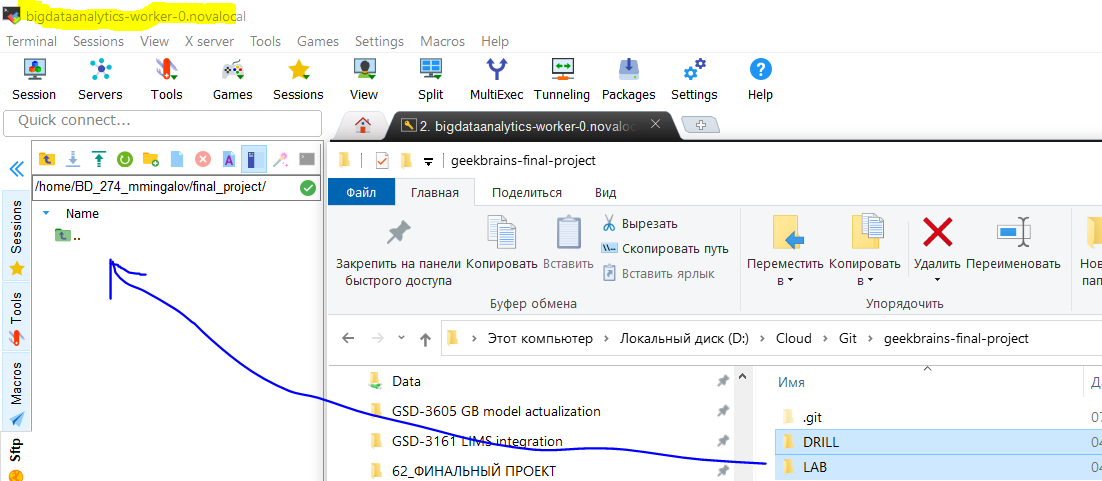
[**https://github.com/mmingalov/geekbrains-final-project/**](https://github.com/mmingalov/geekbrains-final-project/)

# **ПОДГОТОВКА ДАННЫХ**

**Открываем сессию mobaXterm**

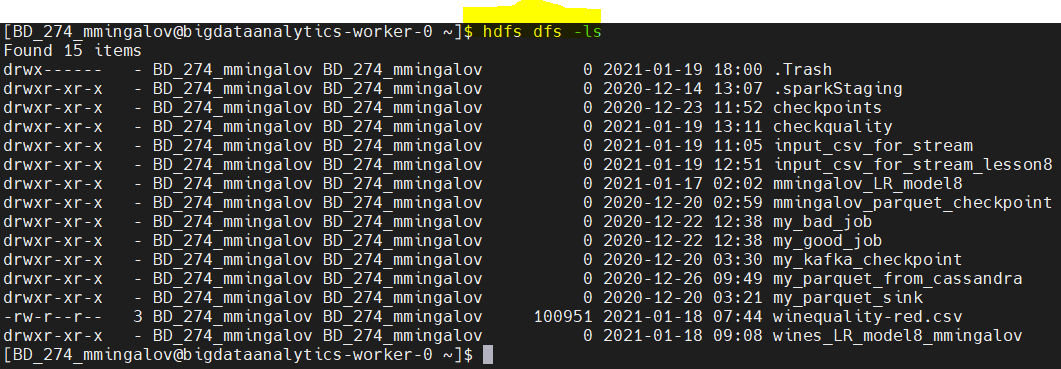
**Выполняем drag&drop копирование файлов в директорию проекта:**



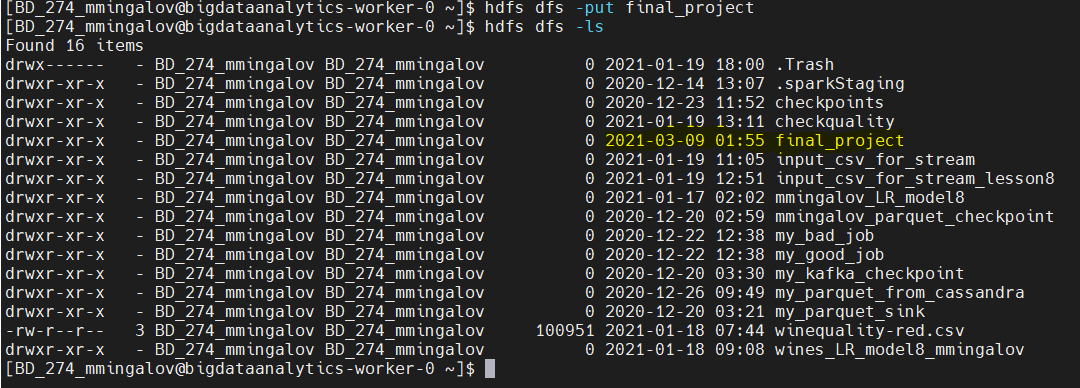
**Файлы скопированы в локальный каталог worker-0**

