PySpark Version!

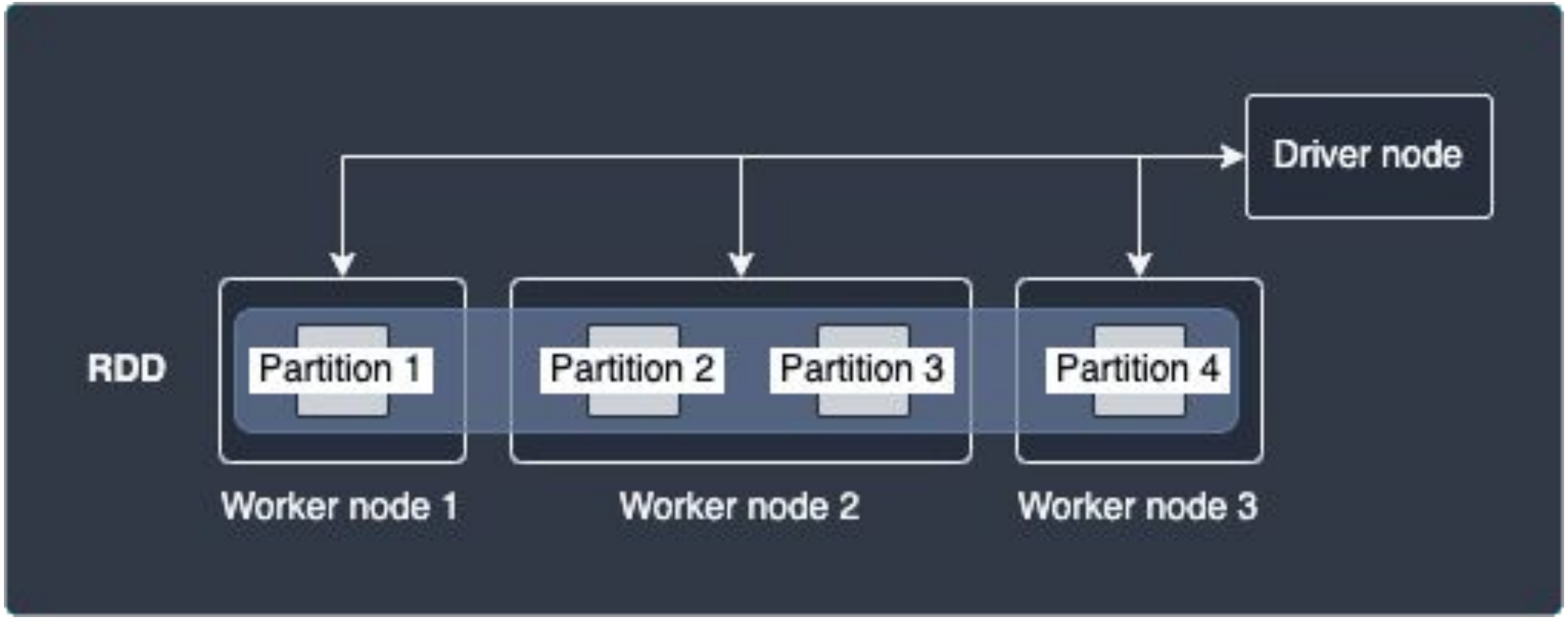
Dr. Ernesto Lee

Setup and Validate Your Environment

```
columns = ["name", "age"]

data = [("Alex", 15), ("Bob", 20), ("Cathy", 25)]

df = spark.createDataFrame(data, columns)

df.show()
```

RDD's Redux

What is an RDD?

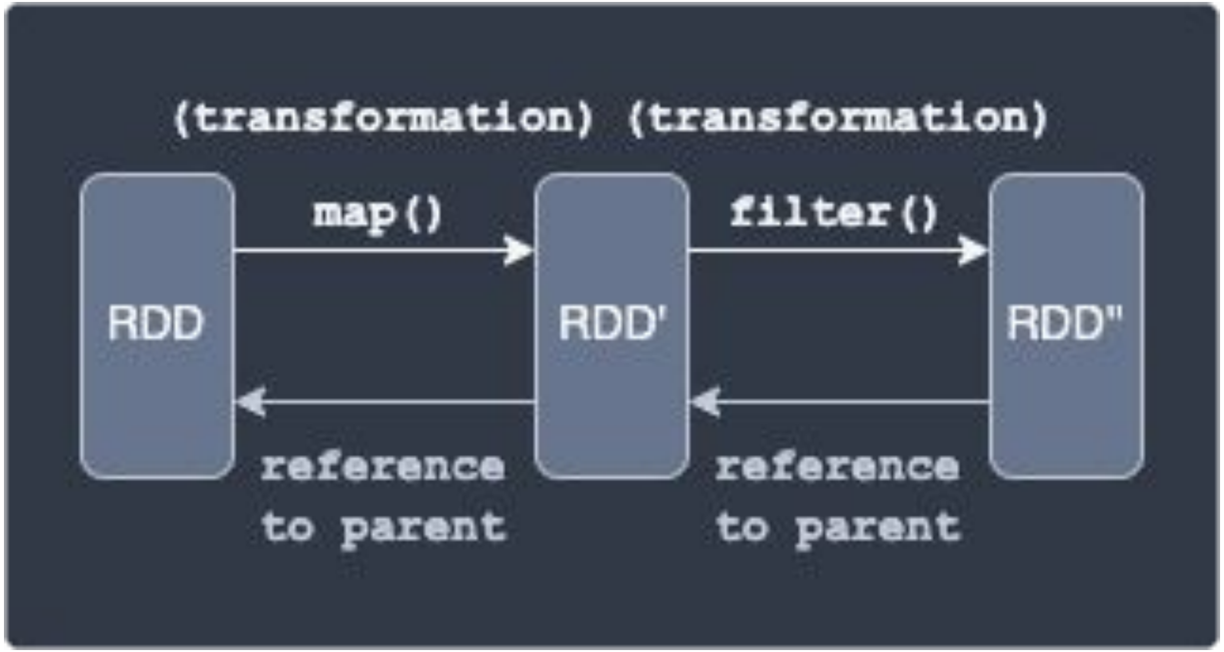


Transformation and Actions

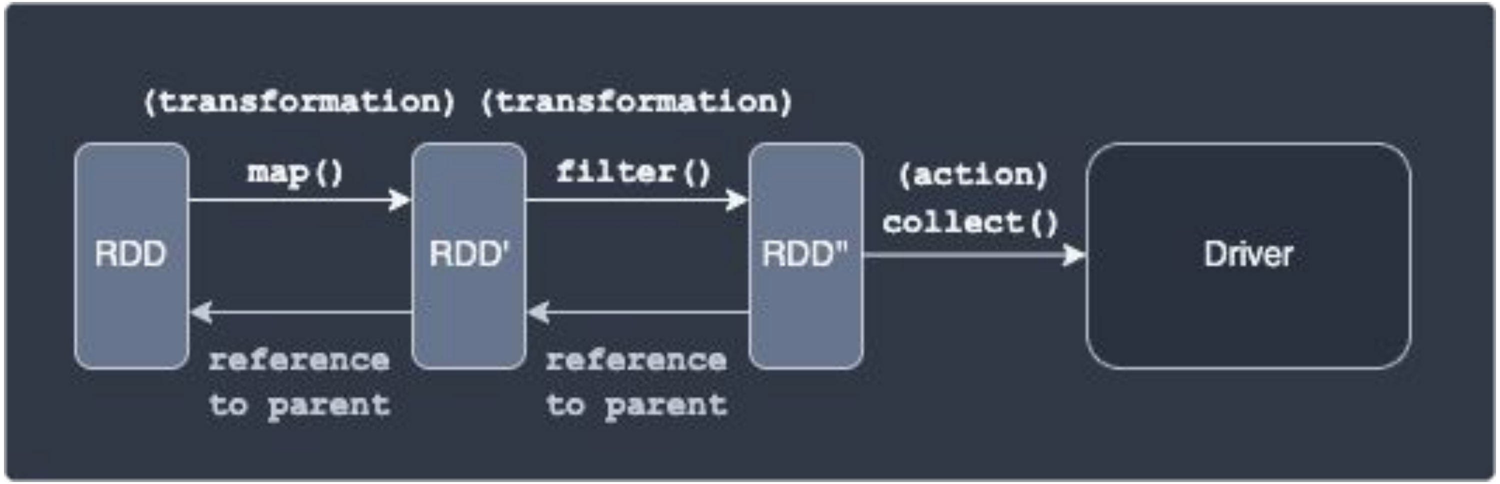
There are two operations we can perform on a RDD:

