LSML #4
SPARK DATAFRAME API
JOIN И ИХ ТИПЫ ПЛАНЫ ЗАПРОСА В SPARK CATALYST

Обсудим следующие темы

- Spark DataFrame Api — более удобная абстракция для работы с данными, в сравнении с RDD
- Виды join и их физическую реализацию на движке Spark
- Планы ваших сформированных запросов и как их стоит читать
- Оптимизатор запросов Catalyst
- Разберем на практике эти аспекты работы со Spark

Напомню про Spark RDD API

Абстракция RDD — resilient distributed dataset

- Восстанавливаемый распределенный набор данных
- Входы операций должны быть RDD
- Все промежуточные результаты — также RDD
- Вокруг коллекции формируется DAG
  (ацикличный направленный граф) вычислений

Напомню про Spark RDD API

Абстракция RDD — resilient distributed dataset

- Восстанавливаемый распределенный набор данных
- Входы операций должны быть RDD
- Все промежуточные результаты — также RDD
- Вокруг коллекции формируется DAG
  (ацикличный направленный граф) вычислений

Как можно сформировать RDD

- Из файлов из хранилища (прим. Данные из HDFS)
- Подать на вход коллекцию данных из Python (прим. список, словарь)
- Трансформациями из другого RDD

Напомню про Spark RDD API

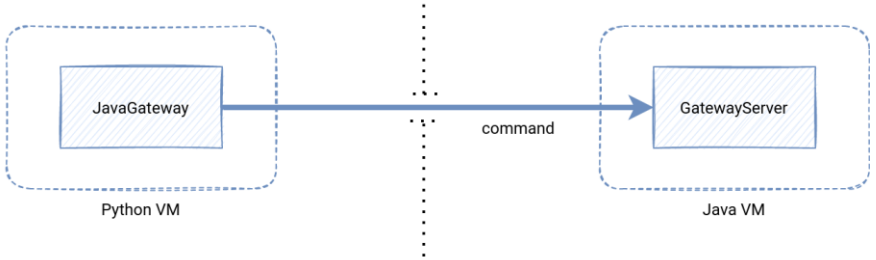
- Вычисления над RDD — применение операций вида «map» и «reduce» с постоянным использованием интерфейса lambda ф-ий

- Важный момент: все реализации нужных нам вычислений мы сначала типизируем на Python, после чего они интерпретируются в Java код через Py4J для дальнейшей обработке запроса в JVM.

- Это преобразование кода тоже занимает время

```
rdd = sc.parallelize(["Moscow", "Paris", "Madrid", "London"])

#пример: вычисляем число символов в коллекции
count = rdd.map(lambda x: len(x)).reduce(lambda x,y: x + y)
```



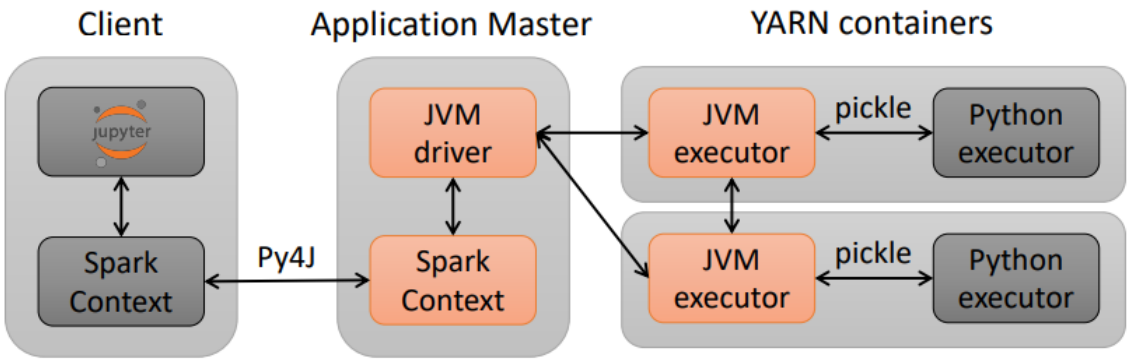
Py4J simplified architecture

Напомню про Spark RDD API

- Вычисления над RDD — применение операций вида «map» и «reduce» с постоянным использованием интерфейса lambda ф-ий

- Важный момент: все реализации нужных нам вычислений мы сначала типизируем на Python, после чего они интерпретируются в Java код через Py4J для дальнейшей обработке запроса в JVM.

- Это преобразование кода тоже занимает время



Решение оптимизации — использовать DataFrame API

Что из себя представляет DataFrame API

- Абстракция над RDD с определением колоночной структуры данных
- Методы полностью реализованы на JVM, пользователь лишь обращается к ним через обертку на Python
- Поддерживает работу через полноценные SQL запросы
- Трансформации над этой абстракцией также поддерживают «ленивую» и неизменяемую структуру
- Трансформации также создают DAG со своим набором вычислений

Фактически абстракция состоит из следующих типов

- схема pyspsark.sql.StructType
- колонки pyspark.sql.Column
- данные pyspark.sql.Row

Чтение DataFrame из источников

- Общая структура чтения данных из источника, где:

  - datasource_type — тип источника (orc, parquet, json и т.д.)
  - datasource_options — опции для работы с источником (креды доступа, адреса для подключения, параметры чтения и т.д.)
  - object_name — имя объекта/таблицы/топика

```
// Пример кода чтения в DataFrame
df = (
            spark
            .read
            .format(datasource_type)
            .option(datasource_options)
            .load(object_name)
)
```

- DataframeReader
- Выводит схему данных с выбранными типами (на свое усмотрение)
- является ленивой трансформацией над RDD
- Возвращает абстракцию Dataframe, с которой мы и будем работать

