

# СИСТЕМЫ ОБРАБОТКИ БОЛЬШИХ ДАННЫХ

## Разработка приложений



К.Т.Н.  
Папулин Сергей Юрьевич  
*[papulin\\_bmstu@mail.ru](mailto:papulin_bmstu@mail.ru)*

# Лекция 5. Apache Spark



# Основные темы

- Особенности Spark
- Архитектура Spark
- Развертывание Spark на YARN
- Запуск приложения Spark
- Dataframe, Dataset, SQL API
- Операции над RDD
- Обзор некоторых трансформаций и действий
- Кэширование RDD
- Переменные Accumulator и Broadcast

# Apache Spark

**Spark** расширяет возможности MapReduce за счет поддержки:

- нескольких типов вычислений (batch-обработки, интерактивные запросы, потоковой и графовой обработки)
- работы с оперативной памятью для хранения промежуточных результатов
- повторного использования данных в оперативной памяти
- итеративных вычислений (например, для алгоритмов машинного обучения)
- DAG операций

# Стек Spark



Streaming

GraphX

MLlib

Spark SQL

Spark

YARN

Mesos

Standalone



➤	RDD		
➤	Partition		➤ Driver
➤	Job	➤ Transformation	➤ Executor
➤	Task	➤ Action	➤ SparkContext/Session
➤	Stage		

**Resilient Distributed Dataset (RDD)** – **надежная, распределенная, неизменяемая коллекция** записей

**RDD** – основная абстракция в Spark, представляет неизменяемую разделенную коллекцию элементов, с которыми можно работать параллельно.

Единицей параллельности является **часть (partition)**

Количеством **частей** можно управлять посредством специальных операций

Каждая **часть (partition)** состоит из **записей (records)**

**RDD** хранит историю операций над RDD в виде **графа операций (RDD lineage graph)**.  
С его помощью можно восстановить **partition**



# Особенности RDD

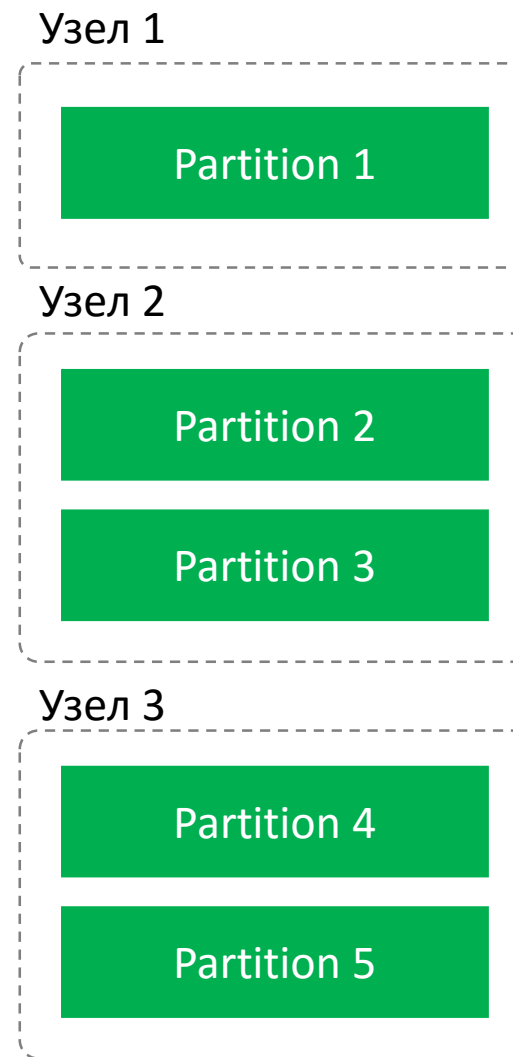
- Неизменяемо (Immutable)
- В оперативной памяти (In-Memory)
- Выполнение по необходимости (Lazy evaluated)
- Можно повторно использовать при сохранении в оперативной памяти или на диске (Cacheable)
- Разделено на части (Partitions)
- Части обрабатываются параллельно на различных узлах (Parallel)
- Типизировано (Typed)

# Часть RDD (Partition)

Логическое представление



Физическое расположение



- Список родительских RDD
- Список частей (partitions)
- Функция вычисления
- Делитель (partitioner) (опционально)
- Предпочтительное расположение частей в кластере (опционально)



## **Трансформации** (transformation)

Возвращает новую RDD

Примеры, `map()`, `filter()` и др.



## **Действия** (action)

Возвращает итоговый результат на Driver или записывает в постоянную память (например, HDFS)

Примеры, `count()`, `collect()` и др.



## **Выполнение по требованию** (Lazy evaluation)

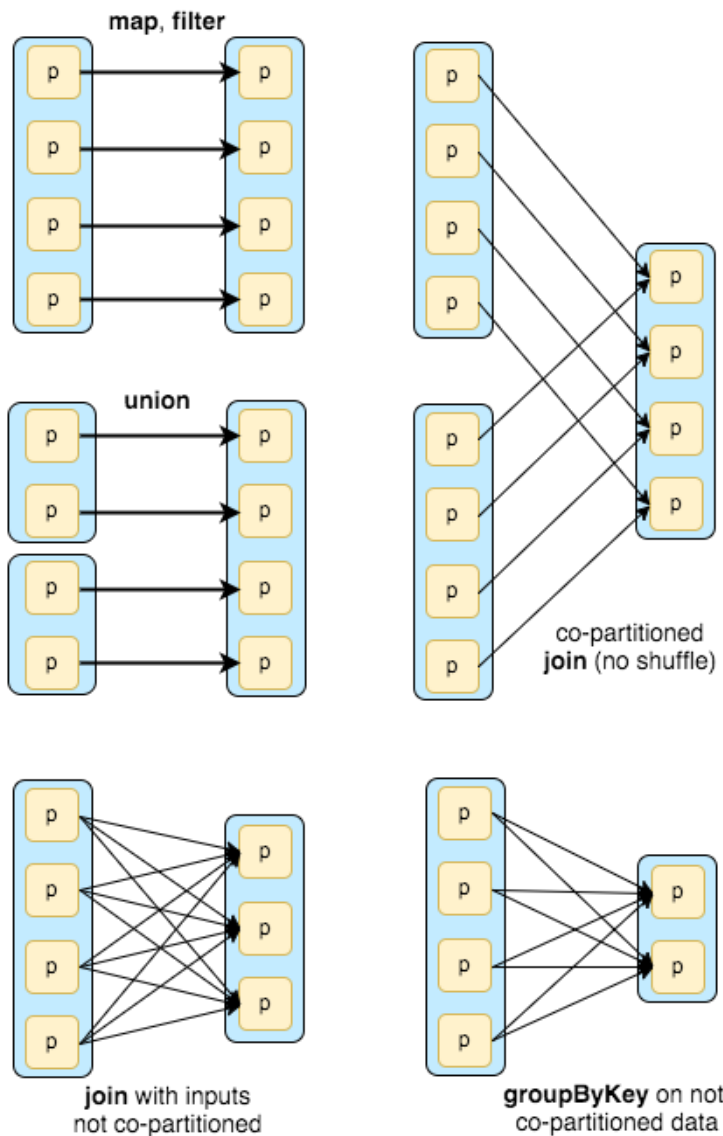
Непосредственно выполнение операций начинается только после действия, даже если до него было несколько трансформаций

# Состав работы (Job)

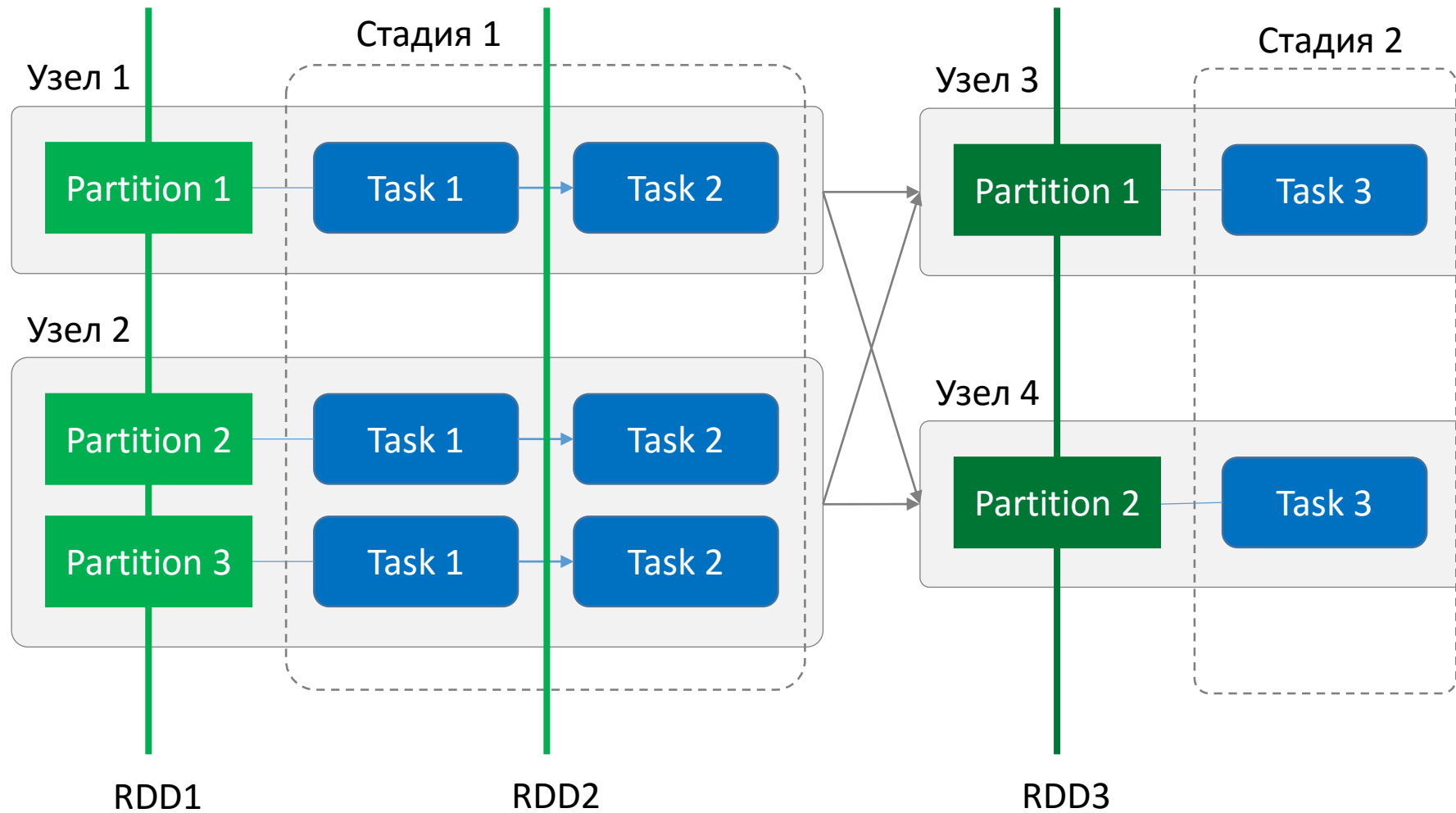
- **Работа (job)** – единица вычисления топ уровня  
  
Запускается после того, как в программе встретилась операция-действие (action)
- **Работа (job)** разбивается на **стадии (stages)**
- **Стадии (stages)** состоят из **задач (tasks)**. Задача – минимальная единица выполнения



# Зависимости между RDD



# Взаимосвязь между частями, задачами и стадиями



# Архитектура Spark



- Spark построен на master/slave архитектуре с одним **центральный координатором** и множеством распределенных **worker'ов**
- Центральный координатор – **driver**
- Driver взаимодействует с множеством распределенных worker'ов – **executors**
- **Driver и executor'ы** запускаются в отдельных JVM
- Spark приложение – **driver + executors**
- **Spark context** – по существу клиент среды выполнения Spark, выполняет роль мастера Spark приложения. Запускается на **driverе**
- Spark приложение запускается на множестве машин с использованием внешнего сервиса – **cluster manager**

