# Data Analytics with Spark

1. Overview
2. Spark Applications
3. Resilient Distributed Datasets
4. Datasets
5. Architecture
6. Operations
7. Tools
8. Conclusion
9. Spark Practice

# 1. Overview

- Apache software, from UC Berkeley
  - Compatible with Hadoop (HDFS)
- Extension of MapReduce for two classes of analytics applications
  - Iterative processing (machine learning, graphs)
  - Interactive data mining (R, Excel, Python)
- Major performance improvement (up to 100*)
  - In-memory processing
  - Optimization of the task graph
- Major usability improvement
  - APIs for Java, Python, R and Scala (functional extension of Java)
  - Interactive use from a Scala interpreter
- Major adoption from industry
  - Databricks: a successful startup from UC Berkeley
  - Will replace MapReduce

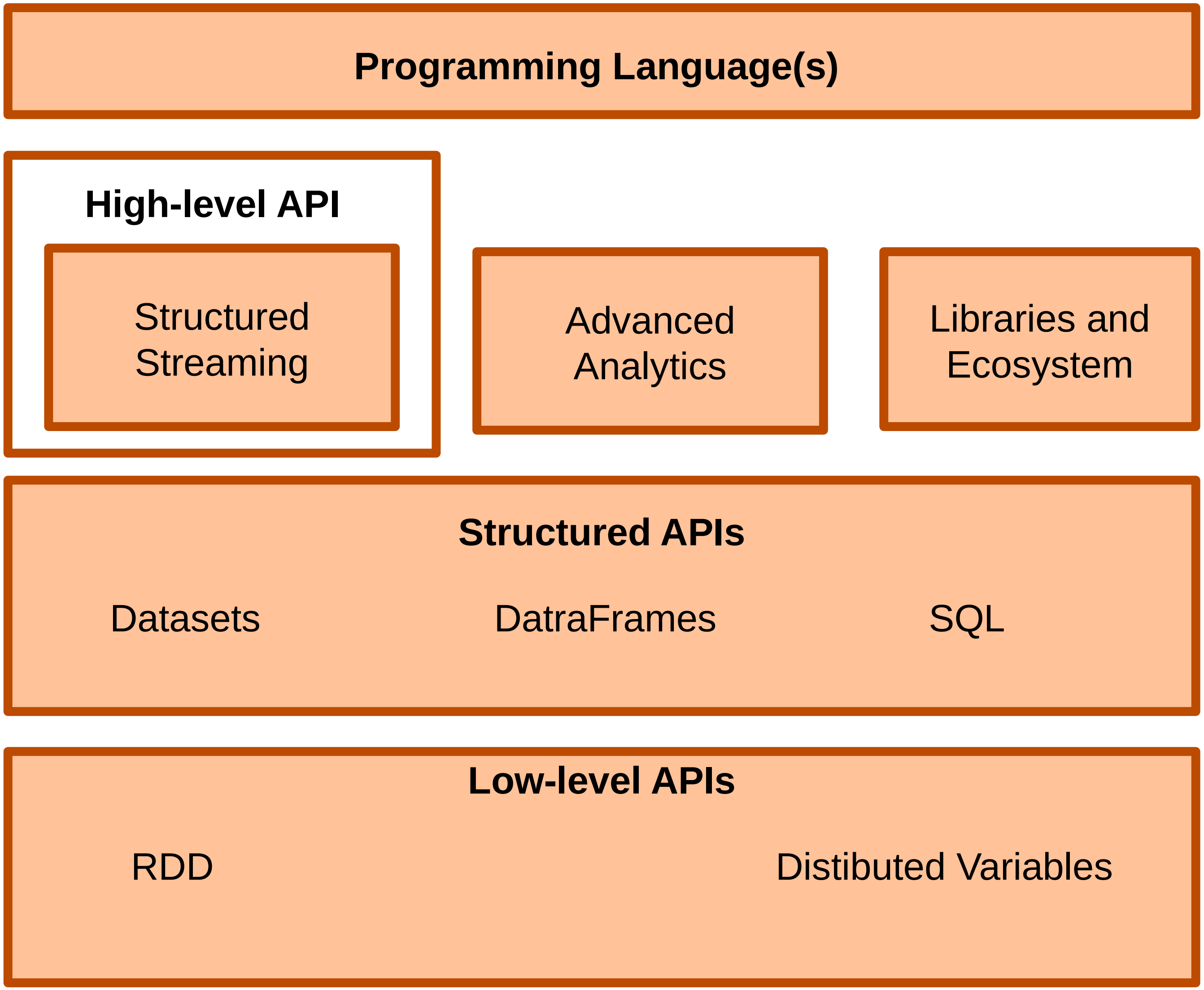
# What is Spark?

- Unified computing engine and set of libraries for parallel data processing on computer cluster
  - Unified platform for writing big data applications
  - Computing engine:
    - Spark’s focus on computation
    - Loading data from storage systems and performing computations on it
    - Using Spark with a wide variety of persistent storage systems
  - Libraries: unified API for common data analysis tasks
    - SQL and structured data (Spark SQL)
    - Machine learning (Mlib)
    - Stream processing (Spark Streaming and the newer Structured Streaming)
    - Graph analytics (GraphX)

# Spark Model

- Concept : Resilient Distributed Dataset (RDD)
  - A collection of objects distributed in a cluster
  - Built from parallel transformations (map, filter, etc.)
  - Can be maitained in memory for efficient processing
  - Preserves the nice properties of MapReduce
    - Fault-tolerance, data locality, scalability
- An RDD comes with its provenance information
  - How it has been produced from other RDDs
  - Useful to rebuild an RDD that has been lost after a crash
- The user can control
  - Data persistence (disk or RAM)
  - Data partitioning (hashing, range, [<k, v>])
- Operators: transformations and actions

# Spark Toolset (1)



# Spark Toolset (2)

- Widely used programming languages (Python, Java, Scala, and R)
- Structured streaming: high-level API for stream processing
  - Allows to rapidly  and quickly extract value out of streaming systems with virtually no code change
  - Easy to conceptualize
- Advanced analytics
  - Techniques aimed at solving the core problem of deriving insights
  - Making predictions or recommendations based on data
- Ecosystem
  - Packages and tools created by the community
  - Spark-packages.org

# Spark Toolset (3)

- Libraries
  - Unified API for data analysis
  - Provides more and more types of functionality
  - Hundreds of open source external libraries
- Structured APIs for manipulating all sorts of data such as
  - Unstructured log files
  - Semi-structured CSV files
  - Highly structured Parquet files
- Low-level APIs
  - Manipulating distributed data (RDDs)
  - Distributing and manipulating distributed shared variables
    - Broadcast variables
    - Accumulators

# 2. Spark Application (1)

- Set of Spark jobs defined by one Spark context in the driver program
- Two components
  - Driver process: runs the main() function, sits on a node in the cluster
    - Maintains information about the Spark application
    - Responds to a user's program or input
    - Analyzes, distributes, and schedules work across the executors
  - Executor: carrying out the work that the driver assigns them
    - Executes code assigned by the driver
    - Reports the state of computation on that executor back to the driver node

# Spark Application (2)

- SparkSession is a driver process that enables to control Spark Application

```
In Scala:
res0: org.apache.spark.sql.SparkSession =
org.apache.spark.sql.SparkSession@...

In Python:
<pyspark.sql.session.SparkSession at 0x7efda4c1ccd0>
```

- The SparkSession instance is the way Spark executes user-defined manipulations across the cluster

# Spark Application (3)

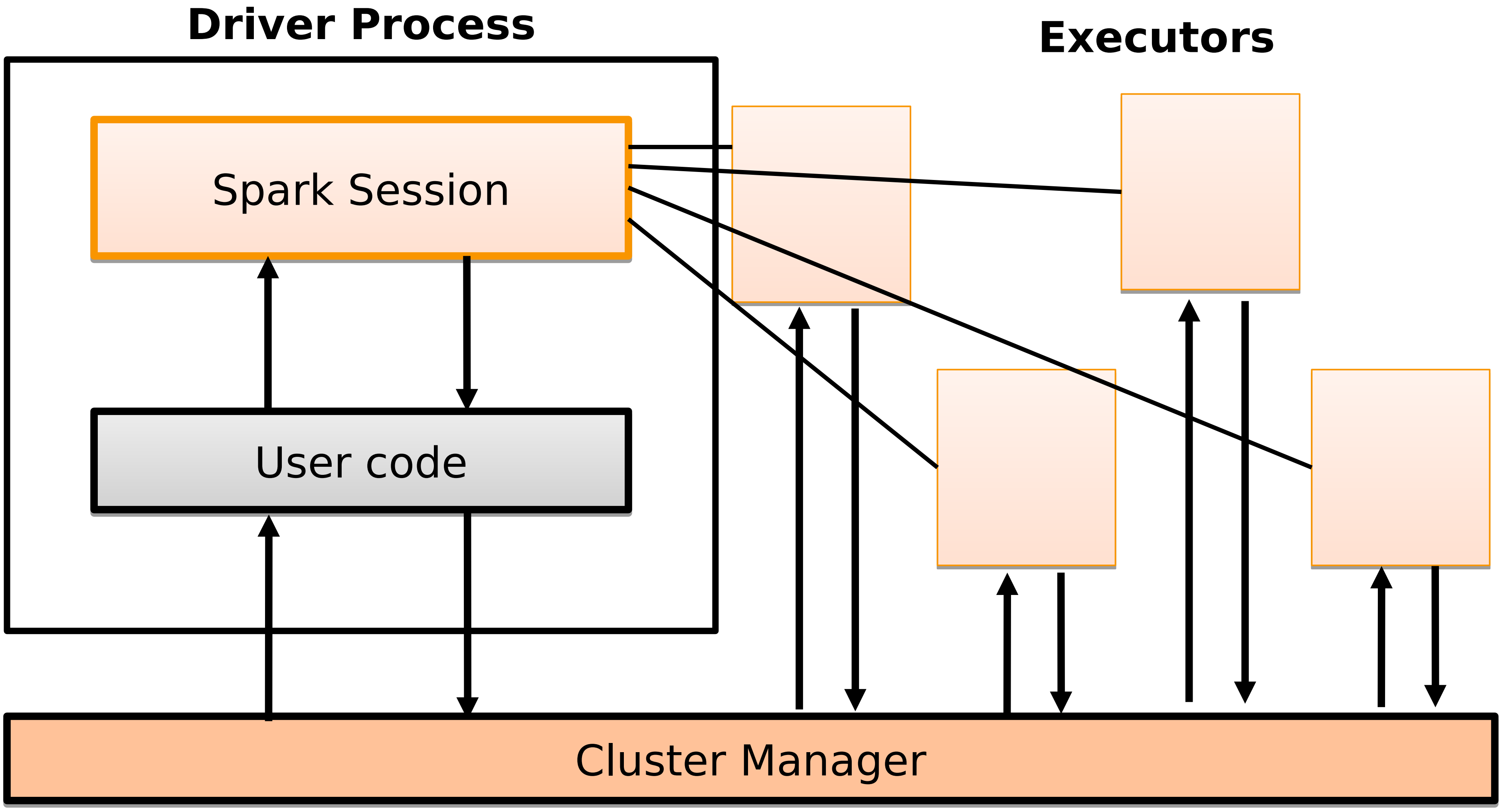
- SparkContext object is the driver program that communicates with the appropriate cluster manager to run the tasks.

```
# in Python
from pyspark.sql import Row
schema = df.schema
newRows = [
Row("New Country", "Other Country", 5L),
Row("New Country 2", "Other Country 3", 1L)
]
parallelizedRows = spark.sparkContext.parallelize(newRows)
newDF = spark.createDataFrame(parallelizedRows, schema)

# in Python
df.union(newDF)\
.where("count = 1")\
.where(col("ORIGIN_COUNTRY_NAME") != "United States")\
.show()

Giving the output of:
+-----------------+-------------------+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|count|
+-----------------+-------------------+-----+
| United States   | Croatia           | 1   |
```

# Spark Application Architecture

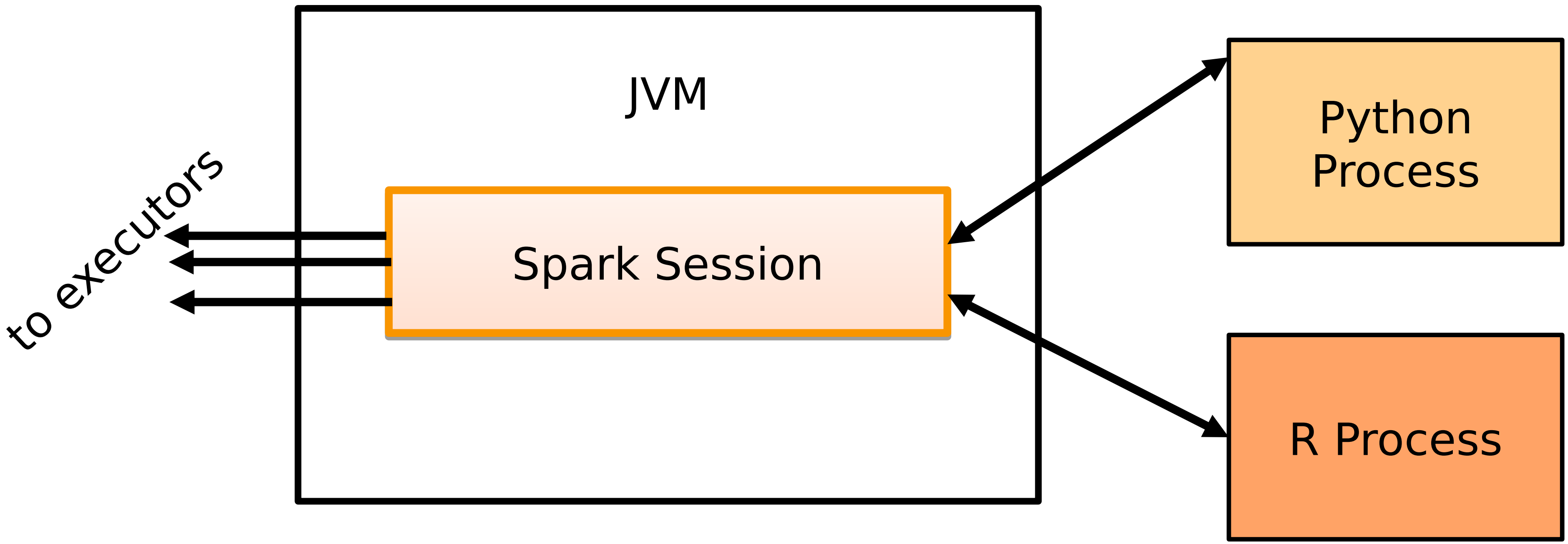


# Spark Language

- Spark code uses several programming languages
  - Scala (Spark's default language)
  - Java
  - Python (supports nearly all construts that Scala supports)
  - SQL (supports a subset of the ANSI SQL 2003 standard)
  - R Spark uses Spark core (SparkR) and R community-driven package (sparklyr)

- Spark has some core concepts in every language
  - Translating concepts into Spark code
  - Runs Spark code on a cluster of machines

# Relationship between SparkSession and Spark's Language API



# 3. Resilient Distributed Datasets

- Resilient Distributed Dataset (RDD)
  - In memory (possibly cached)
  - Read-only collection of objects
  - Partitioned across machines of a cluster
  - Lineage: allows partitions to be rebuilt when failure

# RDD API

- Set of partitions (splits)
  - In memory (possibly cached)
- List of dependencies
- Function to compute a partition
- Preferred locations
- Partitioner

# RDD Construction

- From a file (e.g., in HDFS)

```
val rdd = sc.textFile("hdfs://...")
```

- By parallelizing a collection

```
val data = Range(0, 100)
val rdd = sc.parallelize(data)
```

- By transforming another RDD

```
val rdd2 = rdd1.filter(...)
```

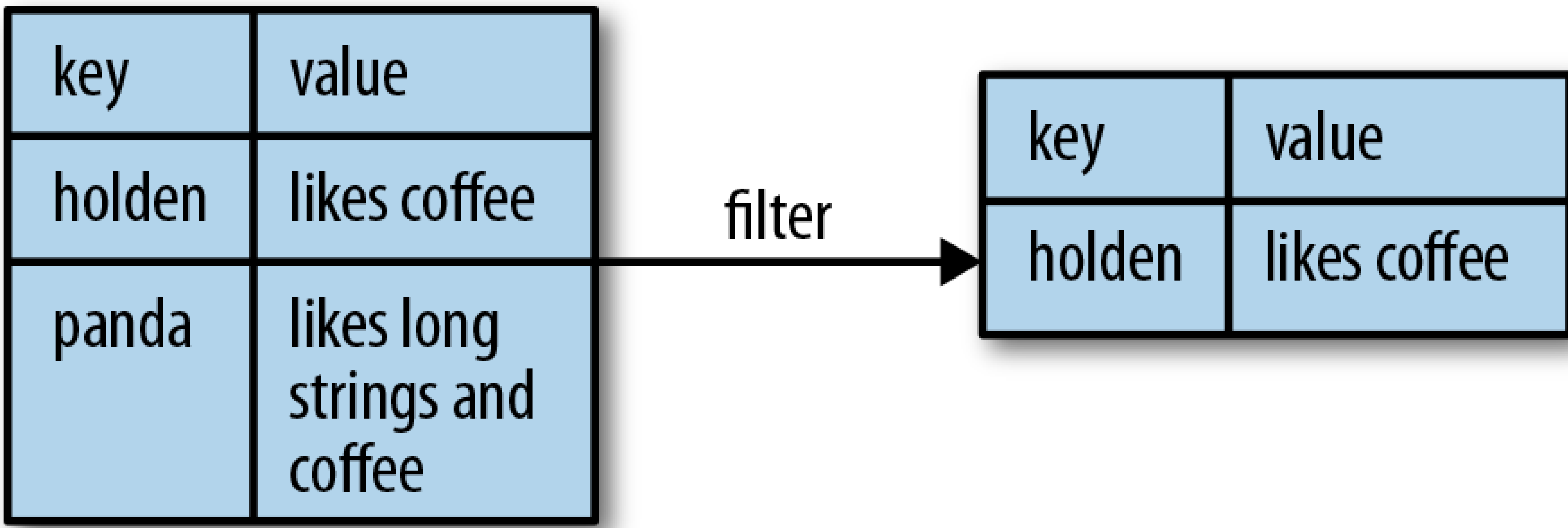
# Operations

- Transformations
  - RDD(s) → RDD
  - Lazily applied
    - Only computed if result is returned
  - Chain of transformations recomputed each time
    - Except if persisted with cache()
- Actions
  - RDD → value

