Spark Internals

Action

- collect
- take
- show
- count
- save
- foreach

Transformation

- map
- select
- filter
- where
- group by
- join
- reduceByKey

Eager and Lazy

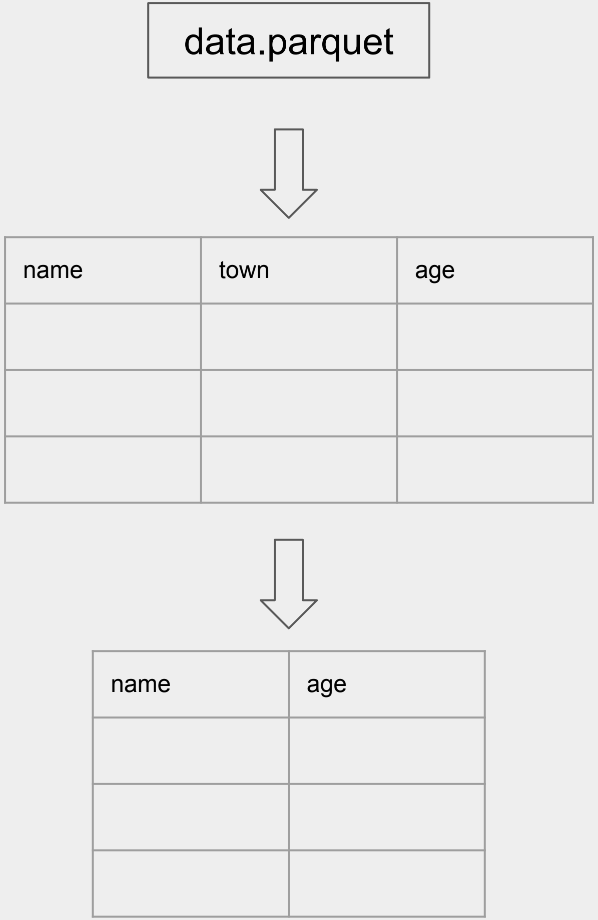
- Eager = as soon as the statement is reached in the code
- Lazy = when the result is referenced
- With Dataset
  - schema are determined eagerly
  - data transformation are executed lazily
- RDD queries are executed lazily
- No action = No data computation
- Dataset = RDD + schema

data.parquet

name	town	age

Example

```
val myDF = spark.read.parquet(“data”)
    .select(“name”,”age”)
```



Example

```
val myDF = spark.read.parquet(“data”)
    .select(“name”,”age”)
    .where(“age = 42”)
```

Xebia

Example

data.parquet

name	town	age
turring	london	21
hopper	new york	42
...	...	...

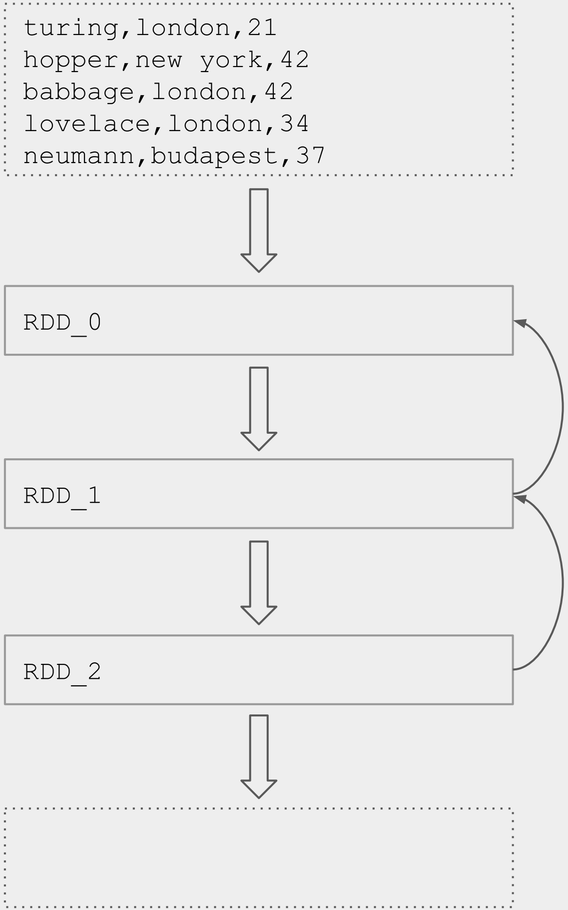
name	age
turring	21
hopper	42
...	...

```
val myDF = spark.read.parquet("data")
     .select("name","age")
     .where("age = 42")

myDF.show(2)
```

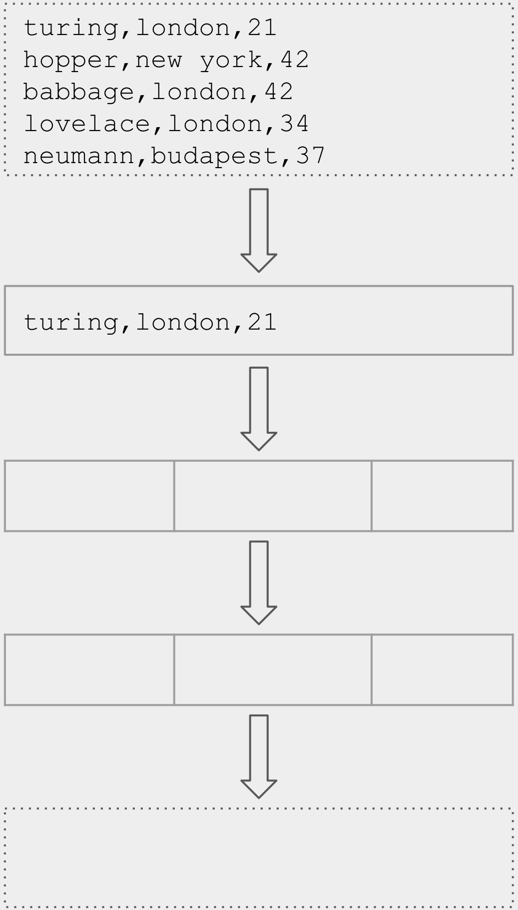
Lineage

- An RDD is a sequence of transformation
- Transformation create new RDD
  - result RDD are children
  - Child RDD depend on their parent RDD
- RDD Lineage is the sequence of parents RDD
- When an action is called, the lineage is executed starting from source (read)

Lineage



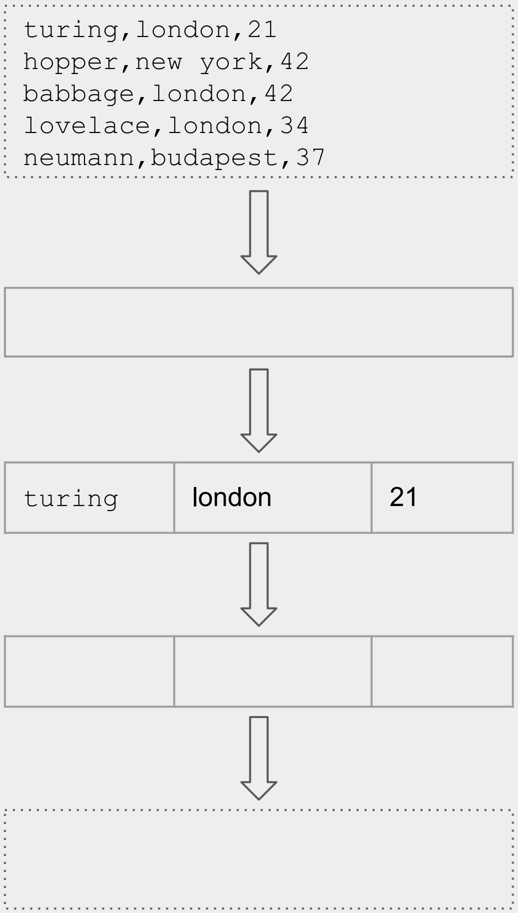

```
val myRDD = sc.textFile(“data.csv”)    RDD 0
     .map( row => row.split(“,”))      RDD 1
     .filter(row => row(2) == 42)      RDD 2

myRDD.take(2)
```



Pipelining

spark perform sequences of transformation row by row

```
val myRDD = sc.textFile("data.csv")
     .map( row => row.split(","))
     .filter(row => row(2) == 42)

myRDD.take(2)
```

