



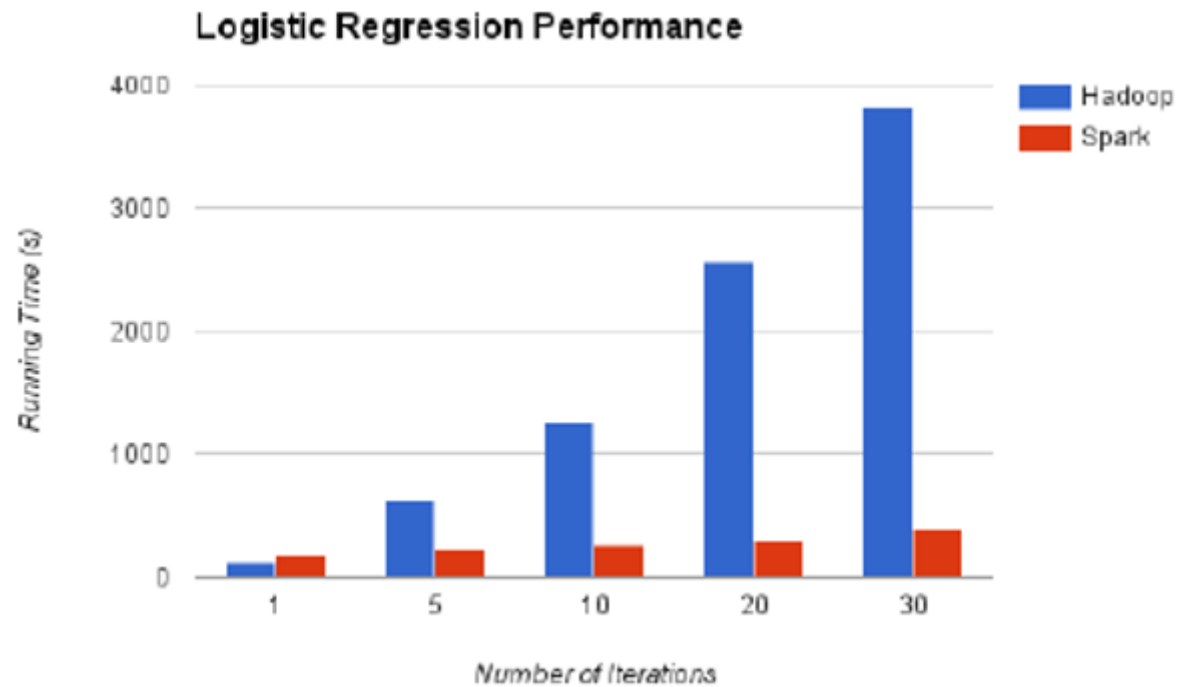
Spark. RDD

SELEZNEV ARTEM
HEAD OF DATA SCIENCE @ SBER

FUNCTIONAL PROGRAMMING

APACHE SPARK



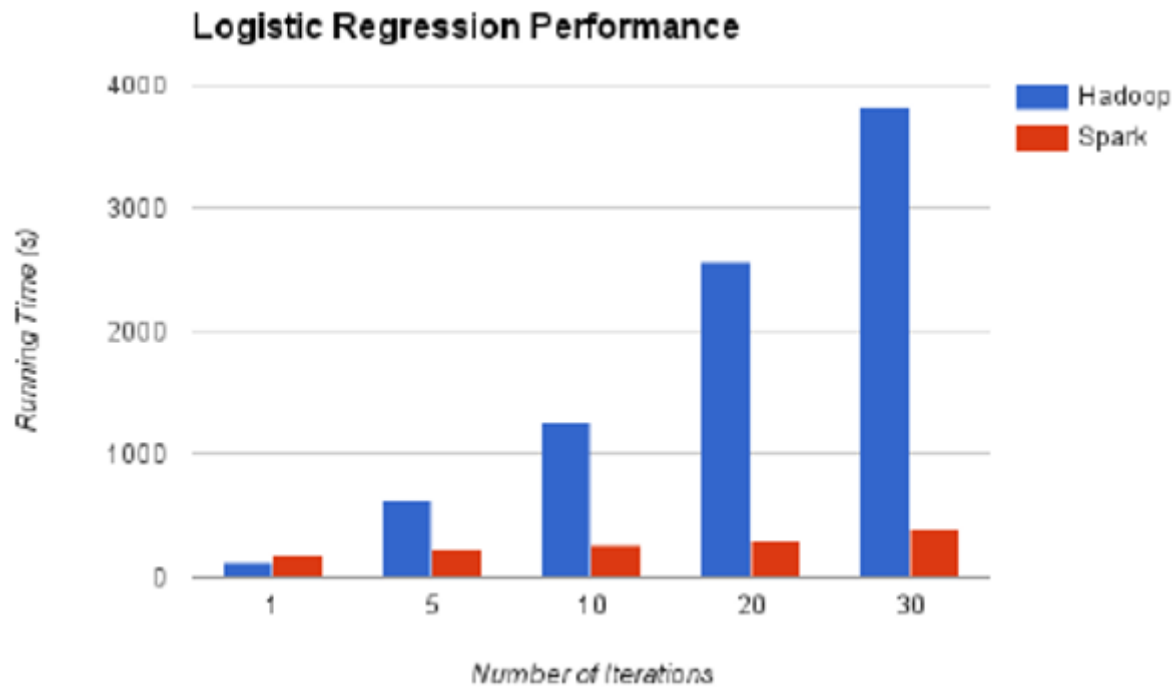


Быстрее Hadoop (?)

DAG

Использование In-Memory (ОЗУ)

Ленивые вычисления

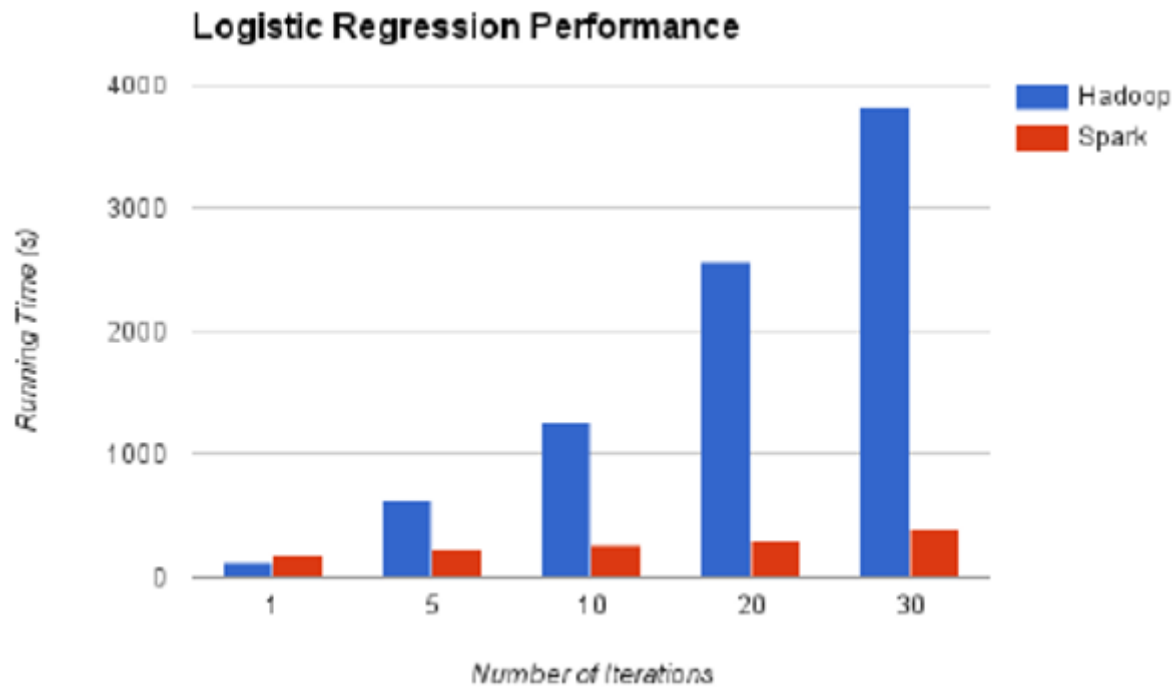


Быстрее Hadoop (?)

DAG

Использование In-Memory (ОЗУ)

Ленивые вычисления



Быстрее Hadoop

DAG

Использование In-Memory (ОЗУ)

Ленивые вычисления

Простое API

Модульность (библиотеки)

