# 第七章Spark SQL

本章介绍的是Spark 处理结构化数据、半结构化数据的一个高级模块——Spark SQL。结构化数据就是指记录内容具有明确的结构信息且数据集内的每一条记录都符合结构规范的数据集合，是由二维表结构来逻辑表达和实现的数据集合。可以类比传统数据库表来理解该定义，所谓的“明确结构”即是由预定义的表头表示的每一条记录由哪些字段组成以及各个字段的名称、类型、属性等信息。当数据符合这样的条件时，Spark SQL就会使得针对这些数据的读取和查询变得更加简单高效。

本章重点讲解Spark SQL概述，包括架构、特点和程序入口等一系列知识；以及Spark SQL的DataFrame编程抽象，从DataFrame与RDD的区别，如何创建DataFrame，RDD如何转换成DataFrame以及DataFrame的操作几个方面进行了说明；随后对DataFrame的数据源进行了归纳总结。

## Spark SQL编程入门

### Spark SQL概述

Spark SQL是用于结构化数据、半结构化数据处理的Spark高级模块，可用于各种结构化数据源，例如在JSON（半结构化）文件、CSV文件、ORC文件（ORC文件格式是一种Hive的文件存储格式，可以提高Hive表的读、写以及处理数据的性能）、Hive表 、Parquest文件（新型列式存储格式，具有降低查询成本、高效压缩等优点，广泛用于大数据存储、分析领域）中读取数据，然后在Spark程序内通过SQL语句对数据进行交互式查询，进而实现数据分析需求，也可以通过标准数据库连接器（JDBC/ODBC）连接传统关系型数据库，取出并转化关系型数据库表，利用Spark SQL进行数据分析。

1. **架构分析**

Spark SQL与传统的DBMS的查询优化器+执行器的架构较为相似，只不过其执行器是在分布式环境中实现，并采用Spark作为执行引擎。Spark SQL的查询优化是Catalyst，其基于Scala语言开发的，可以灵活利用Scala原生的语言特性方便地扩展功能，奠定了Spark SQL的发展空间。Catalyst将SQL翻译成最终的执行计划，并在这个过程中进行查询优化。这里和传统不太一样的地方就在于，SQL经过查询优化器最后转换为可执行的查询计划，传统的DB就可以执行这个查询计划了。而Spark SQL最后是会在Spark内将执行计划转换为Spark的有向无环图DAG再执行。Catalyst的整体架构如下图所示。



图7-1 Spark SQL查询引擎Catalyst的架构

由图7-1中可以看到整个Catalyst是Spark SQL的调度核心，其性能优劣将影响到整体的性能。Catalyst主要的实现组件如下：

1）SQLParse（Scala实现）：完成SQL语句的语法解析功能，目前只提供了一个简单的SQL解析器，将输入的SQL，解析为Unresolved logical plan（未被解析的逻辑计划）。

2）Analyzer：主要完成绑定工作，将不同来源的Unresolved LogicalPlan和数据元数据（如Hive metastore、Schema Catalog）进行绑定，生成Resolved LogicalPlan。

3）Optimizer：对Resolved LogicalPlan进行优化，生成Optimized LogicalPlan。

4）Planner：将Optimized LogicalPlan转换为PhysicalPlan。

5）CostModel：主要根据过去的性能统计数据，选择最佳的物理执行计划。

Catalyst遵循着传统数据库的查询解析步骤，对SQL进行解析，转换为逻辑查询计划和物理查询计划，最终转换为Spark的DAG执行，Catalyst的执行流程如下图所示。



图7-2 Catalyst的执行流程

从图7-2中可知SQLParser将SQL语句转换为逻辑查询计划，Analyzer对逻辑查询计划进行属性和关系关联检验，之后Optimizer通过逻辑查询优化将逻辑查询计划转换为优化的逻辑查询计划，QueryPlanner将优化的逻辑查询计划转换为物理查询计划，prepareForExecution调整数据分布，最后将物理查询计划转换为执行计划进入Spark执行任务。

1. **Spark SQL特点**

①数据兼容方面：不但兼容Hive，还可以从RDD、Parquet文件、JSON文件中获取数据。可以在Scala代码里访问Hive元数据，执行Hive语句，并且把结果取回作为RDD使用。支持Parquet文件的读写，且保留Schema。

②组件扩展方面：无论是SQL的语法解析器、分析器还是优化器都可以重新定义，进行扩展。

③性能优化方面：除了采取内存列存储，动态字节码生成等优化技术，还采取了内存缓存数据。

④支持多种语言：包括Scala、Java、Python、R等，可以在Scala代码中写SQL，支持简单的SQL语法检查，能把RDD转化为DataFrame存储起来。

1. **程序主入口——SparkSession**

Spark SQL模块的编程主入口点是SparkSession，SparkSession对象不仅为用户提供了创建DataFrame对象、读取外部数据源并转化为DataFrame对象以及执行sql查询的API，还负责记录着用户希望Spark应用如何在Spark集群运行的控制、调优参数，是SparkSQL的上下文环境，是运行的基础。

如以下代码所示，可以通过SparkSession.builder()创建一个基本的SparkSession对象，并为该Spark SQL应用配置一些初始化参数，例如设置应用的名称以及通过config方法配置相关运行参数。

import org.apache.spark.sql.SparkSession

val sparkSession = SparkSession

.builder()

.appName("Spark SQL basic example")

.config("spark.some.config.option", "some-value")

.getOrCreate()

// 引入spark.implicits.\_，以便于RDDs和DataFrames之间的隐式转换

import spark.implicits.\_

Spark 2.0中的SparkSession为Hive提供了强大的内置支持，包括使用HiveQL编写查询语句，访问Hive UDF以及从Hive表读取数据的功能。

SparkSession为用户提供了直接执行sql语句的SparkSession.sql(sqlText:String)方法，sql语句可直接作为字符串传入sql()方法中，sql查询所得到的结果依然是DataFrame对象。在Spark SQL模块上直接进行sql语句的查询需要首先将标志着结构化数据源的DataFrame对象注册成临时表，进而在sql语句中对该临时表进行查询操作，具体步骤如下：

scala> val df = sparkSession.read.json("/home/ubuntu01/people.json")

//调用DataFrame的createOrReplaceTempView方法，将df注册成people临时表

df.createOrReplaceTempView("people")

//调用sparkSession提供的sql接口，对people临时表进行sql查询，sql()返回的也是DataFrame对象

val sqlDF = sparkSession.sql("SELECT \* FROM people")

sqlDF.show()

// +-----+------------+

// | age| name|

// +-----+----------+

// |null|Michael|

// | 30| Andy|

// | 19| Justin|

// +-----+----------+

### 核心抽象DataFrame

RDD是作为Spark平台一种基本、通用的数据抽象，基于其不关注元素内容及结构的特点，我们对结构化数据、半结构化数据、非结构化数据一视同仁，都可转换为由同一类型元素组成的RDD。但是作为一种通用、普适的工具，其必然无法高效、便捷地处理一些专门领域具有特定结构特点的数据，于是Spark在推出基础、通用的RDD编程后，提供了Spark四大高级模块之一Spark SQL所处理典型结构化数据源或可通过简易处理可形成鲜明结构的数据源的核心抽象——DataFrame。

**1、DataFrame概念**

DataFrame的定义与RDD类似，即都是Spark平台用以分布式计算的不可变分布式数据集合。从编程的角度来说，DataFrame是Spark SQL模块所需要处理的结构化数据的核心抽象，即在Spark程序中若想要使用简易的SQL接口对数据进行分析，首先需要将所处理的数据源转化为DataFrame对象，进而在DataFrame对象上调用各种API来实现需求，也可以将DataFrame注册成表，直接使用SQL语句在数据表上进行交互式查询。

DataFrame可以从许多结构化数据源加载并构造得到，如：结构化数据文件，Hive中的表，外部数据库，已有的DataFrame API支持多种高级程序语言Scala、Java、Python和R。在 Scala 和 Java中, DataFrame 由 DataSet 中的 RowS（多个 Row）来表示。 在Scala API中，DataFrame 仅仅是一个 Dataset[Row]类型的别名。然而，在 Java API中，用户需要去使用 Dataset<Row> 去代表一个 DataFrame。

**2、DataFrame与RDD的区别**

RDD和DataFrame均为Spark平台对数据的一种抽象，一种组织方式，但是两者的地位或者说设计目的却截然不同。RDD是整个Spark平台的存储、计算以及任务调度的逻辑基础，更具有通用性，适用于各类数据源，而DataFrame是只针对结构化数据源的高层数据抽象，其中在DataFrame对象创建过程中必须指定数据集的结构信息（Schema），所以DataFrame生来便是具有专业性的数据抽象，只能读取具有鲜明结构的数据集。DataFrame与RDD最大的不同在于，RDD仅是一条条数据的集合，并不了解每一条数据的内容是怎样的，而DataFrame明确的了解每一条数据有几个命名字段组成，即可以形象地理解为RDD是一条条数据组成的一维表，而DataFrame是每一行数据都有清晰的列划分的二维表。概念上来说，它和关系型数据库或者R和Python中data frame等价，只不过DataFrame在底层实现了更多的优化。

如下图7-3直观地体现了DataFrame和RDD的区别。RDD[Person]虽然以Person类为类型参数，但Spark平台本身不了解Person类的内部结构。而DataFrame却提供了详细的结构信息，使得Spark SQL得以洞察更多的结构信息，清楚地知道该数据集中包含哪些列，每列的名称和类型各是什么，从而对藏于DataFrame背后的数据源以及作用在DataFrame之上的变换进行了针对性的优化。RDD是分布式的Java对象的集合，DataFrame则是分布式的Row对象的集合。DataFrame除了提供了比RDD更丰富的算子操作以外，更能利用已知的结构信息来提升执行效率、减少数据读取以及执行计划的优化，而RDD由于无从得知所存数据元素的具体内部结构，所以RDD提供的API功能上没有DataFrame强大丰富且自带优化，称之为Low-level API，Spark Core只能在Stage层面进行简单、通用的流水线优化。

|  |
| --- |
| Person |
| Person |
| Person |

RDD[Person]

|  |  |  |
| --- | --- | --- |
| Name | Age | Height |
| String | Int | Double |
| String | Int | Double |
| String | Int | Double |

