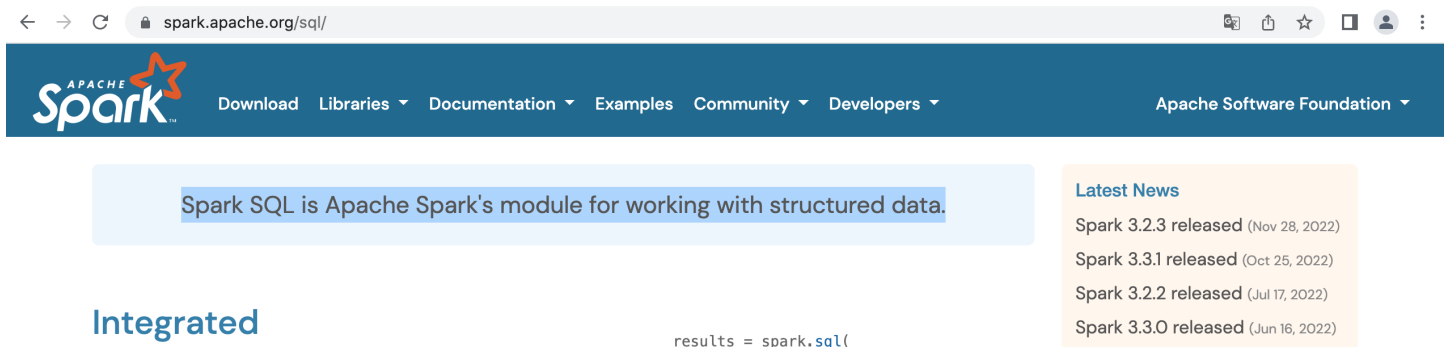


02_SparkSQL

1. Spark SQL概述

1.1 Spark SQL定义



Spark SQL是基于spark core提供的一个用来**处理结构化数据**的模块（库）

它提供了一个编程抽象叫做DataFrame/Dataset，它可以理解为一个基于RDD数据模型的更高级数据模型，带有结构化元信息（schema），**DataFrame其实就是Dataset[Row]**，Spark SQL可以将针对DataFrame/Dataset的各类SQL运算，翻译成RDD的各类算子执行计划，从而大大简化数据运算编程（请联想Hive）

```
1 object SQLDemo01 {
2
3   def main(args: Array[String]): Unit = {
4     //创建SparkSession
5     val spark: SparkSession = SparkSession.builder()
6       .appName(this.getClass.getSimpleName)
7       .master("local[*]")
8       .getOrCreate()
9     //创建RDD
10    val lines: RDD[String] = spark.sparkContext.textFile("data/boy.txt")
11    //将RDD关联了Schema，但是依然是RDD
12    val boyRDD: RDD[Boy] = lines.map(line => {
13      val fields = line.split(",")
14      Boy(fields(0), fields(1).toInt, fields(2).toDouble)
15    })
16    //强制将关联了schema信息的RDD转成DataFrame (DataSet)
17    import spark.implicits._
```

```

18     val df: DataFrame = boyRDD.toDF
19     //打印schema信息
20     df.printSchema()
21     //注册视图
22     df.createTempView("v_boy")
23     //写sql
24     val res: DataFrame = spark.sql("select name, age, fv from v_boy order by fv")
25     //打印执行计划
26     res.explain(true)
27     //触发Action
28     res.show()
29     spark.stop()
30 }
31
32 }
33 //Boy类中的成员变量，包含字段名称和类型
34 case class Boy(name: String, age: Int, fv: Double)

```

执行后的结果

```

23/01/02 08:42:35 INFO DAGScheduler: Job 0 finished: show at C02_SQLDemo2.scala:
23/01/02 08:42:35 INFO CodeGenerator: Code generated in 19.788088 ms
23/01/02 08:42:35 INFO CodeGenerator: Code generated in 53.554433 ms
+-----+---+-----+
|  name|age|    fv|
+-----+---+-----+
|xingge| 18|9999.99|
|hangge| 32|  99.99|
| taoge| 38|  99.99|
+-----+---+-----+

```

打印DataFrame的schema

```

1 //打印schema信息
2 df.printSchema()

```

```

23/01/02 09:00:36 INFO SharedState: spark.sql.warehouse.dir is not set, but hive.metastore.warehouse.dir
23/01/02 09:00:37 INFO SharedState: Warehouse path is 'hdfs://node-1.51doit.cn:9000/user/hive/warehouse'.
root
 |-- name: string (nullable = true)
 |-- age: integer (nullable = false)
 |-- fv: double (nullable = false)
23/01/02 09:00:40 INFO CodeGenerator: Code generated in 504.291275 ms

```

打印执行计划：

```
1 res.explain(true)
```

打印的执行计划如下：

```
1 root
2 |-- name: string (nullable = true)
3 |-- age: integer (nullable = false)
4 |-- fv: double (nullable = false)
5
6 == Parsed Logical Plan ==
7 'Sort ['fv DESC NULLS LAST, 'age ASC NULLS FIRST], true
8 +- 'Project ['name, 'age, 'fv]
9   +- 'UnresolvedRelation [v_boy], [], false
10
11 == Analyzed Logical Plan ==
12 name: string, age: int, fv: double
13 Sort [fv#6 DESC NULLS LAST, age#5 ASC NULLS FIRST], true
14 +- Project [name#4, age#5, fv#6]
15   +- SubqueryAlias v_boy
16     +- View ('v_boy', [name#4,age#5,fv#6])
17       +- SerializeFromObject [staticinvoke(class org.apache.spark.unsafe.type
18         +- ExternalRDD [obj#3]
19
20 == Optimized Logical Plan ==
21 Sort [fv#6 DESC NULLS LAST, age#5 ASC NULLS FIRST], true
22 +- SerializeFromObject [staticinvoke(class org.apache.spark.unsafe.types.UTF8Str
23   +- ExternalRDD [obj#3]
24
25 == Physical Plan ==
26 AdaptiveSparkPlan isFinalPlan=false
27 +- Sort [fv#6 DESC NULLS LAST, age#5 ASC NULLS FIRST], true, 0
28   +- Exchange rangepartitioning(fv#6 DESC NULLS LAST, age#5 ASC NULLS FIRST, 20
29     +- SerializeFromObject [staticinvoke(class org.apache.spark.unsafe.types.U
30       +- Scan[obj#3]
31
```

1.2 Spark SQL的特性

1.易整合

Integrated

Seamlessly mix SQL queries with Spark programs.

Spark SQL lets you query structured data inside Spark programs, using either SQL or a familiar [DataFrame API](#). Usable in Java, Scala, Python and R.

```
results = spark.sql(  
    "SELECT * FROM people")  
names = results.map(lambda p: p.name)
```

Apply functions to results of SQL queries.

Spark SQL使得在spark编程中可以如丝般顺滑地混搭SQL和算子api编程（想想都激动不是！）

2.统一的数据访问方式

Uniform data access

Connect to any data source the same way.

DataFrames and SQL provide a common way to access a variety of data sources, including Hive, Avro, Parquet, ORC, JSON, and JDBC. You can even join data across these sources.

```
spark.read.json("s3n://...")  
    .registerTempTable("json")  
results = spark.sql(  
    """SELECT *  
    FROM people  
    JOIN json ...""")
```

Query and join different data sources.

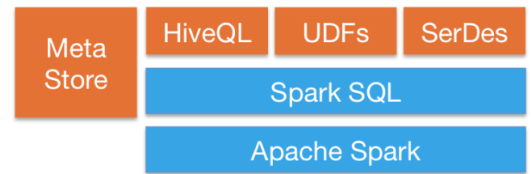
Spark SQL为各类不同数据源提供统一的访问方式，可以跨各类数据源进行愉快的join；所支持的数据源包括但不限于：Hive / Avro / CSV / Parquet / ORC / JSON / JDBC等；（简直太美好了！）

3.兼容Hive

Hive integration

Run SQL or HiveQL queries on existing warehouses.

Spark SQL supports the HiveQL syntax as well as Hive SerDes and UDFs, allowing you to access existing Hive warehouses.



Spark SQL can use existing Hive metastores, SerDes, and UDFs.

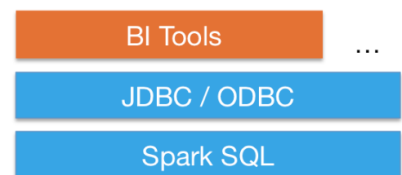
Spark SQL支持HiveQL语法及Hive的SerDes、UDFs，并允许你访问已经存在的Hive数仓数据；（难以置信的贴心！）

4.标准的数据连接

Standard connectivity

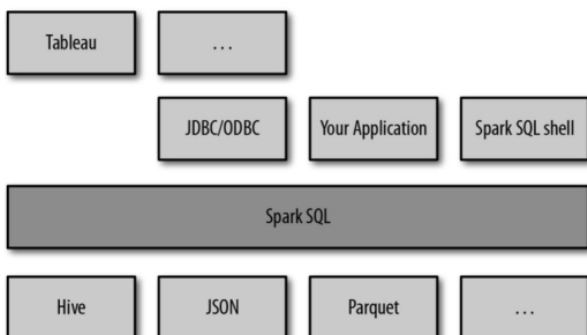
Connect through JDBC or ODBC.

A server mode provides industry standard JDBC and ODBC connectivity for business intelligence tools.



Use your existing BI tools to query big data.

Spark SQL的server模式可为各类BI工具提供行业标准的JDBC/ODBC连接，从而可以为支持标准JDBC/ODBC连接的各类工具提供无缝对接；（开发封装平台很有用哦！）



SparkSQL可以看做一个转换层，向下对接各种不同的结构化数据源，向上提供不同的数据访问方式。

2. SparkSQL快速体验

2.1 命令行使用示例

示例需求：查询大于30岁的用户

1. 启动Spark shell

[illegible]

```

10
11 scala> df.filter($"age" > 21).show
12 +----+-----+
13 |age|   name|
14 +----+-----+
15 | 30|   Andy|
16 | 39| brown|
17 | 34|jassie|
18 +----+-----+
19
20
21 scala> df.createTempView("v_user")
22
23 scala> spark.sql("select * from v_user where age > 21").show
24 +----+-----+
25 |age|   name|
26 +----+-----+
27 | 30|   Andy|
28 | 39| brown|
29 | 34|jassie|
30 +----+-----+
31

```

2.2 新的编程入口SparkSession

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SparkContext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。

```

1 import org.apache.spark.sql.SparkSession
2
3 val spark = SparkSession
4   .builder()
5   .appName("Spark SQL basic example")
6   .config("spark.some.config.option", "some-value")
7   .getOrCreate()
8
9 // 提供隐式转换支持，如 RDDs to DataFrames
10 import spark.implicits._

```

SparkSession.builder 用于创建一个SparkSession。

import spark.implicits._的引入是用于将DataFrames隐式转换成RDD，使df能够使用RDD中的方法。

如果需要Hive支持，则需要以下创建语句：

```
1 import org.apache.spark.sql.SparkSession
2
3 val spark = SparkSession
4   .builder( )
5   .appName("Spark SQL basic example")
6   .config("spark.some.config.option", "some-value")
7   .enableHiveSupport( ) //开启对hive的支持
8   .getOrCreate( )
9
10 // For implicit conversions like converting RDDs to DataFrames
11 import spark.implicits._
```

2.3 IDEA开发SparkSQL程序

IDEA中SparkSQL程序的开发方式和SparkCore类似。

添加Maven依赖：

```
1 <dependency>
2   <groupId>org.apache.spark</groupId>
3   <artifactId>spark-sql_2.12</artifactId>
4   <version>3.2.3</version>
5 </dependency>
```

程序如下：

```
1 object Demo1_HellWorld {
2
3   // 屏蔽掉WARN级别以下的日志 (INFO,DEBUG)
```



```

4  Logger.getLogger("org").setLevel(Level.WARN)
5
6  def main(args: Array[String]): Unit = {
7
8      // 创建一个sparksql的编程入口(包含sparkcontext, 也包含sqlcontext)
9      val spark: SparkSession = SparkSession
10         .builder()
11         .appName(this.getClass.getSimpleName)
12         .master("local[*]")
13         .getOrCreate()
14
15         /*val sc: SparkContext = spark.sparkContext
16         val sqlsc: SQLContext = spark.sqlContext*/
17
18         // 加载json数据文件为dataframe
19         val df: DataFrame = spark.read.json("data/people.dat")
20
21         // 打印df中的schema元信息
22         df.printSchema()
23
24         // 打印df中的数据
25         df.show(50, false)
26
27         // 在df上, 用调api方法的形式实现sql
28         df.where("age > 30").show()
29
30         // 将df注册成一个“表”(视图), 然后写原汁原味的sql
31         df.createTempView("people")
32         spark.sql("select * from people where age > 30 order by age desc").show()
33
34         spark.close()
35     }
36 }

```

3. DataFrame编程详解

3.1 创建DataFrame

在Spark SQL中SparkSession是创建DataFrames和执行SQL的入口

创建DataFrames有三种方式：

- a. 从一个已存在的**RDD**进行转换
- b. 从**JSON/Parquet/CSV/ORC**等结构化文件源创建
- c. 从**Hive/JDBC**各种外部结构化数据源（服务）创建

！ 核心要义：创建DataFrame，需要创建 “RDD + 元信息schema定义” + 执行计划

rdd来自于数据

schema则可以由开发人员定义，或者由框架从数据中推断

3.1.1 使用RDD创建DataFrame

1. 将RDD关联case class创建DataFrame

```
1 object DataFrameDemo1 {
2
3   def main(args: Array[String]): Unit = {
4
5     val conf = new SparkConf()
6       .setAppName(this.getClass.getSimpleName)
7       .setMaster("local[*]")
8     //1. SparkSession, 是对SparkContext的增强
9     val session: SparkSession = SparkSession.builder()
10      .config(conf)
11      .getOrCreate()
12
13     //2. 创建DataFrame
14     //2.1先创建RDD
15     val lines: RDD[String] = session.sparkContext.textFile("data/user.txt")
16     //2.2对数据进行整理并关联Schema
17     val tfBoy: RDD[Boy] = lines.map(line => {
18       val fields = line.split(",")
19       val name = fields(0)
20       val age = fields(1).toInt
21       val fv = fields(2).toDouble
22       Boy(name, age, fv) //字段名称，字段的类型
23     })
24
25     //2.3将RDD关联schema，将RDD转成DataFrame
26     //导入隐式转换
```

```

27     import session.implicit._
28     val df: DataFrame = tfBoy.toDF
29     //打印DataFrame的Schema信息
30     df.printSchema()
31     //3.将DataFrame注册成视图（虚拟的表）
32     df.createTempView("v_users")
33     //4.写sql（Transformation）
34     val df2: DataFrame = session.sql("select * from v_users order by fv desc, ag
35     //5.触发Action
36     df2.show()
37     //6.释放资源
38     session.stop()
39 }
40 }
41
42 case class Boy(name: String, age: Int, fv: Double)

```

2. 将RDD关联普通class创建DataFrame

```

1 object C02_DataFrameDemo2 {
2
3     def main(args: Array[String]): Unit = {
4
5         //1.创建SparkSession
6         val spark = SparkSession.builder()
7             .appName("DataFrameDemo2")
8             .master("local[*]")
9             .getOrCreate()
10
11         //2.创建RDD
12         val lines: RDD[String] = spark.sparkContext.textFile("data/user.txt")
13         //2将数据封装到普通的class中
14         val boyRDD: RDD[Boy2] = lines.map(line => {
15             val fields = line.split(",")
16             val name = fields(0)
17             val age = fields(1).toInt
18             val fv = fields(2).toDouble
19             new Boy2(name, age, fv) //字段名称，字段的类型
20         })
21         //3.将RDD和Schema进行关联
22         val df = spark.createDataFrame(boyRDD, classOf[Boy2])
23         //df.printSchema()
24         //4.使用DSL风格的API
25         import spark.implicit._
26         df.show()

```

```

27     spark.stop()
28 }
29 }
30
31 //参数前面必须有var或val
32 //必须添加给字段添加对应的getter方法，在scala中，可以@BeanProperty注解
33 class Boy2(
34     @BeanProperty
35     val name: String,
36     @BeanProperty
37     val age: Int,
38     @BeanProperty
39     val fv: Double) {
40
41 }

```

! 普通的scala class 必须在成员变量加上@BeanProperty属性，因为sparksql需要通过反射调用getter获取schema信息

3. 将RDD关联java class创建DataFrame

```

1 object SQLDemo4 {
2
3     def main(args: Array[String]): Unit = {
4
5         val spark: SparkSession = SparkSession.builder()
6             .appName(this.getClass.getSimpleName)
7             .master("local[*]")
8             .getOrCreate()
9
10        val lines = spark.sparkContext.textFile("data/boy.txt")
11        //将RDD关联的数据封装到Java的class中，但是依然是RDD
12        val jboyRDD: RDD[JBoy] = lines.map(line => {
13            val fields = line.split(",")
14            new JBoy(fields(0), fields(1).toInt, fields(2).toDouble)
15        })
16        //强制将关联了schema信息的RDD转成DataFrame
17        val df: DataFrame = spark.createDataFrame(jboyRDD, classOf[JBoy])
18        //注册视图
19        df.createTempView("v_boy")
20        //写sql
21        val df2: DataFrame = spark.sql("select name, age, fv from v_boy order by fv

```

```
22     df2.show()
23     spark.stop()
24 }
25 }
```

```
1  public class JBoy {
2
3      private String name;
4
5      private Integer age;
6
7      private Double fv;
8
9      public String getName() {
10         return name;
11     }
12
13     public Integer getAge() {
14         return age;
15     }
16
17     public Double getFv() {
18         return fv;
19     }
20
21     public JBoy(String name, Integer age, Double fv) {
22         this.name = name;
23         this.age = age;
24         this.fv = fv;
25     }
26 }
```

4. 将RDD关联Schema创建DataFrame

```
1  object SQLDemo4 {
2
3      def main(args: Array[String]): Unit = {
4
5          val spark: SparkSession = SparkSession.builder()
6              .appName(this.getClass.getSimpleName)
```

```

7      .master("local[*]")
8      .getOrCreate()
9
10     val lines = spark.sparkContext.textFile("data/boy.txt")
11     //将RDD关联了Schema, 但是依然是RDD
12     val rowRDD: RDD[Row] = lines.map(line => {
13         val fields = line.split(",")
14         Row(fields(0), fields(1).toInt, fields(2).toDouble)
15     })
16
17     val schema = StructType.apply(
18         List(
19             StructField("name", StringType),
20             StructField("age", IntegerType),
21             StructField("fv", DoubleType),
22         )
23     )
24     val df: DataFrame = spark.createDataFrame(rowRDD, schema)
25     //打印schema信息
26     //df.printSchema()
27     //注册视图
28     df.createTempView("v_boy")
29     //写sql
30     val df2: DataFrame = spark.sql("select name, age, fv from v_boy order by fv")
31     df2.show()
32     spark.stop()
33 }
34 }

```

3.1.2 从结构化文件创建DataFrame

(1)从csv文件（不带header）进行创建

csv文件内容：

```

1 1,张飞,21,北京,80.0
2 2,关羽,23,北京,82.0
3 3,赵云,20,上海,88.6
4 4,刘备,26,上海,83.0
5 5,曹操,30,深圳,90.0

```

代码示例：

```
1 val df = spark.read.csv("data_ware/demodata/stu.csv")
2 df.printSchema()
3 df.show()
```

结果如下：

```
+---+---+---+---+---+
|_c0|_c1|_c2|_c3|_c4|
+---+---+---+---+---+
|  1| 张飞| 21| 北京|80.0|
|  2| 关羽| 23| 北京|82.0|
|  3| 赵云| 20| 上海|88.6|
|  4| 刘备| 26| 上海|83.0|
|  5| 曹操| 30| 深圳|90.0|
+---+---+---+---+---+
```

```
root
 |-- _c0: string (nullable = true)
 |-- _c1: string (nullable = true)
 |-- _c2: string (nullable = true)
 |-- _c3: string (nullable = true)
 |-- _c4: string (nullable = true)
```

可以看出，框架对读取进来的csv数据，自动生成的schema中，

字段名为：_c0,_c1,.....

