

# Полезные конструкции Spark в реализациях на Scala и Python

## Содержание

1	Полезные ресурсы по Spark	3
2	Общие сведения	3
3	Управление зависимостями проекта с помощью build.sbt	4
4	Пример build.sbt для работы со Spark и Breeze	5
5	Начало работы со Spark	6
6	Эффективная конфигурация исполнителей	9
7	Особенности работы инфиксных математических операторов с коллекциями в Python и Scala	10
8	Пример использования группировки и агрегации на Scala	11
9	Приемы работы со объектами Spark DataFrame	11
10	Запись объектов Spark DataFrame	12
11	Выбор элементов в DataFrame и Dataset	13
12	Создание объектов Spark DataFrame на Scala	13
13	Создание объектов Spark DataFrame на Python	13
13.1	Создание Spark DataFrame на основе списка . . . . .	13
13.2	Создание Spark DataFrame на основе объекта RDD . . . . .	14
13.3	Создание Spark DataFrame на основе схемы StructType() . . . . .	14
13.4	Создание Spark DataFrame на основе pandas . . . . .	14
14	Зарегистрировать пользовательскую функцию в Spark на Scala	15
15	Зарегистрировать пользовательскую функцию в PySpark	15
16	Соединение объектов DataFrame	15
17	Фильтрация и агрегация	17
18	Сводная информация	17

<b>19 Оконные функции в контексте SQL и Spark DataFrame</b>	<b>18</b>
19.1 Базовые понятия оконных преобразований в PySpark . . . . .	18
19.2 Оконные преобразования в Spark на Scala . . . . .	24
<b>20 Работа с файловой системой Databricks</b>	<b>25</b>
<b>21 Приемы работы с библиотекой Breeze</b>	<b>26</b>
<b>22 Spark ML Pipelines</b>	<b>27</b>
<b>23 Запросы к DataFrame с помощью методов и SQL</b>	<b>29</b>
23.1 Простые примеры . . . . .	29
23.2 Использование сложных агрегаций . . . . .	30
<b>24 Случайный лес в Spark</b>	<b>31</b>
<b>25 Экстремальный градиентный бустинг с XGboost4j</b>	<b>31</b>
<b>26 Простой пример использования Kafka и программная реализация на Scala</b>	<b>33</b>
<b>27 Spark Streaming и Kafka</b>	<b>36</b>
<b>28 Распределенное глубокое обучение с Elephas</b>	<b>36</b>
<b>29 Spark и Microsoft Machine Learning</b>	<b>37</b>
<b>30 Ошибка java.lang.IllegalAccessException</b>	<b>37</b>
<b>31 Структурный и непрерывный стриминг</b>	<b>37</b>
<b>32 Оптимизация гиперпараметров и AutoML</b>	<b>38</b>
<b>33 Apache Zookeeper</b>	<b>38</b>
33.1 Общие сведения . . . . .	38
33.2 Установка и запуск Zookeeper . . . . .	38
<b>34 Apache Kafka</b>	<b>40</b>
34.1 Установка и запуск Kafka . . . . .	40
34.2 Общие сведения . . . . .	40
<b>35 Apache HBase</b>	<b>42</b>
35.1 Установка и запуск . . . . .	42
<b>36 Пакетная и потоковая обработка данных</b>	<b>43</b>
<b>37 Приемы работы со Spark в Apache Zeppelin</b>	<b>45</b>
<b>Список литературы</b>	<b>45</b>

## 1. Полезные ресурсы по Spark

Очень крутая книга *The Internals of Spark SQL* по внутреннему устройству Spark от Jacek Laskowski. GitHub-репозиторий [книги](#).

Официальная документация по ML <https://spark.apache.org/docs/1.2.2/ml-guide.html>.

Официальная документация по MLlib <https://spark.apache.org/docs/latest/ml-guide.html>.

Официальная документация по sbt <https://www.scala-sbt.org/1.x/docs/sbt-by-example.html>.

## 2. Общие сведения

Apache Spark – это универсальная и высокопроизводительная кластерная вычислительная платформа [1]. Благодаря разнопрофильным инструментам для аналитической обработки данных, Apache Spark активно используется в системах интернета вещей на стороне IoT-платформы, а также в различных бизнес-приложениях, в т.ч. на базе методов машинного обучения.

Apache Spark позиционируется как средство потоковой обработки больших данных в реальном времени. Однако, это не совсем так: в отличие, например, от Apache Kafka или Apache Storm, фреймворк Apache Spark разбивает непрерывный поток данных на набор *микро-пакетов*. Поэтому возможны некоторые временные задержки порядка секунды. Официальная документация утверждает, что это не оказывает большого влияния на приложения, поскольку в большинстве случаев аналитика больших данных выполняется не непрерывно, а с довольно большим шагом около пары минут.

Однако, если все же временная задержка обработки данных (latency) – это критичный момент для приложения, то Apache Spark Streaming не подойдет и стоит рассмотреть альтернативу в виде Apache Kafka Streams<sup>1</sup> (задержка не более 1 миллисекунды) или *фреймворков потоковой обработки больших данных* Apache Storm, Apache Flink и [Apache Samza](#).

В отличие от классического MapReduce<sup>2</sup>, реализованном в Apache Hadoop, Spark не записывает промежуточные данные на диск, а размещает их в оперативной памяти. Поэтому сервера, на которых развернут Spark, требуют большого объема оперативной памяти. Это в свою очередь ведет к удорожанию кластера.

Spark вращается вокруг концепции *устойчивого распределенного набора данных* (Resilient Distributed Dataset, RDD) <https://spark.apache.org/docs/latest/rdd-programming-guide.html>, который представляет собой отказоустойчивый набор элементов, с которыми можно работать *параллельно*.

Существует два способа создать RDD:

- распараллеливание существующего набора данных,
- на основе набора данных внешней системы хранения, такой как общая файловая система, HDFS, HBase или на основании любого другого источника, который поддерживает Hadoop.

Модуль [pyspark.sql.Session](#) является базовой «точкой входа» для работы с DataFrame и SQL. Класс SparkSession может использоваться для работы с объектом DataFrame, регистрации его как таблицы, выполнения SQL-запросов, кеширования таблиц и чтения parquet-файлов:

---

<sup>1</sup>Apache Kafka Streams – это клиентская библиотека для разработки *распределенных потоковых приложений* и *микросервисов*, в которых входные и выходные данные хранятся в кластерах Kafka. Поддерживает только Java и Scala

<sup>2</sup>Модель распределенных вычислений

```

In[]: from pyspark.conf import SparkConf
In[]: from pyspark.sql import SparkSession

In[]: spark = (
    SparkSession.
    builder. # создать экземпляр класса SparkSession
    master('local[4]'). # задает URL-адрес
                                # в данном случае подключается локально и использует 4 ядра
    appName('test app'). # задать наименование приложения
    config(conf=SparkConf()). # задать конфигурацию
    getOrCreate() # возвращает существующий сеанс Spark или, если его нет, создает
                    # новый сеанс на основе параметров, заданных в builder
)

```

### 3. Управление зависимостями проекта с помощью build.sbt

При работе со Scala-проектом с помощью sbt или IntelliJ IDEA версия языка определяется параметром `scalaVersion` в файле сборки `build.sbt`, например

```

scalaVersion := "2.12.12"
...

```

Остается только при запуске сессии в REPL набрать `sbt console` (а не `scala`), чтобы загрузить указанную версию Scala и все зависимости проекта.

В файл сборки `build.sbt` следует добавить следующие строки

Пример файла `build.sbt`

```

name := "SparkML"

version := "1.0"

scalaVersion := "2.12.12"

libraryDependencies += Seq(
  "org.apache.spark" %% "spark-sql" % "3.0.1" % "provided",
  "org.apache.spark" % "spark-mllib_2.12" % "3.0.1" % "provided" // в строке используется один
  "%"!!!
)

```

Для того чтобы sbt работал корректно, требуется разместить `AppFileName.scala` и `build.sbt` следующим образом:

- о файл `build.sbt` должен лежать в корне проекта,
- о а scala-скрипт – по пути `src/main/scala/AppFileName.scala`.

Scala-сценарий может лежать и «глубже», например, в `src/main/scala/com/clairvoyant/insight/bigdata`, но тогда в scala-сценарии нужно будет нестандартный путь описать как пакет

сценарий.scala

```

package com.clairvoyant.insight.bigdata
...

```

Теперь можно упаковать приложение

```
sbt package
```

В поддиректории `project` проекта будет файл с версией `sbt`

project/build.properties

```
sbt.version = 1.3.13
```

Там же можно расположить файл с описанием плагинов для `sbt`

project/plugins.sbt

```
addSbtPlugin("org.scalameta" % "sbt-scalafmt" % "2.4.0")
addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.14.10")
```

В корне проекта можно расположить конфигурационный файл для `scalafmt`

.scalafmt

```
version = "2.6.4"

align.preset = more // For pretty alignment
maxColumn = 100 // For my wide 30" display
```

Для запуска scala-приложения используется `spark-submit`

```
spark-submit \
--class "AppFileName" \
--master local \
target/scala-2.12/app-file-name_2.12-1.0.jar
```

## 4. Пример build.sbt для работы со Spark и Breeze

Файл сборки `build.sbt` для работы со Spark и внешними библиотеками

build.sbt

```
name := "Custom estimator"

scalaVersion := "2.12.12"

libraryDependencies ++= {
  val sparkVer = "3.0.1"
  Seq(
    "org.scalanlp" %% "breeze" % "1.1",
    "org.scalanlp" %% "breeze-natives" % "1.1",
    "org.scalanlp" %% "breeze-viz" % "1.1",
    "org.apache.spark" %% "spark-core" % sparkVer withSources(),
    "org.apache.spark" %% "spark-mllib" % sparkVer withSources(),
    "org.apache.spark" %% "spark-sql" % sparkVer withSources(),
    "org.apache.spark" %% "spark-streaming" % sparkVer withSources(),
    "com.esotericsoftware" % "kryo" % "4.0.1"
  )
}

libraryDependencies ++= Seq(
  "org.scalatest" %% "scalatest" % "3.0.6" % "test",
  "ml.dmlc" %% "xgboost4j" % "1.3.1",
  "ml.dmlc" %% "xgboost4j-spark" % "1.3.1"
)
```

## 5. Начало работы со Spark

Отправной точкой является `SparkSession` – создание распределенной системы для исполнения будущих вычислений

```
import org.apache.spark.sql.SparkSession
import spark.implicits._ // важный импорт; здесь много синтаксического сахара
val spark = SparkSession.builder()
    .appName("Example app")
    .master("local[*]")
    .getOrCreate()
```

Примеры использования Spark в ML можно найти здесь <https://github.com/apache/spark/tree/master/examples/src/main/scala/org/apache/spark/examples/ml>

Метод `.master(...)` (или `.setMaster(...)` в конфигурации `SparkContext`) указывает, где нужно выполнить вычисления. Например,

```
.master("yarn") // выполнение на кластере Hadoop
.master("local") // выполнение локально на машине
```

У Spark есть 3 разных API:

- RDD API,
- DataFrame API (он же SQL API): не типизирован,
- DataSet API (только для Scala! В Python это не имеет смысла): Scala-вский DataSet по сути представляет собой коллекцию экземпляров строк определенного типа (то есть это типизированный DataFrame); и поэтому, когда мы применяем например, метод `filter`, то он применяется к каждой строке.

Различаются они в основном тем, в каком виде представлены *распределенные коллекции* при вычислениях. На низком уровне все эти формы представления коллекций являются RDD.

Работу со Spark можно вести и через `spark-shell` (для Scala) или через `pyspark` (для Python).

Для реальных проектов требуется создать проект определенной структуры, например, так

```
sbt new MrPowers/spark-sbt.g8
```

а затем импортировать его в IntelliJ IDEA.

Затем нужно будет собрать проект в jar-файл, перенести этот файл на кластер и запустить `spark-submit` с полученным jar-файлом.

`SparkContext` – это предшественник `SparkSession` и используется для работы с RDD

### Scala

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

### Python

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

Сейчас к `SparkContext` напрямую обращаться не нужно. Лучше сразу создать `SparkSession`, а затем если вдруг возникнет необходимость из-под сессии вызывать контекст.

При построении DAG есть два типа операций:

- *Transformations* – описание вычислений (`map`, `filter`, `groupByKey` etc.),

- *Actions* – действия, запускающие расчеты (`reduce`, `collect`, `take` etc.).

Без *действий* вычисления не запускаются! Чтобы Spark каждый раз не вычислял весь граф заново, можно сказать `sc.textFile("...").cache()`.