**Нам необходимо перенести их в HDFS**

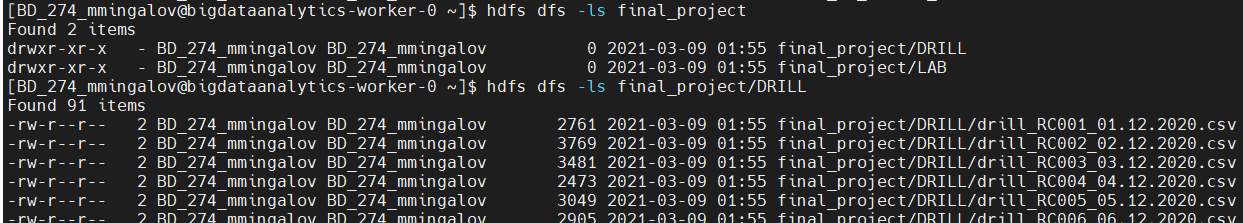
**Посмотрим, что за папки на данный момент есть в HDFS:**



**Создадим директорию под финальный проект: просто перенесем final\_project в HDFS командой** **put**



**Посмотрим внутрь..**



**Видим, что данные на месте и готовы к дальнейшей работе.**

# **ЧТЕНИЕ ДАННЫХ**

**Примечание: нижеприведенный код есть в** [**проекте**](https://github.com/mmingalov/geekbrains-final-project/) **в виде .py файла**

**Запускаем SPARK командой:**

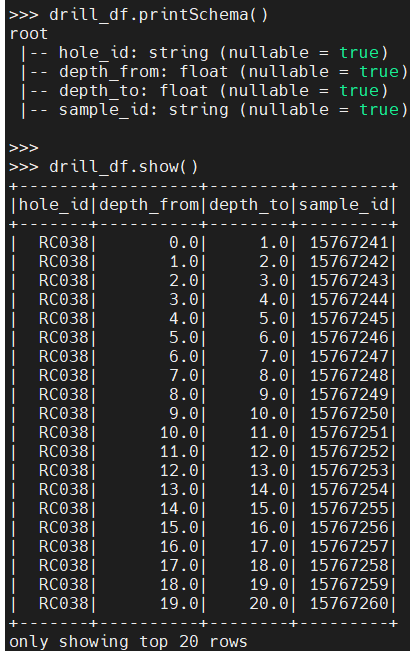
**export SPARK\_KAFKA\_VERSION=0.10**

**/spark2.4/bin/pyspark --packages org.apache.spark:spark-sql-kafka-0-10\_2.11:2.4.5 --driver-memory 512m --num-executors 1 --executor-memory 512m --master local[1]**

drill\_df = spark.read.format("csv")\  
 .option("header", "false")\  
 .schema("hole\_id STRING, depth\_from float, depth\_to float, sample\_id STRING")\  
 .load("final\_project/DRILL/\*.csv")  
lab\_df = spark.read.format("csv")\  
 .option("header", "false")\  
 .schema("sample\_id STRING, result FLOAT")\  
 .load("final\_project/LAB/\*.csv")

drill\_df.printSchema()

