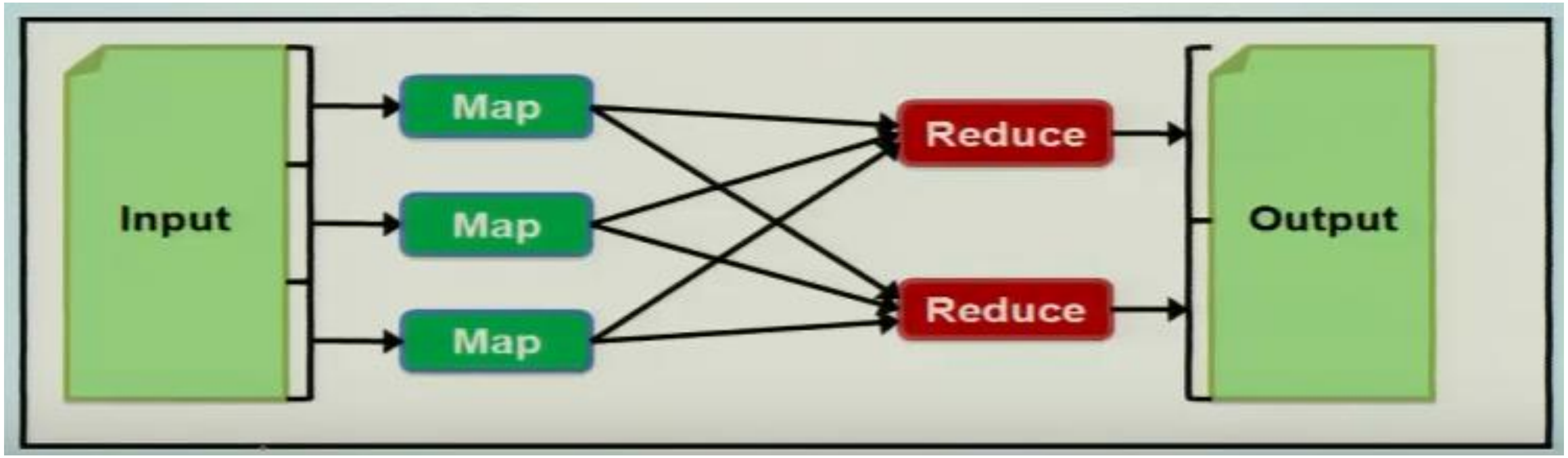
Introduction to SPARK

Motivation

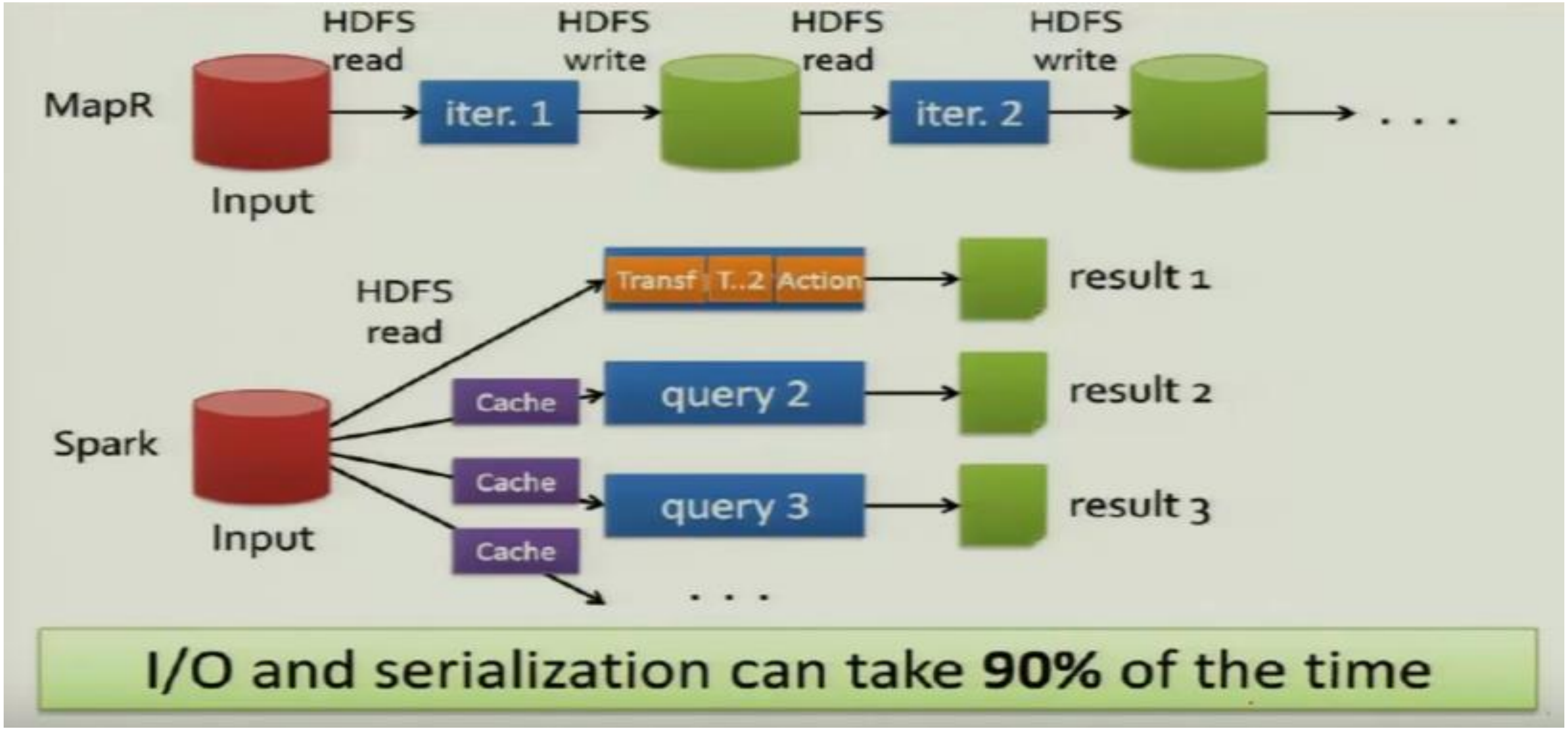
- MapReduce and its variants have been highly successful in implementing large-scale data-intensive application on commodity clusters.



Motivation

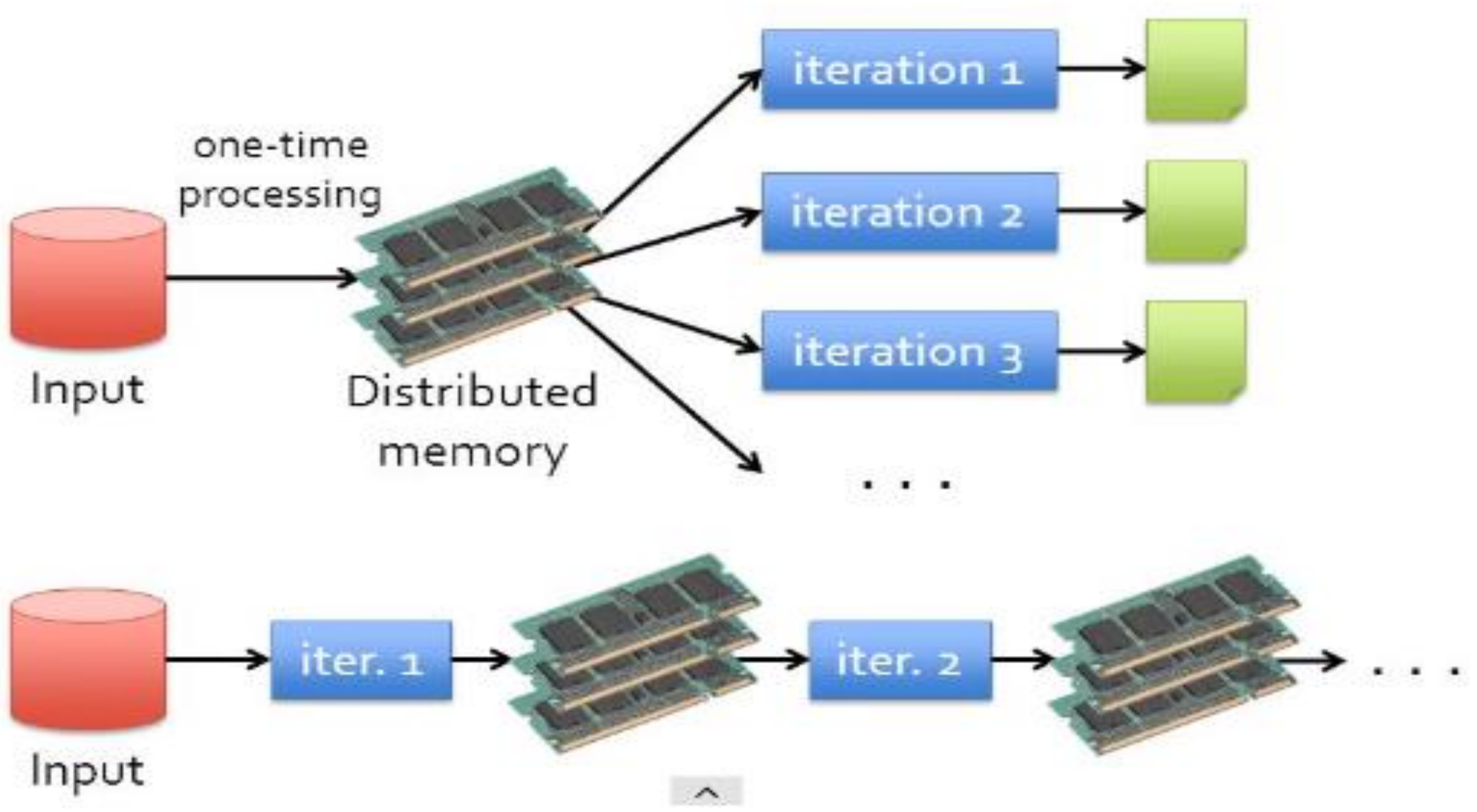
- Most current cluster programming models are based on acyclic data flow from stable storage to stable storage
- Acyclic data flow is inefficient for applications that are repeatedly reuse working set of data:
  - Iterative algorithm (Machine learning algorithm, Graphs)
  - Interactive datamining tools (R, Excel, Python)
- With current frameworks, apps reloads data from stable storage on each query

Why SPARK?



I/O and serialization can take **90%** of the time

Why Spark?


one-time processing
Input
Distributed memory
iteration 1
iteration 2
iteration 3
. . .
Input
iter. 1
iter. 2
. . .

Spark in Industry

- Apache Spark because of it’s amazing features like **in-memory processing**, **polyglot** and **fast processing** are being used by many companies all around the globe for various purposes in various industries:

Spark in Industry

Yahoo

- uses Apache Spark for its Machine Learning capabilities to personalize its news, web pages and also for target advertising.
- They use Spark with python to find out
  - what kind of news users are interested to read and
  - categorizing the news stories to find out what kind of users would be interested in reading each category of news.