**Driver** – процесс, запускающий `main()` метод разработанной программы (создает `SparkContext`, `RDD`, `DAG` выполнения операций)

**Driver** выполняет две основные функции:

- Конвертирует программу в задания (**taskи**)
- Планирует и запускает выполнение заданий на **executorax**

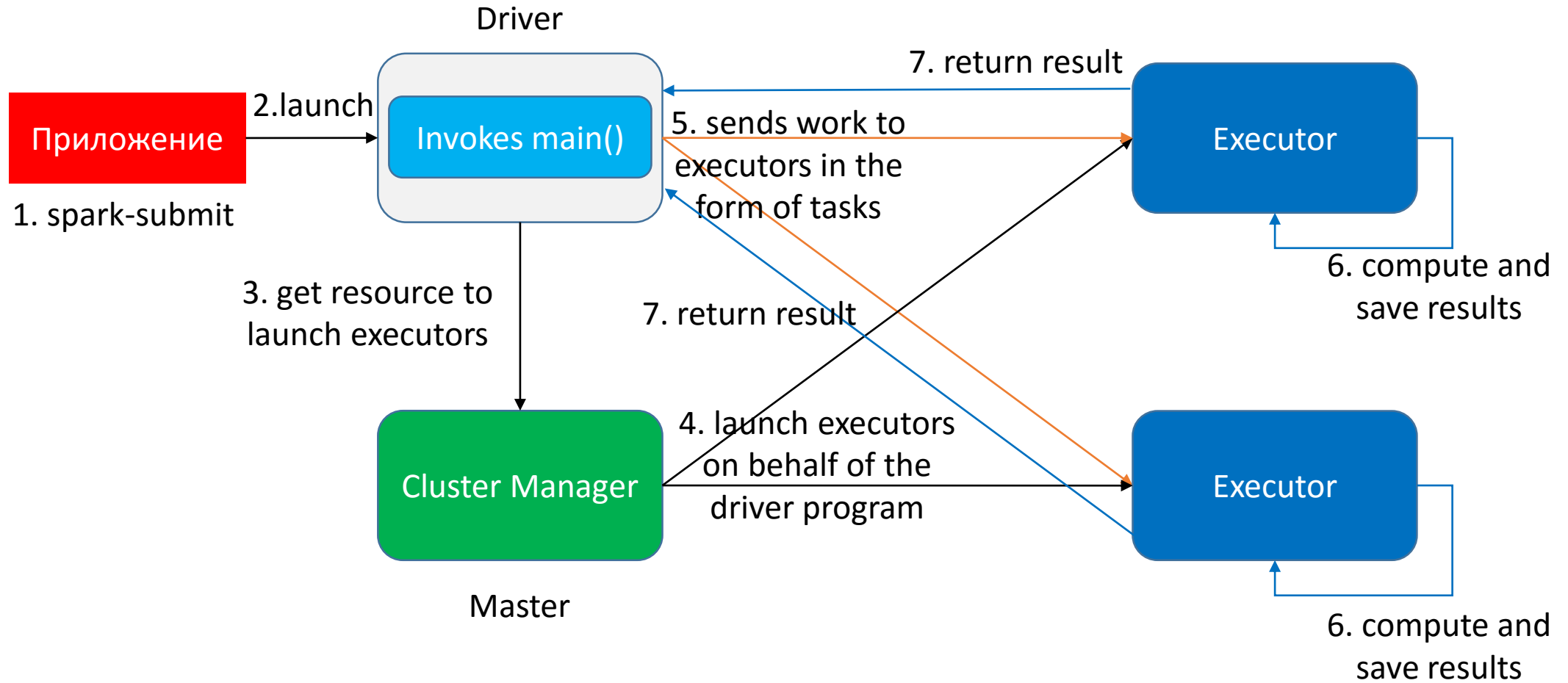
Spark **executor** – процесс, который отвечает за запуск и выполнение отдельных задач (**task**) некоторой Spark работы (**job**)

**Executor** запускается при разворачивании Spark приложения и остается в рабочем состоянии до завершения приложения

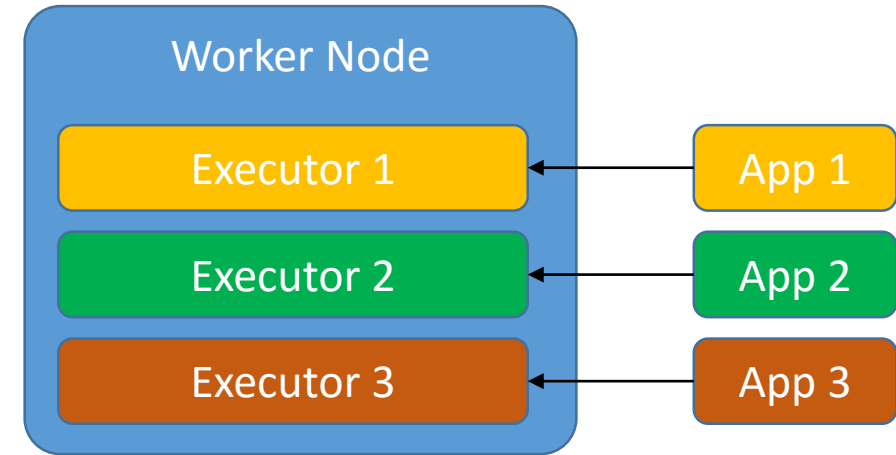
**Executor** выполняет две основные функции:

- Запускает задачи (**task**) приложения и возвращает результат на driver
- Обеспечивает хранение закэшированных **RDD** (`cache()`, `persist()`)

# Развертывание Spark приложения



- Каждое Spark **приложение** имеет свои **executors**
- Задачи (**tasks**) от разных **приложений** запускаются на разных JVM
- Несколько задач (**task**) одного приложения могут запускаться на одном **executore**
- Каждое **приложение** имеет не более одного **executora** на каждом рабочем узле (worker) (standalone)
- **Задача** соответствует отдельному **потоку**
- Каждая **partition** является входом для отдельной **задачи** (task)

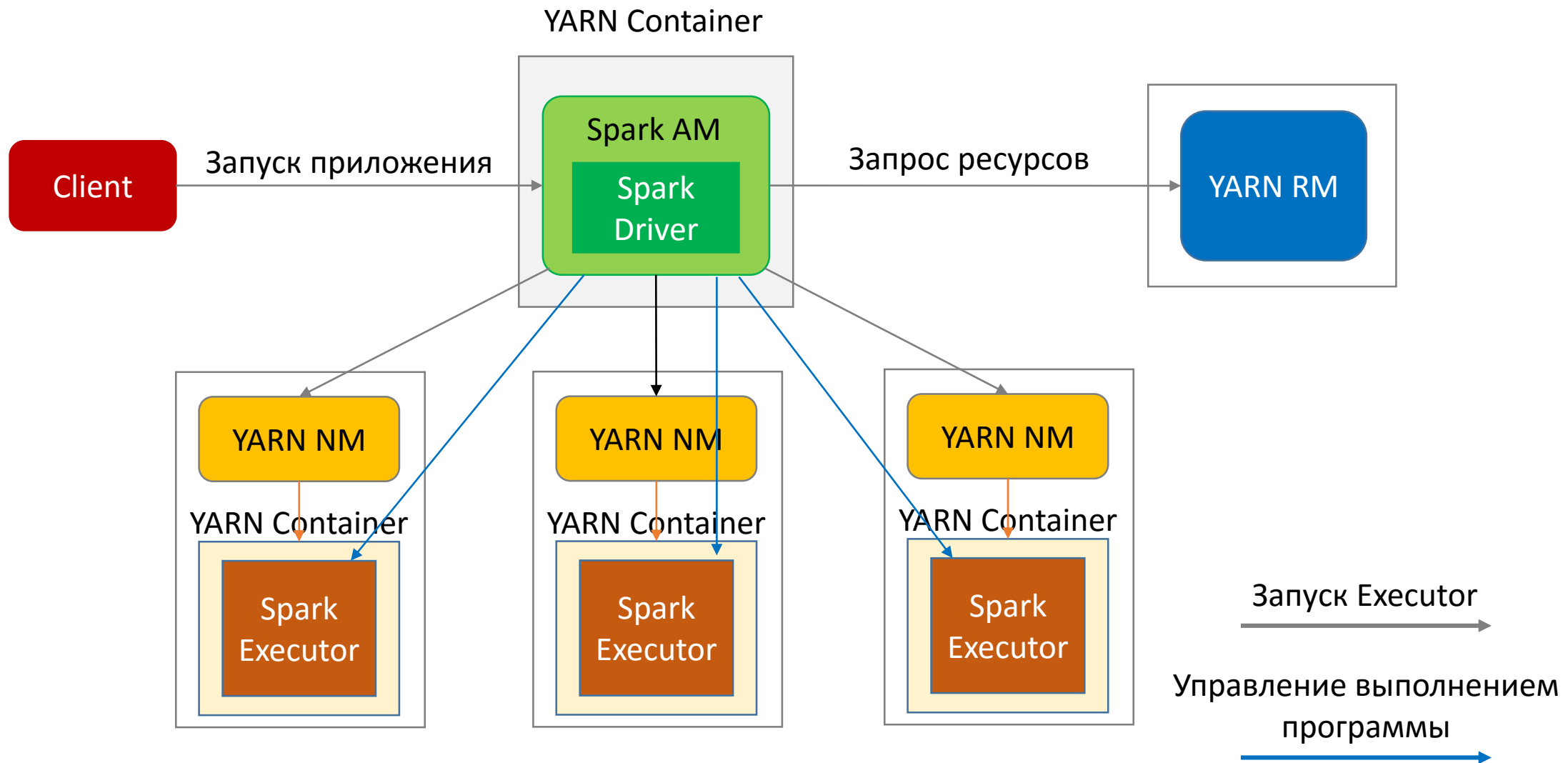


# Сопоставление Driver/Executor/Job/Task/RDD

- Одно Spark приложение может иметь несколько работ (job) (для каждой операции-действия). Работы могут выполняться параллельно, если запущены разными потоками
- По умолчанию количество **partition**ов соответствует количеству всех доступных ядер (**core**)
- Количество **partition**ов  $\geq$  количеству **executor**ов
- Один executor – 1 и более CPU.
- Рекомендация: 2-3 **задачи** на одно CPU ядро
- Количество **partition**ов определяет, как много файлов сгенерируется операцией-действием при сохранении **RDD** на диск

