
# Spark Essentials

Aaron Maxwell

Spark Summit West 2016

# Instructor: Aaron Maxwell

LinkedIn: **linkedin.com/in/aaronmaxwell**
Email: **amax@redsymbol.net**

- Background in Physics
- Software engineer
  - Python programming (book & newsletter)
  - Devops
  - Data science & Machine Learning
- Trainer in all the above
  - O'Reilly Media, NewCircle, and Databricks

# Files and Resources

Documents

- Class notes: `http://tinyurl.com/Spark-Essentials-TE1`
  - `Bookmark this!`
- Databricks URL: http://ssw2016.cloud.databricks.com
- Your username and password are on your slip of paper.

- NOTE: Use Chrome or Firefox
  - Internet Explorer is *not* supported
  - Safari may or may not work.

# Course Objectives

- Describe the motivation and fundamental mechanics of Spark
- Use the core Spark APIs to operate on real data
- Experiment with use cases for Spark
- Build data pipelines using Spark SQL and DataFrames
- Analyze Spark jobs using the administration UIs and logs
- Create Streaming and Machine Learning jobs using the common Spark API

# The Game Plan

- Focus on Spark 1.6
  - Essentially EVERYTHING you learn applies immediately to 2.0
- Regular breaks throughout the day
  - Coffee, Stretching, Networking, etc.
- Lunch from Noon to 1pm
  - It’s on us! Catered right here

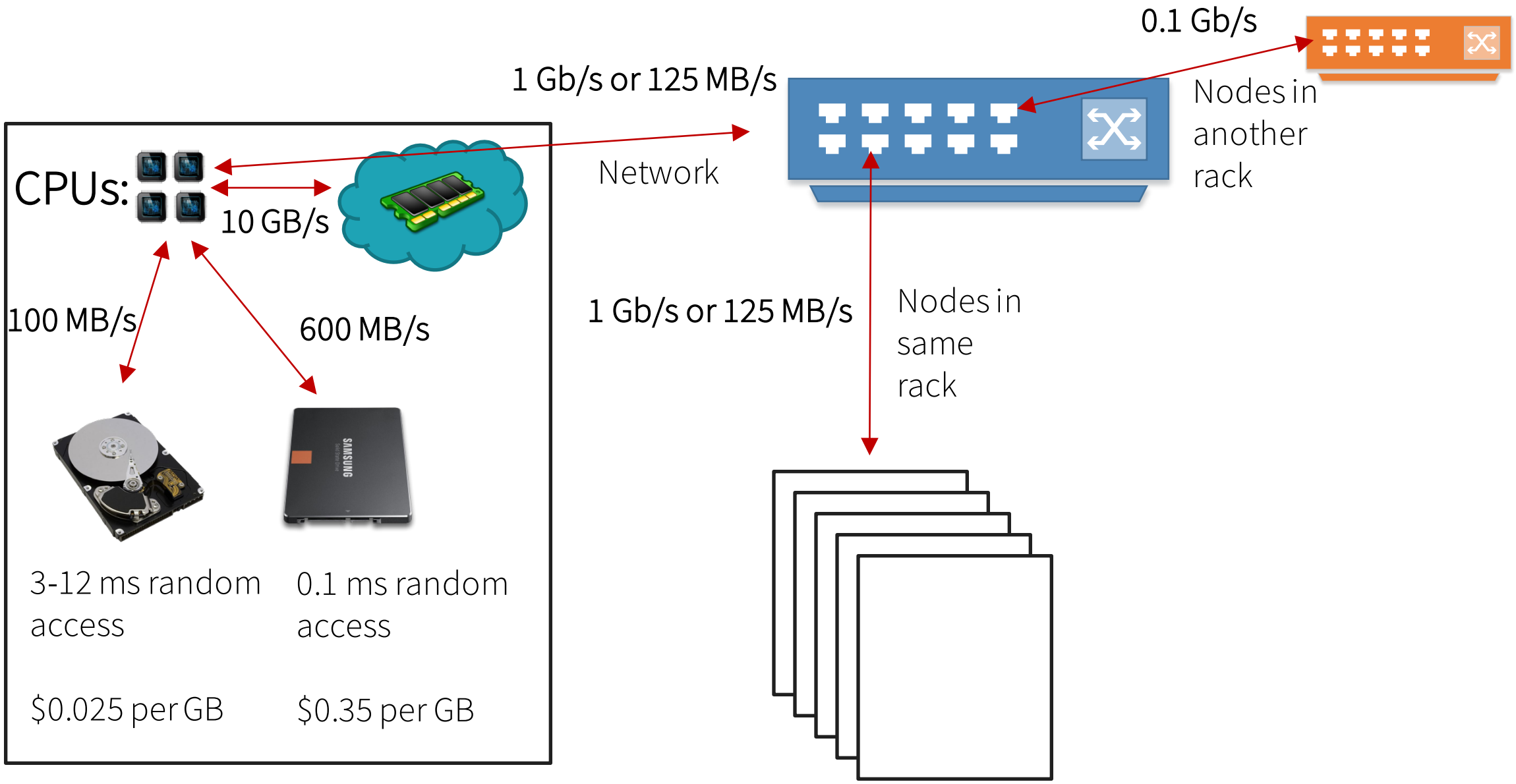
# Hands on

Take a minute to log in and find your home directory.
Then, let me show you how to create a notebook and how we'll be using the labs.
After that, I'll give you a few minutes to create your own notebook and play around.

Scala or Python? Your choice. (**Pro Tip:** Use the Language Guides.)

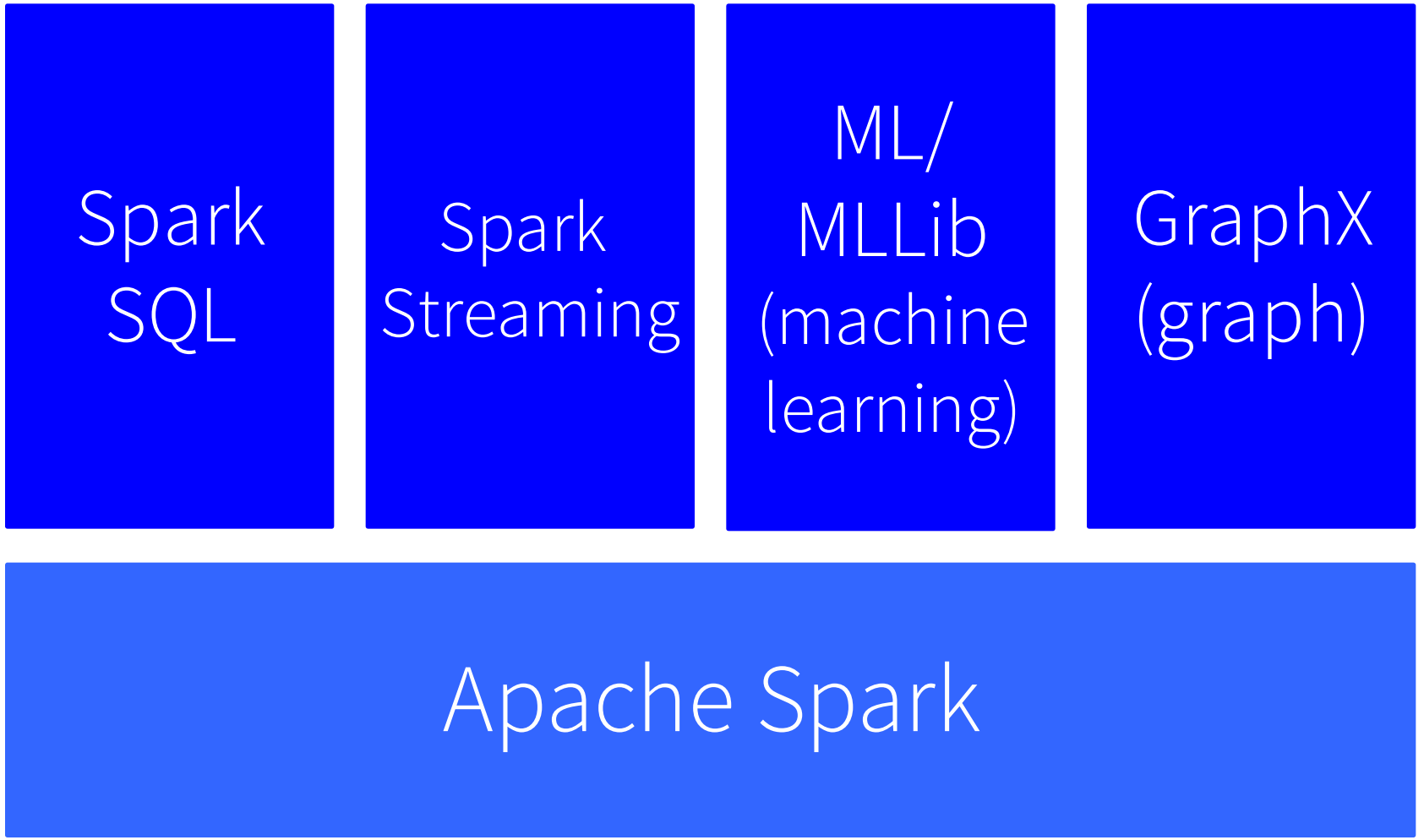

# Overview


- Started as a research project at UC Berkeley in 2009
- Open Source License (Apache 2.0)
- Latest Stable Release: v2.0 (June 2016)
- 600,000 lines of code (75% Scala)
- Built by 800+ developers from 200+ companies


CPUs:
10 GB/s
100 MB/s
600 MB/s
3-12 ms random access
0.1 ms random access
$0.025 per GB
$0.35 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

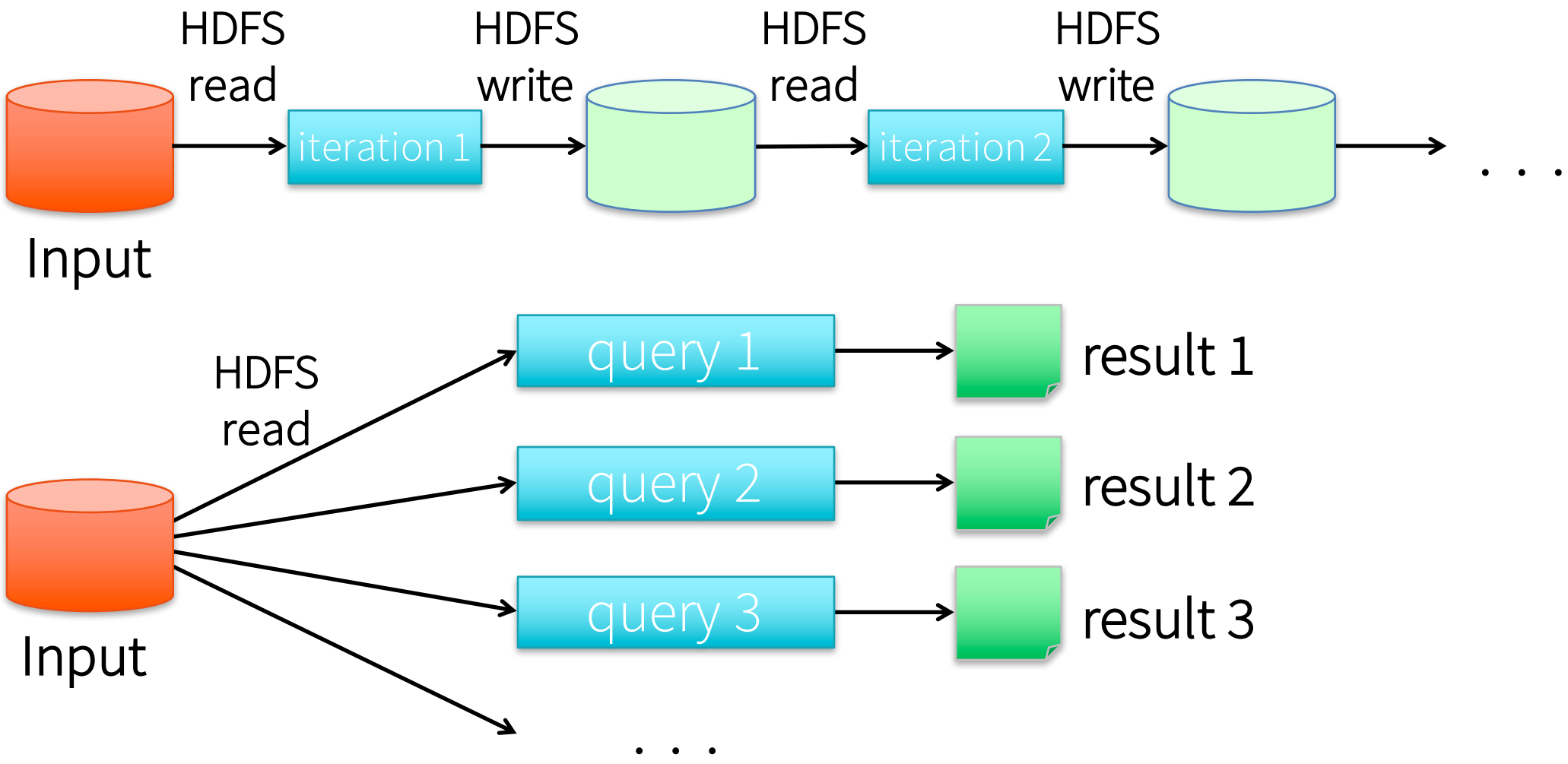
# Opportunity


- Keep more data *in-memory*
- New distributed execution environment
- Bindings for:
  - Python, Java, Scala, R

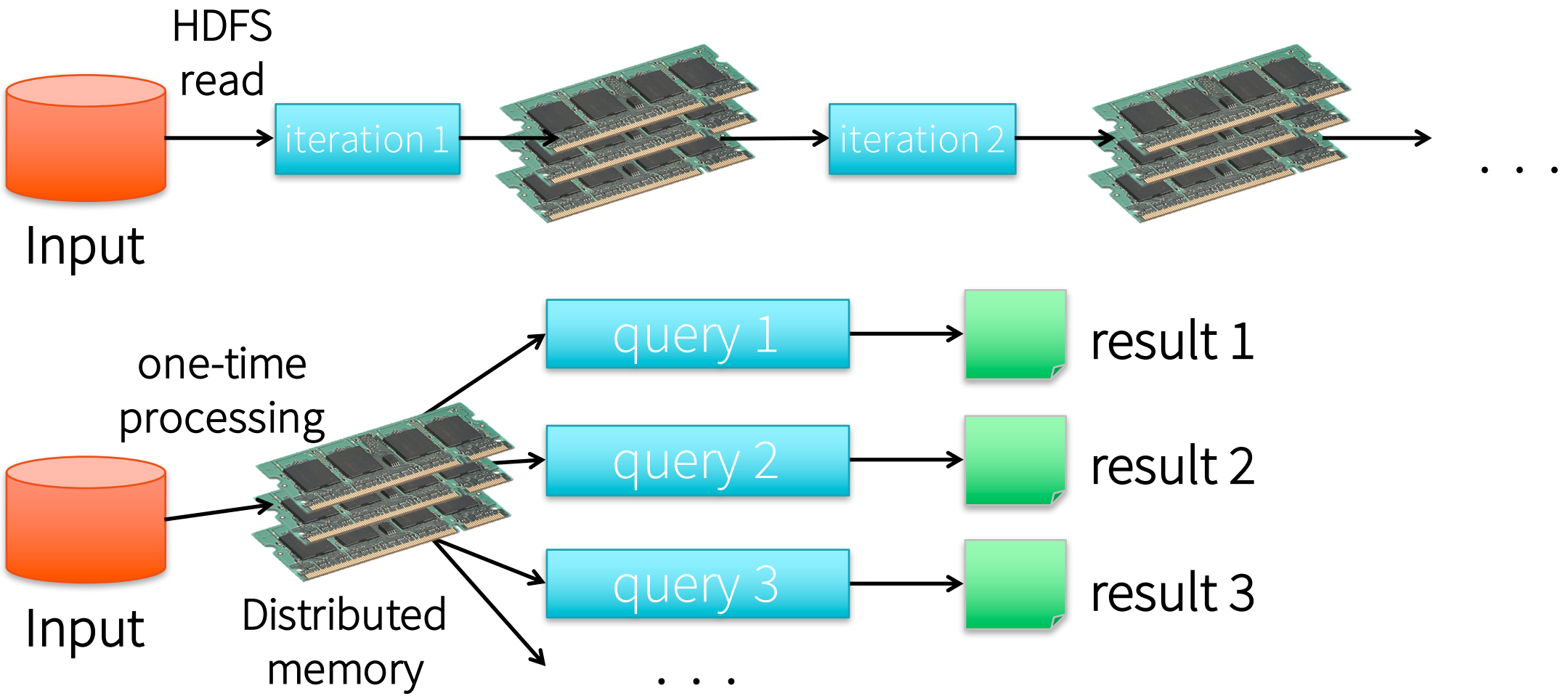



http://people.csail.mit.edu/matei/papers/2010/hotcloud_spark.pdf

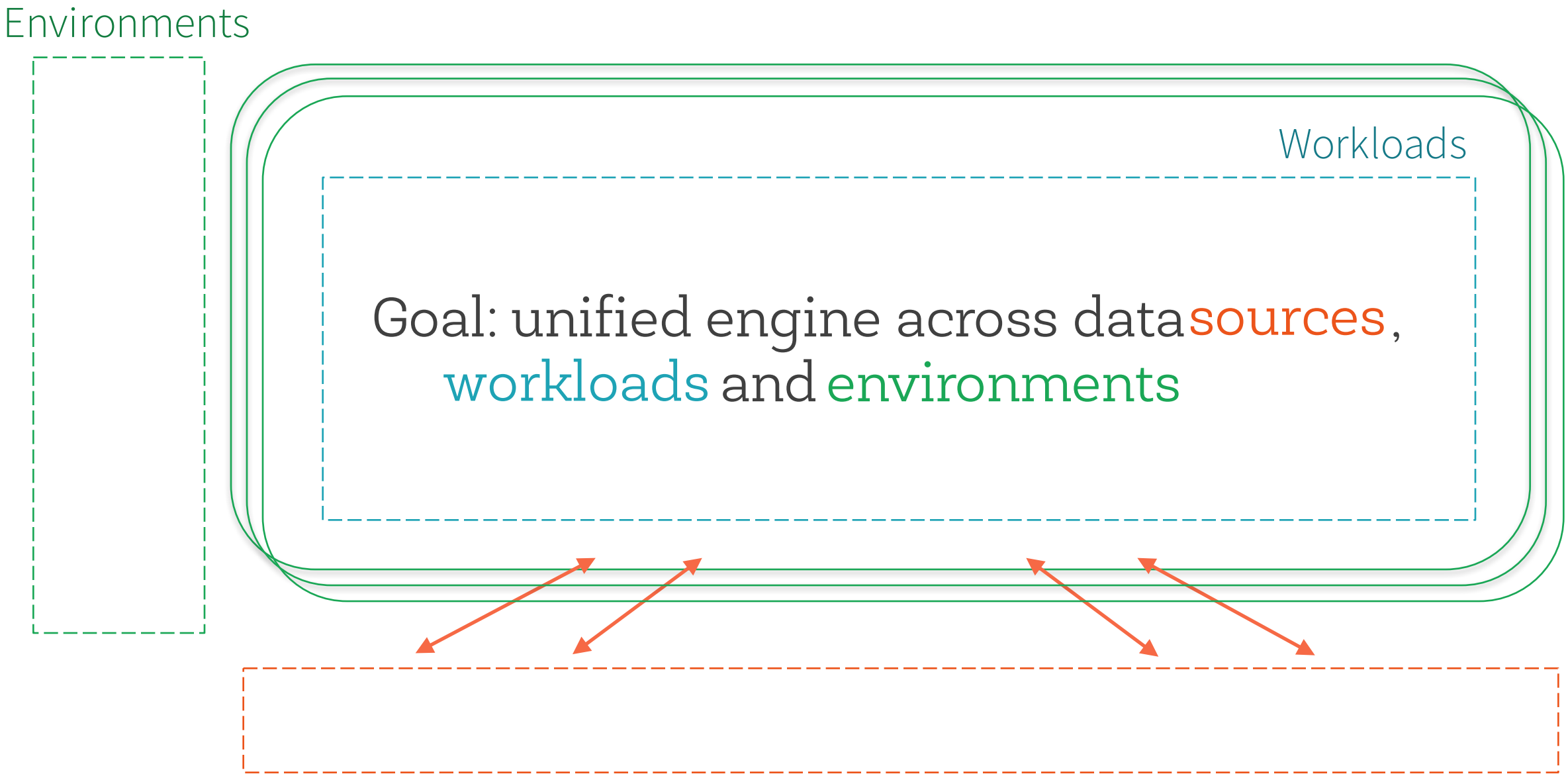
# MapReduce: Use Disk



# Spark: In-Memory Data Sharing

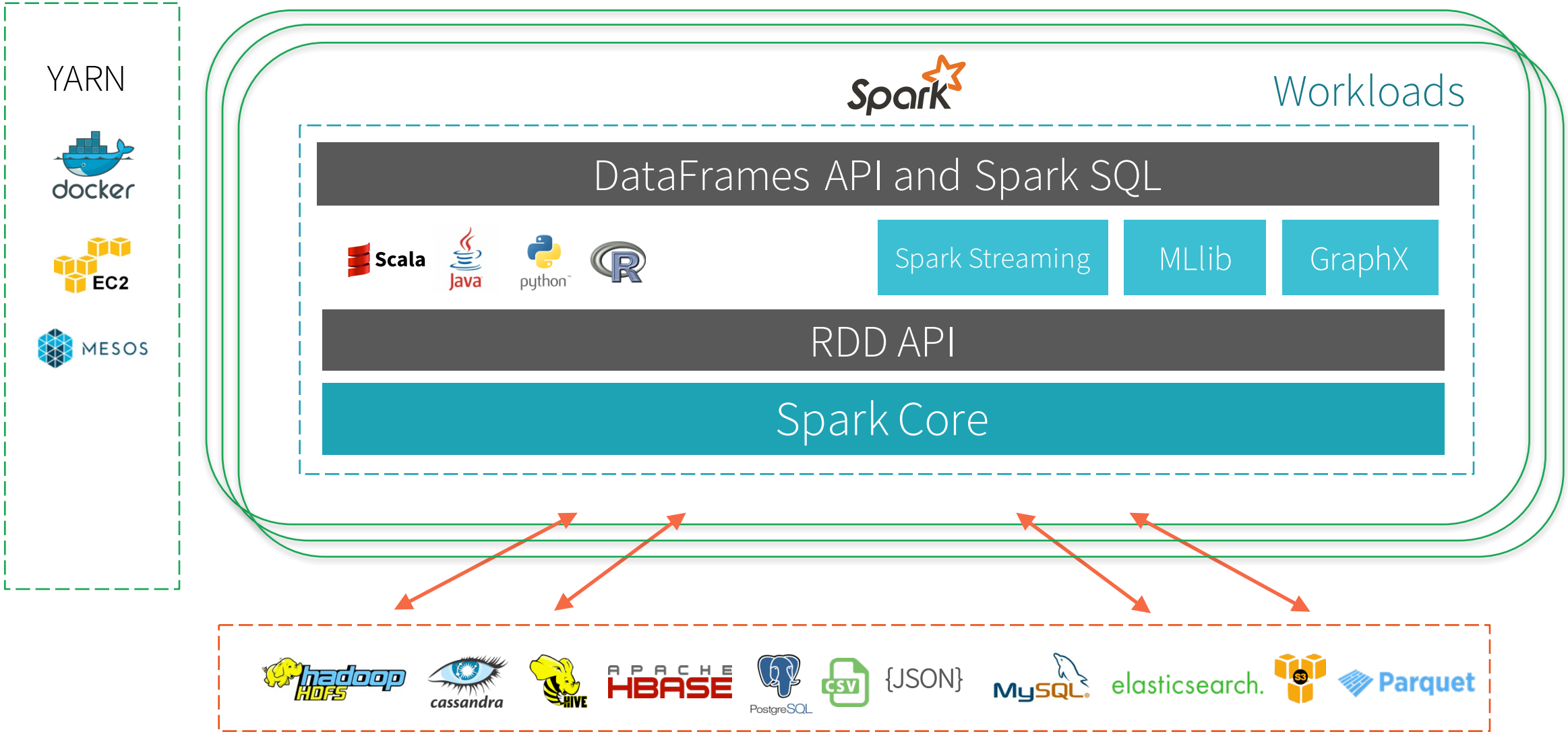



10-100x faster than network and disk


Environments
Workloads
Goal: unified engine across data sources, workloads and environments
Data Sources

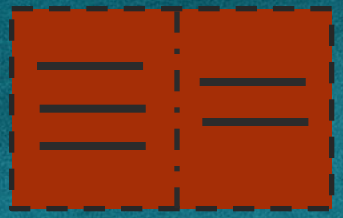
databricks

# Environments




databricks

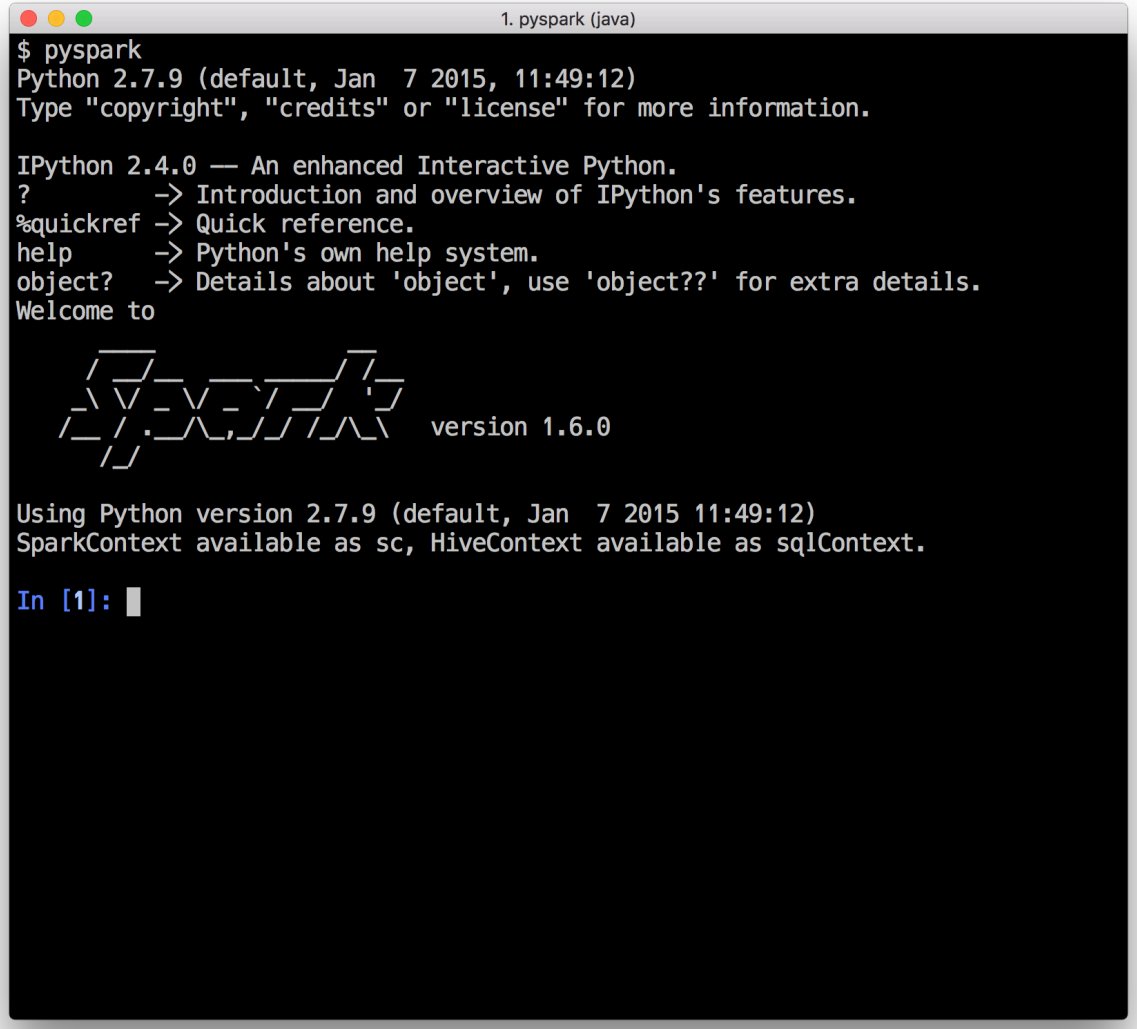
# End of Spark Overview



# RDD Fundamentals

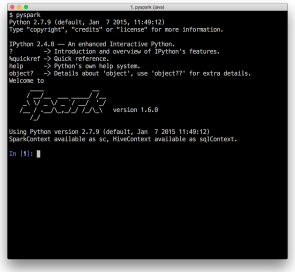
databricks™

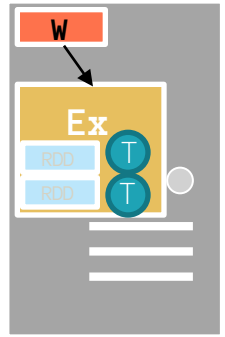
# Interactive Shell



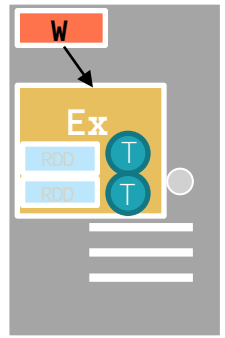

(Scala, Python and R only)

Driver Program




Worker Machine

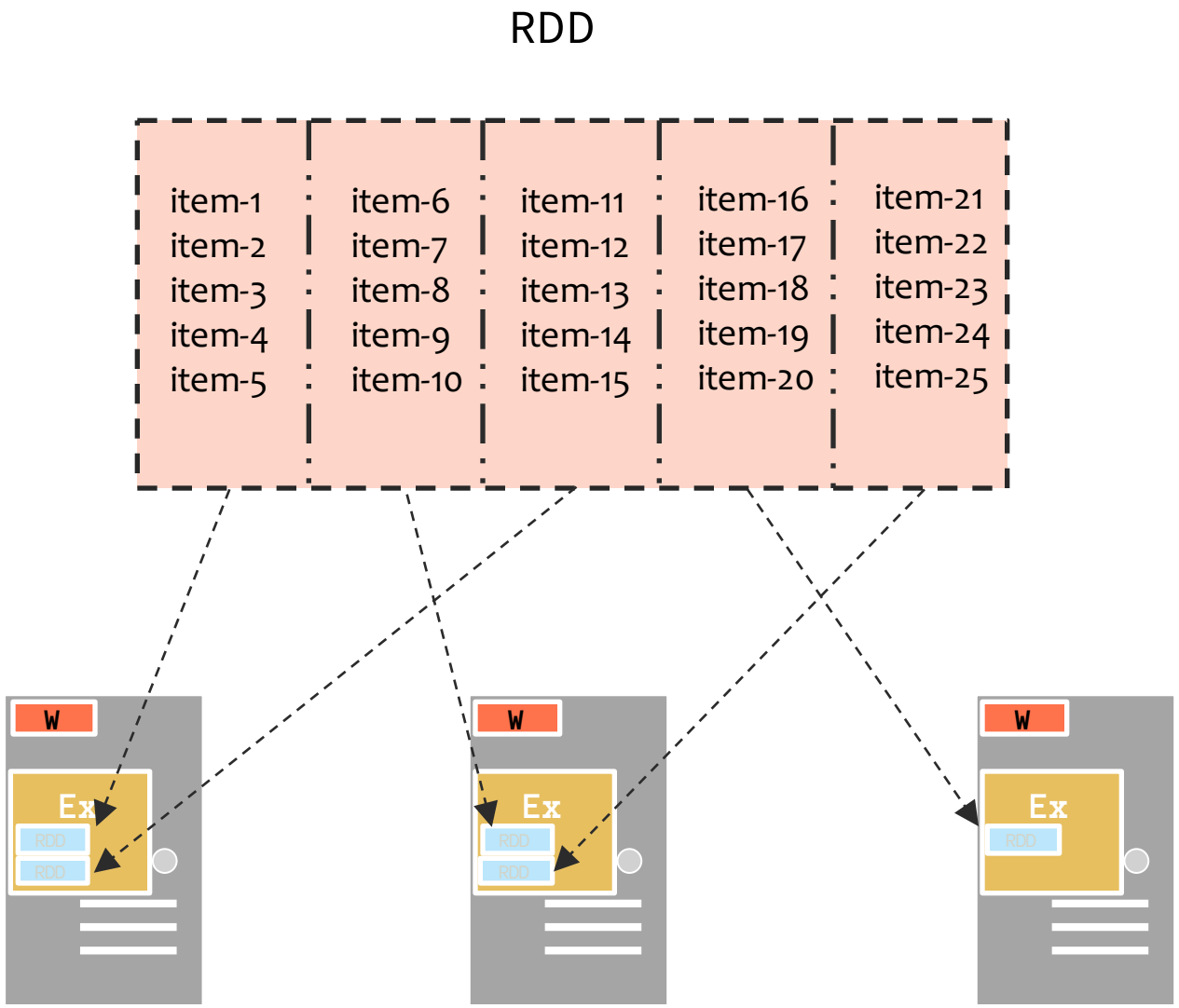


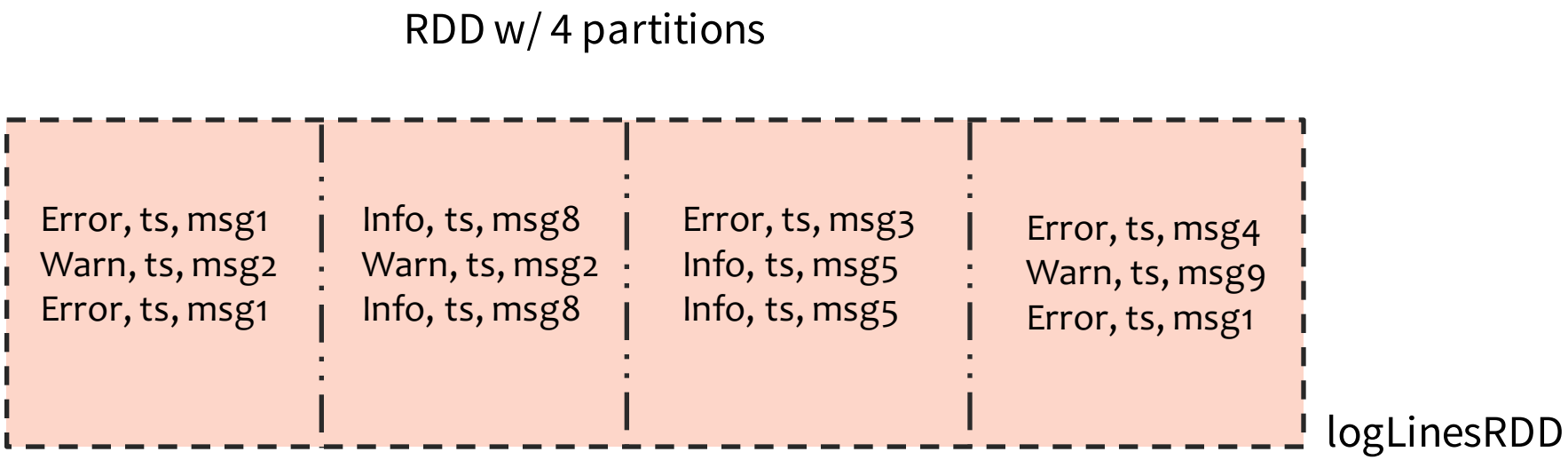

Worker Machine

# Resilient Distributed Datasets (RDDs)

- Write programs in terms of operations on distributed datasets
- Partitioned collections of objects spread across a cluster, stored in memory or on disk
- RDDs built and manipulated through a diverse set of parallel transformations (map, filter, join) and actions (count, collect, save)
- RDDs automatically rebuilt on machine failure