Чтение DataFrame из источников

- Список поддерживаемых типов
  - Файлы — json, csv, orc, parquet
  - Базы данных — jdbc, hive, Cassandra
  - Стриминг данных — kafka

- Важно: для чтения данных в DataFrame из определенной
  базы данных вам необходимо добавить нужные зависимости в инициализацию спарк сессии. Чаще всего они представлены определенным набором .jar пакетов, внутри которых уже собраны специальные коннекторы, которыми вы должны воспользоваться.
- Или же они должны содержаться
  в java_classpath на драйвере и воркерах.

```
// Пример кода чтения csv файла в DataFrame
df = (
            spark
            .read
            .format(“csv”)
            .option(header=True,
                        inferSchema=True)
            .load(“/to_path/file.csv”)
)
```

Запись DataFrame в хранилище

- Общая структура записи данных в хранилище, где
  - datasource_type — тип источника (orc, parquet, json и т.д.)
  - datasource_options — опции для работы с источником (креды доступа, адреса для подключения, параметры чтения и т.д.)
  - savemode — режим записи данных (append, overwrite и т.д.)
  - object_name — имя объекта/таблицы/топика

```
// Пример кода
(
        spark
        .write
        .format(datasource_type)
        .options(datasource_options)
        .mode(savemode)
        .save(object_name)
)
```

- DataframeWriter
  - save — «действие» для DataFrame, триггерит запуск всех вычислений в графе
  - Возможна более тонкая настройка записи (сортировка данных, партиции, влияние на число физических конечных файлов)
  - Часто необходима более тонкая настройка формата конечных данных

Запись DataFrame в хранилище

Общая структура записи данных в хранилище, где

- datasource_type — тип источника (orc, parquet, json и т.д.)
- datasource_options — опции для работы с источником (креды доступа, адреса для подключения, параметры чтения и т.д.)
- savemode — режим записи данных (append, overwrite и т.д.)
- object_name — имя объекта/таблицы/топика

```
// Пример кода записи фрейма данных в orc формат
(
          spark
          .write
          .format(“orc”)
          .options({“compression”: “zlib”})
          .mode(“append”)
          .save(“/to_path/file.parquet”)
)
```

DataframeWriter

- save — «действие» для DataFrame,
  триггерит запуск всех вычислений в графе
- Возможна более тонкая настройка записи (сортировка данных, партиции, влияние на число физических конечных файлов)
- Часто необходима более тонкая
  настройка формата конечных данных

Работа с DataFrame через SQL

Ввод [16]:

```
sc = spark.sparkContext

test_data = [
{"name":"Moscow", "country":"Rossiya", "continent": "Europe", "population": 100_000_000},
{ "name":"Madrid", "country":"Spain" },
{ "name":"Paris", "country":"France", "continent": "Europe", "population" : 205_000_000},
{ "name":"Berlin", "country":"Germany", "continent": "Europe", "population": 140_000_008},
{ "name":"Barselona", "country":"Spain", "continent": "Europe" },
{ "name":"Cairo", "country":"Egypt", "continent": "Africa", "population": 45_000_001 },
{ "name":"Cairo", "country":"Egypt", "continent": "Africa", "population": 45_000_001 },
{ }
]

rdd = sc.parallelize(test_data)

df = spark.read.json(rdd).localCheckpoint()

df
```

Out[16]: `DataFrame[continent: string, country: string, name: string, population: bigint]`

- Над фреймом данных можно создать временную метку, обозначающую набор данных как таблицу, после чего можно работать с данными через полноценный SQL интерфейс,
если вы знакомы с его базовыми методами

Ввод [26]:

```
(
    df.createOrReplaceTempView("test_table"),

    spark.sql("""
    select country, continent

    from test_table

    where continent = 'Europe'

    """)
    .show(10, False)
)
```

```
+-------+---------+
|country|continent|
+-------+---------+
|Rossiya|Europe   |
|France |Europe   |
|Germany|Europe   |
|Spain  |Europe   |
+-------+---------+
```

ПОГОВОРИМ ПРО JOINЫ
И КАК ОНИ РАБОТАЮТ

Вспомним реализацию join из SQL

- Таблица a — выбор товаров у пользователей — user, product, etc.
- Таблица b — инфо о пользователях — user, city, etc.
- Хотим собрать инфу из таблиц вместе чтобы получить инфо о товарах по городам

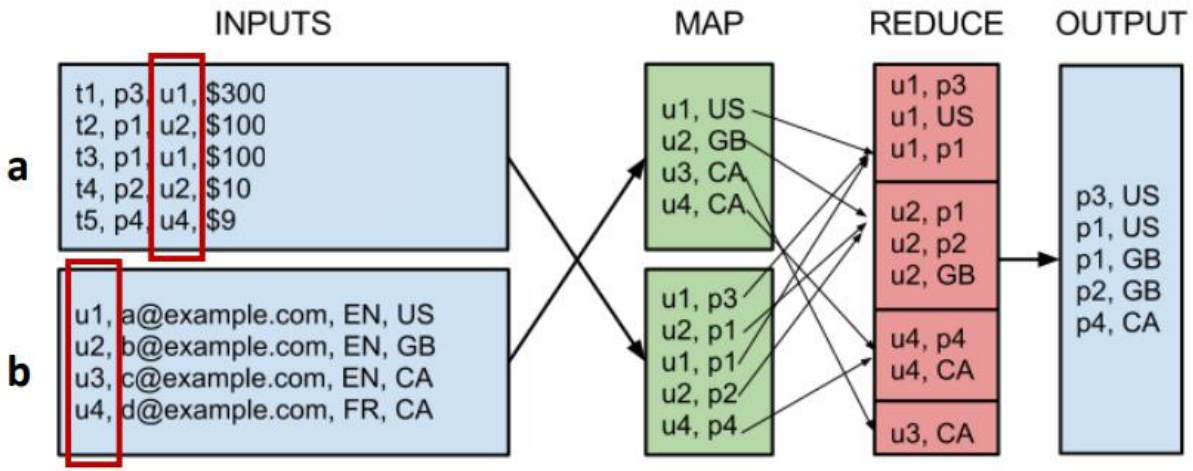
Что нужно сделать?