# Transformations: simple, no shuffle

| **Transformation** | **Types** |
| --- | --- |
| map(f: T => U) | RDD[T] → RDD[U] |
| filter(f: T => boolean) | RDD[T] → RDD[T] |
| flatMap(f: T => Seq[U]) | RDD[T] → RDD[U] |
| mapPartitions(f: Seq[T] => Seq[U]) | RDD[T] → RDD[U] |
| mapPartitionsWithIndex(f: (int, Seq[T]) => Seq[U]) | RDD[T] → RDD[U] |
| sample() | RDD[T] → RDD[T] |
| sortBy(f: T => K) | RDD[T] → RDD[T] |
| keyBy(f: T = K) | RDD[T] → RDD[(K,T)] |

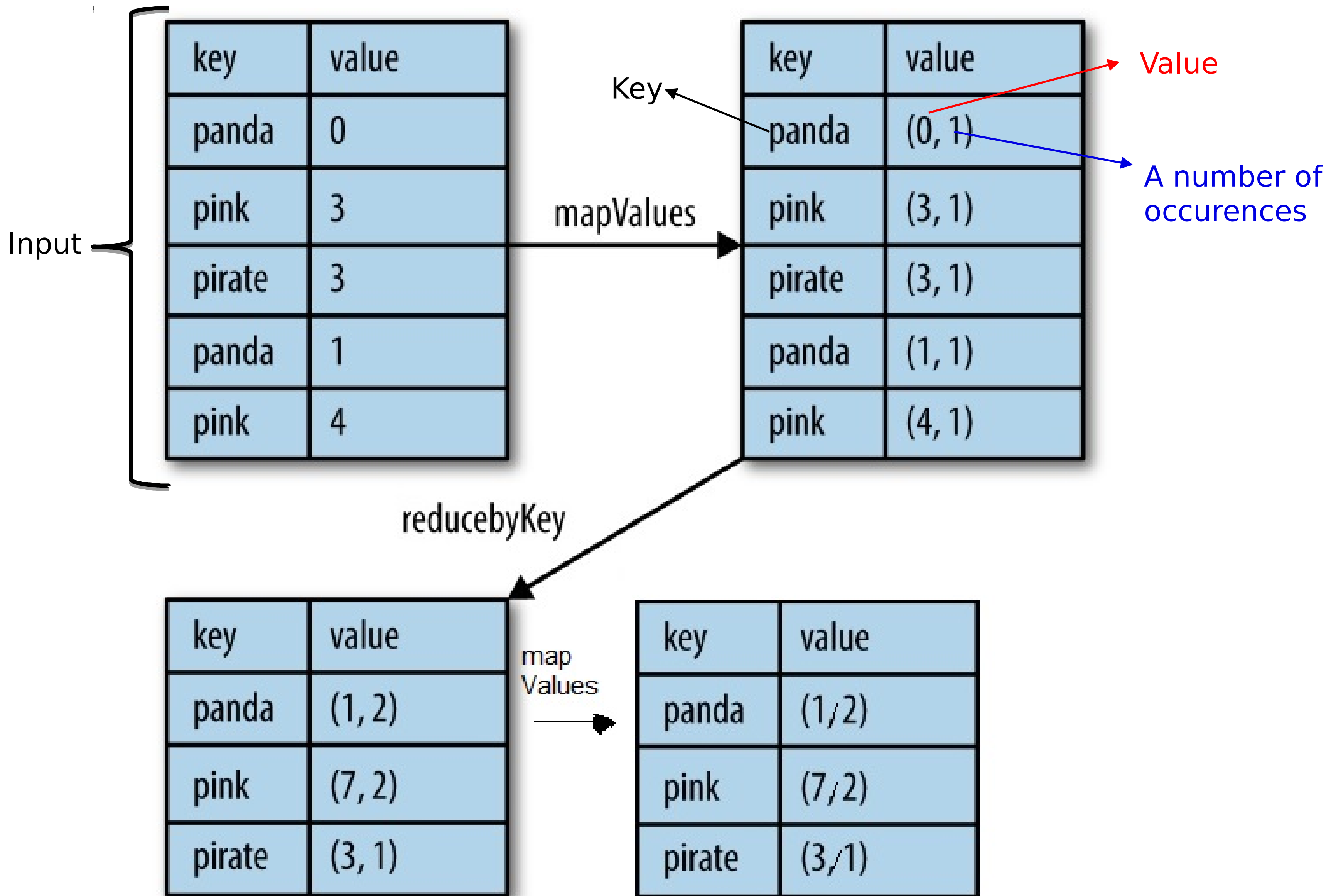
# Transformations: simple, no shuffle



# Transformations: key-value, no shuffle

| Transformation | Types |
|---|---|
| mapValues(f: V => U) | RDD[(K,V)] → RDD[(K,U)] |
| flatMapValues(f: V => Seq[U]) | RDD[(K,V)] → RDD[(K,U)] |
| sampleByKey() | RDD[(K,V)] → RDD[(K,V)] |

# Transformations: key-value, shuffle



# Transformations: key-value, shuffle

| Transformation | Types |
|---|---|
| groupByKey() | RDD[(K,V)] → RDD[(K, Seq[V])] |
| reduceByKey(f: (V,V) => V) | RDD[(K,V)] → RDD[(K,V)] |
| aggregateByKey(zero: U)<br>(f: (U,V) => U, g: (U,U) -> U) | RDD[(K,V)] → RDD[(K,V)] |
| combineByKey(f: V => C,<br>g: (C, V) => C, h: (c, C) => C) | RDD[(K,V)] → RDD[(K,C)] |
| foldByKey(zero: V)(f: (V, V) => V) | RDD[(K,V)] → RDD[(K,V)] |
| groupByKey() | RDD[(K,V)] → RDD[(K, Seq[V])] |
| partitionBy(part: K => int) | RDD[(K,V)] → RDD[(K,V)] |
| reduceByKey(f: (V, V) => V) | RDD[(K,V)] → RDD[(K,V)] |
| keys() | RDD[(K,V)] → RDD[K] |
| values() | RDD[(K,V)] → RDD[V] |

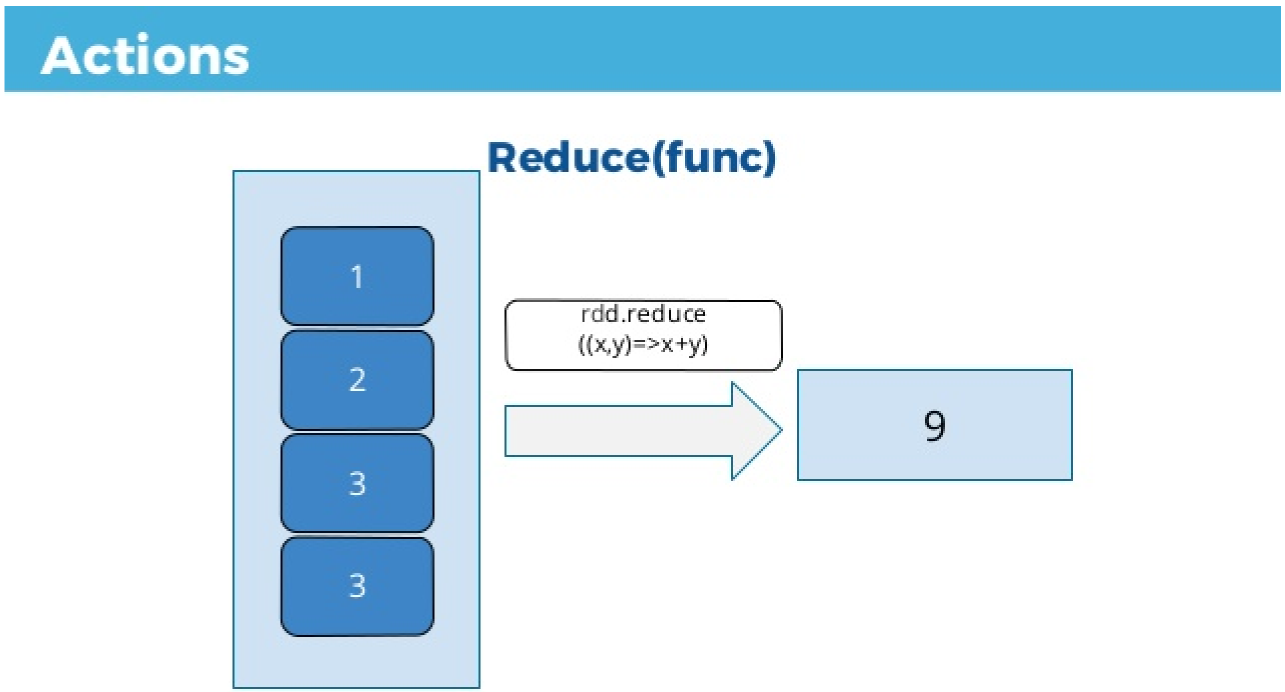
# Transformations: join & grouping

| Transformation | Types |
| --- | --- |
| join(RDD[(K, W]) | (RDD[(K,V)], RDD[(K,W)]) → RDD[(K,(V,W))] |
| leftOuterJoin(RDD[(K, W]) | (RDD[(K,V)], RDD[(K,W)]) → RDD[(K,(V,W))] |
| rightOuterJoin(RDD[(K, W]) | (RDD[(K,V)], RDD[(K,W)]) → RDD[(K,(V,W))] |
| fullOuterJoin(RDD[(K, W]) | (RDD[(K,V)], RDD[(K,W)]) → RDD[(K,(V,W))] |
| cogroup(RDD[(K, W]) | (RDD[(K,V)], RDD[(K,W)]) → RDD[(K,(Seq[V],Seq[W]))] |
| subtractByKey(RDD[(K, W]) | (RDD[(K,V)], RDD[(K,W)]) → RDD[(K,V)] |

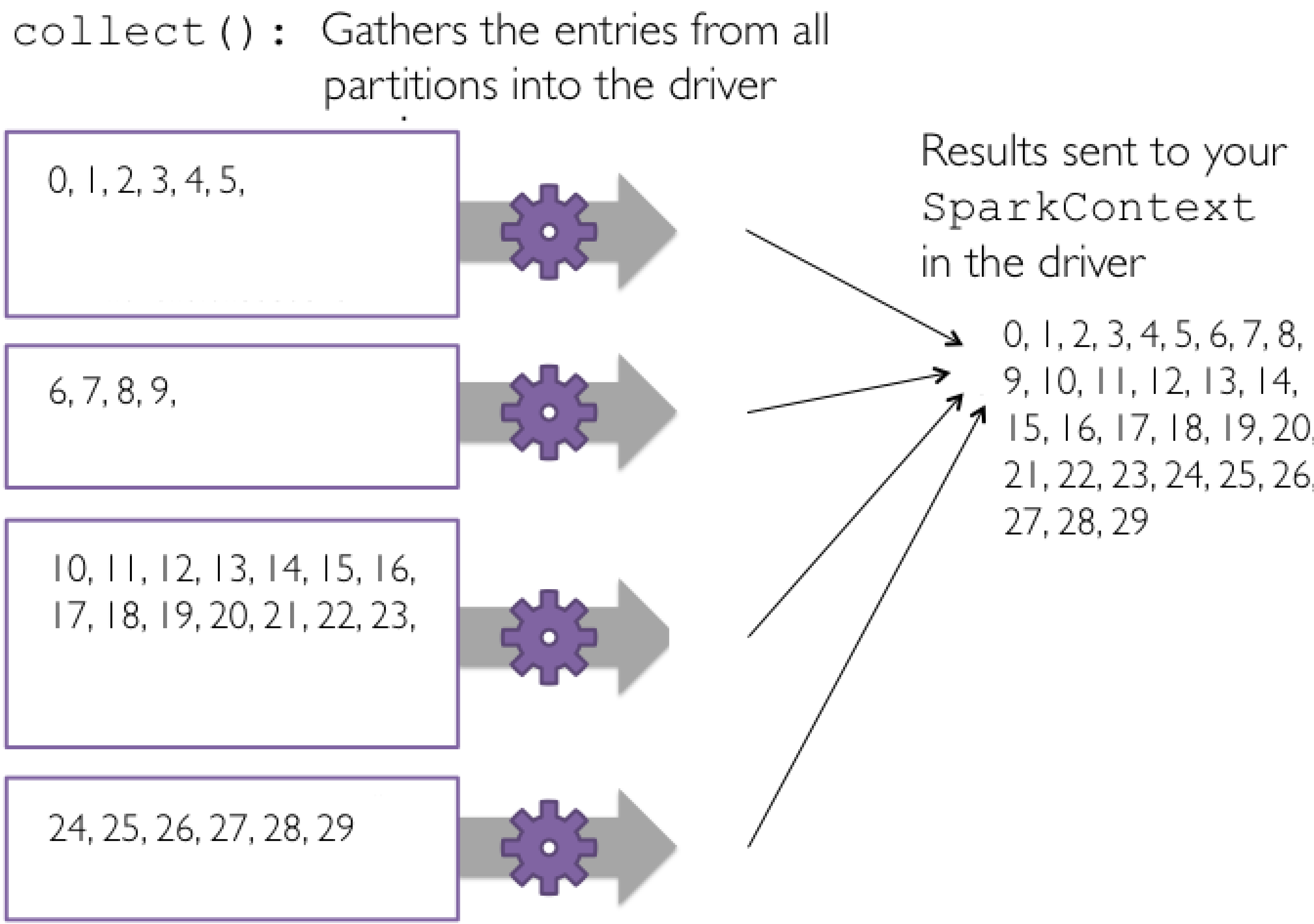
# Actions

| Action | Types |
| --- | --- |
| count() | RDD[T] → Long |
| reduce(f: (T, T) => T) | RDD[T] → T |
| treeReduce(f: (T, T) => T) | RDD[T] → T |
| aggregate(zero: U) (f: (U,T) => U, g: (U,U) => U) | RDD[T] → U |
| treeAggregate(zero: U) (f: (U,T) => U, g: (U,U) => U) | |
| first() | RDD[T] → T |
| take() | RDD[T] → Array[T] |
| takeSample() | RDD[T] → Array[T] |
| max() | RDD[T] → T |

# Actions



# Actions



collect(): Gathers the entries from all partitions into the driver
0, 1, 2, 3, 4, 5,
6, 7, 8, 9,
10, 11, 12, 13, 14, 15, 16,
17, 18, 19, 20, 21, 22, 23,
24, 25, 26, 27, 28, 29
Results sent to your SparkContext in the driver
0, 1, 2, 3, 4, 5, 6, 7, 8,
9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19, 20,
21, 22, 23, 24, 25, 26,
27, 28, 29

# Actions

| Action | Types |
|---|---|
| countByKey() | RDD[(K,V)] → Map[K, Long] |
| collectAsMap() | RDD[(K,V)] → Map[K, V] |
| lookup(K) | RDD[(K,V)] → Seq[K, V] |

# Example : wordcount

```
val textFile = spark.textFile("hdfs://input/path")

val words = textFile.flatMap(line => line.split("\\s+"))
val oneCounts = words.map(word => (word, 1))
val counts = oneCounts.reduceByKey(_ + _)

val counts.saveAsTextFile("hdfs://output/path")
```

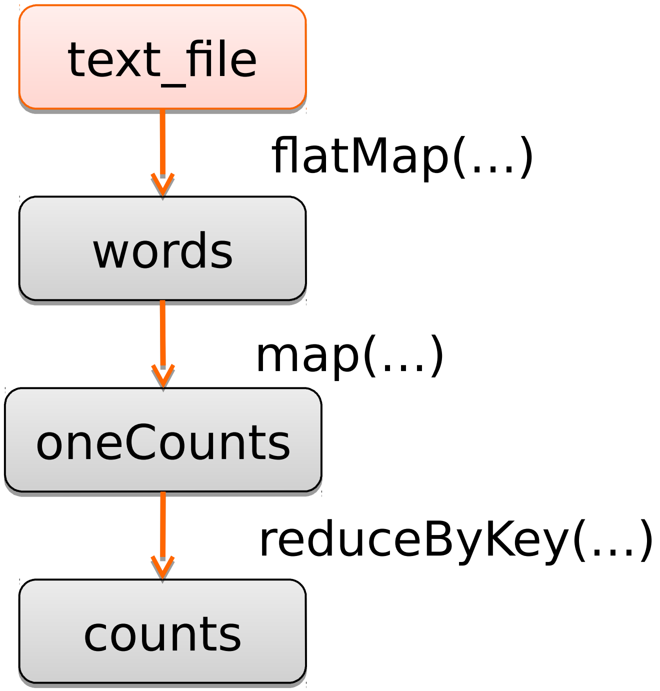
# Example : wordcount

```
val textFile = spark.textFile("hdfs://input/path")

val counts = textFile.flatMap(line => line.split("\\s+"))
                     .map(word => (word, 1))
                     .reduceByKey(_ + _)

val counts.saveAsTextFile("hdfs://output/path")
```

