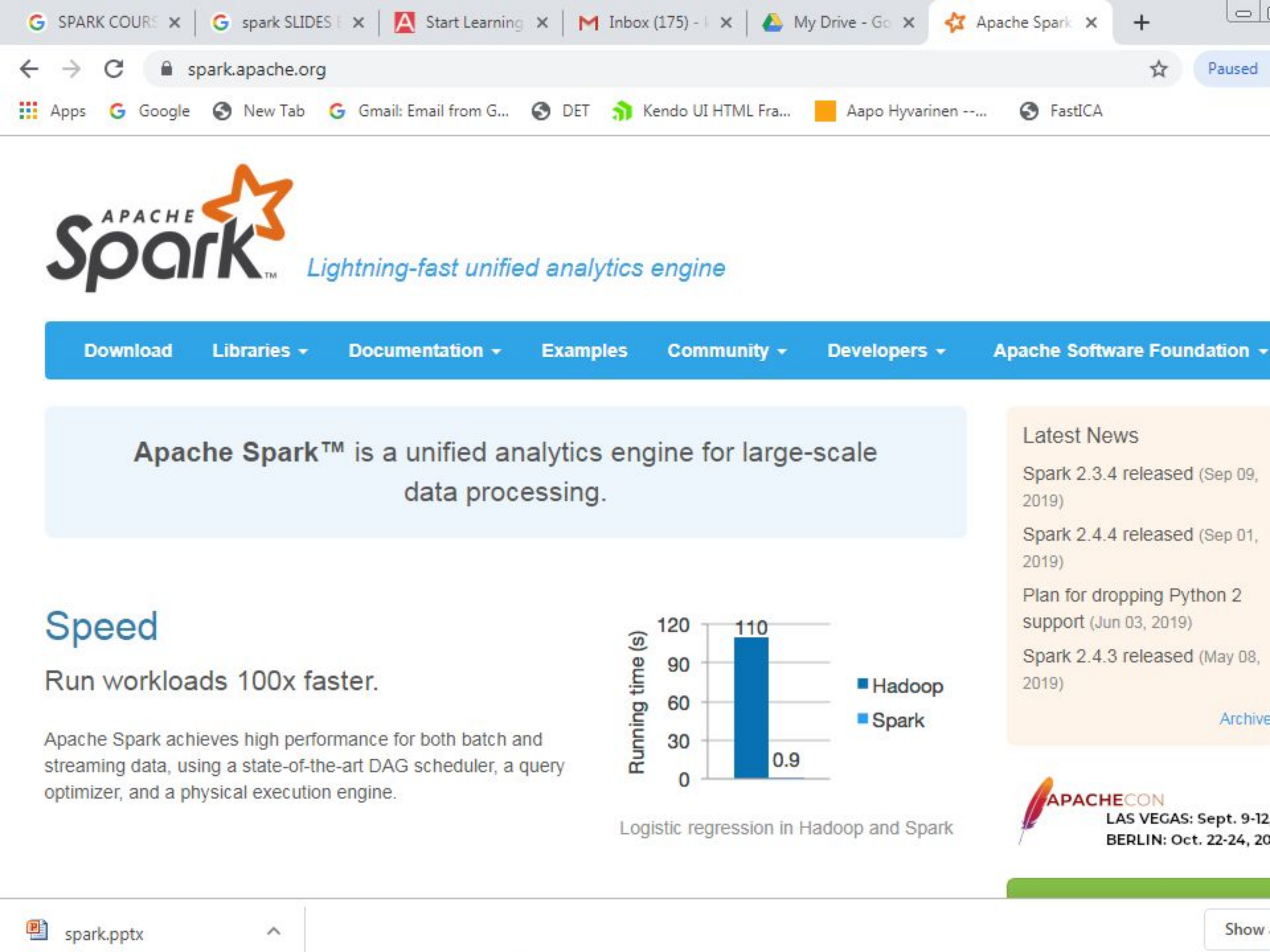
Spark: Importance of Spark Framework, Components of the Spark unified stack,
Batch and Real-Time Analytics with Apache Spark, Resilient Distributed Dataset
(RDD),

Scala: Scala,Environment Set up, Downloading and installing Spark standalone, Functional
Programming, Collections.



Lightning-fast unified analytics engine

Apache Spark™ is a unified analytics engine for large-scale data processing.

Speed

Run workloads 100x faster.

Apache Spark achieves high performance for both batch and streaming data, using a state-of-the-art DAG scheduler, a query optimizer, and a physical execution engine.

Logistic regression in Hadoop and Spark

Latest News

Spark 2.3.4 released (Sep 09, 2019)

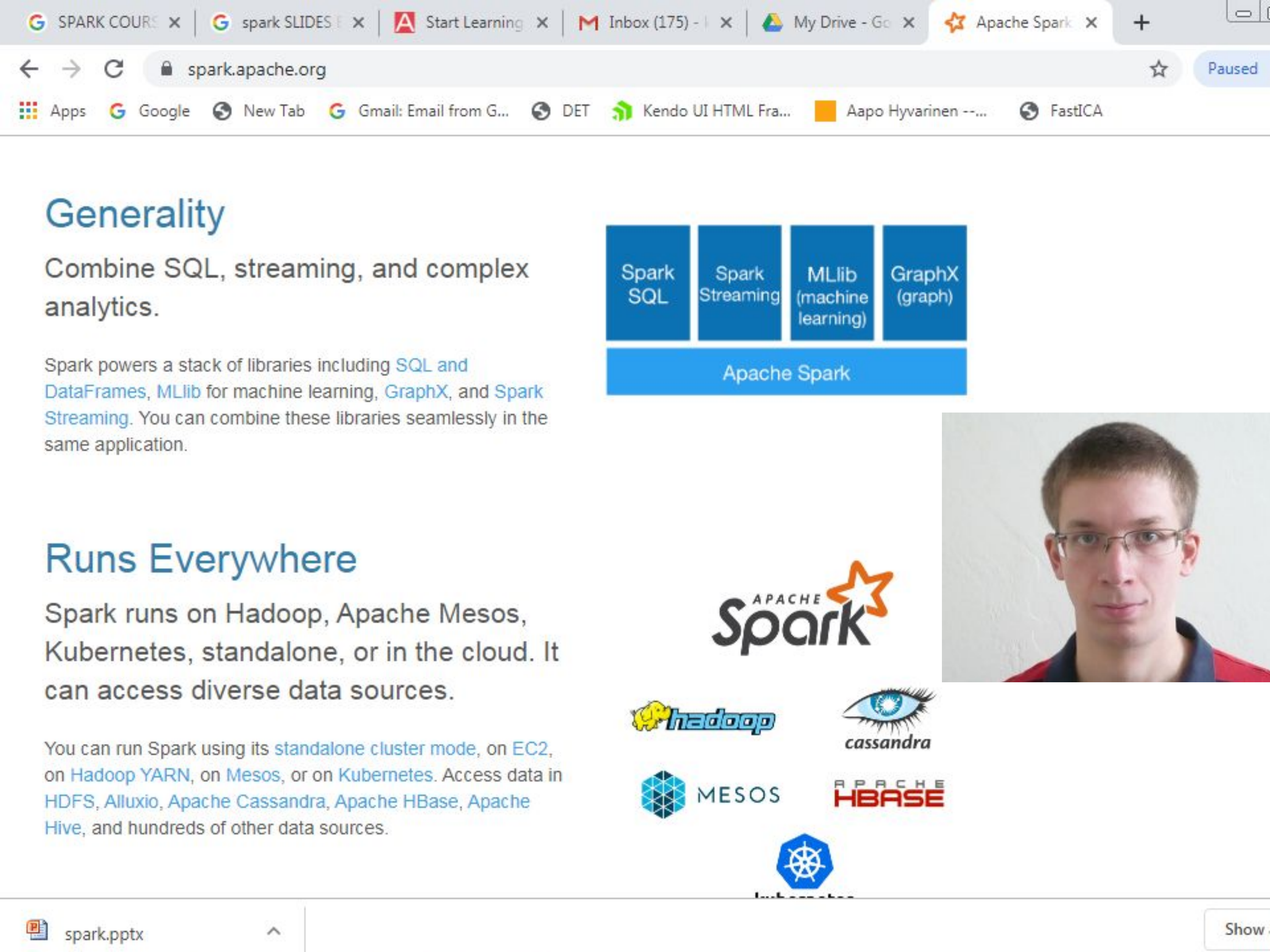
Spark 2.4.4 released (Sep 01, 2019)