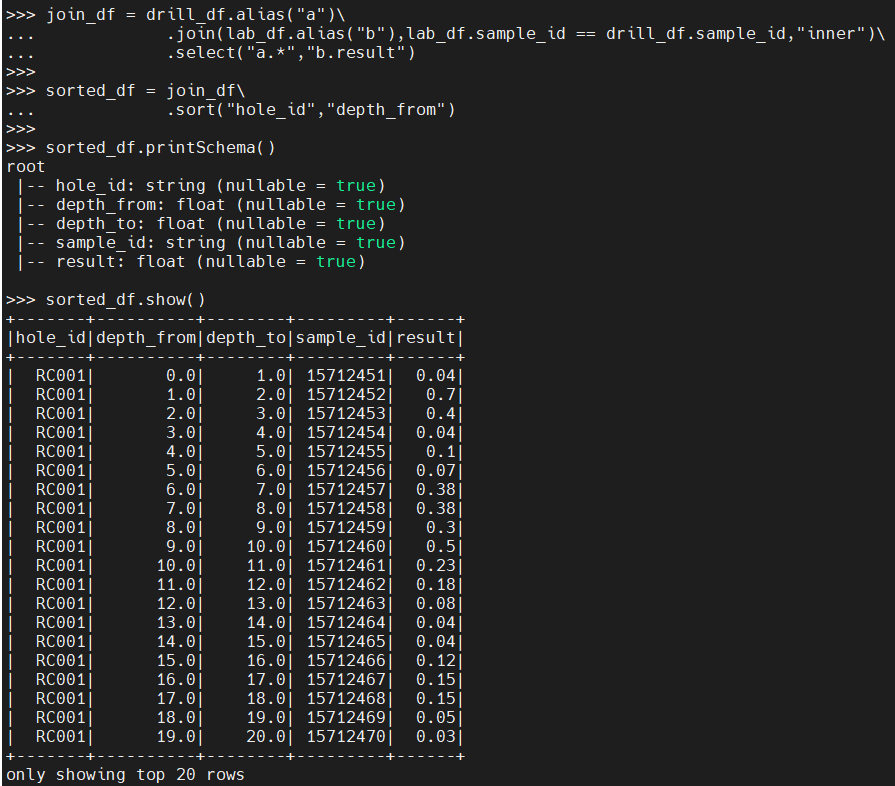
drill\_df.show()



# **СЛИЯНИЕ ДАННЫХ**

**Выполняем слияние двух spark\_df в общий датасет, сортируя по «номеру скважины» и глубине «ОТ»:**

#JOIN AND SORTING  
join\_df = drill\_df.alias("a")\  
 .join(lab\_df.alias("b"),lab\_df.sample\_id == drill\_df.sample\_id,"inner")\  
 .select("a.\*","b.result")  
  
sorted\_df = join\_df\  
 .sort("hole\_id","depth\_from")  
  
sorted\_df.printSchema()  
sorted\_df.show()



# **РАСЧЕТ КОМПОЗИТОВ (UDF) -- FAULT**

**Для расчета композитов нам понадобятся вспомогательные столбцы.**

**Будем рассчитывать их при помощи пользовательских функций**

**UDF**

from pyspark.sql.functions import udf

**при расчете композитов у нас есть глобальные параметры: содержание по борту и мощность рудного интервала**

Cb = 0.37  
Mr = 2

**Подготовим функции под шаги расчета композитов**

**Сперва вспомогательные:**

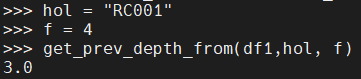
def get\_prev\_depth\_from(df,hol, f):  
 try:  
 result = df.filter((df["depth\_to"] == f) & (df["hole\_id"] == hol)).select(["depth\_from"]).collect()[0].\_\_getitem\_\_("depth\_from")  
 except:  
 result = -1  
 return result  
udf\_gpdf = udf(get\_prev\_depth\_from)  
  
def get\_next\_depth\_to(df, hol,t):  
 try:  
 result = df.filter((df["depth\_from"] == t) & (df["hole\_id"] == hol)).select(["depth\_to"]).collect()[0].\_\_getitem\_\_("depth\_to")  
 except:  
 result = -1  
 return result  
  
udf\_gndt = udf(get\_next\_depth\_to)

