Introduction to Apache Spark

This Lecture

Big Data Problems: Distributing Work, Failures, Slow Machines

HW/SW for Big Data: Map Reduce and Apache Spark

The Structured Query Language (SQL)

SparkSQL

Apache Spark Resources and Community

Apache Web Server Log Files

What is Apache Spark?

Scalable, *efficient analysis* of *Big Data*

What is Apache Spark?

Scalable, *efficient* *analysis* of *Big Data*

This lecture

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

Google Datacenter

How do we program this thing?

The Opportunity

Cloud computing is a game-changer!

Provides access to low-cost computing and storage

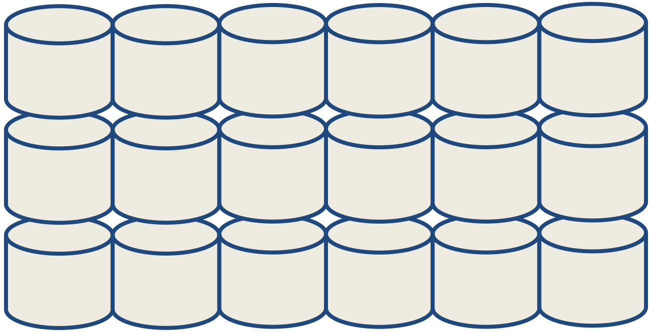
Costs decreasing every year

The challenge is programming the resources

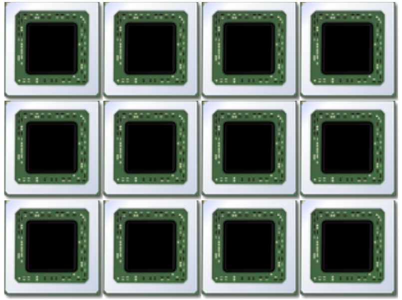
What is Apache Spark?

- *Scalable*, *efficient analysis* of *Big Data*

A Brief History of Big Data Processing



Lots of hard drives



… and CPUs

Yesterday's Hardware for Big Data

One big box!
(1990's solution)

- » All processors share memory

Very expensive

- » Low volume
- » All "premium" hardware

And, still not big enough!

Hardware for Big Data

Image: Steve Jurvetson/Flickr

Consumer-grade hardware

- Not "gold plated"

Many desktop-like servers

- **Easy to add capacity**
- **Cheaper** per CPU/disk

But, requires complexity in software

Problems with Cheap Hardware

Facebook Datacenter (2014)

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

What's Hard About Cluster Computing?

How do we split work across machines?

Let's look at a simple task: word counting

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
…

[redacted]

One Approach: Use a Hash Table

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

{}

[redacted]

One Approach: Use a Hash Table

$\boxed{\text{“I}}$ am Sam
I am Sam
Sam I am
Do you like
Green eggs and ham?”

{I :1}

One 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**}

[redacted]

One 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**}

One Approach: Use a Hash Table

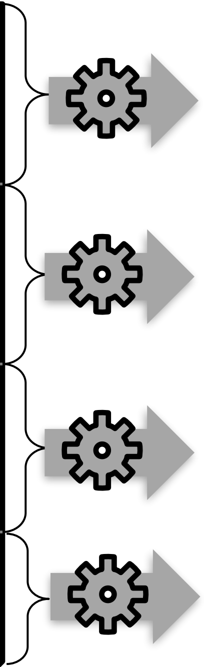
"I am Sam
I am Sam
Sam I am
Do you like
Green eggs and ham?"

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

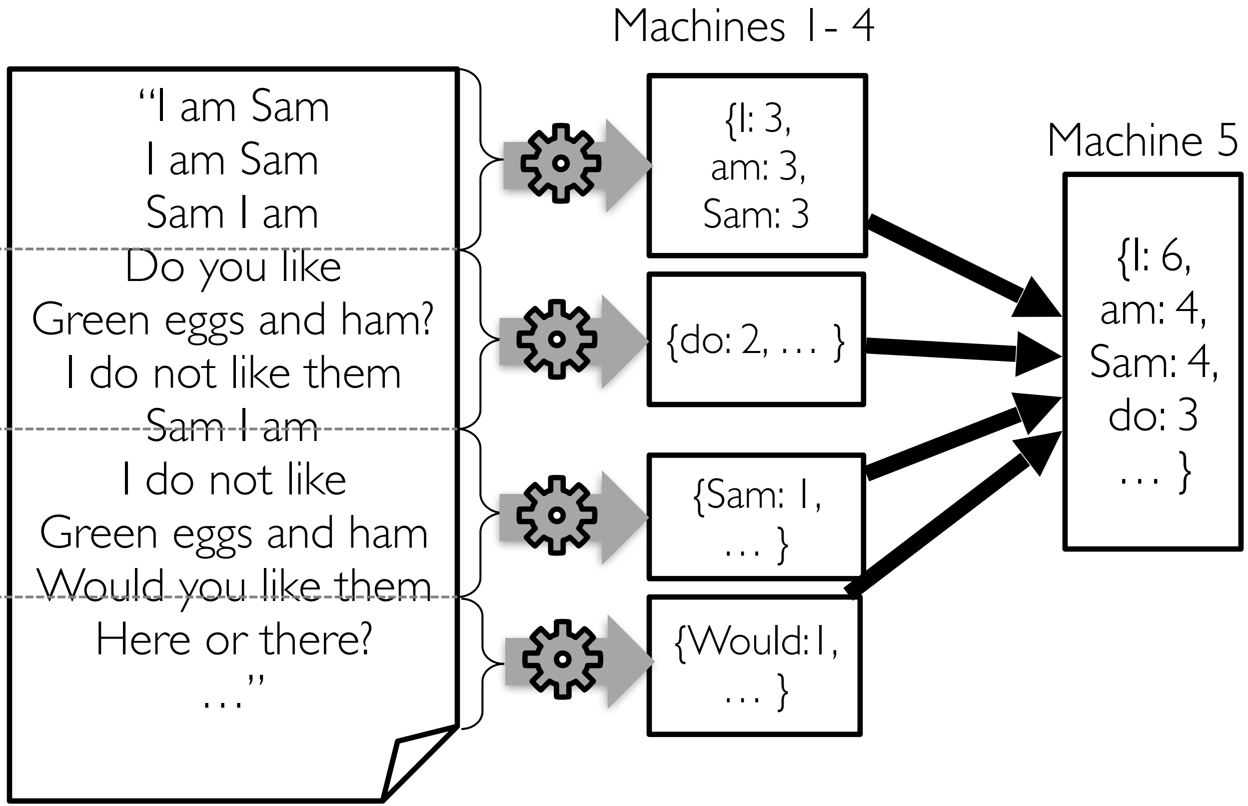
[redacted]

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?
…"



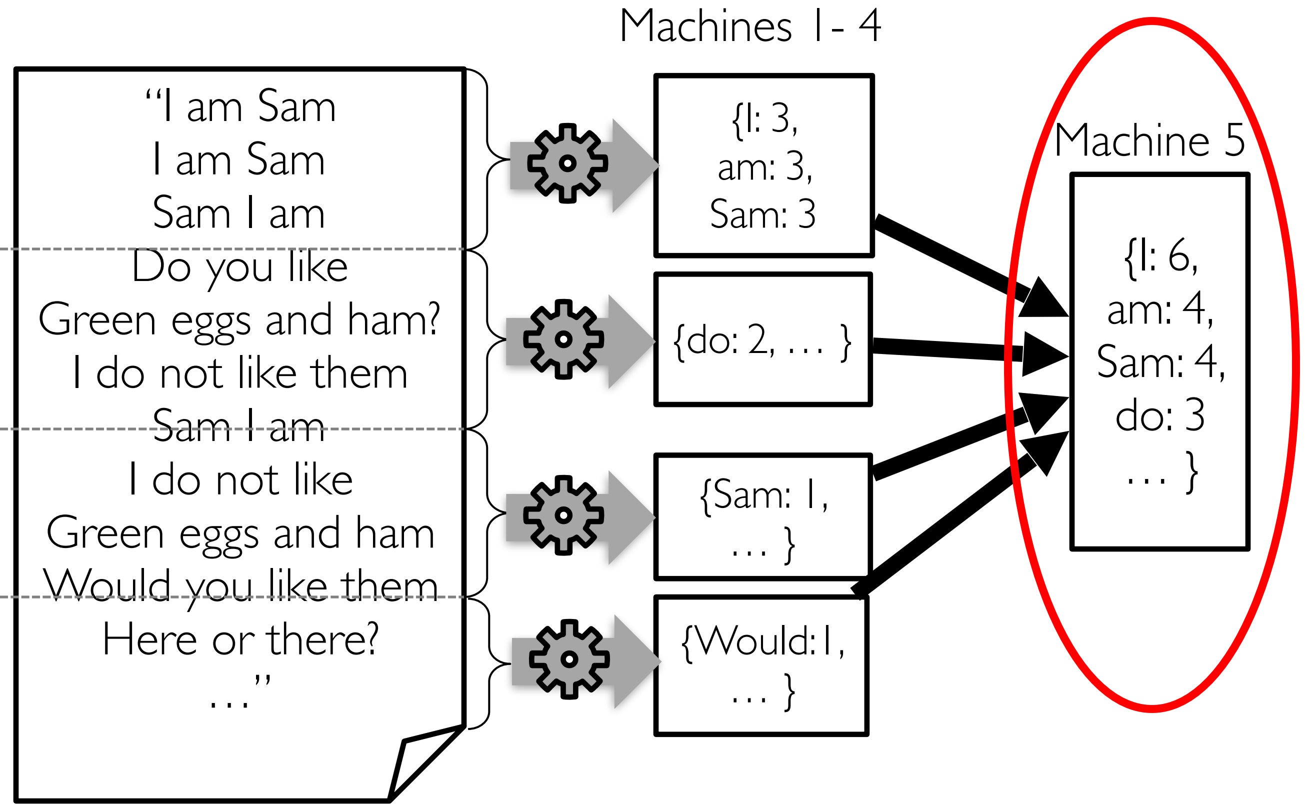
What if the Document is Really Big?



What's the problem with this approach?

[redacted]

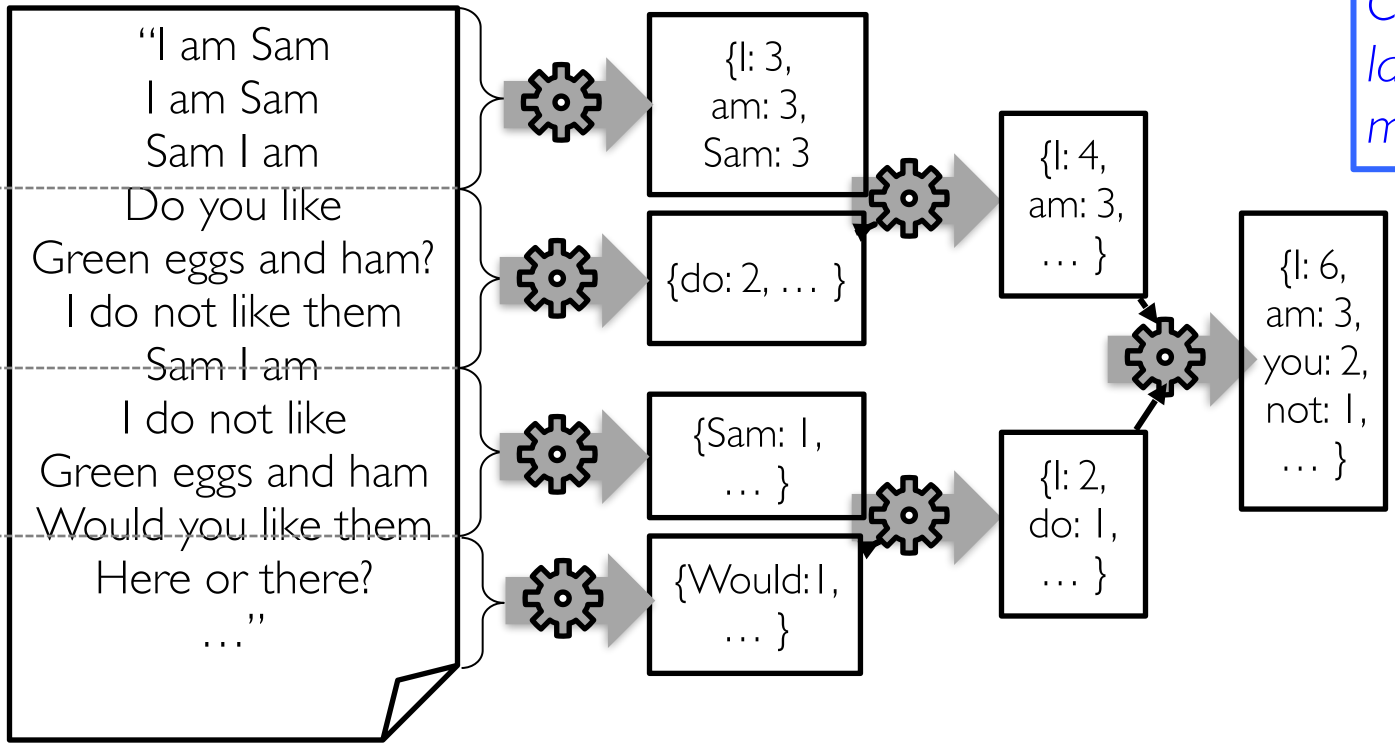
What if the Document is Really Big?



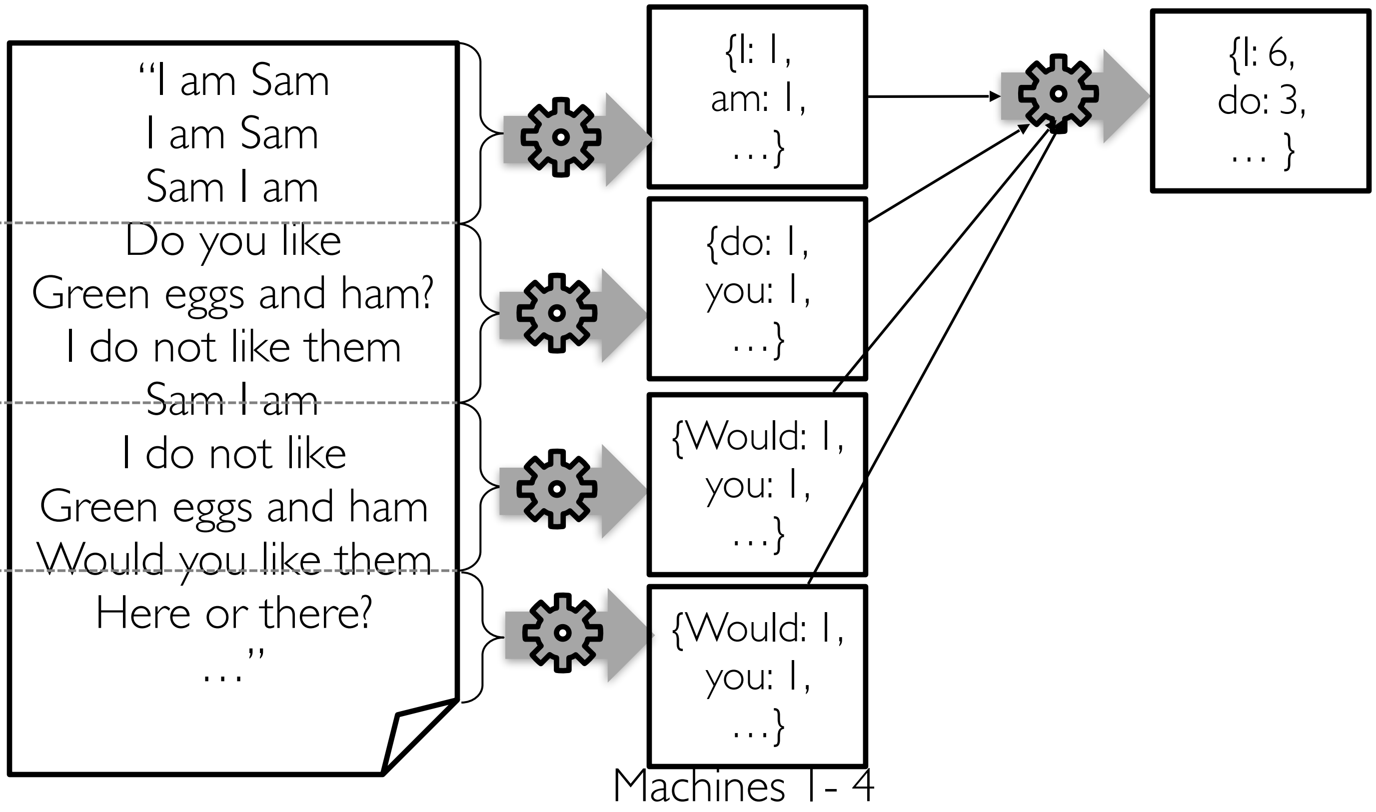
Results have to fit on one machine

What if the Document is Really Big?

