SPARK

What is SPARK

- Apache Spark is an *open-source cluster computing framework* for *real-time processing.*
- It has a thriving open-source community and is the most active Apache project at the moment.
- Spark provides an interface for programming entire clusters with implicit data parallelism and fault-tolerance.
- It was built on top of Hadoop MapReduce and it extends the MapReduce model to efficiently use more types of computations.

Figure: *Real Time Processing In Spark*

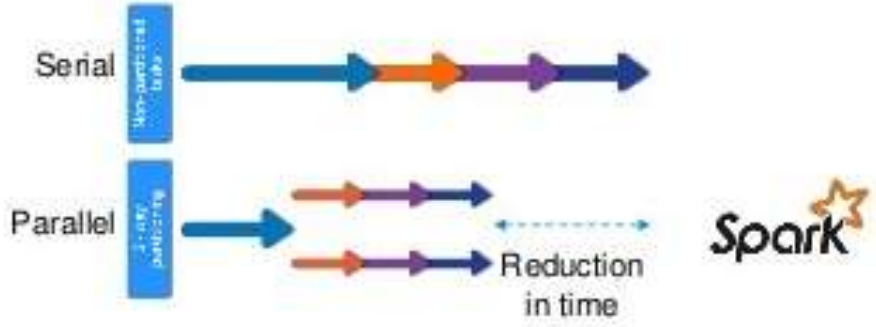


Figure: *Data Parallelism In Spark*

Why SPARK

Speed

- Enables applications in Hadop clusters to run upto:
  - 100x faster in memory
  - 10X on disks

