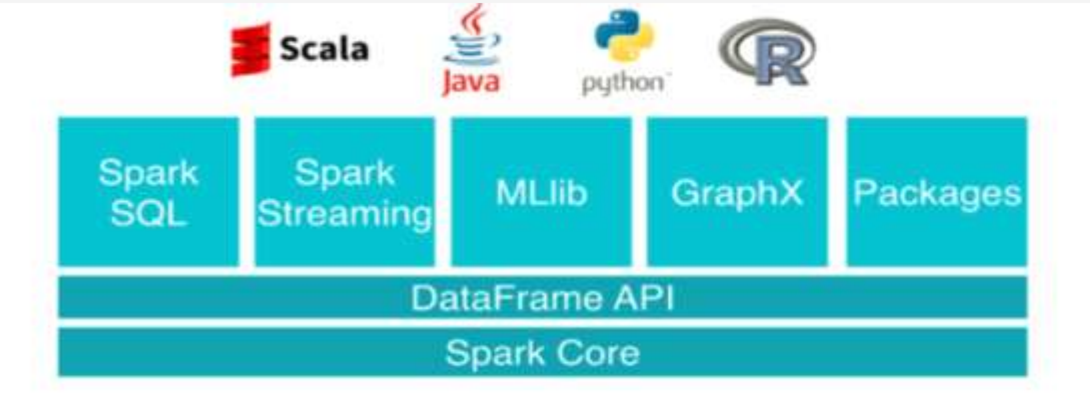
Communauté Big Data

meritis

Octobre 2019

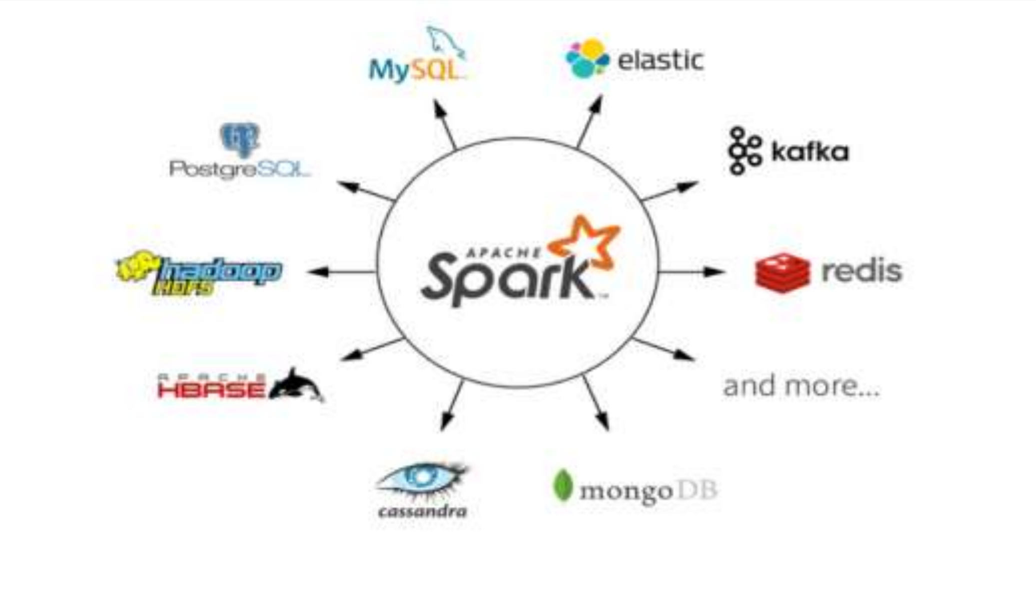
Abdelwahab TOUIL : Cloud&Data engineer

APACHE SPARK ?



AMPlab 2013
Api Scala,java…
Current version 2.4

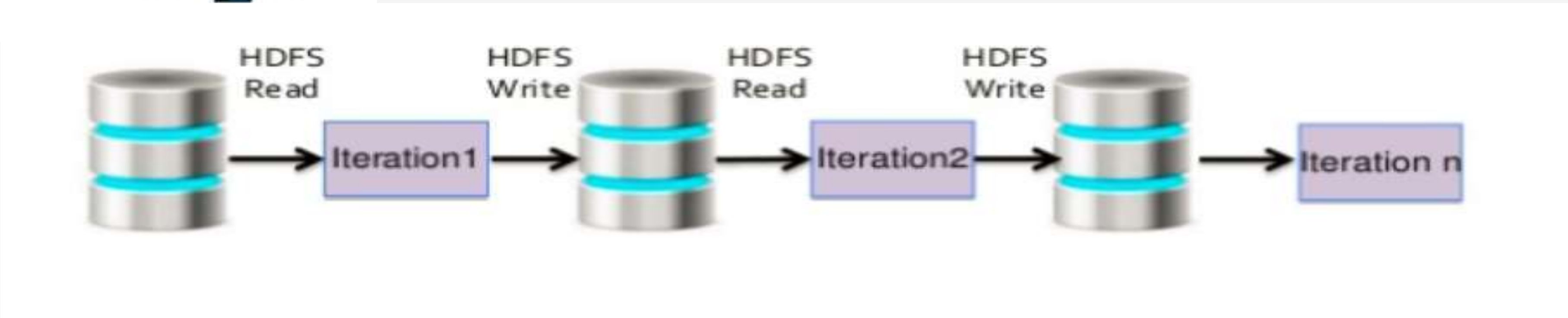
Severals data sources
On premise or Cloud



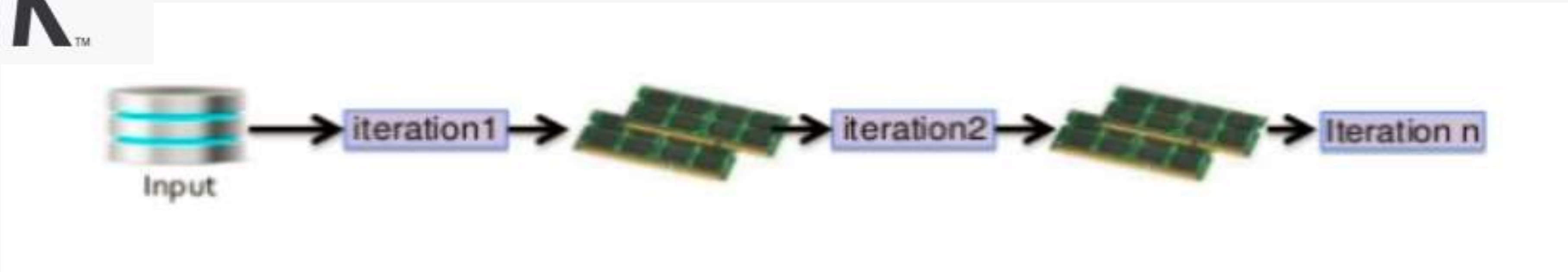
meritis

WHY APACHE SPARK

MapReduce with Hadoop

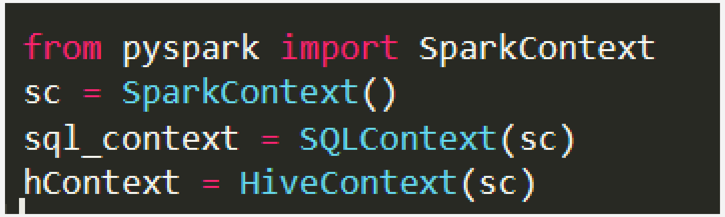


Processing with Spark

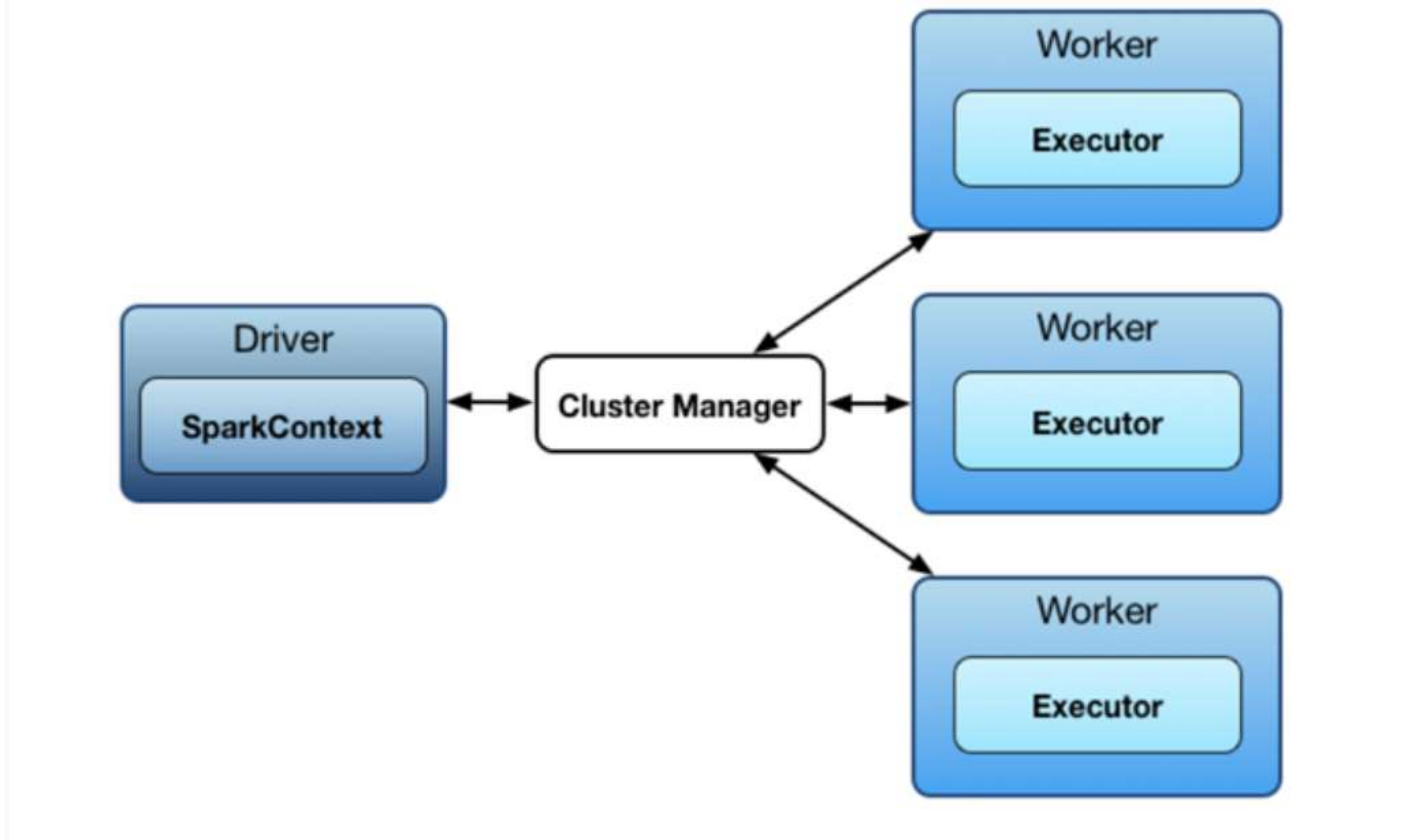


WHY APACHE SPARK

1. **master node**
2. **Driver program**
3. ***Cluster manager (work with Sc)***
4. ***Spark Context (inside driver/conf)***
5. ***Tasks are executed on workers***

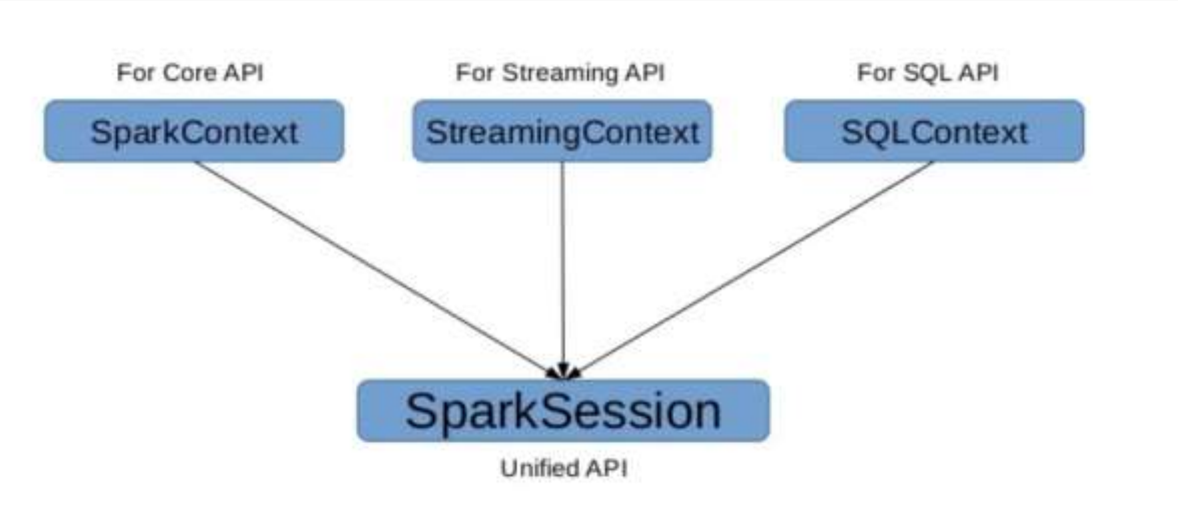


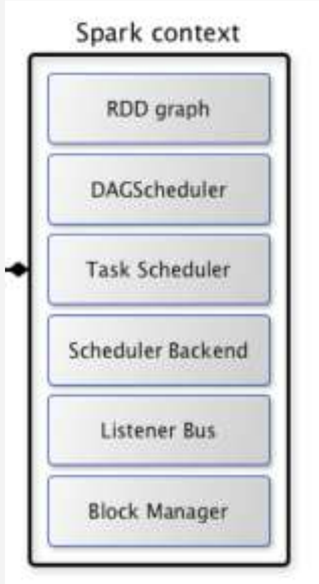
```
from pyspark import SparkContext
sc = SparkContext()
sql_context = SQLContext(sc)
hContext = HiveContext(sc)
```

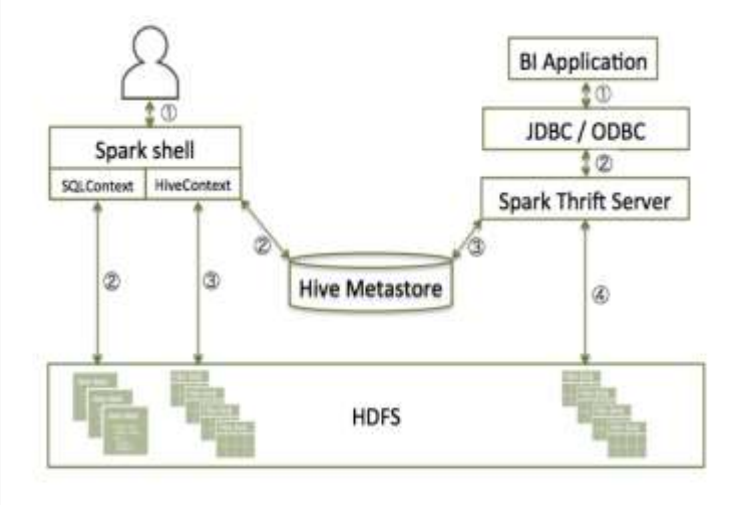


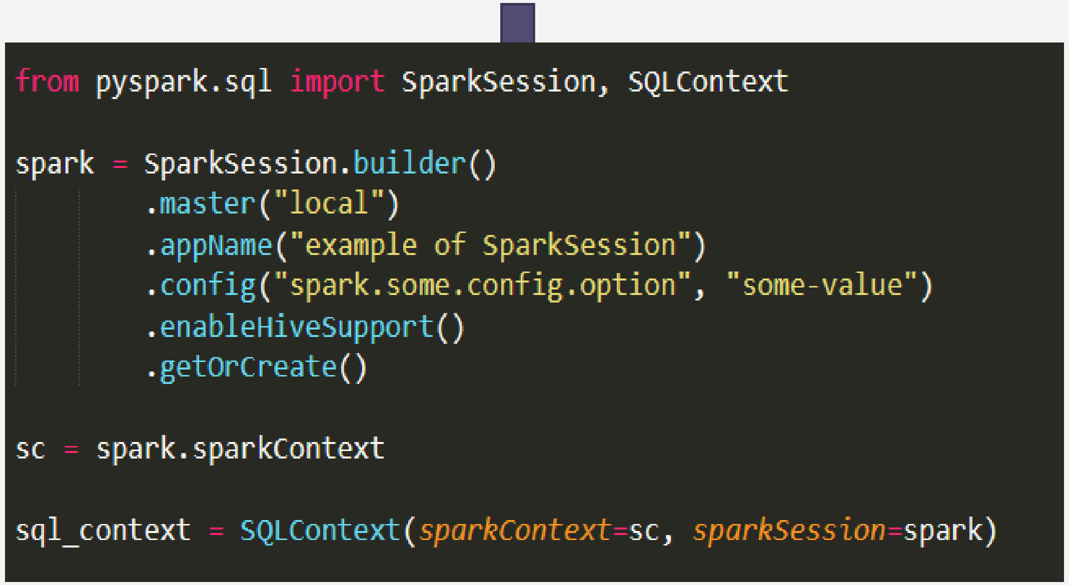
WHY APACHE SPARK

SQLContext, Apache Spark SQL can read data directly from the file system.
HiveContext, Spark SQL can also read data by interacting with the Hive MetaStore.



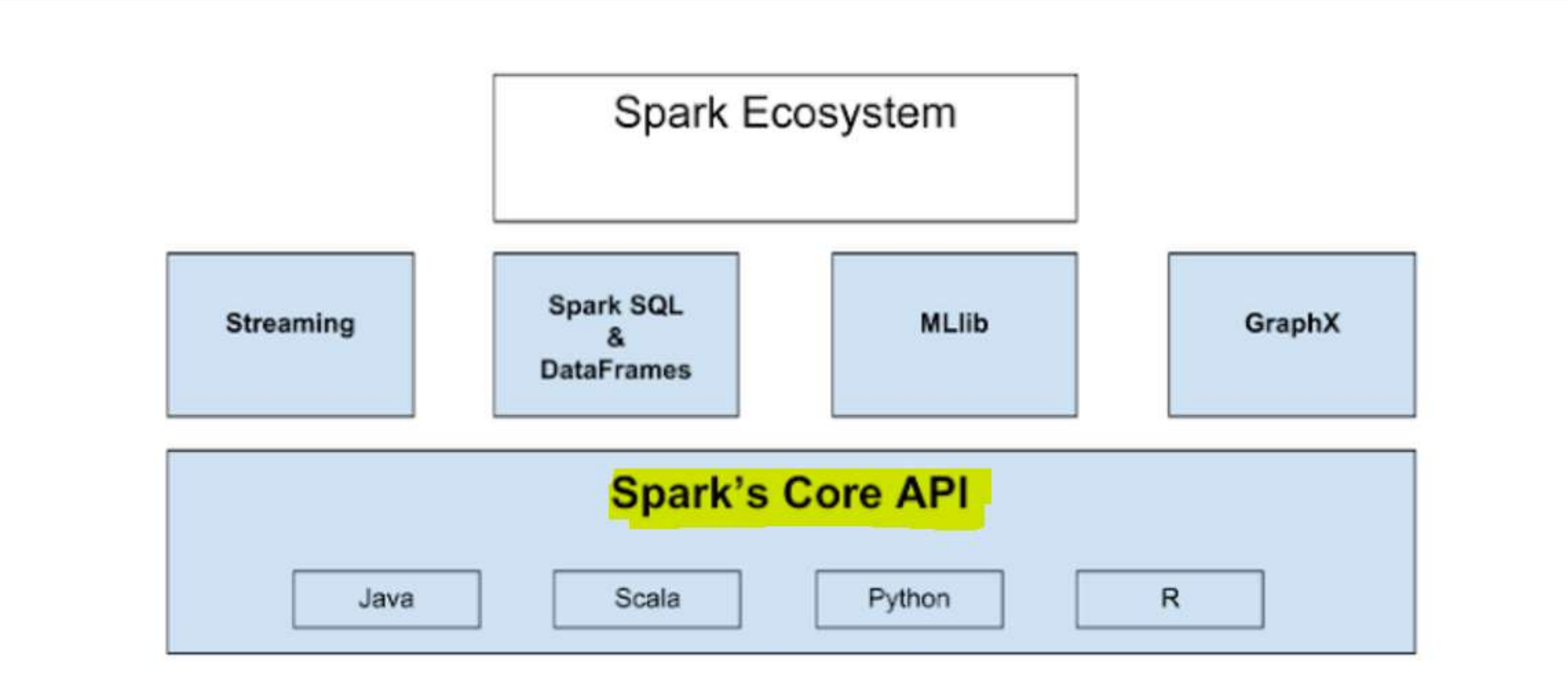




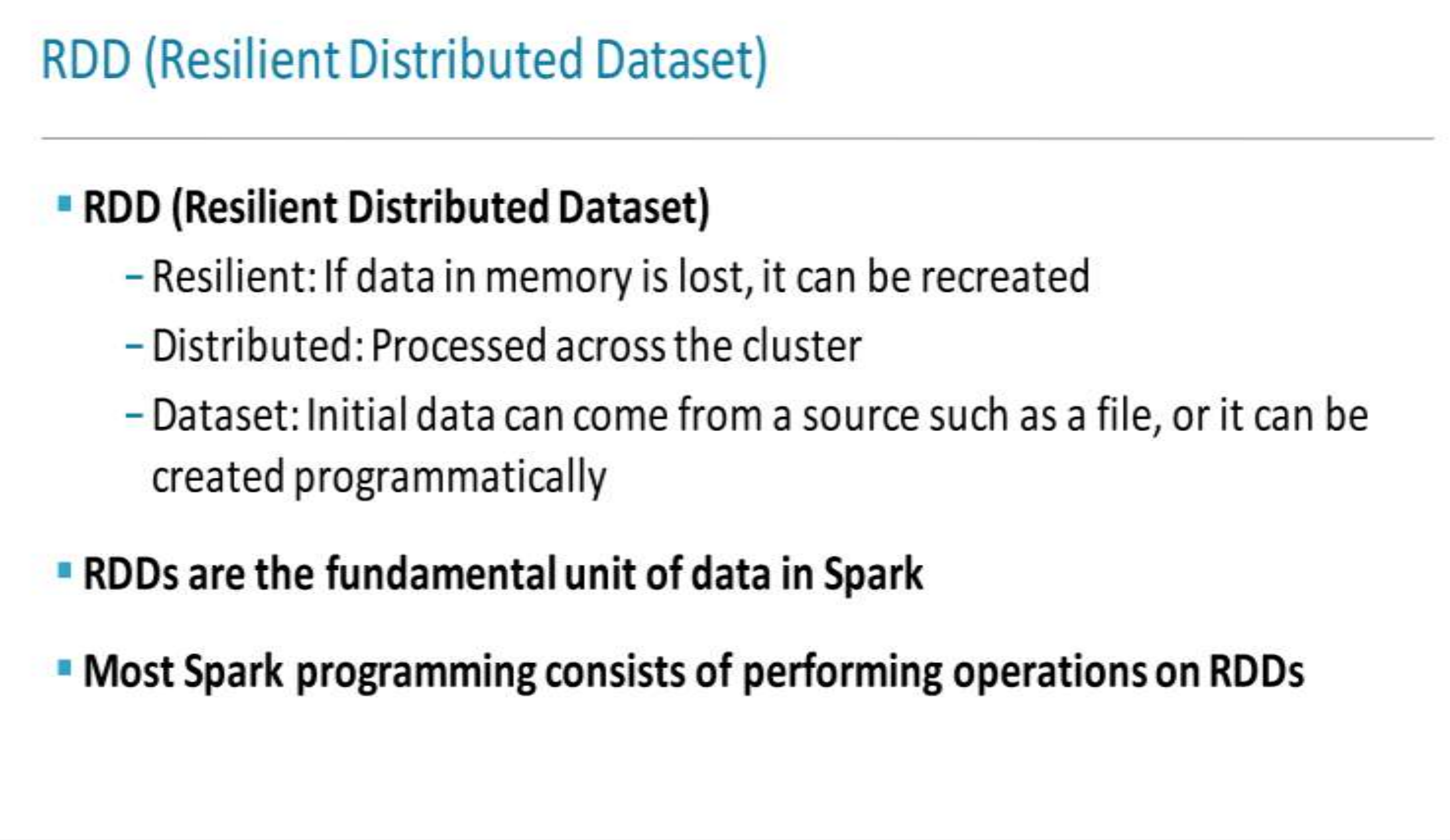


```
from pyspark.sql import SparkSession, SQLContext

spark = SparkSession.builder()
        .master("local")
        .appName("example of SparkSession")
        .config("spark.some.config.option", "some-value")
        .enableHiveSupport()
        .getOrCreate()

sc = spark.sparkContext

sql_context = SQLContext(sparkContext=sc, sparkSession=spark)
```

SPARK CORE

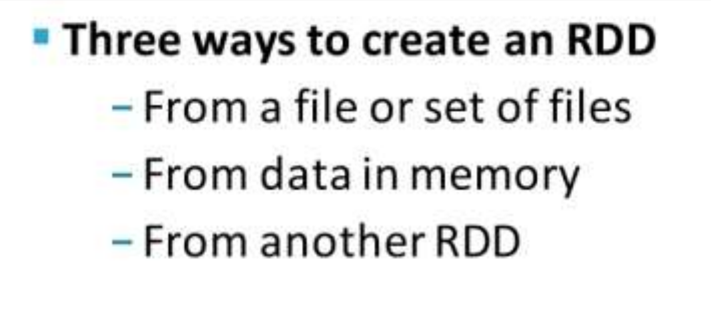

Spark Ecosystem
Streaming
Spark SQL & DataFrames
MLlib
GraphX
Spark's Core API
Java
Scala
Python
R

CDH – Spark – Rdd



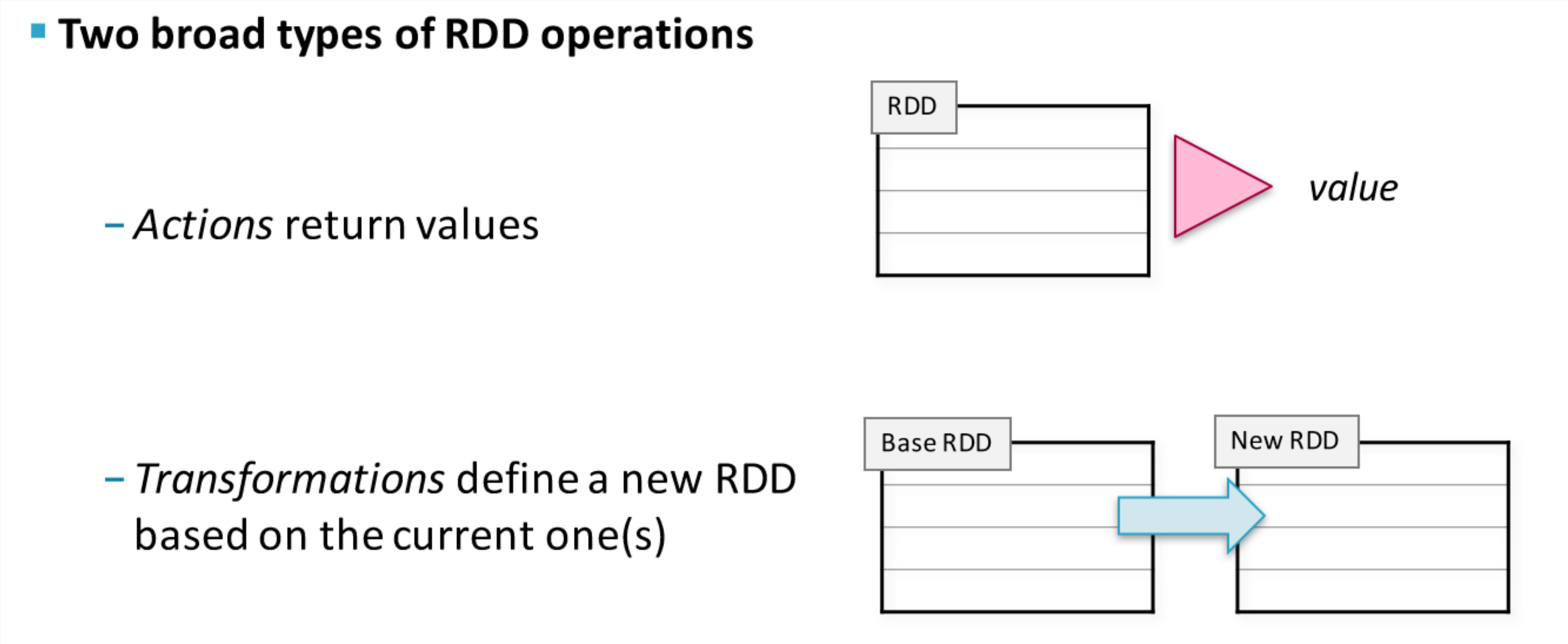
RDD (Resilient Distributed Dataset)

- **RDD (Resilient Distributed Dataset)**
  - Resilient: If data in memory is lost, it can be recreated
  - Distributed: Processed across the cluster
  - Dataset: Initial data can come from a source such as a file, or it can be created programmatically
- **RDDs are the fundamental unit of data in Spark**
- **Most Spark programming consists of performing operations on RDDs**



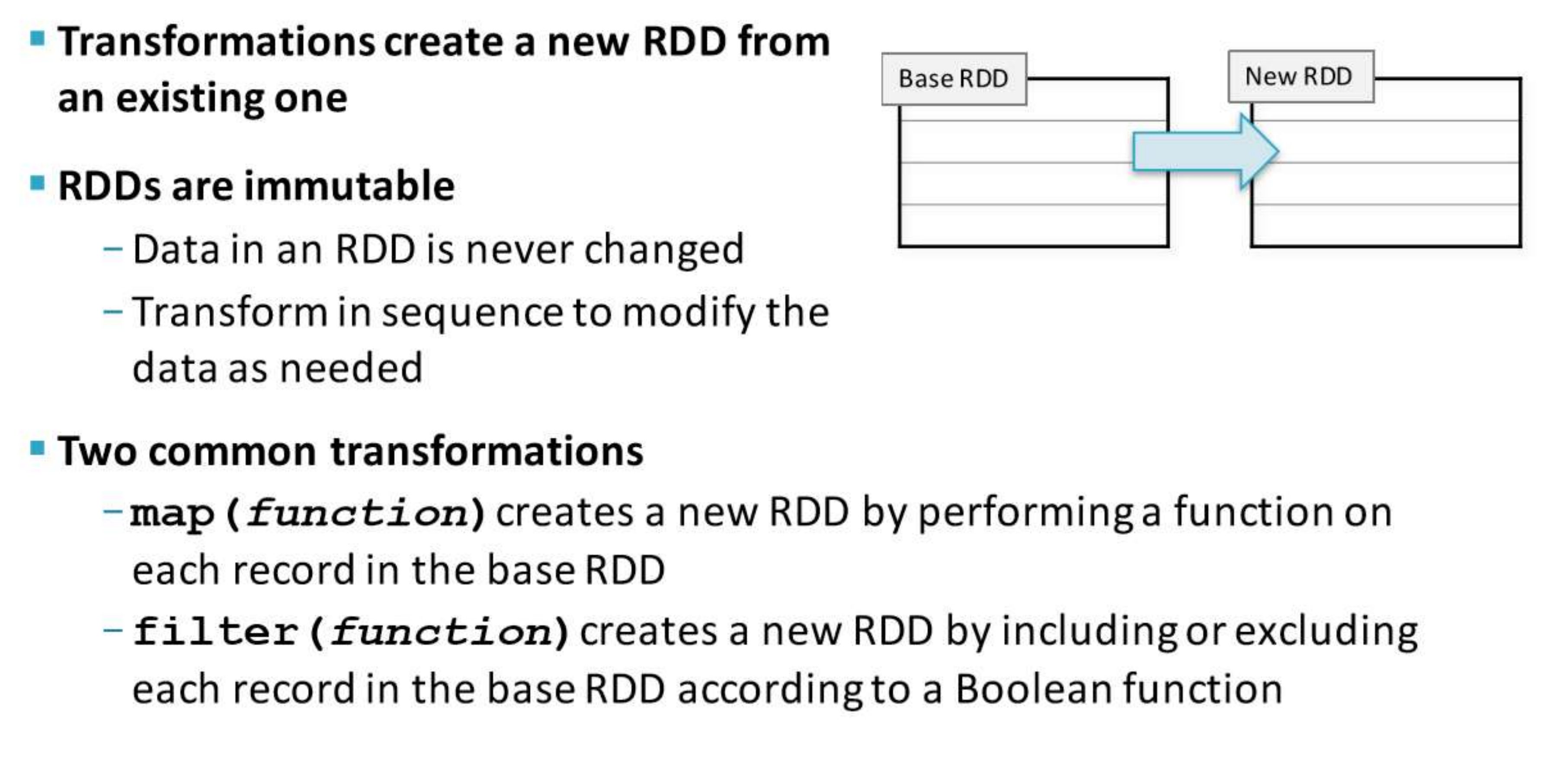
- **Three ways to create an RDD**
  - From a file or set of files
  - From data in memory
  - From another RDD

