目录

[今日任务 1](#_Toc29252)

[教学重点 1](#_Toc11826)

[第一章 SparkSQL入门 2](#_Toc14619)

[1.1 SparkSQL介绍 2](#_Toc30709)

[1.2 SparkSQL的操作方式 4](#_Toc2452)

[1.3 SparkSQL的数据抽象 4](#_Toc25440)

[第二章 SparkSQL应用操作 8](#_Toc32366)

[2.1 Spark-shell基本操作 8](#_Toc30370)

[2.2 IDEA编写SparkSQL 8](#_Toc10719)

[2.3 数据转换 9](#_Toc5326)

[2.4 总结RDD、DataFrame、DataSet的转换 12](#_Toc32124)

[2.5 数据操作方法 13](#_Toc24754)

[2.6 SparkSQL自定义函数 30](#_Toc21056)

[第三章 SparkSQL集成Hive 35](#_Toc4918)

[3.1 使用内置hive 36](#_Toc6627)

[3.2 集成外部hive 36](#_Toc18907)

[3.3 通过代码操作 37](#_Toc19432)

[3.4 连接服务器hive 38](#_Toc13845)

[3.5 连接服务器hive[原始版本] 39](#_Toc29698)

[第四章 SparkSQL的输入和输出 40](#_Toc7315)

[4.1 SparkSQL的输入 40](#_Toc13507)

[4.2 SparkSQL的输出 41](#_Toc19384)

[4.3 JDBC操作 42](#_Toc26809)

[第五章 性能调优 44](#_Toc10952)

[5.1 缓存数据至内存 44](#_Toc7513)

[5.2 调优参数 44](#_Toc20378)

[第六章 JDBC/ODBC服务器 45](#_Toc15983)

[第七章 Spark SQL 的运行原理 45](#_Toc14034)

## 今日任务

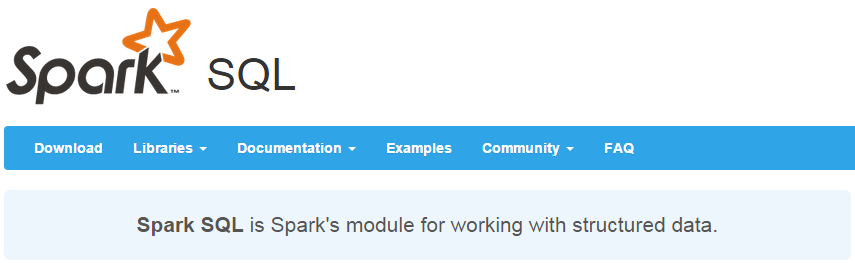
1. 理解DataFrame和DataSet对象  
2. 熟悉对SparkSQL的API操作  
3. 熟悉对Hive-On-Spark的操作

## 教学重点

1. RDD、DataFrame、DataSet互相转换  
2. DSL风格语法的操作  
3. SQL风格语法的操作  
4. Hive-On-Spark的操作  
5. 输入和输出

## 第一章 SparkSQL入门

### 1.1 SparkSQL介绍

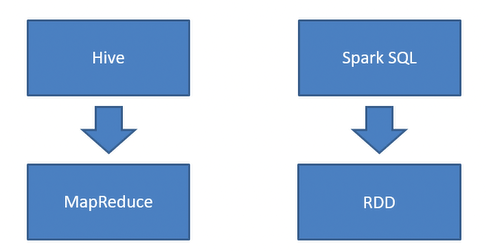


Spark SQL是Spark用来处理结构化数据的一个模块，它提供了一个编程抽象叫做DataFrame并且作为分布式SQL查询引擎的作用。

我们已经学习了Hive，它是将Hive SQL转换成MapReduce然后提交到集群上执行，大大简化了编写MapReduce的程序的复杂性，由于MapReduce这种计算模型执行效率比较慢。所有Spark SQL的应运而生，它是将Spark SQL转换成RDD，然后提交到集群执行，执行效率非常快！

Hive的应用其实就是应对不会写Java的开发人员但是有会写SQL的数据库而言提供的是MR的一种简化

SparkSQL其实也就是对之前所学习的SparkCore中RDD的一种简化,用SQL的语言可以对RDD编程进行开发



ps:Spark是由处理上限的10PB但是超过这个范围还是要使用hive来进行处理的hive应该是在100PB级别

1.易整合



2.统一的数据访问方式



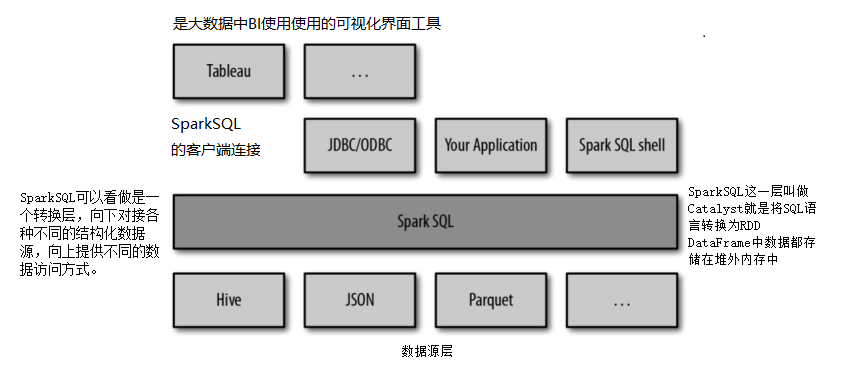
3.兼容Hive



4.标准的数据连接



### 1.2 SparkSQL的操作方式



### 1.3 SparkSQL的数据抽象

对于SparkCore而言对数据的进行操作需要先转换成RDD,对RDD可以使用各种算子进行处理,最终对数据进行统一的操作,所以我们将RDD看做是对数据的封装(抽象)

对于SparkSQL而言对数据进行操作的也需要进行转换,这里提供了两个新的抽象,分别是DataFrame和DataSet

#### 1.3.1 RDD vs DataFrames vs DataSet

首先从版本的产生上来看 RDD (Spark1.0) —> Dataframe(Spark1.3) —> Dataset(Spark1.6)

##### 1.3.1.1 RDD

RDD是一个懒执行的不可变的可以支持Functional(函数式编程)的并行数据集合。

RDD的最大好处就是简单，API的人性化程度很高。

RDD的劣势是性能限制，它是一个JVM驻内存对象，这也就决定了存在GC的限制和数据增加时Java序列化成本的升高。

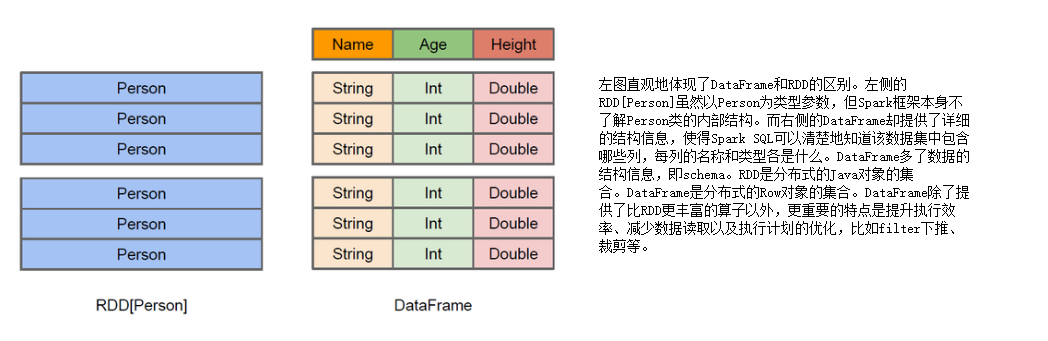
##### 1.3.1.2 DataFrame

简单来数DataFrame是RDD+Schema的集合

什么是Schema?

之前我们学习过MySQL数据库,在数据库中schema是数据库的组织和结构模式中包含了schema对象，可以是**表**(table)、**列**(column)、**数据类型**(data type)、**视图**(view)、**存储过程**(stored procedures)、**关系**(relationships)、**主键**(primary key)、**外键(**foreign key)等,Sechema代表的就是一张表

与RDD类似，DataFrame也是一个分布式数据容器。然而DataFrame更像传统数据库的二维表格，除了数据以外，还记录数据的结构信息，即schema。同时，与Hive类似，DataFrame也支持嵌套数据类型（struct、array和map）。从API易用性的角度上看，DataFrame API提供的是一套高层的关系操作，比函数式的RDD API要更加友好，门槛更低。由于与R和Pandas的DataFrame类似，Spark DataFrame很好地继承了传统单机数据分析的开发体验。

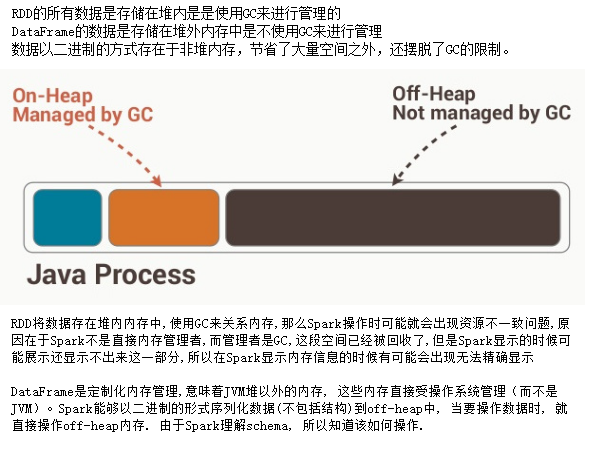


DataFrame是为数据提供了Schema的视图。可以把它当做数据库中的一张表来对待

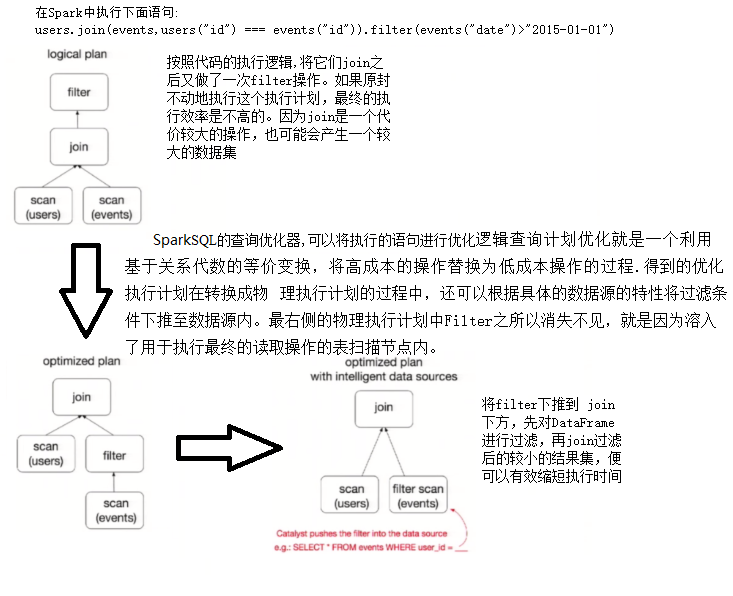
DataFrame也是懒执行的。

性能上比RDD要高，主要有两方面原因：

###### 定制化内存管理



##### 1.3.1.3 优化的执行计划



Dataframe的劣势在于在编译期缺少类型安全检查，导致运行时出错.

ps:DataFrame只是知道字段，但是不知道字段的类型，所以在执行这些操作的时候是没办法在编译的时候检查是否类型失败的，比如你可以对一个String进行减法操作，在执行的时候才报错，而DataSet不仅仅知道字段，而且知道字段类型，所以有更严格的错误检查

##### 1.3.1.4 Dataset

是Dataframe API的一个扩展，是Spark最新的数据抽象

用户友好的API风格，既具有类型安全检查也具有Dataframe的查询优化特性。

Dataset支持编解码器，当需要访问非堆上的数据时可以避免反序列化整个对象，提高了效率。

样例类被用来在Dataset中定义数据的结构信息，样例类中每个属性的名称直接映射到DataSet中的字段名称。

Dataframe是Dataset的特列，DataFrame=Dataset[Row] ，所以可以通过as方法将Dataframe转换为Dataset。Row是一个类型，跟Car、Person这些的类型一样，所有的表结构信息我们都用Row来表示。

DataSet是强类型的。比如可以有Dataset[Car]，Dataset[Person].

ps:DataSet[Row]这个类似于我们学习的泛型Row就是泛型类型

#### 1.3.2 三者的共性

1、RDD、DataFrame、Dataset全都是spark平台下的分布式弹性数据集，为处理超大型数据提供便利

2、三者都有惰性机制，在进行创建、转换，如map方法时，不会立即执行，只有在遇到Action如foreach时，三者才会开始遍历运算，极端情况下，如果代码里面有创建、转换，但是后面没有在Action中使用对应的结果，在执行时会被直接跳过.

