尚硅谷大数据技术之SparkSQL

版本：V3.0

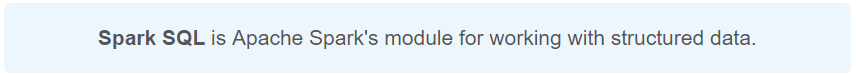


作者：尚硅谷大数据研发部

# **SparkSQL概述**

## **SparkSQL是什么**





Spark SQL是Spark用于结构化数据(structured data)处理的Spark模块。

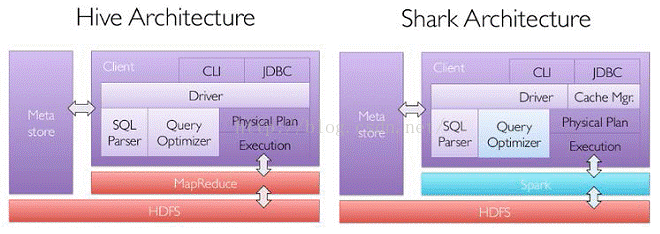
## **Hive and SparkSQL**

SparkSQL的前身是Shark，给熟悉RDBMS但又不理解MapReduce的技术人员提供快速上手的工具。

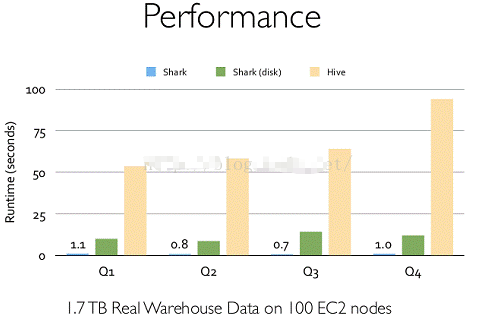
Hive是早期唯一运行在Hadoop上的SQL-on-Hadoop工具。但是MapReduce计算过程中大量的中间磁盘落地过程消耗了大量的I/O，降低的运行效率，为了提高SQL-on-Hadoop的效率，大量的SQL-on-Hadoop工具开始产生，其中表现较为突出的是：

* Drill
* Impala
* Shark

其中Shark是伯克利实验室Spark生态环境的组件之一，是基于Hive所开发的工具，它修改了下图所示的右下角的内存管理、物理计划、执行三个模块，并使之能运行在Spark引擎上。

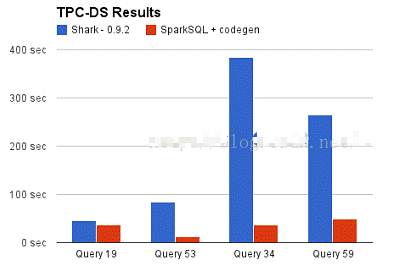


Shark的出现，使得SQL-on-Hadoop的性能比Hive有了10-100倍的提高。

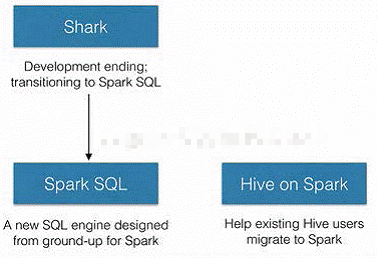


但是，随着Spark的发展，对于野心勃勃的Spark团队来说，Shark对于Hive的太多依赖（如采用Hive的语法解析器、查询优化器等等），制约了Spark的One Stack Rule Them All的既定方针，制约了Spark各个组件的相互集成，所以提出了SparkSQL项目。SparkSQL抛弃原有Shark的代码，汲取了Shark的一些优点，如内存列存储（In-Memory Columnar Storage）、Hive兼容性等，重新开发了SparkSQL代码；由于摆脱了对Hive的依赖性，SparkSQL无论在数据兼容、性能优化、组件扩展方面都得到了极大的方便，真可谓“退一步，海阔天空”。

* 数据兼容方面 SparkSQL不但兼容Hive，还可以从RDD、parquet文件、JSON文件中获取数据，未来版本甚至支持获取RDBMS数据以及cassandra等NOSQL数据；
* 性能优化方面 除了采取In-Memory Columnar Storage、byte-code generation等优化技术外、将会引进Cost Model对查询进行动态评估、获取最佳物理计划等等；
* 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。



2014年6月1日Shark项目和SparkSQL项目的主持人Reynold Xin宣布：停止对Shark的开发，团队将所有资源放SparkSQL项目上，至此，Shark的发展画上了句话，但也因此发展出两个支线：SparkSQL和Hive on Spark。



其中SparkSQL作为Spark生态的一员继续发展，而不再受限于Hive，只是兼容Hive；而Hive on Spark是一个Hive的发展计划，该计划将Spark作为Hive的底层引擎之一，也就是说，Hive将不再受限于一个引擎，可以采用Map-Reduce、Tez、Spark等引擎。

对于开发人员来讲，SparkSQL可以简化RDD的开发，提高开发效率，且执行效率非常快，所以实际工作中，基本上采用的就是SparkSQL。Spark SQL为了简化RDD的开发，提高开发效率，提供了2个编程抽象，类似Spark Core中的RDD

* DataFrame
* DataSet

## **SparkSQL特点**

### 易整合

无缝的整合了 SQL 查询和 Spark 编程



### 统一的数据访问

使用相同的方式连接不同的数据源



### 兼容Hive

在已有的仓库上直接运行 SQL 或者 HiveQL



### 标准数据连接

通过 JDBC 或者 ODBC 来连接



## **DataFrame是什么**

在Spark中，DataFrame是一种以RDD为基础的分布式数据集，类似于传统数据库中的二维表格。DataFrame与RDD的主要区别在于，前者带有schema元信息，即DataFrame所表示的二维表数据集的每一列都带有名称和类型。这使得Spark SQL得以洞察更多的结构信息，从而对藏于DataFrame背后的数据源以及作用于DataFrame之上的变换进行了针对性的优化，最终达到大幅提升运行时效率的目标。反观RDD，由于无从得知所存数据元素的具体内部结构，Spark Core只能在stage层面进行简单、通用的流水线优化。