CDH – Spark – Rdd



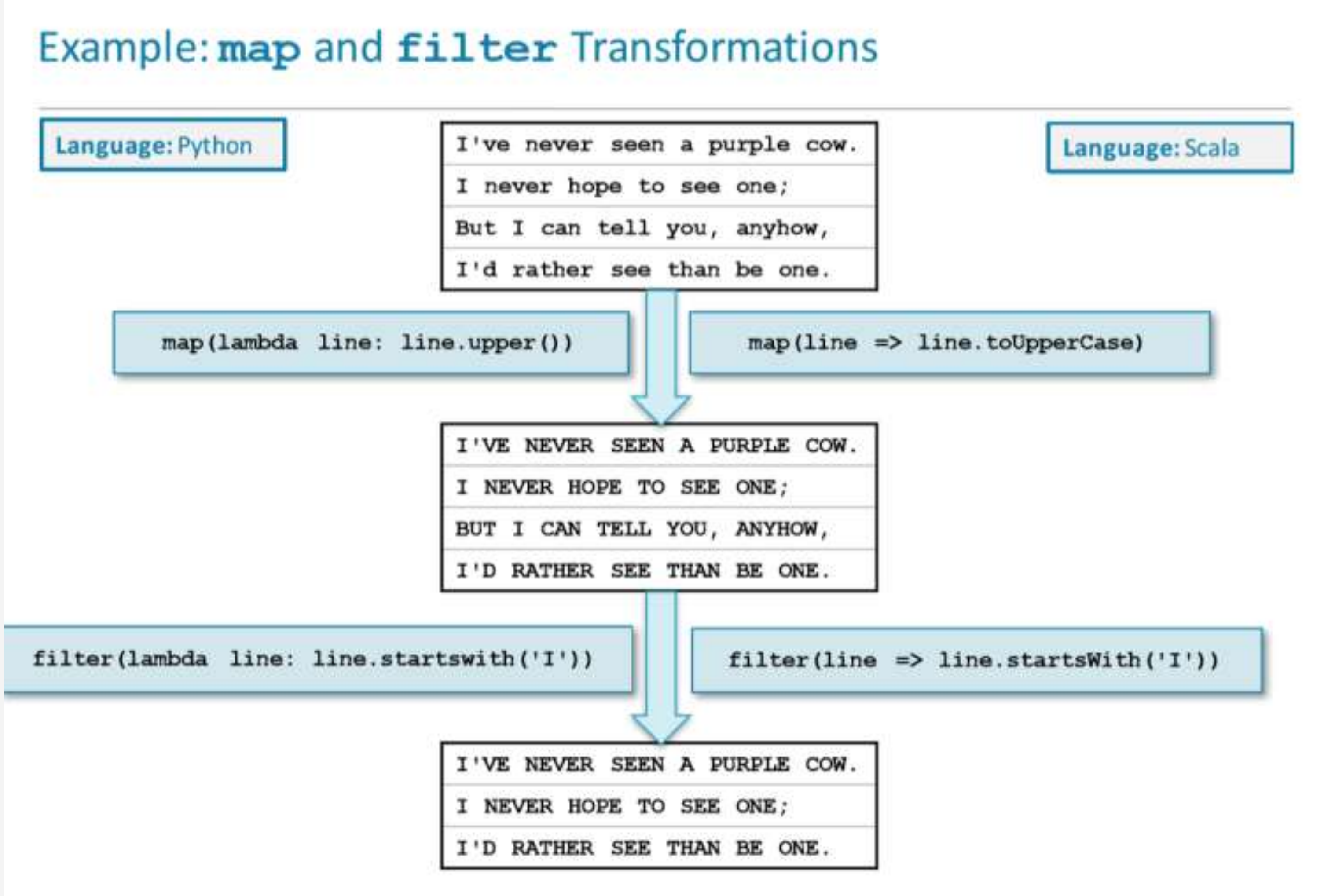
- **Two broad types of RDD operations**
  - *Actions* return values
  - *Transformations* define a new RDD based on the current one(s)

CDH – Spark – Rdd



- **Transformations create a new RDD from an existing one**
- **RDDs are immutable**
  - Data in an RDD is never changed
  - Transform in sequence to modify the data as needed
- **Two common transformations**
  - **`map(`*`function`*`)`** creates a new RDD by performing a function on each record in the base RDD
  - **`filter(`*`function`*`)`** creates a new RDD by including or excluding each record in the base RDD according to a Boolean function

CDH – Spark – Rdd



Example: **map** and **filter** Transformations

Language: Python

Language: Scala

```
I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
```

```
map(lambda line: line.upper())
```

```
map(line => line.toUpperCase)
```

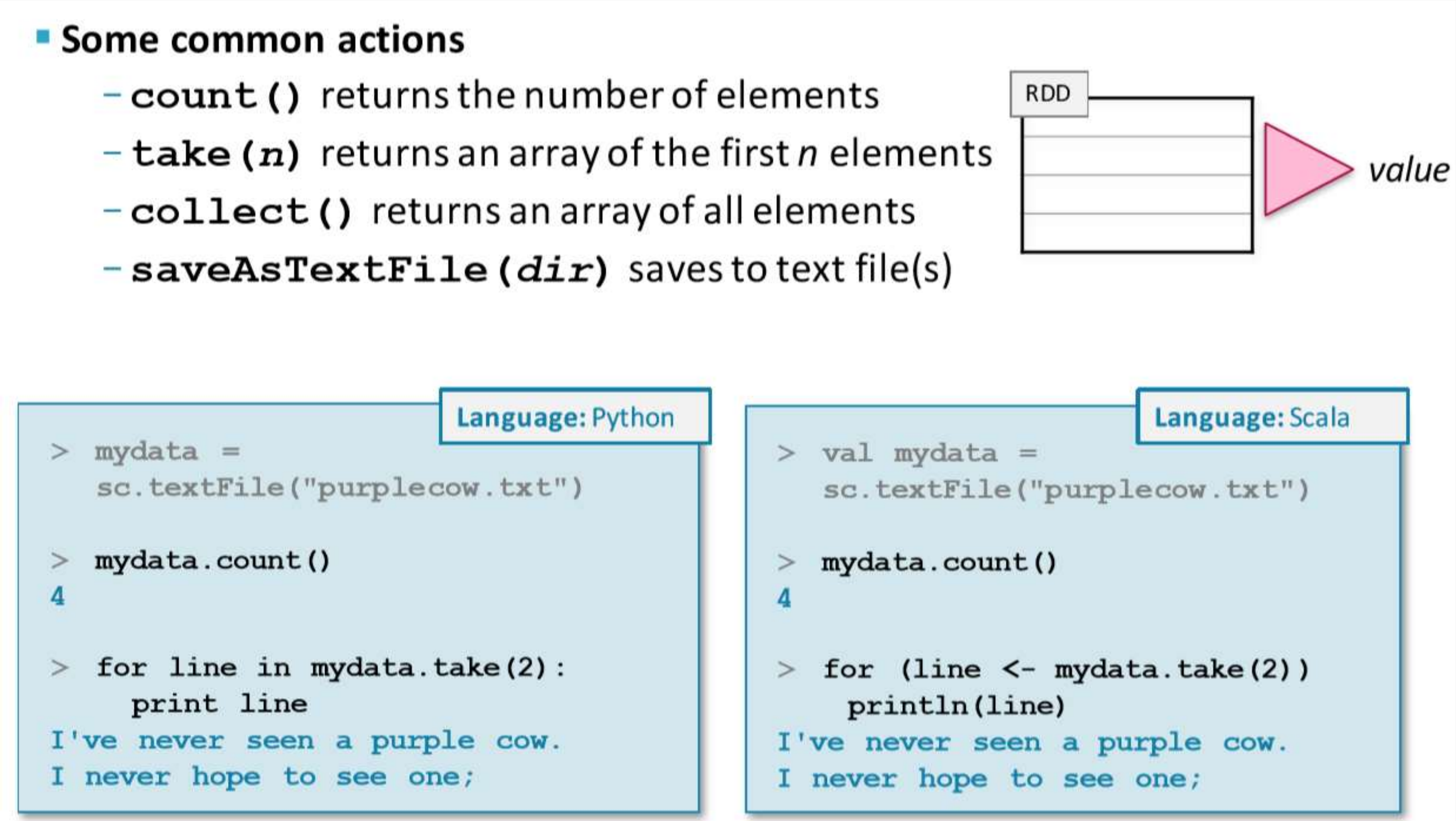
```
I'VE NEVER SEEN A PURPLE COW.
I NEVER HOPE TO SEE ONE;
BUT I CAN TELL YOU, ANYHOW,
I'D RATHER SEE THAN BE ONE.
```

```
filter(lambda line: line.startswith('I'))
```

```
filter(line => line.startsWith('I'))
```

```
I'VE NEVER SEEN A PURPLE COW.
I NEVER HOPE TO SEE ONE;
I'D RATHER SEE THAN BE ONE.
```

CDH – Spark – Rdd



- **Some common actions**
  - **`count()`** returns the number of elements
  - **`take(`*`n`*`)`** returns an array of the first *n* elements
  - **`collect()`** returns an array of all elements
  - **`saveAsTextFile(`*`dir`*`)`** saves to text file(s)

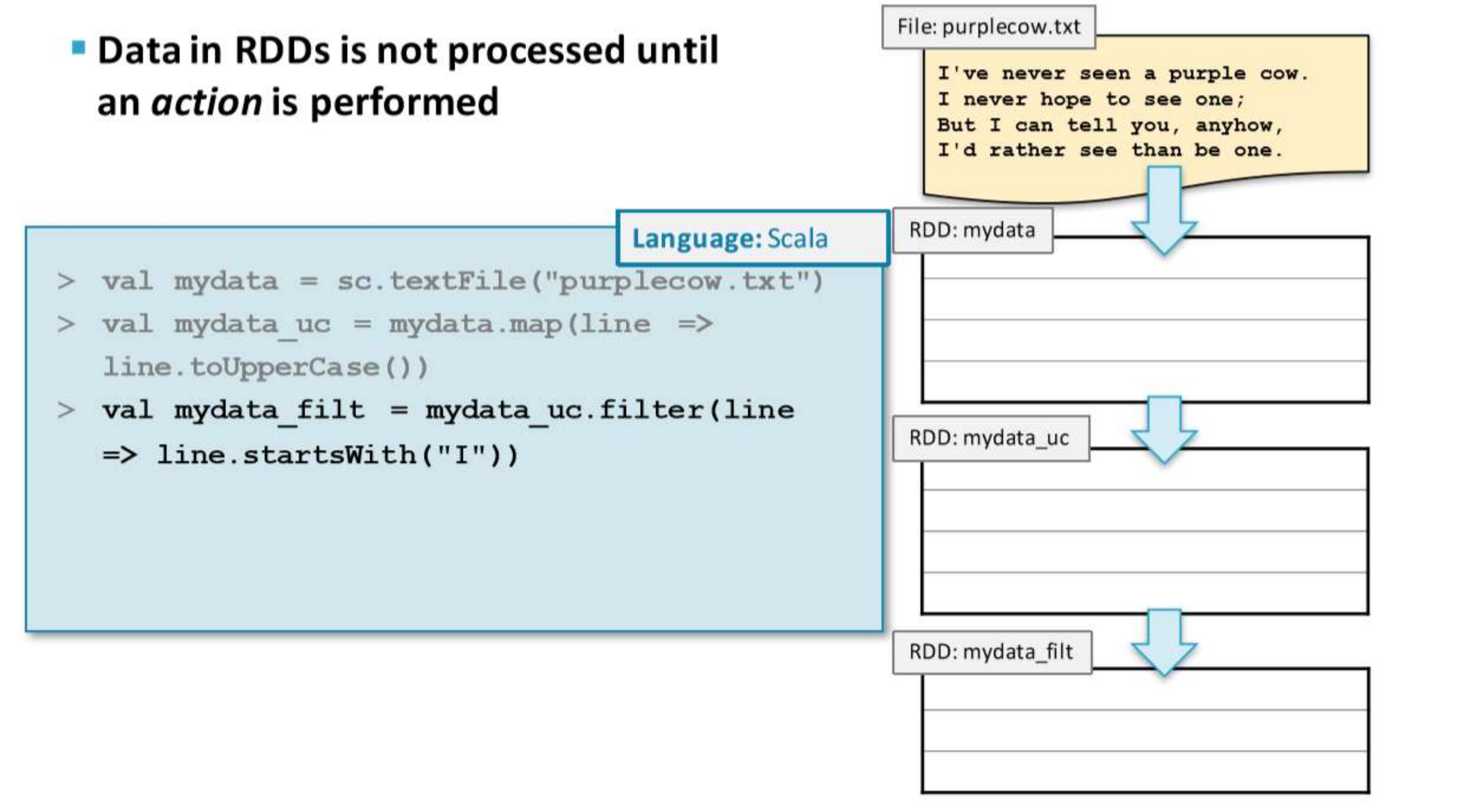
Language: Python

```
> mydata =
  sc.textFile("purplecow.txt")

> mydata.count()
4

> for line in mydata.take(2):
    print line
I've never seen a purple cow.
I never hope to see one;
```

Language: Scala

```
> val mydata =
  sc.textFile("purplecow.txt")

> mydata.count()
4

> for (line <- mydata.take(2))
    println(line)
I've never seen a purple cow.
I never hope to see one;
```

CDH – Spark – Rdd – Lazy



- **Data in RDDs is not processed until an *action* is performed**

Language: Scala

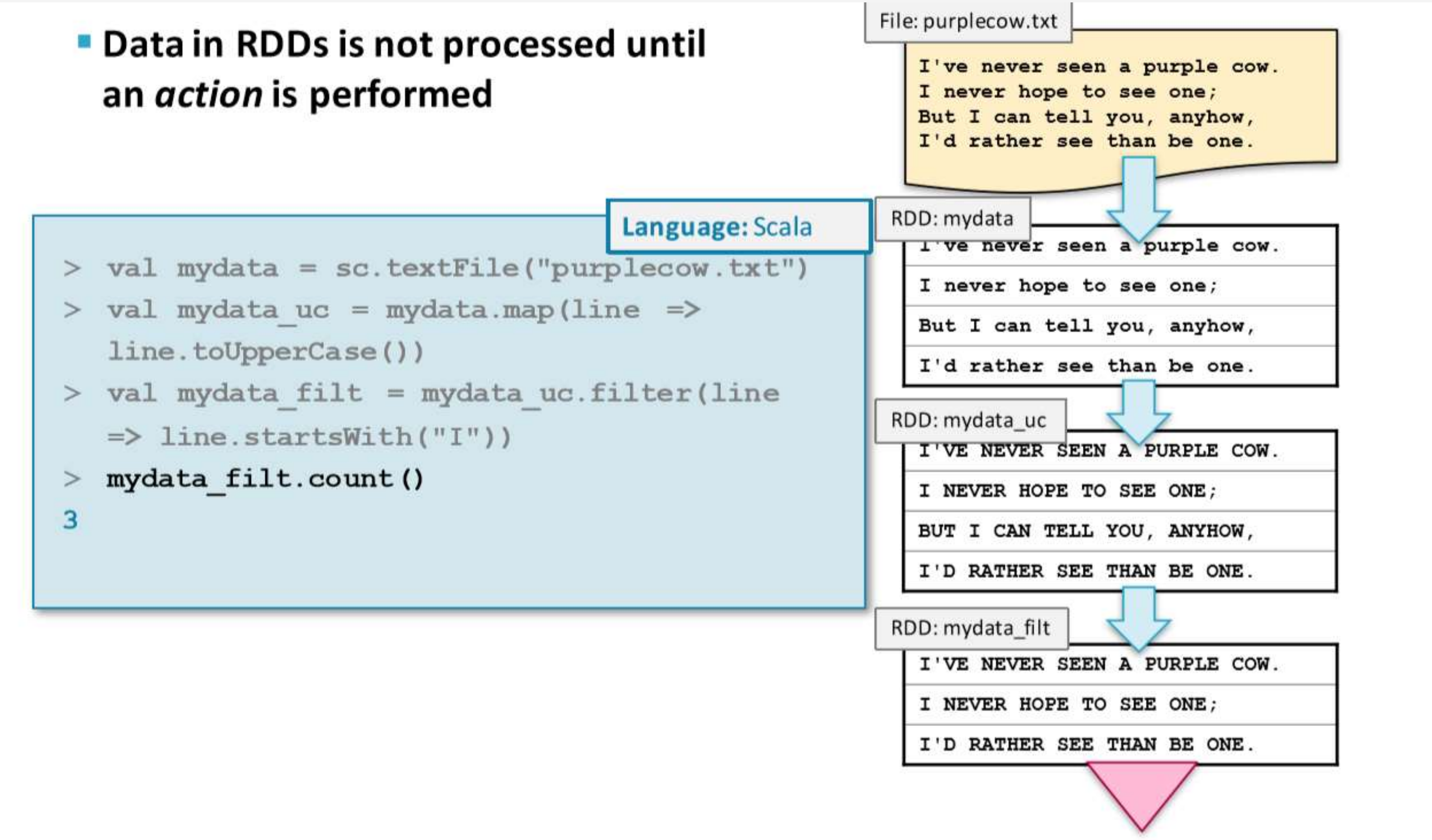
```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
```

File: purplecow.txt

```
I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
```

RDD: mydata
RDD: mydata_uc
RDD: mydata_filt

CDH – Spark – Rdd – Lazy



- Data in RDDs is not processed until an *action* is performed

Language: Scala

```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
> mydata_filt.count()
3
```

File: purplecow.txt

```
I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
```

RDD: mydata

```
I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
```

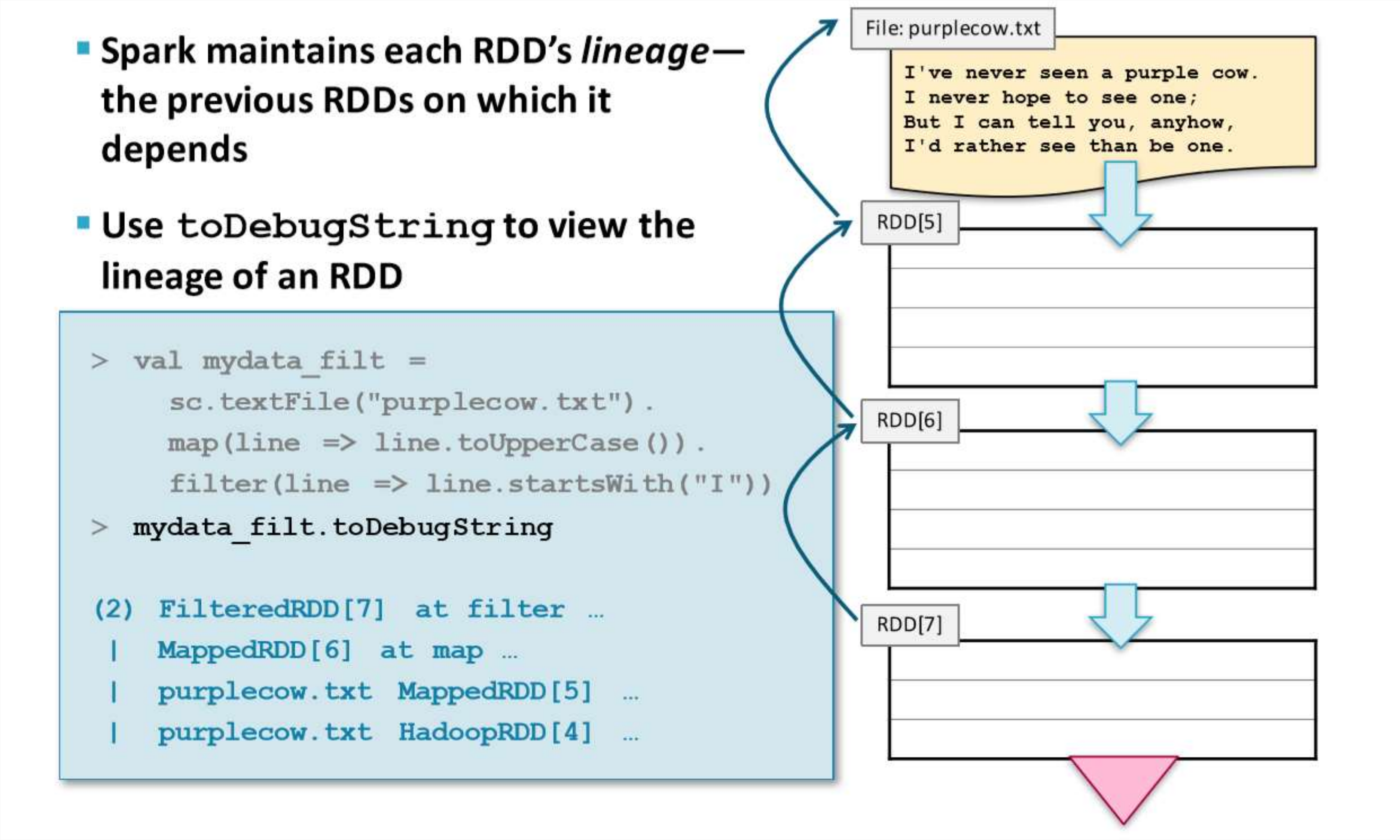
RDD: mydata_uc

```
I'VE NEVER SEEN A PURPLE COW.
I NEVER HOPE TO SEE ONE;
BUT I CAN TELL YOU, ANYHOW,
I'D RATHER SEE THAN BE ONE.
```

RDD: mydata_filt

```
I'VE NEVER SEEN A PURPLE COW.
I NEVER HOPE TO SEE ONE;
I'D RATHER SEE THAN BE ONE.
```

CDH – Spark – Rdd – Lineage



- Spark maintains each RDD's *lineage*—the previous RDDs on which it depends
- Use `toDebugString` to view the lineage of an RDD

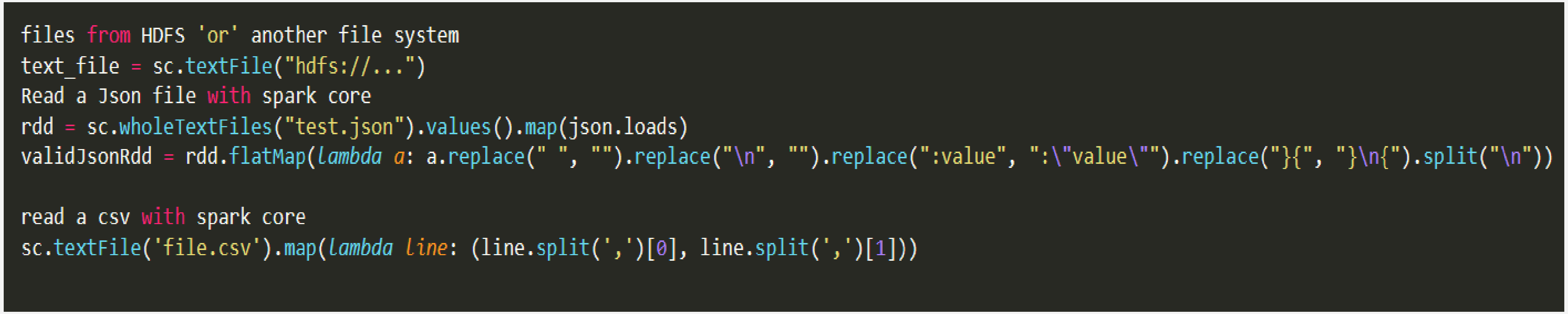
```
> val mydata_filt =
    sc.textFile("purplecow.txt").
    map(line => line.toUpperCase()).
    filter(line => line.startsWith("I"))
> mydata_filt.toDebugString

(2) FilteredRDD[7] at filter …
 |  MappedRDD[6] at map …
 |  purplecow.txt MappedRDD[5] …
 |  purplecow.txt HadoopRDD[4] …
```

File: purplecow.txt

```
I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
```

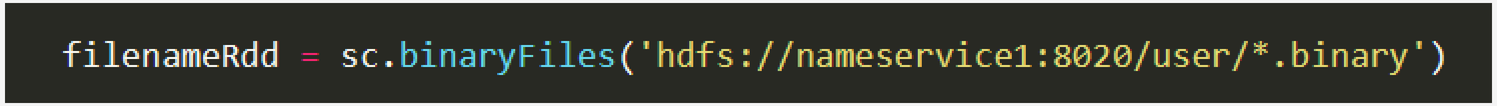
CDH – Spark – Rdd – Read data

TextFile & wholeTextFiles for text file format

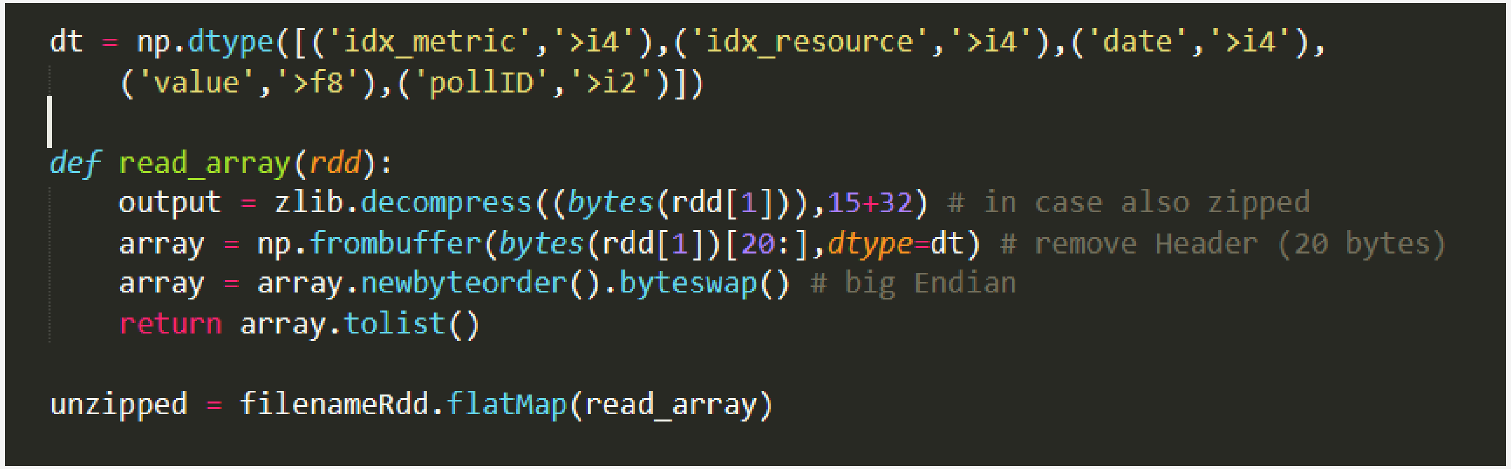


```
files from HDFS 'or' another file system
text_file = sc.textFile("hdfs://...")
Read a Json file with spark core
rdd = sc.wholeTextFiles("test.json").values().map(json.loads)
validJsonRdd = rdd.flatMap(lambda a: a.replace(" ", "").replace("\n", "").replace(":value", ":\"value\"").replace("}{", "}\n{").split("\n"))

read a csv with spark core
sc.textFile('file.csv').map(lambda line: (line.split(',')[0], line.split(',')[1]))
```

Binary files for binaryFiles



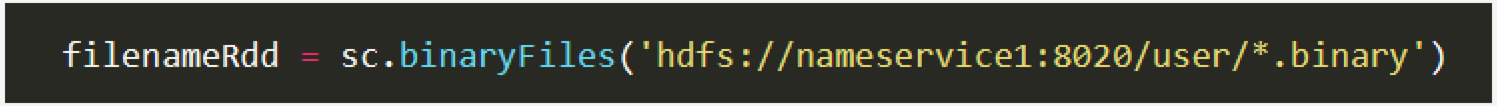
```
filenameRdd = sc.binaryFiles('hdfs://nameservice1:8020/user/*.binary')
```



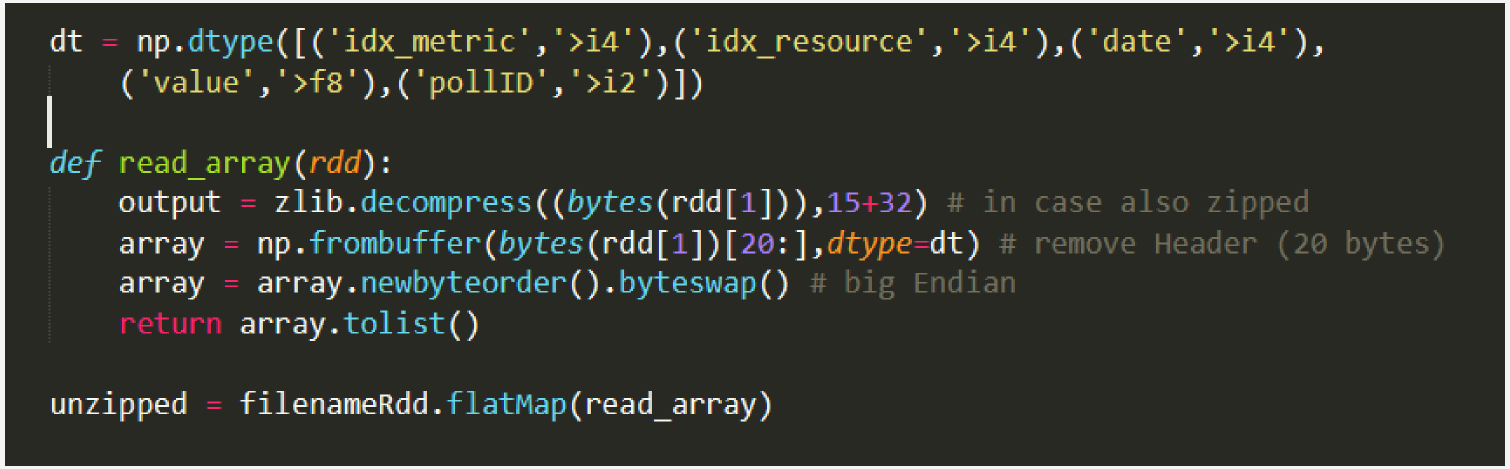
```
dt = np.dtype([('idx_metric','>i4'),('idx_resource','>i4'),('date','>i4'),
    ('value','>f8'),('pollID','>i2')])

def read_array(rdd):
    output = zlib.decompress((bytes(rdd[1])),15+32) # in case also zipped
    array = np.frombuffer(bytes(rdd[1])[20:],dtype=dt) # remove Header (20 bytes)
    array = array.newbyteorder().byteswap() # big Endian
    return array.tolist()

unzipped = filenameRdd.flatMap(read_array)
```

CDH – Spark – Rdd – Read data

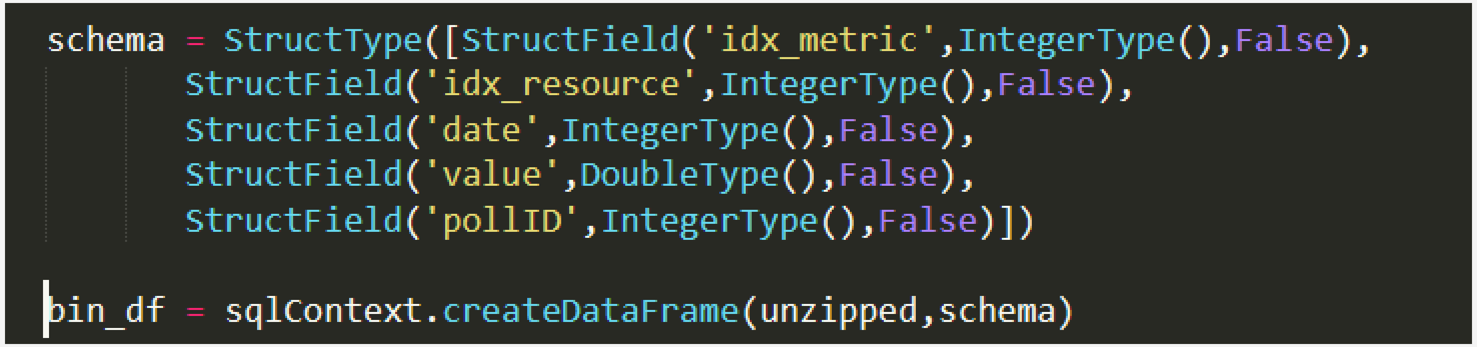
Binary files for binaryFiles



```
filenameRdd = sc.binaryFiles('hdfs://nameservice1:8020/user/*.binary')
```



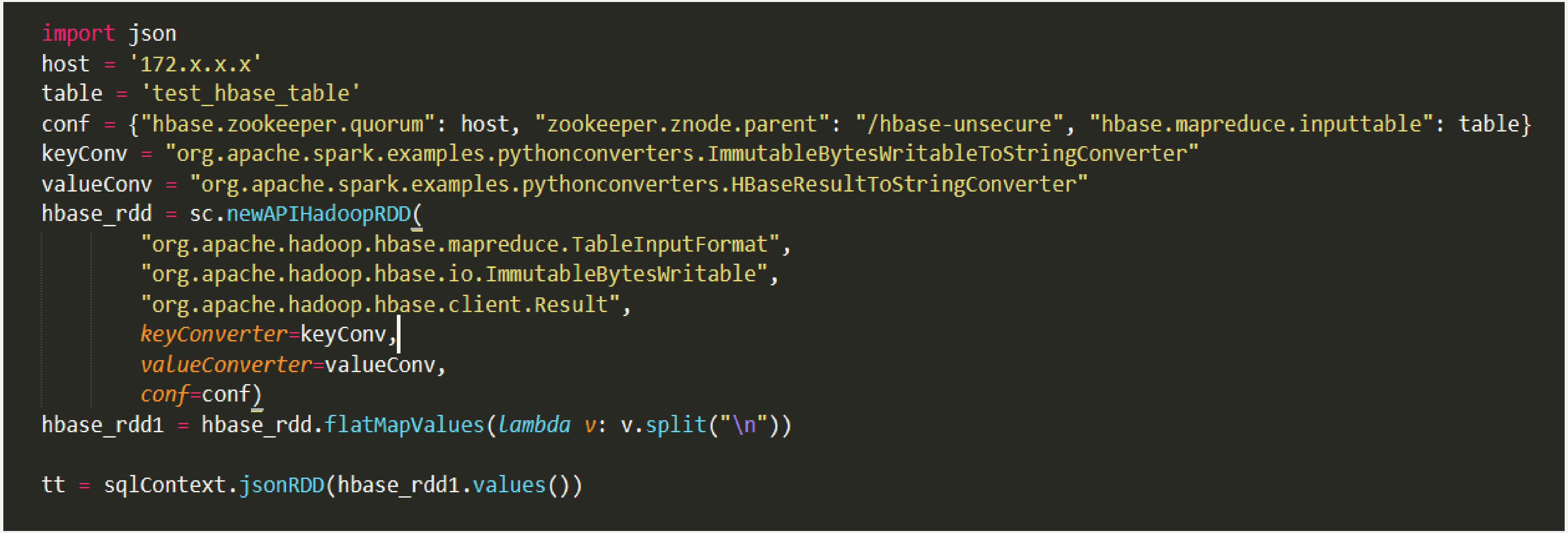
```
dt = np.dtype([('idx_metric','>i4'),('idx_resource','>i4'),('date','>i4'),
    ('value','>f8'),('pollID','>i2')])

def read_array(rdd):
    output = zlib.decompress((bytes(rdd[1])),15+32) # in case also zipped
    array = np.frombuffer(bytes(rdd[1])[20:],dtype=dt) # remove Header (20 bytes)
    array = array.newbyteorder().byteswap() # big Endian
    return array.tolist()

unzipped = filenameRdd.flatMap(read_array)
```