*more partitions = more parallelism*






A base RDD can be created 2 ways:

- Parallelize a collection
- Read data from an external source (S3, C*, HDFS, etc)

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

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

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

## Parallelize

- 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

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

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

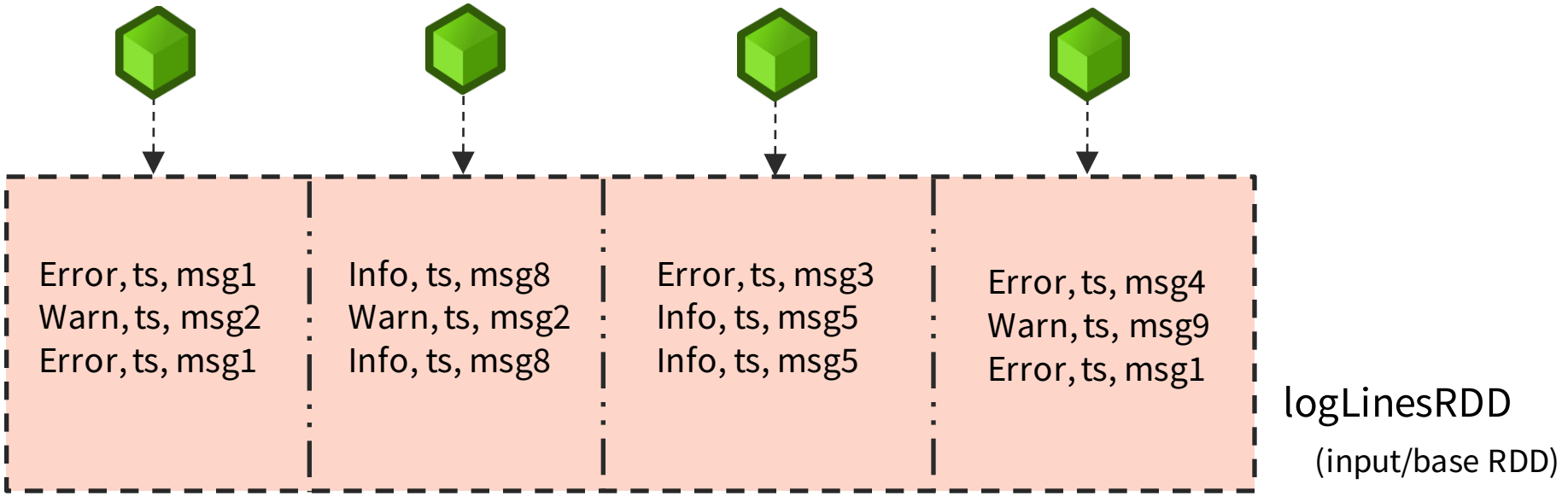
```java
// Read a local txt file in Java
JavaRDD<String> lines = sc.textFile("/path/to/README.md");
```

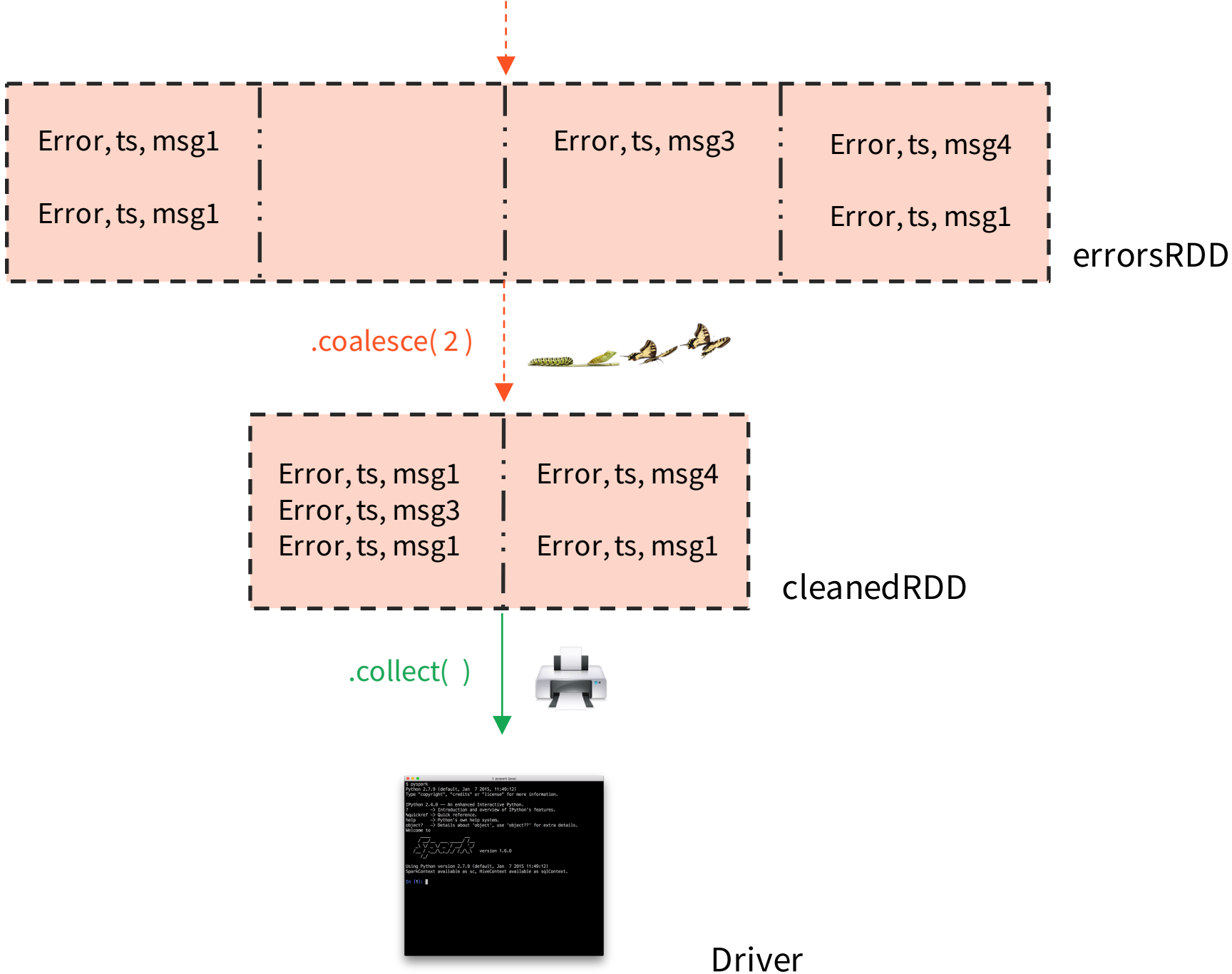
There are other methods to read data from HDFS, C*, S3, HBase, etc.

# Operations on Distributed Data

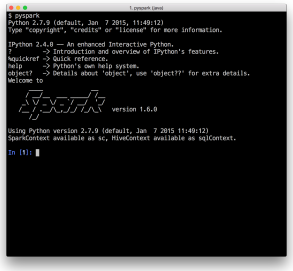
- Two types of operations: *transformations* and *actions*
- Transformations are lazy (*not computed immediately*)
- Transformations are executed when an action is run
- Persist (cache) distributed data in memory or disk


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( λ )
Error, ts, msg1
Error, ts, msg1
Error, ts, msg3
Error, ts, msg4
Error, ts, msg1
errorsRDD

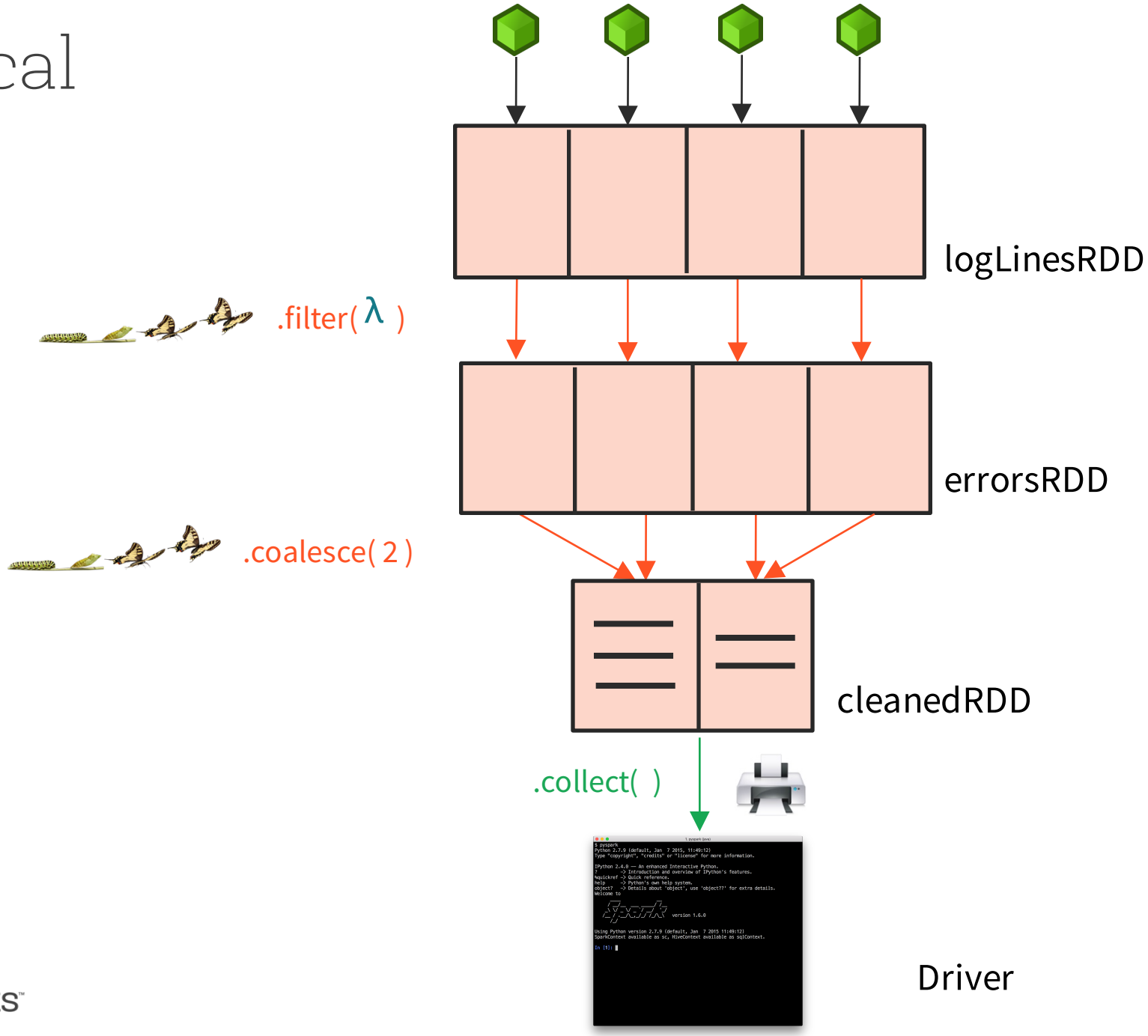

Error, ts, msg1
Error, ts, msg1
Error, ts, msg3
Error, ts, msg4
Error, ts, msg1
errorsRDD
.coalesce( 2 )
Error, ts, msg1
Error, ts, msg3
Error, ts, msg1
Error, ts, msg4
Error, ts, msg1
cleanedRDD
.collect( )
Driver


Execute DAG!
.collect( )

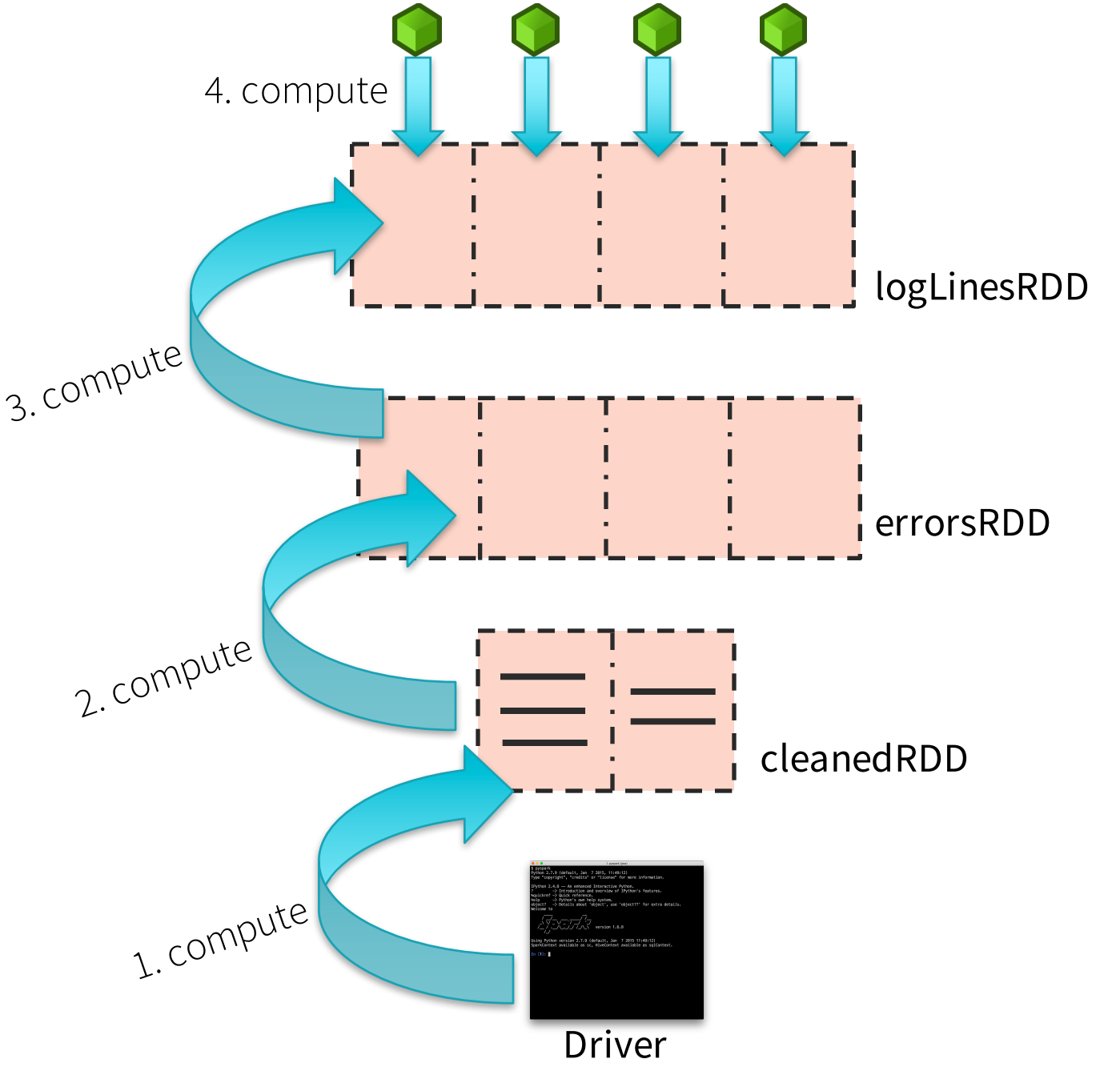

$ pyspark
Python 2.7.9 (default, Jan 7 2015, 11:49:12)
Type "copyright", "credits" or "license" for more information.
IPython 2.4.0 -- An enhanced Interactive Python.
? -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help -> Python's own help system.
object? -> Details about 'object', use 'object??' for extra details.
Welcome to
version 1.6.0
Using Python version 2.7.9 (default, Jan 7 2015 11:49:12)
SparkContext available as sc, HiveContext available as sqlContext.
In [1]:

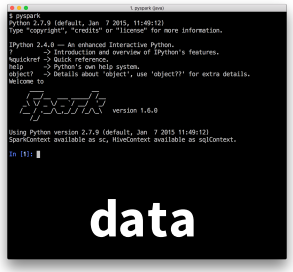
Driver

# Logical

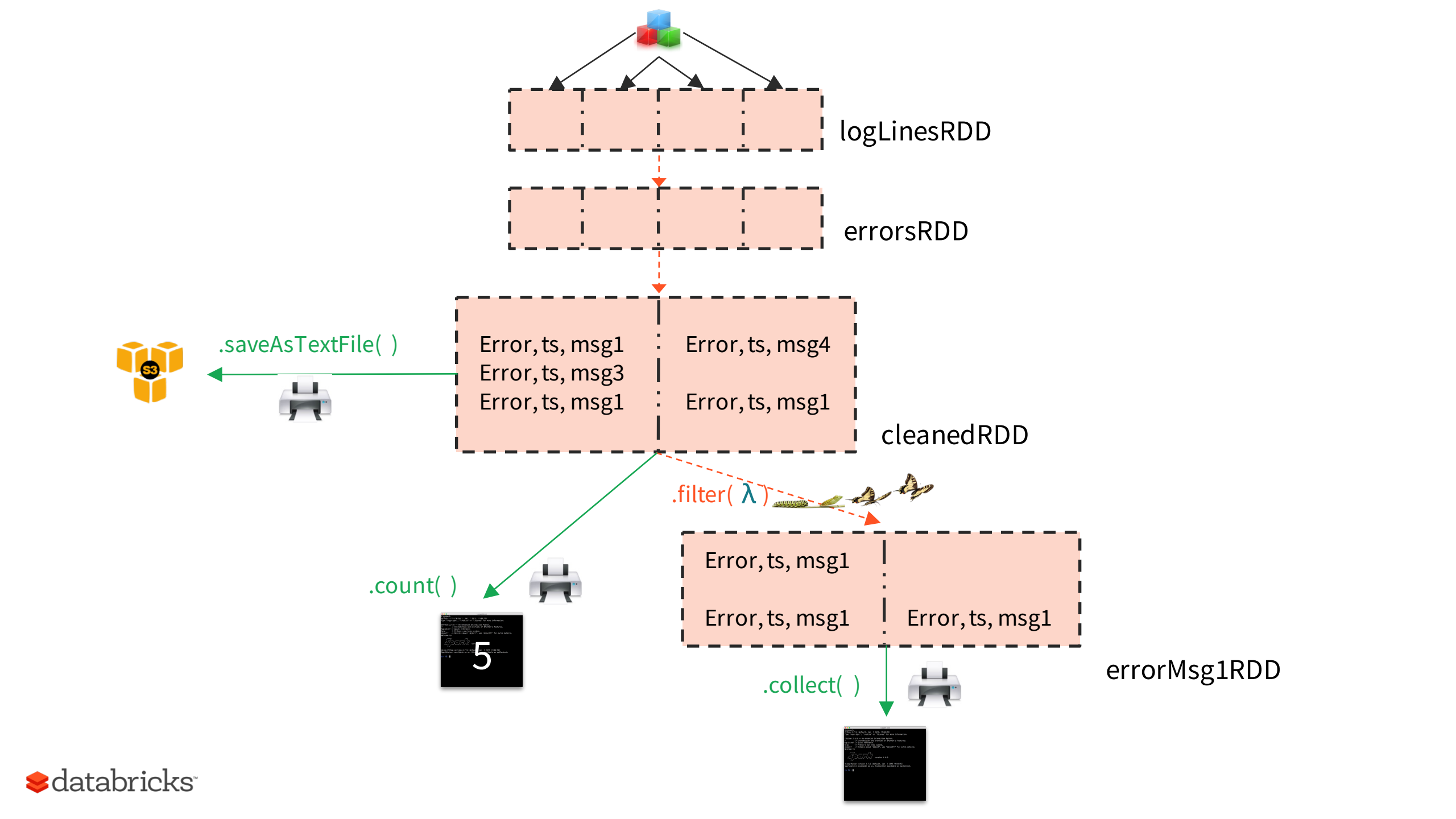

logLinesRDD
.filter( λ )
errorsRDD
.coalesce( 2 )
cleanedRDD
.collect( )
Driver

# Physical

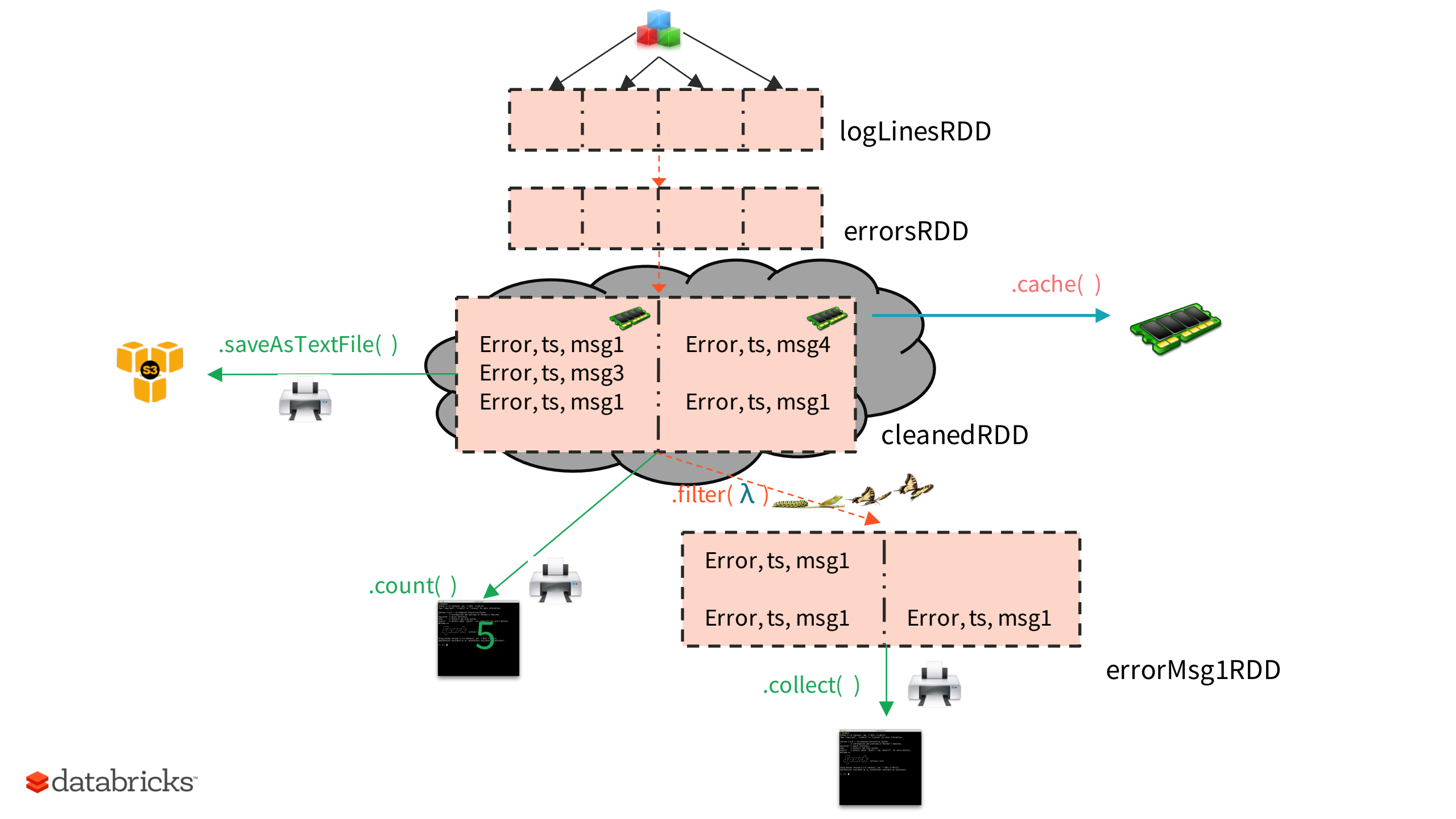

4. compute
logLinesRDD
3. compute
errorsRDD
2. compute
cleanedRDD
1. compute
Driver


$ pyspark
Python 2.7.9 (default, Jan 7 2015, 11:49:12)
Type "copyright", "credits" or "license" for more information.
IPython 2.4.0 -- An enhanced Interactive Python.
? -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help -> Python's own help system.
object? -> Details about 'object', use 'object??' for extra details.
Welcome to
version 1.6.0
Using Python version 2.7.9 (default, Jan 7 2015 11:49:12)
SparkContext available as sc, HiveContext available as sqlContext.
In [1]:
data

Driver


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

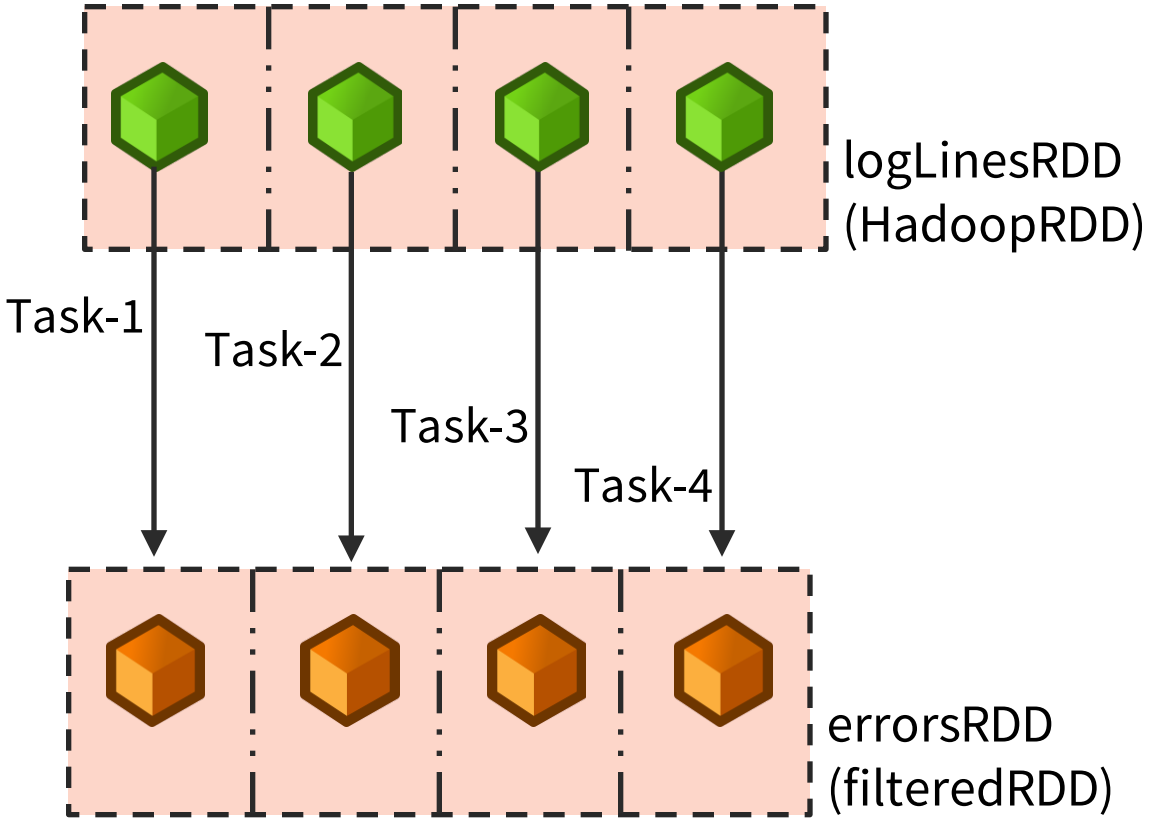
databricks


logLinesRDD
errorsRDD
.cache( )
.saveAsTextFile( )
Error, ts, msg1
Error, ts, msg3
Error, ts, msg1
Error, ts, msg4
Error, ts, msg1
cleanedRDD
.filter( λ )
.count( )
5
Error, ts, msg1
Error, ts, msg1
Error, ts, msg1
errorMsg1RDD
.collect( )

databricks

# Partition >>> Task >>> Partition

.filter( λ )



# 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()      cartesian()

flatMap()                      distinct()          pipe()

filter()                       groupByKey()        coalesce()

mapPartitions()                reduceByKey()       repartition()

mapPartitionsWithIndex()       sortByKey()         partitionBy()

sample()                       join()              ...

union()                        cogroup()           ...
```

# Actions

```
reduce()
collect()
count()
first()
take()
takeSample()
saveToCassandra()
```

```
takeOrdered()
saveAsTextFile()
saveAsSequenceFile()
saveAsObjectFile()
countByKey()
foreach()
...
```

# Some Types of RDDs

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

- DoubleRDD
- JdbcRDD
- JsonRDD
- VertexRDD
- EdgeRDD

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

# Hands-on with RDDs

In the Databricks shard, look in

Essentials > RDD-Fundamentals

In there, you'll find a Washington-Crime folder, with Python and Scala folders beneath it.

- Clone either the Python or Scala folder to your home folder
- Open the lab notebook
- Attach the lab notebook to your cluster
- Follow the instructions in the notebook
- Stop when you get to the section entitled "Demonstration".

If you have trouble with any of the exercises, consulting the appropriate Solutions notebook.

# End of RDD Fundamentals

# Intro to DataFrames and Spark SQL


# Spark SQL

- Part of the core distribution since 1.0 (April 2014)
- Runs SQL / HiveQL queries, optionally alongside or replacing existing Hive deployments


```
SELECT COUNT(*)
FROM hiveTable
WHERE hive_udf(data)
```

**Improved multi-version support in 1.4**

# DataFrames API

- Enable wider audiences beyond “Big Data” engineers to leverage the power of distributed processing
- Inspired by data frames in R and Python (Pandas)
- Designed from the ground-up to support modern big data and data science applications
- Extension to the existing RDD API

See

- https://spark.apache.org/docs/latest/sql-programming-guide.html
- databricks.com/blog/2015/02/17/introducing-dataframes-in-spark-for-large-scale-data-science.html

# DataFrames

- The preferred abstraction in Spark (introduced in 1.3)
  - Strongly typed collection of distributed elements
    - Built on Resilient Distributed Datasets
  - Immutable once constructed
  - Track lineage information to efficiently recompute lost data
  - Enable operations on collection of elements in parallel

- You construct DataFrames
  - by *parallelizing* existing collections (e.g., Pandas DataFrames)
  - by *transforming* an existing DataFrames
  - from *files* in HDFS or any other storage system (e.g., Parquet)

# DataFrames Features

- Ability to scale from kilobytes of data on a single laptop to petabytes on a large cluster
- Support for a wide array of data formats and storage systems
- State-of-the-art optimization and code generation through the Spark SQL Catalyst optimizer
- Seamless integration with all big data tooling and infrastructure via Spark
- APIs for Python, Java, Scala, and R

# DataFrames versus RDDs

- For new users familiar with data frames in other programming languages, this API should make them feel at home
- For existing Spark users, the API will make Spark easier to program than using RDDs
- For both sets of users, DataFrames will improve performance through intelligent optimizations and code-generation

# Write Less Code: Input & Output

Unified interface to reading/writing data in a variety of formats.