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。

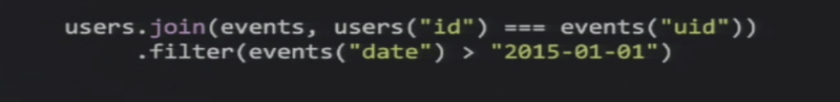


上图直观地体现了DataFrame和RDD的区别。

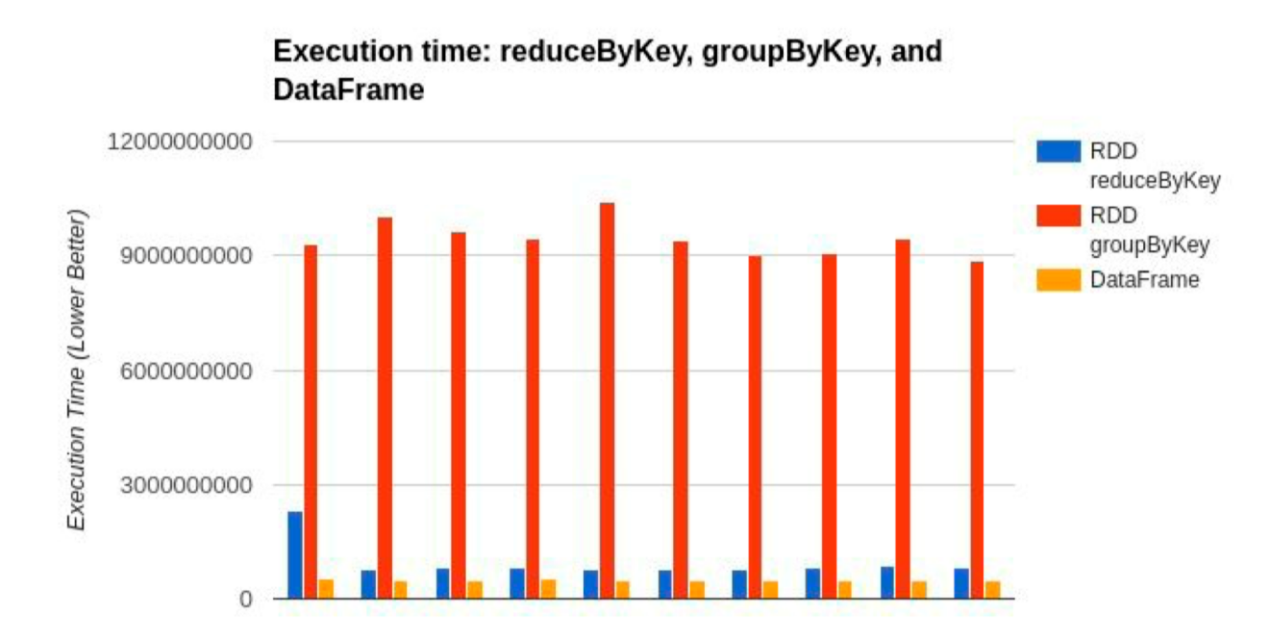
左侧的RDD[Person]虽然以Person为类型参数，但Spark框架本身不了解Person类的内部结构。而右侧的DataFrame却提供了详细的结构信息，使得 Spark SQL 可以清楚地知道该数据集中包含哪些列，每列的名称和类型各是什么。

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，但性能上比RDD要高，主要原因：优化的执行计划，即查询计划通过Spark catalyst optimiser进行优化。比如下面一个例子:



为了说明查询优化，我们来看上图展示的人口数据分析的示例。图中构造了两个DataFrame，将它们join之后又做了一次filter操作。如果原封不动地执行这个执行计划，最终的执行效率是不高的。因为join是一个代价较大的操作，也可能会产生一个较大的数据集。如果我们能将filter下推到 join下方，先对DataFrame进行过滤，再join过滤后的较小的结果集，便可以有效缩短执行时间。而Spark SQL的查询优化器正是这样做的。简而言之，逻辑查询计划优化就是一个利用基于关系代数的等价变换，将高成本的操作替换为低成本操作的过程。



## **DataSet是什么**

DataSet是分布式数据集合。DataSet是Spark 1.6中添加的一个新抽象，是DataFrame的一个扩展。它提供了RDD的优势（强类型，使用强大的lambda函数的能力）以及Spark SQL优化执行引擎的优点。DataSet也可以使用功能性的转换（操作map，flatMap，filter等等）。

* DataSet是DataFrame API的一个扩展，是SparkSQL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；
* 用样例类来对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；
* DataSet是强类型的。比如可以有DataSet[Car]，DataSet[Person]。
* 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，所有的表结构信息都用Row来表示。获取数据时需要指定顺序

# **SparkSQL核心编程**

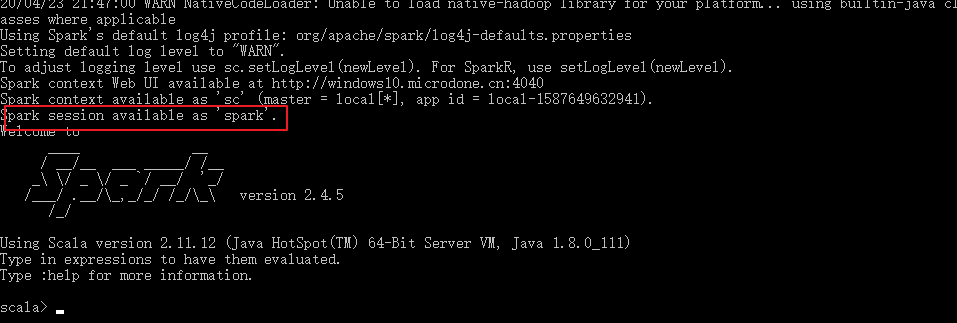
本课件重点学习如何使用 Spark SQL所提供的 DataFrame和DataSet模型进行编程.，以及了解它们之间的关系和转换，关于具体的SQL语法不是我们的重点。

