DATA ANALYSIS WITH APACHE SPARK & PYTHON

Data science summer school 2017

Filip Wójcik, senior data scientist
filip.wojcik@outlook.com

AGENDA

1. Apache Spark intro
    1. What is apache Spark?
    2. Key components of Apache Spark
    3. Apache Spark architecture
    4. Distributed computation model
    5. Map-reduce model vs split-apply-combine
2. Computation graphs
    1. RDDs
    2. Transformations
    3. Actions
3. Data structures
    1. RDD - older API
    2. Data Frames
    3. DataSets

APACHE SPARK INTRO

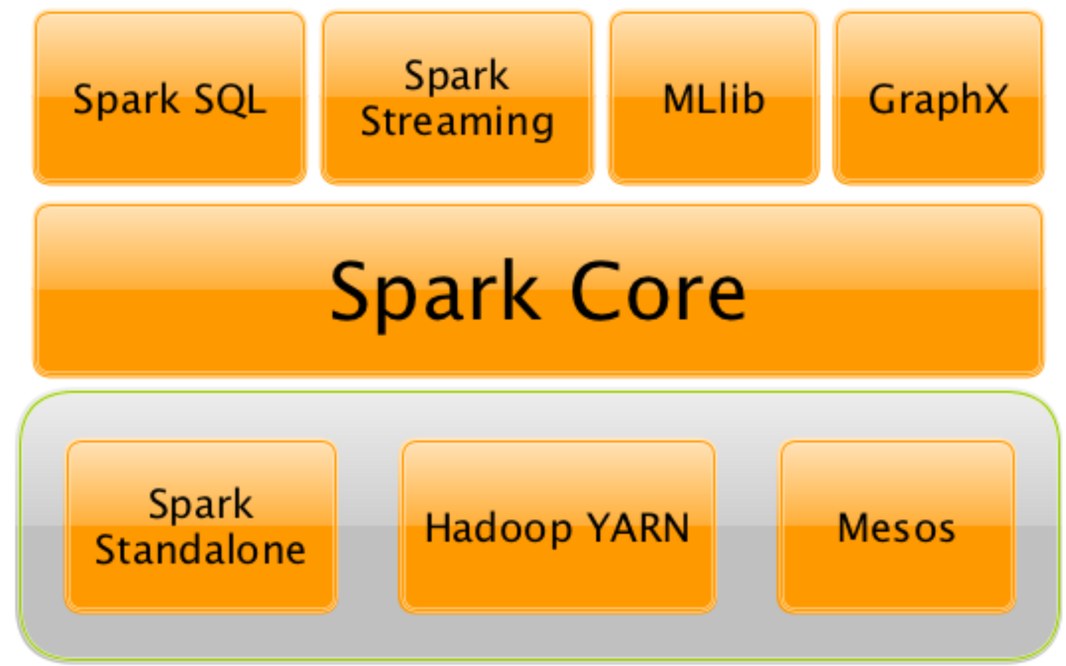
basic concepts and techniques

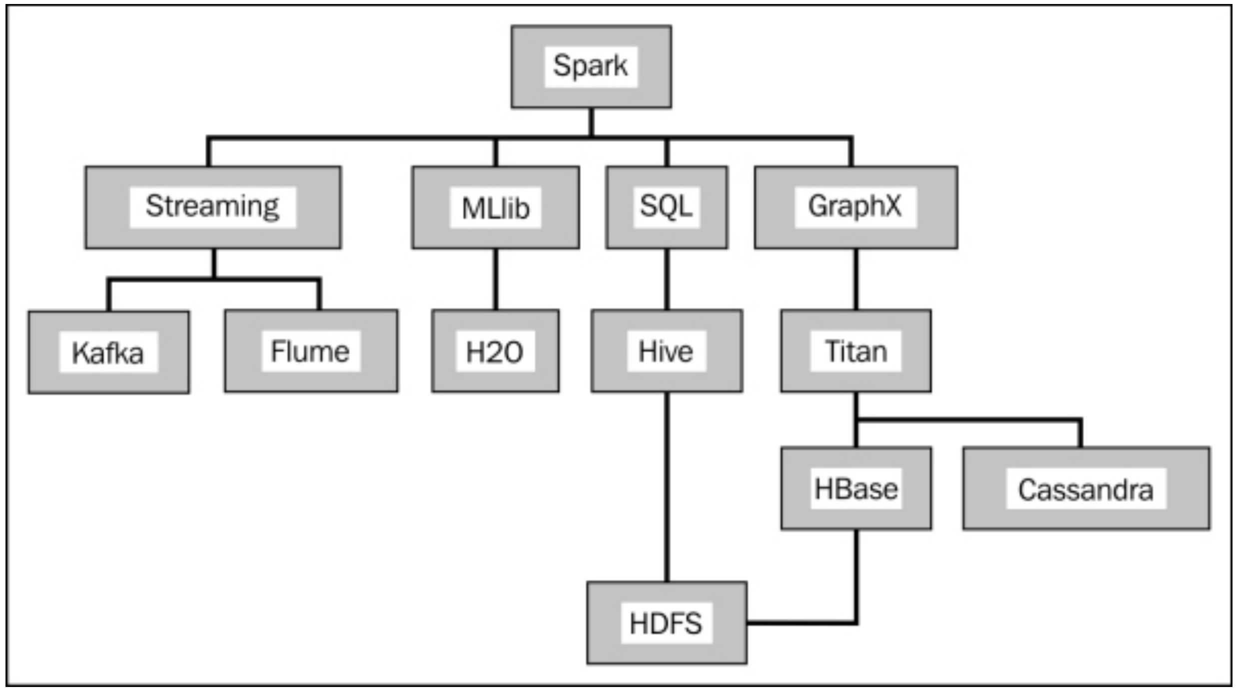
WHAT IS APACHE SPARK?

