Spark Essentials

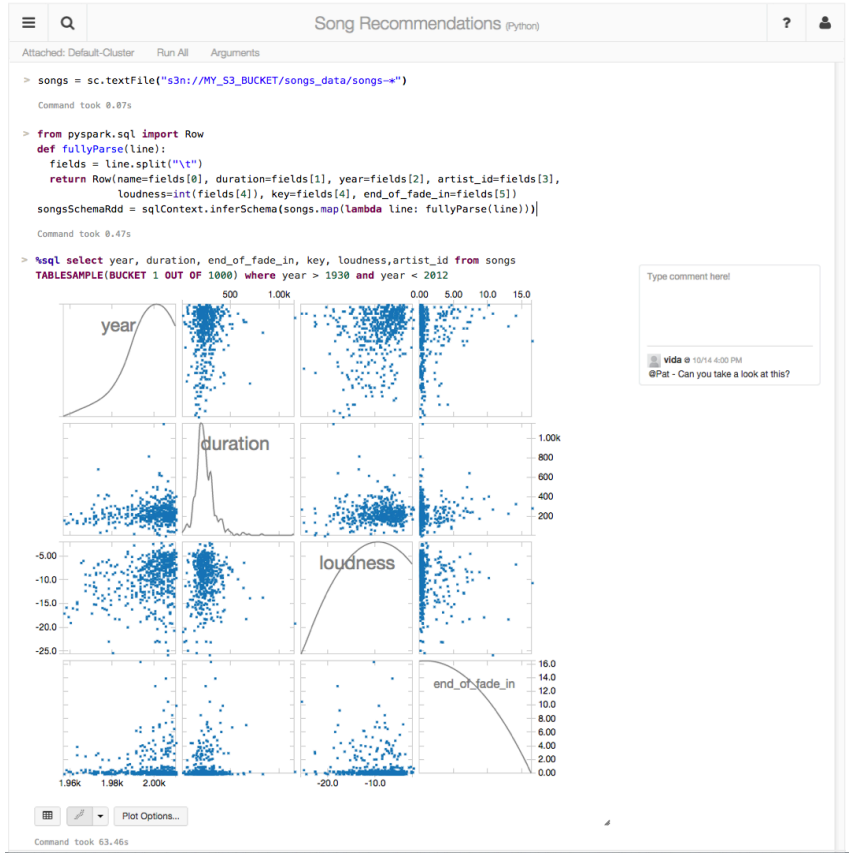
at Spark Summit East 2016

Brian Clapper

databricks™

making big data simple

- Founded in late 2013
- by the creators of Apache Spark
- Original team from UC Berkeley AMPLab
- Raised $47 Million in 2 rounds
- ~60 employees
- We're hiring! (http://databricks.workable.com)
- Level 2/3 support partnerships with
  - Hortonworks
  - MapR
  - DataStax



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

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

Instructor: Brian Clapper

LinkedIn: https://www.linkedin.com/in/bclapper
Twitter: @brianclapper

- 32+ years building systems for startups and large enterprises

- 2+ years teaching front- and back-end technologies
  - Scala, Java, AngularJS, Ruby, Python, JavaScript, Spark
  - I've taught more 15 Spark classes since June, 2015

- Scala programmer since early 2009
- Founder and organizer of Philly Area Scala Enthusiasts (PHASE)
- Co-organizer of annual Northeast Scala Symposium

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

Files and Resources

Documents

- Slides: http://tinyurl.com/sse2016-intro-class-slides-pdf
- Labs (HTML format): http://tinyurl.com/sse2016-intro-class-labs.zip

Databricks

- URL: https://sse2016-es01.cloud.databricks.com
- Your username and password should be on the back of your badge.
- NOTE: Use Chrome or Firefox
  - If you have a recent version of Chrome, you may encounter some Spark UI issues. Firefox works fine.
  - Internet ~~Exploder~~ Explorer is *not* supported
  - Safari may or may not work.

Hands on

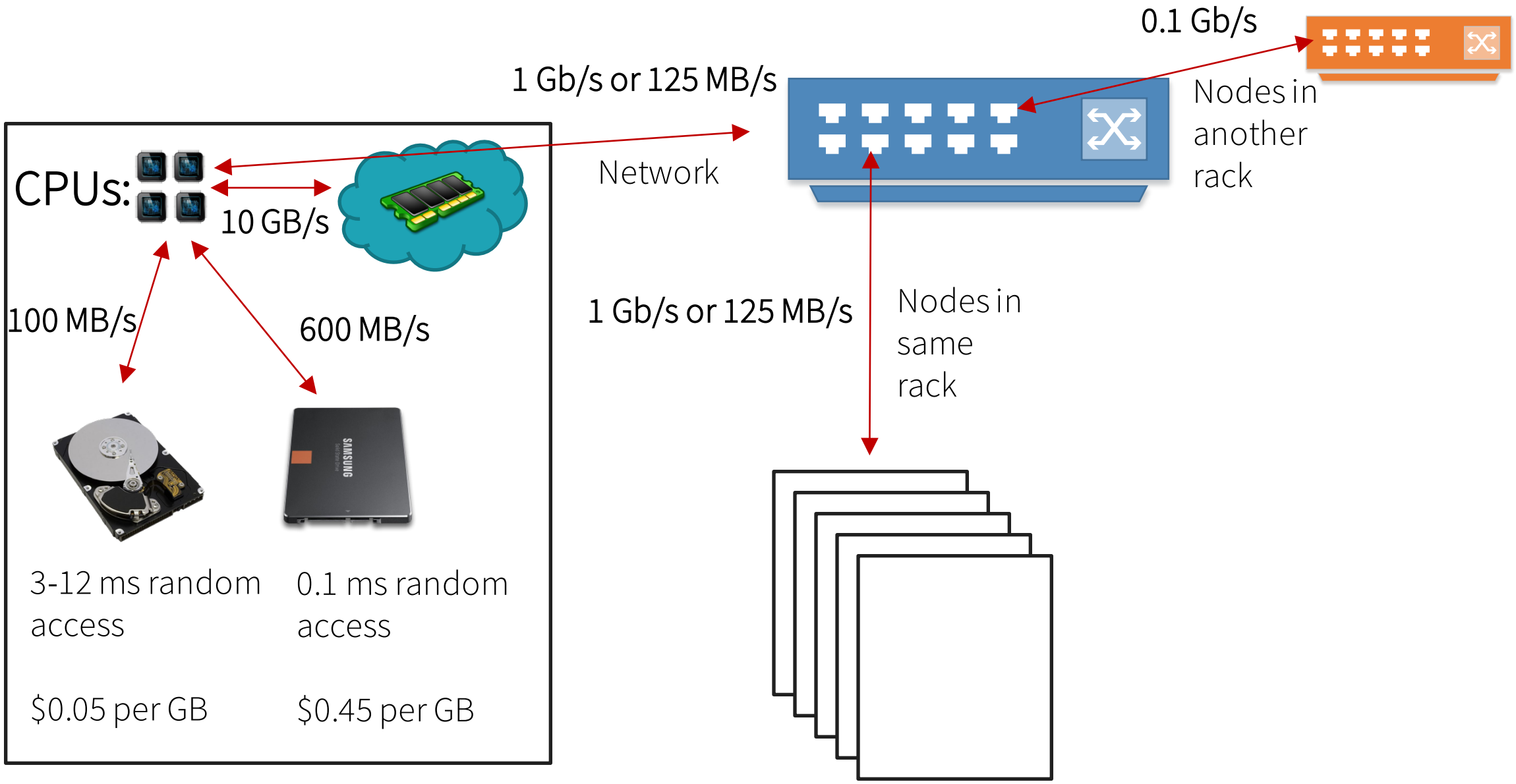
Take a minute to log into your Databricks account and find your home directory.

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.

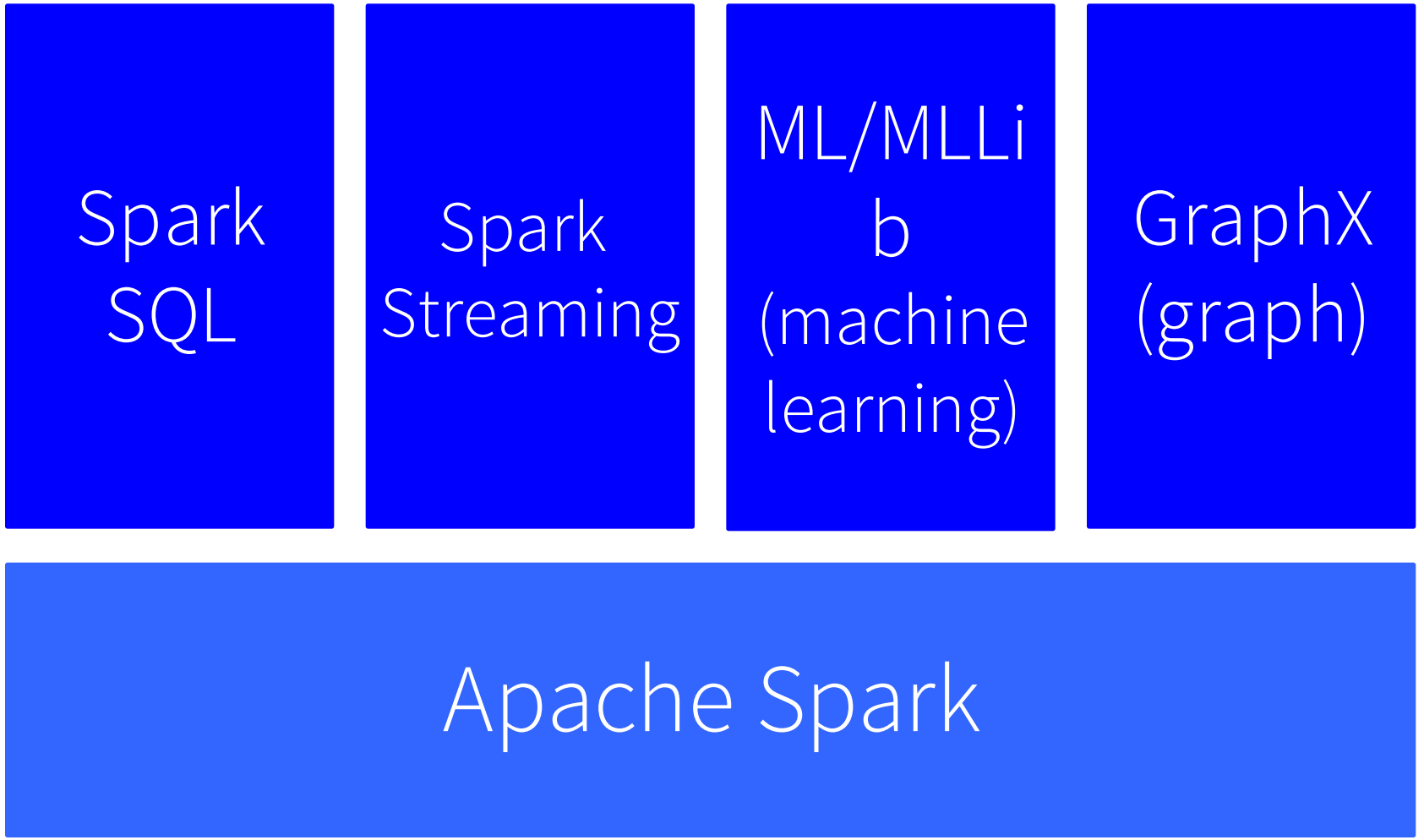
Overview

- Started as a research project at UC Berkeley in 2009
- Open Source License (Apache 2.0)
- Latest Stable Release: v1.5.1 (Sept 2015)
- 600,000 lines of code (75% Scala)
- Built by 800+ developers from 200+ companies


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

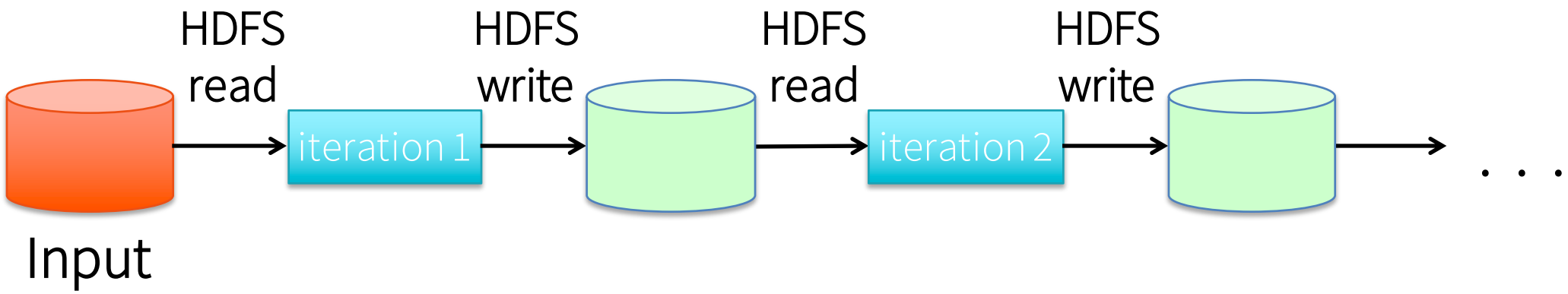
Opportunity

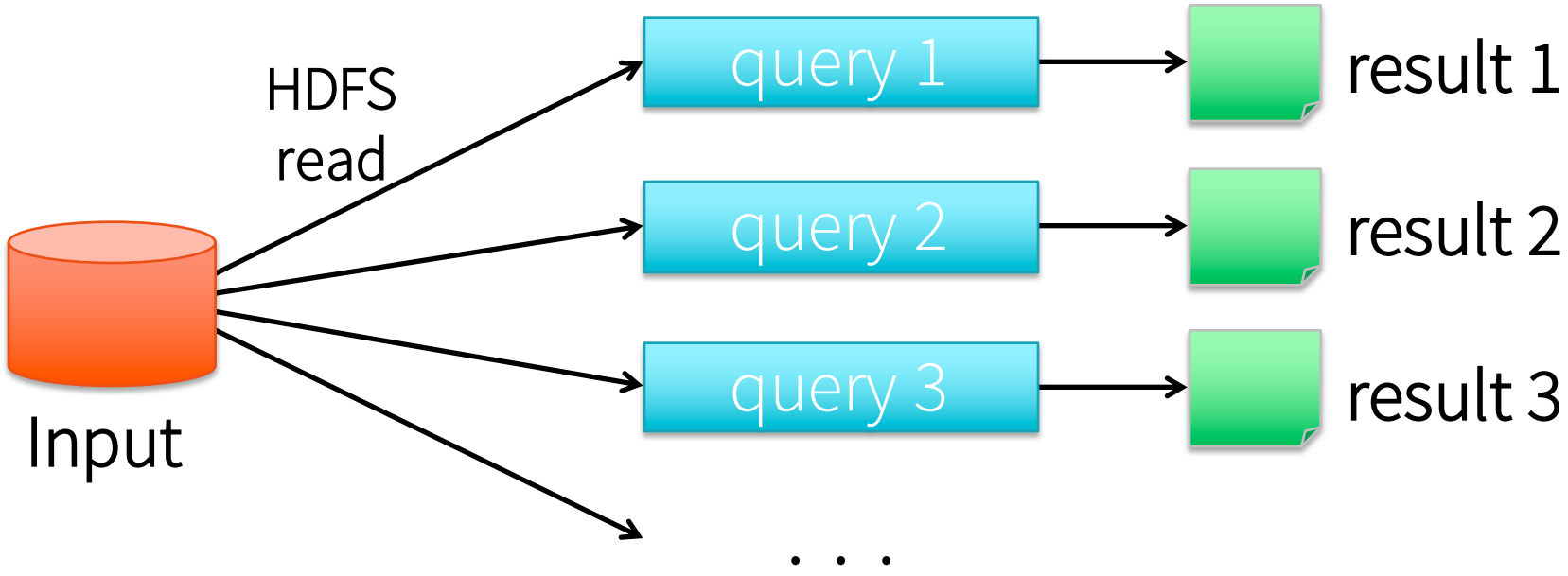
- Keep more data *in-memory*
- New distributed execution environment
- Bindings for:
  - Python, Java, Scala, R



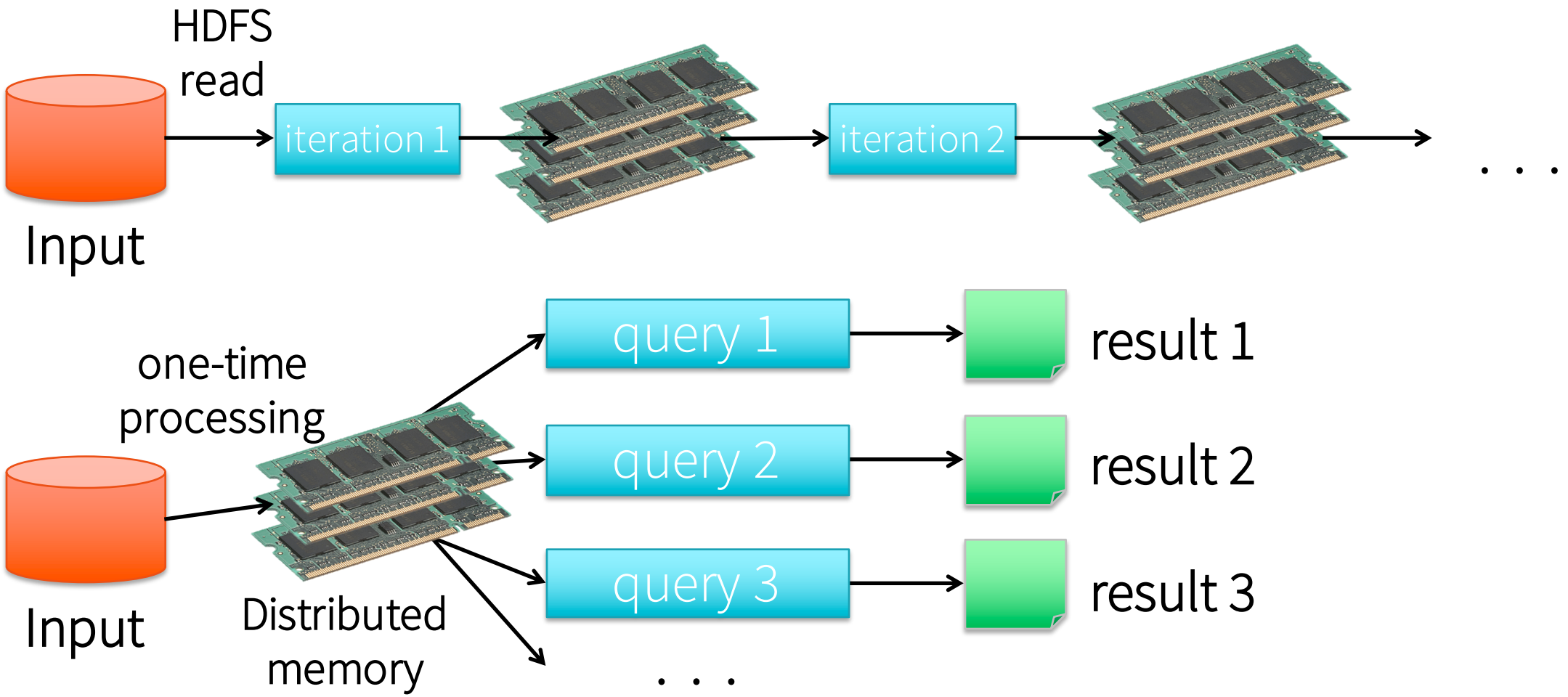
http://people.csail.mit.edu/matei/papers/2010/hotcloud_spark.pdf