DataFrame[Person]

图7-3 DataFrame和RDD的区别

正如RDD的各种变换实际上只是构造RDD DAG，DataFrame的各种变化同样也是惰性的，它们并不直接求出计算结果，而是将各种变幻组装成与RDD DAG类似的逻辑查询计划，经过优化的逻辑执行计划被翻译为物理执行计划，并最终落实为RDD DAG。

### 创建DataFrame

应用程序使用创建的SparkSession对象提供的API可以有两种方式创建DataFrame，一种是调用SparkSession.read().json/csv/orc/parquet/jdbc方法从各种外部结构化数据源创建DataFrame对象；另一种是将已有的RDD转化为DataFrame对象。

**1、外部结构化数据源创建DataFrame**

作为示例，以下将基于JSON文件的内容创建DataFrame。

scala> val df = sparkSession.read.json("/home/ubuntu01/people.json")

df: org.apache.spark.sql.DataFrame = [age: string, institute: string ... 3 more fields]

//使用sparkSession对象提供的read()方法可读取数据源（read方法返回了一个DataFrameReader对象），进而通过json()方法标识数据源具体类型为Json格式

scala> df.show(6)

//在返回的DataFrame对象使用show(n)方法，展示数据集的前n条数据

+---+---------------------------+-----------+------------------+---------+

|age| institute| name| phone| sex|

+---+---------------------------+-----------+------------------+----------+

| 20| computer science..| Michael|16695845876| male|

| 21| medical college| cindy|18755698695| male|

| 25|chemical engineer..| andy|15585749562| female|

| 19| SEM| Aaron|16695845876| male|

| 24| school of materials| Abel|18755698695| male|

| 21| school of materials| Abner|16695845876| male|

+----+-------------------------+------------+------------------+----------+

only showing top 6 rows

**2、将RDDs转化为DataFrame**

Spark SQL支持已有的RDD转换为DataFrame对象，但是需要注意的是，并不是所有任意类型对象组成的RDD均可以转换为DataFrame对象，只有当组成RDD[T]的每一个T对象内部具有公有且鲜明的字段结构时，才能隐式或者显式地总结出创建DataFrame对象所必要的结构信息（Schema）进行转换，进而在DataFrame上调用RDD所不具备的强大丰富的API，或者执行简洁的SQL查询。

Spark SQL 支持两种不同的方法用于转换已存在的 RDD 成为 DataFrame。第一种方法是使用反射机制去自动推断一个包含指定的对象类型的RDD的Schema进行隐式转化。在你的Spark应用程序中，当你已知 Schema 时这个基于反射的方法可以让你的代码更简洁，并且运行效果良好。第二种方法是通过编程接口，允许构造一个 Schema ，然后把它应用到一个已存在的 RDD[Row] （将RDD[T]转化为Row对象组成的RDD）。然而这种方法更繁琐，但它允许你在运行之前不知道其中的列和对应的类型的情况下构建，提供了更高的自由度，更加灵活。

1. **使用反射机制推理出schema（结构信息）**

Spark SQL的Scala接口支持自动转换一个包含样例类（case class）的RDD为一个DataFrame对象。在样例类的声明中已预先定义了表的结构信息，内部通过反射机制读取case类的参数名为列名，case类不仅可以包含Int、Double、String这样的简单数据类型，也可以是嵌套的或者包含复杂类型，如序列或者数组。RDD可以隐式转换为一个DataFrame，并注册成一个表，表可以用于后续的SQL查询语句。

SparkSession的sql函数使应用程序运行SQL查询并返回DataFrame，只有存在case类时，会自动发生隐式转换，map操作的结果才能变成一个DataFrame，转换过程有三个重点：

①必须创建case类，只有case类才能隐式转换为DataFrame。

②必须生成DataFrame，进行注册临时表操作

③必须将标志着结构化数据源的DataFrame对象注册成临时表，才能在sql语句中对临时表进行查询操作。

scala> case class Person(name:String,age:Long)

defined class Person

scala> import spark.implicits.\_

import spark.implicits.\_

scala> val personRDD = sparkSession.sparkContext.textFile("/home/ubuntu01/people.txt").map(\_.split(",")).map(attributes => Person(attributes(0),attributes(1).trim.toInt))

personRDD: org.apache.spark.rdd.RDD[Person] = MapPartitionsRDD[11] at map at <console>:34

scala> val peopleDF = personRDD.toDF()

peopleDF: org.apache.spark.sql.DataFrame = [name: string, age: bigint]

scala> peopleDF.createOrReplaceTempView("people")

scala> val teenagersDF = sparkSession.sql("SELECT name,age FROM people WHERE age BETWEEN 13 AND 19")

teenagersDF: org.apache.spark.sql.DataFrame = [name: string, age: bigint]

scala> teenagersDF.map(teenager => "Name: " + teenager(0)).show()

+-----------------+

| value|

+-----------------+

|Name: Aaron|

|Name: chayu|

+-----------------+

RDD[Person]隐式转化为DataFrame的实质是，内部自动生成了包含着结构信息的Person样例类的编码器（encode），并将该编码器用于DataFrame的初始化，编码器对于DataFrame对象意义重大，用于将JVM对象转换为Spark SQL的对象，以及将对象序列化，以便缓存和网络传输。

1. **以编程的方式指定schema**

当case类不能提前定义时（例如：数据集的结构信息已经包含在每一行中、一个文本数据集的字段对不同用户来说需要被解析成不同的字段名），这时，DataFrame就可以通过编程方式创建，主要有以下三个步骤：

①根据需求从原始RDD中创建一个Rows的RDD。

②创建一个表示为StructType类型的Schema，匹配在第一步创建的RDD的Rows的结构。

③通过SparkSession提供的createDataFrame方法，应用Schema到Rows的RDD。

//导入Spark SQL的data types包

scala> import org.apache.spark.sql.types.\_

import org.apache.spark.sql.types.\_

//导入Spark SQL的Row包

scala> import org.apache.spark.sql.Row

import org.apache.spark.sql.Row

// 创建peopleRDD

scala> val peopleRDD =

sparkSession.sparkContext.textFile("/home/ubuntu01/people.txt")

peopleRDD: org.apache.spark.rdd.RDD[String] = /home/ubuntu01/people.txt MapPartitionsRDD[33] at textFile at <console>:36

//创建所需要的schema

scala> val schemaString = "name age"

schemaString: String = name age

//将schemaString按空格分隔返回字符串数组，对字符串数组进行遍历，并对数组中的每一个元素进一步封装成StructField对象，进而构成了Array[StructField]

scala> val fields = schemaString.split(" ").map(fieldName => StructField(fieldName,StringType,nullable = true))

fields: Array[org.apache.spark.sql.types.StructField] =

Array(StructField(name,StringType,true), StructField(age,StringType,true))

//将fields强制转换为StructType对象，形成了真正可用于构建DataFrame对象的Schema

scala> val schema = StructType(fields)

schema: org.apache.spark.sql.types.StructType =

StructType(StructField(name,StringType,true), StructField(age,StringType,true))

//将peopleRDD（RDD[String]）转化为RDD[Rows]

scala> val rowRDD = peopleRDD.map(\_.split(",")).map(attributes => Row(attributes(0),attributes(1).trim))

rowRDD: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row] = MapPartitionsRDD[7] at map at <console>:33

//将schema应用到rowRDD上，完成DataFrame的转换

scala> val peopleDF = sparkSession.createDataFrame(rowRDD,schema)

peopleDF: org.apache.spark.sql.DataFrame = [name: string, age: string]

//将peopleDF注册成临时表“people”

scala> peopleDF.createOrReplaceTempView("people")

//调用sql接口，运行SQL表达式，进行SQL查询

scala> val results = sparkSession.sql("SELECT name FROM people")

results: org.apache.spark.sql.DataFrame = [name: string]

//sql()返回值依然是DataFrame对象，map遍历Row对象时依然可以通过下表或者字段名对行里面的特定列进行访问

scala> results.map(attributes => "Name: "+attributes(0)).show()

+--------------------+

| value|

+--------------------+

|Name: Michael|

| Name: cindy|

| Name: andy|

| Name: Aaron|

| Name: Abel|

| Name: Abner|

| Name: Adam|

| Name: Addim|

| Name: Bob|

| Name: Bboy|

| Name: chad|

| Name: chayu|

| Name: Vichey|

| Name: Coco|

| Name: Mike|

| Name: Mark|

| Name: dennis|

| Name: Dennio|

| Name: Wert|

| Name: jnny|

+----------------------+

//默认显示20行

only showing top 20 rows

由此可见，将RDDS转化为DataFrame/Datasets[Rows]的实质就是，赋予RDD内部包含类型对象的结构信息，使得DataFrame掌握更丰富的结构与信息（可想象成传统数据库表的表头，表头包含各字段名称、类型等信息），如此一来，便更好的说明DataFrame为什么相较于RDDs提供更强大丰富的功能、支持SQL查询了。

## DataFrame的操作

DataFrame提供的API有实现逻辑的Transformation操作，如select、where、orderBy、groupBy等负责指定结果列、过滤、排序、分组的方法，和负责触发计算、回收结果的Action操作。接下来对 DataFrame提供的这些丰富强大的API进行研究，进而帮助读者轻松高效地组合使用DataFrame所提供的API来实现业务需求。