Прочитать файлы (с заголовком) с локальной файловой системы в DataFrame можно так

#### Scala

```
val df = spark.read.option("header", true).csv("file.csv")
// или так
// требуется указывать полный абсолютный путь (~ не понимает)
val df = spark.read.option("header", true).csv("/Users/leor.finkelberg/Scala_projects/citibike.csv")
df.show()
val tf = spark.read.option("header", true).text("file.txt")
tf.head
```

Аналогично на Python

#### Python

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("test").master("local[*]").getOrCreate()
df = spark.read.option("header", True).csv("/Users/leor.finkelberg/Python_projects/file.csv")
```

Для того, чтобы типы полей файлов распознавались при загрузке можно использовать опцию `inferSchema`

```
val dataCsv = spark.read
    .option("header", "true")
    .option("inferSchema", "true") // <- NB
    .csv("filename.csv")
```

Результат будет таким

```
dataCsv.printSchema
root
|-- fieldname1: integer (nullable = true)
|-- fieldname2: double (nullable = true)
...
```

Можно передать сразу несколько пар с помощью `options` через ассоциативный массив

```
val df = spark.read.options(Map("delimiter"->"", "header"->"true")).csv("file.csv")
```

К слову, можно считать все csv-файлы из директории просто указав путь к ней

```
val collect_csv = spark.read.csv("folder_with_csv")
```

Аналогичным образом можно записать результат вычислений в файл

```
df.write.option("header", true).csv("from_spark.csv") // в текущей директории будет создана директория (!) from_spark_csv, в которой будет лежать csv-файл
// или
df.write.options(Map("header"->"true", "delimiter"->"", "header"->"true")).csv("from_spark_again.csv")
```

Дополнительно можно управлять поведением с помощью класса `SaveMode`

```
import org.apache.spark.sql.SaveMode

df.write.mode(SaveMode.Overwrite).csv("file.csv")
df.write.mode(SaveMode.ErrorIfExists).csv("file.csv")
...
```

В Spark лучше передавать НЕ csv-файлы (НЕ следует использовать!), а Parquet/ORC (наилучший вариант). Для потоковой обработки (или для случаев, когда не получается работать с колоночными данными) лучше использовать Avro вместо JSON.

Для того чтобы результаты вычислений, представленных в виде большого числа маленьких файлов, сохранить в виде одного относительно большого нужно провести репартиционирование

```
// hdfs не любит мелкие файлы!  
df.repartition(1).write.parquet("hdfs:///parquet-files/") // сжимаем до 1 партиции
```


Можно провести партицирование папками

```
df.write.partitionBy("year", "month").parquet("hdfs:///parquet-files/")
```

Для запуска приложения на кластере используется spark-submit

```
export HADOOP_CONF_DIR=...  
./bin/spark-submit \  
  --class org.apache.spark.examples.SparkPi \  
  --master yarn \  
  --deploy-mode cluster \  
  --executor-memory 20G \  
  --num-executors 50 \  
  /path/to/examples.jar 1000
```

Здесь 1000 – это аргумент, который попадет в наше приложение.

Найти скрипт spark-submit можно, например, здесь  HOME ▸ Anaconda3 ▸ Lib ▸ site-packages ▸ pyspark ▸ bin.

Основные аргументы spark-submit:

- `--driver-cores/--executor-cores` – количество ядер для каждого из элементов приложения (на контейнер!); executors выполняются в отдельных контейнерах; сколько будет контейнеров зависит от YARN,
- `--driver-memory/--executor-memory` – количество памяти для каждого из элементов приложения (на контейнер!),
- `--queue` – очередь в YARN, в которой будет выполняться приложение,
- `--num-executors` – количество executors (может быть динамическим)

Spark-приложение упаковывается в uber-jar (жирный jar), содержащий необходимые зависимости. Его можно располагать как на локальной файловой системе, так и на HDFS.

Такой jar можно собрать командой (нужен плагин `sbt-assembly`)

```
sbt assembly
```

Если хочется тащить с собой лишние зависимости, есть три варианта:

- `--jars` – указание пути к дополнительным jar-файлам,
- `--packages` – подключение зависимости из удаленных репозиториях (см. <https://spark-packages.org/>); полезно скорее для интерактивных приложений

```
--packages datastax:spark-cassandra-connector_2.11:2.0.7
```

- `CLASSPATH` – переменная окружения, в которой можно указать дополнительные jar-файлы.

Есть два режима деплоя приложения:

- `client` – драйвер запускается локально, executors – на кластере,
- `cluster` – драйвер, как и executors, запускается на кластере.



## 6. Эффективная конфигурация исполнителей

Подробности в статье <https://medium.com/expedia-group-tech/part-3-efficient-executor-configuration>

Первым шагом в определении эффективной конфигурации *исполнителей* (Executor) является выяснение числа *фактических* процессоров на узлах кластера. К примеру, пусть на каждом узле доступны 16 процессоров.

Здесь есть одна терминологическая тонкость. Иногда под процессорами или *виртуальными процессорами* понимают *ядра* процессора. Когда речь идет о кластере виртуальных машин о виртуальных процессорах все-таки удобнее думать как о *ядрах* процессора.

На каждом узле следует зарезервировать один процессор под операционную систему и диспетчер кластера. Таким образом, на каждом узле остается 15 процессоров.

Теперь, когда известно сколько процессоров доступно для использования на каждом рабочем узле, следует определить сколько ядер (cores) мы хотим назначить *каждому исполнителю*.

Итак, у нас в распоряжении 15 ядер на узел, поэтому возможны следующие варианты:

1. 1 исполнитель, 15 ядер на исполнителя: НЕ ГОДИТСЯ!!! Одноядерные исполнители не эффективны, так как не используют преимущества параллелизма, которые обеспечивают несколько ядер внутри одного исполнителя,
2. 3 исполнителя, 5 ядер на исполнителя: 5 ядер на одного исполнителя это оптимальное решение с точки зрения параллельной обработки; кроме того меньшее количество исполнителей на узел требует меньшее количество служебной памяти,
3. 5 исполнителей, 3 ядра на исполнитель: неплохо, но лучше 5 ядер и 3 исполнителя,
4. 15 исполнителей, 1 ядро на исполнителя: НЕ ГОДИТСЯ! Дело в том, что «жирные» исполнители, поддерживающие такое количество ядер, обычно имеют такой большой пул памяти (64 Гб+), что задержки на сборку мусора будут колоссальными.

Теперь требуется понять сколько памяти следует назначить каждому исполнителю. Для нужно выяснить сколько физической памяти доступно на узле. Пусть, для примера, на узел выделяется 128 Гб памяти.

Однако исполнителями будут доступны не все 128 Гб, так как память нужно выделить под операционную систему и диспетчер кластера. Пусть остается 112 Гб. Теперь нужно решить как распределить эту память по исполнителям.

Решение простое: доступный объем памяти следует равномерно разделить по исполнителям в узле с учетом коэффициента накладных расходов на память (в Spark этот коэффициент принимают равным 1.1), то есть каждому исполнителю следует выделить  $\frac{1}{1.1} \frac{112}{3} \approx 34$  (Гб).

Оптимальное количество исполнителей должно быть кратным 3 минус один исполнитель, чтобы освободить место под *драйвер* (по умолчанию выделяется 1 ядро).

Обычной практикой для data-инженеров является выделение относительно небольшого размера памяти под драйвер по сравнению с исполнителями. Однако AWS на самом деле рекомендует устанавливать размер памяти вашего драйвера таким же, как и у исполнителей.

По умолчанию количество ядер на драйвер равно одному. Однако, для больших кластеров (более 500 ядер Spark), можно повысить производительность, если количество ядер на драйвер установить в соответствии с количеством ядер на исполнителя.

Однако не стоит сразу менять дефолтное количество ядер в драйвере. Просто протестируйте это на своих наиболее крупных задачах, чтобы увидеть, ощутите ли вы прирост производительности.

Таким образом, базовая конфигурация исполнителей для узла из 16 ядер и 128 Гб памяти выглядит так

```
...
--driver-memory 34G \
--executor-memory 34G \
--num-executors (3x -1) \
--executor-cores 5
```

## 7. Особенности работы инфиксных математических операторов с коллекциями в Python и Scala

В Python *простой инфиксный* математический оператор (+, \* и пр.) всегда создает *новый объект* независимо от того является левый операнд изменяемым объектом или нет.

*Составной инфиксный* оператор (+=, например) изменяет левый операнд на месте, если последний относится к объектам изменяемого типа данных, и создает новый объект, а затем перепривязывает его к переменной, если левый операнд является объектом неизменяемого типа данных.

В Scala добавить элемент в *упорядоченную* коллекцию слева или справа можно так

```
import scala.collection.mutable.ListBuffer

val lst = List[Int]() // неизменяемый список
val lstBuf = ListBuffer[Int]() // изменяемый список
// между добавляемым элементом и оператором + не должно быть никаких символов кроме пробела
lst :+ 10 // возвращает новый объект. List(10)
20 += lst // возвращает новый объект. List(20)
20 += (lst :+ 10) // List(20, 10)
lstBuf :+ 10 // возвращает новый объект
```

Добавить сразу несколько элементов

```
lst ++ List(10, 20) // возвращает новый объект
lstBuf ++ List(10, 20) // возвращает новый объект
lstBuf += 10 // изменяет левый операнд. ListBuffer(10)
lst += 10 // ОШИБКА!!! List[Int] - неизменяемый объект. В Python новый объект просто перепривязывается к переменной
lstBuf ++= List(10, 20) // изменяет левый операнд. ListBuffer(10, 20)
lst ++= List(10, 20) // ОШИБКА!!!
```

В Python *составной инфиксный оператор* (например, +=) может работать как с *изменяемыми*, так и с *неизменяемыми* операндами слева от себя без каких-либо специальных преобразований (просто в случае изменяемых объектов оператор += изменяет левый операнд на месте, а в случае неизменяемых объектов создает новый объект и перепривязывает его к переменной).

В Scala составной инфиксный оператор тоже может работать как с объектами *изменяемого* типа данных (например, `ListBuffer`), так и с объектами *неизменяемого* типа данных (например, `immutable.Map`), но в последнем случае изменяемые объекты должны быть объявлены как *var-переменные*.

Оператор ++ в Scala ведет себя также, как метод `extend()` в Python, т.е. ожидает увидеть объект, поддерживающий протокол итераций (или другими словами коллекцию). Затем эта коллекция распаковывается и элементы добавляются по одному. Но при этом ++ возвращает *новый* объект, а не изменяет существующий, даже в случае `ListBuffer`.

Оператор `+=` в Scala по отношению к *изменяемым* объектам ведет себя также как метод `append()` в Python и изменяет левый операнд.

Оператор `++=` в Scala по отношению к *изменяемым* объектам ведет себя также как метод `extend()` в Python и изменяет левый операнд.

Для добавления одного элемента в неупорядоченную коллекцию (например, во множество) используется оператор `+`, а для добавления нескольких элементов – оператор `++`.

ВАЖНО: массив `Array` в Scala строго говоря является *изменяемым* объектом фиксированной длины, а буферный массив `ArrayBuffer` – *изменяемым* объектом переменной длины. То есть и в том, и в другом случае можно обращаться к элементам массива и *изменять* их значение.

Пример

```
val arr = new Array[Int](3) // Array(0, 0, 0). ВНИМАНИЕ! ключевое слово new
arr(0) += 10 // Array(10, 0, 0)
arr(2) += 100 // Array(10, 0, 100)
arr(1) = -1 // Array(10, -1, 100)
```

## 8. Пример использования группировки и агрегации на Scala

Пример подсчета слов в файле

Не самый удачный вариант

```
> val lines = spark.read.text("file_name.txt")
> val linesMap = lines.flatMap(_.mkString.split(" ")).map( (_, 1))
// после этой операции имена столбцов будут иметь вид "_1", "_2" и т.д.
// чтобы переименовать столбцы придется воспользоваться следующей конструкцией
> val renameCols = Map("_1" -> "Language", "_2" -> "Numbers")
> val linesMapRen = linesMap
  .select(linesMap.columns.map(c => col(c))
    .as(renameCols.getOrElse(c, c))): _*)
> val wordCounts = linesMapRen.groupBy("Language").agg(sum("Counts"))
> wordCounts.show
```

Тоже самое одним запросом с фильтрацией по числу слов

```
linesMap
  .select(
    linesMap.columns.map(
      c => col(c).as(renameCols.getOrElse(c, c))
    ): _* // обязательно распаковать
  )
  .groupBy("Language")
  .agg(sum("Counts"))
  .where($"sum(Counts)" === 2)
  .show
```

К слову, переименовать столбцы в объекте `DataFrame` можно и проще

```
val df = ... // "_1", "_2", "_3"
val dfRenamed = df.toDF("newName1", "newName2", "newName3")
```

Или так (пожалуй это самый естественный способ)

```
val dfRenamed = df.withColumnRenamed("oldName", "newName")
```

## 9. Приемы работы со объектами Spark DataFrame

Пусть есть набор данных

```
// читаем в объект DataFrame
val df = spark.read.option("header", "true").csv("state-population.csv")
// выводим схему
df.printSchema
// root
// |-- state: string (nullable = true)
// |-- ages: string (nullable = true)
// |-- year: string (nullable = true)
// |-- population: string (nullable = true)
```

Теперь пусть требуется вывести столбцы «year» и «population» с фильтрацией по столбцам «state» и «year» и упорядочить по столбцу «year»

