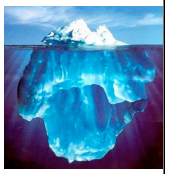
CS162
Operating Systems and Systems Programming
Lecture 24

Berkeley Data Analytics Stack (BDAS)

April 26th, 2017
Prof. Ion Stoica
http://cs162.eecs.Berkeley.edu

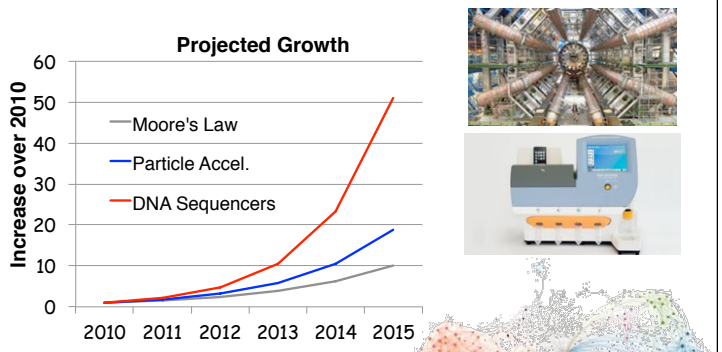
Data Deluge

- Billions of users connected through the net
  - WWW, FB, twitter, cell phones, …
  - 80% of the data on FB was produced last year
  - FB building Exabyte ($2^{60} \cong 10^{18}$) data centers



- It's all happening online – could record every:
  - Click, ad impression, billing event, server request, transaction, network msg, fault, fast forward, pause, skip, …
- User Generated Content (Web & Mobile)
  - Facebook, Instagram, Yelp, TripAdvisor, Twitter, YouTube, …

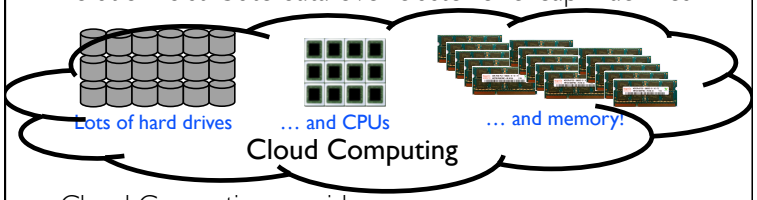
Data Grows Faster than Moore's Law



And Moore's law is ending!!

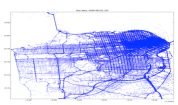
The Big Data Solution: Cloud Computing

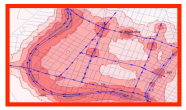
- One machine can not process or even store all the data!
- Solution: distribute data over cluster of cheap machines



- Cloud Computing provides:
  - Illusion of infinite resources
  - Short-term, on-demand resource allocation
  - Can be much less expensive than owning computers
  - Access to latest technologies (SSDs, GPUs, …)

What Can You do with Big Data?



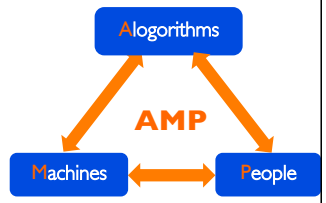
Crowdsourcing + Physical modeling + Sensing + Data Assimilation

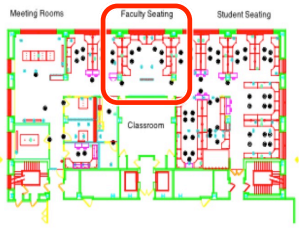
=

http://traffic.berkeley.edu

The Berkeley AMPLab

- January 2011 – 2016
  - 8 faculty
  - > 50 students
  - 3 software engineer team
- Organized for collaboration





3 day retreats (twice a year)

AMPCamp (since 2012)

400+ campers (100s companies)

The Berkeley AMPLab

- Governmental and industrial funding:

Goal: Next generation of open source data analytics stack for industry & academia: Berkeley Data Analytics Stack (BDAS)

