Parallel Programming with Apache Spark

Matei Zaharia

CS 341, Spring 2017

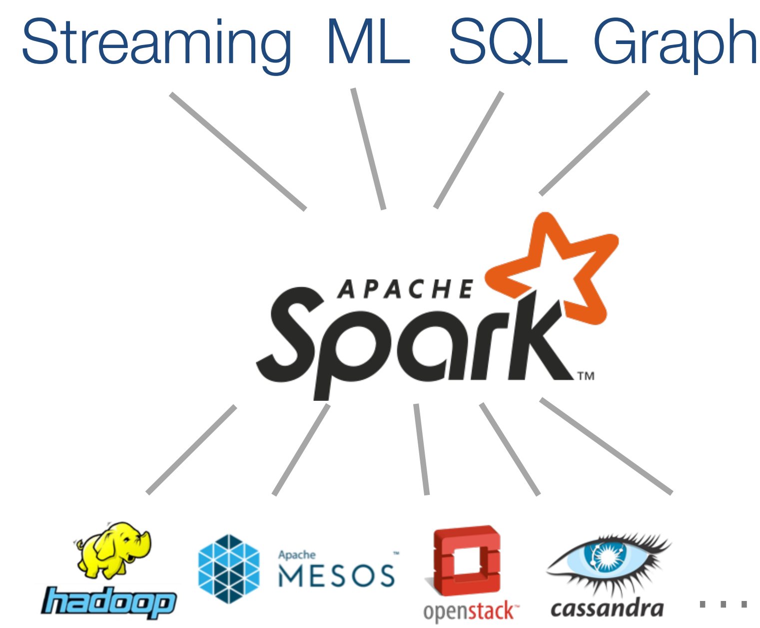
What is Apache Spark?

Open source computing engine for clusters

- » Generalizes MapReduce

Rich set of APIs & libraries

- » APIs in Scala, Java, Python, R
- » SQL, machine learning, graphs



Project History

Started as research project at Berkeley in 2009

Open sourced in 2010

Joined Apache foundation in 2013

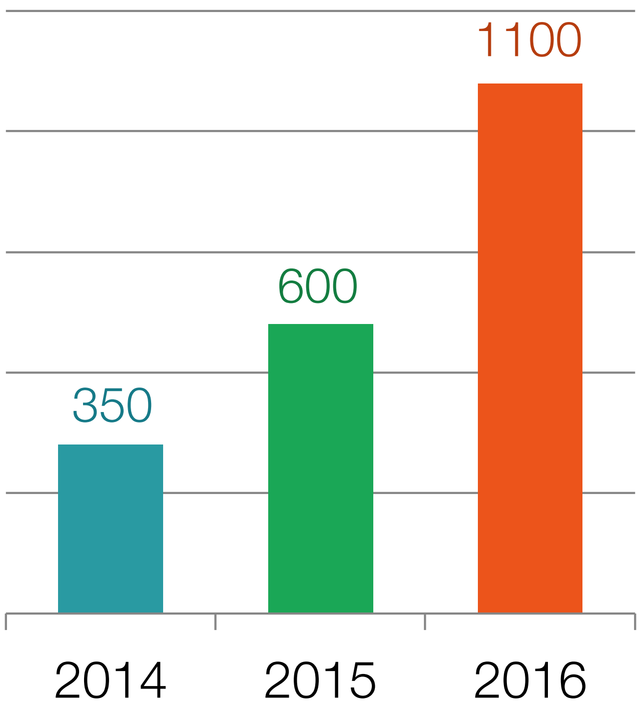
1000+ contributors to date

Spark Community

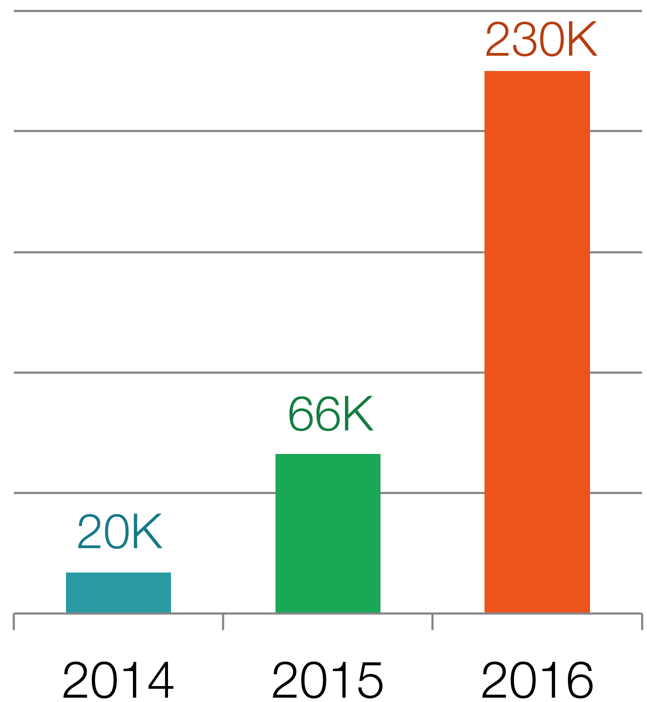
1000+ companies, clusters up to 8000 nodes

Community Growth

Developers Contributing



Spark Meetup Members



This Talk

Introduction to Spark

Tour of Spark operations

Job execution

Higher-level libraries

Key Idea

Write apps in terms of transformations on distributed datasets

Resilient distributed datasets (RDDs)

- » Collections of objects spread across a cluster
- » Built through parallel transformations (map, filter, etc)
- » Automatically rebuilt on failure
- » Controllable persistence (e.g. caching in RAM)

Operations

Transformations (e.g. map, filter, groupBy)

