

Лекция 7. Apache Spark RDD/Dataframe/Dataset

в 90% случаях
используют



Dataframe и Dataset

В отличие от RDD API, Spark может предоставлять 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

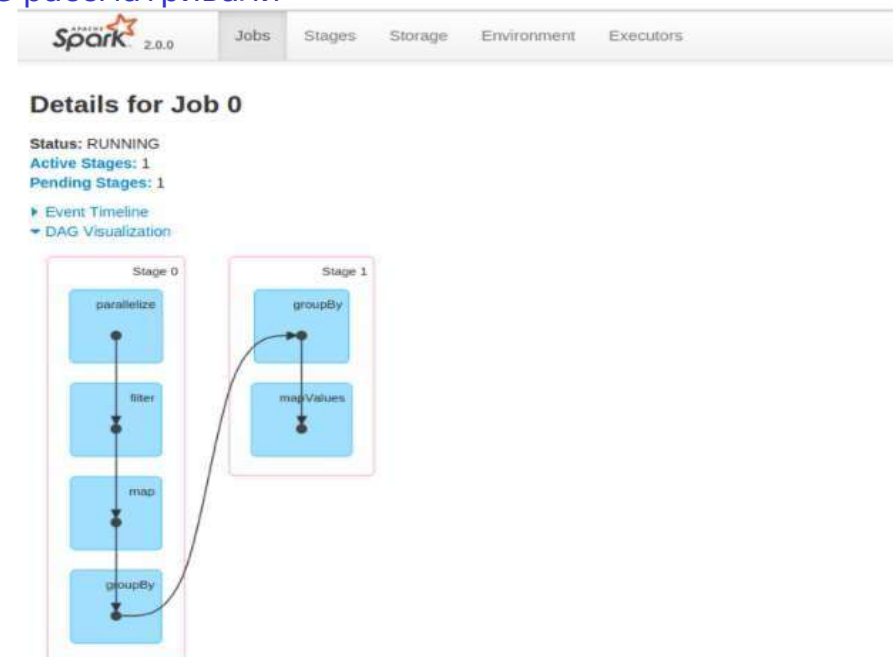
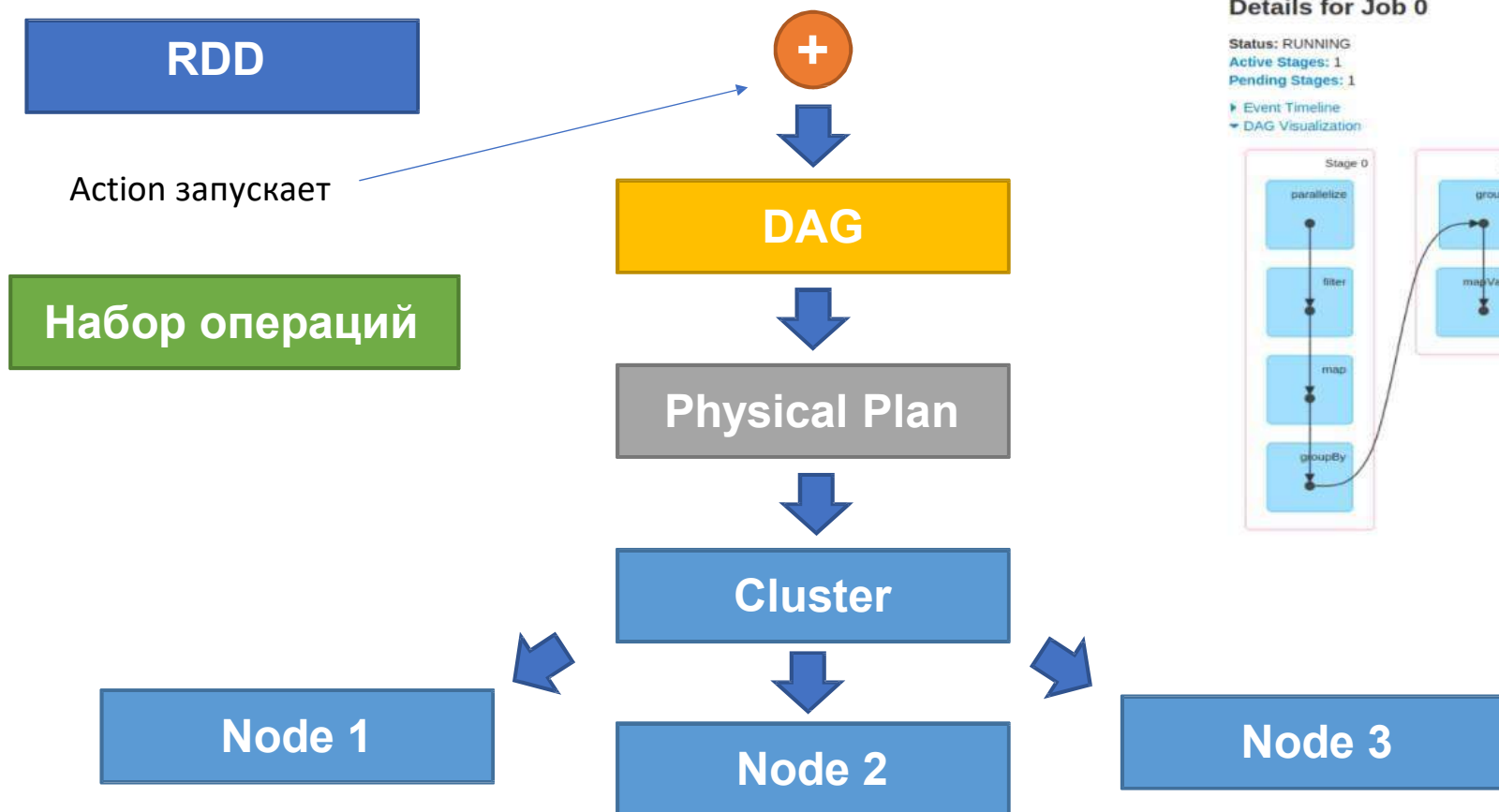
Dataset - абстракция специфичная для Spark SQL