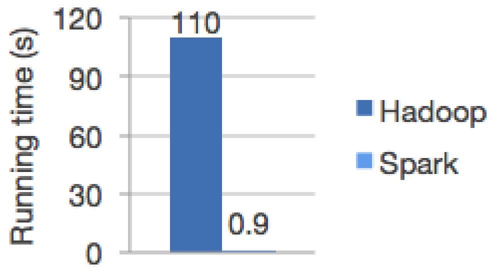
Spark in Industry

TripAdvisor

- Uses apache spark to provide advice to millions of travelers by comparing hundreds of websites to find the best hotel prices for its customers.
- The time taken to read and process the reviews of the hotels in a readable format is done with the help of Apache Spark.
- One of the world’s largest e-commerce platform **Alibaba** runs some of the largest Apache Spark jobs in the world in order to analyze hundreds of petabytes of data on its e-commerce platform.

Apache SPARK

- Distributed computing framework provide an efficient way of executing operations
- Supports for application that reuse a working set of data across multiple parallel operations.
- Need to support such applications while retaining the scalability and fault tolerance of MapReduce
- To achieve this, Spark introduces an abstraction called resilient distributed datasets (RDDs).
- Spark can outperform Hadoop by 10x in iterative machine learning jobs and can be used to interactively query a 39 GB dataset with sub-second response time



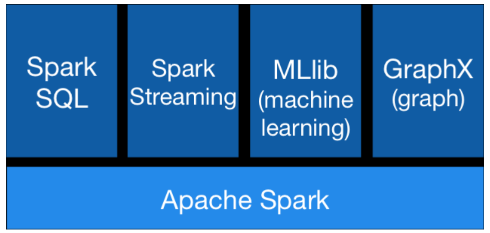
Speed

Logistic regression in Hadoop and Spark

Runs Everywhere

runs on Hadoop, Apache Mesos, Kubernetes, standalone, or in the cloud.
It can access diverse data sources.

Generality

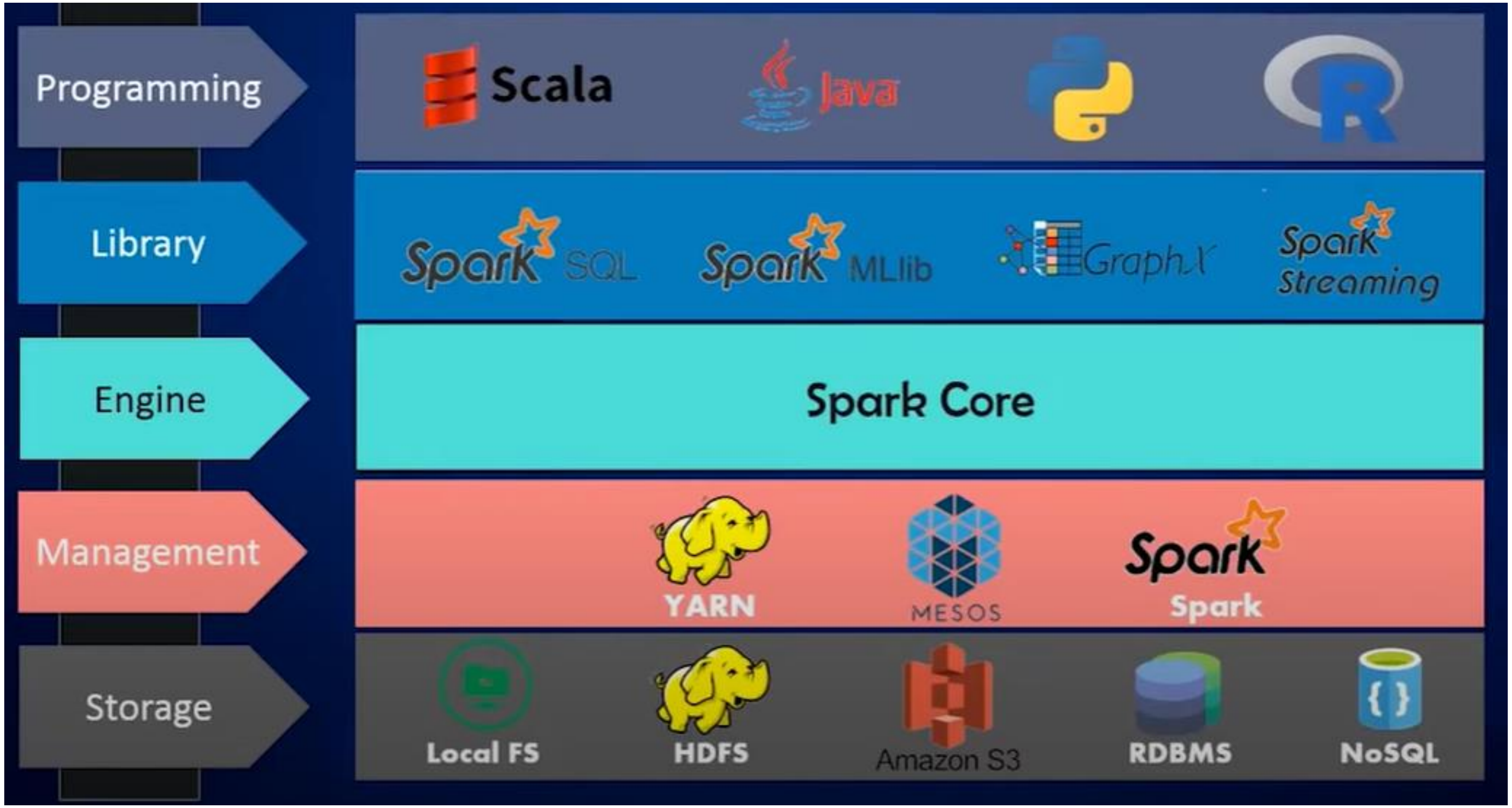


Combine SQL, streaming, and complex analytics.

Ease of Use

Write applications quickly in Java, Scala, Python, R, and SQL.

Spark Eco-system



Hadoop MapReduce VS Apache Spark

Hadoop MapReduce	Apache Spark
Fast	100x faster than MapReduce
Batch Processing	Real-time processing
Stores Data on Disk	Stores data in Memory
Written in Java	Written in Scala, Python, Java, R

Scala Programing Language

- Ideal language for big data anlaytics and distributed processing
- Scala is an object-oriented programming language.
  - Everything in Scala is an object and any operations you perform is a method call.
- Scala is a functional language
  - It has implemented major functional programming concepts
  - every computation is treated as a mathematical function which avoids states and mutable data

https://www.analyticsvidhya.com/blog/2017/01/scala/

Scala Programing Language

- Scala is a compiler based language
- Scala execution very fast if you compare it with Python (which is an interpreted language)
- The compiler in Scala works in similar fashion as Java compiler
  - It gets the source code and generates Java byte-code that can be executed independently on any standard JVM

https://www.tutorialspoint.com/scala/scala_functions.htm

Scala Programing Language

- Companies using Scala
  - It is used by many companies to develop the commercial software.
  - These are the following notable big companies which are using Scala as a programming alternative.

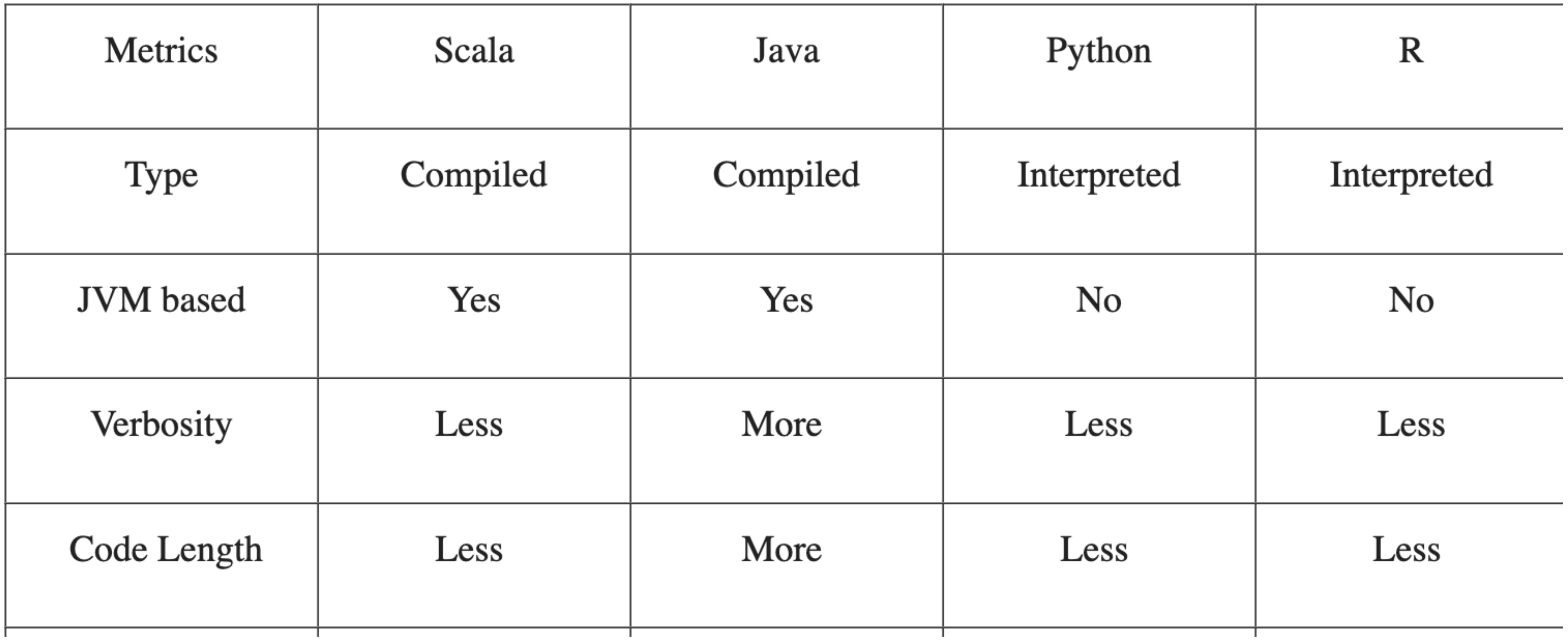
https://www.scala-lang.org/old/node/1658