字段类型全为String

不一定是符合我们需求的

(2)从csv文件（不带header） 自定义Schema进行创建

// 创建DataFrame时，传入自定义的schema

// schema在api中用StructType这个类来描述，字段用StructField来描述

```
1 val schema = new StructType()
2   .add("id", DataTypes.IntegerType)
3   .add("name", DataTypes.StringType)
4   .add("age", DataTypes.IntegerType)
5   .add("city", DataTypes.StringType)
6   .add("score", DataTypes.DoubleType)
7
8 val df = spark.read.schema(schema).csv("data_ware/demodata/stu.csv")
9 df.printSchema()
10 df.show()
```

Schema信息：

```
1 root
2 |-- id: integer (nullable = true)
3 |-- name: string (nullable = true)
4 |-- age: integer (nullable = true)
5 |-- city: string (nullable = true)
6 |-- score: double (nullable = true)
```

输出数据信息：

```
1 +---+-----+---+-----+---+
2 | id|name  |age|city  |score|
3 +---+-----+---+-----+---+
4 |  1|  张飞| 21|  北京| 80.0|
5 |  2|  关羽| 23|  北京| 82.0|
6 |  3|  赵云| 20|  上海| 88.6|
7 |  4|  刘备| 26|  上海| 83.0|
```



```
8 | 5 | 曹操 | 30 | 深圳 | 90.0 |
9 +---+---+---+---+---+---+
```

(3)从csv文件（带header）进行创建

csv文件内容：

```
1 id,name,age,city,score
2 1,张飞,21,北京,80.0
3 2,关羽,23,北京,82.0
4 3,赵云,20,上海,88.6
5 4,刘备,26,上海,83.0
6 5,曹操,30,深圳,90.0
```

注意：此文件的第一行是字段描述信息，需要特别处理，否则会被当做rdd中的一行数据

代码示例：关键点设置一个header=true的参数

```
1 val df = spark.read
2   .option("header",true) //读取表头信息
3   .csv("data_ware/demodata/stu.csv")
4 df.printSchema()
5 df.show()
```

结果如下：

root

```
1 |-- id: string (nullable = true)
2 |-- name: string (nullable = true)
3 |-- age: string (nullable = true)
4 |-- city: string (nullable = true)
5 |-- score: string (nullable = true)
6
7 +---+---+---+---+---+---+
8 | id|name|age|city|score|
```

```

9  +---+---+---+---+
10 | 1 | 张飞 | 21 | 北京 | 80.0 |
11 | 2 | 关羽 | 23 | 北京 | 82.0 |
12 | 3 | 赵云 | 20 | 上海 | 88.6 |
13 | 4 | 刘备 | 26 | 上海 | 83.0 |
14 | 5 | 曹操 | 30 | 深圳 | 90.0 |
15 +---+---+---+---+
16

```

问题：虽然字段名正确指定，但是字段类型还是无法确定，默认情况下全视作String对待，当然，可以开启一个参数 `inferSchema=true` 来让框架对csv中的数据字段进行合理的类型推断

```

1 val df = spark.read
2   .option("header",true)
3   .option("inferSchema",true) //推断字段类型
4   .csv("data_ware/demodata/stu.csv")
5 df.printSchema()
6 df.show()

```

如果推断的结果不如人意，当然可以指定自定义schema

让框架自动推断schema，效率低不建议！

(4)从JSON文件进行创建

准备json数据文件

```

1 {"name":"Michael"}
2 {"name":"Andy", "age":30}
3 {"name":"Justin", "age":19}

```

代码示例

```

1 val df = spark.read.json("data_ware/demodata/people.json")
2 df.printSchema()
3 df.show()

```

(5)从Parquet文件进行创建

Parquet文件是一种列式存储文件格式，文件自带schema描述信息

准备测试数据

任意拿一个dataframe，调用write.parquet()方法即可将df保存为一个parquet文件

代码示例：

```
1 val df = spark.read.parquet("data/parquet/")
```

(6)从orc文件进行创建

代码示例

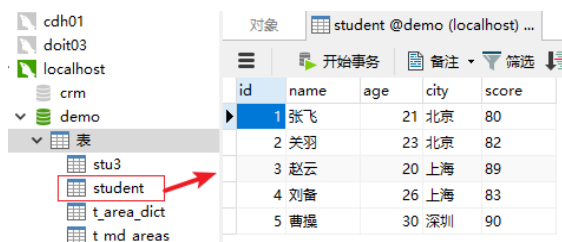
```
1 val df = spark.read.orc("data/orcfiles/")
```

3.2.3外部存储服务创建DF

(1) 从JDBC连接数据库服务器进行创建

实验准备

在一个mysql服务器中，创建一个数据库demo，创建一个表student，如下：



id	name	age	city	score
1	张飞	21	北京	80
2	关羽	23	北京	82
3	赵云	20	上海	89
4	刘备	26	上海	83
5	曹操	30	深圳	90

注：要使用jdbc连接读取数据库的数据，需要引入jdbc的驱动jar包依赖

```
1 <dependency>
2     <groupId>mysql</groupId>
```

```

3     <artifactId>mysql-connector-java</artifactId>
4     <version>8.0.30</version>
5 </dependency>

```

代码示例

```

1 val props = new Properties()
2 props.setProperty("user","root")
3 props.setProperty("password","root")
4 val df = spark.read.jdbc("jdbc:mysql://localhost:3306/demo","student",props)
5 df.show()

```

结果如下：

```

1 +---+-----+---+-----+
2 | id|name  |age|city  |score|
3 +---+-----+---+-----+
4 | 1| 张飞| 21| 北京| 80.0|
5 | 2| 关羽| 23| 北京| 82.0|
6 | 3| 赵云| 20| 上海| 88.6|
7 | 4| 刘备| 26| 上海| 83.0|
8 | 5| 曹操| 30| 深圳| 90.0|
9 +---+-----+---+-----+

```

SparkSql添加了spark-hive的依赖，并在sparkSession构造时开启enableHiveSupport后，就整合了hive的功能（通俗说，就是sparksql具备了hive的功能）；

```

</dependency>

<!-- spark整合hive的依赖，即可以读取hive的源数据库，使用hive -->
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-hive_2.12</artifactId>
  <version>${spark.version}</version>
</dependency>

<!-- 解析JSON的依赖 -->

```

Dependencies

- org.scala-lang:scala-library:2.12.15 (provided)
- org.apache.spark:spark-core_2.12:3.2.3 (provided)
- mysql:mysql-connector-java:8.0.30
- org.apache.spark:spark-hive_2.12:3.2.3
 - org.apache.spark:spark-core_2.12:3.2.3 (omitted for duplicate)
 - org.apache.spark:spark-sql_2.12:3.2.3
 - org.apache.hive:hive-common:2.3.9
 - org.apache.hive:hive-exec:core:2.3.9
 - org.apache.hive:hive-metastore:2.3.9
 - org.apache.hive:hive-serde:2.3.9
 - org.apache.hive:hive-shims:2.3.9
 - org.apache.hive:hive-llap-common:2.3.9
 - org.apache.hive:hive-llap-client:2.3.9
 - org.apache.avro:avro:1.10.2 (omitted for duplicate)

既然具备了hive的功能，那么就可以执行一切hive中能执行的动作：

- 建表
- show 表
- 建库
- show 库
- alter表
-

只不过，此时看见的表是spark中集成的hive的本地元数据库中的表！

如果想让spark中集成的hive，看见你外部集群中的hive的表，只要修改配置：把spark端的hive的元数据服务地址，指向外部集群中hive的元数据服务地址；

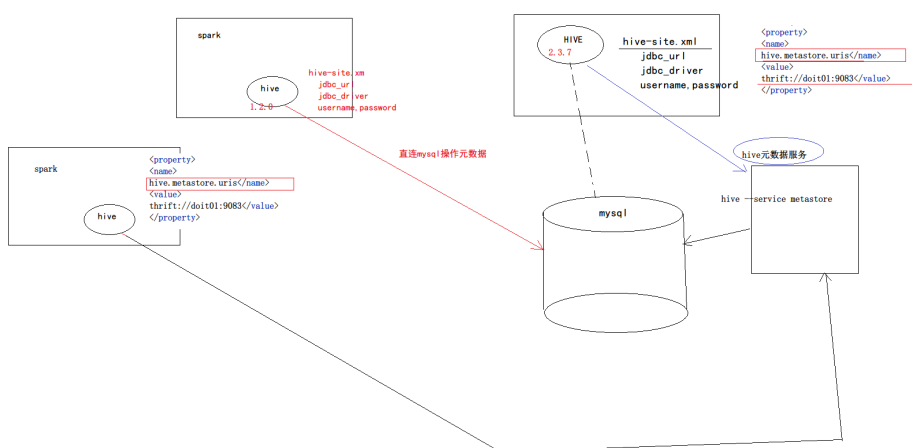
有两种指定办法：

- 在spark端加入hive-site.xml，里面配置 目标元数据库 mysql的连接信息

这会使得spark中集成的hive直接访问mysql元数据库

- 在spark端加入hive-site.xml，里面配置 目标hive的元数据服务器地址

这会使得spark中集成的hive通过外部独立的hive元数据服务来访问元数据库



(2) 从Hive创建DataFrame

Sparksql通过spark-hive整合包，来集成hive的功能

Sparksql加载“外部独立hive”的数据，本质上是不需要“外部独立hive”参与的，因为“外部独立hive”的表数据就在hdfs中，元数据信息在mysql中

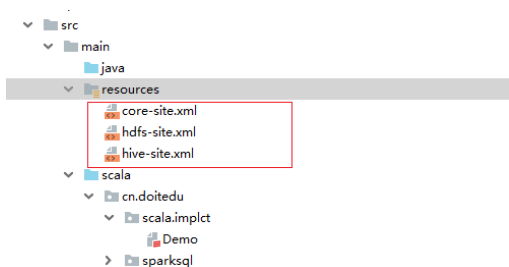
不管数据还是元数据，sparksql都可以直接去获取！

步骤：

1. 要在工程中添加spark-hive的依赖jar以及mysql的jdbc驱动jar

```
1 <dependency>
2     <groupId>mysql</groupId>
3     <artifactId>mysql-connector-java</artifactId>
4     <version>8.0.30</version>
5 </dependency>
6
7 <!-- spark整合hive的依赖，即可以读取hive的源数据库，使用hive特点的sql -->
8 <dependency>
9     <groupId>org.apache.spark</groupId>
10    <artifactId>spark-hive_2.12</artifactId>
11    <version>3.2.3</version>
12 </dependency>
```

2. 要在工程中添加hive-site.xml/core-site.xml配置文件



3. 创建sparksession时需要调用.enableHiveSupport()方法

```
1 val spark = SparkSession
2   .builder()
3   .appName(this.getClass.getSimpleName)
4   .master("local[*]")
```

```
5 // 启用hive支持,需要调用enableHiveSupport, 还需要添加一个依赖 spark-hive
6 // 默认sparksql内置了自己的hive
7 // 如果程序能从classpath中加载到hive-site配置文件, 那么它访问的hive元数据库就不是本地,
8 // 如果程序能从classpath中加载到core-site配置文件, 那么它访问的文件系统也不再是本地文件
9 .enableHiveSupport()
10 .getOrCreate()
```

4. 加载hive中的表

```
1 val df = spark.sql("select * from t1")
```

注意点：如果自己也用dataframe注册了一个同名的视图，那么这个视图名会替换掉hive的表

(3) 从Hbase创建DataFrame

其实，sparksql可以连接任意外部数据源（只要有对应的“连接器”即可）

Sparksql对hbase是有第三方连接器（华为）的，但是久不维护！

建议用hive作为连接器（hive可以访问hbase，而sparksql可以集成hive）

在hbase中建表

```
1 create 'doitedu_stu','f'
```

插入数据到hbase表

```
1 put 'doitedu_stu','001','f:name','zhangsan'
2 put 'doitedu_stu','001','f:age','26'
3 put 'doitedu_stu','001','f:gender','m'
4 put 'doitedu_stu','001','f:salary','28000'
5 put 'doitedu_stu','002','f:name','lisi'
```

```
6 put 'doitedu_stu','002','f:age','22'
7 put 'doitedu_stu','002','f:gender','m'
8 put 'doitedu_stu','002','f:salary','26000'
9 put 'doitedu_stu','003','f:name','wangwu'
10 put 'doitedu_stu','003','f:age','21'
11 put 'doitedu_stu','003','f:gender','f'
12 put 'doitedu_stu','003','f:salary','24000'
13 put 'doitedu_stu','004','f:name','zhao1iu'
14 put 'doitedu_stu','004','f:age','22'
15 put 'doitedu_stu','004','f:gender','f'
16 put 'doitedu_stu','004','f:salary','25000'
```

创建hive外部表映射hbase中的表

```
1 CREATE EXTERNAL TABLE doitedu_stu
2 (
3     id          string ,
4     name        string ,
5     age         int    ,
6     gender      string ,
7     salary      double
8 )
9 STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
10 WITH SERDEPROPERTIES ( 'hbase.columns.mapping'=':key,f:name,f:age,f:gender,f:sal
11 TBLPROPERTIES ( 'hbase.table.name'='default:doitedu_stu')
12 ;
```

工程中放置hbase-site.xml配置文件

```
1 <configuration>
2     <property>
3         <name>hbase.rootdir</name>
4         <value>hdfs://doit01:8020/hbase</value>
5     </property>
6
7     <property>
8         <name>hbase.cluster.distributed</name>
9         <value>true</value>
10    </property>
11
```



```

12     <property>
13         <name>hbase.zookeeper.quorum</name>
14         <value>doit01:2181,doit02:2181,doit03:2181</value>
15     </property>
16
17     <property>
18         <name>hbase.unsafe.stream.capability.enforce</name>
19         <value>false</value>
20     </property>
21 </configuration>

```

工程中添加hive-hbase-handler连接器依赖

```

1 <dependency>
2     <groupId>org.apache.hive</groupId>
3     <artifactId>hive-hbase-handler</artifactId>
4     <version>2.3.7</version>
5     <exclusions>
6         <exclusion>
7             <groupId>org.apache.hadoop</groupId>
8             <artifactId>hadoop-common</artifactId>
9         </exclusion>
10    </exclusions>
11 </dependency>

```

以读取hive表的方式直接读取即可

```
1 spark.sql("select * from doitedu_stu")
```

3.2 输出DF的各种方式

3.2.1 展现在控制台

```

1 df.show()
2 df.show(10)           //输出10行

```

```
3 df.show(10,false) // 不要截断列
```

3.2.2 保存为文件

```
1 object Demo17_SaveDF {
2
3   def main(args: Array[String]): Unit = {
4
5     val spark = SparkUtil.getSpark()
6
7     val df = spark.read.option("header",true).csv("data/stu2.csv")
8
9     val res = df.where("id>3").select("id","name")
10
11     // 展示结果
12     res.show(10,false)
13 }
```

保存结果为文件：parquet,json,csv,orc,textfile

文本文件是自由格式，框架无法判断该输出什么样的形式

```
1 res.write.parquet("out/parquetfile/")
2
3 res.write.csv("out/csvfile")
4
5 res.write.orc("out/orcfile")
6
7 res.write.json("out/jsonfile")
```

要将df输出为普通文本文件，则需要将df变成一个列

```
1 res.selectExpr("concat_ws('\001',id,name)")
2   .write.text("out/textfile")
3
```

3.2.3 保存到RDBMS

将dataframe写入mysql的表

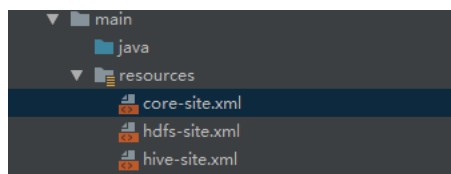
```
1 // 将dataframe通过jdbc写入mysql
2 val props = new Properties()
3 props.setProperty("user","root")
4 props.setProperty("password","root")
5
6 // 可以通过SaveMode来控制写入模式: SaveMode.Append/Ignore/Overwrite/ErrorIfExists(黑
7 res.write.mode(SaveMode.Append).jdbc("jdbc:mysql://localhost:3306/demo?character
```

3.2.4 写入hive

开启spark的hive支持

```
1 val spark = SparkSession
2   .builder()
3   .appName("")
4   .master("local[*]")
5   .enableHiveSupport()
6   .getOrCreate()
```

放入配置文件



写代码:

```
1 // 将dataframe写入hive, saveAsTable就是保存为hive的表
```

```
2 // 前提, spark要开启hiveSupport支持, spark-hive的依赖, hive的配置文件
3 res.write.saveAsTable("res")
```

3.2.5 DF输出时的分区操作

Hive中对表数据的存储, 可以将数据分为多个子目录!

比如:

```
1 create table tx(id int,name string) partitioned by (city string);
2 load data inpath '/data/1.dat' into table tx partition(city="beijing")
3 load data inpath '/data/2.dat' into table tx partition(city="shanghai")
```

Hive的表tx的目录结构如下:

```
1 /user/hive/warehouse/tx/
2             city=beijing/1.dat
3             city=shanghai/2.dat
```

查询的时候, 分区标识字段, 可以看做表的一个字段来用

```
1 Select * from tx where city='shanghai'
```

那么, sparksql既然是跟hive兼容的, 必然也有对分区存储支持的机制!

1. 能识别解析分区

有如下数据结构形式:

/bbb

Go!

Show

25

▼

entries

Search:

<div><div></div><div></div></div>	Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name	<div><div></div><div></div></div>
<div><div></div><div></div></div>	-rw-r--r--	coder	supergroup	0 B	Oct 11 15:11	1	128 MB	_SUCCESS	<div><div></div><div></div></div>
<div><div></div><div></div></div>	drwxr-xr-x	coder	supergroup	0 B	Oct 11 15:11	0	0 B	city=上海	<div><div></div><div></div></div>
<div><div></div><div></div></div>	drwxr-xr-x	coder	supergroup	0 B	Oct 11 15:11	0	0 B	city=北京	<div><div></div><div></div></div>
<div><div></div><div></div></div>	drwxr-xr-x	coder	supergroup	0 B	Oct 11 15:11	0	0 B	city=深圳	<div><div></div><div></div></div>

```

1 /**
2  * sparksql对分区机制的支持
3  * 识别已存在分区结构 /aaa/city=a; /aaa/city=b;
4  * 会将所有子目录都理解为数据内容，会将子目录中的city理解为一个字段
5  */
6 spark.read.csv("/aaa").show()

```

2. 能将数据按分区机制输出

```

1 /**
2  * sparksql对分区机制的支持
3  * 将dataframe存储为分区结构
4  */
5 val dfp = spark.read.option("header",true).csv("data/stu2.csv")
6 dfp.write.partitionBy("city").csv("/bbb")
7
8
9 /**
10 * 将数据分区写入hive
11 * 注意：写入hive的默认文件格式是parquet
12 */
13 dfp.write.partitionBy("sex").saveAsTable("res_p")

```

输入结构如下：

Show entries
Search:

<input type="checkbox"/>	<input type="checkbox"/> Permission	<input type="checkbox"/> Owner	<input type="checkbox"/> Group	<input type="checkbox"/> Size	<input type="checkbox"/> Last Modified	<input type="checkbox"/> Replication	<input type="checkbox"/> Block Size	<input type="checkbox"/> Name	<input type="checkbox"/>
<input type="checkbox"/>	-rw-r--r--	coder	supergroup	0 B	Oct 11 15:11	1	128 MB	_SUCCESS	<input type="checkbox"/>
<input type="checkbox"/>	drwxr-xr-x	coder	supergroup	0 B	Oct 11 15:11	0	0 B	city=上海	<input type="checkbox"/>
<input type="checkbox"/>	drwxr-xr-x	coder	supergroup	0 B	Oct 11 15:11	0	0 B	city=北京	<input type="checkbox"/>
<input type="checkbox"/>	drwxr-xr-x	coder	supergroup	0 B	Oct 11 15:11	0	0 B	city=深圳	<input type="checkbox"/>

Showing 1 to 4 of 4 entries

Previous
1
Next

3.3 DF数据运算操作

3.3.1 纯SQL操作

核心要义：将DataFrame 注册为一个临时视图view，然后就可以针对view直接执行各种sql

临时视图有两种：session级别视图，global级别视图；

session级别视图是Session范围内有效的，Session退出后，表就失效了；

全局视图则在application级别有效；

注意使用全局表时需要全路径访问：global_temp.people

```
1 // application全局有效
2 df.createGlobalTempView("stu")
3 spark.sql(
4     """
5     |select * from global_temp.stu a order by a.score desc
6     """).stripMargin)
7     .show()
8
```

```
1 // session有效
2 df.createTempView("s")
3 spark.sql(
4     """
5     |select * from s order by score
6     """).stripMargin)
7     .show()
```

```
1 val spark2 = spark.newSession()
2 // 全局有效的view可以在session2中访问
3 spark2.sql("select id,name from global_temp.stu").show()
4 // session有效的view不能在session2中访问
5 spark2.sql("select id,name from s").show()
```

以上只是对语法的简单示例，可以扩展到任意复杂的sql

挑战一下？

求出每个城市中，分数最高的学生信息；

Go go go !

