Spark. RDD / SQL

SELEZNEV ARTEM
HEAD OF CVM ANALYTICS @ MAGNIT

APACHE SPARK

ФУНКЦИОНАЛЬНАЯ РАЗНИЦА

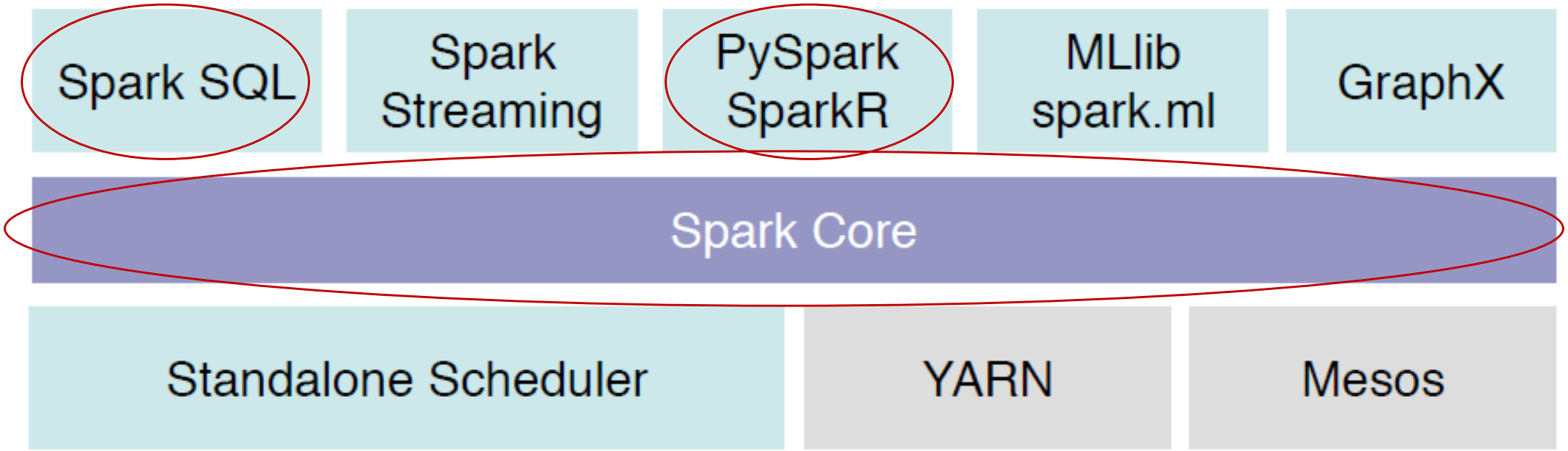

```
map()
reduce()
```

ФУНКЦИОНАЛЬНАЯ РАЗНИЦА


```
map()        reduce()
filter()     sortBy()
join()       groupByKey()
first()      count()
```


```
map()
reduce()
```

APACHE SPARK



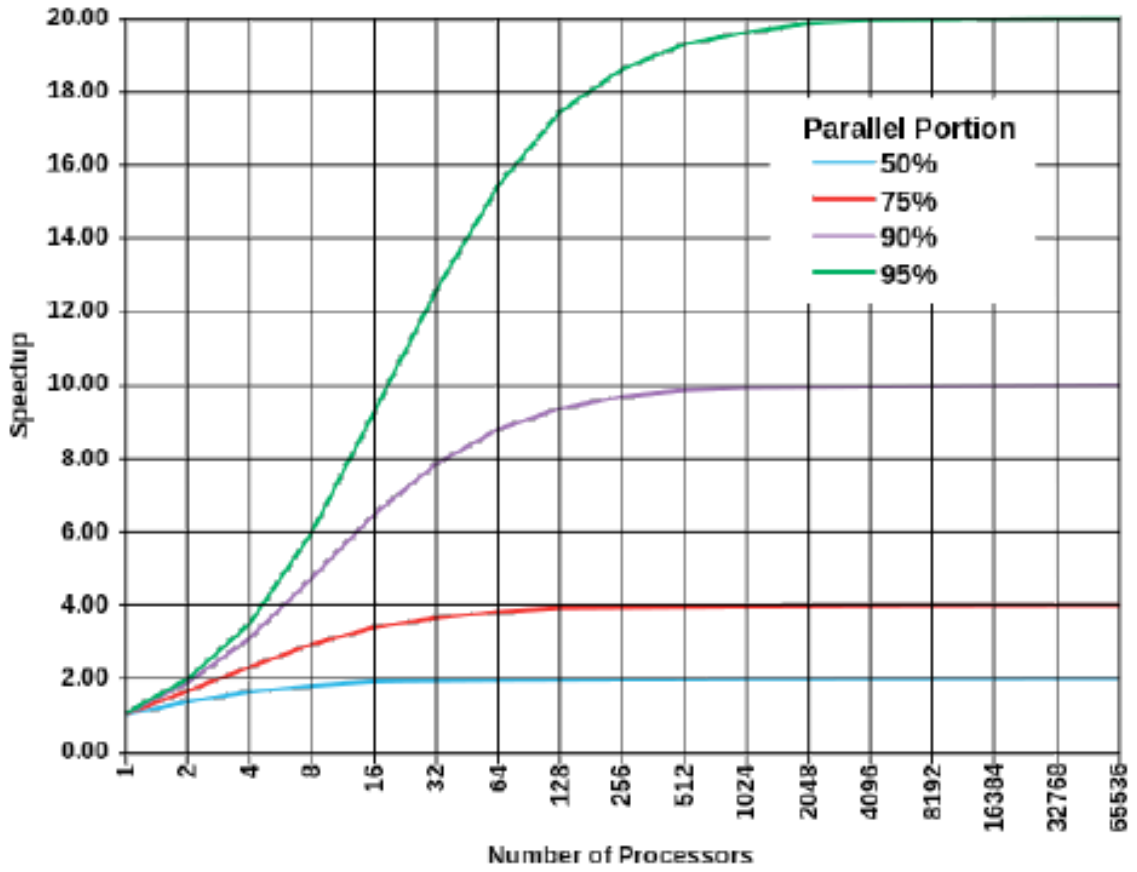
APACHE SPARK – ЗАВИСИМОСТИ

РЕСУРСЫ

ПАРТИЦИОНИРОВАНИЕ

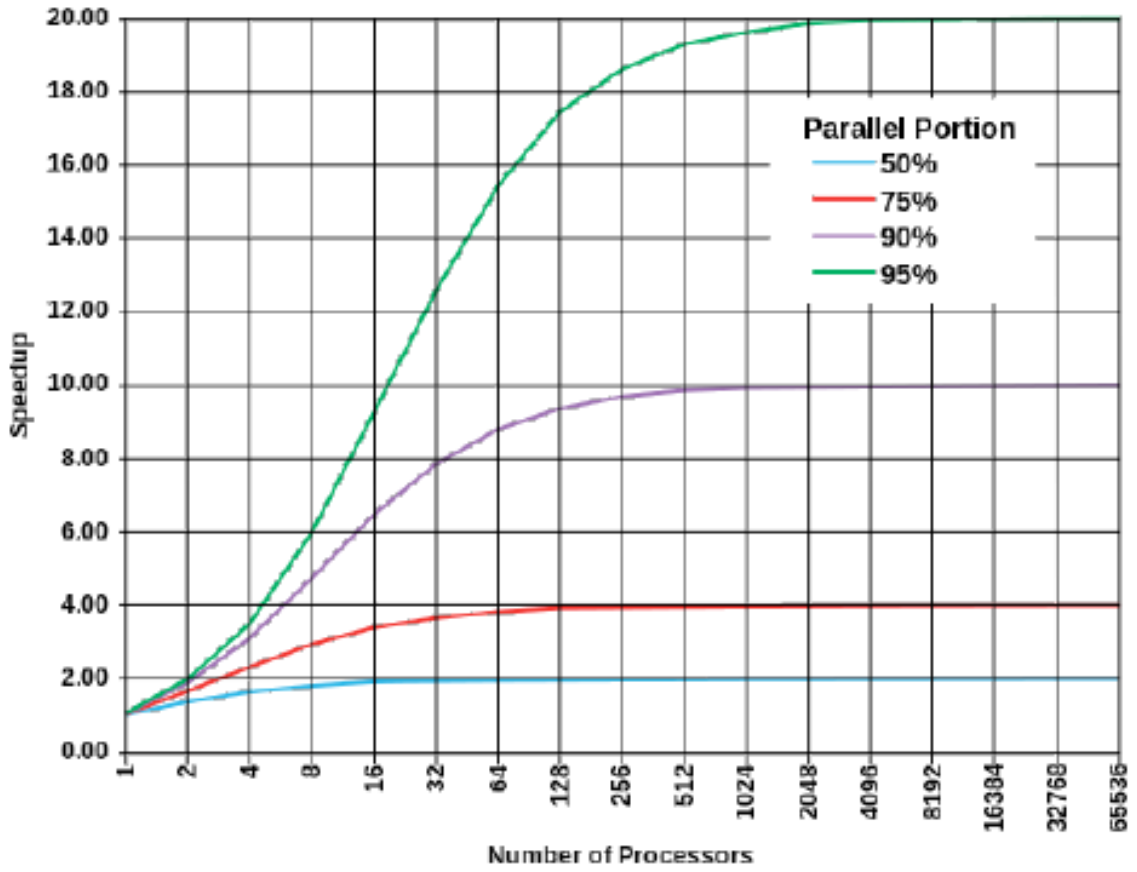
APACHE SPARK RUN-RESOURCES

ЗАКОН АМДАЛА



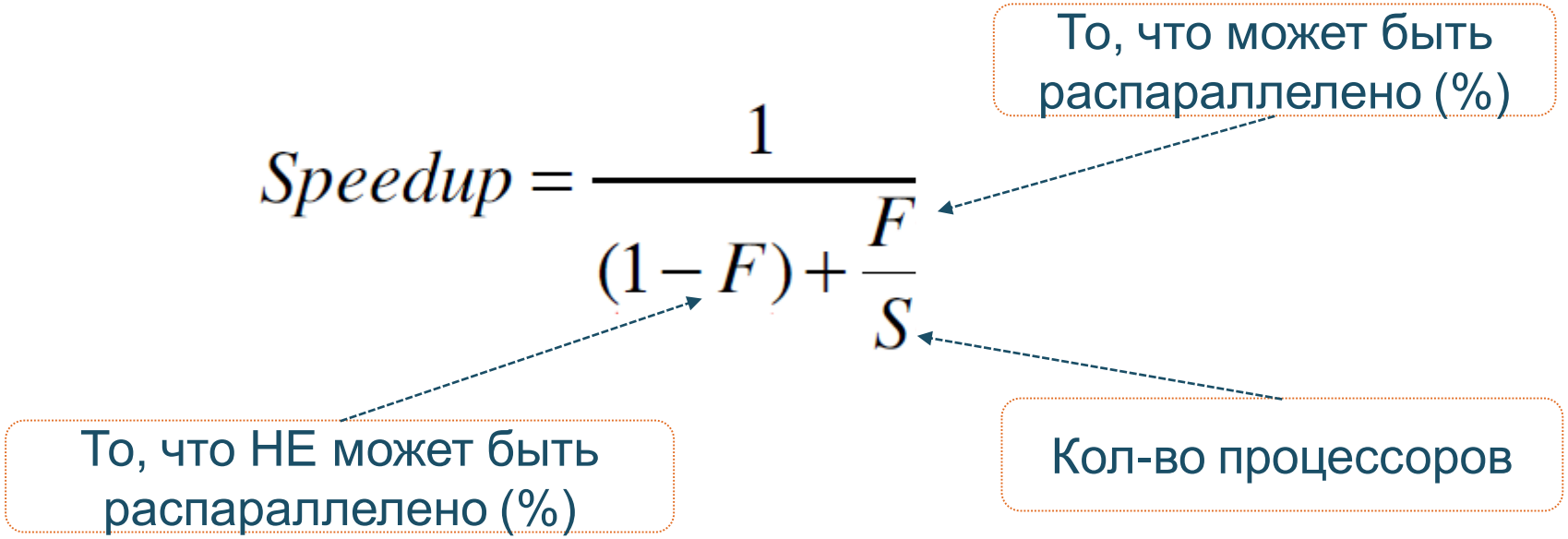
- Есть ограничение в приросте производительности
- Все будет работать не быстрее самого медленно

ЗАКОН АМДАЛА

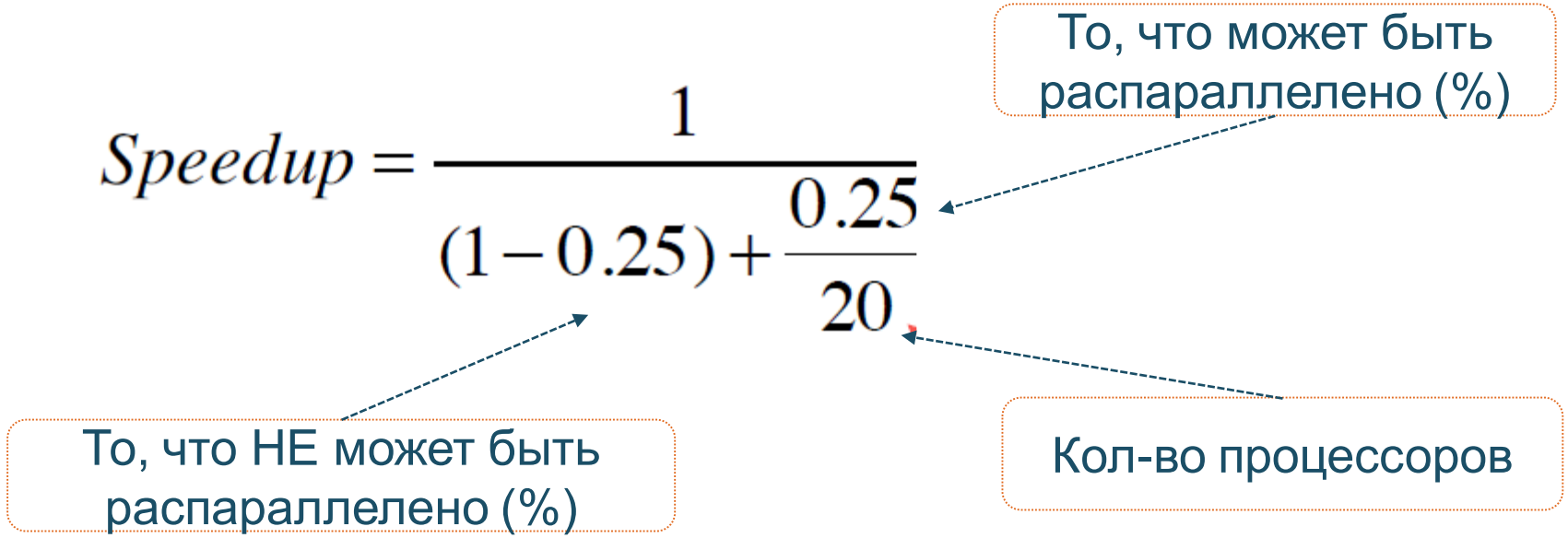


$$Speedup = \frac{1}{(1-F)+\frac{F}{S}}$$

ЗАКОН АМДАЛА



ЗАКОН АМДАЛА



20 - процессоров
25% - распараллелено

ЗАКОН АМДАЛА

$$Speedup = \frac{1}{0.75 + \frac{0.25}{20}} = 1.31$$

То, что может быть распараллелено (%)

