Spark SQL

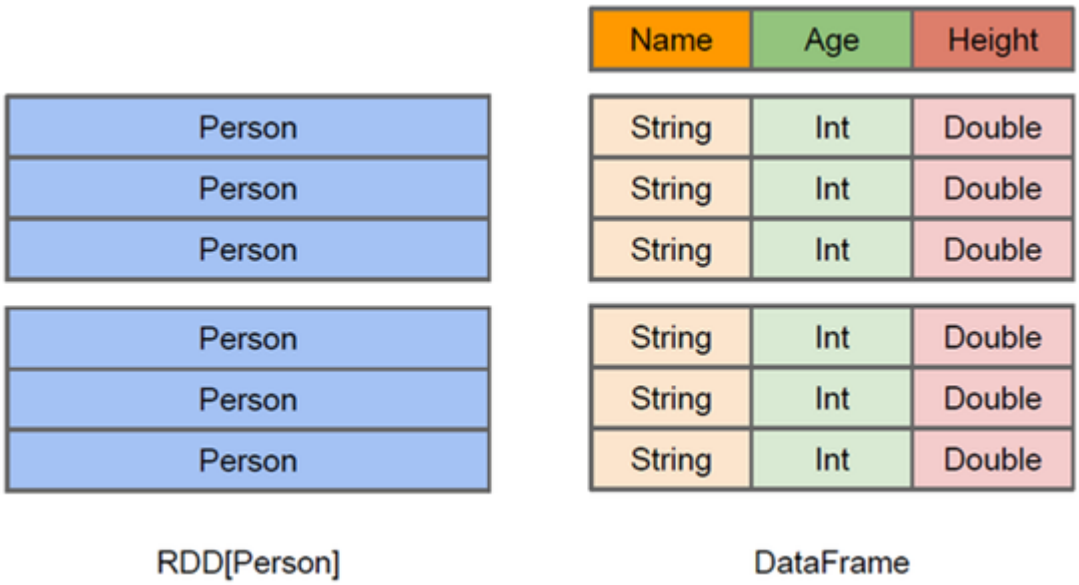
一、Spark SQL概述

Spark SQL是Spark用来处理结构化数据的一个模块，它提供了2个编程抽象：DataFrame和DataSet，并且作为分布式SQL查询引擎的作用。

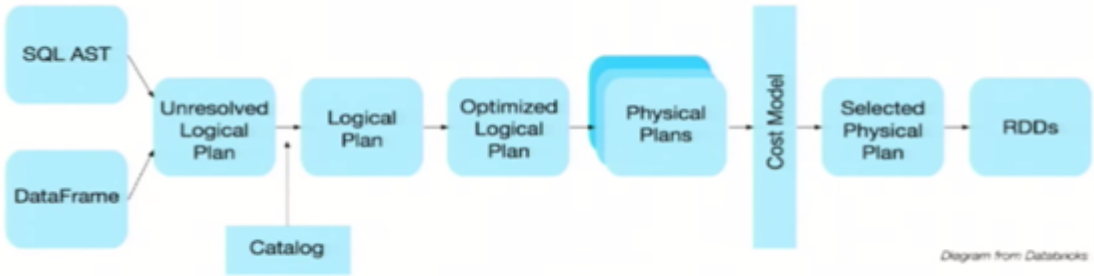
我们已经学习了Hive，它是将Hive SQL转换成MapReduce然后提交到集群上执行，大大简化了编写MapReduc的程序的复杂性，由于MapReduce这种计算模型执行效率比较慢。所有Spark SQL应运而生，它是将Spark SQL转换成RDD，然后提交到集群执行，执行效率非常快！

1.什么是DataFrame

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。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。性能上比RDD要高，主要原因：优化的执行计划：查询计划通过Spark catalyst optimiser进行优化。



2.什么是DataSet

1）是Dataframe API的一个扩展，是Spark最新的数据抽象。

2）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。

3）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。

4） 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都用Row来表示。

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

6）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。就跟JSON对象和类对象之间的类比。

二、Spark SQL编程

- 添加依赖

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-core_2.11</artifactId>
    <version>2.4.3</version>
</dependency>

<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-sql_2.11</artifactId>
    <version>2.4.3</version>
</dependency>
```

1.DataFrame

1.1 创建DataFrame

1) 创建a.json

```
{"name":"zhangsan","age":20}
{"name":"李四","age":22}
{"name":"wanguw","age":24}
```

2) 读取上边创建的文件,创建DataFrame对象

```
val spark: SparkSession =
      SparkSession.builder()
        .appName("ceshi")
        .master("local[*]")
        .getOrCreate()

val df: DataFrame = spark.read.json("F:\\datas\\a.json")
df.show()

spark.stop()
```

3) 展示结果

```
+---+--------+
|age|    name|
+---+--------+
| 20|zhangsan|
| 22|    李四|
| 24|  wanguw|
+---+--------+
```

1.2 SQL风格语法 [重点]

需求：读取json文件创建DataFrame，建立临时表，通过SQL语句查询数据

