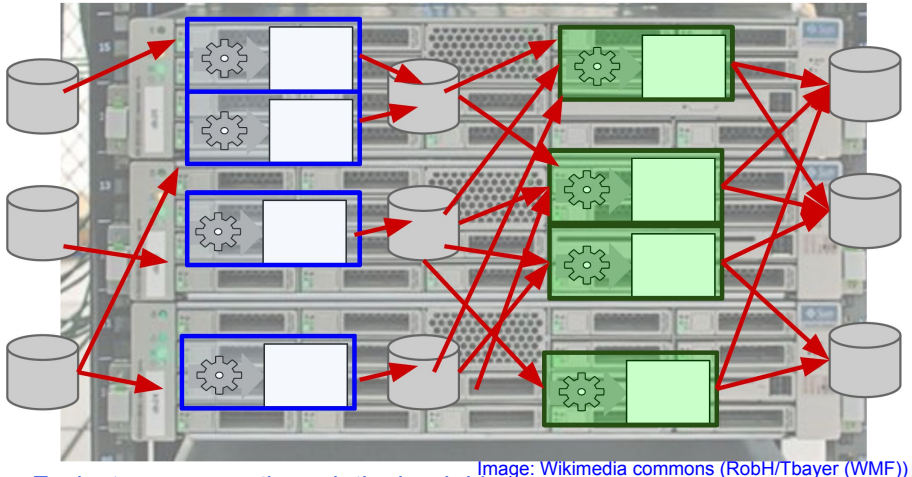
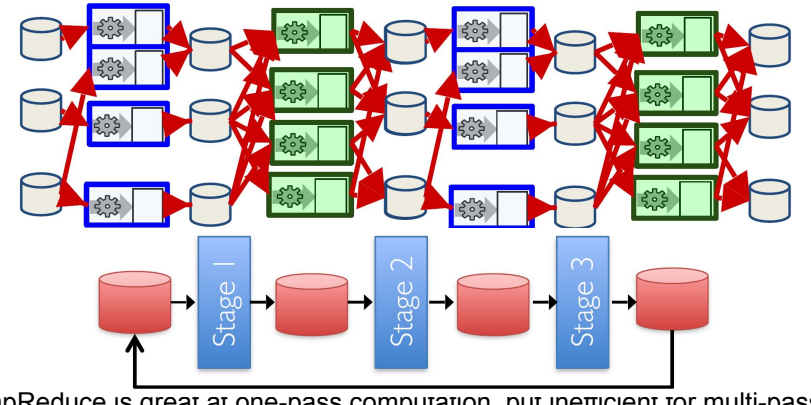


Image: Wikimedia commons (RobH/Tbayer (WMF))

- Each stage passes through the hard drives

Map Reduce: Iterative Jobs

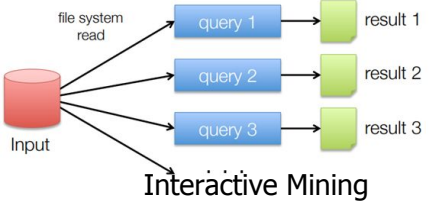
- Iterative jobs involve a lot of disk I/O for each repetition
  - Disk I/O is very slow!



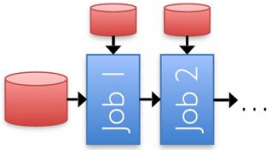
- MapReduce is great at one-pass computation, but inefficient for multi-pass algorithms

Apache Spark Motivation

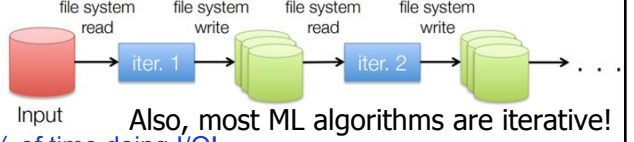
- While MapReduce is simple, it can require asymptotically los lots of disk I/O for complex jobs, interactive queries and online processing



Interactive Mining



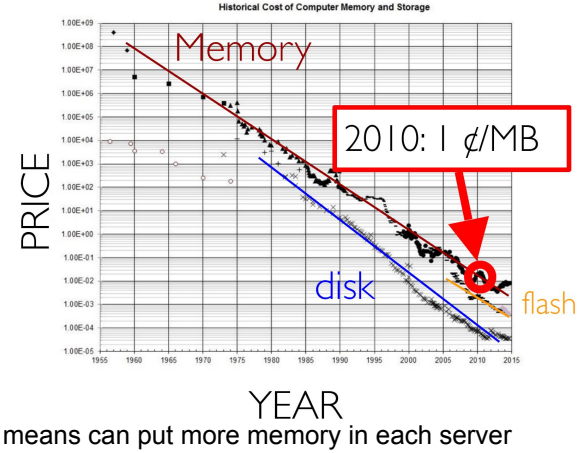
Stream Processing



Also, most ML algorithms are iterative!