- Transformations
- Actions

Transformations



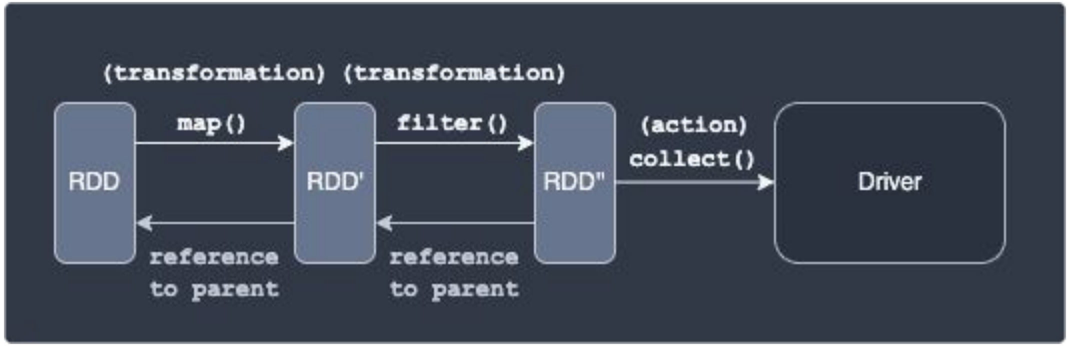
Actions



(transformation)
(transformation)
map()
filter()
(action)
collect()
RDD
RDD'
RDD"
Driver
reference
to parent
reference
to parent

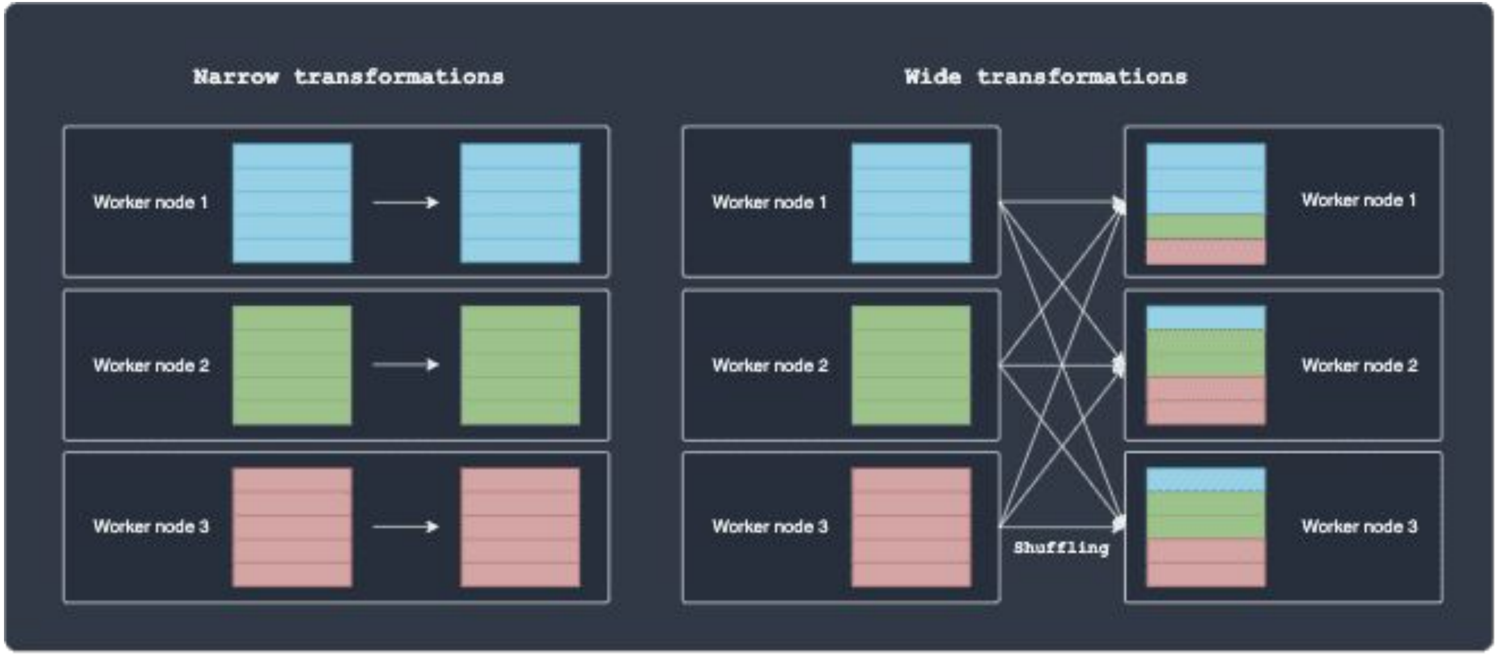
Demo

```
rdd = sc.parallelize(["Alex","Bob","Cathy"], numSlices=3)

rdd.collect()
```

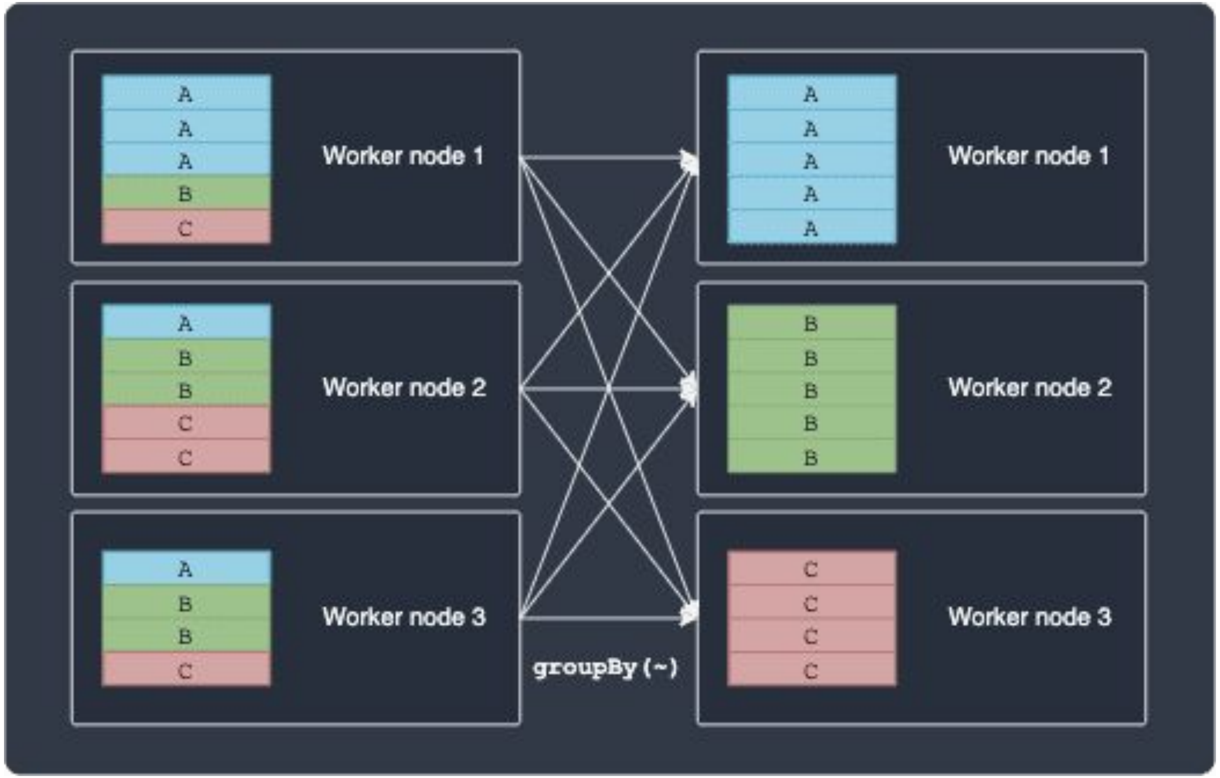


```
rdd2 = rdd1.map(lambda x: x.upper())

rdd3 = rdd2.filter(lambda name: name == "ALEX")

rdd3.collect()
```