Plan for dropping Python 2 support (Jun 03, 2019)

Spark 2.4.3 released (May 08, 2019)

Archive

APACHECON
LAS VEGAS: Sept. 9-12
BERLIN: Oct. 22-24, 20



Generality

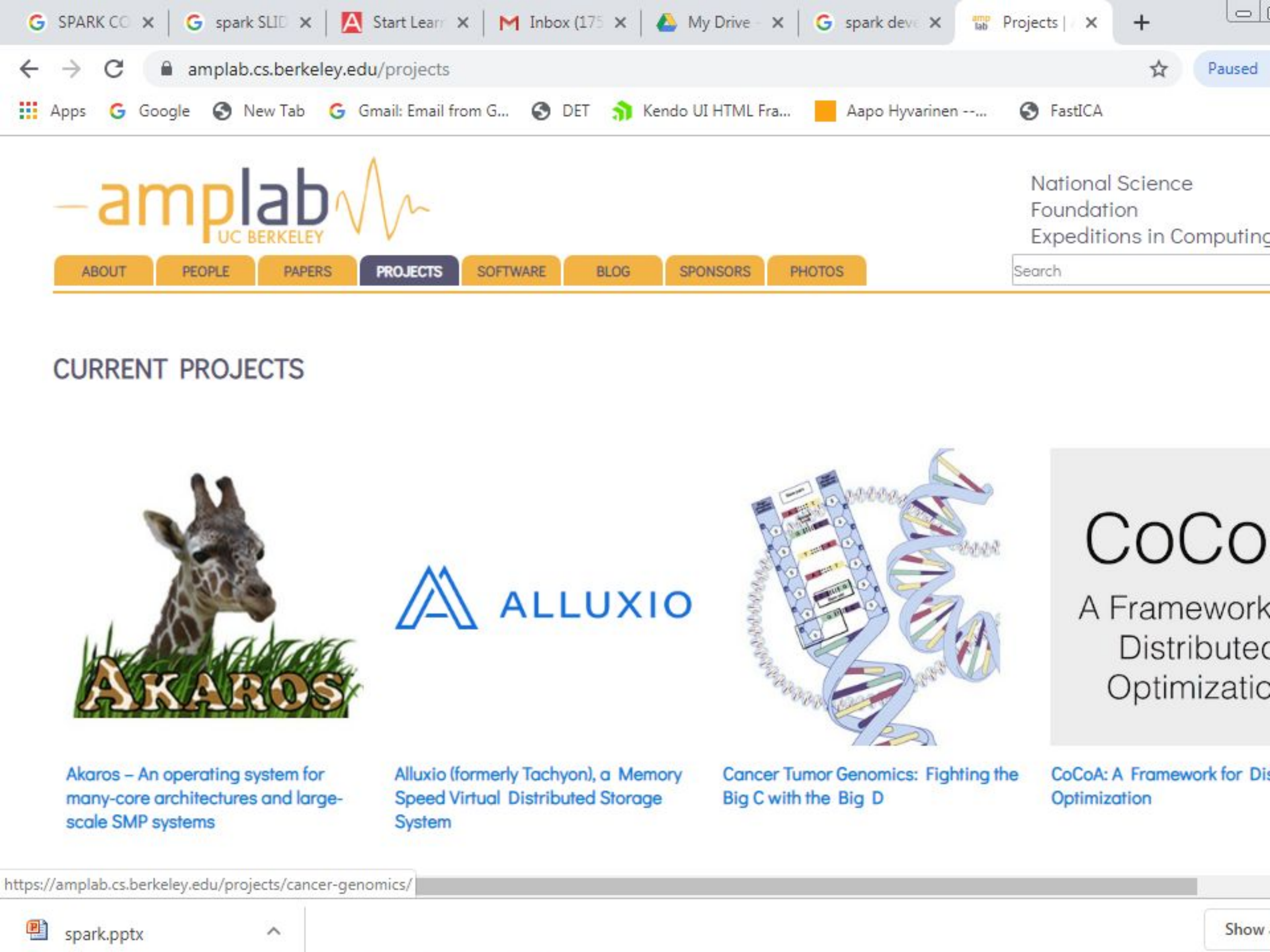
Combine SQL, streaming, and complex analytics.

Spark powers a stack of libraries including SQL and DataFrames, MLlib for machine learning, GraphX, and Spark Streaming. You can combine these libraries seamlessly in the same application.

Runs Everywhere

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

You can run Spark using its standalone cluster mode, on EC2, on Hadoop YARN, on Mesos, or on Kubernetes. Access data in HDFS, Alluxio, Apache Cassandra, Apache HBase, Apache Hive, and hundreds of other data sources.



amplab.cs.berkeley.edu/projects

amplab UC BERKELEY

National Science Foundation Expeditions in Computing

ABOUT | PEOPLE | PAPERS | PROJECTS | SOFTWARE | BLOG | SPONSORS | PHOTOS

Search

CURRENT PROJECTS

Akaros – An operating system for many-core architectures and large-scale SMP systems

Alluxio (formerly Tachyon), a Memory Speed Virtual Distributed Storage System

Cancer Tumor Genomics: Fighting the Big C with the Big D

CoCoA: A Framework for Dis Optimization

https://amplab.cs.berkeley.edu/projects/cancer-genomics/

What is Apache Spark?

- Fast and general cluster computing system, interoperable with Hadoop, included in all major distros

- Improves efficiency through:
  - In-memory computing primitives
  - General computation graphs
- Improves usability through:
  - Rich APIs in Scala, Java, Python
  - Interactive shell

► Up to 100× faster (2-10× on disk)

► 2-5× less code

Spark Model

- *Write programs in terms of transformations on distributed datasets*
- Resilient Distributed Datasets (RDDs)
  - Collections of objects that can be stored in memory or disk across a cluster
  - Parallel functional transformations (map, filter, …)
  - Automatically rebuilt on failure

On-Disk Sort Record:

Time to sort 100TB

2013 Record:
Hadoop

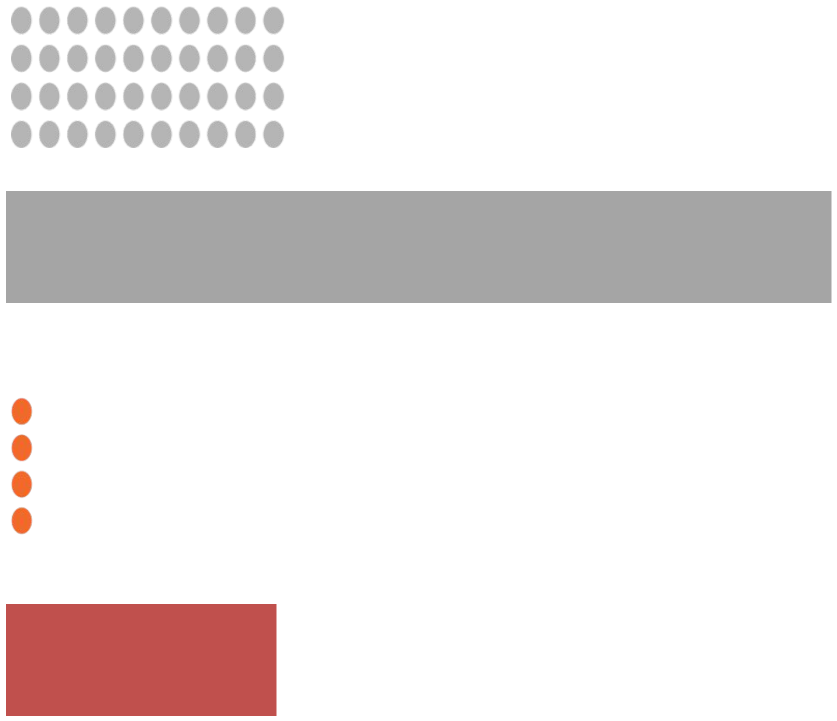
2100 machines

72 minutes

2014 Record:
Spark

207 machines

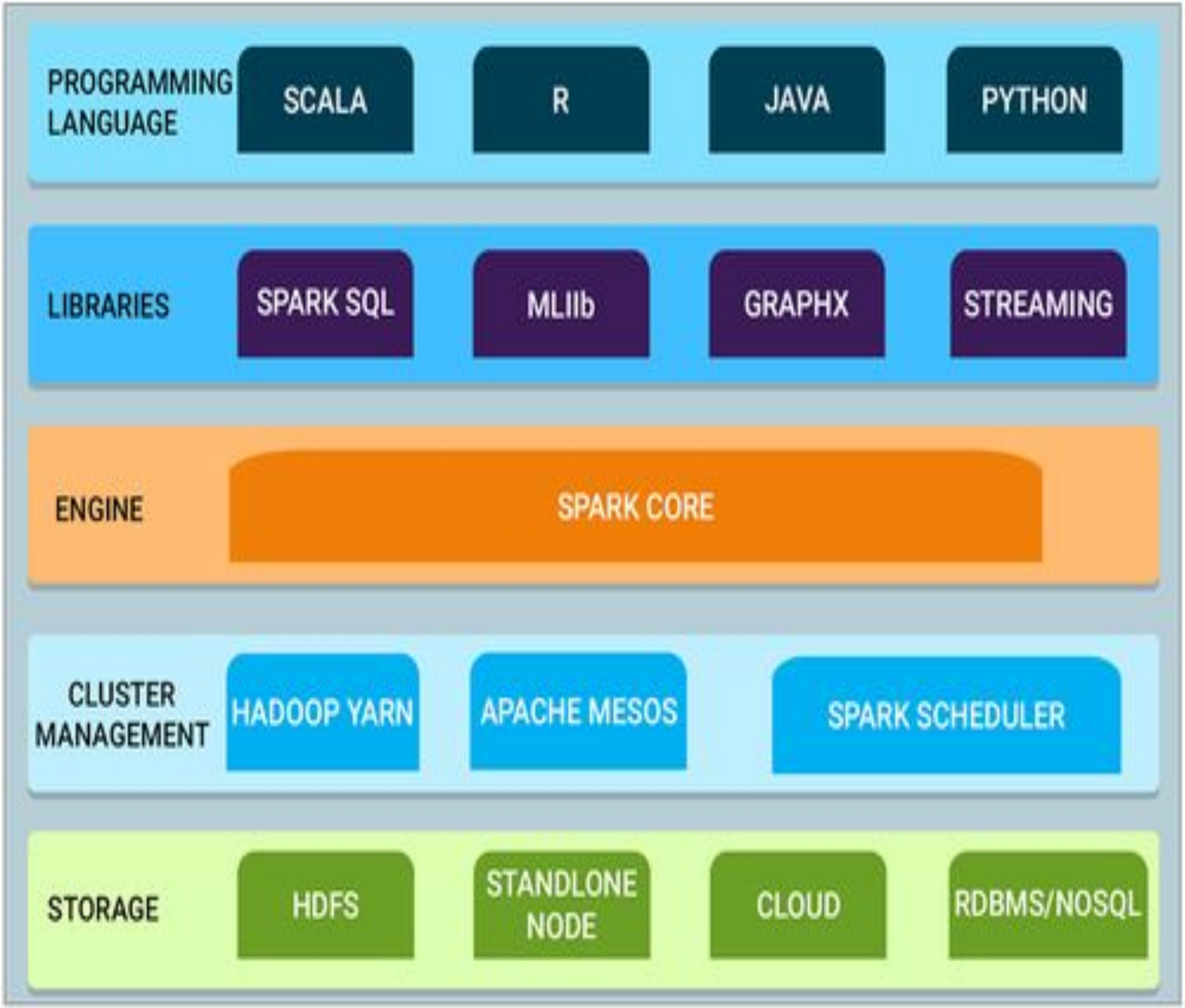
23 minutes



Also sorted 1PB in 4 hours

Source: Daytona GraySort benchmark, sortbenchmark.org

Components of Spark


PROGRAMMING LANGUAGE
SCALA
R
JAVA
PYTHON
LIBRARIES
SPARK SQL
MLIIb
GRAPHX
STREAMING
ENGINE
SPARK CORE
CLUSTER MANAGEMENT
HADOOP YARN
APACHE MESOS
SPARK SCHEDULER
STORAGE
HDFS
STANDLONE NODE
CLOUD
RDBMS/NOSQL

- **Spark Core** — Spark Core is the base engine for large-scale parallel and distributed data processing.
- additional libraries which are built on top of the core allow diverse workloads for streaming, SQL, and machine learning.
- It is responsible for memory management and fault recovery, scheduling, distributing and monitoring jobs on a cluster & interacting with storage systems.

- **Cluster management** — A cluster manager is used to acquire cluster resources for executing jobs.
- Spark core runs over diverse cluster managers including Hadoop YARN, Apache Mesos, Amazon EC2 and Spark’s built-in cluster manager.
- The cluster manager handles resource sharing between Spark applications.
- Spark can access data in HDFS, Cassandra, HBase, Hive, Alluxio, and any Hadoop data source

- **Spark Streaming** — Spark Streaming is the component of Spark which is used to process real-time streaming data.

- **Spark SQL**: Spark SQL is a new module in Spark which integrates relational processing with Spark’s functional programming API.
- It supports querying data either via SQL or via the Hive Query Language. The DataFrame and Dataset APIs of Spark SQL provide a higher level of abstraction for structured data.

- **GraphX**: GraphX is the Spark API for graphs and graph-parallel computation.

- used in machine learning, social network analysis, recommendation systems, and bioinformatics

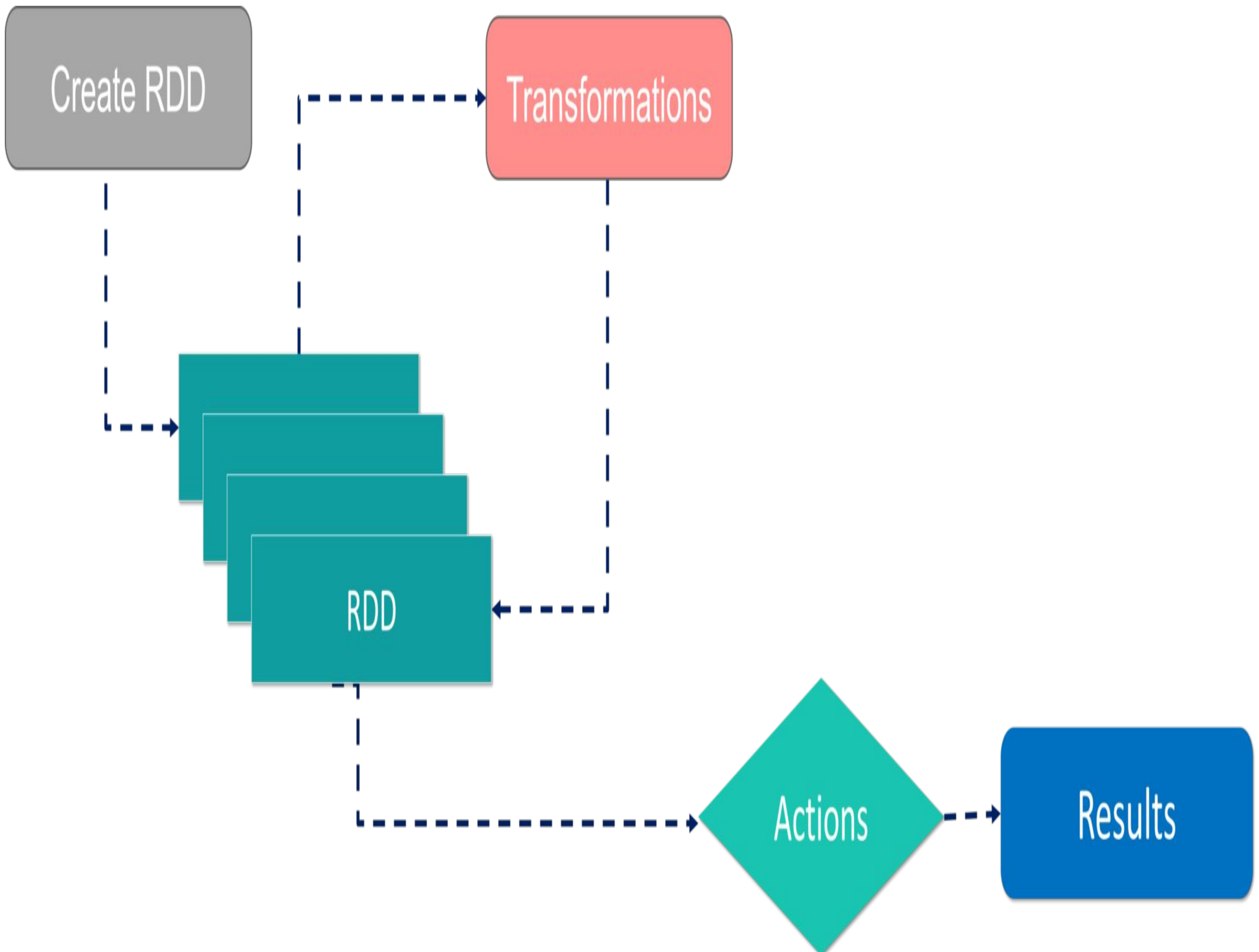
- **MLlib** (Machine Learning): MLlib stands for Machine Learning Library. Spark MLlib is used to perform machine learning in Apache Spark.