- Distributed processing system
  - Initially designed on Berkeley University
  - Later developed by Apache Software Foundation
  - Main contributor - The Databricks
- Open-source
- Designed to work with big volumes of data
- Works on top of JVM Hadoop environment

KEY COMPONENTS OF APACHE SPARK

- Spark core sits on top of Hadoop resource managers
- Spark API contains several specialized libraries - each of them is designed to work with different type of tasks
- Spark SQL - emulates query language on a large scale
- Spark Streaming - live-processing of incoming messages
- MLlib - machine learning library for supervised/ unsupervised learning
- GraphX - graph databse

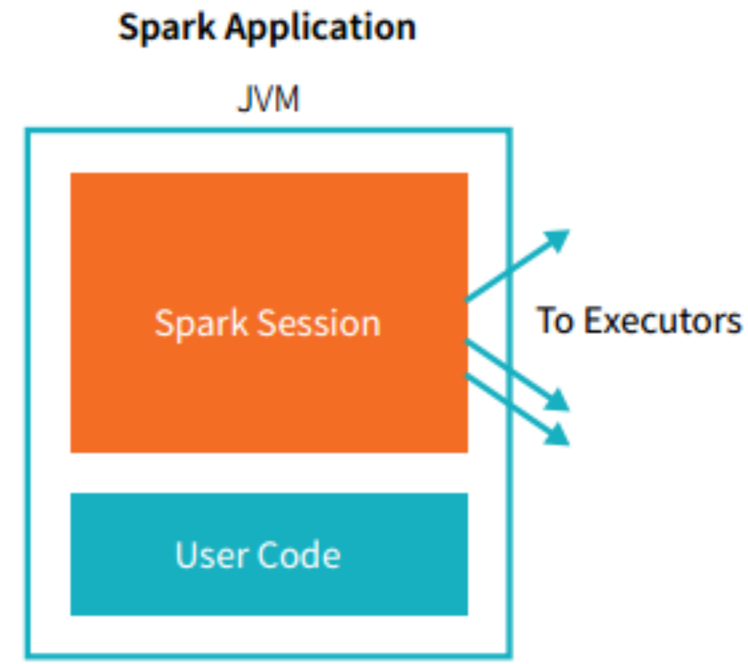


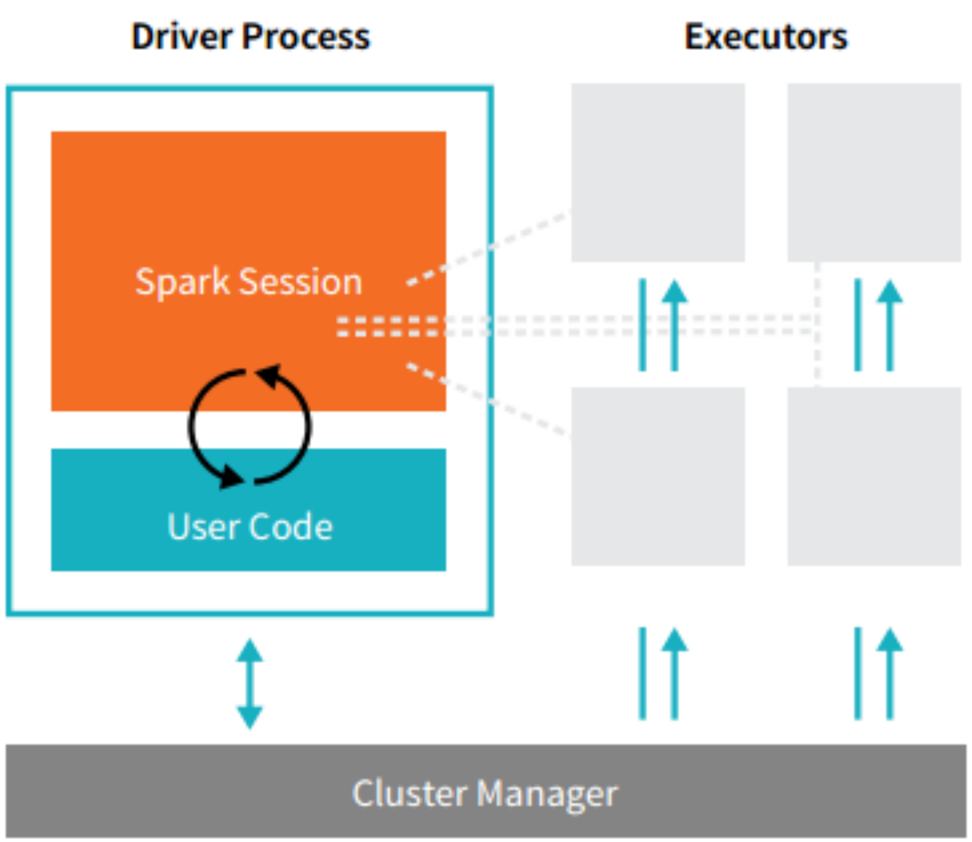


Source: https://databricks.com/product/getting-started-guide

APACHE SPARK ARCHITECTURE

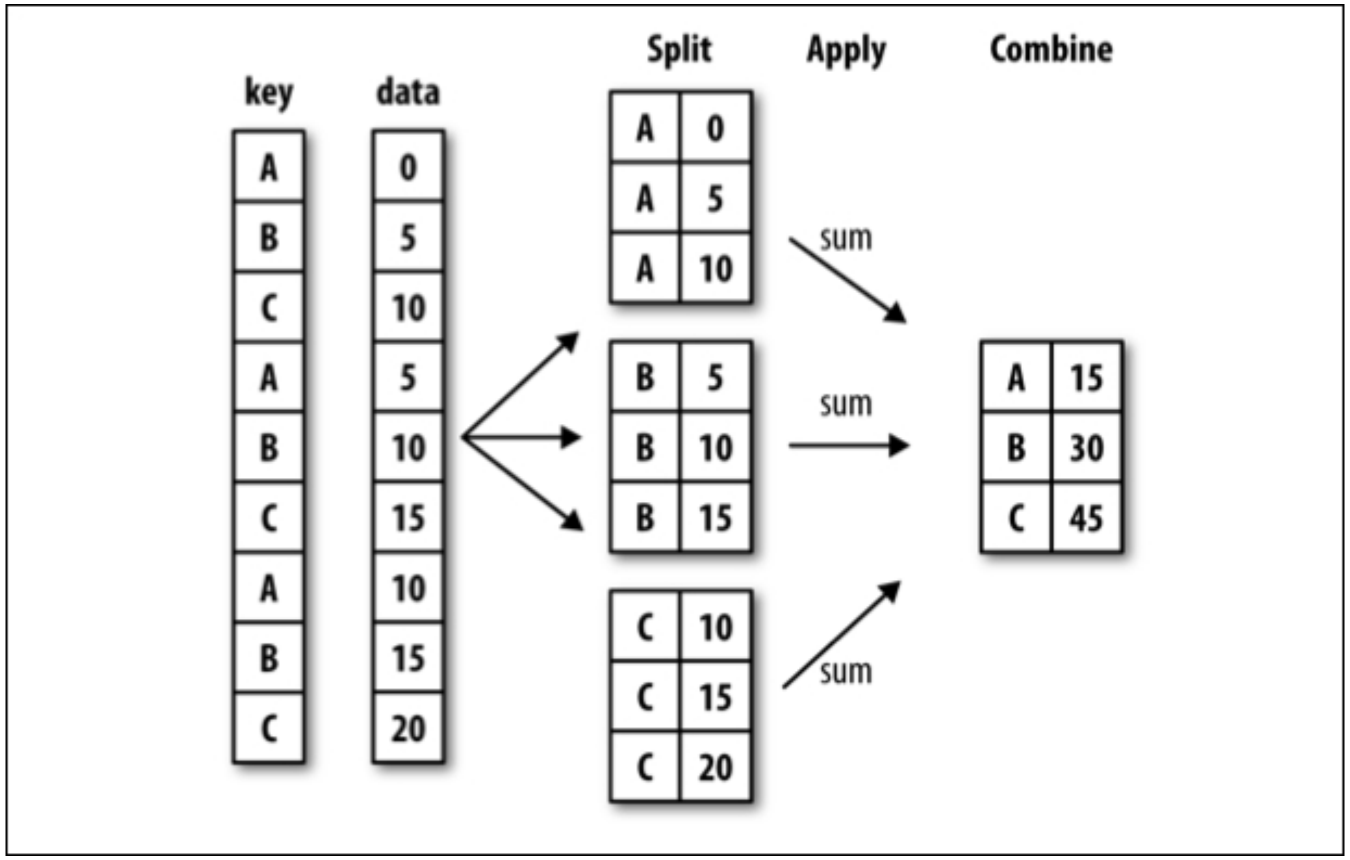
- Code is developed on user's machine and scheduled from there - it is called **DRIVER PROCESS**
- So called **EXECUTOR** are responsible for doing actual work on *some machines*
- **CLUSTER MANAGER's** responsibility is to delegate resources and balance server load
- There can be many executors - depending on the cluster configuration and resources
- It is possible to invoke Spark locally, without delageation to the cluster - but rather for learning & debugging purposes