## 2.1 新的起点

Spark Core中，如果想要执行应用程序，需要首先构建上下文环境对象SparkContext，Spark SQL其实可以理解为对Spark Core的一种封装，不仅仅在模型上进行了封装，上下文环境对象也进行了封装。

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SQLContex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。当我们使用 spark-shell 的时候, spark框架会自动的创建一个名称叫做spark的SparkSession对象, 就像我们以前可以自动获取到一个sc来表示SparkContext对象一样



## 2.2 DataFrame

Spark SQL的DataFrame API 允许我们使用 DataFrame 而不用必须去注册临时表或者生成 SQL 表达式。DataFrame API 既有 transformation操作也有action操作。

### 2.2.1 创建DataFrame

在Spark SQL中SparkSession是创建DataFrame和执行SQL的入口，创建DataFrame有三种方式：通过Spark的数据源进行创建；从一个存在的RDD进行转换；还可以从Hive Table进行查询返回。

1. 从Spark数据源进行创建

* 查看Spark支持创建文件的数据源格式

scala> spark.read.

csv format jdbc json load option options orc parquet schema table text textFile

* 在spark的bin/data目录（windows环境）中创建user.json文件

{"username":"zhangsan","age":20}

* 读取json文件创建DataFrame

scala> val df = spark.read.json("data/user.json")

df: org.apache.spark.sql.DataFrame = [age: bigint， username: string]

注意：如果从内存中获取数据，spark可以知道数据类型具体是什么。如果是数字，默认作为Int处理；但是从文件中读取的数字，不能确定是什么类型，所以用bigint接收，可以和Long类型转换，但是和Int不能进行转换

* 展示结果:show

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

|age|username|

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

| 20|zhangsan|

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

1. 从RDD进行转换

在后续章节中讨论

1. 从Hive Table进行查询返回

在后续章节中讨论

### 2.2.2 SQL语法

SQL语法风格是指我们查询数据的时候使用SQL语句来查询，这种风格的查询必须要有临时视图或者全局视图来辅助

1. 读取JSON文件创建DataFrame

scala> val df = spark.read.json("data/user.json")

df: org.apache.spark.sql.DataFrame = [age: bigint， username: string]

1. 对DataFrame创建一个临时表(视图)

scala> df.createOrReplaceTempView("people")

1. 通过SQL语句实现查询全表

scala> val sqlDF = spark.sql("SELECT \* FROM people")

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

1. 结果展示

scala> sqlDF.show

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

|age|username|

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

| 20|zhangsan|

| 30| lisi|

| 40| wangwu|

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

注意：普通临时表是Session范围内的，如果想应用范围内有效，可以使用全局临时表。使用全局临时表时需要全路径访问，如：global\_temp.people

1. 对于DataFrame创建一个全局表

scala> df.createGlobalTempView("people")

1. 通过SQL语句实现查询全表

scala> spark.sql("SELECT \* FROM global\_temp.people").show()

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

|age|username|

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

| 20|zhangsan|

| 30| lisi|

| 40| wangwu|

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

scala> spark.newSession().sql("SELECT \* FROM global\_temp.people").show()

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

|age|username|

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

| 20|zhangsan|

| 30| lisi|

| 40| wangwu|

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

### 2.2.3 DSL语法

DataFrame提供一个特定领域语言(domain-specific language, DSL)去管理结构化的数据。可以在 Scala, Java, Python 和 R 中使用 DSL，使用 DSL 语法风格不必去创建临时视图了

1. 创建一个DataFrame

scala> val df = spark.read.json("data/user.json")

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

1. 查看DataFrame的Schema信息

scala> df.printSchema

root

|-- age: Long (nullable = true)

|-- username: string (nullable = true)

1. 只查看"username"列数据，

scala> df.select("username").show()

+--------+

|username|

+--------+

|zhangsan|

| lisi|

| wangwu|

+--------+

1. 查看"username"列数据以及"age+1"数据

注意:涉及到运算的时候, 每列都必须使用$, 或者采用引号表达式：单引号+字段名

scala> df.select($"username",$"age" + 1).show

scala> df.select('username, 'age + 1).show()

scala> df.select('username, 'age + 1 as "newage").show()

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

|username|(age + 1)|

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

|zhangsan| 21|

| lisi| 31|

| wangwu| 41|

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

1. 查看"age"大于"30"的数据

scala> df.filter($"age">30).show

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

|age| username|

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

| 40| wangwu|

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

1. 按照"age"分组，查看数据条数

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

+---+-----+

|age|count|

+---+-----+

| 20| 1|

| 30| 1|

| 40| 1|

+---+-----+

### 2.2.4 RDD转换为DataFrame

在IDEA中开发程序时，如果需要RDD与DF或者DS之间互相操作，那么需要引入 import spark.implicits.\_

这里的spark不是Scala中的包名，而是创建的sparkSession对象的变量名称，所以必须先创建SparkSession对象再导入。这里的spark对象不能使用var声明，因为Scala只支持val修饰的对象的引入。

spark-shell中无需导入，自动完成此操作。

scala> val idRDD = sc.textFile("data/id.txt")

scala> idRDD.toDF("id").show

+---+

| id|

+---+

| 1|

| 2|

| 3|

| 4|

+---+

实际开发中，一般通过样例类将RDD转换为DataFrame

scala> case class User(name:String, age:Int)

defined class User

scala> sc.makeRDD(List(("zhangsan",30), ("lisi",40))).map(t=>User(t.\_1, t.\_2)).toDF.show

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

| name|age|

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

|zhangsan| 30|

| lisi| 40|

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

### 2.2.5 DataFrame转换为RDD

DataFrame其实就是对RDD的封装，所以可以直接获取内部的RDD

scala> val df = sc.makeRDD(List(("zhangsan",30), ("lisi",40))).map(t=>User(t.\_1, t.\_2)).toDF