**потом основная логика:**

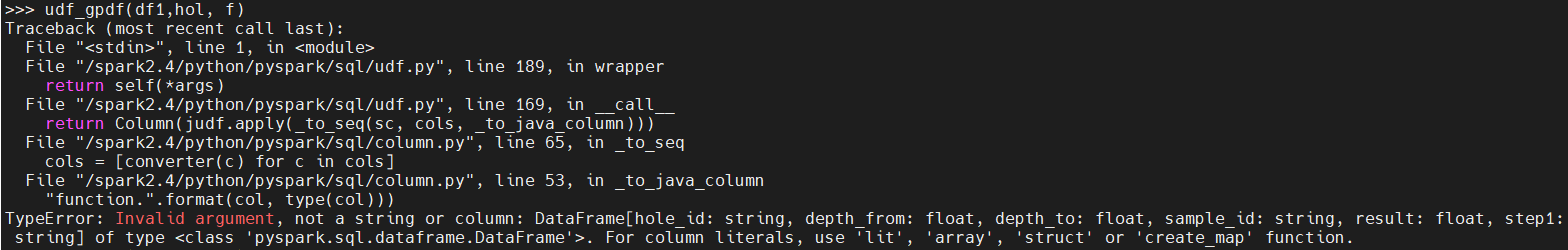
def calc\_step1(res):  
 if res < Cb:  
 result = 0 #'П'  
 else:  
 result = 1 #'Р?'  
 return result  
udf\_step1 = udf(calc\_step1)

#STEPS EXECUTING  
df1 = sorted\_df \  
 .withColumn("step1", udf\_step1(join\_df["result"]))

**Производя отладку несложных вспомогательных функций, уткнулся в проблему, что исходная функция срабатывает**



**А зарегистрированная по ней UDF функция выдает ошибку:**



**Комментарий преподавателя:** мне кажется вы неправильно udf пишете. udf функция должна принимать колонку какую то, и все операции делать на колонке или на нескольких колонках, но не на всех датафрейме. функцию collect тоже запрещаю использовать :) collect ( как и pandas) все результаты вычислений будет скидывать на драйвер, а нам наоборот нужно все распаралелить

**МОЙ ВЫВОД:** я это интуитивно конечно же понимаю. Но! дело в том, что у меня по условию алгоритма при расчете значения в текущей строке датафрейма учитываются значения раных столбцов в предыдущей и в последующей строке --- вот такое вот хитрое композитирование :) и я понимаю, что UDF на spark dataframe здесь не выход

Попробую реализацию через SQL

# **РАСЧЕТ КОМПОЗИТОВ (SQL) -- SUCCESS**

**Для расчета композитов нам понадобится пройти несколько шагов – создать 4 вычисляемых столбца по исходным данным. Финальный шаг и будет столбцом с признаком композита.**

**Исходная SQL-логика добавлена в раздел ПРИЛОЖЕНИЕ.**

**РЕАЛИЗАЦИЯ**

**При расчете композитов у нас есть глобальные параметры: содержание по борту и мощность рудного интервала. Задаем их:**

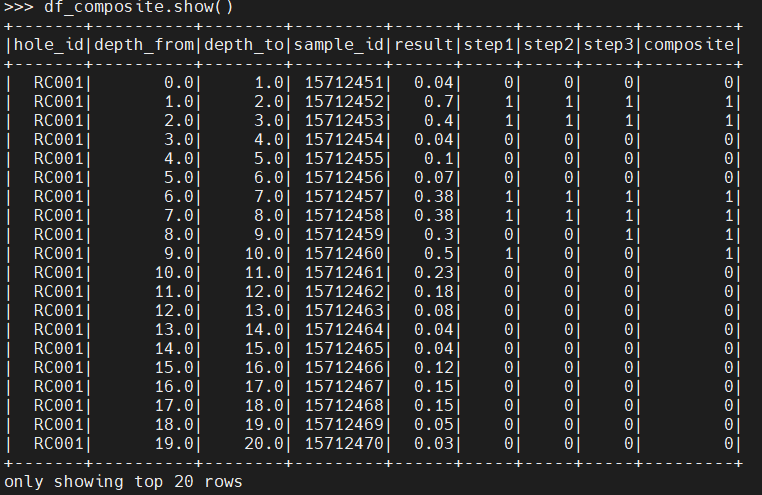
Cb = 0.37  
Mr = 2

**По итогам каждого шага у нас образуется временный объект View**

#STEPS EXECUTING  
sorted\_df.createOrReplaceTempView("sorted\_view")  
  
#step1  
sql1 = "select \*," \  
 " case " \  
 "when result >= "+ str(Cb) + " then 1 " \  
 "else 0 " \  
 "end as step1 " \  
 "from sorted\_view"  
# df1 = spark.sql(sql1)  
spark.sql(sql1).createOrReplaceTempView("df1")  
  