3.3.2 DSL风格API(TableApi)语法

DSL风格API，就是用编程api的方式，来实现sql语法

DSL：特定领域语言

dataset的tableApi有一个特点：运算后返回值必回到dataframe

因为select后，得到的结果，无法预判返回值的具体类型，只能用通用的Row封装

数据准备

```
1 val df = spark.read
2   .option("header", true)
3   .option("inferSchema", true)
4   .csv("data_ware/demodata/stu.csv")
```

(1) 基本select及表达式

```
1 /**
2  * 逐行运算
3  */
4 // 使用字符串表达"列"
5 df.select("id","name").show()
6
7 // 如果要用字符串形式表达sql表达式，应该使用selectExpr方法
8 df.selectExpr("id+1","upper(name)").show
9 // select方法中使用字符串sql表达式，会被视作一个列名从而出错
10 // df.select("id+1","upper(name)").show()
11
12 import spark.implicits._
13 // 使用$符号创建Column对象来表达"列"
14 df.select($"id",$"name").show()
15
16 // 使用单边单引号创建Column对象来表达"列"
17 df.select('id,'name).show()
18
19 // 使用col函数来创建Column对象来表达"列"
20 import org.apache.spark.sql.functions._
21 df.select(col("id"),col("name")).show()
22
```

```
23 // 使用DataFrame的apply方法创建Column对象来表达列
24 df.select(df("id"),df("name")).show()
25
26 // 对Column对象直接调用Column的方法，或调用能生成Column对象的functions来实现sql中的运算
27 df.select('id.plus(2).leq("4").as("id2"),upper('name')).show()
28 df.select('id+2 <= 4 as "id2",upper('name')).show()
```

(3) 字段重命名

```
1 /**
2  * 字段重命名
3  */
4 // 对column对象调用as方法
5 df.select('id as "id2",$"name".as("n2"),col("age") as "age2").show()
6
7 // 在selectExpr中直接写sql的重命名语法
8 df.selectExpr("cast(id as string) as id2","name","city").show()
9
10 // 对dataframe调用withColumnRenamed方法对指定字段重命名
11 df.select("id","name","age").withColumnRenamed("id","id2").show()
12
13 // 对dataframe调用toDF对整个字段名全部重设
14 df.toDF("id2","name","age","city2","score").show()
```

(2) 条件过滤

```
1 /**
2  * 逐行过滤
3  */
4 df.where("id>4 and score>95")
5 df.where('id > 4 and 'score > 95').select("id","name","age").show()
```

(4) 分组聚合


```

1  /**
2   * 分组聚合
3   */
4  df.groupBy("city").count().show()
5  df.groupBy("city").min("score").show()
6  df.groupBy("city").max("score").show()
7  df.groupBy("city").sum("score").show()
8  df.groupBy("city").avg("score").show()
9
10 df.groupBy("city").agg(("score", "max"), ("score", "sum")).show()
11 df.groupBy("city").agg("score" -> "max", "score" -> "sum").show()

```

3.3.2.1 子查询

```

1  /**
2   * 子查询
3   * 相当于:
4   * select
5   * *
6   * from
7   * (
8   *   select
9   *   city, sum(score) as score
10  *   from stu
11  *   group by city
12  * ) o
13  * where score > 165
14  */
15 df.groupBy("city")
16   .agg(sum("score") as "score")
17   .where("score > 165")
18   .select("city", "score")
19   .show()

```

3.3.2.2 Join关联查询

```

1 package cn.doitedu.sparksql
2
3 import org.apache.spark.sql.{DataFrame, SparkSession}
4
5 /**
6  * 用 DSL风格api来对dataframe进行运算
7  */
8 object Demo14_DML_DSLapi {
9     def main(args: Array[String]): Unit = {
10
11         val spark = SparkUtil.getSpark()
12         import spark.implicits._
13
14         val df = spark.read.option("header",true).csv("data/stu2.csv")
15         val df2 = spark.read.option("header",true).csv("data/stu22.csv")
16
17         /**
18          * SQL中都有哪些运算?
19          *     1. 查询字段 (id)
20          *     2. 查询表达式 (算术运算, 函数 age+10, upper(name))
21          *     3. 过滤
22          *     4. 分组聚合
23          *     5. 子查询
24          *     6. 关联查询
25          *     7. union查询
26          *     8. 窗口分析
27          *     9. 排序
28          */
29
30         //selectOp(spark,df)
31         //whereOp(spark,df)
32         //groupByOp(spark,df)
33         joinOp(spark,df,df2)
34
35         spark.close()
36     }
37
38     /**
39     * 关联查询
40     * @param spark
41     * @param df1
42     */
43     def joinOp(spark:SparkSession,df1:DataFrame,df2:DataFrame): Unit ={
44
45         // 笛卡尔积
46         //df1.crossJoin(df2).show()
47

```

```

48 // 给join传入一个连接条件; 这种方式, 要求, 你的join条件字段在两个表中都存在且同名
49 df1.join(df2,"id").show()
50
51 // 传入多个join条件列, 要求两表中这多个条件列都存在且同名
52 df1.join(df2,Seq("id","sex")).show()
53
54
55 // 传入一个自定义的连接条件表达式
56 df1.join(df2,df1("id") + 1 === df2("id")).show()
57
58
59 // 还可以传入join方式类型: inner(默认), left, right, full, left_semi, left_an
60 df1.join(df2,df1("id")+1 === df2("id"),"left").show()
61 df1.join(df2,Seq("id"),"right").show()
62
63
64 /**
65  * 总结:
66  * join方式:   joinType: String
67  * join条件:
68  * 可以直接传join列名: usingColumn/usingColumns : Seq(String) 注意: 右表的join:
69  * 可以传join自定义表达式: Column.+(1) === Column      df1("id")+1 === df2("
70  */
71 }
72 }

```

3.3.2.3 Union操作

```

1 Sparksql中的union, 其实是union all
2 df.union(df).show()

```

3.3.2.4 窗口分析函数调用

测试数据:

```

1 id,name,age,sex,city,score
2 1,张飞,21,M,北京,80
3 2,关羽,23,M,北京,82
4 7,周瑜,24,M,北京,85
5 3,赵云,20,F,上海,88

```

```
6 4,刘备,26,M,上海,83
7 8,孙权,26,M,上海,78
8 5,曹操,30,F,深圳,90.8
9 6,孔明,35,F,深圳,77.8
10 9,吕布,28,M,深圳,98
```

求每个城市中成绩最高的两个人的信息，如果用sql写：

```
1 select
2 id,name,age,sex,city,score
3 from
4 (
5     select
6         id,name,age,sex,city,score,
7         row_number() over(partition by city order by score desc) as rn
8     from t
9 ) o
10 where rn<=2
```

DSL风格的API实现：

```
1 package cn.doitedu.sparksql
2
3 import org.apache.spark.sql.expressions.Window
4
5 /**
6  * 用dsl风格api实现sql中的窗口分析函数
7  */
8 object Demo15_DML_DSLAPI_WINDOW {
9
10     def main(args: Array[String]): Unit = {
11
12         val spark = SparkUtil.getSpark()
13
14         val df = spark.read.option("header",true).csv("data/stu2.csv")
15
16         import spark.implicits._
17         import org.apache.spark.sql.functions._
18
19     }
```

```

20     val window = Window.partitionBy('city').orderBy('score.desc')
21
22     df.select('id','name','age','sex','city','score',row_number().over(window) as "rn"
23         .where('rn <= 2)
24         .drop("rn") // 最后结果中不需要rn列, 可以drop掉这个列
25         .select('id','name','age','sex','city','score') // 或者用select指定你所需要的列
26         .show()
27
28     spark.close()
29 }
30 }

```

Dataset提供与RDD类似的编程算子，即map/flatMap/reduceByKey等等，不过本方式使用略少：

- 如果方便使用sql表达的逻辑，首选sql
- 如果不方便使用sql表达，则可以把Dataset转成RDD后使用 RDD的算子

直接在Dataset上调用类似RDD风格算子的代码示例如下：

```

1  /**
2      * 四、 dataset/dataframe 调 RDD算子
3      *
4      * dataset调rdd算子, 返回的还是dataset[U], 不过需要对应的 Encoder[U]
5      *
6      */
7  val ds4: Dataset[(Int, String, Int)] = ds2.map(p => (p.id, p.name, p.age + 1)
8  val ds5: Dataset[JavaPerson] = ds2.map(p => new JavaPerson(p.id,p.name,p.age
9
10 val ds6: Dataset[Map[String, String]] = ds2.map(per => Map("name" -> per.name
11 ds6.printSchema()
12 /**
13     * root
14         |-- value: map (nullable = true)
15         |   |-- key: string
16         |   |-- value: string (valueContainsNull = true)
17     */
18 // 从ds6中查询每个人的姓名
19 ds6.selectExpr("value['name']")
20

```

```

21
22 // dataframe上调RDD算子, 就等价于 dataset[Row]上调rdd算子
23 val ds7: Dataset[(Int, String, Int)] = frame.map(row=>{
24     val id: Int = row.getInt(0)
25     val name: String = row.getAs[String]("name")
26     val age: Int = row.getAs[Int]("age")
27     (id,name,age)
28 })
29
30 // 利用模式匹配从row中抽取字段数据
31 val ds8: Dataset[Per] = frame.map({
32     case Row(id:Int,name:String,age:Int) => Per(id,name,age*10)
33 })

```

3.5 Dataset与RDD混编

3.5.1 DataSet和Dataframe的区别

狭义上, Dataset中装的是用户自定义的类型

那么在抽取数据时, 比较方便 stu.id 且类型会得到编译时检查

狭义上, dataframe中装的是Row (框架内置的一个通用类型)

那么在抽取数据时, 不太方便, 得通过脚标, 或者字段名, 而且还得强转

```

1 val x:Any = row.get(1)
2 val x:Double = row.getDouble(1)
3 val x:Double = row.getAs[Double]("salary")

```

```

1 /**
2  * dataset存在的意义?
3  * 意义要从它的特点说起:
4  * ds的特点是, 可以存储各种自定义类型, 自定义类型中, 各字段是有类型约束 (所以ds是强类型约
5  * df只能存储row类型, 而row类型中的字段是没有“类型约束“, 全是any (所以df是弱类型约束的)
6  */
7 ds.map(bean => {
8     // val id:String = bean.id // 提取数据时不会产生类型匹配错误, 编译时就会检查
9

```

```

10 })
11
12
13 val _df: Dataset[Row] = ds.toDF()
14 _df.map(row => {
15     val name: Int = row.getInt(1) // 明明类型不匹配，但是编译时无法检查，运行时才会抛异常
16 })

```

3.5.3 DataFrame/dataset转成RDD后取数

要义：有些运算场景下，通过SQL语法实现计算逻辑比较困难，可以将DataFrame转成RDD算子来操作，而DataFrame中的数据是以RDD[Row]类型封装的，因此，要对DataFrame进行RDD算子操作，只需要掌握如何从Row中解构出数据即可

示例数据stu.csv

```

1 id,name,age,city,score
2 1,张飞,21,北京,80.0
3 2,关羽,23,北京,82.0
4 3,赵云,20,上海,88.6
5 4,刘备,26,上海,83.0
6 5,曹操,30,深圳,90.0

```

(1) 从Row中取数方式1：索引号

```

val df = spark.read
    .option("header", true)
    .option("inferSchema", true)
    .csv(path = "data_ware/demodata/stu.csv")

// 获取dataframe中的rdd
val rdd: RDD[Row] = df.rdd
rdd.map(row => {
    row.get
})

```

@get(i: Int)	Any
@getAs[T](i: Int)	T
@getAs[T](fieldName: String)	T
@getValuesMap[T](fieldNames: Seq[String])	Map[String, T]
@getMap[K, V](i: Int)	Map[K, V]
@getDouble(i: Int)	Double
@getList[T](i: Int)	util.List[T]
@getBoolean(i: Int)	Boolean
@getByte(i: Int)	Byte
@getDate(i: Int)	Date
@getDecimal(i: Int)	BigDecimal
@getFloat(i: Int)	Float
@getInt(i: Int)	Int
@getJavaMap[K, V](i: Int)	util.Map[K, V]
@getLong(i: Int)	Long
@getSeq[T](i: Int)	Seq[T]
@getShort(i: Int)	Short
@getString(i: Int)	String
@getStruct(i: Int)	Row
@getTimestamp(i: Int)	Timestamp
@getClass()	Class[_]

示例代码

```
1 val rdd: RDD[Row] = df.rdd
2 rdd.map(row=>{
3     val id = row.get(0).asInstanceOf[Int]
4     val name = row.getString(1)
5     (id,name)
6 }).take(10).foreach(println)
```

(2) 从Row中取数方式2：字段名

```
1 rdd.map(row=>{
2     val id = row.getAs[Int]("id")
3     val name = row.getAs[String]("name")
4     val age = row.getAs[Int]("age")
5     val city = row.getAs[String]("city")
6     val score = row.getAs[Double]("score")
7     (id,name,age,city,score)
8 }).take(10).foreach(println)
```

(3) 从Row中取数方式3：模式匹配

```
1 rdd.map({
2     case Row(id: Int, name: String, age: Int, city: String, score: Double)
3     => {
4         // do anything
5         (id,name,age,city,score)
6     }
7 }).take(10).foreach(println)
```

(4) 完整示例


```

1 package cn.doitedu.sparksql
2
3 import org.apache.spark.rdd.RDD
4 import org.apache.spark.sql.{Dataset, Row}
5 import org.apache.spark.sql.types.{DoubleType, IntegerType, StringType, StructFi
6
7 /**
8  * 有些场景下，逻辑不太方便用sql去实现，可能需要将dataframe退化成RDD来计算
9  * 示例需求： 求每种性别的成绩总和
10 */
11 object Demo16_DML_RDD {
12   def main(args: Array[String]): Unit = {
13
14     val spark = SparkUtil.getSpark()
15     // val schema = new StructType(Array(StructField("id",IntegerType),StructFie
16
17     // val schema = new StructType((StructField("id",IntegerType):: StructField(
18
19     val schema = new StructType()
20       .add("id",IntegerType)
21       .add("name",StringType)
22       .add("age",IntegerType)
23       .add("sex",StringType)
24       .add("city",StringType)
25       .add("score",DoubleType)
26
27     val df = spark.read.schema(schema).option("header",true).csv("data/stu2.csv"
28
29     // 可以直接在dataframe上用map等rdd算子
30     // 框架会把算子返回的结果RDD 再转回dataset，需要一个能对RDD[T]进行解析的Encoder[T].
31     // 好在大部分T类型都可以有隐式的Encoder来支持
32     import spark.implicits._
33     val ds2: Dataset[(Int, String)] = df.map(row=>{
34       val id = row.getAs[Int]("id")
35       val name = row.getAs[String]("name")
36       (id,name)
37     })
38
39
40     // dataframe中取出rdd后，就是一个RDD[Row]
41     val rd: RDD[Row] = df.rdd
42     // 从Row中取数据，就可以变成任意你想要的类型
43     val rdd2: RDD[(Int, String, Int, String, String, Double)] = rd.map(row=>{
44
45       // dataframe是一种弱类型结构（在编译时无法检查类型，因为数据被装在了一个array[any.
46       // val id = row.getDouble(1) // 如果类型取错，编译时是无法检查的，运行时才会报错
47

```

```

48     // 可以根据字段的脚标去取
49     val id: Int = row.getInt(0)
50     val name: String = row.getString(1)
51     val age: Int = row.getAs[Int](2)
52
53     // 可以根据字段名称去取
54     val sex: String = row.getAs[String]("sex")
55     val city: String = row.getAs[String]("city")
56     val score: Double = row.getAs[Double]("score")
57
58     (id,name,age,sex,city,score)
59 })
60
61
62 /**
63  * 用模式匹配从Row中抽取数据
64  * 效果跟上面的方法是一样的，但是更简洁！
65  */
66 val rdd22 = rd.map({
67     case Row(id:Int,name:String,age:Int,sex:String,city:String,score:Double)=>
68         (id,name,age,sex,city,score)
69 })
70
71
72 // 后续就跟dataframe没关系了，跟以前的rdd是一样的了
73 val res: RDD[(String, Double)] = rdd22.groupBy(tp=>tp._4).mapValues(iter=>{
74     iter.map(_._6).sum
75 })
76
77 res.foreach(println)
78
79 spark.close()
80 }
81 }

```

3.5.4从RDD创建DataFrame

准备测试用的数据和RDD

后续示例都起源于如下RDD

数据文件：doit_stu.txt

```
1 1,张飞,21,北京,80.0
2 2,关羽,23,北京,82.0
3 3,赵云,20,上海,88.6
4 4,刘备,26,上海,83.0
5 5,曹操,30,深圳,90.0
```

创建RDD:

```
1 val rdd:RDD[String] = spark.sparkContext.textFile("data_ware/demodata/stu.txt")
```

(1) 从RDD[Case class类]创建DataFrame

注: 定义一个case class来封装数据, 如下, Stu是一个case class类

```
1 val rdd:RDD[String] = spark.sparkContext.textFile("data_ware/demodata/stu.txt")
```

示例代码:

```
1 val rddStu: RDD[Stu] = rdd
2 // 切分字段
3 .map(_.split(","))
4 // 将每一行数据变形成一个多元组tuple
5 .map(arr => Stu(arr(0).toInt, arr(1), arr(2).toInt, arr(3), arr(4).toDouble))
6 // 创建DataFrame
7 val df = spark.createDataFrame(rddStu)
8 df.show()
```

结果如下:

```
1 +---+-----+---+-----+-----+
```

```

2 | id | name | age | city | score |
3 +---+-----+---+-----+
4 | 1 | 张飞 | 21 | 北京 | 80.0 |
5 | 2 | 关羽 | 23 | 北京 | 82.0 |
6 | 3 | 赵云 | 20 | 上海 | 88.6 |
7 | 4 | 刘备 | 26 | 上海 | 83.0 |
8 | 5 | 曹操 | 30 | 深圳 | 90.0 |
9 +---+-----+---+-----+

```

可以发现，框架成功地从case class的类定义中推断出了数据的schema：字段类型和字段名称

Schema获取手段：反射

当然，还有更简洁的方式，利用框架提供的隐式转换

```

1 // 更简洁办法
2 import spark.implicits._
3 val df = rddStu.toDF

```

(2) 从RDD[Tuple]创建DataFrame

```

1 val rddTuple: RDD[(Int, String, Int, String, Double)] = rdd
2 // 切分字段
3 .map(_.split(","))
4 // 将每一行数据变形成一个多元组tuple
5 .map(arr => (arr(0).toInt, arr(1), arr(2).toInt, arr(3), arr(4).toDouble))
6
7 //创建DataFrame
8 val df = spark.createDataFrame(rddTuple)
9 df.printSchema() // 打印schema信息
10 df.show()

```

结果如下：

```

1 root
2 |-- _1: integer (nullable = false)
3 |-- _2: string (nullable = true)

```

```

4  |-- _3: integer (nullable = false)
5  |-- _4: string (nullable = true)
6  |-- _5: double (nullable = false)
7
8
9  +---+-----+-----+-----+
10 | _1| _2 | _3| _4| _5 |
11 +---+-----+-----+-----+
12 |  1| 张飞| 21| 北京|80.0|
13 |  2| 关羽| 23| 北京|82.0|
14 |  3| 赵云| 20| 上海|88.6|
15 |  4| 刘备| 26| 上海|83.0|
16 |  5| 曹操| 30| 深圳|90.0|
17 +---+-----+-----+-----+

