Московский Государственный Технический Университет им. Н.Э. Баумана

СИСТЕМЫ ОБРАБОТКИ БОЛЬШИХ ДАННЫХ

Разработка приложений

к.т.н.
Папулин Сергей Юрьевич
papulin_bmstu@mail.ru

Москва, 2016/18

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

Особенности 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

Словарь Spark

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

Resilient Distributed Dataset

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

RDD

- Partition 1
- Partition 2
- Partition 3
- Partition 4
- Partition 5

Физическое расположение

Узел 1
- Partition 1

Узел 2
- Partition 2
- Partition 3

Узел 3
- Partition 4
- Partition 5

Параметры RDD

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

Операции 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

Состав работы (Job)

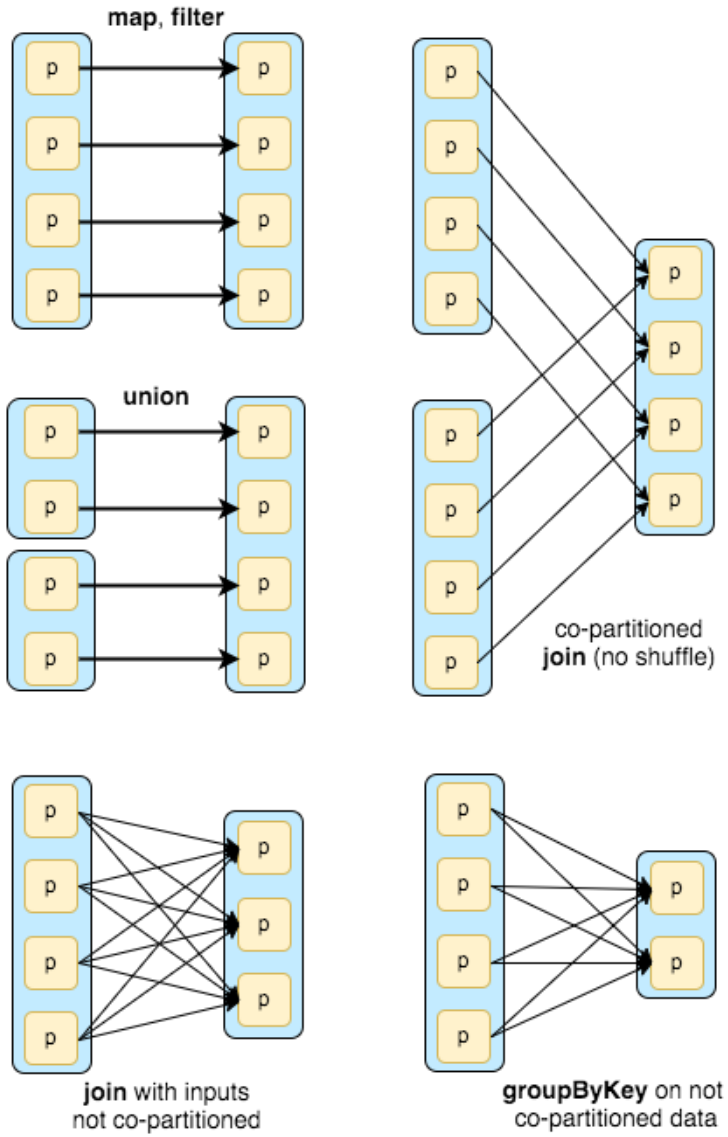
- **Работа** (job) – единица вычисление топ уровня

  Запускается после того, как в программе встретилась операция-действие (action)

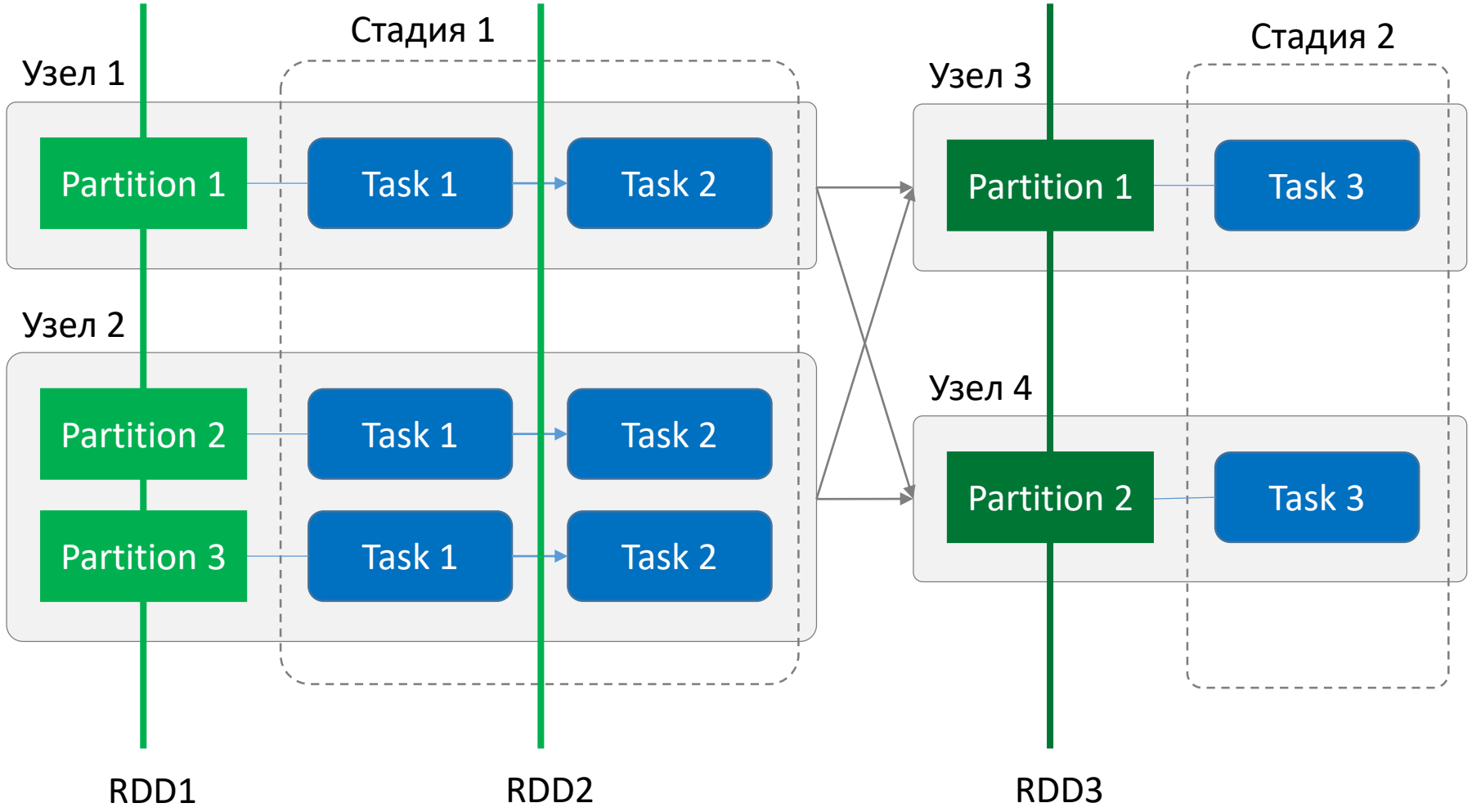
- **Работа** (job) разбивается на **стадии** (stages)

- **Стадии** (stages) состоят из **задач** (tasks). Задача – минимальная единица выполнения

Зависимости между RDD



Взаимосвязь между частями, задачами и стадиями



Архитектура Spark

Driver/Executor

- Spark построен на master/slave архитектуре с одним **центральным координатором** и множеством распределенных **worker’ов**
- Центральный координатор – **driver**
- Driver взаимодействует с множеством распределенных woker’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 **drivere**
- Spark приложение запускается на множестве машин с использованием внешнего сервиса – **cluster manager**

Driver

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

Driver выполняет две основные функции:

- Конвертирует программу в задания (**task**и)
- Планирует и запускает выполнение заданий на **executor**ах

Executor

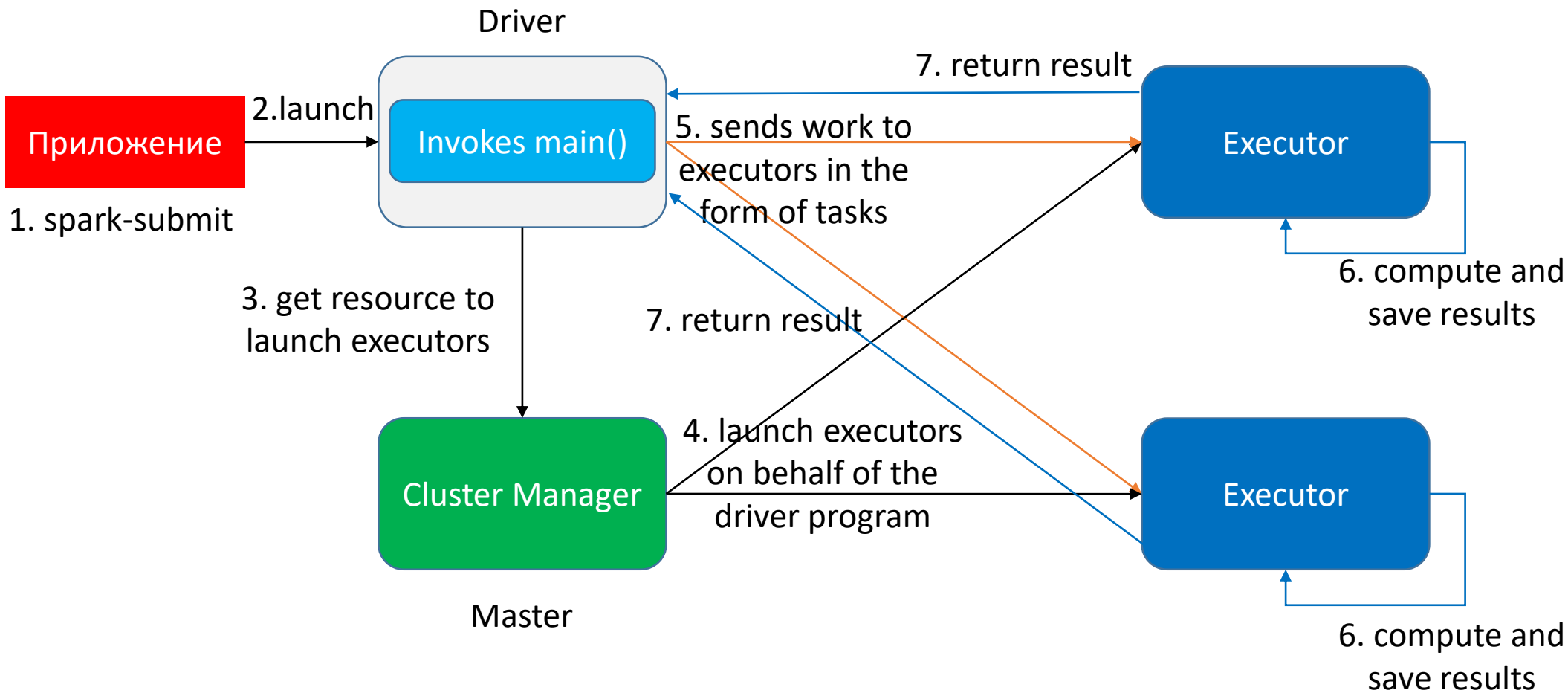
Spark **executor** – процесс, который отвечает за запуск и выполнение отдельных задач (**task**) некоторой Spark работы (**job**)