ФУНКЦИОНАЛЬНАЯ РАЗНИЦА



`map()`

`reduce()`

ФУНКЦИОНАЛЬНАЯ РАЗНИЦА



`map()`

`filter()`

`join()`

`first()`

`reduce()`

`sortBy()`

`groupByKey()`

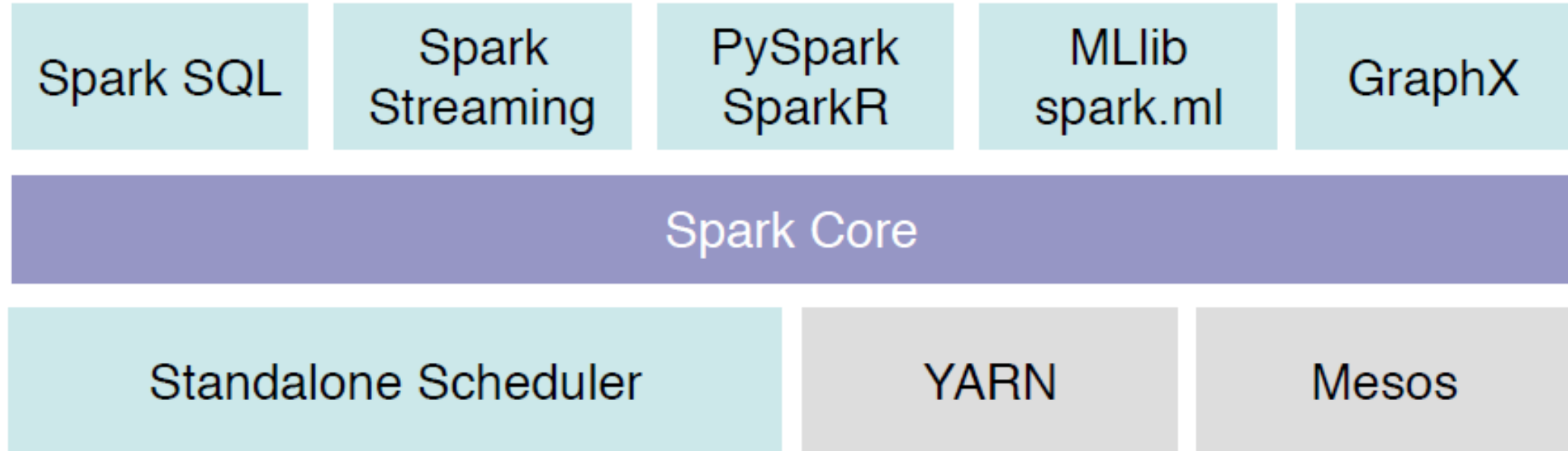
`count()`



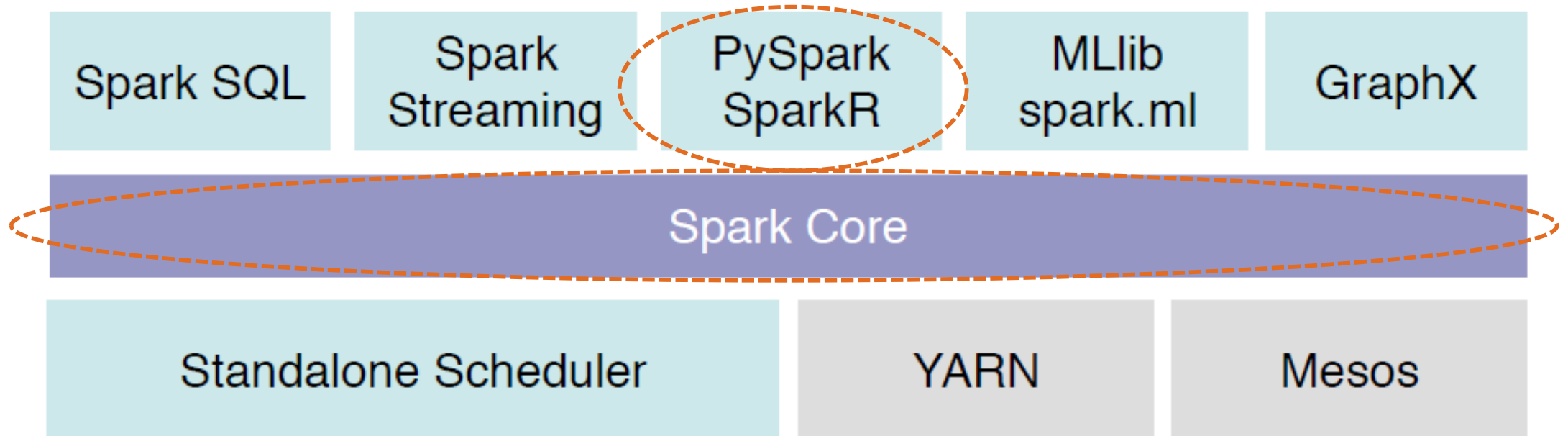
`map()`

`reduce()`

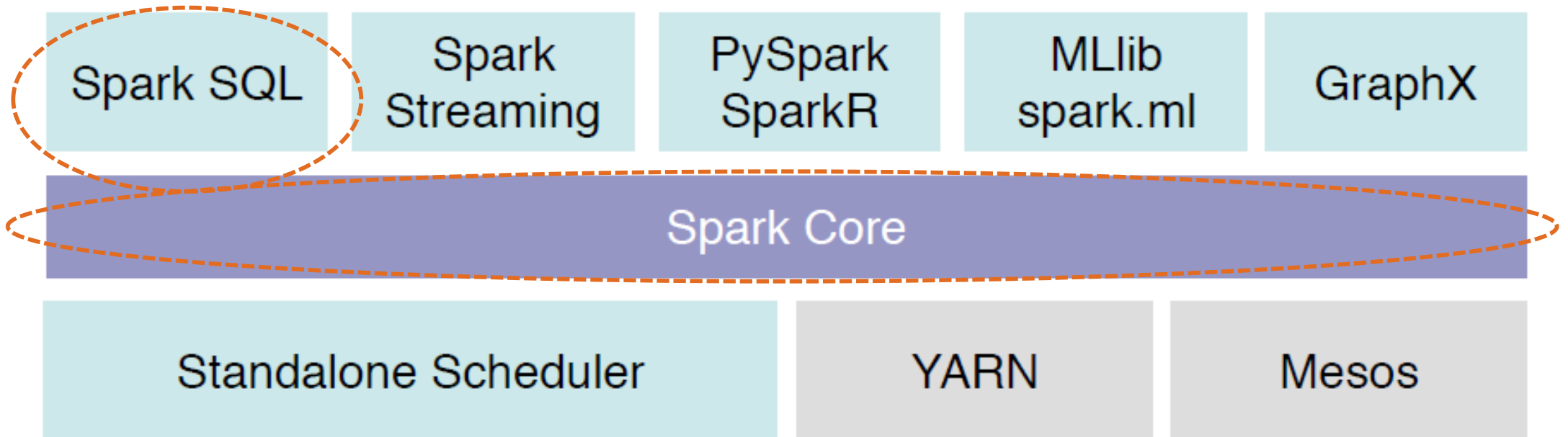
APACHE SPARK



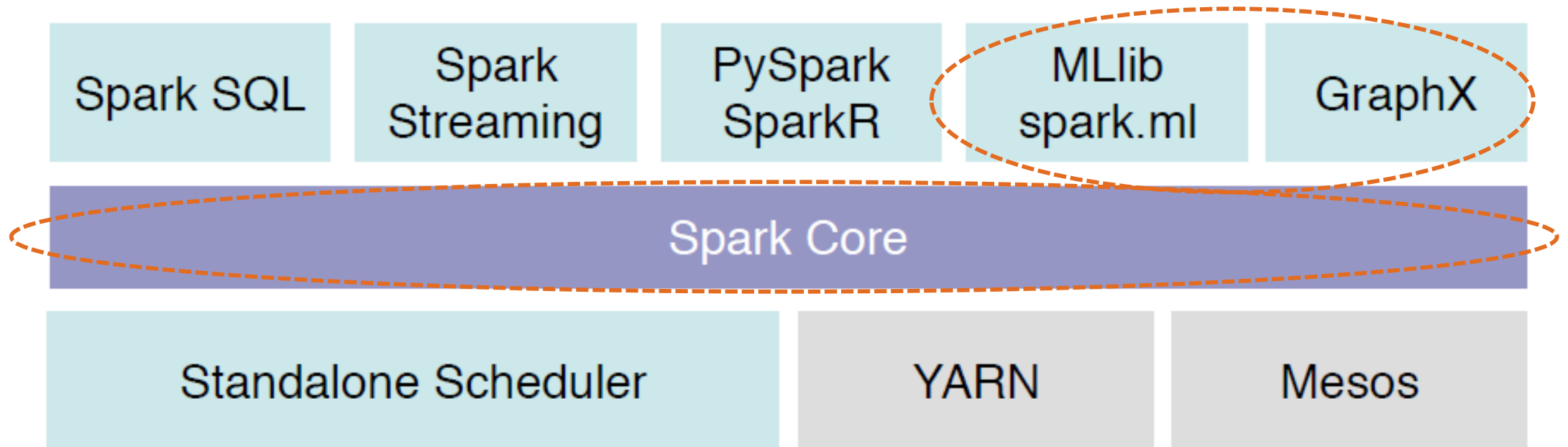
APACHE SPARK - RDD



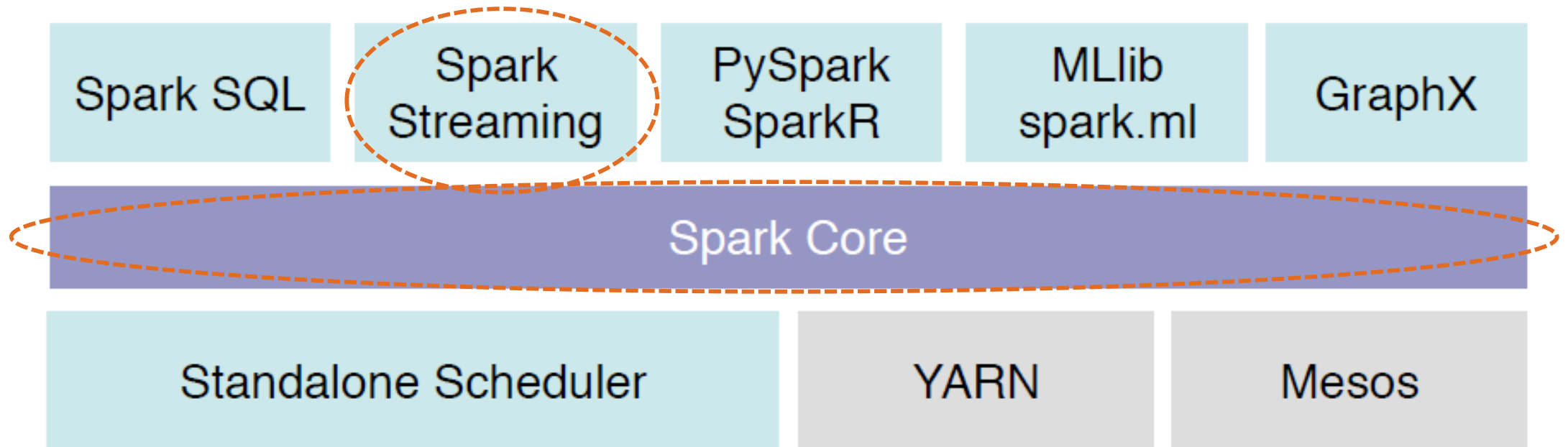
APACHE SPARK - SQL



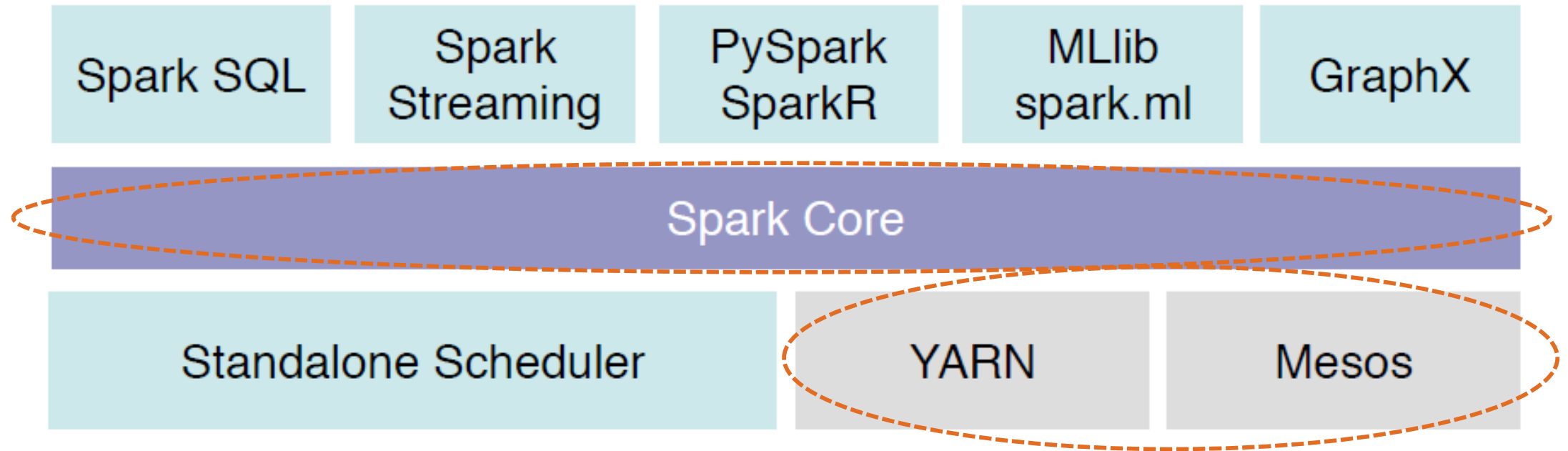
APACHE SPARK - ML



APACHE SPARK - STREAMING



APACHE SPARK – DEPLOY/CLUSTER



APACHE SPARK – ТЕРМИНЫ

DRIVER

WORKER

APACHE SPARK – ТЕРМИНЫ

DRIVER

Процесс содержащий Spark Context

WORKER

Приложение (node) выполняющее код/команды

APACHE SPARK – ТЕРМИНЫ

DRIVER

Процесс содержащий Spark Context

WORKER

Приложение (node) выполняющее код/команды

APACHE SPARK – ТЕРМИНЫ (КЛАСТЕР)