Вспомним реализацию join из SQL

- Таблица a — выбор товаров у пользователей — user, product, etc.
- Таблица b — инфо о пользователях — user, city, etc.
- Хотим собрать инфу из таблиц вместе чтобы получить инфо о товарах по городам

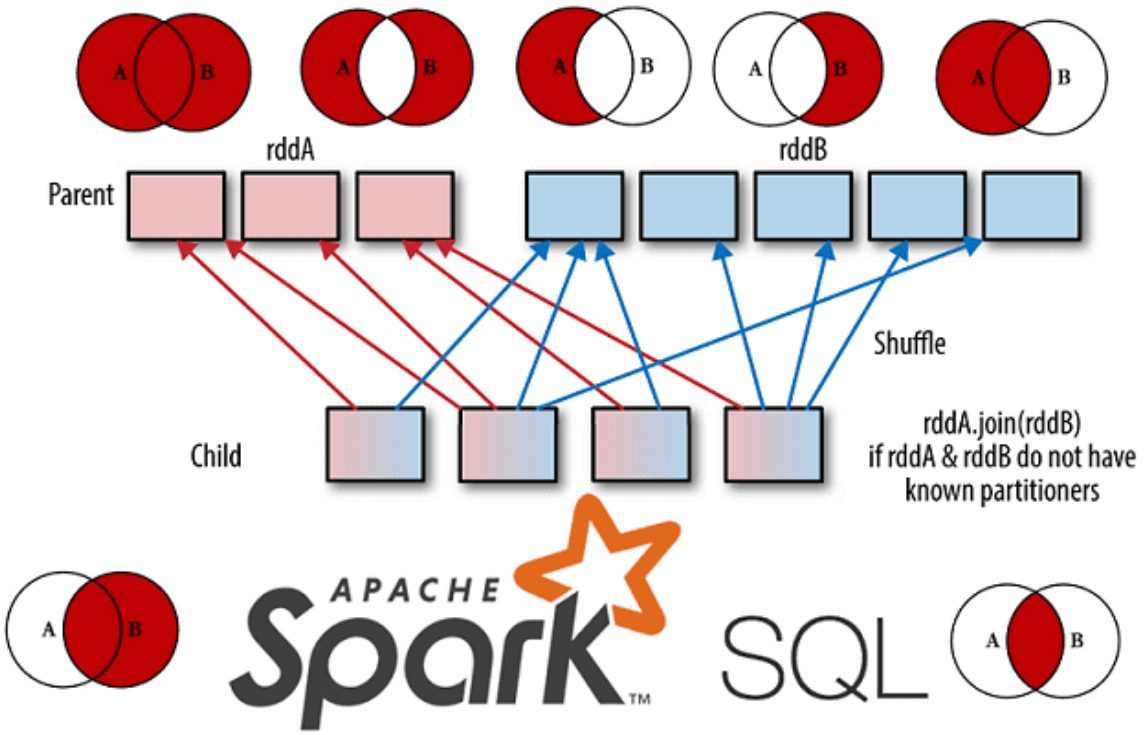
Что нужно сделать?

Вспомним реализацию join из SQL



- Реализация похожего join на map reduce

SQL join на движке Spark



- Join на движке Spark реализован по похожей логике и включает в себя те же логические возможности
  - Inner join — ищем логические пересечение по ключам
  - left/right join — только левое/правое включение
  - full (outer) join — полное объединение фреймов данных
  - left anti / right anti — данные без левого или правого включения

Join на движке Spark

- Join в Spark вызывается через следующий метод, где вы связываете два фрейма данных

- Влияющие на процесс параметры

  - on — условие джойна, включает либо перечисление колонок ключей, либо более сложное условие соотв-вия ключей
  - how — логическое условие джойна: inner, full, left, left anti, etc.

- Важно: чем сложнее ваша логика объединения двух фреймов данных, тем более сложным спарк выберет подход для реализации этого joina

```
// Пример кода
left       #левый датафрейм данных
right      #правый датафрейм данных
joined = left.join(
           right,
           on=[ “column” ],
           how= “inner”
)
# Другое возможное условие
from pyspark.sql.functions import col

joined = left.alias("left") \
               .join(right.alias("right"),
                     col("left.column") ==
col("right.column"), 'inner')
```

Виды возможных сортировок ключей

Движок Sparka, опираясь на вашу логику объединения данных, может выбрать один из следующих походов обработки join

BroadcastHashJoin

- соединение по равенству одного или большего набора ключей
- необходима операция broadcast над одним из фреймов

SortMergeJoin

- соединение по равенству одного или большего набора ключей
- необходима сортировка ключей по обоим фреймам

BroadcastNestedLoopJoin

- соединение по условию, отличному от равенства одного или более ключей
- необходима операция broadcast над одним из фреймов

CartesianProduct

- соединение по условию, отличному от равенства одного или более ключей
- формирование всех возможных пар ключей для проверки условия joina