Executor запускается при развертывании 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

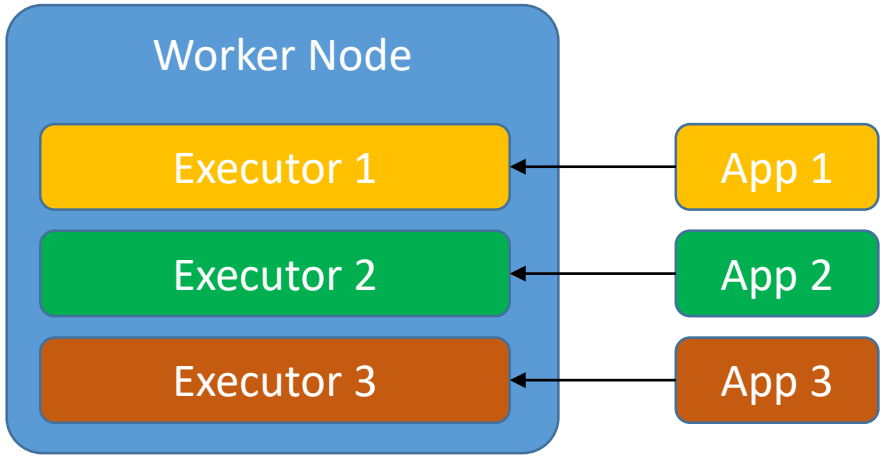


Развертывание Spark приложения

- Пользователь дает команду на запуск приложения посредством команды **spark-submit**
- **spark-submit** запускает программу **driver** с указанием основного входа программы пользователя (main class)
- **Driver** создает объект **SparkContext**, который используется для доступа к кластеру
- Cluster Manager запускает **executor'**ы на рабочих узлах
- Jar и python файлы, переданные в **SparkContext,** отправляются **executor'**ам
- **Driver** запускает пользовательское приложение, которые отправляется в виде **задач** (tasks) на **executor'**ы с использованием **SparkContext**
- **Задачи** выполняются на **executor'**ах и результат собирается на **driver'**е или обрабатывается иным способом (например, foreach()) посредством операций-действий
- По завершению приложения или при вызове SparkContext.stop(), останавливаются **executor'**ы и освобождаются ресурсы

Сопоставление Driver/Executor/Job/Task/RDD

- Каждое Spark **приложение** имеет свои **executor**ы
- Задачи (**tasks**) от разных **приложений** запускаются на разных JVM
- Несколько задач (**task**) одного приложения могут запускаться на одном **executor**е



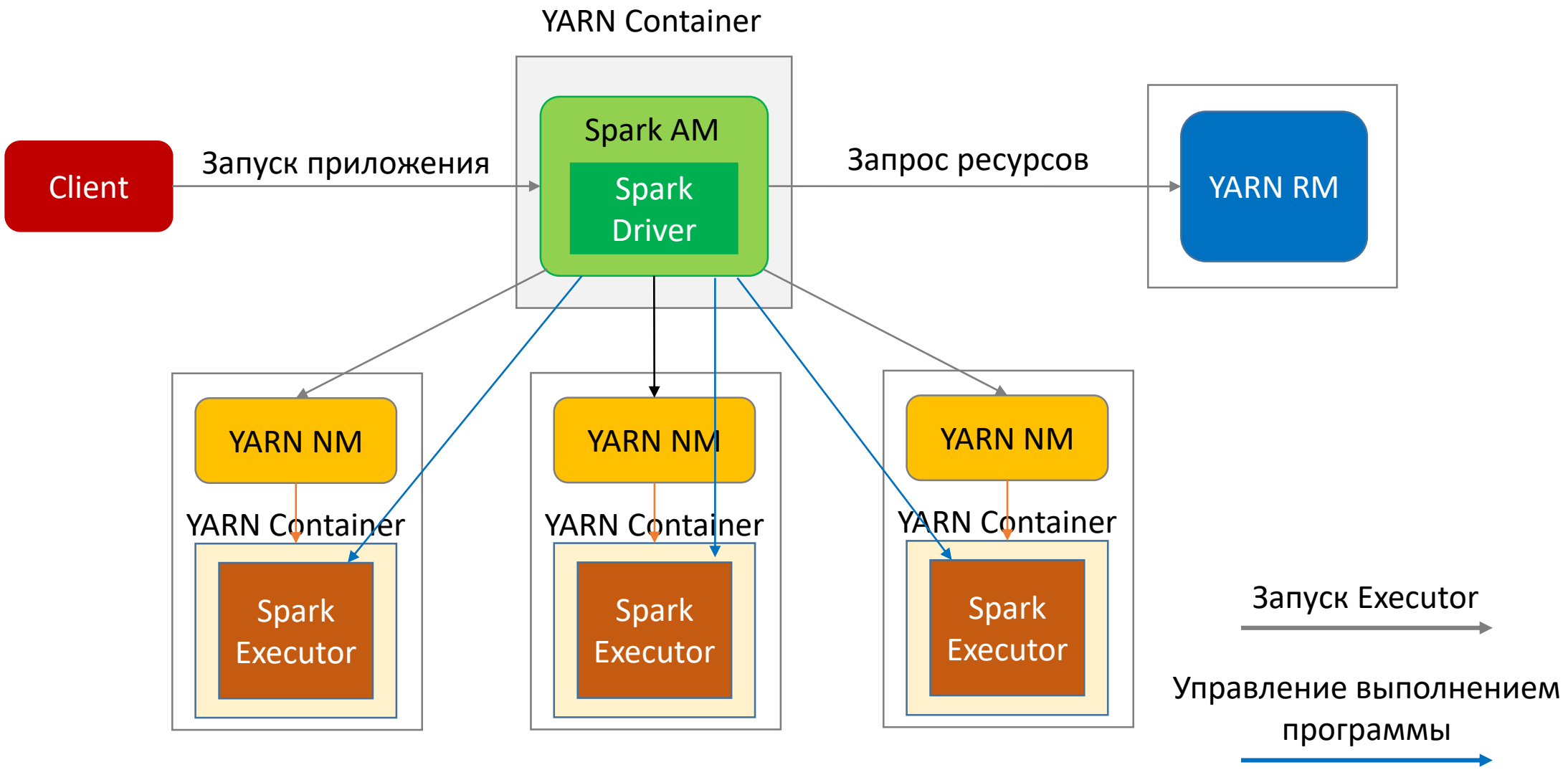
- Каждое **приложение** имеет не более одного **executor**а на каждом рабочем узле (worker) (standalone)
- **Задача** соответствует отдельному **потоку**
- Каждая **partition** является входом для отдельной **задачи** (task)

Сопоставление Driver/Executor/Job/Task/RDD

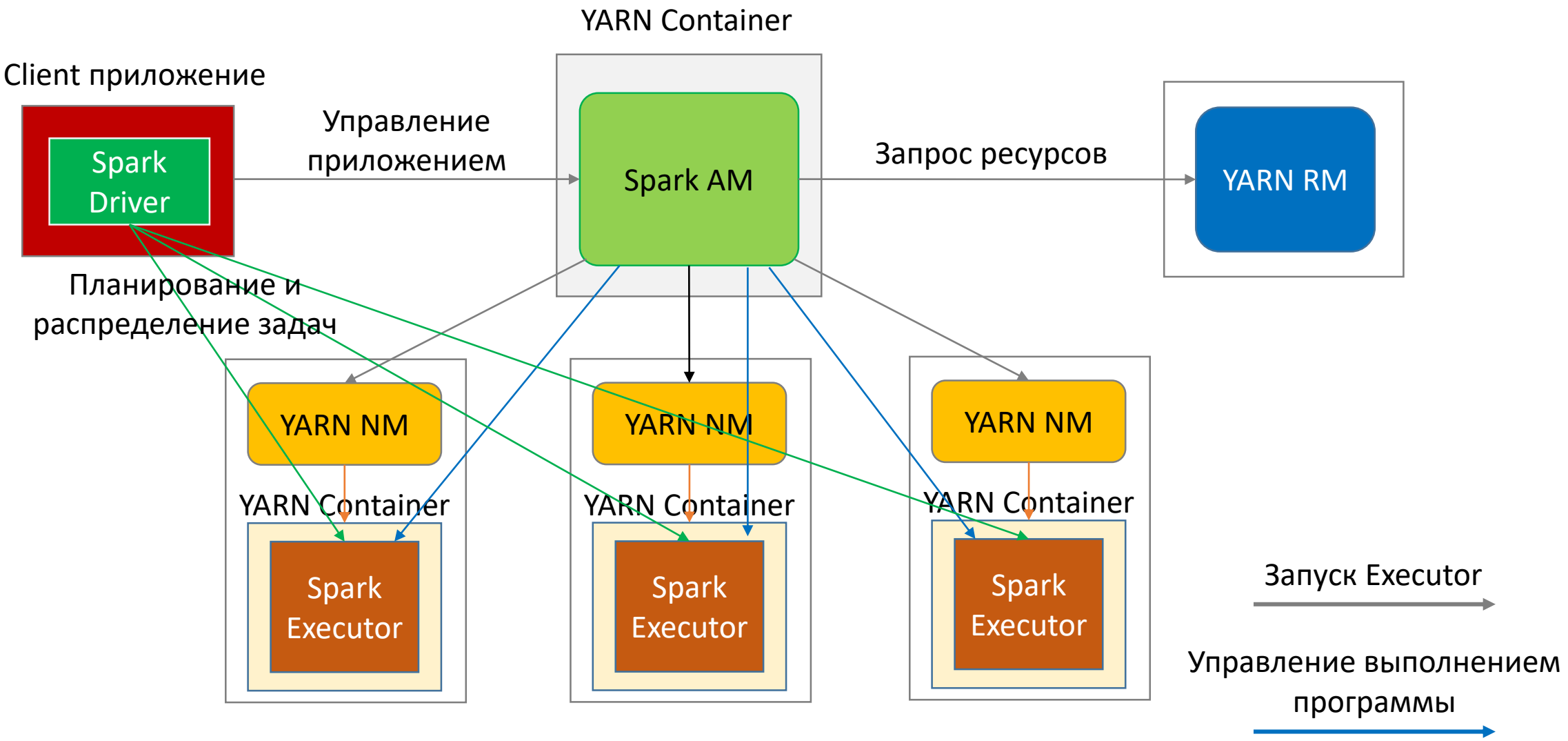
- Одно Spark приложение может иметь несколько работ (job) (для каждой операции-действия). Работы могут выполняться параллельно, если запущены разными потоками
- По умолчанию количество **partition**ов соответствует количеству всех доступных ядер (**core**)
- Количество **partition**ов >= количеству **executor**ов
- 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