» Lazy operations to build RDDs from other RDDs

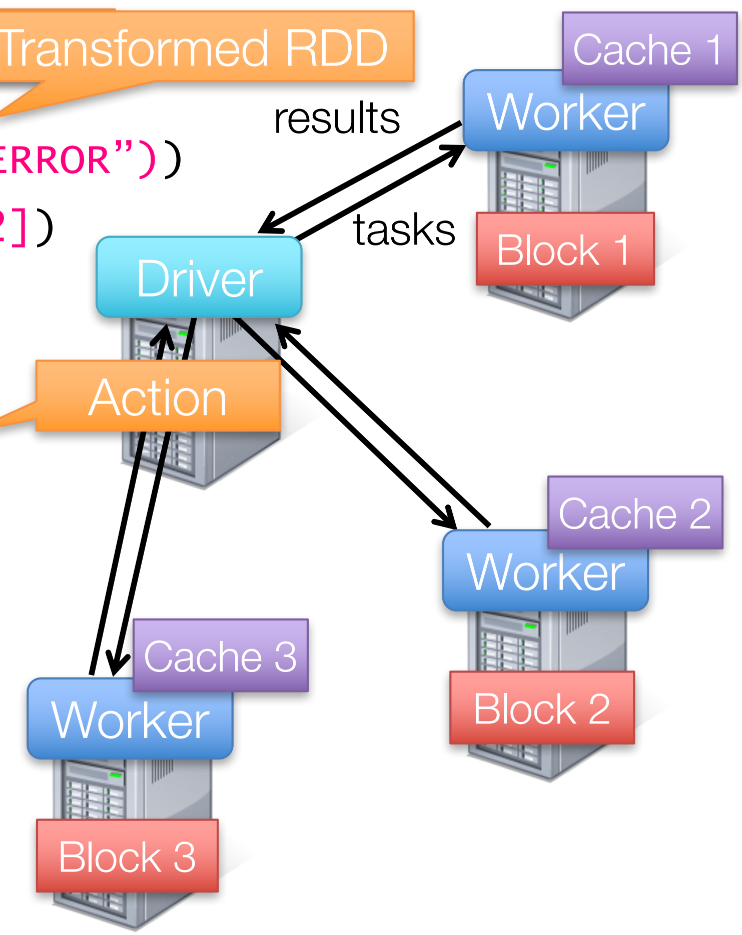
Actions (e.g. count, collect, save)