3、三者都会根据spark的内存情况自动缓存运算，这样即使数据量很大，也不用担心会内存溢出

4、三者都有partition的概念

5、三者有许多共同的函数，如filter，排序等

6、在对DataFrame和Dataset进行操作许多操作都需要这个包进行支持

7、DataFrame和Dataset均可使用模式匹配获取各个字段的值和类型

#### 1.3.3 三者的区别

##### 1.3.3.1 RDD

1、RDD一般和spark mlib同时使用

2、RDD不支持sparksql操作

##### 1.3.3.2 DataFrame

1、与RDD和Dataset不同，DataFrame每一行的类型固定为Row，只有通过解析才能获取各个字段的值每一列的值没法直接访问

2、DataFrame与Dataset一般不与spark mlib同时使用

3、DataFrame与Dataset均支持sparksql的操作，比如select，groupby之类，还能注册临时表/视窗，进行sql语句操作

4、DataFrame与Dataset支持一些特别方便的保存方式，比如保存成csv，可以带上表头，这样每一列的字段名一目了然

利用这样的保存方式，可以方便的获得字段名和列的对应，而且分隔符（delimiter）可以自由指定。

##### 1.3.3.3 Dataset

Dataset和DataFrame拥有完全相同的成员函数，区别只是每一行的数据类型不同。

DataFrame也可以叫Dataset[Row],每一行的类型是Row，不解析，每一行究竟有哪些字段，各个字段又是什么类型都无从得知，只能用上面提到的getAS方法或者共性中的第七条提到的模式匹配拿出特定字段

而Dataset中，每一行是什么类型是不一定的，在自定义了case class之后可以很自由的获得每一行的信息

Dataset在需要访问列中的某个字段时是非常方便的，然而，如果要写一些适配性很强的函数时，如果使用Dataset，行的类型又不确定，可能是各种case class，无法实现适配，这时候用DataFrame即Dataset[Row]就能比较好的解决问题

## 第二章 SparkSQL应用操作

在老的版本中，SparkSQL提供两种SQL查询起始点，一个叫SQLContext，用于Spark自己提供的SQL查询，一个叫HiveContext，用于连接Hive的查询，SparkSession是Spark最新的SQL查询起始点，实质上是SQLContext和HiveContext的组合，所以在SQLContext和HiveContext上可用的API在SparkSession上同样是可以使用的。SparkSession内部封装了sparkContext，所以计算实际上是由sparkContext完成的。

### 2.1 Spark-shell基本操作

ps:数据使用的是Spark中所提供的样例数据

spark.read.json("/opt/software/spark-2.2.0-bin-hadoop2.7/examples/src/main/resources/people.json")  
df.show()  
df.filter($"age" > 21).show()  
df.createOrReplaceTempView("persons")  
spark.sql("select\* from persons").show()  
spark.sql("select \* from persons where age > 21").show()

### 2.2 IDEA编写SparkSQL

#### 2.2.1 在maven的pom.xml配置文件中添加配置

<dependency>  
 <groupId>org.apache.spark</groupId>  
 <artifactId>spark-sql\_2.11</artifactId>  
 <version>${spark.version}</version>  
 </dependency>

#### 2.2.2 SparkSession的三种创建方式

import org.apache.spark.{SparkConf, SparkContext}  
import org.apache.spark.sql.{DataFrame, SparkSession}  
/\*\*  
 \* SparkSession三种创建方式  
 \*/  
object SparkSQLDemo {  
 def main(args: Array[String]): Unit = {  
 /\*\*  
 \* 创建SparkSession方式1  
 \* builder用于创建一个SparkSession。  
 \* appName设置App的名字  
 \* master设置运行模式(集群模式不用设置)  
 \* getOrCreate 进行创建  
 \*/  
 val sparks1 = SparkSession.builder().appName("SparkSQLDemo").master("local").getOrCreate()  
 /\*\*  
 \* 创建SparkSession方式2  
 \* 先通过SparkConf创建配置对象  
 \* SetAppName设置应用的名字  
 \* SetMaster设置运行模式(集群模式不用设置)  
 \* 在通过SparkSession创建对象  
 \* 通过config传入conf配置对象来创建SparkSession  
 \* getOrCreate 进行创建  
 \*/  
 val conf = new SparkConf().setAppName("SparkSQLDemo").setMaster("local")  
 val sparks2 = SparkSession.builder().config(conf).getOrCreate()  
 /\*\*  
 \* 创建SparkSession方式3(操作hive)  
 \* uilder用于创建一个SparkSession。  
 \* appName设置App的名字  
 \* master设置运行模式(集群模式不用设置)  
 \* enableHiveSupport 开启hive操作  
 \* getOrCreate 进行创建  
 \*/  
 val sparkh = SparkSession.builder().appName("SparkSQLDemo"). master("local").enableHiveSupport().getOrCreate()  
   
 //关闭  
 sparks1.stop()  
 sparks2.stop()  
 sparkh.stop()  
 }  
}

### 2.3 数据转换

#### 2.3.1 RDD转换为DataFrame

##### 直接手动确定

import org.apache.spark.{SparkConf, SparkContext}  
import org.apache.spark.sql.{DataFrame, SparkSession}  
  
import org.apache.spark.rdd.RDD  
/\*\*  
 \* RDD--->DataFrame 直接手动确定  
 \*/  
object SparkSQLDemo {  
 def main(args: Array[String]): Unit = {  
 //创建SparkConf()并设置App名称  
 val conf = new SparkConf().setAppName("SparkSQLDemo").setMaster("local")  
 //SQLContext要依赖SparkContext  
 val sc = new SparkContext(conf)  
 //从指定的地址创建RDD  
 val lineRDD = sc.textFile("dir/people.txt").map(\_.split(","))  
 //这里是将数据转换为元组,数据量少可以使用这种方式  
 val tuple: RDD[(String, Int)] = lineRDD.map(x => (x(0),x(1).trim().toInt))  
 val spark = SparkSession.builder().config(conf).getOrCreate()  
 //如果需要RDD于DataFrame之间操作,那么需要引用 import spark.implicits.\_ [spark不是包名,是SparkSession对象]  
 import spark.implicits.\_  
 val frame: DataFrame = tuple.toDF("name","age")  
 frame.show()  
 sc.stop();  
 }  
}

##### 通过反射获取Scheam

import org.apache.spark.{SparkConf, SparkContext}  
import org.apache.spark.sql.{DataFrame, SparkSession}  
/\*\*  
 \* RDD--->DataFrame 通过反射推断Schema  
 \*/  
object SparkSQLDemo {  
 def main(args: Array[String]): Unit = {  
 //创建SparkConf()并设置App名称  
 val conf = new SparkConf().setAppName("SparkSQLDemo").setMaster("local")  
 //SQLContext要依赖SparkContext  
 val sc = new SparkContext(conf)  
 //从指定的地址创建RDD  
 val lineRDD = sc.textFile("dir/people.txt").map(\_.split(","))  
 //除了这个方式之外我们还可以是用样例类的形式来做复杂的转换操作  
 val tuple = lineRDD.map(x => People(x(0),x(1).trim().toInt));  
 val spark = SparkSession.builder().config(conf).getOrCreate()  
 import spark.implicits.\_  
 //此时不需要使用指定列名会根据样例类中定义的属性来确定类名  
 val frame: DataFrame = tuple.toDF  
 frame.show()  
 sc.stop()  
 }  
}  
case class People(name:String,age:Int)

##### 1.1.3通过StructType直接指定Schema

import org.apache.spark.sql.types.{IntegerType, StringType, StructField, StructType}  
import org.apache.spark.{SparkConf, SparkContext}  
import org.apache.spark.sql.{DataFrame, Row, SparkSession}  
/\*\*  
 \* RDD--->DataFrame 通过StructType直接指定Schema  
 \*/  
object SparkSQLDemo {  
 def main(args: Array[String]): Unit = {  
 //创建SparkConf()并设置App名称  
 val conf = new SparkConf().setAppName("SparkSQLDemo").setMaster("local")  
 //创建SparkContext对象  
 val sc = new SparkContext(conf)  
 //创建SparkSession对象  
 val spark = SparkSession.builder().config(conf).getOrCreate()  
 //从指定的地址创建RDD  
 val lineRDD = sc.textFile("dir/people.txt").map(\_.split(","))  
 //通过StructType直接指定每个字段的schema  
 val schema = StructType(  
 List(  
 StructField("name", StringType, true),  
 StructField("age", IntegerType, true)  
 )  
 )  
 //将RDD映射到rowRDD  
 val rowRDD = lineRDD.map(p => Row(p(0), p(1).trim.toInt))  
 //将schema信息应用到rowRDD上  
 val peopleDataFrame = spark.createDataFrame(rowRDD, schema)  
 val frame: DataFrame = peopleDataFrame.toDF  
 frame.show()  
 sc.stop()  
 }  
}

#### 2.3.2 DataFrame转换成RDD

无论是通过那这种方式获取的DataFrame都可以使用一个方法转换  
val frameToRDD: RDD[Row] = frame.rdd  
frameToRDD.foreach(x => println(x.getString(0)+","+x.getInt(1)))

#### 2.3.3 RDD转换为DataSet

##### 通过反射获取Scheam(样例类模式)

import org.apache.spark.rdd.RDD  
import org.apache.spark.sql.types.{IntegerType, StringType, StructField, StructType}  
import org.apache.spark.{SparkConf, SparkContext}  
import org.apache.spark.sql.{DataFrame, Dataset, Row, SparkSession}  
/\*\*  
 \* RDD--->DataSet 通过反射获取Scheam  
 \*/  
object SparkSQLDemo {  
 def main(args: Array[String]): Unit = {  
 //创建SparkConf()并设置App名称  
 val conf = new SparkConf().setAppName("SparkSQLDemo").setMaster("local")  
 //创建SparkContext对象  
 val sc = new SparkContext(conf)  
 //创建SparkSession对象  
 val spark = SparkSession.builder().config(conf).getOrCreate()  
 val value: RDD[String] = sc.textFile("dir/people.txt")  
 import spark.implicits.\_  
 val dataSet: Dataset[People] = value.map {  
 x =>  
 val para = x.split(",");  
 People(para(0), para(1).trim().toInt);  
 }.toDS()  
 dataSet.show()  
 sc.stop()  
 }  
}  
case class People(name:String,age:Int)

#### 2.3.4 DataSet转化到RDD

//调用rdd方法  
val dataSetTORDD:RDD[People] = dataSet.rdd  
dataSetTORDD.foreach(x => println(x.name+","+x.age));

#### 2.3.5 DataSet转换DataFrame

//调用toDF方法,直接服用case class中定义的属性  
val frame: DataFrame = dataSet.toDF()  
frame.show()

#### 2.3.6 DataFrame转换DataSet

val value: Dataset[People] = frame.as[People]  
 case class People(name:String,age:Int)

### 2.4 总结RDD、DataFrame、DataSet的转换

SparkSQL支持两种类型分别为DataSet和DataFrame,这两种类型都支持从RDD转换为DataSet或DataFrame

#### 2.4.1 RDD转DataFrame有三种方法是

1.直接转换即使用元组的模式存储在转换

2.使用样例类的模式匹配Scheam在转换

3.StructType直接指定Schema在转换

#### 2.4.2 RDD转DataSet

1.使用样例类的模式匹配Scheam在转换

ps:其余读取文件的方式可以直接获取对应的DataFrame

#### 2.4.3 DataSet和DataFrame之间的互相转换

##### DataSet转换DataFrame

调用toDF方法,直接服用case class中定义的属性

##### DataFrame转换DataSet

调用as[对应样例类类名]