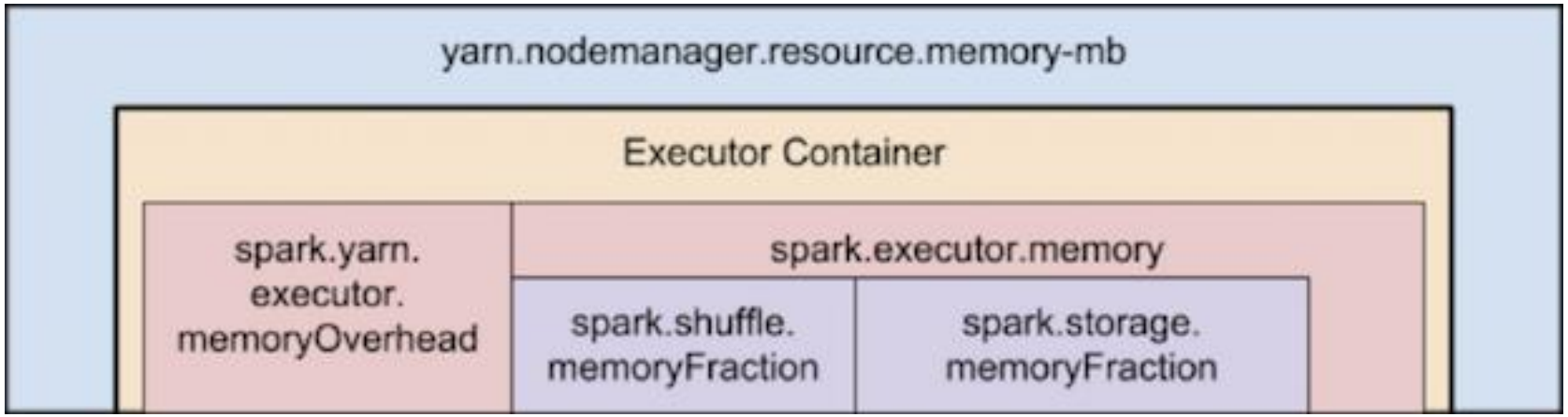
Развертывание Spark приложений на YARN. Режим кластера



Развертывание Spark приложений на YARN. Режим клиента



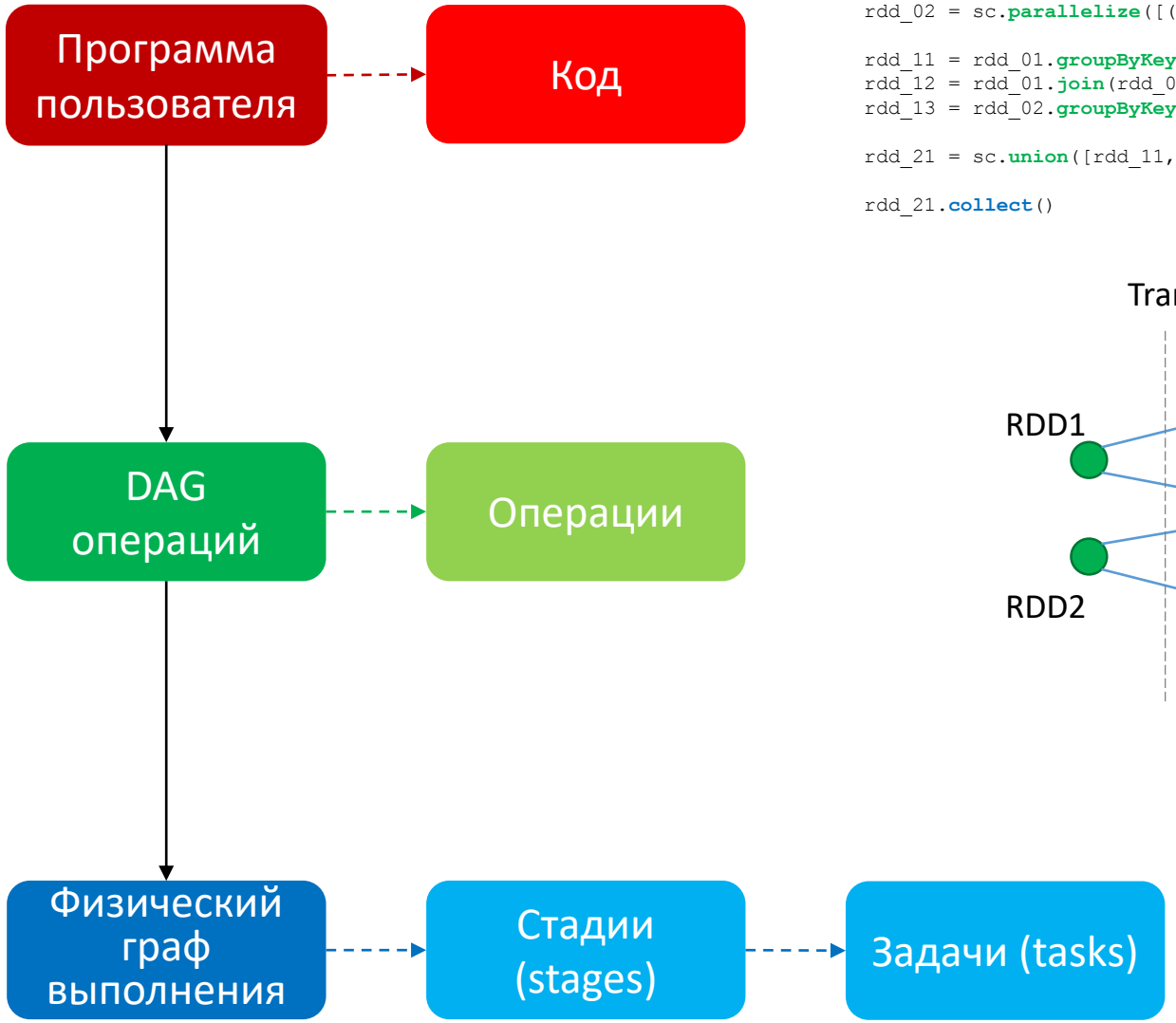
Память



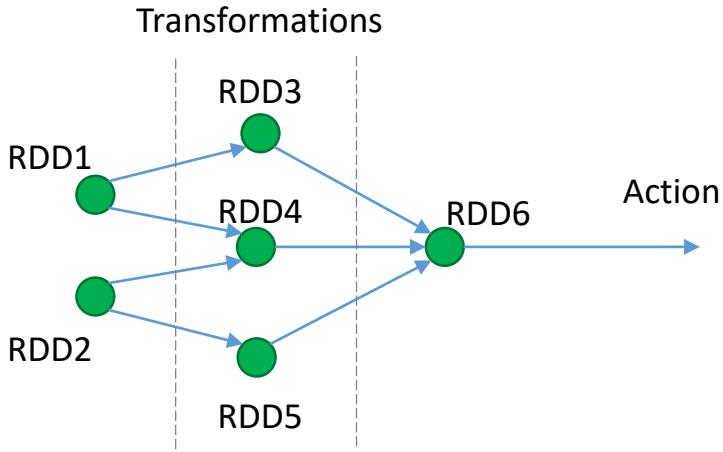
http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/

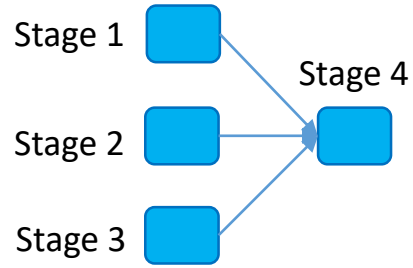
Запуск приложения Spark

Запуск приложения на Driver



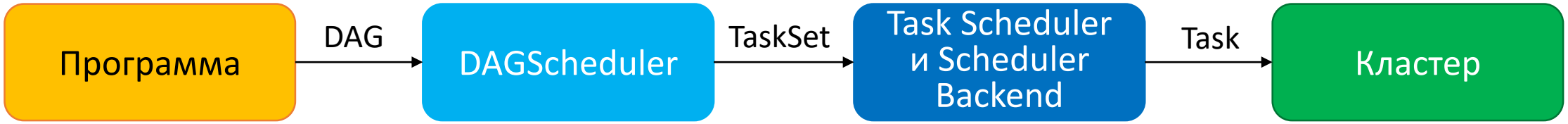

```
rdd_01 = sc.parallelize([(1, 2), (1, 3), (1, 2), (4, 4)], 2)
rdd_02 = sc.parallelize([(2, 2), (1, 1), (3, 3), (2, 2)], 2)

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



Построение DAG

- Разбиение DAG на стадии с задачами

- Получение **executor**ов в кластере
- Отправка **задач стадии** для запуска на **executor**ах
- Повторный запуск при неудаче

- Запуск и выполнение задач

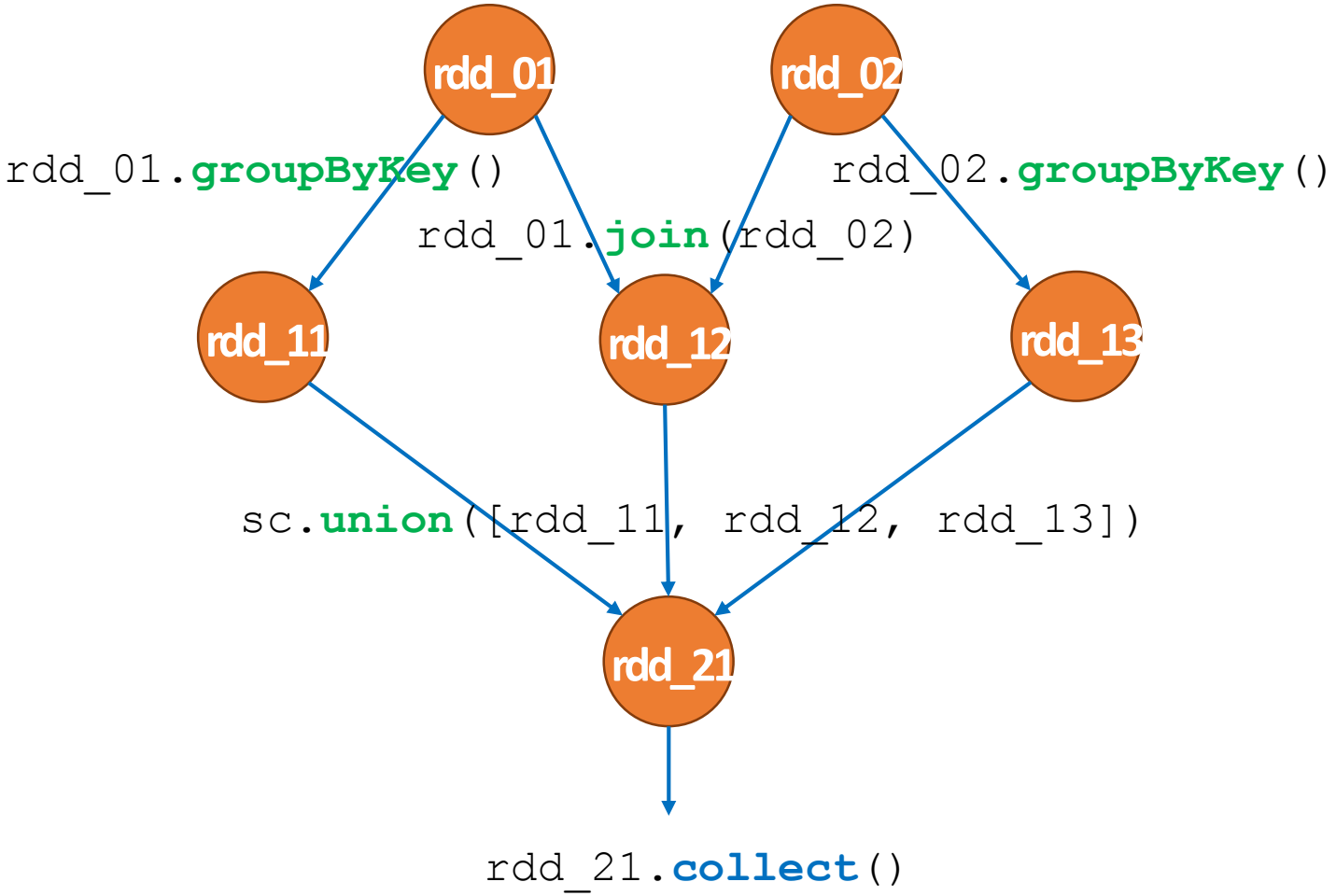
Чтобы создать **SparkContext** необходимо сначала создать **SparkConf**, который хранит конфигурации кластера

В Spark 2.x – **SparkSession**

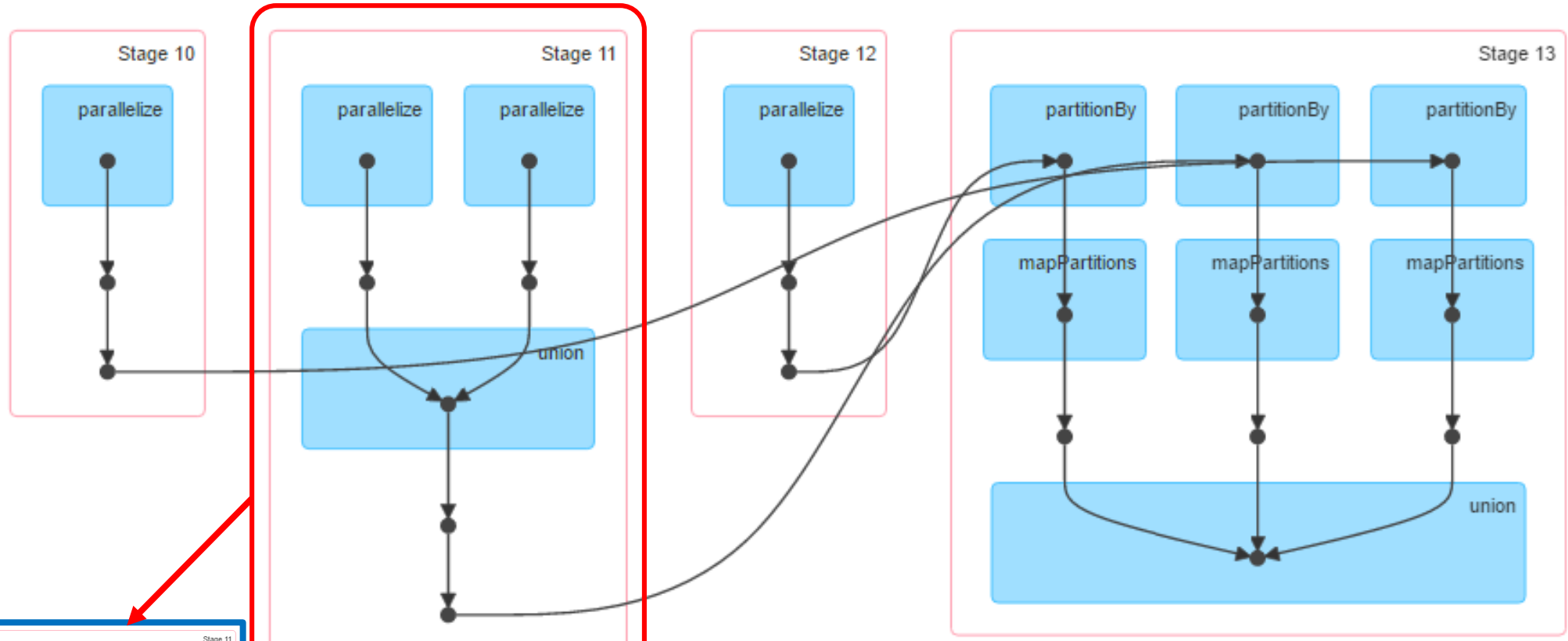
Пример кода программы

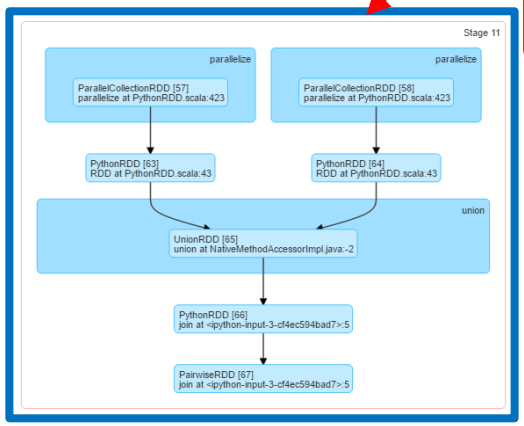
```
rdd_01 = sc.parallelize([(1, 2), (1, 3), (1, 2), (4, 4)], 2)
rdd_02 = sc.parallelize([(2, 2), (1, 1), (3, 3), (2, 2)], 2)