DRIVER

Процесс содержащий Spark Context

MASTER

Процесс управляющий приложением
на всем кластере

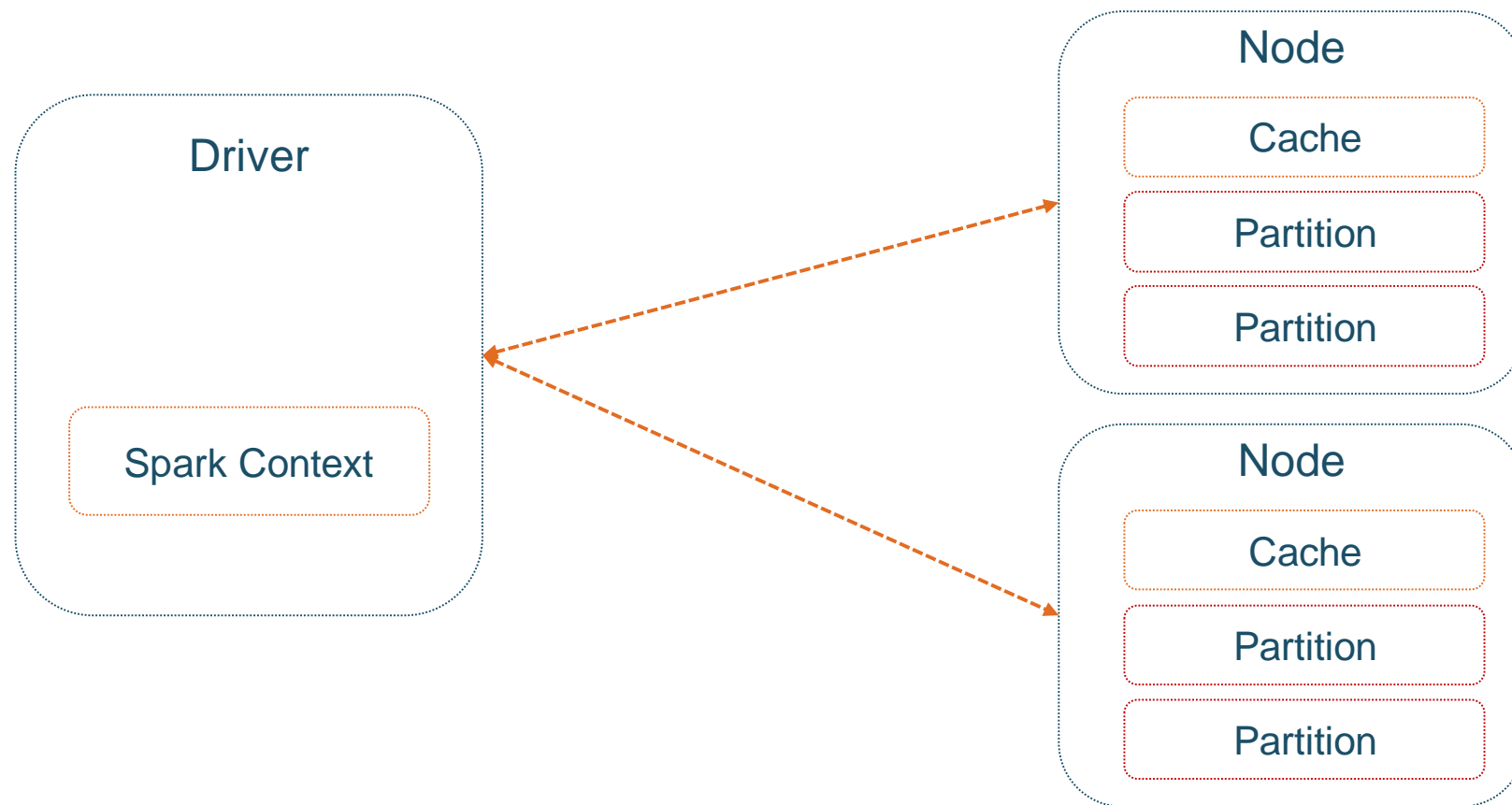
EXECUTOR

Процесс который выполняет Spark задачи

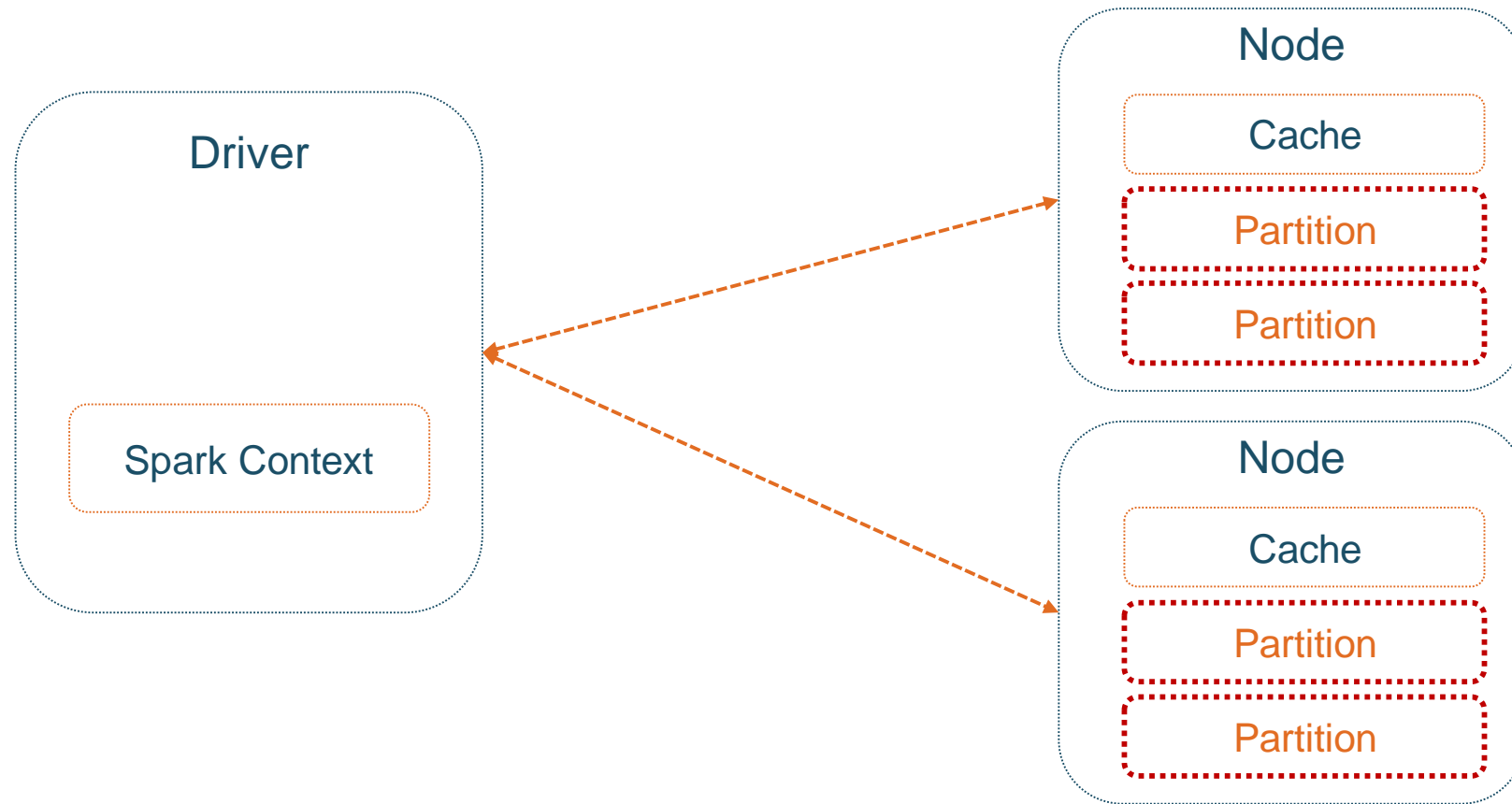
WORKER

Процесс управляющий Executor'ами
на конкретной ноде кластера

РАБОТА КОНТЕКСТА



РАБОТА КОНТЕКСТА (?)



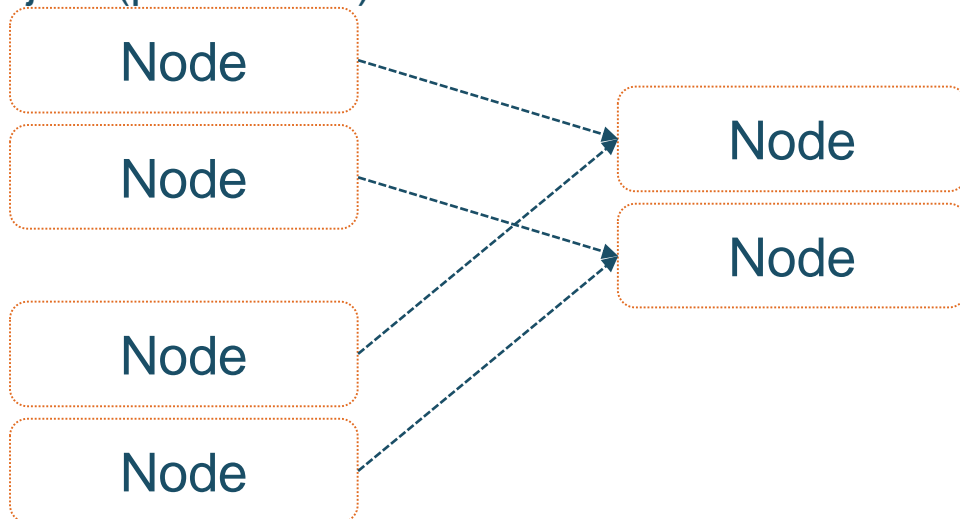
ЗАВИСИМОСТЬ ОТ ПАРТИЦИЙ

(НЕ БОЛЬШАЯ)

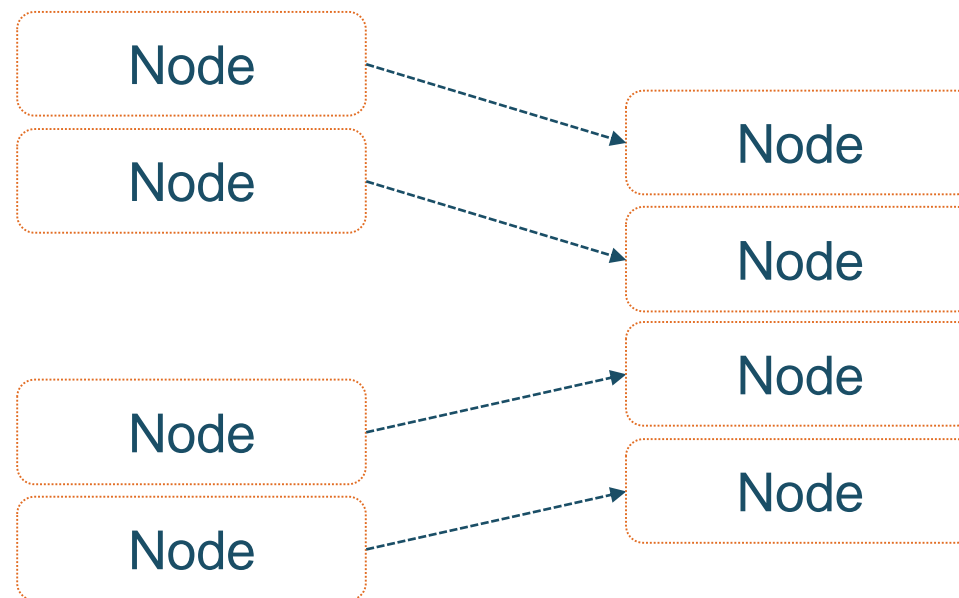
map, filter



join (partitioned)



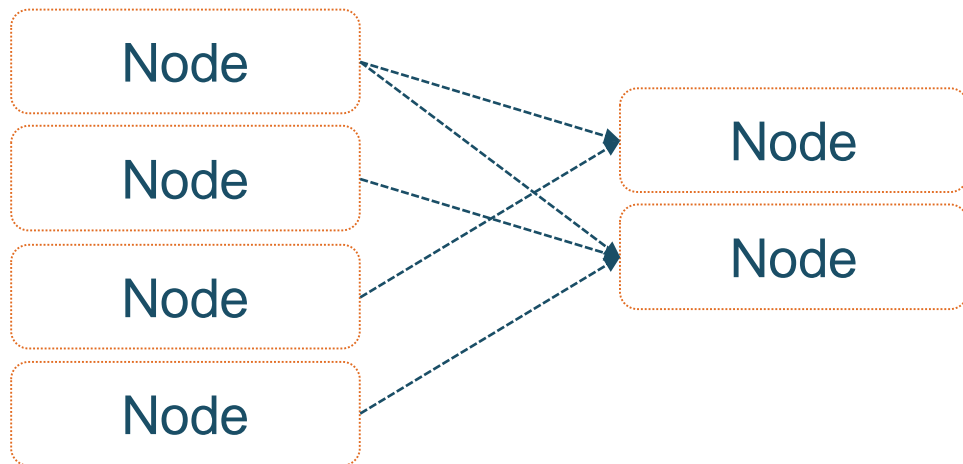
union



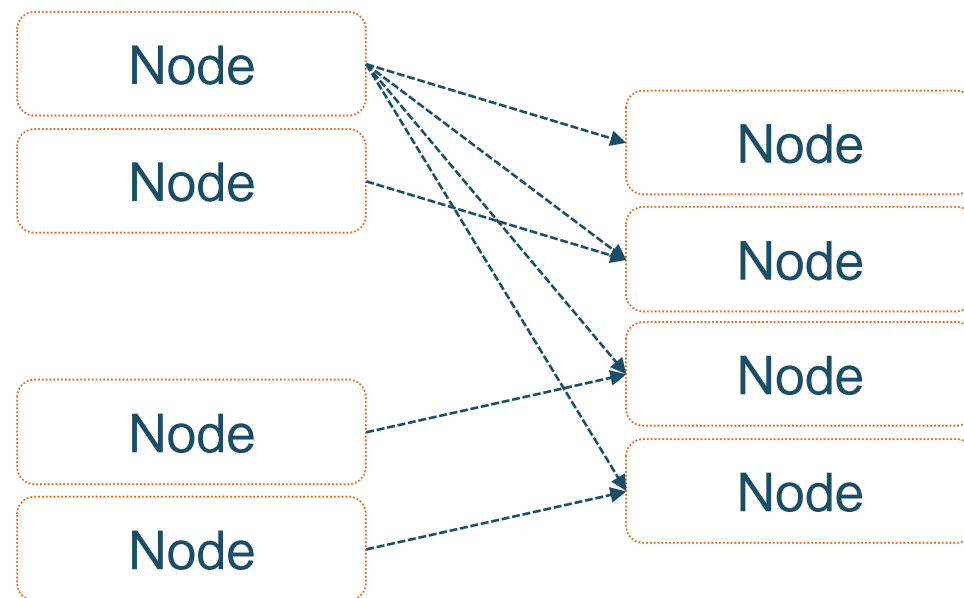
ЗАВИСИМОСТЬ ОТ ПАРТИЦИЙ

(БОЛЬШАЯ)

groupByKey



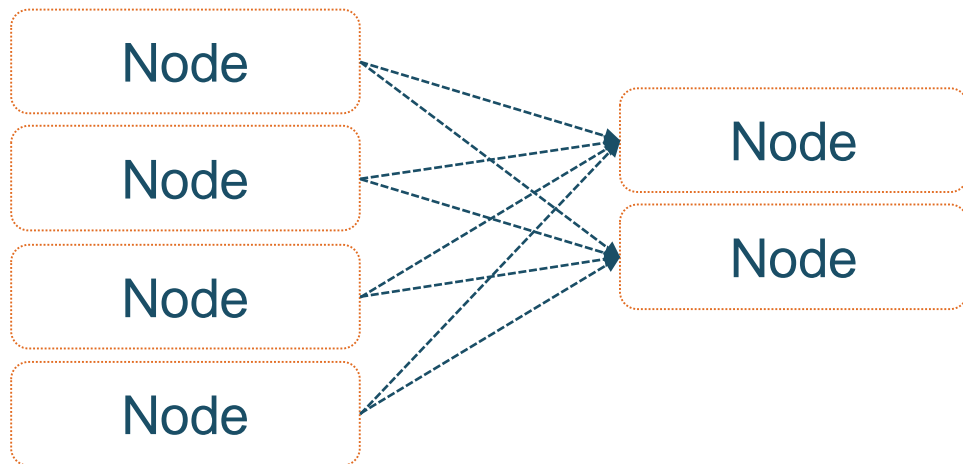
join (not partitioned)