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

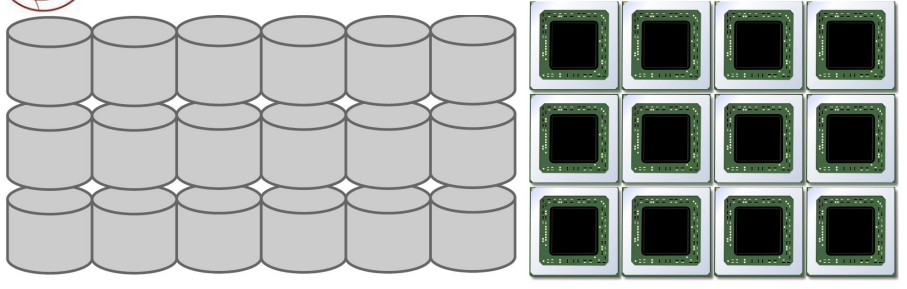
Tech Trend: Cost of Memory



- 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

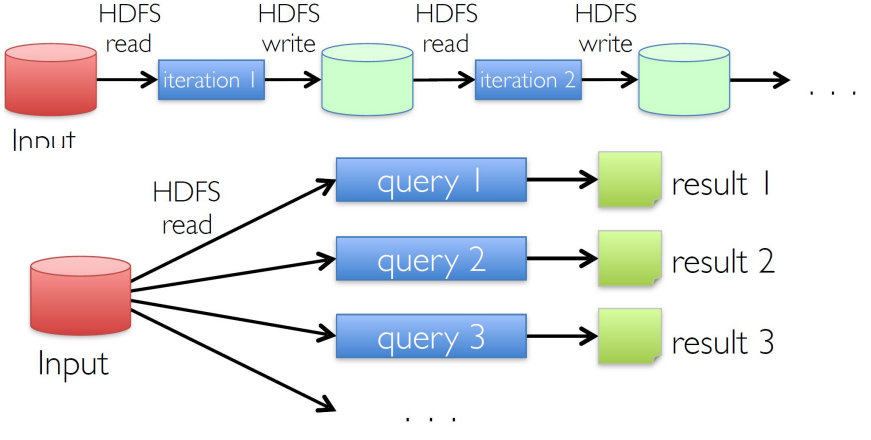
… and memory!

Opportunity

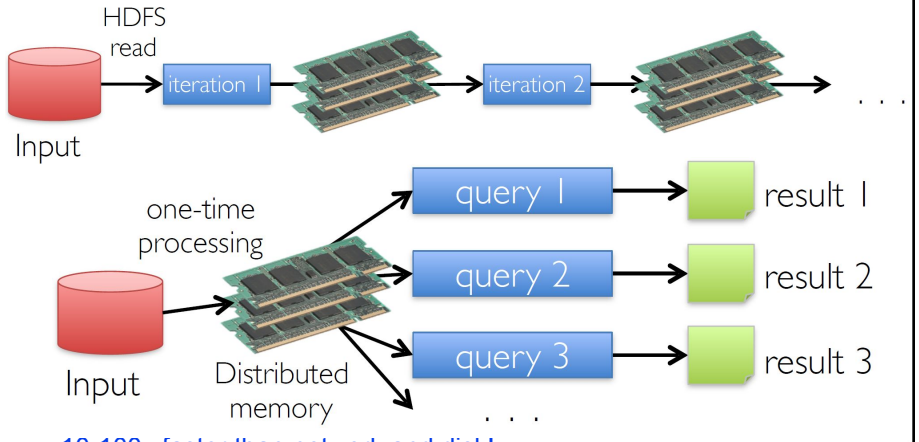
- Keep more data in-memory
- Create new distributed execution engine : Apache Spark™
- One of the most efficient programming frameworks offering abstraction and parallelism for clusters
- It hides complexities of:
  - Fault Tolerance
  - Slow machines
  - Network Failures

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

Use Memory Instead of Disk



In-Memory Data Sharing



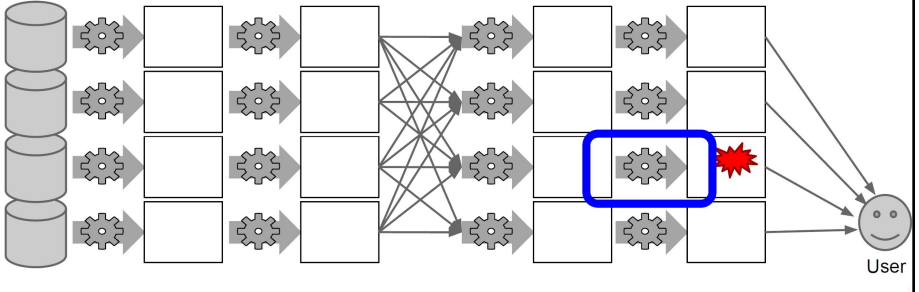
- 10-100x faster than network and disk!