```

从结果中可以发现一个问题：框架从tuple元组结构中，对schema的推断，也是成功的，只是字段名是tuple中的数据访问索引。

当然，还有更简洁的方式，利用框架提供的隐式转换可以直接调用toDF创建，并指定字段名

```

1  // 更简洁办法
2  import spark.implicits._
3  val df2 = rddTuple.toDF("id","name","age","city","score")

```

(3) 从RDD[JavaBean]创建DataFrame

注：此处所说的Bean，指的是用java定义的bean

```

1  public class Stu2 {
2      private int id;
3      private String name;
4      private int age;
5      private String city;
6      private double score;
7
8      public Stu2(int id, String name, int age, String city, double score) {
9          this.id = id;
10         this.name = name;
11         this.age = age;

```

```
12         this.city = city;
13         this.score = score;
14     }
15
16     public int getId() {
17         return id;
18     }
19
20     public void setId(int id) {}
```

示例代码：

```
1 val rddBean: RDD[Stu2] = rdd
2 // 切分字段
3 .map(_.split(","))
4 // 将每一行数据变形成一个JavaBean
5 .map(arr => new Stu2(arr(0).toInt, arr(1), arr(2).toInt, arr(3), arr(4).toDouble))
6 val df = spark.createDataFrame(rddBean, classOf[Stu2])
7 df.show()
```

结果如下：

```
1 +---+-----+---+-----+---+
2 |age|city | id|name |score|
3 +---+-----+---+-----+---+
4 | 1| 张飞| 21| 北京|80.0|
5 | 2| 关羽| 23| 北京|82.0|
6 | 3| 赵云| 20| 上海|88.6|
7 | 4| 刘备| 26| 上海|83.0|
8 | 5| 曹操| 30| 深圳|90.0|
9 +---+-----+---+-----+---+
```

注：RDD[JavaBean]在spark.implicits._中没有toDF的支持

(4) 从RDD[普通Scala类]中创建DataFrame

注：此处的普通类指的是scala中定义的非case class的类

框架在底层将其视作java定义的标准bean类型来处理

而scala中定义的普通bean类，不具备字段的java标准getters和setters，因而会处理失败

可以如下处理来解决

普通scala bean类定义：

```
1 class Stu3(  
2     @BeanProperty  
3     val id: Int,  
4     @BeanProperty  
5     val name: String,  
6     @BeanProperty  
7     val age: Int,  
8     @BeanProperty  
9     val city: String,  
10    @BeanProperty  
11    val score: Double)
```

示例代码：

```
1 val rddStu3: RDD[Stu3] = rdd  
2 // 切分字段  
3 .map(_.split(","))  
4 // 将每一行数据变形成一个普通Scala对象  
5 .map(arr => new Stu3(arr(0).toInt, arr(1), arr(2).toInt, arr(3), arr(4).toDouble)  
6 val df = spark.createDataFrame(rddStu3, classOf[Stu3])  
7 df.show()
```

(5) 从RDD[Row]中创建DataFrame

注：DataFrame中的数据，本质上还是封装在RDD中，而RDD[T]总有一个T类型，DataFrame内部的RDD中的元素类型T即为框架所定义的Row类型；

```

1 val rddRow = rdd
2   // 切分字段
3   .map(_.split(","))
4   // 将每一行数据变形形成一个Row对象
5   .map(arr => Row(arr(0).toInt, arr(1), arr(2).toInt, arr(3), arr(4).toDouble))
6
7 val schema = new StructType()
8   .add("id", DataTypes.IntegerType)
9   .add("name", DataTypes.StringType)
10  .add("age", DataTypes.IntegerType)
11  .add("city", DataTypes.StringType)
12  .add("score", DataTypes.DoubleType)
13
14 val df = spark.createDataFrame(rddRow, schema)
15 df.show()

```

(6) 从RDD[set/seq/map]中创建DataFrame

版本2.2.0, 新增了对SET/SEQ的编解码支持

版本2.3.0, 新增了对Map的编解码支持

```

1 object Demo7_CreateDF_SetSeqMap {
2
3   def main(args: Array[String]): Unit = {
4
5     val spark = SparkSession.builder().appName("").master("local[*]").getOrCreate()
6
7     val seq1 = Seq(1,2,3,4)
8     val seq2 = Seq(11,22,33,44)
9
10    val rdd: RDD[Seq[Int]] = spark.sparkContext.parallelize(List(seq1, seq2))
11
12    import spark.implicits._
13    val df = rdd.toDF()
14
15    df.printSchema()
16    df.show()
17
18
19    df.selectExpr("value[0]", "size(value)").show()
20
21

```



```

22      /**
23       * set类型数据rdd的编解码
24       */
25      val set1 = Set("a","b")
26      val set2 = Set("c","d","e")
27      val rdd2: RDD[Set[String]] = spark.sparkContext.parallelize(List(set1,set2))
28
29      val df2 = rdd2.toDF("members")
30      df2.printSchema()
31      df2.show()
32
33
34      /**
35       * map类型数据rdd的编解码
36       */
37
38      val map1 = Map("father"->"mayun","mother"->"tangyan")
39      val map2 = Map("father"->"huateng","mother"->"yifei","brother"->"sicong")
40      val rdd3: RDD[Map[String, String]] = spark.sparkContext.parallelize(List(map
41
42      val df3 = rdd3.toDF("jiaren")
43      df3.printSchema()
44      df3.show()
45
46      df3.selectExpr("jiaren['mother']", "size(jiaren)", "map_keys(jiaren)", "map_val
47          .show(10, false)
48
49
50      spark.close()
51  }
52 }

```

set/seq 结构出来的字段类型为： array

Map 数据类型解构出来的字段类型为： map

3.5.5 从RDD创建DataSet

```

import spark.implicits._

spark.createdataset|
● createDataset[T](data: Seq[T])(implicit evidence$4: Encoder[T])
● createDataset[T](data: RDD[T])(implicit evidence$5: Encoder[T])
● createDataset[T](data: util.List[T])(implicit evidence$6: Encoder[T])
Press Ctrl+I to choose the selected for first suggestion and insert a dot afterwards.

```

(1) 从RDD[Case class类]创建Dataset

```

1 val rdd: RDD[Person] = spark.sparkContext.parallelize(Seq(
2   Person(1, "zs"),
3   Person(2, "ls")
4 ))
5
6 import spark.implicit._
7
8 // case class 类型的rdd, 转dataset
9 val ds: Dataset[Person] = spark.createDataset(rdd)
10 val ds2: Dataset[Person] = rdd.toDS()
11 ds.printSchema()
12 ds.show()

```

```

1 /**
2  * 创建一个javaBean 的RDD
3  * 隐式转换中没有支持好对javabean的encoder机制
4  * 需要自己传入一个encoder
5  * 可以构造一个简单的encoder, 具备序列化功能, 但是不具备字段解构的功能
6  * 但是, 至少能够把一个RDD[javabean] 变成一个 dataset[javabean]
7  * 后续可以通过rdd的map算子将数据从对象中提取出来, 组装成tuple元组, 然后toDF即可进入sql
8  */
9 val rdd2: RDD[JavaStu] = spark.sparkContext.parallelize(Seq(
10   new JavaStu(1, "a", 18, "上海", 99.9),
11   new JavaStu(2, "b", 28, "北京", 99.9),
12   new JavaStu(3, "c", 38, "西安", 99.9)
13 ))
14
15
16 val encoder = Encoders.kryo(classOf[JavaStu])
17 val ds2: Dataset[JavaStu] = spark.createDataset(rdd2)(encoder)
18
19 val df2: Dataset[Row] = ds2.map(stu => {
20   (stu.getId, stu.getName, stu.getAge)
21 }).toDF("id", "name", "age")
22 ds2.printSchema()
23 ds2.show()
24 df2.show()

```

3.3.2.5 从RDD[其他类]创建Dataset

```
1  /**
2   * 将一个RDD[Map] 变成 Dataset[Map]
3   * 2.3.0版才支持
4   *
5   */
6
7  val rdd3: RDD[Map[String, String]] = spark.sparkContext.parallelize(Seq(
8    Map("id"->"1", "name"->"zs1"),
9    Map("id"->"2", "name"->"zs2"),
10   Map("id"->"3", "name"->"zs3")
11 ))
12
13 val ds3: Dataset[Map[String, String]] = rdd3.toDS()
14 ds3.printSchema()
15 ds3.show()
```

3.6 RDD/DS/DF互转

RDD、DataFrame、Dataset三者有许多共性，有各自适用的场景常常需要在三者之间转换

DataFrame/Dataset转RDD:

```
1 val rdd1: RDD[Row] = testDF.rdd
2 val rdd2: RDD[T] = testDS.rdd
```

RDD转DataFrame:

```
1 import spark.implicits._
2 val testDF = rdd.map {line=>
3   (line._1, line._2)
4 }.toDF("col1", "col2")
```

一般用元组把一行的数据写在一起，然后在toDF中指定字段名

RDD转Dataset:

```
1 import spark.implicits._
2 case class Person(col1:String,col2:Int)extends Serializable //定义字段名和类型
3 val testDS:Dataset[Person] = rdd.map {line=>
4     Person(line._1,line._2)
5 }.toDS
```

可以注意到，定义每一行的类型（case class）时，已经给出了字段名和类型，后面只要往case class里面添加值即可

Dataset转DataFrame:

这个也很简单，因为只是把case class封装成Row

```
1 import spark.implicits._
2 val testDF:Dataset[Row] = testDS.toDF
```

DataFrame转Dataset:

```
1 import spark.implicits._
2 case class Coltest(col1:String,col2:Int)extends Serializable //定义字段名和类型
3 val testDS = testDF.as[Coltest]
```

这种方法就是在给出每一列的类型后，使用as方法，转成Dataset，这在数据类型是DataFrame又需要针对各个字段处理时极为方便。

在使用一些特殊的操作时，一定要加上 `import spark.implicits._` 不然toDF、toDS无法使用

4. 用户自定义函数

通过spark.udf功能用户可以自定义函数。

4.1用户自定义UDF函数

```
1 scala> val df = spark.read.json("examples/src/main/resources/people.json")
2 df: org.apache.spark.sql.DataFrame = [age: bigint, name: string]
3
4 scala> df.show()
5 +-----+-----+
6 | age | name |
7 +-----+-----+
8 | null | Michael |
9 | 30 | Andy |
10 | 19 | Justin |
11 +-----+-----+
12
13
14 scala> spark.udf.register("addName", (x:String)=> "Name:"+x)
15 res5: org.apache.spark.sql.expressions.UserDefinedFunction = UserDefinedFunction
16
17 scala> df.createOrReplaceTempView("people")
18
19 scala> spark.sql("Select addName(name), age from people").show()
20 +-----+-----+
21 | UDF:addName(name) | age |
22 +-----+-----+
23 | Name:Michael | null |
24 | Name:Andy | 30 |
25 | Name:Justin | 19 |
26 +-----+-----+
```

UDF案例2

需求，有如下数据

```

1 id,name,age,height,weight,yanzhi,score
2 1,a,18,172,120,98,68.8
3 2,b,28,175,120,97,68.8
4 3,c,30,180,130,94,88.8
5 4,d,18,168,110,98,68.8
6 5,e,26,165,120,98,68.8
7 6,f,27,182,135,95,89.8
8 7,g,19,171,122,99,68.8

```

需要计算每一个和其他人之间的余弦相似度（特征向量之间的余弦相似度）

$$\begin{aligned}
 \cos(\theta) &= \frac{\sum_{i=1}^n (x_i \times y_i)}{\sqrt{\sum_{i=1}^n (x_i)^2} \times \sqrt{\sum_{i=1}^n (y_i)^2}} \\
 &= \frac{a \bullet b}{||a|| \times ||b||}
 \end{aligned}$$

公式(4)

代码实现：

```

1 package cn.doitedu.sparksql.udf
2
3 import cn.doitedu.sparksql.dataframe.SparkUtil
4 import org.apache.spark.sql.Row
5 import org.apache.spark.sql.expressions.UserDefinedFunction
6
7 import scala.collection.mutable
8
9
10 /**
11  * UDF 案例2： 用一个自定义函数实现两个向量之间的余弦相似度计算
12  */
13
14 case class Human(id: Int, name: String, features: Array[Double])
15
16 object CosinSimilarity {
17
18   def main(args: Array[String]): Unit = {
19

```

```

20
21 val spark = SparkUtil.getSpark()
22 import spark.implicits._
23 import spark.sql
24 // 加载用户特征数据
25 val df = spark.read.option("inferSchema", true).option("header", true).csv("
26 df.show()
27
28
29
30 // id,name,age,height,weight,yanzhi,score
31 // 将用户特征数据组成一个向量(数组)
32 // 方式1:
33 df.rdd.map(row => {
34     val id = row.getAs[Int]("id")
35     val name = row.getAs[String]("name")
36     val age = row.getAs[Double]("age")
37     val height = row.getAs[Double]("height")
38     val weight = row.getAs[Double]("weight")
39     val yanzhi = row.getAs[Double]("yanzhi")
40     val score = row.getAs[Double]("score")
41
42     (id, name, Array(age, height, weight, yanzhi, score))
43 }).toDF("id", "name", "features")
44
45 // 方式2:
46 df.rdd.map({
47     case Row(id: Int, name: String, age: Double, height: Double, weight: Double, yanzhi: Double, score: Double)
48     => (id, name, Array(age, height, weight, yanzhi, score))
49 })
50     .toDF("id", "name", "features")
51
52
53 // 方式3: 直接利用sql中的函数array来生成一个数组
54 df.selectExpr("id", "name", "array(age,height,weight,yanzhi,score) as features")
55 import org.apache.spark.sql.functions._
56 df.select('id, 'name, array('age, 'height, 'weight, 'yanzhi, 'score) as "features")
57
58 // 方式4: 返回case class
59 val features = df.rdd.map({
60     case Row(id: Int, name: String, age: Double, height: Double, weight: Double, yanzhi: Double, score: Double)
61     => Human(id, name, Array(age, height, weight, yanzhi, score))
62 })
63     .toDF()
64
65 // 将表自己和自己join, 得到每个人和其他所有人的连接行
66 val joined = features.join(features.toDF("bid", "bname", "bfeatures"), 'id < 'bid)

```

```

67     joined.show(100, false)
68
69     // 定义一个计算余弦相似度的函数
70     // val cosinSim = (f1:Array[Double],f2:Array[Double])=>{ /* 余弦相似度 */ }
71     // 开根号的api:    Math.pow(4.0,0.5)
72     val cosinSim = (f1:mutable.WrappedArray[Double], f2:mutable.WrappedArray[Dou
73
74         val fenmu1 = Math.pow(f1.map(Math.pow(_,2)).sum,0.5)
75         val fenmu2 = Math.pow(f2.map(Math.pow(_,2)).sum,0.5)
76
77         val fenzi = f1.zip(f2).map(tp=>tp._1*tp._2).sum
78
79         fenzi/(fenmu1*fenmu2)
80     }
81
82     // 注册到sql引擎:    spark.udf.register("cosin_sim",cosinSim)
83     spark.udf.register("cos_sim",cosinSim)
84     joined.createTempView("temp")
85
86     // 然后在这个表上计算两人之间的余弦相似度
87     sql("select id,bid,cos_sim(features,bfeatures) as cos_similary from temp").s
88
89     // 可以自定义函数简单包装一下，就成为一个能生成column结果的dsl风格函数了
90     val cossim2: UserDefinedFunction = udf(cosinSim)
91     joined.select('id,'bid,coossim2('features,'bfeatures) as "cos_sim").show()
92
93     spark.close()
94 }
95 }

```

4.2用户自定义聚合函数UDAF

弱类型的DataFrame和强类型的Dataset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UDAF聚合函数。

UDAF的编程模板：

```
/**
 * @date: 2019/10/12
 * @site: www.doitedu.cn
 * @author: hunter.d 涛哥
 * @qq: 657270652
 * @description:
 *   用户自定义UDAF入门示例：求薪资的平均值
 */
object MyAvgUDAF extends UserDefinedAggregateFunction{

    // 函数输入的字段schema（字段名-字段类型）
    override def inputSchema: StructType = ???

    // 聚合过程中，用于存储局部聚合结果的schema
    // 比如求平均薪资，中间缓存(局部数据薪资总和,局部数据人数总和)
    override def bufferSchema: StructType = ???

    // 函数的最终返回结果数据类型
    override def dataType: DataType = ???

    // 你这个函数是否是稳定一致的？（对一组相同的输入，永远返回相同的结果），只要是确定的，就写true
    override def deterministic: Boolean = true

    // 对局部聚合缓存的初始化方法
    override def initialize(buffer: MutableAggregationBuffer): Unit = ???

    // 聚合逻辑所在方法，框架会不断地传入一个新的输入row，来更新你的聚合缓存数据
    override def update(buffer: MutableAggregationBuffer, input: Row): Unit = ???
```

// 全局聚合：将多个局部缓存中的数据，聚合成一个缓存

// 比如：薪资和薪资累加，人数和人数累加

```
override def merge(buffer1: MutableAggregationBuffer, buffer2: Row): Unit = ???
```

// 最终输出

// 比如：从全局缓存中取薪资总和/人数总和

```
override def evaluate(buffer: Row): Any = ???
```

```
}
```

核心要义：

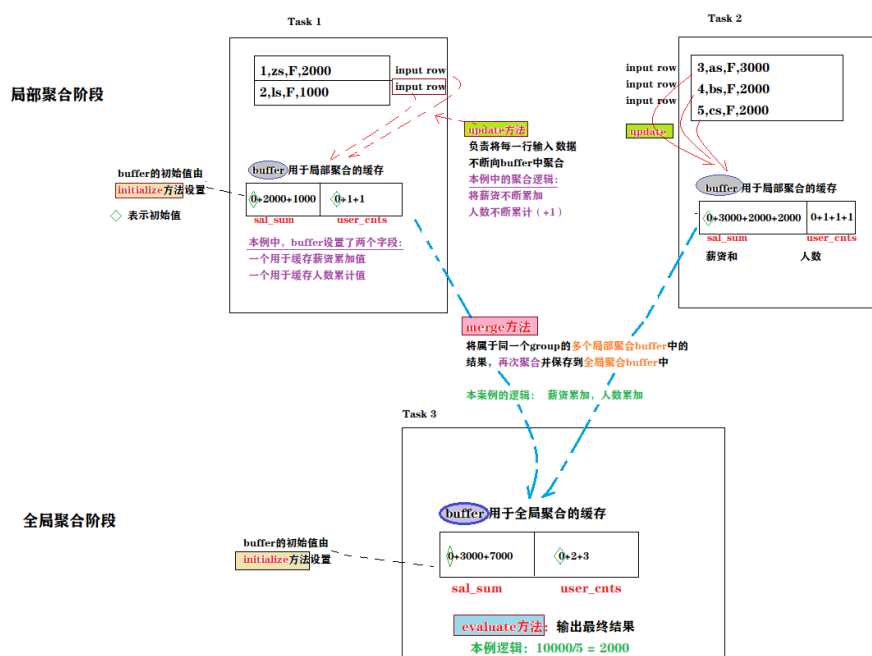
聚合是分步骤进行：先局部聚合，再全局聚合

局部聚合（update）的结果是保存在一个局部buffer中的

全局聚合(merge)就是将多个局部buffer再聚合成一个buffer

最后通过evaluate将全局聚合的buffer中的数据做一个运算得出你要的结果

如下图所示：



4.2.1弱类型用户自定义聚合函数UDAF

(1) 需求说明

示例数据：

```
+---+-----+-----+-----+-----+
| id | name      | sales | discount | state | saleDate |
+---+-----+-----+-----+-----+
| 1 | Widget Co | 1000.0 | 0.0 | AZ | 2014-01-01 |
| 2 | Acme Widgets | 2000.0 | 500.0 | CA | 2014-02-01 |
| 3 | Widgetry | 1000.0 | 200.0 | CA | 2015-01-11 |
| 4 | Widgets R Us | 2000.0 | 0.0 | CA | 2015-02-19 |
| 5 | Ye Olde Widgete | 3000.0 | 0.0 | MA | 2015-02-28 |
+---+-----+-----+-----+-----+
```