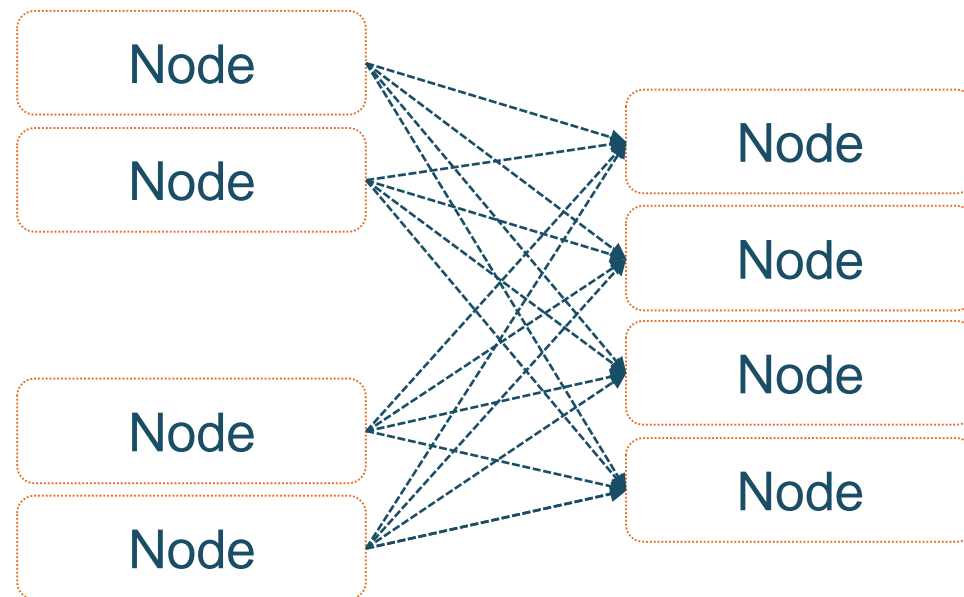
ЗАВИСИМОСТЬ ОТ ПАРТИЦИЙ

(БОЛЬШАЯ)

groupByKey

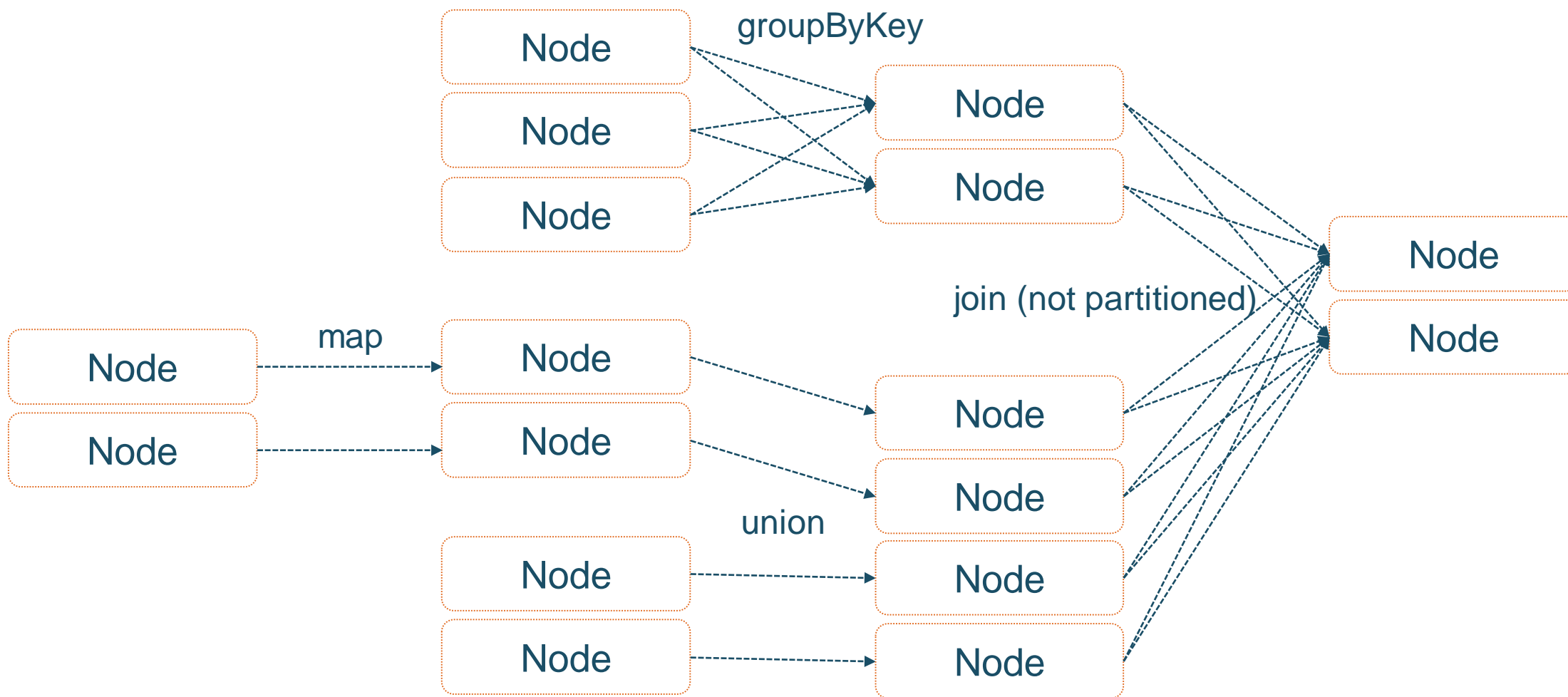


join (not partitioned)



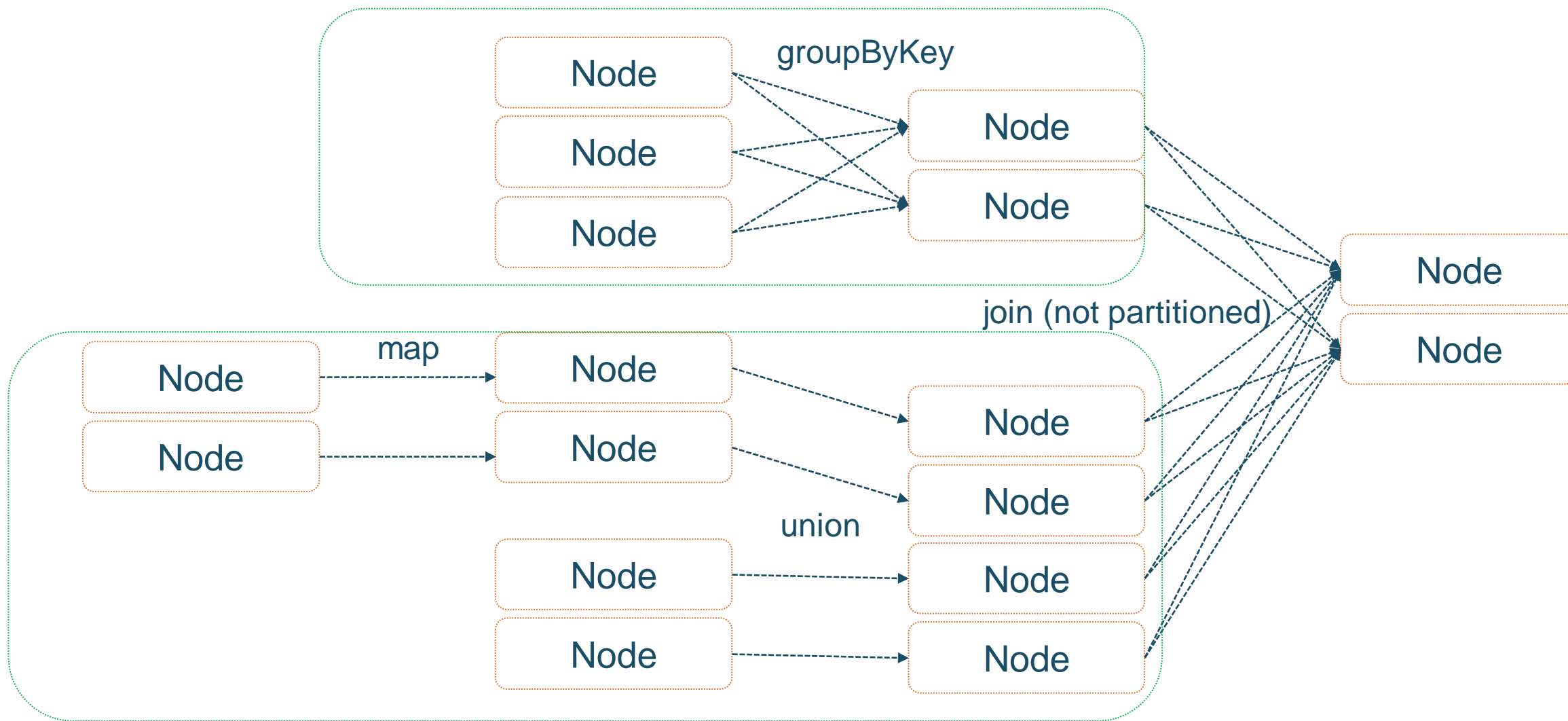
ЗАВИСИМОСТЬ ОТ ПАРТИЦИЙ

(В ПРОЦЕССЕ ЗАДАЧИ)



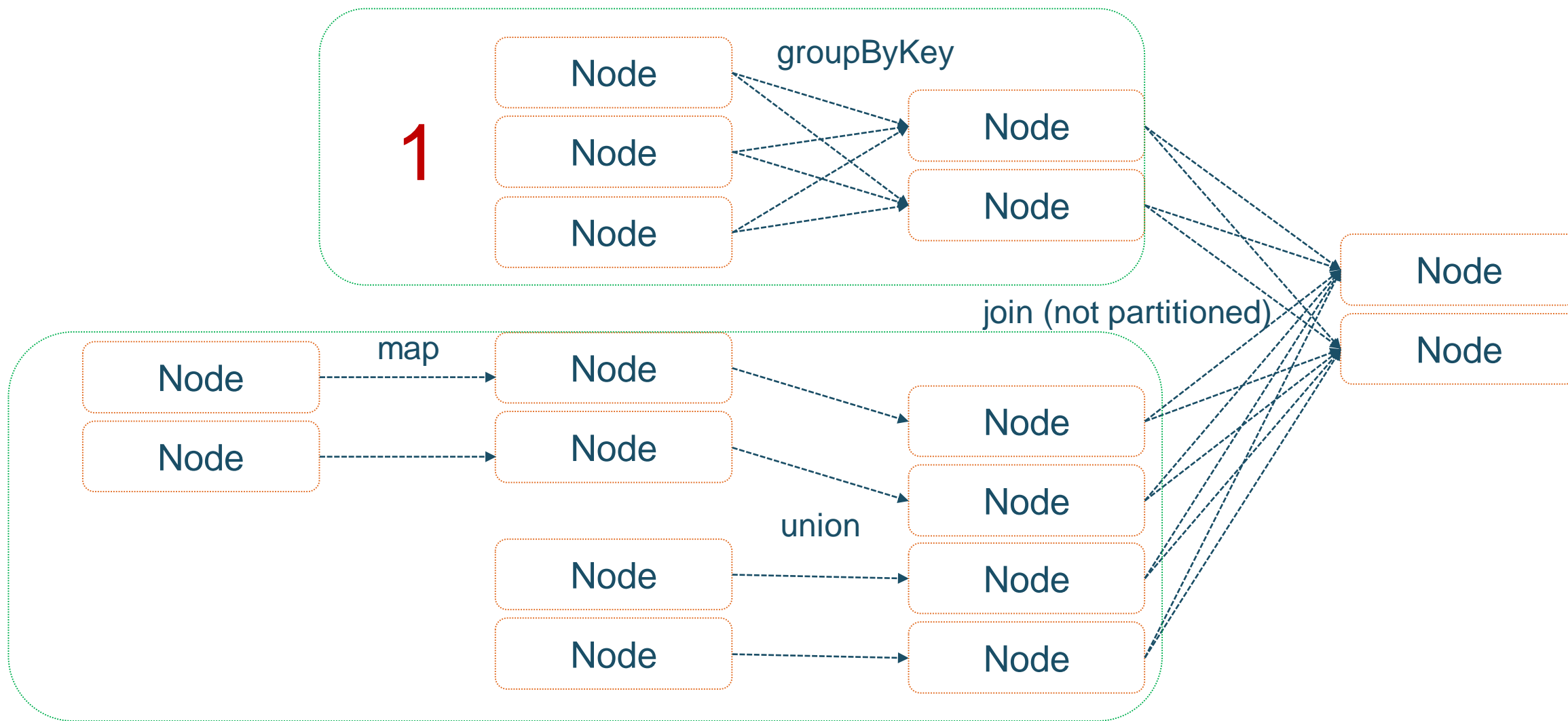
ЗАВИСИМОСТЬ ОТ ПАРТИЦИЙ

(В ПРОЦЕССЕ ЗАДАЧИ)



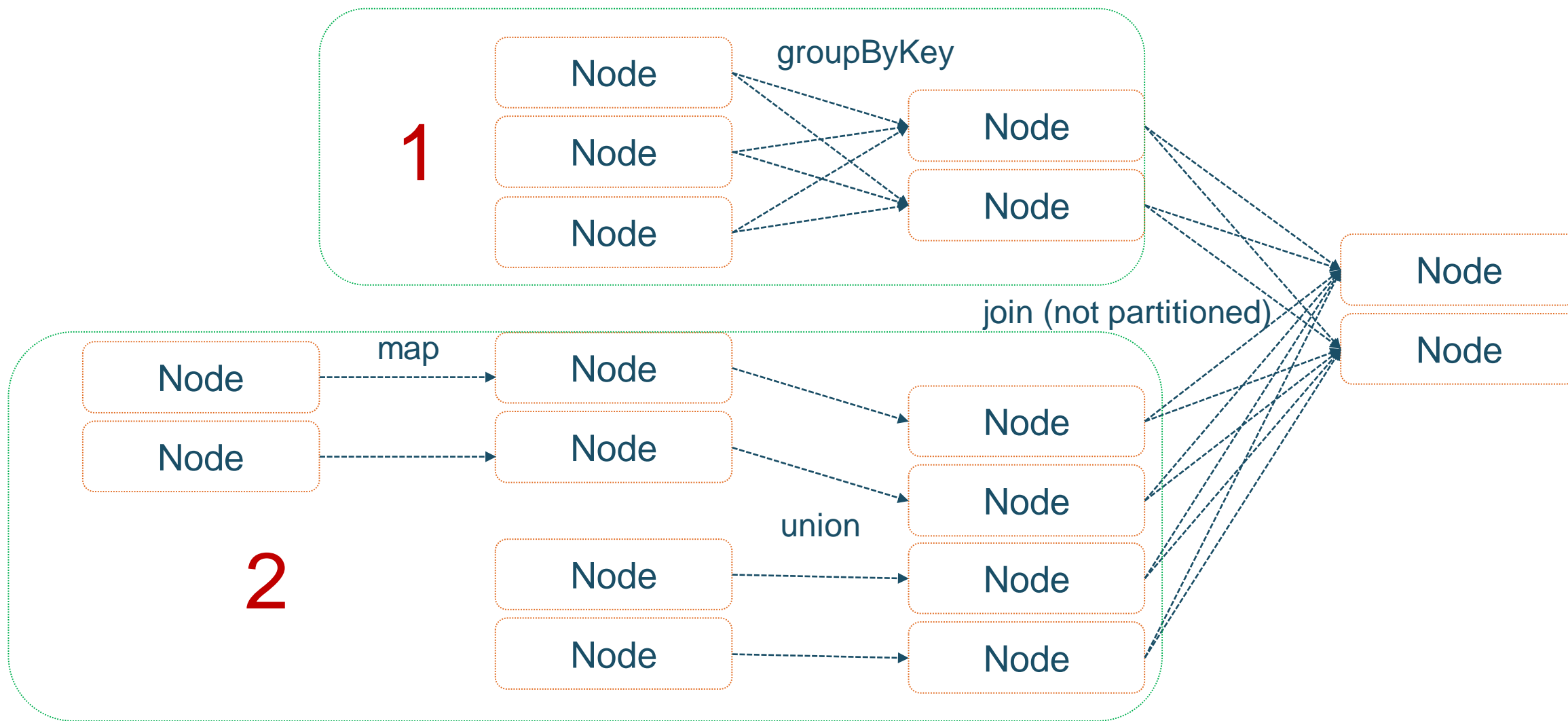
ЗАВИСИМОСТЬ ОТ ПАРТИЦИЙ

(В ПРОЦЕССЕ ЗАДАЧИ)