#step2  
sql2 = "select df1.\*" \  
 ", case when (df1.depth\_from <> (select min(t2.depth\_from) from df1 t2 where t2.hole\_id = df1.hole\_id)) " \  
 "and (df1.step1 = 1) and " \  
 "((((df1.depth\_to - (select avg(t2.depth\_from) from df1 t2 where t2.hole\_id = df1.hole\_id and t2.depth\_to = df1.depth\_from ))>=" + str(Mr) + ") " \  
 "and ((select avg(t2.result) from df1 t2 where t2.hole\_id = df1.hole\_id and t2.depth\_to = df1.depth\_from) >= " + str(Cb) + ")) " \  
 "or ((((select avg(t2.depth\_to) from df1 t2 where t2.hole\_id = df1.hole\_id and t2.depth\_from = df1.depth\_to ) - df1.depth\_from)>=" + str(Mr) + ") " \  
 "and ((select avg(t2.result) from df1 t2 where t2.hole\_id = df1.hole\_id and t2.depth\_from = df1.depth\_to) >= " + str(Cb) + "))) " \  
 "then 1 else 0 end as step2 from df1 order by df1.hole\_id, df1.depth\_from"  
#note to code above:  
# WHY I USE AVG() in sub-queries?  
# Because:  
# You have to make sure that your sub-query by definition (and not by data) only returns a single row. Otherwise Spark Analyzer complains while parsing the SQL statement.  
# So when catalyst can't make 100% sure just by looking at the SQL statement (without looking at your data) that the sub-query only returns a single row, this exception is thrown.  
# If you are sure that your subquery only gives a single row you can use one of the following aggregation standard functions, so Spark Analyzer is happy:  
# first # avg # max # min  
  
spark.sql(sql2).createOrReplaceTempView("df2")  
  
# step3  
sql3 = "select df2.\*, " \  
 "case when " \  
 "(df2.depth\_from <> (select min(t2.depth\_from) from df2 t2 where t2.hole\_id = df2.hole\_id)) " \  
 "and result < " + str(Cb) + " " \  
 "and (select avg(t2.result) from df2 t2 where t2.hole\_id = df2.hole\_id and t2.depth\_from = df2.depth\_to)>=" + str(Cb) + " " \  
 "and (select avg(t2.result) from df2 t2 where t2.hole\_id = df2.hole\_id and t2.depth\_to = df2.depth\_from)>=" + str(Cb) + " " \  
 "then 1 else step2 end step3 from df2 order by df2.hole\_id, df2.depth\_from"  
  
spark.sql(sql3).createOrReplaceTempView("df3")  
  
# step4  
sql4 = "select df3.\*, " \  
 "case when " \  
 "(df3.depth\_from <> (select min(t2.depth\_from) from df3 t2 where t2.hole\_id = df3.hole\_id)) " \  
 "and ((df3.step1 = 1) or (df3.step3 = 1)) " \  
 "and ( ((select avg(t2.step3) from df3 t2 where t2.hole\_id = df3.hole\_id and t2.depth\_to = df3.depth\_from)<>0) " \  
 "or ((select avg(t2.step3) from df3 t2 where t2.hole\_id = df3.hole\_id and t2.depth\_from = df3.depth\_to)<>0) " \  
 "or ( ((select avg(t2.step1) from df3 t2 where t2.hole\_id = df3.hole\_id and t2.depth\_to = df3.depth\_from)<>0) " \  
 "and ((select avg(t2.step1) from df3 t2 where t2.hole\_id = df3.hole\_id and t2.depth\_from = df3.depth\_to)<>0) " \  
 "and ( (select avg(t2.depth\_to) from df3 t2 where t2.hole\_id = df3.hole\_id and t2.depth\_from = df3.depth\_to)" \  
 "-(select avg(t2.depth\_from) from df3 t2 where t2.hole\_id = df3.hole\_id and t2.depth\_to = df3.depth\_from) ) >= " + str(Mr) + ")) " \  
 "then 1 else 0 end composite from df3 order by df3.hole\_id, df3.depth\_from"  
  
spark.sql(sql4).createOrReplaceTempView("df4")

**Все шаги выполнены. Запишем результат в финальный df и прочитаем его содержимое:**

df\_composite = spark.sql("select \* from df4 order by hole\_id, depth\_from")  
df\_composite.show()