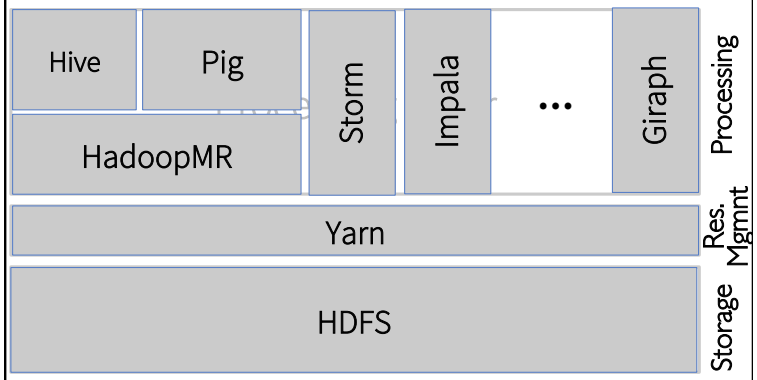
Generic Big Data Stack

Processing Layer

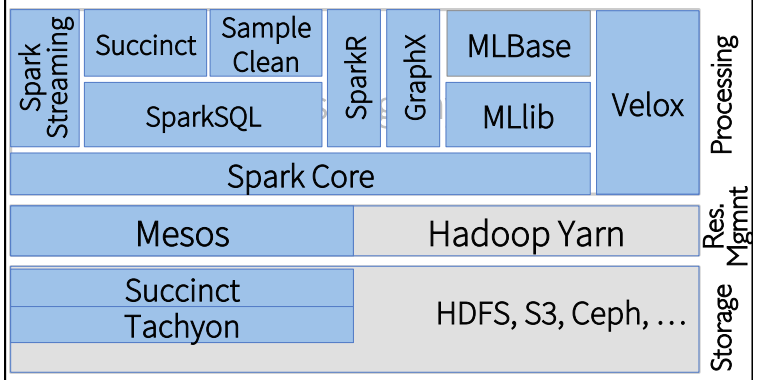
Resource Management Layer

Storage Layer

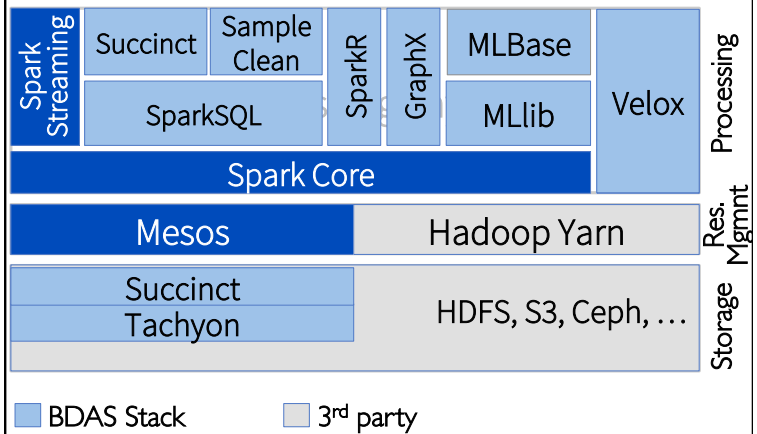
Hadoop Stack



BDAS Stack



Today's Lecture



Summary

- Mesos
- Spark

Summary

- Mesos
- Spark

A Short History

- Started at UC Berkeley in Spring 2009
  - A class project of cs294 (Cloud Computing: Infrastructure, Services, and Applications)
- Open Source: 2010
- Apache Project: 2011
- Today: one of the most popular cluster resource management systems (OS for datacenters)

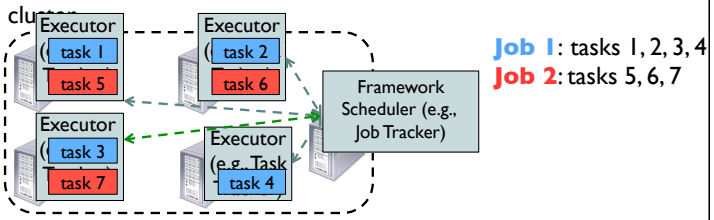
Motivation

- Rapid innovation in cloud computing (aka 2008)

- No single framework optimal for all applications
- Each framework runs on its dedicated cluster or cluster partition

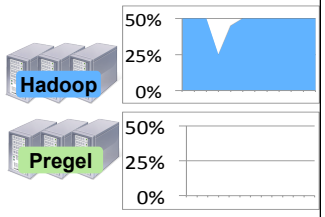
Computation Model: Frameworks

- A *framework* (e.g., Hadoop, MPI) manages one or more *jobs* in a computer cluster
- A *job* consists of one or more *tasks*
- A *task* (e.g., map, reduce) is implemented by one or more processes running on a single machine

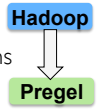


One Framework Per Cluster Challenges

- Inefficient resource usage
  - E.g., Hadoop cannot use available resources from Pregel's cluster
  - No opportunity for stat. multiplexing



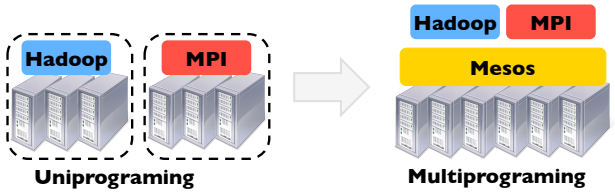
- Hard to share data
  - Copy or access remotely, expensive
- Hard to cooperate
  - E.g., Not easy for Pregel to use graphs generated by Hadoop



Need to run multiple frameworks on same cluster

Solution: Apache Mesos

- Common resource sharing layer
  - abstracts ("virtualizes") resources to frameworks
  - enable diverse frameworks to share cluster



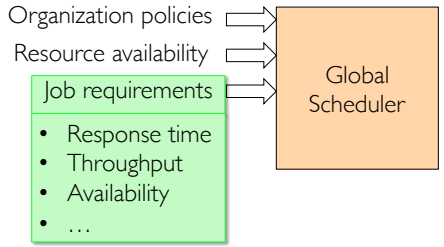
Fine Grained Resource Sharing

- Task granularity both in **time** & **space**
  - Multiplex node/time between tasks belonging to different jobs/frameworks
- Tasks typically short; median ~= 10 sec, minutes
- Why fine grained?
  - Improve data locality
  - Easier to handle node failures

