## Udf和缺失值处理

import org.apache.spark.sql.functions.{udf => normalUdf}

import scala.reflect.runtime.universe.TypeTag

/\*\*

\* editor: xuhao

\* date: 2018.03.20 09:00:00

\*/

/\*\*

\* 用于处理sparkSQL中创建udf函数时null的输出问题，使udf函数可以实现null => null的映射

\* ----

\* 描述：

\* sparkSQL中的udf有以下弊端：

\* 输入可能带有null值，但输出不行，数据会发生信息损失。

\* eg,

\* {{{

\* Long(with null) => Double中不能有null => null的映射：

\* udf((s: Long) => s match {

\* case null => null

\* case e => e / width

\* } // not compile

\* }}}

\* ----

\* 示例：

\* NullableFunctions的udf暗含了的null => null的映射，数据不会损失信息

\* eg,

\* {{{

\* NullableFunctions.udf((s: Long) => s / width)

\* }}}

\* ----

\* The main source idea by Martin Senne's answer on Stack OverFlow.

\*/

object NullableFunctions {

def udf[RT: TypeTag, A1: TypeTag](f: Function1[A1, RT])

: UserDefinedFunction = normalUdf[Option[RT],A1](

(i: A1) => i match {

case null => None

case s => Some(f(s))

})

def udf[RT: TypeTag, A1: TypeTag, A2: TypeTag](f: Function2[A1, A2, RT])

: UserDefinedFunction = normalUdf[Option[RT], A1, A2](

(i1: A1, i2: A2) => (i1, i2) match {

case (null, \_) => None

case (\_, null) => None

case (s1, s2) => Some(f(s1, s2))

})

}

## 多列输入的udf如何写

**VectorAssembler中的源码：**

// Data transformation.  
**val** assembleFunc = *udf* { r: Row =>  
 VectorAssembler.*assemble*(r.toSeq: \_\*)  
}  
**val** args = $(*inputCols*).map { c =>  
 schema(c).dataType **match** {  
 **case** DoubleType => dataset(c)  
 **case** \_: VectorUDT => dataset(c)  
 **case** \_: NumericType | BooleanType => dataset(c).cast(DoubleType).as(s"**$**{c}\_double\_**$**uid")  
 }  
}  
  
dataset.select(*col*("\*"), assembleFunc(*struct*(args : \_\*)).as($(*outputCol*), metadata))

## 一列输入一个string输入的Expression如何写

**case class** StringSplit(str: Expression, pattern: Expression)  
 **extends** BinaryExpression **with** ImplicitCastInputTypes {  
  
 **override def** left: Expression = str  
 **override def** right: Expression = pattern  
 **override def** dataType: DataType = *ArrayType*(StringType)  
 **override def** inputTypes: Seq[DataType] = *Seq*(StringType, StringType)  
  
 **override def** nullSafeEval(string: Any, regex: Any): Any = {  
 **val** strings = string.asInstanceOf[UTF8String].split(regex.asInstanceOf[UTF8String], -1)  
 **new** GenericArrayData(strings.asInstanceOf[Array[Any]])  
 }  
  
 **override def** genCode(ctx: CodeGenContext, ev: GeneratedExpressionCode): String = {  
 **val** arrayClass = *classOf*[GenericArrayData].getName  
 nullSafeCodeGen(ctx, ev, (str, pattern) =>  
 // Array in java is covariant, so we don't need to cast UTF8String[] to Object[].  
 s"""**$**{ev.value} = new **$**arrayClass(**$**str.split(**$**pattern, -1));""")  
 }  
  
 **override def** prettyName: String = "split"  
}

起始lit也很好用，自己试验的发现udf中以lit形式传递参数两者差不多（其实测的时候要快1.6%，忽略不计了）。

另外我发现测试一个较大数据的transformation的时候select(“”).count也没用，还是要用到一个agg+collect才好

**val** jd = **new** java.util.Random(1123L)  
  
**val** move1 = *udf*((id: Int, moveLength: Int) => {  
 id - moveLength  
})  
  
**val** move2 = *udf*((id: Int) => {  
 id - 2  
})  
  
**var** i = 0  
**var** time1 = 0L  
**var** time2 = 0L  
  
**val** data = *sqlc*.createDataFrame(Array.*fill*(100000)(*Tuple1*(jd.nextLong()))).toDF("id")  
data.cache()  
*println*(data.count())  
  
**while**(i < 20) {  
 **val** startTime1 = System.*nanoTime* //系统纳米时间  
  
 data.select(*sum*(move1(*col*("id"), *lit*(2)))).head().get(0)  
  