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

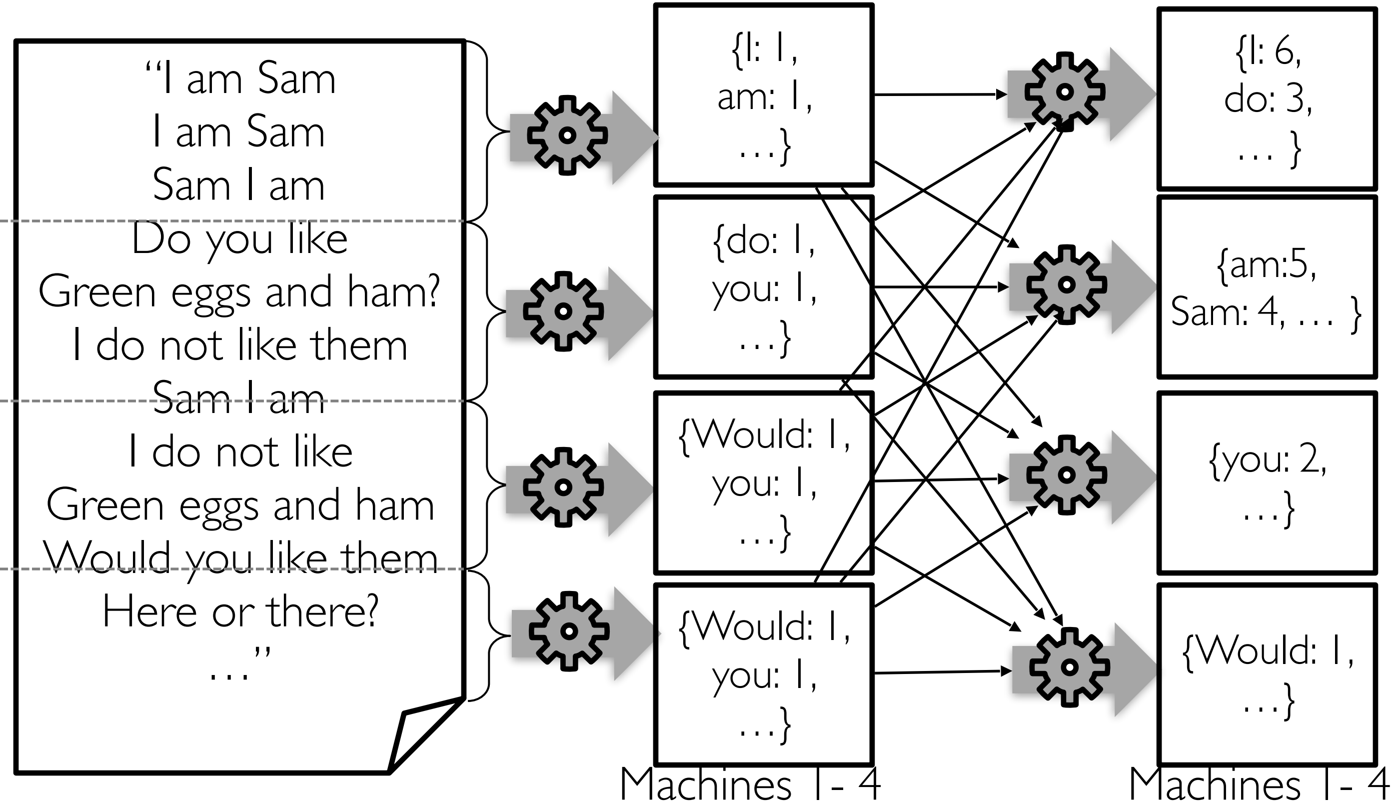


What if the Document is Really Big?



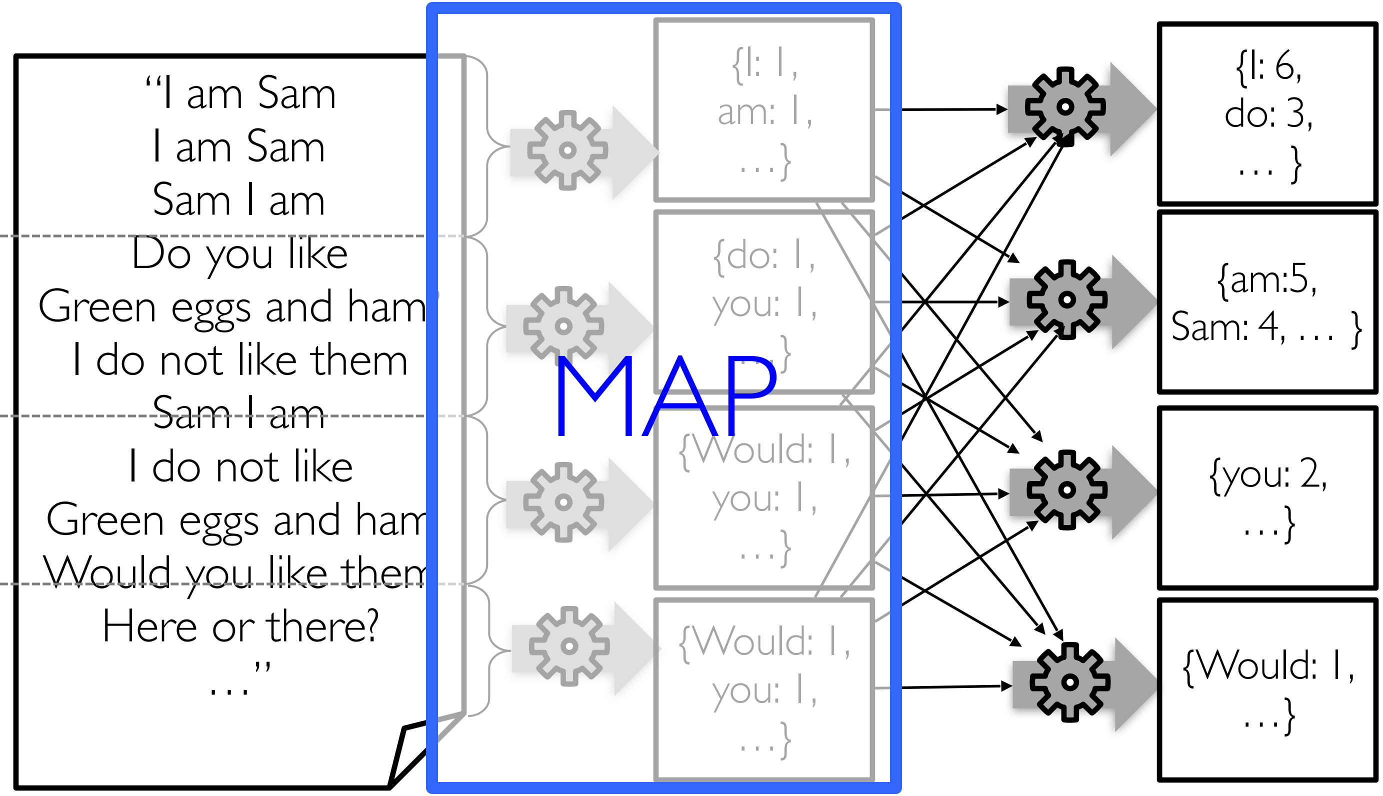
Use Divide and Conquer!!

What if the Document is Really Big?



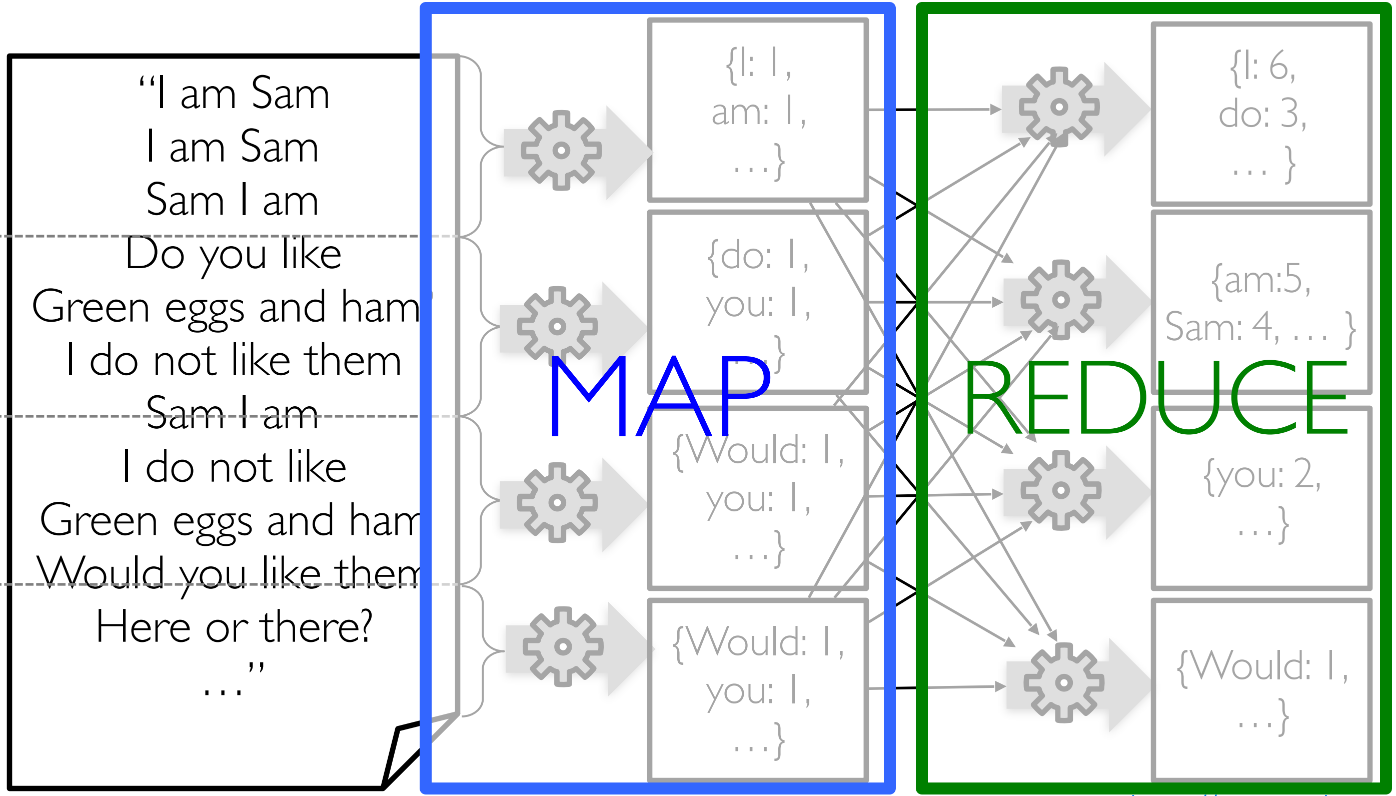
Use Divide and Conquer!!

What if the Document is Really Big?



Use Divide and Conquer!!

What if the Document is Really Big?

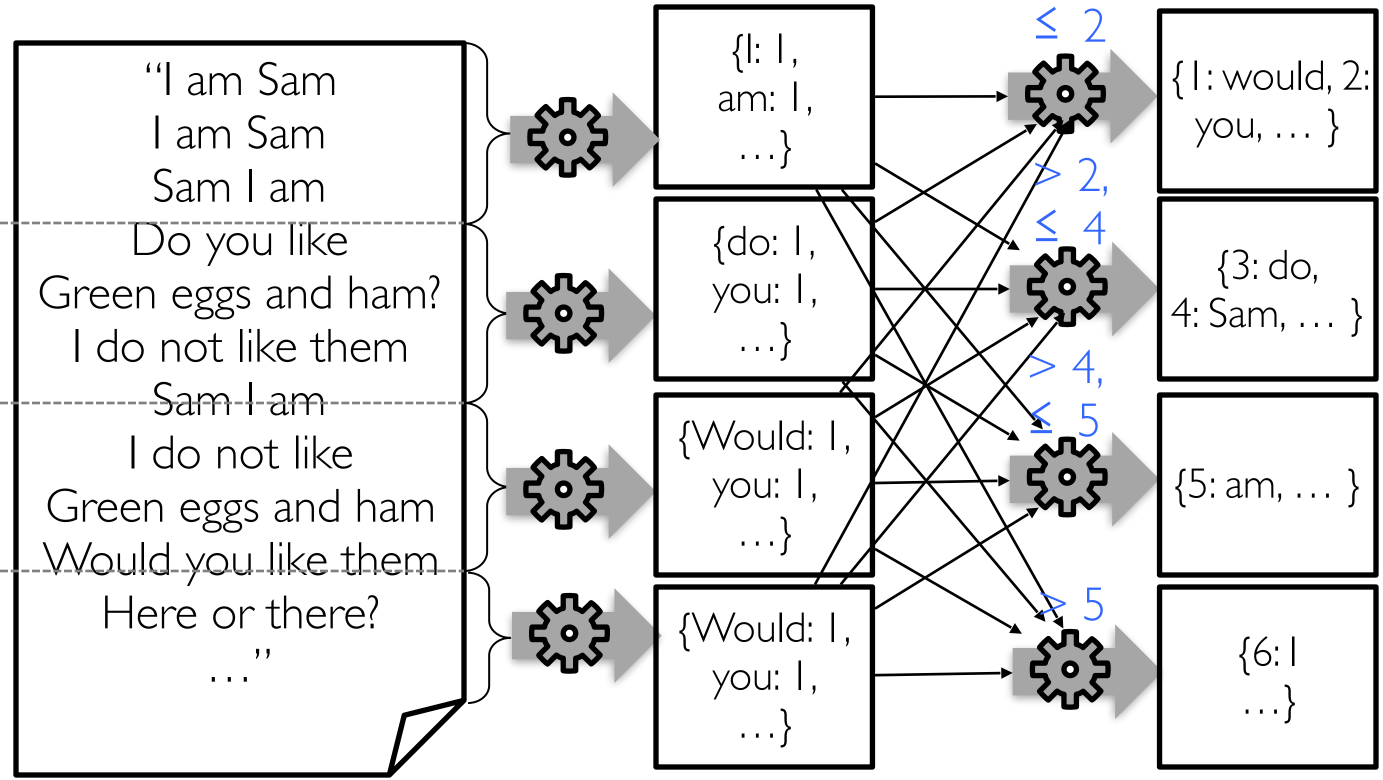


Google
Map Reduce 2004

Apache Hadoop

http://research.google.com/archive/mapreduce.html

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

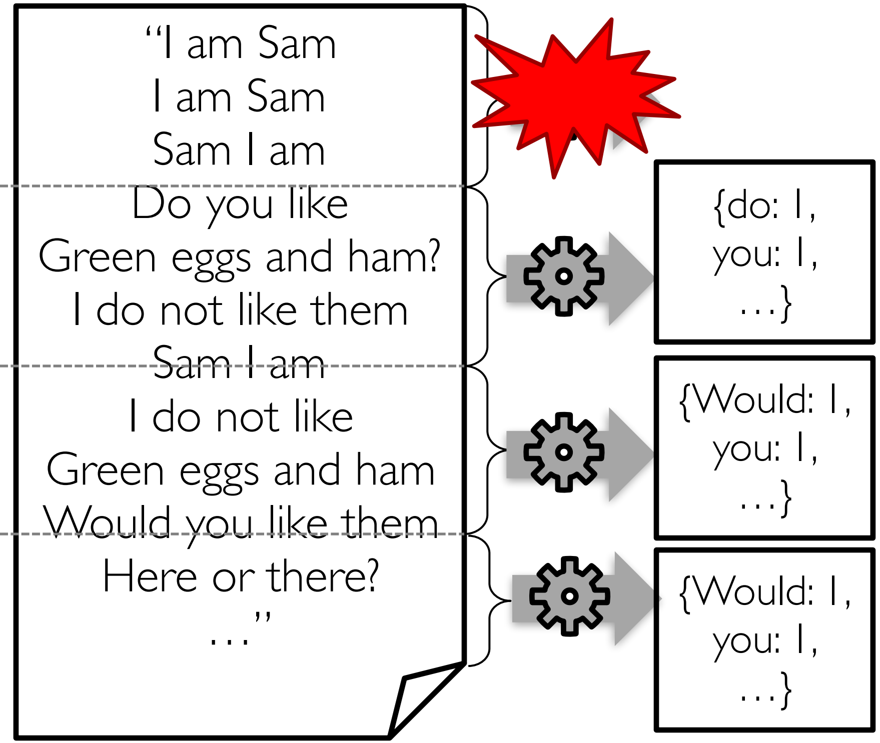
» Moving data may be very expensive

How to deal with failures?