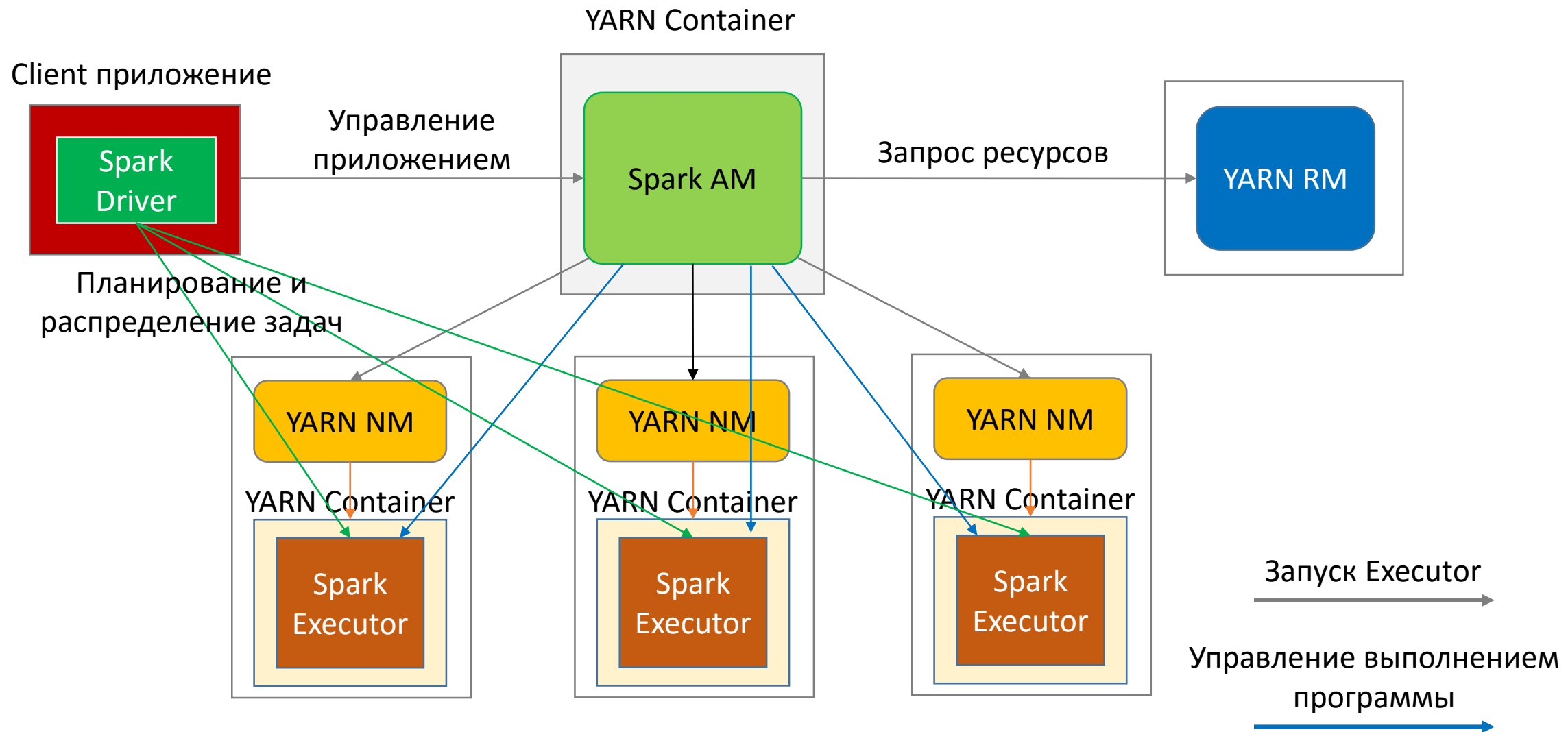
# Развертывание Spark на YARN

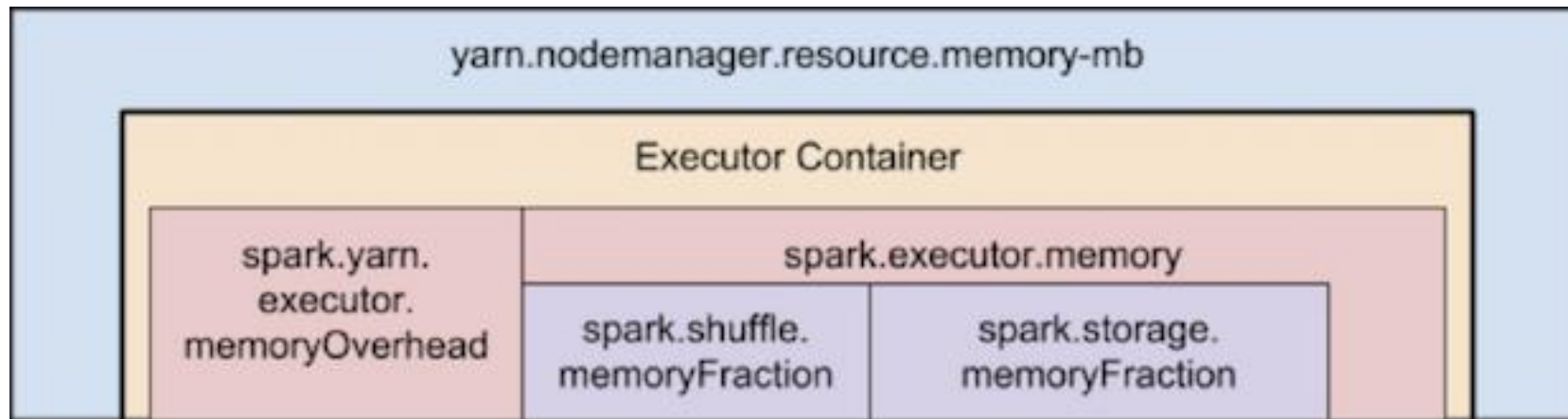
# Развертывание Spark приложений на YARN. Режим кластера





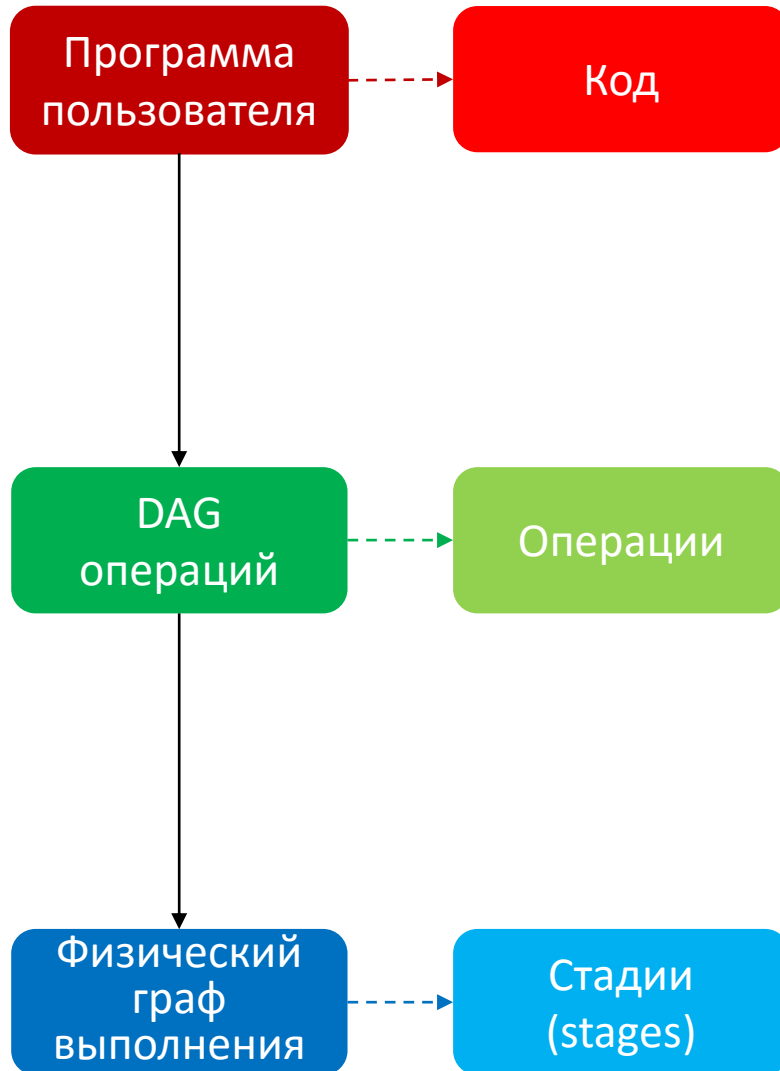
# Развертывание Spark приложений на YARN. Режим клиента





# Запуск приложения Spark

# Запуск приложения на Driver



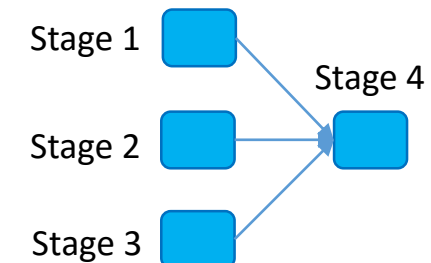
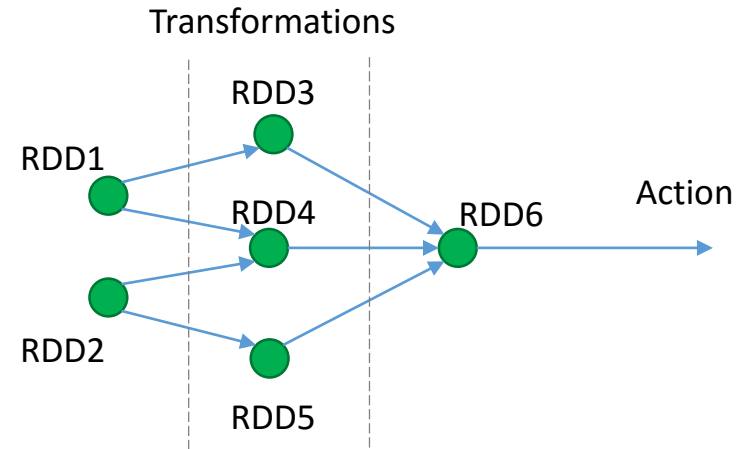
```

rdd_01 = sc.parallelize([(1, 2), (1, 3), (1, 2), (4, 4)], 2)
rdd_02 = sc.parallelize([(2, 2), (1, 1), (3, 3), (2, 2)], 2)

rdd_11 = rdd_01.groupByKey()
rdd_12 = rdd_01.join(rdd_02)
rdd_13 = rdd_02.groupByKey()

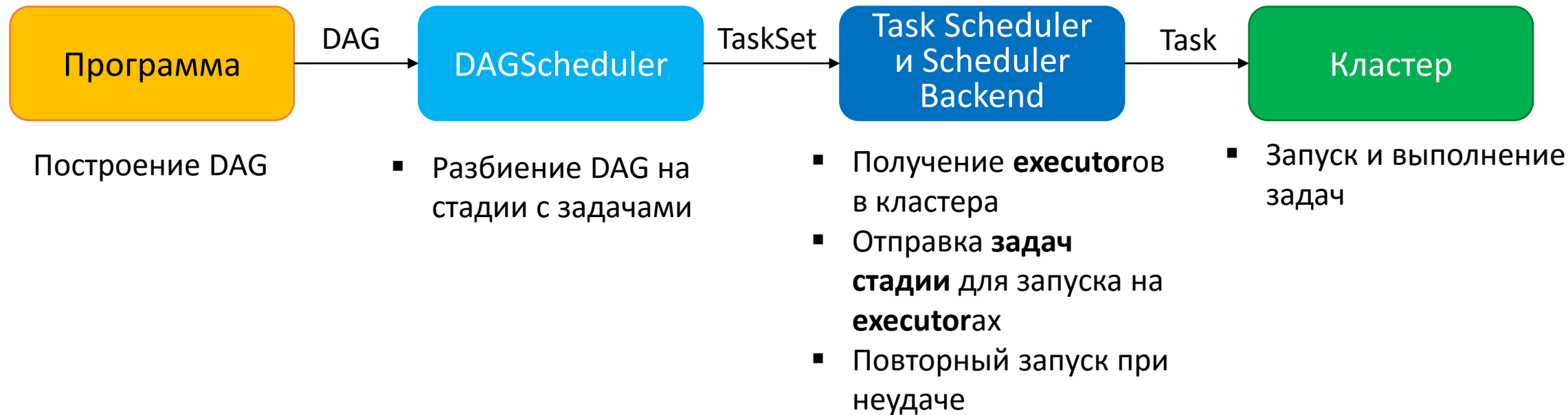
rdd_21 = sc.union([rdd_11, rdd_12, rdd_13])

rdd_21.collect()
  
```



# Запуск приложения Spark

**SparkContext** позволяет получить Spark-приложению доступ к кластеру



Чтобы создать **SparkContext** необходимо сначала создать **SparkConf**, который хранит конфигурации кластера