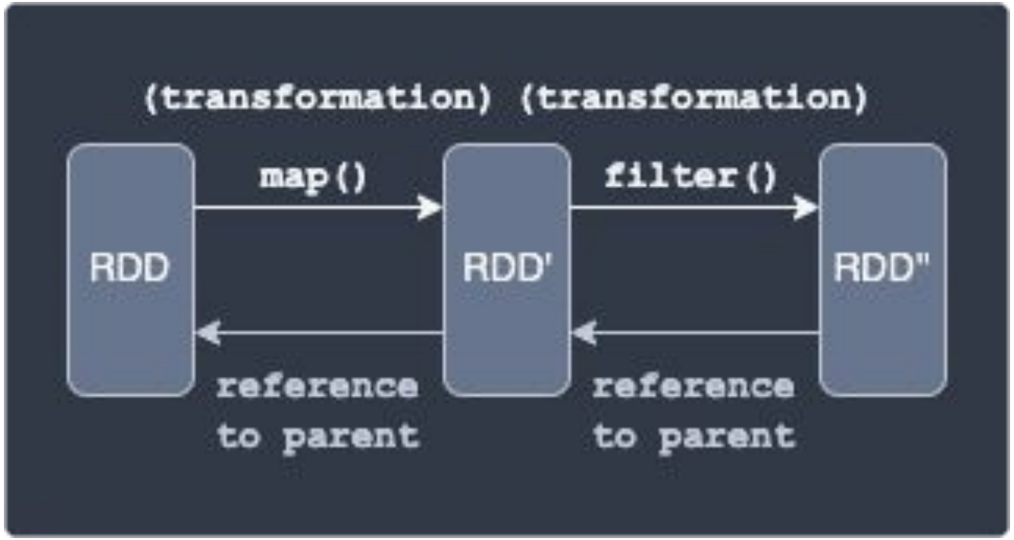
Narrow and Wide Transformations



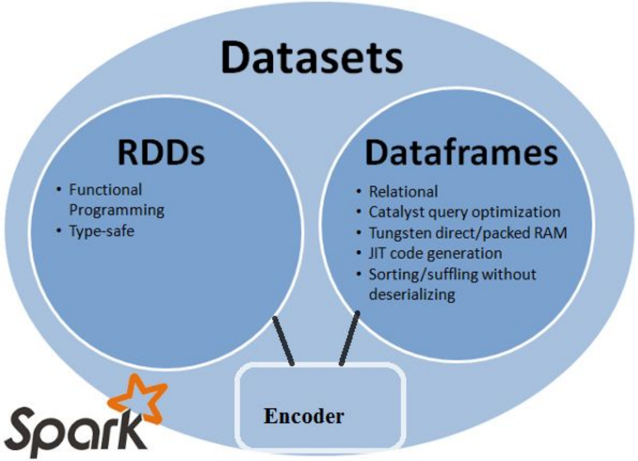
Wide Transformations



Fault Tolerance



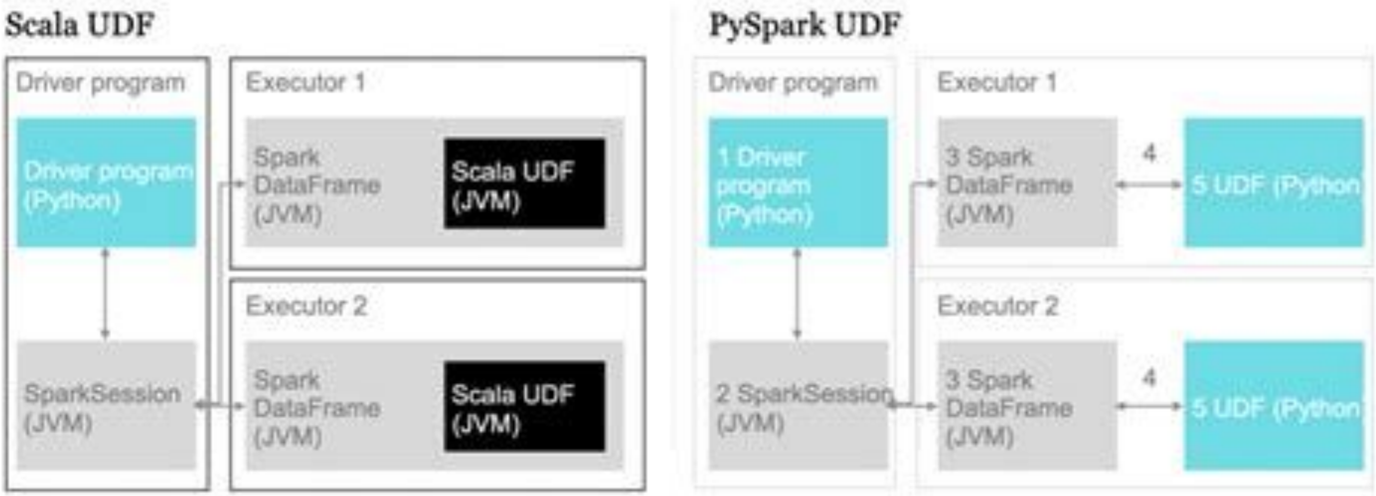
RDDs versus Dataframes/Datasets



	RDD	Dataframe	Dataset
Optimization	No optimization engine, never use catalyst optimizer and tungsten execution engine.	Use catalyst optimizer tungsten execution engine.	More advance than DF.
Serialization	Use java serialization to store or distribute the data and its expensive and require sending both data and structure nodes.	Serialize data into off-heap (in-memory)in binary format and apply transformations and use tungsten execution engine to manage memory and dynamically generates bytecode.	Dataset API use encoder concept and store tabular representation using tungsten.
Garbage Collection	Overhead of garbage collection.	Avoids the garbage collection costs in constructing individual objects for each row in the dataset.	There is also no need for the garbage collector to destroy object because serialization takes place through Tungsten. That uses off heap data serialization.

User Defined Functions

What is a UDF?