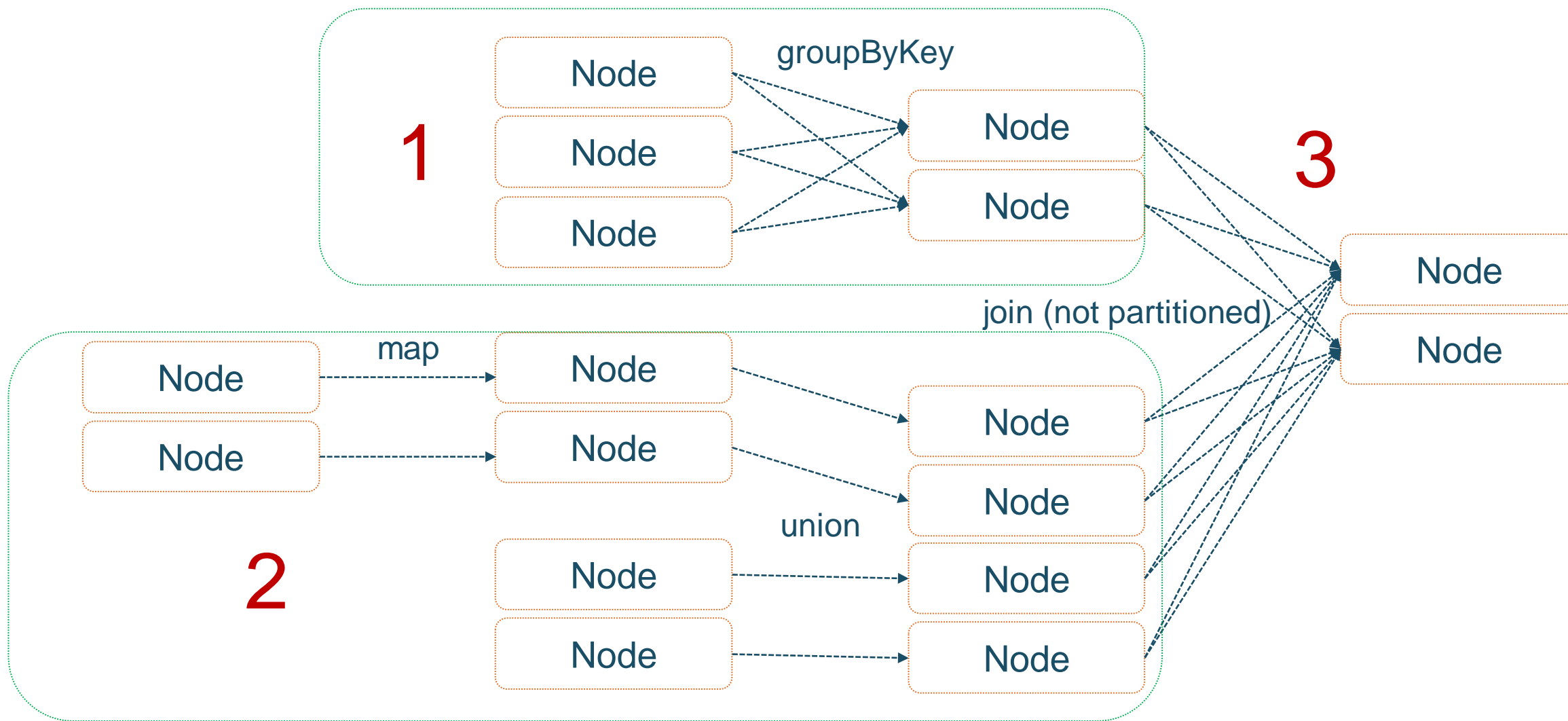
ЗАВИСИМОСТЬ ОТ ПАРТИЦИЙ

(В ПРОЦЕССЕ ЗАДАЧИ)



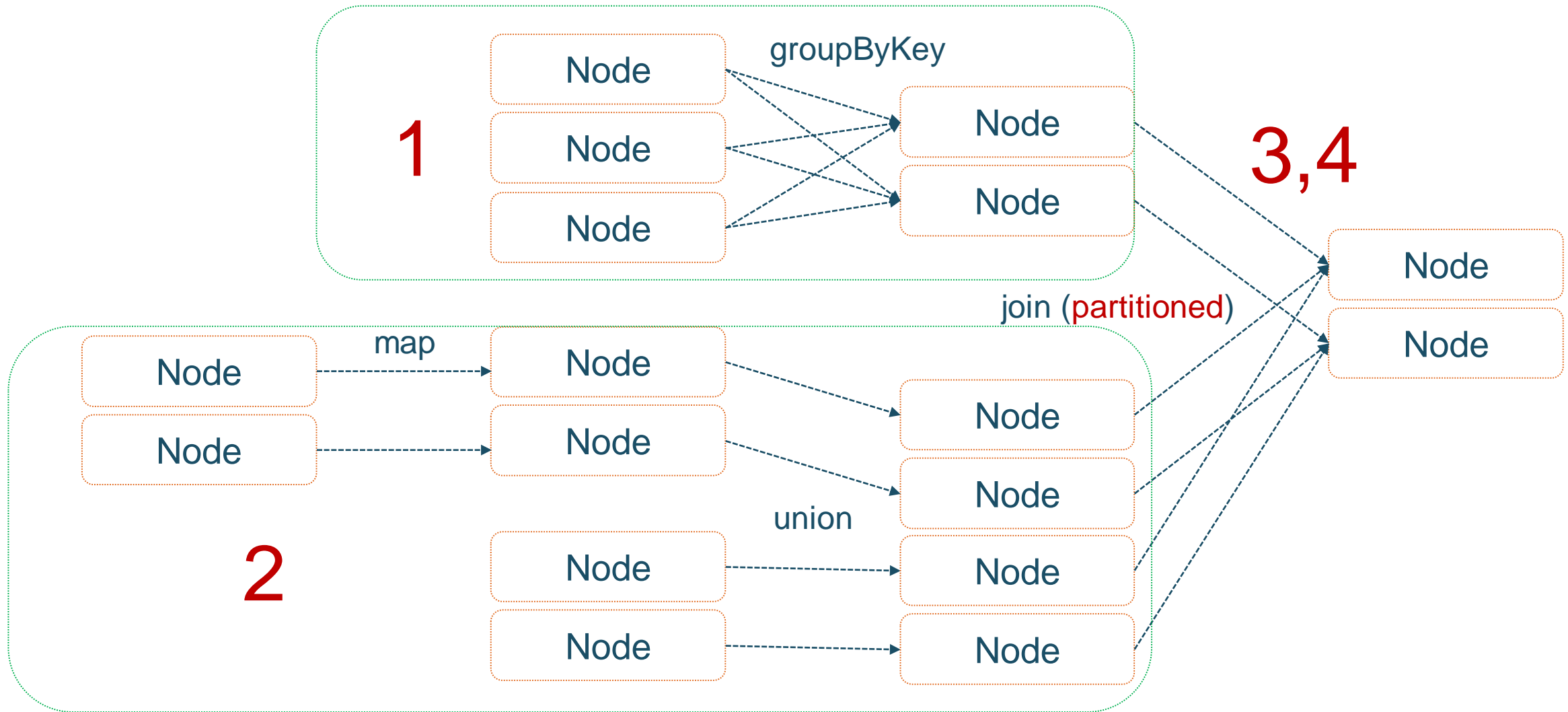
ЗАВИСИМОСТЬ ОТ ПАРТИЦИЙ

(В ПРОЦЕССЕ ЗАДАЧИ)



ЗАВИСИМОСТЬ ОТ ПАРТИЦИЙ

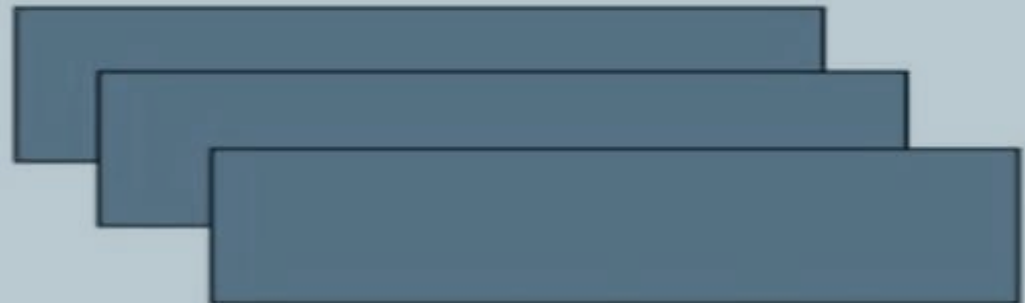
(В ПРОЦЕССЕ ЗАДАЧИ)



ФУНКЦИИ УПРАВЛЕНИЯ ПАРТИЦИЯМИ

<code>.partitions()</code>	Список объектов партиций
<code>.dependencies()</code>	Список объектов зависимостей
<code>.compute(p, parent)</code>	Кол-во элементов в партиции Р в родительском объекте
<code>.practitioner()</code>	Метадата по партиции
<code>preferredLocations(p)</code>	Список нод, где партиция Р расположена

SPARK RDD



RDD– ТЕРМИНЫ

RESILIENT

DISTRIBUTED

DATASET

RDD– ТЕРМИНЫ

RESILIENT

Если данные будут потеряны из процесса, они будут (могут быть) восстановлены из памяти

DISTRIBUTED

Данные разделены на партиции по нодам кластера

DATASET

Это данные – они изначально должны быть в файле или созданы

RDD– ТЕРМИНЫ

IMMUTABLE, READ-ONLY

RESILIENT

Если данные будут потеряны из процесса, они будут (могут быть) восстановлены из памяти

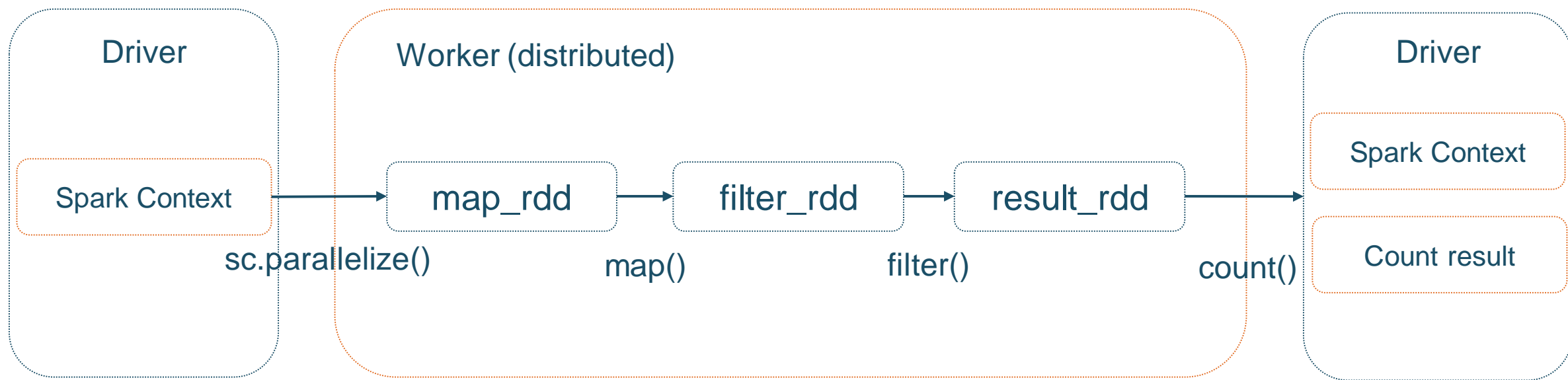
DISTRIBUTED

Данные разделены на партиции по нодам кластера

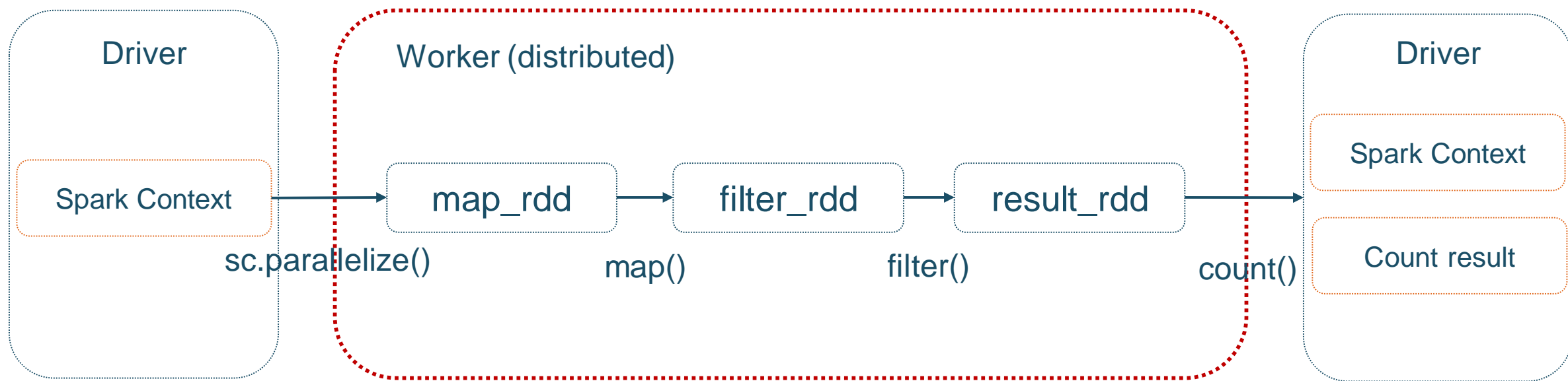
DATASET

Это данные – они изначально должны быть в файле или созданы

ОБЛАСТЬ СУЩЕСТВОВАНИЯ RDD



ОБЛАСТЬ СУЩЕСТВОВАНИЯ RDD



RDD (KEY – VALUE)

```
pets = sc.parallelize([("cat", 1), ("dog", 1), ("cat", 2)])
```

```
pets.reduceByKey(lambda x, y: x + y) # => {(cat, 3), (dog, 1)}
```

```
pets.groupByKey() # => {(cat, [1, 2]), (dog, [1])}
```

```
pets.sortByKey() # => {(cat, 1), (cat, 2), (dog, 1)}
```