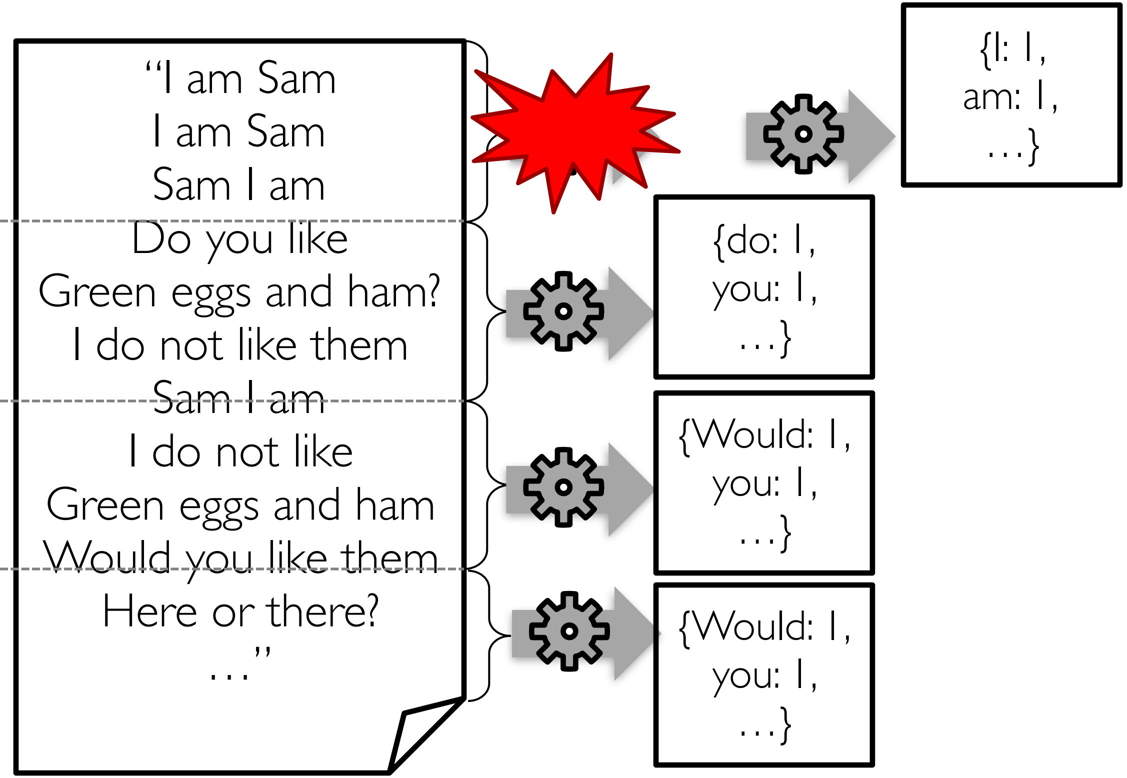
» 1 server fails every 3 years ⇒ with 10,000 nodes see 10 faults/day

» Even worse: stragglers (not failed, but slow nodes)

How Do We Deal with Failures?

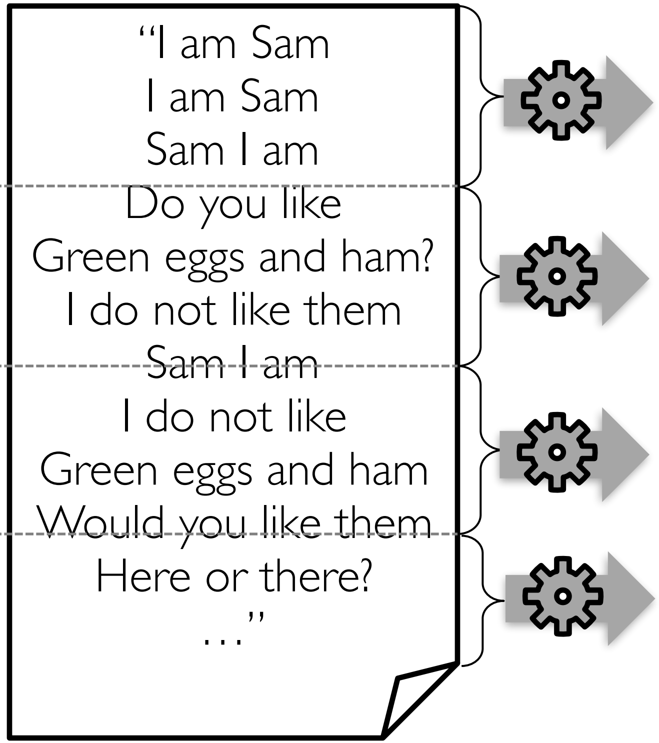


How Do We Deal with Machine Failures?



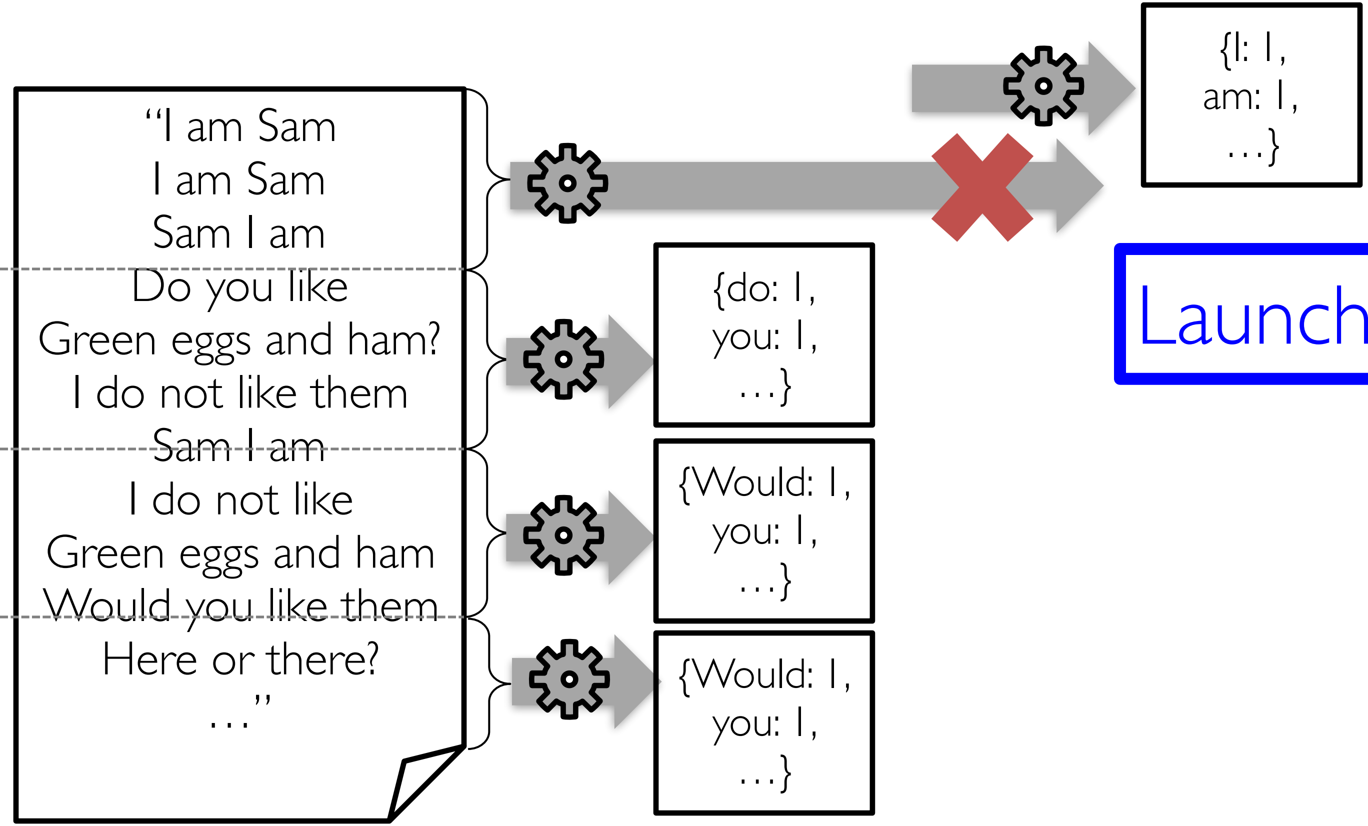
Launch another task!

How Do We Deal with Slow Tasks?



[redacted]

How Do We Deal with Slow Tasks?

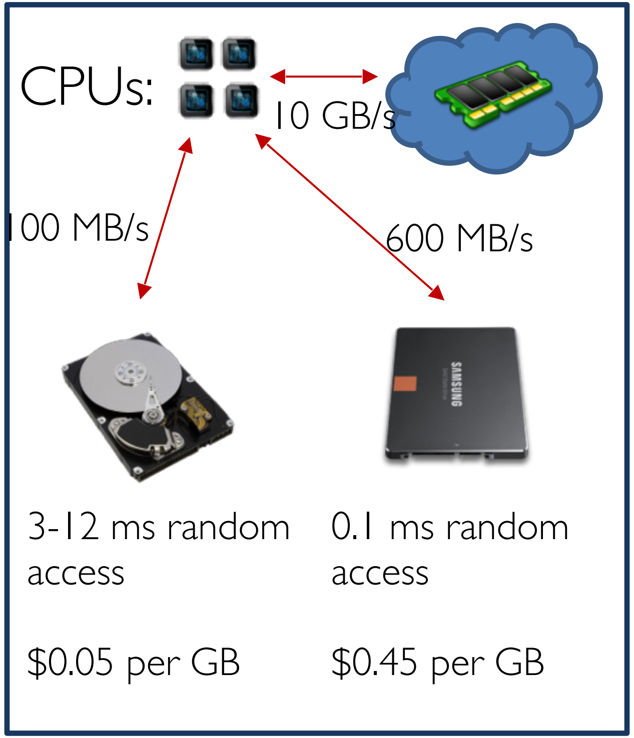


Launch another task!

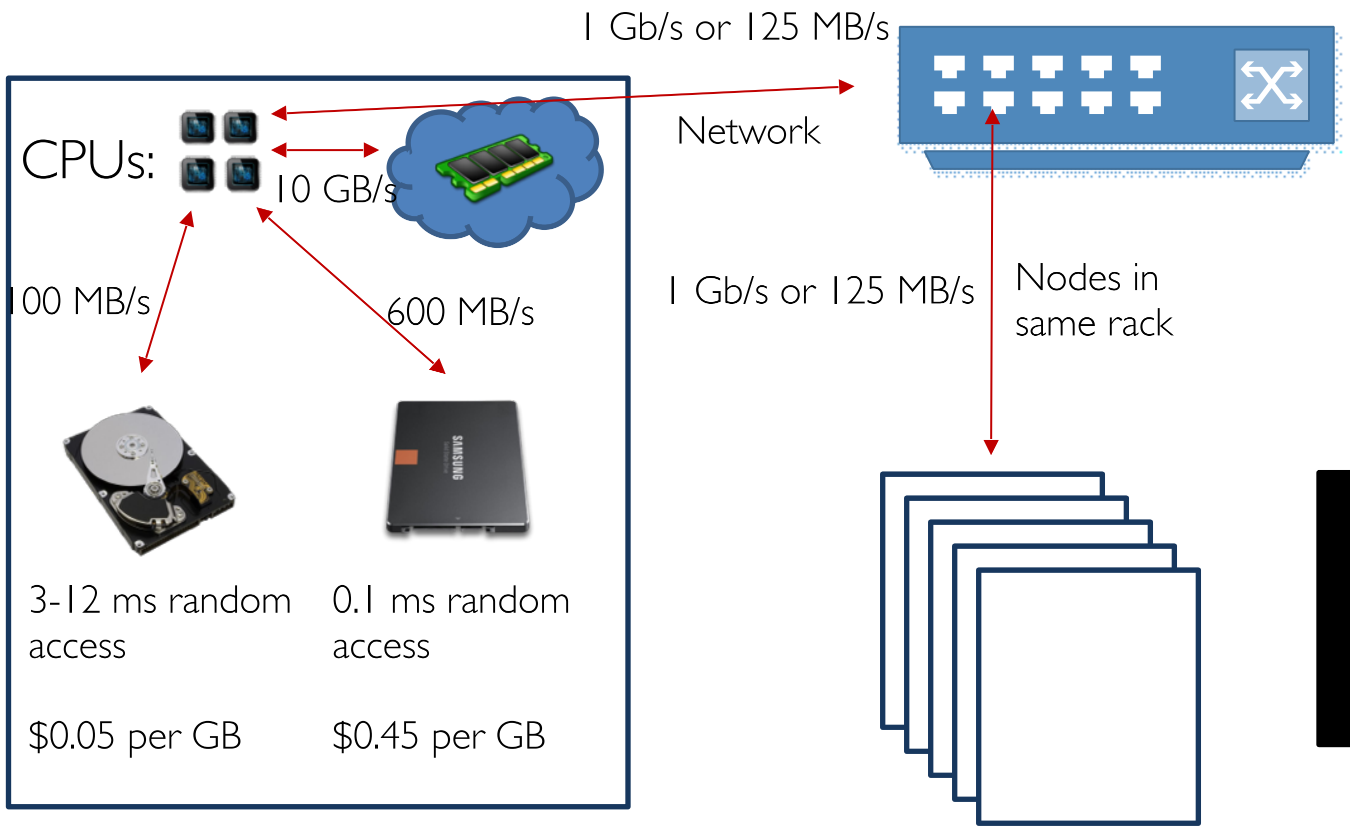
What is Apache Spark?