Resilient Distributed Datasets (RDD)

What is RDD?

- **Resilient Distributed Dataset(RDD)**
- RDDs are the building blocks of any Spark application. RDDs Stands for:
- ***Resilient:*** Fault tolerant and is capable of rebuilding data on failure
- ***Distributed:*** Distributed data among the multiple nodes in a cluster
- ***Dataset:*** Collection of partitioned data with values
- It is a layer of abstracted data over the distributed collection. It is immutable in nature and follows *lazy transformations*.

- Formally, an RDD is a read-only, partitioned collection of records. RDDs can be created through deterministic operations on either data on stable storage or other RDDs. RDD is a fault-tolerant collection of elements that can be operated on in parallel.
- In the spark distributed environment, each dataset in RDD is divided into logical partitions, which may be computed on different nodes of the cluster. Due to this, you can perform transformations or actions on the complete data parallelly.
-


Create RDD
Transformations
RDD
Actions
Results

Spark Operations = TRANSFORMATIONS + ACTIONS

- There are two ways to create RDDs − parallelizing an existing collection in your driver program, or by referencing a dataset in an external storage system, such as a shared file system, HDFS, HBase, etc.
- With RDDs, you can perform two types of operations:
- **Transformations:** They are the operations that are applied to create a new RDD.
- **Actions:** They are applied on an RDD to instruct Apache Spark to apply computation and pass the result back to the driver.

Essential Core & Intermediate Spark Operations

TRANSFORMATIONS

General

- map
- filter
- flatMap
- mapPartitions
- mapPartitionsWithIndex
- groupBy
- sortBy

Math / Statistical

- sample
- randomSplit

Set Theory / Relational

- union
- intersection
- subtract
- distinct
- cartesian
- zip

Data Structure / I/O

- keyBy
- zipWithIndex
- zipWithUniqueID
- zipPartitions
- coalesce
- repartition
- repartitionAndSortWithinPartitions
- pipe

ACTIONS

General

- reduce
- collect
- aggregate
- fold
- first
- take
- forEach
- top
- treeAggregate
- treeReduce
- forEachPartition
- collectAsMap

Math / Statistical

- count
- takeSample
- max
- min
- sum
- histogram
- mean
- variance
- stdev
- sampleVariance
- countApprox
- countApproxDistinct

Set Theory / Relational

- takeOrdered

Data Structure / I/O

- saveAsTextFile
- saveAsSequenceFile
- saveAsObjectFile
- saveAsHadoopDataset
- saveAsHadoopFile
- saveAsNewAPIHadoopDataset
- saveAsNewAPIHadoopFile

= easy   = medium

Essential Core & Intermediate PairRDD Operations

TRANSFORMATIONS

General

- flatMapValues
- groupByKey
- reduceByKey
- reduceByKeyLocally
- foldByKey
- aggregateByKey
- sortByKey
- combineByKey

Math / Statistical

- sampleByKey

Set Theory / Relational

- cogroup (=groupWith)
- join
- subtractByKey
- fullOuterJoin
- leftOuterJoin
- rightOuterJoin

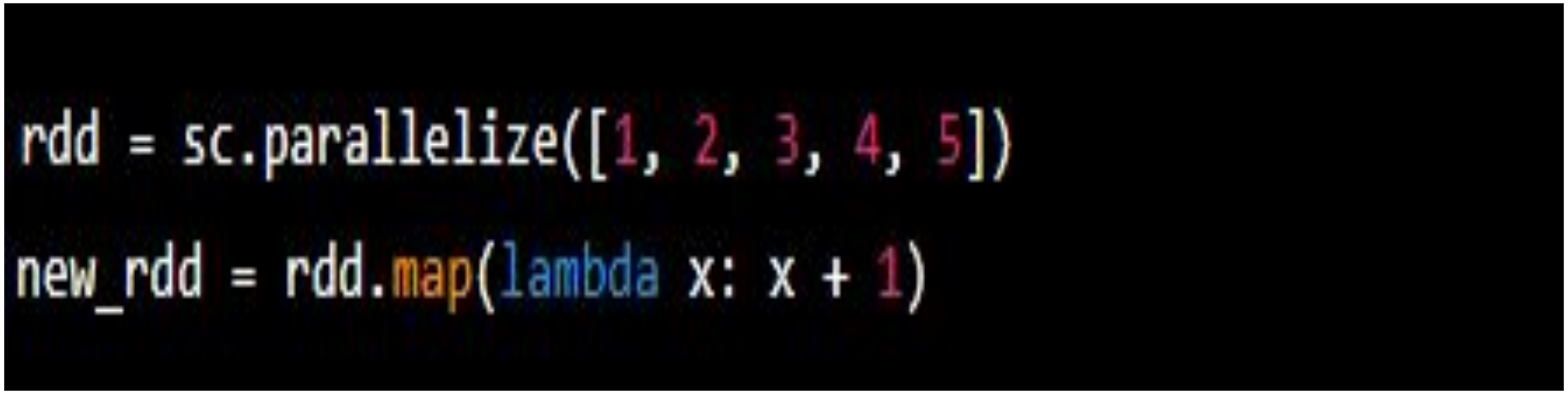
Data Structure

- partitionBy

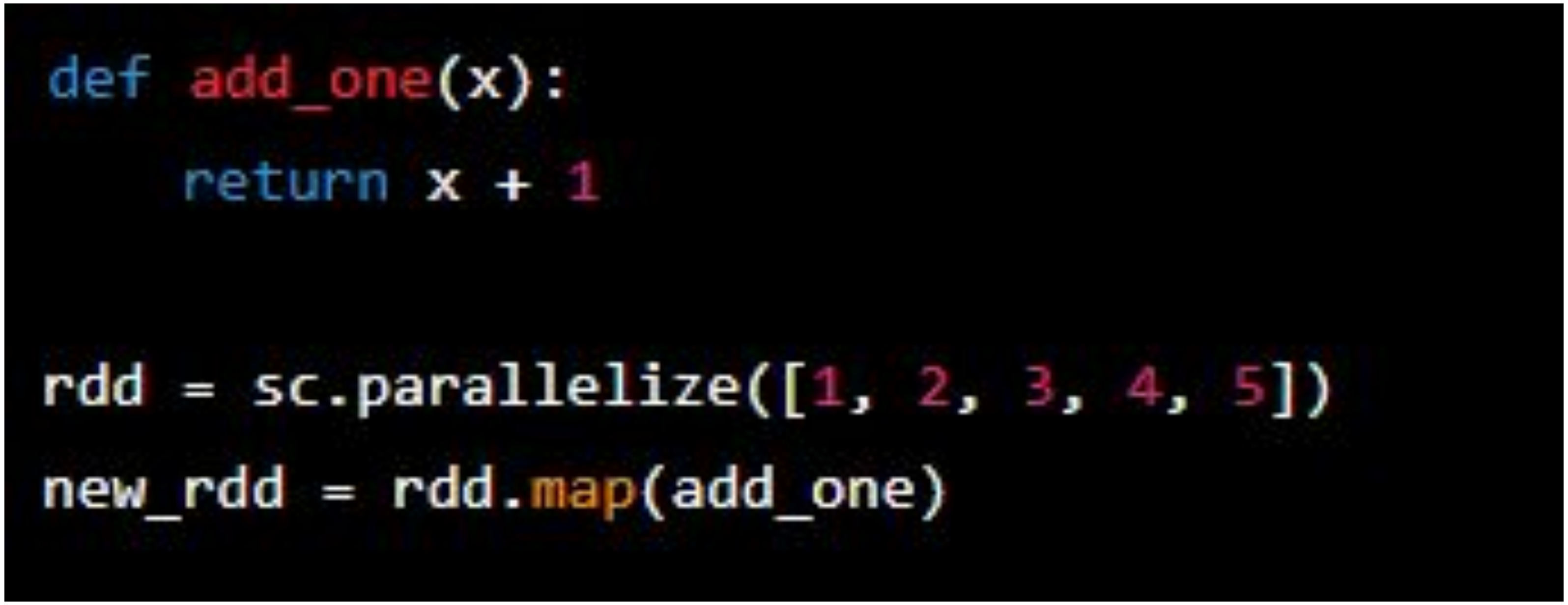
ACTIONS

- keys
- values

- countByKey
- countByValue
- countByValueApprox
- countApproxDistinctByKey
- countApproxDistinctByKey
- countByKeyApprox
- sampleByKeyExact