То, что НЕ может быть распараллелено (%)

Кол-во процессоров

20 - процессоров
25% - распараллелено
1.31 – быстрее процесс

APACHE SPARK PARTITIONS

APACHE SPARK
(2 СТРУКТУРЫ ДЛЯ PYTHON)

	RDD	DataFrame
Immutability	✓	✓
Schéma	X	✓
Apache Spark 1	✓	✓
Apache Spark 2	✓	✓ (it does not exist in Java anymore)
Performance optimization	X	✓
Level	Low	High (built upon RDD)
Typed	✓	X
Syntax Error	Compile time	Compile time
Analysis Error	Compile time	Runtime

APACHE SPARK – ТЕРМИНЫ

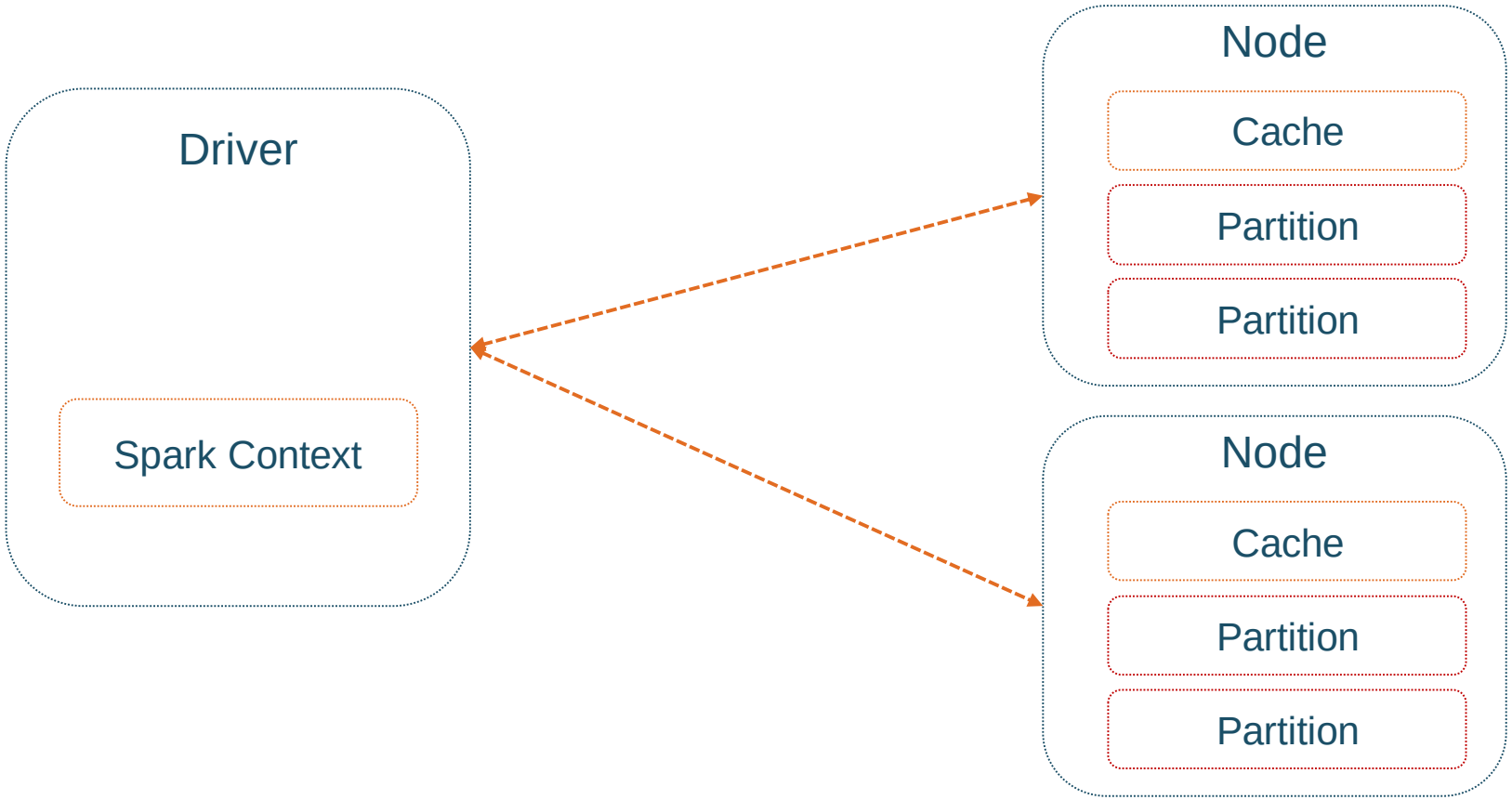
DRIVER

WORKER

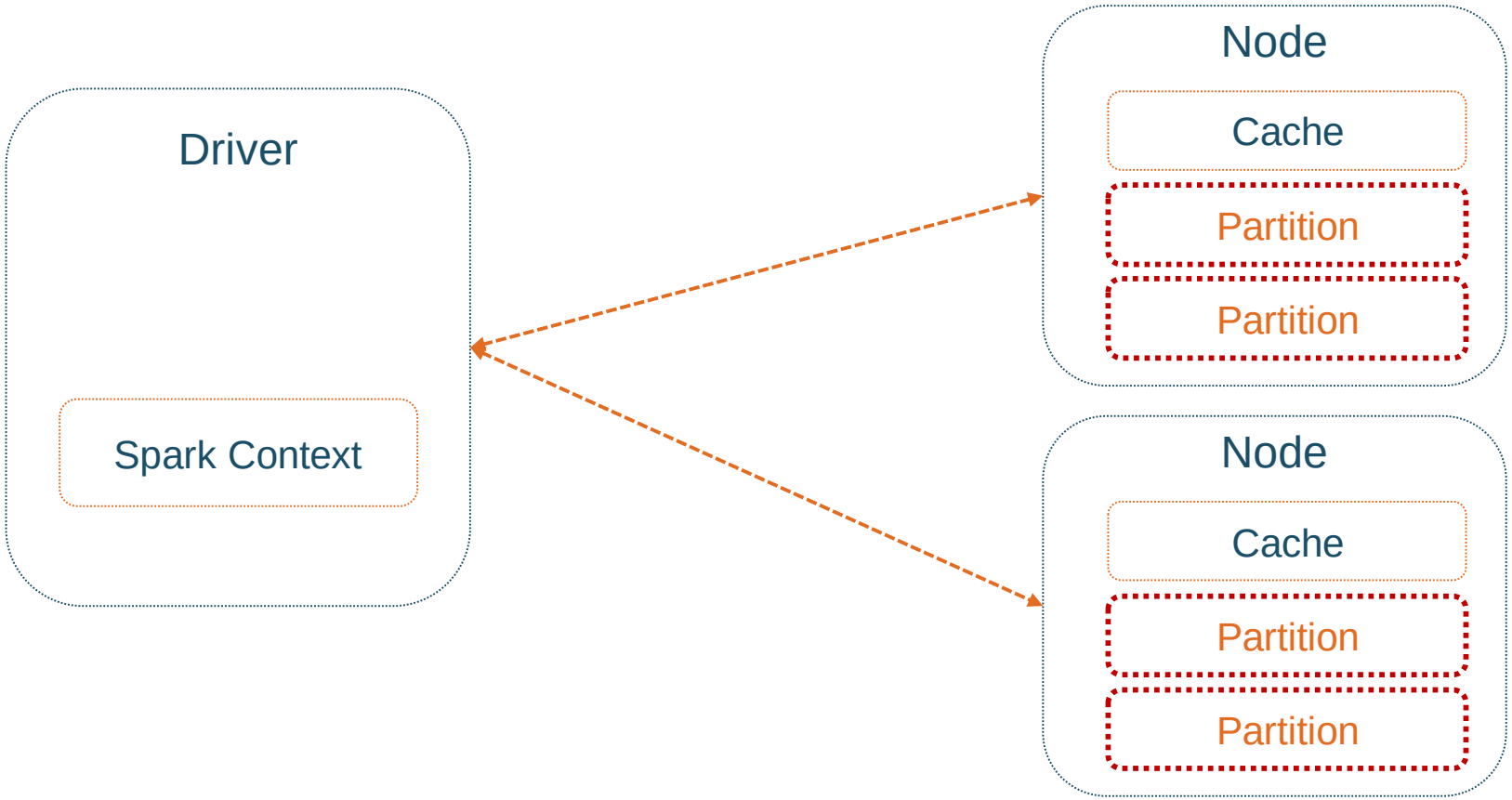
APACHE SPARK – ТЕРМИНЫ

DRIVER	Процесс содержащий Spark Context
WORKER	Приложение (node) выполняющее код/команды

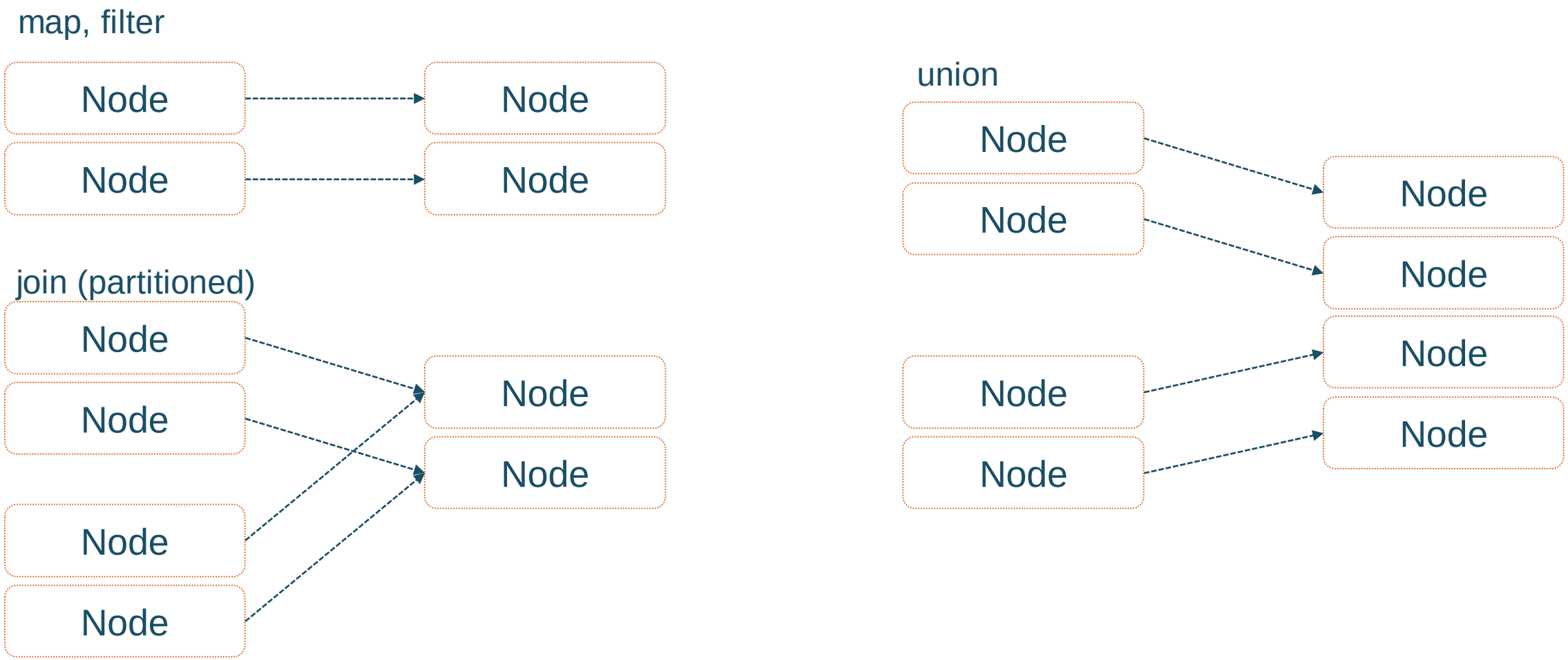
РАБОТА КОНТЕКСТА



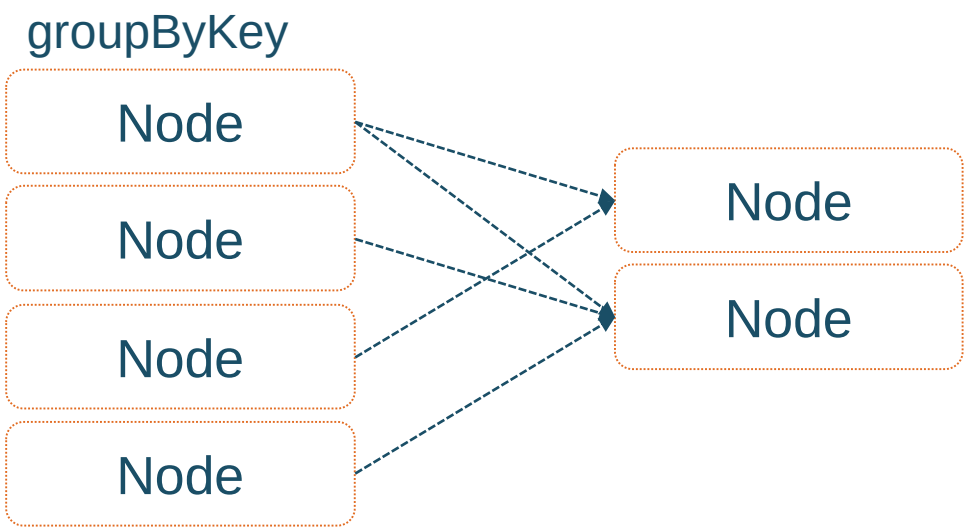
РАБОТА КОНТЕКСТА (?)