```
val df: DataFrame = spark.read.json("F:\\datas\\a.json")
df.createOrReplaceTempView("t_user")

val sqlDF: DataFrame = spark.sql("select * from t_user")
sqlDF .show()
```

展示结果

```
+---+--------+
|age|    name|
+---+--------+
| 20|zhangsan|
| 22|    李四|
| 24|  wanguw|
+---+--------+
```

1.3 DSL风格语法

需求：读取json文件创建DataFrame，通过DSL方法查询

```
import spark.implicits._

val df = spark.read.json("F:\\datas\\a.json")
df.printSchema()                    //打印Schema结构
//指定列查询
df.select($"name",$"age").show()        //select name,age from xxx
//查询大于20岁的
df.filter($"age">20).show()             //select * from xx where age > 20
//根据年龄分组，统计个数
df.groupBy("age").count().show()
```

1.4 RDD转换为DataFrame

> 注意：如果需要RDD与DF或者DS之间操作，那么都需要引入 import spark.implicits._

1) 创建a.txt

```
zhangsan    20
lisi    20
wangwu  23
zhaoliu 45
```

2) 读取上边创建的文件,创建RDD后，转换为DataFrame对象

```
import spark.implicits._        //这是一行代码,做隐式转换 spark是上边定义的变量

val rdd1: RDD[String] =  spark.sparkContext.textFile("F:\\datas\\a.txt")
val df: DataFrame = rdd1.map(v => {
    val p: Array[String] = v.split("\t")
    (p(0), p(1))
}).toDF("name", "age")
df.show()
```

1.5 DataFrame转换为RDD

```
val df = spark.read.json("F:\\datas\\a.json")
val rdd: RDD[Row] = df.rdd
rdd.collect().foreach(v=>println(v))
```

2.DataSet

Dataset是具有强类型的数据集合，需要提供对应的类型信息。

2.1 创建DataSet

1) 定义case class

```
case class Person(name:String,age:Long)
```

2) 创建List集合，转换为DataSet对象

```
import spark.implicits._

val ds: Dataset[Person] = List(Person("zhangsan",20),Person("lisi",22)).toDS()
ds.show()
```

2.2 RDD转换为DataSet

```
import spark.implicits._

val rdd1: RDD[String] = spark.sparkContext.textFile("F:\\datas\\a.txt")
rdd1.map(v=>{
      val arr: Array[String] = v.split("\t")
      Person(arr(0),arr(1).toLong)
}).toDS().show()
```

2.3 DataSet转换为RDD

```
val ds: Dataset[Person] = List(Person("zhangsan",20),Person("lisi",23)).toDS()
val rdd: RDD[Person] = ds.rdd
```

3.DataFrame和DataSet的互操作

3.1 DataFrame转换为DataSet

```
import spark.implicits._

val df: DataFrame = spark.read.json("F:\\datas\\a.json")
val ds: Dataset[Person] = df.as[Person]
ds.show()
```

3.2 DataSet转换为DataFrame

```
import spark.implicits._

val ds: Dataset[Person] = List(Person("zhangsan",20),Person("lisi",23)).toDS()
val df: DataFrame = ds.toDF()
df.show()
```

三、自定义函数

1. 单行函数

```
import spark.implicits._

val df: DataFrame = List(("zhangsan", 20, 1), ("lisi", 20, 1), ("xiaohong", 18, 0))
                    .toDF("name", "age", "sex")

df.createOrReplaceTempView("t_user")

spark.udf.register("convert_sex",(sex:Int)=>{
    sex match {
        case 0 => "女"
        case 1 => "男"
    }
})

spark.sql("select name,age,convert_sex(sex) sex from t_user").show()
```

2. 聚合函数

1) 定义求和的聚合函数

```
// 聚合函数(x1,x2)
class CustomSum extends UserDefinedAggregateFunction{   //extends UDAF

  //聚会函数输入参数的类型
  override def inputSchema: StructType = {
    new StructType().add("input1",IntegerType)
  }

  //聚会缓冲区中值的数据类型
  override def bufferSchema: StructType = {
    new StructType().add("sum",IntegerType).add("c",IntegerType)
  }

  //返回值的数据类型
  override def dataType: DataType = IntegerType

  //对于相同的输入是否一直返回相同的输出
  override def deterministic: Boolean = true

  //初始化
  override def initialize(buffer: MutableAggregationBuffer): Unit = {
    buffer(0) = 0
  }