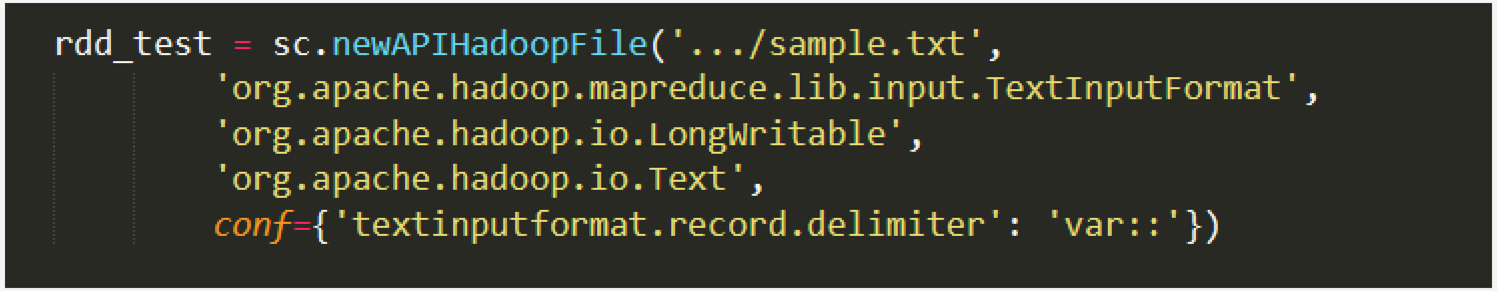
```
schema = StructType([StructField('idx_metric',IntegerType(),False),
        StructField('idx_resource',IntegerType(),False),
        StructField('date',IntegerType(),False),
        StructField('value',DoubleType(),False),
        StructField('pollID',IntegerType(),False)])

bin_df = sqlContext.createDataFrame(unzipped,schema)
```

CDH – Spark – Rdd – Read data

newAPIHadoopRDD & newAPIHadoopFile for text file format



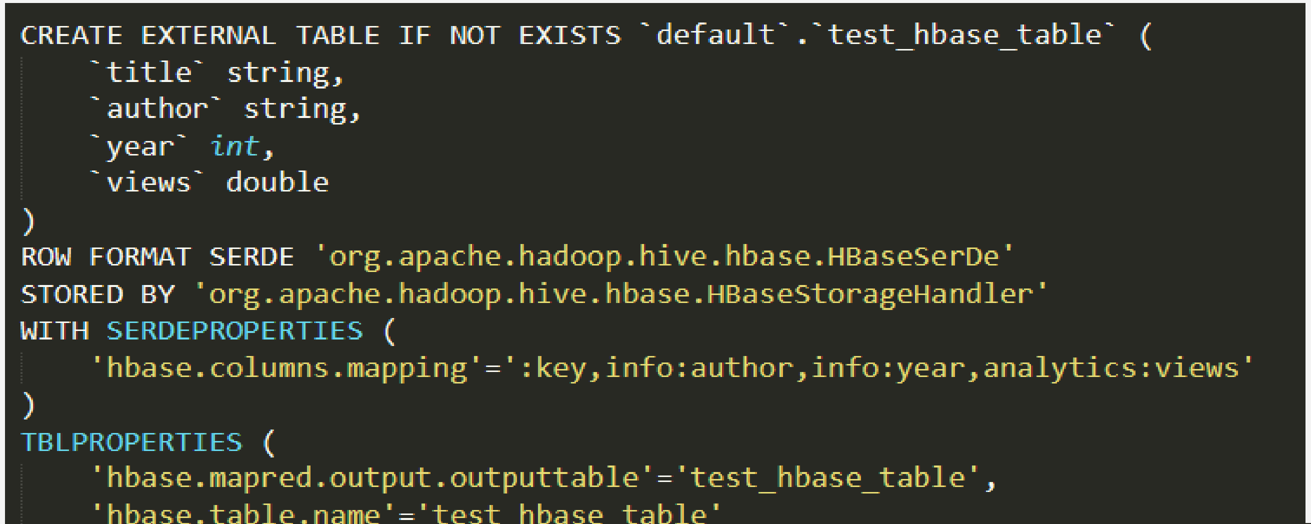
```
import json
host = '172.x.x.x'
table = 'test_hbase_table'
conf = {"hbase.zookeeper.quorum": host, "zookeeper.znode.parent": "/hbase-unsecure", "hbase.mapreduce.inputtable": table}
keyConv = "org.apache.spark.examples.pythonconverters.ImmutableBytesWritableToStringConverter"
valueConv = "org.apache.spark.examples.pythonconverters.HBaseResultToStringConverter"
hbase_rdd = sc.newAPIHadoopRDD(
        "org.apache.hadoop.hbase.mapreduce.TableInputFormat",
        "org.apache.hadoop.hbase.io.ImmutableBytesWritable",
        "org.apache.hadoop.hbase.client.Result",
        keyConverter=keyConv,
        valueConverter=valueConv,
        conf=conf)
hbase_rdd1 = hbase_rdd.flatMapValues(lambda v: v.split("\n"))

tt = sqlContext.jsonRDD(hbase_rdd1.values())
```



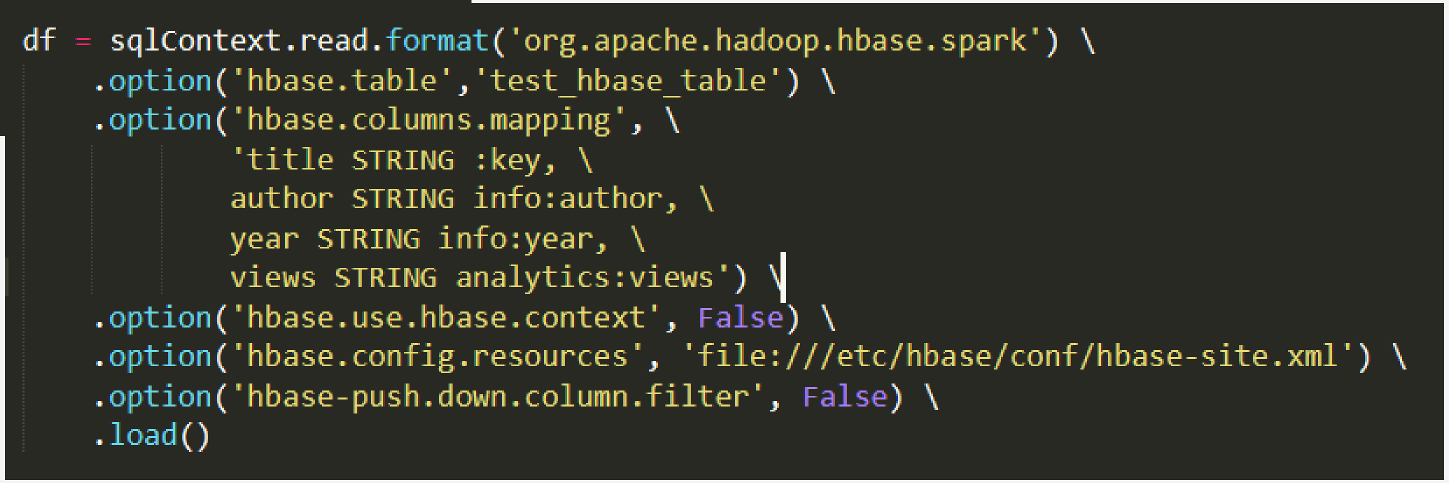
```
rdd_test = sc.newAPIHadoopFile('.../sample.txt',
        'org.apache.hadoop.mapreduce.lib.input.TextInputFormat',
        'org.apache.hadoop.io.LongWritable',
        'org.apache.hadoop.io.Text',
        conf={'textinputformat.record.delimiter': 'var::'})
```

CDH – Spark – Rdd – Read data

Hbase with Spark DataFrame

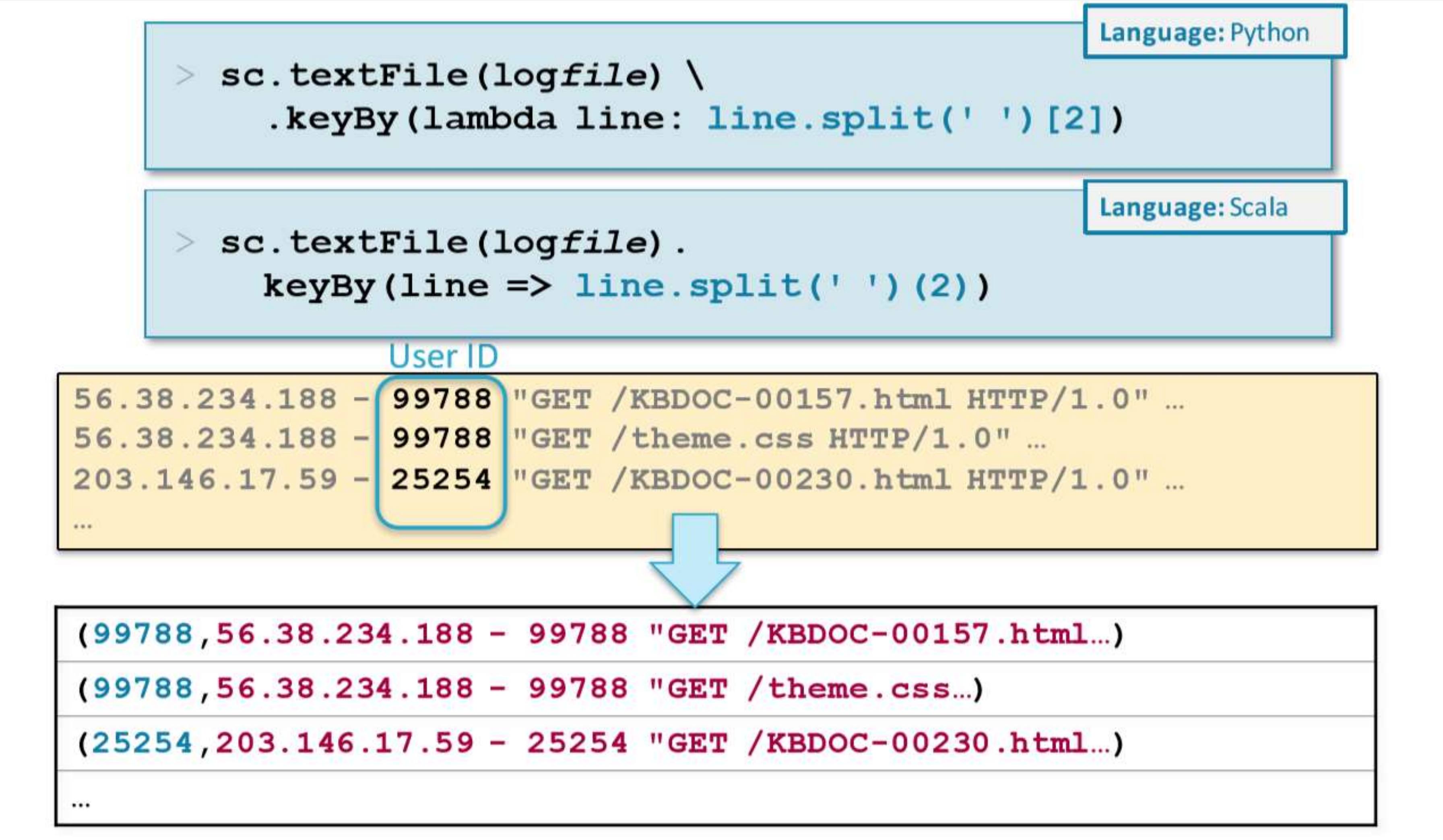


```
CREATE EXTERNAL TABLE IF NOT EXISTS `default`.`test_hbase_table` (
    `title` string,
    `author` string,
    `year` int,
    `views` double
)
ROW FORMAT SERDE 'org.apache.hadoop.hive.hbase.HBaseSerDe'
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES (
    'hbase.columns.mapping'=':key,info:author,info:year,analytics:views'
)
TBLPROPERTIES (
    'hbase.mapred.output.outputtable'='test_hbase_table',
    'hbase.table.name'='test_hbase_table'
);

sqlContext.table("default.test_hbase_table")
```



```
df = sqlContext.read.format('org.apache.hadoop.hbase.spark') \
    .option('hbase.table','test_hbase_table') \
    .option('hbase.columns.mapping', \
            'title STRING :key, \
            author STRING info:author, \
            year STRING info:year, \
            views STRING analytics:views') \
    .option('hbase.use.hbase.context', False) \
    .option('hbase.config.resources', 'file:///etc/hbase/conf/hbase-site.xml') \
    .option('hbase-push.down.column.filter', False) \
    .load()
```

CDH - Spark - Rdd - keyby



Language: Python

```
> sc.textFile(logfile) \
    .keyBy(lambda line: line.split(' ')[2])
```

Language: Scala

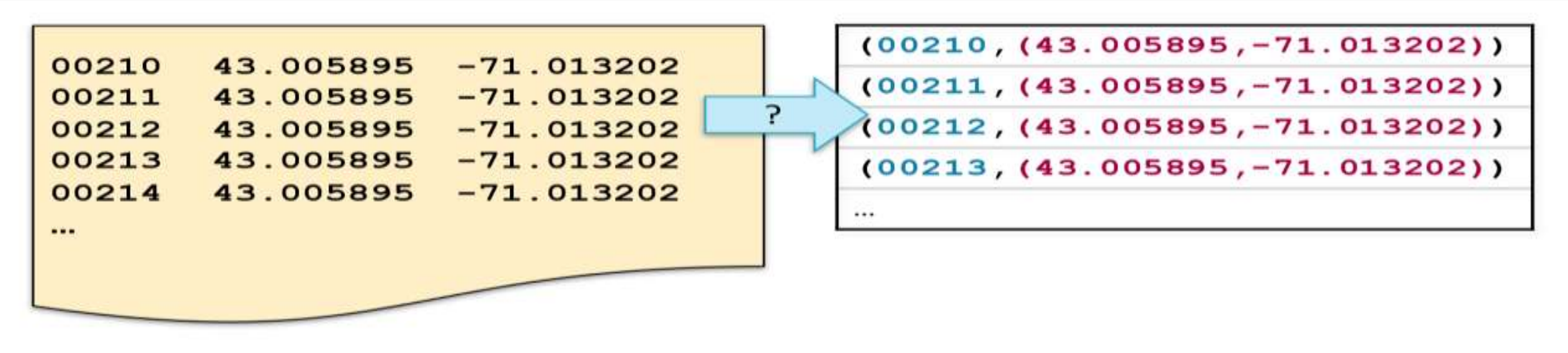
```
> sc.textFile(logfile).
    keyBy(line => line.split(' ')(2))
```

User ID

```
56.38.234.188 – 99788 "GET /KBDOC-00157.html HTTP/1.0" …
56.38.234.188 – 99788 "GET /theme.css HTTP/1.0" …
203.146.17.59 – 25254 "GET /KBDOC-00230.html HTTP/1.0" …
…
```

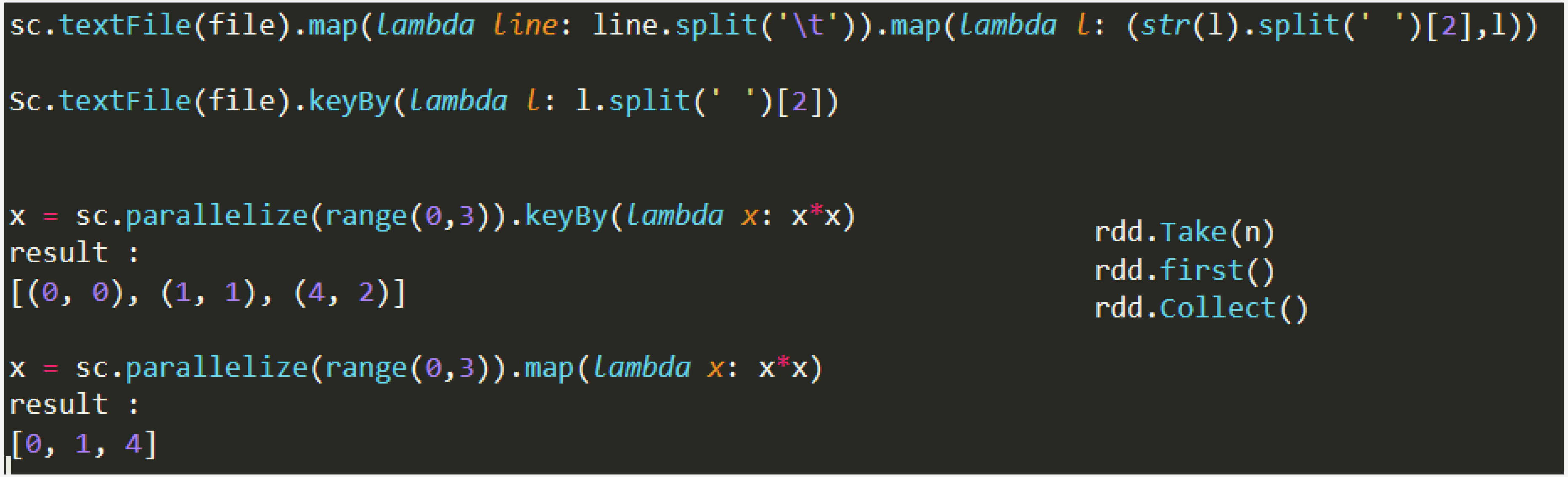
```
(99788,56.38.234.188 - 99788 "GET /KBDOC-00157.html…)
(99788,56.38.234.188 - 99788 "GET /theme.css…)
(25254,203.146.17.59 - 25254 "GET /KBDOC-00230.html…)
…
```

CDH - Spark - Rdd - keyby



```
00210   43.005895   -71.013202
00211   43.005895   -71.013202
00212   43.005895   -71.013202
00213   43.005895   -71.013202
00214   43.005895   -71.013202
...
```

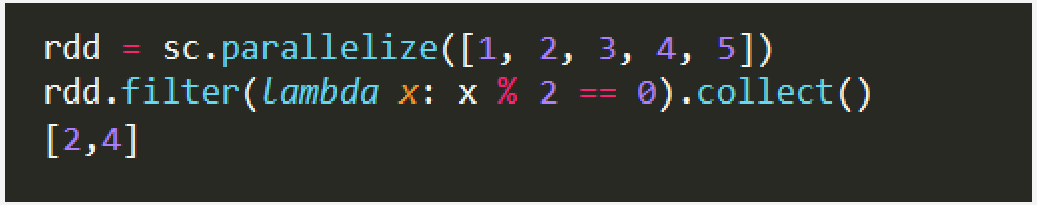
```
(00210,(43.005895,-71.013202))
(00211,(43.005895,-71.013202))
(00212,(43.005895,-71.013202))
(00213,(43.005895,-71.013202))
...
```



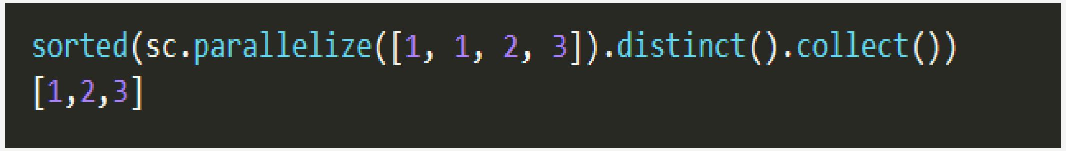
```
sc.textFile(file).map(lambda line: line.split('\t')).map(lambda l: (str(l).split(' ')[2],l))

Sc.textFile(file).keyBy(lambda l: l.split(' ')[2])


x = sc.parallelize(range(0,3)).keyBy(lambda x: x*x)
result :
[(0, 0), (1, 1), (4, 2)]

x = sc.parallelize(range(0,3)).map(lambda x: x*x)
result :
[0, 1, 4]
```

```
rdd.Take(n)
rdd.first()
rdd.Collect()
```

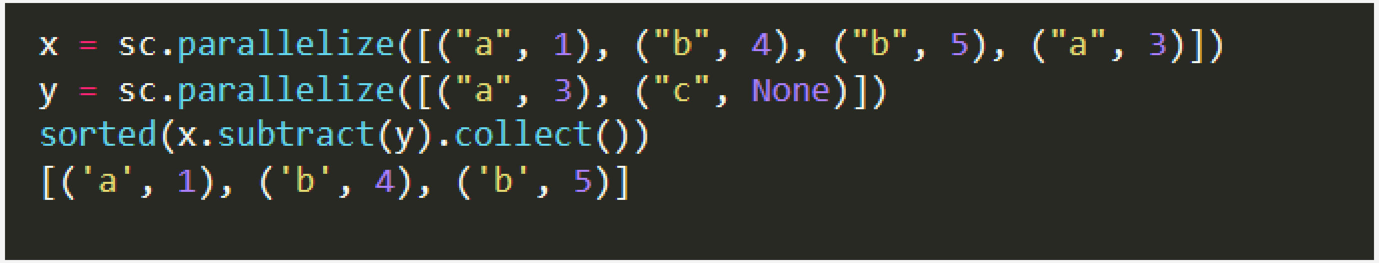
CDH – Spark – Rdd



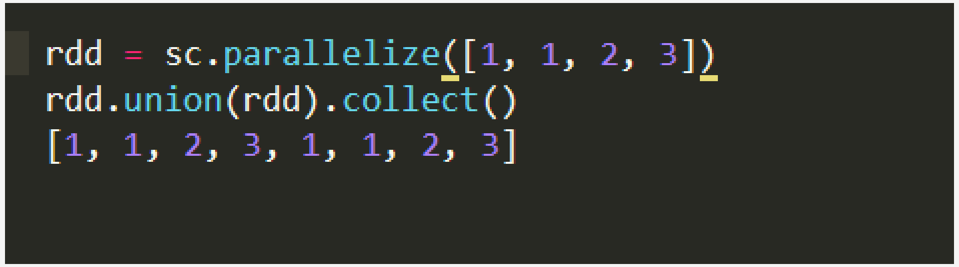
```
rdd = sc.parallelize([1, 2, 3, 4, 5])
rdd.filter(lambda x: x % 2 == 0).collect()
[2,4]
```



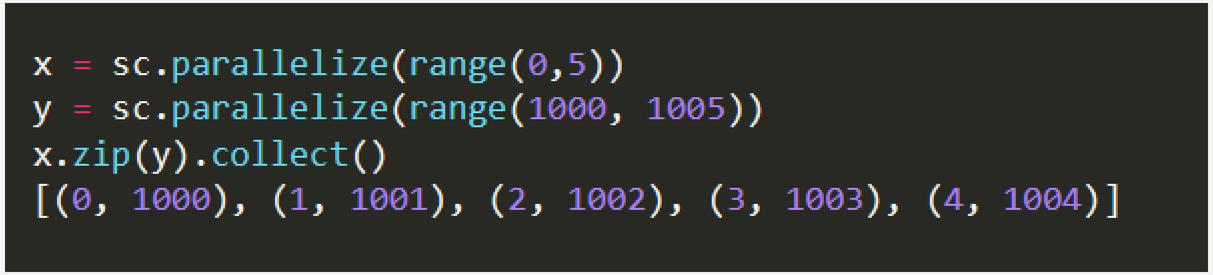
```
sorted(sc.parallelize([1, 1, 2, 3]).distinct().collect())
[1,2,3]
```



```
x = sc.parallelize([("a", 1), ("b", 4), ("b", 5), ("a", 3)])
y = sc.parallelize([("a", 3), ("c", None)])
sorted(x.subtract(y).collect())
[('a', 1), ('b', 4), ('b', 5)]
```

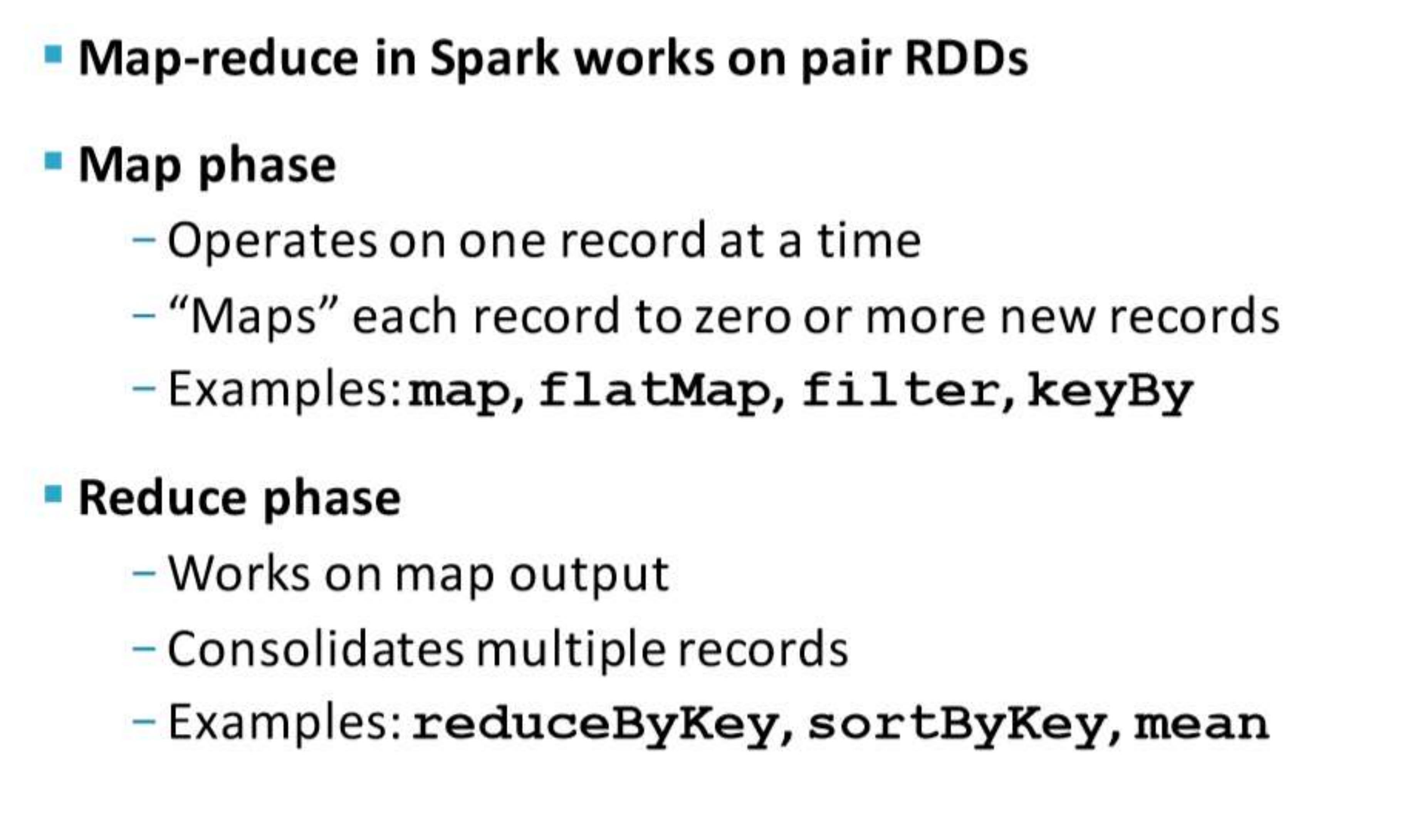


```
rdd = sc.parallelize([1, 1, 2, 3])
rdd.union(rdd).collect()
[1, 1, 2, 3, 1, 1, 2, 3]
```



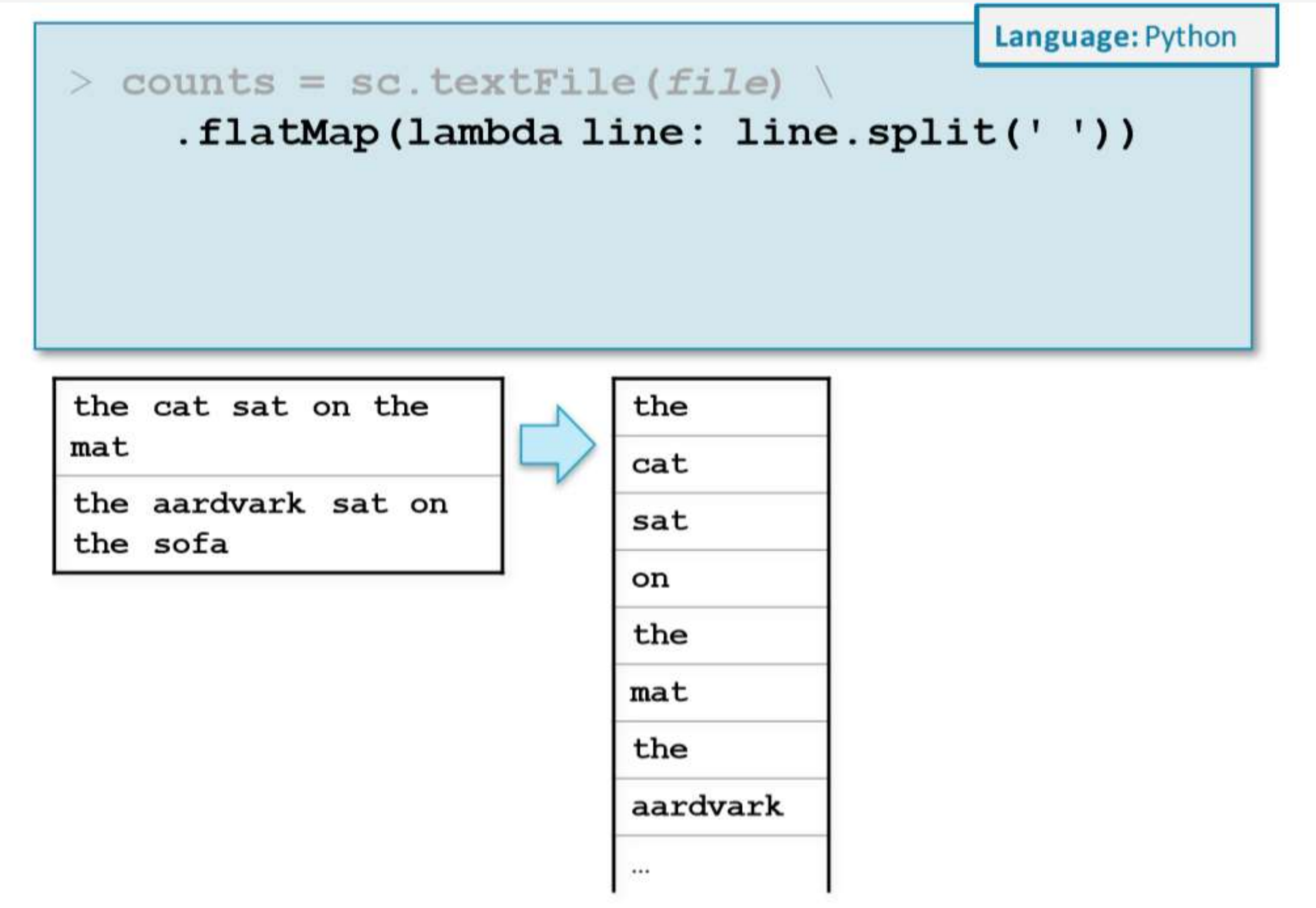
```
x = sc.parallelize(range(0,5))
y = sc.parallelize(range(1000, 1005))
x.zip(y).collect()
[(0, 1000), (1, 1001), (2, 1002), (3, 1003), (4, 1004)]
```

CDH – Spark – Rdd – MapReduce



- **Map-reduce in Spark works on pair RDDs**
- **Map phase**
  - Operates on one record at a time
  - “Maps” each record to zero or more new records
  - Examples: **`map`, `flatMap`, `filter`, `keyBy`**
- **Reduce phase**
  - Works on map output
  - Consolidates multiple records
  - Examples: **`reduceByKey`, `sortByKey`, `mean`**