Use Memory Instead of Disk

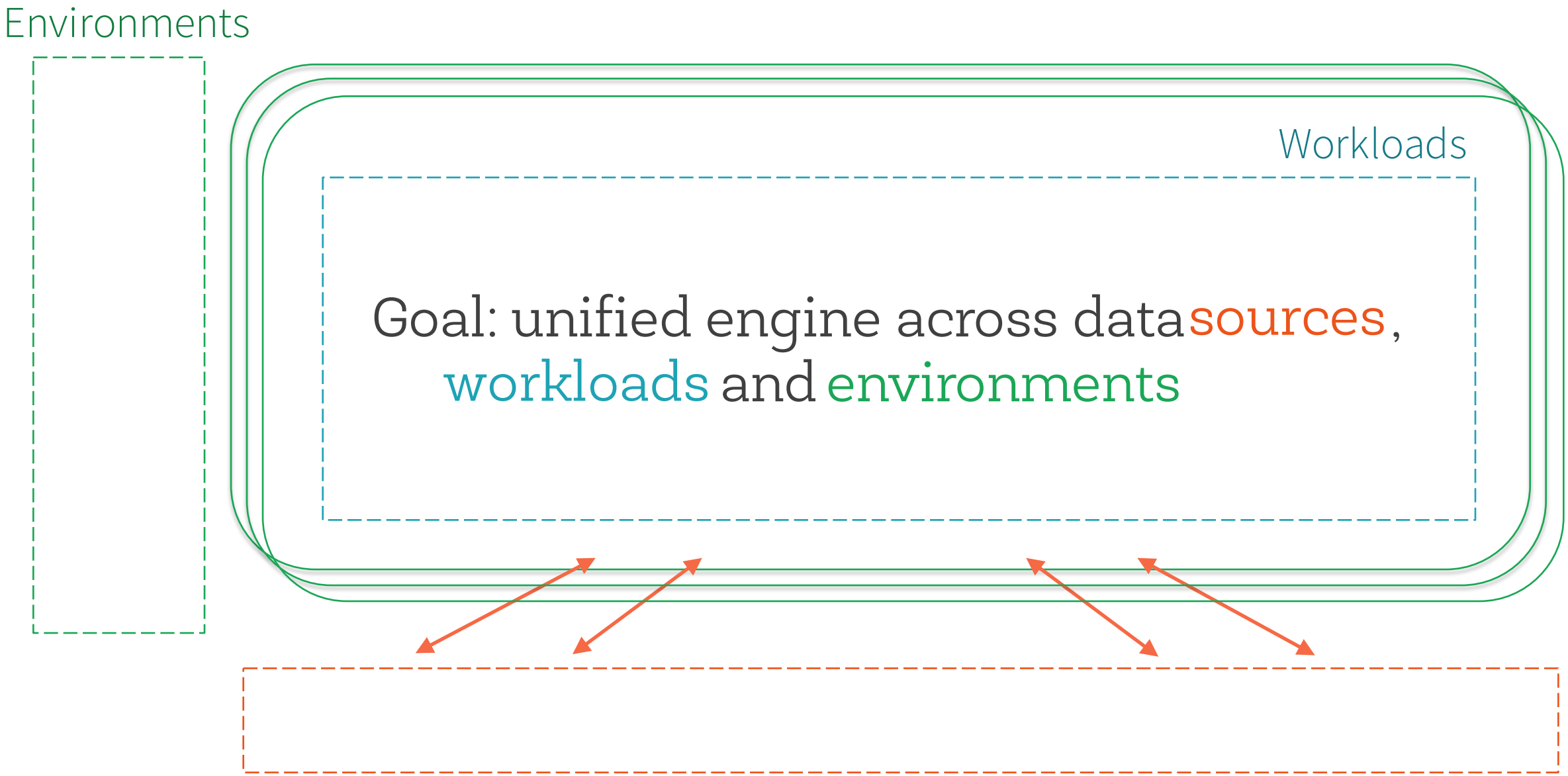




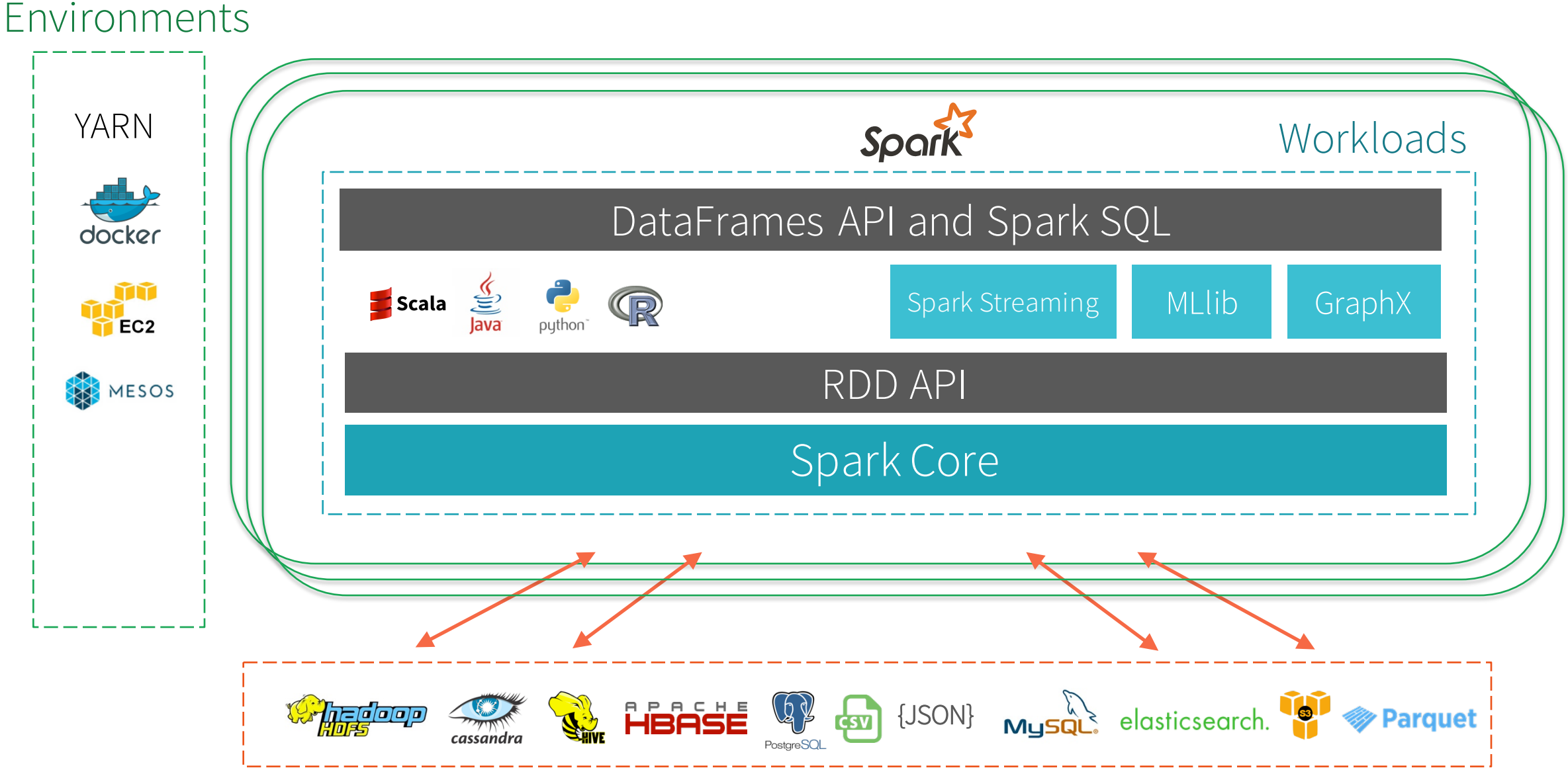
In-Memory Data Sharing



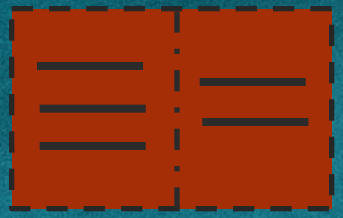
10-100x faster than network and disk


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

databricks


Environments
YARN
docker
EC2
MESOS
Spark
Workloads
DataFrames API and Spark SQL
Scala
Java
python
Spark Streaming
MLlib
GraphX
RDD API
Spark Core
hadoop HDFS
cassandra
HIVE
APACHE HBASE
PostgreSQL
CSV
{JSON}
MySQL
elasticsearch.
Parquet
Data Sources

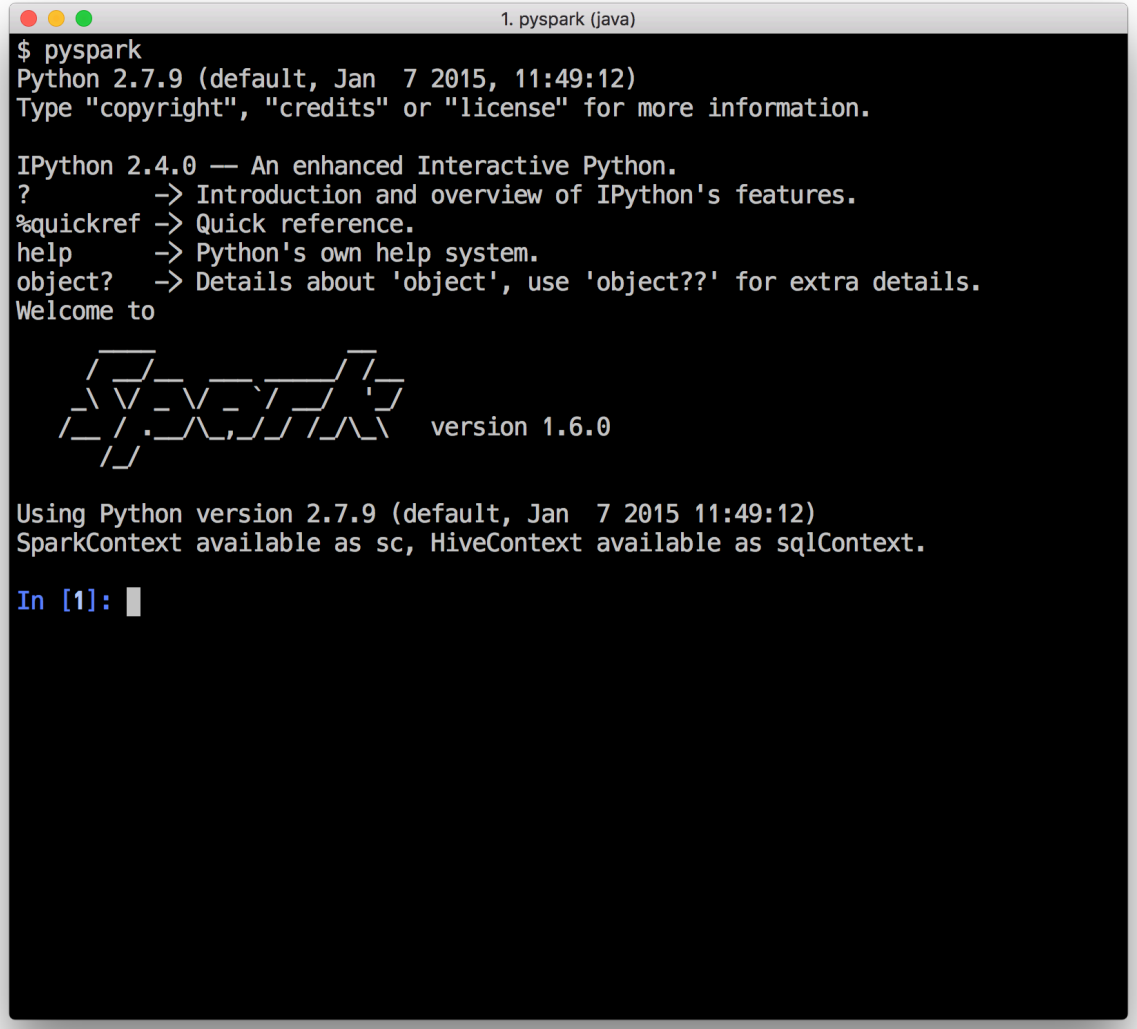
End of Spark Overview



RDD Fundamentals

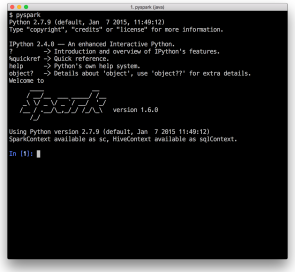
databricks™

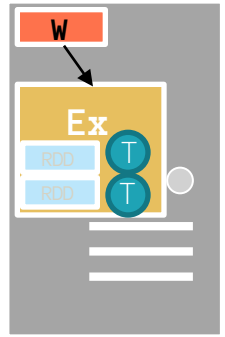
Interactive Shell



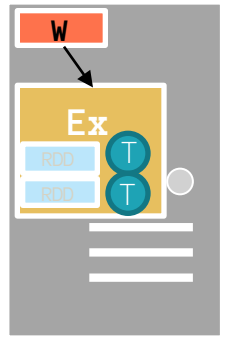
(Scala, Python and R only)