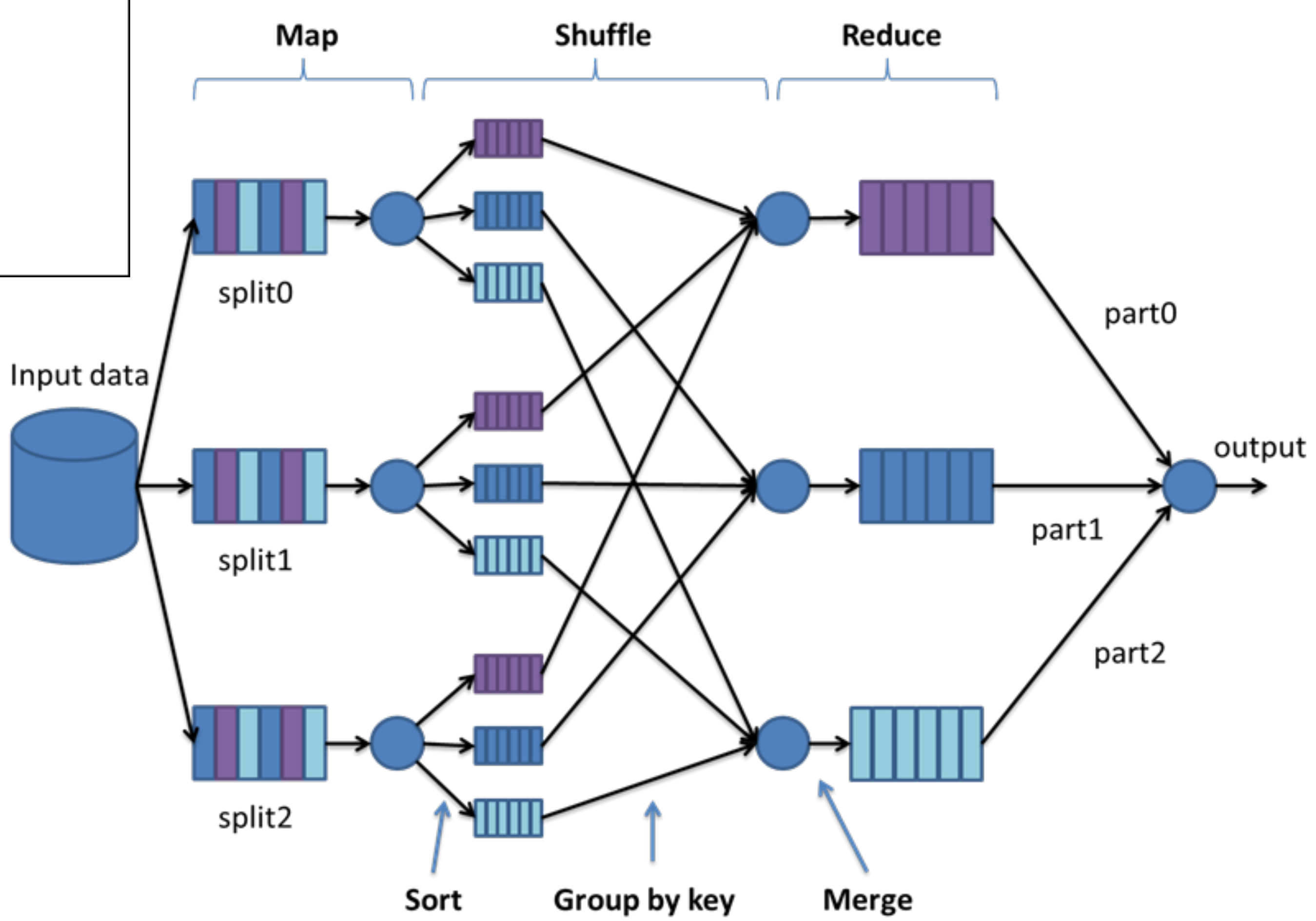
Source: Databricks, Apache Spark: Definitive Guide

DISTRIBUTED CALCULATIONS MODEL



Source: William McKinney, Python for Data Analysis, 2nd Edition

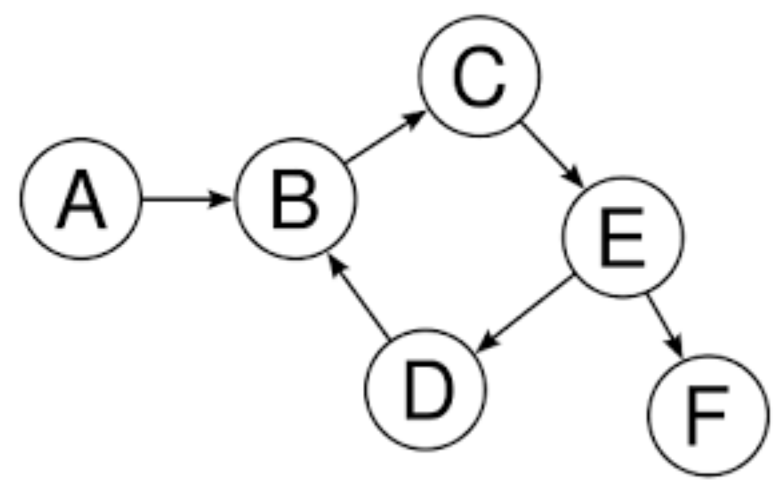
- Parallelized operations
- Split-apply-combine distributed across servers
- PARTITIONING THE DATA and shuffling to allocate on servers
- Each **partition goes to different physical machine**