Scala UDF
Driver program
Driver program (Python)
SparkSession (JVM)
Executor 1
Spark DataFrame (JVM)
Scala UDF (JVM)
Executor 2
Spark DataFrame (JVM)
Scala UDF (JVM)
PySpark UDF
Driver program
1 Driver program (Python)
2 SparkSession (JVM)
Executor 1
3 Spark DataFrame (JVM)
4
5 UDF (Python
Executor 2
3 Spark DataFrame (JVM)
4
5 UDF (Python

Apply a Custom Function on a Feature

```
df = spark.createDataFrame([['Alex',10], ['Bob',20],
['Cathy',30]], ['name','age'])
```

```
df.show()
```

Custom Function with an Argument

```
def to_upper(some_string):
    return some_string.upper()
```

Register your Function

```
from pyspark.sql.functions import udf

# Register our custom function

udf_upper = udf(to_upper)

# We can use our custom function just like we would for any
SQL function

df.select(udf_upper('name')).show()
```

I know… it's built in

```
from pyspark.sql import functions as F

df.select(F.upper('name')).show()
```

Multi-Column Function

```
# Takes in as argument two column values
def my_func(str_name, int_age):
    return f'{str_name} is {int_age} years old'

my_udf = udf(my_func)
# Pass in two columns to our my_udf
df_result = df.select(my_udf('name', 'age'))
df_result.show()
```

Define the Column Type

```
def my_double(int_age):
    return 2 * int_age


# Register the function
udf_double = udf(my_double)
df_result = df.select(udf_double('age'))
df_result.show()
```

Set the responses

```
df_result.printSchema()

udf_double = udf(my_double, 'int')

df_result = df.select(udf_double('age'))

df_result.printSchema()
```

All together now

```
from pyspark.sql.types import IntegerType

udf_double = udf(my_double, IntegerType())

df_result = df.select(udf_double('age'))

df_result.printSchema()
```

UDF in SQL Expressions

```
def to_upper(some_string):
    return some_string.upper()


spark.udf.register('udf_upper', to_upper)
df.selectExpr('udf_upper(name)').show()
```

Register the DF as a SQL Table and Specify the return type

```
# Register PySpark DataFrame as a SQL table
df.createOrReplaceTempView('my_table')
spark.sql('SELECT udf_upper(name) FROM my_table').show()
def my_double(int_age):
    return 2 * int_age

spark.udf.register('udf_double', my_double, 'int')
df.selectExpr('udf_double(age)').printSchema()
```

Import explicit

```
from pyspark.sql.types import IntegerType

spark.udf.register('udf_double', my_double, IntegerType())

df.selectExpr('udf_double(age)').printSchema()
```

Gotchas with UDFs

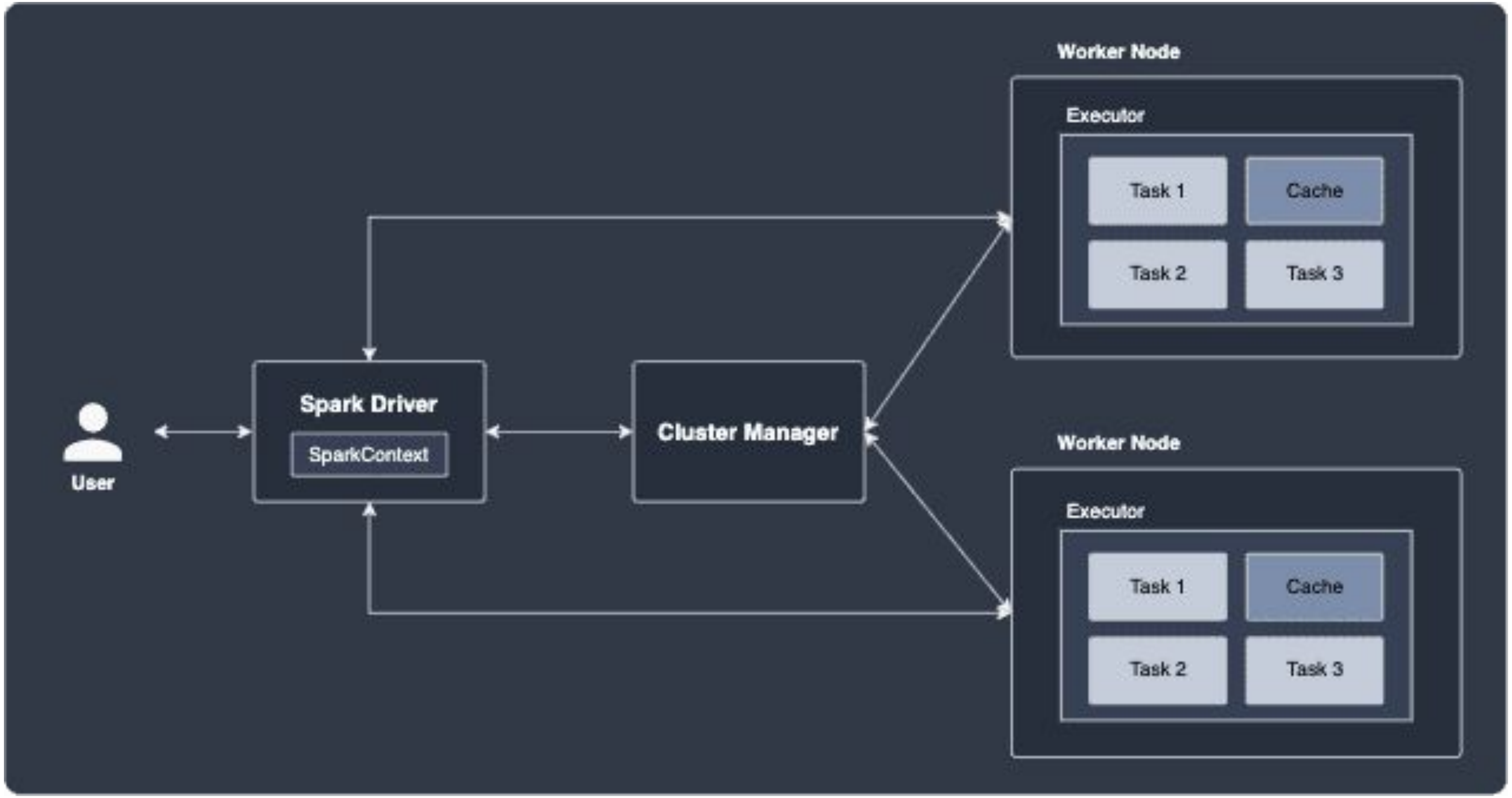
```
spark.udf.register('my_double', lambda val: 2 * val, 'int')

spark.sql('SELECT * from my_table WHERE age IS NOT NULL AND
my_double(age) > 5').show()
```