В Spark 2.x – **SparkSession**

# Пример кода программы

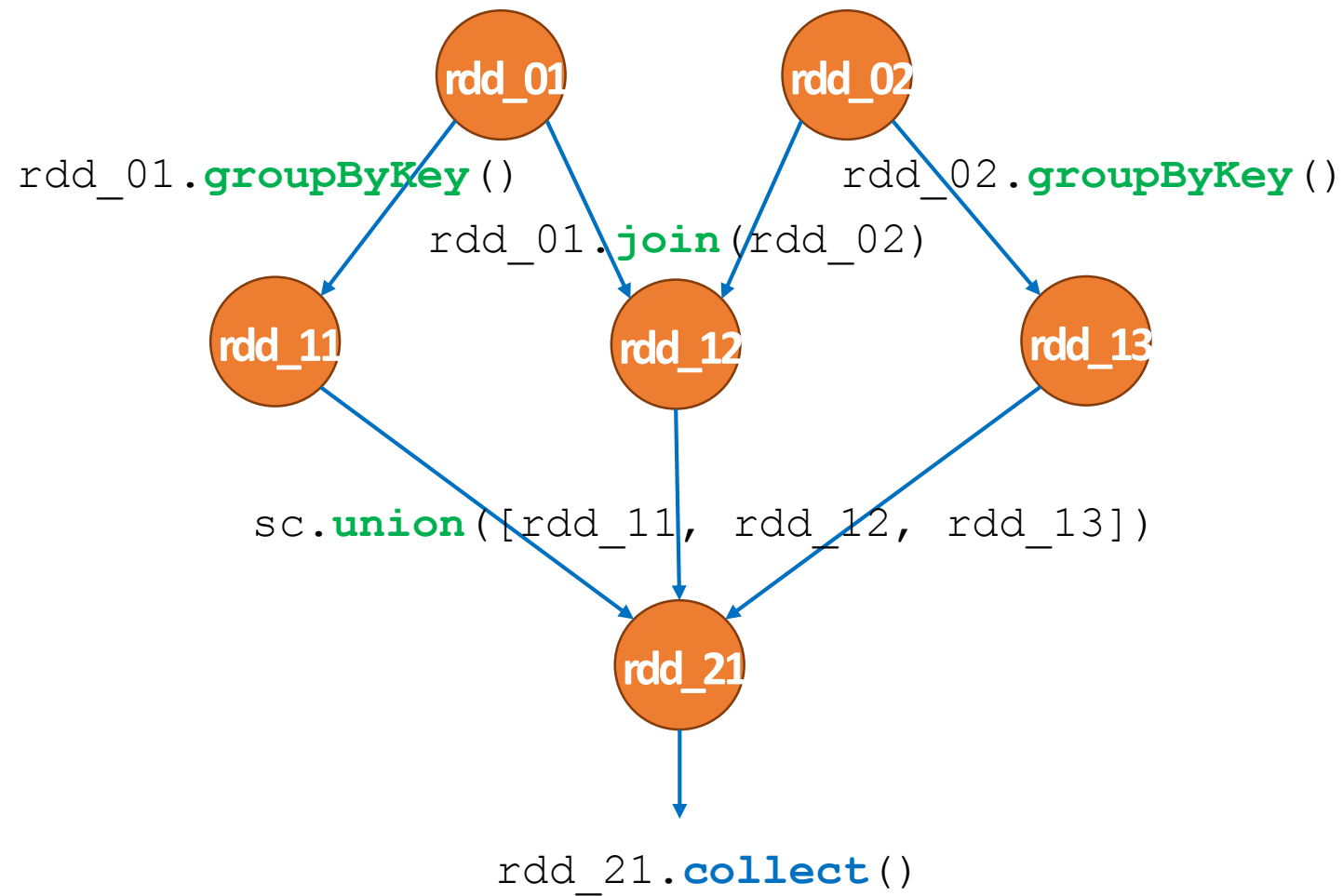
```
rdd_01 = sc.parallelize([(1, 2), (1, 3), (1, 2), (4, 4)], 2)
rdd_02 = sc.parallelize([(2, 2), (1, 1), (3, 3), (2, 2)], 2)

rdd_11 = rdd_01.groupByKey()
rdd_12 = rdd_01.join(rdd_02)
rdd_13 = rdd_02.groupByKey()

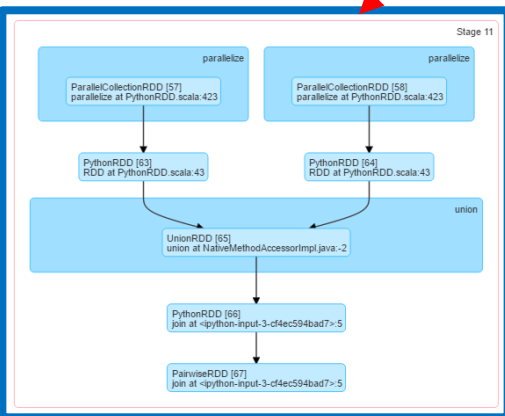
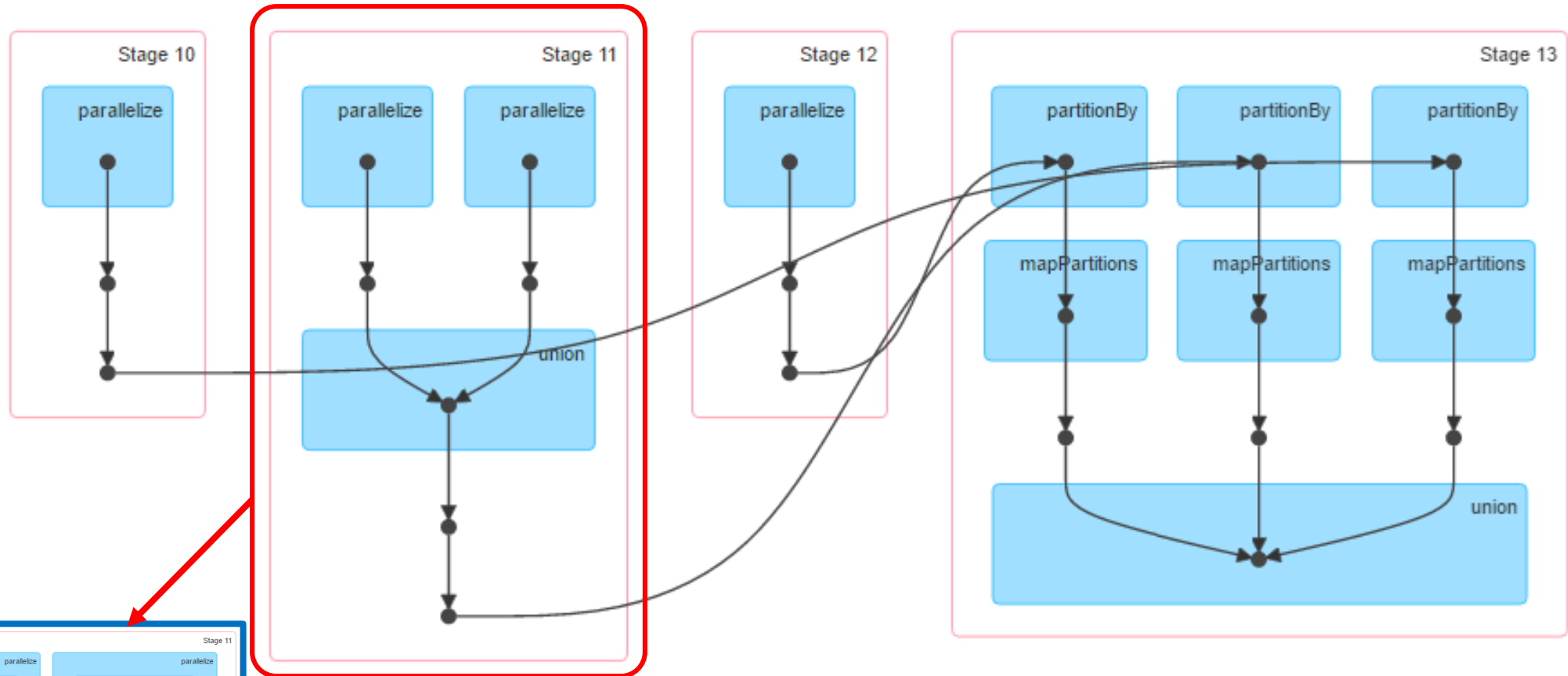
rdd_21 = sc.union([rdd_11, rdd_12, rdd_13])

rdd_21.collect()
```

# Пример DAG операций



# Пример плана выполнения – стадии





# Dataframe, Dataset, SQL API

Spark SQL – Spark модуль для обработки структурированных данных

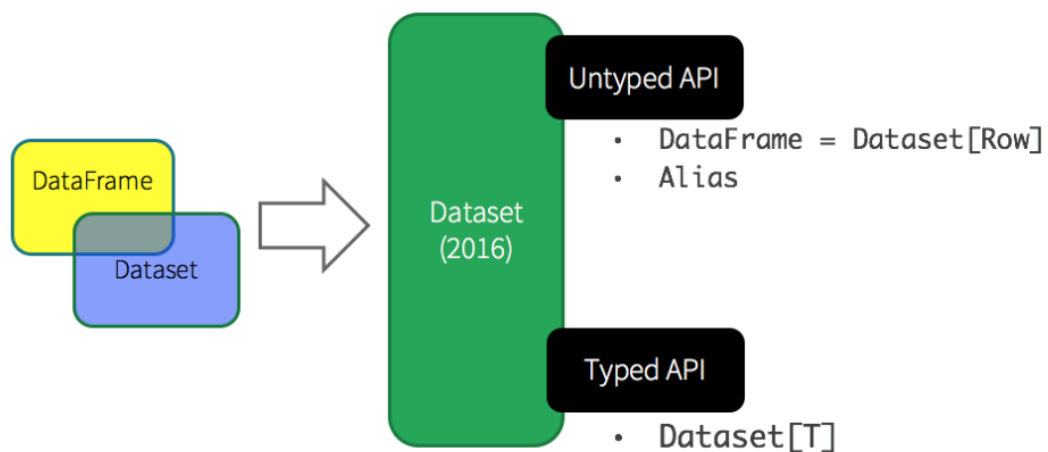
В отличие от RDD API, интерфейс предоставляет Spark больше информации о структуре данных и вычислениях, которые должны быть произведены.

**Dataset** – распределенная коллекция данных (для Scala и Java). Обеспечивает строгое типизирование записей RDD и оптимизацию выполнения на базе Spark SQL

**Dataframe** – **dataset** с именованными столбцами (аналог таблиц в реляционных БД, или dataframeam в R/Python со своими средствами оптимизации)