CDH – Spark – Rdd – Word Count



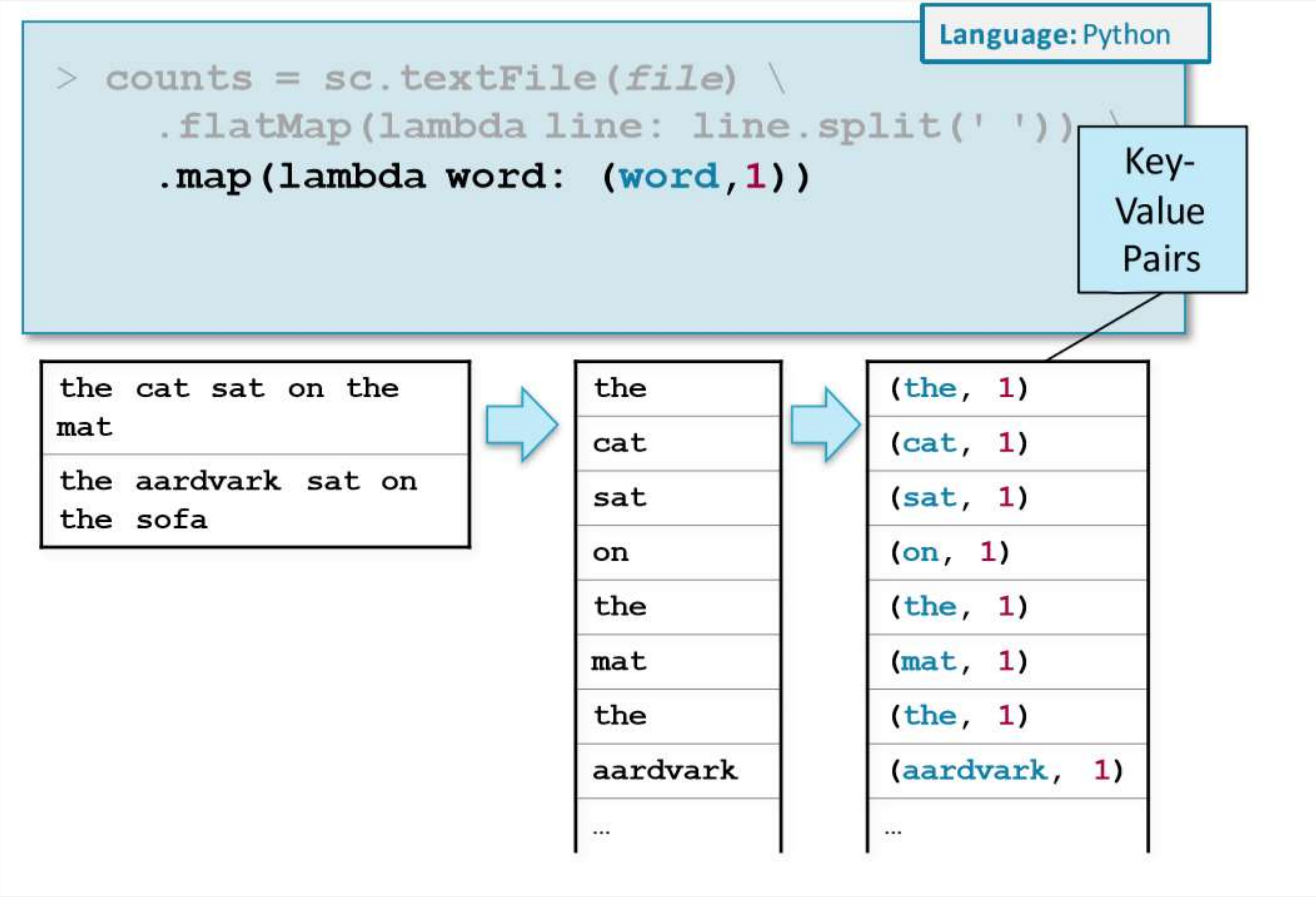
Language: Python

```
> counts = sc.textFile(file) \
    .flatMap(lambda line: line.split(' '))
```

the cat sat on the mat
the aardvark sat on the sofa

the
cat
sat
on
the
mat
the
aardvark
…

CDH – Spark – Rdd – Word Count



Language: Python

```
> counts = sc.textFile(file) \
    .flatMap(lambda line: line.split(' ')) \
    .map(lambda word: (word,1))
```

Key-Value Pairs

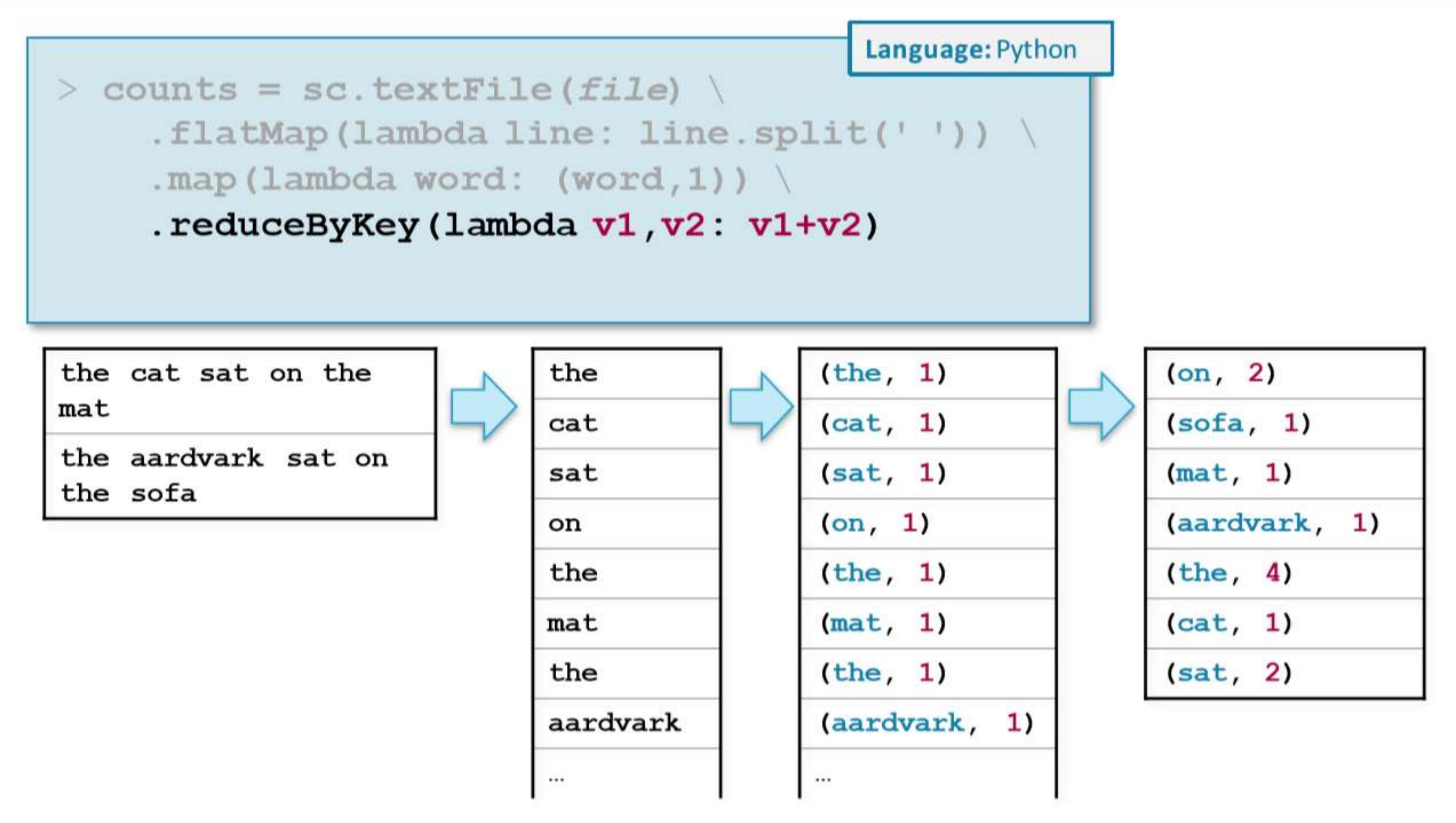
the cat sat on the mat

the aardvark sat on the sofa

the
cat
sat
on
the
mat
the
aardvark
…

(the, 1)
(cat, 1)
(sat, 1)
(on, 1)
(the, 1)
(mat, 1)
(the, 1)
(aardvark, 1)
…

CDH – Spark – Rdd – Word Count



Language: Python

```
> counts = sc.textFile(file) \
    .flatMap(lambda line: line.split(' ')) \
    .map(lambda word: (word,1)) \
    .reduceByKey(lambda v1,v2: v1+v2)
```

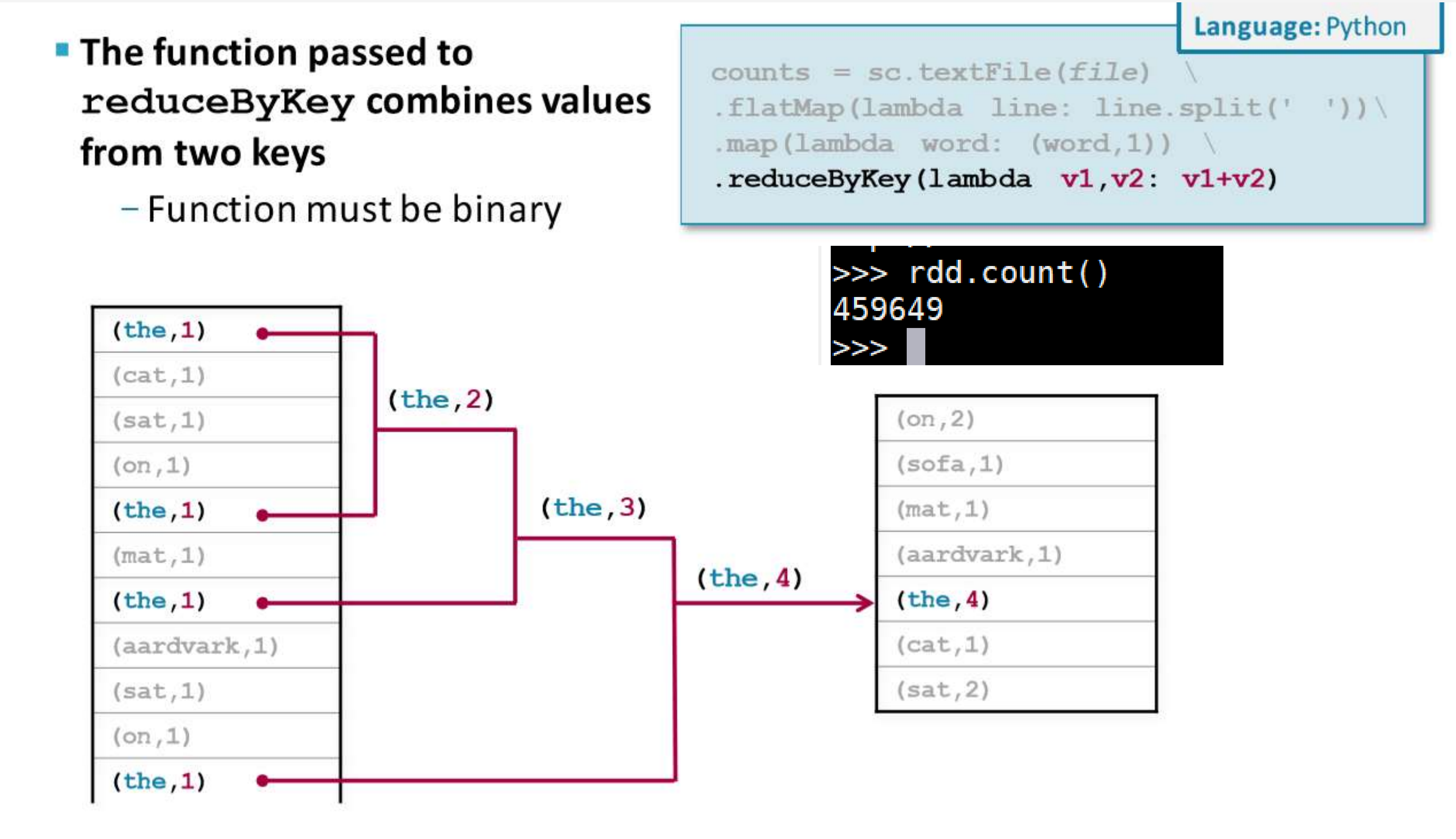
```
the cat sat on the
mat

the aardvark sat on
the sofa
```

```
the
cat
sat
on
the
mat
the
aardvark
…
```

```
(the, 1)
(cat, 1)
(sat, 1)
(on, 1)
(the, 1)
(mat, 1)
(the, 1)
(aardvark, 1)
…
```

```
(on, 2)
(sofa, 1)
(mat, 1)
(aardvark, 1)
(the, 4)
(cat, 1)
(sat, 2)
```

CDH – Spark – Rdd – Word Count



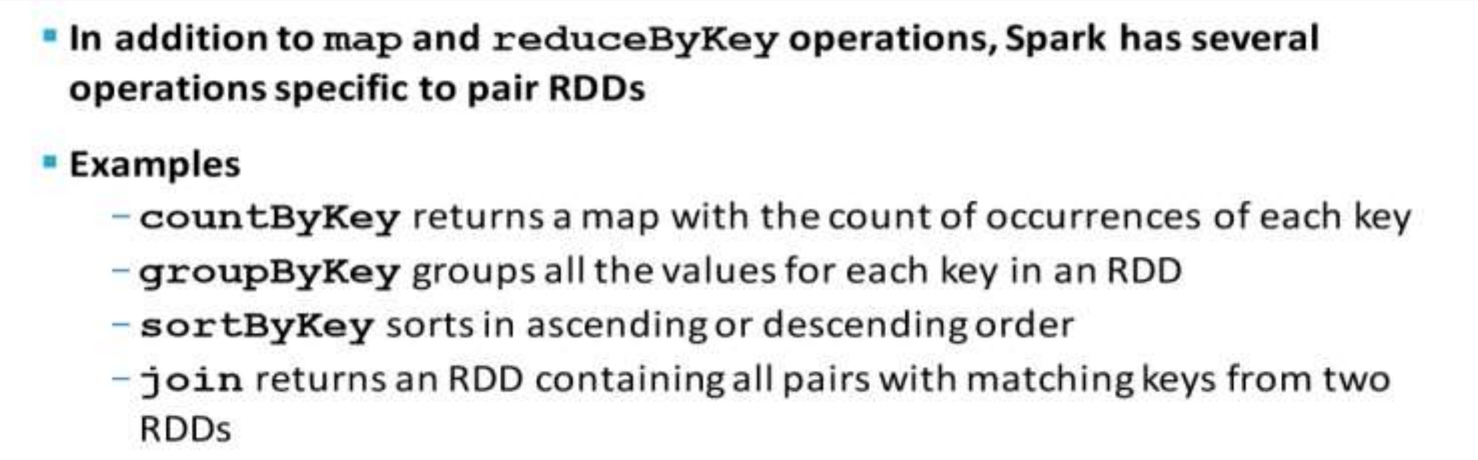
- **The function passed to `reduceByKey` combines values from two keys**
  - Function must be binary

Language: Python

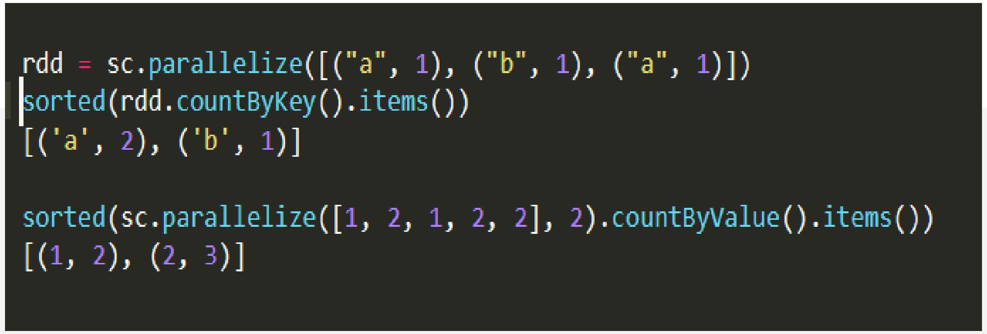
```
counts = sc.textFile(file) \
.flatMap(lambda line: line.split(' '))\
.map(lambda word: (word,1)) \
.reduceByKey(lambda v1,v2: v1+v2)
```

```
>>> rdd.count()
459649
>>>
```

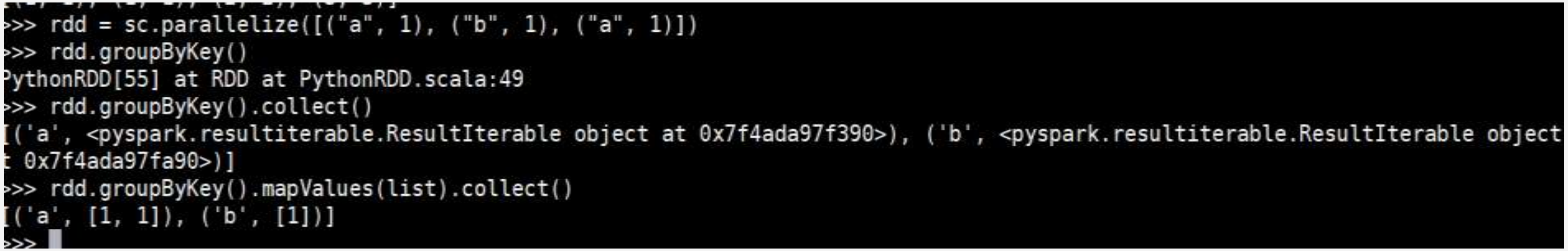
CDH – Spark – Rdd – others



- In addition to `map` and `reduceByKey` operations, Spark has several operations specific to pair RDDs
- Examples
  - `countByKey` returns a map with the count of occurrences of each key
  - `groupByKey` groups all the values for each key in an RDD
  - `sortByKey` sorts in ascending or descending order
  - `join` returns an RDD containing all pairs with matching keys from two RDDs



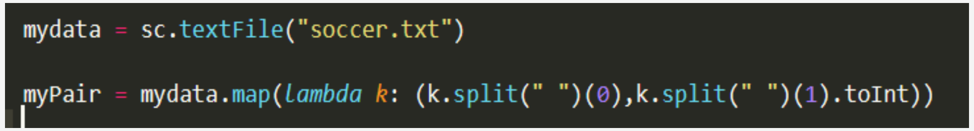
```
rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
sorted(rdd.countByKey().items())
[('a', 2), ('b', 1)]

sorted(sc.parallelize([1, 2, 1, 2, 2], 2).countByValue().items())
[(1, 2), (2, 3)]
```



```
>>> rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
>>> rdd.groupByKey()
PythonRDD[55] at RDD at PythonRDD.scala:49
>>> rdd.groupByKey().collect()
[('a', <pyspark.resultiterable.ResultIterable object at 0x7f4ada97f390>), ('b', <pyspark.resultiterable.ResultIterable object
t 0x7f4ada97fa90>)]
>>> rdd.groupByKey().mapValues(list).collect()
[('a', [1, 1]), ('b', [1])]
>>> 
```

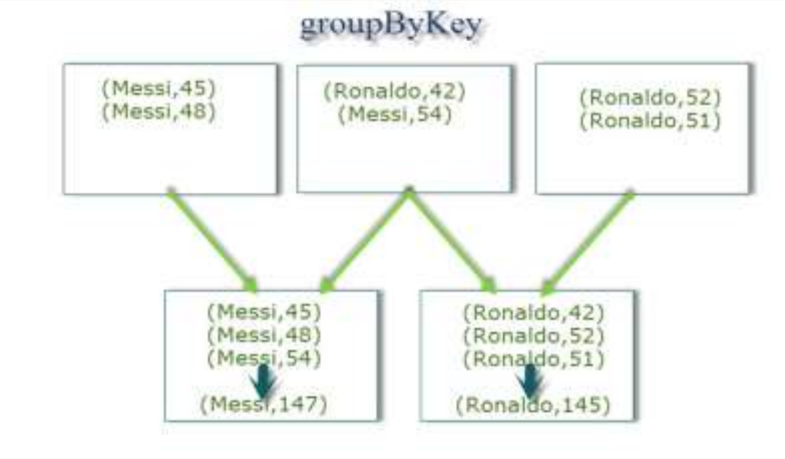
CDH – Spark – Rdd – others

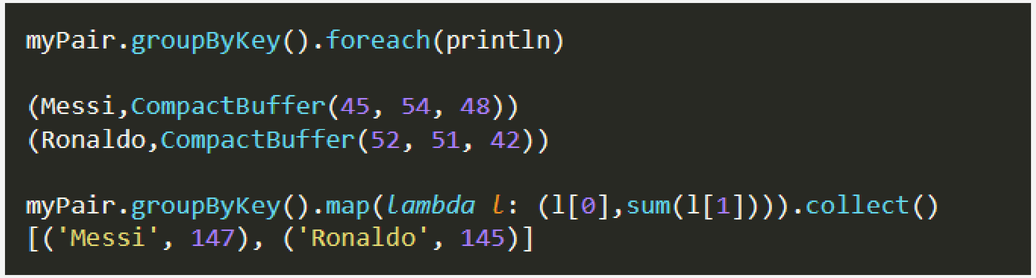
soccer.txt
Messi 45
Ronaldo 52
Messi 54
Ronaldo 51
Messi 48
Ronaldo 42



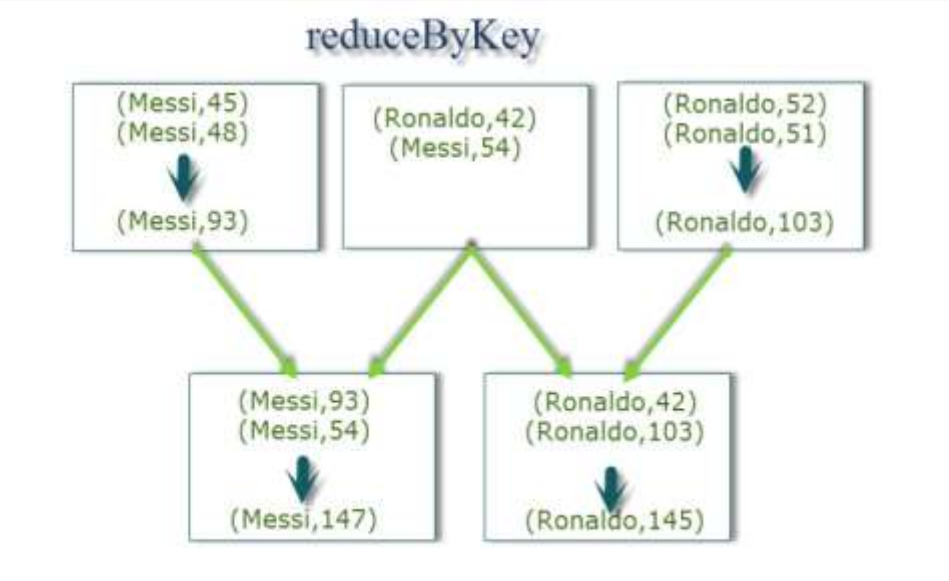
```
mydata = sc.textFile("soccer.txt")

myPair = mydata.map(lambda k: (k.split(" ")(0),k.split(" ")(1).toInt))
```

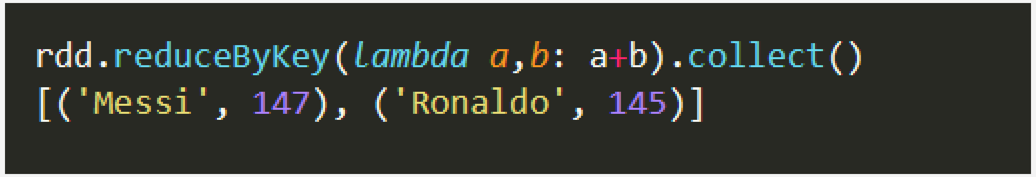
(Messi, 45)
(Ronaldo, 52)
(Messi, 54)
(Ronaldo ,51)
(Messi, 48)
(Ronaldo, 42)





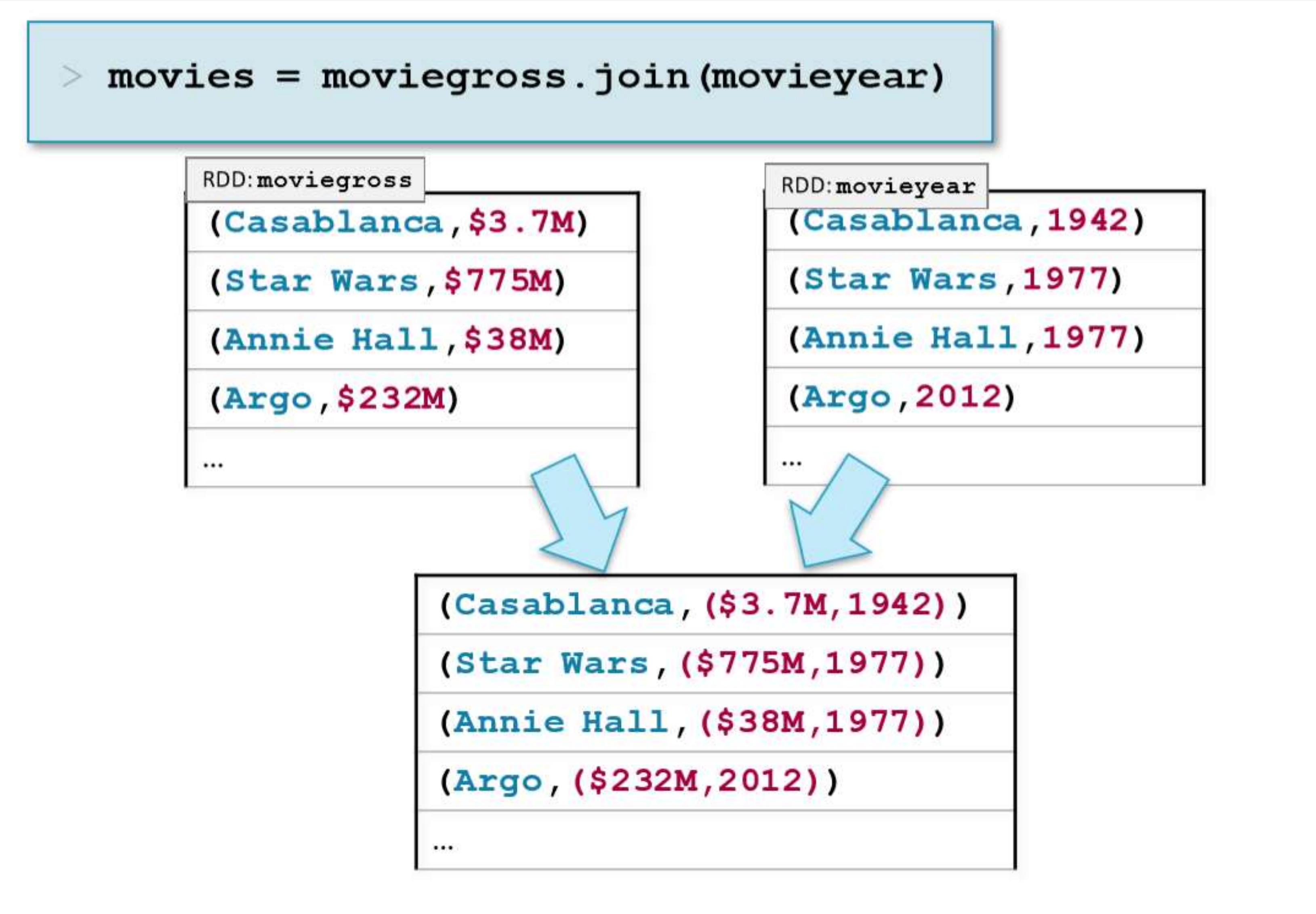
```
myPair.groupByKey().foreach(println)

(Messi,CompactBuffer(45, 54, 48))
(Ronaldo,CompactBuffer(52, 51, 42))

myPair.groupByKey().map(lambda l: (l[0],sum(l[1]))).collect()
[('Messi', 147), ('Ronaldo', 145)]
```





```
rdd.reduceByKey(lambda a,b: a+b).collect()
[('Messi', 147), ('Ronaldo', 145)]
```

CDH – Spark – Rdd – others



```
> movies = moviegross.join(movieyear)
```

RDD: moviegross

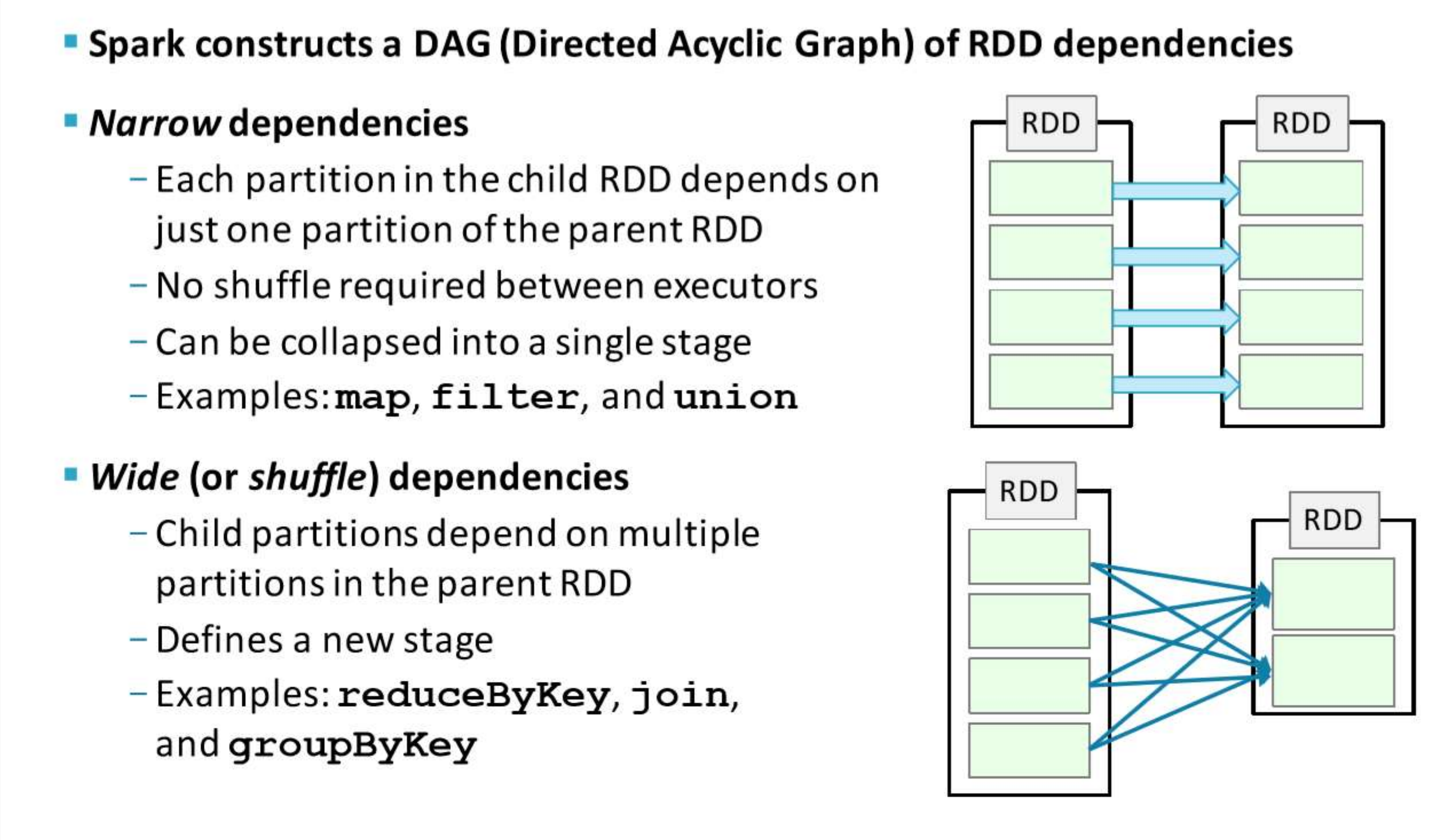
```
(Casablanca,$3.7M)
(Star Wars,$775M)
(Annie Hall,$38M)
(Argo,$232M)
…
```

RDD: movieyear

```
(Casablanca,1942)
(Star Wars,1977)
(Annie Hall,1977)
(Argo,2012)
…
```

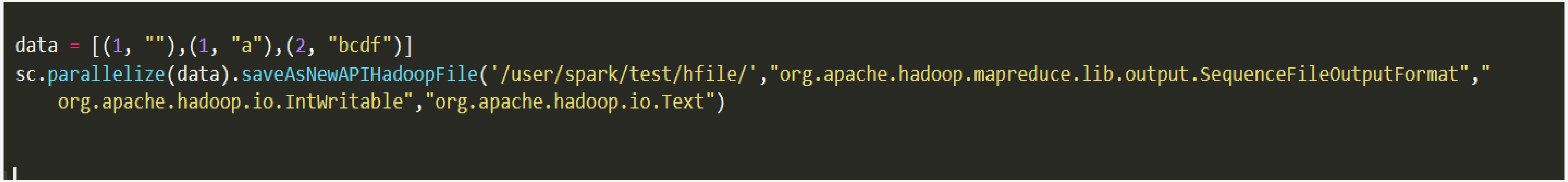
```
(Casablanca,($3.7M,1942))
(Star Wars,($775M,1977))
(Annie Hall,($38M,1977))
(Argo,($232M,2012))
…
```

CDH – Spark – Rdd – others

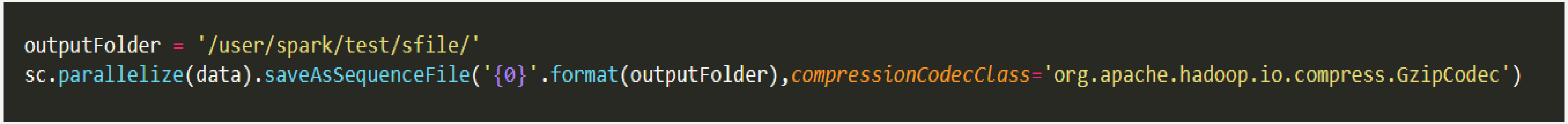


- **Spark constructs a DAG (Directed Acyclic Graph) of RDD dependencies**
- ***Narrow* dependencies**
  - Each partition in the child RDD depends on just one partition of the parent RDD
  - No shuffle required between executors
  - Can be collapsed into a single stage
  - Examples: **`map`**, **`filter`**, and **`union`**
- ***Wide* (or *shuffle*) dependencies**
  - Child partitions depend on multiple partitions in the parent RDD
  - Defines a new stage
  - Examples: **`reduceByKey`**, **`join`**, and **`groupByKey`**