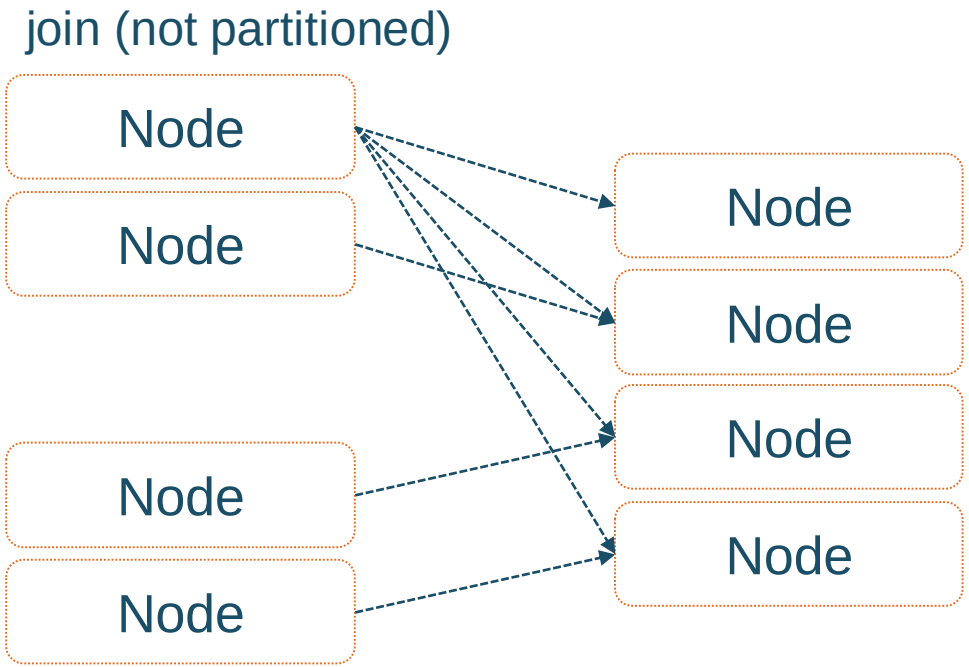


ЗАВИСИМОСТЬ ОТ ПАРТИЦИЙ
(НЕ БОЛЬШАЯ)

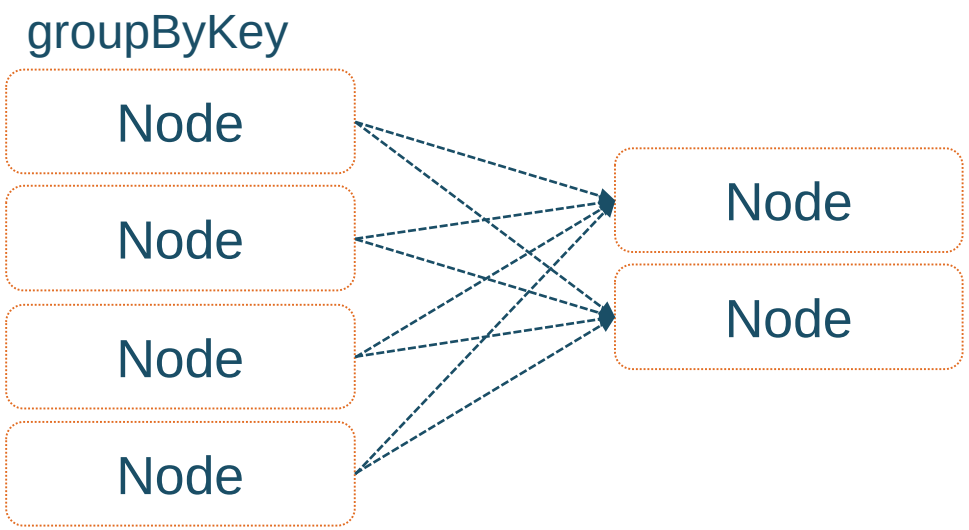


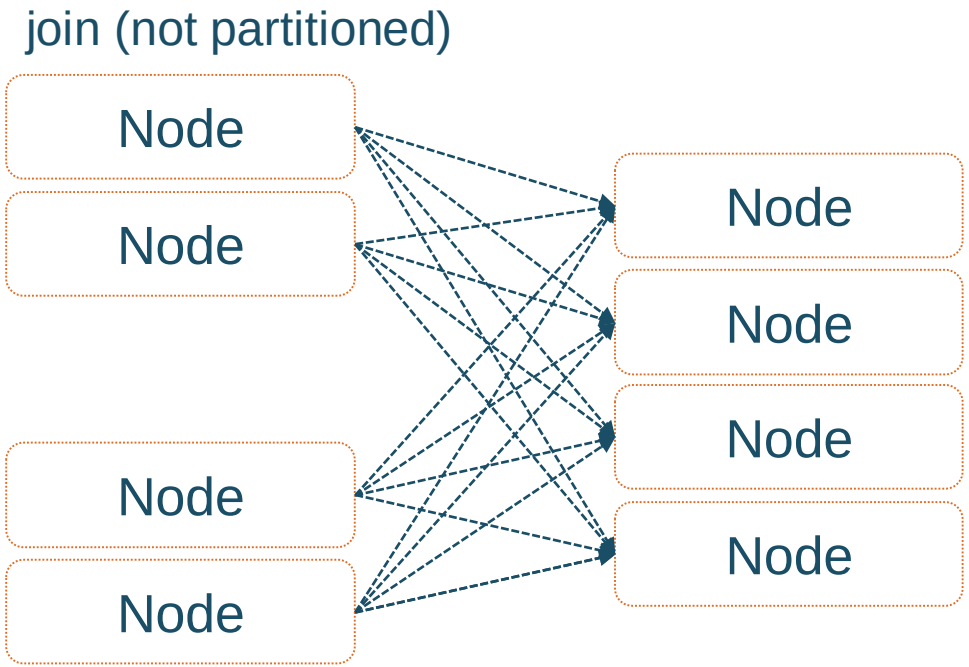
ЗАВИСИМОСТЬ ОТ ПАРТИЦИЙ
(БОЛЬШАЯ)



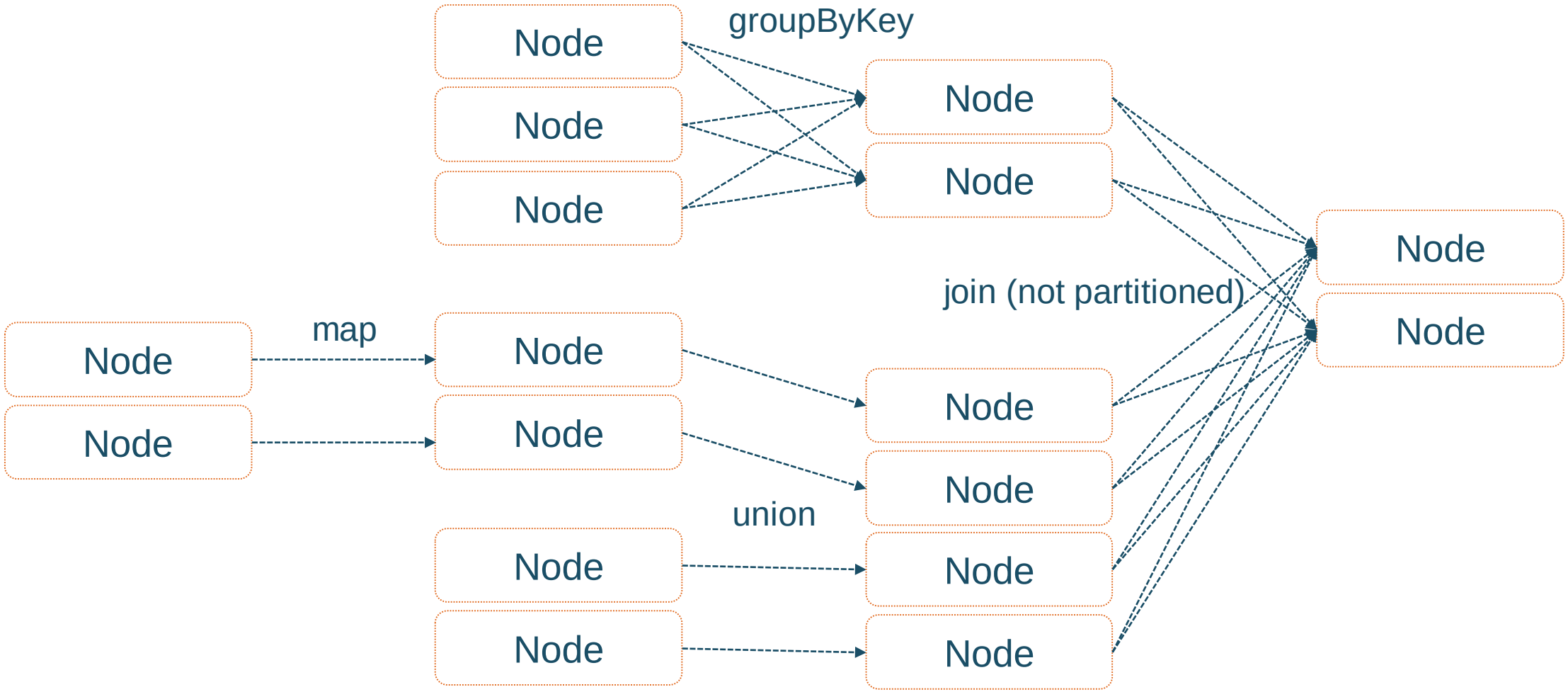


ЗАВИСИМОСТЬ ОТ ПАРТИЦИЙ
(БОЛЬШАЯ)

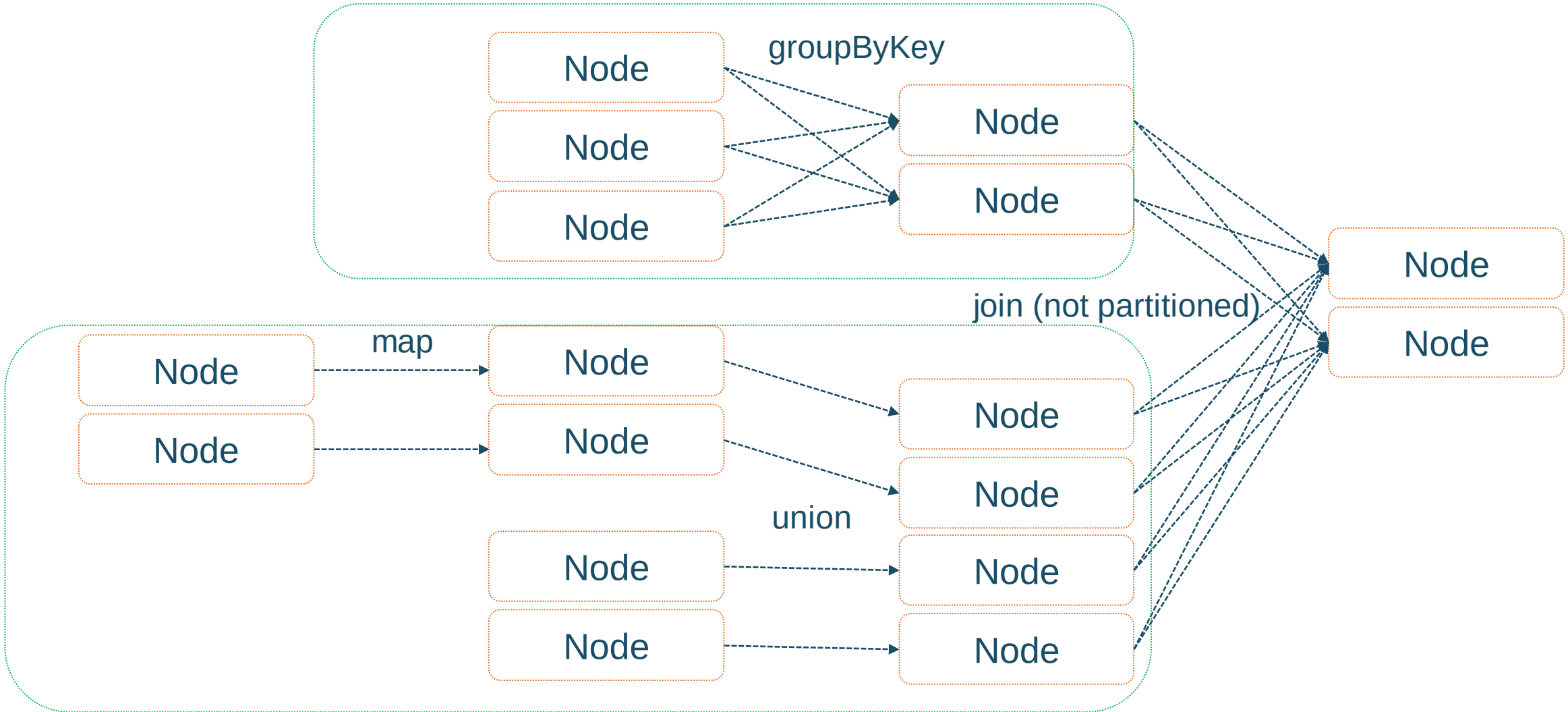




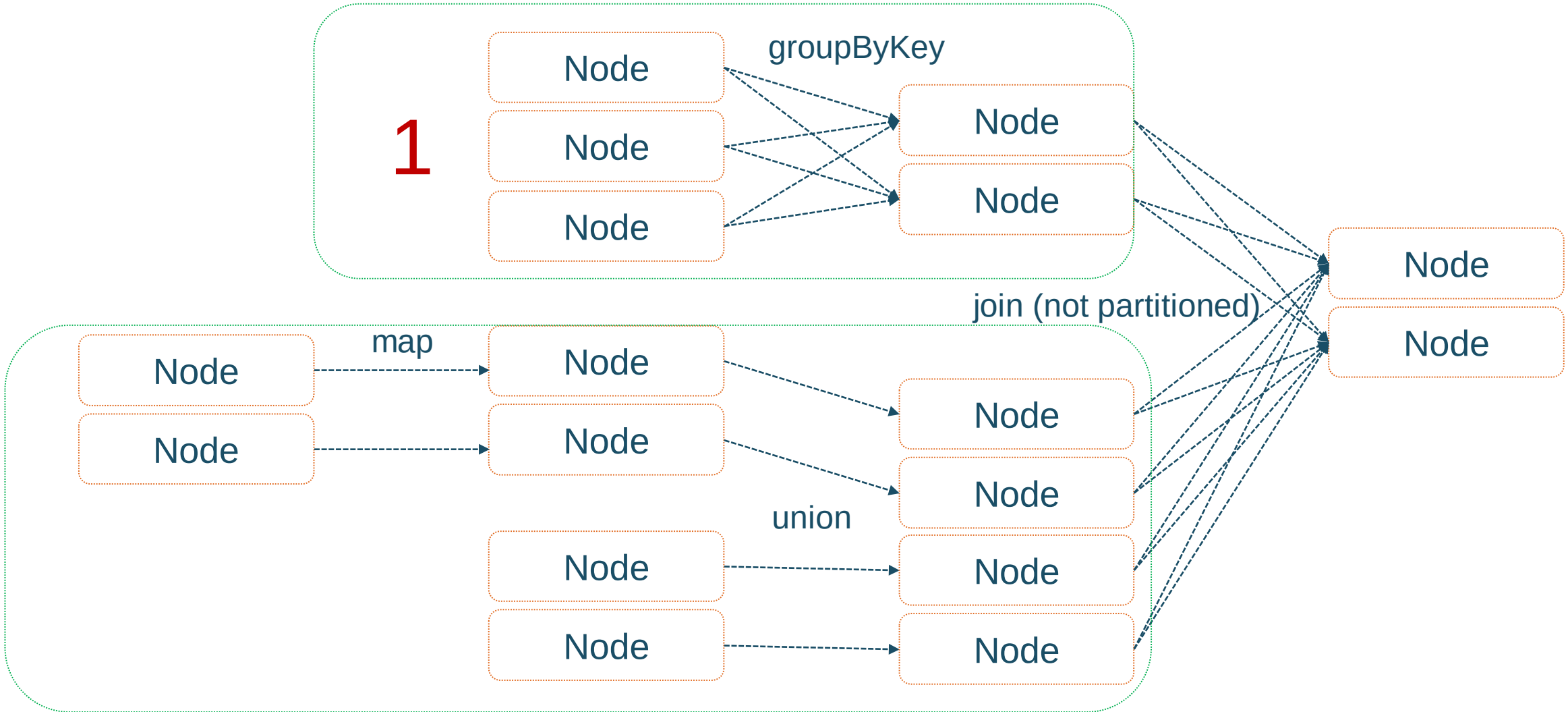
ЗАВИСИМОСТЬ ОТ ПАРТИЦИЙ
(В ПРОЦЕССЕ ЗАДАЧИ)



