DEVOPS ADVANCED CLASS

March 2015: Spark Summit East 2015

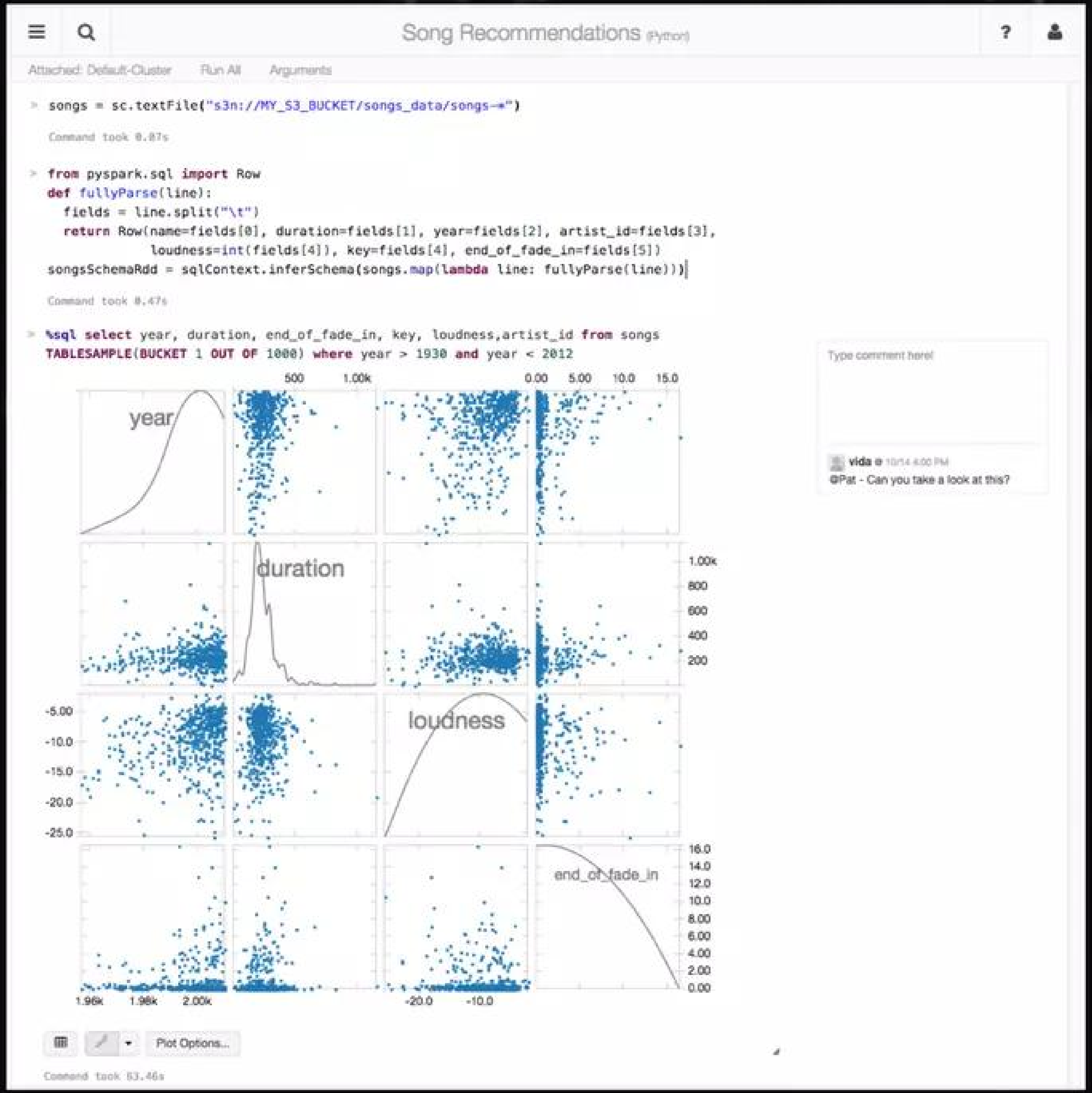
http://slideshare.net/databricks

www.linkedin.com/in/blueplastic

databricks

making big data simple

- Founded in late 2013
- by the creators of Apache Spark
- Original team from UC Berkeley AMPLab
- Raised $47 Million in 2 rounds
- ~50 employees
- We're hiring! (http://databricks.workable.com)
- Level 2/3 support partnerships with
  - Cloudera
  - Hortonworks
  - MapR
  - DataStax



Databricks Cloud:

"A unified platform for building Big Data pipelines – from ETL to Exploration and Dashboards, to Advanced Analytics and Data Products."

The Databricks team contributed more than **75%** of the code added to Spark in the past year

databricks

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

Some slides will be skipped

Please keep Q&A low during class

(5pm – 5:30pm for Q&A with instructor)

2 anonymous surveys: Pre and Post class

Lunch: noon – 1pm

2 breaks (before lunch and after lunch)

Algorithms
Machines
People

amp lab @ Berkeley University of California

- AMPLab project was launched in Jan 2011, 6 year planned duration
- Personnel: ~65 students, postdocs, faculty & staff
- Funding from Government/Industry partnership, NSF Award, Darpa, DoE, 20+ companies
- Created BDAS, Mesos, SNAP. Upcoming projects: Succinct & Velox.

"Unknown to most of the world, the University of California, Berkeley's AMPLab has already left an indelible mark on the world of information technology, and even the web. But we haven't yet experienced the full impact of the group[…] Not even close"

- Derrick Harris, GigaOm, Aug 2014

Scheduling

Monitoring

Distributing


hadoop
HDFS
FILE://
S3
Distributions:
- CDH
- HDP
- MapR
- DSE
FLUME
SQL
Streaming
DataFrames API
Tachyon
GraphX
MLlib
BlinkDB
mongoDB
RDBMS
Neo4j
APACHE
HBASE
Hadoop Input Format
Apps

Rick Richardson
@eigenrick

Follow

Just realized Berkeley AMPLab is the Xerox PARC of this century. #sparksummit

RETWEETS
11

FAVORITES
17

11:06 AM - 30 Jun 2014

(2004 – 2013)

hadoop Map Reduce

General Batch Processing

(2007 – 2015?)

Pregel

Giraph

Tez

Drill

Dremel

Mahout

Storm

S4

Impala

GraphLab

Specialized Systems

(iterative, interactive, ML, streaming, graph, SQL, etc)

(2014 – ?)

Spark

General Unified Engine

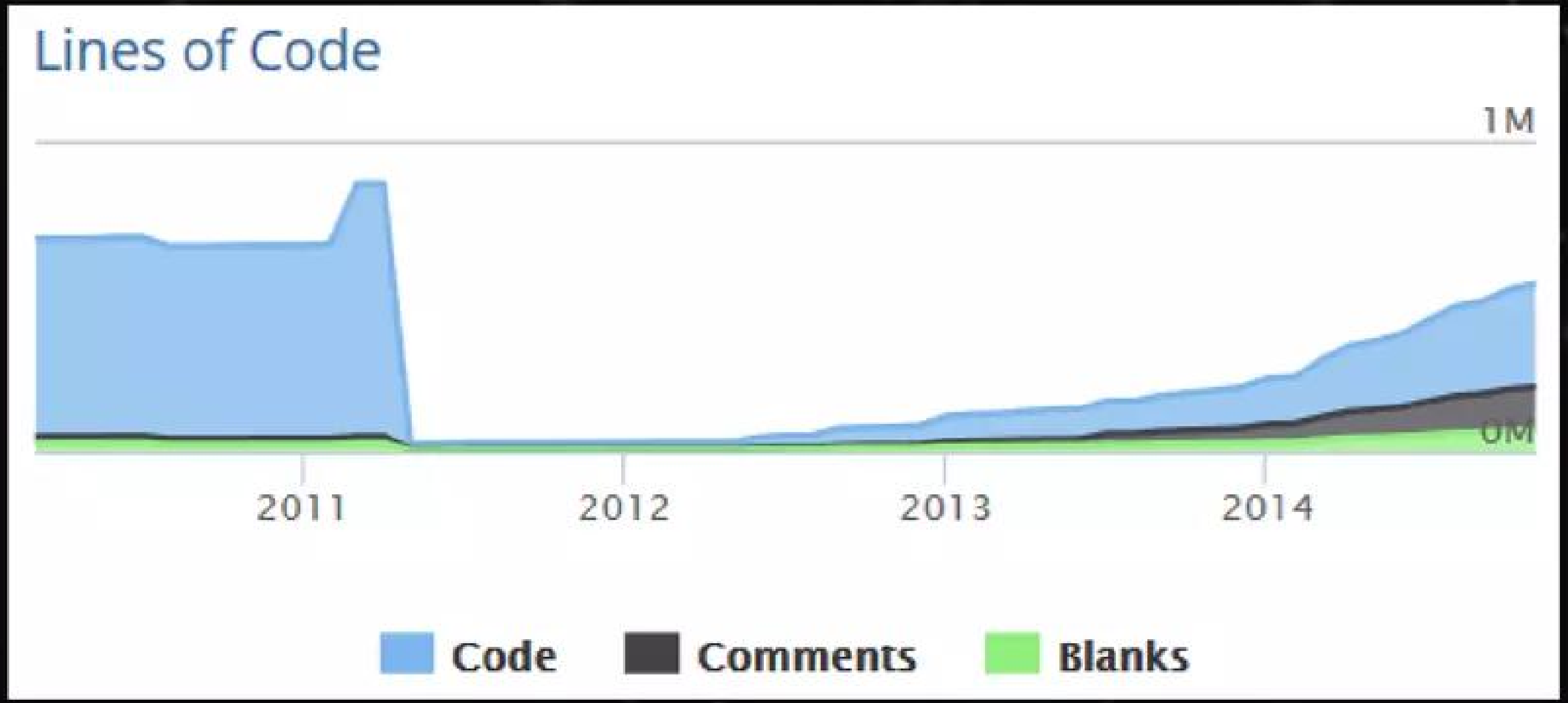
In a Nutshell, Apache Spark...

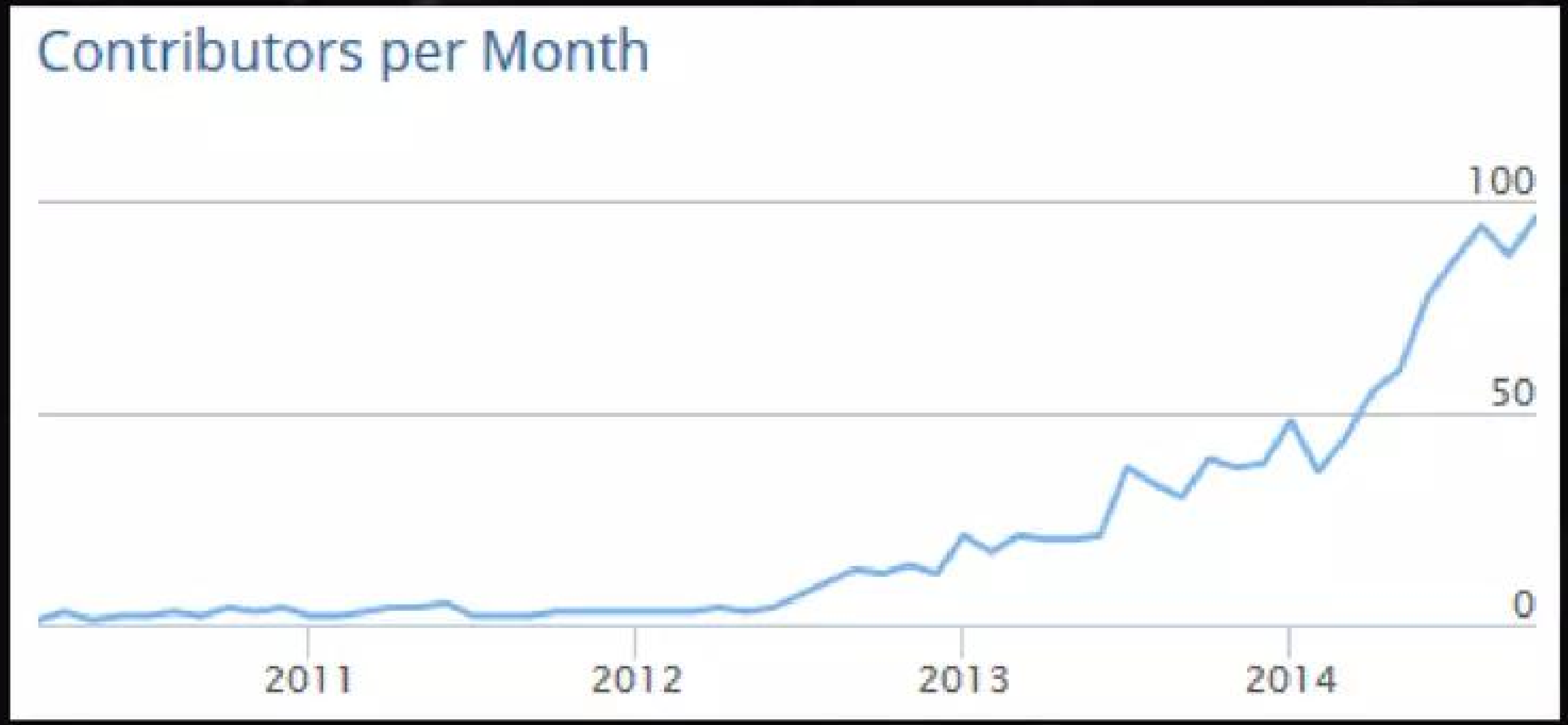
... has had 17,297 commits made by 448 contributors
representing 332,309 lines of code

... is mostly written in Scala
with a well-commented source code

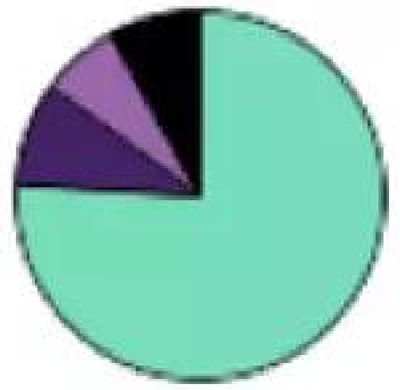
... has a codebase with a long source history
maintained by a very large development team
with stable Y-O-Y commits

... 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





Languages



Scala	76%	Python	9%
Java	7%	9 Other	8%

...in June 2013

Source: openhub.net

DISTRIBUTORS

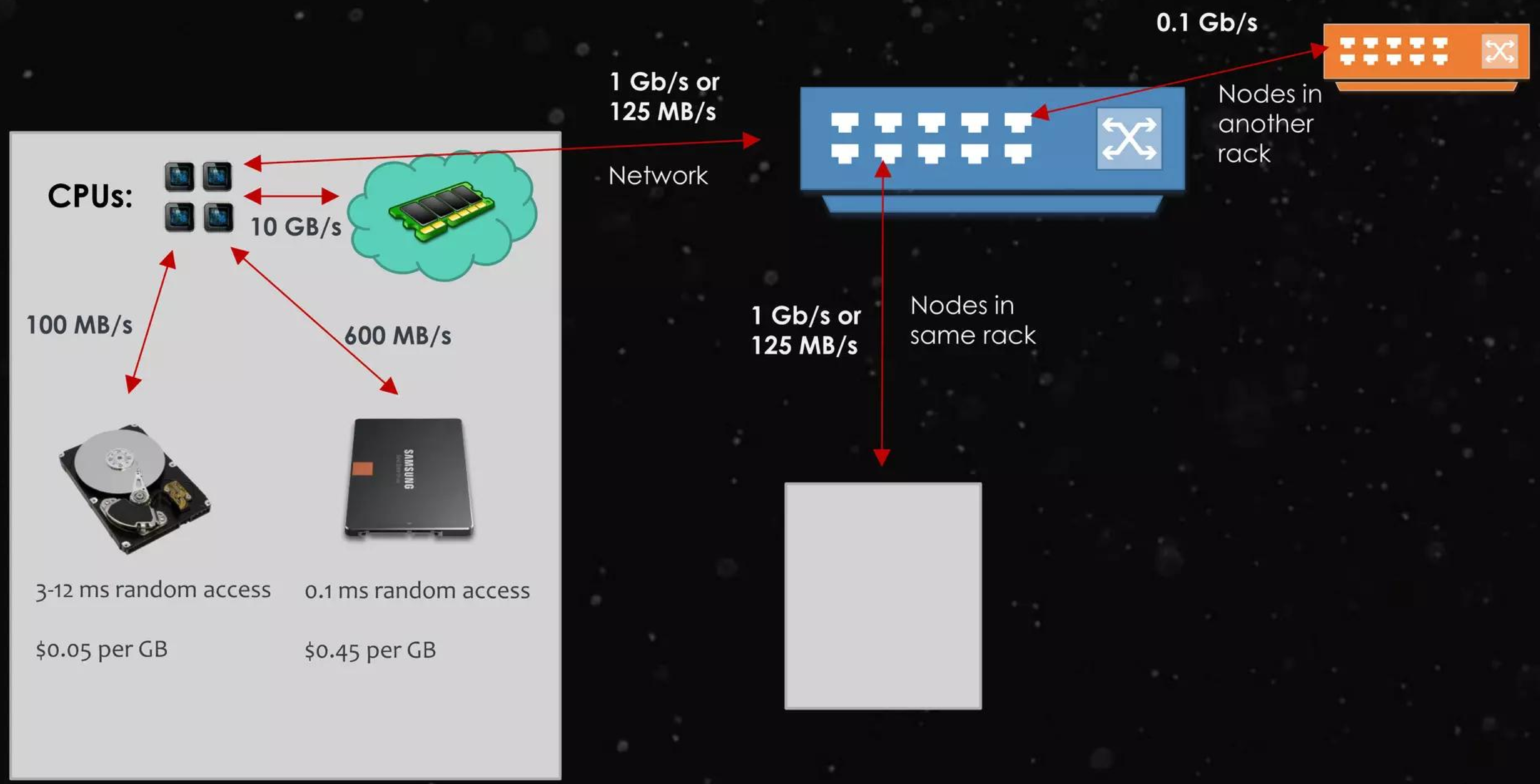
databricks

Hortonworks
MAPR
cloudera
IBM
Pivotal
ORACLE
DATASTAX
SAP
guavus
bluedata
STRATIO

APPLICATIONS

tableau SOFTWARE
MicroStrategy
Qlik
elasticsearch.
pentaho
talend
tresata
TRIFACTA
SKYTREE THE MACHINE LEARNING COMPANY
Alpine
atscale
looker
technicolor virdata
FAIMDATA
ADATAO DATA INTELLIGENCE FOR ALL
DIYOTTA
ZOOMDATA DATA in Motion
platfora
ZAPERVI
NUBE
Atigeo
日志易 rizhiyi.com
ZALONI
Typesafe


hadoop HDFS
hadoop Map Reduce
OOZIE
hadoop HDFS
hadoop Map Reduce
hadoop HDFS
hadoop HDFS
Spark
Spark
10x – 100x



CPUs:
10 GB/s
100 MB/s
600 MB/s
3-12 ms random access
$0.05 per GB
0.1 ms random access
$0.45 per GB
1 Gb/s or
125 MB/s
Network
0.1 Gb/s
Nodes in
another
rack
1 Gb/s or
125 MB/s
Nodes in
same rack

Spark: Cluster Computing with Working Sets

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

Abstract

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

1 Introduction

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

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

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

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

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

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

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

This paper is organized as follows. Section 2 describes

"The main abstraction in Spark is that of a resilient distributed dataset (RDD), which represents a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost.

Users can explicitly cache an RDD in memory across machines and reuse it in multiple MapReduce-like parallel operations.

RDDs achieve fault tolerance through a notion of lineage: if a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to be able to rebuild just that partition."

June 2010

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

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

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

Abstract

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

1 Introduction

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

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

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

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

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

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

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

"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."

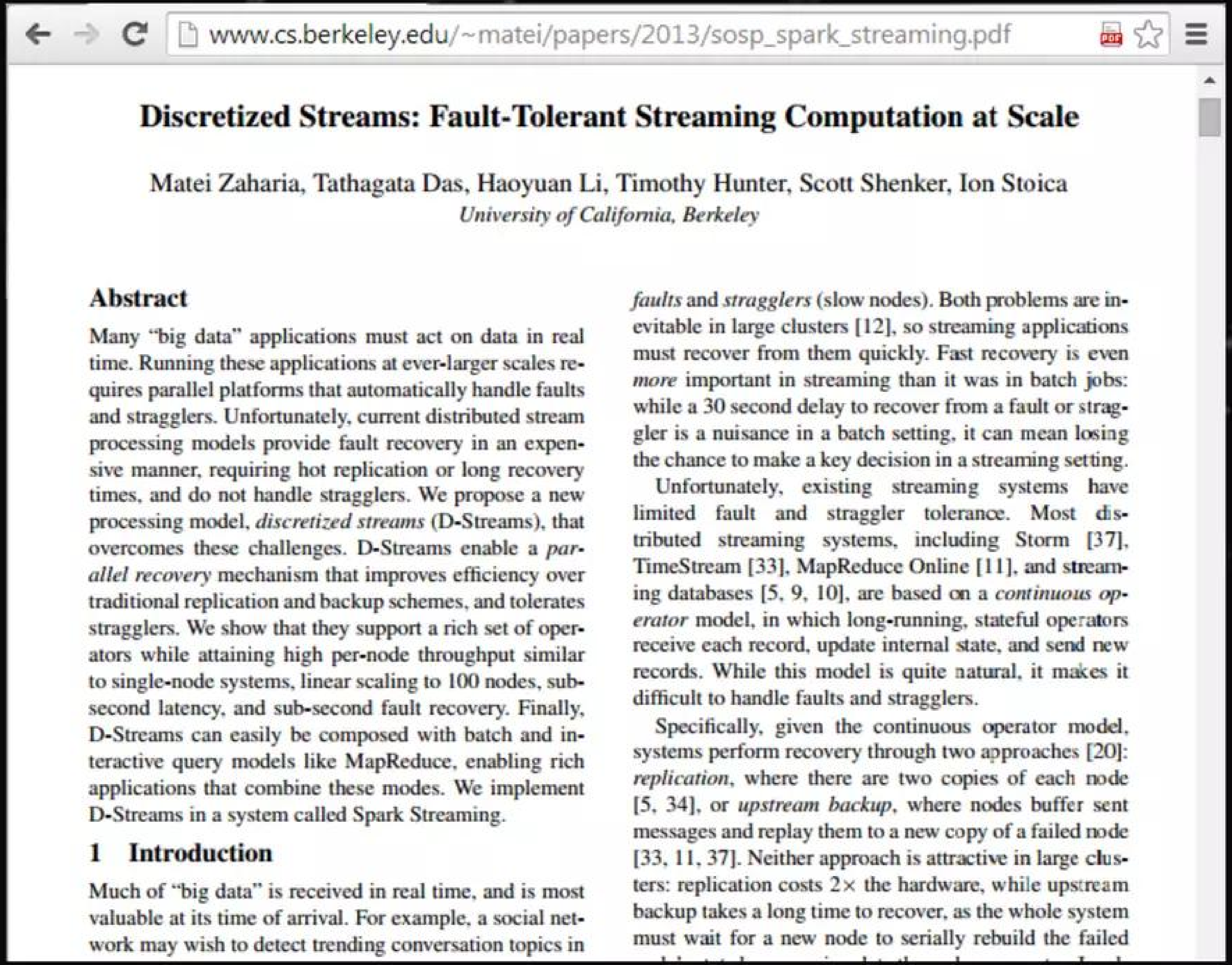
"Best Paper Award and Honorable Mention for Community Award"
- NSDI 2012

- Cited 392 times!

April 2012

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

Spark STREAMING



www.cs.berkeley.edu/~matei/papers/2013/sosp_spark_streaming.pdf

Discretized Streams: Fault-Tolerant Streaming Computation at Scale

Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, Ion Stoica
University of California, Berkeley

Abstract

Many "big data" applications must act on data in real time. Running these applications at ever-larger scales requires parallel platforms that automatically handle faults and stragglers. Unfortunately, current distributed stream processing models provide fault recovery in an expensive manner, requiring hot replication or long recovery times, and do not handle stragglers. We propose a new processing model, *discretized streams* (D-Streams), that overcomes these challenges. D-Streams enable a *parallel recovery* mechanism that improves efficiency over traditional replication and backup schemes, and tolerates stragglers. We show that they support a rich set of operators while attaining high per-node throughput similar to single-node systems, linear scaling to 100 nodes, sub-second latency, and sub-second fault recovery. Finally, D-Streams can easily be composed with batch and interactive query models like MapReduce, enabling rich applications that combine these modes. We implement D-Streams in a system called Spark Streaming.

1 Introduction

Much of "big data" is received in real time, and is most valuable at its time of arrival. For example, a social network may wish to detect trending conversation topics in *faults* and *stragglers* (slow nodes). Both problems are inevitable in large clusters [12], so streaming applications must recover from them quickly. Fast recovery is even *more* important in streaming than it was in batch jobs: while a 30 second delay to recover from a fault or straggler is a nuisance in a batch setting, it can mean losing the chance to make a key decision in a streaming setting.

Unfortunately, existing streaming systems have limited fault and straggler tolerance. Most distributed streaming systems, including Storm [37], TimeStream [33], MapReduce Online [11], and streaming databases [5, 9, 10], are based on a *continuous operator* model, in which long-running, stateful operators receive each record, update internal state, and send new records. While this model is quite natural, it makes it difficult to handle faults and stragglers.

Specifically, given the continuous operator model, systems perform recovery through two approaches [20]: *replication*, where there are two copies of each node [5, 34], or *upstream backup*, where nodes buffer sent messages and replay them to a new copy of a failed node [33, 11, 37]. Neither approach is attractive in large clusters: replication costs 2× the hardware, while upstream backup takes a long time to recover, as the whole system must wait for a new node to serially rebuild the failed

```
TwitterUtils.createStream(...)
    .filter(_.getText.contains("Spark"))
    .countByWindow(Seconds(5))
```

- 2 Streaming Paper(s) have been cited 138 times

Spark SQL

Seemlessly mix SQL queries with Spark programs.

Coming soon!

(Will be published in the upcoming weeks for SIGMOD 2015)

```
sqlCtx = new HiveContext(sc)
results = sqlCtx.sql(
  "SELECT * FROM people")
names = results.map(lambda p: p.name)
```

Spark GRAPHX

GraphX: A Resilient Distributed Graph System on Spark

Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, Ion Stoica

AMPLab, EECS, UC Berkeley
{rxin, jegonzal, franklin, istoica}@cs.berkeley.edu

ABSTRACT

From social networks to targeted advertising, big graphs capture the structure in data and are central to recent advances in machine learning and data mining. Unfortunately, directly applying existing data-parallel tools to graph computation tasks can be cumbersome and inefficient. The need for intuitive, scalable tools for graph computation has lead to the development of new *graph-parallel* systems (*e.g.*, Pregel, PowerGraph) which are designed to efficiently execute graph algorithms. Unfortunately, these new graph-parallel systems do not address the challenges of graph construction and transformation which are often just as problematic as the subsequent computation. Furthermore, existing graph-parallel systems provide limited fault-tolerance and support for interactive data mining.

We introduce GraphX, which combines the advantages of both data-parallel and graph-parallel systems by efficiently expressing graph computation within the Spark data-parallel framework. We leverage new ideas in distributed graph representation to efficiently distribute graphs as tabular data-structures. Similarly, we leverage advances in data-flow systems to exploit in-memory computation and fault-tolerance. We provide powerful new operations to simplify graph construction and transformation. Using these primitives we implement the PowerGraph and Pregel abstractions in less than 20 lines of code. Finally, by exploiting the Scala foundation of Spark, we enable users to interactively load, transform, and compute on massive graphs.

1. INTRODUCTION

From social networks to advertising and the web, big graphs can be found in a wide range of important applications. By modeling the

and distributed systems. By abstracting away the challenges of large-scale distributed system design, these frameworks simplify the design, implementation, and application of new sophisticated graph algorithms to large-scale real-world graph problems.

While existing graph-parallel frameworks share many common properties, each presents a slightly different view of graph computation tailored to either the originating domain or a specific family of graph algorithms and applications. Unfortunately, because each framework relies on a separate runtime, it is difficult to compose these abstractions. Furthermore, while these frameworks address the challenges of graph computation, they do not address the challenges of data ETL (preprocessing and construction) or the process of interpreting and applying the results of computation. Finally, few frameworks have built-in support for interactive graph computation.

Alternatively *data-parallel* systems like MapReduce and Spark [12] are designed for scalable data processing and are well suited to the task of graph construction (ETL). By exploiting data-parallelism, these systems are highly scalable and support a range of fault-tolerance strategies. More recent systems like Spark even enable interactive data processing. However, naively expressing graph computation and graph algorithms in these data-parallel abstractions can be challenging and typically leads to complex joins and excessive data movement that does not exploit the graph structure.

To address these challenges we introduce GraphX, a graph computation system which runs in the Spark data-parallel framework. GraphX extends Spark's Resilient Distributed Dataset (RDD) abstraction to introduce the Resilient Distributed Graph (RDG), which associates records with vertices and edges in a graph and provides a collection of expressive computational primitives. Using these

```
graph = Graph(vertices, edges)
messages = spark.textFile("hdfs://...")
graph2 = graph.joinVertices(messages) {
  (id, vertex, msg) => ...
}
```

https://amplab.cs.berkeley.edu/wp-content/uploads/2013/05/grades-graphx_with_fonts.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, cessing 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

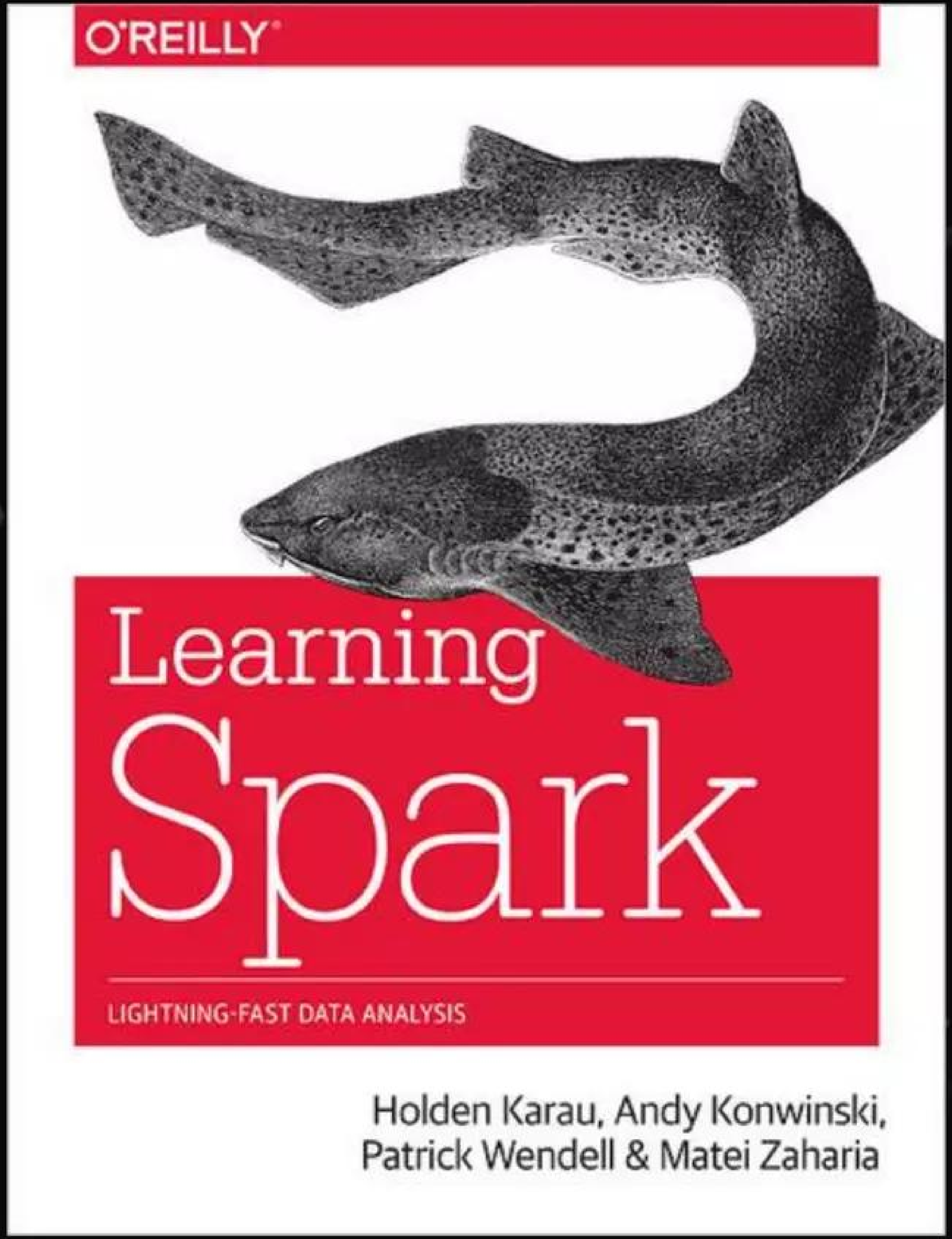
```
SELECT avg(sessionTime)
FROM Table
WHERE city='San Francisco'
WITHIN 2 SECONDS
```

Queries with Time Bounds

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

Queries with Error Bounds

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

Community | Apache Spa

https://spark.apache.org/community.html

Spark Lightning-fast cluster computing

Download | Libraries | Documentation | Examples | Community | FAQ

Latest News

Spark 1.2.1 released (Feb 09, 2015)

Spark Summit East agenda posted, CFP open for West (Jan 21, 2015)

Spark 1.2.0 released (Dec 18, 2014)

Spark 1.1.1 released (Nov 26, 2014)

Archive

Spark Community

Mailing Lists

Get help using Spark or contri[...] to the project on our mailing lists:

- user@spark.apache.org is for usage questions, help, and announcements. (subscribe) (unsubscribe) (archives)
- dev@spark.apache.org is for people who want to contribute code to Spark. (subscribe) (unsubscribe) (archives)

databricks

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 Kołaczkowski (DS), Holden Karau (DB), Pat McDonough (DB), Patrick Wendell (DB) and Matei Zaharia (DB).

Estimated lab completion time: **2.5 hours**

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)

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

Spark Packages

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

@pwendell / Latest release: 0.1 (11/27/14) / Apache-2.0 / ★★★★★ (5)

3 sql   3 input   2 library

dibbhatt/kafka-spark-consumer

Low Level Kafka-Spark Consumer

@dibbhatt / No release yet / ★★★★★ (3)

2 streaming   1 kafka

sigmoidanalytics/spork

Pig on Apache Spark

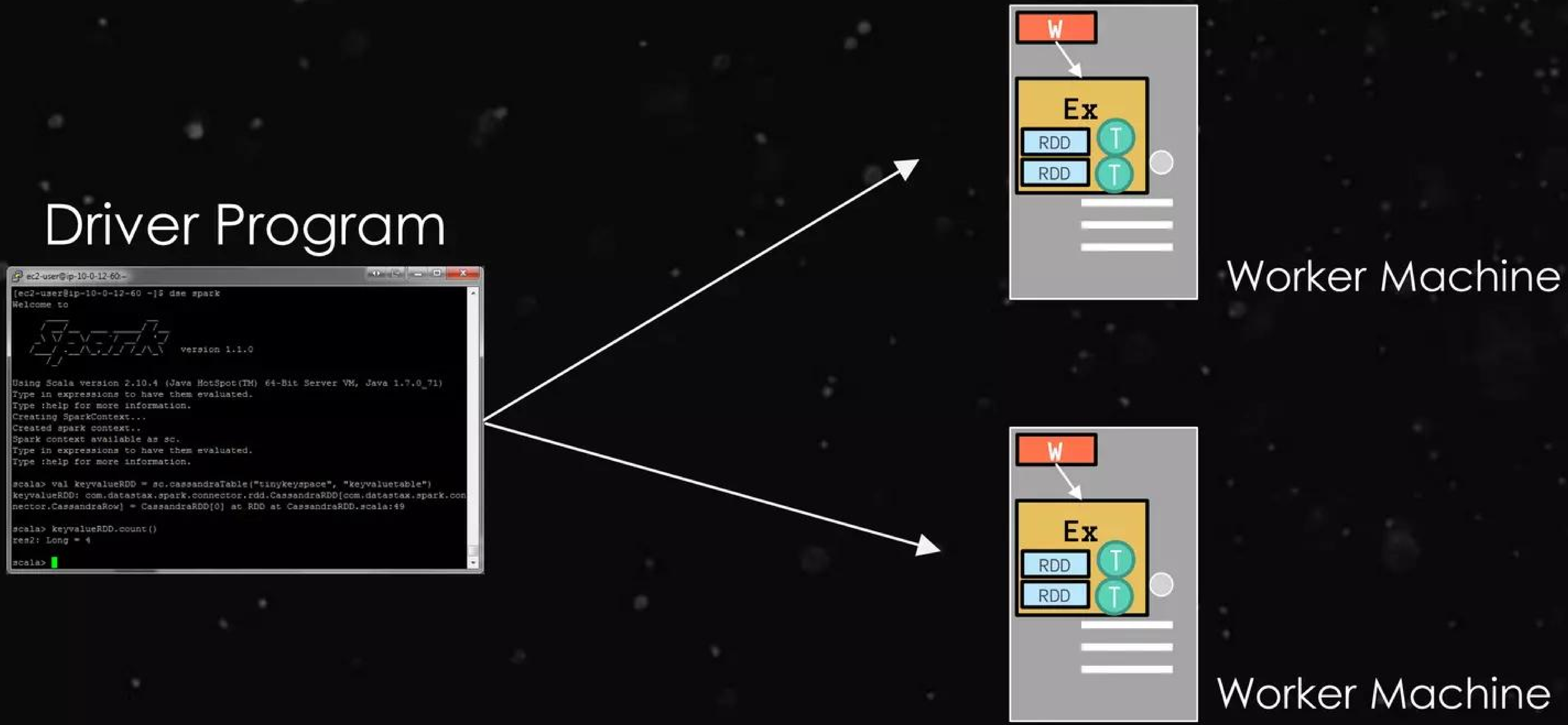
RDD FUNDAMENTALS

databricks

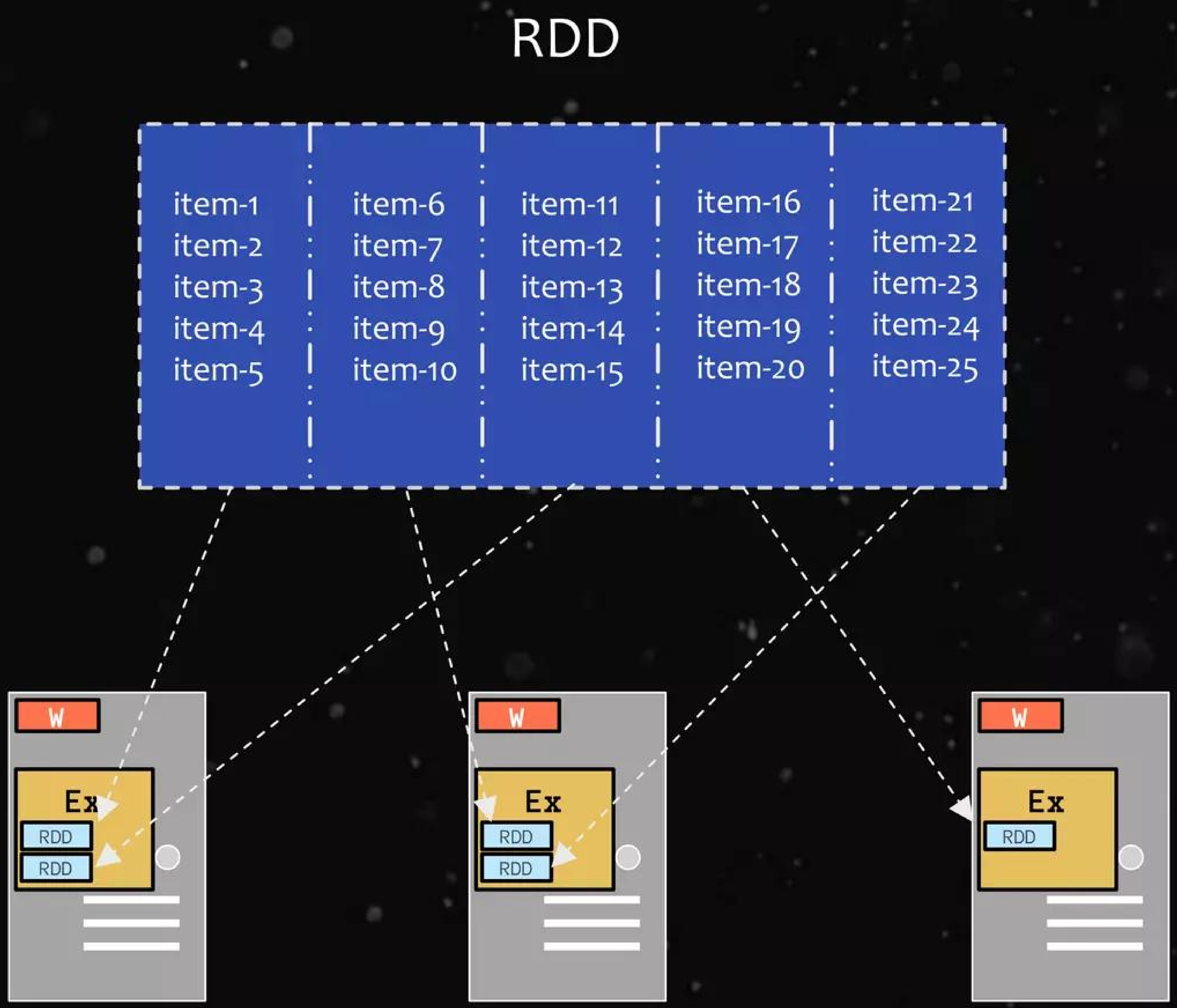
INTERACTIVE SHELL

```
ubuntu@ip-10-0-53-24: ~
ubuntu@ip-10-0-53-24:~$ dse spark
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> val myRDD = sc.cassandraTable("tinykeyspace", "keyvaluetable")
myRDD: com.datastax.bdp.spark.CassandraRDD[com.datastax.bdp.spark.CassandraRow] = Cassan
draRDD[0] at RDD at CassandraRDD.scala:32

scala> myRDD.count()
res2: Long = 5

scala>
```

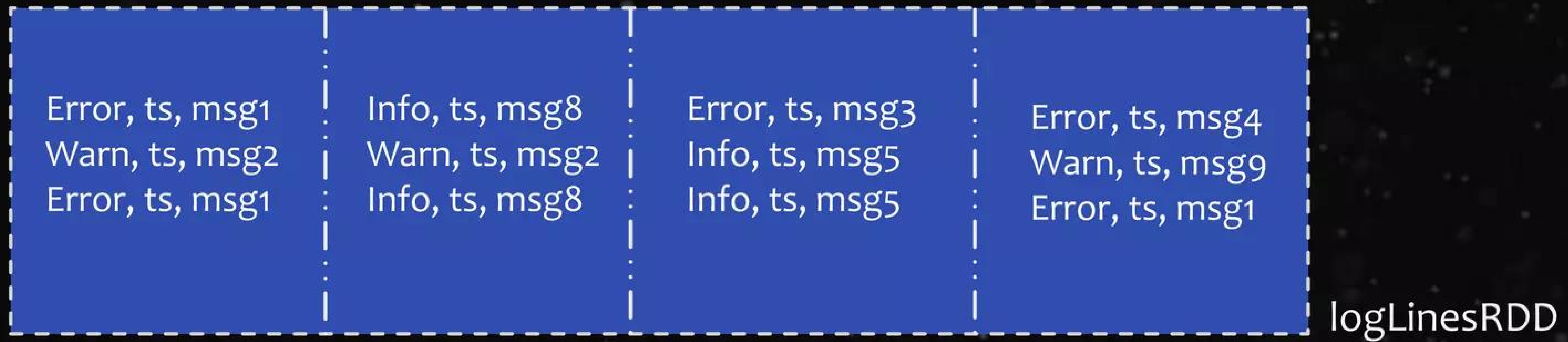
(Scala & Python only)


Driver Program
ec2-user@ip-10-0-12-60:~
[ec2-user@ip-10-0-12-60 ~]$ dse spark
Welcome to
version 1.1.0
Using Scala version 2.10.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_71)
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> val keyvalueRDD = sc.cassandraTable("tinykeyspace", "keyvaluetable")
keyvalueRDD: com.datastax.spark.connector.rdd.CassandraRDD[com.datastax.spark.connector.CassandraRow] = CassandraRDD[0] at RDD at CassandraRDD.scala:49
scala> keyvalueRDD.count()
res2: Long = 4
scala>
W
Ex
RDD
RDD
T
T
Worker Machine
W
Ex
RDD
RDD
T
T
Worker Machine

more partitions = more parallelism



RDD w/ 4 partitions



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

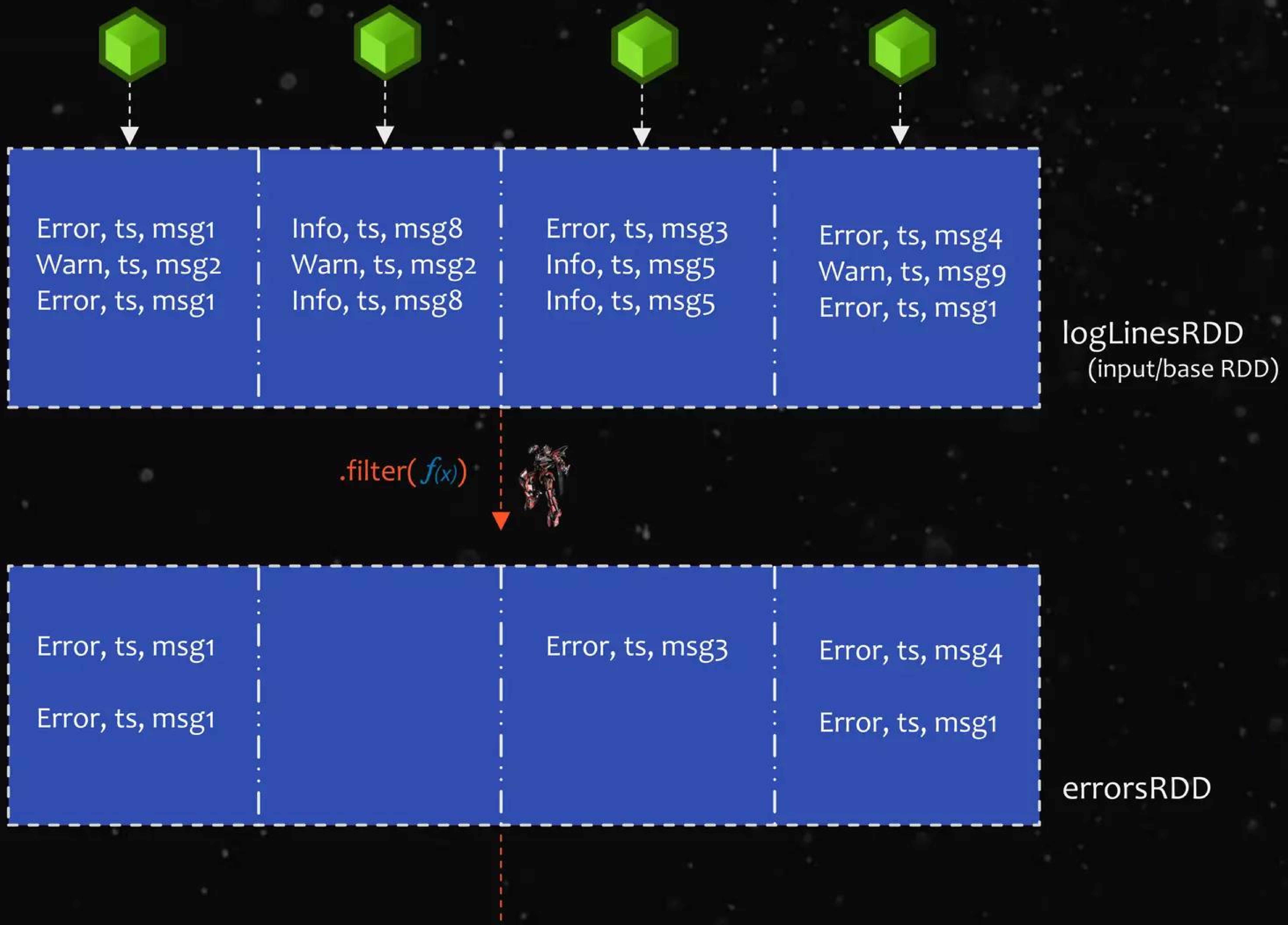
READ FROM TEXT FILE

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

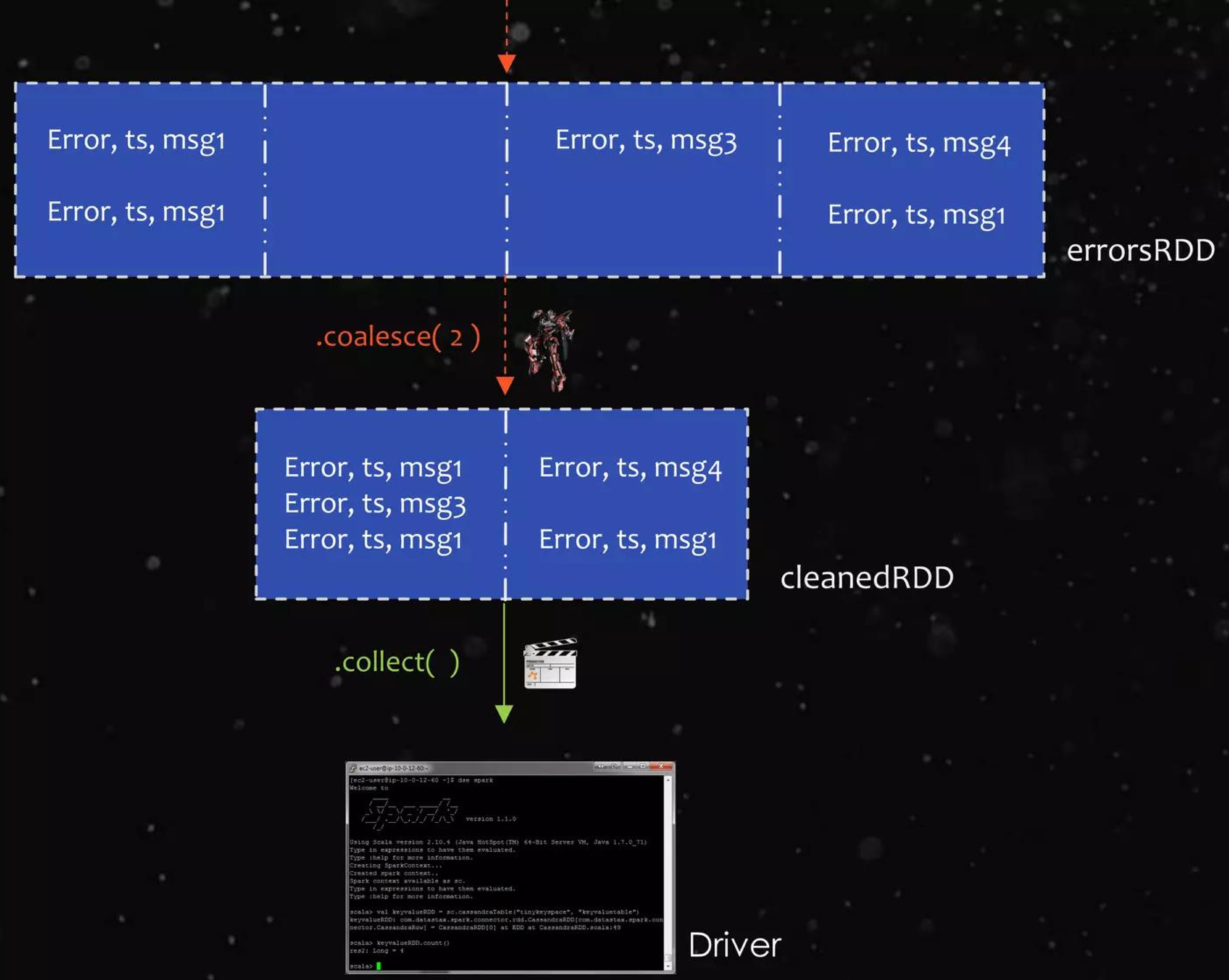
```scala
// Read a local txt file in Scala
val linesRDD = sc.textFile("/path/to/README.md")
```

```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.

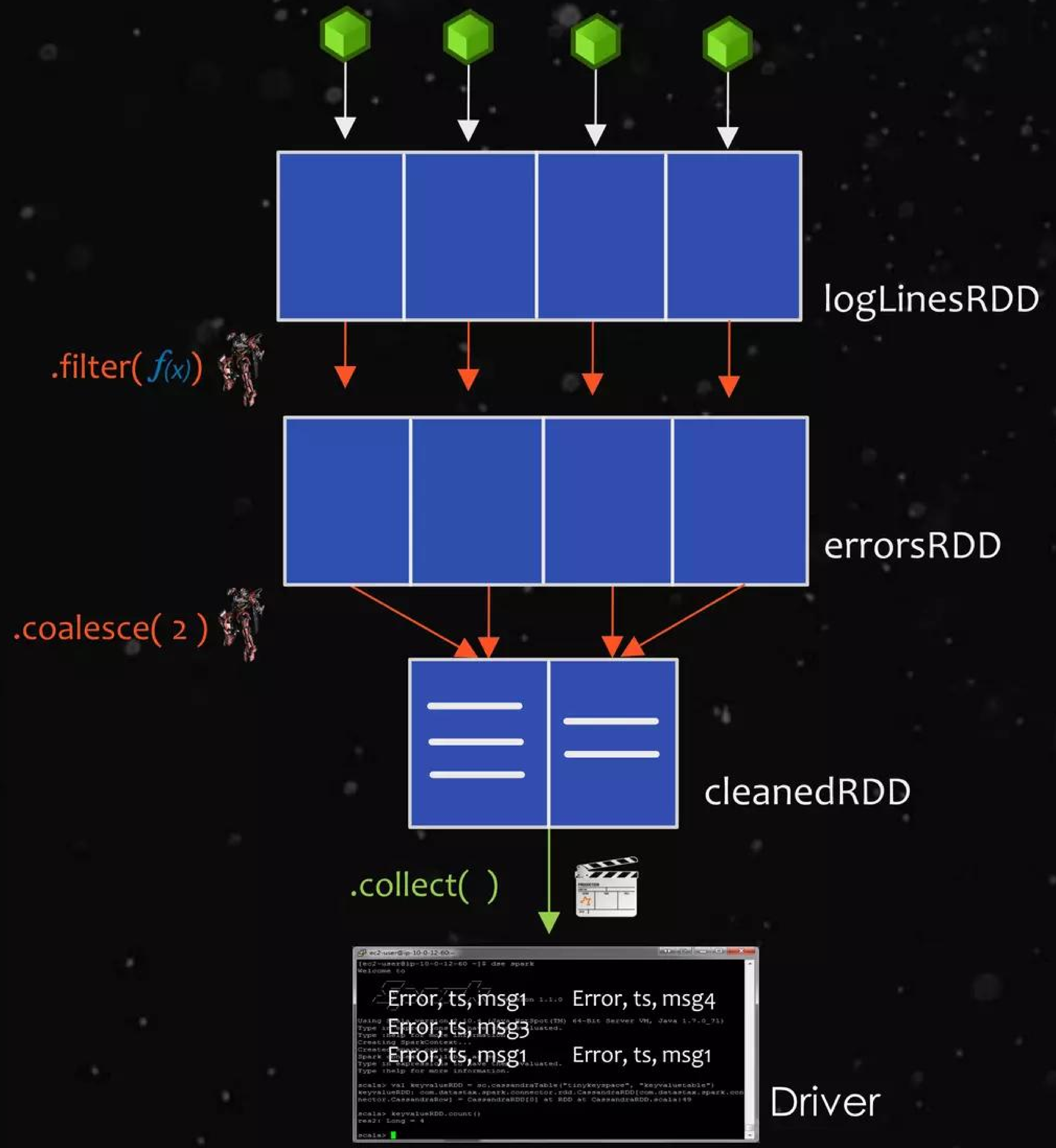


Error, ts, msg1
Warn, ts, msg2
Error, ts, msg1
Info, ts, msg8
Warn, ts, msg2
Info, ts, msg8
Error, ts, msg3
Info, ts, msg5
Info, ts, msg5
Error, ts, msg4
Warn, ts, msg9
Error, ts, msg1
logLinesRDD
(input/base RDD)
.filter(f(x))
Error, ts, msg1
Error, ts, msg1
Error, ts, msg3
Error, ts, msg4
Error, ts,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

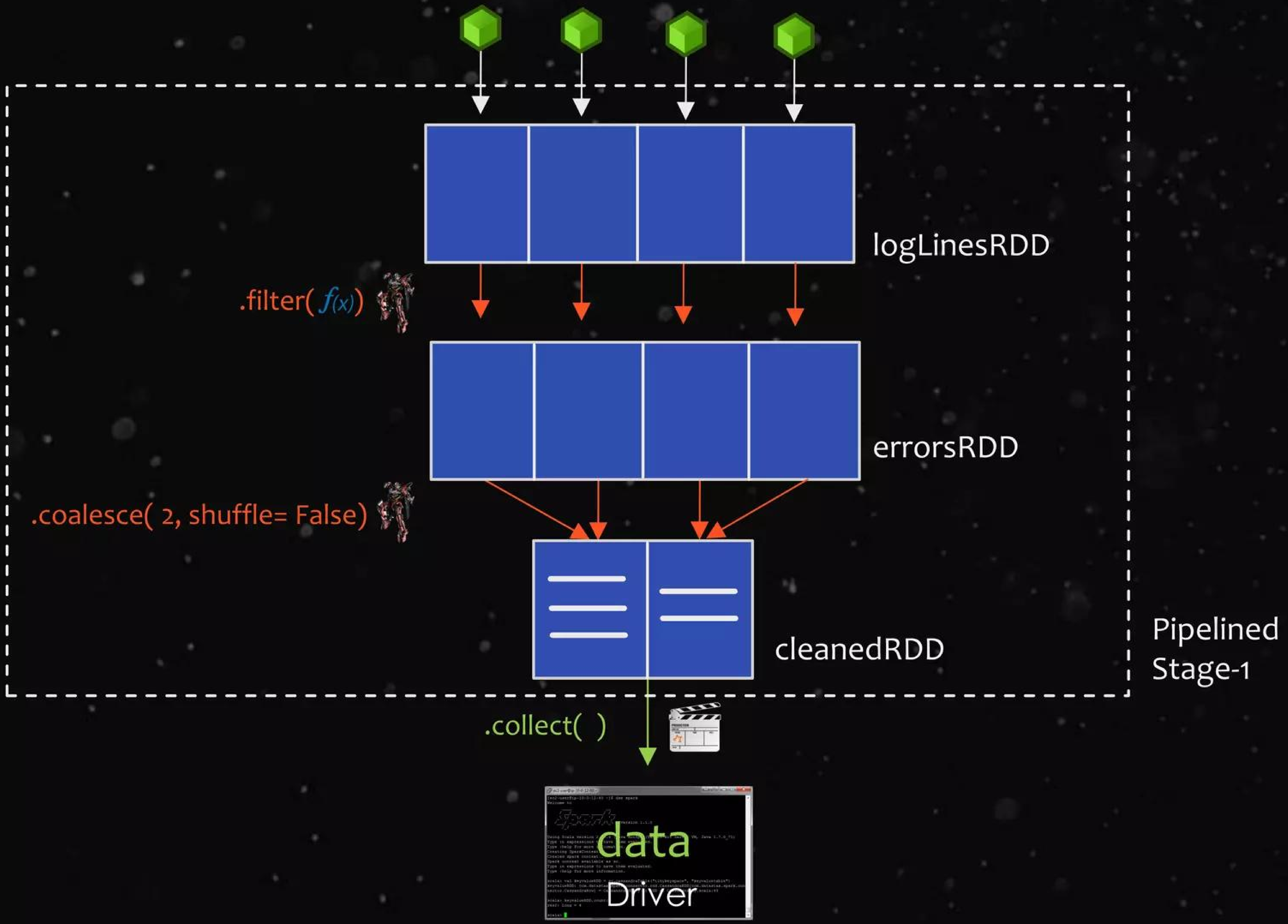

Error, ts, msg1
Error, ts, msg1
Error, ts, msg3
Error, ts, msg4
Error, ts, msg1
errorsRDD
.coalesce(2)
Error, ts, msg1
Error, ts, msg3
Error, ts, msg1
Error, ts, msg4
Error, ts, msg1
cleanedRDD
.collect( )
Driver


Execute DAG!
.collect( )
ec2-user@ip-10-0-12-60:~
[ec2-user@ip-10-0-12-60 ~]$ dse spark
Welcome to
version 1.1.0
Using Scala version 2.10.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_71)
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> val keyvalueRDD = sc.cassandraTable("tinykeyspace", "keyvaluetable")
keyvalueRDD: com.datastax.spark.connector.rdd.CassandraRDD[com.datastax.spark.con
nector.CassandraRow] = CassandraRDD[0] at RDD at CassandraRDD.scala:49
scala> keyvalueRDD.count()
res2: Long = 4
scala>
Driver

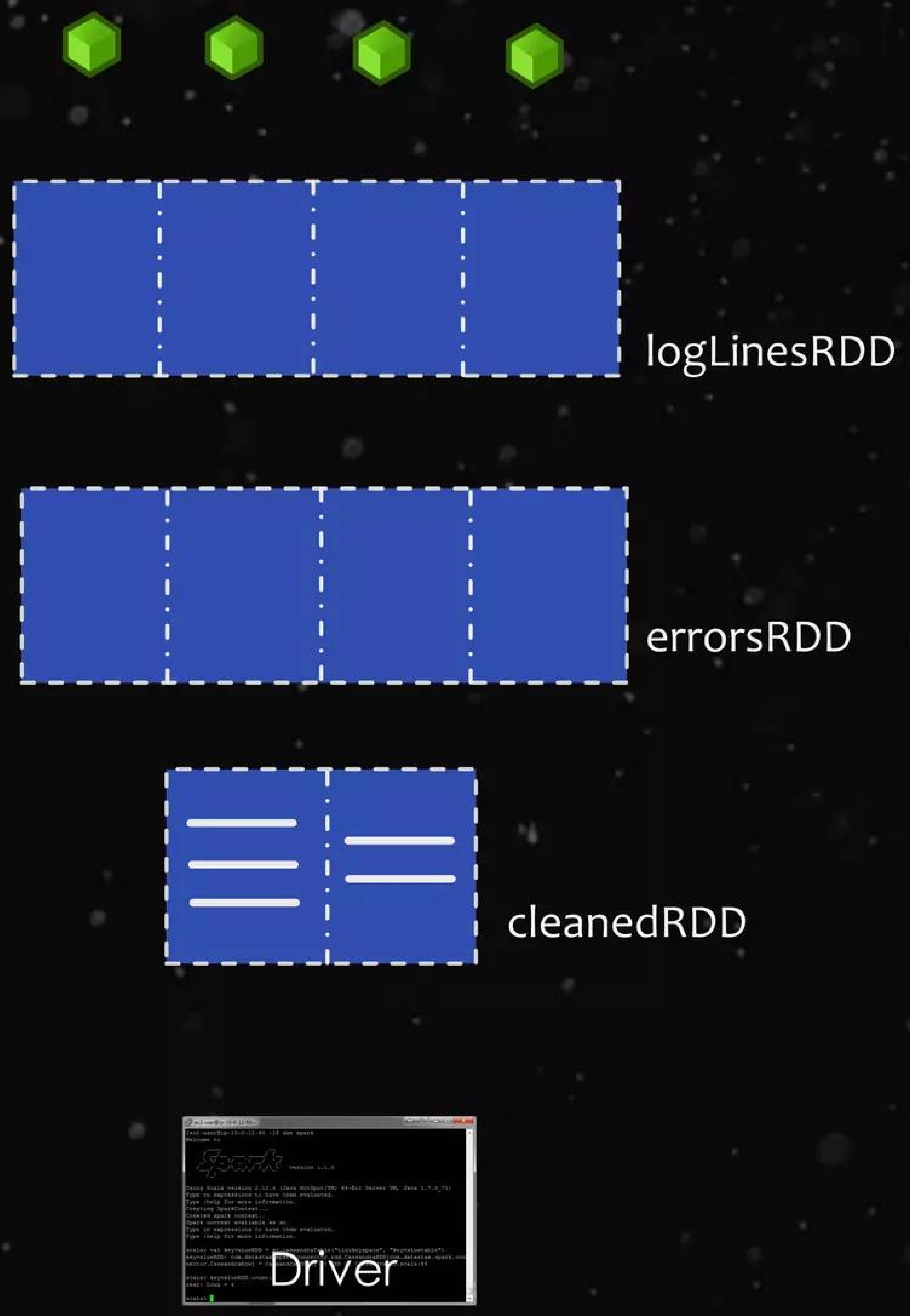

logLinesRDD
.collect( )
Driver



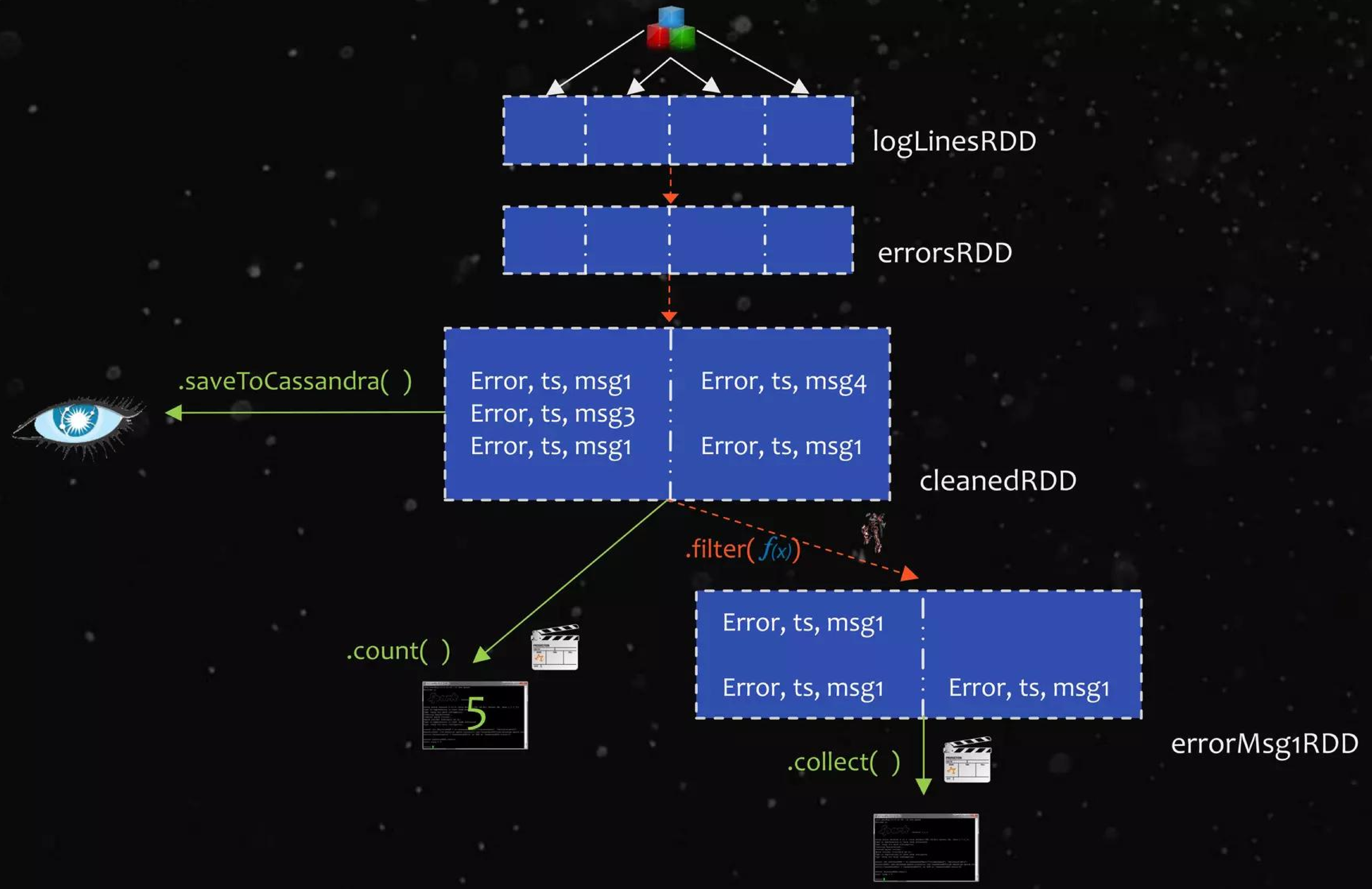
logLinesRDD
.filter(f(x))
errorsRDD
.coalesce(2)
cleanedRDD
.collect( )
Error, ts, msg1
Error, ts, msg4
Error, ts, msg3
Error, ts, msg1
Error, ts, msg1
Driver

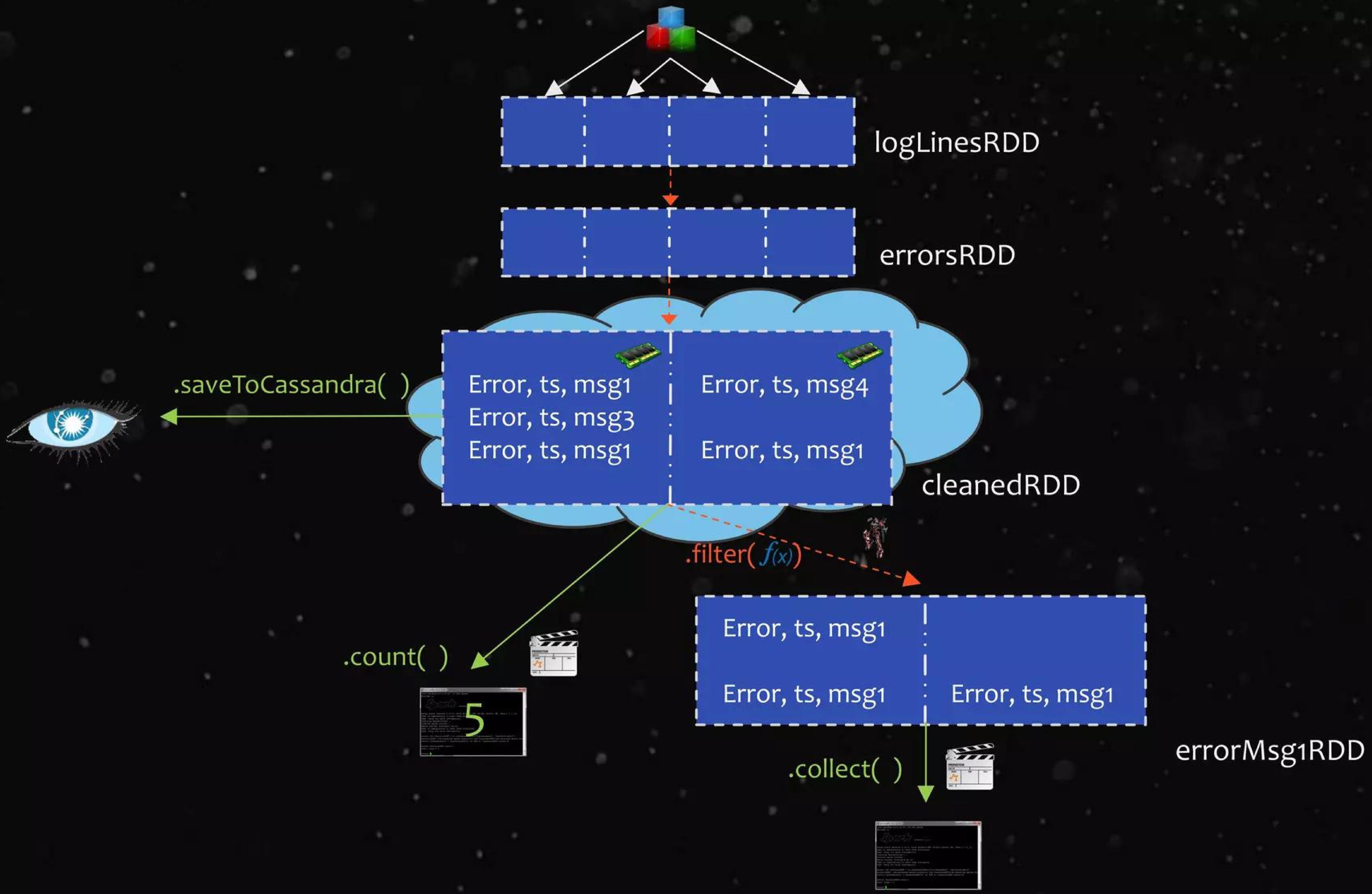


logLinesRDD
.filter(f(x))
errorsRDD
.coalesce(2, shuffle= False)
cleanedRDD
Pipelined
Stage-1
.collect( )
data
Driver


logLinesRDD
errorsRDD
cleanedRDD
Driver


data
Driver

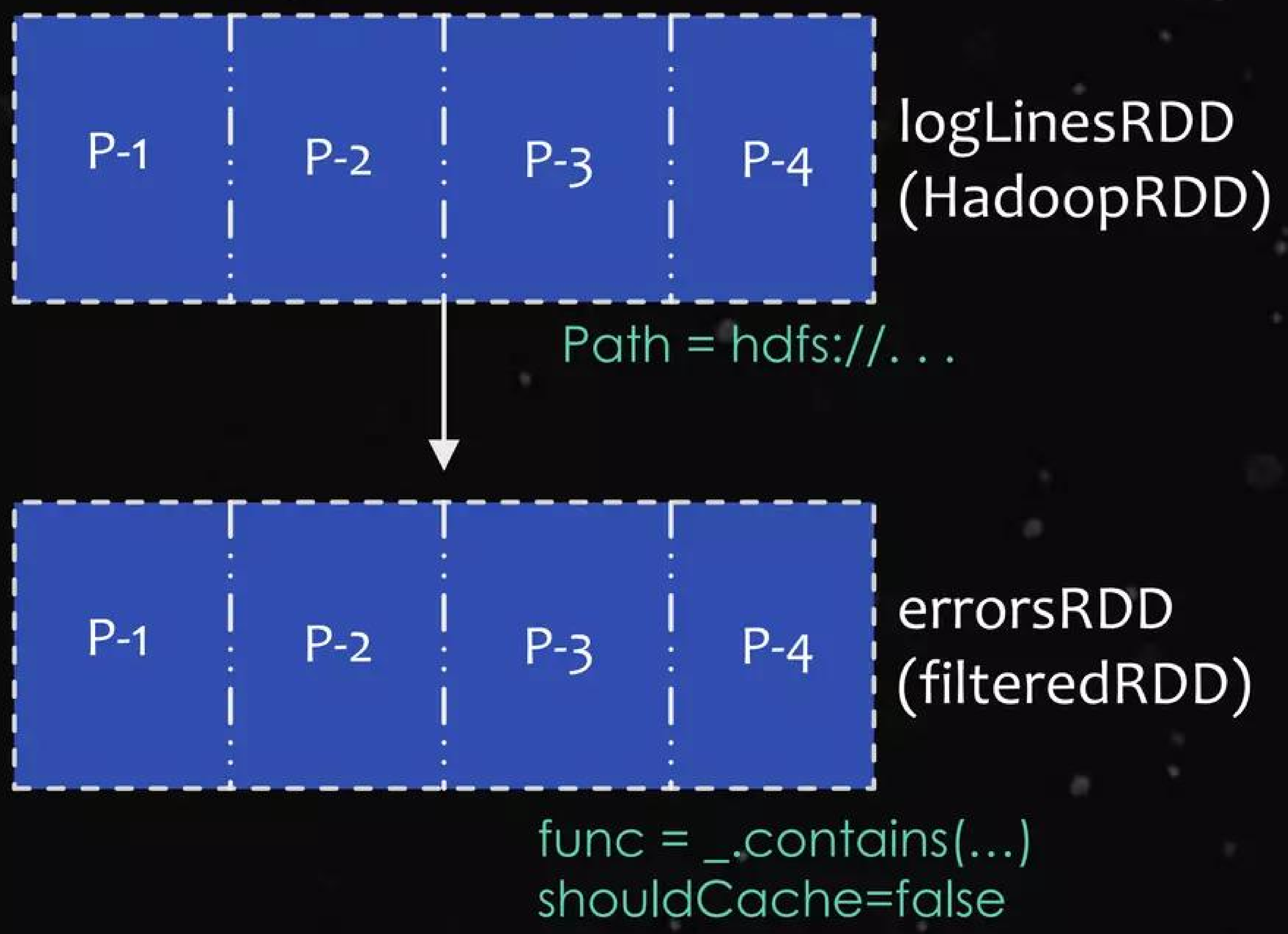

logLinesRDD
errorsRDD
.saveToCassandra( )
Error, ts, msg1
Error, ts, msg3
Error, ts, msg1
Error, ts, msg4
Error, ts, msg1
cleanedRDD
.filter(f(x))
.count( )
5
Error, ts, msg1
Error, ts, msg1
Error, ts, msg1
errorMsg1RDD
.collect( )



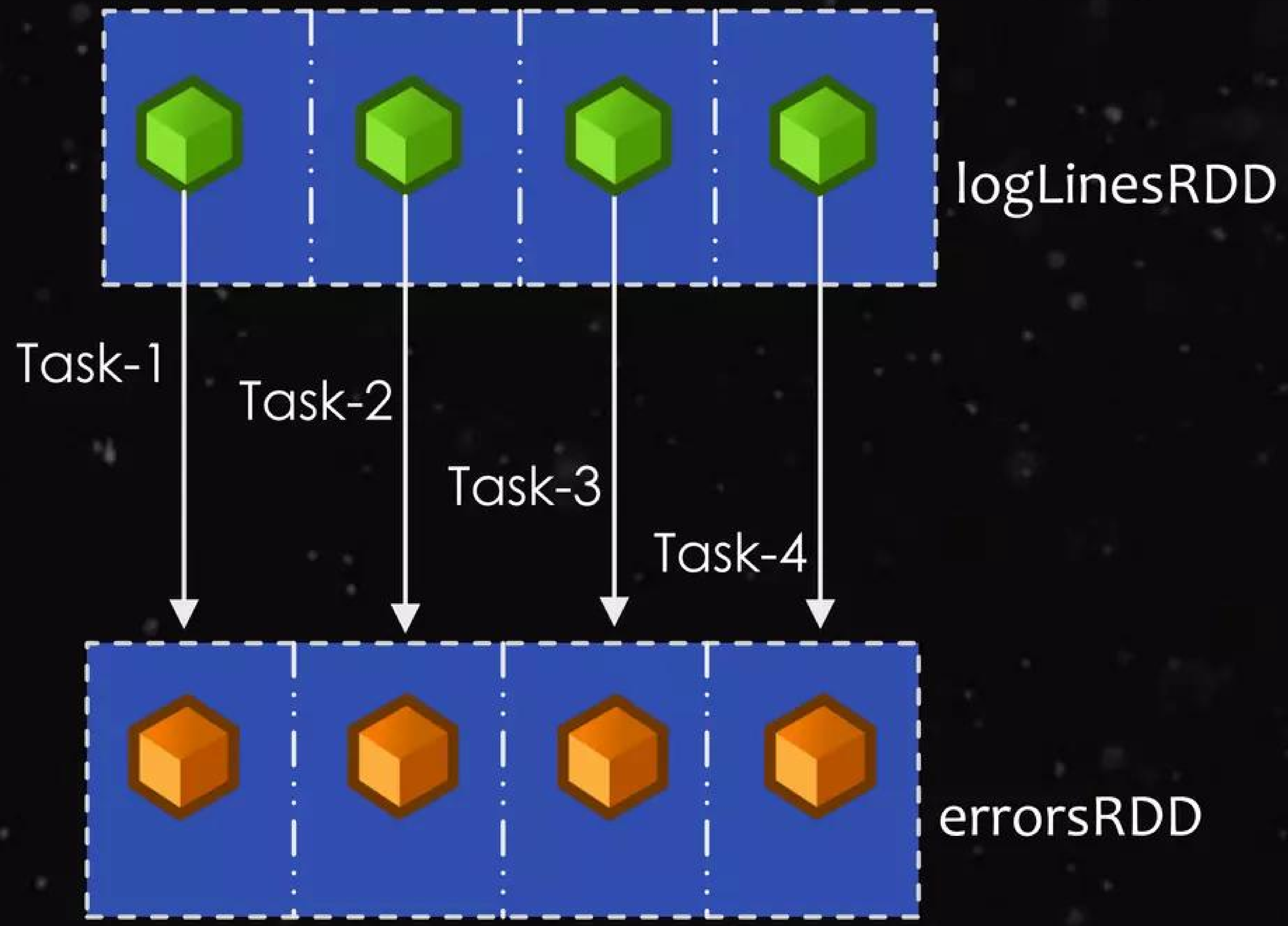
logLinesRDD
errorsRDD
.saveToCassandra( )
Error, ts, msg1
Error, ts, msg3
Error, ts, msg1
Error, ts, msg4
Error, ts, msg1
cleanedRDD
.filter(f(x))
.count( )
5
Error, ts, msg1
Error, ts, msg1
Error, ts, msg1
.collect( )
errorMsg1RDD

RDD GRAPH

Dataset-level view:



Partition-level view:



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

TYPES OF RDDS

- HadoopRDD
- FilteredRDD
- MappedRDD
- PairRDD
- ShuffledRDD
- UnionRDD
- PythonRDD

- DoubleRDD
- JdbcRDD
- JsonRDD
- SchemaRDD
- VertexRDD
- EdgeRDD

- CassandraRDD *(DataStax)*
- GeoRDD *(ESRI)*
- EsSpark *(ElasticSearch)*

spark/RDD.scala at 6c98c2

GitHub, Inc. [US] https://github.com/apache/spark/blob/6c98c29ae0033556fd4424f41d1de005c509e511/core/src/main/scala/org/apach

GitHub This repository Search Explore Features Enterprise Blog Sign up Sign in

apache / spark
mirrored from git://git.apache.org/spark.git

Watch 538 Star 2,884 Fork 2,520

tree: 6c98c29ae0 spark / core / src / main / scala / org / apache / spark / rdd / **RDD.scala**

aarondav on Oct 21, 2014 [SPARK-3994] Use standard Aggregator code path for countByKey and cou...

44 contributors and others

1384 lines (1235 sloc) 55.398 kb Raw Blame History

```
/*
 * Licensed to the Apache Software Foundation (ASF) under one or more
 * contributor license agreements.  See the NOTICE file distributed with
 * this work for additional information regarding copyright ownership.
 * The ASF licenses this file to You under the Apache License, Version 2.0
 * (the "License"); you may not use this file except in compliance with
 * the License.  You may obtain a copy of the License at
 *
 *    http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package org.apache.spark.rdd
```

spark/core/src/main/scala

GitHub, Inc. [US] https://github.com/apache/spark/tree/master/core/src/main/scala/org/apache/spark/rdd

GitHub This repository Search Explore Features Enterprise Blog Sign up Sign in

apache / spark
mirrored from git://git.apache.org/spark.git

Watch 541 Star 2,890 Fork 2,526

branch: master spark / core / src / main / scala / org / apache / spark / **rdd** / +

SPARK-5239 [CORE] JdbcRDD throws "java.lang.AbstractMethodError: orac… …

srowen authored 7 days ago latest commit 2d1e916730

..

AsyncRDDActions.scala	[SPARK-4397][Core] Cleanup 'import SparkContext._' in core	3 months ago
BinaryFileRDD.scala	[SPARK-4719][API] Consolidate various narrow dep RDD classes with Map…	3 months ago
BlockRDD.scala	[SPARK-4027][Streaming] WriteAheadLogBackedBlockRDD to read received …	4 months ago
CartesianRDD.scala	[SPARK-4080] Only throw IOException from [write|read][Object|External]	4 months ago
CheckpointRDD.scala	[SPARK-4014] Add TaskContext.attemptNumber and deprecate TaskContext.…	a month ago
CoGroupedRDD.scala	[SPARK-3288] All fields in TaskMetrics should be private and use gett…	29 days ago
CoalescedRDD.scala	[SPARK-4759] Fix driver hanging from coalescing partitions	2 months ago
DoubleRDDFunctions.scala	[SPARK-4397][Core] Cleanup 'import SparkContext._' in core	3 months ago
EmptyRDD.scala	SPARK-1093: Annotate developer and experimental API's	10 months ago
HadoopRDD.scala	[SPARK-4874] [CORE] Collect record count metrics	10 days ago
JdbcRDD.scala	SPARK-5239 [CORE] JdbcRDD throws "java.lang.AbstractMethodError: orac…	7 days ago

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

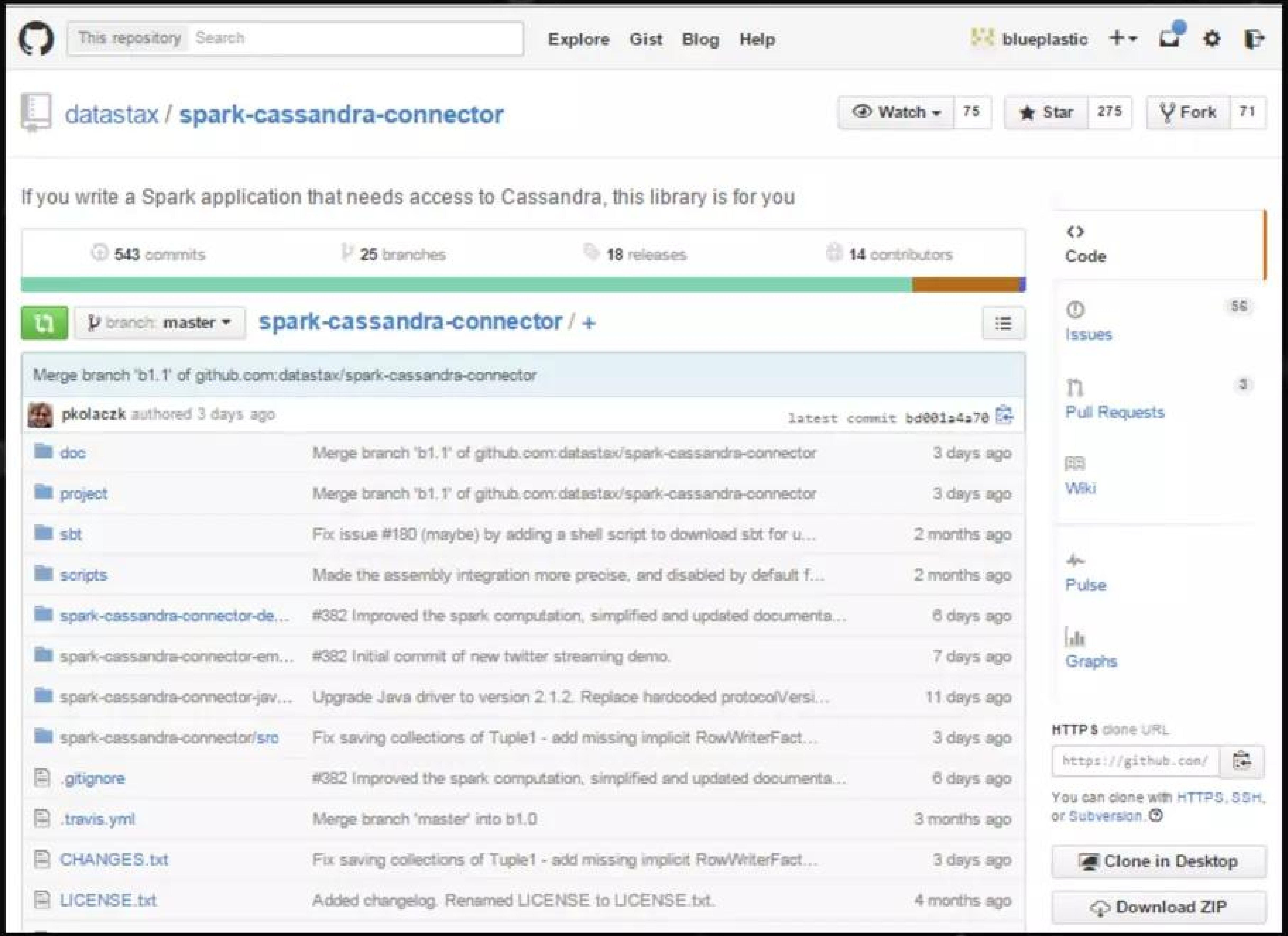
(for dealing with wide rows)

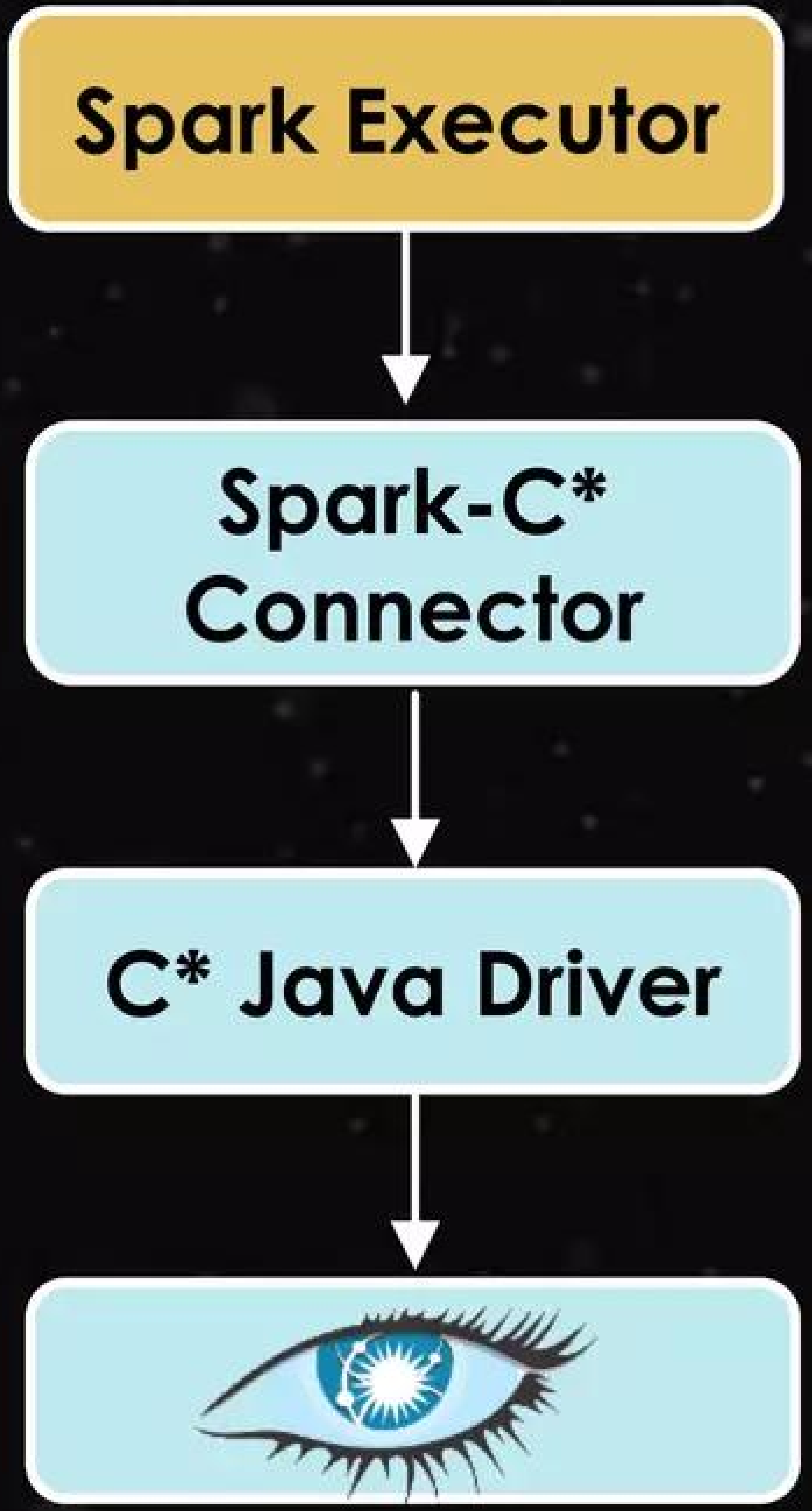
- **Start the Spark shell by passing in a custom cassandra.input.split.size:**

```
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

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

Spark Cassandra Connector build passing

Lightning-fast cluster computing with Spark and Cassandra

This library lets you expose Cassandra tables as Spark RDDs, write Spark RDDs to Cassandra tables, and execute arbitrary CQL queries in your Spark applications.

Features

- Compatible with Apache Cassandra version 2.0 or higher and DataStax Enterprise 4.5
- Compatible with Apache Spark 1.0 and 1.1
- Exposes Cassandra tables as Spark RDDs
- Maps table rows to CassandraRow objects or tuples
- Offers customizable object mapper for mapping rows to objects of user-defined classes
- Saves RDDs back to Cassandra by implicit `saveToCassandra` call
- Converts data types between Cassandra and Scala
- Supports all Cassandra data types including collections
- Filters rows on the server side via the CQL `WHERE` clause
- Allows for execution of arbitrary CQL statements
- Plays nice with Cassandra Virtual Nodes

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

- Alan Kay

DEMO: DATABRICKS CLOUD GUI

databricks

SPARK RESOURCE MANAGERS

databricks

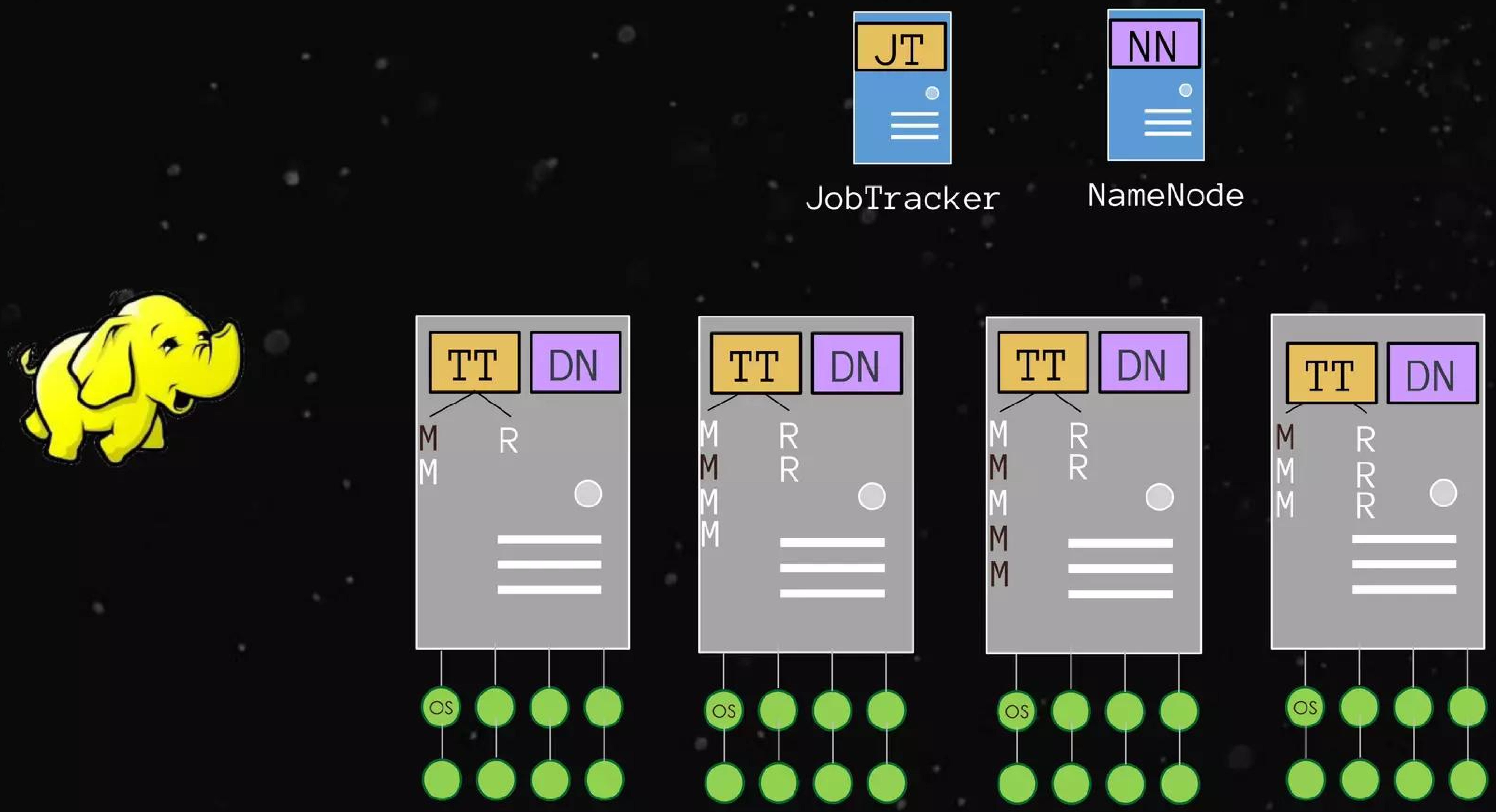
WAYS TO RUN SPARK

- Local
- Standalone Scheduler
- YARN
- Mesos

Static Partitioning

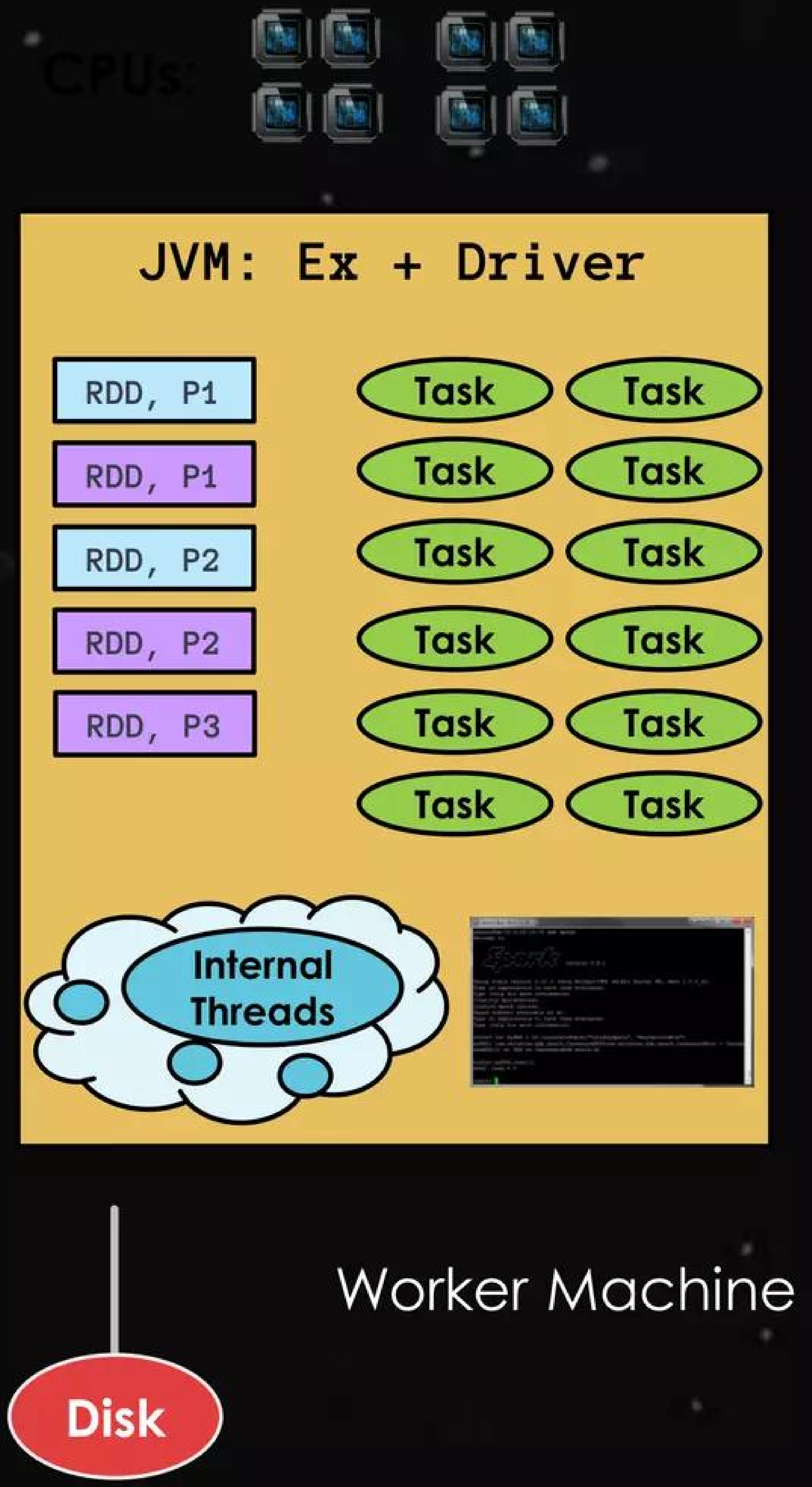
Dynamic Partitioning

History: 2 MR APPS RUNNING



LOCAL MODE

LOCAL MODE




```
3 options:
- local
- local[N]
- local[*]
```

```
> ./bin/spark-shell --master local[12]
```

```
> ./bin/spark-submit --name "MyFirstApp"
                     --master local[12] myApp.jar
```

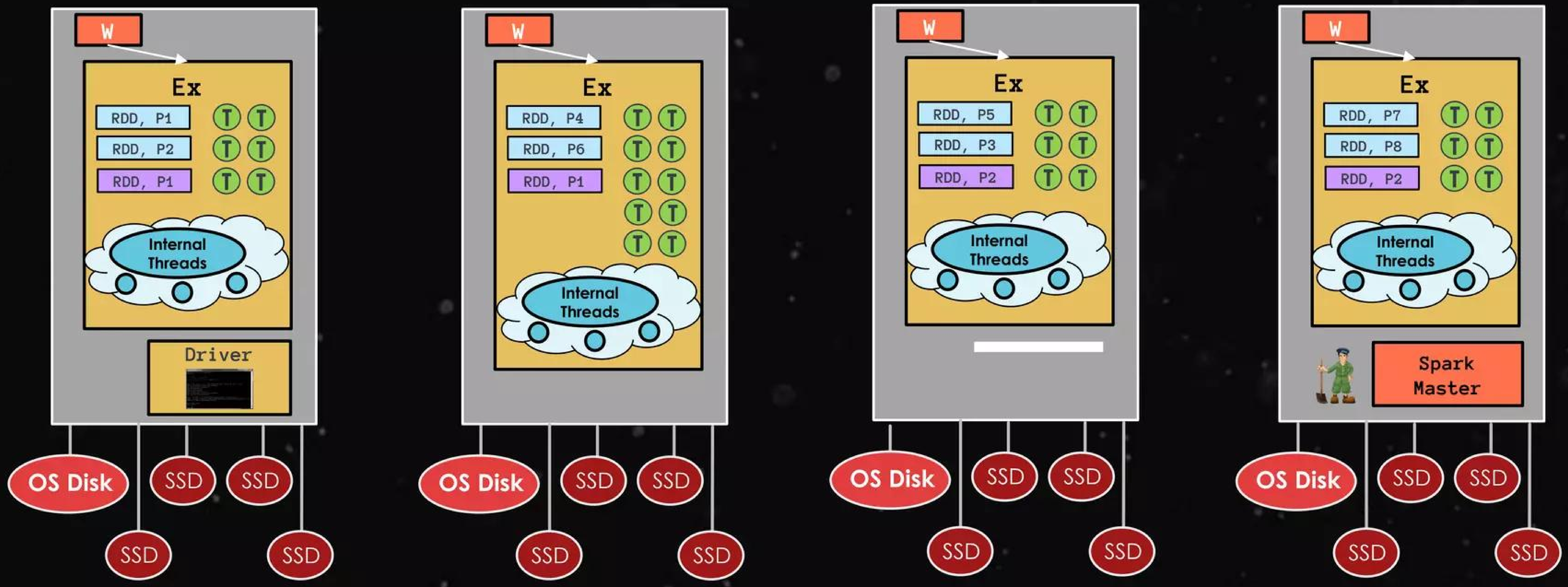
```
val conf = new SparkConf()
                 .setMaster("local[12]")
                 .setAppName("MyFirstApp")
                 .set("spark.executor.memory", "3g")
val sc = new SparkContext(conf)
```

STANDALONE MODE

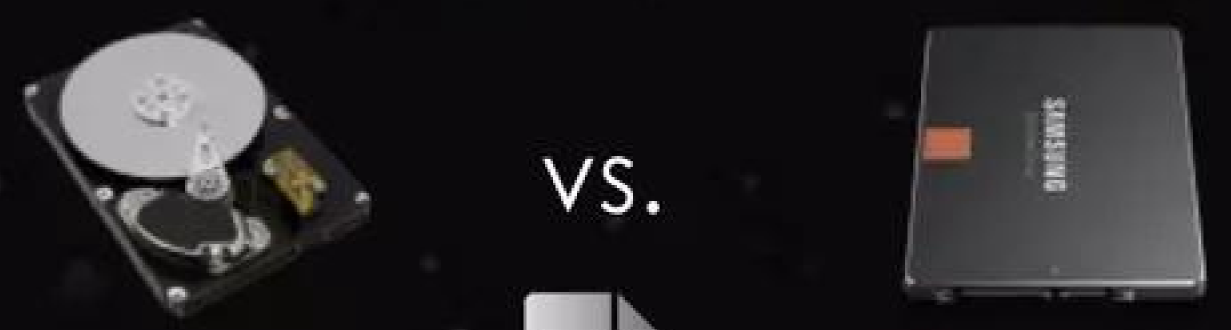
SPARK STANDALONE

different spark-env.sh

- SPARK_WORKER_CORES




```
> ./bin/spark-submit --name "SecondApp"
                     --master spark://host1:port1
                     myApp.jar
```



spark-env.sh

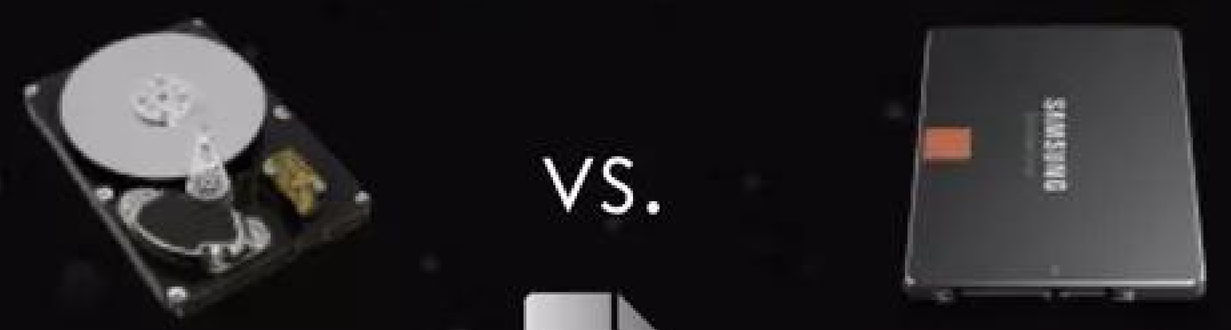
- SPARK_LOCAL_DIRS

SPARK STANDALONE

different spark-env.sh

- SPARK_WORKER_CORES


```
> ./bin/spark-submit --name "SecondApp"
        --master spark://host1:port1,host2:port2
        myApp.jar
```

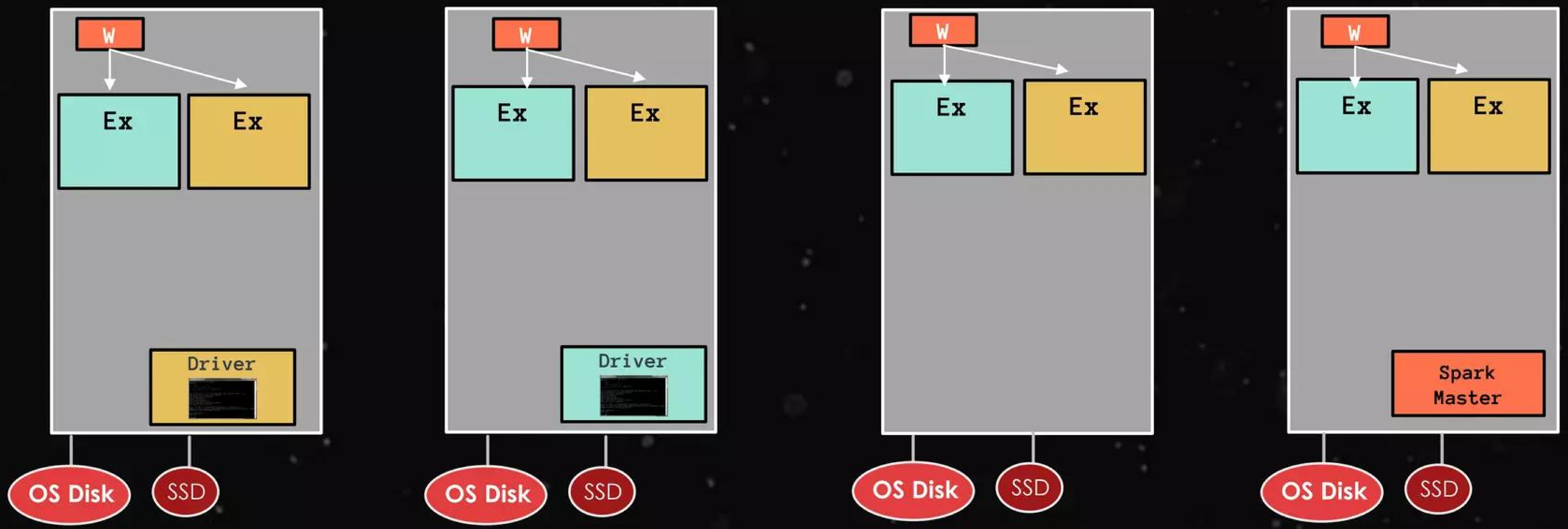


spark-env.sh

- SPARK_LOCAL_DIRS

SPARK STANDALONE

(multiple apps)



SPARK STANDALONE

(single app)

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

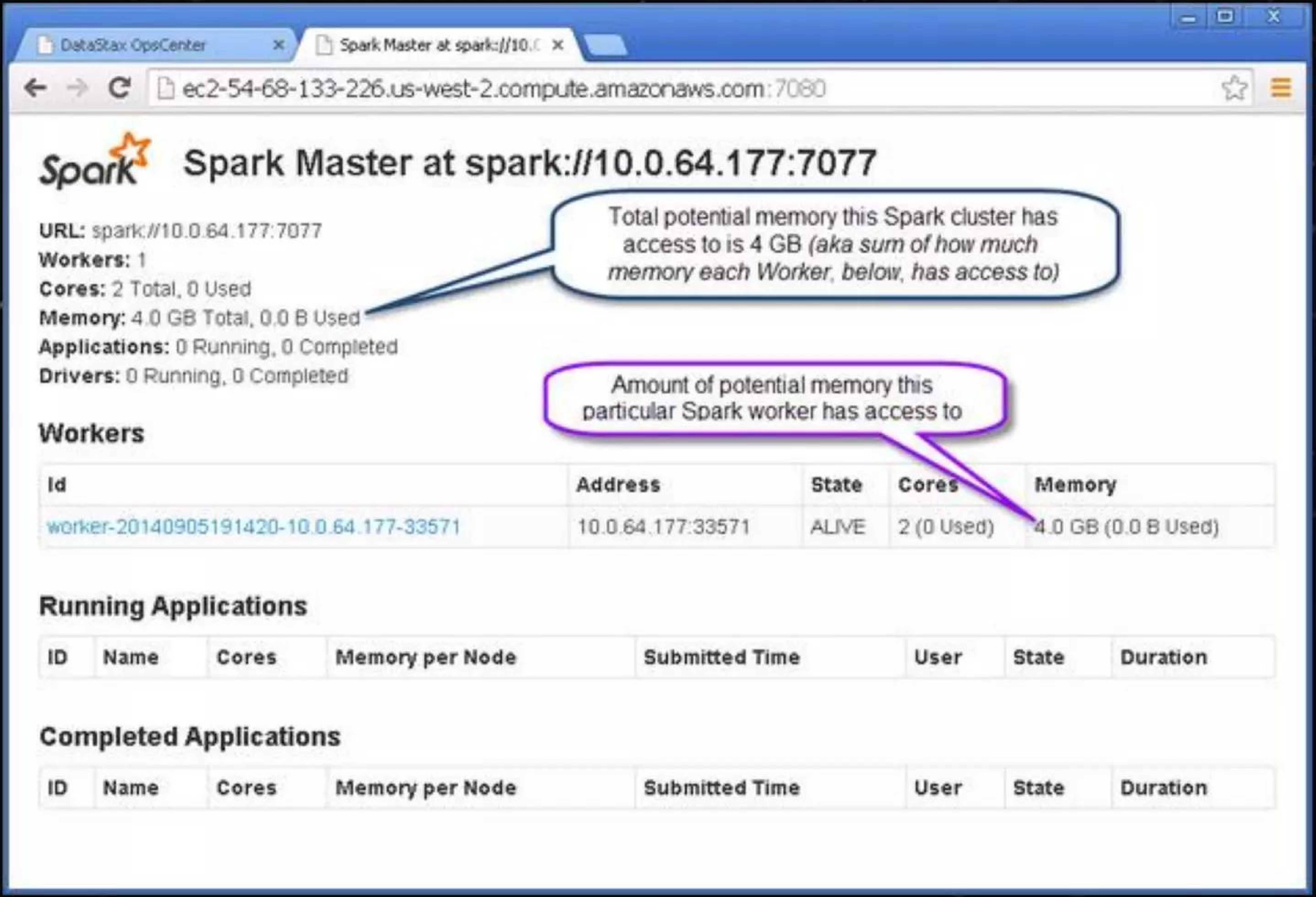
SPARK_DAEMON_MEMORY: [default: 512 MB] Memory to allocate to the Spark master and worker daemons themselves

Standalone 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


DataStax OpsCenter
Spark Master at spark://10.(
ec2-54-68-133-226.us-west-2.compute.amazonaws.com:7080
Spark Master at spark://10.0.64.177:7077
URL: spark://10.0.64.177:7077
Workers: 1
Cores: 2 Total, 0 Used
Memory: 4.0 GB Total, 0.0 B Used
Applications: 0 Running, 0 Completed
Drivers: 0 Running, 0 Completed
Total potential memory this Spark cluster has access to is 4 GB (aka sum of how much memory each Worker, below, has access to)
Amount of potential memory this particular Spark worker has access to
Workers
Id
Address
State
Cores
Memory
worker-20140905191420-10.0.64.177-33571
10.0.64.177:33571
ALIVE
2 (0 Used)
4.0 GB (0.0 B Used)
Running Applications
ID
Name
Cores
Memory per Node
Submitted Time
User
State
Duration
Completed Applications
ID
Name
Cores
Memory per Node
Submitted Time
User
State
Duration

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 Worker at 10.0.12.60:35935

ec2-54-187-238-98.us-west-2.compute.amazonaws.com:7081

Spark Worker at 10.0.12.60:35935

ID: worker-20141110195851-10.0.12.60-35935
Master URL: spark://10.0.12.60:7077
Cores: 3 (3 Used)
Memory: 7.7 GB (512.0 MB Used)

Back to Master

Running Executors (1)

ExecutorID	Cores	State	Memory	Job Details	Logs
0	3	RUNNING	512.0 MB	**ID:** app-20141110204831-0000 **Name:** Spark shell **User:** cassandra	stdout stderr

Spark shell - Spark Jobs

ec2-54-148-231-117.us-west-2.compute.amazonaws.com:4040/jobs/

Spark | 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

ec2-54-148-231-117.us-west-2.compute.amazonaws.com:4040/jobs/

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)

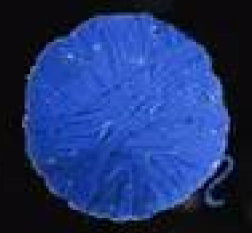
Total Duration: 39 min
Scheduling Mode: FIFO
Active Stages: 0
Completed Stages: 5
Failed Stages: 0

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)

Spark shell - Thread dump

ec2-54-148-231-117.us-west-2.compute.amazonaws.com:4040/executors/threadDump/?executorId=<driver>

Thread 56: qtp1837961181-56 (TIMED_WAITING)

Thread 57: qtp1837961181-57 (TIMED_WAITING)

Thread 58: qtp1837961181-58 (TIMED_WAITING)

Thread 59: Timer-0 (WAITING)

Thread 60: Driver Heartbeater (TIMED_WAITING)

Thread 69: shuffle-server-0 (RUNNABLE)

```
sun.nio.ch.EPollArrayWrapper.epollWait(Native Method)
sun.nio.ch.EPollArrayWrapper.poll(EPollArrayWrapper.java:269)
sun.nio.ch.EPollSelectorImpl.doSelect(EPollSelectorImpl.java:79)
sun.nio.ch.SelectorImpl.lockAndDoSelect(SelectorImpl.java:87)
sun.nio.ch.SelectorImpl.select(SelectorImpl.java:98)
io.netty.channel.nio.NioEventLoop.select(NioEventLoop.java:622)
io.netty.channel.nio.NioEventLoop.run(NioEventLoop.java:310)
io.netty.util.concurrent.SingleThreadEventExecutor$2.run(SingleThreadEventExecutor.java:116)
java.lang.Thread.run(Thread.java:745)
```

Thread 78: Spark Context Cleaner (TIMED_WAITING)

Thread 79: sparkDriver-akka.actor.default-dispatcher-14 (TIMED_WAITING)

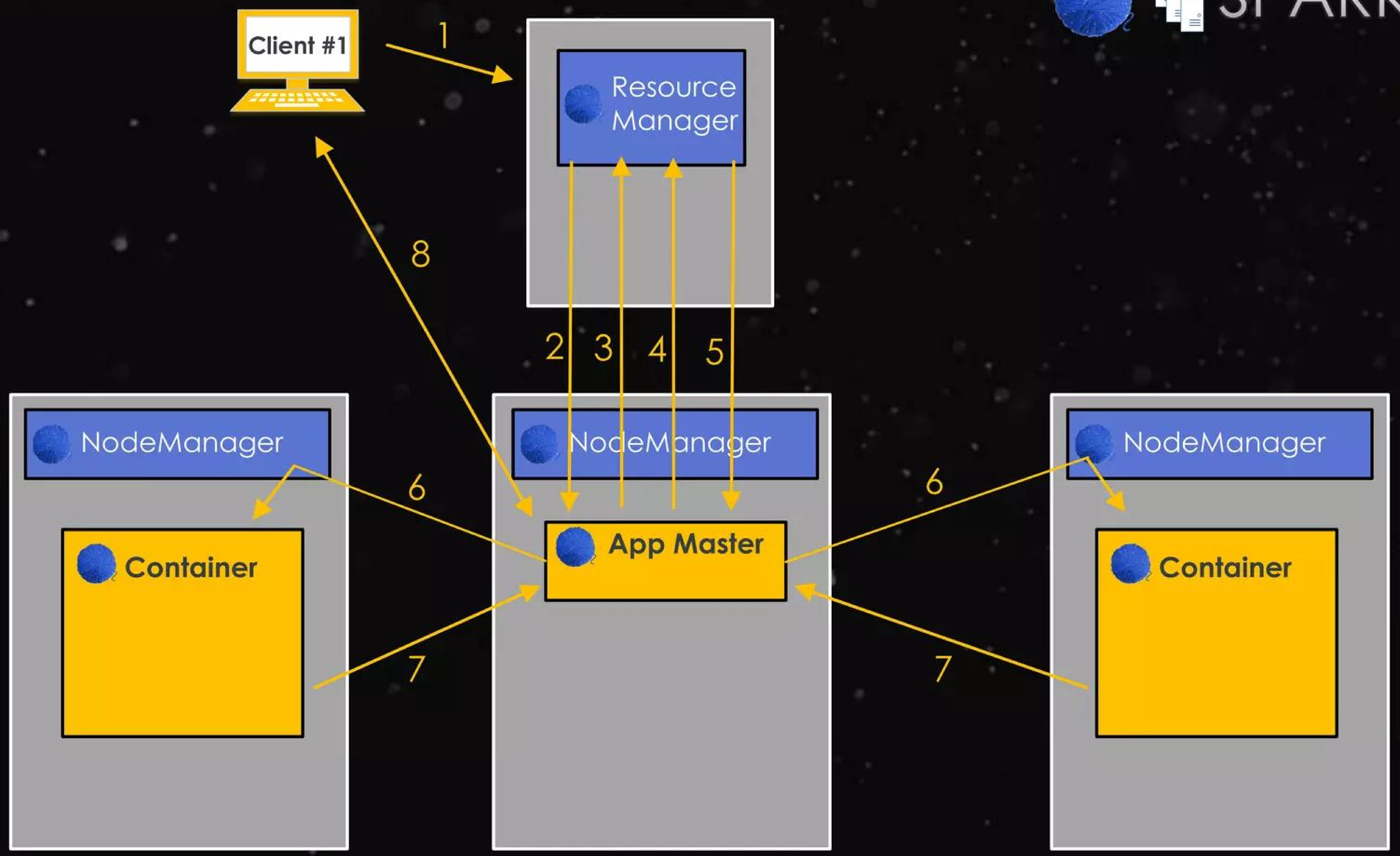
Thread 83: task-result-getter-0 (WAITING)

Thread 84: task-result-getter-1 (WAITING)

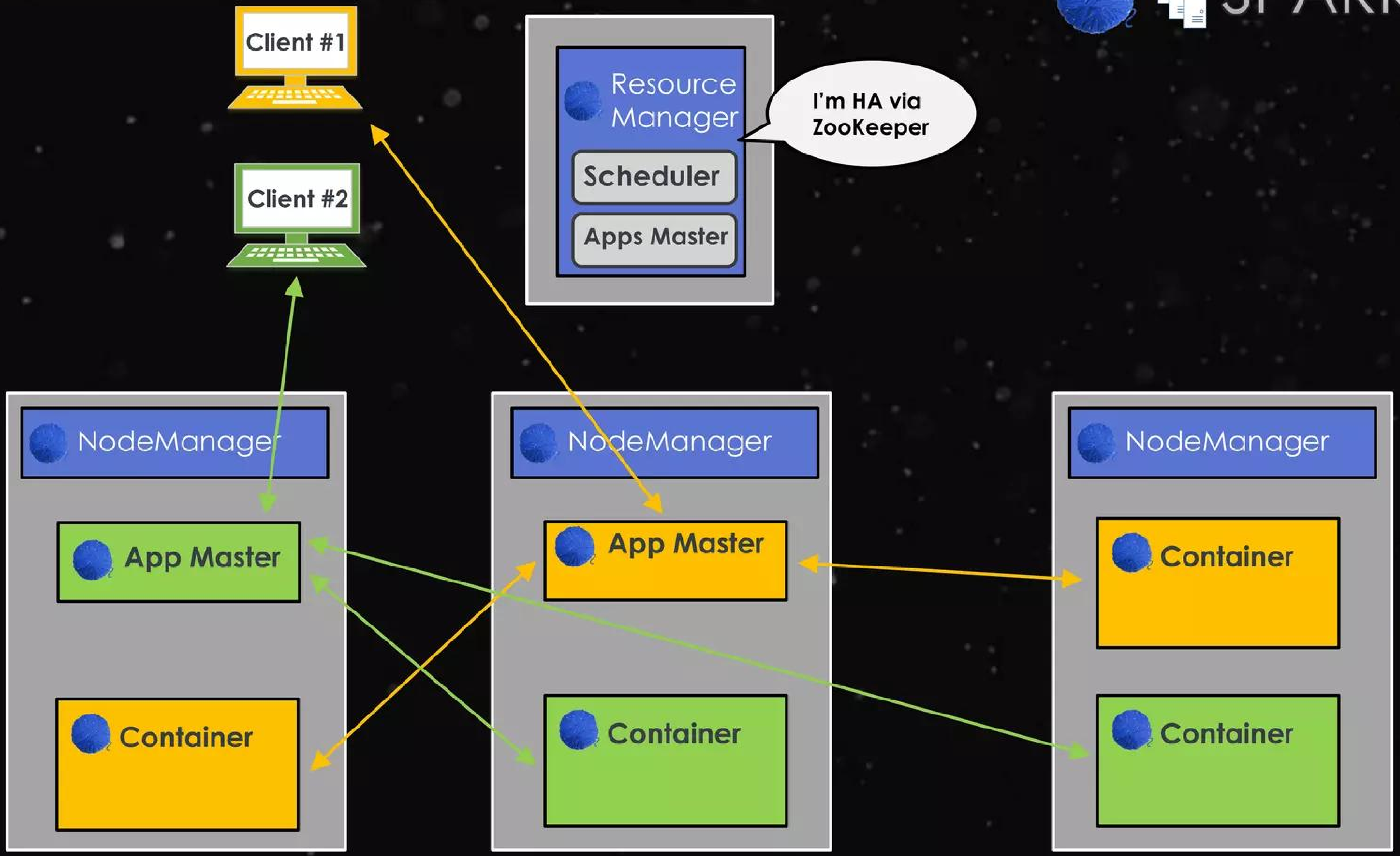
Thread 85: ForkJoinPool-3-worker-7 (WAITING)

YARN MODE

SPARK YARN

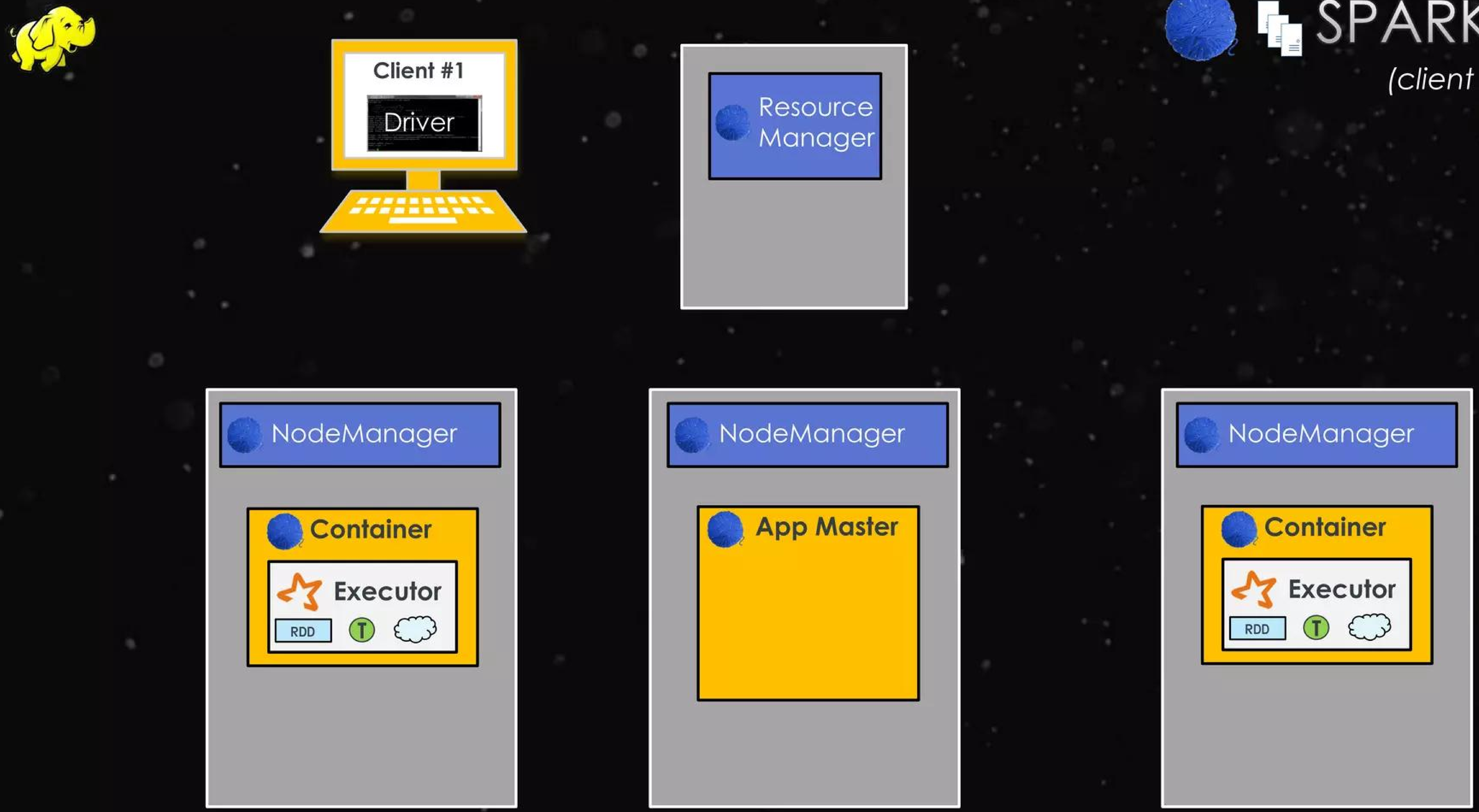

Client #1
1
Resource
Manager
8
2
3
4
5
NodeManager
NodeManager
NodeManager
6
6
App Master
Container
Container
7
7

SPARK YARN


Client #1
Client #2
Resource Manager
Scheduler
Apps Master
I'm HA via ZooKeeper
NodeManager
App Master
Container
NodeManager
App Master
Container
NodeManager
Container
Container

SPARK YARN

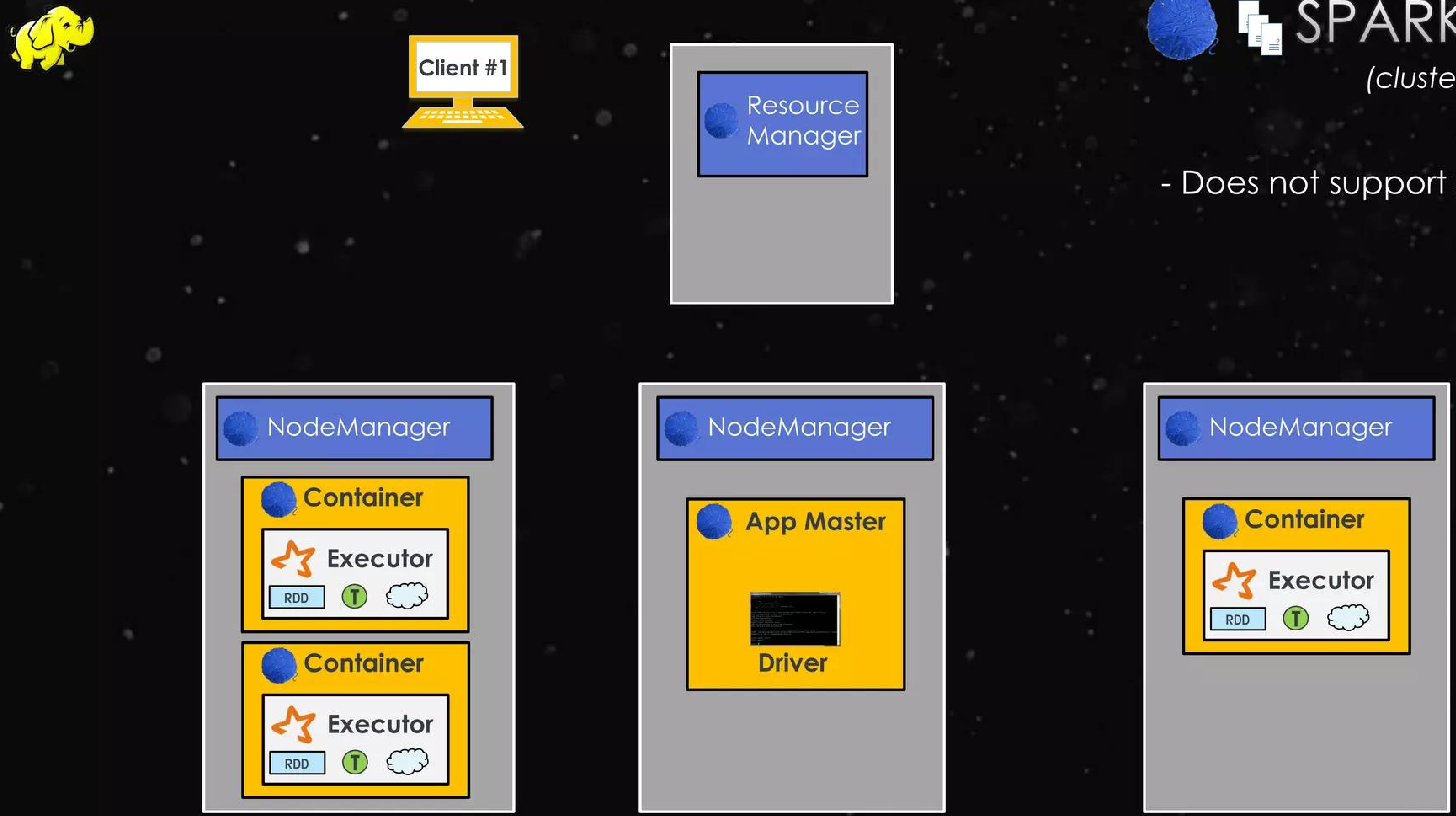
(client mode)



SPARK YARN

(cluster mode)

- Does not support Spark Shells



YARN settings

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

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

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

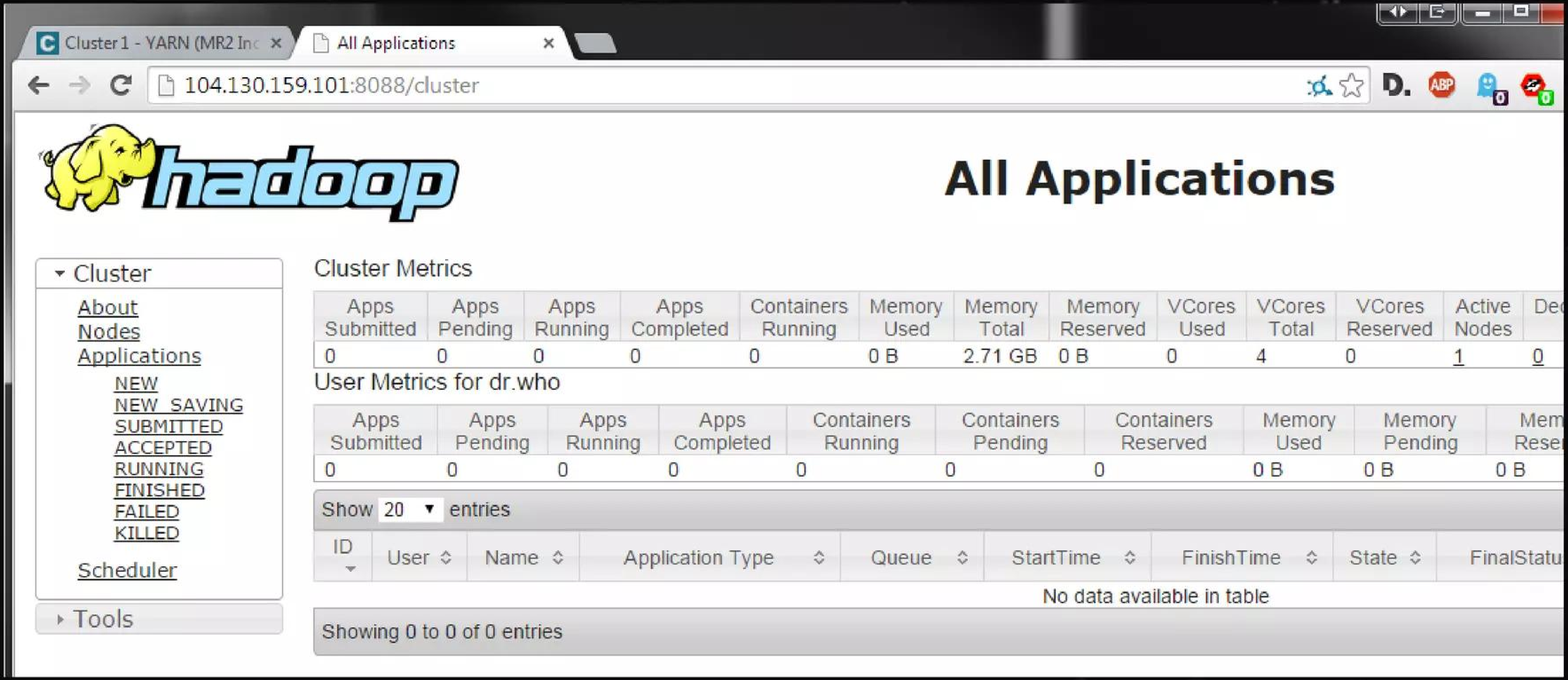
Dynamic Allocation:

```
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

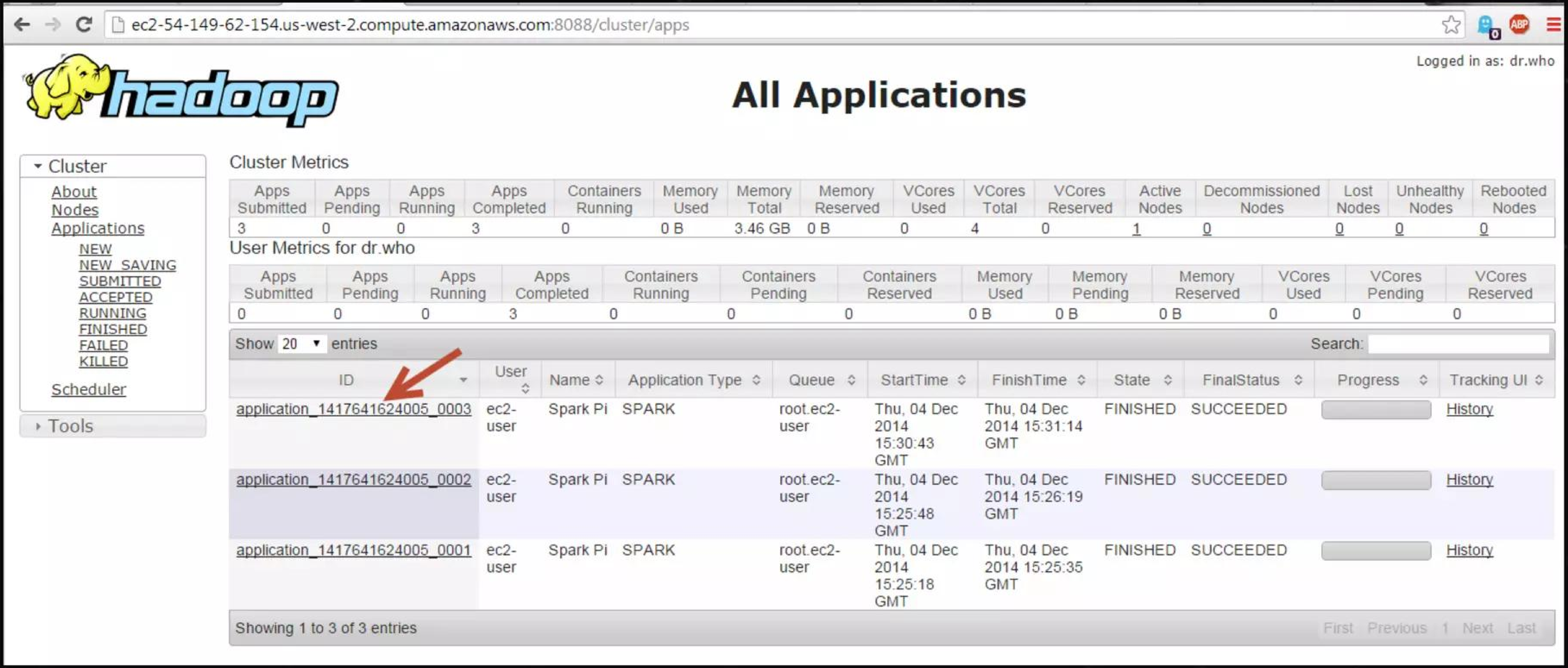
YARN resource manager UI: `http://<ip address>:8088`

(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



ec2-54-149-62-154.us-west-2.compute.amazonaws.com:8088/cluster/apps

Logged in as: dr.who

All Applications

- Cluster
  - About
  - Nodes
  - Applications
    - NEW
    - NEW_SAVING
    - SUBMITTED
    - ACCEPTED
    - RUNNING
    - FINISHED
    - FAILED
    - KILLED
  - Scheduler
- Tools

Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	VCores Used	VCores Total	VCores Reserved	Active Nodes	Decommissioned Nodes	Lost Nodes	Unhealthy Nodes	Rebooted Nodes
3	0	0	3	0	0 B	3.46 GB	0 B	0	4	0	1	0	0	0	0

User Metrics for dr.who

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Containers Pending	Containers Reserved	Memory Used	Memory Pending	Memory Reserved	VCores Used	VCores Pending	VCores Reserved
0	0	0	3	0	0	0	0 B	0 B	0 B	0	0	0

Show 20 entries Search:

ID	User	Name	Application Type	Queue	StartTime	FinishTime	State	FinalStatus	Progress	Tracking UI
application_1417641624005_0003	ec2-user	Spark Pi	SPARK	root.ec2-user	Thu, 04 Dec 2014 15:30:43 GMT	Thu, 04 Dec 2014 15:31:14 GMT	FINISHED	SUCCEEDED		History
application_1417641624005_0002	ec2-user	Spark Pi	SPARK	root.ec2-user	Thu, 04 Dec 2014 15:25:48 GMT	Thu, 04 Dec 2014 15:26:19 GMT	FINISHED	SUCCEEDED		History
application_1417641624005_0001	ec2-user	Spark Pi	SPARK	root.ec2-user	Thu, 04 Dec 2014 15:25:18 GMT	Thu, 04 Dec 2014 15:25:35 GMT	FINISHED	SUCCEEDED		History

Showing 1 to 3 of 3 entries

First Previous 1 Next Last

ec2-54-149-62-154.us-west-2.compute.amazonaws.com:8088/cluster/app/application_1417641624005_0003

Logged in as: dr.who

- Cluster
  - About
  - Nodes
  - Applications
    - NEW
    - NEW_SAVING
    - SUBMITTED
    - ACCEPTED
    - RUNNING
    - FINISHED
    - FAILED
    - KILLED
  - Scheduler
- Tools

Application Overview

User:	ec2-user
Name:	Spark Pi
Application Type:	SPARK
Application Tags:	
State:	FINISHED
FinalStatus:	SUCCEEDED
Started:	4-Dec-2014 10:30:43
Elapsed:	31sec
Tracking URL:	History
Diagnostics:	

Application Metrics

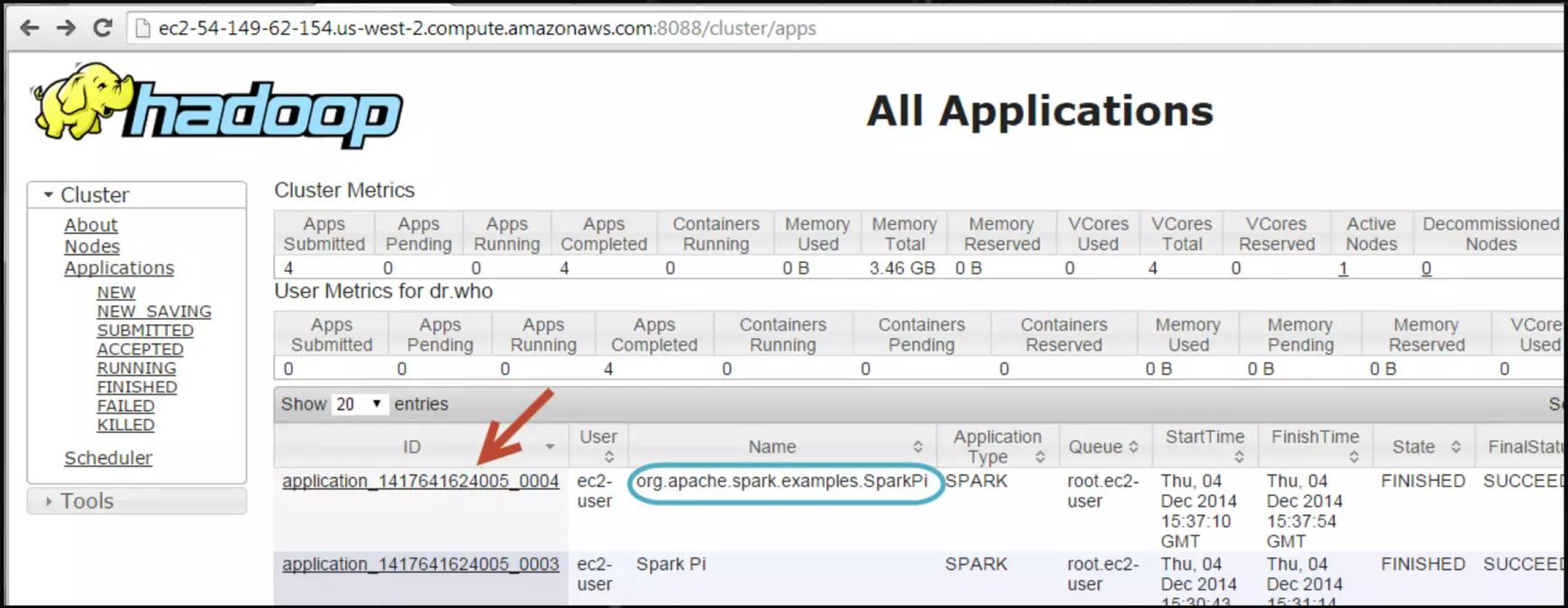
Total Resource Preempted:	<memory:0, vCores:0>
Total Number of Non-AM Containers Preempted:	0
Total Number of AM Containers Preempted:	0
Resource Preempted from Current Attempt:	<memory:0, vCores:0>
Number of Non-AM Containers Preempted from Current Attempt:	0
Aggregate Resource Allocation:	57388 MB-seconds, 45 vcore-seconds

ApplicationMaster

Attempt Number	Start Time	Node	Logs
1	4-Dec-2014 10:30:43	ip-10-0-72-36.us-west-2.compute.internal:8042	logs

```
[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
```

App running in cluster mode



ec2-54-149-62-154.us-west-2.compute.amazonaws.com:8088/cluster/apps

hadoop

All Applications

- Cluster
  - About
  - Nodes
  - Applications
    - NEW
    - NEW_SAVING
    - SUBMITTED
    - ACCEPTED
    - RUNNING
    - FINISHED
    - FAILED
    - KILLED
  - Scheduler
- Tools

Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	VCores Used	VCores Total	VCores Reserved	Active Nodes	Decommissioned Nodes
4	0	0	4	0	0 B	3.46 GB	0 B	0	4	0	1	0

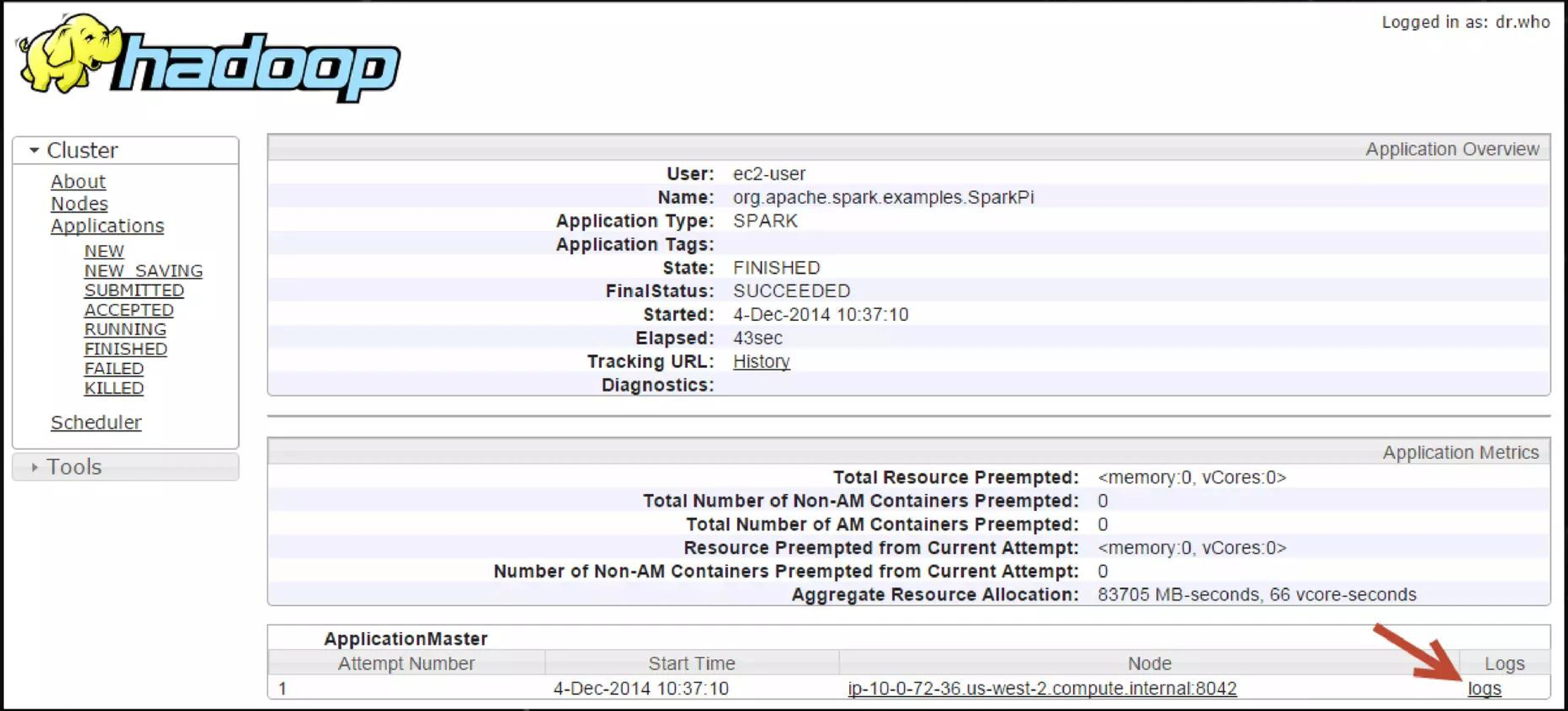
User Metrics for dr.who

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Containers Pending	Containers Reserved	Memory Used	Memory Pending	Memory Reserved	VCore Used
0	0	0	4	0	0	0	0 B	0 B	0 B	0

Show 20 entries

ID	User	Name	Application Type	Queue	StartTime	FinishTime	State	FinalStat
application_1417641624005_0004	ec2-user	org.apache.spark.examples.SparkPi	SPARK	root.ec2-user	Thu, 04 Dec 2014 15:37:10 GMT	Thu, 04 Dec 2014 15:37:54 GMT	FINISHED	SUCCEEI
application_1417641624005_0003	ec2-user	Spark Pi	SPARK	root.ec2-user	Thu, 04 Dec 2014	Thu, 04 Dec 2014	FINISHED	SUCCEEI

App running in cluster mode



App running in cluster mode

ec2-54-149-62-154.us-west-2.compute.amazonaws.com:19888/jobhistory/logs/ip-10-0-72-36.us-west-2.compute.internal

hadoop

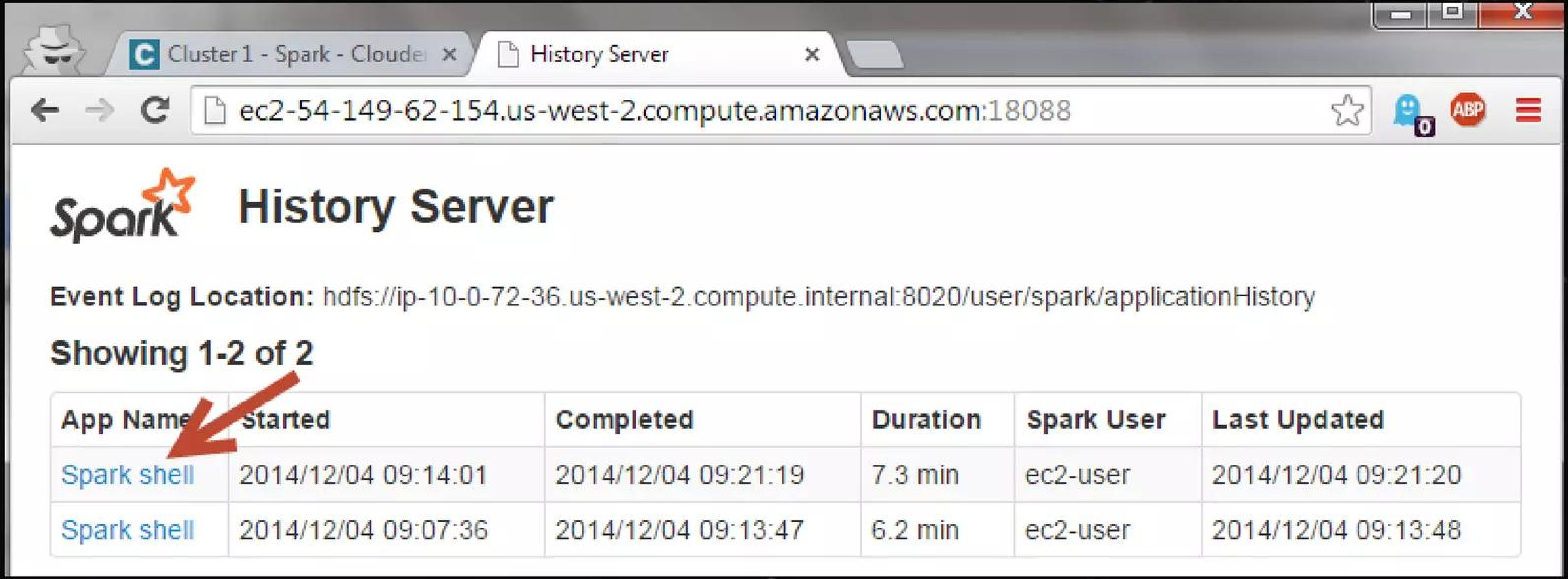
- Application
  - About
  - Jobs
- Tools

Log Type: stderr
Log Length: 22704
Showing 4096 bytes of 22704 total. Click here for the full log.

```
/.sparkStaging/application_1417641624005_0004/spark-assembly-1.1.0-cdh5.2.1-hadoop2.5.0-cdh5.2.1.jar" } size: 95571683 timestam
14/12/04 10:37:52 INFO yarn.YarnAllocationHandler: Completed container container_1417641624005_0004_01_000002 (state: COMPLETE,
14/12/04 10:37:52 INFO yarn.ExecutorRunnable: Setting up executor with environment: Map(CLASSPATH -> $PWD:$PWD/__spark__.jar:$H
14/12/04 10:37:52 INFO yarn.ExecutorRunnable: Setting up executor with commands: List($JAVA_HOME/bin/java, -server, -XX:OnOutOf
14/12/04 10:37:52 INFO impl.ContainerManagementProtocolProxy: Opening proxy : ip-10-0-72-36.us-west-2.compute.internal:8041
14/12/04 10:37:52 INFO yarn.ApplicationMaster: Allocating 1 containers to make up for (potentially) lost containers
14/12/04 10:37:52 INFO yarn.YarnAllocationHandler: Will Allocate 1 executor containers, each with 1408 memory
14/12/04 10:37:52 INFO yarn.YarnAllocationHandler: Container request (host: Any, priority: 1, capability: <memory:1408, vCores:
14/12/04 10:37:53 INFO spark.MapOutputTrackerMasterActor: MapOutputTrackerActor stopped!
14/12/04 10:37:53 INFO network.ConnectionManager: Selector thread was interrupted!
14/12/04 10:37:53 INFO network.ConnectionManager: ConnectionManager stopped
14/12/04 10:37:53 INFO storage.MemoryStore: MemoryStore cleared
14/12/04 10:37:53 INFO storage.BlockManager: BlockManager stopped
14/12/04 10:37:53 INFO storage.BlockManagerMaster: BlockManagerMaster stopped
14/12/04 10:37:53 INFO remote.RemoteActorRefProvider$RemotingTerminator: Shutting down remote daemon.
14/12/04 10:37:53 INFO remote.RemoteActorRefProvider$RemotingTerminator: Remote daemon shut down; proceeding with flushing remo
14/12/04 10:37:53 INFO Remoting: Remoting shut down
14/12/04 10:37:53 INFO remote.RemoteActorRefProvider$RemotingTerminator: Remoting shut down.
14/12/04 10:37:54 INFO spark.SparkContext: Successfully stopped SparkContext
14/12/04 10:37:54 INFO yarn.ApplicationMaster: Unregistering ApplicationMaster with SUCCEEDED
14/12/04 10:37:54 INFO impl.AMRMClientImpl: Waiting for application to be successfully unregistered.
14/12/04 10:37:54 INFO yarn.ApplicationMaster: All executors have launched.
14/12/04 10:37:54 INFO yarn.ApplicationMaster: Started progress reporter thread - heartbeat interval : 5000
14/12/04 10:37:54 INFO yarn.ApplicationMaster: AppMaster received a signal.
14/12/04 10:37:54 INFO yarn.ApplicationMaster: Deleting staging directory .sparkStaging/application_1417641624005_0004
14/12/04 10:37:54 INFO yarn.ApplicationMaster$$anon$1: Invoking sc stop from shutdown hook
14/12/04 10:37:54 INFO ui.SparkUI: Stopped Spark web UI at http://ip-10-0-72-36.us-west-2.compute.internal:41025
14/12/04 10:37:54 INFO spark.SparkContext: SparkContext already stopped
```

Log Type: stdout
Log Length: 23

```
Pi is roughly 3.142392
```



History Server

Event Log Location: hdfs://ip-10-0-72-36.us-west-2.compute.internal:8020/user/spark/applicationHistory

Showing 1-2 of 2

App Name	Started	Completed	Duration	Spark User	Last Updated
Spark shell	2014/12/04 09:14:01	2014/12/04 09:21:19	7.3 min	ec2-user	2014/12/04 09:21:20
Spark shell	2014/12/04 09:07:36	2014/12/04 09:13:47	6.2 min	ec2-user	2014/12/04 09:13:48

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
```

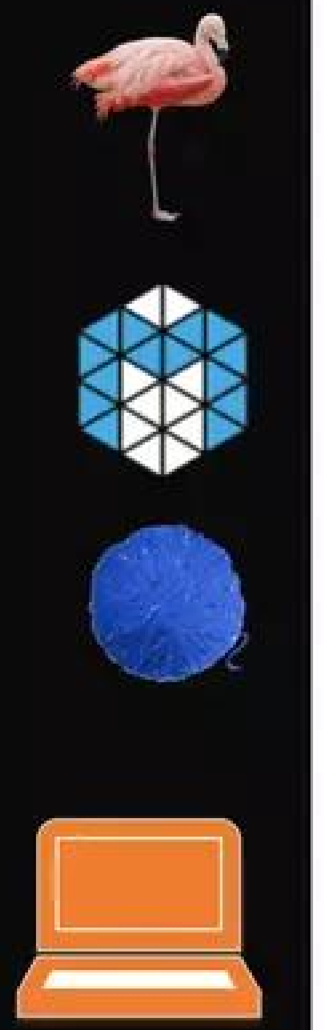


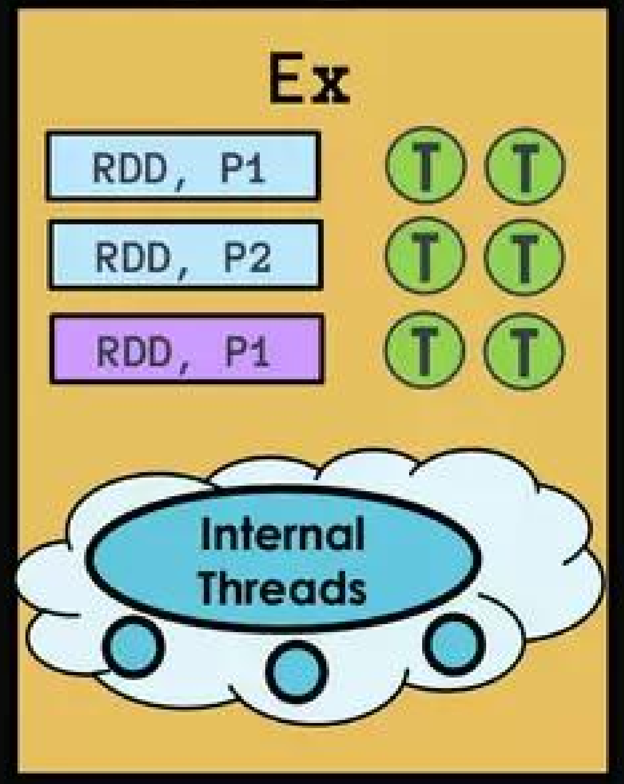
Table 7-2. Possible values for the `--master` flag in `spark-submit`

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

MEMORY AND PERSISTENCE

databricks

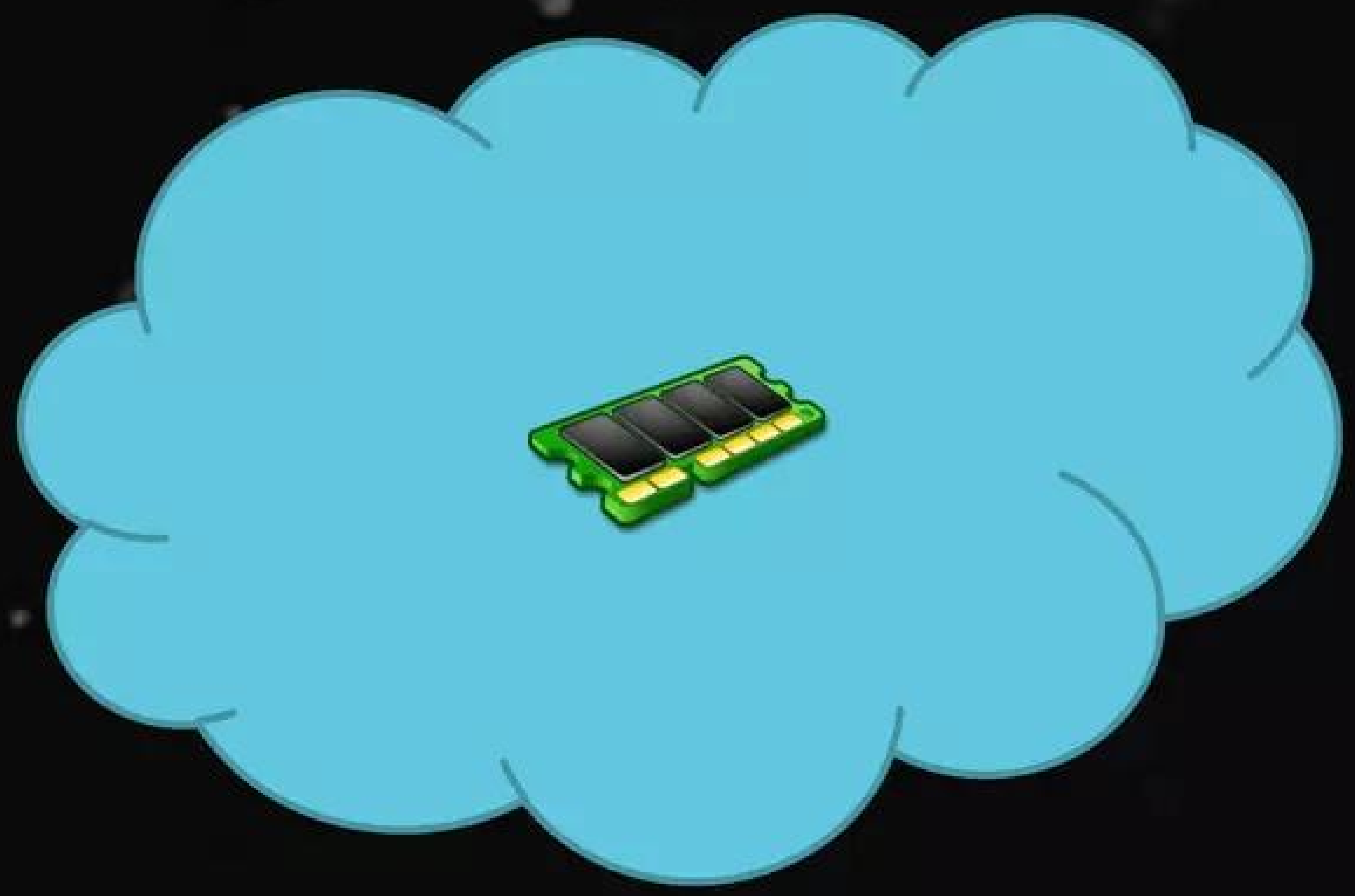


Recommended to use at most only 75% of a machine's memory for Spark

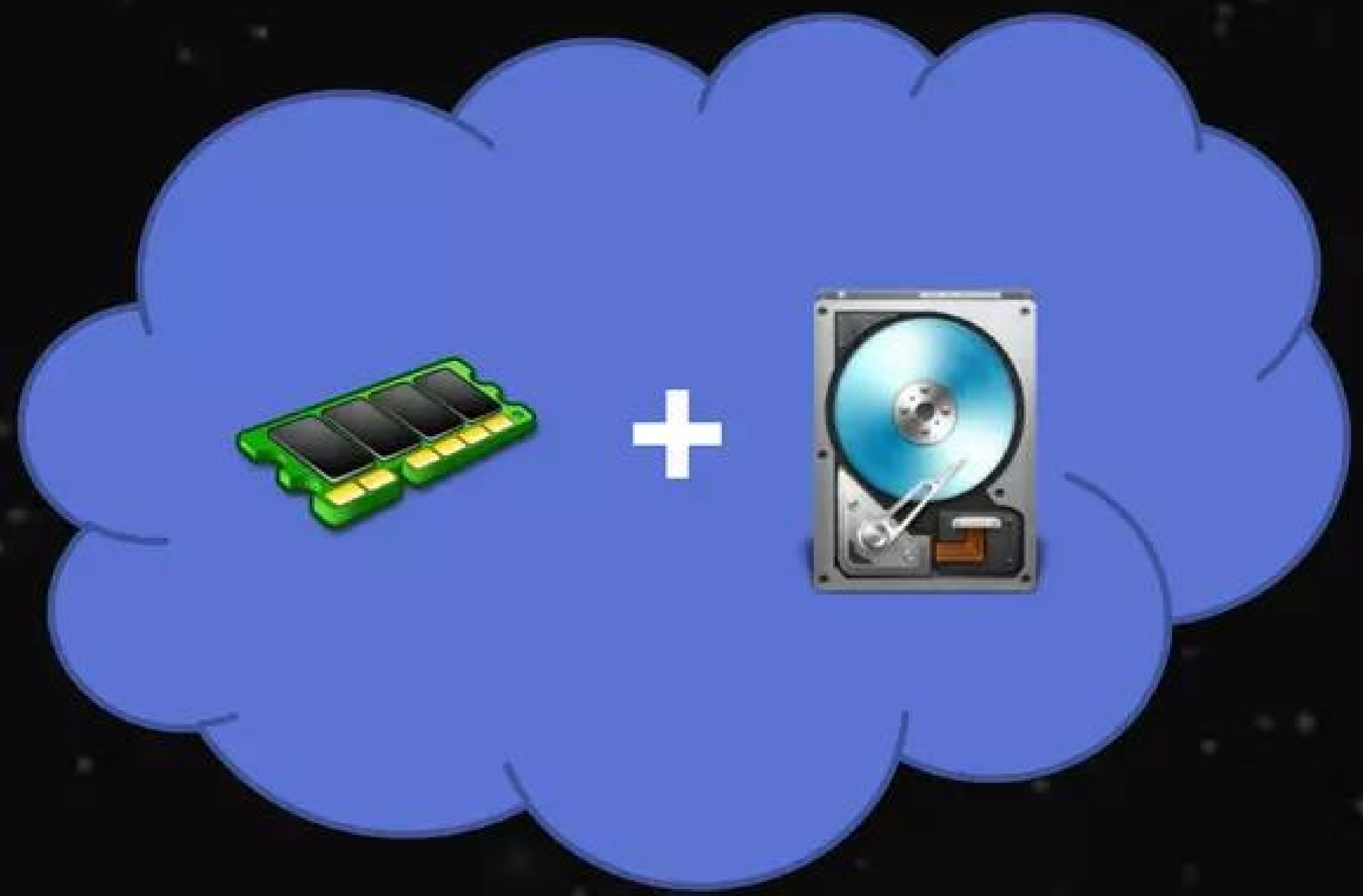
Minimum Executor heap size should be 8 GB

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

Memory usage is greatly affected by storage level and serialization format



Vs.



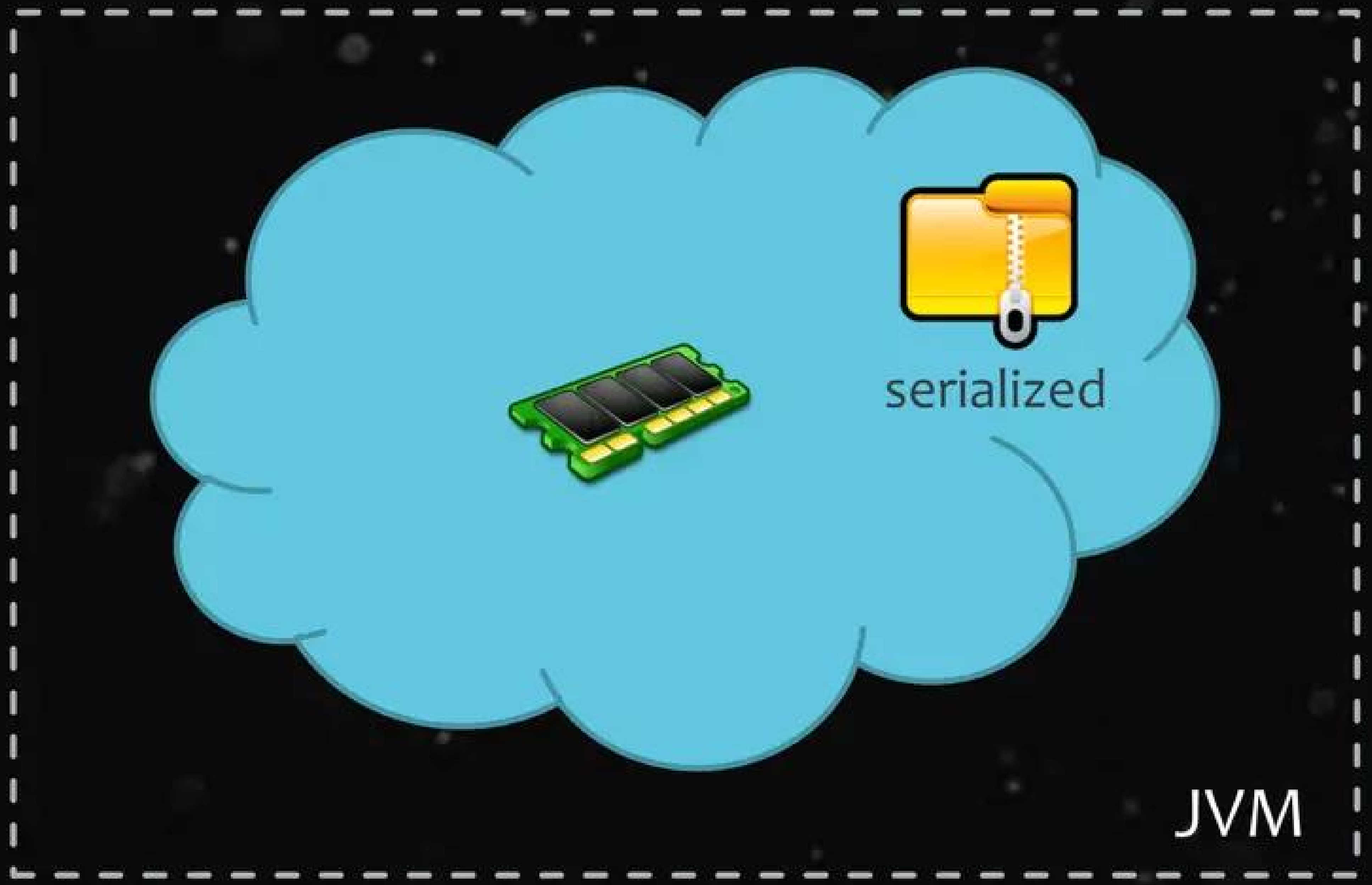
`RDD.cache() == RDD.persist(MEMORY_ONLY)`

most CPU-efficient option

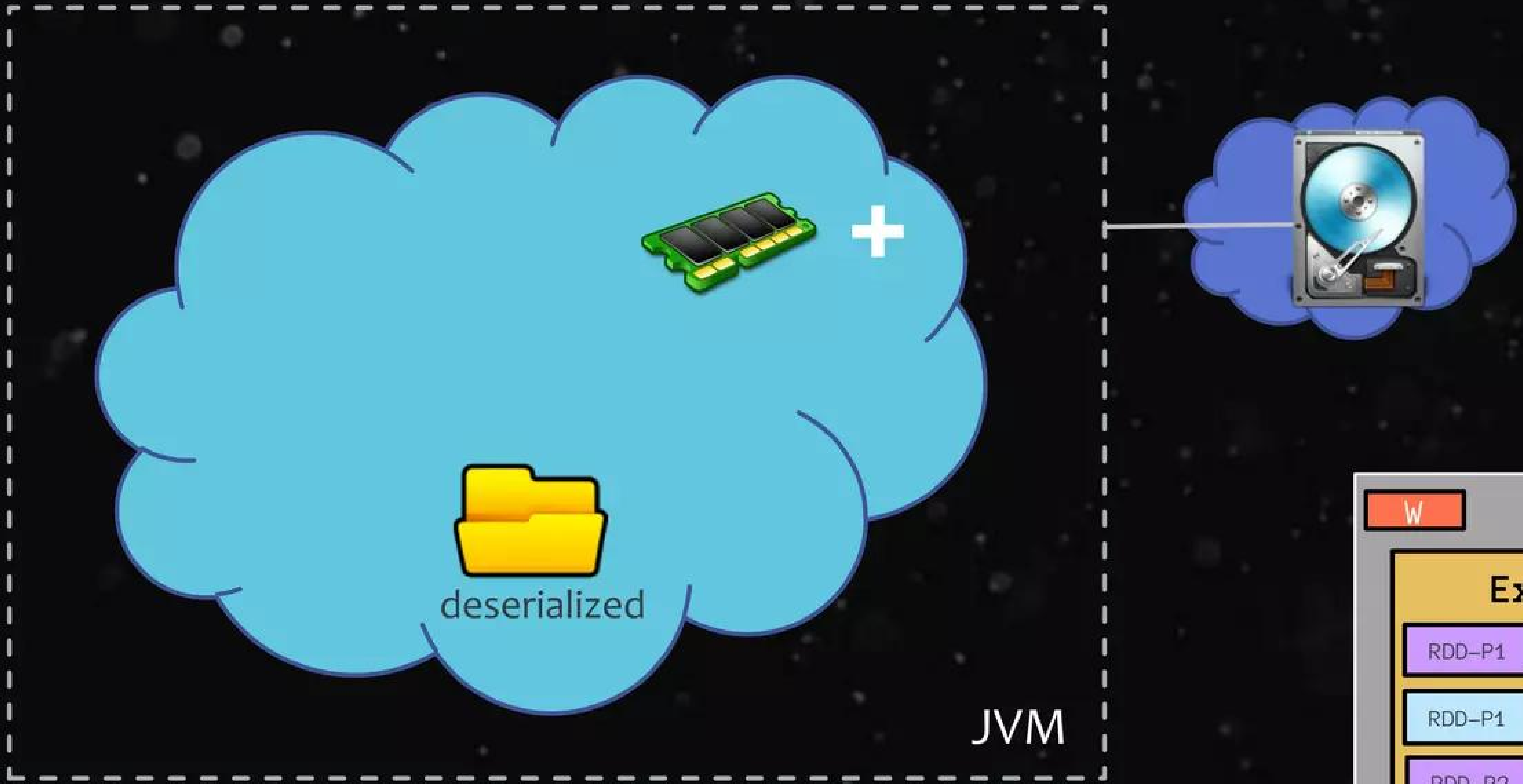
Spark Stages Storage Environment Executors **Spark shell** application UI

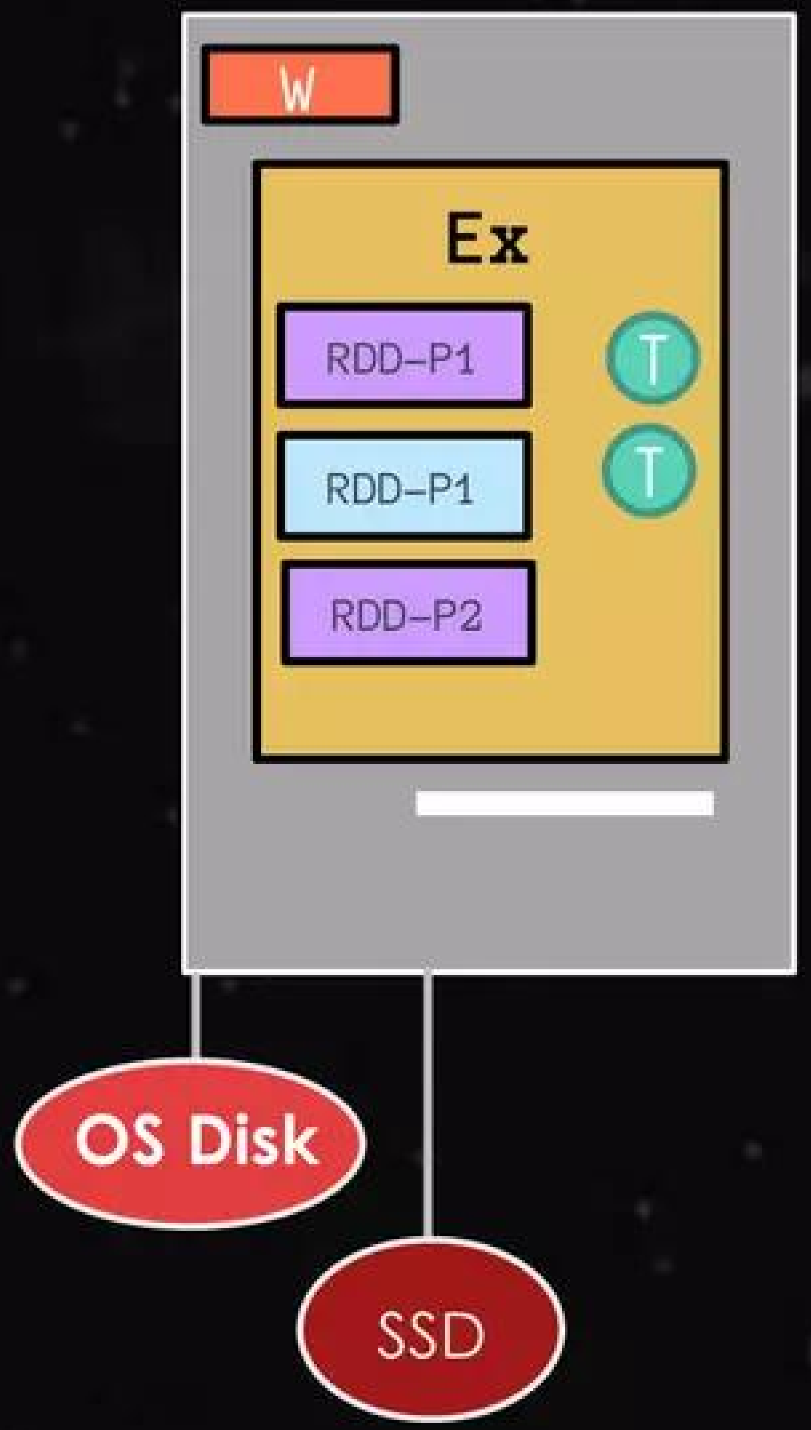
Storage

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
0	Memory Deserialized 1x Replicated	2	100%	55.6 KB	0.0 B

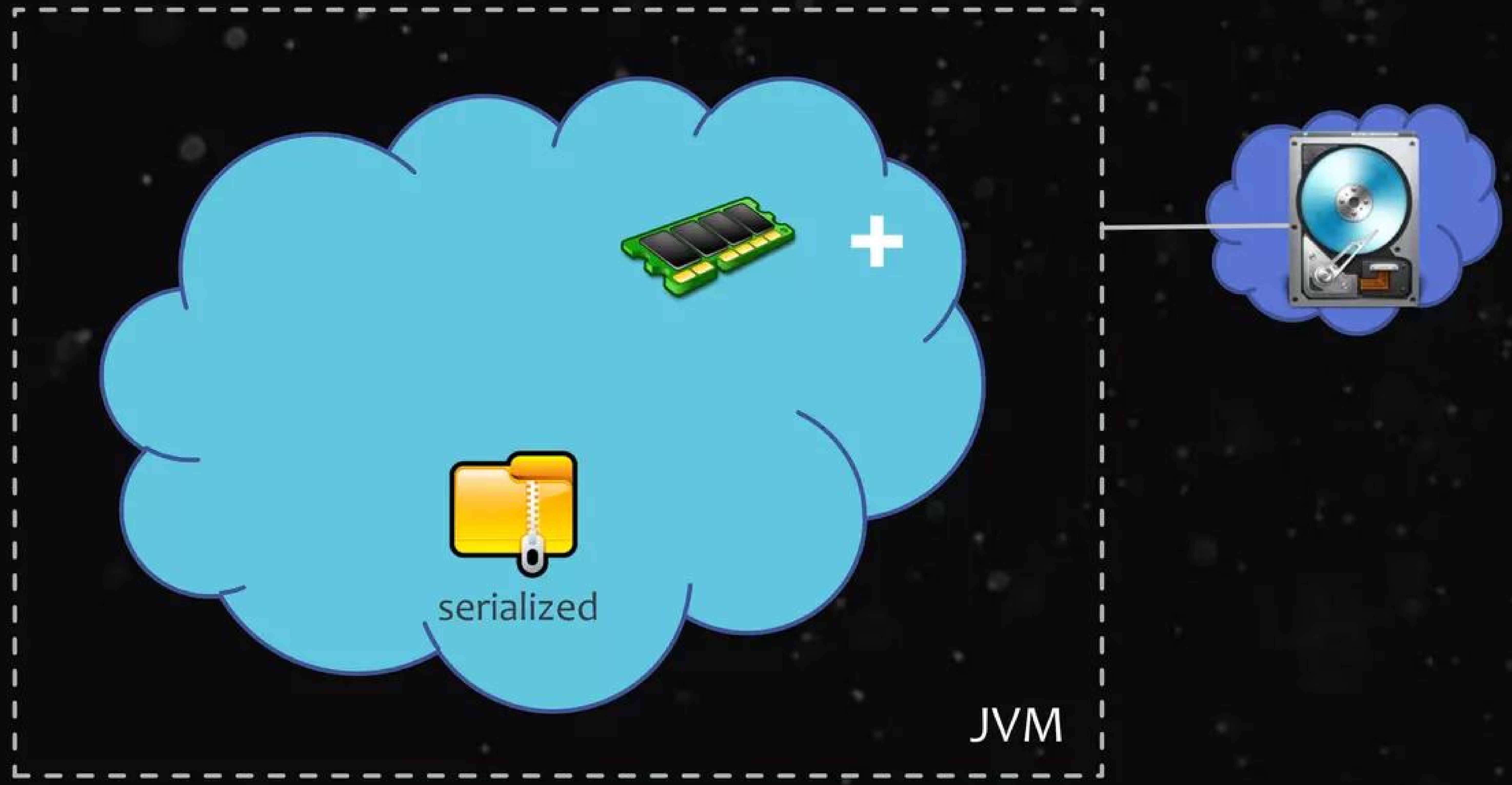


RDD.persist(MEMORY_ONLY_SER)



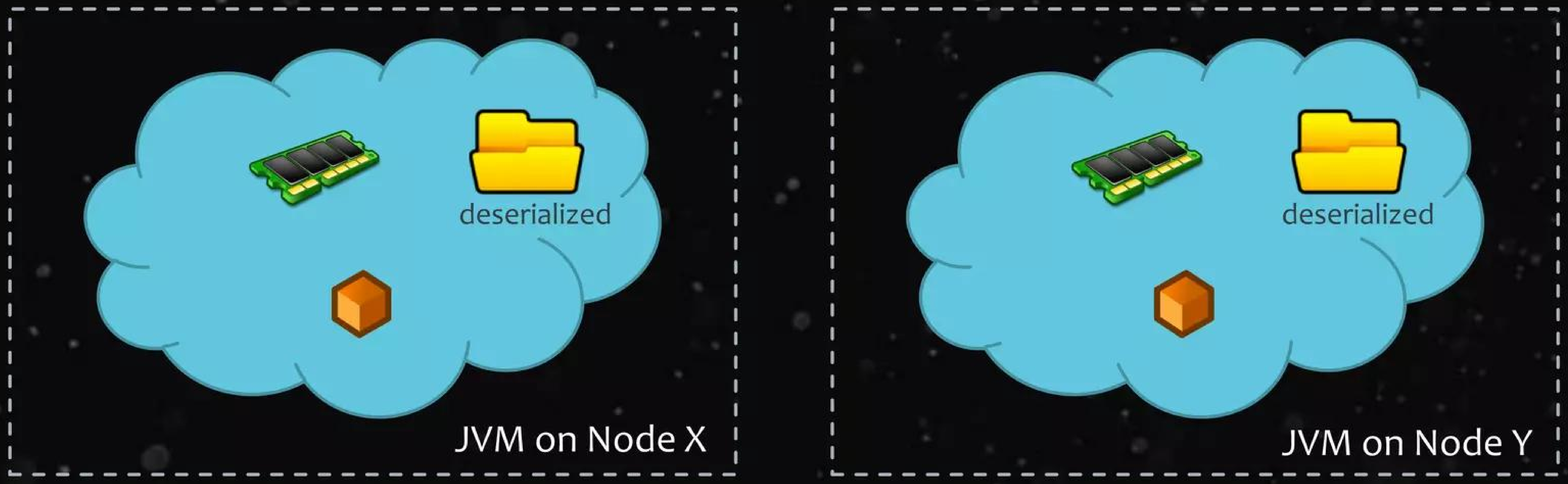



```
.persist(MEMORY_AND_DISK)
```

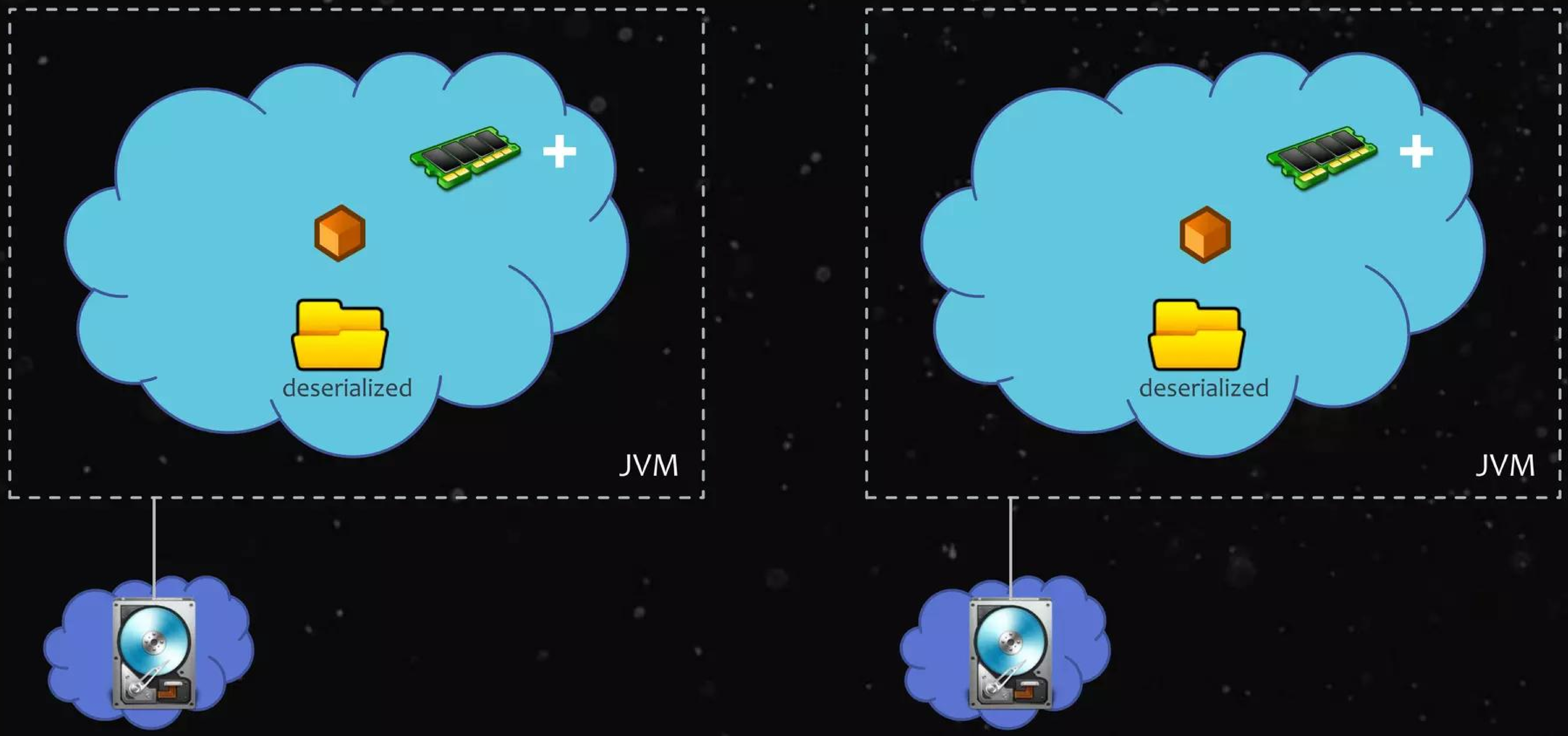


.persist(MEMORY_AND_DISK_SER)

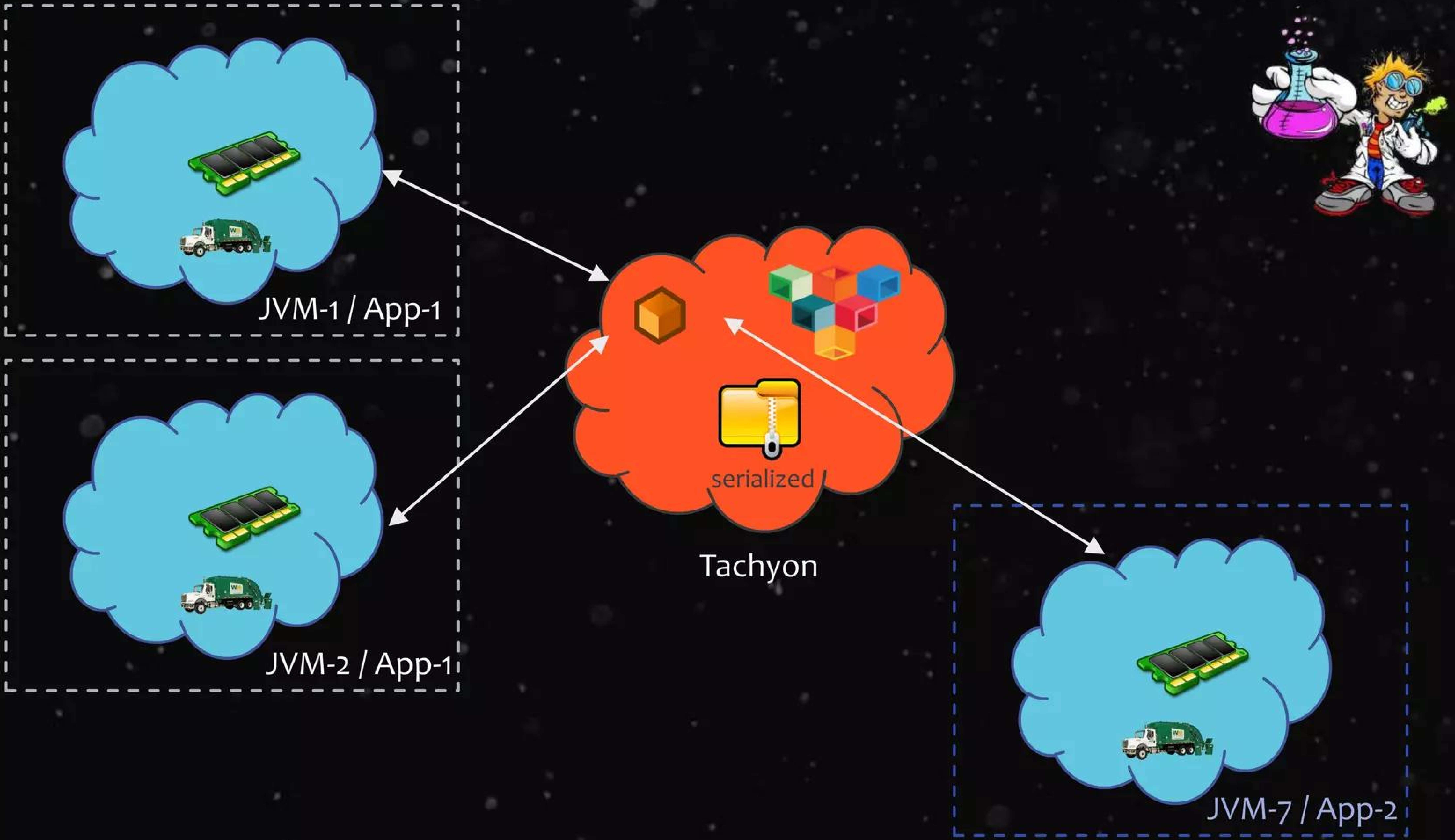
.persist(DISK_ONLY)



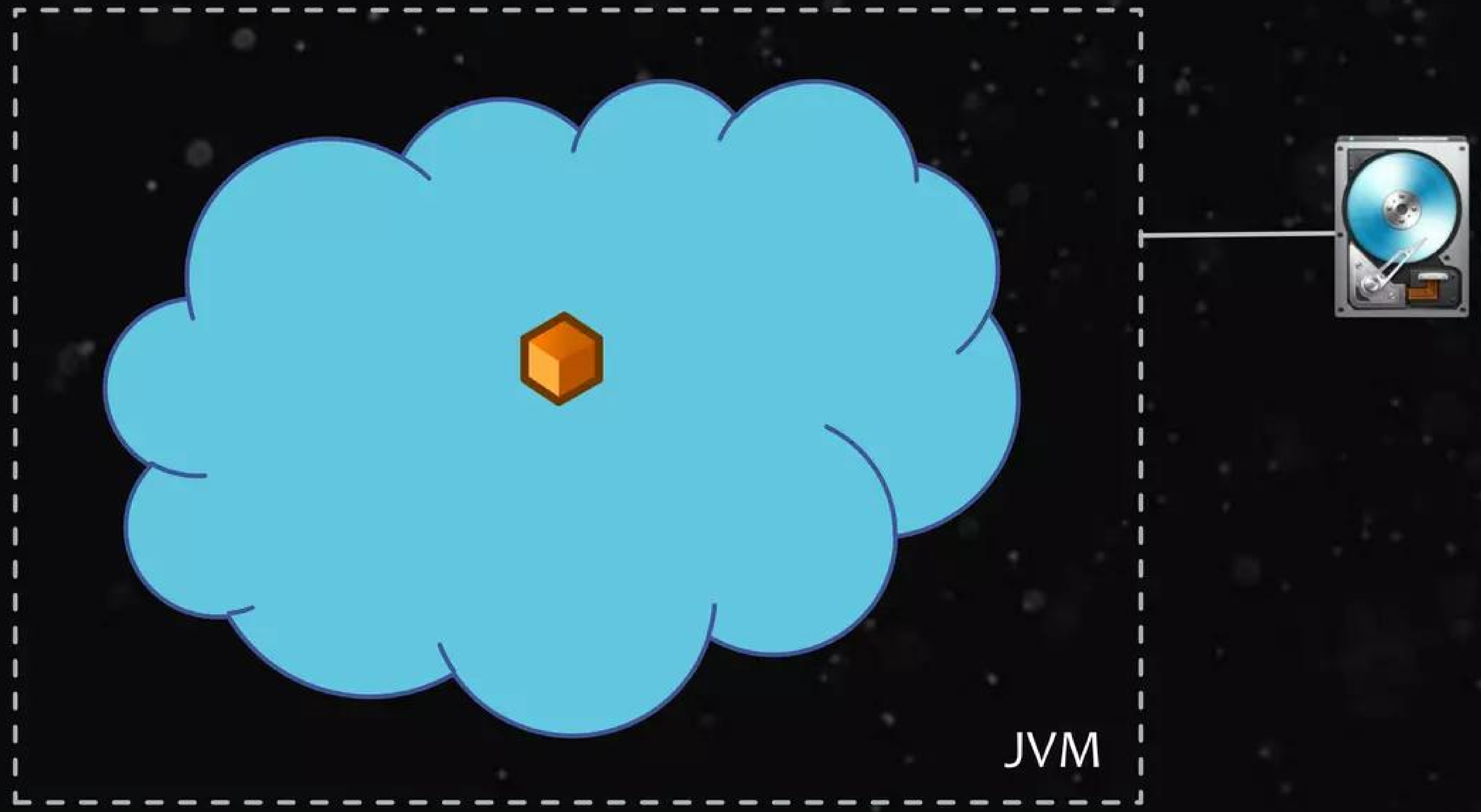

```
RDD.persist(MEMORY_ONLY_2)
```



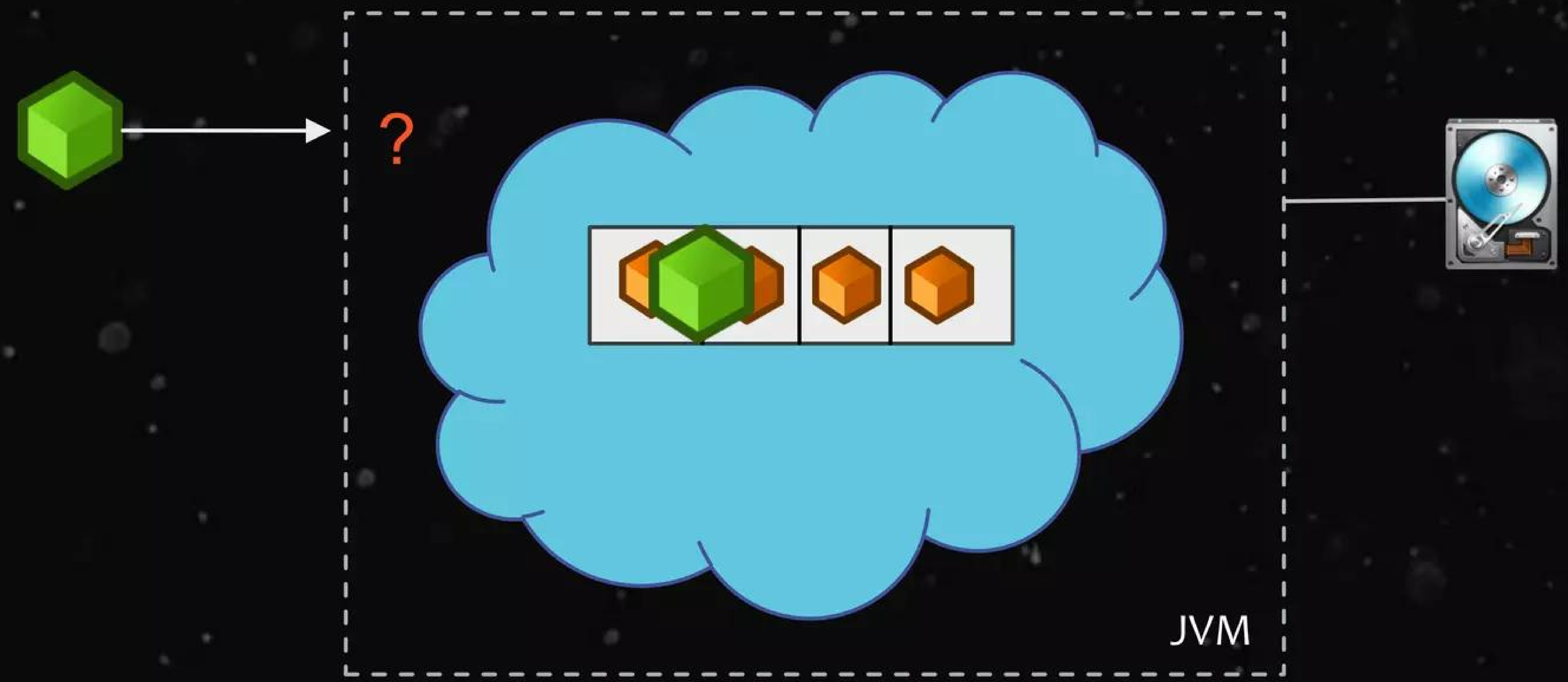

```
.persist(MEMORY_AND_DISK_2)
```



`.persist(OFF_HEAP)`



`.unpersist()`


?
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!

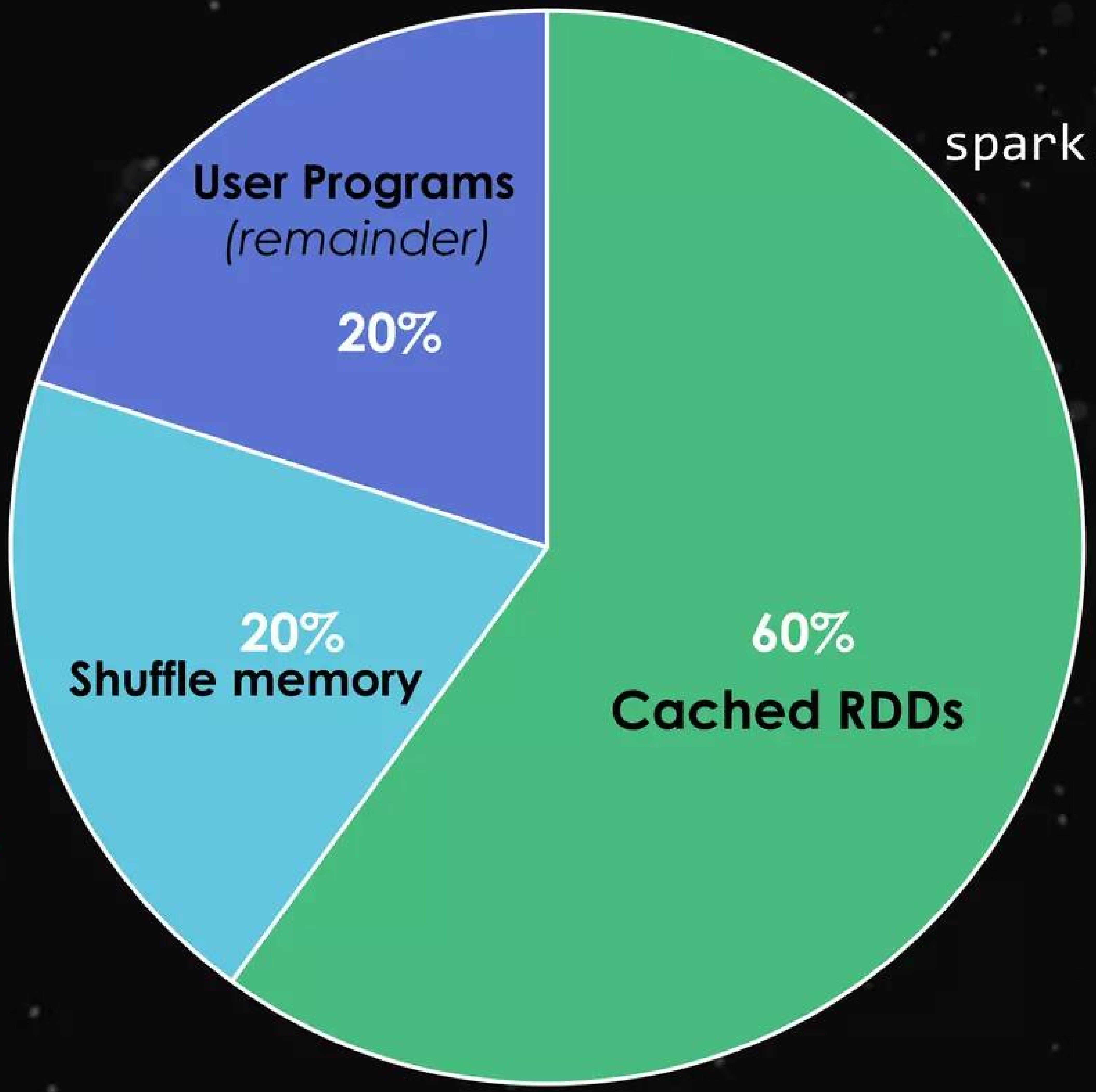
Intermediate data is automatically persisted during shuffle operations

=

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

`spark.storage.memoryFraction`



FIX THIS

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

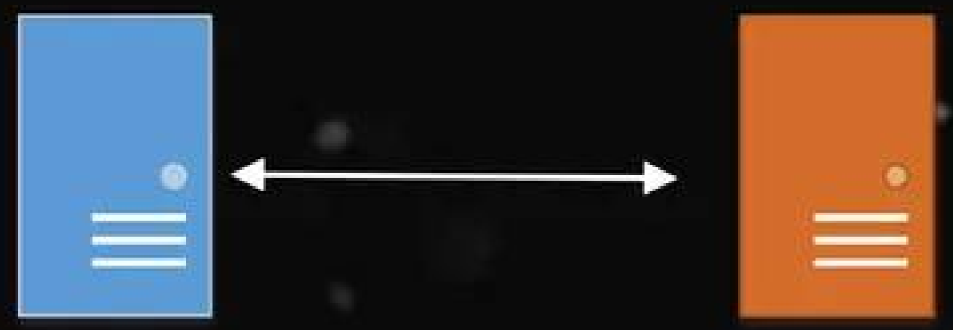
```
INFO BlockManagerMasterActor: Added rdd_0_1 in memory on mbk.local:50311 (size: 717.5 KB, free: 332.3 MB)
```

DATA SERIALIZATION

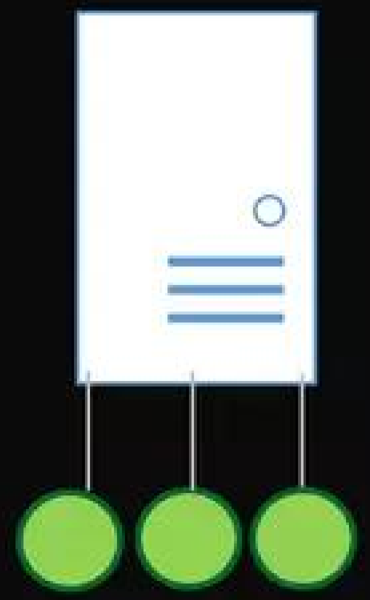
databricks

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

```
conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
```

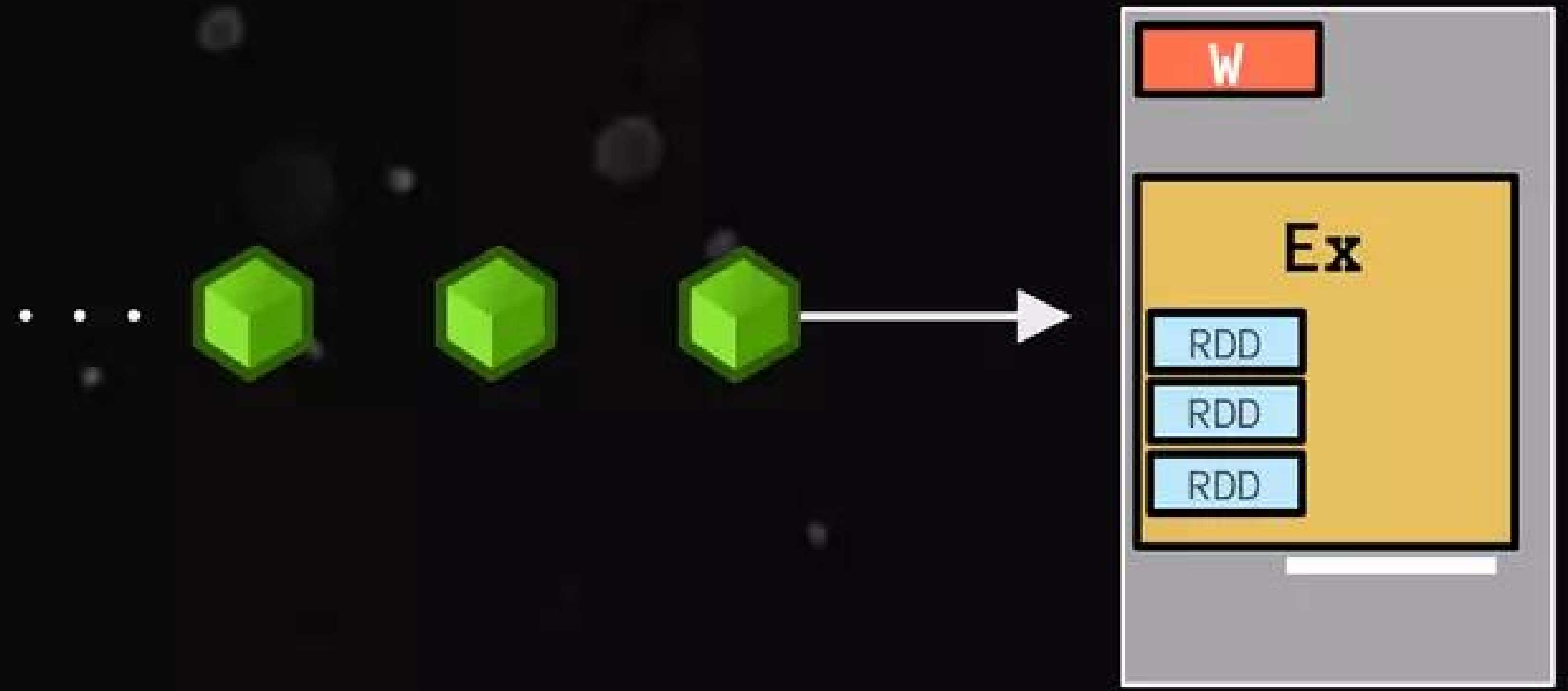
To register your own custom classes with Kryo, use the `registerKryoClasses` method:

```
val conf = new SparkConf().setMaster(...).setAppName(...)
conf.registerKryoClasses(Seq(classOf[MyClass1], classOf[MyClass2]))
val sc = new SparkContext(conf)
```

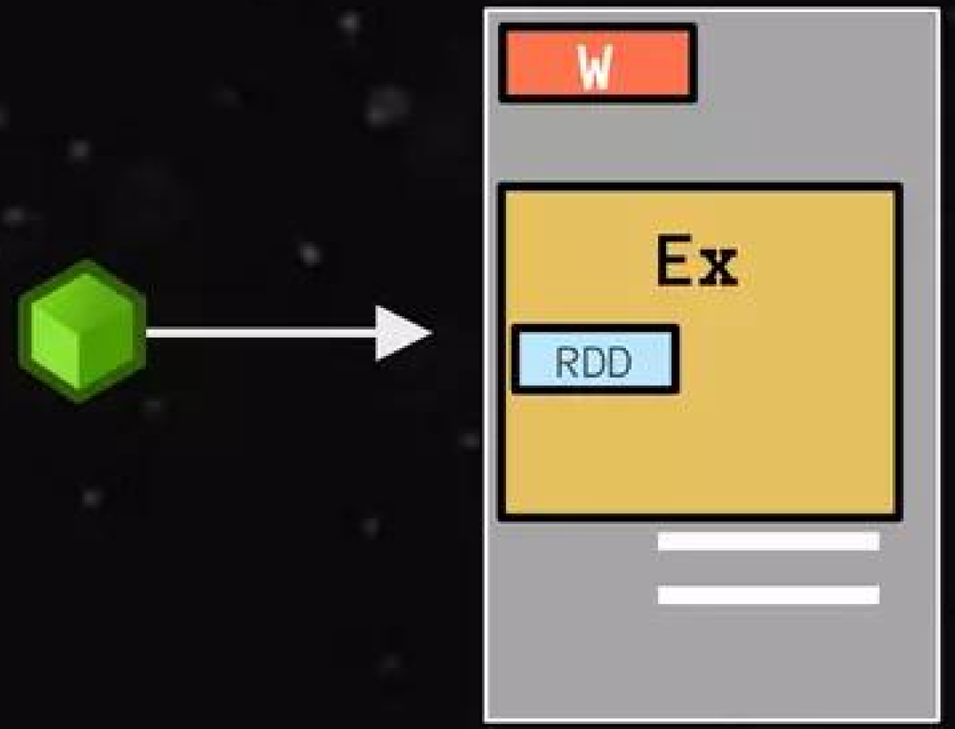
- If your objects are large, you may need to increase `spark.kryoserializer.buffer.mb` config property

- The default is 2, but this value needs to be large enough to hold the *largest* object you will serialize.

TUNING FOR Spark



High churn

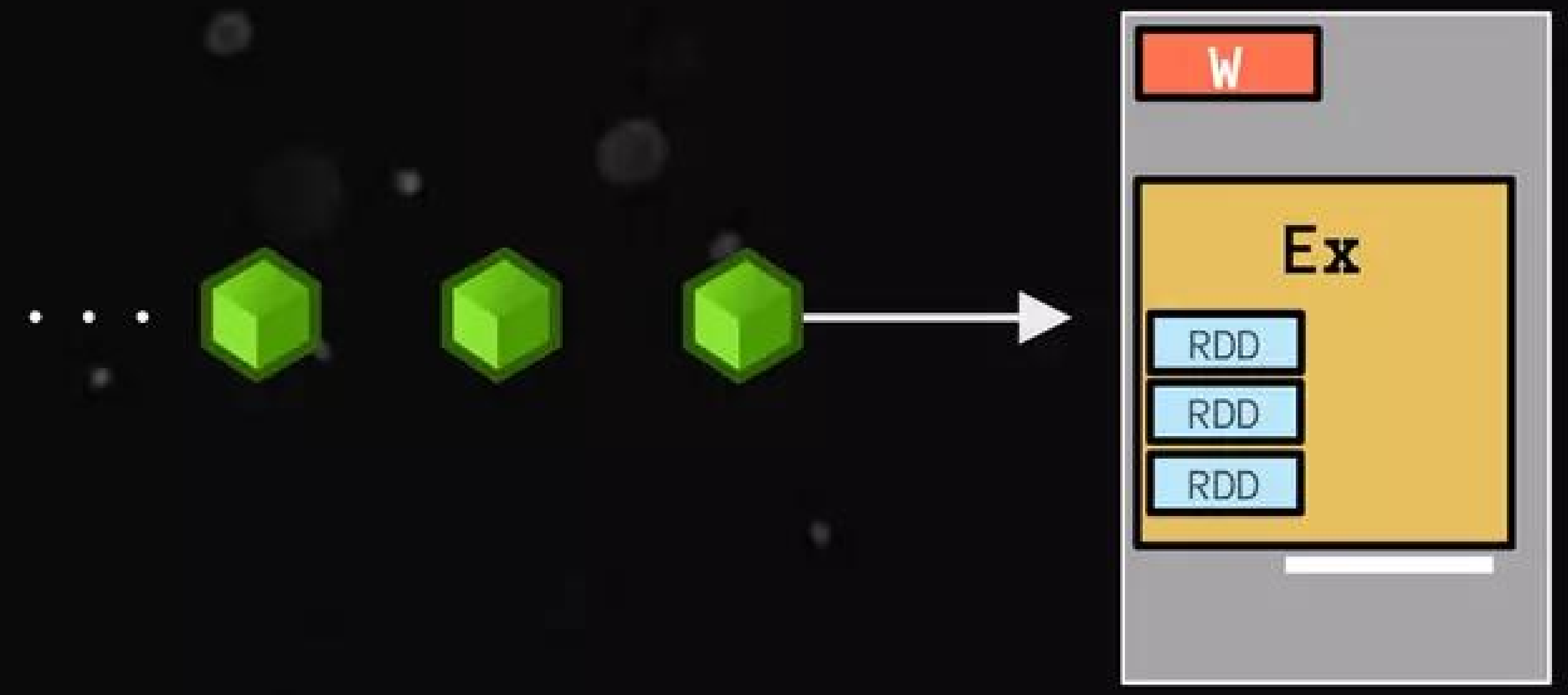


Low churn

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: 

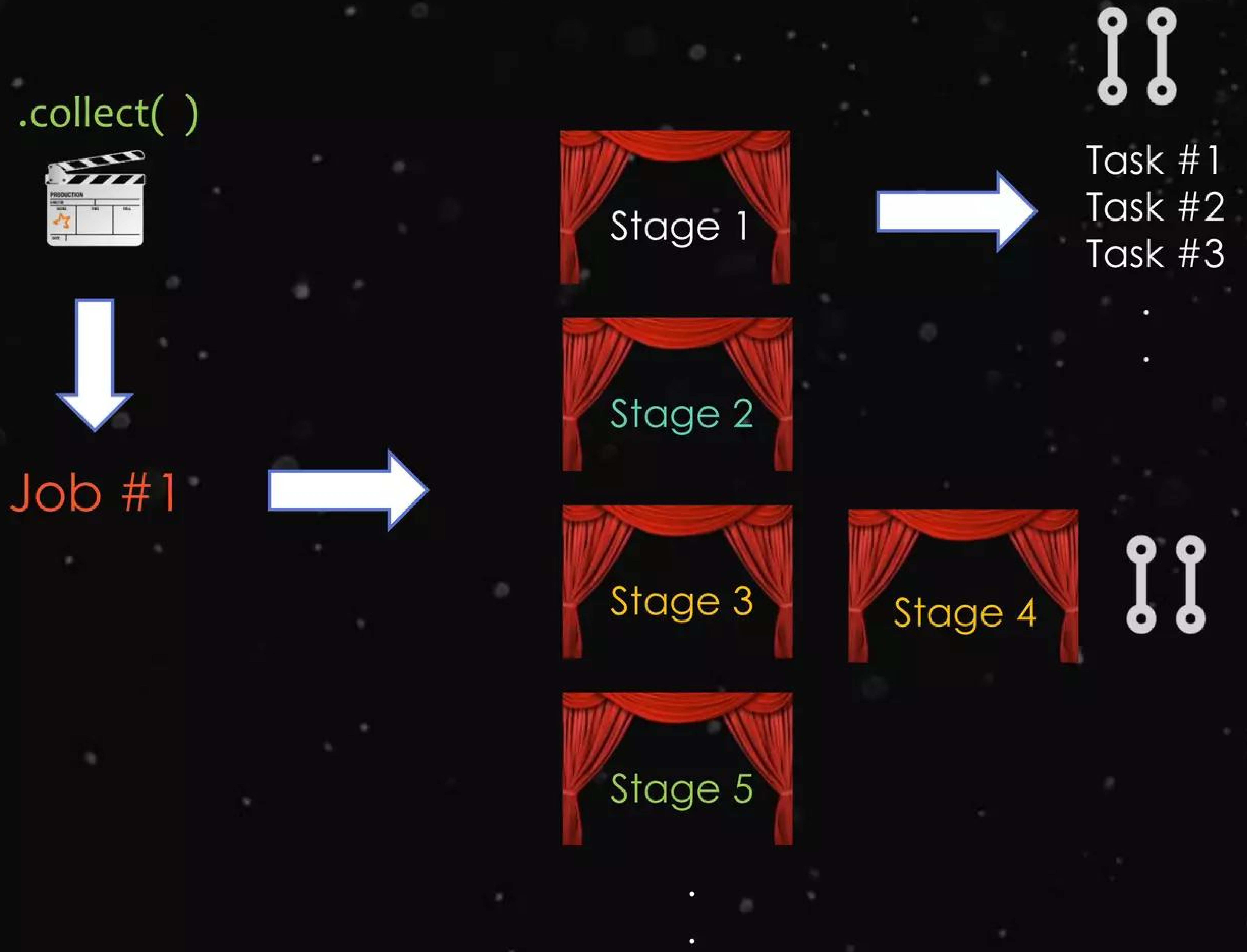
G1 GC

-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	Executor


```
Rdd1.join(rdd2)
    .groupBy(…)
    .filter(…)
```

RDD Objects

- Build operator DAG

DAG Scheduler

- Split graph into *stages* of tasks
- Submit each stage as ready

Task Scheduler

- Launches individual tasks
- Retry failed or straggling tasks

Executor

- Execute tasks
- Store and serve blocks

Agnostic to operators ← Stage failed — **Doesn't know about stages**

LINEAGE

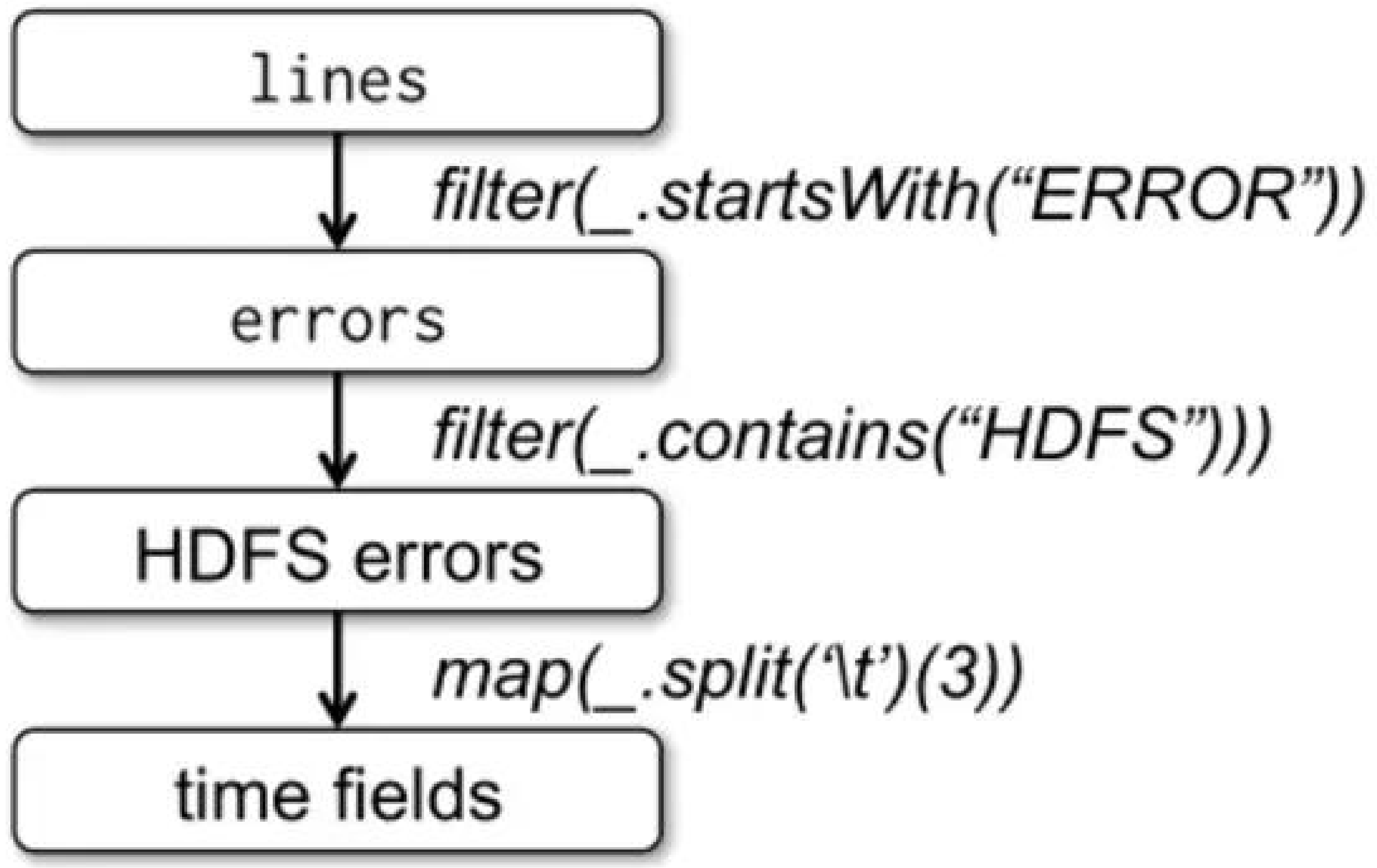


Figure 1: Lineage graph for the third query in our example. Boxes represent RDDs and arrows represent transformations.

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

LINEAGE


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

"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."

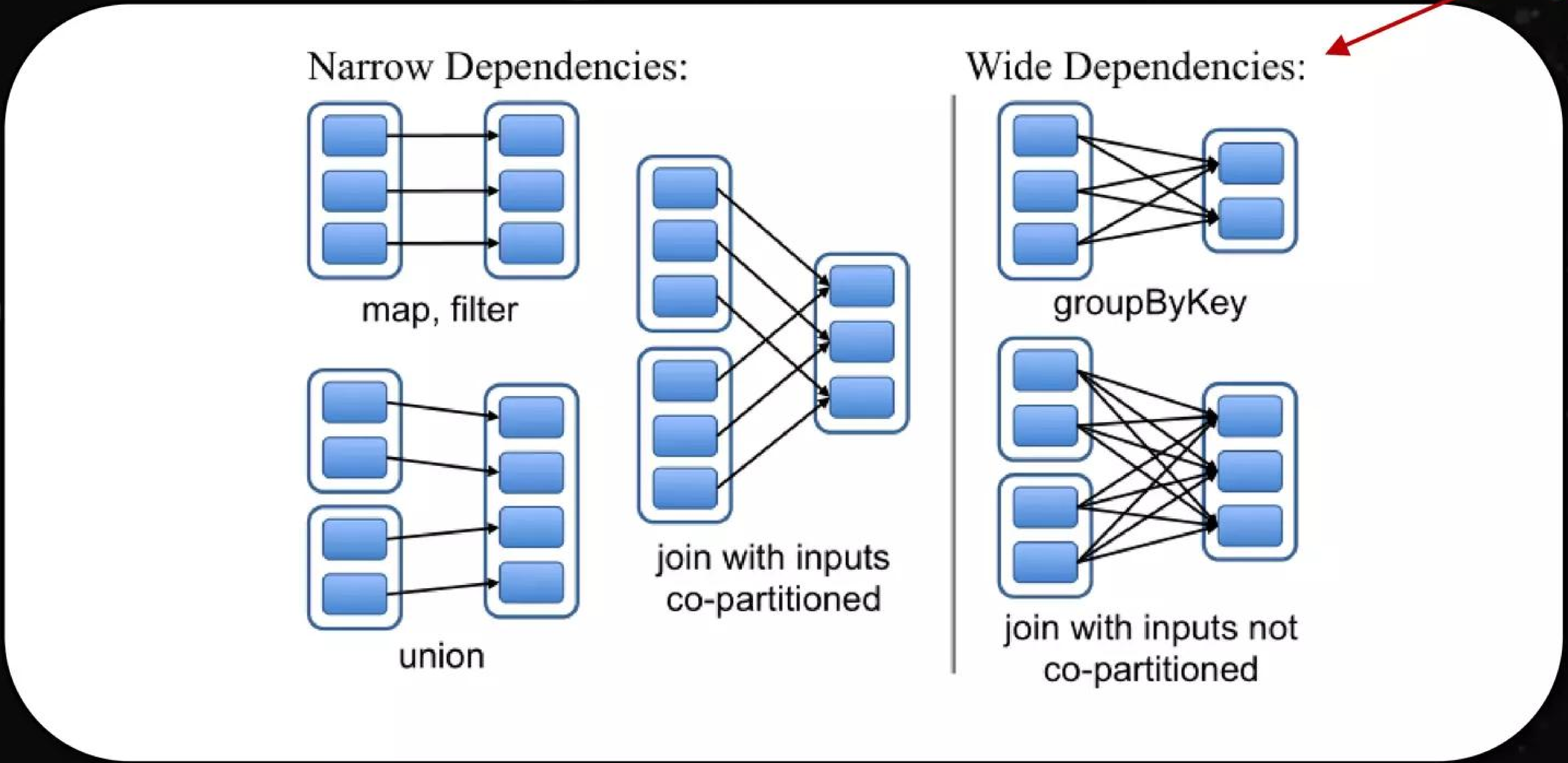
"The most interesting question in designing this interface is how to represent dependencies between RDDs."

"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

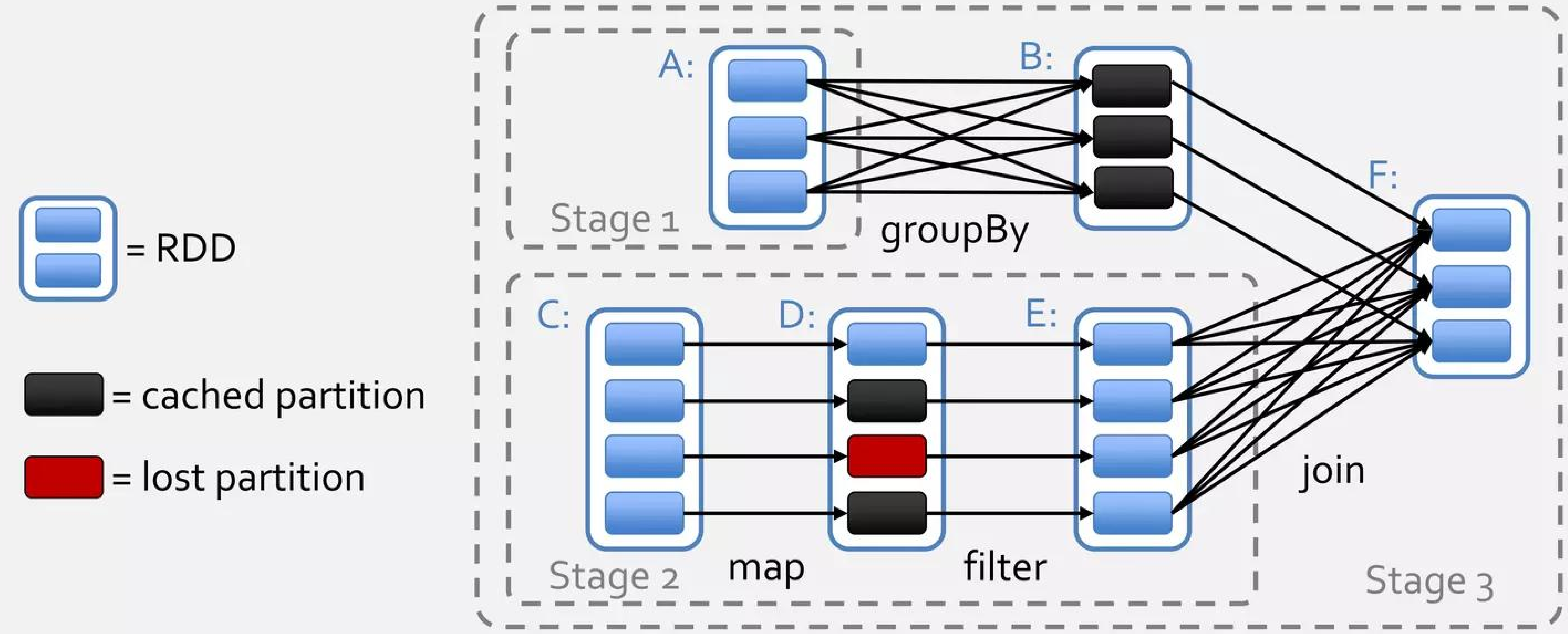
Requires shuffle



Examples of narrow and wide dependencies.

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

STAGES


= RDD
= cached partition
= lost partition
A:
B:
Stage 1
groupBy
F:
C:
D:
E:
join
Stage 2
map
filter
Stage 3

LINEAGE

Dependencies: Narrow vs Wide

"This distinction is useful for two reasons:

1) Narrow dependencies allow for pipelined execution on one cluster node, which can compute all the parent partitions. For example, one can apply a map followed by a filter on an element-by-element basis.

In contrast, wide dependencies require data from all parent partitions to be available and to be shuffled across the nodes using a MapReduce-like operation.

2) Recovery after a node failure is more efficient with a narrow dependency, as only the lost parent partitions need to be recomputed, and they can be recomputed in parallel on different nodes. In contrast, in a lineage graph with wide dependencies, a single failed node might cause the loss of some partition from all the ancestors of an RDD, requiring a complete re-execution."

To display the lineage of an RDD, Spark provides a **`toDebugString`** method:


```
scala> input.toDebugString

res85: String =
(2) data.text MappedRDD[292] at textFile at <console>:13
 | data.text HadoopRDD[291] at textFile at <console>:13

scala> counts.toDebugString
res84: String =
(2) ShuffledRDD[296] at reduceByKey at <console>:17
 +-(2) MappedRDD[295] at map at <console>:17
    | FilteredRDD[294] at filter at <console>:15
    | MappedRDD[293] at map at <console>:15
    | data.text MappedRDD[292] at textFile at <console>:13
    | data.text HadoopRDD[291] at textFile at <console>:13
```

How do you know if a shuffle will be called on a Transformation?

- repartition , join, cogroup, and any of the *By or *ByKey transformations can result in shuffles

- If you declare a numPartitions parameter, it'll probably shuffle

- If a transformation constructs a shuffledRDD, it'll probably shuffle

- combineByKey calls a shuffle (so do other transformations like groupByKey, which actually end up calling combineByKey)

Note that repartition just calls coalese w/ True:

RDD.scala

```
 def repartition(numPartitions: Int)(implicit
ord: Ordering[T] = null): RDD[T] = {
    coalesce(numPartitions, shuffle = true)
  }
```

How do you know if a shuffle will be called on a Transformation?

Transformations that use “numPartitions” like distinct will probably shuffle:

```
def distinct(numPartitions: Int)(implicit ord: Ordering[T] =
null): RDD[T] =
    map(x => (x, null)).reduceByKey((x, y) => x,
numPartitions).map(_._1)
```

PERSERVES PARTITIONING

- An extra parameter you can pass a k/v transformation to let Spark know that you will not be messing with the keys at all

- All operations that shuffle data over network will benefit from partitioning

- Operations that benefit from partitioning:

`cogroup, groupWith, join, leftOuterJoin, rightOuterJoin, groupByKey, reduceByKey, combineByKey, lookup, . . .`

https://github.com/apache/spark/blob/master/core/src/main/scala/org/apache/spark/rdd/RDD.scala#L302

```scala
/**
 * Return a new RDD containing only the elements that satisfy a predicate.
 */
def filter(f: T => Boolean): RDD[T] = {
  val cleanF = sc.clean(f)
  new MapPartitionsRDD[T, T](
    this,
    (context, pid, iter) => iter.filter(cleanF),
    preservesPartitioning = true)
}
```

Support Developers Contact Us Downloads

cloudera
Ask Bigger Questions

Search

COMMUNITY DOCUMENTATION DOWNLOADS TRAINING BLOGS

- Hadoop & Big Data
- Our Customers
- FAQs
- Blog
  - Accumulo (1)
  - Avro (16)
  - Bigtop (6)
  - Books (11)
  - Careers (14)
  - CDH (150)
  - Cloud (20)
  - Cloudera Labs (4)
  - Cloudera Life (6)
  - Cloudera Manager (73)
  - Community (208)

How-to: Tune Your Apache Spark Jobs (Part 1)

by Sandy Ryza | March 09, 2015 | no comments

Learn techniques for tuning your Apache Spark jobs for optimal efficiency.

(Editor's note: Sandy presents on **"Estimating Financial Risk with Spark"** at Spark Summit East on March 18.)

When you write Apache Spark code and page through the public APIs, you come across words like *transformation*, *action*, and *RDD*. Understanding Spark at this level is vital for writing Spark programs. Similarly, when things start to fail, or when you venture into the web UI to try to understand why your application is taking so long, you're confronted with a new vocabulary of words like *job*, *stage*, and *task*. Understanding Spark at this level is vital for writing *good* Spark programs, and of course by *good*, I mean *fast*. To write a Spark program that will execute efficiently, it is very, very helpful to understand Spark's underlying execution model.

In this post, you'll learn the basics of how Spark programs are actually executed on a cluster. Then, you'll get some practical recommendations about what Spark's execution model means for writing efficient programs.

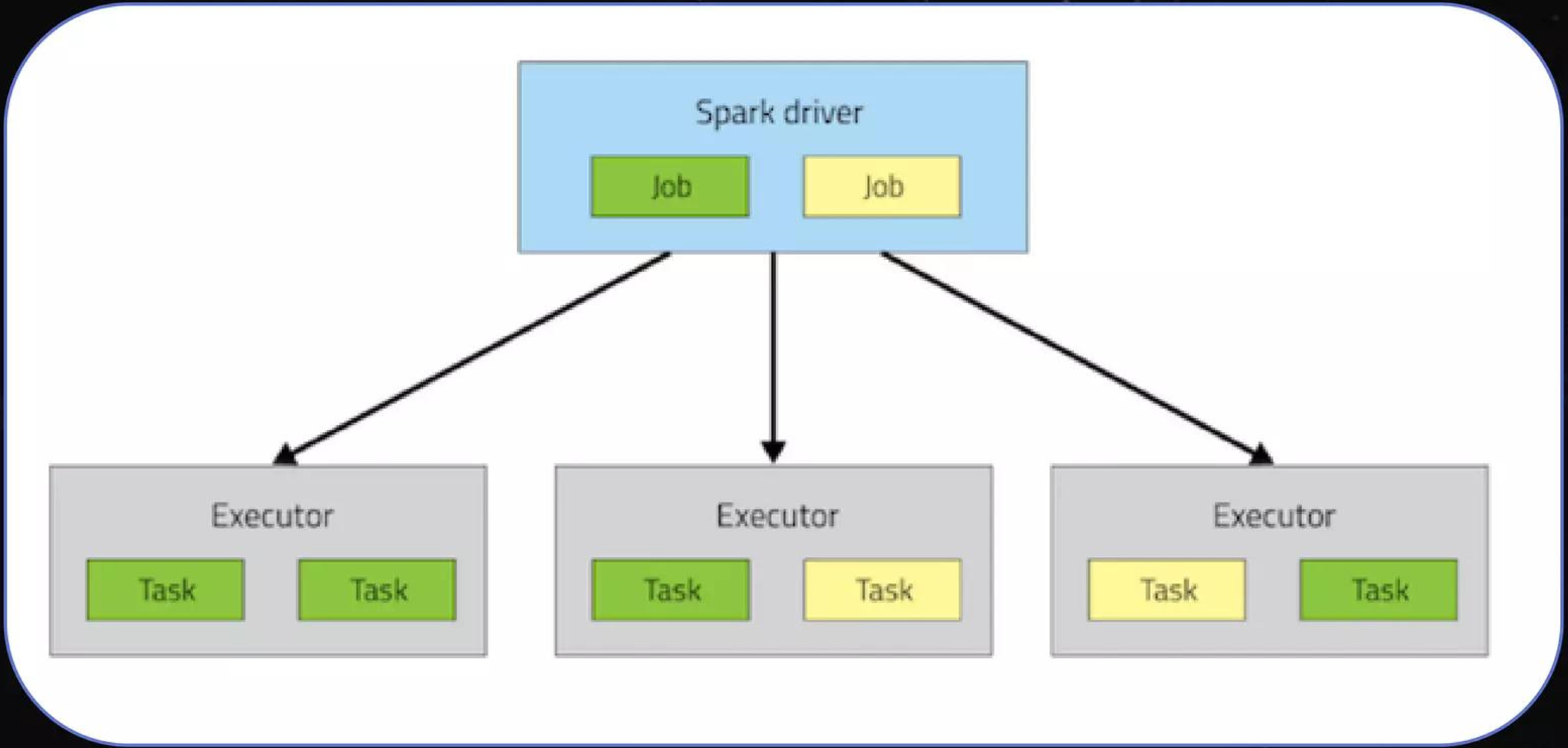
How Spark Executes Your Program

A Spark application consists of a single *driver* process and a set of *executor* processes scattered across nodes on the cluster.

The driver is the process that is in charge of the high-level control flow of work that needs to be done. The executor processes are responsible for executing this work, in the form of *tasks*, as well as for storing any data that the user chooses to cache. Both the driver and the executors typically stick around for the entire time the application is running, although **dynamic resource allocation** changes that for the latter. A single executor has a number of slots for running tasks, and will run many concurrently throughout its lifetime. Deploying these processes on the cluster is up to the cluster manager in use (YARN, Mesos, or Spark Standalone), but the driver and executor themselves exist in every Spark application.

Link

Source: Cloudera



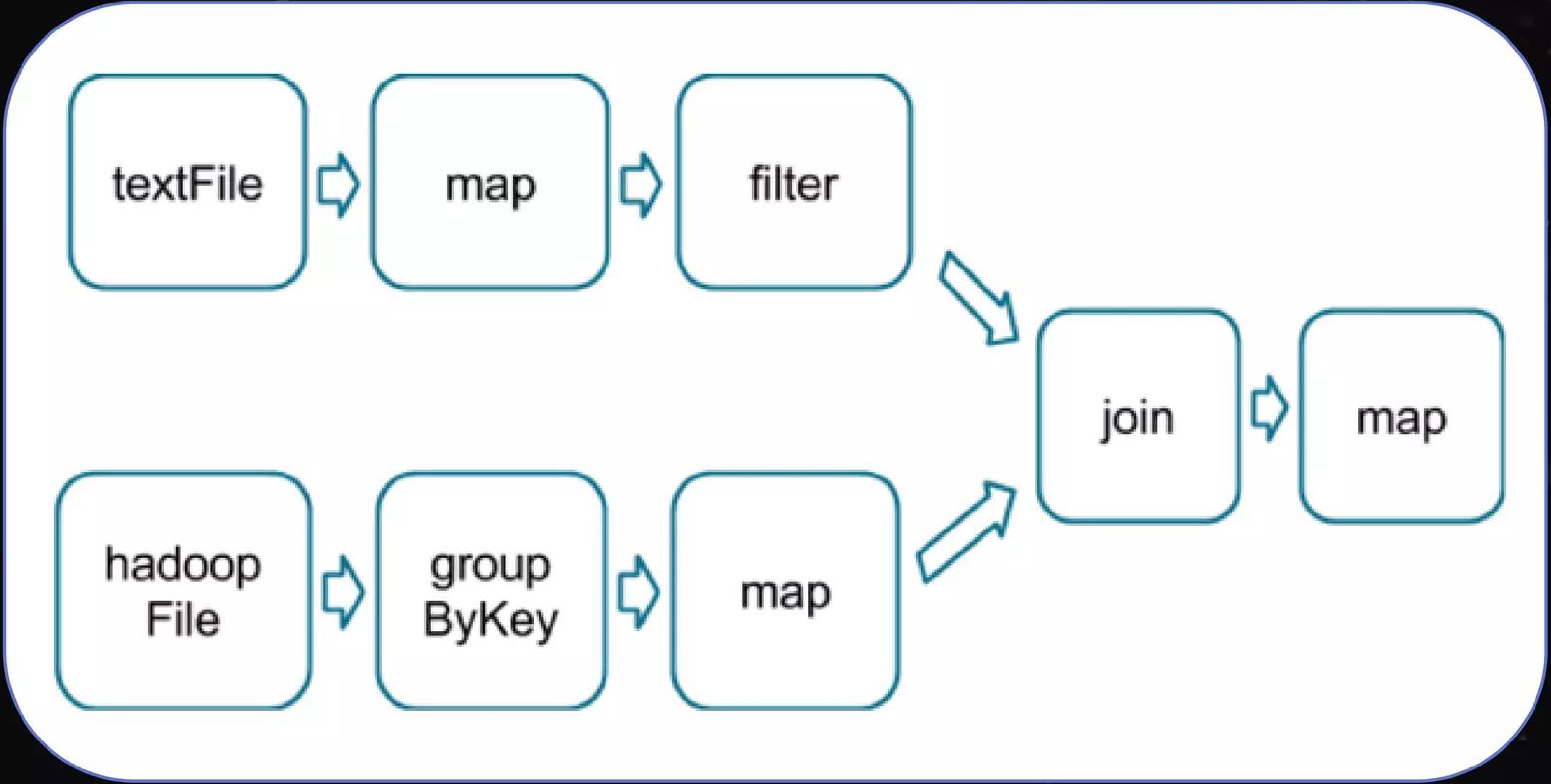
Source: Cloudera

How many Stages will this code require?

```
sc.textFile("someFile.txt").
  map(mapFunc).
  flatMap(flatMapFunc).
  filter(filterFunc).
  count()
```

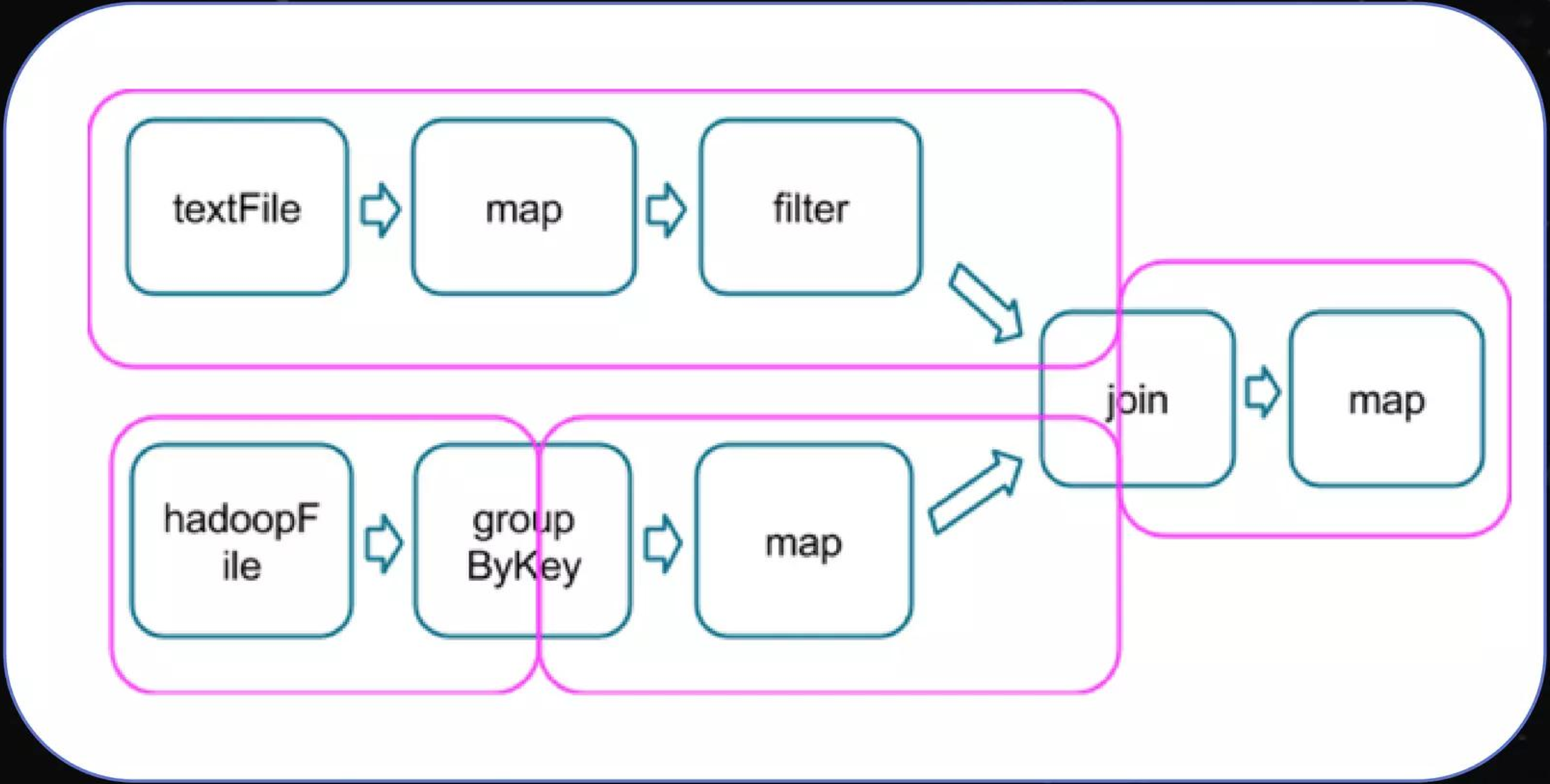
Source: Cloudera

How many Stages will this DAG require?



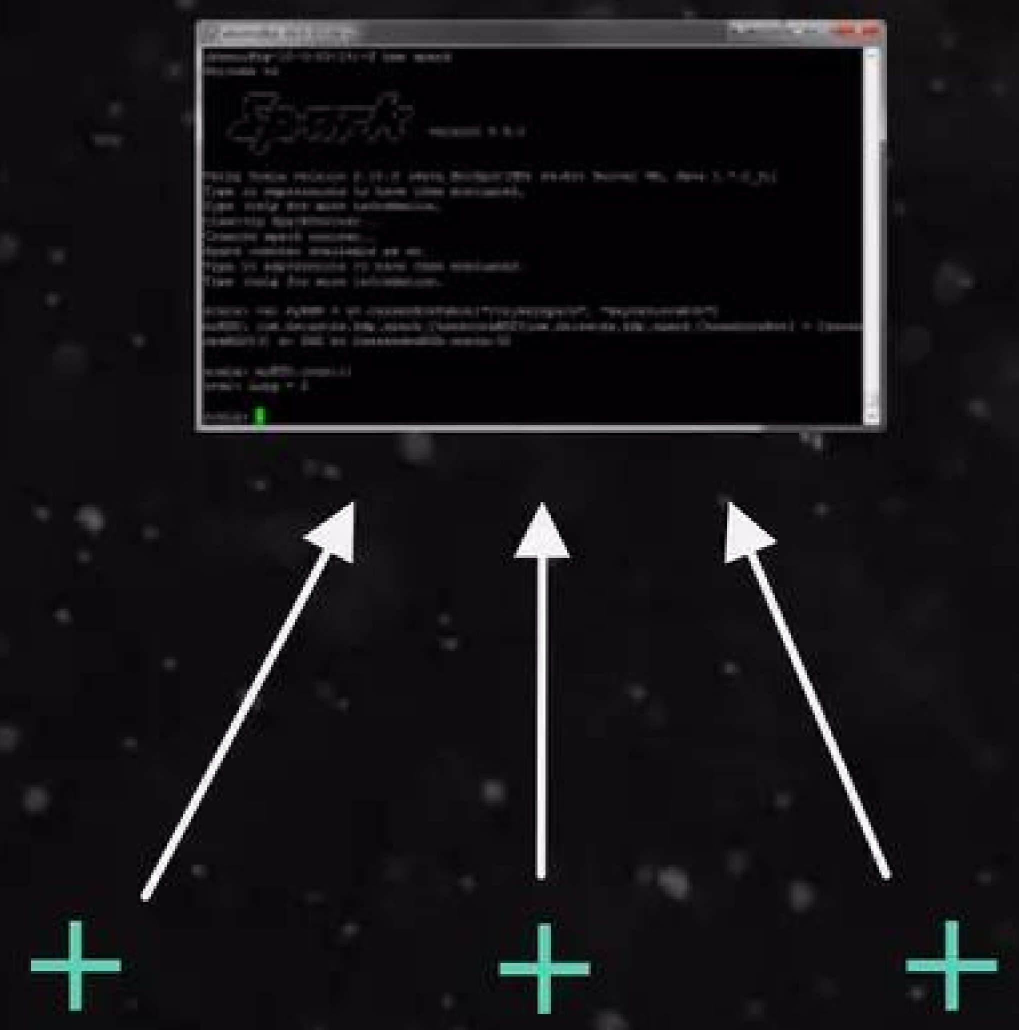
Source: Cloudera

How many Stages will this DAG require?



Source: Cloudera

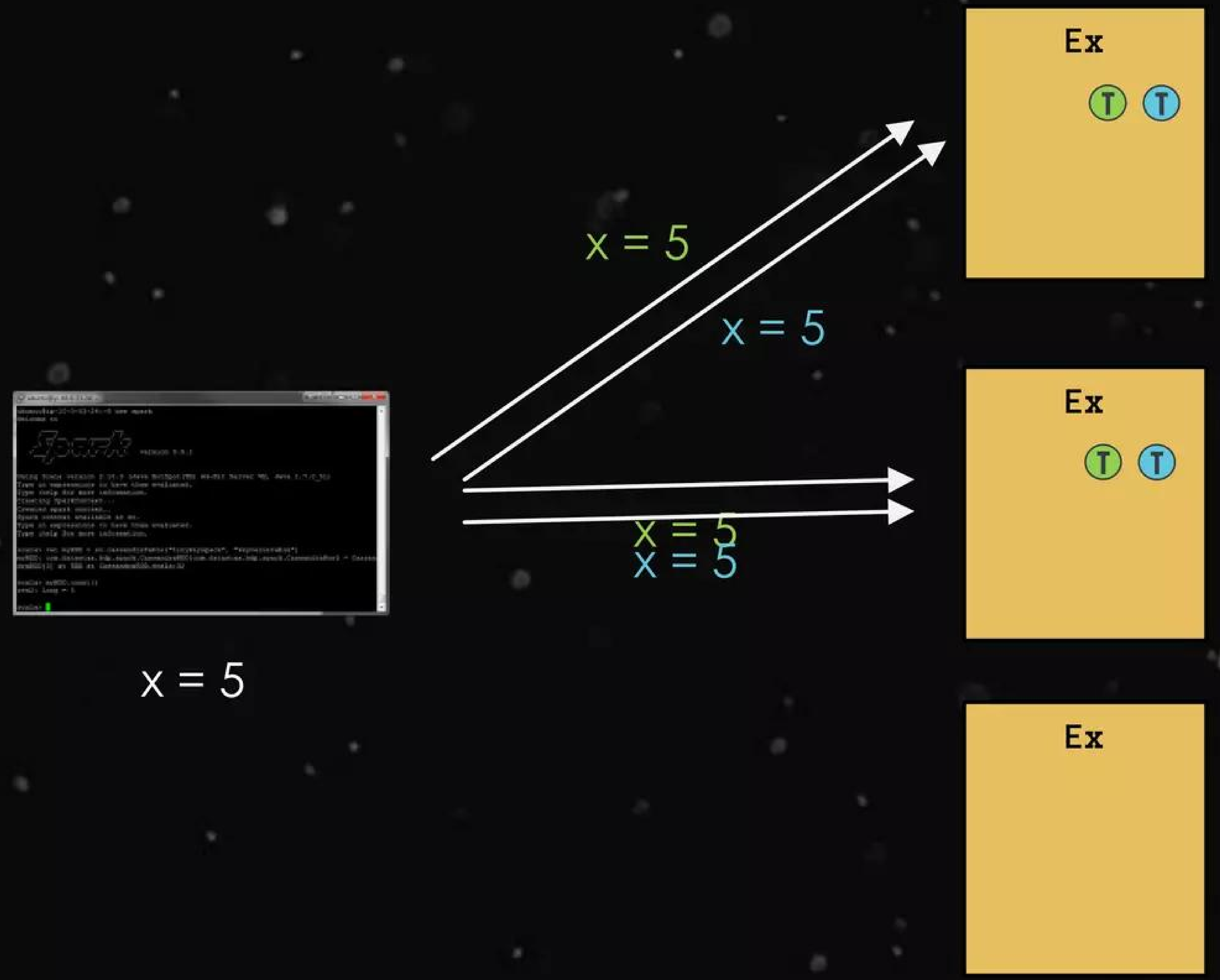
BROADCAST VARIABLES



&

ACCUMULATORS

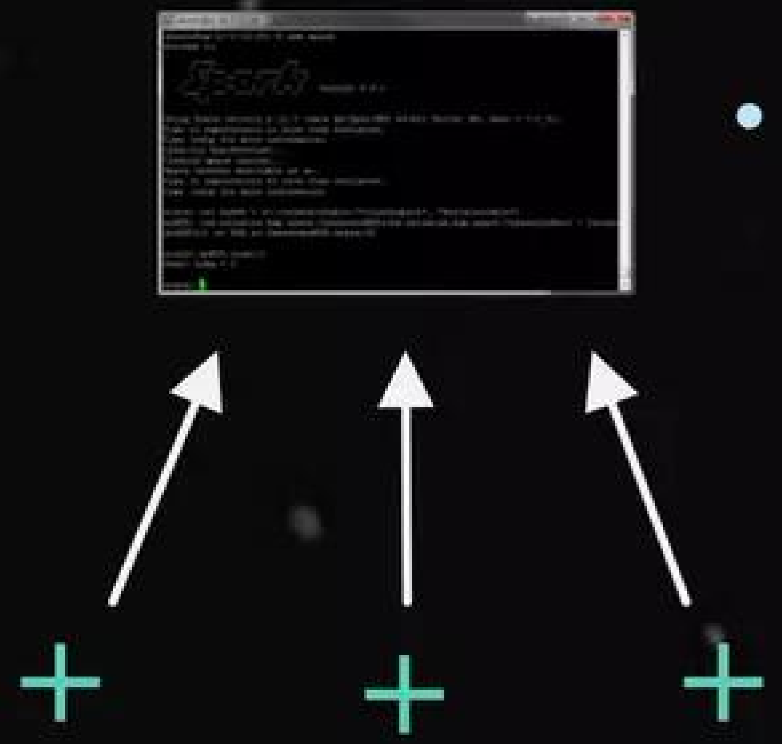
databricks



Ex
T T
Ex
T T
Ex
x = 5
x = 5
x = 5
x = 5
x = 5

USE CASES:

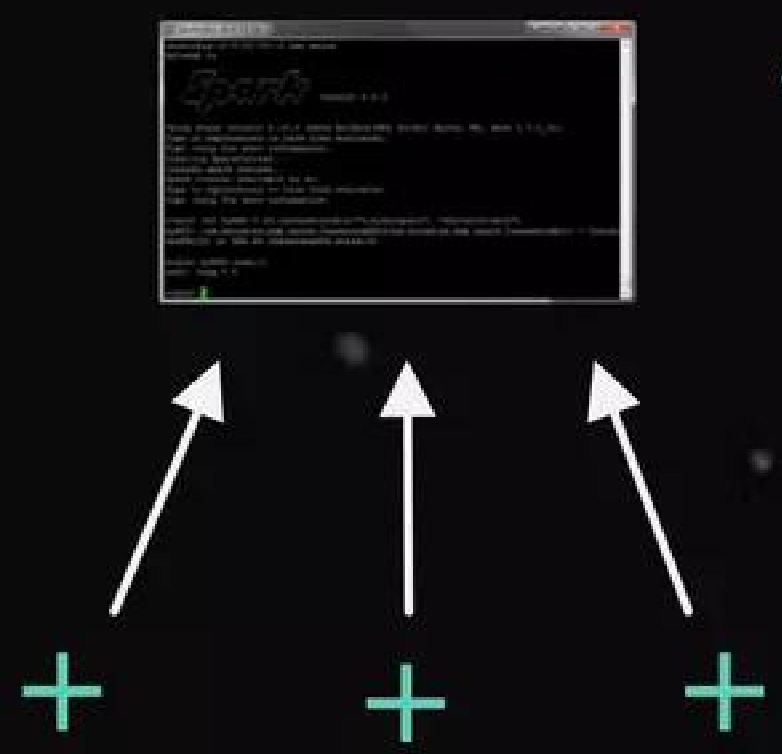
- Broadcast variables – Send a large read-only lookup table to all the nodes, or send a large feature vector in a ML algorithm to all nodes



- Accumulators – count events that occur during job execution for debugging purposes. Example: How many lines of the input file were blank? Or how many corrupt records were in the input dataset?

Spark supports 2 types of shared variables:

- Broadcast variables – allows your program to efficiently send a large, read-only value to all the worker nodes for use in one or more Spark operations. Like sending a large, read-only lookup table to all the nodes.



- Accumulators – allows you to aggregate values from worker nodes back to the driver program. Can be used to count the # of errors seen in an RDD of lines spread across 100s of nodes. Only the driver can access the value of an accumulator, tasks cannot. For tasks, accumulators are write-only.

BROADCAST VARIABLES

Broadcast variables let programmer keep a read-only variable cached on each machine rather than shipping a copy of it with tasks

For example, to give every node a copy of a large input dataset efficiently

Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost

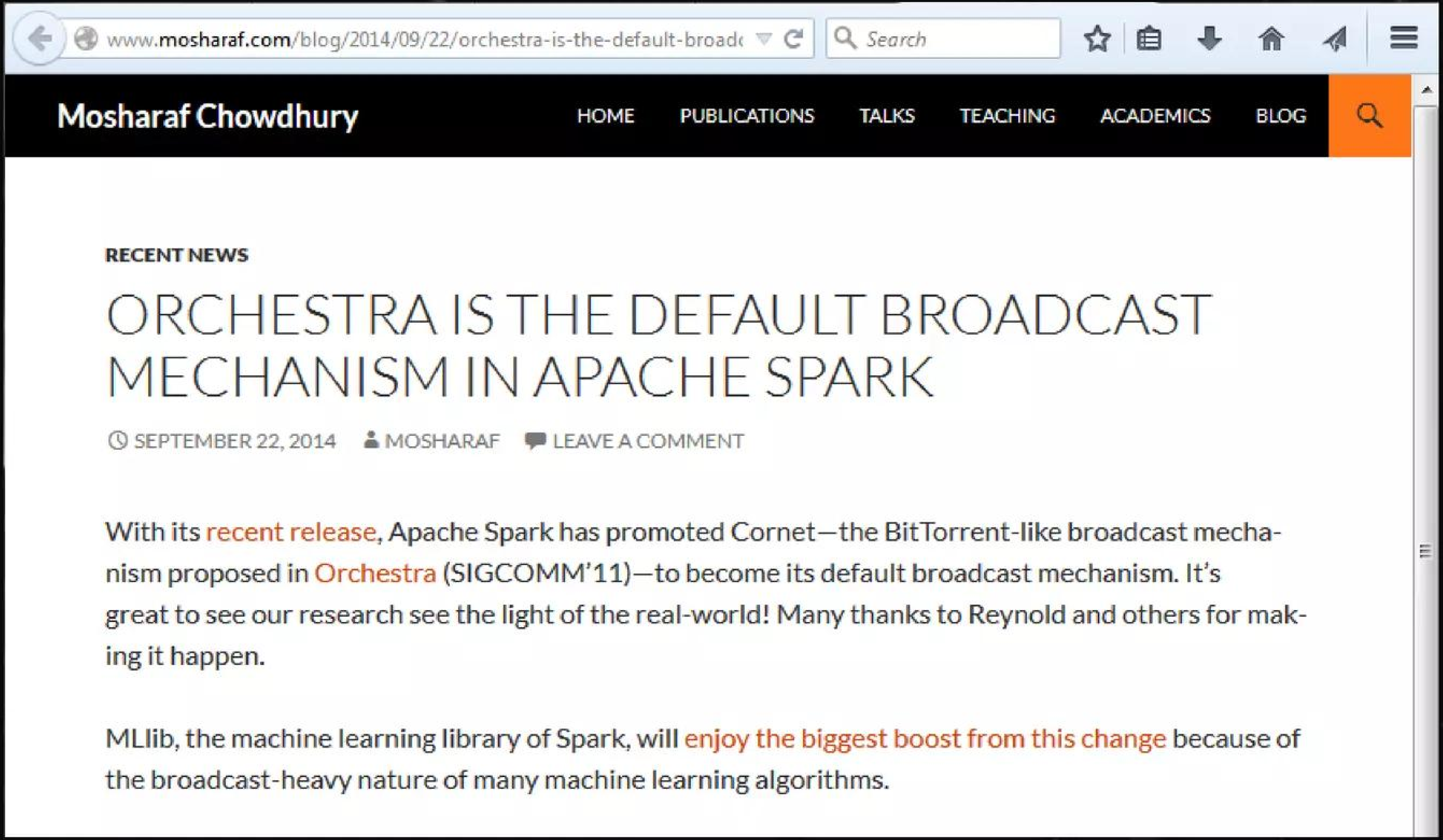
BROADCAST VARIABLES

Scala:

```
val broadcastVar = sc.broadcast(Array(1, 2, 3))
broadcastVar.value
```

Python:

```
broadcastVar = sc.broadcast(list(range(1, 4)))
broadcastVar.value
```



RECENT NEWS

ORCHESTRA IS THE DEFAULT BROADCAST MECHANISM IN APACHE SPARK

SEPTEMBER 22, 2014 MOSHARAF LEAVE A COMMENT

With its recent release, Apache Spark has promoted Cornet—the BitTorrent-like broadcast mechanism proposed in Orchestra (SIGCOMM'11)—to become its default broadcast mechanism. It's great to see our research see the light of the real-world! Many thanks to Reynold and others for making it happen.

MLlib, the machine learning library of Spark, will enjoy the biggest boost from this change because of the broadcast-heavy nature of many machine learning algorithms.

Link

Managing Data Transfers in Computer Clusters with Orchestra

Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I. Jordan, Ion Stoica
University of California, Berkeley
{mosharaf, matei, jtma, jordan, istoica}@cs.berkeley.edu

ABSTRACT

Cluster computing applications like MapReduce and Dryad transfer massive amounts of data between their computation stages. These transfers can have a significant impact on job performance, accounting for more than 50% of job completion times. Despite this impact, there has been relatively little work on optimizing the performance of these data transfers, with networking researchers traditionally focusing on per-flow traffic management. We address this limitation by proposing a global management architecture and a set of algorithms that (1) improve the transfer times of common communication patterns, such as broadcast and shuffle, and (2) allow scheduling policies at the transfer level, such as prioritizing a transfer over other transfers. Using a prototype implementation, we show that our solution improves broadcast completion times by up to 4.5× compared to the status quo in Hadoop. We also show that transfer-level scheduling can reduce the completion time of high-priority transfers by 1.7×.

Categories and Subject Descriptors

C.2 [**Computer-communication networks**]: Distributed systems—*Cloud computing*

General Terms

Algorithms, design, performance

Keywords

Data-intensive applications, data transfer, datacenter networks

1 Introduction

The last decade has seen a rapid growth of cluster computing frameworks to analyze the increasing amounts of data collected and generated by web services like Google, Facebook and Yahoo!. These clusters, operators aim to maximize the cluster utilization, while accommodating a variety of applications, workloads, and user requirements. To achieve these goals, several solutions have recently been proposed to reduce job completion times [11, 29, 43], accommodate interactive workloads [29, 43], and increase utilization [26, 29]. While in large part successful, these solutions have so far been focusing on scheduling and managing computation and storage resources, while mostly ignoring network resources.

However, managing and optimizing network activity is critical for improving job performance. Indeed, Hadoop traces from Facebook show that, on average, transferring data between successive stages accounts for 33% of the running times of jobs with reduce phases. Existing proposals for full bisection bandwidth networks [21, 23, 24, 35] along with flow-level scheduling [10, 21] can improve network performance, but they do not account for collective behaviors of flows due to the lack of job-level semantics.

In this paper, we argue that to maximize job performance, we need to optimize at the level of transfers, instead of individual flows. We define a *transfer* as the set of all flows transporting data between two stages of a job. In frameworks like MapReduce and Dryad, a stage cannot complete (or sometimes even start) before it receives all the data from the previous stage. Thus, the job running time depends on the time it takes to complete the *entire* transfer, rather than the duration of individual flows comprising it. To this end, we focus on two transfer patterns that occur in virtually all cluster computing frameworks and are responsible for most of the network traffic in these clusters: *shuffle* and *broadcast*. Shuffle captures the many-to-many communication pattern between the map and reduce stages in MapReduce, and between Dryad's stages. Broadcast captures the one-to-many communication pattern employed by iterative optimization algorithms [45] as well as fragment-replicate joins in Hadoop [6].

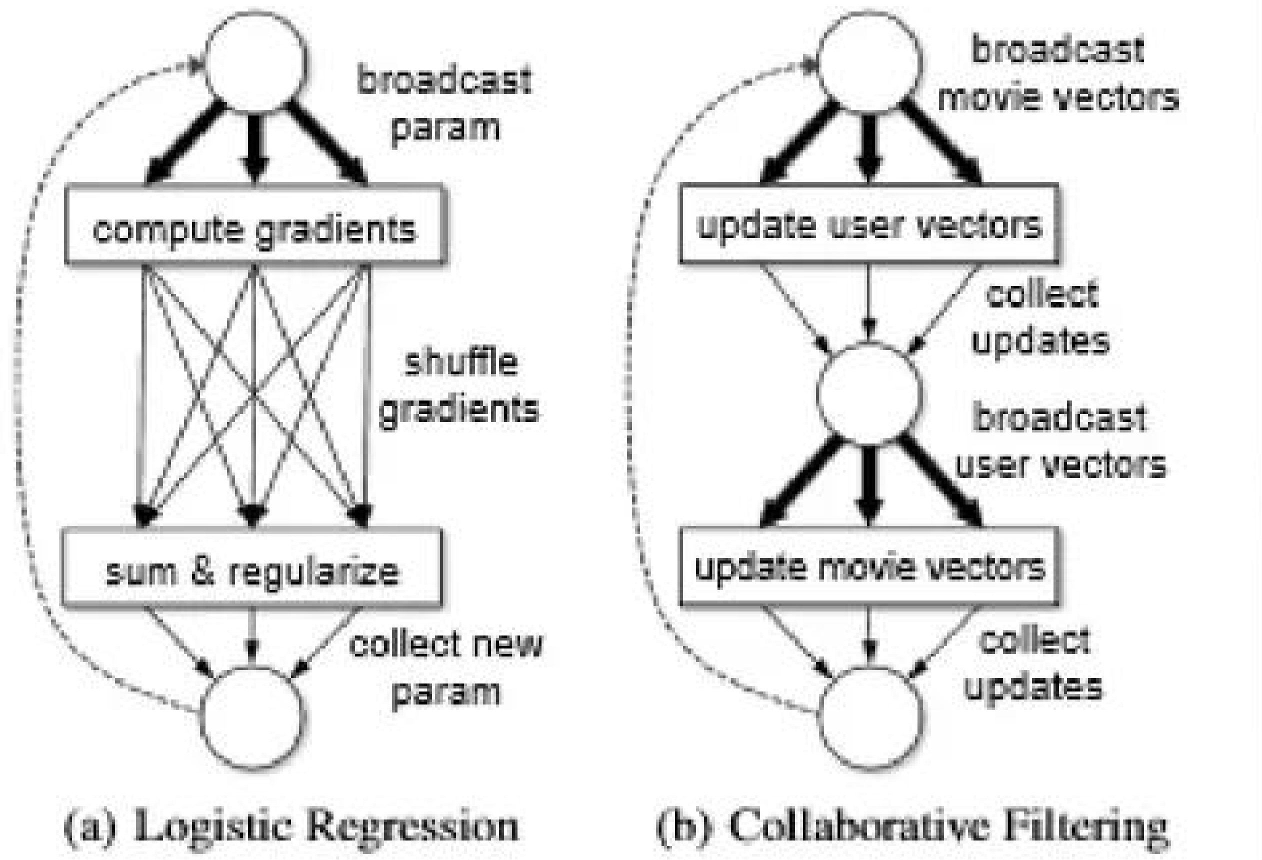


Figure 2: Per-iteration work flow diagrams for our motivating machine learning applications. The circle represents the master node and the boxes represent the set of worker nodes.

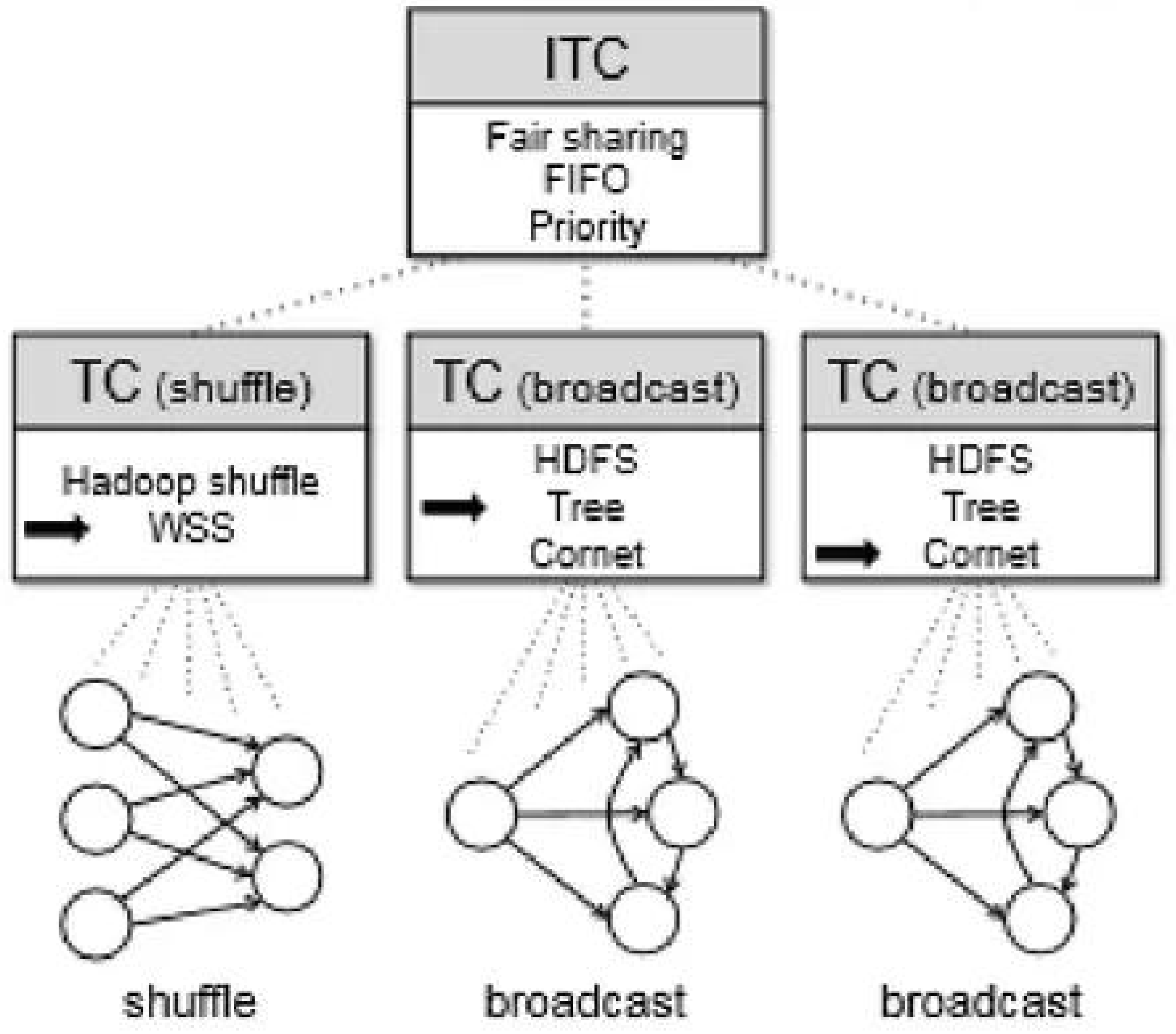
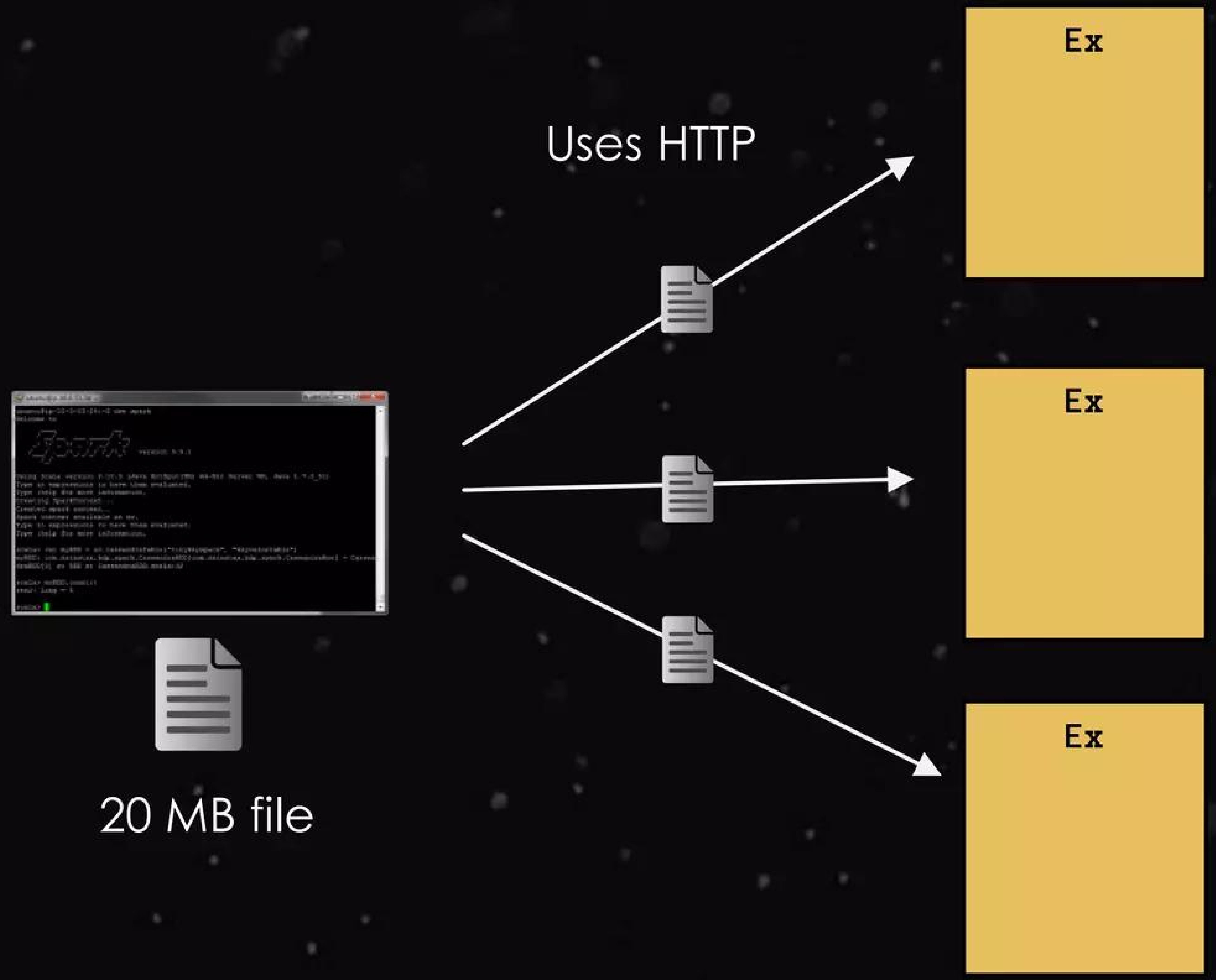
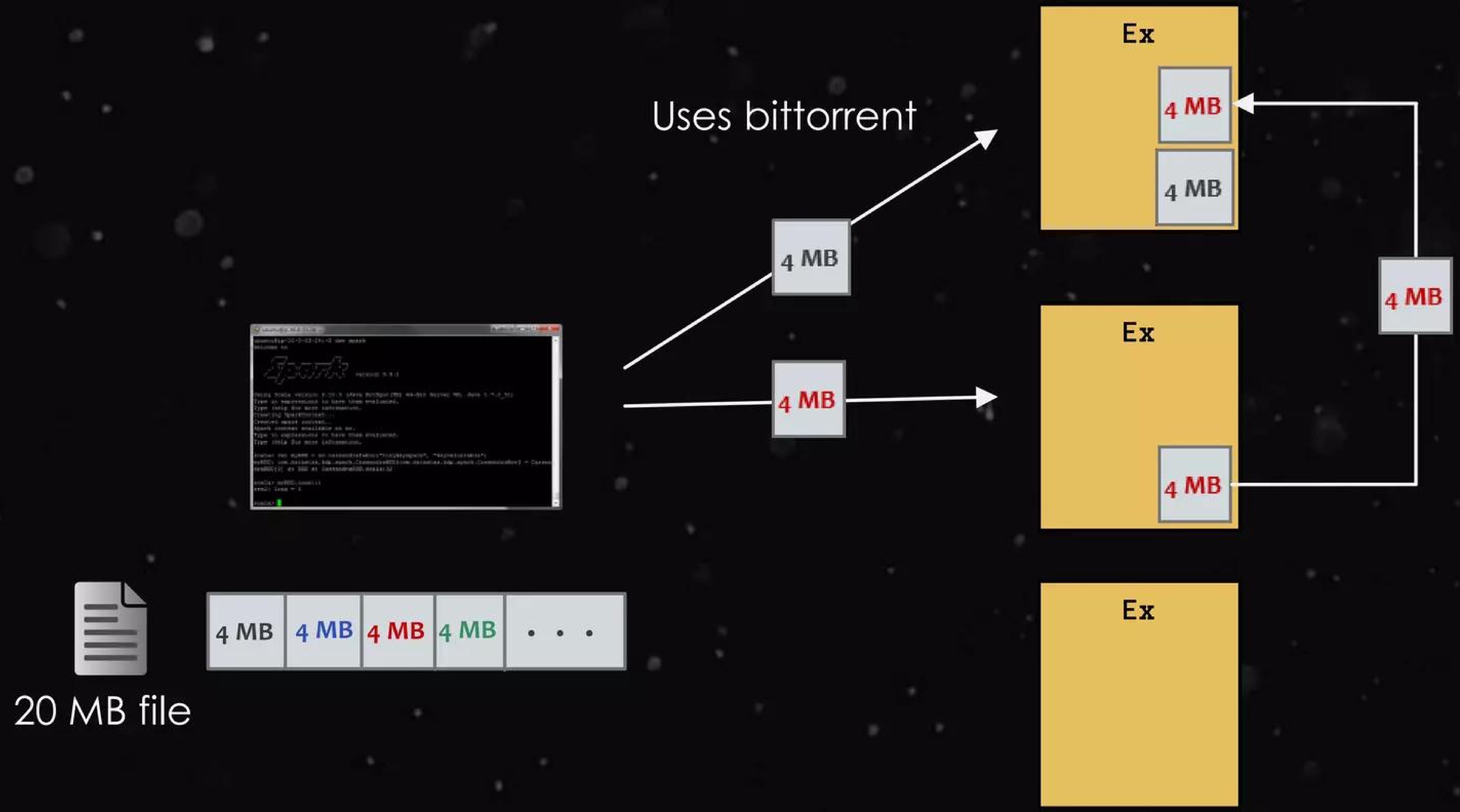


Figure 4: Orchestra architecture. An Inter-Transfer Controller (ITC) manages Transfer Controllers (TCs) for the active transfers. Each TC can choose among multiple transfer mechanisms depending on data size, number of nodes, and other factors. The ITC performs inter-transfer scheduling.

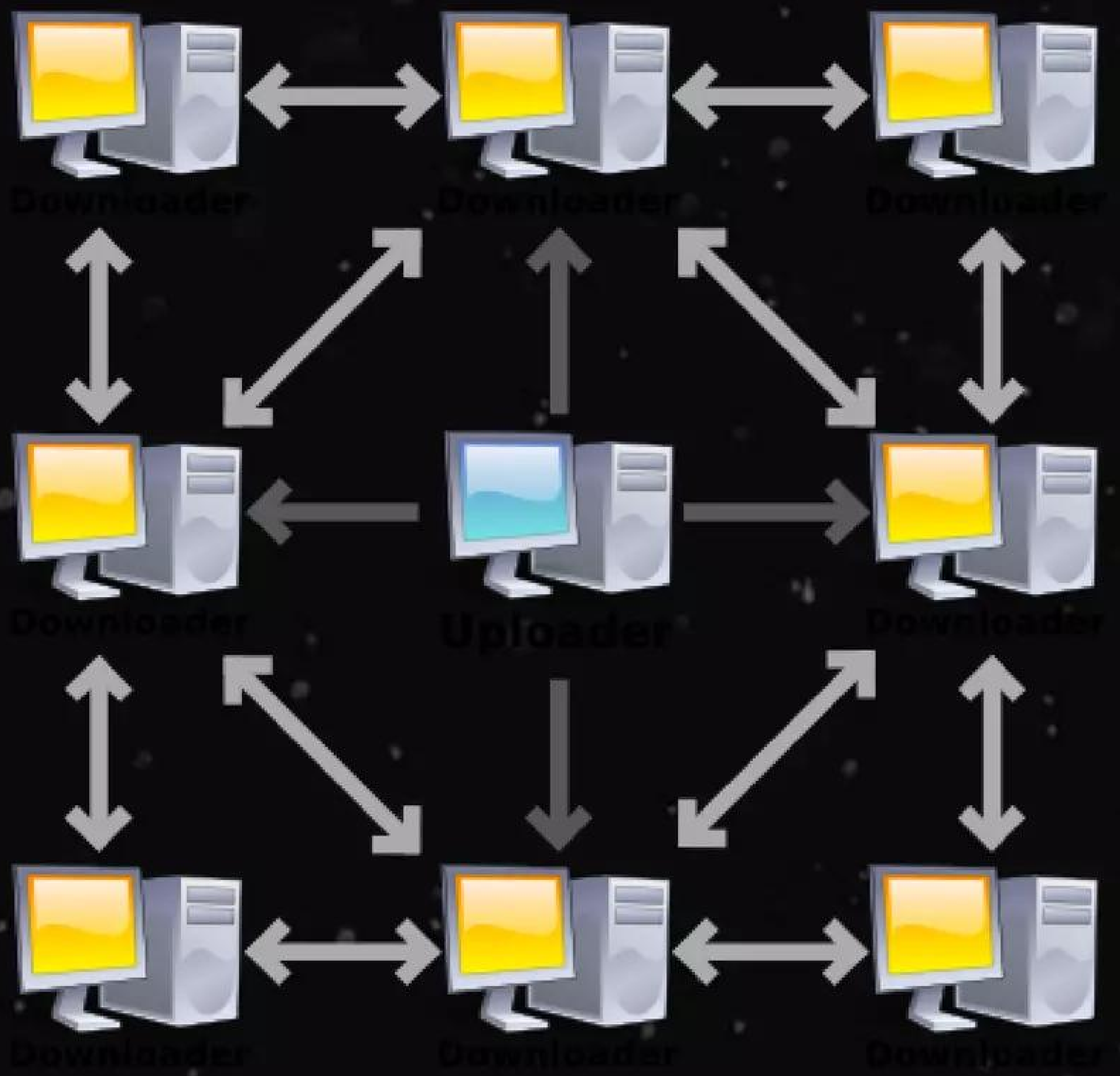
History: OLD TECHNIQUE FOR BROADCAST



BITTORENT TECHNIQUE FOR BROADCAST

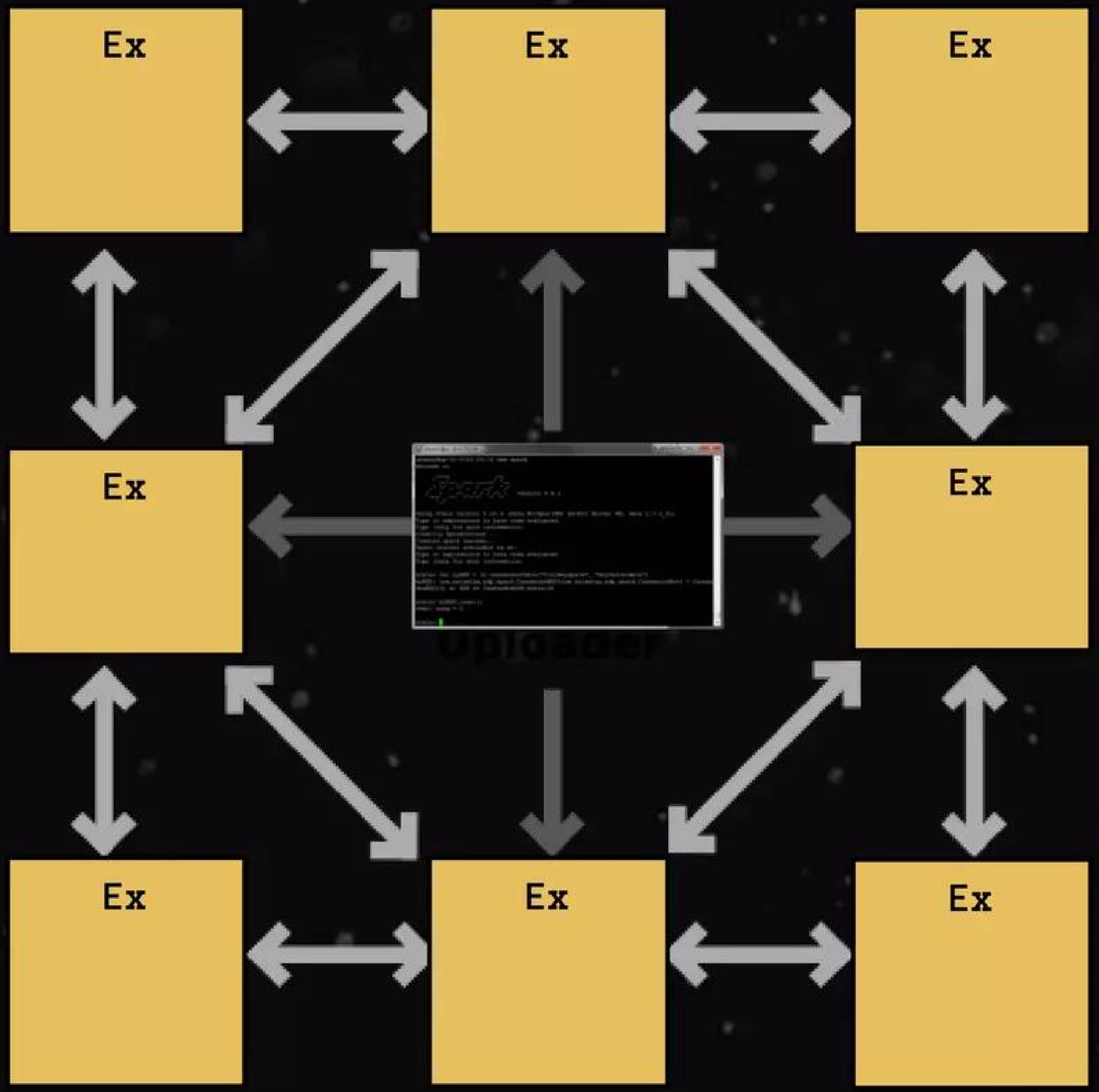


BITTORENT TECHNIQUE FOR BROADCAST



Source: Scott Martin

BITTORENT TECHNIQUE FOR BROADCAST



ACCUMULATORS

Accumulators are variables that can only be "added" to through an *associative* operation

Used to implement counters and sums, efficiently in parallel

Spark natively supports accumulators of numeric value types and standard mutable collections, and programmers can extend for new types

Only the driver program can read an accumulator's value, not the tasks

ACCUMULATORS

Scala:

```scala
val accum = sc.accumulator(0)

sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum += x)

accum.value
```

Python:

```python
accum = sc.accumulator(0)
rdd = sc.parallelize([1, 2, 3, 4])
def f(x):
   global accum
   accum += x

rdd.foreach(f)

accum.value
```

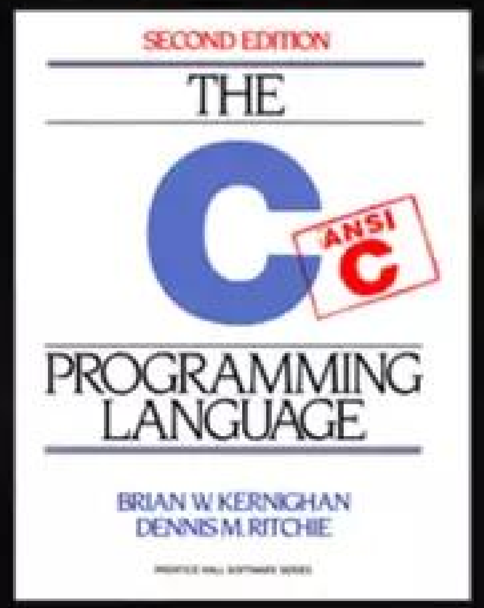
SCALA / PYTHON / JAVA / R

databricks

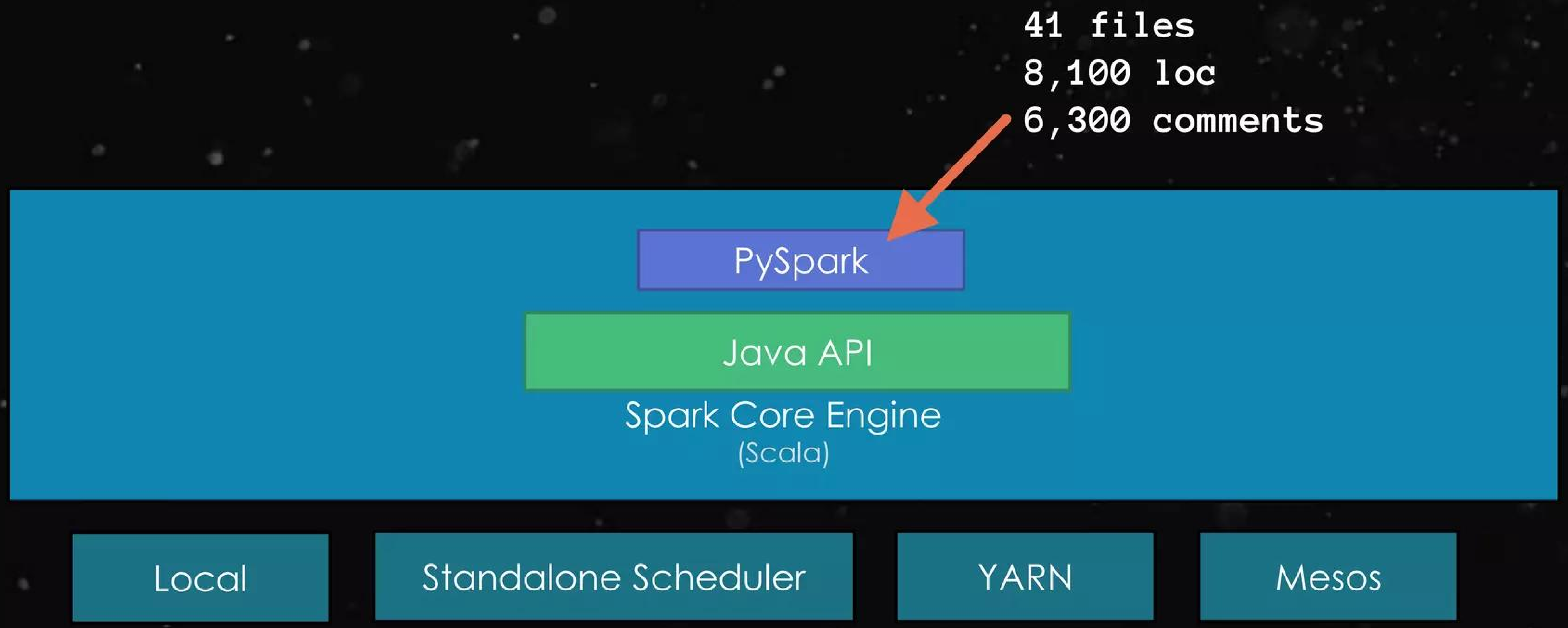
PySpark at a Glance

Write Spark jobs in Python

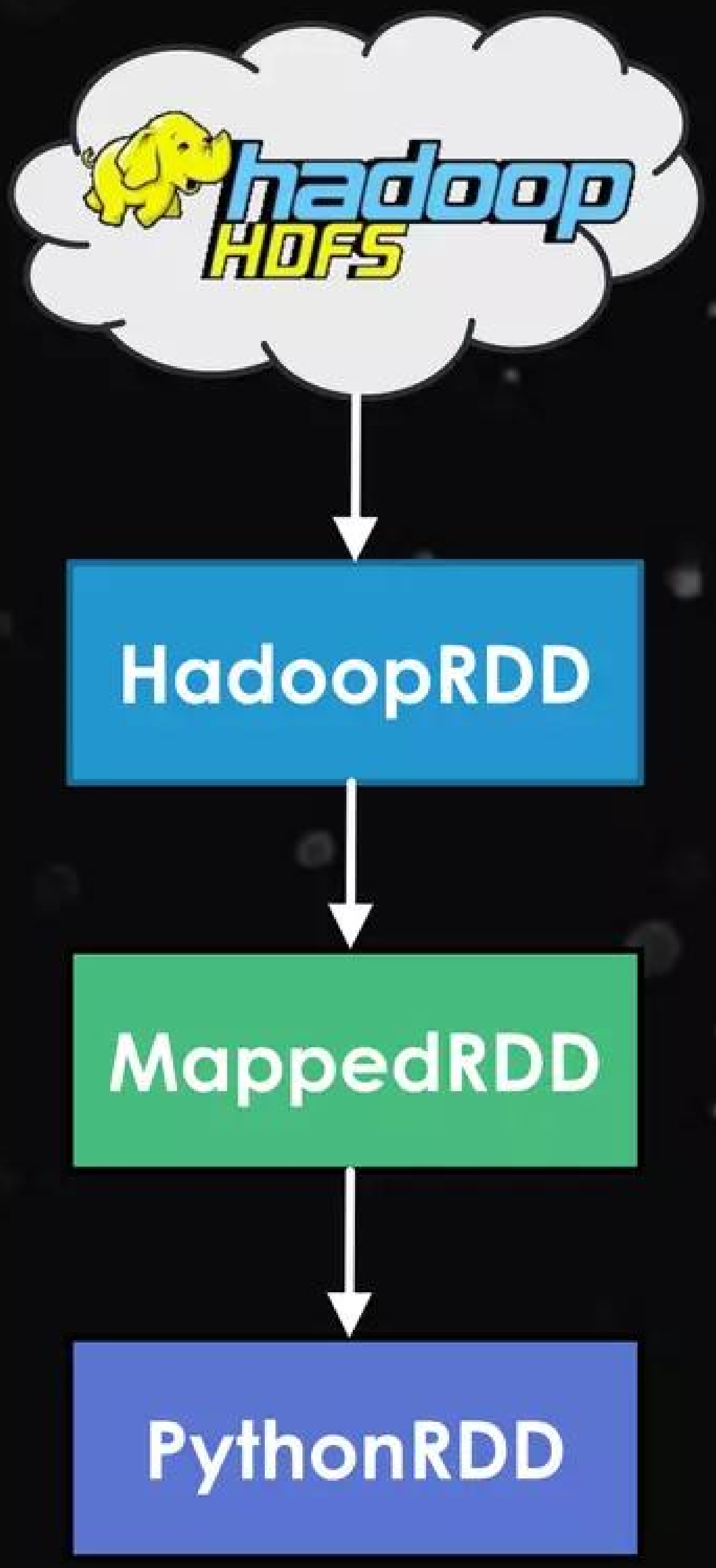
Run interactive jobs in the shell



Supports C extensions


41 files
8,100 loc
6,300 comments
PySpark
Java API
Spark Core Engine
(Scala)
Local
Standalone Scheduler
YARN
Mesos

PYSPARK ARCHITECTURE

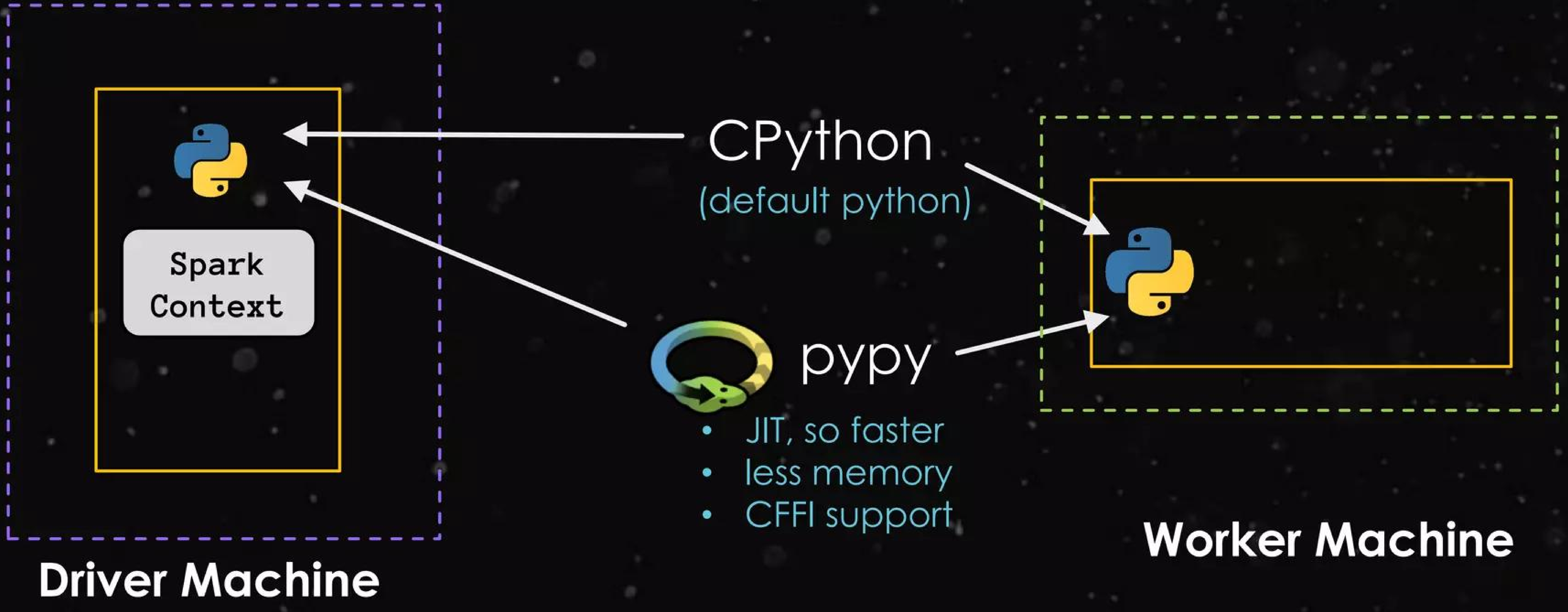


Data is stored as Pickled objects in an RDD[Array[Byte]]

RDD[Array[ , , ,]]

(100 KB – 1MB each picked object)

Choose Your Python Implementation



```
$ PYSPARK_DRIVER_PYTHON=pypy PYSPARK_PYTHON=pypy ./bin/pyspark
```

OR

```
$ PYSPARK_DRIVER_PYTHON=pypy PYSPARK_PYTHON=pypy ./bin/spark-submit wordcount.py
```

The performance speed up will depend on work load (from 20% to 3000%).

Here are some benchmarks:

Job	CPython 2.7	PyPy 2.3.1	Speed up
Word Count	41 s	15 s	2.7 x
Sort	46 s	44 s	1.05 x
Stats	174 s	3.6 s	48 x

Here is the code used for benchmark:

```
rdd = sc.textFile("text")
def wordcount():
    rdd.flatMap(lambda x:x.split('/'))\
        .map(lambda x:(x,1)).reduceByKey(lambda x,y:x+y).collectAsMap()
def sort():
    rdd.sortBy(lambda x:x, 1).count()
def stats():
    sc.parallelize(range(1024), 20).flatMap(lambda x: xrange(5024)).stats()
```

https://github.com/apache/spark/pull/2144

`spark.python.worker.memory`	512m	Amount of memory to use per python worker process during aggregation, in the same format as JVM memory strings (e.g. `512m`, `2g`). If the memory used during aggregation goes above this amount, it will spill the data into disks.

NEXT GEN SHUFFLE

databricks

100TB Daytona Sort Competition 2014

databricks

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

Spark sorted the same data **3X faster** using **10X fewer machines** than Hadoop MR in 2013.

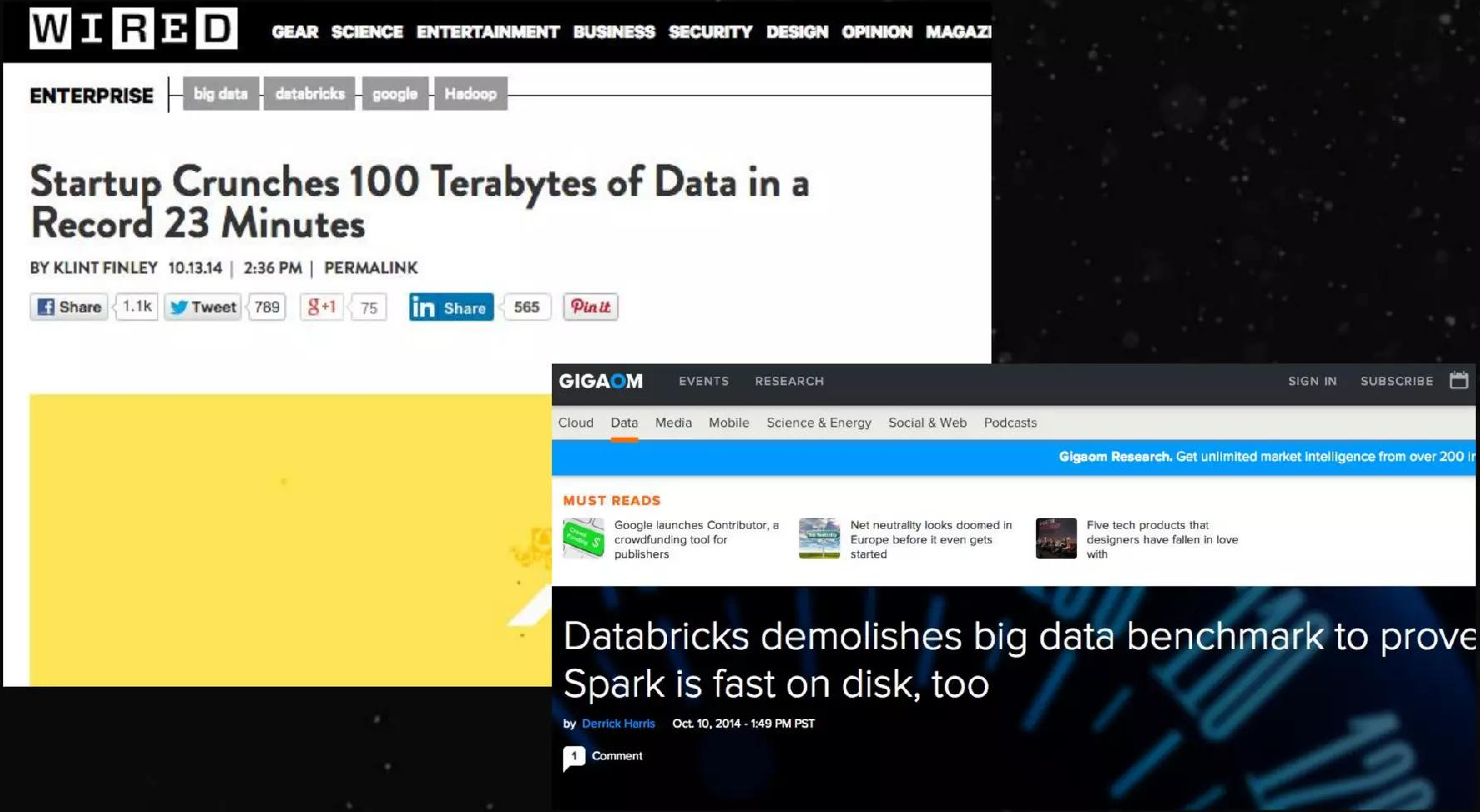
All the sorting took place on disk (HDFS) without using Spark's in-memory cache!

More info:

http://sortbenchmark.org

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

Work by Databricks engineers: Reynold Xin, Parviz Deyhim, Xiangrui Meng, Ali Ghodsi, Matei Zaharia



WIRED GEAR SCIENCE ENTERTAINMENT BUSINESS SECURITY DESIGN OPINION MAGAZI

ENTERPRISE | big data | databricks | google | Hadoop

Startup Crunches 100 Terabytes of Data in a Record 23 Minutes

BY KLINT FINLEY 10.13.14 | 2:36 PM | PERMALINK

Share 1.1k Tweet 789 +1 75 Share 565 Pin it

GIGAOM EVENTS RESEARCH SIGN IN SUBSCRIBE

Cloud Data Media Mobile Science & Energy Social & Web Podcasts

Gigaom Research. Get unlimited market intelligence from over 200 i

MUST READS

Google launches Contributor, a crowdfunding tool for publishers

Net neutrality looks doomed in Europe before it even gets started

Five tech products that designers have fallen in love with

Databricks demolishes big data benchmark to prove Spark is fast on disk, too

by Derrick Harris Oct. 10, 2014 - 1:49 PM PST

1 Comment

WHY SORTING?

- Stresses "shuffle" which underpins everything from SQL to Mllib

- Sorting is challenging b/c there is no reduction in data

- Sort 100 TB = 500 TB disk I/O and 200 TB network

Engineering Investment in Spark:

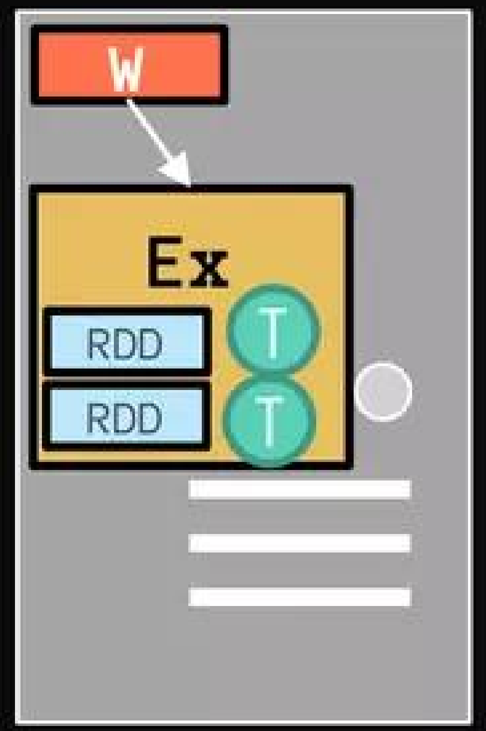
- Sort-based shuffle (SPARK-2045)
- Netty native network transport (SPARK-2468)
- External shuffle service (SPARK-3796)

Clever Application level Techniques:

- GC and cache friendly memory layout
- Pipelining

TECHNIQUE USED FOR 100 TB SORT

- Intel Xeon CPU E5 2670 @ 2.5 GHz w/ 32 cores
- 244 GB of RAM
- 8 x 800 GB SSD and RAID 0 setup formatted with /ext4
- ~9.5 Gbps (*1.1 GBps*) bandwidth between 2 random nodes



EC2: i2.8xlarge

(206 workers)

- 32 slots per machine
- 6,592 slots total

- Each record: 100 bytes (10 byte key & 90 byte value)
- OpenJDK 1.7
- HDFS 2.4.1 w/ short circuit local reads enabled
- Apache Spark 1.2.0
- Speculative Execution off
- Increased Locality Wait to infinite
- Compression turned off for input, output & network
- Used Unsafe to put all the data off-heap and managed it manually (i.e. never triggered the GC)

=

groupByKey

sortByKey

reduceByKey

`spark.shuffle.spill=false`

(Affects reducer side and keeps all the data in memory)

EXTERNAL SHUFFLE SERVICE

- Worker JVM serves files

- Must turn this on for dynamic allocation in YARN

- Node Manager serves files

OLD TECHNIQUE FOR SERVING MAP OUTPUT FILES

- Was slow because it had to copy the data 3 times

Map output file
on local dir

NETTY NATIVE TRANSPORT

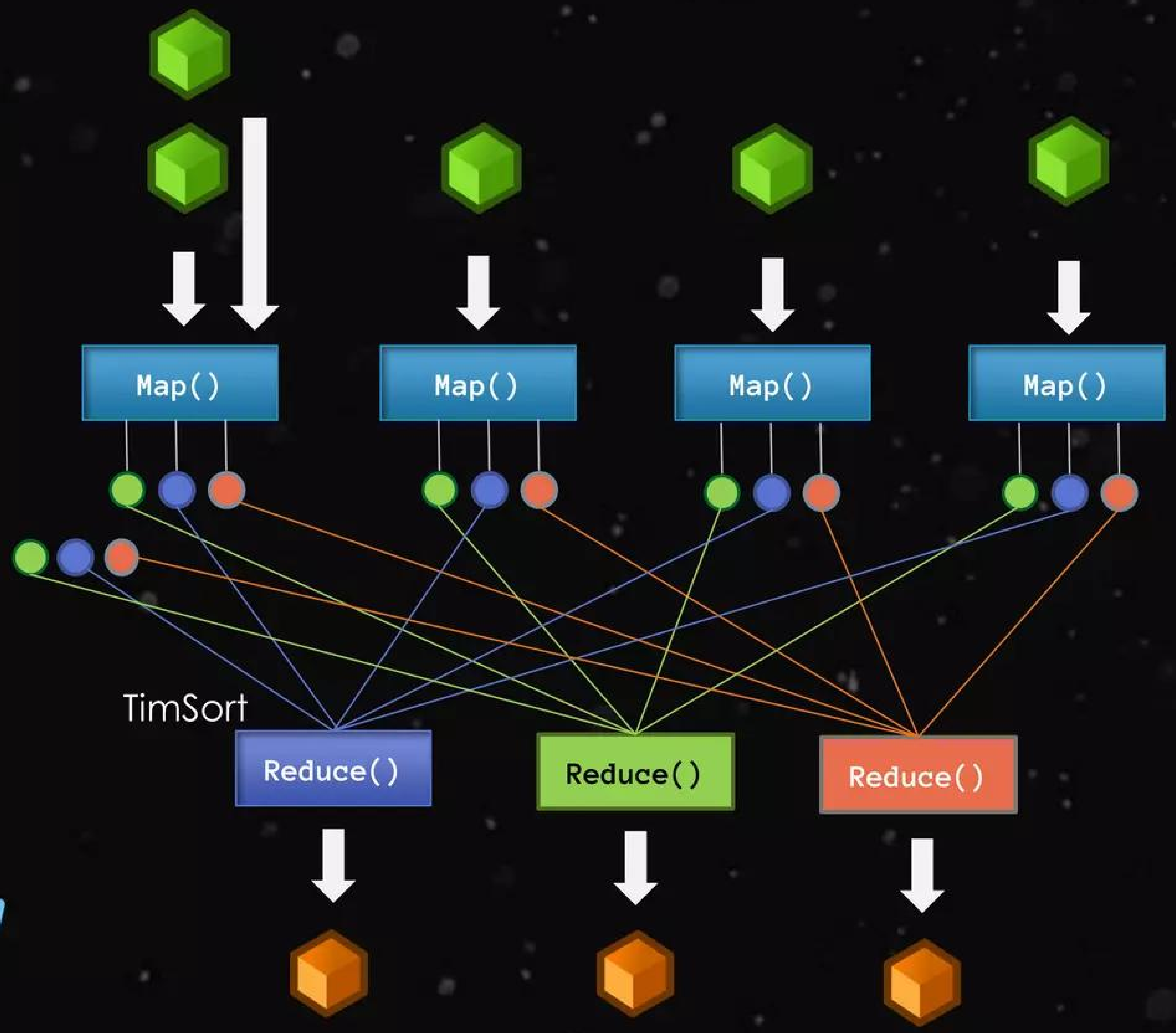
- Uses a technique called zero-copy
- Is a map-side optimization to serve data very quickly to requesting reducers

HASH BASED SHUFFLE

< 10,000 reducers

= 5 blocks

- Notice that map has to keep 3 file handles open



- Entirely bounded by I/O reading from HDFS and writing out locally sorted files

- Mostly network bound

SORT BASED SHUFFLE

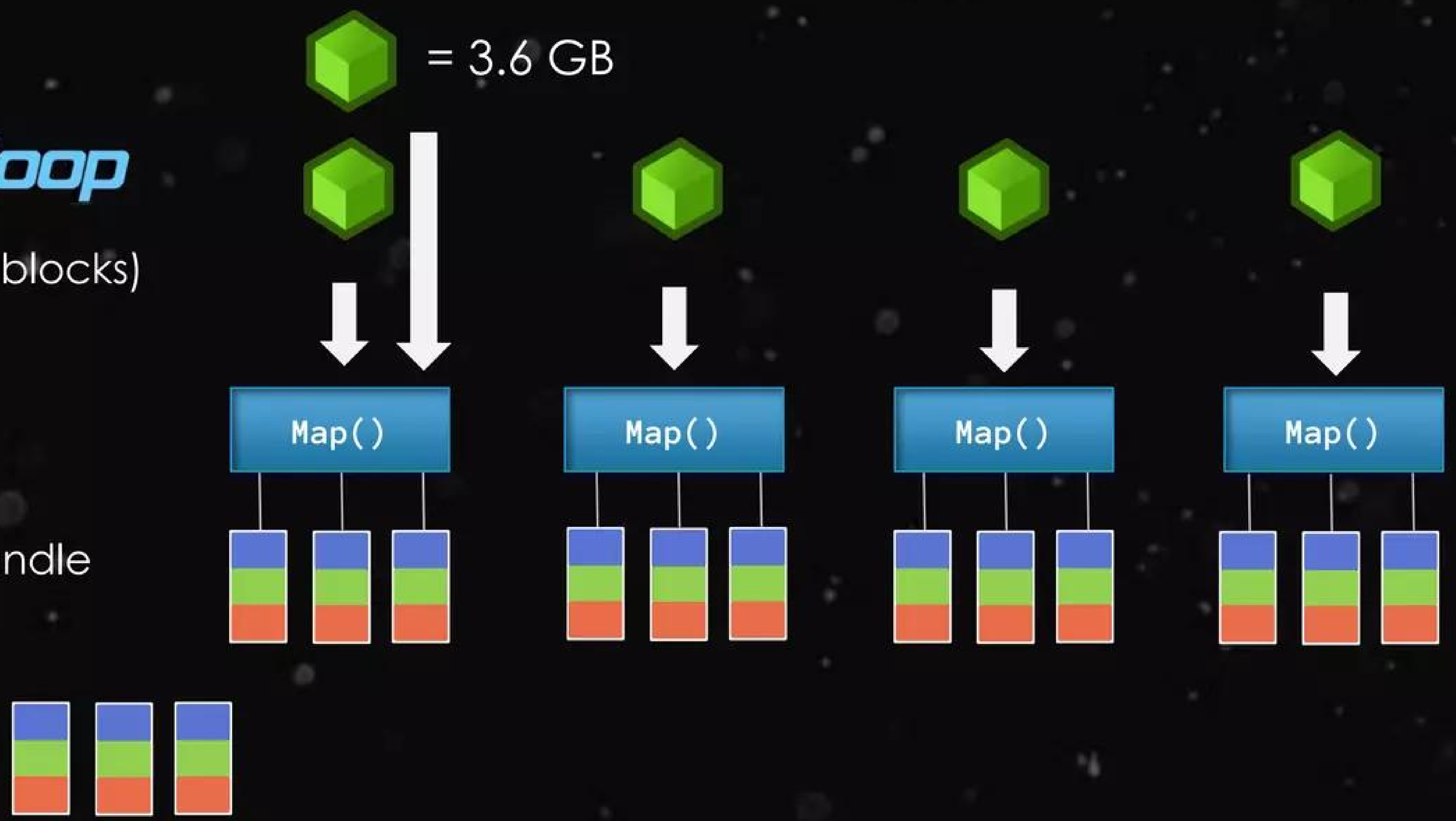
NEW

250,000+ reducers!

hadoop HDFS

(28,000 unique blocks)
RF = 2

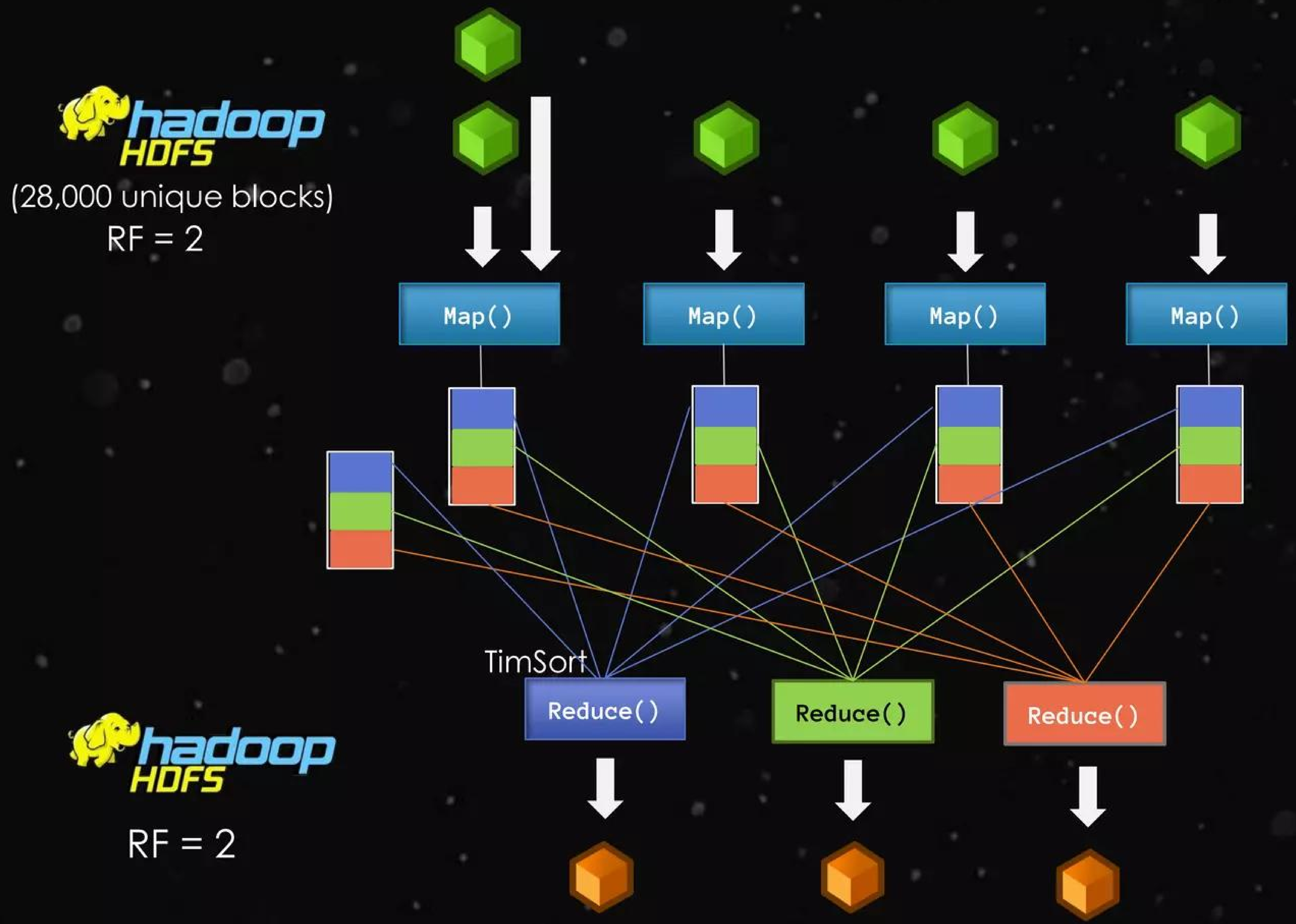
- Only one file handle open at a time



SORT BASED SHUFFLE

NEW

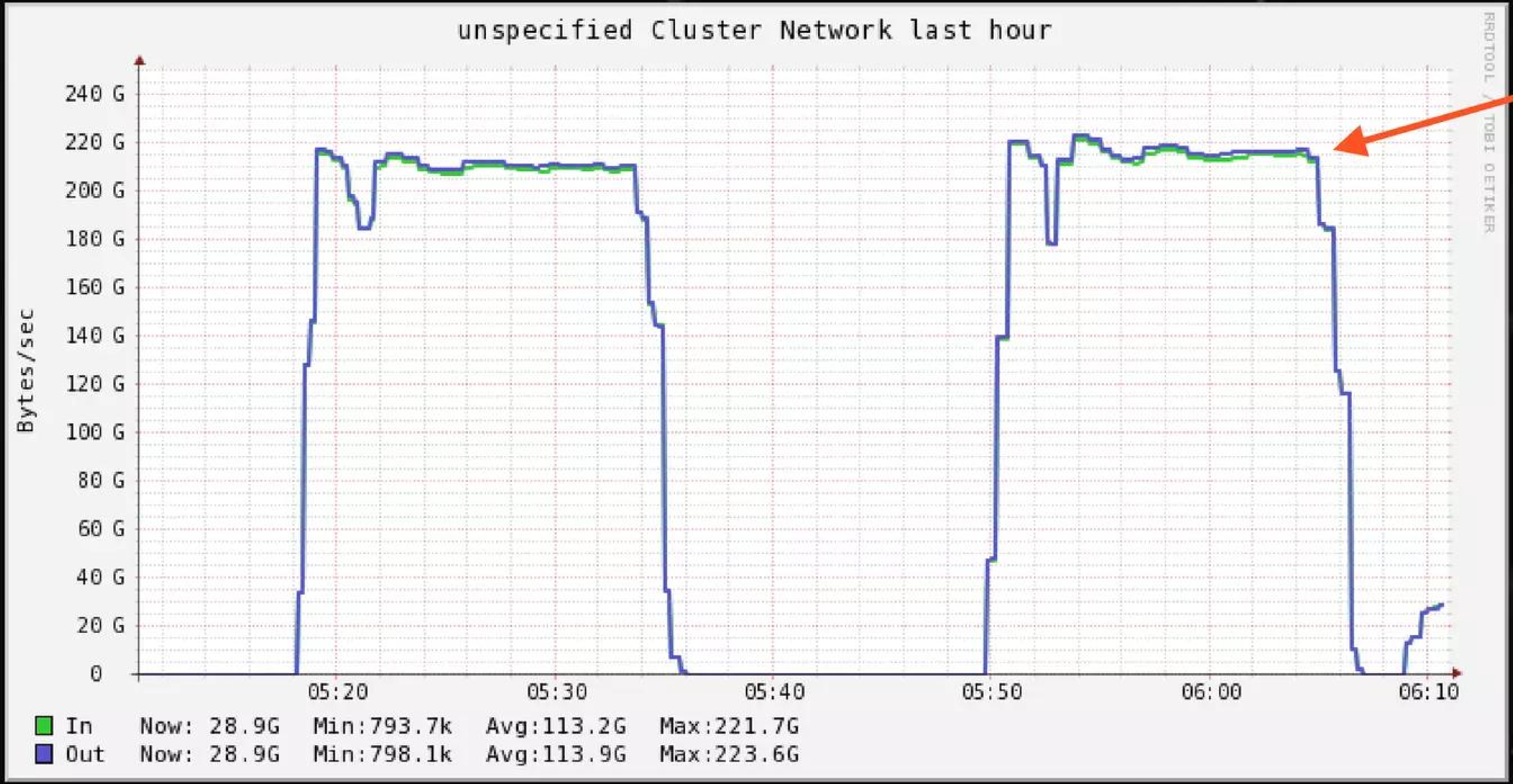
250,000+ reducers!



- 5 waves of maps
- 5 waves of reduces

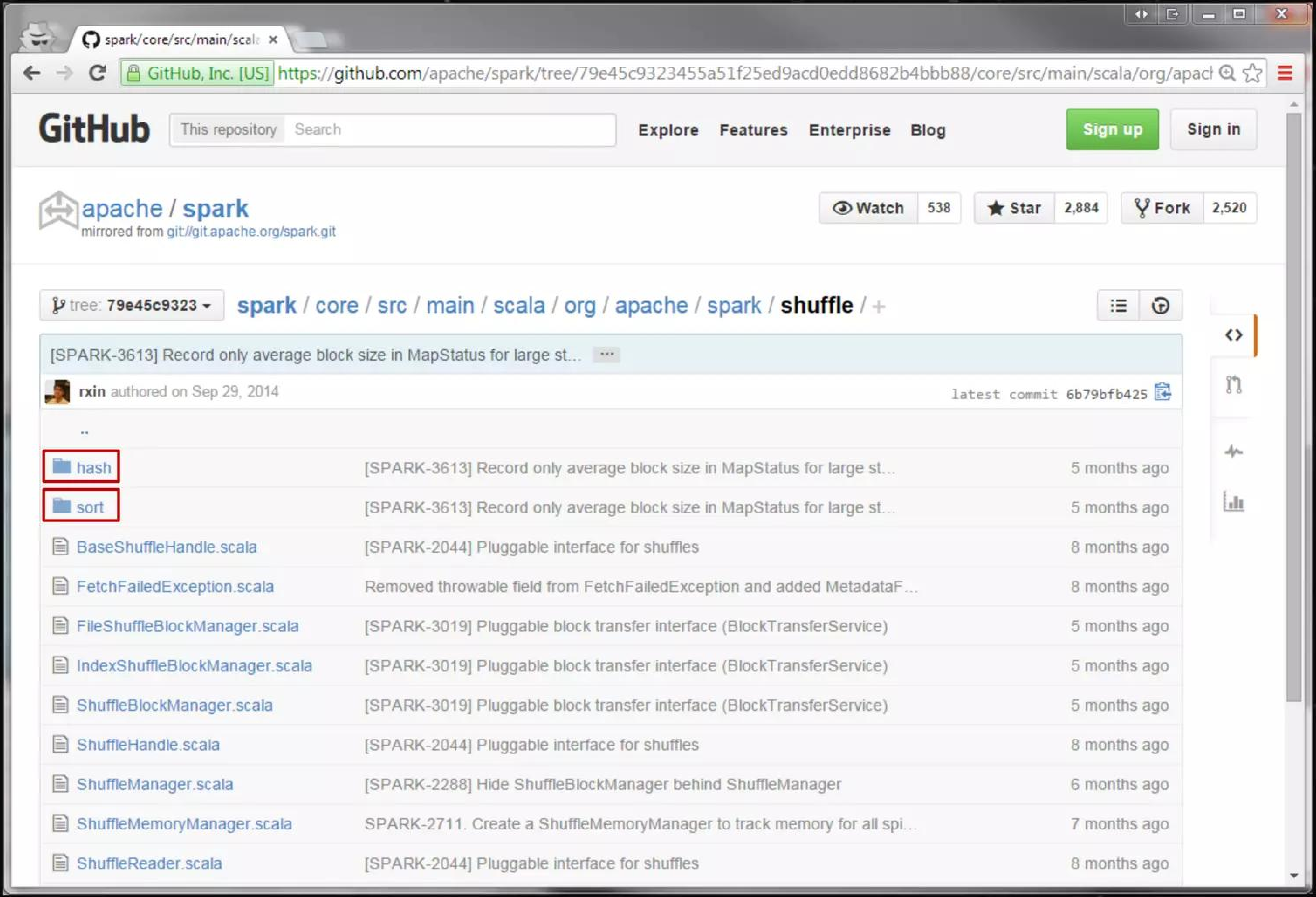
MergeSort!

NETWORK TRANSPORT



- Actual final run
- Fully saturated the 10 Gbit link

Sustaining 1.1GB/s/node during shuffle


spark/core/src/main/scala
GitHub, Inc. [US] https://github.com/apache/spark/tree/79e45c9323455a51f25ed9acd0edd8682b4bbb88/core/src/main/scala/org/apach
GitHub
This repository Search
Explore Features Enterprise Blog
Sign up
Sign in
apache / spark
mirrored from git://git.apache.org/spark.git
Watch 538
Star 2,884
Fork 2,520
tree: 79e45c9323
spark / core / src / main / scala / org / apache / spark / shuffle / +
[SPARK-3613] Record only average block size in MapStatus for large st...
rxin authored on Sep 29, 2014
latest commit 6b79bfb425
..
hash [SPARK-3613] Record only average block size in MapStatus for large st... 5 months ago
sort [SPARK-3613] Record only average block size in MapStatus for large st... 5 months ago
BaseShuffleHandle.scala [SPARK-2044] Pluggable interface for shuffles 8 months ago
FetchFailedException.scala Removed throwable field from FetchFailedException and added MetadataF... 8 months ago
FileShuffleBlockManager.scala [SPARK-3019] Pluggable block transfer interface (BlockTransferService) 5 months ago
IndexShuffleBlockManager.scala [SPARK-3019] Pluggable block transfer interface (BlockTransferService) 5 months ago
ShuffleBlockManager.scala [SPARK-3019] Pluggable block transfer interface (BlockTransferService) 5 months ago
ShuffleHandle.scala [SPARK-2044] Pluggable interface for shuffles 8 months ago
ShuffleManager.scala [SPARK-2288] Hide ShuffleBlockManager behind ShuffleManager 6 months ago
ShuffleMemoryManager.scala SPARK-2711. Create a ShuffleMemoryManager to track memory for all spi... 7 months ago
ShuffleReader.scala [SPARK-2044] Pluggable interface for shuffles 8 months ago

Link

UserID	Name	Age	Location	Pet
28492942	John Galt	32	New York	Sea Horse
95829324	Winston Smith	41	Oceania	Ant
92871761	Tom Sawyer	17	Mississippi	Raccoon
37584932	Carlos Hinojosa	33	Orlando	Cat
73648274	Luis Rodriguez	34	Orlando	Dogs

SPARK SQL

databricks

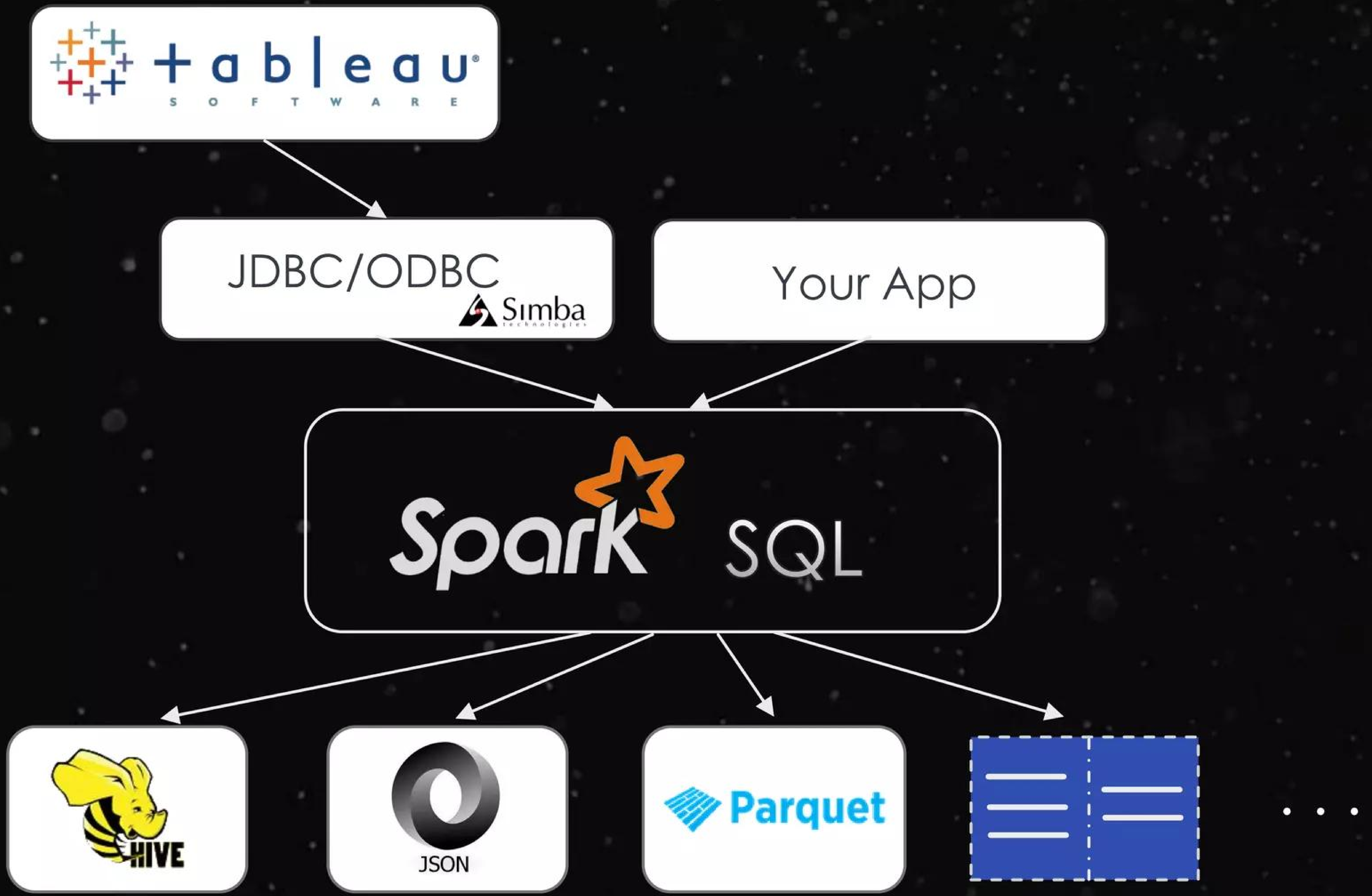
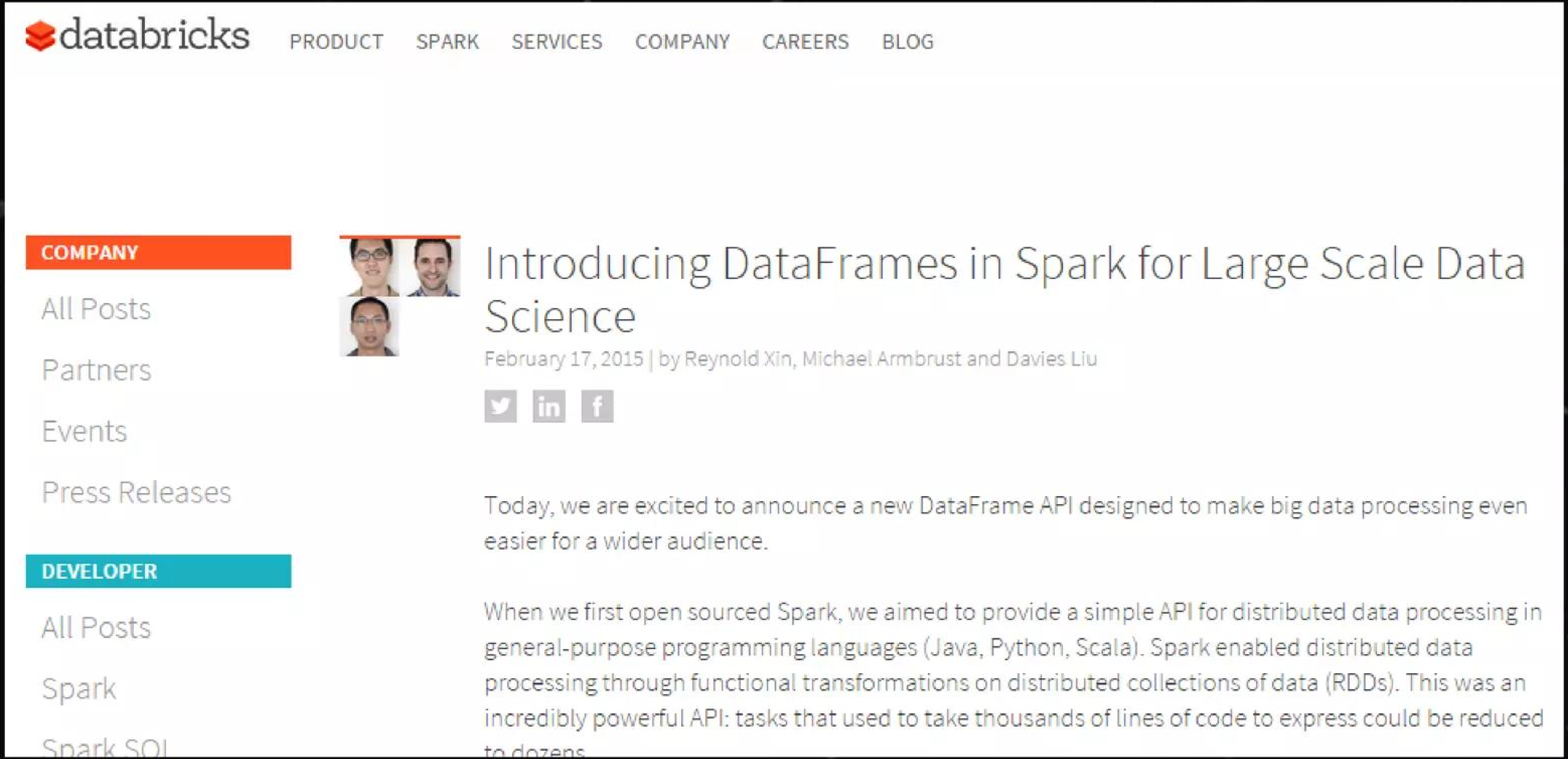


tableau
SOFTWARE
JDBC/ODBC
Simba
Your App
Spark SQL
HIVE
JSON
Parquet
. . .

SchemaRDD

- RDD of Row objects, each representing a record
- Row objects = type + col. name of each
- Stores data very efficiently by taking advantage of the schema
- SchemaRDDs are also regular RDDs, so you can run transformations like map() or filter()
- Allows new operations, like running SQL on objects

https://databricks.com/blog/2015/02/17/introducing-dataframes-in-spark-for-large-scale-data-science.html



databricks PRODUCT SPARK SERVICES COMPANY CAREERS BLOG

COMPANY

- All Posts
- Partners
- Events
- Press Releases

DEVELOPER

- All Posts
- Spark
- Spark SQL

Introducing DataFrames in Spark for Large Scale Data Science

February 17, 2015 | by Reynold Xin, Michael Armbrust and Davies Liu

Today, we are excited to announce a new DataFrame API designed to make big data processing even easier for a wider audience.

When we first open sourced Spark, we aimed to provide a simple API for distributed data processing in general-purpose programming languages (Java, Python, Scala). Spark enabled distributed data processing through functional transformations on distributed collections of data (RDDs). This was an incredibly powerful API: tasks that used to take thousands of lines of code to express could be reduced to dozens.

INFERRING THE SCHEMA USING REFLECTION

Warning!

Only looks at first row

```
# sc is an existing SparkContext.
from pyspark.sql import SQLContext, Row
sqlContext = SQLContext(sc)

# Load a text file and convert each line to a Row.
lines = sc.textFile("examples/src/main/resources/people.txt")
parts = lines.map(lambda l: l.split(","))
people = parts.map(lambda p: Row(name=p[0], age=int(p[1])))

# Infer the schema, and register the SchemaRDD as a table.
schemaPeople = sqlContext.inferSchema(people)
schemaPeople.registerTempTable("people")

# SQL can be run over SchemaRDDs that have been registered as a table.
teenagers = sqlContext.sql("SELECT name FROM people WHERE age >= 13 AND age <= 19")

# The results of SQL queries are RDDs and support all the normal RDD operations.
teenNames = teenagers.map(lambda p: "Name: " + p.name)
for teenName in teenNames.collect():
  print teenName
```

PROGRAMMATICALLY SPECIFYING THE SCHEMA

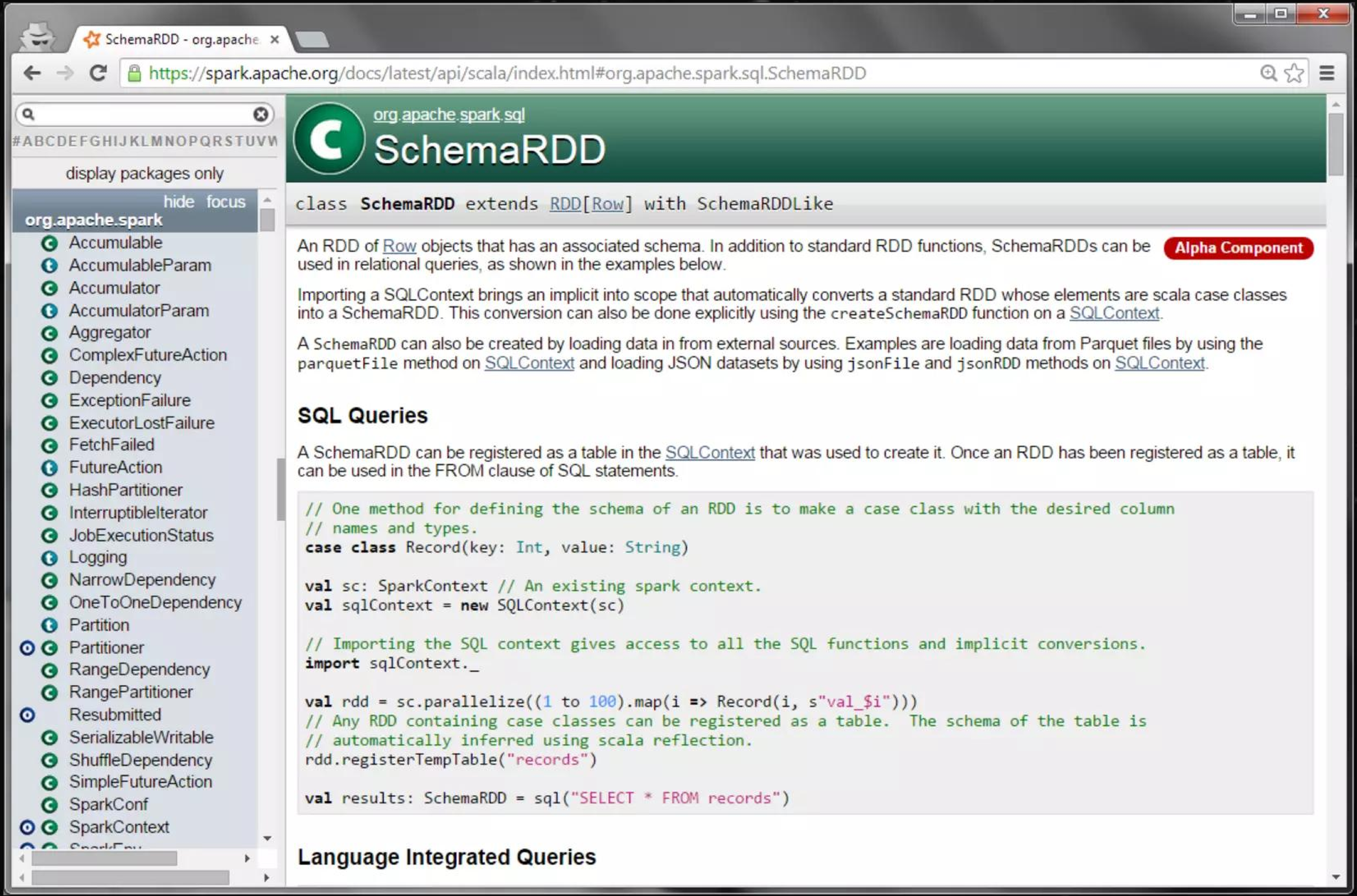
```python
# Import SQLContext and data types
from pyspark.sql import *

# sc is an existing SparkContext.
sqlContext = SQLContext(sc)

# Load a text file and convert each line to a tuple.
lines = sc.textFile("examples/src/main/resources/people.txt")
parts = lines.map(lambda l: l.split(","))
people = parts.map(lambda p: (p[0], p[1].strip()))

# The schema is encoded in a string.
schemaString = "name age"

fields = [StructField(field_name, StringType(), True) for field_name in schemaString.split()]
schema = StructType(fields)

# Apply the schema to the RDD.
schemaPeople = sqlContext.applySchema(people, schema)

# Register the SchemaRDD as a table.
schemaPeople.registerTempTable("people")

# SQL can be run over SchemaRDDs that have been registered as a table.
results = sqlContext.sql("SELECT name FROM people")

# The results of SQL queries are RDDs and support all the normal RDD operations.
names = results.map(lambda p: "Name: " + p.name)
for name in names.collect():
  print name
```

```python
# sqlContext from the previous example is used in this example.

schemaPeople # The SchemaRDD from the previous example.

# SchemaRDDs can be saved as Parquet files, maintaining the schema information.
schemaPeople.saveAsParquetFile("people.parquet")

# Read in the Parquet file created above.  Parquet files are self-describing so the schema is preserved.
# The result of loading a parquet file is also a SchemaRDD.
parquetFile = sqlContext.parquetFile("people.parquet")

# Parquet files can also be registered as tables and then used in SQL statements.
parquetFile.registerTempTable("parquetFile");
teenagers = sqlContext.sql("SELECT name FROM parquetFile WHERE age >= 13 AND age <= 19")
teenNames = teenagers.map(lambda p: "Name: " + p.name)
for teenName in teenNames.collect():
  print teenName
```

Parquet

Configuration of Parquet can be done using the `setConf` method on SQLContext or by running `SET key=value` commands using SQL.

Property Name	Default	Meaning
`spark.sql.parquet.binaryAsString`	false	Some other Parquet-producing systems, in particular Impala and older versions of Spark SQL, do not differentiate between binary data and strings when writing out the Parquet schema. This flag tells Spark SQL to interpret binary data as a string to provide compatibility with these systems.
`spark.sql.parquet.cacheMetadata`	true	Turns on caching of Parquet schema metadata. Can speed up querying of static data.
`spark.sql.parquet.compression.codec`	gzip	Sets the compression codec use when writing Parquet files. Acceptable values include: uncompressed, snappy, gzip, lzo.
`spark.sql.parquet.filterPushdown`	false	Turn on Parquet filter pushdown optimization. This feature is turned off by default because of a known bug in Paruet 1.6.0rc3 (PARQUET-136). However, if your table doesn't contain any nullable string or binary columns, it's still safe to turn this feature on.
`spark.sql.hive.convertMetastoreParquet`	true	When set to false, Spark SQL will use the Hive SerDe for parquet tables instead of the built in support.


SchemaRDD - org.apache

https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.SchemaRDD

#ABCDEFGHIJKLMNOPQRSTUVW

display packages only

hide focus

org.apache.spark

- Accumulable
- AccumulableParam
- Accumulator
- AccumulatorParam
- Aggregator
- ComplexFutureAction
- Dependency
- ExceptionFailure
- ExecutorLostFailure
- FetchFailed
- FutureAction
- HashPartitioner
- InterruptibleIterator
- JobExecutionStatus
- Logging
- NarrowDependency
- OneToOneDependency
- Partition
- Partitioner
- RangeDependency
- RangePartitioner
- Resubmitted
- SerializableWritable
- ShuffleDependency
- SimpleFutureAction
- SparkConf
- SparkContext

org.apache.spark.sql

SchemaRDD

`class SchemaRDD extends RDD[Row] with SchemaRDDLike`

An RDD of Row objects that has an associated schema. In addition to standard RDD functions, SchemaRDDs can be used in relational queries, as shown in the examples below. **Alpha Component**

Importing a SQLContext brings an implicit into scope that automatically converts a standard RDD whose elements are scala case classes into a SchemaRDD. This conversion can also be done explicitly using the `createSchemaRDD` function on a SQLContext.

A `SchemaRDD` can also be created by loading data in from external sources. Examples are loading data from Parquet files by using the `parquetFile` method on SQLContext and loading JSON datasets by using `jsonFile` and `jsonRDD` methods on SQLContext.

SQL Queries

A SchemaRDD can be registered as a table in the SQLContext that was used to create it. Once an RDD has been registered as a table, it can be used in the FROM clause of SQL statements.

```
// One method for defining the schema of an RDD is to make a case class with the desired column
// names and types.
case class Record(key: Int, value: String)

val sc: SparkContext // An existing spark context.
val sqlContext = new SQLContext(sc)

// Importing the SQL context gives access to all the SQL functions and implicit conversions.
import sqlContext._

val rdd = sc.parallelize((1 to 100).map(i => Record(i, s"val_$i")))
// Any RDD containing case classes can be registered as a table.  The schema of the table is
// automatically inferred using scala reflection.
rdd.registerTempTable("records")

val results: SchemaRDD = sql("SELECT * FROM records")
```

Language Integrated Queries

Link

```
TwitterUtils.createStream(...)
    .filter(_.getText.contains("Spark"))
    .countByWindow(Seconds(5))
```

Spark STREAMING

databricks

TCP socket

Kafka

Flume

HDFS

S3

Kinesis

Twitter

- Scalable
- High-throughput
- Fault-tolerant

Spark STREAMING

HDFS

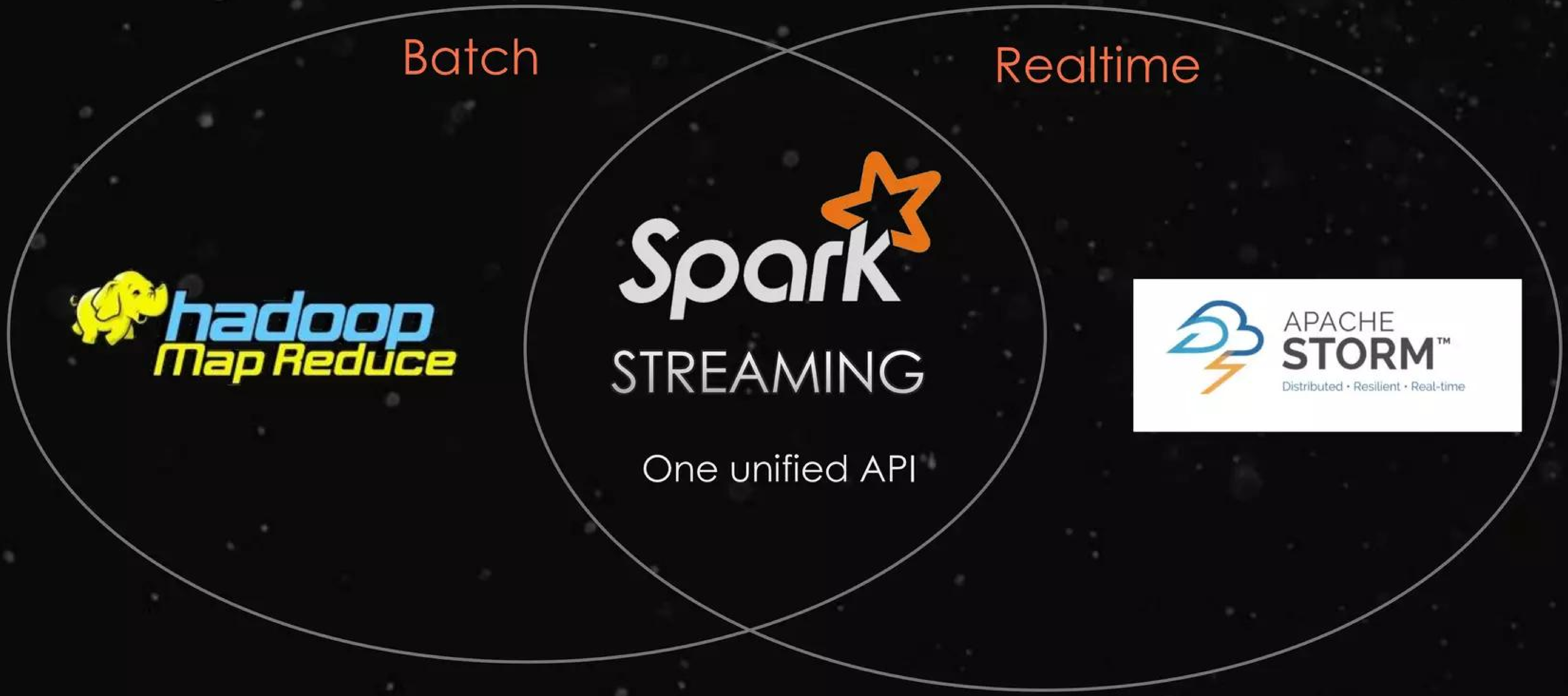
Cassandra

Dashboards

Databases

Complex algorithms can be expressed using:

- Spark transformations: `map()`, `reduce()`, `join()`, `etc`
- MLlib + GraphX
- SQL


Batch
Realtime
hadoop
Map Reduce
Spark
STREAMING
One unified API
APACHE
STORM™
Distributed • Resilient • Real-time

Tathagata Das (TD)

- Lead developer of Spark Streaming + Committer on Apache Spark core
- Helped re-write Spark Core internals in 2012 to make it 10x faster to support Streaming use cases
- On leave from UC Berkeley PhD program
- Ex: Intern @ Amazon, Intern @ Conviva, Research Assistant @ Microsoft Research India
- 1 guy; does not scale

Spark STREAMING

- Scales to 100s of nodes
- Batch sizes as small at half a second
- Processing latency as low as 1 second
- Exactly-once semantics no matter what fails

USE CASES (live statistics)

Page views

Kafka for buffering

Spark STREAMING

Spark for processing

USE CASES (Anomaly Detection)

Smart meter readings

Live weather data

Join 2 live data sources

Spark STREAMING


Batches every X seconds
Input data stream
Spark
STREAMING
Spark
CORE
Batches of
processed data

DSTREAM

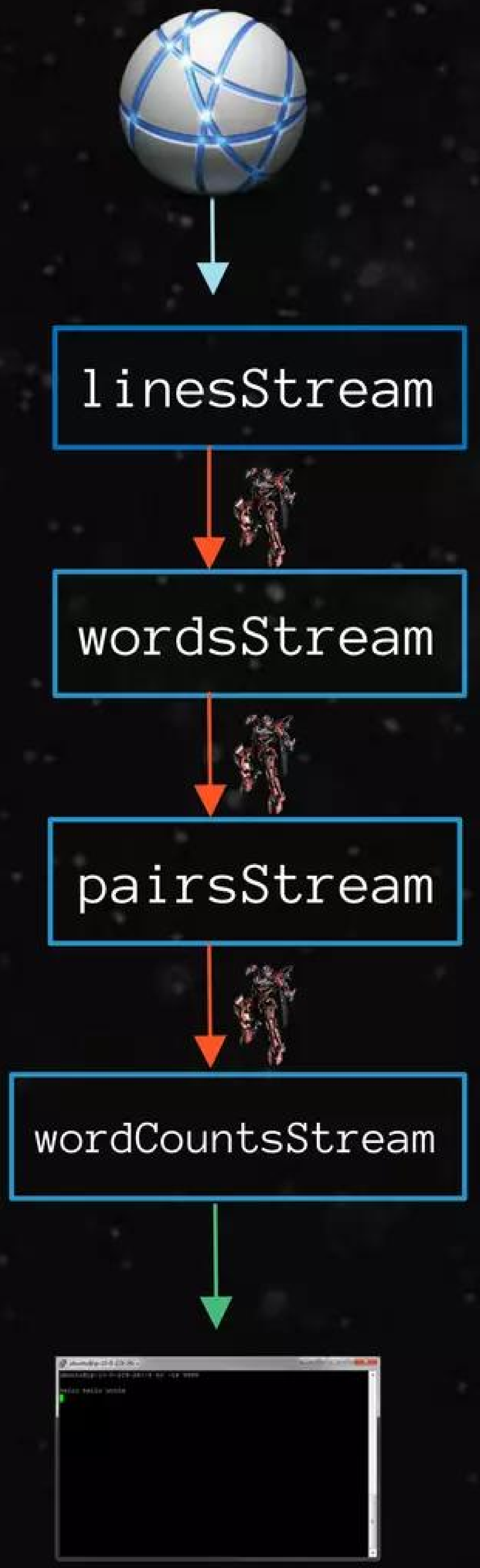
(Discretized Stream)

Batch interval = 5 seconds

One RDD is created every 5 seconds

TRANSFORMING DSTREAMS

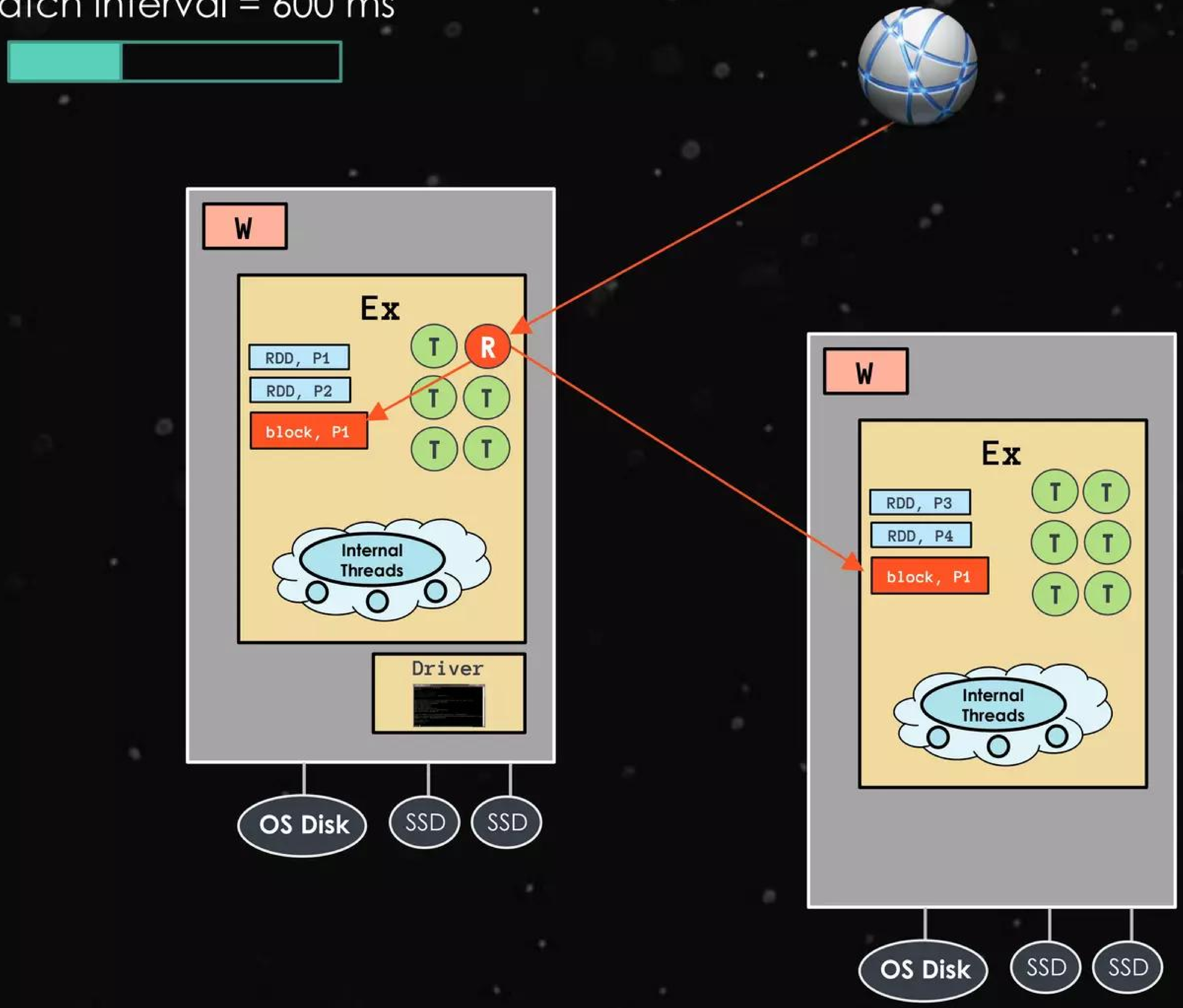
```python
from pyspark import SparkContext
from pyspark.streaming import StreamingContext

# Create a local StreamingContext with two working thread and batch interval of 1 second
sc = SparkContext("local[2]", "NetworkWordCount")
ssc = StreamingContext(sc, 5)


# Create a DStream that will connect to hostname:port, like localhost:9999
linesDStream = ssc.socketTextStream("localhost", 9999)


# Split each line into words
wordsDStream = linesDStream.flatMap(lambda line: line.split(" "))


# Count each word in each batch
pairsDStream = wordsDStream.map(lambda word: (word, 1))
wordCountsDStream = pairsDStream.reduceByKey(lambda x, y: x + y)

# Print the first ten elements of each RDD generated in this DStream to the console
wordCountsDStream.pprint()


ssc.start()             # Start the computation
ssc.awaitTermination()  # Wait for the computation to terminate
```




```
$ nc -lk 9999

hello world
```

```
$ ./network_wordcount.py localhost 9999

. . .
---------------------------
Time: 2015-04-25 15:25:21
---------------------------
(hello, 2)
(world, 1)
```

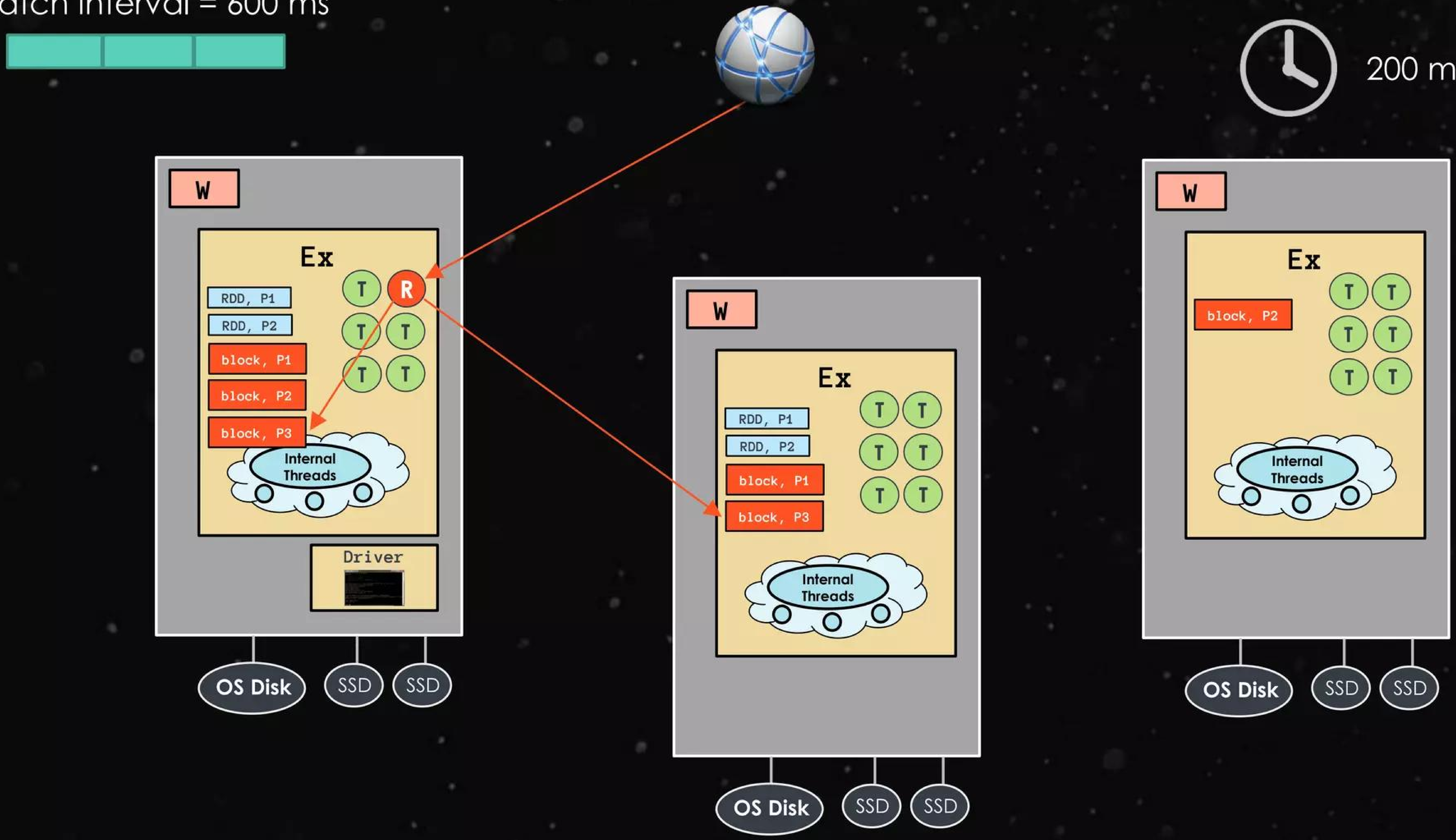
Batch interval = 600 ms



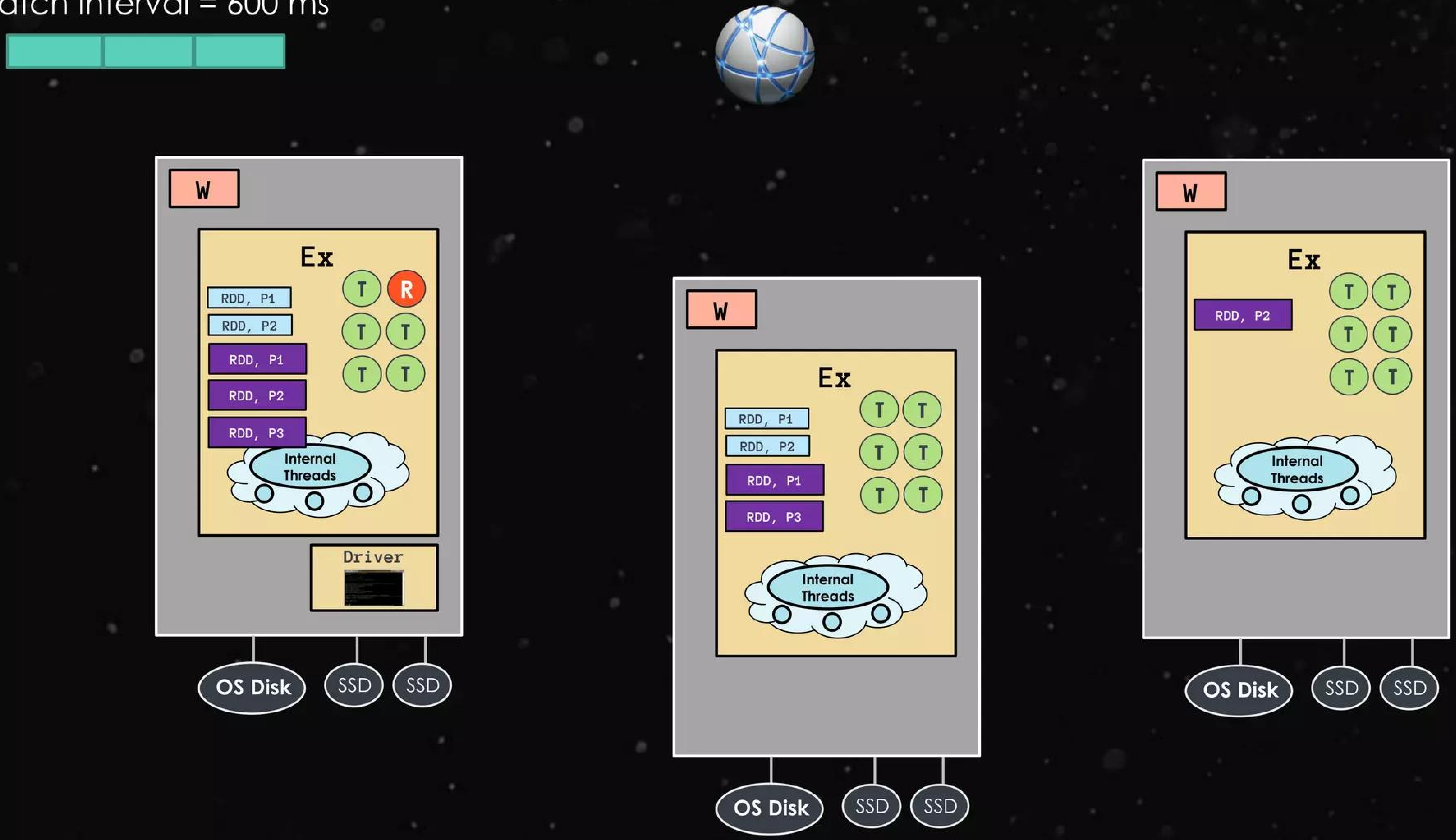
Batch interval = 600 ms
200 ms later
W
Ex
RDD, P1
RDD, P2
block, P1
block, P2
R
T
Internal Threads
Driver
OS Disk
SSD
SSD
W
Ex
RDD, P3
RDD, P4
block, P1
Internal Threads
OS Disk
SSD
SSD
W
Ex
block, P2
Internal Threads
OS Disk
SSD
SSD

Batch interval = 600 ms

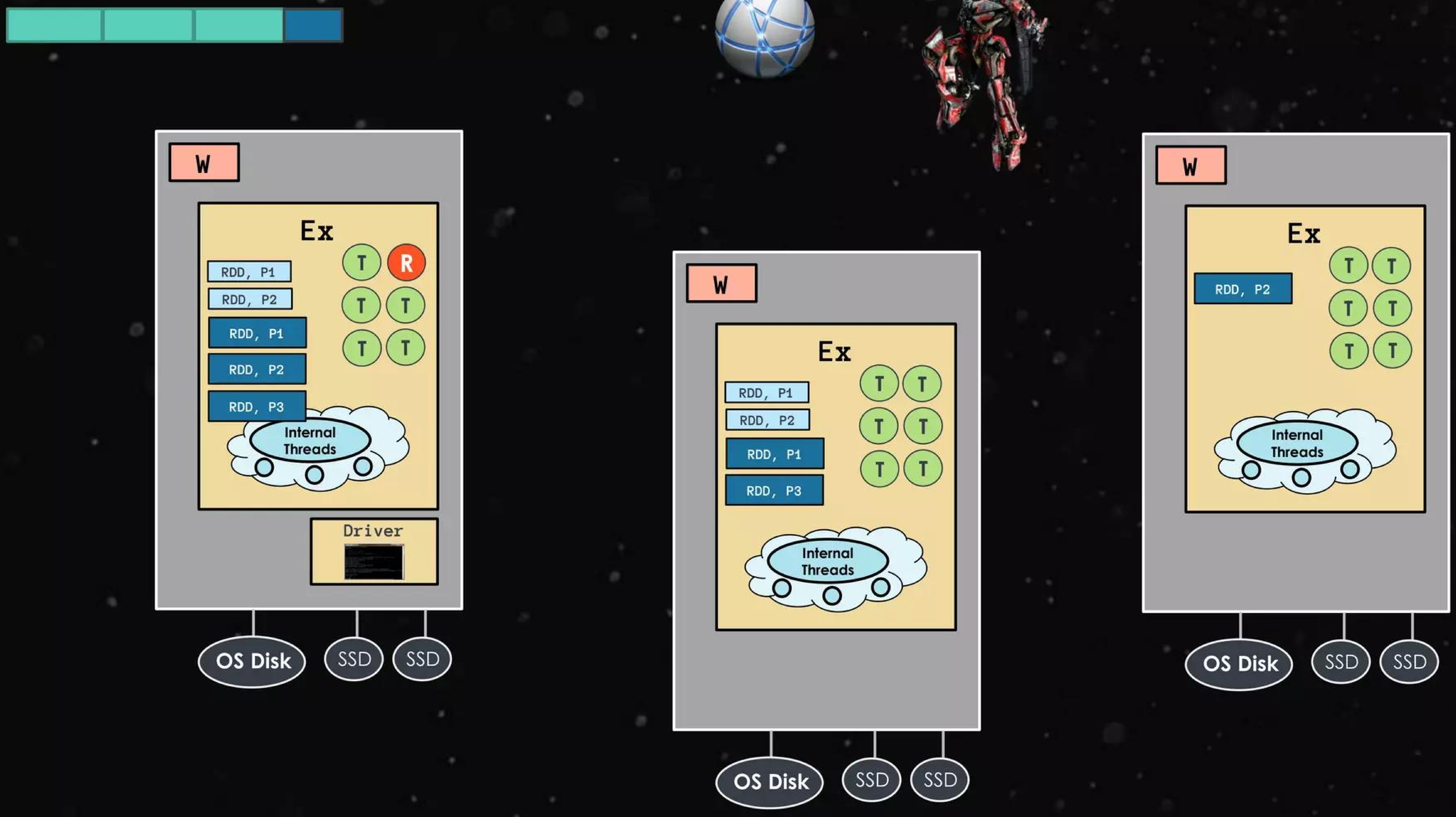
200 ms later



Batch interval = 600 ms



Batch interval = 600 ms



IndexTweetsLive - St ×

localhost:4040/streaming/

Spark | Stages | Storage | Environment | Executors | Streaming | IndexTweetsLive application UI

Streaming

Started at: Wed Oct 22 06:11:53 PDT 2014
Time since start: 27 minutes 20 seconds
Network receivers: 1
Batch interval: 1 second
Processed batches: 1641
Waiting batches: 0

Statistics over last 100 processed batches

Receiver Statistics

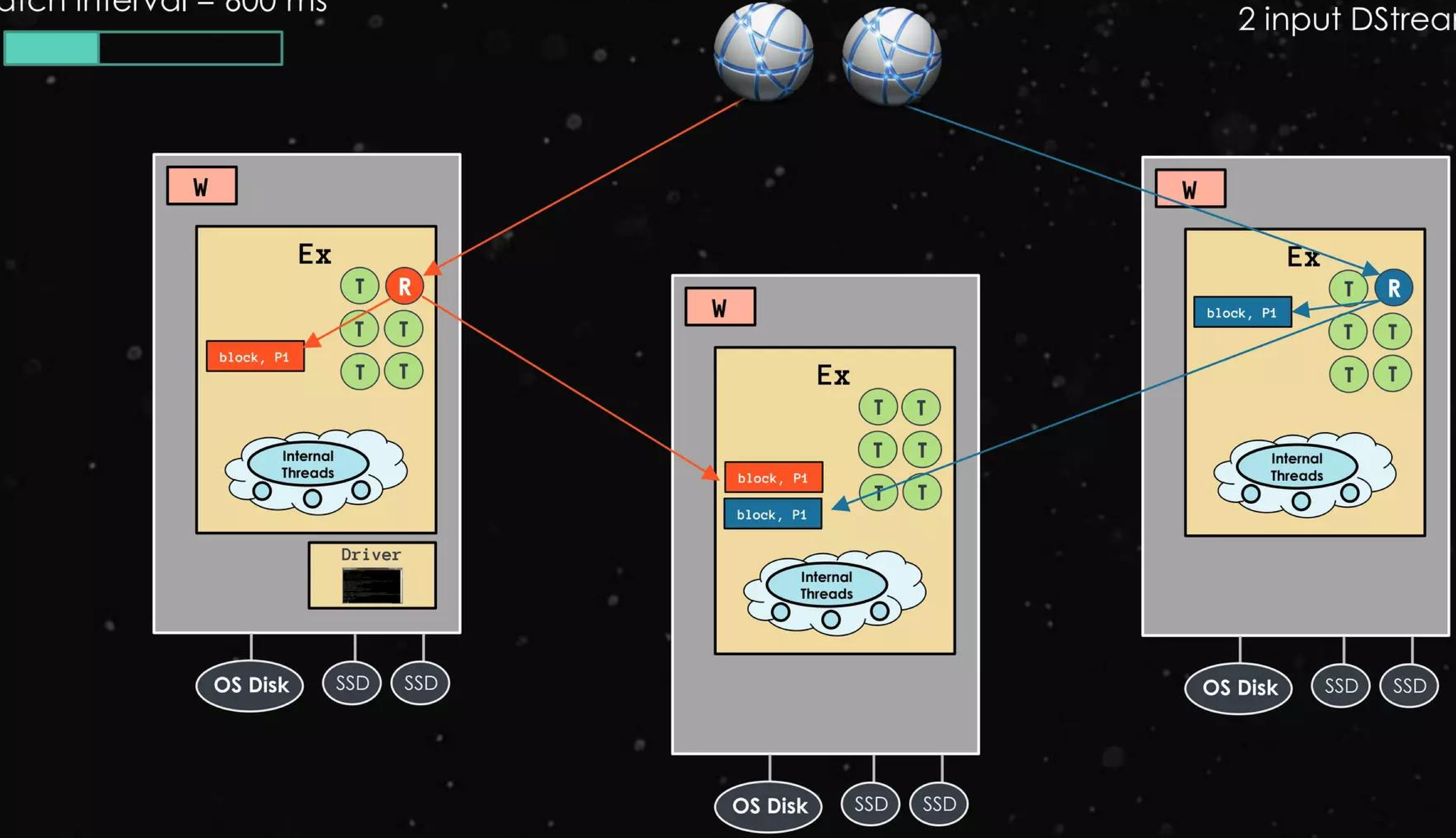
Receiver	Status	Location	Records in last batch [2014/10/22 06:39:14]	Minimum rate [records/sec]	Median rate [records/sec]	Maximum rate [records/sec]	Last Error
TwitterReceiver-0	ACTIVE	localhost	39	0	61	151	-

Batch Processing Statistics

Metric	Last batch	Minimum	25th percentile	Median	75th percentile	Maximum
Processing Time	31 ms	5 ms	39 ms	56 ms	457 ms	2 seconds 289 ms
Scheduling Delay	0 ms	0 ms	0 ms	0 ms	1 ms	803 ms
Total Delay	31 ms	31 ms	40 ms	57 ms	499 ms	2 seconds 289 ms

Batch interval = 600 ms

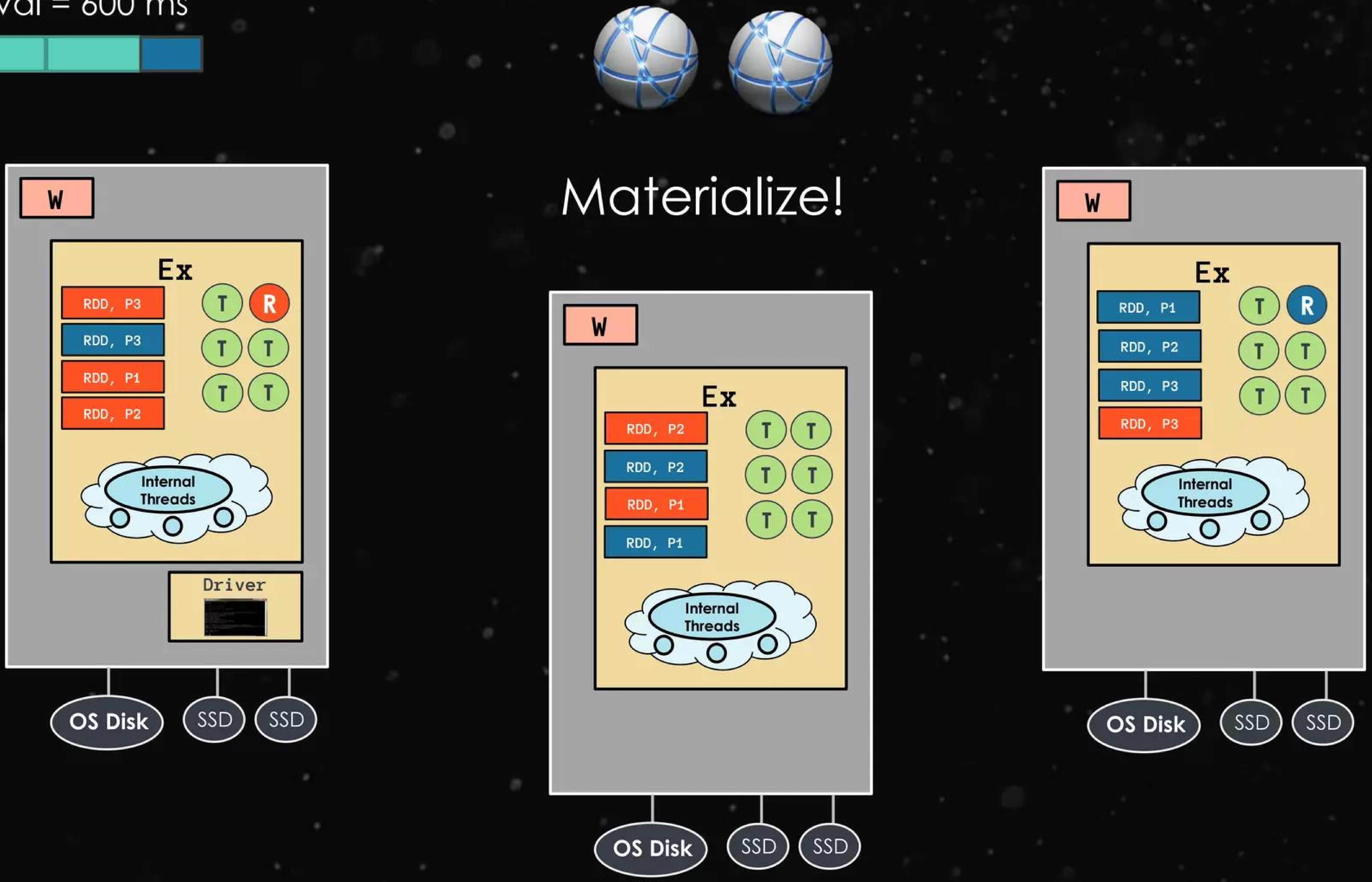
2 input DStreams



Batch interval = 600 ms

Batch interval = 600 ms

Materialize!



Batch interval = 600 ms
Union!
W
Ex
RDD, P1
RDD, P2
RDD, P3
RDD, P5
T
R
Internal Threads
Driver
OS Disk
SSD
SSD
W
Ex
RDD, P1
RDD, P2
RDD, P4
RDD, P6
T
Internal Threads
OS Disk
SSD
SSD
W
Ex
RDD, P3
RDD, P4
RDD, P5
RDD, P6
T
R
Internal Threads
OS Disk
SSD
SSD

BASIC

- File systems
- Socket Connections
- Akka Actors

Sources directly available in `StreamingContext` API

ADVANCED

- Kafka
- Flume
- Twitter

Requires linking against extra dependencies

CUSTOM

- Anywhere

Requires implementing user-defined receiver https://spark.apache.org/docs/latest/streaming-flume-integration.html

Spark 1.2.0 Overview Programming Guides API Docs Deploying More

Spark Streaming + Flume Integration Guide

Apache Flume is a distributed, reliable, and available service for efficiently collecting, aggregating, and moving large amounts of log data. Here we explain how to configure Flume and Spark Streaming to receive data from Flume. There are two approaches to this.

Approach 1: Flume-style Push-based Approach

Flume is designed to push data between Flume agents. In this approach, Spark Streaming essentially sets up a receiver that acts an Avro agent for Flume, to which Flume can push the data. Here are the configuration steps.

General Requirements

Choose a machine in your cluster such that

- When your Flume + Spark Streaming application is launched, one of the Spark workers must run on that machine.
- Flume can be configured to push data to a port on that machine.

Due to the push model, the streaming application needs to be up, with the receiver scheduled and listening on the chosen port, for Flume to be able push data.

Configuring Flume

Configure Flume agent to send data to an Avro sink by having the following in the configuration file.

```
agent.sinks = avrosink
```

https://spark.apache.org/docs/latest/streaming-kafka-integration.html

Spark 1.2.0 Overview Programming Guides API Docs Deploying More

Spark Streaming + Kafka Integration Guide

Apache Kafka is publish-subscribe messaging rethought as a distributed, partitioned, replicated commit log service. Here we explain how to configure Spark Streaming to receive data from Kafka.

1. **Linking:** In your SBT/Maven projrect definition, link your streaming application against the following artifact (see Linking section in the main programming guide for further information).

```
groupId = org.apache.spark
artifactId = spark-streaming-kafka_2.10
version = 1.2.0
```

2. **Programming:** In the streaming application code, import `KafkaUtils` and create input DStream as follows.

Scala **Java**

```
import org.apache.spark.streaming.kafka._

val kafkaStream = KafkaUtils.createStream(
```

TRANSFORMATIONS ON DSTREAMS

map($f(x)$)

reduce($f(x)$)

union(otherStream)

updateStateByKey($f(x)$)*

flatMap($f(x)$)

join(otherStream , [numTasks])

filter($f(x)$)

cogroup(otherStream , [numTasks])

RDD

transform($f(x)$)

RDD

repartition(numPartitions)

count( )

reduceAByKey($f(x)$, [numTasks])

countByValue( )

TRANSFORMATIONS ON DSTREAMS

pairs = (word, 1)
(cat, 1)

`updateStateByKey(` $f(x)$ `)`* : allows you to maintain arbitrary state while continuously updating it with new information.

To use:

1) Define the state
(an arbitrary data type)

2) Define the state update function
(specify with a function how to update the state using the previous state and new values from the input stream)

* Requires a checkpoint directory to be configured

To maintain a running count of each word seen in a text data stream *(here running count is an integer type of state)*:

```python
def updateFunction(newValues, runningCount):
    if runningCount is None:
        runningCount = 0
    return sum(newValues, runningCount)  # add the
        # new values with the previous running count
        # to get the new count
```

```python
runningCounts = pairs.updateStateByKey(updateFunction)
```

TRANSFORMATIONS ON DSTREAMS

RDD

`transform( f(x) )` : can be used to apply any RDD operation that is not exposed in the DStream API.

RDD

For example:

- Functionality to join every batch in a data stream with another dataset is not directly exposed in the DStream API.

- If you want to do real-time data cleaning by joining the input data stream with pre-computed spam information and then filtering based on it.

```python
spamInfoRDD = sc.pickleFile(...) # RDD containing spam information

# join data stream with spam information to do data cleaning
cleanedDStream = wordCounts.transform(lambda rdd:
                                      rdd.join(spamInfoRDD).filter(...))
```

or

MLlib

GraphX

WINDOW OPERATIONS

Window Length*: 3 time units

Sliding Interval*: 2 time units

* Both of these must be multiples of the batch interval of the source DStream


```
# Reduce last 30 seconds of data, every 10 seconds
windowedWordCounts = pairs.reduceByKeyAndWindow(lambda x, y: x + y, lambda x, y: x - y, 30, 10)
```

COMMON WINDOW OPERATIONS

`window(windowLength, slideInterval)`

`countByValueAndWindow(windowLength, slideInterval, [numTasks])`

`countByWindow(windowLength, slideInterval)`

`reduceByWindow(` $f(x)$ `, windowLength, slideInterval)`

`reduceByKeyAndWindow(` $f(x)$ `, windowLength, slideInterval, [numTasks])`

`reduceByKeyAndWindow(` $f(x)$ `,` $f^{-1}(x)$ `, windowLength, slideInterval, [numTasks])`

API Docs

- DStream
- PairDStreamFunctions

- JavaDStream
- JavaPairDStream

- DStream

OUTPUT OPERATIONS ON DSTREAMS

`print()`

`foreachRDD(` $f_{(x)}$ `)`

`saveAsTextFile(prefix, [suffix])`

`saveAsObjectFiles(prefix, [suffix])`

`saveAsHadoopFiles(prefix, [suffix])`

databricks