Ease of Use

- Allows quickly write applications in Java, SCALA or Python

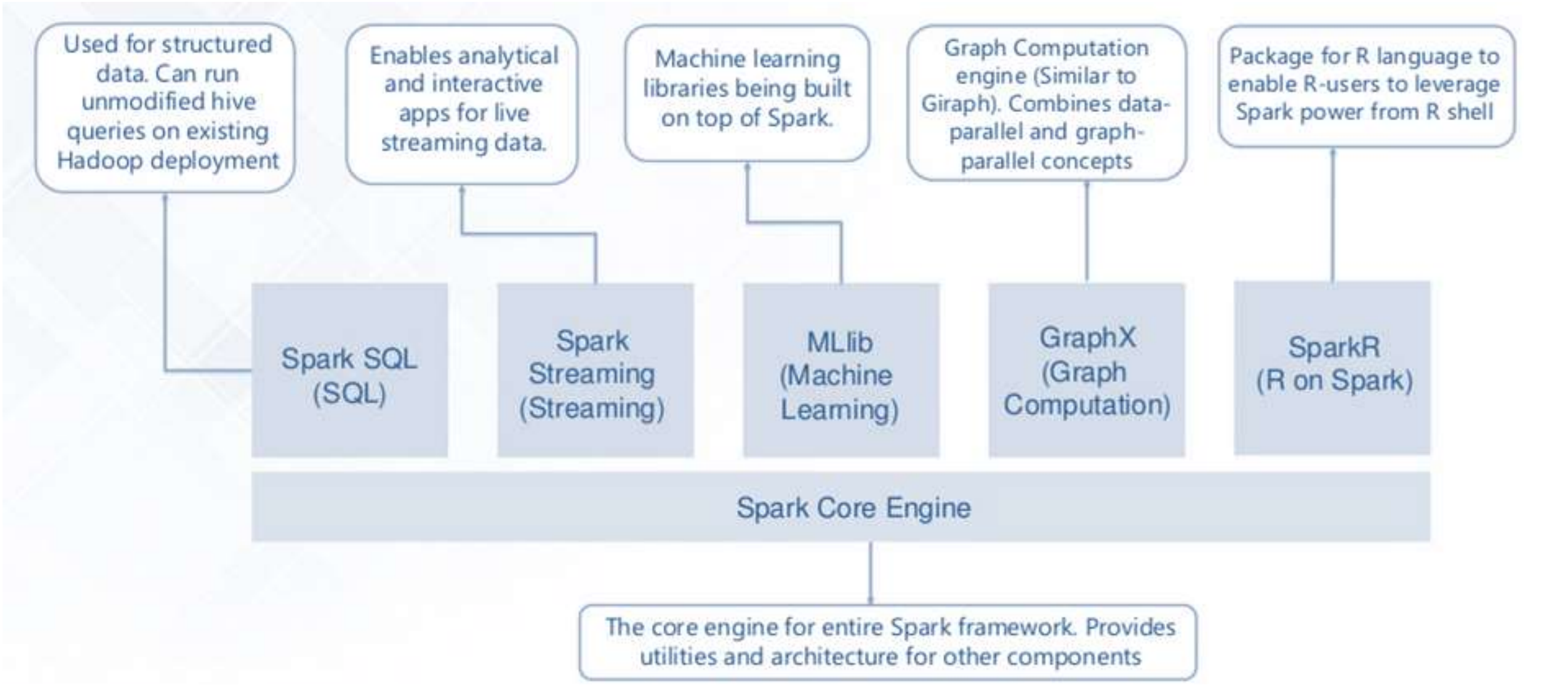
Sophisticated Analytics

- Supports SQL queries, streaming data and complex analytics

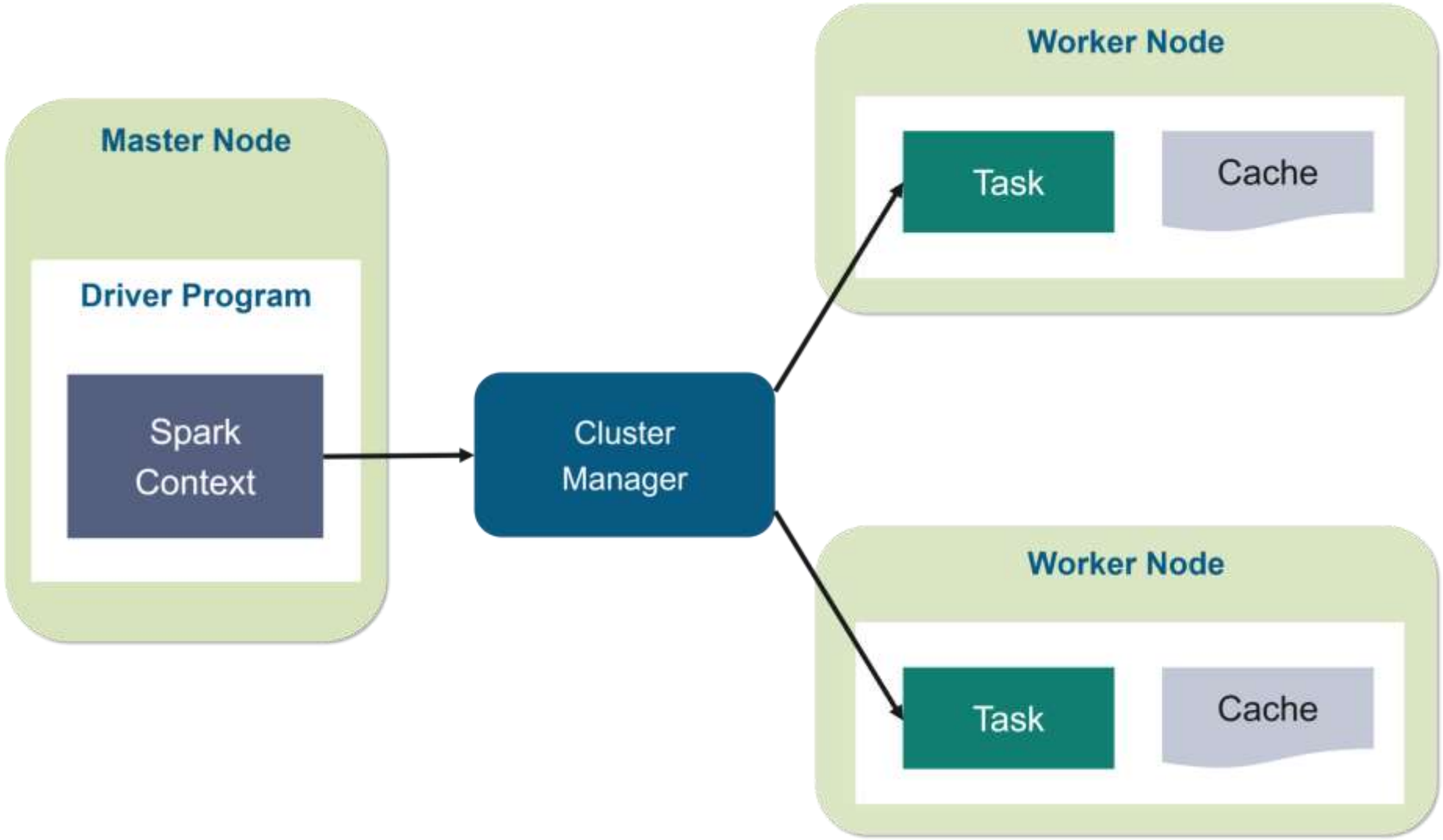
Spark Vs Map Reduce

MapReduce	Spark
MapReduce is fast but less speed as every transaction requires accessing the disk	Spark is 100X more faster than MR as every transaction doesn't require accessing the disk
Batch processing	Real-time processing
Stores data on disk (ROM)	Stores data on memory(RAM)
Written in JAVA	Written in Scala
Not much useful for Machine Learning algorithms	Useful for ML ,streaming etc