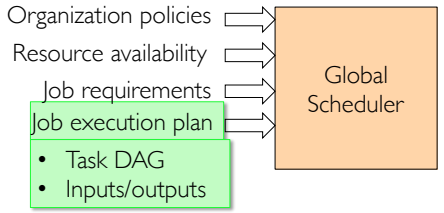
Mesos Goals

- **High utilization** of resources
- **Support diverse frameworks** (existing & future)
- **Scalability** to 10,000's of nodes
- **Reliability** in face of node failures
- Focus of this talk: ***resource management & scheduling***

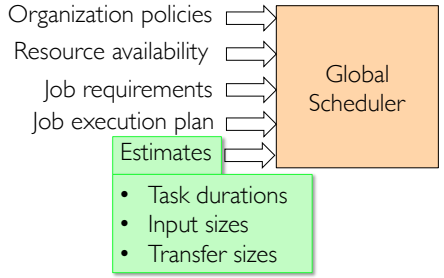
Approach: Global Scheduler



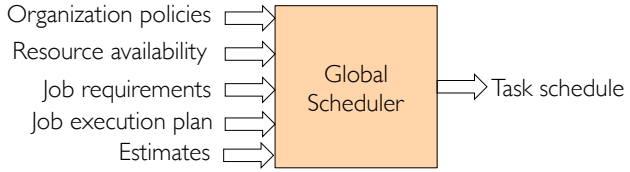
Approach: Global Scheduler



Approach: Global Scheduler

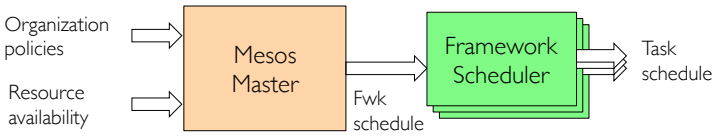


Approach: Global Scheduler



- Advantages: can achieve optimal schedule
- Disadvantages:
  - Complexity → hard to scale and ensure resilience
  - Hard to anticipate future frameworks' requirements
  - Need to refactor existing frameworks

Our Approach: Distributed Scheduler



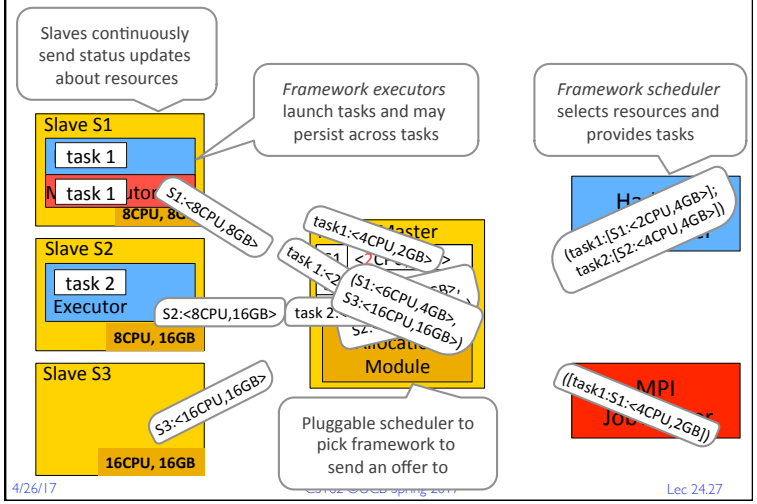
- Advantages:
  - Simple → easier to scale and make resilient
  - Easy to port existing frameworks, support new ones
- Disadvantages:
  - Distributed scheduling decision → not optimal

Resource Offers

- Unit of allocation: ***resource offer***
  - Vector of available resources on a node
  - E.g., node1: <1CPU, 1GB>, node2: <4CPU, 16GB>
- Master sends resource offers to frameworks
- Frameworks select which offers to accept and which tasks to run

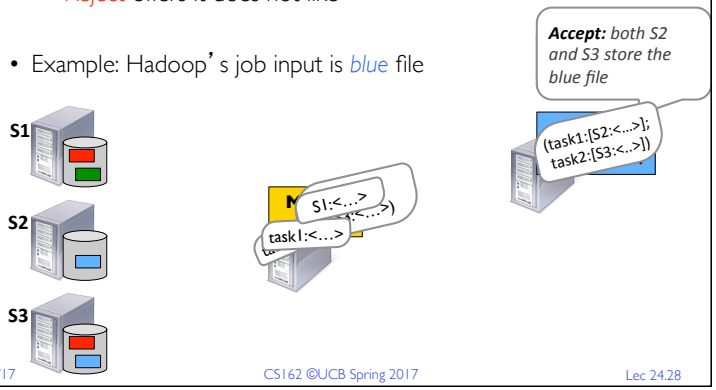
Push task scheduling to frameworks

Mesos Architecture: Example



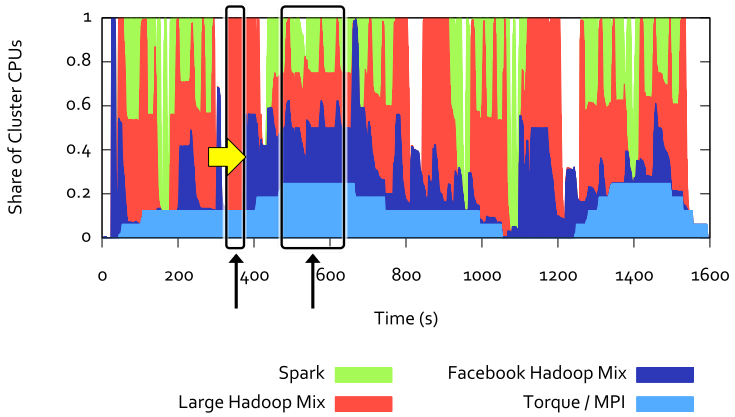
Why does it Work?