Source: Databricks, Apache Spark: Definitive Guide

COMPUTATION GRAPHS

basic operations and transformations

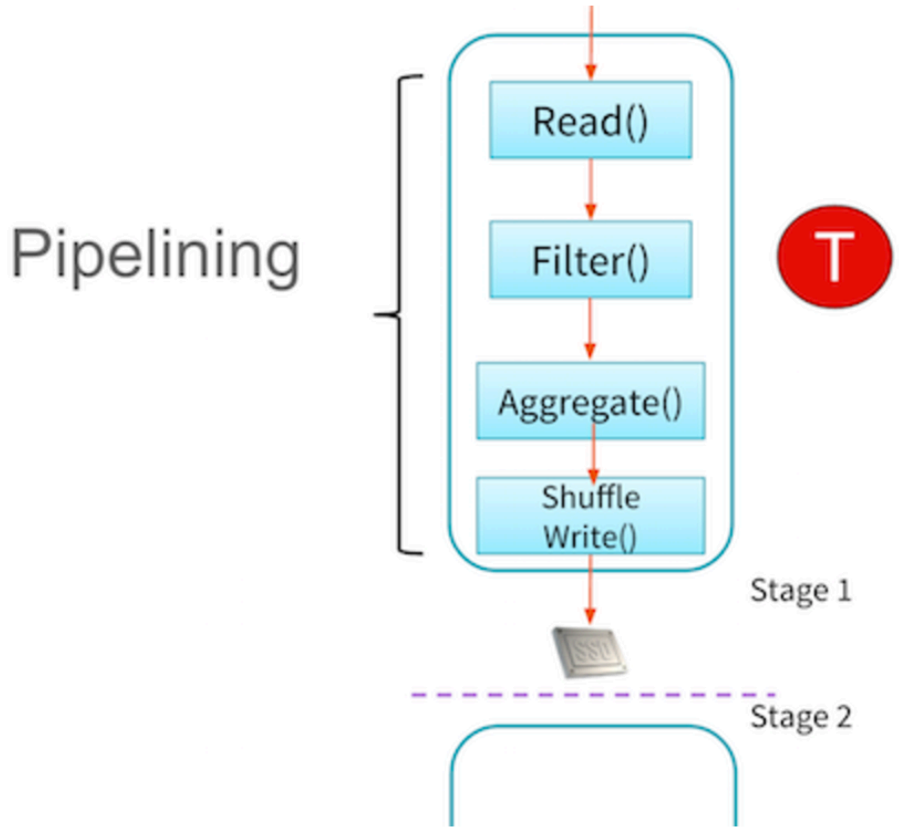


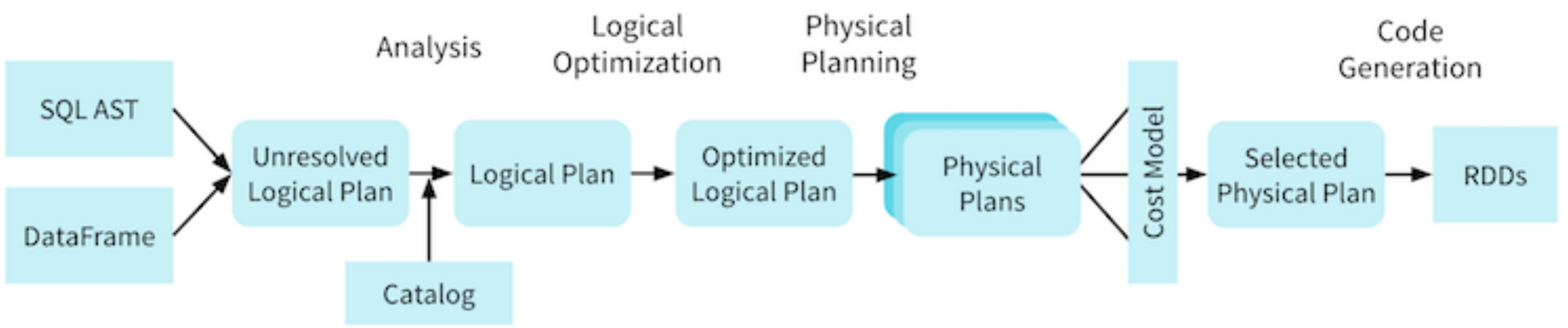
RDDS

- Resilient distributed datasets - low-level, older API, basic data structure in Spark
- Key features:
  - immutable
  - parallelized and distributed across nodes/servers
  - partitioned according to some key (natural or artificial)
  - fault-tolerant (archived on worker nodes with fallback procedures)
  - lazy evaluated
- Transformed step-by-step by deterministinc operations

TRANSFORMATIONS

- No data physical modifications
- User defines **chain of transformations - subsequent operations to reshape the data**
- Spark engine keeps track of those changes
- OPTIMIZER finds the best way to allocate data
- Real data operations are planned and the whole graph is executed

