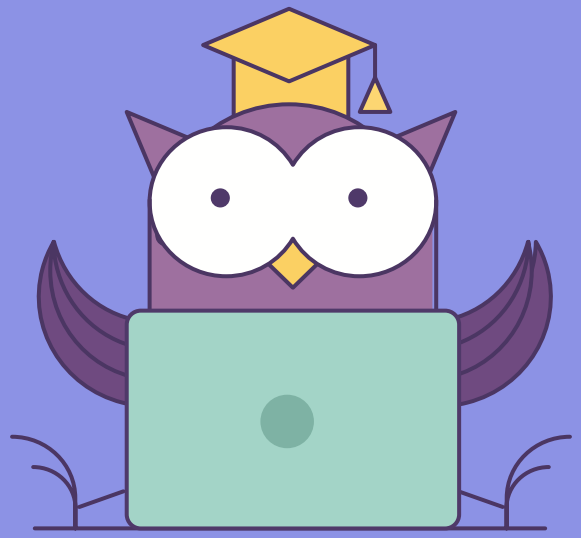
ОНЛАЙН-ОБРАЗОВАНИЕ

Меня хорошо слышно && видно?



Напишите в чат, если есть проблемы!

Ставьте + если все хорошо

OTUS

Apache Spark

Основы и API

Егор Матешук

Цели сегодняшнего занятия

OTUS

После этого занятия вы будете знать

- В каких случаях применять Apache Spark
- Какие виды API есть у Spark
- Какие функции эти API предоставляют

После этого занятия вы сможете

- Создать простой проект с использованием Spark
- Написать программу, обрабатывающую данные в распределенном режиме

01

Что такое Apache Spark

Когда MapReduce недостаточно

OTUS

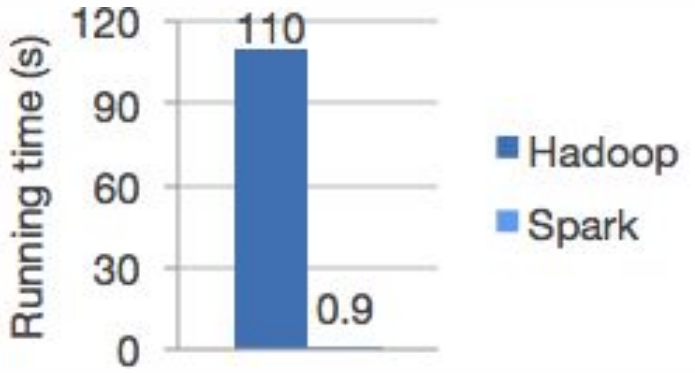
Недостатки MapReduce

- Постоянное использование диска
- Отсутствие оптимизатора запросов
- Неоптимальные форматы данных

И тут появляется Apache Spark

OTUS

В 2010 году появляется Apache Spark, который решает все перечисленные ранее проблемы, представляя программы в виде DAG вычислений



Что такое Apache Spark

Apache Spark - фреймворк для распределенных вычислений на базе Hadoop.

Является стандартом де-факто в задачах обработки больших данных.

Отлично подходит для работы со слабоструктурированными данными.

Расширения Apache Spark

OTUS

- Spark Streaming - потоковая аналитика
- MLlib - machine learning на спарке
- GraphX - обработка графов
- GeoSpark - для работы с гео-данными

Поддерживает 3 языка для разработки:

- Python (без DataSet API)
- Java
- Scala

Когда стоит применять Spark?

Если у вас много данных и вам нужна аналитика

- Распределенная архитектура
- Тесная интеграция с Hadoop
- Поддержка колоночных форматов
- Поддержка всевозможных хранилищ и баз
- SQL-подобные интерфейсы

Когда стоит применять Spark?

Если ваши данные неструктурированные или слабоструктурированные

- Любые кастомные трансформации (без ограничений SQL)
- Поддержка text/json/csv из коробки

Когда стоит применять Spark?