- A framework can just wait for an offer that matches its constraints or preferences!
  - Reject offers it does not like
- Example: Hadoop's job input is *blue* file



Dynamic Resource Sharing

- 100 node cluster

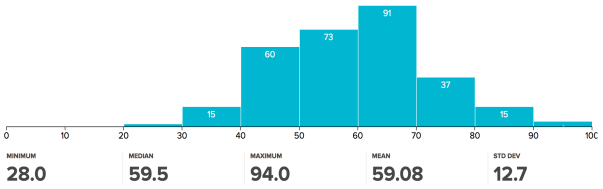


Apache Mesos Today

- Hundreds of of contributors
- Hundreds of deployments in productions
  - E.g., Twitter, GE, Apple
  - Managing 10K node datacenterts!
- Mesosphere, startup to commercialize Apache Spark

MESOSPHERE

Administrivia



- Midterm #3 grades published
  - Regrade request deadline Wednesday 5/3 at midnight
  - Will open regrading tomorrow, Thursday, 4/27
  - *Please only submit regrade requests for grading errors!*
- Project 3
  - Code due Monday 5/1
  - Final report due Wednesday 5/5

BREAK

Summary

- Mesos
- Spark

A Short History

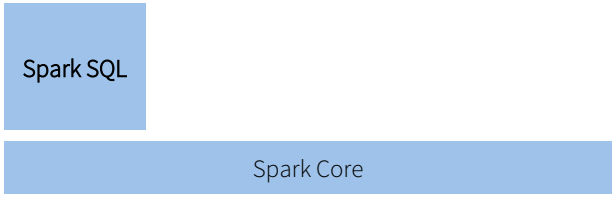
- Started at UC Berkeley in 2009
- Open Source: 2010
- Apache Project: 2013
- Today: most popular big data project

What Is Spark?

- Parallel execution engine for big data processing
- Easy to use: 2-5x less code than Hadoop MR
  - High level API's in Python, Java, and Scala
-
- Fast: up to 100x faster than Hadoop MR
  - Can exploit in-memory when available
  - Low overhead scheduling, optimized engine
- General: support multiple computation models

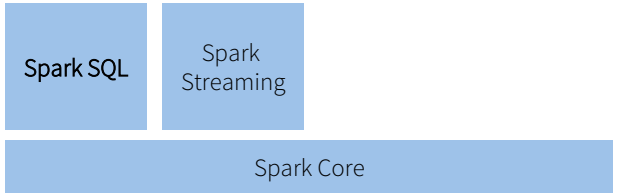
General

- Unifies ***batch, interactive*** comp.



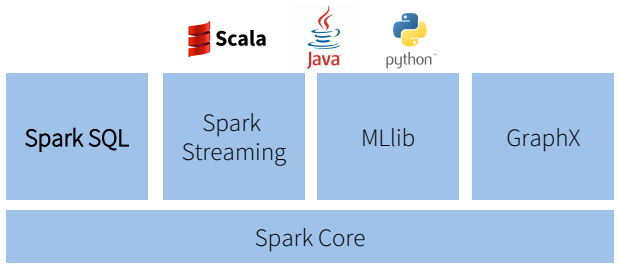
General

- Unifies ***batch, interactive, streaming*** comp.



General

- Unifies ***batch, interactive, streaming*** comp.
- Easy to build sophisticated applications
  - Support iterative, graph-parallel algorithms
  - Powerful APIs in Scala, Python, Java



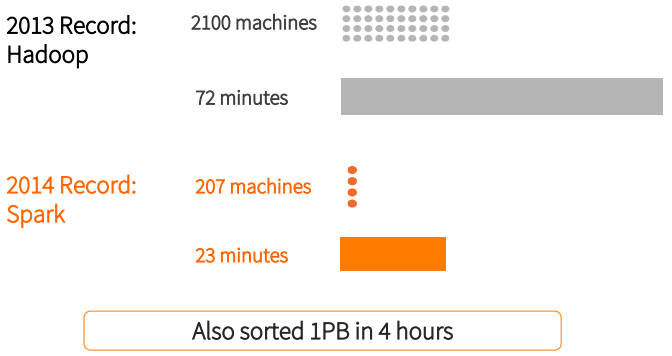
Easy to Write Code

WordCount in 50+ lines of Java MR

```
1 val f = sc.textFile(inputPath)
2 val w = f.flatMap(l => l.split(" ")).map(word => (word, 1)).cache()
3 w.reduceByKey(_ + _).saveAsText(outputPath)
```