Немного про broadcast

- Broadcast – метод, который рассылает датафрейм данных на все воркеры в полном его объеме
- Удобно для датафреймов небольшого размера
- Это «ленивая» трансформация, отправка данных на воркеры запускается только при триггере расчетов
- Данные рассылаются на воркеры один раз и могут быть использованы во многих операциях
- Так как датафрейм размещен на воркерах в полном объеме, шаффлы над его данными больше проводить не нужно, это сильно экономит время в трансформациях над ним

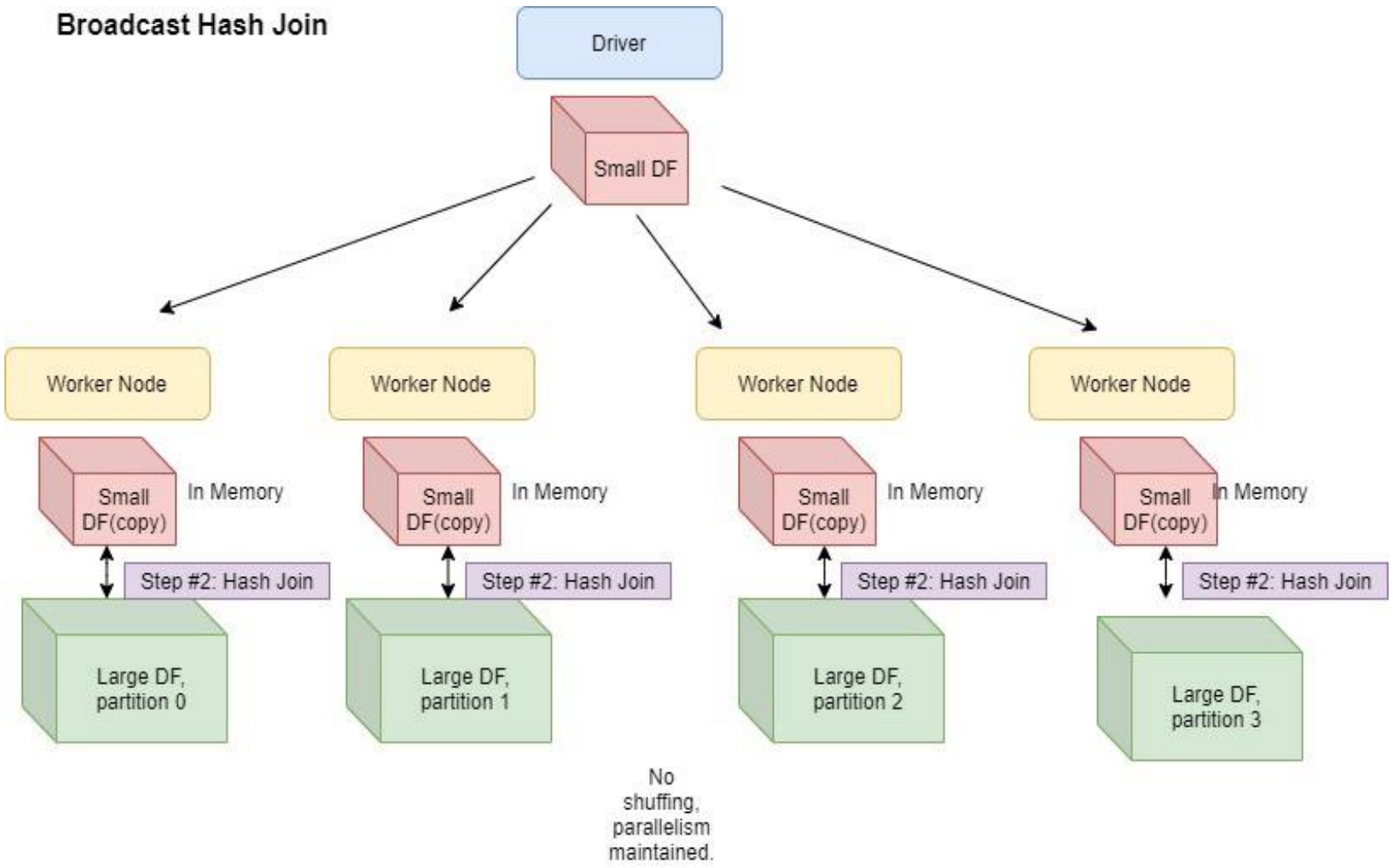
```
// Пример кода
from pyspark.sql.functions import broadcast

broad_right = broadcast(right)
```

BroadcastHashJoin

- Выполняется, когда условие joina — равенство по набору ключей
- Выполняется, когда один из датасетов можно полностью разместить на воркерах
- Составляет hash map из малого фрейма данных, где ключ это кортеж из набора ключей из условия joina
- Итерируется по партициям большего фрейма данных и проверяет наличие ключей в hash map
- Используется автоматически под капотом или из-за явного вызова broadcast над малым фреймом данных