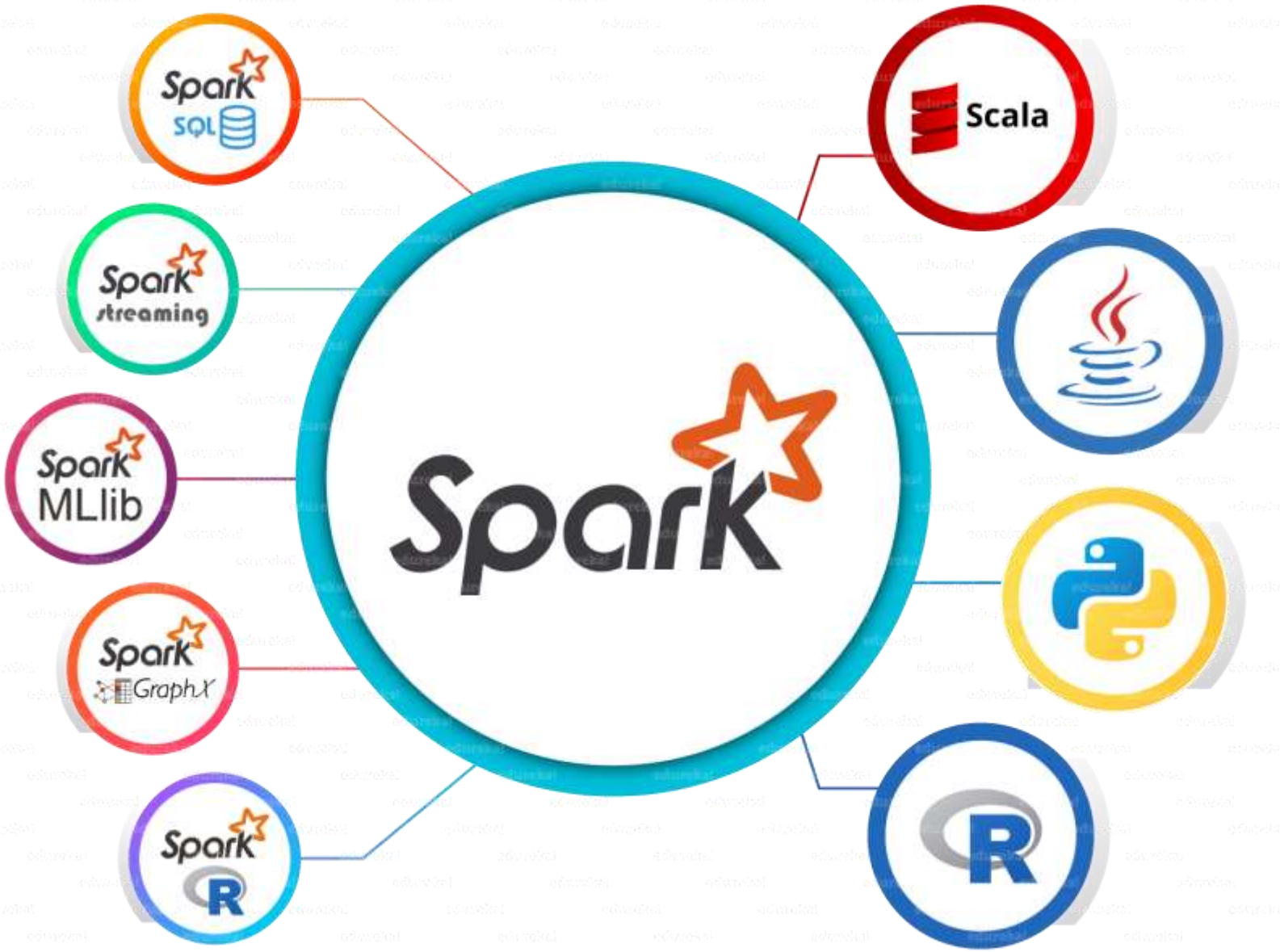
SPARK Components



SPARK Architecture

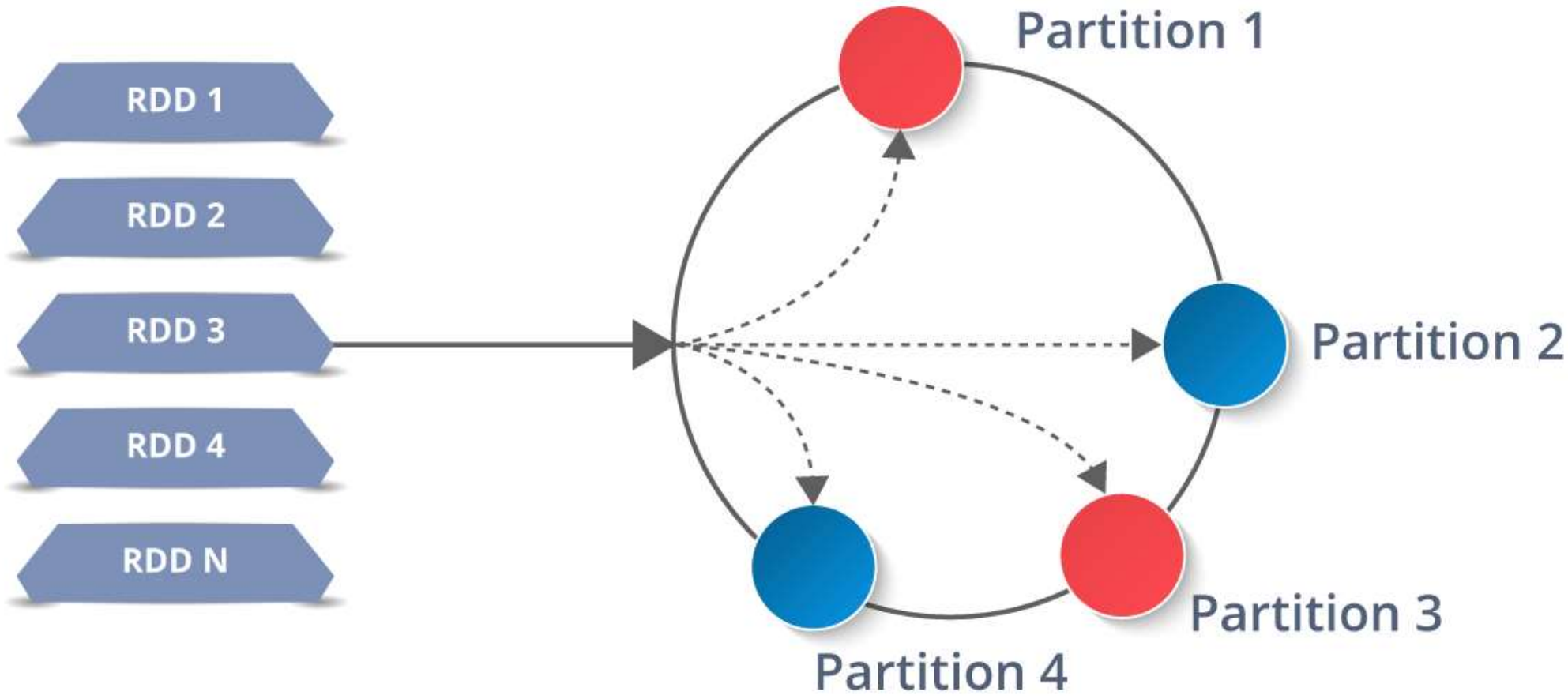


Spark Ecosystem



SPARK RDD

Resilient Distributed Dataset(RDD)



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

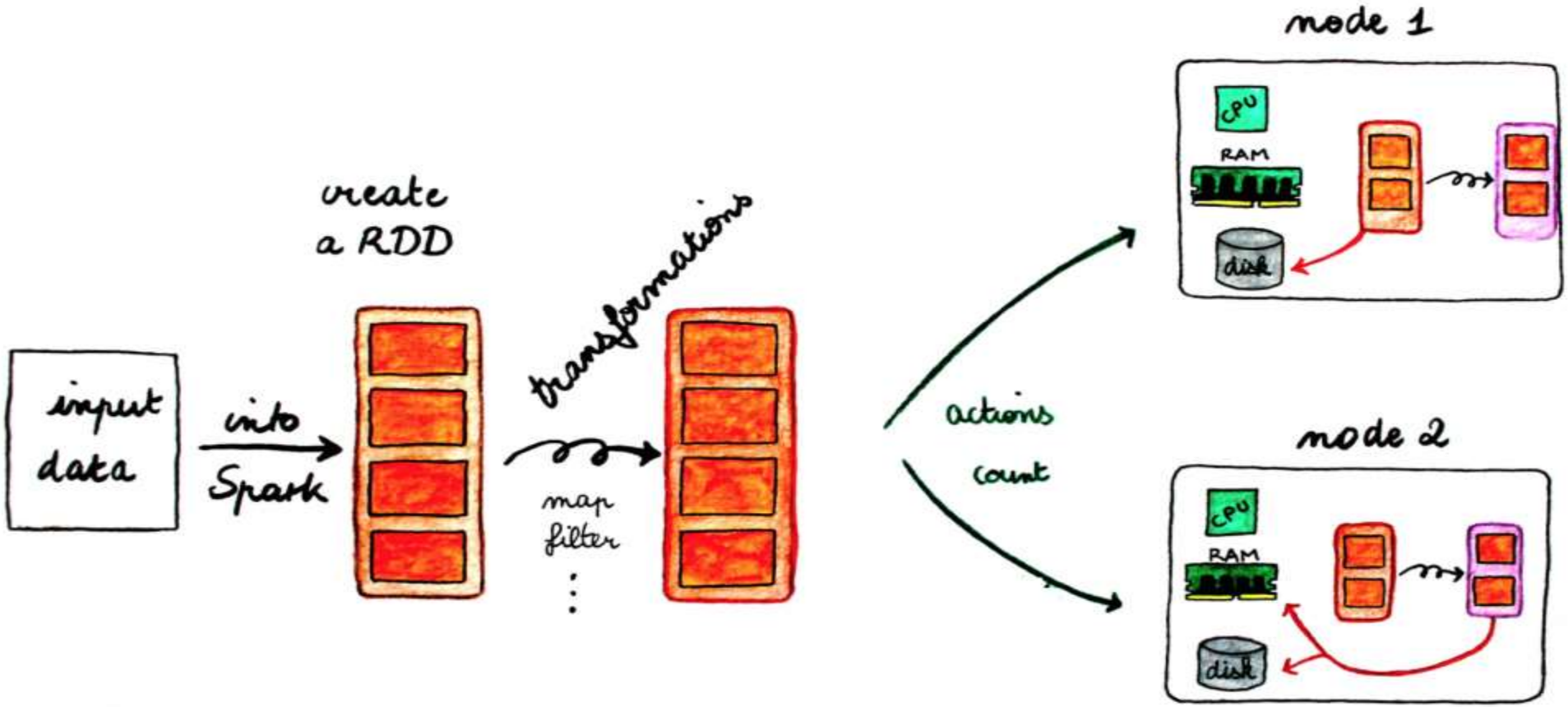
Resilient Distributed Dataset(RDD)

- It is a layer of abstracted data over the distributed collection.
- It is immutable in nature and follows *lazy transformations*.
- RDD is split into chunks based on a key.
- RDDs are highly resilient, same data chunks are replicated across multiple executor nodes.
- Moreover, once you create an RDD it becomes ***immutable***.
- RDD Object whose state cannot be modified after it is created, but they can surely be transformed.

Resilient Distributed Dataset(RDD) Operations

RDD in Apache Spark supports two types of operations:

- **Transformation** creates another RDD and it is lazy operation (for example: map, flatmap, filter, groupBy…)
- **Action** returns a value after running a computation (for example: count, first, take, collect…)