WordCount in 3 lines of Spark

Fast: Time to sort 100TB



Source: Daytona GraySort benchmark, sortbenchmark.org

Large-Scale Usage

Largest cluster: 8000 nodes Tencent 腾讯

Largest single job: 1 petabyte 阿里巴巴 Alibaba.com databricks

Top streaming intake: 1 TB/hour HHMI janelia farm research campus

2014 on-disk sort record

RDD: Core Abstraction

Write programs in terms of distributed datasets

and operations on them

- Resilient Distributed Datasets (RDDs)
  - Collections of objects distr. across a cluster, stored in RAM or on Disk
  - Built through parallel transformations
  - Automatically rebuilt on failure

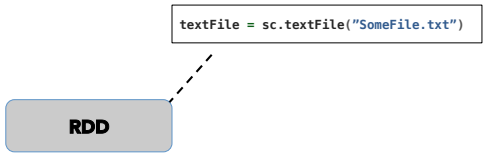
Operations

- Transformations (e.g. map, filter, groupBy)
- Actions (e.g. count, collect, save)

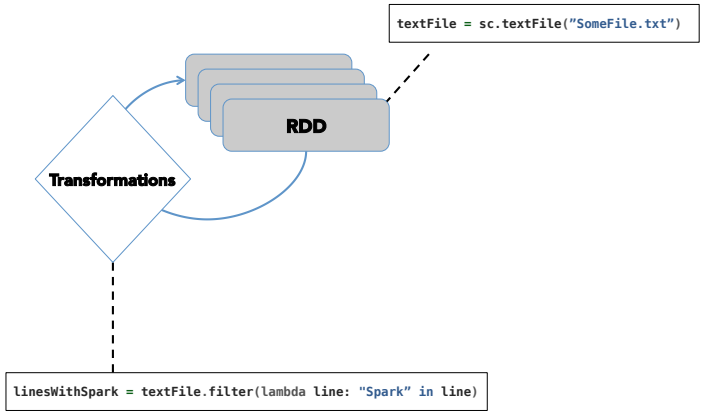
Operations on RDDs

- Transformations f(RDD) => RDD
  - Lazy (not computed immediately)
  - E.g. "map"
- Actions:
  - Triggers computation
  - E.g. "count", "saveAsTextFile"

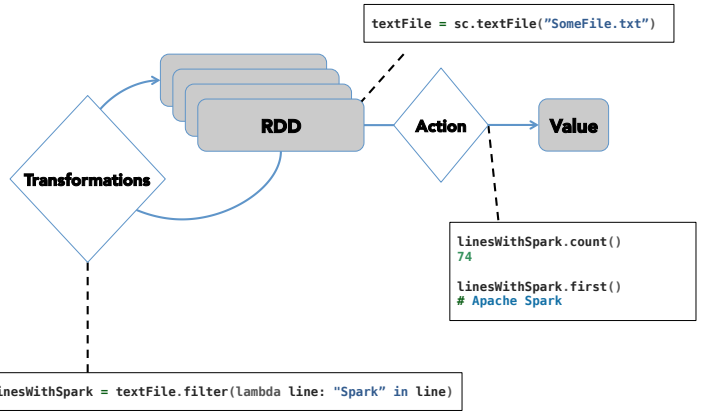
Working With RDDs



Working With RDDs



Working With RDDs

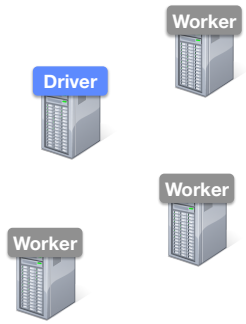


Example: Log Mining

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

Example: Log Mining

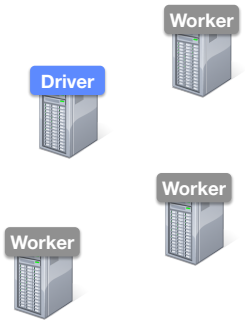
Load error messages from a log into memory, then interactively search for various patterns



Example: Log Mining

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

```
lines = spark.textFile("hdfs://...")
```

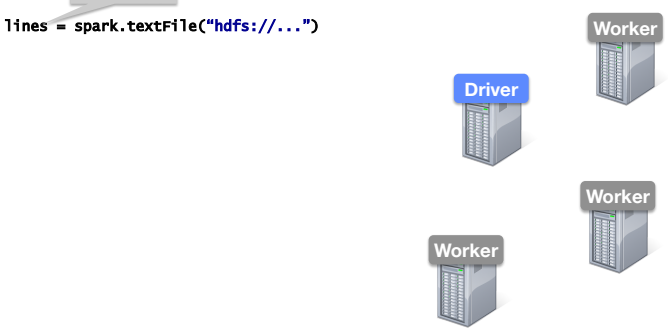


Example: Log Mining

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

Base RDD

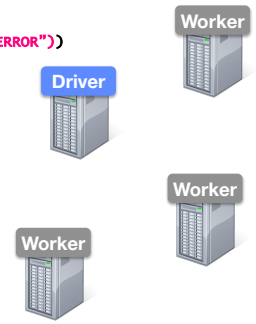
```
lines = spark.textFile("hdfs://...")
```



