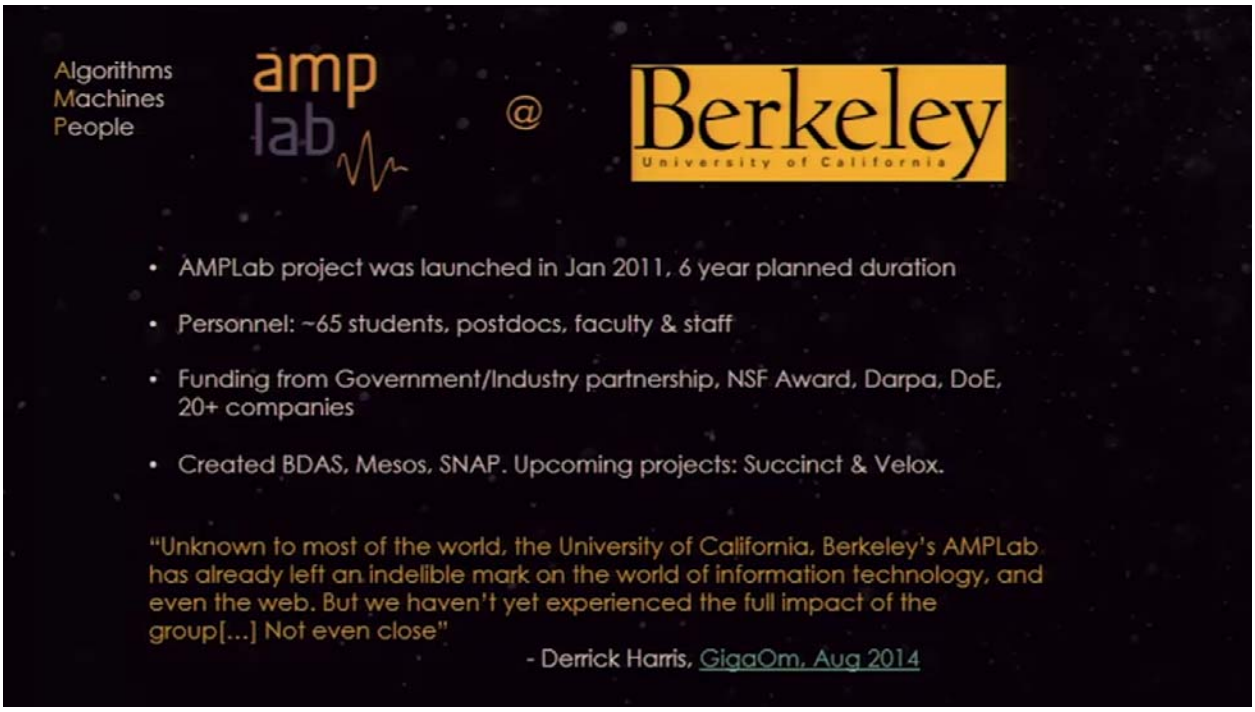
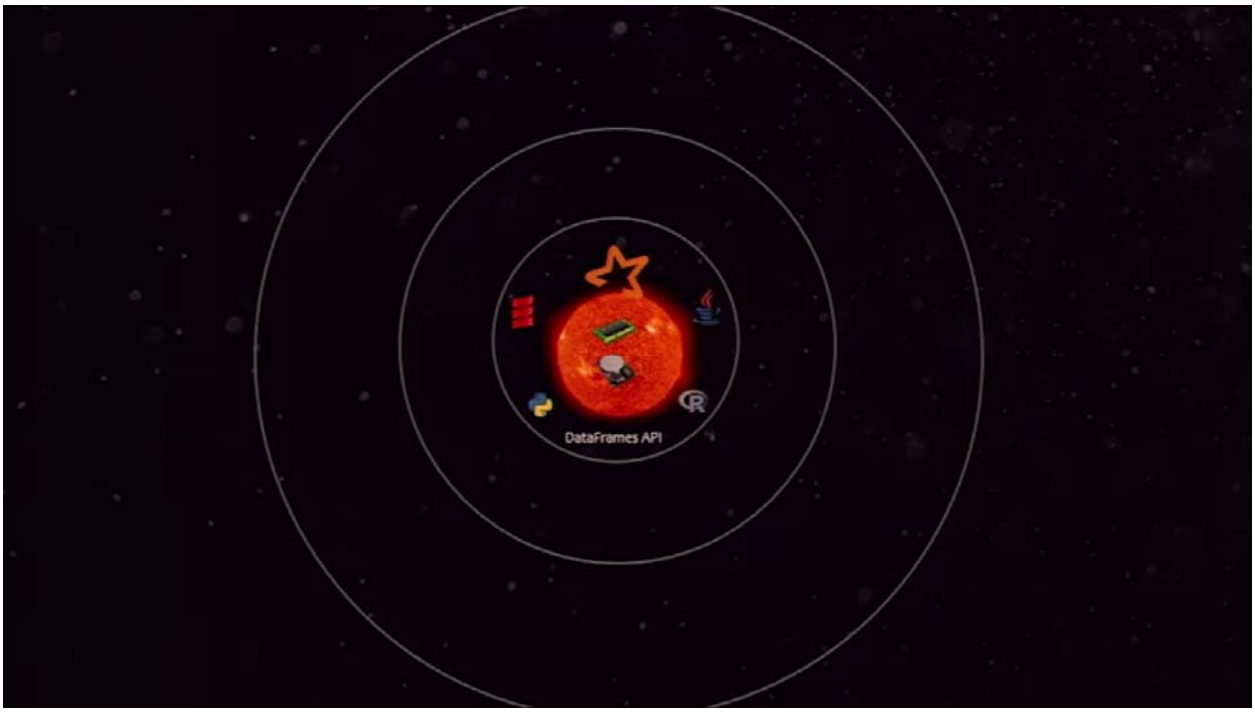
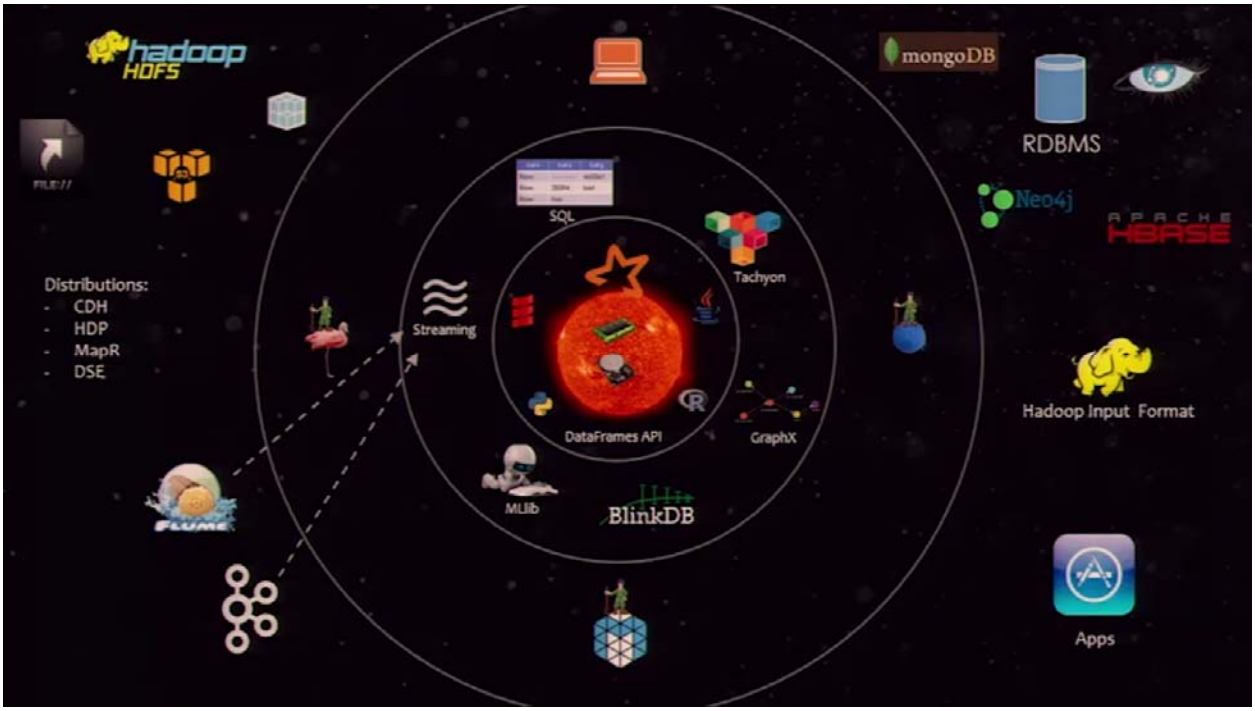
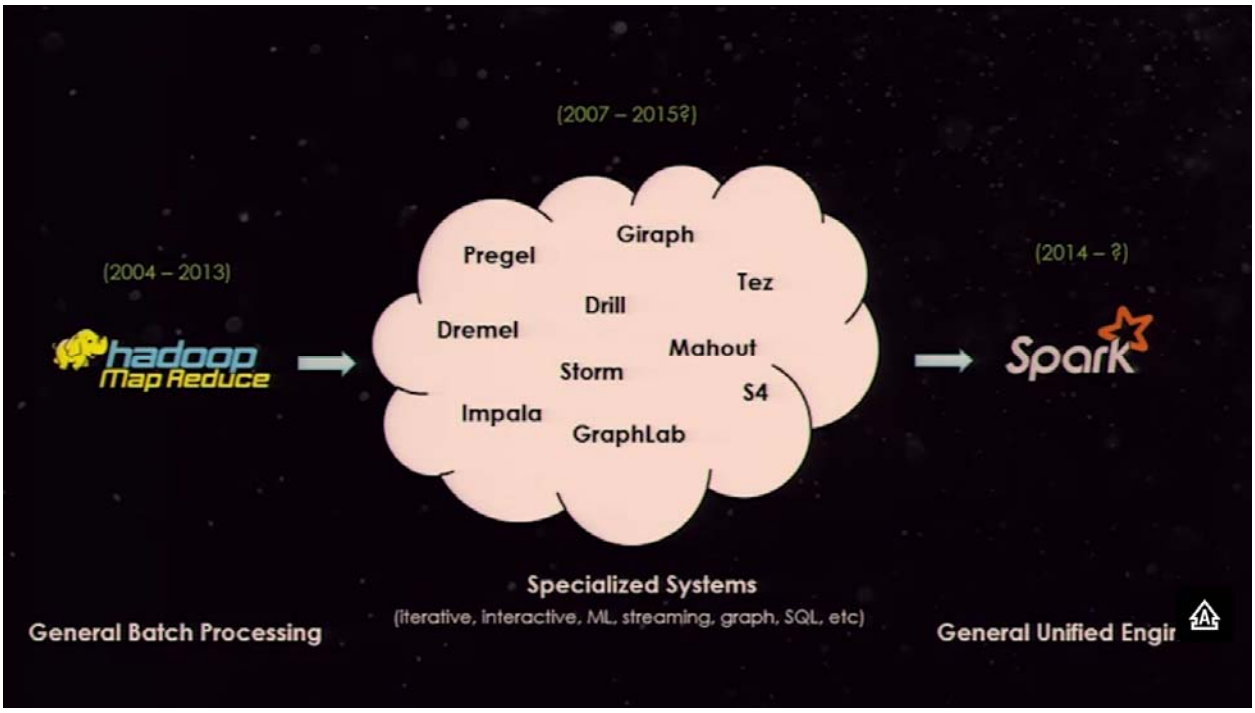
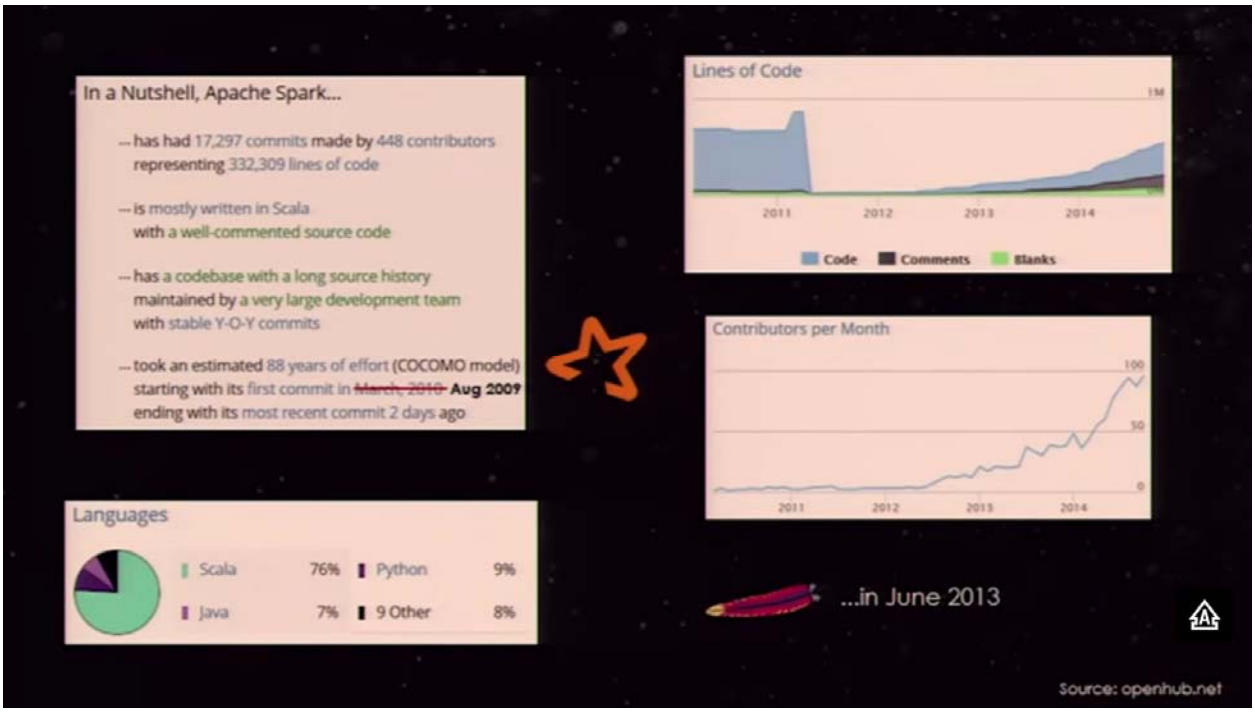
https://www.youtube.com/watch?v=7ooZ4S7Ay6Y



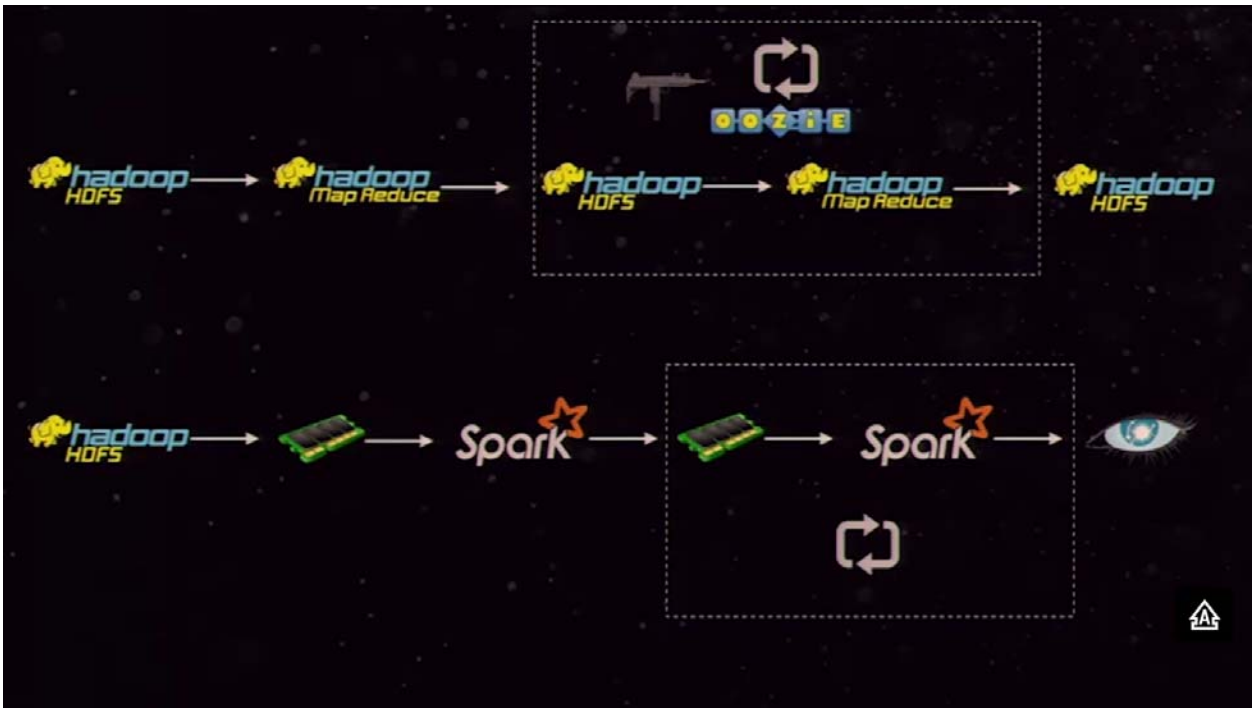

DataFrames API

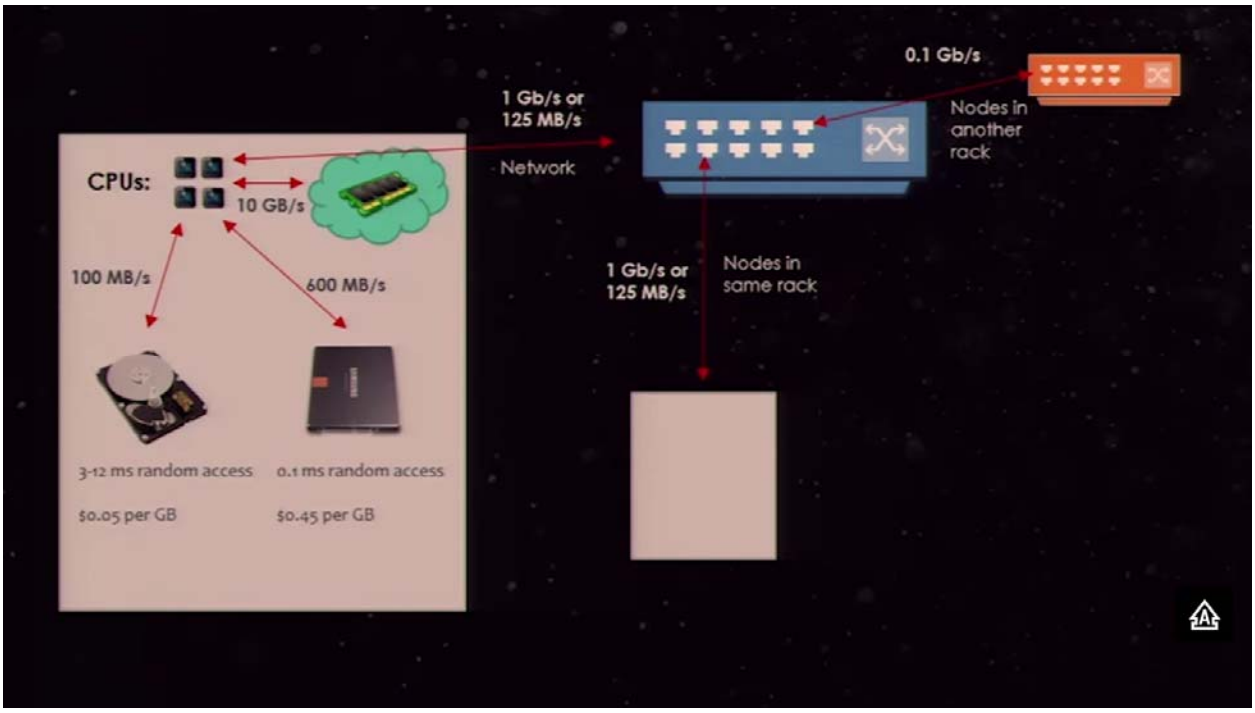

hadoop
HDFS
FILE://
mongoDB
RDBMS
Neo4j
APACHE
HBASE
SQL
Tachyon
Distributions:
- CDH
- HDP
- MapR
- DSE
Streaming
DataFrames API
GraphX
MLlib
BlinkDB
FLUME
Hadoop Input Format
Apps


(2007 – 2015?)
(2004 – 2013)
(2014 – ?)
hadoop
Map Reduce
Pregel
Giraph
Tez
Drill
Dremel
Mahout
Storm
S4
Impala
GraphLab
Spark
Specialized Systems
(iterative, interactive, ML, streaming, graph, SQL, etc)
General Batch Processing
General Unified Engi


In a Nutshell, Apache Spark...
-- has had 17,297 commits made by 448 contributors representing 332,309 lines of code
-- is mostly written in Scala with a well-commented source code
-- has a codebase with a long source history maintained by a very large development team with stable Y-O-Y commits
-- took an estimated 88 years of effort (COCOMO model) starting with its first commit in ~~March, 2010~~ Aug 2009 ending with its most recent commit 2 days ago
Lines of Code
1M
2011
2012
2013
2014
Code
Comments
Blanks
Contributors per Month
100
50
0
2011
2012
2013
2014
Languages
Scala 76%
Python 9%
Java 7%
9 Other 8%
...in June 2013
Source: openhub.net


DISTRIBUTORS
databricks
Hortonworks
MAPR
cloudera
IBM
Pivotal
ORACLE
DATASTAX
SAP
guavus
bluedata
STRATIO
APPLICATIONS
+ableau
MicroStrategy
Qlik
elasticsearch.
pentaho
talend
tresata
TRIFACTA
SKYTREE
Alpine
atscale
looker
technicolor
FAiMDATA
ADATAO
DiYOTTA
ZOOMDATA
platfora
APERVI
NUBE
Atigeo
ZALONI
Typesafe


hadoop HDFS
hadoop Map Reduce
OOZIE
hadoop HDFS
hadoop Map Reduce
hadoop HDFS
hadoop HDFS
Spark
Spark



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 overhead 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 §5.3.

"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."

"Best Paper Award and Honorable Mention for Community Award"
- NSDI 2012

- Cited 392 times!

April 2012

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

Spark BLINKDB

BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data

Sameer Agarwal*, Barzan Mozafari*, Aurojit Panda*, Henry Milner*, Samuel Madden*, Ion Stoica*†

†*University of California, Berkeley* **Massachusetts Institute of Technology* *†Conviva Inc.*
{sameerag, apanda, henrym, istoica}@cs.berkeley.edu, {barzan, madden}@csail.mit.edu

Abstract

In this paper, we present BlinkDB, a massively parallel, approximate query engine for running interactive SQL queries on large volumes of data. BlinkDB allows users to trade-off query accuracy for response time, enabling interactive queries over massive data by running queries on data samples and presenting results annotated with meaningful error bars. To achieve this, BlinkDB uses two key ideas: (1) an adaptive optimization framework that builds and maintains a set of multi-dimensional stratified samples from original data over time, and (2) a dynamic sample selection strategy that selects an appropriately sized sample based on a query's accuracy or response time requirements. We evaluate BlinkDB against the well-known TPC-H benchmarks and a real-world analytic workload derived from Conviva Inc., a company that manages video distribution over the Internet. Our experiments on a 100 node cluster show that BlinkDB can answer queries on up to 17 TBs of data in less than 2 seconds (over 200× faster than Hive), within an error of 2–10%.

1. Introduction

Modern data analytics applications involve computing aggregates over a large number of records to roll-up web clicks, crossing of large amounts of data by trading result accuracy for response time and space. These techniques include sampling [10, 14], sketches [12], and on-line aggregation [15]. To illustrate the utility of such techniques, consider the following simple query that computes the average SessionTime over all users originating in New York:

```
SELECT AVG(SessionTime)
FROM Sessions
WHERE City = 'New York'
```

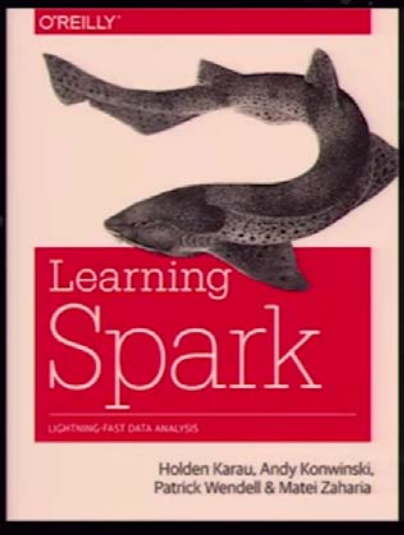
Suppose the Sessions table contains 100 million tuples for New York, and cannot fit in memory. In that case, the above query may take a long time to execute, since disk reads are expensive, and such a query would need multiple disk accesses to stream through all the tuples. Suppose we instead executed the same query on a sample containing only 10,000 New York tuples, such that the entire sample fits in memory. This would be orders of magnitude faster, while still providing an approximate result within a few percent of the actual value, an accuracy good enough for many practical purposes. Using sampling theory we could even provide confidence bounds on the accuracy of the answer [16].

Previously described approximation techniques make different trade-offs between efficiency and the generality of the

```
SELECT avg(sessionTime)
FROM Table
WHERE city='San Francisco'
WITHIN 2 SECONDS
```

```
SELECT avg(sessionTime)
FROM Table
WHERE city='San Francisco'
ERROR 0.1 CONFIDENCE 95.0%
```

https://www.cs.berkeley.edu/~sameerag/blinkdb_eurosys13.pdf

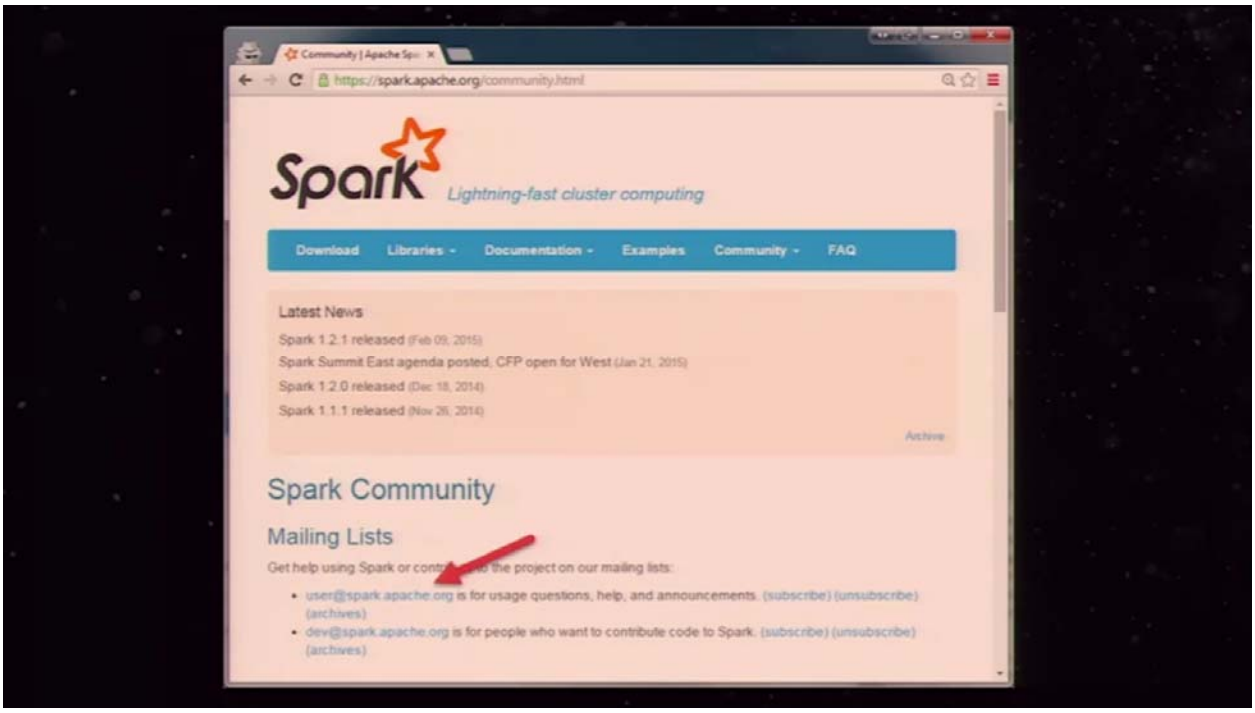


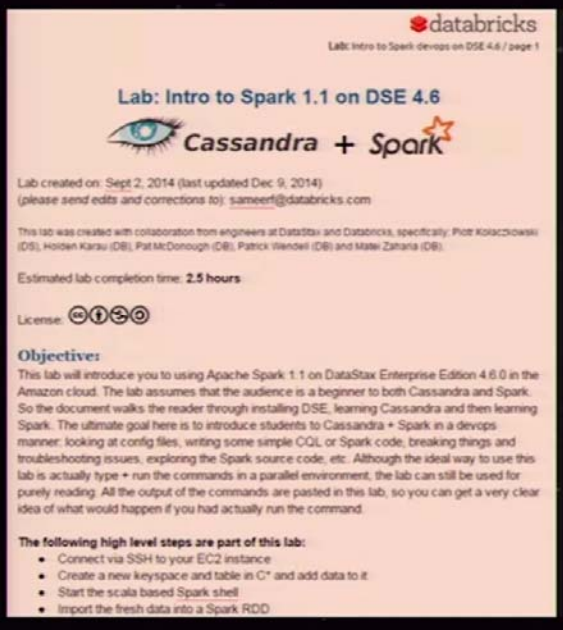
http://shop.oreilly.com/product/0636920028512.do

eBook: $33.99 PDF, ePub, Mobi, DAISY
Print: $39.99 Shipping now!