- LinkedIn
- Twitter
- Foursquare
- Netflix
- Tumblr
- The Guardian

- Precog
- Sony
- AirBnB
- Klout
- Apple

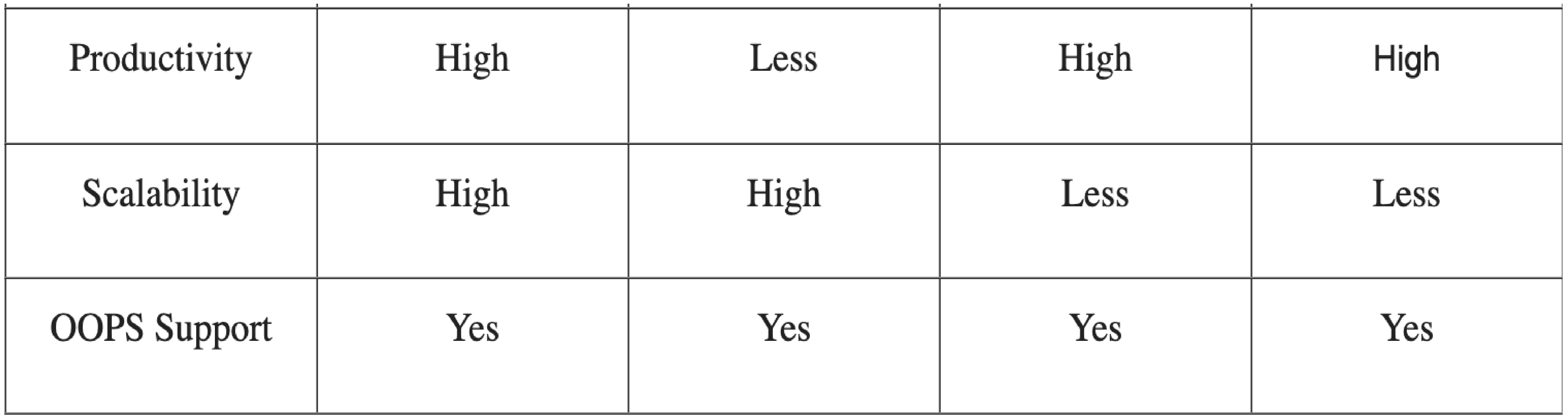
https://www.tutorialspoint.com/scala/scala_functions.htm

Comparing Scala, Java, Python and R APIs in Apache Spark



Metrics	Scala	Java	Python	R
Type	Compiled	Compiled	Interpreted	Interpreted
JVM based	Yes	Yes	No	No
Verbosity	Less	More	Less	Less
Code Length	Less	More	Less	Less

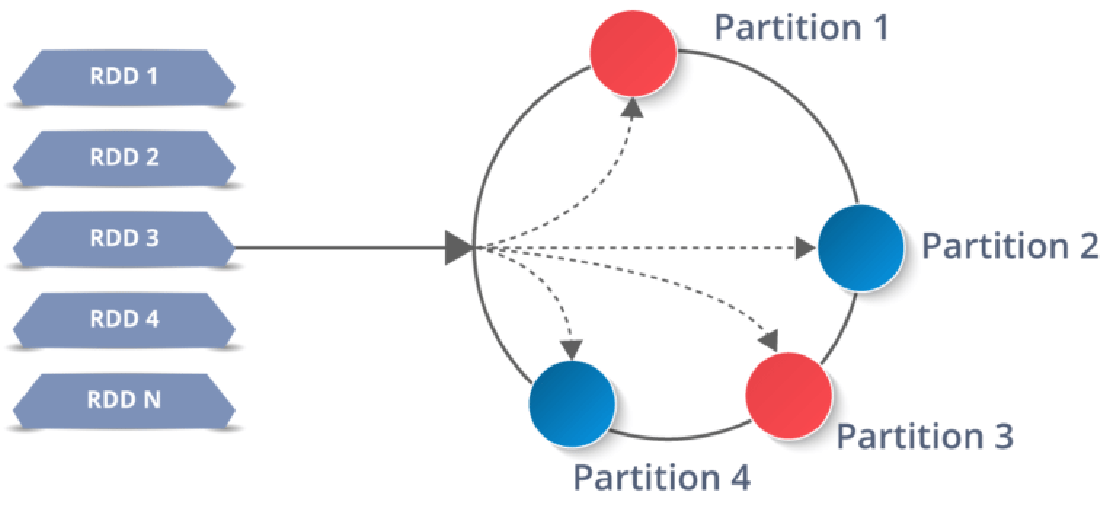
Comparing Scala, Java, Python and R APIs in Apache Spark



Productivity	High	Less	High	High
Scalability	High	High	Less	Less
OOPS Support	Yes	Yes	Yes	Yes

Apache Spark – Resilient Distributed Dataset (RDD)

- RDD represents collection of items distributed across many computer nodes that can be manipulated in parallel
- A distributed memory abstraction that helps a programmer to perform in-memory computations on large clusters that too in a fault-tolerant manner.



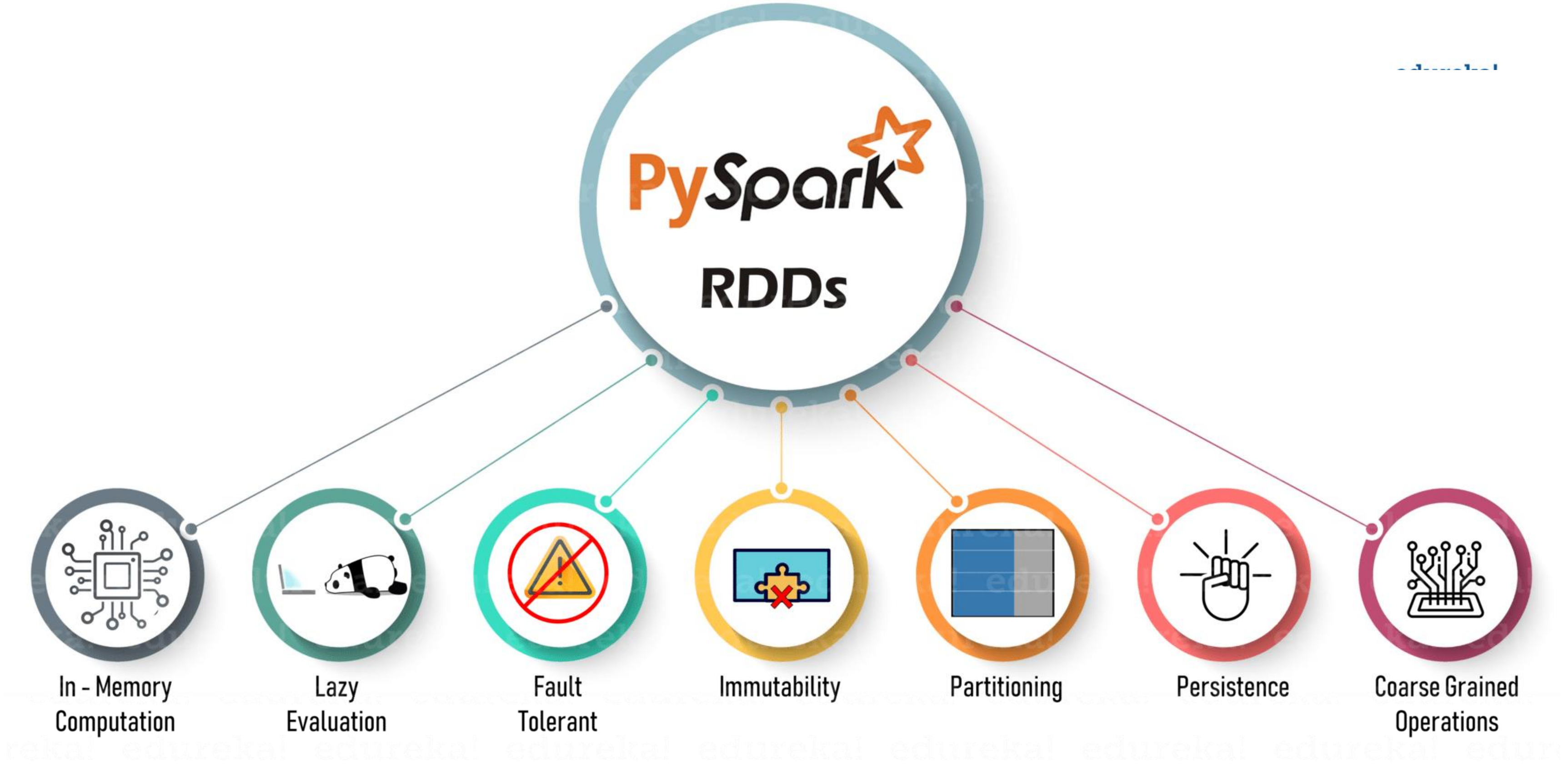
Apache Spark - RDD

- Represents a **read-only collection** of objects partitioned across a set of machines that can be rebuilt if a partition is lost.
- If a partition of an RDD is lost, the RDD has enough information about how it was derived from the other RDDs to be able to rebuild just that partition.
- Users can explicitly cache an RDD in memory across machines and reuse it in multiple Map-reduce like parallel operations

Apache Spark - RDDs

- The elements of an RDD need not exists in physical storage;
- Instead, a handle to an RDD contains enough information to compute the RDD starting from data in reliable storage.
- Advantages of using Scala for Apache Spark
  - Working with Scala is more productive than working with Java
  - Scala is faster than Python and R because it is compiled language
  - Scala is a functional language

Features of RDDs

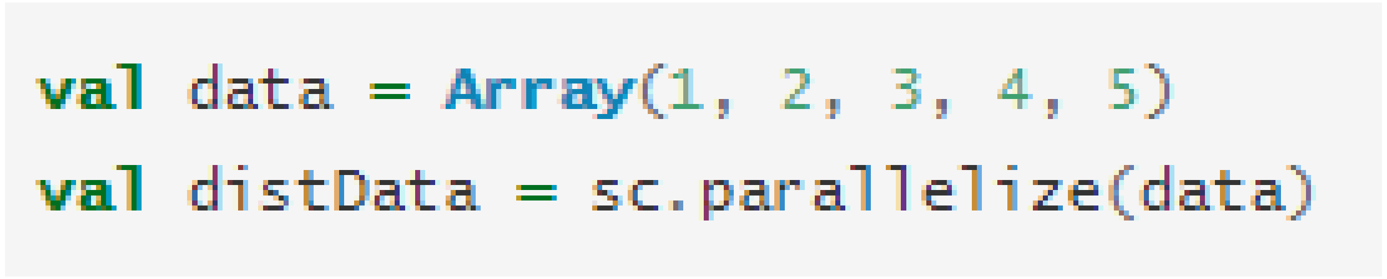


Create RDD in SPARK

- Way to create RDD
  - Parallelized collections
  - From existing RDDs
  - External Data
    - or referencing a dataset in an external storage system, such as a shared filesystem, HDFS, HBase, or any data source offering a Hadoop Input Format.