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

Dataframe API доступна на Scala, Java, Python и R

RDD в цепочке выполнения Spark программы

Spark умеет выстраивать цепочки ранее рассматривали



Создание RDD из Scala коллекции

1.Инициализация Spark контекста, который поддерживает RDD

2. Инициализация Scala коллекции

3. Конвертация Scala коллекции в RDD

Создание RDD из Scala коллекции

ЧЕРЕЗ SPARK CONTEXT

```
val conf = new SparkConf() // Задаем настройки  
  .setAppName("spark-test") // указание имени приложения (должно быть уникальным)  
  .setMaster("local[2]") // на чем выполняется Spark приложение [2] - указание использовать 2 ядра  
                           // иначе все возможные ядра использует или можно указать *, что означает использо-  
                           // вать все ядра  
val sc = new SparkContext(conf)
```

2. Инициализация Scala коллекции

3. Конвертация Scala коллекции в RDD

Создание RDD из Scala коллекции

```
val conf = new SparkConf()  
  .setAppName("spark-test")  
  .setMaster("local[2]")
```

```
val sc = new SparkContext(conf)
```

```
val alphabet: Seq[Char] = 'a' to 'я' создаем массив от а до я - это и будет нашей коллекцией
```

3. Конвертация Scala коллекции в RDD

Создание RDD из Scala коллекции

Создание RDD из Scala коллекции

```
val conf = new SparkConf()  
  .setAppName("spark-test")  
  .setMaster("local[2]")
```

```
val sc = new SparkContext(conf)
```

```
val alphabet: Seq[Char] = 'a' to 'я'
```

```
val alphabetRDD = sc.parallelize(alphabet)
```


Создание RDD из файла

```
val taxiZoneRDD: RDD[String] = sc.textFile("taxi_zones.csv")
```

Создание RDD из файла

```
val taxiZoneRDD: RDD[String] = sc.textFile("taxi_zones.csv")
```

Создание RDD из файла

```
val taxiZoneRDD: RDD[String] = sc.textFile("taxi_zones.csv")
```

```
"LocationID","Borough","Zone","service_zone"  
1,"EWR","Newark Airport","EWR"  
2,"Queens","Jamaica Bay","BORO"  
3,"Bronx","Allerton/Pelham Gardens","BORO"  
4,"Manhattan","Alphabet City","YELLOW"  
5,"Staten Island","Arden Heights","BORO"  
6,"Staten Island","Arrochar/Fort Wadsworth","BORO"  
7,"Queens","Astoria","BORO"  
8,"Queens","Astoria Park","BORO"  
9,"Queens","Auburndale","BORO"  
10,"Queens","Baisley Park","BORO"  
11,"Brooklyn","Bath Beach","BORO"  
12,"Manhattan","Battery Park","YELLOW"  
13,"Manhattan","Battery Park City","YELLOW"  
14,"Brooklyn","Bay Ridge","BORO"  
15,"Queens","Bay Terrace/Fort Totten","BORO"  
16,"Queens","Bayside","BORO"  
17,"Brooklyn","Bedford","BORO"  
18,"Bronx","Bedford Park","BORO"
```

Что тут плохо?

Создание RDD из файла

```
val taxiZoneRDD: RDD[String] = sc.textFile("taxi_zones.csv")
```

```
"LocationID","Borough","Zone","service_zone"  
1,"EWR","Newark Airport","EWR"  
2,"Queens","Jamaica Bay","BORO"  
3,"Bronx","Allerton/Pelham Gardens","BORO"  
4,"Manhattan","Alphabet City","yellow"  
5,"Staten Island","Arden Heights","BORO"  
6,"Staten Island","Arrochar/Fort Wadsworth","BORO"  
7,"Queens","Astoria","BORO"  
8,"Queens","Astoria Park","BORO"  
9,"Queens","Auburndale","BORO"  
10,"Queens","Baisley Park","BORO"  
11,"Brooklyn","Bath Beach","BORO"  
12,"Manhattan","Battery Park","yellow"  
13,"Manhattan","Battery Park City","yellow"  
14,"Brooklyn","Bay Ridge","BORO"  
15,"Queens","Bay Terrace/Fort Totten","BORO"  
16,"Queens","Bayside","BORO"  
17,"Brooklyn","Bedford","BORO"  
18,"Bronx","Bedford Park","BORO"
```

RDD из Строк

```
val taxiZoneRDD: RDD[String] = sc.textFile("taxi_zones.csv")
```

```
"LocationID","Borough","Zone","service_zone"  
1,"EWR","Newark Airport","EWR"  
2,"Queens","Jamaica Bay","BORO"  
3,"Bronx","Allerton/Pelham Gardens","BORO"  
4,"Manhattan","Alphabet City","yellow"  
5,"Staten Island","Arden Heights","BORO"  
6,"Staten Island","Arrochar/Fort Wadsworth","BORO"  
7,"Queens","Astoria","boro"  
8,"Queens","Astoria Park","BORO"  
9,"Queens","Auburndale","BORO"  
10,"Queens","Baisley Park","BORO"  
11,"Brooklyn","Bath Beach","Boro"  
12,"Manhattan","Battery Park","YELLOW"  
13,"Manhattan","Battery Park City","yellow"  
14,"Brooklyn","Bay Ridge","BORO"  
15,"Queens","Bay Terrace/Fort Totten","BORO"  
16,"Queens","Bayside","BORO"  
17,"Brooklyn","Bedford","BORO"  
18,"Bronx","Bedford Park","BORO"
```