##### DataSet和DataFrame转换我RDD

DataSet对象或DataFrame对象调用rdd方法就可以转换为rdd

### 2.5 数据操作方法

#### 2.5.1 DSL风格语法

import org.apache.spark.sql.{DataFrame, SparkSession}  
import org.apache.spark.{SparkConf, SparkContext}  
  
object SparkSQL {  
 def main(args:Array[String]):Unit = {  
 //创建SparkConf()并设置App名称  
 val conf = new SparkConf().setAppName("SparkSQLDemo").setMaster("local")  
 val spark = SparkSession.builder().config(conf).getOrCreate()  
 val df: DataFrame = spark.read.json("dir/people.json")  
 //DSL风格语法:  
 df.show()  
 import spark.implicits.\_  
 // 打印Schema信息  
 df.printSchema()  
 df.select("name").show()  
 df.select($"name", $"age" + 1).show()  
 df.filter($"age" > 21).show()  
 df.groupBy("age").count().show()  
 spark.stop()  
 }  
}

#### 2.5.2 DSL常用方法练习

准备两个数据文件到HDFS，内容如下：

员工信息

{"name": "Leo", "age": 25, "depId": 1, "gender": "male", "salary": 20000.126}  
{"name": "Marry", "age": 30, "depId": 2, "gender": "female", "salary": 25000}  
{"name": "Jack", "age": 35, "depId": 1, "gender": "male", "salary": 15000}  
{"name": "Tom", "age": 42, "depId": 3, "gender": "male", "salary": 18000}  
{"name": "Kattie", "age": 21, "depId": 3, "gender": "female", "salary": 21000}  
{"name": "Jen", "age": 19, "depId": 2, "gender": "male", "salary": 8000}  
{"name": "Jen", "age": 30, "depId": 2, "gender": "female", "salary": 28000}  
{"name": "Tom", "age": 42, "depId": 3, "gender": "male", "salary": 18000}  
{"name": "XiaoFang", "age": 18, "depId": 3, "gender": "female", "salary": 58000}

部门信息

{"id": 1, "name": "Technical Department"}  
{"id": 2, "name": "Financial Department"}  
{"id": 3, "name": "HR Department"}

获取数据

val employeeDF = spark.read.json("hdfs://node01:9000/company/employee.json")  
val departmentDF = spark.read.json("hdfs://node01:9000/company/department.json")

##### Action操作

###### show

以表格的形式在输出中展示DataFrame中的数据，该方法有四种调用方式，分别为：

**（1）show** 只显示前20条记录。且过长的字符串会被截取

scala> employeeDF.show  
+---+-----+------+--------+---------+  
|age|depId|gender| name| salary|  
+---+-----+------+--------+---------+  
| 25| 1| male| Leo|20000.126|  
| 30| 2|female| Marry| 25000.0|  
| 35| 1| male| Jack| 15000.0|  
| 42| 3| male| Tom| 18000.0|  
| 21| 3|female| Kattie| 21000.0|  
| 19| 2| male| Jen| 8000.0|  
| 30| 2|female| Jen| 28000.0|  
| 42| 3| male| Tom| 18000.0|  
| 18| 3|female|XiaoFang| 58000.0|  
+---+-----+------+--------+---------+

**（2）show(numRows: Int)**

显示numRows条

scala> employeeDF.show(3)  
+---+-----+------+-----+---------+  
|age|depId|gender| name| salary|  
+---+-----+------+-----+---------+  
| 25| 1| male| Leo|20000.126|  
| 30| 2|female|Marry| 25000.0|  
| 35| 1| male| Jack| 15000.0|  
+---+-----+------+-----+---------+  
only showing top 3 rows

**（3）show(truncate: Boolean)** 是否截取20个字符，默认为true。

scala> employeeDF.show(true)  
+---+-----+------+--------+---------+  
|age|depId|gender| name| salary|  
+---+-----+------+--------+---------+  
| 25| 1| male| Leo|20000.126|  
| 30| 2|female| Marry| 25000.0|  
| 35| 1| male| Jack| 15000.0|  
| 42| 3| male| Tom| 18000.0|  
| 21| 3|female| Kattie| 21000.0|  
| 19| 2| male| Jen| 8000.0|  
| 30| 2|female| Jen| 28000.0|  
| 42| 3| male| Tom| 18000.0|  
| 18| 3|female|XiaoFang| 58000.0|  
+---+-----+------+--------+---------+

**（4）show(numRows: Int, truncate: Int)** 显示记录条数，以及截取字符个数，为0时表示不截取

scala> employeeDF.show(3, 2)  
+---+-----+------+----+------+  
|age|depId|gender|name|salary|  
+---+-----+------+----+------+  
| 25| 1| ma| Le| 20|  
| 30| 2| fe| Ma| 25|  
| 35| 1| ma| Ja| 15|  
+---+-----+------+----+------+  
only showing top 3 rows

###### collect

获取DataFrame中的所有数据并返回一个Array对象

scala> employeeDF.collect  
res6: Array[org.apache.spark.sql.Row] = Array([25,1,male,Leo,20000.126], [30,2,female,Marry,25000.0], [35,1,male,Jack,15000.0], [42,3,male,Tom,18000.0], [21,3,female,Kattie,21000.0], [19,2,male,Jen,8000.0], [30,2,female,Jen,28000.0], [42,3,male,Tom,18000.0], [18,3,female,XiaoFang,58000.0])

###### collectAsList

功能和collect类似，只不过将返回结构变成了List对象

scala> employeeDF.collectAsList  
res7: java.util.List[org.apache.spark.sql.Row] = [[25,1,male,Leo,20000.126], [30,2,female,Marry,25000.0], [35,1,male,Jack,15000.0], [42,3,male,Tom,18000.0], [21,3,female,Kattie,21000.0], [19,2,male,Jen,8000.0], [30,2,female,Jen,28000.0], [42,3,male,Tom,18000.0], [18,3,female,XiaoFang,58000.0]]

###### describe

这个方法可以动态的传入一个或多个String类型的字段名，结果仍然为DataFrame对象，用于统计数值类型字段的统计值，比如count, mean, stddev, min, max等。

scala> employeeDF.describe("age", "salary").show()  
+-------+-----------------+------------------+  
|summary| age| salary|  
+-------+-----------------+------------------+  
| count| 9| 9|  
| mean|29.11111111111111|23444.458444444444|  
| stddev|9.198429817697752|14160.779261027332|  
| min| 18| 8000.0|  
| max| 42| 58000.0|  
+-------+-----------------+------------------+

###### first, head, take, takeAsList

（1）first获取第一行记录 （2）head获取第一行记录，head(n: Int)获取前n行记录 （3）take(n: Int)获取前n行数据 （4）takeAsList(n: Int)获取前n行数据，并以List的形式展现 以Row或者Array[Row]的形式返回一行或多行数据。first和head功能相同。 take和takeAsList方法会将获得到的数据返回到Driver端，所以，使用这两个方法时需要注意数据量，以免Driver发生OutOfMemoryError。

以上方法简单，使用过程和结果略。

##### 条件查询和Join操作

以下返回为DataFrame类型的方法，可以连续调用。

###### where条件

等同于SQL语言中where关键字。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。

scala> employeeDF.where("depId = 3").show  
+---+-----+------+--------+-------+  
|age|depId|gender| name| salary|  
+---+-----+------+--------+-------+  
| 42| 3| male| Tom|18000.0|  
| 21| 3|female| Kattie|21000.0|  
| 42| 3| male| Tom|18000.0|  
| 18| 3|female|XiaoFang|58000.0|  
+---+-----+------+--------+-------+  
  
scala> employeeDF.where("depId = 3 and salary > 18000").show  
+---+-----+------+--------+-------+  
|age|depId|gender| name| salary|  
+---+-----+------+--------+-------+  
| 21| 3|female| Kattie|21000.0|  
| 18| 3|female|XiaoFang|58000.0|  
+---+-----+------+--------+-------+

###### filter过滤

传入筛选条件表达式，得到DataFrame类型的返回结果。和where使用条件相同。

scala> employeeDF.filter("depId = 3 or salary > 18000").show  
+---+-----+------+--------+---------+  
|age|depId|gender| name| salary|  
+---+-----+------+--------+---------+  
| 25| 1| male| Leo|20000.126|  
| 30| 2|female| Marry| 25000.0|  
| 42| 3| male| Tom| 18000.0|  
| 21| 3|female| Kattie| 21000.0|  
| 30| 2|female| Jen| 28000.0|  
| 42| 3| male| Tom| 18000.0|  
| 18| 3|female|XiaoFang| 58000.0|  
+---+-----+------+--------+---------+

###### 查询指定字段

**select**

根据传入的String类型字段名，获取指定字段的值，以DataFrame类型返回。

scala> employeeDF.select("name", "age").show(3)  
+-----+---+  
| name|age|  
+-----+---+  
| Leo| 25|  
|Marry| 30|  
| Jack| 35|  
+-----+---+  
only showing top 3 rows

还有一个重载的select方法，不是传入String类型参数，而是传入Column类型参数。可以实现select id, id+1 from table这种逻辑。

scala> employeeDF.select(employeeDF("age"), employeeDF("age") + 1 ).show(5)  
+---+---------+  
|age|(age + 1)|  
+---+---------+  
| 25| 26|  
| 30| 31|  
| 35| 36|  
| 42| 43|  
| 21| 22|  
+---+---------+  
only showing top 5 rows

能得到Column类型的方法是column和col方法，col方法更简便一些。

scala> employeeDF.select(col("age"), col("age") + 1 ).show(5)  
+---+---------+  
|age|(age + 1)|  
+---+---------+  
| 25| 26|  
| 30| 31|  
| 35| 36|  
| 42| 43|  
| 21| 22|  
+---+---------+  
only showing top 5 rows

**selectExpr**

可以直接对指定字段调用UDF函数，或者指定别名等。传入String类型参数，得到DataFrame对象。

scala> employeeDF.selectExpr("name" , "depId as departmentId" , "round(salary)").show()  
+--------+------------+----------------+  
| name|departmentId|round(salary, 0)|  
+--------+------------+----------------+  
| Leo| 1| 20000.0|  
| Marry| 2| 25000.0|  
| Jack| 1| 15000.0|  
| Tom| 3| 18000.0|  
| Kattie| 3| 21000.0|  
| Jen| 2| 8000.0|  
| Jen| 2| 28000.0|  
| Tom| 3| 18000.0|  
|XiaoFang| 3| 58000.0|  
+--------+------------+----------------+

**col**

只能获取一个字段，返回对象为Column类型。

scala> employeeDF.select(col("name")).show(3)  
+-----+  
| name|  
+-----+  
| Leo|  
|Marry|  
| Jack|  
+-----+  
only showing top 3 rows

**drop**

返回一个新的DataFrame对象，其中不包含去除的字段，一次只能去除一个字段。

scala> employeeDF.drop("gender").show(3)  
+---+-----+-----+---------+  
|age|depId| name| salary|  
+---+-----+-----+---------+  
| 25| 1| Leo|20000.126|  
| 30| 2|Marry| 25000.0|  
| 35| 1| Jack| 15000.0|  
+---+-----+-----+---------+  
only showing top 3 rows

###### limit

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。

scala> employeeDF.limit(3).show  
+---+-----+------+-----+---------+  
|age|depId|gender| name| salary|  
+---+-----+------+-----+---------+  
| 25| 1| male| Leo|20000.126|  
| 30| 2|female|Marry| 25000.0|  
| 35| 1| male| Jack| 15000.0|  
+---+-----+------+-----+---------+

###### order by

**orderBy和sort**

按指定字段排序，默认为升序。加个“-”号表示降序排序。只对数字类型和日期类型生效。sort和orderBy使用方法相同。

scala> employeeDF.orderBy(- col("salary")).show(3)  
scala> employeeDF.sort(- col("salary")).show(3)  
+---+-----+------+--------+-------+  
|age|depId|gender| name| salary|  
+---+-----+------+--------+-------+  
| 18| 3|female|XiaoFang|58000.0|  
| 30| 2|female| Jen|28000.0|  
| 30| 2|female| Marry|25000.0|  
+---+-----+------+--------+-------+  
only showing top 3 rows