Resilient Distributed Dataset(RDD) Operations

RDD Transformation

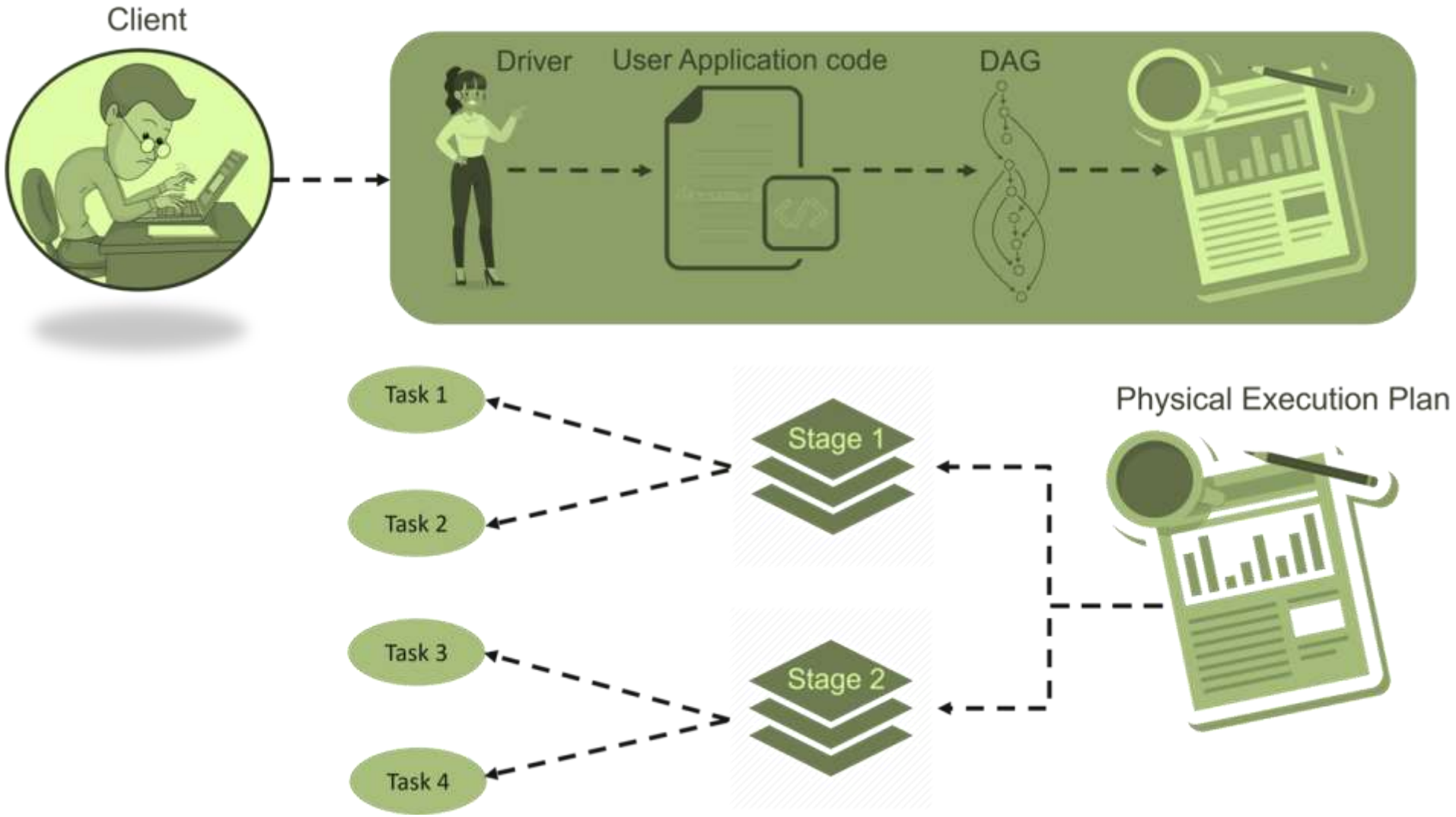
- Spark **RDD Transformations** are *functions* that take an RDD as the input and produce one or many RDDs as the output.
- They do not change the input RDD but always produce one or more new RDDs by applying the computations they represent e.g. Map(), filter(), reduceByKey() etc.
- Transformations are **lazy** operations on an RDD in Apache Spark.
- Creates new RDDs, which executes when an Action occurs.
- Hence, Transformation creates a new dataset from an existing one.

Resilient Distributed Dataset(RDD) Operations

RDD Action

- An **Action** in Spark returns final result of RDD computations.
- It triggers execution using lineage graph to load the data into original RDD, carry out all intermediate transformations and return final results to Driver program or write it out to file system.
- Lineage graph is dependency graph of all parallel RDDs of RDD.
- An Action is one of the ways to send result from executors to the driver. First(), take(), reduce(), collect(), the count() is some of the Actions in spark.
- When the Action occurs it does not create the new RDD, unlike transformation.
- Thus, actions are RDD operations that give no RDD values.
- Action stores its value either to drivers or to the external storage system. It brings laziness of RDD into motion.

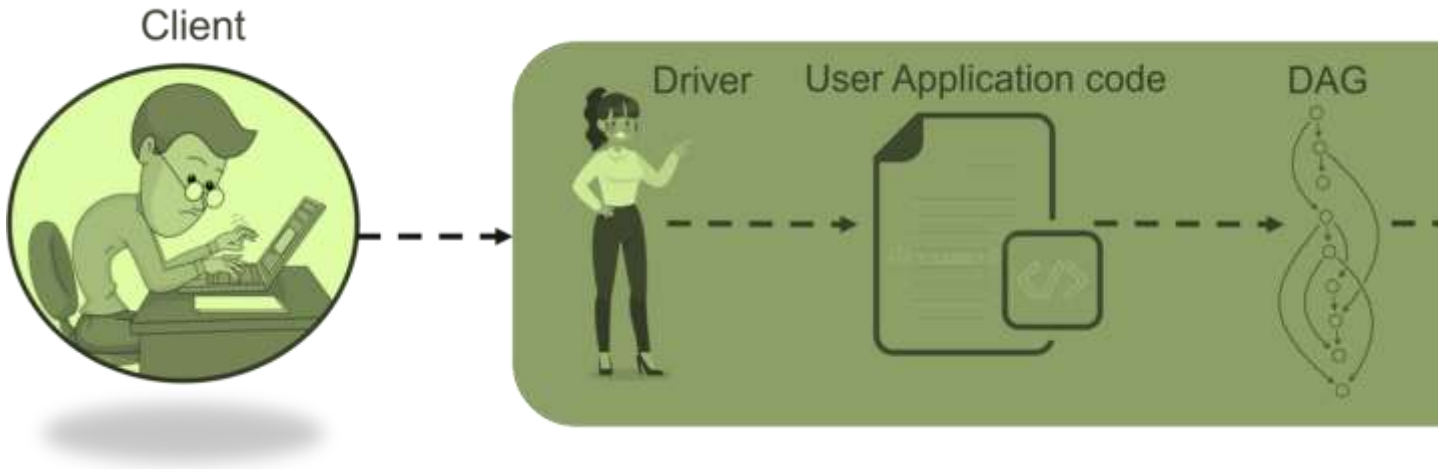
Workflow of Spark Architecture



Workflow of Spark Architecture

STEP 1: The client submits spark user application code.

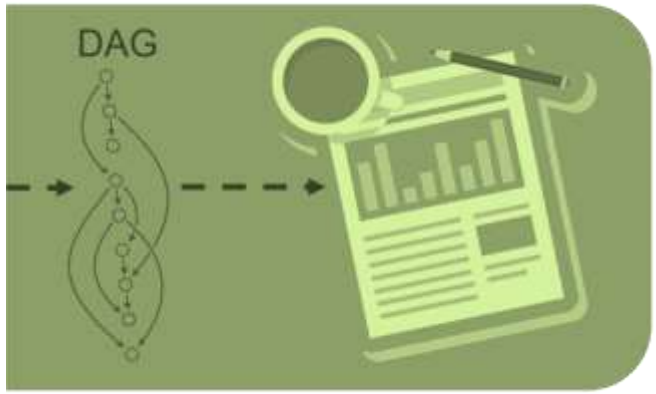
When an application code is submitted, the driver implicitly converts user code that contains transformations and actions into a logically *directed acyclic graph* called ***DAG.*** At this stage, it also performs optimizations such as pipelining transformations.



Workflow of Spark Architecture

STEP 2: After that, it converts the logical graph called DAG into physical execution plan with many stages.