- *Scalable*, *efficient* *analysis* of *Big Data*

Datacenter Organization

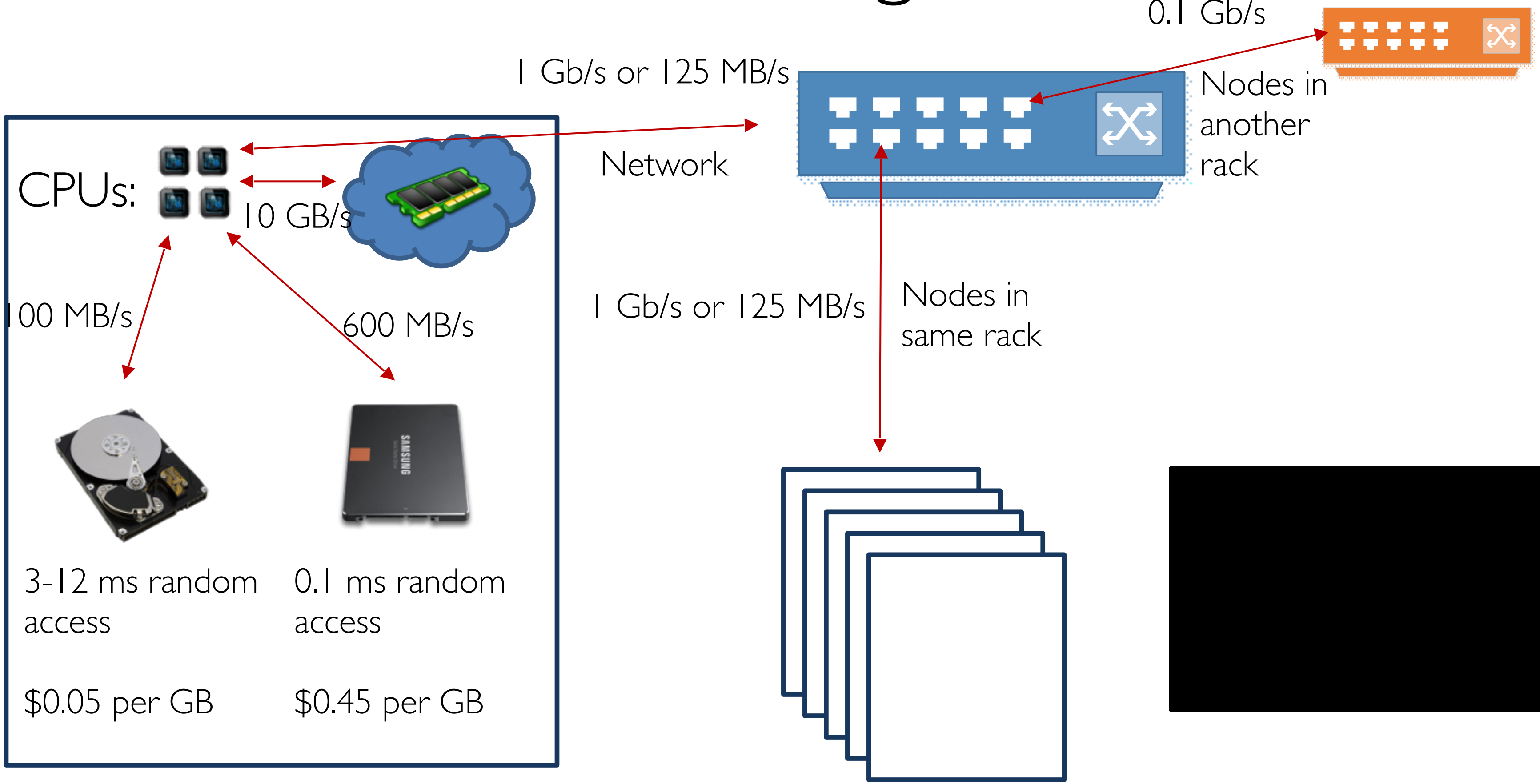


Datacenter Organization

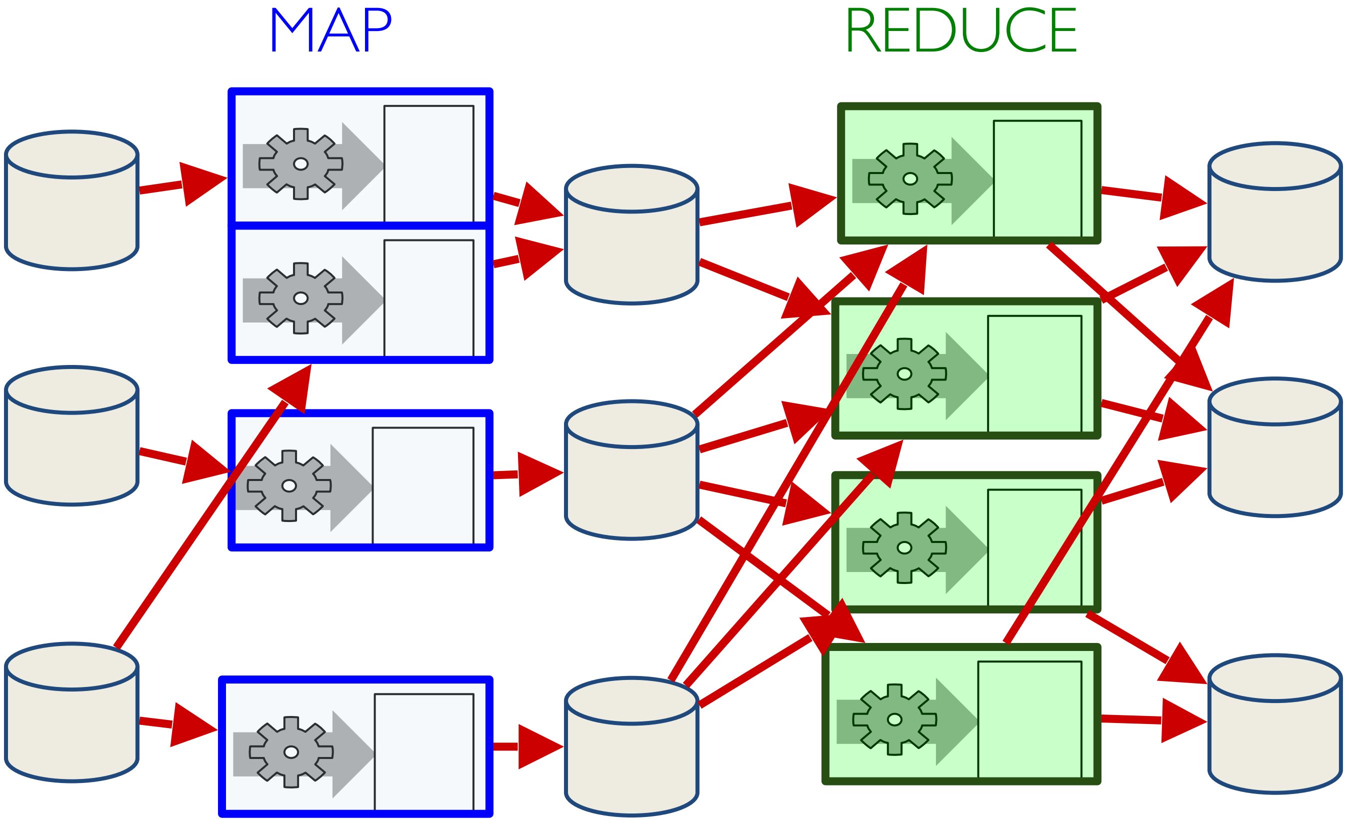


[redacted]

Datacenter Organization



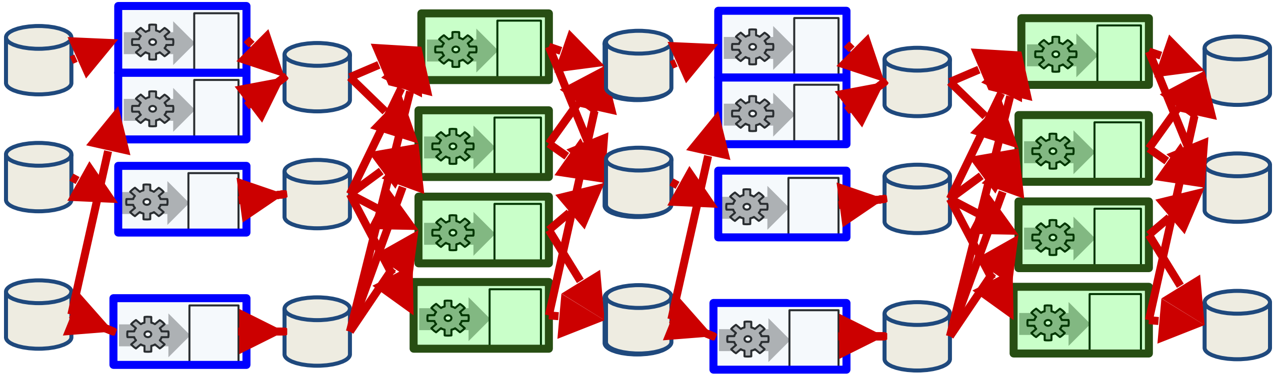
Map Reduce: Distributed Execution



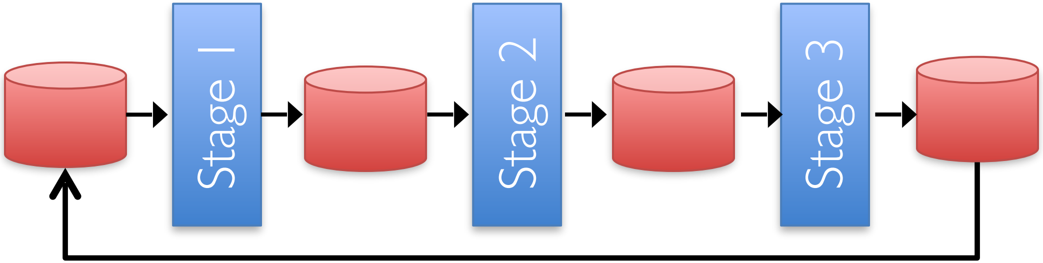
Each stage passes through the hard drives

Map Reduce: Iterative Jobs

- Iterative jobs involve a lot of disk I/O for each repetition

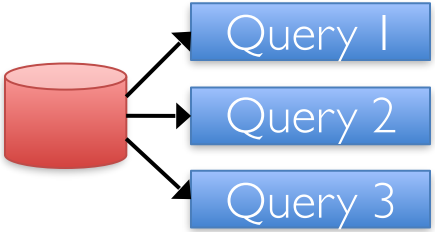


Disk I/O is very slow!

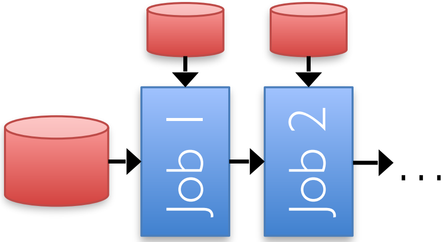


Apache Spark Motivation

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



Interactive mining

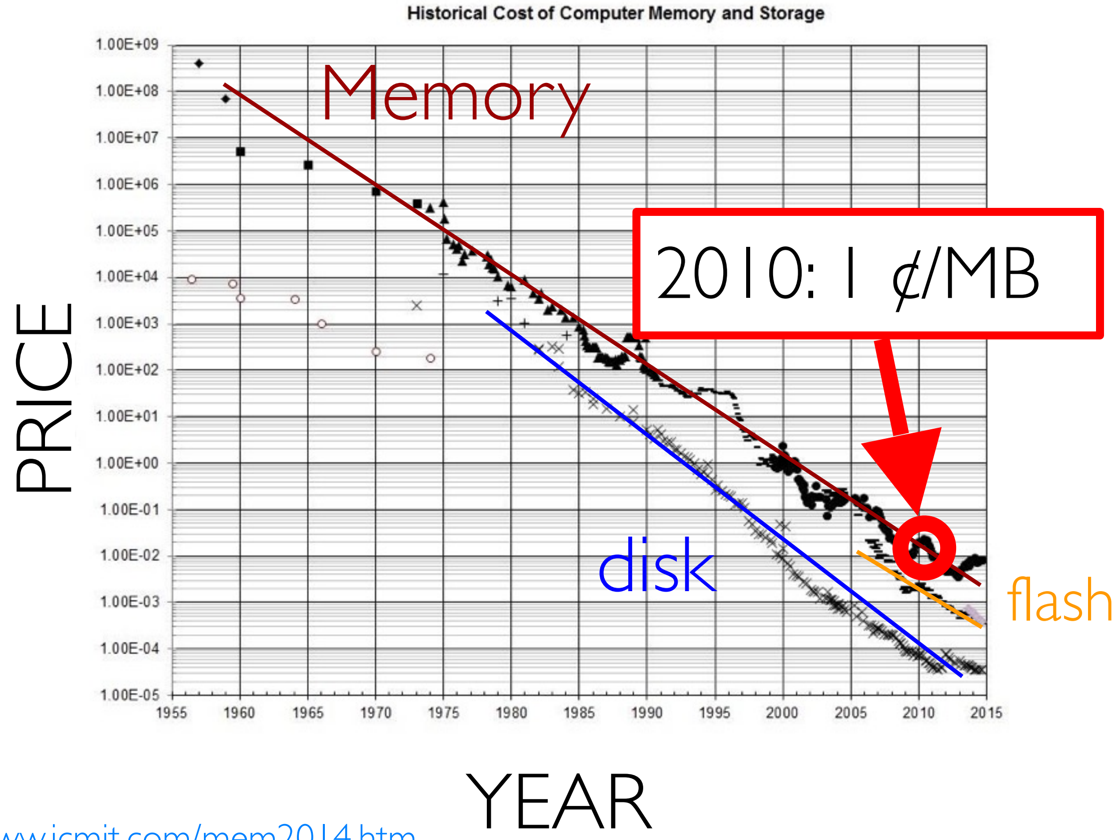


Stream processing

Also, iterative jobs

Disk I/O is very slow

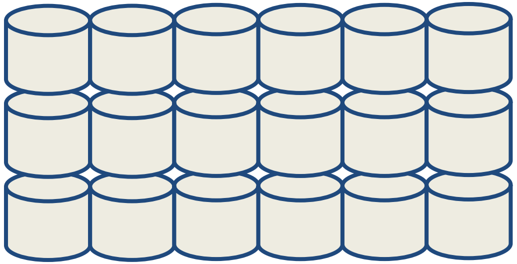
Tech Trend: Cost of Memory



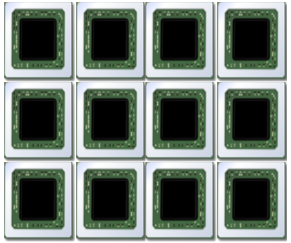
Lower cost means can put more memory in each server

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

Modern Hardware for Big Data



Lots of hard drives



… and CPUs

… and memory!

Opportunity

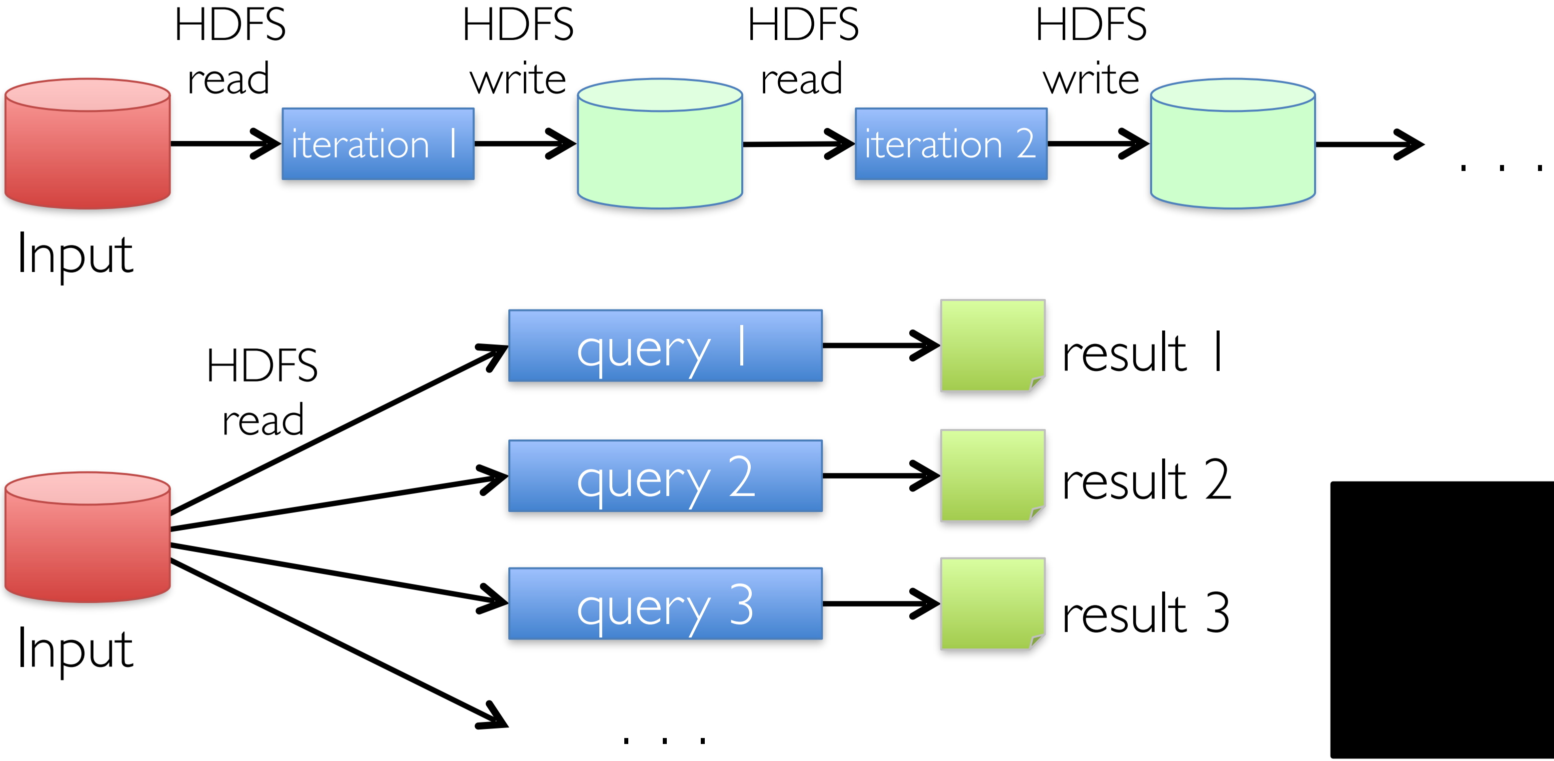
- Keep more data *in-memory*

- Create new distributed execution engine:

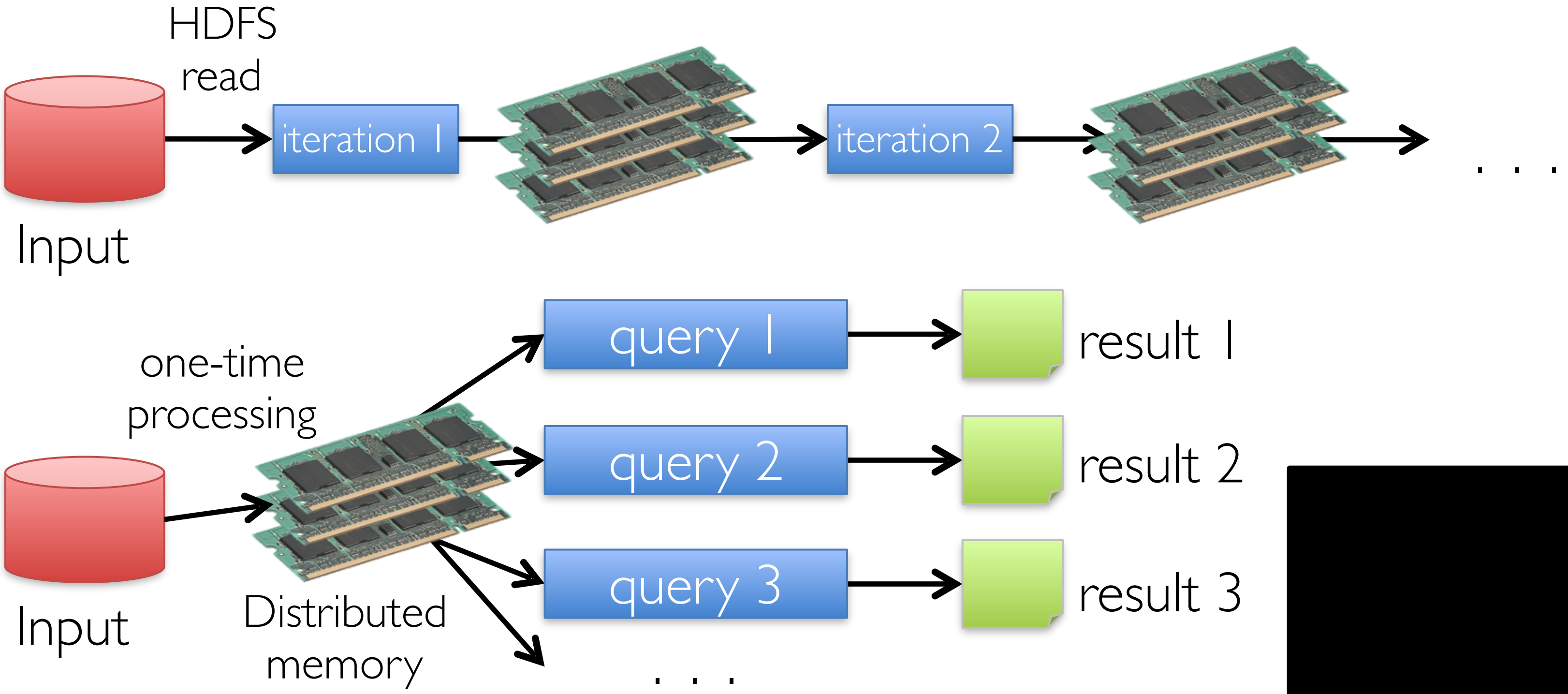
[redacted]

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

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

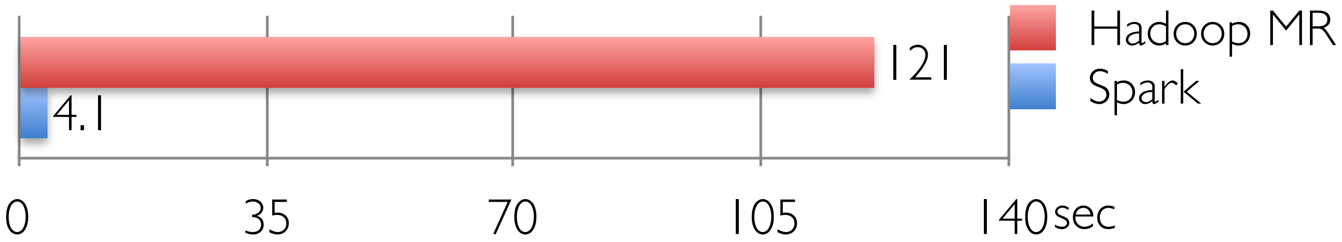
Lower overhead for starting jobs

Less expensive shuffles

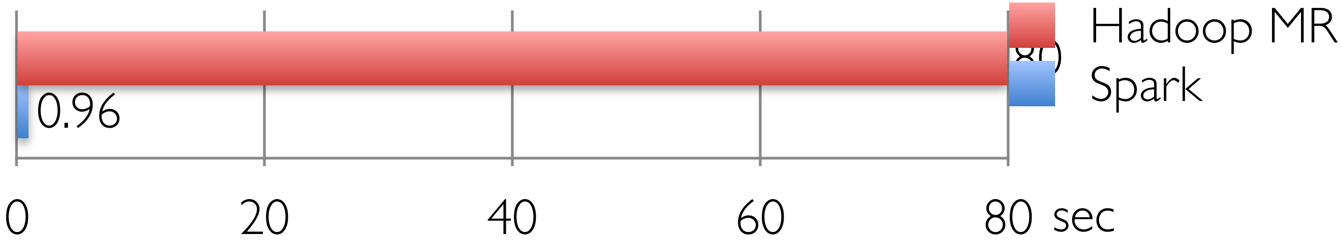
In-Memory Can Make a Big Difference

(2013) Two iterative Machine Learning algorithms:

K-means Clustering



Logistic Regression



[redacted]

First Public Cloud Petabyte Sort (2014)

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

Daytona Gray 100 TB sort benchmark record (tied for 1st place)

http://databricks.com/blog/2014/11/05/spark-officially-sets-a-new-record-in-large-scale-sorting.html

Recent Spark Performance Optimizations

Spark has added two key performance optimizations

» In addition to using memory instead of disk

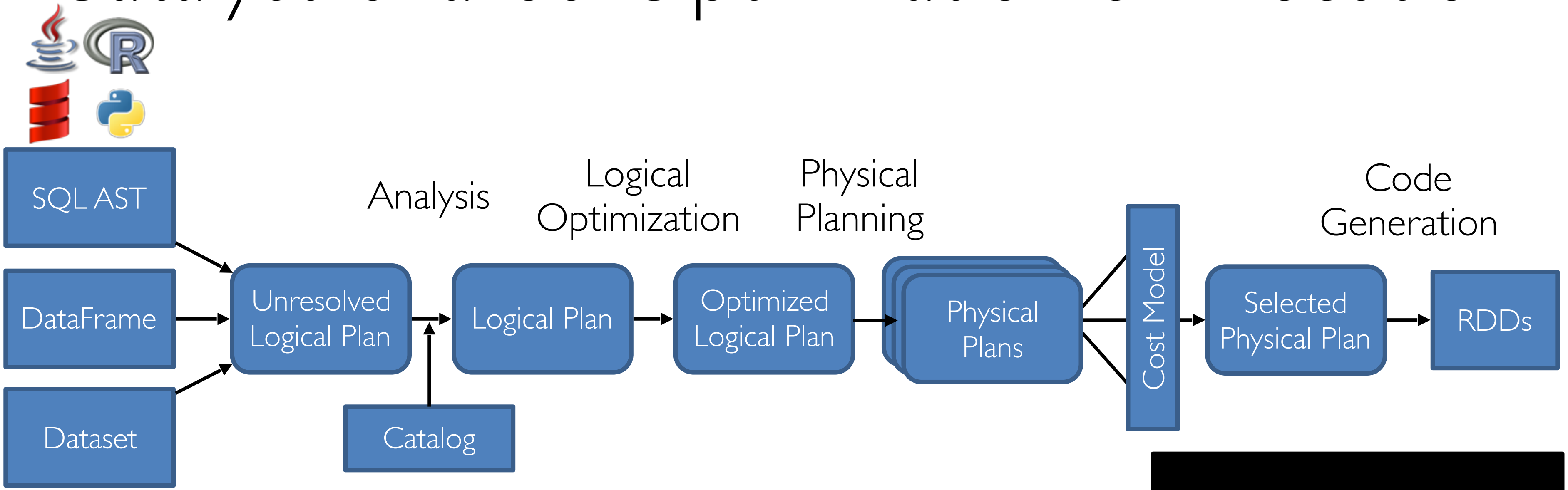
Catalyst Optimization Engine

» 75% reduction in execution time

Project Tungsten off-heap memory management

» 75+% reduction in memory usage (less GC)

Catalyst: Shared Optimization & Execution



DataFrames, Datasets, and Spark SQL
share the same optimization/execution pipeline

Java Virtual Machine Object Overhead

"abcd" →

Native: 4 bytes with UTF-8 encoding

Java: 48 bytes

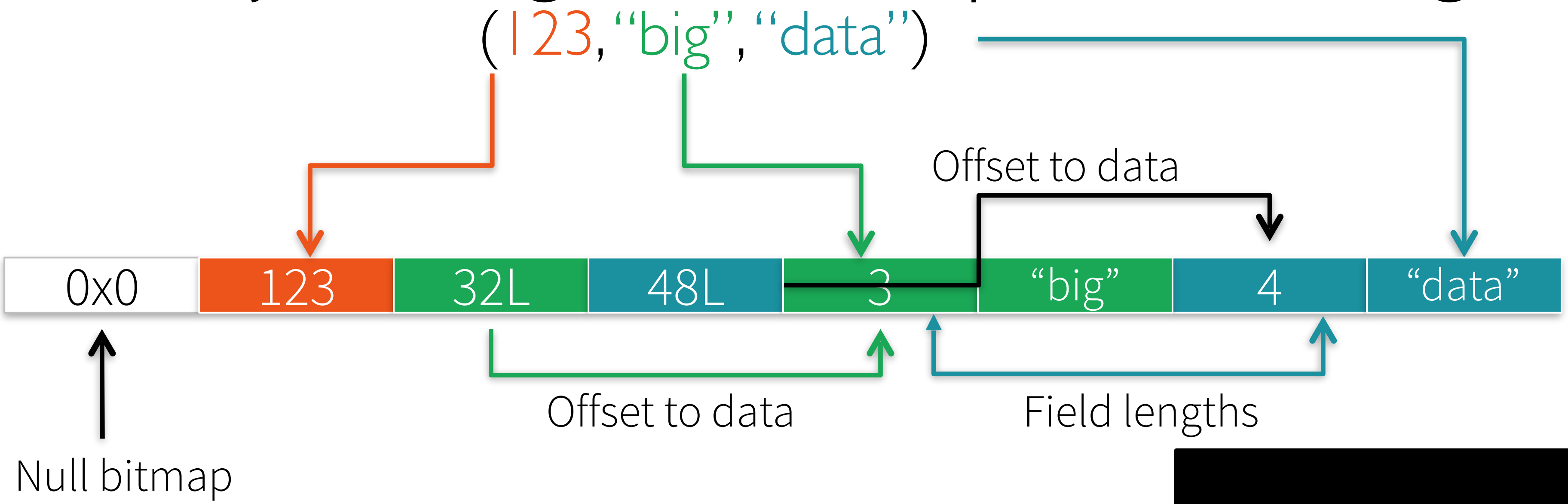
```
java.lang.String object internals:
OFFSET  SIZE   TYPE DESCRIPTION          VALUE
     0     4        (object header)      ...
     4     4        (object header)      ...
     8     4        (object header)      ...
    12     4 char[] String.value         []
    16     4    int String.hash          0
    20     4    int String.hash32        0

Instance size: 24 bytes (reported by Instrumentation API)
```

12 byte object header

20 bytes data + overhead

8 byte hashcode

Project Tungsten's Compact Encoding



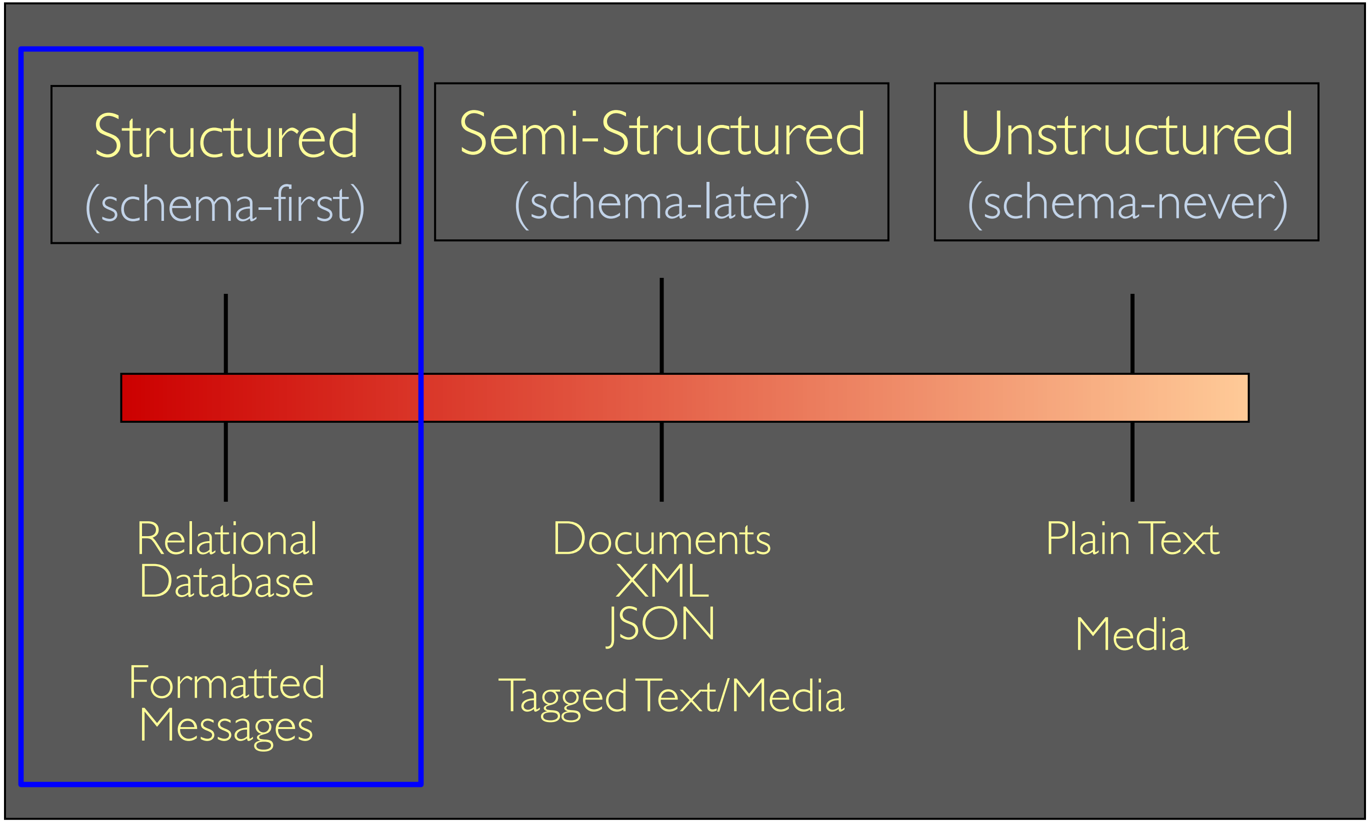
Review: Key Data Management Concepts

- A **data model** is a collection of concepts for describing data
- A **schema** is a description of a particular collection of data, using a given data model
- A **relational data model** is the most used data model
  - » **Relation**, a table with rows and columns
  - » Every relation has a **schema** defining fields in columns

Review: Key Data Management Concepts

- A **data model** is a collection of concepts for describing data
- A **schema** is a description of a particular collection of data, using a given data model
- A **relational data model** is the most used data model
  - » **Relation**, a table with rows and columns
  - » Every relation has a **schema** defining fields in columns

The Structure Spectrum



This lecture

Relational Database: Definitions

- *Relational database:* a set of *relations*

- Two parts to a *Relation*:

Schema: specifies name of relation, plus each column's name and type

```
Students(sid: string, name: string, email: string,
         age: integer, gpa: real)
```

Instance: the actual data at a given time

- #rows = *cardinality*
- #fields = *degree*

What is a Database?