Driver Program





Worker Machine

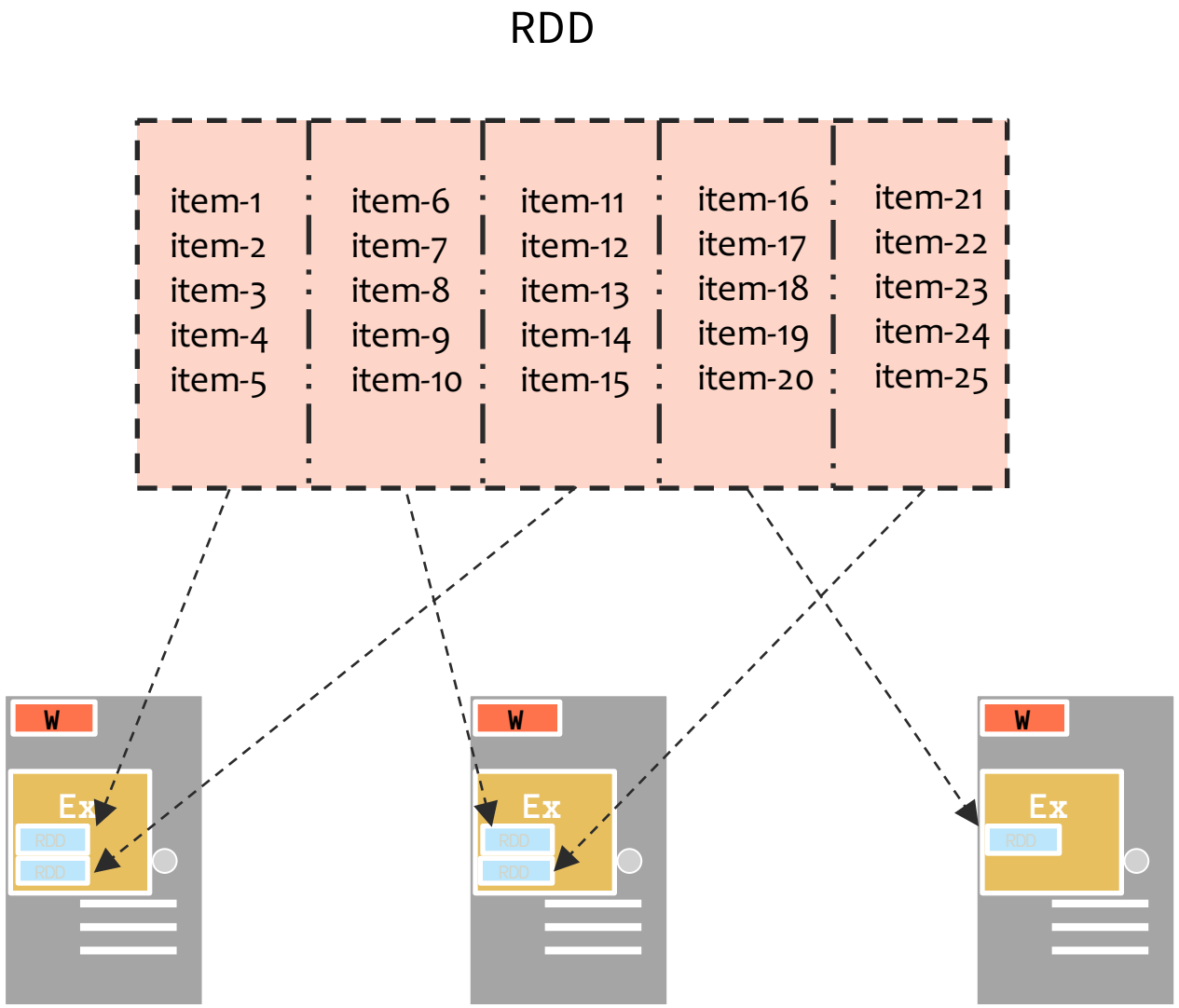


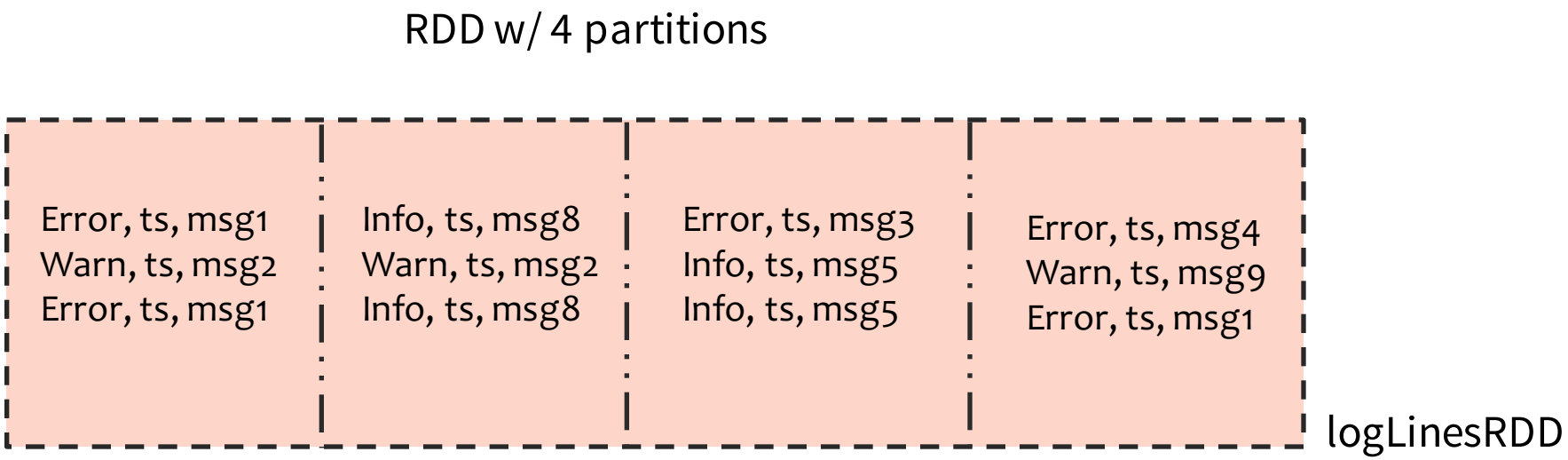
Worker Machine

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("hdfs:/path/to/README.md")
```

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

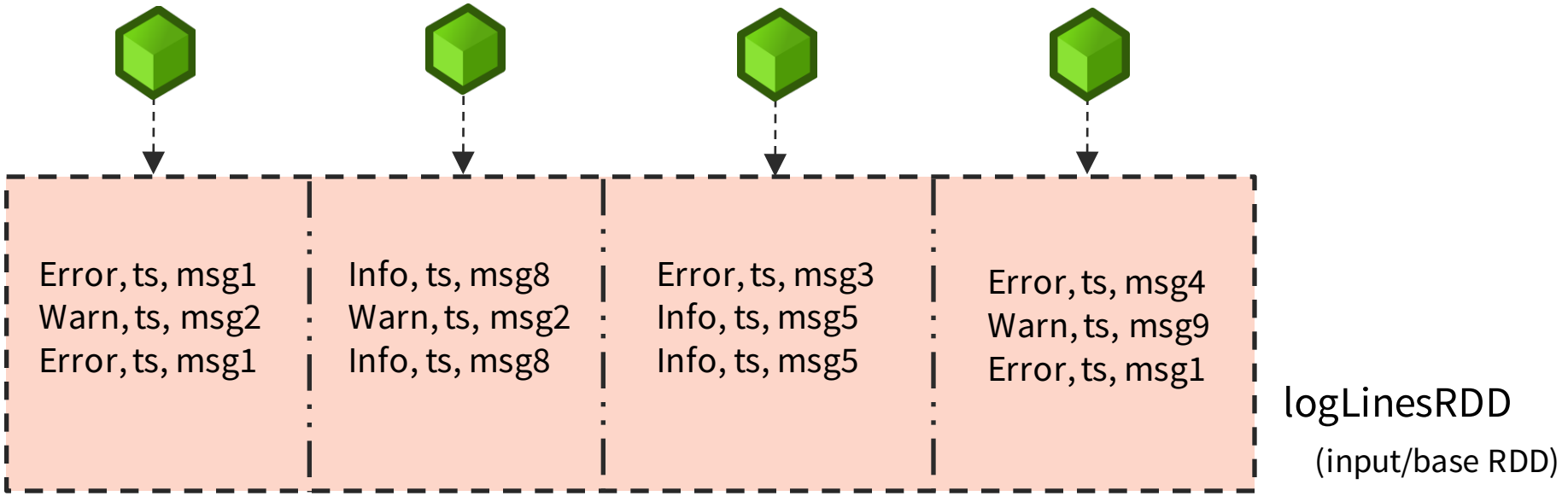
```java
// Read a local txt file in Java
JavaRDD<String> lines = sc.textFile("hdfs:/path/to/README.md");
```

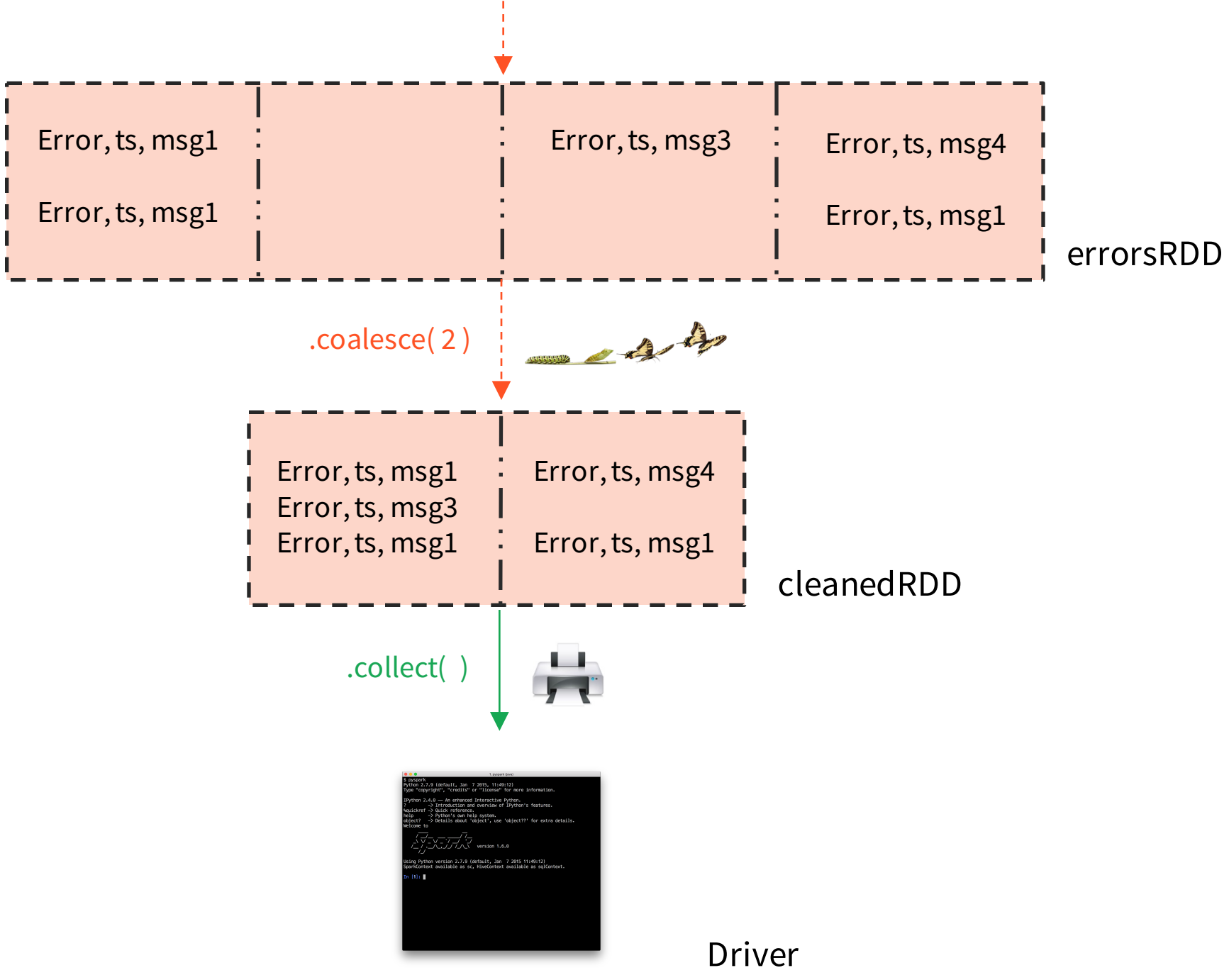
There are other methods to read data from HDFS, C*, S3, HBase, etc.

Operations on Distributed Data

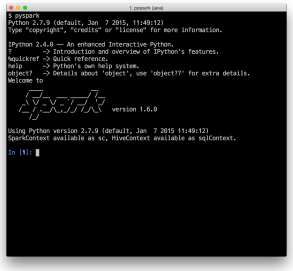
- Two types of operations: *transformations* and *actions*
- Transformations are lazy (*not computed immediately*)
- Transformations are executed when an action is run
- Persist (cache) distributed data in memory or disk


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

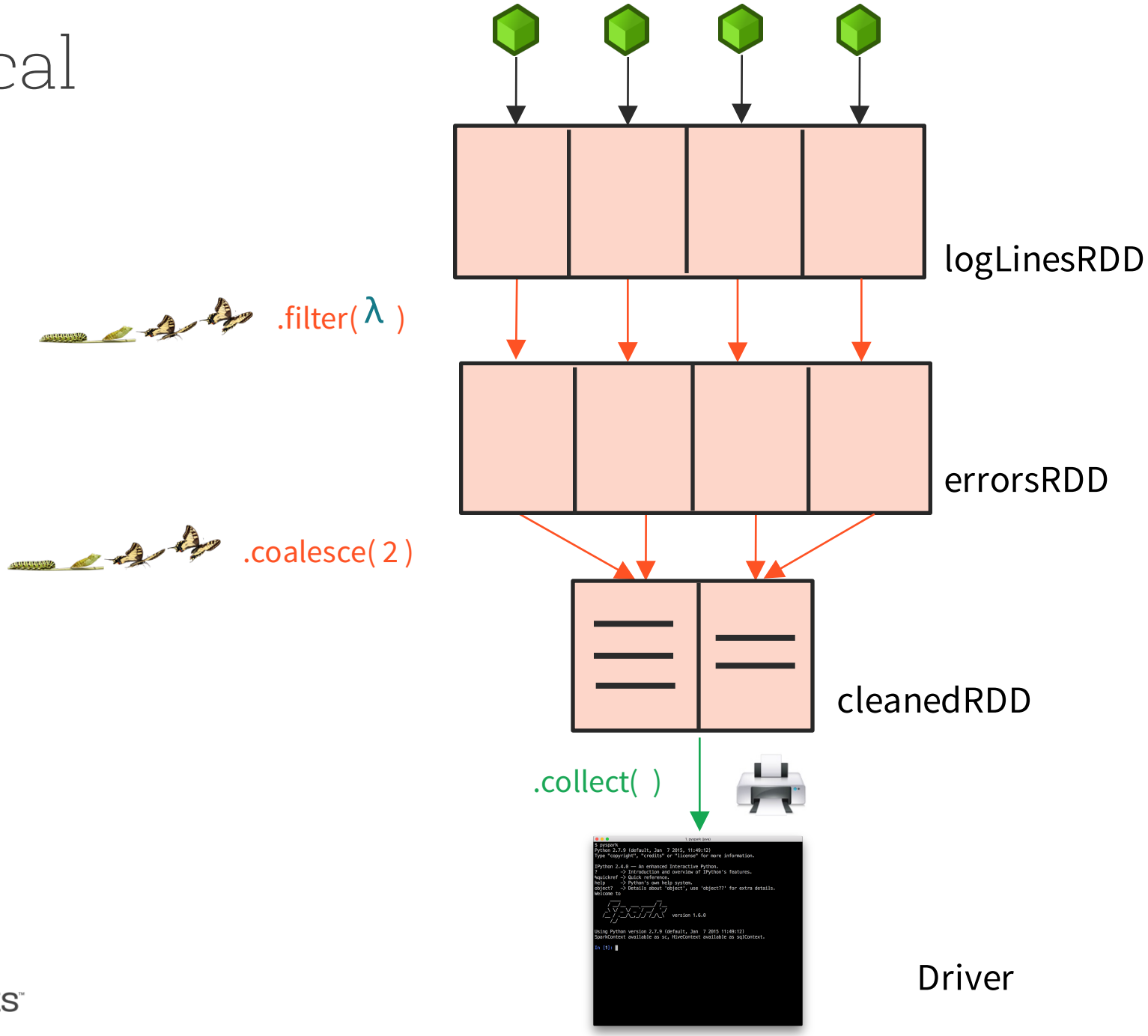
.collect( )



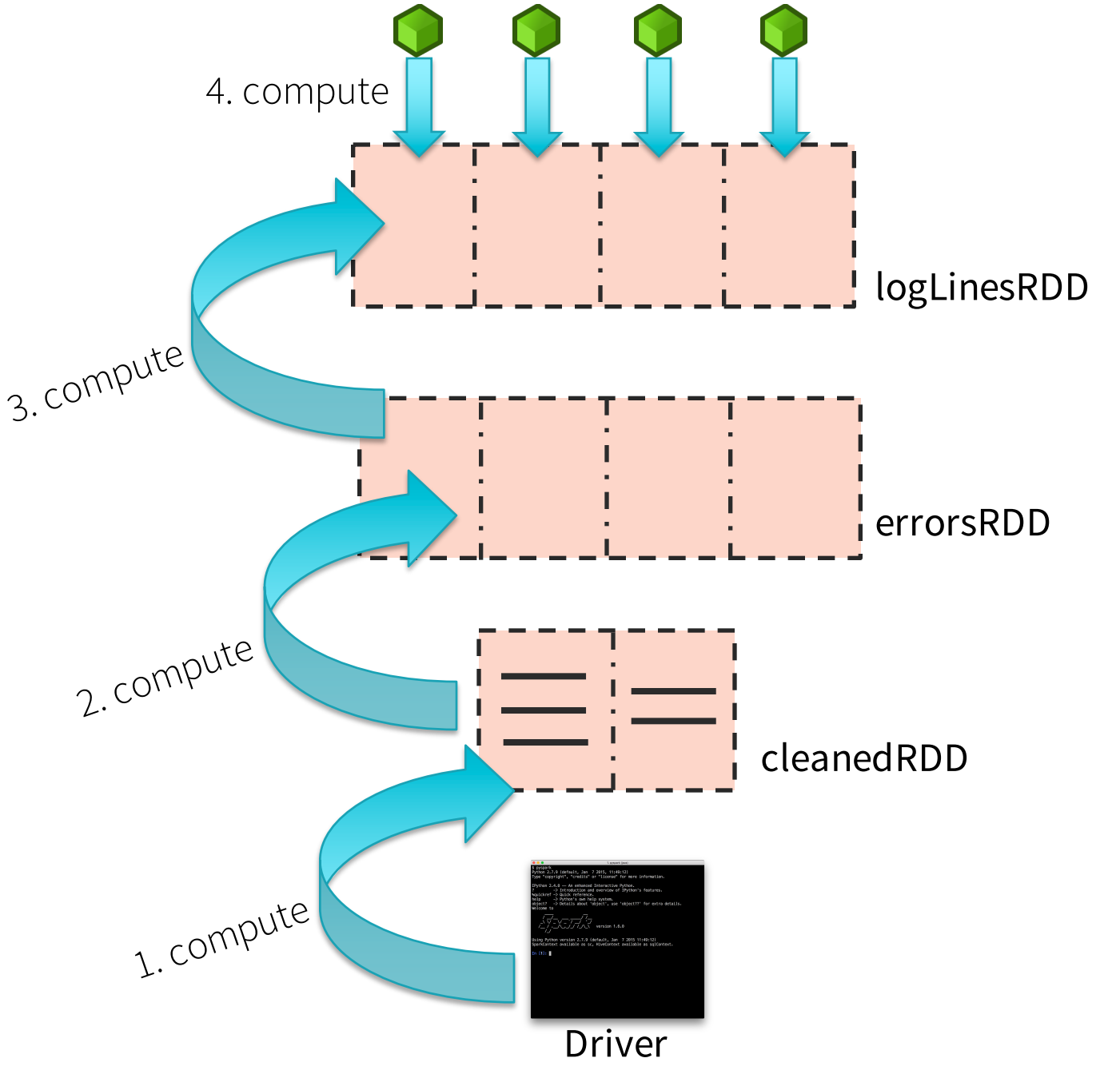
Driver

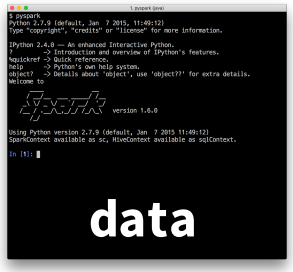
(I call this the Batman Slide.)

Logical

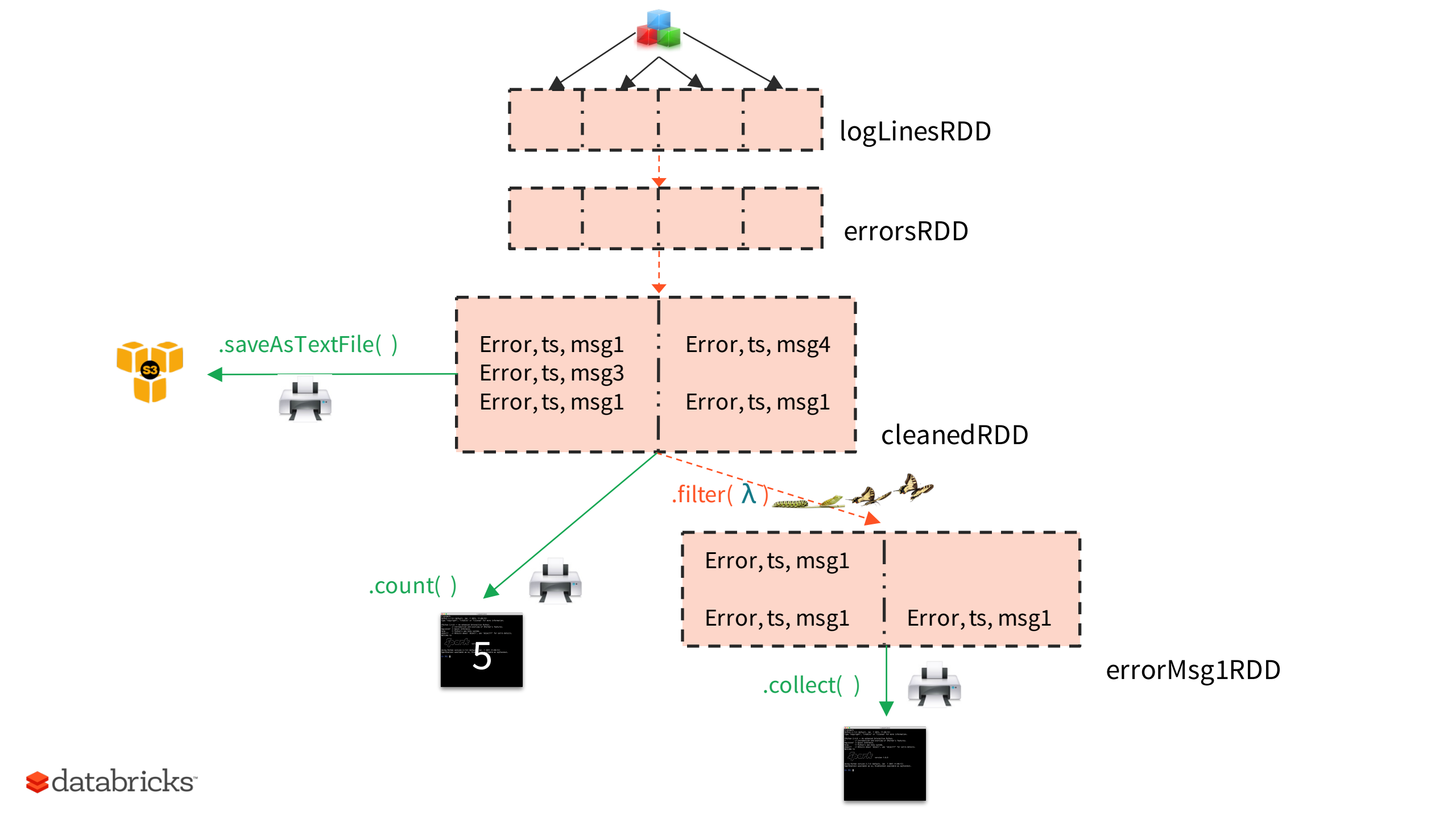

logLinesRDD
.filter(λ)
errorsRDD
.coalesce(2)
cleanedRDD
.collect( )
Driver

Physical

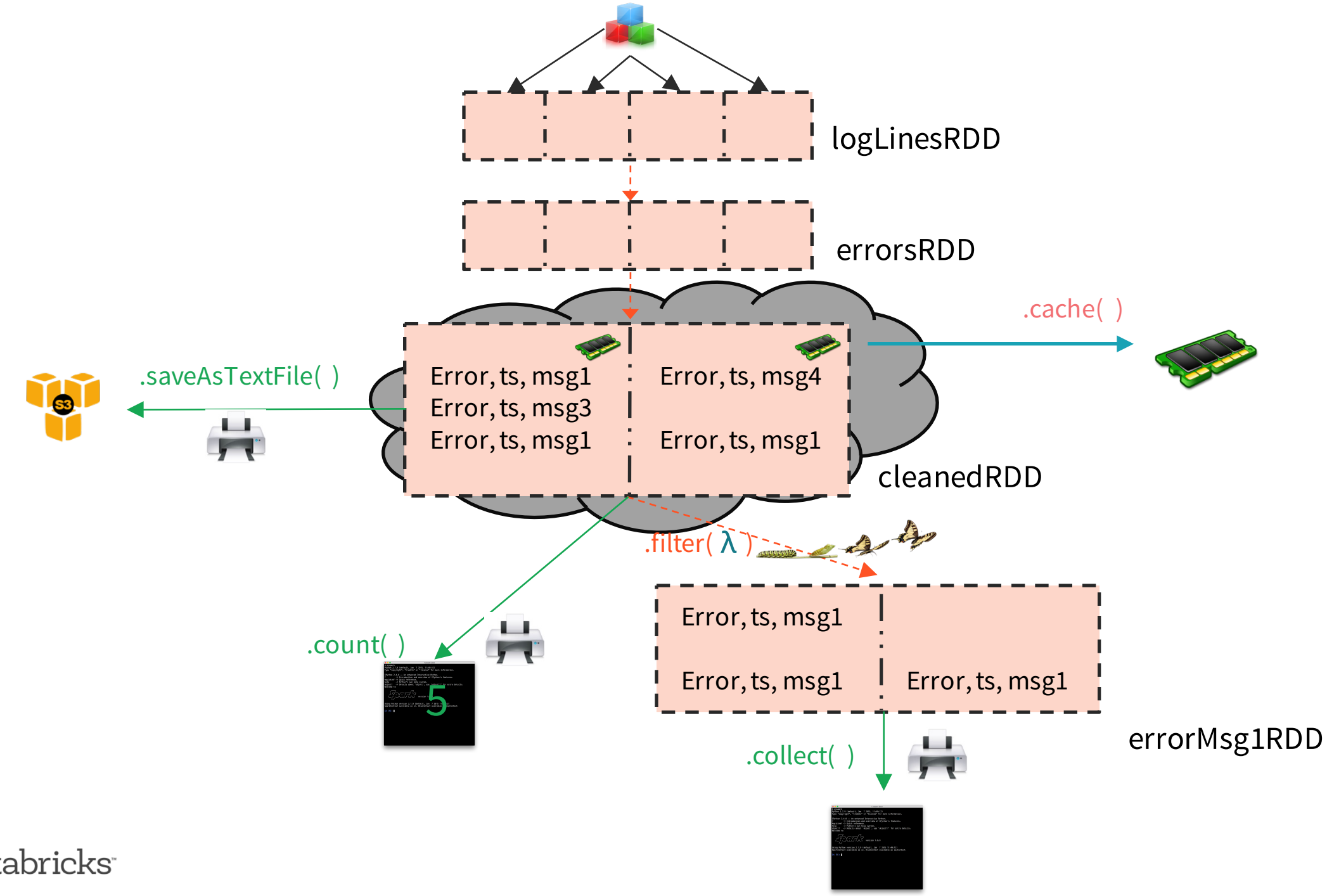

4. compute
logLinesRDD
3. compute
errorsRDD
2. compute
cleanedRDD
1. compute
Driver


$ 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
spark
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

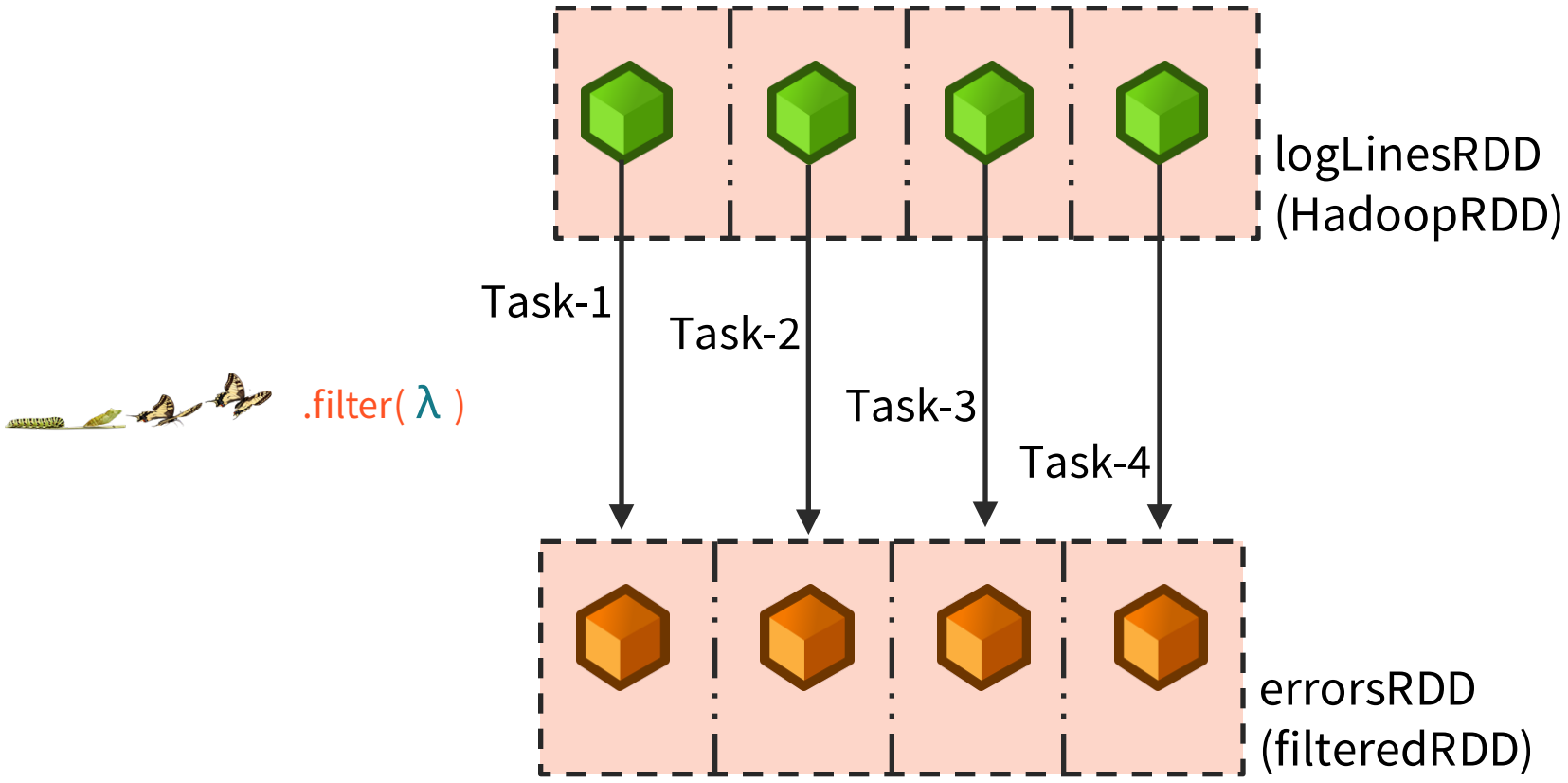
Driver


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( )

Partition >>> Task >>> Partition



RDD Demo

In the Databricks shard, look in your home folder. You should see a folder called "Hands-On". Underneath that folder, you'll find:

RDD > Wikipedia Page Counts

That's a Scala notebook, and we're going to walk through it together.

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 your "Hands-On > RDD" folder, and you'll find a "Washington Crime Lab" folder.

In *there*, you'll find Python and Scala folders.

- Decide whether you want to work in Python or Scala
- Open the appropriate lab notebook
- Attach the lab notebook to your cluster
- Follow the instructions in the notebook

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 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")
```

This is Scala, but the Python API is similar.

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

JDBC

{ JSON }

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

databricks

Hands on: Less Code

Let's take a quick look at that in action. In your home folder, under:

```
Hands-On > 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 the instructions.

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")
```

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"])
```

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("hdfs:/path/to/data.parquet")
val df2 = sqlContext.read.json("hdfs:/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("hdfs:/path/to/data.parquet")
df2 = sqlContext.read.json("hdfs:/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.,Parquet) 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.

Hands On with Schema Inference

Let's take a look at a concrete example. In the Databricks shard, in your "Hands-On" folder, you'll find the following notebook.

`SQL and DataFrames > Schema Inference`

Let's walk through it together.

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`
		`Fred,Hoover,91`
		`Joaquin,Hernandez,24`
		…

SQL Table	`dataFrame3`	**first** / **last** / **age** Joe / Smith / 42 Jill / Jones / 33

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

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`
		`Fred,Hoover,91`
		`Joaquin,Hernandez,24`
		…

SQL Table	dataFrame3	first last age Joe Smith 42 Jill Jones 33

dataFrame1
column: "first"

dataFrame2
column: "first"

dataFrame3
column: "first"

databricks

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"]` `df.first`†	`df.col("first")`	`df("first")` `$"first"`‡	`df$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()

As we saw in the last hand-on notebook, 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

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|
+---------+---+----------+----------+
```

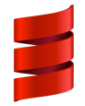
filter()

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, in your "Hands-On" folder, you find the following folder:

SQL and DataFrames > Operations

In there, you'll find a Scala notebook and a Python notebook.

- Open either the Python or Scala notebook
- Attach the notebook to your cluster
- Follow along with the instructor

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.

More hands-on with DataFrames

In the Databricks shard, look in

Workspace > Labs > 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, consult the Solutions notebook in the cloned folder.