**sortWithinPartitions**

和上面的sort方法功能类似，区别在于sortWithinPartitions方法返回的是按Partition排好序的DataFrame对象。

val repartitioned = employeeDF.repartition(2)  
repartitioned.sortWithinPartitions("salary").show  
+---+-----+------+--------+---------+  
|age|depId|gender| name| salary|  
+---+-----+------+--------+---------+  
| 35| 1| male| Jack| 15000.0|  
| 25| 1| male| Leo|20000.126|  
| 21| 3|female| Kattie| 21000.0|  
| 30| 2|female| Jen| 28000.0|  
| 18| 3|female|XiaoFang| 58000.0|  
| 19| 2| male| Jen| 8000.0|  
| 42| 3| male| Tom| 18000.0|  
| 42| 3| male| Tom| 18000.0|  
| 30| 2|female| Marry| 25000.0|  
+---+-----+------+--------+---------+

###### group by

**groupBy**

根据指定字段进行group by操作，可以指定多个字段。

groupBy方法有两种调用方式，可以传入String类型的字段名，也可传入Column类型的对象。

employeeDF.groupBy("depId").count.show  
employeeDF.groupBy(employeeDF("depId")).count.show  
+-----+-----+  
|depId|count|  
+-----+-----+  
| 1| 2|  
| 3| 4|  
| 2| 3|  
+-----+-----+

**cube和rollup**

功能类似于SQL中的group by cube/rollup。

cube：为指定表达式集的每个可能组合创建分组集。首先会对(A、B、C)进行group by，然后依次是(A、B)，(A、C)，(A)，(B、C)，(B)，( C)，最后对全表进行group by操作。

rollup：在指定表达式的每个层次级别创建分组集。group by A,B,C with rollup首先会对(A、B、C)进行group by，然后对(A、B)进行group by，然后是(A)进行group by，最后对全表进行group by操作。

employeeDF.cube("depId","gender").sum("salary").show()  
+-----+------+------------------+  
|depId|gender| sum(salary)|  
+-----+------+------------------+  
| 3|female| 79000.0|  
| 1| male|35000.126000000004|  
| 1| null|35000.126000000004|  
| null| null| 211000.126|  
| 3| null| 115000.0|  
| 2| male| 8000.0|  
| 3| male| 36000.0|  
| null|female| 132000.0|  
| 2| null| 61000.0|  
| null| male| 79000.126|  
| 2|female| 53000.0|  
+-----+------+------------------+  
  
employeeDF.rollup("depId","gender").sum("salary").show()  
+-----+------+------------------+  
|depId|gender| sum(salary)|  
+-----+------+------------------+  
| 3|female| 79000.0|  
| 1| male|35000.126000000004|  
| 1| null|35000.126000000004|  
| null| null| 211000.126|  
| 3| null| 115000.0|  
| 2| male| 8000.0|  
| 3| male| 36000.0|  
| 2| null| 61000.0|  
| 2|female| 53000.0|  
+-----+------+------------------+

**GroupedData对象**

该方法得到的是GroupedData类型对象，在GroupedData的API中提供了group by之后的操作，比如：

* max(colNames: String\*)，获取分组中指定字段或者所有的数字类型字段的最大值，只能作用于数字类型字段
* min(colNames: String\*)，获取分组中指定字段或者所有的数字类型字段的最小值，只能作用于数字类型字段
* mean(colNames: String\*)，获取分组中指定字段或者所有的数字类型字段的平均值，只能作用于数字类型字段
* sum(colNames: String\*)，获取分组中指定字段或者所有的数字类型字段的和值，只能作用于数字类型字段
* count()，获取分组中的元素个数
* 运行结果示例：
* employeeDF.groupBy("depId").count.show  
  +-----+-----+  
  |depId|count|  
  +-----+-----+  
  | 1| 2|  
  | 3| 4|  
  | 2| 3|  
  +-----+-----+  
    
  employeeDF.groupBy("depId").max("salary").show  
  +-----+-----------+  
  |depId|max(salary)|  
  +-----+-----------+  
  | 1| 20000.126|  
  | 3| 58000.0|  
  | 2| 28000.0|  
  +-----+-----------+

###### distinct

**distinct**

返回当前DataFrame中不重复的Row记录。

employeeDF.distinct  
+---+-----+------+--------+---------+  
|age|depId|gender| name| salary|  
+---+-----+------+--------+---------+  
| 35| 1| male| Jack| 15000.0|  
| 30| 2|female| Jen| 28000.0|  
| 19| 2| male| Jen| 8000.0|  
| 18| 3|female|XiaoFang| 58000.0|  
| 25| 1| male| Leo|20000.126|  
| 42| 3| male| Tom| 18000.0|  
| 30| 2|female| Marry| 25000.0|  
| 21| 3|female| Kattie| 21000.0|  
+---+-----+------+--------+---------+

**dropDuplicates**

根据指定字段去重。类似于select distinct a, b操作。

employeeDF.dropDuplicates(Seq("name")).show  
+---+-----+------+--------+---------+  
|age|depId|gender| name| salary|  
+---+-----+------+--------+---------+  
| 35| 1| male| Jack| 15000.0|  
| 42| 3| male| Tom| 18000.0|  
| 19| 2| male| Jen| 8000.0|  
| 30| 2|female| Marry| 25000.0|  
| 21| 3|female| Kattie| 21000.0|  
| 25| 1| male| Leo|20000.126|  
| 18| 3|female|XiaoFang| 58000.0|  
+---+-----+------+--------+---------+

###### 聚合

聚合操作调用的是agg方法，该方法有多种调用方式。一般与groupBy方法配合使用。 以下示例其中最简单直观的一种用法，对age字段求最大值，对salary字段求和。

employeeDF.agg("age" -> "max", "salary" -> "sum").show  
+--------+-----------+  
|max(age)|sum(salary)|  
+--------+-----------+  
| 42| 211000.126|  
+--------+-----------+

###### union

对两个DataFrame进行合并

employeeDF.union(employeeDF.limit(1)).show  
+---+-----+------+--------+---------+  
|age|depId|gender| name| salary|  
+---+-----+------+--------+---------+  
| 25| 1| male| Leo|20000.126|  
| 30| 2|female| Marry| 25000.0|  
| 35| 1| male| Jack| 15000.0|  
| 42| 3| male| Tom| 18000.0|  
| 21| 3|female| Kattie| 21000.0|  
| 19| 2| male| Jen| 8000.0|  
| 30| 2|female| Jen| 28000.0|  
| 42| 3| male| Tom| 18000.0|  
| 18| 3|female|XiaoFang| 58000.0|  
| 25| 1| male| Leo|20000.126|  
+---+-----+------+--------+---------+

###### join

在SQL语言中很多场景需要用join操作，DataFrame中同样也提供了join的功能。 并且提供了六个重载的join方法。

**using一个字段形式**

下面这种join类似于a join b using column1的形式，需要两个DataFrame中有相同的一个列名。

val employeeDF2 = spark.read.json("hdfs://node01:9000/company/employee.json")  
employeeDF.join(employeeDF2, "depId").show  
+-----+---+------+------+---------+---+------+--------+---------+  
|depId|age|gender| name| salary|age|gender| name| salary|  
+-----+---+------+------+---------+---+------+--------+---------+  
| 1| 25| male| Leo|20000.126| 35| male| Jack| 15000.0|  
| 1| 25| male| Leo|20000.126| 25| male| Leo|20000.126|  
| 2| 30|female| Marry| 25000.0| 30|female| Jen| 28000.0|  
| 2| 30|female| Marry| 25000.0| 19| male| Jen| 8000.0|  
| 2| 30|female| Marry| 25000.0| 30|female| Marry| 25000.0|  
| 1| 35| male| Jack| 15000.0| 35| male| Jack| 15000.0|  
| 1| 35| male| Jack| 15000.0| 25| male| Leo|20000.126|  
| 3| 42| male| Tom| 18000.0| 18|female|XiaoFang| 58000.0|  
| 3| 42| male| Tom| 18000.0| 42| male| Tom| 18000.0|  
| 3| 42| male| Tom| 18000.0| 21|female| Kattie| 21000.0|  
| 3| 42| male| Tom| 18000.0| 42| male| Tom| 18000.0|  
| 3| 21|female|Kattie| 21000.0| 18|female|XiaoFang| 58000.0|  
| 3| 21|female|Kattie| 21000.0| 42| male| Tom| 18000.0|  
| 3| 21|female|Kattie| 21000.0| 21|female| Kattie| 21000.0|  
| 3| 21|female|Kattie| 21000.0| 42| male| Tom| 18000.0|  
| 2| 19| male| Jen| 8000.0| 30|female| Jen| 28000.0|  
| 2| 19| male| Jen| 8000.0| 19| male| Jen| 8000.0|  
| 2| 19| male| Jen| 8000.0| 30|female| Marry| 25000.0|  
| 2| 30|female| Jen| 28000.0| 30|female| Jen| 28000.0|  
| 2| 30|female| Jen| 28000.0| 19| male| Jen| 8000.0|  
+-----+---+------+------+---------+---+------+--------+---------+

**using多个字段形式**

除了上面这种using一个字段的情况外，还可以using多个字段，如下：

employeeDF.join(employeeDF2, Seq("name", "depId")).show  
+--------+-----+---+------+---------+---+------+---------+  
| name|depId|age|gender| salary|age|gender| salary|  
+--------+-----+---+------+---------+---+------+---------+  
| Leo| 1| 25| male|20000.126| 25| male|20000.126|  
| Marry| 2| 30|female| 25000.0| 30|female| 25000.0|  
| Jack| 1| 35| male| 15000.0| 35| male| 15000.0|  
| Tom| 3| 42| male| 18000.0| 42| male| 18000.0|  
| Tom| 3| 42| male| 18000.0| 42| male| 18000.0|  
| Kattie| 3| 21|female| 21000.0| 21|female| 21000.0|  
| Jen| 2| 19| male| 8000.0| 30|female| 28000.0|  
| Jen| 2| 19| male| 8000.0| 19| male| 8000.0|  
| Jen| 2| 30|female| 28000.0| 30|female| 28000.0|  
| Jen| 2| 30|female| 28000.0| 19| male| 8000.0|  
| Tom| 3| 42| male| 18000.0| 42| male| 18000.0|  
| Tom| 3| 42| male| 18000.0| 42| male| 18000.0|  
|XiaoFang| 3| 18|female| 58000.0| 18|female| 58000.0|  
+--------+-----+---+------+---------+---+------+---------+

**指定join类型**

在上面的using多个字段的join情况下，可以写第三个String类型参数，指定join的类型，如下所示：

inner：内连

outer,full,full\_outer：全连

left, left\_outer：左连

right，right\_outer：右连

left\_semi：过滤出joinDF1中和joinDF2共有的部分

left\_anti：过滤出joinDF1中joinDF2没有的部分

employeeDF.join(employeeDF2, Seq("name", "depId"), "inner").show  
+--------+-----+---+------+---------+---+------+---------+  
| name|depId|age|gender| salary|age|gender| salary|  
+--------+-----+---+------+---------+---+------+---------+  
| Leo| 1| 25| male|20000.126| 25| male|20000.126|  
| Marry| 2| 30|female| 25000.0| 30|female| 25000.0|  
| Jack| 1| 35| male| 15000.0| 35| male| 15000.0|  
| Tom| 3| 42| male| 18000.0| 42| male| 18000.0|  
| Tom| 3| 42| male| 18000.0| 42| male| 18000.0|  
| Kattie| 3| 21|female| 21000.0| 21|female| 21000.0|  
| Jen| 2| 19| male| 8000.0| 30|female| 28000.0|  
| Jen| 2| 19| male| 8000.0| 19| male| 8000.0|  
| Jen| 2| 30|female| 28000.0| 30|female| 28000.0|  
| Jen| 2| 30|female| 28000.0| 19| male| 8000.0|  
| Tom| 3| 42| male| 18000.0| 42| male| 18000.0|  
| Tom| 3| 42| male| 18000.0| 42| male| 18000.0|  
|XiaoFang| 3| 18|female| 58000.0| 18|female| 58000.0|  
+--------+-----+---+------+---------+---+------+---------+