```
val df = sqlContext.
  read.
  format("json").
  option("samplingRatio", "0.1").
  load("/Users/spark/data/stuff.json")

df.write.
   format("parquet").
   mode("append").
   partitionBy("year").
   saveAsTable("faster-stuff")
```

# Write Less Code: Input & Output

Unified interface to reading/writing data in a variety of formats.

```
val df = sqlContext.
  read.
  format("json").
  option("samplingRatio", "0.1").
  load("/Users/spark/data/stuff.json")

df.write.
   format("parquet").
   mode("append").
   partitionBy("year").
   saveAsTable("faster-stuff")
```

`read` and `write` functions create new builders for doing I/O

# Write Less Code: Input & Output

Unified interface to reading/writing data in a variety of formats.

```
val df = sqlContext.
  read.
  format("json").
  option("samplingRatio", "0.1").
  load("/Users/spark/data/stuff.json")

df.write.
   format("parquet").
   mode("append").
   partitionBy("year").
   saveAsTable("faster-stuff")
```

Builder methods specify:

- format
- partitioning
- handling of existing data

# Write Less Code: Input & Output

Unified interface to reading/writing data in a variety of formats.

```
val df = sqlContext.
  read.
  format("json").
  option("samplingRatio", "0.1").
  load("/Users/spark/data/stuff.json")

df.write.
   format("parquet").
   mode("append").
   partitionBy("year").
   saveAsTable("faster-stuff")
```

load(…), save(…), or saveAsTable(…) finish the I/O specification

# Data Sources supported by DataFrames

built-in









external








and more …

# Write Less Code: High-Level Operations

Solve common problems concisely with DataFrame functions:

- selecting columns and filtering
- joining different data sources
- aggregation (count, sum, average, etc.)
- plotting results (e.g., with Pandas)

# Write Less Code: Compute an Average


```
private IntWritable one = new IntWritable(1);
private IntWritable output =new IntWritable();
protected void map(LongWritable key,
                   Text value,
                   Context context) {
    String[] fields = value.split("\t");
    output.set(Integer.parseInt(fields[1]));
    context.write(one, output);
}

------------------------------------------------------------------

IntWritable one = new IntWritable(1)
DoubleWritable average = new DoubleWritable();

protected void reduce(IntWritable key,
                      Iterable<IntWritable> values,
                      Context context) {
    int sum = 0;
    int count = 0;
    for (IntWritable value: values) {
        sum += value.get();
        count++;
    }
    average.set(sum / (double) count);
    context.write(key, average);
}
```


```
rdd = sc.textFile(...).map(_.split(" "))
rdd.map { x => (x(0), (x(1).toFloat, 1)) }.
    reduceByKey { case ((num1, count1), (num2, count2)) =>
      (num1 + num2, count1 + count2)
    }.
    map { case (key, (num, count)) => (key, num / count) }.
    collect()
```

```
rdd = sc.textFile(...).map(lambda s: s.split())
rdd.map(lambda x: (x[0], (float(x[1]), 1))).\
    reduceByKey(lambda t1, t2: (t1[0] + t2[0], t1[1] + t2[1])).\
    map(lambda t: (t[0], t[1][0] / t[1][1])).\
    collect()
```

# Write Less Code: Compute an Average

## Using RDDs

```
rdd = sc.textFile(...).map(_.split(" "))
rdd.map { x => (x(0), (x(1).toFloat, 1)) }.
    reduceByKey { case ((num1, count1), (num2, count2)) =>
      (num1 + num2, count1 + count2)
    }.
    map { case (key, (num, count)) => (key, num / count) }.
    collect()
```

## Using DataFrames

```
import org.apache.spark.sql.functions._

val df = rdd.map(a => (a(0), a(1))).toDF("key", "value")
df.groupBy("key")
  .agg(avg("value"))
  .collect()
```

## Full API Docs

- Scala
- Java
- Python
- R

# Hands on

Let's take a quick look at that in action. Under

**`Essentials > Sql-and-Dataframes`**

you'll find a **`less-code`** folder. In that folder, you'll find a Python notebook and a Scala notebook. Clone one of them to your home directory, attach it to your cluster, and follow along.

# Construct a DataFrame (Scala)

```scala
# Construct a DataFrame from a "users" table in Hive.
val df = sqlContext.table("users")

# Construct a DataFrame from a log file in S3.
val df = sqlContext.load("s3n://aBucket/path/to/data.json", "json")
```

# Use DataFrames (Scala)

```scala
// Create a new DataFrame that contains only "young" users
val young = users.filter($"age" < 21)

// Increment everybody's age by 1
young.select($"name", $"age" + 1)

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

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

# Use DataFrames (Python)

```python
# Create a new DataFrame that contains only "young" users
young = users.filter(users["age"] < 21)

# Alternatively, using a 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, logs
young.join(log, logs["userId"] == users["userId"], "left_outer")
```

# DataFrames and Spark SQL

```
young.registerTempTable("young")

sqlContext.sql("SELECT  count(*)  FROM  young")
```

# DataFrames and Spark SQL

- DataFrames are fundamentally tied to Spark SQL
- The DataFrames API provides a *programmatic* interface—really, a *domain-specific language* (DSL)—for interacting with your data.
- Spark SQL provides a *SQL-like* interface.
- Anything you can do in Spark SQL, you can do in DataFrames
- … and vice versa

# What, exactly, is Spark SQL?

- Spark SQL allows you to manipulate distributed data with SQL queries. Currently, two SQL dialects are supported.
  - If you're using a Spark `SQLContext`, the only supported dialect is "sql," a rich subset of SQL 92.
  - If you're using a `HiveContext`, the default dialect is "hiveql", corresponding to Hive's SQL dialect. "sql" is also available, but "hiveql" is a richer dialect.

# Spark SQL

- You issue SQL queries through a `SQLContext` or `HiveContext`, using the `sql()` method.
- The `sql()` method returns a `DataFrame`.
- You can mix DataFrame methods and SQL queries in the same code.
- To use SQL, you *must* either:
  - query a persisted Hive table, or
  - make a *table alias* for a DataFrame, using `registerTempTable()`

# Transformations, Actions, Laziness

Like RDDs, DataFrames are *lazy*. *Transformations* contribute to the query plan, but they don't execute anything.

*Actions* cause the execution of the query.

**Transformation examples**

- filter
- select
- drop
- intersect
- join

**Action examples**

- count
- collect
- show
- head
- take

# Transformations, Actions, Laziness

*Actions* cause the execution of the query.

What, exactly, does “execution of the query” mean? It means:

- Spark initiates a distributed read of the data source
- The data flows through the transformations (the RDDs resulting from the Catalyst query plan)
- The result of the action is pulled back into the driver JVM.

# Creating a DataFrame

- You create a DataFrame with a `SQLContext` object (or one of its descendants)
- In the Spark Scala shell (*spark-shell*) or *pyspark*, you have a `SQLContext` available automatically, as `sqlContext`.
- In an application, you can easily create one yourself, from a `SparkContext`.
- The DataFrame *data source API* is consistent, across data formats.
  - "Opening" a data source works pretty much the same way, no matter what.

# Creating a DataFrame in Scala

Program: including setup; the DataFrame reads are 1 line each

```scala
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.sql.SQLContext

val conf = new SparkConf().setAppName(appName).
                           setMaster(master)
// Returns existing SparkContext, if there is one;
// otherwise, creates a new one from the config.
val sc = SparkContext.getOrCreate(conf)
// Ditto.
val sqlContext = SQLContext.getOrCreate(sc)

val df = sqlContext.read.parquet("/path/to/data.parquet")
val df2 = sqlContext.read.json("/path/to/data.json")
```

# Creating a DataFrame in Python

Program: including setup; the DataFrame reads are 1 line each

```
# The import isn't necessary in the SparkShell or Databricks
from pyspark import SparkContext, SparkConf

# The following three lines are not necessary
# in the pyspark shell
conf = SparkConf().setAppName(appName).setMaster(master)
sc = SparkContext(conf=conf)
sqlContext = SQLContext(sc)

df = sqlContext.read.parquet("/path/to/data.parquet")
df2 = sqlContext.read.json("/path/to/data.json")
```

# DataFrames Have Schemas

- In the previous example, we created DataFrames from Parquet and JSON data.
- A Parquet table has a schema (column names and types) that Spark can use. Parquet also allows Spark to be efficient about how it pares down data.
- Spark can *infer* a Schema from a JSON file.

# printSchema()

You can have Spark tell you what it thinks the data schema is, by calling the `printSchema()` method. (This is mostly useful in the shell.)

```
> df.printSchema()
root
 |-- firstName: string (nullable = true)
 |-- lastName: string (nullable = true)
 |-- gender: string (nullable = true)
 |-- age: integer (nullable = false)
```

# Schema Inference

Some data sources (e.g.,P) can expose a formal schema; others (e.g., plain text files) don't. How do we fix that?

- You can create an RDD of a particular type and let Spark infer the schema from that type. We'll see how to do that in a moment.
- You can use the API to specify the schema programmatically.

# Schema Inference Example

- Suppose you have a (text) file that looks like this:

```
Erin,Shannon,F,42
Norman,Lockwood,M,81
Miguel,Ruiz,M,64
Rosalita,Ramirez,F,14
Ally,Garcia,F,39
Claire,McBride,F,23
Abigail,Cottrell,F,75
José,Rivera,M,59
Ravi,Dasgupta,M,25
…
```

The file has no schema, but it's obvious there *is* one:

First name: *string*
Last name: *string*
Gender: *string*
Age: *integer*

Let's see how to get Spark to infer the schema.

# Schema Inference (Scala)

```scala
import sqlContext.implicits._

case class Person(firstName: String,
                  lastName:  String,
                  gender:    String,
                  age:       Int)

val rdd = sc.textFile("people.csv")
val peopleRDD = rdd.map { line =>
  val cols = line.split(",")
  Person(cols(0), cols(1), cols(2), cols(3).toInt)
}
val df = peopleRDD.toDF
// df: DataFrame = [firstName: string, lastName: string, gender: string,
age: int]
```

# Schema Inference (Python)

```python
from pyspark.sql import Row

rdd = sc.textFile("people.csv")
Person = Row('first_name', 'last_name', 'gender', 'age')

def line_to_person(line):
    cells = line.split(",")
    cells[3] = int(cells[3])
    return Person(*cells)

peopleRDD = rdd.map(line_to_person)

df = peopleRDD.toDF()
# DataFrame[first_name: string, last_name: string, \
# gender: string, age: bigint]
```

# Schema Inference

We can also force schema inference

- *without* creating our own People type,
- by using a fixed-length data structure (such as a *tuple*)
- and supplying the column names to the `toDF()` method.

# Schema Inference (Scala)

```scala
val rdd = sc.textFile("people.csv")
val peopleRDD = rdd.map { line =>
  val cols = line.split(",")
  (cols(0), cols(1), cols(2), cols(3).toInt)
}

val df = peopleRDD.toDF("firstName", "lastName",
                        "gender", "age")
```



If you don't supply the column names, the API defaults to "_1", "_2", etc.

# Schema Inference (Python)

```python
from collections import namedtuple

Person = namedtuple('Person',
    ['first_name', 'last_name', 'gender', 'age']
)
rdd = sc.textFile("people.csv")

def line_to_person(line):
    cells = line.split(",")
    return Person(cells[0], cells[1], cells[2], int(cells[3]))

peopleRDD = rdd.map(line_to_person)

df = peopleRDD.toDF()
# DataFrame[first_name: string, last_name: string, \
# gender: string, age: bigint]
```

# What can I do with a DataFrame?

- Once you have a DataFrame, there are a number of operations you can perform.
- Let's look at a few of them.
- But, first, let's talk about columns.

# Columns

- When we say “column” here, what do we mean?
  - a DataFrame *column* is an abstraction. It provides a common column-oriented view of the underlying data, *regardless* of how the data is really organized.
  - Columns are important because much of the DataFrame API consists of functions that take or return columns (even if they don’t look that way at first).

# Columns

| Input Source Format | Data Frame Variable Name | Data |
|---|---|---|
| JSON | `dataFrame1` | `[ {"first": "Amy", "last": "Bello", "age": 29 }, {"first": "Ravi", "last": "Agarwal", "age": 33 }, … ]` |
| CSV | `dataFrame2` | `first,last,age`<br>`Fred,Hoover,91`<br>`Joaquin,Hernandez,24`<br>… |
| SQL Table | `dataFrame3` | **first** / **last** / **age**<br>Joe / Smith / 42<br>Jill / Jones / 33 |

Let's see how DataFrame columns map onto some common data sources.

# Columns

Assume we have a DataFrame, `df`, that reads a data source that has "first", "last", and "age" columns.

| Python | Java | Scala | R |
|---|---|---|---|
| `df["first"]`<br>`df.first`† | `df.col("first")` | `df("first")`<br>`$"first"`‡ | `df$first` |

†In Python, it's possible to access a DataFrame's columns either by attribute (`df.age`) or by indexing (`df['age']`). While the former is convenient for interactive data exploration, you should *use the index form.* It's future proof and won't break with column names that are also attributes on the DataFrame class.

‡The $ syntax can be ambiguous, if there are multiple DataFrames in the lineage.

# show()

You can look at the first *n* elements in a DataFrame with the `show()` method. If not specified, *n* defaults to 20.

This method is an *action*. It:

- reads (or re-reads) the input source
- executes the RDD DAG across the cluster
- pulls the *n* elements back to the driver JVM
- displays those elements in a tabular form

# show()

```
> df.show()
+---------+--------+------+---+
|firstName|lastName|gender|age|
+---------+--------+------+---+
|     Erin| Shannon|     F| 42|
|   Claire| McBride|     F| 23|
|   Norman|Lockwood|     M| 81|
|   Miguel|    Ruiz|     M| 64|
| Rosalita| Ramirez|     F| 14|
|     Ally|  Garcia|     F| 39|
|  Abigail|Cottrell|     F| 75|
|     José|  Rivera|     M| 59|
+---------+--------+------+---+
```

# select()

`select()` is like a SQL SELECT, allowing you to limit the results to specific columns.

```
scala> df.select($"firstName", $"age").show(5)
+---------+---+
|firstName|age|
+---------+---+
|     Erin| 42|
|   Claire| 23|
|   Norman| 81|
|   Miguel| 64|
| Rosalita| 14|
+---------+---+
```

# select()

```
In[1]: df.select(df['first_name'], df['age'], df['age'] > 49).show(5)
+----------+---+----------+
|first_name|age|(age > 49)|
+----------+---+----------+
|      Erin| 42|     false|
|    Claire| 23|     false|
|    Norman| 81|      true|
|    Miguel| 64|      true|
|  Rosalita| 14|     false|
+----------+---+----------+
```

# select()

The DSL also allows you create on-the-fly *derived columns*.

```
scala> df.select($"firstName",
                 $"age",
                 $"age" > 49,
                 $"age" + 10).show(5)
+---------+---+----------+----------+
|firstName|age|(age > 49)|(age + 10)|
+---------+---+----------+----------+
|     Erin| 42|     false|        52|
|   Claire| 23|     false|        33|
|   Norman| 81|      true|        91|
|   Miguel| 64|      true|        74|
| Rosalita| 14|     false|        24|
+---------+---+----------+----------+
```

# filter()

The `filter()` method allows you to filter rows out of your results.

```
scala> df.filter($"age" > 49).
          select($"first_name", $"age").
          show()
+---------+---+
|firstName|age|
+---------+---+
|   Norman| 81|
|   Miguel| 64|
|  Abigail| 75|
+---------+---+
```

# filter()

The `filter()` method allows you to filter rows out of your results.

```
scala> df.filter($"age" > 49).select($"firstName", $"age").show()
+---------+---+
|firstName|age|
+---------+---+
|   Norman| 81|
|   Miguel| 64|
|  Abigail| 75|
+---------+---+
```

# orderBy()

The `orderBy()` method allows you to sort the results.

```
scala> df.filter($"age" > 49).
          select($"firstName", $"age").
          orderBy($"age", $"firstName").
          show()
+---------+---+
|firstName|age|
+---------+---+
|   Miguel| 64|
|  Abigail| 75|
|   Norman| 81|
+---------+---+
```

# orderBy()

It's easy to reverse the sort order.

```
scala> df.filter($"age" > 49).
          select($"firstName", $"age").
          orderBy($"age".desc, $"firstName").
          show()
+---------+---+
|firstName|age|
+---------+---+
|   Norman| 81|
|  Abigail| 75|
|   Miguel| 64|
+---------+---+
```

# Hands-on with DataFrames

In the Databricks shard, look in

Essentials > sql-and-dataframes

In there, you'll find a **`hands-on`** folder, with Python and Scala notebooks beneath it.

- Clone either the Python or Scala notebook to your home folder
- Open the cloned notebook
- Attach the notebook to your cluster
- Follow the instructions in the notebook

# as() or alias()

- `as()` or `alias()` allows you to rename a column.
- It's especially useful with generated columns.

```
scala> df.select($"firstName", $"age", ($"age" < 30).as("young")).show()
+---------+---+-----+
|firstName|age|young|
+---------+---+-----+
|     Erin| 42|false|
|   Claire| 23| true|
|   Norman| 81|false|
|   Miguel| 64|false|
| Rosalita| 14| true|
+---------+---+-----+
```

# as() or alias()

```
In [7]: df.select(df['first_name'],\
                  df['age'],\
                  (df['age'] < 30).alias('young')).show(5)
+----------+---+-----+
|first_name|age|young|
+----------+---+-----+
|      Erin| 42|false|
|    Claire| 23| true|
|    Norman| 81|false|
|    Miguel| 64|false|
|  Rosalita| 14| true|
+----------+---+-----+
```



Note: In Python, you *must* use `alias`, because `as` is a keyword.

# as()

And, of course, SQL:

```
sqlContext.sql("SELECT firstName, age, age < 30 AS young FROM names")
```

```
+---------+---+-----+
|firstName|age|young|
+---------+---+-----+
|     Erin| 42|false|
|   Claire| 23| true|
|   Norman| 81|false|
|   Miguel| 64|false|
| Rosalita| 14| true|
+---------+---+-----+
```

# groupBy()

Often used with `count()`, `groupBy()` groups data items by a specific column value.

```
> df.groupBy("age").count().show()
+---+-----+
|age|count|
+---+-----+
| 39|    1|
| 42|    2|
| 64|    1|
| 75|    1|
| 81|    1|
| 14|    1|
| 23|    2|
+---+-----+
```

# groupBy()

- And SQL, of course, isn't surprising:

```
scala> sqlContext.sql("SELECT age, count(age) FROM names "GROUP BY age")
+---+-----+
|age|count|
+---+-----+
| 39|    1|
| 42|    2|
| 64|    1|
| 75|    1|
| 81|    1|
| 14|    1|
| 23|    2|
+---+-----+
```

# Joins

Let's assume we have a second file, a JSON file with records like this:

```
[
  {
    "firstName": "Erin",
    "lastName": "Shannon",
    "medium": "oil on canvas"
  },
  {
    "firstName": "Norman",
    "lastName": "Lockwood",
    "medium": "metal (sculpture)"
  },
  …
]
```

# Joins

We can load that into a second DataFrame and join it with our first one.

```
scala> val df2 = sqlContext.read.json("artists.json")
# Schema inferred as DataFrame[firstName: string, lastName: string, medium: string]
scala> df.join(df2, (df("first_name") === df2("firstName") & (df("last_name" === df2("lastName")).show()
+----------+---------+------+---+---------+--------+----------------+
|first_name|last_name|gender|age|firstName|lastName|          medium|
+----------+---------+------+---+---------+--------+----------------+
|    Norman| Lockwood|     M| 81|   Norman|Lockwood|metal (sculpture)|
|      Erin|  Shannon|     F| 42|     Erin| Shannon|    oil on canvas|
|  Rosalita|  Ramirez|     F| 14| Rosalita| Ramirez|         charcoal|
|    Miguel|     Ruiz|     M| 64|   Miguel|    Ruiz|    oil on canvas|
+----------+---------+------+---+---------+--------+----------------+
```

# Joins

We can load that into a second DataFrame and join it with our first one.

```
In [1]: df2 = sqlContext.read.json("artists.json")
# Schema inferred as DataFrame[firstName: string, lastName: string, medium: string]
In [2]: df.join(
            df2,
            df.first_name == df2.firstName and df.last_name == df2.lastName
        ).show()
+----------+---------+------+---+---------+--------+----------------+
|first_name|last_name|gender|age|firstName|lastName|          medium|
+----------+---------+------+---+---------+--------+----------------+
|    Norman| Lockwood|     M| 81|   Norman|Lockwood|metal (sculpture)|
|      Erin|  Shannon|     F| 42|     Erin| Shannon|    oil on canvas|
|  Rosalita|  Ramirez|     F| 14| Rosalita| Ramirez|         charcoal|
|    Miguel|     Ruiz|     M| 64|   Miguel|    Ruiz|    oil on canvas|
+----------+---------+------+---+---------+--------+----------------+
```

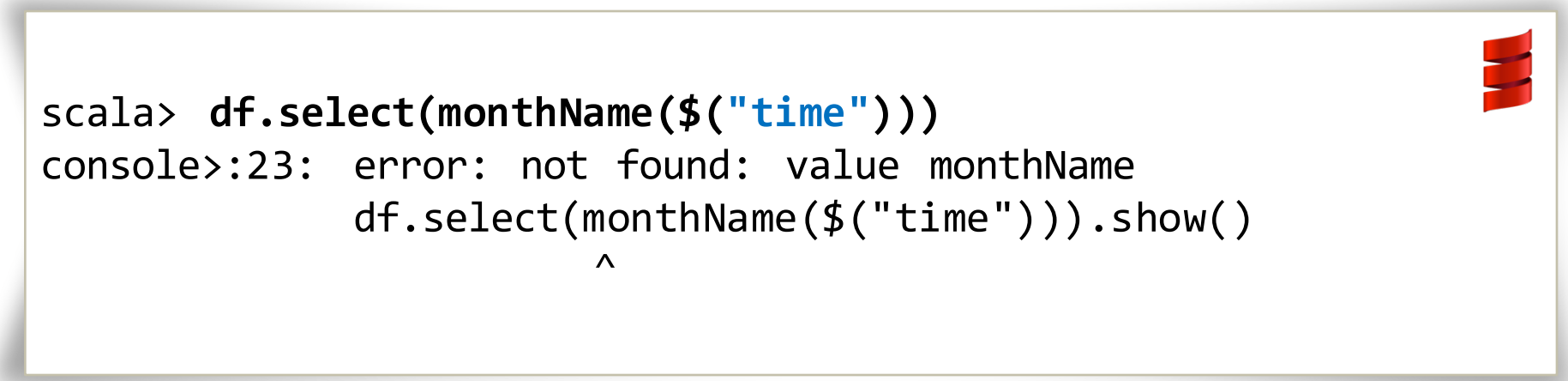
# Joins

Let's make that a little more readable by only selecting *some* of the columns.

```
scala> val df3 = df.join(
         df2, (df("first_name") === df2("firstName") & (df("last_name") === df2("lastName")
       )
scala> df3.select("first_name", "last_name", "age", "medium").show()

+----------+---------+---+-----------------+
|first_name|last_name|age|           medium|
+----------+---------+---+-----------------+
|    Norman| Lockwood| 81|metal (sculpture)|
|      Erin|  Shannon| 42|    oil on canvas|
|  Rosalita|  Ramirez| 14|         charcoal|
|    Miguel|     Ruiz| 64|    oil on canvas|
+----------+---------+---+-----------------+
```

# User Defined Functions

Let's invent a function that doesn't exist. We'll create a function to extract the month field from a timestamp-typed column and return the month name.



```
scala> df.select(monthName($("time")))
console>:23: error: not found: value monthName
             df.select(monthName($("time"))).show()
                       ^
```

# User Defined Functions

Let's fix that problem.

```
scala> val monthName = sqlContext.udf.register("monthName", (t: Timestamp) => {
  import java.text.SimpleDateFormat
  import java.util.Date
  val fmt = new SimpleDateFormat("MMM")
  fmt.format(new Date(t.getTime))
})
monthName: org.apache.spark.sql.UserDefinedFunction =
UserDefinedFunction(<function1>,StringType,List())

scala> df.select(monthName($"time")).show(5)
+---------+
|UDF(time)|
+---------+
|      Oct|
|      Oct|
+---------+
```

# User Defined Functions

## And… in Python

```
In [8]: from pyspark.sql.functions import udf
In [9]: from datetime import datetime
In [10]: month_name = udf(lambda d: datetime.strftime(d, "%b"))
In [11]: df.select(month_name(df['birth_date'])).show(5)
+------------------------------+
|PythonUDF#<lambda>(birth_date)|
+------------------------------+
|                           Jan|
|                           Feb|
|                           Dec|
|                           Aug|
|                           Aug|
+------------------------------+
```

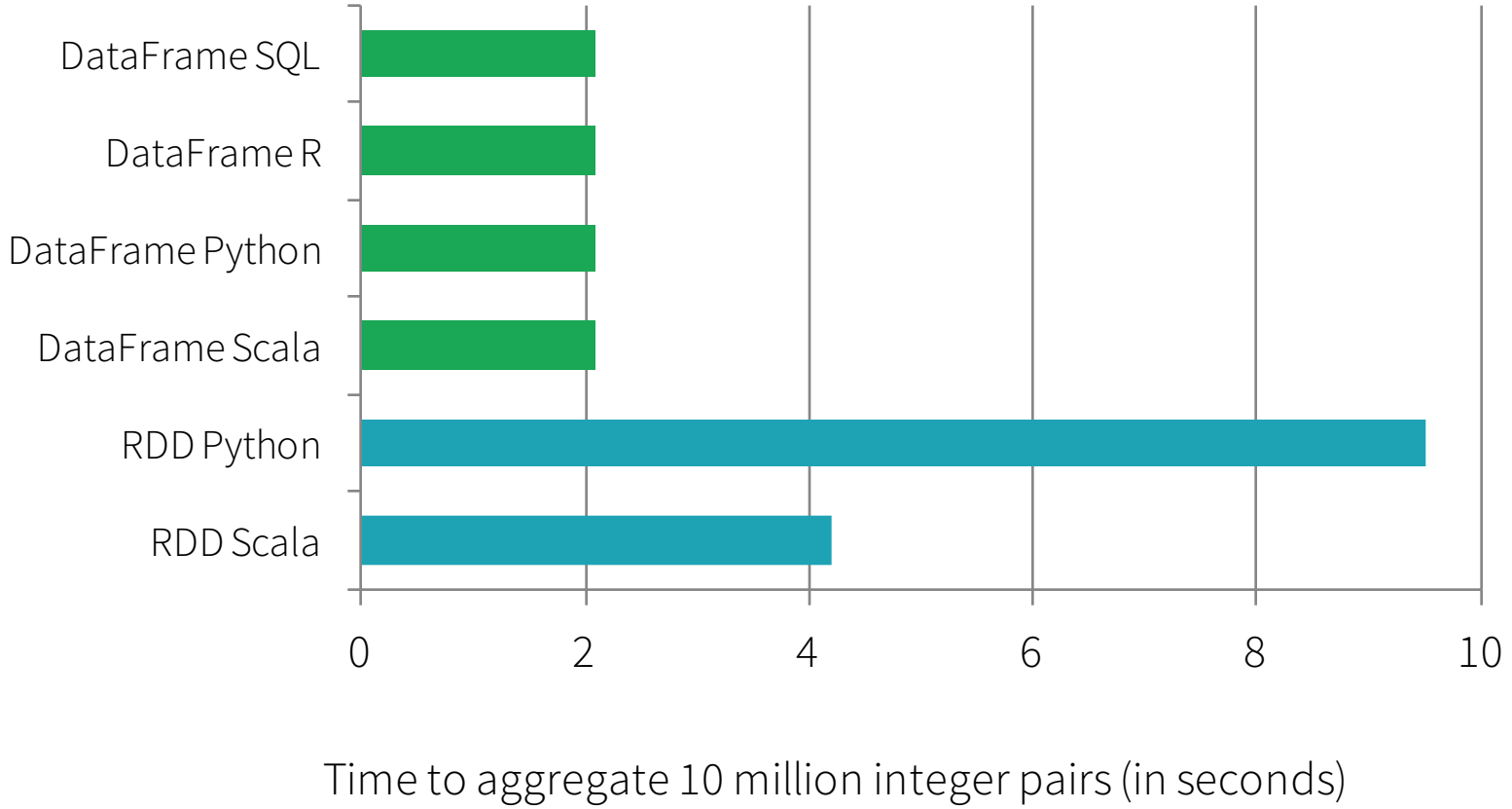
`alias()` would "fix" this generated column name.

# Writing DataFrames

You can write DataFrames out, as well. When doing ETL, this is a *very* common requirement.

- In most cases, if you can read a data format, you can write that data format, as well.
- If you're writing to a text file format (e.g., JSON), you'll typically get multiple output files.

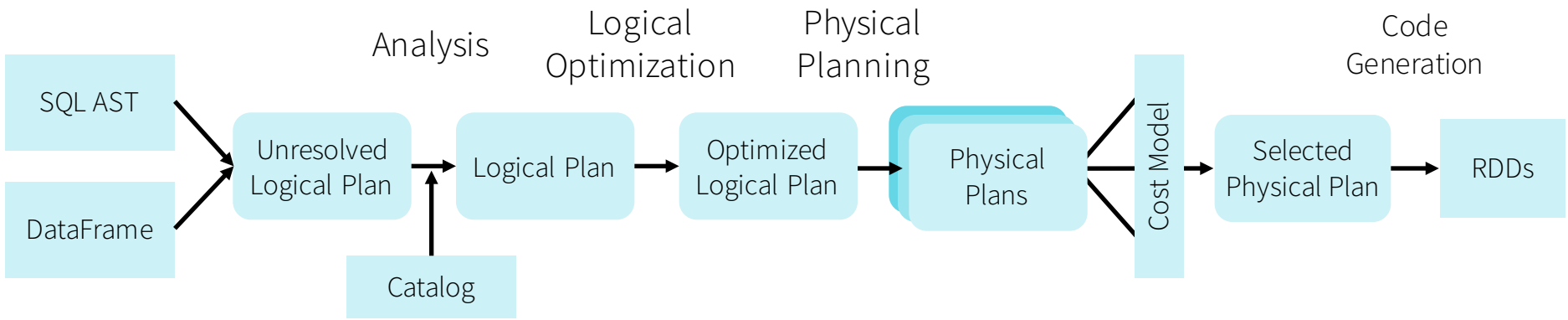
# DataFrames can be *significantly* faster than RDDs. And they perform the same, regardless of language.