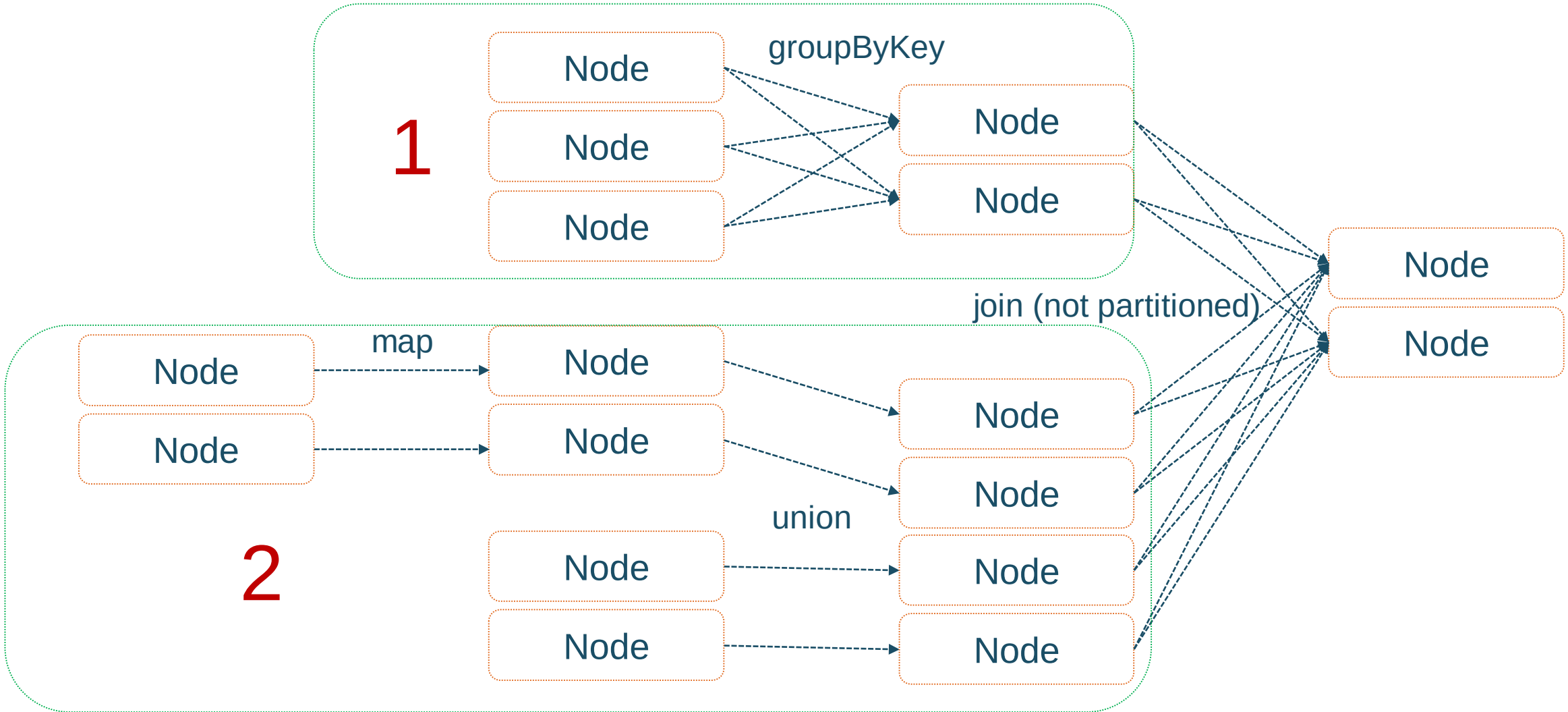
ЗАВИСИМОСТЬ ОТ ПАРТИЦИЙ
(В ПРОЦЕССЕ ЗАДАЧИ)



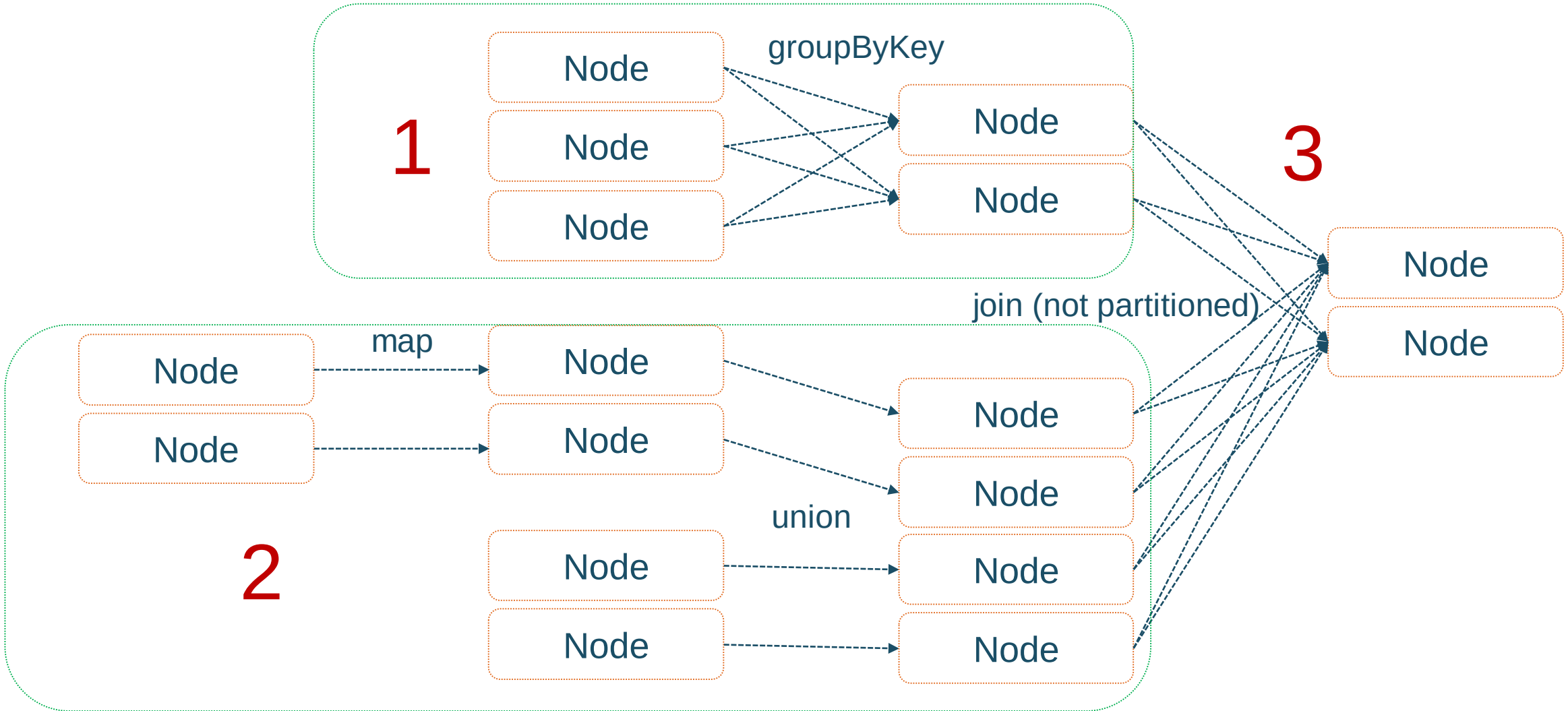
ЗАВИСИМОСТЬ ОТ ПАРТИЦИЙ
(В ПРОЦЕССЕ ЗАДАЧИ)



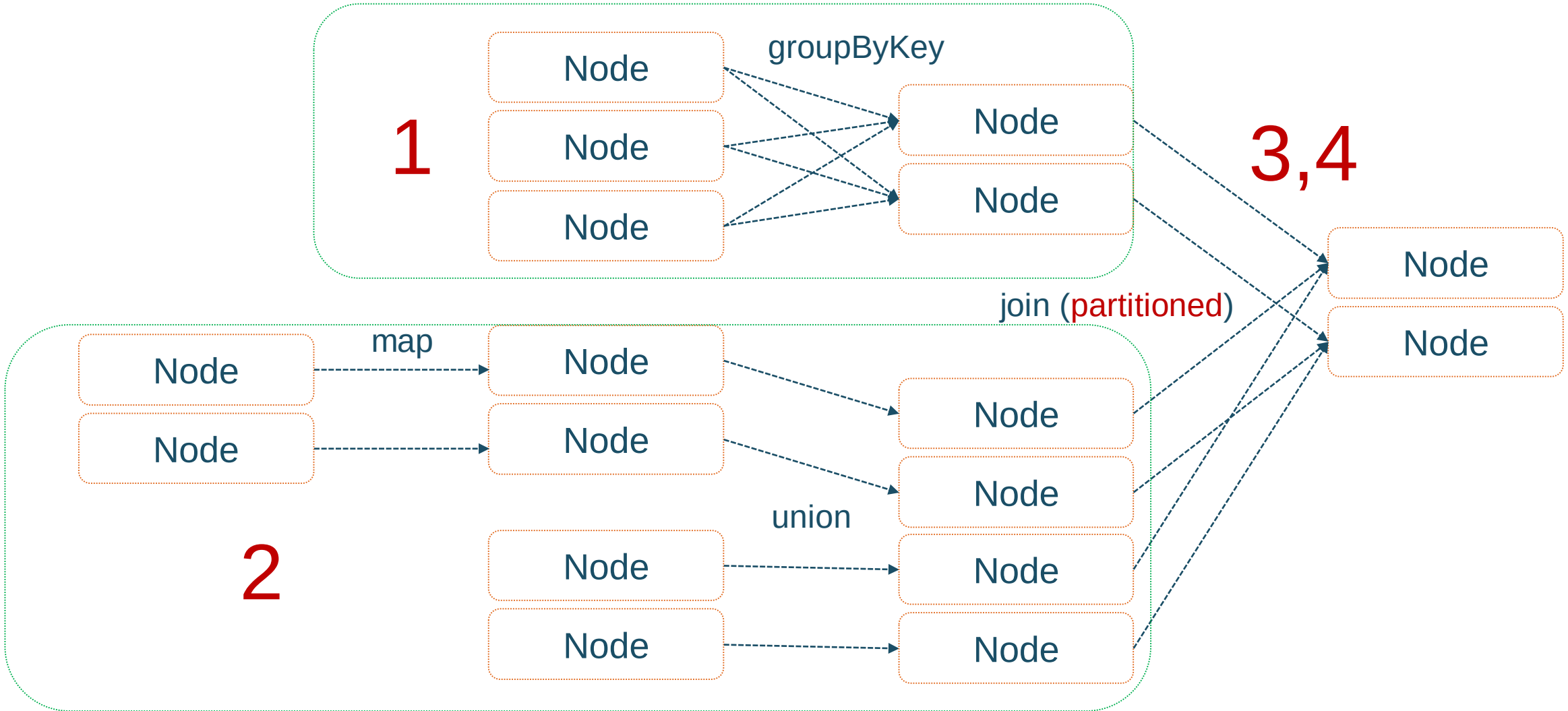
ЗАВИСИМОСТЬ ОТ ПАРТИЦИЙ
(В ПРОЦЕССЕ ЗАДАЧИ)



ЗАВИСИМОСТЬ ОТ ПАРТИЦИЙ
(В ПРОЦЕССЕ ЗАДАЧИ)



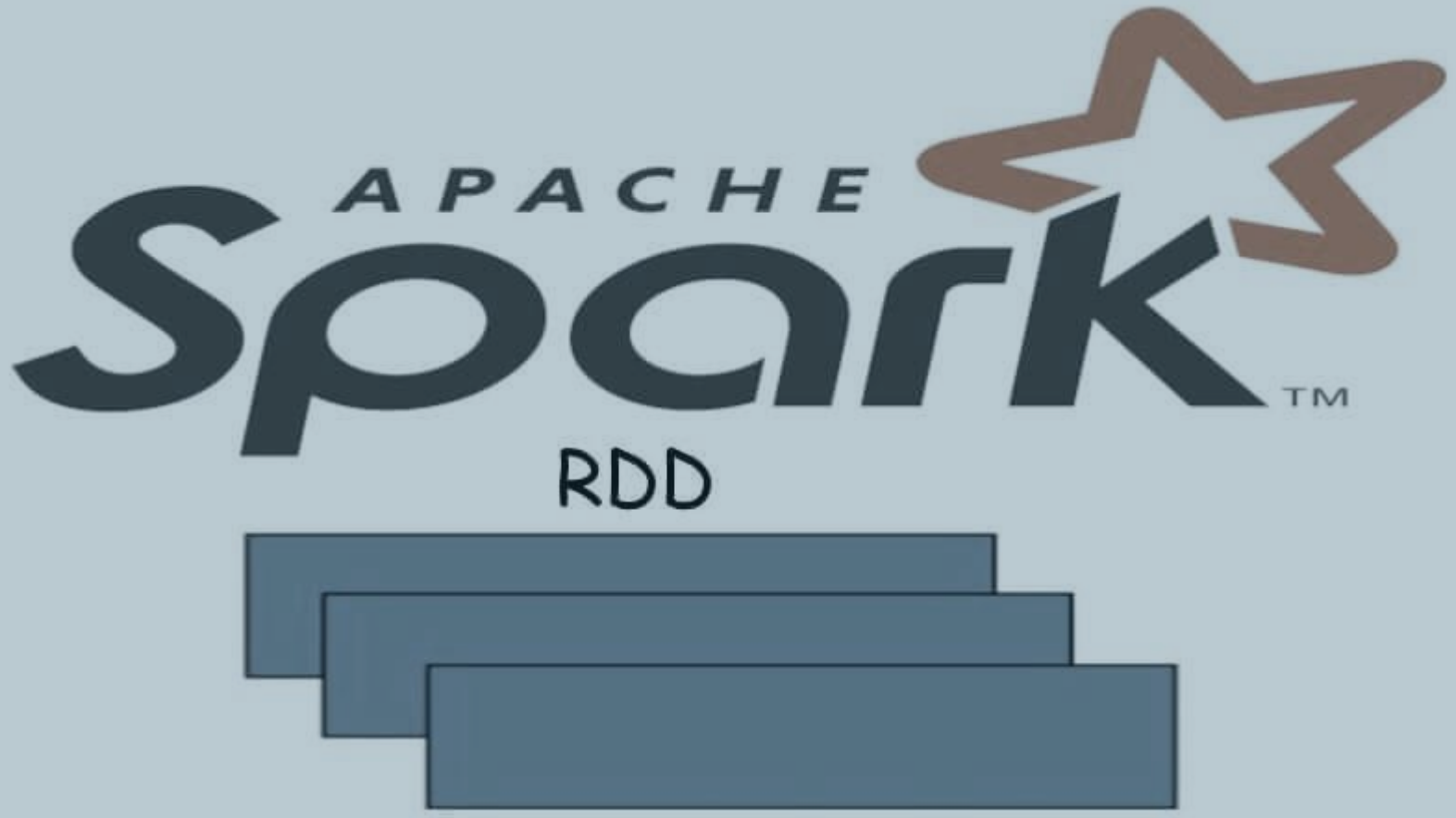
ЗАВИСИМОСТЬ ОТ ПАРТИЦИЙ
(В ПРОЦЕССЕ ЗАДАЧИ)



ФУНКЦИИ УПРАВЛЕНИЯ ПАРТИЦИЯМИ

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

SPARK RDD



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 ПРИЧИНЫ ИСПОЛЬЗОВАНИЯ

ИСПОЛЬЗОВАТЬ BROADCAST

spark.sparkContext.broadcast(VARIABLE)

ИСПОЛЬЗОВАТЬ ACCUMULATOR

spark.sparkContext.accumulator(Type, Func())

ACCUMULATOR

```
accum = sc.accumulator(0)
```

```
from collections import Counter
```

```
class CounterAccumulatorParam(ps.accumulators.AccumulatorParam):
    def zero(self, initialValue):
        return Counter()

    def addInPlace(self, v1, v2):
        v1 += v2
        return v1
```

```
accum = sc.accumulator(Counter(), CounterAccumulatorParam())
```

```
def count_null(record):
    global accum

    c = Counter()

    for key, value in record.items():
            if value == '':
                c[key] += 1

    accum.add(c)
```

```
rdd_dict.foreach(count_null)
```

```
accum.value
```