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

scala> val rdd = df.rdd

rdd: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row] = MapPartitionsRDD[46] at rdd at <console>:25

scala> val array = rdd.collect

array: Array[org.apache.spark.sql.Row] = Array([zhangsan,30], [lisi,40])

注意：此时得到的RDD存储类型为Row

scala> array(0)

res28: org.apache.spark.sql.Row = [zhangsan,30]

scala> array(0)(0)

res29: Any = zhangsan

scala> array(0).getAs[String]("name")

res30: String = zhangsan

## 2.3 DataSet

DataSet是具有强类型的数据集合，需要提供对应的类型信息。

### 2.3.1 创建DataSet

1. 使用样例类序列创建DataSet

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

defined class Person

scala> val caseClassDS = Seq(Person("zhangsan",2)).toDS()

caseClassDS: org.apache.spark.sql.Dataset[Person] = [name: string, age: Long]

scala> caseClassDS.show

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

| name|age|

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

| zhangsan| 2|

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

1. 使用基本类型的序列创建DataSet

scala> val ds = Seq(1,2,3,4,5).toDS

ds: org.apache.spark.sql.Dataset[Int] = [value: int]

scala> ds.show

+-----+

|value|

+-----+

| 1|

| 2|

| 3|

| 4|

| 5|

+-----+

注意：在实际使用的时候，很少用到把序列转换成DataSet，更多的是通过RDD来得到DataSet

### 2.3.2 RDD转换为DataSet

SparkSQL能够自动将包含有case类的RDD转换成DataSet，case类定义了table的结构，case类属性通过反射变成了表的列名。Case类可以包含诸如Seq或者Array等复杂的结构。

scala> case class User(name:String, age:Int)

defined class User

scala> sc.makeRDD(List(("zhangsan",30), ("lisi",49))).map(t=>User(t.\_1, t.\_2)).toDS

res11: org.apache.spark.sql.Dataset[User] = [name: string, age: int]

### 2.3.3 DataSet转换为RDD

DataSet其实也是对RDD的封装，所以可以直接获取内部的RDD

scala> case class User(name:String, age:Int)

defined class User

scala> sc.makeRDD(List(("zhangsan",30), ("lisi",49))).map(t=>User(t.\_1, t.\_2)).toDS

res11: org.apache.spark.sql.Dataset[User] = [name: string, age: int]

scala> val rdd = res11.rdd

rdd: org.apache.spark.rdd.RDD[User] = MapPartitionsRDD[51] at rdd at <console>:25

scala> rdd.collect

res12: Array[User] = Array(User(zhangsan,30), User(lisi,49))

## 2.4 DataFrame和DataSet转换

DataFrame其实是DataSet的特例，所以它们之间是可以互相转换的。

* DataFrame转换为DataSet

scala> case class User(name:String, age:Int)

defined class User

scala> val df = sc.makeRDD(List(("zhangsan",30), ("lisi",49))).toDF("name","age")

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

scala> val ds = df.as[User]

ds: org.apache.spark.sql.Dataset[User] = [name: string, age: int]

* DataSet转换为DataFrame

scala> val ds = df.as[User]

ds: org.apache.spark.sql.Dataset[User] = [name: string, age: int]

scala> val df = ds.toDF

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

## 2.5 RDD、DataFrame、DataSet三者的关系

在SparkSQL中Spark为我们提供了两个新的抽象，分别是DataFrame和DataSet。他们和RDD有什么区别呢？首先从版本的产生上来看：

* Spark1.0 => RDD
* Spark1.3 => DataFrame
* Spark1.6 => Dataset

如果同样的数据都给到这三个数据结构，他们分别计算之后，都会给出相同的结果。不同是的他们的执行效率和执行方式。在后期的Spark版本中，DataSet有可能会逐步取代RDD和DataFrame成为唯一的API接口。

### 2.5.1 三者的共性

* 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;
* 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;
* 三者有许多共同的函数，如filter，排序等;
* 在对DataFrame和Dataset进行操作许多操作都需要这个包:import spark.implicits.\_（在创建好SparkSession对象后尽量直接导入）
* 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
* 三者都有partition的概念
* DataFrame和DataSet均可使用模式匹配获取各个字段的值和类型

### 2.5.2 三者的区别

1. RDD

* RDD一般和spark mllib同时使用
* RDD不支持sparksql操作

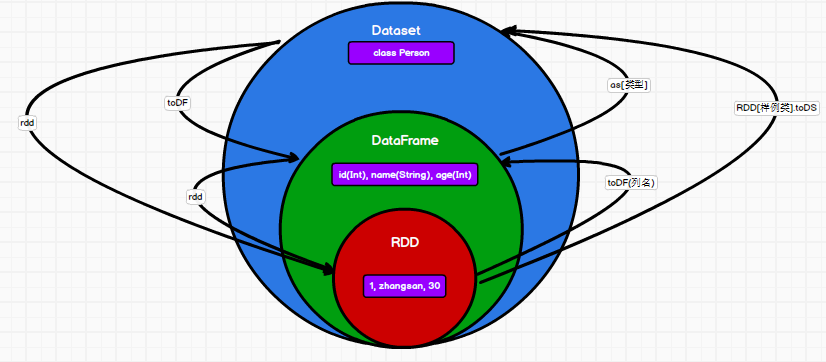
1. DataFrame

* 与RDD和Dataset不同，DataFrame每一行的类型固定为Row，每一列的值没法直接访问，只有通过解析才能获取各个字段的值
* DataFrame与DataSet一般不与 spark mllib 同时使用
* 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
* 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(后面专门讲解)

1. 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其实就是DataSet的一个特例 type DataFrame = Dataset[Row]
* 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