Create RDD in SPARK -Parallelized collection

- Parallelized collections are created by calling ***SparkContext's parallelize*** method on an existing collection in your driver program (a Scala Seq).
- The elements of the collection are copied to form a distributed dataset that can be operated on in parallel.
- For example, here is how to create a parallelized collection holding the numbers 1 to 5:



```
val data = Array(1, 2, 3, 4, 5)
val distData = sc.parallelize(data)
```

Create RDD in SPARK - Parallelized collection

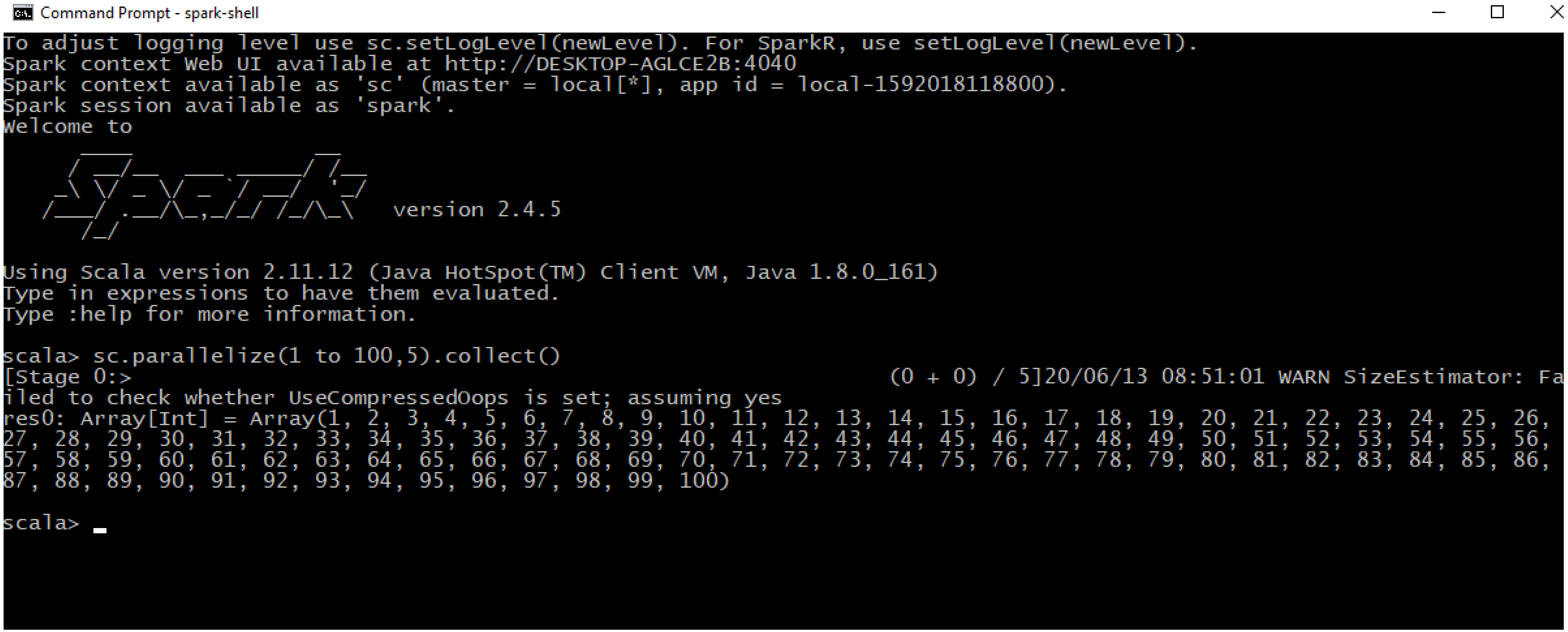
- Once created, the distributed dataset (distData) can be operated on in parallel.
- For example, we might call ***distData.reduce((a, b) => a + b)*** to add up the elements of the array.
- Number of *partitions* to cut the dataset into is the important parameter for parallel collections

Parallelized collection

- Spark will run one task for each partition of the cluster.
- Typically you want 2-4 partitions for each CPU in your cluster.
- Spark tries to set the number of partitions automatically based on your cluster.
- it can also be done manually by passing it as a second parameter to parallelize (e.g. ***sc.parallelize(data, 10)).***

Create RDDs in SPARK – Hand on Experience

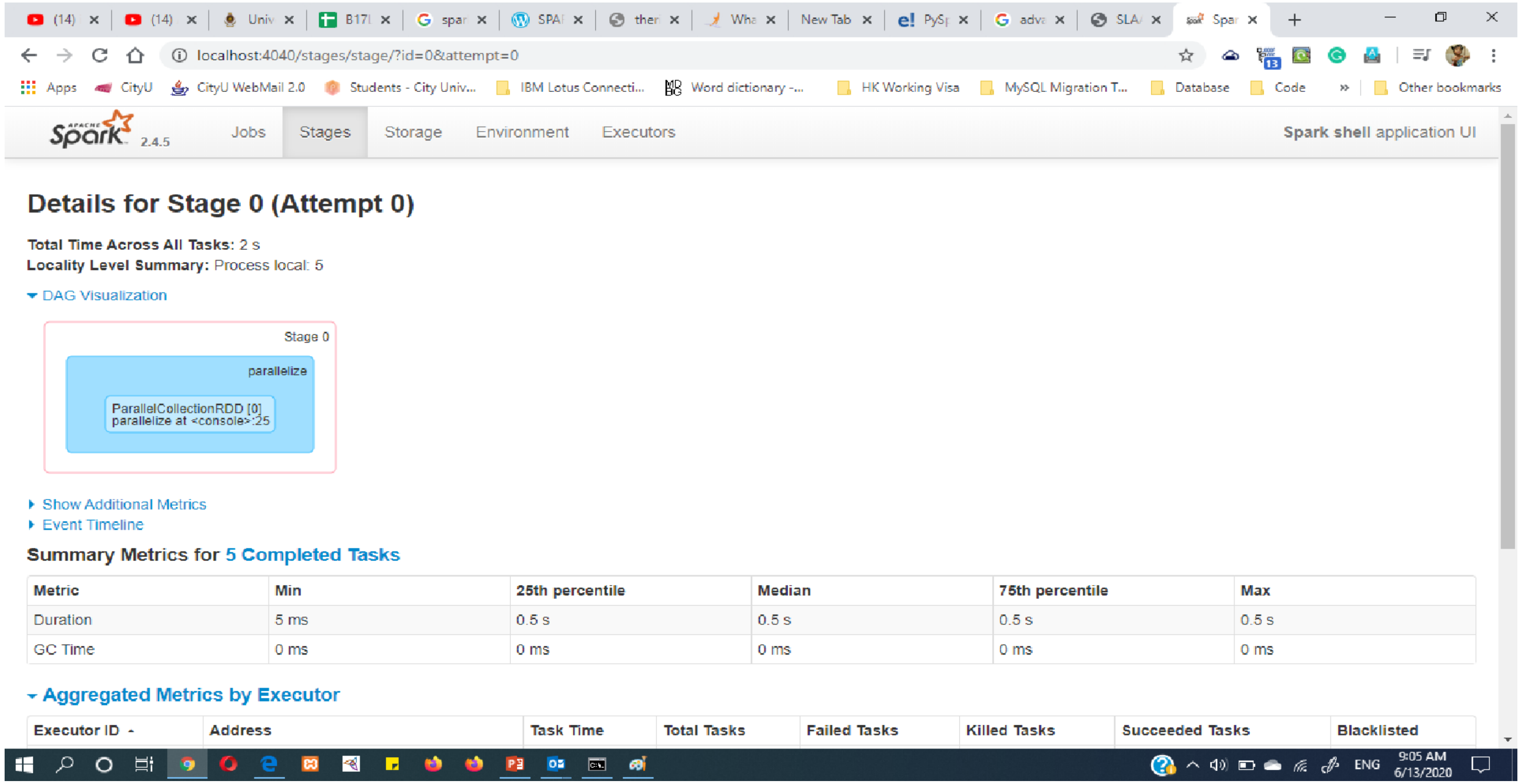
- Go to command prompt and type spark-shell
- In Scala shell apply operation sc.parallelize(1 to 100, 5).collect()
- – it is a method used to create a parallelized collection of RDD
  - This method here generate numbers between 1 to 100 in five different partitions
- collect() is used as an execution/action function



```
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
Spark context Web UI available at http://DESKTOP-AGLCE2B:4040
Spark context available as 'sc' (master = local[*], app id = local-1592018118800).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.4.5
      /_/

Using Scala version 2.11.12 (Java HotSpot(TM) Client VM, Java 1.8.0_161)
Type in expressions to have them evaluated.
Type :help for more information.

scala> sc.parallelize(1 to 100,5).collect()
[Stage 0:>                                                          (0 + 0) / 5]20/06/13 08:51:01 WARN SizeEstimator: Fa
iled to check whether UseCompressedOops is set; assuming yes
res0: Array[Int] = Array(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, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56,
57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86,
87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100)

scala> _
```

Create RDD in SPARK

- Web host for Spark is localhost:4040


localhost:4040/stages/stage/?id=0&attempt=0

Spark 2.4.5 | Jobs | Stages | Storage | Environment | Executors | **Spark shell** application UI

Details for Stage 0 (Attempt 0)

Total Time Across All Tasks: 2 s
Locality Level Summary: Process local: 5

DAG Visualization

Show Additional Metrics
Event Timeline

Summary Metrics for 5 Completed Tasks

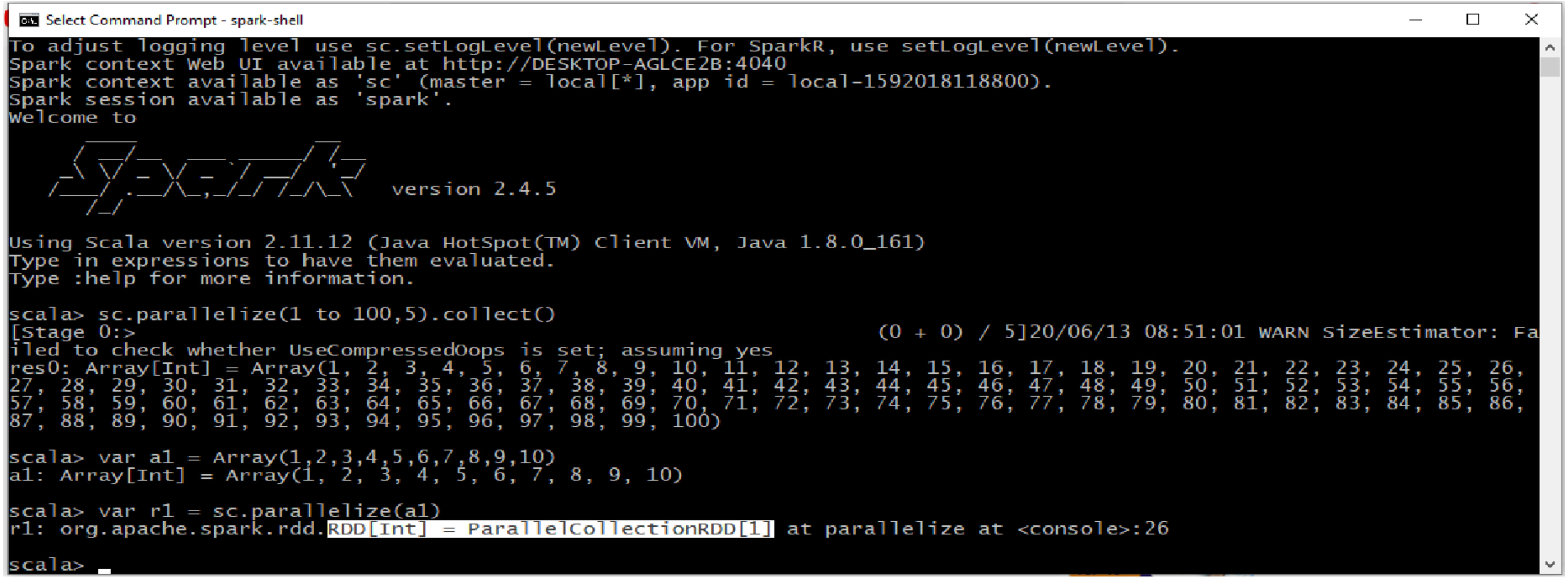
Metric	Min	25th percentile	Median	75th percentile	Max
Duration	5 ms	0.5 s	0.5 s	0.5 s	0.5 s
GC Time	0 ms	0 ms	0 ms	0 ms	0 ms