Un-Gotcha

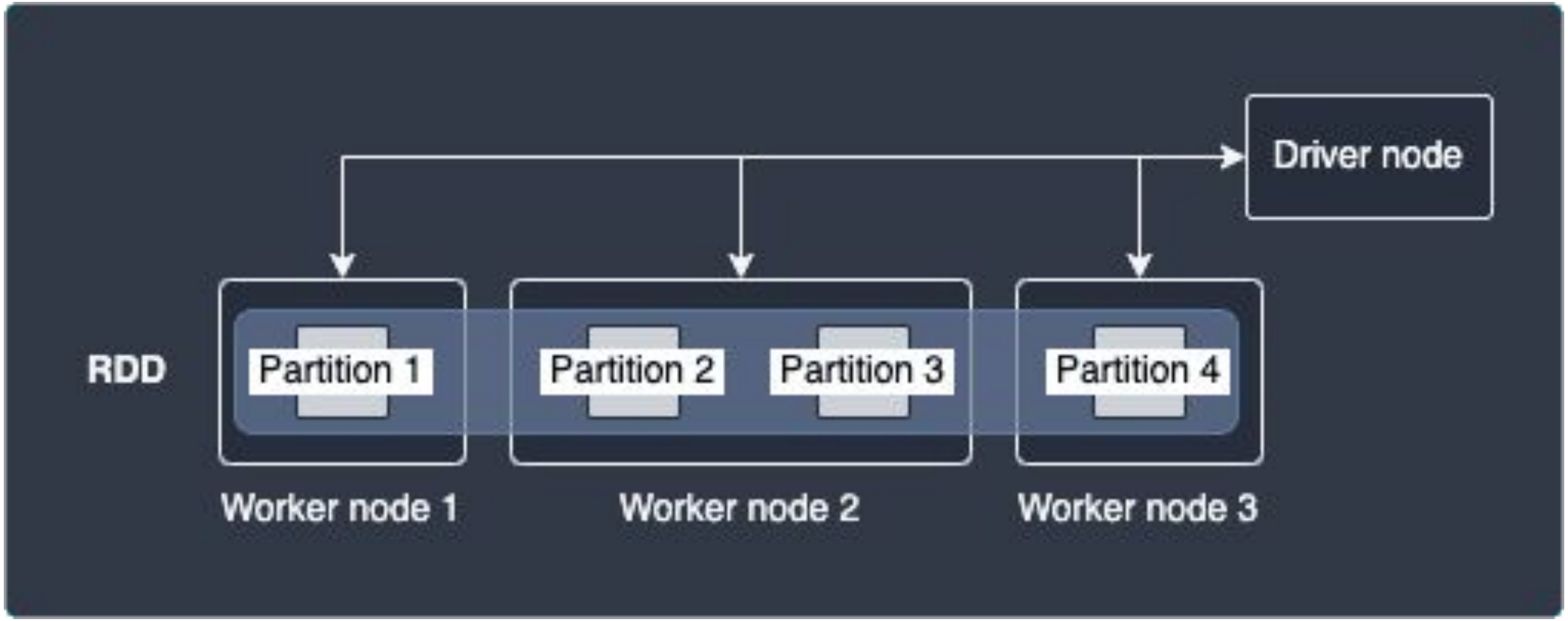
```
spark.udf.register('my_double', lambda val: 2 * val, 'int')

spark.sql('SELECT * from my_table WHERE IF(age IS NOT NULL,
my_double(age) > 5, null) IS NOT NULL').show()
```

Big Picture

BLUF

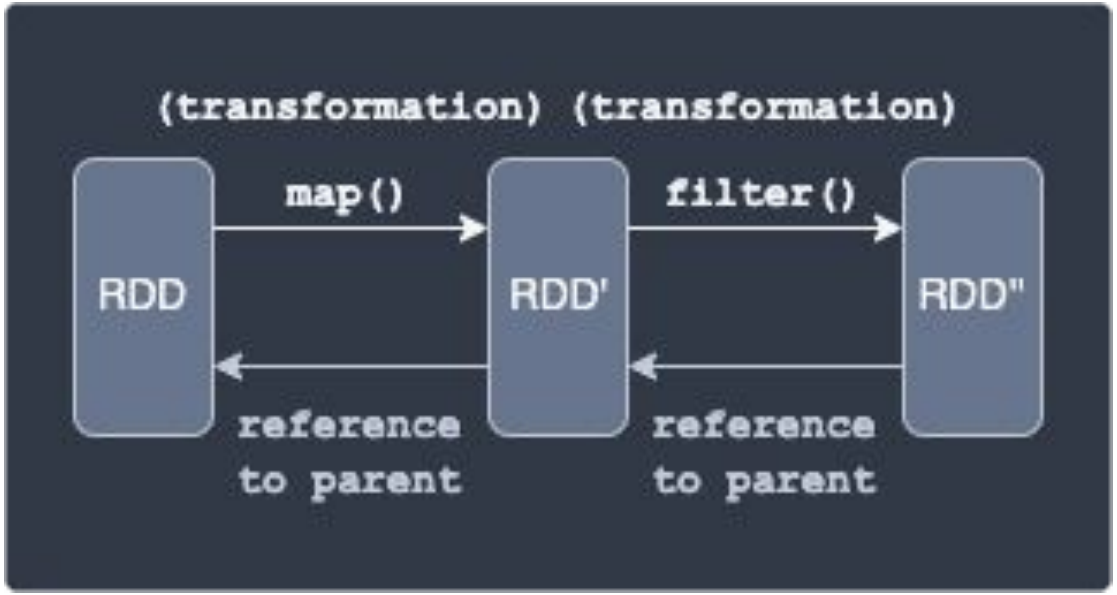

Worker Node
Executor
Task 1
Cache
Task 2
Task 3
Spark Driver
SparkContext
User
Cluster Manager
Worker Node
Executor
Task 1
Cache
Task 2
Task 3

RDDs

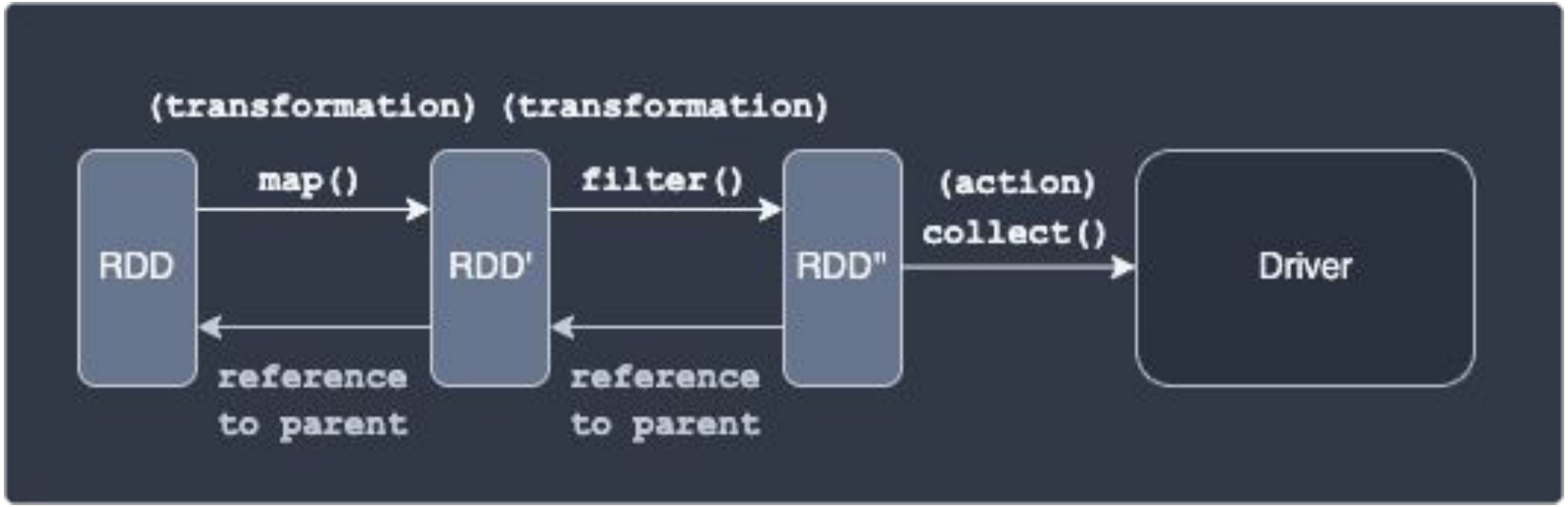


Driver node
RDD
Partition 1
Partition 2
Partition 3
Partition 4
Worker node 1
Worker node 2
Worker node 3

