



# SparkSQL

在Spark上运用SQL处理结构化数据



黑马程序员 | 传智教育旗下  
高端IT教育品牌  
[www.itheima.com](http://www.itheima.com)



# 学习目标

Learning Objectives

1. 了解SparkSQL框架模块的基础概念和发展历史
2. 掌握SparkSQL DataFrame API开发
3. 理解SparkSQL的运行流程
4. 掌握SparkSQL和Hive的集成



# 目录

Contents

- ◆ 第一章 快速入门
- ◆ 第二章 SparkSQL概述
- ◆ 第三章 DataFrame入门和操作
- ◆ 第四章 SparkSQL函数定义
- ◆ 第五章 SparkSQL的运行流程
- ◆ 第六章 SparkSQL整合Hive
- ◆ 第七章 分布式SQL引擎配置



# SparkSQL快速入门

- 1.1 什么是SparkSQL
- 1.2 为什么要学习SparkSQL
- 1.3 SparkSQL特点
- 1.4 SparkSQL发展历史



## 1.1 什么是SparkSQL

[Download](#)[Libraries](#) ▾[Documentation](#) ▾[Examples](#)[Community](#) ▾[FAQ](#)

**Spark SQL** is Spark's module for working with structured data.

SparkSQL 是Spark的一个模块, 用于处理海量结构化数据

限定: 结构化数据处理



## 1.2 为什么学习SparkSQL



多挣钱



SparkSQL是非常成熟的 海量结构化数据处理框架。

学习SparkSQL主要在2个点：

- SparkSQL本身十分优秀, 支持SQL语言\性能强\可以自动优化\API简单\兼容HIVE等等
- 企业大面积在使用SparkSQL处理业务数据
  - 离线开发
  - 数仓搭建
  - 科学计算
  - 数据分析



## 1.3 SparkSQL的特点



### 融合性

1 SQL可以无缝集成在代码中,随时用SQL处理数据

### 统一数据访问

2 一套标准API可读写不同数据源

### Hive兼容

3 可以使用SparkSQL直接计算并生成Hive数据表

### 标准化连接

4 支持标准化JDBC\ODBC连接,方便和各种数据库进行数据交互.

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

Apply functions to results of SQL queries.



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

## 1.4 SparkSQL发展历史 - 前身 Shark框架

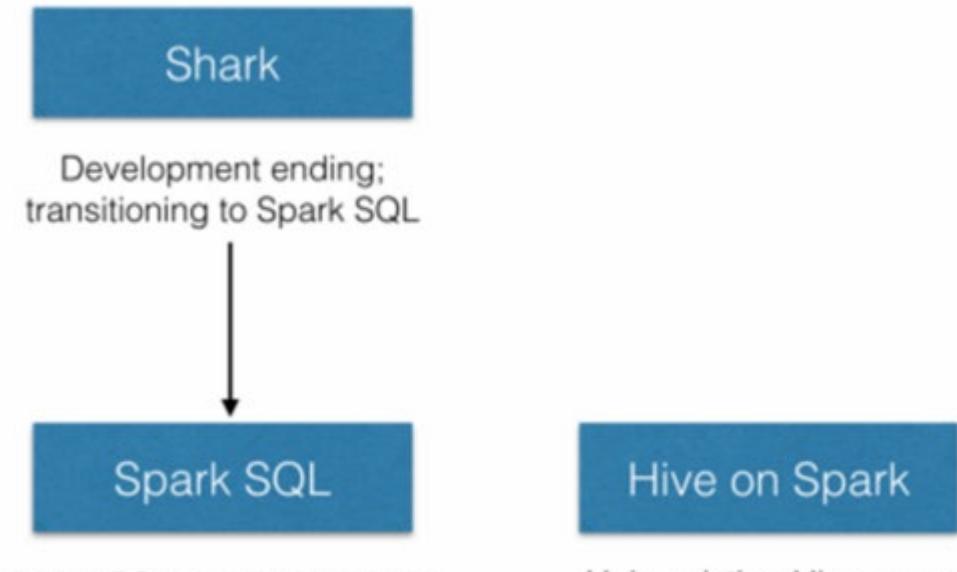
在许多年前(2012\2013左右)Hive逐步火热起来, 大片抢占分布式SQL计算市场

Spark作为通用计算框架, 也不可能放弃这一细分领域.  
于是, Spark官方模仿Hive推出了Shark框架(Spark 0.9版本)

Shark框架是几乎100%模仿Hive, 内部的配置项\优化项等都是直接模仿而来.不同的在于将执行引擎由MapReduce更换为了Spark.