ACCUMULATOR

```
accum = sc.accumulator(0)
```

```
from collections import Counter
```

```
class CounterAccumulatorParam(ps.accumulators.AccumulatorParam):
    def zero(self, initialValue):
        return Counter()

    def addInPlace(self, v1, v2):
        v1 += v2
        return v1
```

```
accum = sc.accumulator(Counter(), CounterAccumulatorParam())
```

```
def count_null(record):
    global accum

    c = Counter()

    for key, value in record.items():
            if value == '':
                c[key] += 1

    accum.add(c)
```

```
rdd_dict.foreach(count_null)
```

```
accum.value
```

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

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

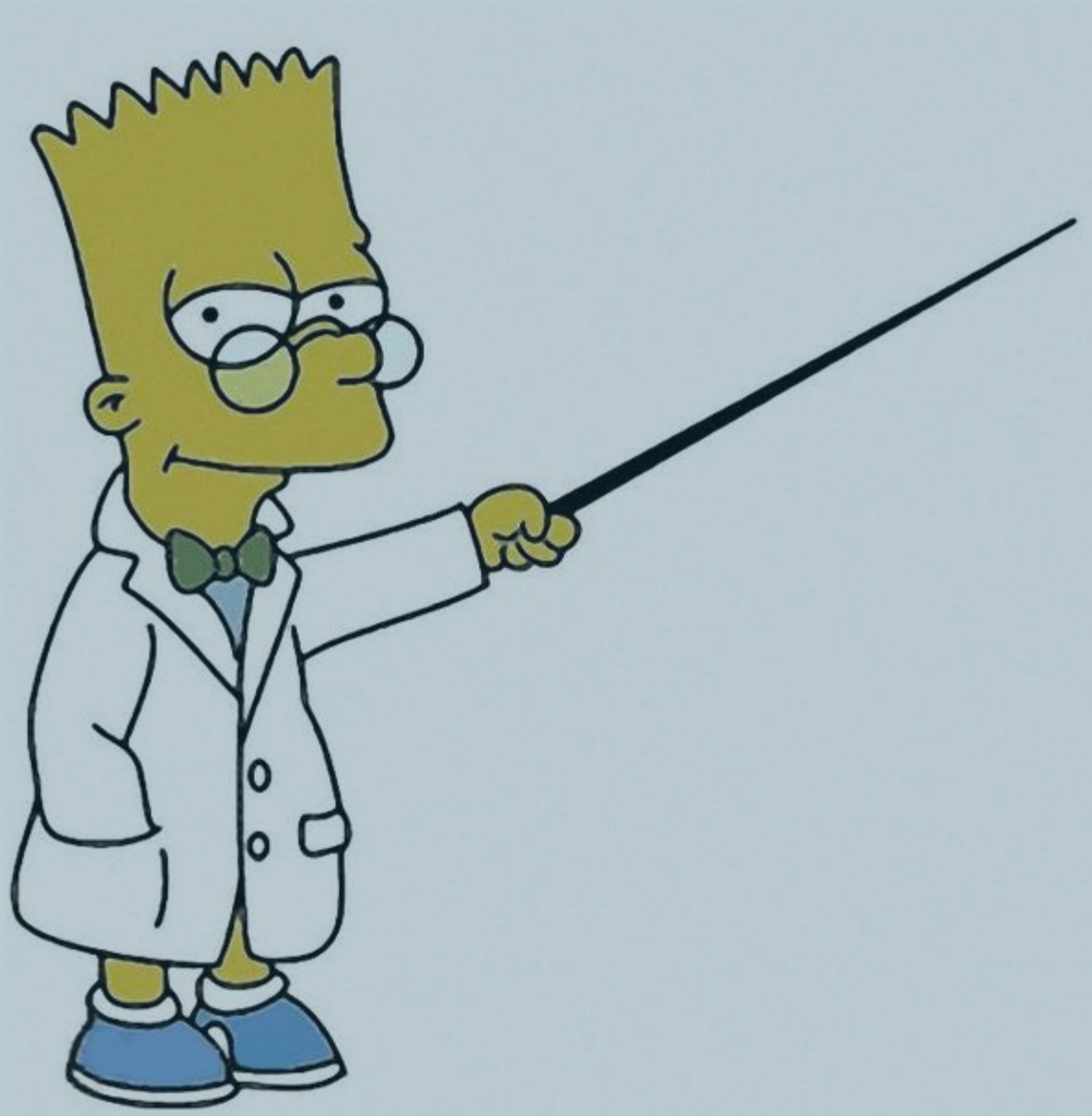
RDD (KEY – VALUE) | MAP – REDUCE

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

RDD НА ПРИМЕРЕ



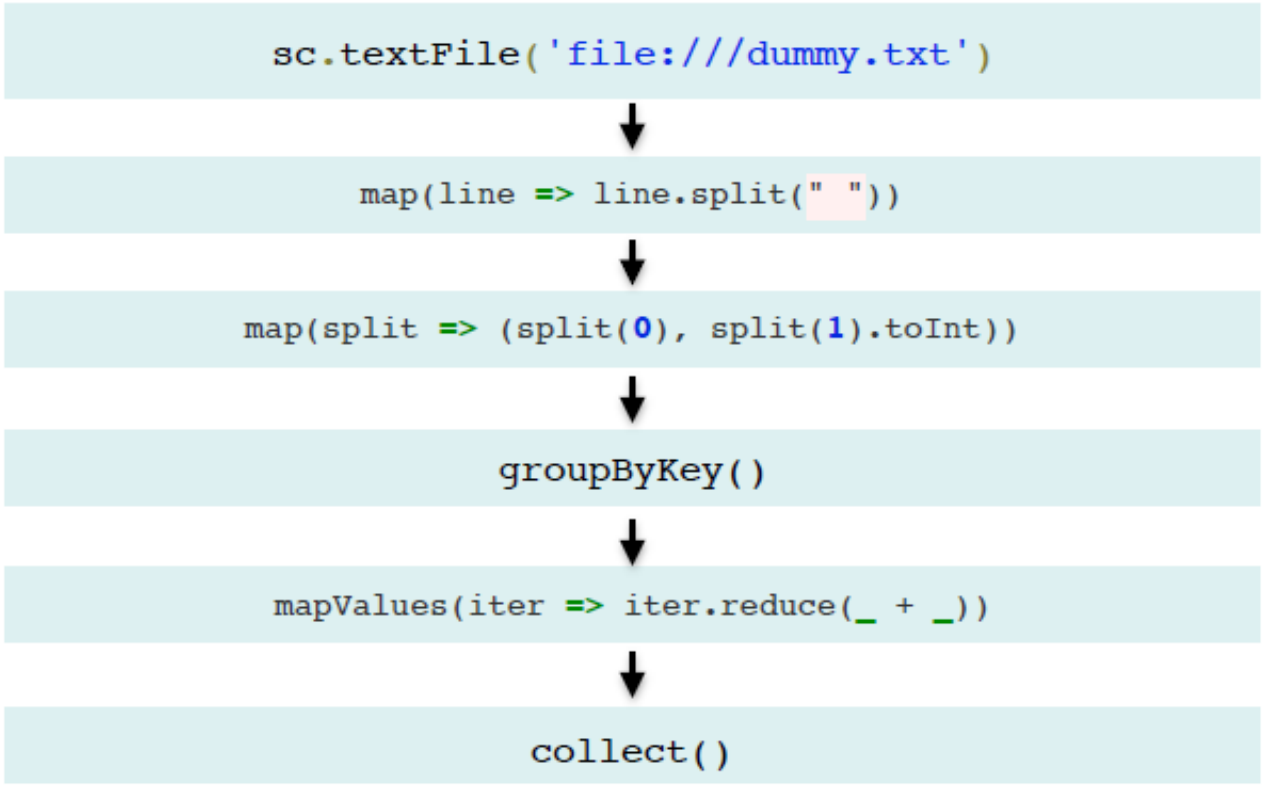
RDD | WORDCOUNT

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

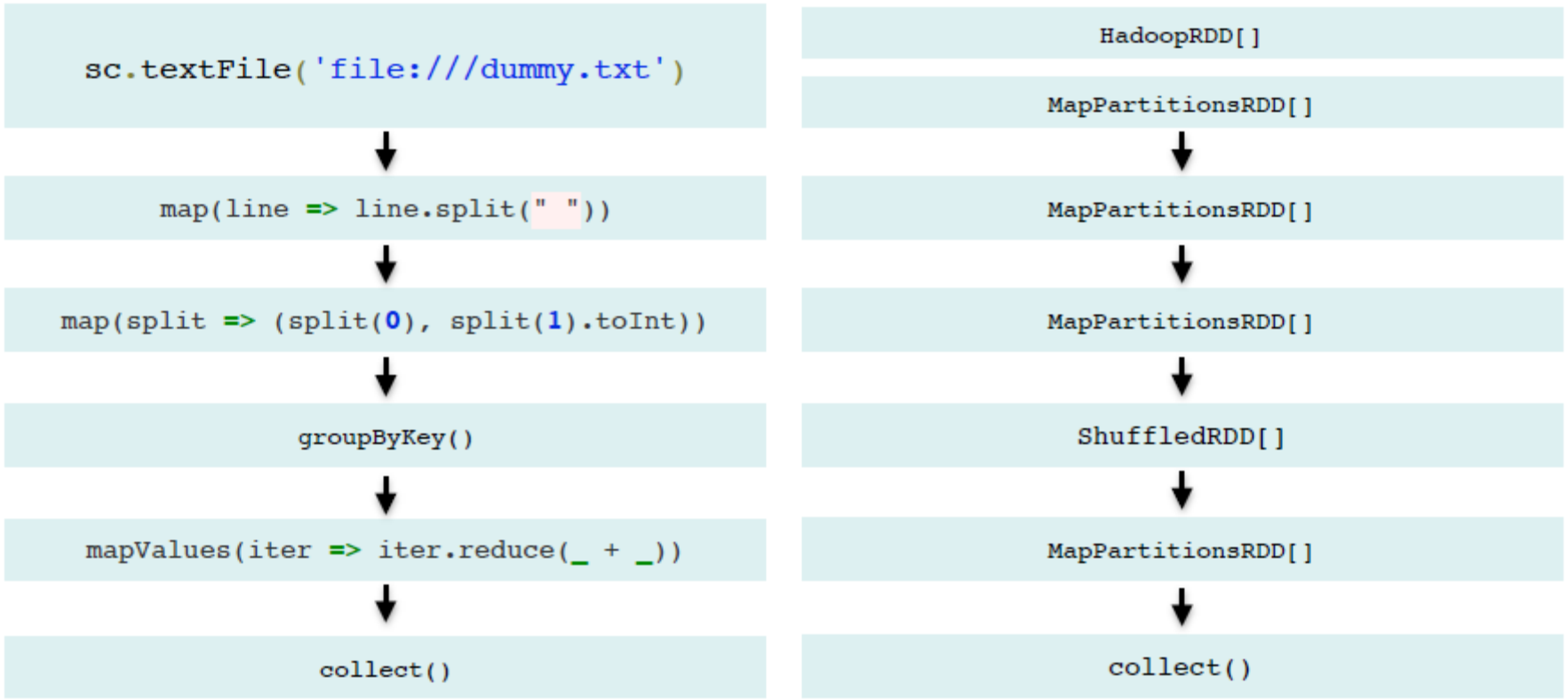
dummy.txt

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

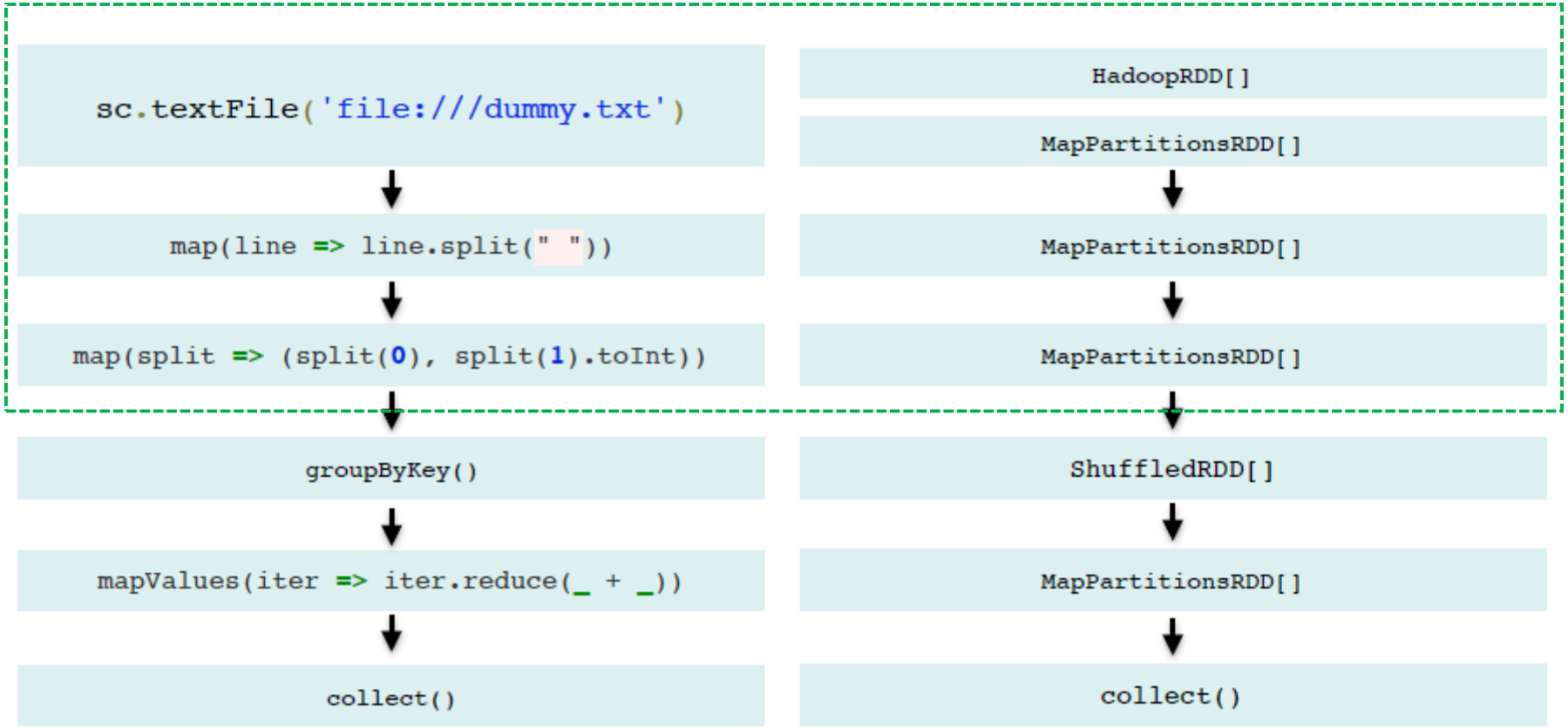
RDD | PLAN


sc.textFile('file:///dummy.txt')
map(line => line.split(" "))
map(split => (split(0), split(1).toInt))
groupByKey()
mapValues(iter => iter.reduce(_ + _))
collect()