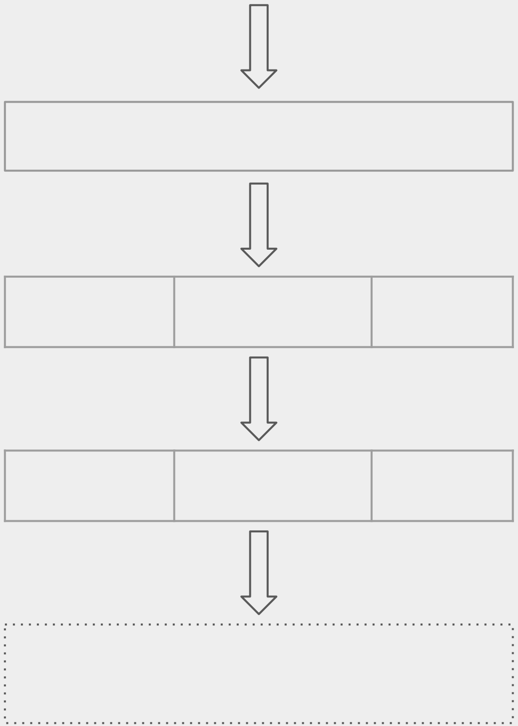


Pipelining

spark perform sequences of transformation row by row

```
val myRDD = sc.textFile("data.csv")
     .map( row => row.split(","))
     .filter(row => row(2) == 42)

myRDD.take(2)
```

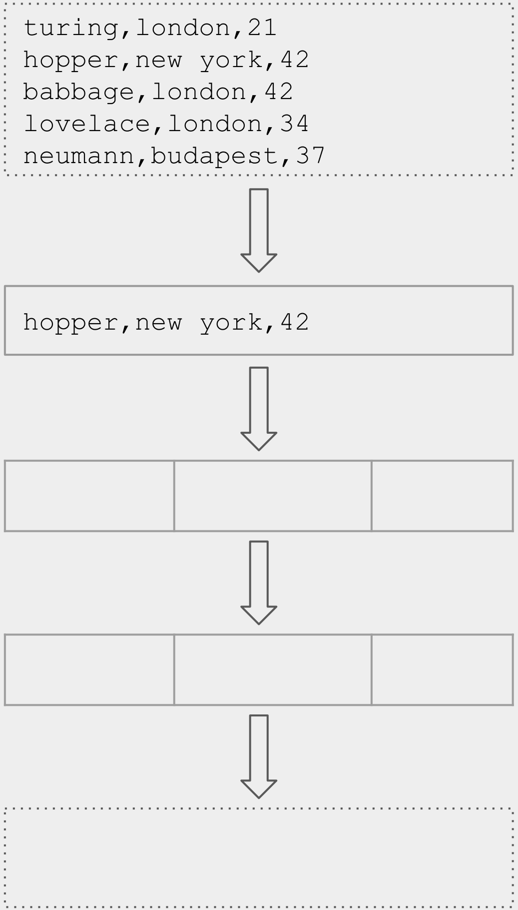
```
turing,london,21
hopper,new york,42
babbage,london,42
lovelace,london,34
neumann,budapest,37
```



Pipelining

spark perform sequences of transformation row by row

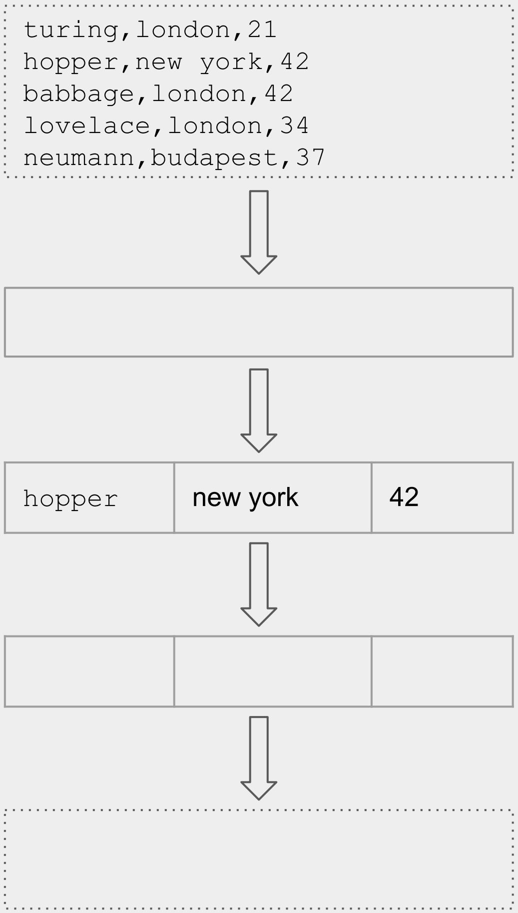
```
val myRDD = sc.textFile("data.csv")
     .map( row => row.split(","))
     .filter(row => row(2) == 42)

myRDD.take(2)
```



Pipelining

spark perform sequences of transformation row by row

```
val myRDD = sc.textFile("data.csv")
      .map( row => row.split(","))
      .filter(row => row(2) == 42)

myRDD.take(2)
```

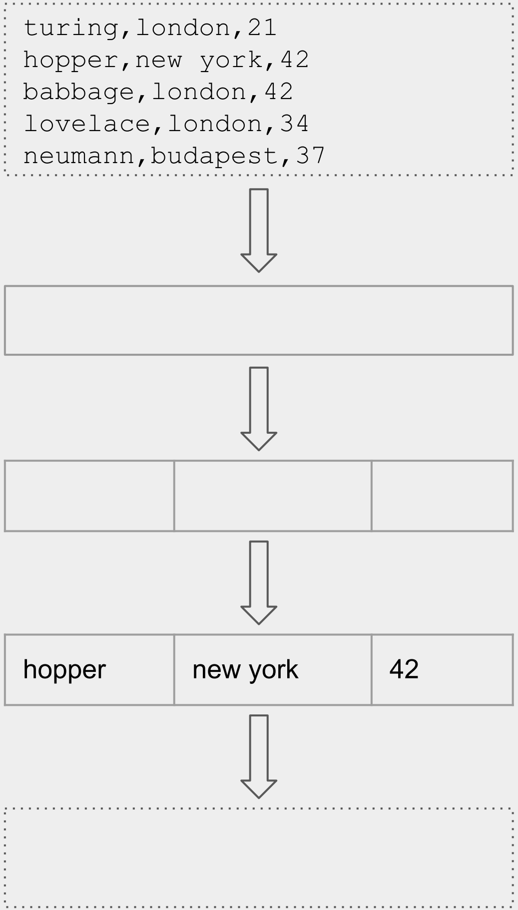


Pipelining

spark perform sequences of transformation row by row

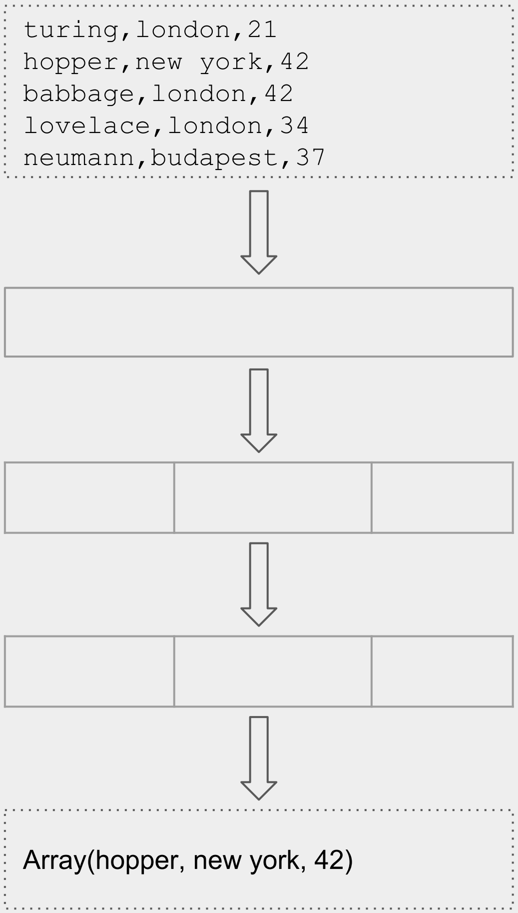
```
val myRDD = sc.textFile("data.csv")
     .map( row => row.split(","))
     .filter(row => row(2) == 42)

myRDD.take(2)
```

Pipelining



spark perform sequences of transformation row by row

```
val myRDD = sc.textFile("data.csv")
     .map( row => row.split(","))
     .filter(row => row(2) == 42)

myRDD.take(2)
```



Pipelining

spark perform sequences of transformation row by row

```
val myRDD = sc.textFile(“data.csv”)
     .map( row => row.split(“,”))
     .filter(row => row(2) == 42)

myRDD.take(2)
```