# Plan Optimization & Execution

- Represented internally as a “logical plan”
- Execution is lazy, allowing it to be optimized by Catalyst

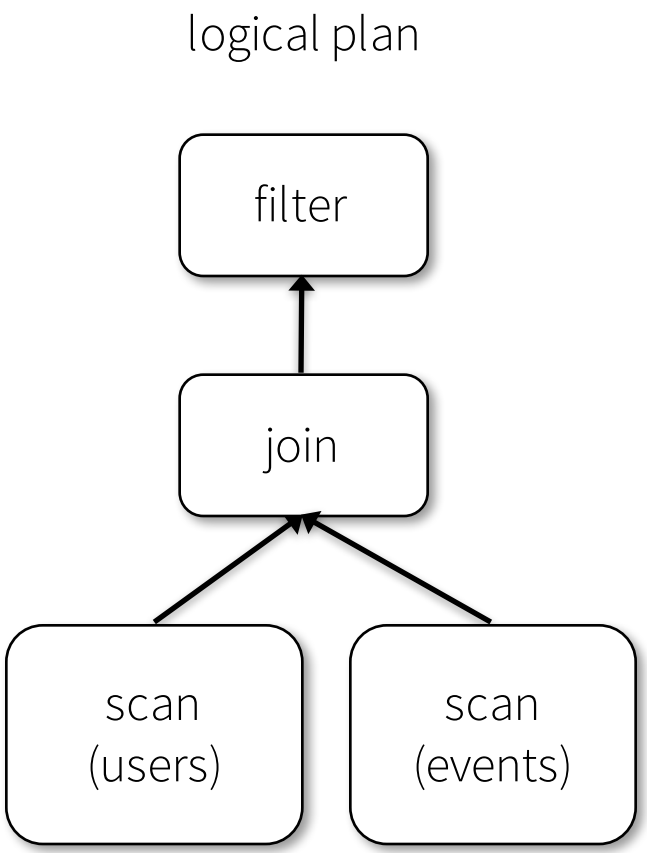
# Plan Optimization & Execution




DataFrames and SQL share the same optimization/execution pipeline

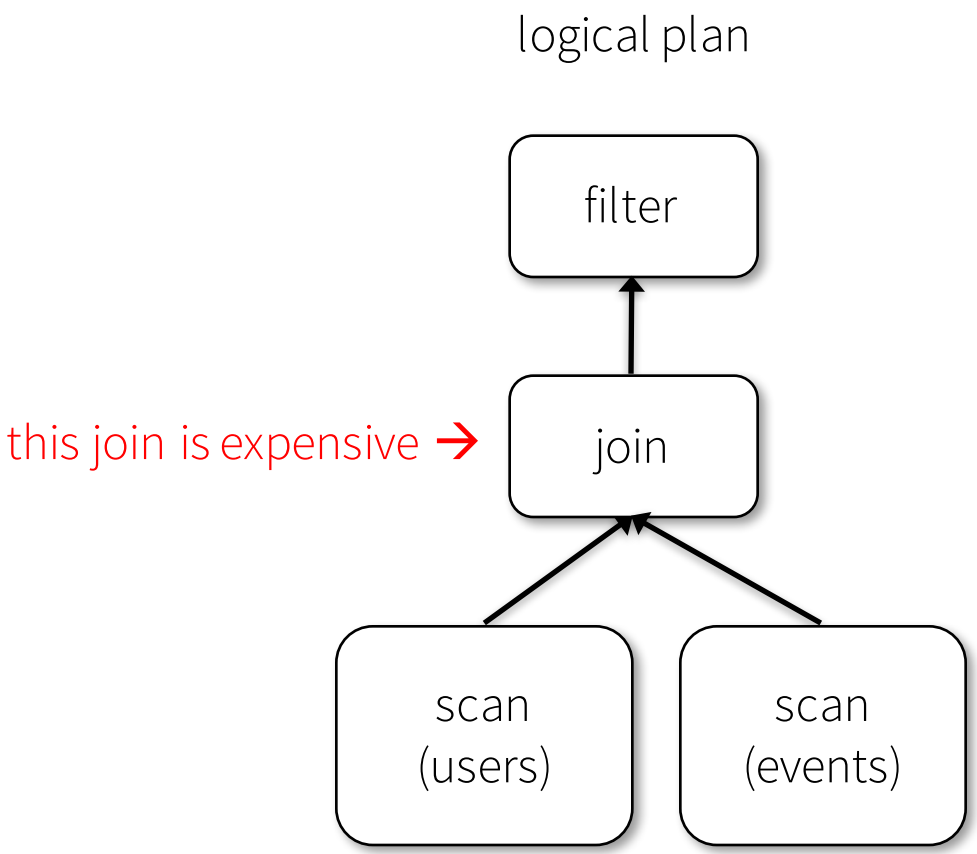
# Plan Optimization & Execution

```
joined = users.join(events, users.id == events.uid)
filtered = joined.filter(events.date >= "2015-01-01")
```



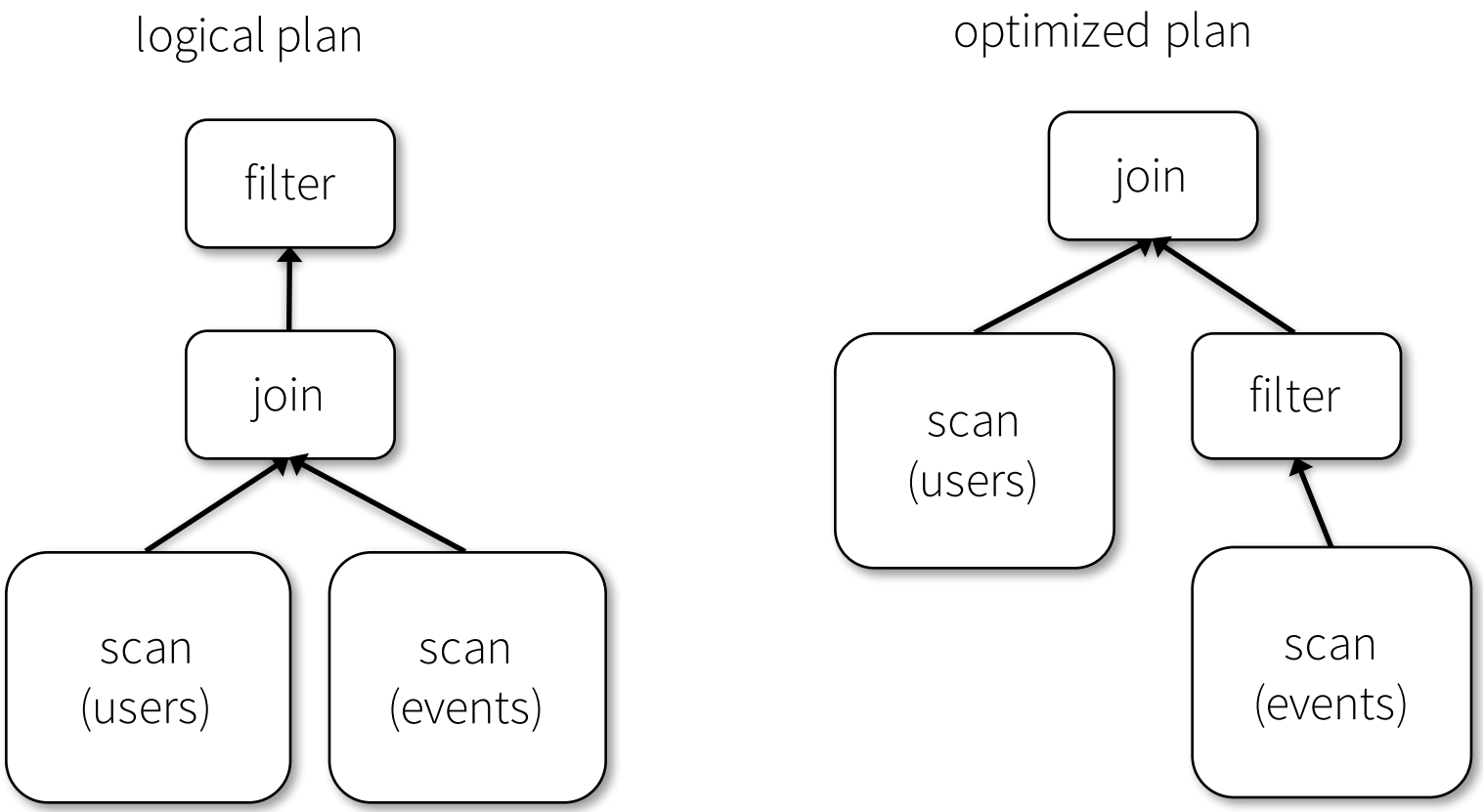
# Plan Optimization & Execution

```
joined = users.join(events, users.id == events.uid)
filtered = joined.filter(events.date >= "2015-01-01")
```



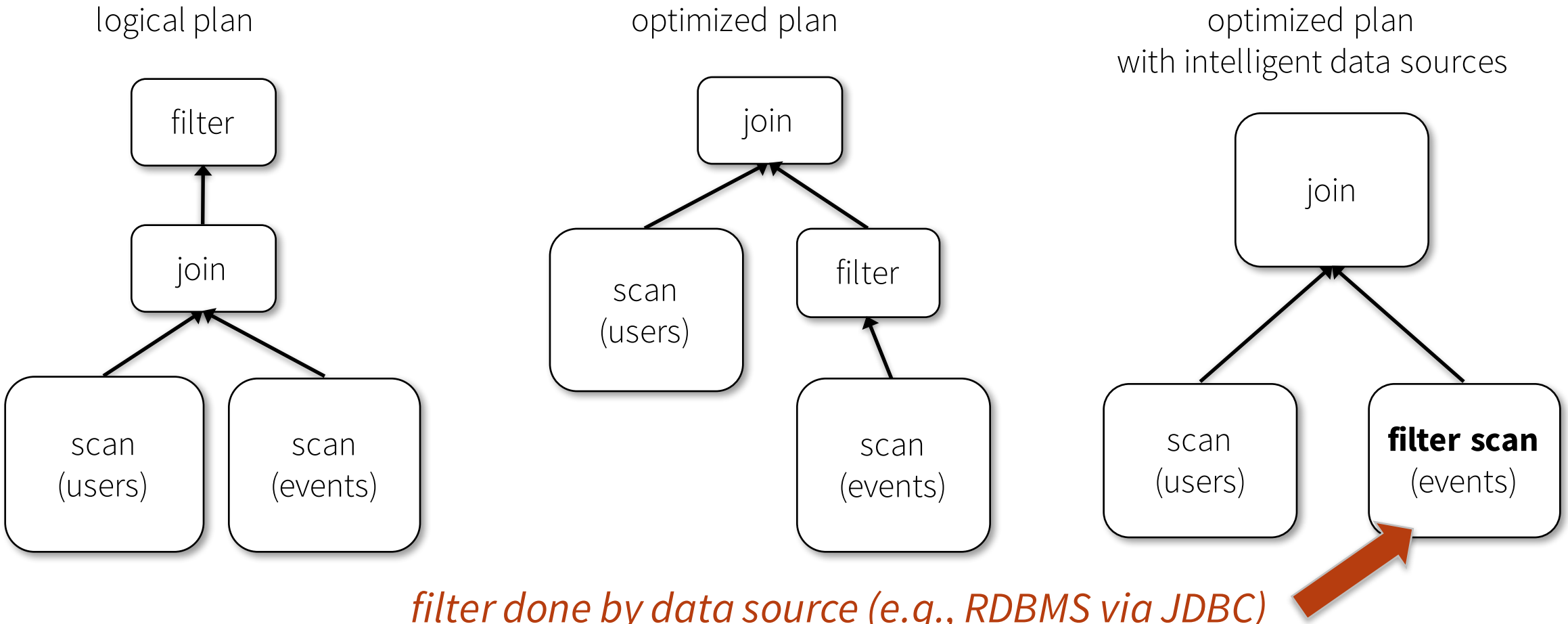
# Plan Optimization & Execution

```
joined = users.join(events, users.id == events.uid)
filtered = joined.filter(events.date >= "2015-01-01")
```



# Plan Optimization & Execution

```
joined = users.join(events, users.id == events.uid)
filtered = joined.filter(events.date >= "2015-01-01")
```



# Plan Optimization: “Intelligent” Data Sources

- The Data Sources API can automatically prune columns and push filters to the source

  **Parquet:** skip irrelevant columns and blocks of data; turn string comparison into integer comparisons for dictionary encoded data

  **JDBC:** Rewrite queries to push predicates down

# Explain

- You can dump the query plan to standard output, so you can get an idea of how Spark will execute your query.

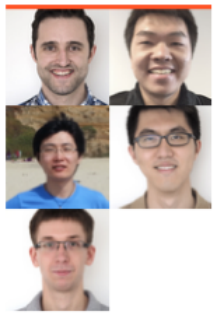
```
scala> val df3 = df.join(df2,
  (df("first_name") === df2("firstName")) & (df("last_name") === df2("lastName"))
scala> df3.explain()

ShuffledHashJoin [last_name#18], [lastName#36], BuildRight
 Exchange (HashPartitioning 200)
  PhysicalRDD [first_name#17,last_name#18,gender#19,age#20L], MapPartitionsRDD[41]
at applySchemaToPythonRDD at NativeMethodAccessorImpl.java:-2
 Exchange (HashPartitioning 200)
  PhysicalRDD [firstName#35,lastName#36,medium#37], MapPartitionsRDD[118] at
executedPlan at NativeMethodAccessorImpl.java:-2
```

# Explain

- Pass `true` to get a more detailed query plan.

```
scala> df.join(df2, lower(df("firstName")) === lower(df2("firstName"))).explain(true)
== Parsed Logical Plan ==
Join Inner, Some((Lower(firstName#1) = Lower(firstName#13)))
 Relation[birthDate#0,firstName#1,gender#2,lastName#3,middleName#4,salary#5L,ssn#6]
org.apache.spark.sql.json.JSONRelation@7cbb370e
 Relation[firstName#13,lastName#14,medium#15] org.apache.spark.sql.json.JSONRelation@e5203d2c

== Analyzed Logical Plan ==
birthDate: string, firstName: string, gender: string, lastName: string, middleName: string, salary: bigint, ssn: string,
firstName: string, lastName: string, medium: string
Join Inner, Some((Lower(firstName#1) = Lower(firstName#13)))
 Relation[birthDate#0,firstName#1,gender#2,lastName#3,middleName#4,salary#5L,ssn#6]
org.apache.spark.sql.json.JSONRelation@7cbb370e
 Relation[firstName#13,lastName#14,medium#15] org.apache.spark.sql.json.JSONRelation@e5203d2c

== Optimized Logical Plan ==
Join Inner, Some((Lower(firstName#1) = Lower(firstName#13)))
 Relation[birthDate#0,firstName#1,gender#2,lastName#3,middleName#4,salary#5L,ssn#6]
org.apache.spark.sql.json.JSONRelation@7cbb370e
 Relation[firstName#13,lastName#14,medium#15] org.apache.spark.sql.json.JSONRelation@e5203d2c

== Physical Plan ==
ShuffledHashJoin [Lower(firstName#1)], [Lower(firstName#13)], BuildRight
 Exchange (HashPartitioning 200)
  PhysicalRDD [birthDate#0,firstName#1,gender#2,lastName#3,middleName#4,salary#5L,ssn#6], MapPartitionsRDD[40] at explain
at <console>:25
 Exchange (HashPartitioning 200)
  PhysicalRDD [firstName#13,lastName#14,medium#15], MapPartitionsRDD[43] at explain at <console>:25

Code Generation: false
== RDD ==
```

# Catalyst Internals



## Deep Dive into Spark SQL's Catalyst Optimizer

April 13, 2015 | by Michael Armbrust, Yin Huai, Cheng Liang, Reynold Xin and Matei Zaharia

Spark SQL is one of the newest and most technically involved components of Spark. It powers both SQL queries and the new DataFrame API. At the core of Spark SQL is the Catalyst optimizer, which leverages advanced programming language features (e.g. Scala's pattern matching and quasiquotes) in a novel way to build an extensible query optimizer.

We recently published a paper on Spark SQL that will appear in SIGMOD 2015 (co-authored with Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, and Ali Ghodsi). In this blog post we are republishing a section in the paper that explains the internals of the Catalyst optimizer for broader consumption.

https://databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html

# DataFrame limitations

- Catalyst does not automatically repartition DataFrames optimally
- During a DF shuffle, Spark SQL will just use
  - `spark.sql.shuffle.partitions`
  - to determine the number of partitions in the downstream RDD
- All SQL configurations can be changed
  - via `sqlContext.setConf(key, value)`
  - or in Databricks: `"%sql SET key=val"`

# Machine Learning Integration

- Spark 1.2 introduced a new package called **`spark.ml`**, which aims to provide a uniform set of high-level APIs that help users create and tune practical machine learning pipelines.

- Spark ML standardizes APIs for machine learning algorithms to make it easier to combine multiple algorithms into a single *pipeline*, or *workflow*.

# Machine Learning Integration

- Spark ML uses DataFrames as a dataset which can hold a variety of data types.

- For instance, a dataset could have different columns storing text, feature vectors, true labels, and predictions.

# More hands-on with DataFrames

In the Databricks shard, look in

Essentials > sql-and-dataframes

In there, you'll find a **`lab01`** folder, with Python and Scala folders beneath it.

- Clone either the Python or Scala folder to your home folder
- Open the lab notebook underneath the cloned folder
- Attach the notebook to your cluster
- Follow the instructions in the notebook

If you have problems with any of the exercises, ask for help from a TA, or consult the Solutions notebook in the cloned folder.

# End of DataFrames and Spark SQL

# App ➔ Jobs ➔ Stages ➔ Tasks


databricks™

"The key to tuning Spark apps is a sound grasp of Spark's internal mechanisms"

- Patrick Wendell, Databricks

Founder, Spark Committer, Spark PMC

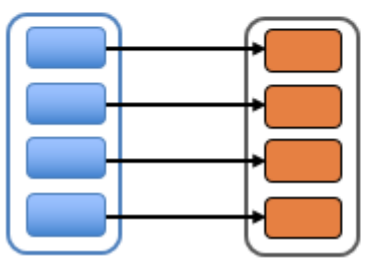
# Terminology

- Job: The work required to compute an RDD.
- Stage: A wave of work within a job, corresponding to one or more pipelined RDD's
- Tasks: A unit of work within a stage, corresponding to one RDD partition
- Shuffle: The transfer of data between stages

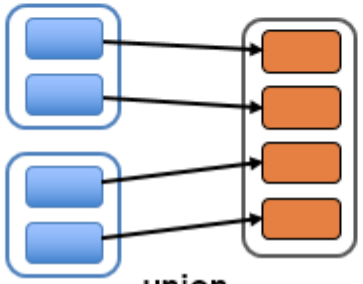
# Narrow vs. Wide Dependencies

## narrow

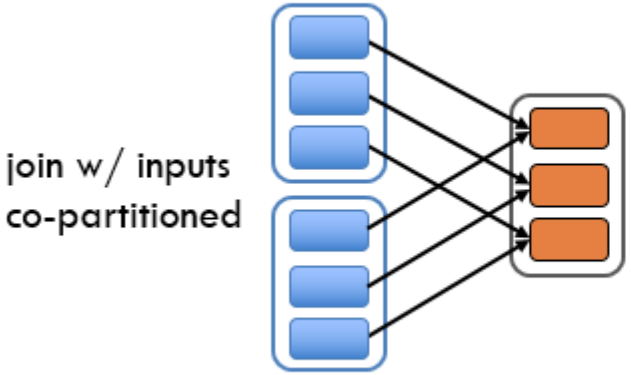
*each partition of the parent RDD is used by at most one partition of the child RDD*



map, filter



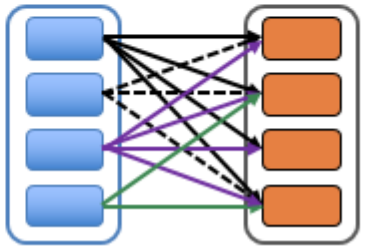
union



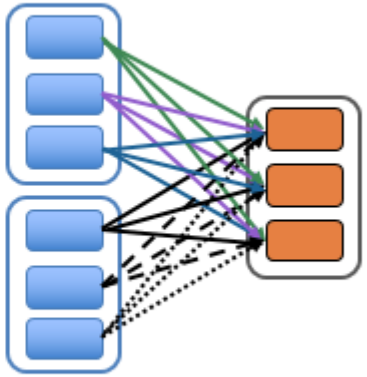
join w/ inputs co-partitioned

## wide

*multiple child RDD partitions may depend on a single parent RDD partition*



groupByKey

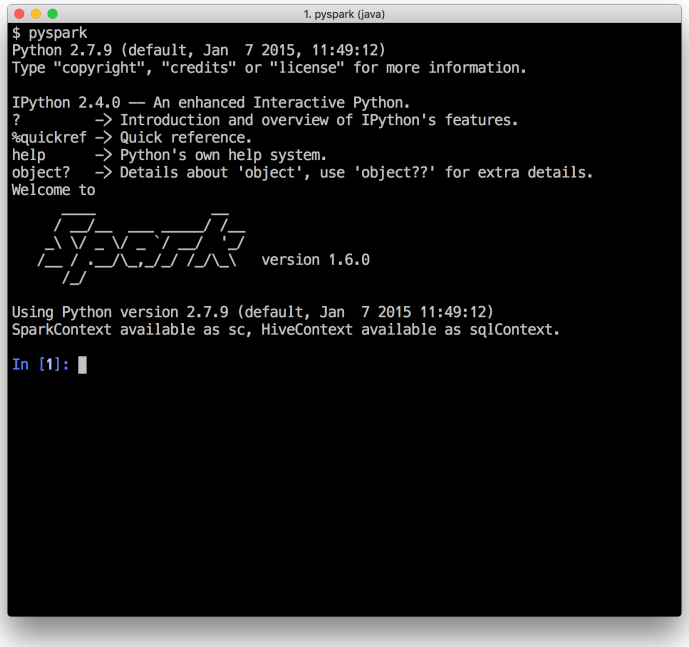


join w/ inputs not co-partitioned

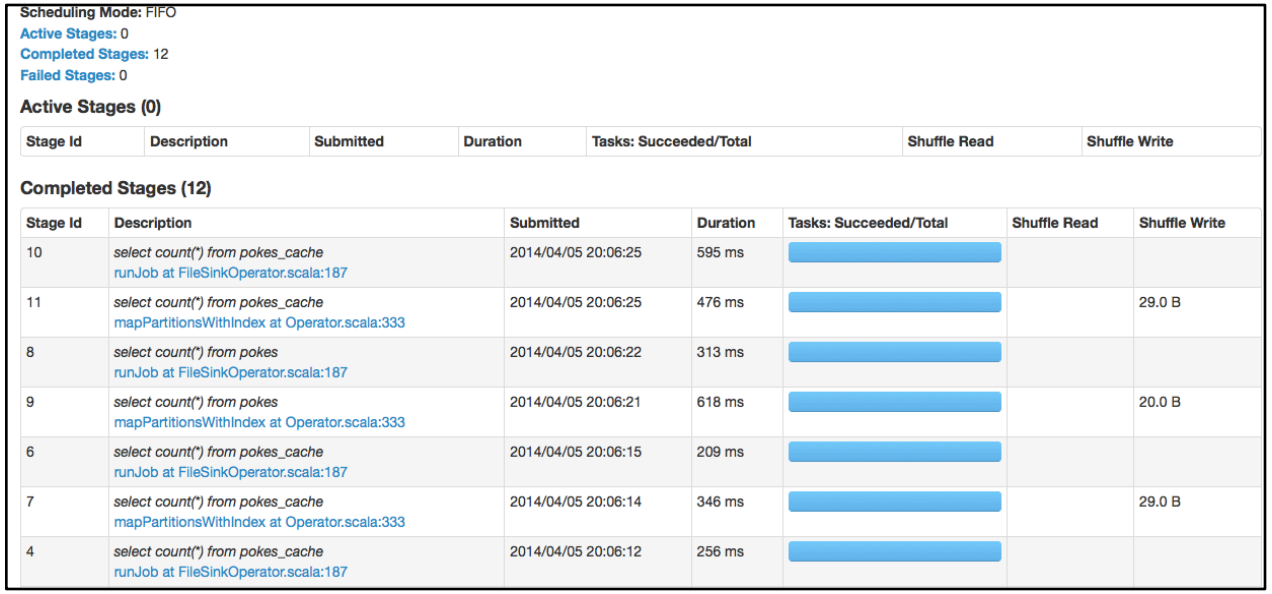
# Planning Physical Execution

How does a user program get translated into units of physical execution?

application >> jobs >> stages >> tasks







**Scheduling Mode:** FIFO
**Active Stages:** 0
**Completed Stages:** 12
**Failed Stages:** 0

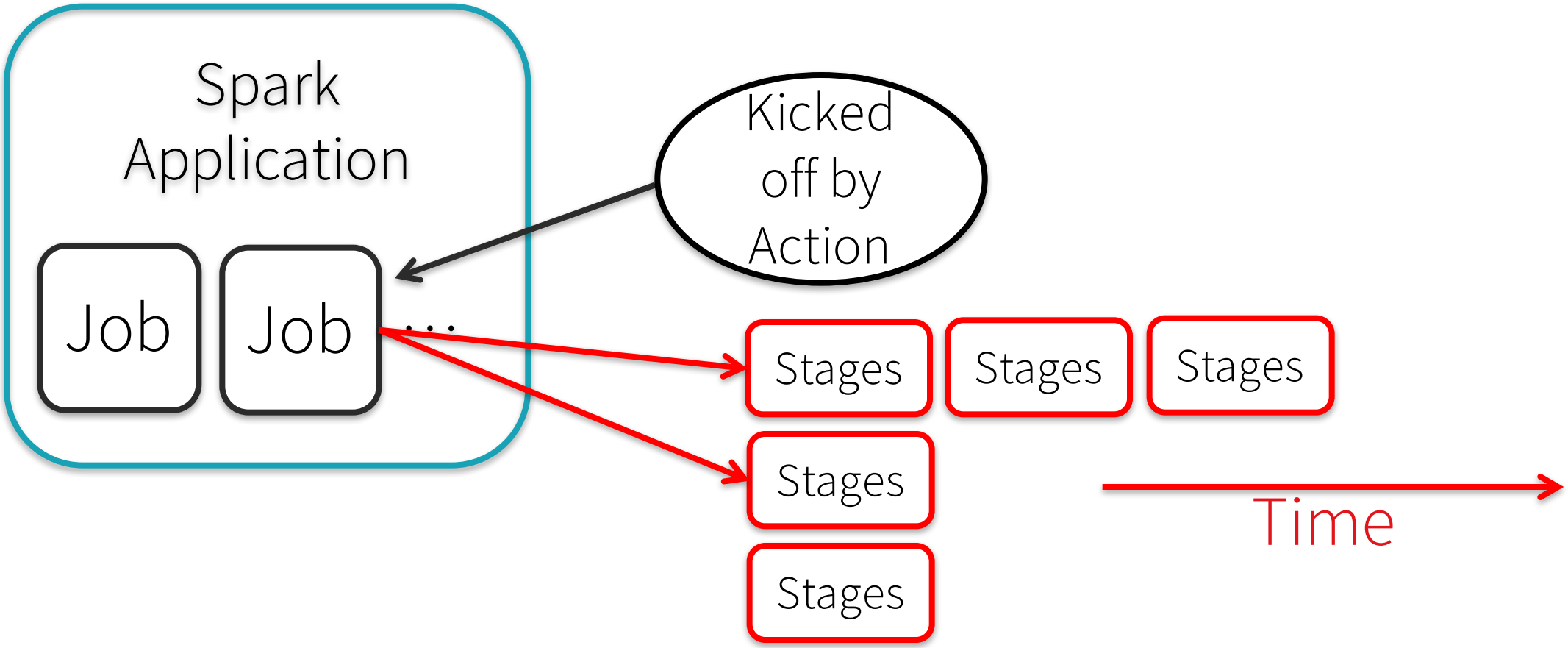
**Active Stages (0)**

| Stage Id | Description | Submitted | Duration | Tasks: Succeeded/Total | Shuffle Read | Shuffle Write |
|---|---|---|---|---|---|---|

**Completed Stages (12)**

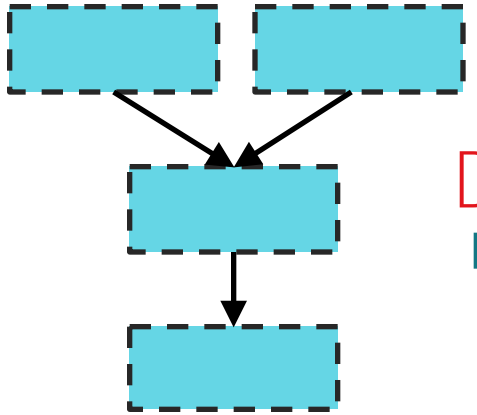
| Stage Id | Description | Submitted | Duration | Tasks: Succeeded/Total | Shuffle Read | Shuffle Write |
|---|---|---|---|---|---|---|
| 10 | *select count(*) from pokes_cache*<br>runJob at FileSinkOperator.scala:187 | 2014/04/05 20:06:25 | 595 ms | | | |
| 11 | *select count(*) from pokes_cache*<br>mapPartitionsWithIndex at Operator.scala:333 | 2014/04/05 20:06:25 | 476 ms | | | 29.0 B |
| 8 | *select count(*) from pokes*<br>runJob at FileSinkOperator.scala:187 | 2014/04/05 20:06:22 | 313 ms | | | |
| 9 | *select count(*) from pokes*<br>mapPartitionsWithIndex at Operator.scala:333 | 2014/04/05 20:06:21 | 618 ms | | | 20.0 B |
| 6 | *select count(*) from pokes_cache*<br>runJob at FileSinkOperator.scala:187 | 2014/04/05 20:06:15 | 209 ms | | | |
| 7 | *select count(*) from pokes_cache*<br>mapPartitionsWithIndex at Operator.scala:333 | 2014/04/05 20:06:14 | 346 ms | | | 29.0 B |
| 4 | *select count(*) from pokes_cache*<br>runJob at FileSinkOperator.scala:187 | 2014/04/05 20:06:12 | 256 ms | | | |

# Scheduling Process



# Scheduling Process

RDD Objects

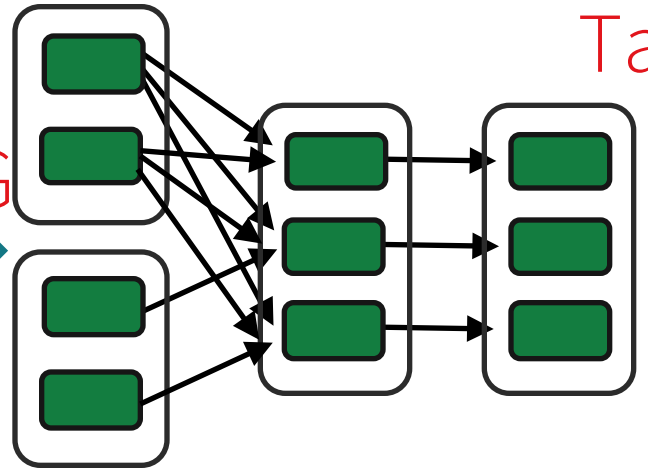


```
Rdd1.join(rdd2)
  .groupBy(…)
  .filter(…)
```

- Build operator DAG

DAG Scheduler

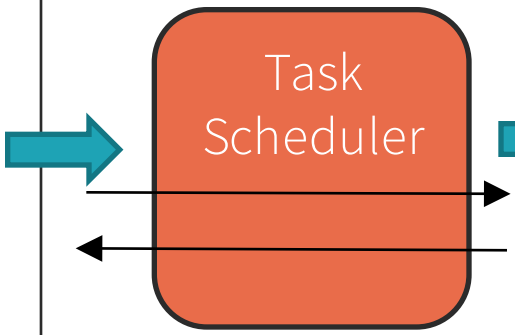
DAG



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

Task Scheduler

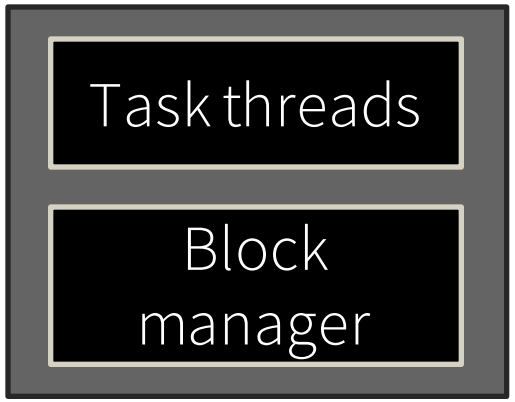
TaskSet




- Launches individual tasks
- Retry failed or straggling tasks

Executor

Task




- Execute tasks
- Store and serve blocks

# RDD API Example


```
INFO Server started
INFO Bound to port 8080

WARN Cannot find srv.conf
```

input.txt

```
// Read input file
val input = sc.textFile("input.txt")

val tokenized = input
  .map(line => line.split(" "))
  .filter(words => words.size > 0) // remove empty lines

val counts = tokenized   // frequency of log levels
  .map(words => (words(0), 1))
  .reduceByKey( (a, b) => a + b, 2 )
```

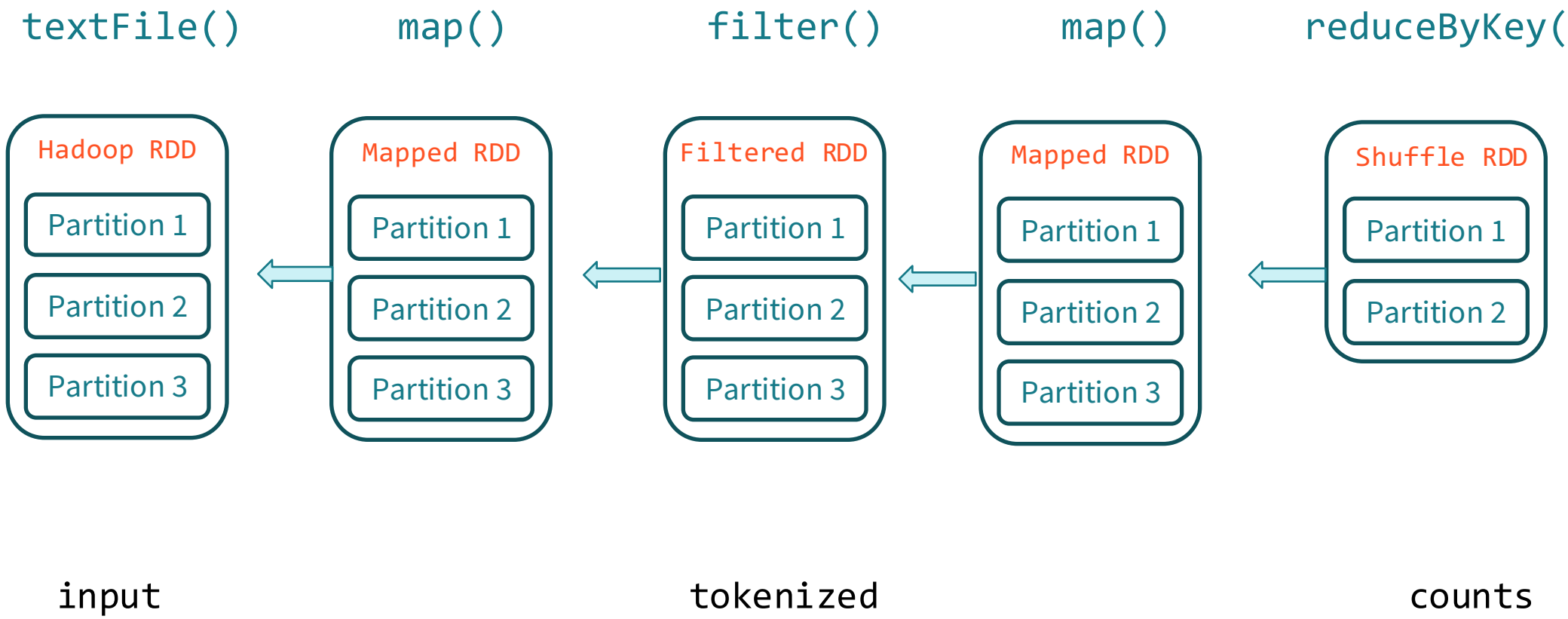
# RDD API Example

```
// Read input file
val input = sc.textFile(                    )

val tokenized = input
  .map(                                )
  .filter(                                ) // remove empty lines

val counts = tokenized   // frequency of log levels
  .map(                                )
  .reduceByKey{                                }
```