Spark and Map Reduce Differences

	Apache Hadoop Map Reduce	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

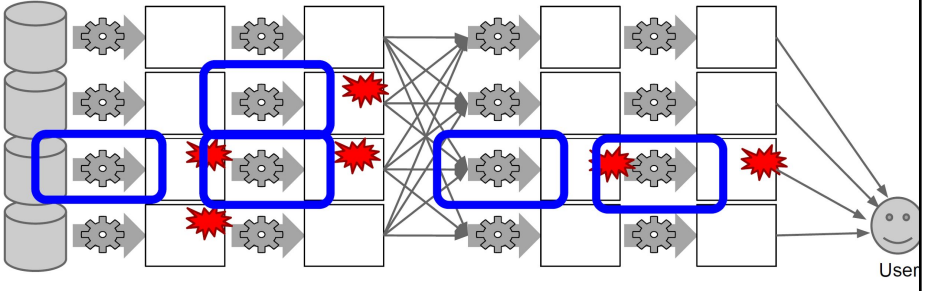
Spark: Fault Tolerance

- 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

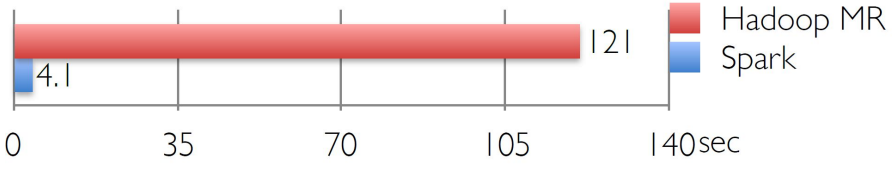


Other Spark and Map Reduce Differences

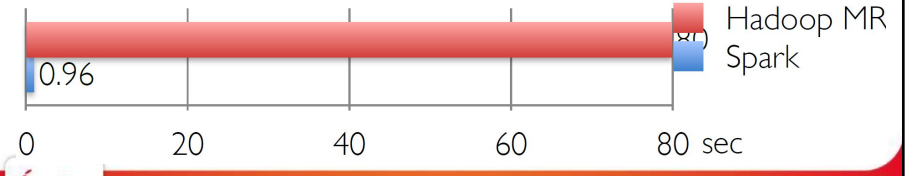
- Generalized patterns for computation
  - provide unified engine for many use cases
  - require 2-5x less code
- Lazy evaluation of the lineage graph
  - can optimize, reduce wait states, pipeline better
- Lower overhead for starting jobs
- Less expensive shuffles

In-Memory Can Make a Big Difference

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

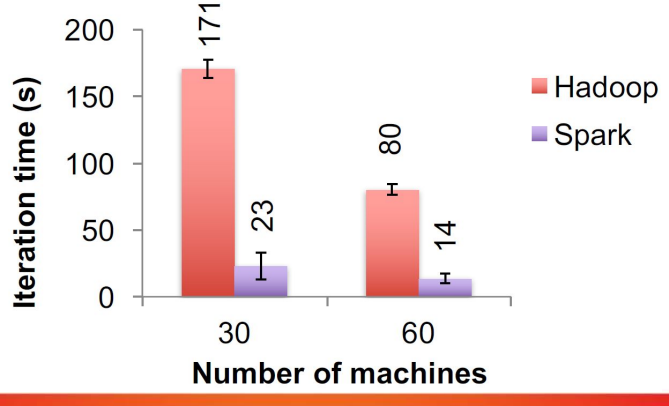


  - Logistic Regression



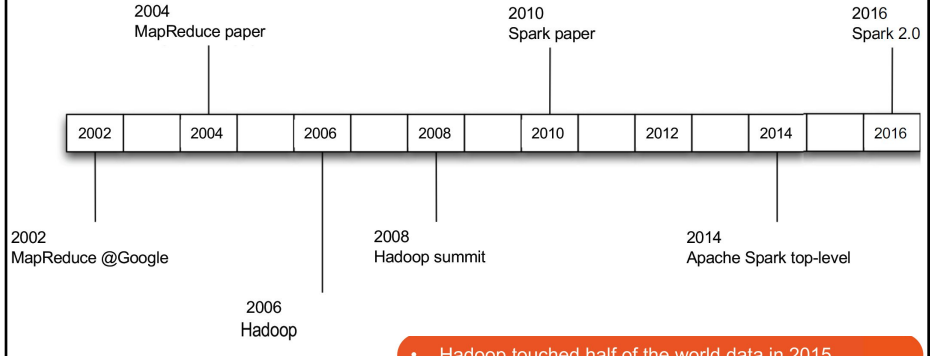
In-Memory Can Make a Big Difference

- PageRank



What is Apache Spark?

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

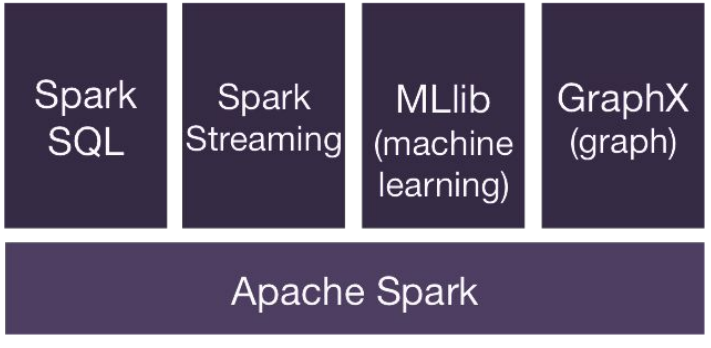
Recall What's Hard with Big Data

- 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
  - Provides high-level APIs with space to optimize
    - RDDs, Data Frames, Data Pipelines
  - and tools for many different languages
    - Fluent Scala/Java/Python/R API
    - Interactive shell (repl)
    - 2-10x less code (than Hadoop 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