$30 @ Amazon:

http://www.amazon.com/Learning-Spark-Lightning-Fast-Data-Analysis/dp/1449358624




databricks

Lab: Intro to Spark devops on DSE 4.6 / page 1

Lab: Intro to Spark 1.1 on DSE 4.6

Cassandra + Spark

Lab created on: Sept 2, 2014 (last updated Dec 9, 2014)
(please send edits and corrections to): sameerf@databricks.com

This lab was created with collaboration from engineers at DataStax and Databricks, specifically: Piotr Kolaczkowski (DS), Holden Karau (DB), Pat McDonough (DB), Patrick Wendell (DB) and Matei Zaharia (DB).

Estimated lab completion time: **2.5 hours**

License:

Objective:

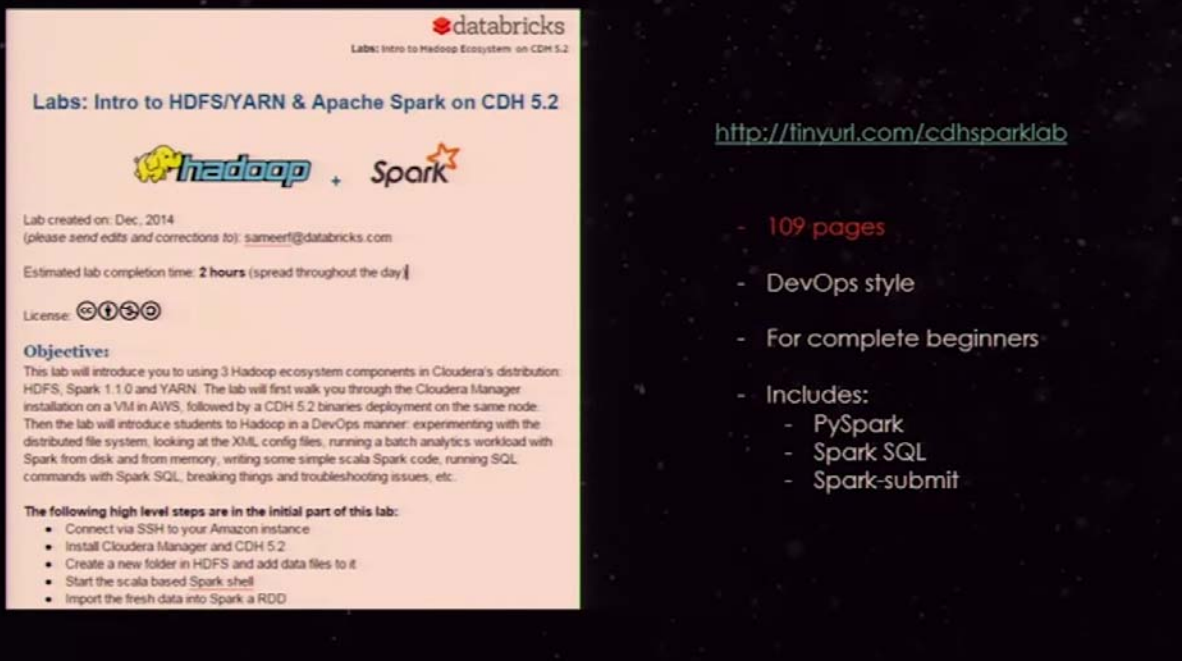
This lab will introduce you to using Apache Spark 1.1 on DataStax Enterprise Edition 4.6.0 in the Amazon cloud. The lab assumes that the audience is a beginner to both Cassandra and Spark. So the document walks the reader through installing DSE, learning Cassandra and then learning Spark. The ultimate goal here is to introduce students to Cassandra + Spark in a devops manner: looking at config files, writing some simple CQL or Spark code, breaking things and troubleshooting issues, exploring the Spark source code, etc. Although the ideal way to use this lab is actually type + run the commands in a parallel environment, the lab can still be used for purely reading. All the output of the commands are pasted in this lab, so you can get a very clear idea of what would happen if you had actually run the command.

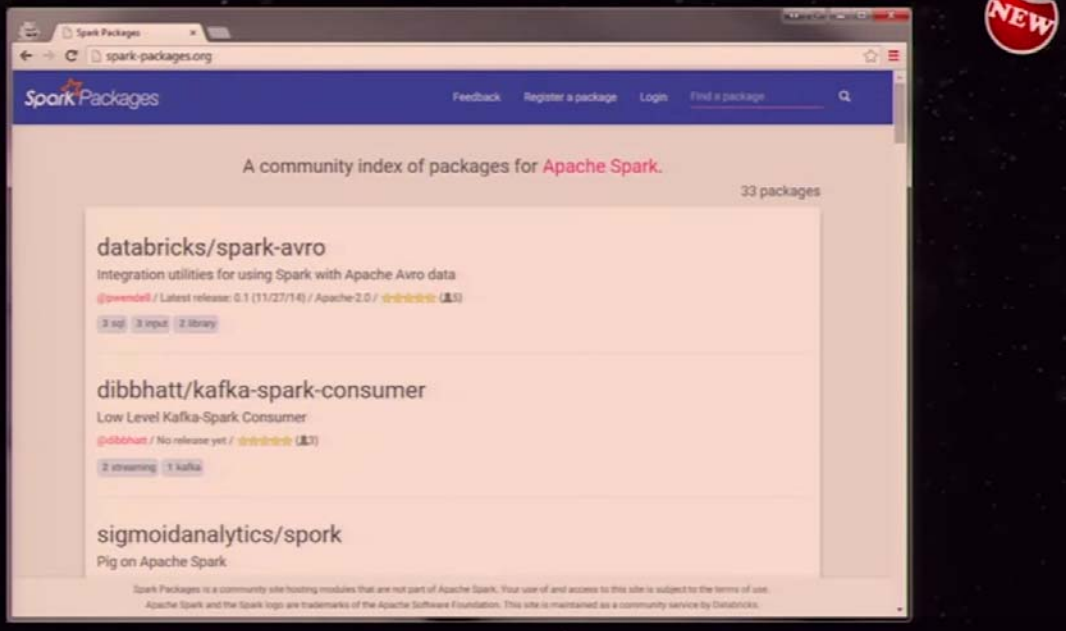
The following high level steps are part of this lab:

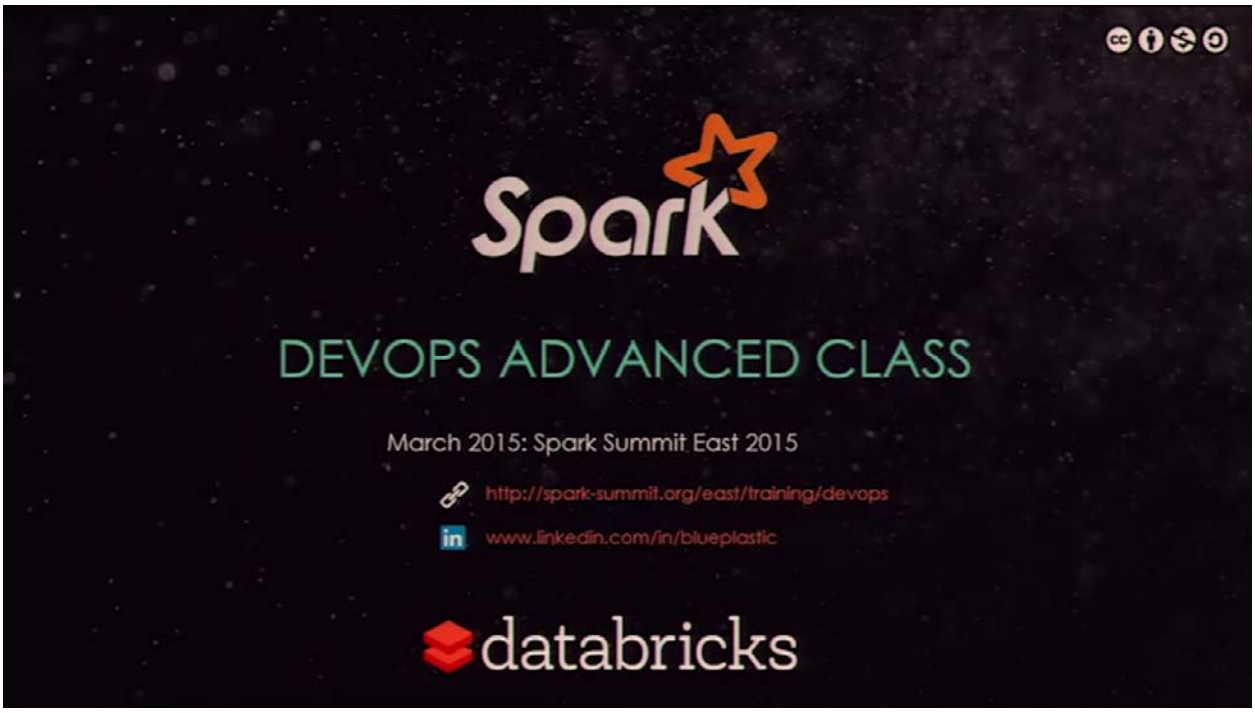
- Connect via SSH to your EC2 instance
- Create a new keyspace and table in C* and add data to it
- Start the scala based Spark shell
- Import the fresh data into a Spark RDD

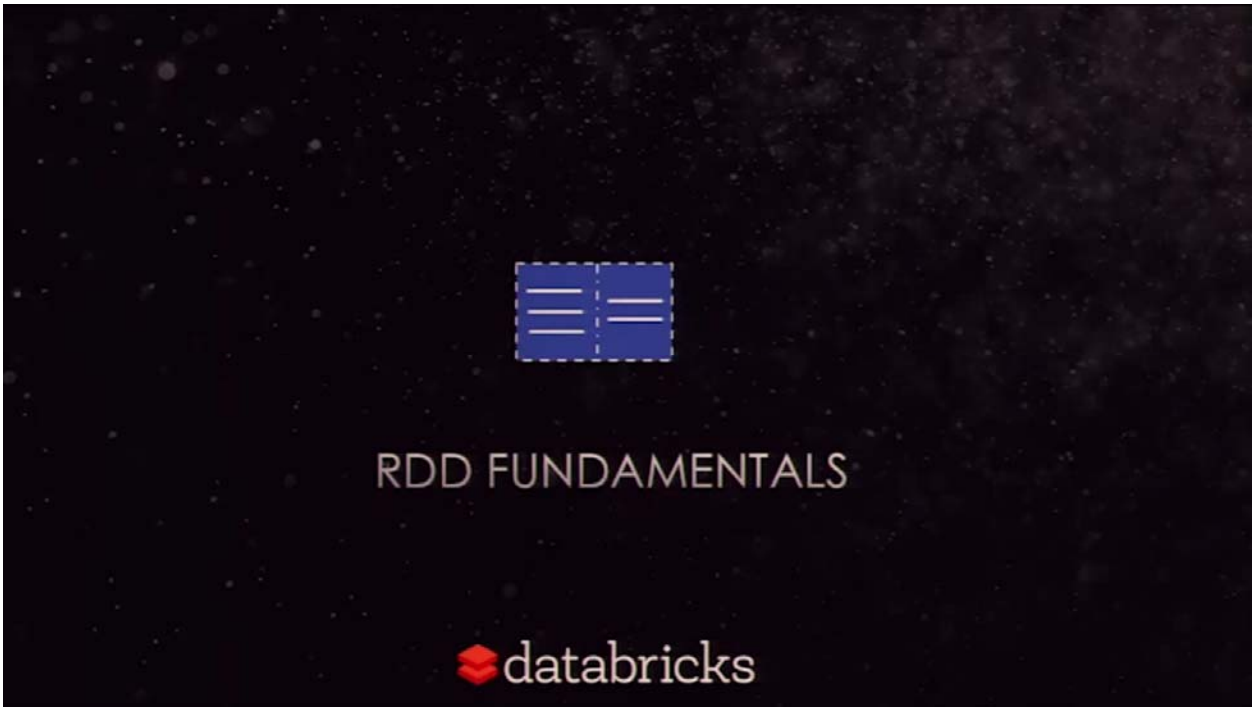
http://tinyurl.com/dsesparklab

- 102 pages
- DevOps style
- For complete beginners
- Includes:
  - Spark Streaming
  - Dangers of GroupByKey vs. ReduceByKey

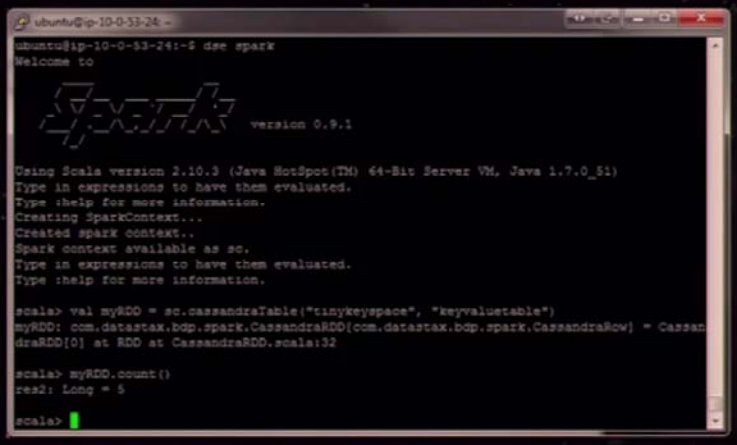

databricks
Labs: Intro to Hadoop Ecosystem on CDH 5.2
Labs: Intro to HDFS/YARN & Apache Spark on CDH 5.2
hadoop + Spark
Lab created on: Dec., 2014
(please send edits and corrections to): sameerf@databricks.com
Estimated lab completion time: 2 hours (spread throughout the day)
License:
Objective:
This lab will introduce you to using 3 Hadoop ecosystem components in Cloudera's distribution: HDFS, Spark 1.1.0 and YARN. The lab will first walk you through the Cloudera Manager installation on a VM in AWS, followed by a CDH 5.2 binaries deployment on the same node. Then the lab will introduce students to Hadoop in a DevOps manner: experimenting with the distributed file system, looking at the XML config files, running a batch analytics workload with Spark from disk and from memory, writing some simple scala Spark code, running SQL commands with Spark SQL, breaking things and troubleshooting issues, etc.
The following high level steps are in the initial part of this lab:
Connect via SSH to your Amazon instance
Install Cloudera Manager and CDH 5.2
Create a new folder in HDFS and add data files to it
Start the scala based Spark shell
Import the fresh data into Spark a RDD
http://tinyurl.com/cdhsparklab
109 pages
DevOps style
For complete beginners
Includes:
PySpark
Spark SQL
Spark-submit


NEW
spark-packages.org
Spark Packages
Feedback
Register a package
Login
Find a package
A community index of packages for Apache Spark.
33 packages
databricks/spark-avro
Integration utilities for using Spark with Apache Avro data
dibbhatt/kafka-spark-consumer
Low Level Kafka-Spark Consumer
sigmoidanalytics/spork
Pig on Apache Spark

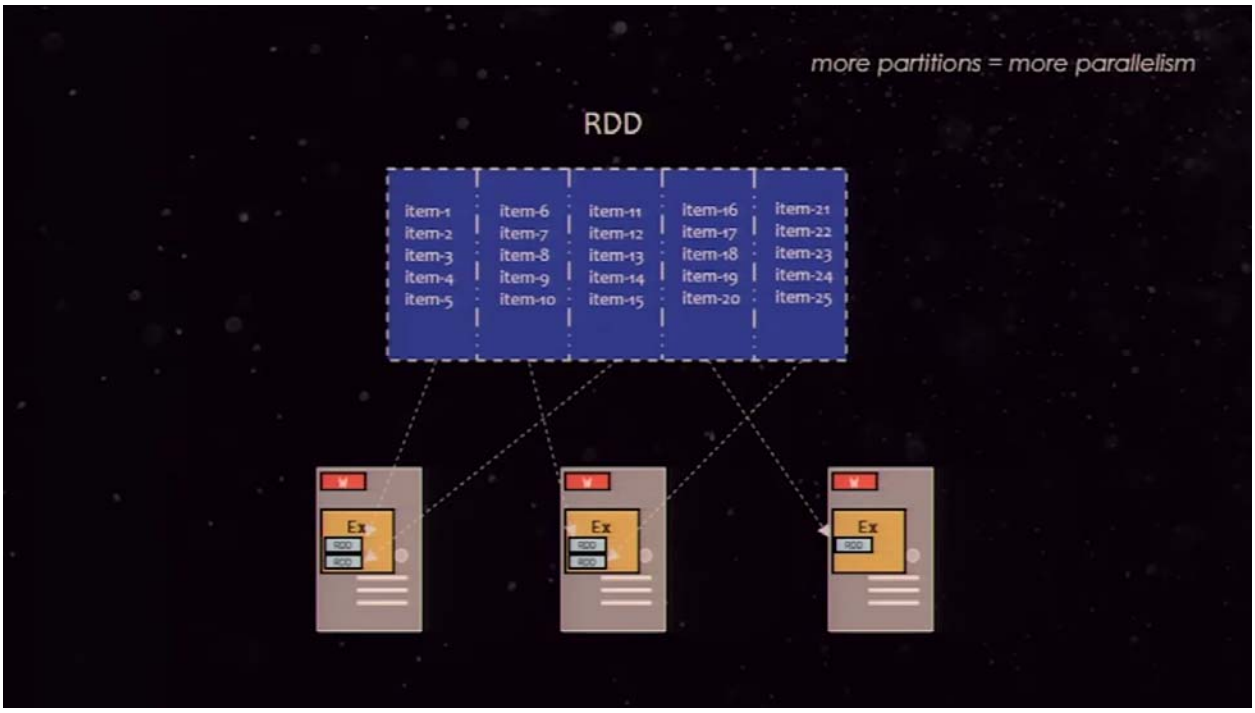

Spark
DEVOPS ADVANCED CLASS
March 2015: Spark Summit East 2015
http://spark-summit.org/east/training/devops
www.linkedin.com/in/blueplastic
databricks

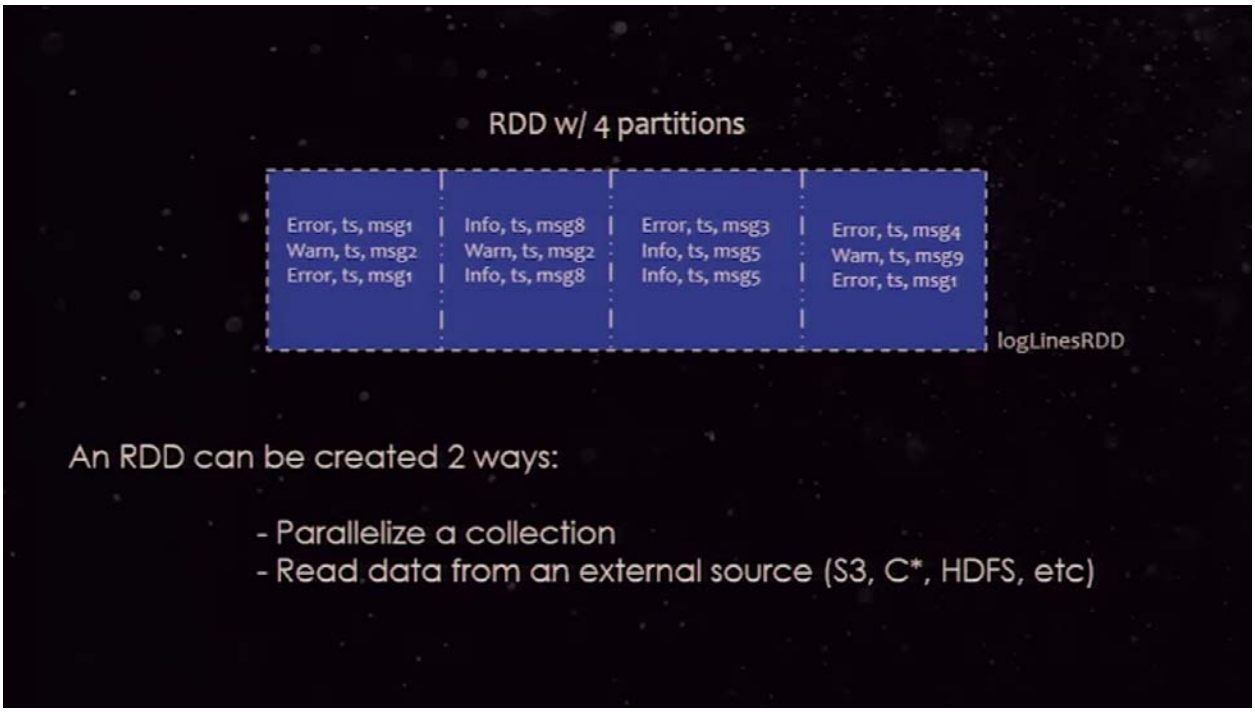

RDD FUNDAMENTALS
databricks

INTERACTIVE SHELL



(Scala & Python only)


more partitions = more parallelism
RDD
item-1
item-2
item-3
item-4
item-5
item-6
item-7
item-8
item-9
item-10
item-11
item-12
item-13
item-14
item-15
item-16
item-17
item-18
item-19
item-20
item-21
item-22
item-23
item-24
item-25
W
Ex
RDD
RDD


RDD w/ 4 partitions
Error, ts, msg1
Warn, ts, msg2
Error, ts, msg1
Info, ts, msg8
Warn, ts, msg2
Info, ts, msg8
Error, ts, msg3
Info, ts, msg5
Info, ts, msg5
Error, ts, msg4
Warn, ts, msg9
Error, ts, msg1
logLinesRDD
An RDD can be created 2 ways:
- Parallelize a collection
- Read data from an external source (S3, C*, HDFS, etc)

PARALLELIZE

```python
# Parallelize in Python
wordsRDD = sc.parallelize(["fish", "cats", "dogs"])
```

```scala
// Parallelize in Scala
val wordsRDD= sc.parallelize(List("fish", "cats", "dogs"))
```

```java
// Parallelize in Java
JavaRDD<String> wordsRDD = sc.parallelize(Arrays.asList("fish", "cats", "dogs"));
```

- Take an existing in-memory collection and pass it to SparkContext's parallelize method
- Not generally used outside of prototyping and testing since it requires entire dataset in memory on one machine

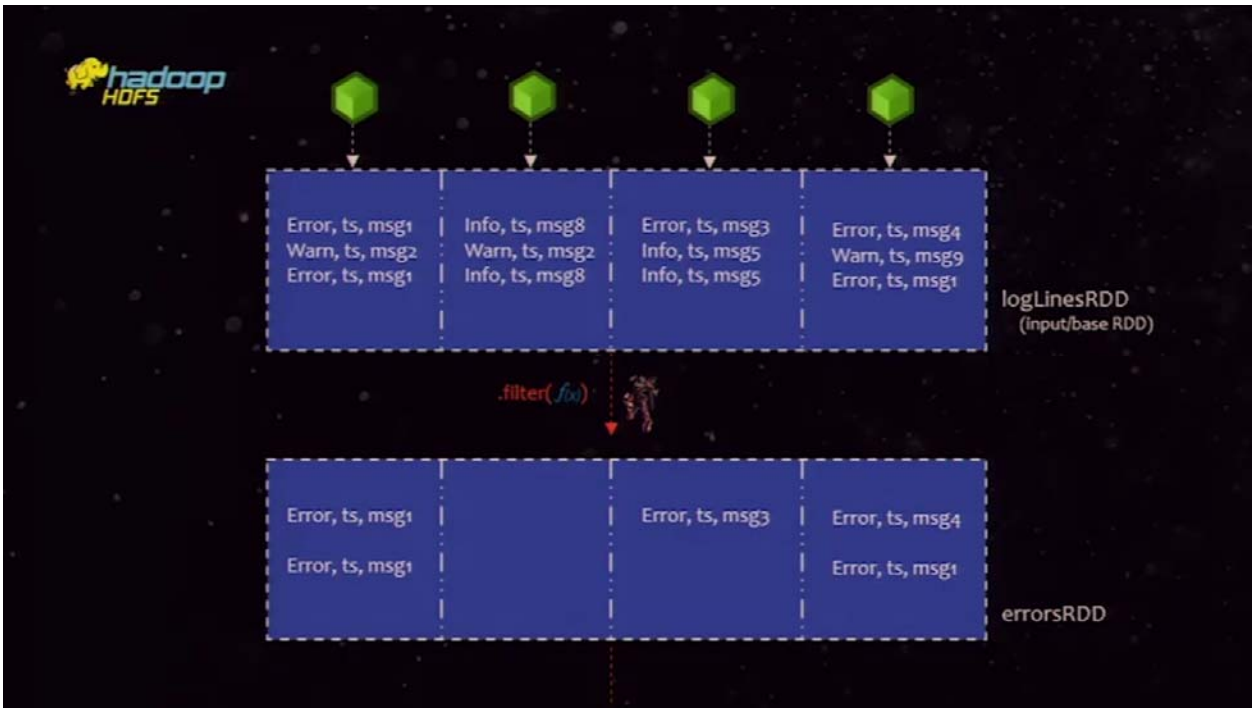
READ FROM TEXT FILE

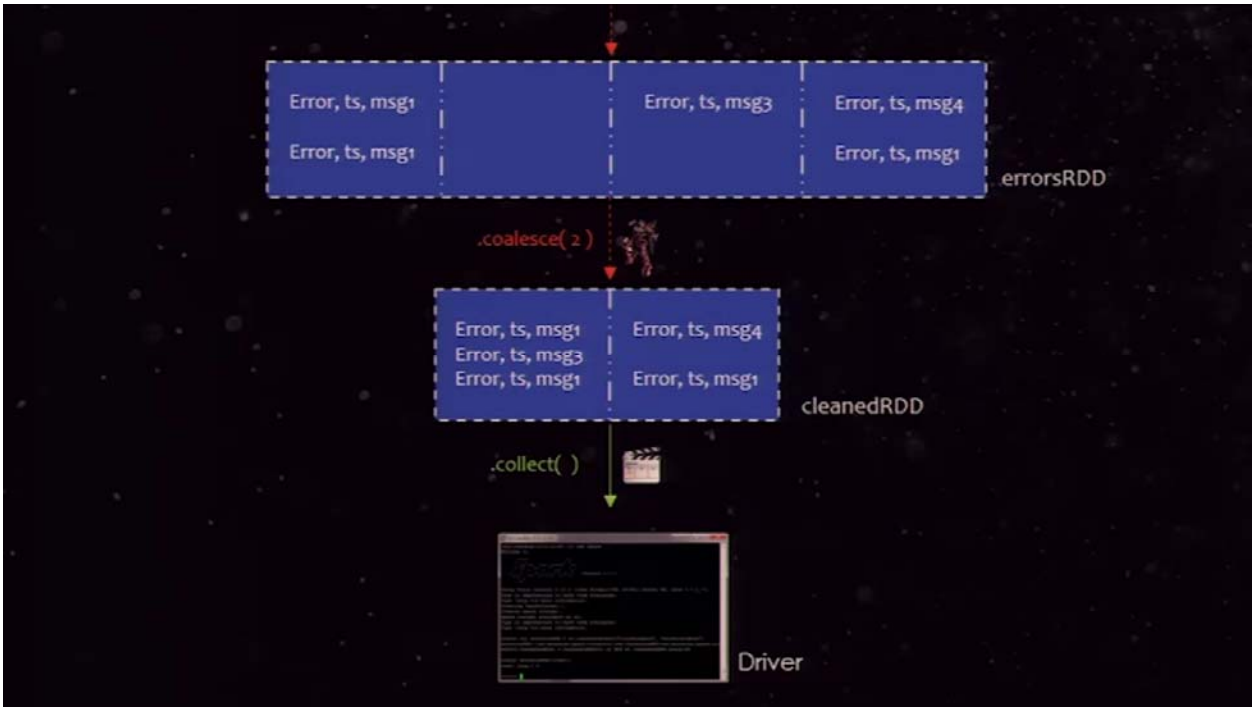
```python
# Read a local txt file in Python
linesRDD = sc.textFile("/path/to/README.md")
```

```scala
// Read a local txt file in Scala
val linesRDD = sc.textFile("/path/to/README.md")
```

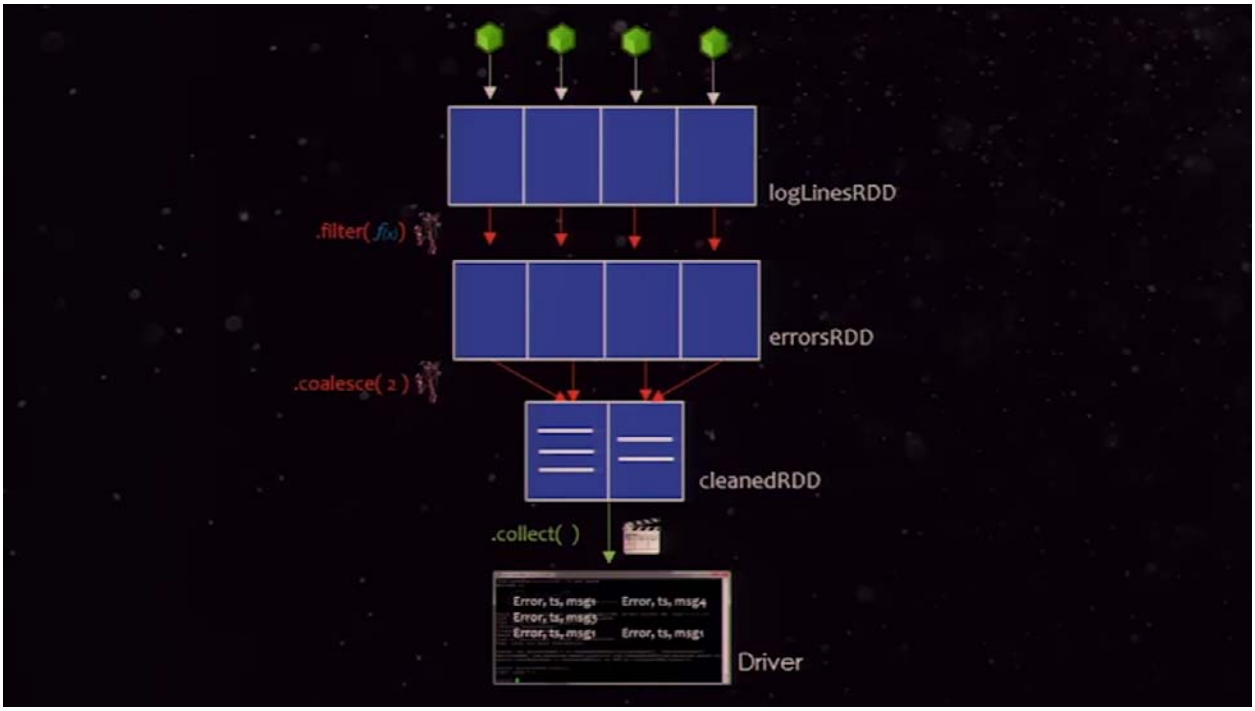
```java
// Read a local txt file in Java
JavaRDD<String> lines = sc.textFile("/path/to/README.md");
```

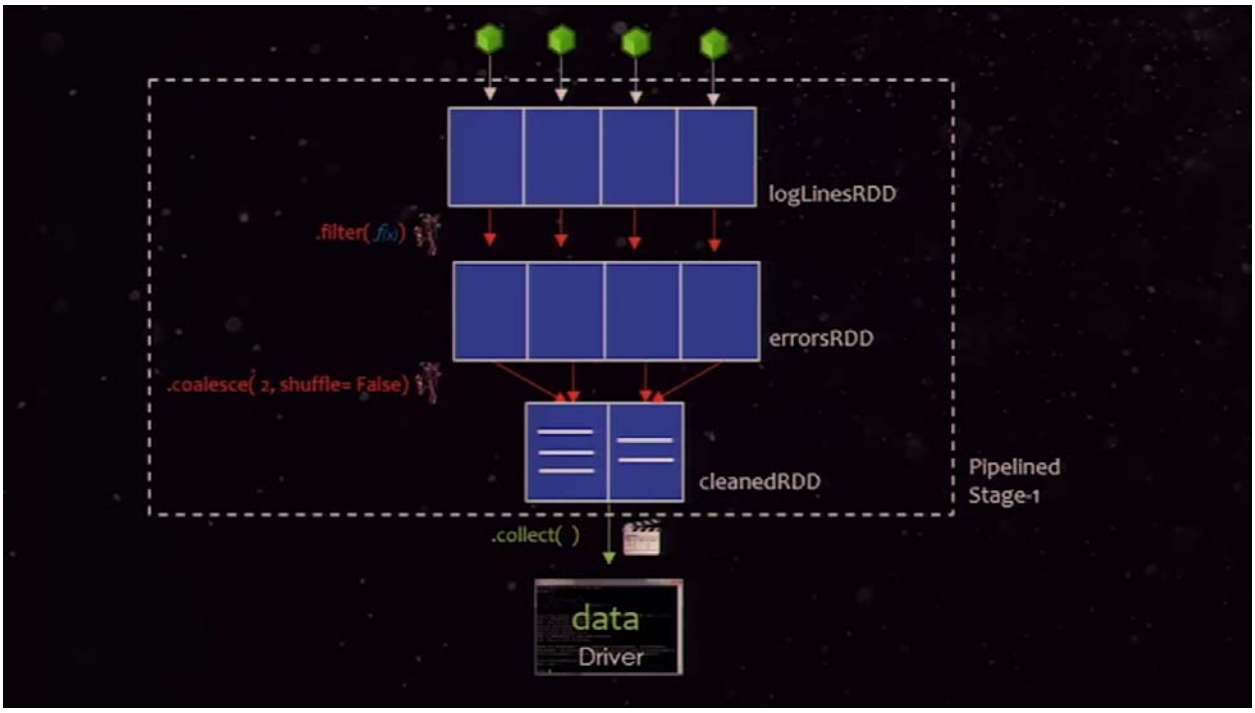
- There are other methods to read data from HDFS, C*, S3, HBase, etc.