Если вам нужно делать интерактивный анализ данных без Python

- Самый удобный API для аналитических запросов на Scala/Java
- Поддержка Jupyter Notebook, Zeppelin, CLI (spark-shell)

02

Немного о Scala

Почему именно Scala

OTUS

Description

Takes one element and produces one element. A map function that doubles the values of the input stream:
`DataStream<Integer> dataStream = //...`
`dataStream.map(new MapFunction<Integer, Integer>() {`
`  @Override`
`  public Integer map(Integer value) throws Exception {`
`  return 2 * value;`
`  }`
`});`

Takes one element and produces zero, one, or more elements. A flatmap function that splits sentences to words:
`dataStream.flatMap(new FlatMapFunction<String, String>() {`
`  @Override`
`  public void flatMap(String value, Collector<String> out)`
`  throws Exception {`
`  for(String word: value.split(" ")){`
`  out.collect(word);`
`  }`
`  }`
`});`

Java

Description
Takes one element and produces one element. A map function that doubles the values of the input stream: `dataStream.map { x => x * 2 }`
Takes one element and produces zero, one, or more elements. A flatmap function that splits sentences to words: `dataStream.flatMap { str => str.split(" ") }`

Scala

Что такое Scala

- Объектно-ориентированный
- Функциональный
- Статически типизированный
- Работает на JVM
- Может выполнять код Java

Типы данных

- String
- Int, Double, ...
- TupleN
- Collections: List, Set, Map, etc.
- Mutable collections
- Option
- Unit (= Void in Java)

Значения и переменные

OTUS

- **val** – неизменяемый

```
scala> val xVal: Int = 1
xVal: Int = 1
scala> xVal = 2
<console>:12: error:
reassignment to val
```

- **var** – изменяемый

```
scala> var xVar: Int = 1
xVar: Int = 1
scala> xVar = 2
xVar: Int = 2
```

Функции

OTUS

Анонимные функции:

```
x => x + 1
```

Методы:

```
def incr(x: Int): Int = x + 1
```

Составные выражения

Не требуется ключевое слово «return» (это необязательно). Последнее выражение возвращает значение.

```
val c = {
  val a = 11
  a + 42
}
```

Если выражение не возвращает значение, то его тип “Unit”

```
def printer(s: String): Unit = println(s)
```

Управляющие структуры: if

Синтаксис похож на другие C-подобные языки

```
if ( x < 20 ) {
   println("This is if statement")
}
```

Также возвращает значение

```
scala> val whichOne = if (false) "Not that one"
else "This one"
whichOne: String = This one
```

Управляющие структуры: for

OTUS

```
for ( a <- 1 to 10) {
   println( "Value of a: " + a )
}
```

Синтаксис очень похож, не правда ли?

Управляющие структуры: for

```
scala> for {
  a <- 1 to 5 // Коллекция 1
  b <- 1 to 5 // Коллекция 2
  if a + b < 6 // Условие
} yield a + b // Генератор
```

```
res3: Vector(2, 3, 4, 5, 3, 4, 5, 4, 5, 5)
```

Коллекции: Списки

OTUS

```scala
val fruit: List[String] = List("apples", "oranges", "pears")

val fruit = "apples" :: ("oranges" :: ("pears" :: Nil))

val fruit = List.fill(10)("apples")
```

Коллекции: Словари

```
val colors = Map("red" -> "#FF0000", "azure" -> "#F0FFFF")

println( "Keys in colors : " + colors.keys )
println( "Values in colors : " + colors.values )
println( "Check if colors is empty : " + colors.isEmpty )
```

Коллекции: Кортежи

OTUS

Коллекция объектов смешанного типа

```
val t = (1, "hello", Console)
```

N-ый элемент может быть вызван командой “x._N”

```
val t = (4,3,2,1)
val sum = t._1 + t._2 + t._3 + t._4
```

Кортежи являются базовыми структурами для других типов (например, case классы)

Тип Option означает «Может быть пустым» и содержит Some(_) или None.

```
val a: Option[Int] = Some(5)
val b: Option[Int] = None
println("a.getOrElse(0): " + a.getOrElse(0) )
println("b.getOrElse(10): " + b.getOrElse(10) )
scala> a.getOrElse(0): 5
scala> b.getOrElse(10): 10
```

Option может рассматриваться как коллекция с 0 или 1 элементом.