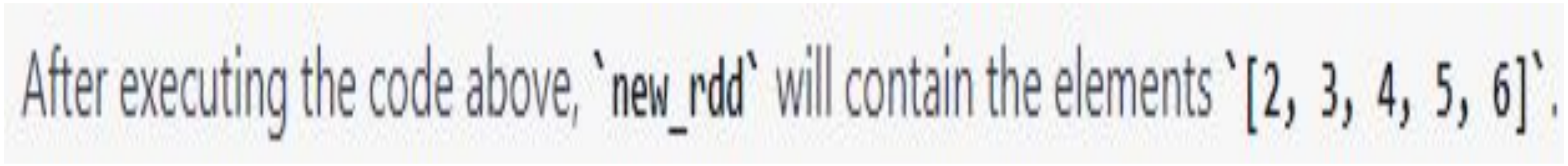


```python
rdd = sc.parallelize([1, 2, 3, 4, 5])
new_rdd = rdd.map(lambda x: x + 1)
```

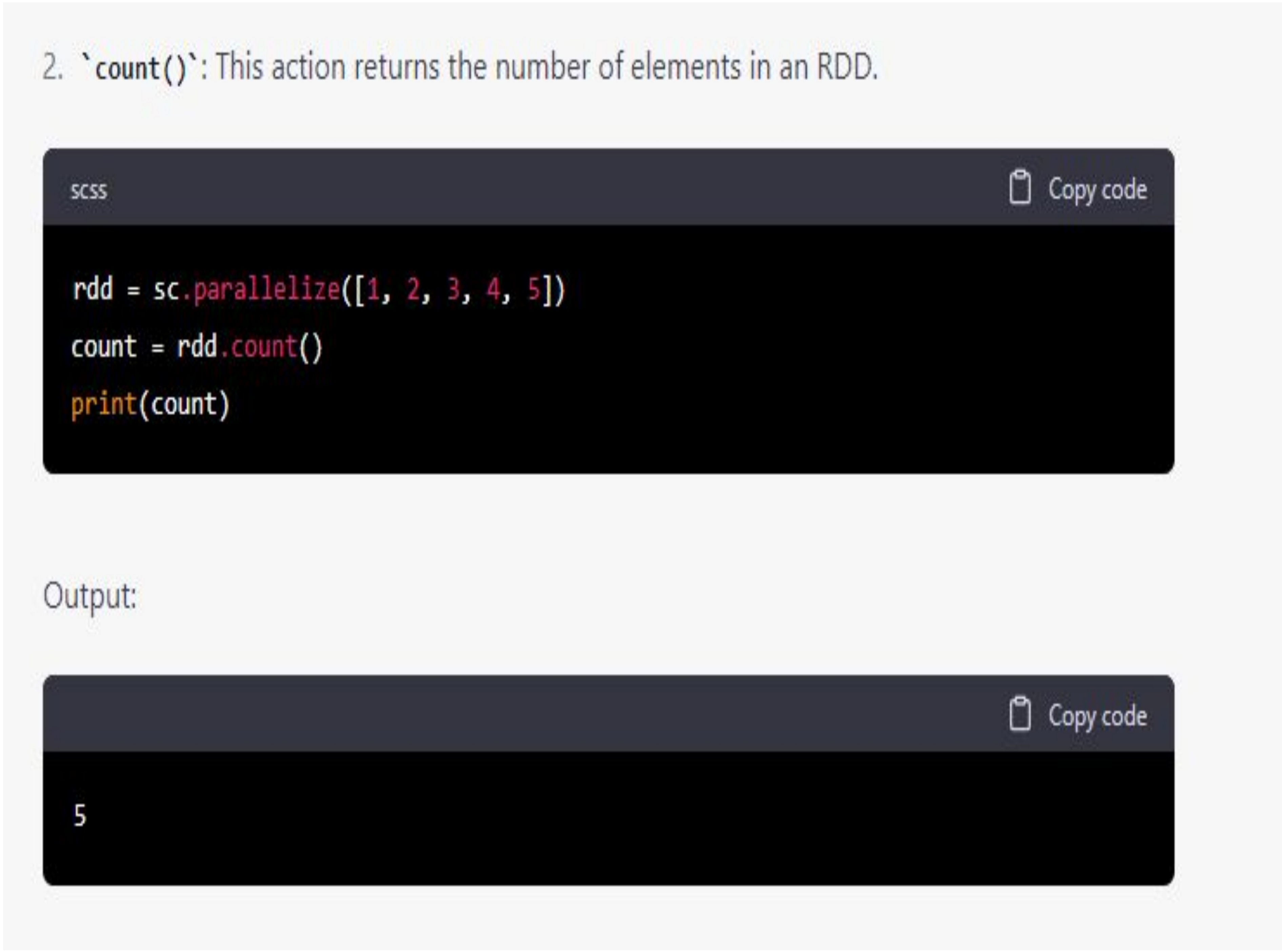


```python
def add_one(x):
    return x + 1

rdd = sc.parallelize([1, 2, 3, 4, 5])
new_rdd = rdd.map(add_one)
```



After executing the code above, `new_rdd` will contain the elements `[2, 3, 4, 5, 6]`.



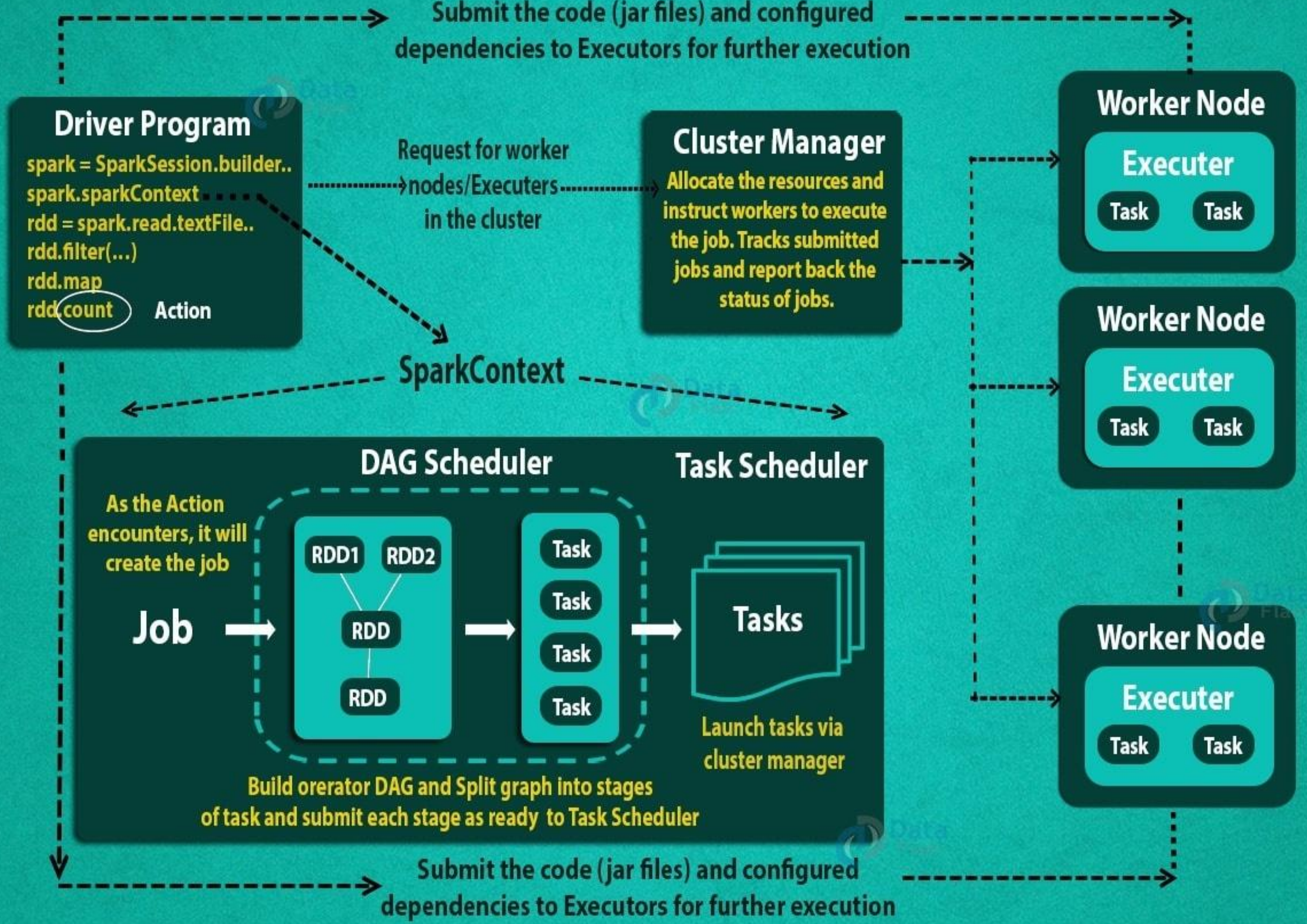
2. `count()`: This action returns the number of elements in an RDD.

```scss
rdd = sc.parallelize([1, 2, 3, 4, 5])
count = rdd.count()
print(count)
```