 **val** endTime1=System.*nanoTime* **val** delta1= endTime1 - startTime1  
 time1 += delta1/1000000  
  
 **val** startTime2 = System.*nanoTime* //系统纳米时间  
  
 data.select(*sum*(move2(*col*("id")))).head().get(0)  
  
 **val** endTime2=System.*nanoTime* **val** delta2= endTime2 - startTime2  
 time2 += delta2/1000000  
 i += 1  
}  
  
*println*("time1", time1)  
*println*("time2", time2)

## 开窗函数

一个例子

import org.apache.spark.sql.Row

import org.apache.spark.sql.types.{IntegerType, StringType, StructField, StructType}

val lst = Array.range(10, 20, 2).flatMap(i => Array("A", "B", "C", "D").map(x => (x, i)))

val rdd = sc.parallelize(lst).map(Row.fromTuple)

val bindDF = hqlc.createDataFrame(rdd, StructType(Array(StructField("category", StringType), StructField("count", IntegerType))))

bindDF.registerTempTable("tableName")

val sqlExpr = "SELECT `category`, `count` from (SELECT `category`, `count`, rank() OVER (PARTITION BY category ORDER BY count DESC) as rank FROM `tableName`) tmp WHERE rank <= 2"

val resultDF = hqlc.sql(sqlExpr)

resultDF.show()

## Schema的apply方法

Schema中有apply方法注意用，要比fieldIndex在索引省劲

## groupBy之后重命名的问题

1）使用sql语句最方便

2）如果不使用sql语句，下面的方式可以

DF.agg(*sum*(*col*("count")).alias("count"))

## Spark对Where子查询的支持

1）running this query in Spark shell but it gives me error

sqlContext.sql(

"select sal from samplecsv where sal < (select MAX(sal) from samplecsv)"

).collect().foreach(println)

error:

java.lang.RuntimeException: [1.47] failure: ``)'' expected but identifier MAX found

select sal from samplecsv where sal < (select MAX(sal) from samplecsv) ^ at scala.sys.package$.error(package.scala:27) Can anybody explan me,thanks

2）Spark SQL should support both correlated and uncorrelated subqueries. See SubquerySuite for details. Some examples include:

select \* from l where exists (select \* from r where l.a = r.c)

select \* from l where not exists (select \* from r where l.a = r.c)

select \* from l where l.a in (select c from r)

select \* from l where a not in (select c from r)

3)Unfortunately as for now (Spark 2.0) it is impossible to express the same logic using DataFrame DSL.

2.0和2.0+支持一下子查询

select \* from l where exists (select \* from r where l.a = r.c)

select \* from l where not exists (select \* from r where l.a = r.c)

select \* from l where l.a in (select c from r)

select \* from l where a not in (select c from r

2.0-只支持有继承关系的子查询 Hive <= 0.12

SELECT col FROM (SELECT \* FROM t1 WHERE bar) t2

It simply doesn't support subqueries in the WHERE clause.Generally speaking arbitrary subqueries (in particular correlated subqueries) couldn't be expressed using Spark without promoting to Cartesian join.

具体sparkSQL支持的子查询语句可以查看apache/spark项目的

[spark](https://github.com/apache/spark/tree/df89f1d43d4eaa1dd8a439a8e48bca16b67d5b48)/[sql](https://github.com/apache/spark/tree/df89f1d43d4eaa1dd8a439a8e48bca16b67d5b48/sql)/[core](https://github.com/apache/spark/tree/df89f1d43d4eaa1dd8a439a8e48bca16b67d5b48/sql/core)/[src](https://github.com/apache/spark/tree/df89f1d43d4eaa1dd8a439a8e48bca16b67d5b48/sql/core/src)/[test](https://github.com/apache/spark/tree/df89f1d43d4eaa1dd8a439a8e48bca16b67d5b48/sql/core/src/test)/[scala](https://github.com/apache/spark/tree/df89f1d43d4eaa1dd8a439a8e48bca16b67d5b48/sql/core/src/test/scala)/[org](https://github.com/apache/spark/tree/df89f1d43d4eaa1dd8a439a8e48bca16b67d5b48/sql/core/src/test/scala/org)/[apache](https://github.com/apache/spark/tree/df89f1d43d4eaa1dd8a439a8e48bca16b67d5b48/sql/core/src/test/scala/org/apache)/[spark](https://github.com/apache/spark/tree/df89f1d43d4eaa1dd8a439a8e48bca16b67d5b48/sql/core/src/test/scala/org/apache/spark)/[sql](https://github.com/apache/spark/tree/df89f1d43d4eaa1dd8a439a8e48bca16b67d5b48/sql/core/src/test/scala/org/apache/spark/sql)/SubquerySuite.scala文件

4）通过join来实现子查询功能

Since subquery performance is usually a significant issue in a typical relational system and every subquery can be expressed using JOIN there is no loss-of-function here.