» Return a result or write it to storage

Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "foo" in s).count()
messages.filter(lambda s: "bar" in s).count()
. . .
```

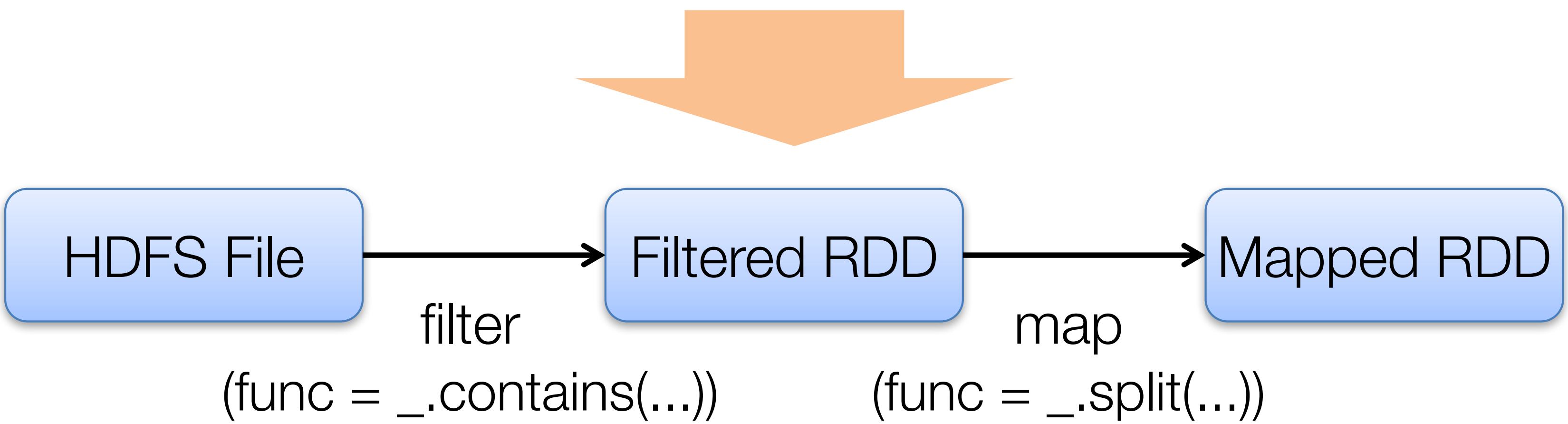


Result: full-text search of Wikipedia in 0.5 sec (vs 20 s for on-disk data)

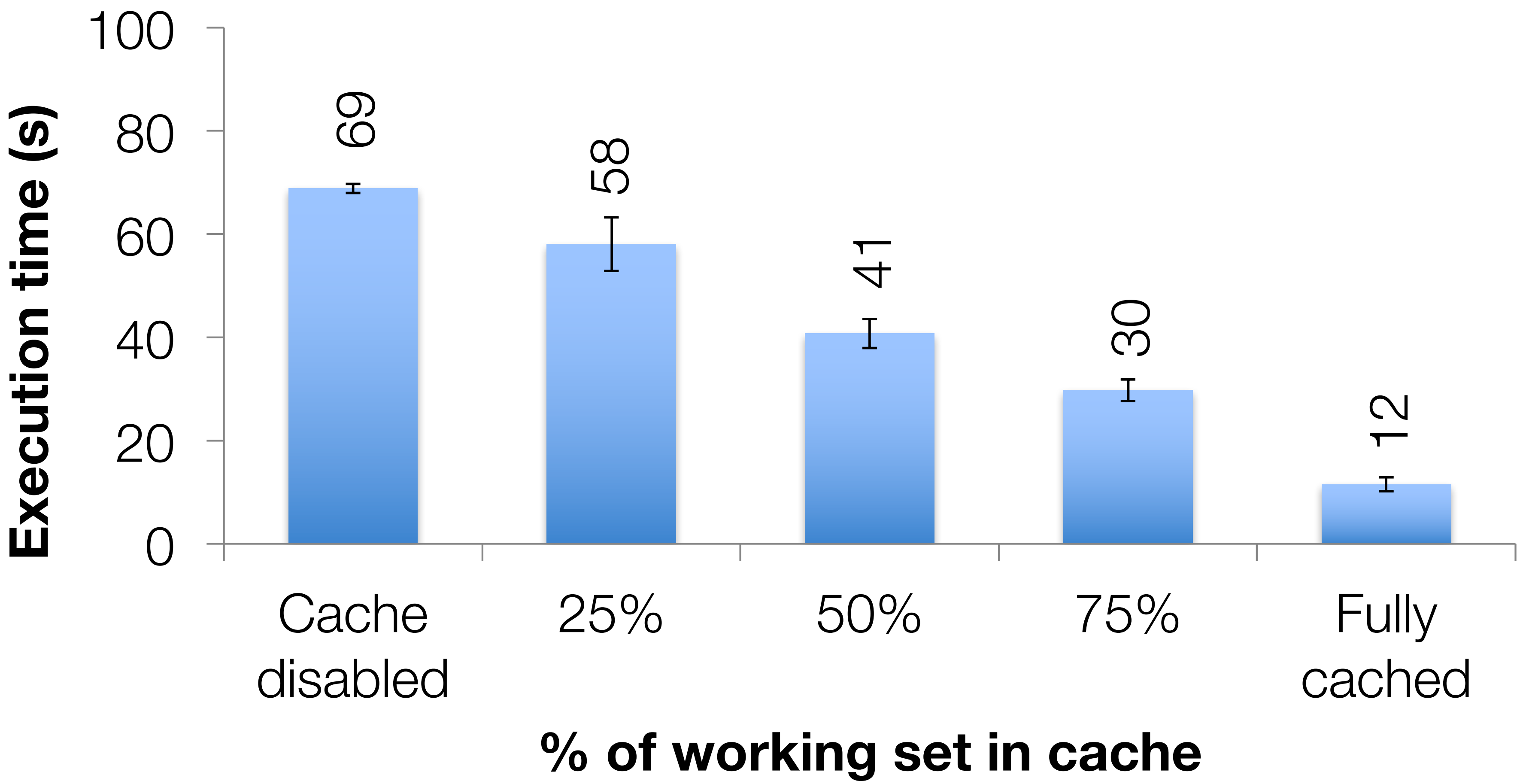
Fault Recovery

RDDs track *lineage* information that can be used to efficiently recompute lost data

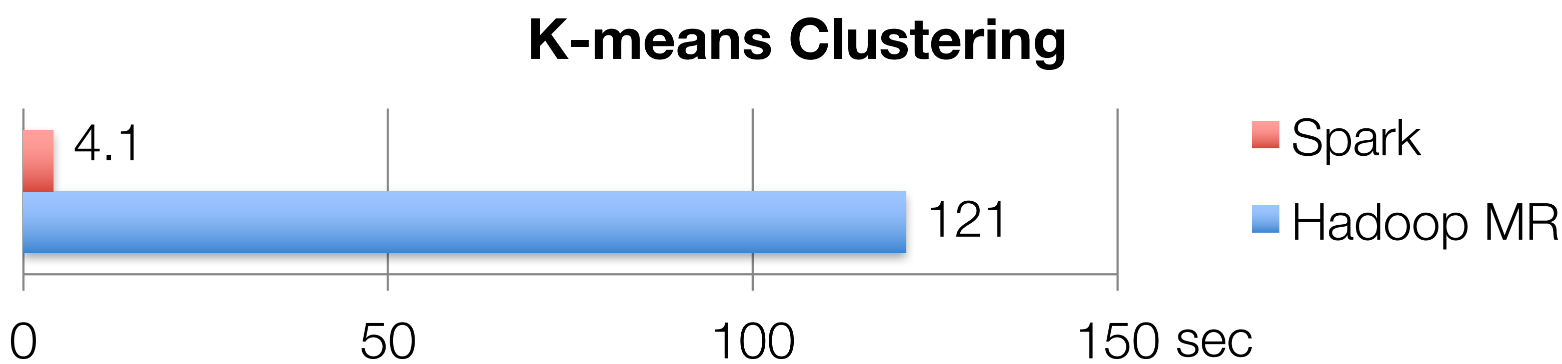
Ex:
```
msgs = textFile.filter(lambda s: s.startsWith("ERROR"))
               .map(lambda s: s.split("\t")[2])
```

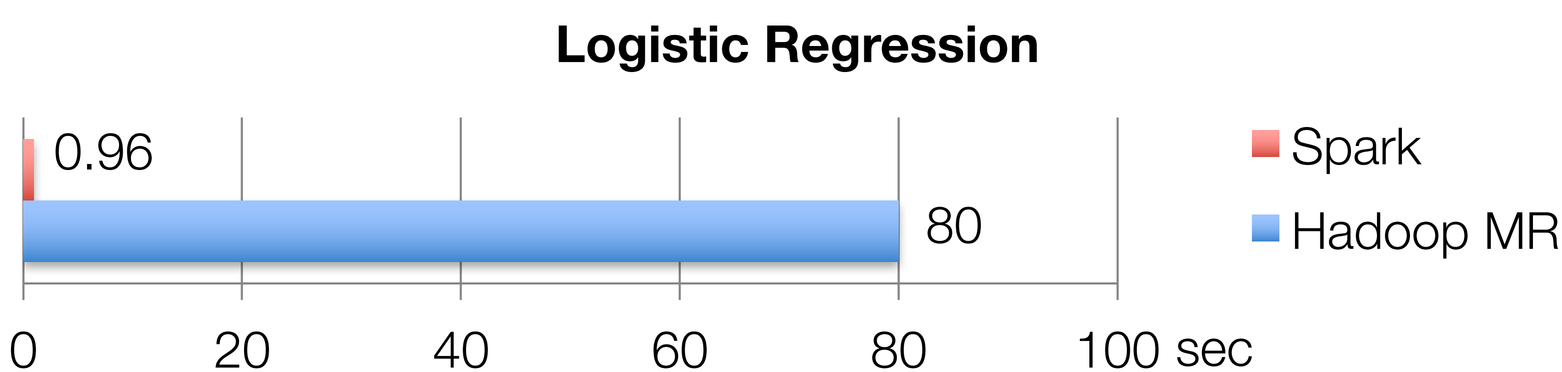


Behavior with Less RAM



Iterative Algorithms





Spark in Scala and Java

```
// Scala:

val lines = sc.textFile(...)
lines.filter(x => x.contains("ERROR")).count()


// Java:

JavaRDD<String> lines = sc.textFile(...);
lines.filter(s -> s.contains("error")).count();
```

Installing Spark

Spark runs on your laptop: download it from spark.apache.org

Cloud services:

» Google Cloud DataProc
» Databricks Community Edition

This Talk

Introduction to Spark

Tour of Spark operations

Job execution

Higher-level libraries

Learning Spark

Easiest way: the shell (`spark-shell` or `pyspark`)

- Special Scala/Python interpreters for cluster use

Runs in local mode on all cores by default, but can connect to clusters too (see docs)

First Stop: SparkContext

Main entry point to Spark functionality

Available in shell as variable `sc`

In standalone apps, you create your own

Creating RDDs

```
# Turn a Python collection into an RDD
sc.parallelize([1, 2, 3])