需求：计算x年份的同比上一年份的总销售增长率；比如2015 vs 2014的同比增长

显然，没有任何一个内置聚合函数可以完成上述需求；

可以多写一些sql逻辑来实现，但如果能自定义一个聚合函数，当然更方便高效！

Select yearOnyear(saleDate,sales) from t

(2) 自定义UDAF实现销售额同比计算

通过继承UserDefinedAggregateFunction来实现用户自定义聚合函数。

自定义UDAF的代码骨架如下：

```
class UdfMy extends UserDefinedAggregateFunction{
  override def inputSchema: StructType = ???

  override def bufferSchema: StructType = ???

  override def dataType: DataType = ???

  override def deterministic: Boolean = ???
```

```
override def initialize(buffer: MutableAggregationBuffer): Unit = ???
```

```
override def update(buffer: MutableAggregationBuffer, input: Row): Unit = ???
```

```
override def merge(buffer1: MutableAggregationBuffer, buffer2: Row): Unit = ???
```

```
override def evaluate(buffer: Row): Any = ???
```

```
}
```

完整实现代码如下：

```
/**
 * 工具类
 * @param startDate
 * @param endDate
 */
case class DateRange(startDate: Timestamp, endDate: Timestamp) {
  def contain(targetDate: Date): Boolean = {
    targetDate.before(endDate) && targetDate.after(startDate)
  }
}

/**
 * @date: 2019/10/10
 * @site: www.doitedu.cn
 * @author: hunter.d 涛哥
 * @qq: 657270652
 * @description: 自定义UDAF实现年份销售额同比增长计算
 */
class YearOnYearBasis(current: DateRange) extends UserDefinedAggregateFunction{
```

// 聚合函数输入参数的数据类型

```
override def inputSchema: StructType = {  
  StructType(StructField("metric", DoubleType) :: StructField("timeCategory", DateType) :: Nil)  
}
```

// 聚合缓冲区中值得数据类型

```
override def bufferSchema: StructType = {  
  StructType(StructField("sumOfCurrent", DoubleType) :: StructField("sumOfPrevious",  
DoubleType) :: Nil)  
}
```

// 返回值的数据类型

```
override def dataType: DataType = DoubleType
```

// 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.update(0, 0.0)  
  buffer.update(1, 0.0)  
}
```

// 相同Execute间的数据合并。

```
override def update(buffer: MutableAggregationBuffer, input: Row): Unit = {  
  if (current.contain(input.getAs[Date](1))) {  
    buffer(0) = buffer.getAs[Double](0) + input.getAs[Double](0)  
  }  
  
  val previous = DateRange(subtractOneYear(current.startDate),  
subtractOneYear(current.endDate))
```

```

if (previous.contain(input.getAs[Date](1))) {
    buffer(1) = buffer.getAs[Double](0) + input.getAs[Double](0)
}
}

```

// 不同Execute间的数据合并

```

override def merge(buffer1: MutableAggregationBuffer, buffer2: Row): Unit = {
    buffer1(0) = buffer1.getAs[Double](0) + buffer2.getAs[Double](0)
    buffer1(1) = buffer1.getAs[Double](1) + buffer2.getAs[Double](1)
}

```

// 计算最终结果

```

override def evaluate(buffer: Row): Any = {
    if (buffer.getDouble(1) == 0.0)
        0.0
    else
        (buffer.getDouble(0) - buffer.getDouble(1)) / buffer.getDouble(1) * 100
}

```

```

def subtractOneYear(d:Timestamp):Timestamp={
    Timestamp.valueOf(d.toLocalDateTime.minusYears(1))
}
}

```

(3) 补充示例：自定义UDAF实现平均薪资计算

下面展示一个求平均工资的自定义聚合函数。

```

1 package cn.doitedu.sparksql.udf
2

```

```

3 import org.apache.spark.sql.Row
4 import org.apache.spark.sql.expressions.{MutableAggregationBuffer, UserDefinedAg
5 import org.apache.spark.sql.types.{DataType, DataTypes, StructField, StructType}
6
7 /**
8  * @description:
9  * 用户自定义UDAF入门示例：求薪资的平均值
10  */
11 object MyAvgUDAF extends UserDefinedAggregateFunction {
12
13     // 函数输入的字段schema (字段名-字段类型)
14     override def inputSchema: StructType = StructType(Seq(StructField("salary", Da
15
16     // 聚合过程中，用于存储局部聚合结果的schema
17     // 比如求平均薪资，中间缓存(局部数据薪资总和,局部数据人数总和)
18     override def bufferSchema: StructType = StructType(Seq(
19         StructField("sum", DataTypes.DoubleType),
20         StructField("cnts", DataTypes.LongType)
21
22     ))
23
24     // 函数的最终返回结果数据类型
25     override def dataType: DataType = DataTypes.DoubleType
26
27     // 你这个函数是否是稳定一致的？（对一组相同的输入，永远返回相同的结果），只要是确定的，就
28     override def deterministic: Boolean = true
29
30     // 对局部聚合缓存的初始化方法
31     override def initialize(buffer: MutableAggregationBuffer): Unit = {
32         buffer.update(0, 0.0)
33         buffer.update(1, 0L)
34     }
35
36     // 聚合逻辑所在方法，框架会不断地传入一个新的输入row，来更新你的聚合缓存数据
37     override def update(buffer: MutableAggregationBuffer, input: Row): Unit = {
38
39         // 从输入中获取那个人的薪资，加到buffer的第一个字段上
40         buffer.update(0, buffer.getDouble(0) + input.getDouble(0))
41
42         // 给buffer的第2个字段加1
43         buffer.update(1, buffer.getLong(1) + 1)
44
45     }
46
47     // 全局聚合：将多个局部缓存中的数据，聚合成一个缓存
48     // 比如：薪资和薪资累加，人数和人数累加
49     override def merge(buffer1: MutableAggregationBuffer, buffer2: Row): Unit = {

```

```

50
51 // 把两个buffer的字段1（薪资和）累加到一起，并更新回buffer1
52 buffer1.update(0, buffer1.getDouble(0) + buffer2.getDouble(0))
53
54 // 更新人数
55 buffer1.update(1, buffer1.getLong(1) + buffer2.getLong(1))
56
57 }
58
59 // 最终输出
60 // 比如：从全局缓存中取薪资总和/人数总和
61 override def evaluate(buffer: Row): Any = {
62
63     if (buffer.getLong(1) != 0)
64         buffer.getDouble(0) / buffer.getLong(1)
65     else
66         0.0
67
68 }
69 }

```

4.2.2强类型用户自定义聚合函数

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

```

1 import org.apache.spark.sql.expressions.Aggregator
2 import org.apache.spark.sql.Encoder
3 import org.apache.spark.sql.Encoders
4 import org.apache.spark.sql.Session
5
6 // 既然是强类型，可能有case类
7 case class Employee(name: String, salary: Long)
8 case class Average(var sum: Long, var count: Long)
9
10 object MyAverage extends Aggregator[Employee, Average, Double] {
11     // 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
12     def zero: Average = Average(0L, 0L)
13     // Combine two values to produce a new value. For performance, the function
14     // and return it instead of constructing a new object
15     def reduce(buffer: Average, employee: Employee): Average = {
16         buffer.sum += employee.salary
17         buffer.count += 1

```



```

18     buffer
19 }
20 // 聚合不同execute的结果
21 def merge(b1: Average, b2: Average): Average = {
22     b1.sum += b2.sum
23     b1.count += b2.count
24     b1
25 }
26 // 计算输出
27 def finish(reduction: Average): Double = reduction.sum.toDouble / reduction.
28 // 设定之间值类型的编码器, 要转换成case类
29 // Encoders.product是进行scala元组和case类转换的编码器
30 def bufferEncoder: Encoder[Average] = Encoders.product
31 // 设定最终输出值的编码器
32 def outputEncoder: Encoder[Double] = Encoders.scalaDouble
33 }
34 import spark.implicits._
35
36 val ds = spark.read.json("examples/src/main/resources/employees.json").as[Em
37 ds.show()
38 // +-----+-----+
39 // |   name|salary|
40 // +-----+-----+
41 // |Michael|  3000|
42 // |   Andy|  4500|
43 // | Justin|  3500|
44 // |   Berta| 4000|
45 // +-----+-----+
46
47 // Convert the function to a `TypedColumn` and give it a name
48 val averageSalary = MyAverage.toColumn.name("average_salary")
49 val result = ds.select(averageSalary)
50 result.show()
51 // +-----+
52 // |average_salary|
53 // +-----+
54 // |           3750.0|
55 // +-----+
56 }

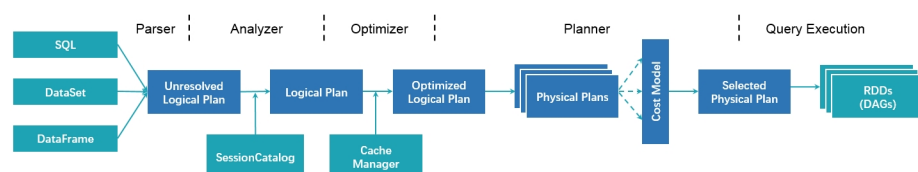
```

5. Spark SQL 的运行原理

正常的 SQL 执行先会经过 SQL Parser 解析 SQL，然后经过 Catalyst 优化器处理，最后到 Spark 执行。而 Catalyst 的过程又分为很多个过程，其中包括：

- Analysis：主要利用 Catalog 信息将 Unresolved Logical Plan 解析成 Analyzed logical plan；
- Logical Optimizations：利用一些 Rule（规则）将 Analyzed logical plan 解析成 Optimized Logical Plan；
- Physical Planning：前面的 logical plan 不能被 Spark 执行，而这个过程是把 logical plan 转换成多个 physical plans，然后利用代价模型（cost model）选择最佳的 physical plan；
- Code Generation：这个过程会把 SQL逻辑生成Java字节码。

所以整个 SQL 的执行过程可以使用下图表示：



其中蓝色部分就是 Catalyst 优化器处理的部分，也是本章主要讲解的内容。

5.1 元数据管理SessionCatalog

SessionCatalog 主要用于各种函数资源信息和元数据信息（数据库、数据表、数据视图、数据分区与函数等）的统一管理。

创建临时表或者视图，其实是往SessionCatalog注册；

Analyzer在进行逻辑计划元数据绑定时，也是从catalog中获取元数据；

5.2 SQL解析成逻辑执行计划

当调用SparkSession的sql或者SQLContext的sql方法，就会使用[SparkSqlParser](#)进行SQL解析。

Spark 2.0.0开始引入了第三方语法解析器工具 ANTLR，对 SQL 进行词法分析并构建语法树。

（Antlr 是一款强大的语法生成器工具，可用于读取、处理、执行和翻译结构化的文本或二进制文件，是当前 Java 语言中使用最为广泛的语法生成器工具，我们常见的大数据 SQL 解析都用到了这个工具，包括 Hive、Cassandra、Phoenix、Pig 以及 presto 等）目前最新版本的 Spark 使用的是 ANTLR4)

它分为2个步骤来生成Unresolved LogicalPlan：

- 词法分析（SqlBaseLexer）：Lexical Analysis，负责将token分组为符号类
- 语法分析（SqlBaseParser）：构建一棵分析树(parse tree)或者抽象语法树AST（abstract syntax tree）

```

1  /**
2   * The AstBuilder converts an ANTLR4 ParseTree into a catalyst Expression, Logic
3   * TableIdentifier.
4   */
5  class AstBuilder(conf: SQLConf) extends SqlBaseBaseVisitor[AnyRef] with Logging
6    import ParserUtils._
7
8    def this() = this(new SQLConf())
9
10   protected def typedVisit[T](ctx: ParseTree): T = {
11
12     ...
13   }
14 }

```

具体来说，Spark 基于presto的语法文件定义了Spark SQL语法文件SqlBase.g4

(路径 spark-2.4.3\sql\catalyst\src\main\antlr4\org\apache\spark\sql\catalyst\parser\SqlBase.g4)

这个文件定义了 Spark SQL 支持的 SQL 语法。

master
spark / sql / catalyst / src / main / antlr4 / org / apache / spark / sql / catalyst / parser / SqlBase.g4


sarutak [SPARK-36371][SQL] Support raw string literal ... ✓













































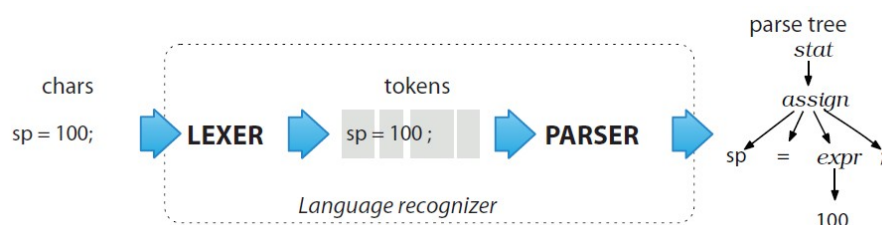


```

1 SELECT sum(v)
2   FROM (
3     SELECT
4       t1.id,
5       1 + 2 + t1.value AS v
6     FROM t1 JOIN t2
7     WHERE
8       t1.id = t2.id AND
9       t1.cid = 1 AND
10      t1.did = t1.cid + 1 AND
11      t2.id > 5) o

```

整个过程就类似于下图。



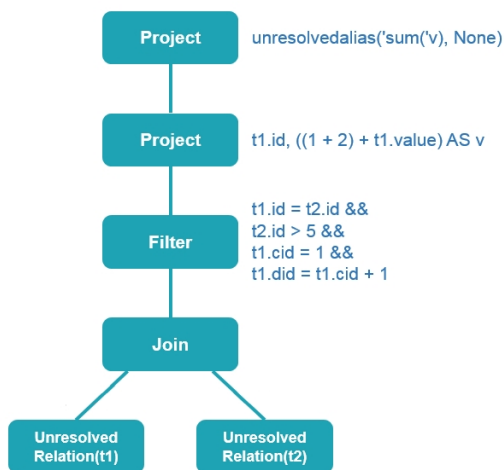
生成语法树之后，使用 AstBuilder 将语法树转换成 LogicalPlan，这个 LogicalPlan 也被称为 Unresolved LogicalPlan。解析后的逻辑计划如下：

```

1 == Parsed Logical Plan ==
2 'Project [unresolvedalias('sum('v), None)]
3 +- 'SubqueryAlias `doitedu_stu`
4   +- 'Project ['t1.id, ((1 + 2) + 't1.value) AS v#16]
5     +- 'Filter (((('t1.id = 't2.id) && ('t1.cid = 1)) && (('t1.did = ('t1.cid +
6       +- 'Join Inner
7         :- 'UnresolvedRelation `t1`
8         +- 'UnresolvedRelation `t2`

```

图片表示如下：



Unresolved LogicalPlan 是从下往上看，t1 和 t2 两张表被生成了 UnresolvedRelation，过滤的条件、选择的列以及聚合字段都知道了。

Unresolved LogicalPlan 仅仅是一种数据结构，不包含任何数据信息，比如不知道数据源、数据类型，不同的列来自于哪张表等。

5.3 Analyzer绑定逻辑计划

Analyzer 阶段会使用事先定义好的 Rule 以及 SessionCatalog 等信息对 Unresolved LogicalPlan 进行元数据绑定。

```

1  /**
2   * Provides a logical query plan analyzer, which translates [[UnresolvedAttribut
3   * [[UnresolvedRelation]]s into fully typed objects using information in a [[Ses
4   */
5  class Analyzer(
6    catalog: SessionCatalog,
7    conf: SQLConf,
8    maxIterations: Int)
9    extends RuleExecutor[LogicalPlan] with CheckAnalysis {
10
11
12  class SparkSqlParser(conf: SQLConf) extends AbstractSqlParser(conf) {
13    val astBuilder = new SparkSqlAstBuilder(conf)
14
15
16    override def parsePlan(sqlText: String): LogicalPlan = parse(sqlText) { parse
17      astBuilder.visitSingleStatement(parser.singleStatement()) match {
18        case plan: LogicalPlan => plan
19        case _ =>
20          val position = Origin(None, None)
21          throw new ParseException(Option(sqlText), "Unsupported SQL statement", p
22      }
  
```

```

23     }
24
25 Rule 是定义在 Analyzer 里面的，具体如下：
26 lazy val batches: Seq[Batch] = Seq(
27     Batch("Hints", fixedPoint,
28         new ResolveHints.ResolveBroadcastHints(conf),
29         ResolveHints.ResolveCoalesceHints,
30         ResolveHints.RemoveAllHints),
31     Batch("Simple Sanity Check", Once,
32         LookupFunctions),
33     Batch("Substitution", fixedPoint,
34         CTESubstitution,
35         WindowsSubstitution,
36         EliminateUnions,
37         new SubstituteUnresolvedOrdinals(conf)),
38     Batch("Resolution", fixedPoint,
39         ResolveTableValuedFunctions :: //解析表的函数
40         ResolveRelations :: //解析表或视图
41         ResolveReferences :: //解析列
42         ResolveCreateNamedStruct ::
43         ResolveDeserializer :: //解析反序列化操作类
44         ResolveNewInstance ::
45         ResolveUpCast :: //解析类型转换
46         ResolveGroupingAnalytics ::
47         ResolvePivot ::
48         ResolveOrdinalInOrderByAndGroupBy ::
49         ResolveAggAliasInGroupBy ::
50         ResolveMissingReferences ::
51         ExtractGenerator ::
52         ResolveGenerate ::
53         ResolveFunctions :: //解析函数
54         ResolveAliases :: //解析表别名
55         ResolveSubquery :: //解析子查询
56         ResolveSubqueryColumnAliases ::
57         ResolveWindowOrder ::
58         ResolveWindowFrame ::
59         ResolveNaturalAndUsingJoin ::
60         ResolveOutputRelation ::
61         ExtractWindowExpressions ::
62         GlobalAggregates ::
63         ResolveAggregateFunctions ::
64         TimeWindowing ::
65         ResolveInlineTables(conf) ::
66         ResolveHigherOrderFunctions(catalog) ::
67         ResolveLambdaVariables(conf) ::
68         ResolveTimeZone(conf) ::
69         ResolveRandomSeed ::

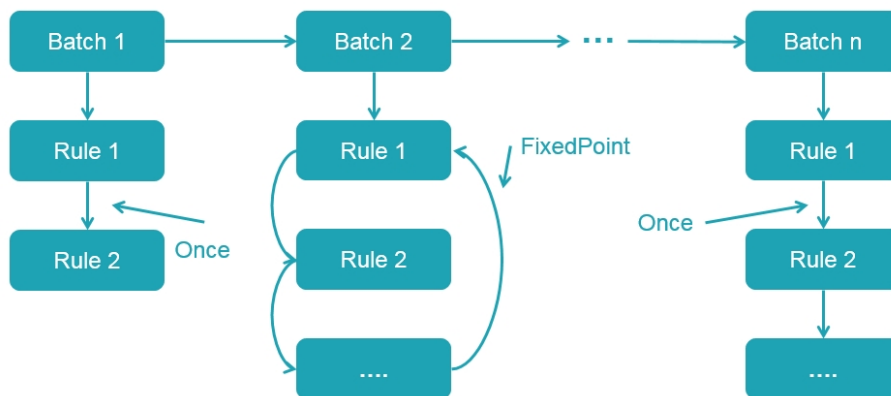
```

```

70     TypeCoercion.typeCoercionRules(conf) ++
71     extendedResolutionRules : _*),
72     Batch("Post-Hoc Resolution", Once, postHocResolutionRules: _*),
73     Batch("View", Once,
74         AliasViewChild(conf)),
75     Batch("Nondeterministic", Once,
76         PullOutNondeterministic),
77     Batch("UDF", Once,
78         HandleNullInputsForUDF),
79     Batch("FixNullability", Once,
80         FixNullability),
81     Batch("Subquery", Once,
82         UpdateOuterReferences),
83     Batch("Cleanup", fixedPoint,
84         CleanupAliases)
85 )