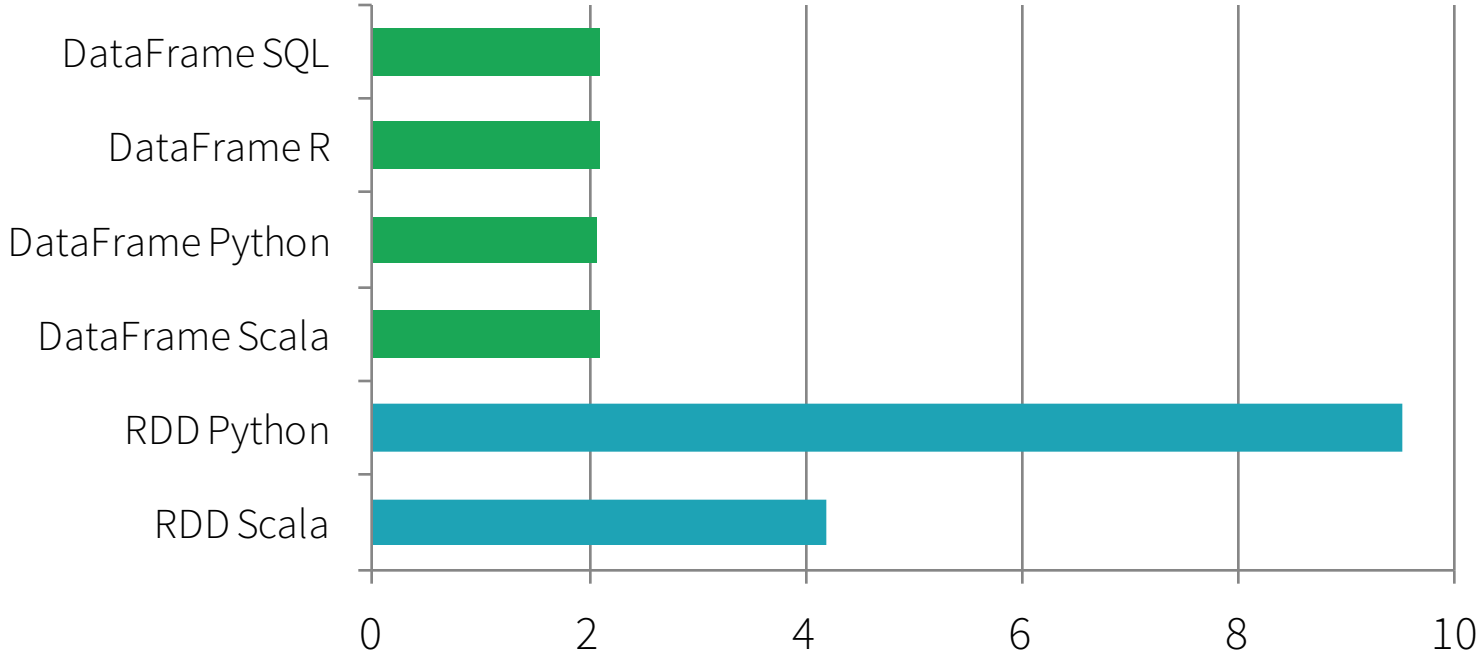
Writing DataFrames

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 are Optimized

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

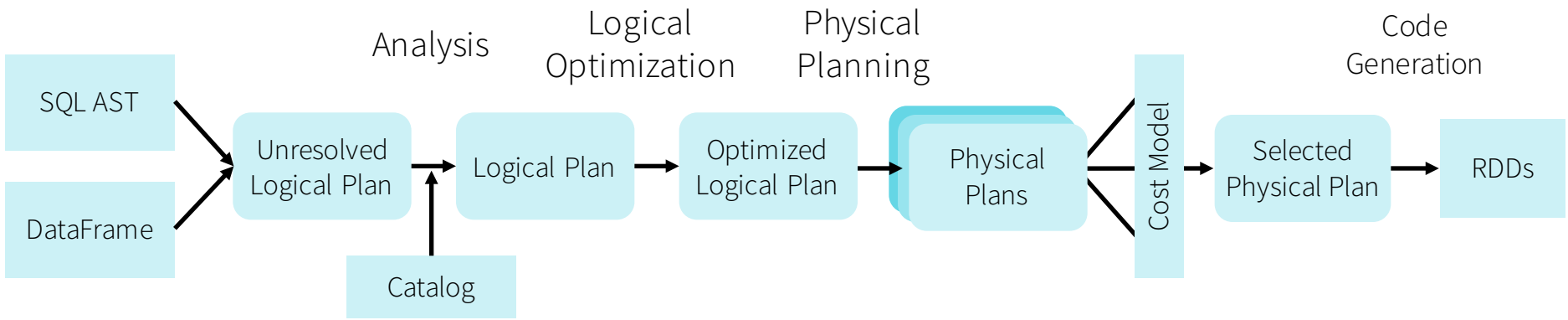


Time to aggregate 10 million integer pairs (in seconds)

Plan Optimization & Execution

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

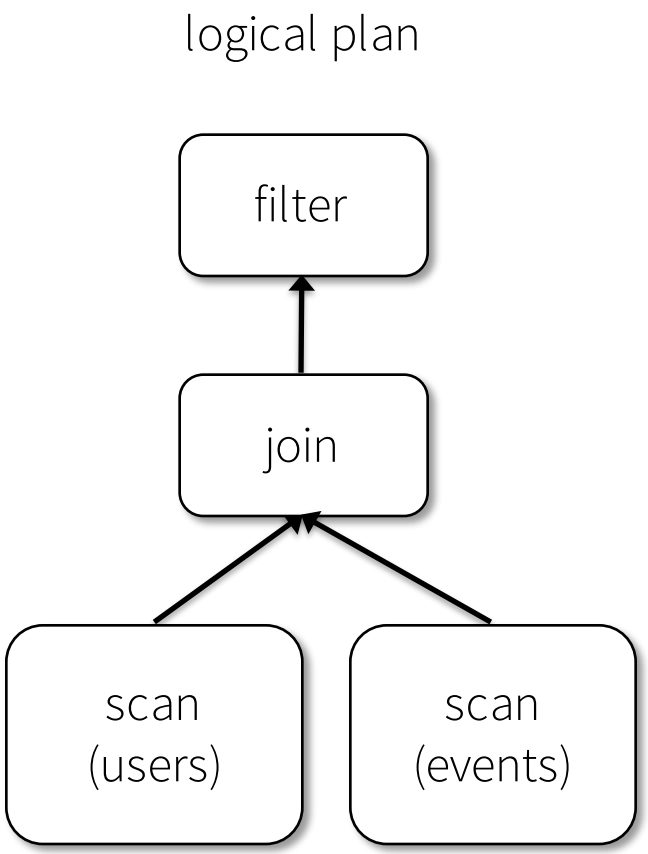
Plan Optimization & Execution



DataFrames and SQL share the same optimization/execution pipeline

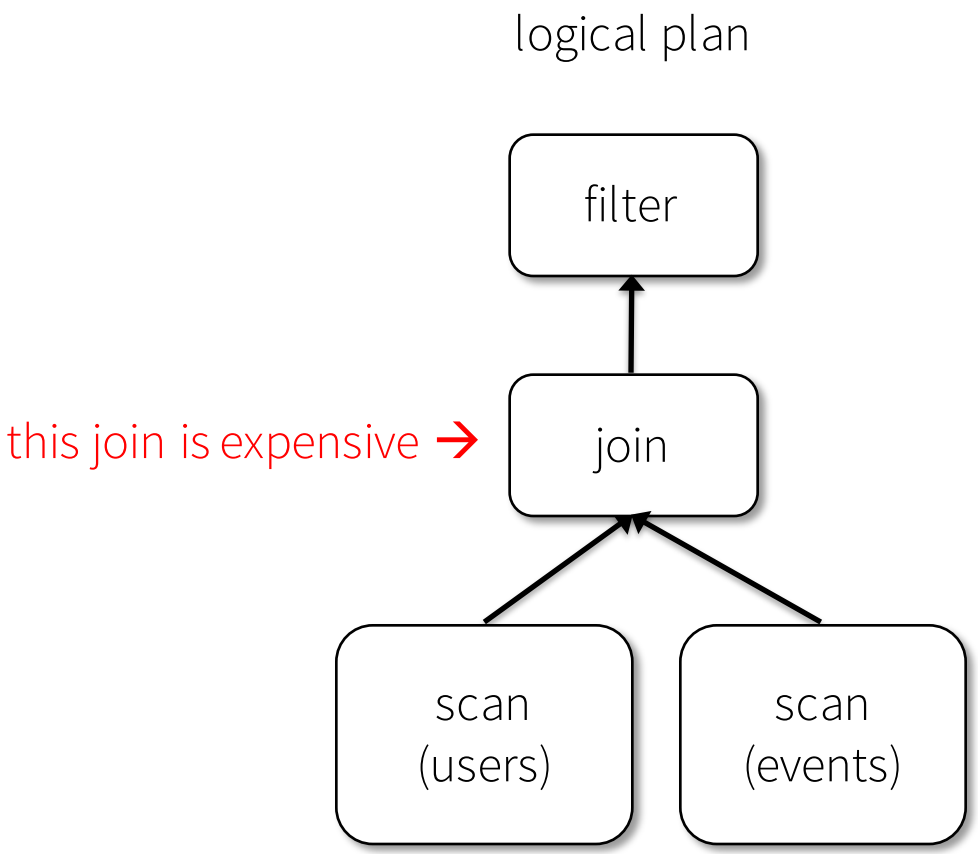
Plan Optimization & Execution

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



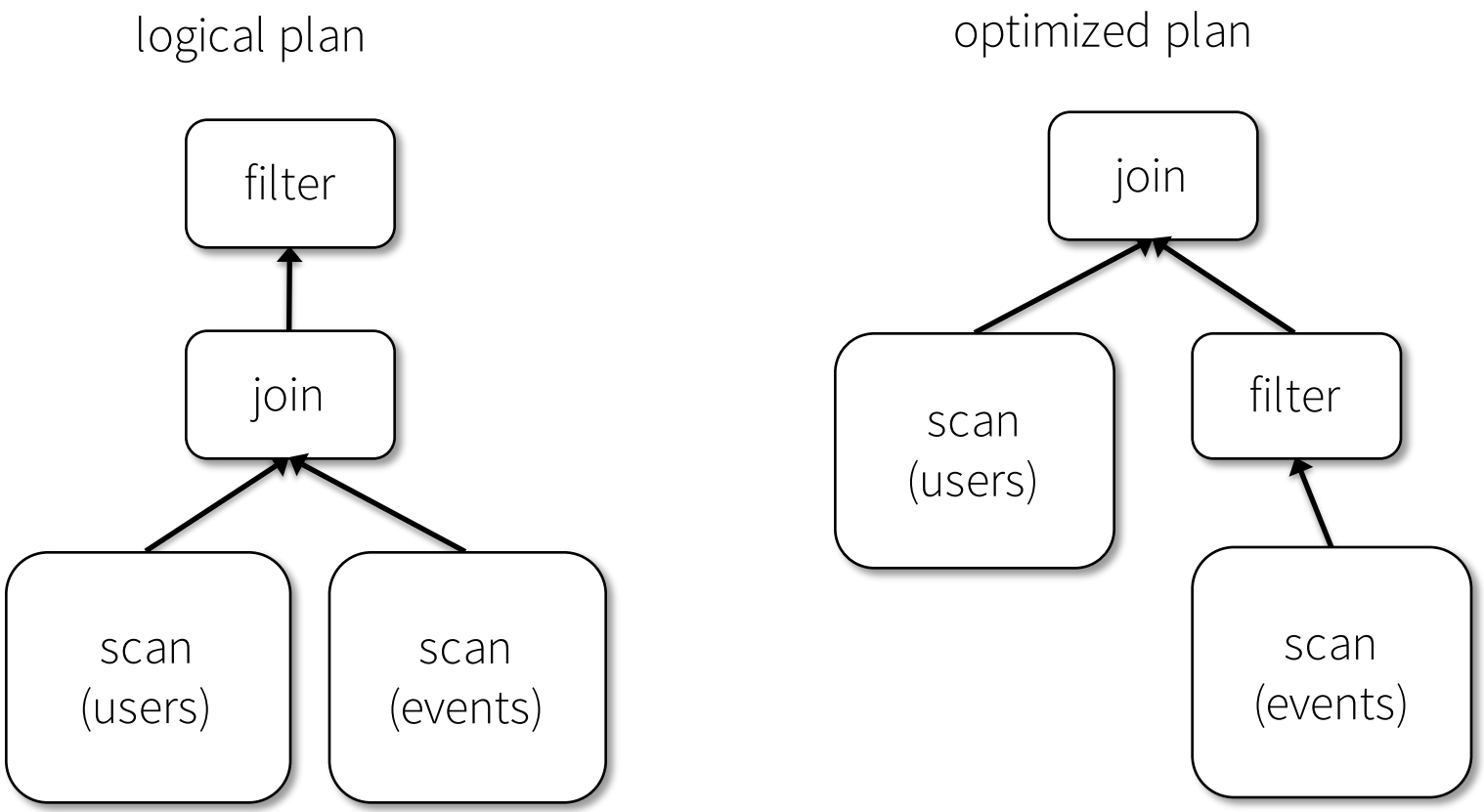
Plan Optimization & Execution

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



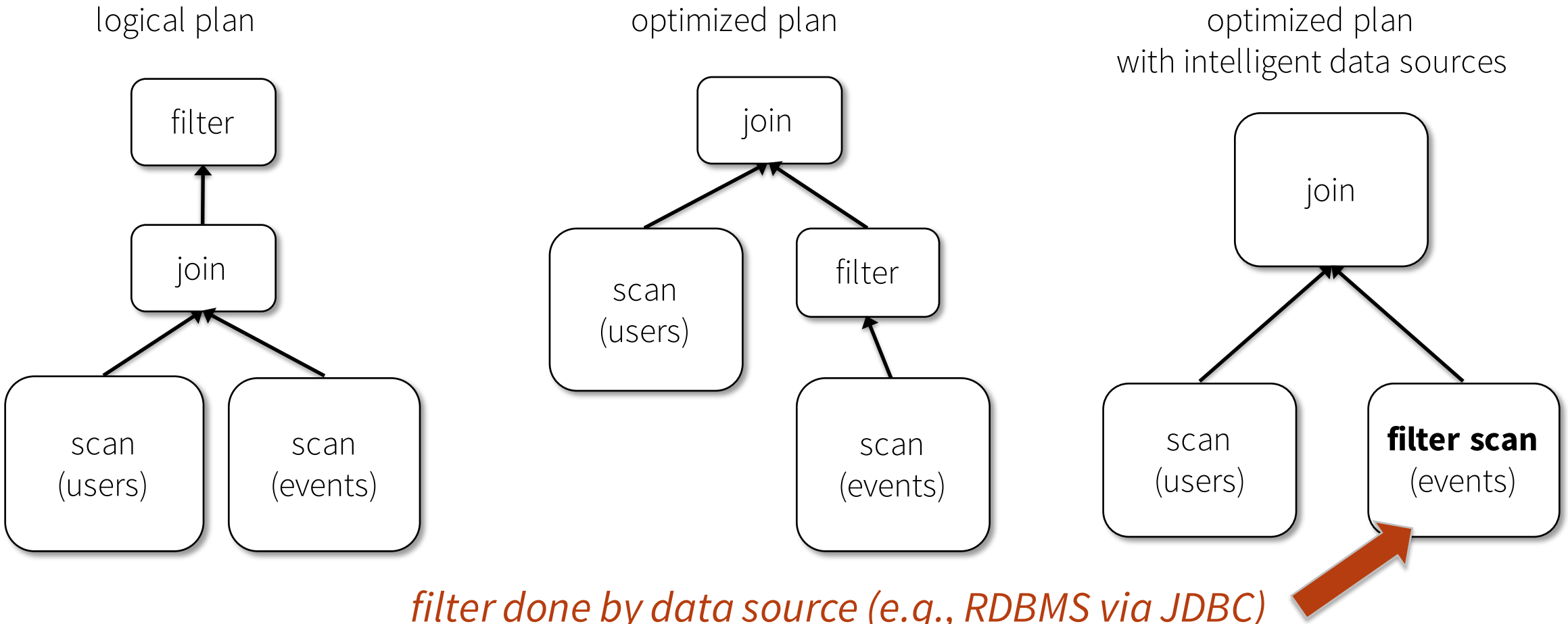
Plan Optimization & Execution

```
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

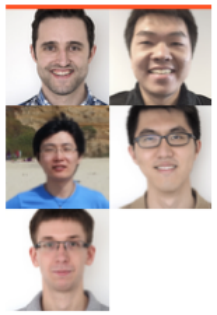
Explain

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

Let's take a look in a notebook.

In your **`Hands On > SQL and DataFrames`** folder, you'll see a notebook called **`Explain`**. Let's open it up.

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.

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

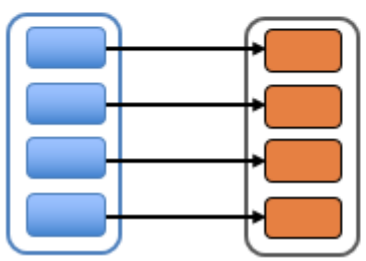
Terminology

- 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

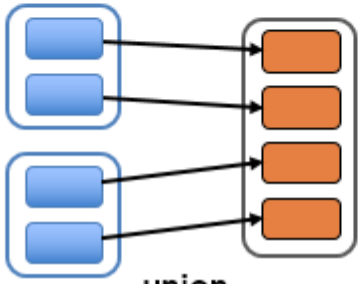
Narrow vs. Wide Dependencies

narrow

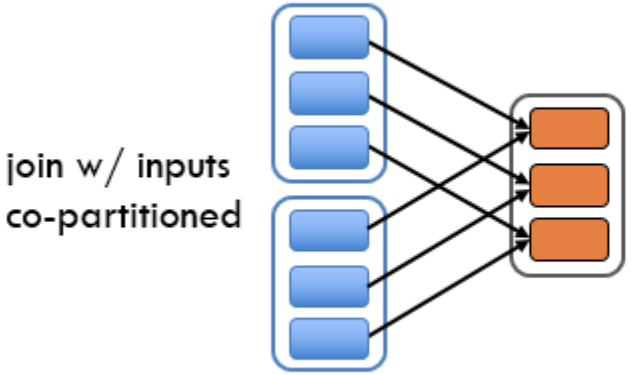
each partition of the parent RDD is used by at most one partition of the child RDD



map, filter



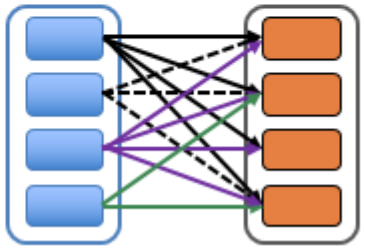
union



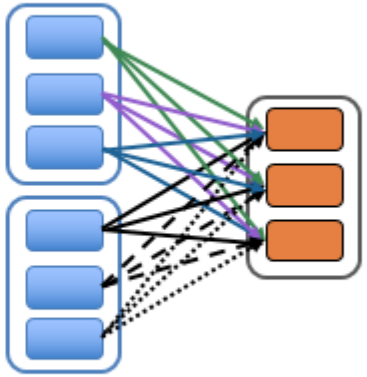
join w/ inputs co-partitioned

wide

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



groupByKey

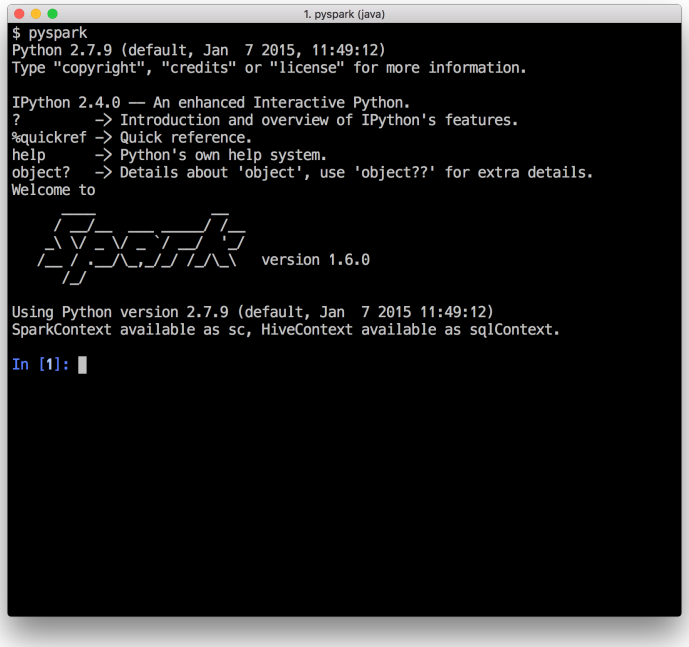


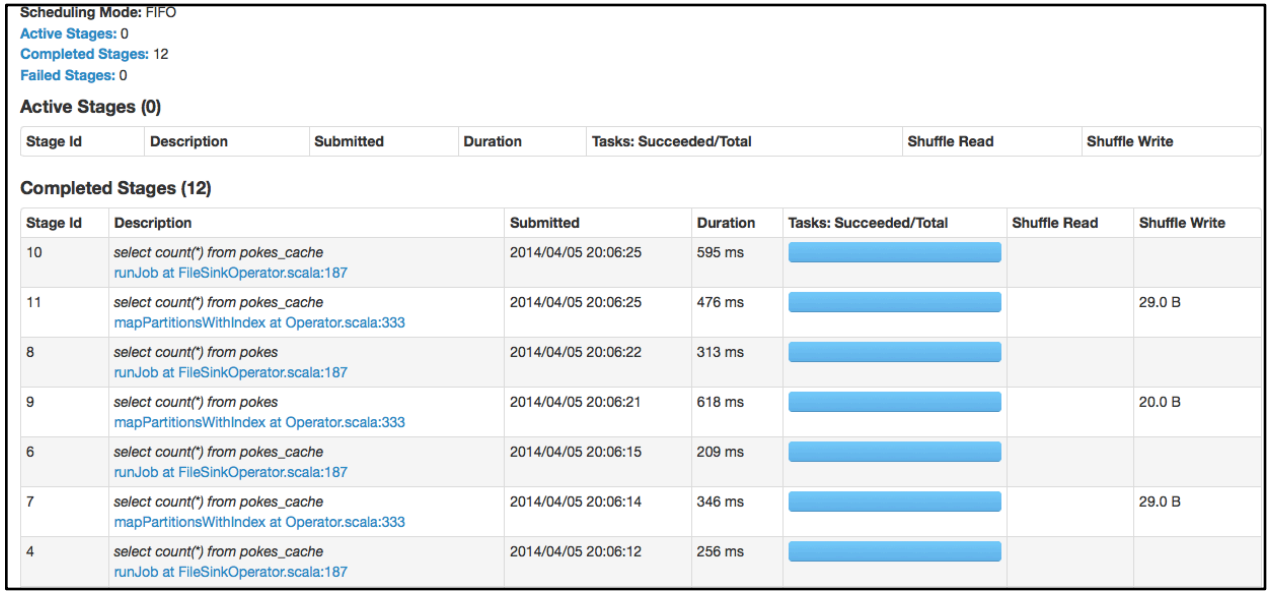
join w/ inputs not co-partitioned

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

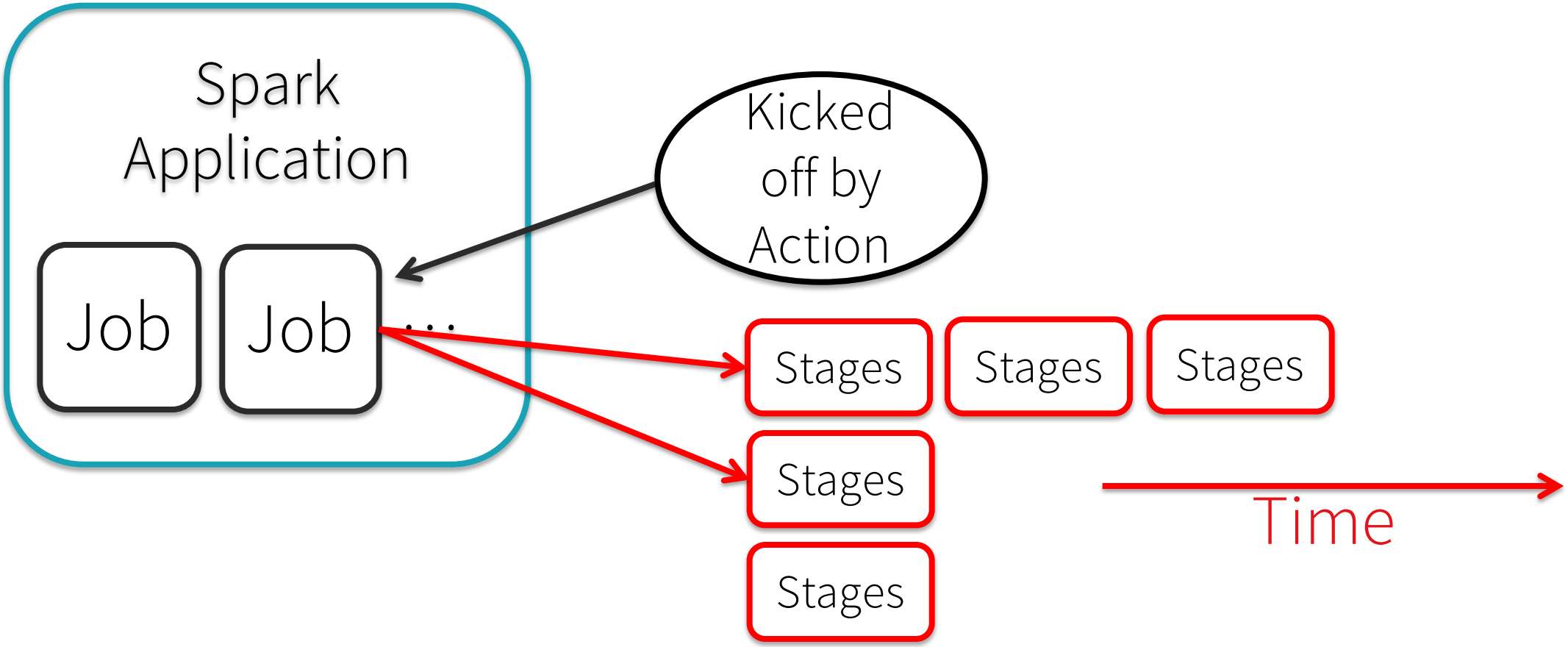
Active Stages (0)

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

Completed Stages (12)

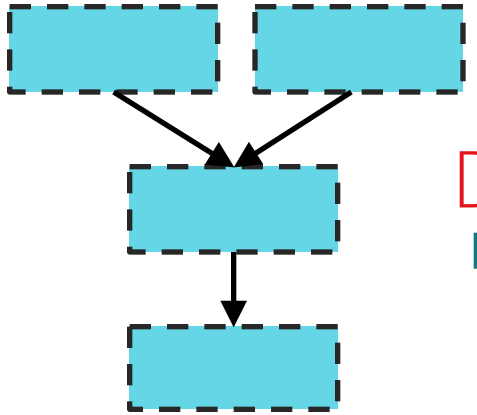
Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Shuffle Read	Shuffle Write
10	*select count(*) from pokes_cache* runJob at FileSinkOperator.scala:187	2014/04/05 20:06:25	595 ms			
11	*select count(*) from pokes_cache* mapPartitionsWithIndex at Operator.scala:333	2014/04/05 20:06:25	476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

Scheduling Process



Scheduling Process

RDD Objects

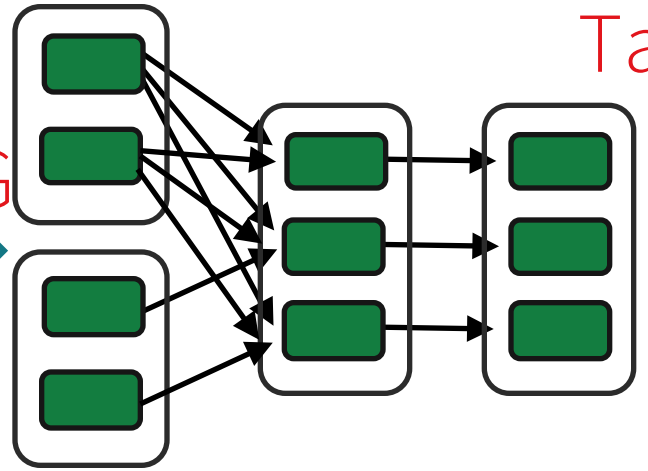