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

SparkSQL, Dataframe и Dataset

Spark SQL – Spark модуль для обработки структурированных данных

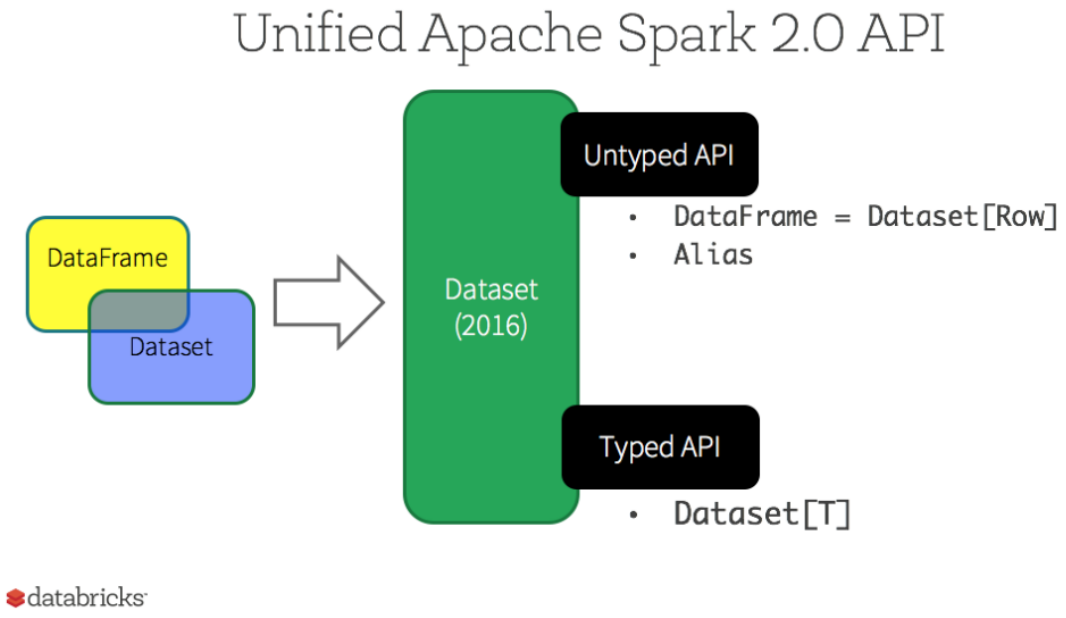
В отличие от RDD API, интерфейс предоставляет Spark больше информации о структуре данных и вычислениях, которые должны быть произведены.

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

Dataframe и Dataset

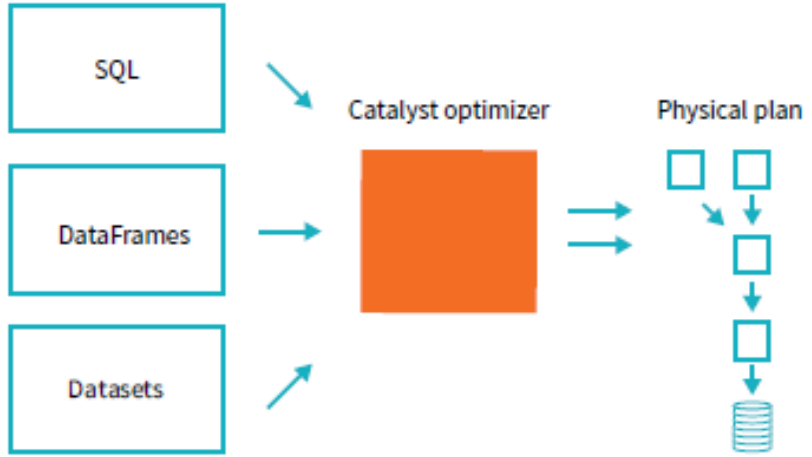


Язык программирование	Основная абстракция
Scala	Dataset[T] & DataFrame (alias for Dataset[Row])
Java	Dataset[T]
Python	DataFrame
R	DataFrame

https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html

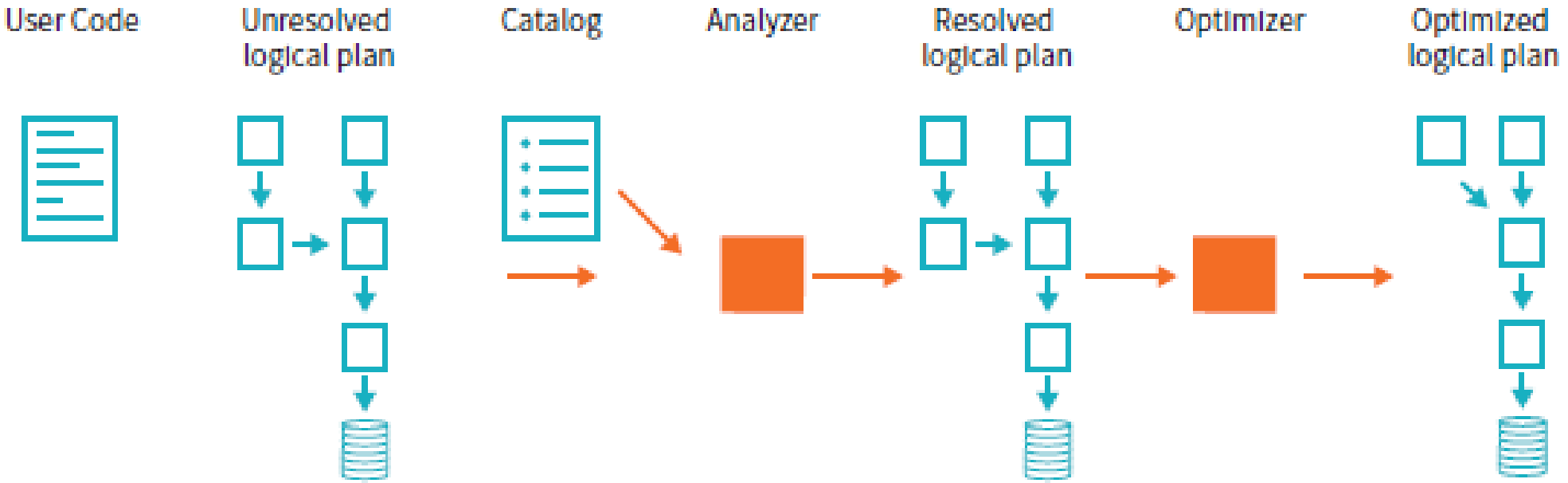
Этапы выполнения кода Dataframe/Dataset/SQL API

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



Spark: The Definitive Guide By Bill Chambers & Matei Zaharia (book)