- A large organized collection of data
  - » Transactions used to modify data

- Models real world, e.g., enterprise
  - » Entities (e.g., teams, games)
  - » Relationships, e.g.,
  - » A plays against B in The World Cup

Large Databases

- US Internal Revenue Service: 150 Terabytes
- Australian Bureau of Stats: 250 Terabytes
- AT&T call records: 312 Terabytes
- eBay database: 1.4 Petabytes
- Yahoo click data: 2 Petabytes
- *What matters for these databases?*

Large Databases

- US Internal Revenue Service: 150 Terabytes ← Accuracy, Consistency, Durability, Rich queries
- Australian Bureau of Stats: 250 Terabytes ← Fast, Rich queries
- AT&T call records: 312 Terabytes ← Accuracy, Consistency, Durability
- eBay database: 1.4 Petabytes
- Yahoo click data: 2 Petabytes

Availability
Timeliness

- *What matters for these databases?*

Example: Instance of St... ...on

Attribute names

`Students(sid:string, name:string, login:string, age:integer, gpa:real)`

Table name

sid	name	login	age	gpa
...66	Jones	jones@eecs	18	3.4
53688	Smith	smith@statistics	18	3.2
53650	Smith	smith@math	19	3.8

Tuples or rows

- Cardinality = 3 (rows)
- ... 5 (columns)
- All rows (tuples) are distinct

SQL - A language for Relational DBs

- SQL = Structured Query Language
- Supported by Spark DataFrames (SparkSQL)
- Some of the functionality SQL provides:
  » Create, modify, delete relations
  » Add, modify, remove tuples
  » *Specify queries to find tuples matching criteria*

Queries in SQL

- Single-table queries are straightforward
- To find all 18 year old students, we can write:

```
SELECT *
  FROM Students S
 WHERE S.age=18
```

- To find just names and logins:

```
SELECT S.name, S.login
  FROM Students S
 WHERE S.age=18
```

Querying Multiple Relations

- Can specify a ***join*** over two tables as follows:

```
SELECT S.name, E.cid
 FROM Students S, Enrolled E
 WHERE S.sid=E.sid
```

Students

S

S.sid	S.name	S.login	S.age	S.gpa
53341	Jones	jones@cs	18	3.4
53831	Smith	smith@ee	18	3.2

Enrolled

E

E.sid	E.cid	E.grade
53831	Physics203	A
53650	Topology112	A
53341	History105	B

- s, S and E

Cross Join

- Cartesian product of two tables (E **x** S):

Enrolled

E

E.sid	E.cid	E.grade
53831	Physics203	A
53650	Topology112	A
53341	History105	B

Students

S

S.sid	S.name	S.login	S.age	S.gpa
53341	Jones	jones@cs	18	3.4
53831	Smith	smith@ee	18	3.2

Cross Join

- Cartesian product of two tables (E **x** S):

Enrolled

E

E.sid	E.cid	E.grade
53831	Physics203	A
53650	Topology112	A
53341	History105	B

Students

S

S.sid	S.name	S.login	S.age	S.gpa
53341	Jones	jones@cs	18	3.4
53831	Smith	smith@ee	18	3.2

E.sid	E.cid	E.grade	S.sid	S.name	S.login	S.age	S.gpa
53831	Physics203	A	53341	Jones	jones@cs	18	3.4
53650	Topology112	A	53341	Jones	jones@cs	18	3.4
53341	History105	B	53341	Jones	jones@cs	18	3.4
53831	Physics203	A	53831	Smith	smith@ee	18	3.2
53650	Topology112	A	53831	Smith	smith@ee	18	3.2
53341	History105	B	53831	Smith	smith@ee	18	3.2

Where Clause

- Choose matching rows using Where clause:

```
SELECT S.name, E.cid
 FROM Students S, Enrolled E
 WHERE S.sid=E.sid
```

E.sid	E.cid	E.grade	S.sid	S.name	S.login	S.age	S.gpa
53831	Physics203	A	53341	Jones	jones@cs	18	3.4
53650	Topology112	A	53341	Jones	jones@cs	18	3.4
53341	History105	B	53341	Jones	jones@cs	18	3.4
53831	Physics203	A	53831	Smith	smith@ee	18	3.2
53650	Topology112	A	53831	Smith	smith@ee	18	3.2
53341	History105	B	53831	Smith	smith@ee	18	3.2

[redacted]

Select Clause

- Filter columns using Select clause:

```
SELECT S.name, E.cid
 FROM Students S, Enrolled E
 WHERE S.sid=E.sid
```

E.sid	E.cid	E.grade	S.sid	S.name	S.login	S.age	S.gpa
53831	Physics203	A	53341	Jones	jones@cs	18	3.4
53650	Topology112	A	53341	Jones	jones@cs	18	3.4
53341	History105	B	53341	Jones	jones@cs	18	3.4
53831	Physics203	A	53831	Smith	smith@ee	18	3.2
53650	Topology112	A	53831	Smith	smith@ee	18	3.2
53341	History105	B	53831	Smith	smith@ee	18	3.2

[redacted]

Result

- Can specify a ***join*** over two tables as follows:

```
SELECT S.name, E.cid
 FROM Students S, Enrolled E
 WHERE S.sid=E.sid
```

Enrolled

E

E.sid	E.cid	E.grade
53831	Physics203	A
53650	Topology112	A
53341	History105	B

Students

S

S.sid	S.name	S.login	S.age	S.gpa
53341	Jones	jones@cs	18	3.4
53831	Smith	smith@ee	18	3.2

[redacted]

Result =

S.name	E.cid
Jones	History105
Smith	Physics203

Explicit SQL Joins

```
SELECT S.name, E.classid
 FROM Students S INNER JOIN Enrolled E ON S.sid=E.sid
```

S

S.name	S.sid
Jones	11111
Smith	22222
Brown	33333

E

E.sid	E.classid
11111	History105
11111	DataScience194
22222	French150
44444	English10

Result

S.name	E.classid
Jones	History105
Jones	DataScience194
Smith	French150

Equivalent SQL Join Notations

- Explicit Join notation (preferred):

```
SELECT S.name, E.classid
 FROM Students S INNER JOIN Enrolled E ON S.sid=E.sid
```

```
SELECT S.name, E.classid
 FROM Students S JOIN Enrolled E ON S.sid=E.sid
```

- Implicit join notation (deprecated):

```
SELECT S.name, E.cid
 FROM Students S, Enrolled E
 WHERE S.sid=E.sid
```

SQL Types of Joins

```
SELECT S.name, E.classid
 FROM Students S INNER JOIN Enrolled E ON S.sid=E.sid
```

S

S.name	S.sid
Jones	11111
Smith	22222
Brown	33333

E

E.sid	E.classid
11111	History105
11111	DataScience194
22222	French150
44444	English10

Result

S.name	E.classid
Jones	History105
Jones	DataScience194
Smith	French150

Unmatched keys

[redacted]

The type of join controls how unmatched keys are handled

SQL Joins: Left Outer Join

```
SELECT S.name, E.classid
 FROM Students S LEFT OUTER JOIN Enrolled E ON S.sid=E.sid
```

S

S.name	S.sid
Jones	11111
Smith	22222
Brown	33333

E

E.sid	E.classid
11111	History105
11111	DataScience194
22222	French150
44444	English10

Result

S.name	E.classid
Jones	History105
Jones	DataScience194
Smith	French150
Brown	<NULL>

Unmatched keys

SQL Joins: Right Outer Join

```
SELECT S.name, E.classid
 FROM Students S RIGHT OUTER JOIN Enrolled E ON
S.sid=E.sid
```

S

S.name	S.sid
Jones	11111
Smith	22222
Brown	33333

E

E.sid	E.classid
11111	History105
11111	DataScience194
22222	French150
44444	English10

Result

S.name	E.classid
Jones	History105
Jones	DataScience194
Smith	French150
<NULL>	English10

Unmatched keys

Spark Joins

- SparkSQL and Spark DataFrames support joins
- `join(other, on, how)`:
  - » other – right side of the join
  - » on –join column name, list of column (names), or join expression
  - » how – inner, outer, left_outer, right_outer, left_semi

Spark Join Examples(I)

```
>>> df = sqlContext.createDataFrame(data, ['name', 'age'])
[Row(name=u'Alice', age=1), Row(name=u'Bob', age=2)]
>>> df2 = sqlContext.createDataFrame(data2,['name', 'height'])
[Row(name=u'Chris', height=80), Row(name=u'Bob', height=85)]

>>> df.join(df2, 'name')
[Row(name=u'Bob', age=2, height=85)]
```

[redacted]

Inner Join – `X.join(Y, cols)`

» Return DF of rows with matching **cols** in both X and Y

Spark Join Examples(II)

```
>>> df = sqlContext.createDataFrame(data, ['name', 'age'])
[Row(name=u'Alice', age=1), Row(name=u'Bob', age=2)]
>>> df2 = sqlContext.createDataFrame(data2,['name', 'height'])
[Row(name=u'Chris', height=80), Row(name=u'Bob', height=85)]

>>> df.join(df2, 'name').select(df.name, df2.height)
[Row(name=u'Bob', height=85)]
```

[redacted]

Inner Join – `X.join(Y, cols)`

» Return DF of rows with matching **cols** in both X and Y

Spark Join Examples(III)

```
>>> df = sqlContext.createDataFrame(data, ['name', 'age'])
[Row(name=u'Alice', age=1), Row(name=u'Bob', age=2)]
>>> df2 = sqlContext.createDataFrame(data2,['name', 'height'])
[Row(name=u'Chris', height=80), Row(name=u'Bob', height=85)]

>>> df.join(df2, 'name', 'outer')
[Row(name=u'Chris', age=None, height=80),
 Row(name=u'Alice', age=1, height=None),
 Row(name=u'Bob', age=2, height=85)]
```

Outer Join – `X.join(Y, cols,'outer')`

[redacted]

- » Return DF of rows with matching **cols** in either X and Y

Spark Join Examples(IV)

```
>>> df = sqlContext.createDataFrame(data, ['name', 'age'])
[Row(name=u'Alice', age=1), Row(name=u'Bob', age=2)]
>>> df2 = sqlContext.createDataFrame(data2,['name', 'height'])
[Row(name=u'Chris', height=80), Row(name=u'Bob', height=85)]

>>> df.join(df2, 'name', 'outer').select('name', 'height')
[Row(name=u'Chris', height=80),
 Row(name=u'Alice', height=None),
 Row(name=u'Bob', height=85)]
```

[redacted]

Outer Join – `X.join(Y, cols,'outer')`

» Return DF of rows with matching **cols** in either X and Y

Spark Join Examples(V)

```
>>> df = sqlContext.createDataFrame(data, ['name', 'age'])
[Row(name=u'Alice', age=1), Row(name=u'Bob', age=2)]
>>> df2 = sqlContext.createDataFrame(data2,['name', 'height'])
[Row(name=u'Chris', height=80), Row(name=u'Bob', height=85)]

>>> df.join(df2, 'name', 'left_outer')
[Row(name=u'Alice', age=1, height=None),
 Row(name=u'Bob', age=2, height=85)]
```

Left Outer Join – **`X.join(Y, cols, 'left_outer')`**

[redacted]

» Return DF of rows with matching **cols** in X

Spark Join Examples(VI)

```
>>> df = sqlContext.createDataFrame(data, ['name', 'age'])
[Row(name=u'Alice', age=1), Row(name=u'Bob', age=2)]
>>> df2 = sqlContext.createDataFrame(data2,['name', 'height'])
[Row(name=u'Chris', height=80), Row(name=u'Bob', height=85)]

>>> df.join(df2, 'name', 'right_outer')
[Row(name=u'Chris', age=None, height=80),
 Row(name=u'Bob', age=2, height=85)]
```

Right Outer Join – **`X.join(Y, cols, 'right_outer')`**

» Return DF of rows with matching **cols** in **Y**

Spark Online Documentation

https://spark.apache.org/docs/latest/

Spark 1.6.1 Overview Programming Guides API Docs Deploying More

Spark Overview

Apache Spark is a fast and general-purpose cluster computing system. It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs. It also supports a rich set of higher-level tools including Spark SQL for SQL and structured data processing, MLlib for machine learning, GraphX for graph processing, and Spark Streaming.

Downloading

Get Spark from the downloads page of the project website. This documentation is for Spark version 1.6.1. Spark uses Hadoop's client libraries for HDFS and YARN. Downloads are pre-packaged for a handful of popular Hadoop versions. Users can also download a "Hadoop free" binary and run Spark with any Hadoop version by augmenting Spark's classpath.

If you'd like to build Spark from source, visit Building Spark.

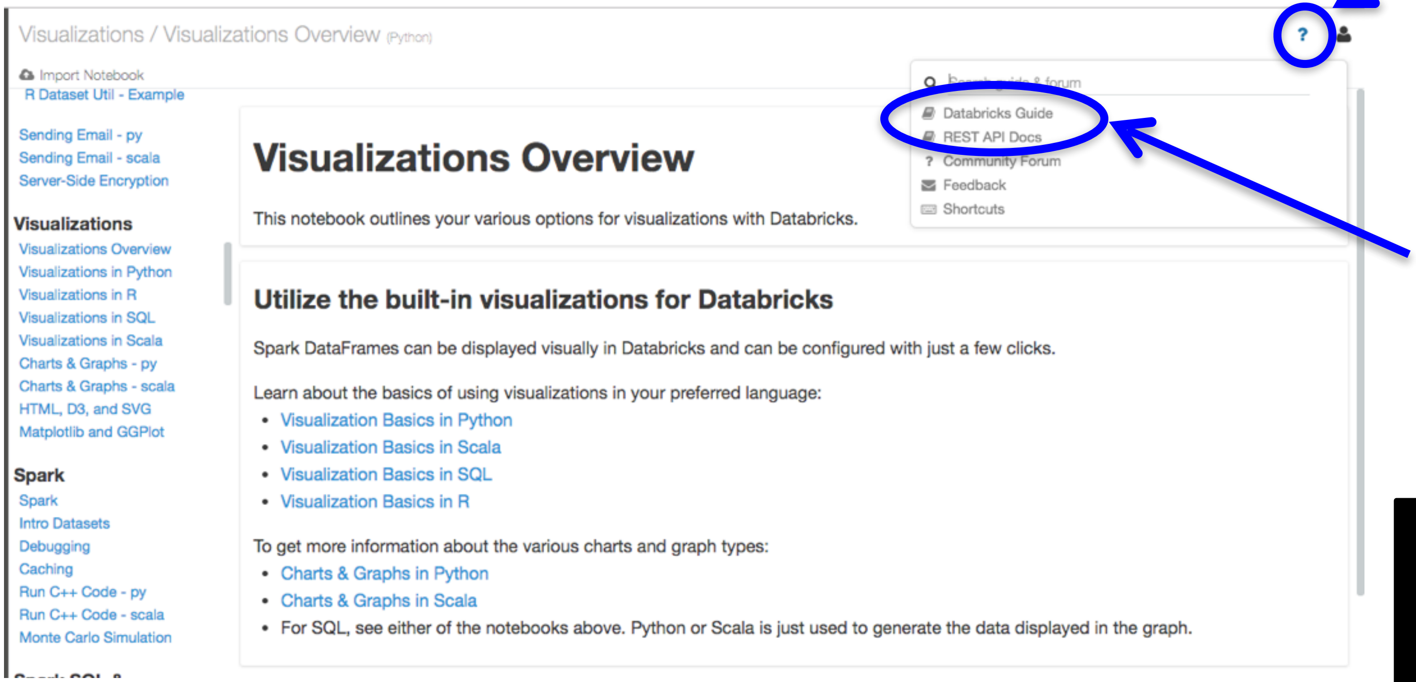
Spark runs on both Windows and UNIX-like systems (e.g. Linux, Mac OS). It's easy to run locally on one machine — all you need is to have `java` installed on your system `PATH`, or the `JAVA_HOME` environment variable pointing to a Java installation.