### Transformation操作

针对大多与RDD中相关操作功能类似（例如map、flatMap等操作）在本节不再赘述。DataFrame对象常用的Transformation操作如下表所示。

表7-1 DataFrame常见的Transformation操作

|  |  |
| --- | --- |
| 基础函数 | 说明 |
| agg(expr:[Column](http://spark.apache.org/docs/latest/api/scala/org/apache/spark/sql/Column.html),exprs: [Column](http://spark.apache.org/docs/latest/api/scala/org/apache/spark/sql/Column.html)\*): [DataFrame](http://spark.apache.org/docs/latest/api/scala/org/apache/spark/sql/package.html" \l "DataFrame=org.apache.spark.sql.Dataset[org.apache.spark.sql.Row]) | 在整体DataFrame不分组聚合 |
| apply(colName: String): Column | 基于列名选择列，并以一个Column的形式返回 |
| col(colName: String): Column | 基于列名选择列，并以一个Column的形式返回 |
| select(col: String, cols: String\*): DataFrame | 获取指定字段值 |
| distinct(): DataFrame | 返回一个新的DataFrame，仅包含DataFrame的unique rows |
| drop(col: Column): DataFrame | drop一个列，并返回一个新的DataFrame |
| except(other: DataFrame): DataFrame | 返回一个新的DataFrame，在当前的dataFrame但是不在另外一个DataFrame |
| filter(conditionExpr: String): DataFrame | 使用给定的SQL表达式过滤 |
| groupBy(col1: String, cols: String\*): RelationalGroupedDataset | 使用给定的列分组，以便能够进行聚合操作 |
| intersect(other: DataFrame): DataFrame | 返回当前DataFrame与另外的DataFrame的交集DataFrame |
| limit(n: Int): DataFrame | 获取前面几行数据，返回一个新的DataFrame |
| orderBy(sortExprs: Column\*): DataFrame | 使用给定的表达式进行排序，返回一个新的DataFrame |
| sort(sortExprs: Column\*):DataFrame | 返回一个给定表达式排序的新的DataFrame |
| sample(withReplacement: Boolean, fraction: Double):DataFrame | 使用一个随机种子，抽样一部分行返回一个新的DataFrame |
| where() | 按照指定条件对数据进行过滤筛选并返回新的DataFrame |
| join() | 对两个DataFrame做关联操作 |
| na: DataFrameNaFunctions | 对具有空值列的行数据进行处理 |

1. **agg(expr:Column,exprs: Column\*): DataFrame**

agg是一种聚合操作，该方法输入的是对于聚合操作的表达，可同时对多个列进行聚合操作，agg为DataFrame提供数据列不需要经过分组就可以执行统计操作。也可以与groupBy方法配合使用。

scala> df.agg("age" -> "mean","phone" -> "min").show()

+---------------------------+-------------------+

| avg(age)| min(phone)|

+---------------------------+-------------------+

|21.23913043478261|13958695789|

+---------------------------+------------------+

1. **apply(colName: String): Column**

该方法用来指定列名返回DataFrame的列。下列两种获取column的方法等效，返回的皆为对应的Column。

scala> df.apply("name")

res2: org.apache.spark.sql.Column = name

scala> df("phone")

res3: org.apache.spark.sql.Column = phone

1. **col(colName: String): Column**

该方法用来获取指定字段，apply()和col()参数类型、个数以及返回值类型均相同，只能获取某一列，返回对象为Column类型。

scala> df.col("name")

res4: org.apache.spark.sql.Column = name

1. **select(col: String, cols: String\*): DataFrame**

该方法用来获取指定字段值，根据传入的String类型的字段名，获取指定字段的值，以DataFrame类型返回。还有一个重载的select方法，传入的参数是Column类型的，可以用来实现select id,id+1 from table这种逻辑。

scala> df.select("name","sex").show(10)

+------------+---------+

| name| sex|

+------------+---------+

| Michael| male|

| cindy| male|

| andy|female|

| Aaron| male|

| Abel| male|

| Abner| male|

| Adam|female|

| Addim|female|

| Bob| male|

| Bboy| male|

+-----------+----------+

only showing top 10 rows

scala> df.select(df("name"),df("sex"),df("age")-1).show(10)

+-----------+---------+-----------+

| name| sex|(age - 1)|

+-----------+---------+-----------+

|Michael| male| 19.0|

| cindy| male| 20.0|

| andy|female| 24.0|

| Aaron| male| 19.0|

| Abel| male| 23.0|

| Abner| male| 20.0|

| Adam|female| 19.0|

| Addim|female| 19.0|

| Bob| male| 19.0|

| Bboy| male| 19.0|

+----------+---------+-----------+

only showing top 10 rows

1. **distinct(): DataFrame**

该方法用来返回对DataFrame的数据记录去重后的DataFrame。实例中，选择了有重复记录的“age”列，最后调用distinct方法进行去重。

scala> df.select("age").show(5)

+---+

|age|

+---+

| 20|

| 21|

| 25|

| 20|

| 24|

+---+

only showing top 5 rows

scala> df.select("age").distinct.show()

+---+

|age|

+---+

| 25|

| 24|

| 20|

| 21|

+---+

1. **drop(col: Column): DataFrame**

该方法用来去除指定字段，保留其他字段，返回一个新的DataFrame，其中不包含去除的字段，一次只能去除一个字段。drop方法有两种重载函数：df.drop(“id”)和df.drop(df(“id”))，前者的输入参数是描述列名称的String，而后者传入的是Column类型的列。

scala> df.show(5,false)

+---+--------------------------------------------------------------+----------+-------------------+---------+

|age|institute |name |phone |sex |

+---+--------------------------------------------------------------+----------+-------------------+---------+

|20 |computer science and technology department|Michael|16695845876|male |

|21 |medical college |cindy |18755698695|male |

|25 |chemical engineering institude |andy |15585749562|female|

|20 |SEM |Aaron |16695845876|male |

|24 |school of materials |Abel |18755698695|male |

+---+---------------------------------------------------------------+-----------+------------------+---------+

only showing top 5 rows

// scala> df.drop(df("institute")).show(5)的结果与以下结果一样

scala> df.drop("institute").show(5)

+-----+----------+-----------------+-----------+

|age| name| phone| sex|

+-----+----------+-----------------+----------+

| 20|Michael|16695845876| male|

| 21| cindy|18755698695| male|

| 25| andy|15585749562|female|

| 20| Aaron|16695845876| male|

| 24| Abel|18755698695| male|

+-----+----------+-----------------+----------+

only showing top 5 rows

1. **except(other: DataFrame): DataFrame**

返回DataFrame，包含当前DataFrame的数据记录，同时这些Rows不在另一个DataFrame中，相当于两个DataFrame做减法。

scala> val newdf = sparkSession.read.json("/home/ubuntu01/newstudent.json")

newdf: org.apache.spark.sql.DataFrame = [age: string, institute: string ... 3 more fields]

scala> newdf.show(false)

+---+-------------------------------------------------------------+-----------+------------------+----+

|age|institute |name |phone |sex |

+---+-------------------------------------------------------------+-----------+-------------------+----+

|20 |computer science and technology department|Michael|16695845876|male|

+---+--------------------------------------------------------------+-----------+------------------+-----+

scala> df.show(false)

+---+-------------------------------------------------------------+-----------+------------------+------+

|age|institute |name |phone |sex |

+---+-------------------------------------------------------------+-----------+------------------+-------+

|20 |computer science and technology department|Michael|16695845876|male |

|21 |medical college |cindy |18755698695|male |

|25 |chemical engineering institude |andy |15585749562|female|

|20 |SEM |Aaron |16695845876|male |

|24 |school of materials |Abel |18755698695|male |

|21 |school of materials |Abner |16695845876|male |

|20 |medical college |Adam |18755698695|female|

|20 |computer science and technology department|Addim |16695845876|female|

|20 |chemical engineering institude |Bob |18755698695|male |

|20 |SEM |Bboy |16695845876|male |

+---+-------------------------------------------------------------+-----------+------------------+-------+

scala> df.except(newdf).show(false)

+---+-------------------------------------------------------------+-----------+------------------+-------+

|age|institute |name |phone |sex |

+---+-------------------------------------------------------------+-----------+------------------+-------+

|20 |SEM |Bboy |16695845876|male |

|21 |medical college |cindy|18755698695|male |

|25 |chemical engineering institude |andy |15585749562|female|

|20 |SEM |Aaron|16695845876|male |

|24 |school of materials |Abel |18755698695|male |

|20 |chemical engineering institude |Bob |18755698695|male |

|20 |medical college |Adam |18755698695|female|

|20 |computer science and technology department|Addim|16695845876|female|

|21 |school of materials |Abner|16695845876|male |

+---+-------------------------------------------------------------+-----------+------------------+-------+

1. **filter(conditionExpr: String): DataFrame**

按参数指定的SQL表达式的条件过滤DataFrame。

scala> df.filter("age >24 ").show(false)

+-----+----------------------------------------+-------+---------------+---------+

|age|institute |name|phone |sex |

+---+------------------------------------------+-------+---------------+---------+

|25 |chemical engineering institude|andy|15585749562|female|

+---+------------------------------------------+-------+---------------+---------+

1. **groupBy(col1: String, cols: String\*): RelationalGroupedDataset**

使用一个或者多个指定的列对DataFrame进行分组，以便对它们执行聚合操作。示例中搭配agg()使用，先根据sex列对df进行分组，分组后求年龄的平均值和最小电话号码。

scala> df.groupBy("sex").agg("age" -> "mean","phone" -> "min").show()

+---------+-----------------------------+----------------+

| sex| avg(age)| min(phone)|

+---------+-----------------------------+-----------------+

|female|21.666666666666668|15585749562|

| male|20.857142857142858|16695845876|

+---------+-----------------------------+-------------------+

groupBy()方法得到的是RelationalGroupedDataset对象，在RelationalGroupedDataset的API中提供了groupBy()之后的操作，比如：

max(colNames:String\*)方法，获取分组中指定字段或者所有的数字类型字段的最大值，只能作用于数字型字段。

min(colNames:String\*)方法，获取分组中指定字段或者所有的数字类型字段的最小值，只能作用于数字型字段。

mean(colNames:String\*)方法，获取分组中指定字段或者所有的数字类型字段的平均值，只能作用于数字型字段。

sum(colNames:String\*)方法，获取分组中指定字段或者所有的数字类型字段的和值，只能作用于数字型字段。

Count()方法，获取分组中的元素个数。

scala> df.groupBy("age").count().show()

+-----+-------+

|age|count|

+-----+-------+

| 25| 1|

| 24| 1|

| 20| 6|

| 21| 2|

+-----+-------+

1. **intersect(other: DataFrame): DataFrame**

取两个DataFrame中同时存在的数据记录，返回DataFrame。

scala> df.intersect(newdf).show(false)