**сохраним командой:**

#SAVING  
df\_composite.select("hole\_id", "depth\_from", "depth\_to", "sample\_id", "result", "composite")\  
 .write\  
 .format("csv")\  
 .save("final\_project/composite.csv")  
  
#checking a loading our CSV  
df = spark.read.format("csv")\  
 .option("header", "false")\  
 .schema("hole\_id STRING, depth\_from float, depth\_to float, sample\_id STRING, result float, composite integer")\  
 .load("final\_project/composite.csv")\  
 .sort("hole\_id","depth\_from")

# **РАСЧЕТ КОМПОЗИТОВ (SQL +Window) -- SUCCESS**

**По результату получения обратной связи от преподавателя было решено доработать предыдущее решение, применив Window функции lead, lag.**

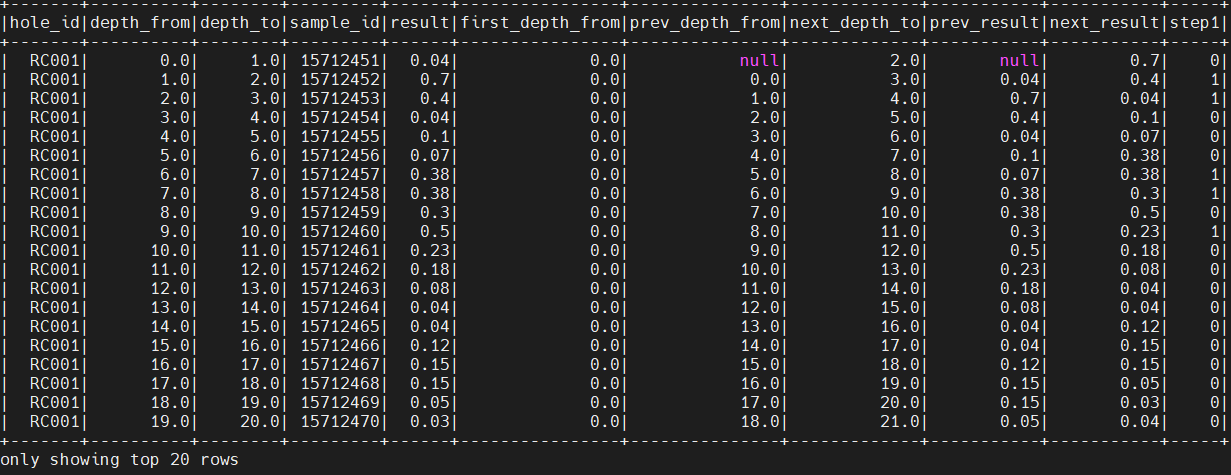
**Это позволило избежать применения многочисленных sub-query в SQL сценариях, что значительно повысило быстродействие кода, сделало его более понятным и оптимальным.**

Cb = 0.37  
Mr = 2  
#STEPS EXECUTING  
  
#step1  
from pyspark.sql.functions import udf  
def step1(r):  
 if r >= Cb:  
 result = 1  
 else:  
 result = 0  
 return result  
udf\_step1 = udf(step1)

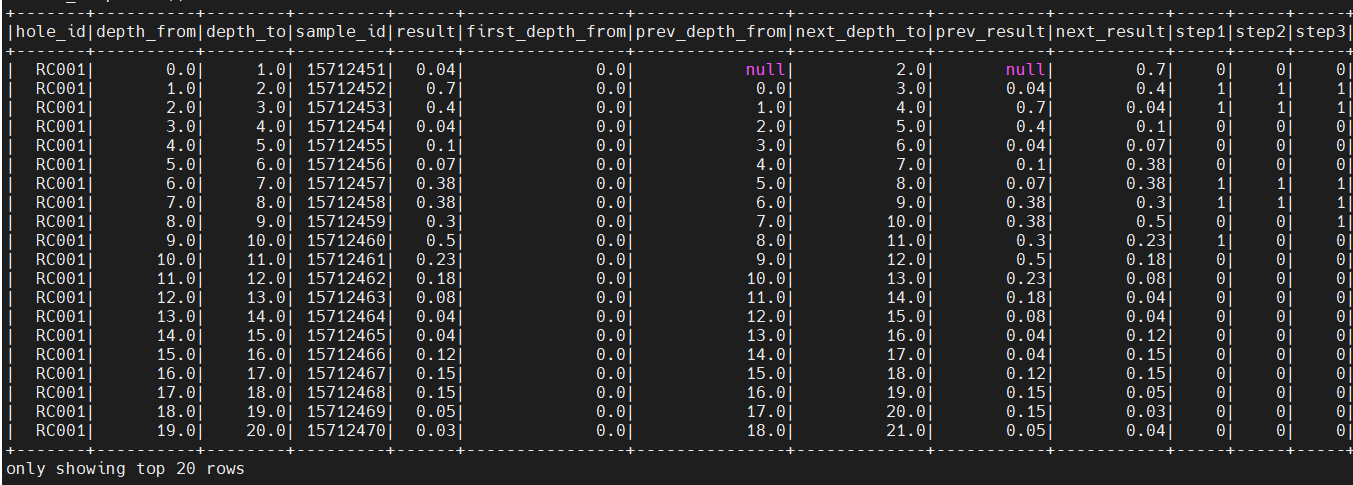
**Данная udf-функция будет использована для вычисления столбца step1.**