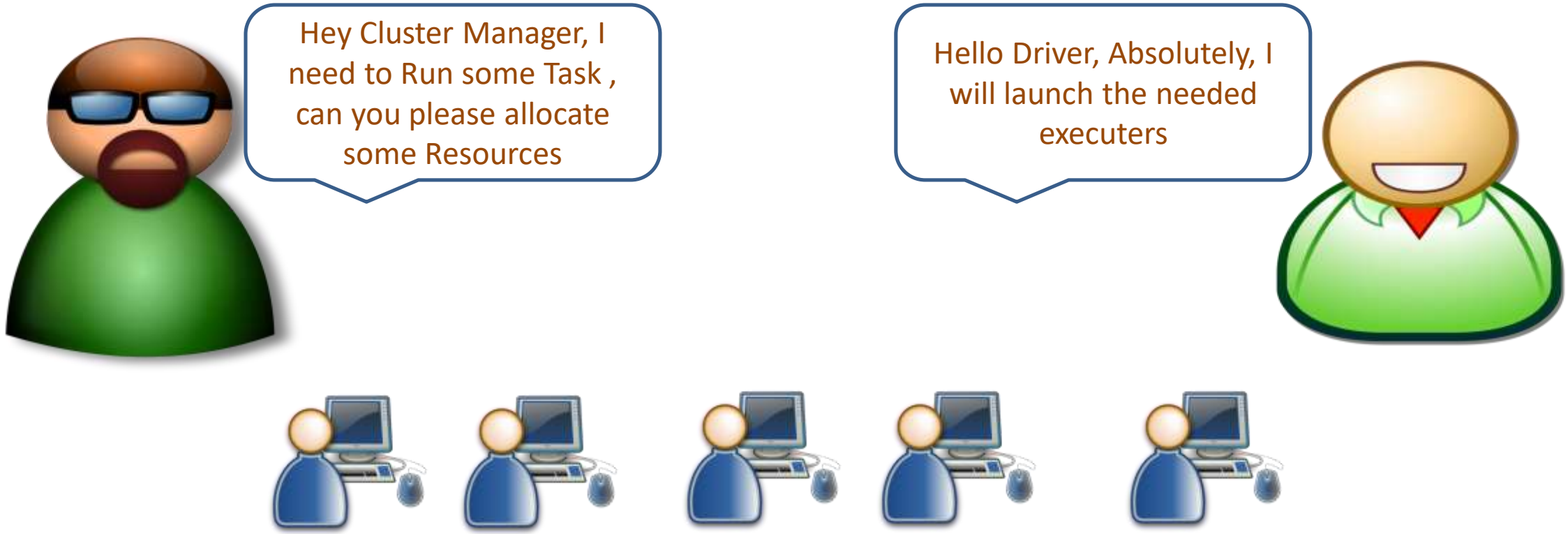
After converting into a physical execution plan, it creates physical execution units called tasks under each stage. Then the tasks are bundled and sent to the cluster.



Workflow of Spark Architecture

STEP 3: Now the driver talks to the cluster manager and negotiates the resources.

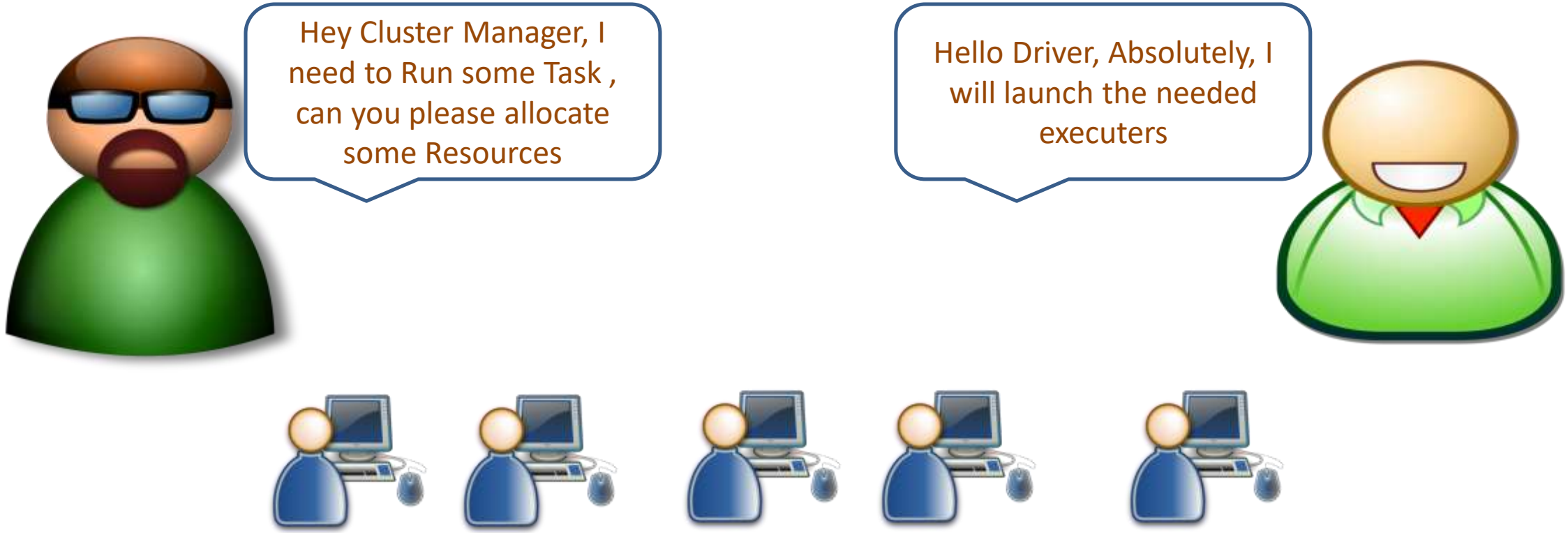
- Cluster manager launches executors in worker nodes on behalf of the driver.
- At this point, the driver will send the tasks to the executors based on data placement.
- When executors start, they register themselves with drivers. So, the driver will have a complete view of executors that are executing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