# Transformations

```
sc.textFile().map().filter().map().reduceByKey()
```

# DAG View of RDDs



# Evaluation of the DAG

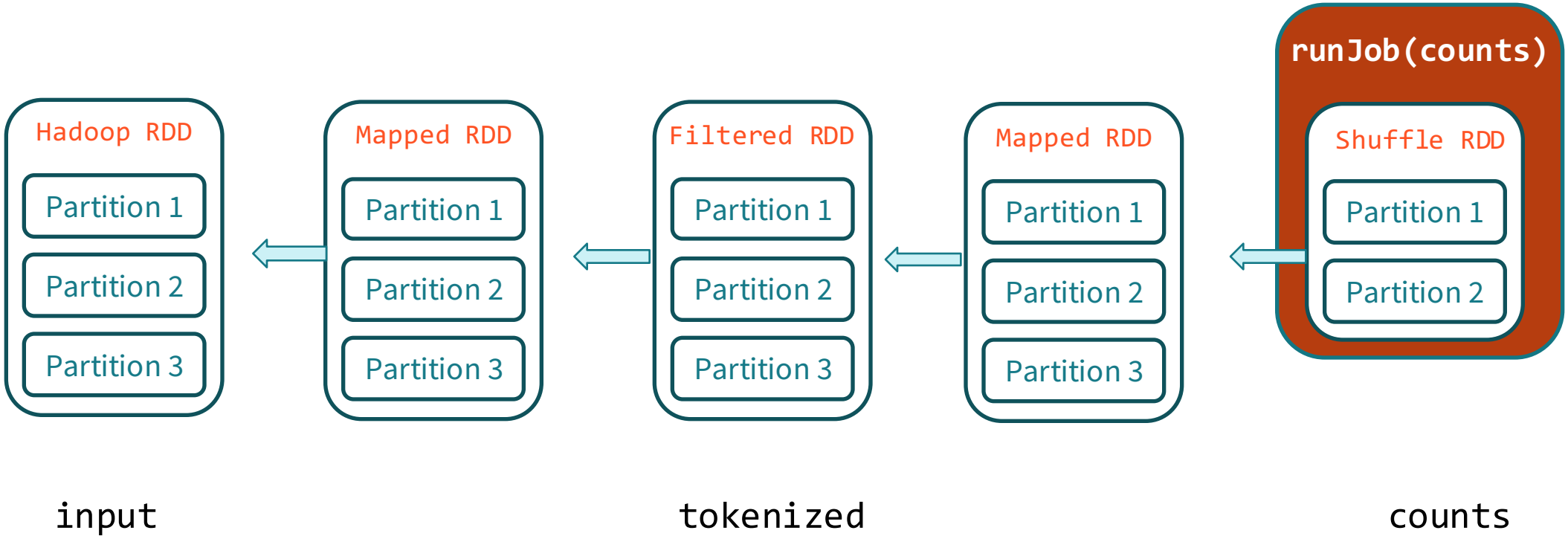
DAGs are materialized through a method `sc.runJob`:

```
def runJob[T, U](
    rdd: RDD[T],                          1. RDD to compute
    partitions: Seq[Int],                 2. Which partitions
    func: (Iterator[T]) => U))            3. Fn to produce results
: Array[U]                                          → results
```

*for each partition*

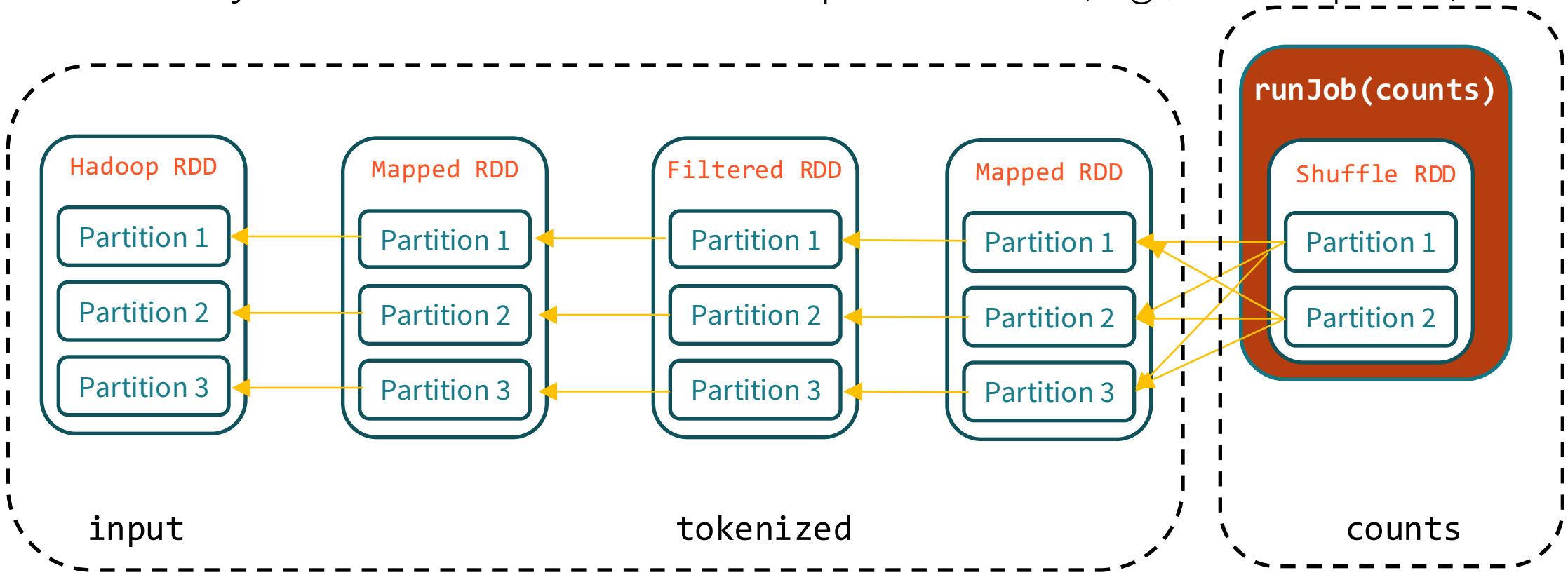
# How runJob Works

Needs to compute target's parents, parents' parents, etc.
… all the way back to an RDD with no dependencies (e.g., HadoopRDD)



# How runJob Works

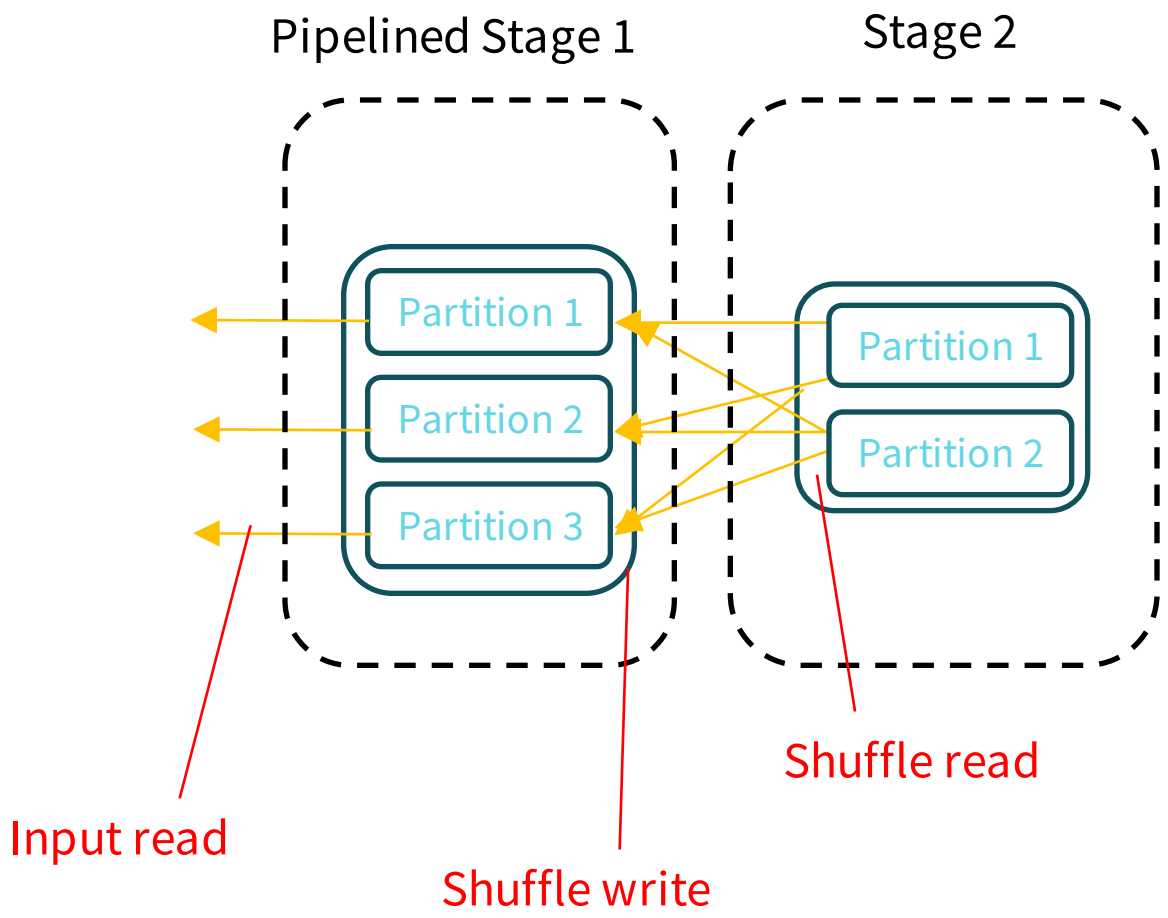
Needs to compute target's parents, parents' parents, etc.
… all the way back to an RDD with no dependencies (e.g., HadoopRDD)



# Stage Graph

Each task will:

1) Read Hadoop input
2) Perform maps & filters
3) Write partial sums



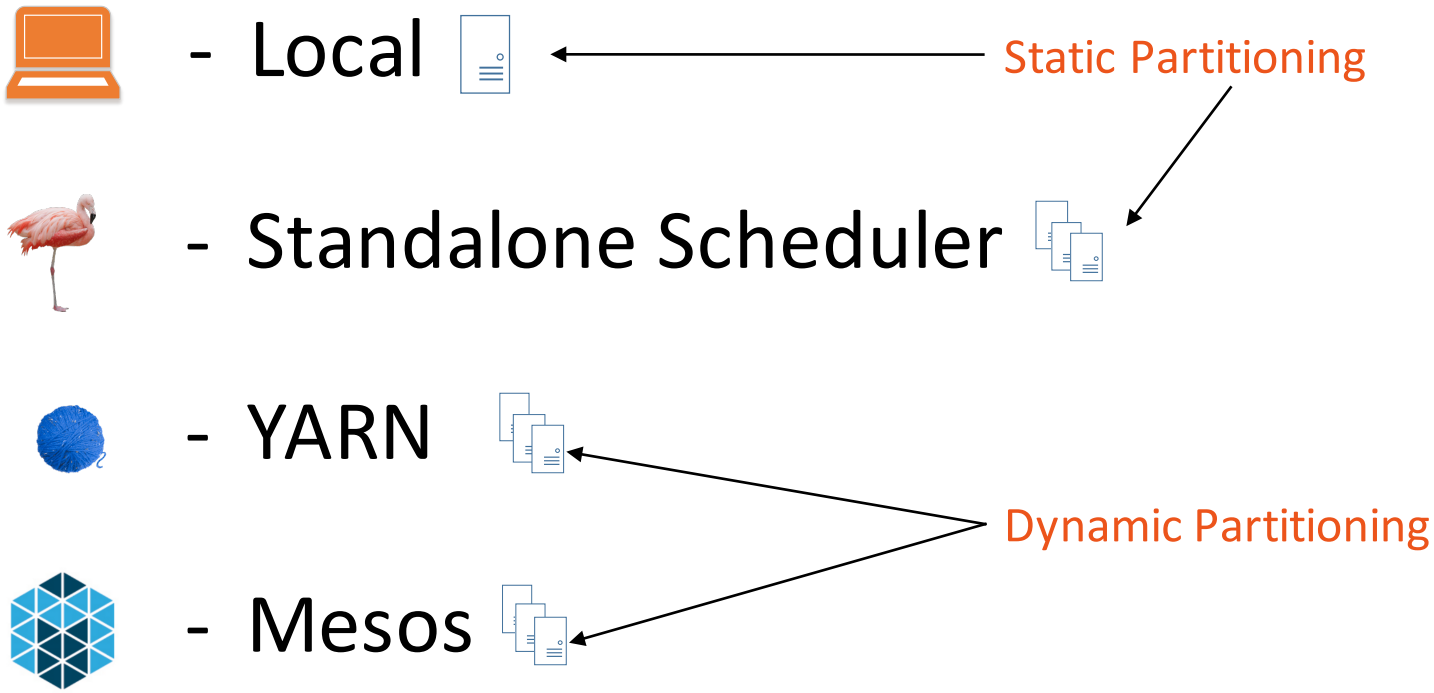

Each task will:

1) Read partial sums
2) Invoke user function passed to runJob

# End of Spark DAG

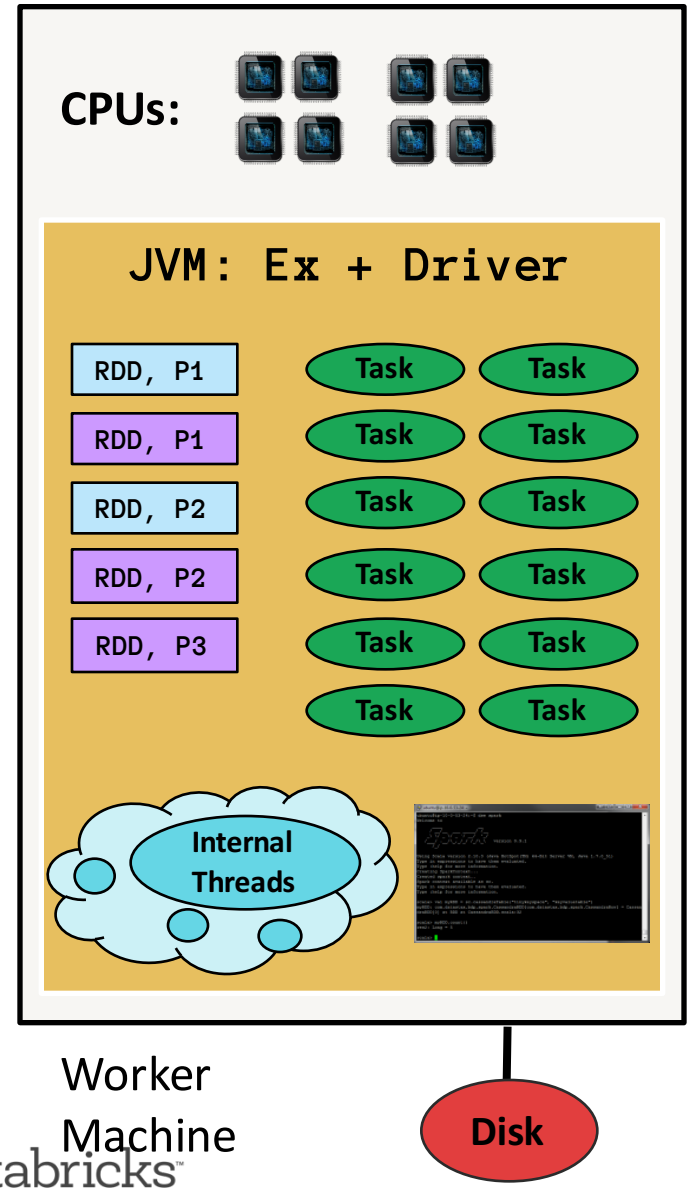
# Spark Clusters: Architecture & Deployment

# Ways to Run Spark



# Local Mode

# Local Mode




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

3 options: local, local[N], local[*]

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

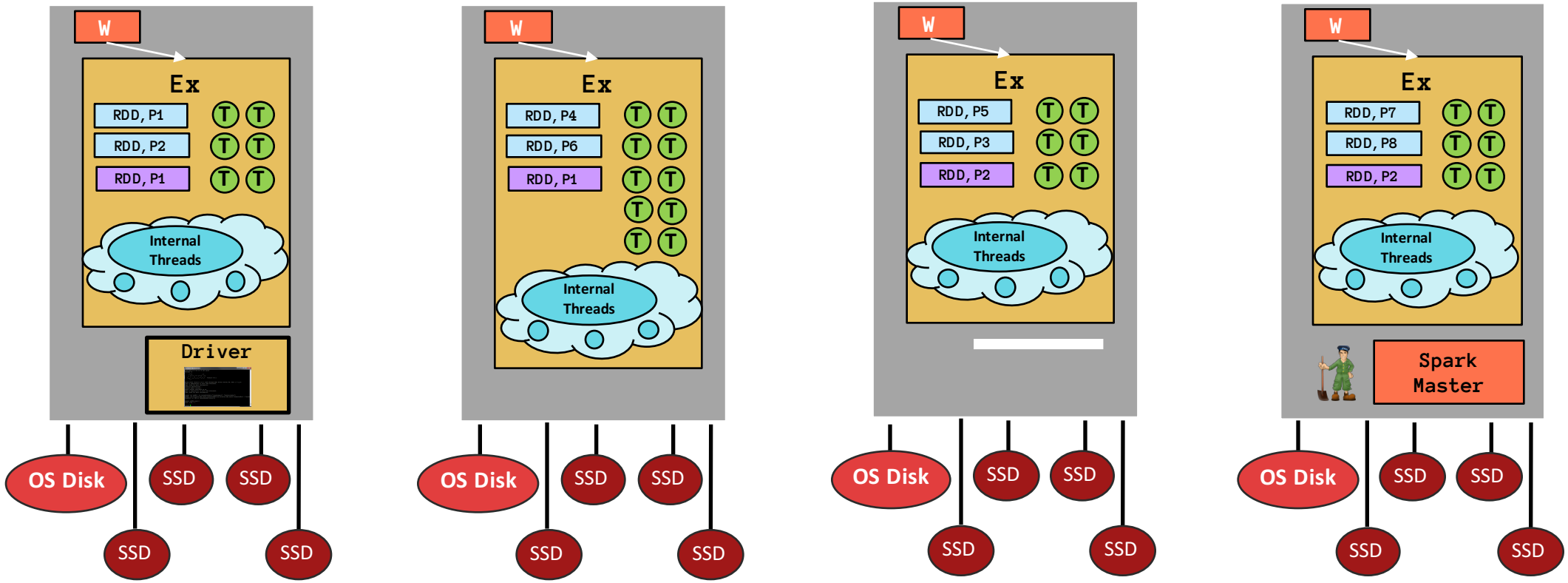
```
val conf = new SparkConf()
             .setMaster("local[12]")
             .setAppName("MyFirstApp")
             .set("spark.executor.memory", "3g")

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
```

vs.

spark-env.sh

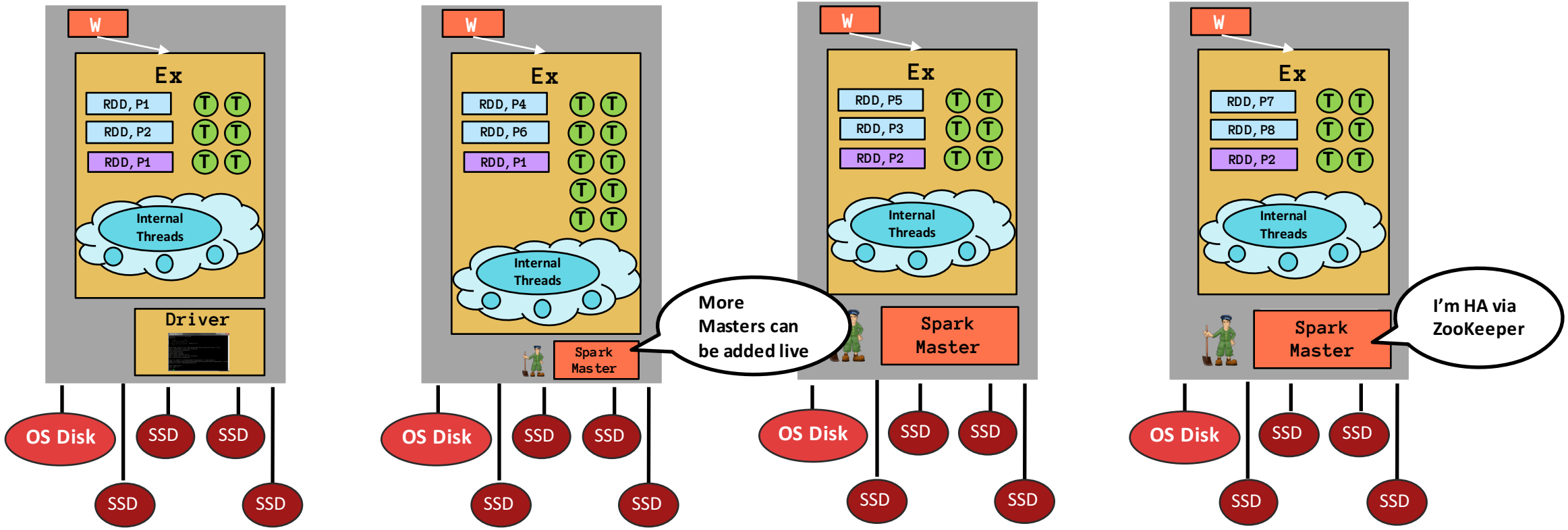
SPARK_LOCAL_DIRS

databricks

# Spark Standalone

different spark-env.sh

SPARK_WORKER_CORES




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

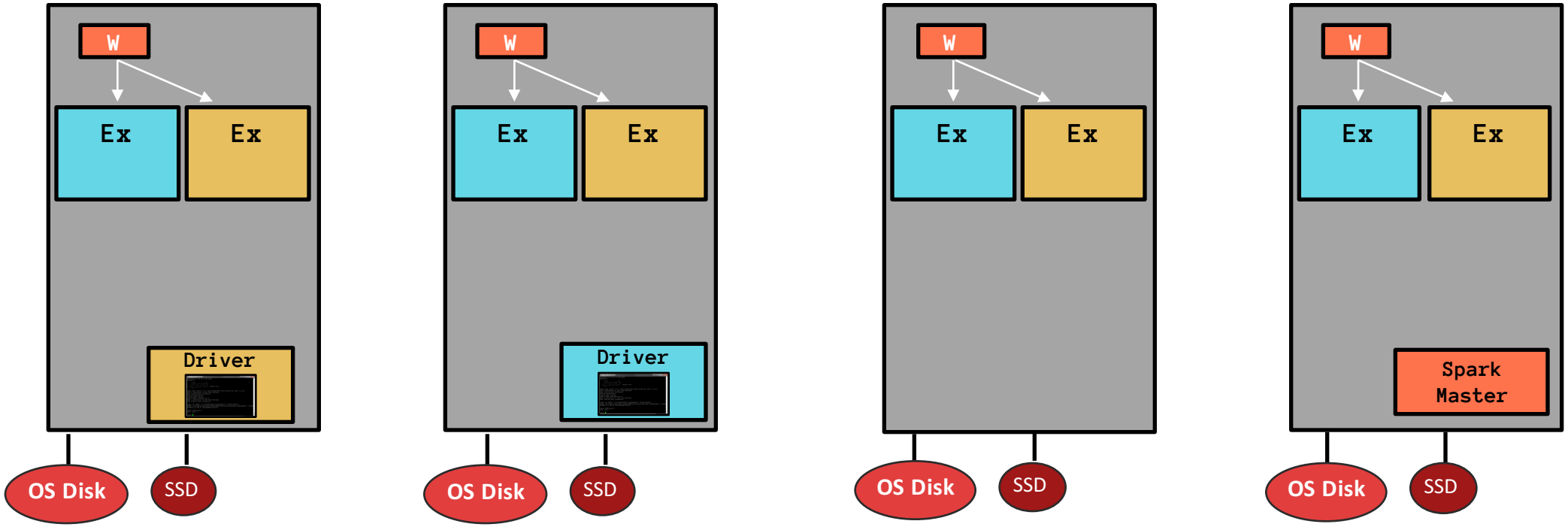
vs.

spark-env.sh

SPARK_LOCAL_DIRS

databricks

# Spark Standalone (multiple apps)



# 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

# YARN Mode

# YARN Benefits

Allows for multiple data processing engines against HDFS or HBase

Dynamically allocates cluster resources and improves utilization over static MR-v

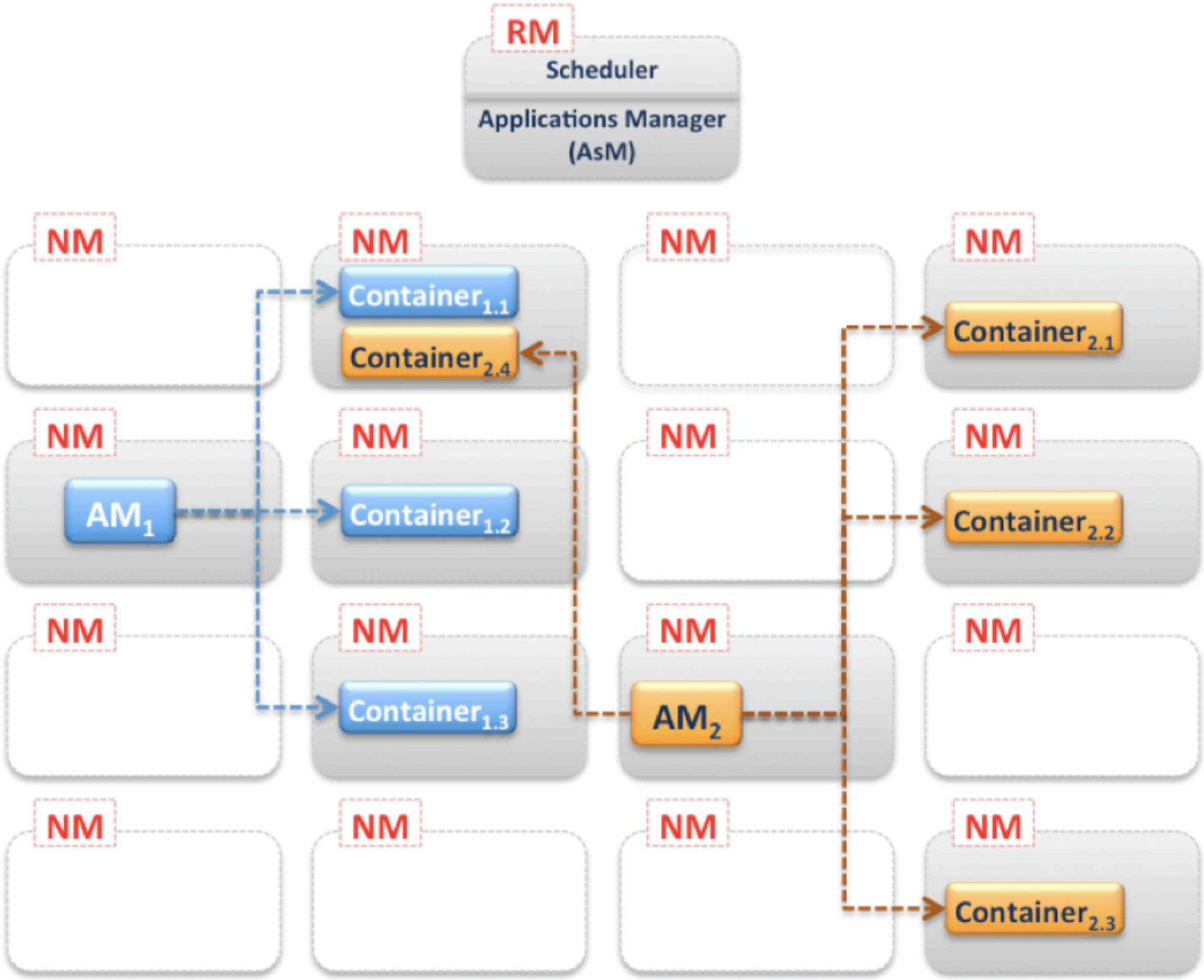
Better scalability in the future (>5,000 node clusters)

Can run multiple Executors on each node for a single Spark App w/ just one NodeManager