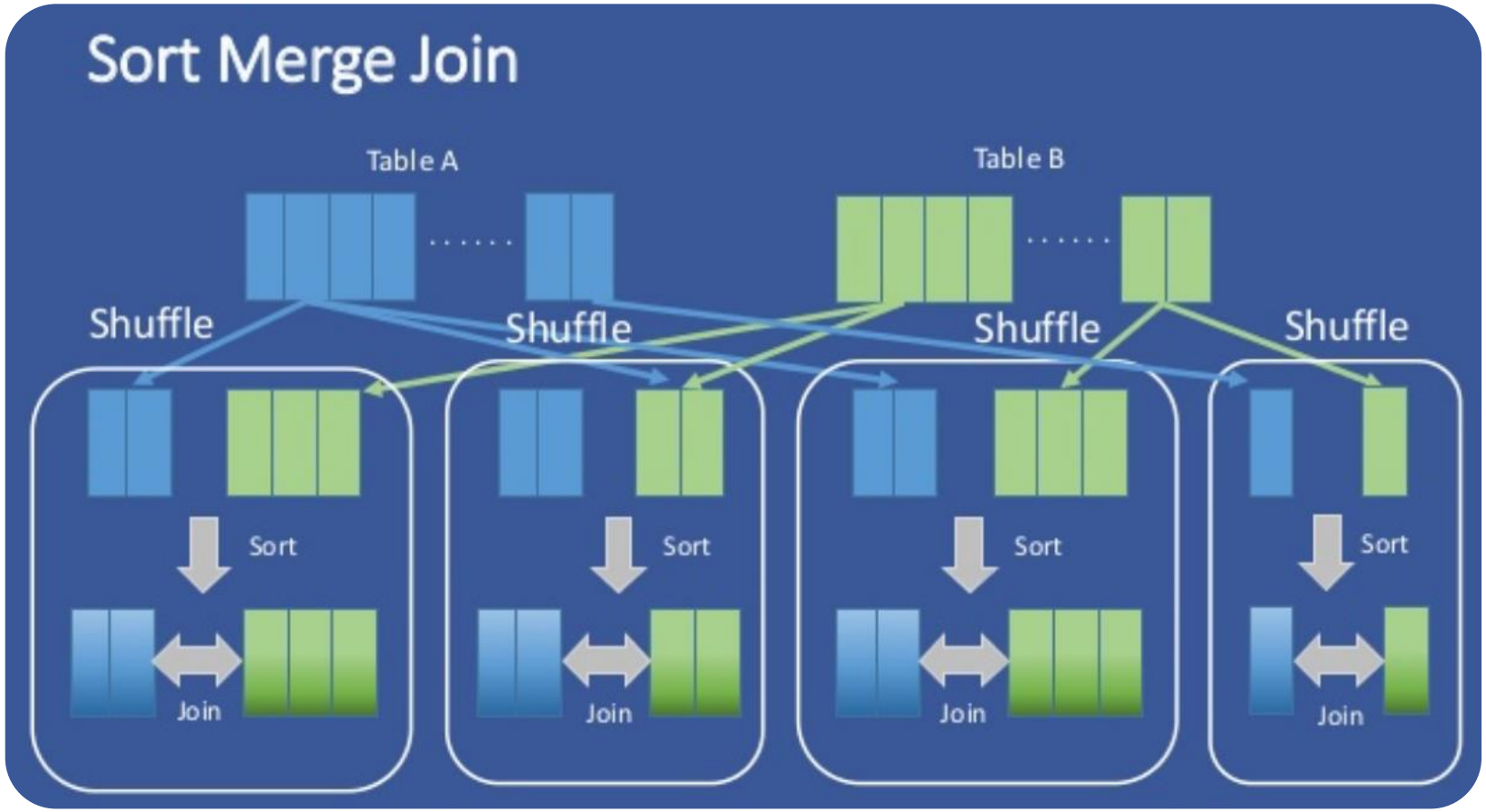
BroadcastHashJoin



SortMergeJoin

- Выполняется, когда ключи в обоих фреймах данных возможно отсортировать для выполнения условия
- Запускает шаффл ключей в обоих фреймах данных, для их последующего распределения на воркеры
- Сортирует партиции каждого из фрейма данных по набору ключей из условия
- Через сравнения значений ключей из обоих частей фреймов данных (отсортированные и уже на конкретном воркере) и соединяет данные по одинаковым значениям ключей