Example: Log Mining

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

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
```

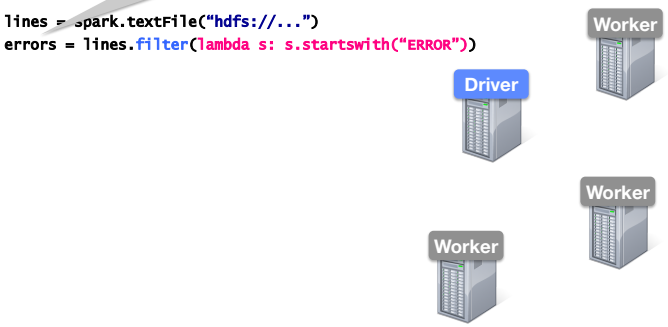


Example: Log Mining

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

Transformed RDD

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
```

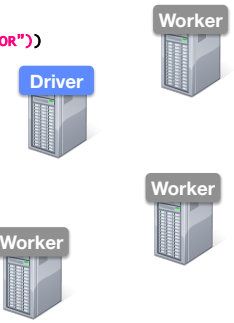


Example: Log Mining

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

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
```

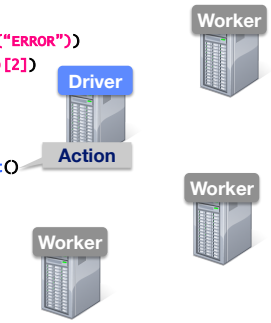


Example: Log Mining

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

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
```

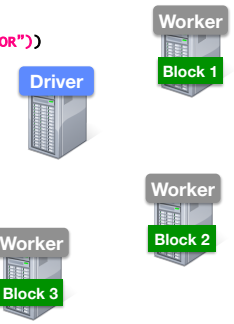


Example: Log Mining

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

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
```

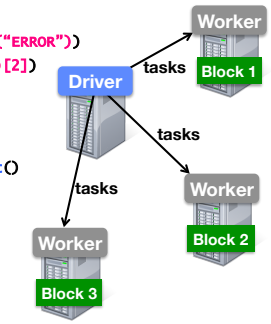


Example: Log Mining

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

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
```

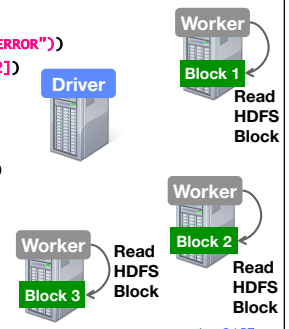


Example: Log Mining

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

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
```

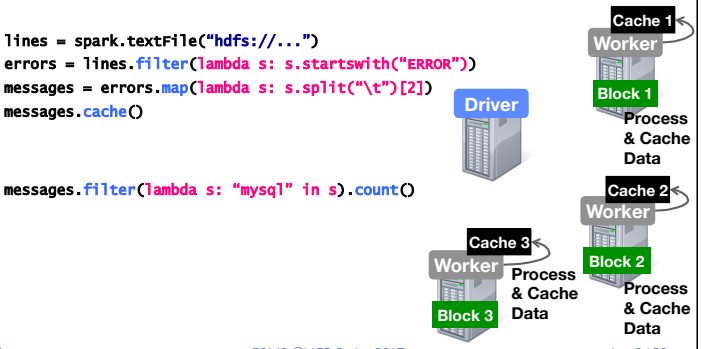


Example: Log Mining

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

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
```

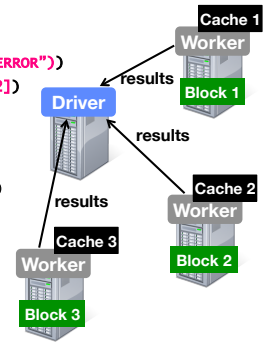


Example: Log Mining

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

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
```

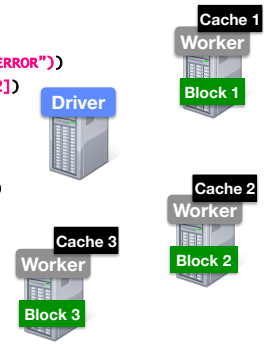


Example: Log Mining

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

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
```

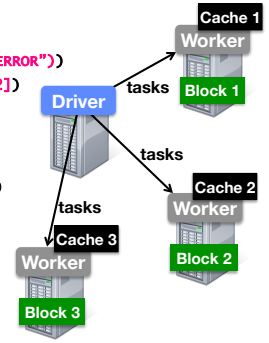


Example: Log Mining

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

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
```

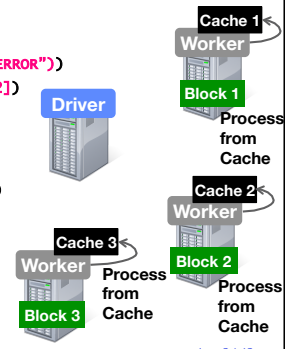


Example: Log Mining

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

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
```