Transformation

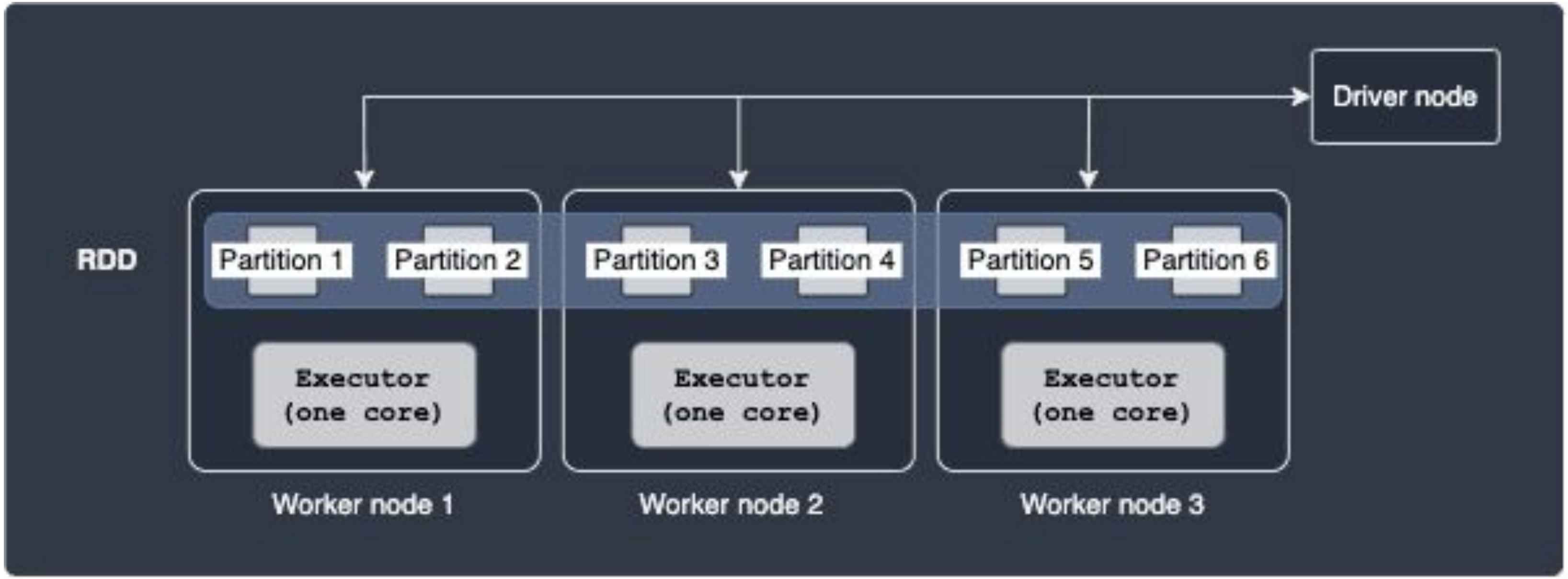


Actions



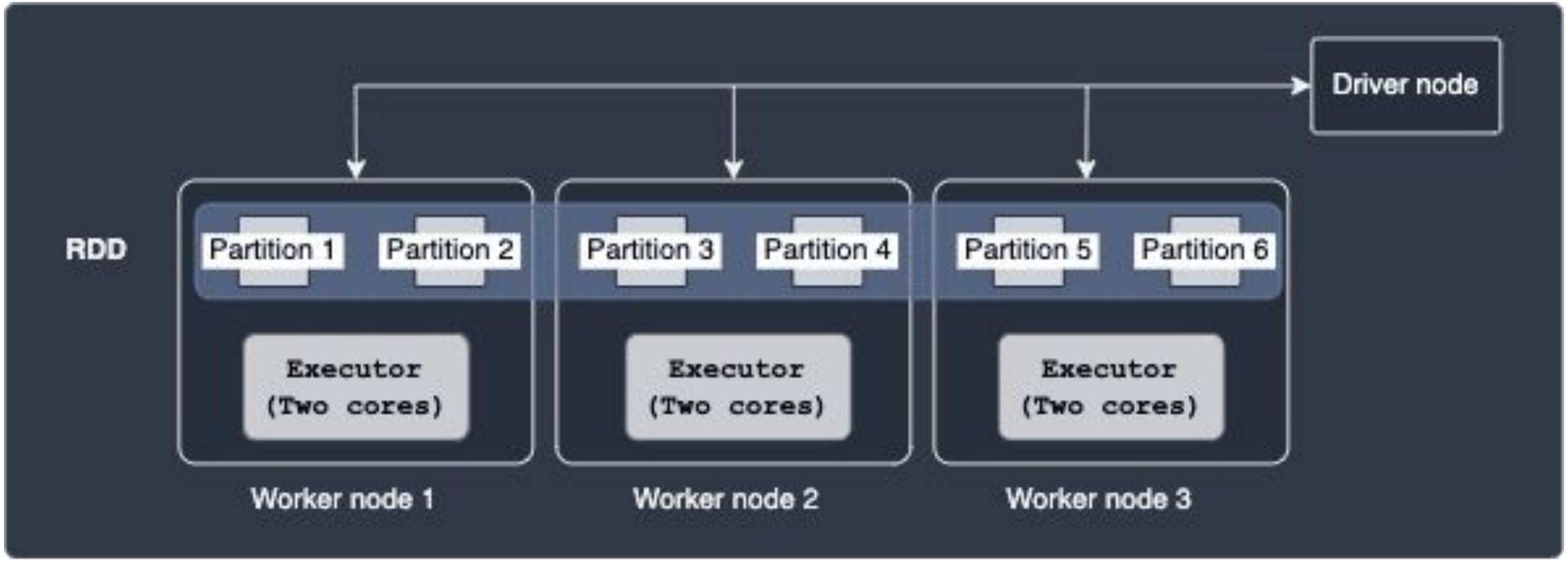
(transformation) (transformation)
map()
filter()
(action)
collect()
RDD
RDD'
RDD"
Driver
reference
to parent
reference
to parent