+-----+-----------------------------------------------------------+-----------+------------------+-----+

|age|institute |name |phone |sex |

+-----+-----------------------------------------------------------+-----------+------------------+-----+

|20 |computer science and technology department|Michael|16695845876|male|

+-----+-----------------------------------------------------------+-----------+------------------+-----+

1. **limit(n: Int): DataFrame**

limit()方法获取指定DataFrame的前n行记录，得到一个新的DataFrame对象。和take与head不同的是，limit方法不是Action操作，因为take/head获得的均为Array（数组），而limit返回的是一个新的转化生成的DataFrame对象。

scala> df.limit(3).show(false)

+-----+-----------------------------------------------------------+------------+-------------------+--------+

|age|institute |name |phone |sex |

+-----+-----------------------------------------------------------+------------+-------------------+---------+

|20 |computer science and technology department|Michael|16695845876|male |

|21 |medical college |cindy |18755698695|male |

|25 |chemical engineering institude |andy |15585749562|female|

+-----+------------------------------------------------------------+-----------+-------------------+---------+

1. **orderBy(sortExprs: Column\*): DataFrame**

按照给定的表达式对指定的一列或者多列进行排序，返回一个新的DataFrame，输入参数为多个Column类。

scala> df.orderBy("age","phone").show(false)

+-----+-----------------------------------------------------------+-------------+------------------+---------+

|age|institute |name |phone |sex |

+-----+------------------------------------------------------------+-------------+-----------------+---------+

|20 |computer science and technology department|Addim |16695845876|female|

|20 |SEM |Bboy |16695845876|male |

|20 |computer science and technology department|Michael|16695845876|male |

|20 |SEM |Aaron |16695845876|male |

|20 |medical college |Adam |18755698695|female|

|20 |chemical engineering institude |Bob |18755698695|male |

|21 |school of materials |Abner |16695845876|male |

|21 |medical college |cindy |18755698695|male |

|24 |school of materials |Abel |18755698695|male |

|25 |chemical engineering institude |andy |15585749562|female|

+-----+------------------------------------------------------------+------------+------------------+---------+

1. **sort(sortExprs: Column\*):DataFrame**

按照给定的表达式对指定的一列或者多列进行排序，返回一个新的DataFrame，输入参数为多个Column类。按指定字段排序，默认为升序，在Column后面加.desc表示降序排序，加.asc表示升序排序，sort()和orderBy()方法效果等效。

所有列升序排序，以下三种写法等价：

df.sort(“sortcol”)

df.sort($”sortcol”)

df.sort($”sortcol”.asc)

例子如下：

scala> df.sort($"age".desc).show(false)

+-----+-----------------------------------------------------------+------------+------------------+---------+

|age|institute |name |phone |sex |

+----+------------------------------------------------------------+-----------+-------------------+----------+

|25 |chemical engineering institude |andy |15585749562|female|

|24 |school of materials |Abel |18755698695|male |

|21 |medical college |cindy |18755698695|male |

|21 |school of materials |Abner |16695845876|male |

|20 |computer science and technology department|Michael|16695845876|male |

|20 |computer science and technology department|Addim |16695845876|female|

|20 |SEM |Aaron |16695845876|male |

|20 |chemical engineering institude |Bob |18755698695|male |

|20 |SEM |Bboy |16695845876|male |

|20 |medical college |Adam |18755698695|female|

+-----+------------------------------------------------------------+------------+-------------------+--------+

1. **sample(withReplacement: Boolean, fraction: Double):DataFrame**

sample对数据集进行采样，返回一个新的DataFrame。withReplacement=true，表示重复抽样；withReplacement=false，表示不重复抽样；fraction参数是生成行的比例。

scala> df.sample(true,0.5).show()

+-----+--------------------+--------+---------------+-------+

|age| institute| name| phone| sex|

+-----+--------------------+--------+---------------+-------+

| 21|medical college|cindy|18755698695|male|

| 21|medical college|cindy|18755698695|male|

| 21|medical college|cindy|18755698695|male|

| 20| SEM| Bboy|16695845876|male|

+-----+--------------------+--------+---------------+-------+

scala> df.sample(false,0.5).show(false)

+-----+-------------------------------------------------------------+--------+-----------------+--------+

|age|institute |name |phone |sex |

+-----+-------------------------------------------------------------+--------+-----------------+---------+

|25 |chemical engineering institude |andy |15585749562|female|

|20 |SEM |Aaron|16695845876|male |

|20 |computer science and technology department|Addim|16695845876|female|

|20 |SEM |Bboy |16695845876|male |

+-----+-------------------------------------------------------------+-------+-------------------+---------+

1. **where()**

where方法根据参数类型以及数目不同进行了同名函数重载如下：

where(conditionExpr:String):DataFrame

where(condition:Column):DataFrame

其中where(conditionExpr:String)输入更像一种传统的SQL的where子句的条件整体描述，传入筛选条件表达式，可以用and和or，得到DataFrame类型的返回结果；而where(condition:Column)该方法的输入则是要把where子句的对于每一个column的要求进行分别描述，使用$”列名”提取列数据做比较时，用到了隐式转换，故需在程序中引入相应包（import spark.implicits.\_）。具体实现如下：

scala> df.where("name = 'Bboy'").show()

+-----+----------+--------+------------------+-----+

|age|institute|name| phone| sex|

+-----+----------+--------+------------------+-----+

| 20| SEM|Bboy|16695845876|male|

+-----+----------+--------+------------------+-----+

scala> df.where($"age">24).show(false)

+----+---------------------------------------+--------+------------------+---------+

|age|institute |name|phone |sex |

+----+---------------------------------------+--------+------------------+---------+

|25 |chemical engineering institude|andy|15585749562|female|

+----+----------------------------------------+-------+------------------+---------+

1. **join()**

对两个DataFrame执行join操作，join根据传入的参数不同有多种实现方式（各种方式的具体定义可以参考官网）。第一个参数是进行关联操作的另一个DataFrame，还可以传入其他参数，如usingColumn：String，usingColumn:Seq[String]参数：单个字段或者多个字段名进行join操作，或者通过传入joinExprs参数：列名或多个列名组成的序列的指定连接条件的方式，还可以传入参数joinType指定具体的join操作，例如：inner、leftouter、rightouter等类型。

scala> joindf.show()

+-----+--------+--------+

|age|height| name|

+-----+--------+--------+

| 21| 180|cindy|

| 22| 175|Aaron|

| 24| 165| kk|

+----+---------+--------+

scala> df.join(joindf,"age").show()

+-----+-----------------------+---------+----------------+-------+--------+--------+

|age| institute| name| phone| sex|height|name|

+-----+-----------------------+---------+----------------+-------+--------+--------+

| 21| medical college|cindy|18755698695|male| 180|cindy|

| 24|school of materials| Abel|18755698695|male| 165| kk|

| 21|school of materials|Abner|16695845876|male| 180|cindy|

+----+-------------------------+--------+-----------------+-------+--------+-------+

scala> df.join(joindf,Seq("name","age")).show()

+-------+-----+-------------------+------------------+-----+--------+

|name|age| institute| phone| sex|height|

+-------+-----+-------------------+------------------+-----+--------+

|cindy| 21|medical college|18755698695|male| 180|

+-------+-----+-------------------+------------------+------+-------+

scala> df.join(joindf,df("name")===joindf("name")).show()

+-----+--------------------+--------+-----------------+-----+----+--------+---------+

|age| institute| name| phone| sex|age|height| name|

+-----+--------------------+--------+-----------------+-----+----+--------+---------+

| 21|medical college|cindy|18755698695|male| 21| 180| cindy|

| 20| SEM|Aaron|16695845876|male| 22| 175|Aaron|

+-----+--------------------+--------+-----------------+-----+----+--------+---------+

1. **na: DataFrameNaFunctions**

使用na方法对具有空值列的行数据进行处理，例如缺失某一列值的行或用指定值（缺失值）替换控制列的值，需要注意的是，在DataFrame对象上使用na方法后返回的是对应的DataFrameNaFunction对象，进而需要调用对应的drop、fill方法来处理指定列为空值的行。drop用来删除指定列为空值的行，fill使用指定的值替换指定空值列的值。具体实现如下：

scala> df1.show(3)

+-----+---------------------------+----------+-------------------+-----+

|age| institute| name| phone| sex|

+-----+---------------------------+----------+-------------------+-----+

|null| null| null| null| null|

|null| medical college| cindy|18755698695|male|

| 20| computer science ...|Michael|16695845876|male|

+-----+---------------------------+-----------+-------------------+------+

only showing top 3 rows

scala> df1.na.drop().show(3)

+-----+-------------------------+-----------+------------------+---------+

|age| institute| name| phone| sex|

+-----+-------------------------+-----------+------------------+---------+

| 20|computer science ...|Michael|16695845876| male|

| 21| medical college| cindy|18755698695| male|

| 25|chemical engineer...| andy|15585749562|female|

+-----+-------------------------+-----------+------------------+---------+

only showing top 3 rows

scala> df1.na.fill(Map(("age",28),("institute","jsj"))).show(3)

+-----+---------------------------+---------+-----------------+-----+

|age| institute| name| phone| sex|

+-----+---------------------------+---------+-----------------+-----+

| 28| jsj| null| null| null|

| 28| medical college| cindy|18755698695|male|

| 20|computer science ...|Michael|16695845876|male|

+-----+---------------------------+---------+------------------+------+

only showing top 3 rows

### Action操作

以下主要学习在DataFrame上触发真正计算，返回结果的行动操作。

表7-2 Action操作

|  |  |
| --- | --- |
| Action操作 | 说明 |
| collect(): Array[Row] | 以Array形式返回DataFrame的所有的Rows |
| collectAsList(): List[Row] | 类似collect()，返回结构变成了List对象 |
| count(): Long | 返回DataFrame的Rows数目 |
| describe(cols: String\*): DataFrame | 获取指定字段的统计信息 |
| first():Row | 返回第一行数据 |
| head():Row | 返回第一行数据 |
| show(): Unit | 以表格形式显示DataFrame的前n行数据，默认为20行 |
| take(n: Int): Array[Row] | 返回DataFrame前面N行数据 |