**Dataframe** API доступна на Scala, Java, Python и R

## Unified Apache Spark 2.0 API

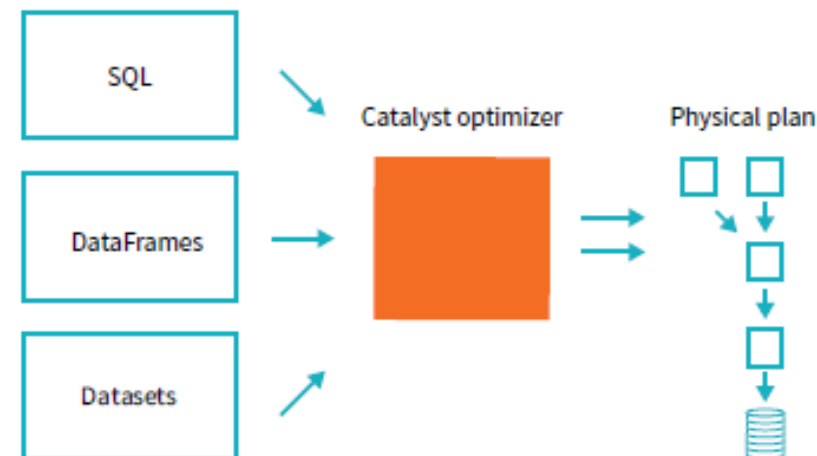


 databricks

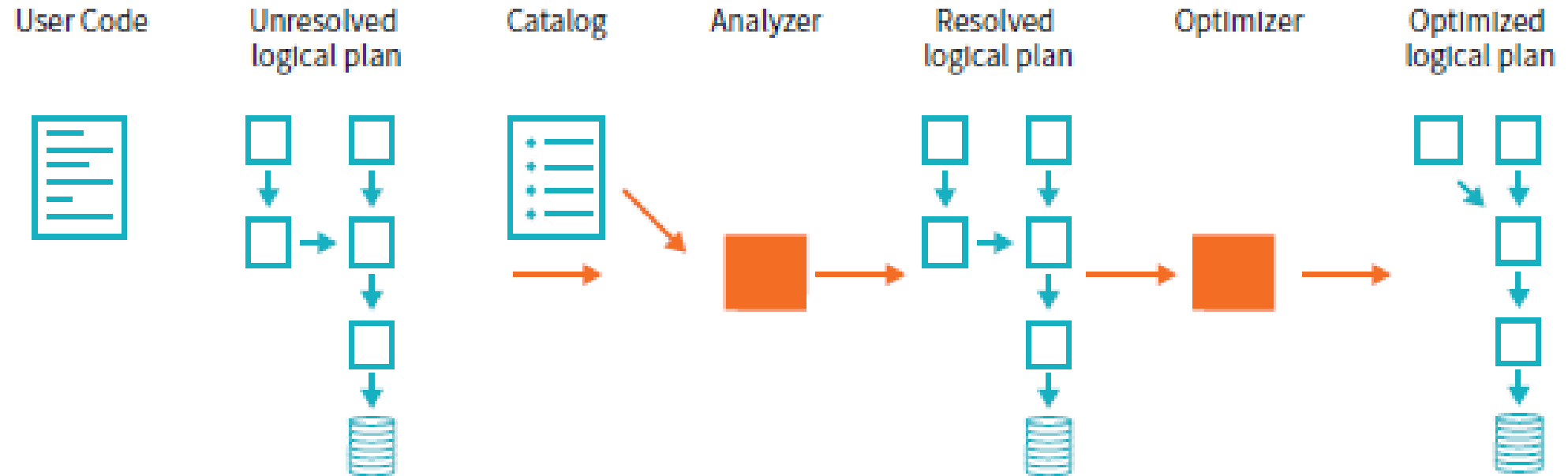
Язык программирования	Основная абстракция
Scala	Dataset[T] & DataFrame (alias for Dataset[Row])
Java	Dataset[T]
Python	DataFrame
R	DataFrame

# Этапы выполнения кода Dataframe/Dataset/SQL API

- 1. Проверка валидности кода
- 2. Преобразование кода в логический план
- 3. Трансформация логического плана в физический план
- 4. Выполнение физического плана на кластере



# Формирование логического плана



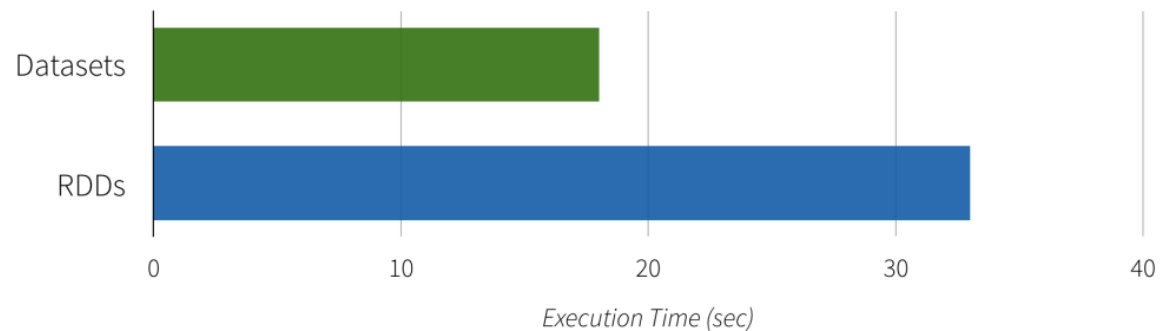
# Формирование физического плана выполнения



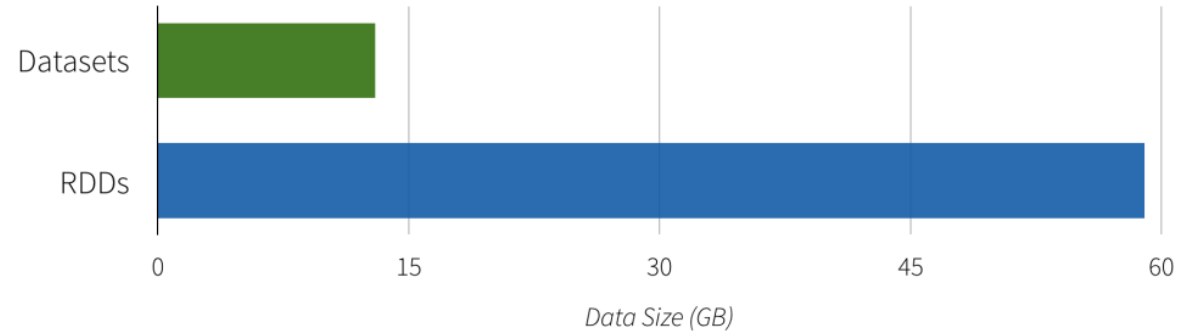
- Физический план представляет из себя набор RDD и трансформаций
- Таким образом, запросы в форме Dataframe/Dataset/SQL в конечно счете преобразуются в RDD

# Сравнение RDD, Dataframe и Dataset

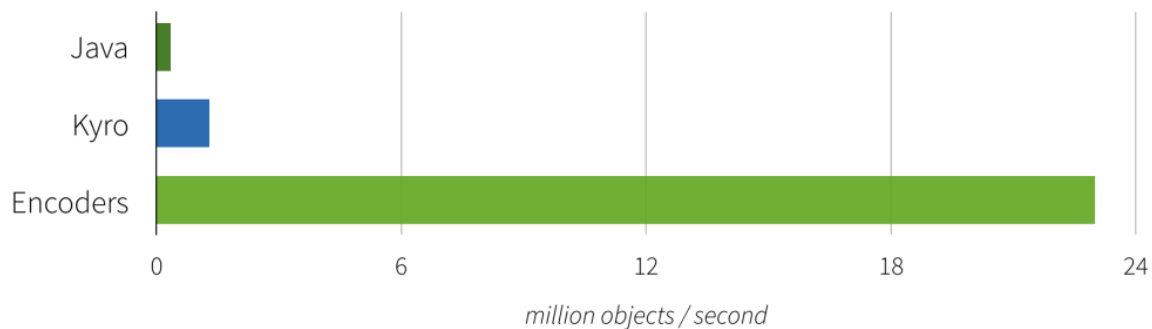
Distributed Wordcount



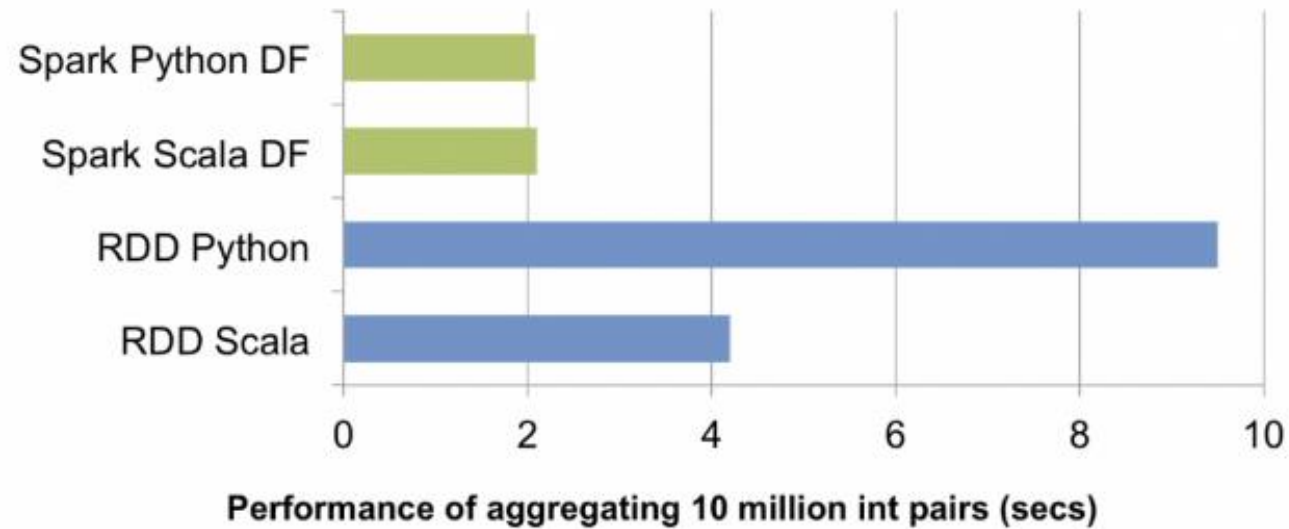
Memory Usage when Caching



Serialization / Deserialization Performance



# Сравнение RDD, Dataframe и Dataset





# Операции над RDD



## **Трансформации** (transformation)

Возвращает новую RDD

Примеры, `map()`, `filter()` и др.



## **Действия** (action)

Возвращает итоговый результат на Driver или записывает в постоянную память (например, HDFS)

Примеры, `count()`, `collect()` и др.



## **Выполнение по требованию** (Lazy evaluation)

Непосредственно выполнение операций начинается только после действия, даже если до него было несколько трансформаций

# Трансформации

Трансформация	Коллекция RDD	
	Запись	Ключ-Значение
map(func)	+	+
filter(func)	+	+
flatMap(func)	+	+
mapPartitions(func)	+	+
mapPartitionsWithIndex(func)	+	+
sample(withReplacement, fraction, seed)	+	+
union(otherDataset)	+	+
intersection(otherDataset)	+	+
distinct([numTasks])	+	+
groupByKey([numTasks])	-	+
reduceByKey(func, [numTasks])	-	+
aggregateByKey(zeroValue)(seqOp, combOp, [numTasks])	-	+
sortByKey([ascending], [numTasks])	-	+
join(otherDataset, [numTasks])	-	+
cogroup(otherDataset, [numTasks])	-	+
cartesian(otherDataset)	+	+
coalesce(numPartitions)	+	+
repartition(numPartitions)	+	+
repartitionAndSortWithinPartitions(partitioner)	+	+

Действия	Коллекция RDD	
	Запись	Ключ-Значение
<code>reduce(func)</code>	+	+
<code>collect()</code>	+	+
<code>count()</code>	+	+
<code>first()</code>	+	+
<code>take(n)</code>	+	+
<code>takeSample(withReplacement,num, [seed])</code>	+	+
<code>takeOrdered(n, [ordering])</code>	+	+
<code>saveAsTextFile(path)</code>	+	+
<code>saveAsSequenceFile(path)</code> (Java and Scala)	+	+
<code>saveAsObjectFile(path)</code> (Java and Scala)	+	+
<code>countByKey()</code>	-/+	+
<code>foreach(func)</code>	+	+