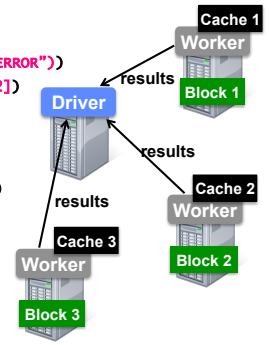


Example: Log Mining

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

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
```

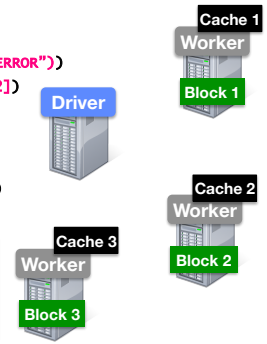


Example: Log Mining

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

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
```



Cache your data ➔ Faster Results
Full-text search of Wikipedia
- 60GB on 20 EC2 machines
- 0.5 sec from mem vs. 20s for on-disk

Language Support

Python

```
lines = sc.textFile(...)
lines.filter(lambda s: "ERROR" in s).count()
```

Scala

```
val lines = sc.textFile(...)
lines.filter(x => x.contains("ERROR")).count()
```

Java

```
JavaRDD<String> lines = sc.textFile(...);
lines.filter(new Function<String, Boolean>() {
  Boolean call(String s) {
    return s.contains("error");
  }
}).count();
```

Standalone Programs
- Python, Scala, & Java

Interactive Shells
- Python & Scala

Performance
- Java & Scala are faster due to static typing
- …but Python is often fine

Expressive API

- `map` `reduce`

Expressive API

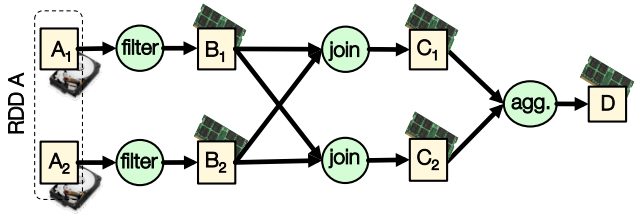
- `map` `reduce` **`sample`**
- `filter` `count` **`take`**
- `groupBy` `fold` **`first`**
- `sort` `reduceByKey` **`partitionBy`**
- `union` `groupByKey` **`mapWith`**
- `join` `cogroup` **`pipe`**
- `leftOuterJoin` `cross` **`save`** **`...`**
- `rightOuterJoin` `zip`

Fault Recovery

RDDs track *lineage* information that can be used to efficiently reconstruct lost partitions

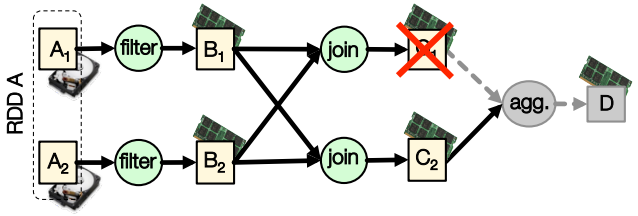
Fault Recovery Example

- Two-partition RDD A={A_1, A_2} stored on disk
  1) filter and cache → RDD B
  2) join→ RDD C
  3) aggregate → RDD D



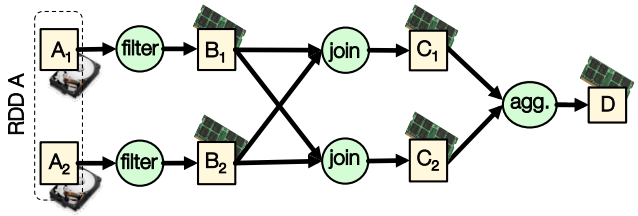
Fault Recovery Example

- C_1 lost due to node failure before reduce finishes

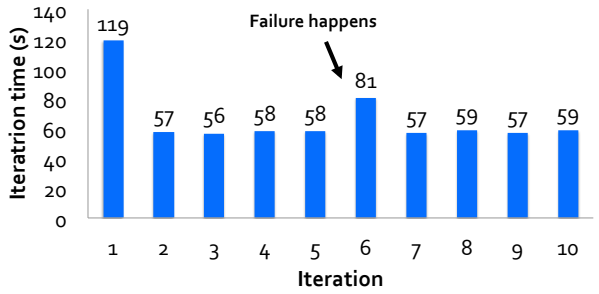


Fault Recovery Example

- C_1 lost due to node failure before reduce finishes
- Reconstruct C_1, eventually, on different node



Fault Recovery Example

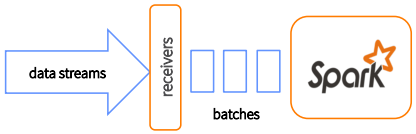


Spark Streaming: Motivation

- Many important apps must process large data streams at second-scale latencies
  - Site statistics, intrusion detection, online ML
- To build and scale these apps users want:
  - **Integration:** with offline analytical stack
  - **Fault-tolerance:** both for crashes and stragglers

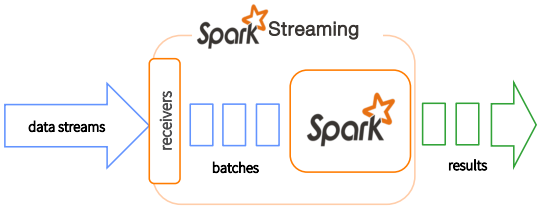
How does it work?