# Example : wordcount



# 4. Datasets

- Foundational type of the structured APIs
- A strictly Java Virtual Machine language feature that works only with Scala and Java
- When to use Datasets?
  - When the operation(s) you would like to perform cannot be expressed using dataframe manipulation
  - When you want or need type-safety
  - The cost of performance

# Creating Datasets

```
In Java: Encoders

import org.apache.spark.sql.Encoders;
public class Flight implements Serializable{
String DEST_COUNTRY_NAME;
String ORIGIN_COUNTRY_NAME;
Long DEST_COUNTRY_NAME;
}
Dataset<Flight> flights = spark.read
.parquet("/data/flight-data/parquet/2010-
summary.parquet/")
.as(Encoders.bean(Flight.class));
```

# Creating Datasets

```
In Scala: Case Classes

case class Flight(DEST_COUNTRY_NAME: String,
ORIGIN_COUNTRY_NAME: String, count: BigInt)
val flightsDF = spark.read
.parquet("/data/flight-data/parquet/2010-
summary.parquet/")
val flights = flightsDF.as[Flight]
```

# Creating Datasets

- To create Datasets in Scala, you define a Scala case class. A case class is a regular class
- That has the following characteristics:
  - Immutable
  - Decomposable through pattern matching
  - Allows for comparison based on structure instead of reference
  - Easy to use and manipulate

# Operations

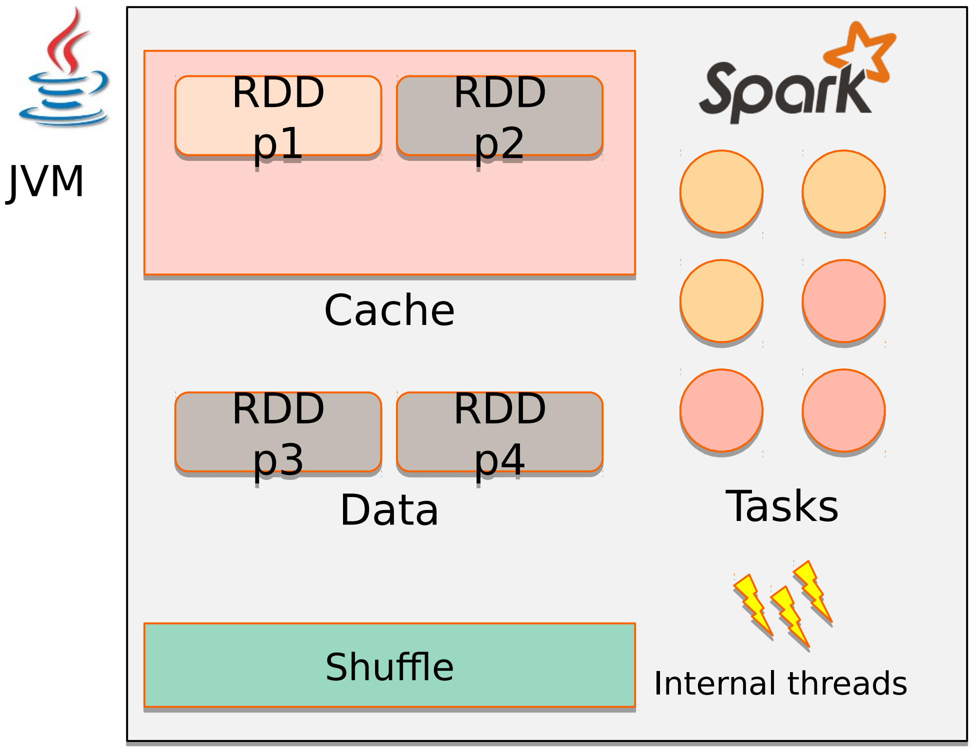
- Actions
  - Collect
  - Take
  - Count

- Transformations
  - Filtering
  - Mapping
  - Mapping example

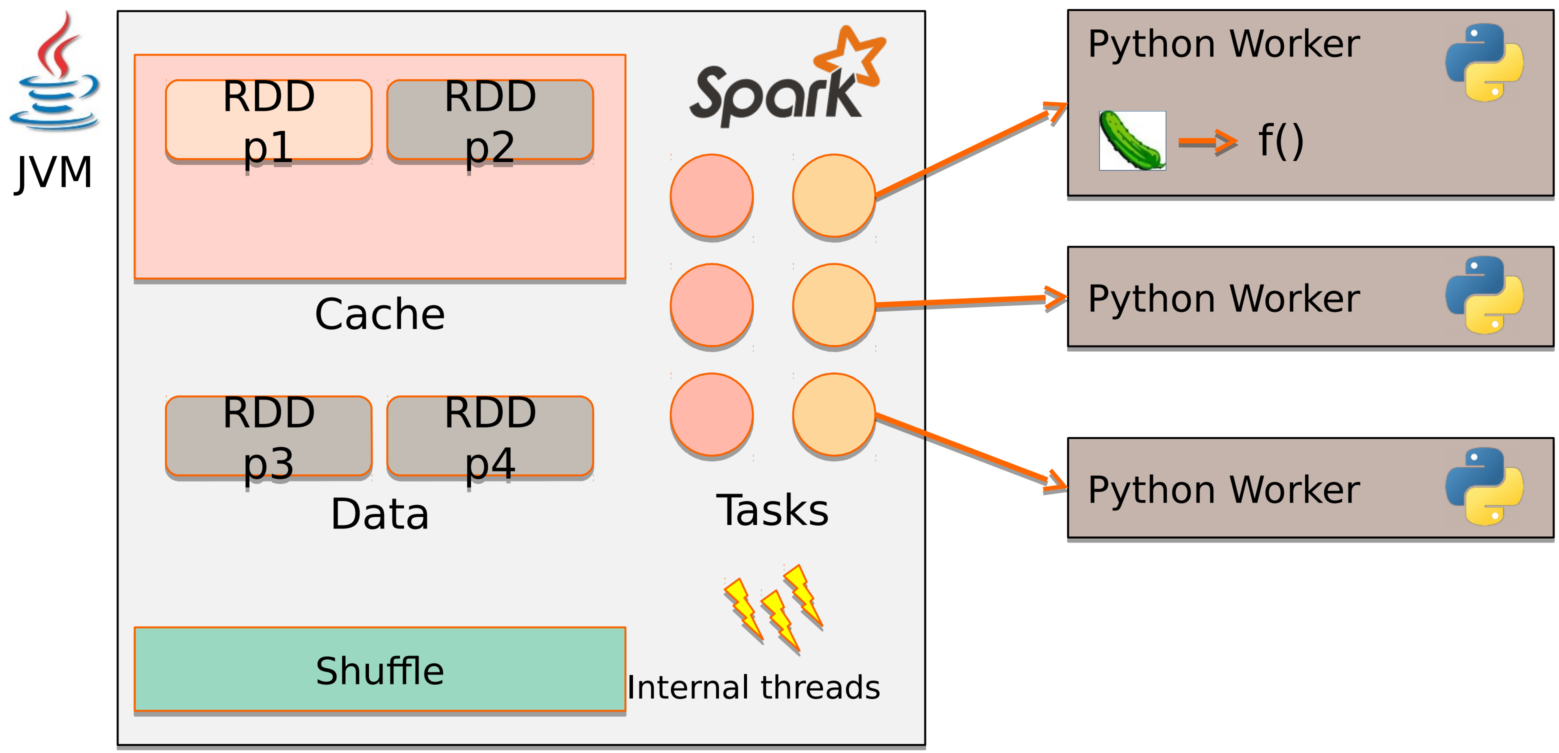
```
val destinations = flights.map(f =>
f.DEST_COUNTRY_NAME)
```

- Joins
- Grouping and Aggregations

# 5. Executor Architecture



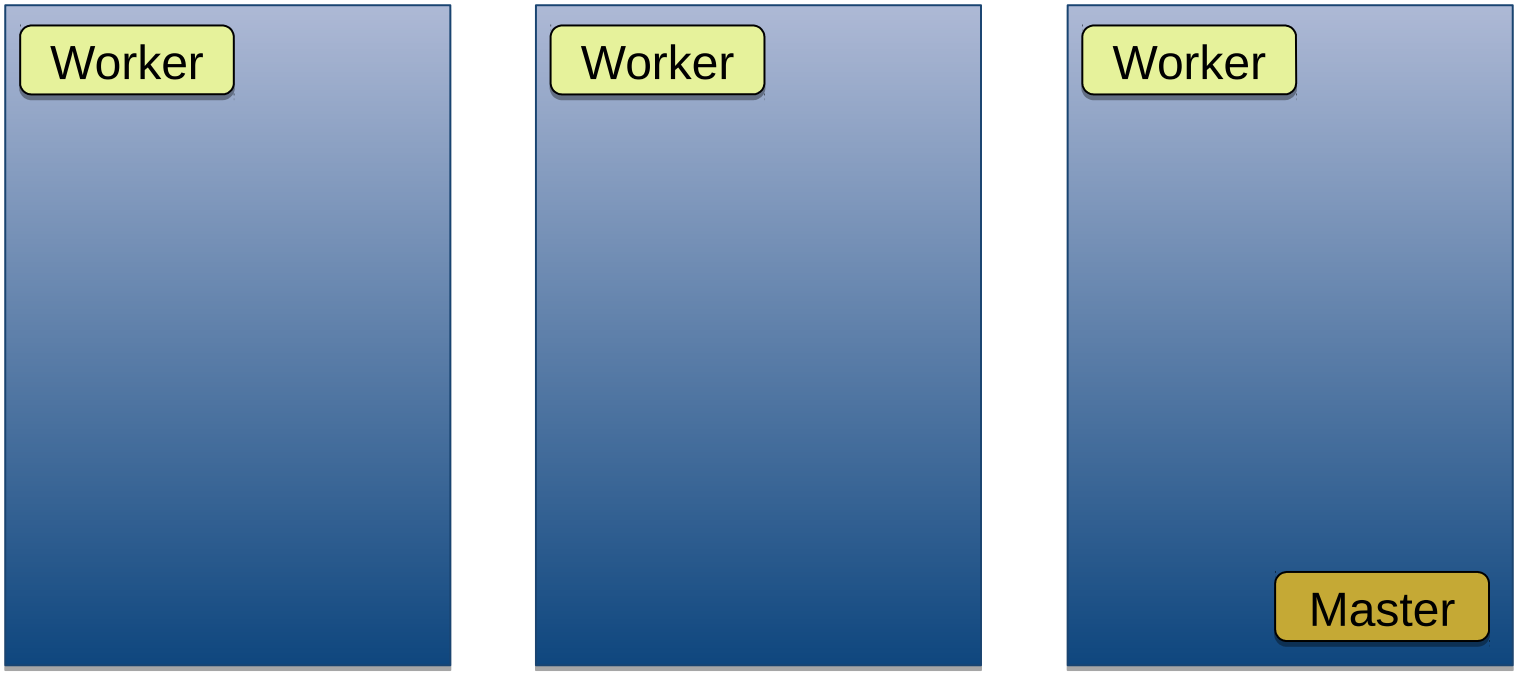
# Executor with PySpark



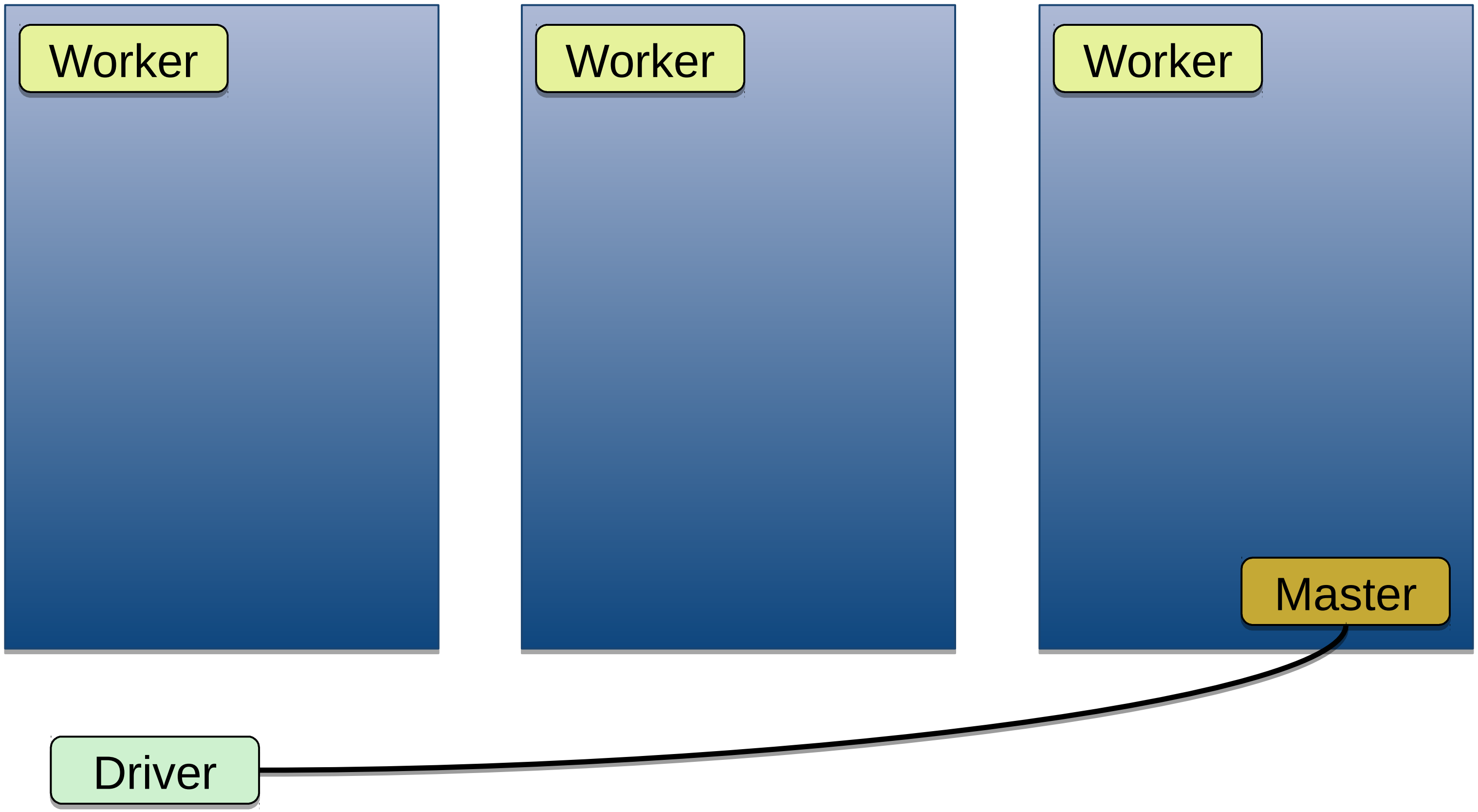
# Deployment

- Roles
  - Coordinator ▭ Driver (SparkContext)
  - Executor
- Deployment
  - Local
    - One executor (driver inside)
  - Distributed
    - Driver + several executors
- Cluster manager maintains a cluster of machines that runs Spark applications
  - Standalone
  - YARN
  - Mesos