Pipelining

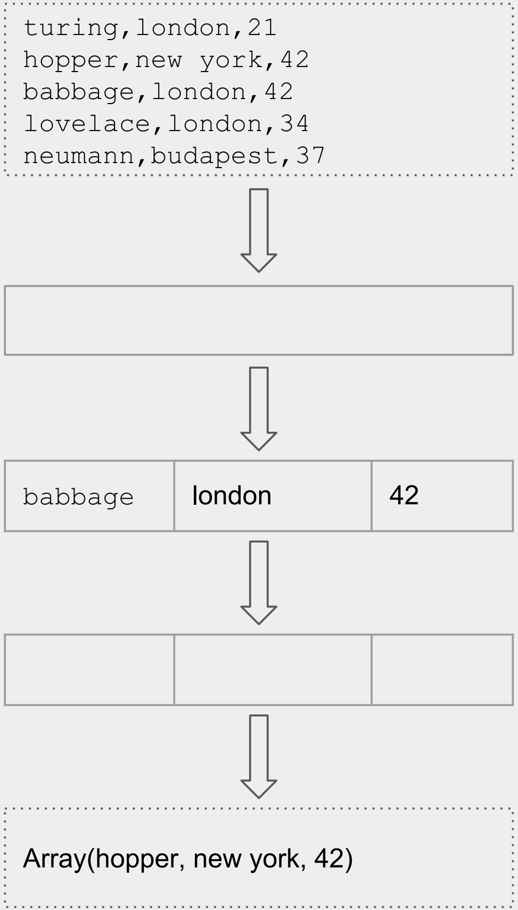
```
turing,london,21
hopper,new york,42
babbage,london,42
lovelace,london,34
neumann,budapest,37
```

```
babbage,london,42
```

Array(hopper, new york, 42)

spark perform sequences of transformation row by row

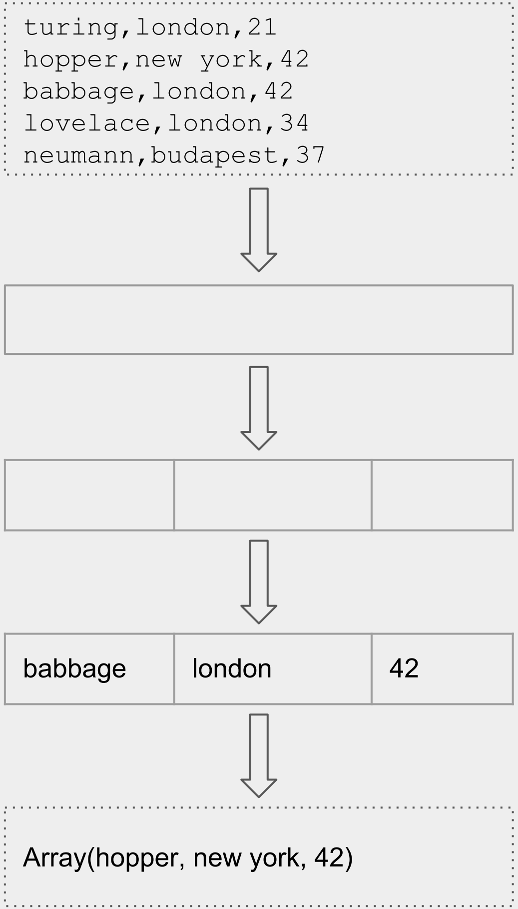
```
val myRDD = sc.textFile("data.csv")
     .map( row => row.split(","))
     .filter(row => row(2) == 42)

myRDD.take(2)
```



Pipelining

spark perform sequences of transformation row by row

```
val myRDD = sc.textFile("data.csv")
     .map( row => row.split(","))
     .filter(row => row(2) == 42)

myRDD.take(2)
```

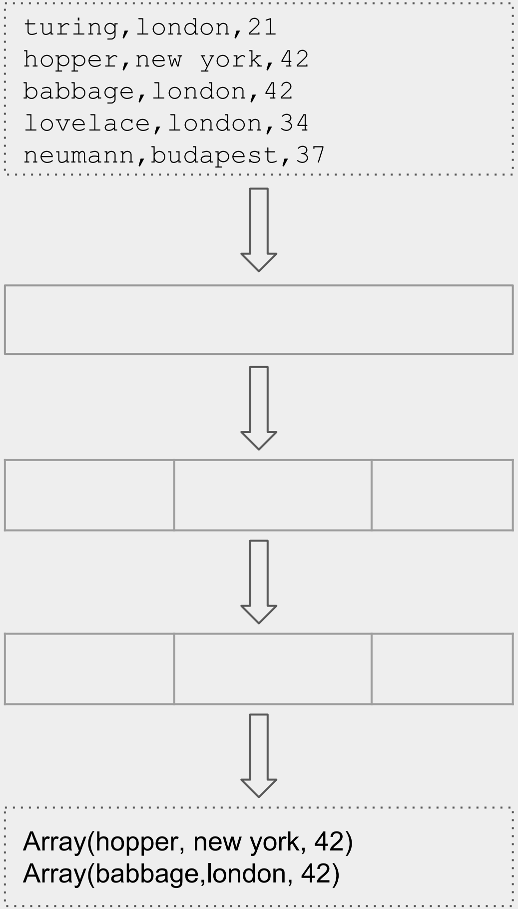


Pipelining

spark perform sequences of transformation row by row

```
val myRDD = sc.textFile("data.csv")
     .map( row => row.split(","))
     .filter(row => row(2) == 42)

myRDD.take(2)
```

Pipelining



spark perform sequences of transformation row by row

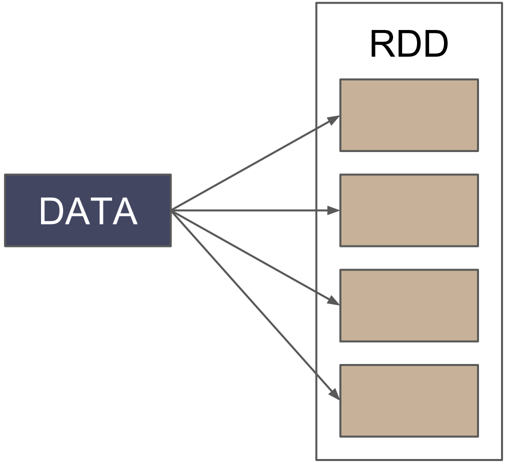
```
val myRDD = sc.textFile(“data.csv”)
     .map( row => row.split(“,”))
     .filter(row => row(2) == 42)

myRDD.take(2)
```

Partition

- Data in RDD are partitioned across executor
- Data partitioning is automatic with Dataset
- you can control partitioning with RDD
- More partitions = More parallelism

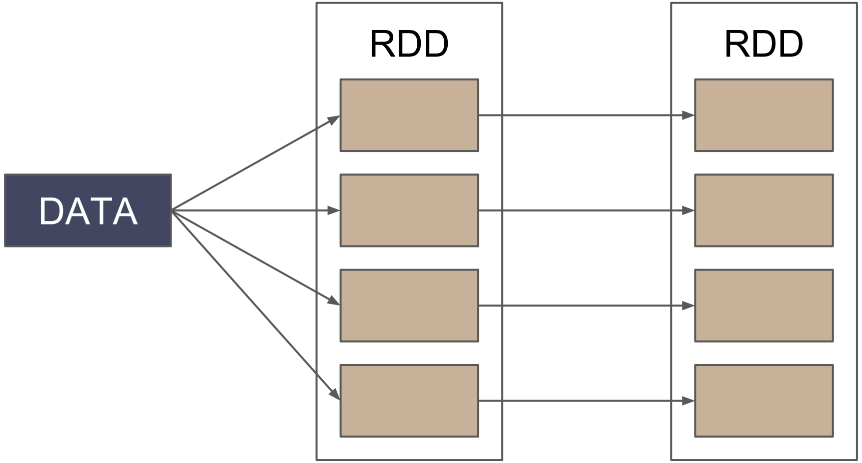
Task and Stage

```
val wc = sc.textFile(myData)
```



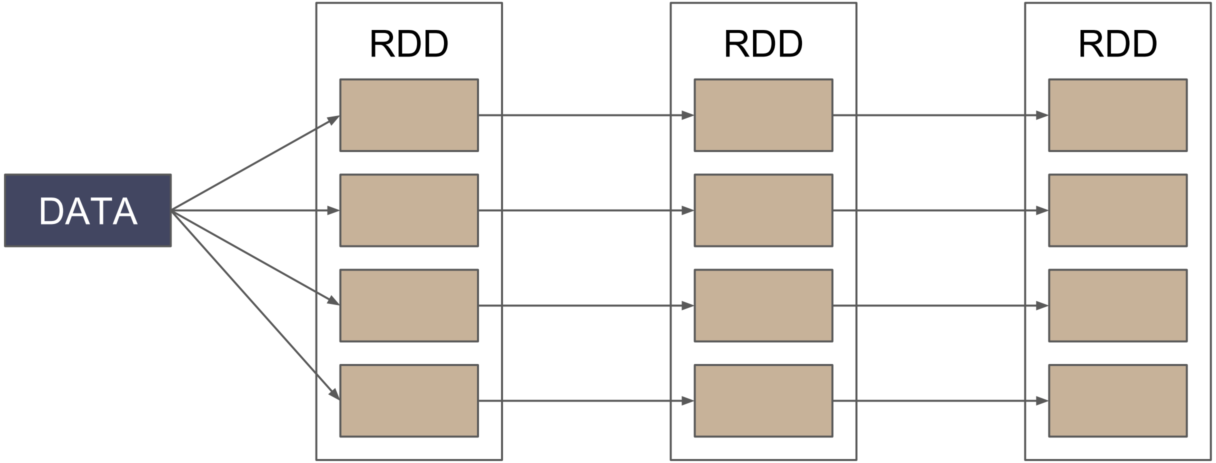
Task and Stage

```
val wc = sc.textFile(myData)
       .map(_.split(“,”))
```