Apache Spark Software Stack: Unified Vision



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

Vs Apache Hadoop

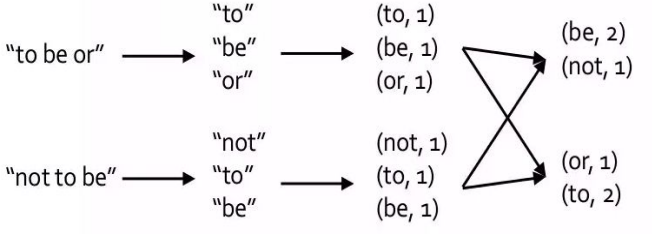
General Batching	Specialized systems			
	Streaming	Iterative	Ad-hoc / SQL	Graph
MapReduce	Storm	Mahout	Pig	Giraph
	S4		Hive	
	Samza		Drill	
			Impala	

- Sparse Modules
- Diversity of APIs
- Higher Operational Costs

Spark Speaks your Language

Spark: Descriptive Computing Model

- Organize computation into multiple stages in processing pipeline
  - Transformations apply user code to distributed data in parallel
  - Actions assemble final output of an algorithm from distributed data



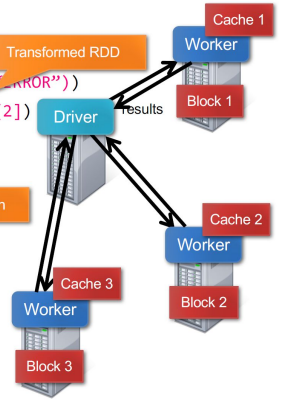
Resilient Distributed Datasets (RDDs)

- 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)

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()
. . .
```



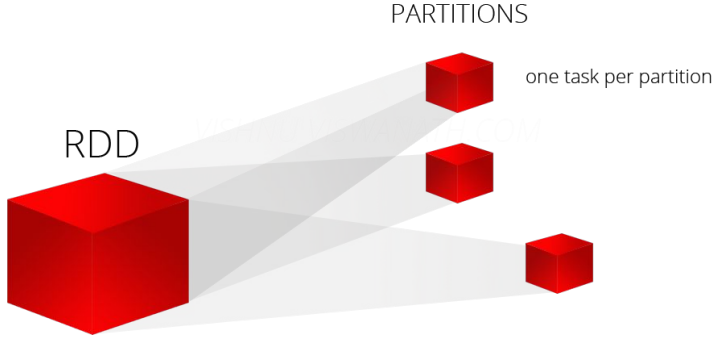
Result: full-text search of Wikipedia in < 5 sec (vs 20 sec for on-disk data)

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

RDD: Partitions

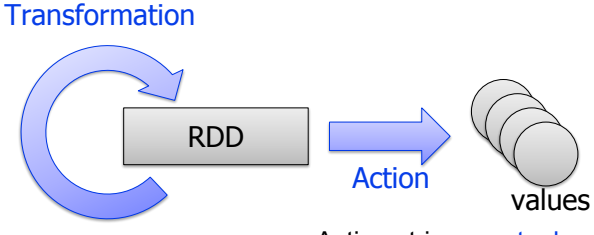
- RDDs are automatically distributed across the network by means of partitions
  - A partition is a logical division of data
  - RDD data is just a collection of partitions
  - Spark automatically decides the number of partitions when creating an RDD
    - All input, intermediate and output data will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

RDD: Partitions



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

Two Types of Operations on RDDs



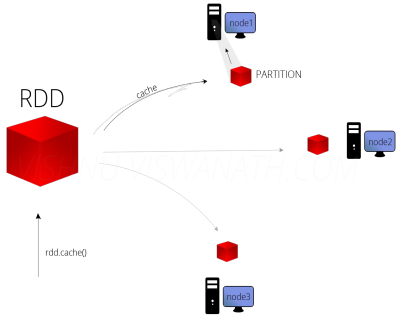
Transformations are lazy:
Framework keeps track of lineage

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/

RDD operations - Transformations

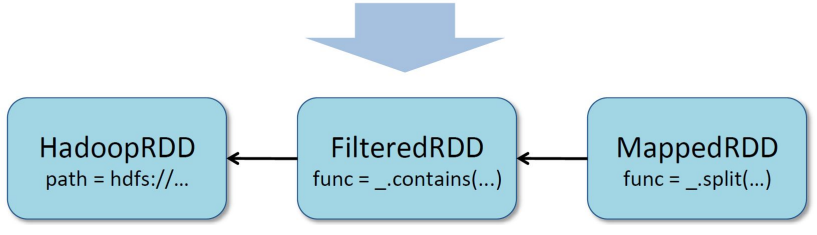
- As in relational algebra, the application of a transformation to an RDD yields a new RDD (immutability)
- Transformations are lazily evaluated which allow for optimizations to take place before execution
  - The lineage keeps track of all transformations that have to be applied when an action happens

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

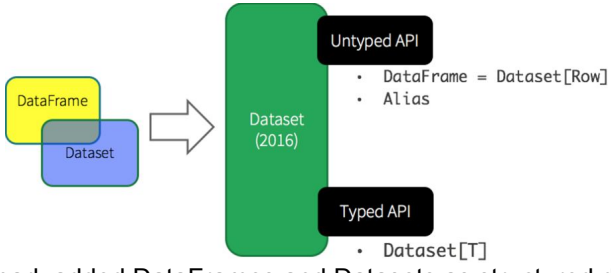
RDD Lineage (aka Logical Logging)

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

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



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]
- Both run on Tungsten
- Spark 2.0 merged these APIs
- Operators take expression in a special DSL that Spark can optimize

Static-Typing and Runtime Type-safety in Spark

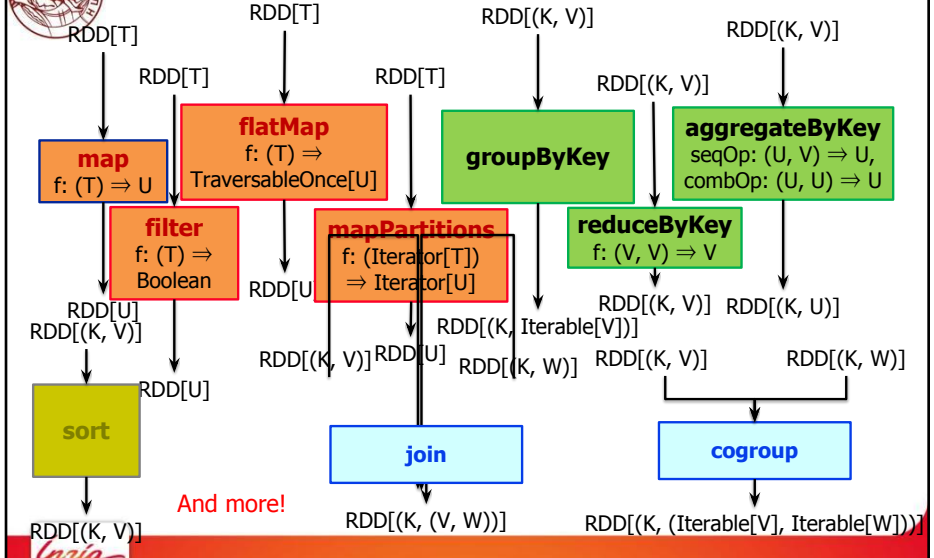
	SQL	DataFrames	Datasets
Syntax Errors	Runtime	Compile Time	Compile Time
Analysis Errors	Runtime	Runtime	Compile Time

- 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
```