# Обзор некоторых трансформаций и действий

**partitionBy**(*numPartitions*, *partitionFunc*=<function portable\_hash at 0x7f7534d08d70>)

Возвращает перетасованное RDD с использованием функции *partitioner*

**coalesce**(*numPartitions*, *shuffle*=False)

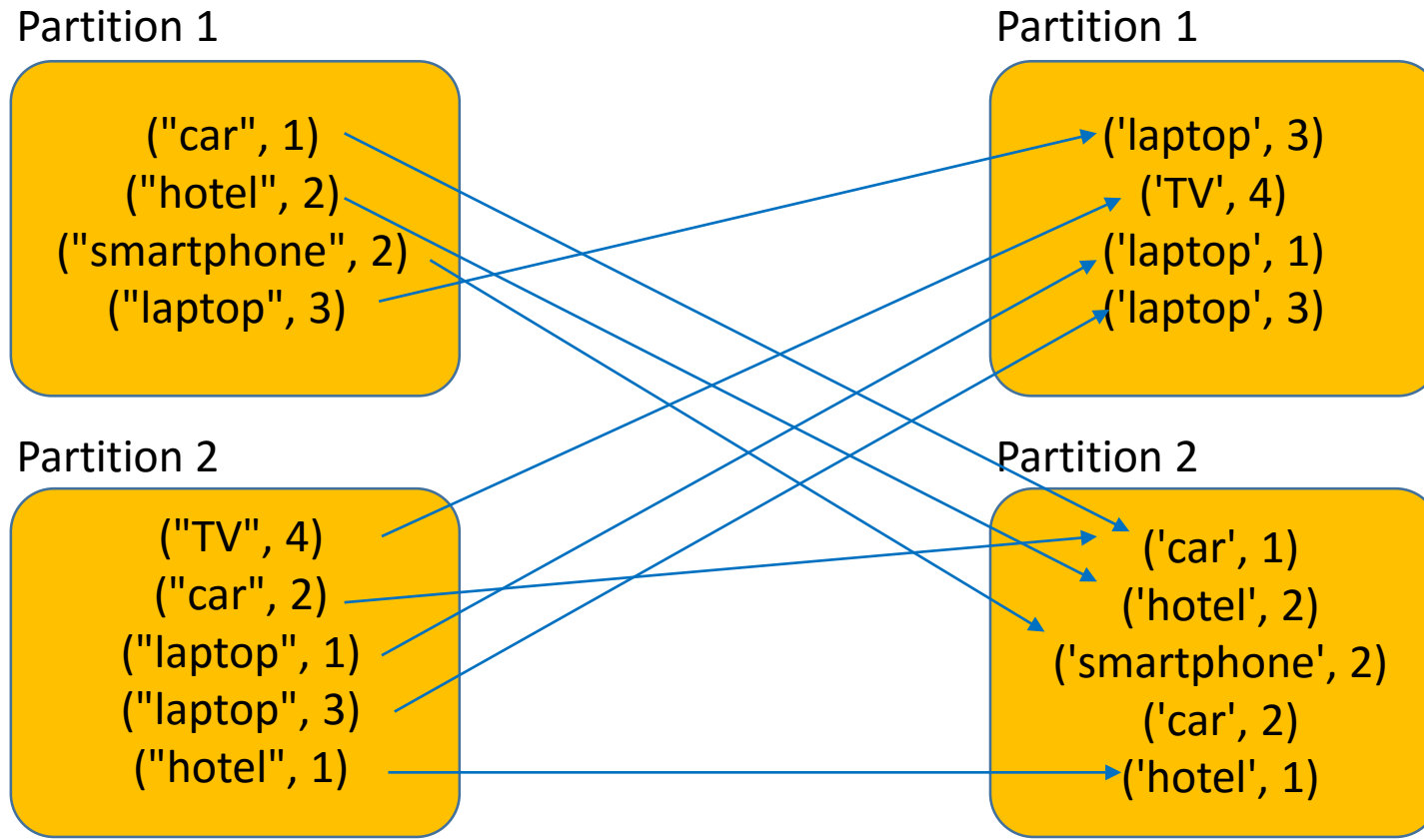
Возвращает новую RDD с уменьшенным количеством *partitions* до *numPartitions*

**repartition**(*numPartitions*)

Возвращает новую RDD с количеством *partitions* равным *numPartitions*

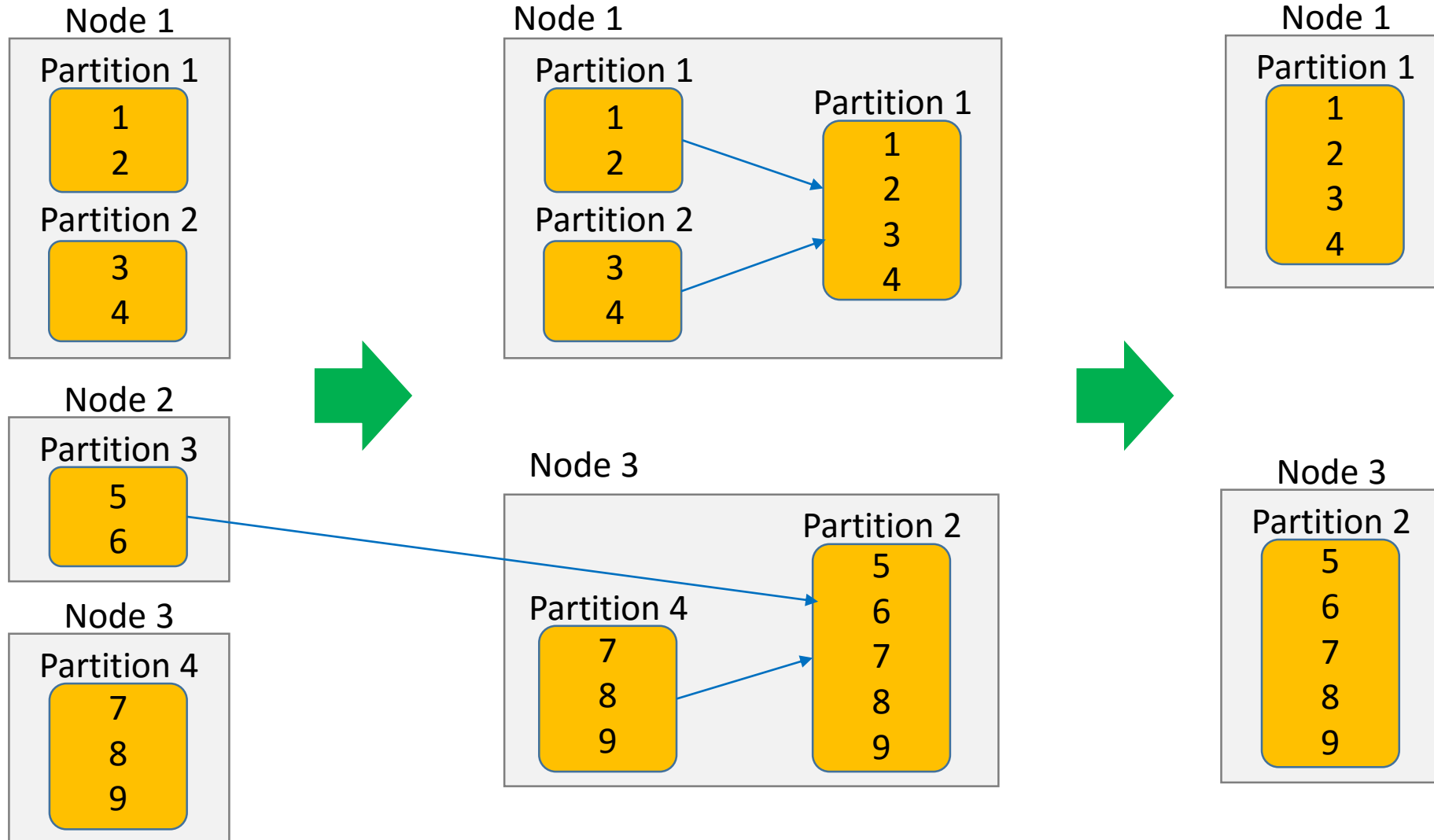
# Пример PartitionBy

```
purchases_price_rdd = sc.parallelize(purchases_price, 2)
part_rdd = purchases_price_rdd.partitionBy(2)
```



# Пример Coalesce

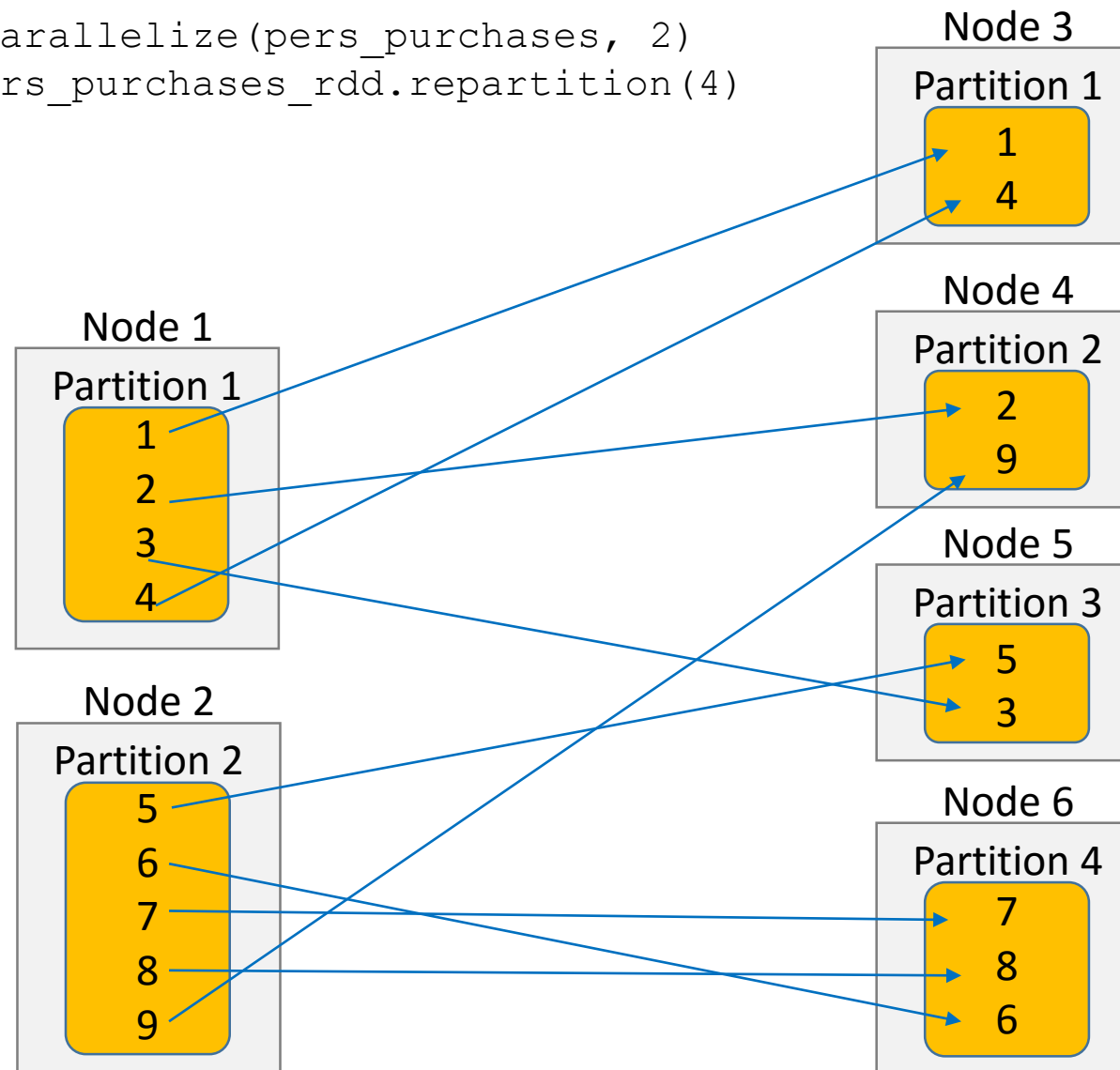
```
data_rdd = sc.parallelize(data, 4)
data_coarse_rdd = data_rdd.coalesce(2)
```





# Repartition

```
pers_purchases_rdd = sc.parallelize(pers_purchases, 2)
data_repart_decr_rdd = pers_purchases_rdd.repartition(4)
```



## Действие

**aggregate**(*zeroValue, seqOp, combOp*)

Агрегирует элементы каждой partition и потом результат от каждого partition

## Трансформация

**aggregateByKey**(*zeroValue, seqFunc, combFunc,*  
*numPartitions=None,*  
*partitionFunc=<function portable\_hash at 0x7f7534d08d70>*)

Агрегирует значения по ключу

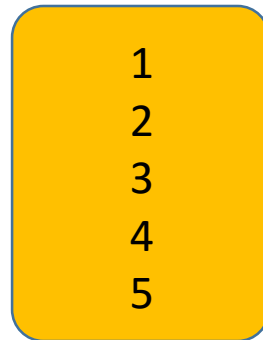
**combineByKey**(*createCombiner, mergeValue, mergeCombiners,*  
*numPartitions=None,*  
*partitionFunc=<function portable\_hash at 0x7f7534d08d70>*)

Комбинирует элементы по ключу. Более общая функция по сравнению с **aggregateByKey**.