Добавляем типизацию

case class - помогает типизировать данные для Spark-a

```
case class TaxiZone(locationID: Int,  
                    borough: String,  
                    zone: String,  
                    serviceZone: String)  
  
val taxiZoneRDD: RDD[TaxiZone] = sc.textFile("taxi_zones.csv")  
    .map(tokens => TaxiZone(tokens(0).toInt, tokens(1),  
                            tokens(2), tokens(3))) тут обращение по индексу как в массиве
```

Добавляем типизацию

```
case class TaxiZone(locationID: Int,  
                    borough: String,  
                    zone: String,  
                    serviceZone: String)  
  
val taxiZoneRDD: RDD[TaxiZone] = sc.textFile("taxi_zones.csv")  
  .map(tokens => TaxiZone(tokens(0).toInt, tokens(1),  
                          tokens(2), tokens(3)))
```


Добавляем фильтрацию

```
case class TaxiZone(locationID: Int,  
                    borough: String,  
                    zone: String,  
                    serviceZone: String)  
  
val taxiZoneRDD: RDD[TaxiZone] = sc.textFile("taxi_zones.csv")  
  .map(tokens => TaxiZone(tokens(0).toInt, tokens(1),  
                          tokens(2), tokens(3)))  
  .filter(tz => tz.zone.toUpperCase() == tz.zone)
```

Добавляем фильтрацию

```
case class TaxiZone(locationID: Int,  
                    borough: String,  
                    zone: String,  
                    serviceZone: String)  
  
val taxiZoneRDD: RDD[TaxiZone] = sc.textFile("taxi_zones.csv")  
  .map(tokens => TaxiZone(tokens(0).toInt, tokens(1),  
                          tokens(2), tokens(3)))  
  .filter(tz => tz.zone.toUpperCase() == tz.zone)
```

Трансформации – функции высшего порядка

```
case class TaxiZone(locationID: Int,  
                    borough: String,  
                    zone: String,  
                    serviceZone: String)  
  
val taxiZoneRDD: RDD[TaxiZone] = sc.textFile("taxi_zones.csv")  
  .map(tokens => TaxiZone(tokens(0).toInt, tokens(1),  
                          tokens(2), tokens(3)))  
  .filter(tz => tz.zone.toUpperCase() == tz.zone)
```

Трансформации – функции высшего порядка

```
case class TaxiZone(locationID: Int,  
                    borough: String,  
                    zone: String,  
                    serviceZone: String)  
  
val taxiZoneRDD: RDD[TaxiZone] = sc.textFile("taxi_zones.csv")  
  .map(tokens => TaxiZone(tokens(0).toInt, tokens(1),  
                          tokens(2), tokens(3)))  
  .filter(tz => tz.zone.toUpperCase() == tz.zone)
```

Выполняющие пользовательские функции

```
case class TaxiZone(locationID: Int,  
                    borough: String,  
                    zone: String,  
                    serviceZone: String)  
  
val taxiZoneRDD: RDD[TaxiZone] = sc.textFile("taxi_zones.csv")  
  .map(tokens => TaxiZone(tokens(0).toInt, tokens(1),  
                          tokens(2), tokens(3)))  
  .filter(tz => tz.zone.toUpperCase() == tz.zone)
```

Новые RDD с narrow зависимостями

```
case class TaxiZone(locationID: Int,  
                    borough: String,  
                    zone: String,  
                    serviceZone: String)  
  
val taxiZoneRDD: RDD[TaxiZone] = sc.textFile("taxi_zones.csv")  
  .map(tokens => TaxiZone(tokens(0).toInt, tokens(1),  
                          tokens(2), tokens(3)))  
  .filter(tz => tz.zone.toUpperCase() == tz.serviceZone)
```

Не вызывают шафлинга и lazy

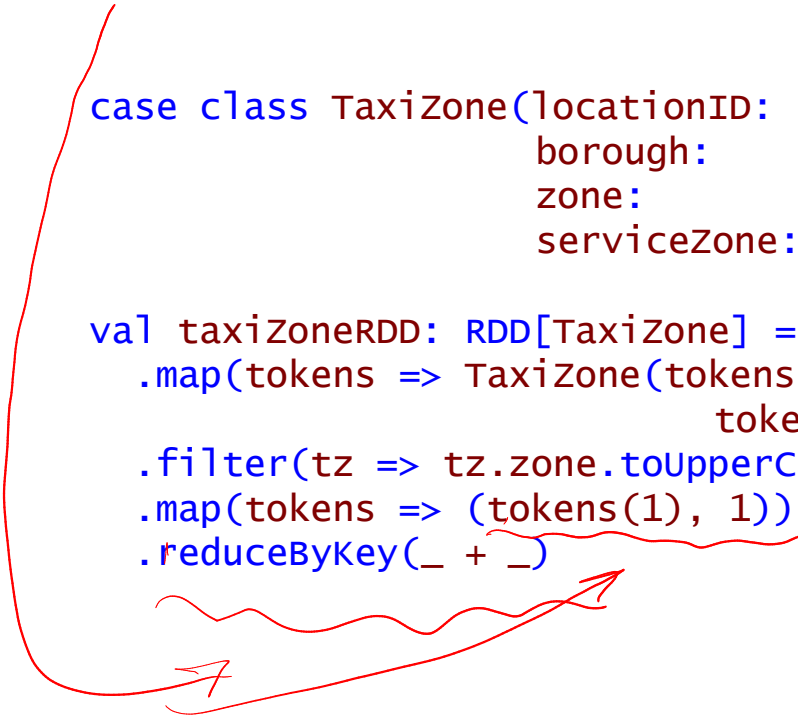
```
case class TaxiZone(locationID: Int,  
                    borough: String,  
                    zone: String,  
                    serviceZone: String)  
  
val taxiZoneRDD: RDD[TaxiZone] = sc.textFile("taxi_zones.csv")  
  .map(tokens => TaxiZone(tokens(0).toInt, tokens(1),  
                          tokens(2), tokens(3)))  
  .filter(tz => tz.zone.toUpperCase() == tz.zone)
```