**使用Column类型来join**

如果不用using模式，灵活指定join字段的话，可以使用如下形式：

employeeDF.join(departmentDF, employeeDF("depId") === departmentDF("id")).show  
+---+-----+------+--------+---------+---+--------------------+  
|age|depId|gender| name| salary| id| name|  
+---+-----+------+--------+---------+---+--------------------+  
| 25| 1| male| Leo|20000.126| 1|Technical Department|  
| 30| 2|female| Marry| 25000.0| 2|Financial Department|  
| 35| 1| male| Jack| 15000.0| 1|Technical Department|  
| 42| 3| male| Tom| 18000.0| 3| HR Department|  
| 21| 3|female| Kattie| 21000.0| 3| HR Department|  
| 19| 2| male| Jen| 8000.0| 2|Financial Department|  
| 30| 2|female| Jen| 28000.0| 2|Financial Department|  
| 42| 3| male| Tom| 18000.0| 3| HR Department|  
| 18| 3|female|XiaoFang| 58000.0| 3| HR Department|  
+---+-----+------+--------+---------+---+--------------------+

**在指定join字段同时指定join类型**

employeeDF.join(departmentDF, employeeDF("depId") === departmentDF("id"), "inner").show  
+---+-----+------+--------+---------+---+--------------------+  
|age|depId|gender| name| salary| id| name|  
+---+-----+------+--------+---------+---+--------------------+  
| 25| 1| male| Leo|20000.126| 1|Technical Department|  
| 30| 2|female| Marry| 25000.0| 2|Financial Department|  
| 35| 1| male| Jack| 15000.0| 1|Technical Department|  
| 42| 3| male| Tom| 18000.0| 3| HR Department|  
| 21| 3|female| Kattie| 21000.0| 3| HR Department|  
| 19| 2| male| Jen| 8000.0| 2|Financial Department|  
| 30| 2|female| Jen| 28000.0| 2|Financial Department|  
| 42| 3| male| Tom| 18000.0| 3| HR Department|  
| 18| 3|female|XiaoFang| 58000.0| 3| HR Department|  
+---+-----+------+--------+---------+---+--------------------+

###### 获取指定字段统计信息

stat方法可以用于计算指定字段或指定字段之间的统计信息，比如方差，协方差等。这个方法返回一个DataFramesStatFunctions类型对象。 下面代码演示根据字段，统计该字段值出现频率在30%以上的内容。在employeeDF中字段depId的内容为"1,2,3"。其中"1,2,3"出现频率均大于0.3。

employeeDF.stat.freqItems(Seq ("depId") , 0.3).show()  
+---------------+  
|depId\_freqItems|  
+---------------+  
| [2, 1, 3]|  
+---------------+

###### 获取两个DataFrame中共有的记录

intersect方法可以计算出两个DataFrame中相同的记录。

employeeDF.intersect(employeeDF.limit(1)).show()  
+---+-----+------+----+---------+  
|age|depId|gender|name|salary |  
+---+-----+------+----+---------+  
|25 |1 |male |Leo |20000.126|  
+---+-----+------+----+---------+

###### 获取一个DataFrame中有另一个DataFrame中没有的记录

employeeDF.except(employeeDF.limit(1)).show()  
+---+-----+------+--------+-------+  
|age|depId|gender|name |salary |  
+---+-----+------+--------+-------+  
|42 |3 |male |Tom |18000.0|  
|21 |3 |female|Kattie |21000.0|  
|19 |2 |male |Jen |8000.0 |  
|30 |2 |female|Jen |28000.0|  
|35 |1 |male |Jack |15000.0|  
|30 |2 |female|Marry |25000.0|  
|18 |3 |female|XiaoFang|58000.0|  
+---+-----+------+--------+-------+

###### 操作字段名

**withColumnRenamed**

如果指定的字段名不存在，不进行任何操作。下面示例中将employeeDF中的depId字段重命名为departmentId。

employeeDF.withColumnRenamed("depId", "departmentId").show  
+---+------------+------+--------+---------+  
|age|departmentId|gender| name| salary|  
+---+------------+------+--------+---------+  
| 25| 1| male| Leo|20000.126|  
| 30| 2|female| Marry| 25000.0|  
| 35| 1| male| Jack| 15000.0|  
| 42| 3| male| Tom| 18000.0|  
| 21| 3|female| Kattie| 21000.0|  
| 19| 2| male| Jen| 8000.0|  
| 30| 2|female| Jen| 28000.0|  
| 42| 3| male| Tom| 18000.0|  
| 18| 3|female|XiaoFang| 58000.0|  
+---+------------+------+--------+---------+

**withColumn**

whtiColumn(colName: String , col: Column)方法根据指定colName往DataFrame中新增一列，如果colName已存在，则会覆盖当前列。

employeeDF.withColumn("salary2", employeeDF("salary")).show  
+---+-----+------+--------+---------+---------+  
|age|depId|gender| name| salary| salary2|  
+---+-----+------+--------+---------+---------+  
| 25| 1| male| Leo|20000.126|20000.126|  
| 30| 2|female| Marry| 25000.0| 25000.0|  
| 35| 1| male| Jack| 15000.0| 15000.0|  
| 42| 3| male| Tom| 18000.0| 18000.0|  
| 21| 3|female| Kattie| 21000.0| 21000.0|  
| 19| 2| male| Jen| 8000.0| 8000.0|  
| 30| 2|female| Jen| 28000.0| 28000.0|  
| 42| 3| male| Tom| 18000.0| 18000.0|  
| 18| 3|female|XiaoFang| 58000.0| 58000.0|  
+---+-----+------+--------+---------+---------+

###### 行转列

有时候需要根据某个字段内容进行分割，然后生成多行，这时可以使用explode方法 。 下面代码中，根据name字段中的空格将字段内容进行分割，分割的内容存储在新的字段name\_中，如下所示：

departmentDF.explode("name", "name\_"){x:String => x.split(" ")}.show  
+---+--------------------+----------+  
| id| name| name\_|  
+---+--------------------+----------+  
| 1|Technical Department| Technical|  
| 1|Technical Department|Department|  
| 2|Financial Department| Financial|  
| 2|Financial Department|Department|  
| 3| HR Department| HR|  
| 3| HR Department|Department|  
+---+--------------------+----------+

##### DSL风格语法案例练习

创建employee.json文件，内容如下：

{"name": "Leo", "age": 25, "depId": 1, "gender": "male", "salary": 20000.126}  
{"name": "Marry", "age": 30, "depId": 2, "gender": "female", "salary": 25000}  
{"name": "Jack", "age": 35, "depId": 1, "gender": "male", "salary": 15000}  
{"name": "Tom", "age": 42, "depId": 3, "gender": "male", "salary": 18000}  
{"name": "Kattie", "age": 21, "depId": 3, "gender": "female", "salary": 21000}  
{"name": "Jen", "age": 19, "depId": 2, "gender": "male", "salary": 8000}  
{"name": "Jen", "age": 30, "depId": 2, "gender": "female", "salary": 28000}  
{"name": "Tom", "age": 42, "depId": 3, "gender": "male", "salary": 18000}  
{"name": "XiaoFang", "age": 18, "depId": 3, "gender": "female", "salary": 58000}

创建department.json文件，内容如下：

{"id": 1, "name": "Technical Department"}  
{"id": 2, "name": "Financial Department"}  
{"id": 3, "name": "HR Department"}

需求：统计各个部门不同性别大于20岁的平均薪资和平均年龄

object DSLDemo {  
 def main(args: Array[String]): Unit = {  
 val spark = SparkSession.builder()  
 .appName("DSLDemo")  
 .master("local[2]")  
 .getOrCreate()  
  
 // 获取数据  
 val employee = spark.read.json("c://employee.json")  
 val department = spark.read.json("c://department.json")  
  
 import spark.implicits.\_  
 import org.apache.spark.sql.functions.\_  
  
 // 开始统计  
 employee  
 .filter("age > 20")  
 .join(department, $"depId" === $"id")  
 .groupBy(department("name"), employee("gender"))  
 .agg(avg(employee("salary")).as("salary"), avg(employee("age")).as("age"))  
 .show()  
  
 spark.stop()  
 }  
}

#### 2.5.3 SQL风格语法

import org.apache.spark.sql.{DataFrame, SparkSession}  
import org.apache.spark.{SparkConf, SparkContext}  
  
object SparkSQL {  
 def main(args:Array[String]):Unit = {  
 //创建SparkConf()并设置App名称  
 val conf = new SparkConf().setAppName("SparkSQLDemo").setMaster("local")  
 val spark = SparkSession.builder().config(conf).getOrCreate()  
 val df: DataFrame = spark.read.json("dir/people.json")  
 //SQL风格语法:  
 //临时表是Session范围内的，Session退出后，表就失效了  
 //一个SparkSession结束后,表自动删除  
 df.createOrReplaceTempView("people")  
 val sqlDF = spark.sql("SELECT \* FROM people")  
 sqlDF.show()  
 //如果想应用范围内有效，可以使用全局表。注意使用全局表时需要全路径访问，如：global\_temp.people  
 //应用级别内可以访问,一个SparkContext结束后,表自动删除 一个SparkContext可以多次创建SparkSession  
 //使用的比较少  
 df.createGlobalTempView("people")  
 //创建名后需要必须添加global\_temp才可以  
 spark.sql("SELECT \* FROM global\_temp.people").show()  
 spark.newSession().sql("SELECT \* FROM global\_temp.people").show()  
 spark.stop()  
 }  
}

ps:需要打包上传集群,可以在集群中使用这个jar包

在安装spark的目录下进入到bin目录执行spark-submit

例如 :/opt/software/spark-2.2.0-bin-hadoop2.7/bin

--class 全类名 \

--master 指定主节点 \

上传jar所在的位置 \

数据输入路径 \

数据输出路径 --> 没有可以不写

#### 2.5.4 Schema合并

像ProtocolBuffer、Avro和Thrift那样，Parquet也支持Schema evolution（Schema演变）。用户可以先定义一个简单的Schema，然后逐渐的向Schema中增加列描述。通过这种方式，用户可以获取多个有不同Schema但相互兼容的Parquet文件。现在Parquet数据源能自动检测这种情况，并合并这些文件的schemas。

因为Schema合并是一个高消耗的操作，在大多数情况下并不需要，所以Spark SQL从1.5.0开始默认关闭了该功能。可以通过下面两种方式开启该功能：

当数据源为Parquet文件时，将数据源选项mergeSchema设置为true

设置全局SQL选项spark.sql.parquet.mergeSchema为true

代码实例如下：

import spark.implicits.\_  
  
// 创建一个DataFrame并写入到HDFS分区目录中  
val df1 = sc.makeRDD(1 to 5).map(i => (i, i \* 2)).toDF("single", "double")  
df1.write.parquet("hdfs://master01:9000/data/test\_table/key=1")  
  
// 创建另一个DataFrame并写入到HDFS分区目录中  
val df2 = sc.makeRDD(6 to 10).map(i => (i, i \* 3)).toDF("single", "triple")  
df2.write.parquet("hdfs://master01:9000/data/test\_table/key=2")  
  
// 读取分区表  
val df3 = spark.read.option("mergeSchema", "true").parquet("hdfs://master01:9000/data/test\_table")  
df3.printSchema()  
  
// 最终合并后的Schema信息  
// |-- single: int (nullable = true)  
// |-- double: int (nullable = true)  
// |-- triple: int (nullable = true)  
// |-- key : int (nullable = true)

### 2.6 SparkSQL自定义函数

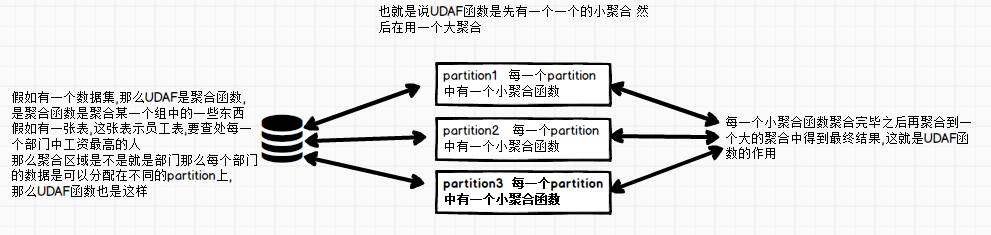
#### 2.6.1 UDF函数

用户自定义函数

import org.apache.spark.sql.{DataFrame, SparkSession}  
import org.apache.spark.{SparkConf}  
  
object SparkSQL {  
 def main(args:Array[String]):Unit = {  
 //创建SparkConf()并设置App名称  
 val conf = new SparkConf().setAppName("SparkSQLDemo").setMaster("local")  
 val spark = SparkSession.builder().config(conf).getOrCreate()  
 val df: DataFrame = spark.read.json("dir/people.json")  
 //注册函数,在整个应用中可以使用  
 val addName = spark.udf.register("addName", (x: String) => "Name:" + x)  
 df.createOrReplaceTempView("people")  
 spark.sql("Select addName(name), age from people").show()  
 spark.stop()  
 }  
}