Aggregated Metrics by Executor

Executor ID	Address	Task Time	Total Tasks	Failed Tasks	Killed Tasks	Succeeded Tasks	Blacklisted

Create RDD in SPARK - Using Existing RDDs

- Create an array in Scala
  - `scala> var a1 = Array(1,2,3,4,5,6,7,8,9,10)`
- Parallelize the array a1
  - `scala> var r1 = sc.parallelize(a1)`
  - `scala> val newrdd = sc.parallelize(a1.map(data => (data*2)))`
    - `newrdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[1] at parallelize at <console>:2`



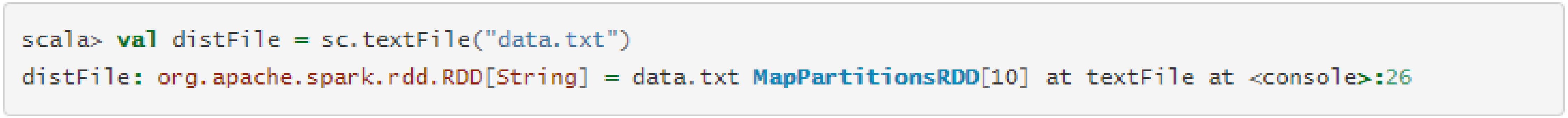
Select Command Prompt - spark-shell

```
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
Spark context Web UI available at http://DESKTOP-AGLCE2B:4040
Spark context available as 'sc' (master = local[*], app id = local-1592018118800).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.4.5
      /_/

Using Scala version 2.11.12 (Java HotSpot(TM) Client VM, Java 1.8.0_161)
Type in expressions to have them evaluated.
Type :help for more information.

scala> sc.parallelize(1 to 100,5).collect()
[Stage 0:>                                                          (0 + 0) / 5]20/06/13 08:51:01 WARN SizeEstimator: Fa
iled to check whether UseCompressedOops is set; assuming yes
res0: Array[Int] = Array(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, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56,
57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86,
87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100)

scala> var a1 = Array(1,2,3,4,5,6,7,8,9,10)
a1: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> var r1 = sc.parallelize(a1)
r1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[1] at parallelize at <console>:26

scala> _
```

External Datasets

- Spark can create distributed datasets from any storage source supported by Hadoop, including your local file system, HDFS, Cassandra, HBase, etc..
- Text file RDDs can be created using *SparkContext's textFile method*
- This method takes an URI for the file (either a local path on the machine) and reads it as a collection of lines.

External Datasets



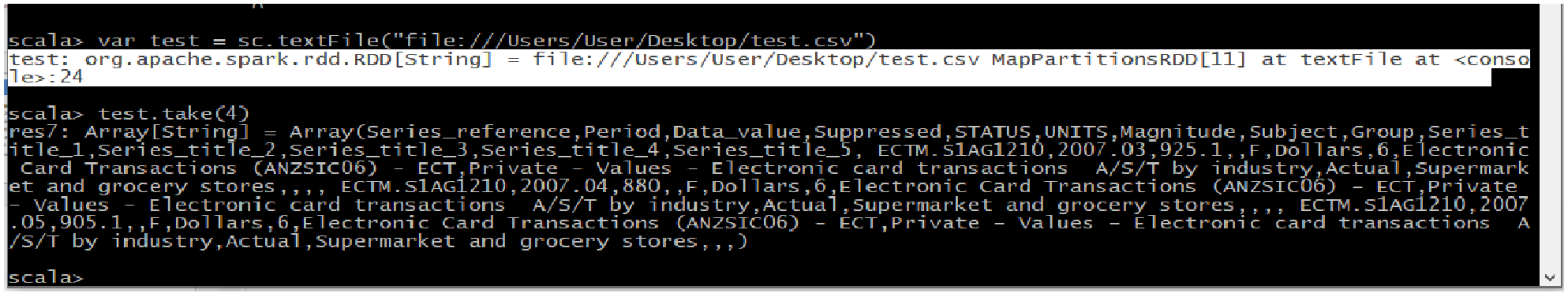
```
scala> val distFile = sc.textFile("data.txt")
distFile: org.apache.spark.rdd.RDD[String] = data.txt MapPartitionsRDD[10] at textFile at <console>:26
```

- Once created, distFile can be acted on by dataset operations.
- For example,
  - we can add up the sizes of all the lines using the map and reduce operations as follows:

    distFile.map(s => s.length).reduce((a, b) => a + b).

Create RDD in SPARK - External Data

– scala> var test = sc.textFile("file:///Users/User/Desktop/test.csv")
  test: org.apache.spark.rdd.RDD[String] = C:/Users/User/Desktop/test MapPartitionsRDD[3] at textFile at <console>:24



```
scala> var test = sc.textFile("file:///Users/User/Desktop/test.csv")
test: org.apache.spark.rdd.RDD[String] = file:///Users/User/Desktop/test.csv MapPartitionsRDD[11] at textFile at <conso
le>:24

scala> test.take(4)
res7: Array[String] = Array(Series_reference,Period,Data_value,Suppressed,STATUS,UNITS,Magnitude,Subject,Group,Series_t
itle_1,Series_title_2,Series_title_3,Series_title_4,Series_title_5, ECTM.S1AG1210,2007.03,925.1,,F,Dollars,6,Electronic
 Card Transactions (ANZSIC06) - ECT,Private - Values - Electronic card transactions  A/S/T by industry,Actual,Supermark
et and grocery stores,,,, ECTM.S1AG1210,2007.04,880,,F,Dollars,6,Electronic Card Transactions (ANZSIC06) - ECT,Private
- Values - Electronic card transactions  A/S/T by industry,Actual,Supermarket and grocery stores,,,, ECTM.S1AG1210,2007
.05,905.1,,F,Dollars,6,Electronic Card Transactions (ANZSIC06) - ECT,Private - Values - Electronic card transactions  A
/S/T by industry,Actual,Supermarket and grocery stores,,,)

scala>
```

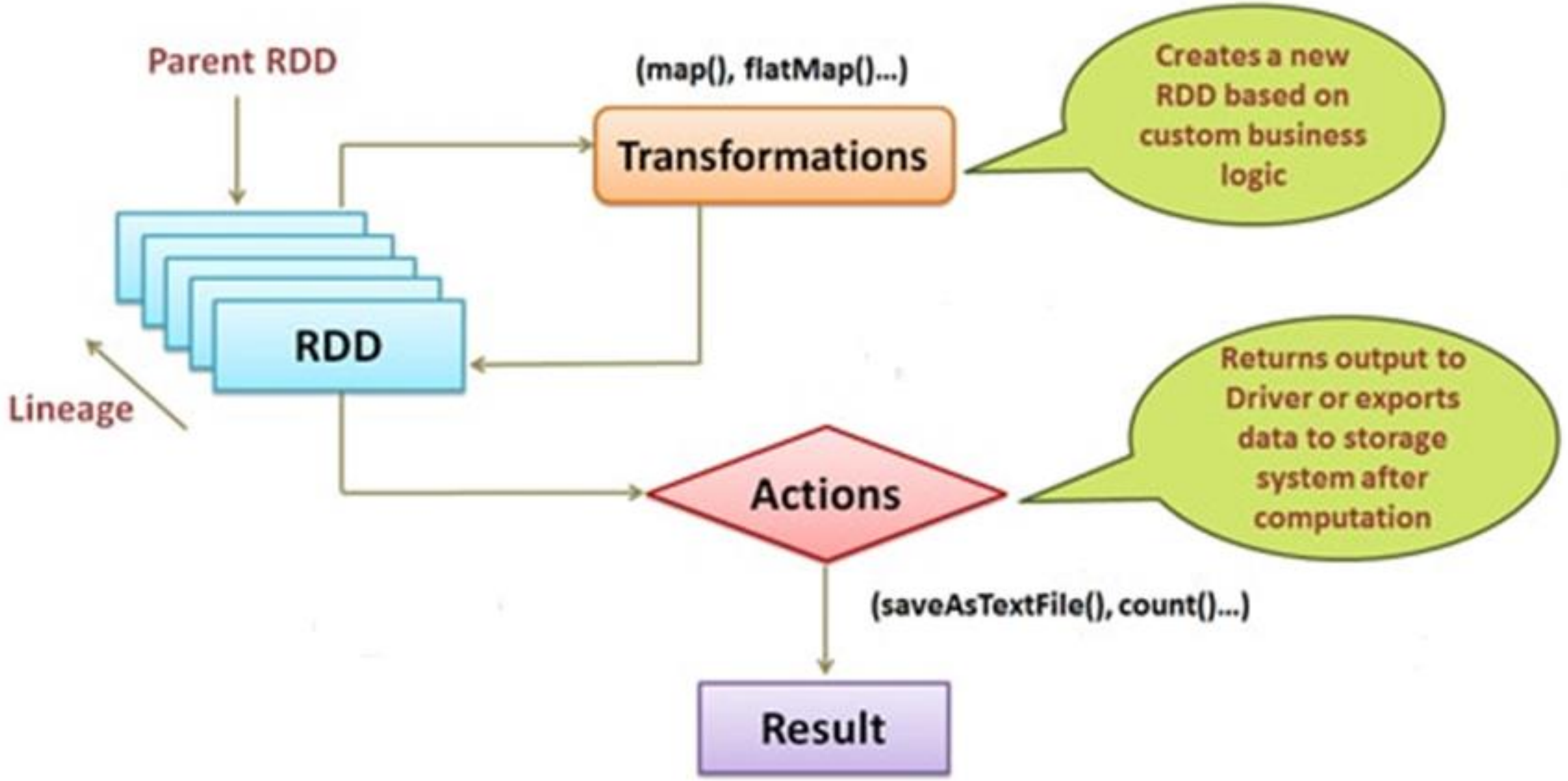
- Try with commands

  scala> test.take(4)

  scala> test.take(4).foreach(println)

  scala> test.first

Apache Spark - RDD Operations


Parent RDD
(map(), flatMap()...)
Transformations
Creates a new RDD based on custom business logic
RDD
Lineage
Actions
Returns output to Driver or exports data to storage system after computation
(saveAsTextFile(), count()...)
Result

Apache Spark - RDD Operations

- ***Transformations***: These are the operations which are applied to an RDD to create a new RDD.
- Transformations follow the principle of **Lazy Evaluations** (which means that the execution will not start until an action is triggered).
- Let you execute the operations at any time by just calling an action on the data.
- Few of the transformations provided by RDDs are:
  - map
  - flatMap
  - filter
  - distinct
  - reduceByKey
  - mapPartitions
  - sortBy

Apache Spark - RDD Operations

- **Actions:** Actions are the operations which are applied on an RDD to instruct Apache Spark to apply computation and pass the result back to the driver. Few of the actions include:
  - collect
  - collectAsMap
  - reduce
  - countByKey/countByValue
  - take
  - first

Apache Spark - RDD Operations

- **Creating RDDs with key-value pair**

```
scala> var array1 = Array(('a',2),('b',3))
scala> var array2 = Array(('a',9),('b',7),('c',10))

scala> var a = sc.parallelize(array1)
scala> var b = sc.parallelize(array2)
scala> var c = a.join(b)
scala> c.collect()

[Stage 5:===============>
(1
res10: Array[(Char, (Int, Int))] = Array((a,(2,9)), (b,(3,7)))
```

Apache Spark - RDD Operations

- Defining a function in Spark – Scala Shell

- `scala> def myConcat(word1: String)(word2: String): String = {word1 + word2}`
- `scala> myConcat("beautiful ")("picture")`
  - `res13: String = beautiful picture`

RDD Operations

- All transformations in ***Spark are lazy***, so they do not compute their results right away.
- Instead, they just remember the transformations applied to some base dataset (e.g. a file).
- The transformations are only computed when an action requires a result to be returned to the driver program.