Spark runs on Java 7+, Python 2.6+ and R 3.1+. For the Scala API, Spark 1.6.1 uses Scala 2.10. You will need to use a compatible Scala version (2.10.x).

API Docs

Scala
Java
Python
R

Databricks Guide



Spark Technical Blogs

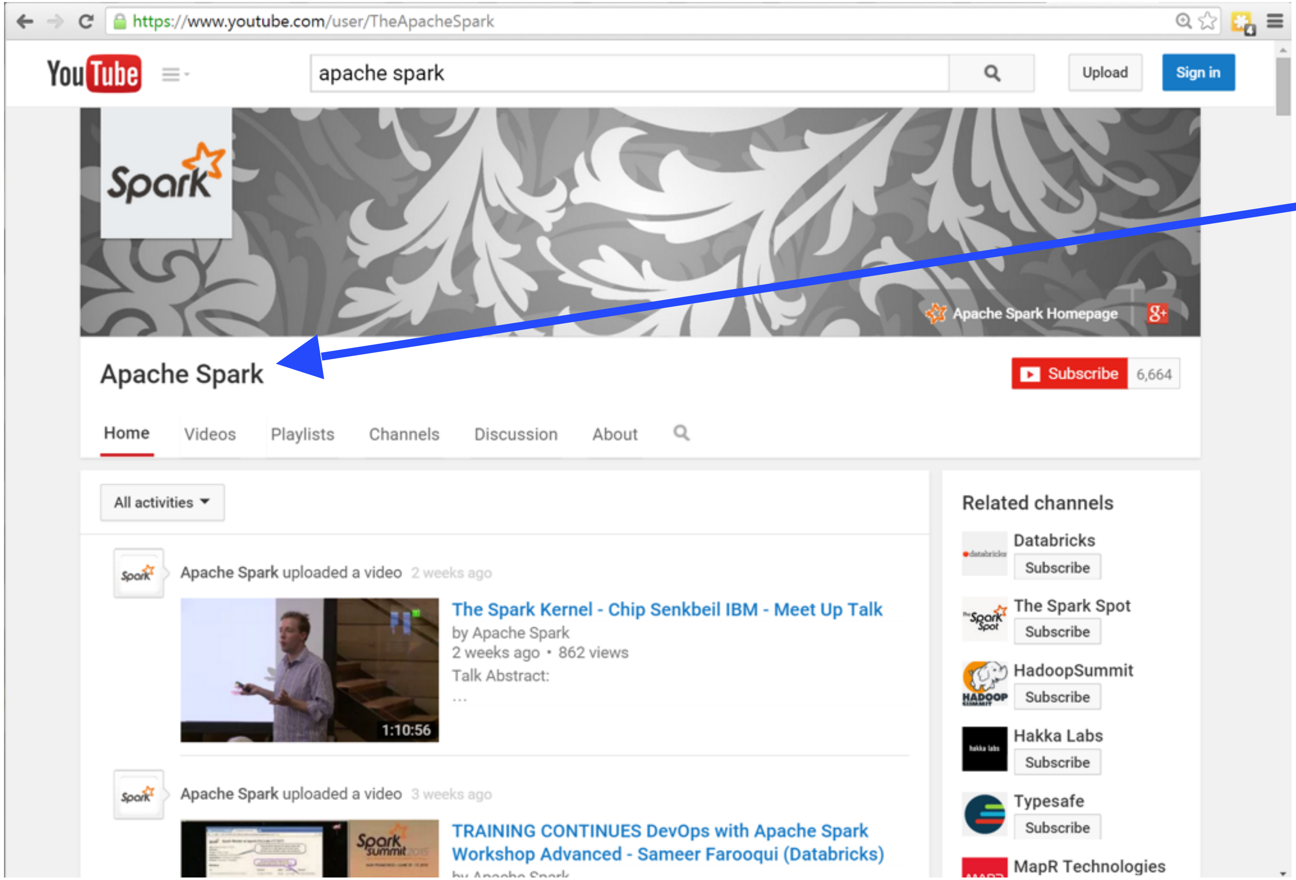
Databricks: https://databricks.com/blog/category/engineering

Cloudera: http://blog.cloudera.com/blog/category/spark/

IBM: http://www.spark.tc/blog/

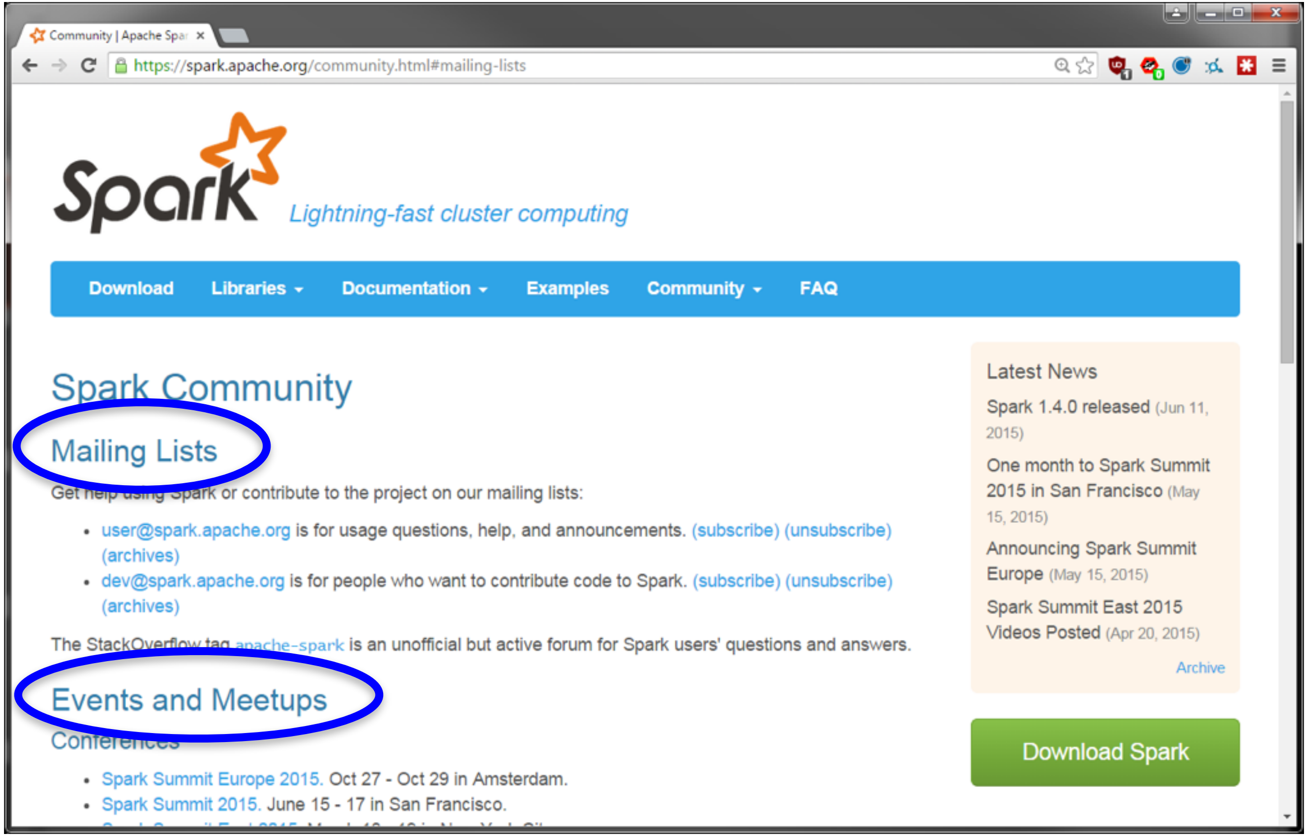
- Hortonworks: http://hortonworks.com/blog/category/spark/
- Many more! (eBay, AWS, MapR, Datastax, etc)

Spark on YouTube



Check out the [Apache Spark](https://www.youtube.com/user/TheApacheSpark) YouTube Channel!

Spark Community http://spark.apache.org/community.html



Spark Meetups

Apache Spark

Find out what's happening in Apache Spark Meetup groups around the world and start meeting up with the ones near you.

186,279 members | **421** Meetups

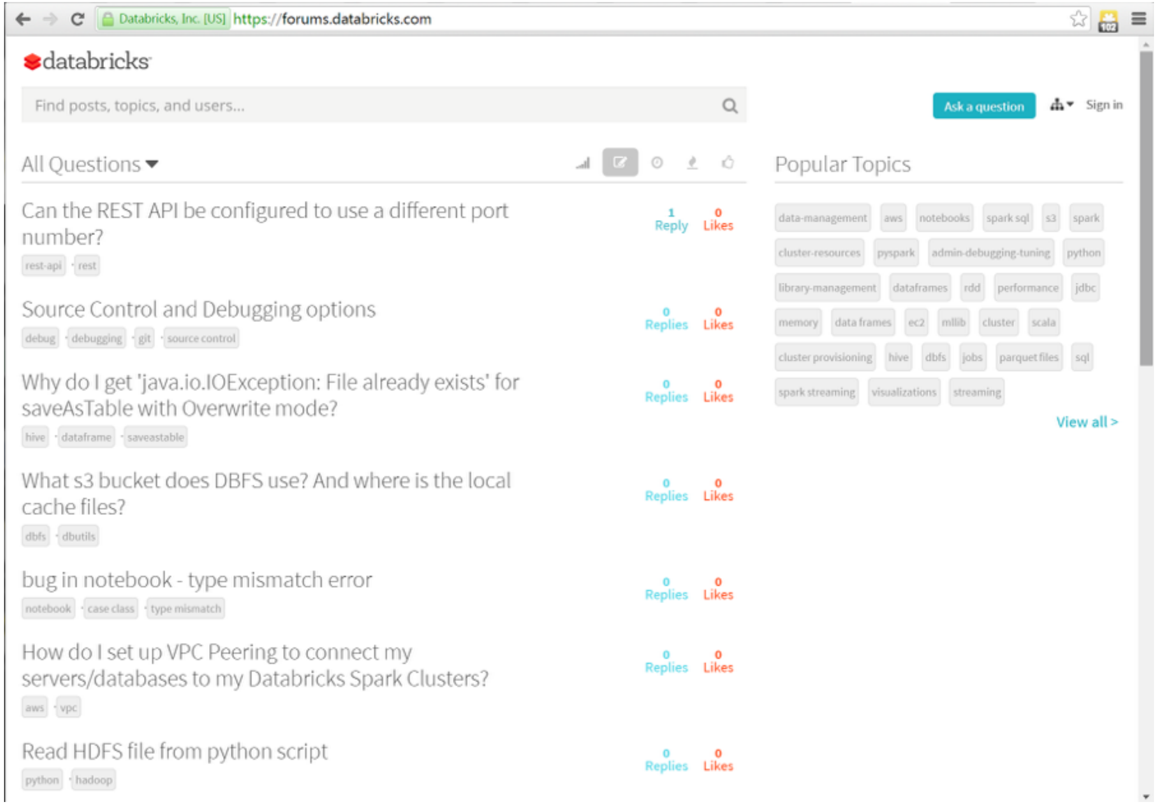
Join Apache Spark Meetups

http://spark.meetup.com/

Related topics: Big Data · Hadoop · Machine Learning · Data Analytics · Big Data Analytics · Data Science · Apache Kafka · MapReduce · Data Mining · Scala