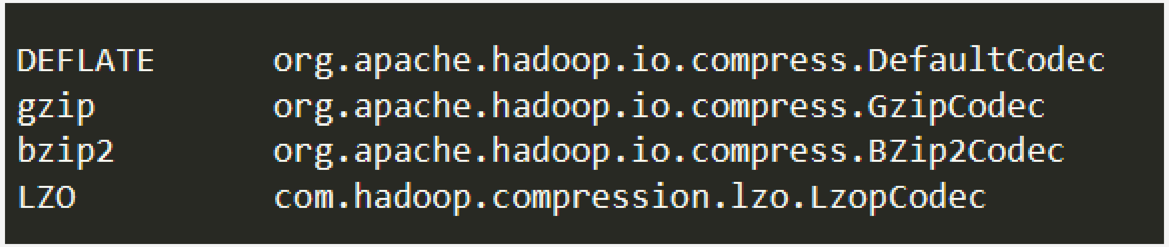
CDH – Spark – Rdd – Write files



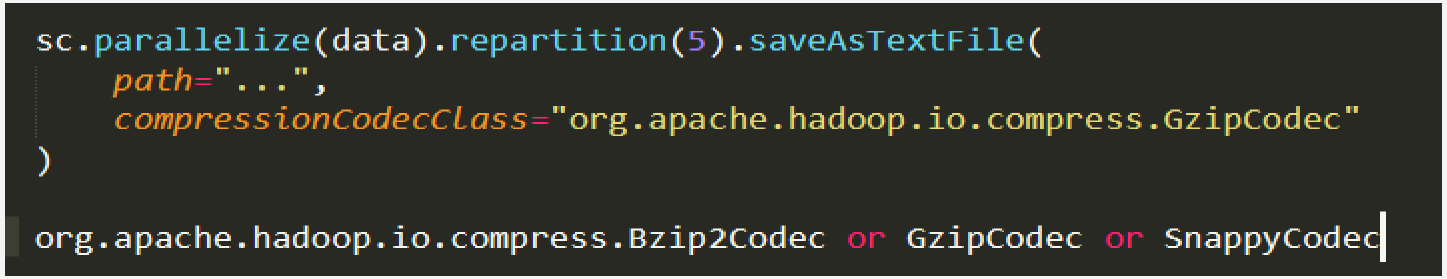
```
data = [(1, ""),(1, "a"),(2, "bcdf")]
sc.parallelize(data).saveAsNewAPIHadoopFile('/user/spark/test/hfile/',"org.apache.hadoop.mapreduce.lib.output.SequenceFileOutputFormat","
    org.apache.hadoop.io.IntWritable","org.apache.hadoop.io.Text")
```



```
outputFolder = '/user/spark/test/sfile/'
sc.parallelize(data).saveAsSequenceFile('{0}'.format(outputFolder),compressionCodecClass='org.apache.hadoop.io.compress.GzipCodec')
```

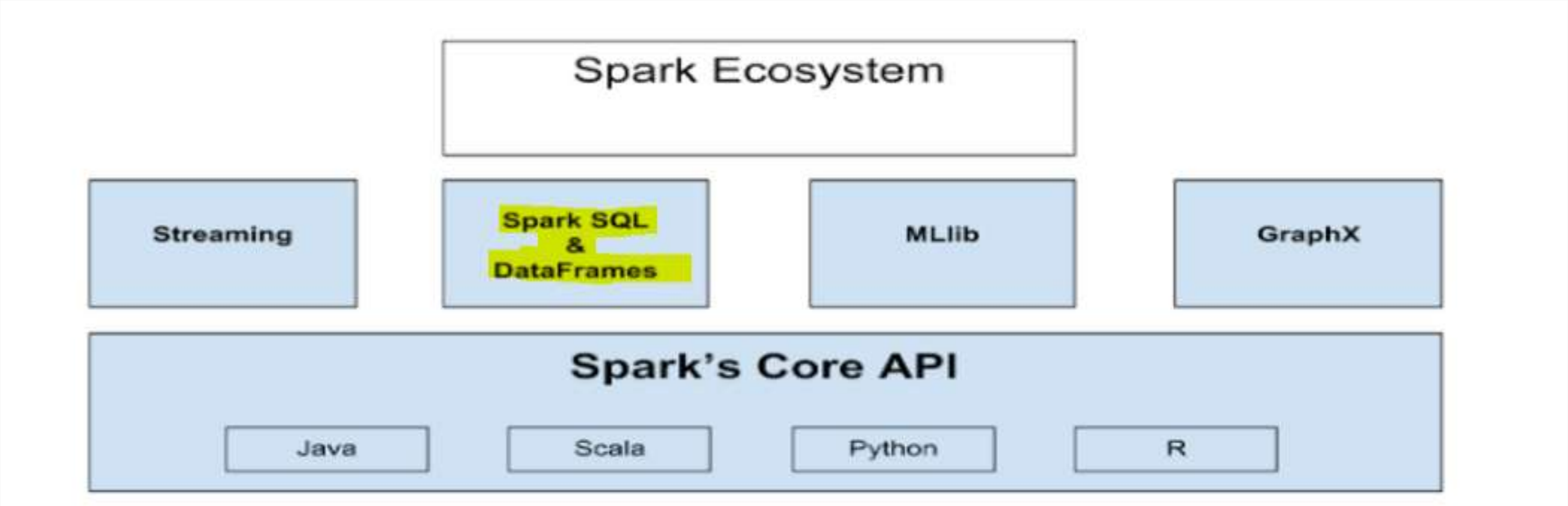


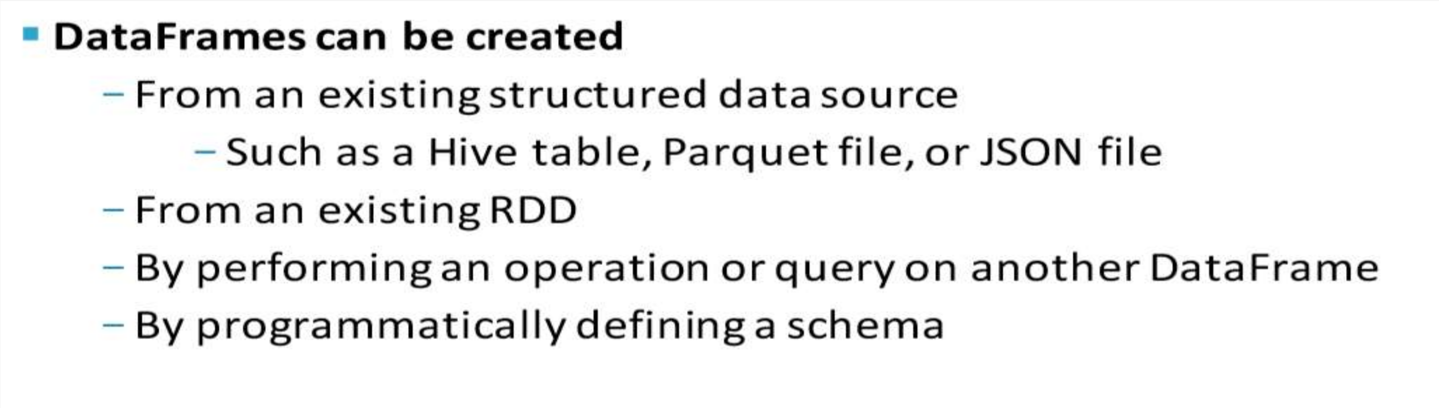
```
DEFLATE      org.apache.hadoop.io.compress.DefaultCodec
gzip         org.apache.hadoop.io.compress.GzipCodec
bzip2        org.apache.hadoop.io.compress.BZip2Codec
LZO          com.hadoop.compression.lzo.LzopCodec
```



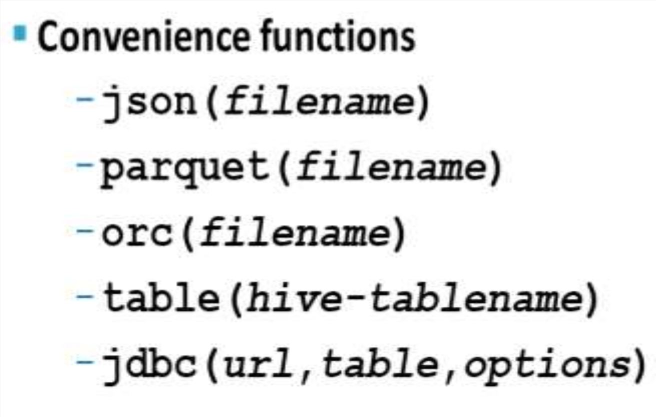
```
sc.parallelize(data).repartition(5).saveAsTextFile(
    path="...",
    compressionCodecClass="org.apache.hadoop.io.compress.GzipCodec"
)

org.apache.hadoop.io.compress.Bzip2Codec or GzipCodec or SnappyCodec
```

CDH – Spark- DF





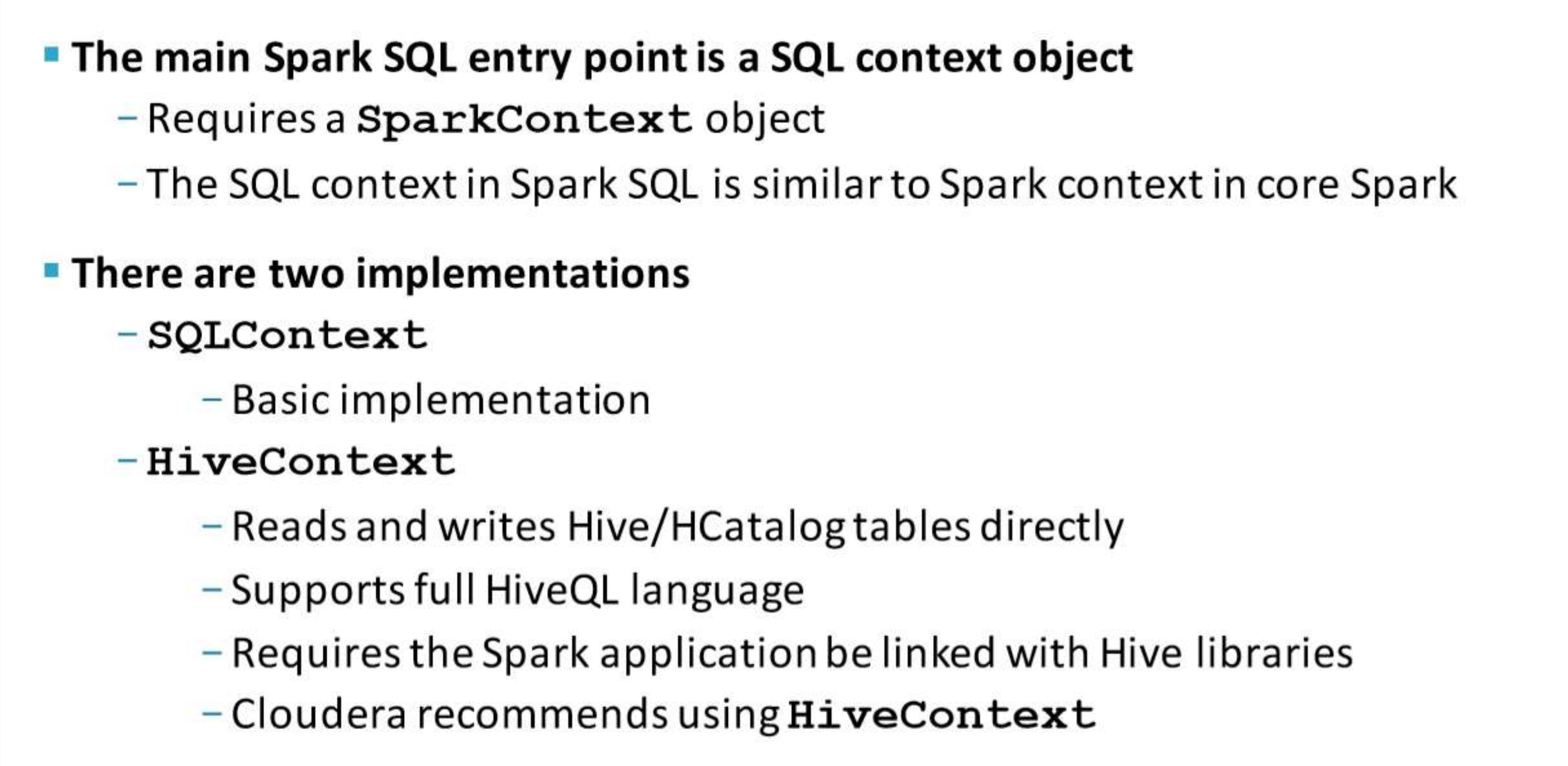
- **DataFrames can be created**
  - From an existing structured data source
    - Such as a Hive table, Parquet file, or JSON file
  - From an existing RDD
  - By performing an operation or query on another DataFrame
  - By programmatically defining a schema



- Convenience functions
  - `json(`*`filename`*`)`
  - `parquet(`*`filename`*`)`
  - `orc(`*`filename`*`)`
  - `table(`*`hive-tablename`*`)`
  - `jdbc(`*`url`*`,`*`table`*`,`*`options`*`)`

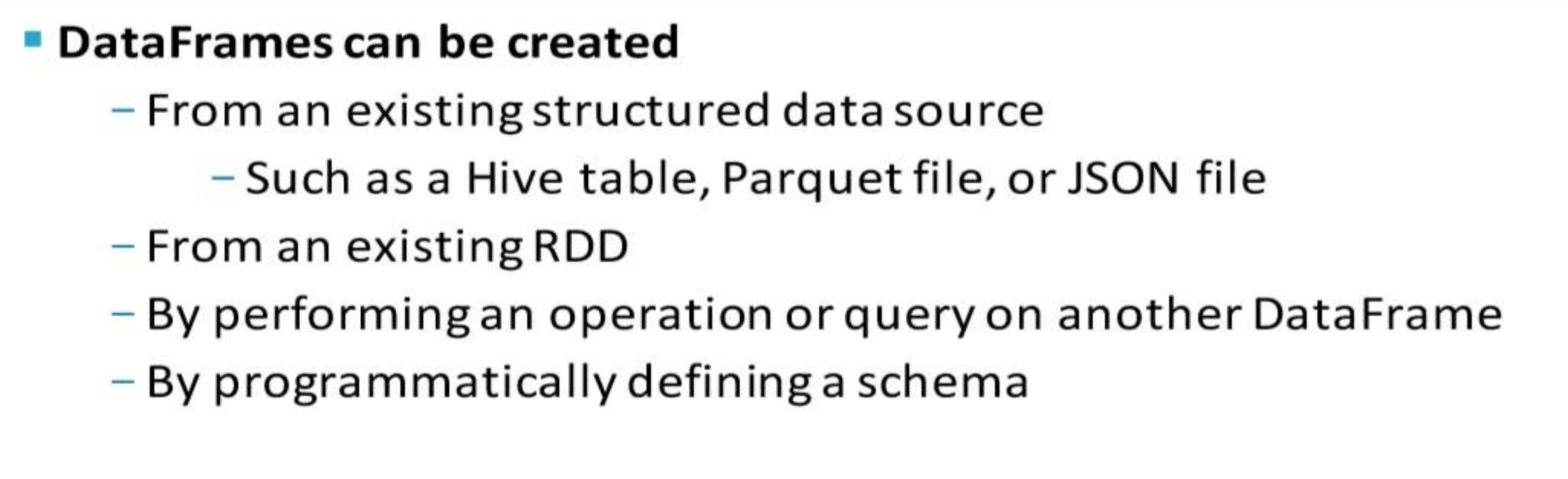
CDH – Spark- DF

Spark SQL

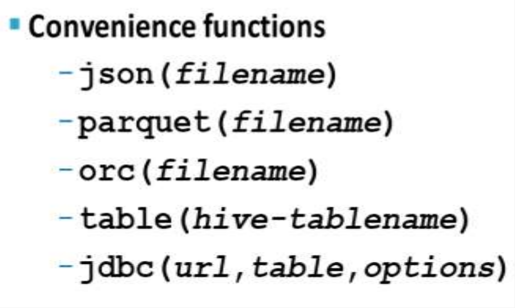


- **The main Spark SQL entry point is a SQL context object**
  - Requires a **`SparkContext`** object
  - The SQL context in Spark SQL is similar to Spark context in core Spark
- **There are two implementations**
  - **`SQLContext`**
    - Basic implementation
  - **`HiveContext`**
    - Reads and writes Hive/HCatalog tables directly
    - Supports full HiveQL language
    - Requires the Spark application be linked with Hive libraries
    - Cloudera recommends using **`HiveContext`**

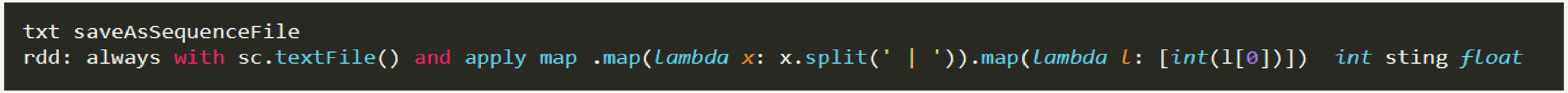
CDH – Spark- DF



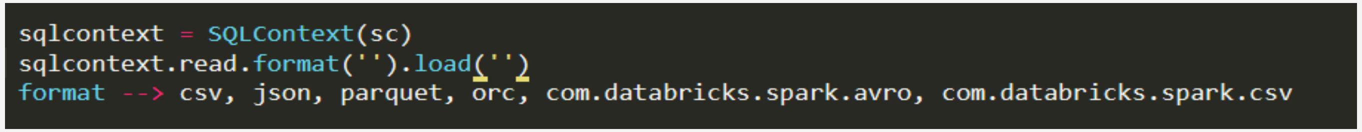
- **DataFrames can be created**
  - From an existing structured data source
    - Such as a Hive table, Parquet file, or JSON file
  - From an existing RDD
  - By performing an operation or query on another DataFrame
  - By programmatically defining a schema



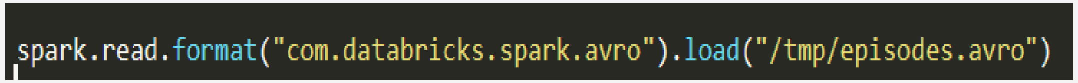
- Convenience functions
  - json(*filename*)
  - parquet(*filename*)
  - orc(*filename*)
  - table(*hive-tablename*)
  - jdbc(*url*,*table*,*options*)



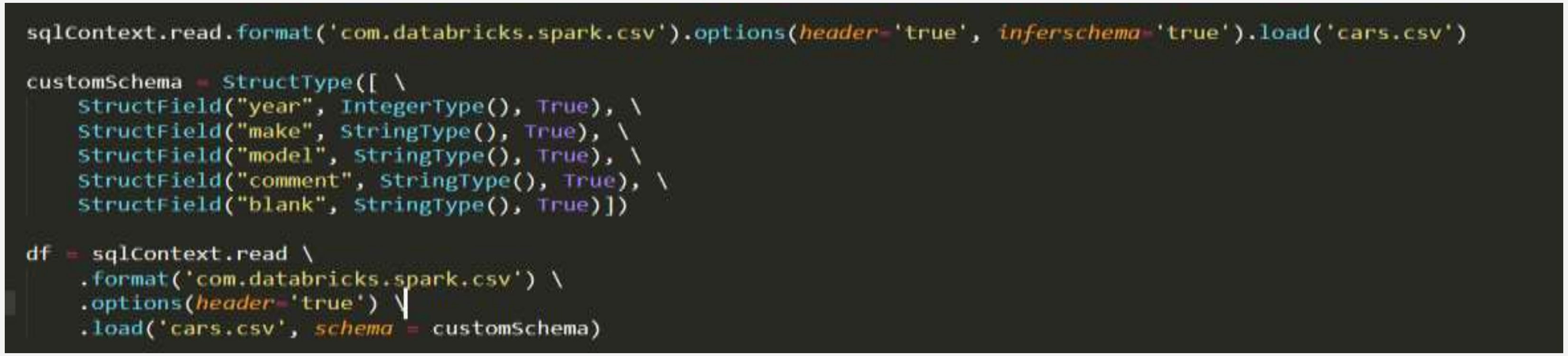
```
txt saveAsSequenceFile
rdd: always with sc.textFile() and apply map .map(lambda x: x.split(' | ')).map(lambda l: [int(l[0])])  int sting float
```



```
sqlcontext = SQLContext(sc)
sqlcontext.read.format('').load('')
format --> csv, json, parquet, orc, com.databricks.spark.avro, com.databricks.spark.csv
```

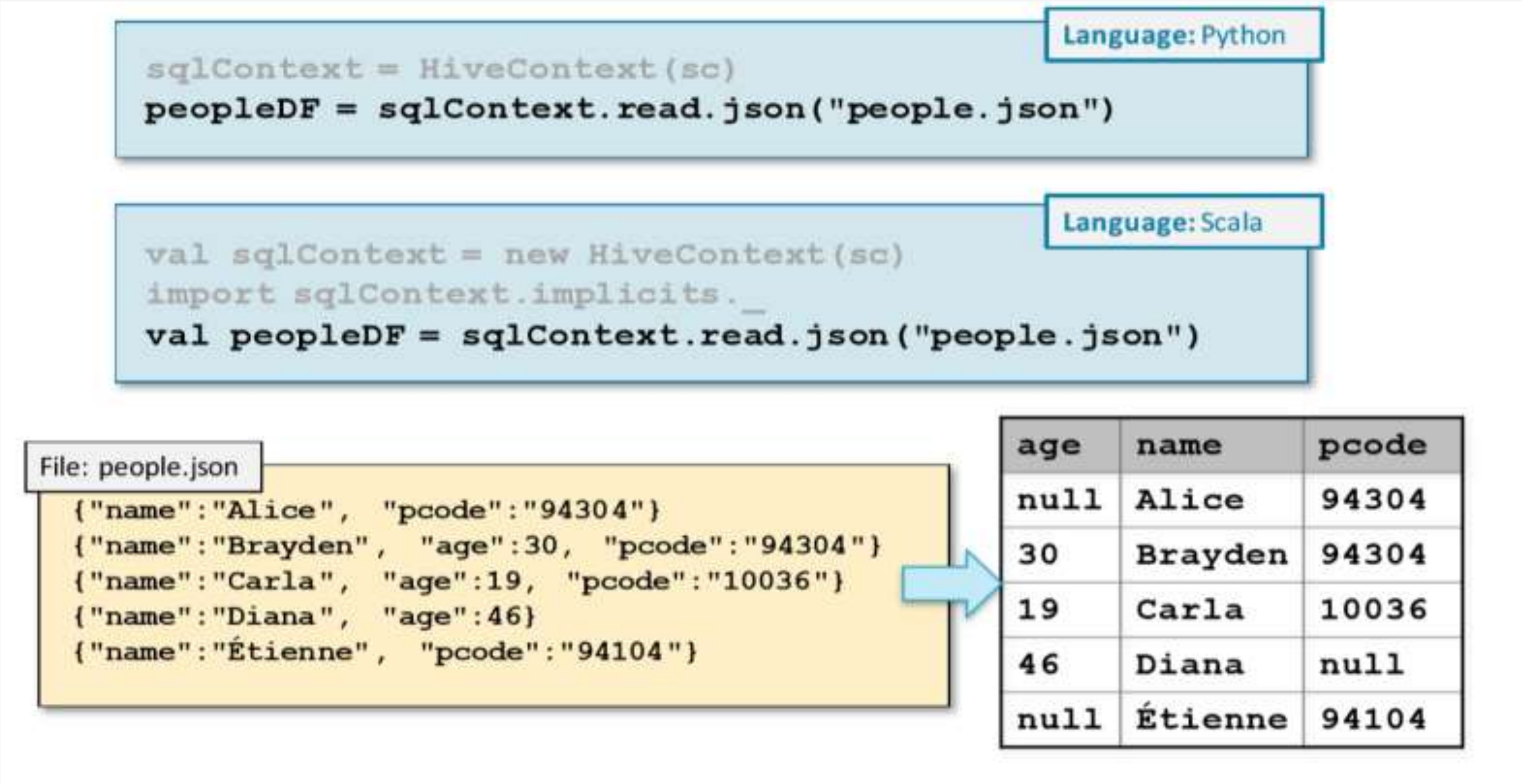


```
spark.read.format("com.databricks.spark.avro").load("/tmp/episodes.avro")
```



```
sqlContext.read.format('com.databricks.spark.csv').options(header='true', inferschema='true').load('cars.csv')

customSchema = StructType([ \
    StructField("year", IntegerType(), True), \
    StructField("make", StringType(), True), \
    StructField("model", StringType(), True), \
    StructField("comment", StringType(), True), \
    StructField("blank", StringType(), True)])

df = sqlContext.read \
    .format('com.databricks.spark.csv') \
    .options(header='true') \
    .load('cars.csv', schema = customSchema)
```

CDH – Spark- DF



Language: Python

```
sqlContext = HiveContext(sc)
peopleDF = sqlContext.read.json("people.json")
```

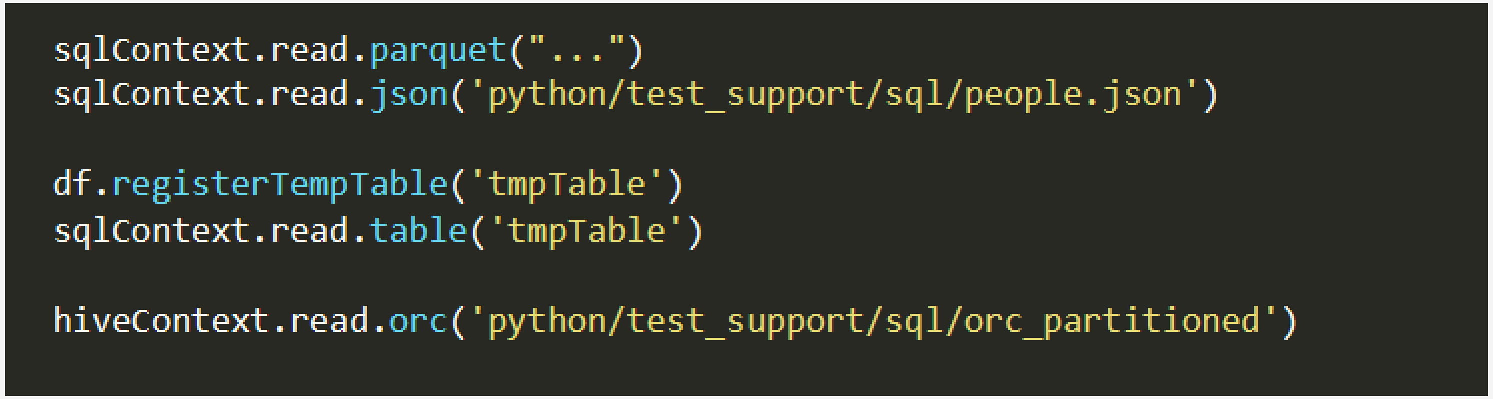
Language: Scala

```
val sqlContext = new HiveContext(sc)
import sqlContext.implicits._
val peopleDF = sqlContext.read.json("people.json")
```

File: people.json

```
{"name":"Alice", "pcode":"94304"}
{"name":"Brayden", "age":30, "pcode":"94304"}
{"name":"Carla", "age":19, "pcode":"10036"}
{"name":"Diana", "age":46}
{"name":"Étienne", "pcode":"94104"}
```

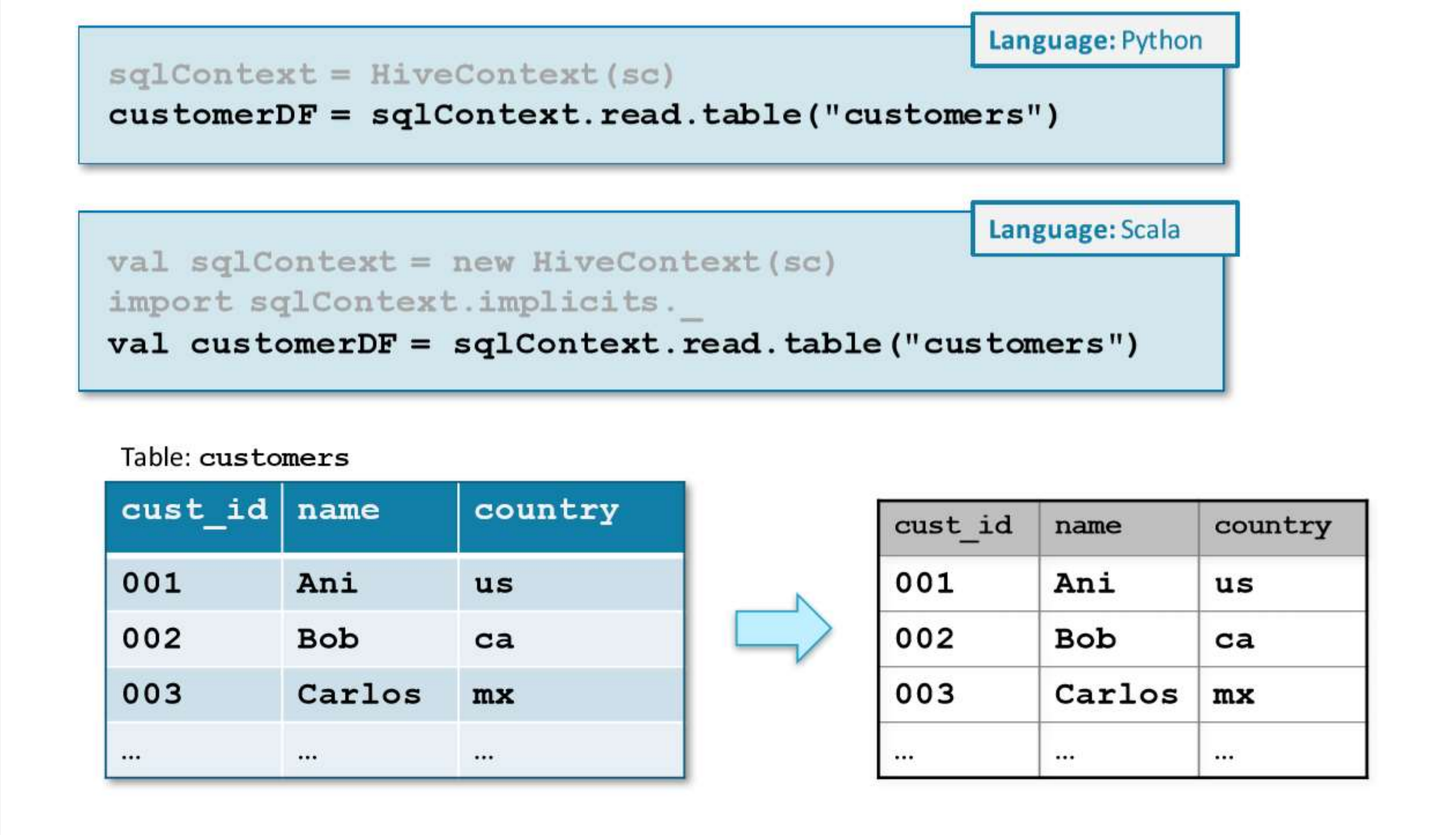
age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104



```
sqlContext.read.parquet("...")
sqlContext.read.json('python/test_support/sql/people.json')

df.registerTempTable('tmpTable')
sqlContext.read.table('tmpTable')

hiveContext.read.orc('python/test_support/sql/orc_partitioned')
```

CDH – Spark- DF

Spark SQL



Language: Python

```
sqlContext = HiveContext(sc)
customerDF = sqlContext.read.table("customers")
```

Language: Scala

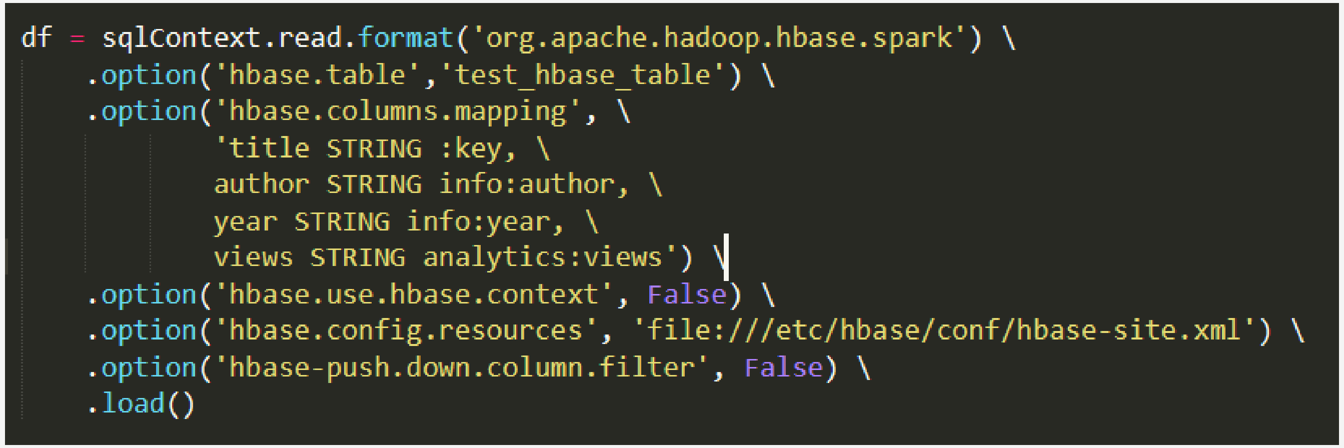
```
val sqlContext = new HiveContext(sc)
import sqlContext.implicits._
val customerDF = sqlContext.read.table("customers")
```