Useful Transformations on RDDs



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	RDD	Other	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)}`

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

- `reduce()`: Takes a function that operates on two elements of the type in your RDD and returns a new element of the same type. The function is applied on all elements.
- `collect()`: returns the entire RDD's contents (commonly used in unit tests where the entire contents of the RDD are expected to fit in memory). The restriction here is that all of your data must fit on a single machine, as it all needs to be copied to the driver.
- `take()`: returns n elements from the RDD and tries to minimize the number of partitions it accesses. No expected order
- `count()`: returns the number of elements

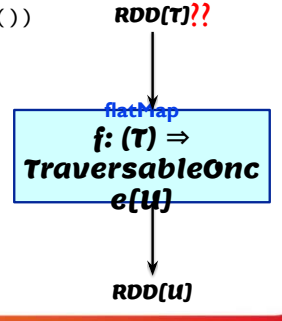
RDD Actions: Examples

	RDD	Result
`rdd.reduce((x, y) => x + y)`	{1,2,3}	6

Example	RDD	Result
`rdd.collect()`	{1,2,3}	{1,2,3}
`rdd.take(2)`	{1,2,3,4}	{1,3}
`rdd.count()`	{1,2,3,3}	4

Spark Word Count

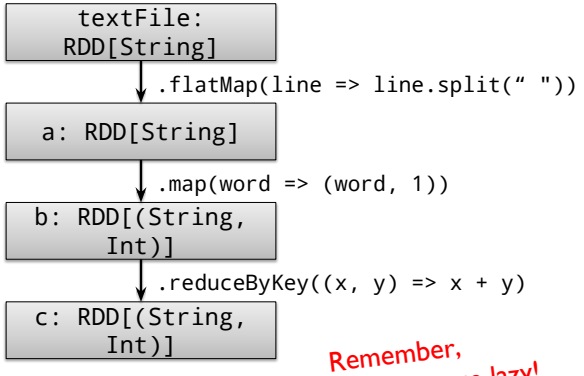
```
val textFile = sc.textFile(args.input())

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



Spark Word Count

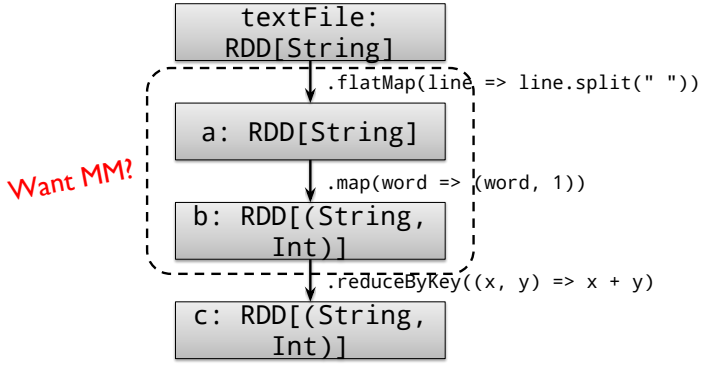
```
val textFile = sc.textFile(args.input())