Case classes

OTUS

- Удобны для моделирования неизменяемых данных
- Сравнение по структуре, а не по ссылке
- Имеет метод .copy для быстрого копирования неизменяемых объектов
- Не нужно ключевое слово “new” при создании

```
case class Person(name: String, age: Int)
val garry = Person("Garry", 22)
val oldGarry = garry.copy(age=60)
```

Это как оператор “switch” на стероидах

```
x match {
    case 1 => "one"
    case "two" => 2
    case y: Int => "scala.Int"
    case _ => "many"
}
```

Pattern matching + case class

OTUS

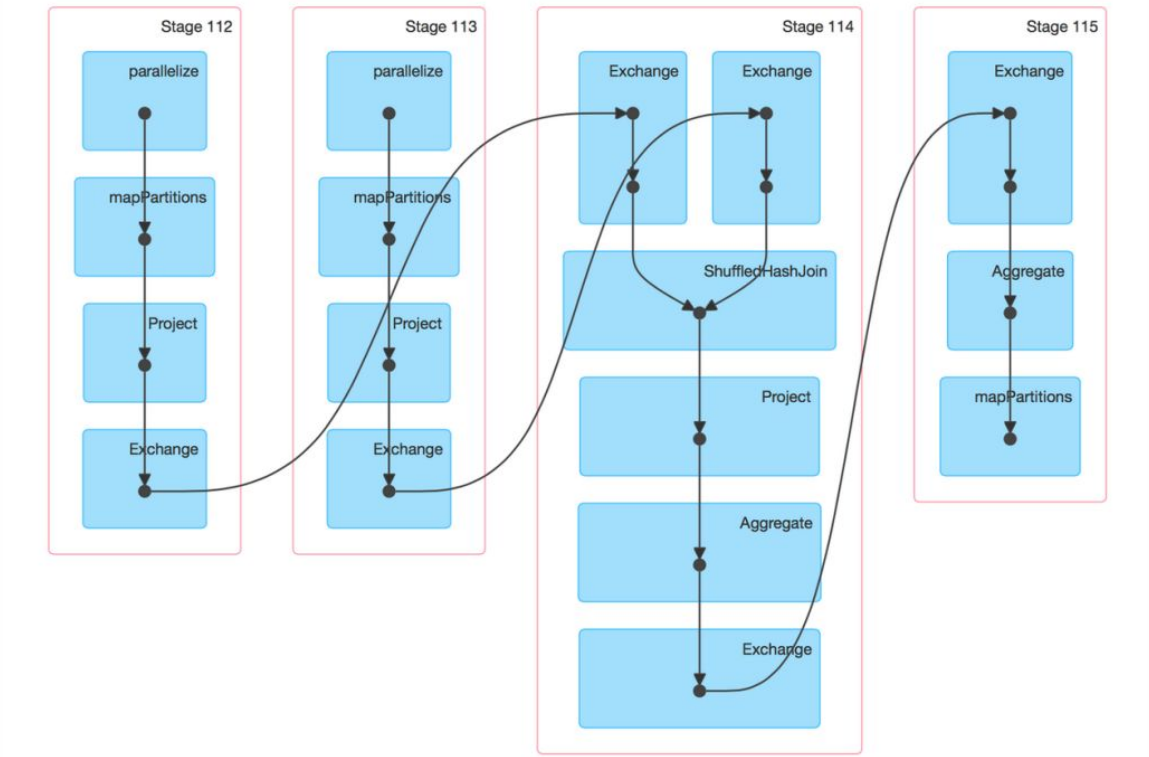
```
val alice = Person("Alice", 25)
val bob = Person("Bob", 32)
val charlie = Person("Charlie", 32)

for (person <- List(alice, bob, charlie)) {
  person match {
    case Person("Alice", _) => println("Hi Alice!")
    case Person("Bob", 32) => println("Hi Bob!")
    case Person(name, age) => println(
      "Age: " + age + " year, name: " + name + "?")
  }
}
```