示例：

/\*\*

\* 创造一个数据

\* [id, class, name, score]

\*/

val lst = List(

Array(1, 1, "张三", 92),

Array(2, 1, "李四", 70),

Array(3, 1, "王二", 30),

Array(4, 2, "赵武", 100),

Array(5, 2, "周六", 85),

Array(6, 2, "孙七", 40),

Array(7, 3, "吴八", 96),

Array(8, 3, "郑九", 85),

Array(9, 3, "冯十", 65))

val rowRdd = sc.parallelize(lst).map(Row.fromSeq(\_))

val df = sqlc.createDataFrame(rowRdd,

StructType(Array(StructField("id", IntegerType),

StructField("class", IntegerType),

StructField("name", StringType),

StructField("score", IntegerType)

)))

df.show()

df.registerTempTable("testscore")

val subSql\_byWhere = "SELECT `testscore`.\* FROM `testscore` WHERE score = (SELECT MAX(score) from `testscore`)"

val SubSql\_byJoin = "SELECT `testscore`.\* FROM `testscore` JOIN (select max(`score`) as score FROM `testscore`) tmp ON testscore.score = tmp.score"

// val result = sqlc.sql(SubSql\_byJoin) // not compile

val result = sqlc.sql(SubSql\_byJoin)

result.show()

5）为什么不支持子查询

我估计可能和lazy模式有关，子查询的模式可能和lazy有关，WHERE子查询如果实现应该需要获取数据的，即出发action的，而spark.sql只是transformation操作，由于某种机制（估计，这个我也不清楚两者暂时不能混在一起）。

## SparkSQL中udf函数报错处理：No TypeTag available

描述：

val arr = List(

Array("1", "00:00:00"),

Array("1", "01:2:10"),

Array("1", "02:2:10"),

Array("1", "01:2:10"),

Array("1", "3:2:10"),

Array("1", "5:2:10"),

Array("1", "00:2:10"),

Array("1", "01:59:10"),

Array("1", null),

Array("1", "10:2:32"),

Array("1", "23:00:10"),

Array("1", "A"))

val rdd = sc.parallelize(arr).map(Row.fromSeq(\_))

var df = sqlc.createDataFrame(rdd, StructType(Array(StructField("id", StringType), StructField("dayTime", StringType))))

df = df.withColumn("gmtStart", lit("1970-01-01"))

val paste = udf((s1: String, s2: String) => s1 + "," + s2)

df = df

.withColumn("binnerTime", paste(col("gmtStart"), col("dayTime")))

.withColumn("binnerStampTime", unix\_timestamp(col("binnerTime"), "yyyy-MM-dd HH:mm:ss").+(28800))

.drop("gmtStart")

.drop("binnerTime")

编译报错：

Error1: No TypeTag available for String

val paste = udf((s1: String) => s1)

Error2: not enough arguments for method udf: (implicit evidence$2:reflect.runtime.universe.TypeTag[String], implicit evidence$3: reflect.runtime.universe.TypeTag[String])org.apache.spark.sql.UserDefinedFunction.

Unspecified value parameters evidence$2, evidence$3.

val paste = udf((s1: String) => s1)

原因分析：

无法识别String这一类型，很可能时spark(1.6.0)版本和scala(2.11.0)版本不一致造成了。换成2.10.6

解决方式:

卸载scala2.11.0安装scala2.10.6

## Spark sql中的Row模式以及get元素的方法

## Spark sql利用ddl语句cast类型

### 类型转换操作

一般sql：SELECT CAST(score AS CHAR(20)) as score\_string FROM `testscore`;

Spark sql：SELECT CAST(score AS string) as score\_string FROM `testscore`

### 通过case实现map + match类的操作

select \*, cast(`年龄` as string) as `年龄string` ,cast(`任职时间` as string) as `任职时间string` ,

cast(case when 利润 >= 4500 then 1.0 else 0.0 end as string) as `利润达标`

from `关系型数据库\_1\_gFmImc3F`

### 通过case语句实现where（filter）操作

val df = sqlc.createDataFrame(Seq((1, 1, "张三", 92)

,(2, 1, "李四", 70)

,(3, 1, "王二", 30)

,(4, 2, "赵武", 100)