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())
```


RDDs and Lineage



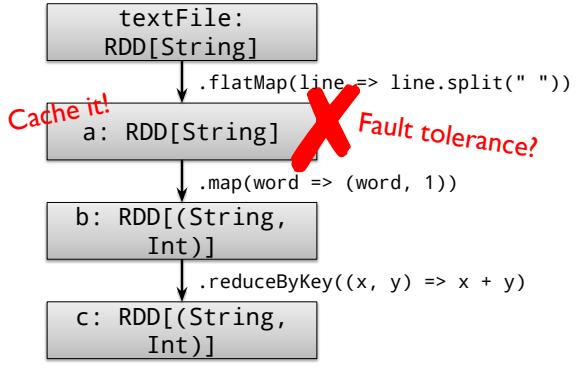
Remember, transformations are lazy!

RDDs and Optimizations



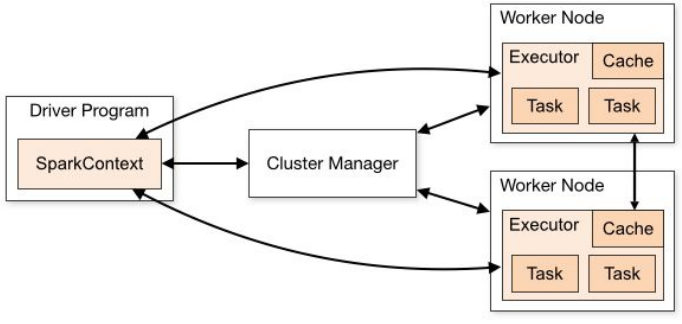
RDDs and Caching

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

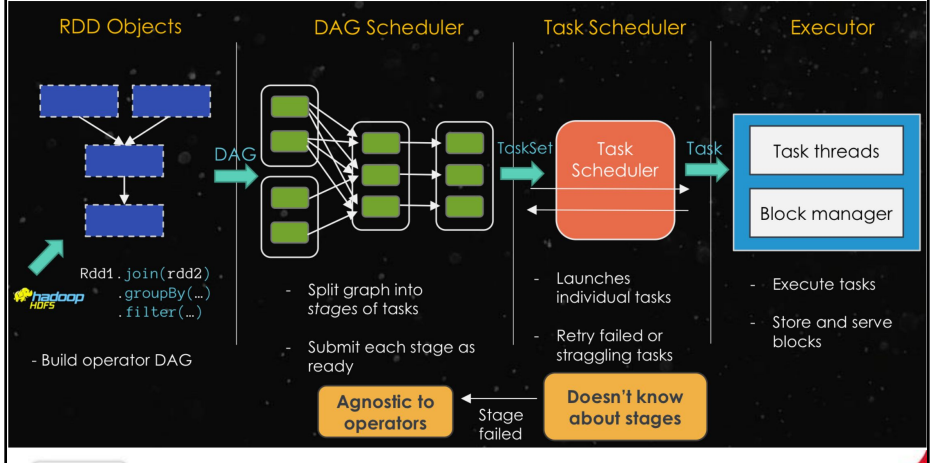


Spark works even if the RDDs are *partially* cached!

Spark Architecture



Scheduling Process

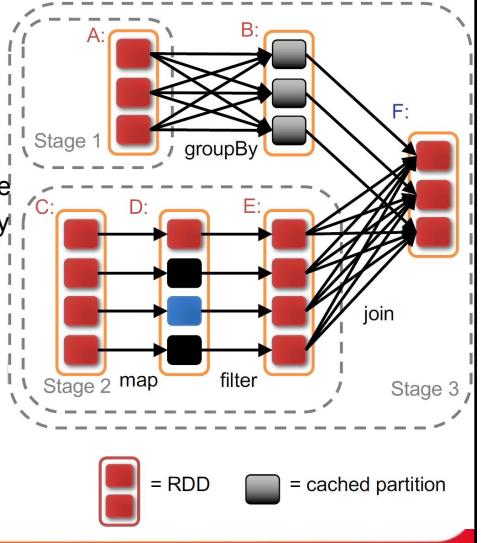


Scheduling Problems

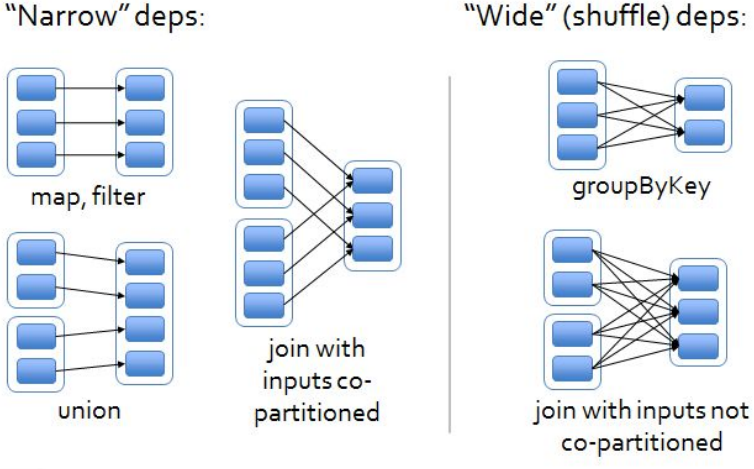
- Supports general task graphs
- Pipelines functions where is possible
- Cache-aware data reuse and locality
- Partitioning-aware to avoid shuffles

Potential bottleneck?

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