Table: customers

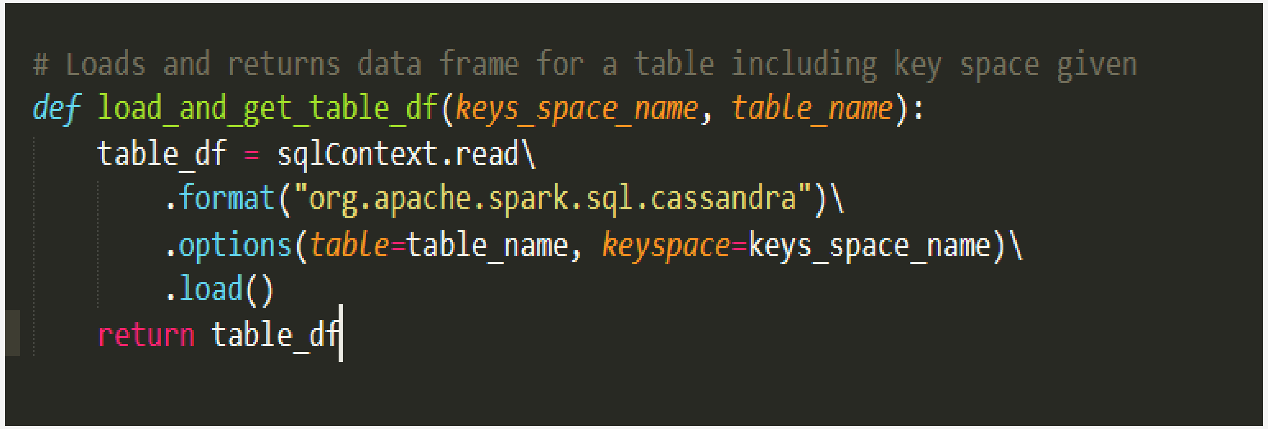
cust_id	name	country
001	Ani	us
002	Bob	ca
003	Carlos	mx
…	…	…

cust_id	name	country
001	Ani	us
002	Bob	ca
003	Carlos	mx
…	…	…

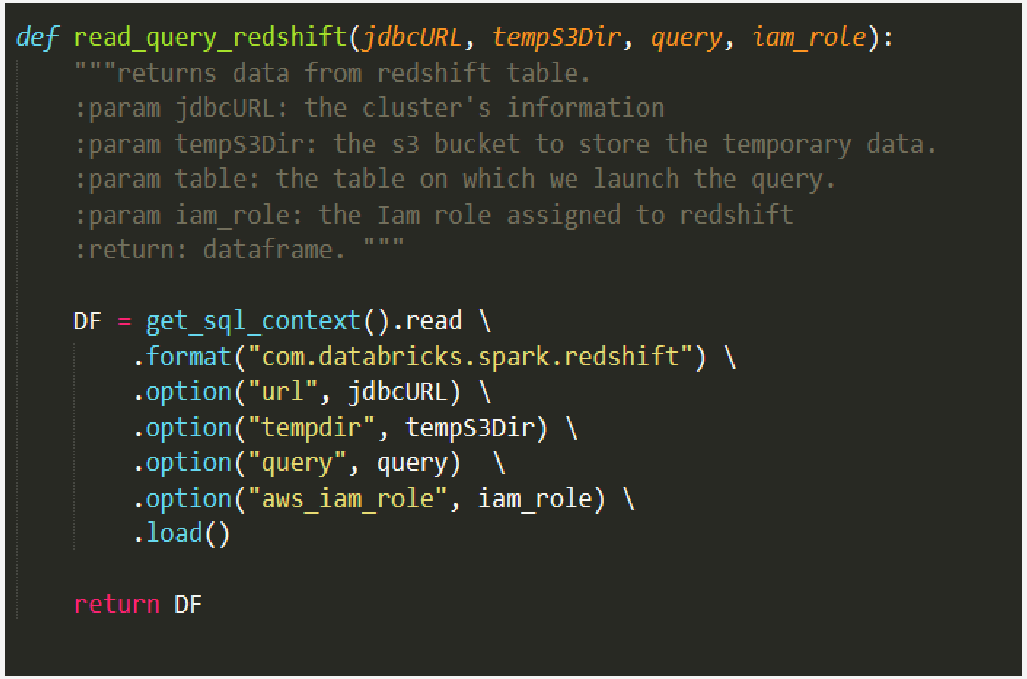
CDH – Spark – DF



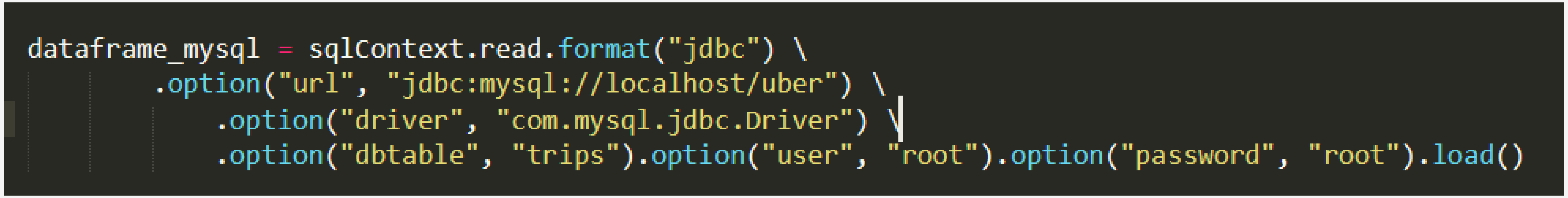
```
df = sqlContext.read.format('org.apache.hadoop.hbase.spark') \
    .option('hbase.table','test_hbase_table') \
    .option('hbase.columns.mapping', \
            'title STRING :key, \
            author STRING info:author, \
            year STRING info:year, \
            views STRING analytics:views') \
    .option('hbase.use.hbase.context', False) \
    .option('hbase.config.resources', 'file:///etc/hbase/conf/hbase-site.xml') \
    .option('hbase-push.down.column.filter', False) \
    .load()
```



```
# Loads and returns data frame for a table including key space given
def load_and_get_table_df(keys_space_name, table_name):
    table_df = sqlContext.read\
        .format("org.apache.spark.sql.cassandra")\
        .options(table=table_name, keyspace=keys_space_name)\
        .load()
    return table_df
```



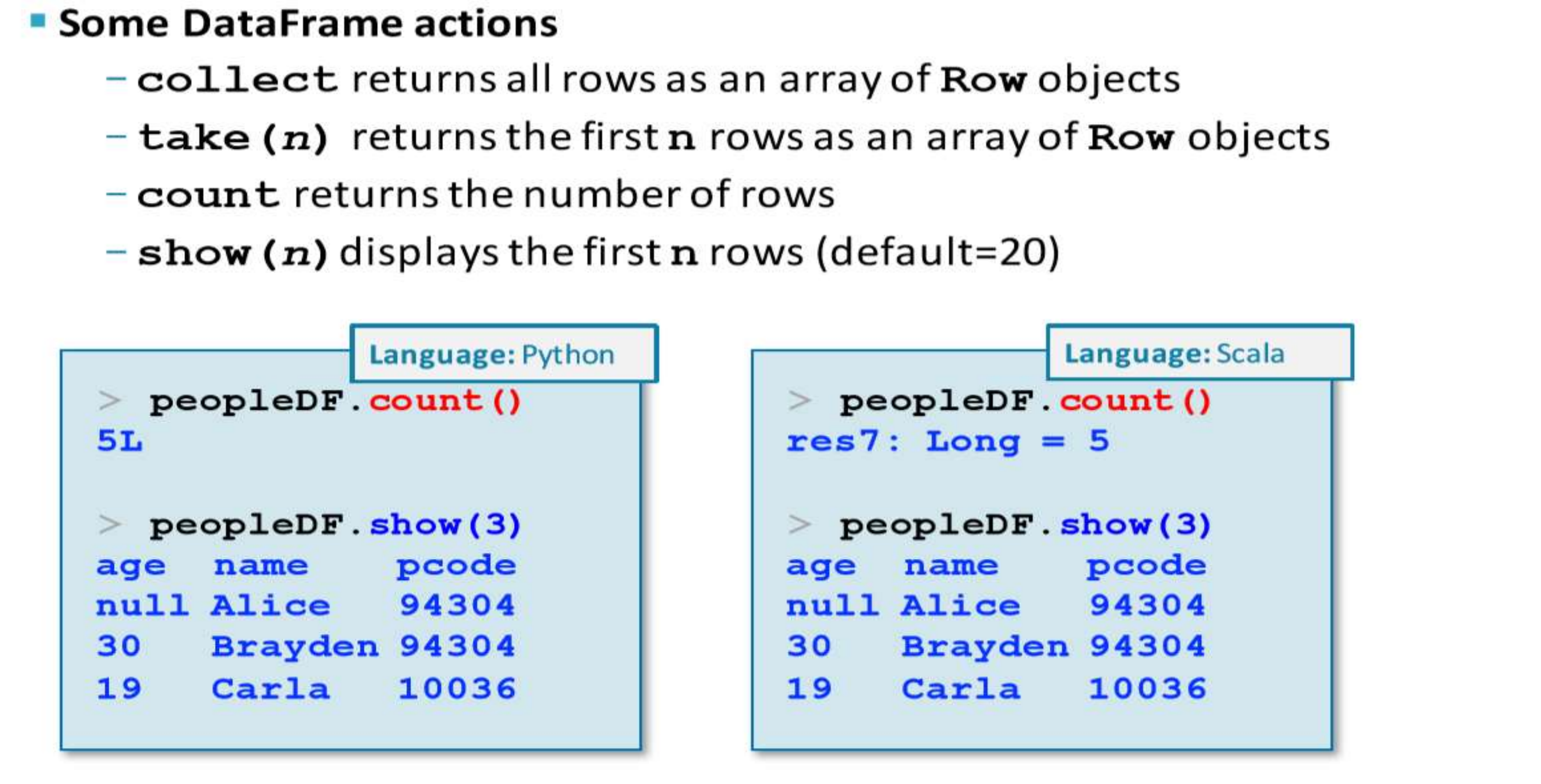
```
def read_query_redshift(jdbcURL, tempS3Dir, query, iam_role):
    """returns data from redshift table.
    :param jdbcURL: the cluster's information
    :param tempS3Dir: the s3 bucket to store the temporary data.
    :param table: the table on which we launch the query.
    :param iam_role: the Iam role assigned to redshift
    :return: dataframe. """

    DF = get_sql_context().read \
        .format("com.databricks.spark.redshift") \
        .option("url", jdbcURL) \
        .option("tempdir", tempS3Dir) \
        .option("query", query)  \
        .option("aws_iam_role", iam_role) \
        .load()

    return DF
```



```
dataframe_mysql = sqlContext.read.format("jdbc") \
        .option("url", "jdbc:mysql://localhost/uber") \
            .option("driver", "com.mysql.jdbc.Driver") \
            .option("dbtable", "trips").option("user", "root").option("password", "root").load()
```

CDH – Spark- DF

Spark SQL



- **Some DataFrame actions**
  - `collect` returns all rows as an array of `Row` objects
  - `take(n)` returns the first `n` rows as an array of `Row` objects
  - `count` returns the number of rows
  - `show(n)` displays the first `n` rows (default=20)

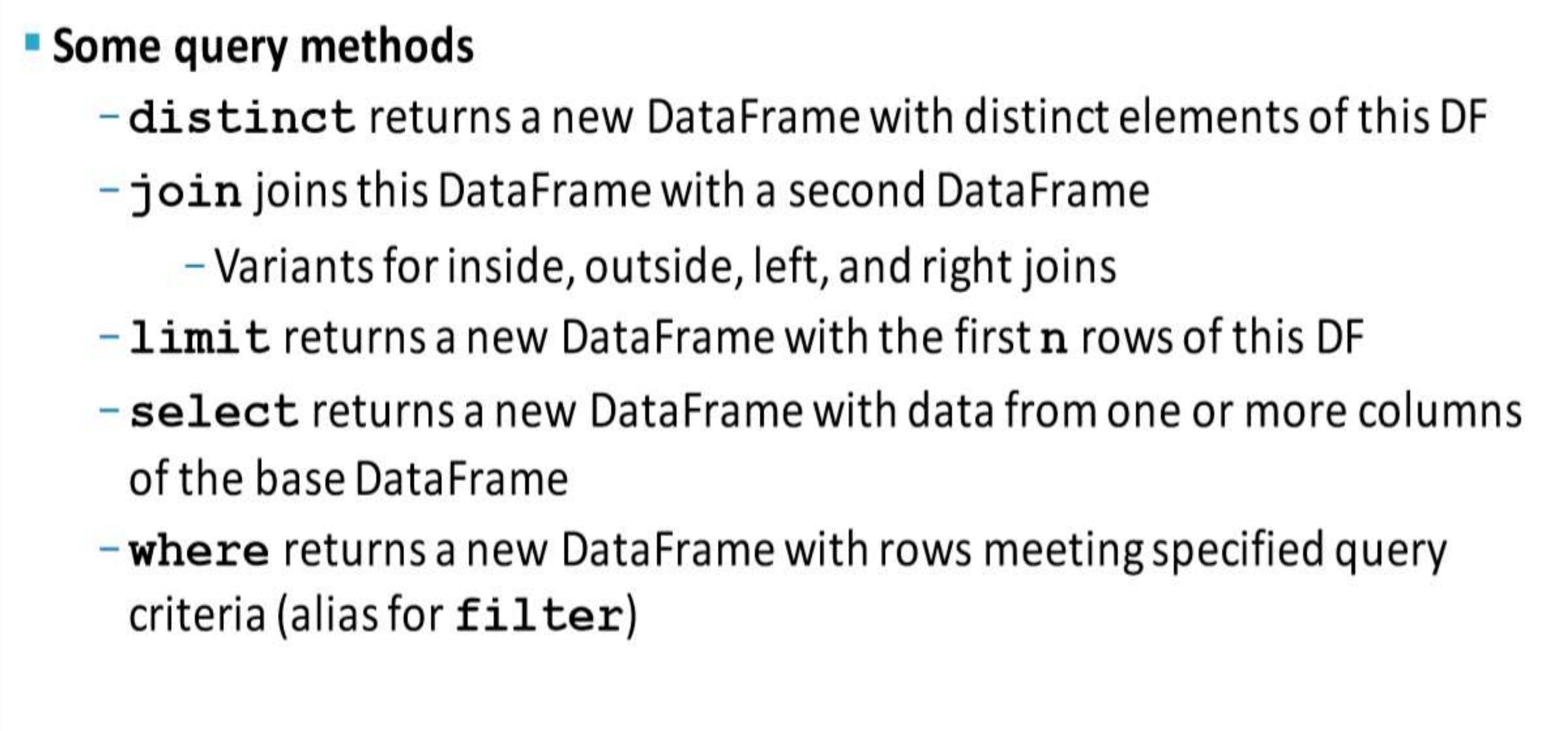
Language: Python

```
> peopleDF.count()
5L

> peopleDF.show(3)
age  name    pcode
null Alice   94304
30   Brayden 94304
19   Carla   10036
```

Language: Scala

```
> peopleDF.count()
res7: Long = 5

> peopleDF.show(3)
age  name    pcode
null Alice   94304
30   Brayden 94304
19   Carla   10036
```

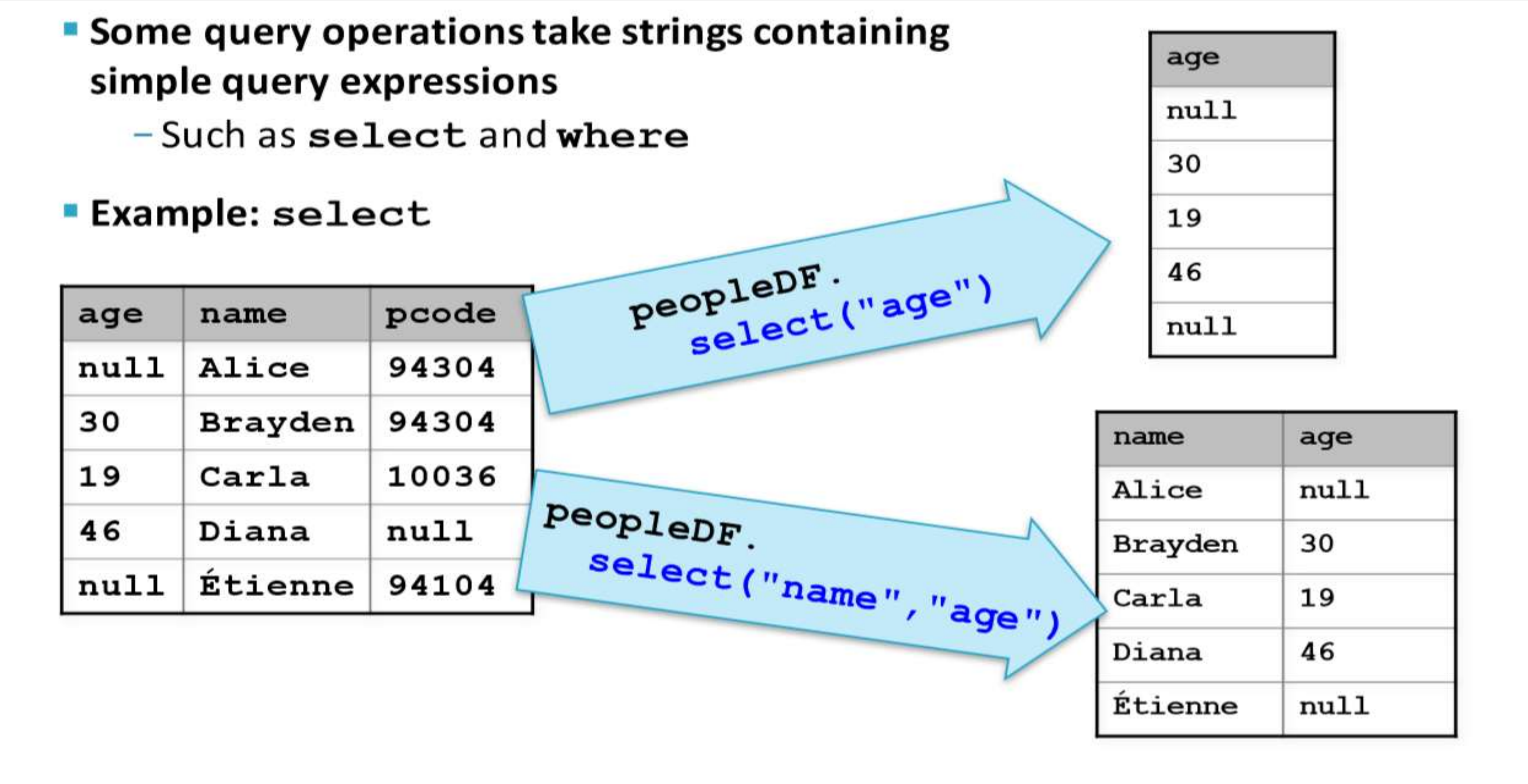
CDH – Spark- DF



- **Some query methods**
  - **`distinct`** returns a new DataFrame with distinct elements of this DF
  - **`join`** joins this DataFrame with a second DataFrame
    - Variants for inside, outside, left, and right joins
  - **`limit`** returns a new DataFrame with the first **`n`** rows of this DF
  - **`select`** returns a new DataFrame with data from one or more columns of the base DataFrame
  - **`where`** returns a new DataFrame with rows meeting specified query criteria (alias for **`filter`**)

CDH – Spark- DF

Spark SQL



- **Some query operations take strings containing simple query expressions**
  - Such as **`select`** and **`where`**
- **Example: `select`**

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104

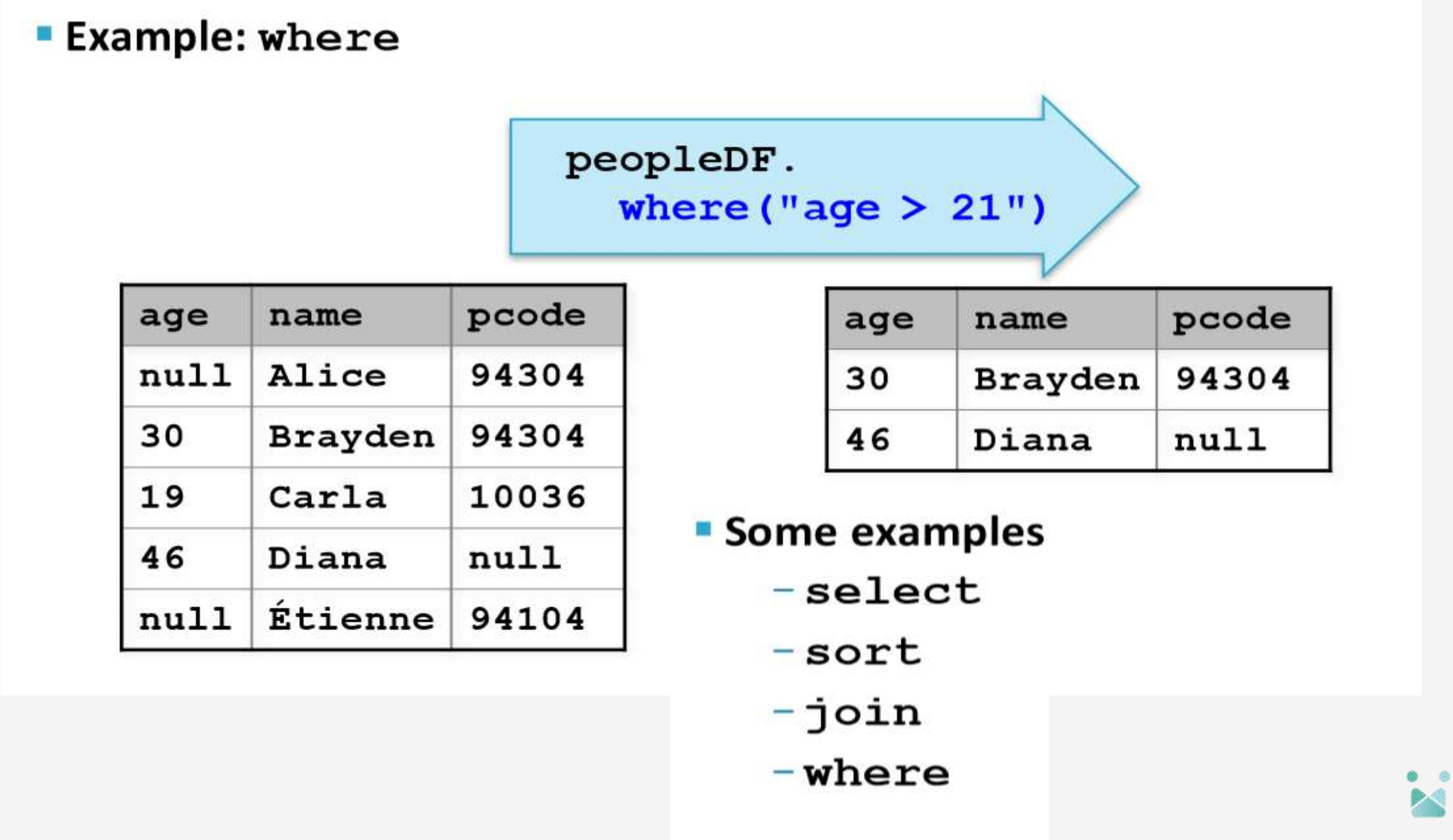
```
peopleDF.
  select("age")
```

age
null
30
19
46
null

```
peopleDF.
  select("name","age")
```

name	age
Alice	null
Brayden	30
Carla	19
Diana	46
Étienne	null

CDH – Spark- DF



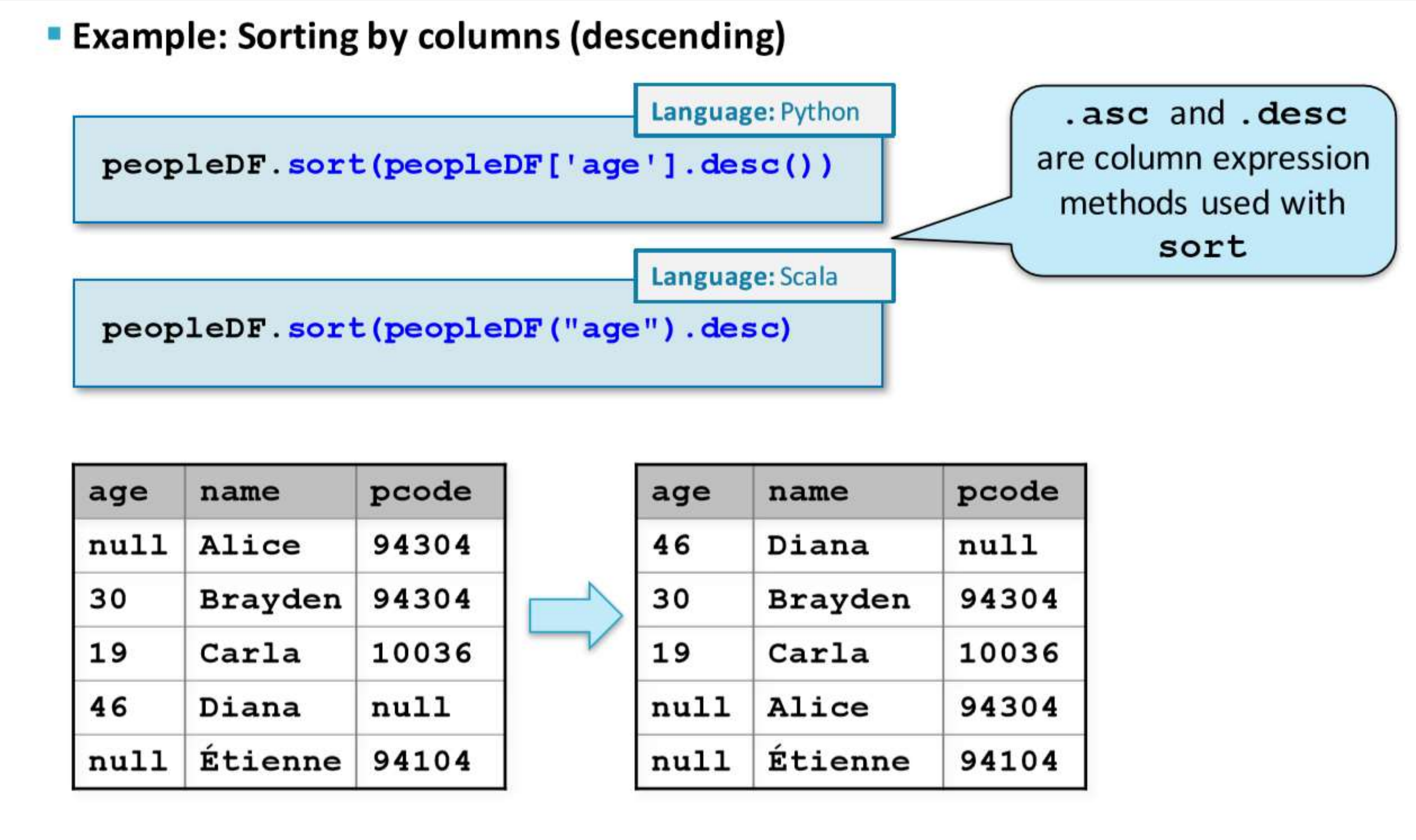
- Example: where

```
peopleDF.
  where("age > 21")
```

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104

age	name	pcode
30	Brayden	94304
46	Diana	null

- Some examples
  - select
  - sort
  - join
  - where



- **Example: Sorting by columns (descending)**

Language: Python

```
peopleDF.sort(peopleDF['age'].desc())
```

Language: Scala

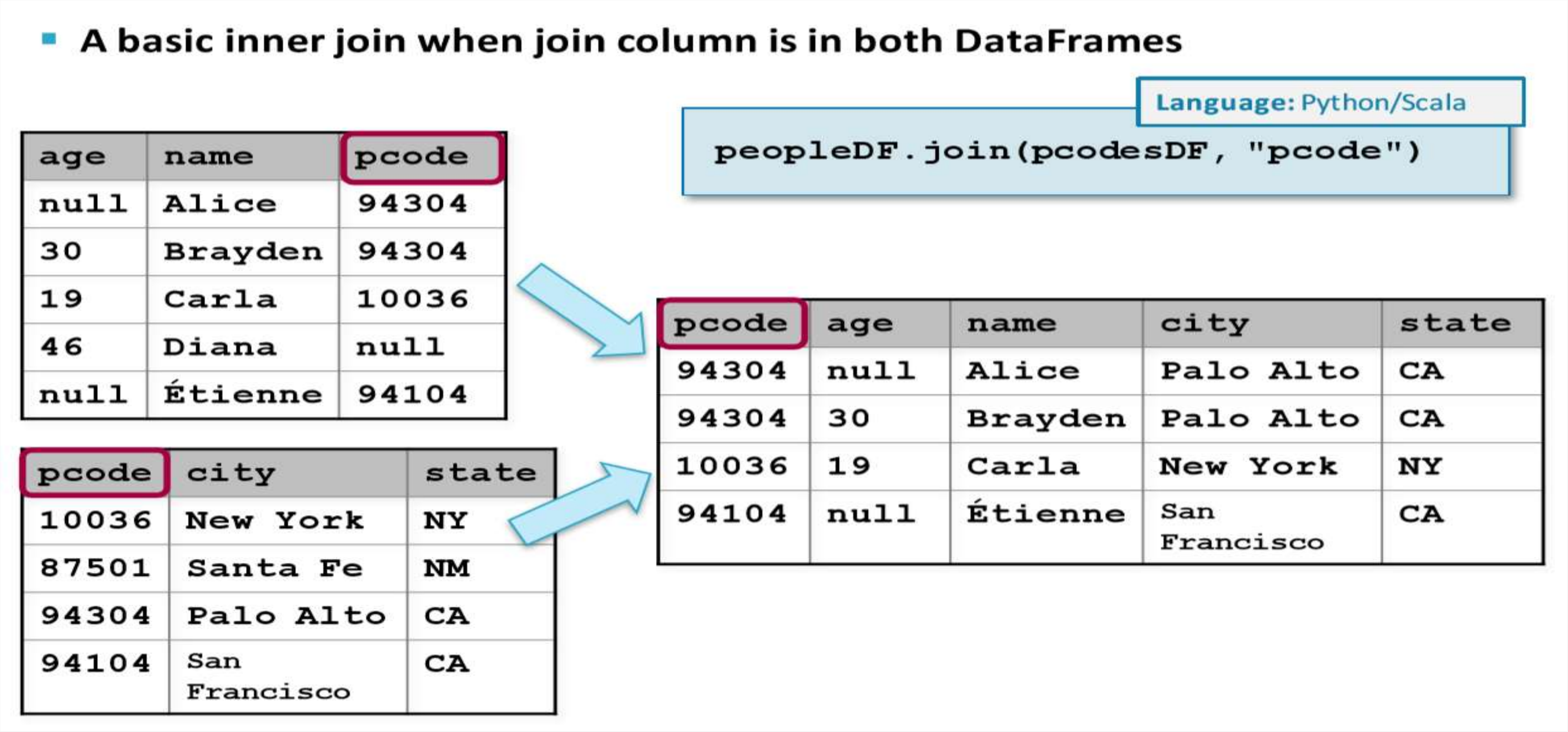
```
peopleDF.sort(peopleDF("age").desc)
```

`.asc` and `.desc` are column expression methods used with `sort`

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104

age	name	pcode
46	Diana	null
30	Brayden	94304
19	Carla	10036
null	Alice	94304
null	Étienne	94104

CDH – Spark- DF



- **A basic inner join when join column is in both DataFrames**

Language: Python/Scala

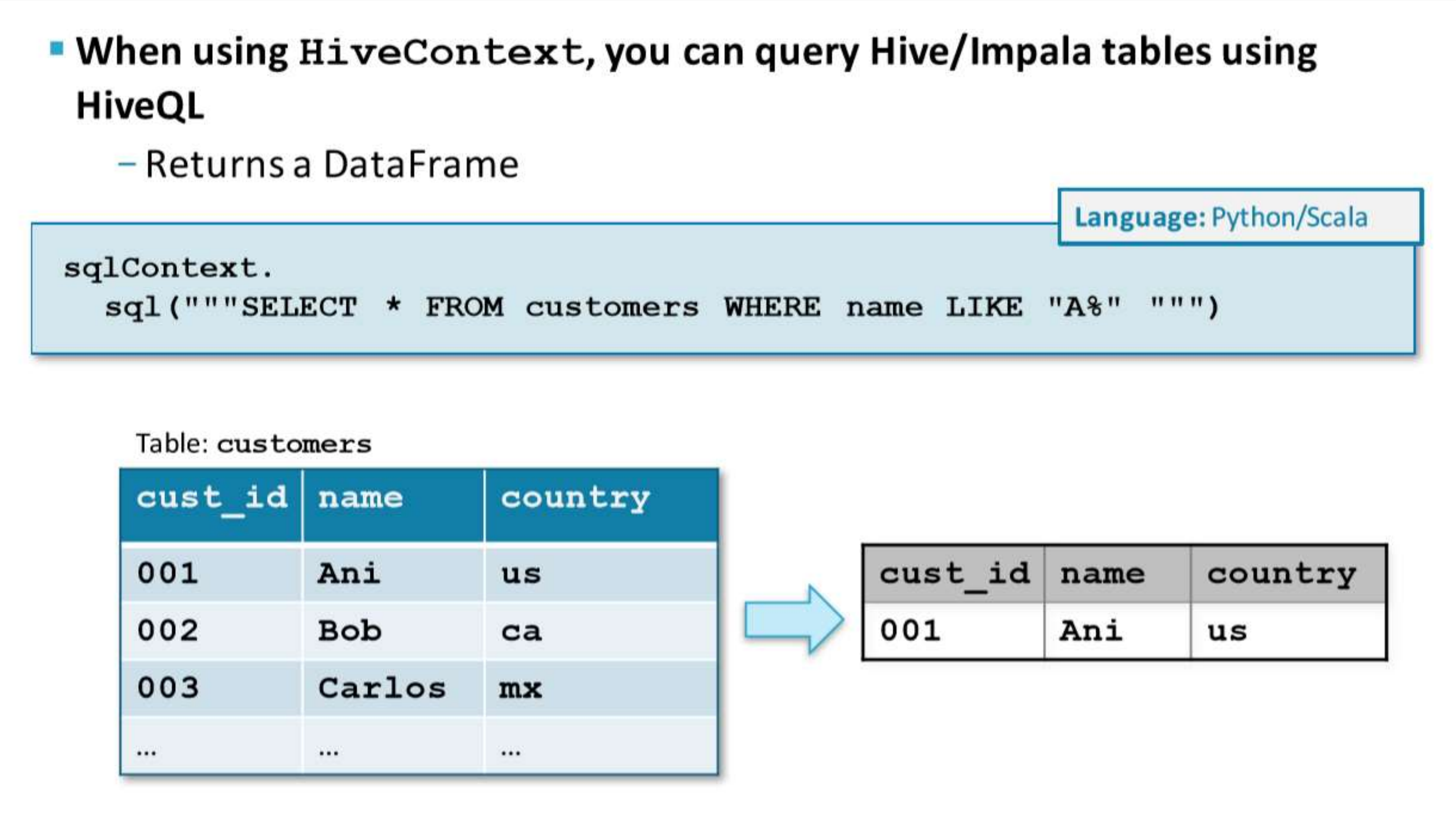
```
peopleDF.join(pcodesDF, "pcode")
```