#### 2.6.2 UDAF函数

用户自定义聚合函数



##### UDAF函数支持DataFrame(弱类型)

过继承UserDefinedAggregateFunction来实现用户自定义聚合函数。下面展示一个求平均工资的自定义聚合函数。

ps:弱类型指的是在编译阶段是无法确定数据类型的,而是在运行阶段才能创建类型

import org.apache.spark.sql.expressions.MutableAggregationBuffer  
import org.apache.spark.sql.expressions.UserDefinedAggregateFunction  
import org.apache.spark.sql.types.\_  
import org.apache.spark.sql.Row  
import org.apache.spark.sql.SparkSession  
//自定义UDAF函数  
class MyAverage extends UserDefinedAggregateFunction {  
 // 输入数据  
 def inputSchema: StructType = StructType(List(StructField("Salary",DoubleType,true)))  
 // 每一个分区中的 共享变量 存储记录的值  
 def bufferSchema: StructType = {  
 // 工资的总和 工资的总数  
 StructType(StructField("sum", DoubleType):: StructField("count", DoubleType) :: Nil)  
 }  
 // 返回值的数据类型表示UDAF函数的输出类型  
 def dataType: DataType = DoubleType  
  
 //如果有相同的输入,那么是否UDAF函数有相同的输出,有true 否则false  
 //UDAF函数中如果输入的数据掺杂着时间,不同时间得到的结果可能是不一样的所以这个值可以设置为false  
 //若不掺杂时间,这个值可以输入为true  
 def deterministic: Boolean = true  
  
 // 初始化对Buffer中的属性初始化即初始化分区中每一个共享变量  
 def initialize(buffer: MutableAggregationBuffer): Unit = {  
 // 存工资的总额  
 buffer(0) = 0.0//取得就是sum  
 // 存工资的个数  
 buffer(1) = 0.0//取得就是count  
 }  
 // 相同Execute间的数据合并,合并小聚合中的数据即每一个分区中的每一条数据聚合的时候需要调用的方法  
 /\*  
 第一个参数buffer还是共享变量  
 第二个参数是一行数据即读取到的数据是以一行来看待的  
 \*/  
 def update(buffer: MutableAggregationBuffer, input: Row): Unit = {  
 if (!input.isNullAt(0)) {  
 //获取这一行中的工资,然后将工资添加到该方法中  
 buffer(0) = buffer.getDouble(0) + input.getDouble(0)  
 //将工资的个数进行加1操作最终是为了计算所有的工资的个数  
 buffer(1) = buffer.getDouble(1) + 1  
 }  
 }  
 // 不同Execute间的数据合并,合并大数据中的数即将每一个区分的输出合并形成最后的数据  
 def merge(buffer1: MutableAggregationBuffer, buffer2: Row): Unit = {  
 //合并总的工资  
 buffer1(0) = buffer1.getDouble(0) + buffer2.getDouble(0)  
 //合并总的工资个数  
 buffer1(1) = buffer1.getDouble(1) + buffer2.getDouble(1)  
 }  
 // 计算最终结果  
 def evaluate(buffer: Row): Double = buffer.getDouble(0) / buffer.getDouble(1)  
  
  
}  
  
object MyAverage{  
 def main(args: Array[String]): Unit = {  
 val spark = SparkSession.builder().appName("MyAverage").master("local[\*]").getOrCreate()  
 // 注册函数  
 spark.udf.register("myAverage",new MyAverage)  
  
 val df = spark.read.json("dir/employees.json")  
 df.createOrReplaceTempView("employees")  
 df.show()  
 //虽然没有使用groupby那么会将整个数据作为一个组  
 val result = spark.sql("SELECT myAverage(salary) as average\_salary FROM employees")  
 result.show()  
  
 }  
  
}

##### UDAF函数支持DataSet(强类型)

通过继承Aggregator来实现强类型自定义聚合函数，同样是求平均工资

ps:在编译阶段就确定了数据类型

package Day01  
  
import org.apache.spark.sql.expressions.{Aggregator}  
import org.apache.spark.sql.Encoder  
import org.apache.spark.sql.Encoders  
import org.apache.spark.sql.SparkSession  
//自定义UDAF函数  
// 既然是强类型，可能有case类  
case class Employee(name: String, salary: Double)  
case class Average(var sum: Double, var count: Double)  
//依次配置输入,共享变量,输出的类型,需要使用到case class  
class MyAverage extends Aggregator[Employee, Average, Double] {  
 // 初始化方法 初始化每一个分区中的 共享变量即定义一个数据结构，保存工资总数和工资总个数，初始都为0  
 def zero: Average = Average(0.0, 0.0)  
 //每一个分区中的每一条数据聚合的时候需要调用该方法  
 def reduce(buffer: Average, employee: Employee): Average = {  
 buffer.sum += employee.salary  
 buffer.count += 1  
 buffer  
 }  
 //将每一个分区的输出 合并 形成最后的数据  
 def merge(b1: Average, b2: Average): Average = {  
 b1.sum += b2.sum  
 b1.count += b2.count  
 b1  
 }  
 // 给出计算结果  
 def finish(reduction: Average): Double = reduction.sum / reduction.count  
 // 设定之间值类型的编码器，要转换成case类  
 // Encoders.product是进行scala元组和case类转换的编码器  
 def bufferEncoder: Encoder[Average] = Encoders.product  
 // 设定最终输出值的编码器  
 def outputEncoder: Encoder[Double] = Encoders.scalaDouble  
}  
  
object MyAverage{  
 def main(args: Array[String]): Unit = {  
 val spark = SparkSession.builder().appName("MyAverage").master("local[\*]").getOrCreate()  
 import spark.implicits.\_  
 val ds = spark.read.json("dir/employees.json").as[Employee]  
 ds.show()  
 val averageSalary = new MyAverage().toColumn.name("average\_salary")  
 val result = ds.select(averageSalary)  
 result.show()  
 }  
  
}

#### 2.6.3 开窗函数

说明:  
rank（）跳跃排序，有两个第二名时后边跟着的是第四名  
dense\_rank() 连续排序，有两个第二名时仍然跟着第三名  
over（）开窗函数：  
 在使用聚合函数后，会将多行变成一行，而开窗函数是将一行变成多行；  
 并且在使用聚合函数后，如果要显示其他的列必须将列加入到group by中，  
 而使用开窗函数后，可以不使用group by，直接将所有信息显示出来。  
 开窗函数适用于在每一行的最后一列添加聚合函数的结果。  
常用开窗函数：  
 1.为每条数据显示聚合信息.(聚合函数() over())  
 2.为每条数据提供分组的聚合函数结果(聚合函数() over(partition by 字段) as 别名)   
 --按照字段分组，分组后进行计算  
 3.与排名函数一起使用(row number() over(order by 字段) as 别名)  
常用分析函数：（最常用的应该是1.2.3 的排序）  
 1、row\_number() over(partition by ... order by ...)  
 2、rank() over(partition by ... order by ...)  
 3、dense\_rank() over(partition by ... order by ...)  
 4、count() over(partition by ... order by ...)  
 5、max() over(partition by ... order by ...)  
 6、min() over(partition by ... order by ...)  
 7、sum() over(partition by ... order by ...)  
 8、avg() over(partition by ... order by ...)  
 9、first\_value() over(partition by ... order by ...)  
 10、last\_value() over(partition by ... order by ...)  
 11、lag() over(partition by ... order by ...)  
 12、lead() over(partition by ... order by ...)  
lag 和lead 可以 获取结果集中，按一定排序所排列的当前行的上下相邻若干offset 的某个行的某个列(不用结果集的自关联）；  
lag ，lead 分别是向前，向后；  
lag 和lead 有三个参数，第一个参数是列名，第二个参数是偏移的offset，第三个参数是 超出记录窗口时的默认值  
import org.apache.spark.sql.SparkSession  
  
//开窗窗函数  
object MyAverage {  
  
 def main(args: Array[String]): Unit = {  
 val spark = SparkSession.builder().appName("MyAverage").master("local[\*]").getOrCreate()  
 import spark.implicits.\_  
 val df = spark.read.json("dir/Score.json")  
 df.createOrReplaceTempView("score")  
 df.show()  
   
 println("/\*\*\*\*\*\*\*\*\*\*\*\*\*\* 求每个班最高成绩学生的信息（groupBY） \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/")  
 //先写的 进行查询 这样可以进行班级分组并得到分数  
 spark.sql("select class, max(score) max from score group by class").show()  
 //这句话在集合Sparkcore想一样  
 /\*这句话select class, max(score) max from score group by class  
 相当于key是class 然后执行了group by 求了一个max(score) 得到一个RDD然后和原有RDD执行了一个Join  
 直白一些(class相当于key,score相当于value) 然后执行groupbyKey --> 生成了(class,max) 这个RDD  
 然后将(class,max)转了一下 --> (class\_max,1)后面的数无所谓  
 然就相当于将数据中也做了一个(class\_max,score) -->和之前的RDD -->进行join --> 在进行map  
 \*/  
 spark.sql("select a.name, b.class, b.max from score a, " +  
 "(select class, max(score) max from score group by class) as b " +  
 "where a.score = b.max").show()  
 //添加一个函数可以对表中动态添加一列即 以对表中的分数添加排序   
 //这里的over就是开窗函数 , row\_number是分析函数(排名函数)  
 spark.sql("select name,class,score,row\_number() over(partition by class order by score desc) rank from score").show()  
 println(" /\*\*\*\*\*\*\* 计算结果的表 \*\*\*\*\*\*\*")  
 spark.sql("select \* from " +  
 "( select name,class,score,rank() over(partition by class order by score desc) rank from score) " +  
 "as t " +  
 "where t.rank=1").show()  
  
 println("//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 求每个班最高成绩学生的信息 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/")  
 println(" /\*\*\*\*\*\*\* 开窗函数的表 \*\*\*\*\*\*\*\*/")  
 spark.sql("select name,class,score, rank() over(partition by class order by score desc) rank from score").show()  
  
 println(" /\*\*\*\*\*\*\* 计算结果的表 \*\*\*\*\*\*\*")  
 spark.sql("select \* from " +  
 "( select name,class,score,rank() over(partition by class order by score desc) rank from score) " +  
 "as t " +  
 "where t.rank=1").show()  
 }  
}