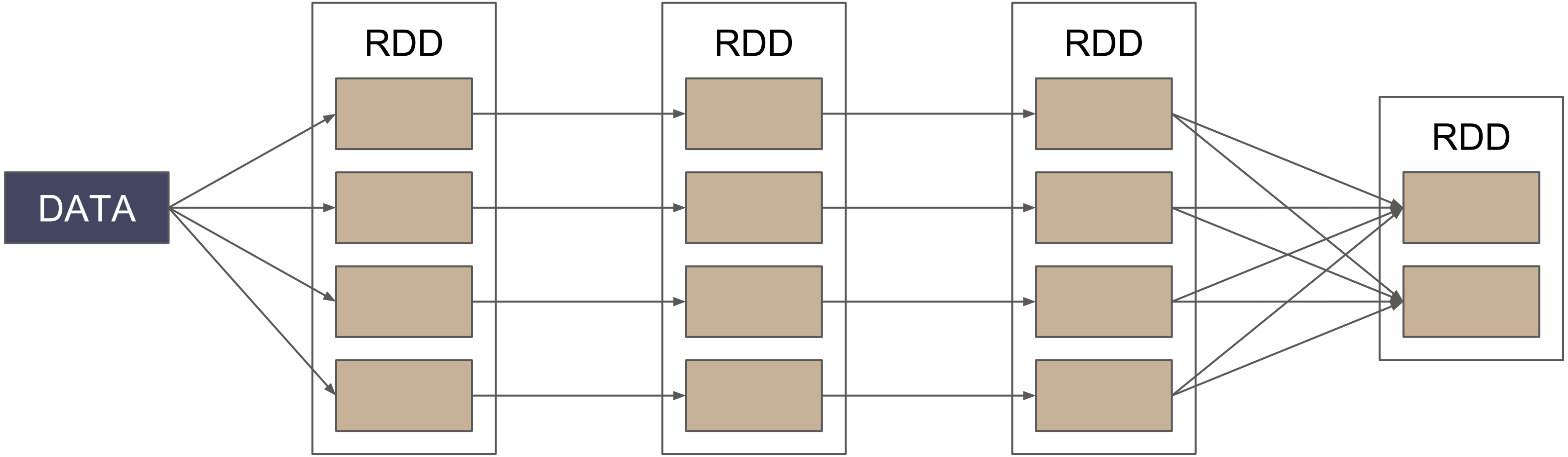
Task and Stage

```
val wc = sc.textFile(myData)
       .map(_.split(“,”))
       .map(row => (row(0), (row(14), 1))
```



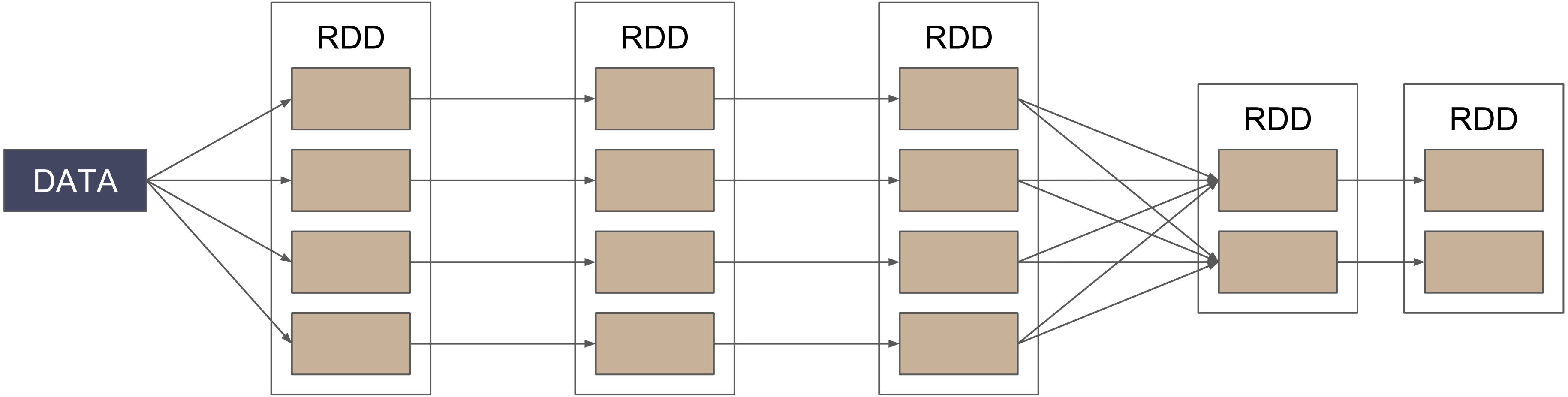
Task and Stage

```
val wc = sc.textFile(myData)
      .map(_.split(“,”))
      .map(row => (row(0), (row(14), 1))
      .reduceByKey( (x,y) => (x._1 + y._1, x._2 + y._2))
```



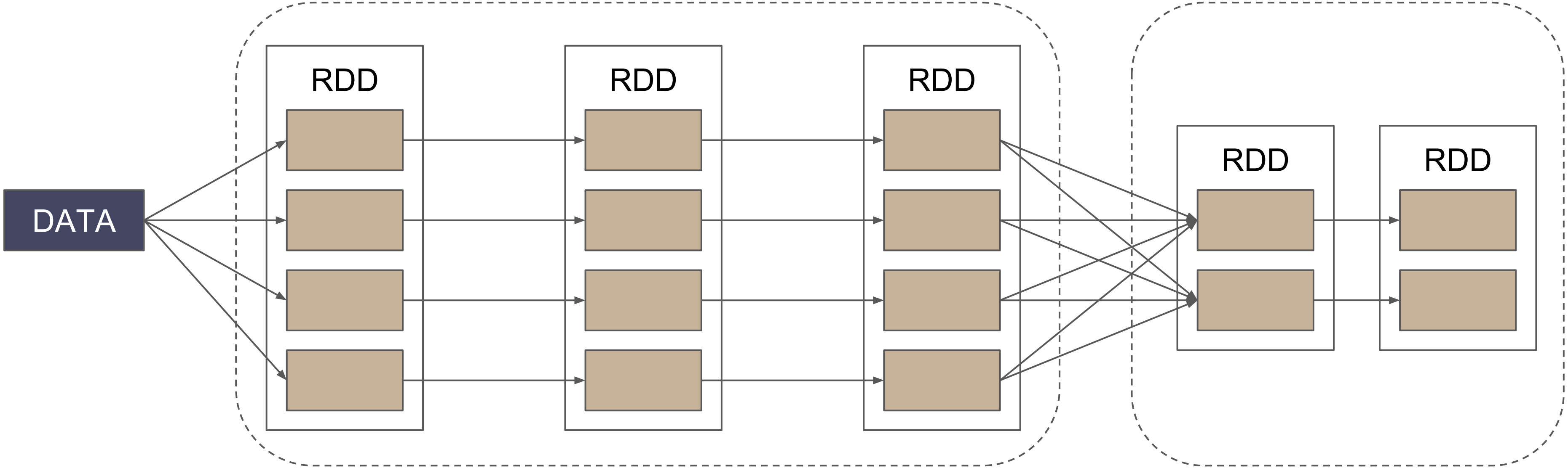
Task and Stage

```
val wc = sc.textFile(myData)
      .map(_.split(“,”))
      .map(row => (row(0), (row(14), 1))
      .reduceByKey( (x,y) => (x._1 + y._1, x._2 + y._2))
      .mapValues { case (dataSum, cpt) => dataSum/cpt }
```