```
df.select($"year", $"population")\
  .where($"state" === "AL" && $"year" > 2010)
  .orderBy($"year")
  .show
```

Можно еще зарегистрировать DataFrame-объект как таблицу и получить тот же самый результат с помощью SQL-запроса

```
df.createOrReplaceTempView("state_population")
spark.sql(
  "SELECT year, population FROM state_population WHERE state = 'AL' and year > 2010 ORDER BY
  1;"
).show
```

или с логической разбивкой по строкам

```
spark.sql(
  "SELECT year, population " +
  "FROM state_population " +
  "WHERE state = 'AL' and year > 2010 " +
  "ORDER BY 1"
).show
```

К слову на Pandas эта задача решалась бы так

```
df[(df["year"] > 2010) & (df["state"] == "AL")][["year", "population"]].sort_values("year")
# или с помощью метода query
df.query("year > 2010 & state == 'AL')[["year", "population"]].sort_values("year")
```

## 10. Запись объектов Spark DataFrame

Если все данные хранятся в одном объекте DataFrame, то можно легко задать информацию о секционировании во время записи данных с помощью API класса DataFrameWriter. Функция partitionBy принимает в качестве параметра список столбцов, по которым необходимо секционировать результаты

```
// пусть у DataFrame есть категориальный столбец solverType с двумя категориями direct и
  iterative
data
  .write
  .partitionBy("solverType") // секционирование
```

```
.format("csv")
.save("output")
```

После этой операции в текущей директории будет создана поддиректория **output** следующего содержания

```
$ ls -l output
-rw-r--r--  1 leor.finkelberg  staff      0B 27 янв 19:47 _SUCCESS
drwxr-xr-x  6 leor.finkelberg  staff    192B 27 янв 19:47 solverType=direct/
drwxr-xr-x  4 leor.finkelberg  staff    128B 27 янв 19:47 solverType=iterative/
$ ls -l output/solverType=direct
-rw-r--r--  1 leor.finkelberg  staff     11B 27 янв 19:47 part-00001-66f80308-b856-4661-9e8f-
d1ea8df84d1a.c000.csv
-rw-r--r--  1 leor.finkelberg  staff      9B 27 янв 19:47 part-00003-66f80308-b856-4661-9e8f-
d1ea8df84d1a.c000.csv
$ cat output/solverType=direct/part-00001-66f80308-b856-4661-9e8f-d1ea8df84d1a.c000.csv
Nastran,10
```

## 11. Выбор элементов в DataFrame и Dataset

При выборе элементов коллекции Dataset с помощью метода **select** без указания типа полей возвращается объект DataFrame

```
df.select(
  $"packageName",
  $"solverType"
) // org.apache.spark.sql.DataFrame
```

Если же указать тип поля при выборе, то будет возвращен объект Dataset

```
df.select(
  $"packageName".as[String],
  $"solverType".as[String]
) // org.apache.spark.sql.Dataset[(String, String)]
```

И если выбрать поля из объекта DataFrame методом **select** без указания типа полей, то будет возвращен объект DataFrame. Если же указать тип полей, то вернется объект Dataset.

К слову, у наборов Dataset отсутствуют простые агрегирующие функции типа **min**, **max** и т.п., поэтому задавать такие функции, следует через **agg**

```
df.groupBy("solverType").agg(
  min("counts").as[Int].as("minCounts")
)
```

## 12. Создание объектов Spark DataFrame на Scala

Примеры создания объектов DataFrame на базе различных примитивных структур

```
// Обычная коллекция кортежей
val seqTuples = Seq(
  ("Nastan", 10, "direct"),
  ("Ansys", 20, "iterative")
)

seqTuples.toDF("packageName", "count", "solverType") // org.apache.spark.sql.DataFrame
```

## 13. Создание объектов Spark DataFrame на Python

### 13.1. Создание Spark DataFrame на основе списка

Создание объекта Spark DataFrame на основе списка

```
In[]: packages = [
    ('Ansys', 'direct', 15550),
    ('Nastran', 'iterative', 40000),
    ('Comsol', 'direct', 45000)
]

In[]: spark.createDataFrame(packages, ['package_name', 'solver', 'price']).collect()
Out[]:
[Row(package_name='Ansys', solver='direct', price=15550),
 Row(package_name='Nastran', solver='iterative', price=40000),
 Row(package_name='Comsol', solver='direct', price=45000)]
```

### 13.2. Создание Spark DataFrame на основе объекта RDD

Создание объекта DataFrame на основе объекта RDD

```
# RDD
In[]: lst = [('Alice', 18),
            ('Jhon', 22),
            ('Alex', 48)]
In[]: sc = spark.sparkContext # <--

In[]: rdd = sc.parallelize(lst) # pyspark.rdd.RDD; Resilient Distributed Dataset
In[]: spark.createDataFrame(rdd).collect()

In[]: df = spark.createDataFrame(rdd, ['name', 'age'])
In[]: df.collect()
# [Row(name='Alice', age=18),
#  Row(name='Jhon', age=22),
#  Row(name='Alex', age=48)]
```

### 13.3. Создание Spark DataFrame на основе схемы StructType()

Создание объекта DataFrame на основе схемы

```
# schema
In[]: from pyspark.sql.types import (StringType, IntegerType,
                                     StructField, StructType)

In[]: schema = StructType([
    StructField('name', StringType(), True), # поле 'name'
    StructField('age', IntegerType(), True)  # поле 'age'
])

In[]: df = spark.createDataFrame(rdd, schema)

In[]: df.collect()
# [Row(name='Alice', age=18),
#  Row(name='Jhon', age=22),
#  Row(name='Alex', age=48)]
```

## 13.4. Создание Spark DataFrame на основе pandas

Создание объекта Spark DataFrame на основе pandas DataFrame

```
In[]: data = pd.read_csv('file.csv')
In[]: df_spark = spark.createDataFrame(data).collect()
```

Использование SQL-запросов с объектами Spark DataFrame

```
In[]: type(df) # pyspark.sql.dataframe.DataFrame
In[]: df.collect()
Out[]:
# [Row(url='url1', ts='2018-08-15 00:00:00', service='tw', delta=1),
#  Row(url='url1', ts='2018-08-15 00:05:00', service='tw', delta=3),
#  Row(url='url1', ts='2018-08-15 00:11:00', service='tw', delta=1),
#  ...
#  Row(url='url2', ts='2018-08-15 00:26:00', service='fb', delta=13)]

In[]: df.createOrReplaceTempView('social_delta_tab') # создать временную таблицу
                                                # с именем 'social_delta_tab'

In[]: sql_result = spark.sql('''
        SELECT url, service, sum(delta) AS summa
        FROM social_delta_tab
        GROUP BY url, service
    ''')

In[]: sql_result.collect() # результат SQL-запроса
Out[]:
[Row(url='url1', service='fb', summa=360),
 Row(url='url2', service='tw', summa=1200),
 Row(url='url2', service='fb', summa=38),
 Row(url='url1', service='tw', summa=59)]
```

## 14. Зарегистрировать пользовательскую функцию в Spark на Scala

Зарегистрировать пользовательскую функцию для использования в контексте SQL-запросов можно так

```
spark.udf.register("power2", (elem: Int) => scala.math.pow(elem, 2))
\end{lstlisting}

Теперь можно использовать эту функцию внутри SQL-запроса
\begin{lstlisting}[
style = scala,
numbers = none
]
df.createOrReplaceTempView("test") // регистрируем DataFrame как временное представление
spark.sql("SELECT power2(counts) FROM test;")
```

## 15. Зарегистрировать пользовательскую функцию в PySpark

Зарегистрировать пользовательскую функцию

```
In[]: power_2 = spark.udf.register('power_2', lambda x: x**2)
In[]: spark.sql("SELECT power_2(11)").collect() # [Row(power_2(11)=121)]

In[]: from pyspark.sql.types import IntegerType
In[]: stringLength = spark.udf.register('stringLength', lambda x: len(x), IntegerType())
```



```
In[]: spark.sql("SELECT stringLength('test')").collect() # [Row(stringLength(test)=4)]
```

## 16. Соединение объектов DataFrame

Соединение двух объектов DataFrame/Dataset можно выполнить следующим образом

```
import spark.sqlContext.implicits._
import org.apache.spark.sql.catalyst.plans.{LeftOuter, Inner, RightOuter}

val emp = Seq((1, "Smith", -1, "2018", "10", "M", 3000),
  (2, "Rose", 1, "2010", "20", "M", 4000),
  (3, "Williams", 1, "2010", "10", "M", 1000),
  (4, "Jones", 2, "2005", "10", "F", 2000),
  (5, "Brown", 2, "2010", "40", "", -1),
  (6, "Brown", 2, "2010", "50", "", -1)
)

val empColumns = Seq("emp_id", "name", "superior_emp_id", "year_joined",
  "emp_dept_id", "gender", "salary")

val empDF = emp.toDF(empColumns:_) // один DataFrame
empDF.show(false)

val dept = Seq(("Finance", 10),
  ("Marketing", 20),
  ("Sales", 30),
  ("IT", 40)
)

val deptColumns = Seq("dept_name", "dept_id")
val deptDF = dept.toDF(deptColumns:_) // второй DataFrame
deptDF.show(false)
```

Внутреннее соединение по ключам emp\_dept\_id и dept\_id

```
empDF.join(deptDF, empDF("emp_dept_id") === deptDF("dept_id"), "inner")
.show(false)
```

Внешнее соединение

```
// это синонимические конструкции
empDF.join(deptDF, empDF("emp_dept_id") === deptDF("dept_id"), "outer").show(false)
empDF.join(deptDF, empDF("emp_dept_id") === deptDF("dept_id"), "full").show(false)
empDF.join(deptDF, empDF("emp_dept_id") === deptDF("dept_id"), "fullouter").show(false)
```

Аналогично выполняется левое и правое соединение

```
empDF.join(deptDF, empDF("emp_dept_id") === deptDF("dept_id"), "left").show
empDF.join(deptDF, empDF("emp_dept_id") === deptDF("dept_id"), "right").show
```

Еще возможно выполнить *полулевое* соединение. Этот вариант соединения ведет себя аналогично внутреннему соединению, но возвращает только столбцы из левого объекта DataFrame (столбцы из правого объекта DataFrame игнорируются)

```
empDF.join(deptDF, empDF("emp_dept_id") === deptDF("dept_id"), "leftsemi").show(false)
```

Соединение типа «left anti» оставляет только столбцы левого объекта DataFrame и только те строки, которые не совпадают с правыми

```
empDF.join(deptDF, empDF("emp_dept_id") === deptDF("dept_id"), "leftanti").show
```



Самосоединение выполняется так

```
empDF.as("emp1").join(
    empDF.as("emp2"), $"emp1.superior_emp_id" === $"emp2.emp_id", "inner"
).select(
    $"emp1.emp_id",
    $"emp1.name",
    $"emp2.emp_id".as("superior_emp_id"),
    $"emp2.name".as("superior_emp_name")
).show
```

Разумеется можно выполнить соединение из-под SQL-запроса

```
spark.sql("SELECT * FROM EMP AS e INNER JOIN DEPT AS d ON e.emp_dept_id = d.dept_id").show
```

## 17. Фильтрация и агрегация

Конструкция запроса Spark очень похожа на конструкцию pandas

```
In[]: df_spark = spark.createDataFrame(pd.read_csv('data.csv'))
In[]: df_spark.filter(df_spark.delta >= 30).collect()
# или так
In[]: df_spark.where(df_spark.delta >= 30).collect()

In[]: (df_spark.filter(df_spark.delta >= 30).
      groupBy('url').agg({'delta' : 'sum'}).
      collect() # возвращает все записи в формате списка строк Row()
)
Out[]:
[Row(url='url1', sum(delta)=293),
 Row(url='url2', sum(delta)=1180)]
```

Пример агрегации в PySpark с помощью SQL-запроса

```
In[]: spark.sql('''
      SELECT gender,
             usertype,
             max(tripduration)
      FROM data
      GROUP BY gender, usertype
      ORDER BY gender
      ''').show()
Out[]:
+-----+-----+-----+
|gender| usertype|max(tripduration)|
+-----+-----+-----+
|    0| Customer|          126180|
|    0|Subscriber|           342|
|    1|Subscriber|          40339|
|    2|Subscriber|          15905|
+-----+-----+-----+
```

В pandas решение этой задачи может быть записано в виде

```
In[]: (data.groupby(['gender', 'usertype']).
      agg(np.
           max))
Out[]:
           tripduration
gender usertype
0      Customer      126180
      Subscriber       342
1      Subscriber      40339
2      Subscriber      15905
```

## 18. Сводная информация