```
Rdd1.join(rdd2)
  .groupBy(…)
  .filter(…)
```

- Build operator DAG

DAG Scheduler

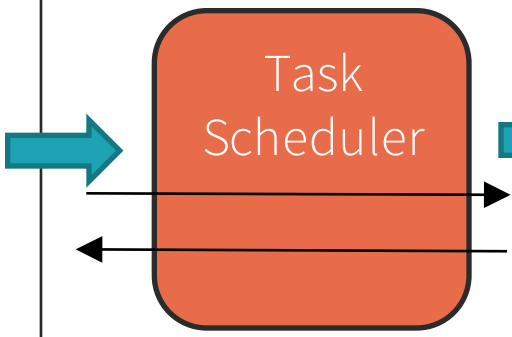
DAG



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

Task Scheduler

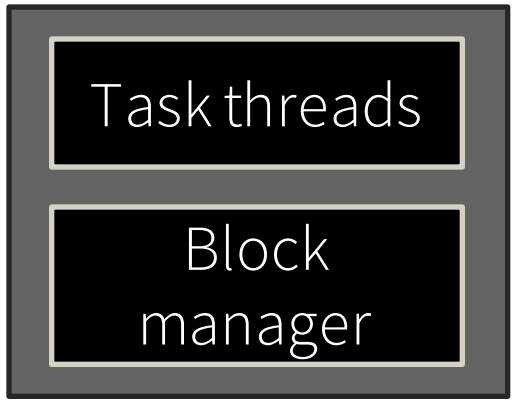
TaskSet



- 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 )
```

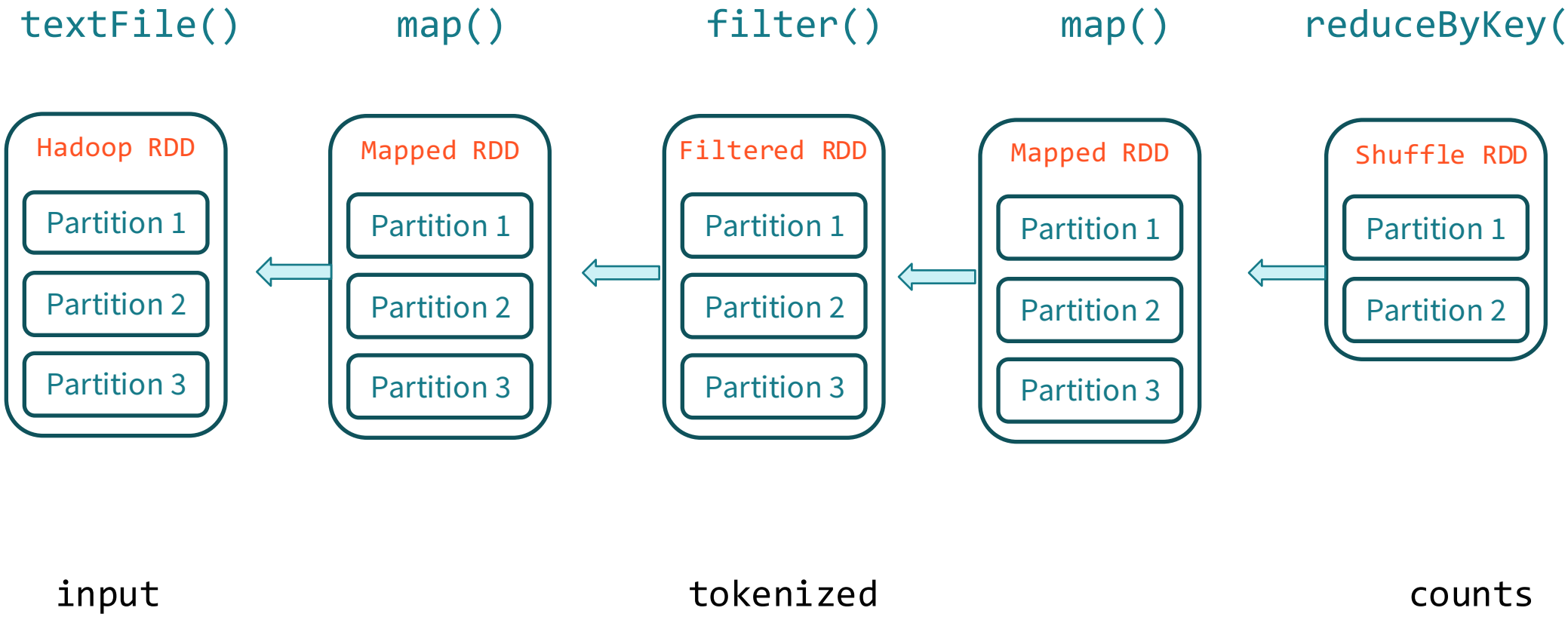
RDD API Example

```
// 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

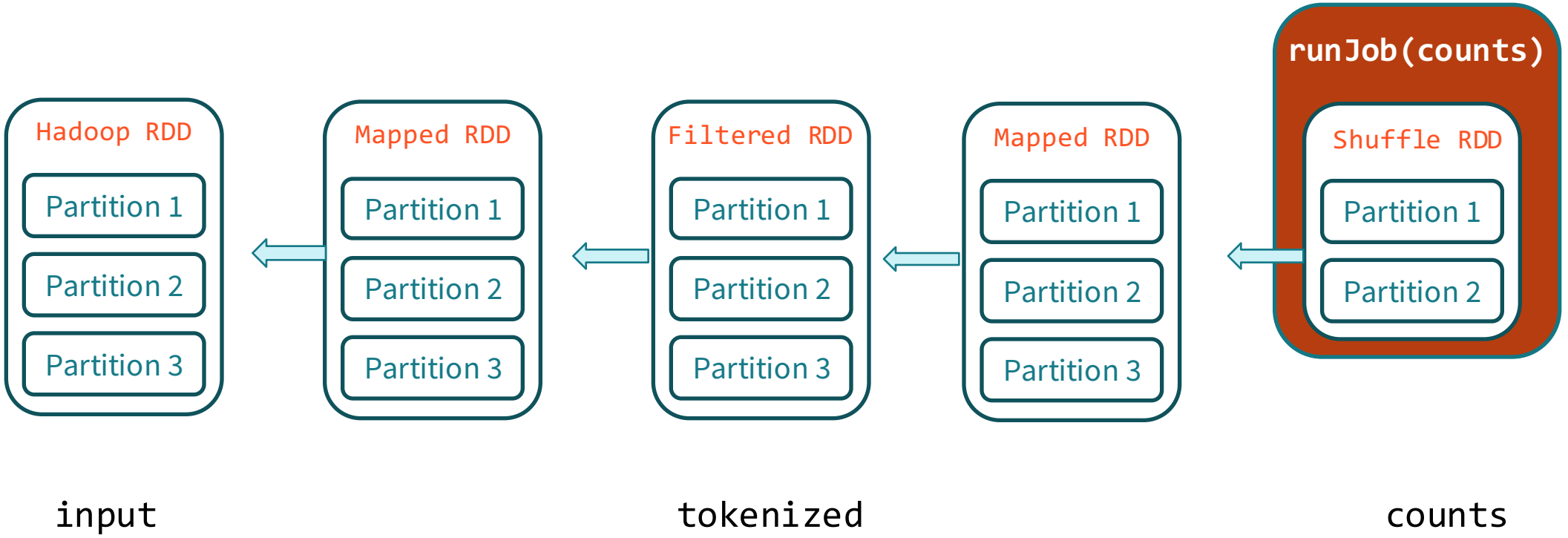
```
sc.textFile().map().filter().map().reduceByKey()
```

DAG View of RDDs



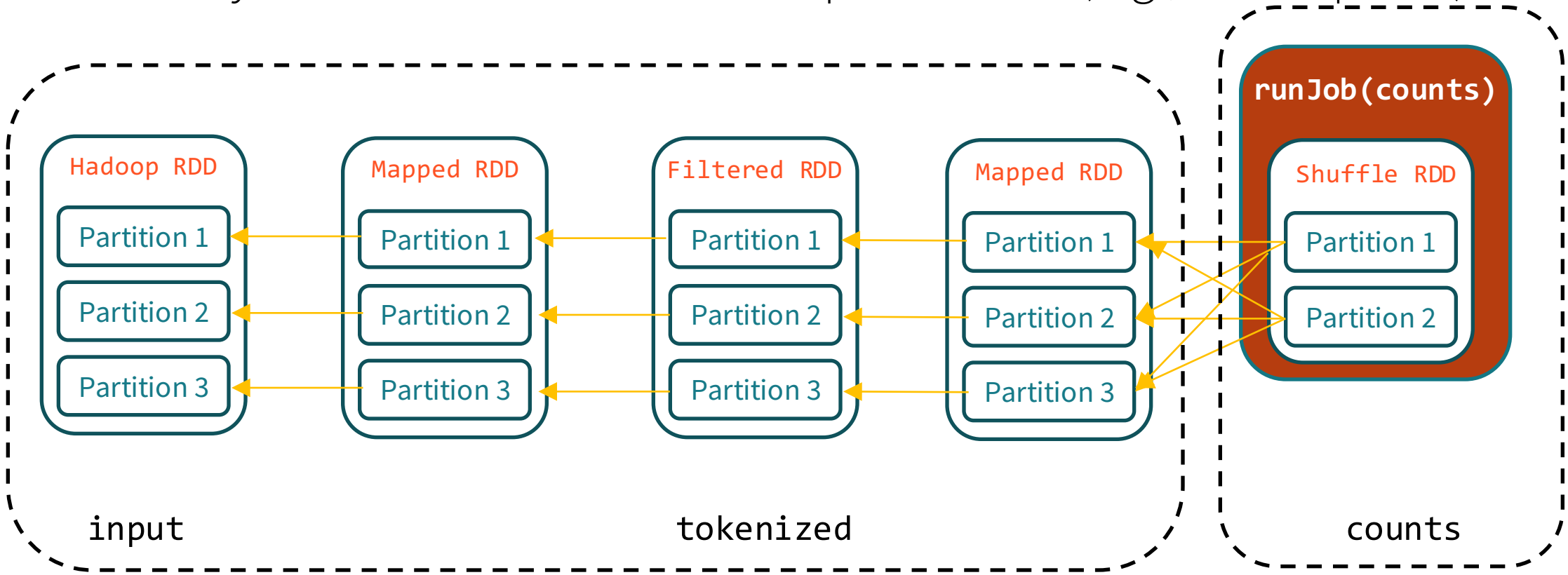
Evaluation of the DAG

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



Evaluation of the DAG

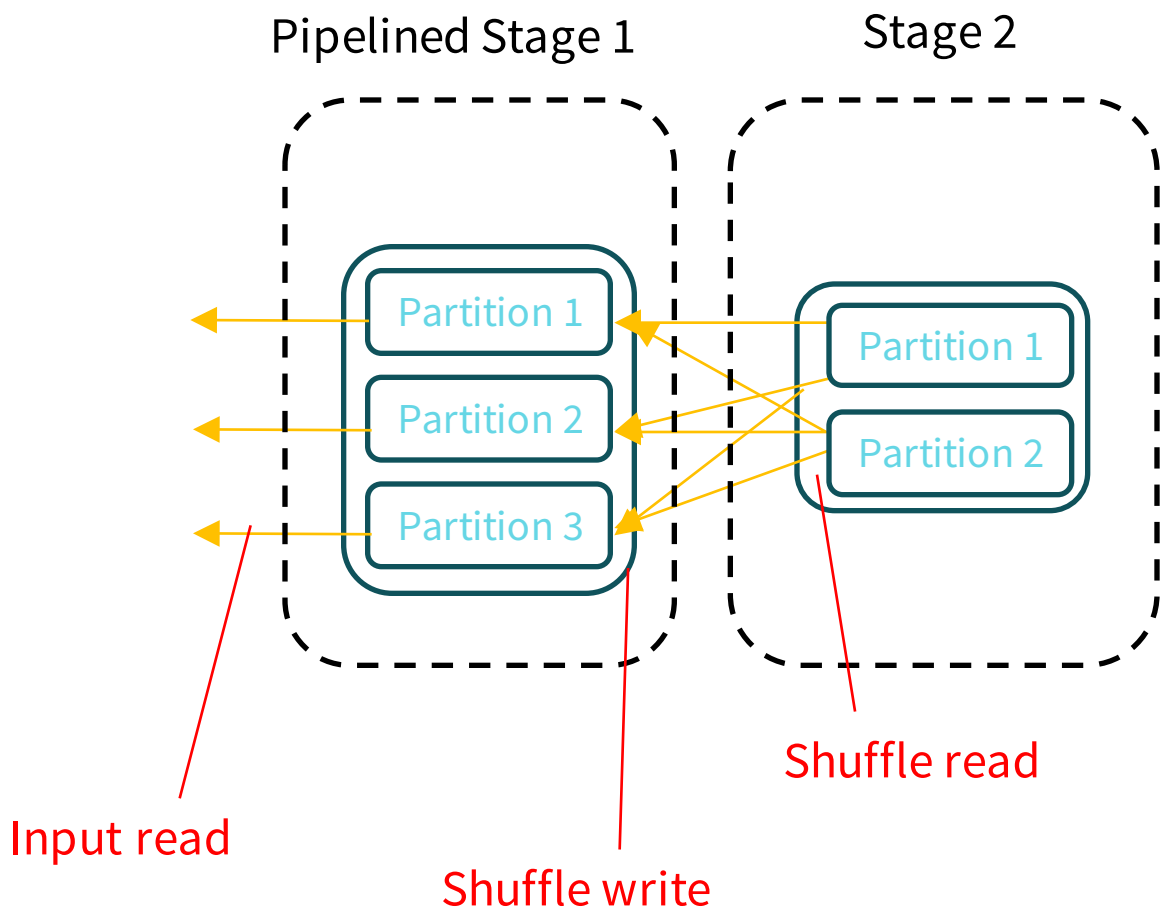
Needs to compute target's parents, parents' parents, etc.
… all the way back to an RDD with no dependencies (e.g., HadoopRDD)



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

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

Spark Streaming

databricks™

Spark Streaming

- Scalable
- High-throughput
- Fault-tolerant

TCP socket

Kafka

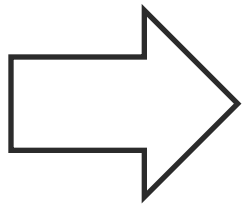
Flume

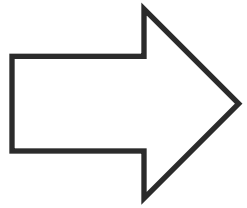
HDFS

S3

Kinesis

Twitter





HDFS / S3

Cassandra

HBase

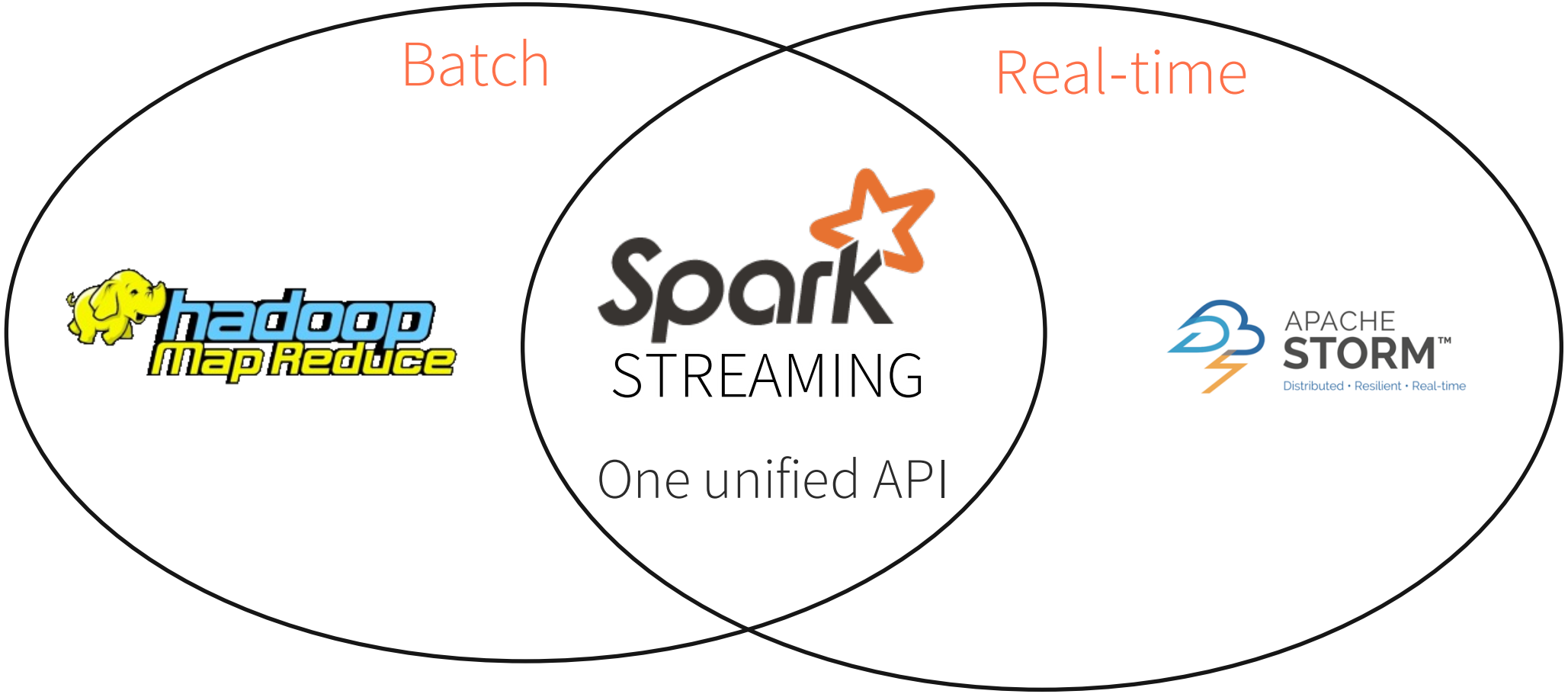
Dashboards

Databases

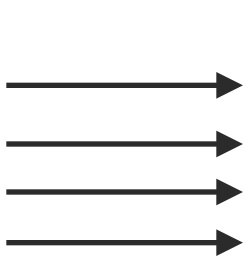
Complex algorithms can be expressed using:

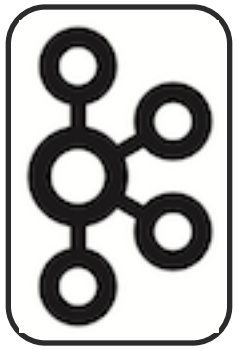
- Spark transformations: `map()`, `reduce()`, `join()`…
- MLlib + GraphX
- SQL

Batch and Real-time



Use Cases





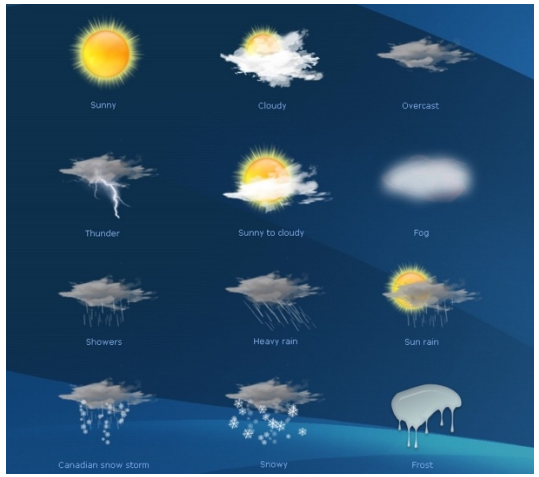
Page views

Kafka for buffering

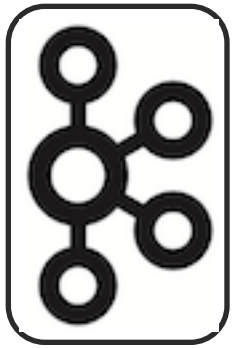
Spark for processing

Use Cases

Smart meter readings



Live weather data

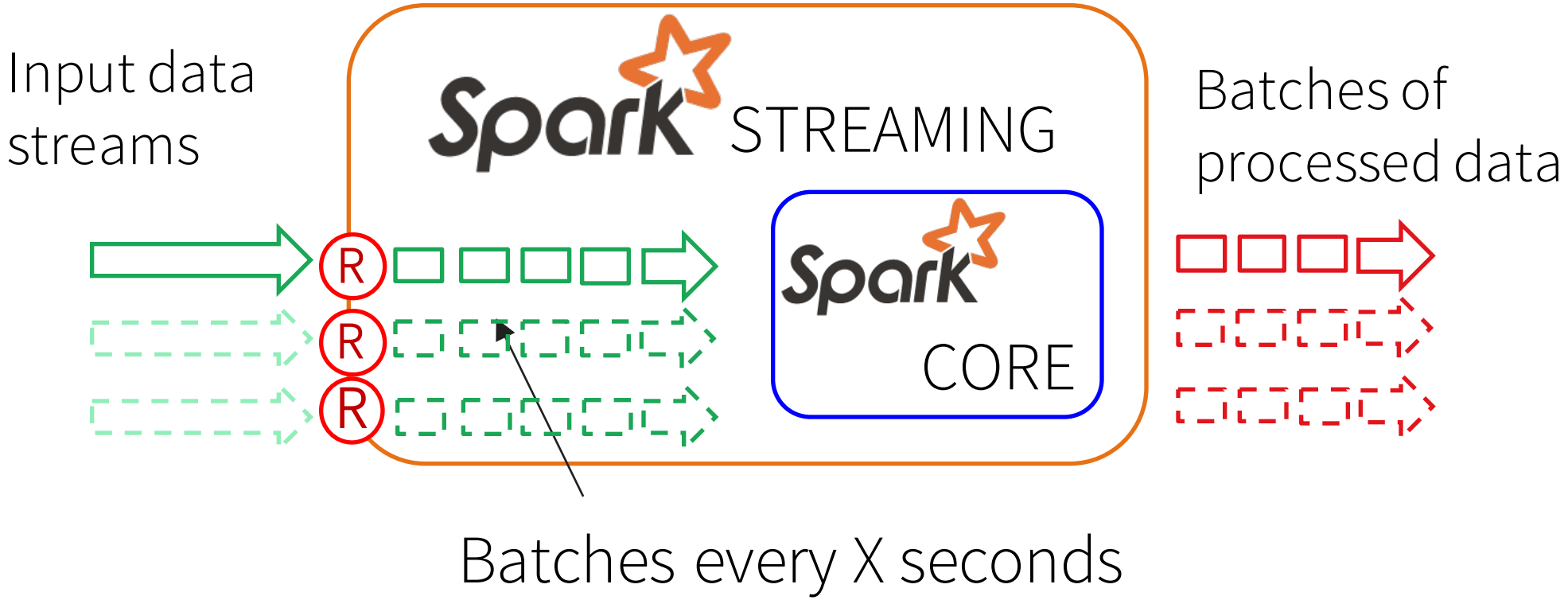


Join 2 live data sources

Spark STREAMING

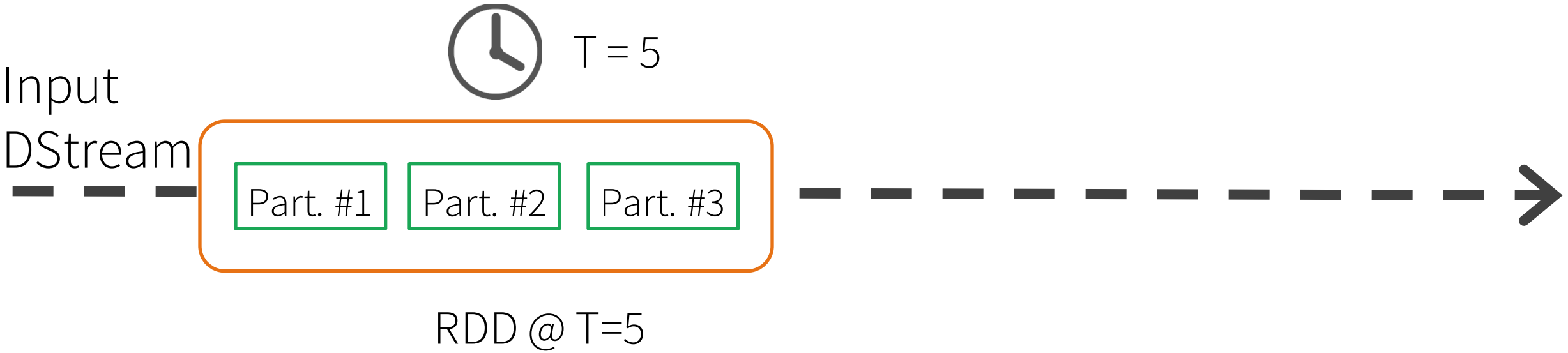
databricks

Data Model


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

DStream (Discretized Stream)

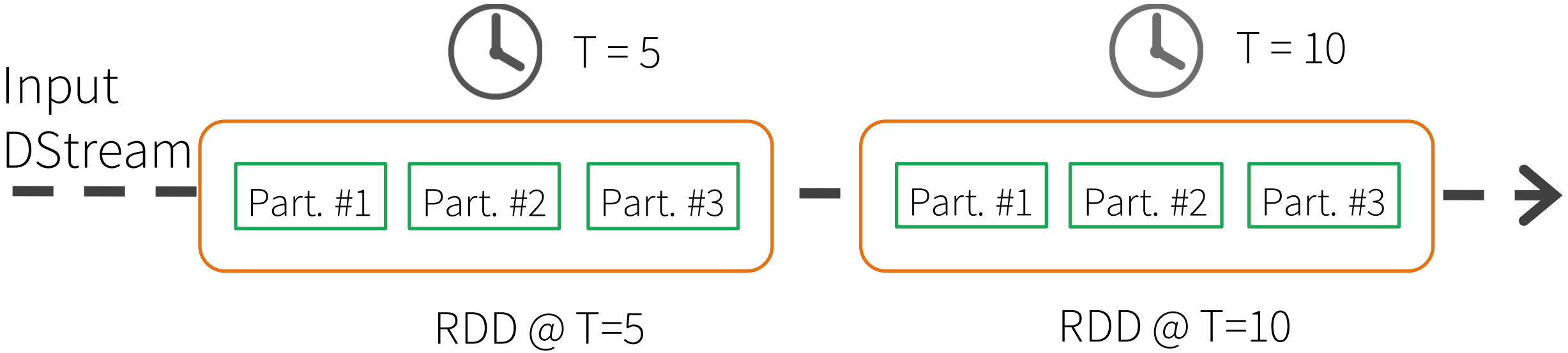
Batch interval = 5 seconds



One RDD is created every 5 seconds

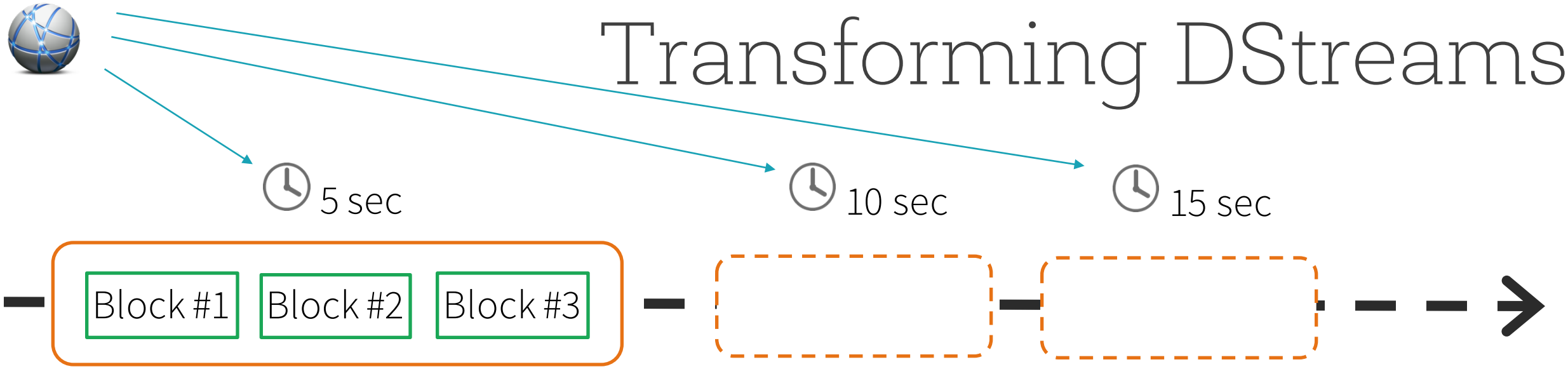
DStream (Discretized Stream)

Batch interval = 5 seconds

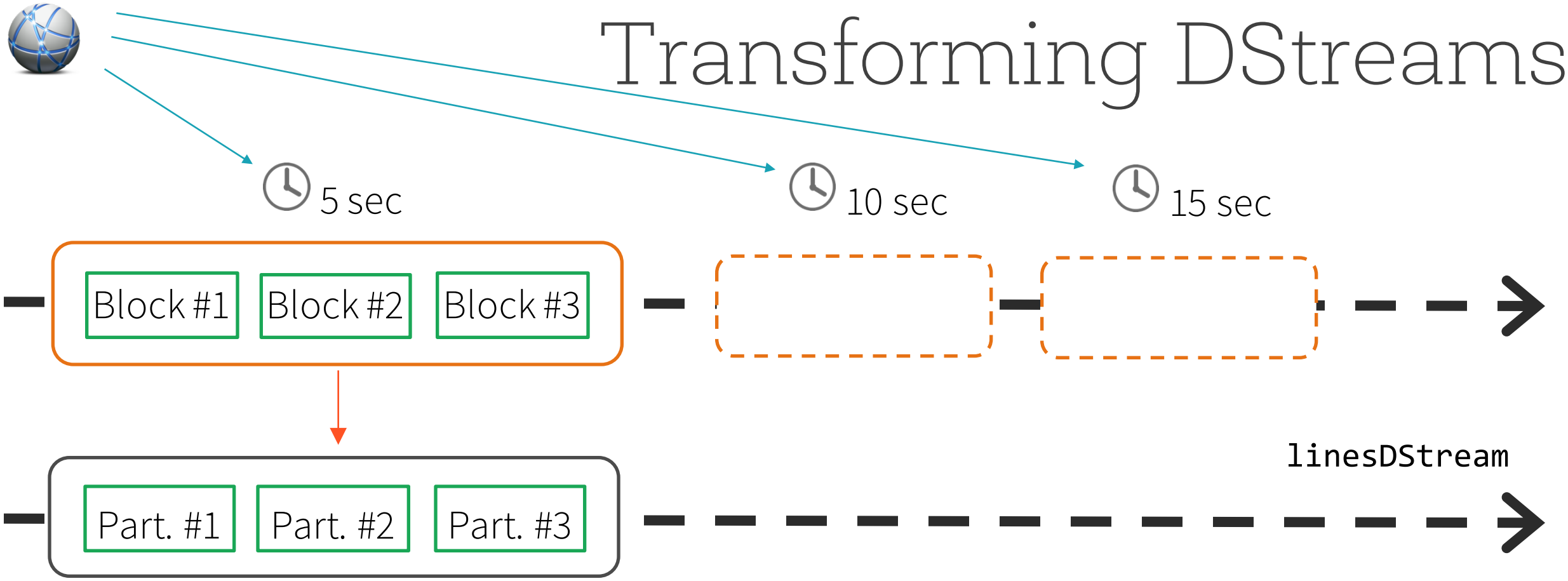


One RDD is created every 5 seconds

Transforming DStreams



Transforming DStreams



Transforming DStreams

5 sec

10 sec

15 sec

Block #1 | Block #2 | Block #3

Part. #1 | Part. #2 | Part. #3

`linesDStream`

`flatMap()`

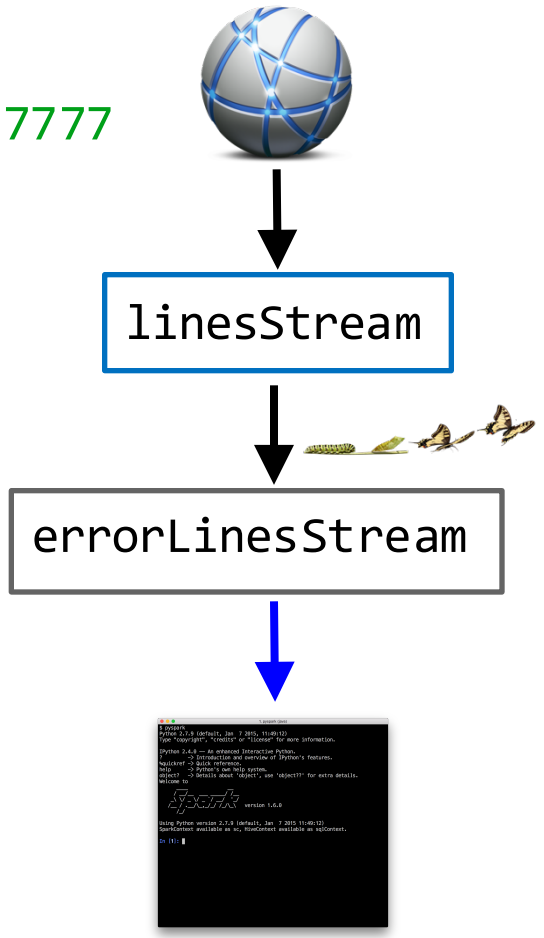
Part. #1 | Part. #2 | Part. #3

`wordsDStream`

databricks

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
```