Формирование логического плана



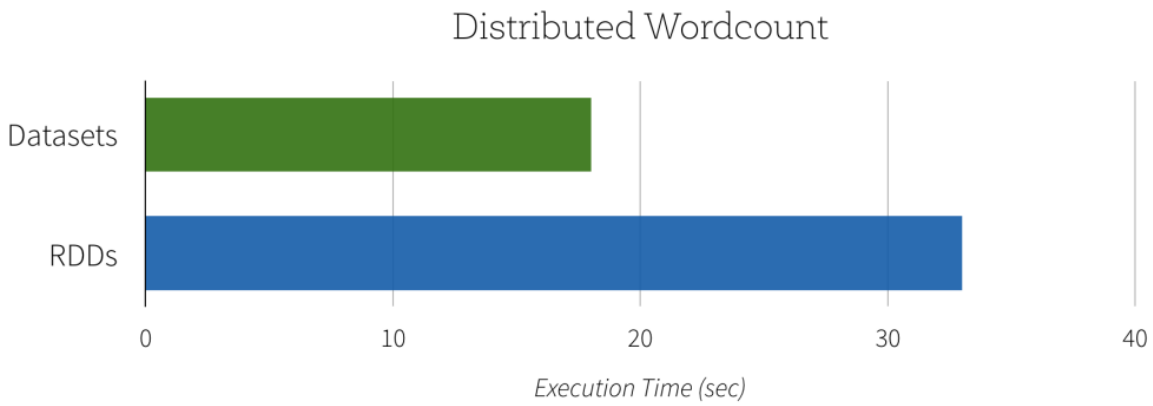
Spark: The Definitive Guide By Bill Chambers & Matei Zaharia (book)

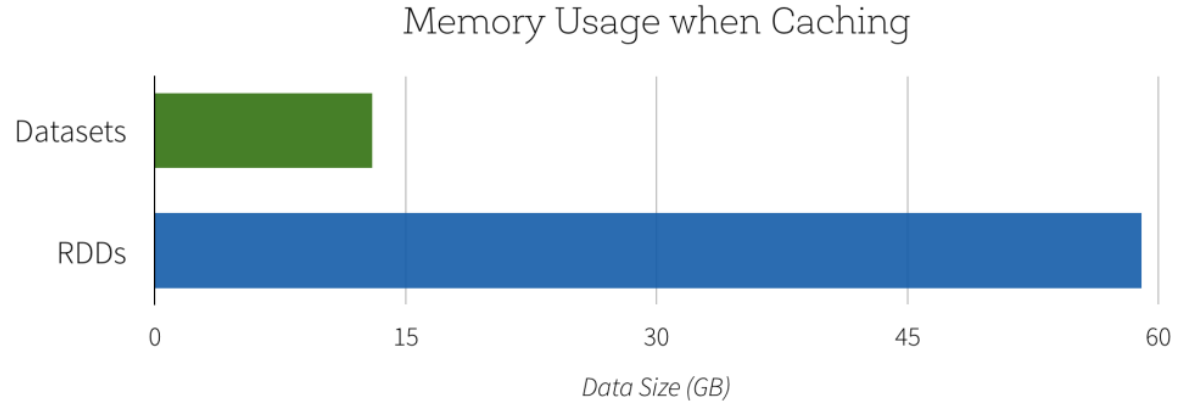
Формирование физического плана выполнения

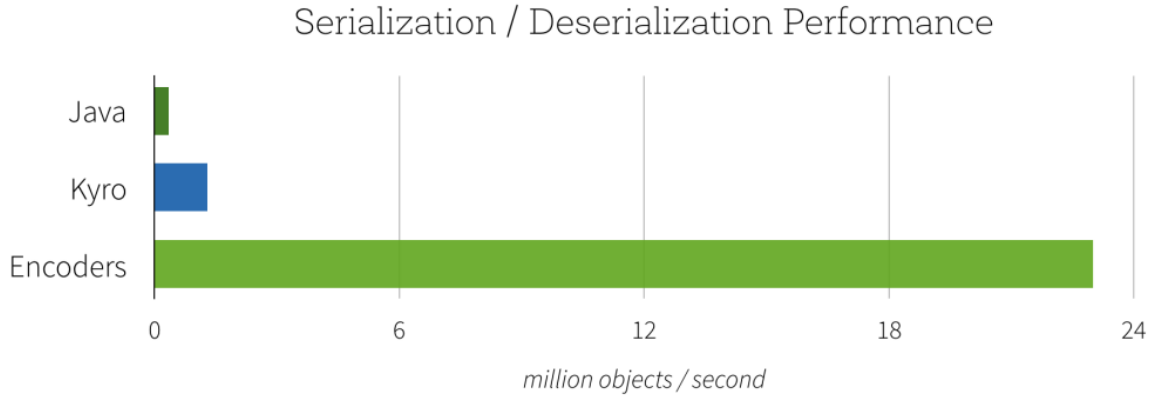
- Физический план представляет из себя набор RDD и трансформаций
- Таким образом, запросы в форме Datafame/Dataset/SQL в конечно счете преобразуются в RDD

Spark: The Definitive Guide By Bill Chambers & Matei Zaharia (book)

Сравнение RDD, Dataframe и Dataset

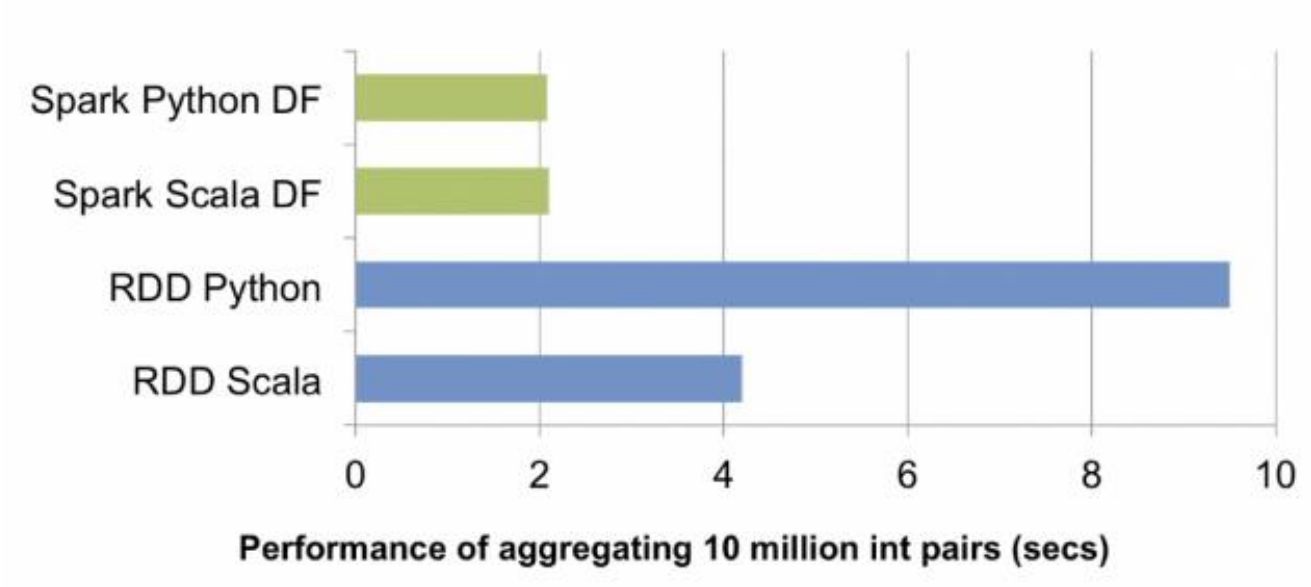






https://databricks.com/blog/2016/01/04/introducing-apache-spark-datasets.html

Сравнение RDD, Dataframe и Dataset



https://databricks.com/blog/2015/02/17/introducing-dataframes-in-spark-for-large-scale-data-science.html

Операции над RDD

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

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

Обзор некоторых трансформаций и действий

Операции

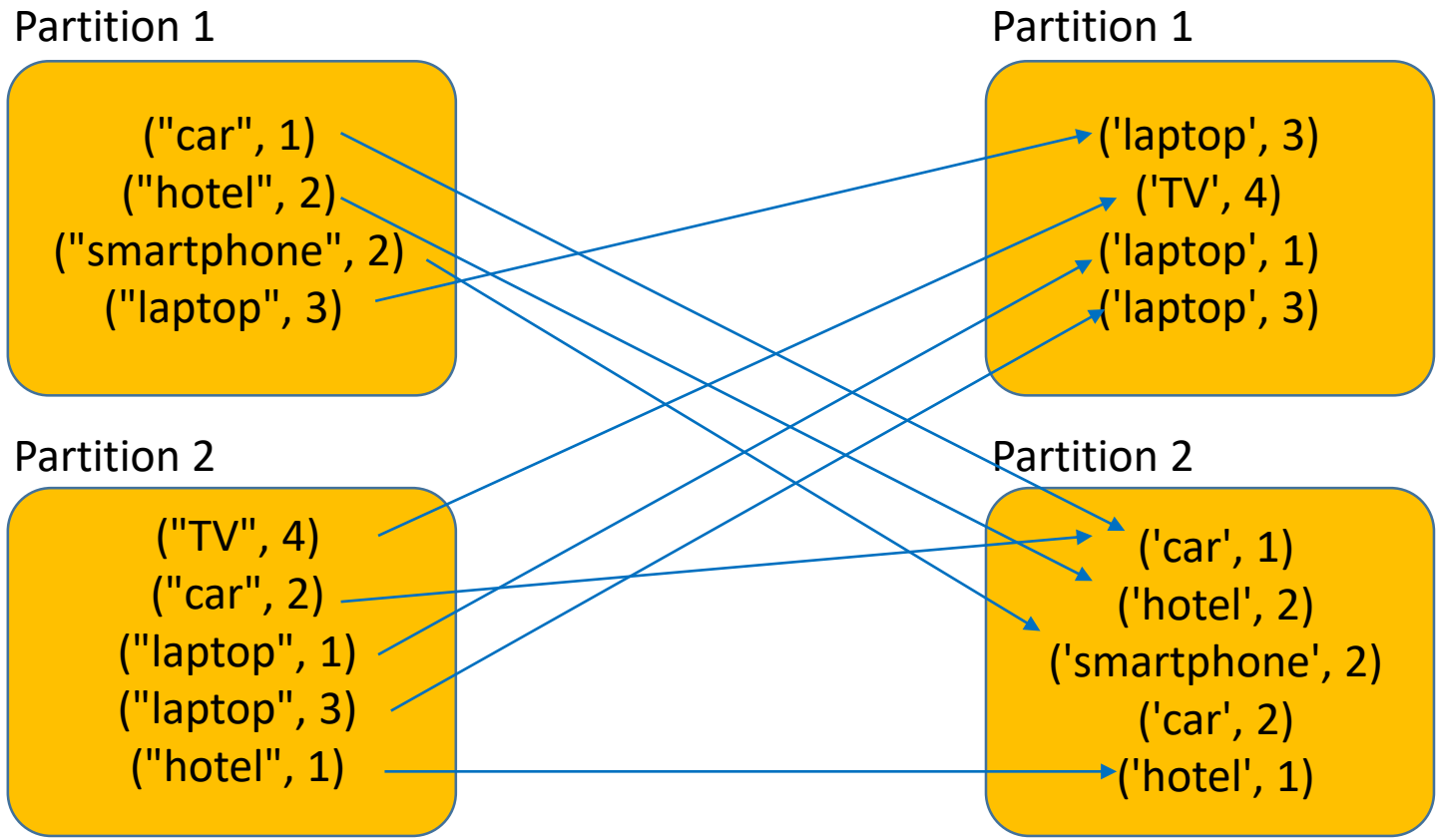
partitionBy(*numPartitions*, *partitionFunc=<function portable_hash at 0x7f7534d08d70>*)
Возвращает перетасованное RDD с использованием функции partitioner

coalesce(*numPartitions*, *shuffle=False*)
Возвращает новую RDD с уменьшенным количеством partitions до *numPartitions*

repartition(*numPartitions*)
Возвращает новую RDD с количеством partitions равным *numPartitions*