Output:

```
5
```

Internals of Job Execution In Spark



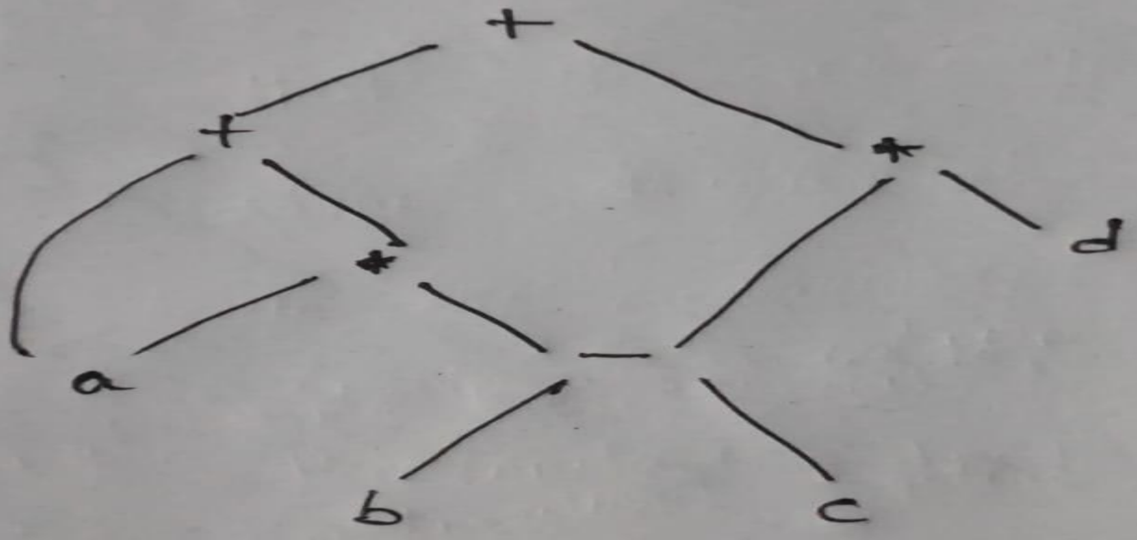
Submit the code (jar files) and configured dependencies to Executors for further execution
Driver Program
spark = SparkSession.builder..
spark.sparkContext
rdd = spark.read.textFile..
rdd.filter(...)
rdd.map
rdd.count
Action
Request for worker nodes/Executers in the cluster
Cluster Manager
Allocate the resources and instruct workers to execute the job. Tracks submitted jobs and report back the status of jobs.
Worker Node
Executer
Task
Task
Worker Node
Executer
Task
Task
Worker Node
Executer
Task
Task
SparkContext
DAG Scheduler
Task Scheduler
As the Action encounters, it will create the job
Job
RDD1
RDD2
RDD
RDD
Task
Task
Task
Task
Tasks
Launch tasks via cluster manager
Build orerator DAG and Split graph into stages of task and submit each stage as ready to Task Scheduler
Submit the code (jar files) and configured dependencies to Executors for further execution

Flow of execution

- **STEP 1:** The client submits spark user application code. When an application code is submitted, the driver implicitly converts user code that contains transformations and actions into a logically *directed acyclic graph* called ***DAG.*** At this stage, it also performs optimizations such as pipelining transformations.