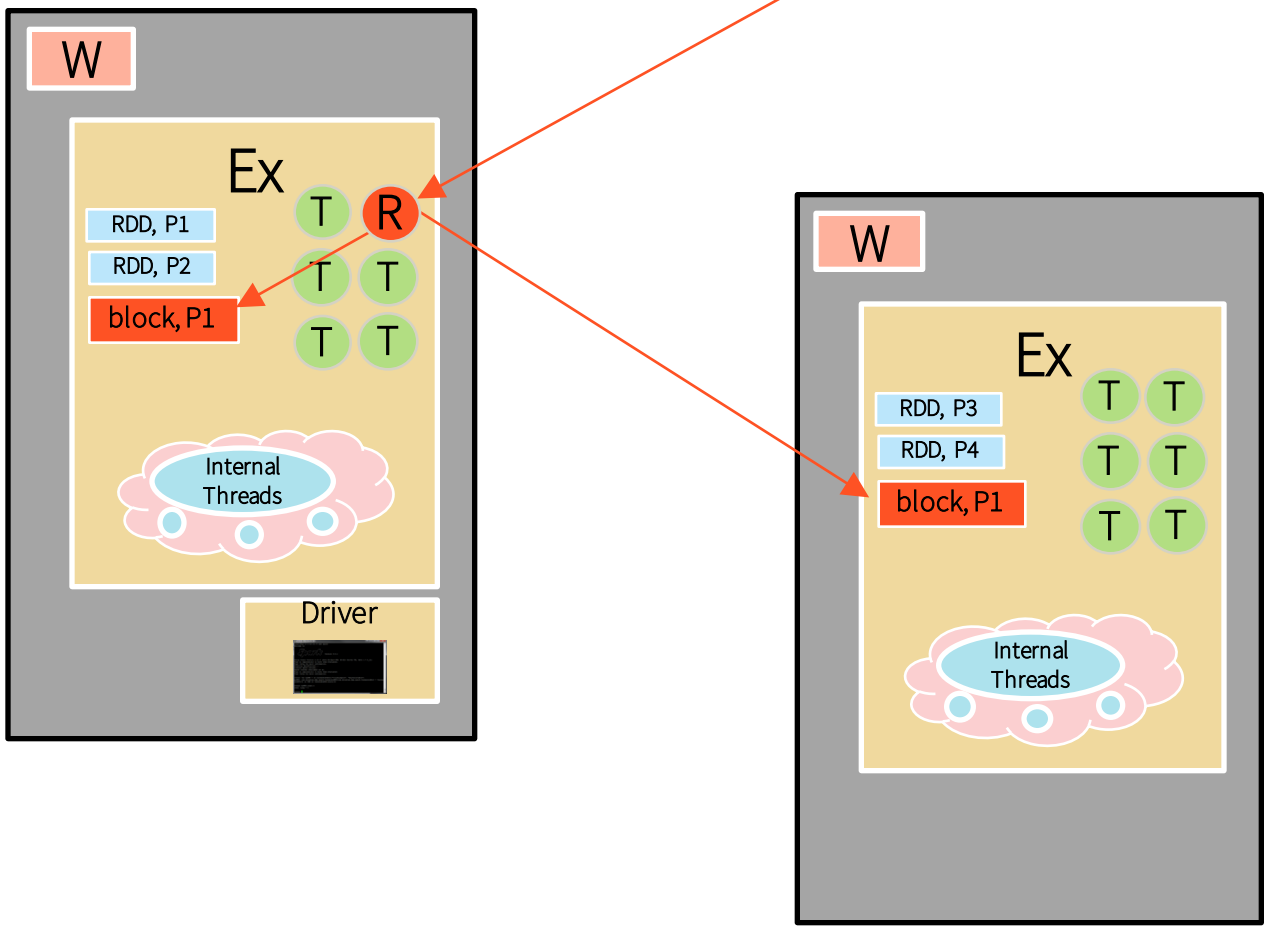

```
$   spark-submit --class com.examples.Scala.StreamingLog \
    $ASSEMBLY_JAR local[4]
. . .
-------------------------
Time: 2015-05-26 15:25:21
-------------------------
there was an error

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

databricks

Example

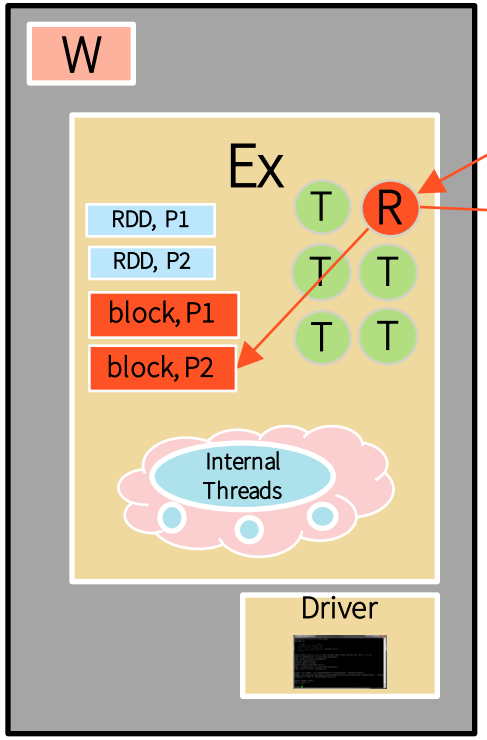
Batch interval = 600 ms

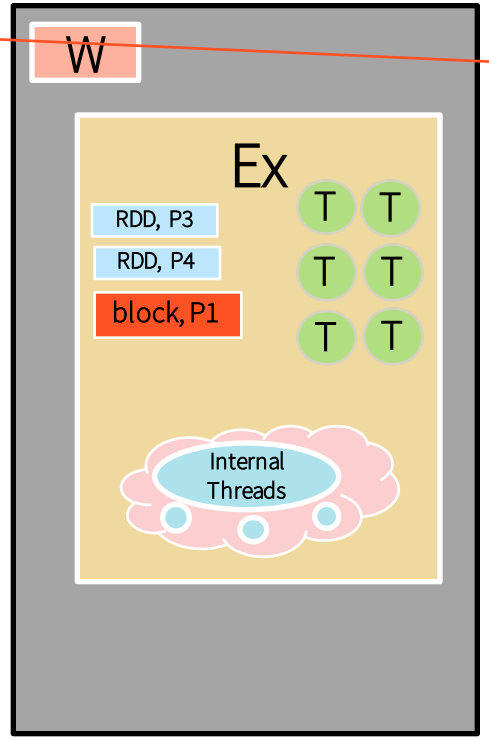
localhost 7777

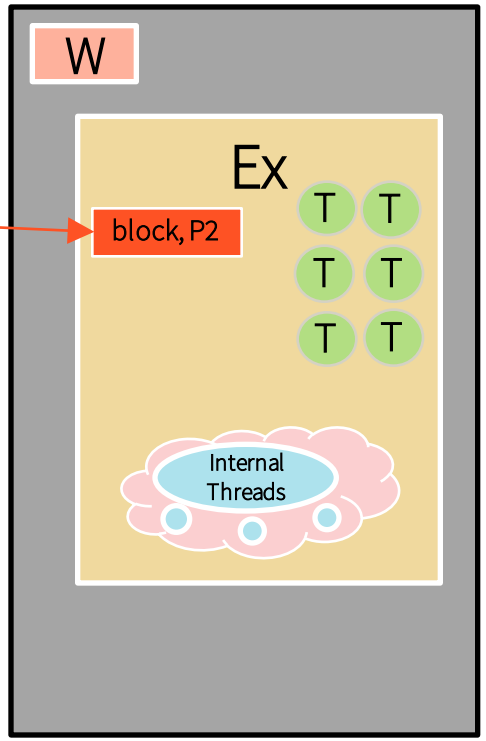


Batch interval = 600 ms

Example





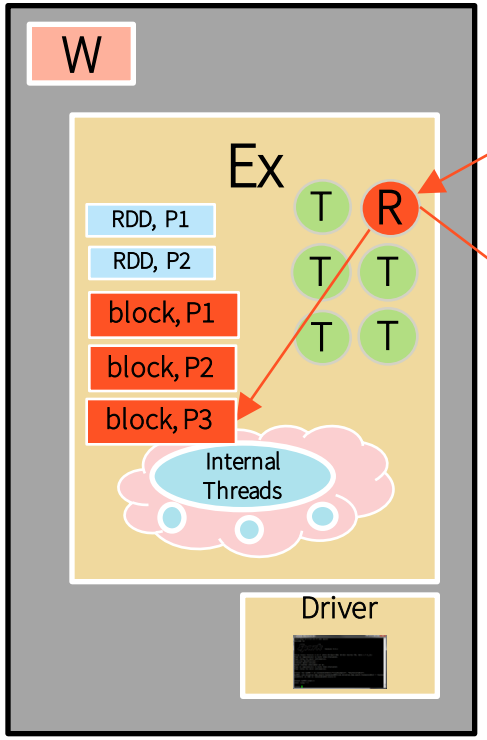


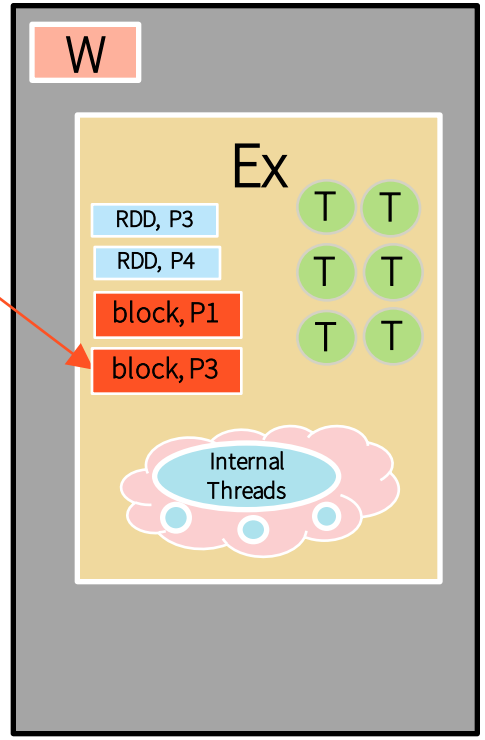
200 ms later

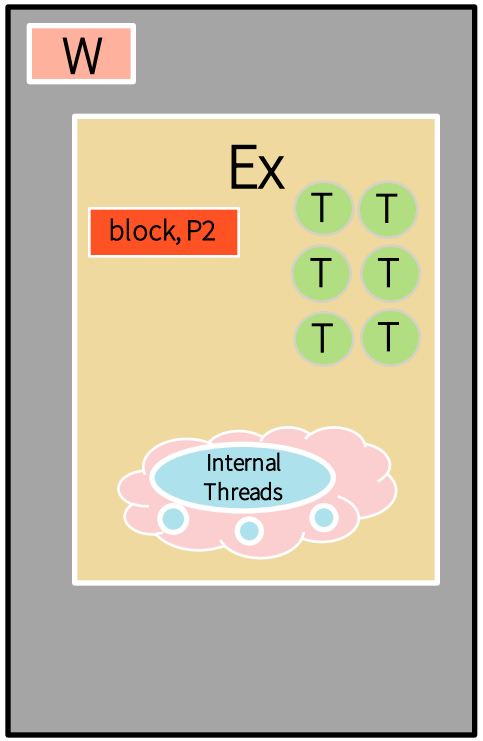
databricks

Batch interval = 600 ms

Example



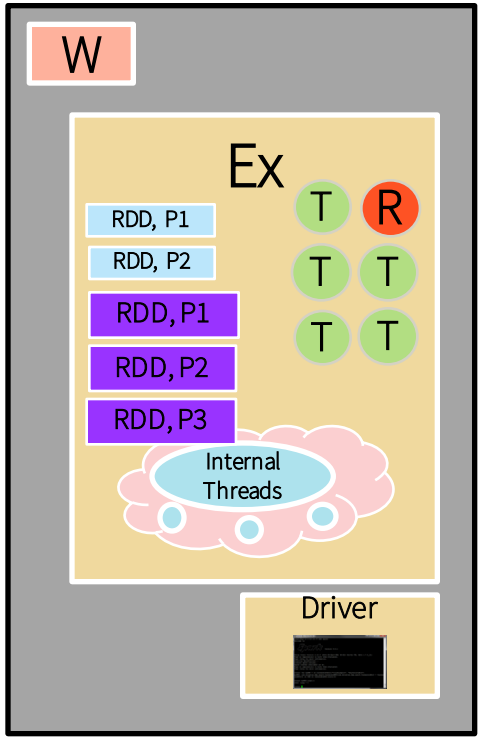


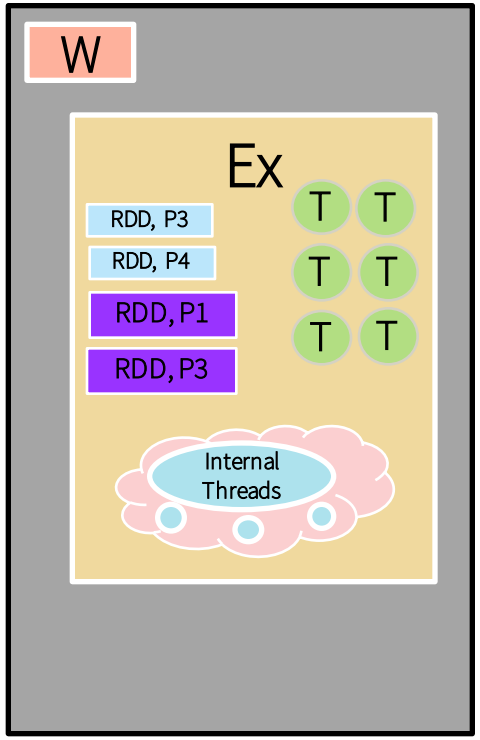


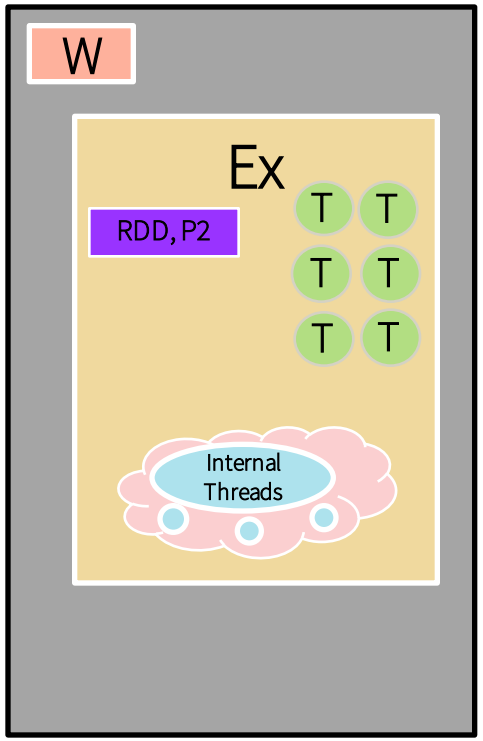
200 ms later

Batch interval = 600 ms

Example

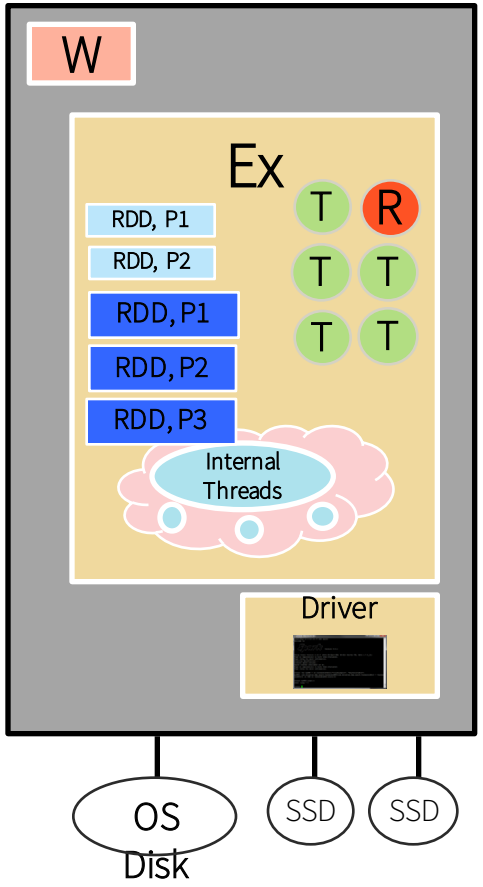


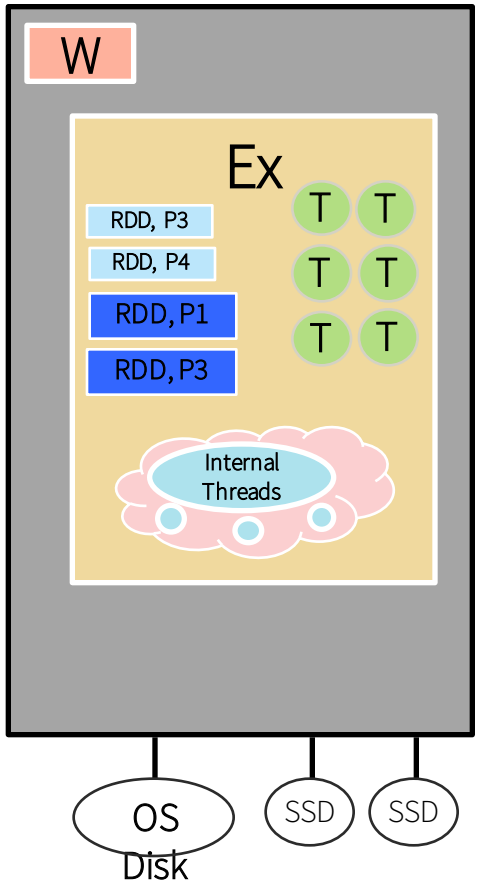


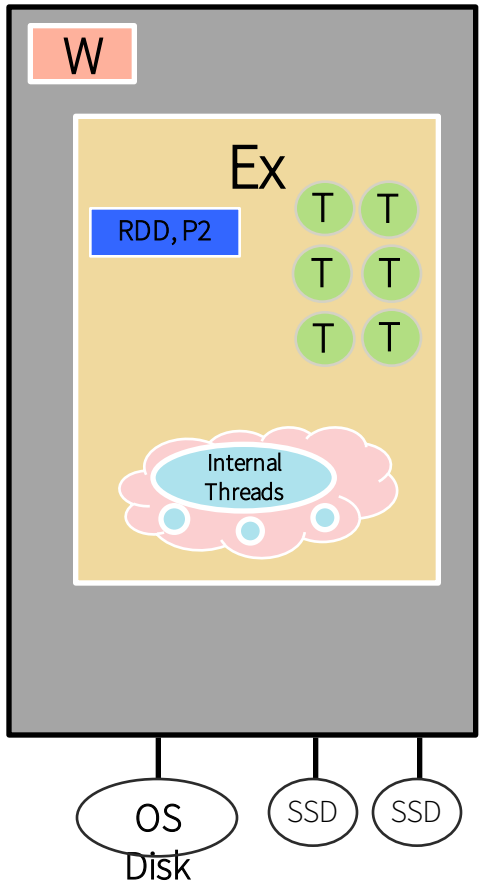


Batch interval = 600 ms

Example



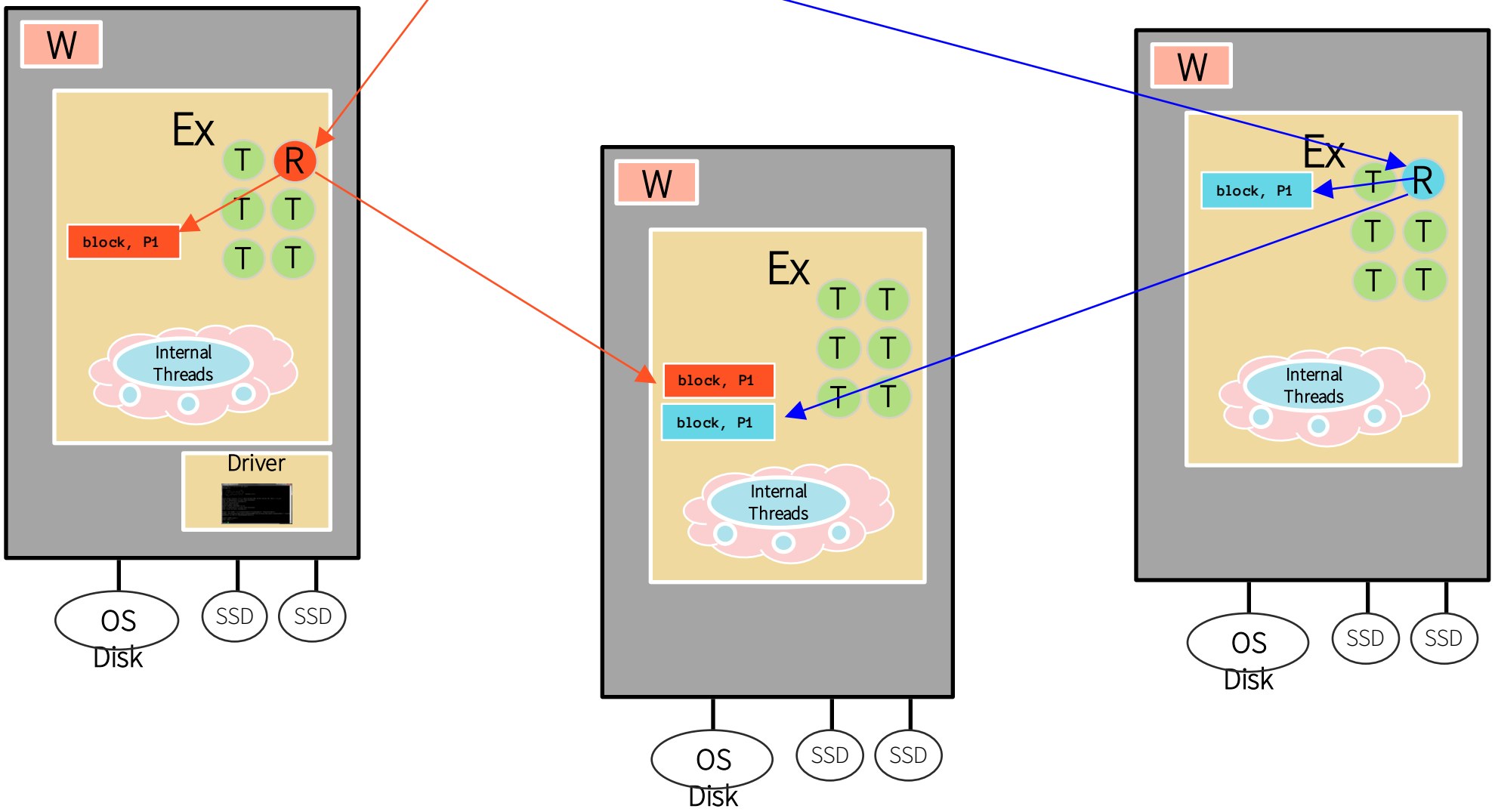




databricks

Batch interval = 600 ms

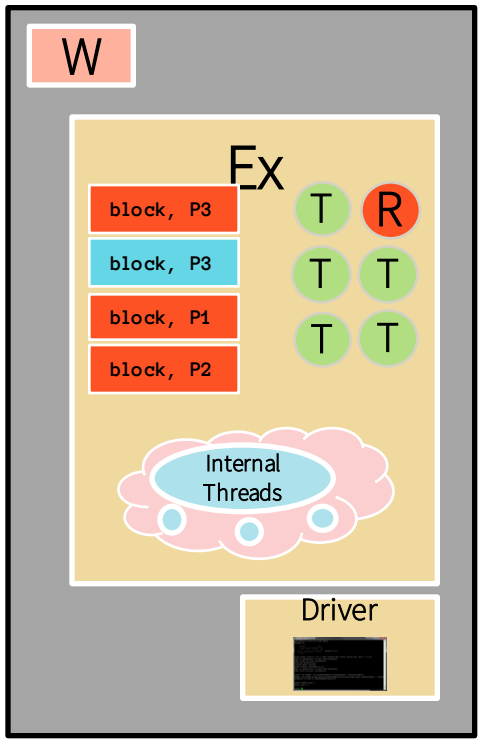
2 Input DStreams

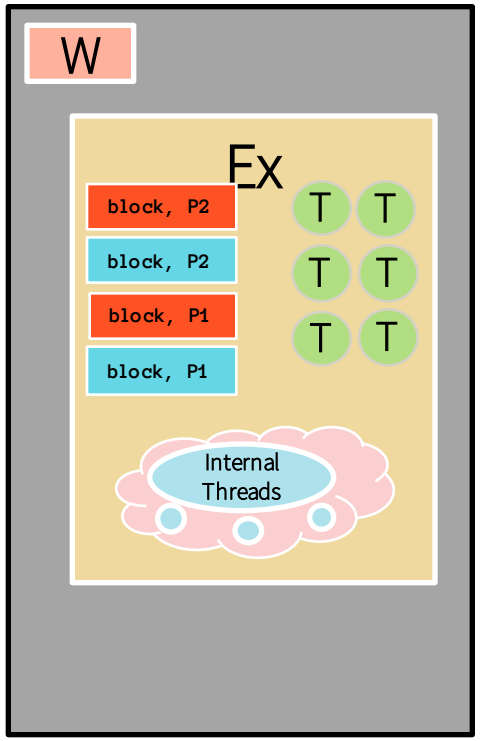


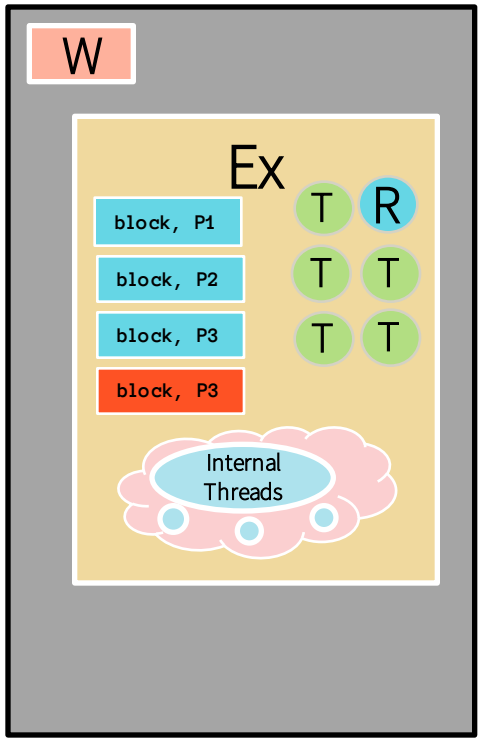
databricks

Batch interval = 600 ms

2 Input DStreams



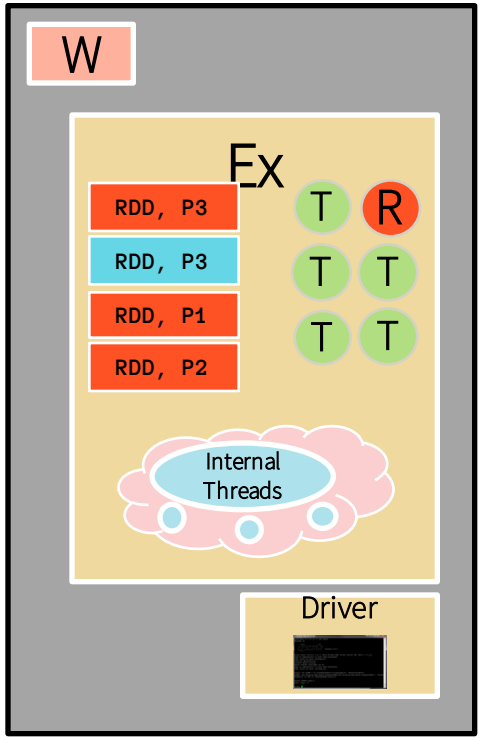




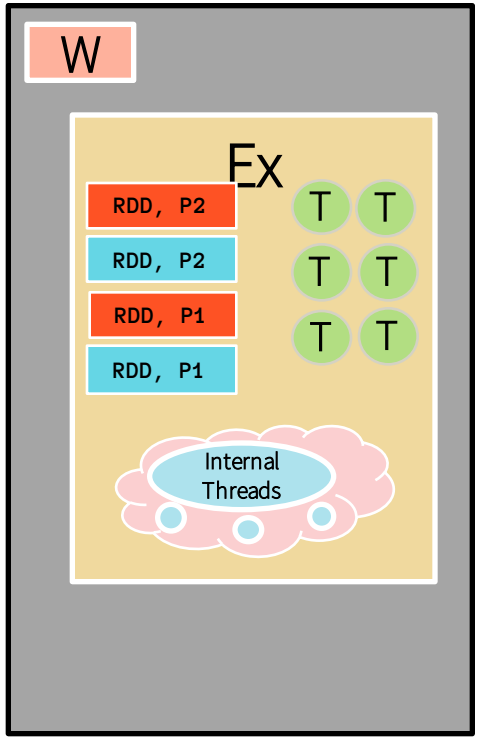
databricks

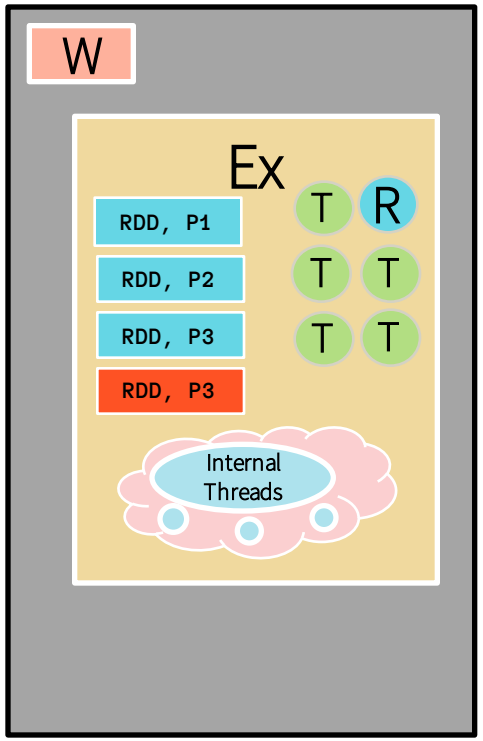
Batch interval = 600 ms

2 Input DStreams



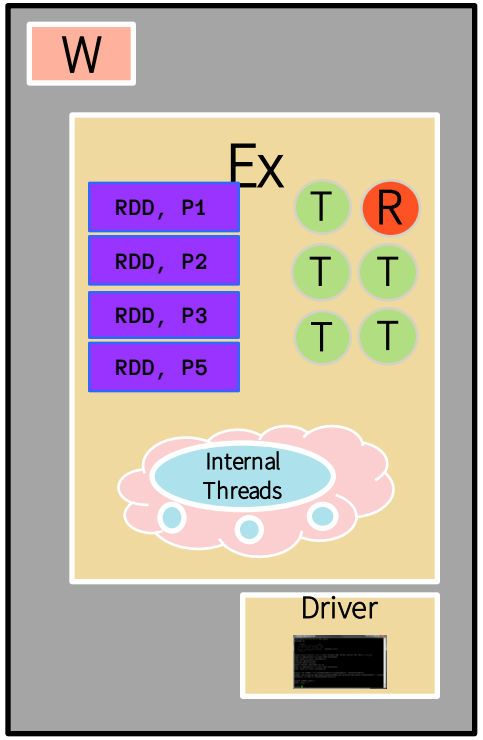
Materialize!



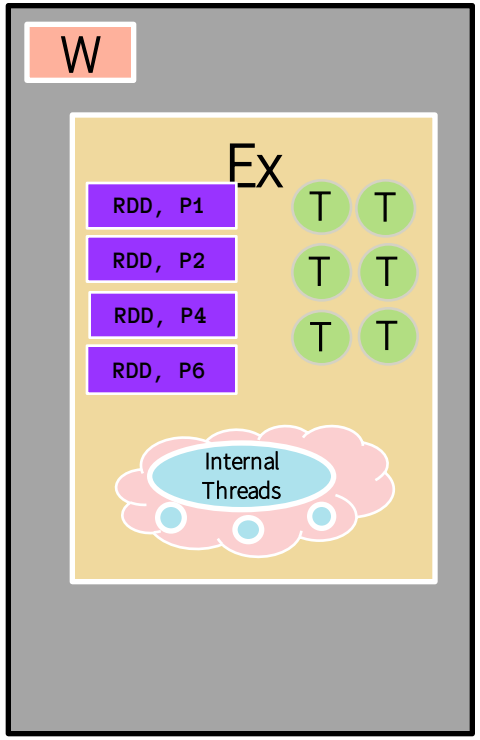


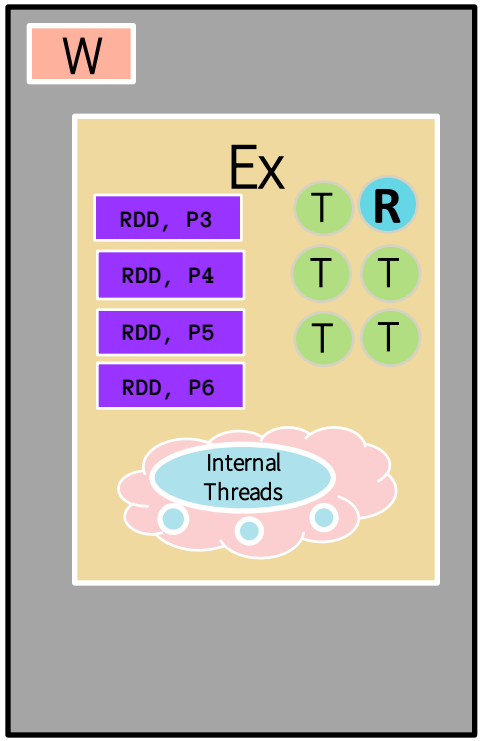
Batch interval = 600 ms

2 Input DStreams



Union!





DStream-Dstream Unions

Stream–stream Union