Solid Kerberos integration

Allow applications to request specific nodes for scheduling tasks

# YARN



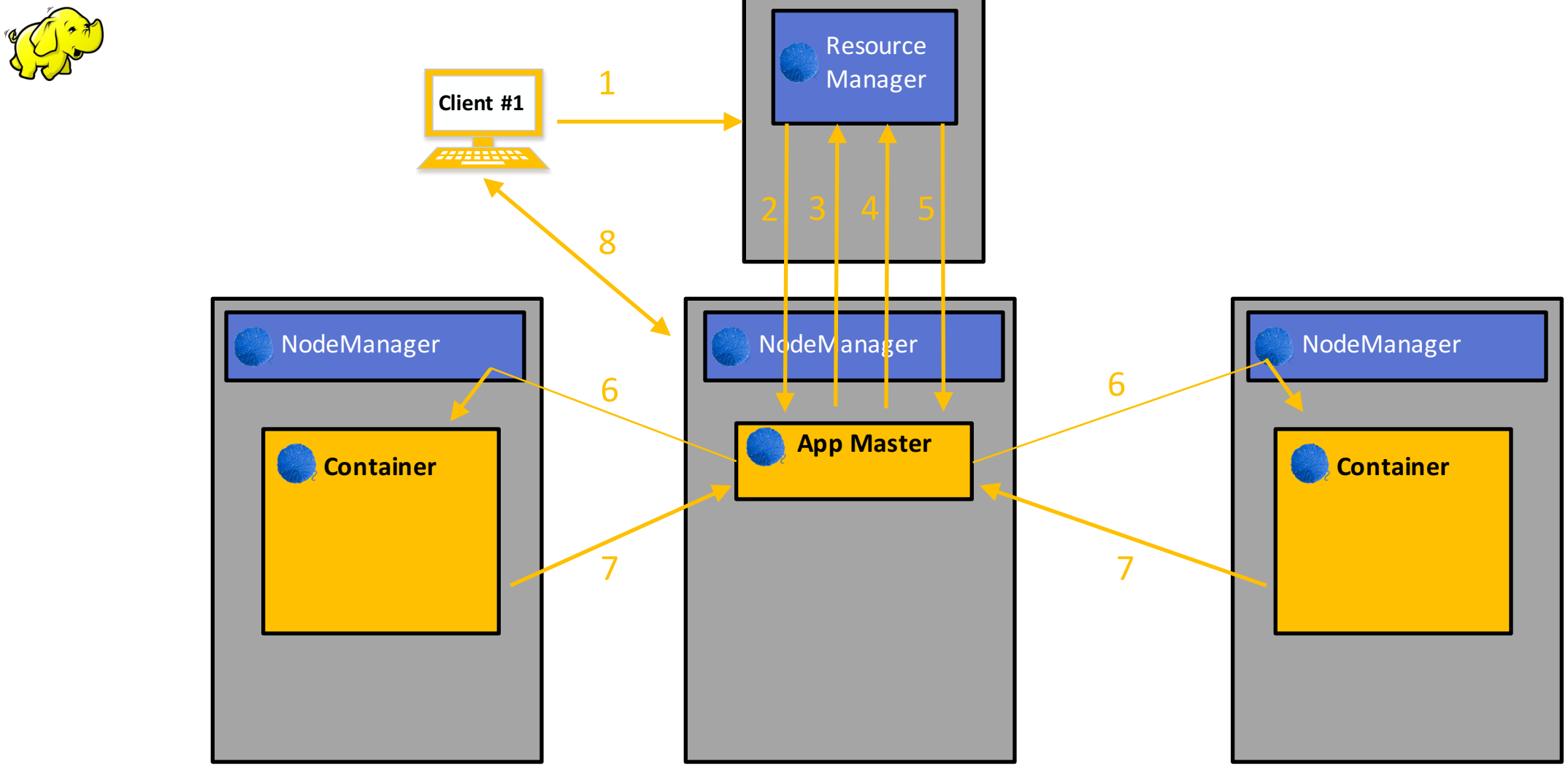

databricks

Source: Cloudera Blog

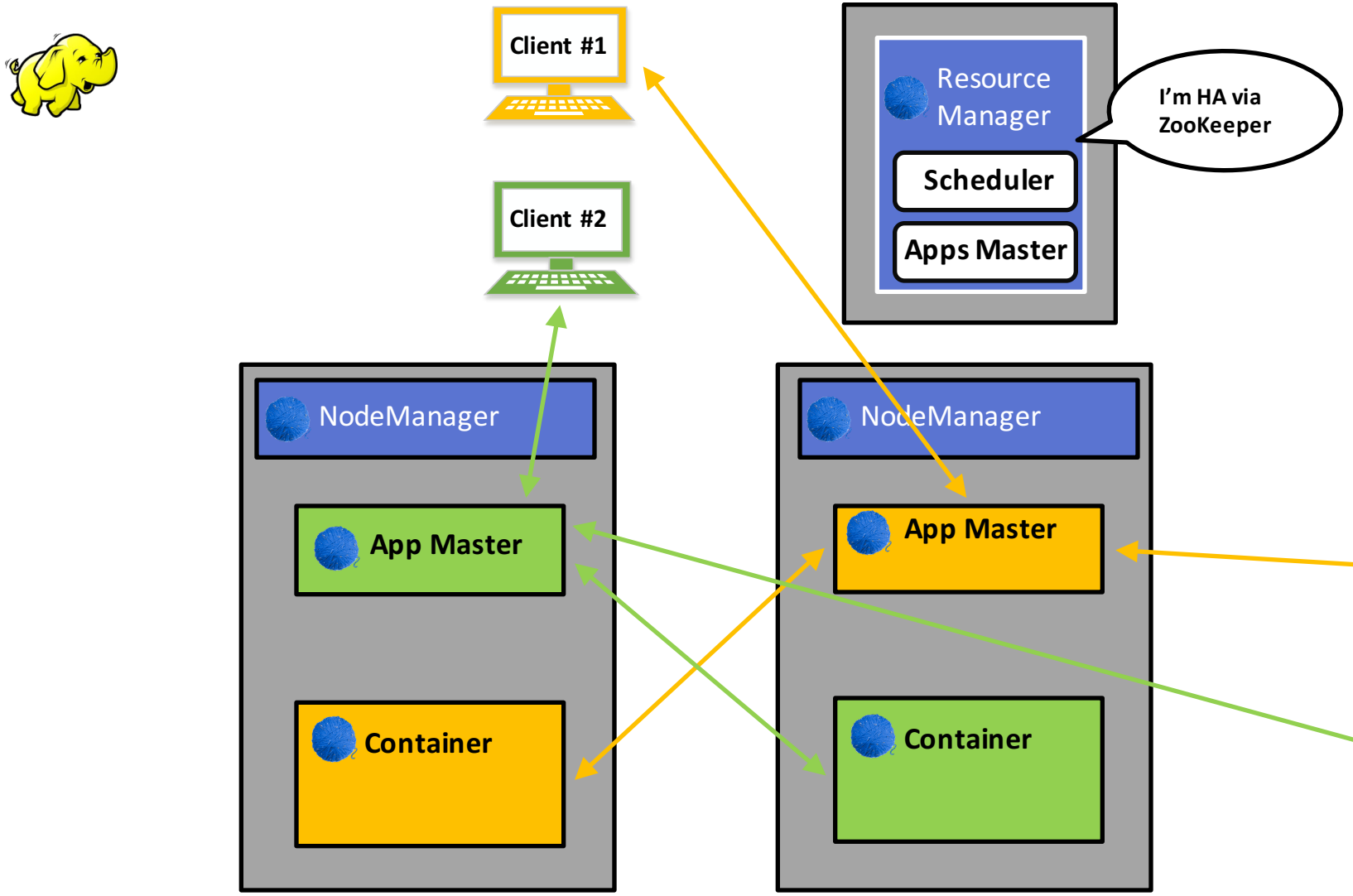
# YARN Settings

http://spark.apache.org/docs/latest/running-on-yarn.html

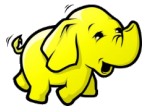
# Spark YARN

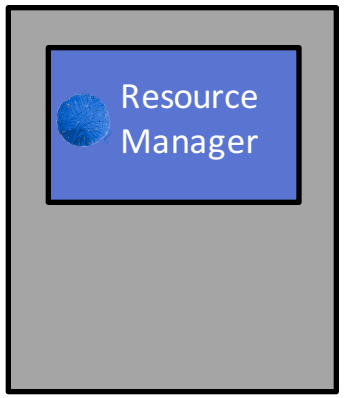

Client #1
1
Resource Manager
2
3
4
5
8
NodeManager
Container
6
NodeManager
App Master
6
NodeManager
Container
7
7

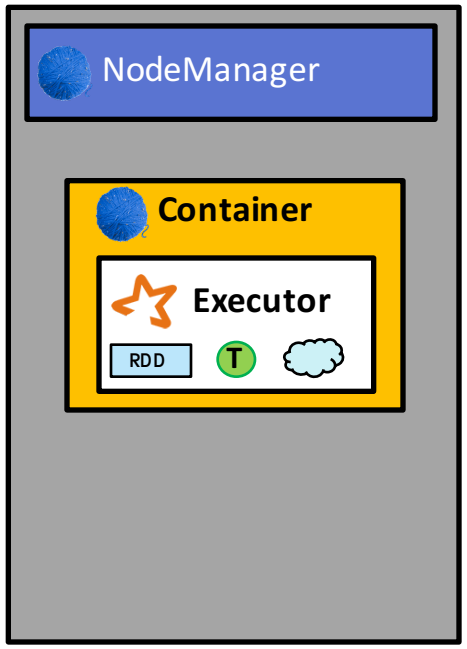
# Spark YARN

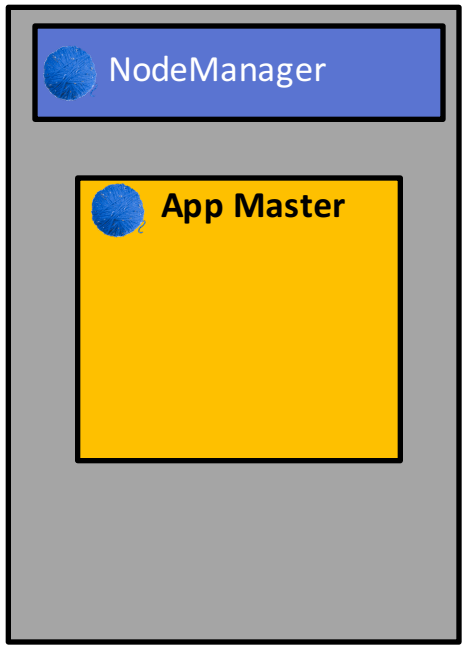

Client #1
Client #2
Resource Manager
I'm HA via ZooKeeper
Scheduler
Apps Master
NodeManager
App Master
Container
NodeManager
App Master
Container
NodeManager
Container
Container

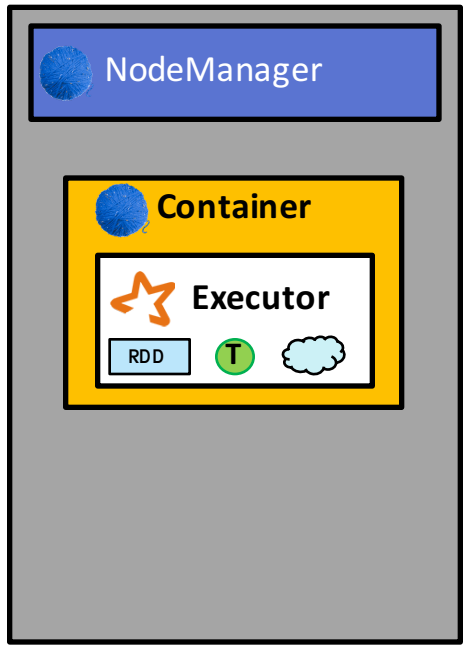
# Spark YARN (client mode)



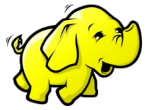



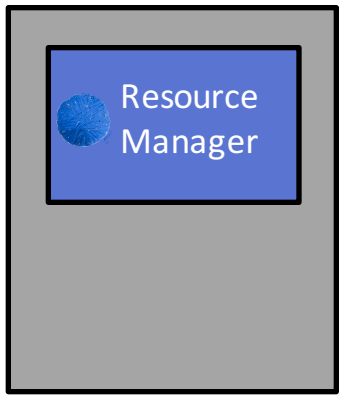



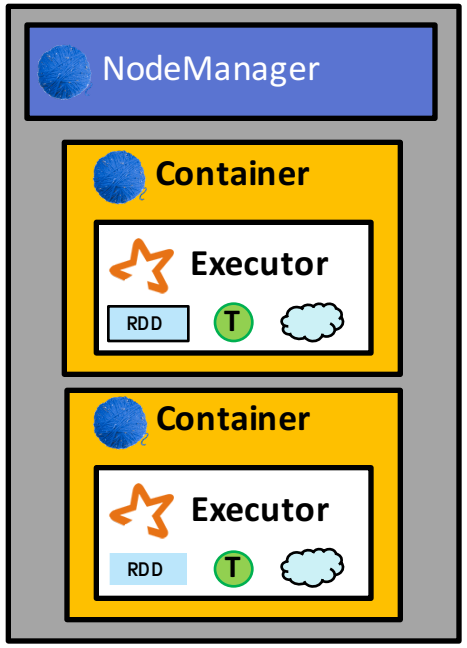



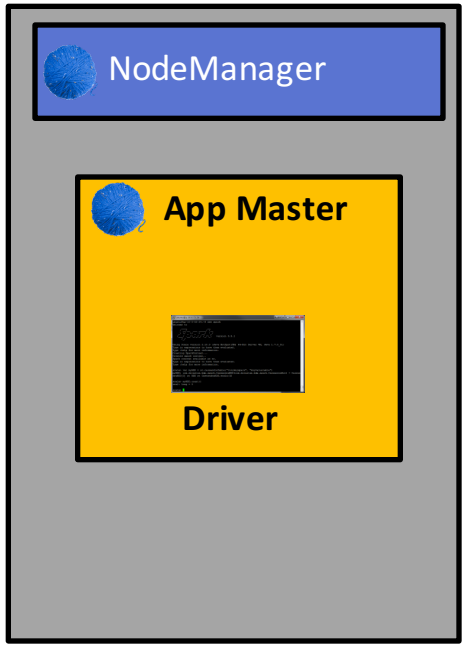
# Spark YARN (cluster mode)

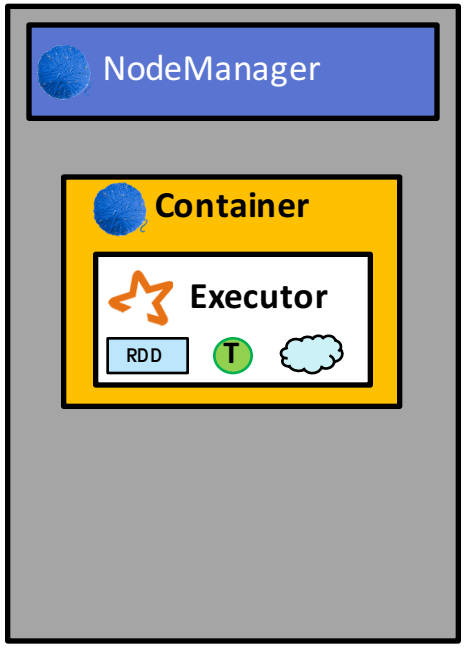



**Does not support Spark Shells**



# 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)
```

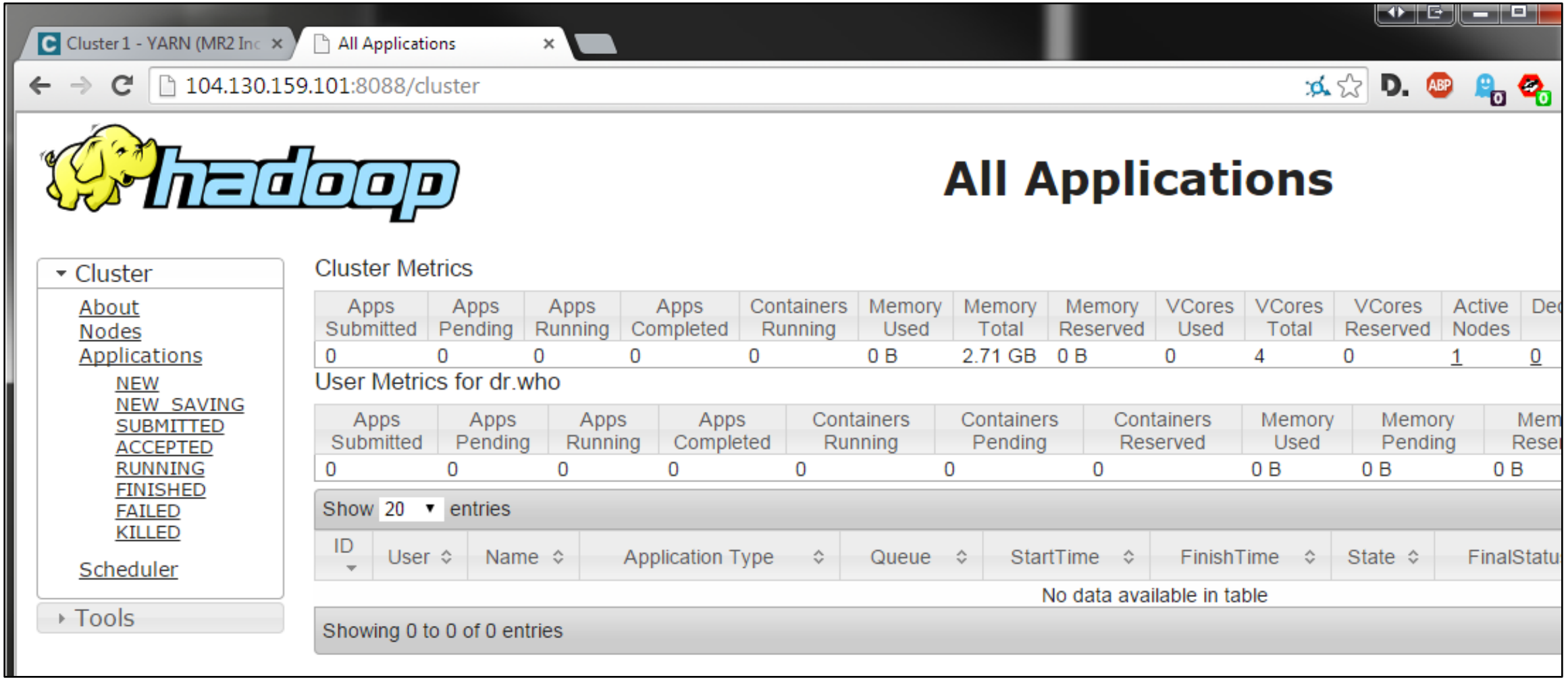


https://github.com/apache/spark/blob/master/core/src/main/scala/org/apache/spark/ExecutorAllocationManager.scala

# YARN Resource Manager UI

No apps running

http://<ip address>:8088



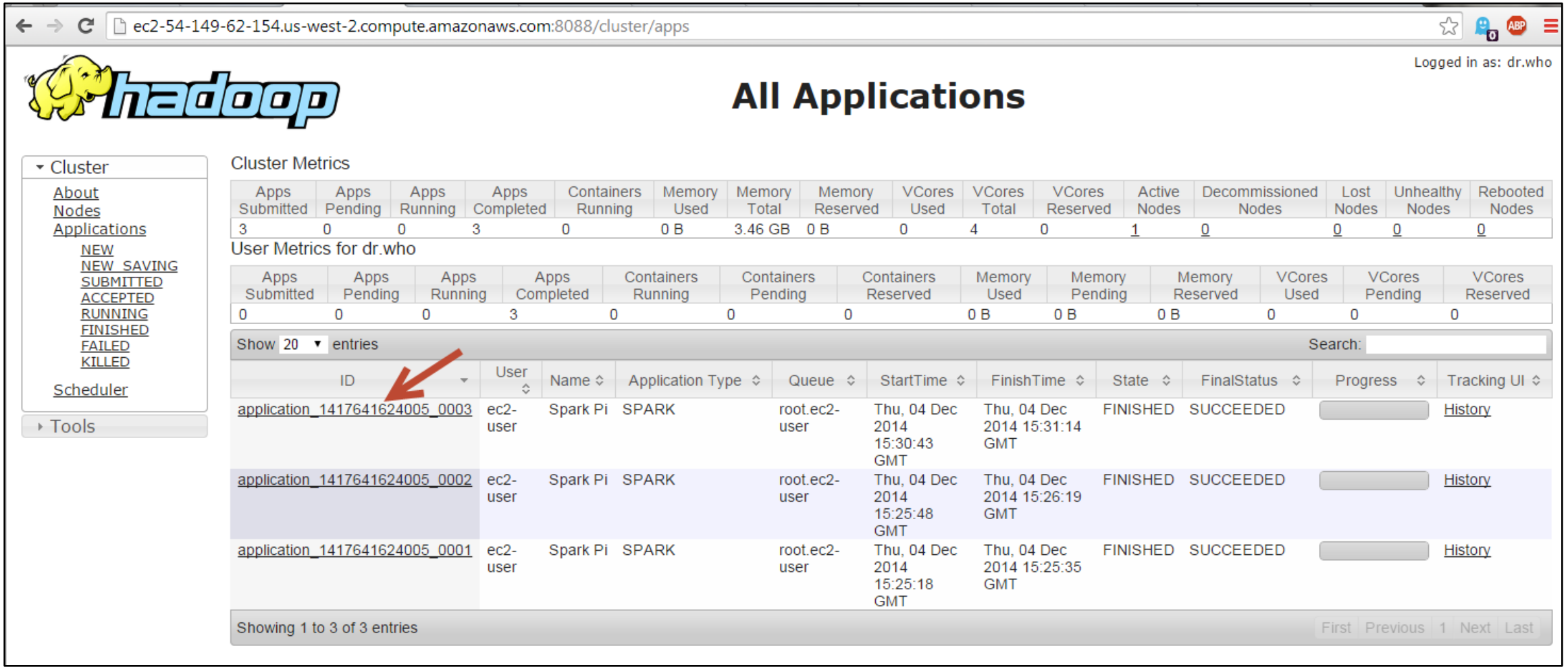

databricks

# Starting in “client” mode