SortMergeJoin

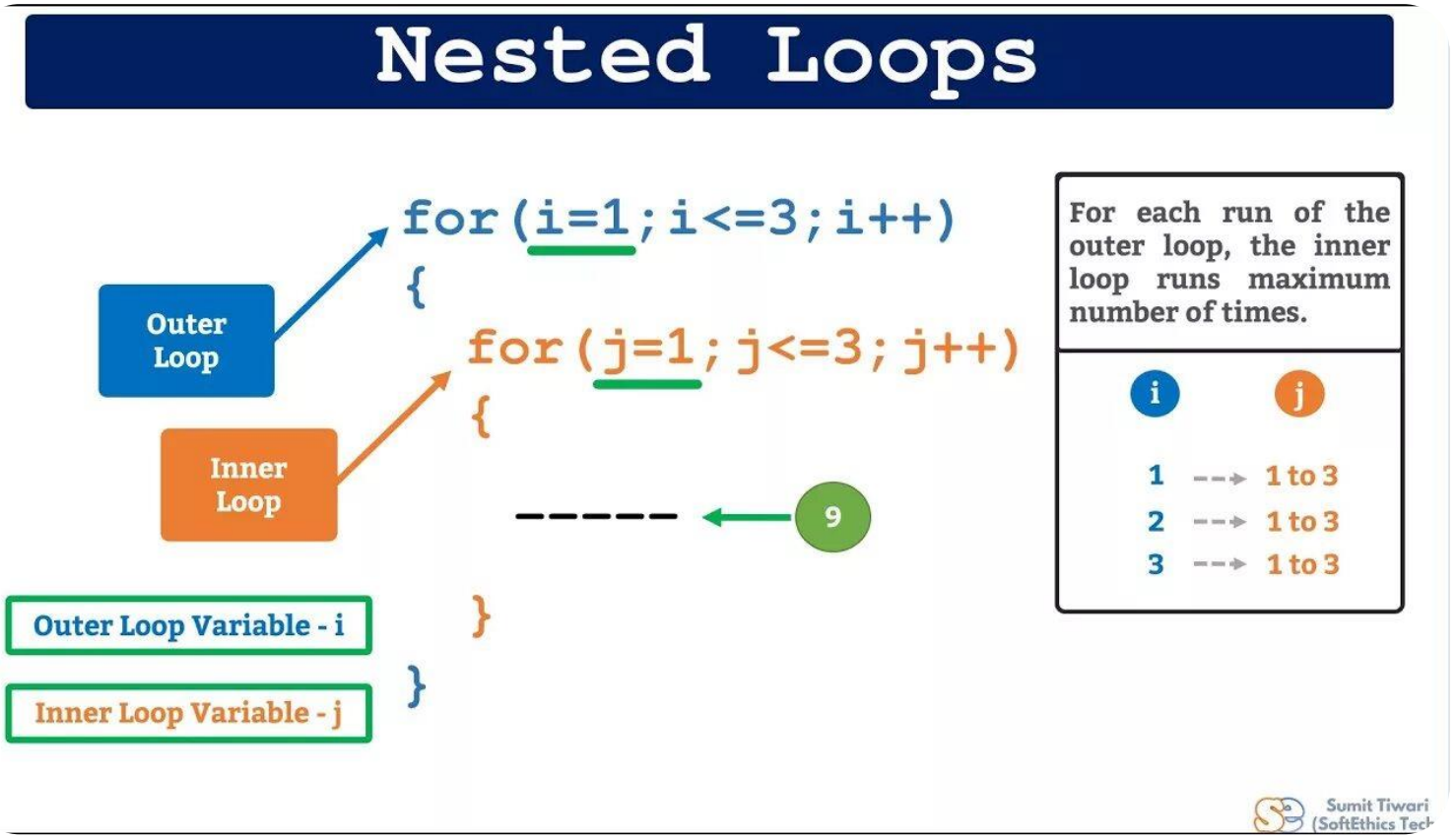


BroadcastNestedLoopJoin

- Выполняется, когда один из фреймов данных можно полностью разместить на воркерах
- Распределяет партиции большего фрейма данных по воркерам
- Проходится вложенным циклом по партициям большего фрейма данных на воркерах, и по копии меньшего датасета проверят условие для joina данных (hash map нет из-за сложности условия)
- Может быть выбран автоматически или при явном использовании broadcast на малом фрейме данных

BroadcastNestedLoopJoin

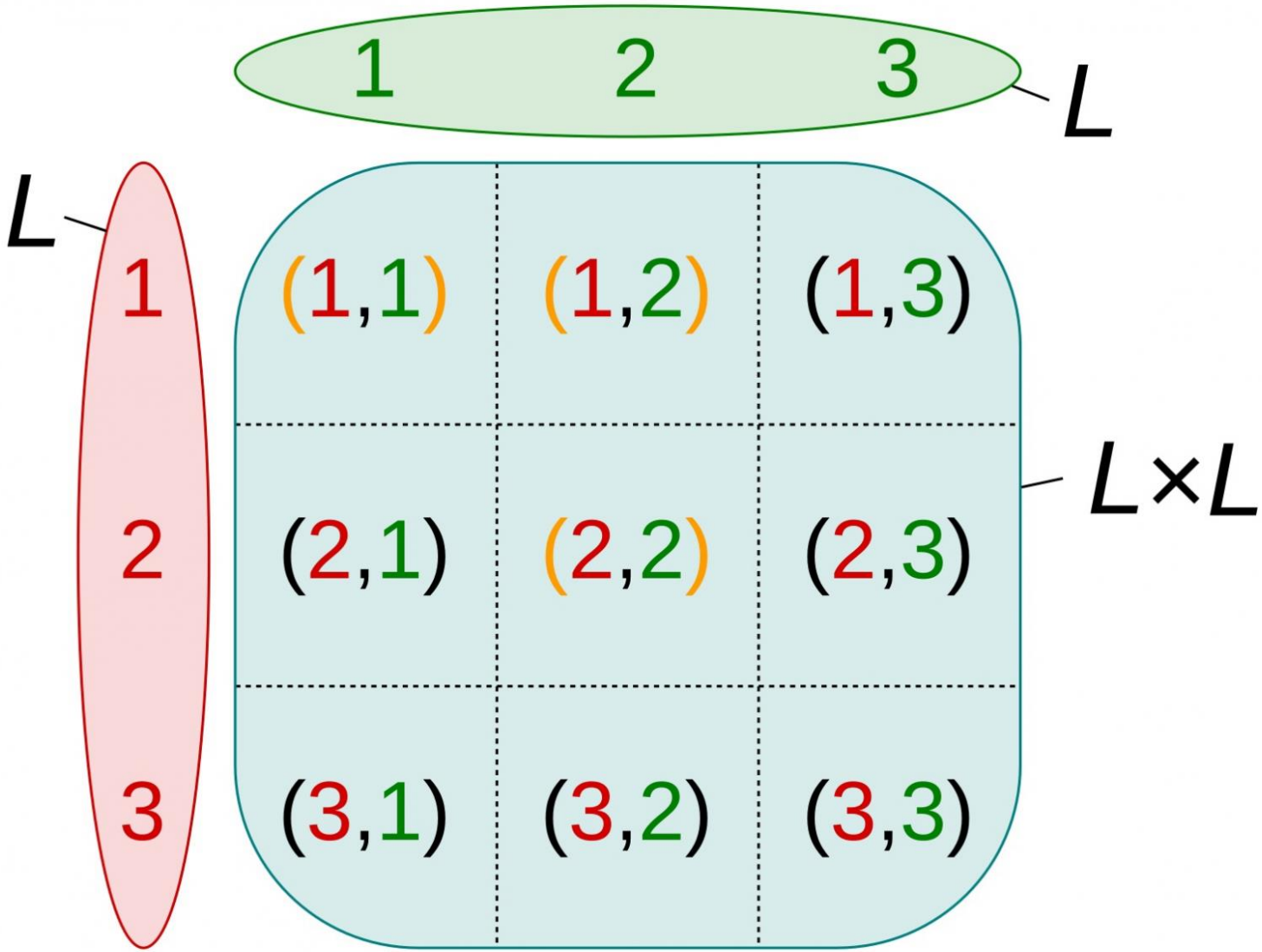
- Более понятной картинке в интернете
  не нашлось, поэтому вот вам интерпретация вложенного цикла для этого джйона