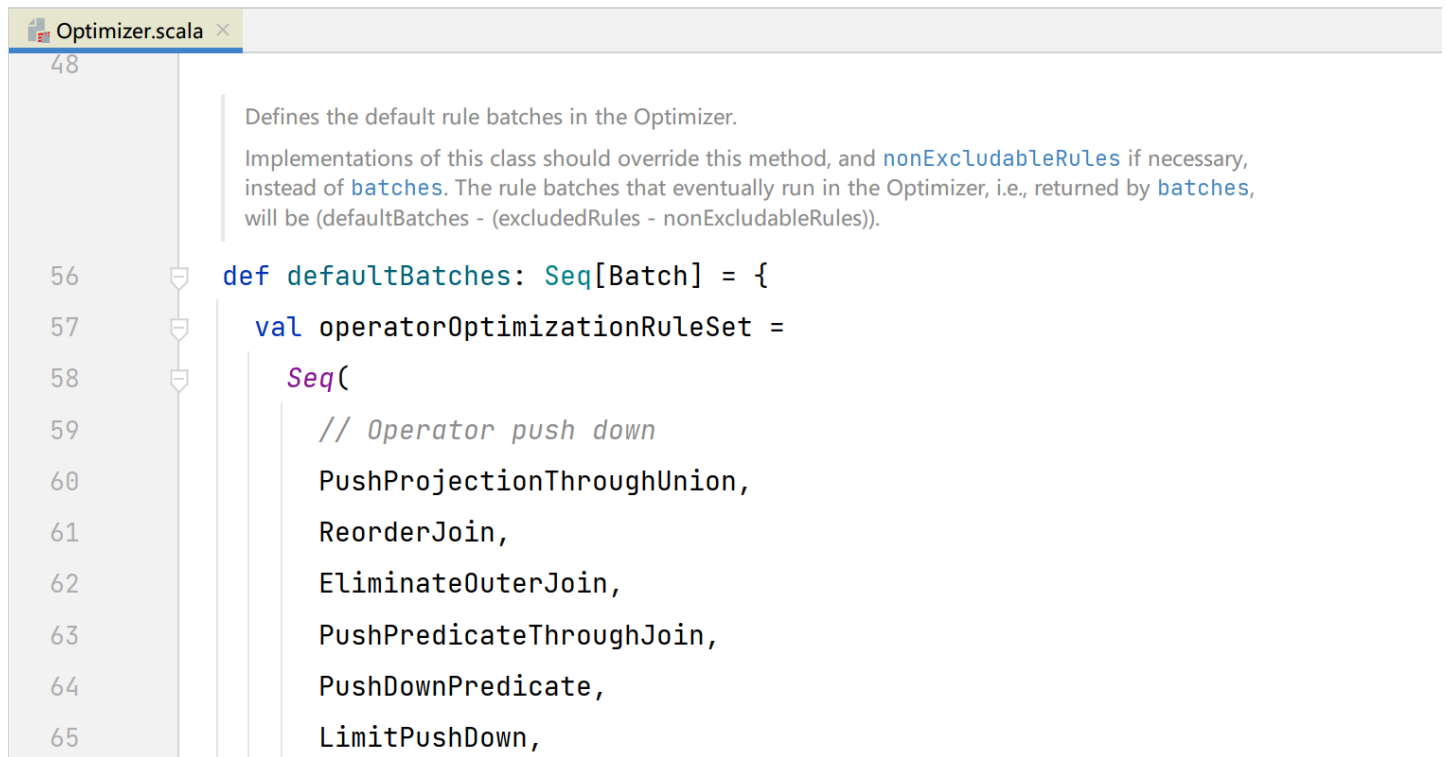
```

从上面代码可以看出，多个性质类似的 Rule 组成一个 Batch；而多个 Batch 构成一个 batches。这些 batches 会由 RuleExecutor 执行，先按一个一个 Batch 顺序执行，然后对 Batch 里面的每个 Rule 顺序执行。每个 Batch 会执行一次（Once）或多次（FixedPoint，由 spark.sql.optimizer.maxIterations 参数决定），执行过程如下：



5.4 Optimizer优化逻辑计划

优化器也是会定义一套Rules，利用这些Rule对逻辑计划和Expression进行迭代处理，从而使得树的节点进行合并和优化



```
Optimizer.scala x
48
    Defines the default rule batches in the Optimizer.
    Implementations of this class should override this method, and nonExcludableRules if necessary,
    instead of batches. The rule batches that eventually run in the Optimizer, i.e., returned by batches,
    will be (defaultBatches - (excludedRules - nonExcludableRules)).

56  def defaultBatches: Seq[Batch] = {
57      val operatorOptimizationRuleSet =
58          Seq(
59              // Operator push down
60              PushProjectionThroughUnion,
61              ReorderJoin,
62              EliminateOuterJoin,
63              PushPredicateThroughJoin,
64              PushDownPredicate,
65              LimitPushDown,
```

在前文的绑定逻辑计划阶段对 Unresolved LogicalPlan 进行相关 transform 操作得到了 Analyzed Logical Plan，这个 Analyzed Logical Plan 是可以直接转换成 Physical Plan 然后在spark中执行。但是如果直接这么弄的话，得到的 Physical Plan 很可能不是最优的，因为在实际应用中，很多低效的写法会带来执行效率的问题，需要进一步对Analyzed Logical Plan 进行处理，得到更优的逻辑算子树。于是，针对SQL 逻辑算子树的优化器 Optimizer 应运而生。

这个阶段的优化器主要是基于规则的（Rule-based Optimizer，简称 RBO），而绝大部分的规则都是启发式规则，也就是基于直观或经验而得出的规则，比如列裁剪（过滤掉查询不需要使用到的列）、谓词下推（将过滤尽可能地下沉到数据源端）、常量累加（比如 1 + 2 这种事先计算好）以及常量替换（比如 SELECT * FROM table WHERE i = 5 AND j = i + 3 可以转换成 SELECT * FROM table WHERE i = 5 AND j = 8）等等。

与绑定逻辑计划阶段类似，这个阶段所有的规则也是实现 Rule 抽象类，多个规则组成一个 Batch，多个 Batch 组成一个 batches，同样也是在 RuleExecutor 中进行执行。

核心源码骨架如下列截图所示：

67

Executes the batches of rules defined by the subclass. The batches are executed serially using the defined execution strategy. Within each batch, rules are also executed serially.

72

```
def execute(plan: TreeType): TreeType = {
```

73

```
  var curPlan = plan
```

74

```
  val queryExecutionMetrics = RuleExecutor.queryExecutionMeter
```

75

76

```
    batches.foreach { batch =>
```

77

```
      val batchStartPlan = curPlan
```

78

```
      var iteration = 1
```

79

```
      var lastPlan = curPlan
```

80

```
      var continue = true
```

Returns (defaultBatches - (excludedRules - nonExcludableRules)), the rule batches that eventually run in the Optimizer.

Implementations of this class should override `defaultBatches`, and `nonExcludableRules` if necessary, instead of this method.

244

```
final override def batches: Seq[Batch] = {
```

245

```
  val excludedRulesConf =
```

246

```
    SQLConf.get.optimizerExcludedRules.toSeq.flatMap(Utills.stringToSeq)
```

247

```
  val excludedRules = excludedRulesConf.filter { ruleName =>
```

248

```
    val nonExcludable = nonExcludableRules.contains(ruleName)
```

249

```
    if (nonExcludable) {
```

250

```
      logWarning(msg = s"Optimization rule '${ruleName}' was not excluded from the optimizer " +
```

251

```
        s"because this rule is a non-excludable rule.")
```

252

```
    }
```

253

```
    !nonExcludable
```

254

```
  }
```

255

```
  if (excludedRules.isEmpty) {
```

256

```
    defaultBatches
```

257

```
  } else {
```

258

```
    defaultBatches.flatMap { batch =>
```

259

```
      val filteredRules = batch.rules.filter { rule =>
```

RuleExecutor.scala x Optimizer.scala x

Implementations of this class should override this method, and `nonExcludableRules` if necessary, instead of `batches`. The rule batches that eventually run in the Optimizer, i.e., returned by `batches`, will be `(defaultBatches - (excludedRules - nonExcludableRules))`.

```

56 def defaultBatches: Seq[Batch] = {
57   val operatorOptimizationRuleSet =
58     Seq(
59       // Operator push down
60       PushProjectionThroughUnion,
61       ReorderJoin,
62       EliminateOuterJoin,
63       PushPredicateThroughJoin,
64       PushDownPredicate,
65       LimitPushDown,
66       ColumnPruning,
67       InferFiltersFromConstraints,
68       // Operator combine
69       CollapseRepartition,
70       CollapseProject,
71       CollapseWindow,
72       CombineFilters,
73       CombineLimits,
74       CombineUnions,
75       // Constant folding and strength reduction
76       NullPropagation,
77       ConstantPropagation,
78       FoldablePropagation,

```

Structure RuleExecutor.scala x Optimizer.scala x expressions.scala x Rule.scala x

529

Replaces `Expressions` that can be statically evaluated with equivalent `Literal` values. This rule is more specific with `Null` value propagation from bottom to top of the expression tree.

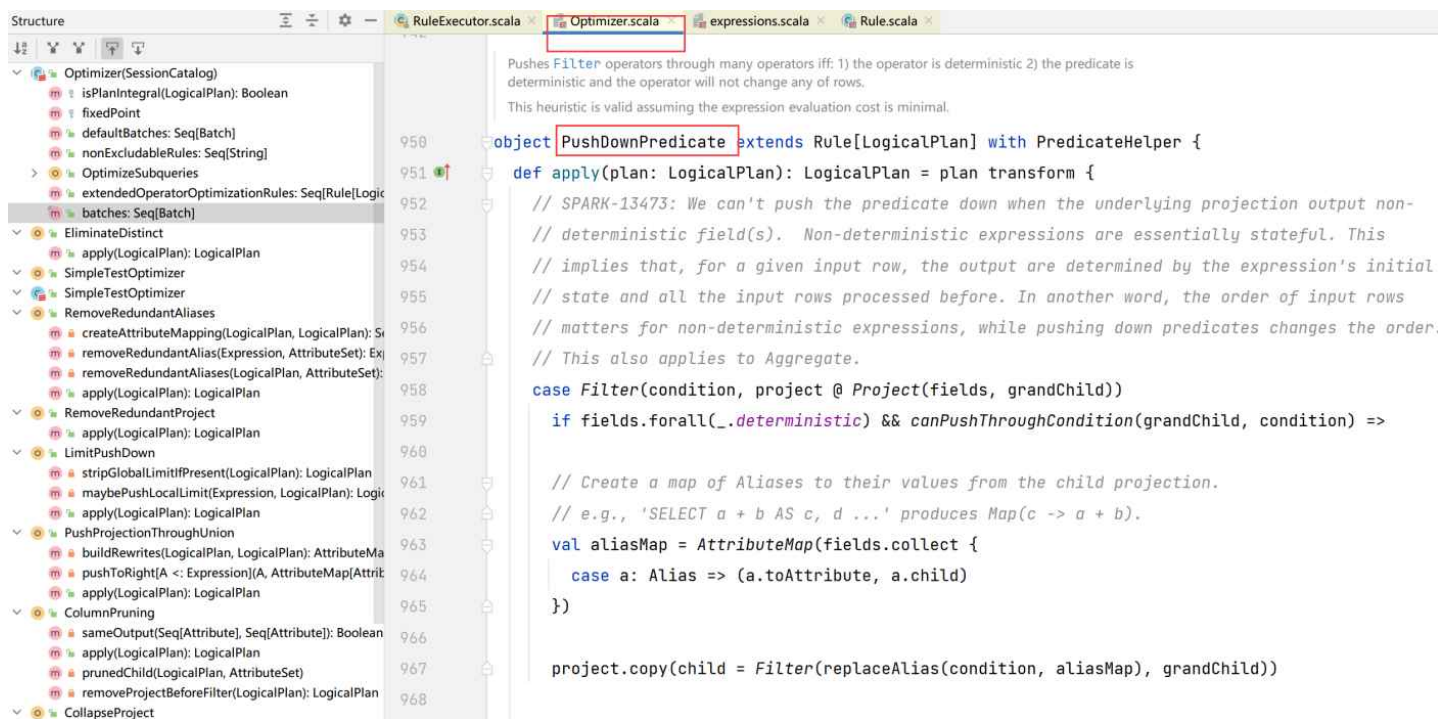
```

535 object NullPropagation extends Rule[LogicalPlan] {
536   private def isNullLiteral(e: Expression): Boolean = e match {
537     case Literal(null, _) => true
538     case _ => false
539   }
540
541   def apply(plan: LogicalPlan): LogicalPlan = plan transform {
542     case q: LogicalPlan => q transformExpressionsUp {
543       case e @ WindowExpression(Cast(Literal(0L, _), _, _, _), _) =>
544         Cast(Literal(0L), e.dataType, Option(SQLConf.get.sessionLocalTimeZone))
545       case e @ AggregateExpression(Count(exprs), _, _, _) if exprs.forall(isNullLiteral) =>
546         Cast(Literal(0L), e.dataType, Option(SQLConf.get.sessionLocalTimeZone))
547       case ae @ AggregateExpression(Count(exprs), _, false, _) if !exprs.exists(_.nullable) =>
548         // This rule should be only triggered when isDistinct field is false.
549         ae.copy(aggregateFunction = Count(Literal(1)))
550
551       case IsNull(c) if !c.nullable => Literal.create(false, BooleanType)
552       case IsNotNull(c) if !c.nullable => Literal.create(true, BooleanType)
553
554       case EqualNullSafe(Literal(null, _), r) => IsNull(r)
555       case EqualNullSafe(l, Literal(null, _)) => IsNull(l)
556
557       case AssertNotNull(c, _) if !c.nullable => c
558
559       // For Coalesce, remove null literals.
560       case e @ Coalesce(children) =>

```

Structure

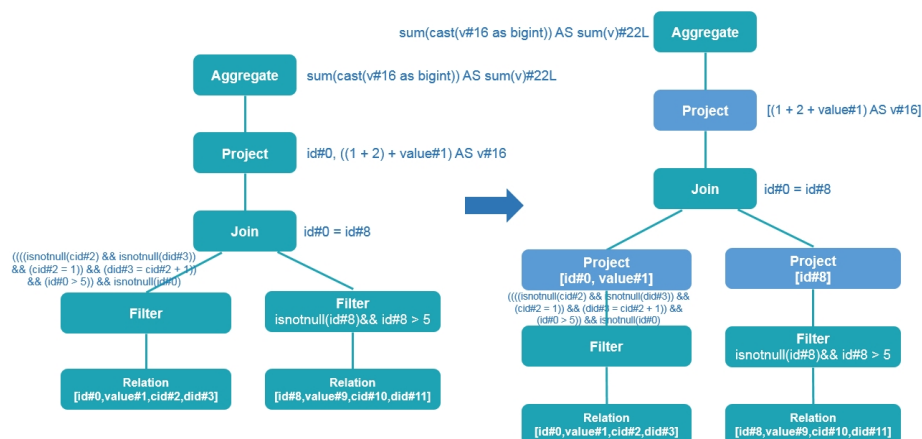
- ConstantFolding
 - apply(LogicalPlan): LogicalPlan
- ConstantPropagation
 - apply(LogicalPlan): LogicalPlan
 - EqualityPredicates
 - traverse(Expression, Boolean, Boolean): (Option[Expr], Boolean)
 - safeToReplace(AttributeReference, Boolean)
 - replaceConstants(Expression, EqualityPredicates): Expression
- ReorderAssociativeOperator
 - flattenAdd(Expression, ExpressionSet): Seq[Expression]
 - flattenMultiply(Expression, ExpressionSet): Seq[Expression]
 - collectGroupingExpressions(LogicalPlan): Expression
 - apply(LogicalPlan): LogicalPlan
- Optimize
 - apply(LogicalPlan): LogicalPlan
- BooleanSimplification
 - apply(LogicalPlan): LogicalPlan
- SimplifyBinaryComparison
 - apply(LogicalPlan): LogicalPlan
- SimplifyConditionals
 - falseOrNullLiteral(Expression): Boolean
 - apply(LogicalPlan): LogicalPlan
- LikeSimplification
 - startsWith
 - endsWith
 - startsWithAndEndsWith
 - contains
 - equalTo
 - apply(LogicalPlan): LogicalPlan
- NullPropagation
 - isNullLiteral(Expression): Boolean
 - apply(LogicalPlan): LogicalPlan
- FoldablePropagation
 - apply(LogicalPlan): LogicalPlan
 - propagateFoldables(LogicalPlan): (LogicalPlan, AttributeMap[Alias])
 - replaceFoldable(LogicalPlan, AttributeMap[Alias]): LogicalPlan
 - canPropagateFoldables(UnaryNode): Boolean
- SimplifyCasts
 - apply(LogicalPlan): LogicalPlan
- RemoveDispensableExpressions
 - apply(LogicalPlan): LogicalPlan
- SimplifyCaseConversionExpressions



那么针对前文的 SQL 语句，这个过程都会执行哪些优化呢？下文举例说明。

5.4.1谓词下推

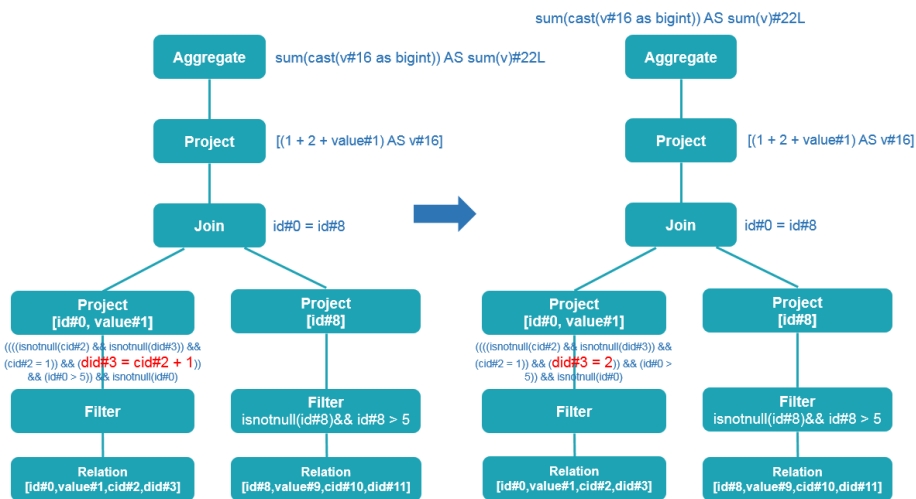
谓词下推在 Spark SQL 是由 PushDownPredicate 实现的，这个过程主要将过滤条件尽可能地下推到
底层，最好是数据源。上面介绍的 SQL，使用谓词下推优化得到的逻辑计划如下：



从上图可以看出，谓词下推将 Filter 算子直接下推到 Join 之前了（注意，上图是从下往上看）。也就是在扫描 t1 表的时候会先使用 (((isnotnull(cid#2) && isnotnull(did#3)) && (cid#2 = 1)) && (did#3 = 2)) && (id#0 > 50000)) && isnotnull(id#0) 过滤条件过滤出满足条件的数据；同时在扫描 t2 表的时候会先使用 isnotnull(id#8) && (id#8 > 50000) 过滤条件过滤出满足条件的数据。经过这样的操作，可以大大减少 Join 算子处理的数据量，从而加快计算速度。

5.4.2列裁剪

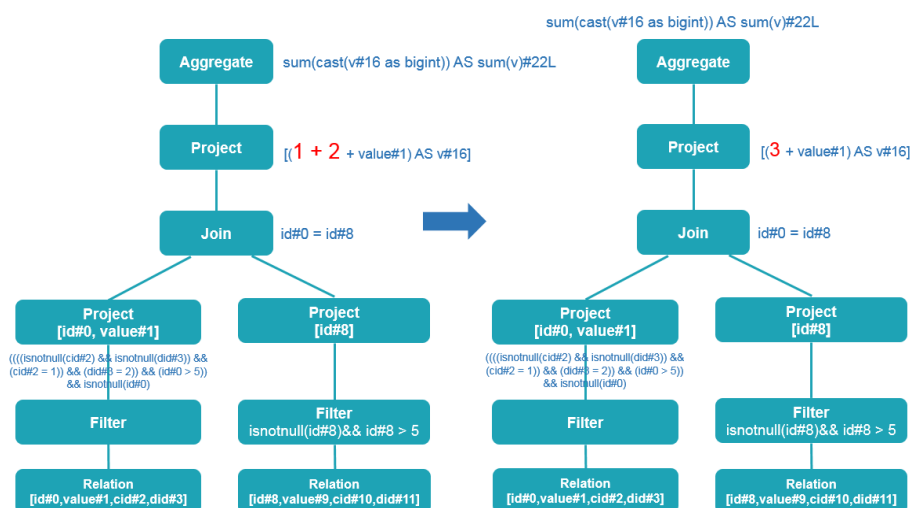
列裁剪在 Spark SQL 是由 ColumnPruning 实现的。因为我们查询的表可能有很多个字段，但是每次查询我们很可能不需要扫描出所有的字段，这个时候利用列裁剪可以把那些查询不需要的字段过滤掉，使得扫描的数据量减少。所以针对我们上面介绍的 SQL，使用列裁剪优化得到的逻辑计划如下：



从上图可以看出，经过列裁剪后，t1 表只需要查询 id 和 value 两个字段；t2 表只需要查询 id 字段。这样减少了数据的传输，而且如果底层的文件格式为列存（比如 Parquet），可以大大提高数据的扫描速度的。

5.4.3 常量替换

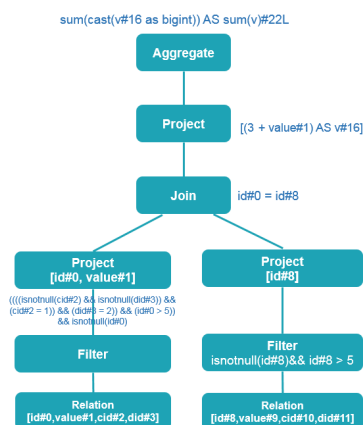
常量替换在 Spark SQL 是由 ConstantPropagation 实现的。也就是将变量替换成常量，比如 `SELECT * FROM table WHERE i = 5 AND j = i + 3` 可以转换成 `SELECT * FROM table WHERE i = 5 AND j = 8`。这个看起来好像没什么的，但是如果扫描的行数非常多可以减少很多的计算时间的开销的。经过这个优化，得到的逻辑计划如下：



我们的查询中有 `t1.cid = 1 AND t1.did = t1.cid + 1` 查询语句，从里面可以看出 t1.cid 其实已经是确定的值了，所以我们完全可以使用它计算出 t1.did。

5.4.4 常量累加

常量累加在 Spark SQL 是由 ConstantFolding 实现的。这个和常量替换类似，也是在这个阶段把一些常量表达式事先计算好。这个看起来改动的不大，但是在数据量非常大的时候可以减少大量的计算，减少 CPU 等资源的使用。经过这个优化，得到的逻辑计划如下：



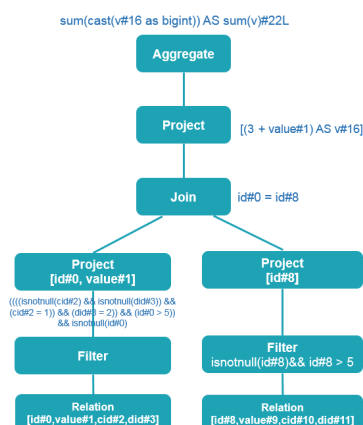
经过上面四个步骤的优化之后，得到的优化之后的逻辑计划为：