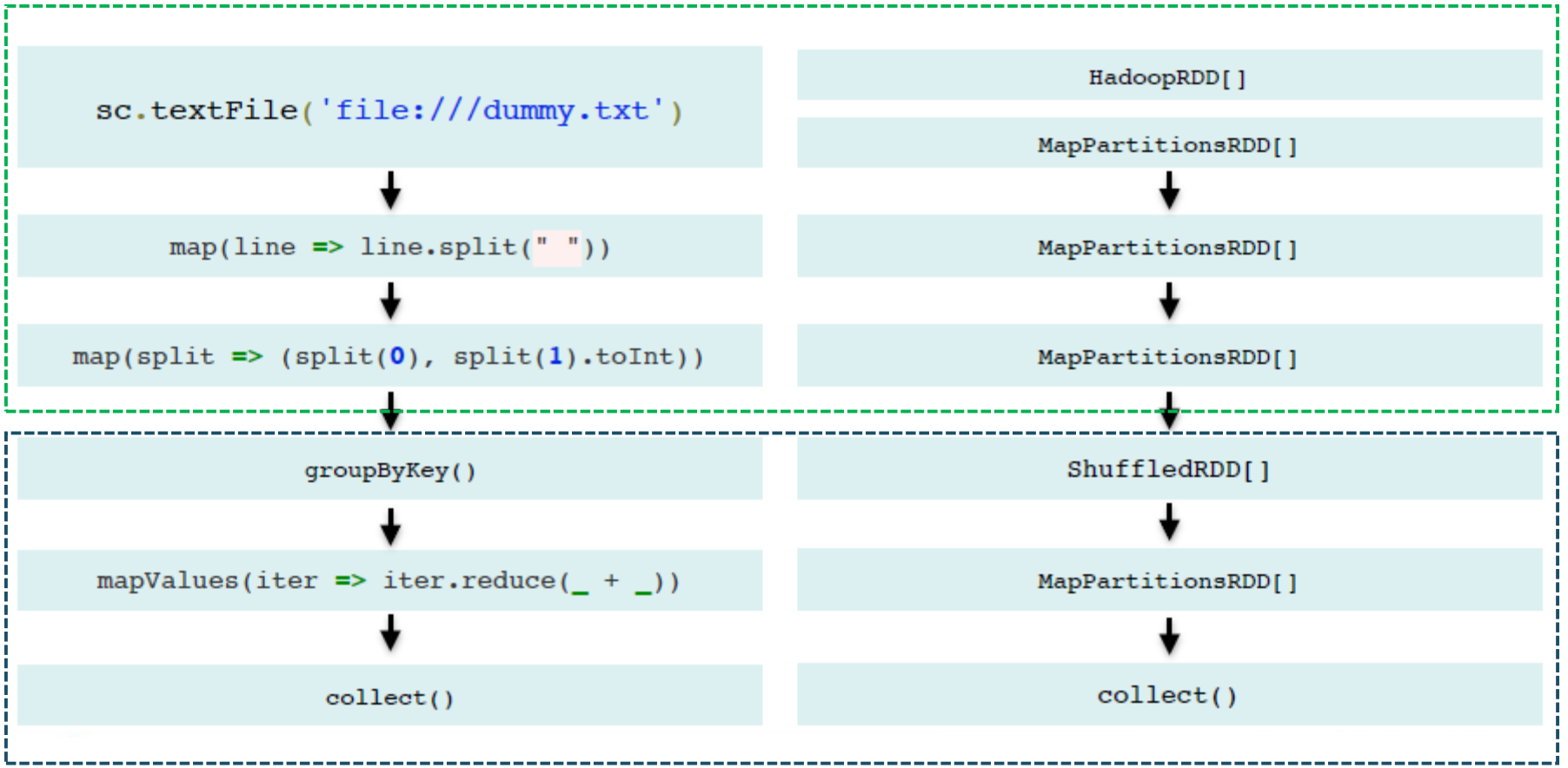
RDD | PLAN RDD


sc.textFile('file:///dummy.txt')
map(line => line.split(" "))
map(split => (split(0), split(1).toInt))
groupByKey()
mapValues(iter => iter.reduce(_ + _))
collect()
HadoopRDD[]
MapPartitionsRDD[]
MapPartitionsRDD[]
MapPartitionsRDD[]
ShuffledRDD[]
MapPartitionsRDD[]
collect()

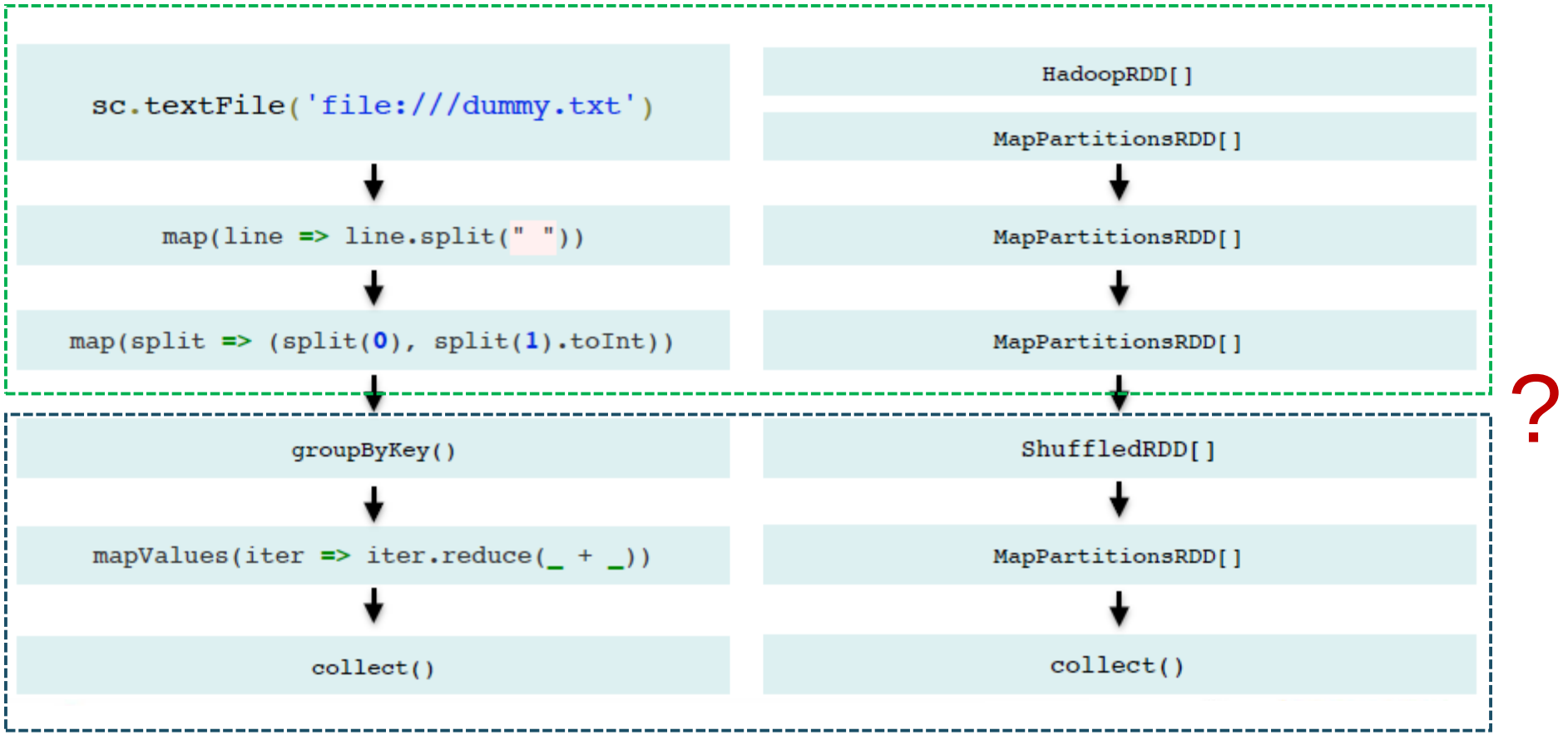
RDD | БАРЬЕР


sc.textFile('file:///dummy.txt')
HadoopRDD[]
MapPartitionsRDD[]
map(line => line.split(" "))
MapPartitionsRDD[]
map(split => (split(0), split(1).toInt))
MapPartitionsRDD[]
groupByKey()
ShuffledRDD[]
mapValues(iter => iter.reduce(_ + _))
MapPartitionsRDD[]
collect()
collect()

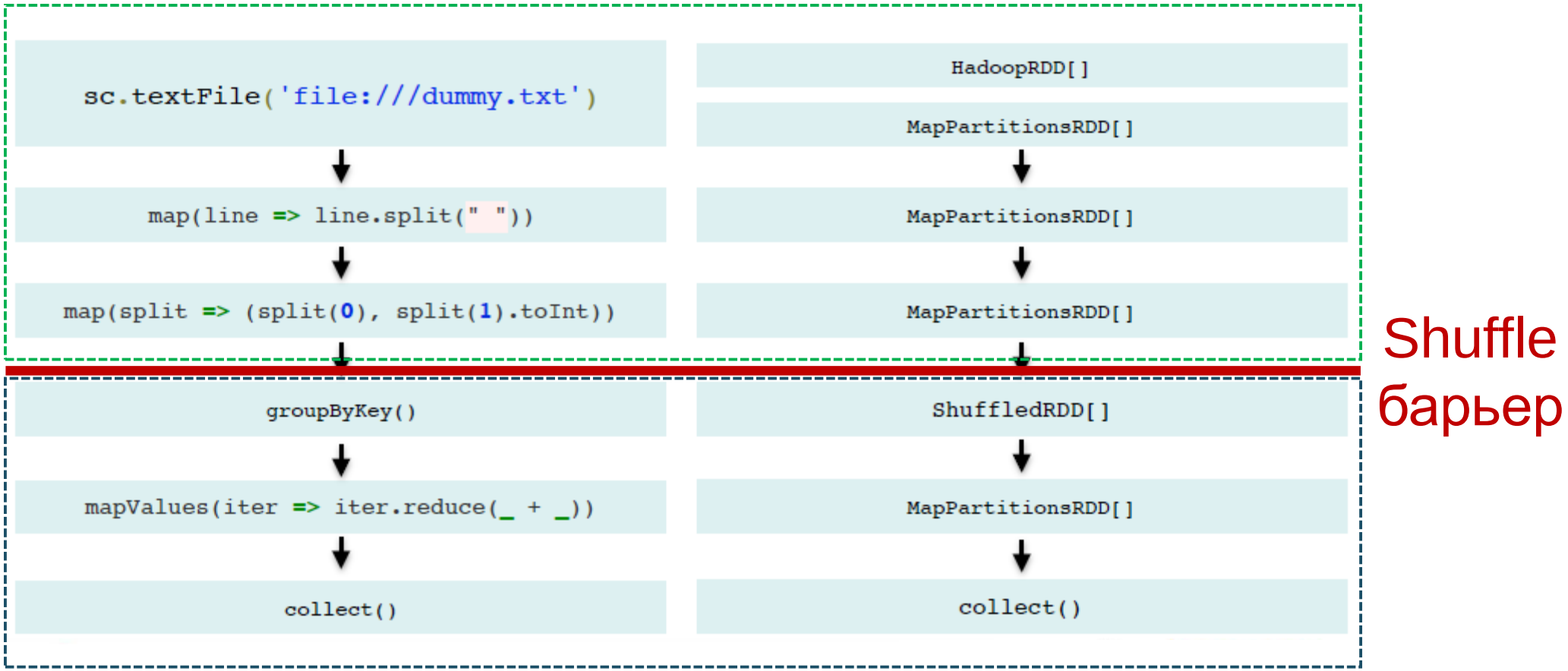
RDD | БАРЬЕР


sc.textFile('file:///dummy.txt')
HadoopRDD[]
MapPartitionsRDD[]
map(line => line.split(" "))
MapPartitionsRDD[]
map(split => (split(0), split(1).toInt))
MapPartitionsRDD[]
groupByKey()
ShuffledRDD[]
mapValues(iter => iter.reduce(_ + _))
MapPartitionsRDD[]
collect()
collect()

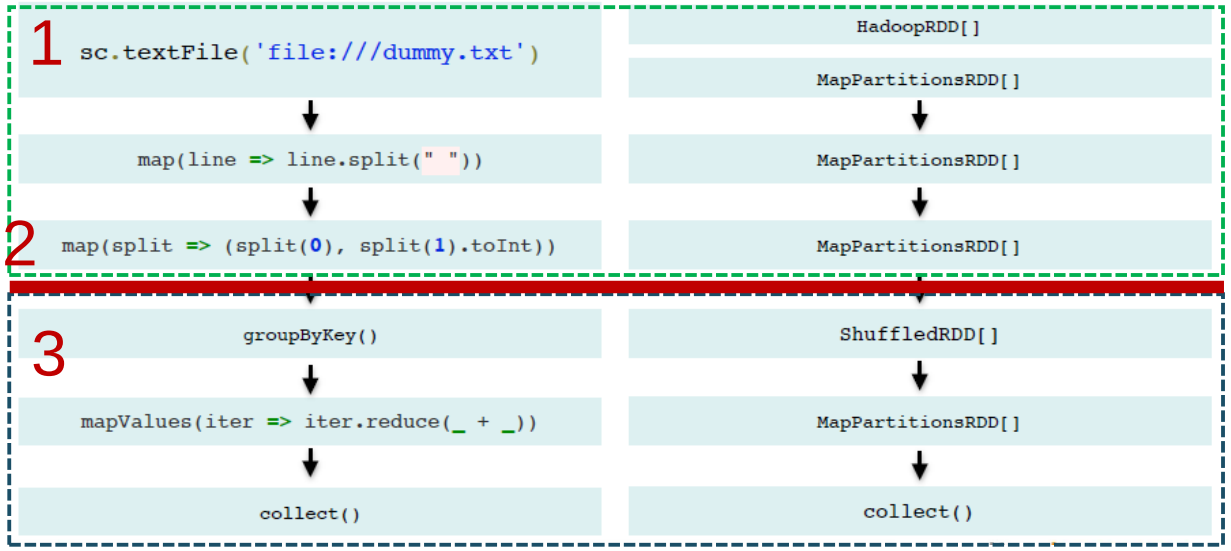
RDD | БАРЬЕР


sc.textFile('file:///dummy.txt')
HadoopRDD[]
MapPartitionsRDD[]
map(line => line.split(" "))
MapPartitionsRDD[]
map(split => (split(0), split(1).toInt))
MapPartitionsRDD[]
groupByKey()
ShuffledRDD[]
mapValues(iter => iter.reduce(_ + _))
MapPartitionsRDD[]
collect()
collect()
?