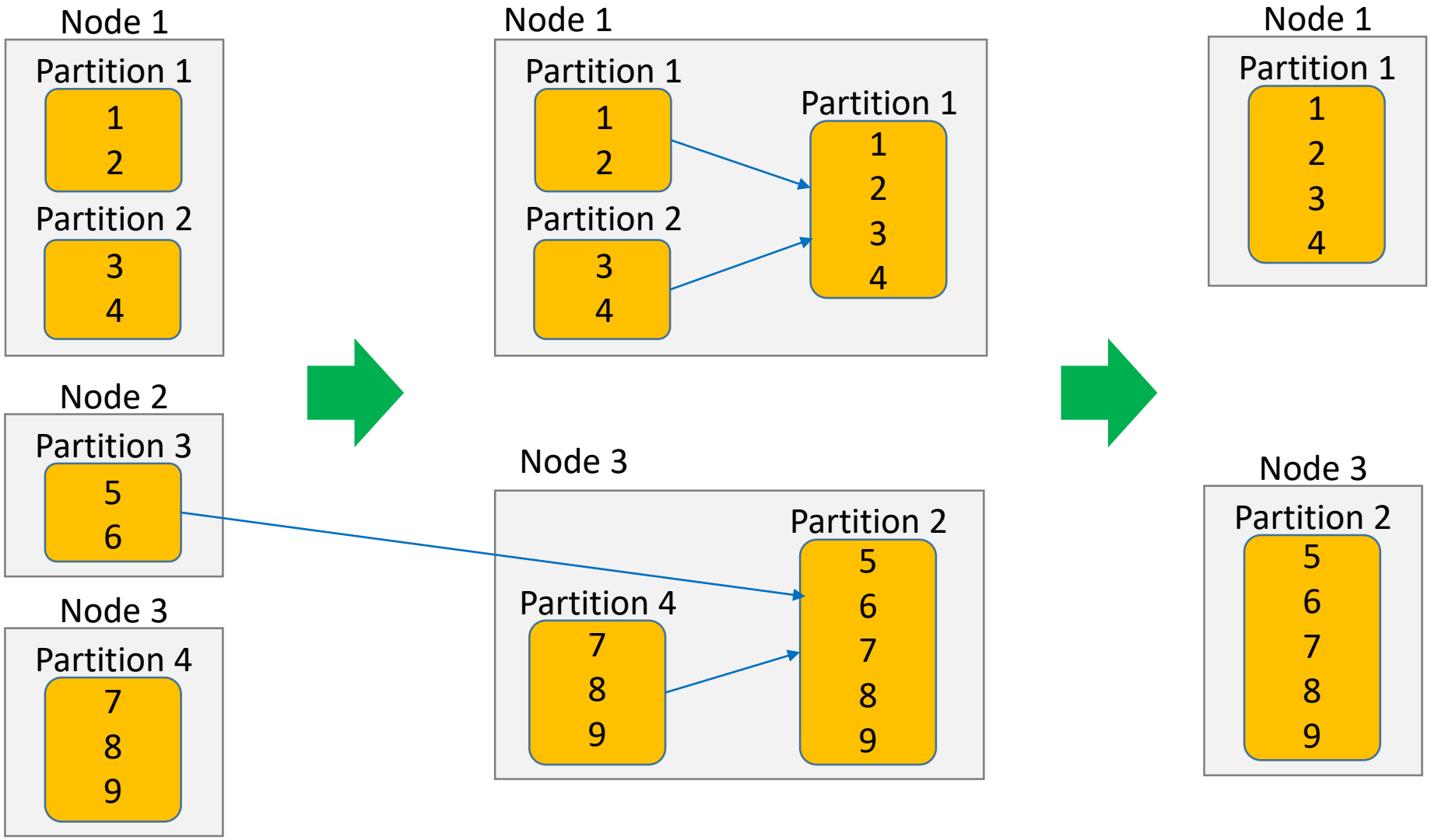
Пример PartitionBy

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



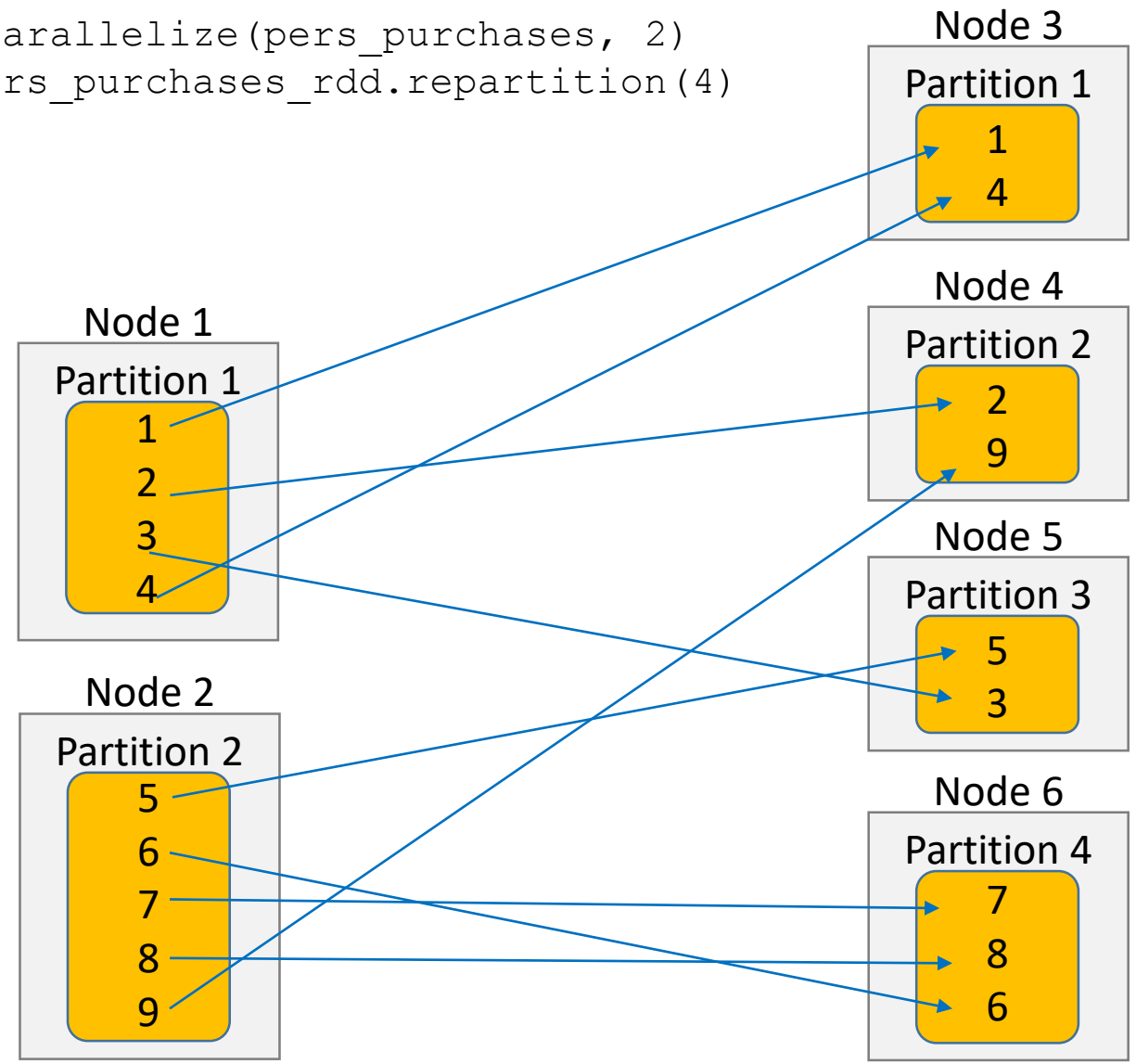
Пример Coalesce

```
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)
```



Операции агрегирования

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>)