```

1 == Optimized Logical Plan ==
2 Aggregate [sum(cast(v#16 as bigint)) AS sum(v)#22L]
3 +- Project [(3 + value#1) AS v#16]
4   +- Join Inner, (id#0 = id#8)
5     :- Project [id#0, value#1]
6     : +- Filter (((isnotnull(cid#2) && isnotnull(did#3)) && (cid#2 = 1)) &&
7     :   +- Relation[id#0,value#1,cid#2,did#3] csv
8     +- Project [id#8]
9       +- Filter (isnotnull(id#8) && (id#8 > 5))
10        +- Relation[id#8,value#9,cid#10,did#11] csv

```

对应的图如下：



到这里，优化逻辑计划阶段就算完成了。另外，Spark 内置提供了多达70个优化 Rule，详情请参见 <https://github.com/apache/spark/blob/master/sql/catalyst/src/main/scala/org/apache/spark/sql/catalyst/optimizer/Optimizer.scala#L59>

5.5使用SparkPlanner生成物理计划

SparkSpanner使用Planning Strategies，对优化后的逻辑计划进行转换，生成可以执行的物理计划SparkPlan。

```
1  /**
2   * 将逻辑计划转成物理计划的抽象类。
3   * 各实现类通过各种GenericStrategy来生成各种可行的待选物理计划。
4   * 如一个策略无法对逻辑计划树的所有操作转换，则会调用[GenericStrategy#planLater planLa
5
6   * TODO: 目前为止，永远只生成一个物理计划
7   *       后续迭代中会对“多计划”予以实现
8   */
9  abstract class QueryPlanner[PhysicalPlan <: TreeNode[PhysicalPlan]] {
10    /** A list of execution strategies that can be used by the planner */
11    def strategies: Seq[GenericStrategy[PhysicalPlan]]
12
13    def plan(plan: LogicalPlan): Iterator[PhysicalPlan] = {
14      // 显然，此处还有大量工作需要做，可依然...
15
16      // 收集所有可选的物理计划。
17      val candidates = strategies.iterator.flatMap(_(plan))
18
19    abstract class SparkStrategies extends QueryPlanner[SparkPlan] {
20      self: SparkPlanner =>
21
22      /**
23       * Plans special cases of limit operators.
24       */
25      object SpecialLimits extends Strategy {
26
27
28      class SparkPlanner(
29        val sparkContext: SparkContext,
30        val conf: SQLConf,
31        val experimentalMethods: ExperimentalMethods)
32        extends SparkStrategies {
```

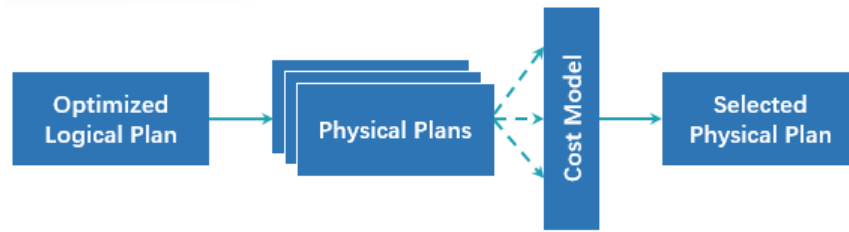
逻辑计划翻译成物理计划时，使用的是策略（Strategy）；

前面介绍的逻辑计划绑定和优化经过 Transformations 动作之后，树的类型并没有改变，

Logical Plan转化成物理计划后，树的类型改变了，由 Logical Plan 转换成 Physical Plan 了。

一个逻辑计划（Logical Plan）经过一系列的策略处理之后，得到多个物理计划（Physical Plans），物理计划在 Spark 是由 SparkPlan 实现的。

多个物理计划经过代价模型（Cost Model）得到选择后的物理计划（Selected Physical Plan），整个过程如下所示：



Cost Model 对应的就是基于代价的优化（Cost-based Optimizations, CBO，主要由华为的大佬们实现的，详见 SPARK-16026），核心思想是计算每个物理计划的代价，然后得到最优的物理计划。目前，这一部分并没有实现，直接返回多个物理计划列表的第一个作为最优的物理计划，如下：

```
1 lazy val sparkPlan: SparkPlan = {
2     SparkSession.setActiveSession(sparkSession)
3     // TODO: We use next(), i.e. take the first plan returned by the planner, he
4     //         but we will implement to choose the best plan.
5     planner.plan(ReturnAnswer(optimizedPlan)).next()
6 }
```

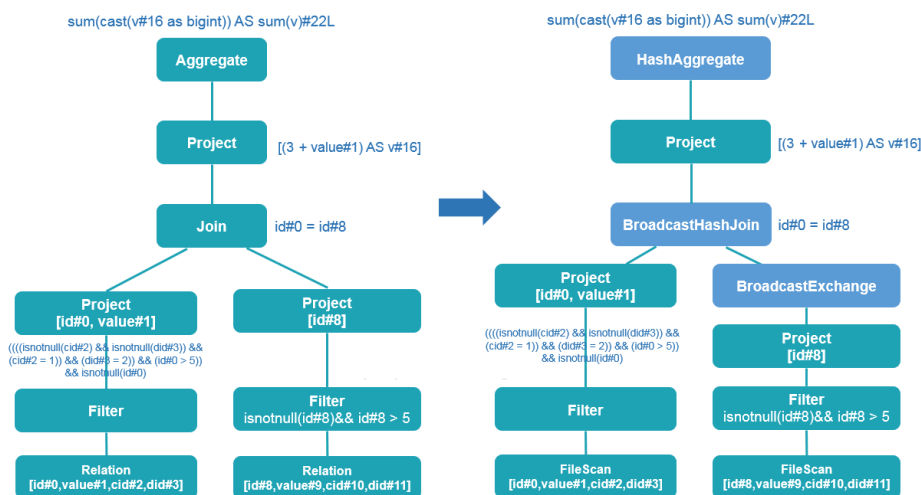
而 SPARK-16026 引入的 CBO 优化主要是在前面介绍的优化逻辑计划阶段 - Optimizer 阶段进行的，对应的 Rule 为 CostBasedJoinReorder，并且默认是关闭的，需要通过 `spark.sql.cbo.enabled` 或 `spark.sql.cbo.joinReorder.enabled` 参数开启。

所以到了这个节点，最后得到的物理计划如下：

```
1 == Physical Plan ==
2 *(3) HashAggregate(keys=[], functions=[sum(cast(v#16 as bigint))], output=[sum(v
3 +- Exchange SinglePartition
4     +- *(2) HashAggregate(keys=[], functions=[partial_sum(cast(v#16 as bigint))],
5         +- *(2) Project [(3 + value#1) AS v#16]
6             +- *(2) BroadcastHashJoin [id#0], [id#8], Inner, BuildRight
7                 :- *(2) Project [id#0, value#1]
8                     : +- *(2) Filter ((((((isnotnull(cid#2) && isnotnull(did#3)) && (cid
9                     :     +- *(2) FileScan csv [id#0,value#1,cid#2,did#3] Batched: false
10                +- BroadcastExchange HashedRelationBroadcastMode(List(cast(input[0,
11                    +- *(1) Project [id#8]
12                    +- *(1) Filter (isnotnull(id#8) && (id#8 > 5))
13                    +- *(1) FileScan csv [id#8] Batched: false, Format: CSV, Lo
```

从上面的结果可以看出，物理计划阶段已经知道数据源是从 csv 文件里面读取了，也知道文件的路径，数据类型等。而且在读取文件的时候，直接将过滤条件（PushedFilters）加进去了。

同时，这个 Join 变成了 BroadcastHashJoin，也就是将 t2 表的数据 Broadcast 到 t1 表所在的节点。图表示如下：



到这里，Physical Plan 就完全生成了。

5.6从物理执行计划获取inputRdd执行

从物理计划上，获取inputRdd

从物理计划上，生成全阶段代码，并编译反射出迭代器newBiIteator的Clazz

[真名：BufferedRowIteator]

然后将inputRDD做一个transformation得到最终要执行的rdd

```
1 inputRdd.mapPartitionsWithIndex((index,iter)=>{
2     new newBiIteator(){
3         hasNext(){
4             iter.hasNext
5         }
6         next(){
7             processNext(iter.next())
8         }
9     }
10 })
11
12 然后，对最后返回的rdd，执行你所需要的行动算子
13 rdd.collect().foreach(println)
```


6. Spark SQL实战

6.1 数据说明

数据集是货品交易数据集。

tbDate	tbStockDetail	tbStock
dateid: date 日期	ordernumber: varchar 订单号	ordernumber: varchar 订单号
years: varchar 年月	rownum: varchar 行号	locationid: varchar 交易位置
theyear: varchar 年	itemid: varchar 货品	dateid: date 交易日期
month: varchar 月	number: varchar 数量	
day: varchar 日	price: varchar 单价	
weekday: varchar 周几	amount: int 销售额	
week: varchar 第几周		
quarter: varchar 季度		
period: varchar 旬		
halfmonth: varchar 半月		

每个订单可能包含多个货品，每个订单可以产生多次交易，不同的货品有不同的单价。

6.2 加载数据

tbStock:

```
scala> case class tbStock(ordernumber:String,locationid:String,dateid:String) extends Serializable
```

```
defined class tbStock
```

```
scala> val tbStockRdd = spark.sparkContext.textFile("tbStock.txt")
```

```
tbStockRdd: org.apache.spark.rdd.RDD[String] = tbStock.txt MapPartitionsRDD[1] at textFile at <console>:23
```

```
scala> val tbStockDS =
```

```
tbStockRdd.map(_._split(",")).map(attr=>tbStock(attr(0),attr(1),attr(2))).toDS
```

```
tbStockDS: org.apache.spark.sql.Dataset[tbStock] = [ordernumber: string, locationid: string ... 1 more field]
```

```
scala> tbStockDS.show()
```

```
+-----+-----+-----+
```

```
| ordernumber | locationid | dataid |
```

```
+-----+-----+-----+
```

```
| BYSL00000893 | ZHAO | 2007-8-23 |
```

```
| BYSL00000897 | ZHAO | 2007-8-24 |
```

```
| BYSL00000898 | ZHAO | 2007-8-25 |
```

BYSL00000899	ZHAO 2007-8-26
BYSL00000900	ZHAO 2007-8-26
BYSL00000901	ZHAO 2007-8-27
BYSL00000902	ZHAO 2007-8-27
BYSL00000904	ZHAO 2007-8-28
BYSL00000905	ZHAO 2007-8-28
BYSL00000906	ZHAO 2007-8-28
BYSL00000907	ZHAO 2007-8-29
BYSL00000908	ZHAO 2007-8-30
BYSL00000909	ZHAO 2007-9-1
BYSL00000910	ZHAO 2007-9-1
BYSL00000911	ZHAO 2007-8-31
BYSL00000912	ZHAO 2007-9-2
BYSL00000913	ZHAO 2007-9-3
BYSL00000914	ZHAO 2007-9-3
BYSL00000915	ZHAO 2007-9-4
BYSL00000916	ZHAO 2007-9-4

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

only showing top 20 rows

tbStockDetail:

```
scala> case class tbStockDetail(ordernumber:String, rownum:Int, itemid:String, number:Int, price:Double, amount:Double) extends Serializable
```

```
defined class tbStockDetail
```

```
scala> val tbStockDetailRdd = spark.sparkContext.textFile("tbStockDetail.txt")
```

```
tbStockDetailRdd: org.apache.spark.rdd.RDD[String] = tbStockDetail.txt MapPartitionsRDD[13]  
at textFile at <console>:23
```

```
scala> val tbStockDetailDS = tbStockDetailRdd.map(_._split(",")).map(attr=>
tbStockDetail(attr(0),attr(1).trim().toInt,attr(2),attr(3).trim().toInt,attr(4).trim().toDouble,
attr(5).trim().toDouble)).toDS
```

```
tbStockDetailDS: org.apache.spark.sql.Dataset[tbStockDetail] = [ordernumber: string, rownum:
int ... 4 more fields]
```

```
scala> tbStockDetailDS.show()
```

```
+-----+-----+-----+-----+-----+
| ordernumber|rownum|  itemid|number|price|amount|
+-----+-----+-----+-----+-----+
| BYSL00000893|  0|FS527258160501| -1|268.0|-268.0|
| BYSL00000893|  1|FS527258169701|  1|268.0| 268.0|
| BYSL00000893|  2|FS527230163001|  1|198.0| 198.0|
| BYSL00000893|  3|24627209125406|  1|298.0| 298.0|
| BYSL00000893|  4|K9527220210202|  1|120.0| 120.0|
| BYSL00000893|  5|01527291670102|  1|268.0| 268.0|
| BYSL00000893|  6|QY527271800242|  1|158.0| 158.0|
| BYSL00000893|  7|ST040000010000|  8|  0.0|  0.0|
| BYSL00000897|  0|04527200711305|  1|198.0| 198.0|
| BYSL00000897|  1|MY627234650201|  1|120.0| 120.0|
| BYSL00000897|  2|01227111791001|  1|249.0| 249.0|
| BYSL00000897|  3|MY627234610402|  1|120.0| 120.0|
| BYSL00000897|  4|01527282681202|  1|268.0| 268.0|
| BYSL00000897|  5|84126182820102|  1|158.0| 158.0|
| BYSL00000897|  6|K9127105010402|  1|239.0| 239.0|
| BYSL00000897|  7|QY127175210405|  1|199.0| 199.0|
| BYSL00000897|  8|24127151630206|  1|299.0| 299.0|
| BYSL00000897|  9|G1126101350002|  1|158.0| 158.0|
| BYSL00000897| 10|FS527258160501|  1|198.0| 198.0|
| BYSL00000897| 11|ST040000010000| 13|  0.0|  0.0|
+-----+-----+-----+-----+-----+
```

only showing top 20 rows

tbDate:

```
scala> case class tbDate(dateid:String, years:Int, theyear:Int, month:Int, day:Int, weekday:Int,
week:Int, quarter:Int, period:Int, halfmonth:Int) extends Serializable
```

```
defined class tbDate
```

```
scala> val tbDateRdd = spark.sparkContext.textFile("tbDate.txt")
```

```
tbDateRdd: org.apache.spark.rdd.RDD[String] = tbDate.txt MapPartitionsRDD[20] at textFile at
<console>:23
```

```
scala> val tbDateDS = tbDateRdd.map(_._split(",")).map(attr=> tbDate(attr(0),attr(1).trim().toInt,
attr(2).trim().toInt,attr(3).trim().toInt, attr(4).trim().toInt, attr(5).trim().toInt, attr(6).trim().toInt,
attr(7).trim().toInt, attr(8).trim().toInt, attr(9).trim().toInt)).toDS
```

```
tbDateDS: org.apache.spark.sql.Dataset[tbDate] = [dateid: string, years: int ... 8 more fields]
```

```
scala> tbDateDS.show()
```

```
+-----+-----+-----+-----+---+-----+---+-----+-----+-----+
| dateid|years|theyear|month|day|weekday|week|quarter|period|halfmonth|
+-----+-----+-----+-----+---+-----+---+-----+-----+-----+
| 2003-1-1|200301| 2003| 1| 1| 3| 1| 1| 1| 1|
| 2003-1-2|200301| 2003| 1| 2| 4| 1| 1| 1| 1|
| 2003-1-3|200301| 2003| 1| 3| 5| 1| 1| 1| 1|
| 2003-1-4|200301| 2003| 1| 4| 6| 1| 1| 1| 1|
| 2003-1-5|200301| 2003| 1| 5| 7| 1| 1| 1| 1|
| 2003-1-6|200301| 2003| 1| 6| 1| 2| 1| 1| 1|
| 2003-1-7|200301| 2003| 1| 7| 2| 2| 1| 1| 1|
| 2003-1-8|200301| 2003| 1| 8| 3| 2| 1| 1| 1|
```

2003-1-9	200301	2003	1	9	4	2	1	1	1
2003-1-10	200301	2003	1	10	5	2	1	1	1
2003-1-11	200301	2003	1	11	6	2	1	2	1
2003-1-12	200301	2003	1	12	7	2	1	2	1
2003-1-13	200301	2003	1	13	1	3	1	2	1
2003-1-14	200301	2003	1	14	2	3	1	2	1
2003-1-15	200301	2003	1	15	3	3	1	2	1
2003-1-16	200301	2003	1	16	4	3	1	2	2
2003-1-17	200301	2003	1	17	5	3	1	2	2
2003-1-18	200301	2003	1	18	6	3	1	2	2
2003-1-19	200301	2003	1	19	7	3	1	2	2
2003-1-20	200301	2003	1	20	1	4	1	2	2

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

only showing top 20 rows

注册表：