03

Spark API

Что такое DAG

DAG (Directed Acyclic Graph) - граф вычислений, который будет выполнять Spark



Что такое DAG

Процесс вычислений состоит из следующих элементов

- Job - задача выполнения всего DAG
- Stage - этап вычислений, исполняемых без обмена данными между нодами
- Task - задача выполнения Stage на конкретном куске данных

Вся эта работа выполняется над распределенными коллекциями

Из чего состоит Spark

OTUS

В распределенной системе спарка есть две роли

- Driver - координатор всего процесса
- Executors - “рабочие лошадки” выполняющие распределенные вычисления

SparkSession

Отправной точкой для программы на Spark является SparkSession - создание распределенной системы для исполнения будущих вычислений

```
import org.apache.spark.sql.SparkSession

val spark = SparkSession.builder()
  .appName("Example app")
  .master("local[*]")
  .getOrCreate()


import spark.implicits._
```

Set master

Метод .setMaster в конфигурации SparkSession/SparkContext указывает, где нужно выполнять вычисления. Например

.master(“yarn”) - выполнение на кластере Hadoop

.master(“local”) - выполнение локально на машине

Spark API

У спарка есть 3 разных API, с которыми мы сегодня познакомимся:

- RDD API
- DataFrame API (он же SQL API)
- DataSet API

Все они отличаются в основном тем, в каком виде представлены распределенные коллекции при вычислениях

Как начать работать со Spark

- В интерактивном формате:
  - ядро для Jupyter Notebook - Almond
  - в консоли - spark-shell
- В качестве приложения:
  - создать новый проект с помощью

    ```
    sbt new holdenk/sparkProjectTemplate.g8
    ```
  - и импортировать его в IntelliJ IDEA

RDD API

OTUS

Resilient Distributed Datasets (RDDs) - это отказоустойчивая коллекция элементов, которая может обрабатываться параллельно.

- Коллекция разбита на партиции - физические куски данных
- Внутри коллекции элементы представлены сериализуемыми классами
- API очень похоже на классический MapReduce

SparkContext - предшественник SparkSession, который используется для работы с RDD

Scala:

```scala
val conf = new SparkConf().setAppName(appName)
new SparkContext(conf)
```

Python:

```python
conf = SparkConf().setAppName(appName)
sc = SparkContext(conf=conf)
```

Создание простейшего RDD из коллекции

```
val data = Array(1, 2, 3, 4, 5)
val distData = sc.parallelize(data)
```

Scala

```scala
val lines = sc.textFile("data.txt")
val lineLengths = lines.map(s => s.length)
val totalLength = lineLengths
  .reduce((a, b) => a + b)
```

Python

```python
lines = sc.textFile("data.txt")
lineLengths = lines.map(lambda s: len(s))
totalLength = lineLengths \
  .reduce(lambda a, b: a + b)
```

RDD API - Key-Value

RDD API поддерживает агрегации/сортировки по ключу:

```
val lines = sc.textFile("data.txt")
val pairs = lines.map(s => (s, 1))
val counts = pairs.reduceByKey((a, b) => a + b)
```

Transformations vs Actions

OTUS

При построении DAG есть два типа операций:

- Transformations - описание вычислений (map, filter, groupByKey и т. д.)
- Actions - действия, запускающие расчеты (reduce, collect, take и т. д.)

Если написать DAG без actions, то никакие вычисления запущены не будут (lazy).