```
val numStreams = 5

val kafkaStreams = (1 to numStreams).map { i => KafkaUtils.createStream(...) }

val unifiedStream = streamingContext.union(kafkaStreams)

unifiedStream.print()
```

DStream-Dstream Joins

Stream–stream Join

```
val stream1: DStream[String, String] = ...
val stream2: DStream[String, String] = ...

val joinedStream = stream1.join(stream2)
```

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, in your **`Hands On`** folder, you should see a notebook entitled

`Streaming Wikipedia Anonymous Edits`

Let's walk through it together. Then, you'll have some time to play with it on your own.

End of Spark Streaming

Spark Machine Learning

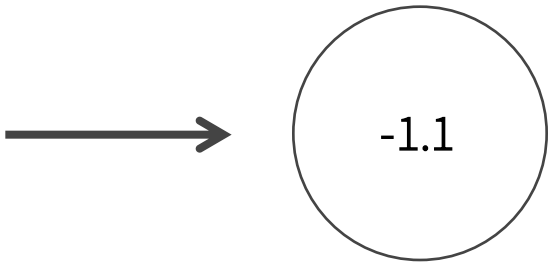
Example: Log prioritization

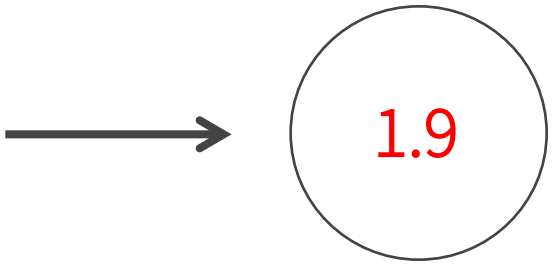
Data: Test logs

```
Running test: pyspark/conf.py
Spark assembly has been built
with Hive, including
Datanucleus jars on classpath
14/12/15 18:36:12 WARN Utils:
Your hostname,
```

```
Running test:
pyspark/broadcast.py
Spark assembly has been built
with Hive, including
Datanucleus jars on classpath
14/12/15 18:36:30 ERROR Aliens
attacked the
```

Goal: Prioritize logs to investigate





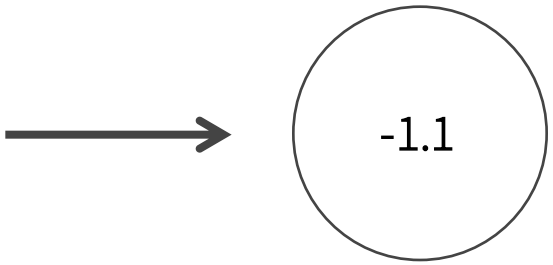
Example: Log prioritization

Data: Test logs *Instance*