1. **collect(): Array[Row]**

该方法返回一个数组，包含DataFrame中包含的全部数据记录。

scala> df.collect()

res105: Array[org.apache.spark.sql.Row] = Array([22,medical college,cindy,18755698695,male], [20,computer science and technology department,Michael,16695845876,male], [21,medical college,cindy,18755698695,male], [25,chemical engineering institude,andy,15585749562,female], [20,SEM,Aaron,16695845876,male], [24,school of materials,Abel,18755698695,male], [21,school of materials,Abner,16695845876,male], [20,medical college,Adam,18755698695,female], [20,computer science and technology department,Addim,16695845876,female], [20,chemical engineering institude,Bob,18755698695,male], [20,SEM,Bboy,16695845876,male])

1. **collectAsList(): List[Row]**

该方法返回一个List，包含DataFrame中包含的全部数据记录。

scala> df.collectAsList()

res106: java.util.List[org.apache.spark.sql.Row] = [[22,medical college,cindy,18755698695,male], [20,computer science and technology department,Michael,16695845876,male], [21,medical college,cindy,18755698695,male], [25,chemical engineering institude,andy,15585749562,female], [20,SEM,Aaron,16695845876,male], [24,school of materials,Abel,18755698695,male], [21,school of materials,Abner,16695845876,male], [20,medical college,Adam,18755698695,female], [20,computer science and technology department,Addim,16695845876,female], [20,chemical engineering institude,Bob,18755698695,male], [20,SEM,Bboy,16695845876,male]]

1. **count(): Long**

返回DataFrame的数据记录的条数。

scala> df.count()

res114: Long = 11

1. **describe(cols: String\*): DataFrame**

该方法可以动态的传入一个或多个String类型的字段名，结果仍然是DataFrame对象，用来统计数值类型字段的统计值。在DataFrame下只需调用describe()子函数，便可以得到以下信息：Count（记录条数）、Mean（平均值）、Stddev（样本标准差）、Min（最小值）、Max（最大值），进而掌握大规模结构化数据集的某字段的统计特性。

scala> df.describe("age").show()

+------------+-----------------------------+

|summary| age|

+------------+-----------------------------+

| count| 11|

| mean|21.181818181818183|

| stddev| 1.778661396566632|

| min| 20|

| max| 25|

+------------+----------------------------+

1. **first():Row**

返回DataFrame的第一行，等同于head()方法。

scala> df.first()

res108: org.apache.spark.sql.Row = [22,medical college,cindy,18755698695,male]

1. **head():Row**

不带参数的head方法，返回DataFrame的第一条数据记录，指定参数n时，则返回前n条数据记录。

scala> df.head(2)

res109: Array[org.apache.spark.sql.Row] = Array([22,medical college,cindy,18755698695,male], [20,computer science and technology department,Michael,16695845876,male])

1. **show(): Unit**

不带参数时，用表格对的形式显示DataFrame的前20行记录；指定参数numRows时，用表格的形式显示DataFrame指定的行数记录；指定参数truncate默认为true，表示最多显示20个字符，若设为false，则将过长字符串全部显示。

scala> df.show(2,false)

+-----+------------------------------------------------------------+----------+------------------+-----+

|age|institute |name |phone |sex |

+-----+------------------------------------------------------------+----------+------------------+-----+

|22 |medical college |cindy |18755698695|male|

|20 |computer science and technology department|Michael|16695845876|male|

+-----+------------------------------------------------------------+-----------+------------------+-----+

only showing top 2 rows

scala> df.show(2,true)

+-----+-------------------------+-----------+------------------+-----+

|age| institute| name| phone| sex|

+-----+-------------------------+-----------+------------------+-----+

| 22| medical college| cindy|18755698695|male|

| 20|computer science ...|Michael|16695845876|male|

+-----+-------------------------+-----------+------------------+------+

only showing top 2 rows

1. **take(n: Int): Array[Row]**

类似head方法，返回DataFrame中指定的前n行的值，返回的是Array[Row]。对于takeAsList方法，也是获取前n行记录，只不过以List的形式展现。这两种方法都是将获得的数据返回到Driver端，所以在使用这两个方法的时候需要注意数据量，以免Driver发生OutOfMemoryError。

scala> df.take(2)

res112: Array[org.apache.spark.sql.Row] = Array([22,medical college,cindy,18755698695,male], [20,computer science and technology department,Michael,16695845876,male])

## 数据源

Spark SQL支持通过DataFrame接口操作多种不同的数据源。DataFrame可以使用关系转换操作，也可用于创建临时表，将DataFrame注册为临时表进而对数据运行SQL查询。DataFrame提供统一接口加载和保存数据源中的数据，包括：结构化数据、Parquet文件、JSON文件、Hive表，以及通过JDBC连接外部数据源。接下来介绍使用Spark数据源加载和保存数据的一般方法以及Spark SQL可处理的各种数据源，使读者了解Spark SQL在大数据领域对典型结构化数据源的皆可处理性，从而使读者掌握一门结构化数据的分析利器。

### 加载保存操作

Spark SQL的默认数据源格式为Parquet格式，Parquet是面向分析型业务的列式存储格式。数据源为Parquet格式文件时，Spark SQL可以方便地进行读取，甚至可以直接在Parquet文件上执行查询操作。修改配置项spark.sql.sources.default，可以修改默认数据源格式。以下示例通过通用的load/save方法对Parquet文件进行读取、存储。

//读取parquet格式数据

val usersDF = spark.read.load("examples/src/main/resources/users.parquet")

//写数据并保存成Parquet格式

usersDF.select("name", "favorite\_color").write.save("namesAndFavColors.parquet")

1. **指定选项**

当数据源不是Parquet格式文件时，需要手动指定数据源的格式。数据源格式需要指定全名（如org.apache.spark.sql.parquet），如果数据源为内置格式，则只需指定简称（json、parquet、jdbc、orc、libsvm、csv、text）即可，通过指定数据源格式名，还可以通过format对任何类型的DataFrame进行类型转换操作。如下示例是将原有的JSON格式的数据源转储为Parquet格式文件。

scala> val peopleDf = sparkSession.read.format("json").load("/home/ubuntu01/people.json")

peopleDf: org.apache.spark.sql.DataFrame = [age: string, institute: string ... 3 more fields]

scala> peopleDf.select("name","age").write.format("parquet").save("nameAndAges.parquet")

1. **在文件上直接进行SQL查询**

相比于使用read API将文件加载到DataFrame并对其进行查询，还可以使用SQL直接查询该文件。

scala> val sqlDF = spark.sql("SELECT \* FROM parquet.`/home/ubuntu01/nameAndAges.parquet`")

sqlDF: org.apache.spark.sql.DataFrame = [name: string, age: string]

1. **存储模式**

保存操作可以选择使用存储模式（SaveMode），从而指定如何处理现有数据（如果存在），例如将数据追加到文件或者是覆盖文件内容。需要注意的是，要意识到这些保存模式不会使用任何锁定，也不是原子的。另外，当执行覆盖的时候，在写入新数据之前，数据将被删除。

表7-3 存储模式

|  |  |  |
| --- | --- | --- |
| Scala/Java | Any Language | 含义 |
| SaveMode.ErrorIfExists (default) | "error" (default) | 将 DataFrame 保存到 data source （数据源）时, 如果数据已经存在, 则会抛出异常. |
| SaveMode.Append | "append" | 将 DataFrame 保存到 data source （数据源）时, 如果 data/table 已存在, 则 DataFrame 的内容将被 append （附加）到现有数据中. |
| SaveMode.Overwrite | "overwrite" | Overwrite mode （覆盖模式）意味着将 DataFrame 保存到 data source （数据源）时, 如果 data/table 已经存在, 则预期 DataFrame 的内容将 overwritten （覆盖）现有数据. |
| SaveMode.Ignore | "ignore" | Ignore mode （忽略模式）意味着当将 DataFrame 保存到 data source （数据源）时, 如果数据已经存在, 则保存操作预期不会保存 DataFrame 的内容, 并且不更改现有数据. 这与 SQL 中的 CREATE TABLE IF NOT EXISTS 类似. |

1. **持久到表**

DataFrames也可以使用saveAsTable命令作为持久表保存到Hive metastore中，请注意，即使用户并没有在集群中部署现有的Hive数据仓库以供持久表的存储，也可以使用该功能，因为 Spark 将创建默认的本地 Hive metastore使用Derby。与createOrReplaceTempView 方法不同，saveAsTable将实现DataFrame的内容，并创建一个指向Hive metastore中指定持久表的数据的指针。只要保持与同一个metastore的连接，即使Spark程序重新启动，持久性表仍然存在，可以通过使用表的名称在SparkSession上调用table(“table\_name”)方法来创建持久表的DataFrame对象。

对于基于文件的数据源，例如text，parquet，json等，可以通过路径选项指定自定义表存储路径，例如df.write.option("path", "/some/path").saveAsTable("t")。当表被删除时，自定义表路径将不会被删除，并且表数据仍然存在。如果未指定自定义表路径，Spark将把数据写入仓库目录下的默认表路径。当表被删除时，默认的表路径也将被删除。

从Spark 2.1开始，持久性数据源表将每个分区元数据存储在Hive metastore中。这带来了几个好处：

◆由于metastore只能返回查询的必要分区数据，因此不再需要为每一个查询都遍历查询表的所有数据。

◆涉及到表的分区Hive DDL语句，如ALTER TABLE PARTITION ... SET LOCATION现在可用于使用Datasource API创建的表。

请注意，创建外部数据源表（带有路径选项的表）时，默认情况下不会收集分区信息。要同步转移中的分区信息，可以调用MSCK重新建立表的分区信息。

**分桶, 排序和分区**

对于基于文件的数据源，也可以对输出进行bucket（bucket（桶）操作是指将表或分区中指定列的值为key进行hash，hash到指定的桶中，这样可以支持高效采样工作，提升某些查询操作效率）、sort和partition操作，分桶和排序仅适用于持久化表：

peopleDF.write.bucketBy(42, "name").sortBy("age").saveAsTable("people\_bucketed")

将DataFrame对象存储为按照favorite\_color列值分区的parquet文件：

usersDF.write.partitionBy("favorite\_color").format("parquet").save("namesPartByColor.parquet")

