Дата последней сборки документа:
15 декабря 2020 г. в 13:40

Базовый курс Spark в реализациях на Scala и Python

Содержание

1. 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 некотрые 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[1] (задержка не более 1 миллисекунды) или *фреймворков потоковой обработки больших данных* Apache Storm, Apache Flink и Apache Samza.

В отличие от классического MapReduce[2], 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.SparkSession 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:

```
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
      )
```

2. 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,

[1] 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

[2] Модель распределенных вычислений

```
.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),
- DataSet API (только для `Scala`! В `Python` это не имеет смысла): Scala-вский DataSet по сути представляет собой коллекцию экземпляров строк определенного типа; и поэтому, когда мы применяемнапример, 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 ItelliJ 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 дир
    ектория (!) from_spark_csv, в которой будет лежать csv-файл
// или
df.write.options(Map("header"->"true", "delimiter"->",")).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 репозиториев (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*.

3. Создание Spark DataFrame на основе списка

Создание объекта `Spark DataFrame` на основе списка

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

4. 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)]
```

5. 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)]
```

6. 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)]
```

7. Зарегистрировать пользовательскую функцию

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)]
```

8. 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
```

9. 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|
+-------+----+
```

10. Оконные функции в контексте SQL и Spark DataFrame

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

- ранжирующие,
- аналитические,
- агрегатные (любую агрегатную функцию[3] можно использовать в качестве оконной функции)

[3] Например, AVG, SUM, COUNT и пр.

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 стоки будут включены в фрейм для текущей строки, основываясь на их положении относительно текущей строки.

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`

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

```
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

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

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

11. 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 фалов текущей директории
dbutils.fs.ls('dbfs:/FileStore/tables')
# удалить файл из DBFS
dbuitls.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', 'Comsole'],
    '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 формает 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)
```

12. 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. 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`.

Список литературы

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