Workflow of Spark Architecture

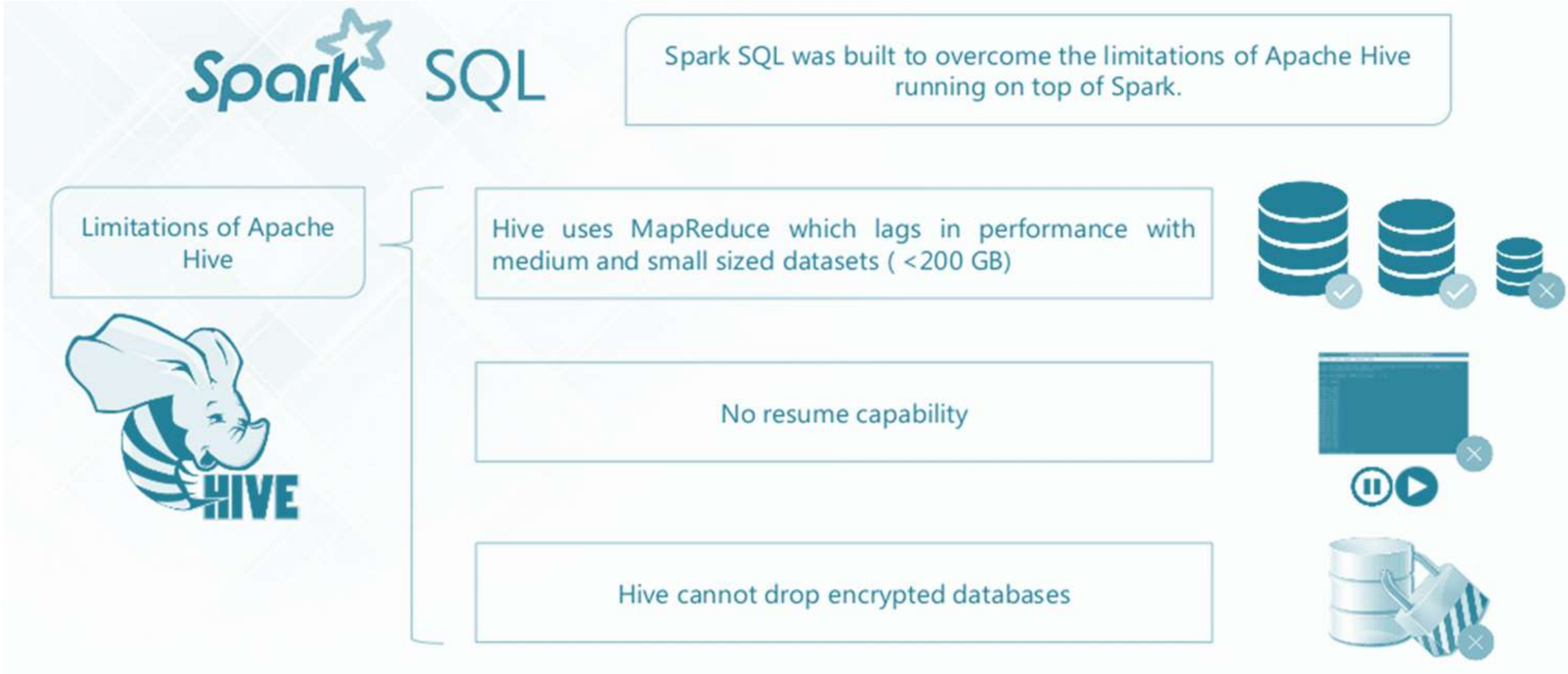
STEP 4: During the course of execution of tasks, driver program will monitor the set of executors that runs.

Driver node also schedules future tasks based on data placement.