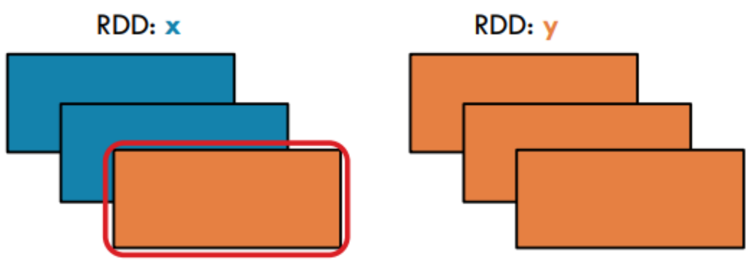




Source: https://databricks.com/product/getting-started-guide

BASIC TRANSFORMATIONS

MAP



`map(f, preservesPartitioning=False)`

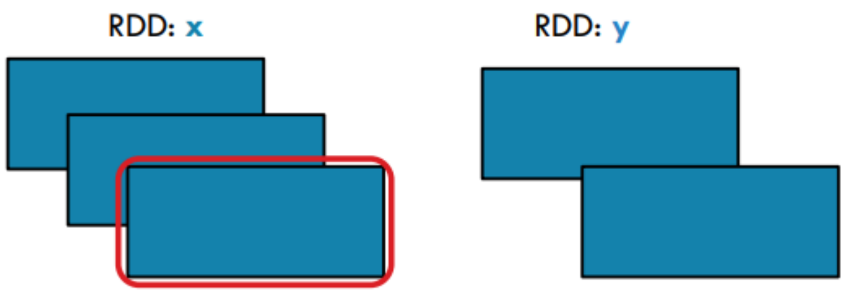
Return a new RDD by applying a function to each element of this RDD

```
x = sc.parallelize(["b", "a", "c"])
y = x.map(lambda z: (z, 1))
print(x.collect())
print(y.collect())
```

```
x: ['b', 'a', 'c']

y: [('b', 1), ('a', 1), ('c', 1)]
```

FILTER



`filter(f)`

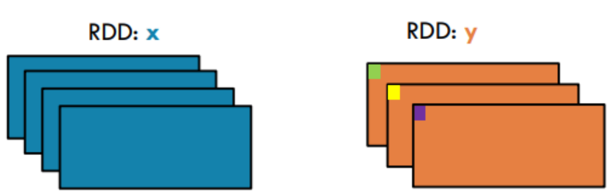
Return a new RDD containing only the elements that satisfy a predicate

```
x = sc.parallelize([1,2,3])
y = x.filter(lambda x: x%2 == 1) #keep odd values
print(x.collect())
print(y.collect())
```

```
x: [1, 2, 3]

y: [1, 3]
```

GROUPBY



`groupBy(f, numPartitions=None)`

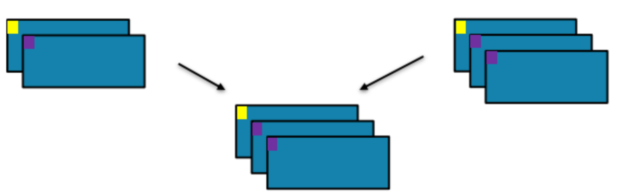
Group the data in the original RDD. Create pairs where the key is the output of a user function, and the value is all items for which the function yields this key.

```
x = sc.parallelize(['John', 'Fred', 'Anna', 'James'])
y = x.groupBy(lambda w: w[0])
print [(k, list(v)) for (k, v) in y.collect()]
```

```
x: ['John', 'Fred', 'Anna', 'James']

y: [('A',['Anna']),('J',['John','James']),('F',['Fred'])]
```

JOIN



Return a new RDD containing all pairs of elements having the same key in the original RDDs

`union(otherRDD, numPartitions=None)`

```
x = sc.parallelize([("a", 1), ("b", 2)])
y = sc.parallelize([("a", 3), ("a", 4), ("b", 5)])
z = x.join(y)
print(z.collect())
```

```
x: [("a", 1), ("b", 2)]

y: [("a", 3), ("a", 4), ("b", 5)]

z: [('a', (1, 3)), ('a', (1, 4)), ('b', (2, 5))]
```

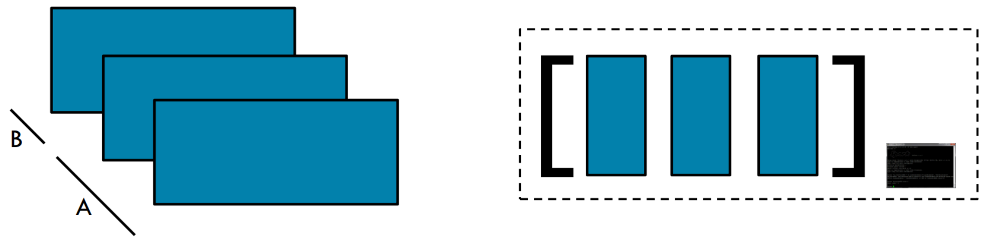
Source: http://training.databricks.com/visualapi.pdf

ACTIONS

- Cause data materialization
- All calculations are triggered and executed
- Data is being returned to the DRIVER MACHINE
- Data is calculated in memory and collected back on driver
- Potentially a bottleneck in whole processing - the most expensive operations

BASIC ACTIONS

COLLECT



collect()