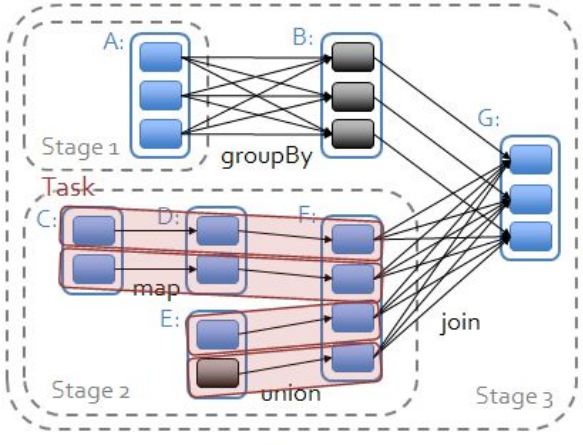


Narrow vs Wide Dependencies



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

Narrow vs Wide Dependencies



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

Where "Database Thinking" Can Get In The Way

Traditional Database Thinking

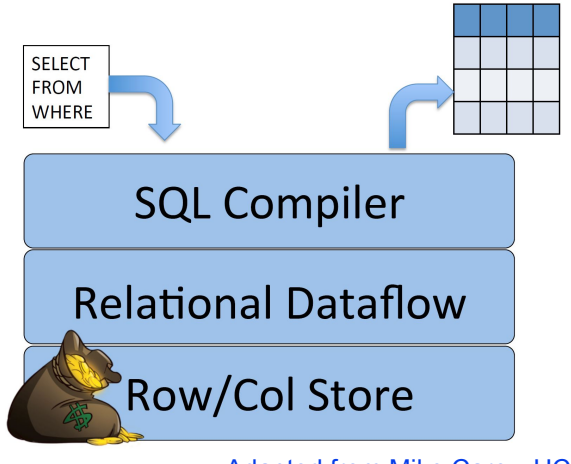
Pros

- 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

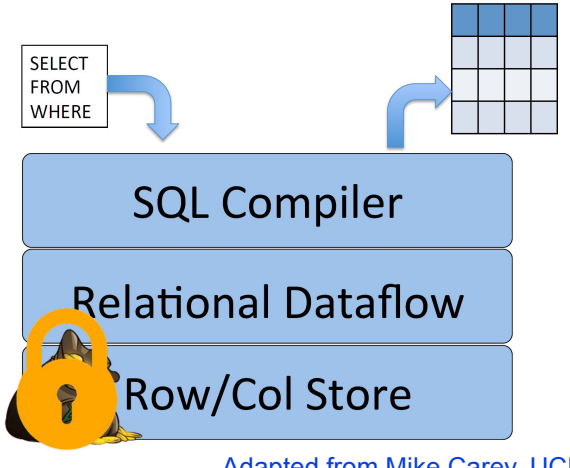
- Monolithic Systems and Control
- Schema First & High Friction
- The DB Lament: "We've seen it all before"

Database Systems: One Way In/Out



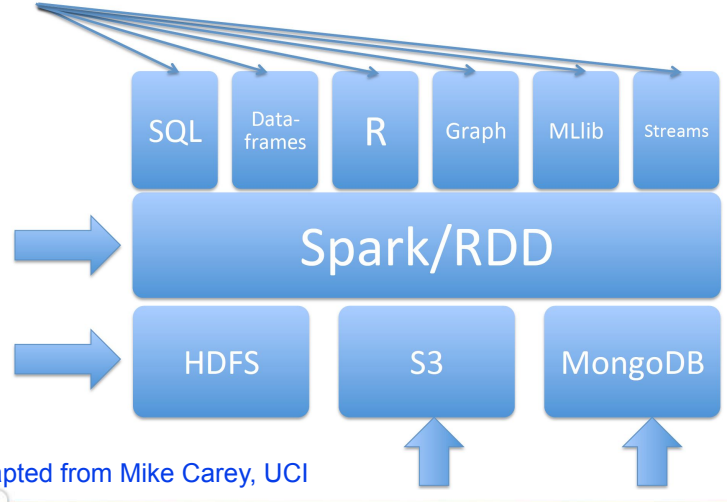
Adapted from Mike Carey, UCI

Database Systems: One Way In/Out

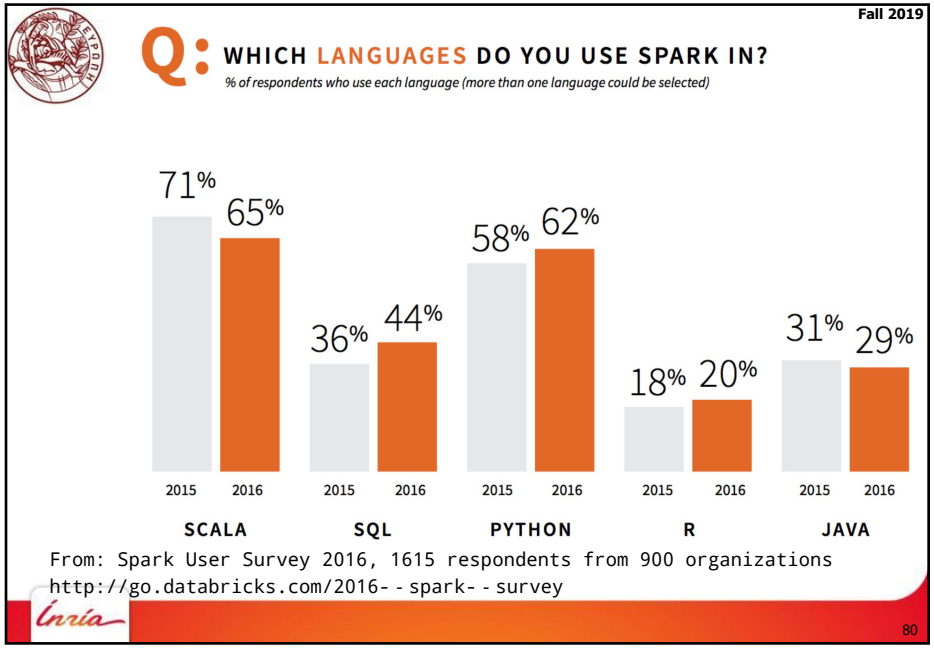