### 2.5.3 三者的互相转换



## 2.6 IDEA开发SparkSQL

实际开发中，都是使用IDEA进行开发的。

### 2.6.1 添加依赖

<dependency>

<groupId>org.apache.spark</groupId>

<artifactId>spark-sql\_2.12</artifactId>

<version>3.0.0</version>

</dependency>

### 2.6.2 代码实现

object SparkSQL01\_Demo {

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

//创建上下文环境配置对象

val conf: SparkConf = new SparkConf().setMaster("local[\*]").setAppName("SparkSQL01\_Demo")

//创建SparkSession对象

val spark: SparkSession = SparkSession.builder().config(conf).getOrCreate()

//RDD=>DataFrame=>DataSet转换需要引入隐式转换规则，否则无法转换

//spark不是包名，是上下文环境对象名

import spark.implicits.\_

//读取json文件 创建DataFrame {"username": "lisi","age": 18}

val df: DataFrame = spark.read.json("input/test.json")

//df.show()

//SQL风格语法

df.createOrReplaceTempView("user")

//spark.sql("select avg(age) from user").show

//DSL风格语法

//df.select("username","age").show()

//\*\*\*\*\*RDD=>DataFrame=>DataSet\*\*\*\*\*

//RDD

val rdd1: RDD[(Int, String, Int)] = spark.sparkContext.makeRDD(List((1,"zhangsan",30),(2,"lisi",28),(3,"wangwu",20)))

//DataFrame

val df1: DataFrame = rdd1.toDF("id","name","age")

//df1.show()

//DateSet

val ds1: Dataset[User] = df1.as[User]

//ds1.show()

//\*\*\*\*\*DataSet=>DataFrame=>RDD\*\*\*\*\*

//DataFrame

val df2: DataFrame = ds1.toDF()

//RDD 返回的RDD类型为Row，里面提供的getXXX方法可以获取字段值，类似jdbc处理结果集，但是索引从0开始

val rdd2: RDD[Row] = df2.rdd

//rdd2.foreach(a=>println(a.getString(1)))

//\*\*\*\*\*RDD=>DataSet\*\*\*\*\*

rdd1.map{

case (id,name,age)=>User(id,name,age)

}.toDS()

//\*\*\*\*\*DataSet=>=>RDD\*\*\*\*\*

ds1.rdd

//释放资源

spark.stop()

}

}

case class User(id:Int,name:String,age:Int)

## 2.7 用户自定义函数

用户可以通过spark.udf功能添加自定义函数，实现自定义功能。

### 2.7.1 UDF

1. 创建DataFrame

scala> val df = spark.read.json("data/user.json")

df: org.apache.spark.sql.DataFrame = [age: bigint， username: string]

1. 注册UDF

scala> spark.udf.register("addName",(x:String)=> "Name:"+x)

res9: org.apache.spark.sql.expressions.UserDefinedFunction = UserDefinedFunction(<function1>,StringType,Some(List(StringType)))

1. 创建临时表

scala> df.createOrReplaceTempView("people")

1. 应用UDF

scala> spark.sql("Select addName(name),age from people").show()

### 2.7.2 UDAF

强类型的Dataset和弱类型的DataFrame都提供了相关的聚合函数， 如 count()，countDistinct()，avg()，max()，min()。除此之外，用户可以设定自己的自定义聚合函数。通过继承UserDefinedAggregateFunction来实现用户自定义弱类型聚合函数。从Spark3.0版本后，UserDefinedAggregateFunction已经不推荐使用了。可以统一采用强类型聚合函数Aggregator

**需求：计算平均工资**

一个需求可以采用很多种不同的方法实现需求

1. **实现方式 - RDD**

val conf: SparkConf = new SparkConf().setAppName("app").setMaster("local[\*]")

val sc: SparkContext = new SparkContext(conf)

val res: (Int, Int) = sc.makeRDD(List(("zhangsan", 20), ("lisi", 30), ("wangw", 40))).map {

case (name, age) => {

(age, 1)

}

}.reduce {

(t1, t2) => {

(t1.\_1 + t2.\_1, t1.\_2 + t2.\_2)

}

}

println(res.\_1/res.\_2)

// 关闭连接

sc.stop()

1. **实现方式 - 累加器**

class MyAC extends AccumulatorV2[Int,Int]{

var sum:Int = 0

var count:Int = 0

override def isZero: Boolean = {

return sum ==0 && count == 0

}

override def copy(): AccumulatorV2[Int, Int] = {

val newMyAc = new MyAC

newMyAc.sum = this.sum

newMyAc.count = this.count

newMyAc

}

override def reset(): Unit = {

sum =0

count = 0

}

override def add(v: Int): Unit = {

sum += v

count += 1

}

override def merge(other: AccumulatorV2[Int, Int]): Unit = {

other match {

case o:MyAC=>{

sum += o.sum

count += o.count

}

case \_=>

}

}

override def value: Int = sum/count

}

1. **实现方式 - UDAF - 弱类型**

/\*

定义类继承UserDefinedAggregateFunction，并重写其中方法

\*/

class MyAveragUDAF extends UserDefinedAggregateFunction {

// 聚合函数输入参数的数据类型

def inputSchema: StructType = StructType(Array(StructField("age",IntegerType)))

// 聚合函数缓冲区中值的数据类型(age,count)

def bufferSchema: StructType = {

StructType(Array(StructField("sum",LongType),StructField("count",LongType)))

}

// 函数返回值的数据类型

def dataType: DataType = DoubleType

// 稳定性：对于相同的输入是否一直返回相同的输出。

def deterministic: Boolean = true

// 函数缓冲区初始化

def initialize(buffer: MutableAggregationBuffer): Unit = {

// 存年龄的总和

buffer(0) = 0L

// 存年龄的个数

buffer(1) = 0L

}

// 更新缓冲区中的数据

def update(buffer: MutableAggregationBuffer,input: Row): Unit = {

if (!input.isNullAt(0)) {

buffer(0) = buffer.getLong(0) + input.getInt(0)

buffer(1) = buffer.getLong(1) + 1

}

}

// 合并缓冲区

def merge(buffer1: MutableAggregationBuffer,buffer2: Row): Unit = {

buffer1(0) = buffer1.getLong(0) + buffer2.getLong(0)

buffer1(1) = buffer1.getLong(1) + buffer2.getLong(1)