,(5, 2, "周六", 85)

,(6, 2, "孙七", 40)

,(7, 3, "吴八", 96)

,(8, 3, "郑九", 85)

,(9, 3, "冯十", 65))).toDF("id", "class", "name", "score")

val countCol = "score"

val conditions = Array(("张三", 95.0), ("郑九", 90.0), ("冯十", 90)).map(tup => tup.\_1 + s"') then $countCol > ${tup.\_2}")

**val u = "case when (name = '张三') then score > 95 " +**

**"when (name = '郑九') then score > 90 " +**

**"when (name = '冯十') then score > 90 else TRUE end"**

val filterSentence = conditions.mkString("case when (name = '", " when (name = '", " else TRUE end")

df.show()

df.filter(filterSentence).show()

### 一般的列转换的map操作

SELECT \* from (SELECT \*,cast(`gps业务` as double)/ (`gps业务` + `信令业务`) as gps\_percent, cast(`信令业务` as double)/ (`gps业务` + `信令业务` ) as 信令\_percent FROM `数据字段分裂new\_1\_f63VX2gB`) a WHERE (a.gps\_percent >= 0.7 or a.信令\_percent >= 0.7 )

## 创建一个空DataFrame

val rdd = sc.parallelize(Seq.empty[Array[Any]]).map(Row.fromSeq(\_))

val df = sqlc.createDataFrame(Seq(("1", 1.0), ("2", 2.0)))

df.show()

val emptyDF = sqlc.createDataFrame(rdd, df.schema)

emptyDF.show()

println(emptyDF.schema)

**能够编译，可见没有数据触发的情况下，是不会报某个类型can not cast to 另一个类型的**

## 创建一个树结构

今天居然通过sql先学习到了树结构如何搞。

这里是泛型嵌套。是为LogicalPlan等做准备的。

**abstract class** TreeNode[BaseType <: TreeNode[BaseType]] **extends** Product {  
 self: BaseType =>  
  
 **val** *origin*: Origin = CurrentOrigin.*get  
  
 /\*\*  
 \* Returns a Seq of the children of this node.  
 \* Children should not change. Immutability required for containsChild optimization  
 \*/* **def** children: Seq[BaseType]  
  
 **lazy val** *containsChild*: Set[TreeNode[\_]] = children.toSet

## DataFrame跑到rdd中用schema.fieldIndex会nullPoint

为什么还不知道。

1. 状况

1） val rdd = df.rdd.map(r => {

val key = keyArray.map(name => {

val value = r.getAs[String](name)

if(value == null) "" else value.toString

}).mkString(separator)

val typeName = r.get(**df.schema.**fieldIndex(typeCol)).asInstanceOf[String]

**报空指针异常**

2） **val schema = df.schema**

val rdd = df.rdd.map(r => {

val key = keyArray.map(name => {

val value = r.getAs[String](name)

if(value == null) "" else value.toString

}).mkString(separator)

val typeName = r.get(schema.fieldIndex(typeCol)).asInstanceOf[String]

**没问题**

1. 分析分析源码猜测原因可能和lazy有关是：

1）证据1是：

**DataFrame中**

**def schema: StructType = queryExecution.analyzed.schema**

**而GenericRowWithSchema中**

class GenericRowWithSchema(values: Array[Any], override val schema: StructType)

extends GenericRow(values) {

。。。

override def fieldIndex(name: String): Int = schema.fieldIndex(name)

}因此row.fieldIndex是可以的

**猜测可能是queryExecution.analyzed.schema调用**

**lazy** val analyzed: LogicalPlan = sqlContext.analyzer.execute(logical)

**其中analyzed是lazy模式的，在rdd中调用会出现空指针的问题。**

**2）证据2是：**

**val getIndex = (name: String) => df.schema.fieldIndex(name)**

val schema = df.schema

val rdd = df.rdd.map(r => {

val key = keyArray.map(name => {

val value = r.getAs[String](name)

if(value == null) "" else value.toString

}).mkString(separator)

val typeName = r.get(getIndex(typeCol)).asInstanceOf[String]

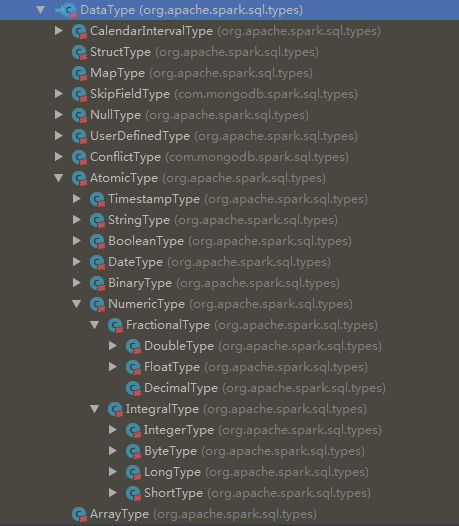
val count = r.get(getIndex(countCol)).toString.toDouble