```
In[]: df_spark.describe().show()
Out[]:
```

```

+-----+-----+-----+-----+-----+
|summary| url|                ts|service|                delta|
+-----+-----+-----+-----+-----+
| count| 30|                30|    30|                30|
| mean|null|                null| null|55.233333333333334|
| stddev|null|                null| null|140.58049193484734|
| min|url1|2018-08-15 00:00:00| fb|                1|
| max|url2|2018-08-15 00:41:00| tw|                645|
+-----+-----+-----+-----+-----+

In[]: df_spark.describe(['url']).show()
Out[]:
+-----+-----+
|summary| url|
+-----+-----+
| count| 30|
| mean|null|
| stddev|null|
| min|url1|
| max|url2|
+-----+-----+

```

## 19. Оконные функции в контексте SQL и Spark DataFrame

### 19.1. Базовые понятия оконных преобразований в PySpark

Spark SQL поддерживает три вида оконных функций (см. табл. 1):

- ранжирующие,
- аналитические,
- агрегатные (любую агрегатную функцию<sup>3</sup> можно использовать в качестве оконной функции)

Чтобы использовать оконную функцию, следует указать, что функция должна использоваться как *оконная* одним из следующих способов:

- добавить ключевое слово **OVER** после функции поддерживаемой SQL, например, **AVG(revenue) OVER (...)** или
- вызвать метод **over**, например, **rank().over(...)**.

Итак, функция «помечена» как оконная. Теперь можно определить спецификацию окна. Спецификация окна включает три части:

- спецификация секционирования (группировка строк): определяет какие строки будут входить в одну группу,
- спецификация сортировки: определяет в каком порядке будут располагаться строки в группе,
- спецификация фрейма: определяет какие строки будут включены в фрейм для текущей строки, основываясь на их положении относительно текущей строки.

В контексте SQL ключевые слова **PARTITION BY** и **ORDER BY** используются для определения групп в *спецификации секционирования* и *спецификации сортировки*, соответственно

```
OVER (PARTITION BY ... ORDER BY ...)
```

В контексте DataFrame API *оконную функцию* можно объявить следующим образом

<sup>3</sup>Например, AVG, SUM, COUNT и пр.

Таблица 1. Ранжирующие и аналитические функции *PySpark*

	контекст SQL	DataFrame API
Ранжирующие функции	rank	rank
	dense_rank	denseRank
	percent_rank	percentRank
	ntile	ntile
	row_number	rowNumber
Аналитические функции	cume_dist	cumeDist
	first_value	firstValue
	last_value	lastValue
	lag	lag
	lead	lead

```
from pyspark.sql.window import Window

windowSpec = Window.partitionBy(...).orderBy(...)
```

Дополнительно требуется определить:

- начальную границу фрейма,
- конечную границу фрейма,
- тип фрейма.

Существует пять типов границ:

- UNBOUNDED PRECEDING: первая строка в группе,
- UNBOUNDED FOLLOWING: последняя строка в группе,
- CURRENT ROW: текущая строка,
- <value> PRECEDING: ,
- <value> FOLLOWING.

Различают два типа фреймов:

- строковый фрейм **ROWframe**: базируется на физическом смещении относительно текущей строки. Если в качестве границы используется CURRENT ROW, то это означает, что речь идет о текущей строке. <value> PRECEDING и <value> FOLLOWING указывают число строк до и после текущей строки, соответственно.
- диапазонный фрейм **RANGEframe**: базируется на логическом смещении относительно положения текущей строки.

Visual representation of frame  
ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING

	product	category	revenue
	Bendable	Cell phone	3000
	Foldable	Cell phone	3000 <- 1 PRECEDING
Current input row ->	Ultra thin	Cell phone	6000
	Thin	Cell phone	6000 <- 1 FOLLOWING
	Very thin	Cell phone	6000

Рассмотрим работу **RANGEframe**. Рассмотрим пример. В этом примере сортировка проводится по «revenue», в качестве начальной границы используется в 2000 PRECEDING, в качестве конечной границы - 1000 FOLLOWING.

В контексте SQL этот фрейм определяется как

## RANGE BETWEEN 2000 PRECEDING AND 1000 FOLLOWING

Границы фрейма вычисляются следующим образом: [current revenue value - 2000; current revenue value + 1000], т.е. границы фрейма пересчитываются в зависимости от текущего значения строки в столбце «revenue»

Visual representation of frame  
RANGE BETWEEN 2000 PRECEDING AND 1000 FOLLOWING  
(ordering expression: revenue)

```
# 1 step
      product | category | revenue
      -----+-----+-----
Current input row -> Bendable | Cell phone | 3000 <-- revenue range [3000-2000=1000;
      3000+1000=4000]
      Foldable | Cell phone | 3000 <--
      Ultra thin | Cell phone | 5000
      Thin       | Cell phone | 6000
      Very thin  | Cell phone | 6000

# 2 step
      product | category | revenue
      -----+-----+-----
      3000+1000=4000]
Current input row -> Bendable | Cell phone | 3000 <-- revenue range [3000-2000=1000;
      Foldable | Cell phone | 3000 <--
      Ultra thin | Cell phone | 5000
      Thin       | Cell phone | 6000
      Very thin  | Cell phone | 6000

# 3 step
      product | category | revenue
      -----+-----+-----
      5000+1000=6000]
Current input row -> Bendable | Cell phone | 3000 <-- revenue range [5000-2000=3000;
      Foldable | Cell phone | 3000 <--
      Ultra thin | Cell phone | 5000 <--
      Thin       | Cell phone | 6000 <--
      Very thin  | Cell phone | 6000 <--

# 4 step
      product | category | revenue
      -----+-----+-----
      6000+1000=7000]
Current input row -> Bendable | Cell phone | 3000
      Foldable | Cell phone | 3000
      Ultra thin | Cell phone | 5000 <-- revenue range [6000-2000=4000;
      Thin      | Cell phone | 6000 <--
      Very thin  | Cell phone | 6000 <--

# 5 step
      product | category | revenue
      -----+-----+-----
      6000+1000=7000]
Current input row -> Bendable | Cell phone | 3000
      Foldable | Cell phone | 3000
      Ultra thin | Cell phone | 5000 <-- revenue range [6000-2000=4000;
      Thin      | Cell phone | 6000 <--
      Very thin  | Cell phone | 6000 <--
```

Итак, чтобы определить спецификацию окна в контексте SQL используется конструкция

```
OVER (PARTITION BY ... ORDER BY ... frame_type BETWEEN start AND end)
```

где `frame_type` может быть либо `ROWS` (`ROWframe`), либо `RANGE` (`RANGEframe`); `start` может принимать одно из следующих значений `UNBOUNDED PRECEDING`, `CURRENT ROW`, `<value> PRECEDING` и `<value> FOLLOWING`; `end` может принимать `UNBOUNDED FOLLOWING`, `CURRENT ROW`, `<value> PRECEDING` и `<value> FOLLOWING`.

В контексте `DataFrame` API используется следующий шаблон

```
In[]: windowSpec = Window.partitionBy(...).orderBy(...)
In[]: windowSpec.rowsBetween(start, end) # для ROW frame
In[]: windowSpec.rangeBetween(start, end) # для RANGE frame
```

Рассмотрим другой пример

```
In[]: from pyspark.sql.functions import pandas_udf, PandasUDFType
In[]: from pyspark.sql import Window

In[]: df = spark.createDataFrame(
    [(1, 1.0), (1, 2.0), (2, 3.0), (2, 5.0), (2, 10.0)],
    ('id', 'v')
)

In[]: @pandas_udf('double', PandasUDFType.GROUPED_AGG)
def mean_udf(v):
    return v.mean()
# оконное преобразование
In[]: w = Window.partitionBy('id').rowsBetween(Window.unboundedPreceding, Window.unboundedFollowing)
In[]: df.withColumn('mean_v', mean_udf(df['v']).over(w)).show()
Out[]:
+---+-----+
| id | v | mean_v |
+---+-----+
| 1 | 1.0 | 1.5 |
| 1 | 2.0 | 1.5 |
| 2 | 3.0 | 6.0 |
| 2 | 5.0 | 6.0 |
| 2 | 10.0 | 6.0 |
+---+-----+
```

Построить кумулятивную сумму для каждой группы `PARTITION BY` (первый элемент столбца `delta` используется в качестве первого элемента нового столбца `total`, затем первый элемент столбца `delta` суммируется со вторым элементом этого же столбца, а результат записывается как второй элемент столбца `total` и т.д.)

```
In[]: df = spark.createDataFrame(pd.read_csv('social_delta.csv'))
In[]: df.createOrReplaceTempView('social_del_tab')
In[]: spark.sql('''
    SELECT *,
        sum(delta) OVER (PARTITION BY url, service ORDER BY ts) AS total
    FROM social_del_tab
''').show(3)
Out[]:
+---+-----+-----+-----+-----+
| url | ts | service | delta | total |
+---+-----+-----+-----+-----+
| url1 | 2018-08-15 00:00:00 | fb | 5 | 5 | # <- 5
| url1 | 2018-08-15 00:05:00 | fb | 15 | 20 | # <- 5 + 15 = 20
| url1 | 2018-08-15 00:11:00 | fb | 11 | 31 | # <- 20 + 11 = 31
+---+-----+-----+-----+-----+
only showing top 3 rows
```

Вычислить скользящее среднее для каждой группы PARTITION BY

```
In[]: df = spark.createDataFrame(pd.read_csv('social_totals.csv'))
In[]: df.createOrReplaceTempView('social_tot_tab')

In[]: df = spark.sql('''
    SELECT *,
        AVG(total) OVER (PARTITION BY url, service ORDER BY ts
        ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) AS total_avg3
    FROM social_tot_tab
''').show(3)

Out[]:
+---+-----+-----+-----+-----+
| url|          ts|service|total|          total_avg3|
+---+-----+-----+-----+-----+
|url1|2018-08-15 00:00:00|fb|5|5.0| # <- 5/1 = 5
|url1|2018-08-15 00:05:00|fb|20|12.5| # <- (5 + 20)/2 = 12.5
|url1|2018-08-15 00:11:00|fb|31|18.666666666666668| # <- (5 + 20 + 31)/3 = 18.666
+---+-----+-----+-----+-----+
only showing top 3 rows
```

Вычислить скользящее среднее для каждой группы, включая записи, которые отстают от текущей записи на «5 мин назад»

```
In[]: df = spark.createDataFrame(pd.read_csv('social_totals.csv', parse_dates=['ts']))
In[]: df.createOrReplaceTempView('df')

In[]: spark.sql('''
    SELECT *, AVG(total) OVER (PARTITION BY url, service ORDER BY ts
    RANGE BETWEEN INTERVAL 5 MINUTES PRECEDING AND CURRENT ROW) AS total_avg5min
    FROM df
''').show(3)

Out[]:
+---+-----+-----+-----+-----+
| url|          ts|service|total|total_avg5min|
+---+-----+-----+-----+-----+
|url1|2018-08-15 00:00:00|fb|5|5.0| # <- 5
|url1|2018-08-15 00:05:00|fb|20|12.5| # <- (5 + 20)/2 = 12.5 (5 мин)
|url1|2018-08-15 00:11:00|fb|31|31.0| # <- 31 (6 мин)
|url1|2018-08-15 00:18:00|fb|45|45.0| # <- 45 (7 мин)
|url1|2018-08-15 00:21:00|fb|59|52.0| # <- (45 + 59)/2 = 52 (3 мин)
|url1|2018-08-15 00:30:00|fb|67|67.0|
+---+-----+-----+-----+-----+
only showing top 6 rows
```

Ту же задачу в pandas можно решить следующим образом

```
In[]: df = pd.read_csv('social_totals.csv', parse_dates=['ts'])
In[]: df.groupby(['url', 'service']).rolling('5min', on='ts', min_periods=1).mean().reset_index(
    drop=True)

Out[]:
          ts  total
0 2018-08-15 00:00:00    5.0
1 2018-08-15 00:05:00   20.0
2 2018-08-15 00:11:00   31.0
3 2018-08-15 00:18:00   45.0
4 2018-08-15 00:21:00   52.0
5 2018-08-15 00:30:00   67.0
```

Пусть задан объект PySpark DataFrame

```
In[]: productRevenue = spark.createDataFrame([
    ('Thin', 'Cell phone', 6000),
    ('Normal', 'Tablet', 1500),
    ('Mini', 'Tablet', 5500),
    ('Ultra thin', 'Cell phone',
     5000),
    ('Very thin', 'Cell phone',
     6000),
    ('Big', 'Tablet', 2500),
    ('Bendable', 'Cell phone',
     3000),
    ('Foldable', 'Cell phone',
     3000),
    ('Pro', 'Tablet', 4500),
    ('Pro2', 'Tablet', 6500)],
    ['product', 'category', 'revenue'])
```