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104

pcode	city	state
10036	New York	NY
87501	Santa Fe	NM
94304	Palo Alto	CA
94104	San Francisco	CA

pcode	age	name	city	state
94304	null	Alice	Palo Alto	CA
94304	30	Brayden	Palo Alto	CA
10036	19	Carla	New York	NY
94104	null	Étienne	San Francisco	CA

CDH – Spark- DF



- **When using `HiveContext`, you can query Hive/Impala tables using HiveQL**
  - Returns a DataFrame

Language: Python/Scala

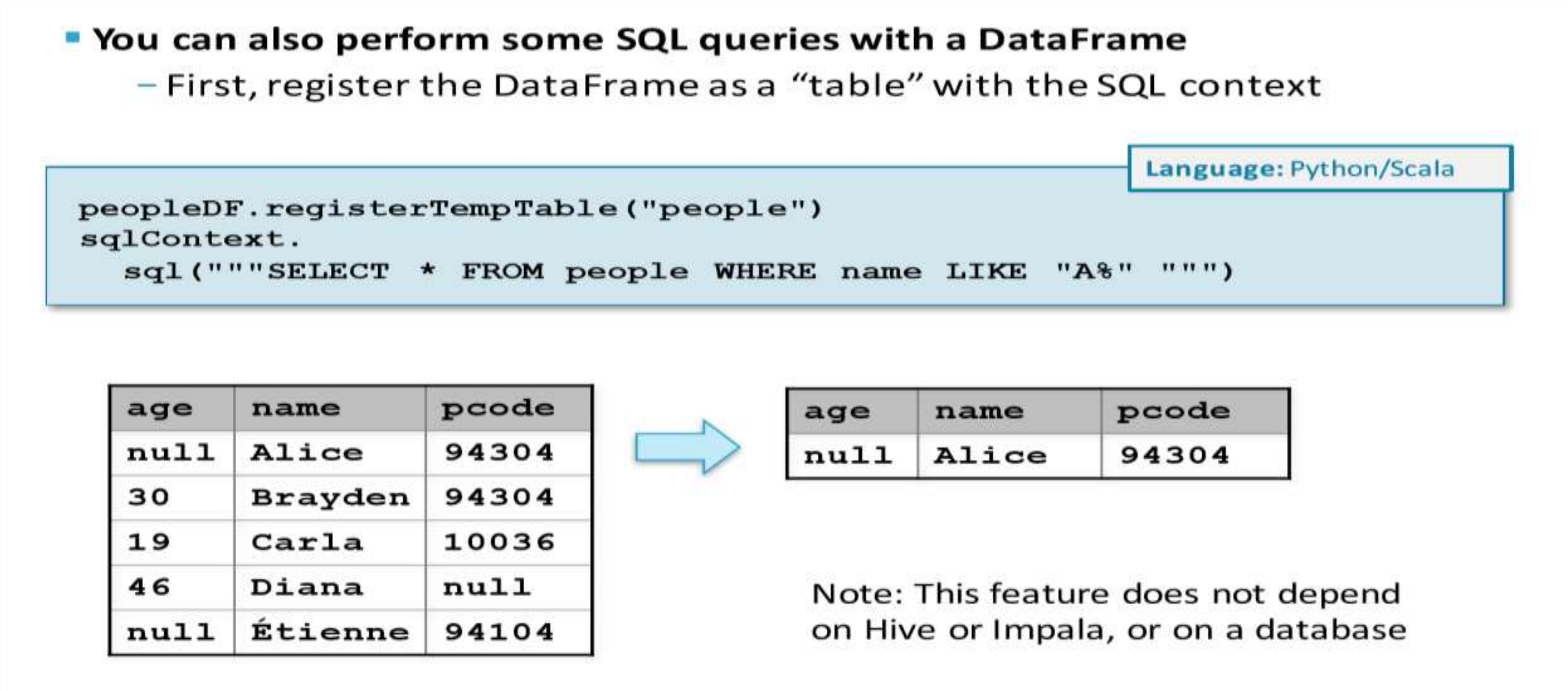
```
sqlContext.
  sql("""SELECT * FROM customers WHERE name LIKE "A%" """)
```

Table: `customers`

cust_id	name	country
001	Ani	us
002	Bob	ca
003	Carlos	mx
…	…	…

cust_id	name	country
001	Ani	us

CDH – Spark- DF



- **You can also perform some SQL queries with a DataFrame**
  - First, register the DataFrame as a “table” with the SQL context

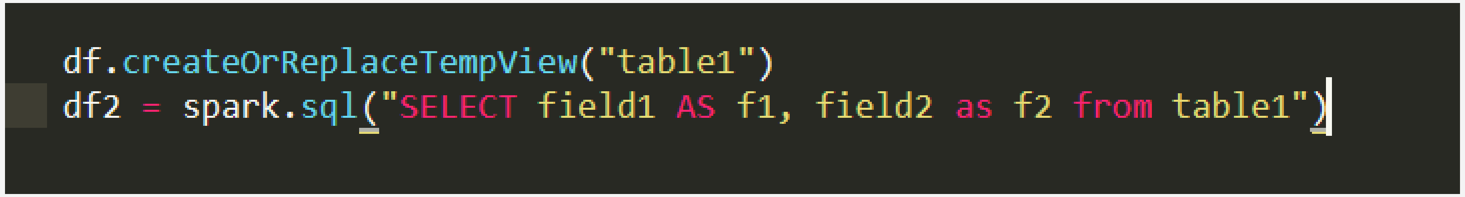
Language: Python/Scala

```
peopleDF.registerTempTable("people")
sqlContext.
   sql("""SELECT * FROM people WHERE name LIKE "A%" """)
```

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104

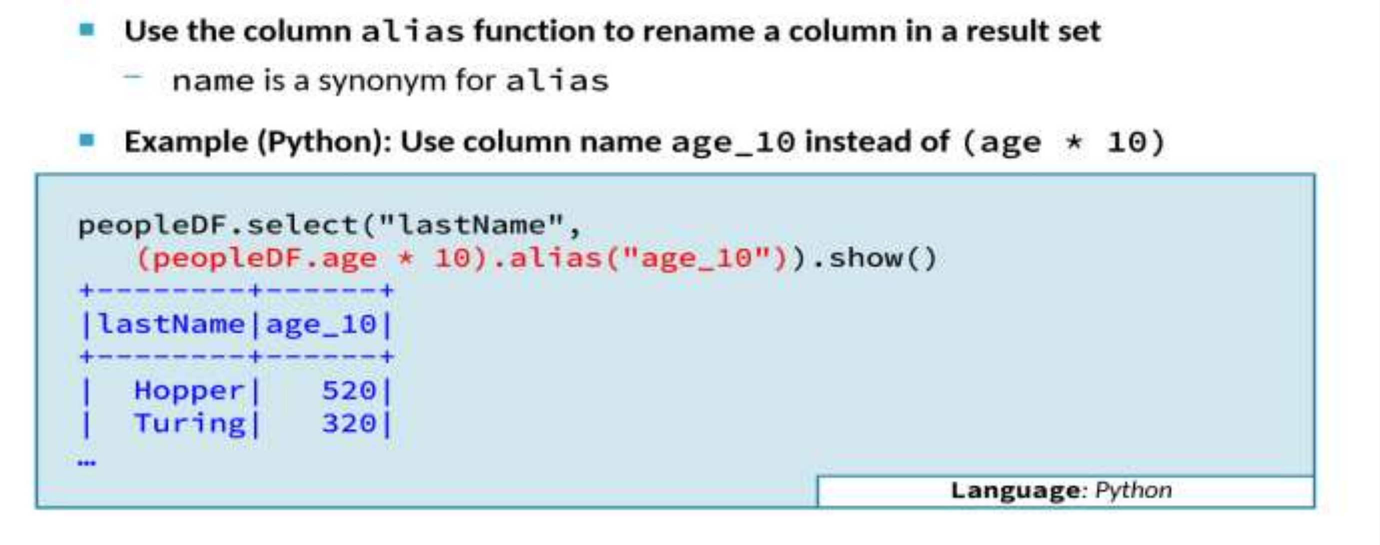
age	name	pcode
null	Alice	94304

Note: This feature does not depend on Hive or Impala, or on a database



```
df.createOrReplaceTempView("table1")
df2 = spark.sql("SELECT field1 AS f1, field2 as f2 from table1")
```

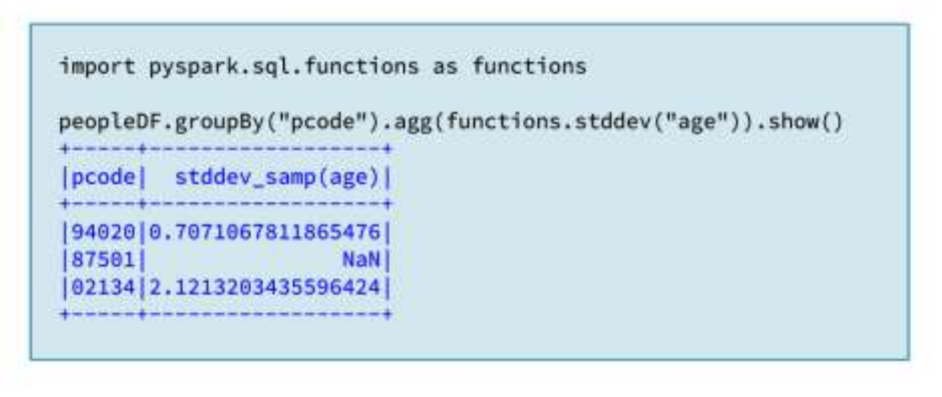
CDH – Spark - DF



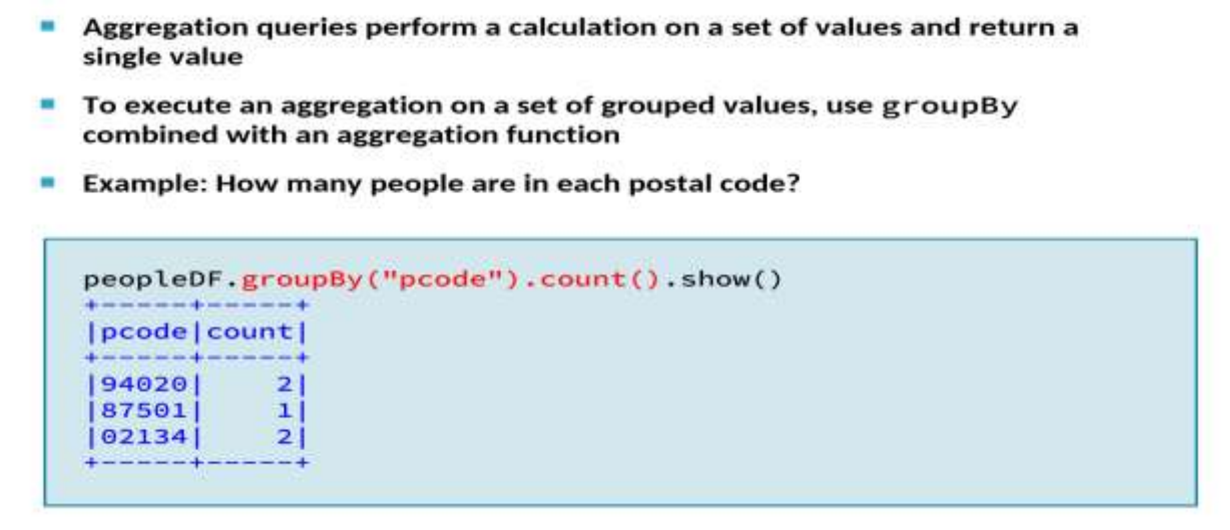
- Use the column `alias` function to rename a column in a result set
  - name is a synonym for `alias`
- Example (Python): Use column name `age_10` instead of `(age * 10)`

```
peopleDF.select("lastName",
    (peopleDF.age * 10).alias("age_10")).show()
+--------+------+
|lastName|age_10|
+--------+------+
|  Hopper|   520|
|  Turing|   320|
…
```

Language: *Python*



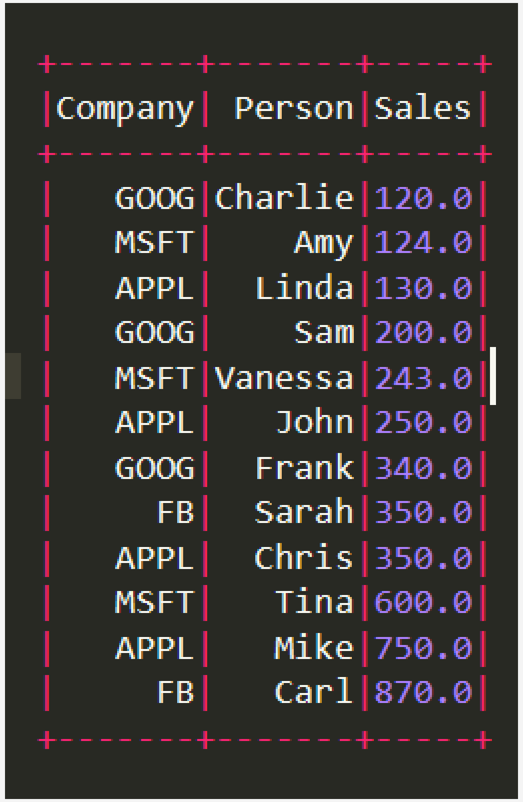
```
import pyspark.sql.functions as functions

peopleDF.groupBy("pcode").agg(functions.stddev("age")).show()
+-----+------------------+
|pcode|  stddev_samp(age)|
+-----+------------------+
|94020|0.7071067811865476|
|87501|               NaN|
|02134|2.1213203435596424|
+-----+------------------+
```



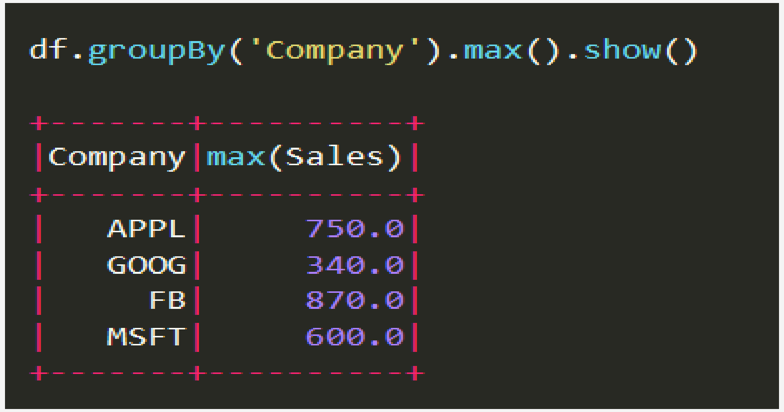
- Aggregation queries perform a calculation on a set of values and return a single value
- To execute an aggregation on a set of grouped values, use `groupBy` combined with an aggregation function
- Example: How many people are in each postal code?

```
peopleDF.groupBy("pcode").count().show()
+-----+-----+
|pcode|count|
+-----+-----+
|94020|    2|
|87501|    1|
|02134|    2|
+-----+-----+
```

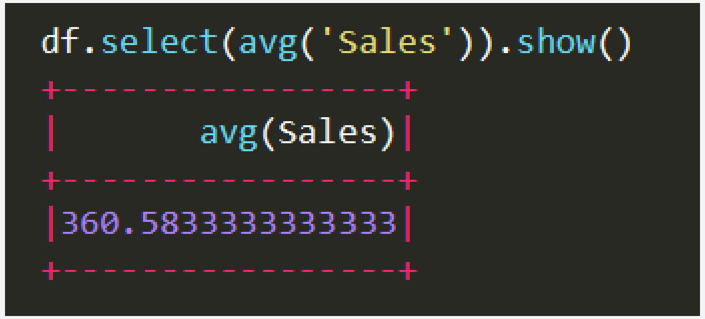
CDH – Spark – DF



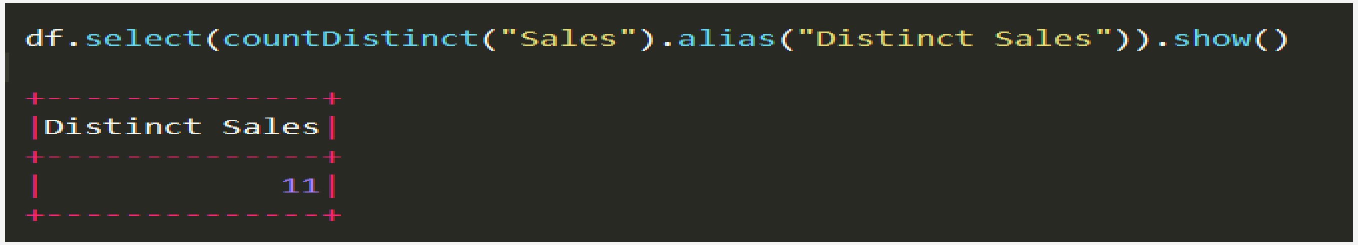
```
+-------+-------+-----+
|Company| Person|Sales|
+-------+-------+-----+
|   GOOG|Charlie|120.0|
|   MSFT|    Amy|124.0|
|   APPL|  Linda|130.0|
|   GOOG|    Sam|200.0|
|   MSFT|Vanessa|243.0|
|   APPL|   John|250.0|
|   GOOG|  Frank|340.0|
|     FB|  Sarah|350.0|
|   APPL|  Chris|350.0|
|   MSFT|   Tina|600.0|
|   APPL|   Mike|750.0|
|     FB|   Carl|870.0|
+-------+-------+-----+
```



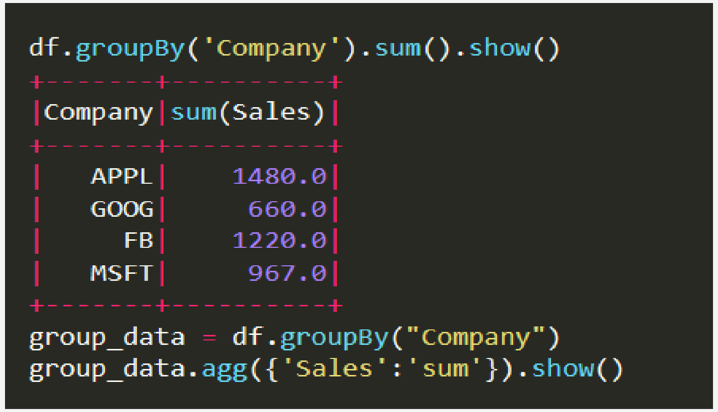
```
df.groupBy('Company').max().show()

+-------+----------+
|Company|max(Sales)|
+-------+----------+
|   APPL|     750.0|
|   GOOG|     340.0|
|     FB|     870.0|
|   MSFT|     600.0|
+-------+----------+
```



```
df.select(avg('Sales')).show()
+-----------------+
|       avg(Sales)|
+-----------------+
|360.5833333333333|
+-----------------+
```

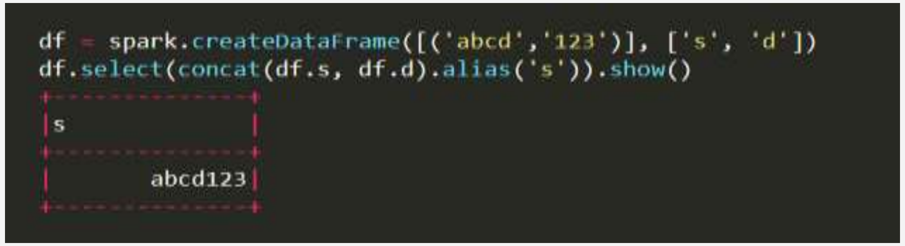


```
df.select(countDistinct("Sales").alias("Distinct Sales")).show()

+--------------+
|Distinct Sales|
+--------------+
|            11|
+--------------+
```

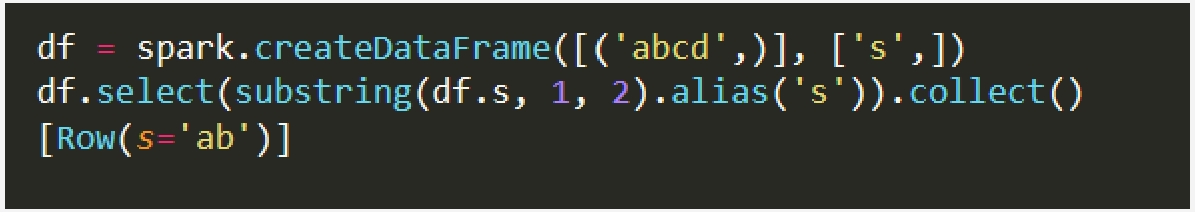


```
df.groupBy('Company').sum().show()
+-------+----------+
|Company|sum(Sales)|
+-------+----------+
|   APPL|    1480.0|
|   GOOG|     660.0|
|     FB|    1220.0|
|   MSFT|     967.0|
+-------+----------+
group_data = df.groupBy("Company")
group_data.agg({'Sales':'sum'}).show()
```

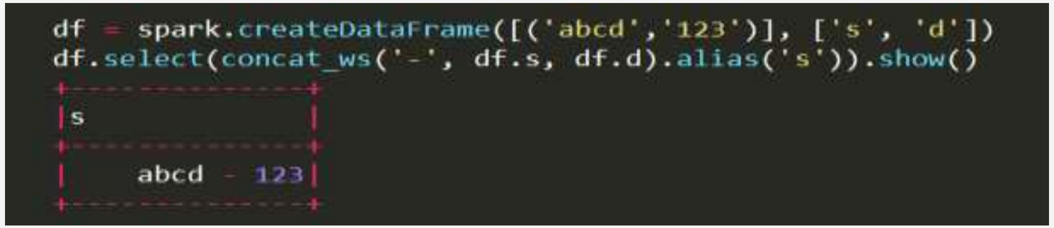
CDH - Spark - DF - Sql functions



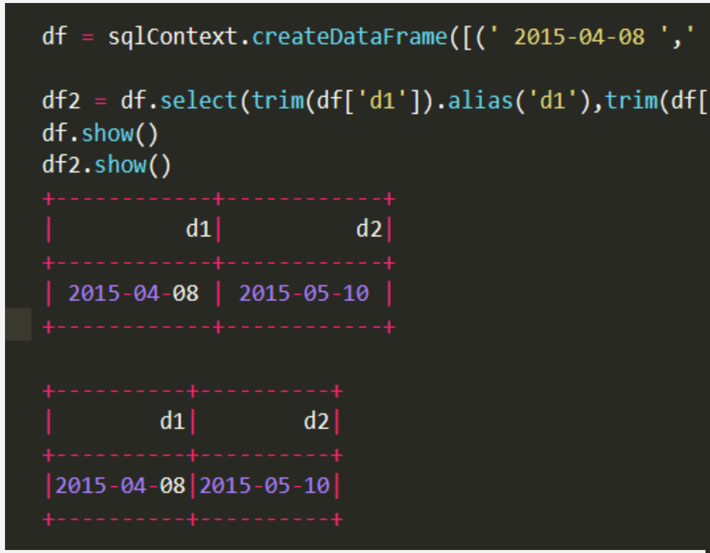
```
df = spark.createDataFrame([('abcd','123')], ['s', 'd'])
df.select(concat(df.s, df.d).alias('s')).show()
+---------------+
|s              |
+---------------+
|        abcd123|
+---------------+
```



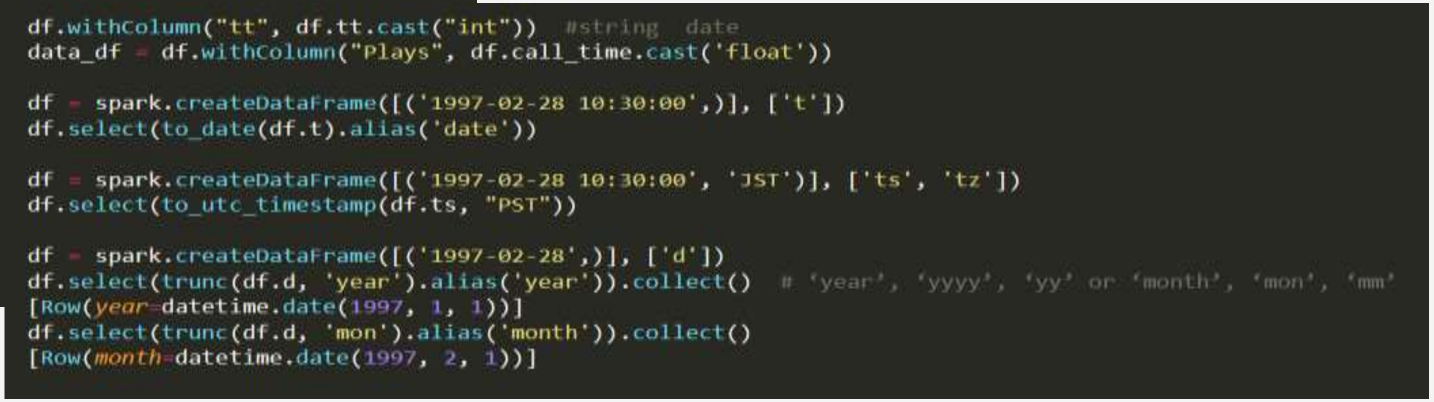
```
df = spark.createDataFrame([('abcd',)], ['s',])
df.select(substring(df.s, 1, 2).alias('s')).collect()
[Row(s='ab')]
```



```
df = spark.createDataFrame([('abcd','123')], ['s', 'd'])
df.select(concat_ws('-', df.s, df.d).alias('s')).show()
+---------------+
|s              |
+---------------+
|     abcd-123|
+---------------+
```

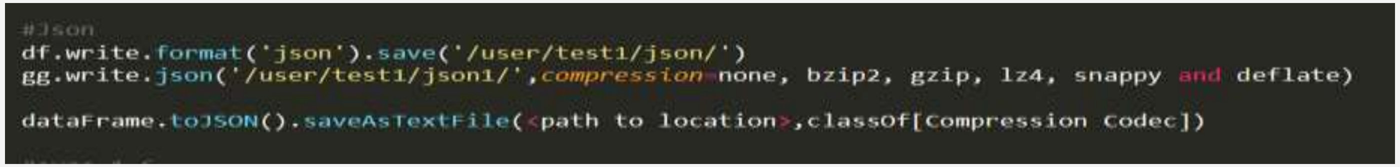


```
df = sqlContext.createDataFrame([(' 2015-04-08 ',' 2015-05-10 ')], ['d1', 'd2'])

df2 = df.select(trim(df['d1']).alias('d1'),trim(df['d2']).alias('d2'))
df.show()
df2.show()
+------------+------------+
|          d1|          d2|
+------------+------------+
| 2015-04-08 | 2015-05-10 |
+------------+------------+

+----------+----------+
|        d1|        d2|
+----------+----------+
|2015-04-08|2015-05-10|
+----------+----------+
```

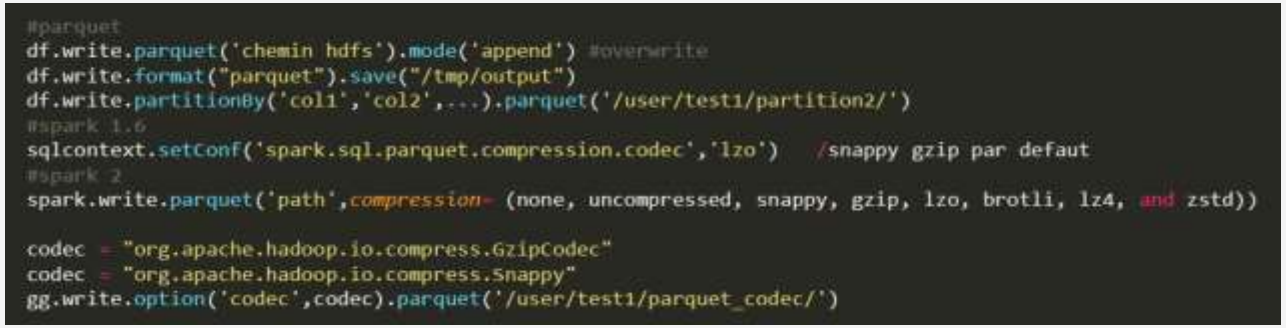


```
df.withColumn("tt", df.tt.cast("int"))  #string  date
data_df = df.withColumn("Plays", df.call_time.cast('float'))

df = spark.createDataFrame([('1997-02-28 10:30:00',)], ['t'])
df.select(to_date(df.t).alias('date'))

df = spark.createDataFrame([('1997-02-28 10:30:00', 'JST')], ['ts', 'tz'])
df.select(to_utc_timestamp(df.ts, "PST"))

df = spark.createDataFrame([('1997-02-28',)], ['d'])
df.select(trunc(df.d, 'year').alias('year')).collect()  # 'year', 'yyyy', 'yy' or 'month', 'mon', 'mm'
[Row(year=datetime.date(1997, 1, 1))]
df.select(trunc(df.d, 'mon').alias('month')).collect()
[Row(month=datetime.date(1997, 2, 1))]
```

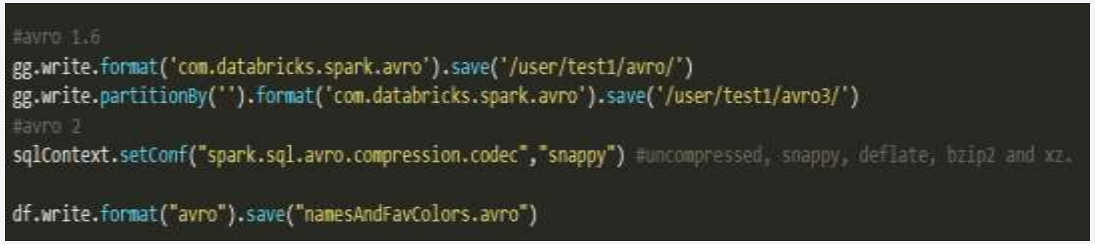
CDH – Spark– DF – Write files



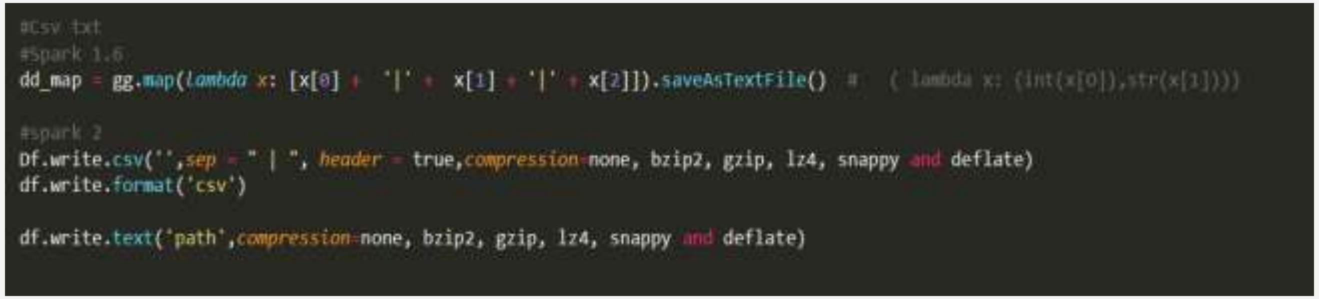
```
#Json
df.write.format('json').save('/user/test1/json/')
gg.write.json('/user/test1/json1/',compression=none, bzip2, gzip, lz4, snappy and deflate)

dataFrame.toJSON().saveAsTextFile(<path to location>,classOf[Compression Codec])
```



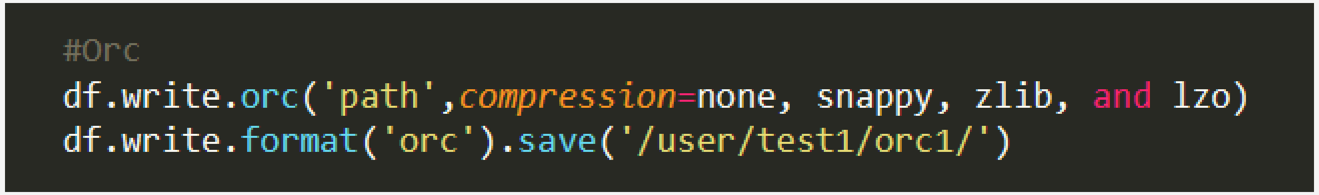
```
#parquet
df.write.parquet('chemin hdfs').mode('append') #overwrite
df.write.format("parquet").save("/tmp/output")
df.write.partitionBy('col1','col2',...).parquet('/user/test1/partition2/')
#spark 1.6
sqlcontext.setConf('spark.sql.parquet.compression.codec','lzo')   /snappy gzip par defaut
#spark 2
spark.write.parquet('path',compression= (none, uncompressed, snappy, gzip, lzo, brotli, lz4, and zstd))

codec = "org.apache.hadoop.io.compress.GzipCodec"
codec = "org.apache.hadoop.io.compress.Snappy"
gg.write.option('codec',codec).parquet('/user/test1/parquet_codec/')
```



```
#avro 1.6
gg.write.format('com.databricks.spark.avro').save('/user/test1/avro/')
gg.write.partitionBy('').format('com.databricks.spark.avro').save('/user/test1/avro3/')
#avro 2
sqlContext.setConf("spark.sql.avro.compression.codec","snappy") #uncompressed, snappy, deflate, bzip2 and xz.

df.write.format("avro").save("namesAndFavColors.avro")
```