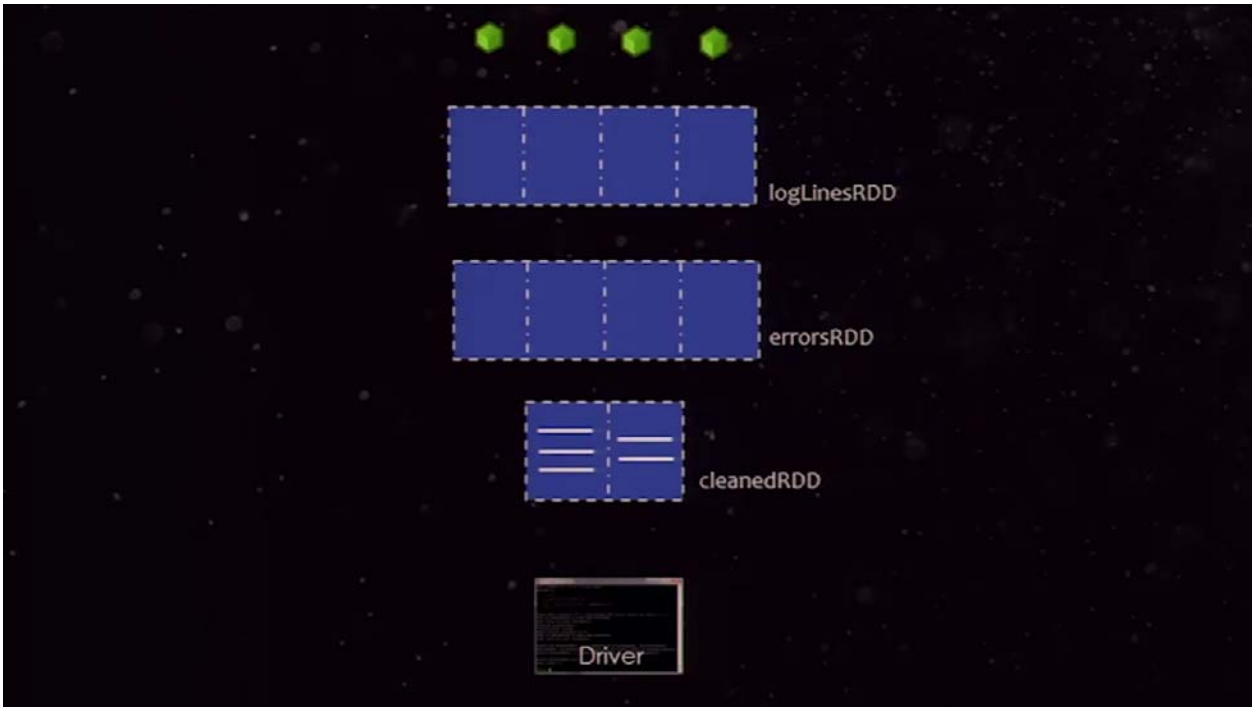

hadoop
HDFS
Error, ts, msg1
Warn, ts, msg2
Error, ts, msg1
Info, ts, msg8
Warn, ts, msg2
Info, ts, msg8
Error, ts, msg3
Info, ts, msg5
Info, ts, msg5
Error, ts, msg4
Warn, ts, msg9
Error, ts, msg1
logLinesRDD
(input/base RDD)
.filter(fx)
Error, ts, msg1
Error, ts, msg1
Error, ts, msg3
Error, ts, msg4
Error, ts, msg1
errorsRDD

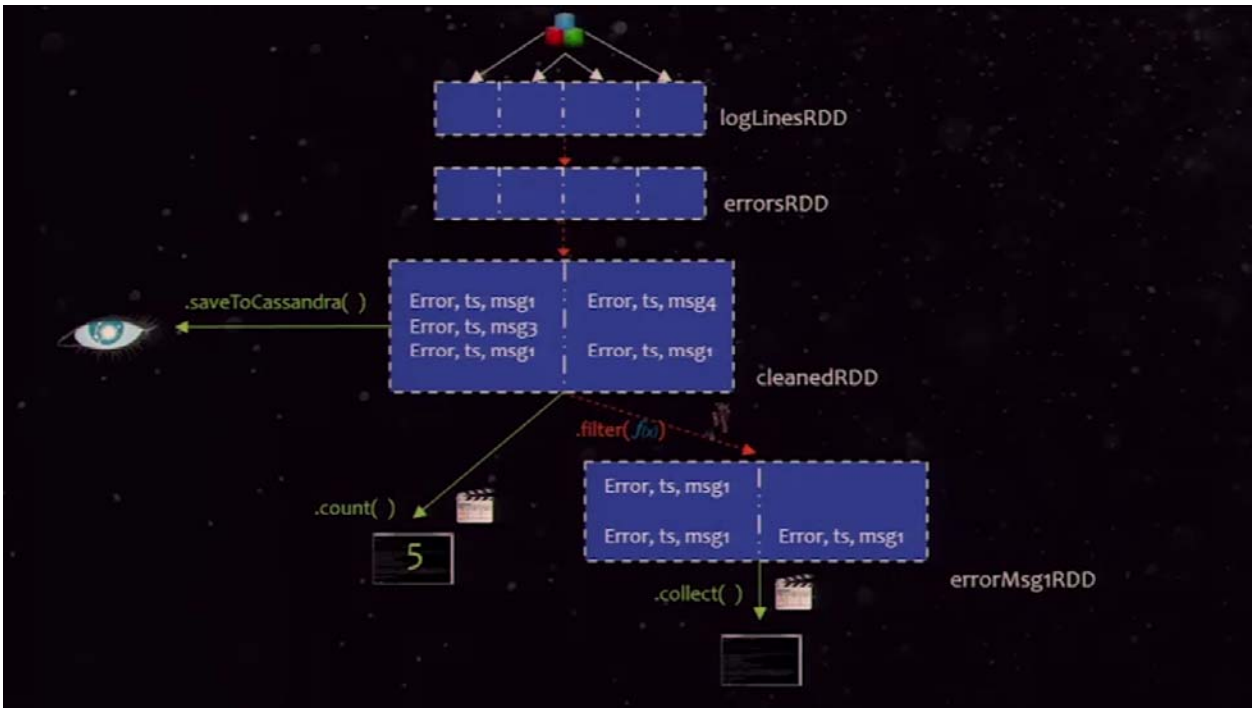

Error, ts, msg1
Error, ts, msg1
Error, ts, msg3
Error, ts, msg4
Error, ts, msg1
errorsRDD
.coalesce(2)
Error, ts, msg1
Error, ts, msg3
Error, ts, msg1
Error, ts, msg4
Error, ts, msg1
cleanedRDD
.collect( )
Driver

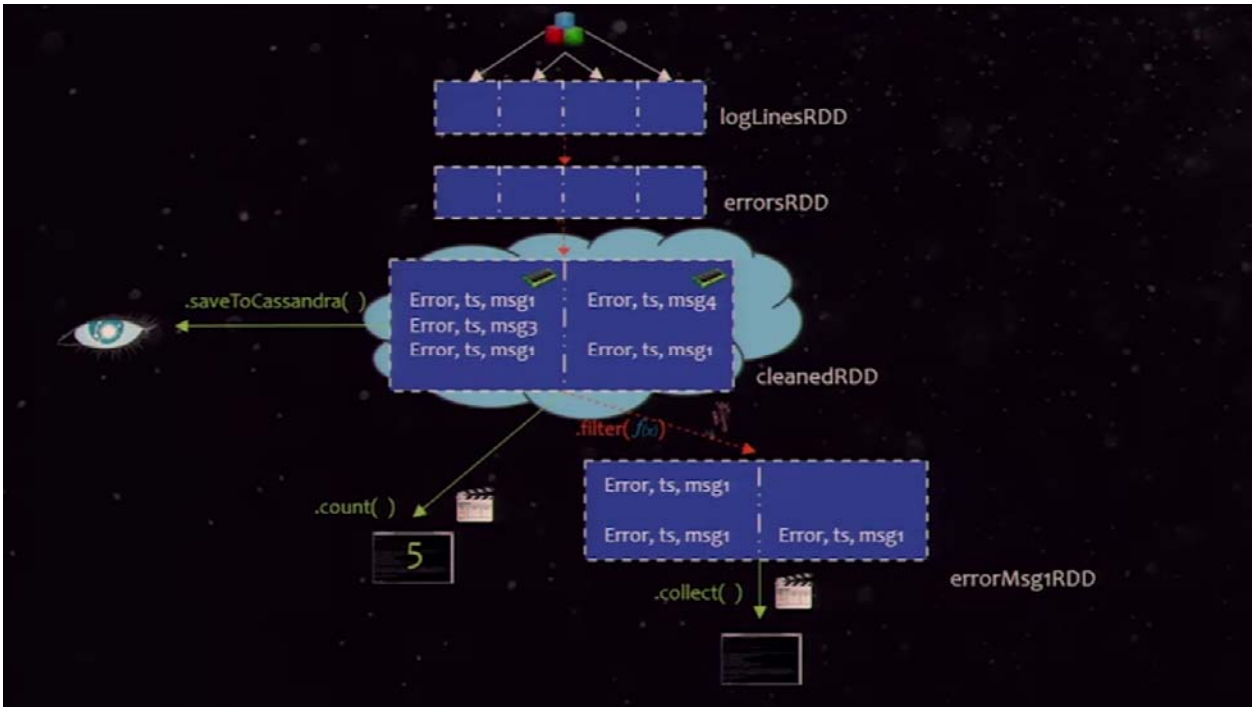

Execute DAG!
.collect( )

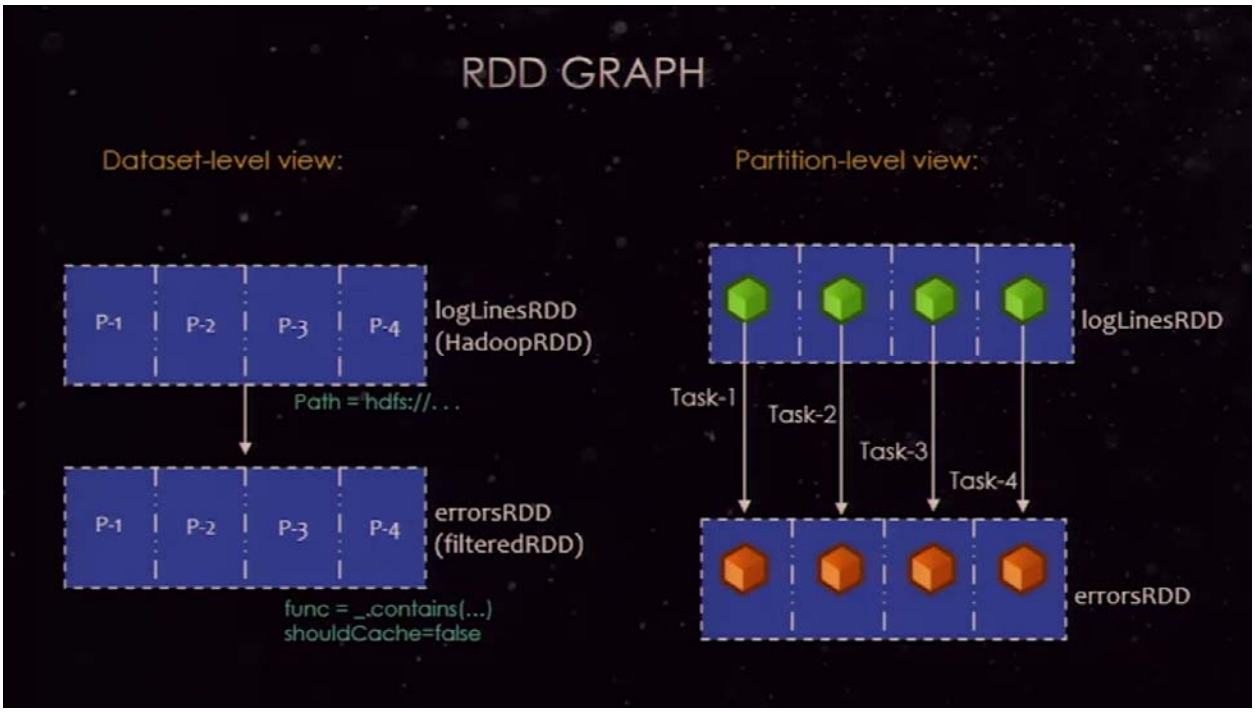

logLinesRDD
.filter(fx)
errorsRDD
.coalesce(2)
cleanedRDD
.collect( )
Error, ts, msg1
Error, ts, msg4
Error, ts, msg3
Error, ts, msg2
Error, ts, msg1
Driver


logLinesRDD
.filter(fn)
errorsRDD
.coalesce(2, shuffle= False)
cleanedRDD
Pipelined
Stage-1
.collect( )
data
Driver


logLinesRDD
errorsRDD
cleanedRDD
Driver


logLinesRDD
errorsRDD
.saveToCassandra( )
Error, ts, msg1
Error, ts, msg3
Error, ts, msg1
Error, ts, msg4
Error, ts, msg1
cleanedRDD
.filter(fbs)
.count( )
5
Error, ts, msg1
Error, ts, msg1
Error, ts, msg1
errorMsg1RDD
.collect( )


logLinesRDD
errorsRDD
.saveToCassandra( )
Error, ts, msg1
Error, ts, msg3
Error, ts, msg1
Error, ts, msg4
Error, ts, msg1
cleanedRDD
.filter(fbs)
.count( )
5
Error, ts, msg1
Error, ts, msg1
Error, ts, msg1
errorMsg1RDD
.collect( )



LIFECYCLE OF A SPARK PROGRAM

1) Create some input RDDs from external data or parallelize a collection in your driver program.

2) Lazily transform them to define new RDDs using transformations like `filter()` or `map()`

3) Ask Spark to `cache()` any intermediate RDDs that will need to be reused.

4) Launch actions such as `count()` and `collect()` to kick off a parallel computation, which is then optimized and executed by Spark.

TRANSFORMATIONS (lazy)

map()	intersection()	cartesion()
flatMap()	distinct()	pipe()
filter()	groupByKey()	coalesce()
mapPartitions()	reduceByKey()	repartition()
mapPartitionsWithIndex()	sortByKey()	partitionBy()
sample()	join()	...
union()	cogroup()	...

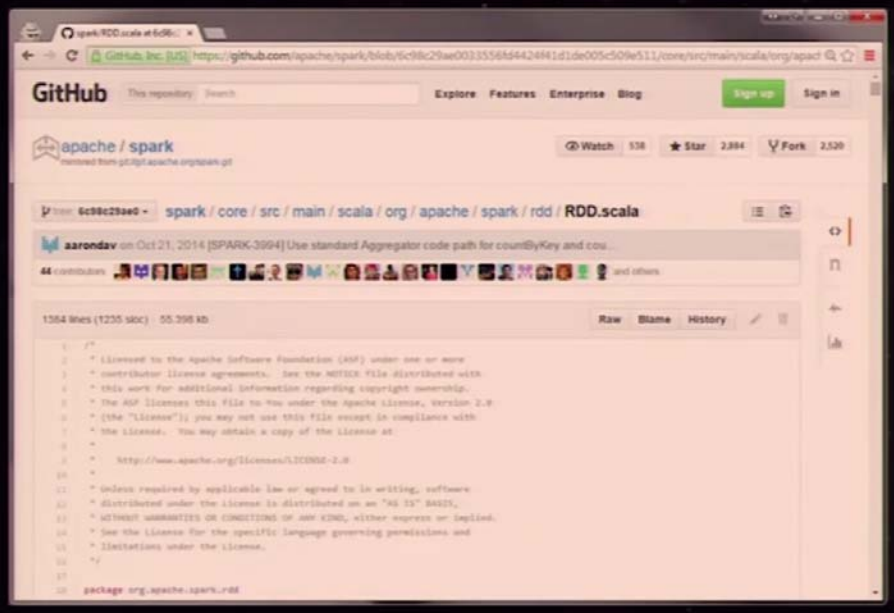
- Most transformations are element-wise (they work on one element at a time), but this is not true for all transformations

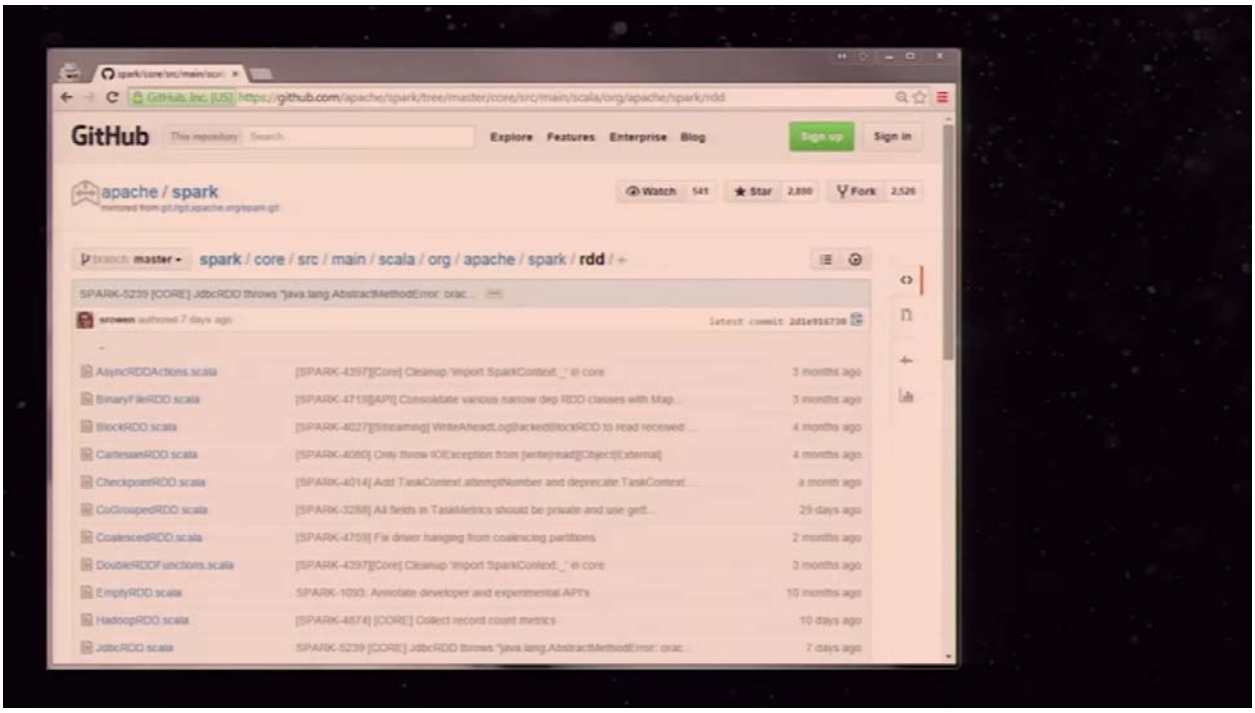
ACTIONS

reduce()	takeOrdered()
collect()	saveAsTextFile()
count()	saveAsSequenceFile()
first()	saveAsObjectFile()
take()	countByKey()
takeSample()	foreach()
saveToCassandra()	...

TYPES OF RDDS

- HadoopRDD
- FilteredRDD
- MappedRDD
- PairRDD
- ShuffledRDD
- UnionRDD
- PythonRDD
- DoubleRDD
- JdbcRDD
- JsonRDD
- SchemaRDD
- VertexRDD
- EdgeRDD
- CassandraRDD *(DataStax)*
- GeoRDD *(ESRI)*
- EsSpark *(ElasticSearch)*




GitHub
Explore
Features
Enterprise
Blog
Sign up
Sign in
apache / spark
Watch
Star
Fork
branch: master
spark / core / src / main / scala / org / apache / spark / rdd /
AsyncRDDActions.scala
BinaryFileRDD.scala
BlockRDD.scala
CartesianRDD.scala
CheckpointRDD.scala
CoGroupedRDD.scala
CoalescedRDD.scala
DoubleRDDFunctions.scala
EmptyRDD.scala
HadoopRDD.scala
JdbcRDD.scala

RDD INTERFACE

* 1) Set of partitions ("splits")
* 2) List of dependencies on parent RDDs
* 3) Function to compute a partition given parents
* 4) Optional preferred locations
* 5) Optional partitioning info for k/v RDDs (Partitioner)

This captures all current Spark operations!

EXAMPLE: HADOOPRDD

* Partitions = one per HDFS block
* Dependencies = none
* Compute (partition) = read corresponding block

* preferredLocations (part) = HDFS block location
* Partitioner = none

EXAMPLE: FILTEREDRDD

- Partitions = same as parent RDD
- Dependencies = "one-to-one" on parent
- Compute (partition) = compute parent and filter it

- preferredLocations (part) = none (ask parent)
- Partitioner = none

EXAMPLE: JOINEDRDD

- Partitions = One per reduce task
- Dependencies = "shuffle" on each parent
- Compute (partition) = read and join shuffled data

- preferredLocations (part) = none
- Partitioner = HashPartitioner(numTasks)

READING DATA USING THE C* CONNECTOR

Keyspace Table

```
val cassandraRDD = sc
                   .cassandraTable("ks", "mytable")
                   .select("col-1", "col-3")
                   .where("col-5 = ?", "blue")
```

Server side column & row selection

INPUT SPLIT SIZE

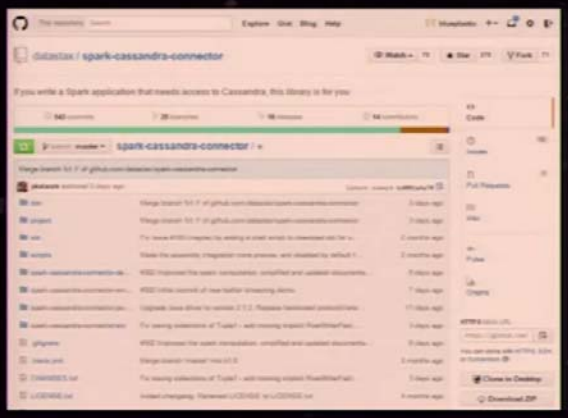
(for dealing with wide rows)