```
In[]: productRevenue.show()
Out[]:
```

```
+-----+-----+-----+
| product| category|revenue|
+-----+-----+-----+
|      Thin|Cell phone|  6000|
|    Normal|   Tablet|  1500|
|      Mini|   Tablet|  5500|
|Ultra thin|Cell phone|  5000|
| Very thin|Cell phone|  6000|
|       Big|   Tablet|  2500|
|Bendable|Cell phone|  3000|
|Foldable|Cell phone|  3000|
|       Pro|   Tablet|  4500|
|      Pro2|   Tablet|  6500|
+-----+-----+-----+
```

Требуется выявить первые два наименования наиболее дорогих продуктов из групп «Cell phone» и «Tablet».

Решение этой задачи на основе оконных функций может выглядеть следующим образом

```
In[]: productRevenue.createOrReplaceTempView('prod_rev')
In[]: spark.sql('''
    SELECT
        product,
        category,
        revenue
    FROM (
        SELECT
            *,
            dense_rank() OVER (PARTITION BY category ORDER BY revenue DESC) AS rank
        FROM prod_rev)
    WHERE rank <= 2''').show()
```

Out[] :

```
+-----+-----+-----+
| product| category|revenue|
+-----+-----+-----+
|      Thin|Cell phone|  6000| # <- first group
| Very thin|Cell phone|  6000|
|Ultra thin|Cell phone|  5000|
|      Pro2|   Tablet|  6500| # <- second group
|      Mini|   Tablet|  5500|
+-----+-----+-----+
```

То есть к каждой найденной группе применяется функция `dense_rank` с помощью `PARTITION BY` выполняется группировка по столбцу «category». Внутри группа упорядочивается по убыванию (`ORDER BY`) по столбцу «revenue».

Пусть теперь требуется вычислить на сколько отличается по стоимости самый дорогой продукт в группе от прочих продуктов из той же группы. Задача может быть решена так

```
In[]: import sys
In[]: from pyspark.sql.window import Window
In[]: import pyspark.sql.functions as func

In[]: df = productRevenue

In[]: windowSpec = (
    Window.partitionBy(df['category']).
    orderBy(df['revenue'].desc()).
    rangeBetween(-sys.maxsize, sys.maxsize))

In[]: revenue_diff = func.max(df['revenue']).over(windowSpec) - df['revenue']

In[]: df.select( # выбрать из объекта df соответствующие столбцы
    df['product'],
    df['category'],
    df['revenue'],
    revenue_diff.alias('revenue_diff') # добавить в вывод этот столбец
).show()

Out[]:
+-----+-----+-----+-----+
| product| category|revenue|revenue_diff|
+-----+-----+-----+-----+
|      Thin|Cell phone|  6000|           0| # <- первая группа
| Very thin|Cell phone|  6000|           0|
|Ultra thin|Cell phone|  5000|          1000|
| Bendable|Cell phone|  3000|          3000|
| Foldable|Cell phone|  3000|          3000|
|      Pro2|   Tablet|  6500|           0| # <- вторая группа
|      Mini|   Tablet|  5500|          1000|
|       Pro|   Tablet|  4500|          2000|
|       Big|   Tablet|  2500|          4000|
|   Normal|   Tablet|  1500|          5000|
+-----+-----+-----+-----+
```

## 19.2. Оконные преобразования в Spark на Scala

Чтобы использовать функцию как оконную требуется лишь: 1) задать спецификацию окна и 2) вызвать метод `over` на функции, которую нужно сделать *оконной*, и передать спецификацию окна как аргумент этому методу.

Затем, например, с помощью метода `withColumn` создаем новый столбец по шаблону

```
val windowSpec = Window.partitionBy(...).orderBy(...).rowsBetween(-2, 2) // например!
df.withColumn("colName", function().over(windowSpec))
```

Простой пример

```
import spark.implicits._
import org.apache.spark.sql.functions._
import org.apache.spark.sql.expressions.Window // NB
```



```
// описываем спецификацию окна
val windowSpec = Window.partitionBy("department").orderBy("salary")
// создаем новый столбец row_number
df.withColumn(
  "row_number",
  row_number().over(windowSpec) // метод over делает функцию оконной
).show
```

Если нужно сместить вниз в пределах группы отсортированный набор элементов этой группы, то можно воспользоваться функцией `lag` (функция `lead` смещает вверх)

```
df.withColumn(
  "lag",
  lag("salary", 2).over(windowSpec)
).show
```

Можно использовать несколько спецификаций окна в одном запросе

```
// описываем новую спецификацию окна
val windowSpecAgg = Window.partitionBy("department") // для агрегатов не нужно указывать orderBy

df.withColumn(
  "row",
  row_number().over(windowSpec)
).withColumn(
  "avg",
  avg($"salary").over(windowSpecAgg)
).withColumn(
  "sum",
  sum($"salary").over(windowSpecAgg)
).select("department", "avg", "sum").show
```

Естественно можно оперировать вычисленными столбцами через, например, `withColumn`

```
df.withColumn(
  "avgWinSalary",
  avg($"salary").over(windowSpec) // оконная функция
).withColumn(
  "diffSalary",
  $"salary" - $"avgWinSalary" // вычисляем разницу
).show
```

Можно использовать результаты динамических вычислений

```
df.withColumn(// создаем новый столбец
  "diff",
  $"salary" - avg($"salary").over(windowSpec) // оконная функция
).show
```

## 20. Работа с файловой системой Databricks

Databricks <https://databricks.com/product/unified-data-analytics-platform> – это платформа для анализа больших данных, построенная вокруг Apache Spark. DBFS – распределенная файловая система Databricks.

Работа с файловой системой в рамках платформы Databricks осуществляется через модуль `dbutils`

```
# вывести список файлов текущей директории
dbutils.fs.ls('dbfs:/FileStore/tables')
# удалить файл из DBFS
dbutils.fs.rm('dbfs:/FileStore/tables/file_name.csv', True)
```

Записать Spark-объект DataFrame можно записать, к примеру, на DBFS

```
pandas_data = pd.DataFrame({
    'package_name' : ['Ansys', 'Nastran', 'Abaqus', 'LMS Virtual Lab', 'Comsol'],
    'solver_type' : ['direct', 'iterative', 'direct', 'iterative', 'iterative'],
    'language' : ['IronPython', 'Java', 'C++', 'Python', 'Erlang'],
    'performance' : np.abs(10*np.random.RandomState(42).randn(5))
})
data = spark.createDataFrame(pandas_data)

# сохранить объект на DBFS в формате csv
data.write.save('dbfs:/FileStore/tables/data.csv', format='csv')

# прочитать объект
spark.sql('''
    SELECT * FROM csv.`dbfs:/FileStore/tables/data.csv`
''').show()

# сохранить объект на DBFS в формате parquet
data.write.save('dbfs:/FileStore/tables/cae_packages.parquet', format='parquet')

# прочитать объект
spark.sql('''
    SELECT * FROM parquet.`dbfs:/FileStore/tables/cae_packages.parquet`
''').show()
```

Формат Parquet – это колончный (столбцово-ориентированный) формат хранения данных, который поддерживается системой Hadoop. Он сжимает и кодирует данные, и может работать с вложенными структурами – все это делает его очень эффективным.

К слову, удалить таблицы, находящиеся в оперативной памяти, можно так

```
from pyspark.sql import SQLContext

sqlcont = SQLContext(sc)

for tab in sqlcont.tableName():
    sqlcont.dropTempTable(tab)
```

## 21. Приемы работы с библиотекой Breeze

Математика в Spark, как правило, реализована с помощью Breeze. Библиотека Breeze нераспределенная!!! Предполагается, что Breeze работает над небольшими блоками данных.

Оптимизация в Breeze

```
import breeze.linalg._
import breeze.numerics._
import breeze.optimize.{DiffFunction, LBFGS}

val X = DenseMatrix.rand(2000, 3)
val y = X*DenseVector(0.5, -0.1, 0.2) // цель

val J = new DiffFunction[DenseVector[Double]] {
```

```

def calculate(w: DenseVector[Double]) = {
  val e = X*w - y
  val loss = sum(e ^ 2.0) / (2 * X.rows)
  val grad = (e.t * X) / (2.0 * X.rows)
  (loss, grad.t)
}

val optimizer = new LBFGS[DenseVector[Double]]()
println(optimizer.minimize(J, DenseVector(0.0, 0.0, 0.0)))
// DenseVector(0.4999999885533594, -0.10000001104504522, 0.200000002605021208) // приближение к
цели

```

Здесь используется метод L-BFGS. Это алгоритм оптимизации семейства квази-ньютоновских методов, который аппроксимирует алгоритм Бройдена-Флетчера-Гольфарба-Шанно с учетом ограниченного объема компьютерной памяти. L-BFGS – популярный алгоритм оценки параметров в машинном обучении.

## 22. Spark ML Pipelines

Различают

- *Transformer*: принимают на вход данные, возвращает преобразованные данные,
- *Estimator*: принимает на вход данные, возвращает Transformer.

Существуют еще *Model* – это трансформер, который был получен с помощью Estimator.

Эти объекты можно собирать в конвейеры, например: `Transformer1 → Estimator1 → Estimator2`.

Сам по себе конвейер является Estimator, т.е. ему можно подать на вход данные. Если в конвейере есть Transformer, то данные будут преобразованы и переданы дальше. Если в конвейере будет Estimator, то данные будут поданы на вход этому Estimator, который вернет Transformer, который в свою очередь будет применен к данным и вернет преобразованные данные.

Пример

```

import org.apache.spark.sql.SparkSession

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

val X = DenseMatrix.rand(10000, 3)
val y = X*DenseVector(0.5, -0.1, 0.2)
val data = DenseMatrix.horzcat(X, y.asDenseMatrix.t)

val df = spark.createDataFrame(
  data(*, ::).iterator // итератор по строкам
  .map(row => (row(0), row(1), row(2), row(3)))
  .toSeq
).toDF("x1", "x2", "x3", "y")
df.show(1)
// Вывод
+-----+-----+-----+-----+
|          x1|          x2|          x3|          y|
+-----+-----+-----+-----+
|0.9528102359167567|0.7292676335740298|0.5690442082761085|0.5172871962561971|

```

```
+-----+-----+-----+-----+
only showing top 1 row
```

### Подготовка конвейера

```
import org.apache.spark.ml.Pipeline
import org.apache.spark.ml.feature.VectorAssembler
import org.apache.spark.ml.regression.{
  LinearRegression, // модель ML
  LinearRegressionModel // обученная модель ML
}

val pipeline = new Pipeline().setStages(
  Array( // массив этапов
    new VectorAssembler() // трансформер
      .setInputCols(Array("x1", "x2", "x3"))
      .setOutputCol("features"), // (x1, x2, x3) -> features
    new LinearRegression().setLabelCol("y") // эстиматор
  )
)

val model = pipeline.fit(df)
val w = model.stages.last
  .asInstanceOf[LinearRegressionModel].coefficients
// w: org.apache.spark.ml.linalg.Vector =
//    [0.5000000000000163,-0.0999999999999584,0.2000000000000076]

val pred = model.transform(df)
```

Здесь модель линейной регрессии принимает на вход вектор с именем "features" и целевой вектор "y", на который мы указываем с помощью `setLabelCol("y")`.

ВАЖНО: здесь `model` это не линейная регрессия, а конвейер, в котором модель линейной регрессии лежит на последнем этапе.

Метод `transform` возвращает объект, у которого будет два новых поля (`features` и `prediction`). Поле `features` добавил `VectorAssembler`, когда собирал данные. Поле `prediction` очевидно добавила модель в качестве прогноза.

ВАЖНО: если сейчас посмотреть на схему данных `pred`, то она будет выглядеть примерно так

```
root
 |-- x1: double (nullable = false)
 |-- x2: double (nullable = false)
 |-- x3: double (nullable = false)
 |-- y: double (nullable = false)
 |-- features: vector (nullable = true) # <-- потеряли информацию о природе признаков
 |-- pred: double (nullable = false)
```

В глубоких конвейерах могут возникнуть сложности из-за потери информации о природе признаков, например, на этапе построения интерпретации.

В Spark есть свой собственный тип векторов и матриц (это не то же самое, что векторы и матрицы Breeze)

- Vector:
  - DenseVector,
  - SparseVector

- Matrix
  - DenseMatrix,
  - SparseMatrix (CSC)

Есть метод `compressed`, который в зависимости от структуры вектора/матрицы принимает решение о том, в каком виде имеет смысл хранить данные (в полносвязанном или в разреженном).

Для сложных преобразований можно превратить spark-вектор/матрицу в breeze-вектор/матрицу с помощью `asBreeze`.

Извлечение атрибутов

```
import org.apache.spark.ml.attribute.AttributeGroup

AttributeGroup.fromStructField(pred.schema("features"))
  .attributes.get.foreach(println)
// Вывод
{"type": "numeric", "idx": 0, "name": "x1"}
{"type": "numeric", "idx": 1, "name": "x2"}
{"type": "numeric", "idx": 2, "name": "x3"}
```

Для модульного тестирования используется ScalaTest [https://www.scalatest.org/user\\_guide](https://www.scalatest.org/user_guide).

## 23. Запросы к DataFrame с помощью методов и SQL

### 23.1. Простые примеры

Пример запроса к объекту DataFrame с использованием методов

Scala