对持久化表组合进行分区操作、桶操作：

usersDF

.write

.partitionBy("favorite\_color")

.bucketBy(42, "name")

.saveAsTable("users\_partitioned\_bucketed")

### 典型结构化数据源

1. **Parquet文件**

Parquet是一种流行的列式存储格式。Spark SQL提供对Parquet文件的读写支持，而且Parquet文件能够自动保存原始数据的schema。写Parquet文件的时候，所有的字段都会自动转成nullable，以便向后兼容。

1. 以编程方式加载数据

// 常见类的编码器可以通过导入spark.implicits.\_自动提供

import spark.implicits.\_

val peopleDF = spark.read.json("examples/src/main/resources/people.json")

// peopleDF保存为parquet文件时，依然会保留着结构信息

peopleDF.write.parquet("people.parquet")

// 读取上述创建的parquet文件，Parquet文件是自描述的，所以结构信息被保留

//读取一个Parquet文件的结果是一个已经具有完整结构信息的DataFrame对象

val parquetFileDF = spark.read.parquet("people.parquet")

// Parquet 文件也可以用来创建一个临时视图，然后在SQL语句中使用

parquetFileDF.createOrReplaceTempView("parquetFile")

val namesDF = spark.sql("SELECT name FROM parquetFile WHERE age BETWEEN 13 AND 19")

namesDF.map(attributes => "Name: " + attributes(0)).show()

// +-----------------+

// | value|

// +-----------------+

// |Name: Justin|

// +-----------------+

1. 分区发现

表分区是在像Hive这样的系统中使用的常见的优化方法。在分区表中，数据通常存储在不同的目录中，根据分区列的值不同，编码了在每个分区目录不同的路径。目前Parquet 数据源现在可以自动发现和推断分区信息。例如，可以使用以下目录结构存储所有以前使用的数据到分区表中，只需要额外地添加两个列gender和country作为分区列：

path

└── to

└── table

├── gender=male

│ ├── ...

│ │

│ ├── country=US

│ │ └── data.parquet

│ ├── country=CN

│ │ └── data.parquet

│ └── ...

└── gender=female

├── ...

│

├── country=US

│ └── data.parquet

├── country=CN

│ └── data.parquet

└── ...

在这个例子中，如果需要读取Parquet文件数据，只需要把path/to/table 作为参数传递给 SparkSession.read.parquet或SparkSession.read.load，Spark SQL将自动从路径中提取出分区信息，并识别数据表的结构信息来创建DataFrame对象。在返回的DataFrame对象上调用printSchema()方法，可看到结构信息如下：

root

|-- name: string (nullable = true)

|-- age: long (nullable = true)

|-- gender: string (nullable = true)

|-- country: string (nullable = true)

请注意，分区列的数据类型是自动推断的。目前，支持数值型数据和字符串型数据。有些用户可能不想自动推断分区列的数据类型。对于这些情况，自动类型推断可以由 spark.sql.sources.partitionColumnTypeInference.enabled参数配置，默认为true。当禁用类型推断时，分区列的类型将为字符串型。

从Spark 1.6.0开始，默认情况下，分区发现只能找到给定路径下的分区。对于上述示例，如果用户将path/to/table/gender=male传递给SparkSession.read.parquet或SparkSession.read.load ，则gender将不被视为分区列。如果用户需要指定分区发现应该开始的基本路径，则可以在数据源选项中设置basePath。例如，当path/to/table/gender=male是数据的路径，并且用户将basePath设置为path/to/table/时，gender将是一个分区列。

1. 模式合并

像ProtocolBuffer，Avro和Thrift一样，Parquet也支持模式演进。用户可以从一个简单的Schema开始，并根据需要逐渐向schema添加更多的列。以这种方式，用户可能会使用不同但相互兼容的schema多个Parquet文件。Parquet数据源现在能够自动检测这种情况并合并所有这些文件的模式。

由于模式合并是一个相对昂贵的操作，并且在大多数情况下不是必需的，所以从1.5.0开始，我们默认将它关闭，可以通过以下方法使它生效：

* 1. 读取Parquet文件时，将数据源选项mergeSchema设置为true（如下面的例子所示）。
  2. 将全局SQL选项spark.sql.parquet.mergeSchema设置为true。

//引用spark.implicits.\_用于将RDD隐式转换为DataFrame

import spark.implicits.\_

// 创建一个简单的DataFrame，包含value、square两列，并将其存储到一个分区目录，该分区目录表示额外的分区列为key，对应的值为1

val squaresDF = spark.sparkContext.makeRDD(1 to 5).map(i => (i, i \* i)).toDF("value", "square")

squaresDF.write.parquet("data/test\_table/key=1")

// 创建一个新的DataFrame，包含value、cube两列，并将其存储到相同表下的新的分区目录（data/test\_table/key=2），表示额外的分区列为key，对应的值为2

// 增加了一个cube列，去掉了一个已存在的square列

val cubesDF = spark.sparkContext.makeRDD(6 to 10).map(i => (i, i \* i \* i)).toDF("value", "cube")

cubesDF.write.parquet("data/test\_table/key=2")

//读取完整的分区表，自动实现了两个分区（key=1/2）的合并

val mergedDF = spark.read.option("mergeSchema", "true").parquet("data/test\_table")

mergedDF.printSchema()

//最终的Schema不仅包含两个Parquet分区文件出现的所有三列，还包含了作为分区目录的额外分区列key

// root

// |-- value: int (nullable = true)

// |-- square: int (nullable = true)

// |-- cube: int (nullable = true)

// |-- key: int (nullable = true)

1. Hive metastore Parquet table转换

当读取和写入Hive metastore Parquet表时，Spark SQL将使用自带的Parquet支持库，而不是Hive SerDe来获得更好的性能。这一行为由spark.sql.hive.convertMetastoreParquet配置控制，而且默认是启用的。

**Hive表与Parquet文件的Schema转化兼容**

从表的schema处理的角度来说，Hive和Parquet之间有两个关键的区别：

* Hive是类型敏感的，而Parquet并不是。
* Hive中所有列都是非空的，而Parquet中非空是很重要的特性。

由于这个原因，当将Hive metastore Parquet表转换为Spark SQL Parquet表时，我们必须调整Hive metastore schema与Parquet schema进行一致化。一致化的规则是：

* 不管是否可为空值，在两个schema中具有相同名称的字段必须具有相同的数据类型，这种一致化后的字段应该有与Parquet一方相同的数据类型，因此可为空值的特性很重要。
* 一致化后的schema只包含Hive metastore schema中定义的字段。
* 任何只出现在Parquet schema中的字段都会在一致化后的schema中被丢弃。
* 只在Hive metastore schema中出现的字段在一致化后的schema中设为nullable字段被添加。

**元数据刷新**

Spark SQL缓存Parquet元数据以获得更好的性能。当启用Hive metastore Parquet表转换时，这些被转换表的元数据同样被缓存，如果这些表由Hive或其他外部工具更新，则需要手动刷新以确保元数据的一致性。

// spark是存在的SparkSession对象

spark.catalog.refreshTable("my\_table")

1. 配置

Parquet的配置可以使用SparkSession中的setConf方法进行，或者使用SQL执行SET key=value设置来完成。如下表所示：

表7-4 Parquet的配置

|  |  |  |
| --- | --- | --- |
| 属性名 | 默认值 | 含义 |
| spark.sql.parquet.binaryAsString | false | 有些老系统，如：特定版本的Impala、Hive，或者老版本的Spark SQL，不区分二进制数据和字符串类型数据。这个标志的意思是，让Spark SQL把二进制数据当字符串处理，以兼容老系统。 |
| spark.sql.parquet.int96AsTimestamp | true | 有些老系统，如特定版本的Impala、Hive，把时间戳存成INT96。这个配置的作用是，让Spark SQL把这些INT96解释为timestamp，以兼容老系统。 |
| spark.sql.parquet.cacheMetadata | true | 打开 Parquet 模式元数据的缓存。可以加快查询静态数据。 |
| spark.sql.parquet.compression.codec | snappy | 设置写入Parquet文件时压缩编解码器。可接受的值包括：uncompressed、snappy、gzip、lzo。 |
| spark.sql.parquet.filterPushdown | true | 设置值为true时启用Parquet过滤器下推优化。 |
| spark.sql.hive.convertMetastoreParquet | true | 当设置为false时，Spark SQL将对Parquet表使用Hive SerDe来实现序列化、反序列化，替代内置支持的SerDev。 |
| spark.sql.parquet.mergeSchema | false | 如果为true，则Parquet数据源合并从所有数据文件收集的Schema，否则如果没有摘要文件可用，则从摘要文件或随机数据文件中选取Schema。 |
| spark.sql.optimizer.metadataOnly | true | 如果为true，则启用使用表元数据的仅限元数据查询优化来生成分区列，而不是表扫描。它适用于扫描的所有列都是分区列并且查询具有满足不同语义的聚合运算符的情况。 |

1. **JSON数据集**

Spark SQL可以自动推断JSON数据集的schema，并将其作为Dataset[Row]加载。这个转换可以在Dataset[String]或JSON文件上使用SparkSession.read.json()来完成。请注意，这里的JSON文件不是典型的JSON文件。每行必须包含单独的、独立的、有效的JSON对象。对于常规的多行式JSON文件，将multiLine选项设置为true。

import spark.implicits.\_

//数据集是由路径指定的，路径既可以是单个文件，也可以是存储文本文件的目录

val path = "examples/src/main/resources/people.json"

val peopleDF = spark.read.json(path)

// 推导出来的schema，可用printSchema打印出来

peopleDF.printSchema()

// root

// |-- age: long (nullable = true)

// |-- name: string (nullable = true)

// 将DataFrame注册成table

peopleDF.createOrReplaceTempView("people")

//运行SQL语句

val teenagerNamesDF = spark.sql("SELECT name FROM people WHERE age BETWEEN 13 AND 19")

teenagerNamesDF.show()

// +--------+

// | name|

// +--------+

// |Justin|

// +--------+

// 另一种方法是，用一个包含JSON字符串的RDD来创建DataFrame

val otherPeopleDataset = spark.createDataset(

"""{"name":"Yin","address":{"city":"Columbus","state":"Ohio"}}""" :: Nil)

val otherPeople = spark.read.json(otherPeopleDataset)

otherPeople.show()