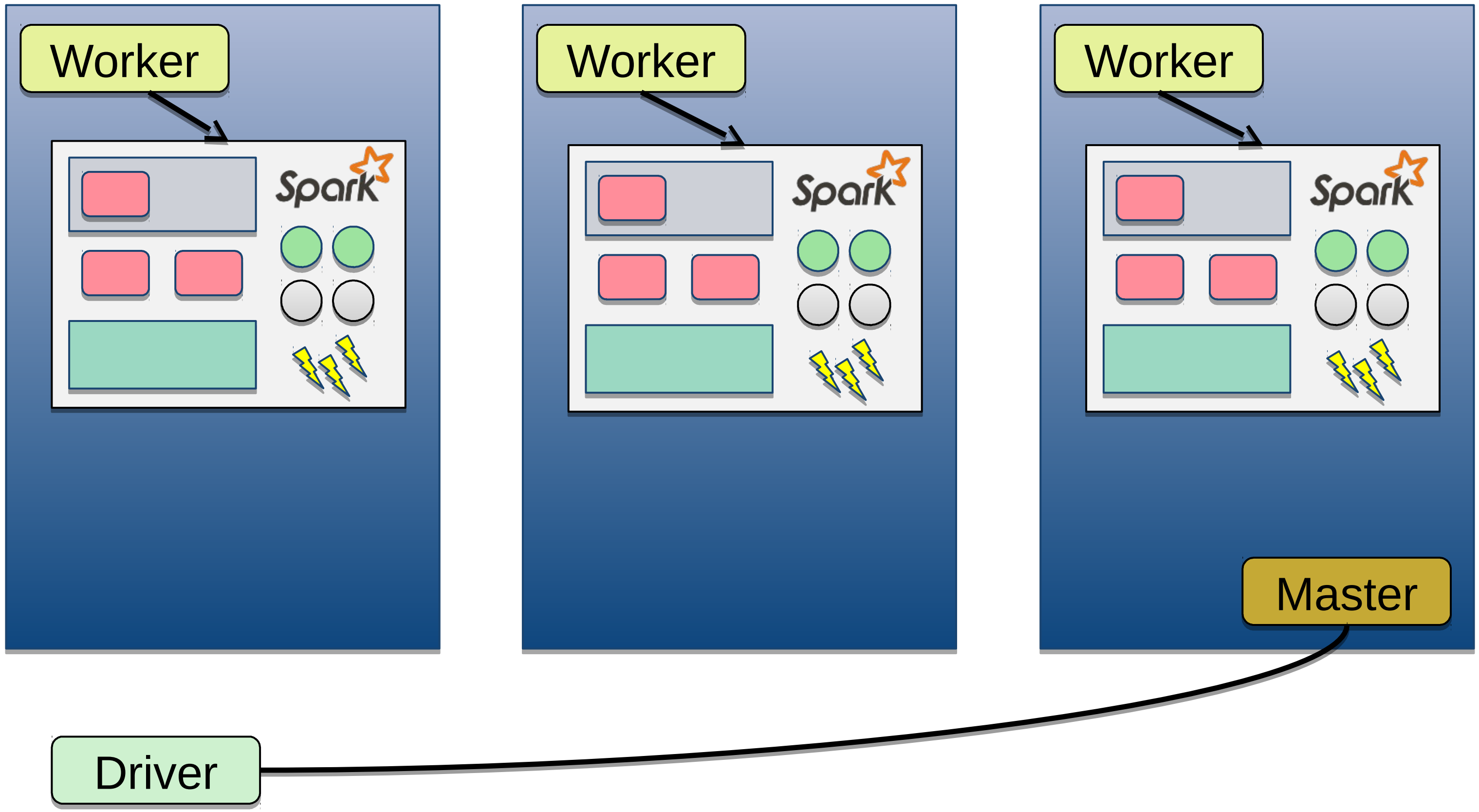
# Spark Standalone



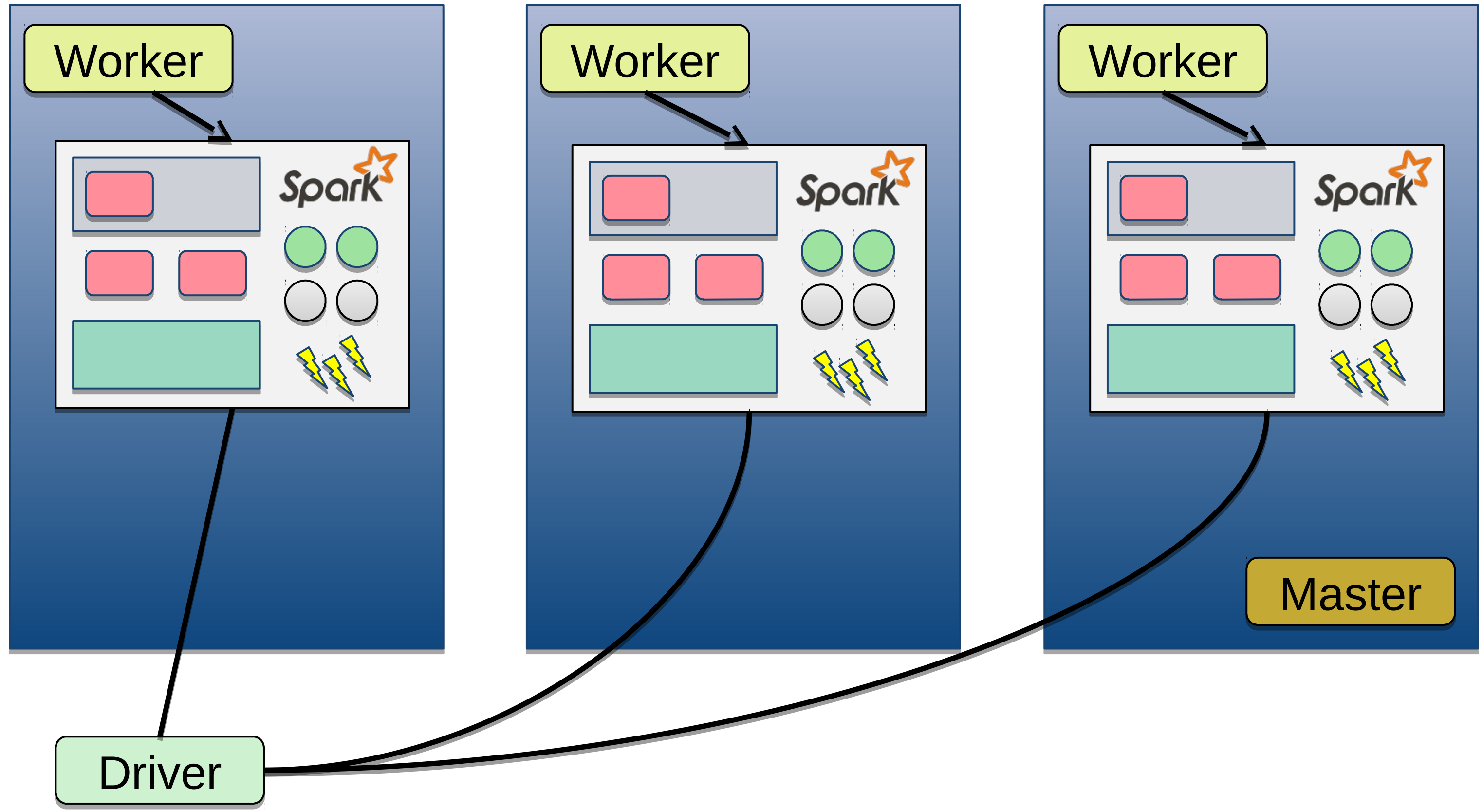
# Spark Standalone



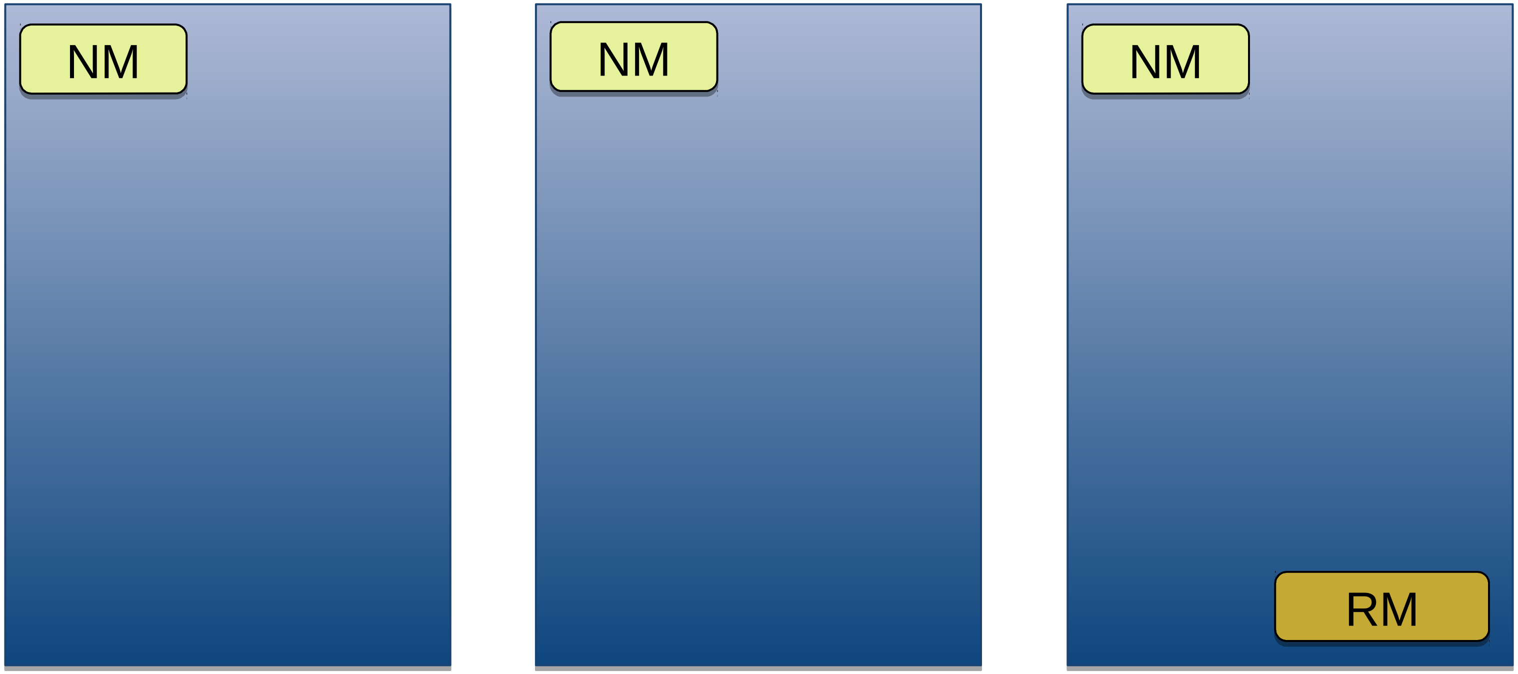
# Spark Standalone



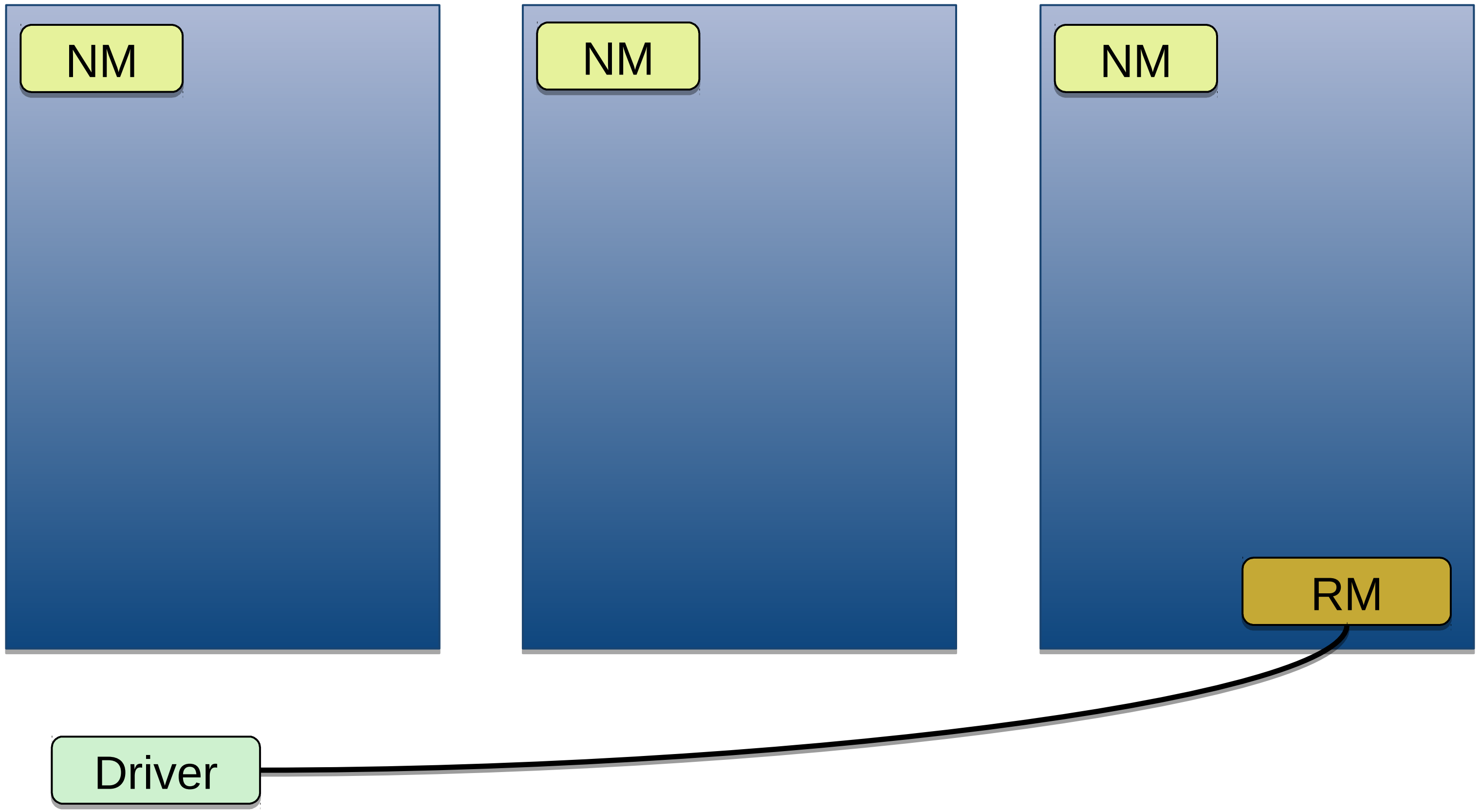
# Spark Standalone



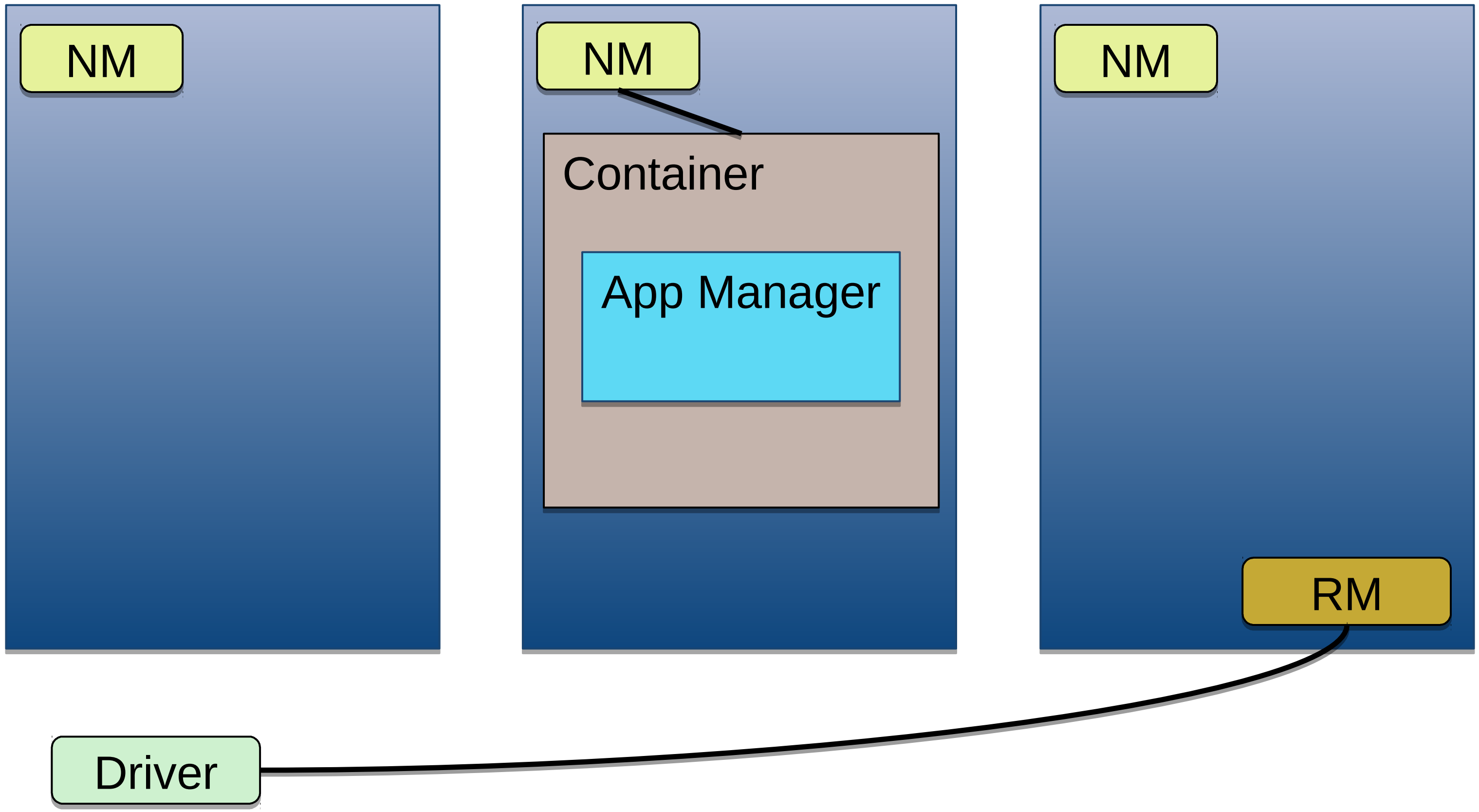
# Spark YARN



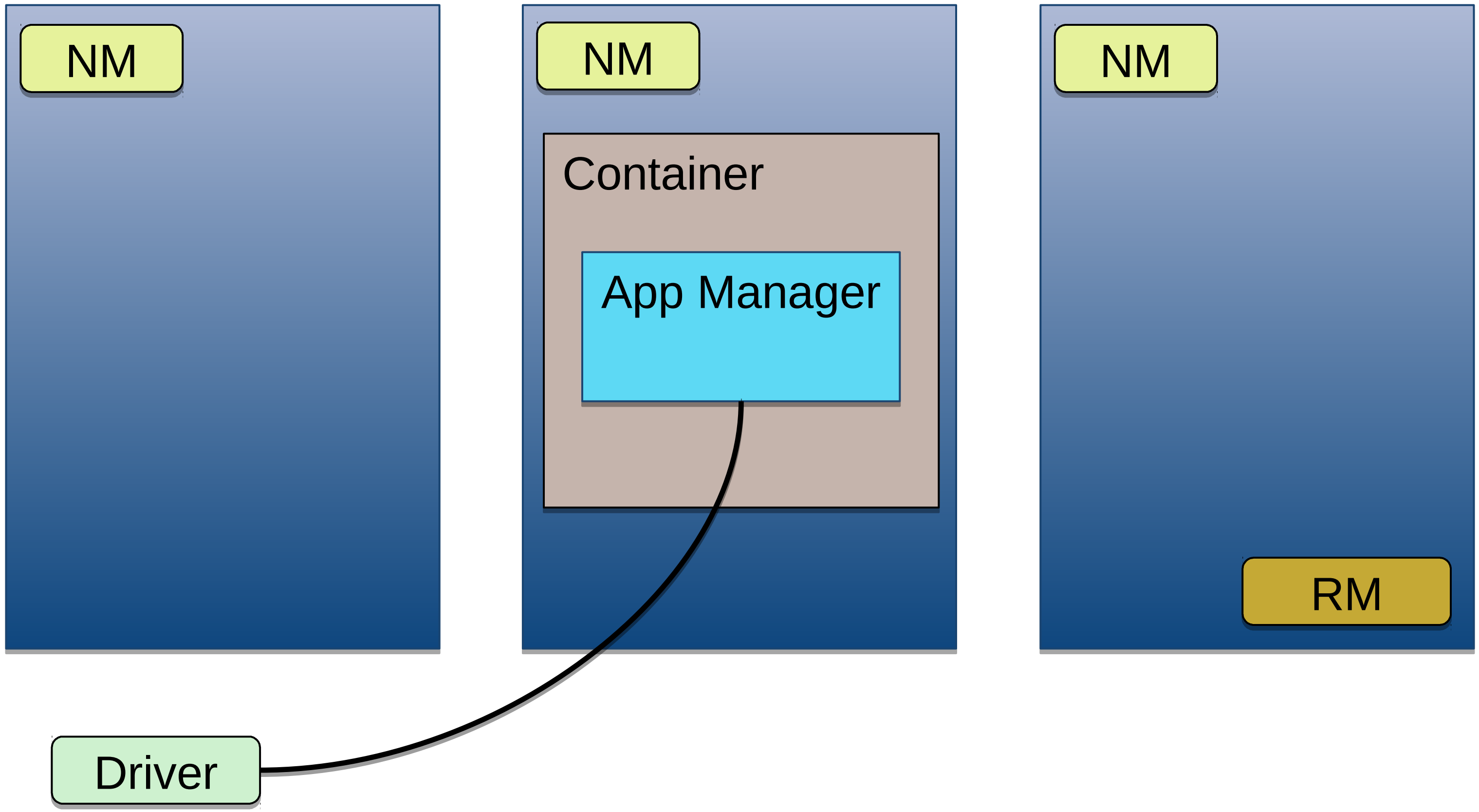
# Spark YARN



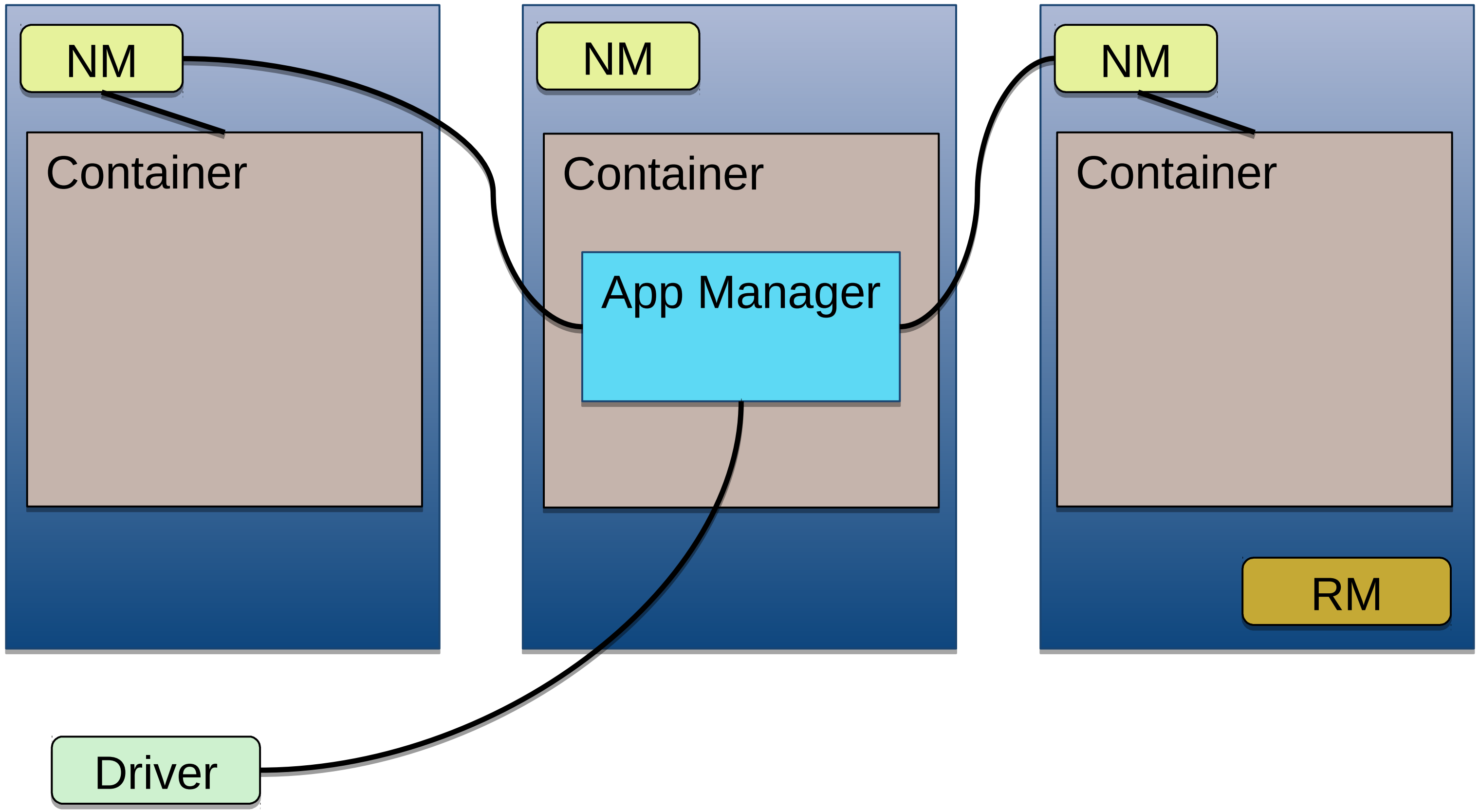