## 第三章 SparkSQL集成Hive

Apache Hive是Hadoop上的SQL引擎，Spark SQL编译时可以包含Hive支持，也可以不包含。包含Hive支持的Spark SQL可以支持Hive表访问、UDF(用户自定义函数)以及 Hive 查询语言(HiveQL/HQL)等。需要强调的 一点是，如果要在Spark SQL中包含Hive的库，并不需要事先安装Hive。

ps：一般来说，最好还是在编译Spark SQL时引入Hive支持，这样就可以使用这些特性了。

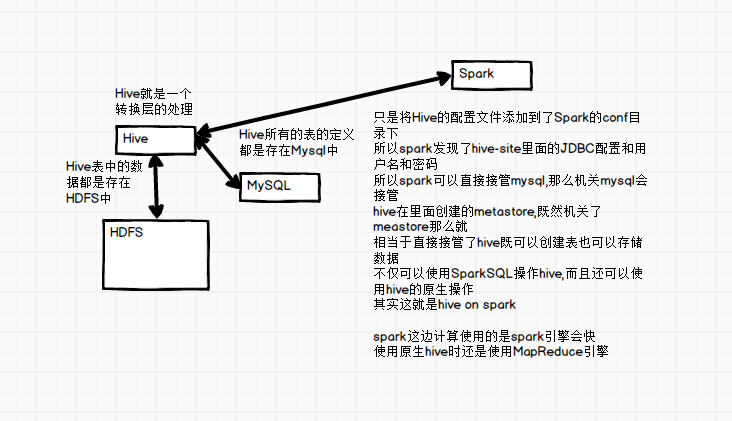
### 3.1 使用内置hive

ps:版本为1.2.1

ps:需要注意内置hive是非常容易出现问题的  
1.先启动集群/opt/software/spark-2.2.0-bin-hadoop2.7/sbin/start-all.sh  
2.进入到spark-shell模式/opt/software/spark-2.2.0-bin-hadoop2.7/bin/spark-shell --master spark://hadoop01:7077  
3.在spark-shell下操作hive  
spark.sql("show tables").show 查询所有hive的表  
spark.sql("CREATE TABLE IF NOT EXISTS src (key INT,value STRING)") 创建表  
spark.sql("LOAD DATA LOCAL INPATH '/opt/software/spark-2.2.0-bin-hadoop2.7/examples/src/main/resources/kv1.txt' INTO TABLE src") 添加数据  
spark.sql("SELECT \* FROM src").show 查询表中数据  
会出现一个问题FileNotFoundException 没有找到文件  
通过在主节点和从节点查询可以发现,主节点存在spark-warehouse目录 目录中是存在数据的  
但是在从节点中没有这个文件夹,所以此时将文件夹分发到从节点  
scp -r ./spark-warehouse/ root@hadoop02:$PWD   
再次执行查询  
ps:这样表面看查询是没什么问题了,但是实际问题在于是讲master节点上的数据分发到从节点上的,那么不可能说每次操作有了数据都执行拷贝操作,所以此时就需要使用HDFS来进行存储数据了  
所以先将所有从节点上的spark-warehouse删除掉  
删除完成后加工主节点上的spark-warehouse和metastor\_db删除掉  
然后在启动集群的时候添加一个命令  
 --conf spark.sql.warehouse.dir=hdfs://hadoop01:8020/spark\_warehouse  
 此方法只做一次启动即可 后续再启动集群的时候就无需添加这个命了 因为记录在metastore\_db中了  
 ps:spark-sql中可以直接使用SQL语句操作

### 3.2 集成外部hive

1.将Hive中的hive-site.xml软连接到Spark安装目录下的conf目录下。[主节点有即可]  
ln -s /opt/software/apache-hive-1.2.1-bin/conf/hive-site.xml /opt/software/spark-2.2.0-bin-hadoop2.7/conf/hive-site.xml   
2.打开spark shell，注意带上访问Hive元数据库的JDBC客户端  
 将mysql驱动jar包拷贝到spark的bin目录下  
./spark-shell --master spark://hadoop01:7077 --jars mysql-connector-java-5.1.36.jar  
  
ps:做完外部hive链接需要注意,因为hive-site.xml文件是在Spark的conf目录下,若直接启动spark-shell无论是单机版还是集群版都会出现报错 Error creating transactional connection factory 原因在于,启动时会加载hive-site.xml文件,所以必须添加jar路径, 为了以后使用建议删除软连接,需要的时候在做外部hive的连接  
删除软连接方式:  
rm -rf 软连接方式



总结:

若要把Spark SQL连接到一个部署好的Hive上，你必须把hive-site.xml复制到 Spark的配置文件目录中($SPARK*HOME/conf)。即使没有部署好Hive，Spark SQL也可以运行。 需要注意的是，如果你没有部署好Hive，Spark SQL会在当前的工作目录中创建出自己的Hive 元数据仓库，叫作 metastore*db。此外，如果你尝试使用 HiveQL 中的 CREATE TABLE (并非 CREATE EXTERNAL TABLE)语句来创建表，这些表会被放在你默认的文件系统中的 /user/hive/warehouse 目录中(如果你的 classpath 中有配好的 hdfs-site.xml，默认的文件系统就是 HDFS，否则就是本地文件系统)。

### 3.3 通过代码操作

ps:需要有Hadoop本地环境

需要添加依赖

<dependency>  
 <groupId>org.apache.spark</groupId>  
 <artifactId>spark-hive\_2.11</artifactId>  
 <version>2.2.0</version>  
</dependency>  
<dependency>  
 <groupId>org.apache.hive</groupId>  
 <artifactId>hive-exec</artifactId>  
 <version>1.2.1</version>  
</dependency>

import org.apache.spark.sql.{Row, SparkSession}  
object HiveCode {  
 def main(args: Array[String]): Unit = {  
 val spark = SparkSession  
 .builder()  
 .appName("HiveCode")  
 .config("spark.sql.warehouse.dir", "D:\\spark-warehouse")  
 .master("local[\*]")  
 .enableHiveSupport()  
 .getOrCreate()  
 import spark.implicits.\_  
 import spark.sql  
 // sql("CREATE TABLE IF NOT EXISTS src\_1 (key INT, value STRING)")  
 sql("LOAD DATA LOCAL INPATH 'dir/kv1.txt' INTO TABLE src\_1")  
 sql("SELECT \* FROM src\_1").show()  
 sql("SELECT COUNT(\*) FROM src\_1").show()  
 val sqlDF = sql("SELECT key, value FROM src\_1 WHERE key < 10 ORDER BY key")  
 sqlDF.as("mixing").show()  
 val recordsDF = spark.createDataFrame((1 to 100).map(i => Record(i, s"val\_$i")))  
 recordsDF.createOrReplaceTempView("records")  
 sql("SELECT \* FROM records r JOIN src\_1 s ON r.key = s.key").show()  
 }  
}  
case class Record(key: Int, value: String)

ps:本地若出现"Error while instantiating 'org.apache.spark.sql.hive.HiveSessionStateBuilder和权限异常问题

在cmd命令下进入到D:\hadoop2.7.1\bin目录下执行命了即可

winutils.exe chmod 777 \tmp\hive

### 3.4 连接服务器hive

需要添加hive-site.xml 添加到IDEA中resources文件夹中

我的Hadoop集群是高可用所以我在windows下配置了C:\Windows\System32\drivers\etc路径下的

hosts文件 只要应对我的 mycluster (这一个不配置应该是可以的)

高可用一定要让第一个台集群为active 即hadoop01

Hadoop01IP地址 mycluster

Hadoop02IP地址 mycluster

<dependency>  
 <groupId>org.apache.spark</groupId>  
 <artifactId>spark-hive\_2.11</artifactId>  
 <version>2.2.0</version>  
</dependency>  
  
<dependency>  
 <groupId>org.apache.hive</groupId>  
 <artifactId>hive-exec</artifactId>  
 <version>1.2.1</version>  
</dependency>  
<!-- 是因为需要使用到mysql数据库(因为元数据在这里面)-->  
<dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 <version>5.1.38</version>  
 </dependency>

import java.io.File  
  
import org.apache.spark.sql.{Row, SparkSession}  
  
object HiveCode {  
 def main(args: Array[String]): Unit = {  
 val spark = SparkSession  
 .builder()  
 .appName("HiveCode")  
 //这句话可以不写  
 //.config("spark.sql.warehouse.dir", "hdfs://hadoop01:8020/spark\_warehouse")  
 .master("spark://10.211.55.101:7077")  
 .enableHiveSupport()  
 .getOrCreate()  
 // import spark.implicits.\_  
 //import spark.sql  
 默认是在hive的default数据库中  
 //spark.sql("CREATE TABLE IF NOT EXISTS src\_1 (key INT, value STRING)row format delimited fields terminated by ' '")  
 // spark.sql("LOAD DATA LOCAL INPATH 'dir/kv1.txt' INTO TABLE src\_1")  
 spark.sql("SELECT \* FROM src\_1").show()  
 spark.sql("SELECT COUNT(\*) FROM src\_1").show()  
 val sqlDF = spark.sql("SELECT key, value FROM src\_1 WHERE key < 10 ORDER BY key")  
 sqlDF.as("mixing").show()  
 val recordsDF = spark.createDataFrame((1 to 100).map(i => Record(i, s"val\_$i")))  
 recordsDF.createOrReplaceTempView("records")  
 spark. ql("SELECT \* FROM records r JOIN src\_1 s ON r.key = s.key").show()  
 }  
}  
case class Record(key: Int, value: String)

### 3.5 连接服务器hive[原始版本]

ps:需要添加hive-site.xml hdfs-site.xml core-site.xml

这里出现了一个异常 Could not find the uri with key [dfs.encryption.key.provider.uri] to create a KeyProvider !! 这个异常并没有找到解决问题的办法,结果自己能执行了!!!!! 谁能解决麻烦备注说明!

我的Hadoop集群是高可用所以我在windows下配置了C:\Windows\System32\drivers\etc路径下的

hosts文件 只要应对我的 mycluster (这一个不配置应该是可以的)

192.168.223.111 mycluster 192.168.223.112 mycluster

import java.io.File  
  
import org.apache.spark.sql.{Row, SparkSession}  
  
object HiveCode {  
 def main(args: Array[String]): Unit = {  
 val spark = SparkSession  
 .builder()  
 .appName("HiveCode")  
 .config("spark.sql.warehouse.dir", "hdfs://hadoop01:8020/spark\_warehouse")  
 .master("spark://hadoop01:7077")  
 .enableHiveSupport()  
 .getOrCreate()  
 import spark.implicits.\_  
 import spark.sql  
 //sql("CREATE TABLE IF NOT EXISTS src\_1 (key INT, value STRING)")  
 // sql("LOAD DATA LOCAL INPATH 'dir/kv1.txt' INTO TABLE src\_1")  
 sql("SELECT \* FROM src\_1").show()  
 sql("SELECT COUNT(\*) FROM src\_1").show()  
 val sqlDF = sql("SELECT key, value FROM src\_1 WHERE key < 10 ORDER BY key")  
 sqlDF.as("mixing").show()  
 val recordsDF = spark.createDataFrame((1 to 100).map(i => Record(i, s"val\_$i")))  
 recordsDF.createOrReplaceTempView("records")  
 sql("SELECT \* FROM records r JOIN src\_1 s ON r.key = s.key").show()  
 }  
}  
case class Record(key: Int, value: String)

## 第四章 SparkSQL的输入和输出

### 4.1 SparkSQL的输入

写法一:

SparkSession对象.read.json("路径")

SparkSession对象.read.jdbc("路径")

SparkSession对象.read.csv("路径")

SparkSession对象.read. parquet("路径") Parquet格式经常在Hadoop生态圈中被使用，它也支持Spark SQL的全部数据型

SparkSession对象.read.orc("路径")

SparkSession对象.read.table("路径")

SparkSession对象.read.text("路径")

SparkSession对象.read. textFile("路径")

写法二:

SparkSession对象.read.format("json").load("路径")

ps:若不执行format默认是parquet格式

### 4.2 SparkSQL的输出

写法一:

DataFrame或DataSet对象.write.json("路径")

DataFrame或DataSet对象.write.jdbc("路径")

DataFrame或DataSet对象.write.csv("路径")

DataFrame或DataSet对象.write.parquet("路径")

DataFrame或DataSet对象.write.orc("路径")

DataFrame或DataSet对象.write.table("路径")

DataFrame或DataSet对象.write.text("路径")

写法二:

DataFrame或DataSet对象.write.fomat("jdbc").中间可能其他的参数.save()

ps:典型的是saveMode模式 即 mode方法

|  |  |  |
| --- | --- | --- |
| **Scala/Java** | **Any Language** | **Meaning** |
| **SaveMode.ErrorIfExists(default)** | "error"(default) | 如果文件存在，则报错 |
| **SaveMode.Append** | "append" | 追加 |
| **SaveMode.Overwrite** | "overwrite" | 覆写 |
| **SaveMode.Ignore** | "ignore" | 数据存在，则忽略 |

若不执行format默认是parquet格式

Parquet是一种流行的列式存储格式，可以高效地存储具有嵌套字段的记录。Parquet格式经常在Hadoop生态圈中被使用，它也支持Spark SQL的全部数据类型。Spark SQL 提供了直接读取和存储 Parquet 格式文件的方法。

ORC文件格式是一种Hadoop生态圈中的列式存储格式，最初产生自Apache Hive，用于降低Hadoop数据存储空间和加速Hive查询速度。和Parquet类似，列式存储格式,目前也被Spark SQL所使用