// +----------------------+-------+

// | address|name|

// +----------------------+-------+

// |[Columbus,Ohio]| Yin|

// +----------------------+-------+

1. **Hive表**

Spark SQL同样支持从Apache Hive中读写数据。但是，自从Hive有大量依赖之后，这些依赖就不包括在Spark发布版本中了。如果Hive的依赖可以在环境变量中找到，Spark将自动加载它们。注意这些Hive依赖项同样必须在每个Worker节点上存在，因为他们需要访问Hive序列化和反序列化库以便可以访问Hive中存储的数据。可以在conf/目录中的hive-site.xml、core-site.xml（安全配置）和hdfs-site.xml（HDFS配置）这几个文件中进行配置。

当在Hive上工作时，必须实例化SparkSession对Hive的支持，包括对持久化Hive云存储的连通性、对Hive序列化反序列化、Hive用户自定义函数的支持。当没有在hive-site.xml配置时，context会自动在当前目录创建metastore\_db并且创建一个被spark.sql.warehouse.dir配置的目录，默认在Spark应用启动的当前目录的spark-warehouse中配置。注意从Spark2.0.0开始，hive-site.xml中的hive.metastore.warehouse.dir参数被弃用。作为替代，使用spark.sql.warehouse.dir来指定仓库中数据库的位置。你可能需要授予写权限给启动Spark应用的用户。

import java.io.File

import org.apache.spark.sql.{Row, SaveMode, SparkSession}

case class Record(key: Int, value: String)

// warehouseLocation指向托管数据库和表的默认位置

val warehouseLocation = new File("spark-warehouse").getAbsolutePath

val spark = SparkSession

.builder()

.appName("Spark Hive Example")

.config("spark.sql.warehouse.dir", warehouseLocation)

.enableHiveSupport()

.getOrCreate()

import spark.implicits.\_

import spark.sql

//通过sql接口在Hive中创建src表，并将指定位置的原始数据存储src Hive表中

sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING) USING hive")

sql("LOAD DATA LOCAL INPATH 'examples/src/main/resources/kv1.txt' INTO TABLE src")

// 使用HiveQL进行查询

sql("SELECT \* FROM src").show()

// +----+-----------+

// |key| value|

// +----+-----------+

// |238|val\_238|

// | 86| val\_86|

// |311|val\_311|

// ...

// 包含着Hive聚合函数COUNT()的查询依然被支持

sql("SELECT COUNT(\*) FROM src").show()

// +-----------+

// |count(1)|

// +-----------+

// | 500 |

// +-----------+

// SQL查询的结果本身就是DataFrame，并支持所有正常的功能

val sqlDF = sql("SELECT key, value FROM src WHERE key < 10 ORDER BY key")

// DataFrame中的元素是Row类型的，允许按顺序访问每个列

val stringsDS = sqlDF.map {

case Row(key: Int, value: String) => s"Key: $key, Value: $value"

}

stringsDS.show()

// +-------------------------+

// | value|

// +-------------------------+

// |Key: 0, Value: val\_0|

// |Key: 0, Value: val\_0|

// |Key: 0, Value: val\_0|

// ...

// 你也可以使用DataFrame在 SparkSession中创建临时视图

val recordsDF = spark.createDataFrame((1 to 100).map(i => Record(i, s"val\_$i")))

recordsDF.createOrReplaceTempView("records")

//sql查询中可以对DataFrame注册的临时表和Hive表执行Join连接操作

sql("SELECT \* FROM records r JOIN src s ON r.key = s.key").show()

// +-----+-------+-----+-------+

// |key| value|key| value|

// +-----+-------+-----+-------+

// | 2| val\_2| 2| val\_2|

// | 4| val\_4| 4| val\_4|

// | 5| val\_5| 5| val\_5|

// ...

1. **用JDBC连接其他数据库**

Spark SQL同样包括可以使用JDBC从其他数据库读取数据的数据源。此功能优先使用jdbcRDD，这是因为返回的结果作为一个DataFrame，并且可以轻松地使用Spark SQL处理或者与其他数据源进行连接。利用Java或者Python可以更容易地使用JDBC数据源，因为它们不需要用户提供的ClassTag。（注意这与Spark SQLJDBC服务器可以允许其他应用使用Spark SQL执行查询语句不同）

在开始之前你需要将你指定的数据库的JDBC driver包含在Spark的环境变量中。例如，为了从Spark Shell连接到postgres数据库，你需要执行以下命令：

bin/spark-shell --driver-class-path postgresql-9.4.1207.jar --jars postgresql-9.4.1207.jar

远程数据库的表可以通过Data Source API，用DataFrame或者Spark SQL临时表来装载。用户可以在数据源选项中指定JDBC连接的几个必要属性。除了必要的连接属性外，Spark还支持以下不区分大小写的选项：

表7-5

|  |  |
| --- | --- |
| 属性名 | 含义 |
| url | |  |  | | --- | --- | |  | 要连接的JDBC URL。 源特定的连接属性可以在URL中指定。 例如：  jdbc:postgresql://localhost/test?user=fred&password=secret | |
| dbtable | 应该读取的 JDBC 表。请注意，可以使用在SQL查询的 FROM 子句中有效的任何内容。 例如，也可以在括号中使用子查询，而不是一个完整表。 |
| driver | 用于连接到此 URL 的 JDBC driver 程序的类名。 |
| partitionColumn, lowerBound, upperBound | |  |  | | --- | --- | |  | 如果指定了这三个选项中的任意一个，则这三个选项均需指定。另外，必须指定 numPartitions. 他们描述如何从多个 worker 并行读取数据时对表进行分区。partitionColumn 必须是相关表中的数字列。请注意，lowerBound 和 upperBound 仅用于决定分区跨度，而不是用于过滤表中的行。因此，表中的所有行将被分区并返回。此选项仅适用于读操作。 | |
| numPartitions | 在表读写中可以用于并行的最大分区数。这也决定了并发JDBC连接的最大数量。 如果要写入的分区数超过此限制，则在写入之前通过调用 coalesce(numPartitions) 将其减少到此限制。 |
| fetchsize | |  |  | | --- | --- | |  | JDBC 提取大小，决定每次数据往返取多少行。这有利于默认为低读取大小的JDBC驱动程序（例如: Oracle 是 10 行）的性能优化。。 该选项仅适用于读取操作。 | |
| batchsize | JDBC批量大小，用于确定每次往返要插入多少行。这有利于提升JDBC驱动程序的性能。这个选项只适用于写入表。它默认行为1000。 |
| isolationLevel | 事务隔离级别，适用于当前连接。它可以是NONE, READ\_COMMITTED,READ\_UNCOMMITTED,REPEATABLE\_READ, 或 SERIALIZABLE 之一，对应于 JDBC 连接对象定义的标准事务隔离级别，默认为 READ\_UNCOMMITTED。 此选项仅适用于写操作。请参考 java.sql.Connection 中的文档。 |
| truncate | |  |  | | --- | --- | |  | 这是一个与JDBC 编写器相关的选项。启用 SaveMode.Overwrite时，此选项会导致 Spark 截断现有表，而不是删除并重新创建。这可以更有效，并且防止表元数据（例如，索引）被移除。但是，在某些情况下，例如当新数据具有不同的模式时，它将不起作用。 它默认为 false。此选项仅适用于写操作。 | |
| createTableOptions | |  |  | | --- | --- | |  | 这是一个与JDBC编写器相关的选项。如果指定，此选项允许在创建表时设置数据库特定的表和分区选项（例如：CREATE TABLE t (name string) ENGINE=InnoDB）。此选项仅适用于写操作。 | |
| createTableColumnTypes | 创建表时使用数据库列数据类型而不是使用默认值。应该使用与 CREATE TABLE 列语法（例如："name CHAR(64), comments VARCHAR(1024)"）相同的格式指定数据类型信息。 指定的类型应该是有效的Spark SQL数据类型。此选项仅适用于写操作。 |

以下是读取JDBC数据源以及将DataFrame对象作为表写入其他数据库的例子：

//在SparkSession对象的read方法返回的DataFrameReader对象上通过//format(“jdbc”)方法标识读取的是JDBC数据源，并通过多个option(“key”,”value”)

//方法组合分别实现JDBC必要连接属性（url、username、password、dbtable），

//最后通过load()方法加载数据表，返回相应数据表内数据的DataFrame对象

val jdbcDF = spark.read

.format("jdbc")

.option("url", "jdbc:postgresql:dbserver")

.option("dbtable", "schema.tablename")

.option("user", "username")

.option("password", "password")

.load()

//读取JDBC数据源，除了上述采用多个option组合表示连接属性外，也可以将//url、dbtable这两个连接属性和包含除url、dbtable其他所有连接属性的Properties

//对象，直接传入jdbc方法中实现。

//实例化Properties类对象，并将添加相应的JDBC连接属性以键值对形式

val connectionProperties = new Properties()

connectionProperties.put("user", "username")

connectionProperties.put("password", "password")

val jdbcDF2 = spark.read

.jdbc("jdbc:postgresql:dbserver", "schema.tablename", connectionProperties)

connectionProperties.put("customSchema", "id DECIMAL(38, 0), name STRING")

val jdbcDF3 = spark.read

.jdbc("jdbc:postgresql:dbserver", "schema.tablename", connectionProperties)

//在DataFrame.write()方法返回的DataFrameWriter对象上通过format(“jdbc”)方法//标识JDBC，并通过多个option(“key”,”value”)方法组合分别实现JDBC必要连接属//性（url、username、password、dbtable），最后通过save()方法将DataFrame对//象以数据表的形式写入数据库

jdbcDF.write

.format("jdbc")

.option("url", "jdbc:postgresql:dbserver")

.option("dbtable", "schema.tablename")

.option("user", "username")

.option("password", "password")

.save()

//与读取JDBC数据源相同，也可以将connectionProperties对象传入write.jdbc()方法中来实现数据表的写入

jdbcDF2.write

.jdbc("jdbc:postgresql:dbserver", "schema.tablename", connectionProperties)