Adapted from Mike Carey, UCI

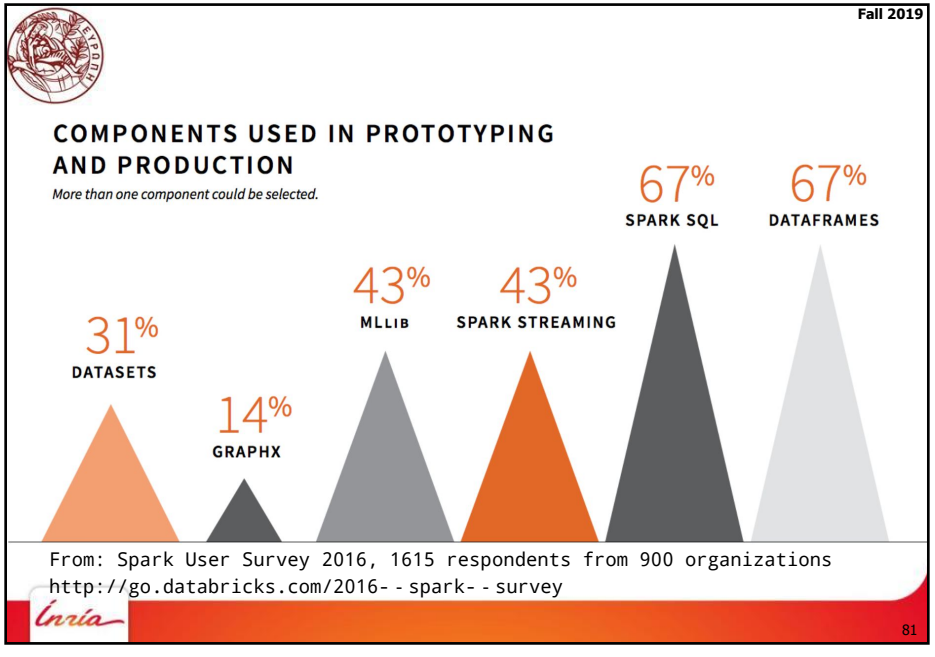
Mix and Match Data Access



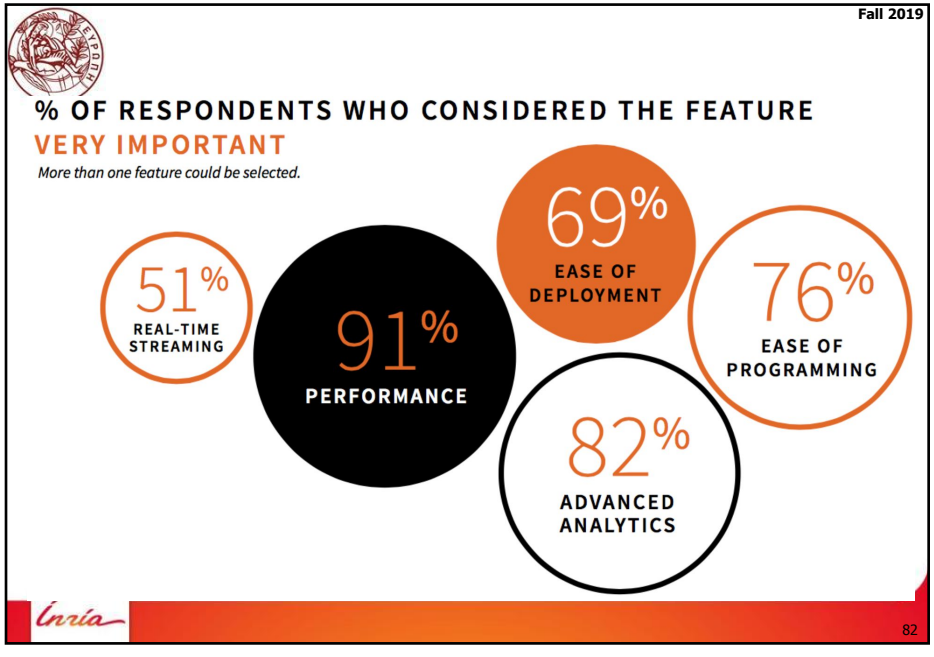
Adapted from Mike Carey, UCI



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



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

A Data Management Inflection Point

Scale Out Computing	• Processing • Storage
Elastic Resources	• Pay-as-you-go Processing • Pay-as-you-go Storage
Flexible Data Formats	• Schema on Read vs. on Write • Direct access to stored data
Multimodal Advanced Analytics	• Search, Query, Analytics • Machine Learning, AI
Open Source Ecosystem	• Rapid Adoption • Rapid Innovation

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

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