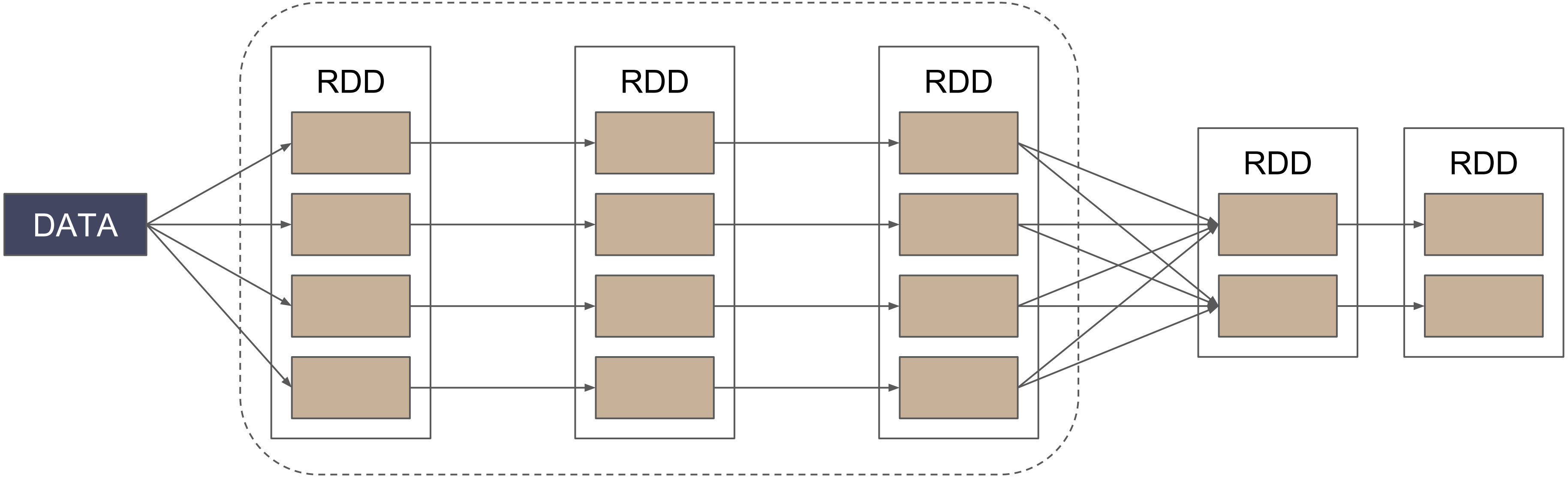
Task and Stage

```
val wc = sc.textFile(myData)
       .map(_.split(“,”))
       .map(row => (row(0), (row(14), 1))
       .reduceByKey( (x,y) => (x._1 + y._1, x._2 + y._2))
       .mapValues { case (dataSum, cpt) => dataSum/cpt }
```



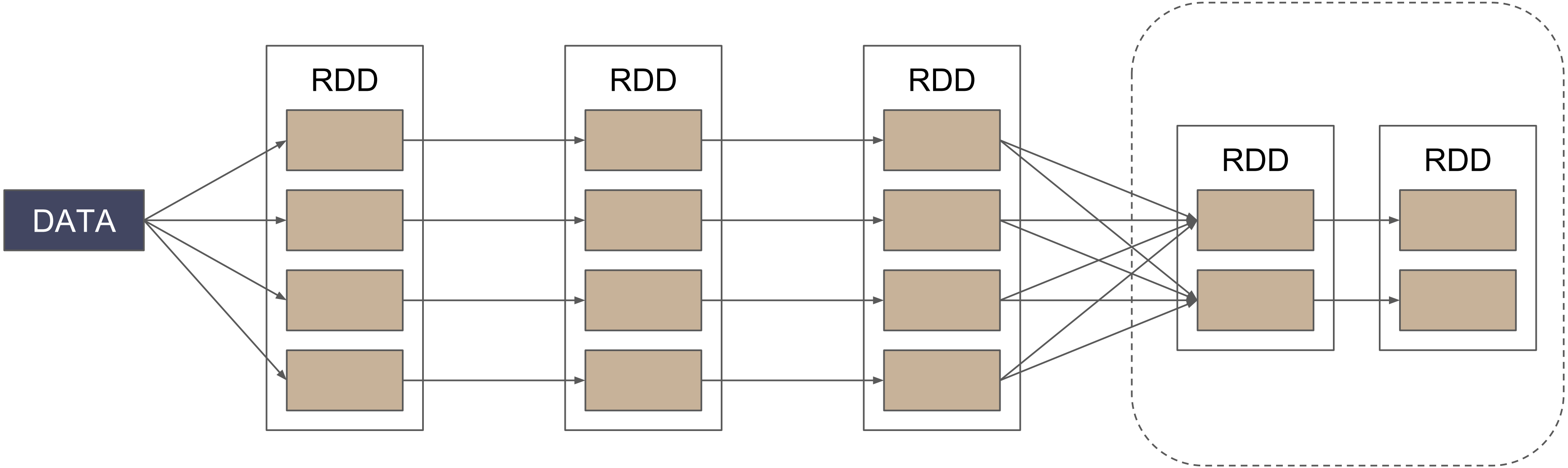
Task and Stage

```
val wc = sc.textFile(myData)
       .map(_.split(“,”))
       .map(row => (row(0), (row(14), 1))
       .reduceByKey( (x,y) => (x._1 + y._1, x._2 + y._2))
       .mapValues { case (dataSum, cpt) => dataSum/cpt }
```



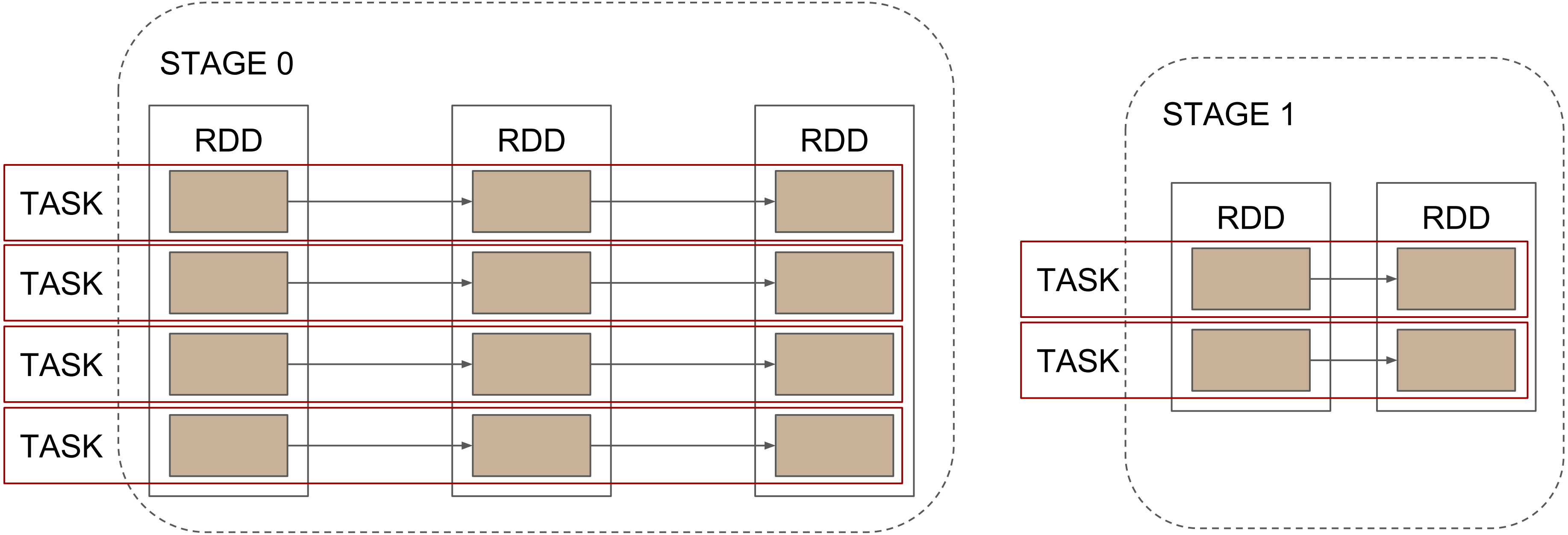
Task and Stage

```
val wc = sc.textFile(myData)
       .map(_.split(“,”))
       .map(row => (row(0), (row(14), 1))
       .reduceByKey( (x,y) => (x._1 + y._1, x._2 + y._2))
       .mapValues { case (dataSum, cpt) => dataSum/cpt }
```