//写入数据表时，也可以通过option方法指定对应数据库创建该表时列的具体类型信息

jdbcDF.write

.option("createTableColumnTypes", "name CHAR(64), comments VARCHAR(1024)")

.jdbc("jdbc:postgresql:dbserver", "schema.tablename", connectionProperties)

## 综合实例

### 经典问题之简单四表联查

已知学生信息（student）、教师信息（teacher）、课程信息（course）和成绩信息（score），对这些信息进行指定查询，分别得到需要的结果。

学生信息如下：

108,ZhangSan,male,1995/9/1,95033

105,KangWeiWei,female,1996/6/1,95031

107,GuiGui,male,1992/5/5,95033

101,WangFeng,male,1993/8/8,95031

106,LiuBing,female,1996/5/20,95033

109,DuBingYan,male,1995/5/21,95031

教师信息如下 ：

825,LinYu,male,1958/1/1,Associate professor,department of computer

804,DuMei,female,1962/1/1,Assistant professor,computer science department

888,RenLi,male,1972/5/1,Lecturer,department of electronic engneering

852,GongMOMO,female,1986/1/5,Associate professor,computer science department

864,DuanMu,male,1985/6/1,Assistant professor,department of computer

课程信息如下：

3-105,Introduction to computer,825

3-245,The operating system,804

6-101,Spark SQL,888

6-102,Spark,852

9-106,Scala,864

成绩信息如下：

108,3-105,99

105,3-105,88

107,3-105,77

105,3-245,87

108,3-245,89

107,3-245,82

106,3-245,74

107,6-101,75

108,6-101,82

106,6-101,65

109,6-102,99

101,6-102,79

105,9-106,81

106,9-106,97

107,9-106,65

108,9-106,100

109,9-106,82

105,6-102,85

代码如下：

package com.cdpsql1

import org.apache.spark.sql.{Row, SparkSession}

import org.apache.spark.sql.types.\_

import scala.collection.mutable

import java.text.SimpleDateFormat

object SparkSQL01 {

def main(args: Array[String]): Unit = {

/\*\*

\* sparksession

\*/

val spark = SparkSession

.builder()

.master("local")

.appName("test")

.config("spark.sql.shuffle.partitions", "5")

.getOrCreate()

/\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* student表结构\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

val studentRDD = spark.sparkContext.textFile("/home/ubuntu01/SqlExample/student.txt")

val StudentSchema: StructType = StructType(mutable.ArraySeq( //学生表

StructField("Sno", StringType, nullable = false), //学号

StructField("Sname", StringType, nullable = false), //学生姓名

StructField("Ssex", StringType, nullable = false), //学生性别

StructField("Sbirthday", StringType, nullable = true), //学生出生年月

StructField("SClass", StringType, nullable = true) //学生所在班级

))

val studentData = studentRDD.map(\_.split(",")).map(attributes => Row(attributes(0),attributes(1),attributes(2),attributes(3),attributes(4)))

val studentDF = spark.createDataFrame(studentData,StudentSchema)

studentDF.createOrReplaceTempView("student")

/\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* teacher表结构\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

val teacherRDD =

spark.sparkContext.textFile("/home/ubuntu01/SqlExample/teacher.txt")

val TeacherSchema: StructType = StructType(mutable.ArraySeq( //教师表

StructField("Tno", StringType, nullable = false), //教工编号（主键）

StructField("Tname", StringType, nullable = false), //教工姓名

StructField("Tsex", StringType, nullable = false), //教工性别

StructField("Tbirthday", StringType, nullable = true), //教工出生年月

StructField("Prof", StringType, nullable = true), //职称

StructField("Depart", StringType, nullable = false) //教工所在部门

))

val teacherData = teacherRDD.map(\_.split(",")).map(attributes => Row(attributes(0),attributes(1),attributes(2),attributes(3),attributes(4),attributes(5)))

val teacherDF = spark.createDataFrame(teacherData,TeacherSchema)

teacherDF.createOrReplaceTempView("teacher")

/\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* course表结构\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

val courseRDD = spark.sparkContext.textFile("/home/ubuntu01/SqlExample/course.txt")

val CourseSchema: StructType = StructType(mutable.ArraySeq( //课程表

StructField("Cno", StringType, nullable = false), //课程号

StructField("Cname", StringType, nullable = false), //课程名称

StructField("Tno", StringType, nullable = false) //教工编号

))

val courseData = courseRDD.map(\_.split(",")).map(attributes => Row(attributes(0),attributes(1),attributes(2)))

val courseDF = spark.createDataFrame(courseData,CourseSchema)

courseDF.createOrReplaceTempView("course")

/\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* score表结构\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

val scoreRDD = spark.sparkContext.textFile("/home/ubuntu01/SqlExample/score.txt")

val ScoreSchema: StructType = StructType(mutable.ArraySeq( //成绩表

StructField("Sno", StringType, nullable = false), //学号（外键）

StructField("Cno", StringType, nullable = false), //课程号（外键）

StructField("Degree", IntegerType, nullable = true) //成绩

))

val scoreData = scoreRDD.map(\_.split(",")).map(attributes => Row(attributes(0),attributes(1),attributes(2)))

val scoreDF = spark.createDataFrame(scoreData,ScoreSchema)

scoreDF.createOrReplaceTempView("score")

/\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*对各表的处理\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//按照班级降序排序显示所有学生信息

spark.sql("SELECT \* FROM student ORDER BY SClass DESC").show()

//+-----+---------------+-----------+------------+---------+

//|Sno| Sname| Ssex|Sbirthday| SClass|

//+-----+---------------+-----------+------------+---------+

//|107| GuiGui| male| 1992/5/5| 95033|

//|108| ZhangSan| male| 1995/9/1| 95033|

//|106| LiuBing| female|1996/5/20| 95033|

//|105|KangWeiWei|female| 1996/6/1| 95031|

//|101| WangFeng| male| 1993/8/8| 95031|

//|109| DuBingYan| male|1995/5/21| 95031|

//+-----+---------------+-----------+--------------+---------+

//查询“计算机系”与“电子工程系“不同职称的教师的Tname和Prof。

spark.sql("SELECT tname, prof " +

"FROM Teacher " +

"WHERE prof NOT IN (SELECT a.prof " +

"FROM (SELECT prof " +

"FROM Teacher " +

"WHERE depart = 'department of computer' " +

") a " +

"JOIN (SELECT prof " +

"FROM Teacher " +

"WHERE depart = 'department of electronic engineering' " +

") b ON a.prof = b.prof) ").show(false)

//+-----------------+-------------------------+

//|tname |prof |

//+-----------------+-------------------------+

//|LinYu |Associate professor|

//|DuMei |Assistant professor|

//|RenLi |Lecturer |

//|GongMOMO|Associate professor|

//|DuanMu | Assistant professor|

//+-----------------+-------------------------+

//显示Sname列名，以一个Column的形式返回

println(studentDF.col("Sname"))

// Sname

//显示student表中记录数

println(studentDF.count())

//6

//显示student表中名字和性别的信息

studentDF.select("Sname","Ssex").show()

//+-----------------+---------+

//| Sname| Ssex|

//+-----------------+---------+

//| ZhangSan| male|

//|KangWeiWei|female|

//| GuiGui| male|

//| WangFeng| male|

//| LiuBing| female|

//| DuBingYan| male|

//+--------------+----------+

//显示性别为男的教师信息

teacherDF.filter("Tsex = 'male'").show(false)

//+-----+---------+------+------------+-----------------------+-------------------------------------+

//|Tno|Tname |Tsex|Tbirthday| Prof | Depart |

//+-----+---------+------+------------+------------------------+------------------------------------+

//|825|LinYu|male|1958/1/1|Associate professor| department of computer|

//|888|RenLi |male|1972/5/1 |Lecturer |department of electronic engneering|

//|864|DuanMu|male|1985/6/1|Assistant professor|department of computer|

//+-----+---------+------+------------+------------------------+-------------------------------------+

//显示不重复的教师部门信息

teacherDF.select("Depart").distinct().show(false)

//+------------------------------------------------+

//|Depart |

//+------------------------------------------------+

//|department of computer |

//|computer science department |

//|department of electronic engneering|

//+-------------------------------------------------+

//显示学号为101的学生信息

studentDF.where("Sno = '101'").show()

//+-----+--------------+-----+-------------+--------+

//|Sno| Sname|Ssex|Sbirthday| SClass|

//+-----+--------------+-----+-------------+---------+

//|101|WangFeng|male|1993/8/8|95031|

//+-----+--------------+-----+-------------+---------+

//将教师信息以List的形式显示

println(teacherDF.collectAsList())

//[[825,LinYu,male,1958/1/1,Associate professor,department of computer], //[804,DuMei,female,1962/1/1,Assistant professor,computer science department], //[888,RenLi,male,1972/5/1,Lecturer,department of electronic engneering], //[852,GongMOMO,female,1986/1/5,Associate professor,computer science //department], [864,DuanMu,male,1985/6/1,Assistant professor,department of //computer]]

//查询所有“女”教师和“女”同学的name、sex和birthday

spark.sql("SELECT sname, ssex, sbirthday " +

"FROM Student " +

"WHERE ssex = 'female' " +

"UNION " +

"SELECT tname, tsex, tbirthday " +

"FROM Teacher " +

"WHERE tsex = 'female'").show()

//+-----------------+----------+------------+

//| sname| ssex|sbirthday|

//+-----------------+----------+------------+

//| GongMOMO|female| 1986/1/5|

//|KangWeiWei|female| 1996/6/1|

//| LiuBing|female|1996/5/20|

//| DuMei|female| 1962/1/1|

//+----------------+----------+-------------+

}

}

## 本章小结

本章重点讲解Spark SQL和DataFrame相关知识，从Spark SQL的概念、架构信息讲起，通过DataFrame与RDD的区别，引入DataFrame。重点讲解了创建DataFrame，以及DataFrame的Transformation操作和Action操作等，同时讲解了两种由RDD转换生成DataFrame的操作。针对DataFrame的数据源，介绍了加载保存操作，如何通过parquet文件、JSON数据集、Hive表、JDBC连接数据库构建数据源。Spark SQL多元一体的结构化数据处理能力正在逐渐释放，令Spark的生态更加健壮和多样。