RDD | БАРЬЕР

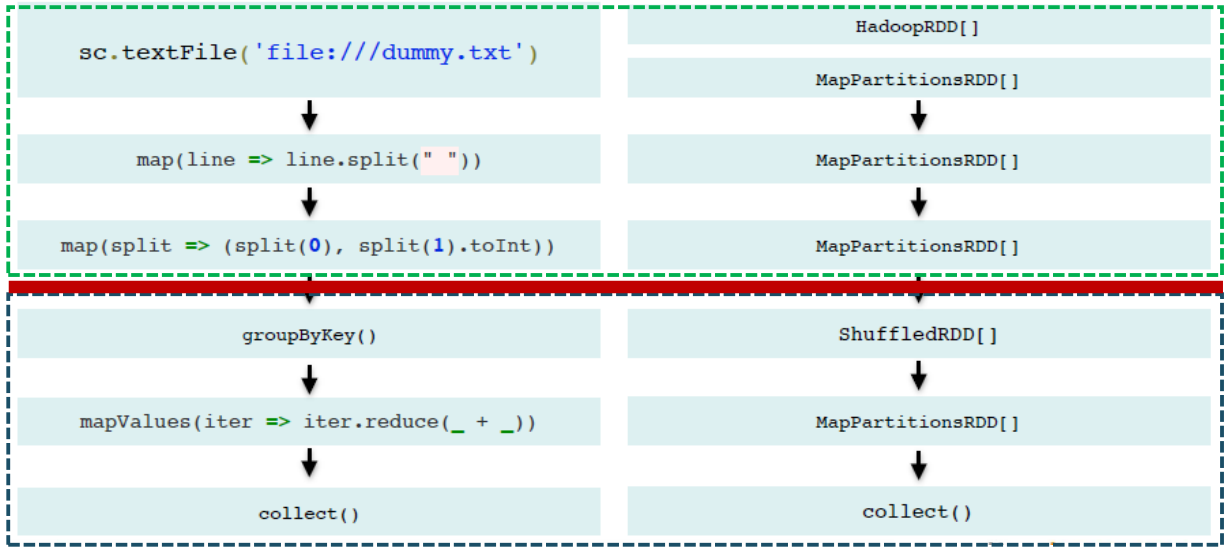

sc.textFile('file:///dummy.txt')
HadoopRDD[]
MapPartitionsRDD[]
map(line => line.split(" "))
MapPartitionsRDD[]
map(split => (split(0), split(1).toInt))
MapPartitionsRDD[]
groupByKey()
ShuffledRDD[]
mapValues(iter => iter.reduce(_ + _))
MapPartitionsRDD[]
collect()
collect()
Shuffle
барьер

RDD | ВЫПОЛНЕНИЕ



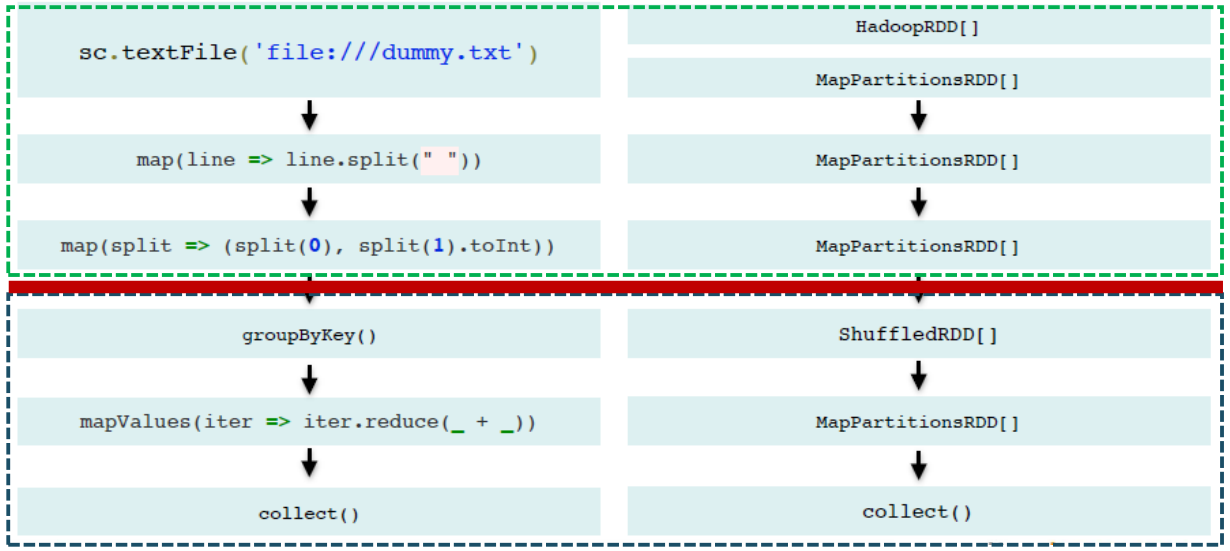
- Разбиение уровней на задачи для Executor’s

RDD | ВЫПОЛНЕНИЕ



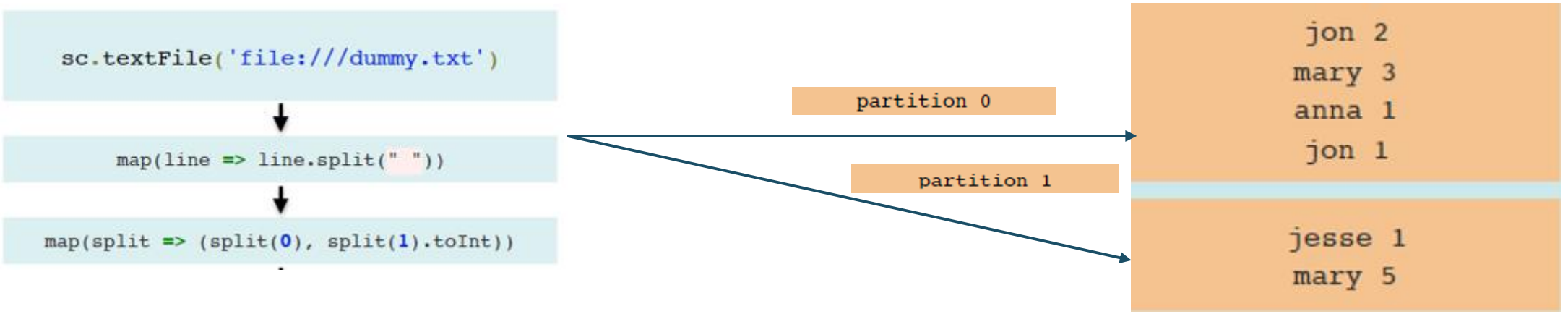
- Разбиение уровней на задачи для Executor’s
- Задача – это процесс партиционирования данных и вычисления

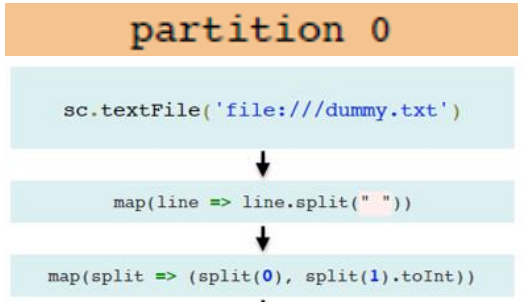
RDD | ВЫПОЛНЕНИЕ

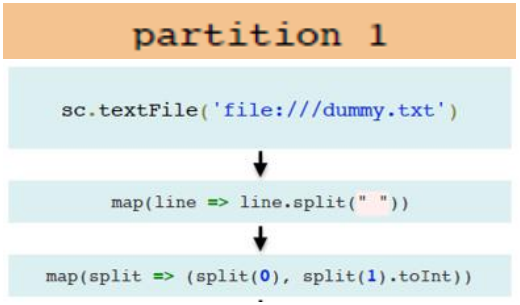


- Разбиение уровней на задачи для Executor’s
- Задача – это процесс партиционирования данных и вычисления
- Выполнение каждой задачи

RDD | ПАРТИЦИОНИРОВАНИЕ ЗАДАЧ







RDD | SHUFFLE

```
jon 2
mary 3
anna 1
jon 1
```

```
jesse 1
mary 5
```

```
groupByKey()
```



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

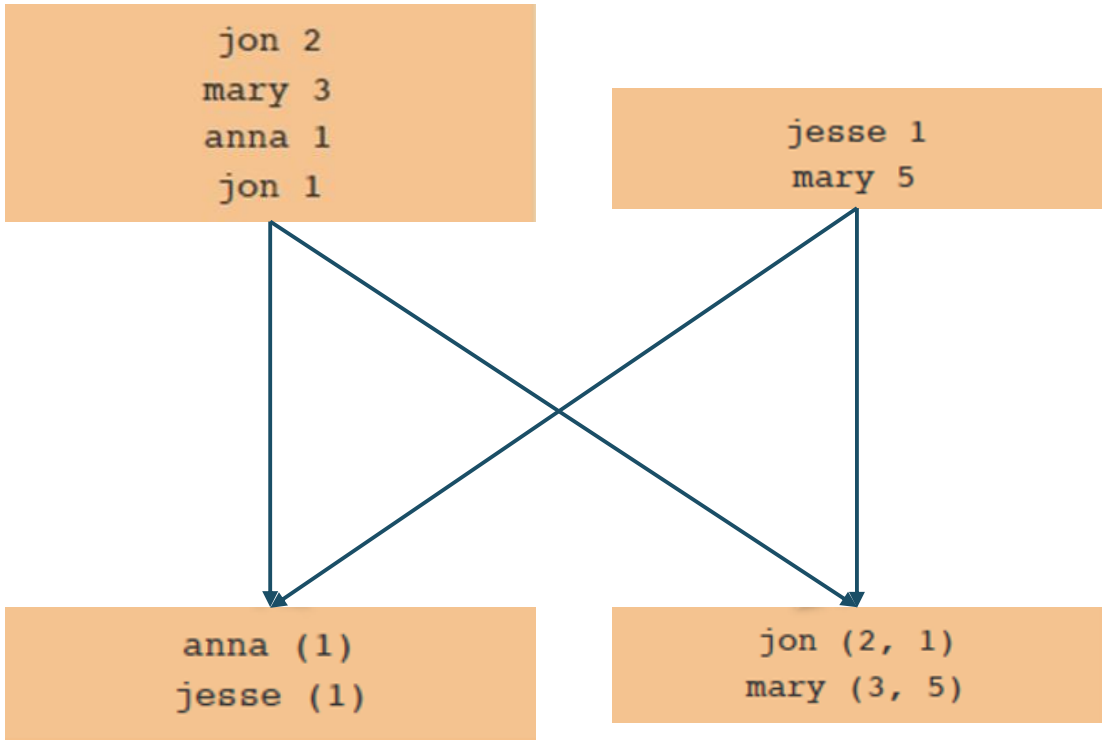
RDD | SHUFFLE

```
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 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

```
In [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()
                )
```

Generator

Создаем RDD

Transformations

Action

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()
```

DATAFRAMES

	name	age	state	num_children
0	john	23	iowa	2
1	mary	78	dc	2
2	peter	22	california	0
3	jeff	19	texas	1
4	bill	45	washington	2
5	lisa	33	dc	1

wild DATAFRAME appeared!

ПРИНЦИП РАБОТЫ С DF

Создать DataFrame из ресурсов

Применить трансформаторы к DataFrame
(select, filter, etc.)

Применить экшены к DataFrame
(show, saveAs…., etc.)

КАК С RDD?

НОВАЯ СТРУКТУРА!

TABLE	Состоит из строк и колонок
ROW	Из объекта rows – namedtuple - rdd

НОВАЯ МОДЕЛЬ РАЗРАБОТКИ

```
:  df = sqlContext.read.text("")

   onlyComments = df.filter("status == 'comments'")
```

«Ленивые» вычисления

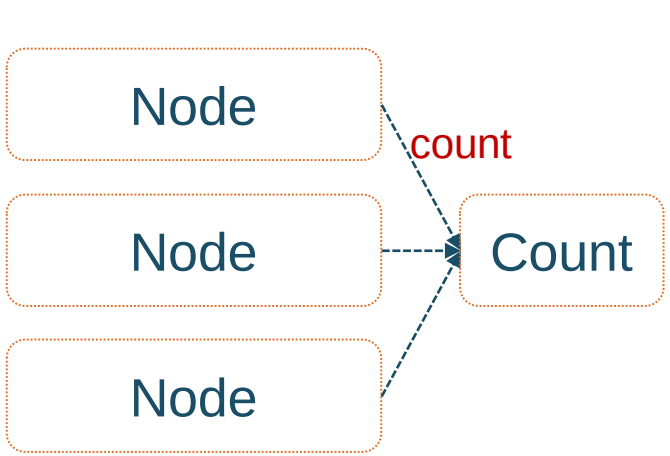
НОВАЯ МОДЕЛЬ РАЗРАБОТКИ

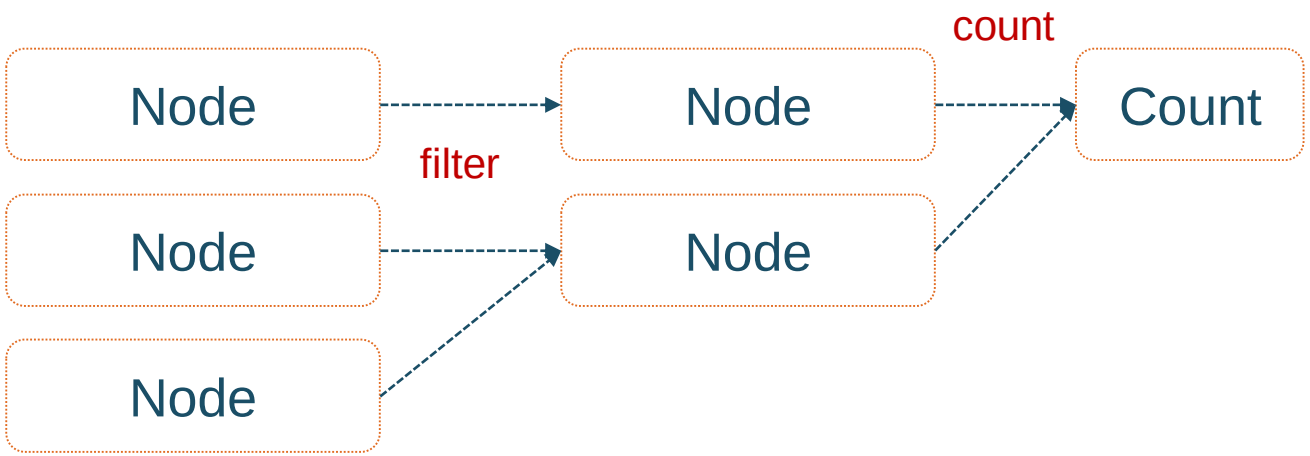
```
:  df = sqlContext.read.text("")

   onlyComments = df.filter("status == 'comments'")

   onlyComments.count() / df.count()
```

?