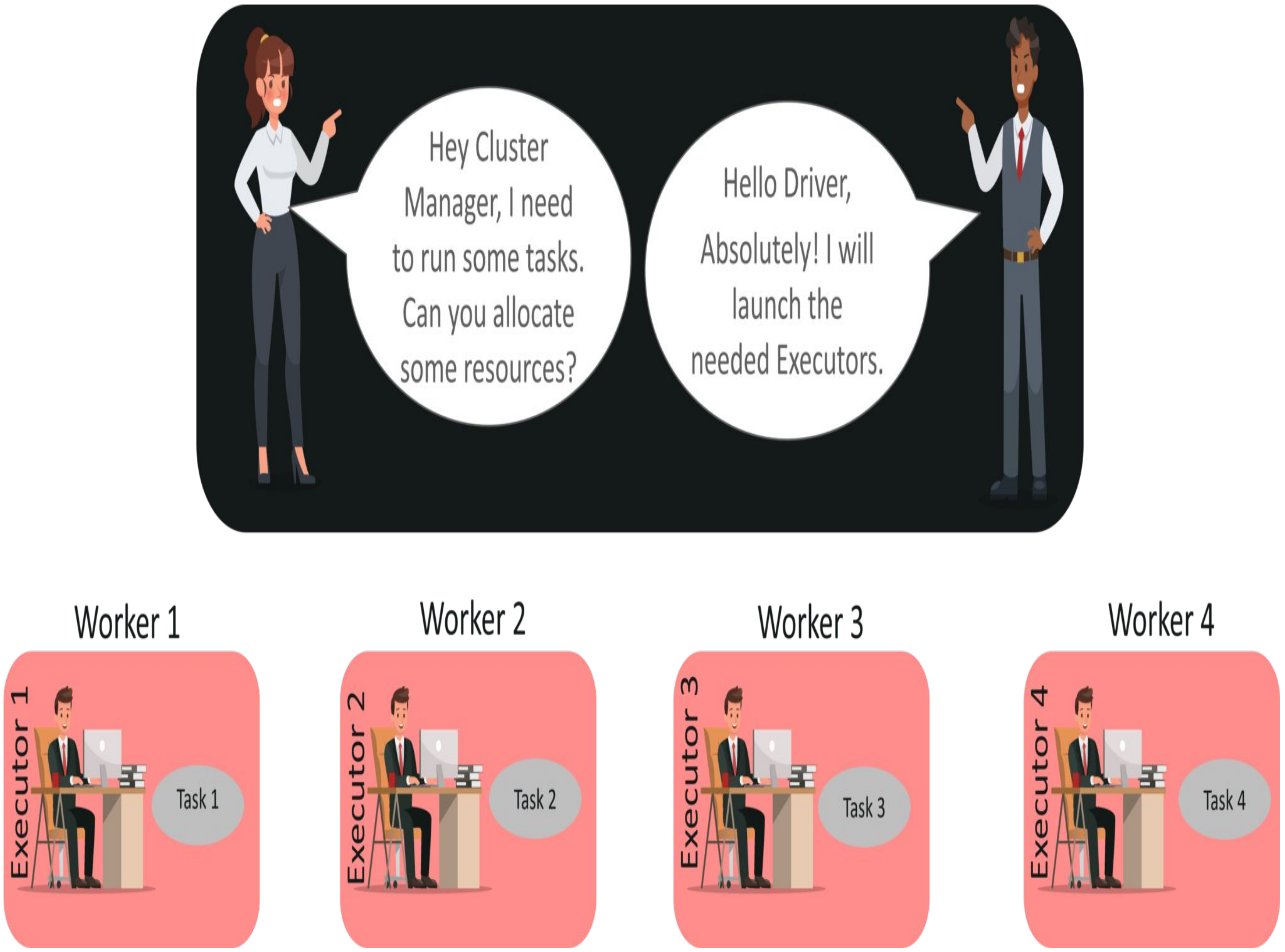
DAG for the expression

$a + a * (b - c) + (b - c) * d$

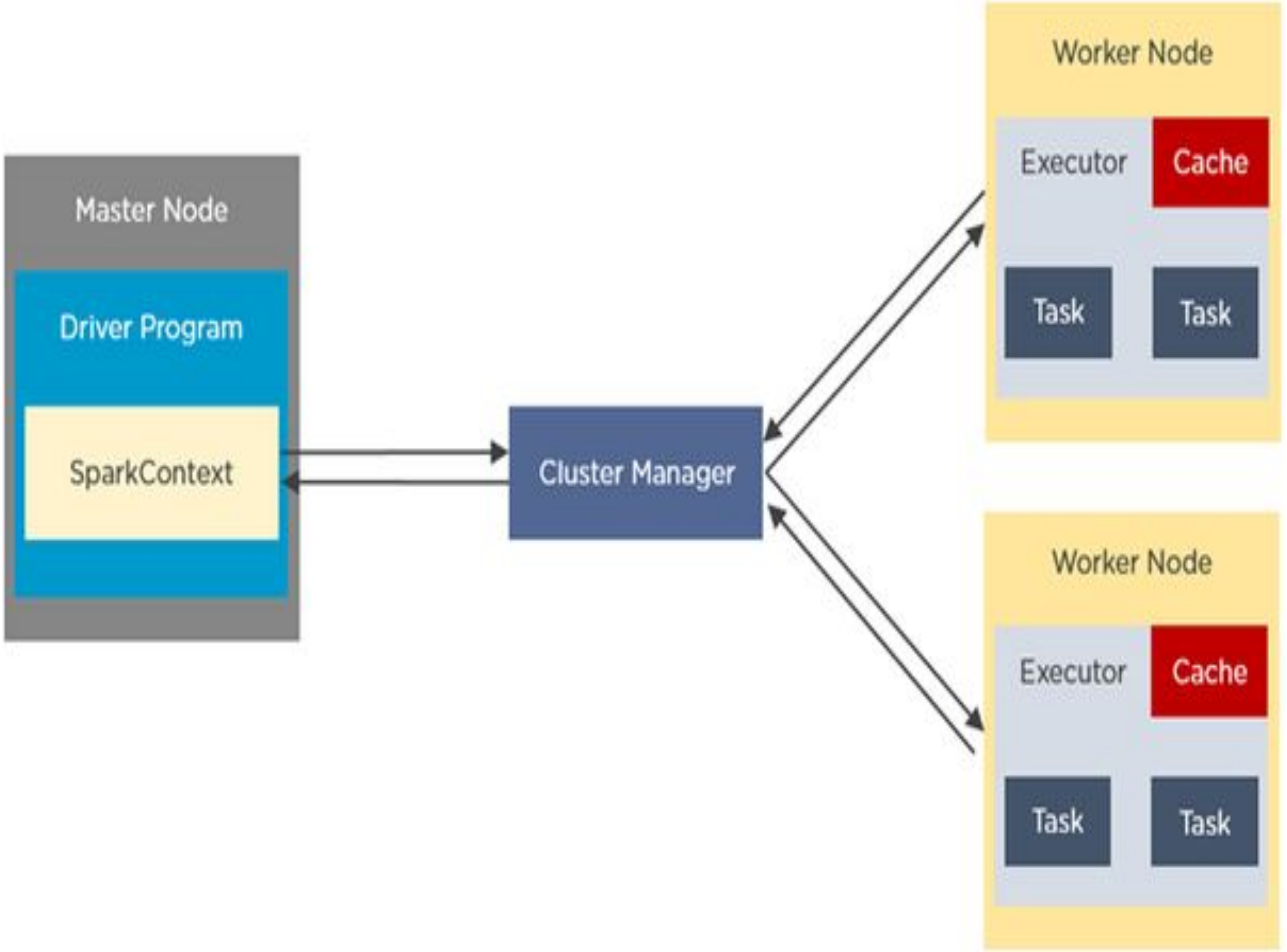


- **STEP 2:** After that, it converts the logical graph called DAG into physical execution plan with many stages. After converting into a physical execution plan, it creates physical execution units called tasks under each stage. Then the tasks are bundled and sent to the cluster.

- **STEP 3:** Now the driver talks to the cluster manager and negotiates the resources. Cluster manager launches executors in worker nodes on behalf of the driver. At this point, the driver will send the tasks to the executors based on data placement. When executors start, they register themselves with drivers. So, the driver will have a complete view of executors that are executing the task.


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

- **STEP 4:** During the course of execution of tasks, driver program will monitor the set of executors that runs. Driver node also schedules future tasks based on data placement


Master Node
Driver Program
SparkContext
Cluster Manager
Worker Node
Executor
Cache
Task
Task
Worker Node
Executor
Cache
Task
Task

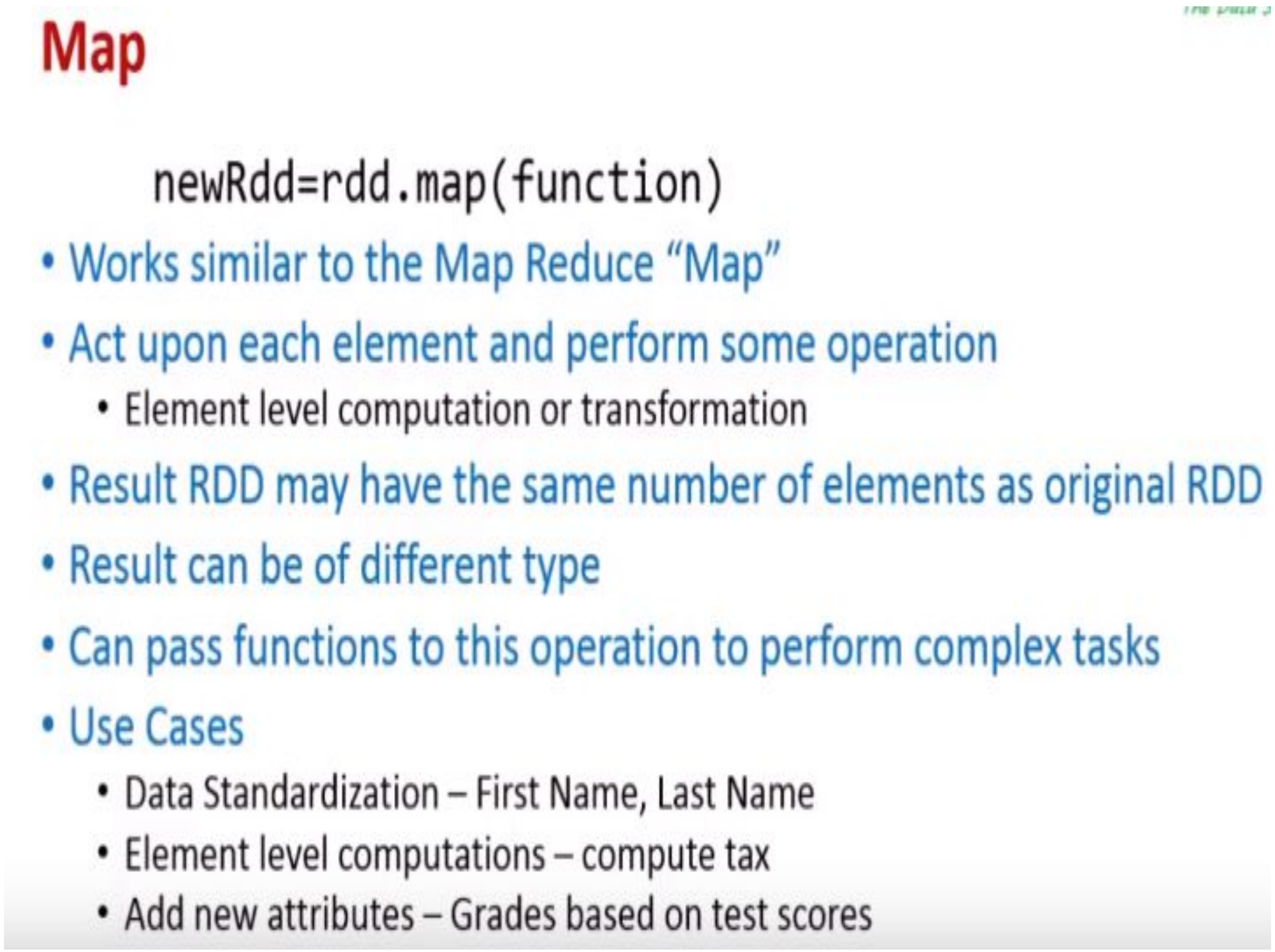
- In your **master node**, you have the *driver program*, which drives your application. The code you are writing behaves as a driver program or if you are using the interactive shell, the shell acts as the driver program.
- Inside the driver program, the first thing you do is, you *create* a ***Spark Context.*** Assume that the Spark context is a gateway to all the Spark functionalities. It is similar to your database connection. Any command you execute in your database goes through the database connection. Likewise, anything you do on Spark goes through Spark context.