Добавим агрегацию

Парадигма MapReduce:

```
case class TaxiZone(locationID: Int,  
                    borough: String,  
                    zone: String,  
                    serviceZone: String)  
  
val taxiZoneRDD: RDD[TaxiZone] = sc.textFile("taxi_zones.csv")  
  .map(tokens => TaxiZone(tokens(0).toInt, tokens(1),  
                          tokens(2), tokens(3)))  
  .filter(tz => tz.zone.toUpperCase() == tz.zone)  
  .map(tokens => (tokens(1), 1))  
  .reduceByKey(_ + _)
```



Новые RDD с wide зависимостями

```
case class TaxiZone(locationID: Int,  
                    borough: String,  
                    zone: String,  
                    serviceZone: String)  
  
val taxiZoneRDD: RDD[TaxiZone] = sc.textFile("taxi_zones.csv")  
  .map(tokens => TaxiZone(tokens(0).toInt, tokens(1),  
                          tokens(2), tokens(3)))  
  .filter(tz => tz.zone.toUpperCase() == tz.zone)  
  .map(tokens => (tokens(1), 1))  
  .reduceByKey(_ + _)
```

Вызывают шафлинг и тоже lazy

```
case class TaxiZone(locationID: Int,  
                    borough: String,  
                    zone: String,  
                    serviceZone: String)  
  
val taxiZoneRDD: RDD[TaxiZone] = sc.textFile("taxi_zones.csv")  
  .map(tokens => TaxiZone(tokens(0).toInt, tokens(1),  
                          tokens(2), tokens(3)))  
  .filter(tz => tz.zone.toUpperCase() == tz.zone)  
  .map(tokens => (tokens(1), 1))  
  .reduceByKey(_ + _)
```



Добавим запуск вычислений

```
case class TaxiZone(locationID: Int,  
                    borough: String,  
                    zone: String,  
                    serviceZone: String)  
  
val taxiZoneRDD: RDD[TaxiZone] = sc.textFile("taxi_zones.csv")  
  .map(tokens => TaxiZone(tokens(0).toInt, tokens(1),  
                          tokens(2), tokens(3)))  
  .filter(tz => tz.zone.toUpperCase() == tz.zone)  
  .map(tokens => (tokens(1), 1))  
  .reduceByKey(_ + _)  
  
taxiZonesRdd  
  .foreach(x => println(x._1, x._2))
```

Action – операции запускающие расчет

```
case class TaxiZone(locationID: Int,  
                    borough: String,  
                    zone: String,  
                    serviceZone: String)  
  
val taxiZoneRDD: RDD[TaxiZone] = sc.textFile("taxi_zones.csv")  
  .map(tokens => TaxiZone(tokens(0).toInt, tokens(1),  
                          tokens(2), tokens(3)))  
  .filter(tz => tz.zone.toUpperCase() == tz.zone)  
  .map(tokens => (tokens(1), 1))  
  .reduceByKey(_ + _)  
  
taxiZonesRdd  
  .foreach(x => println(x._1, x._2))
```



Action – операции запускающие расчет

```
case class TaxiZone(locationID: Int,  
                    borough: String,  
                    zone: String,  
                    serviceZone: String)
```

```
val taxiZoneRDD: RDD[TaxiZone] = sc.textFile("taxi_zones.csv")  
  .map(tokens => TaxiZone(tokens(0).toInt, tokens(1),  
                          tokens(2), tokens(3)))  
  .filter(tz => tz.zone.toUpperCase() == tz.zone)  
  .map(tokens => (tokens(1), 1))  
  .reduceByKey(_ + _)
```

```
taxiZonesRdd  
  .foreach(x => println(x._1, x._2))
```



"Bronx"	-> 43
"Manhattan"	-> 67
"Brooklyn"	-> 60
"Queens"	-> 68
"EWR"	-> 1
"Unknown"	-> 2
"Staten Island"	-> 20

Вот такой
историче го
быт 6

RDD из верхнеуровневых API

```
val taxiZoneDF: DataFrame = spark.read  
  .option("header", "true")  
  .option("inferSchema", "true")  
  .csv("taxi_zones.csv")
```

```
val taxiZoneRDD: RDD[Row] = taxiZoneDF.rdd
```

RDD Итоги:

Плюсы:

- Управление performance с помощью контроля партиционирования и порядка операций
- Типизация

Минусы:

- Для сложных операций необходимо хорошо знать внутренности Спарка
- Не использует преимущества колоночных типов
- Shuffle данных между партициями в произвольном месте

Программист просто пишет, где вызывать эту операцию shuffling-a, но это не всегда оптимально, поэтому все так любят Dataframe / Dataset потому что тут эти операции вызываются там, где это оптимально, а не там где программист написал...