RDD (KEY – VALUE) | MAP – REDUCE

```
pets = sc.parallelize([("cat", 1), ("dog", 1), ("cat", 2)])
```

```
pets.reduceByKey(lambda x, y: x + y) # => {(cat, 3), (dog, 1)}
```

```
pets.groupByKey() # => {(cat, [1, 2]), (dog, [1])}
```

```
pets.sortByKey() # => {(cat, 1), (cat, 2), (dog, 1)}
```

MAPPER

- `str`(обычный файл\тс вашими\тданными)
- `list(list(str(обычный файл),
 str(с вашими),
 str(данными)
))`
- `function(object) <- list(str)`
- `return: key – value`

RDD (KEY – VALUE) | MAP – REDUCE

```
pets = sc.parallelize([("cat", 1), ("dog", 1), ("cat", 2)])
```

```
pets.reduceByKey(lambda x, y: x + y) # => {(cat, 3), (dog, 1)}
```

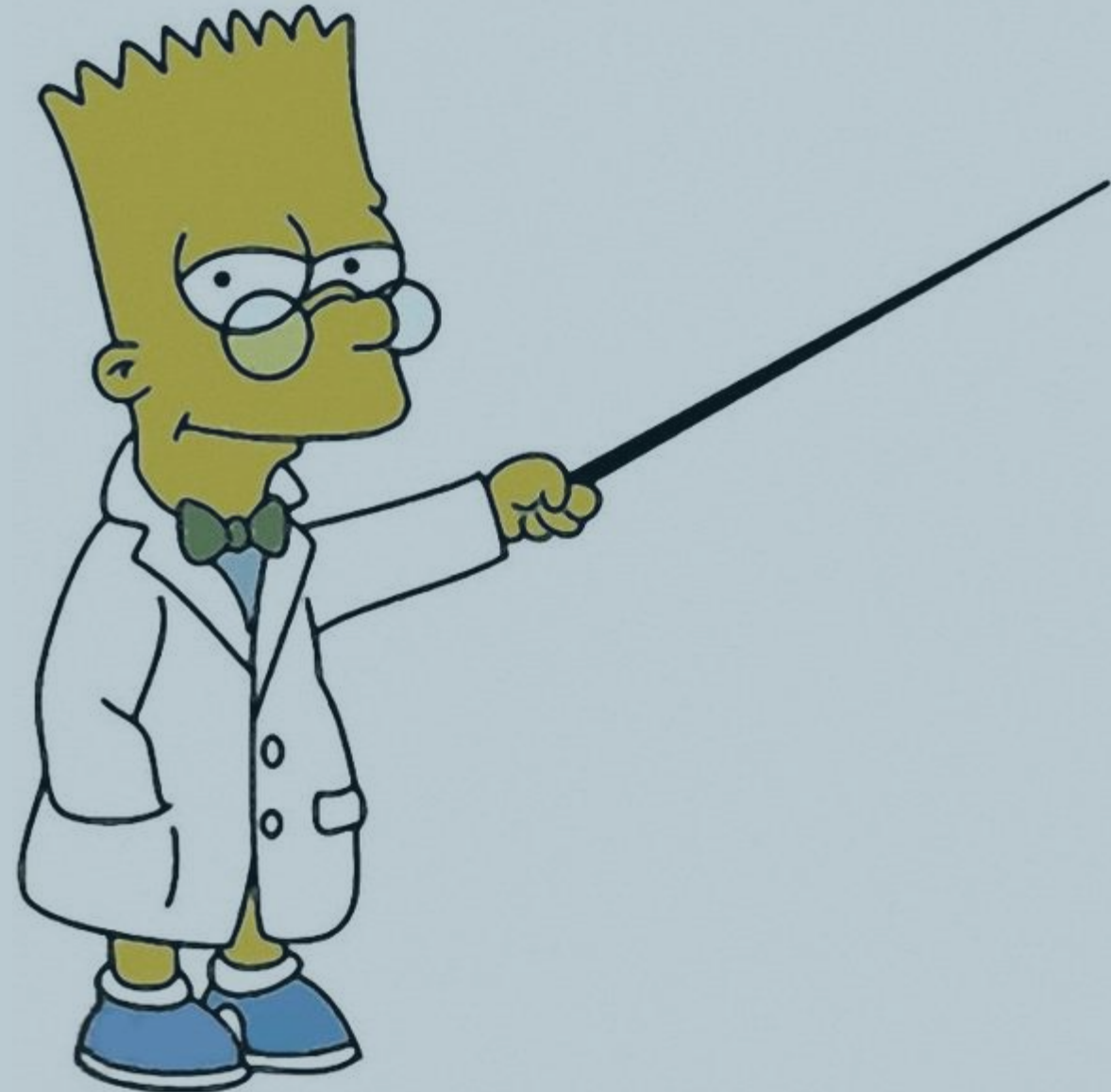
```
pets.groupByKey() # => {(cat, [1, 2]), (dog, [1])}
```

```
pets.sortByKey() # => {(cat, 1), (cat, 2), (dog, 1)}
```

MAPPER

- `str`(обычный файл\тс вашими\тданными)
- `list(list(str(обычный файл),
 str(с вашими),
 str(данными)
))`
- `function(object) <- list(str)`
- `return: key – value`

RDD НА ПРИМЕРЕ



RDD | WORDCOUNT

```
file_rdd.map(line => line.split(" "))  
  .map(split => (split(0), split(1).toInt))  
  .groupByKey()  
  .mapValues(iter => iter.reduce(_ + _)).collect()
```

dummy.txt

```
jon 2  
mary 3  
anna 1  
jon 1  
jesse 3  
mary 5
```

RDD | PLAN

```
sc.textFile('file:///dummy.txt')
```



```
map(line => line.split(" "))
```



```
map(split => (split(0), split(1).toInt))
```



```
groupByKey()
```

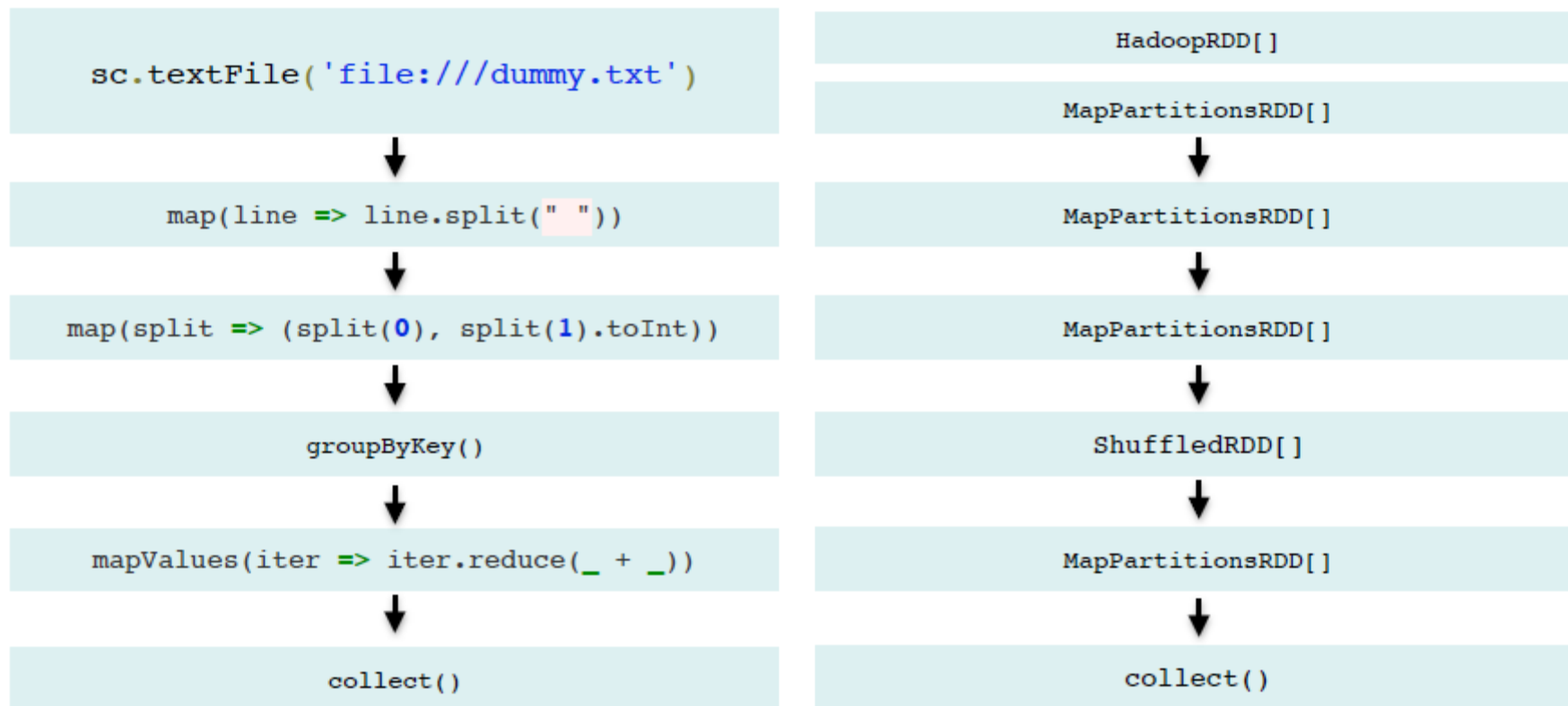


```
mapValues(iter => iter.reduce(_ + _))
```



```
collect()
```