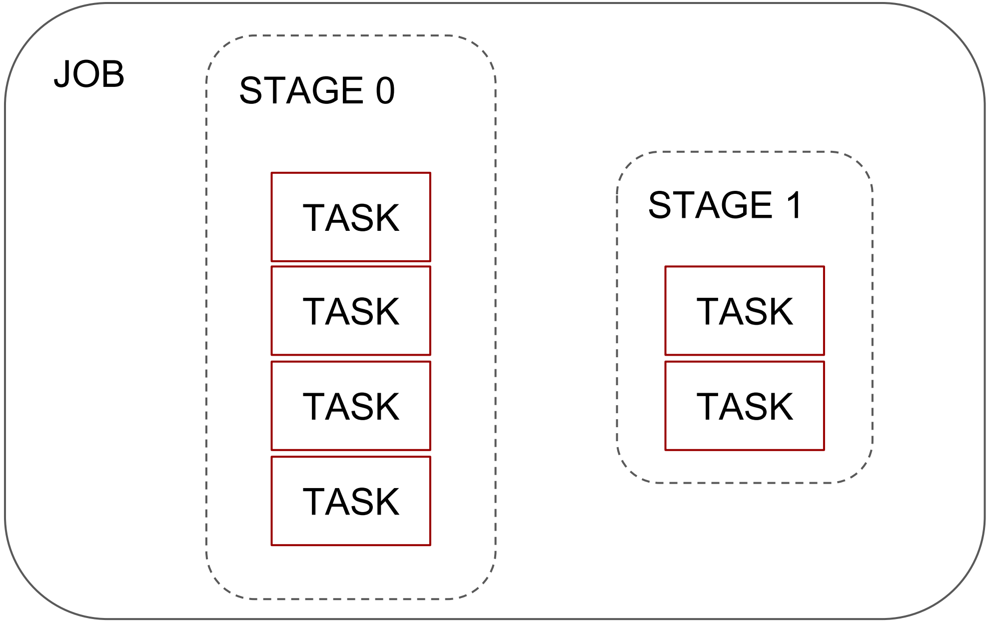
Task and Stage

```
val wc = sc.textFile(myData)
       .map(_.split(“,”))
       .map(row => (row(0), (row(14), 1))
       .reduceByKey( (x,y) => (x._1 + y._1, x._2 + y._2))
       .mapValues { case (dataSum, cpt) => dataSum/cpt }
```



Task and Stage

```
val wc = sc.textFile(myData)
        .map(_.split(“,”))
        .map(row => (row(0), (row(14), 1))
        .reduceByKey( (x,y) => (x._1 + y._1, x._2 + y._2))
        .mapValues { case (dataSum, cpt) => dataSum/cpt }
```



Catalyst Plans

- Parsed Logical Plan: sequence of operation describe in query
- Analyzed Logical Plan: resolve relation between column and data
- Optimized Logical Plan: rule based optimisation
- Physical Plan: actual sequence of operation
- Code Generation

Catalyst Plans

```
peopleDF.join(pcodesDF,  “pcode”).explain(True)
```

Catalyst Plans

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

```
== Parsed Logical Plan == (sequence of operation describe in query)
'Join UsingJoin(Inner,ArrayBuffer('pcode))
:- Relation[pcode#0,lastName#1,firstName#2,age#3] csv
+- Relation[pcode#9,city#10,state#11] csv
```

Catalyst Plans

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

```
== Parsed Logical Plan == (sequence of operation describe in query)
'Join UsingJoin(Inner,ArrayBuffer('pcode))
:- Relation[pcode#0,lastName#1,firstName#2,age#3] csv
+- Relation[pcode#9,city#10,state#11] csv

== Analyzed Logical Plan == (resolve relation between column and data)
pcode: string, lastName: string, firstName: string, age:string, city: string, state:
string
Project [pcode#0, lastName#1, firstName#2, age#3, city#10, state#11]
+- Join Inner, (pcode#0 = pcode#9)
  :- Relation[pcode#0,lastName#1,firstName#2,age#3] csv
  +- Relation[pcode#9,city#10,state#11] csv
```

Catalyst Plans

```
== Analyzed Logical Plan == (resolve relation between column and data)
pcode: string, lastName: string, firstName: string, age:string, city: string, state:
string
Project [pcode#0, lastName#1, firstName#2, age#3, city#10, state#11]
+- Join Inner, (pcode#0 = pcode#9)
  :- Relation[pcode#0,lastName#1,firstName#2,age#3] csv
  +- Relation[pcode#9,city#10,state#11] csv

== Optimized Logical Plan == (rule based optimisation)
Project [pcode#0, lastName#1, firstName#2, age#3, city#10, state#11]
+- Join Inner, (pcode#0 = pcode#9)
  :- Filter isnotnull(pcode#0)
  : +- Relation[pcode#0,lastName#1,firstName#2,age#3] csv
  +- Filter isnotnull(pcode#9)
    +- Relation[pcode#9,city#10,state#11] csv
```

Catalyst Plans

```
== Physical Plan == (actual sequence of operation)
*Project [pcode#0, lastName#1, firstName#2, age#3, city#10, state#11]
+- *BroadcastHashJoin [pcode#0], [pcode#9], Inner, BuildRight
   :- *Project [pcode#0, lastName#1, firstName#2, age#3]
   :  +- *Filter isnotnull(pcode#0)
   :     +- *Scan csv [pcode#0,lastName#1,firstName#2,age#3]
 Format: CSV, InputPaths: file:/data/people.csv,
 PushedFilters: [IsNotNull(pcode)], ReadSchema:
 struct<pcode:string,lastName:string,firstName:string,age:string>
+- BroadcastExchange HashedRelationBroadcastMode(List(input[0,string,true]))
   +- *Project [pcode#9, city#10, state#11]
      +- *Filter isnotnull(pcode#9)
         +- *Scan csv [pcode#9,city#10,state#11]
 Format: CSV, InputPaths: file:/data/pcodes.csv,
 PushedFilters: [IsNotNull(pcode)], ReadSchema:
 struct<pcode:string,city:string,state:string></pcode:string,city:string,state:string>
```

Shuffle

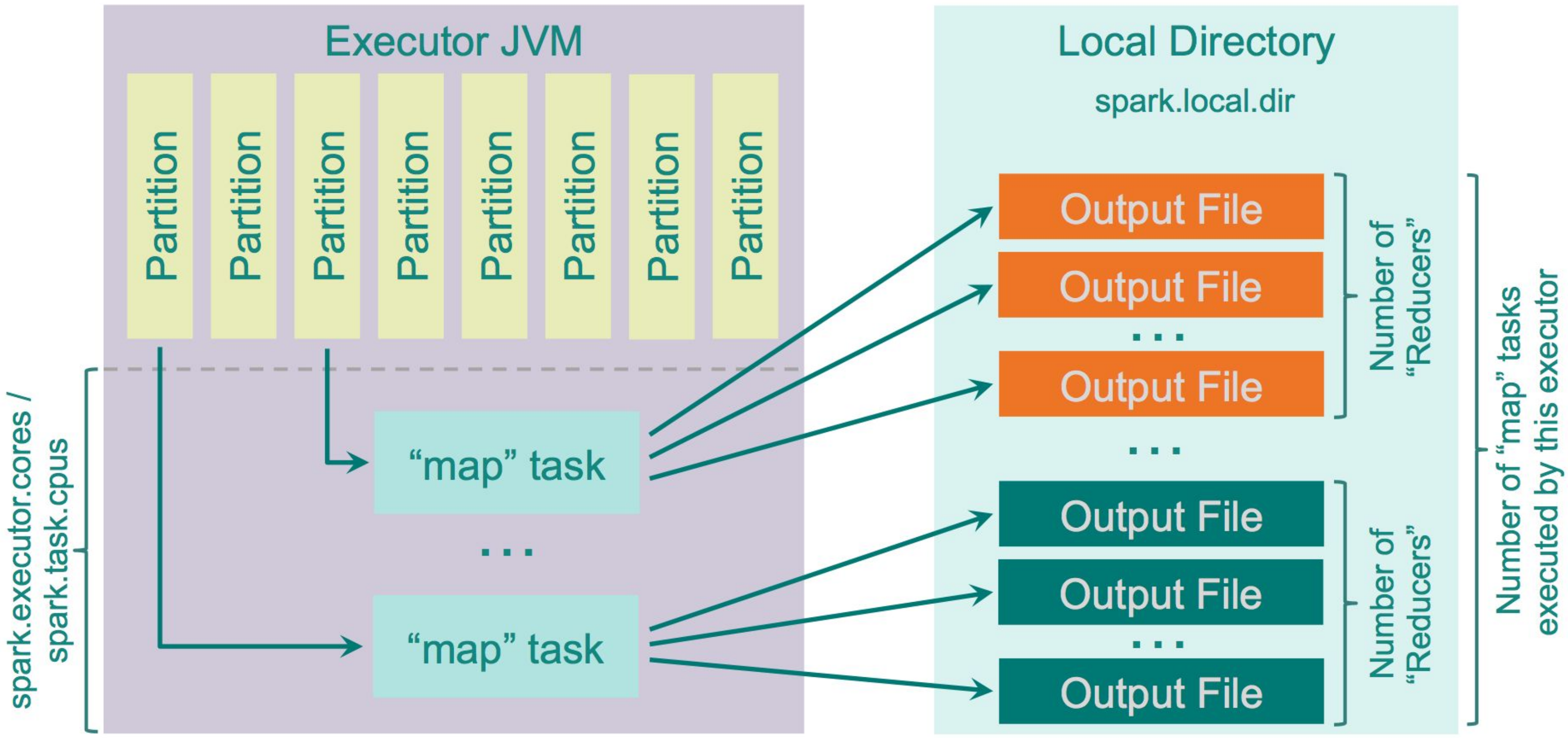
- create File in local FS (spark.shuffle.spill)
- different kind of shuffle
  - hash shuffle
  - consolidate hash shuffle
  - sort shuffle
  - tungsten sort shuffle