CartesianProduct

- Создает пары из каждой партиции первого фрейма данных с каждой партицией второго фрейма данных
- Созданный набор пар отправляет на воркер, где проверяет их на условие соединения
- В процессе создается M x N партиций
- Проверяется очень много пар, поэтому работает сильно дольше остальных методов

CartesianProduct



Рекомендации по оптимизации операций join

- Фильтровать данные в фреймах данных до нужных наборов колонок и строк
- Стараться использовать joinы с условием равенства значений на ключах
- Если можно перейти от сложного условия joina к поиску равенства ключей, но данных в joine будет использоваться больше, то лучше выбрать этот подход. Поиск равенства ключей всегда лучше оптимизирован в расчетах
- Если один из фреймов данных небольшого размера — стараться использовать на нем broadcast
- Дополнительно думайте над вашим условием joina — чем оно проще вычисляется, тем лучше, возможно, над данными стоит провести еще какие-то вычисления, чтобы упростить этот процесс.
- CartesianProduct из-за своей реализации может не добежать до конечного результата и начать падать с ошибками по нехватки памяти

Как можно посмотреть на собранный граф вычислений без явно триггера расчетов?

- Собирая граф вычислений вокруг фрейма данных, бывает достаточно полезно посмотреть на уже собранные расчеты, набор трансформаций.
  Это может помочь выявить проблемы на этапе уже имеющихся трансформаций, или же увидеть, какой из методов джойна был выбран для вашего условия.
- При этом хотелось бы, чтоб эти вычисления не запускались на расчет и не отнимали у вас ваше время

- Это можно сделать через метод «.explain()», обратившись к фрейму данных, вокруг которого у вас уже собраны вычисления

Как можно посмотреть на собранный граф вычислений без явно триггера расчетов?

- Рассмотрим следующий пример входных данных для удобного анализа

Ввод [16]:

```
sc = spark.sparkContext

test_data = [
{"name":"Moscow", "country":"Rossiya", "continent": "Europe", "population": 100_000_000},
{ "name":"Madrid", "country":"Spain" },
{ "name":"Paris", "country":"France", "continent": "Europe", "population" : 205_000_000},
{ "name":"Berlin", "country":"Germany", "continent": "Europe", "population": 140_000_008},
{ "name":"Barselona", "country":"Spain", "continent": "Europe" },
{ "name":"Cairo", "country":"Egypt", "continent": "Africa", "population": 45_000_001 },
{ "name":"Cairo", "country":"Egypt", "continent": "Africa", "population": 45_000_001 },
{ }
]

rdd = sc.parallelize(test_data)

df = spark.read.json(rdd).localCheckpoint()

df
```

Out[16]:

```
DataFrame[continent: string, country: string, name: string, population: bigint]
```

Как можно посмотреть на собранный граф вычислений без явно триггера расчетов?

- От метода «select» мы получаем следующий набор планов расчета:
- **Parsed Logical Plan** – интерпретация ваших расчетов на уровне логических методов, которые мы вызываем через обертку в Python
- **Analyzed Logical Plan** – Анализ логического плана, на этом этапе движок проверяет, что применяемые вами методы трансформаций соответсвуют с типами данных колонок, к которым они могут быть применимы
- **Optimized Logical Plan** – план вычислений с оптимизациями движка через Catalyst, здесь может быть изменен порядок созданных вами вычислений, с учетом их времени работы и оптимизаций в движке
- **Physical Plan** – план сводится до явных физических методов, используемых под капотом движка Spark, фактически тут отображены более сложные подобия map/reduce задач

```
Ввод [17]:  1 df.select("continent", "country").explain(True)

== Parsed Logical Plan ==
'Project ['continent, 'country]
+- LogicalRDD [continent#99, country#100, name#101, population#102L], false

== Analyzed Logical Plan ==
continent: string, country: string
Project [continent#99, country#100]
+- LogicalRDD [continent#99, country#100, name#101, population#102L], false

== Optimized Logical Plan ==
Project [continent#99, country#100]
+- LogicalRDD [continent#99, country#100, name#101, population#102L], false

== Physical Plan ==
*(1) Project [continent#99, country#100]
+- *(1) Scan ExistingRDD[continent#99,country#100,name#101,population#102L]
```

Как можно посмотреть на собранный граф вычислений без явно триггера расчетов?

- Изменение состава плана с добавлением метода «.filter()»

Ввод [18]:

```
(
    df
    .select("continent", "country")
    .filter(F.col("name") == "Moscow")
    .explain(True)
)
```

```
== Parsed Logical Plan ==
'Filter ('name = Moscow)
+- Project [continent#99, country#100]
   +- LogicalRDD [continent#99, country#100, name#101, population#102L], false

== Analyzed Logical Plan ==
continent: string, country: string
Project [continent#99, country#100]
+- Filter (name#101 = Moscow)
   +- Project [continent#99, country#100, name#101]
      +- LogicalRDD [continent#99, country#100, name#101, population#102L], false

== Optimized Logical Plan ==
Project [continent#99, country#100]
+- Filter (isnotnull(name#101) AND (name#101 = Moscow))
   +- LogicalRDD [continent#99, country#100, name#101, population#102L], false

== Physical Plan ==
*(1) Project [continent#99, country#100]
+- *(1) Filter (isnotnull(name#101) AND (name#101 = Moscow))
   +- *(1) Scan ExistingRDD[continent#99,country#100,name#101,population#102L]
```

Как можно посмотреть на собранный граф вычислений без явно триггера расчетов?