**Для вычисления шагов 2-4 нам понадобится обогатить наши данные доп. вычисляемыми столбцами на основе window-функций.**

from pyspark.sql.functions import lag,lead,min  
from pyspark.sql.window import Window  
window = Window.partitionBy("hole\_id").orderBy("depth\_from")

df = sorted\_df \  
 .withColumn("first\_depth\_from", min("depth\_from").over(window))\  
 .withColumn("prev\_depth\_from", lag("depth\_from", 1).over(window))\  
 .withColumn("next\_depth\_to", lead("depth\_to", 1).over(window))\  
 .withColumn("prev\_result", lag("result", 1).over(window))\  
 .withColumn("next\_result", lead("result", 1).over(window))\  
 .withColumn("step1", udf\_step1(join\_df["result"]))\  
 .sort("hole\_id","depth\_from").sort("hole\_id","depth\_from")  
  
df.createOrReplaceTempView("df1")  
spark.sql("select \* from df1").show()  
  


#step2  
sql2 = "select df1.\*" \  
 ", case when (df1.depth\_from <> df1.first\_depth\_from) " \  
 "and (df1.step1 = 1) and " \  
 "((((df1.depth\_to - df1.prev\_depth\_from)>=" + str(Mr) + ") " \  
 "and (df1.prev\_result >= " + str(Cb) + ")) " \  
 "or (((df1.next\_depth\_to - df1.depth\_from)>=" + str(Mr) + ") " \  
 "and (df1.next\_result >= " + str(Cb) + "))) " \  
 "then 1 else 0 end as step2 from df1 order by df1.hole\_id, df1.depth\_from"  
  
spark.sql(sql2).createOrReplaceTempView("df2")  
spark.sql("select \* from df2").show()  
  
#step3  
sql3 = "select df2.\*, " \  
 "case when " \  
 "(df2.depth\_from <> df2.first\_depth\_from) " \  
 "and df2.result < " + str(Cb) + " " \  
 "and df2.prev\_result>=" + str(Cb) + " " \  
 "and df2.next\_result>=" + str(Cb) + " " \  
 "then 1 else step2 end step3 from df2 order by df2.hole\_id, df2.depth\_from"  
  
#before final step we need to add other columns  
df\_temp = spark.sql(sql3)  
df\_temp.show()  
  
df\_temp = df\_temp\  
 .withColumn("prev\_step1", lag("step1", 1).over(window))\  
 .withColumn("next\_step1", lead("step1", 1).over(window))\  
 .withColumn("prev\_step3", lag("step3", 1).over(window))\  
 .withColumn("next\_step3", lead("step3", 1).over(window))  
# df\_temp.show()  
  
df\_temp.createOrReplaceTempView("df3")  
spark.sql("select \* from df3").show()



# step4  
sql4 = "select df3.\*, " \  
 "case when " \  
 "(df3.depth\_from <> df3.first\_depth\_from) " \  
 "and ((df3.step1 = 1) or (df3.step3 = 1)) " \  
 "and ( (df3.prev\_step3<>0) " \  
 "or (df3.next\_step3<>0) " \  
 "or ( (df3.prev\_step1<>0) " \  
 "and (df3.next\_step1<>0) " \  
 "and ( df3.next\_depth\_to - df3.prev\_depth\_from ) >= " + str(Mr) + ")) " \  
 "then 1 else 0 end composite from df3 order by df3.hole\_id, df3.depth\_from"  
  
spark.sql(sql4).createOrReplaceTempView("df4")  
df\_composite = spark.sql("select \* from df4 order by hole\_id, depth\_from")  
df\_composite.show()