Hash Shuffle

- spark.shuffle.manager=hash
- 1file for each reducer for each mapper
- number of file = M*R
- fast
- big amount of files written to FS
- random IO

M: mapper task
R: reducer task

Hash Shuffle

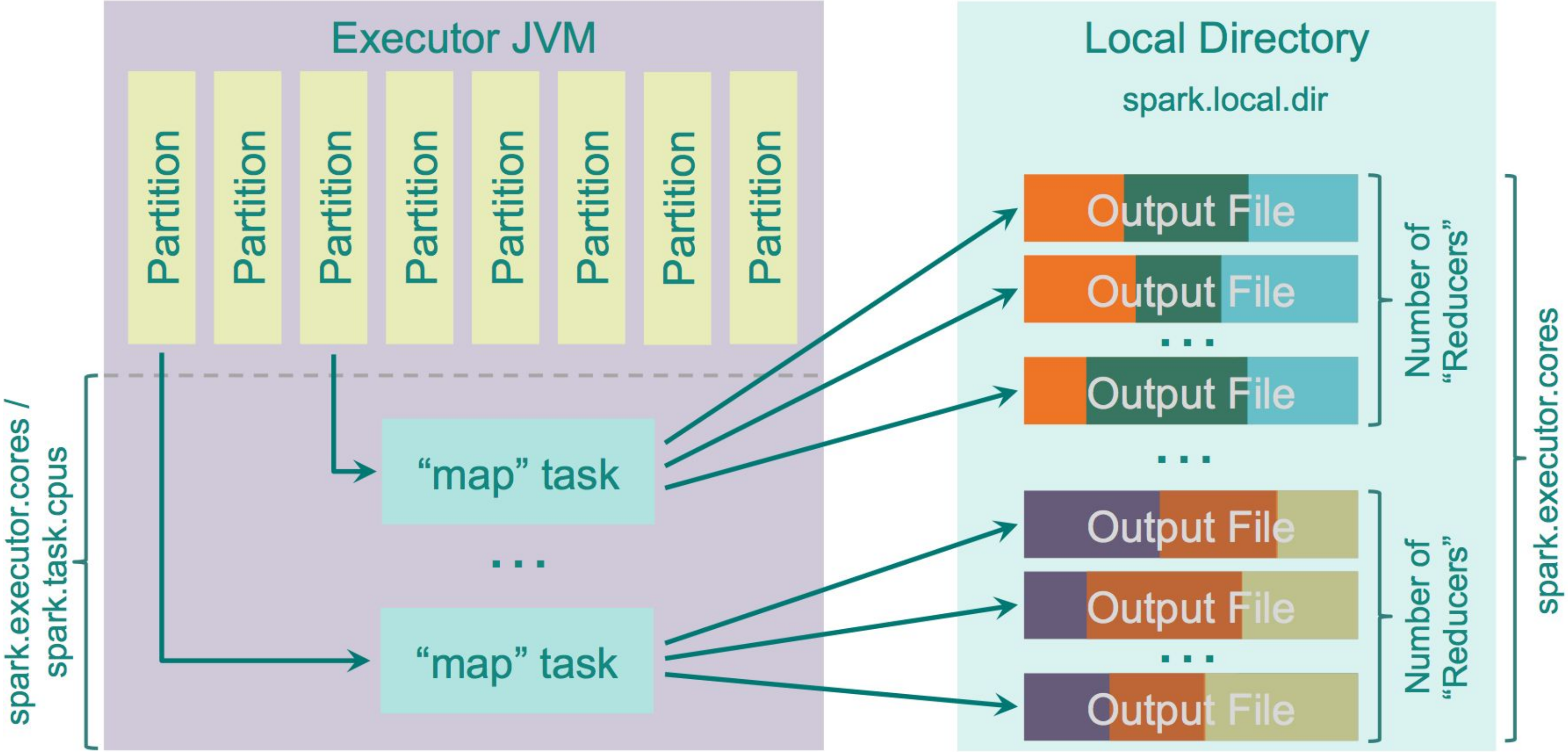


Consolidate Hash Shuffle

- spark.shuffle.manager=hash
- spark.shuffle.consolidateFiles :=true
- 1 file foreach reducer for each task in parallel by executor
- number of files = E * C/T * R
- less file written

M: mapper task
R: reducer task
E: num-executor
C: executor-cores
T: tasks.cpu

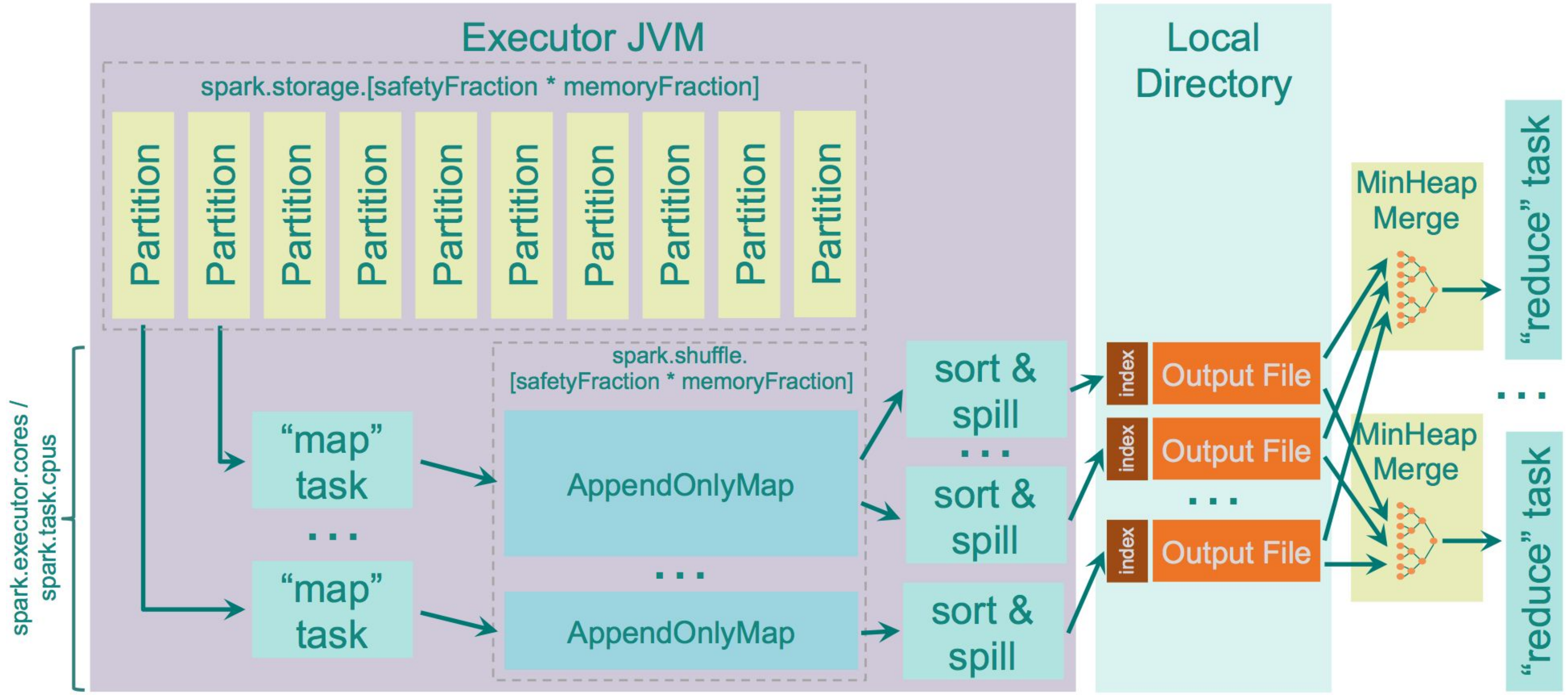
Consolidate Hash Shuffle



Sort Shuffle

- spark.shuffle.manager=sort
- 1 file by mapper ordered by reducer and indexed
- if R<200 then hash (spark.shuffle.sort.bypassMergeThreshold)
- sort data on map side using TimSort
- merge by reducer before sending to reducer
- sort after shuffle is faster