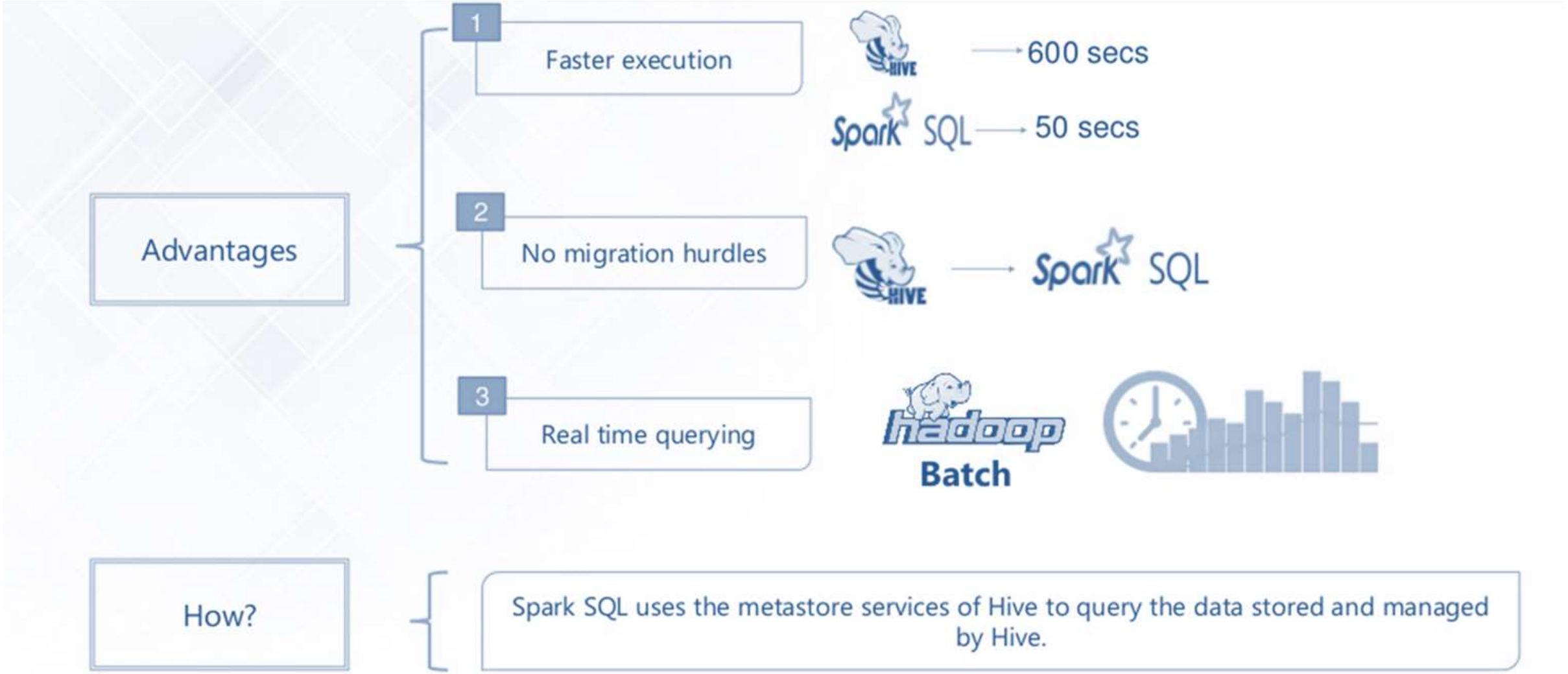


SPARK SQL

Why SPARK SQL



Advantages of Spark SQL over Hive



SPARK SQL Process

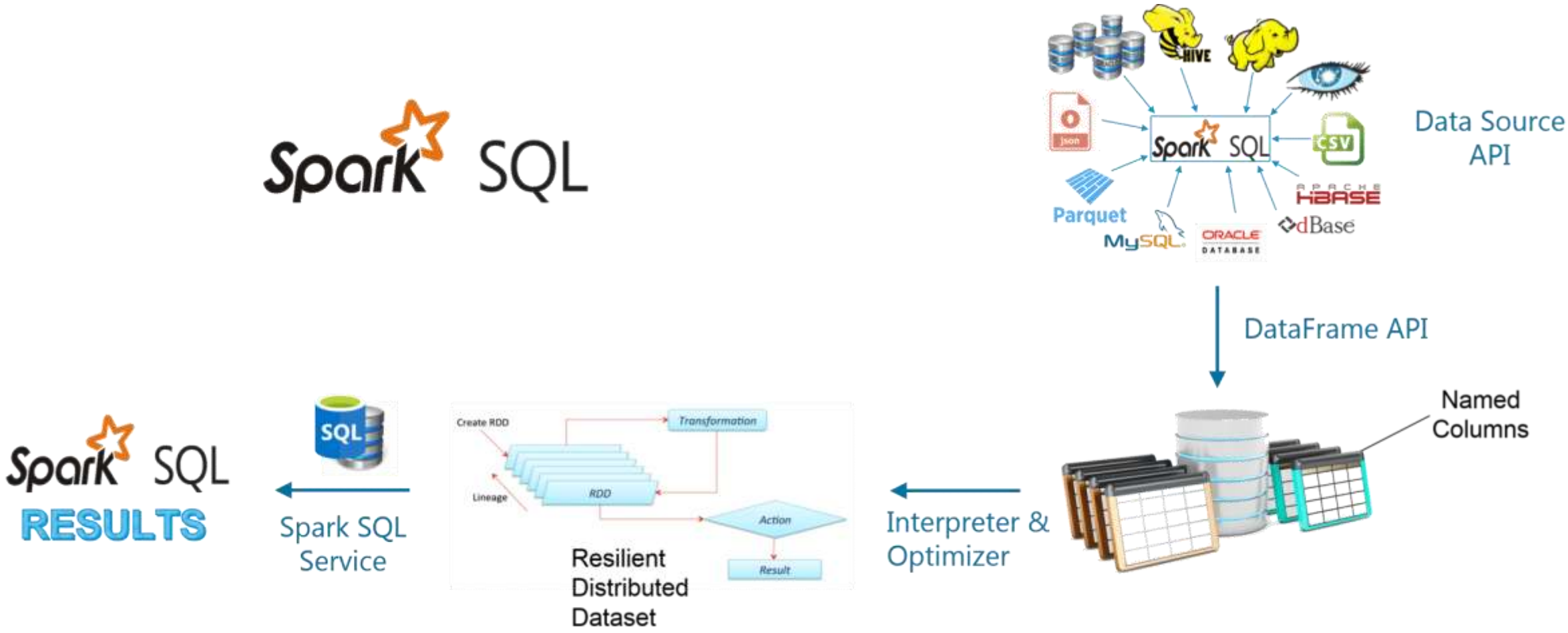


Figure: *The flow diagram represents a Spark SQL process using all the four libraries in sequence*

Generic Load/Save Functions of SPARK SQL Python

```
df = spark.read.load("examples/src/main/resources/users.parquet")
df.select("name","favorite_color").write.save("namesAndFavColors.parquet")
```

Giving Options Manually of which files to Read

```
df = spark.read.load("examples/src/main/resources/people.json", format="json")
df.select("name", "age").write.save("namesAndAges.parquet", format="parquet")
```

Reading file type csv

```
df = spark.read.load("examples/src/main/resources/people.csv", format="csv",
sep=":", inferSchema="true", header="true")
```

Spark SQL Lab

Microsoft Word Document

Spark Word Count Example

In this example, we use a few transformations to build a dataset of (String, Int) pairs called counts and then save it to a file.

```
// Please make sure you have context ready before following code
from pyspark.sql import Row
sc = spark.sparkContext
text_file = sc.textFile("hdfs://...") counts = text_file.flatMap(lambda line:
line.split(" ")) \ .map(lambda word: (word, 1)) \ .reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("hdfs://...")
```

Spark Error Log file Count Example

- In this example, we search through the error messages in a log file.
- Consider the log file have word “ERROR”

```
// Please make sure you have context ready before following codetextFile =
sc.textFile("hdfs://...")

# Creates a DataFrame having a single column named "line"
df = textFile.map(lambda r: Row(r)).toDF(["line"])
errors = df.filter(col("line").like("%ERROR%"))

# Counts all the errors
errors.count()

# Counts errors mentioning MySQL
errors.filter(col("line").like("%MySQL%")).count()

# Fetches the MySQL errors as an array of strings
errors.filter(col("line").like("%MySQL%")).collect()
```

More PySpark Examples

https://github.com/apache/spark/tree/master/examples/src/main/python

ml	[SPARK-26970][PYTHON][ML] Add Spark ML interaction transformer to PyS...	2 months ago
mllib	[SPARK-26640][CORE][ML][SQL][STREAMING][PYSPARK] Code cleanup from lg...	5 months ago
sql	[SPARK-27834][SQL][R][PYTHON] Make separate PySpark/SparkR vectorizat...	12 days ago
streaming	[MINOR][EXAMPLES] Add missing return keyword streaming word count exa...	3 months ago
als.py	[MINOR] Remove unused arg in als.py	3 years ago
avro_inputformat.py	[SPARK-23522][PYTHON] always use sys.exit over builtin exit	last year
kmeans.py	[SPARK-23522][PYTHON] always use sys.exit over builtin exit	last year
logistic_regression.py	[SPARK-23522][PYTHON] always use sys.exit over builtin exit	last year
pagerank.py	[SPARK-23522][PYTHON] always use sys.exit over builtin exit	last year
parquet_inputformat.py	[SPARK-23522][PYTHON] always use sys.exit over builtin exit	last year
pi.py	[SPARK-20779][EXAMPLES] The ASF header placed in an incorrect locatio...	2 years ago
sort.py	[SPARK-23522][PYTHON] always use sys.exit over builtin exit	last year
status_api_demo.py	[SPARK-19134][EXAMPLE] Fix several sql, mllib and status api examples...	3 years ago
transitive_closure.py	[SPARK-15773][CORE][EXAMPLE] Avoid creating local variable `sc` in ex...	3 years ago
wordcount.py	[SPARK-23522][PYTHON] always use sys.exit over builtin exit	last year

Introduction to SCALA

Scala

- Created by Martin Odersky and he released the first version in 2003
- Scala is a object-oriented language , every value is an object
- Scala is also a functional language in the sense that every function is a value

  and every value is an object so ultimately every function is an object.
- Provides a lightweight syntax for defining **anonymous functions**
- Allows functions to be **nested**
- Scala is compiled into Java Byte Code which is executed by the Java Virtual Machine (JVM)
- Scala can execute Java Code

Scala

- Scala and Java are similar ,the *';'* is optional in Scala
- Objects : have state and behaviour, it is an instance of class
- Class : class can be defined as a blueprint that describes the behaviour/state that relates to the class
- Method : A method is a behaviour ,class can contain many methods
- Fields : Each object has its unique set of instance variables, which are called as fields
- Closure : Function whose return value depends on value of one or more variables declared outside the function
- Traits : it encapsulates method and field definitions, which can then be reused by mixing them into classes.

Scala : Basic Syntax

- scala> println(“Hello World”);
- Scala is case sensitive
- First letter of class names should be in Upper case (eg : HelloWorldClass)
- Method names should start with lower case letters ,whereas if multiple words are used then the inner word’s names first letter should be in Upper case
- Name of the Program file should exactly match the object name
- File should be saved using object name and append ‘**.scala’** to the end of the name

Scala Identifiers

- Identifiers are names used for objects, classes, variables and methods
- Alphanumeric Identifiers : starts with letter or underscore
- ‘$’ character is reserved keyword in Scala and shouldn’t be used in identifiers
- eg : name, age, _amount, _1_amount
- Operator identifier : consist of one or more operator characters , these are printable ASCII characters such as + , : , ? , ~, #
- Mixed identifier : it consist of an alphanumeric identifier, followed by an underscore and an operator identifier

Reserved Keyword in Scala

1. abstract
2. case
3. catch
4. class
5. def
6. do
7. extends
8. else
9. false
10. final
11. finally
12. for
13. forsome
14. if
15. implicit
16. import
17. lazy
18. match
19. new
20. null
21. object
22. override
23. package
24. private
25. protected
26. return
27. sealed
28. super
29. this
30. throw
31. trait
32. try
33. true
34. type
35. Val
36. Var
37. While
38. With
39. yield

Scala Packages

- Package can be considered to be named module of code
- Lift utility package is net.liftweb.util
- package is declared as : package com.liftcode
- Scala packages can be imported so as to reference them in current compilation
- We can also import one or a few classes or objects from a package

Data Type and Literal

- **Scala has same data type as Java**
- But in Scala data types are also objects i.e. we can call methods on an

  Int, Long etc.
- **Integral Literals :** Integer literals are of type Int or Long when followed by a L or l
- **Floating Point :** when followed by F or f the literals are of type float
- **Character literal :** enclosed in single(‘ ‘)quotes
- **String literal :** enclosed in double quotes (“ ”)
- **Symbol literal :** A symbol literal ‘x is equivalent to **scala.Symbol(“x”)**