**Сохраняем вычисленные датафрейм в CSV.**

#SAVING  
df\_composite.select("hole\_id", "depth\_from", "depth\_to", "sample\_id", "result", "composite")\  
 .write\  
 .format("csv")\  
 .save("final\_project/composite.csv")  
  
#checking a loading our CSV  
df = spark.read.format("csv")\  
 .option("header", "false")\  
 .schema("hole\_id STRING, depth\_from float, depth\_to float, sample\_id STRING, result float, composite integer")\  
 .load("final\_project/composite.csv")\  
 .sort("hole\_id","depth\_from")

# **ПРИЛОЖЕНИЕ**

Для отладки SQL-логики использована SQL Server Management Studio.

Ниже приведен алгоритм расчета композитов, реализованный в T-SQL

--this script is debugging of working code for final project

--data were loaded into [df] table

declare @Cb as float = 0.37

declare @Mr as int = 2

--;with sorted\_df as (select \* from df order by 1,2)

--IF OBJECT\_ID('dbo.#df') IS NOT NULL

DROP TABLE #df

create table #df(hole\_id nvarchar(16), depth\_from float, depth\_to float, sample\_id nvarchar(16), result float)

insert into #df

select \* from df order by 1,2

--step1

;with df1 as (select

\*

,case

when [result] >= @Cb then 1 else 0

end as step1

from #df

),

df2 as (

select

df1.\*

,case

when

(df1.depth\_from <> (select min(t2.depth\_from) from df1 t2 where t2.hole\_id = df1.hole\_id)) and

(df1.step1 = 1) and (

(

((df1.depth\_to - (select t2.depth\_from from df1 t2 where t2.hole\_id = df1.hole\_id and t2.depth\_to = df1.depth\_from ))>=@Mr) and ((select t2.result from df1 t2 where t2.hole\_id = df1.hole\_id and t2.depth\_to = df1.depth\_from)>=@Cb)

)

or

(

(((select t2.depth\_to from df1 t2 where t2.hole\_id = df1.hole\_id and t2.depth\_from = df1.depth\_to ) - df1.depth\_from)>=@Mr) and ((select t2.result from df1 t2 where t2.hole\_id = df1.hole\_id and t2.depth\_from = df1.depth\_to)>=@Cb)

)

)

then 1

else 0

end as step2

from df1

),

df3 as (

select

df2.\*

,case

when

(df2.depth\_from <> (select min(t2.depth\_from) from df2 t2 where t2.hole\_id = df2.hole\_id))

and result < @Cb

and (select t2.result from df2 t2 where t2.hole\_id = df2.hole\_id and t2.depth\_from = df2.depth\_to)>=@Cb

and (select t2.result from df2 t2 where t2.hole\_id = df2.hole\_id and t2.depth\_to = df2.depth\_from)>=@Cb

then 1

else step2 end step3

from df2)

select

df3.\*

,case

when

(df3.depth\_from <> (select min(t2.depth\_from) from df3 t2 where t2.hole\_id = df3.hole\_id))

and ((df3.step1 = 1) or (df3.step3 = 1))

and

(

((select t2.step3 from df3 t2 where t2.hole\_id = df3.hole\_id and t2.depth\_to = df3.depth\_from)<>0)

or ((select t2.step3 from df3 t2 where t2.hole\_id = df3.hole\_id and t2.depth\_from = df3.depth\_to)<>0)

or (

((select t2.step1 from df3 t2 where t2.hole\_id = df3.hole\_id and t2.depth\_to = df3.depth\_from)<>0)

and ((select t2.step1 from df3 t2 where t2.hole\_id = df3.hole\_id and t2.depth\_from = df3.depth\_to)<>0)

and ( (select t2.depth\_to from df3 t2 where t2.hole\_id = df3.hole\_id and t2.depth\_from = df3.depth\_to)

-(select t2.depth\_from from df3 t2 where t2.hole\_id = df3.hole\_id and t2.depth\_to = df3.depth\_from) ) >= @Mr

)

) then 1 else 0 end composite

from df3 order by 1,2