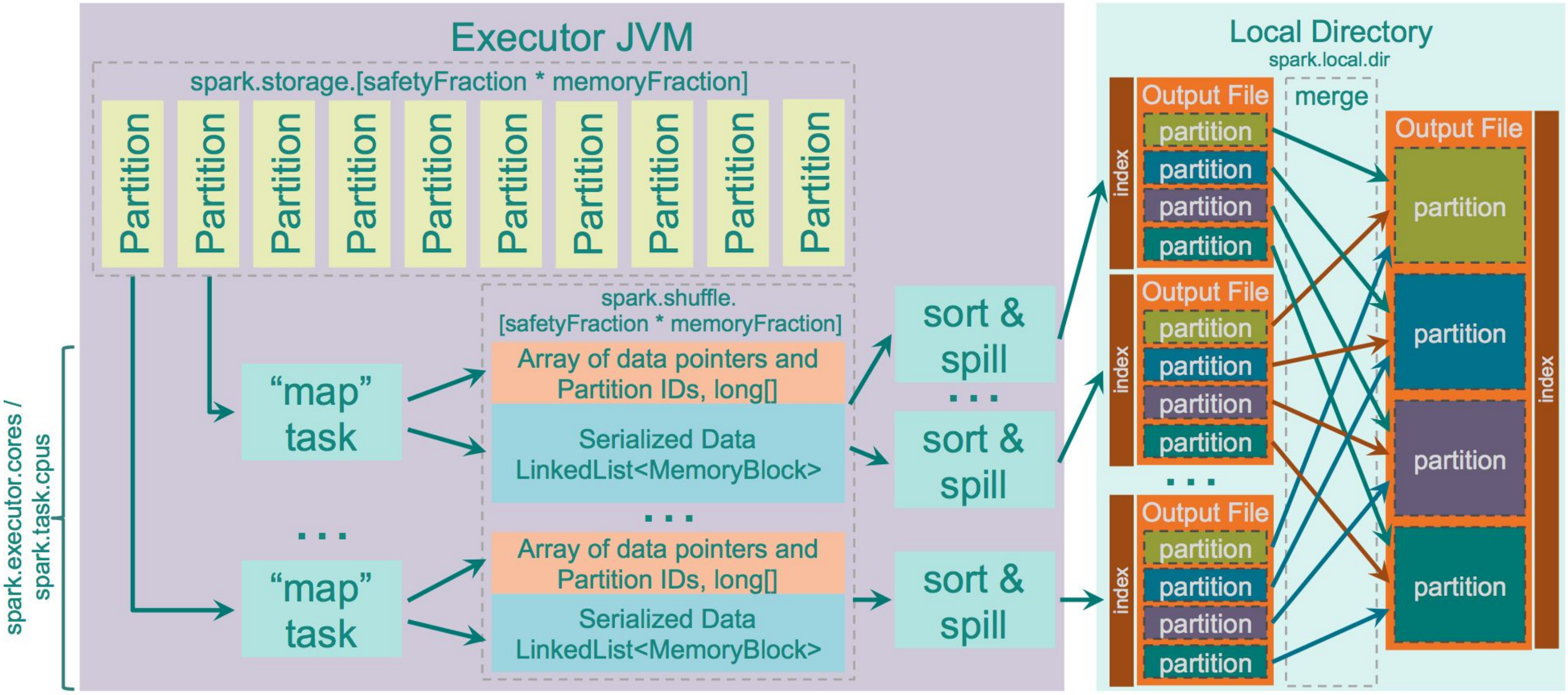
Sort Shuffle



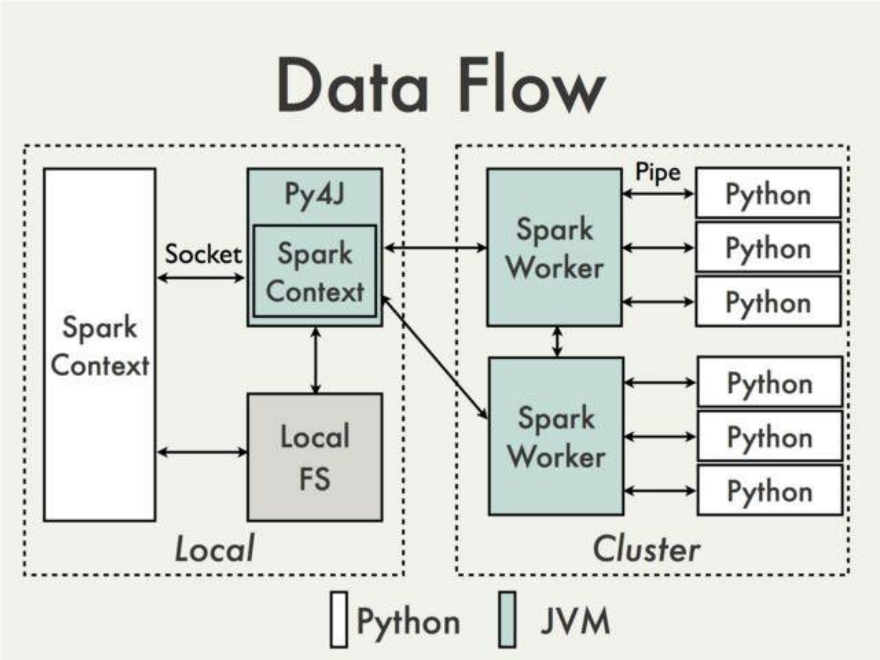
Tungsten Sort Shuffle

- spark.shuffle.manager=tungsten-sort
- operate on serialized data
- cache-efficient sorter
- work only if:
  - no aggregation (deserialisation)
  - less than 16 777 216 output partition
  - row size < 128MB in serialized form
- no more fast sort after shuffle

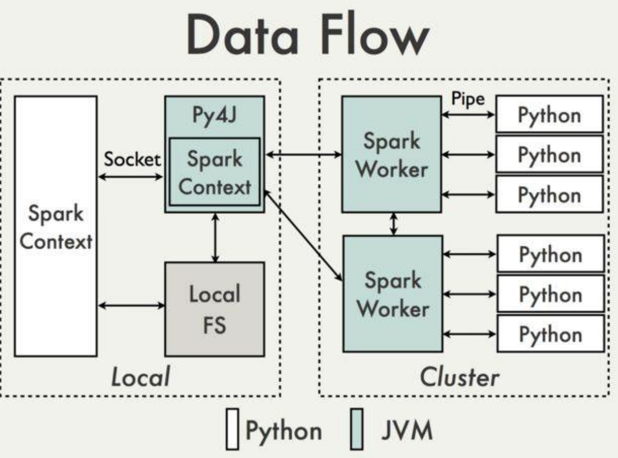
Tungsten Sort Shuffle



Pyspark


Data Flow
Spark Context
Socket
Py4J
Spark Context
Local FS
Local
Pipe
Spark Worker
Spark Worker
Python
Python
Python
Python
Python
Python
Cluster
Python
JVM

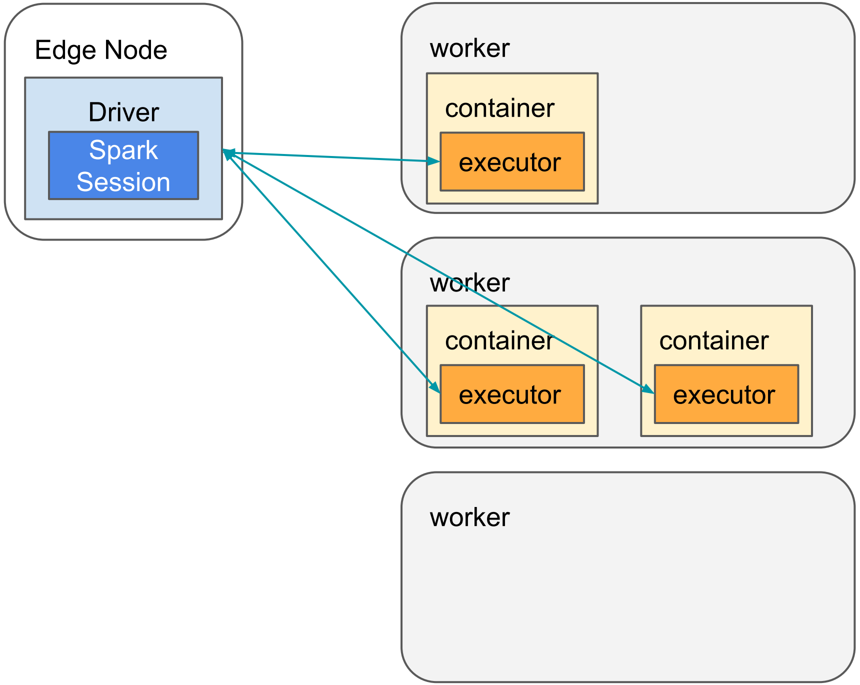
PySpark



- RDD (python) = PythonRDD (Java)
- PythonRDD launch Python subprocesses
- serialization
  - lambda = cloudpickle
  - data = cPickle
  - batch

Thank You

Client mode



Cluster mode