# Spark YARN



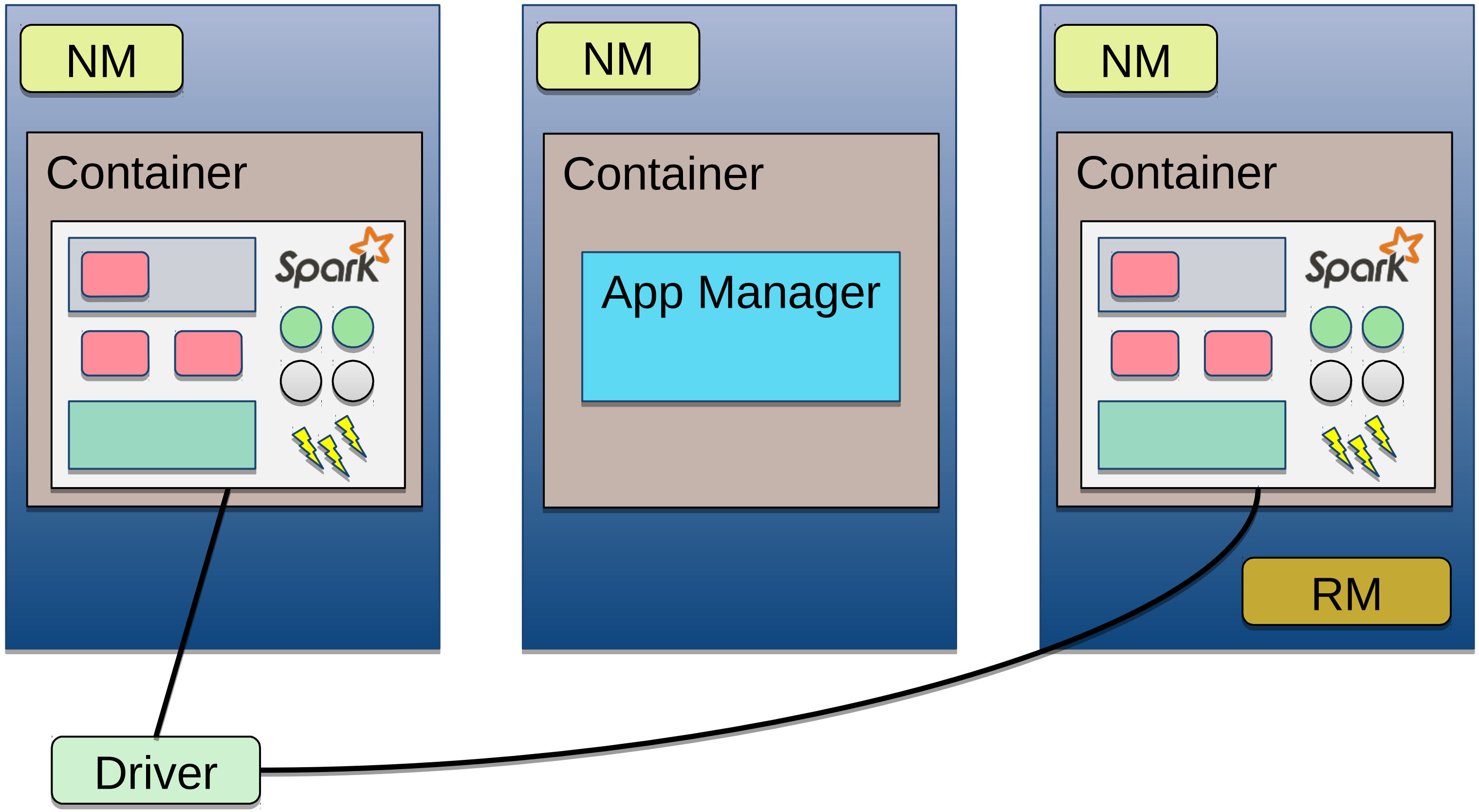
# Spark YARN



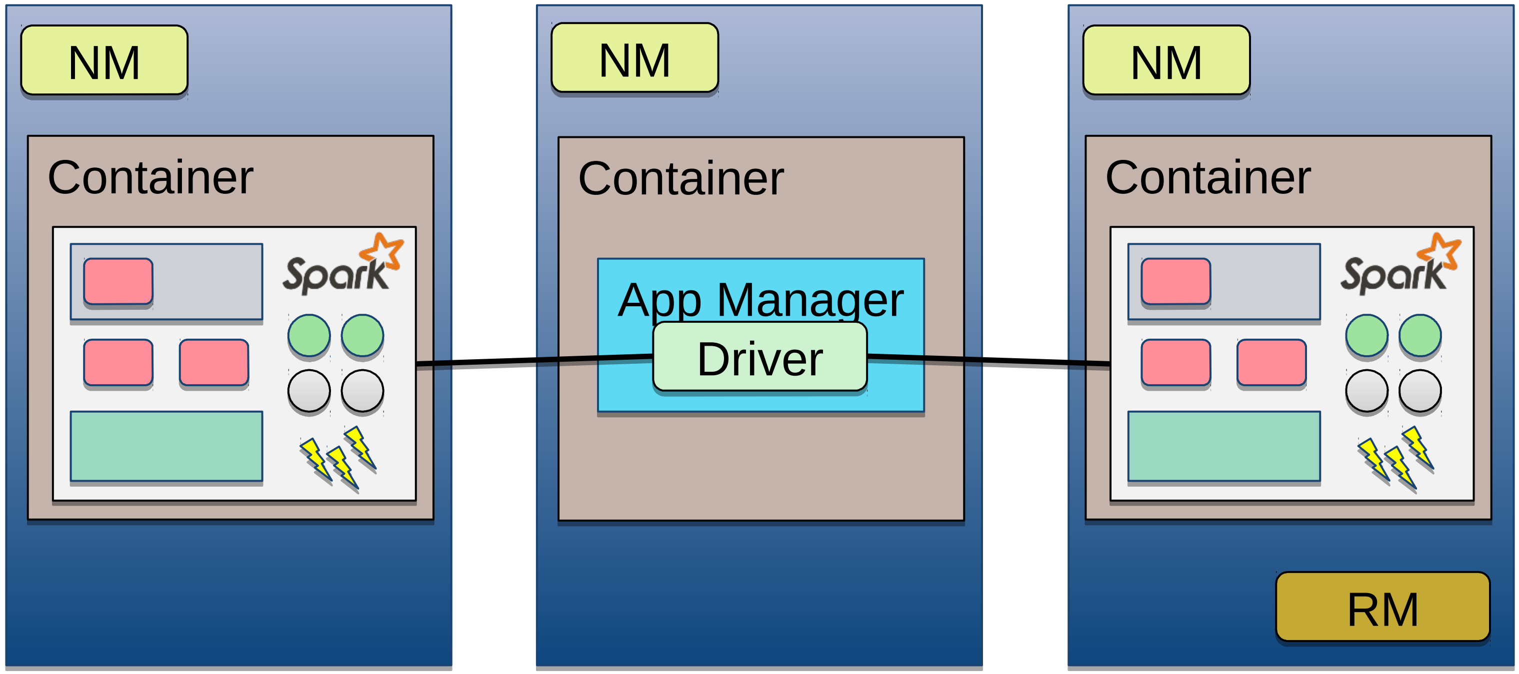
# Spark YARN


NM
Container
NM
Container
App Manager
NM
Container
RM
Driver

# Spark YARN (yarn-client)

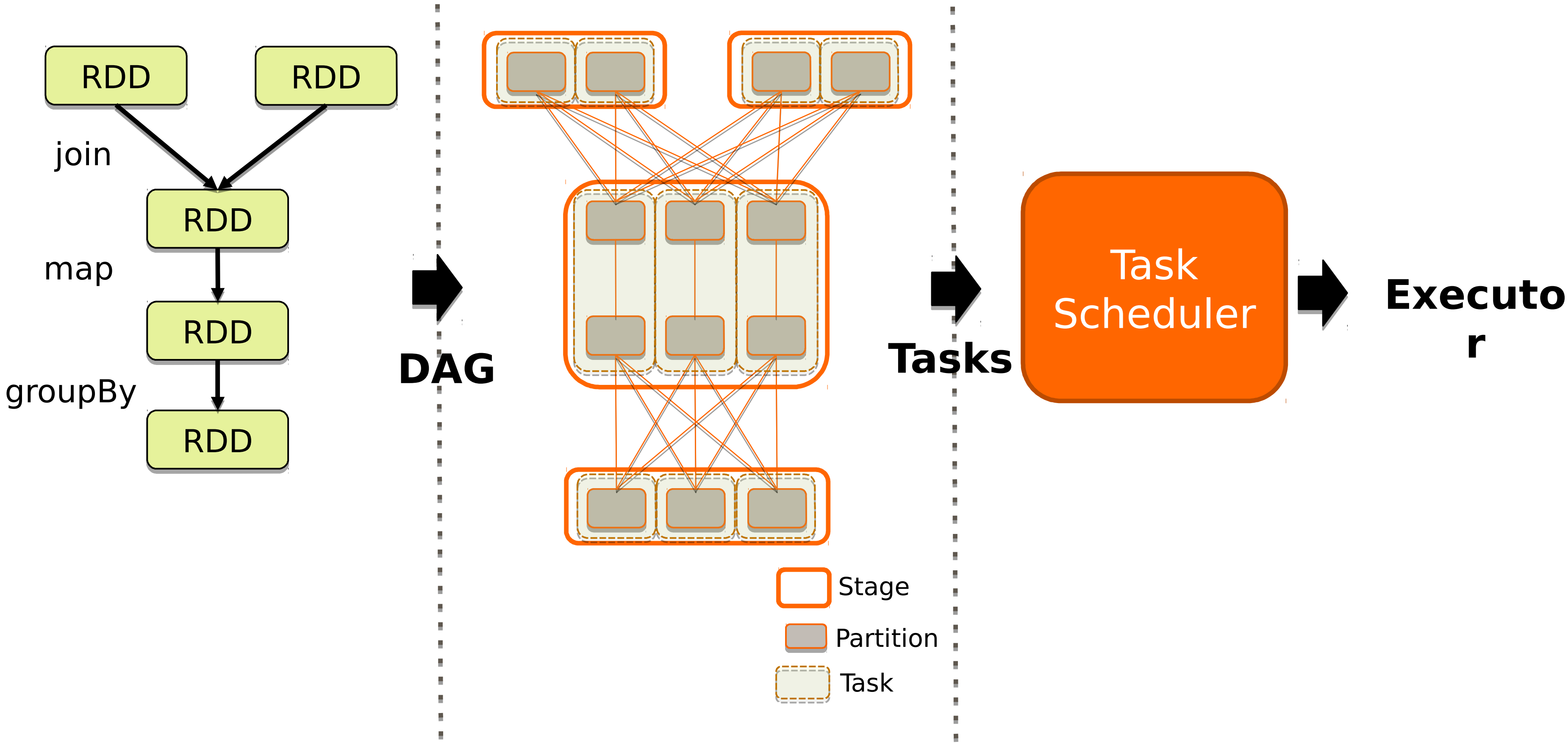


# Spark YARN (yarn-cluster)



# 6. Operations

## Scheduling




Build Operator DAG

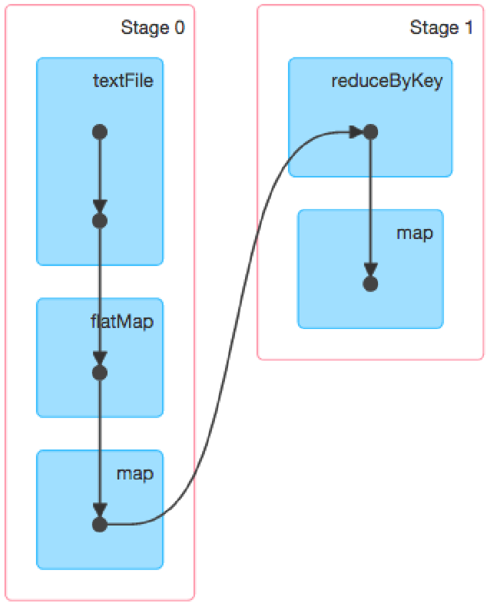
Split DAG into stages
Submit each stage when ready