RDD Operations

- This design enables Spark to run more efficiently.
  - e.g., we can realize that a dataset created through *map* will be used in a reduce and return only the result of the reduce to the driver, rather than the larger mapped dataset.
- By default, each transformed RDD may be recomputed each time you run an action on it.

RDD Operations

- However, you may also **persist an RDD** in memory using the persist (or cache) method
- Spark will keep the persist RDD elements around on the cluster for much faster access the next time you query it.
- There is also support for persisting RDDs on disk, or replicated across multiple nodes.

RDD Operations

- Consider the simple program below:

```
val lines = sc.textFile("data.txt")
val lineLengths = lines.map(s => s.length)
val totalLength = lineLengths.reduce((a, b) => a + b)
```

- Spark does not count the lineLengths due to laziness.
- Finally, we run reduce, which is an action.
- At this point Spark breaks the computation into tasks to run on separate machines, and each machine runs both its part of the map and a local reduction, returning only its answer to the driver program.

```
lineLengths.persist()
```

Transformation

transformation	**Meaning**
map(*func*)	Return a new distributed dataset formed by passing each element of the source through a function *func*.
filter(*func*)	Return a new dataset formed by selecting those elements of the source on which *func* returns true.
flatMap(*func*)	Similar to map, but each input item can be mapped to **0 or more output** items (so *func* should return a Seq rather than a single item).

Transformation

sample(*withReplacement*, *fraction*, *seed*)	Sample a fraction (*fraction)* of the data, with or without replacement, using a given random number generator seed.
union(*otherDataset*)	Return a new dataset that contains the union of the elements in the source dataset and the argument.
intersection(*otherDataset*)	Return a new RDD that contains the intersection of elements in the source dataset and the argument.
distinct([*numPartitions*]))	Return a new dataset that contains the distinct elements of the source dataset.
groupByKey([*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. **Note:** If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using reduceByKey or aggregateByKey will yield much better performance. **Note:** By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional numPartitions argument to set a different number of tasks.

Transformation

reduceByKey(*func*, [*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type **(V, V) => V**. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.
aggregateByKey(*zeroValue*)(*seqOp*, *combOp*, [*numPartitions*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.
sortByKey([*ascending*], [*numPartitions*])	When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument.

Actions

Action	Meaning
reduce(*func*)	Aggregate the elements of the dataset using a function *func* (**which takes two arguments and returns one**). The function should be commutative and associative so that it can be computed correctly in parallel.
collect()	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
count()	Return the number of elements in the dataset.
first()	Return the first element of the dataset (similar to take(1)).
take(*n*)	Return an array with the first *n* elements of the dataset.

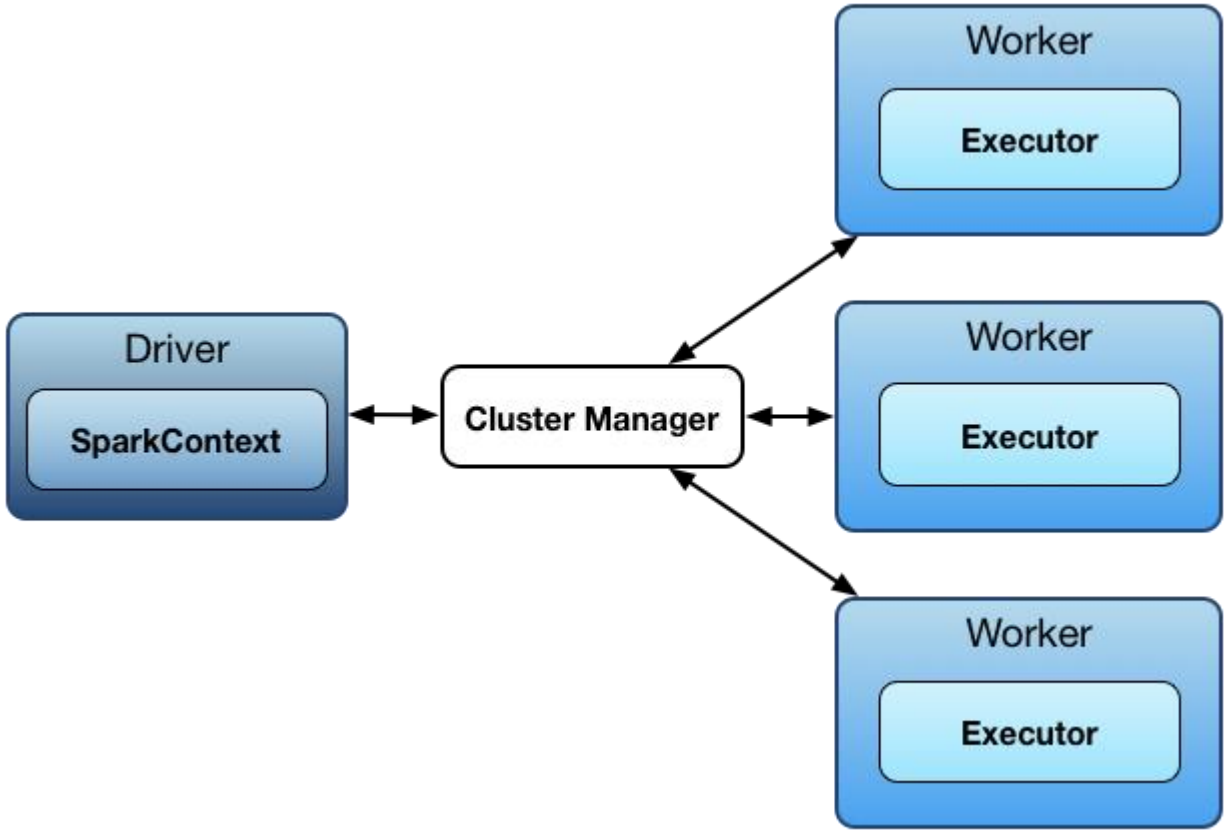
Actions

Action	Meaning
takeSample(*withReplacement*, *num*, [*seed*])	Return an array with a random sample of *num* elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.
takeOrdered(*n*, *[ordering]*)	Return the first *n* elements of the RDD using either their natural order or a custom comparator.
saveAsTextFile(*path*)	Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file.