}

// 计算最终结果

def evaluate(buffer: Row): Double = buffer.getLong(0).toDouble / buffer.getLong(1)

}

。。。

//创建聚合函数

var myAverage = new MyAveragUDAF

//在spark中注册聚合函数

spark.udf.register("avgAge",myAverage)

spark.sql("select avgAge(age) from user").show()

1. **实现方式 - UDAF - 强类型**

//输入数据类型

case class User01(username:String,age:Long)

//缓存类型

case class AgeBuffer(var sum:Long,var count:Long)

/\*\*

\* 定义类继承org.apache.spark.sql.expressions.Aggregator

\* 重写类中的方法

\*/

class MyAveragUDAF1 extends Aggregator[User01,AgeBuffer,Double]{

override def zero: AgeBuffer = {

AgeBuffer(0L,0L)

}

override def reduce(b: AgeBuffer, a: User01): AgeBuffer = {

b.sum = b.sum + a.age

b.count = b.count + 1

b

}

override def merge(b1: AgeBuffer, b2: AgeBuffer): AgeBuffer = {

b1.sum = b1.sum + b2.sum

b1.count = b1.count + b2.count

b1

}

override def finish(buff: AgeBuffer): Double = {

buff.sum.toDouble/buff.count

}

//DataSet默认额编解码器，用于序列化，固定写法

//自定义类型就是product自带类型根据类型选择

override def bufferEncoder: Encoder[AgeBuffer] = {

Encoders.product

}

override def outputEncoder: Encoder[Double] = {

Encoders.scalaDouble

}

}

。。。

//封装为DataSet

val ds: Dataset[User01] = df.as[User01]

//创建聚合函数

var myAgeUdaf1 = new MyAveragUDAF1

//将聚合函数转换为查询的列

val col: TypedColumn[User01, Double] = myAgeUdaf1.toColumn

//查询

ds.select(col).show()

Spark3.0版本可以采用强类型的Aggregate方式代替UserDefinedAggregateFunction

// TODO 创建UDAF函数

val udaf = new MyAvgAgeUDAF

// TODO 注册到SparkSQL中

spark.udf.register("avgAge", functions.udaf(udaf))

// TODO 在SQL中使用聚合函数

// 定义用户的自定义聚合函数

spark.sql("select avgAge(age) from user").show

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

case class Buff( var sum:Long, var cnt:Long )

// totalage, count

class MyAvgAgeUDAF extends Aggregator[Long, Buff, Double]{

override def zero: Buff = Buff(0,0)

override def reduce(b: Buff, a: Long): Buff = {

b.sum += a

b.cnt += 1

b

}

override def merge(b1: Buff, b2: Buff): Buff = {

b1.sum += b2.sum

b1.cnt += b2.cnt

b1

}

override def finish(reduction: Buff): Double = {

reduction.sum.toDouble/reduction.cnt

}

override def bufferEncoder: Encoder[Buff] = Encoders.product

override def outputEncoder: Encoder[Double] = Encoders.scalaDouble

}

## 2.8 数据的加载和保存

### 2.8.1 通用的加载和保存方式

SparkSQL提供了通用的保存数据和数据加载的方式。这里的通用指的是使用相同的API，根据不同的参数读取和保存不同格式的数据，SparkSQL默认读取和保存的文件格式为parquet

1. **加载数据**

spark.read.load 是加载数据的通用方法

scala> spark.read.

csv format jdbc json load option options orc parquet schema table text textFile

如果读取不同格式的数据，可以对不同的数据格式进行设定

scala> spark.read.format("…")[.option("…")].load("…")

* format("…")：指定加载的数据类型，包括"csv"、"jdbc"、"json"、"orc"、"parquet"和"textFile"。
* load("…")：在"csv"、"jdbc"、"json"、"orc"、"parquet"和"textFile"格式下需要传入加载数据的路径。
* option("…")：在"jdbc"格式下需要传入JDBC相应参数，url、user、password和dbtable

我们前面都是使用read API 先把文件加载到 DataFrame然后再查询，其实，我们也可以直接在文件上进行查询: **文件格式.`文件路径`**

scala>spark.sql("select \* from json.`/opt/module/data/user.json`").show

1. **保存数据**

df.write.save 是保存数据的通用方法

scala>df.write.

csv jdbc json orc parquet textFile… …

如果保存不同格式的数据，可以对不同的数据格式进行设定

scala>df.write.format("…")[.option("…")].save("…")

* format("…")：指定保存的数据类型，包括"csv"、"jdbc"、"json"、"orc"、"parquet"和"textFile"。
* save ("…")：在"csv"、"orc"、"parquet"和"textFile"格式下需要传入保存数据的路径。
* option("…")：在"jdbc"格式下需要传入JDBC相应参数，url、user、password和dbtable

保存操作可以使用 SaveMode, 用来指明如何处理数据，使用mode()方法来设置。

有一点很重要: 这些 SaveMode 都是没有加锁的, 也不是原子操作。

SaveMode是一个枚举类，其中的常量包括：

|  |  |  |
| --- | --- | --- |
| Scala/Java | Any Language | Meaning |
| SaveMode.ErrorIfExists(default) | "error"(default) | 如果文件已经存在则抛出异常 |
| SaveMode.Append | "append" | 如果文件已经存在则追加 |
| SaveMode.Overwrite | "overwrite" | 如果文件已经存在则覆盖 |
| SaveMode.Ignore | "ignore" | 如果文件已经存在则忽略 |

df.write.mode("append").json("/opt/module/data/output")

### 2.8.2 Parquet

Spark SQL的默认数据源为Parquet格式。Parquet是一种能够有效存储嵌套数据的列式存储格式。

数据源为Parquet文件时，Spark SQL可以方便的执行所有的操作，不需要使用format。修改配置项spark.sql.sources.default，可修改默认数据源格式。