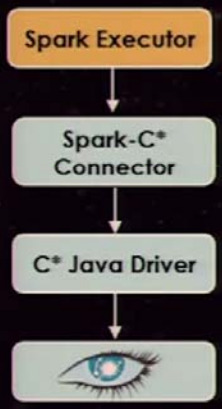
Start the Spark shell by passing in a custom cassandra.input.split.size:

```
ubuntu@ip-10-0-53-24:~$ dse spark -Dspark.cassandra.input.split.size=2000
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 0.9.1
      /_/

Using Scala version 2.10.3 (Java HotSpot(TM) 64-Bit Server VM, Java
1.7.0_51)
Type in expressions to have them evaluated.
Type :help for more information.
Creating SparkContext...
Created spark context..
Spark context available as sc.
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

The `cassandra.input.split.size` parameter defaults to 100,000. This is the approximate number of physical rows in a single Spark partition. If you have really wide rows (thousands of columns), you may need to lower this value. The higher the value, the fewer Spark tasks are created. Increasing the value too much may limit the parallelism level."

https://github.com/datastax/spark-cassandra-connector



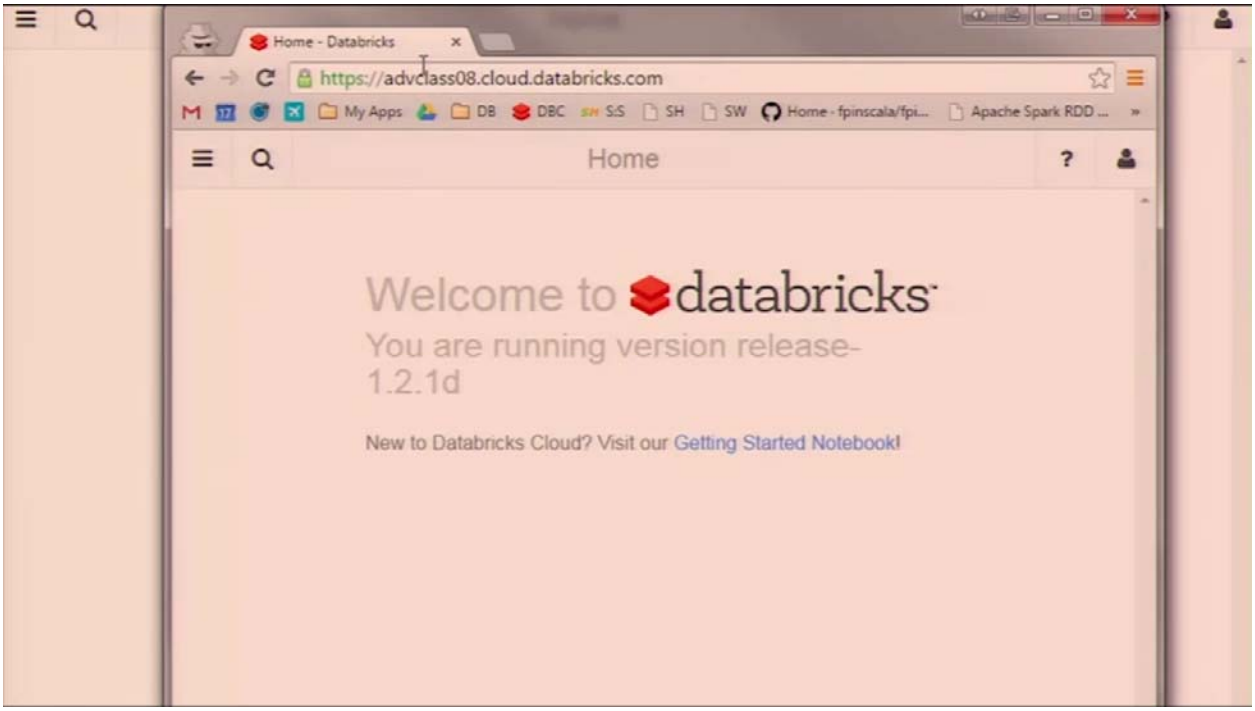


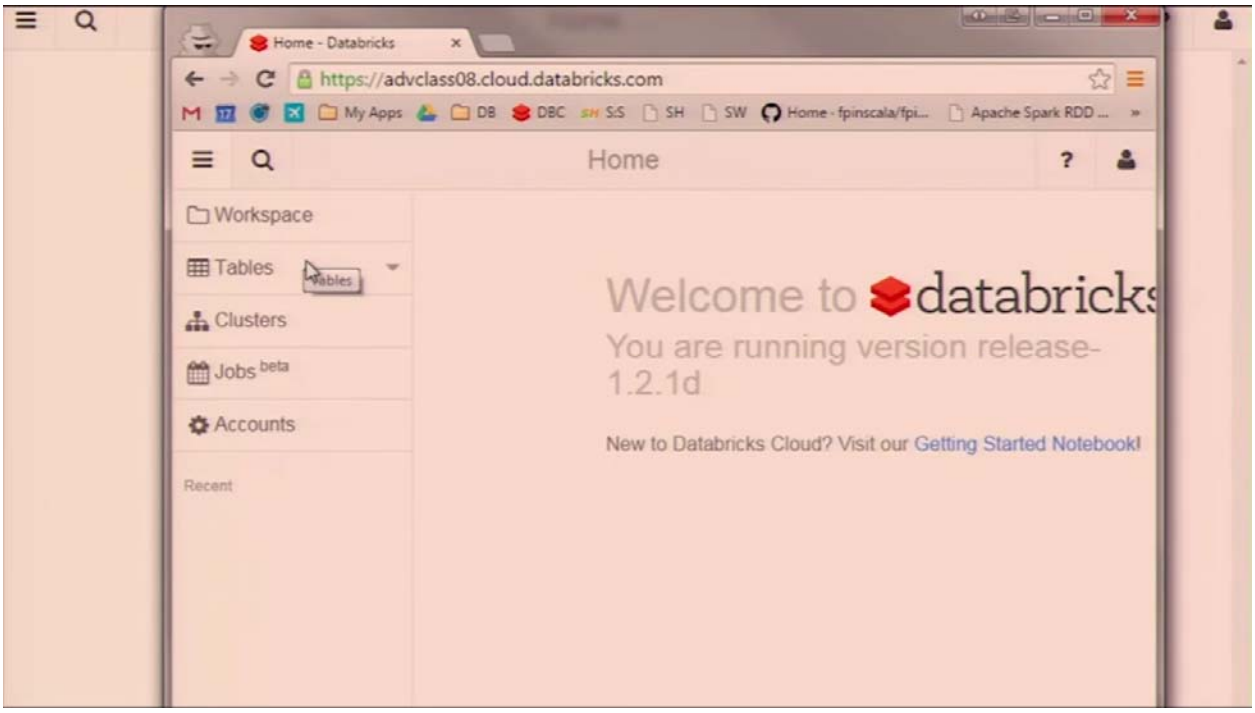
- Open Source
- Implemented mostly in Scala
- Scala + Java APIs
- Does automatic type conversions

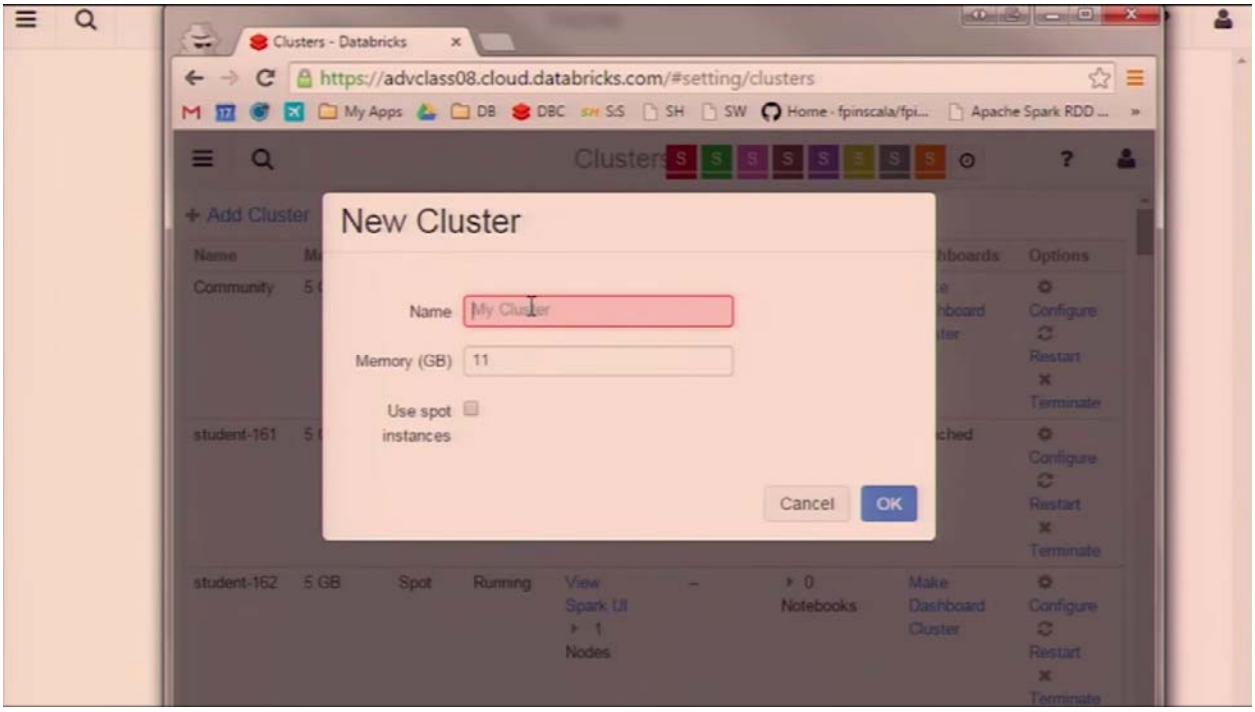
"Simple things
should be simple,
complex things
should be possible"

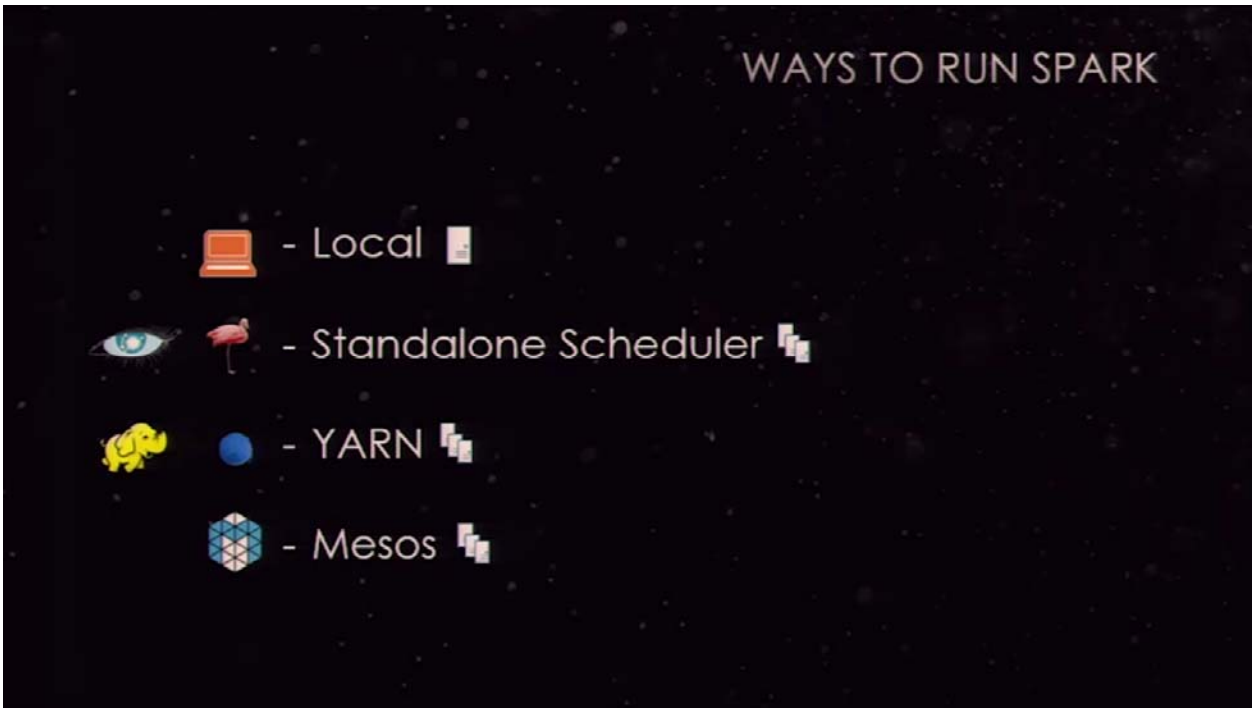
- Alan Kay


DEMO: DATABRICKS CLOUD GUI
databricks

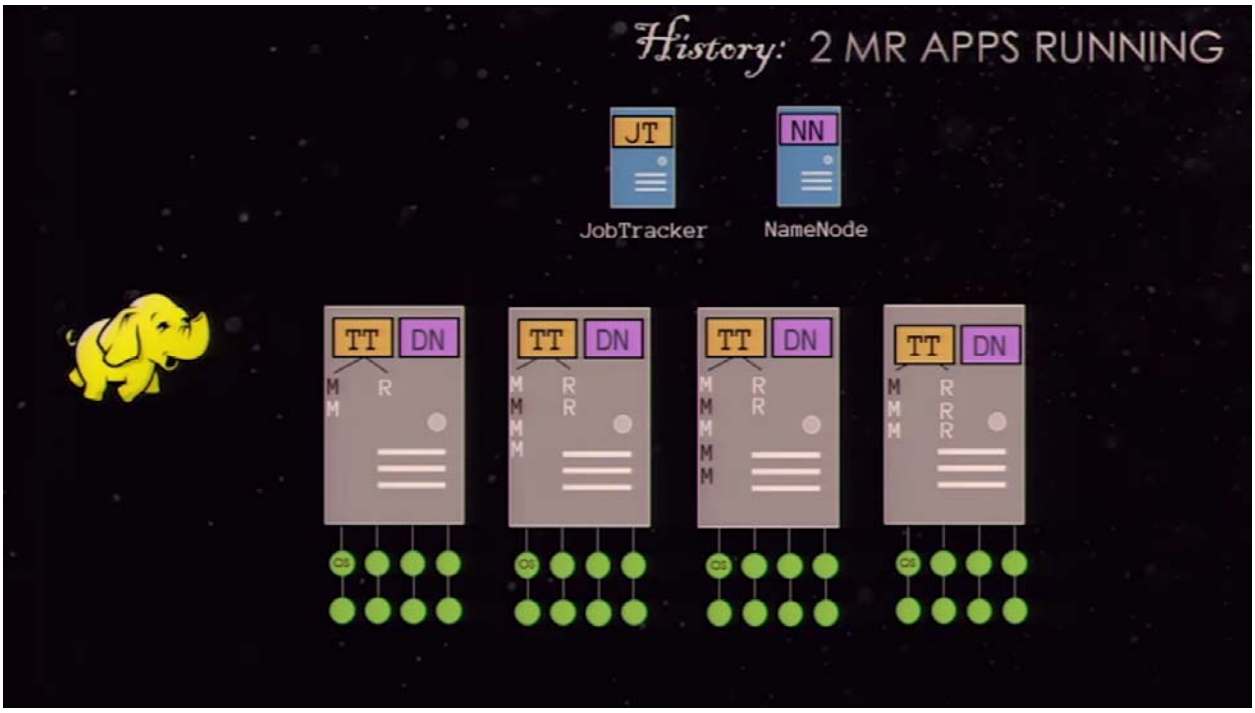

Home - Databricks
https://advclass08.cloud.databricks.com
My Apps
DB
DBC
SH
SW
Home - fpinscala/fpi...
Apache Spark RDD ...
Home
Welcome to databricks
You are running version release-1.2.1d
New to Databricks Cloud? Visit our Getting Started Notebook!

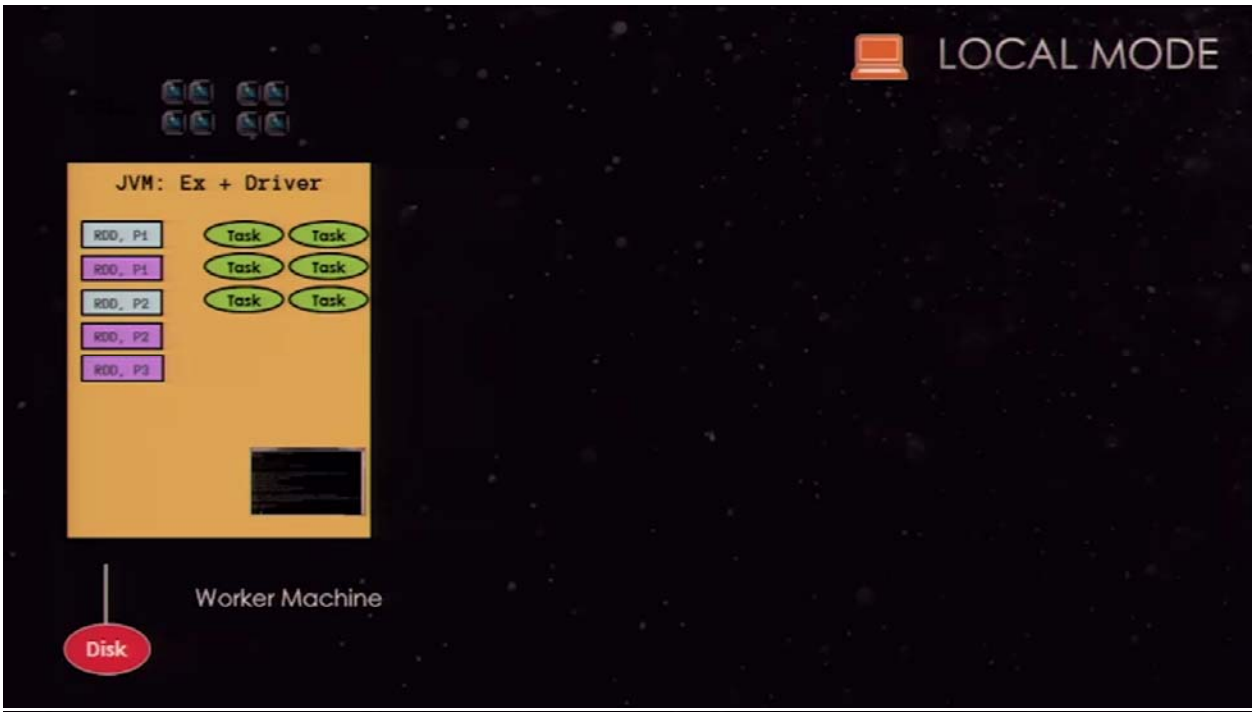

Home - Databricks
https://advclass08.cloud.databricks.com
Home
Workspace
Tables
Clusters
Jobs beta
Accounts
Recent
Welcome to databricks
You are running version release-1.2.1d
New to Databricks Cloud? Visit our Getting Started Notebook!

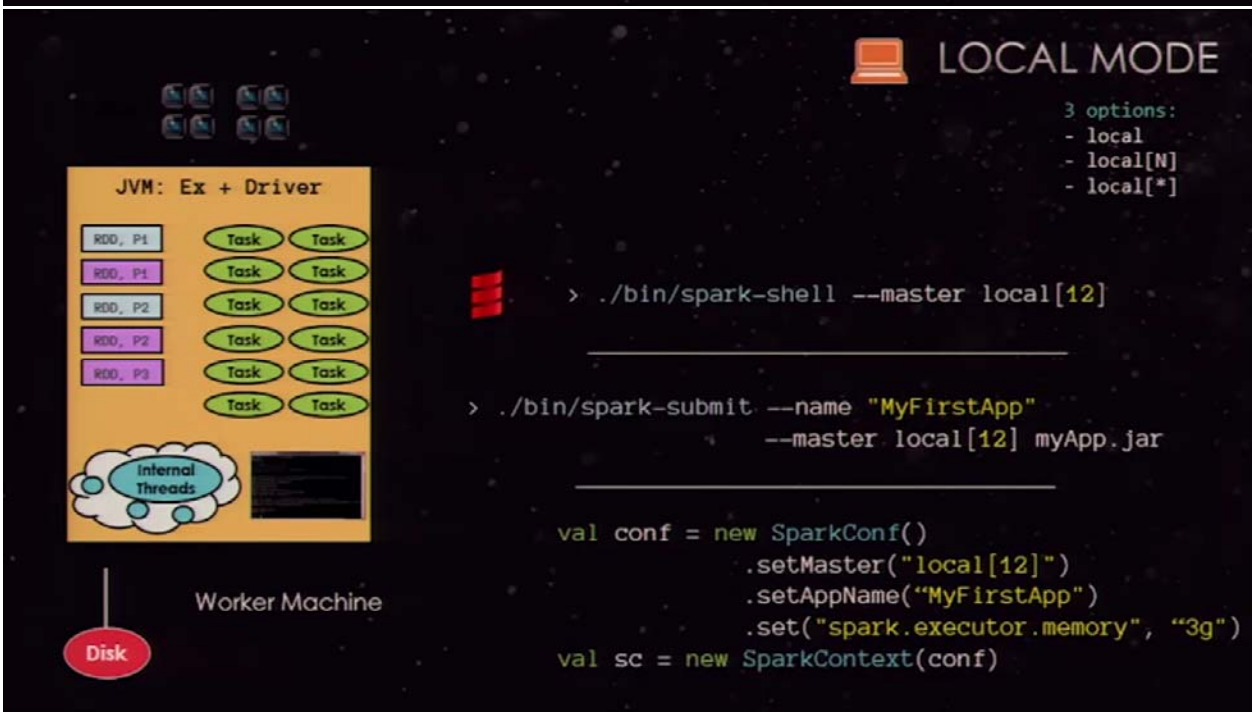

Clusters - Databricks
https://advclass08.cloud.databricks.com/#setting/clusters
Clusters
+ Add Cluster
New Cluster
Name
My Cluster
Memory (GB)
11
Use spot instances
Cancel
OK
Community
student-161
student-162
5 GB
Spot
Running
Configure
Restart
Terminate
Notebooks
Make Dashboard Cluster
Nodes


WAYS TO RUN SPARK
- Local
- Standalone Scheduler
- YARN
- Mesos

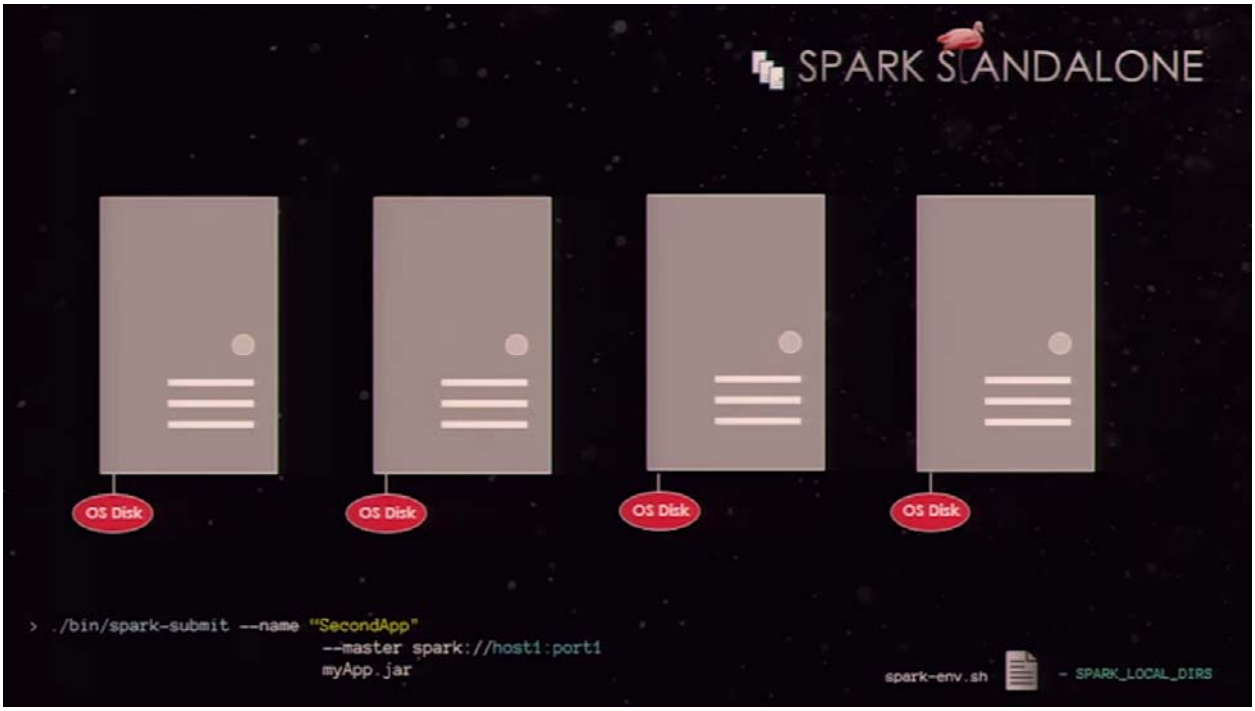

WAYS TO RUN SPARK
- Local
- Standalone Scheduler
- YARN
- Mesos
Static Partitioning
Dynamic Partitioning

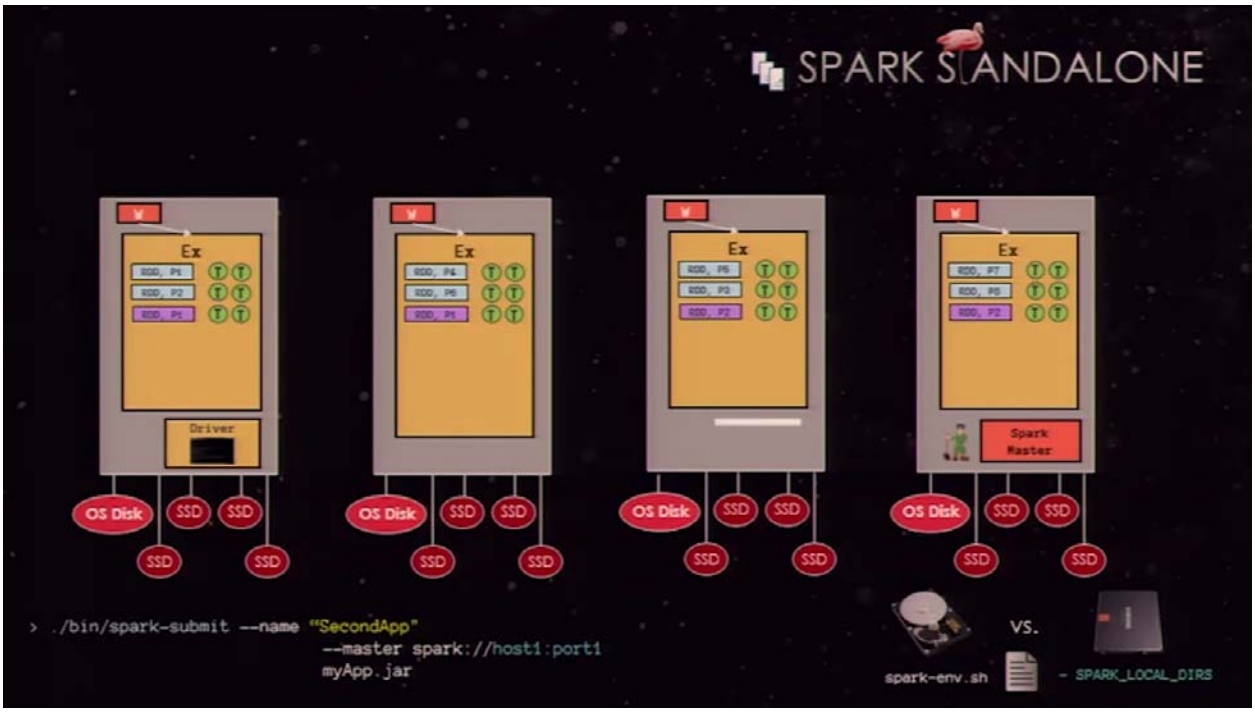

History: 2 MR APPS RUNNING
JT
NN
JobTracker
NameNode
TT
DN
M
R
CS

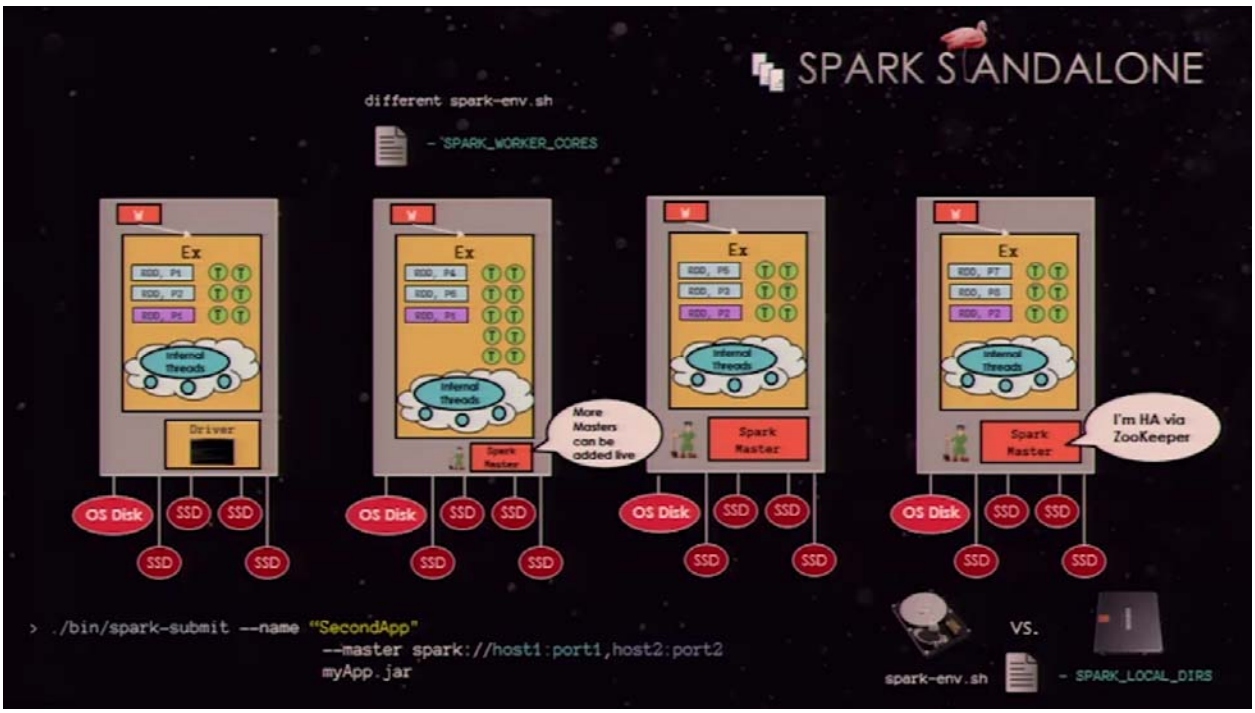

LOCAL MODE
JVM: Ex + Driver
RDD, P1
RDD, P1
RDD, P2
RDD, P2
RDD, P3
Task
Task
Task
Task
Task
Task
Worker Machine
Disk

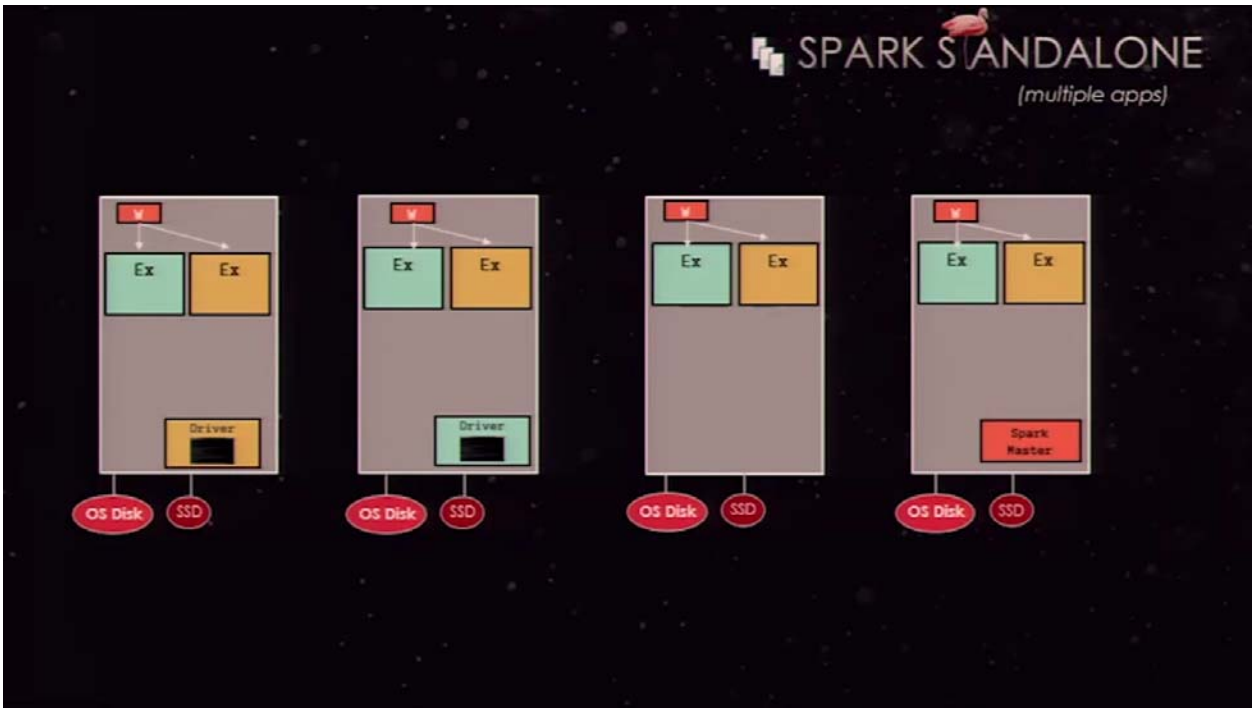

LOCAL MODE
3 options:
- local
- local[N]
- local[*]
JVM: Ex + Driver
RDD, P1
RDD, P1
RDD, P2
RDD, P2
RDD, P3
Task
Internal Threads
Worker Machine
Disk
> ./bin/spark-shell --master local[12]
> ./bin/spark-submit --name "MyFirstApp"
--master local[12] myApp.jar
val conf = new SparkConf()
.setMaster("local[12]")
.setAppName("MyFirstApp")
.set("spark.executor.memory", "3g")
val sc = new SparkContext(conf)

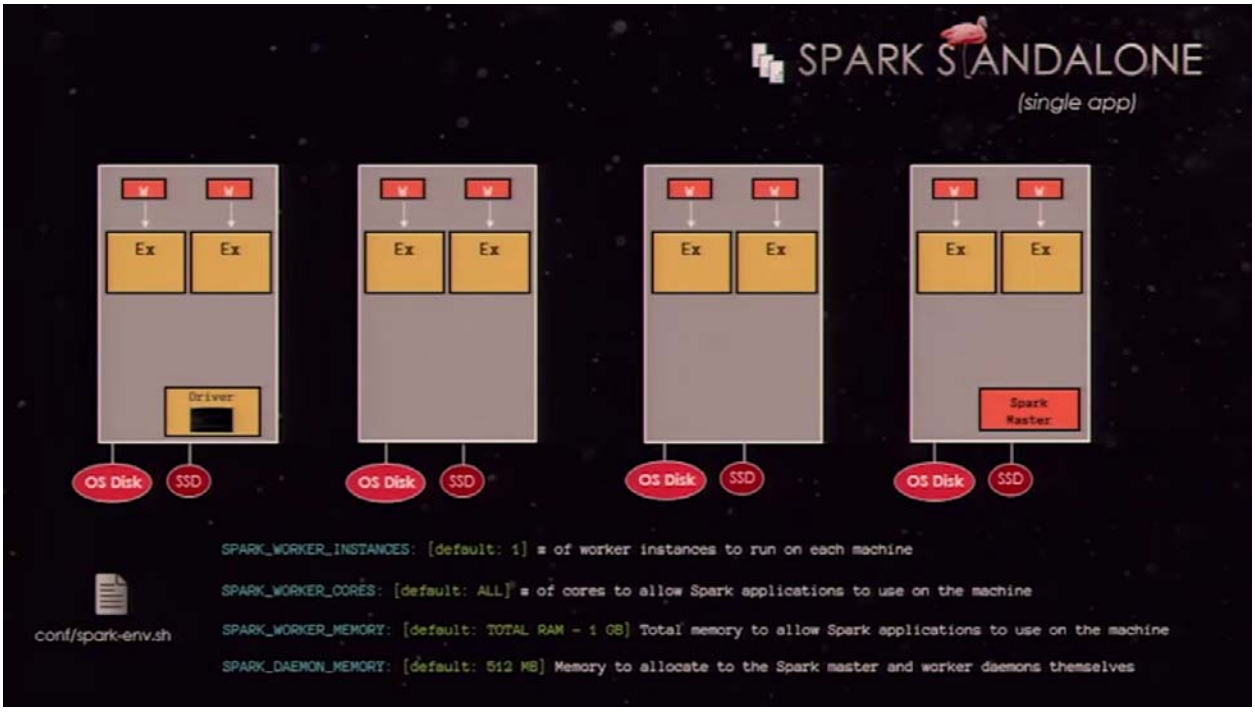

STANDALONE MODE


SPARK STANDALONE
OS Disk
OS Disk
OS Disk
OS Disk
> ./bin/spark-submit --name "SecondApp"
--master spark://host1:port1
myApp.jar
spark-env.sh
- SPARK_LOCAL_DIRS


SPARK STANDALONE
W
Ex
RDD, P1
RDD, P2
RDD, P3
Driver
Spark Master
OS Disk
SSD
> ./bin/spark-submit --name "SecondApp"
--master spark://host1:port1
myApp.jar
VS.
spark-env.sh
- SPARK_LOCAL_DIRS


SPARK STANDALONE
different spark-env.sh
- SPARK_WORKER_CORES
W
Ex
RDD, P1
RDD, P2
RDD, P3
RDD, P4
RDD, P5
RDD, P6
RDD, P7
RDD, P8
Internal threads
Driver
Spark Master
More Masters can be added live
I'm HA via ZooKeeper
OS Disk
SSD
> ./bin/spark-submit --name "SecondApp"
--master spark://host1:port1,host2:port2
myApp.jar
VS.
spark-env.sh
- SPARK_LOCAL_DIRS


SPARK STANDALONE
(multiple apps)
W
Ex
Ex
Driver
Spark Master
OS Disk
SSD


SPARK STANDALONE
(single app)
W
Ex
Ex
Driver
Spark Master
OS Disk
SSD
conf/spark-env.sh
SPARK_WORKER_INSTANCES: [default: 1] # of worker instances to run on each machine
SPARK_WORKER_CORES: [default: ALL] # of cores to allow Spark applications to use on the machine
SPARK_WORKER_MEMORY: [default: TOTAL RAM - 1 GB] Total memory to allow Spark applications to use on the machine
SPARK_DAEMON_MEMORY: [default: 512 MB] Memory to allocate to the Spark master and worker daemons themselves

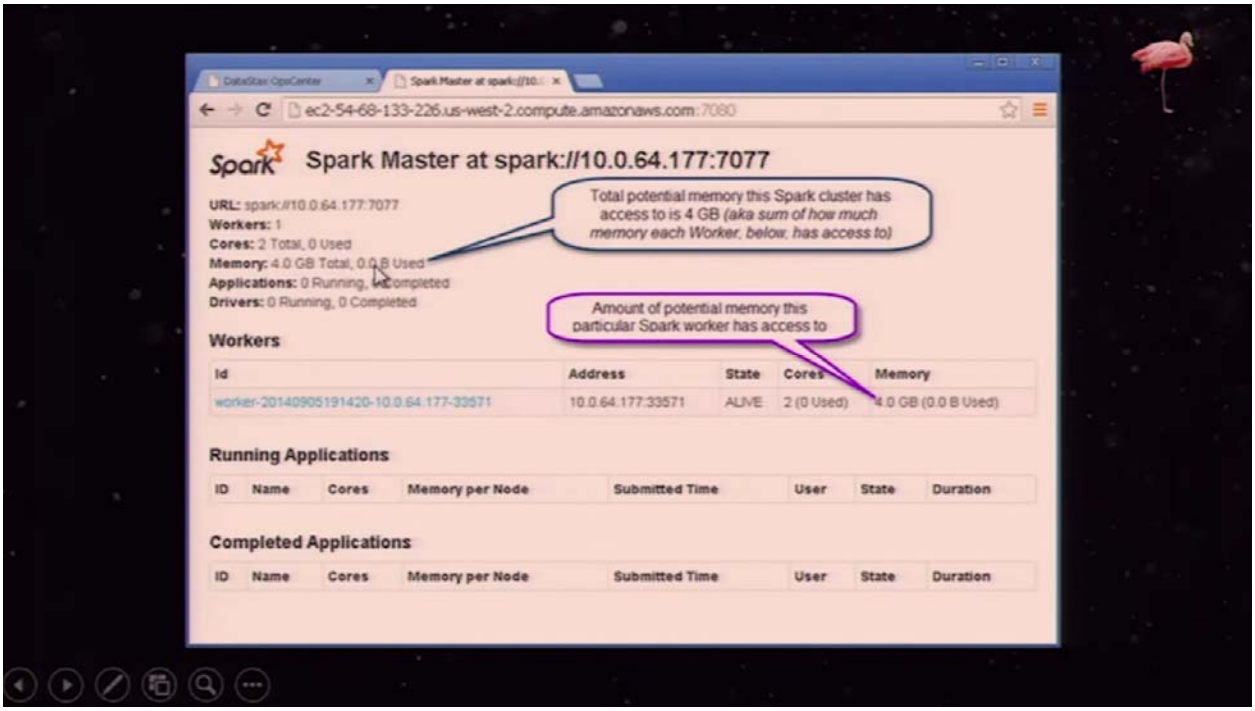
STANDALONE

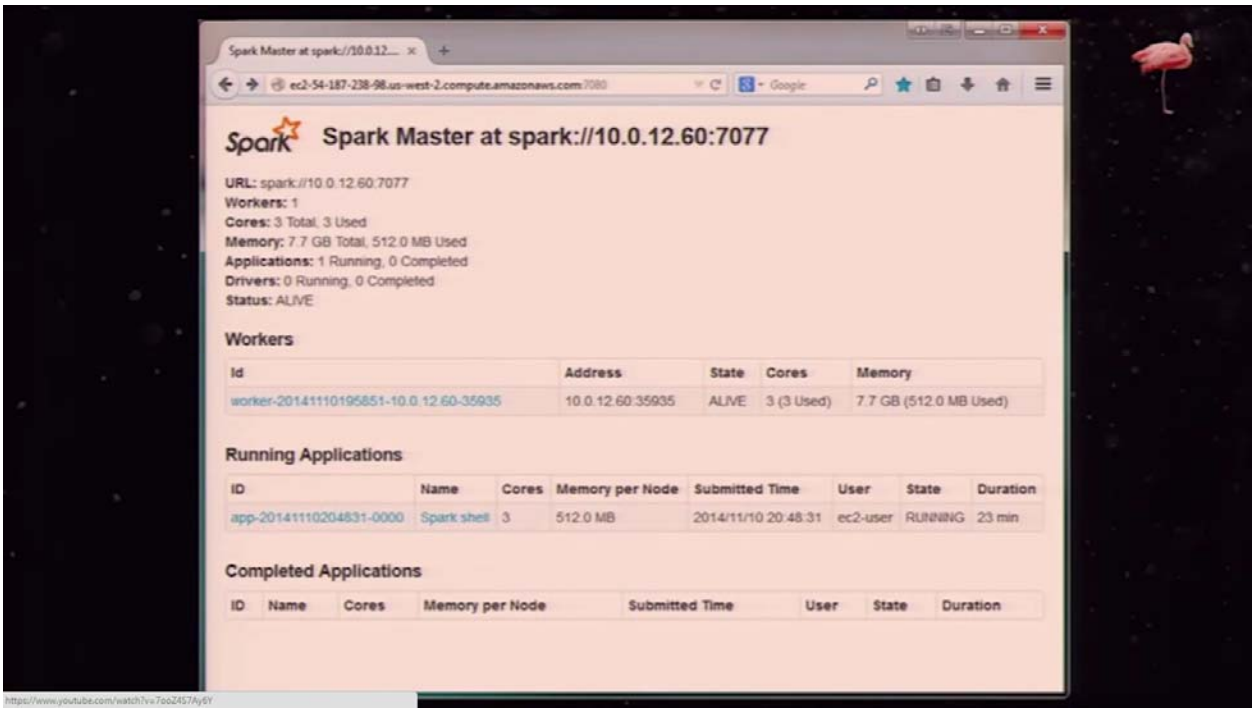
Standalone settings

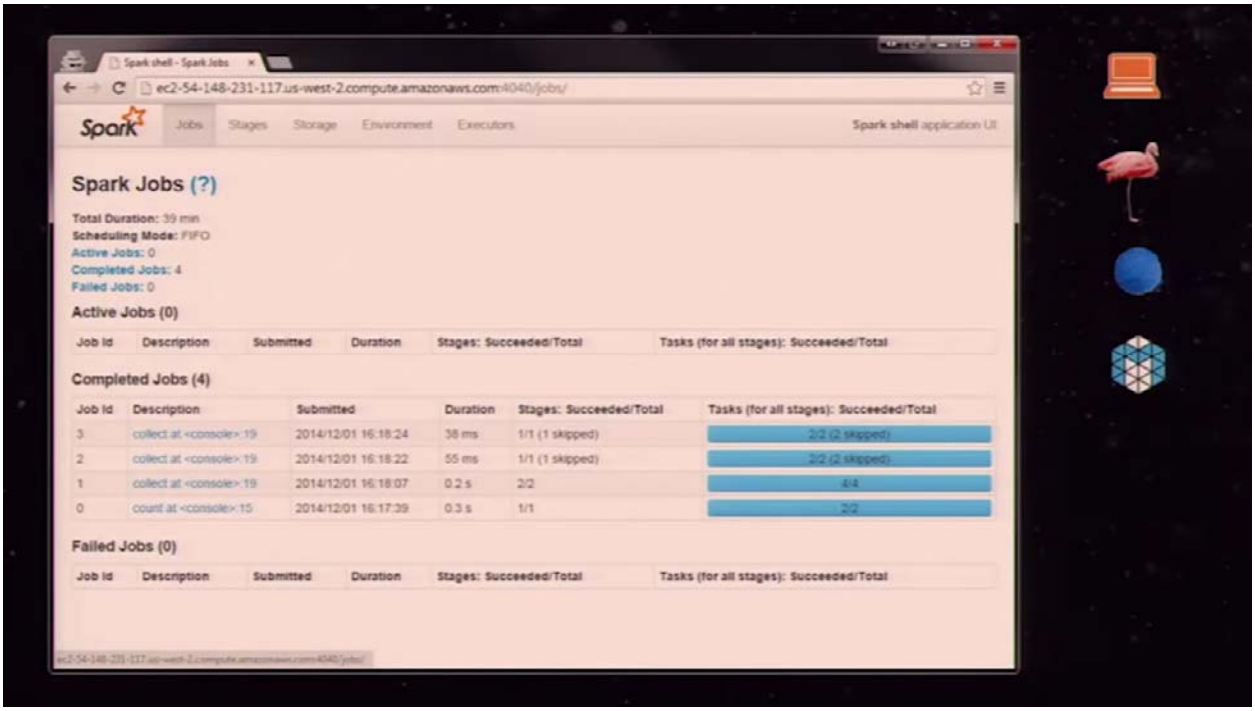
- Apps submitted will run in FIFO mode by default

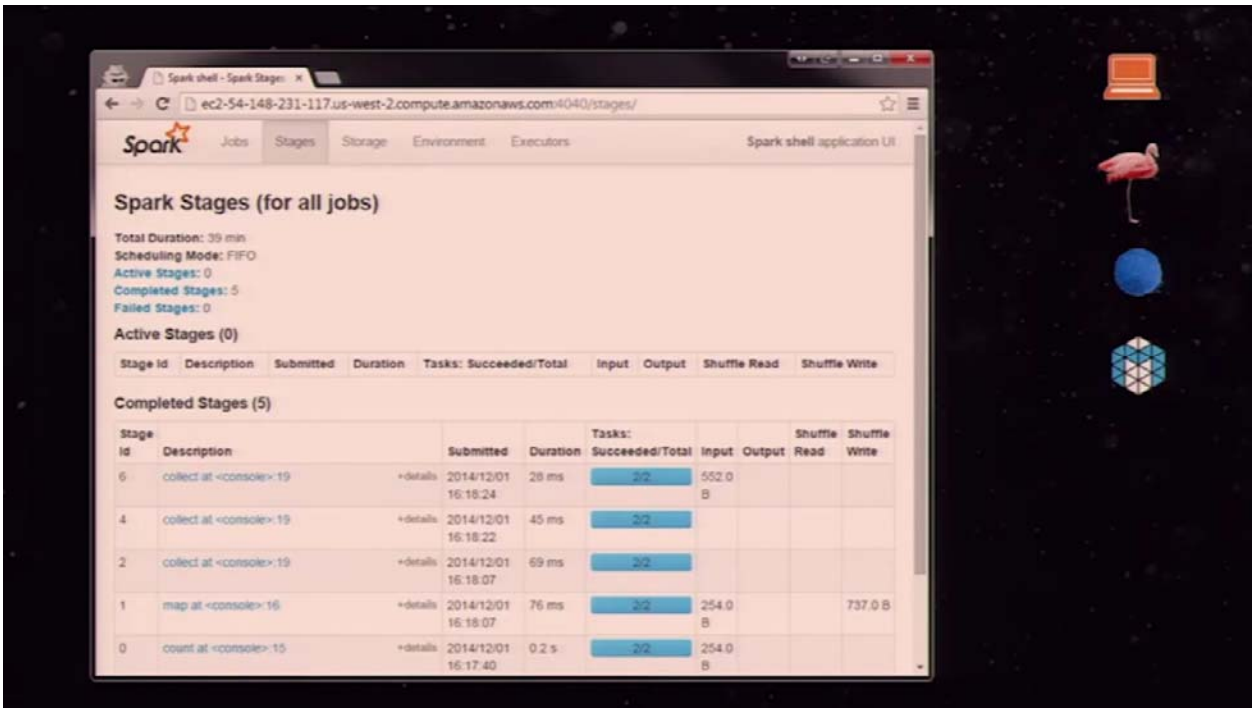
`spark.cores.max`: maximum amount of CPU cores to request for the application from across the cluster

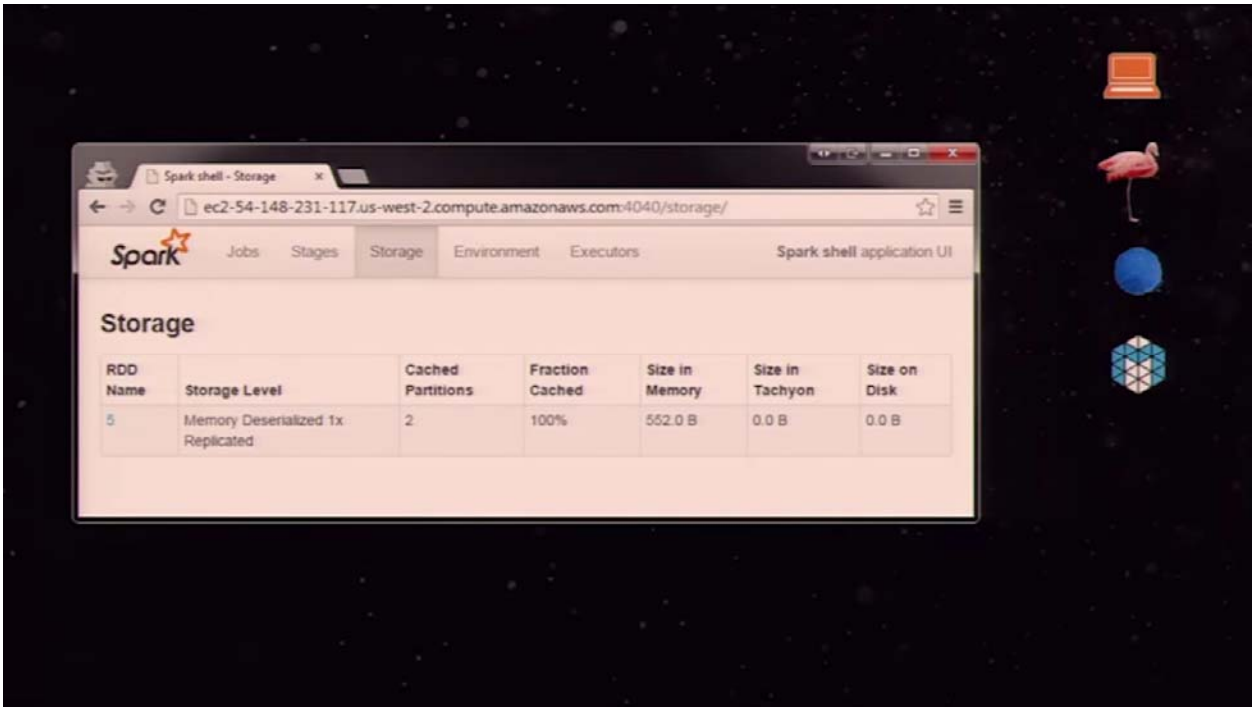
`spark.executor.memory`: Memory for each executor

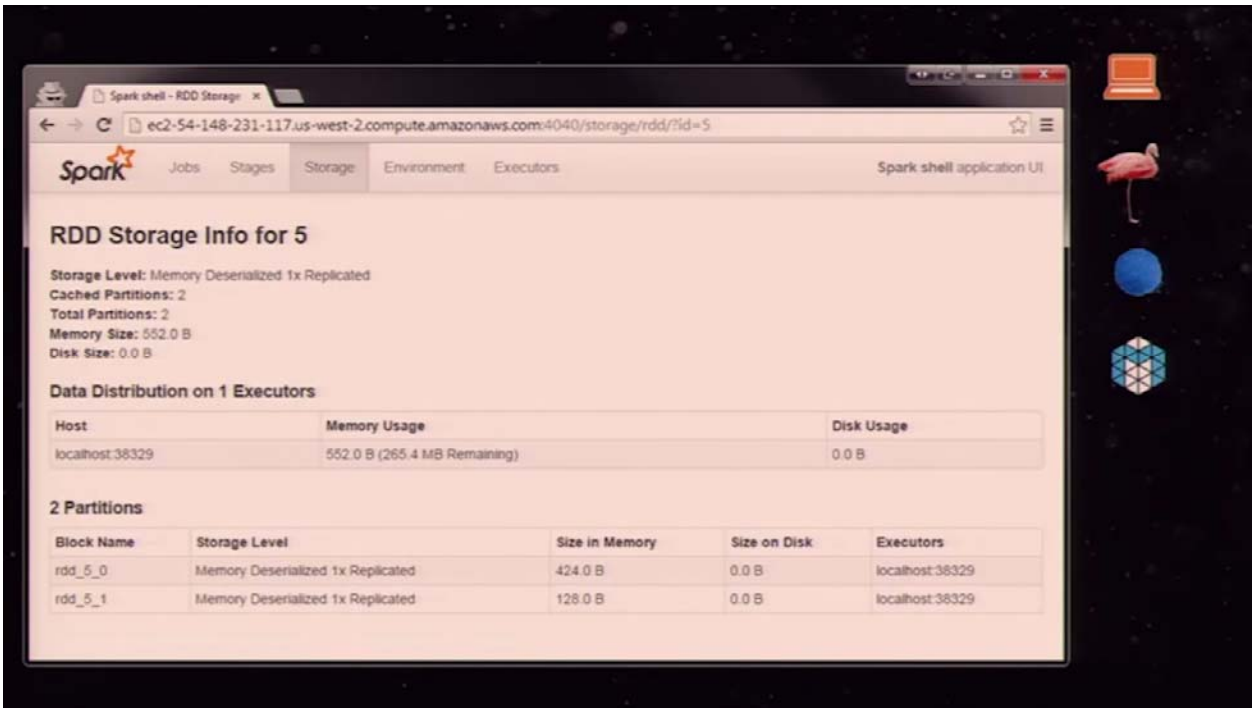


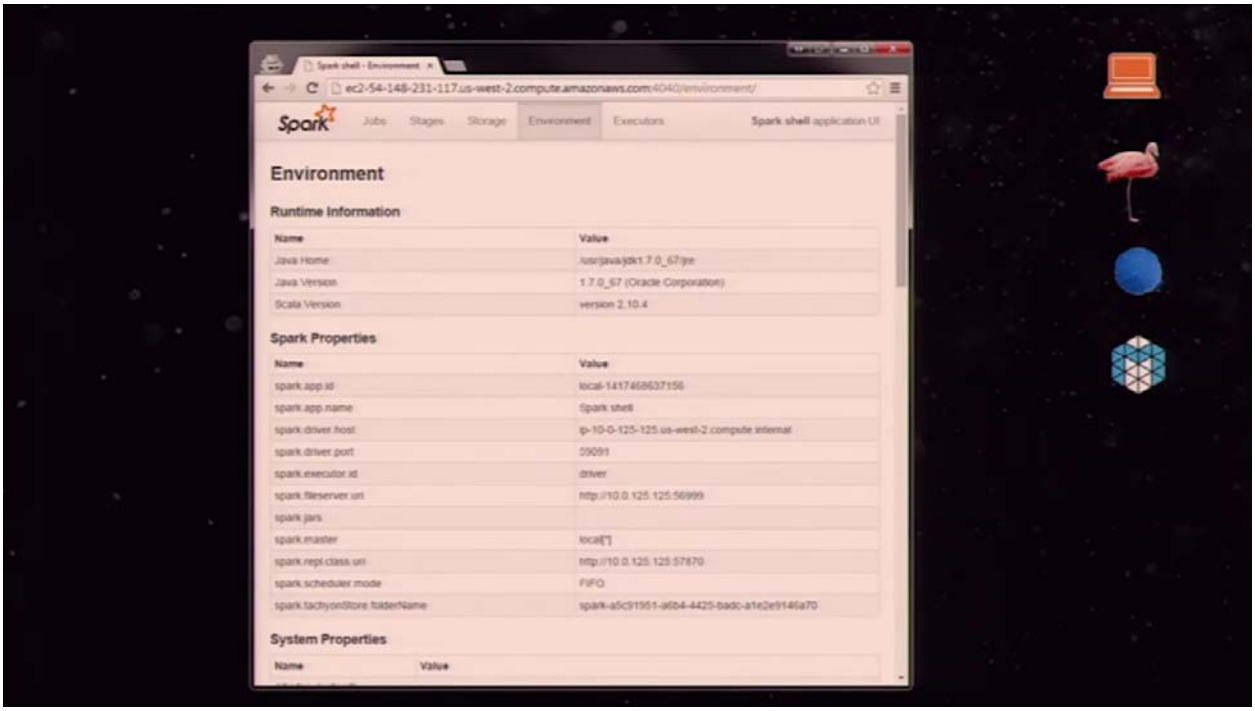

Spark Master at spark://10.0.12....
ec2-54-187-238-98.us-west-2.compute.amazonaws.com:7080
Spark Master at spark://10.0.12.60:7077
URL: spark://10.0.12.60:7077
Workers: 1
Cores: 3 Total, 3 Used
Memory: 7.7 GB Total, 512.0 MB Used
Applications: 1 Running, 0 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE
Workers
Id
Address
State
Cores
Memory
worker-20141110195851-10.0.12.60-35935
10.0.12.60:35935
ALIVE
3 (3 Used)
7.7 GB (512.0 MB Used)
Running Applications
ID
Name
Cores
Memory per Node
Submitted Time
User
State
Duration
app-20141110204831-0000
Spark shell
3
512.0 MB
2014/11/10 20:48:31
ec2-user
RUNNING
23 min
Completed Applications
ID
Name
Cores
Memory per Node
Submitted Time
User
State
Duration

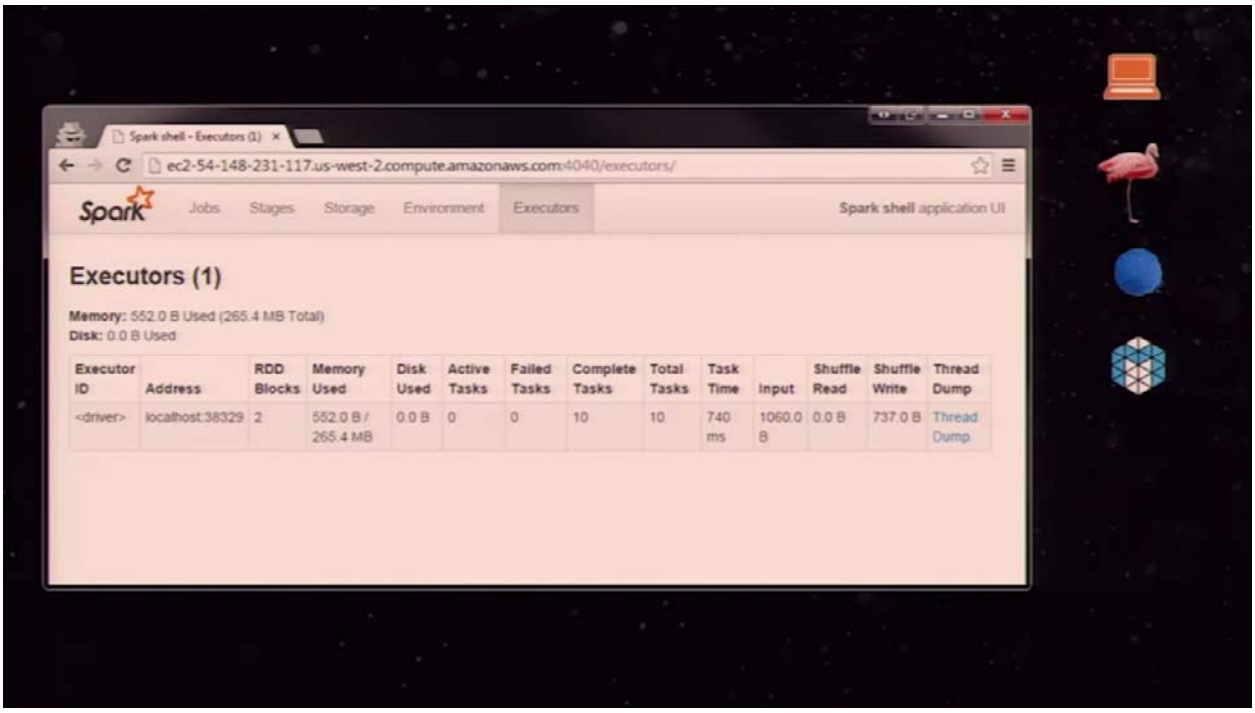

Spark shell - Spark Jobs
ec2-54-148-231-117.us-west-2.compute.amazonaws.com:4040/jobs/
Jobs
Stages
Storage
Environment
Executors
Spark shell application UI
Spark Jobs (?)
Total Duration: 39 min
Scheduling Mode: FIFO
Active Jobs: 0
Completed Jobs: 4
Failed Jobs: 0
Active Jobs (0)
Job Id
Description
Submitted
Duration
Stages: Succeeded/Total
Tasks (for all stages): Succeeded/Total
Completed Jobs (4)
Job Id
Description
Submitted
Duration
Stages: Succeeded/Total
Tasks (for all stages): Succeeded/Total
3
collect at <console>:19
2014/12/01 16:18:24
38 ms
1/1 (1 skipped)
2/2 (2 skipped)
2
collect at <console>:19
2014/12/01 16:18:22
55 ms
1/1 (1 skipped)
2/2 (2 skipped)
1
collect at <console>:19
2014/12/01 16:18:07
0.2 s
2/2
4/4
0
count at <console>:15
2014/12/01 16:17:39
0.3 s
1/1
2/2
Failed Jobs (0)
Job Id
Description
Submitted
Duration
Stages: Succeeded/Total
Tasks (for all stages): Succeeded/Total

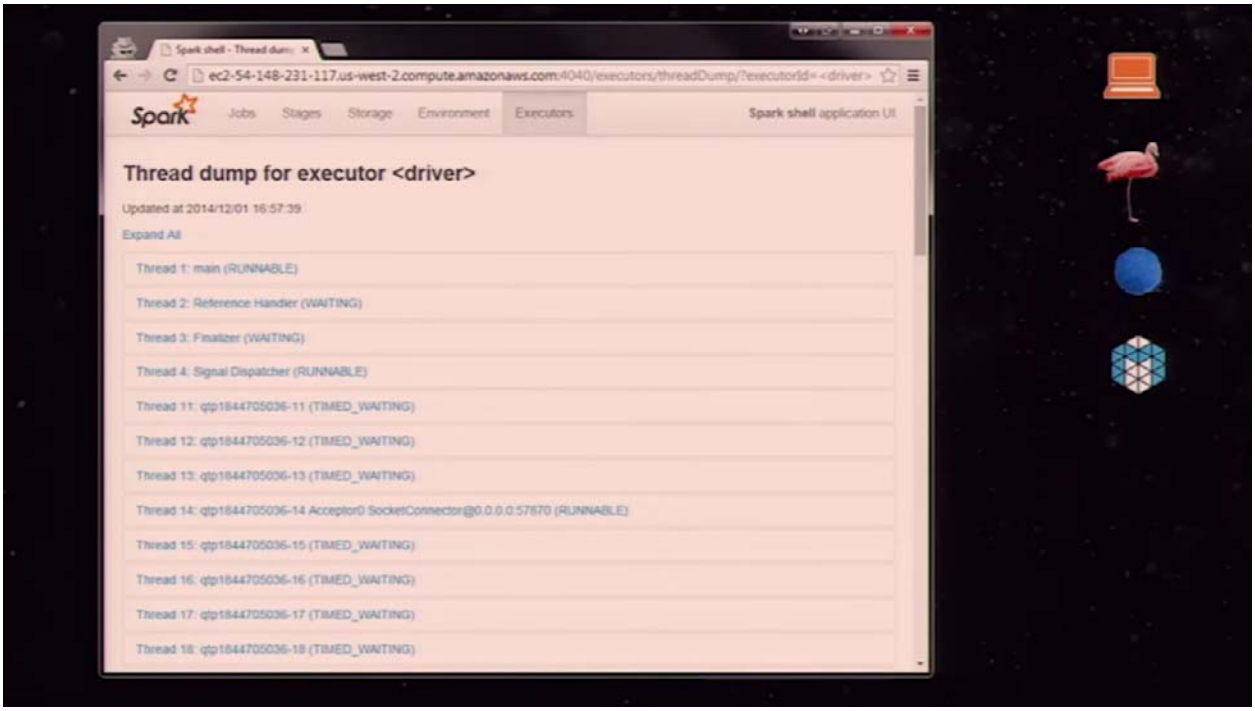

Spark shell - Spark Stages
ec2-54-148-231-117.us-west-2.compute.amazonaws.com:4040/stages/
Spark
Jobs
Stages
Storage
Environment
Executors
Spark shell application UI
Spark Stages (for all jobs)
Total Duration: 39 min
Scheduling Mode: FIFO
Active Stages: 0
Completed Stages: 5
Failed Stages: 0
Active Stages (0)
Stage Id
Description
Submitted
Duration
Tasks: Succeeded/Total
Input
Output
Shuffle Read
Shuffle Write
Completed Stages (5)
Stage Id
Description
Submitted
Duration
Tasks: Succeeded/Total
Input
Output
Shuffle Read
Shuffle Write
6 collect at <console>:19 +details 2014/12/01 16:18:24 28 ms 2/2 552.0 B
4 collect at <console>:19 +details 2014/12/01 16:18:22 45 ms 2/2
2 collect at <console>:19 +details 2014/12/01 16:18:07 69 ms 2/2
1 map at <console>:16 +details 2014/12/01 16:18:07 76 ms 2/2 254.0 B 737.0 B
0 count at <console>:15 +details 2014/12/01 16:17:40 0.2 s 2/2 254.0 B


Spark shell - Storage
ec2-54-148-231-117.us-west-2.compute.amazonaws.com:4040/storage/
Spark
Jobs
Stages
Storage
Environment
Executors
Spark shell application UI
Storage
RDD Name
Storage Level
Cached Partitions
Fraction Cached
Size in Memory
Size in Tachyon
Size on Disk
5 Memory Deserialized 1x Replicated 2 100% 552.0 B 0.0 B 0.0 B

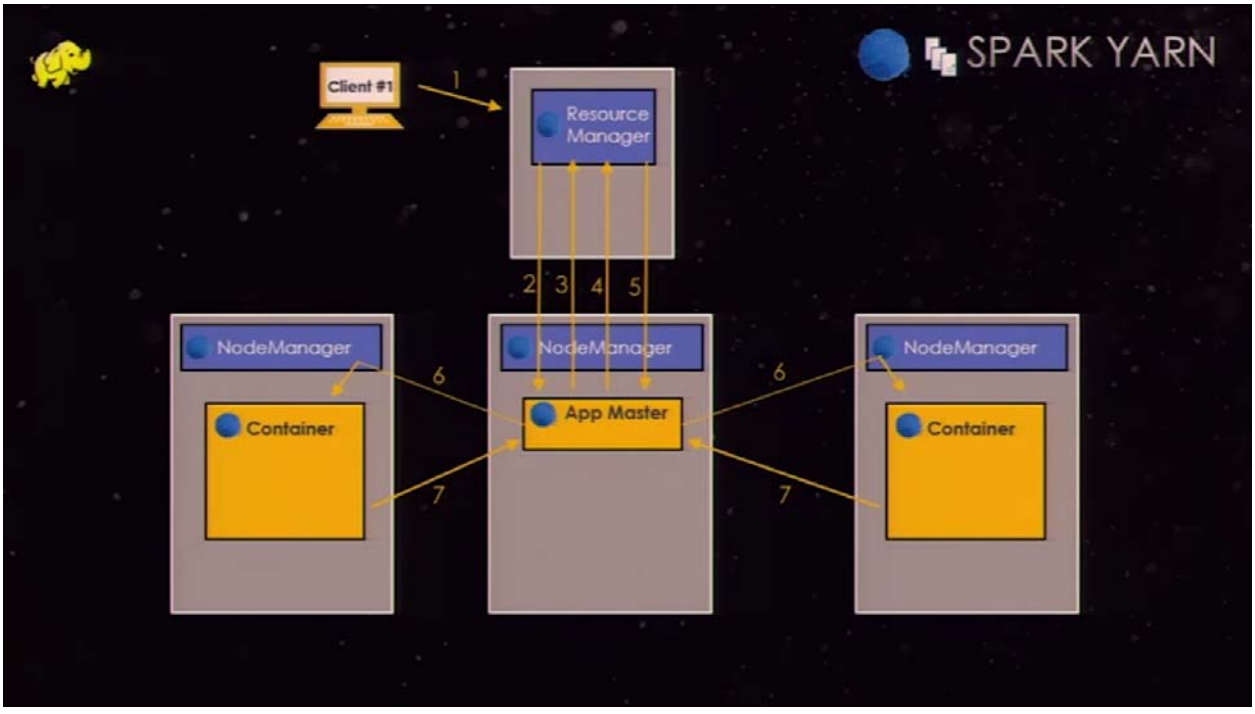

Spark shell - RDD Storage
ec2-54-148-231-117.us-west-2.compute.amazonaws.com:4040/storage/rdd/?id=5
Spark
Jobs
Stages
Storage
Environment
Executors
Spark shell application UI
RDD Storage Info for 5
Storage Level: Memory Deserialized 1x Replicated
Cached Partitions: 2
Total Partitions: 2
Memory Size: 552.0 B
Disk Size: 0.0 B
Data Distribution on 1 Executors
Host
Memory Usage
Disk Usage
localhost:38329
552.0 B (265.4 MB Remaining)
0.0 B
2 Partitions
Block Name
Storage Level
Size in Memory
Size on Disk
Executors
rdd_5_0
Memory Deserialized 1x Replicated
424.0 B
0.0 B
localhost:38329
rdd_5_1
Memory Deserialized 1x Replicated
128.0 B
0.0 B
localhost:38329

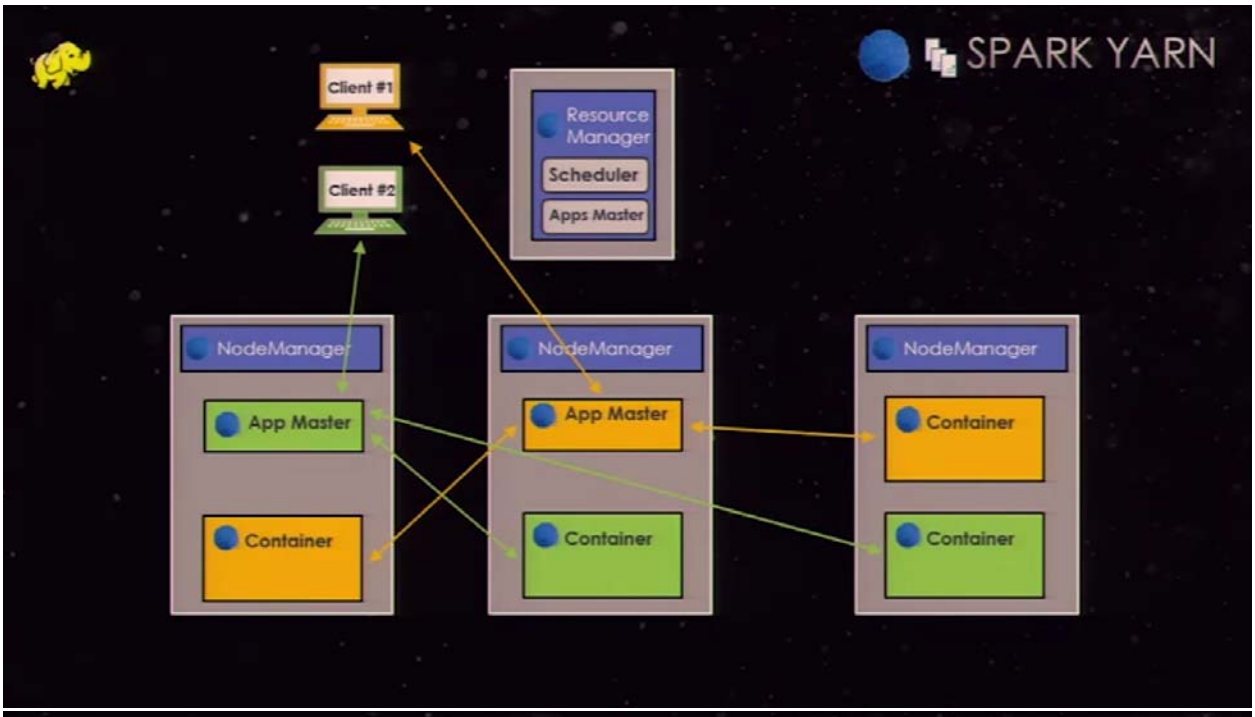

Spark shell - Environment
ec2-54-148-231-117.us-west-2.compute.amazonaws.com:4040/environment/
Spark
Jobs
Stages
Storage
Environment
Executors
Spark shell application UI
Environment
Runtime Information
Name
Value
Java Home
/usr/java/jdk1.7.0_67/jre
Java Version
1.7.0_67 (Oracle Corporation)
Scala Version
version 2.10.4
Spark Properties
Name
Value
spark.app.id
local-1417468637156
spark.app.name
Spark shell
spark.driver.host
ip-10-0-125-125.us-west-2.compute.internal
spark.driver.port
59091
spark.executor.id
driver
spark.fileserver.uri
http://10.0.125.125:56999
spark.jars
spark.master
local[*]
spark.repl.class.uri
http://10.0.125.125:57870
spark.scheduler.mode
FIFO
spark.tachyonStore.folderName
spark-a5c91951-a6b4-4425-badc-a1e2e9146a70
System Properties
Name
Value

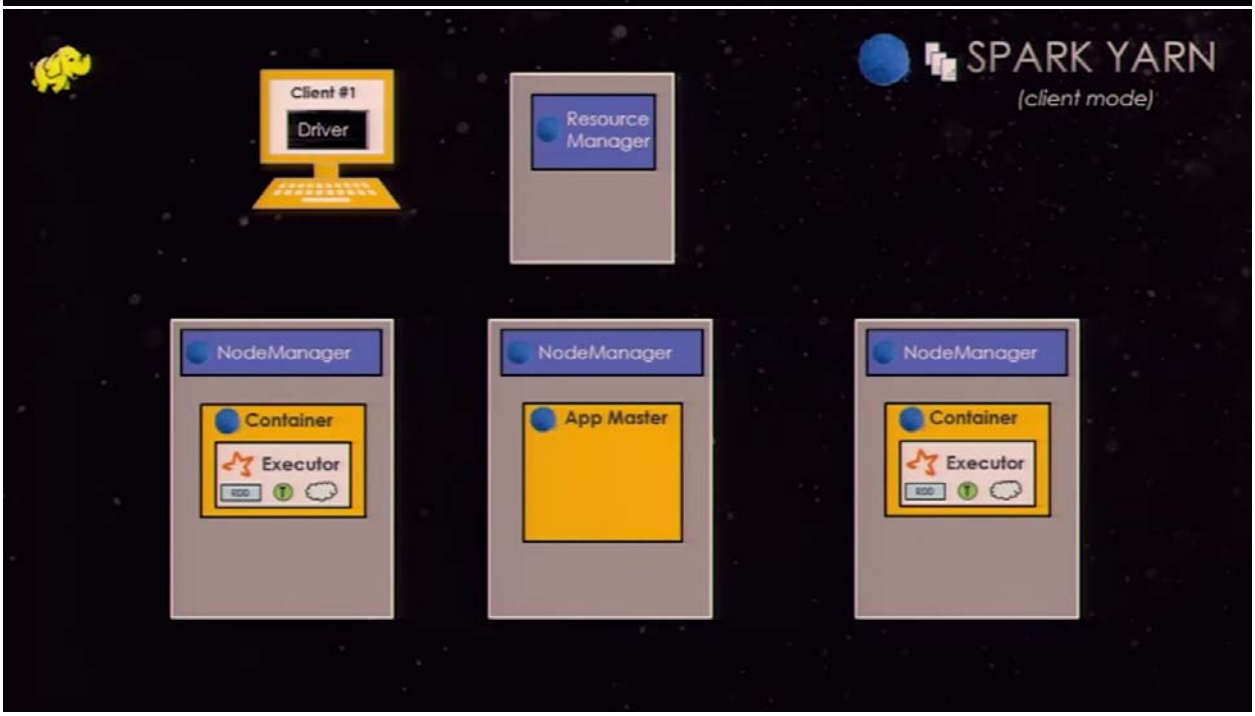

Spark shell - Executors (1)
ec2-54-148-231-117.us-west-2.compute.amazonaws.com:4040/executors/
Spark
Jobs
Stages
Storage
Environment
Executors
Spark shell application UI
Executors (1)
Memory: 552.0 B Used (265.4 MB Total)
Disk: 0.0 B Used
Executor ID
Address
RDD Blocks
Memory Used
Disk Used
Active Tasks
Failed Tasks
Complete Tasks
Total Tasks
Task Time
Input
Shuffle Read
Shuffle Write
Thread Dump
<driver>
localhost:38329
2
552.0 B / 265.4 MB
0.0 B
0
0
10
10
740 ms
1060.0 B
0.0 B
737.0 B
Thread Dump

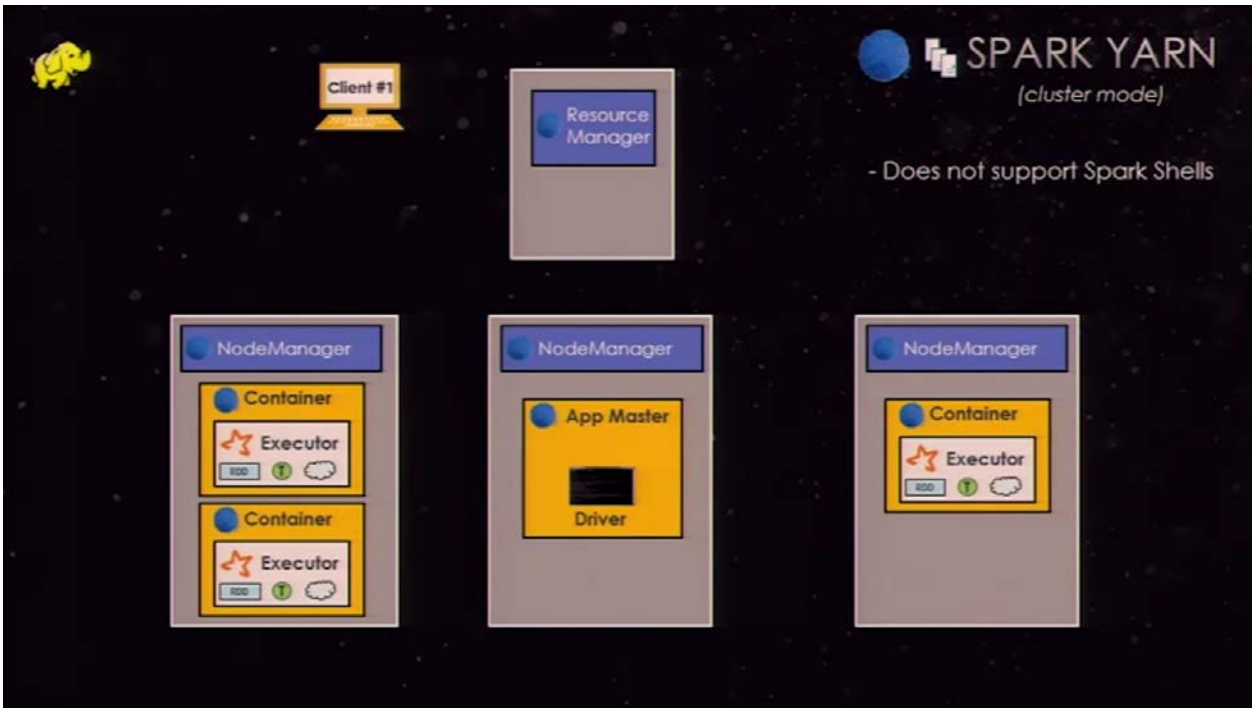

Spark shell - Thread dump
ec2-54-148-231-117.us-west-2.compute.amazonaws.com:4040/executors/threadDump/?executorId=<driver>
Spark
Jobs
Stages
Storage
Environment
Executors
Spark shell application UI
Thread dump for executor <driver>
Updated at 2014/12/01 16:57:39
Expand All
Thread 1: main (RUNNABLE)
Thread 2: Reference Handler (WAITING)
Thread 3: Finalizer (WAITING)
Thread 4: Signal Dispatcher (RUNNABLE)
Thread 11: qtp1844705036-11 (TIMED_WAITING)
Thread 12: qtp1844705036-12 (TIMED_WAITING)
Thread 13: qtp1844705036-13 (TIMED_WAITING)
Thread 14: qtp1844705036-14 Acceptor0 SocketConnector@0.0.0.0:57870 (RUNNABLE)
Thread 15: qtp1844705036-15 (TIMED_WAITING)
Thread 16: qtp1844705036-16 (TIMED_WAITING)
Thread 17: qtp1844705036-17 (TIMED_WAITING)
Thread 18: qtp1844705036-18 (TIMED_WAITING)


YARN MODE


SPARK YARN
Client #1
1
Resource Manager
2
3
4
5
NodeManager
Container
6
7
NodeManager
App Master
6
7
NodeManager
Container


SPARK YARN
Client #1
Client #2
Resource Manager
Scheduler
Apps Master
NodeManager
App Master
Container
NodeManager
App Master
Container
NodeManager
Container
Container


SPARK YARN
(client mode)
Client #1
Driver
Resource Manager
NodeManager
Container
Executor
NodeManager
App Master
NodeManager
Container
Executor



YARN

YARN settings

`--num-executors:` controls how many executors will be allocated

`--executor-memory:` RAM for each executor

`--executor-cores:` CPU cores for each executor

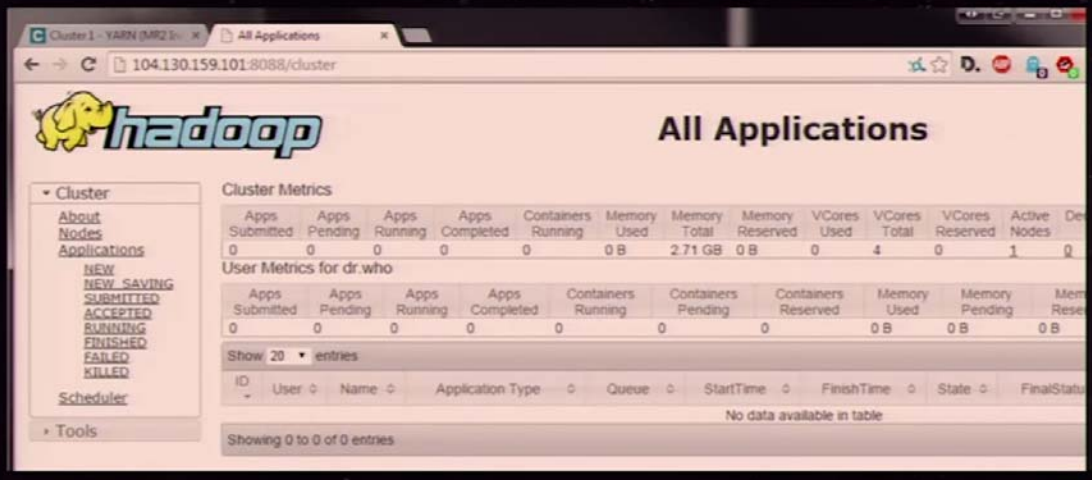
Dynamic Allocation:

```
spark.dynamicAllocation.enabled
spark.dynamicAllocation.minExecutors
spark.dynamicAllocation.maxExecutors
spark.dynamicAllocation.sustainedSchedulerBacklogTimeout (N)
spark.dynamicAllocation.schedulerBacklogTimeout (M)
spark.dynamicAllocation.executorIdleTimeout (K)
```

https://github.com/apache/spark/blob/master/core/src/main/scala/org/apache/spark/ExecutorAllocationManager.scala

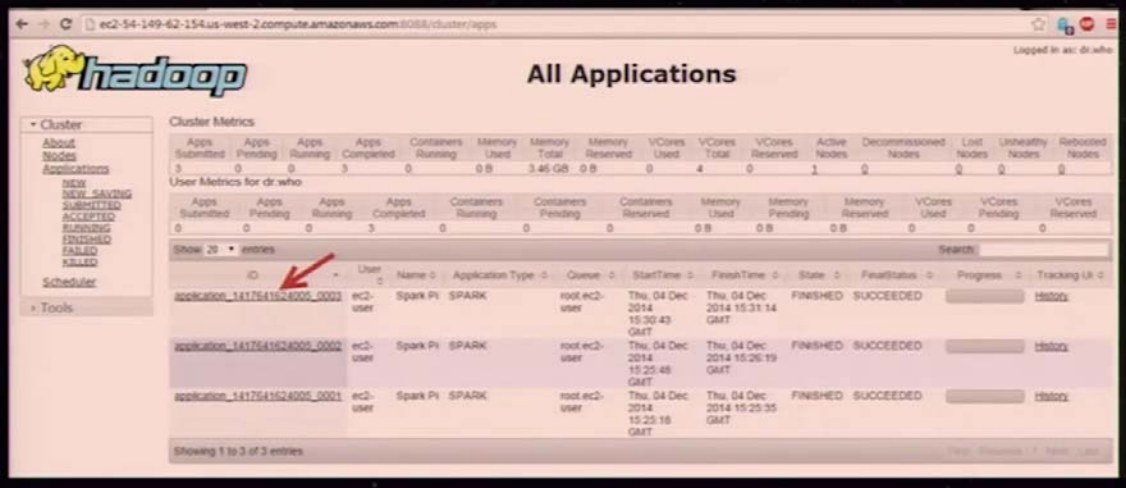
YARN resource manager UI: http://<ip address>:8088

(No apps running)

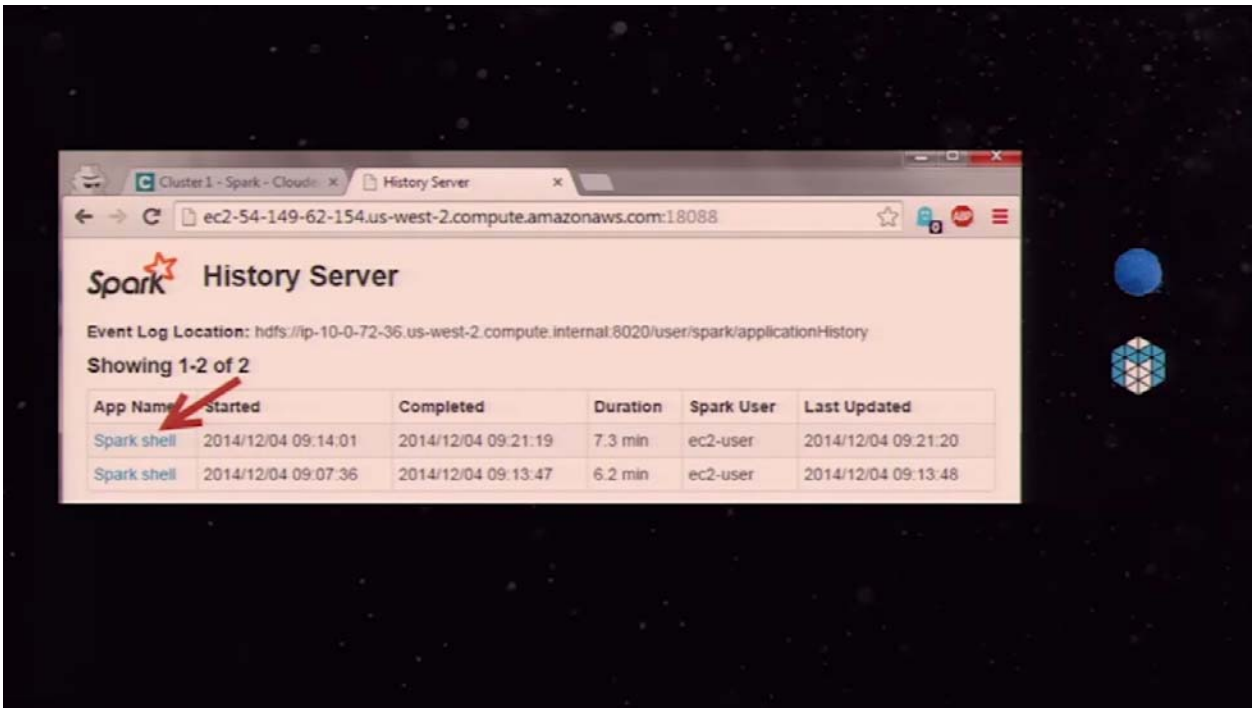



```
[ec2-user@ip-10-0-72-36 ~]$ spark-submit --class
org.apache.spark.examples.SparkPi --deploy-mode client --master yarn
/opt/cloudera/parcels/CDH-5.2.1-1.cdh5.2.1.p0.12/jars/spark-
examples-1.1.0-cdh5.2.1-hadoop2.5.0-cdh5.2.1.jar 10
```

App running in client mode




```
[ec2-user@ip-10-0-72-36 ~]$ spark-submit --class
org.apache.spark.examples.SparkPi --deploy-mode cluster --master
yarn /opt/cloudera/parcels/CDH-5.2.1-1.cdh5.2.1.p0.12/jars/spark-
examples-1.1.0-cdh5.2.1-hadoop2.5.0-cdh5.2.1.jar 10
```



PLUGGABLE RESOURCE MANAGEMENT

	Spark Central Master	Who starts Executors?	Tasks run in
Local	[none]	Human being	Executor
Standalone	Standalone Master	Worker JVM	Executor
YARN	YARN App Master	Node Manager	Executor
Mesos	Mesos Master	Mesos Slave	Executor

DEPLOYING AN APP TO THE CLUSTER

`spark-submit` provides a uniform interface for submitting jobs across all cluster managers

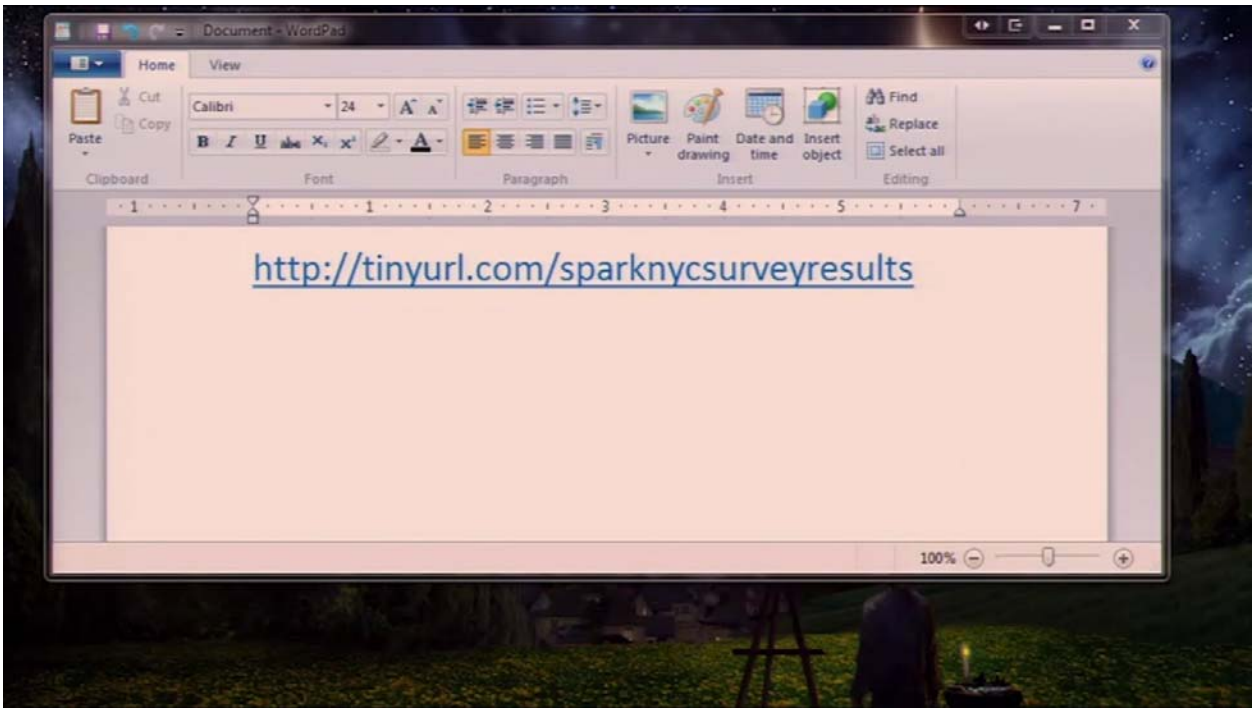
```
bin/spark-submit --master spark://host:7077
                 --executor-memory 10g
                 my_script.py
```

Table 7-2. Possible values for the `--master` flag in `spark-submit`

Value	Explanation
spark://host:port	Connect to a Spark Standalone master at the specified port. By default Spark Standalone master's listen on port 7077 for submitted jobs.
mesos://host:port	Connect to a Mesos cluster master at the specified port. By default Mesos masters listen on port 5050 for submitted jobs.
yarn	Indicates submission to YARN cluster. When running on YARN you'll need to export `HADOOP_CONF_DIR` to point the location of your Hadoop configuration directory.
local	Run in local mode with a single core.
local[N]	Run in local mode with N cores.
local[*]	Run in local mode and use as many cores as the machine has.

Source: Learning Spark

HDFS	**121**	69.9%
MapReduce	**102**	59%
YARN	**66**	38.2%
Mesos	**7**	4%
Cascading	**7**	4%
Kafka	**43**	24.9%
Storm	**20**	11.6%
Flume	**23**	13.3%
HBase	**35**	20.2%
Cassandra	**42**	24.3%
Hive	**69**	39.9%
Impala	**23**	13.3%
Pig	**38**	22%
Parquet	**33**	19.1%
ZooKeeper	**48**	27.7%
MongoDB	**41**	23.7%
Couchbase	**2**	1.2%
Neo4j	**14**	8.1%
Titan	**3**	1.7%



AGENDA

Before Lunch

- History of Spark
- RDD fundamentals
- Spark Runtime Architecture Integration with Resource Managers (Standalone, YARN)
- GUIs
- Lab: DevOps 101

After Lunch

- Memory and Persistence
- Jobs -> Stages -> Tasks
- Broadcast Variables and Accumulators
- PySpark
- DevOps 102
- Shuffle
- Spark Streaming

databricks

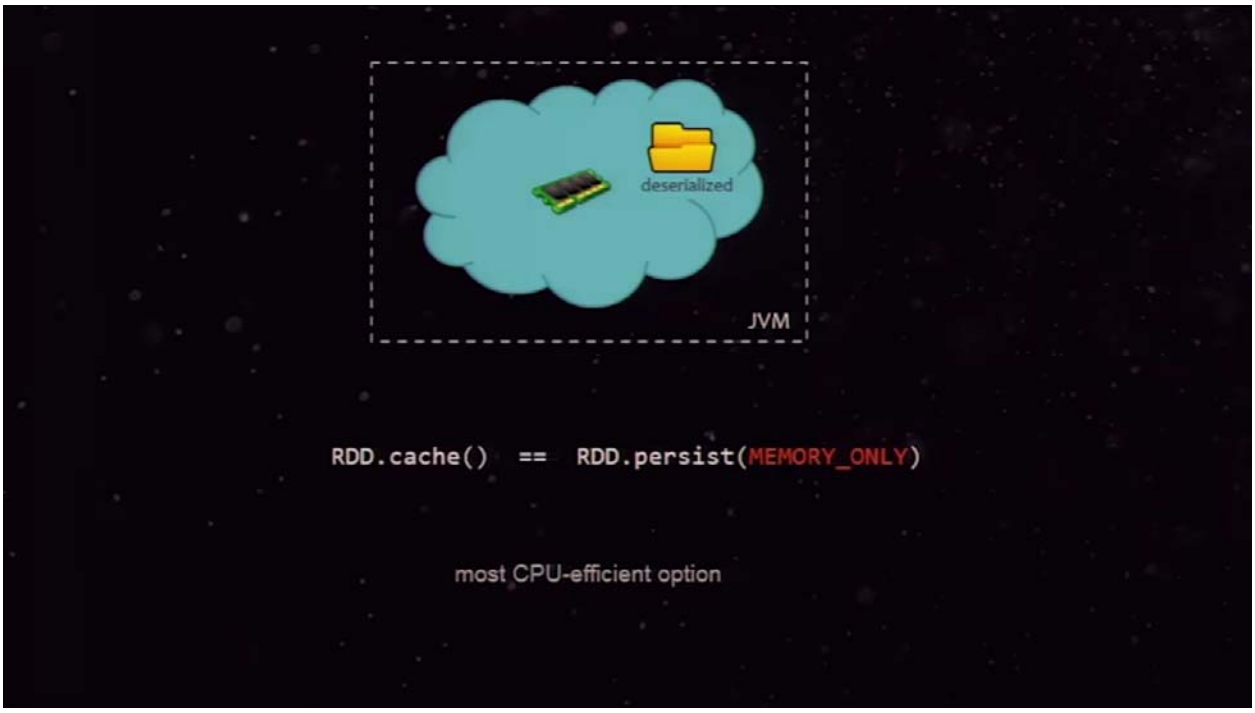
Recommended to use at most only 75% of a machine's memory for Spark

Minimum Executor heap size should be 8 GB

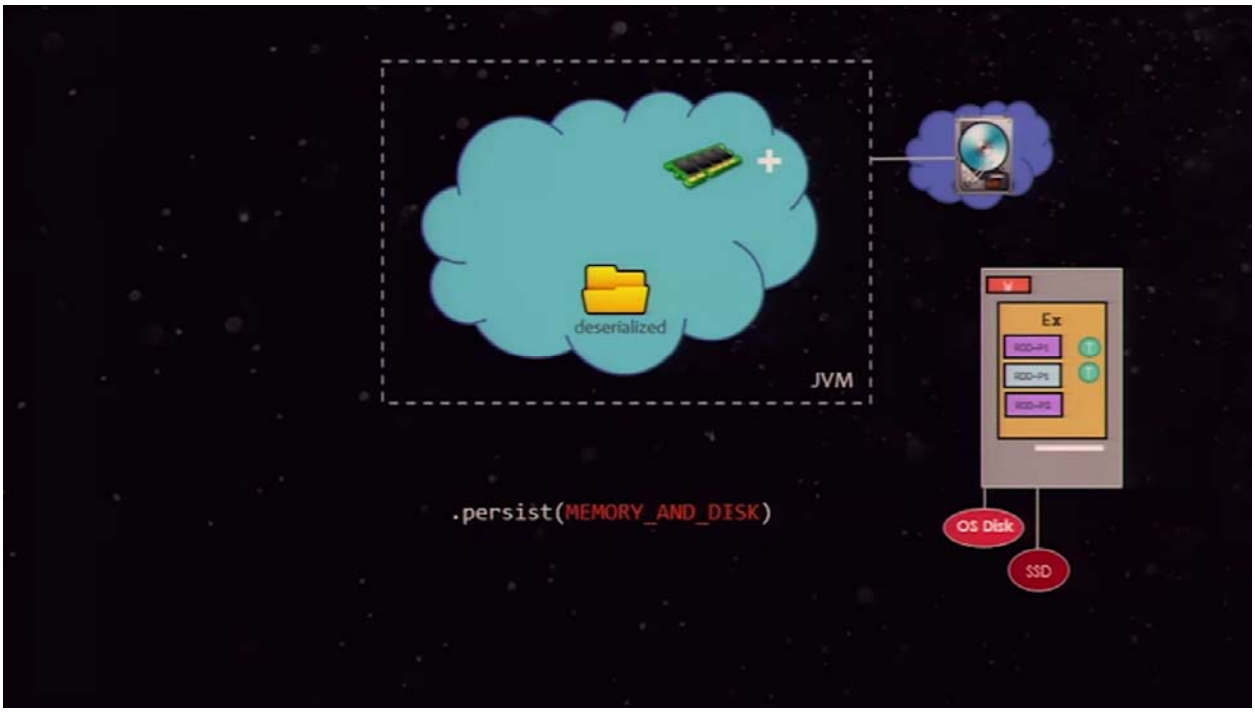
Max Executor heap size depends... maybe 40 GB (watch GC)

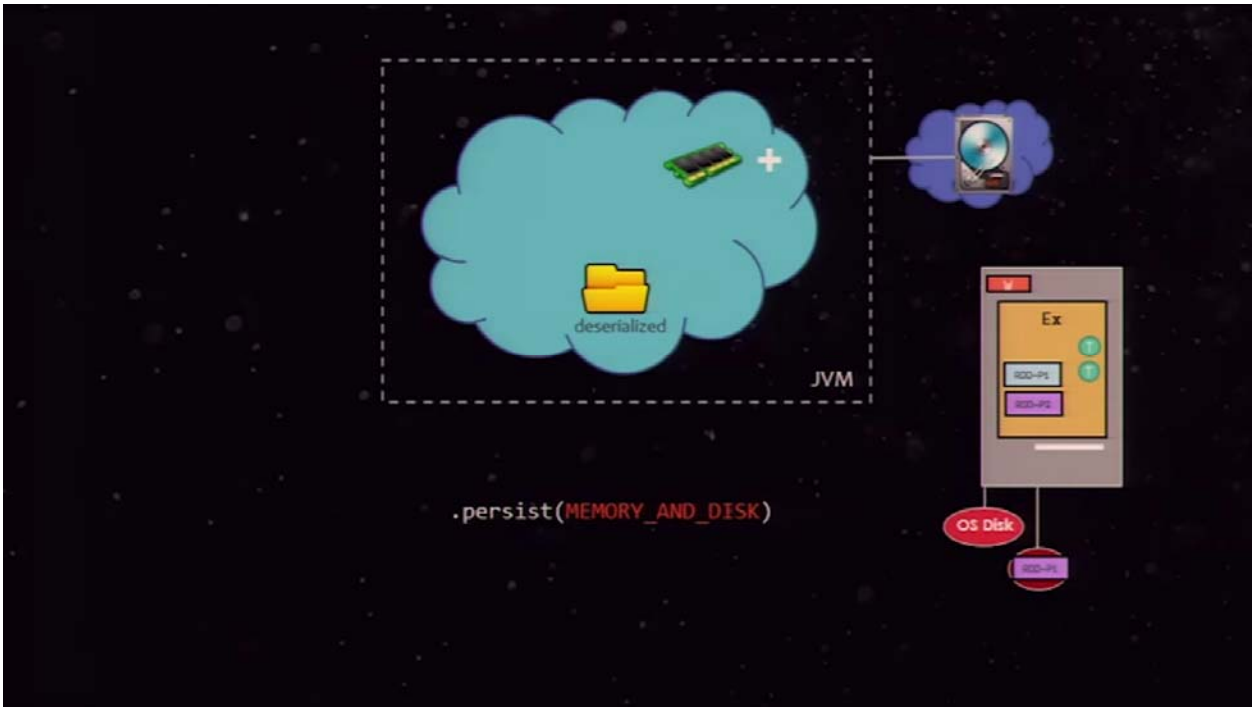
Memory usage is greatly affected by storage level and serialization format

Vs.

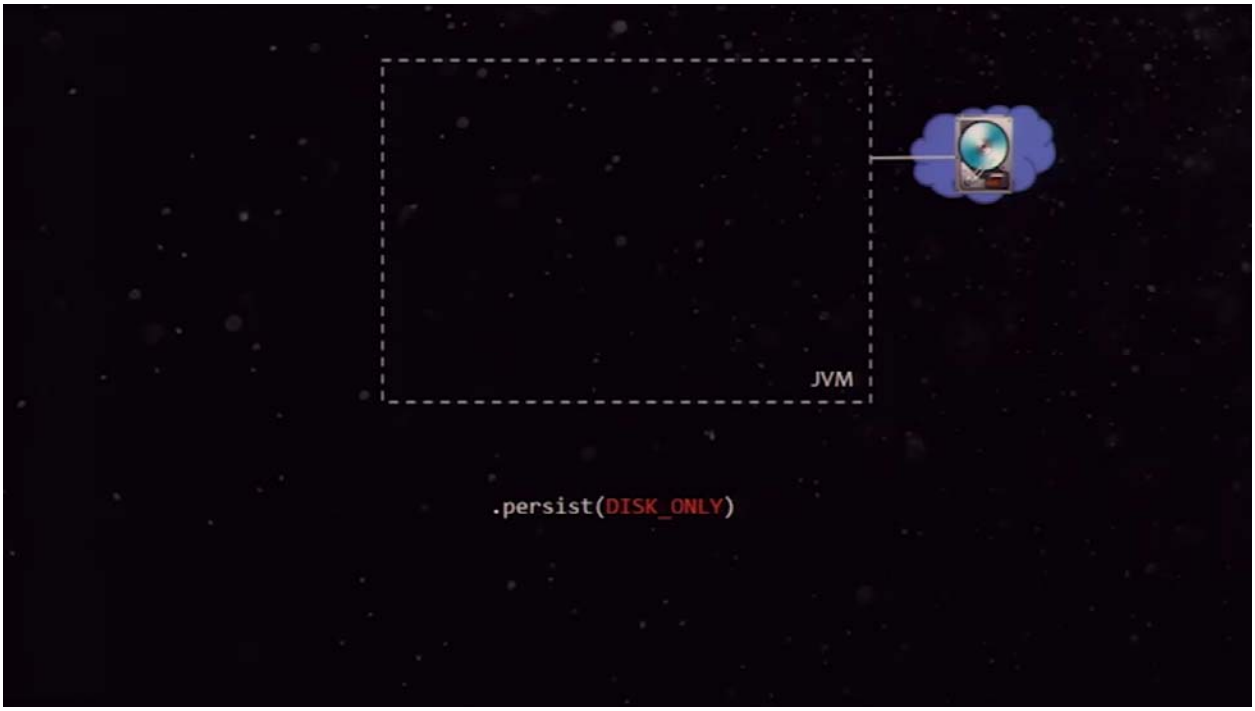

deserialized
JVM
RDD.cache() == RDD.persist(MEMORY_ONLY)
most CPU-efficient option

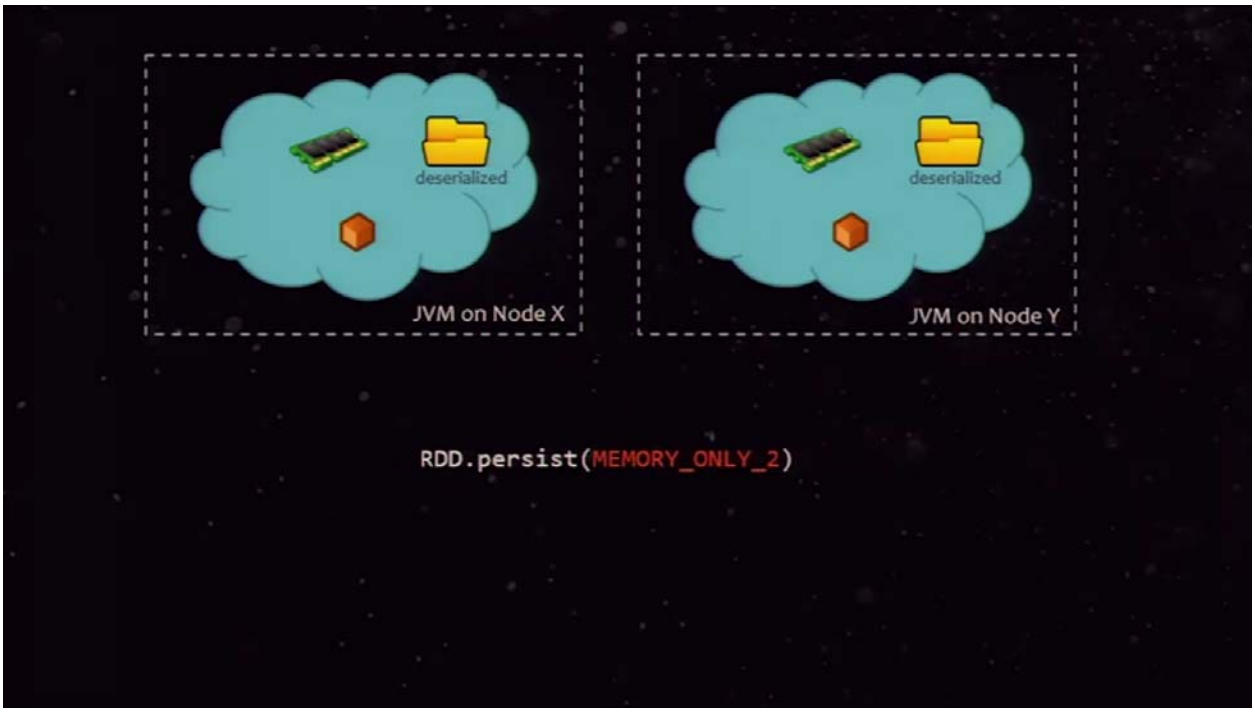

serialized
JVM
RDD.persist(MEMORY_ONLY_SER)

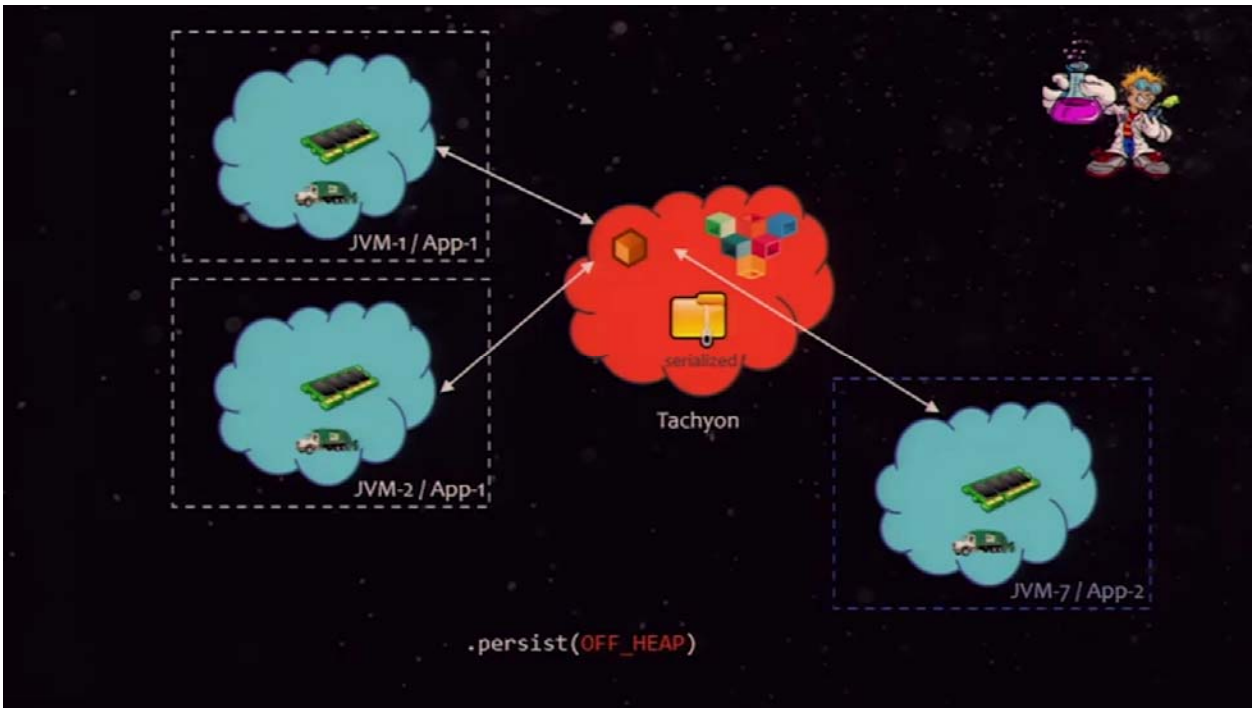

deserialized
JVM
.persist(MEMORY_AND_DISK)
Ex
OS Disk
SSD


deserialized
JVM
.persist(MEMORY_AND_DISK)
Ex
OS Disk


serialized
JVM
.persist(MEMORY_AND_DISK_SER)

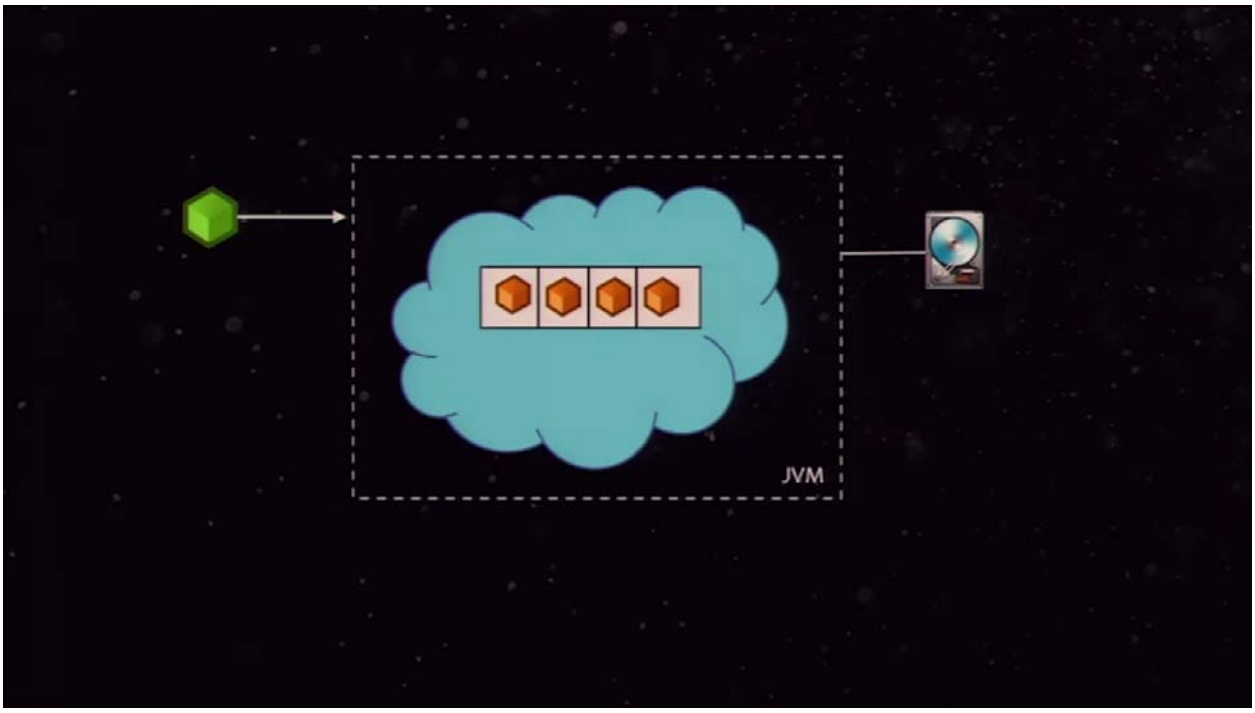

JVM
.persist(DISK_ONLY)

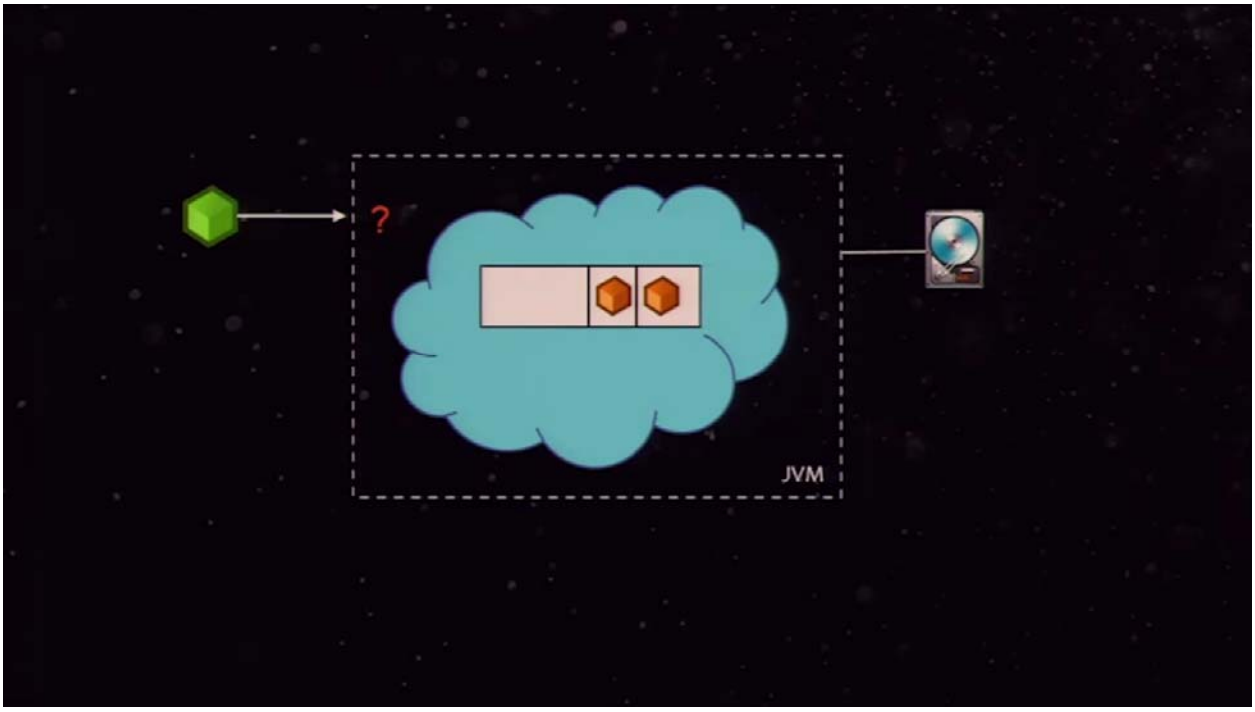

deserialized
JVM on Node X
deserialized
JVM on Node Y
RDD.persist(MEMORY_ONLY_2)

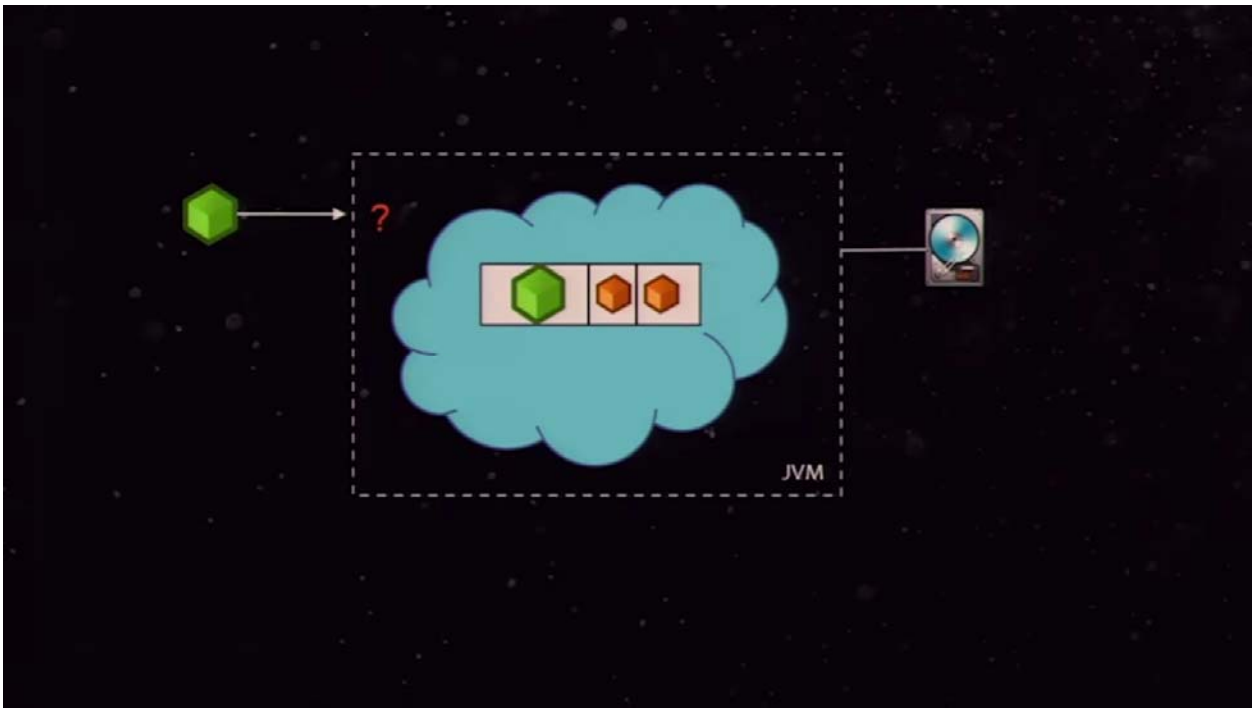

JVM-1 / App-1
JVM-2 / App-1
serialized
Tachyon
JVM-7 / App-2
.persist(OFF_HEAP)


JVM
.unpersist()


JVM
.unpersist()


JVM


?
JVM



?

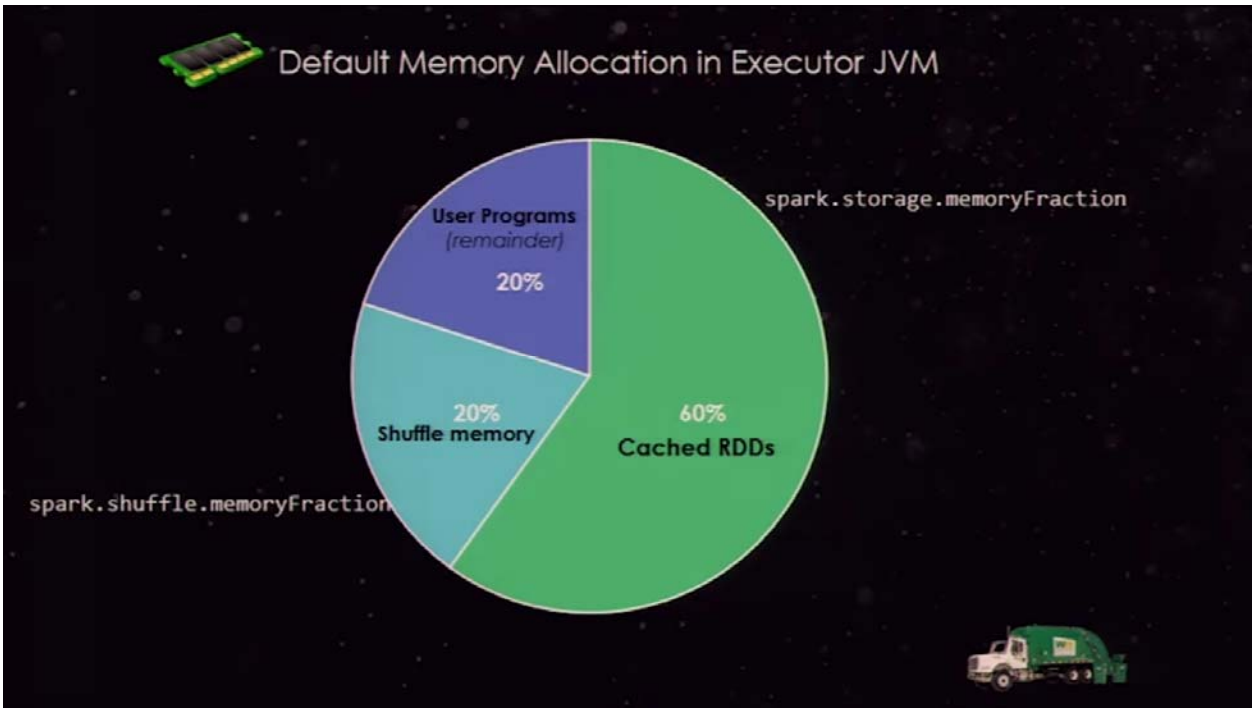
- If RDD fits in memory, choose MEMORY_ONLY
- If not, use MEMORY_ONLY_SER w/ fast serialization library
- Don't spill to disk unless functions that computed the datasets are very expensive or they filter a large amount of data. (recomputing may be as fast as reading from disk)
- Use replicated storage levels sparingly and only if you want fast fault recovery (maybe to serve requests from a web app)

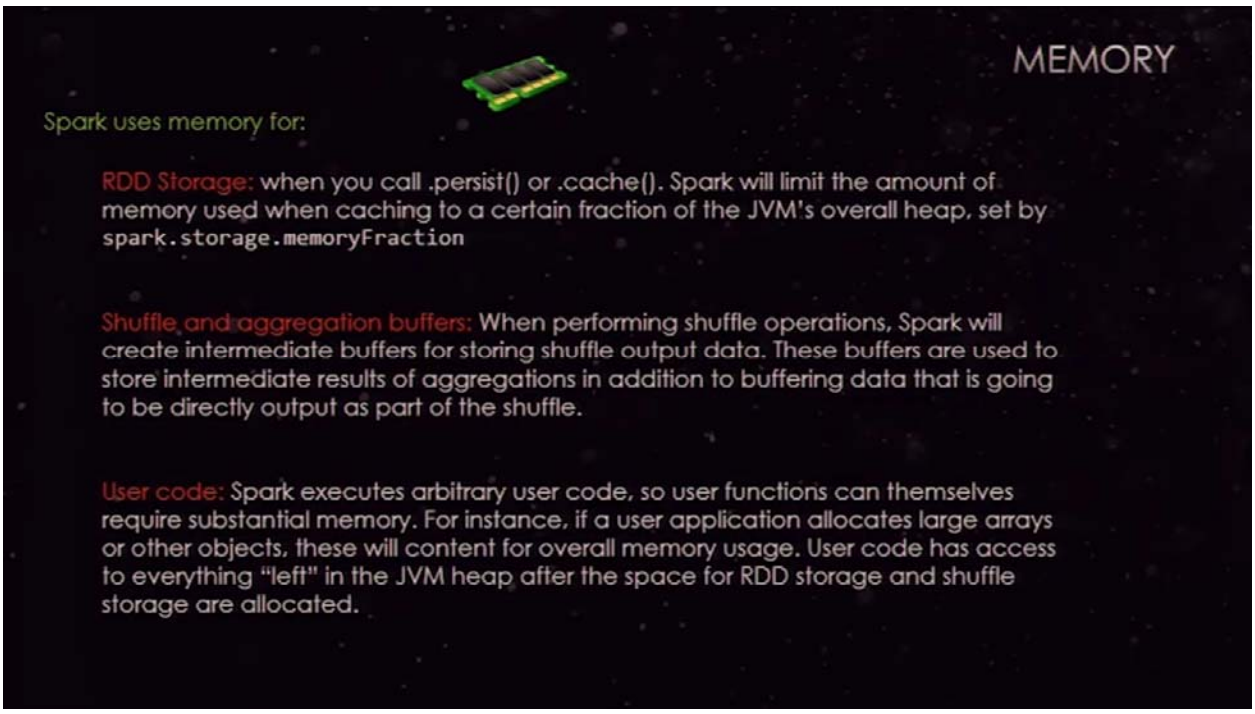
Remember!

Intermediate data is automatically persisted during shuffle operations

=

PySpark: stored objects will always be serialized with Pickle library, so it does not matter whether you choose a serialized level.


Default Memory Allocation in Executor JVM
User Programs
(remainder)
20%
spark.storage.memoryFraction
20%
Shuffle memory
60%
Cached RDDs
spark.shuffle.memoryFraction


MEMORY
Spark uses memory for:
RDD Storage: when you call .persist() or .cache(). Spark will limit the amount of memory used when caching to a certain fraction of the JVM's overall heap, set by spark.storage.memoryFraction
Shuffle and aggregation buffers: When performing shuffle operations, Spark will create intermediate buffers for storing shuffle output data. These buffers are used to store intermediate results of aggregations in addition to buffering data that is going to be directly output as part of the shuffle.
User code: Spark executes arbitrary user code, so user functions can themselves require substantial memory. For instance, if a user application allocates large arrays or other objects, these will content for overall memory usage. User code has access to everything "left" in the JVM heap after the space for RDD storage and shuffle storage are allocated.

DETERMINING MEMORY CONSUMPTION

1. Create an RDD
2. Put it into cache
3. Look at SparkContext logs on the driver program or Spark UI

logs will tell you how much memory each partition is consuming, which you can aggregate to get the total size of the RDD

```
INFO BlockManagerMasterActor: Added rdd_0_1 in memory on mbk.local:50311 (size: 717.5 KB, free: 332.3 MB)
```

DATA SERIALIZATION

databricks

SERIALIZATION

Serialization is used when:

Transferring data over the network

Spilling data to disk

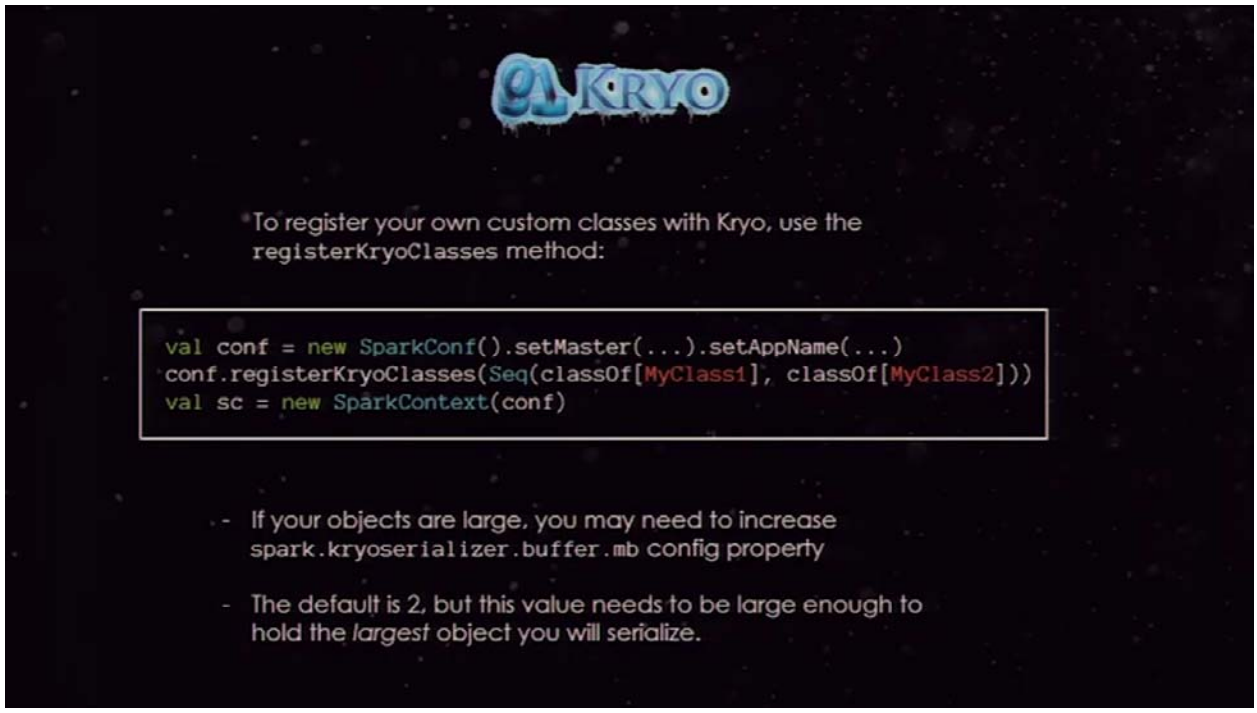
Caching to memory serialized

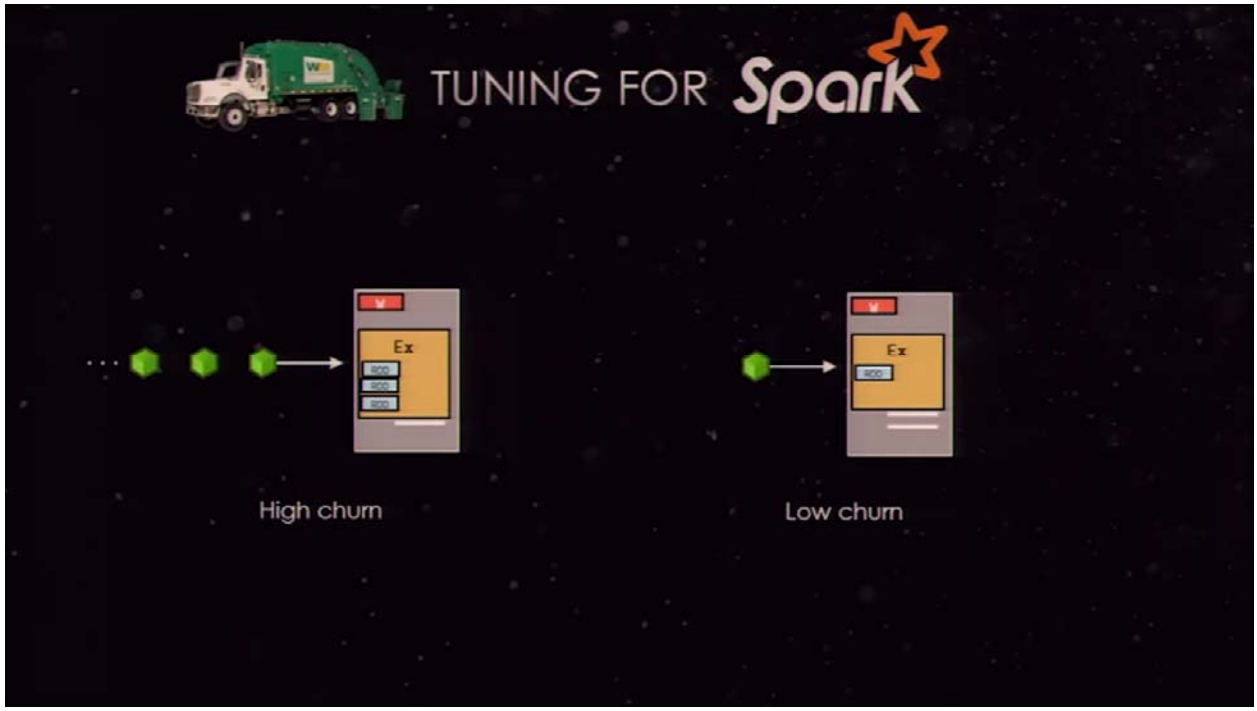
Broadcasting variables

Java serialization vs. Kryo serialization

- Uses Java's `ObjectOutputStream` framework
- Works with any class you create that implements `java.io.Serializable`
- You can control the performance of serialization more closely by extending `java.io.Externalizable`
- Flexible, but quite slow
- Leads to large serialized formats for many classes

- Recommended serialization for production apps
- Use Kyro version 2 for speedy serialization (10x) and more compactness
- Does not support all `Serializable` types
- Requires you to register the classes you'll use in advance
- If set, will be used for serializing shuffle data between nodes and also serializing RDDs to disk


01 KRYO
To register your own custom classes with Kryo, use the registerKryoClasses method:
val conf = new SparkConf().setMaster(...).setAppName(...)
conf.registerKryoClasses(Seq(classOf[MyClass1], classOf[MyClass2]))
val sc = new SparkContext(conf)
- If your objects are large, you may need to increase spark.kryoserializer.buffer.mb config property
- The default is 2, but this value needs to be large enough to hold the largest object you will serialize.


TUNING FOR Spark
Ex
Ex
High churn
Low churn

TUNING FOR Spark

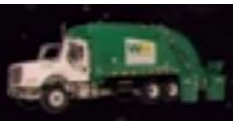
Cost of GC is proportional to the # of Java objects

(so use an array of Ints instead of a `LinkedList`)

High churn

To measure GC impact:

```
-verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps
```



TUNING

Parallel GC

-XX:+UseParallelGC
-XX:ParallelGCThreads=<#>

- Uses multiple threads to do young gen GC
- Will default to Serial on single core machines
- Aka "throughput collector"
- Good for when a lot of work is needed and long pauses are acceptable
- Use cases: batch processing

Parallel Old GC

-XX:+UseParallelOldGC

- Uses multiple threads to do both young gen and old gen GC
- Also a multithreading compacting collector
- HotSpot does compaction only in old gen

CMS GC

-XX:+UseConcMarkSweepGC
-XX:ParallelCMSThreads=<#>

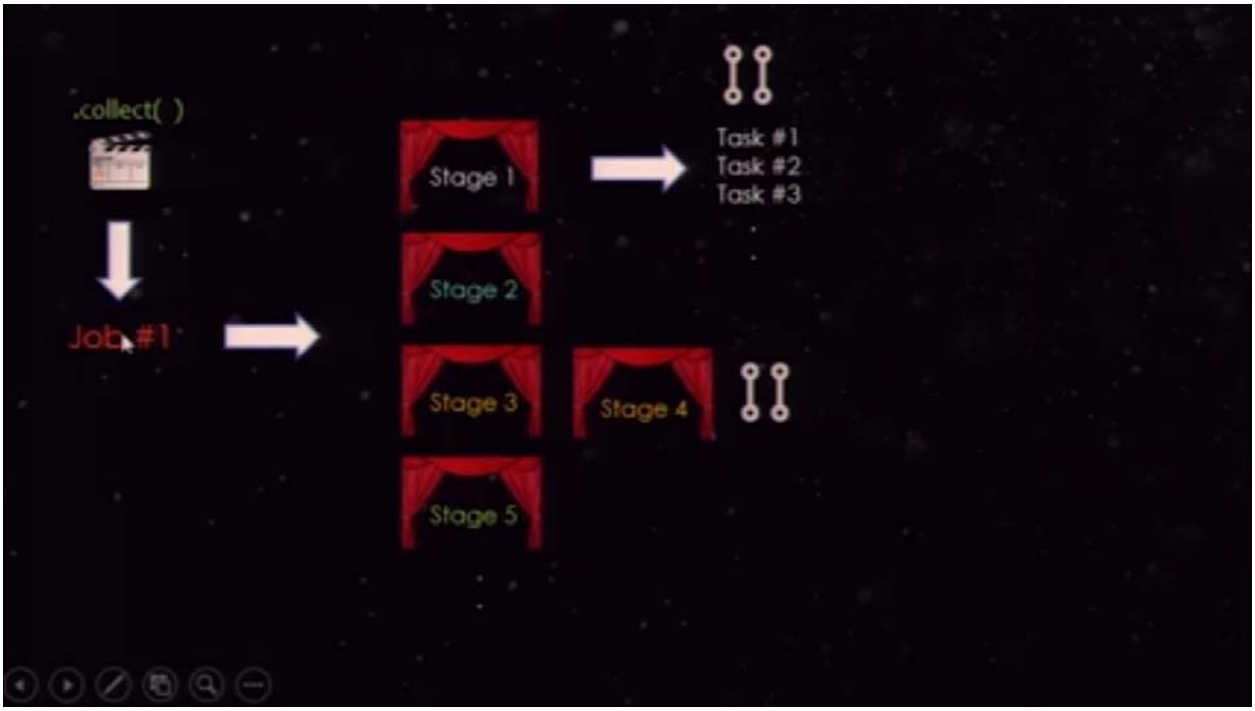
- Concurrent Mark Sweep aka "Concurrent low pause collector"
- Tries to minimize pauses due to GC by doing most of the work concurrently with application threads
- Uses same algorithm on young gen as parallel collector
- Use cases:

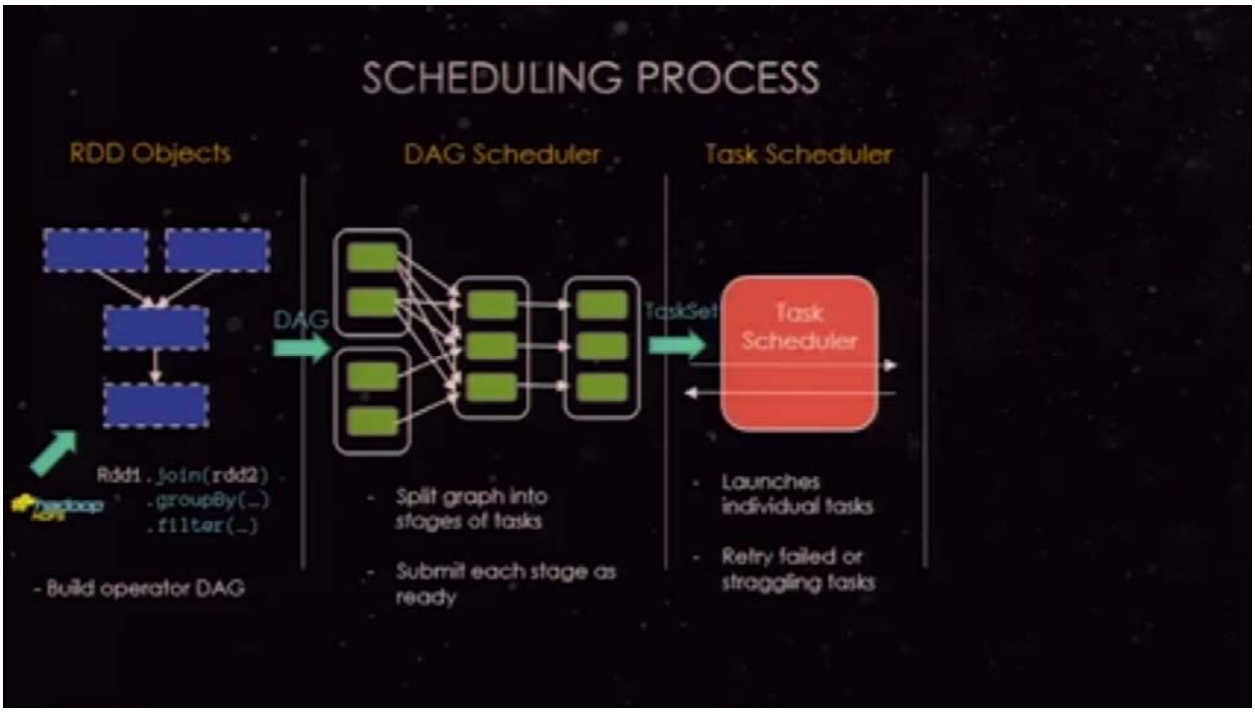
G1 GC

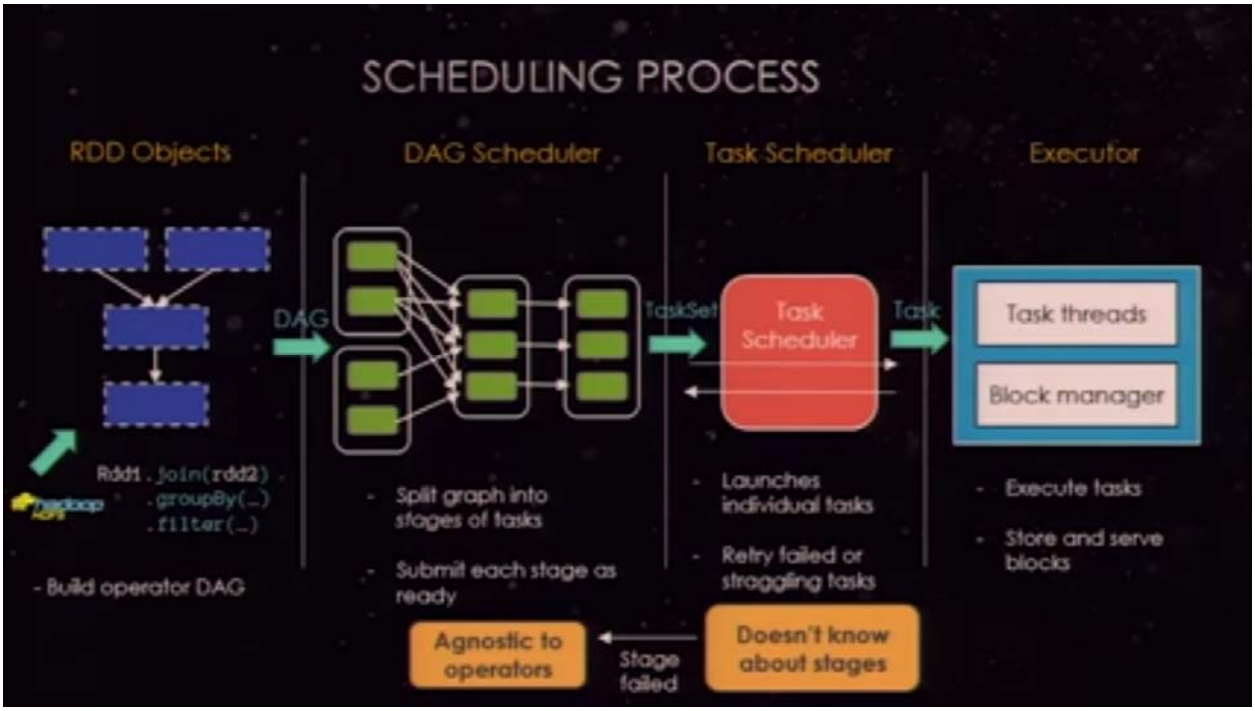
-XX:+UseG1GC

- Garbage First is available starting Java 7
- Designed to be long term replacement for CMS
- Is a parallel, concurrent and incrementally compacting low-pause GC


?
JOBS → STAGES → TASKS
databricks


.collect( )
Job #1
Stage 1
Stage 2
Stage 3
Stage 4
Stage 5
Task #1
Task #2
Task #3


SCHEDULING PROCESS
RDD Objects
DAG Scheduler
Task Scheduler
DAG
TaskSet
Task Scheduler
Rdd1.join(rdd2)
.groupBy(...)
.filter(...)
- Build operator DAG
- Split graph into stages of tasks
- Submit each stage as ready
- Launches individual tasks
- Retry failed or straggling tasks


SCHEDULING PROCESS
RDD Objects
DAG Scheduler
Task Scheduler
Executor
DAG
TaskSet
Task Scheduler
Task
Task threads
Block manager
Rdd1.join(rdd2)
.groupBy(...)
.filter(...)
- Build operator DAG
- Split graph into stages of tasks
- Submit each stage as ready
- Launches individual tasks
- Retry failed or straggling tasks
- Execute tasks
- Store and serve blocks
Agnostic to operators
Stage failed
Doesn't know about stages

LINEAGE

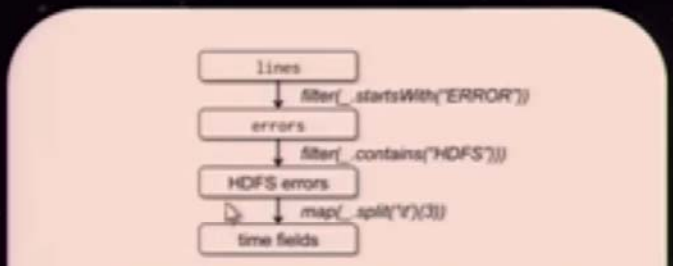


Figure 1: Lineage graph for the third query in our example. Boxes represent RDDs and arrows represent transformations.

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
errors.persist()
```

LINEAGE

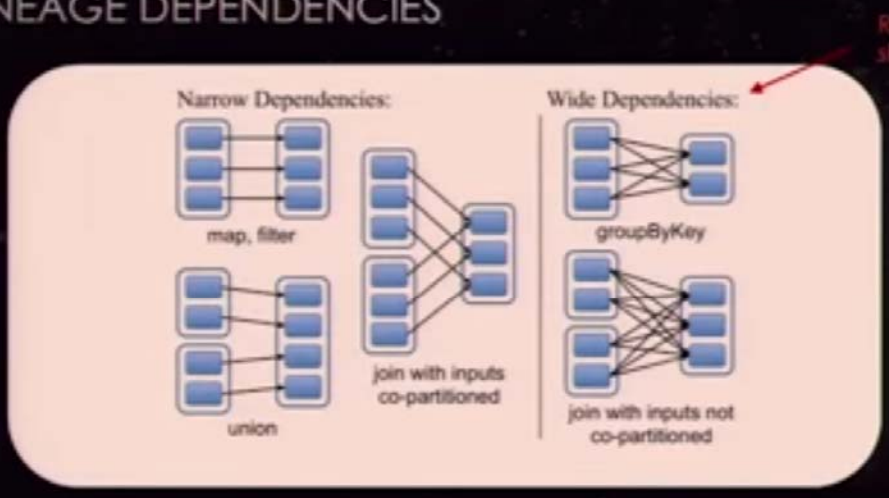
"One of the challenges in providing RDDs as an abstraction is choosing a representation for them that can track lineage across a wide range of transformations."

"The most interesting question in designing this interface is how to represent dependencies between RDDs."

"We found it both sufficient and useful to classify dependencies into two types:

- narrow dependencies, where each partition of the parent RDD is used by at most one partition of the child RDD
- wide dependencies, where multiple child partitions may depend on it."

LINEAGE DEPENDENCIES



Examples of narrow and wide dependencies.

Each box is an RDD, with partitions shown as shaded rectangles.