```
[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
```

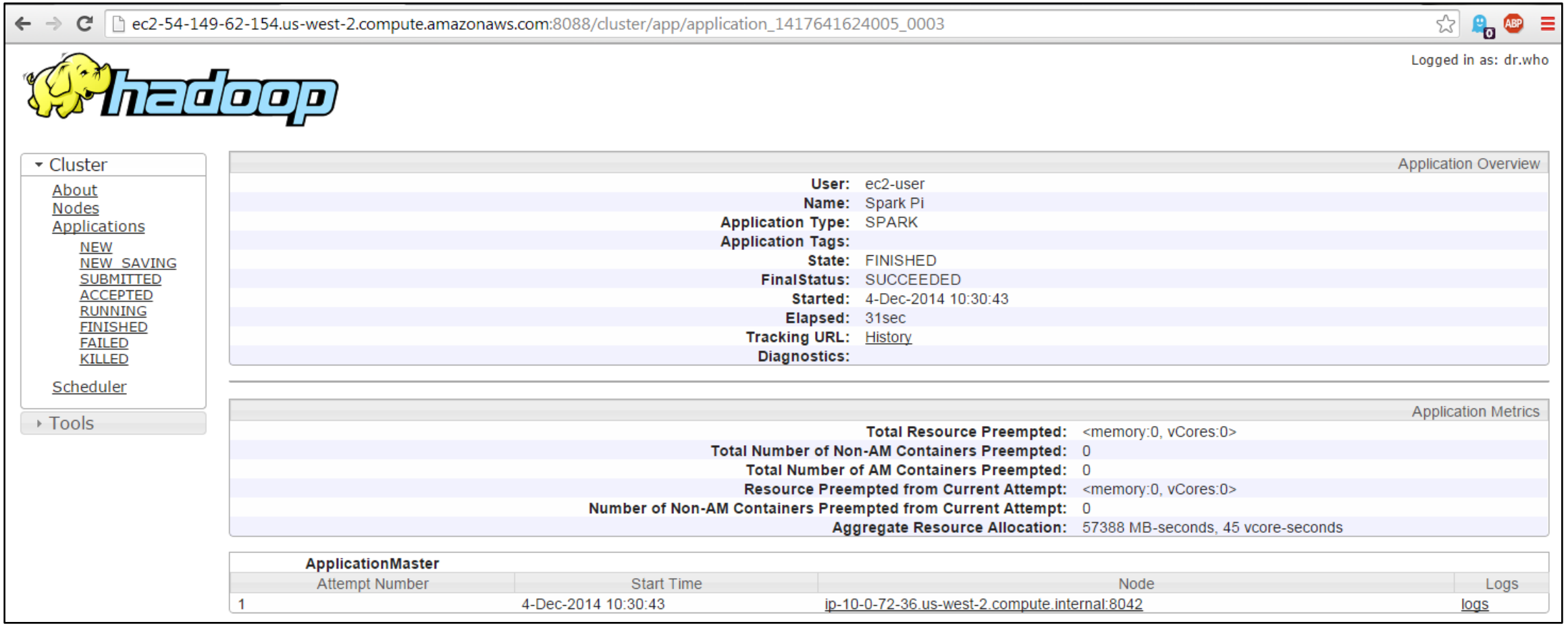
# YARN Resource Manager UI

App running in client mode



# YARN Resource Manager UI

App running in client mode

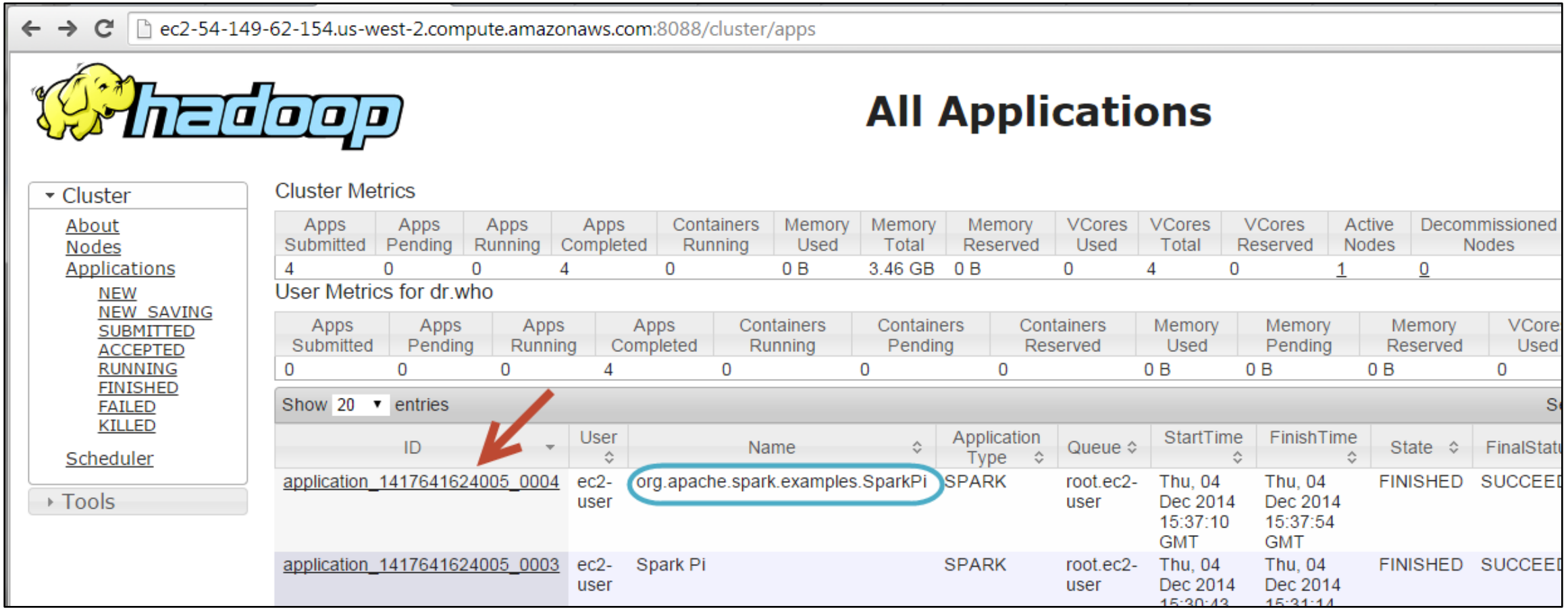


# Starting in “cluster” mode

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

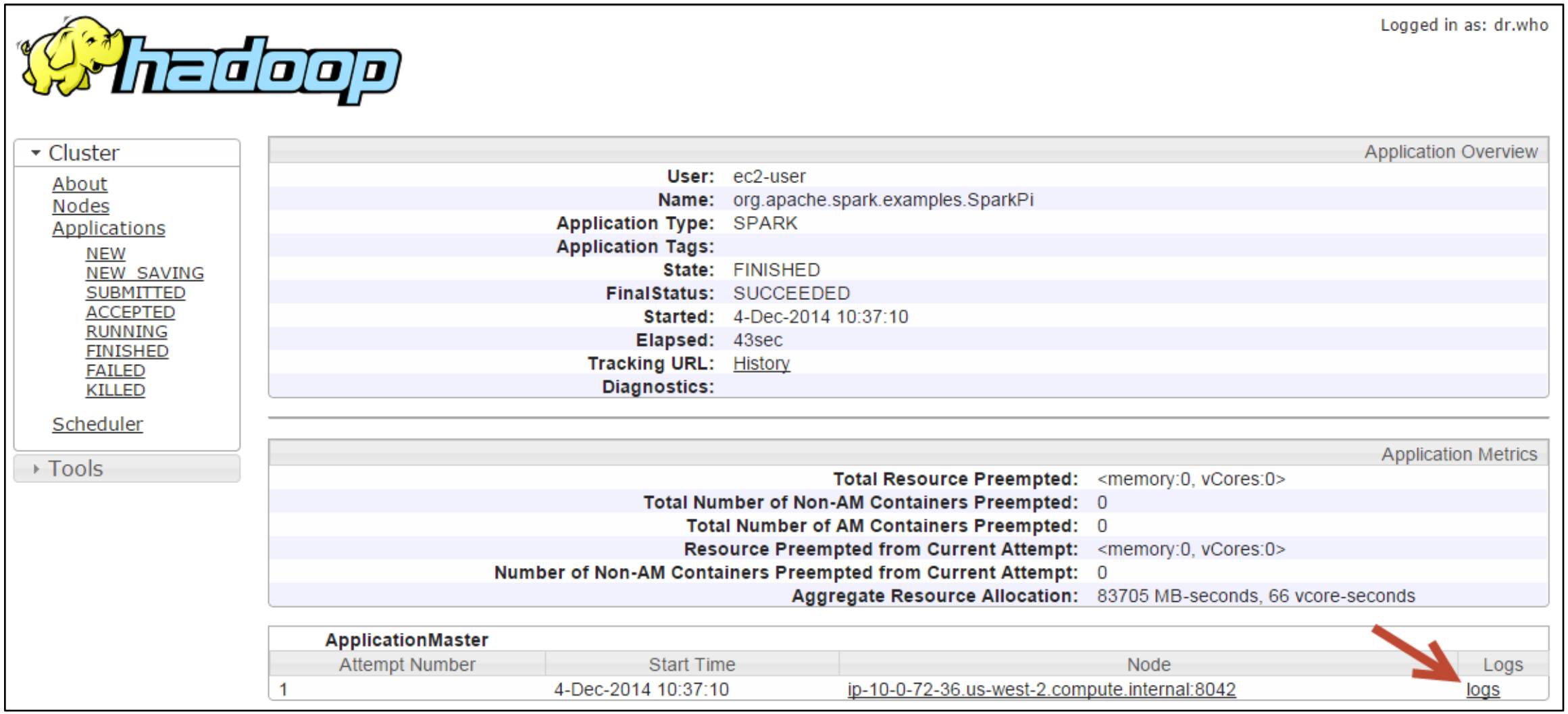
# YARN Resource Manager UI

App running in cluster mode



# YARN Resource Manager UI

App running in cluster mode



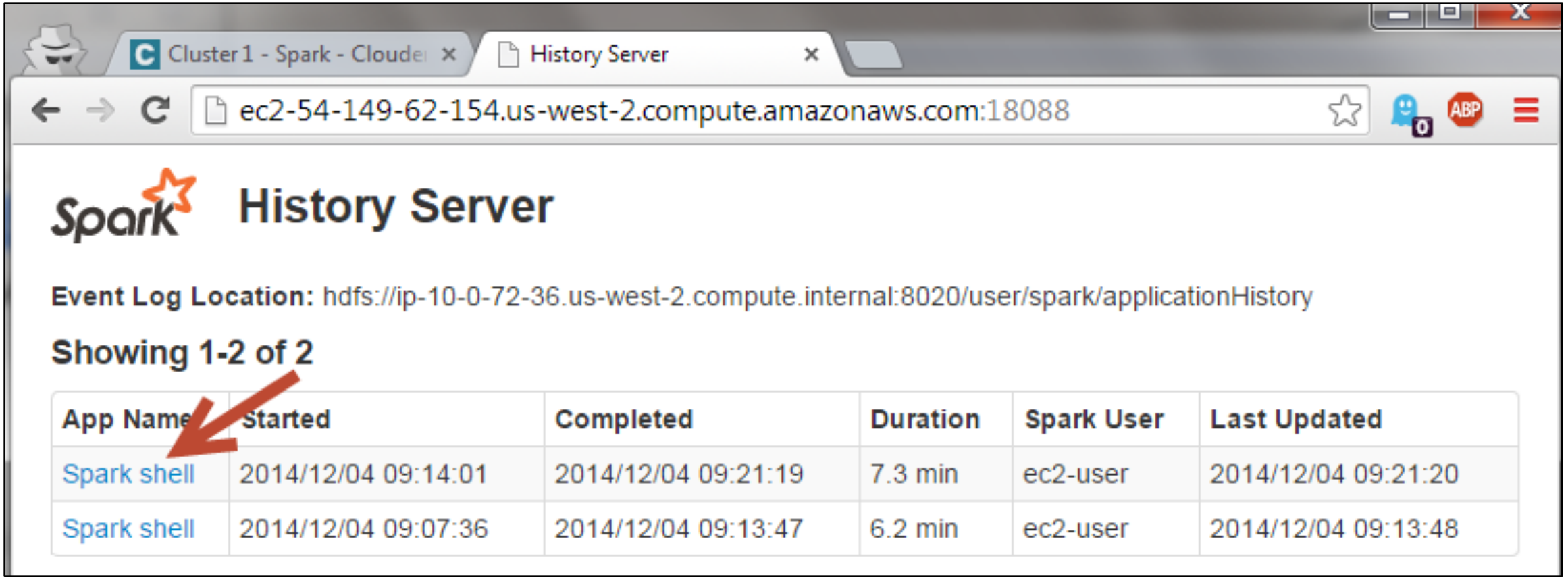

databricks

# YARN Resource Manager UI

App running in cluster mode


databricks

# History Server



# 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

# Summary

To summarize the concepts in this section, let's walk through the exact steps that occur when you run a Spark application on a cluster.

1. The user submits an application using **spark-submit**.

2. **spark-submit** launches the driver program and invokes the main method specified by the user.

3. The driver program contacts the cluster manager to ask for resources to launch executors.

4. The cluster manager launches executors on behalf of the driver program.

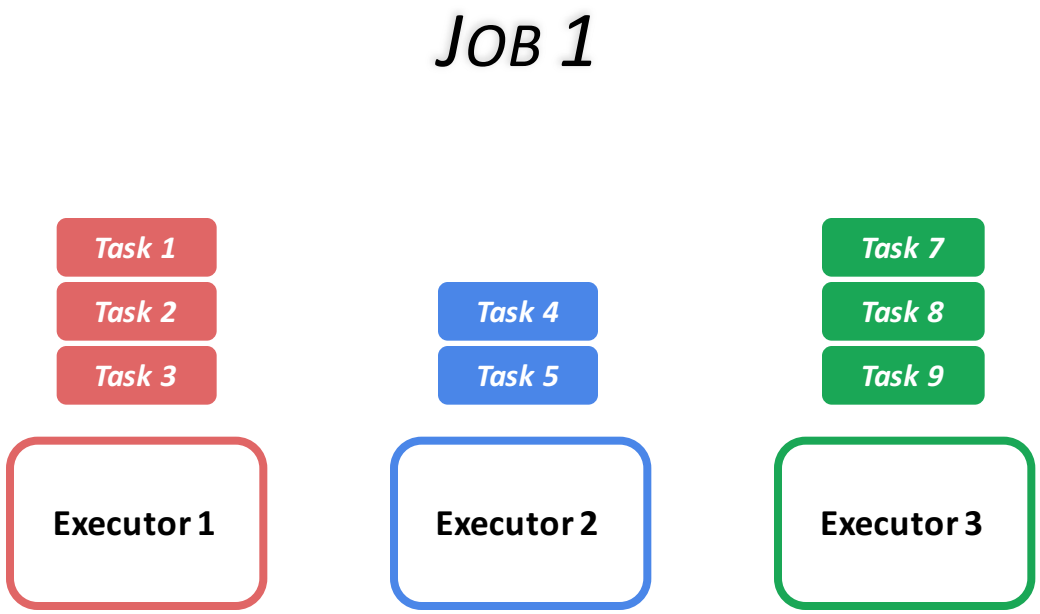
5. The driver process runs through the user application. Based on the RDD actions and transformations in the program, it sends work to executors in the form of tasks.

6. Tasks are run on executor processes to compute and save results.



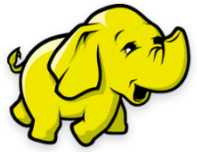
Source: Learning Spark

# Execution Model In




In Spark, each executor is long-running and runs many small tasks

# Execution Model is Different



## *HADOOP MAPREDUCE*


In MapReduce, each container is short-lived and runs one large task

# End of Spark Clusters

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

# Spark Streaming

databricks™

# Spark Streaming

- Scalable
- High-throughput
- Fault-tolerant

TCP socket

Kafka

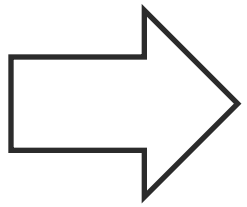
Flume

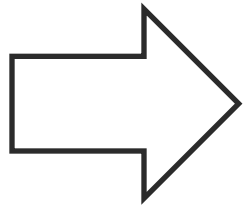
HDFS

S3

Kinesis

Twitter



HDFS / S3

Cassandra

HBase

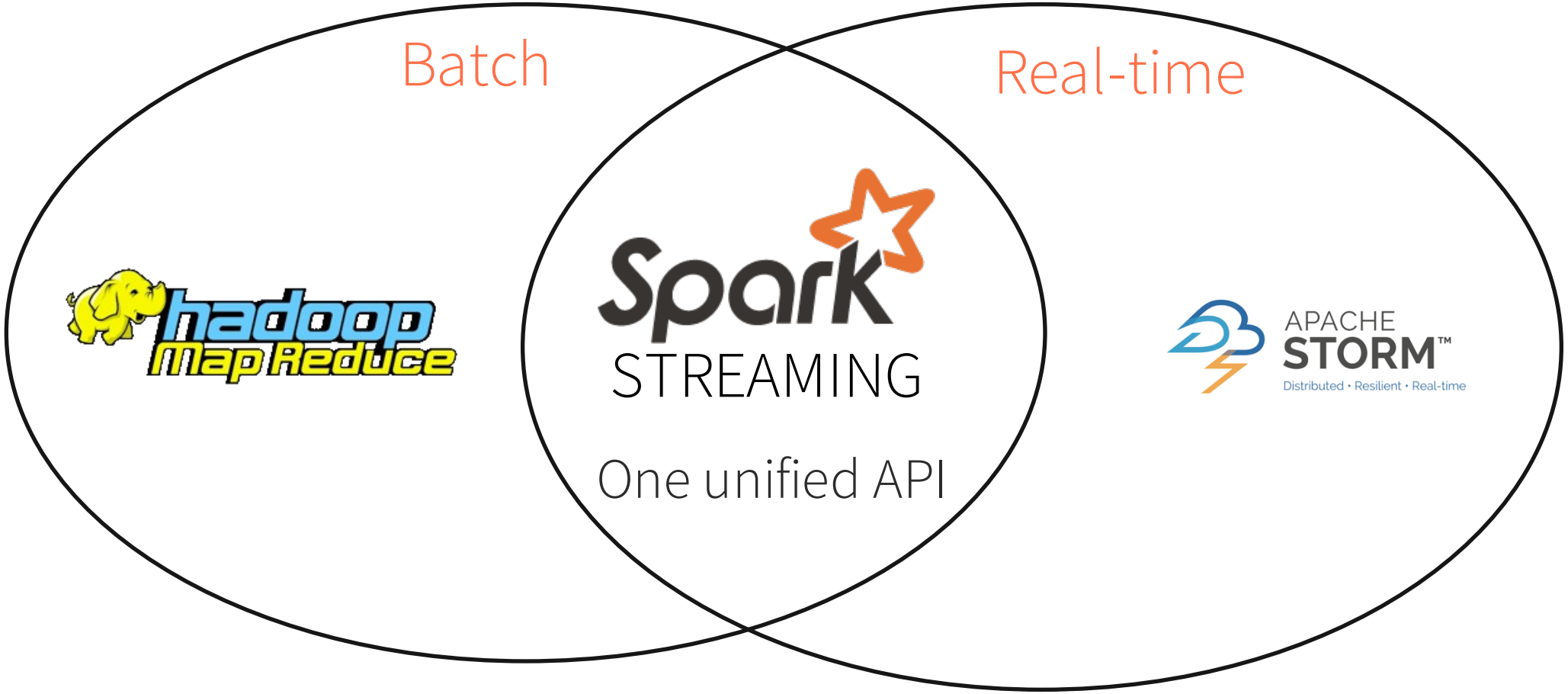
Dashboards

Databases

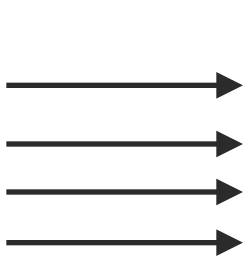
Complex algorithms can be expressed using:

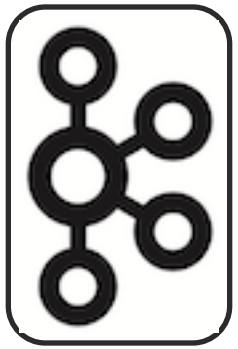
- Spark transformations: `map()`, `reduce()`, `join()`…
- MLlib + GraphX
- SQL

# Batch and Real-time



# Use Cases



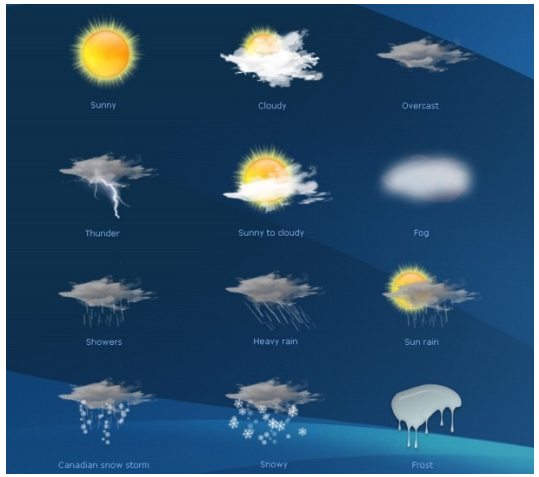

Page views

Kafka for buffering

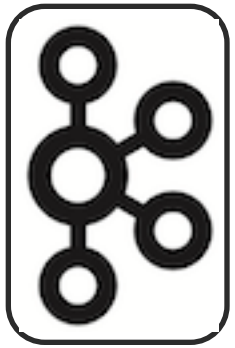
Spark for processing

# Use Cases

Smart meter readings




Live weather data

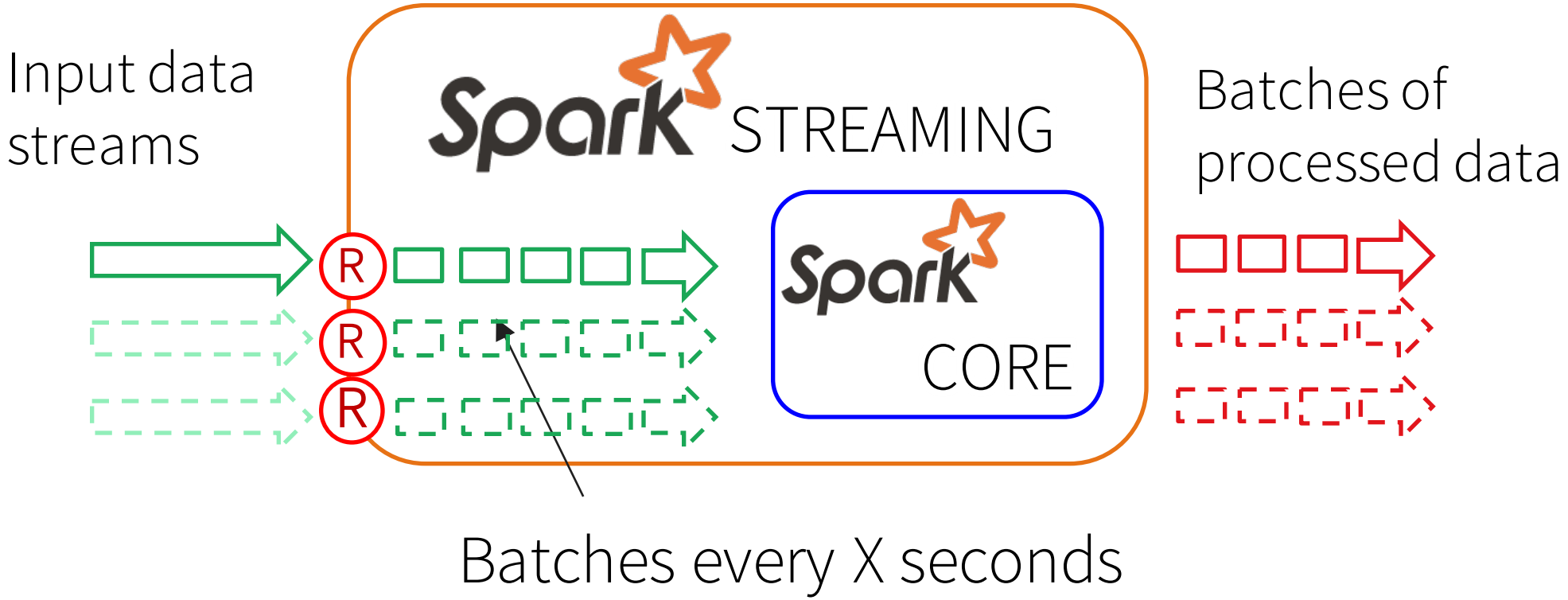



Join 2 live data sources

Spark STREAMING

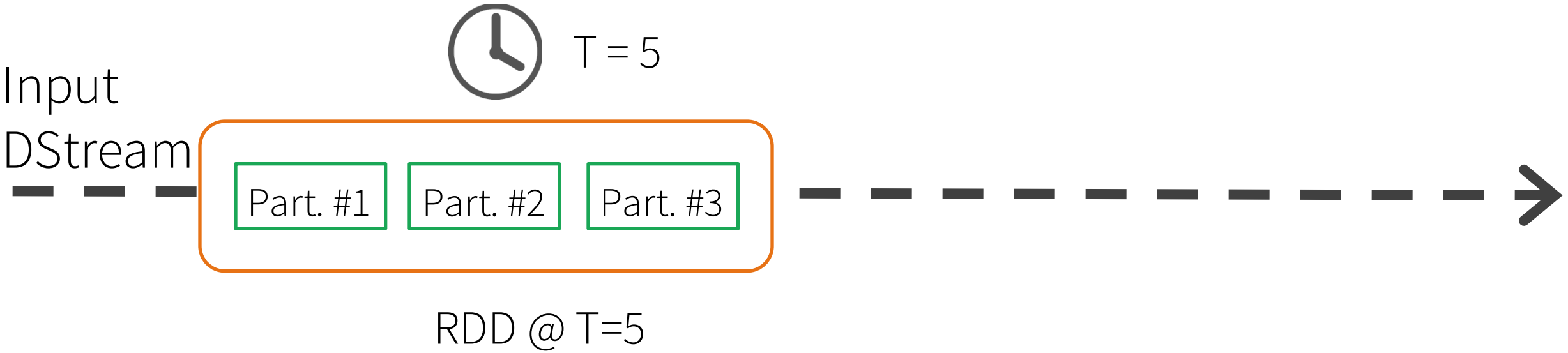
databricks

# Data Model


Input data streams
Spark STREAMING
Spark CORE
R
R
R
Batches of processed data
Batches every X seconds

# DStream (Discretized Stream)

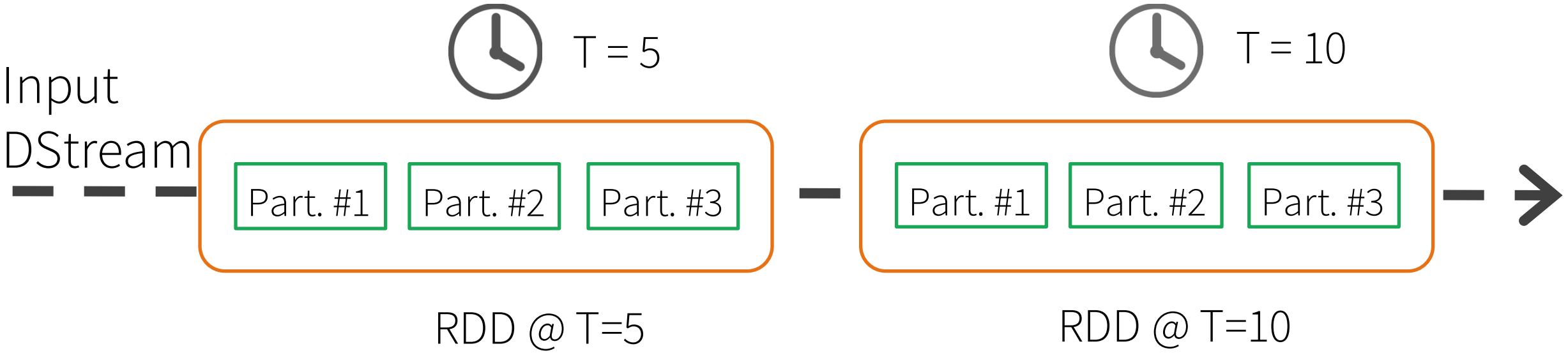
Batch interval = 5 seconds




One RDD is created every 5 seconds

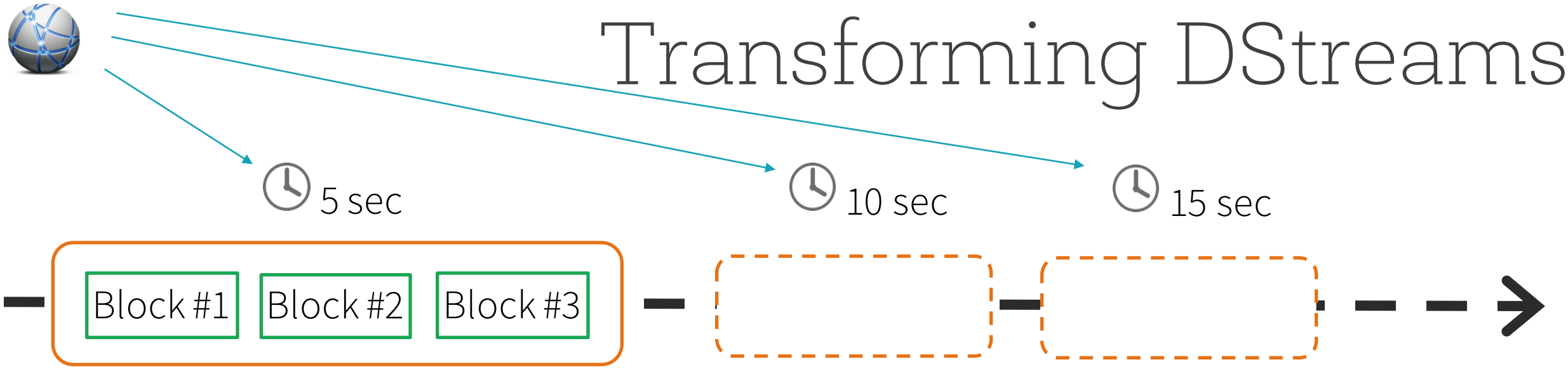
# DStream (Discretized Stream)

Batch interval = 5 seconds

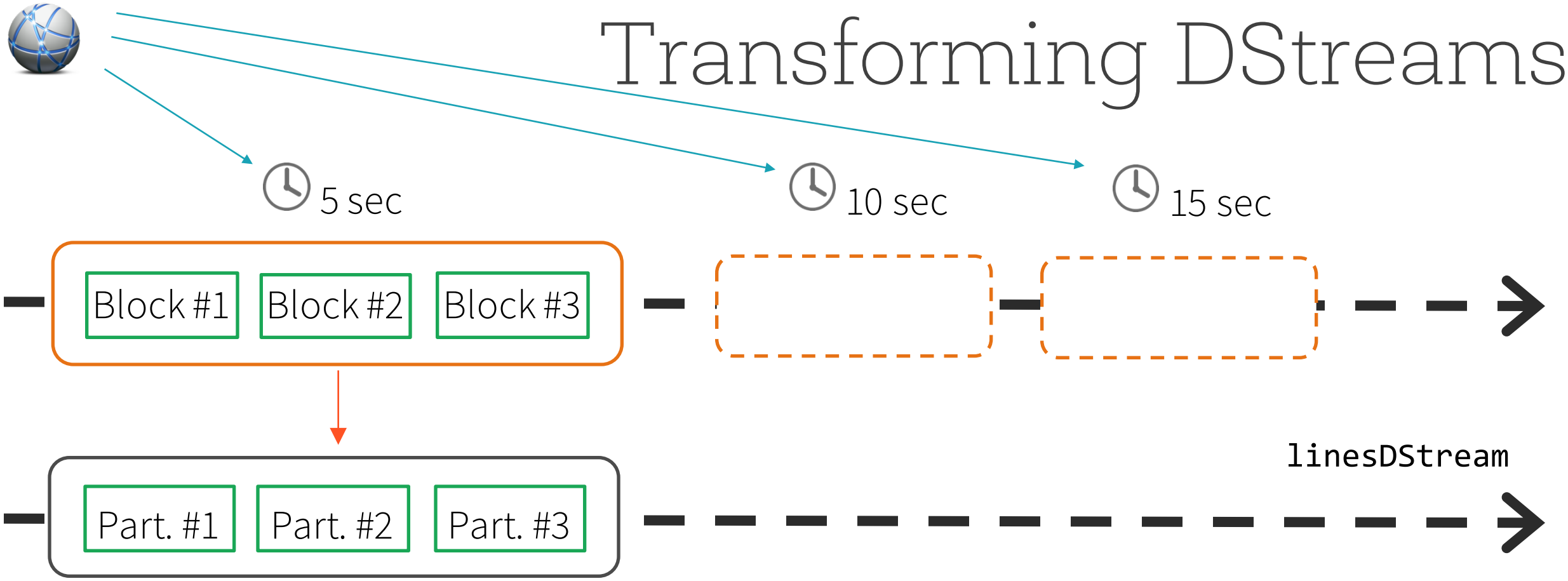



One RDD is created every 5 seconds

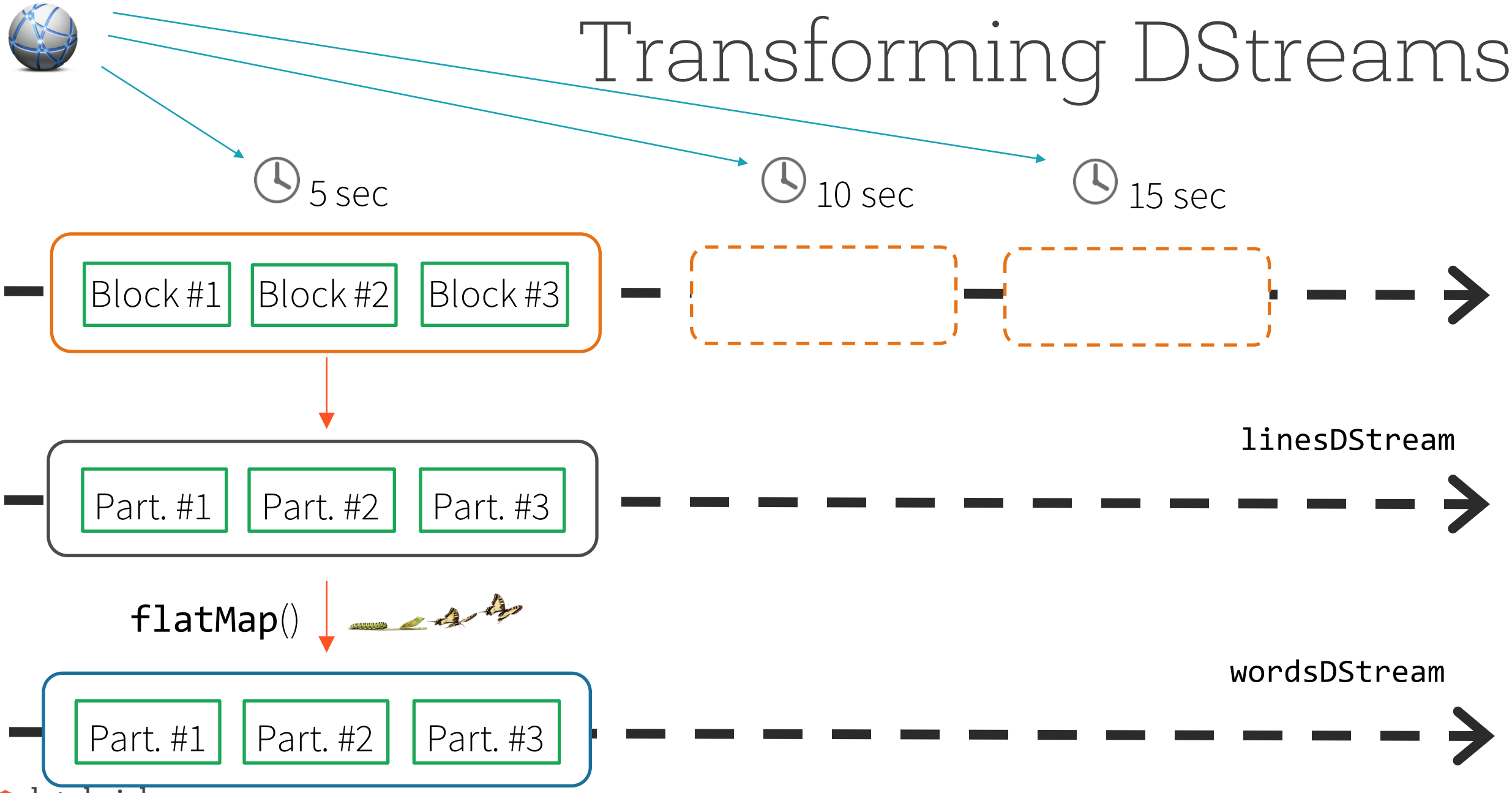
# Transforming DStreams



# Transforming DStreams

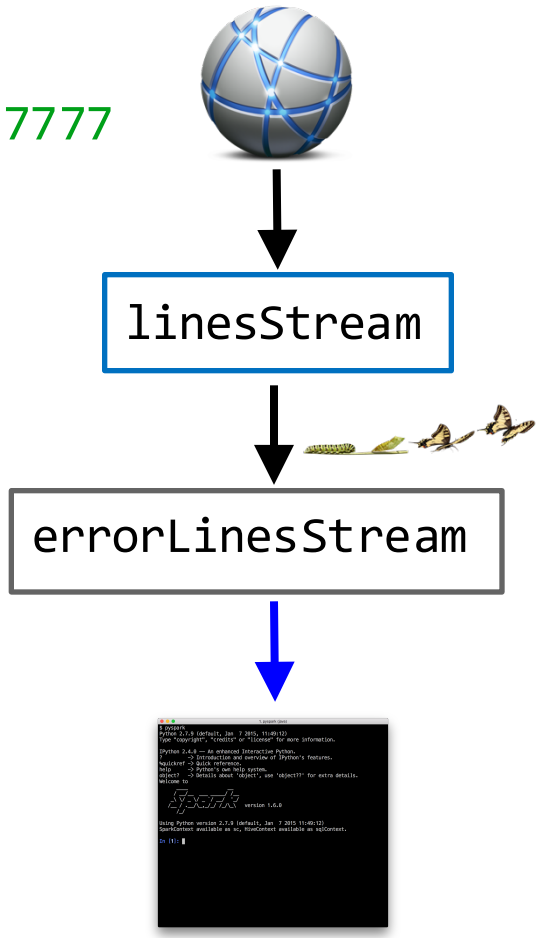


# Transforming DStreams



# Scala Example

```
import org.apache.spark.streaming._
val sc = SparkContext(...)
// Create a StreamingContext with a 1-second batch size from a SparkConf
val ssc = new StreamingContext(sc, Seconds(1))

// Create DStream using data received after connecting to localhost:7777
val linesStream = ssc.socketTextStream("localhost", 7777)

// Filter our DStream for lines with "error"
val errorLinesStream = linesStream.filter {_ contains "error"}

// Print out the lines with errors
errorLinesStream.pprint()

// Start our streaming context and wait for it to "finish"
ssc.start()
ssc.awaitTermination()
```




databricks

# Example


```
$ nc -l -p 7777

all is good
there was an error
good good

error 4 happened
all good now
```

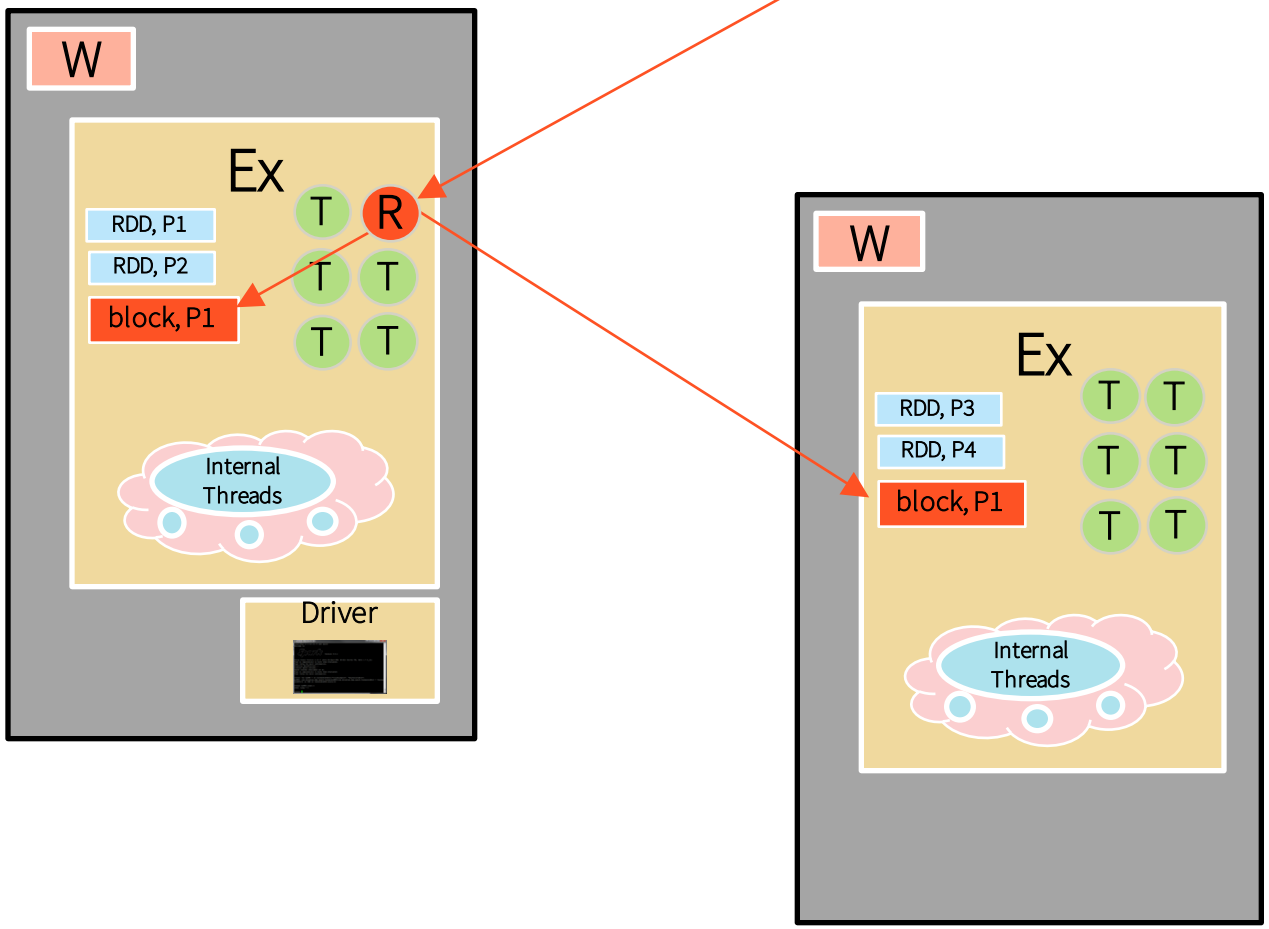

```
$   spark-submit --class com.examples.Scala.StreamingLog \
    $ASSEMBLY_JAR local[4]
. . .
-------------------------
Time: 2015-05-26 15:25:21
-------------------------
there was an error