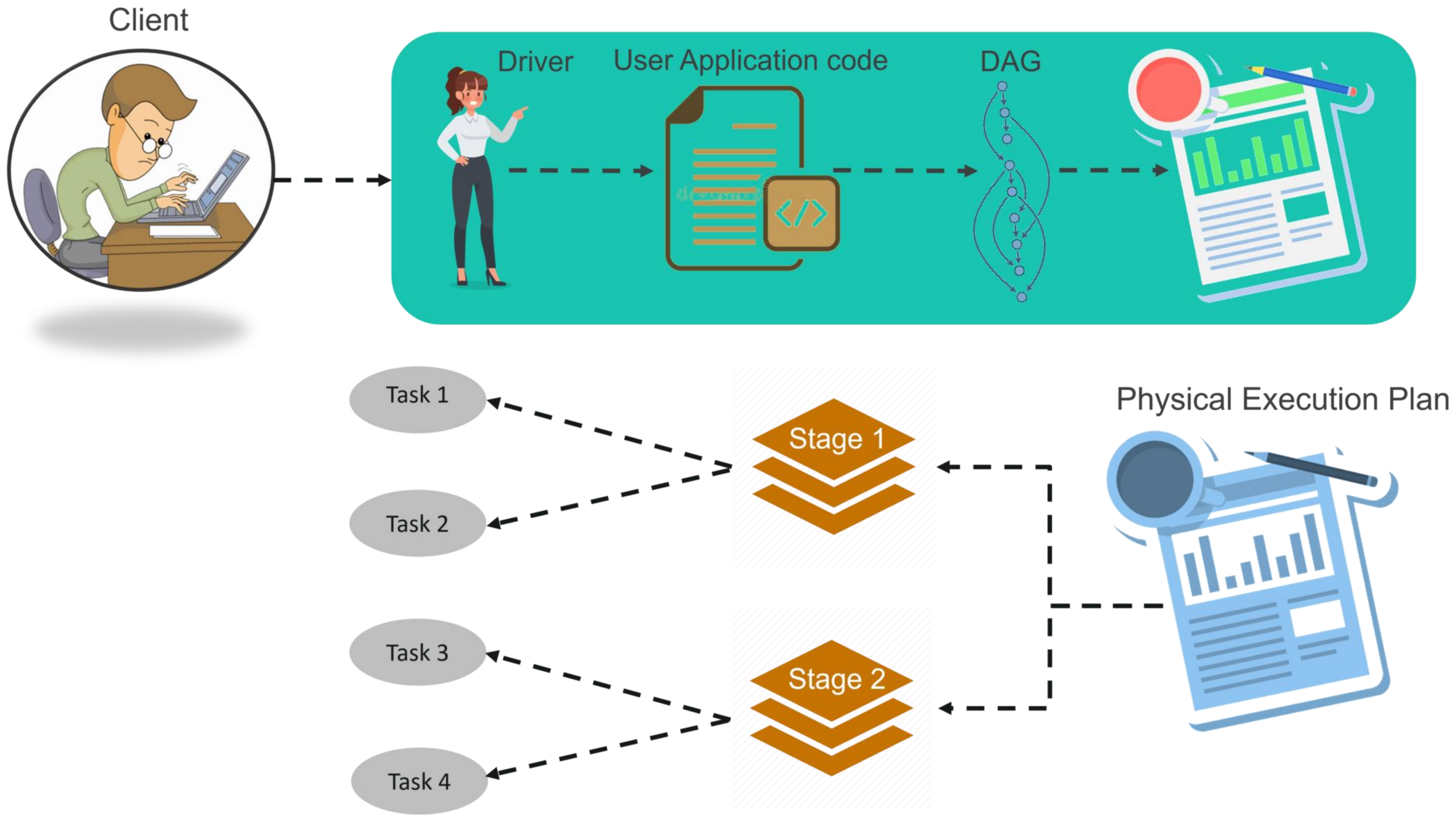
SPARK Architecture

Spark Context

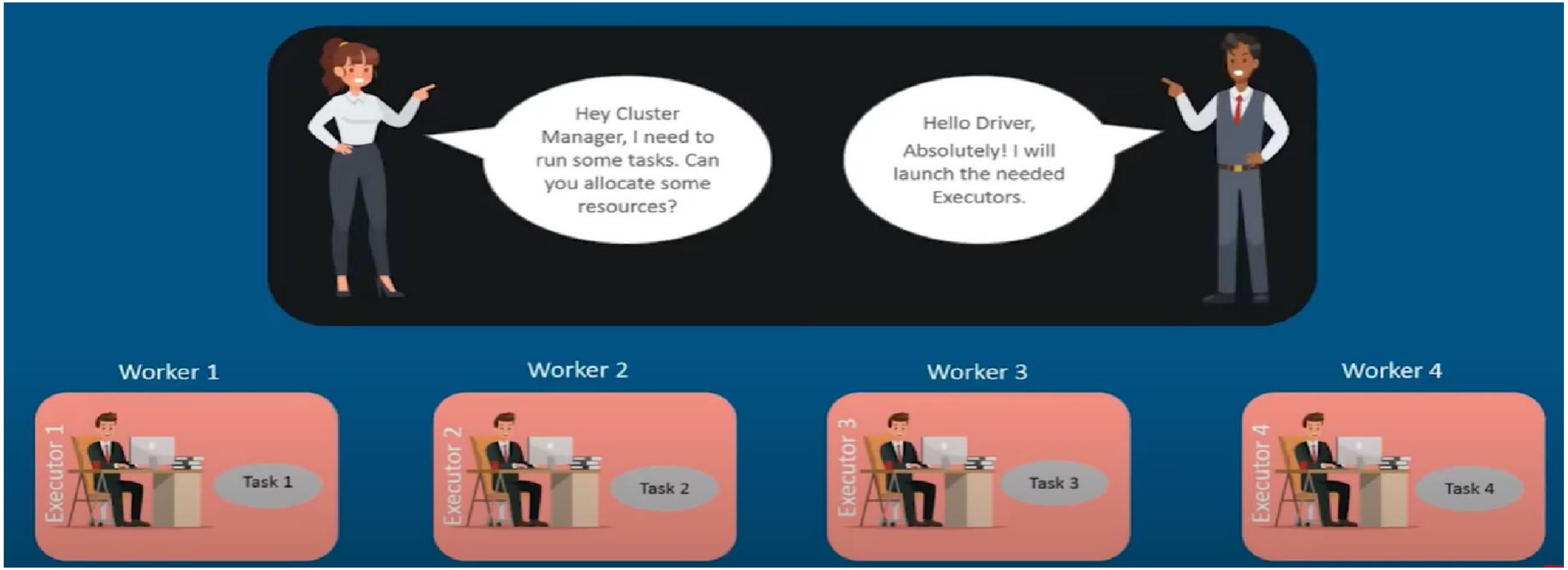
1. **Heart of the Spark application.**
2. **Establishes the connection to the Spark Execution Environment.**



SPARK Architecture



SPARK Architecture


Hey Cluster Manager, I need to run some tasks. Can you allocate some resources?
Hello Driver, Absolutely! I will launch the needed Executors.
Worker 1
Worker 2
Worker 3
Worker 4
Executor 1
Executor 2
Executor 3
Executor 4
Task 1
Task 2
Task 3
Task 4

Word Count problem in SPARK

- `scala> var map = sc.textFile("file:///Users/User/Desktop/testsample.txt").flatMap(line =>line.split(" ")).map(word => (word,1))`

- `scala> var counts = map.reduceByKey(_+_)`
- `scala> counts.saveAsTextFile("file:///Users/User/Desktop/Output9")`

- `scala> map.collect()`

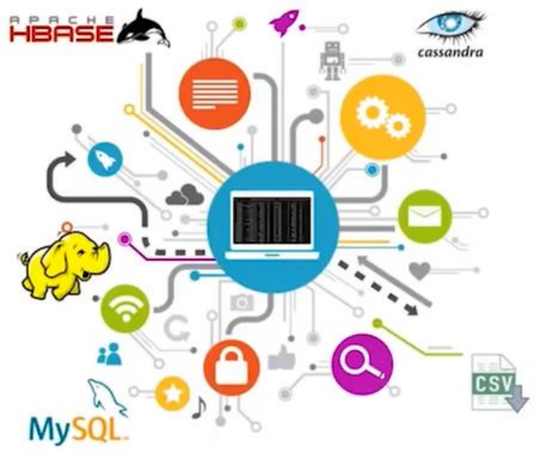
```
res30: Array[(String, Int)] = Array((hadoop,1), (research,1), (data,1), (science,1), (science,1), (Spark,1), (yarn,1), (This,1), (is,1), (a,1), (text,1), (file,1))
```

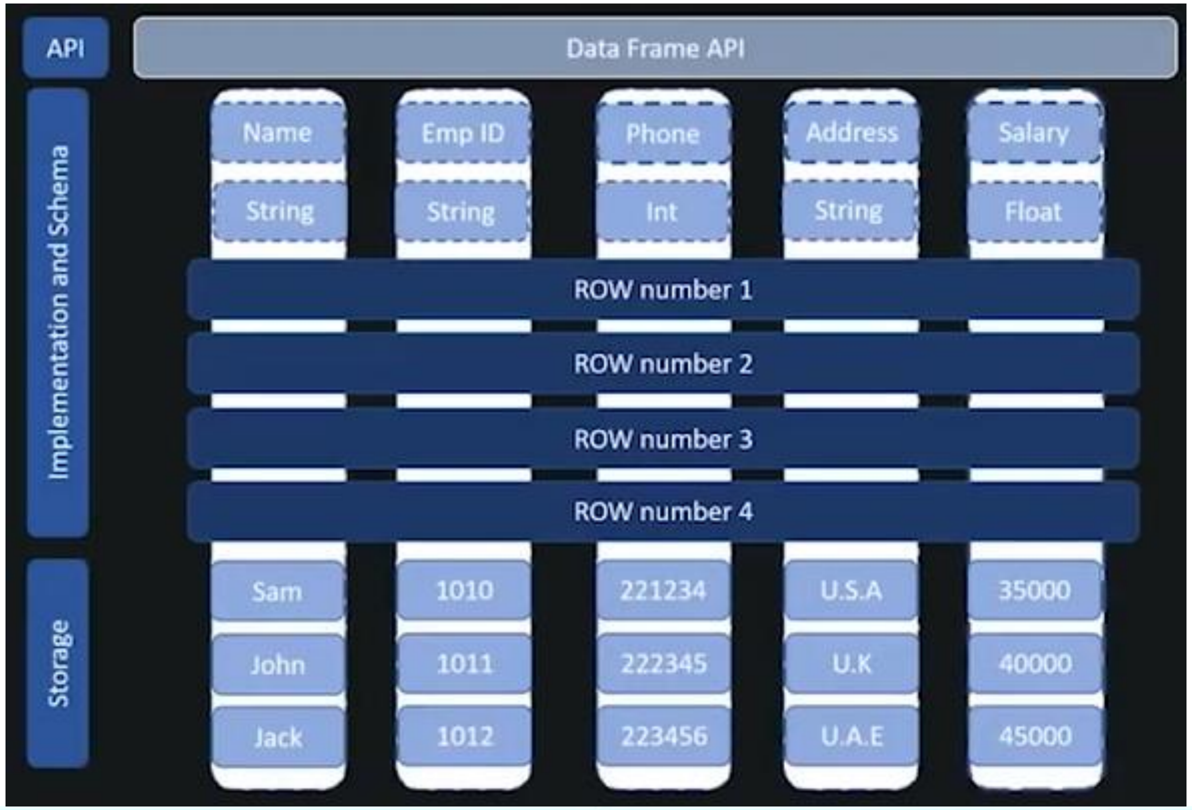
- `scala> counts.collect()`

```
res32: Array[(String, Int)] = Array((is,1), (data,1), (This,1), (science,2), (yarn,1), (file,1), (Spark,1), (hadoop,1), (a,1), (text,1), (research,1))
```

What is a Data Frame?

- Distributed collection of data
- Organized in rows under named columns
- Filter, Group, Aggregate operations are performed
- Used with SparkSQL
- Constructed from multiple sources

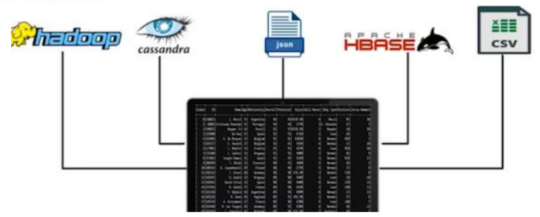


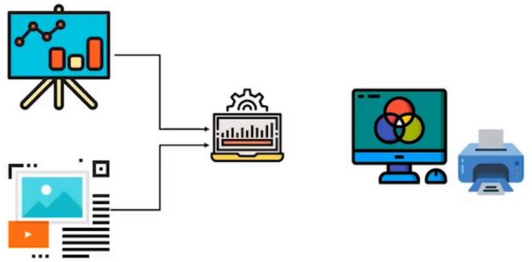


Entity name	Data type
Name of the employee	string datatype
Employee-ID	string datatype
Employee phone number	integer datatype
Employee address	string datatype
Employee salary	float datatype

Data Frames features

- Support for multiple programming languages
  - Scala, Java, Python and R
- Support for a large variety of sources of data





- Processing structured and un-structured data
  - Process big data with ease
  - Uses a table format to store the data along with its schema
- Slicing and Dicing data
  - DataFrame APIs support operations like select and filter upon rows and columns.

Data Frames features

- **Immutability**
  - Data frames are immutable like RDDs
  - Data cannot be altered. Need to apply simple transformation when alteration is required.
- Lazy evaluation
- Fault tolerance
- Distributed storage

Creating data frames

-
  ```
  scala> import org.apache.spark.sql.types._
     import org.apache.spark.sql.types._
  ```

-
  ```
  scala> import org.apache.spark.storage.StorageLevel
        import org.apache.spark.storage.StorageLevel
  ```

-
  ```
  scala> import scala.io.Source
         import scala.io.Source
  ```

-
  ```
  scala> import scala.collection.mutable.HashMap
    import scala.collection.mutable.HashMap
  ```

-
  ```
  scala> import java.io.File
     import java.io.File
  ```

Creating data frames

-
  ```
  scala> import scala.collection.mutable.ListBuffer
      import scala.collection.mutable.ListBuffer
  ```

-
  ```
  scala> import org.apache.spark.util.IntParam
  import org.apache.spark.util.IntParam
  ```

```
scala> import org.apache.spark.sql.Row
import org.apache.spark.sql.Row
```

```scala
import org.apache.spark.sql.types._

val data = Seq(
  Row(8, "bat"),
  Row(64, "mouse"),
  Row(-27, "horse")
)

val schema = StructType(
  List(
    StructField("number", IntegerType, true),
    StructField("word", StringType, true)
  )
)

val df = spark.createDataFrame(
  spark.sparkContext.parallelize(data),
  schema
)
```

The org.apache.spark.sql.types package must be imported to access StructType, StructField, IntegerType, and StringType.

The StructField above sets the name field to "word", the dataType field to StringType, and the nullable field to true.

StructType objects are instantiated with a List of **StructField** objects.