  //将缓冲区的数据加上输入的数据，然后更新到缓冲区中
  override def update(buffer: MutableAggregationBuffer, input: Row): Unit = {
    val d1: Int = buffer.getAs[Int](0)
    val d2: Int = input.getAs[Int](0)
    buffer.update(0,d1+d2)
  }

  //合并缓冲区的数据
  override def merge(buffer1: MutableAggregationBuffer, buffer2: Row): Unit = {
    val d1: Int = buffer1.getAs[Int](0)
    val d2: Int = buffer2.getAs[Int](0)
    buffer1.update(0,d1+d2)
  }

  //返回最终结果
  override def evaluate(buffer: Row): Any = {
    buffer.getAs[Int](0)
  }
}
```

2) 测试自定义求和聚合函数

```
val df: DataFrame = List(("zhangsan", 20, 1), ("lisi", 20, 1), ("xiaohong", 18, 0))
      .toDF("name", "age", "sex")

df.createOrReplaceTempView("t_user")

spark.udf.register("customSum",new CustomSum)

spark.sql("select customSum(age) sex from t_user").show()
```

四、Load/Save

> Spark SQL的DataFrame接口支持多种数据源的操作。

1.parquet

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。

```
//保存数据
import spark.implicits._
val df: DataFrame = List((1,"zhangsan",20),(2,"lisi",23),
(3,"wangwu",23)).toDF("id","name","age")
df.write.parquet("f:\\datas\\parquet")

//读取数据
spark.read.parquet("f:\\datas\\parquet").show()
```

可以采用SaveMode执行存储操作，SaveMode定义了对数据的处理模式。需要注意的是，这些保存模式不使用任何锁定，不是原子操作。此外，当使用Overwrite方式执行时，在输出新数据之前原数据就已经被删除。
SaveMode详细介绍如下表

> 例如：df.write.mode(SaveMode.Overwrite).parquet("f:\datas\parquet")

Scala	Meaning
SaveMode.ErrorIfExists　(default)	如果文件存在，则报错
SaveMode.Append	追加
SaveMode.Overwrite	覆写
SaveMode.Ignore	数据存在，则忽略

2.json

```
import spark.implicits._

val df: DataFrame = List((1,"zhangsan",20),(2,"lisi",23),
(3,"wangwu",23)).toDF("id","name","age")
df.write.mode(SaveMode.Overwrite).json("f:\\datas\\json")

spark.read.json("f:\\datas\\json").show()
```

3.CSV

```
val df: DataFrame = List((1,"zhangsan",20),(2,"lisi",23),
(3,"wangwu",23)).toDF("id","name","age")
    df.write.mode(SaveMode.Overwrite).option("header", "true").csv("f:\\datas\\csv")

spark.read.option("header", "true").csv("f:\\datas\\csv").show()
```

4.JDBC

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。

```
val properties = new Properties()
properties.put("user", "root")
properties.put("password", "123456")

val df: DataFrame = List((1,"zhangsan",20),(2,"lisi",23),
(3,"wangwu",23)).toDF("id","name","age")
df.write.jdbc("jdbc:mysql://localhost:3306/test1", "df_user", properties)


spark.read.jdbc("jdbc:mysql://localhost:3306/test1","df_user",properties).show()
```

> 需要添加jdbc的驱动依赖

QPS: 每秒查询率(Query Per Second) ,每秒的响应请求数，也即是最大吞吐能力。 QPS = req/sec = 请求数/秒 QPS统计方式 [一般使用 http_load 进行统计] QPS = 总请求数 / (进程总数 * 请求时间) QPS: 单个进程每秒请求服务器的成功次数

峰值QPS: 原理：每天80%的访问集中在20%的时间里，这20%时间叫做峰值时间 公式：(总PV数 * 80%) / (每天秒数 * 20%) = 峰值时间每秒请求数(QPS)

PV: 访问量即Page View, 即页面浏览量或点击量，用户每次刷新即被计算一次 单台服务器每天PV计算 公式1：每天总PV = QPS * 3600 * 6 公式2：每天总PV = QPS * 3600 * 8

UV: 独立访客即Unique Visitor,访问您网站的一台电脑客户端为一个访客。00:00-24:00内相同的客户端只被计算一次 服务器数量： 机器：峰值时间每秒QPS / 单台机器的QPS = 需要的机器 机器：ceil(每天总PV / 单台服务器每天总PV)

并发数： 并发用户数是指系统可以同时承载的正常使用系统功能的用户的数量

吐吞量： 吞吐量是指系统在单位时间内处理请求的数量 响应时间（RT): 响应时间是指系统对请求作出响应的时间

例子： 每天500w PV 的在单台机器上，这台机器需要多少QPS？ 答：(5000000 * 0.8) / (86400 * 0.2) = 231 (QPS) 如果一台机器的QPS是90，需要几台机器来支持？ 答：231 / 90 = 3