Return all items in the RDD to the driver in a single list

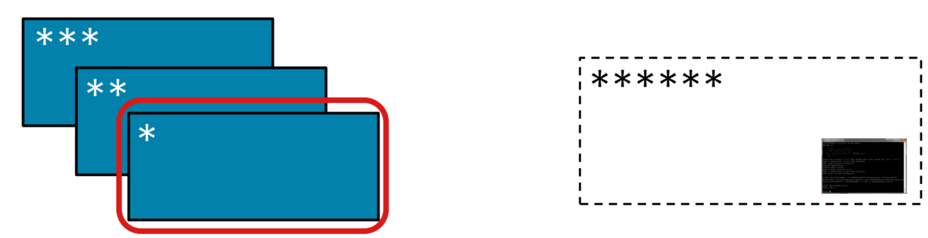
```
x = sc.parallelize([1,2,3], 2)
y = x.collect()

print(x.glom().collect())
print(y)
```

```
x: [[1], [2, 3]]

y: [1, 2, 3]
```

REDUCE



reduce(*f*)

Aggregate all the elements of the RDD by applying a user function pairwise to elements and partial results, and returns a result to the driver

```
x = sc.parallelize([1,2,3,4])
y = x.reduce(lambda a,b: a+b)

print(x.collect())
print(y)
```

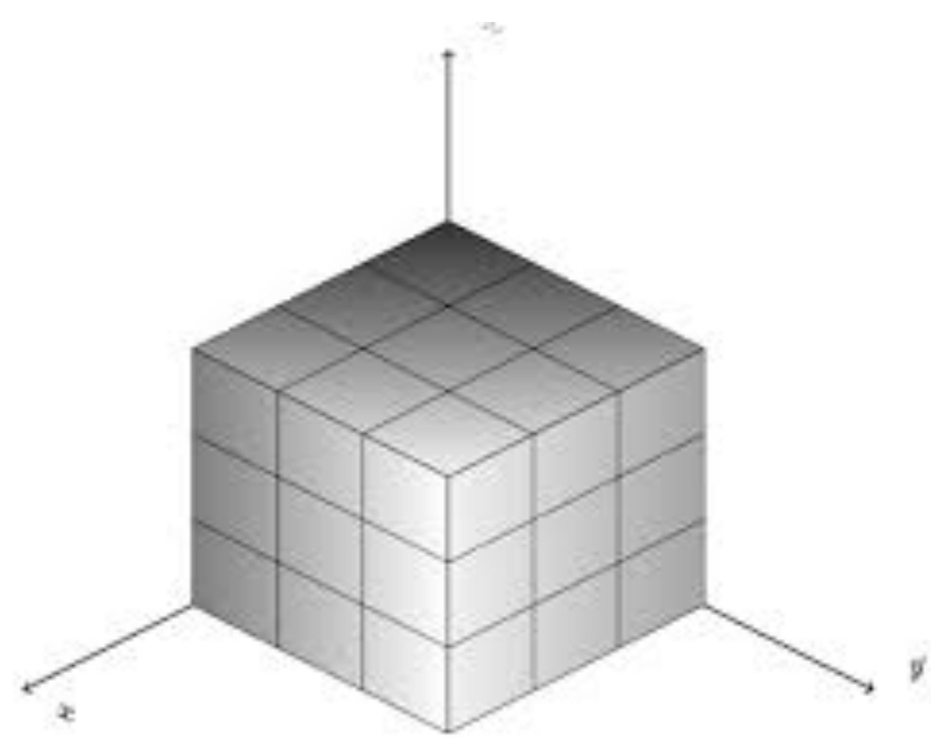
```
x: [1, 2, 3, 4]

y: 10
```

Source: http://training.databricks.com/visualapi.pdf

DATA STRUCTURES

three main APIs in Spark



DATA STRUCTURES

- Spark has 3 main data APIs
- RDDs are historically the first and the most low-level of all
- Slowly, other approaches were becoming more popular, replacing RDD
- What is important to remember is the fact, that all **high level APIs are based on RDDs, which are the core!**