# Load text file from local FS, HDFS, or S3
sc.textFile("file.txt")
sc.textFile("directory/*.txt")
sc.textFile("hdfs://namenode:9000/path/file")

# Use existing Hadoop InputFormat (Java/Scala only)
sc.hadoopFile(keyClass, valClass, inputFmt, conf)
```

Basic Transformations

```
nums = sc.parallelize([1, 2, 3])

# Pass each element through a function
squares = nums.map(lambda x: x*x)   // {1, 4, 9}

# Keep elements passing a predicate
even = squares.filter(lambda x: x % 2 == 0) // {4}

# Map each element to zero or more others
nums.flatMap(lambda x: range(x))
    # => {0, 0, 1, 0, 1, 2}
```

Range object (sequence of numbers 0, 1, …, x-1)

Basic Actions

```
nums = sc.parallelize([1, 2, 3])

# Retrieve RDD contents as a local collection
nums.collect() # => [1, 2, 3]

# Return first K elements
nums.take(2)   # => [1, 2]

# Count number of elements
nums.count()   # => 3

# Merge elements with an associative function
nums.reduce(lambda x, y: x + y)  # => 6

# Write elements to a text file
nums.saveAsTextFile("hdfs://file.txt")
```

Working with Key-Value Pairs

Spark's "distributed reduce" transformations operate on RDDs of key-value pairs

Python:

```python
pair = (a, b)
pair[0] # => a
pair[1] # => b
```

Scala:

```scala
val pair = (a, b)
pair._1 // => a
pair._2 // => b
```

Java:

```java
Tuple2 pair = new Tuple2(a, b);
pair._1 // => a
pair._2 // => b
```

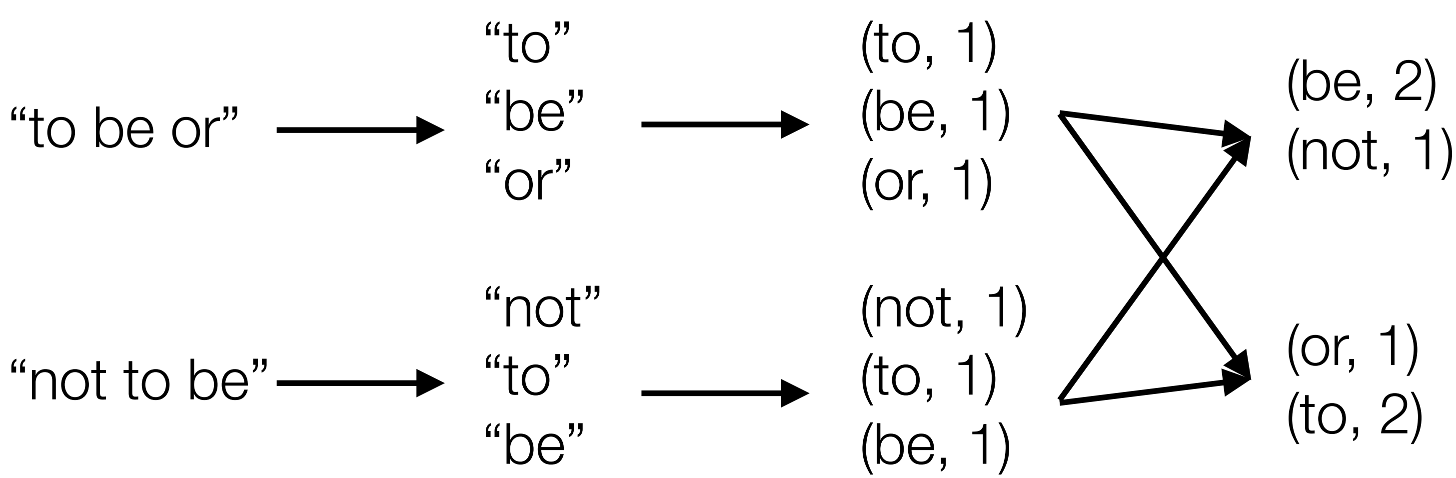
Some Key-Value Operations

```
pets = sc.parallelize(
    [(“cat”, 1), (“dog”, 1), (“cat”, 2)])

pets.reduceByKey(lambda x, y: x + y)
                  # => {(cat, 3), (dog, 1)}

pets.groupByKey() # => {(cat, [1, 2]), (dog, [1])}

pets.sortByKey()  # => {(cat, 1), (cat, 2), (dog, 1)}
```

`reduceByKey` also aggregates on the map side

Example: Word Count

```
lines = sc.textFile("hamlet.txt")

counts = lines.flatMap(lambda line: line.split(" "))
              .map(lambda word: (word, 1))
              .reduceByKey(lambda x, y: x + y)
```



Other Key-Value Operations

```
visits = sc.parallelize([ (“index.html”, “1.2.3.4”),
                          (“about.html”, “3.4.5.6”),
                          (“index.html”, “1.3.3.1”) ])

pageNames = sc.parallelize([ (“index.html”, “Home”),
                             (“about.html”, “About”) ])

visits.join(pageNames)
# (“index.html”, (“1.2.3.4”, “Home”))
# (“index.html”, (“1.3.3.1”, “Home”))
# (“about.html”, (“3.4.5.6”, “About”))

visits.cogroup(pageNames)
# (“index.html”, ([“1.2.3.4”, “1.3.3.1”], [“Home”]))
# (“about.html”, ([“3.4.5.6”], [“About”]))
```

Setting the Level of Parallelism

All the pair RDD operations take an optional second parameter for number of tasks

```
words.reduceByKey(lambda x, y: x + y, 5)

words.groupByKey(5)

visits.join(pageViews, 5)
```

Using Local Variables

Any external variables you use in a closure will automatically be shipped to the cluster:

```
query = sys.stdin.readline()
pages.filter(lambda x: query in x).count()
```

Some caveats:

- Each task gets a new copy (updates aren't sent back)
- Variable must be Serializable / Pickle-able
- Don't use fields of an outer object (ships all of it!)

Other RDD Operators

`map`	`reduce`	`sample`
`filter`	`count`	`take`
`groupBy`	`fold`	`first`
`sort`	`reduceByKey`	`partitionBy`
`union`	`groupByKey`	`mapWith`
`join`	`cogroup`	`pipe`
`leftOuterJoin`	`cross`	`save`
`rightOuterJoin`	`zip`	`...`

More details: spark.apache.org/docs/latest

This Talk

Introduction to Spark

Tour of Spark operations

Job execution

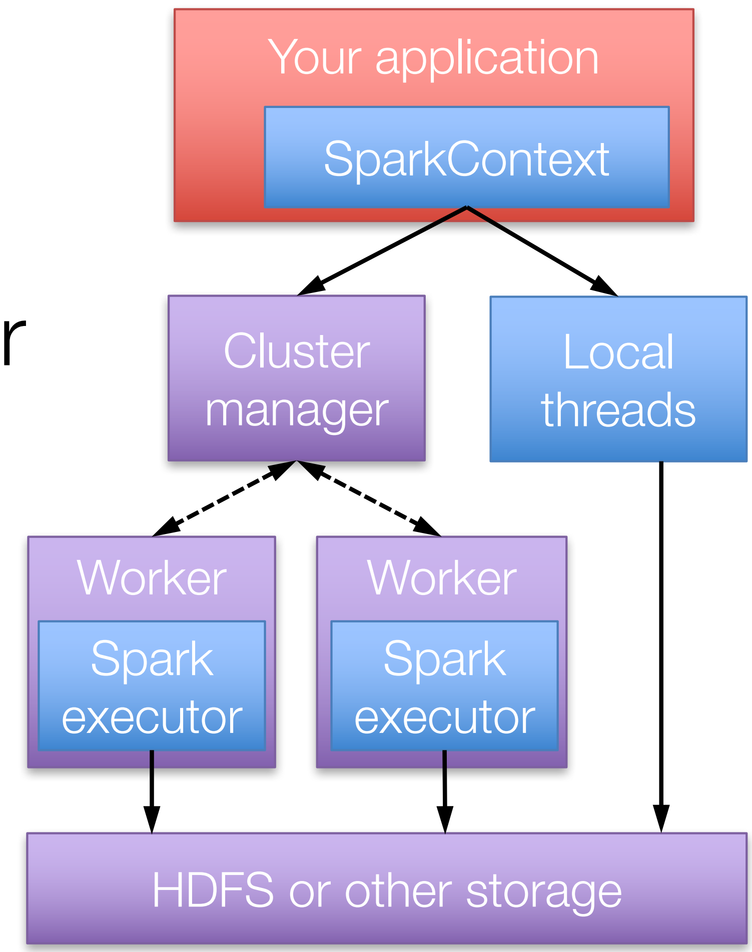
Higher-level libraries

Components

Spark runs as a library in your driver program

Runs tasks locally or on cluster
» Standalone, Mesos or YARN

Accesses storage via data source plugins
» Can use S3, HDFS, GCE, …



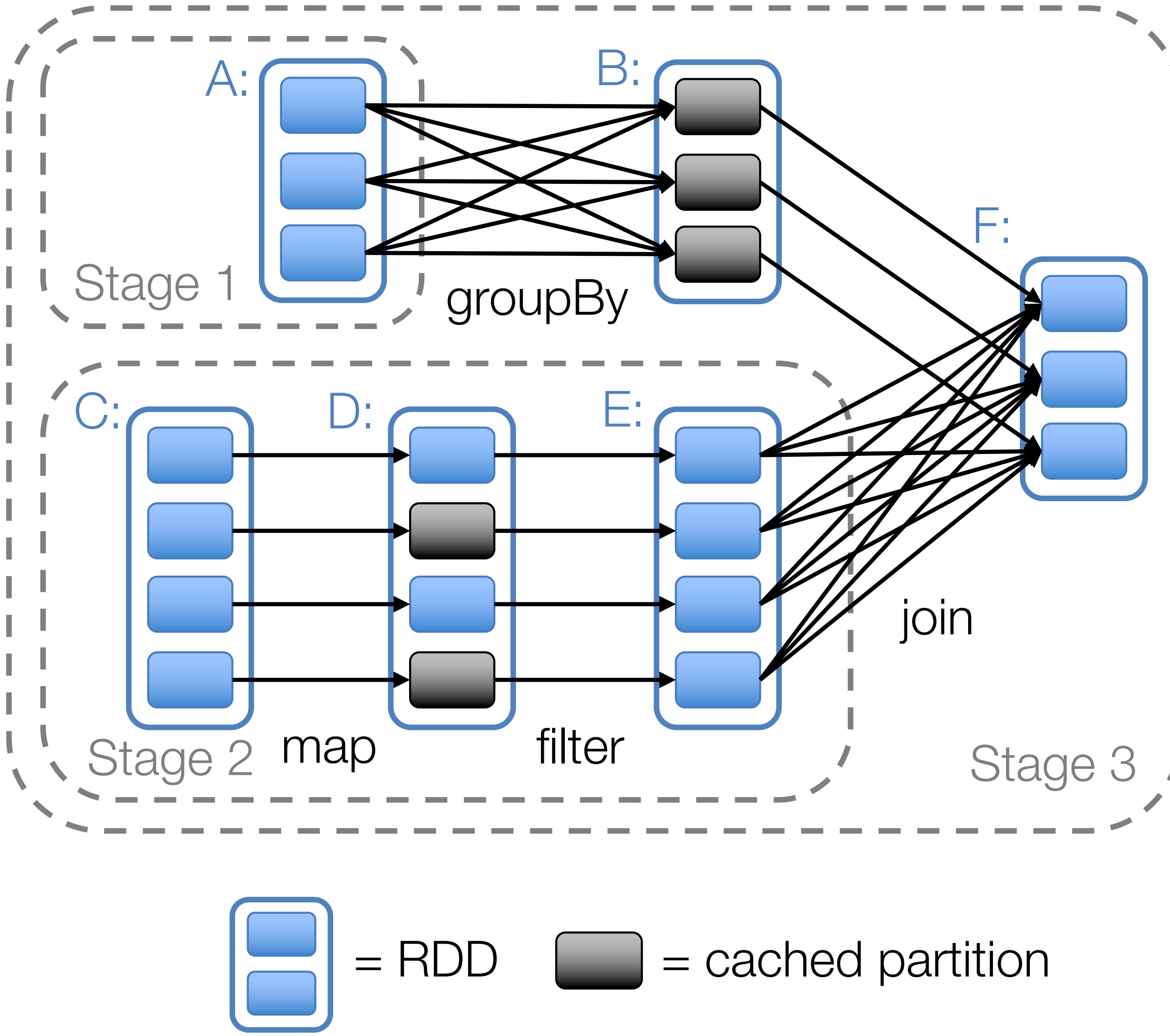
Job Scheduler

General task graphs

Automatically pipelines functions

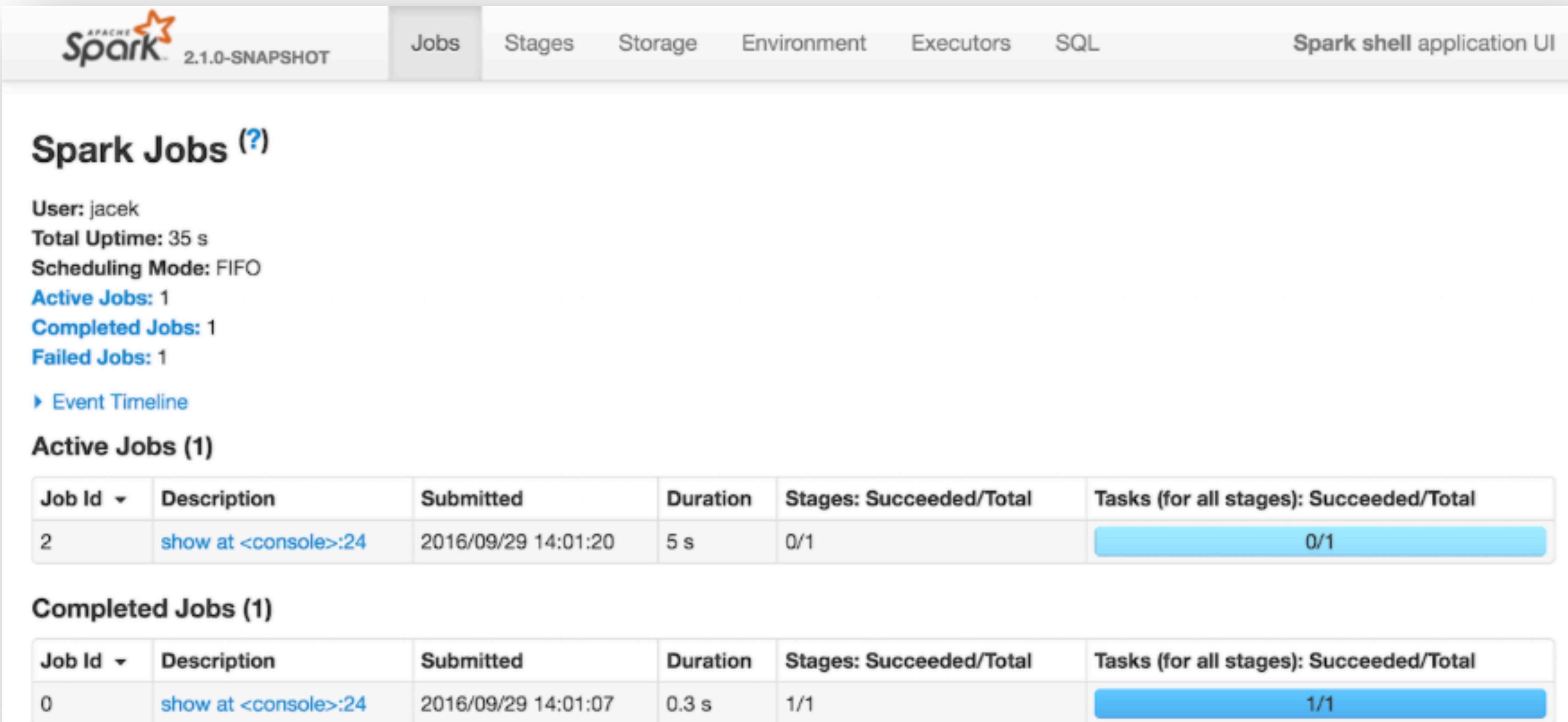
Data locality aware

Partitioning aware to avoid shuffles



Debugging

Spark UI available at http://<master-node>:4040



This Talk