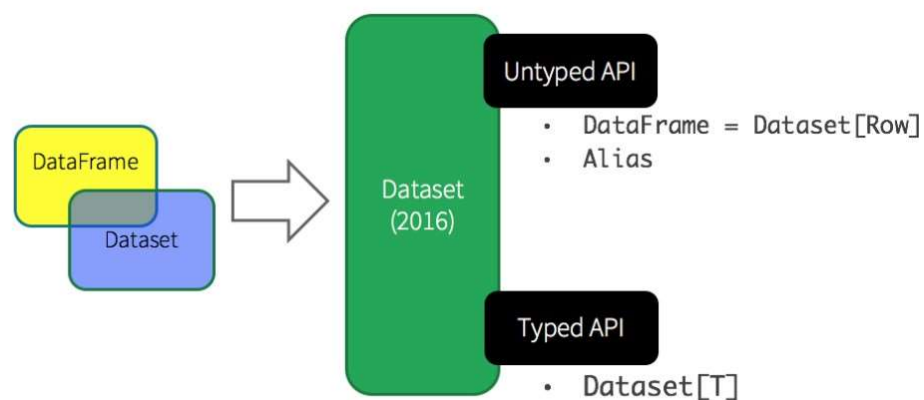
Когда использовать:

- Неструктурированный источник
- Логика описана с помощью Lambda-функций
- Нет схемы или структурированных данных
- Можем пожертвовать оптимизациями Spark

Dataframe и Dataset

В Apache Spark 2.0 появляются ОНИ

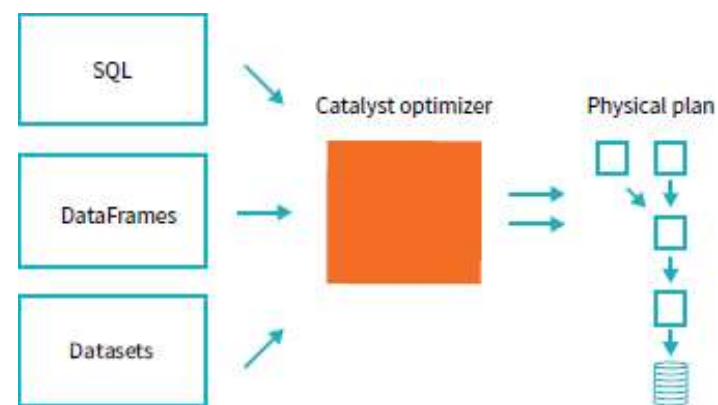
Unified Apache Spark 2.0 API



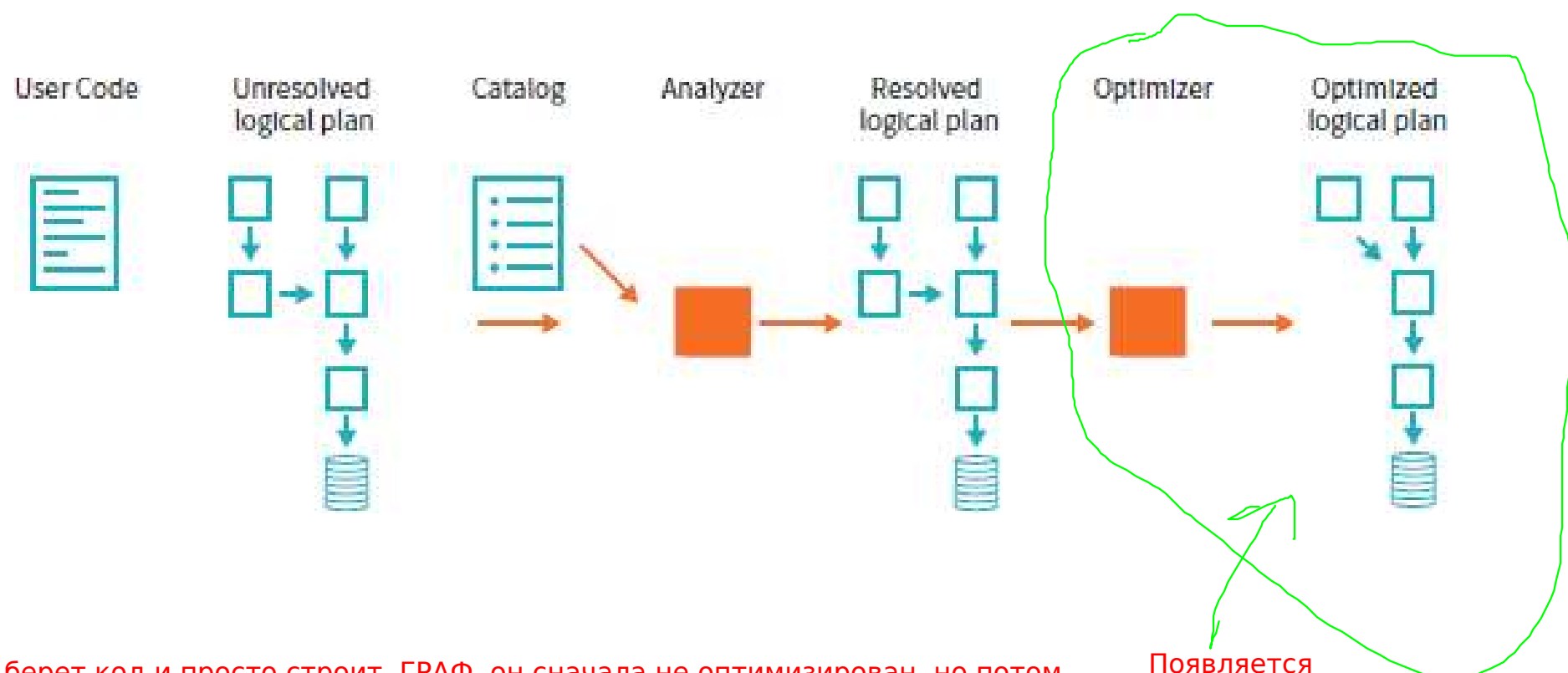
Язык программирование	Основная абстракция
Scala	Dataset[T] & DataFrame (alias for Dataset[Row])
Java	Dataset[T]
Python	DataFrame
R	DataFrame