- Now, this Spark context works with the ***cluster manager*** to manage various jobs. The driver program & Spark context takes care of the job execution within the cluster. A job is split into multiple tasks which are distributed over the worker node. Anytime an RDD is created in Spark context, it can be distributed across various nodes and can be cached there.
- **W*orker nodes*** are the slave nodes whose job is to basically execute the tasks. These tasks are then executed on the partitioned RDDs in the worker node and hence returns back the result to the Spark Context.

- Spark Context takes the job, breaks the job in tasks and distribute them to the worker nodes. These tasks work on the partitioned RDD, perform operations, collect the results and return to the main Spark Context.
- If you increase the number of workers, then you can divide jobs into more partitions and execute them parallel over multiple systems. It will be a lot faster.
- With the increase in the number of workers, memory size will also increase & you can cache the jobs to execute it faster.

executor

- The individual task in the given Spark job runs in the Spark executors. Executors are launched once in the beginning of Spark Application and then they run for the entire lifetime of an application. Even if the Spark executor fails, the Spark application can continue with ease. There are two main roles of the executors:
- Runs the task that makes up the application and returns the result to the driver.
- Provide **in-memory** storage for RDDs that are cached by the user.



Map

```
newRdd=rdd.map(function)
```

- Works similar to the Map Reduce “Map”
- Act upon each element and perform some operation
  - Element level computation or transformation
- Result RDD may have the same number of elements as original RDD
- Result can be of different type
- Can pass functions to this operation to perform complex tasks
- Use Cases
  - Data Standardization – First Name, Last Name
  - Element level computations – compute tax
  - Add new attributes – Grades based on test scores

flatMap

```
newRdd=rdd.flatMap(function)
```

- Works the same way as map
- Can return more elements than the original map

Pair RDDs

- Pair RDDs are a special type of RDDs that can store key value pairs.
- All transformations for regular RDDs available for Pair RDDs
- Spark supports a set of special functions to handle Pair RDD operations
    - mapValues : transform each value without changing the key
    - flatMapValues : generate multiple values with the same key

Introduction to actions

- Act on a RDD and product a result (not a RDD)
- Lazy evaluation – Spark does not act until it sees an action
- Simple actions
  - `collect` – return all elements in the RDD as an array. Use to trigger execution or print values
  - `count` – count the number of elements in the RDD
  - `first` – returns the first element in the RDD
  - `take(n)` – returns the first n elements

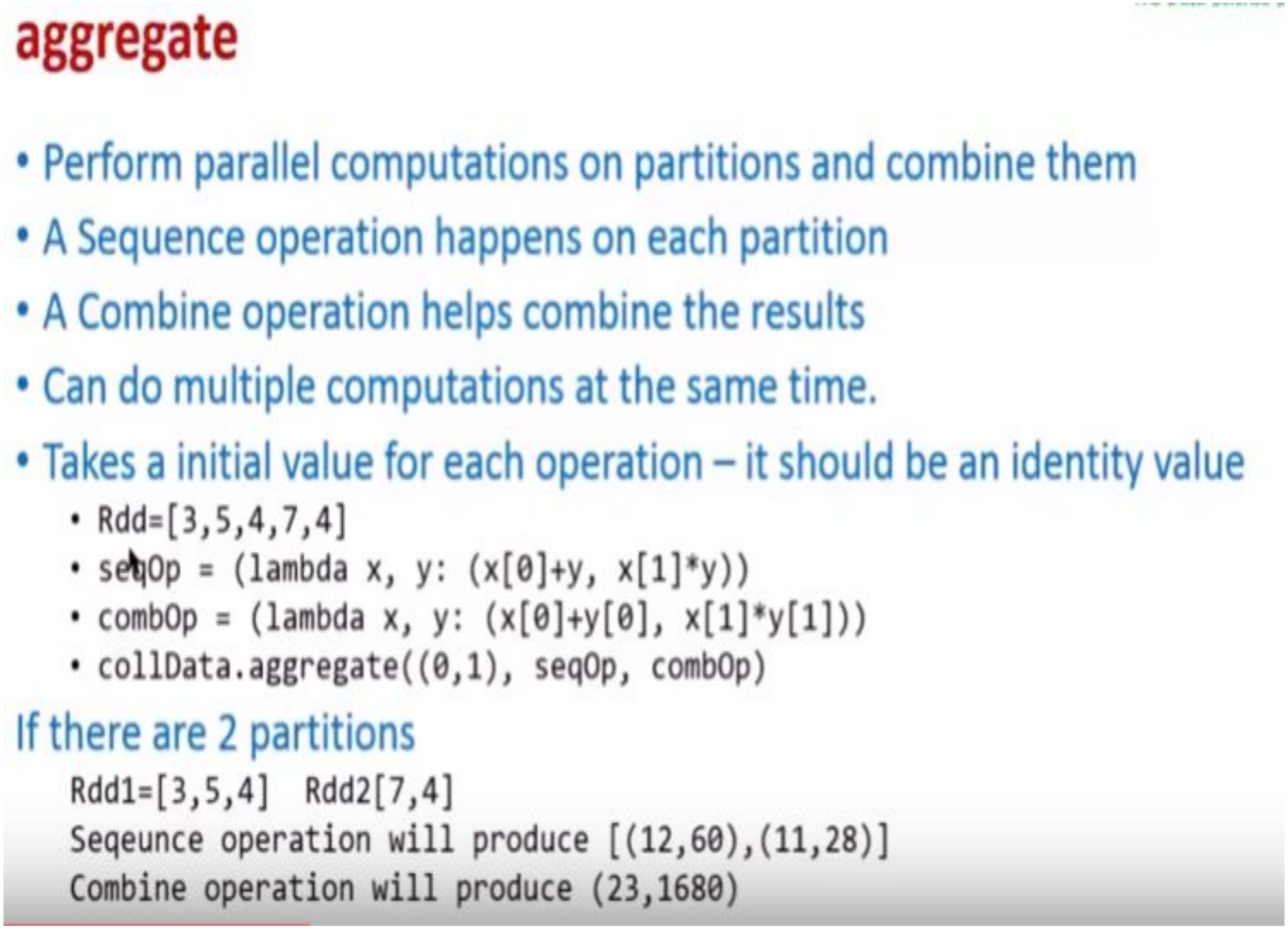
reduce

- Perform an operation across all elements of an RDD
  - sum, count etc.
- The operation is a function that takes as input two values.
- The function is called for every element in the RDD

```
inputRDD = [ a, b, c, d, e ]   and the function is  func(x,y)
func( func( func( func(a,b), c), d), e)
```

- Example

```
vals = [3,5,2,4,1]
sum(x,y) { return x + y }
sum( sum( sum( sum(3,5), 2), 4), 1) = 15
```



aggregate

- Perform parallel computations on partitions and combine them
- A Sequence operation happens on each partition
- A Combine operation helps combine the results
- Can do multiple computations at the same time.
- Takes a initial value for each operation – it should be an identity value
  - `Rdd=[3,5,4,7,4]`
  - `seqOp = (lambda x, y: (x[0]+y, x[1]*y))`
  - `combOp = (lambda x, y: (x[0]+y[0], x[1]*y[1]))`
  - `collData.aggregate((0,1), seqOp, combOp)`

If there are 2 partitions

```
Rdd1=[3,5,4]  Rdd2[7,4]
Seqeunce operation will produce [(12,60),(11,28)]
Combine operation will produce (23,1680)
```

Persistence

- By default, Spark loads an RDD whenever it required. It drops it once the action is over
  - It will load and re-compute the RDD chain, each time a different operation is performed
- Persistence allows the intermediate RDD to be persisted so it need not have to be recomputed.
- persist() can persist the RDD in memory, disk, shared or in other third party sinks