```
> val df = spark.read.option("header", "true").csv("file_name.csv")
> df.show(3)
//+-----+-----+-----+-----+
//|state/region|  ages|year|population|
//+-----+-----+-----+-----+
//|          AL|under18|2012|  1117489|
//|          AL| total|2012|  4817528|
//|          AL|under18|2010|  1130966|
//+-----+-----+-----+-----+
//only showing top 3 rows
> df.select($"ages", $"year").filter(
  $"year" > 2010 && $"state/region" === "AL"
).orderBy($"year").show
```

Тот же самый запрос, но с использованием SQL

Scala

```
> df.createOrReplaceTempView("state_population")
spark.sql(
  "SELECT ages, year FROM state_population WHERE year > 2010 and 'state/region' = \"AL\" ORDER
  BY 2;"
).show
```

В pandas этот запрос выглядел бы так

Python

```
> df = pd.read_csv("file_name.csv", header=0)
```

```
> df[["ages", "year"]][
  (df["year"] > 2010) & (df["state/region"] == "AL")
].sort_values("year")
```

Или так

Python

```
df.query(
  "year > 2010 & 'state/region' == 'AL'"
)[["ages", "year"]].sort_values("year")
```

Еще пример с использованием эквивалента конструкции CASE WHEN

Scala

```
df.select(// список выборки
  $"ages",
  when($"ages" === "under18", 0).
  when($"ages" === "total", 1).
  otherwise(2).as("encodedCol"),
  $"year"
).filter(
  $"year" > 2010 && $"state/region" === "AL"
).show
//+-----+-----+-----+
//|      ages|encodedCol|year|
//+-----+-----+-----+
//|under18| 0|2012|
//|total| 1|2012|
```

Или, используя SQL

Scala

```
spark.sql(
  "SELECT
    ages,
    CASE WHEN (ages = \"under18\") THEN 0
         WHEN (ages = \"total\") THEN 1
         ELSE 2 END AS encodedCol,
    year
  FROM state_population
  WHERE year > 2010 and 'state/region' = \"AL\"
  ORDER BY 3;"
).show
```

## 23.2. Использование сложных агрегаций

Вычислить простые агрегаты можно так

```
df.agg(countDistinct("department")) // число уникальных значений в столбце
df.agg(min("salary")) // минимальное значение столбца
```

Если требуется в одном запросе по сгруппированным данным вычислить и, скажем, среднее, и максимальное значение, то можно воспользоваться следующей конструкцией

```
import spark.implicits._ // NB
import org.apache.spark.sql.functions._

df.groupBy("department").agg(
```

```

    avg("salary").as("avgSalary"),
    min("bonus").as("minBonus"),
    max("bonus").as("maxBonus")
  ).show

// подсчитать число элементов в каждой группе
df.groupBy("state").agg(
  count("salary").as("countSalary")
).show

```

## 24. Случайный лес в Spark

Пример решения задачи с использованием алгоритма случайного леса на платформе Spark

### Случайный лес

```

import org.apache.spark.mllib.tree.RandomForest
import org.apache.spark.mllib.tree.configuration.Strategy
import org.apache.spark.mllib.util.MLUtils

// Загрузка и парсинг данных
val data = MLUtils.loadLibSVMFile(sc, "data.txt")
// Разбиение множества данных на обучение и тест
val splits = data.randomSplit(Array(0.7, 0.3))
val (trainingData, testData) = (splits(0), splits(1))
// Обучение модели
val treeStrategy = Strategy.defaultStrategy("Classification")
val numTrees = 200
val featureSubsetStrategy = "auto"
val model = RandomForest.trainClassifier(
  trainingData,
  treeStrategy,
  numTrees,
  featureSubsetStrategy,
  seed = 12345
)
// Проверка на тестовом наборе
val testErr = testData.map{
  point => {
    val prediction = model.prediction(point.features)
    if (point.label == prediction) 1.0 else 0.0
  }
}.mean()
println(testErr)
println(model.toDebugString)

```

## 25. Экстремальный градиентный бустинг с XGboost4j

Подробности в <https://xgboost.readthedocs.io/en/latest/jvm/>.

Тонкая настройка XGboost4j для Spark: [https://xgboost.readthedocs.io/en/latest/jvm/xgboost4j\\_spark\\_tutorial.html](https://xgboost.readthedocs.io/en/latest/jvm/xgboost4j_spark_tutorial.html)

Установить XGBoost4j для Scala можно с помощью sbt <https://xgboost.readthedocs.io/en/latest/jvm/index.html#>

build.sbt

```
...
libraryDependencies += Seq(
  // здесь нужно заменить "latest_version_num" на стабильную версию, например, на "1.3.1"
  "ml.dmlc" %% "xgboost4j" % "latest_version_num",
  "ml.dmlc" %% "xgboost4j-spark" % "latest_version_num"
)
```

Пример решения задачи с помощью алгоритма экстремального градиентного бустинга

```
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.types.{DoubleType, StringType, StructField, StructType}
import org.apache.spark.ml.features.StringIndexer
import org.apache.spark.ml.feature.VectorAssembler
import ml.dmlc.xgboost4j.scala.spark.XGBoostClassifier

val spark = SparkSession.builder().getOrCreate()
val schema = new StructType(Array(
  StructField("sepal length", DoubleType, true),
  StructField("sepal width", DoubleType, true),
  StructField("petal length", DoubleType, true),
  StructField("petal width", DoubleType, true),
  StructField("class", StringType, true)
))
val rawInput = spark.read.schema(schema).csv("input_path")

val stringIndexer = new StringIndexer().
  setInputCol("class").
  setOutputCol("classIndex").fit(rawInput)
val labelTransformed = stringIndexer.transform(rawInput).drop("class")
val vectorAssembler = new VectorAssembler().
  setInputCols(Array(
    "sepal length",
    "sepal width",
    "petal length",
    "petal width"
  )).setOutputCol("feature")
val xgbInput = vectorAssembler.transform(labelTransformed).select("features", "classIndex") // n
// подготовленный набор (X, y)
// Тренировка
val xgbParam = Map(
  "eta" -> 0.1f,
  "max_depth" -> 2,
  "objective" -> "multi:softprob",
  "num_class" -> 3,
  "num_round" -> 100,
  "num_workers" -> 2
)
val xgbClassifier = new XGBoostClassifier(xgbParam).
  setFeaturesCol("features").
  setLabelCol("classIndex")
val xgbClassificationModel = xgbClassifier.fit(xgbInput)
```

Пример использования XGboost4j для решения задачи регрессии

```
import ml.dmlc.xgboost4j.scala.spark.{XGBoostRegressionModel, XGBoostRegressor} // стандартный
XGBoost
import org.apache.spark.ml.evaluation.{RegressionEvaluator}
import org.apache.spark.ml.tuning.ParamGridBuilder
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.types.{FloatType, IntegerType, StructField, StructType}
```



```

import ml.dmlc.xgboost4j.scala.spark.rapids.CrossValidator // XGBoost4j Rapids: можно использовать для повышения производительности с помощью GPU

val trainParquetPath = "/data/taxi/parquet/train"
val evalParquetPath = "/data/taxi/parquet/eval"

val labelColName = "fare_amount"
val schema =
  StructType(Array(
    StructField("vendor_id", FloatType),
    StructField("passenger_count", FloatType),
    ...
    StructField(labelColName, FloatType),
    StructField("is_weekend", FloatType)
  ))

val spark = SparkSession.builder().appName("taxi-gpu-cv").getOrCreate()
val trainDs = spark.read.parquet(trainParquetPath)

val featureNames = schema.filter(_.name != labelColName).map(_.name)

val regressionParam = Map(
  "learning_rate" -> 0.5,
  "max_depth" -> 8,
  "subsample" -> 0.8,
  "gamma" -> 1,
  "num_round" -> 100,
  "tree_method" -> "gpu_hist"
)

val regressor = new XGBoostRegressor(regressorParam).
  setLabelCol(labelColName). // целевая переменная
  setFeaturesCols(featureNames) // признаки
val paramGrid = new ParamGridBuilder().
  addGrid(regressor.maxDepth, Array(3, 10)).
  addGrid(regressor.eta, Array(0.2, 0.6)).
  build()
val evaluator = new RegressionEvaluator().setLabelCol(labelColName)
val cv = new CrossValidator(). // используется RAPIDS для параллелизации
  setEstimator(regressor).
  setEvaluator(evaluator).
  setEstimatorParamMaps(paramGrid).
  setNumFolds(3)

val model = cv.fit(trainDs).bestModel.asInstanceOf[XGBoostRegressionModel]
val transformDs = spark.read.parquet(evalParquetPath)
val df = model.transform(transformDs).cache() // делаем предсказания
df.select("fare_amount", "prediction").show(5)

//val evaluator = new RegressionEvaluator().setLabelCol(labelColName)
val rmse = evaluator.evaluate(df)
spark.close()

```

## 26. Простой пример использования Kafka и программная реализация на Scala

Для начала следует добавить зависимость Kafka в проект

#### build.sbt

```
name := "hellokafka"

version := "0.1"

scalaVersion := "2.12.12"

libraryDependencies += "org.apache.kafka" %% "kafka" % "2.5.0"
```

Теперь в IntelliJ IDEA можно создать простую связку «продюсер-консьюмер». Как обычно начинаем с создания проекта: **File** » **New** » **Project**. Задаем имя проекта (например, «hellokafka»), выбираем нужную версию Scala и запускаем процедуру создания проекта (занимает некоторое время).

В директории **hellokafka** » **src** » **main** » **scala** создаем поддиректорию **hellokafka** (в терминах IDEA *пакет*<sup>4</sup>). В этом пакете **hellokafka** создаем два scala-сценария (в терминах IDEA Package Object): **MainProducer.scala** и **MainConsumer.scala**.

#### MainProducer.scala

```
package hellokafka

import java.util.Properties
import org.apache.kafka.clients.producer.{KafkaProducer, ProducerRecord}

object MainProducer extends App {
  val props = new Properties()

  props.put("bootstrap.servers", "localhost:9082,localhost:9083")
  // нужны сериализаторы
  props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer")
  props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer")

  val topic = "welcome" // название топика

  val producer = new KafkaProducer[String, String](props)

  for (i <- 1 to 1000) {
    Thread.sleep(5000)
    println(s"sent { message : $i }")
    producer.send(new ProducerRecord[String, String](topic, s"key $i", s"message $i"))
  }
}
```

#### MainConsumer.scala

```
package hellokafka

import java.util.Properties
import org.apache.kafka.clients.consumer.KafkaConsumer

import scala.collection.JavaConverters._

object MainConsumer extends App {
  val props = new Properties()
```

<sup>4</sup>Package

```
// bootstrap-серверы -- это серверы, через которые можно подключиться к кластеру
props.put("bootstrap.servers", "localhost:9082,localhost:9083")
// нужны десериализаторы
props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer")
props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer")
props.put("group.id", "rescuer") // консьюмеры потребляют через группы

val topic = "welcome"

val consumer = new KafkaConsumer[String, String](props)

consumer.subscribe(java.util.Collections.singleton(topic))

while (true) {
    val records = consumer.poll(java.time.Duration.ofSeconds(12))
    for (record <- records.asScala) {
        println(s"${record.timestamp()} : ${record.value()}")
    }
}
}
```

Теперь остается запустить связку:

1. Запустить сервер ZooKeeper (см. 33.2),
  2. Установить соединение с сервером ZooKeeper,
  3. Запустить сервер Kafka (см. 34),
  4. Запустить сценарий `MainProducer.scala` (кликнув правой кнопкой мыши по файлу и выбрав «Run 'MainProducer'»),
  5. Запустить сценарий `MainConsumer.scala` (кликнув правой кнопкой мыши по файлу и выбрав «Run 'MainConsumer'»),
  6. В завершении работы следует остановить продюсер, затем консьюмер, потом сессию Kafka (`Ctrl+C`) и, наконец, сессию ZooKeeper (`Ctrl+C`),
  7. Дополнительно следует вызывать сценарий `zkServer.sh stop`.
- В итоге проект должен иметь структуру, приведенную на рис. 1.

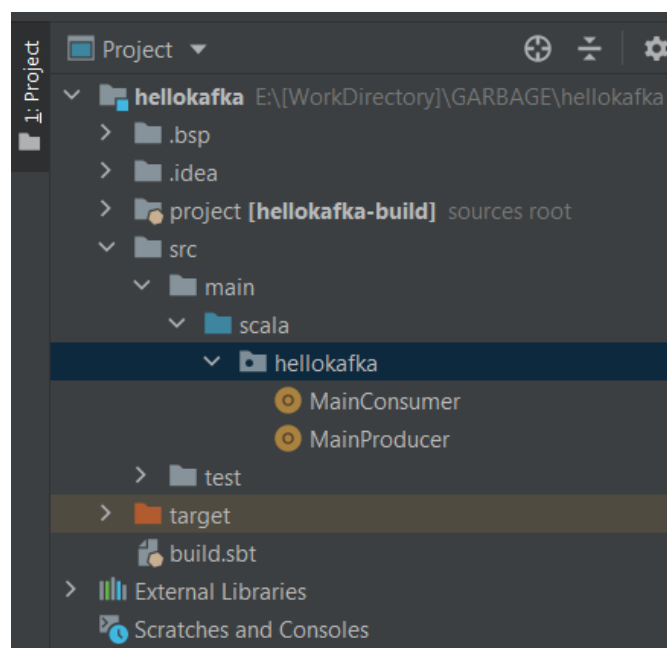


Рис. 1. Структура проекта hellokafka

## 27. Spark Streaming и Kafka

Затем (в самом простом варианте) следует создать продюсер (т.е. сервис порождающий данные) и консьюмер (т.е. сервис потребляющий данные).

Простой пример подписки на один топик