Этапы выполнения кода Dataframe/Dataset

- 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



Druver - берет код и просто строит ГРАФ, он сначала не оптимизирован, но потом все операции старается оптимизировать, например на рисунке сверху операция shuffling-а оптимизирует... Иногда некие таски убирает. Есть возможность отключить оптимизацию, но на начальных этапах лучше сохранить их

Формирование физического плана выполнения



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

Создание DataFrame из внешнего файла

1. Инициализация Spark session

2. Создание DataFrame и чтение из файла

3. Вывод схемы

Создание DataFrame из внешнего файла

```
val sparkSession = SparkSession.builder() // конфигурация SparkSession  
  .appName("Introduction to DataFrame") // имя приложения  
  .config("spark.master", "local") // spark master вызови мне ...  
  .getOrCreate()
```

2. Создание DataFrame и чтение из файла

3. Вывод схемы

Создание DataFrame из внешнего файла

```
val sparkSession = SparkSession.builder()  
  .appName("Introduction to DataFrame")  
  .config("spark.master", "local")  
  .getOrCreate()
```

```
val taxiZoneDF = sparkSession.read  
  .option("inferSchema", "true")  
  .csv("taxi_zones.csv") тут чтение csv...
```

3. Вывод схемы

Создание DataFrame из внешнего файла

```
val sparkSession = SparkSession.builder()  
  .appName("Introduction to DataFrame")  
  .config("spark.master", "local")  
  .getOrCreate()
```

```
val taxiZoneDF = sparkSession.read  
  .option("inferSchema", "true")  
  .csv("taxi_zones.csv")
```

```
taxiZoneDF  
  .printSchema()    выводим нашу схему
```

Spark определяет схему

```
val sparkSession = SparkSession.builder()  
  .appName("Introduction to DataFrame")  
  .config("spark.master", "local")  
  .getOrCreate()
```

```
val taxiZoneDF = sparkSession.read  
  .option("inferSchema", "true")  
  .csv("taxi_zones.csv")
```

```
taxiZoneDF  
  .printSchema()
```



```
"LocationID", "Borough", "Zone", "service_zone"  
1, "EWR", "Newark Airport", "EWR"  
2, "Queens", "Jamaica Bay", "BORO"  
3, "Bronx", "Allerton/Pelham Gardens", "BORO"  
4, "Manhattan", "Alphabet City", "yeLLow"  
5, "Staten Island", "Arden Heights", "BORO"  
6, "Staten Island", "Arrochar/Fort Wadsworth", "BORO"  
7, "Queens", "Astoria", "BORO"  
8, "Queens", "Astoria Park", "BORO"  
9, "Queens", "Auburndale", "BORO"  
10, "Queens", "Baisley Park", "BORO"  
11, "Brooklyn", "Bath Beach", "BORO"  
12, "Manhattan", "Battery Park", "YELLOW"  
13, "Manhattan", "Battery Park City", "yeLLow"  
14, "Brooklyn", "Bay Ridge", "BORO"
```


Spark определяет схему

```
val sparkSession = SparkSession.builder()  
  .appName("Introduction to DataFrame")  
  .config("spark.master", "local")  
  .getOrCreate()
```

```
val taxiZoneDF = sparkSession.read  
  .option("inferSchema", "true")  
  .csv("taxi_zones.csv")
```

```
taxiZoneDF  
  .printSchema()
```

```
"LocationID", "Borough", "Zone", "service_zone"  
1, "EWR", "Newark Airport", "EWR"  
2, "Queens", "Jamaica Bay", "BORO"  
3, "Bronx", "Allerton/Pelham Gardens", "BORO"  
4, "Manhattan", "Alphabet City", "yeLLow"  
5, "Staten Island", "Arden Heights", "BORO"  
6, "Staten Island", "Arrochar/Fort Wadsworth", "BORO"  
7, "Queens", "Astoria", "BORO"  
8, "Queens", "Astoria Park", "BORO"  
9, "Queens", "Auburndale", "BORO"  
10, "Queens", "Baisley Park", "BORO"  
11, "Brooklyn", "Bath Beach", "BORO"  
12, "Manhattan", "Battery Park", "YELLOW"  
13, "Manhattan", "Battery Park City", "yeLLow"  
14, "Brooklyn", "Bay Ridge", "BORO"
```

```
root  
|-- LocationID: string (nullable = true)  
|-- Borough: string (nullable = true)  
|-- Zone: string (nullable = true)  
|-- service_zone: string (nullable = true)
```

Другие способы создания DataFrame

1. SPARK SQL

2. Из Scala коллекций

3. Из RDD

Другие способы создания DataFrame

```
val taxizoneDF = sparkSession.sql("SELECT * FROM taxi_zone")
```

2. Из Scala коллекций

3. Из RDD

Другие способы создания DataFrame

```
val taxizoneDF = sparkSession.sql("SELECT * FROM taxi_zone")
```

```
val cars = Seq(  
  ("bugatti",18,8,307,130,3504,12.0,"1970-01-01","USA"),  
  ("solara",15,8,350,165,3693,11.5,"1970-01-01","USA"))  
val manualCarsDF = spark.createDataFrame(cars)
```