Executors

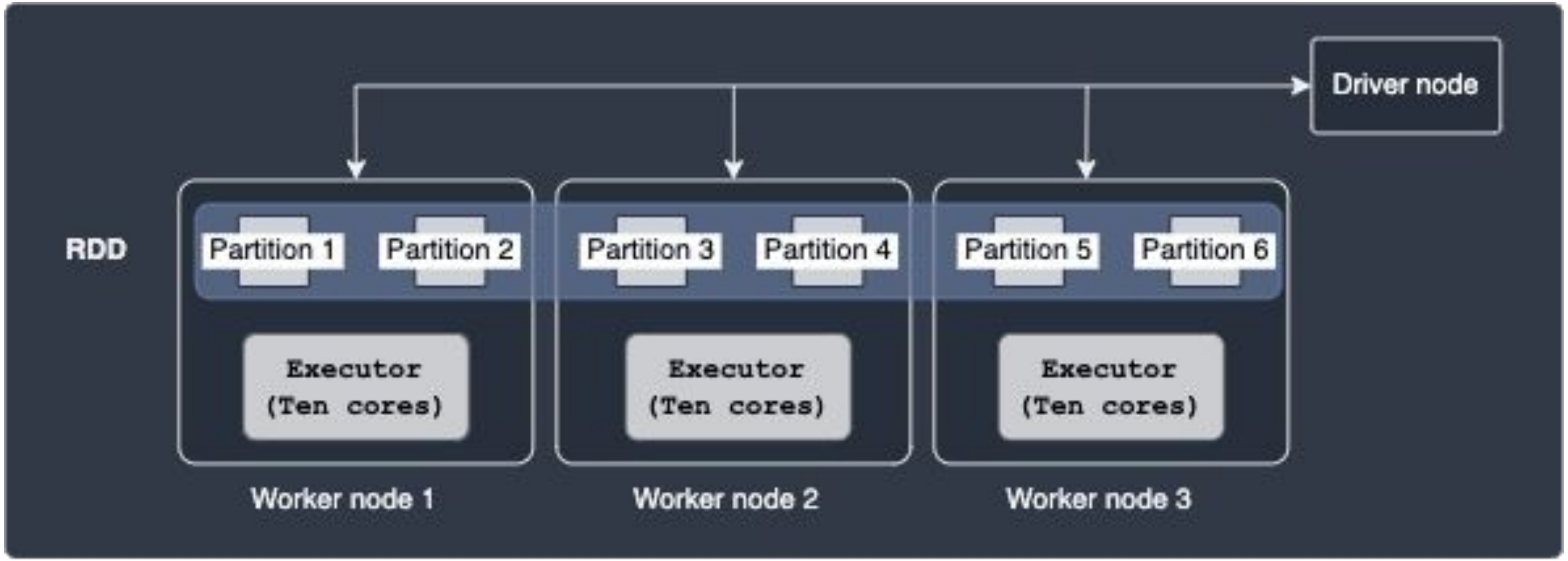


Driver node
RDD
Partition 1
Partition 2
Partition 3
Partition 4
Partition 5
Partition 6
Executor
(one core)
Executor
(one core)
Executor
(one core)
Worker node 1
Worker node 2
Worker node 3

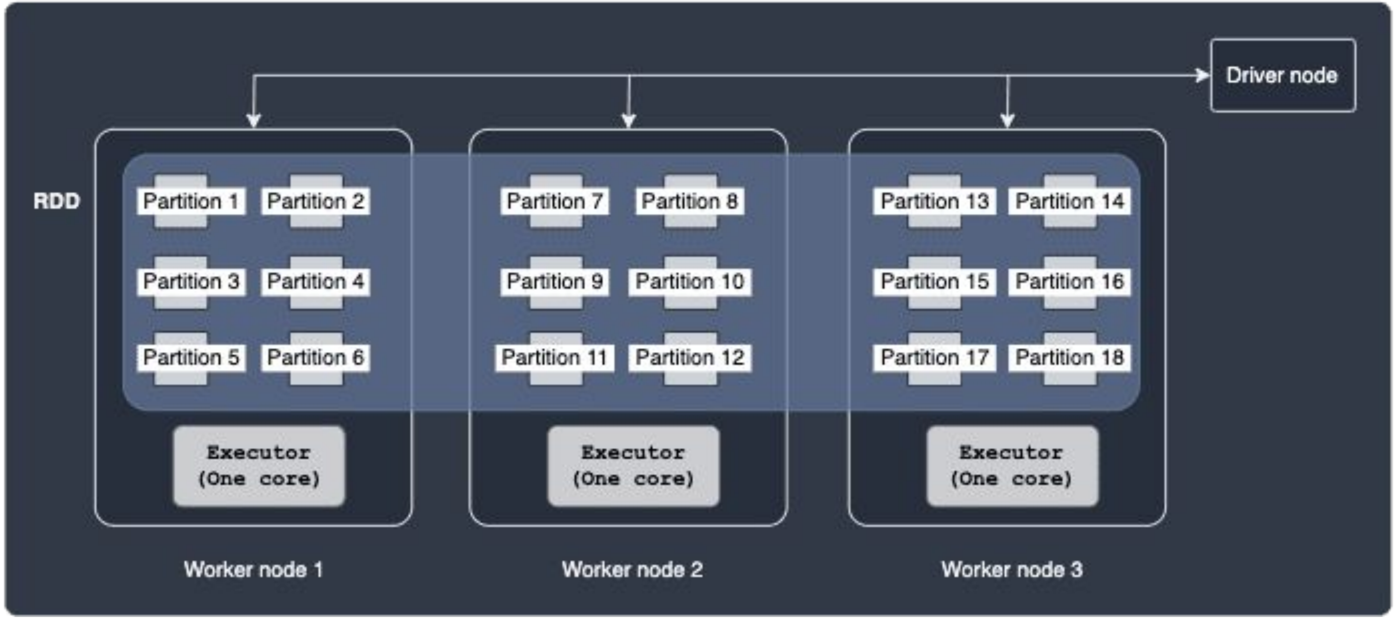
Multiple Cores



Number of Partitions (Under Partitions)



Over Partitions



The Goldilocks rule for the perfect number of partitions

Iterations with PySpark

Map

```
df = spark.createDataFrame([("Alex", 15), ("Bob", 20),
("Cathy", 25)], ["name", "age"])

df.show()
```

Map

```
from pyspark.sql import Row

def my_func(row):
    d = row.asDict()
    d.update({'name': d['name'].upper()})
    updated_row = Row(**d)
    return updated_row

rdd = df.rdd.map(my_func)
rdd.toDF().show()
```

Collect

```
df = spark.createDataFrame([["Alex", 20], ["Bob", 30],
["Cathy", 40]], ["name", "age"])

df.show()

for row in df.collect():

    print(row.name)
```

For Each

```
# This function fires in the worker node
def f(row):
    print(row.name)

df.foreach(f)
```

Machine Learning Use Case on Spark

Getting Started with Spark Context

```
import pyspark

from pyspark import SparkContext

sc =SparkContext()


nums= sc.parallelize([1,2,3,4])


nums.take(1)
```

Transformation

```
squared = nums.map(lambda x: x*x).collect()
for num in squared:
    print('%i ' % (num))
```

SQL Context

```
from pyspark.sql import Row
from pyspark.sql import SQLContext


sqlContext = SQLContext(sc)
```

Step 1) Create the list of tuple with the information

```
[('John',19),('Smith',29),('Adam',35),('Henry',50)]
```

Step 2) Build a RDD

```
rdd = sc.parallelize(list_p)
```

Step 3) Convert the tuples

```
rdd.map(lambda x: Row(name=x[0], age=int(x[1])))
```

Step 4) Create a DataFrame context

```
sqlContext.createDataFrame(ppl)
```

```
list_p = [('John',19),('Smith',29),('Adam',35),('Henry',50)]

rdd = sc.parallelize(list_p)

ppl = rdd.map(lambda x: Row(name=x[0], age=int(x[1])))

DF_ppl = sqlContext.createDataFrame(ppl)
```

If you want to access the type of each feature, you can use printSchema()

```
DF_ppl.printSchema()
```

Following are the steps to build a Machine Learning program with PySpark:

- **Step 1)** Basic operation with PySpark
- **Step 2)** Data preprocessing
- **Step 3)** Build a data processing pipeline
- **Step 4)** Build the classifier: logistic
- **Step 5)** Train and evaluate the model
- **Step 6)** Tune the hyperparameter

Step 1) Basic operation with PySpark

```
#from pyspark.sql import SQLContext

url =
"https://raw.githubusercontent.com/guru99-edu/R-Programming/master/adult_data.csv"

from pyspark import SparkFiles

sc.addFile(url)

sqlContext = SQLContext(sc)
```

Load your data

```
df = sqlContext.read.csv(SparkFiles.get("adult_data.csv"),
header=True, inferSchema= True)

df.printSchema()

df.show(5, truncate = False)
```

inferSchema=False

```
df_string = sqlContext.read.csv(SparkFiles.get("adult.csv"),
header=True, inferSchema=  False)

df_string.printSchema()
```