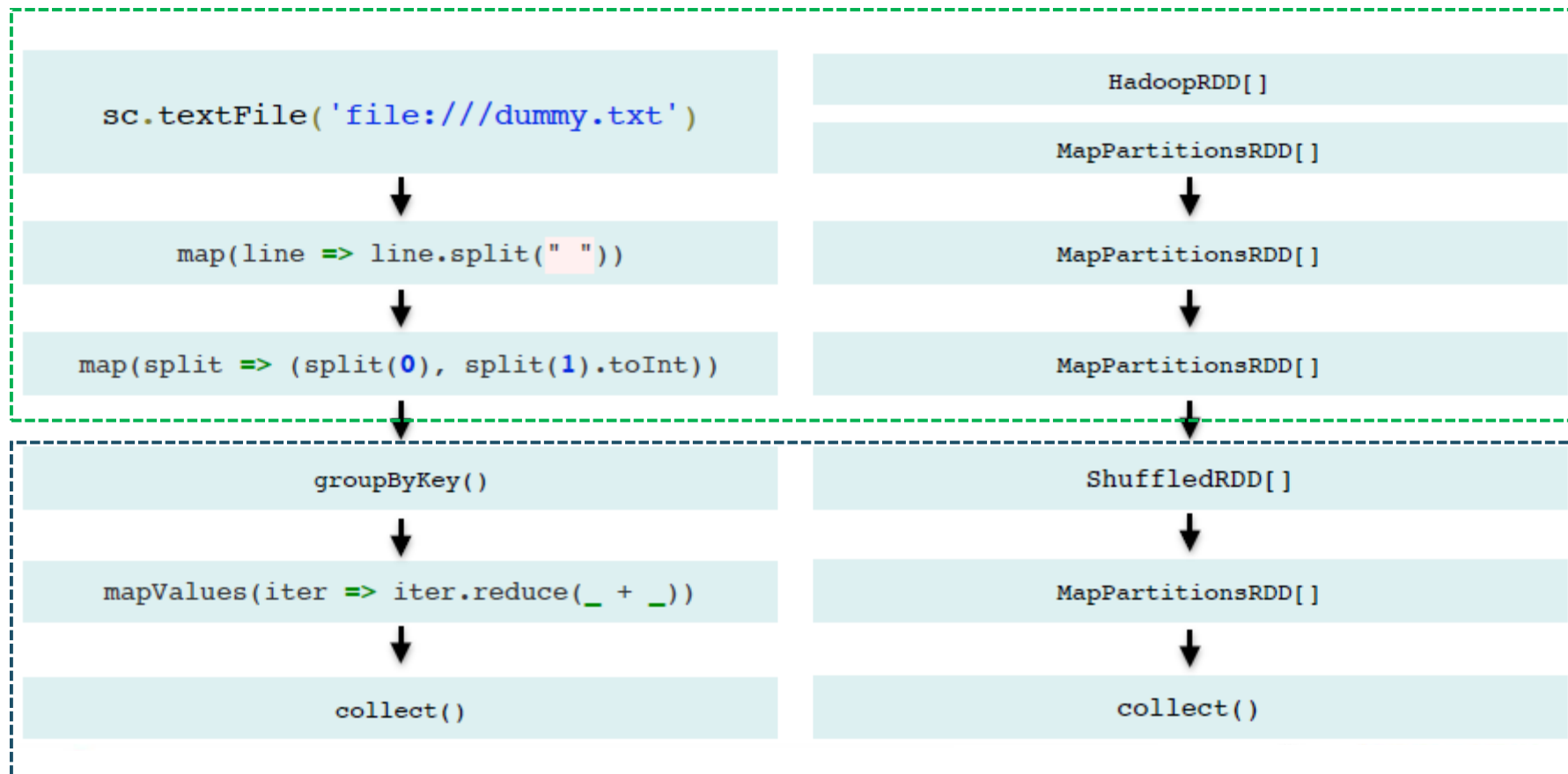
RDD | PLAN RDD



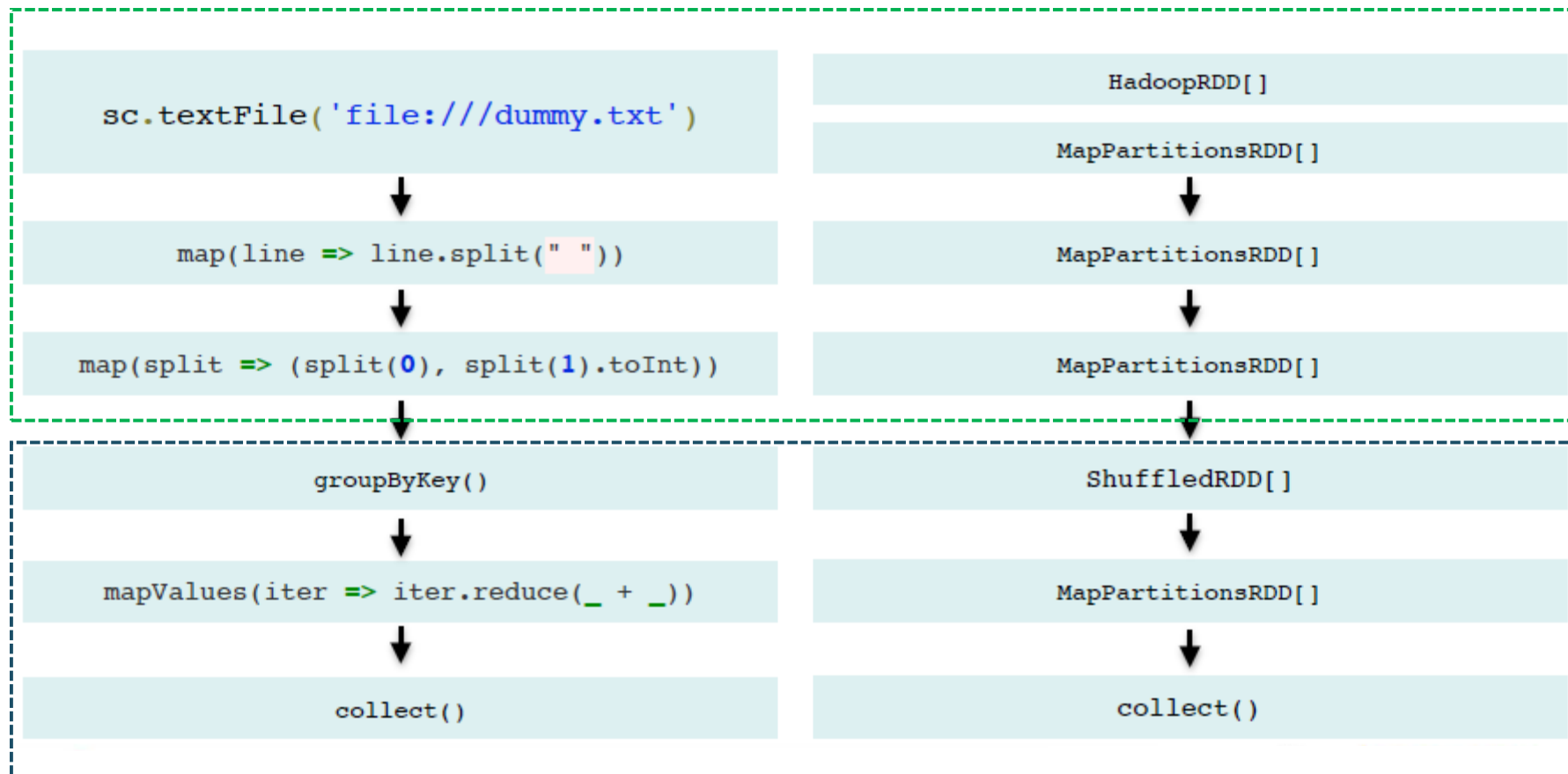
RDD | БАРЬЕР



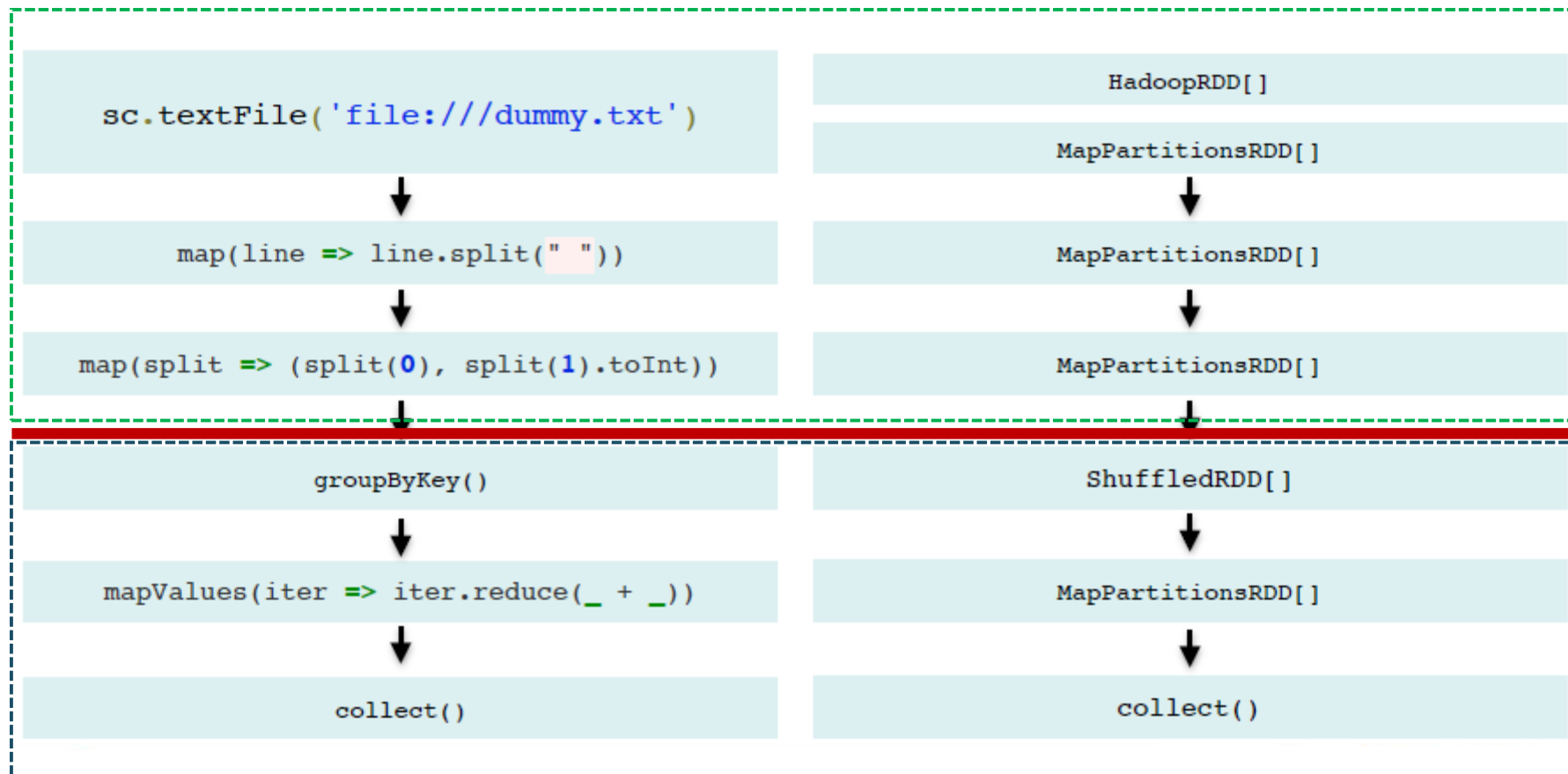
RDD | БАРЬЕР



RDD | БАРЬЕР

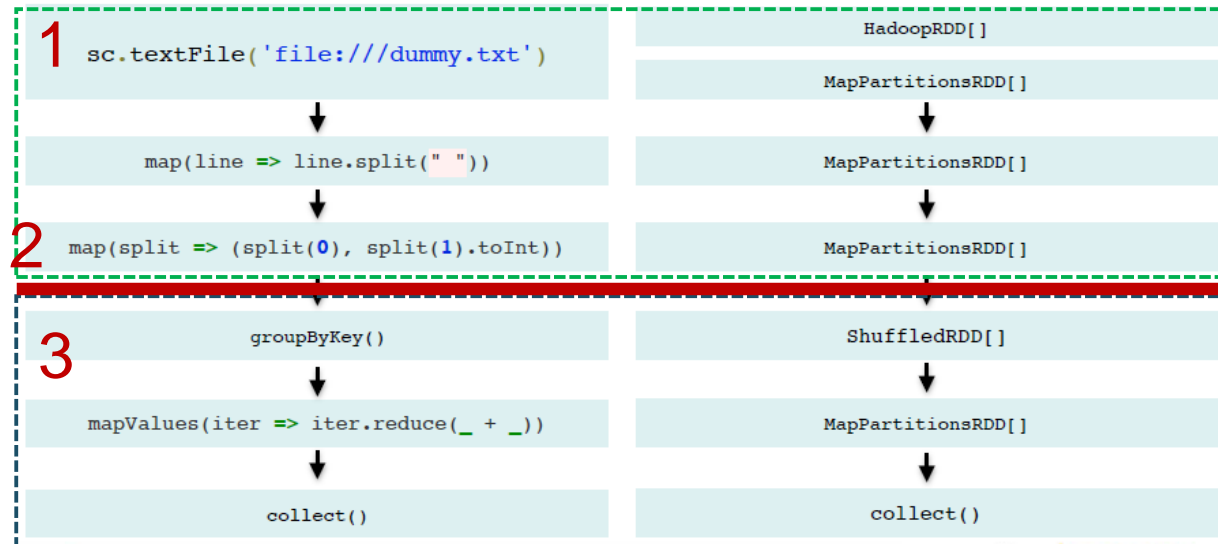


RDD | БАРЬЕР



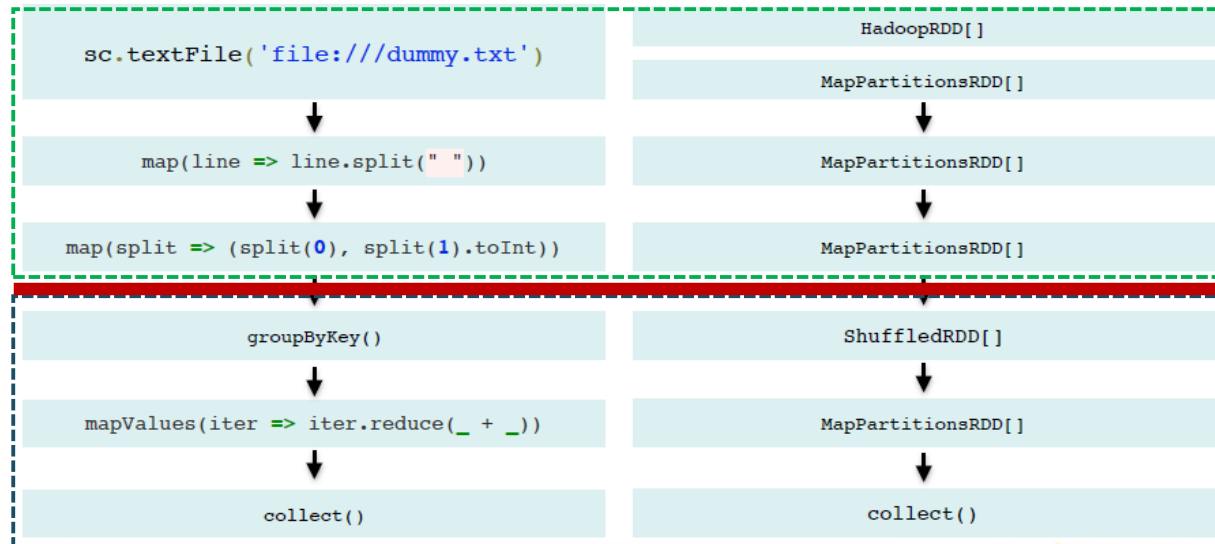
Shuffle
барьер

RDD | ВЫПОЛНЕНИЕ



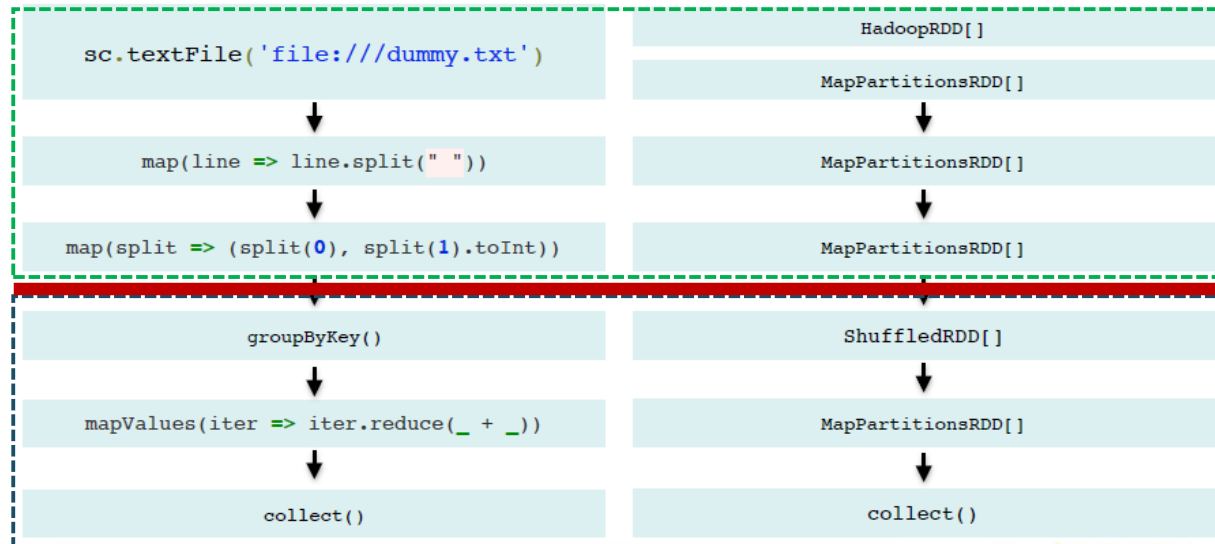
- Разбиение уровней на задачи для Executor's