3. Из RDD

Другие способы создания DataFrame

```
val taxizoneDF = sparkSession.sql("SELECT * FROM taxi_zone")
```

```
val cars = Seq(  
  ("bugatti",18,8,307,130,3504,12.0,"1970-01-01","USA"),  
  ("solara",15,8,350,165,3693,11.5,"1970-01-01","USA"))  
val manualCarsDF = spark.createDataFrame(cars)
```

```
val alphabetRDD = sc.parallelize('a' to 'я')  
val alphabetDF = alphabetRDD.toDF
```

Логика в DataFrame

Реализуется с помощью DSL

domain specific language & spark SQL

Добавим фильтрацию

```
taxiZoneDF  
  .filter(upper(col("service_zone")) === col("service_zone"))
```

Добавим фильтрацию

```
taxiZoneDF  
  .filter(upper(col("service_zone")) == col("service_zone"))
```


Добавим агрегацию

```
taxiZoneDF  
  .filter(upper(col("service_zone")) === col("service_zone"))  
  .groupBy(col("Borough"))  
  .count()
```

Добавим агрегацию

```
taxiZoneDF  
  .filter(upper(col("service_zone")) === col("service_zone"))  
  .groupBy(col("Borough"))  
  .count()
```

Выведем результат

```
taxiZoneDF  
  .filter(upper(col("service_zone")) === col("service_zone"))  
  .groupBy(col("Borough"))  
  .count()  
  .show()
```

Выведем результат

```
taxiZoneDF  
  .filter(upper(col("service_zone")) === col("service_zone"))  
  .groupBy(col("Borough"))  
  .count()  
  .show()
```

Выведем результат

```
taxiZoneDF  
  .filter(upper(col("service_zone")) === col("service_zone"))  
  .groupBy(col("Borough"))  
  .count()  
  .show()
```



Queens	68
EWB	1
Unknown	2
Brooklyn	60
Staten Island	20
Manhattan	67
Bronx	43

Меньше кода лучше читаемость

```
val spark: SparkSession = SparkSession.builder()
  .appName( name= "Introduction to RDDs")
  .config("spark.master", "local")
  .getOrCreate()

val sc: SparkContext = spark.sparkContext

case class TaxiZone(locationID: Int,
                    borough: String,
                    zone: String,
                    serviceZone: String)

val taxiZonesRdd: RDD[(String, Int)] = sc
  .textFile( path= "src/main/resources/data/taxi_zones.csv")
  .map(line => line.split( regex= " "))
  .filter(tokens => tokens(3).toUpperCase() == tokens(3))
  .map(tokens => TaxiZone(tokens(0).toInt, tokens(1), tokens(2), tokens(3)))
  .map(tz => (tz.borough, 1))
  .reduceByKey(_ + _)

taxiZonesRdd
  .foreach(x => println(s"${x._1} -> ${x._2} ")) // Операция действия
```

```
val sparkSession: SparkSession = SparkSession.builder()
  .appName( name= "Introduction to DataFrame")
  .config("spark.master", "local")
  .getOrCreate()

val taxiZoneDF: DataFrame = sparkSession.read
  .option("header", "true")
  .csv( path= "src/main/resources/data/taxi_zones.csv")

taxiZoneDF
  .filter(upper(col( colName= "service_zone")) == col( colName= "service_zone"))
  .groupBy(col( colName= "Borough")).count()
  .show()
```

DataFrame итоги:

Плюсы:

- high level API со своим DSL
- Строгая типизация
- Простота в использовании и надежности

Минусы:

- При работе с неструктурированными типами данных необходимо учитывать особенности

Когда использовать:

- Структурированный источник
- Когда нужна оптимизация и производительность
- Богатый DSL
- Когда есть логика на SQL (например Hive)

Создание Dataset из внешнего файла

1. Создаем класс
2. Читаем DataFrame из файла
3. Импортируем энкодер
4. Конвертируем DataFrame в DataSet

Создание Dataset из внешнего файла

```
case class TaxiZone(LocationID: Int,  
                    Borough: String,  
                    Zone: String,  
                    service_zone: String)
```

Dataset типизированная история...

2. Читаем DataFrame из файла

3. Импортируем энкодер

4. Конвертируем DataFrame в DataSet

Создание DataFrame из внешнего файла

```
case class TaxiZone(LocationID: Int,  
                    Borough: String,  
                    Zone: String,  
                    service_zone: String)
```

taxi_zones.csv

```
"LocationID","Borough","Zone","service_zone"  
1,"EWR","Newark Airport","EWR"  
2,"Queens","Jamaica Bay","BORO"
```

2. Читаем DataFrame из файла

3. Импортируем энкодер

4. Конвертируем DataFrame в DataSet

Создание DataFrame из внешнего файла

```
case class TaxiZone(LocationID: Int,  
                    Borough: String,  
                    Zone: String,  
                    service_zone: String)
```

```
val taxiZoneDF = sparkSession.read
```

3. Импортируем энкодер

4. Конвертируем DataFrame в DataSet

Создание DataFrame из внешнего файла

```
case class TaxiZone(LocationID: Int,  
                    Borough: String,  
                    Zone: String,  
                    service_zone: String)
```

```
val taxiZoneDF = sparkSession.read
```

```
import sparkSession.implicits._
```