```
val df = spark
  .readStream
  .format("kafka")
  .option("kafka.bootstrap.servers", "localhost:9092")
  .option("subscribe", "quickstart-events") // <-
  .load()

df.selectExpr("CAST(key AS STRING)", "CAST(value AS STRING)")
  .writeStream
  .outputMode("append")
  .format("console")
  .start().awaitTermination()
```

## 28. Распределенное глубокое обучение с Elephas

Elephas <https://github.com/maxpumperla/elephas> – это распределенная платформа глубокого обучения, построенная на связке «Keras + Spark».

Пример

```
from pyspark import SparkContext, SparkConf
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Activation
from keras.optimizers import SGD
from elephas.utils.rdd_utils import to_simple_rdd
from elephas.spark_model import SparkModel

conf = SparkConf().setAppName("Elephas_App").setMaster("local[8]")
sc = SparkContext(conf = conf)

model = Sequential()
model.add(Dense(128, input_dim=784))
model.add(Activation("relu"))
model.add(Dropout(0.2))
model.add(Dense(128))
model.add(Activation("relu"))
model.add(Dropout(0.2))
model.add(Dense(10))
model.add(Activation("softmax"))
model.compile(loss="category_crossentropy", optimizer=SGD())

# RDD-подход
rdd = to_simple_rdd(sc, x_train, y_train)
spark_model = SparkModel(model, frequency="epoch")
spark_model.fit(rdd, epochs=20, batch_size=32, verbose=0, validation_split=0.1)

# DataFrame-подход. Лучше так!!!
df = to_data_frame(sc, x_train, y_train, categorical=True)
test_df = to_data_frame(sc, x_test, y_test, categorical=True)
estimator = ElephasEstimator(model, epochs=epochs, batch_size=batch_size, frequency="batch",
                              categorical=True, nb_classes=nb_classes)
fitted_model = estimator.fit(df)
```

## 29. Spark и Microsoft Machine Learning

Очень полезная обертка для Spark <https://github.com/Azure/mmlspark>. Jupyter Notebook с примерами: <https://github.com/Azure/mmlspark/tree/master/notebooks/samples>

Запустить сессию на Python

```
import pyspark
spark = pyspark.sql.SparkSession.builder.appName("MyApp") \
    .config("spark.jars.packages", "com.microsoft.ml.spark:mmlspark_2.11:1.0.0-rc3") \
    .config("spark.jars.repositories", "https://mmlspark.azureedge.net/maven") \
    .getOrCreate()
import mmlspark
```

## 30. Ошибка java.lang.IllegalAccessException

Ошибка `java.lang.IllegalAccessException` возникает (по моим наблюдениям), когда в преобразовании приходится обращаться к объекту, который (судя по всему) доступен только на драйвере

```
val sortedCol: org.apache.spark.rdd.RDD[(Double, Long)] = ...

val ranksOnly = sortedCol.filter{
  val ranks = List[Long](...) // во избежание ошибки должен располагаться внутри преобразования
  elem => ranks.contains(elem._2) // здесь нельзя использовать case!!!
}
```

## 31. Структурный и непрерывный стриминг

Чем больше интервал пакетирования (размер батча), тем лучше (от нескольких миллисекунд до нескольких минут или даже часов).

Можно выделить три основных режима:

- микробатчевый режим (обычный DStream),
- структурированный микробатчевый,
- структурированный непрерывный.

Различают следующие модели:

- Spark Streaming (DStream, микробатчевый режим) (не развивается!): это абстракция высокого уровня, называемая *дискретным потоком* (DStream); представлен как последовательность RDD,
- Structured Streaming: передача не материализует всю таблицу сразу. Можно работать как с реляционной таблицей, которая обновляется в реальном времени. Structured Streaming считывает последние доступные данные из источника потоковых данных, обрабатывает их постепенно для обновления таблицы результатов, а затем отбрасывает первые данные. В этой модели Spark отвечает за обновление таблицы результатов при появлении новых данных. Не нужно думать об абстрактной ширине окна и т.д. как в случае DStream.

Пример *потокowego запроса* (Streaming Query)

```
events.where("state = CA").groupBy(window("time", "30 seconds")).avg("latency")
```

Области применения фреймворков:

- Большой поток, простой агрегат: Spark,
- Быстрая реакция, сложный агрегат (поиск аномалий во входящих данных с датчиков): Flink,
- Минимум зависимостей, деплой в Kubernetes (микро-сервис, считающий статистику в реальном времени): Kafka

## 32. Оптимизация гиперпараметров и AutoML

Есть интересное расширение для Spark ML под именем PravdaML. Это расширение добавляет гибкости в вопросах организации потока данных, повышает коэффициент утилизации ресурсов и улучшения масштабирования ML.

## 33. Apache Zookeeper

### 33.1. Общие сведения

Apache Zookeeper – это сервис распределенной координации – централизованная служба для поддержки информации о конфигурации, обеспечения распределенной синхронизации и предоставления групповых служб. Все эти виды услуг используются в той или иной форме распределенными приложениями.

В конце сеанса следует:

- Остановить продюсер, консаммер с помощью **Ctrl-C**,
- Остановить Kafka с помощью **Ctrl-C**,
- Остановить ZooKeeper с помощью **Ctrl-C**.

Основные свойства Zookeeper:

- пространство ключей образует дерево (иерархию, подобную файловой системе),
- значения могут содержаться в любом узле иерархии, а не только в листьях (как если бы файлы одновременно были бы и каталогами), узел иерархии называется **znode**,
- между клиентом и сервером двунаправленная связь, следовательно, клиент может подписываться как изменение конкретного значения или части иерархии,
- возможно создать временную пару ключ/значение, которая существует, пока клиент, ее создавший, подключен к серверу,
- все данные должны помещаться в память,
- устойчивость к смерти некритического количества узлов кластера.

### 33.2. Установка и запуск Zookeeper

Чтобы установить Zookeeper на MacOS следует с официального сайта проекта скачать tar-архив <https://www.apache.org/dyn/closer.lua/zookeeper/zookeeper-3.6.2/apache-zookeeper-3.6.2-bin.tar.gz> и распаковать его, например, в поддиректорию **zookeeper** домашней директории

```
tar -xvzf apache-zookeeper-3.6.2-bin &&\n
mv apache-zookeeper-3.6.2 zookeeper-3.6.2
```

Затем нужно в конфигурационном файле командной оболочки **.bashrc**, **.zshrc** создать переменную окружения **ZOOKEEPER\_HOME**

~/zshrc

```
export ZOOKEEPER_HOME="/Users/leor.finkelberg/zookeeper/zookeeper-3.6.2/bin"
export PATH="$ZOOKEEPER_HOME:${PATH}"
```

Кроме того необходимо переименовать файл `zoo_sample.cfg` в `zoo.cfg`, а затем заменить значение по умолчанию параметра `dataDir` на следующее <https://zookeeper.apache.org/doc/current/zookeeperStarted.html>

~/zookeeper/zookeeper-3.6.2/conf/zoo.cfg

```
# каталог data должен существовать, иначе Zookeeper не сможет запустить сервер
dataDir=~/.zookeeper/zookeeper-3.6.2/data
```

Остальные два параметра минимальной конфигурации – `tickTime` и `clientPort` – оставим без изменений.

Теперь можно запустить ZooKeeper

```
zkServer.sh start
```

Описанные выше шаги запускают ZooKeeper в автономном режиме. В этом случае не поддерживается репликация и если процесс упадет, то служба выйдет из строя. Такой схемы достаточно для большинства ситуаций, но все же, если требуется запустить ZooKeeper с поддержкой репликации, то следует ознакомиться с [https://zookeeper.apache.org/doc/current/zookeeperStarted.html#sc\\_RunningReplicatedZooKeeper](https://zookeeper.apache.org/doc/current/zookeeperStarted.html#sc_RunningReplicatedZooKeeper).

Далее устанавливаем соединение с ZooKeeper

```
zkCli.sh -server 127.0.0.1:2181
```

Теперь нужно подготовить запуск Apache Kafka. Предварительно бинарные файлы можно скачать здесь <https://kafka.apache.org/downloads>.

Перед запуском Kafka следует указать куда будут писаться логи. Сделать это можно, изменив значение параметра `log.dirs` в файле `server.properties`

~/kafka/kafka\_2.13-2.7.0/config/server.properties

```
log.dirs=~/.kafka/kafka_2.13-2.7.0/kafka-logs
```

и здесь же правим файл `zookeeper.properties`

~/kafka/kafka\_2.13-2.7.0/config/zookeeper.properties

```
dataDir=~/.kafka/kafka_2.13-2.7.0/zookeeper-data
```

А вот теперь можно запускать kafka-сессию

```
./kafka-server-start.sh ~/kafka/kafka_2.13-2.7.0/config/server.properties
```

Для создания топика используем следующий сценарий командной оболочки

```
./kafka-topics.sh --create --topic quickstart-events --bootstrap-server localhost:9092
# Created topic quickstart-events.
```

Посмотреть описание топика можно следующим образом

```
./kafka-topics.sh --describe --topic quickstart-events --bootstrap-server localhost:9092
```

Клиент Kafka общается с брокерами сообщений через сеть для записи (или чтения) событий. Получив сообщение брокеры будут хранить его так долго, как это нужно. Запустим клиент продюсера, чтобы записать в топик несколько событий

```
# запись некоторых событий в топик
./kafka-console-producer.sh --topic quickstart-events --bootstrap-server localhost:9092
>This is my first event
>This is my second events
>^C% # Ctrl-C
```

Теперь можно открыть еще один терминал и прочитать переданные в топик события

```
# чтение событий из топика
./kafka-console-consumer.sh --topic quickstart-events --from-beginning --bootstrap-server
localhost:9092
# Ctrl-C
```

Остановить ZooKeeper можно так