- Data streams are chopped into batches
  - A batch is an RDD holding a few 100s ms worth of data
- Each batch is processed in Spark



How does it work?

- Data streams are chopped into batches
  - A batch is an RDD holding a few 100s ms worth of data
- Each batch is processed in Spark
- Results pushed out in batches



Streaming Word Count

```
val lines = context.socketTextStream("localhost", 9999)
val words = lines.flatMap(_.split(" "))
val wordCounts = words.map(x => (x, 1)).reduceByKey(_ + _)
wordCounts.print()
ssc.start()
```

create DStream from data over socket

split lines into words

count the words

print some counts on screen

start processing the stream

Word Count

```
object NetworkWordCount {
 def main(args: Array[String]) {
  val sparkConf = new SparkConf().setAppName("NetworkWordCount")
  val context = new StreamingContext(sparkConf, Seconds(1))

  val lines = context.socketTextStream("localhost", 9999)
  val words = lines.flatMap(_.split(" "))
  val wordCounts = words.map(x => (x, 1)).reduceByKey(_ + _)

  wordCounts.print()
  ssc.start()
  ssc.awaitTermination()
 }
}
```


Word Count

Spark Streaming

```
object NetworkWordCount {
 def main(args: Array[String]) {
  val sparkConf = new SparkConf().setAppName("NetworkWordCount")
  val context = new StreamingContext(sparkConf, Seconds(1))

  val lines = context.socketTextStream("localhost", 9999)
  val words = lines.flatMap(_.split(" "))
  val wordCounts = words.map(x => (x, 1)).reduceByKey(_ + _)

  wordCounts.print()
  ssc.start()
  ssc.awaitTermination()
 }
}
```

Storm

Benefits for Users

- High performance data sharing
  - Data sharing is the bottleneck in many environments
  - RDD's provide in-place sharing through memory
- Applications can compose models
  - Run a SQL query and then PageRank the results
  - ETL your data and then run graph/ML on it
- Benefit from investment in shared functionality
  - E.g. re-usable components (shell) and performance optimizations

Many Recent Development

- RDDs → Dataframes, and DatSets
  - Distributed collection of data grouped into named columns (i.e. RDD with schema)
  - Think about R and Python Pandas dataframes
- Project Tungsten
  - Fully managed memory: allowed by Dataframe/DatSets schema
  - Code generation
  - Vectorized processing
- Structured streams
  - Adding Dataframe API and optimizations to streaming

Apache Spark Today

- > 1,500 contributors
- > 300K committers world wide
- > 100K students trained

- 1,000s deployments in productions
  - Virtually every large enterprise
  - Available in all clouds (e.g., AWS, Google Compute Engine, MS Azure)
  - Distributed by IBM, Cloudera, Hortonworks, Oracle

- Databricks, startup to commercialize Apache Spark

Summary

- Server → Datacenter
- OS → Datacenter OS (e.g., Apache Mesos)
- Applications → Big data / ML applications (e.g., Apache Spark)

- AMPLab
  - Massive success in industry,…
  - and, academia: faculty at MIT, Stanford, Cornell, etc

- New lab starting: RISELab

RISELab
(Real-time Intelligent Secure Execution)

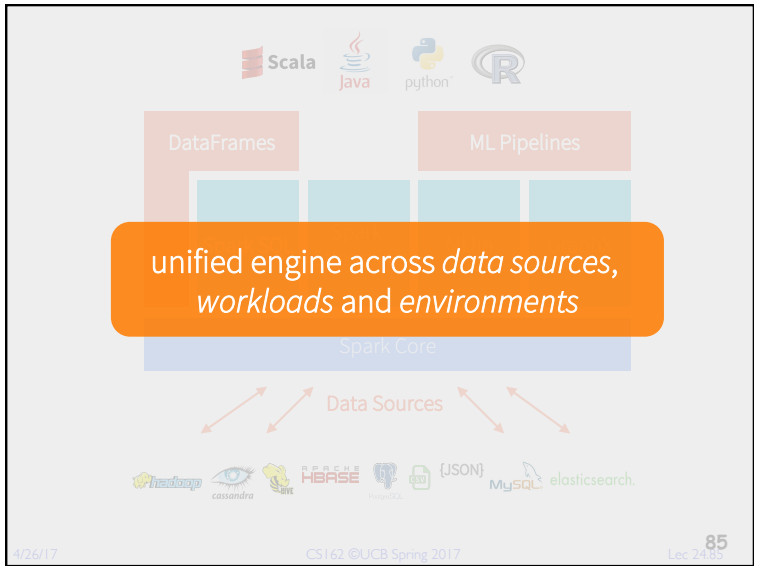
RISELab

From live data to real-time decisions

↑

AMPLab

From batch data to advanced analytics


Scala
Java
python
DataFrames
ML Pipelines
unified engine across *data sources*, *workloads* and *environments*
Spark Core
Data Sources
{JSON}
MySQL
elasticsearch.
4/26/17
CS162 ©UCB Spring 2017
85
Lec 24.85