1. **加载数据**

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

scala> df.show

1. **保存数据**

scala> var df = spark.read.json("/opt/module/data/input/people.json")

//保存为parquet格式

scala> df.write.mode("append").save("/opt/module/data/output")

### 2.8.3 JSON

Spark SQL 能够自动推测JSON数据集的结构，并将它加载为一个Dataset[Row]. 可以通过SparkSession.read.json()去加载JSON 文件。

注意：Spark读取的JSON文件不是传统的JSON文件，每一行都应该是一个JSON串。格式如下：

{"name":"Michael"}

{"name":"Andy"， "age":30}

[{"name":"Justin"， "age":19},{"name":"Justin"， "age":19}]

1）导入隐式转换

import spark.implicits.\_

2）加载JSON文件

val path = "/opt/module/spark-local/people.json"  
val peopleDF = spark.read.json(path)

3）创建临时表

peopleDF.createOrReplaceTempView("people")

4）数据查询

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

teenagerNamesDF.show()

+------+

| name|

+------+

|Justin|

+------+

### 2.8.4 CSV

Spark SQL可以配置CSV文件的列表信息，读取CSV文件,CSV文件的第一行设置为数据列

spark.read.format("csv").option("sep", ";").option("inferSchema", "true").option("header", "true").load("data/user.csv")

### 2.8.5 MySQL

Spark SQL可以通过JDBC从关系型数据库中读取数据的方式创建DataFrame，通过对DataFrame一系列的计算后，还可以将数据再写回关系型数据库中。如果使用spark-shell操作，可在启动shell时指定相关的数据库驱动路径或者将相关的数据库驱动放到spark的类路径下。

bin/spark-shell

--jars mysql-connector-java-5.1.27-bin.jar

我们这里只演示在Idea中通过JDBC对Mysql进行操作

1）导入依赖

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.27</version>

</dependency>

2）读取数据

val conf: SparkConf = new SparkConf().setMaster("local[\*]").setAppName("SparkSQL")

//创建SparkSession对象

val spark: SparkSession = SparkSession.builder().config(conf).getOrCreate()

import spark.implicits.\_

//方式1：通用的load方法读取

spark.read.format("jdbc")

.option("url", "jdbc:mysql://linux1:3306/spark-sql")

.option("driver", "com.mysql.jdbc.Driver")

.option("user", "root")

.option("password", "123123")

.option("dbtable", "user")

.load().show

//方式2:通用的load方法读取 参数另一种形式

spark.read.format("jdbc")

.options(Map("url"->"jdbc:mysql://linux1:3306/spark-sql?user=root&password=123123",

"dbtable"->"user","driver"->"com.mysql.jdbc.Driver")).load().show

//方式3:使用jdbc方法读取

val props: Properties = new Properties()

props.setProperty("user", "root")

props.setProperty("password", "123123")

val df: DataFrame = spark.read.jdbc("jdbc:mysql://linux1:3306/spark-sql", "user", props)

df.show

//释放资源

spark.stop()

3）写入数据

case class User2(name: String, age: Long)

。。。

val conf: SparkConf = new SparkConf().setMaster("local[\*]").setAppName("SparkSQL")

//创建SparkSession对象

val spark: SparkSession = SparkSession.builder().config(conf).getOrCreate()

import spark.implicits.\_

val rdd: RDD[User2] = spark.sparkContext.makeRDD(List(User2("lisi", 20), User2("zs", 30)))

val ds: Dataset[User2] = rdd.toDS

//方式1：通用的方式 format指定写出类型

//方式2：通过jdbc方法

val props: Properties = new Properties()

props.setProperty("user", "root")

props.setProperty("password", "123123")

ds.write.mode(SaveMode.Append).jdbc("jdbc:mysql://linux1:3306/spark-sql", "user", props)

//释放资源

spark.stop()

### 2.8.6 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。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。如果你下载的是二进制版本的 Spark，它应该已经在编译时添加了 Hive 支持。

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)。

spark-shell默认是Hive支持的；代码中是默认不支持的，需要手动指定（加一个参数即可）。

1. **内嵌的HIVE**

如果使用 Spark 内嵌的 Hive, 则什么都不用做, 直接使用即可.

Hive 的元数据存储在 derby 中, 默认仓库地址:$SPARK\_HOME/spark-warehouse

scala> spark.sql("show tables").show

。。。

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

|database|tableName|isTemporary|

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

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

scala> spark.sql("create table aa(id int)")

。。。

scala> spark.sql("show tables").show

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

|database|tableName|isTemporary|

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

| default| aa| false|

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

向表加载本地数据

scala> spark.sql("load data local inpath 'input/ids.txt' into table aa")

。。。

scala> spark.sql("select \* from aa").show

+---+

| id|

+---+

| 1|

| 2|

| 3|

| 4|

+---+

在实际使用中, 几乎没有任何人会使用内置的 Hive

**2）外部的HIVE**

如果想连接外部已经部署好的Hive，需要通过以下几个步骤：

* Spark要接管Hive需要把hive-site.xml拷贝到conf/目录下
* 把Mysql的驱动copy到jars/目录下
* 如果访问不到hdfs，则需要把core-site.xml和hdfs-site.xml拷贝到conf/目录下
* 重启spark-shell

scala> spark.sql("show tables").show

20/04/25 22:05:14 WARN ObjectStore: Failed to get database global\_temp, returning NoSuchObjectException

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

|database| tableName|isTemporary|

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

| default| emp| false|

| default|hive\_hbase\_emp\_table| false|

| default| relevance\_hbase\_emp| false|

| default| staff\_hive| false|

| default| ttt| false|

| default| user\_visit\_action| false|

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

**3）运行Spark SQL CLI**

Spark SQL CLI可以很方便的在本地运行Hive元数据服务以及从命令行执行查询任务。在Spark目录下执行如下命令启动Spark SQL CLI，直接执行SQL语句，类似一Hive窗口

bin/spark-sql

**4）运行Spark beeline**

Spark Thrift Server是Spark社区基于HiveServer2实现的一个Thrift服务。旨在无缝兼容HiveServer2。因为Spark Thrift Server的接口和协议都和HiveServer2完全一致，因此我们部署好Spark Thrift Server后，可以直接使用hive的beeline访问Spark Thrift Server执行相关语句。Spark Thrift Server的目的也只是取代HiveServer2，因此它依旧可以和Hive Metastore进行交互，获取到hive的元数据。