Launch individual tags
Retry failed tags

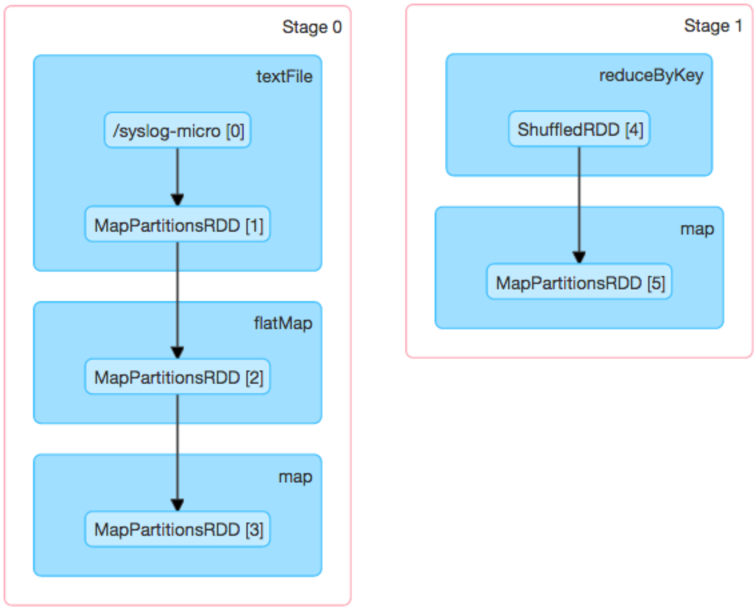
# Scheduling

```
val logMessages = sc.textFile("hdfs://input/path")
val logEntries = logMessages.flatMap(Syslog.parseLine)
val wins = logEntries.map(logEntry => (wgs.winID(logEntry), new LogWinStats(logEntry)))
                 .reduceByKey(_ + _)
                 .map{ case (winID, winStats) => LogWindow(winID, winStats) }
wins.count()
```

DAG



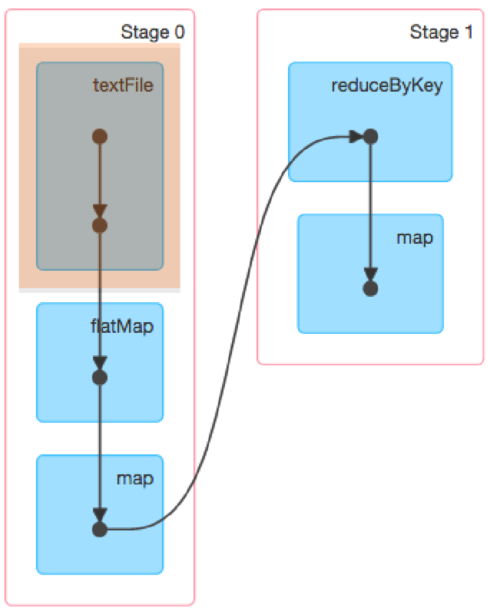

Stages



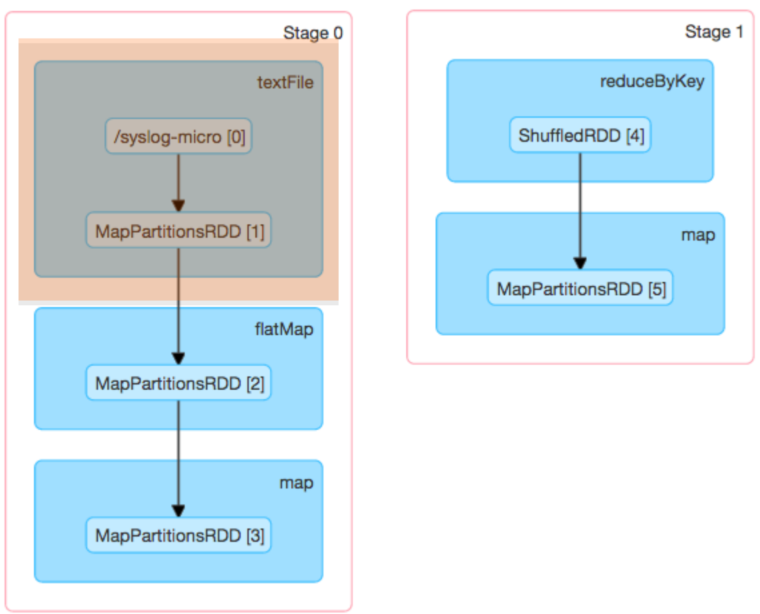
# Scheduling

```
val logMessages = sc.textFile("hdfs://input/path")
val logEntries = logMessages.flatMap(Syslog.parseLine)
val wins = logEntries.map(logEntry => (wgs.winID(logEntry), new LogWinStats(logEntry)))
                 .reduceByKey(_ + _)
                 .map{ case (winID, winStats) => LogWindow(winID, winStats) }
wins.count()
```

DAG

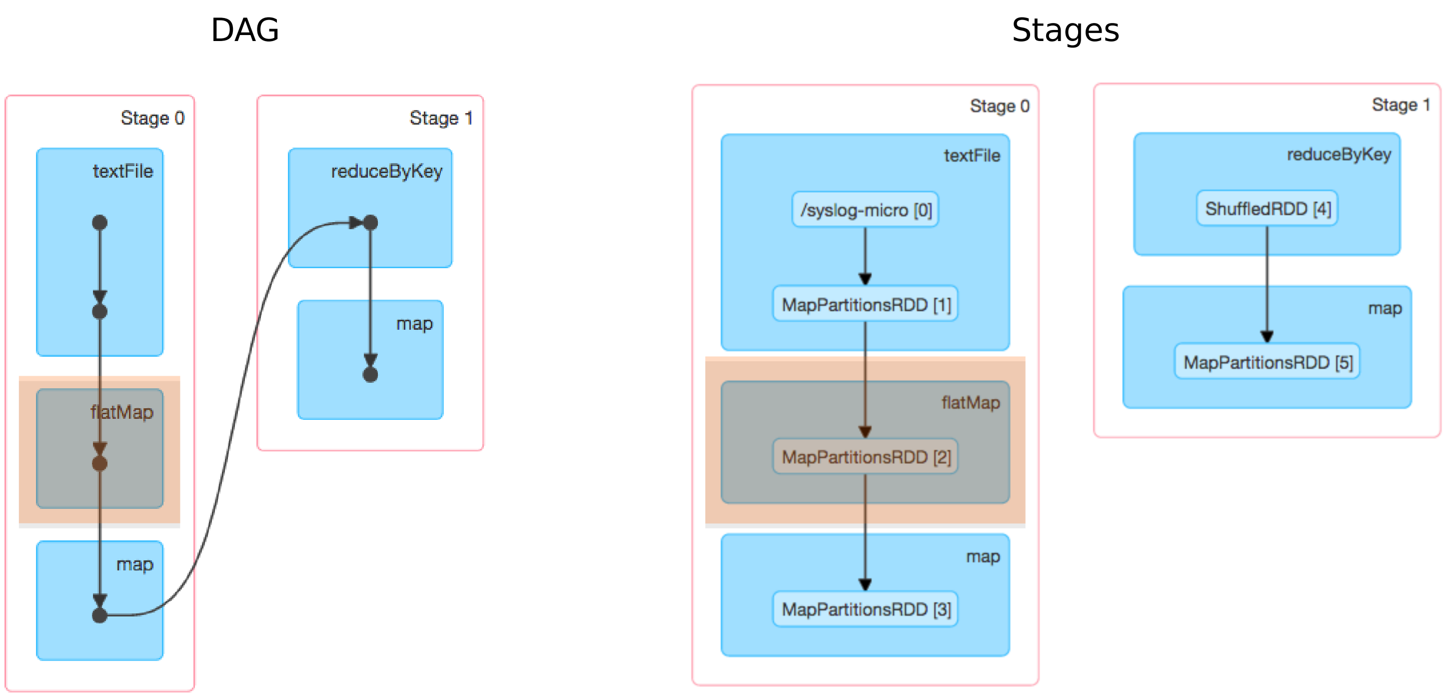



Stages



# Scheduling

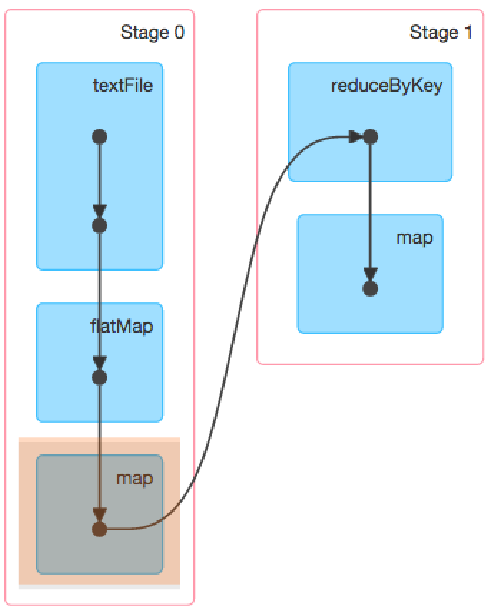
```
val logMessages = sc.textFile("hdfs://input/path")
val logEntries = logMessages.flatMap(Syslog.parseLine)
val wins = logEntries.map(logEntry => (wgs.winID(logEntry), new LogWinStats(logEntry)))
               .reduceByKey(_ + _)
               .map{ case (winID, winStats) => LogWindow(winID, winStats) }
wins.count()
```



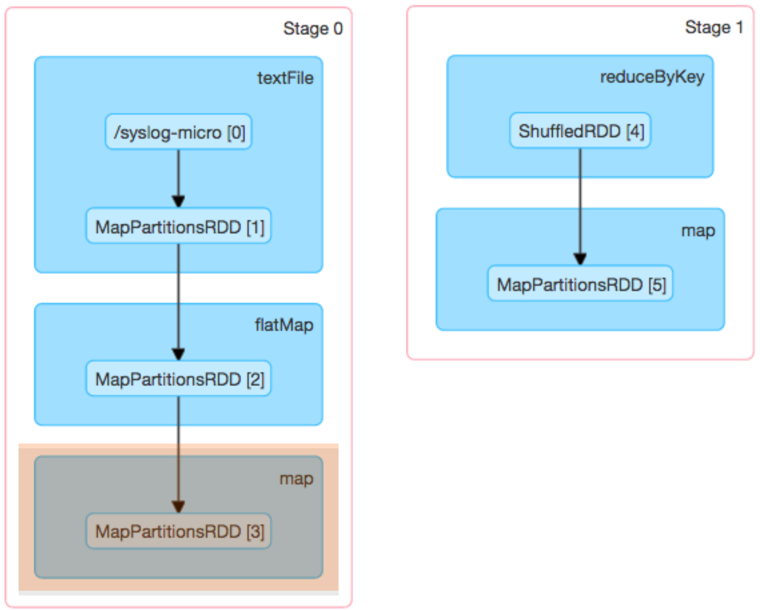
# Scheduling

```
val logMessages = sc.textFile("hdfs://input/path")
val logEntries = logMessages.flatMap(Syslog.parseLine)
val wins = logEntries.map(logEntry => (wgs.winID(logEntry), new
LogWinStats(logEntry)))
           .reduceByKey(_ + _)
           .map{ case (winID, winStats) => LogWindow(winID, winStats) }
wins.count()
```

DAG



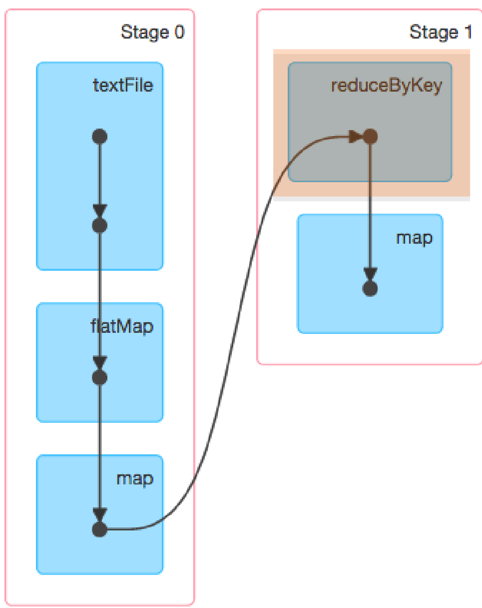

Stages



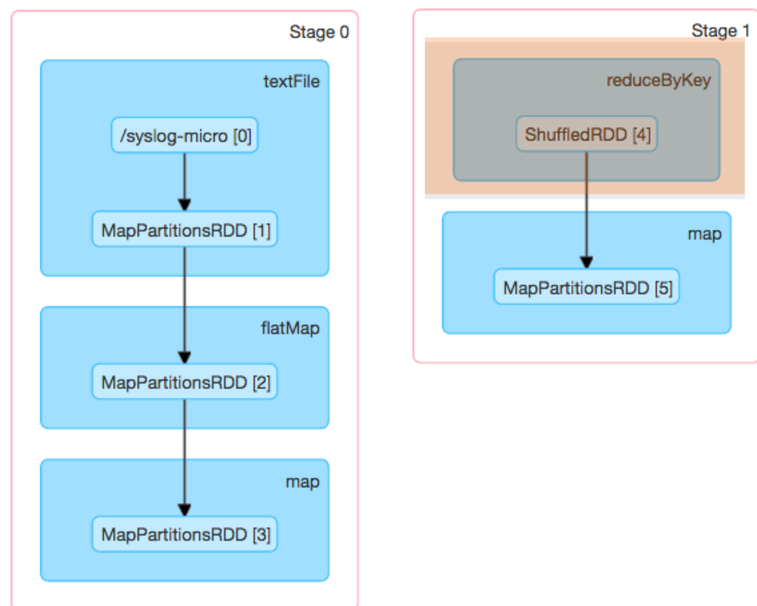
# Scheduling

```
val logMessages = sc.textFile("hdfs://input/path")
val logEntries = logMessages.flatMap(Syslog.parseLine)
val wins = logEntries.map(logEntry => (wgs.winID(logEntry), new LogWinStats(logEntry)))
                .reduceByKey(_ + _)
                .map{ case (winID, winStats) => LogWindow(winID, winStats) }
wins.count()
```

DAG



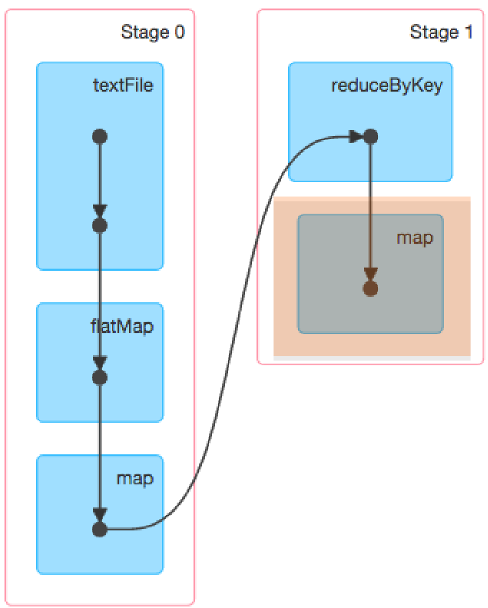

Stages



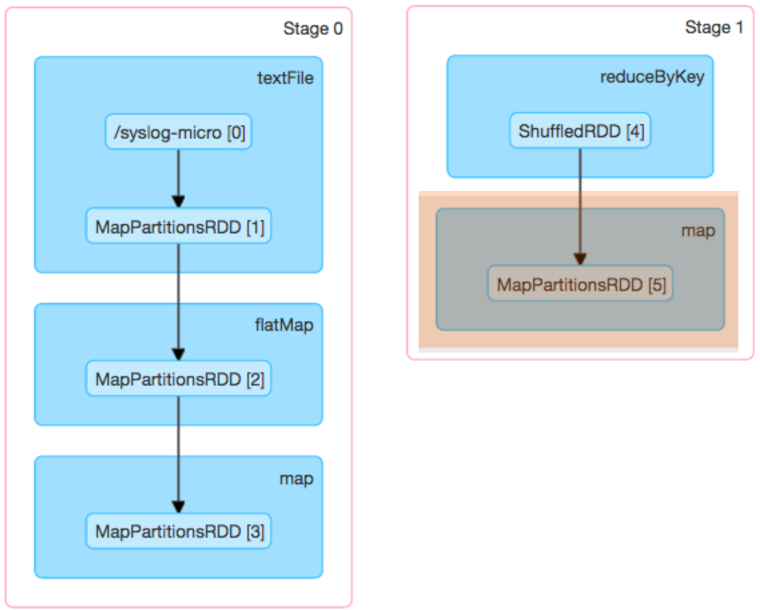
# Scheduling

```
val logMessages = sc.textFile("hdfs://input/path")
val logEntries = logMessages.flatMap(Syslog.parseLine)
val wins = logEntries.map(logEntry => (wgs.winID(logEntry), new LogWinStats(logEntry)))
              .reduceByKey(_ + _)
              .map{ case (winID, winStats) => LogWindow(winID, winStats) }
wins.count()
```

DAG




Stages



# Shuffle

- Represents a physical repartitioning of the data, for instance
  - Sorting a DataFrame
  - Joins
  - Aggregations
  - Grouping data that was loaded from a file by key
- Separates stages in the DAG
- May be triggered by
  - Repartitions
  - *ByKey and *By operations
  - Join
- It takes into account RDD partitioning
  - Narrow dependencies
  - Wide dependencies

# Shuffle

- Sorting is only performed if required in reduce task
- *Hash Shuffle*
  - Each map creates one file per reducer
  - Records are appended to each file
- *Sort shuffle* (default from 1.4)
  - One file per map task
  - File ordered by reducer id and indexed
- *Tungsten shuffle*
  - Operates directly on serialize data
  - To be used only under certain conditions