- Изменение плана расчетов при работе с агрегацией данных, через «.groupBy()»

Ввод [21]:

```
res = (
    df
    .groupBy("continent")
    .agg(F.count("*"), F.sum(F.col("population")))
)

res.show()
res.explain(True)
```

```
+---------+--------+---------------+
|continent|count(1)|sum(population)|
+---------+--------+---------------+
|     null|       2|           null|
|   Europe|       4|      445000008|
|   Africa|       2|       90000002|
+---------+--------+---------------+

== Parsed Logical Plan ==
'Aggregate ['continent], ['continent, count(1) AS count(1)#154L, sum('population) AS sum(population)#155]
+- LogicalRDD [continent#99, country#100, name#101, population#102L], false

== Analyzed Logical Plan ==
continent: string, count(1): bigint, sum(population): bigint
Aggregate [continent#99], [continent#99, count(1) AS count(1)#154L, sum(population#102L) AS sum(population)#155L]
+- LogicalRDD [continent#99, country#100, name#101, population#102L], false

== Optimized Logical Plan ==
Aggregate [continent#99], [continent#99, count(1) AS count(1)#154L, sum(population#102L) AS sum(population)#155L]
+- Project [continent#99, population#102L]
   +- LogicalRDD [continent#99, country#100, name#101, population#102L], false

== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- HashAggregate(keys=[continent#99], functions=[count(1), sum(population#102L)], output=[continent#99, count(1)#154L, sum(population)#155L])
   +- Exchange hashpartitioning(continent#99, 400), ENSURE_REQUIREMENTS, [plan_id=228]
      +- HashAggregate(keys=[continent#99], functions=[partial_count(1), partial_sum(population#102L)], output=[continent#99, count#170L, sum#171L])
         +- Project [continent#99, population#102L]
            +- Scan ExistingRDD[continent#99,country#100,name#101,population#102L]
```

Как можно посмотреть на собранный граф вычислений без явно триггера расчетов?

- Изменение плана расчетов при работе с джойном фреймов данных через «.join()»

```
== Parsed Logical Plan ==
'Join UsingJoin(Inner,Buffer(continent))
:- Aggregate [continent#99], [continent#99, count(1) AS count(1)#154L, sum(population#102L) AS sum(population)#155L]
:  +- LogicalRDD [continent#99, country#100, name#101, population#102L], false
+- LogicalRDD [continent#187, country#188, name#189, population#190L], false

== Analyzed Logical Plan ==
continent: string, count(1): bigint, sum(population): bigint, country: string, name: string, population: bigint
Project [continent#99, count(1)#154L, sum(population)#155L, country#188, name#189, population#190L]
+- Join Inner, (continent#99 = continent#187)
   :- Aggregate [continent#99], [continent#99, count(1) AS count(1)#154L, sum(population#102L) AS sum(population)#155L]
   :  +- LogicalRDD [continent#99, country#100, name#101, population#102L], false
   +- LogicalRDD [continent#187, country#188, name#189, population#190L], false

== Optimized Logical Plan ==
Project [continent#99, count(1)#154L, sum(population)#155L, country#188, name#189, population#190L]
+- Join Inner, (continent#99 = continent#187)
   :- Aggregate [continent#99], [continent#99, count(1) AS count(1)#154L, sum(population#102L) AS sum(population)#155L]
   :  +- Project [continent#99, population#102L]
   :     +- Filter isnotnull(continent#99)
   :        +- LogicalRDD [continent#99, country#100, name#101, population#102L], false
   +- Filter isnotnull(continent#187)
      +- LogicalRDD [continent#187, country#188, name#189, population#190L], false

== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- Project [continent#99, count(1)#154L, sum(population)#155L, country#188, name#189, population#190L]
   +- SortMergeJoin [continent#99], [continent#187], Inner
      :- Sort [continent#99 ASC NULLS FIRST], false, 0
      :  +- HashAggregate(keys=[continent#99], functions=[count(1), sum(population#102L)], output=[continent#99, count(1)#154L, sum(population)#155L])
      :     +- Exchange hashpartitioning(continent#99, 400), ENSURE_REQUIREMENTS, [plan_id=309]
      :        +- HashAggregate(keys=[continent#99], functions=[partial_count(1), partial_sum(population#102L)], output=[continent#99, count#170L, sum
      :           +- Project [continent#99, population#102L]
      :              +- Filter isnotnull(continent#99)
      :                 +- Scan ExistingRDD[continent#99,country#100,name#101,population#102L]
      +- Sort [continent#187 ASC NULLS FIRST], false, 0
         +- Exchange hashpartitioning(continent#187, 400), ENSURE_REQUIREMENTS, [plan_id=313]
            +- Filter isnotnull(continent#187)
               +- Scan ExistingRDD[continent#187,country#188,name#189,population#190L]
```

Ввод [23]:

```
(
    res
    .join(
        df,
        on=['continent'],
        how="inner"
    )
    .explain(True)
)
```

Upd. К лекции мы собрали ноутбук с кодом для изучения + комментариями по материалу (prac_for_lecture_4.ipynb)