```
Running test: pyspark/conf.py
Spark assembly has been built
with Hive, including
Datanucleus jars on classpath
14/12/15 18:36:12 WARN Utils:
Your hostname,
```

Goal: Prioritize logs to investigate

Label



How can we learn?

- Choose a model
- Get training data
- Run a learning algorithm

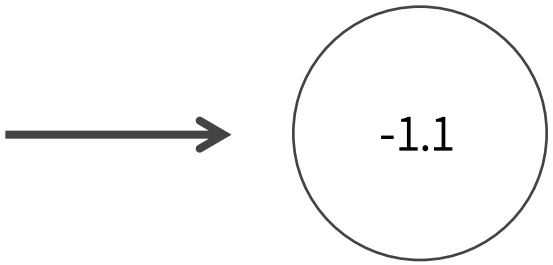
A model is a function f: x → y

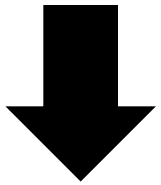
(prediction)

Instance x

```
Running test: pyspark/conf.py
Spark assembly has been built
with Hive, including
Datanucleus jars on classpath
14/12/15 18:36:12 WARN Utils:
Your hostname,
```

Label y





Convert to features,
e.g., word counts

Running	test	Spark	aliens	...
43	67	110	0	

Feature vector x

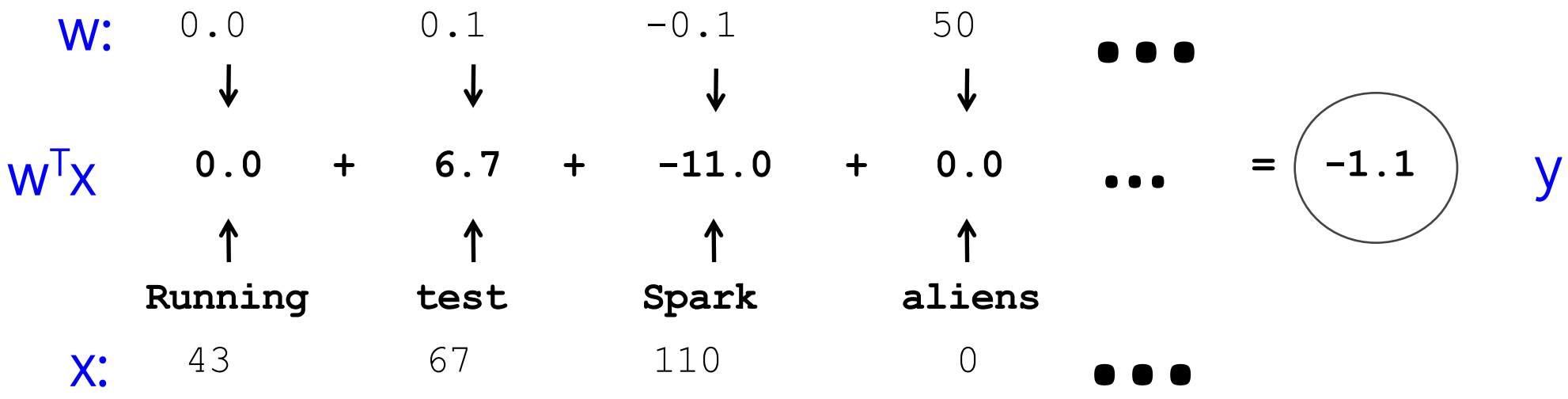
`mllib.linalg.Vector`

A model is a function f: x → y

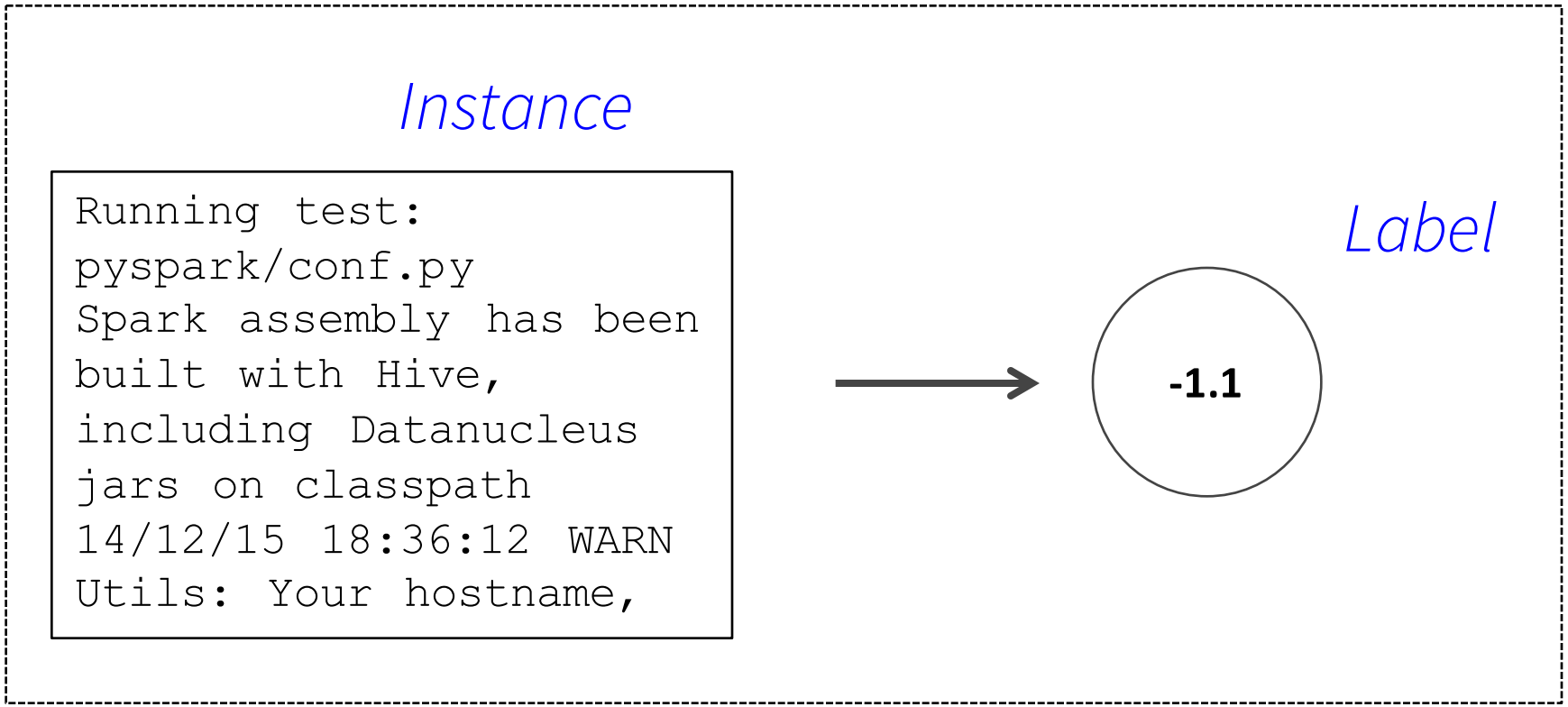
`LinearRegression`

Learning
= choosing parameters w

Our model: *Parameter vector* **w**

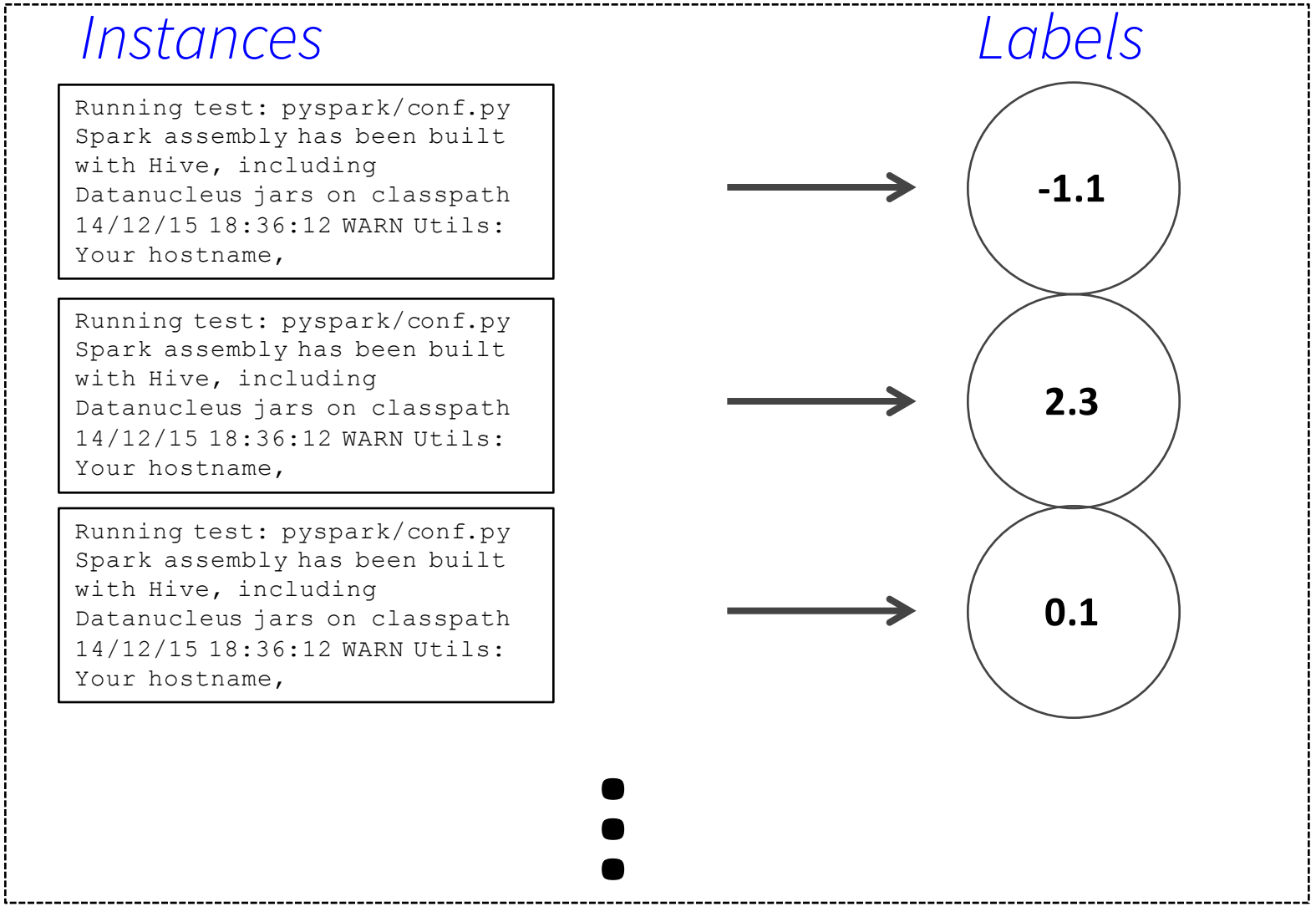


Data for learning




```
LabeledPoint(features: Vector, label: Double)
```

Data for learning



Dataset

`RDD[LabeledPoint]`

ML algorithms

Recall:

A model is a function: features → label

```
LinearRegressionModel.predict(features: Vector): Double
```

A training dataset is a set of (features, label) pairs

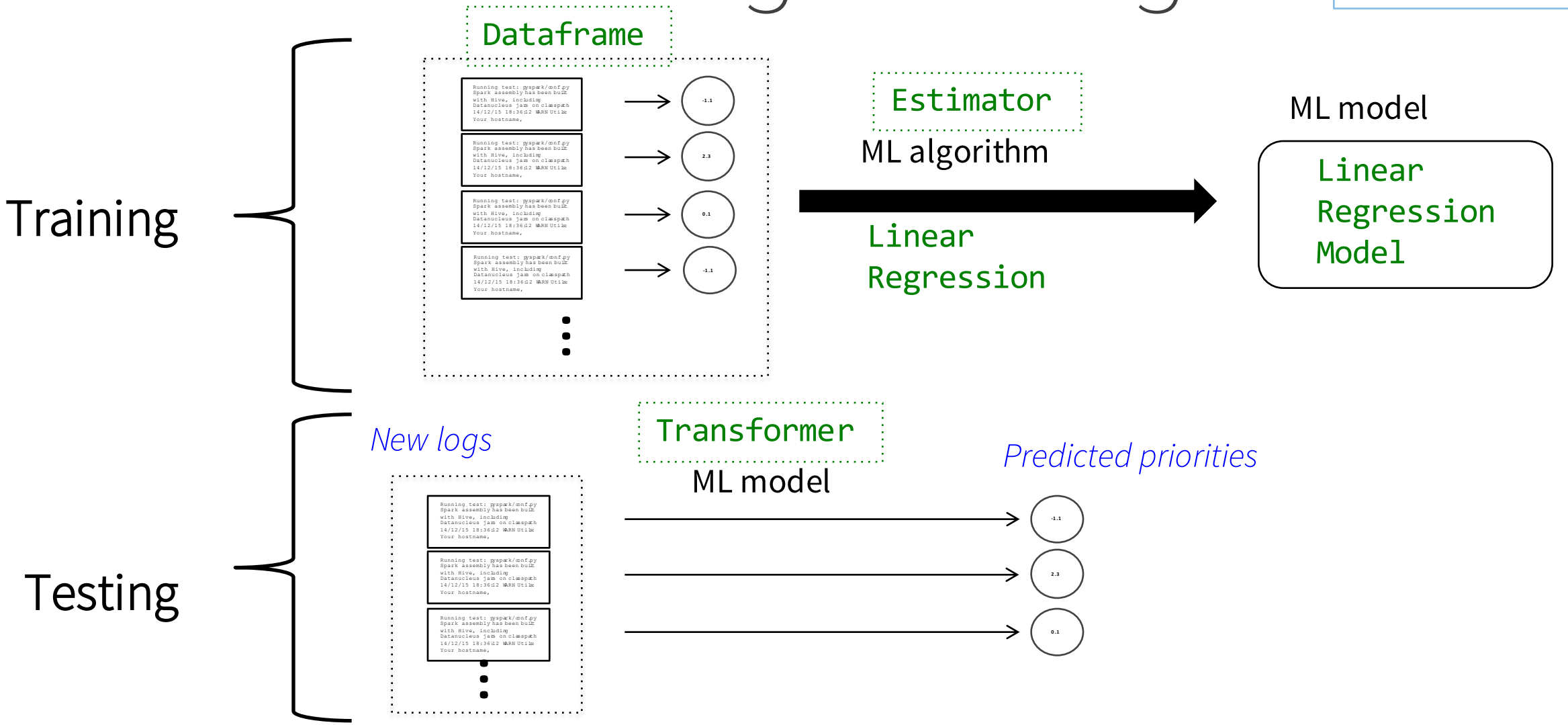
```
RDD[LabeledPoint]
```

An ML algorithm is a function: dataset → model

```
LinearRegression.train(
    data: RDD[LabeledPoint]): LinearRegressionModel
```

Workflow: training + testing

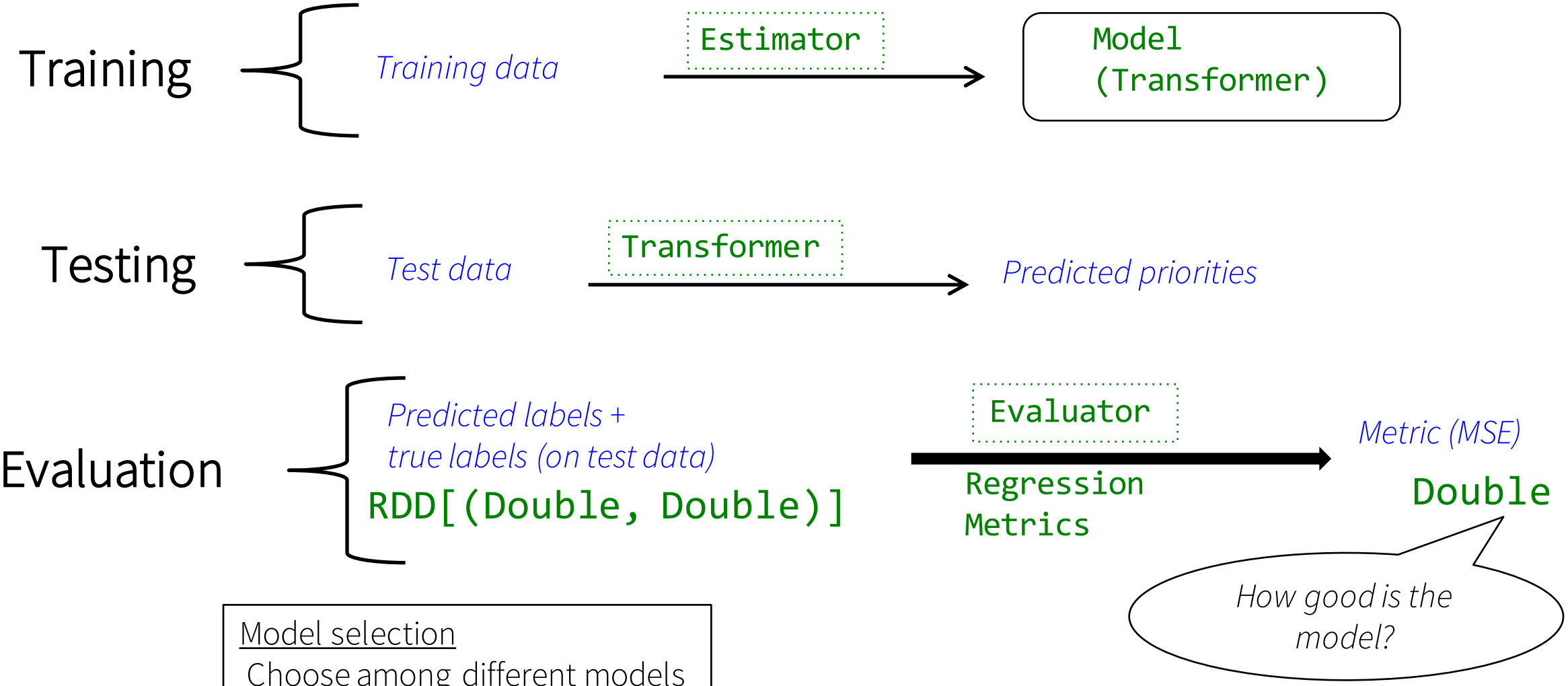
New API ("Pipelines")



databricks

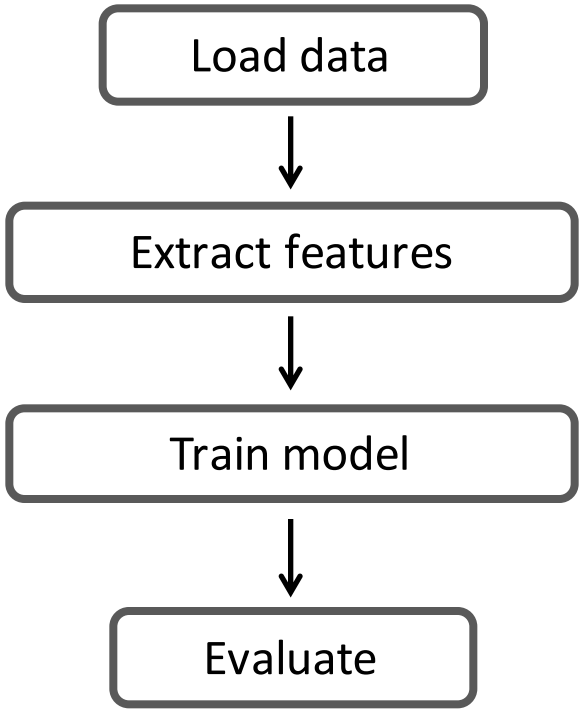
+ Evaluation

New API ("Pipelines")

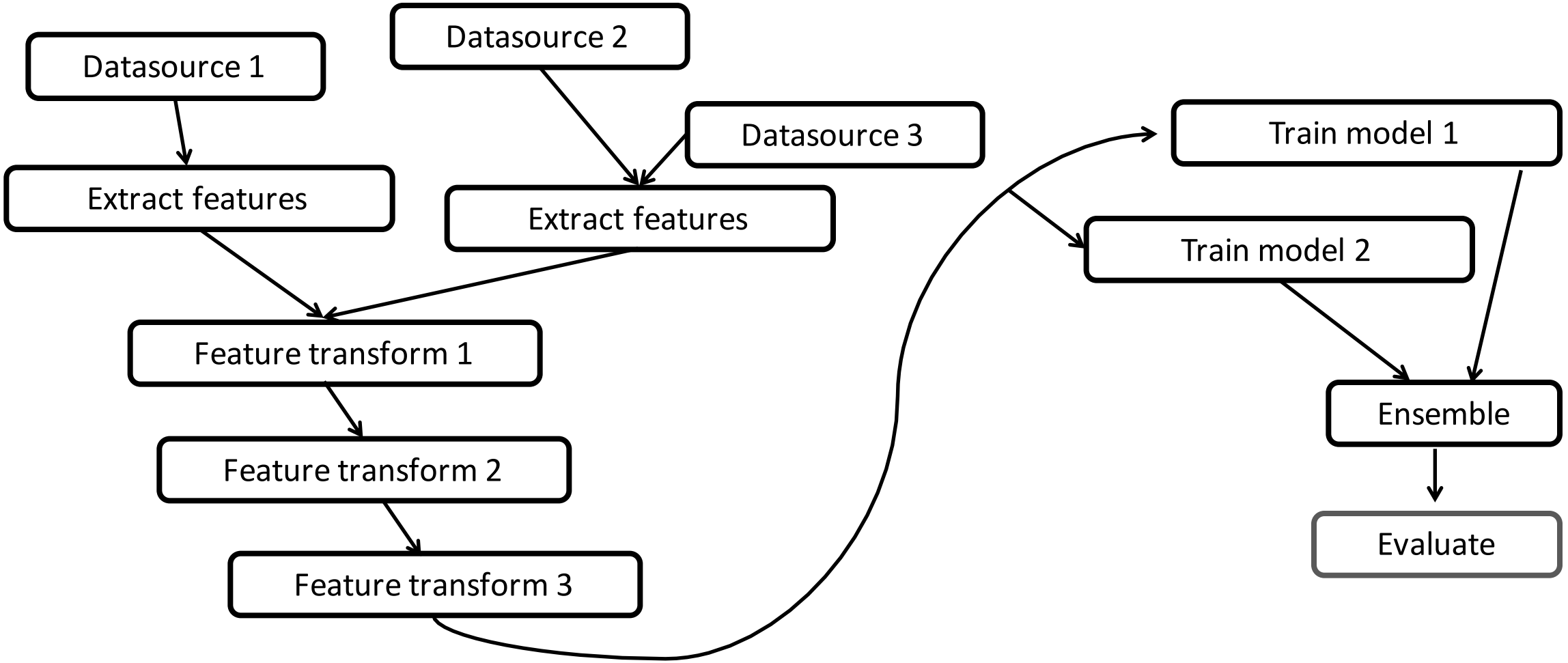


Model selection
Choose among different models or model hyperparameters

ML Pipelines



ML Pipelines



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

Hands On Demo

Algorithms & performance

- Survival analysis, linear algebra, bisecting k-means, autoencoder & RBM, and more
- Model stats, weighted instance support

Pipeline & model persistence

Spark R

- Extend GLM and R formula support
- Model summaries

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

Web Sites

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

Thank you.