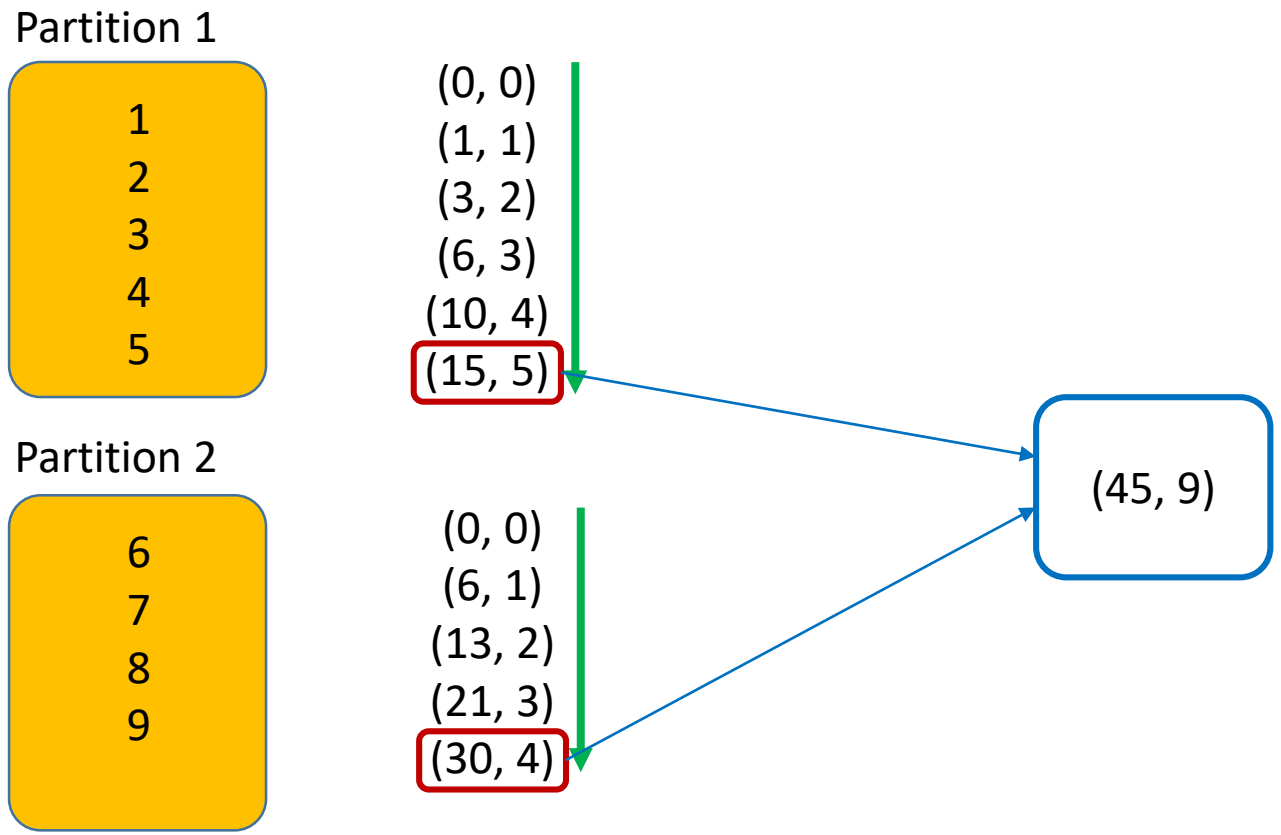
Агрегирует значения по ключу

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

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

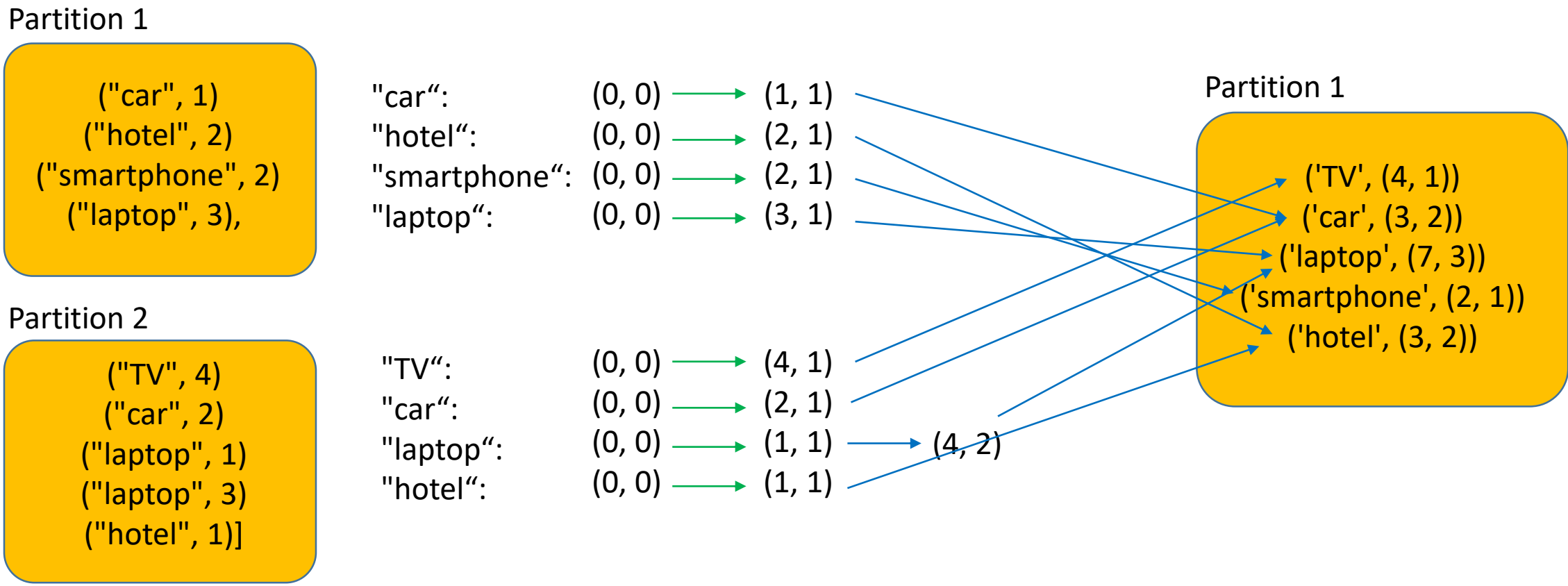
Пример Aggregate

```
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])))
```



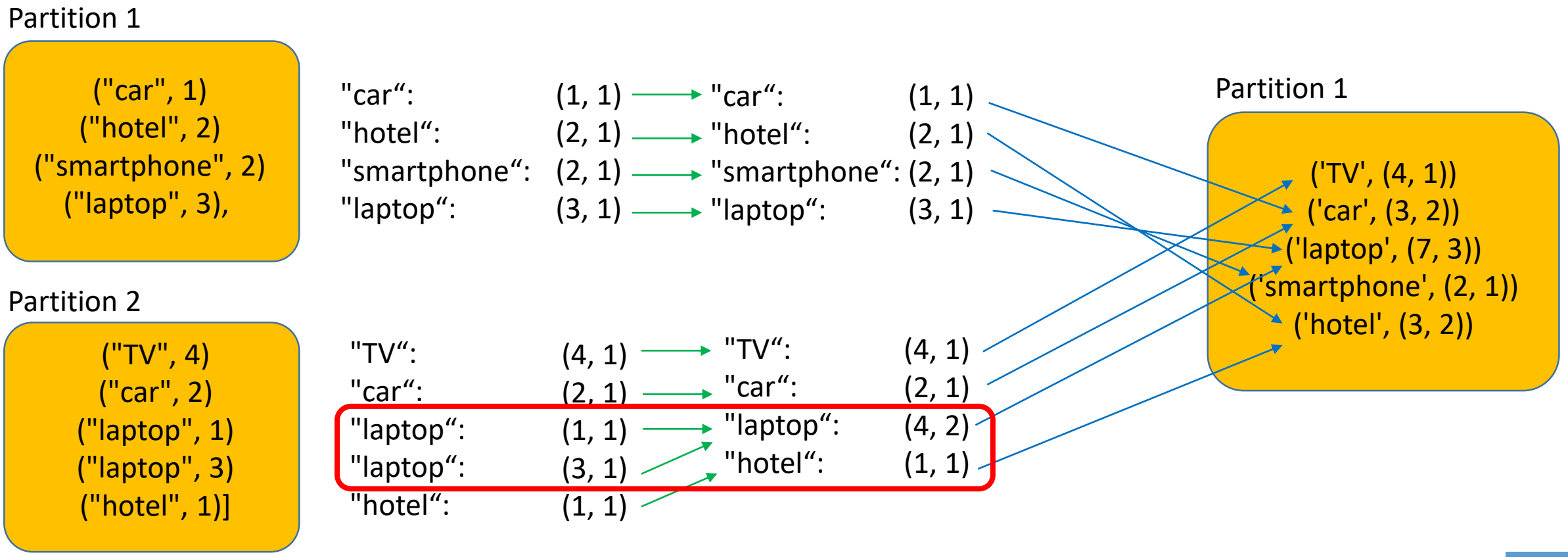
Пример AggregateByKey

```
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])))
```



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)
```



Кэширование RDD

Persist vs Cache

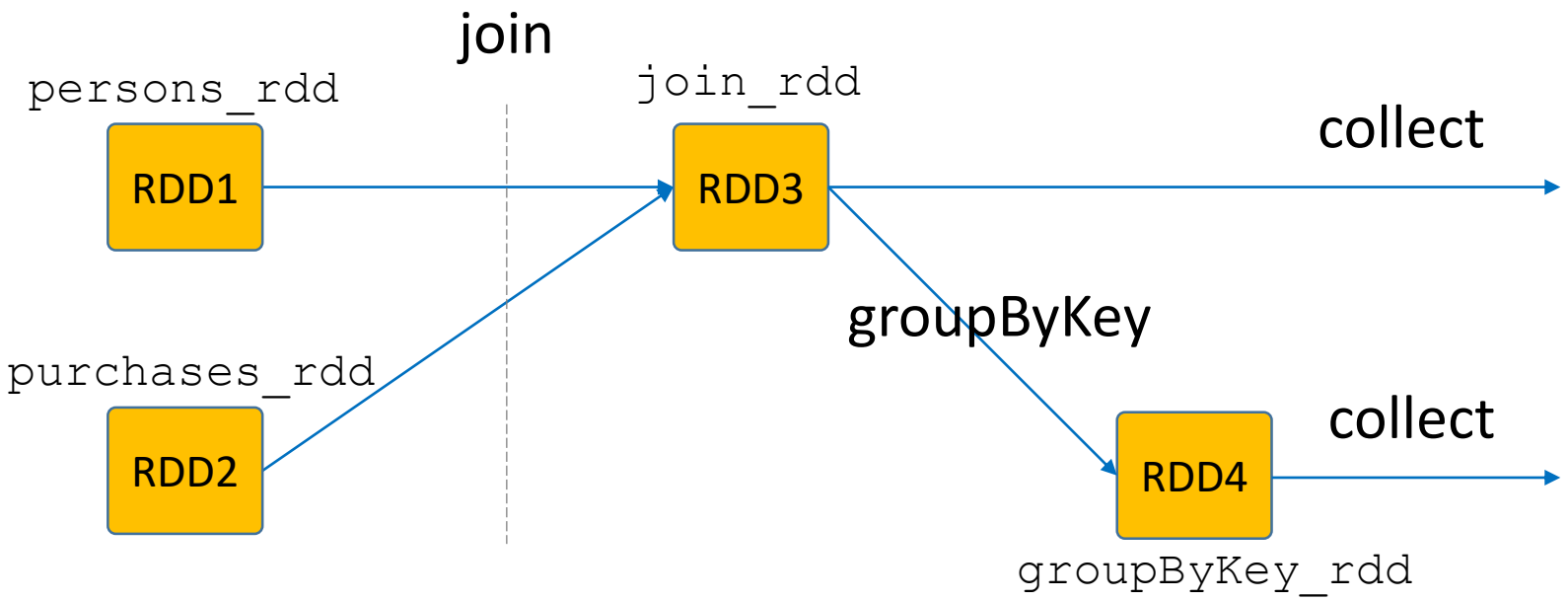
- Применяется для сохранения RDD в памяти и повторного его использования другими работами (job) одного приложения
- Для кэша есть две команды: cache (сохраняет в оперативной памяти) и persist (место и форму хранение можно задать, см. след. слайд)
- Кеш – отказоустойчив – если некоторая части RDD будет 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.

Persist. Уровни хранения

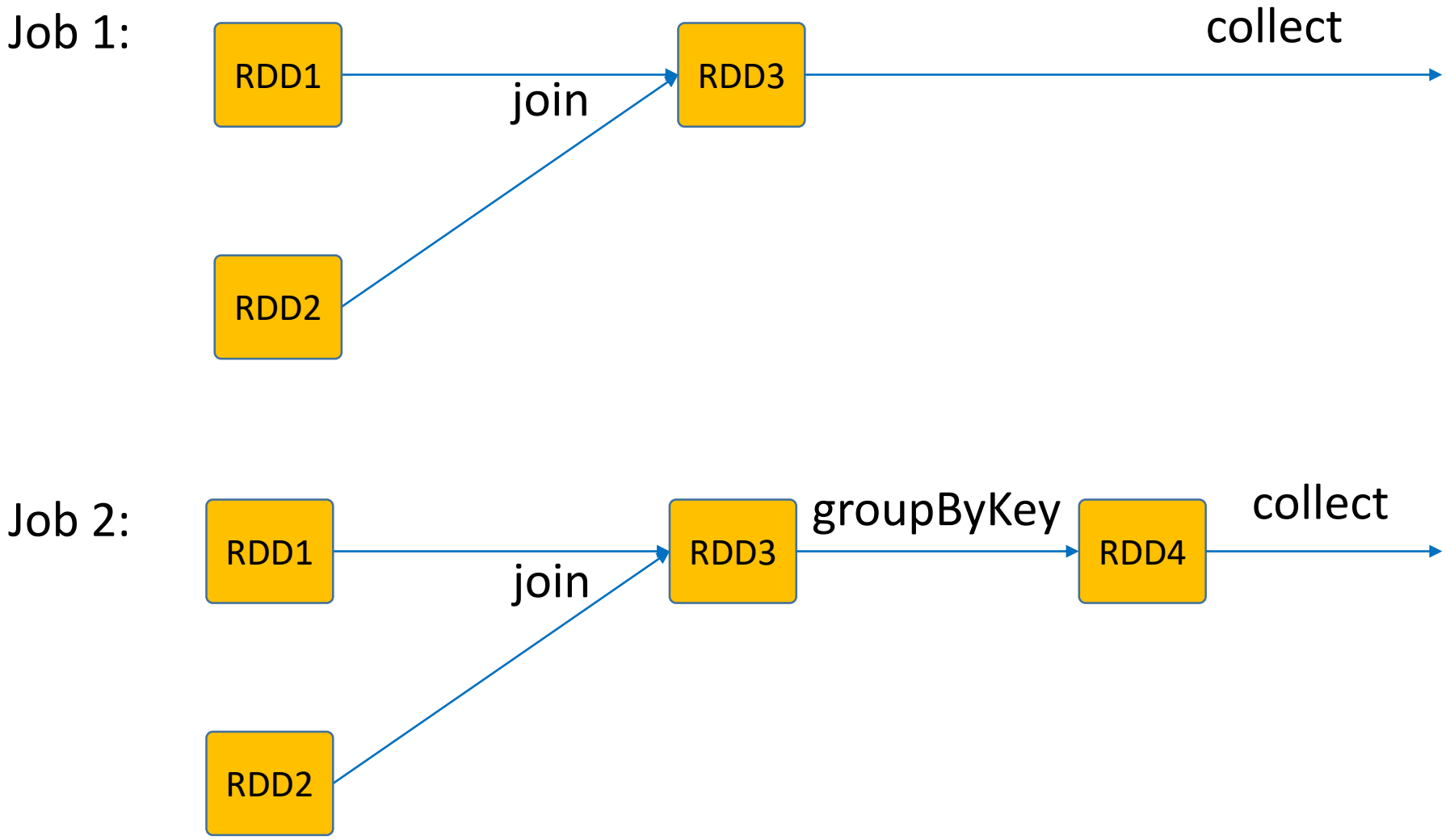
Уровень хранения	Определение
MEMORY_ONLY (по умолчанию)	Сохраняет RDD в виде несериализованных Java объектов в JVM. Если данные не помещаются в память, некоторые части будут пересчитывать на лету при каждом обращении
MEMORY_AND_DISK	Сохраняет RDD в виде несериализованных Java объектов в JVM. Если данные 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()
```