Variables

- Variable can be defined as constant or variable
- Variables declared using keyword *'var' are mutable i.e.* they can change values
- Variables declared using keyword *'val' are immutable i.e.* they cannot change values
- Variable Type Interference : Scala can understand the type of variable by the value assigned to it.
- Multiple assignments can be done using Scala (eg : val (Var1 : String, Var2: Int) = Pair("Hello", 10)

Variable Scope

- Fields : Variables that belong to the object , fields can be accessed from inside every method in the object
- Method Parameters are variables used to pass the value inside a method when a method is called
- Local variables are declared inside a method and are only accessible from inside the method

Access Specifiers

- **Public :** can be modified and accessed from anywhere
- **Private :** visible only inside the class or object that contains the member definition, keyword ‘PRIVATE’ is required
- **Protected :** can be accessed only from subclasses of the class in which member is defined, keyword ‘PROTECTED’ is required

Scala- Classes and Objects

- Class in Scala is defined similar to Java and the class name works as class constructor which can take a number of parameters
- Eg : class Employee(name: String, empId: Int, salary: Double)

  var name_emp: String = name

  var emp_ID: Int = empId

  var salary_emp: Double = salary

- We can **extend** a Scala class(using extends keyword) and design an inherited class as Java
- Method overriding requires '**override**' keyword and only primary constructor can pass parameters to base constructor

Scala- Classes and Objects

- Implicit classes allow implicit conversations with class's primary constructor when the class is in scope
- Scala has **singleton objects** : A singleton can have only one instance, i.e., Object
- **Object** keyword is used to create singleton class
- We cant instantiate a singleton object ,neither pass parameters to the primary constructor

Operators

- Arithmetic Operators : **+, - ,*, /, %** are the operators supported by Scala
- Relational Operators : **==, !=, >, <, >=, <=**
- Logical Operators : **&&, ||, !**
- Bitwise Operators : **&, |, ^**
- Assignment Operator : **=, +=, -=, *=, /=, %=, <<=, >>=, &=, ^=, !=**

If-Else

- ‘If’ statement consists of a Boolean expression followed by statements
- ‘If’ can be followed by an ‘Else’ statement which executes when the Boolean expression is false
- An ‘If’ can be followed by an optional ‘else if….else’ statement, which is very useful to test various conditions using single if…..else if statement.
- We can nest **if-else** statements, which means we can use one **if** or **else-if**
- statement inside another **if** or **else-if** statement

String

- String keyword is not compulsory to be used for creating a String in Scala
- String class is *immutable,* String object once created cannot be changed
- String Length can be obtained using length() method
- Two strings can be concatenated using concat() method
- String Interpolation is the new way to create Strings in Scala programming language
- The literal ‘s’ allows the usage of variable directly in processing a string, when we prepend ‘s’ to it
- The ‘f’ interpolator allows to create a formatted String
- The ‘raw’ interpolator is similar to ‘s’ interpolator except that it performs no escaping of literals within a string.

Arrays

- We need to declare a variable to reference an array, and we need to specify the type of array the variable can reference
- Matrices and Tables are examples of structures that can be realized as 2-D arrays.
- We can concat arrays using concat() method
- range() method can be used to generate an array containing a sequence of increasing integers in a given range
- We can use the inbuilt methods of Scala by importing the array package

Collections

- Containers of things are collections
- Collections may be **strict** or **lazy**
- Lazy collections have elements that may not consume memory until accessed
- Collections can be **mutable or immutable**
- Immutable collections can contain mutable items
- **Lists(linked list), Sets, Maps, Tuples, etc are** commonly used collections

Traits

- A trait encapsulates method and field definitions, which can then be reused by mixing them into classes
- Used to define object types by specifying the signature of the supported methods
- Similar to class definition except that keyword *'trait'* is used
- Trait is similar to *abstract class* in Java
- Value classes :

  1) are used to avoid allocating runtime objects

  2) consists of primary constructor with exactly one **val** parameter

  3) cannot be extended by another class

MLlib: Statistics

MLlib: Statistics

- *MLlib offers widely used statistic functions that work directly on RDDs*
- *Statistics.colStats(rdd) : Computes a statistical summary of RDD of vectors, which stores the min, max and variance for each column in the set of vectors*
- *Statisitcs.corr(rdd, method) : It computes the correlation matrix between the Pearson or Spearman correlation*
- *Statistics.chiSqTest(rdd) : Computes Pearson's independence test for every feature with the label on an RDD of LabeledPoint objects.*
- *Statistics.corr(rdd1, rdd2, method) : Computes correlation between two RDD's of floating-point values, using Pearson or Spearman correlation*

Classifications and Regression

- Classification and Regression are two forms of *supervised learning*
- In classification we predict discrete variable (i.e. it takes on a finite set of values called *classes)*
- In Regression the variable is *continuous* (eg: height of a person given his age and weight)
- Both used LabeledPoint class in MLlib
- LabeledPoint consists of a *label* and a *feature* vector

Linear Regression

- Used to predict the output as a linear combination of features of input
- It supports L^1 and L^2 regularized regression, known as *Lasso and ridge regression*
- The algorithms are available with *LinearRegressionWith SGD, LassoWithSGD and RidgeRegressionWithSGD* classes, SGD refers to Stochastic Gradient Descent
- *numIterations, stepSize and intercept* are several parameters to tune the algorithm

Spark

Linear regression in Python

```
from pyspark.MLlib.regression import LabeledPoint
from pyspark.MLlib.regression import LinearRegressionWithSGD
points = # (create RDD of LabeledPoint)
model = LinearRegressionWithSGD.train(points, iterations=200, intercept=True)
print "weights: %s, intercept: %s" % (model.weights, model.intercept)
```

Linear regression in Scala

```
import org.apache.spark.MLlib.regression.LabeledPoint
import org.apache.spark.MLlib.regression.LinearRegressionWithSGD
val points: RDD[LabeledPoint] = // ...
val lr = new LinearRegressionWithSGD().setNumIterations(100).setIntercept(true)
val model = lr.run(points)
println("weights: %s, intercept: %s".format(model.weights, model.intercept))
```

Logistic Regression

- It’s a method that identifies linear separating plane between positive and negative examples.
- MLlib takes *LabeledPoints* with label 0 or 1 and returns a *LogisticRegressionModel* that can predict new points
- SGD and LBFGS are two algorithms available for logistic regression
- The LogisticRegressionModel computes a score between 0 and 1
- We can change the threshold by *setThreshold()* and disable it using *clearThreshold()*

Example: Python Code

```
# Logistic Regression - iterative machine learning algorithm
# Find best hyperplane that separates two sets of points in a
# multi-dimensional feature space. Applies MapReduce operation
# repeatedly to the same dataset, so it benefits greatly
# from caching the input in RAM

points = spark.textFile(...).map(parsePoint).cache()
w = numpy.random.ranf(size = D) # current separating plane
for i in range(ITERATIONS):
    gradient = points.map(
        lambda p: (1 / (1 + exp(-p.y*(w.dot(p.x)))) - 1) * p.y * p.x
    ).reduce(lambda a, b: a + b)
    w -= gradient
print "Final separating plane: %s" % w
```

Example: Scala Code

```
// Same in Scala

val points = spark.textFile(...).map(parsePoint).cache()
var w = Vector.random(D) // current separating plane
for (i <- 1 to ITERATIONS) {
  val gradient = points.map(p =>
    (1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y * p.x
  ).reduce(_ + _)
  w -= gradient
}
println("Final separating plane: " + w)
```