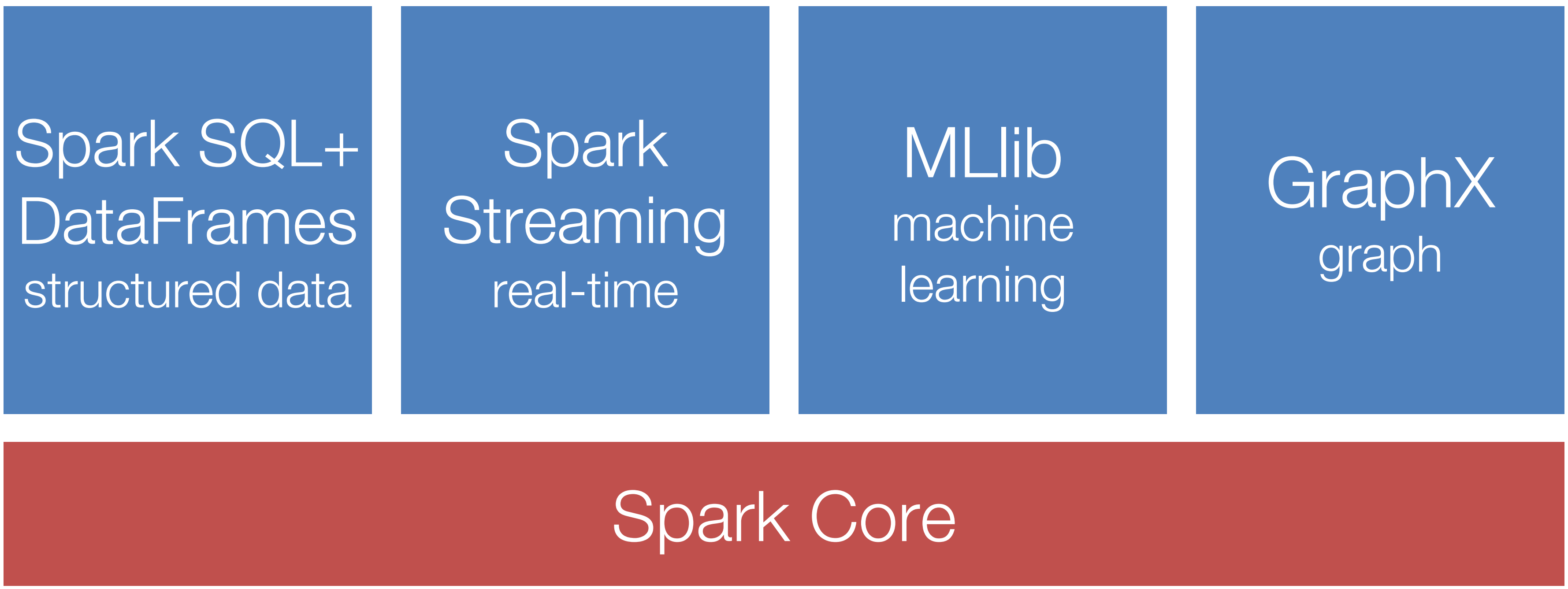
Introduction to Spark

Tour of Spark operations

Job execution

Higher-level libraries

Libraries Built on Spark



Spark SQL & DataFrames

APIs for *structured data* (table-like data)

» SQL
» DataFrames: dynamically typed
» Datasets: statically typed

Similar optimizations to relational databases

DataFrame API

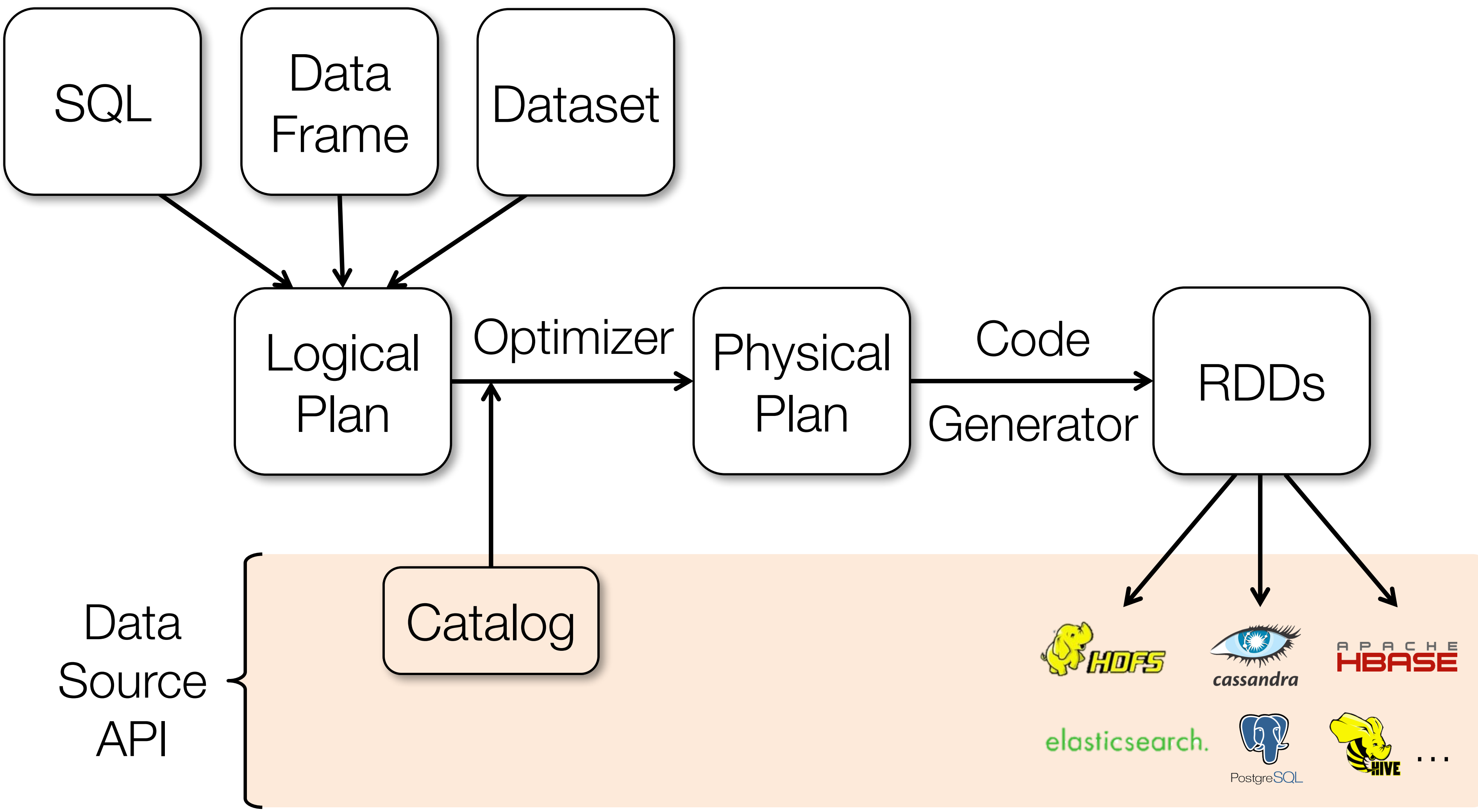
Domain-specific API similar to Pandas and R

» DataFrames are tables with named columns

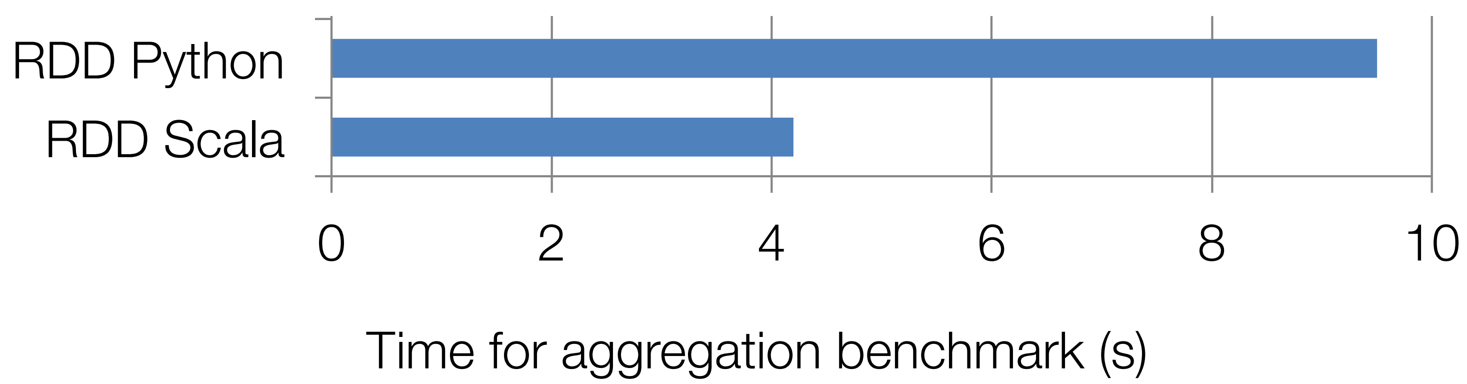
```
users = spark.sql(“select * from users”)