两者都会数据数据进行压缩

* 原始Text格式，未压缩 : 38.1 G
* ORC格式，默认压缩（ZLIB）,一共1800+个分区 : 11.5 G
* Parquet格式，默认压缩（Snappy），一共1800+个分区 ： 14.8 G

问题:使用Json或csv将数据写出文件的时候会根据默认根据的是

spark.sql.shuffle.partitions 进行的分区,默认应该是200 可以在创建SparkSession对象的使用时从  
config("spark.sql.shuffle.partitions","5")进行限制  
也可以在最终输出的使用repartition(1)结果

如果使用read.json出现这个问题的时候,读取json时间格式化问题

Exception in thread "main" java.lang.IllegalArgumentException: Illegal pattern component: XXX

原因应在在于读pom.xml文件加载的时候导入问题,但是具体哪个包引起的暂时不知,但是可以添加一个jar包平衡

<dependency>  
 <groupId>org.apache.commons</groupId>  
 <artifactId>commons-lang3</artifactId>  
 <version>3.5</version>  
</dependency>

### 4.3 JDBC操作

#### 4.3.1 从MySQL中将数据获取

pom.xml文件中配置JDBC连接驱动

<dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 <version>5.1.38</version>  
 </dependency>

import java.util.Properties  
  
import org.apache.spark.sql.SparkSession  
  
object SparkSQLAndMySQL {  
 def main(args: Array[String]): Unit = {  
 val spark = SparkSession.builder().appName("SparkSQLAndMySQL").master("local").getOrCreate()  
 //According to MySQL 5.5.45+, 5.6.26+ and 5.7.6+ requirements SSL 访问这个些版本的数据库时会出现这个警告需要指明是否进行SSL连接  
 //如果只有数据库mysql?useSSL=false就以?连接 mysql?useUnicode=true&characterEncoding=UTF-8&useSSL=false 如果有多个连接就用&  
 //读取方式一  
 val connectionProperties = new Properties()  
 connectionProperties.put("user", "root")  
 connectionProperties.put("password", "123456")  
 val jdbcDF = spark.read.jdbc("jdbc:mysql://localhost:3306/mysql?useSSL=false", "user",connectionProperties)  
 jdbcDF.show();  
  
 //方式二  
 val jdbcDF2 = spark.read.format("jdbc")  
 .option("url","jdbc:mysql://localhost:3306/mysql?useSSL=false").option("dbtable","user").option("user","root").option("password","123456").load()  
 jdbcDF2.show()  
  
 spark.stop()  
  
 }  
}

#### 4.3.2 将数据写入到MySQL中

import java.util.Properties  
  
import org.apache.spark.sql.{DataFrame, SparkSession}  
  
object SparkSQLAndMySQL {  
 def main(args: Array[String]): Unit = {  
 val spark = SparkSession.builder().appName("SparkSQLAndMySQL").master("local").getOrCreate()  
 //读取数据  
 val frame: DataFrame = spark.read.json("dir/dataSource/employees.json")  
 //写入方法一  
 val connectionProperties = new Properties()  
 connectionProperties.put("user", "root")  
 connectionProperties.put("password", "123456")  
 //表可以不存在,通过读取的数据可以直接生成表  
 frame.write.jdbc("jdbc:mysql://localhost:3306/mydb1?useSSL=false", "employees", connectionProperties)  
  
 //写出方式二  
 frame.write.format("jdbc")  
 .option("url", "jdbc:mysql://localhost:3306/mydb1?useSSL=false")  
 .option("dbtable", "employees1")  
 .option("user", "root")  
 .option("password", "123456")  
 .save()  
 //写出方式三 执行创建表的列名和数据类型 数据类型不能大写  
 frame.write  
 .option("createTableColumnTypes", "name varchar(200),salary int")  
 .jdbc("jdbc:mysql://localhost:3306/mydb1?useSSL=false", "employees2", connectionProperties)  
 spark.stop()  
 }  
}

## 第五章 性能调优

### 5.1 缓存数据至内存

Spark SQL可以通过调用sqlContext.cacheTable("tableName") 或者dataFrame.cache()，将表用一种柱状格式（ an in­memory columnar format）缓存至内存中。然后Spark SQL在执行查询任务时，只需扫描必需的列，从而以减少扫描数据量、提高性能。通过缓存数据，Spark SQL还可以自动调节压缩，从而达到最小化内存使用率和降低GC压力的目的。调用sqlContext.uncacheTable("tableName")可将缓存的数据移出内存。

可通过两种配置方式开启缓存数据功能：

* 使用SQLContext的setConf方法
* 执行SQL命令 SET key=value

|  |  |  |
| --- | --- | --- |
| Property Name | Default | Meaning |
| spark.sql.inMemoryColumnarStorage.compressed | true | 如果假如设置为true，SparkSql会根据统计信息自动的为每个列选择压缩方式进行压缩。 |
| spark.sql.inMemoryColumnarStorage.batchSize | 10000 | 控制列缓存的批量大小。批次大有助于改善内存使用和压缩，但是缓存数据会有OOM的风险。 |

### 5.2 调优参数

可以通过配置下表中的参数调节Spark SQL的性能。

|  |  |  |
| --- | --- | --- |
| Property Name | Default | Meaning |
| spark.sql.files.maxPartitionBytes | 134217728 (128 MB) | 获取数据到分区中的最大字节数。 |
| spark.sql.files.openCostInBytes | 4194304 (4 MB) | 该参数默认4M，表示小于4M的小文件会合并到一个分区中，用于减小小文件，防止太多单个小文件占一个分区情况。 |
| spark.sql.broadcastTimeout | 300 | 广播等待超时时间，单位秒。 |
| spark.sql.autoBroadcastJoinThreshold | 10485760 (10 MB) | 最大广播表的大小。设置为-1可以禁止该功能。当前统计信息仅支持Hive Metastore表。 |
| spark.sql.shuffle.partitions | 200 | 设置shuffle分区数，默认200。 |

## 第六章 JDBC/ODBC服务器

Spark SQL也提供JDBC连接支持，Spark SQL的JDBC服务器与Hive中的HiveServer2相一致。由于使用了Thrift通信协议，它也被称为“Thrift server”。 服务器可以通过 Spark 目录中的 sbin/start-thriftserver.sh 启动。

./sbin/start-thriftserver.sh

sbin/start-thriftserver.sh 脚本接收的参数选项大多与 spark-submit 相同。默认情况下，服务器会在 localhost:10000 上进行监听，我们可以通过环境变量(HIVE*SERVER2*THRIFT*PORT 和 HIVE*SERVER2*THRIFT*BIND\_HOST)修改这些设置，也可以通过 Hive配置选项(hive. server2.thrift.port 和 hive.server2.thrift.bind.host)来修改。你也可以通过命令行参 数--hiveconf property=value来设置Hive选项。

./sbin/start-thriftserver.sh \  
--hiveconf hive.server2.thrift.port=<listening-port> \  
--hiveconf hive.server2.thrift.bind.host=<listening-host> \  
--master <master-uri>  
...

在 Beeline 客户端中，你可以使用标准的 HiveQL 命令来创建和查询Hive的数据表。

beeline> !connect jdbc:hive2://localhost:10000

## 第七章 Spark SQL 的运行原理

### 7.1 Spark SQL运行架构

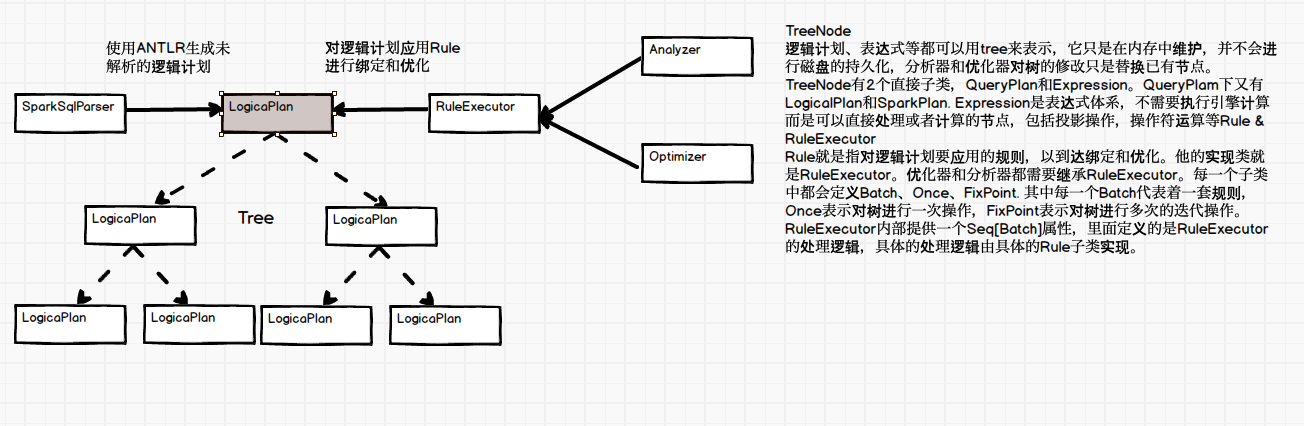
Spark SQL对SQL语句的处理和关系型数据库类似，即词法/语法解析、绑定、优化、执行。Spark SQL会先将SQL语句解析成一棵树，然后使用规则(Rule)对Tree进行绑定、优化等处理过程。Spark SQL由Core、Catalyst、Hive、Hive-ThriftServer四部分构成：

Core: 负责处理数据的输入和输出，如获取数据，查询结果输出成DataFrame等

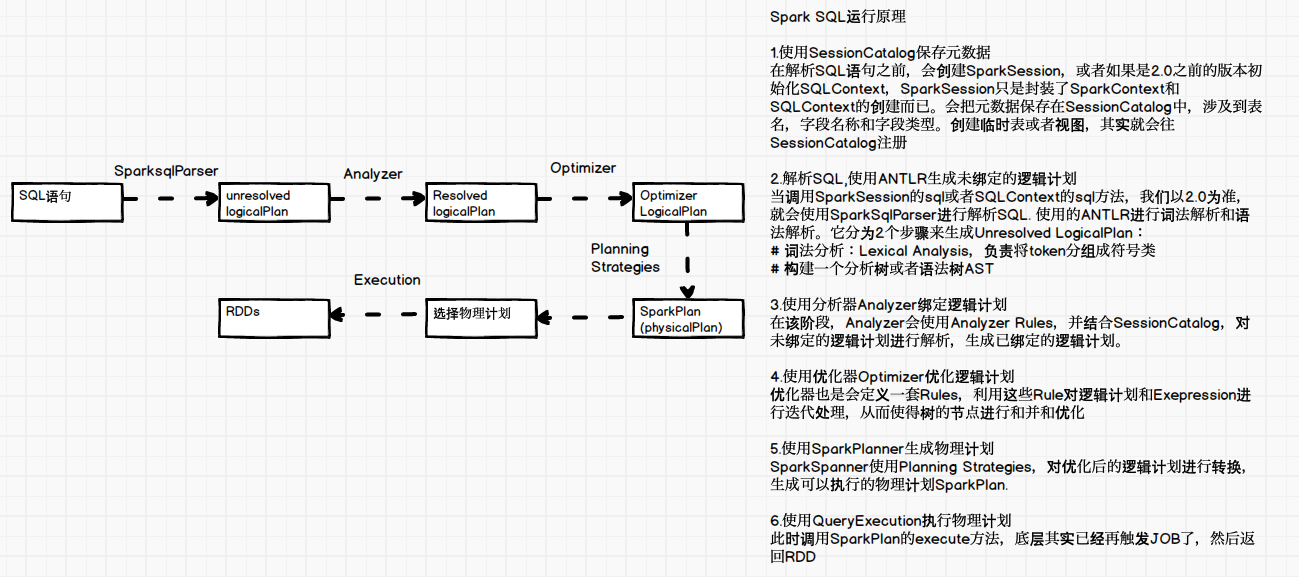
Catalyst: 负责处理整个查询过程，包括解析、绑定、优化等

Hive: 负责对Hive数据进行处理

Hive-ThriftServer: 主要用于对hive的访问



LogicalPlan(逻辑克普兰)-->逻辑计划 Analyzer(按呢来着)-->分析器 Optimizer(奥特来着)-->优化器



unresolved(安瑞绕无得) --> 没有实现 resolved(瑞绕无得) --> 实现