```
zkServer.sh stop
```

## 34. Apache Kafka

### 34.1. Установка и запуск Kafka

ВАЖНО: перед запуском Kafka следует запустить ZooKeeper (см. 33.2).

Apache Kafka – брокер сообщений, работающий поверх сервиса Apache Zookeeper.

Простая схема: создается топик (тема), в которую будут отправляться сообщения от продюсеров, и на которую смогут подписаться консьюмеры, чтобы их получать.

### 34.2. Общие сведения

Очень полезное видео <https://www.youtube.com/watch?v=m5CDfrQLzrs>.

Очень полезный ресурс, на котором обсуждаются тонкие вопросы по работе с Kafka, которых нет даже в документации: <https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Improvement+Proposals>.

Kafka – брокер сообщений. Основная задача – это доставка сообщений от продюсеров (сервисы, порождающие данные) до консьюмеров (сервисы, потребляющие данные).

Кластер Kafka почти всегда состоит из *нод*, которые называют *брокерами*, а координация между нодами осуществляется с помощью демона Zookeeper (см. рис. 2).

Данные в Kafka разбиты по *топикам* и каждый топик состоит из определенного числа *партиций*. Топик находится на брокере. Партиция – основная единица хранения данных в Kafka. Именно в партициях хранятся данные. Топик – это логический контейнер вокруг партиции. Каждая партиция – неизменяемая последовательность записей.

Продюсер может записывать данные только в конец очереди и никак не может повлиять на уже записанные данные (не может ни удалить, ни изменить). Каждому сообщению присваивается уникальный в пределах партиции порядковый номер (называет offset).

Как читать:

- о указываем потребителю топик, указываем номер партиции и указываем offset, с которого надо начать читать. Когда партиция исчерпается, потребитель будет ждать новых сообщений (в этом случае нужно вне Kafka запоминать offset),
- о можно создать группу (groupId) и тогда Kafka будет сама следить за offset.



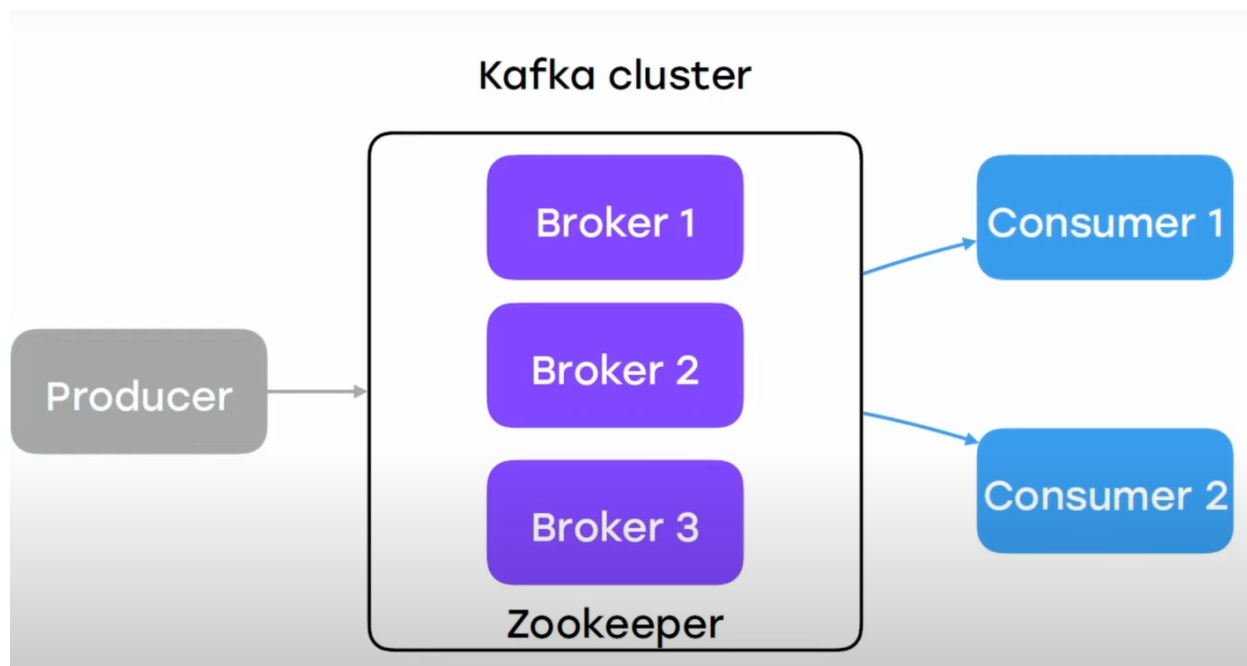


Рис. 2. Базовая схема кластера Kafka

Продюсер всегда пишет в *лидер* (это брокер, т.е. узел кластера Kafka). Консьюмеры читают тоже с лидера (реплики нужны для подстраховки, т.е. если с лидером что-то происходит, то они займут его место).

Для одних партиций какой-то брокер является лидером, а для других он будет репликой!!!

ВАЖНО: даже после того как сообщение будет записано продюсером в лидер, оно не будет сразу же доступно для чтения консьюмерам. Сообщение будет доступно для чтения только после того как это сообщение «подтянут» все реплики!!! Обойти это в Kafka нельзя. То есть нельзя сказать, например, что «пусть сообщение считается доступным сразу после того как будет записано в лидер».

Данные пишутся со скоростью самого медленного брокера.

Для увеличения скорости записи, можно уменьшить значение параметра `replica.fetch.wait.max.ms` (это временной интервал, через который брокер-реплика будет опрашивать брокер-лидера). По умолчанию стоит 500 мс. Можно уменьшить до 200 мс.

Важный конфигурационный параметр – `acks` (тип подтверждения, который будут получать от лидера при записи данных):

- `acks=0`: не ждем никакого подтверждения; очень вероятна потеря сообщений (мы об этом никогда не узнаем); не подходит для важных данных,
- `acks=1`: ждем подтверждение только от лидера, что он записал у себя сообщение; не подходит для важных данных,
- `acks=-1`: ждем подтверждение от всех реплик (подходит).

Параметр `replica.max.lag.ms=10000` (10 сек!): если реплика выпала, лидер будет ожидать ее 10 сек (это очень долго!!!). Можно уменьшить до 1 сек.

Параметр `min.insync.replicas` (очень важный параметр!!!) определяет минимальное число реплик в статусе `insync` (включая лидера), которое необходимо для успешной записи. По умолчанию стоит 1, но практичнее выставить 2. Этот параметр работает только для `acks=-1`.

Чтобы принудительно назначить брокер-реплику со статусом «out of sync» лидером, нужно в настройках топика задать `unclean.leader.election.enable=true`. Этот параметр бывает нужен,

когда требуется что-то починить. В Kafka  $\geq 2.0.0$  можно изменять динамически. Без понимания работы этого параметра лучше оставить значение по умолчанию, т.е. `false`.

Для того чтобы обеспечить и доступность и надежность в кластере из 3 брокеров следует использовать фактор репликации  $Rf=3$ , а  $minISR=2$ . В этом случае может возникнуть сложности с производительностью (в случае, если один из брокеров выпадет, нагрузка на оставшиеся может быть значительной со всеми вытекающими последствиями).

Иногда рекомендуют строить кластер из 5 брокеров (см. рис. 3): можно использовать  $Rf=3$ ,  $minISR=2$ .

## Кластер из 5 брокеров

Параметры	Доступность	Надежность
$Rf = 2, minISR = 1$	Да	Нет
$Rf = 2, minISR = 2$	Нет	Да
$Rf = 3, minISR = 2$	Да	Да
$Rf = 4, minISR = 2$	Да	Да
$Rf = 4, minISR = 3$	Да	Да
$Rf = 5, minISR = 3$	Да	Да

Рис. 3. Доступность и надежность кластера из 5 брокеров

В 3 ноды Zookeeper в отдельном кластере.

Выводы:

- Kafka не автопилотируемый проект; за ней нужно постоянно следить,
- Надежная запись:  $acks=-1$  и  $minISR > 2$ ,
- Мониторинг + учения,
- Регулярные обновления,
- KIP и Jira очень полезны.

## 35. Apache HBase

HBase – распределенная нереляционная (столбцово-ориентирования) база данных формата «ключ-значение».

### 35.1. Установка и запуск


Подробности, связанные с установкой различных режимах (автономном, распределенном и т.д.) можно узнать на странице <https://hbase.apache.org/book.html>.

Скачать tar-архив можно здесь <https://www.apache.org/dyn/closer.lua/hbase/2.4.0/hbase-2.4.0-bin.tar.gz>

```
curl -O https://apache-mirror.rbc.ru/pub/apache/hbase/2.4.0/hbase-2.4.0-bin.tar.gz
```

Теперь следует распаковать архив

```
tar -xvzf hbase-2.4.0...
```

перейти в директорию  hbase-2.4.0 и задать путь до java в файле hbase-env.sh, раскомментировав нужную строку

conf/hbase-env.sh

```
export JAVA_HOME=/usr/local/Cellar/openjdk/15.0.1
```

В конфигурационном файле командной оболочки удобно задать переменные окружения для Java и HBase

~/zshrc

```
# for HBase
export JAVA_HOME="/usr/local/Cellar/openjdk/15.0.1"
export PATH="${PATH}:/Users/leor.finkelberg/hbase/hbase-2.4.0/bin"
```

Директорию размещения java на MacOS X следует искать с помощью менеджера пакетов brew

```
brew list java # /usr/local/Cellar/openjdk/15.0.1/bin/java
```

ВАЖНО: обновить java, можно скачав соответствующую версию с ресурса <https://www.oracle.com/java/technologies/javase-jdk15-downloads.html>.

Запустить HBase можно с помощью сценария командной оболочки из  bin/

```
start-hbase.sh
```

Подключиться к запущенному экземпляру можно так

```
hbase shell
```

Для того чтобы убедиться, что процесс HMaster запущен можно воспользоваться утилитой jps.

Бывает удобно следить за работой приложения с помощью Web-интерфейса, доступного на <http://localhost:16010>.

Закончить сессию можно с помощью команды quit. Затем нужно остановить HBase

```
stop-hbase.sh
```

## 36. Пакетная и потоковая обработка данных

Пакетная обработка – обработка всего за раз без взаимодействия с конечным пользователем. Задача выполняется однократно или по расписанию, триггеру и пр.

Инструменты *пакетной* обработки:

- Spark – стандарт в этой области,
- Flink – псевдо-batch,
- Hive – когда знаешь только SQL.

Инструменты *потоковой* обработки:

- Spark Streaming – микробатчи,
- Flink – реальный стриминг,
- Kafka Streams – Карра-архитектура.

Apache Kafka – это быстрая, масштабируемая, надежная и отказоустойчивая система обмена сообщениями по механизму публикация-подписка. Еще можно сказать, что Kafka это распределенная потоковая платформа.

Если упрощенно, то Kafka предназначена для организации обмена сообщениями и результатами работы между микросервисами приложения.

Kafka работает с другими распределенными фреймверками как Spark, Samza, Flink для анализа и визуализации потоковых данных в реальном времени. Kafka хорошо интегрируется с ML фреймверками для решения ML/AI задач на потоках.

Основные определения:

- Producer – сервис, отправляющий сообщение,
- Consumer – сервис, получающий данные,
- Broker – один узел Kafka,
- Topic – логическая очередь,
- Partition – физическая часть очереди.

Обычно взаимодействие Kafka и Spark Streaming устроено следующим образом:

- исходные данные записываются в топики Apache Kafka,
- приложение Spark Streaming считывает нужные данные и обрабатывает их согласно бизнес-логике,
- полученные результаты приложение Spark Streaming отправляет в место назначения – новый топик Apache Kafka, озеро данных на базе Hadoop HDFS, аналитическую СУБД (HBase, Hive, Greenplum etc.) или BI-систему.

Топик состоит из партиций. Партиция упорядоченная и неизменяемая последовательность сообщений.

Семантика доставки:

- At most once (максимум один раз) – сообщения могут потеряться, но никогда не будут доставлены повторно (не будет дубликатов),
- At least once (минимум один раз) – сообщения никогда не теряются, но могут быть доставлены повторно (возможны дубликаты),
- Exactly once (строго один раз) – это то, чего на самом деле хотят люди; каждое сообщение доставляется только один раз.

Концепция и основные компоненты потоковой обработки

- Structured Streaming – передача не материализует всю таблицу сразу,
- Spark Streaming (Dstream) – предоставляет абстракцию высокого уровня, называемую дискретным потоком или DStream, которая представляет непрерывный поток данных. DStream – последовательность RDD.

Триггеры в Spark Streaming:

- Unspecified (по умолчанию) – если параметр триггера не указан явно, то по умолчанию запрос будет выполняться в режиме micro-batch, в котором микропакеты будут сгенерированы, как только предыдущий микропакет завершит обработку,
- Fixed interval micro-batches – запрос будет выполняться в режиме микропакетов, в котором микропакеты будут запускаться через указанные пользователем интервалы,
- One-time micro-batch – запрос будет выполнять только один микропакет для обработки всех доступных данных, а затем остановится самостоятельно,

- Continuous with fixed checkpoint interval (экспериментально) – запрос будет выполняться в новом режиме непрерывной обработки с малой задержкой.

Интеграция Kafka в Spark Streaming:

- Write Ahead Logs (WAL) для Kafka – это гарантирует, что никакие данные, полученные из любых надежных источников данных (т.е. транзакционных источников, таких как Flume, Kafka и Kinesis), не будут потеряны из-за сбоев. Даже для ненадежных (т.е. нетранзакционных) источников, таких как простые старые сокеты, это сводит к минимуму потерю данных.
- Direct API для Kafka – это позволяет обрабатывать каждую запись Kafka ровно один раз, несмотря на сбои, без использования журналов предварительной записи. Это делает конвейеры Spark Streaming + Kafka более эффективными, обеспечивая гарантию отказоустойчивости.

## 37. Приемы работы со Spark в Apache Zeppelin

Apache Zeppelin <http://zeppelin.apache.org/download.html> – это многофункциональная интерактивная оболочка, которая позволяет выполнять запросы к различным источникам данных, обрабатывать и визуализировать результаты, а самое главное «из коробки» поддерживает Spark. Близкий аналог Jupyter Notebook, но Zeppelin больше ориентирован на работу с базами данных. Он использует концепцию «интерпретаторов» – плагинов, которые обеспечивают бекенд для какого-либо языка и/или БД.

Проще всего запустить Zeppelin с помощью Docker

```
docker run -p 8080:8080 --rm --name zeppelin apache/zeppelin:0.9.0
# или так
docker run -p 8080:8080 --rm \
-v $(pwd)/logs:/logs \
-v $(pwd)/notebook:/notebook \
-e ZEPPELIN_LOG_DIR='/logs' \
-e ZEPPELIN_NOTEBOOK_DIR='/notebook' \
--name zeppelin apache/zeppelin:0.9.0
```

Страница Zeppelin будет доступна в браузере localhost:8080.

Подробное руководство по работе с Apache Zeppelin можно найти по адресу <https://docs.arenadata.io/aaw/Zeppelin/index.html>.

## Список литературы

1. Карау Х., Конвински Э., Венделл П., Захария М. Изучаем Spark: молниеносный анализ данных. – М.: ДМК Пресс, 2015. – 304 с.