The createDataFrame() method takes two arguments:

1. RDD of the data
2. The DataFrame schema (a StructType object)

Creating data frame

-
  ```
  scala> val Employee = Seq(
          Row("Mike", "Robert","Mike09@gmil.com",10000),
          Row("Jhon","Milers","Jhon09@gmail.com",20000),
          Row("Brett","Lee","Brett09@gmail.com",25000),
          Row("Letty","Brown","Brown09@gmail.com",35000))
  ```

-
  ```
  Employee: Seq[org.apache.spark.sql.Row] =
  List([Mike,Robert,Mike09@gmil.com,10000],
  [Jhon,Milers,Jhon09@gmail.com,20000],
  [Brett,Lee,Brett09@gmail.com,25000],
  [Letty,Brown,Brown09@gmail.com,35000])
  ```

Creating data frame

-
  ```
  scala> val EmployeeSchema = List(
      StructField("FirstName", StringType,true),
      StructField("LastName",StringType, true),
      StructField("Mailaddress",StringType,true),
      StructField("Salary",IntegerType,true))
  ```

```
EmployeeSchema:
List[org.apache.spark.sql.types.StructField] =
List(StructField(FirstName,StringType,true),
StructField(LastName,StringType,true),
StructField(Mailaddress,StringType,true),
StructField(Salary,IntegerType,true))
```

Creating data frame

- `scala> val EmpDF = spark.createDataFrame(spark.sparkContext.parallelize(Employee),StructType(EmployeeSchema))`

```
EmpDF: org.apache.spark.sql.DataFrame = [FirstName:
string, LastName: string ... 2 more fields]
```

Operations on data frame

- scala> EmpDF.show()

```
+---------+--------+-----------------+------+
|FirstName|LastName|      Mailaddress|Salary|
+---------+--------+-----------------+------+
|     Mike|  Robert|  Mike09@gmil.com| 10000|
|     Jhon|  Milers| Jhon09@gmail.com| 20000|
|    Brett|     Lee|Brett09@gmail.com| 25000|
|    Letty|   Brown|Brown09@gmail.com| 35000|
+---------+--------+-----------------+------+
```

Operations on data frame

- `scala> EmpDF.printSchema()`

```
root
 |-- FirstName: string (nullable = true)
 |-- LastName: string (nullable = true)
 |-- Mailaddress: string (nullable = true)
 |-- Salary: integer (nullable = true)
```

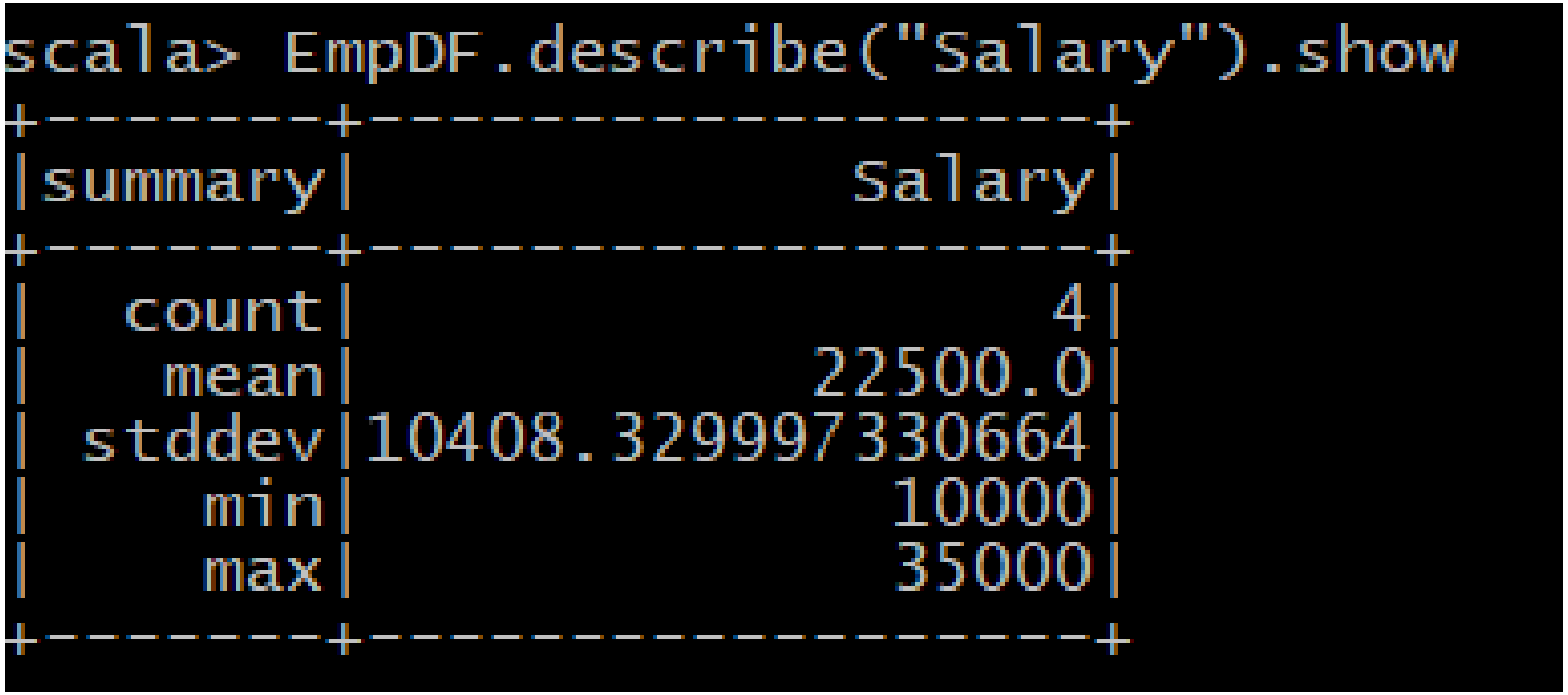
Operations on data frame

-
  ```
  scala> EmpDF.count()
   res6: Long = 4
  ```

-
  ```
  scala> EmpDF.columns.foreach(println)
        FirstName
        LastName
        Mailaddress
        Salary
  ```

Operations on data frame

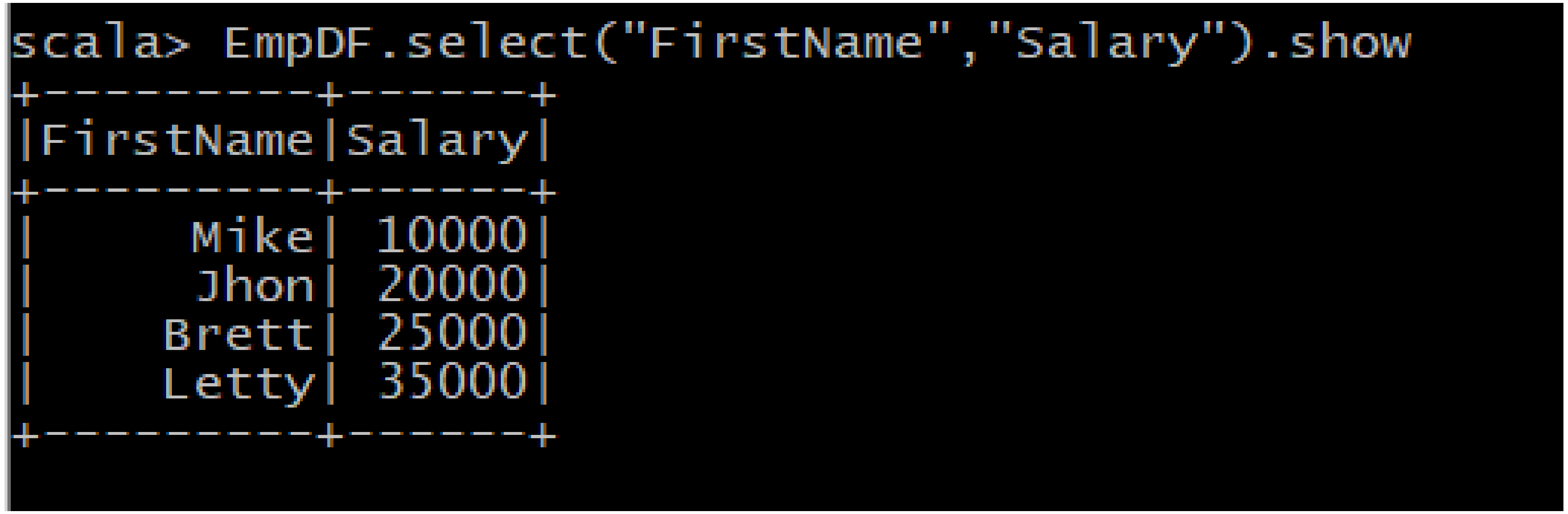
- scala> EmpDF.describe("Salary").show



```
scala> EmpDF.describe("Salary").show
+-------+------------------+
|summary|            Salary|
+-------+------------------+
|  count|                 4|
|   mean|           22500.0|
| stddev|10408.329997330664|
|    min|             10000|
|    max|             35000|
+-------+------------------+
```

Operations on data frame

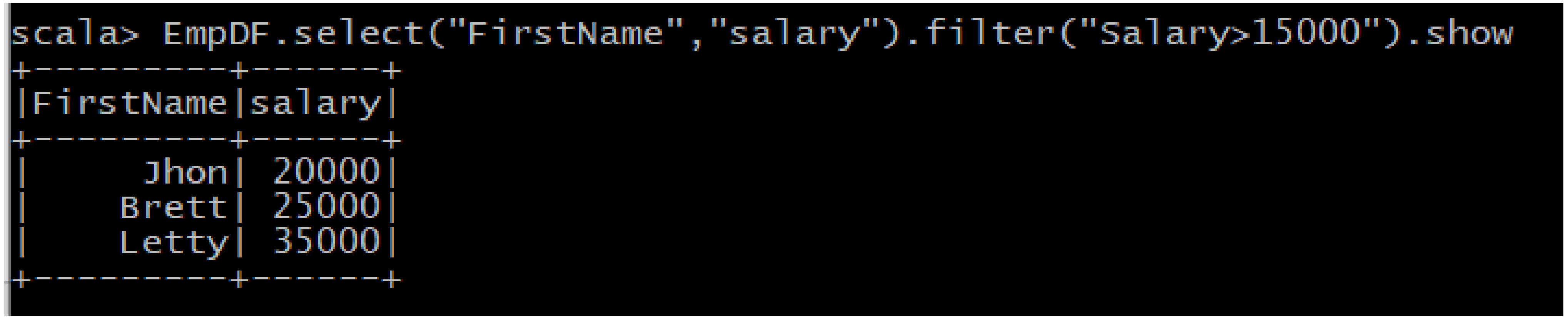
- scala> EmpDF.select("FirstName","Salary").show



```
scala> EmpDF.select("FirstName","Salary").show
+---------+------+
|FirstName|Salary|
+---------+------+
|     Mike| 10000|
|     Jhon| 20000|
|    Brett| 25000|
|    Letty| 35000|
+---------+------+
```

Operations on data frame

- scala> EmpDF.select("FirstName","salary").filter("Salary>15000").show



```
scala> EmpDF.select("FirstName","salary").filter("Salary>15000").show
+---------+------+
|FirstName|salary|
+---------+------+
|     Jhon| 20000|
|    Brett| 25000|
|    Letty| 35000|
+---------+------+
```