# Hash Shuffle

- Each map creates one file for each reducer
  - Files = M * R
- Each record is written directly to target file
- Optimization – File consolidation
  - A pool of output files created by executor
  - Each map tasks requests R files
- Pros / Cons
  - ++ fast / no sort, no memory overhead, no hash table
  - -- # partitions ⇒ a lot of files, random IO

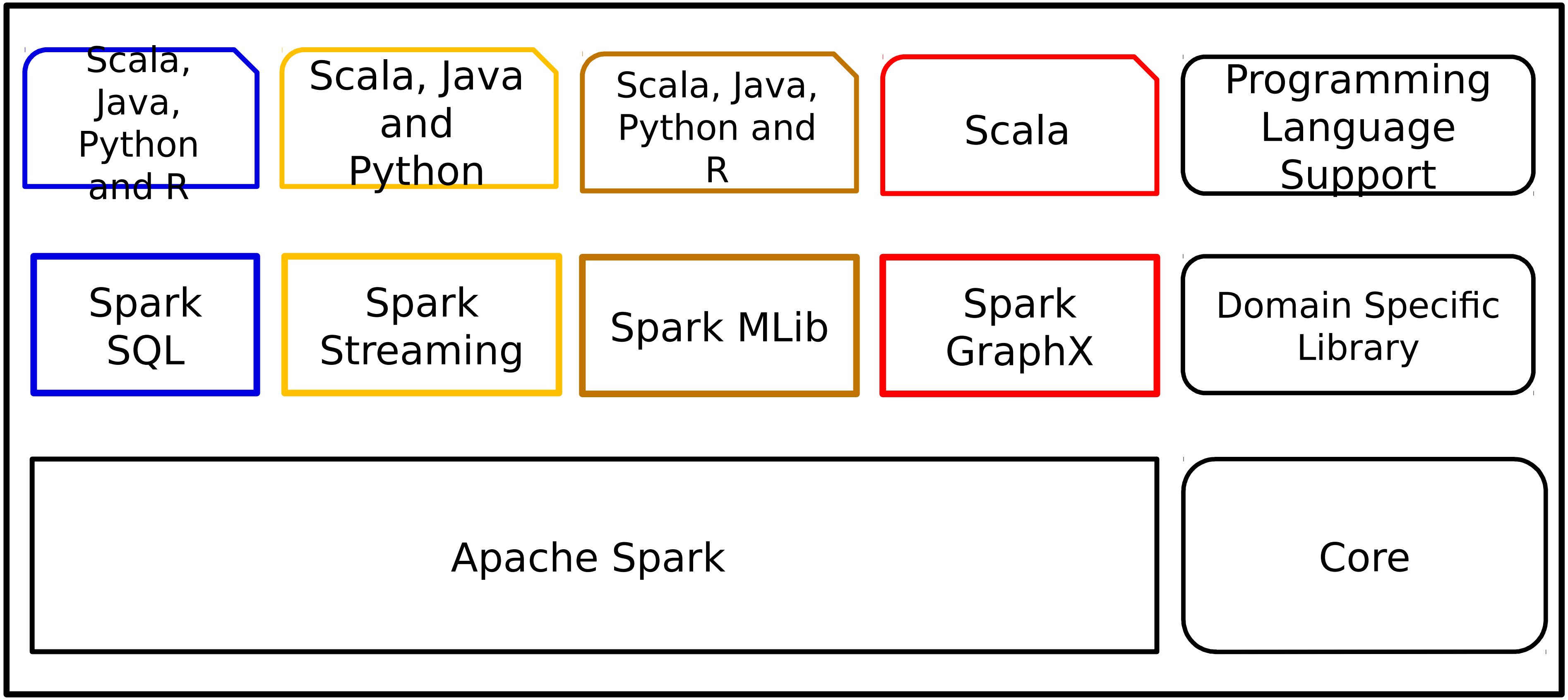
# Sort Shuffle

- Similar to MapReduce shuffle
  - One file per map task
  - File ordered by reducer id and indexed
  - If not enough memory → spill to disk
  - BUT results are not merged in reduce side
- Special implementation of hash table
  - Allows combine in place
- Merging is only done when request from reducerPros / Cons
  - ++ less # of files, less # of random IO
  - -- sorting slower than hashing, SSDs favor hash-based

# Tungsten Shuffle

- Operates directly on serialized binary data
  - Direct spilling
  - May merge spills by direct concatenation
- Uses special cache-efficient sorter
- To be used only when
  - No aggregation
  - Codec supports relocation of serialized values
  - Less than 16777216 partitions
  - Record < 128MB

# 7. Tools


Scala, Java, Python and R
Scala, Java and Python
Scala, Java, Python and R
Scala
Programming Language Support
Spark SQL
Spark Streaming
Spark MLib
Spark GraphX
Domain Specific Library
Apache Spark
Core

# SparkSQL

- Provides the ability to run SQL on top of Spark
- ASpark module for structured data processing
- Defines an interface for a semi-structured data type called Dataframes and a typed version (Datasets)
  - APIs as well support for basic SQL queries
- The Spark SQL library enables
  - Relational processing both within Spark programs and on external data sources using a programmer-friendly API
  - High performance using established DBMS techniques
  - Support of new data sources, including semi structured data and external databases
  - Enables extension with advanced analytics algorithms such as graph processing and machine learning

# SparkSQL Example

Create a DataFrame, manipulate it with SQL, and then manipulate it again as a DataFrame

```
# in Python
 # spark is an existing SparkSession
    spark.sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING) USING hive")
    spark.sql("LOAD DATA LOCAL INPATH 'examples/src/main/resources/kv1.txt' INTO TABLE src")

 # Queries are expressed in HiveQL
    spark.sql("SELECT * FROM src").show()
    # +---+-------+
    # |key|  value|
    # +---+-------+
    # |238|val_238|
    # | 86| val_86|
    # |311|val_311|
```

# Dataframes

- Structured API which represents a table of data with rows and columns
- Distributed collection of data organized into named columns
- DF Creation
  - From existing RDD
    - Schema inference
    - Schema specification
  - Data sources
    - Parquet files, JSON, (csv)
    - Hive, JDBC
- DF Manipulation
  - DF Operations
  - SQL queries programmatically

# Dataframes

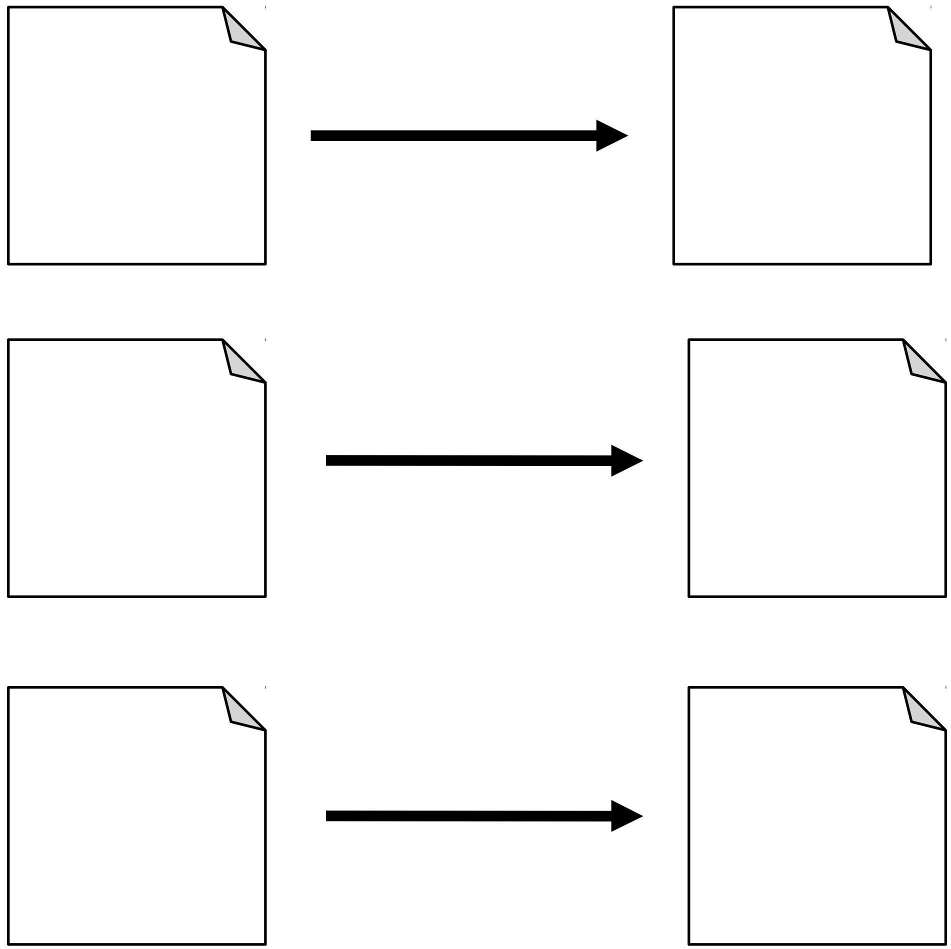
- Dataframe  represents
  - Immutable, lazily evaluated plans
- Specify what operations to apply to data residing at a location to generate some output
- Immutable
  - Spark core data structure cannot changed after there are created
- Lazy evaluation
  - Spark will wait until the last moment  to execute the graph of computation instructions

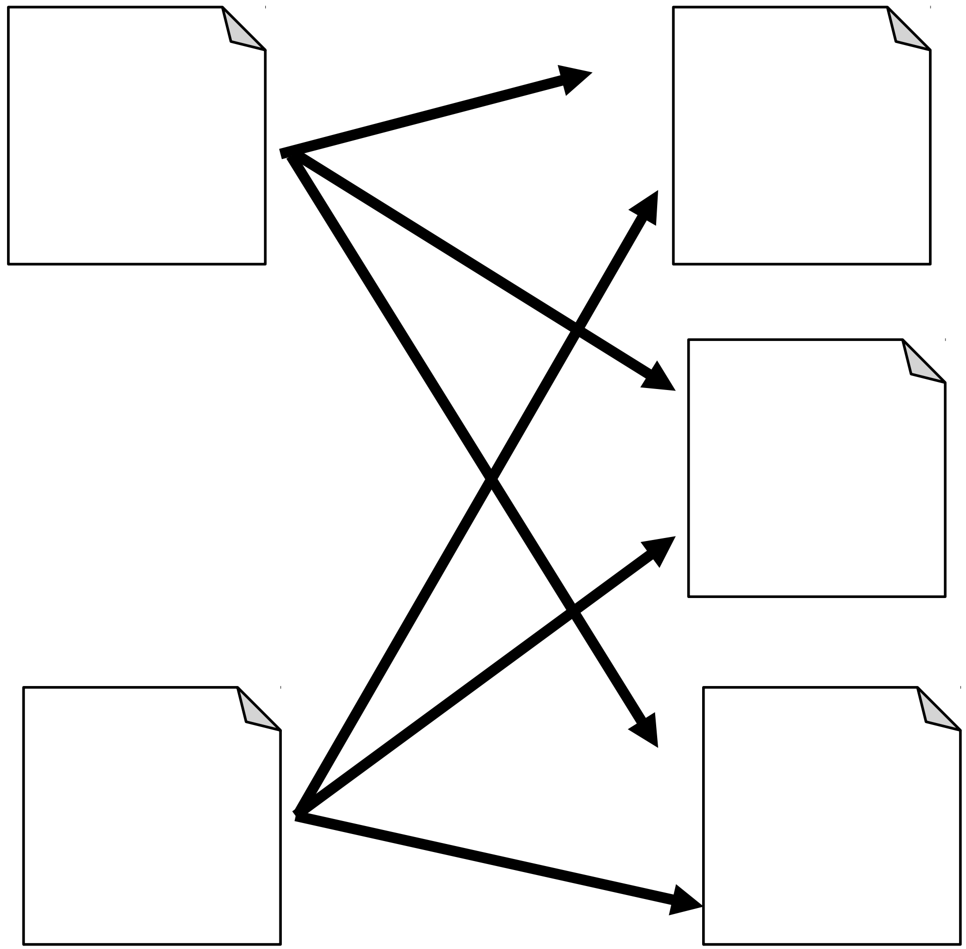
# Dataframe operations

- Transformations: ways of specifying different series of data manipulation.
  - Build up logic transformation plan
  - Narrow dependencies ( each input partition will contribute to only one output partition)
  - Wide dependencies (input partitions contributing to many output partitions)
- Action : instructs Spark to compute a result from a series of transformations (e.g. Count)
  - Actions to view data in the console
  - Actions to collect data to native objects in the respective language
  - Actions to write to output data sources

# Narrow transformations
1 to 1



# Wide transformations
# 1 to N



# Dataframe Example

Schema inference

```
  case class LogEntry(
  timestamp: Long,
  host: String,
  facility: Int,
  severity: Int,
  app: String,
  message: String)

val logMessages = sc.textFile("hdfs://input/path")
val logEntries = logMessages.flatMap(Syslog.parseLine)
val logEntriesDF = logEntries.toDF()
```

# Dataframe Example

## Schema inference

```
logEntriesDF.printSchema()
root
 |-- timestamp: long (nullable = false)
 |-- host: string (nullable = true)
 |-- facility: integer (nullable = false)
 |-- severity: integer (nullable = false)
 |-- app: string (nullable = true)
 |-- message: string (nullable = true)
```

# Dataframe Example

## Querying with DF API

```
val res = logEntriesDF.filter("severity < 3")
                      .groupBy("app")
                      .count()
                      .filter("count > 5")
                      .sort($"count".desc)
res.show(5)
+------------+-----+
|         app|count|
+------------+-----+
|      kernel| 1942|
|lustre-target|  514|
|   OMPI-ERROR|  331|
|       ioadm|  297|
|        crmd|   18|
+------------+-----+
```

# Dataframe Example

## Querying with SQL

```
logEntriesDF.registerTempTable("logEntries")
val res = sqlContext.sql(
  "select app, count(*) as count from logEntries where severity < 3 " +
  "group by app having count > 5 " +
  "order by count desc”
)
res.show(5)
+------------+-----+
|         app|count|
+------------+-----+
|      kernel| 1942|
|lustre-target|  514|
|   OMPI-ERROR|  331|
|       ioadm|  297|
|        crmd|   18|
+------------+-----+
```

# Dataframe Example

Programmatically specifying a schema

```
val schema = StructType(Seq(
   StructField("logCount", LongType),
   StructField("facilityCount", ArrayType(LongType)),
   StructField("severityCount", ArrayType(LongType)),
   StructField("appCount", MapType(StringType, LongType))
))

val windowsRow = windows.map(w => Row(w.stats.logCount,
                                  w.stats.facilityCount,
                                  w.stats.severityCount,
                                  w.stats.appCount))
val windowsDF = sqlContext.createDataFrame(windowsRow, schema)
windowsDF.printSchema()
```

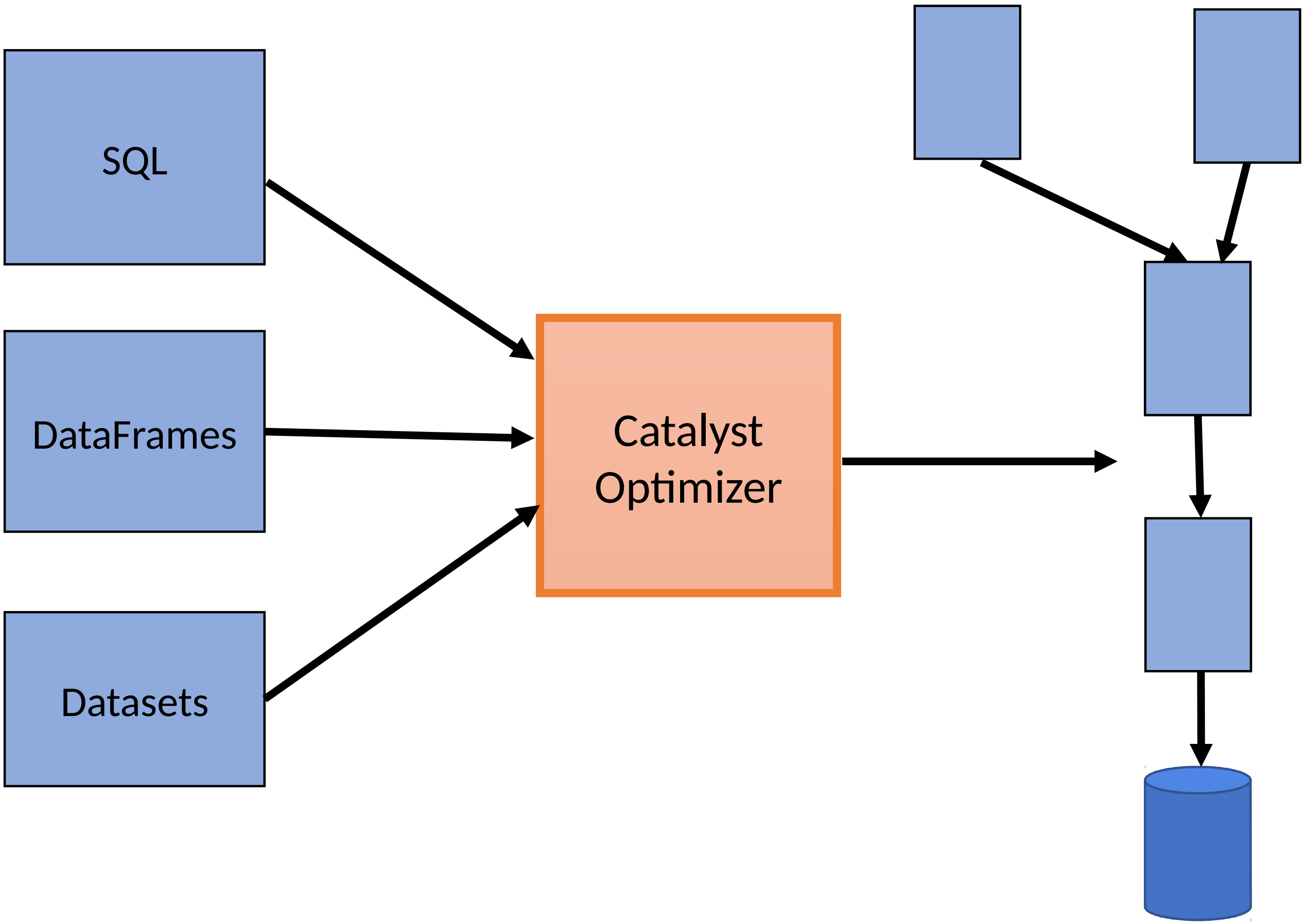
# Dataframe Example

## Programmatically specifying a schema

```
windowsDF.printSchema()
root
 |-- logCount: long (nullable = true)
 |-- facilityCount: array (nullable = true)
 |    |-- element: long (containsNull = true)
 |-- severityCount: array (nullable = true)
 |    |-- element: long (containsNull = true)
 |-- appCount: map (nullable = true)
 |    |-- key: string
 |    |-- value: long (valueContainsNull = true)
```