DATA STRUCTURES

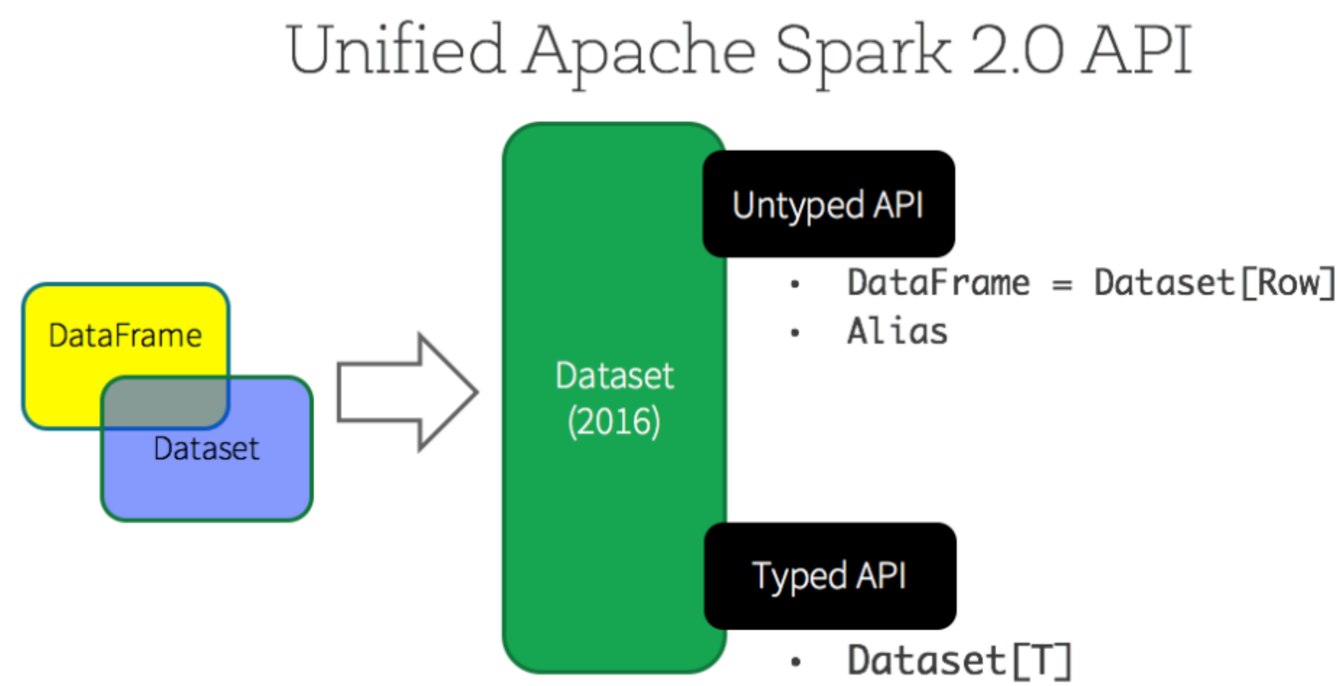
Feature\Structure	RDD	Dataframe	Dataset	SQL
type	untyped	untyped	typed	typed
operations	1. granular 2. basic 3. full control	1. SQL-like operations 2. per column manipulations	1. SQL-like operations 2. per column manipulations	mimics classic sql
optimization	low	moderate	high	ultra-high :)
technology	all	all	Scala, Java	all

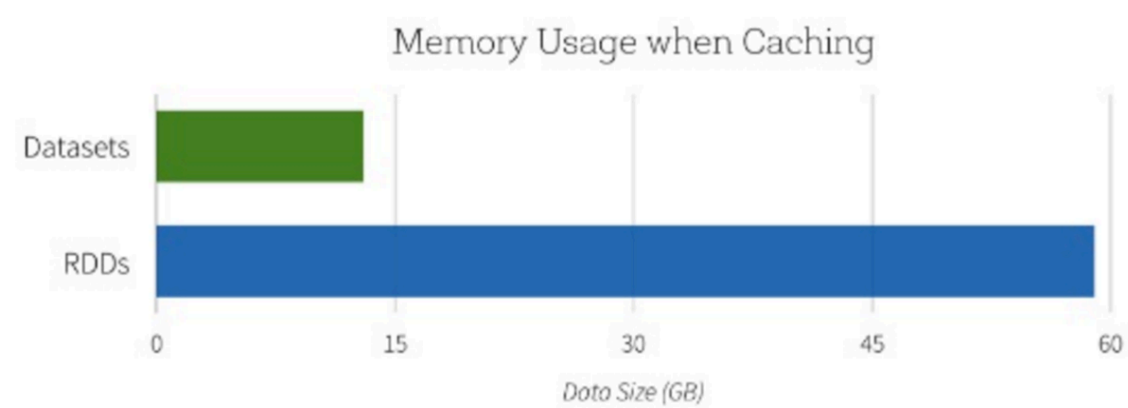
◉ ◉ ◉ ○

DATA STRUCTURES

- „Typed” APIs are available only in compiled languages - Java + Scala
- Main befefit - type safety and syntax checking
- Better optimization due to well-known types in compilation time
- RDDs are not deprecated - they are just used for other purposes!

	SQL	DataFrames	Datasets
Syntax Errors	Runtime	Compile Time	Compile Time
Analysis Errors	Runtime	Runtime	Compile Time





Source: https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html

DATA STRUCTURES

DataFrames - building „by hand”

```
elements = [
    ['Name1', 'Surname1',30],
    ['Name2', 'Surname2',35],
    ['Name3', 'Surname3',21]

  ]

elements_rdd = sc.parallelize(elements)
elements_df = sqlContext
   .createDataFrame(elements_rdd, ['name', 'surname', ,age'])
elements_df.show()
```

```
+-----+--------+---+
| name| surname|age|
+-----+--------+---+
|Name1|Surname1| 30|
|Name2|Surname2| 35|
|Name3|Surname3| 21|
+-----+--------+---+
```

DataFrames - building „from file”

```
file_path = "derinet-products-ch.csv"
separator = ";"
data =
sqlContext.read.format('com.databricks.spark.csv').options(
header='true', inferschema='true',
sep=separator).load(file_path)

data.printSchema()
```

_c0	carat	cut	color	clarity
1	0.23	Ideal	E	SI2
2	0.21	Premium	E	SI1
3	0.23	Good	E	VS1
4	0.29	Premium	I	VS2
5	0.31	Good	J	SI2
6	0.24	Very Good	J	VVS2
7	0.24	Very Good	I	VVS1
8	0.26	Very Good	H	SI1
9	0.22	Fair	E	VS2

Showing the first 1000 rows.