RDD | ВЫПОЛНЕНИЕ



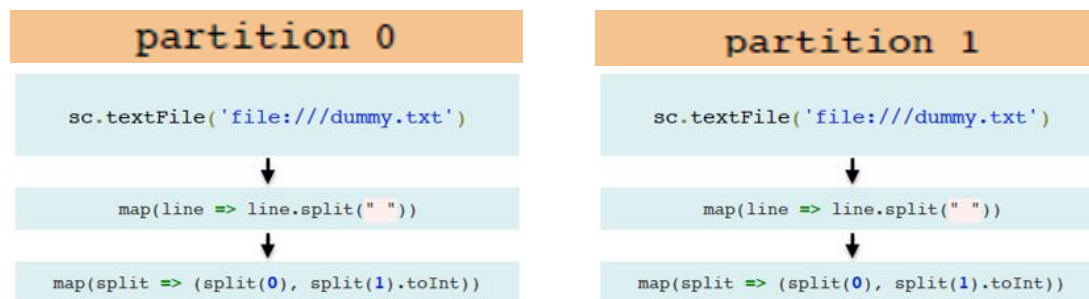
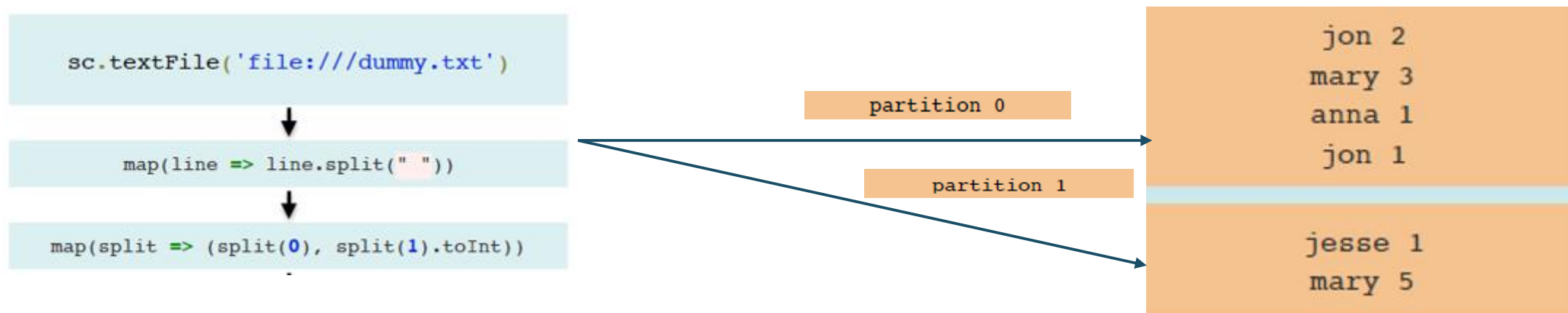
- Разбиение уровней на задачи для Executor's
- Задача – это процесс партиционирования данных и вычисления

RDD | ВЫПОЛНЕНИЕ



- Разбиение уровней на задачи для Executor's
- Задача – это процесс партиционирования данных и вычисления
- Выполнение каждой задачи

RDD | ПАРТИЦИОНИРОВАНИЕ ЗАДАЧ



RDD | SHUFFLE

jon 2
mary 3
anna 1
jon 1

jesse 1
mary 5

`groupByKey()`



`mapValues(iter => iter.reduce(_ + _))`

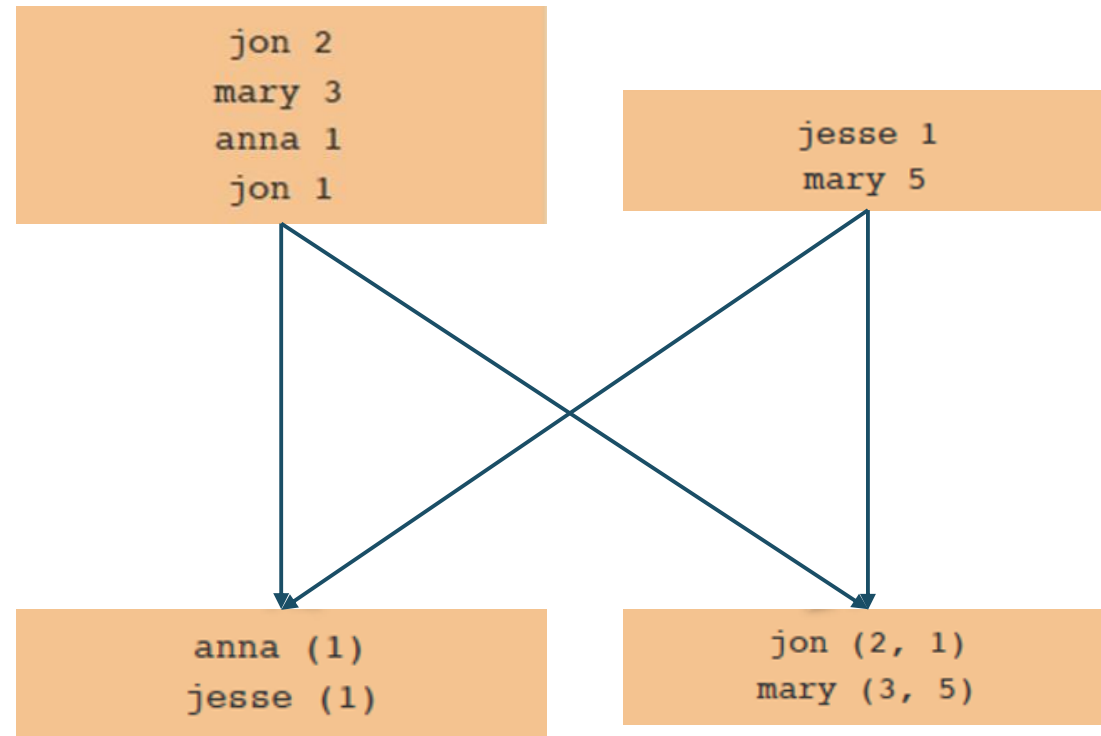
RDD | SHUFFLE

```
groupByKey()
```



```
mapValues(iter => iter.reduce(_ + _))
```

- Перераспределение данных по партициям
- Hash key для создания бакетов
- Выполнение процесса с записью на диск temp файлов (как Hadoop)



RDD?

- Список партиций из данных в виде картежей
- Список зависимостей для выполнения задачи
- Функция для вычисления
(для каждой партиции, в парадигме функционального программирования)
- Партиционирование для оптимизации вычислений
- Список «лучших» расположений на кластере для каждой партиции
(расположение – путь на ноде / диске)

RDD?

- Список партиций из данных в виде картежей
- Список зависимостей для выполнения задачи
- Функция для вычисления
(для каждой партиции, в парадигме функционального программирования)

Не обязательные объекты:

- Партиционирование для оптимизации вычислений
- Список «лучших» расположений на кластере для каждой партиции
(расположение – путь на ноде / диске)

RDD TRANSFORMATIONS

RDD ACTIONS



RDD ТЕРМИНЫ

TRANSFORMATION

«Ленивое» вычисление. Return – **новый RDD**

ACTION

Запускает **выполнение** вычислений над данными.
Return – **финальное значение (на драйвер)**

RDD TRANSFORMATION

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

2
`squared = nums.map(lambda x: x*x) # => {1, 4, 9}`

3
`even = squared.filter(lambda x: x % 2 == 0) # => [4]`

4
`nums.flatMap(lambda x: range(x)) # => {0, 0, 1, 0, 1, 2}`

RDD TRANSFORMATION

1

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

2

```
squared = nums.map(lambda x: x*x) # => {1, 4, 9}
```

3

```
even = squared.filter(lambda x: x % 2 == 0) # => [4]
```

4

```
nums.flatMap(lambda x: range(x)) # => {0, 0, 1, 0, 1, 2}
```

Количество вычислений = 1!

RDD ACTION

1

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

2

```
nums.collect() # => [1, 2, 3]
```

3

```
nums.take(2) # => [1, 2]
```

4

```
nums.count() # => 3
```

5

```
nums.reduce(lambda: x, y: x + y) # => 6
```

6

```
nums.saveAsTextFile("hdfs://file.txt")
```

RDD ACTION

1

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

2

```
nums.collect() # => [1, 2, 3]
```

3

```
nums.take(2) # => [1, 2]
```

4

```
nums.count() # => 3
```

5

```
nums.reduce(lambda: x, y: x + y) # => 6
```

6

```
nums.saveAsTextFile("hdfs://file.txt")
```

Количество
вычислений = 6

RDD SCRIPT

```
B [1]: import random

flips = 100000

coins = range(1, flips + 1)

heads = (
    sc.parallelize(coins)
    .map(lambda i: random.random())
    .filter(lambda r: r < 0.51)
    .count()
)
```

Annotations:

- Generator: points to `coins = range(1, flips + 1)`
- Создаем RDD: points to `sc.parallelize(coins)`
- Transformations: points to `.map()`, `.filter()`, and `.count()`
- Action: points to `.count()`

RDD SCRIPT

```
B [1]: import random

flips = 100000

coins = range(1, flips + 1)

heads = (
    sc.parallelize(coins)
    .map(lambda i: random.random())
    .filter(lambda r: r < 0.51)
    .count()
)
```

- Создаем функцию
- Применяем её к объекту

RDD SCRIPT

```
B [1]: import random

flips = 100000

coins = range(1, flips + 1)

heads = (
    sc.parallelize(coins)
    .map(lambda i: random.random())
    .filter(lambda r: r < 0.51)
    .count()
)
```


RDD SCRIPT

```
[1]: import random

flips = 100000

coins = range(1, flips + 1)

heads = (
    sc.parallelize(coins)
    .map(lambda i: random.random())
    .filter(lambda r: r < 0.51)
    .count()
)
```

==

```
[2]: import random

flips = 100000

coins = range(1, flips + 1)

rdd = sc.parallelize(coins)

flips_rdd = rdd.map(lambda i: random.random())
heads_rdd = flips_rdd.filter(lambda r: r < 0.51)

heads = heads_rdd.count()
```

RDD ПРИЧИНЫ ИСПОЛЬЗОВАНИЯ

НУЖНЫ НЕ ИЗМЕНЯЕМЫЕ ОБЪЕКТЫ

НУЖНА ТИПИЗАЦИЯ `RDD[int]`,
`RDD[string]`

FAULT TOLERANCE

ПРОВЕСТИ КРУПНЫЕ (ГРУБЫЕ) ИЗМЕНЕНИЯ ПО
ВСЕМУ НАБОРУ ДАННЫХ

ВАЖНО ПАРТИЦИОНИРОВАНИЕ, РАСПРЕДЕЛЕНИЕ ПО НОДАМ
(ОПРЕДЕЛЕННОЕ)

ЕСТЬ ОГРАНИЧЕНИЯ ПО
РЕСУРСАМ

КОГДА НУЖНО ИСПОЛЬЗОВАТЬ ДРУГИЕ
ОПТИМИЗАТОРЫ (НЕ CATALYST)

RDD НЕ ИЗБАВЛЯЕТ ОТ ПРОБЛЕМ

НЕТ ОПТИМИЗАТОРА (В DATAFRAME / DATASET ИСПОЛЬЗУЕТСЯ CATALYST)

НУЖНО СЛЕДИТЬ ЗА ТИПАМИ ДАННЫХ

ДЕГРАДАЦИЯ ДАННЫХ ПРИ МАЛОМ КОЛ-ВЕ ОЗУ (КОГДА IN-MEMORY)

НУЖНО ИСПОЛЬЗОВАТЬ GARBAGE COLLECTION

ПОПРОБУЕМ
САМОСТОЯТЕЛЬНО

ТЫ НЕ ДЕЛАЕШЬ ЭТО НЕПРАВИЛЬНО



**ЕСЛИ НИКТО НЕ ЗНАЕТ, ЧТО
КОНКРЕТНО ТЫ ДЕЛАЕШЬ**

SPARK ПРОЕКТ



SPARK ПРОЕКТ

- Цель: Разработать Data Quality «платформу» на Apache Spark