Transformations

- map(func) - выполнить преобразование func над каждым элементом и вернуть новый элемент
- flatMap(func) - выполнить преобразование func над каждым элементом и вернуть коллекцию (в т. ч. пустую)
- mapPartitions(func) - выполнить преобразование func над коллекцией элементов (часть RDD) и вернуть новую коллекцию

Если у вас есть этап вычислений, где одни и те же данные переиспользуются два и более раз, то их есть смысл закэшировать:

```
val lines = sc.textFile("data.txt").cache()
val totalLength = lines
  .map(s => s.length).reduce((a, b) => a + b)
val counts = lines
  .map(s => (s, 1))
  .reduceByKey((a, b) => a + b)
  .take(10)
```

В процессе использования RDD API выявилось несколько недостатков:

- Не похоже на SQL, что затрудняет использование в целях аналитики
- Невозможность использования оптимизаций хранения данных как в колоночных форматах

Решить эти проблемы был призван DataFrame API

DataFrame API

DataFrame - колоночная структура, представляющая из себя подобие SQL-таблицы

- Всё те же физические партиции
- Но элементы теперь имеют специальную колоночную структуру, позволяющую выполнять SQL-запросы
- Быстрее и компактнее RDD, что увеличивает производительность операций
- Отсутствие типизации

DataFrame API

OTUS

```scala
val df =
spark.read.json("examples/src/main/resources/people.json")

// Displays the content of the DataFrame to stdout
df.show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

Синтаксис больше похож на обычные SQL запросы

```
df.select($"name", $"age" + 1).show()
df.filter($"age" > 21).show()
df.groupBy("age").count().show()
```

Более того - можно писать запросы прямым текстом

```
df.createOrReplaceTempView("people")


val sqlDF = spark.sql("SELECT * FROM people")
sqlDF.show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

При необходимости, мы можем использовать функции из org.apache.spark.sql.functions или написать свои:

https://spark.apache.org/docs/latest/sql-getting-started.html#untyped-user-defined-aggregate-functions

DataSet API

Здорово, теперь мы можем писать SQL-запросы.

Но появляются новые проблемы:

- приложения начинают падать в runtime, если мы сделали опечатку в названии поля
- SQL-синтаксис хорош для аналитики, но для более сложных задач не подходит

И тут появляется DataSet API!

DataSet API

DataSet - структура, вобравшая лучшее от предшественников

- Всё те же физические партиции
- Элементы внутри - классы, поддающиеся сериализации специальными методами
- DataFrame = DataSet[Row]
- Наличие строгой типизации
- Поддержка SQL-запросов

Поддерживаемые форматы

Spark умеет писать/читать множество форматов во множество хранилищ:

- CSV
- JSON
- Parquet
- ORC
- JDBC
- Kafka
- ...

Чтение

Чтение в большинстве случаев выглядит так:

```
spark.read.csv("some.csv")

spark.read.parquet("hdfs:///parquet-files/")

spark.read.json("/path-to-json-files/")
```

Запись

OTUS

Запись происходит аналогично

```
df.write.csv("some.csv")
```

```
df.write.parquet("hdfs:///parquet-files/")
```

```
df.write.json("/path-to-json-files/")
```

Классическая ситуация:

Вы провели вычисления и сохранили результат, но в папке результата оказалась куча мелких файлов.

Решение:

```
df.repartition(1)
  .write.parquet("hdfs:///parquet-files/")
```

или

```
df.write.partitionBy("date")
  .parquet("hdfs:///parquet-files/")
```

04

Подведем итоги

Цели сегодняшнего занятия

OTUS

После этого занятия вы будете знать

- В каких случаях применять Apache Spark
- Какие виды API есть у Spark
- Какие функции эти API предоставляют

После этого занятия вы сможете

- Создать простой проект с использованием Spark
- Написать программу, обрабатывающую данные в распределенном режиме

Заполните, пожалуйста, опрос в личном кабинете!

Полезные ссылки

- Документация Spark
- Примеры по Spark
- Отличный блог по Scala с кучей рецептов
- Книга по Scala (Essential Scala Book)

Егор Матешук

egor@mateshuk.com

Спасибо
за внимание!