4. Конвертируем DataFrame в DataSet

Создание DataFrame из внешнего файла

```
case class TaxiZone(LocationID: Int,  
                    Borough: String,  
                    Zone: String,  
                    service_zone: String)
```

```
val taxiZoneDF = sparkSession.read
```

```
import sparkSession.implicits._
```

```
val taxiZoneDS = taxiZoneDF.as[TaxiZone]
```

Логика в Dataset

Реализуется с помощью

domain specific language & spark SQL

Логика в Dataset

Реализуется с помощью

domain specific language & spark SQL



type safety & lambda

Добавляем фильтрацию

```
taxiZoneDS  
  .filter(_.service_zone.toUpperCase == _.service_zone)
```


Добавляем фильтрацию

```
taxiZoneDS  
  .filter(_.service_zone.toUpperCase == _.service_zone)
```



type safety lambda

Добавляем агрегацию

```
taxiZoneDS  
  .filter(_.service_zone.toUpperCase == _.service_zone)  
  .groupBy(col("Borough"))  
  .count()
```

Создание DataFrame из внешнего файла

```
taxiZoneDS  
  .filter(_.service_zone.toUpperCase == _.service_zone)  
  .groupBy(col("Borough"))  
  .count()
```



domain specific language

Добавляем агрегацию

```
taxiZoneDS  
  .filter(_.service_zone.toUpperCase == _.service_zone)  
  .groupBy(col("Borough"))  
  .count()  
  .show()
```



Queens	68
EWB	1
Unknown	2
Brooklyn	60
Staten Island	20
Manhattan	67
Bronx	43

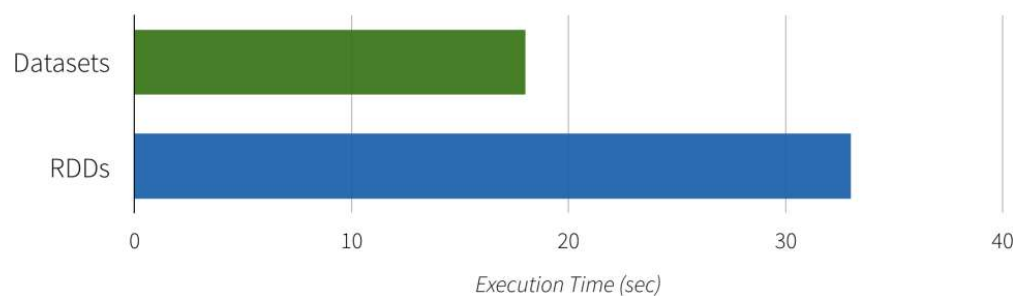
Dataset итоги:

Плюсы:

- От RDD
 - Типизация
 - Lambda выражения
- От Dataframe
 - Оптимизации Catalyst
 - Оптимальное использование ресурсов
 - Shuffling без сериализации

Сравнение RDD, Dataframe и Dataset

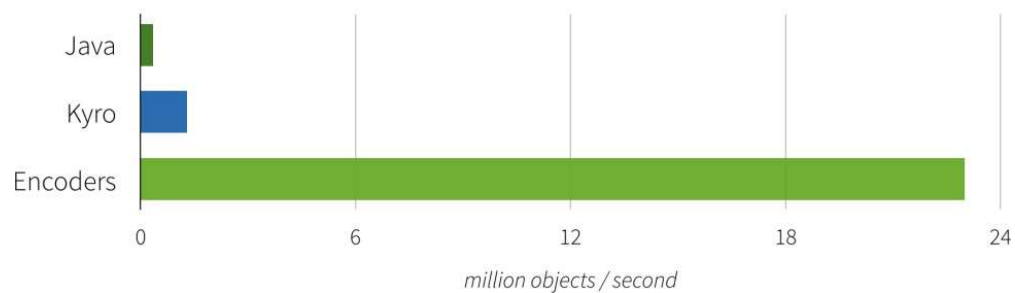
Distributed Wordcount



Memory Usage when Caching

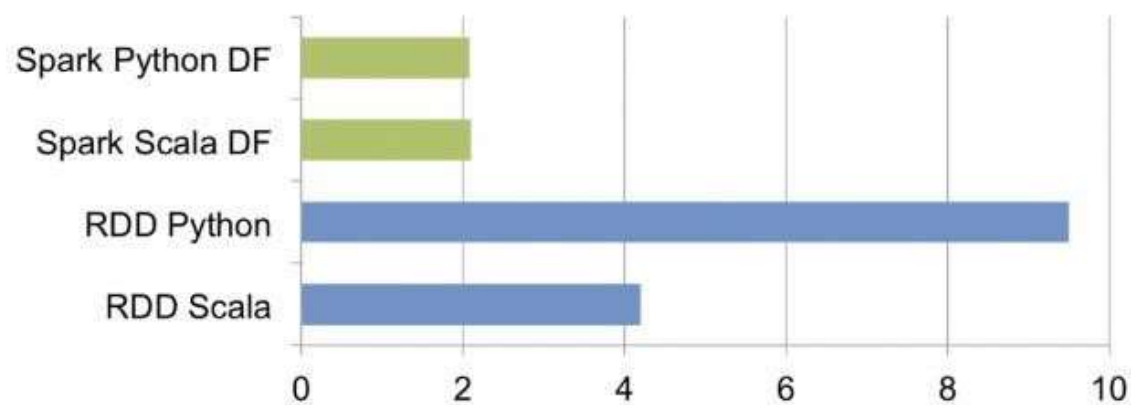


Serialization / Deserialization Performance



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

Сравнение RDD, Dataframe и Dataset



Performance of aggregating 10 million int pairs (secs)

Показывает скорость агрегации данных...