Databricks Forums

https://forums.databricks.com/



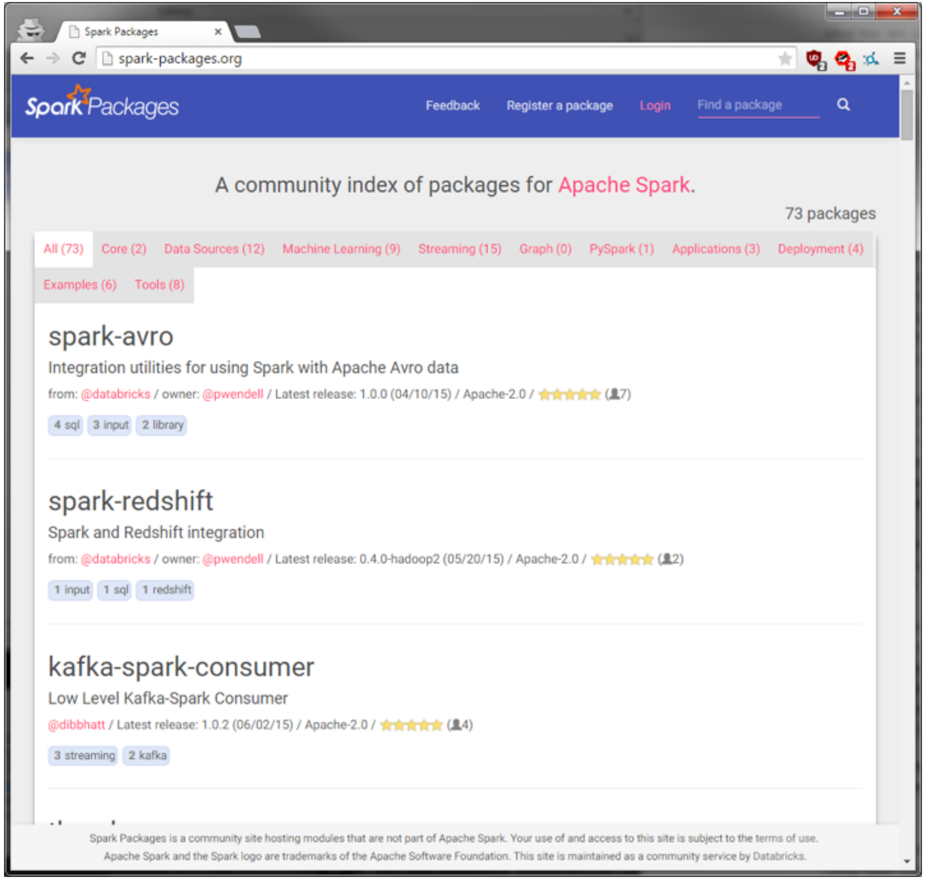
Community forum for Databricks users

Mostly Databricks-specific Q&A

Some general Spark Q&A

Spark Packages

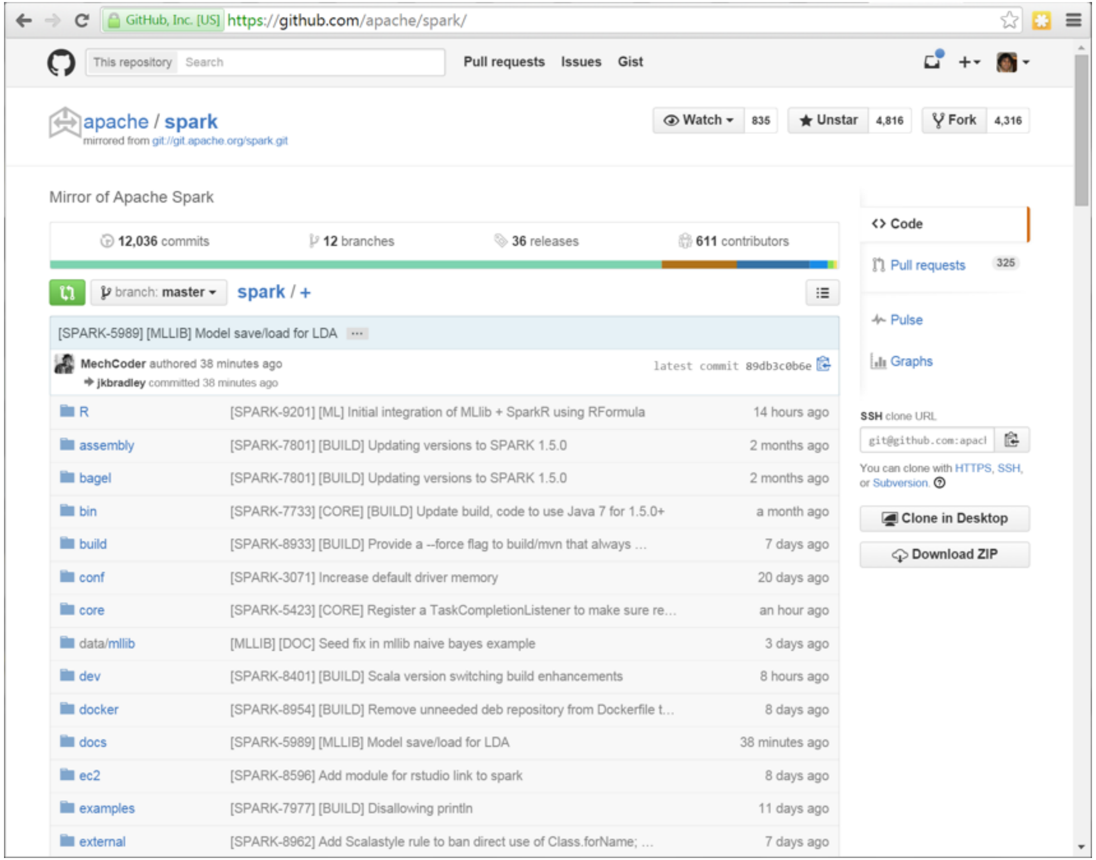
http://spark-packages.org/



232 software packages for Spark

- » User-provided Spark extensions
- » Community votes ★★★★★ (8)

Spark Source Code



https://github.com/apache/spark/

Hint: For detailed explanations, check out comments in code

Research Papers

Spark: Cluster Computing with Working Sets

http://www.cs.berkeley.edu/~matei/papers/2010/hotcloud_spark.pdf

June 2010

Spark: Cluster Computing with Working Sets

Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica
University of California, Berkeley

Abstract

MapReduce and its variants have been highly successful in implementing large-scale data-intensive applications on commodity clusters. However, most of these systems are built around an acyclic data flow model that is not suitable for other popular applications. This paper focuses on one such class of applications: those that reuse a working set of data across multiple parallel operations. This includes many iterative machine learning algorithms, as well as interactive data analysis tools. We propose a new framework called Spark that supports these applications while retaining the scalability and fault tolerance of MapReduce. To achieve these goals, Spark introduces an abstraction called resilient distributed datasets (RDDs). An RDD is a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. Spark can outperform Hadoop by 10x in iterative machine learning jobs, and can be used to interactively query a 39 GB dataset with sub-second response time.

1 Introduction

A new model of cluster computing has become widely popular, in which data-parallel computations are executed on clusters of unreliable machines by systems that automatically provide locality-aware scheduling, fault tolerance, and load balancing. MapReduce [11] pioneered this model, while systems like Dryad [17] and Map-Reduce-Merge [24] generalized the types of data flows supported. These systems achieve their scalability and fault tolerance by providing a programming model where the user creates acyclic data flow graphs to pass input data through a set of operators. This allows the underlying system to manage scheduling and to react to faults without user intervention.

While this data flow programming model is useful for a large class of applications, there are applications that cannot be expressed efficiently as acyclic data flows. In this paper, we focus on one such class of applications: those that reuse a *working set* of data across multiple parallel operations. This includes two use cases where we have seen Hadoop users report that MapReduce is deficient:

- **Iterative jobs**: Many common machine learning algorithms apply a function repeatedly to the same dataset to optimize a parameter (e.g., through gradient descent). While each iteration can be expressed as a MapReduce/Dryad job, each job must reload the data from disk, incurring a significant performance penalty.
- **Interactive analytics**: Hadoop is often used to run ad-hoc exploratory queries on large datasets, through SQL interfaces such as Pig [21] and Hive [1]. Ideally, a user would be able to load a dataset of interest into memory across a number of machines and query it repeatedly. However, with Hadoop, each query incurs significant latency (tens of seconds) because it runs as a separate MapReduce job and reads data from disk.

This paper presents a new cluster computing framework called Spark, which supports applications with working sets while providing similar scalability and fault tolerance properties to MapReduce.

The main abstraction in Spark is that of a *resilient distributed dataset* (RDD), which represents a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. Users can explicitly cache an RDD in memory across machines and reuse it in multiple MapReduce-like *parallel operations*. RDDs achieve fault tolerance through a notion of *lineage*: if a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to be able to rebuild just that partition. Although RDDs are not a general shared memory abstraction, they represent a sweet-spot between expressivity on the one hand and scalability and reliability on the other hand, and we have found them well-suited for a variety of applications.

Spark is implemented in Scala [5], a statically typed high-level programming language for the Java VM, and exposes a functional programming interface similar to DryadLINQ [25]. In addition, Spark can be used interactively from a modified version of the Scala interpreter, which allows the user to define RDDs, functions, variables and classes and use them in parallel operations on a cluster. We believe that Spark is the first system to allow an efficient, general-purpose programming language to be used interactively to process large datasets on a cluster.

Although our implementation of Spark is still a prototype, early experience with the system is encouraging. We show that Spark can outperform Hadoop by 10x in iterative machine learning workloads and can be used interactively to scan a 39 GB dataset with sub-second latency.

This paper is organized as follows. Section 2 describes

Research Papers

Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma,
Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica
University of California, Berkeley

Abstract

We present Resilient Distributed Datasets (RDDs), a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner. RDDs are motivated by two types of applications that current computing frameworks handle inefficiently: iterative algorithms and interactive data mining tools. In both cases, keeping data in memory can improve performance by an order of magnitude. To achieve fault tolerance efficiently, RDDs provide a restricted form of shared memory, based on coarse-grained transformations rather than fine-grained updates to shared state. However, we show that RDDs are expressive enough to capture a wide class of computations, including recent specialized programming models for iterative jobs, such as Pregel, and new applications that these models do not capture. We have implemented RDDs in a system called Spark, which we evaluate through a variety of user applications and benchmarks.

1 Introduction

Cluster computing frameworks like MapReduce [10] and Dryad [19] have been widely adopted for large-scale data analytics. These systems let users write parallel computations using a set of high-level operators, without having to worry about work distribution and fault tolerance.

Although current frameworks provide numerous abstractions for accessing a cluster's computational resources, they lack abstractions for leveraging distributed memory. This makes them inefficient for an important class of emerging applications: those that *reuse* intermediate results across multiple computations. Data reuse is common in many *iterative* machine learning and graph algorithms, including PageRank, K-means clustering, and logistic regression. Another compelling use case is *interactive* data mining, where a user runs multiple ad-hoc queries on the same subset of the data. Unfortunately, in most current frameworks, the only way to reuse data between computations (*e.g.*, between two MapReduce jobs) is to write it to an external stable storage system, *e.g.*, a distributed file system. This incurs substantial overheads due to data replication, disk I/O, and serialization, which can dominate application execution times.

Recognizing this problem, researchers have developed specialized frameworks for some applications that require data reuse. For example, Pregel [22] is a system for iterative graph computations that keeps intermediate data in memory, while HaLoop [7] offers an iterative MapReduce interface. However, these frameworks only support specific computation patterns (*e.g.*, looping a series of MapReduce steps), and perform data sharing implicitly for these patterns. They do not provide abstractions for more general reuse, *e.g.*, to let a user load several datasets into memory and run ad-hoc queries across them.

In this paper, we propose a new abstraction called *resilient distributed datasets (RDDs)* that enables efficient data reuse in a broad range of applications. RDDs are fault-tolerant, parallel data structures that let users explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using a rich set of operators.

The main challenge in designing RDDs is defining a programming interface that can provide fault tolerance *efficiently*. Existing abstractions for in-memory storage on clusters, such as distributed shared memory [24], key-value stores [25], databases, and Piccolo [27], offer an interface based on fine-grained updates to mutable state (*e.g.*, cells in a table). With this interface, the only ways to provide fault tolerance are to replicate the data across machines or to log updates across machines. Both approaches are expensive for data-intensive workloads, as they require copying large amounts of data over the cluster network, whose bandwidth is far lower than that of RAM, and they incur substantial storage overhead.

In contrast to these systems, RDDs provide an interface based on *coarse-grained* transformations (*e.g.*, map, filter and join) that apply the same operation to many data items. This allows them to efficiently provide fault tolerance by logging the transformations used to build a dataset (its *lineage*) rather than the actual data.[1] If a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to recompute

[1] Checkpointing the data in some RDDs may be useful when a lineage chain grows large, however, and we discuss how to do it in §5.4.

Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing

http://www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf

April 2012

Spark SQL

Spark SQL: Relational Data Processing in Spark

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[†*]

[†]Databricks Inc. [*]MIT CSAIL [‡]AMPLab, UC Berkeley

ABSTRACT

Spark SQL is a new module in Apache Spark that integrates relational processing with Spark's functional programming API. Built on our experience with Shark, Spark SQL lets Spark programmers leverage the benefits of relational processing (*e.g.*, declarative queries and optimized storage), and lets SQL users call complex analytics libraries in Spark (*e.g.*, machine learning). Compared to previous systems, Spark SQL makes two main additions. First, it offers much tighter integration between relational and procedural processing, through a declarative DataFrame API that integrates with procedural Spark code. Second, it includes a highly extensible optimizer, Catalyst, built using features of the Scala programming language, that makes it easy to add composable rules, control code generation, and define extension points. Using Catalyst, we have built a variety of features (*e.g.*, schema inference for JSON, machine learning types, and query federation to external databases) tailored for the complex needs of modern data analysis. We see Spark SQL as an evolution of both SQL-on-Spark and of Spark itself, offering richer APIs and optimizations while keeping the benefits of the Spark programming model.

Categories and Subject Descriptors

H.2 [**Database Management**]: Systems

Keywords

Databases; Data Warehouse; Machine Learning; Spark; Hadoop

1 Introduction

Big data applications require a mix of processing techniques, data sources and storage formats. The earliest systems designed for these workloads, such as MapReduce, gave users a powerful, but

While the popularity of relational systems shows that users often prefer writing declarative queries, the relational approach is insufficient for many big data applications. First, users want to perform ETL to and from various data sources that might be semi- or unstructured, requiring custom code. Second, users want to perform advanced analytics, such as machine learning and graph processing, that are challenging to express in relational systems. In practice, we have observed that most data pipelines would ideally be expressed with a combination of both relational queries and complex procedural algorithms. Unfortunately, these two classes of systems—relational and procedural—have until now remained largely disjoint, forcing users to choose one paradigm or the other.

This paper describes our effort to combine both models in Spark SQL, a major new component in Apache Spark [39]. Spark SQL builds on our earlier SQL-on-Spark effort, called Shark. Rather than forcing users to pick between a relational or a procedural API, however, Spark SQL lets users seamlessly intermix the two.

Spark SQL bridges the gap between the two models through two contributions. First, Spark SQL provides a *DataFrame API* that can perform relational operations on both external data sources and Spark's built-in distributed collections. This API is similar to the widely used data frame concept in R [32], but evaluates operations lazily so that it can perform relational optimizations. Second, to support the wide range of data sources and algorithms in big data, Spark SQL introduces a novel extensible optimizer called *Catalyst*. Catalyst makes it easy to add data sources, optimization rules, and data types for domains such as machine learning.

The DataFrame API offers rich relational/procedural integration within Spark programs. DataFrames are collections of structured records that can be manipulated using Spark's procedural API, or using new relational APIs that allow richer optimizations. They can

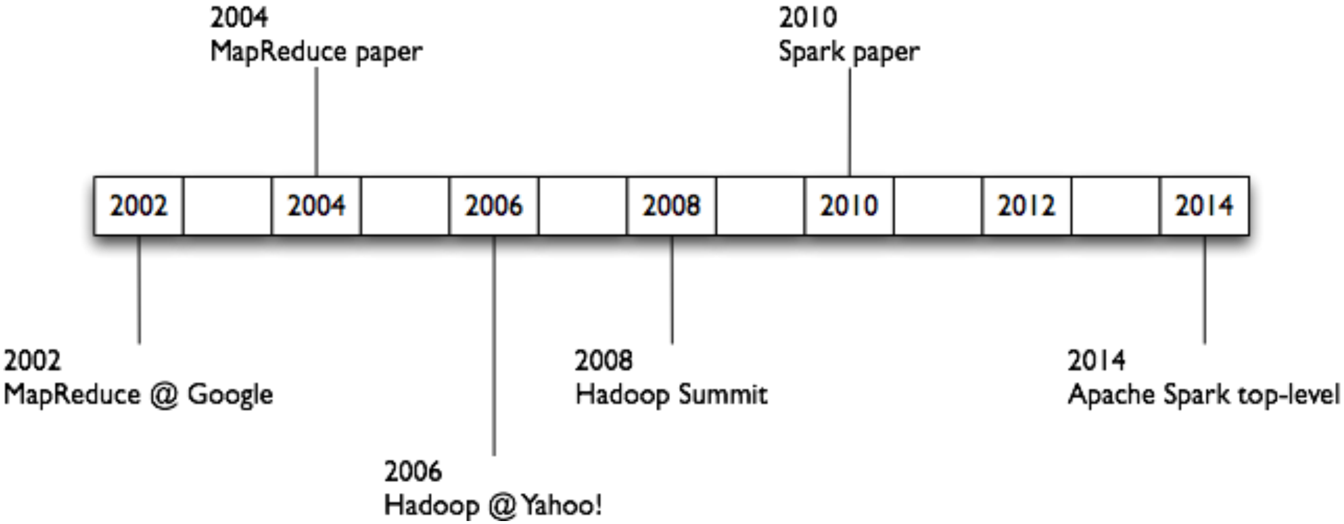
Spark SQL: Relational Data Processing in Spark

Seemlessly mix SQL queries with Spark programs

June 2015

https://amplab.cs.berkeley.edu/wp-content/uploads/2015/03/SparkSQLSigmod2015.pdf

History Summary



Historical References

- circa 1979 – Stanford, MIT, CMU, etc.: set/list operations in LISP, Prolog, etc., for parallel processing
  http://www-formal.stanford.edu/jmc/history/lisp/lisp.html
- circa 2004 – Google: *MapReduce: Simplified Data Processing on Large Clusters*
  Jeffrey Dean and Sanjay Ghemawat
  http://research.google.com/archive/mapreduce.html
- circa 2006 – Apache *Hadoop*, originating from the Yahoo!'s Nutch Project
  Doug Cutting
  http://nutch.apache.org/
- circa 2008 – Yahoo!: web scale search indexing
  Hadoop Summit, HUG, etc.
  http://hadoop.apache.org/
- circa 2009 – Amazon AWS: Elastic MapReduce
  Hadoop modified for EC2/S3, plus support for Hive, Pig, Cascading, etc.
  http://aws.amazon.com/elasticmapreduce/

Spark Research Papers

- *Spark: Cluster Computing with Working Sets*
  Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica
  USENIX HotCloud (2010)
  people.csail.mit.edu/matei/papers/2010/hotcloud_spark.pdf

- *Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing*
  Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica
  NSDI (2012)
  usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf