INTRO TO SPARK DEVELOPMENT

June 2015: Spark Summit West / San Francisco

http://training.databricks.com/intro.pdf

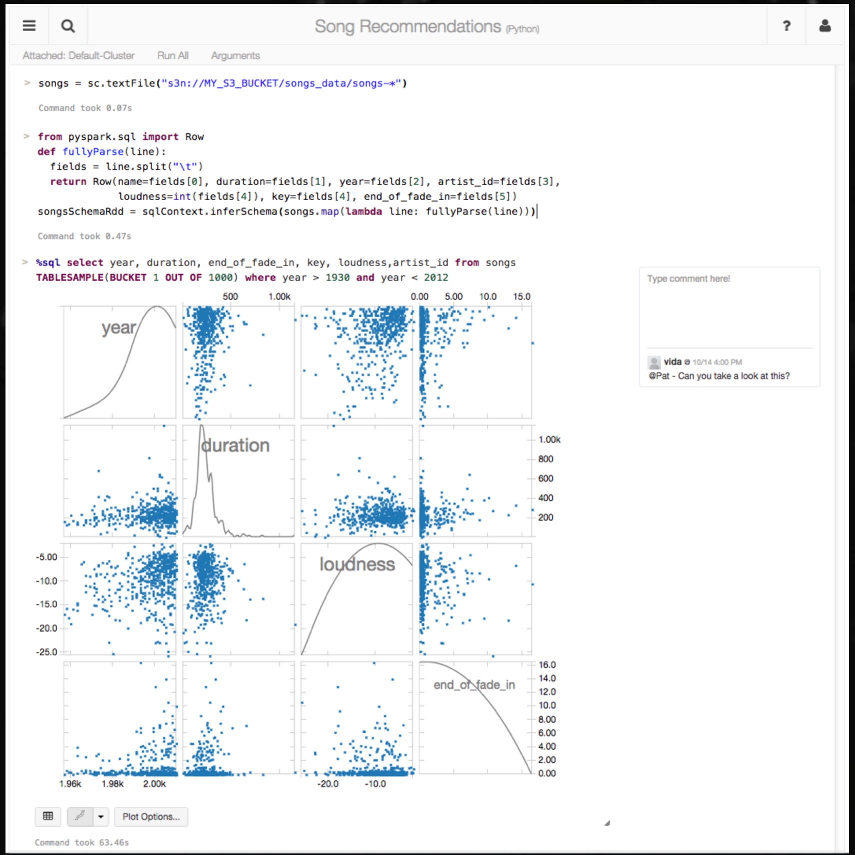
https://www.linkedin.com/profile/view?id=4367352

databricks

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
- ~55 employees
- We're hiring! (http://databricks.workable.com)
- Level 2/3 support partnerships with
  - 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

AGENDA

Before Lunch

- History of Big Data & Spark
- RDD fundamentals
- Databricks UI demo
- Lab: DevOps 101
- Transformations & Actions

After Lunch

- Transformations & Actions (continued)
- Lab: Transformations & Actions
- Dataframes
- Lab: Dataframes
- Spark UIs
- Resource Managers: Local & Stanalone
- Memory and Persistence
- Spark Streaming
- Lab: MISC labs

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 (sometime before lunch and after lunch)

INSTRUCTOR: BRIAN CLAPPER

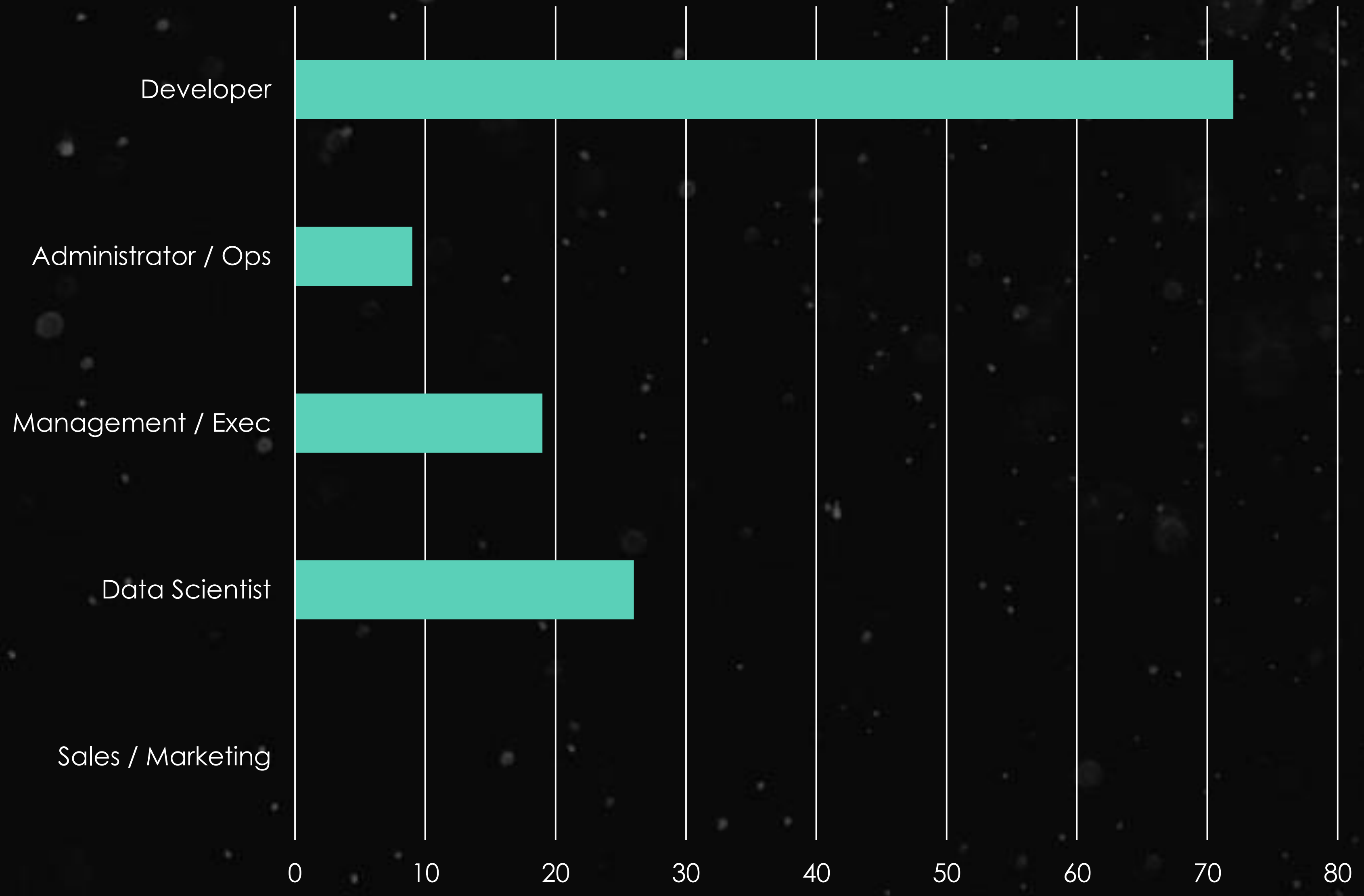
Homepage: http://www.ardentex.com/
LinkedIn: https://www.linkedin.com/in/bclapper

@brianclapper

- 30 years experience building & maintaining software systems
- Scala, Python, Ruby, Java, C, C#
- Founder of Philadelphia area Scala user group (PHASE)
- Spark instructor for Databricks

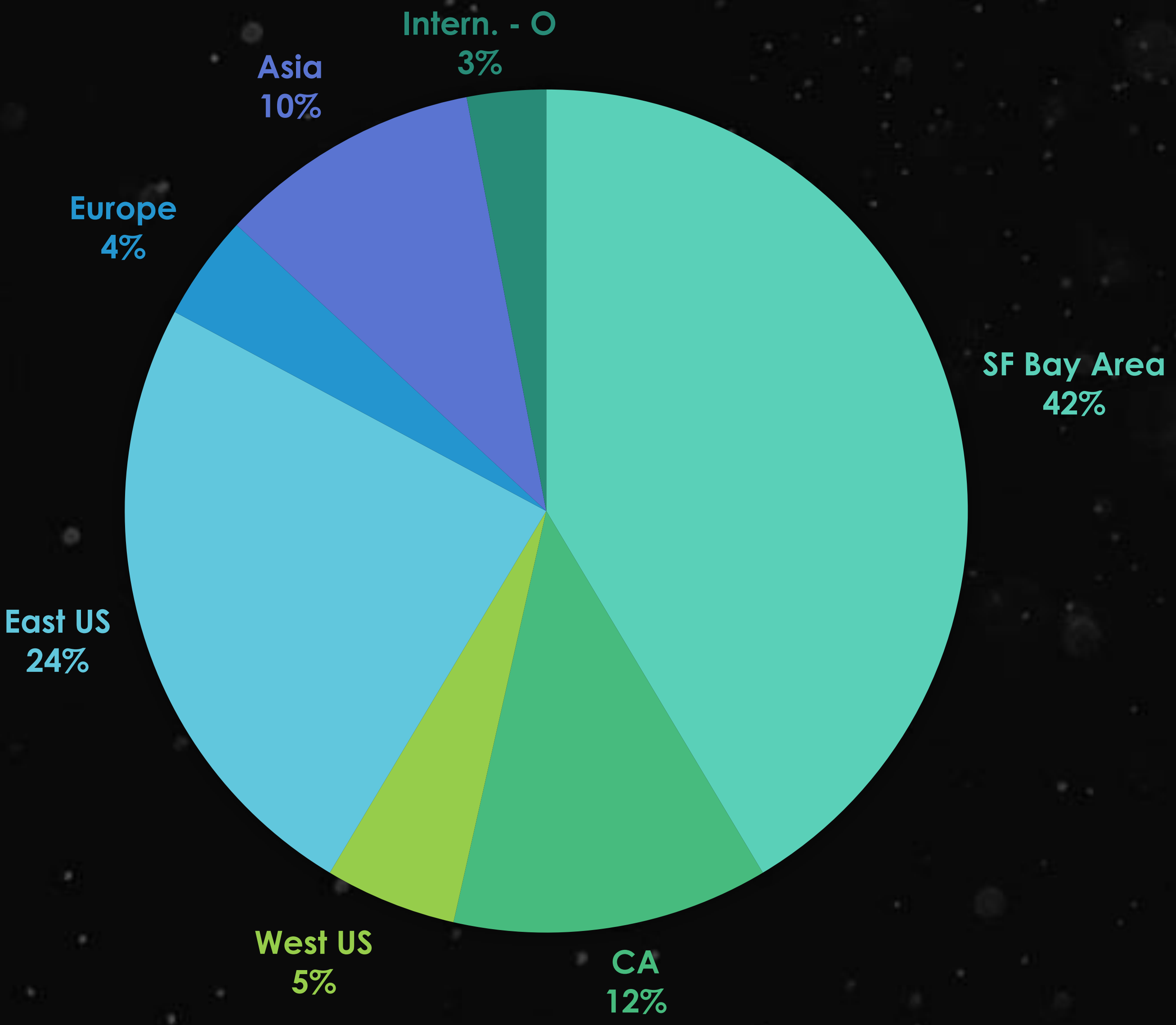
YOUR JOB?

Survey completed by
58 out of 115 students



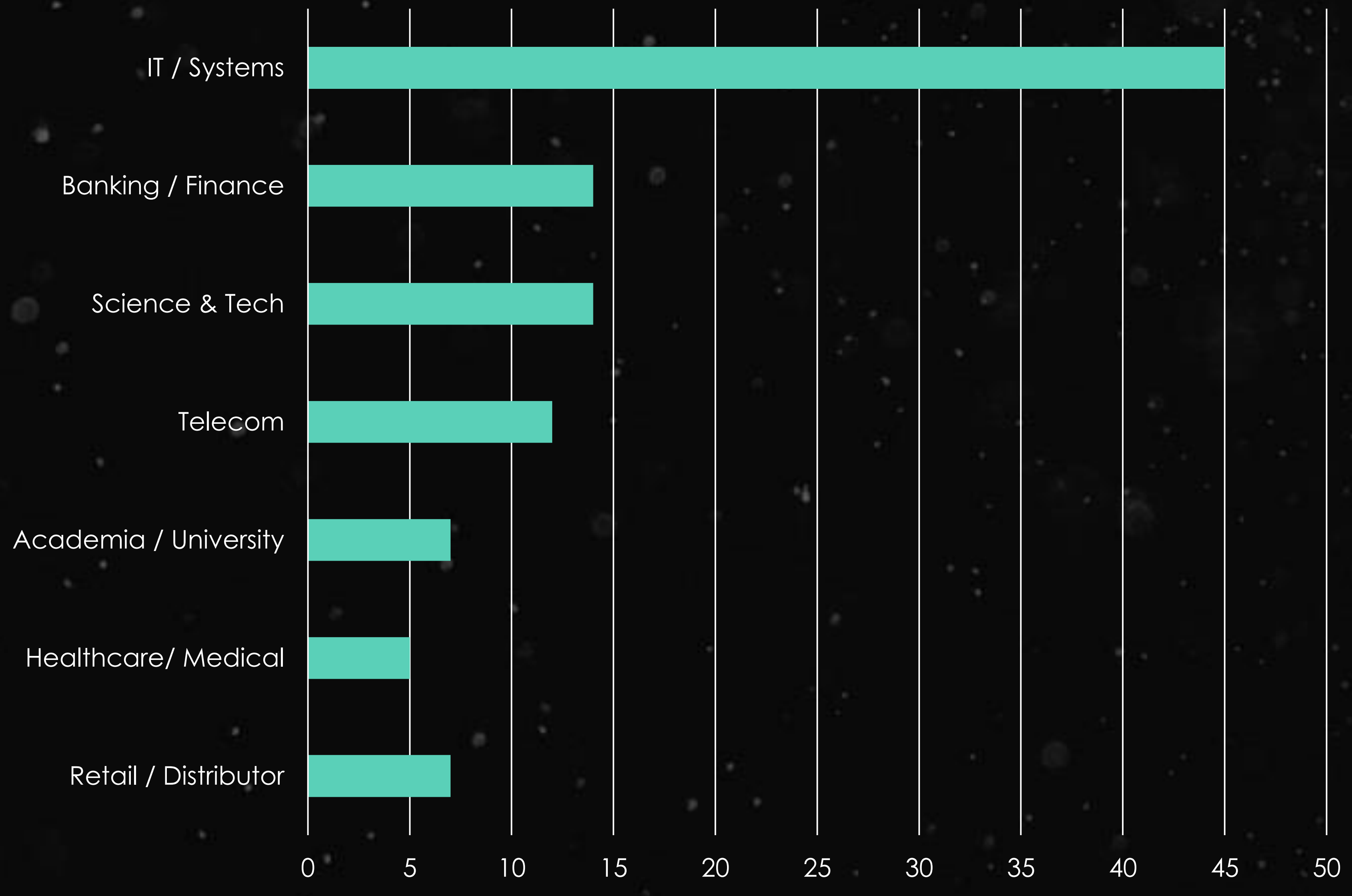
TRAVELED FROM?

Survey completed by
58 out of 115 students



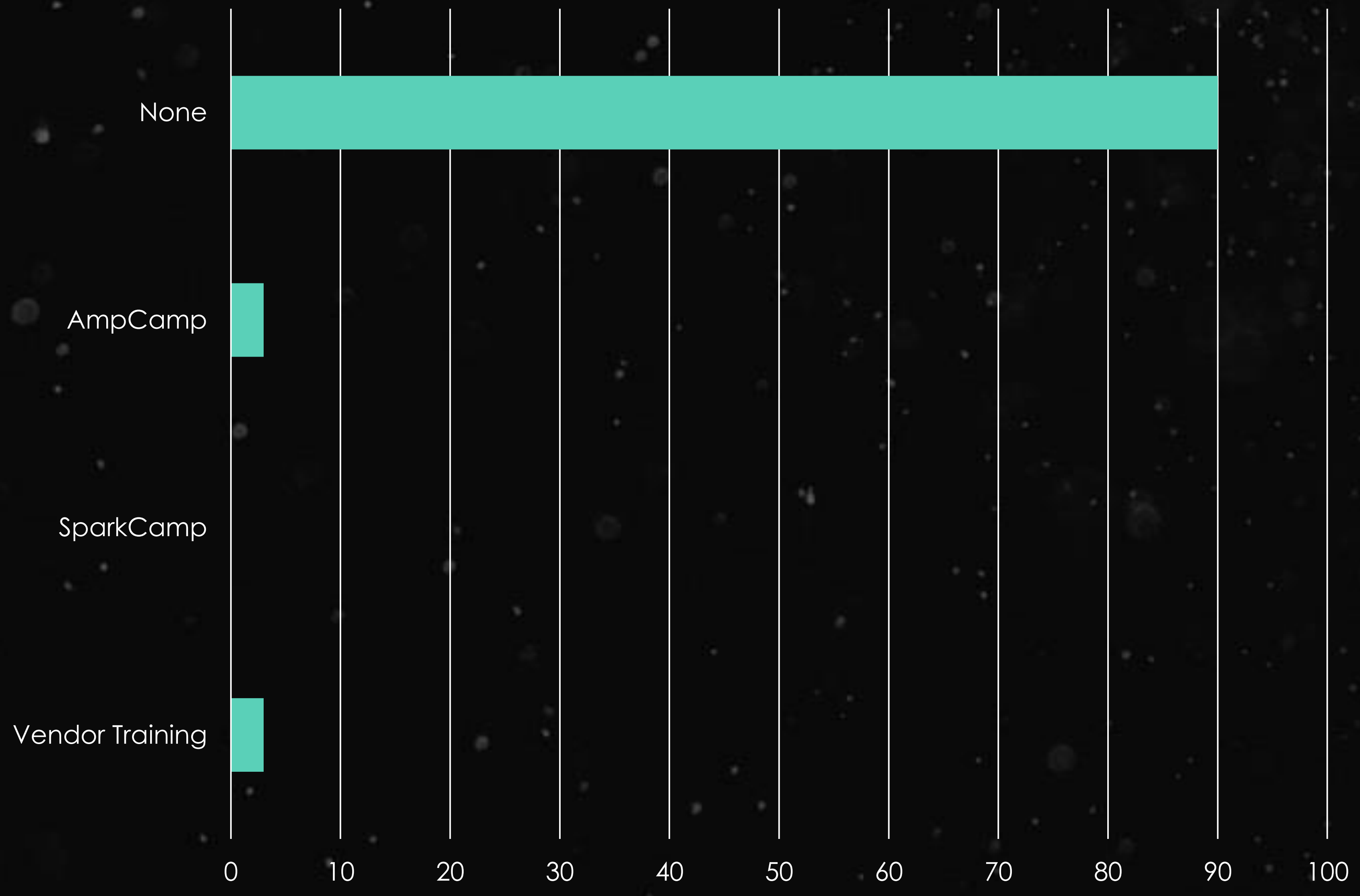
WHICH INDUSTRY?

Survey completed by
58 out of 115 students



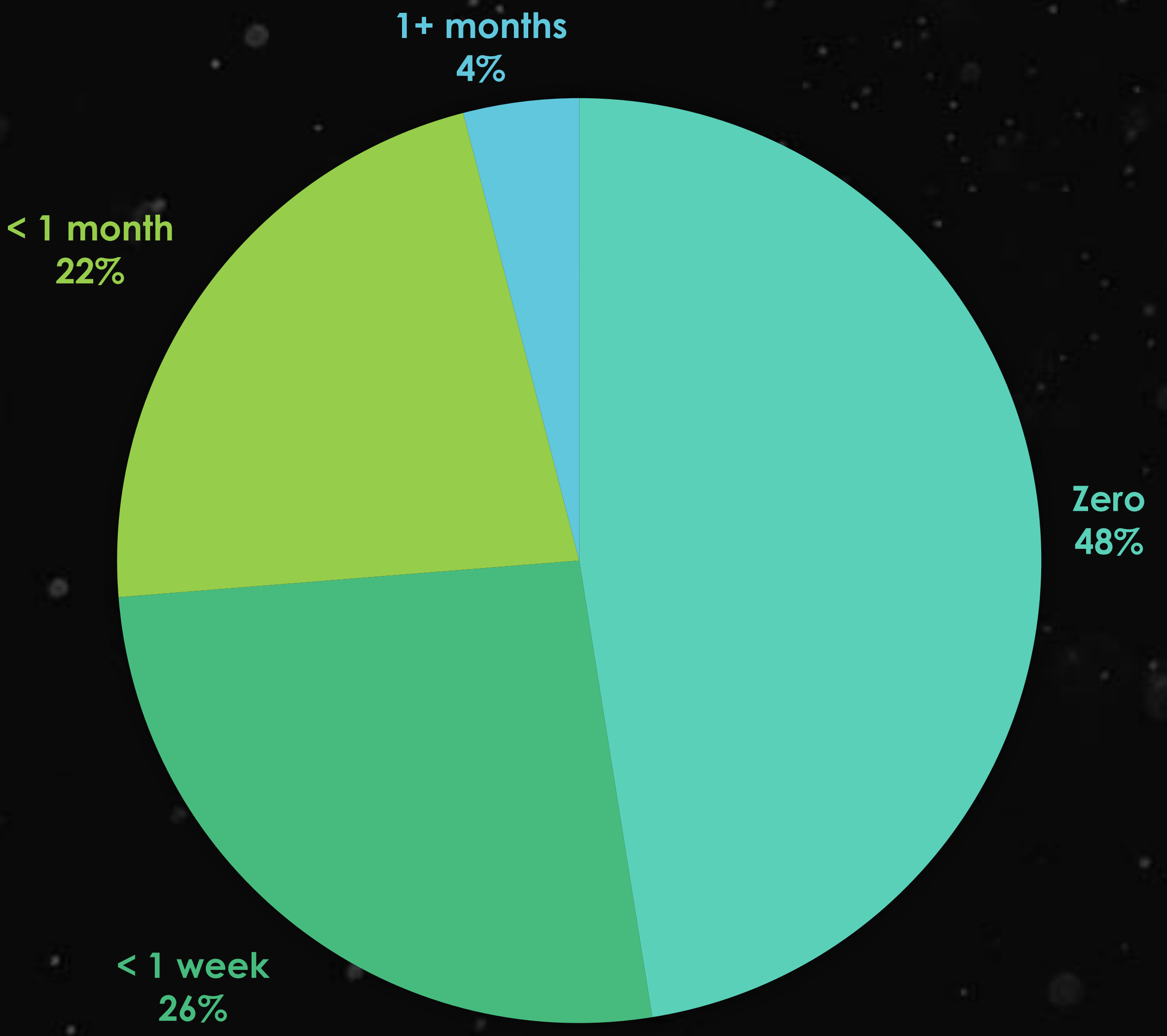
PRIOR SPARK TRAINING?

Survey completed by
58 out of 115 students



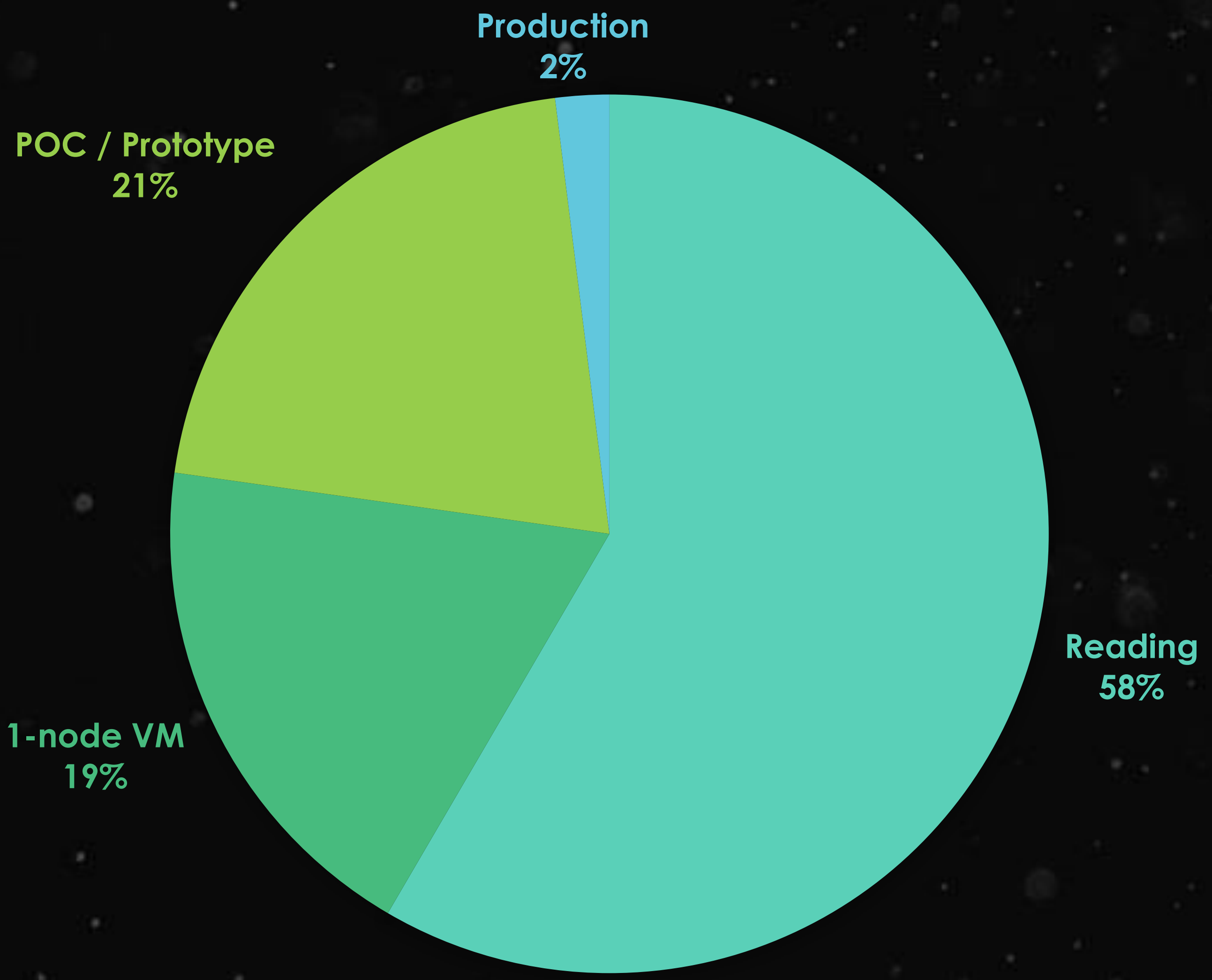
HANDS ON EXPERIENCE WITH SPARK?

Survey completed by
58 out of 115 students



SPARK USAGE LIFECYCLE?

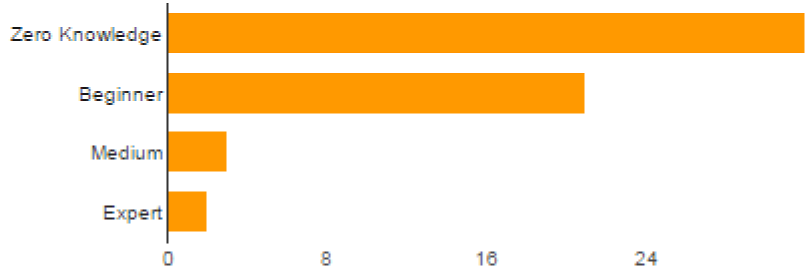
Survey completed by 58 out of 115 students



PROGRAMMING EXPERIENCE

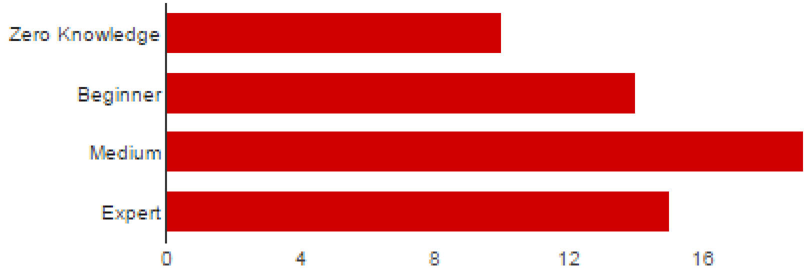
Survey completed by
58 out of 115 students

Scala [Which programming language API of Spark are you most comfortable in?]



Zero Knowledge	**32**	55.2%
Beginner	**21**	36.2%
Medium	**3**	5.2%
Expert	**2**	3.4%

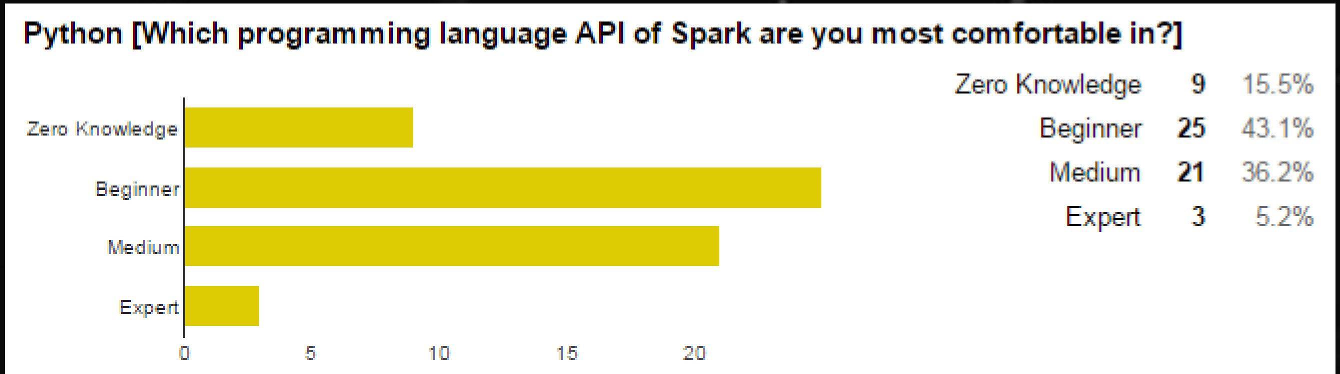
Java [Which programming language API of Spark are you most comfortable in?]

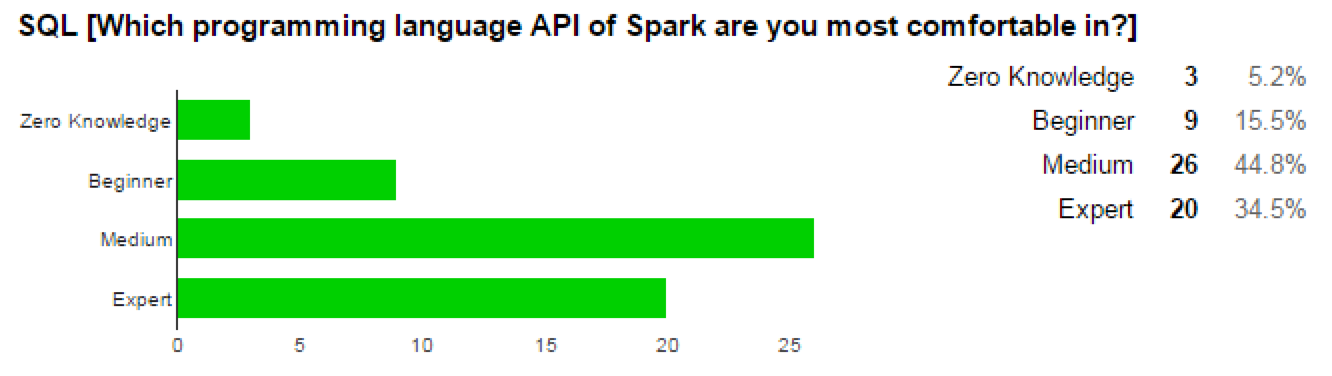


Zero Knowledge	**10**	17.2%
Beginner	**14**	24.1%
Medium	**19**	32.8%
Expert	**15**	25.9%

PROGRAMMING EXPERIENCE

Survey completed by
58 out of 115 students

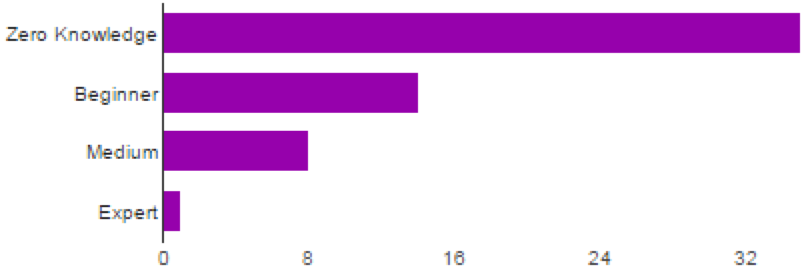




PROGRAMMING EXPERIENCE

Survey completed by
58 out of 115 students

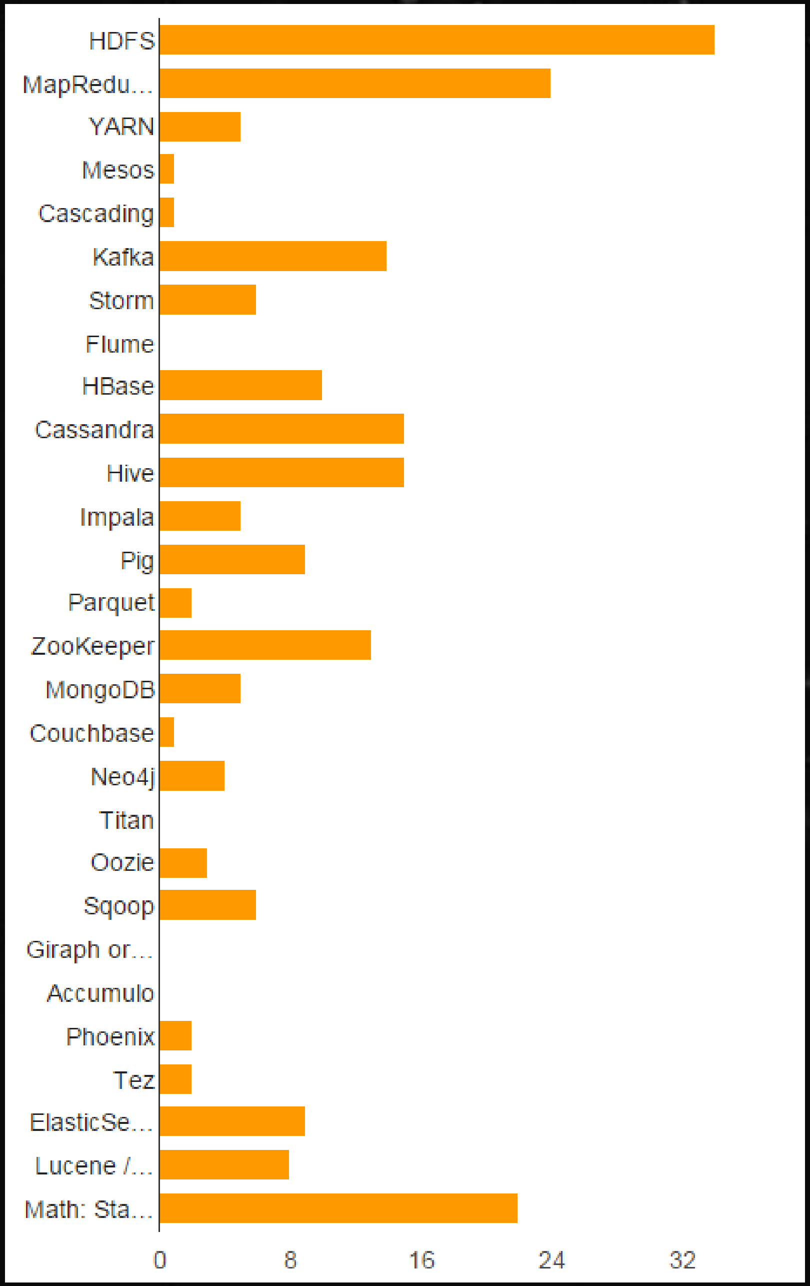
R [Which programming language API of Spark are you most comfortable in?]



Zero Knowledge	35	60.3%
Beginner	14	24.1%
Medium	8	13.8%
Expert	1	1.7%

Survey completed by
58 out of 115 students

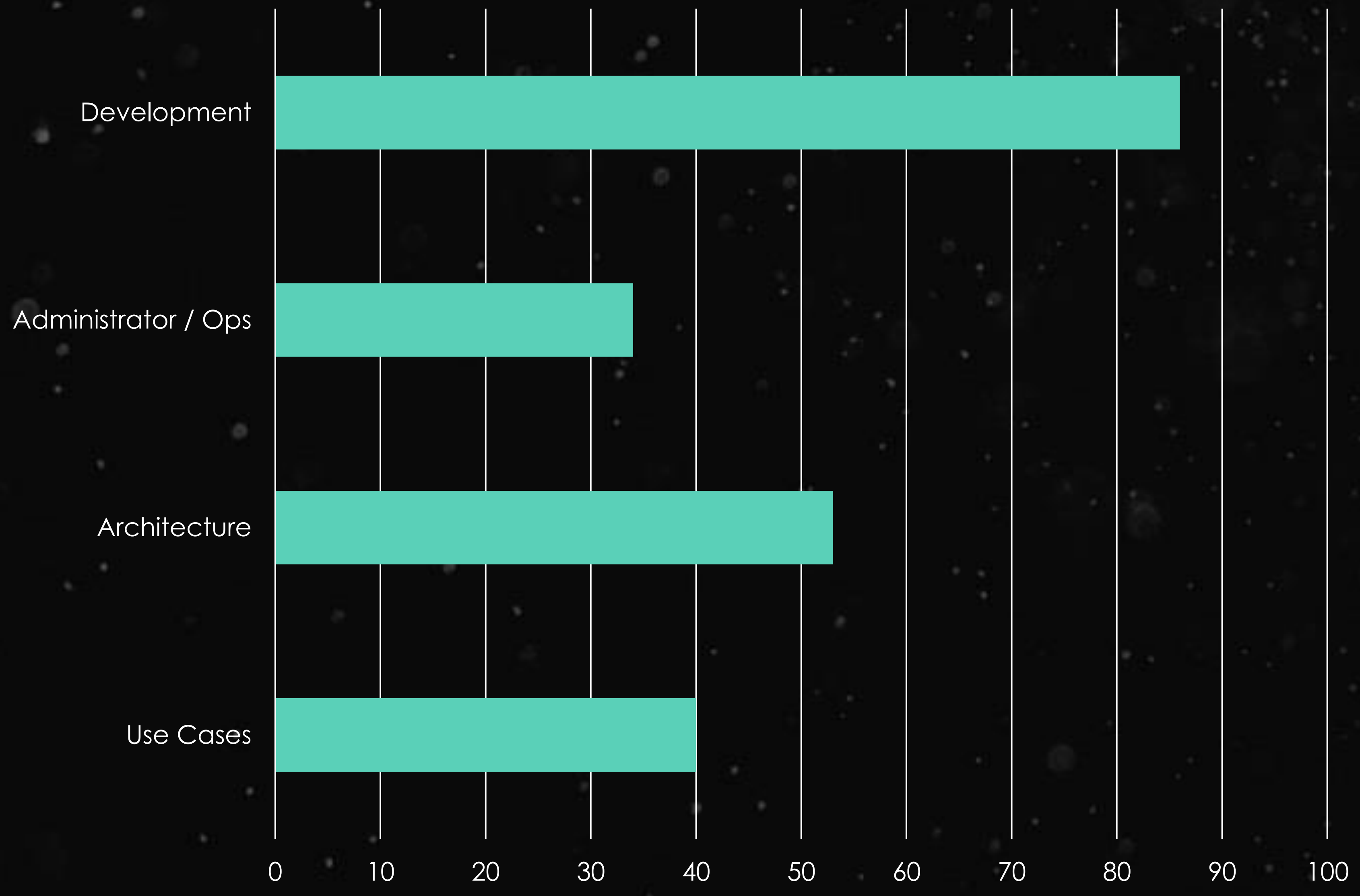
BIG DATA EXPERIENCE



HDFS	34	58.6%
MapReduce	24	41.4%
YARN	5	8.6%
Mesos	1	1.7%
Cascading	1	1.7%
Kafka	14	24.1%
Storm	6	10.3%
Flume	0	0%
HBase	10	17.2%
Cassandra	15	25.9%
Hive	15	25.9%
Impala	5	8.6%
Pig	9	15.5%
Parquet	2	3.4%
ZooKeeper	13	22.4%
MongoDB	5	8.6%
Couchbase	1	1.7%
Neo4j	4	6.9%
Titan	0	0%
Oozie	3	5.2%
Sqoop	6	10.3%
Giraph or Graphlab	0	0%
Accumulo	0	0%
Phoenix	2	3.4%
Tez	2	3.4%
ElasticSearch	9	15.5%
Lucene / Solr	8	13.8%
ıs, Matrix math, etc	22	37.9%

FOCUS OF CLASS?

Survey completed by
58 out of 115 students



STORAGE VS PROCESSING WARS

NoSQL battles
(then)

Relational vs NoSQL

HBase vs Cassanrdra

Redis vs Memcached vs Riak

MongoDB vs CouchDB vs Couchbase

Neo4j vs Titan vs Giraph vs OrientDB

Solr vs Elasticsearch

Compute battles
(now)

MapReduce vs Spark

Spark Streaming vs Storm

Hive vs Spark SQL vs Impala

Mahout vs MLlib vs H20

STORAGE VS PROCESSING WARS

NoSQL battles
(then)

Relational vs NoSQL

HBase vs Cassanrdra

Redis vs Memcached vs Riak

MongoDB vs CouchDB vs Couchbase

Neo4j vs Titan vs Giraph vs OrientDB

Solr vs Elasticsearch

Compute battles
(now)

MapReduce vs Spark

Spark Streaming vs Storm

Hive vs Spark SQL vs Impala

Mahout vs MLlib vs H20

NOSQL POPULARITY WINNERS

DB-ENGINES

Key -> Value	Key -> Doc	Column Family	Graph	Search
Redis - 95	MongoDB - 279	Cassandra - 109	Neo4j - 30	Solr - 81
Memcached - 33	CouchDB - 28	HBase - 62	OrientDB - 4	Elasticsearch - 70
DynamoDB - 16	Couchbase - 24		Titan – 3	Splunk – 41
Riak - 13	DynamoDB – 15		Giraph - 1	
	MarkLogic - 11			

(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

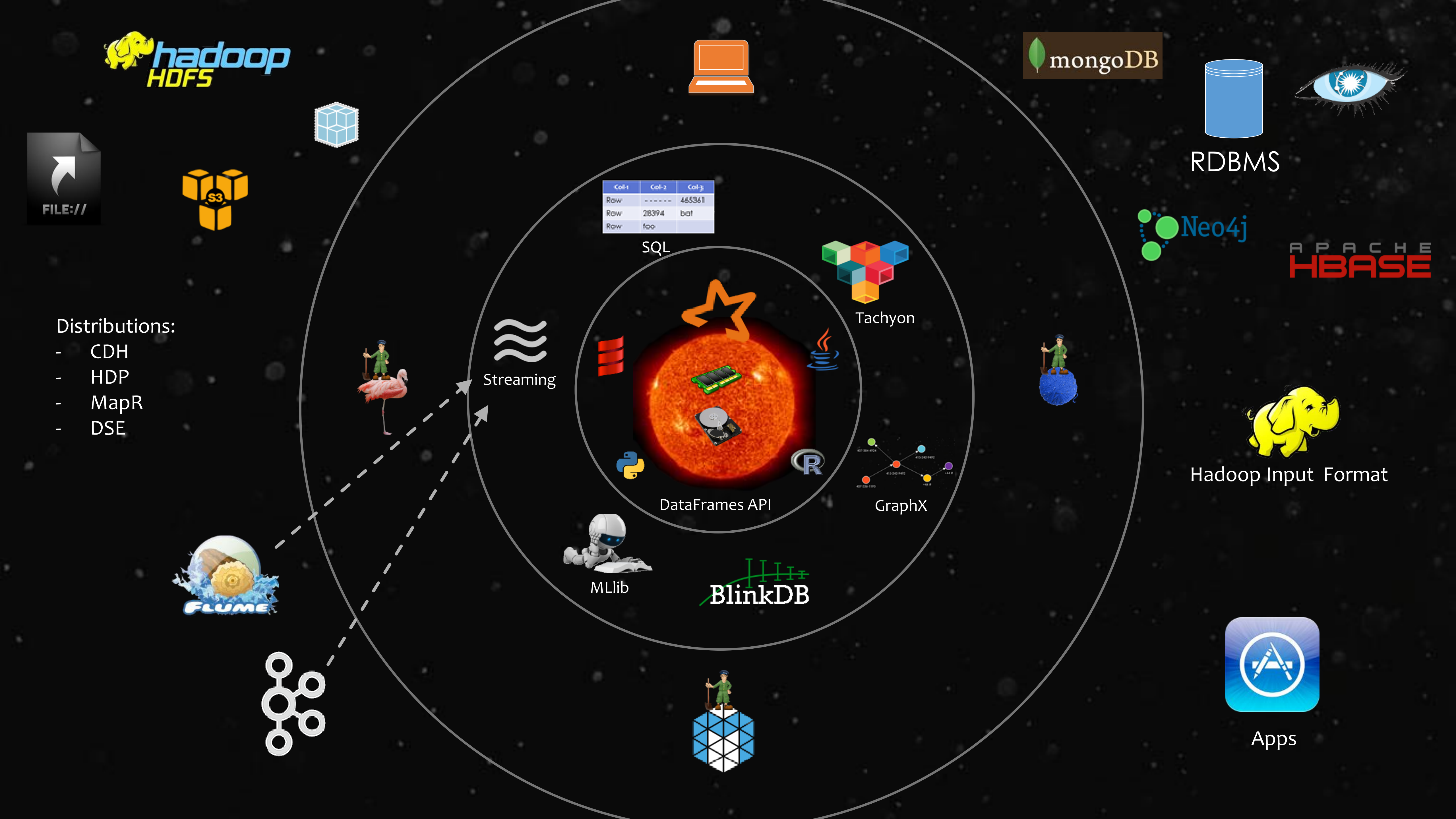
General Unified Engine

Spark

Scheduling

Monitoring

Distributing


hadoop HDFS
FILE://
S3
mongoDB
RDBMS
Neo4j
APACHE HBASE
Distributions:
- CDH
- HDP
- MapR
- DSE
Col-1 Col-2 Col-3
SQL
Tachyon
Streaming
DataFrames API
GraphX
MLlib
BlinkDB
Hadoop Input Format
FLUME
Apps

Rick Richardson
@eigenrick

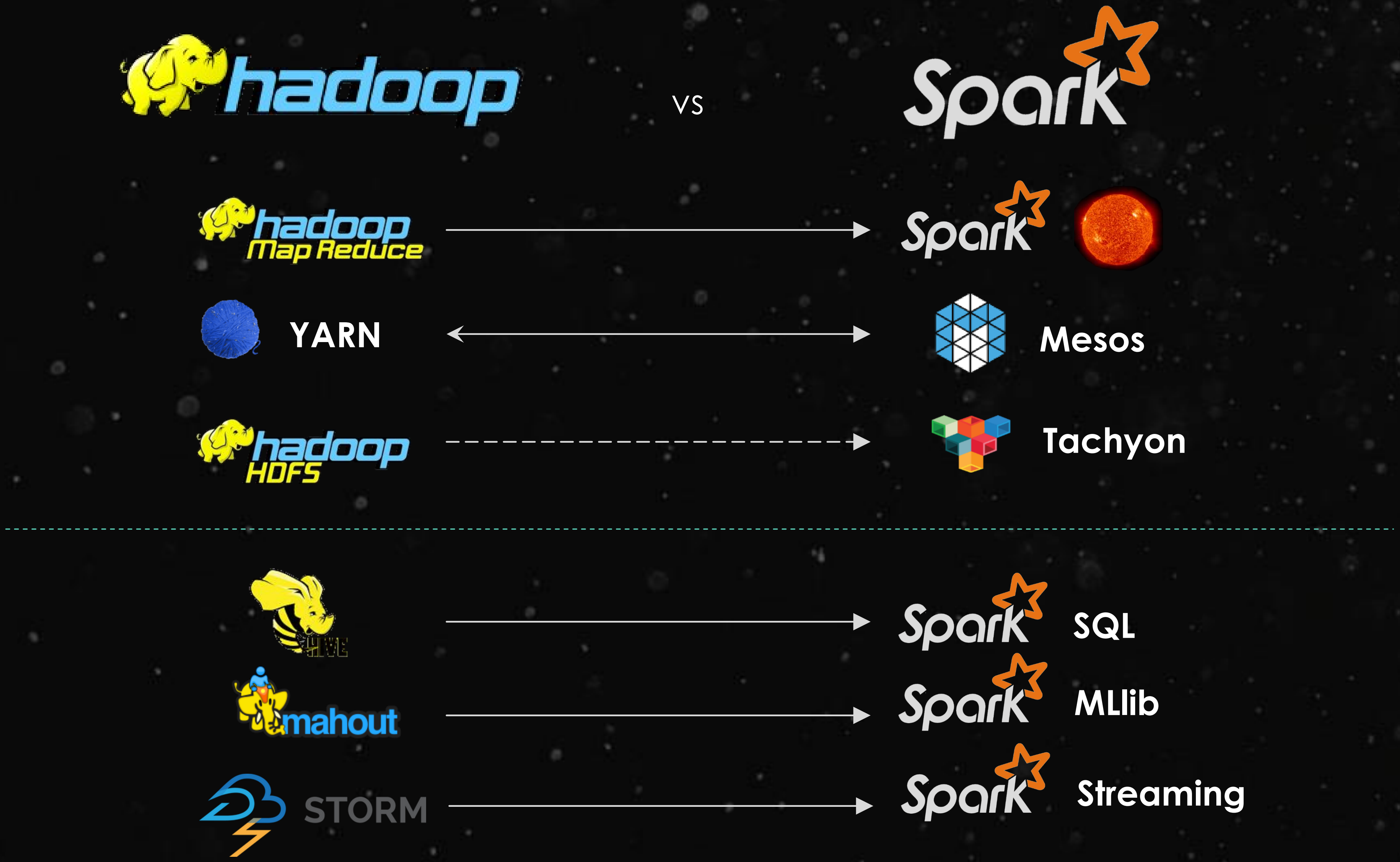
Follow

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

RETWEETS 11 FAVORITES 17

11:06 AM - 30 Jun 2014

- Developers from 50+ companies
- 400+ developers
- Apache Committers from 16+ organizations


hadoop
VS
Spark
hadoop Map Reduce
Spark
YARN
Mesos
hadoop HDFS
Tachyon
HIVE
Spark SQL
mahout
Spark MLlib
STORM
Spark Streaming


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

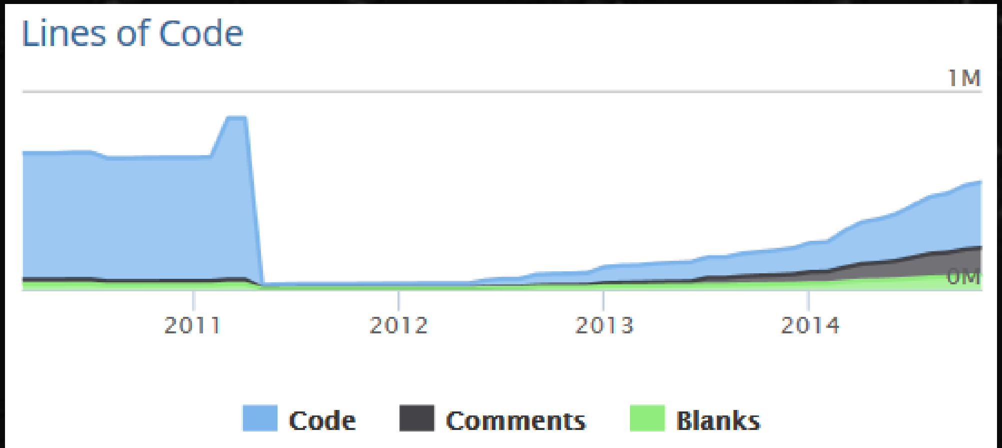
In a Nutshell, Apache Spark...

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

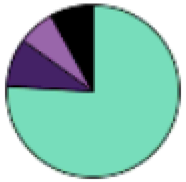
... is mostly written in Scala
with a well-commented source code

... 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

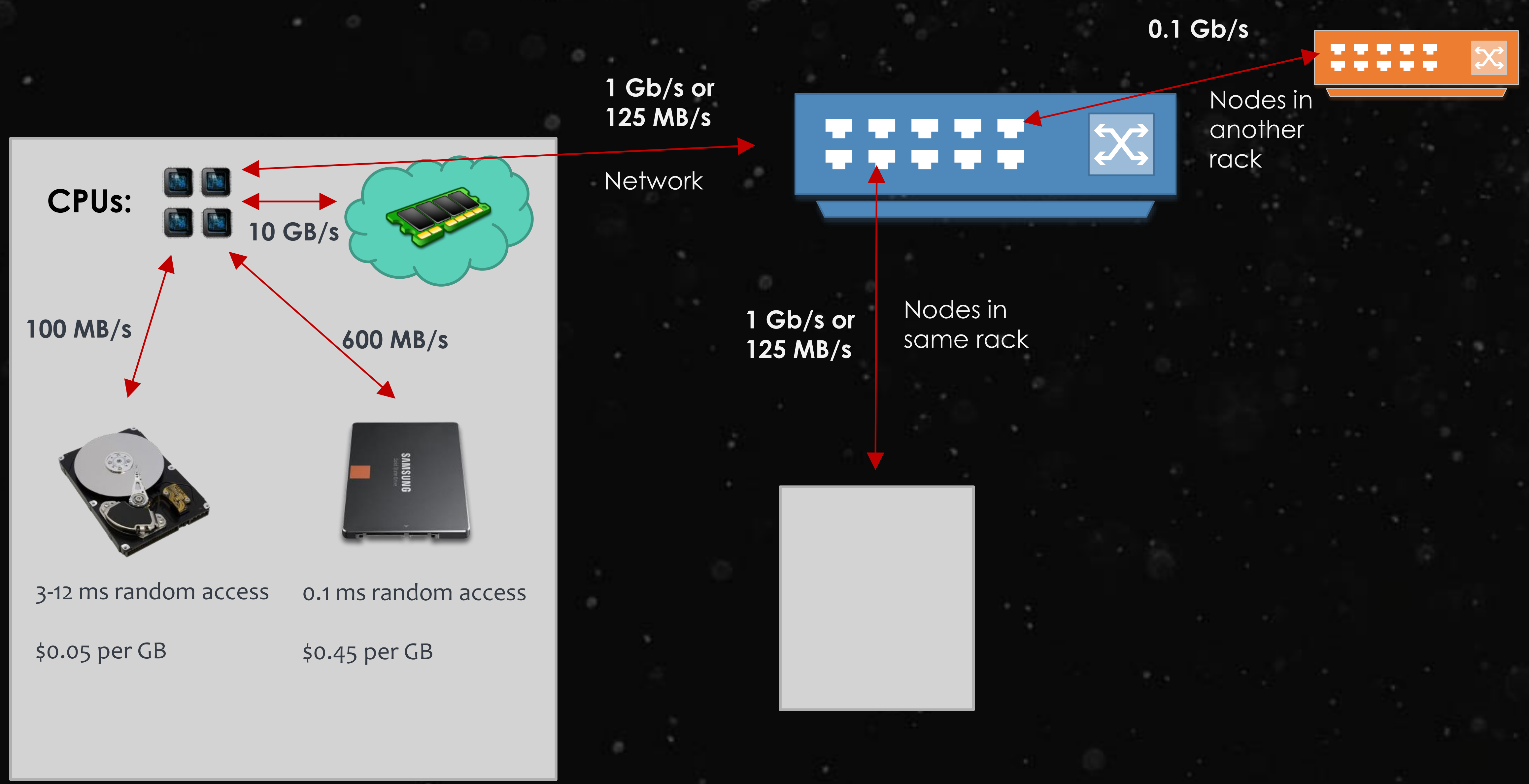
DISTRIBUTORS

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


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

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

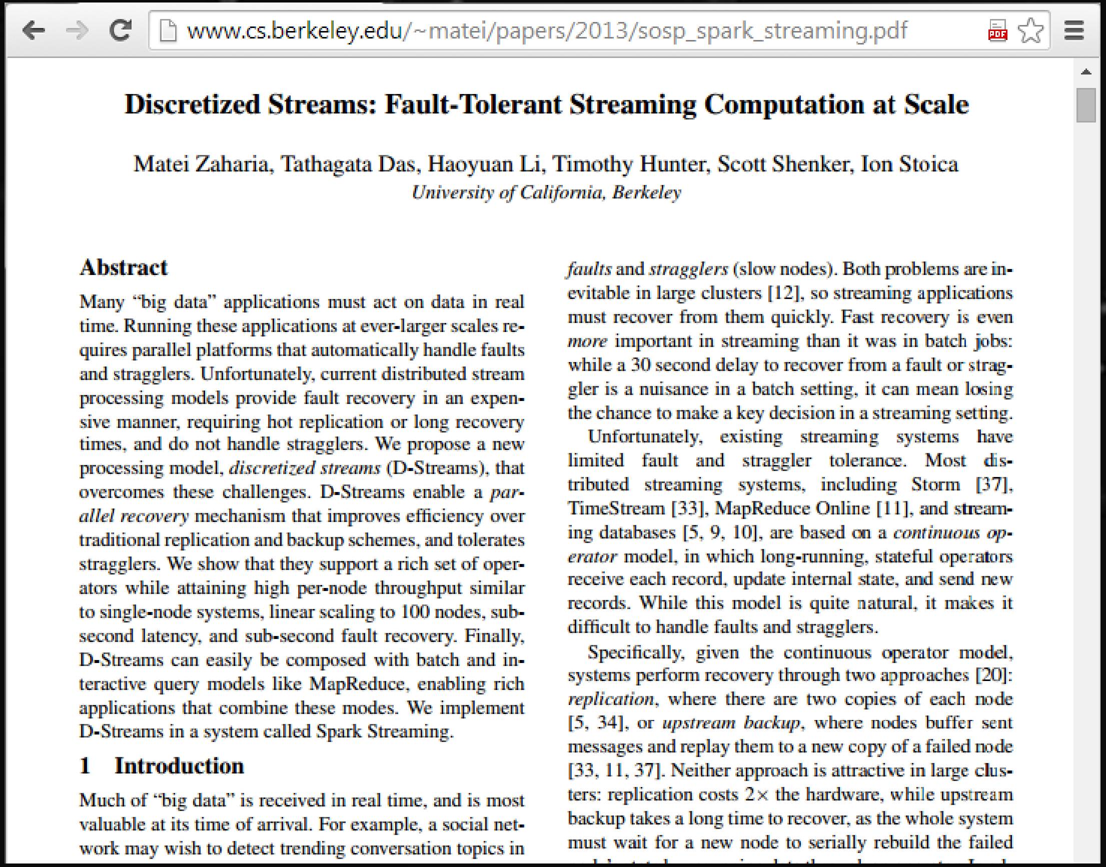
- Cited 400+ times!

April 2012

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

Spark STREAMING

Analyze real time streams of data in ½ second intervals


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.

Spark SQL: Relational Data Processing in Spark

Michael Armbrust†, Reynold S. Xin†, Cheng Lian†, Yin Huai†, Davies Liu†, Joseph K. Bradley†, Xiangrui Meng†, Tomer Kaftan‡, Michael J. Franklin†‡, Ali Ghodsi†, Matei Zaharia†*

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

ABSTRACT

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

Categories and Subject Descriptors

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

Keywords

Databases; Data Warehouse; Machine Learning; Spark; Hadoop

1 Introduction

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

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

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

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

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

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

Spark GRAPHX

Analyze networks of nodes and edges using graph processing

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

SQL queries with Bounded Errors and Bounded Response Times

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

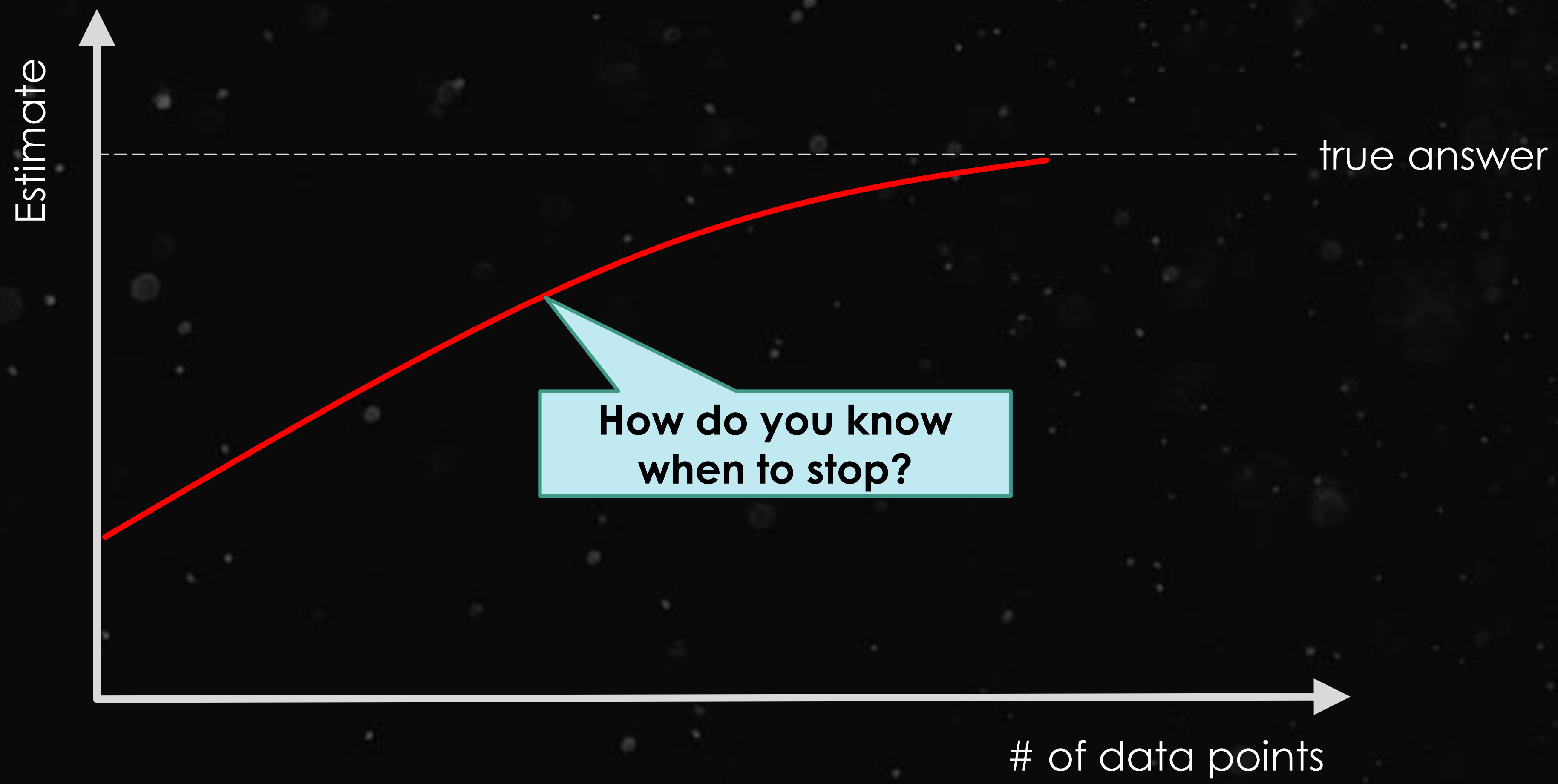
```
SELECT avg(sessionTime)
FROM Table
WHERE city='San Francisco'
WITHIN 2 SECONDS
```

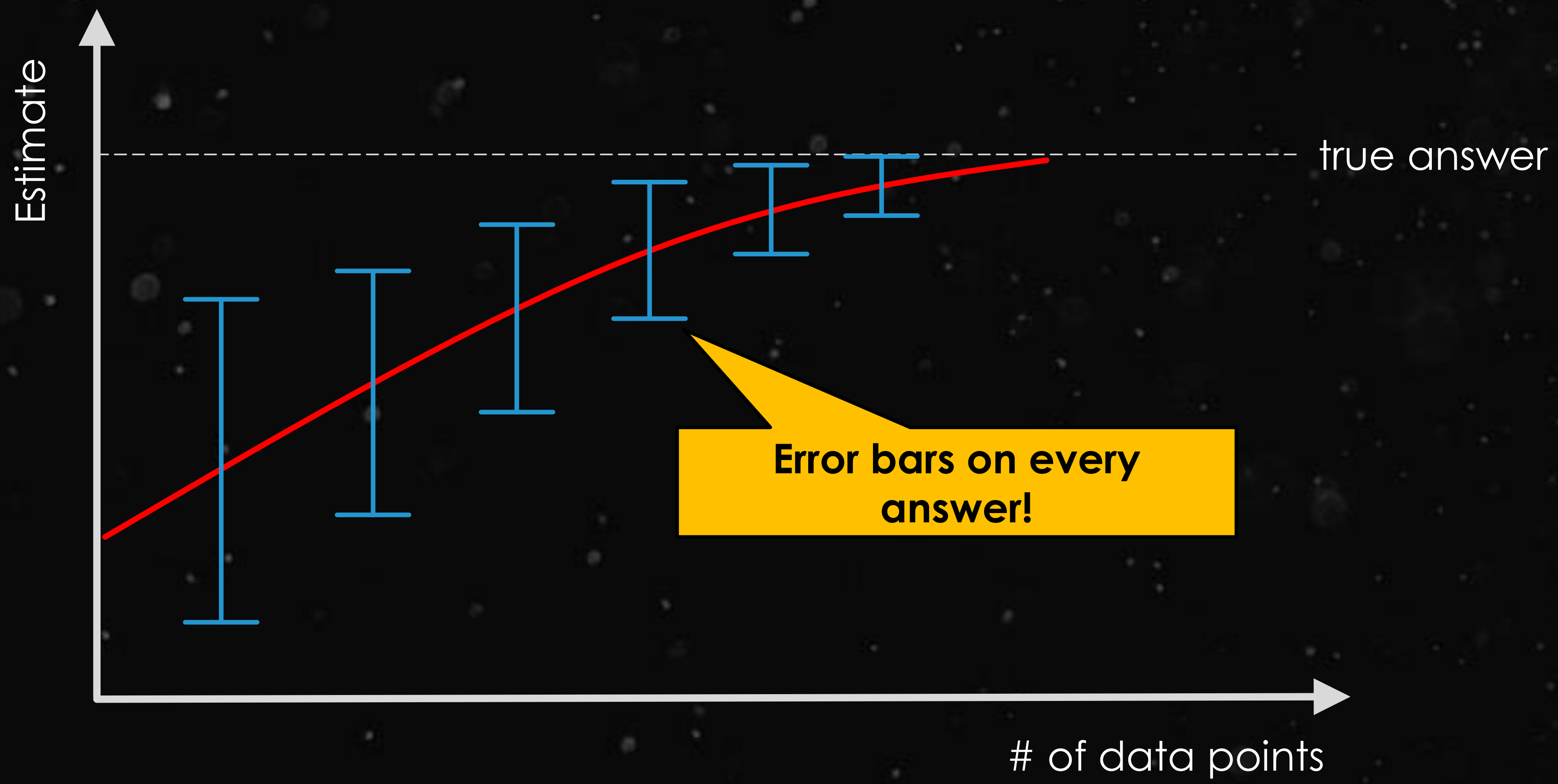
Queries with Time Bounds

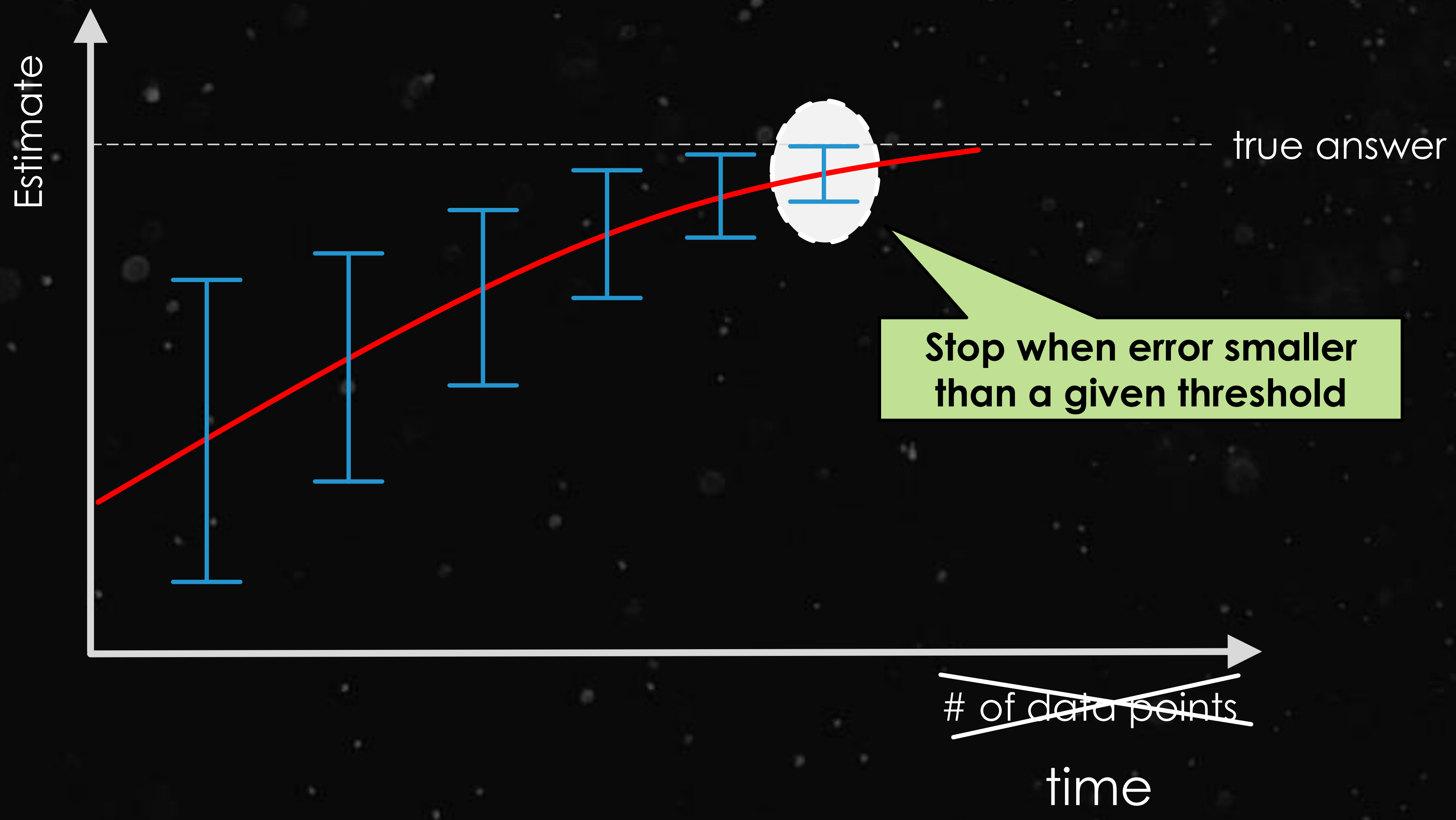
```
SELECT avg(sessionTime)
FROM Table
WHERE city='San Francisco'
ERROR 0.1 CONFIDENCE 95.0%
```

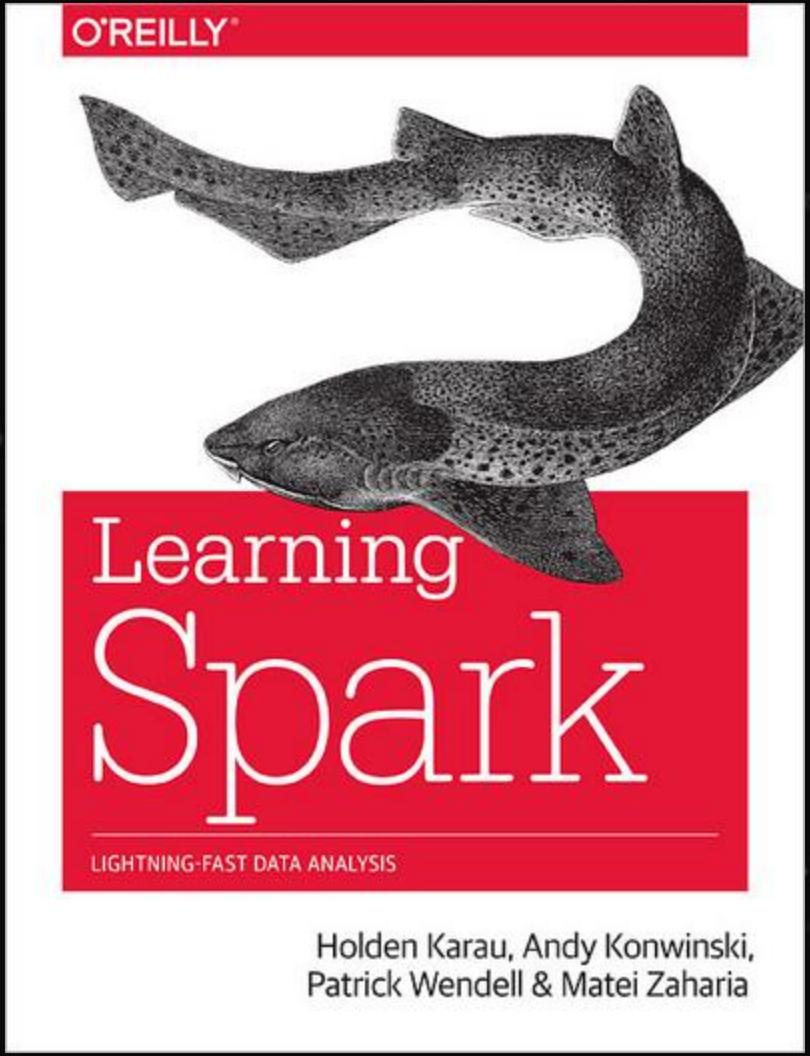
Queries with Error Bounds

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


Estimate
true answer
How do you know
when to stop?
# of data points


Estimate
true answer
Error bars on every answer!
# of data points


Estimate
true answer
Stop when error smaller
than a given threshold
~~# of data points~~
time



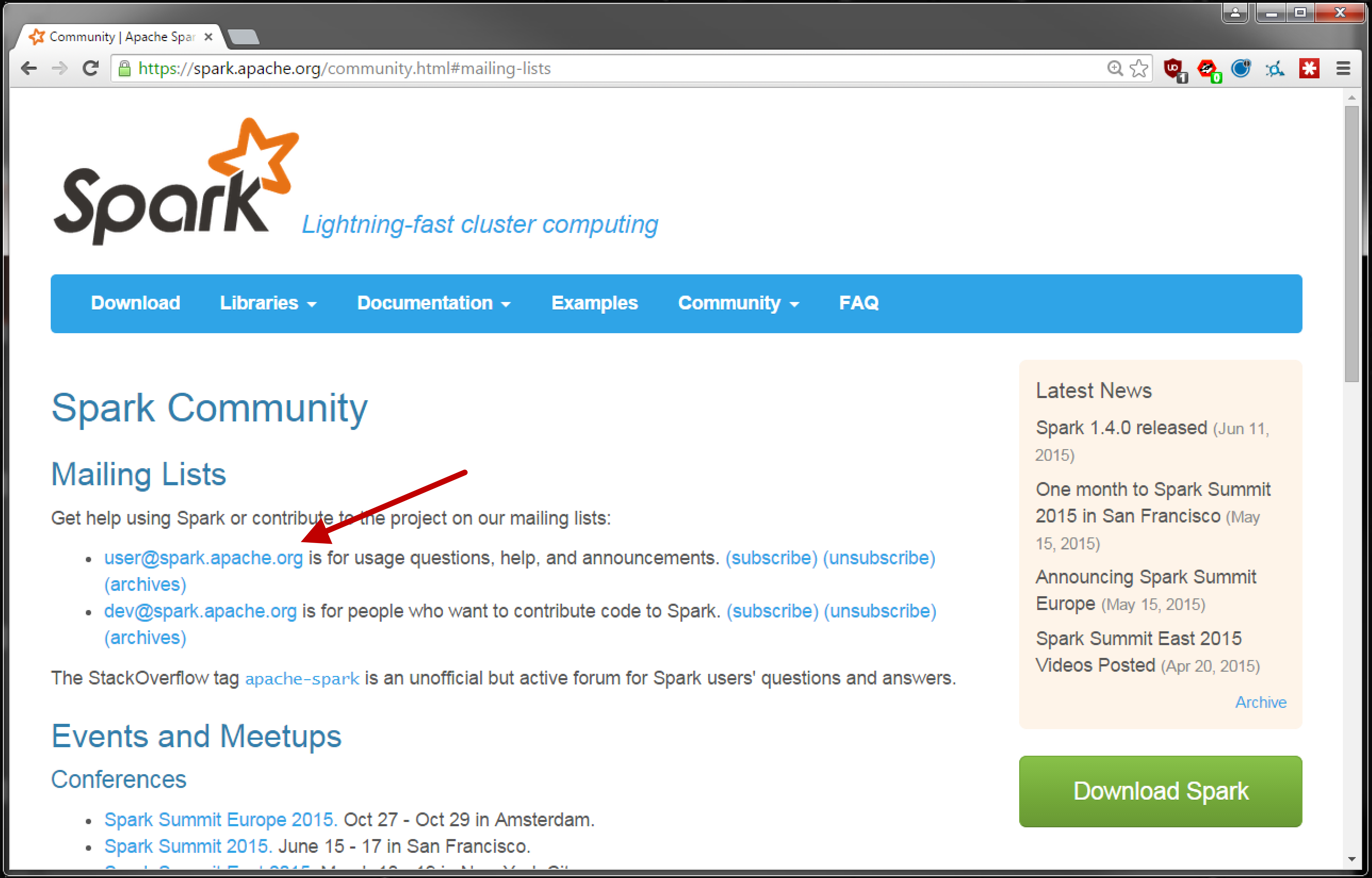
http://shop.oreilly.com/product/0636920028512.do

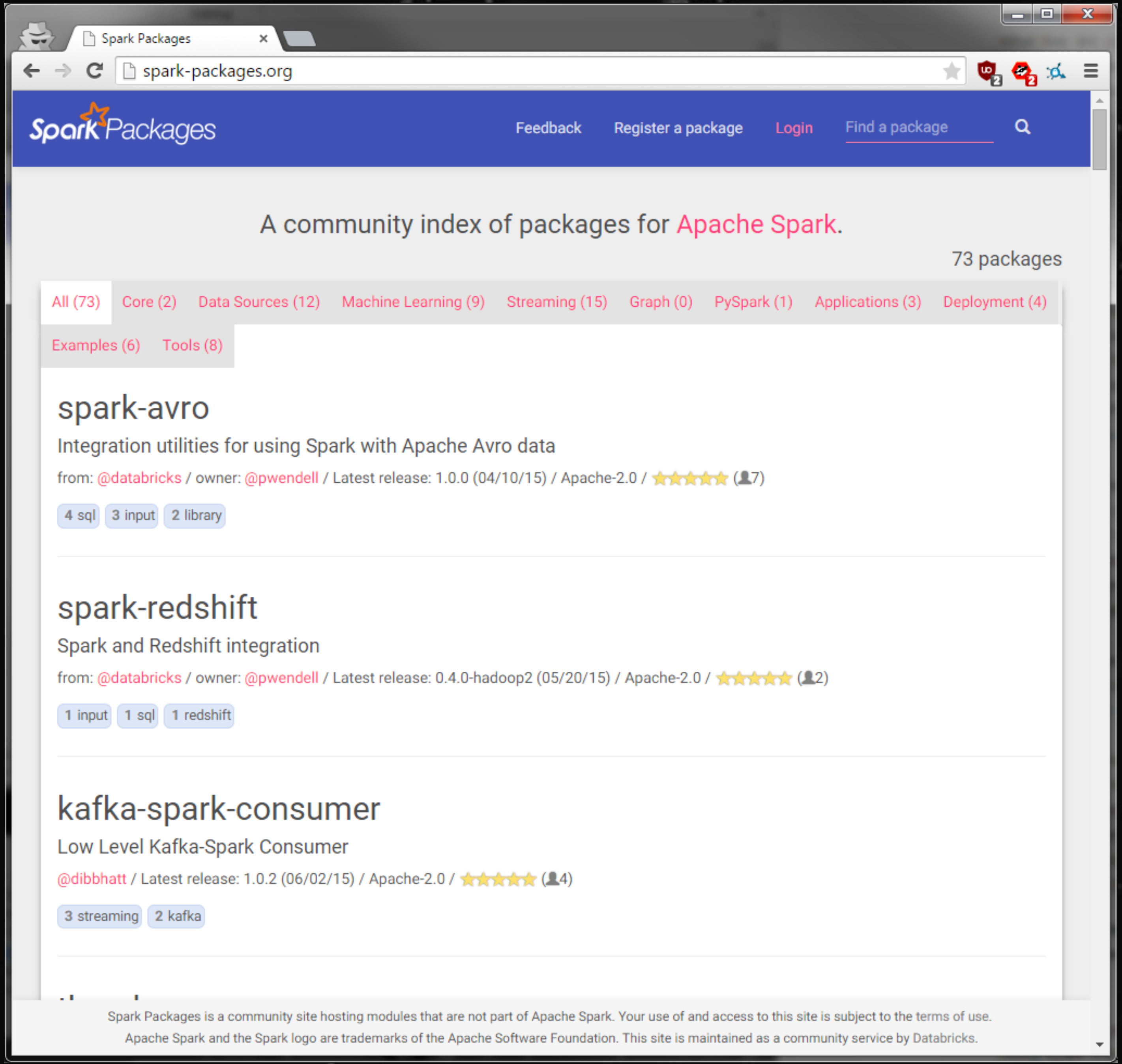
eBook: $33.99 PDF, ePub, Mobi, DAISY

Print: $39.99 Shipping now!

$30 @ Amazon:

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


Community | Apache Spar
https://spark.apache.org/community.html#mailing-lists
Spark
Lightning-fast cluster computing
Download
Libraries
Documentation
Examples
Community
FAQ
Spark Community
Mailing Lists
Get help using Spark or contribute 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)
The StackOverflow tag apache-spark is an unofficial but active forum for Spark users' questions and answers.
Events and Meetups
Conferences
Spark Summit Europe 2015. Oct 27 - Oct 29 in Amsterdam.
Spark Summit 2015. June 15 - 17 in San Francisco.
Latest News
Spark 1.4.0 released (Jun 11, 2015)
One month to Spark Summit 2015 in San Francisco (May 15, 2015)
Announcing Spark Summit Europe (May 15, 2015)
Spark Summit East 2015 Videos Posted (Apr 20, 2015)
Archive
Download Spark



Spark Packages

spark-packages.org

Spark Packages

Feedback Register a package Login Find a package

A community index of packages for Apache Spark.

73 packages

All (73) Core (2) Data Sources (12) Machine Learning (9) Streaming (15) Graph (0) PySpark (1) Applications (3) Deployment (4) Examples (6) Tools (8)

spark-avro

Integration utilities for using Spark with Apache Avro data

from: @databricks / owner: @pwendell / Latest release: 1.0.0 (04/10/15) / Apache-2.0 / (7)

4 sql 3 input 2 library

spark-redshift

Spark and Redshift integration

from: @databricks / owner: @pwendell / Latest release: 0.4.0-hadoop2 (05/20/15) / Apache-2.0 / (2)

1 input 1 sql 1 redshift

kafka-spark-consumer

Low Level Kafka-Spark Consumer

@dibbhatt / Latest release: 1.0.2 (06/02/15) / Apache-2.0 / (4)

3 streaming 2 kafka

Spark Packages is a community site hosting modules that are not part of Apache Spark. Your use of and access to this site is subject to the terms of use.
Apache Spark and the Spark logo are trademarks of the Apache Software Foundation. This site is maintained as a community service by 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.

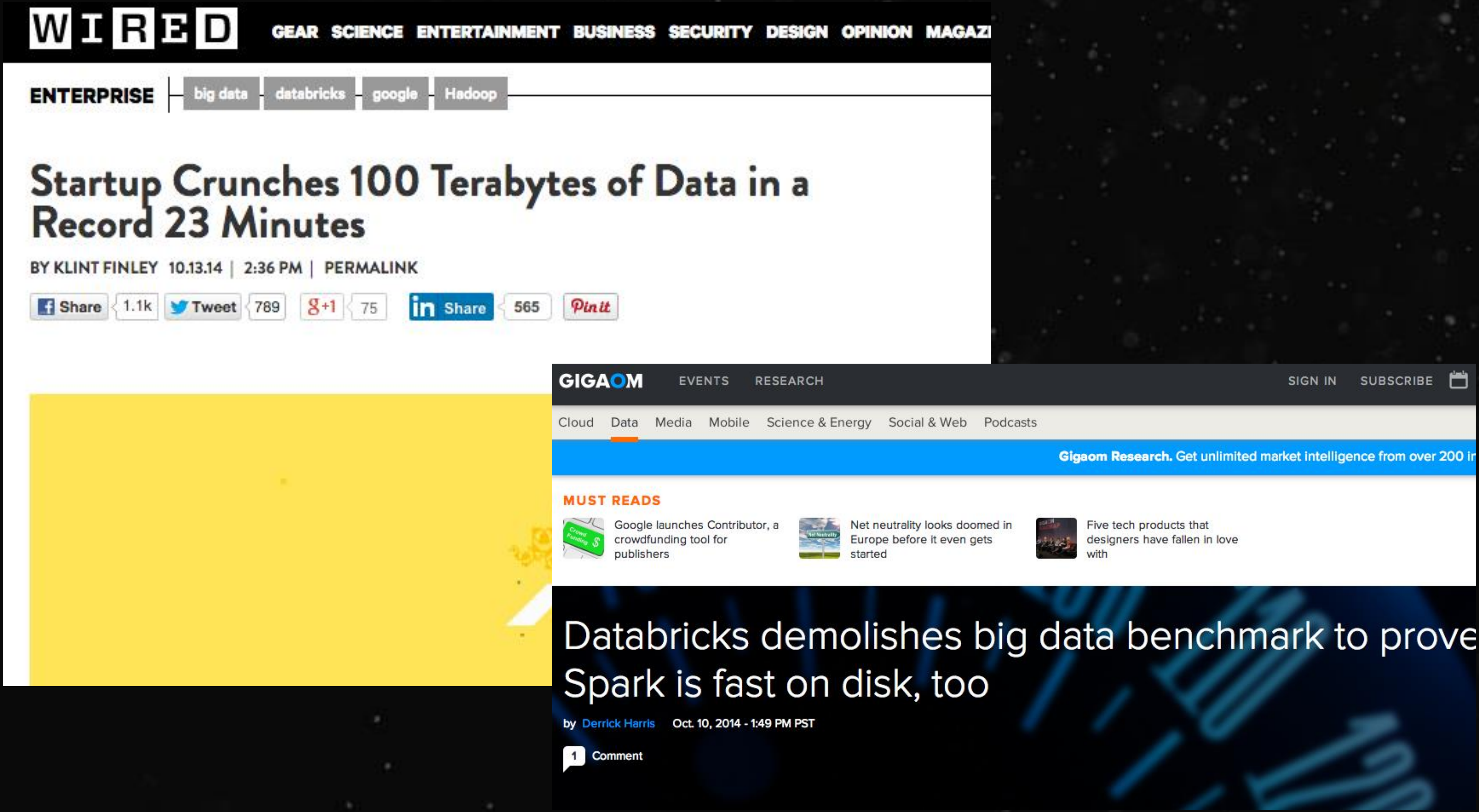
All the sorting took place on disk (HDFS) without using Spark's in-memory cache!

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 ir

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

Reference

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)

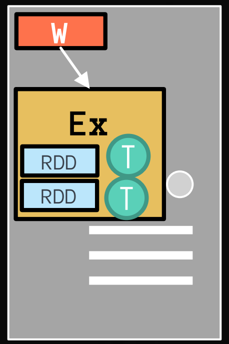
Clever Application level Techniques:

- GC and cache friendly memory layout
- Pipelining

Reference

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

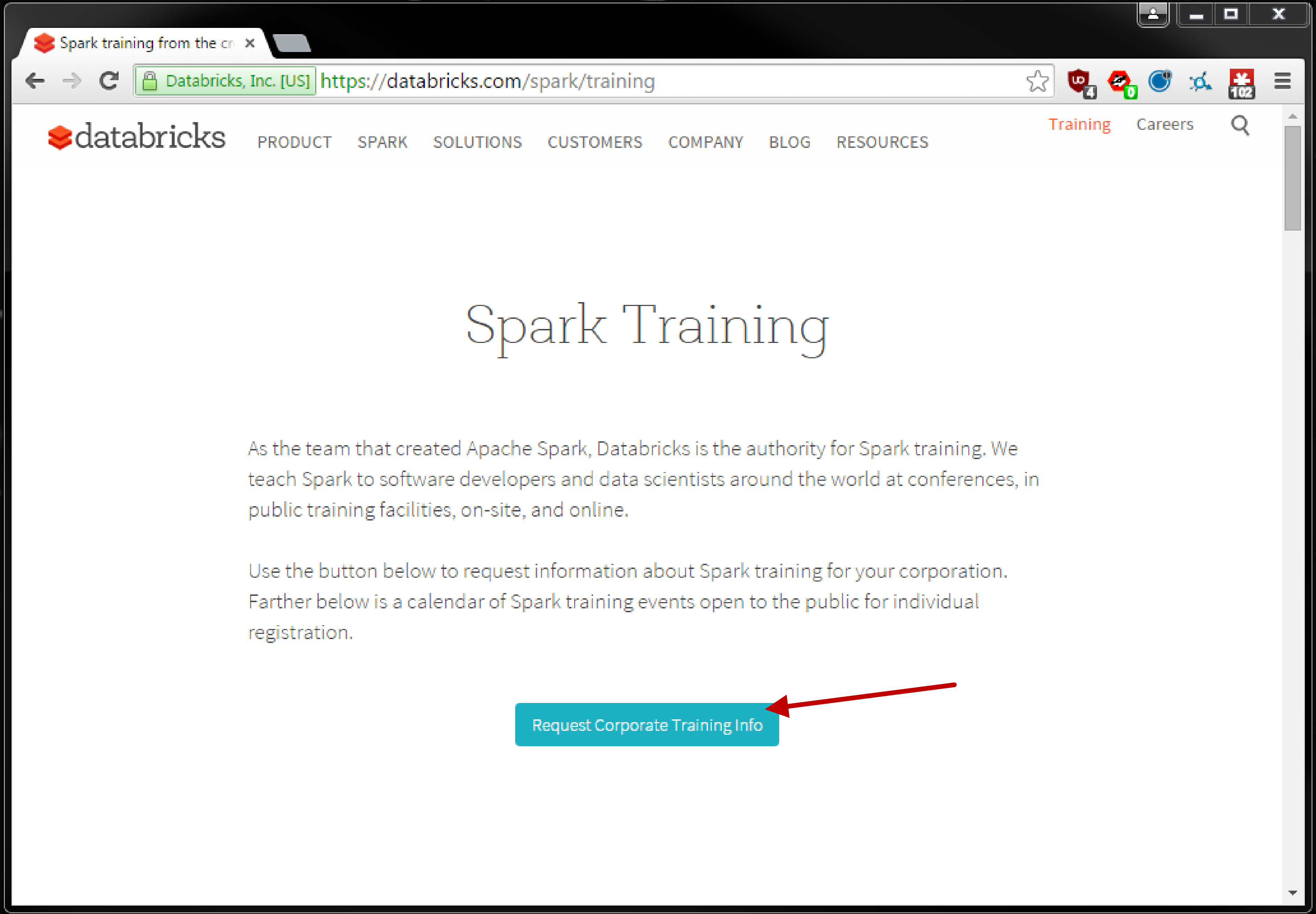


EC2: i2.8xlarge

(206 workers)

- 32 slots per machine
- 6,592 slots total

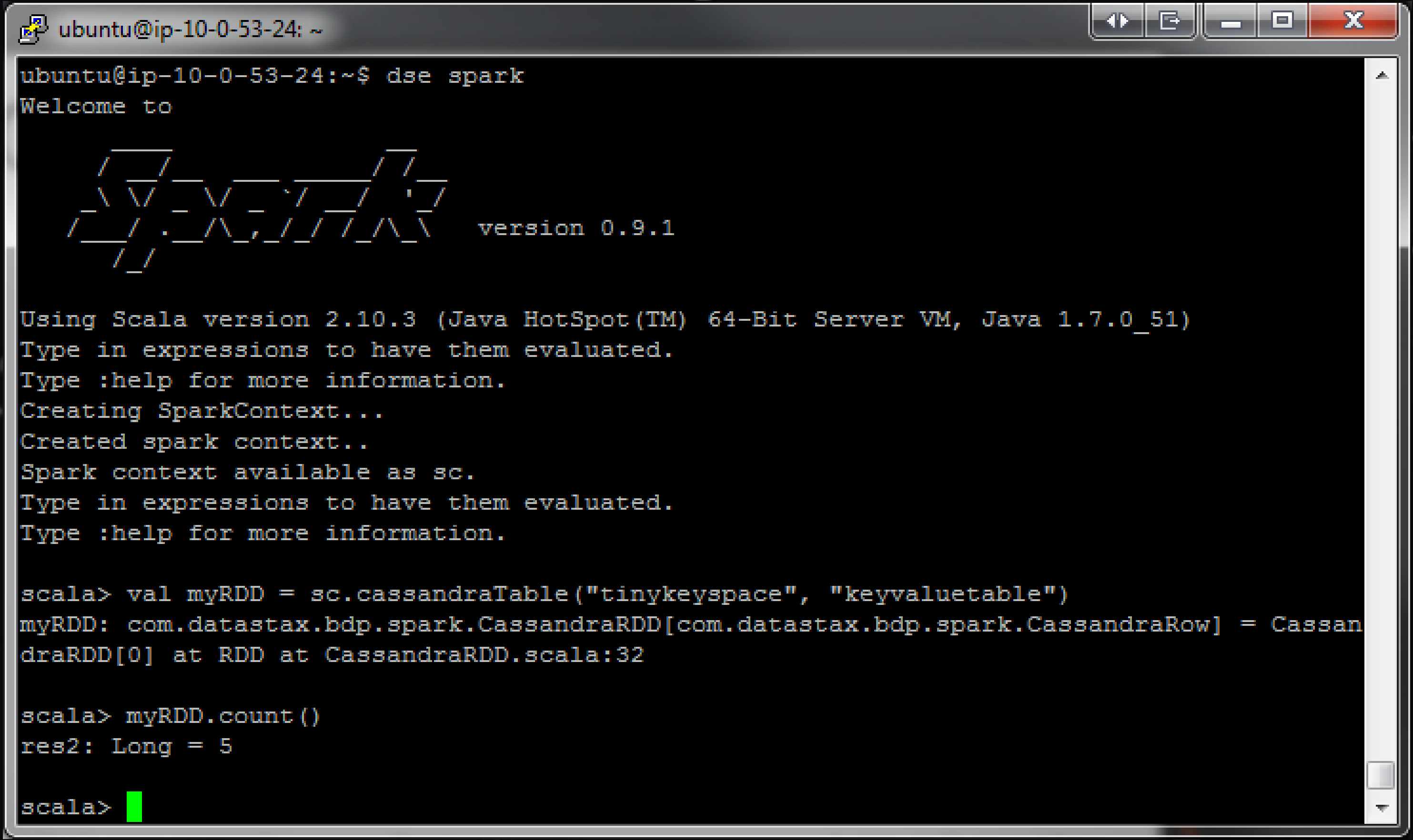
- Each record: 100 bytes (10 byte key & 90 byte value)
- OpenJDK 1.7
- HDFS 2.4.1 w/ short circuit local reads enabled
- Apache Spark 1.2.0
- Speculative Execution off
- Increased Locality Wait to infinite
- Compression turned off for input, output & network
- Used Unsafe to put all the data off-heap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


Spark training from the cr
Databricks, Inc. [US]
https://databricks.com/spark/training
databricks
PRODUCT
SPARK
SOLUTIONS
CUSTOMERS
COMPANY
BLOG
RESOURCES
Training
Careers
Spark Training
As the team that created Apache Spark, Databricks is the authority for Spark training. We teach Spark to software developers and data scientists around the world at conferences, in public training facilities, on-site, and online.
Use the button below to request information about Spark training for your corporation. Farther below is a calendar of Spark training events open to the public for individual registration.
Request Corporate Training Info

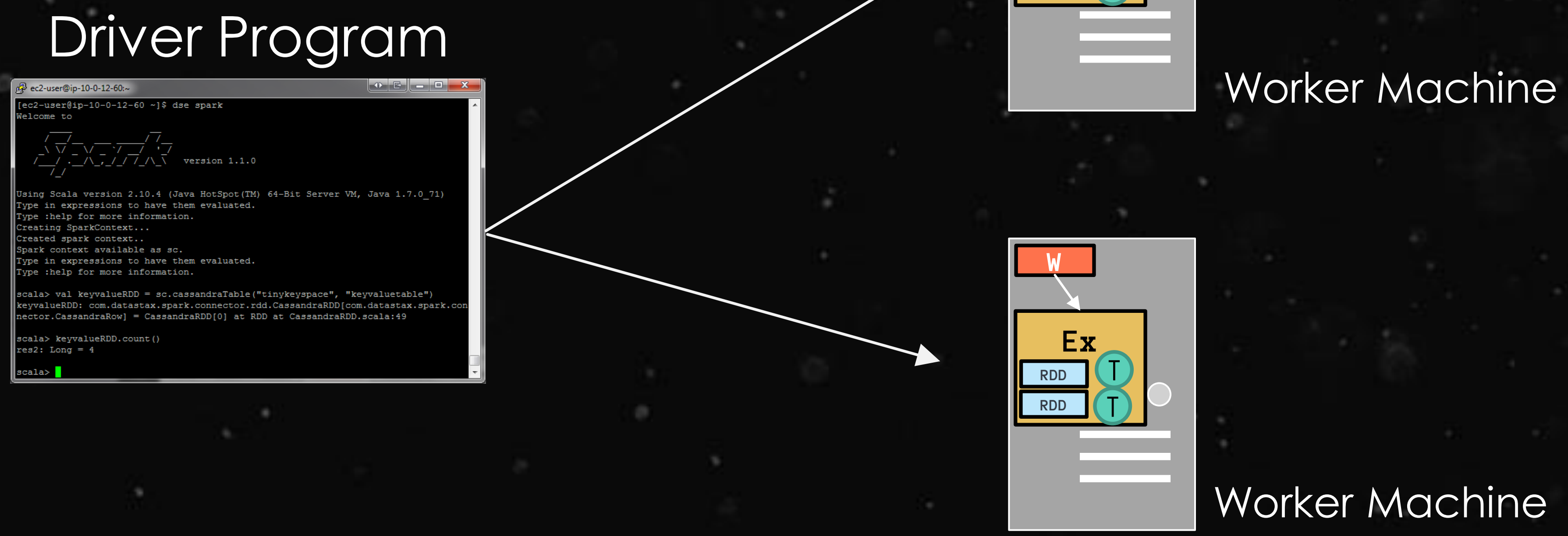
RDD FUNDAMENTALS

databricks

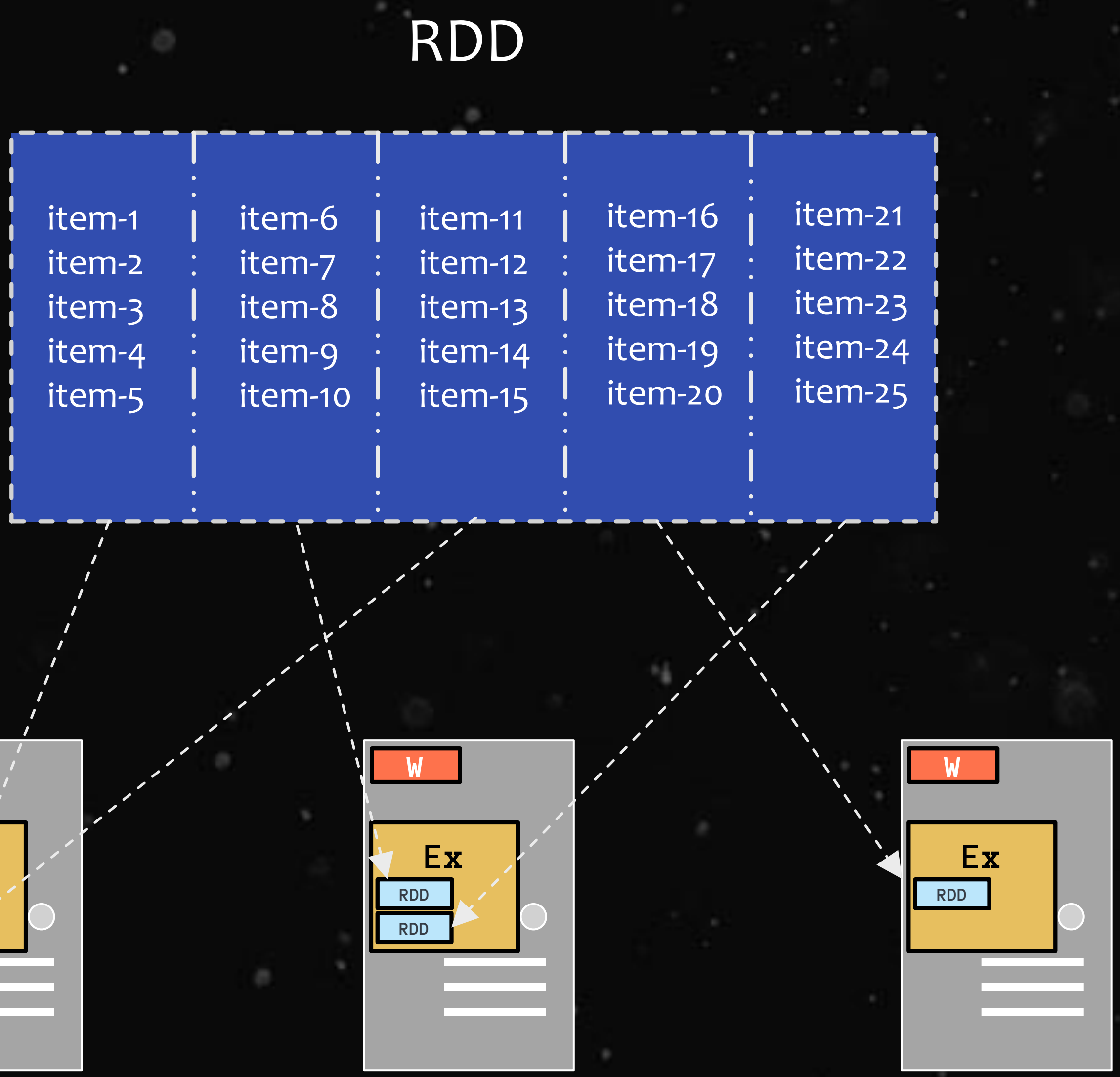
INTERACTIVE SHELL



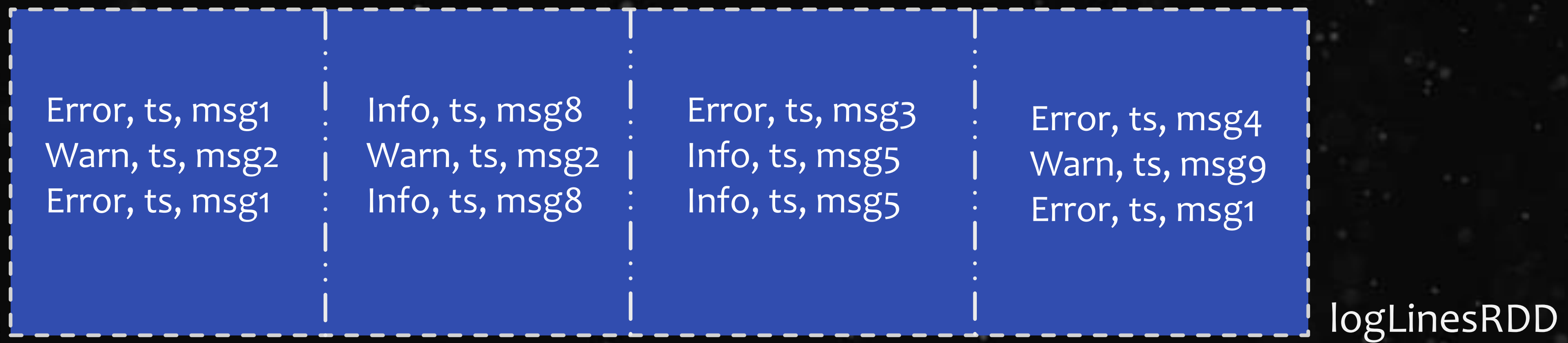
(Scala & Python only)


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

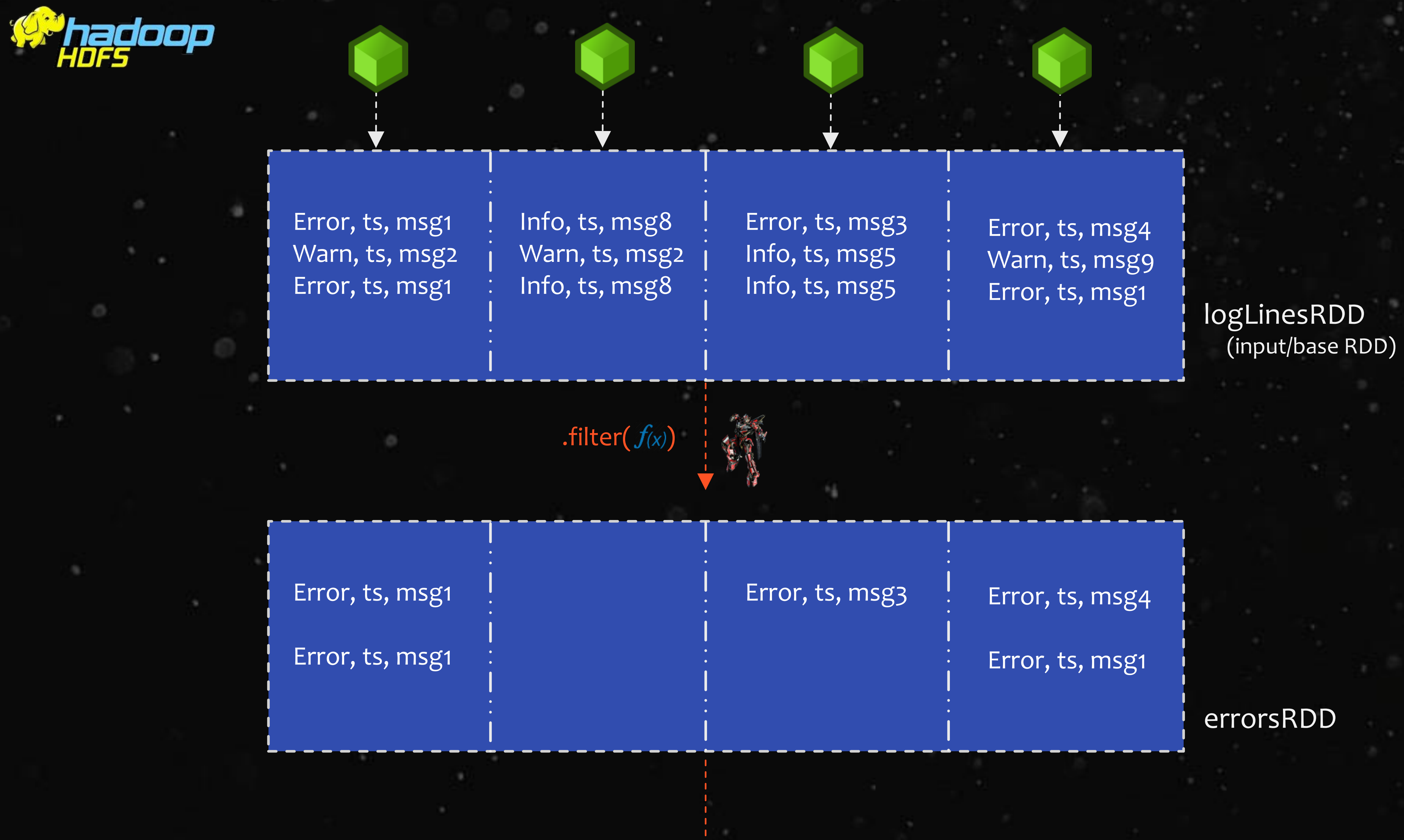
READ FROM TEXT FILE

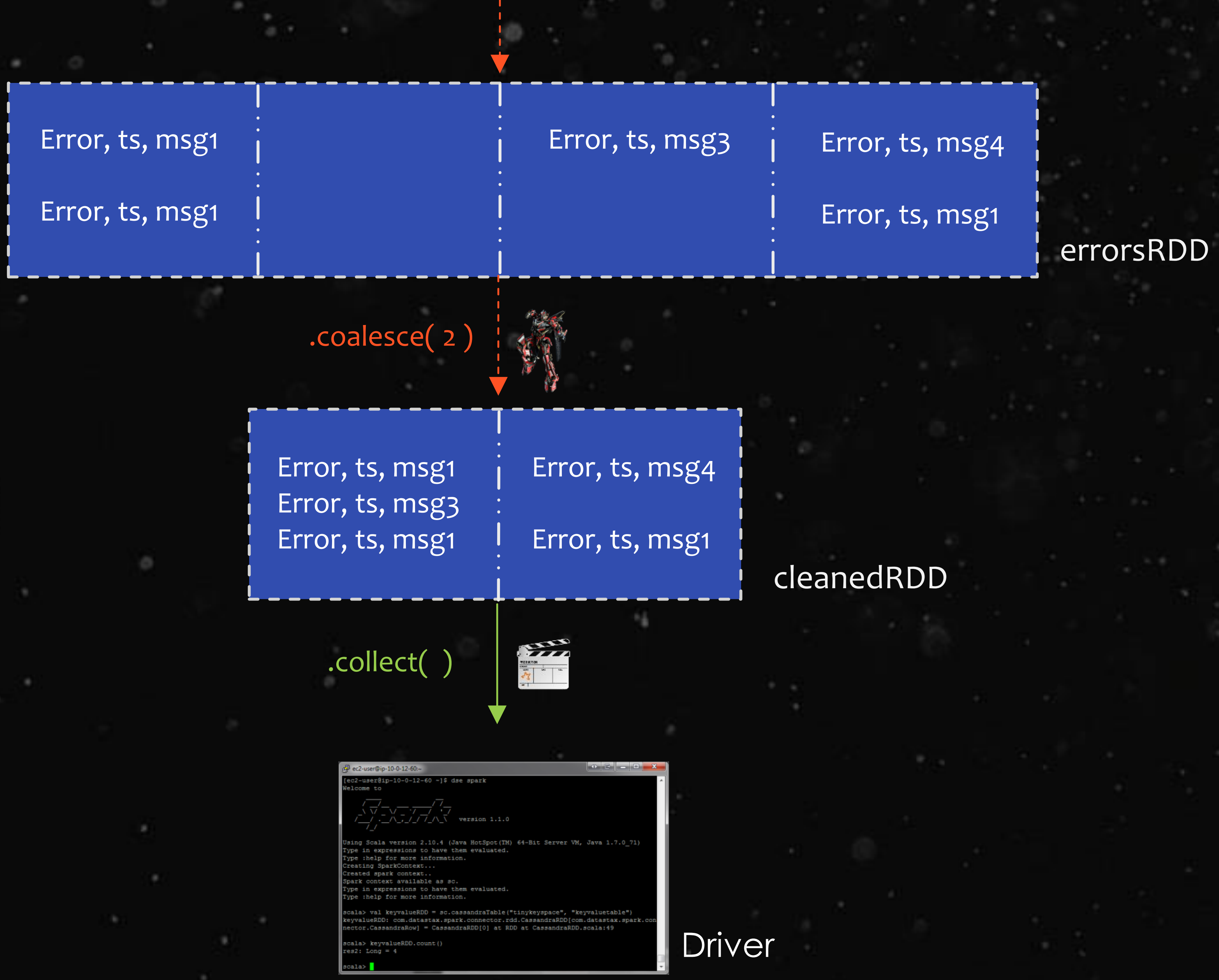
```python
# Read a local txt file in Python
linesRDD = sc.textFile("/path/to/README.md")
```

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

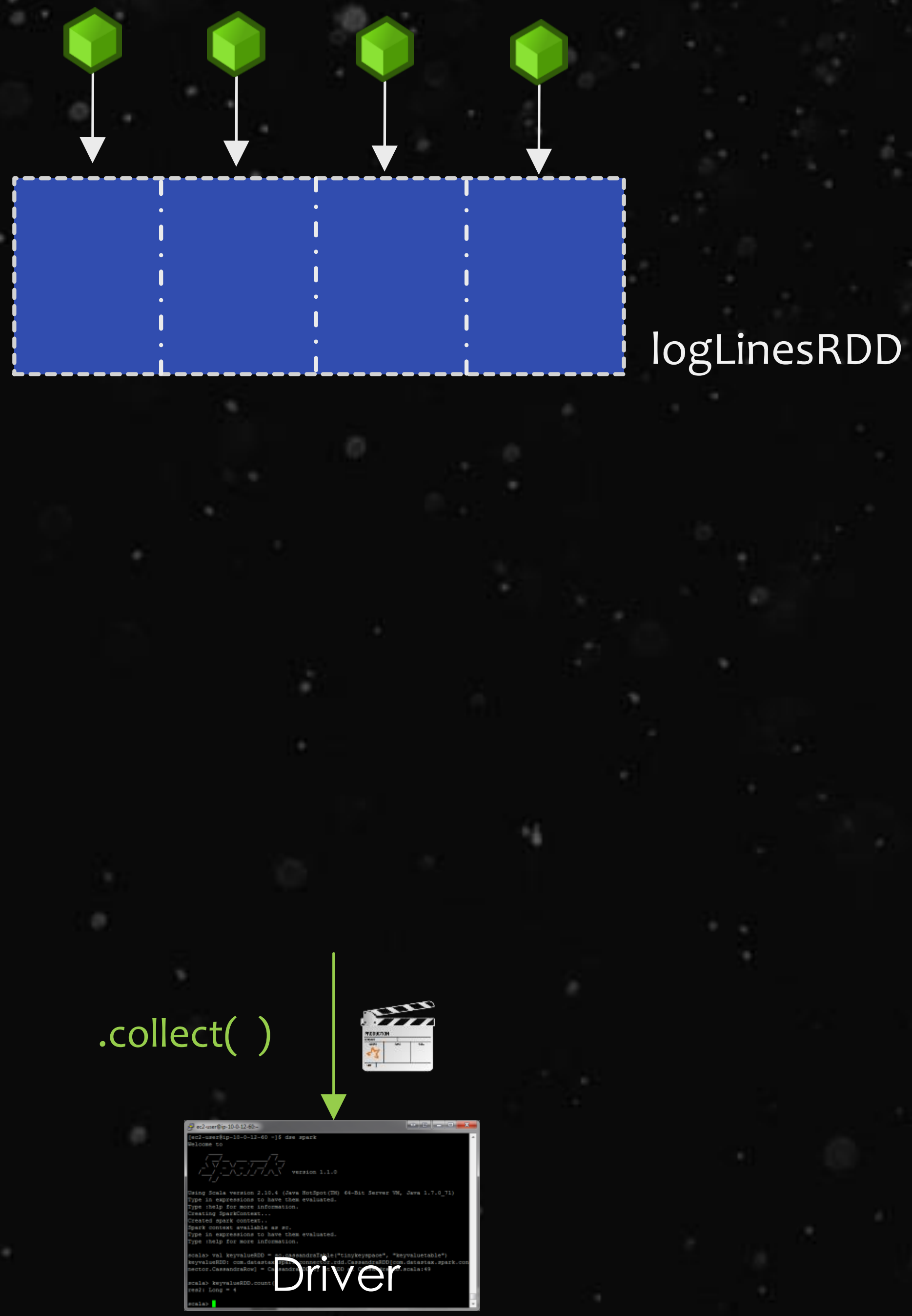
```java
// Read a local txt file in Java
JavaRDD<String> lines = sc.textFile("/path/to/README.md");
```

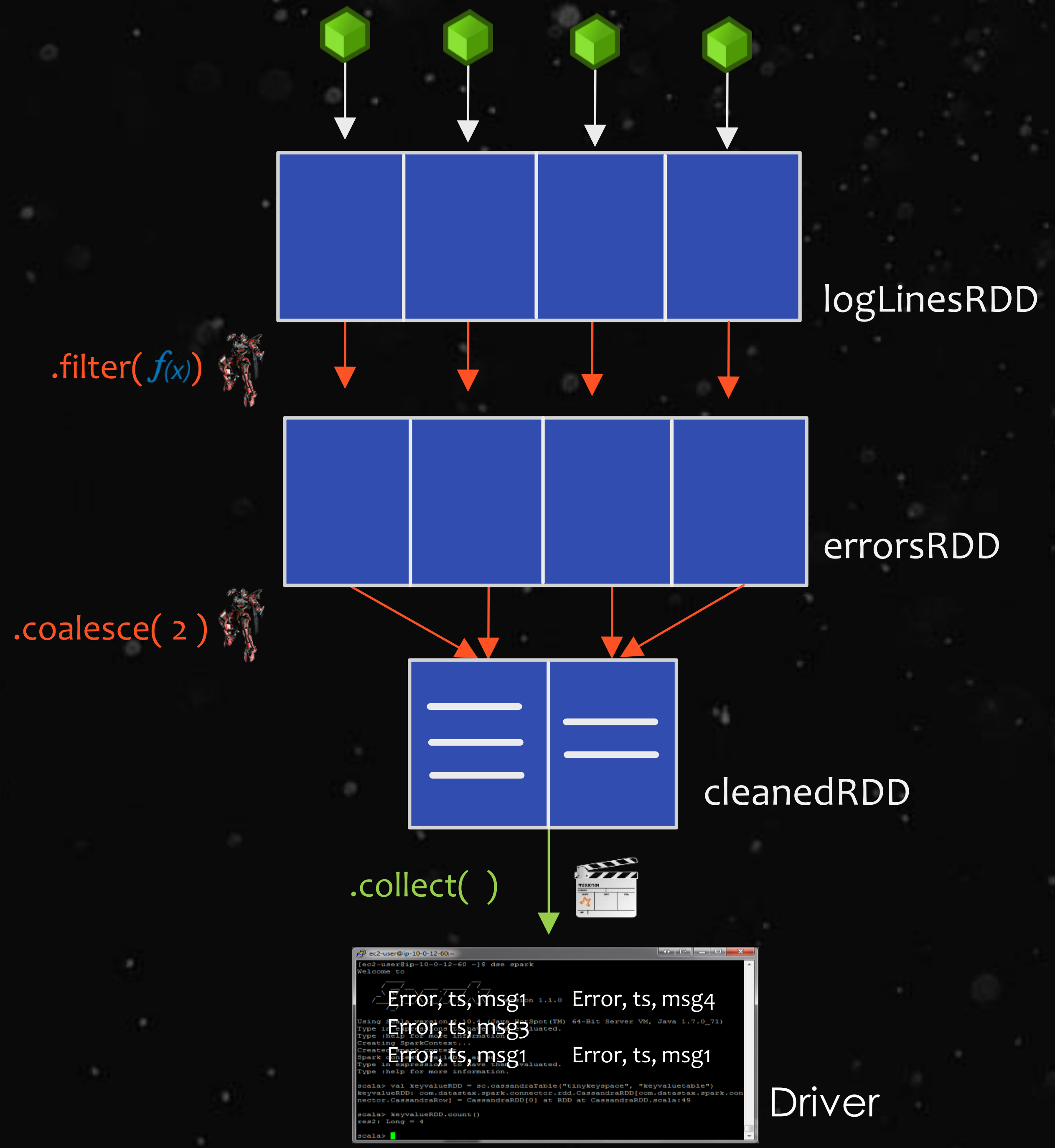
- There are other methods to read data from HDFS, C*, S3, HBase, etc.


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

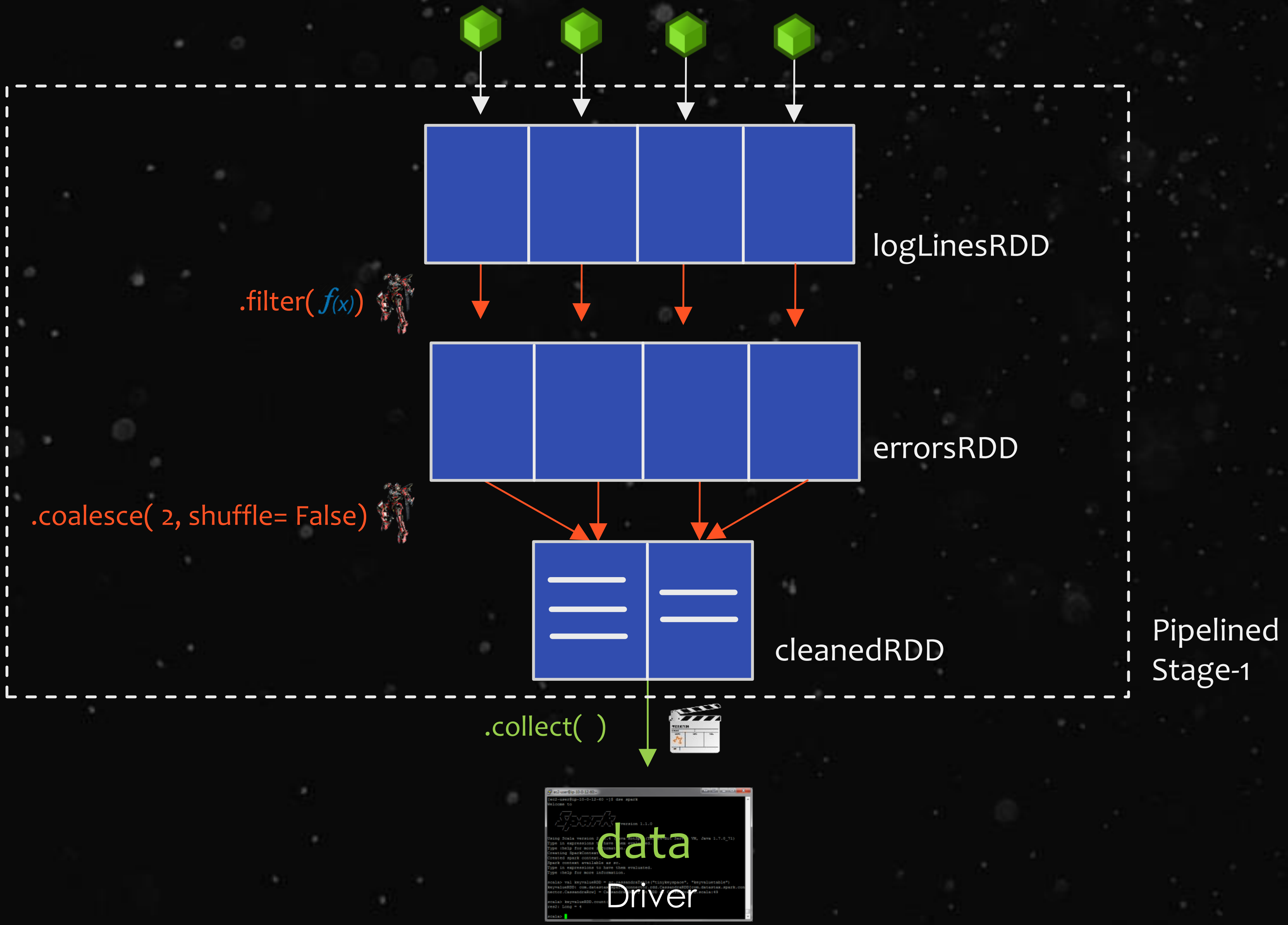

Error, ts, msg1
Error, ts, msg1
Error, ts, msg3
Error, ts, msg4
Error, ts, msg1
errorsRDD
.coalesce(2)
Error, ts, msg1
Error, ts, msg3
Error, ts, msg1
Error, ts, msg4
Error, ts, msg1
cleanedRDD
.collect( )
Driver


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.connector.CassandraRow] = CassandraRDD[0] at RDD at CassandraRDD.scala:49
scala> keyvalueRDD.count()
res2: Long = 4
scala>
Driver


logLinesRDD
.collect( )
Driver

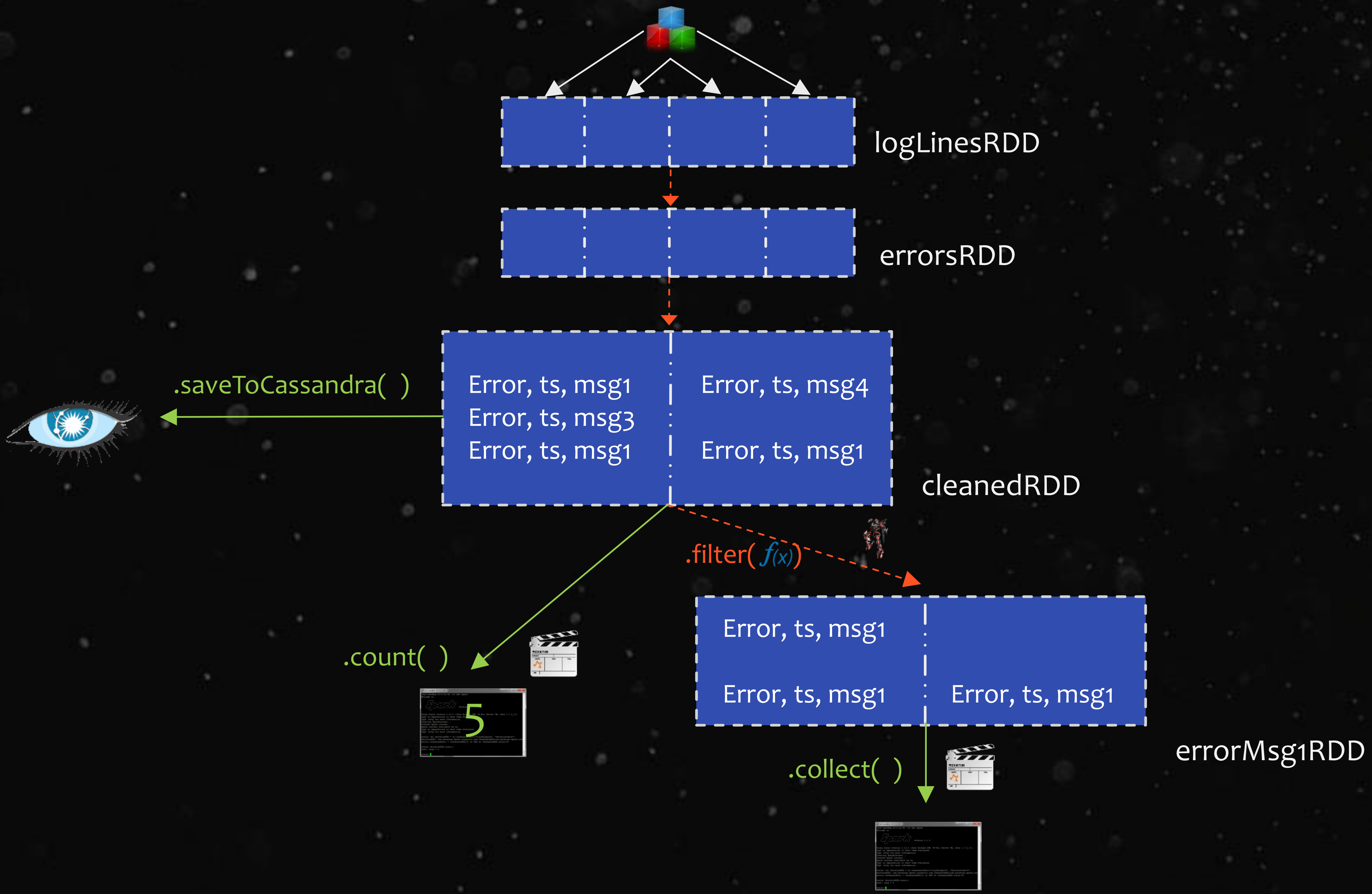


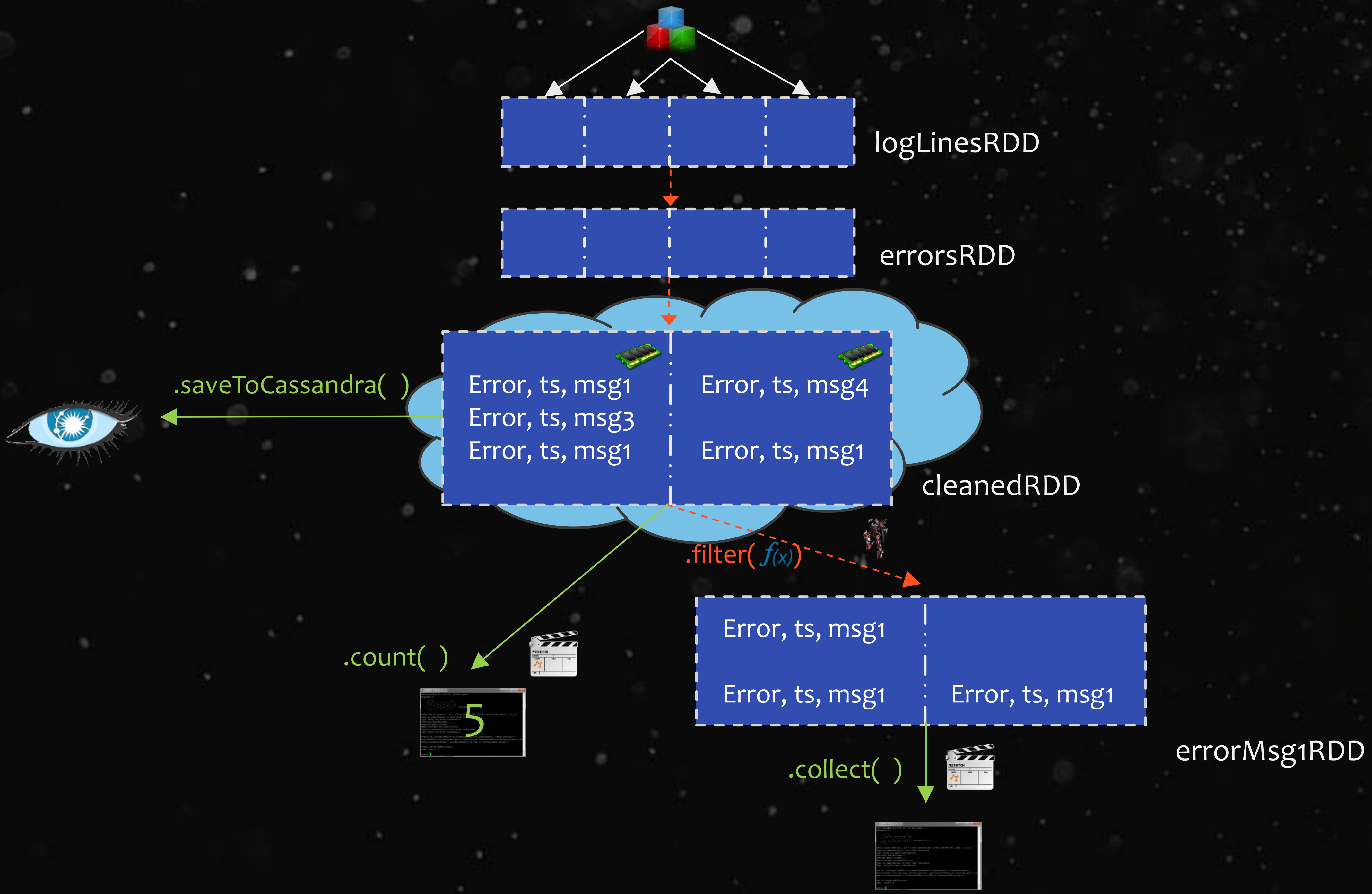
logLinesRDD
.filter(f(x))
errorsRDD
.coalesce(2)
cleanedRDD
.collect( )
Error, ts, msg1
Error, ts, msg3
Error, ts, msg1
Error, ts, msg4
Error, ts, msg1
Driver


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


logLinesRDD
errorsRDD
cleanedRDD
Driver


data
Driver


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



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

LIFECYCLE OF A SPARK PROGRAM

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

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

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

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

TRANSFORMATIONS (lazy)

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

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

ACTIONS

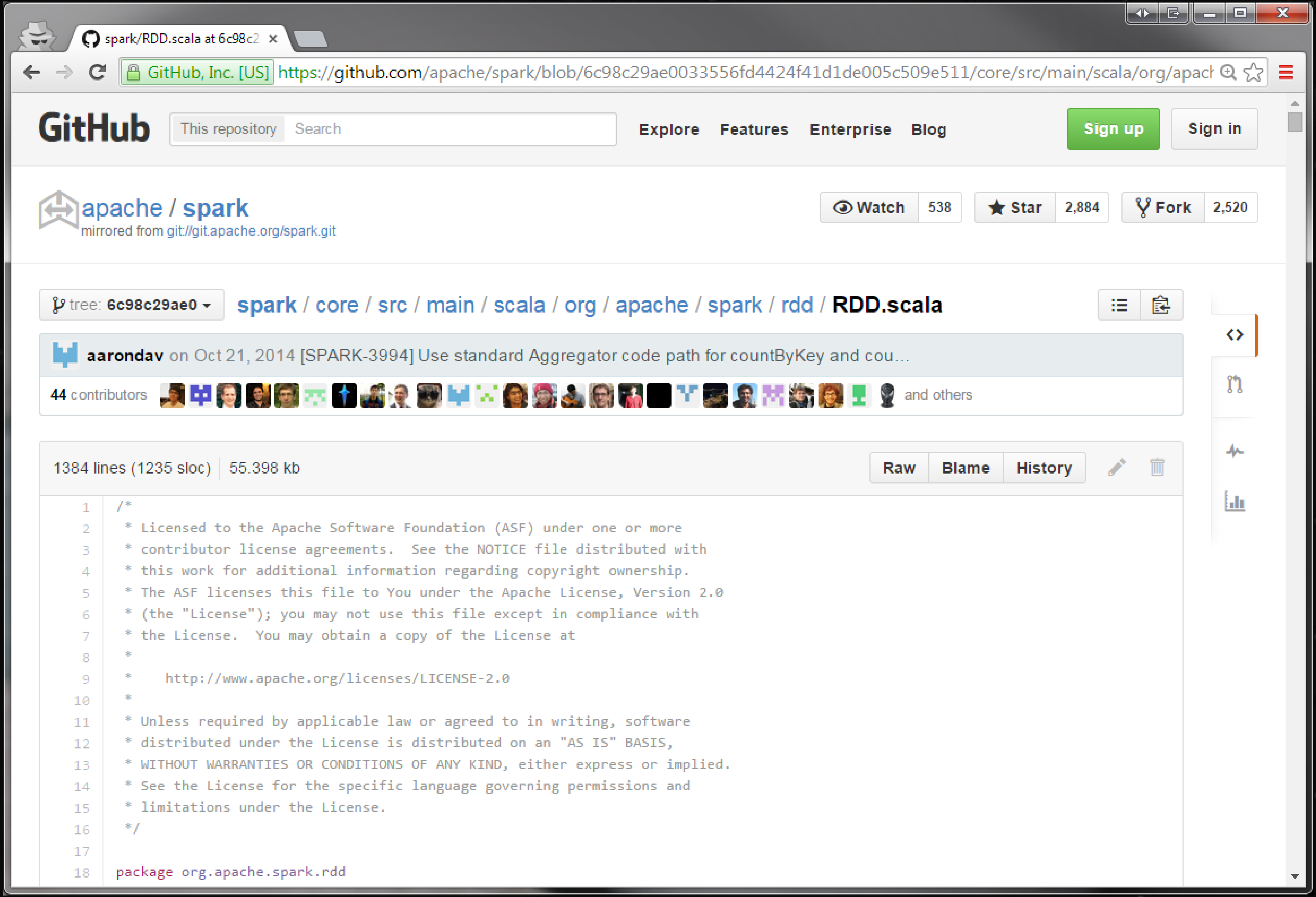
```
reduce()
collect()
count()
first()
take()
takeSample()
saveToCassandra()
takeOrdered()
saveAsTextFile()
saveAsSequenceFile()
saveAsObjectFile()
countByKey()
foreach()
...
```

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

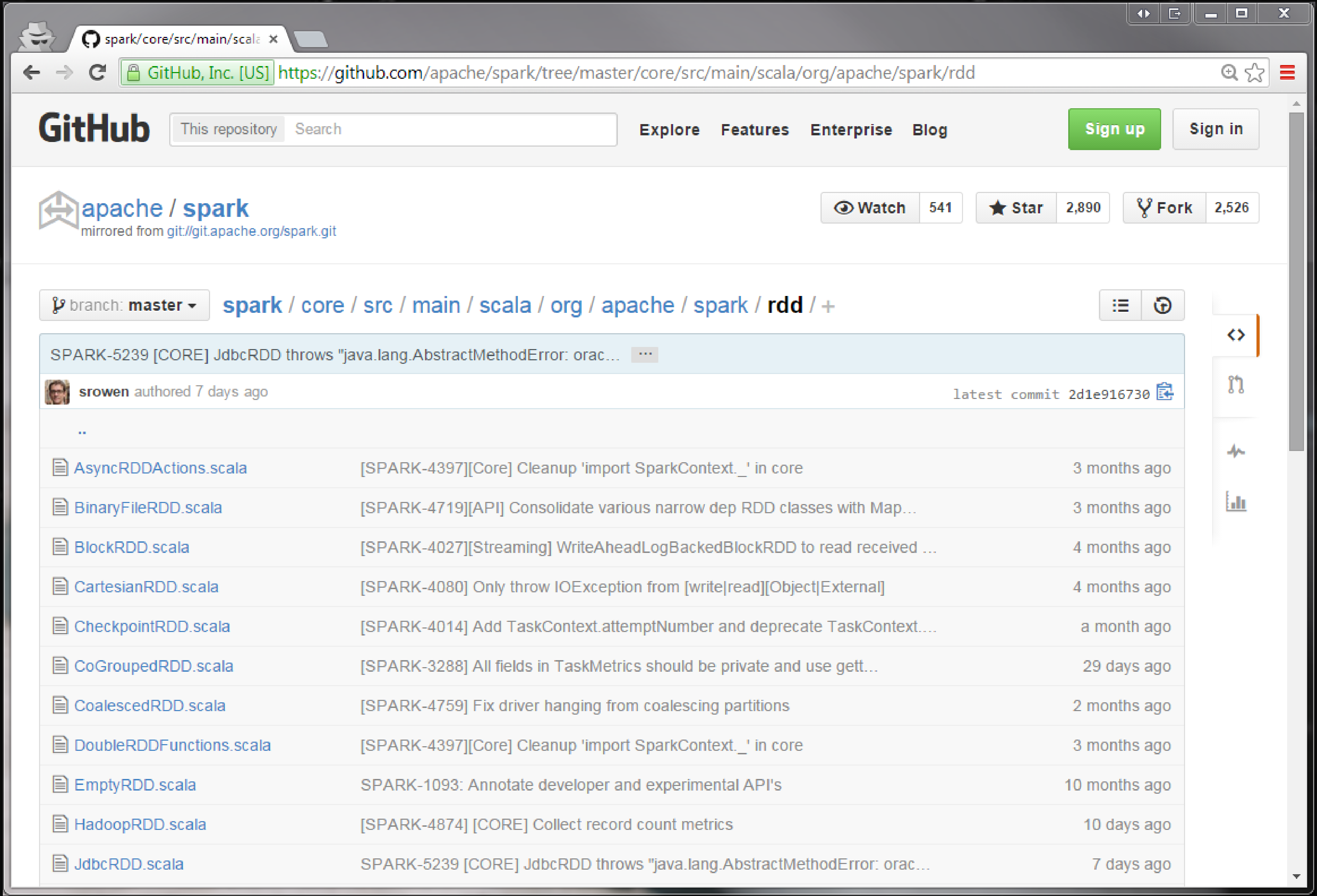
tree: 6c98c29ae0 spark / core / src / main / scala / org / apache / spark / rdd / RDD.scala

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

“Simple things should be simple, complex things should be possible”

- Alan Kay

DEMO: DATABRICKS GUI

databricks

https://classeast01.cloud.databricks.com

- 60 user accounts
- 60 user clusters
- 1 community cluster

https://classeast02.cloud.databricks.com

- 60 user accounts
- 60 user clusters
- 1 community cluster

Databricks Guide (5 mins)

DevOps 101 (30 mins)

Transformations & Actions (30 mins)

DevOps 102 (30 mins)

SQL 101 (30 mins)

Dataframes (20 mins)

Switch to Transformations & Actions slide deck….

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 + DATAFRAMES

databricks

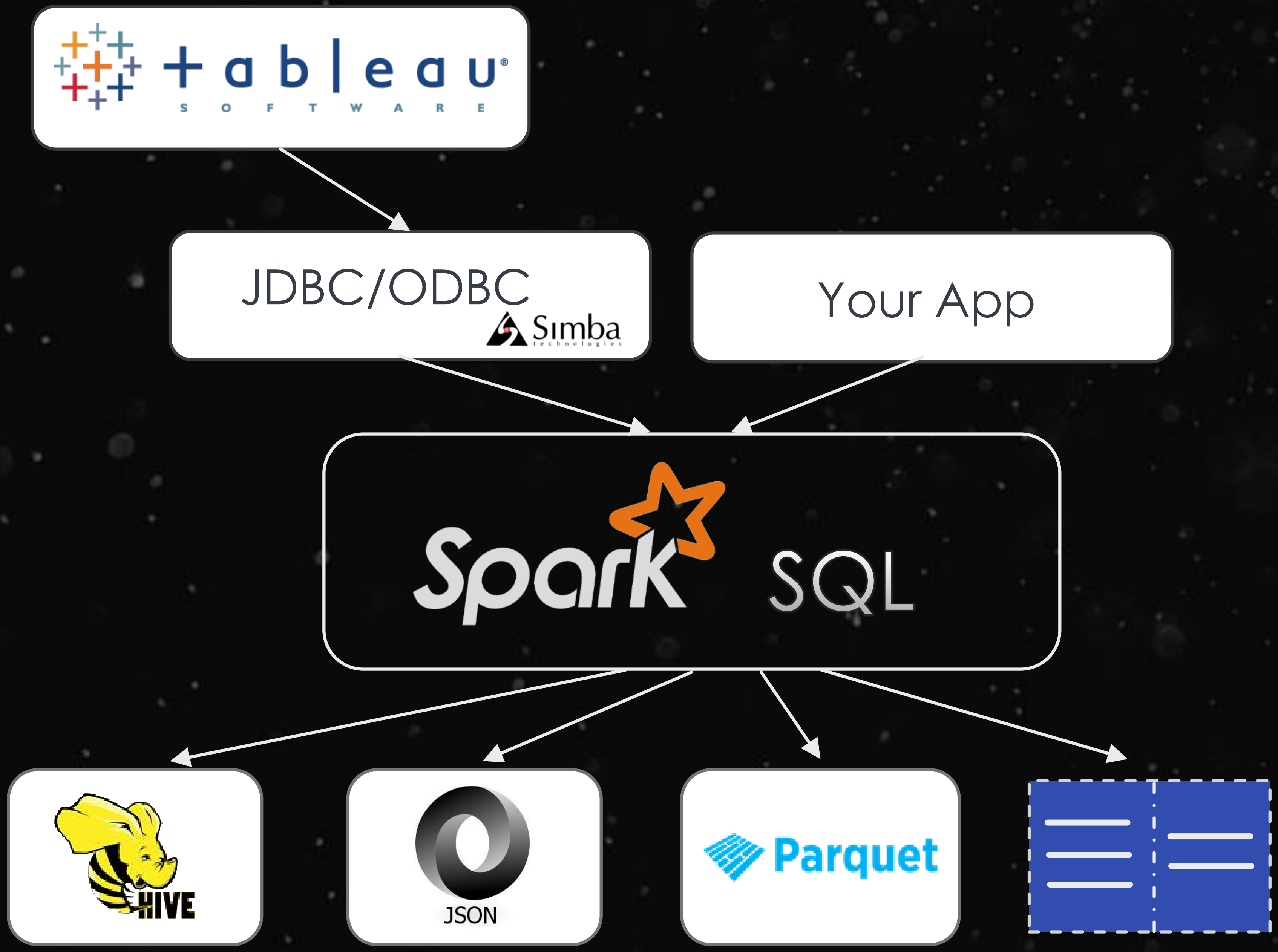


tableau
SOFTWARE
JDBC/ODBC
Simba
Your App
Spark
SQL
HIVE
JSON
Parquet

Spark SQL and DataFrame Guide

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

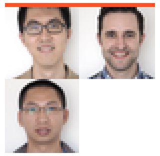
databricks PRODUCT SPARK SERVICES COMPANY CAREERS BLOG

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.

DATAFRAMES

- Announced Feb 2015
- Inspired by data frames in R and Pandas in Python
- Works in:

What is a Dataframe?

- a distributed collection of data organized into named columns
- Like a table in a relational database

Features

- Scales from KBs to PBs
- Supports wide array of data formats and storage systems (Hive, existing RDDs, etc)
- State-of-the-art optimization and code generation via Spark SQL Catalyst optimizer
- APIs in Python, Java

DATAFRAMES

Step 1: Construct a DataFrame

```
from pyspark.sql import SQLContext
sqlContext = SQLContext(sc)

df = sqlContext.jsonFile("examples/src/main/resources/people.json")

# Displays the content of the DataFrame to stdout
df.show()
## age  name
## null Michael
## 30   Andy
## 19   Justin
```

DATAFRAMES

Step 2: Use the DataFrame

```
# Print the schema in a tree format
df.printSchema()
## root
## |-- age: long (nullable = true)
## |-- name: string (nullable = true)

# Select only the "name" column
df.select("name").show()
## name
## Michael
## Andy
## Justin

# Select everybody, but increment the age by 1
df.select("name", df.age + 1).show()
## name    (age + 1)
## Michael null
## Andy    31
## Justin  20
```

DATAFRAMES

SQL Integration

```
from pyspark.sql import SQLContext
sqlContext = SQLContext(sc)

df = sqlContext.sql("SELECT * FROM table")
```

DATAFRAMES

SQL + RDD Integration

2 methods for converting existing RDDs into DataFrames:

(more concise) 1. Use reflection to infer the schema of an RDD that contains different types of objects

(more verbose) 2. Use a programmatic interface that allows you to construct a schema and then apply it to an existing RDD.

DATAFRAMES

SQL + RDD Integration: via reflection

```
# 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 DataFrame as a table.
schemaPeople = sqlContext.inferSchema(people)
schemaPeople.registerTempTable("people")
```

DATAFRAMES

SQL + RDD Integration: via reflection

```
# SQL can be run over DataFrames 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
```

DATAFRAMES

SQL + RDD Integration: via programmatic schema

DataFrame can be created programmatically with 3 steps:

1. Create an RDD of tuples or lists from the original RDD
2. Create the schema represented by a `StructType` matching the structure of tuples or lists in the RDD created in the step 1
3. Apply the schema to the RDD via `createDataFrame` method provided by `SQLContext`

DATAFRAMES

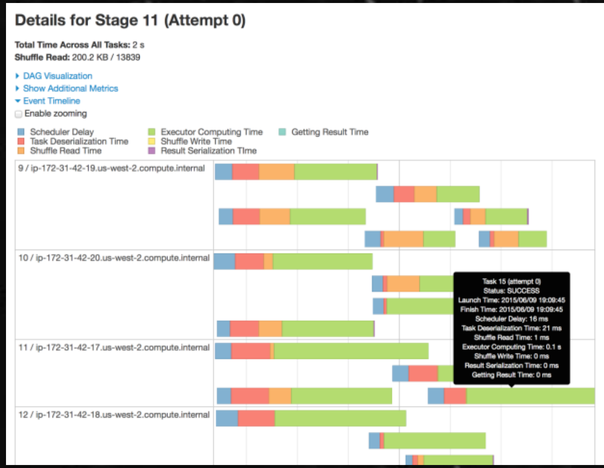
Step 1: Construct a DataFrame

```
# Constructs a DataFrame from the users table in Hive.
users = context.table("users")

# from JSON files in S3
logs = context.load("s3n://path/to/data.json", "json")
```

DATAFRAMES

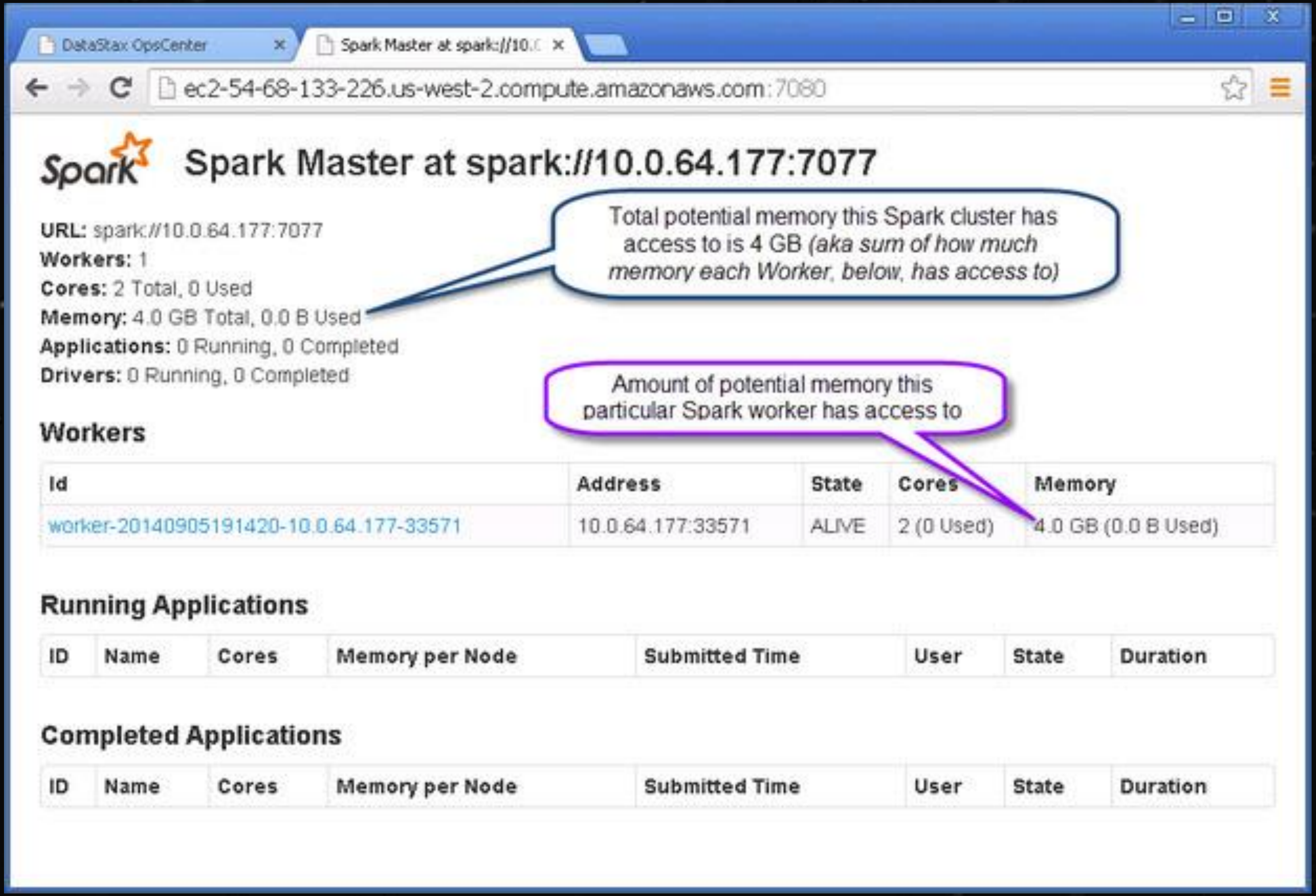
Step 2: Use the DataFrame

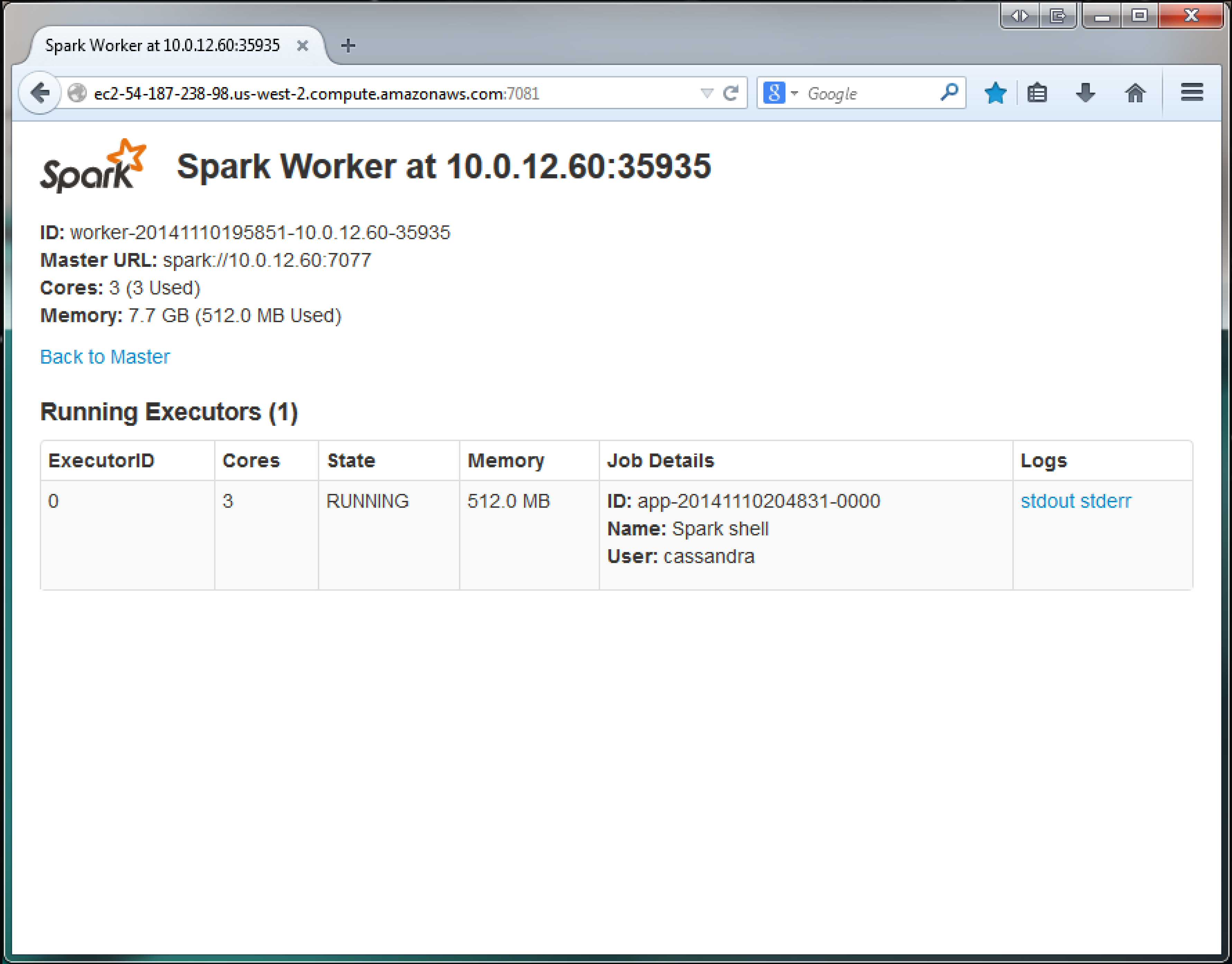
```
# Create a new DataFrame that contains “young users” only
young = users.filter(users.age < 21)

# Alternatively, using Pandas-like syntax
young = users[users.age < 21]

# Increment everybody’s age by 1
young.select(young.name, young.age + 1)

# Count the number of young users by gender
young.groupBy("gender").count()

# Join young users with another DataFrame called logs
young.join(logs, logs.userId == users.userId, "left_outer")
```



SPARK UI

databricks


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 Worker at 10.0.12.60:35935
ec2-54-187-238-98.us-west-2.compute.amazonaws.com:7081
Google
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/

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)

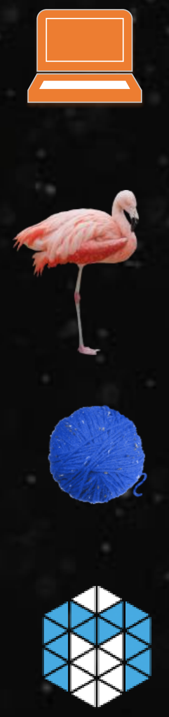
Total Duration: 39 min
Scheduling Mode: FIFO
Active Stages: 0
Completed Stages: 5
Failed Stages: 0

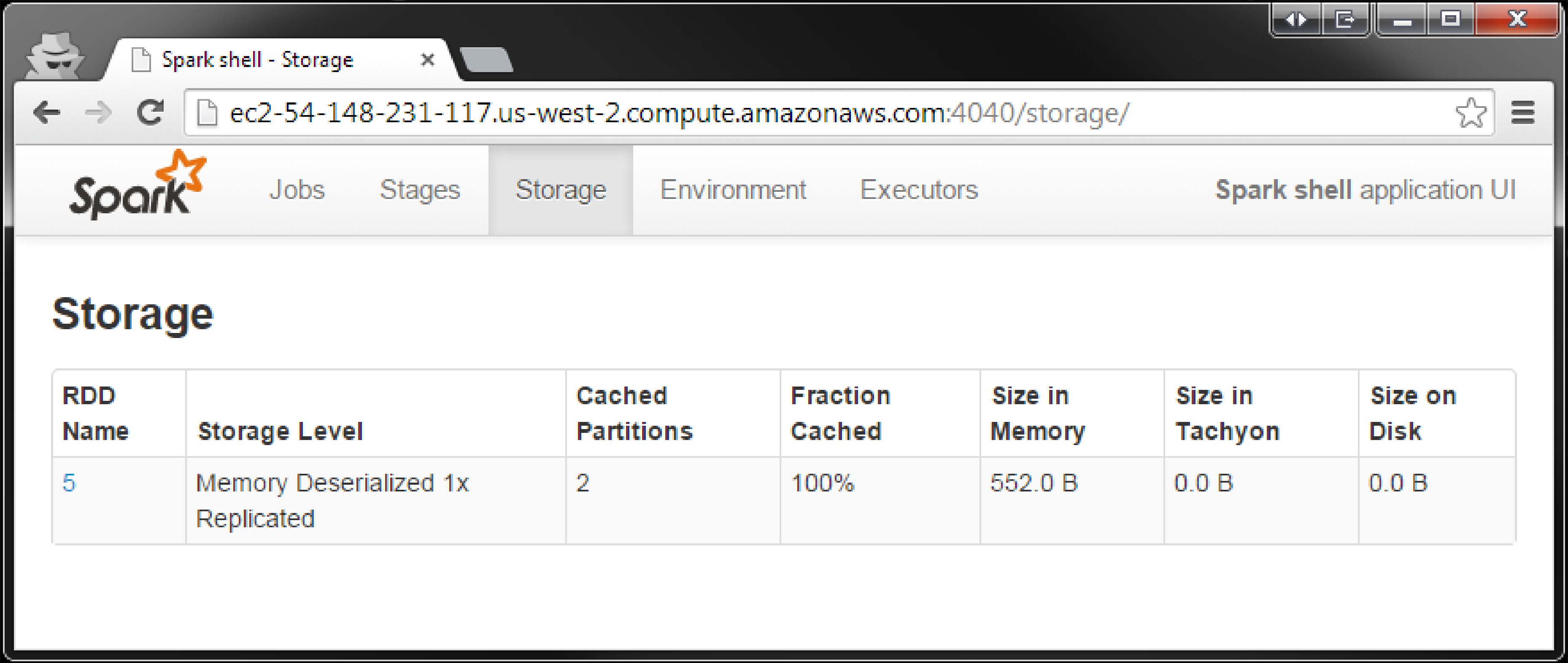
Active Stages (0)

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			





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/

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 1.4.0

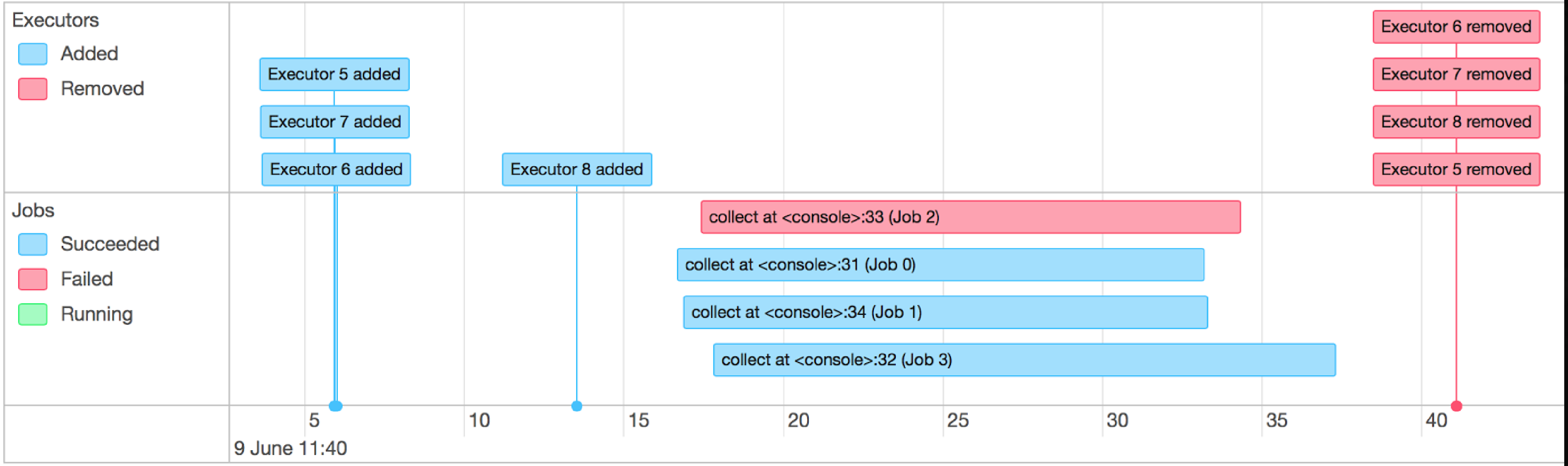
NEW

Event timeline all jobs page

Spark Jobs (?)

Total Uptime: 2.2 min
Scheduling Mode: FIFO
Completed Jobs: 3
Failed Jobs: 1

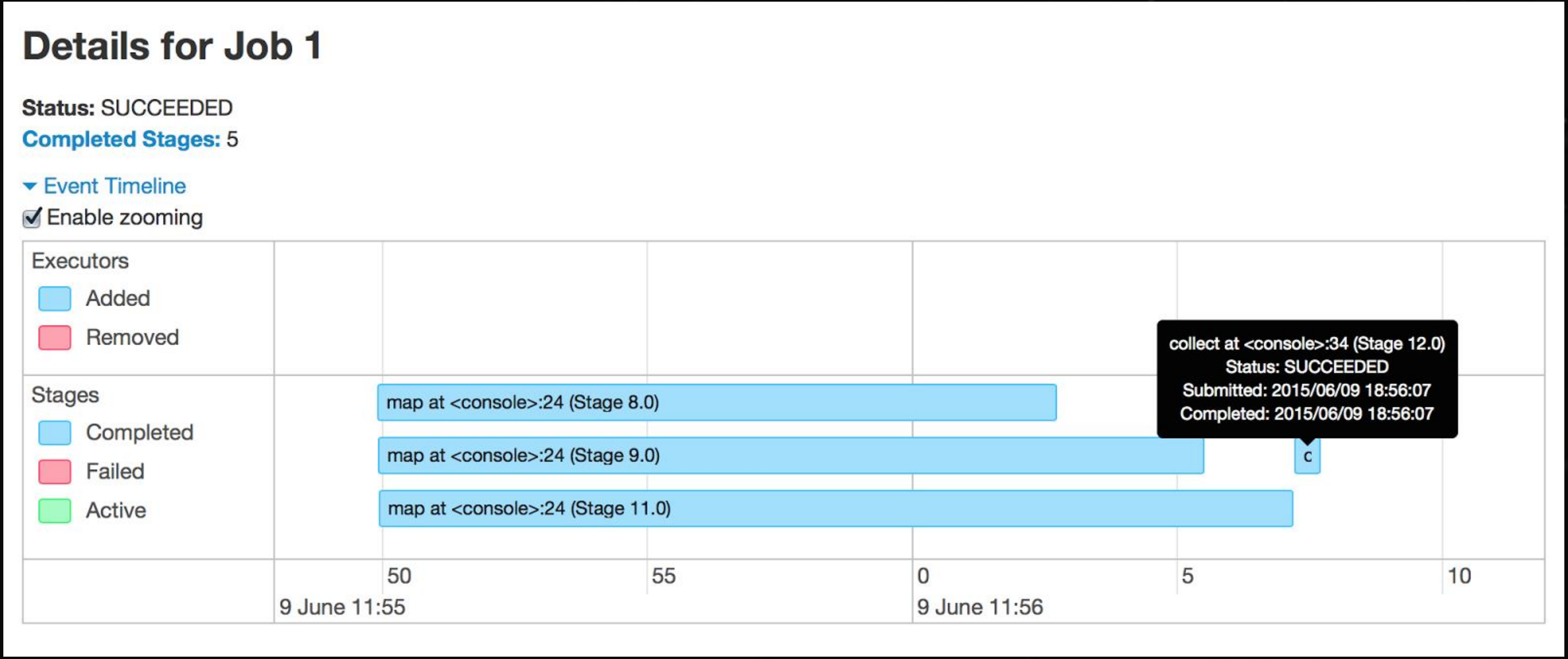
▾ Event Timeline
☑ Enable zooming



1.4.0

NEW

Event timeline within 1 job



Spark 1.4.0

NEW

Event timeline within 1 stage

Details for Stage 11 (Attempt 0)

Total Time Across All Tasks: 2 s
Shuffle Read: 200.2 KB / 13839

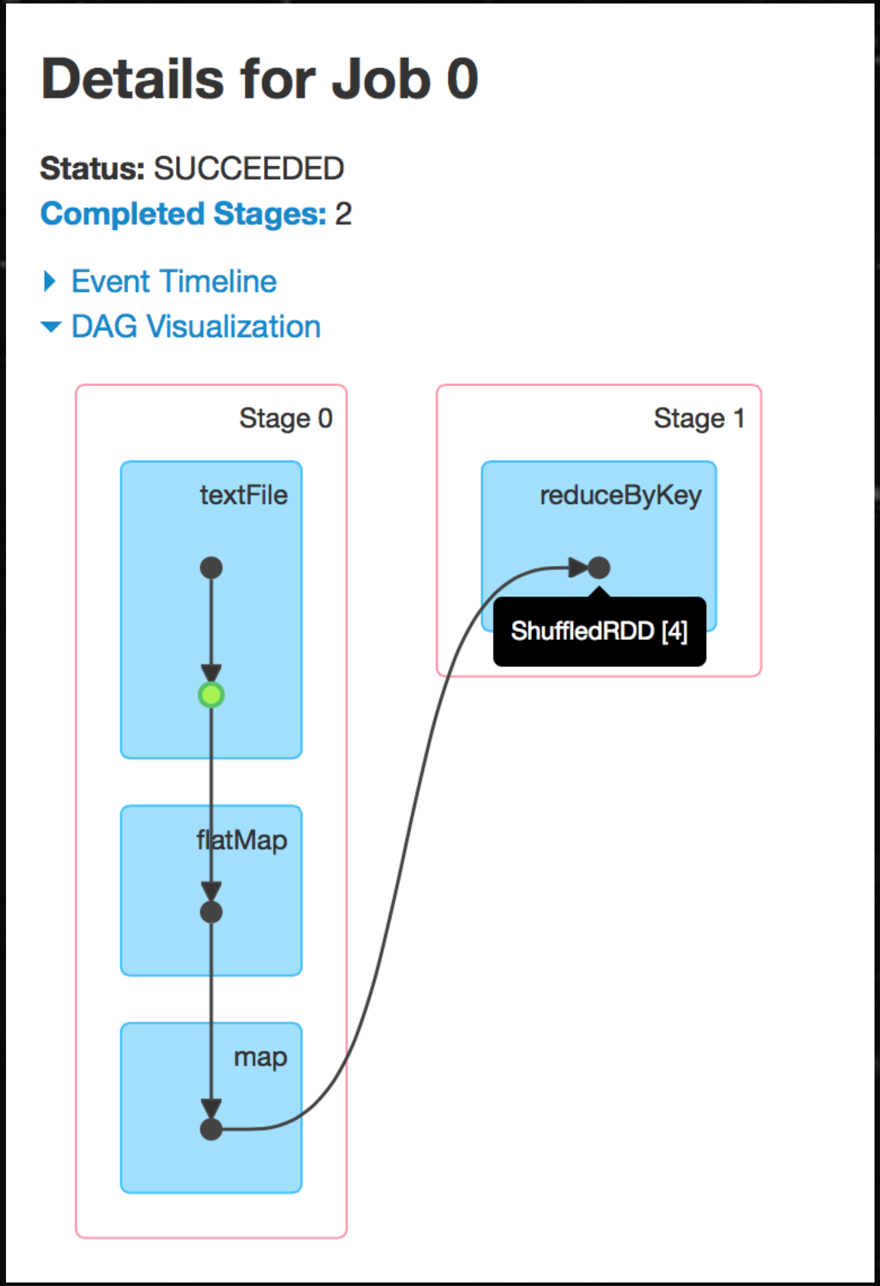
- DAG Visualization
- Show Additional Metrics
- Event Timeline

Enable zooming

Spark 1.4.0

NEW

```
sc.textFile("blog.txt")
 .cache()
 .flatMap { line => line.split(" ") }
 .map { word => (word, 1) }
 .reduceByKey { case (count1, count2) => count1 + count2 }
 .collect()
```



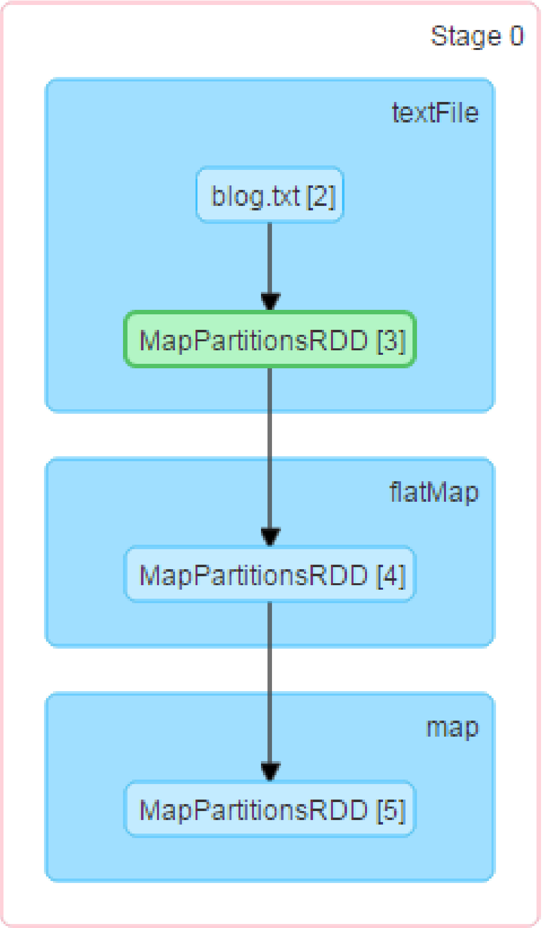
Spark 1.4.0

NEW

Details for Stage 0 (Attempt 0)

Total Time Across All Tasks: 22 s
Input Size / Records: 195.4 MB / 2668833
Shuffle Write: 13.4 KB / 1318

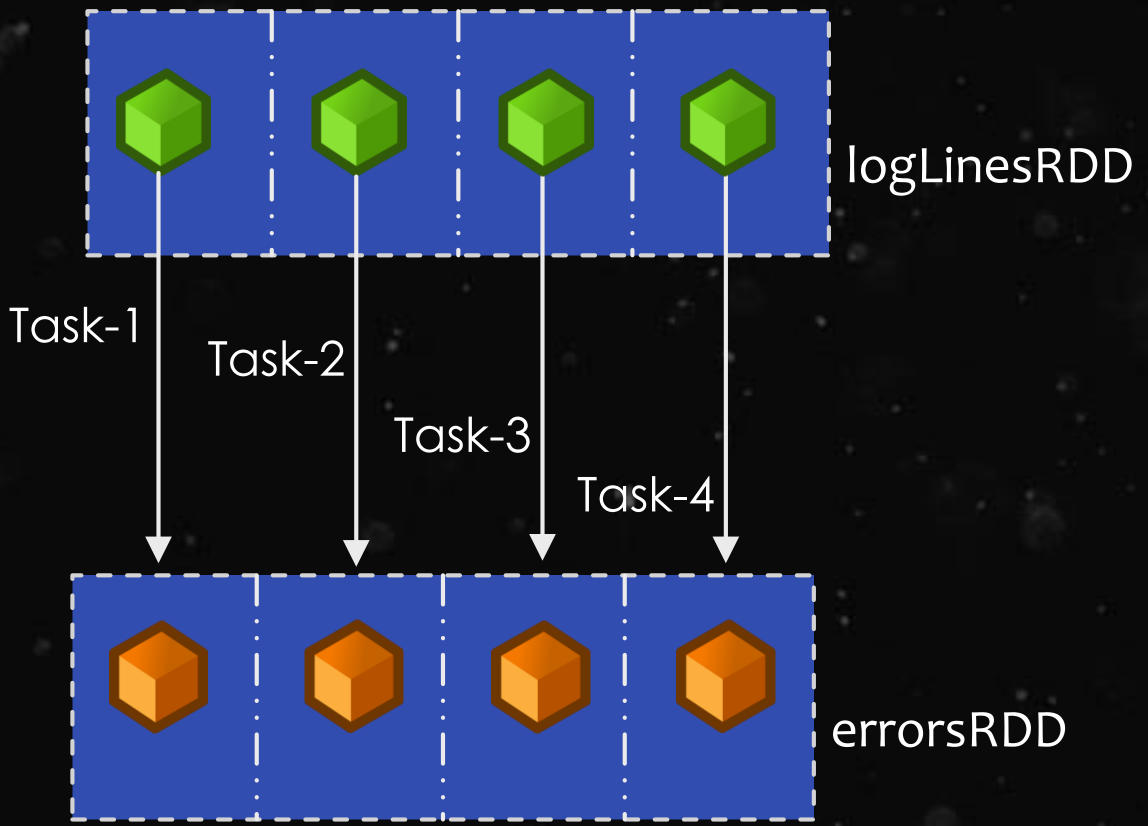
▾ DAG Visualization



SPARK RESOURCE MANAGERS

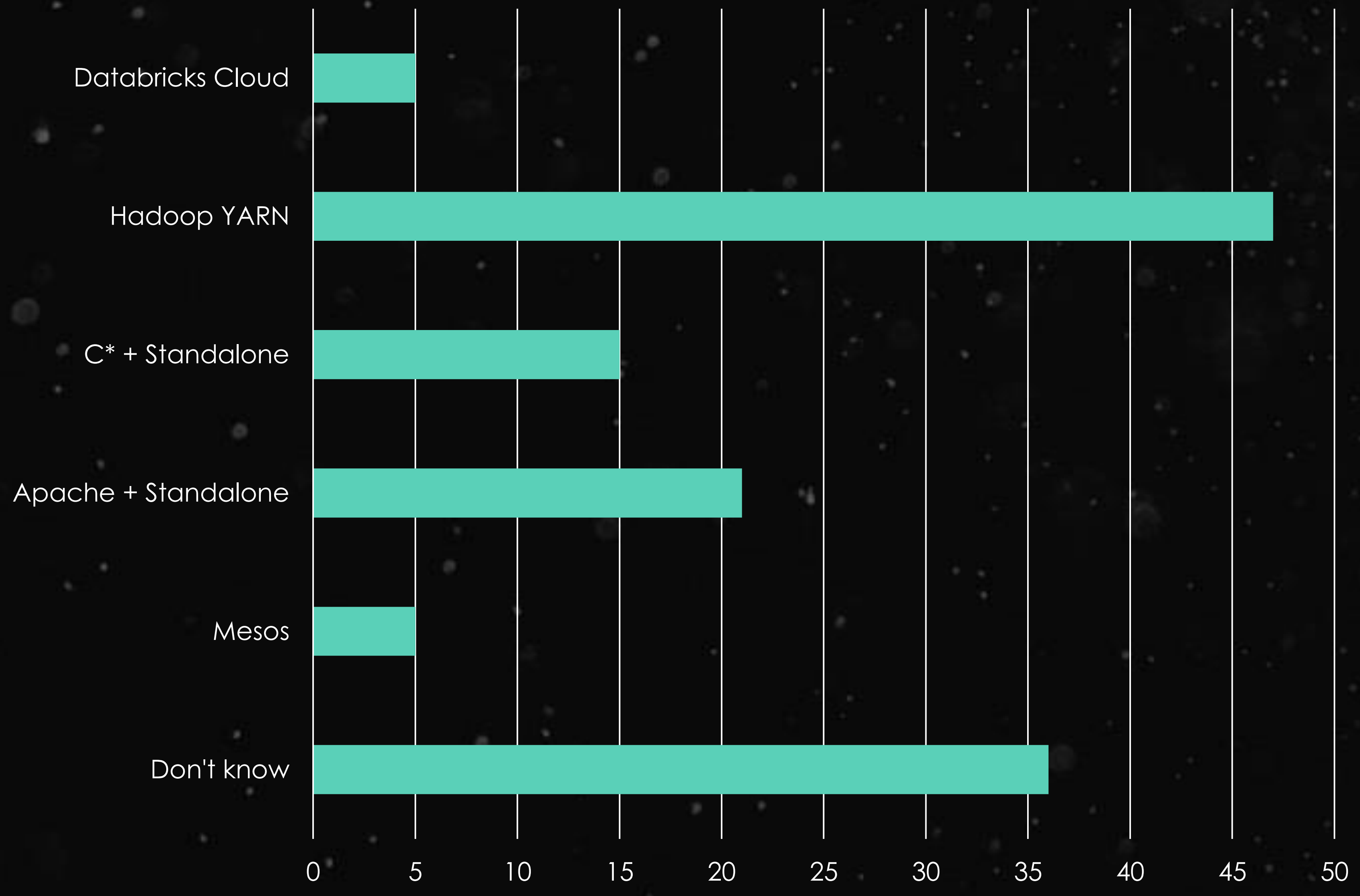
databricks

WHAT ARE TASKS?



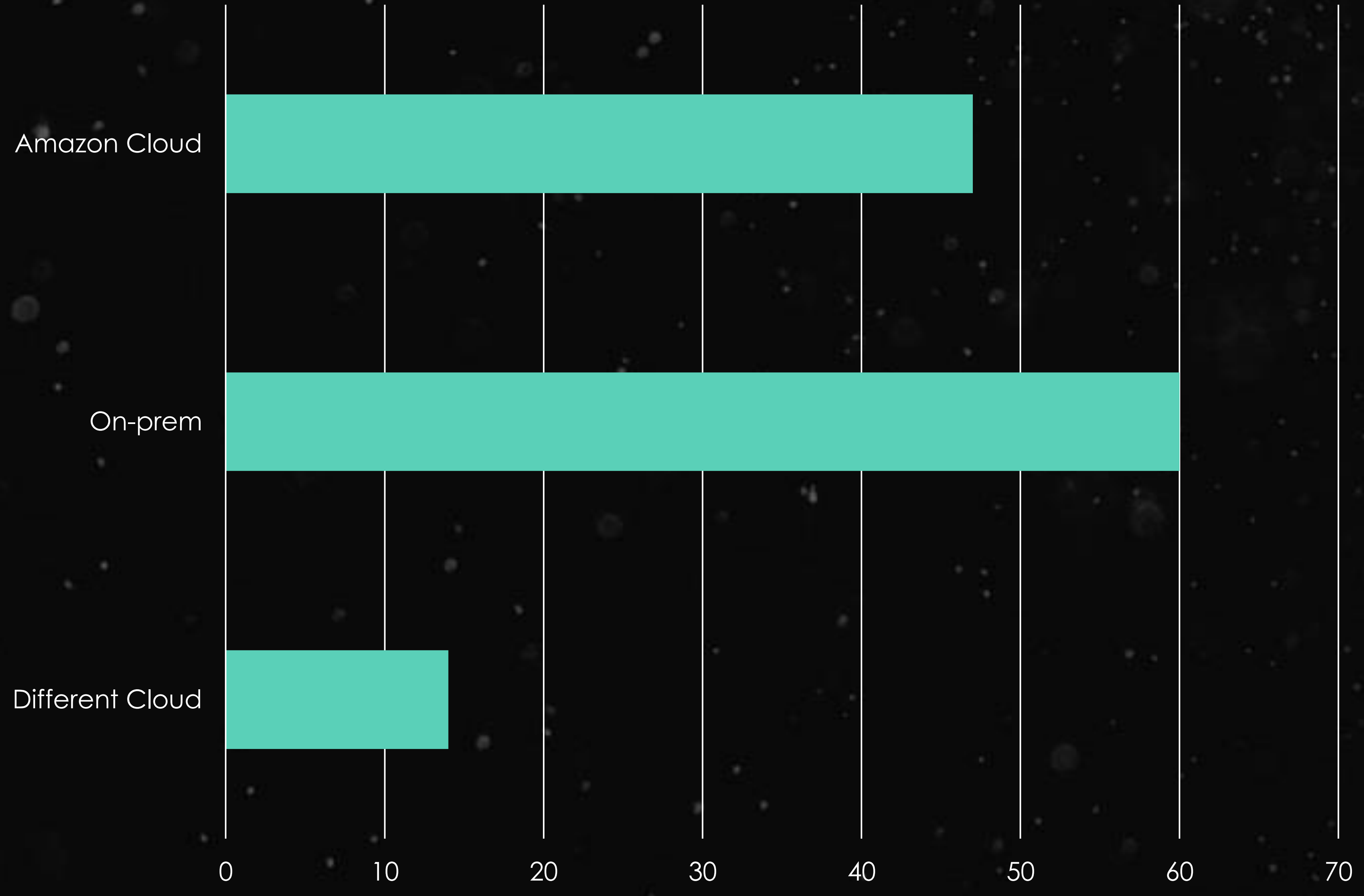
HOW WILL YOU DEPLOY SPARK?

Survey completed by
58 out of 115 students

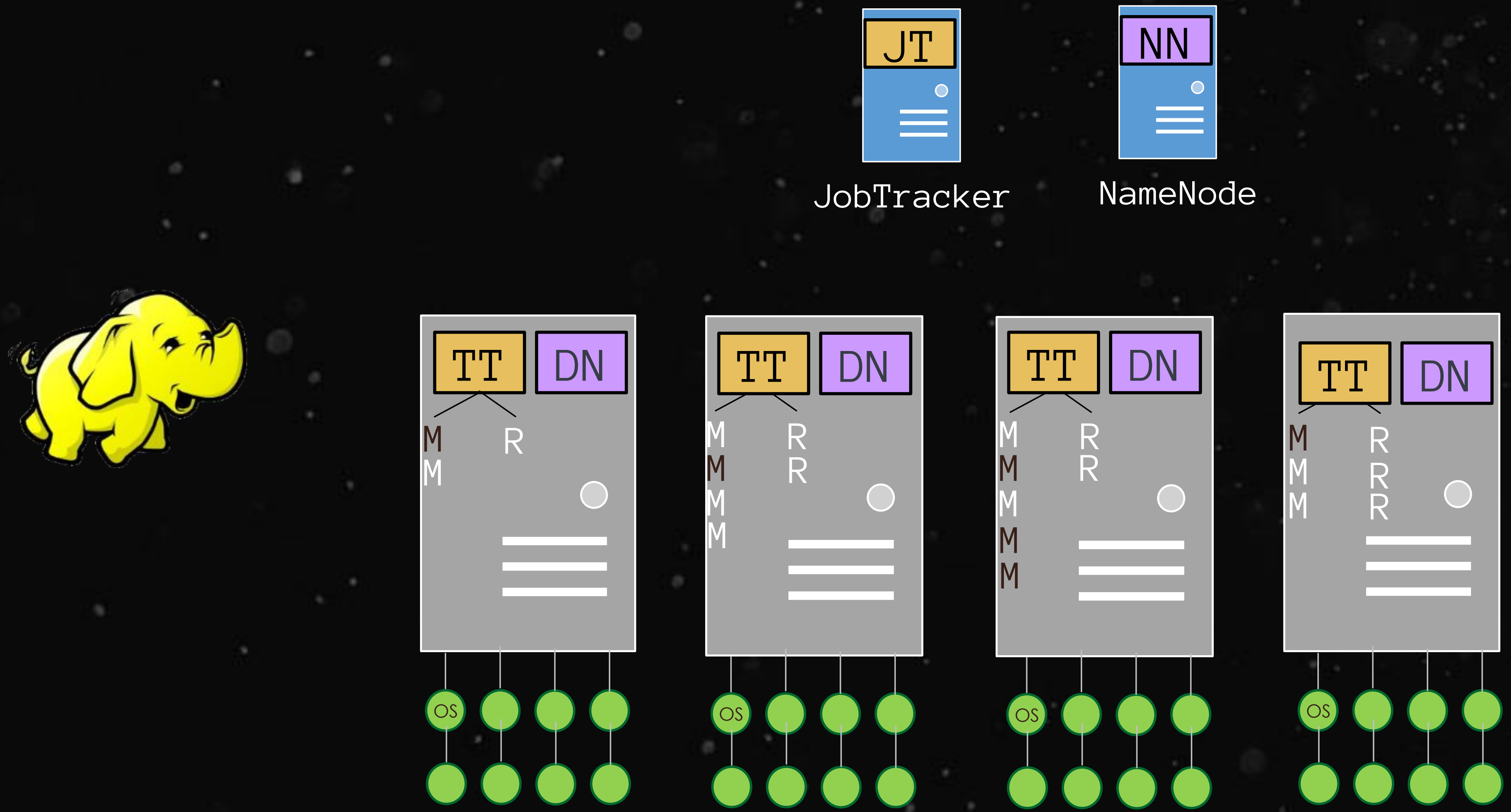


WHERE WILL YOU DEPLOY SPARK?

Survey completed by
58 out of 115 students



History: 2 MR APPS RUNNING



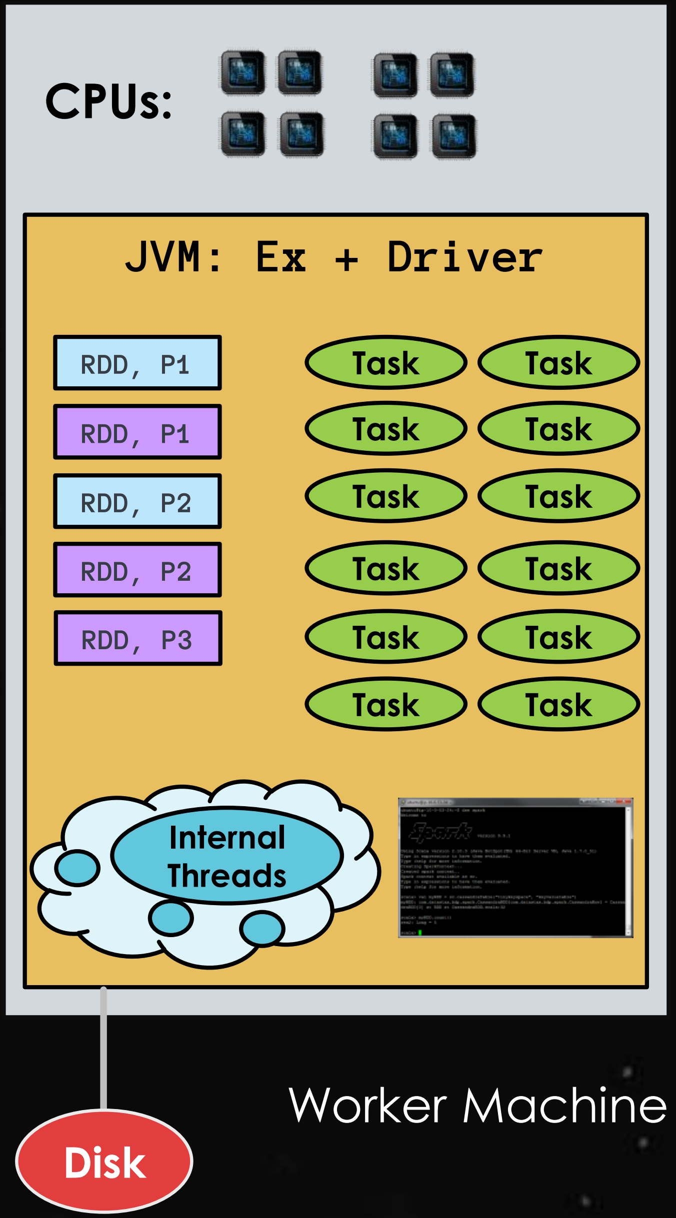
WAYS TO RUN SPARK

- Local
- Standalone Scheduler
- YARN
- Mesos

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
```

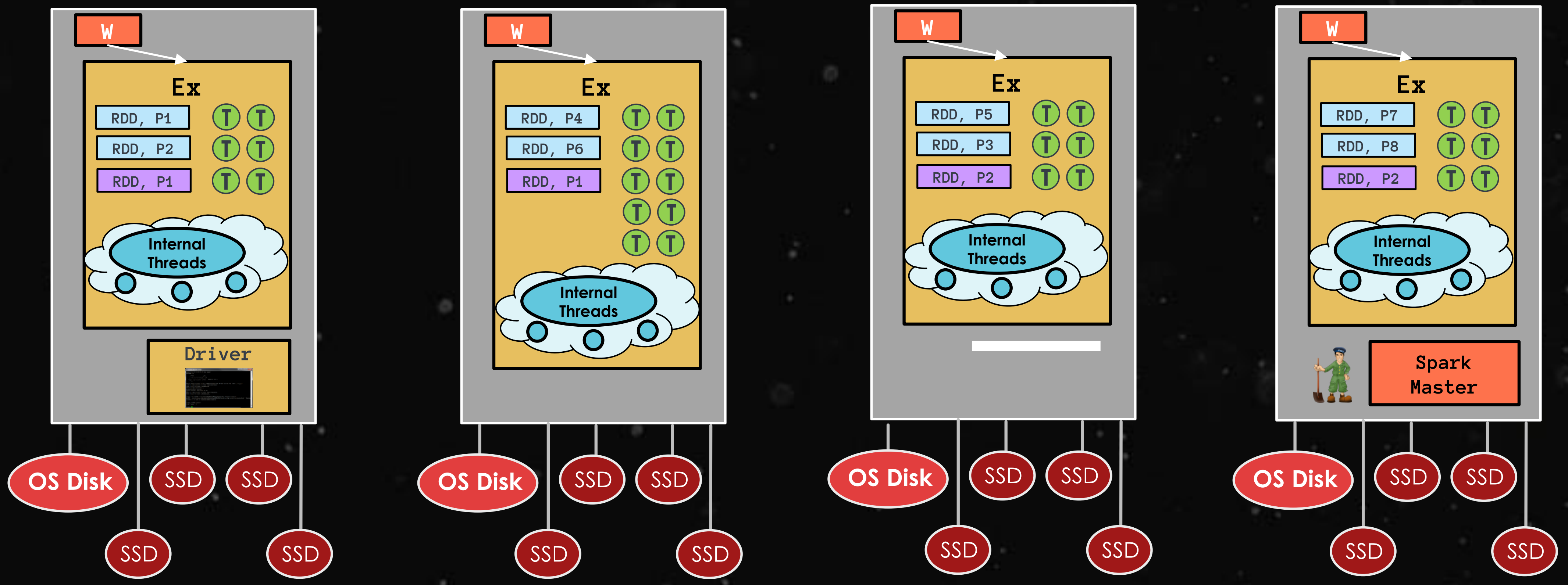
```
val conf = new SparkConf()
                 .setMaster("local[12]")
                 .setAppName(“MyFirstApp")
                 .set("spark.executor.memory", “3g")
val sc = new SparkContext(conf)
```

STANDALONE MODE

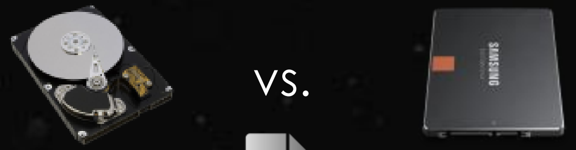
SPARK STANDALONE

different spark-env.sh

- SPARK_WORKER_CORES




```
> ./bin/spark-submit --name "SecondApp"
                     --master spark://host4:port1
                     myApp.jar
```



spark-env.sh

- SPARK_LOCAL_DIRS

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

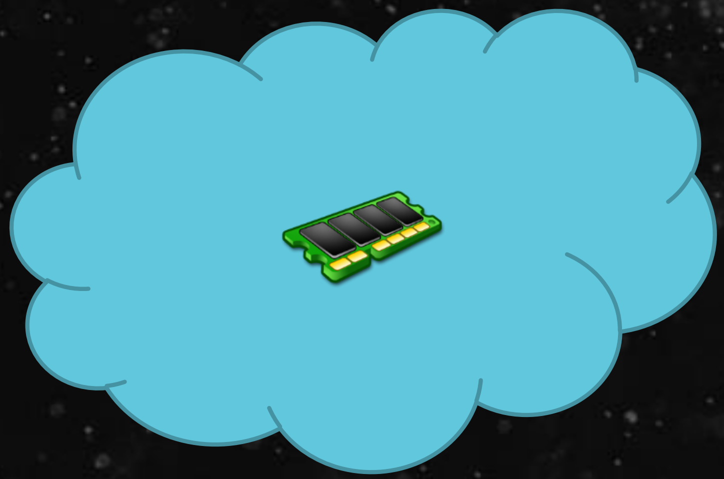
`spark-submit` provides a uniform interface for submitting jobs across all cluster managers

```
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`

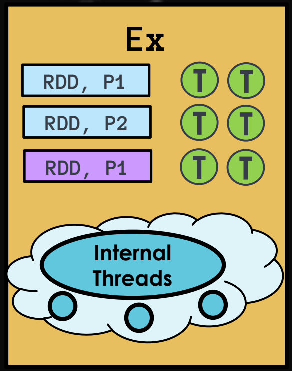
Value	Explanation
spark://host:port	Connect to a Spark Standalone master at the specified port. By default Spark Standalone master's listen on port 7077 for submitted jobs.
mesos://host:port	Connect to a Mesos cluster master at the specified port. By default Mesos masters listen on port 5050 for submitted 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

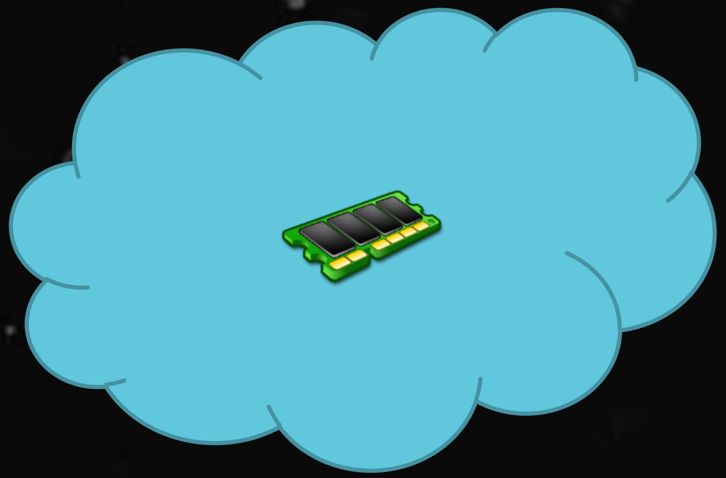


Recommended to use at most only 75% of a machine's memory for Spark

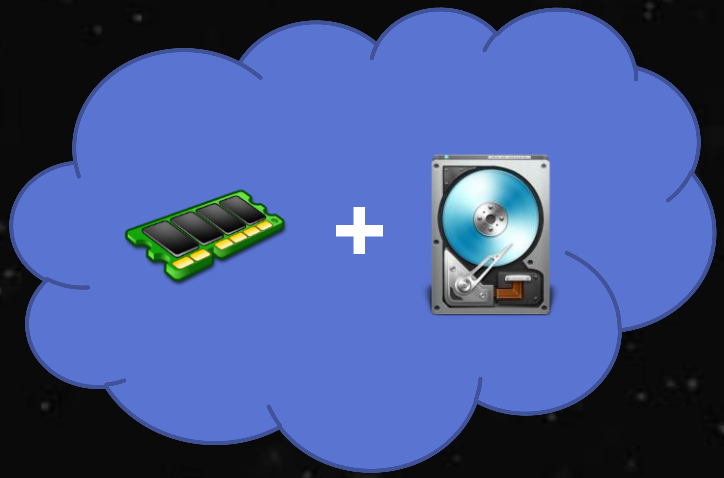
Minimum Executor heap size should be 8 GB

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

Memory usage is greatly affected by storage level and serialization format



Vs.



Persistence	description
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM and spill to disk
MEMORY_ONLY_SER	Store RDD as serialized Java objects (one byte array per partition)
MEMORY_AND_DISK_SER	Spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed
DISK_ONLY	Store the RDD partitions only on disk
MEMORY_ONLY_2, MEMORY_AND_DISK_2	Same as the levels above, but replicate each partition on two cluster nodes
OFF_HEAP	Store RDD in serialized format in Tachyon


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

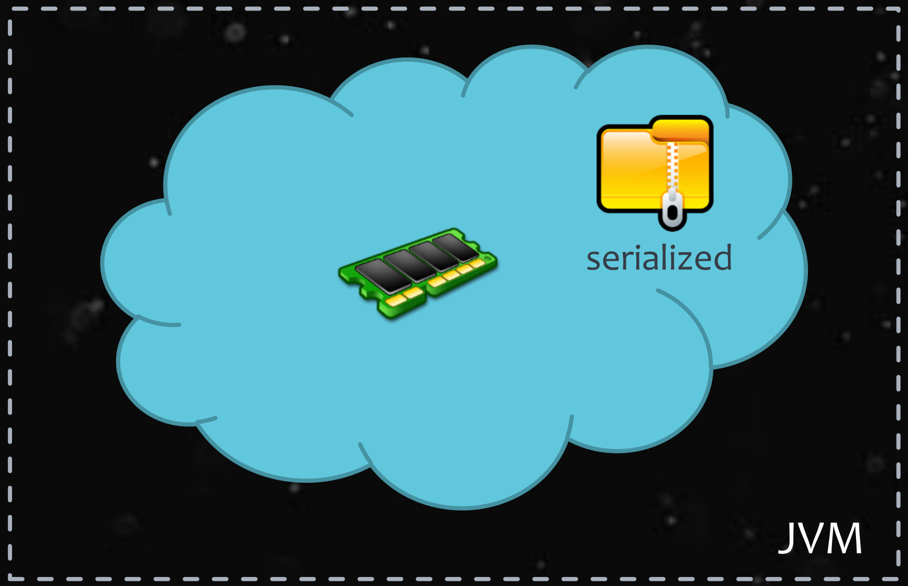
most CPU-efficient option

Stages Storage Environment Executors

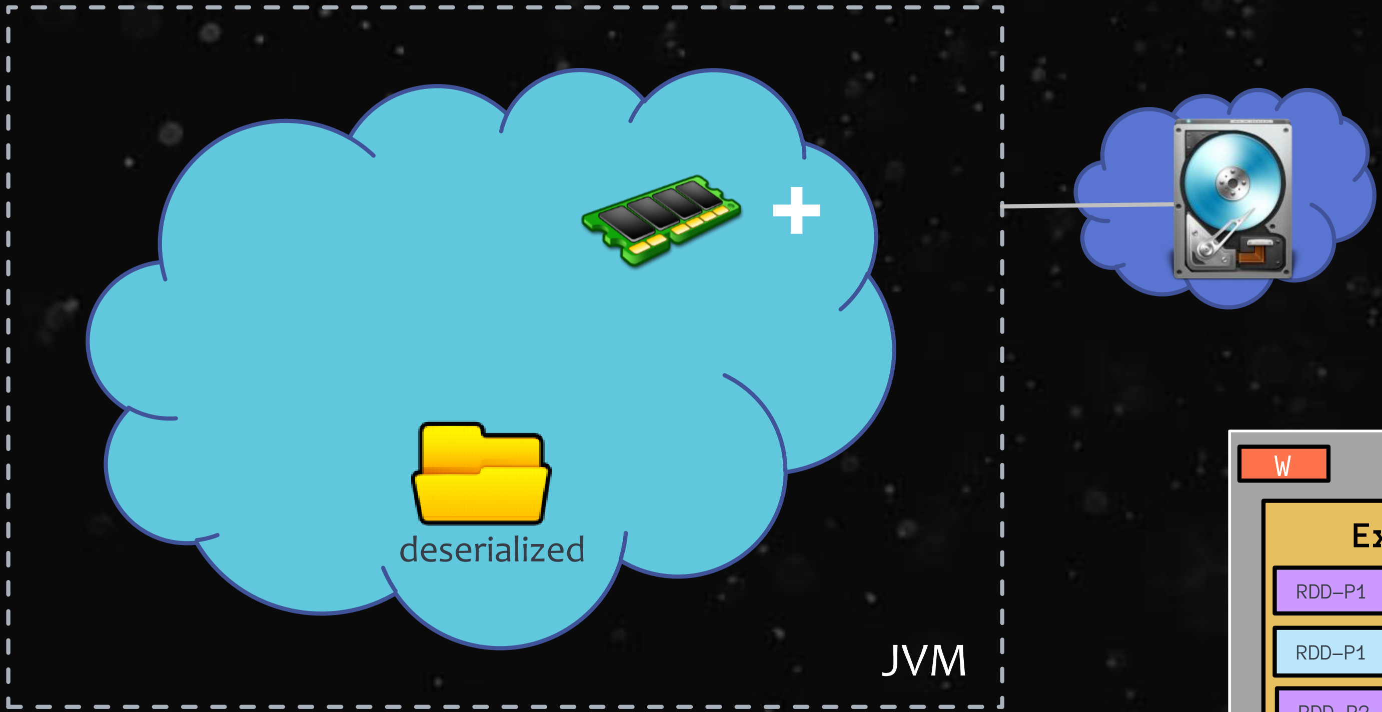
Spark shell application UI

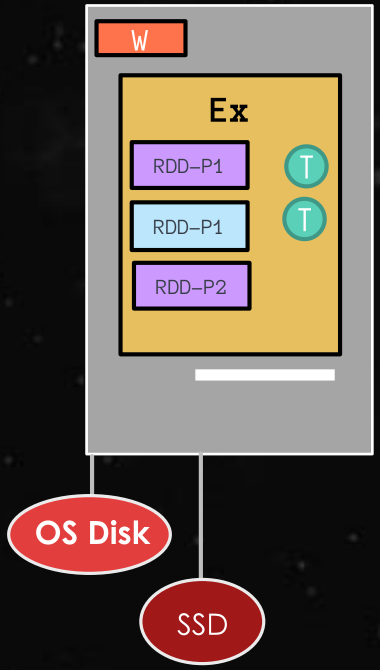
Storage

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

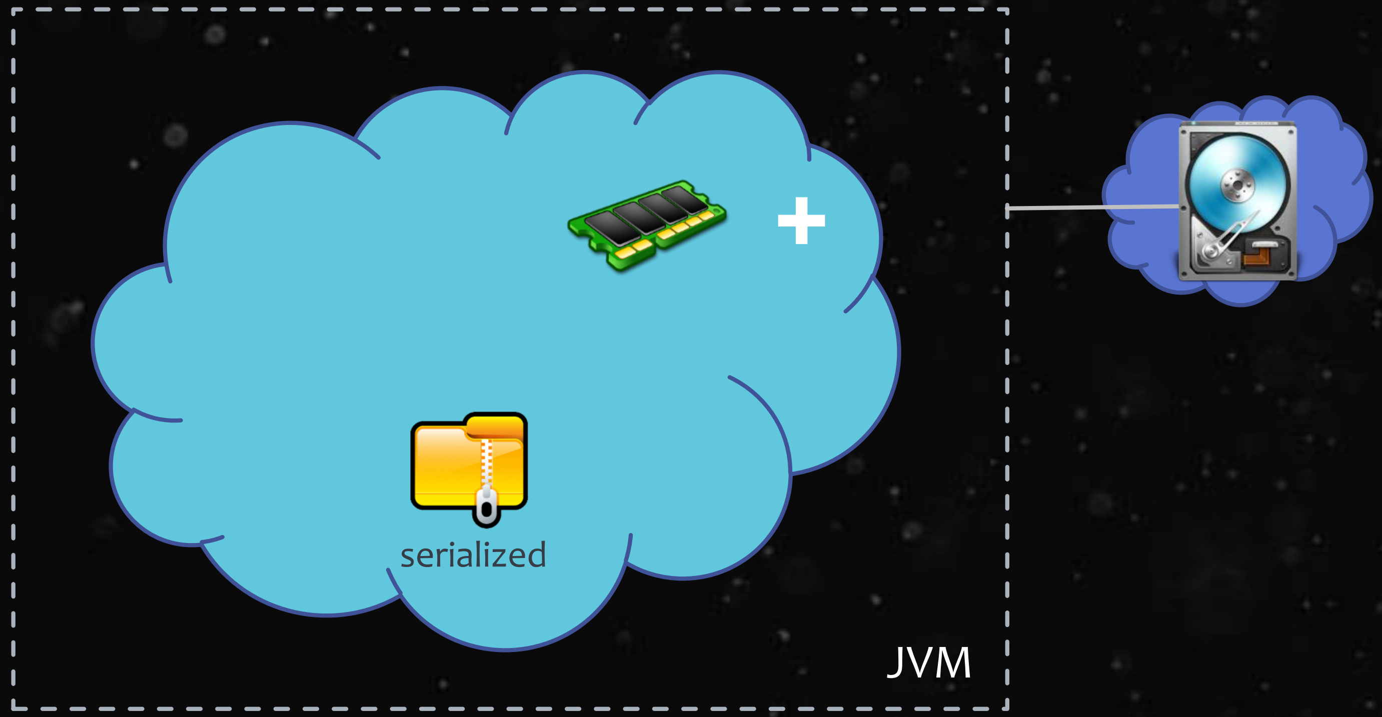



```
RDD.persist(MEMORY_ONLY_SER)
```



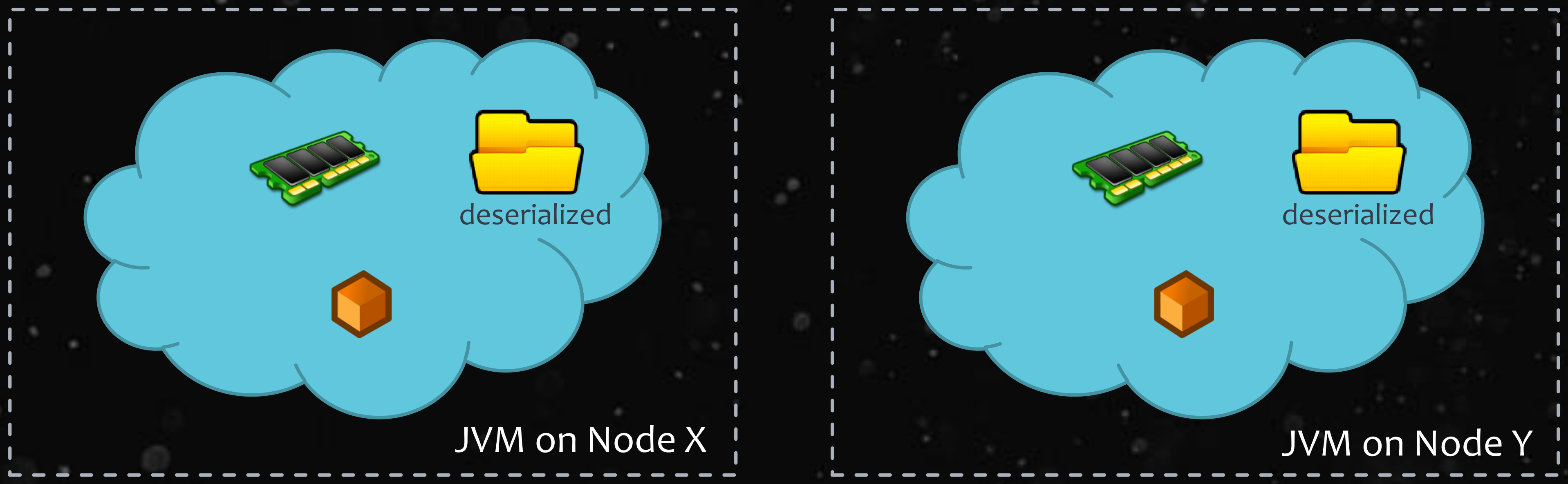


`.persist(MEMORY_AND_DISK)`

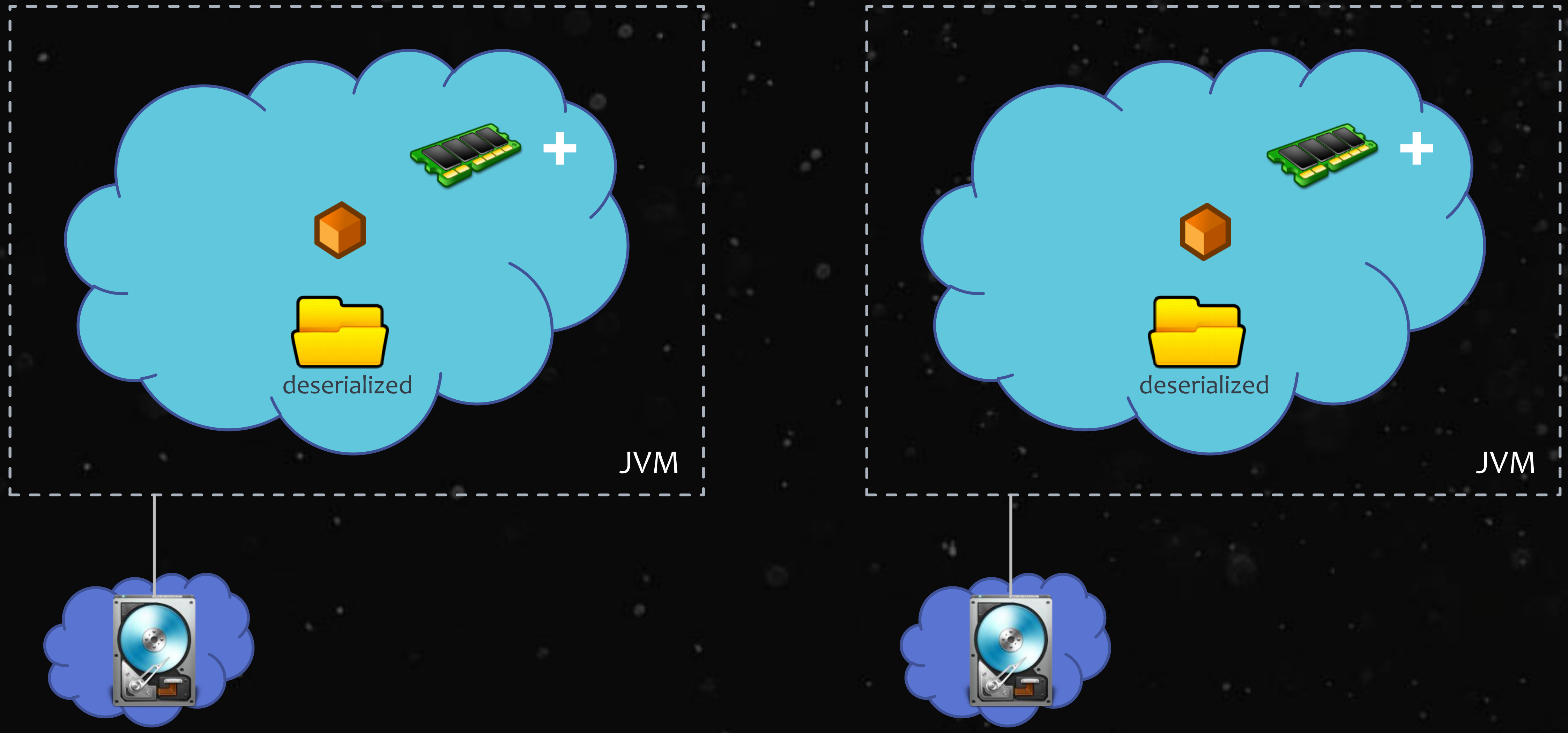



```
.persist(MEMORY_AND_DISK_SER)
```

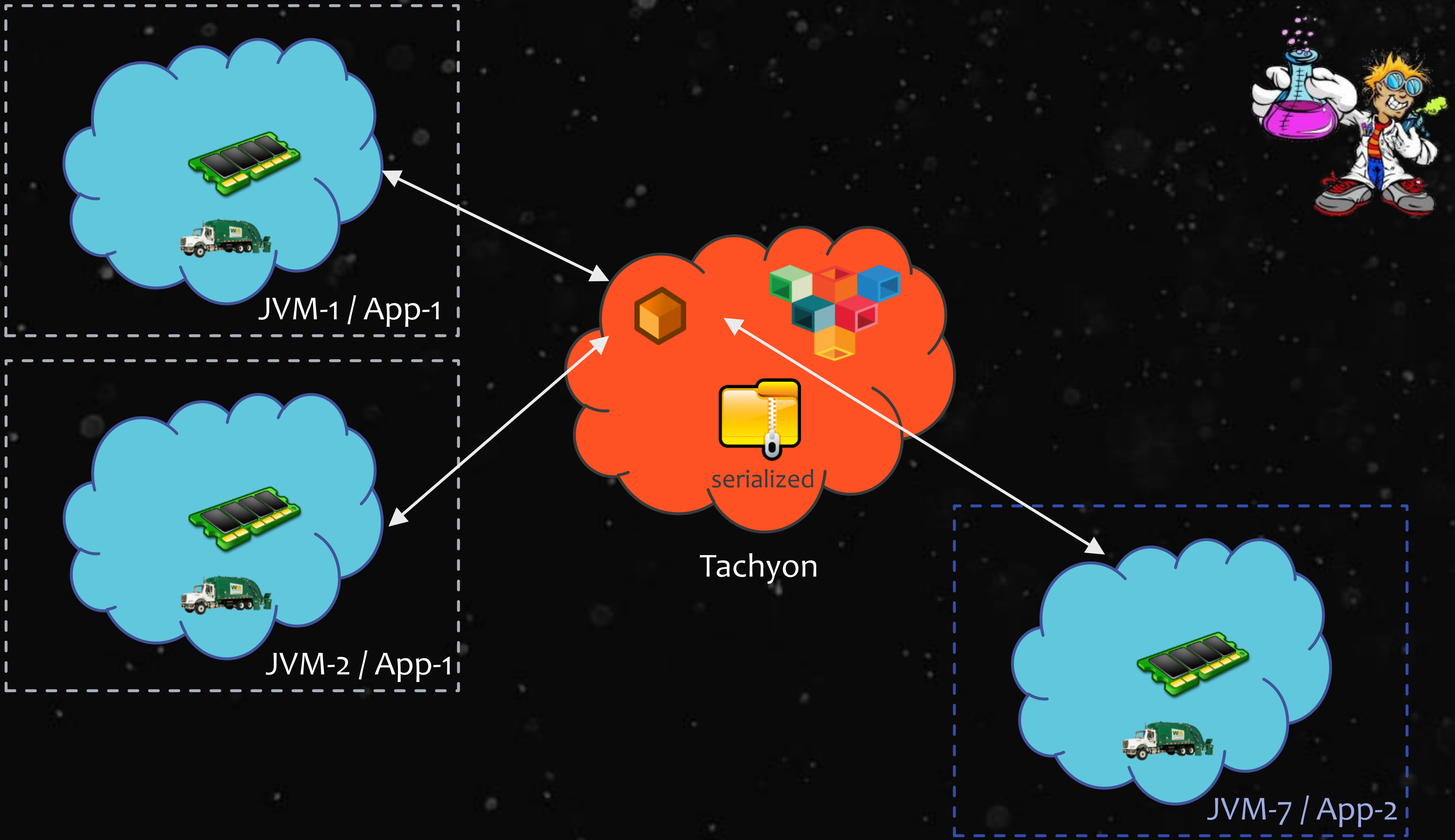
`.persist(DISK_ONLY)`



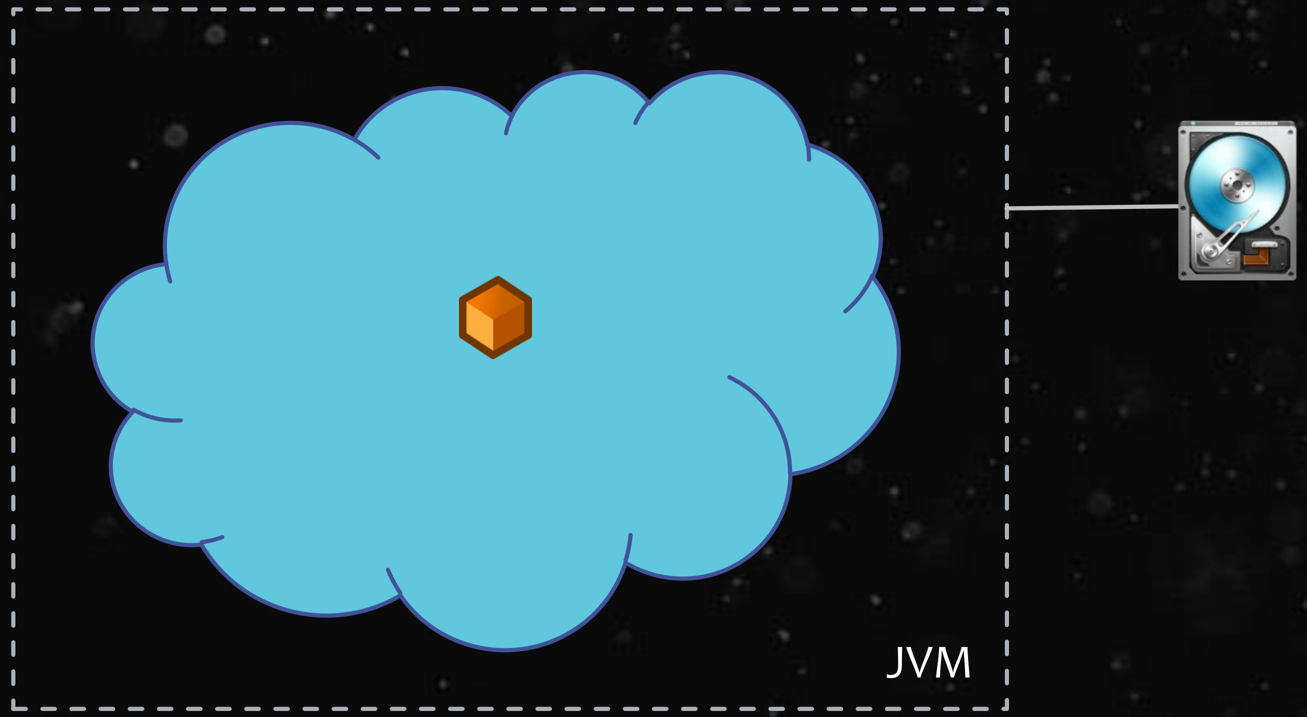
RDD.persist(MEMORY_ONLY_2)



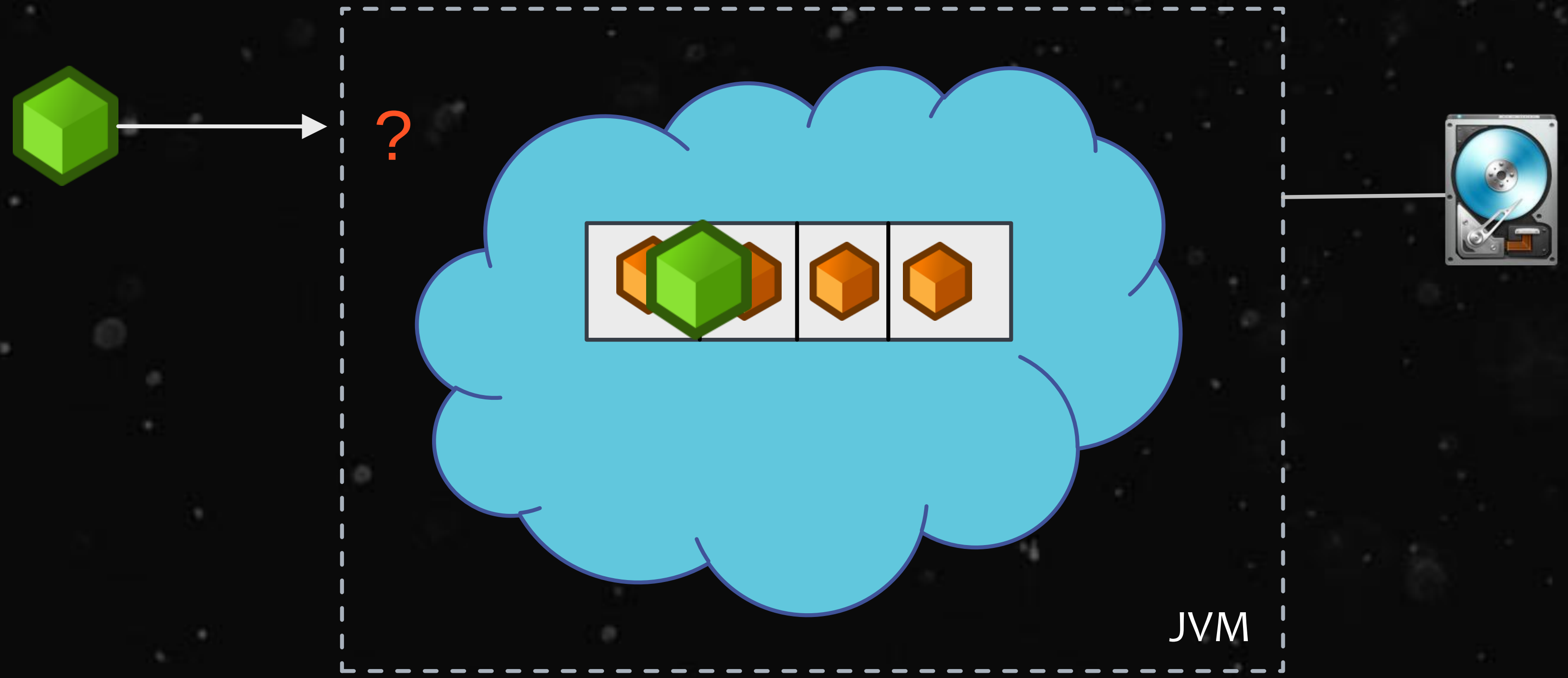

```
.persist(MEMORY_AND_DISK_2)
```



`.persist(OFF_HEAP)`




```
.unpersist()
```

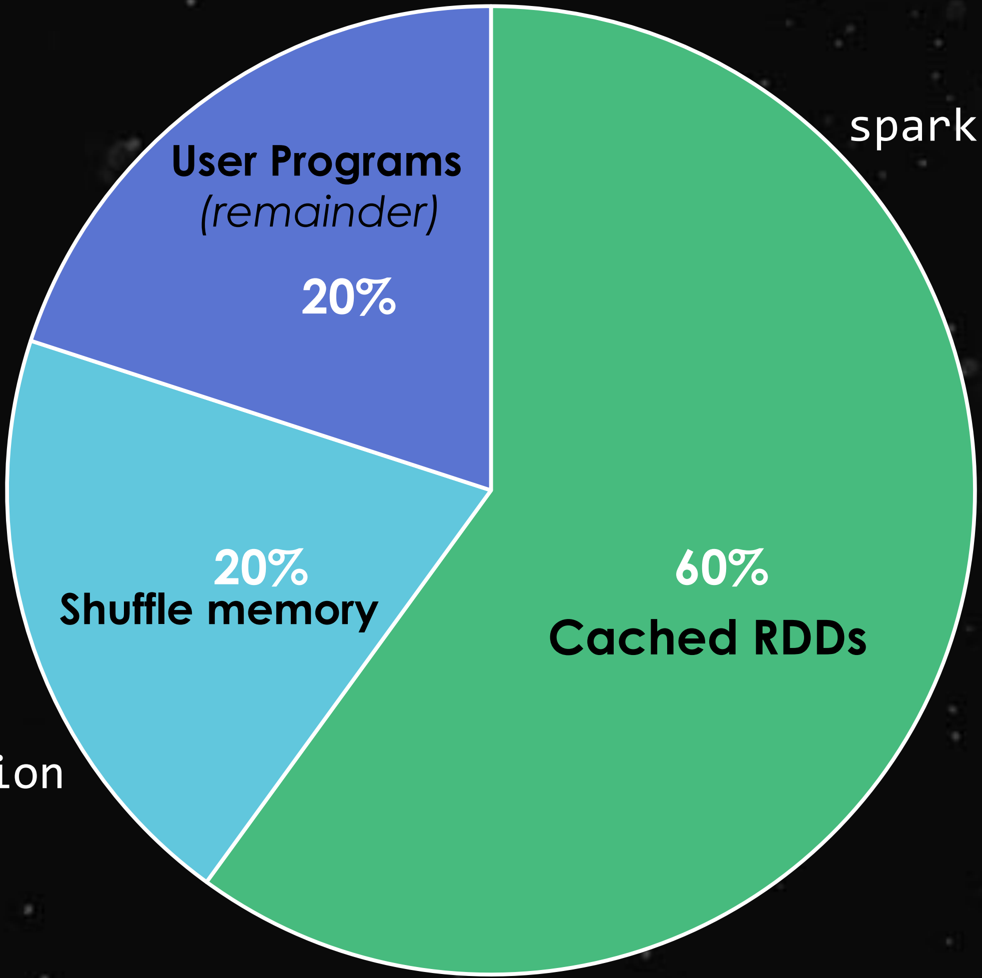

?
JVM

Remember!

Intermediate data is automatically persisted during shuffle operations

Default Memory Allocation in Executor JVM

spark.storage.memoryFraction



spark.shuffle.memoryFraction

MEMORY

Spark uses memory for:

RDD Storage: when you call .persist() or .cache(). Spark will limit the amount of memory used when caching to a certain fraction of the JVM's overall heap, set by `spark.storage.memoryFraction`

Shuffle and aggregation buffers: When performing shuffle operations, Spark will create intermediate buffers for storing shuffle output data. These buffers are used to store intermediate results of aggregations in addition to buffering data that is going to be directly output as part of the shuffle.

User code: Spark executes arbitrary user code, so user functions can themselves require substantial memory. For instance, if a user application allocates large arrays or other objects, these will content for overall memory usage. User code has access to everything "left" in the JVM heap after the space for RDD storage and shuffle storage are allocated.

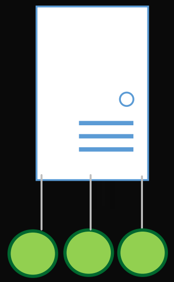
DATA SERIALIZATION

databricks

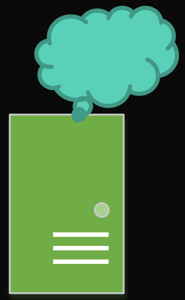
SERIALIZATION

Serialization is used when:

Transferring data over the network



Spilling data to disk



Caching to memory serialized

Broadcasting variables

Java serialization vs. Kryo serialization

Java 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

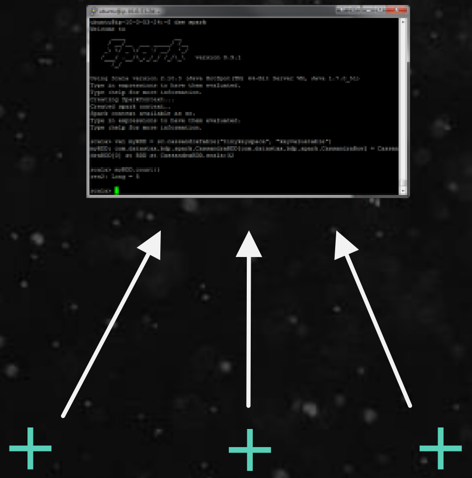
Kryo serialization

- 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")
```

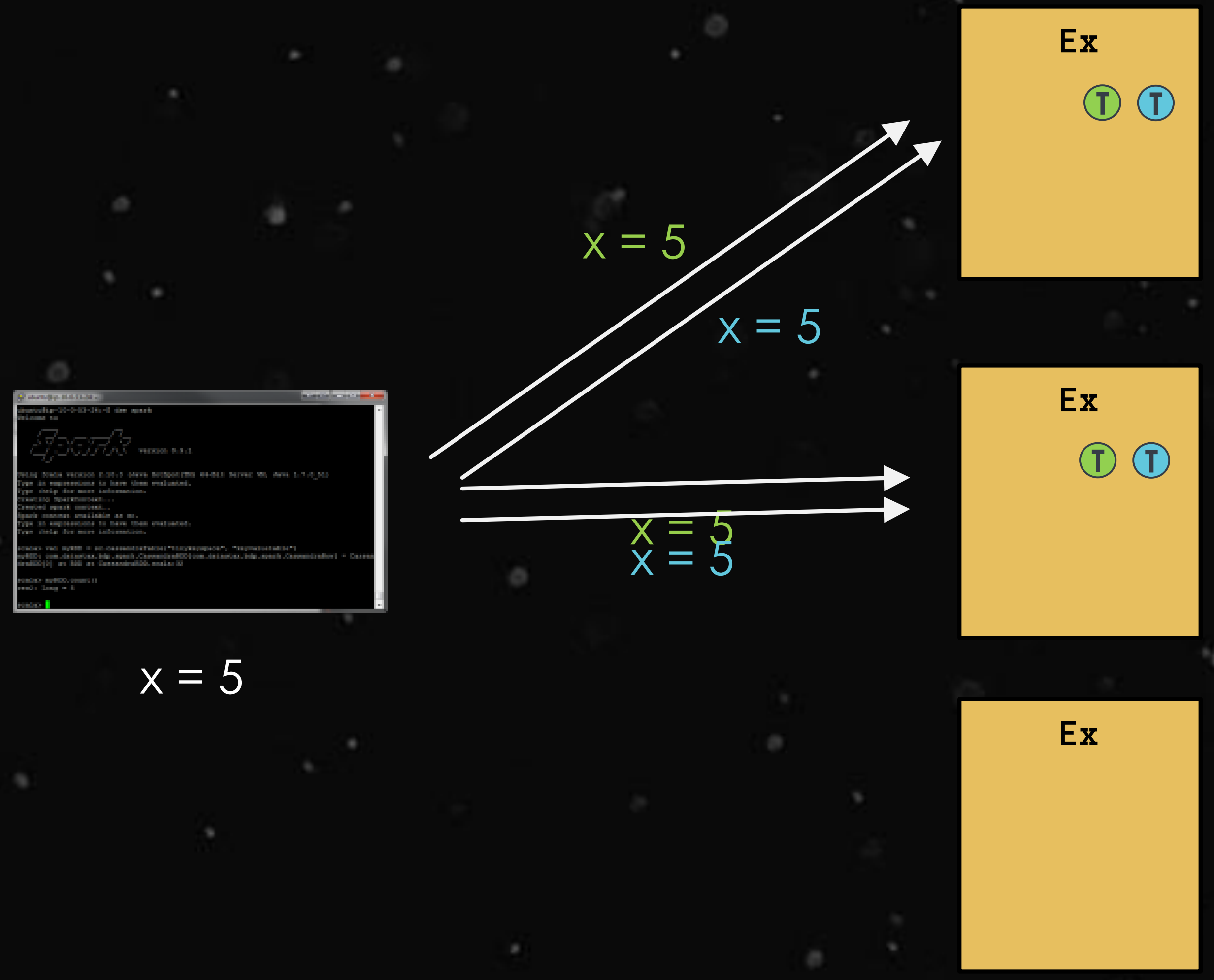
BROADCAST VARIABLES

&



ACCUMULATORS

databricks



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

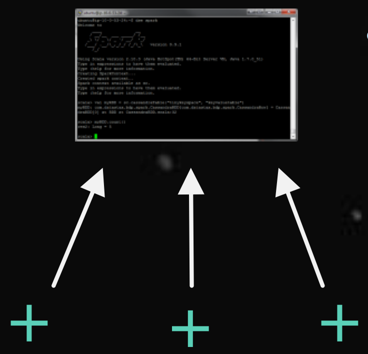
USE CASES:

- 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
```

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
```

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

Spark STREAMING

databricks

- Scalable
- High-throughput
- Fault-tolerant

TCP socket

Kafka

Flume

HDFS

S3

Kinesis

Twitter

Spark STREAMING

HDFS

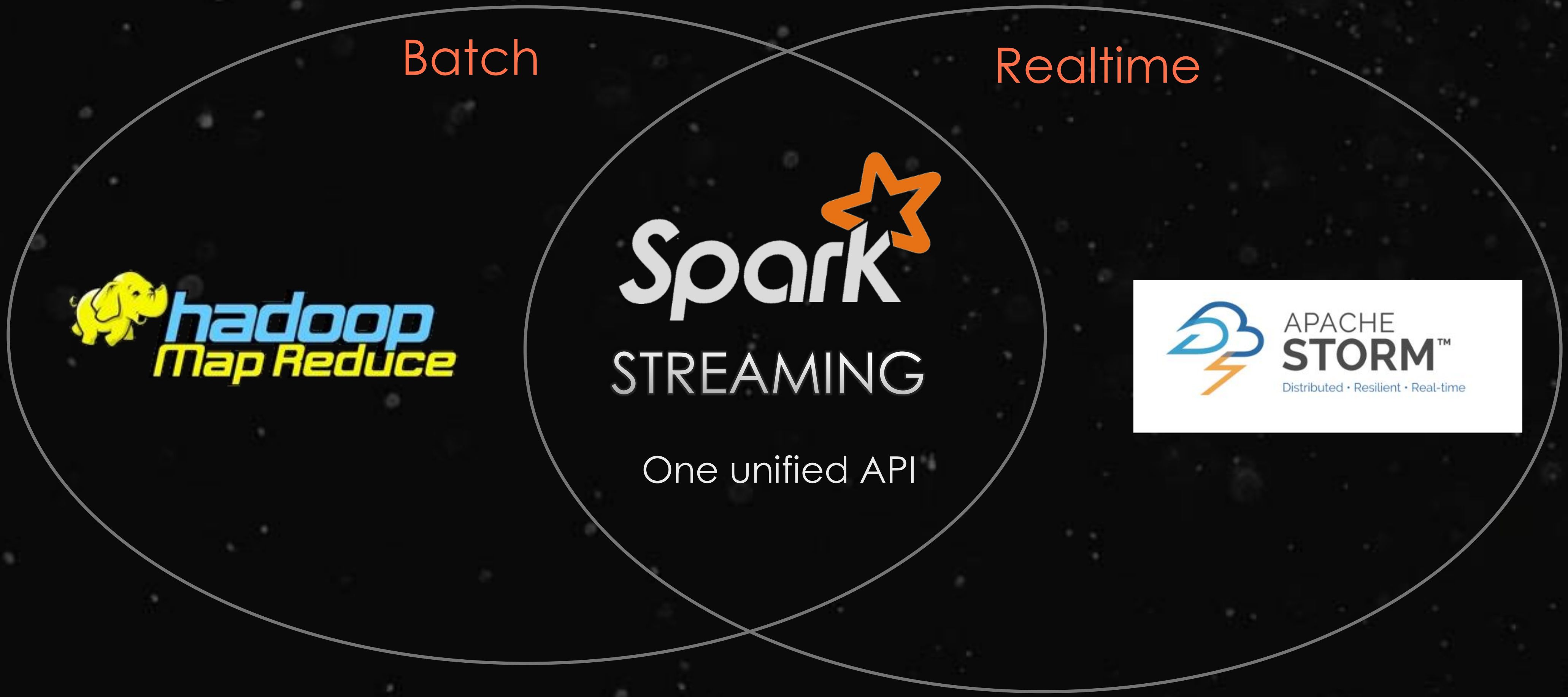
Cassandra

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

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)

Spark STREAMING

Page views

Kafka for buffering

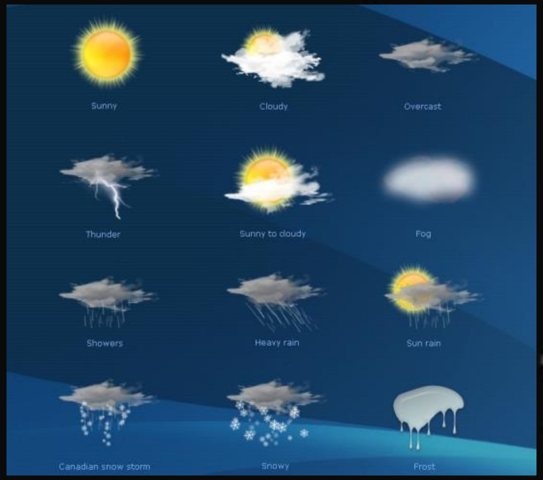
Spark for processing

USE CASES (Anomaly Detection)

Smart meter readings

Join 2 live data sources

Spark STREAMING



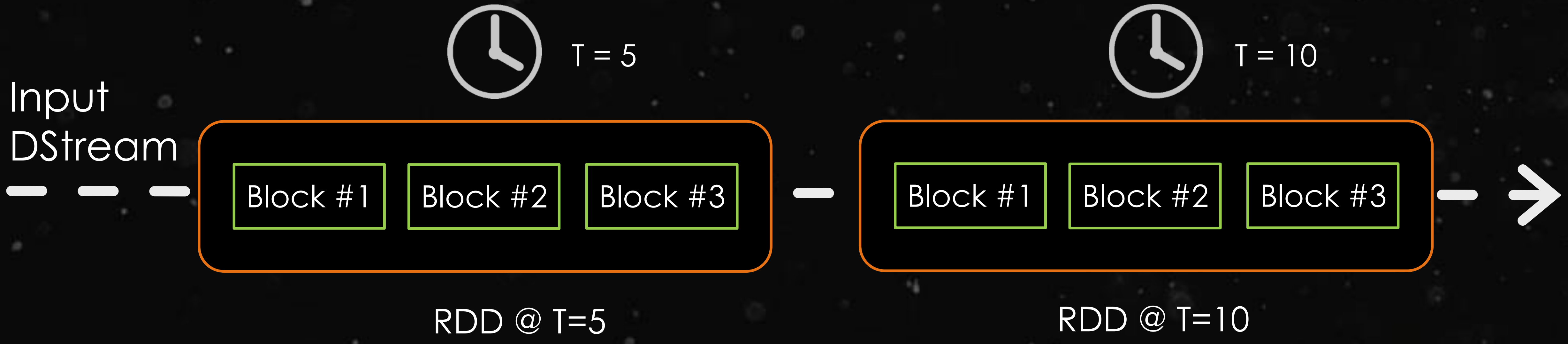
Live weather data


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

DSTREAM

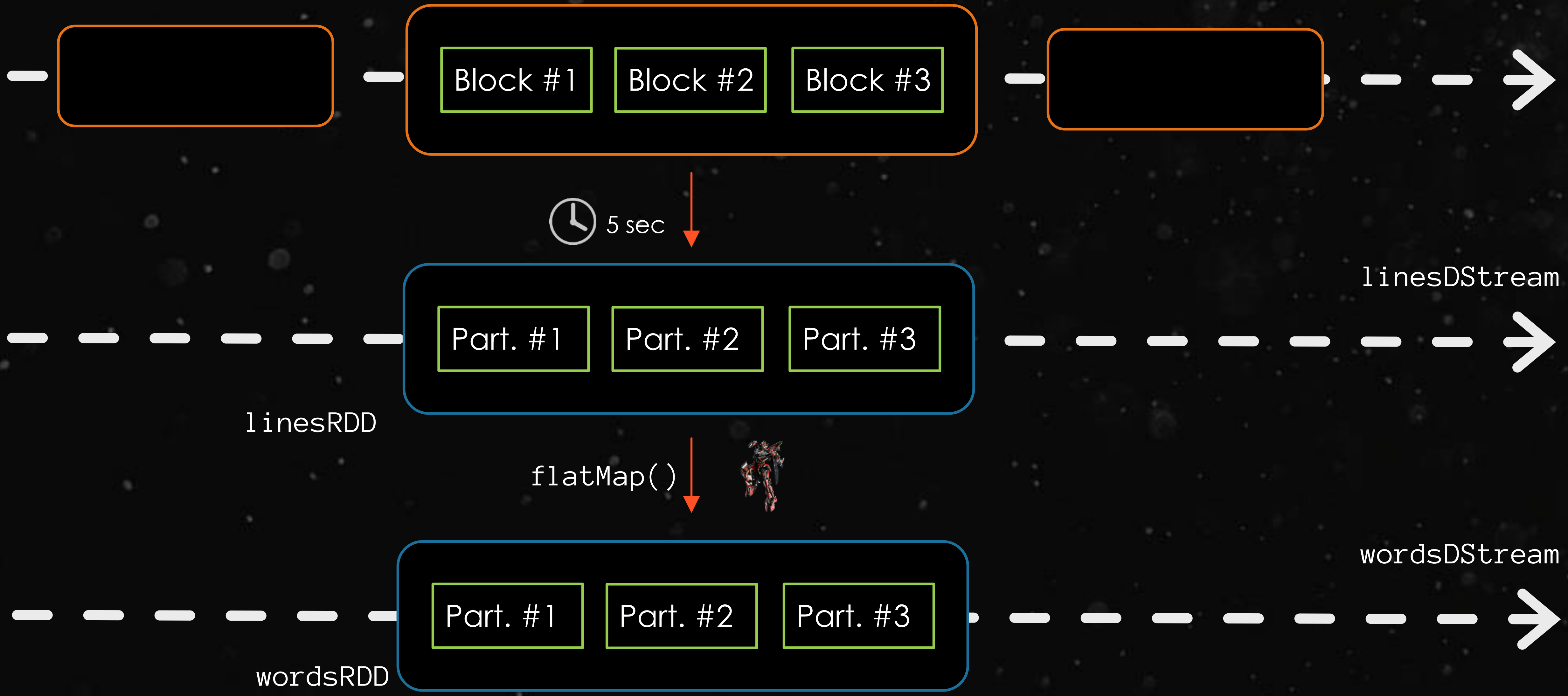
(Discretized Stream)

Batch interval = 5 seconds

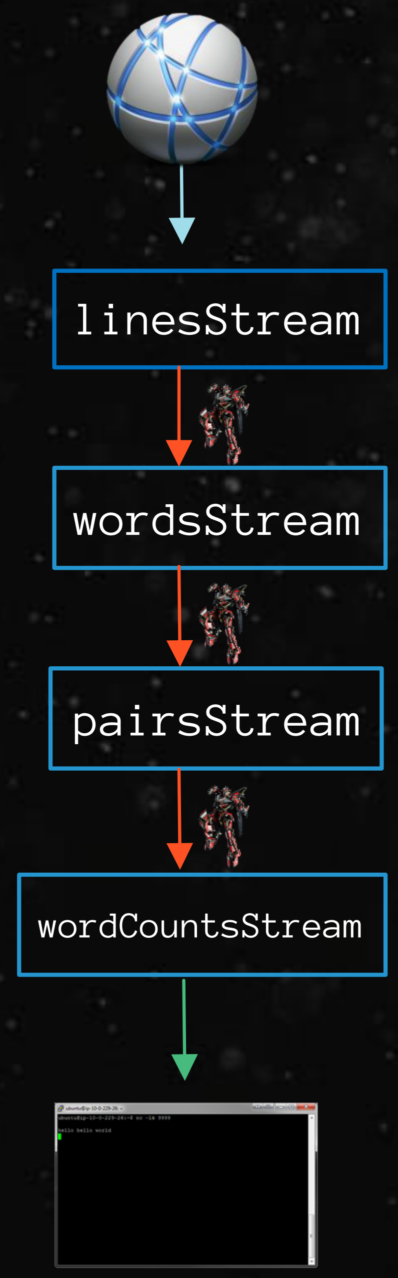


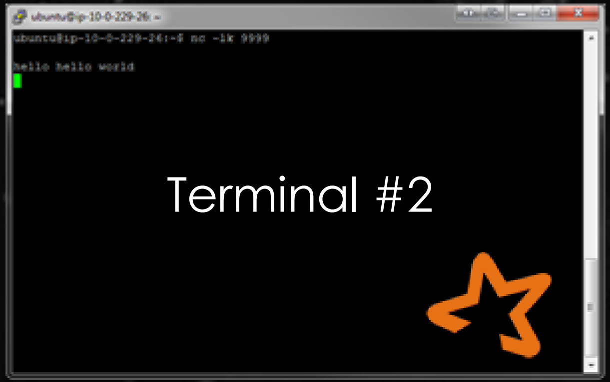
One RDD is created every 5 seconds

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
```

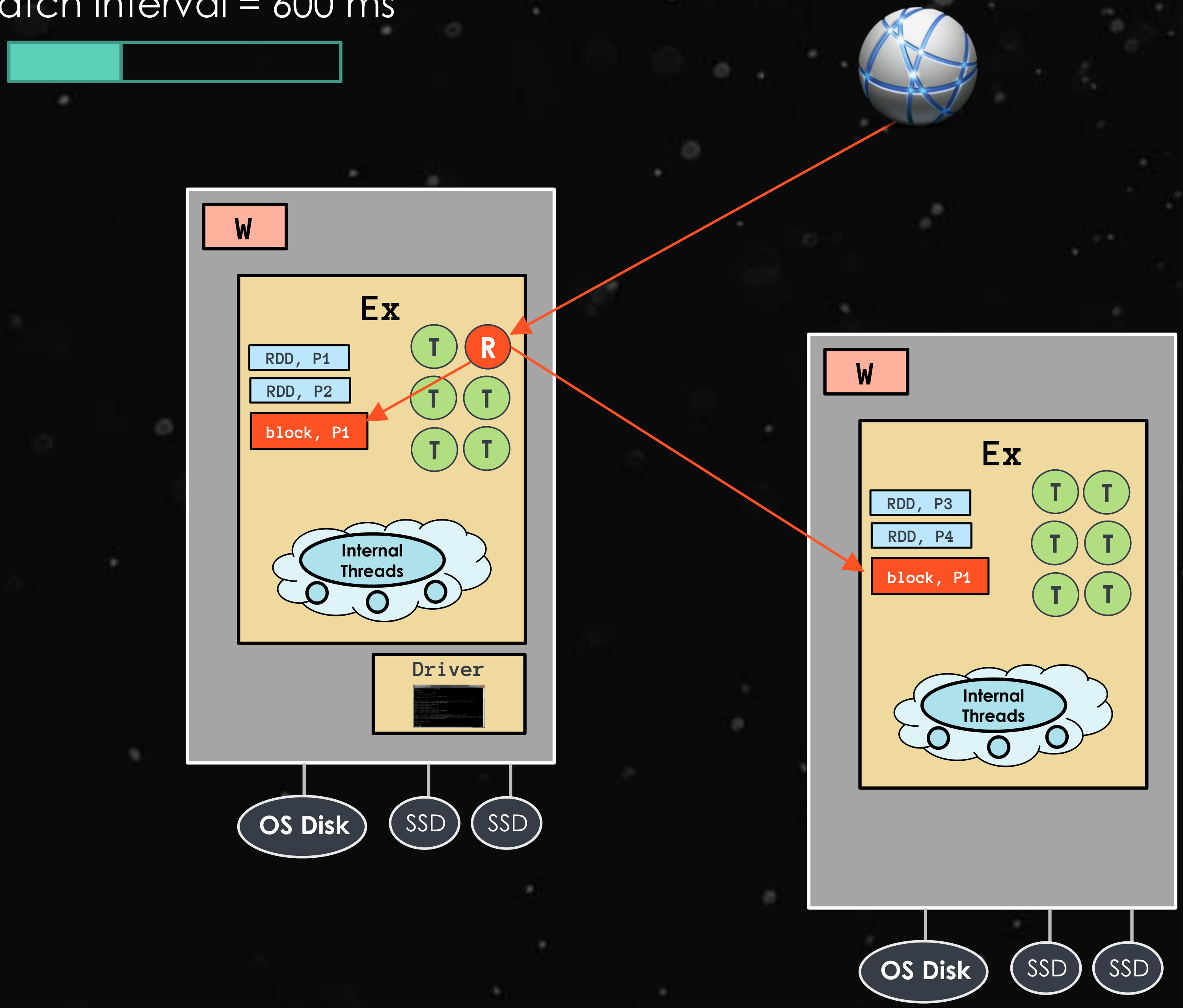


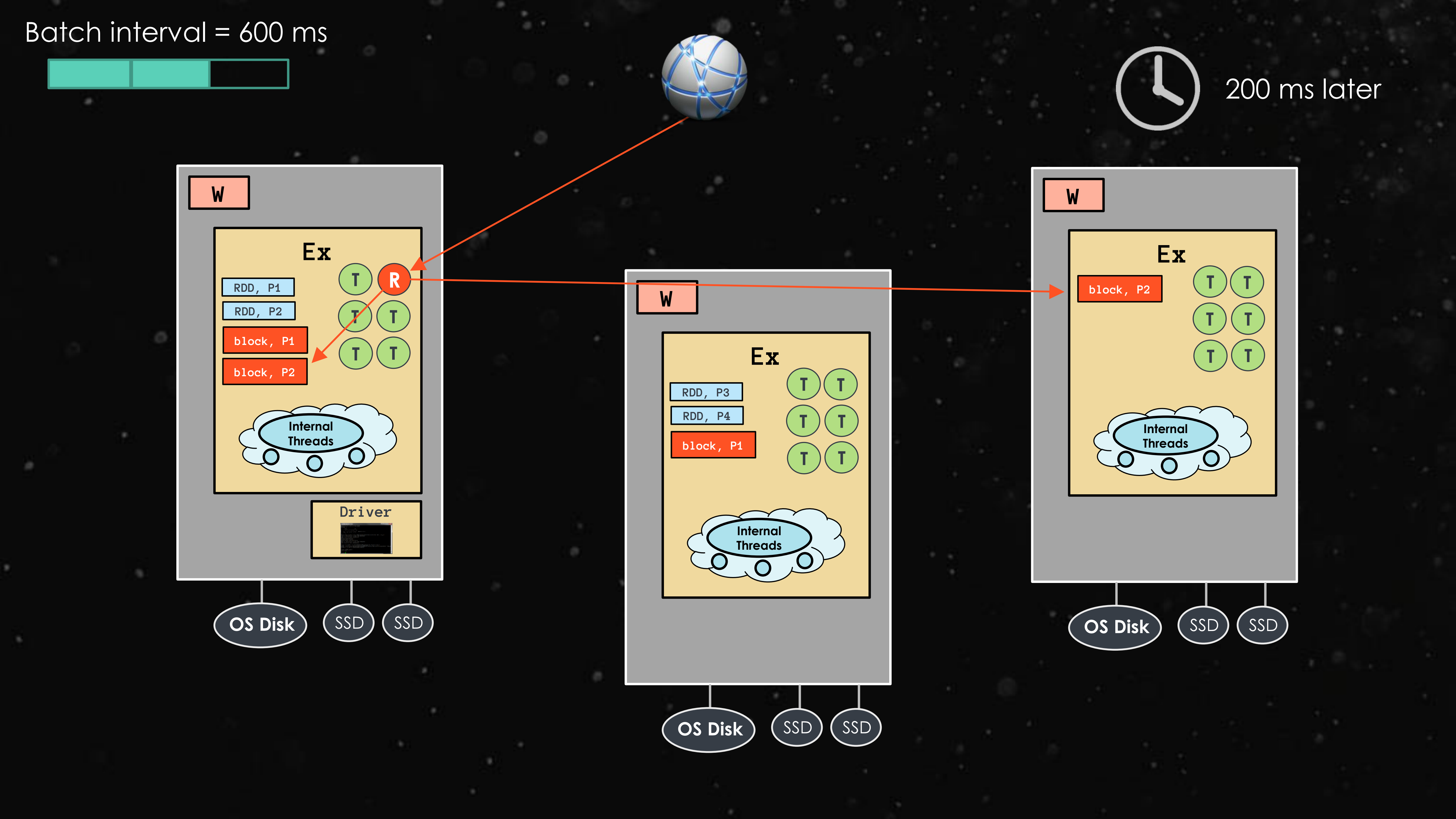



```
$ nc -lk 9999

hello hello world
```

```
$ ./network_wordcount.py localhost 9999

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

Batch interval = 600 ms

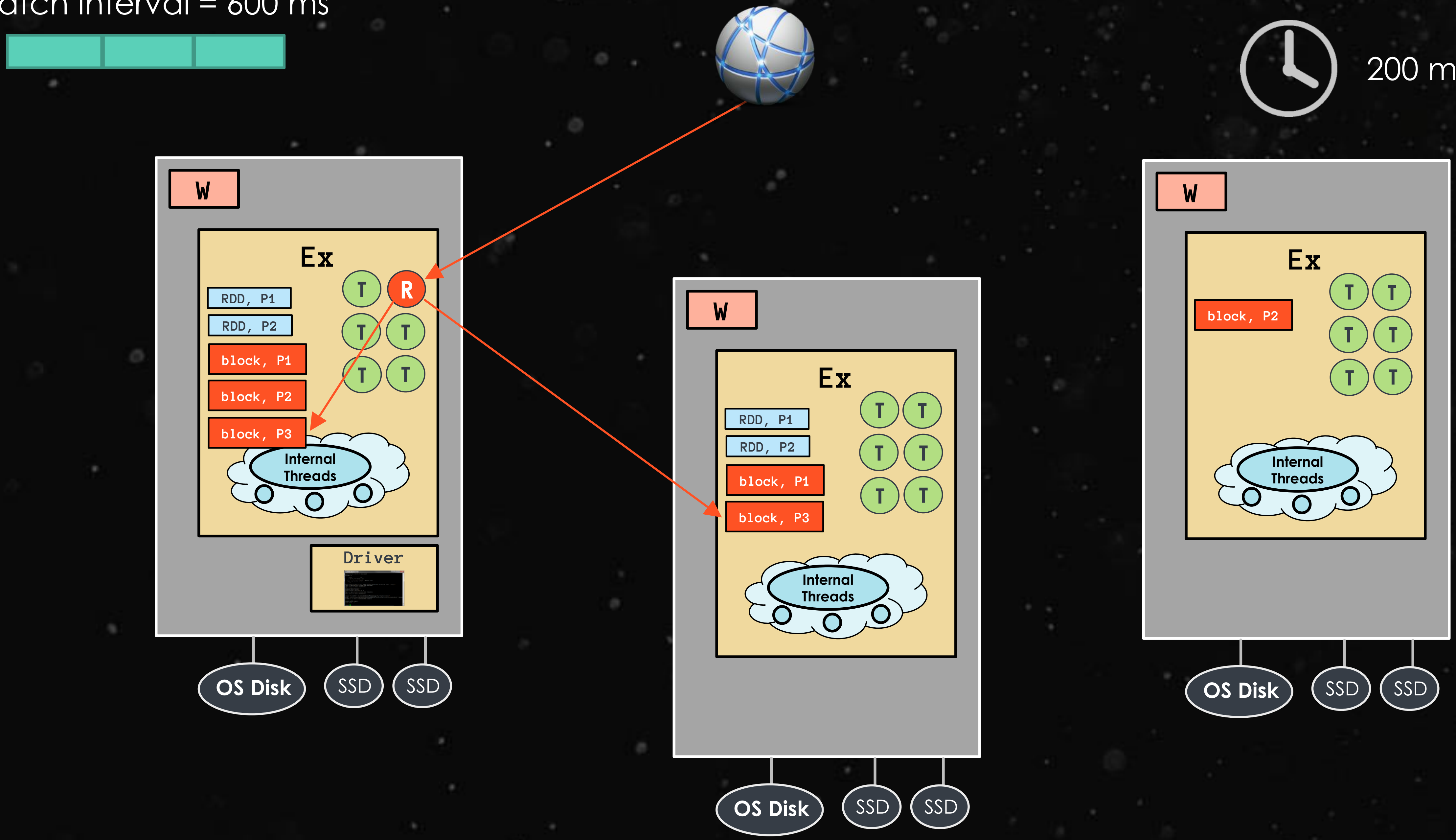




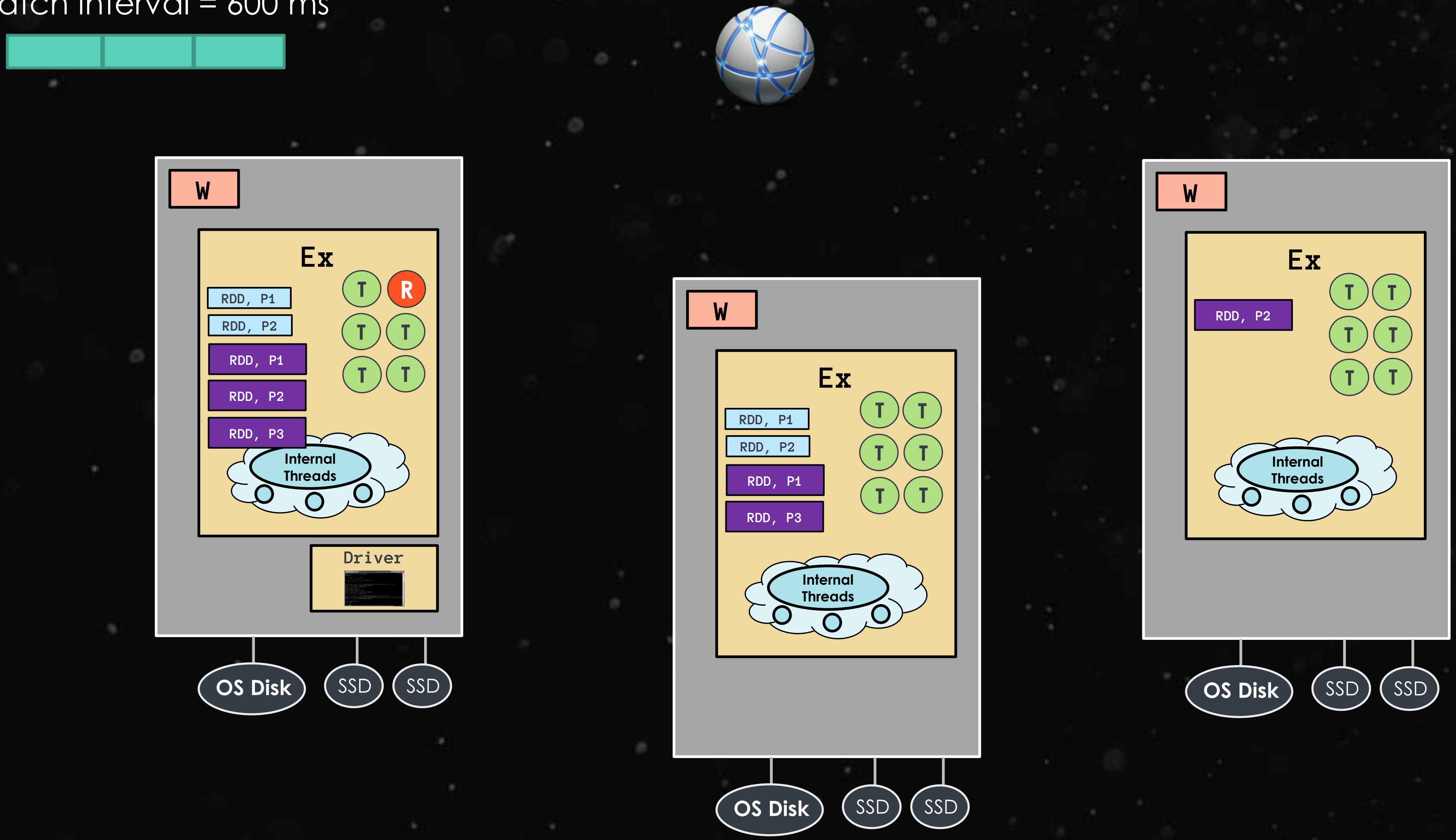
Batch interval = 600 ms
200 ms later
W
Ex
RDD, P1
RDD, P2
block, P1
block, P2
T
R
Internal Threads
Driver
OS Disk
SSD
SSD
W
Ex
RDD, P3
RDD, P4
block, P1
Internal Threads
OS Disk
SSD
SSD
W
Ex
block, P2
Internal Threads
OS Disk
SSD
SSD

Batch interval = 600 ms

200 ms later



Batch interval = 600 ms

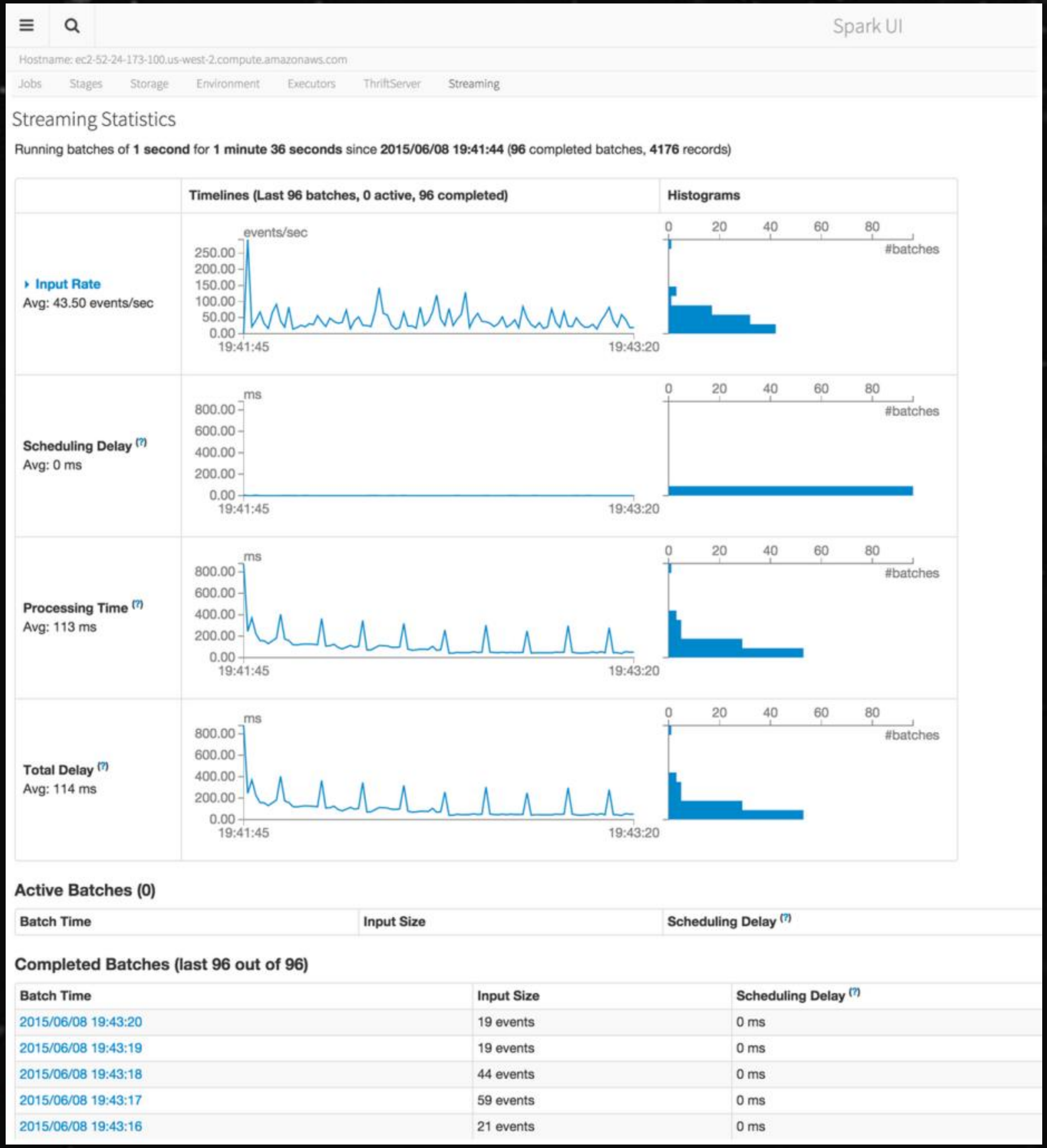


Batch interval = 600 ms

Spark 1.4.0

NEW

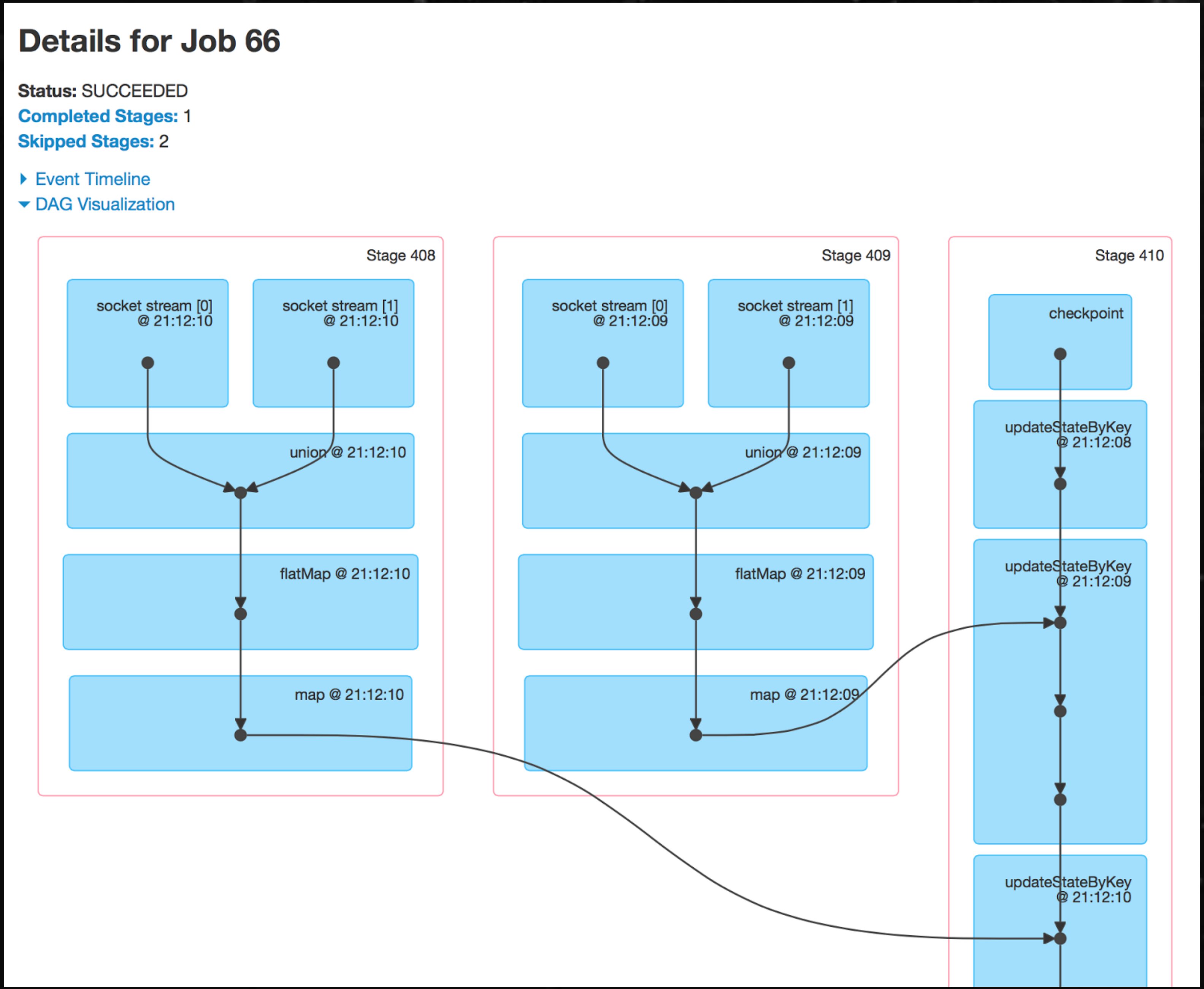
New UI for Streaming

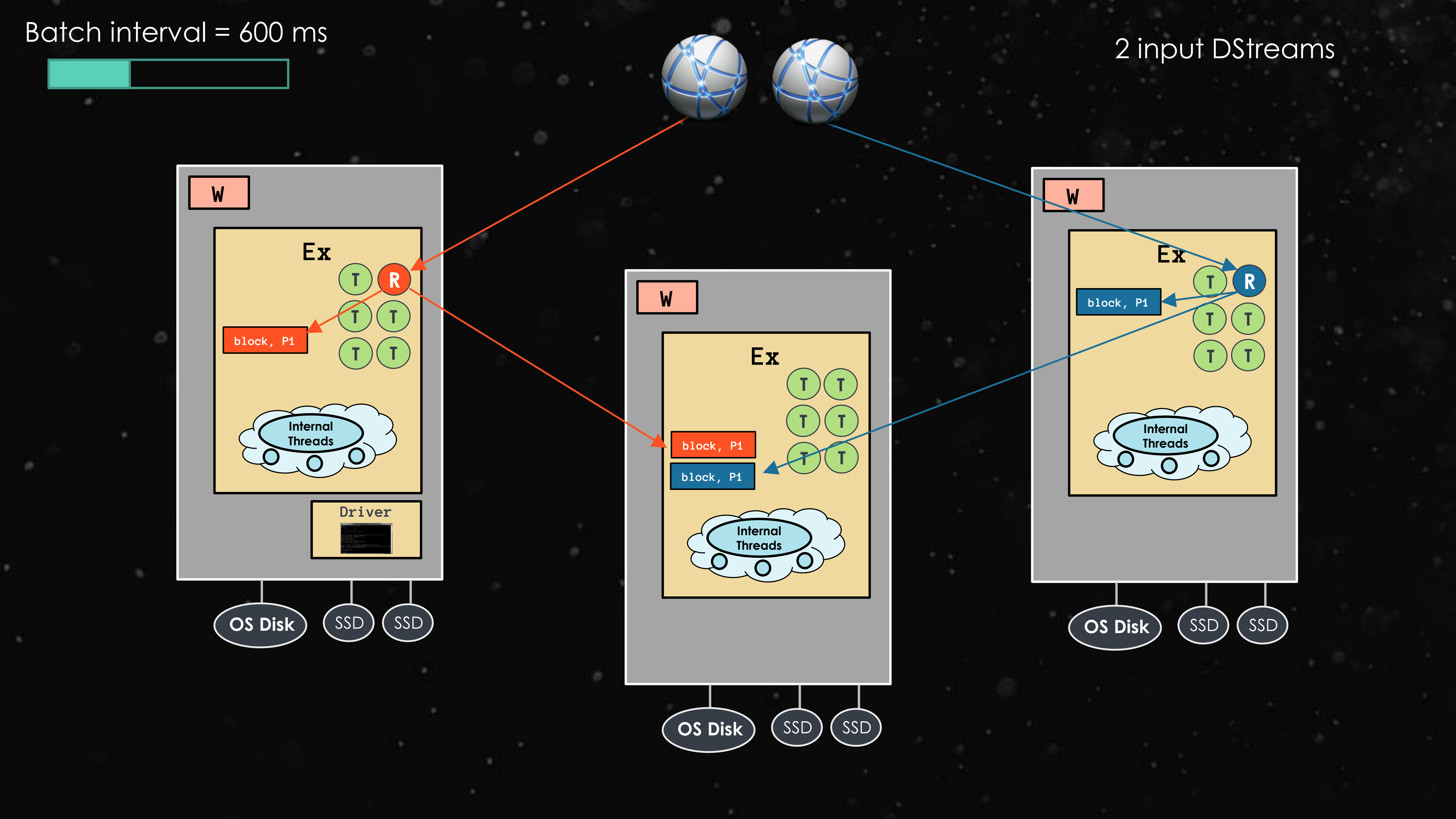


Spark 1.4.0

NEW

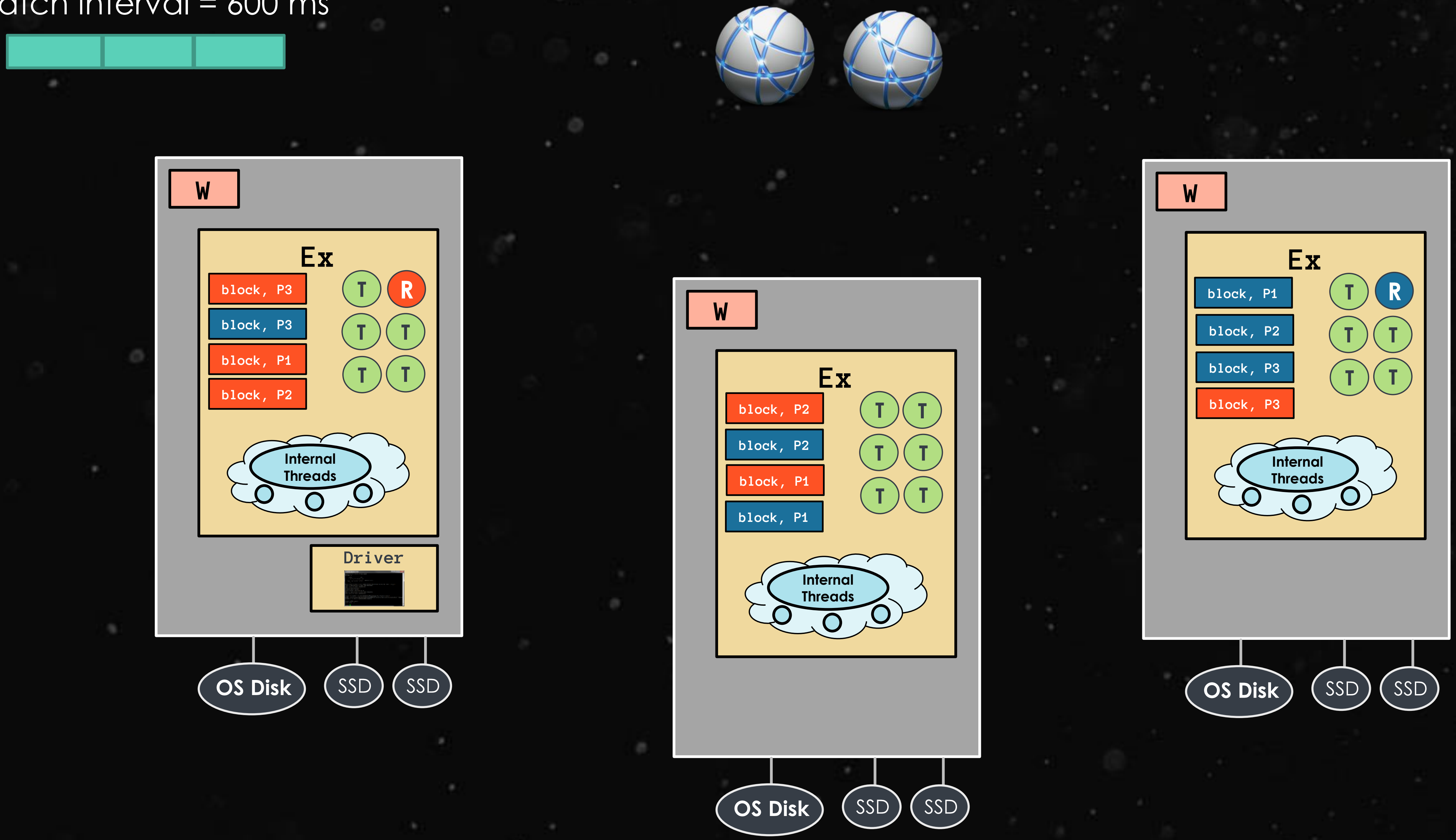
DAG Visualization for Streaming

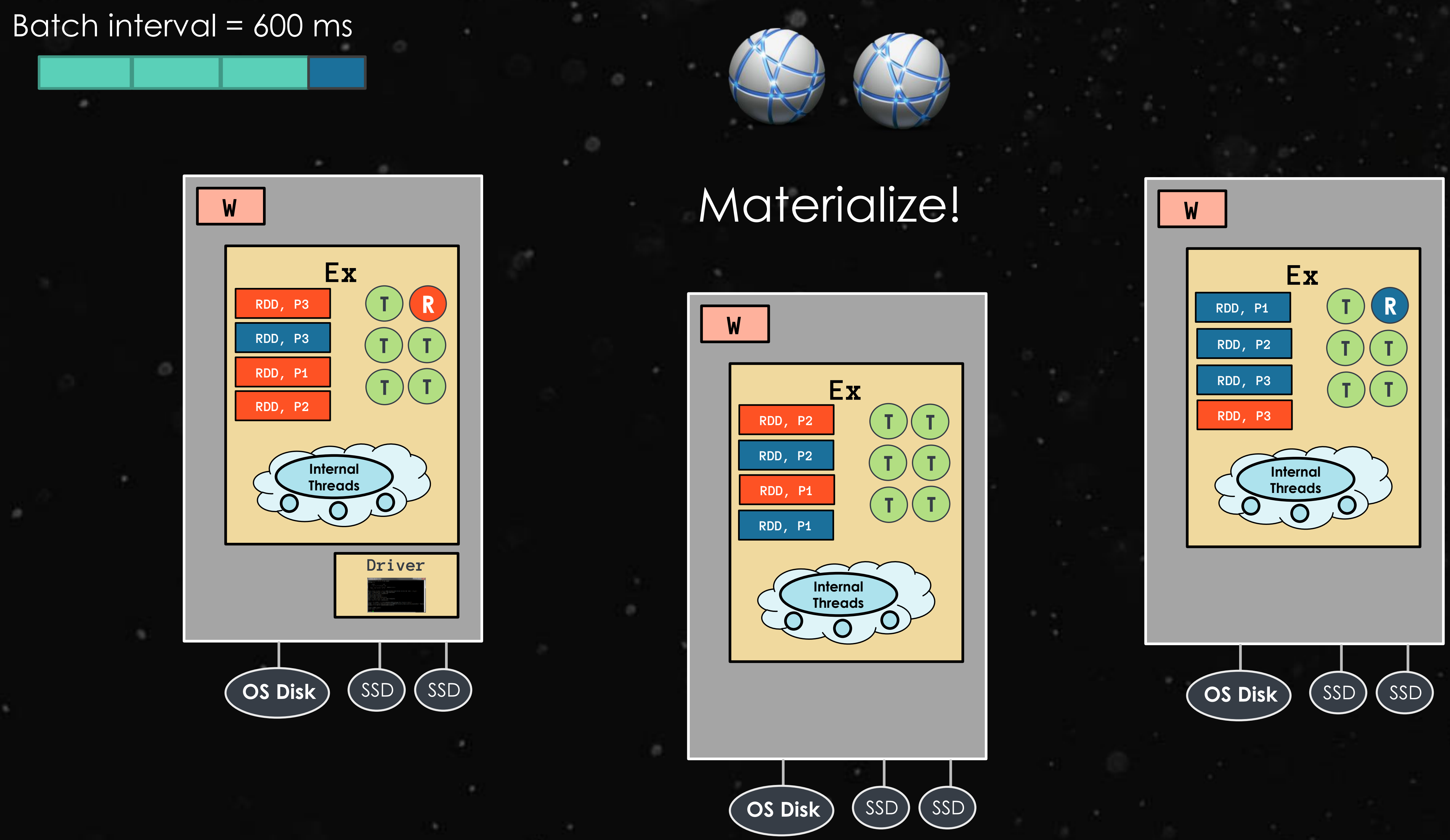




Batch interval = 600 ms
2 input DStreams
W
Ex
T
R
block, P1
Internal Threads
Driver
OS Disk
SSD
SSD
W
Ex
block, P1
block, P1
Internal Threads
OS Disk
SSD
SSD
W
Ex
R
block, P1
Internal Threads
OS Disk
SSD
SSD

Batch interval = 600 ms

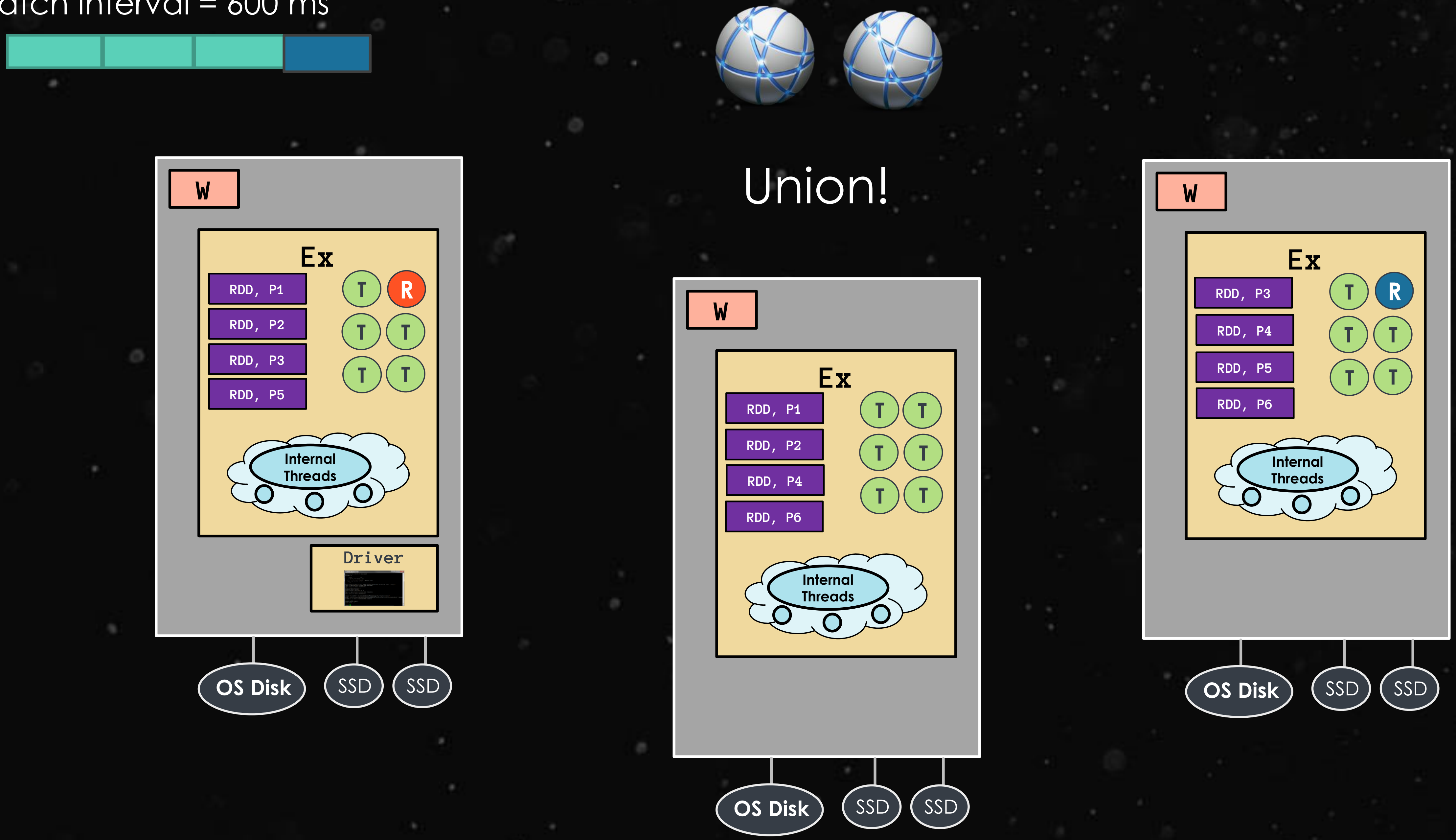




Batch interval = 600 ms
Materialize!
W
Ex
RDD, P3
RDD, P3
RDD, P1
RDD, P2
T
R
Internal Threads
Driver
OS Disk
SSD
SSD
W
Ex
RDD, P2
RDD, P2
RDD, P1
RDD, P1
T
Internal Threads
OS Disk
SSD
SSD
W
Ex
RDD, P1
RDD, P2
RDD, P3
RDD, P3
T
R
Internal Threads
OS Disk
SSD
SSD

Batch interval = 600 ms

Union!



BASIC

- File systems
- Socket Connections

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( )

OUTPUT OPERATIONS ON DSTREAMS

`print()`

`foreachRDD(` $f(x)$ `)`

`saveAsTextFile(prefix, [suffix])`

`saveAsObjectFiles(prefix, [suffix])`

`saveAsHadoopFiles(prefix, [suffix])`

databricks