-------------------------
Time: 2015-05-26 15:25:22
-------------------------
error 4 happened
```

databricks

# Example

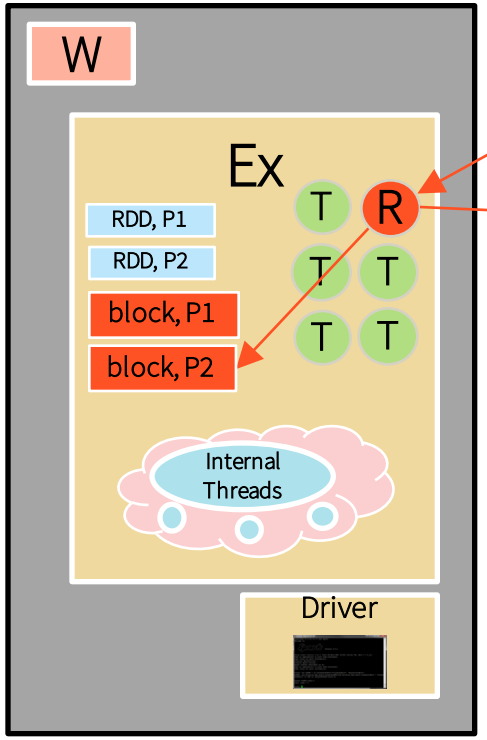
Batch interval = 600 ms

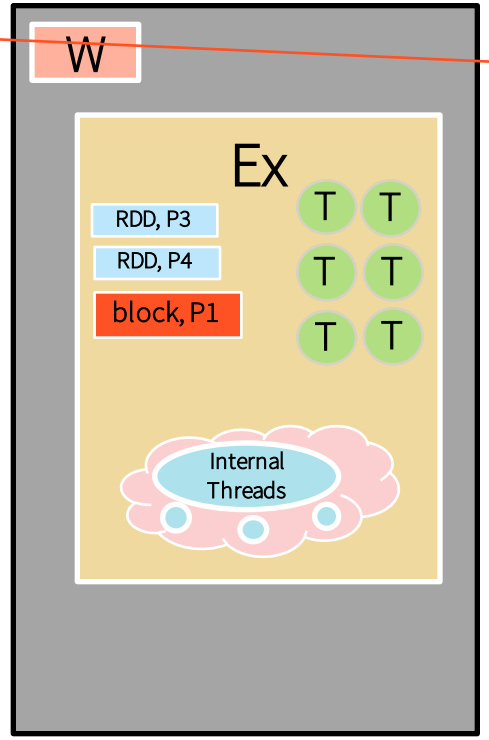
localhost 7777

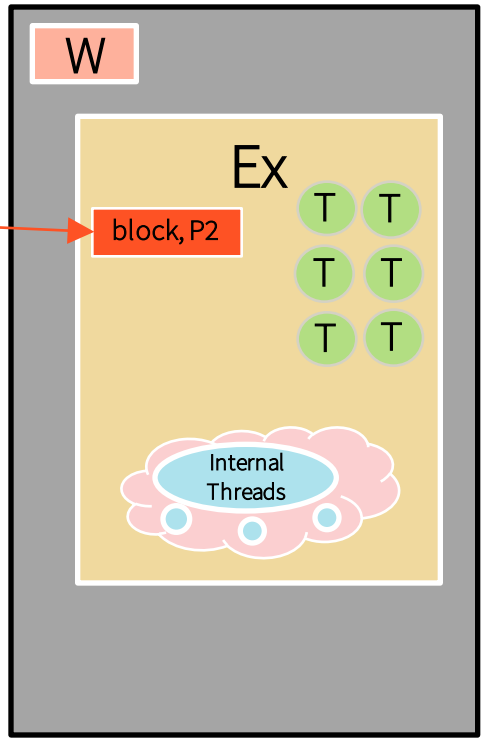


Batch interval = 600 ms

# Example



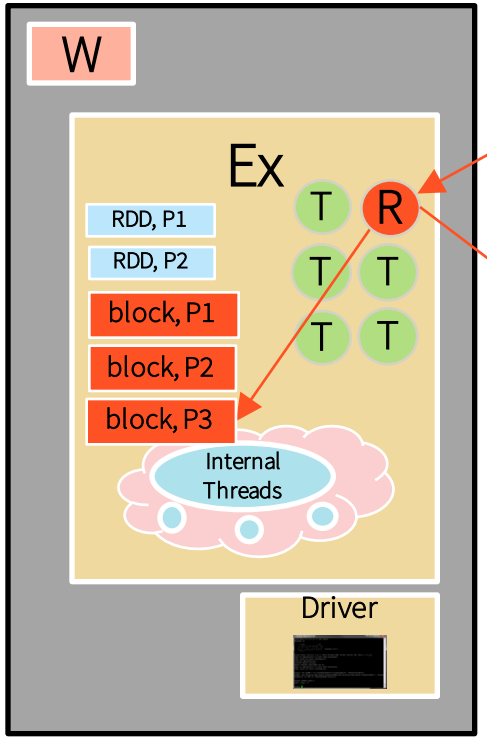



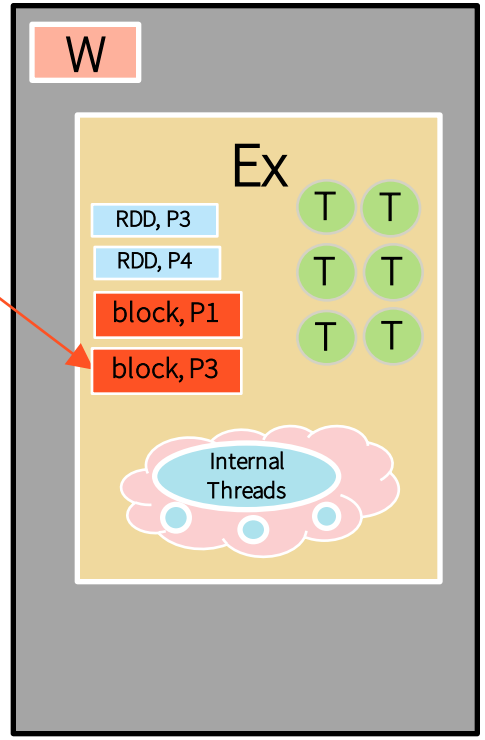

200 ms later

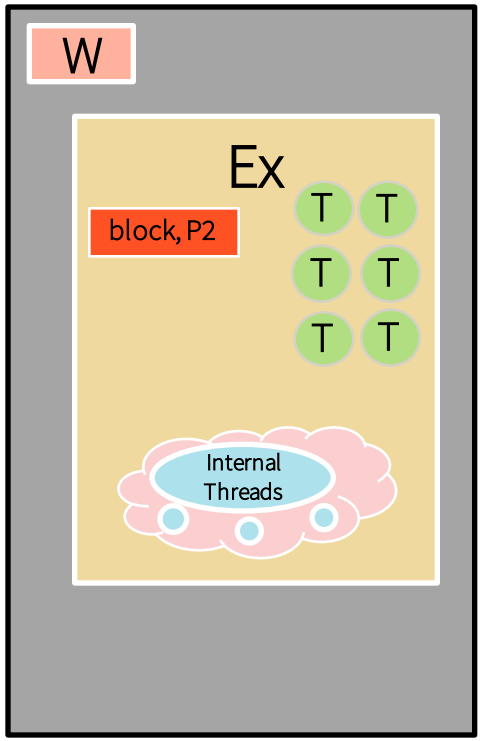
databricks

Batch interval = 600 ms

# Example



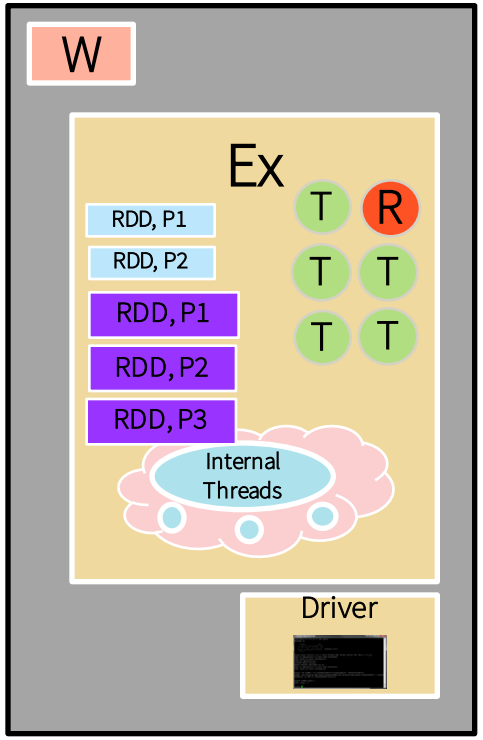



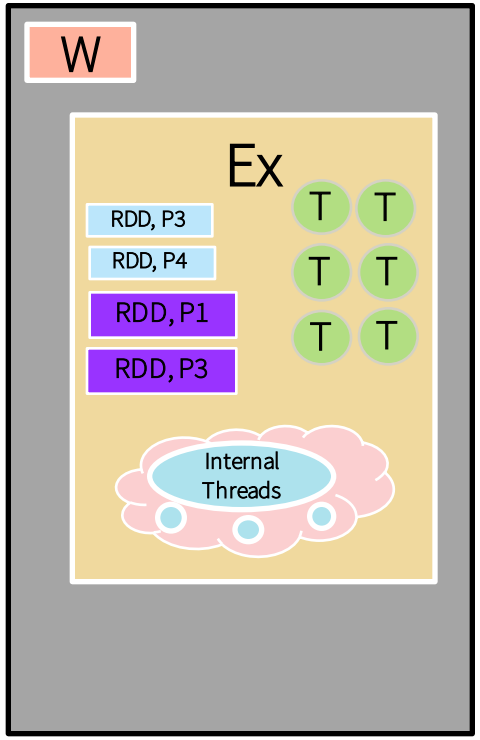



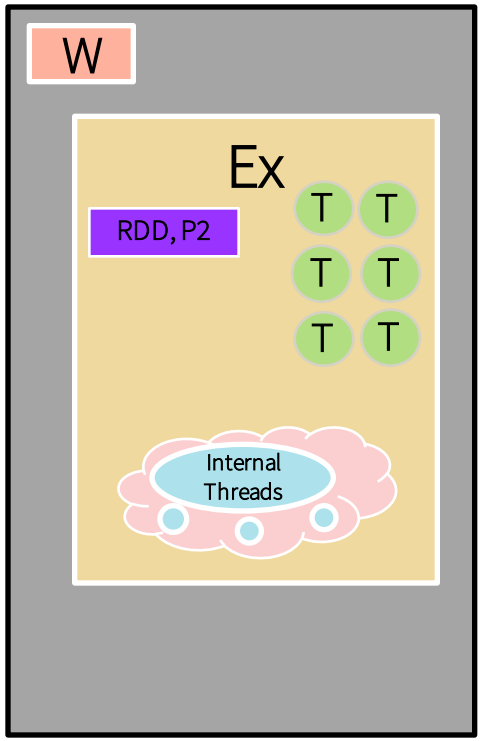

200 ms later

Batch interval = 600 ms

# Example

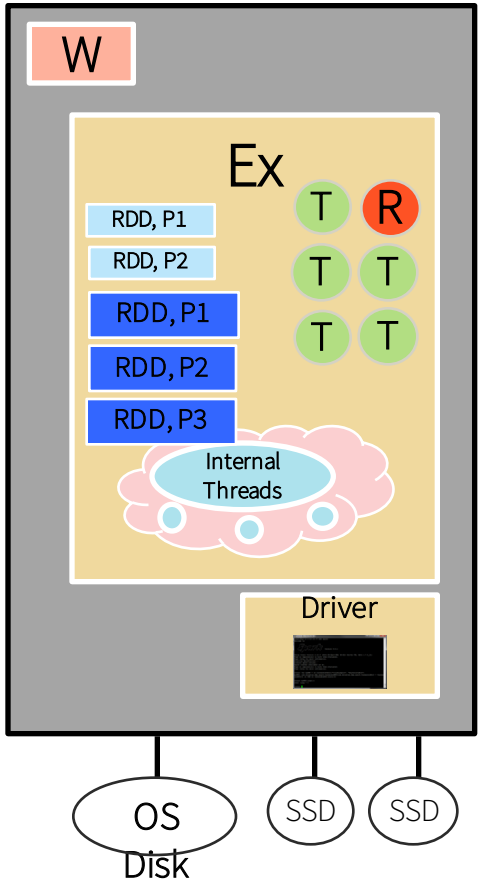


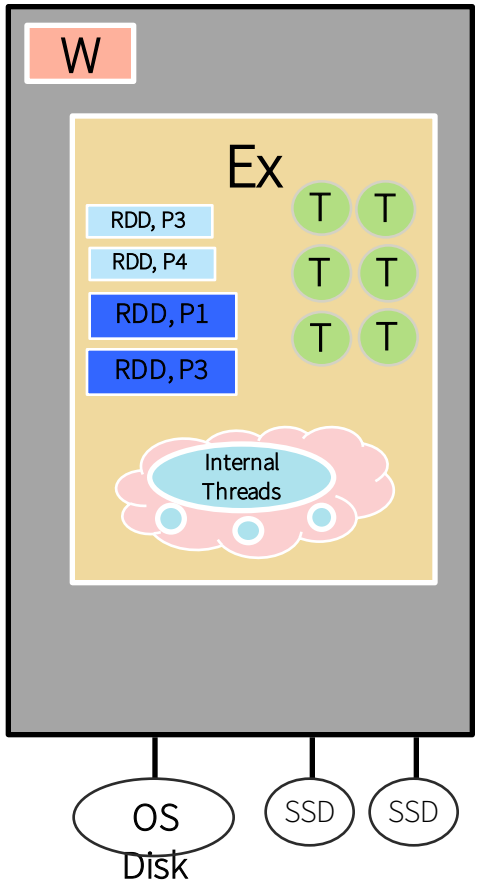



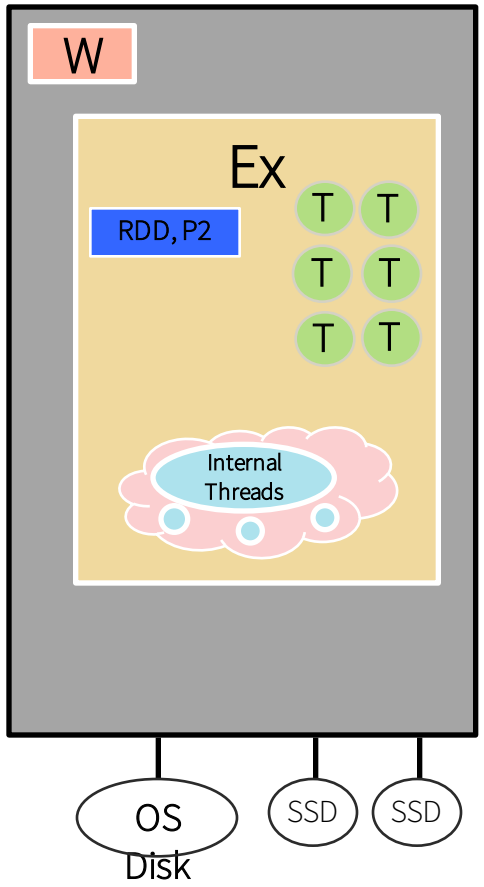



Batch interval = 600 ms

# Example



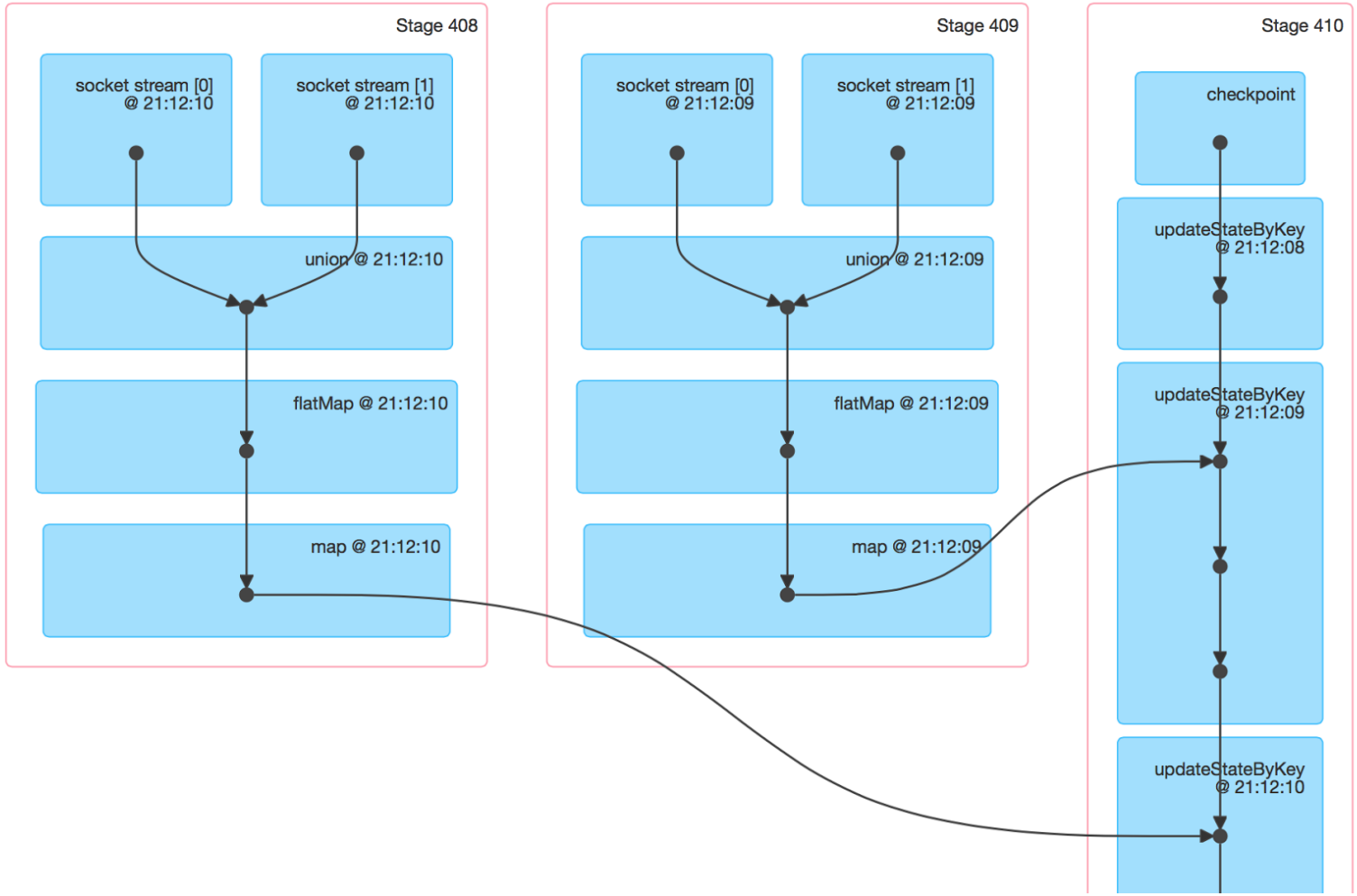

databricks

# Streaming Viz

## Details for Job 66

**Status:** SUCCEEDED
**Completed Stages:** 1
**Skipped Stages:** 2

▸ Event Timeline
▾ DAG Visualization




databricks

# Transformations on DStreams

map( λ )

reduce( λ )

union(otherStream)

updateStateByKey( λ )

flatMap( λ )

filter( λ )

join(otherStream,[numTasks])

cogroup(otherStream,[numTasks])

repartition(numPartitions)

RDD

transform( λ )

RDD

count()

reduceByKey( λ ,[numTasks])

countByValue()

# More hands-on with Streaming

In the Databricks shard, look in

Labs > streaming

Inside: a **`Wikipedia-Anonymous-Edits`** folder, with an **`Anonymous-Edits`** Scala notebook and a **`Solutions`** notebook beneath it. (Scala only)

- Open the **`Anonymous-Edits`** notebook
- Attach the notebook to your cluster
- Follow the instructions in the notebook

If you have problems with any of the exercises, as a TA, or consult the Solutions notebook in the cloned folder.

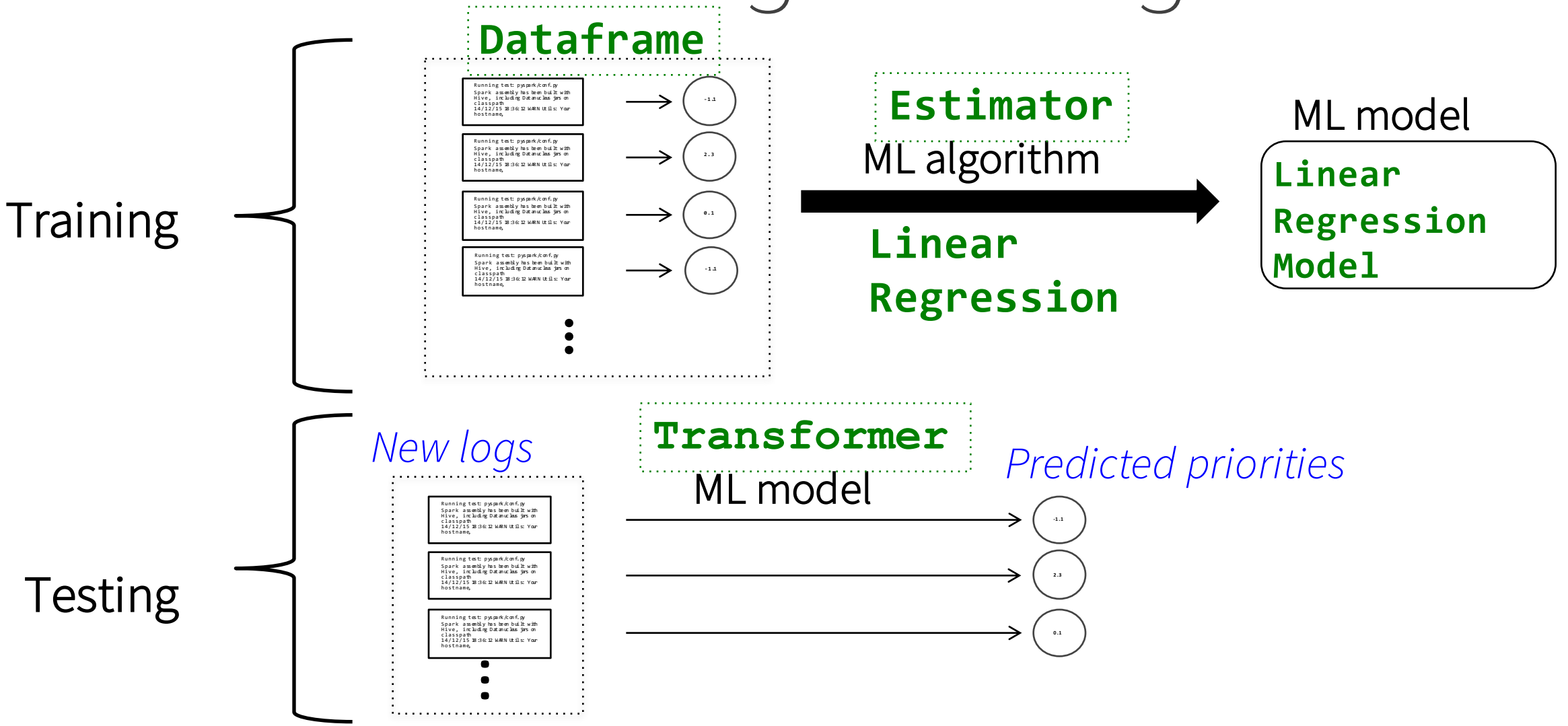
# End of Spark Streaming

# Spark Machine Learning

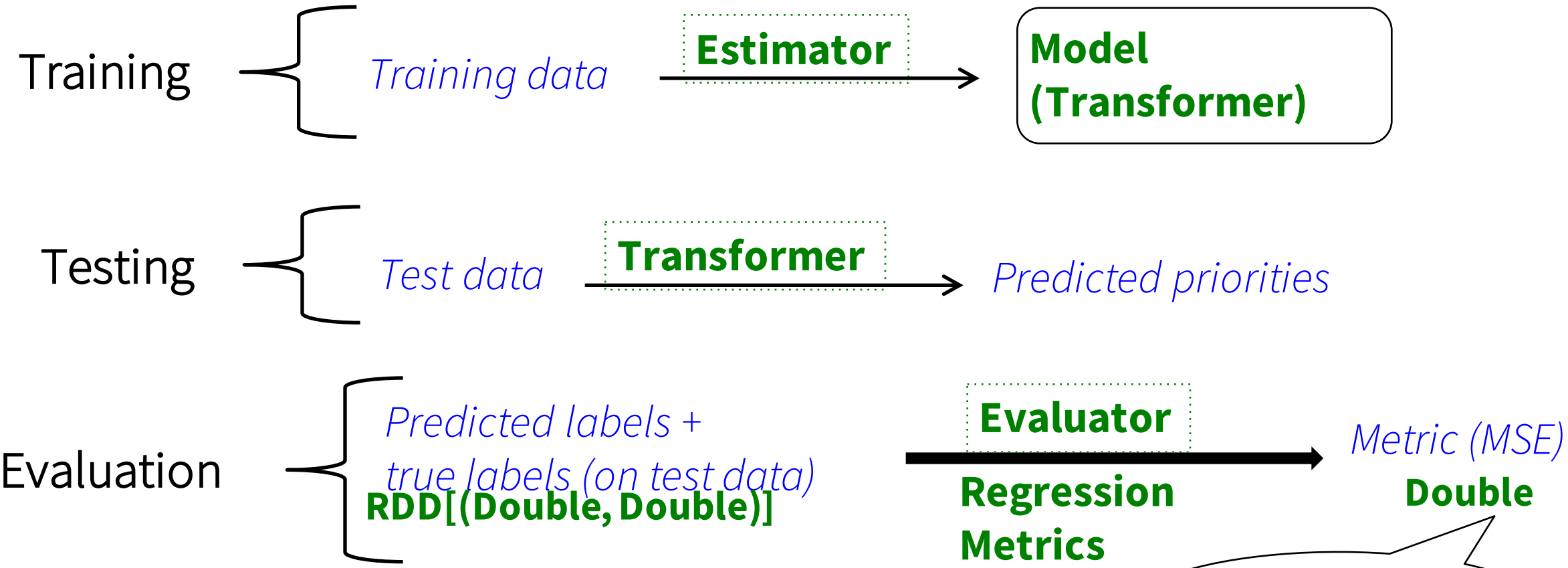
# Spark Machine Learning API

- http://spark.apache.org/docs/latest/ml-guide.html
- "Pipelines" `spark.ml` API
  - Primary high-level API for constructing ML workflows using DataFrames
  - The focus of future development
- Original `spark.mllib` API
  - API using RDDs
  - Preserved and stable

# Workflow: training + testing



# + Evaluation




Model selection

Choose among different models or model hyperparameters

# Summary: ML Overview

Components

- Model
- Dataset
- Algorithm

Processes

- Training – Test – Evaluation
- Model selection

# Overview of ML Algorithms

- Prediction
  - Regression
  - Classification
- Feature transformation
- Recommendation
- Clustering
- Other
  - Statistics
  - Linear algebra
  - Optimization

# Overview of ML Algorithms

- Prediction — feature vector → label
  - Regression (real value label)
  - Classification (categorical label)
- Feature transformation
- Recommendation
- Clustering
- Other
  - Statistics
  - Linear algebra
  - Optimization

E.g.: given log file, predict priority

**`LinearRegression, DecisionTree`**
**`LogisticRegression, NaiveBayes`**

Iterative optimization

Partition data by rows (instances):

- Easy to handle billions of rows
- Hard to scale # features
  - $10^7$ for *Regression, SVM, NB
  - $10^3$ for DecisionTree

# Overview of ML Algorithms

- Prediction
  - Regression
  - Classification
- Feature transformation
- Recommendation
- Clustering
- Other
  - Statistics
  - Linear algebra
  - Optimization

E.g.: convert text to feature vectors

```
Tokenizer, HashingTF, IDF, Word2Vec
```

```
Normalizer, StandardScaler
```

Arguably the *most important* part of machine learning

Per-row transformation

# Overview of ML Algorithms

- Prediction
  - Regression
  - Classification
- Feature transformation
- Recommendation
  user → recommended products
- Clustering
- Other
  - Statistics
  - Linear algebra
  - Optimization

E.g.: Recommend movies to users

**ALS**

Phrased as matrix factorization
Given (users x products) matrix with many missing entries,
Find low-rank factorization.
Fill in missing entries.

Partition data by both users and products.

Scales to millions of users and products.

# Overview of ML Algorithms

- Prediction
  - Regression
  - Classification
- Feature transformation
- Recommendation
- Clustering
  feature vectors → clusters (no labels)
- Other
  - Statistics
  - Linear algebra
  - Optimization

E.g.: Given news articles, automatically group articles by topics

**KMeans, GaussianMixutre, LDA**

Iterative optimization.
Local optima.

Partition data by rows.
Easy to handle billions of rows

# Overview of ML Algorithms

- Prediction
  - Regression
  - Classification
- Feature transformation
- Recommendation
- Clustering
- Other (MLLib)
  - Statistics
  - Linear algebra
  - Optimization

E.g.: Is model A significantly better than model B?

```
ChiSqSelector, Statistics,
  MultivariateOnlineSummarizer
```

E.g.: Matrix decomposition

```
DenseMatrix, SparseMatrix,
  EigenValueDecomposition, ...
```

E.g.: Given function f(x), find x to minimize f(x)

```
GradientDescent, LBFGS
```

# Machine Learning Integration

`spark.ml` provides a uniform set of high-level APIs that help users create and tune practical machine learning pipelines.

Spark ML standardizes APIs for machine learning algorithms to make it easier to combine multiple algorithms into a single *pipeline*, or *workflow*.

# Machine Learning Integration

Spark ML uses DataFrames as a dataset which can hold a variety of data types.

For instance, a dataset could have different columns storing text, feature vectors, true labels, and predictions.

# ML: Transformer

- A *Transformer* is an algorithm which can transform one DataFrame into another DataFrame.
- A `Transformer` object is an abstraction which includes *feature transformers* and *learned models*.
- Technically, a Transformer implements a `transform()` method that converts one DataFrame into another, generally by appending one or more columns.

# ML: Transformer

A *feature transformer* might:

- take a dataset,
- read a column (e.g., text),
- convert it into a new column (e.g., feature vectors),
- append the new column to the dataset, and
- output the updated dataset.

# ML: Transformer

A *learning model* might:

- take a dataset,
- read the column containing feature vectors,
- predict the label for each feature vector,
- append the labels as a new column, and
- output the updated dataset.

# ML: Estimator

An *Estimator* is an algorithm which can be fit on a DataFrame to produce a Transformer.

For instance, a learning algorithm is an Estimator that trains on a dataset and produces a model.

# ML: Estimator

- An Estimator abstracts the concept of any algorithm which fits or trains on data.
- Technically, an Estimator implements a **`fit()`** method that accepts a DataFrame and produces a Transformer.
- For example, a learning algorithm like **`LogisticRegression`** is an Estimator, and calling its **`fit()`** method trains a **`LogisticRegressionModel`**, which is a Transformation.

# ML: Param

All Transformers and Estimators now share a common API for specifying parameters.

# ML: Pipeline

In machine learning, it is common to run a sequence of algorithms to process and learn from data. A simple text document processing workflow might include several stages:

- Split each document's text into words.
- Convert each document's words into a numerical feature vector.
- Learn a prediction model using the feature vectors and labels.

Spark ML represents such a workflow as a **Pipeline**, which consists of a sequence of **PipelineStages** (Transformers and Estimators) to be run in a specific order.

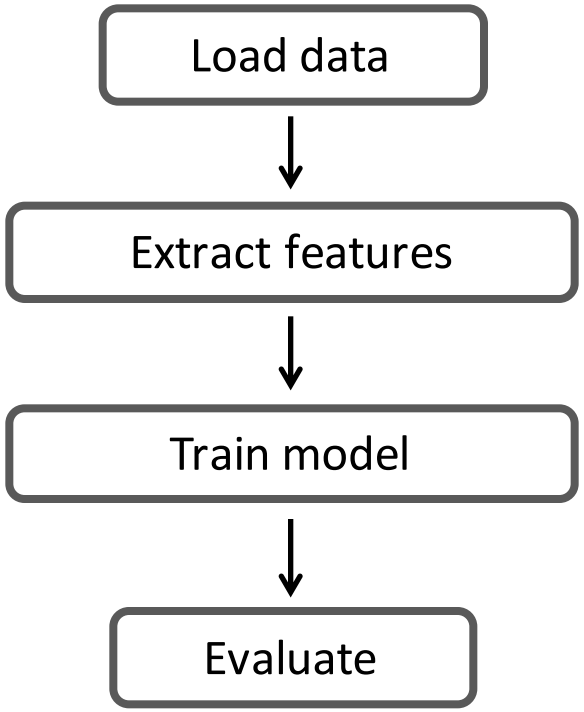
# ML Pipelines

Simple construction, tuning, and testing for ML workflows

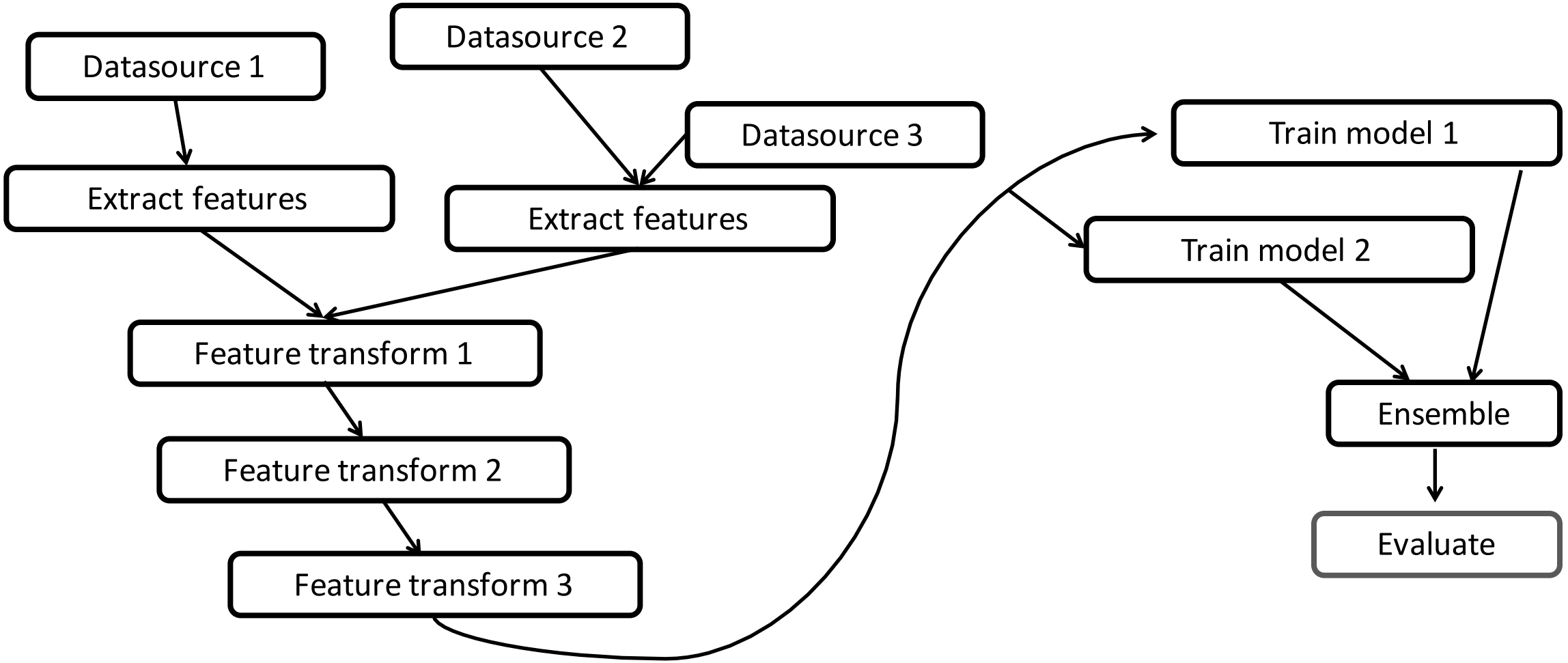
## ML Pipelines provide:

- Familiar API based on scikit-learn
- Integration with DataFrames
- Simple parameter tuning
- User-defined components

# ML Pipelines



# ML Pipelines



# ML: Scala Example

```scala
import org.apache.spark.ml.Pipeline
import org.apache.spark.ml.feature.{HashingTF, Tokenizer}
import org.apache.spark.ml.classification.LogisticRegression

val tokenizer = new Tokenizer().
  setInputCol("text").
  setOutputCol("words")
val hashingTF = new HashingTF().
  setNumFeatures(1000).
  setInputCol(tokenizer.getOutputCol).
  setOutputCol("features")
val lr = new LogisticRegression().
  setMaxIter(10).
  setRegParam(0.01)
val pipeline = new Pipeline().setStages(Array(tokenizer, hashingTF, lr))

val df    = sqlContext.load("/path/to/data")
val model = pipeline.fit(df)
```

# End of Spark Machine Learning

# Additional Resources

Books

- *Learning Spark.* http://shop.oreilly.com/product/0636920028512.do
- *Advanced Analytics with Spark.* http://shop.oreilly.com/product/0636920035091.do
- *Scala for the Impatient.* http://www.informit.com/store/scala-for-the-impatient-9780321774095

Web Sites

- Spark documentation: http://spark.apache.org/docs/latest/
- Databricks blog: https://databricks.com/blog

# Thank you.