Recast to get the right data types

```
# Import all from `sql.types`
from pyspark.sql.types import *


# Write a custom function to convert the data type of DataFrame columns
def convertColumn(df, names, newType):
    for name in names:
        df = df.withColumn(name, df[name].cast(newType))
    return df
# List of continuous features
CONTI_FEATURES  = ['age', 'fnlwgt','capital_gain', 'education_num', 'capital_loss', 'hours_week']
# Convert the type
df_string = convertColumn(df_string, CONTI_FEATURES, FloatType())
# Check the dataset
df_string.printSchema()
```

Exploratory Data Analysis with Spark

```
df.select('age','fnlwgt').show(5)

df.groupBy("education").count().sort("count",ascending=True)
.show()

df.describe().show()

df.describe('capital_gain').show()

df.crosstab('age', 'label').sort("age_label").show()

df.drop('education_num').columns

df.filter(df.age > 40).count()

df.groupby('marital').agg({'capital_gain': 'mean'}).show()
```

Step 2) Data preprocessing

```
from pyspark.sql.functions import *

# 1 Select the column
age_square = df.select(col("age")**2)

# 2 Apply the transformation and add it to the DataFrame
df = df.withColumn("age_square", col("age")**2)

df.printSchema()
```

```
COLUMNS = ['age', 'age_square', 'workclass', 'fnlwgt',
'education', 'education_num', 'marital',
           'occupation', 'relationship', 'race', 'sex',
'capital_gain', 'capital_loss',
           'hours_week', 'native_country', 'label']
df = df.select(COLUMNS)
df.first()
```

Exclude Columns

```
df.filter(df.native_country == 'Holand-Netherlands').count()

df.groupby('native_country').agg({'native_country':
'count'}).sort(asc("count(native_country)")).show()

df_remove = df.filter(df.native_country !='Holand-Netherlands')
```

Step 3) Build a data processing pipeline

```
StringIndexer(inputCol="workclass", outputCol="workclass_encoded")


model = stringIndexer.fit(df)

indexed = model.transform(df)


OneHotEncoder(dropLast=False, inputCol="workclassencoded",
outputCol="workclassvec")
```

```
### Example encoder

from pyspark.ml.feature import StringIndexer, OneHotEncoder, VectorAssembler


stringIndexer = StringIndexer(inputCol="workclass", outputCol="workclass_encoded")

model = stringIndexer.fit(df)

indexed = model.transform(df)

encoder = OneHotEncoder(dropLast=False, inputCol="workclass_encoded",
outputCol="workclass_vec")

encoded = encoder.transform(indexed)

encoded.show(2)
```

Build the Pipeline

- Encode the categorical data
- Index the label feature
- Add continuous variable
- Assemble the steps

1. Encode the categorical data

```
from pyspark.ml import Pipeline

from pyspark.ml.feature import OneHotEncoderEstimator

CATE_FEATURES = ['workclass', 'education', 'marital', 'occupation', 'relationship', 'race',
'sex', 'native_country']

stages = [] # stages in our Pipeline

for categoricalCol in CATE_FEATURES:

    stringIndexer = StringIndexer(inputCol=categoricalCol, outputCol=categoricalCol + "Index")

    encoder = OneHotEncoderEstimator(inputCols=[stringIndexer.getOutputCol()],

                                     outputCols=[categoricalCol + "classVec"])

    stages += [stringIndexer, encoder]
```

2. Index the label feature

```
# Convert label into label indices using the StringIndexer

label_stringIdx =  StringIndexer(inputCol="label",
outputCol="newlabel")

stages += [label_stringIdx]
```

3. Add continuous variable

```
assemblerInputs = [c + "classVec" for c in CATE_FEATURES] +
CONTI_FEATURES
```

4. Assemble the steps

```
assembler = VectorAssembler(inputCols=assemblerInputs,
outputCol="features")stages += [assembler]
```

Publish to the pipeline

```
# Create a Pipeline.
pipeline = Pipeline(stages=stages)
pipelineModel = pipeline.fit(df_remove)
model = pipelineModel.transform(df_remove)



model.take(1)
```

Step 4) Build the classifier: logistic

```
from pyspark.ml.linalg import DenseVector

input_data = model.rdd.map(lambda x: (x["newlabel"],
DenseVector(x["features"])))


df_train = sqlContext.createDataFrame(input_data, ["label",
"features"])

df_train.show(2)
```

Create a train/test set

```
# Split the data into train and test sets

train_data, test_data =
df_train.randomSplit([.8,.2],seed=1234)


train_data.groupby('label').agg({'label': 'count'}).show()

test_data.groupby('label').agg({'label': 'count'}).show()
```

Build the logistic regressor

```
# Import `LinearRegression`
from pyspark.ml.classification import LogisticRegression

# Initialize `lr`
lr = LogisticRegression(labelCol="label",
                        featuresCol="features",
                        maxIter=10,
                        regParam=0.3)

# Fit the data to the model
linearModel = lr.fit(train_data)
```

Check the model coefficients

```
# Print the coefficients and intercept for logistic
regression

print("Coefficients: " + str(linearModel.coefficients))

print("Intercept: " + str(linearModel.intercept))
```

Step 5) Train and evaluate the model

```
# Make predictions on test data using the transform()
method.

predictions = linearModel.transform(test_data)

predictions.printSchema()
```

You are interested in the label, prediction and the probability

```
selected = predictions.select("label", "prediction", "probability")

selected.show(20)
```

Evaluate the model

```
cm = predictions.select("label", "prediction")

cm.groupby('label').agg({'label': 'count'}).show()

cm.groupby('prediction').agg({'prediction': 'count'}).show()

cm.filter(cm.label == cm.prediction).count() / cm.count()
```

Wrap this in a method

```
def accuracy_m(model):
    predictions = model.transform(test_data)
    cm = predictions.select("label", "prediction")
    acc = cm.filter(cm.label == cm.prediction).count() /
cm.count()
    print("Model accuracy: %.3f%%" % (acc * 100))
accuracy_m(model = linearModel)
```

ROC metrics

```
### Use ROC

from pyspark.ml.evaluation import BinaryClassificationEvaluator


# Evaluate model

evaluator =
BinaryClassificationEvaluator(rawPredictionCol="rawPrediction")

print(evaluator.evaluate(predictions))

print(evaluator.getMetricName())

print(evaluator.evaluate(predictions))
```

Step 6) Tune the hyperparameter

```
from pyspark.ml.tuning import ParamGridBuilder,
CrossValidator


# Create ParamGrid for Cross Validation

paramGrid = (ParamGridBuilder()

             .addGrid(lr.regParam, [0.01, 0.5])

             .build())
```

Cross Validation

```
from time import *

start_time = time()

# Create 5-fold CrossValidator

cv = CrossValidator(estimator=lr,

                    estimatorParamMaps=paramGrid,

                    evaluator=evaluator, numFolds=5)

# Run cross validations

cvModel = cv.fit(train_data)

# likely take a fair amount of time

end_time = time()

elapsed_time = end_time - start_time

print("Time to train model: %.3f seconds" % elapsed_time)
```

Choose the best parameters

```
accuracy_m(model = cvModel)

bestModel = cvModel.bestModel

bestModel.extractParamMap()
```

Summary

Convert the dataset to a Dataframe with:

```
rdd.map(lambda x: (x["newlabel"], DenseVector(x["features"])))
```

```
sqlContext.createDataFrame(input_data, ["label", "features"])
```

Note that the label’s column name is newlabel and all the features are gather in features. Change these values if different in your dataset.

Create the train/test set

```
randomSplit([.8,.2],seed=1234)
```

Train the model

```
LogisticRegression(labelCol="label",featuresCol="features",maxIter=10, regParam=0.3)
```

```
lr.fit()
```

Make prediction

```
linearModel.transform()
```