ca_users = users[users[“state”] == “CA”]

ca_users.count()

ca_users.groupBy(“name”).avg(“age”)

caUsers.map(lambda row: row.name.upper())
```

Expression AST

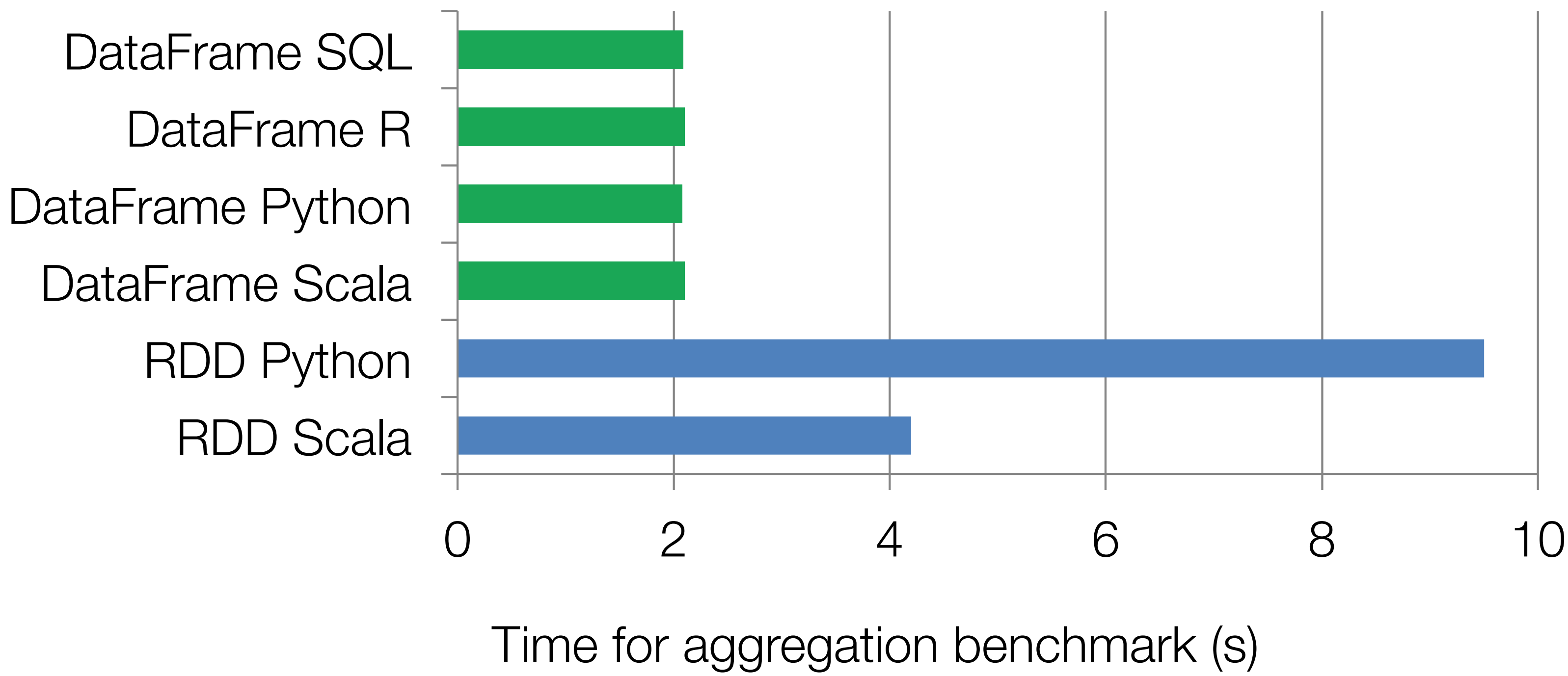
Execution Steps



Performance



Performance

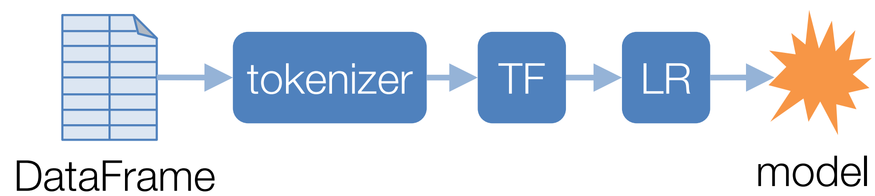


MLlib

High-level *pipeline* API similar to SciKit-Learn

Acts on DataFrames

Grid search and cross validation for tuning

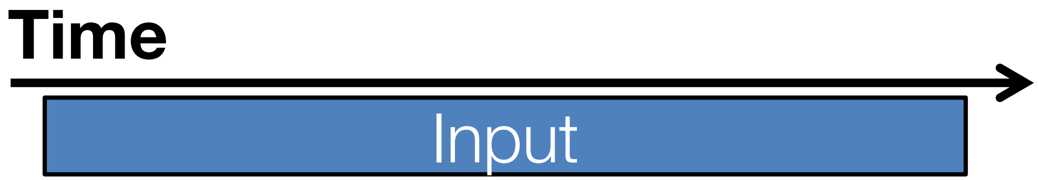



```
tokenizer = Tokenizer()
tf = HashingTF(numFeatures=1000)
lr = LogisticRegression()

pipe = Pipeline(
    [tokenizer, tf, lr])
model = pipe.fit(df)
```

MLlib Algorithms

Generalized linear models
Alternating least squares
Decision trees
Random forests, GBTs
Naïve Bayes
PCA, SVD
AUC, ROC, f-measure
K-means
Latent Dirichlet allocation
Power iteration clustering
Gaussian mixtures
FP-growth
Word2Vec
Streaming k-means

Spark Streaming



Spark Streaming

Represents streams as a series of RDDs over time

```
val spammers = sc.sequenceFile("hdfs://spammers.seq")

sc.twitterStream(...)
  .filter(t => t.text.contains("Stanford"))
  .transform(tweets => tweets.map(t => (t.user, t)).join(spammers))
  .print()
```

Combining Libraries

```
# Load data using Spark SQL
points = spark.sql(
  “select latitude, longitude from tweets”)

# Train a machine learning model
model = KMeans.train(points, 10)

# Apply it to a stream
sc.twitterStream(...)
  .map(lambda t: (model.predict(t.location), 1))
  .reduceByWindow(“5s”, lambda a, b: a + b)
```

Conclusion

Spark offers a wide range of high-level APIs for parallel data processing

Can run on your laptop or a cloud service

Online tutorials:

- spark.apache.org/docs/latest
- Databricks Community Edition