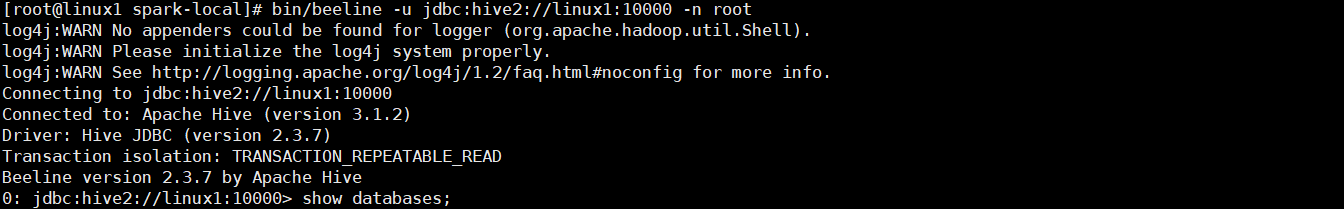
如果想连接Thrift Server，需要通过以下几个步骤：

* Spark要接管Hive需要把hive-site.xml拷贝到conf/目录下
* 把Mysql的驱动copy到jars/目录下
* 如果访问不到hdfs，则需要把core-site.xml和hdfs-site.xml拷贝到conf/目录下
* 启动Thrift Server

sbin/start-thriftserver.sh

* 使用beeline连接Thrift Server

bin/beeline -u jdbc:hive2://linux1:10000 -n root



**5）代码操作Hive**

1）导入依赖

<dependency>

<groupId>org.apache.spark</groupId>

<artifactId>spark-hive\_2.12</artifactId>

<version>3.0.0</version>

</dependency>

<dependency>

<groupId>org.apache.hive</groupId>

<artifactId>hive-exec</artifactId>

<version>1.2.1</version>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.27</version>

</dependency>

2）将hive-site.xml文件拷贝到项目的resources目录中，代码实现

//创建SparkSession

val spark: SparkSession = SparkSession

.builder()

.enableHiveSupport()

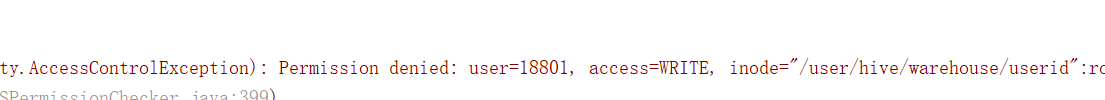
.master("local[\*]")

.appName("sql")

.getOrCreate()

注意：在开发工具中创建数据库默认是在本地仓库，通过参数修改数据库仓库的地址: config("spark.sql.warehouse.dir", "hdfs://linux1:8020/user/hive/warehouse")

如果在执行操作时，出现如下错误：



可以代码最前面增加如下代码解决：

System.setProperty("HADOOP\_USER\_NAME", "root")

此处的root改为你们自己的hadoop用户名称

# **SparkSQL项目实战**

## 3.1 数据准备

我们这次 Spark-sql 操作中所有的数据均来自 Hive，首先在 Hive 中创建表,，并导入数据。

一共有3张表： 1张用户行为表，1张城市表，1 张产品表

CREATE TABLE `user\_visit\_action`(

`date` string,

`user\_id` bigint,

`session\_id` string,

`page\_id` bigint,

`action\_time` string,

`search\_keyword` string,

`click\_category\_id` bigint,

`click\_product\_id` bigint,

`order\_category\_ids` string,

`order\_product\_ids` string,

`pay\_category\_ids` string,

`pay\_product\_ids` string,

`city\_id` bigint)

row format delimited fields terminated by '\t';

load data local inpath 'input/user\_visit\_action.txt' into table user\_visit\_action;

CREATE TABLE `product\_info`(

`product\_id` bigint,

`product\_name` string,

`extend\_info` string)

row format delimited fields terminated by '\t';

load data local inpath 'input/product\_info.txt' into table product\_info;

CREATE TABLE `city\_info`(

`city\_id` bigint,

`city\_name` string,

`area` string)

row format delimited fields terminated by '\t';

load data local inpath 'input/city\_info.txt' into table city\_info;

## 3.2 需求：各区域热门商品 Top3

### 3.2.1 需求简介

这里的热门商品是从点击量的维度来看的，计算各个区域前三大热门商品，并备注上每个商品在主要城市中的分布比例，超过两个城市用其他显示。

例如：

|  |  |  |  |
| --- | --- | --- | --- |
| **地区** | **商品名称** | **点击次数** | **城市备注** |
| **华北** | 商品A | 100000 | 北京21.2%，天津13.2%，其他65.6% |
| **华北** | 商品P | 80200 | 北京63.0%，太原10%，其他27.0% |
| **华北** | 商品M | 40000 | 北京63.0%，太原10%，其他27.0% |
| **东北** | 商品J | 92000 | 大连28%，辽宁17.0%，其他 55.0% |

### 3.2.2 需求分析

* 查询出来所有的点击记录，并与 city\_info 表连接，得到每个城市所在的地区，与 Product\_info 表连接得到产品名称
* 按照地区和商品 id 分组，统计出每个商品在每个地区的总点击次数
* 每个地区内按照点击次数降序排列
* 只取前三名
* 城市备注需要自定义 UDAF 函数

### 3.2.3 功能实现

* 连接三张表的数据，获取完整的数据（只有点击）
* 将数据根据地区，商品名称分组
* 统计商品点击次数总和,取Top3
* 实现自定义聚合函数显示备注