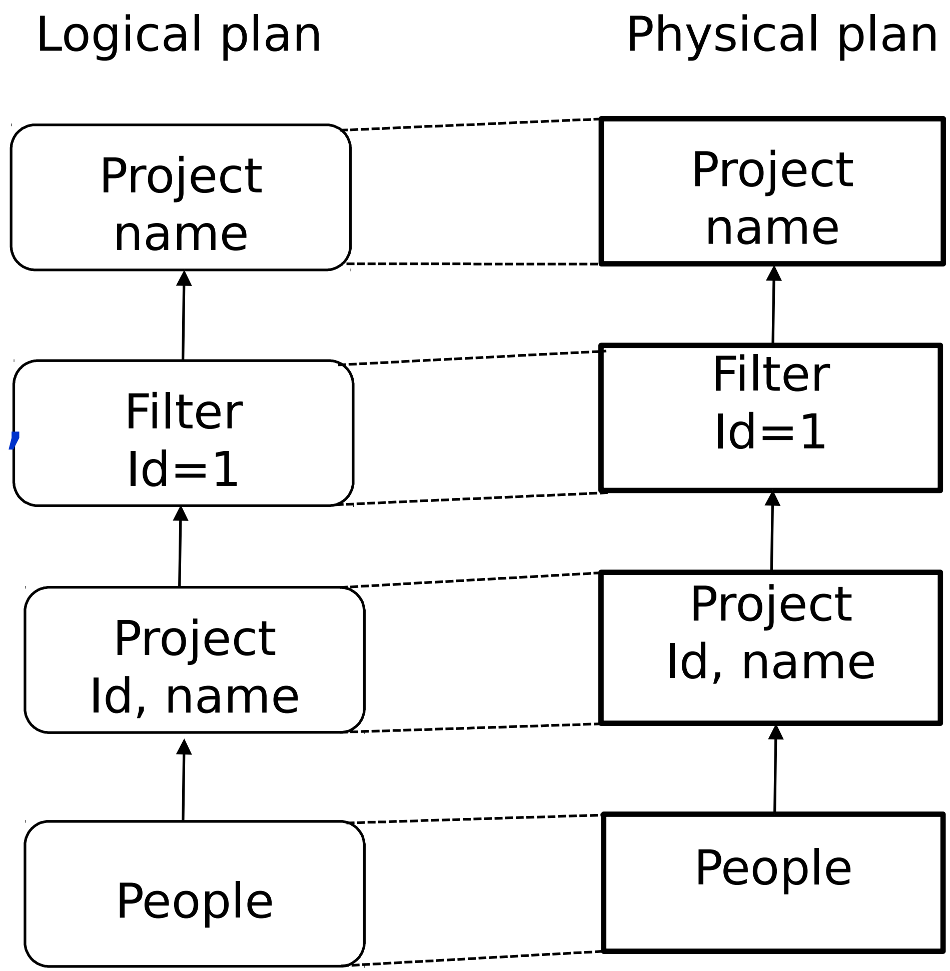
# Dataframe Optimization

**Catalyst optimizer**



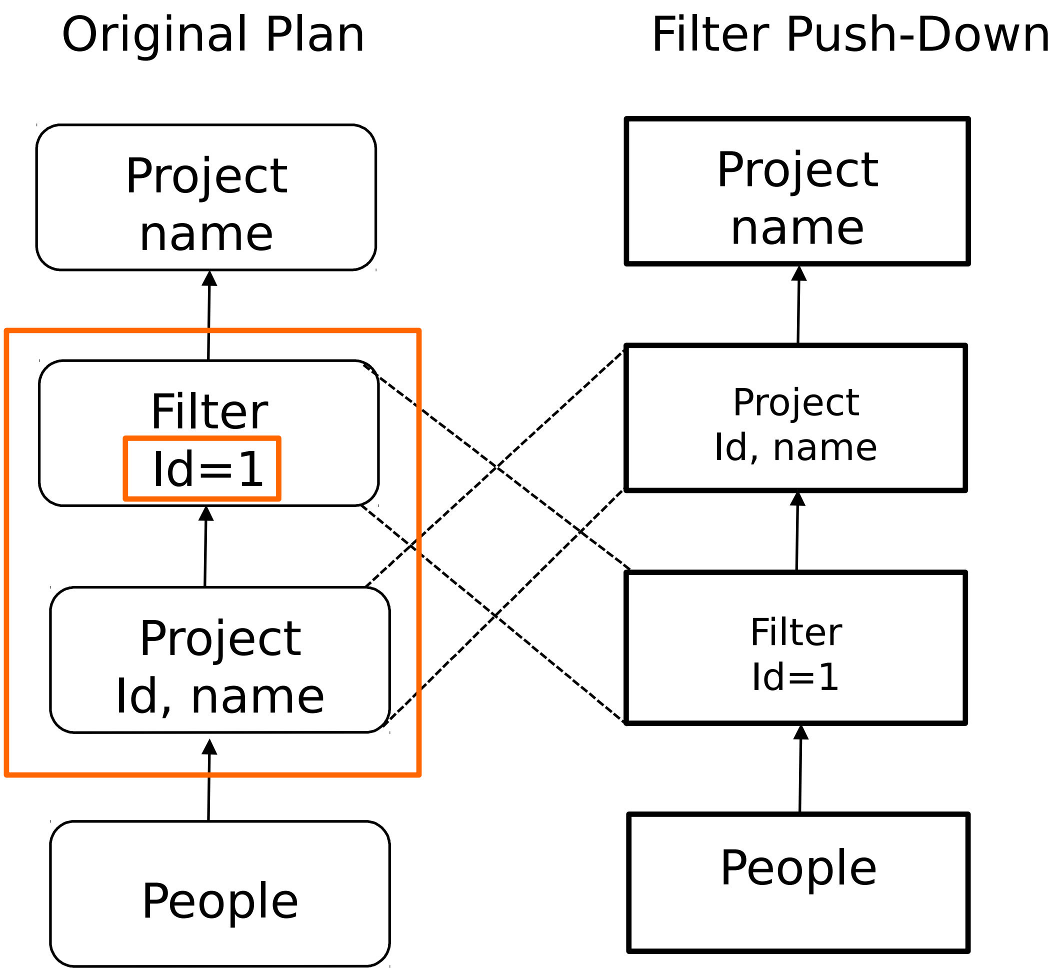
# Spark SQL Optimization

SELECT name
FROM (SELECT id, name
FROM People) p
WHERE p.id = 1



# Spark SQL Optimization

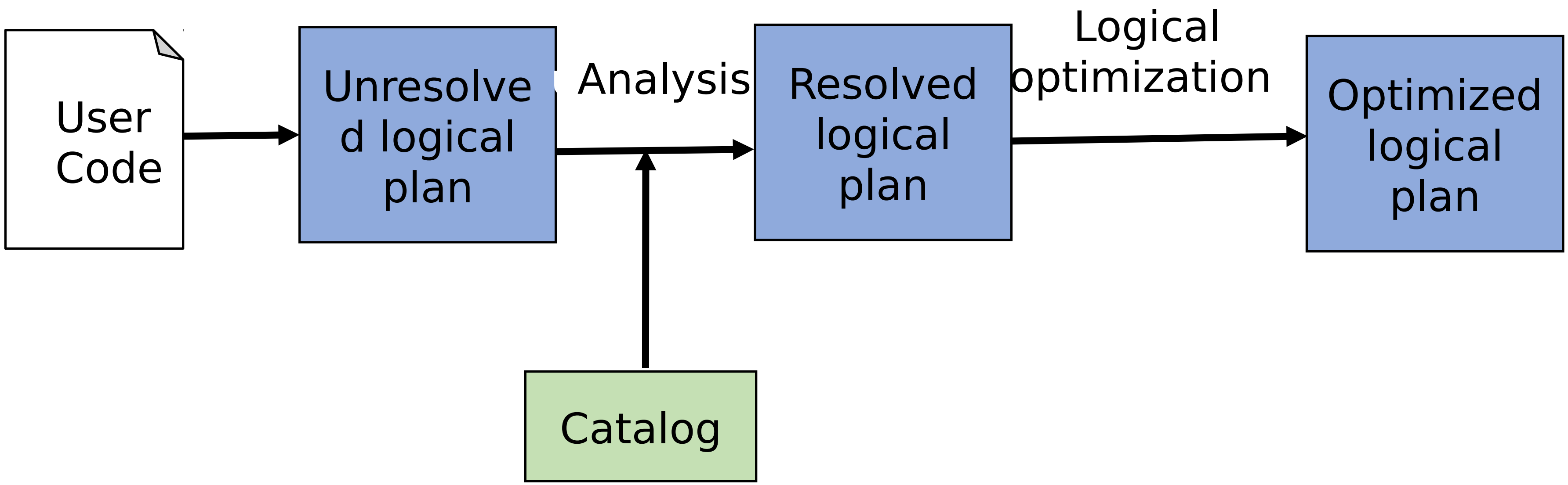
1. Find filters on top projections
2. Check that the filter can be evaluated without the result of the project
3. If so, switch the operators



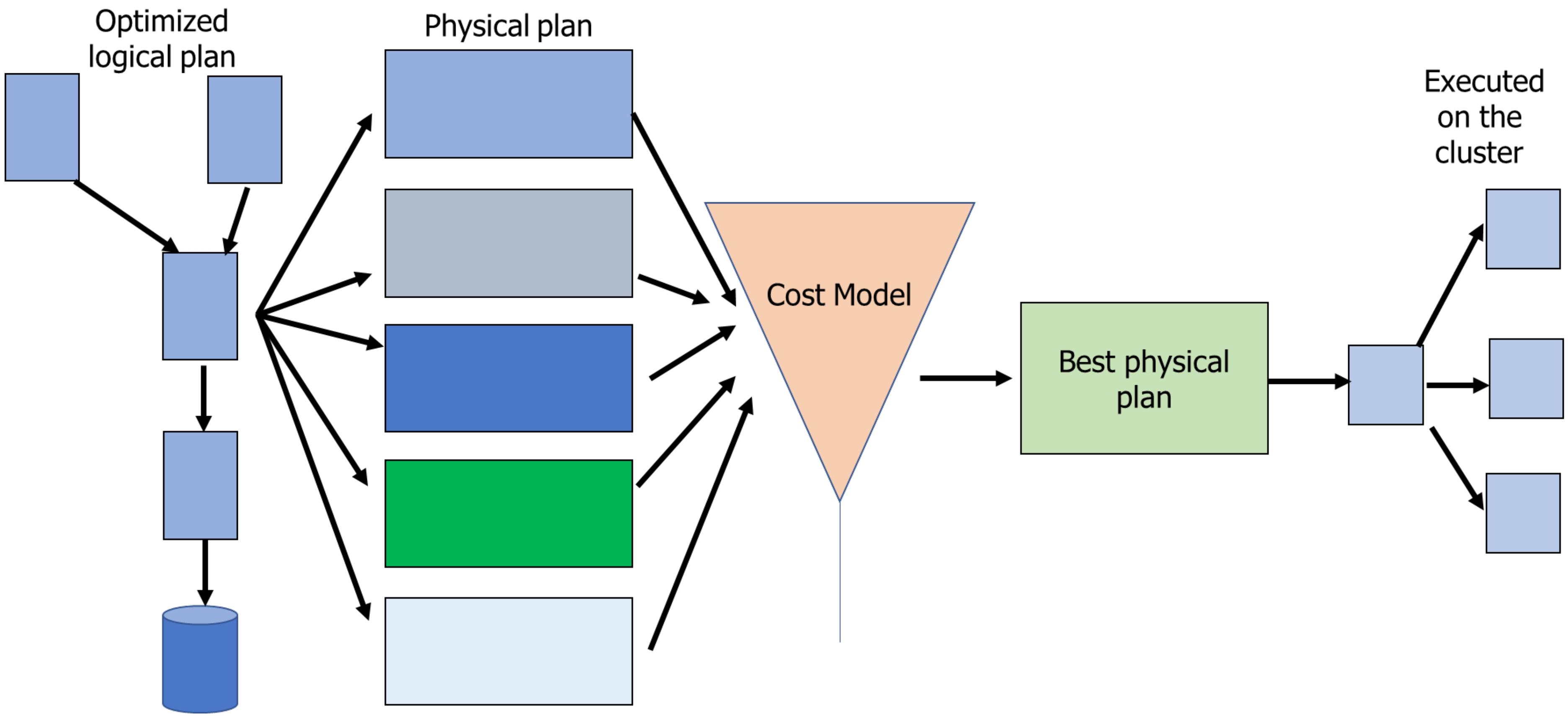
# Dataframe Optimization

- Catalyst optimizer [SIGMOD 2015]
  - Supports rule-based and cost-based optimizations
  - Users can extend optimizer
    - Data source specific rules
      - e.g., push filtering/aggregation into external storage
- Spark performs standard optimization
  - Constant folding
  - Predicate pushdown
  - Projection pruning
  - Null propagation
  - Boolean expression simplification
  - etc.

# Dataframe Optimization



# Physical Planning



# Spark Streaming

- Library on top of Spark, that enables processing of live data streams
- Stream processing API
  - Data divided into mini-batches
- DStream □ Sequence of RDDs
- Sources: HDFS, Kafka, Flume, Kinesis
- Supported languages
  - Scala
  - Java
  - And Python

# SparkMLlib/ ML

- Enables development of machine learning applications
  - Classification and regression (linear models, decision trees)
  - Collaborative filtering (ALS)
  - Clustering (k-means, Gaussian mixture, LDA)
  - Dimensionality reduction (SVD, PCA)
  - etc
- Two packages
  - MLlib: on top of RDDs
  - ML: on top of DataFrames
- Supporting languages
  - Scala
  - Java
  - Python
  - And R

# ML API

- Based on DataFrames
- ML workflows can be specified as pipelines
  - Pipelines can be trained with combination of parameters
- Pipeline elements
  - Transformer
  - Estimator
  - Evaluator

# 8. Conclusion on Spark

- Rich APIs to make data analytics *fast*
- Up to 100x speedups in real applications
- Framework for large-scale parallel processing
- Multiple integrated components

# 9. Spark Practice

- Download and run Spark’s Python shell
- Basic exercise
- Advanced exercise
- Big data project with Spark

# Spark Practice

1. Download and run Spark’s Python shell
2. Basic exercise
3. Advanced exercise

# Download and run Spark's Python shell

- Download Spark binaries from https://spark.apache.org/downloads.html
  - Choose  some « pre-built for Apache Hadoop » package type
  - Or directly from the Apache mirror
- Extract the archive, enter into the extracteddirectory and start Spark's Python shell by running the ./bin/pyspark executable
- Note that the number of workers can be passed with the  -- master argument of pyspark
  - E.g. This command initiates a Spark cluster with 2 workers : ./bin/pyspark --master local [2]

# Basic Exercise

- Word count
- Evaluating the benefit of parallelism
- Filtering the result

# Advanced Exercise

- First goal: get user-keyword pairs
- Count the occurrence of each pair
- Map by keywords
- Final goal: keep only the winner
- Raising the scale
- Filter by particular keywords

# Big Data Project with Spark

- Objectives
- Big datasets
- Project definition
- Tools
- Design and implementation
- Experiments

# Objectives

- Learn, learn, learn
- Why?
  - Good job opportunies
  - Learning big data is fun
- How?
  - By doing a real (yet small-scale) big data project
    - Design, implementation, experimentation
  - By making it interesting and working as hard as you can (want)
    - But be realistic with the goal

# Big Datasets

- Selection from the following list
  - African Economic 2018
  - Gender Poverty and Environmental Indicators
  - Gross Disbursement of Bilateral Official Flows
  - Health Nutrition and Population Statistics
  - Human Development Report, 2018 Statistical
  - Communication information in Benin
  - illegal-waste-dump-coordinates
  - List of food centre in edo state
  - List of tourist attractions and their status
  - Orphanage homes in edo state
  - School projects execution in 2013

# Project Definition

- Understand the dataset
  - Identify opportunities for insights
- Business objectives
  - What can you learn that will make a difference
- Technical objectives
  - In support of business objectives
- Data requirements
  - Characterize the data (at large scale)
- Identify use case queries
  - To be implemented in Spark

# Tools

- Spark
- MLlib,
- HDFS
- Neo4j
- MongoDB

# Design and Implementation

- Python programming of business functions

Carlyna

# Experimentation