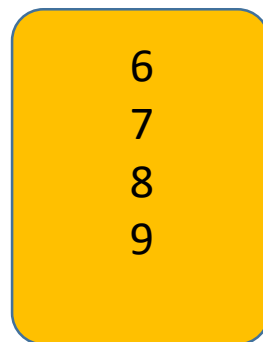
# Пример Aggregate

```
data_agg = data_rdd.aggregate((0, 0),
                               (lambda x, value: (x[0] + value, x[1] + 1)),
                               (lambda x, y: (x[0] + y[0], x[1] + y[1])))
```

Partition 1

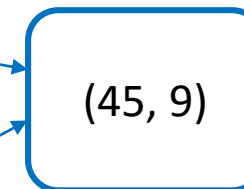


Partition 2



(0, 0)  
(1, 1)  
(3, 2)  
(6, 3)  
(10, 4)  
**(15, 5)**

(0, 0)  
(6, 1)  
(13, 2)  
(21, 3)  
**(30, 4)**



# Пример AggregateByKey

```
agg_key_rdd = pers_purchases_rdd.aggregateByKey((0, 0),
                                                (lambda x, value: (x[0] + value, x[1] + 1)),
                                                (lambda x, y: (x[0] + y[0], x[1] + y[1])))
```

## Partition 1

("car", 1)  
("hotel", 2)  
("smartphone", 2)  
("laptop", 3)

## Partition 2

("TV", 4)  
("car", 2)  
("laptop", 1)  
("laptop", 3)  
("hotel", 1)]

"car": (0, 0) → (1, 1)  
"hotel": (0, 0) → (2, 1)  
"smartphone": (0, 0) → (2, 1)  
"laptop": (0, 0) → (3, 1)

"TV": (0, 0) → (4, 1)  
"car": (0, 0) → (2, 1)  
"laptop": (0, 0) → (1, 1) → (4, 2)  
"hotel": (0, 0) → (1, 1)

## Partition 1

('TV', (4, 1))  
('car', (3, 2))  
('laptop', (7, 3))  
('smartphone', (2, 1))  
('hotel', (3, 2))

# Пример CombineByKey

```
combine_key_rdd = purchases_price_rdd.combineByKey((lambda value: (value, 1)),
                                                    (lambda x, value: (x[0] + value, x[1] + 1)),
                                                    (lambda x, y: (x[0] + y[0], x[1] + y[1])), 1)
```

## Partition 1

("car", 1)  
("hotel", 2)  
("smartphone", 2)  
("laptop", 3),

## Partition 2

("TV", 4)  
("car", 2)  
("laptop", 1)  
("laptop", 3)  
("hotel", 1)]

"car": (1, 1) → "car": (1, 1)  
"hotel": (2, 1) → "hotel": (2, 1)  
"smartphone": (2, 1) → "smartphone": (2, 1)  
"laptop": (3, 1) → "laptop": (3, 1)

"TV": (4, 1) → "TV": (4, 1)  
"car": (2, 1) → "car": (2, 1)  
"laptop": (1, 1) → "laptop": (4, 2)  
"laptop": (3, 1) → "laptop": (4, 2)  
"hotel": (1, 1) → "hotel": (1, 1)

## Partition 1

('TV', (4, 1))  
('car', (3, 2))  
('laptop', (7, 3))  
('smartphone', (2, 1))  
('hotel', (3, 2))

# Кэширование RDD

# Persist vs Cache

- Применяется для сохранения RDD в памяти и повторного его использования другими работами (job) одного приложения
- Для кэша есть две команды: `cache` (сохраняет в оперативной памяти) и `persist` (место и форму хранения можно задать, см. след. слайд)
- Кеш – отказоустойчив – если некоторая часть RDD будет потеряна, она будет автоматически пересчитана посредством трансформаций, в результате которых она была изначально создана
- Spark автоматически сохраняет некоторые промежуточные данные в операциях с перетасовкой (shuffling, например, `reduceByKey`). Это делается, чтобы избежать повторного пересчета всех входных данных, если узел выйдет из строя во время перетасовки.

Уровень хранения	Определение
MEMORY_ONLY (по умолчанию)	Сохраняет RDD в виде несериализованных Java объектов в JVM. Если данные не помещаются в память, некоторые части будут пересчитывать на лету при каждом обращении
MEMORY_AND_DISK	Сохраняет RDD в виде несериализованных Java объектов в JVM. Если данные не помещаются в память, некоторые части будут записана на диск
MEMORY_ONLY_SER (Java and Scala)	Как MEMORY_ONLY, но сохраняет RDD в виде сериализованных Java объектов в JVM. В результате RDD занимает меньше места в памяти, но требует дополнительных CPU затратных операций при чтении
MEMORY_AND_DISK_SER (Java and Scala)	Как MEMORY_AND_DISK, но сохраняет RDD в виде сериализованных Java объектов в JVM
DISK_ONLY	Сохраняет RDD на диске
MEMORY_ONLY_2, MEMORY_AND_DISK_2 и др.	Аналогично предыдущим, но создает две реплики RDD на двух разных узлах кластера
OFF_HEAP (experimental)	Как MEMORY_ONLY_SER, но хранит данные в off-heap памяти

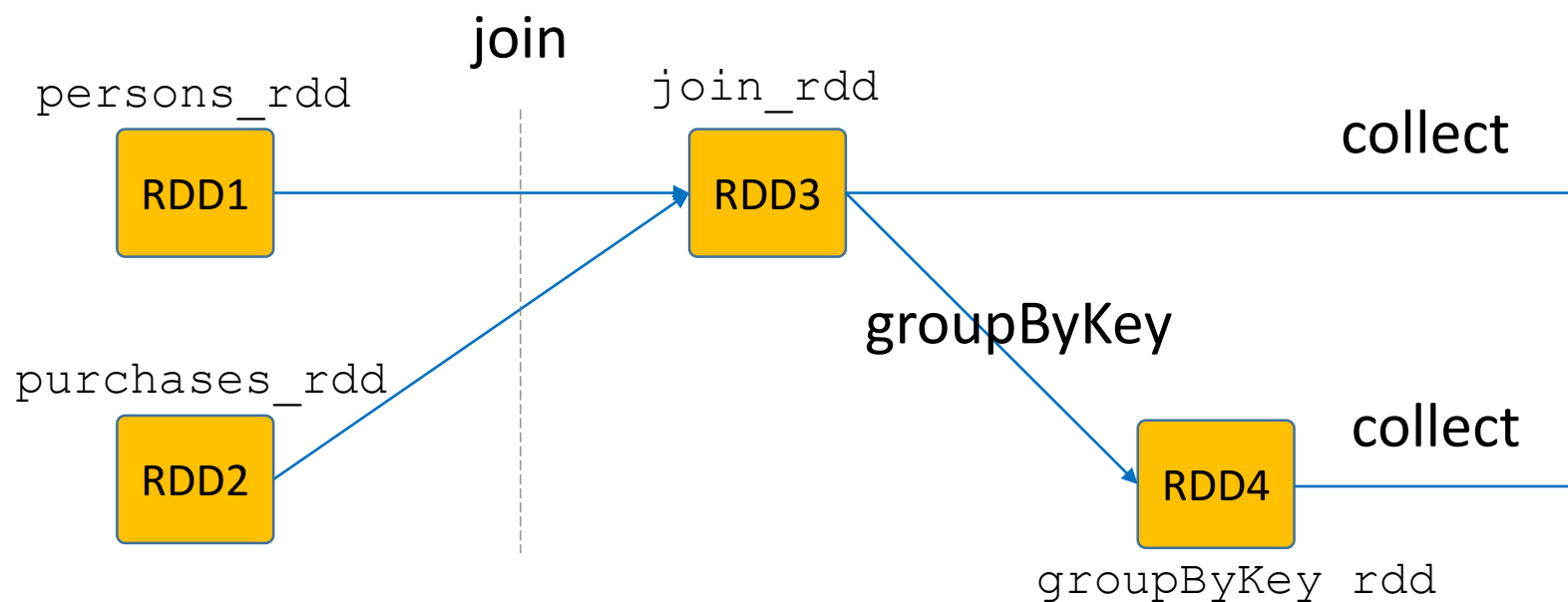


# Пример, RDD без persist()

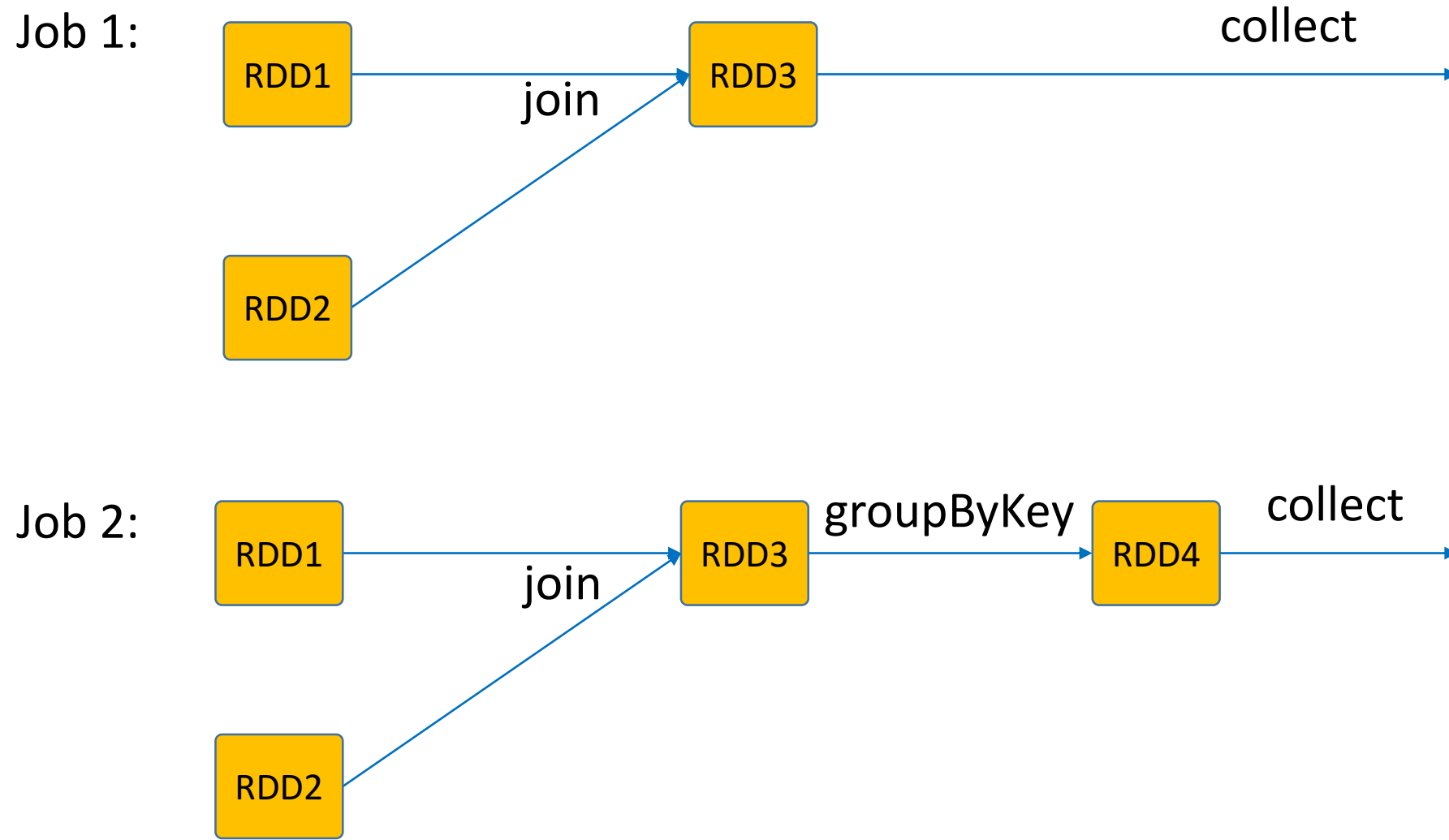
```
persons_rdd = sc.parallelize(persons, 2)
purchases_rdd = sc.parallelize(purchases, 4)

join_rdd = persons_rdd.join(purchases_rdd, numPartitions=2)
join_rdd.collect()

groupByKey_rdd = join_rdd.groupByKey()
groupByKey_rdd.collect()
```