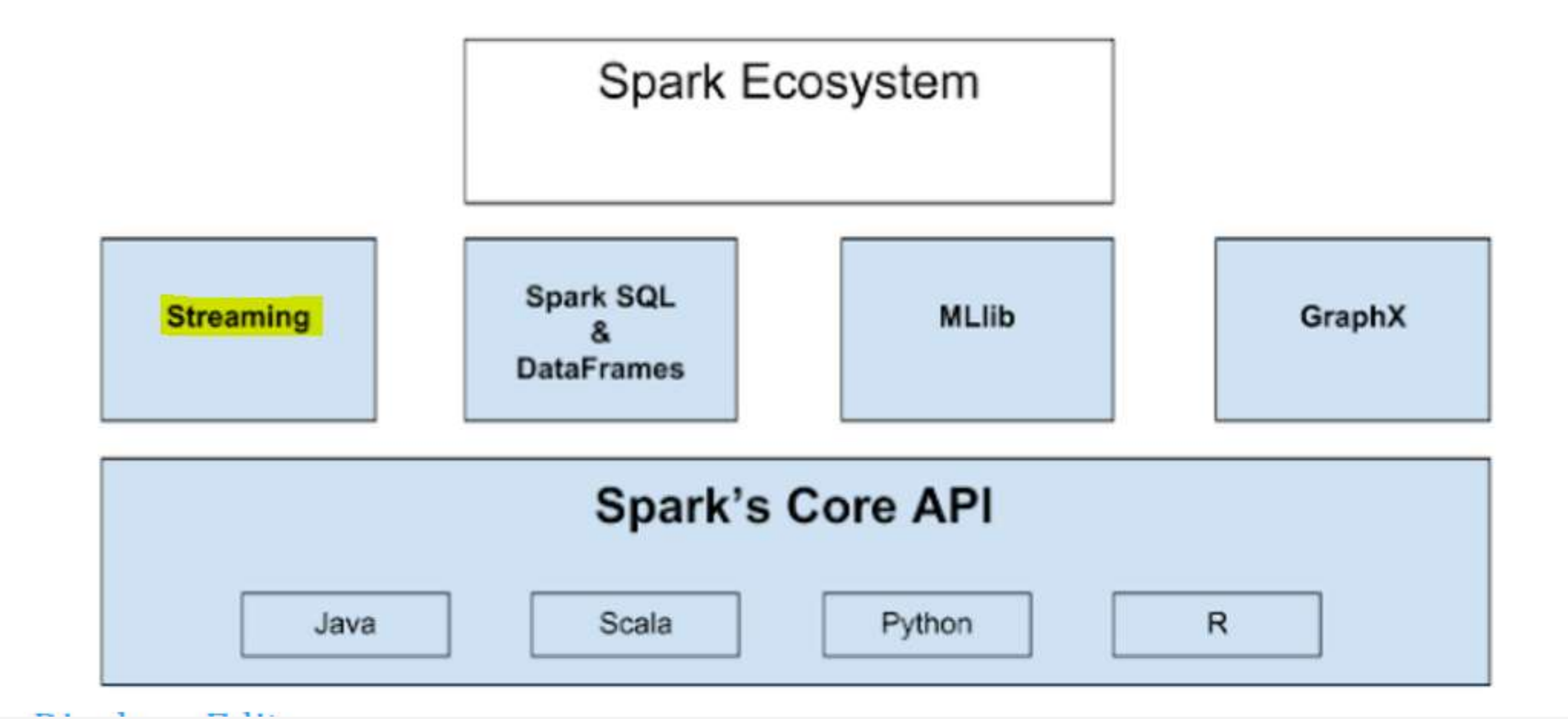


```
#Csv txt
#Spark 1.6
dd_map = gg.map(lambda x: [x[0] +  '|' +  x[1] + '|' + x[2]]).saveAsTextFile()  #  ( lambda x: (int(x[0]),str(x[1])))

#spark 2
Df.write.csv('',sep = " | ", header = true,compression=none, bzip2, gzip, lz4, snappy and deflate)
df.write.format('csv')

df.write.text('path',compression=none, bzip2, gzip, lz4, snappy and deflate)
```

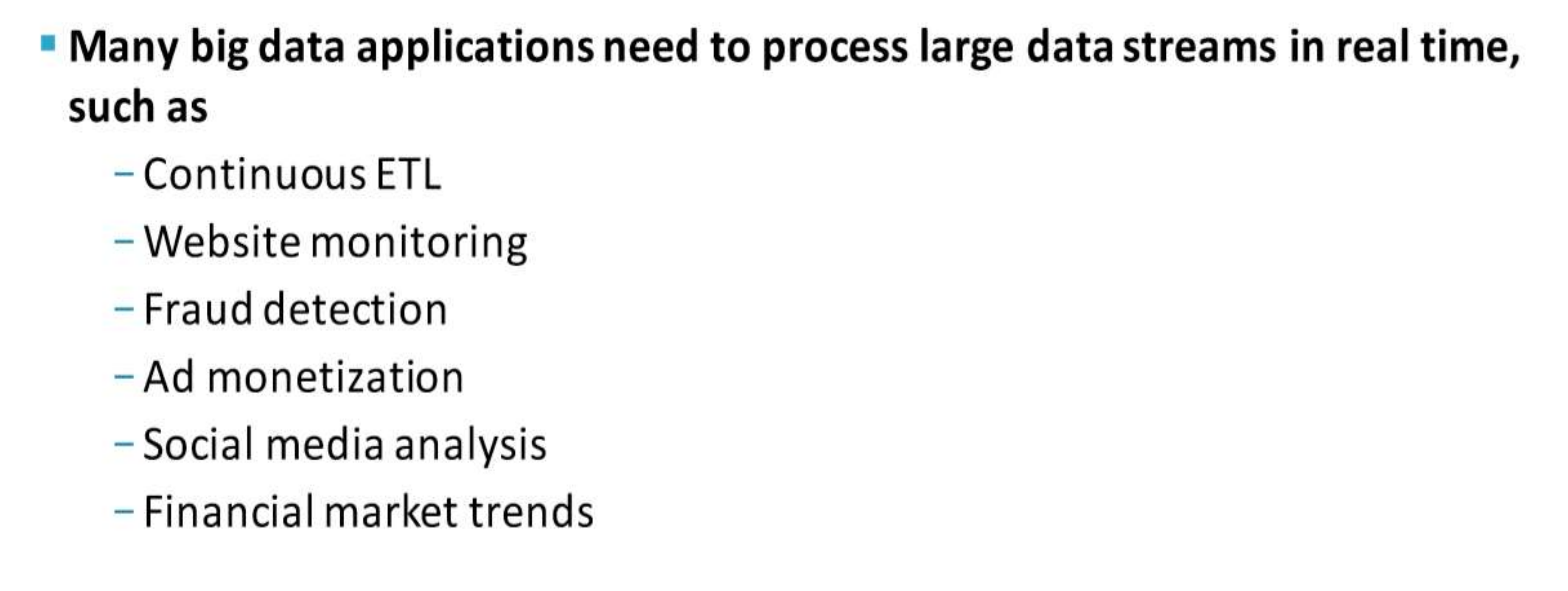


```
#Orc
df.write.orc('path',compression=none, snappy, zlib, and lzo)
df.write.format('orc').save('/user/test1/orc1/')
```

CDH – Spark– Streaming

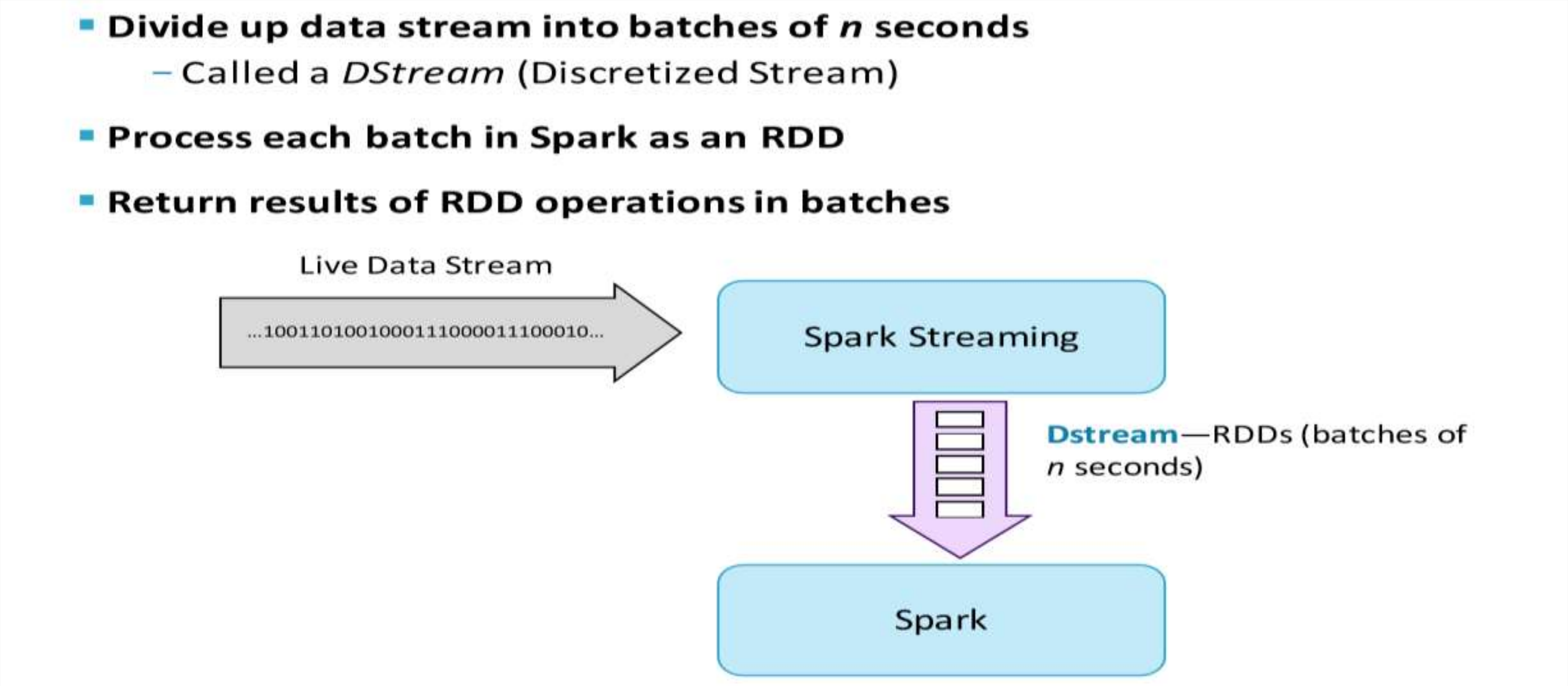


CDH – Spark– Streaming



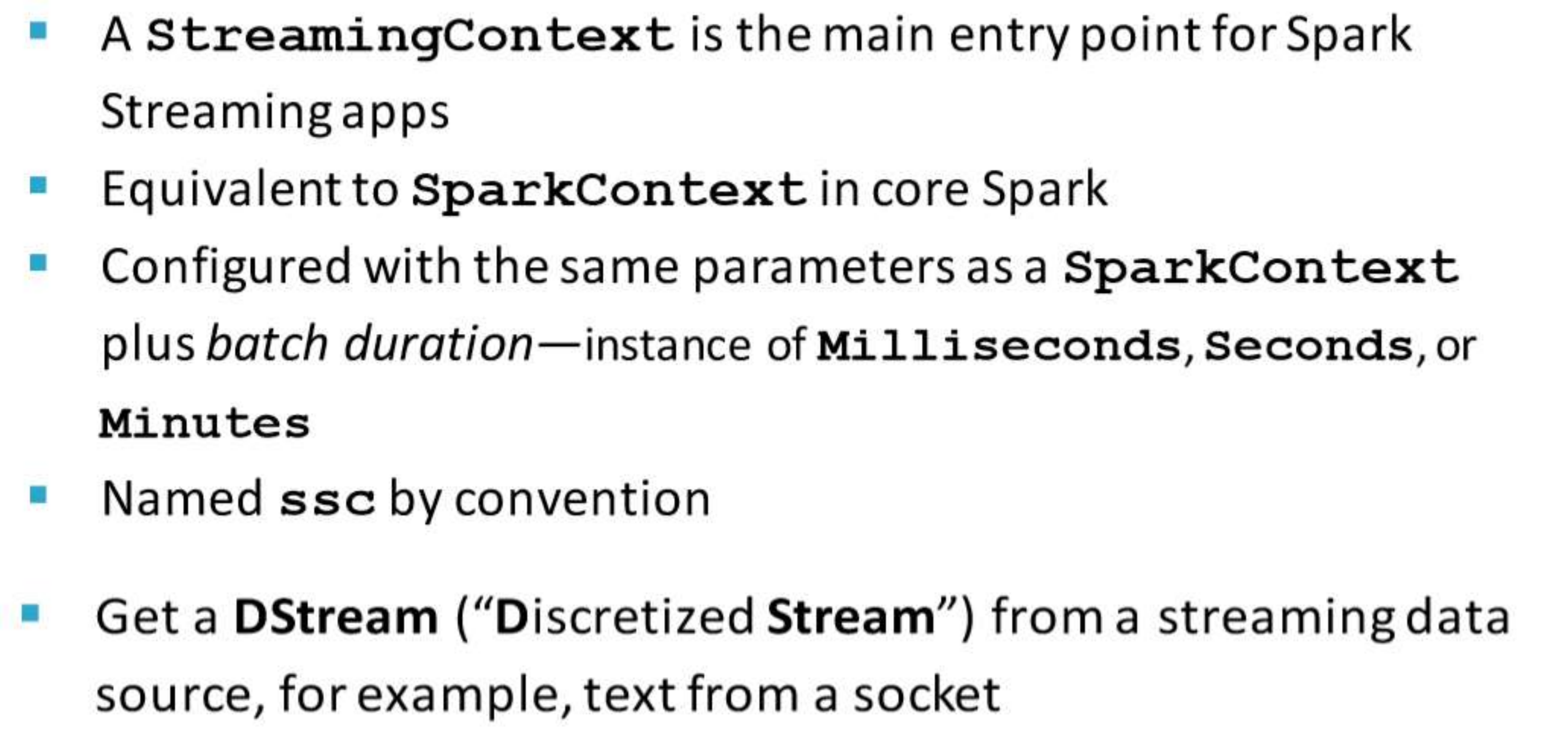
- **Many big data applications need to process large data streams in real time, such as**
  - Continuous ETL
  - Website monitoring
  - Fraud detection
  - Ad monetization
  - Social media analysis
  - Financial market trends

CDH – Spark– Streaming



- **Divide up data stream into batches of *n* seconds**
  - Called a *DStream* (Discretized Stream)
- **Process each batch in Spark as an RDD**
- **Return results of RDD operations in batches**

CDH – Spark– Streaming

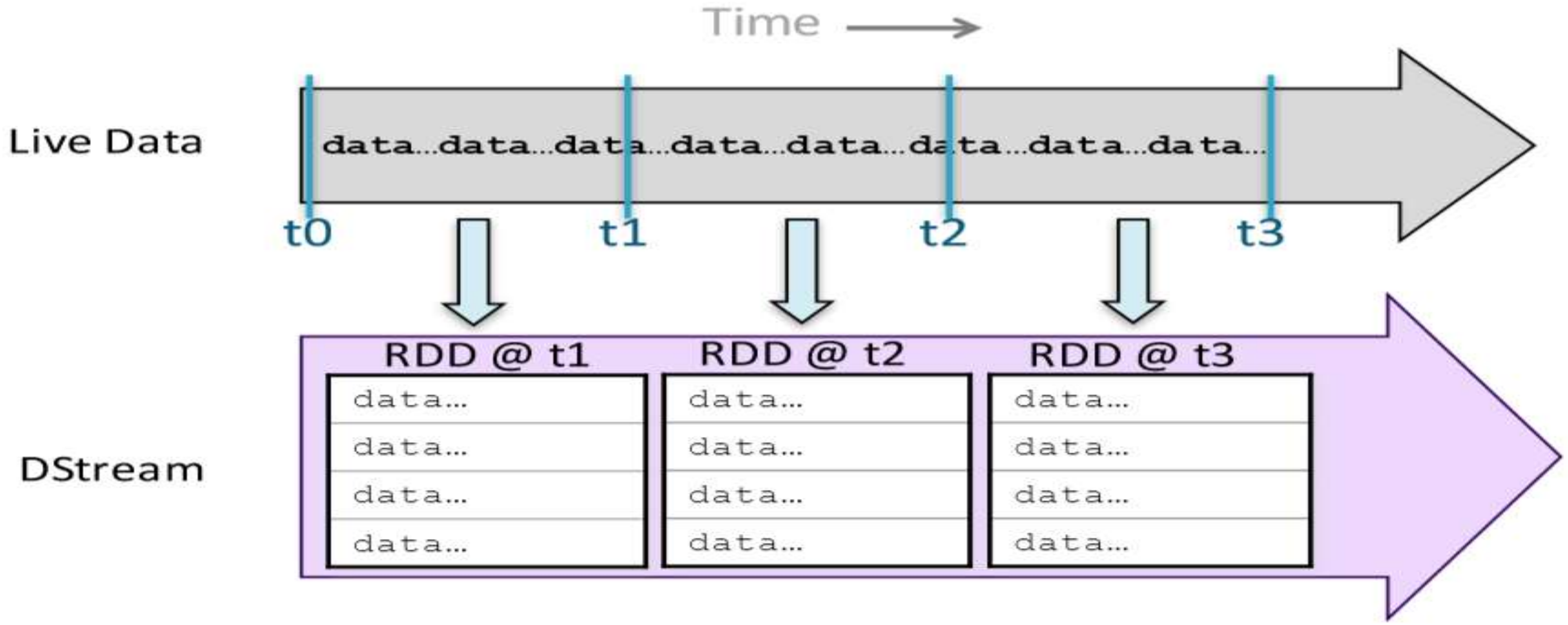


- A **StreamingContext** is the main entry point for Spark Streaming apps
- Equivalent to **SparkContext** in core Spark
- Configured with the same parameters as a **SparkContext** plus *batch duration*—instance of **Milliseconds**, **Seconds**, or **Minutes**
- Named **ssc** by convention
- Get a **DStream** (“**D**iscretized **Stream**”) from a streaming data source, for example, text from a socket

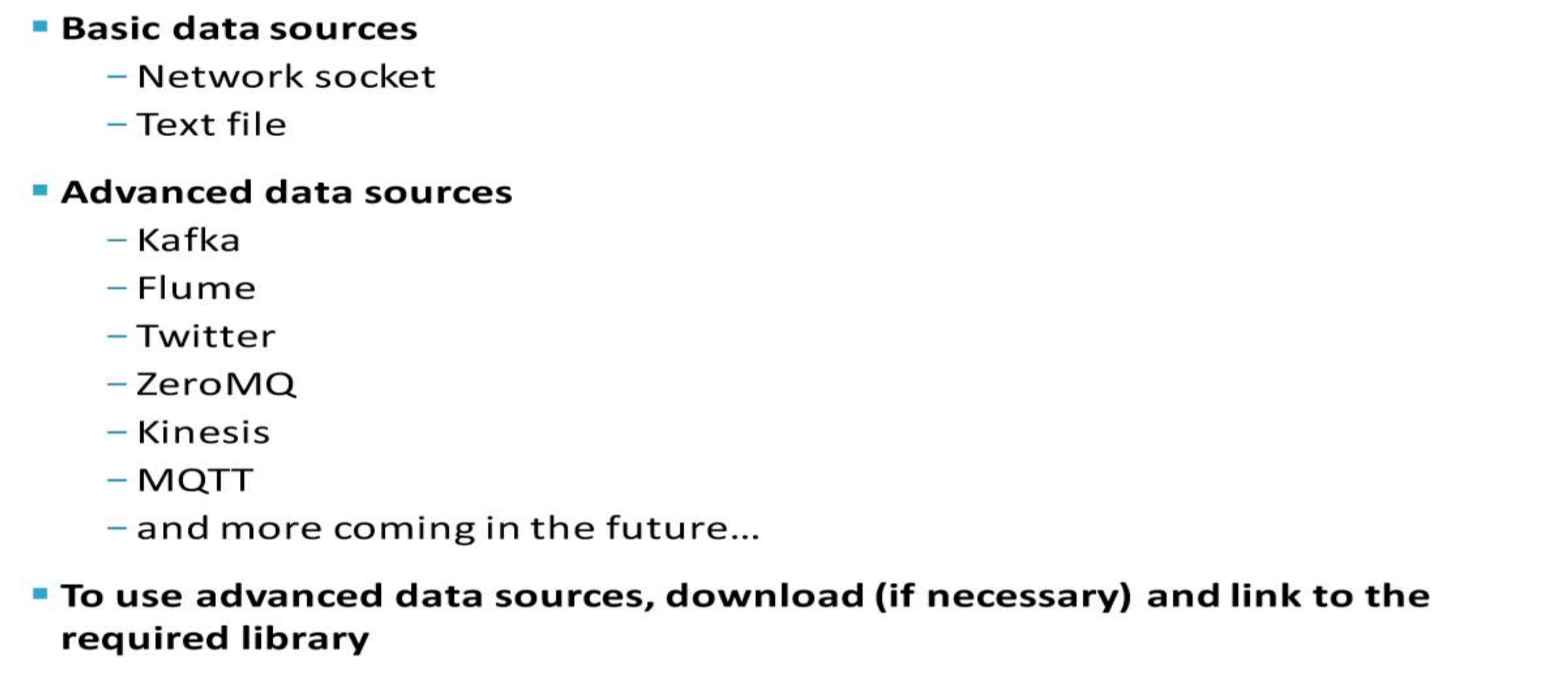
CDH – Spark – Streaming

- DStream operations are applied to each batch RDD in the stream
- Similar to RDD operations—`filter`, `map`, `reduce`, `joinByKey`, and so on.

- **A DStream is a sequence of RDDs representing a data stream**

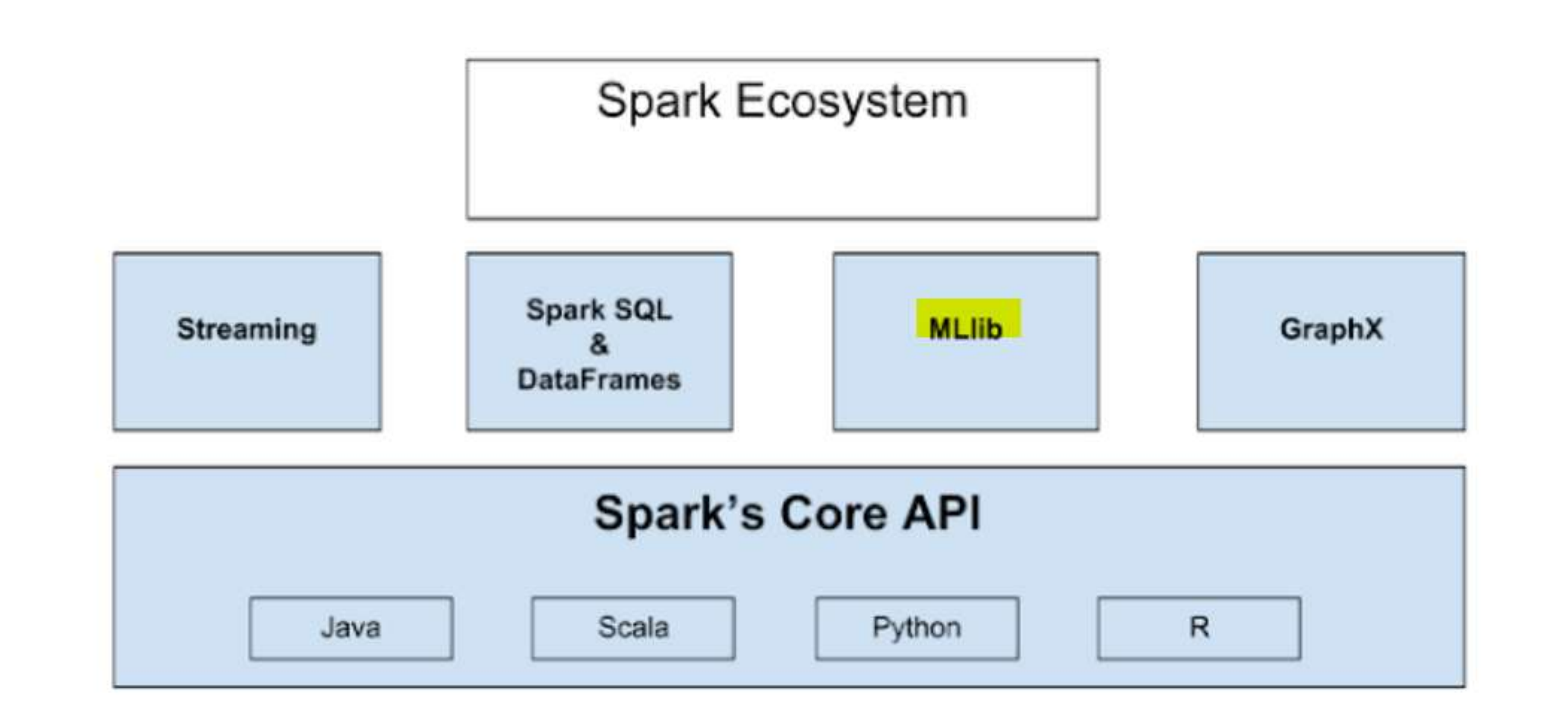


CDH – Spark– Streaming



- **Basic data sources**
  - Network socket
  - Text file
- **Advanced data sources**
  - Kafka
  - Flume
  - Twitter
  - ZeroMQ
  - Kinesis
  - MQTT
  - and more coming in the future…
- **To use advanced data sources, download (if necessary) and link to the required library**

CDH – Spark– Machine learning



Spark Ecosystem

Streaming

Spark SQL
&
DataFrames

MLlib

GraphX

Spark's Core API

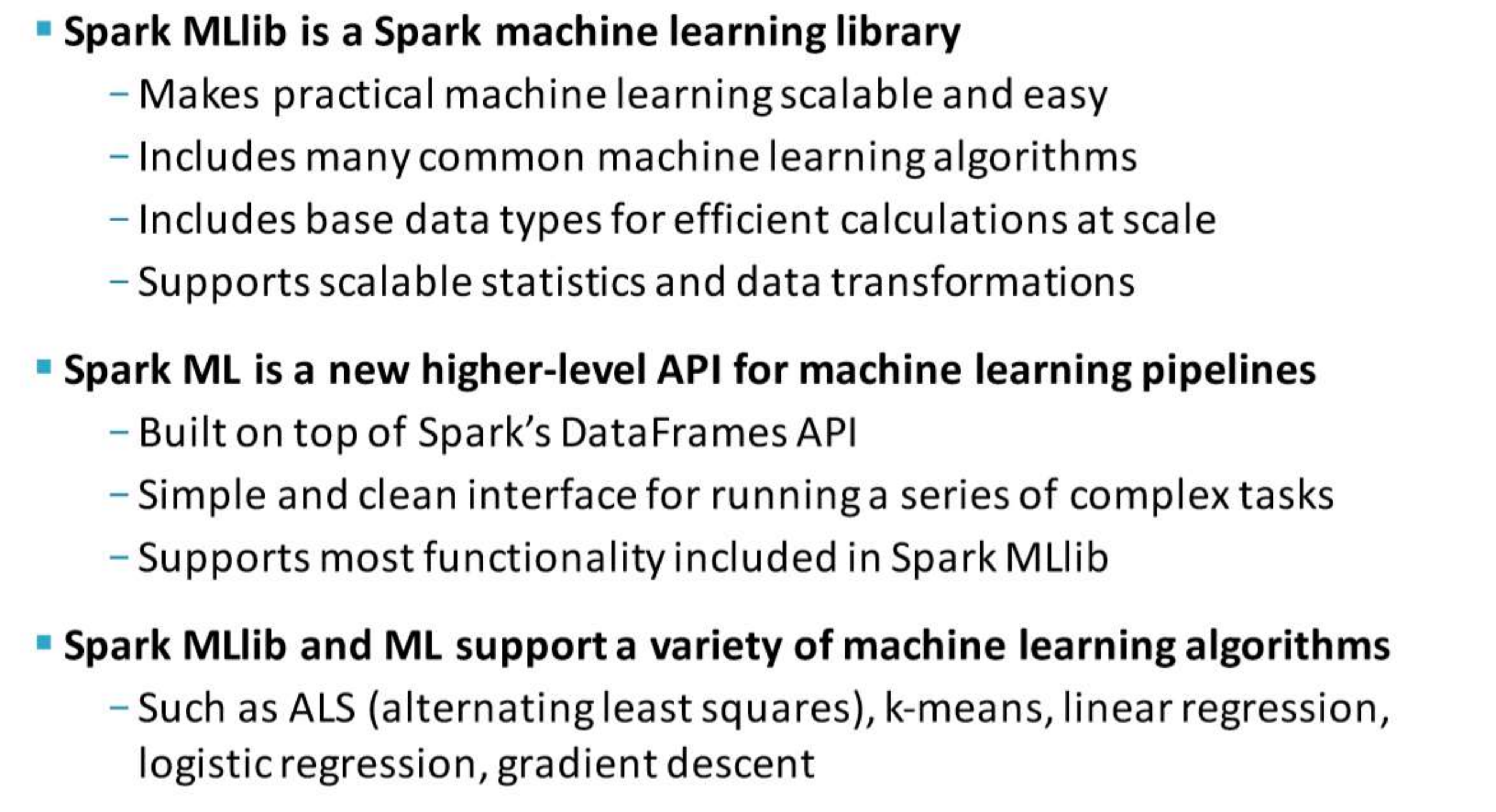
Java

Scala

Python

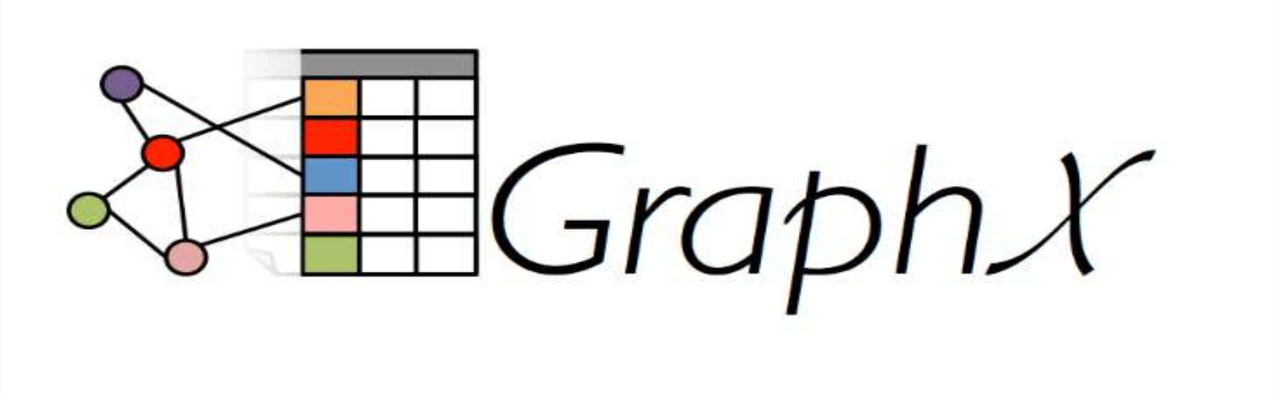
R

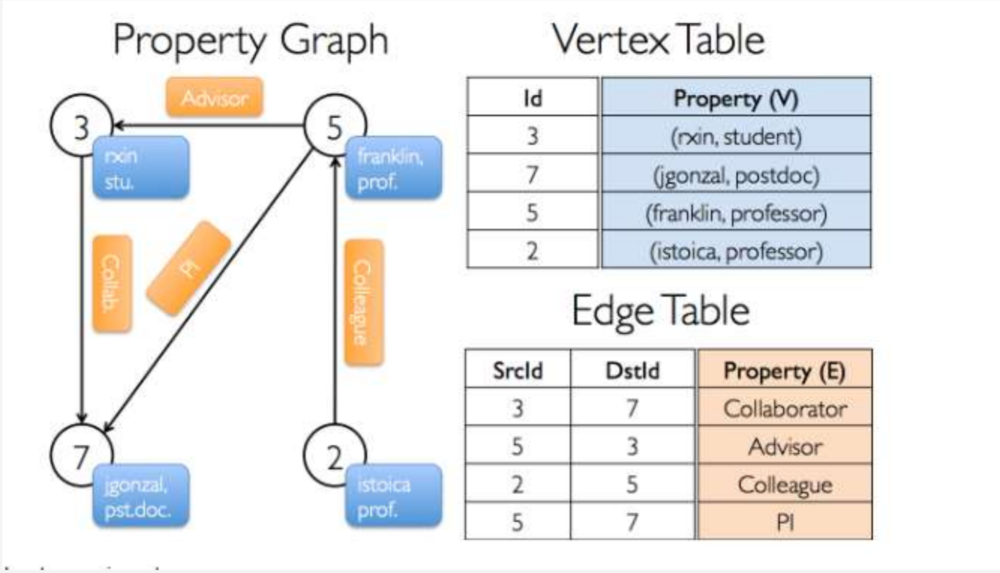
CDH – Spark– Machine learning



- **Spark MLlib is a Spark machine learning library**
  - Makes practical machine learning scalable and easy
  - Includes many common machine learning algorithms
  - Includes base data types for efficient calculations at scale
  - Supports scalable statistics and data transformations
- **Spark ML is a new higher-level API for machine learning pipelines**
  - Built on top of Spark's DataFrames API
  - Simple and clean interface for running a series of complex tasks
  - Supports most functionality included in Spark MLlib
- **Spark MLlib and ML support a variety of machine learning algorithms**
  - Such as ALS (alternating least squares), k-means, linear regression, logistic regression, gradient descent

CDH – Spark– Graphix





Vertex Table

Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

Edge Table

SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI

CDH – Spark– Best practice

Broadcast and caching data
Orc data with Hive
Repartiton or coalese
UDF -> Arrow + Pandas udf
Spark submit tuning -> core,memory,driver …

Questions ?

meritis

INFLUENT CONSULTANTS