(key, (typeName, count))

})

**同样出现该问题。之所以val schema = df.schema没问题是因为再赋值的过程中触发了lazy的操作。**

## SparkSQL中DataFrame的类型体系



## Spark.sql自定义类型之VectorUDT

### Spark可以实现自定义类型

比如之前我用过的简单的person类

### VectorUDT类型

#### 注意vectorUDT的hashCode对所有的实例是同样的常数。

// see [SPARK-8647], this achieves the needed constant hash code without constant no.  
**override def** hashCode(): Int = *classOf*[VectorUDT].getName.hashCode()

此时可以保证类型判断。

### 怎样写一个好的UDT

## SparkSQL中DataFrame的save和write

**def** save(  
 source: String,  
 mode: SaveMode,  
 options: Map[String, String]): Unit = {  
 write.format(source).mode(mode).options(options).save()  
}  
  
*/\*\*  
 \* Adds the rows from this RDD to the specified table, optionally overwriting the existing data.  
 \** ***@group*** *output  
 \** ***@deprecated*** *As of 1.4.0, replaced by  
 \** `*write().mode(SaveMode.Append|SaveMode.Overwrite).saveAsTable(tableName)*`*.  
 \* This will be removed in Spark 2.0.  
 \*/*@deprecated("Use write.mode(SaveMode.Append|SaveMode.Overwrite).saveAsTable(tableName). " +  
 "This will be removed in Spark 2.0.", "1.4.0")  
**def** insertInto(tableName: String, overwrite: Boolean): Unit = {  
 write.mode(**if** (overwrite) SaveMode.*Overwrite* **else** SaveMode.*Append*).insertInto(tableName)  
}

## sparkSQL中两个表模式合并

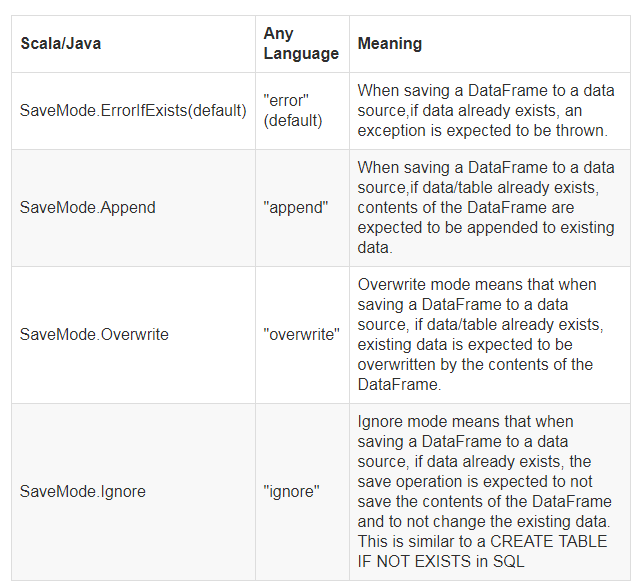
1）例子

// This is used to implicitly convert an RDD to a DataFrame.  
**import** *sqlc*.implicits.\_  
  
// Create a simple DataFrame, store into a partition directory  
**val** squaresDF = *sqlc*.sparkContext.makeRDD(1 to 5, 1).map(i => (i, i \* i)).toDF("value", "square")  
squaresDF.write.mode("overwrite").parquet("data/test\_table/key=1")  
  
// Create another DataFrame in a new partition directory,  
// adding a new column and dropping an existing column  
**val** cubesDF = *sqlc*.sparkContext.makeRDD(6 to 10, 1).map(i => (i, i \* i \* i)).toDF("value", "cube")  
cubesDF.write.mode("overwrite").parquet("data/test\_table/key=1")  
  
// Read the partitioned table  
**val** mergedDF = *sqlc*.read.option("mergeSchema", "true").parquet("data/test\_table")  
mergedDF.printSchema()  
mergedDF.show()

2）该方法在平台上可行，本地不行（无法写入）。

3）该方法还是垂直的合并，本质上。

4）不过还好学会了另一招write的mode设定，还是不错的。



## Spark添加自增列

rawDataFrame.select(*col*("\*"), *monotonicallyIncreasingId*().as(idCol))