# Пример, работы без persist()



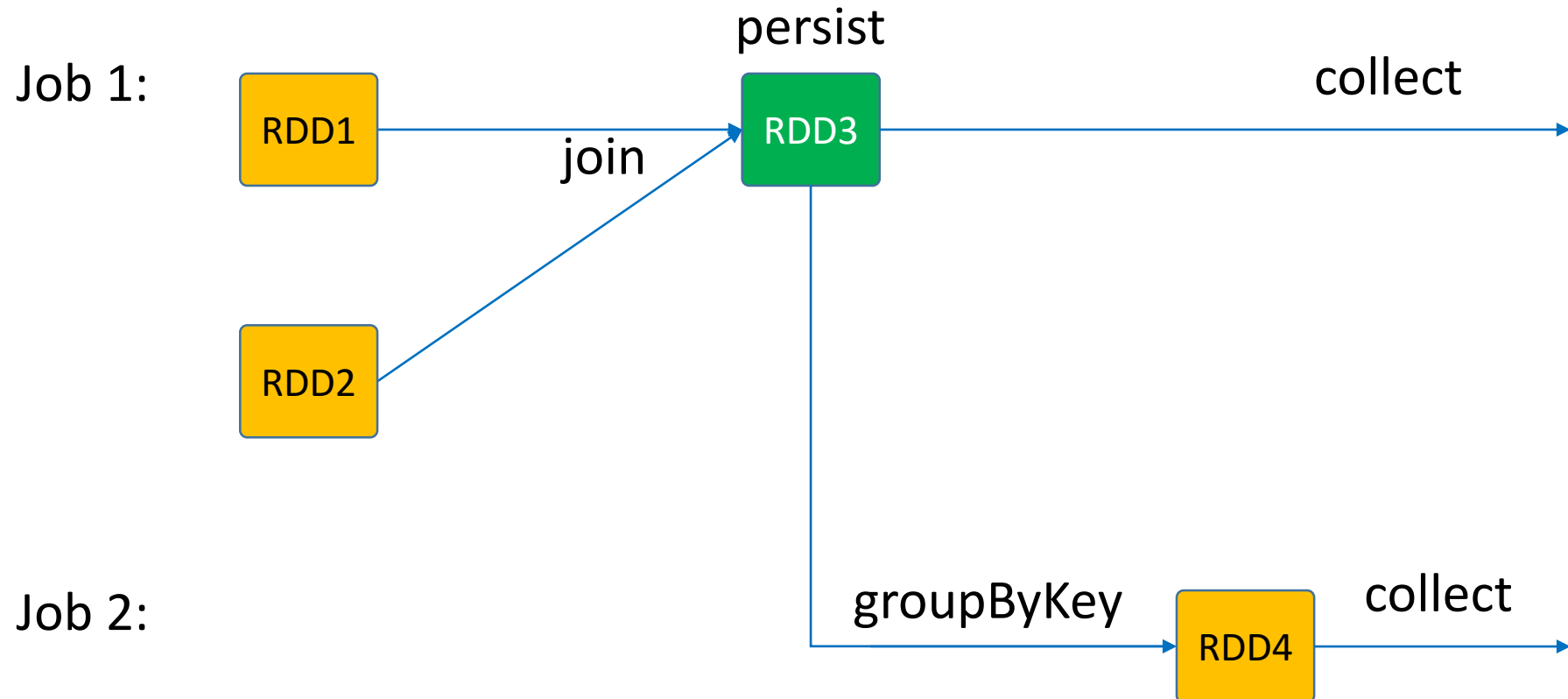
# Пример, RDD с persist()

```
persons_rdd = sc.parallelize(persons, 2)
purchases_rdd = sc.parallelize(purchases, 4)

join_rdd = persons_rdd.join(purchases_rdd, numPartitions=2).persist()
join_rdd.collect()

groupByKey_rdd = join_rdd.groupByKey()
groupByKey_rdd.collect()
```

# Пример, работы с persist()

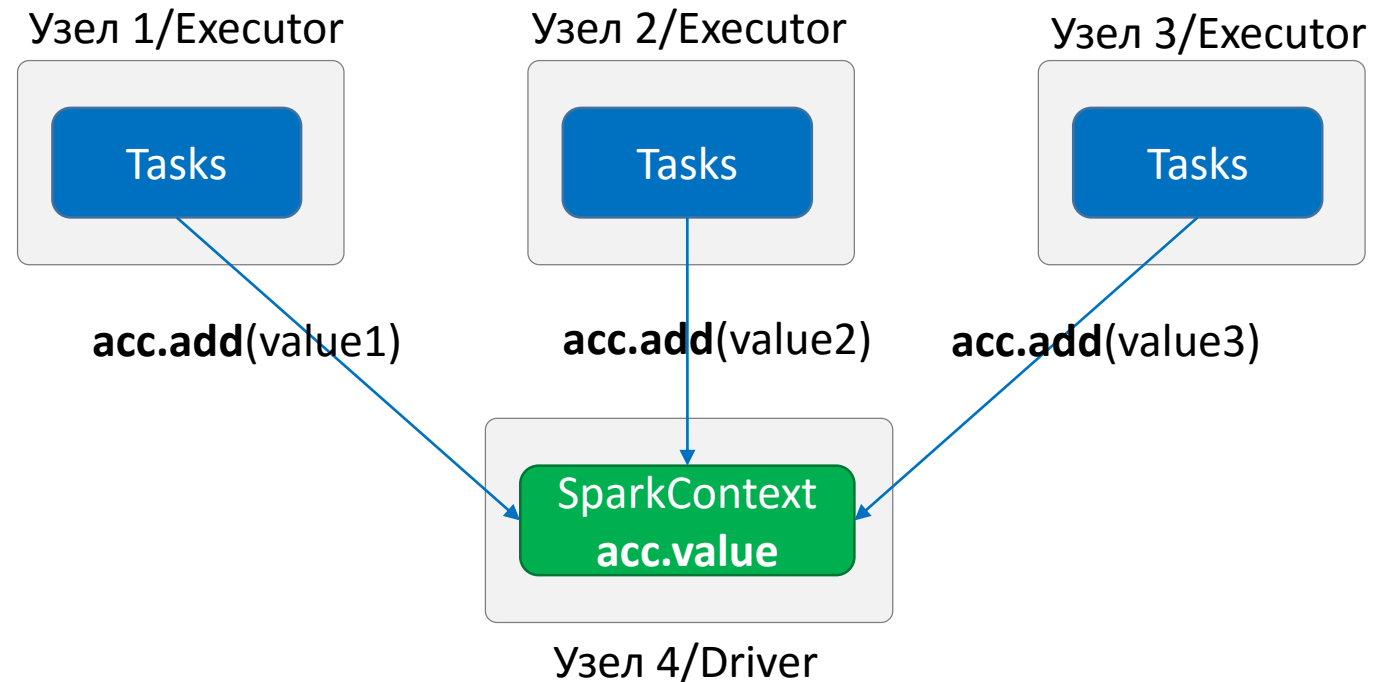


# Переменные Accumulator и Broadcast

# Accumulator

**Accumulator** накапливает значения полученные от выполняемых задач (**task**)

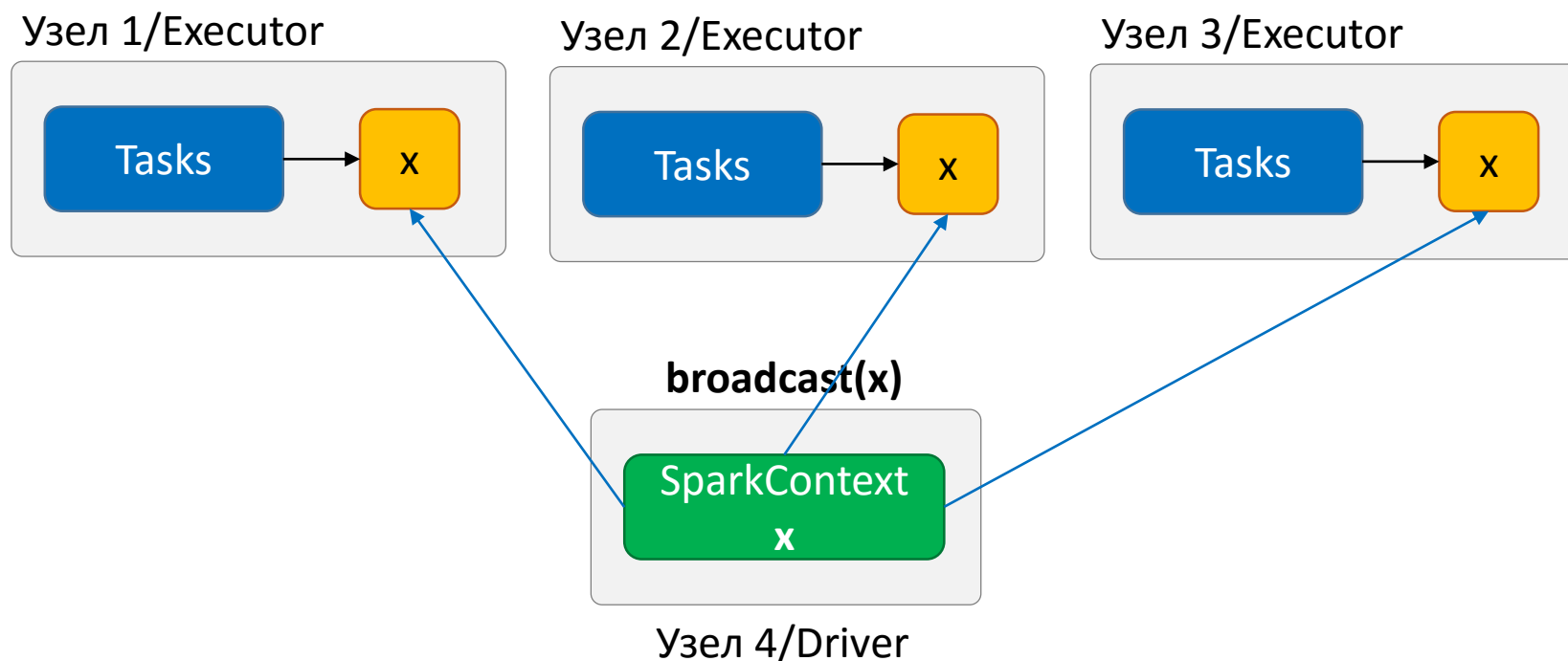
Значение **accumulator** может прочитать только **driver**



# Broadcast

**Broadcast** переменная позволяет передавать данные на узлы в виде доступных только на чтение кэшированных данных, которые затем используются при выполнении задач (**task**) по схеме «один кэш – множество задач».

Это более эффективно, чем использовать глобальные переменные, которые копируются для каждой задачи (**task**)



Learning Spark by H. Karau, A. Konwinski, P. Wendell, and M. Zaharia (book)

[Spark](#) (github source code)

Spark: The Definitive Guide by B. Chambers, M. Zaharia (book)

[RDD — Resilient Distributed Dataset](#) (e-book)

[SparkContext — Entry Point to Spark Core](#) (e-book)

[Cluster Mode Overview](#) (doc)

[Job Scheduling](#) (doc)

[Submitting Applications](#) (doc)

[Tuning Spark](#) (doc)

[Spark Configuration](#) (doc)

[RDD Programming Guide](#) (doc)

[Running Spark Applications on YARN](#) (cloudera doc)

[How-to: Tune Your Apache Spark Jobs \(Part 2\)](#) (blog)

[A Tale of Three Apache Spark APIs: RDDs, DataFrames, and Datasets](#) (blog)

[Distribution of Executors, Cores and Memory for a Spark Application running in Yarn](#) (blog)

[Introducing DataFrames in Apache Spark for Large Scale Data Science](#) (blog)

[Introducing Apache Spark Datasets](#) (blog)