НОВАЯ МОДЕЛЬ РАЗРАБОТКИ

```
:  df = sqlContext.read.text("")

   onlyComments = df.filter("status == 'comments'")

   onlyComments.count() / df.count()
```

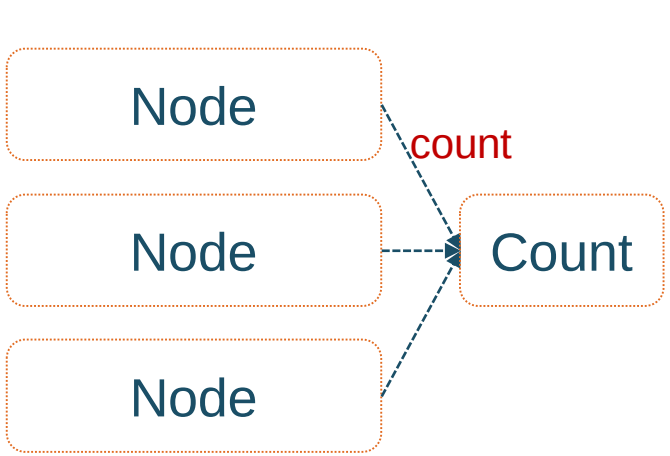


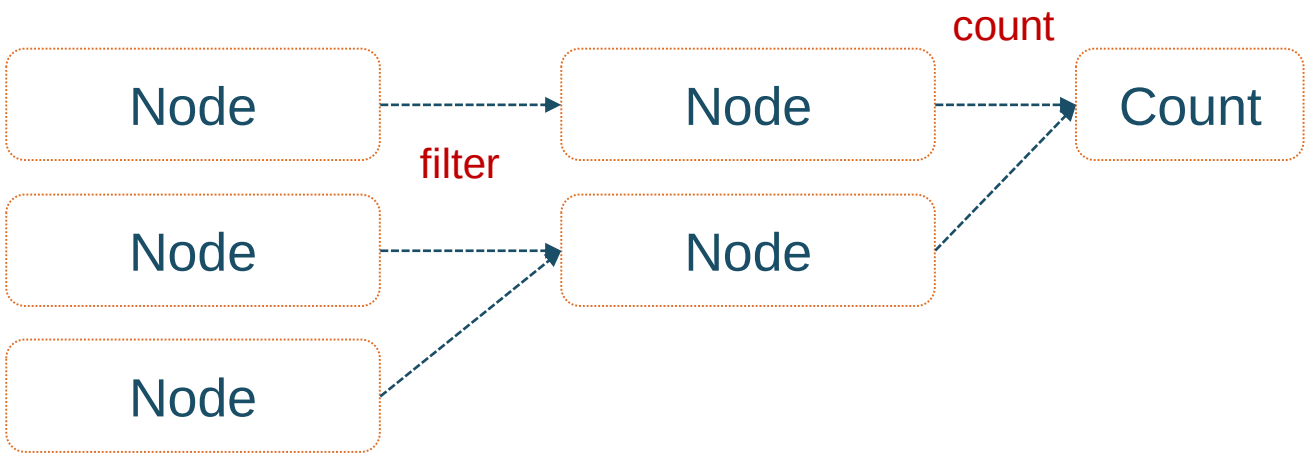


НОВАЯ МОДЕЛЬ РАЗРАБОТКИ

```
df = sqlContext.read.text("")

onlyComments = df.filter("status == 'comments'")

onlyComments.count() / df.count()
```

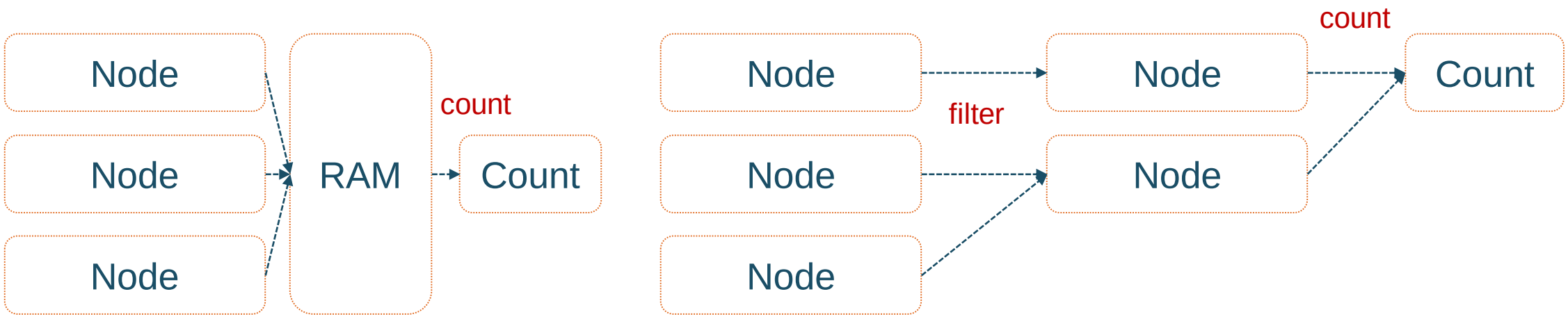
1. Чтение данных (2 раза)
2. Подсчет результата по патрициям (2 раза)
3. Фильтр
4. Соединение результата на драйвере





НОВАЯ МОДЕЛЬ РАЗРАБОТКИ

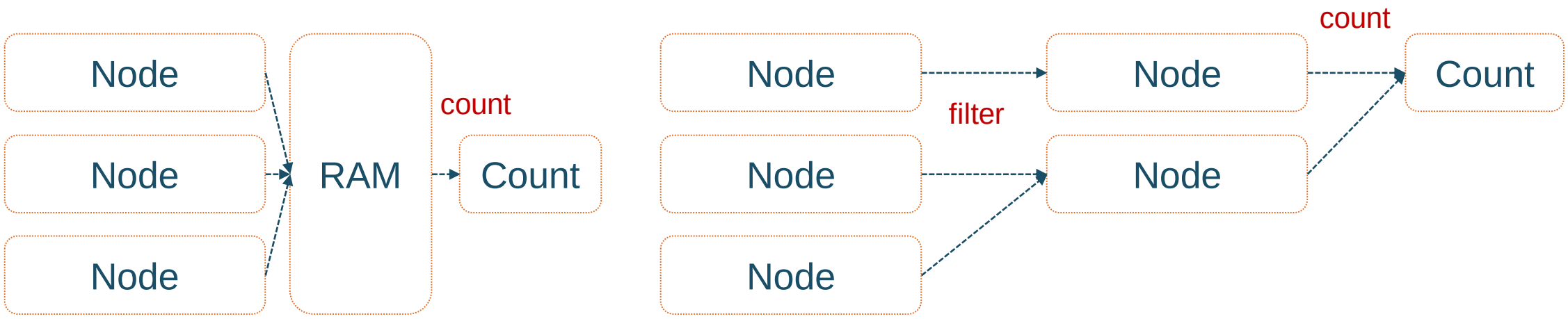
```
:  df = sqlContext.read.text("")
   df.cache()    # cached!
   onlyComments = df.filter("status == 'comments'")
   onlyComments.count() / df.count()
```



НОВАЯ МОДЕЛЬ РАЗРАБОТКИ

```
df = sqlContext.read.text("")
df.cache()    # cached!
onlyComments = df.filter("status == 'comments'")
onlyComments.count() / df.count()
```

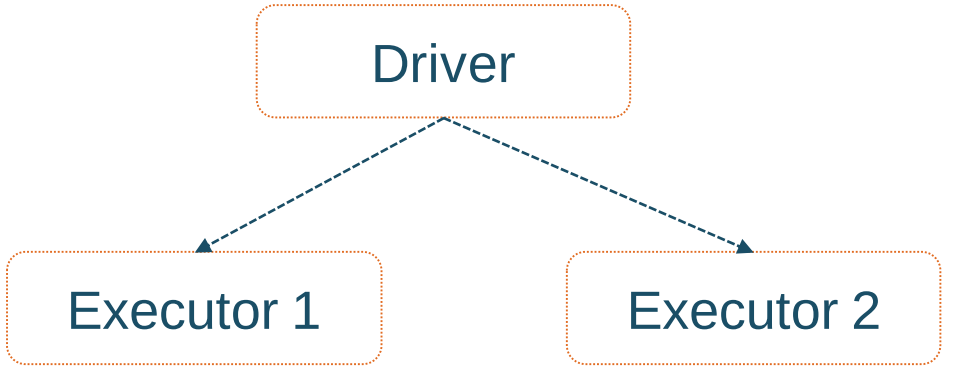
1. Чтение данных
2. Подсчет результата по патрициям (2 раза)
3. Фильтр
4. Соединение результата на драйвере



НОВАЯ МОДЕЛЬ РАЗРАБОТКИ

- Python код работает на driver
- Transformations на executor’ах
- Actions на executor’ах и driver

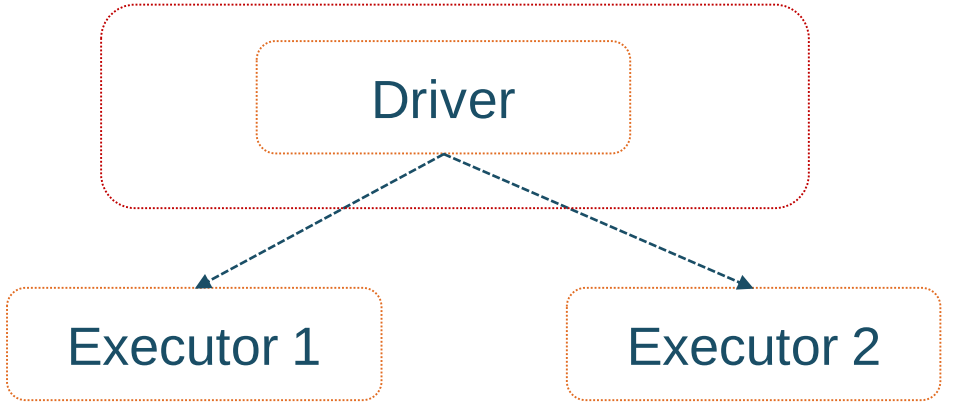
```
:  df = sqlContext.read.text("")

   onlyComments = df.filter("status == 'comments'")

   onlyComments.count() / df.count()
```



НОВАЯ МОДЕЛЬ РАЗРАБОТКИ

- Python код работает на driver
- Transformations на executor’ах
- Actions на executor’ах и driver

```
df = sqlContext.read.text("")

onlyComments = df.filter("status == 'comments'")

onlyComments.count() / df.count()
```

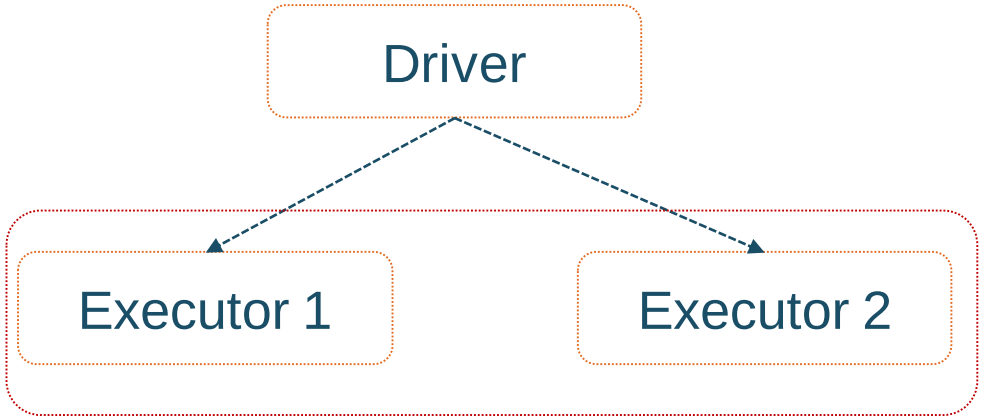


НОВАЯ МОДЕЛЬ РАЗРАБОТКИ

- Python код работает на driver
- Transformations на executor’ах

  (а ещё, во время чтения)
- Actions на executor’ах и driver

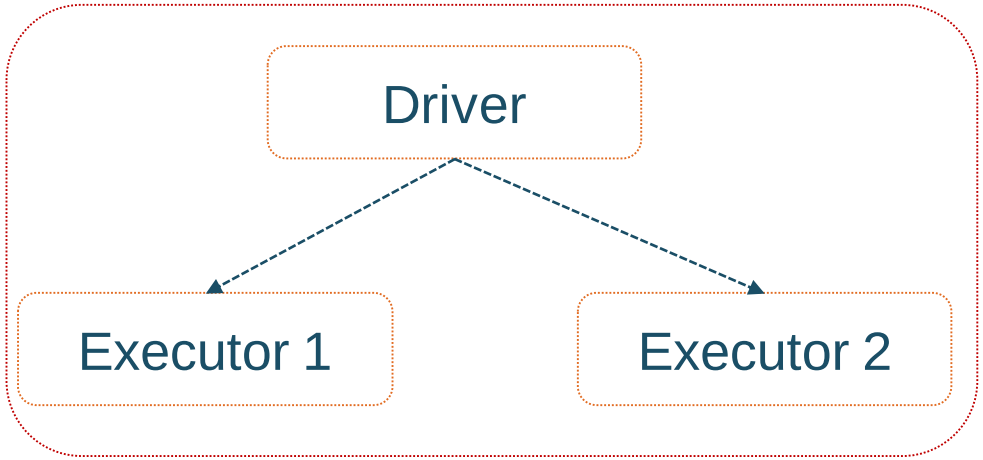
```
:  df = sqlContext.read.text("")

   onlyComments = df.filter("status == 'comments'")

   onlyComments.count() / df.count()
```



НОВАЯ МОДЕЛЬ РАЗРАБОТКИ

- Python код работает на driver
- Transformations на executor’ах
- Actions на executor’ах и driver

```
:  df = sqlContext.read.text("")

   onlyComments = df.filter("status == 'comments'")

   onlyComments.count() / df.count()
```



PERFORMANCE

МОНИТОРИМ И ОПТИМИЗИРУЕМ

Данные: применяем сериализацию и кэширование

Следим за структурами данных, кэшем, количеством шаффлов

Запомним: parallelism + memory + GC

МОНИТОРИМ И ОПТИМИЗИРУЕМ

Данные: применяем сериализацию и кэширование

Следим за структурами данных, кэшем, количеством шаффлов

Запомним: parallelism + memory + GC

МОНИТОРИМ И ОПТИМИЗИРУЕМ

.coalesce() – когда очень много партиций

Много задач заканчиваются быстро, но есть несколько медленных

Много задач в очереди

<100мс на задачу

МОНИТОРИМ И ОПТИМИЗИРУЕМ

.repartition() – когда очень мало партиций

Не эффекта от параллелизма

Данные очень смещены
(смотрим за skew в данных)

МОНИТОРИМ И ОПТИМИЗИРУЕМ

МОНИТОРИМ И ОПТИМИЗИРУЕМ

Storage - Cache

Environment - Configuration

Executors – Workers

МОНИТОРИМ И ОПТИМИЗИРУЕМ

Всегда старайтесь уменьшить данные

- `aggregateByKey()`
- `filter()`

МОНИТОРИМ И ОПТИМИЗИРУЕМ

Остерегайтесь shuffle
Сделайте заранее партиционирование и .persist()

`partitionBy()` ⇢

- `join()`
- `reduceByKey()`
- `sortByKey()`

МОНИТОРИМ И ОПТИМИЗИРУЕМ

Ищите лучшие варианты

`groupByKey().mapValues()` ----------> `reduceByKey()`

УРОВНИ КЭША

MEMORY_ONLY	В ОЗУ, как RDD объект в JVM
MEMORY_AND_DISK	В ОЗУ, как RDD объект в JVM (но на диск помещается то, что не влезло в ОЗУ)
DISK_ONLY	RDD партции хранятся на диске, без участия ОЗУ
*_SER / *_N	Повторяет уровень хранения, но объект становится сериализованный (в байт коде) (более эффективно по памяти, но появляется зависимость процессора / Повторяет уровень хранения, повторяет (реплецирует) на N количестве нод

ЗАПОМНИМ

НЕЛЬЗЯ ИЗМЕРИТЬ – НЕЛЬЗЯ УСКОРИТЬ

ДЕЛАЙТЕ РЕВЬЮ СВОЕГО КОДА (ИЗБЕГАЙТЕ ЦИКЛОВ)

НАБОРЫ ДАННЫХ МОГУТ БЫТЬ ОПТИМИЗИРОВАННЫ ПАРТИЦИОНИРОВАНИЕМ

САМАЯ ДОРОГАЯ ОПЕРАЦИЯ SHUFFLE

SPARK ПРОЕКТ

SPARK ПРОЕКТ

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

  Задача: Выбрать данные

  (https://cseweb.ucsd.edu/~jmcauley/datasets.html)