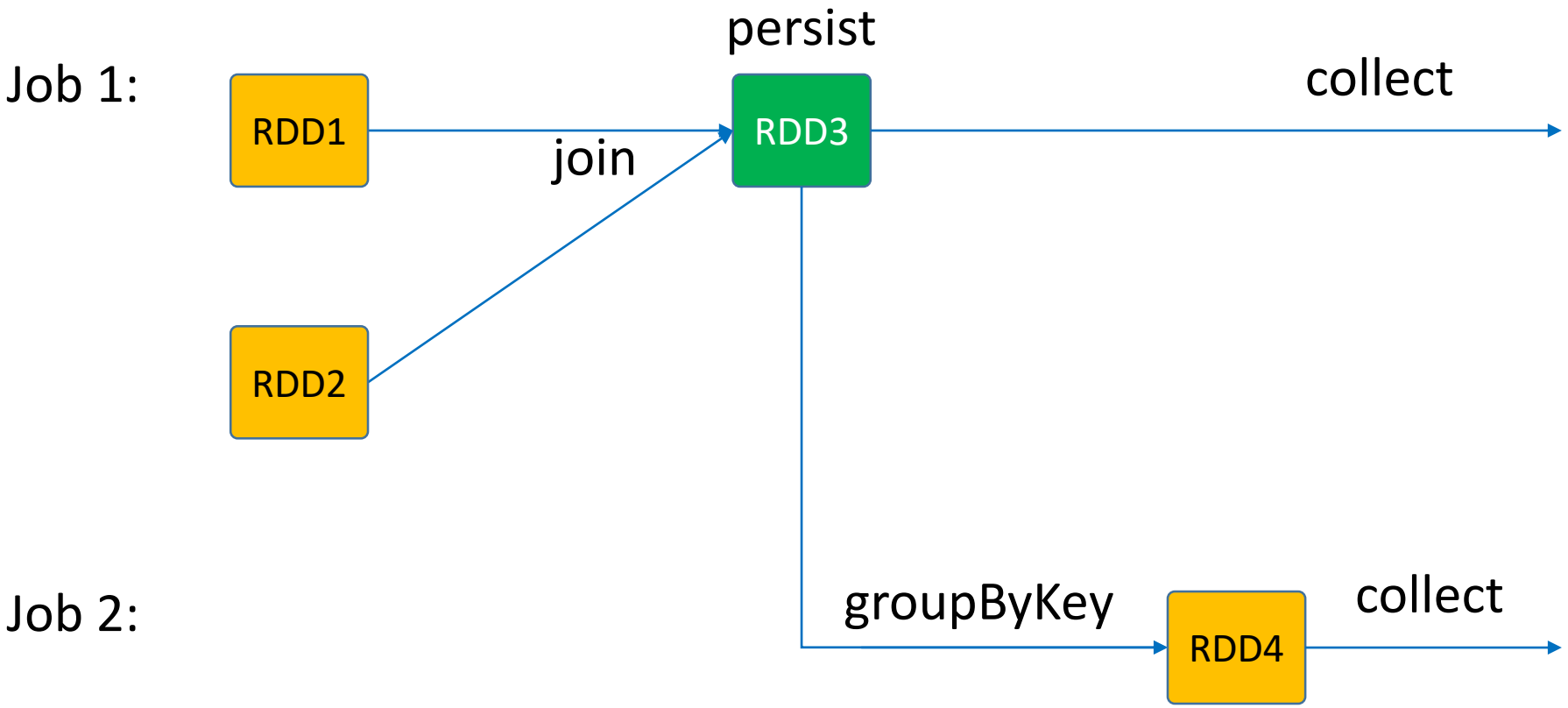
Пример, работы без persist()



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()

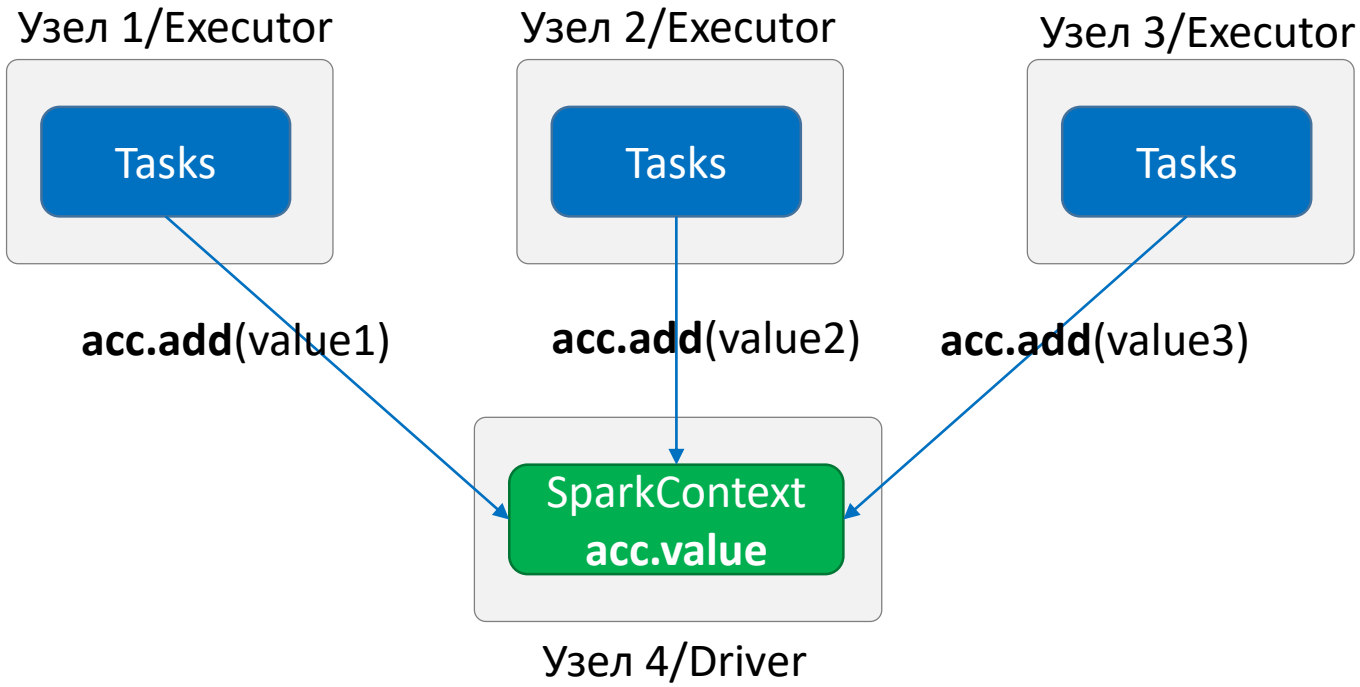


Переменные
Accumulator и Broadcast

Accumulator

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

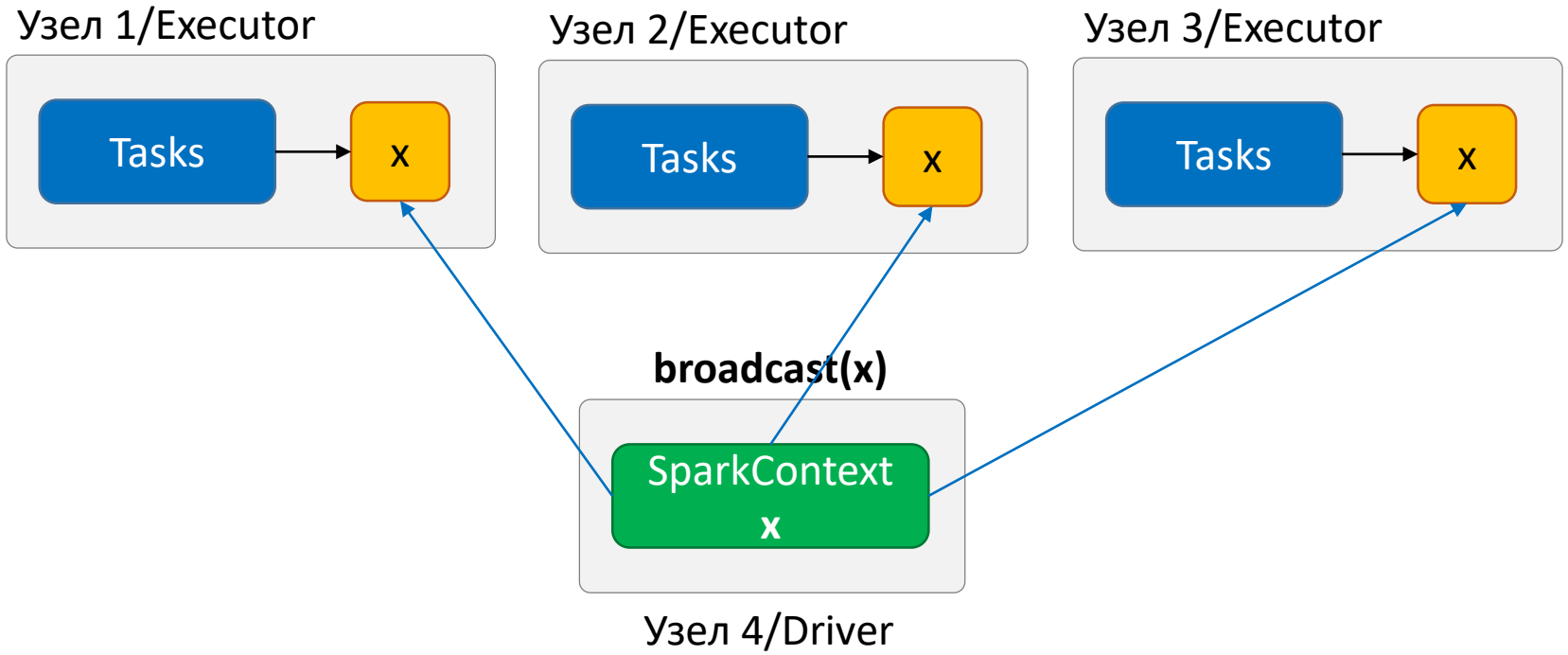
Значение **accumulator**а может прочитать только **driver**



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**)



Источники

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

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

RDD — Resilient Distributed Dataset (e-book)

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

Introduction to Core Spark Concepts (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)

Spark (github source code)

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)