```
scala> tbStockDS.createOrReplaceTempView("tbStock")
```

```
scala> tbDateDS.createOrReplaceTempView("tbDate")
```

```
scala> tbStockDetailDS.createOrReplaceTempView("tbStockDetail")
```

6.3 计算所有数据中每年的销售单数、销售总额

统计所有订单中每年的销售单数、销售总额

三个表连接后以count(distinct a.ordernumber)计销售单数，sum(b.amount)计销售总额

tbDate		tbStockDetail		tbStock	
dateid: date	日期	ordernumber: varchar	订单号	ordernumber: varchar	订单号
years: varchar	年月	rownum: varchar	行号	locationid: varchar	交易位置
theyear: varchar	年	itemid: varchar	货品	dateid: date	交易日期
month: varchar	月	number: varchar	数量		
day: varchar	日	price: varchar	单价		
weekday: varchar	周几	amount: int	销售额		
week: varchar	第几周				
quarter: varchar	季度				
period: varchar	旬				
halfmonth: varchar	半月				

```
SELECT c.theyear, COUNT(DISTINCT a.ordernumber), SUM(b.amount)
FROM tbStock a
JOIN tbStockDetail b ON a.ordernumber = b.ordernumber
JOIN tbDate c ON a.dateid = c.dateid
GROUP BY c.theyear
ORDER BY c.theyear
```

```
spark.sql("SELECT c.theyear, COUNT(DISTINCT a.ordernumber), SUM(b.amount) FROM tbStock a JOIN
tbStockDetail b ON a.ordernumber = b.ordernumber JOIN tbDate c ON a.dateid = c.dateid GROUP BY
c.theyear ORDER BY c.theyear").show
```

结果如下：

```
+-----+-----+-----+
|theyear|count(DISTINCT ordernumber)|    sum(amount)|
+-----+-----+-----+
| 2004|          1094| 3268115.499199999|
| 2005|          3828|1.3257564149999991E7|
| 2006|          3772|1.3680982900000006E7|
| 2007|          4885|1.6719354559999993E7|
| 2008|          4861| 1.4674295300000001E7|
| 2009|          2619| 6323697.189999999|
| 2010|           94| 210949.65999999997|
+-----+-----+-----+
```

6.4 查询每年最大金额的订单及其金额

目标：统计每年最大金额订单的销售额:

tbDate	tbStockDetail	tbStock
dateid: date 日期	ordernumber: varchar 订单号	ordernumber: varchar 订单号
years: varchar 年月	rownum: varchar 行号	locationid: varchar 交易位置
theyear: varchar 年	itemid: varchar 货品	dateid: date 交易日期
month: varchar 月	number: varchar 数量	
day: varchar 日	price: varchar 单价	
weekday: varchar 周几	amount: int 销售额	
week: varchar 第几周		
quarter: varchar 季度		
period: varchar 旬		
halfmonth: varchar 半月		

1. 统计每年，每个订单一共有多少销售额

```
SELECT a.dateid, a.ordernumber, SUM(b.amount) AS SumOfAmount
FROM tbStock a
JOIN tbStockDetail b ON a.ordernumber = b.ordernumber
GROUP BY a.dateid, a.ordernumber
```

```
spark.sql("SELECT a.dateid, a.ordernumber, SUM(b.amount) AS SumOfAmount FROM tbStock a JOIN
tbStockDetail b ON a.ordernumber = b.ordernumber GROUP BY a.dateid, a.ordernumber").show
```

2. 结果如下：

dateid	ordernumber	SumOfAmount
2008-4-9	BYSL00001175	350.0
2008-5-12	BYSL00001214	592.0
2008-7-29	BYSL00011545	2064.0
2008-9-5	DGSL00012056	1782.0
2008-12-1	DGSL00013189	318.0
2008-12-18	DGSL00013374	963.0
2009-8-9	DGSL00015223	4655.0

2009-10-5 DGSL00015585	3445.0
2010-1-14 DGSL00016374	2934.0
2006-9-24 GCSL00000673	3556.10000000000004
2007-1-26 GCSL00000826	9375.199999999999
2007-5-24 GCSL00001020	6171.3000000000002
2008-1-8 GCSL00001217	7601.6
2008-9-16 GCSL00012204	2018.0
2006-7-27 GHSL00000603	2835.6
2006-11-15 GHSL00000741	3951.94
2007-6-6 GHSL00001149	0.0
2008-4-18 GHSL00001631	12.0
2008-7-15 GHSL00011367	578.0
2009-5-8 GHSL00014637	1797.6
+-----+-----+-----+	

3.以上一步查询结果为基础表，和表tbDate使用dateid join，求出每年最大金额订单的销售额

```

SELECT theyear, MAX(c.SumOfAmount) AS SumOfAmount
FROM (SELECT a.dateid, a.ordernumber, SUM(b.amount) AS SumOfAmount
FROM tbStock a
JOIN tbStockDetail b ON a.ordernumber = b.ordernumber
GROUP BY a.dateid, a.ordernumber
) c
JOIN tbDate d ON c.dateid = d.dateid
GROUP BY theyear
ORDER BY theyear DESC

```

```

spark.sql("SELECT theyear, MAX(c.SumOfAmount) AS SumOfAmount FROM (SELECT a.dateid,
a.ordernumber, SUM(b.amount) AS SumOfAmount FROM tbStock a JOIN tbStockDetail b ON
a.ordernumber = b.ordernumber GROUP BY a.dateid, a.ordernumber ) c JOIN tbDate d ON c.dateid =
d.dateid GROUP BY theyear ORDER BY theyear DESC").show

```


4. 结果如下：

+-----+-----+	
theyear	SumOfAmount
+-----+-----+	
2010	13065.280000000002
2009	25813.200000000008
2008	55828.0
2007	159126.0
2006	36124.0
2005	38186.399999999994
2004	23656.79999999997
+-----+-----+	

6.5 计算每年最畅销货品

目标1：统计每年最畅销货品（哪个货品销售额amount在当年最高，哪个就是最畅销货品）

目标2：统计每年最畅销货品（哪个货品销售数量当年最高，哪个就是最畅销货品）

tbDate	tbStockDetail	tbStock
dateid: date 日期	ordernumber: varchar 订单号	ordernumber: varchar 订单号
years: varchar 年月	rownum: varchar 行号	locationid: varchar 交易位置
theyear: varchar 年	itemid: varchar 货品	dateid: date 交易日期
month: varchar 月	number: varchar 数量	
day: varchar 日	price: varchar 单价	
weekday: varchar 周几	amount: int 销售额	
week: varchar 第几周		
quarter: varchar 季度		
period: varchar 旬		
halfmonth: varchar 半月		

第一步、求出每年每个货品的销售额

```
SELECT c.theyear, b.itemid, SUM(b.amount) AS SumOfAmount
```

```
FROM tbStock a
JOIN tbStockDetail b ON a.ordernumber = b.ordernumber
JOIN tbDate c ON a.dateid = c.dateid
GROUP BY c.theyear, b.itemid
```

```
spark.sql("SELECT c.theyear, b.itemid, SUM(b.amount) AS SumOfAmount FROM tbStock a JOIN
tbStockDetail b ON a.ordernumber = b.ordernumber JOIN tbDate c ON a.dateid = c.dateid GROUP BY
c.theyear, b.itemid").show
```

结果如下：

```
+-----+-----+-----+
|theyear|  itemid|  SumOfAmount|
+-----+-----+-----+
| 2004|43824480810202|    4474.72|
| 2006|YA214325360101|     556.0|
| 2006|BT624202120102|     360.0|
| 2007|AK215371910101|24603.639999999992|
| 2008|AK216169120201|29144.199999999997|
| 2008|YL526228310106|16073.099999999999|
| 2009|KM529221590106| 5124.800000000001|
| 2004|HT224181030201|2898.6000000000004|
| 2004|SG224308320206|    7307.06|
| 2007|04426485470201|14468.800000000001|
| 2007|84326389100102|    9134.11|
| 2007|B4426438020201|    19884.2|
| 2008|YL427437320101|12331.799999999997|
| 2008|MH215303070101|    8827.0|
| 2009|YL629228280106|    12698.4|
| 2009|BL529298020602|    2415.8|
| 2009|F5127363019006|     614.0|
| 2005|24425428180101|   34890.74|
```

2007 YA214127270101	240.0
2007 MY127134830105	11099.92
+-----+-----+-----+-----+	

第二步：在第一步的基础上，统计每年单个货品中的最大金额

```

SELECT d.theyear, MAX(d.SumOfAmount) AS MaxOfAmount
FROM (SELECT c.theyear, b.itemid, SUM(b.amount) AS SumOfAmount
FROM tbStock a
JOIN tbStockDetail b ON a.ordernumber = b.ordernumber
JOIN tbDate c ON a.dateid = c.dateid
GROUP BY c.theyear, b.itemid
) d
GROUP BY d.theyear

```

```

spark.sql("SELECT d.theyear, MAX(d.SumOfAmount) AS MaxOfAmount FROM (SELECT c.theyear,
b.itemid, SUM(b.amount) AS SumOfAmount FROM tbStock a JOIN tbStockDetail b ON a.ordernumber =
b.ordernumber JOIN tbDate c ON a.dateid = c.dateid GROUP BY c.theyear, b.itemid ) d GROUP BY
d.theyear").show

```

结果如下：

+-----+-----+	
theyear	MaxOfAmount
+-----+-----+	
2007	70225.1
2006	113720.6
2004	53401.759999999995
2009	30029.2
2005	56627.329999999994

2010	4494.0
2008	98003.600000000003
+-----+	+-----+

第三步：用最大销售额和统计好的每个货品的销售额join，以及用年join，集合得到最畅销货品那一行信息

```
SELECT DISTINCT e.theyear, e.itemid, f.MaxOfAmount
FROM (SELECT c.theyear, b.itemid, SUM(b.amount) AS SumOfAmount
FROM tbStock a
JOIN tbStockDetail b ON a.ordernumber = b.ordernumber
JOIN tbDate c ON a.dateid = c.dateid
GROUP BY c.theyear, b.itemid
) e
JOIN (SELECT d.theyear, MAX(d.SumOfAmount) AS MaxOfAmount
FROM (SELECT c.theyear, b.itemid, SUM(b.amount) AS SumOfAmount
FROM tbStock a
JOIN tbStockDetail b ON a.ordernumber = b.ordernumber
JOIN tbDate c ON a.dateid = c.dateid
GROUP BY c.theyear, b.itemid
) d
GROUP BY d.theyear
) f ON e.theyear = f.theyear
AND e.SumOfAmount = f.MaxOfAmount
ORDER BY e.theyear
```

```
spark.sql("SELECT DISTINCT e.theyear, e.itemid, f.maxofamount FROM (SELECT c.theyear, b.itemid, SUM(b.amount) AS sumofamount FROM tbStock a JOIN tbStockDetail b ON a.ordernumber = b.ordernumber JOIN tbDate c ON a.dateid = c.dateid GROUP BY c.theyear, b.itemid ) e JOIN (SELECT d.theyear, MAX(d.sumofamount) AS maxofamount FROM (SELECT c.theyear, b.itemid, SUM(b.amount) AS sumofamount FROM tbStock a JOIN tbStockDetail b ON a.ordernumber = b.ordernumber JOIN
```

```
tbDate c ON a.dateid = c.dateid GROUP BY c.theyear, b.itemid ) d GROUP BY d.theyear ) f ON e.theyear = f.theyear AND e.sumofamount = f.maxofamount ORDER BY e.theyear").show
```

结果如下：

```
+-----+-----+-----+
|theyear|  itemid|  maxofamount|
+-----+-----+-----+
| 2004|JY424420810101|53401.759999999995|
| 2005|24124118880102|56627.329999999994|
| 2006|JY425468460101|    113720.6|
| 2007|JY425468460101|    70225.1|
| 2008|E2628204040101|98003.600000000003|
| 2009|YL327439080102|    30029.2|
| 2010|SQ429425090101|    4494.0|
+-----+-----+-----+
```

7. SparkSQL整合Hive

sparksql可以使用hive的元数据库，如果没有，sparksql也可以自己创建。

1. 在mysql创建一个普通用户（也可以使用root用户）

```
1 # 创建一个普通用户，并且授权
2 CREATE USER 'spark'@'%' IDENTIFIED BY 'DoIt123!@#';
3 GRANT ALL PRIVILEGES ON hivedb.* TO 'spark'@'%' IDENTIFIED BY 'DoIt123!@#' WITH
4 FLUSH PRIVILEGES;
```

2. 添加一个hive-site.xml到spark的conf目录，里面的内容如下：

```
1 <?xml version="1.0" encoding="UTF-8" standalone="no"?>
```

```

2 <?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
3 <configuration>
4   <property>
5     <name>javax.jdo.option.ConnectionURL</name>
6     <value>jdbc:mysql://node-1.51doit.cn:3306/hivedb?createDatabaseIfNotExis
7     <description>JDBC connect string for a JDBC metastore</description>
8   </property>
9
10  <property>
11    <name>javax.jdo.option.ConnectionDriverName</name>
12    <value>com.mysql.jdbc.Driver</value>
13    <description>Driver class name for a JDBC metastore</description>
14  </property>
15
16  <property>
17    <name>javax.jdo.option.ConnectionUserName</name>
18    <value>spark</value>
19    <description>username to use against metastore database</description>
20  </property>
21
22  <property>
23    <name>javax.jdo.option.ConnectionPassword</name>
24    <value>DoIt123!@#</value>
25    <description>password to use against metastore database</description>
26  </property>
27
28  <property>
29    <name>hive.metastore.schema.verification</name>
30    <value>>false</value>
31  </property>
32  <property>
33    <name>datanucleus.schema.autoCreateAll</name>
34    <value>true</value>
35  </property>
36  <property>
37    <name>hive.metastore.warehouse.dir</name>
38    <value>hdfs://node-1.51doit.cn:9000/user/hive/warehouse</value>
39  </property>
40 </configuration>
41

```

3. 上传一个mysql连接驱动,可以将连接驱动放入到spark的安装包的jars或者使用--driver-class-path指定mysql连接驱动的位置

```
1 bin/spark-sql --master spark://node-4:7077,node-5:7077 --driver-class-path
```

```
/root/mysql-connector-java-5.1.47.jar
```

4. 重新启动SparkSQL的命令行

```
1 bin/spark-sql --master spark://node-1.51doit.cn:7077 --driver-class-path /root/m
```

Spark SQL也提供JDBC连接支持，这对于让商业智能(BI)工具连接到Spark集群上以及在多用户间共享一个集群的场景都非常有用。JDBC 服务器作为一个独立的Spark 驱动器程序运行，可以在多用户之间共享。任意一个客户端都可以在内存中缓存数据表，对表进行查询。集群的资源以及缓存数据都在所有用户之间共享。

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”。

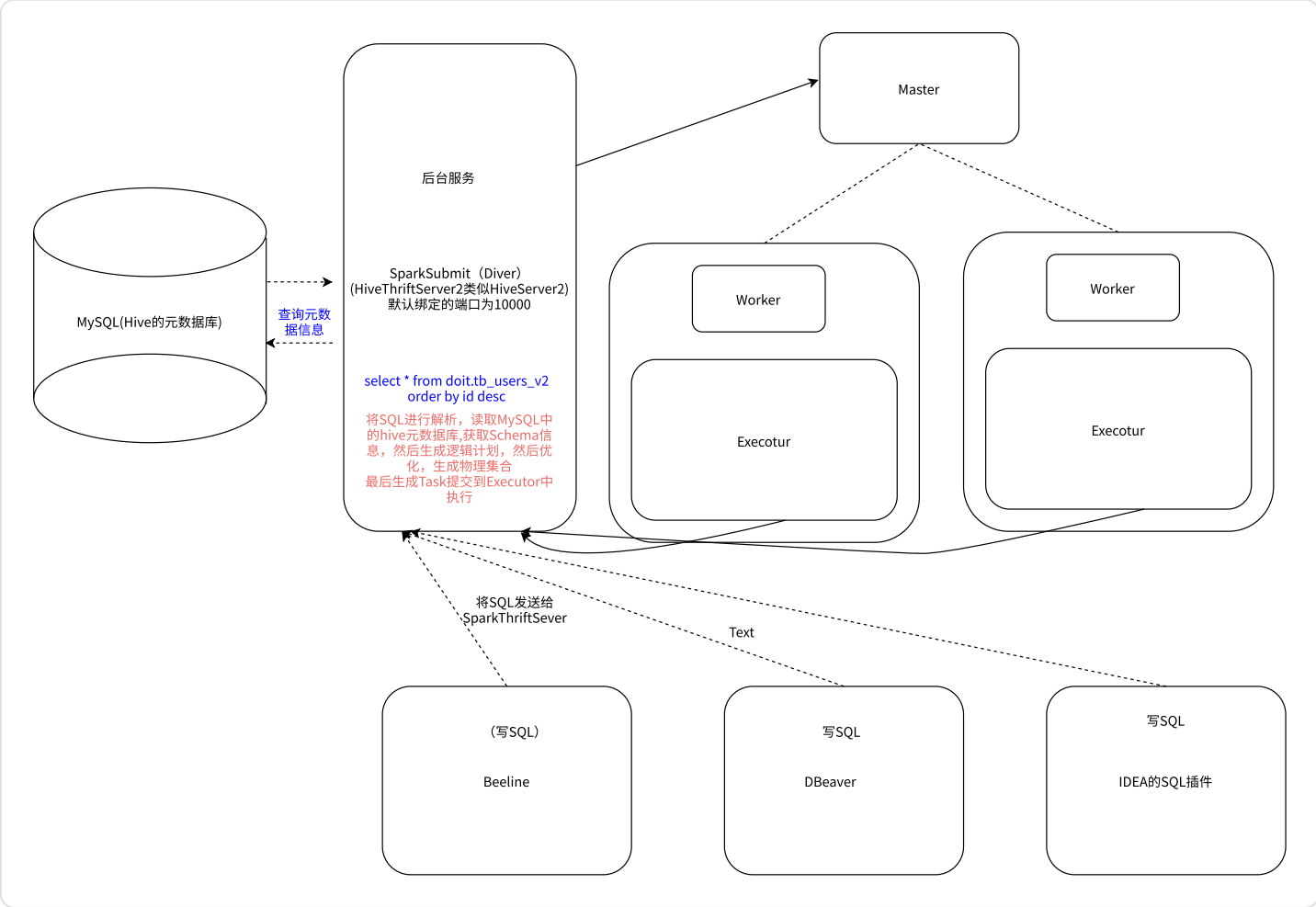
服务器可以通过 Spark 目录中的 sbin/start-thriftserver.sh 启动。这个 脚本接受的参数选项大多与 spark-submit 相同。默认情况下，服务器会在 [localhost:10000](#) 上进行监听，我们可以通过环境变量(HIVE_SERVER2_THRIFT_PORT 和 HIVE_SERVER2_THRIFT_BIND_HOST)修改这些设置，也可以通过 Hive配置选项(hive.server2.thrift.port 和 hive.server2.thrift.bind.host)来修改。

你也可以通过命令行参数：--hiveconf property=value来设置Hive选项。

在 Beeline 客户端中，你可以使用标准的 HiveQL 命令来创建、列举以及查询数据表。

```
1 # spark-sql 启动HiveServer2
2
3 #stand alone 模式
4 sbin/start-thriftserver.sh --master spark://node-1.51doit.cn:7077 --executor-mem
5
6 # on yarn 模式
7 sbin/start-thriftserver.sh --master yarn --deploy-mode client --driver-memory 2g
8
```

Spark的ThriftServer的原理（类似HiveServer2服务）



Workers (2)

Worker Id	Address	State	Cores	Memory	Resources
worker-20230106211444-172.16.100.103-46690	172.16.100.103:46690	ALIVE	2 (2 Used)	1024.0 MiB (1024.0 MiB Used)	
worker-20230106211445-172.16.100.102-38357	172.16.100.102:38357	ALIVE	2 (2 Used)	1024.0 MiB (1024.0 MiB Used)	

Running Applications (1)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
app-20230106223858-0002 (kill)	Thrift JDBC/ODBC Server	4	1024.0 MiB		2023/01/06 22:38:58	root	RUNNING	15 s

Completed Applications (2)

启动beeline客户端连接ThriftServer

```
1
2 #使用beline连接HiveServer
3
4 bin/beeline -u jdbc:hive2://node-1.51doit.cn:10000 -n root
```