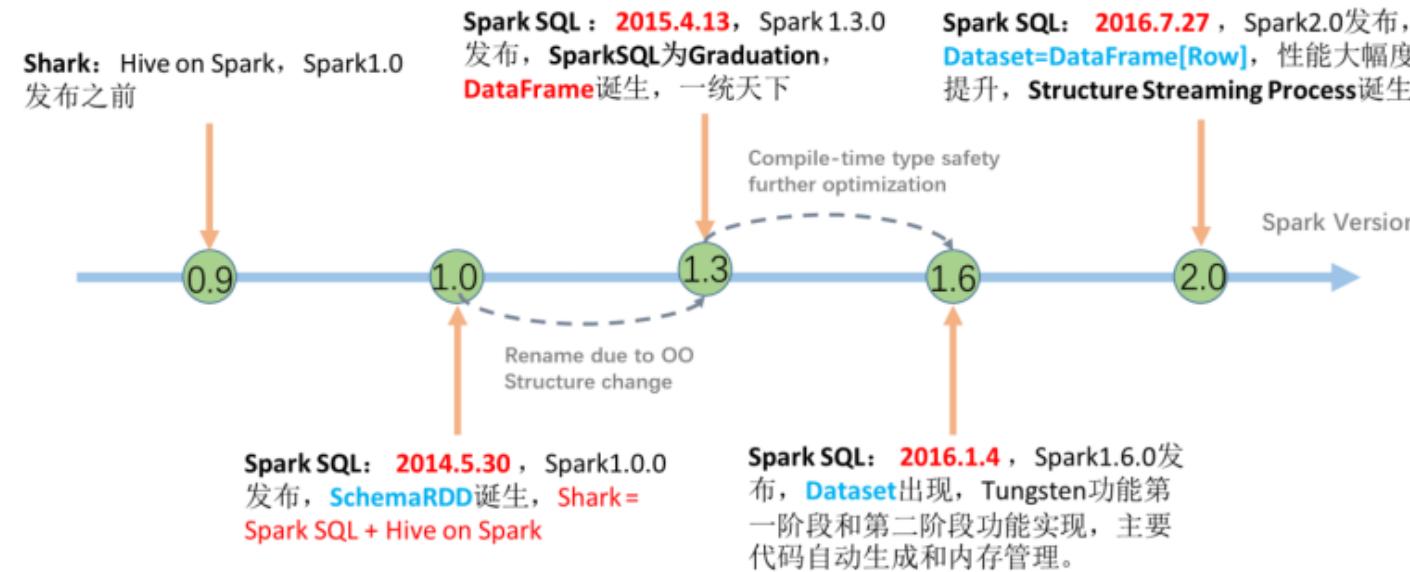
因为Shark框架太模仿Hive, Hive是针对MR优化, 很多地方和SparkCore(RDD)水土不服, 最终被放弃

Spark官方下决心开发一个自己的分布式SQL引擎 也就是诞生了现在的SparkSQL





## 1.4 SparkSQL发展历史



- 2014年 1.0正式发布
- 2015年 1.3 发布DataFrame数据结构, 沿用至今
- 2016年 1.6 发布Dataset数据结构(带泛型的DataFrame), 适用于支持泛型的语言(Java\Scala)
- 2016年 2.0 统一了Dataset 和 DataFrame, 以后只有Dataset了, Python用的DataFrame就是 没有泛型的Dataset
- 2019年 3.0 发布, 性能大幅度提升, SparkSQL变化不大



## 总结

1. SparkSQL用于处理大规模结构化数据的计算引擎
2. SparkSQL在企业中广泛使用，并性能极好，学习它不管是工作还是就业都有很大帮助
3. SparkSQL：使用简单、API统一、兼容HIVE、支持标准化JDBC和ODBC连接
4. SparkSQL 2014年正式发布，当下使用最多的2.0版  
Spark发布于2016年，当下使用的最新3.0版发布于2019年

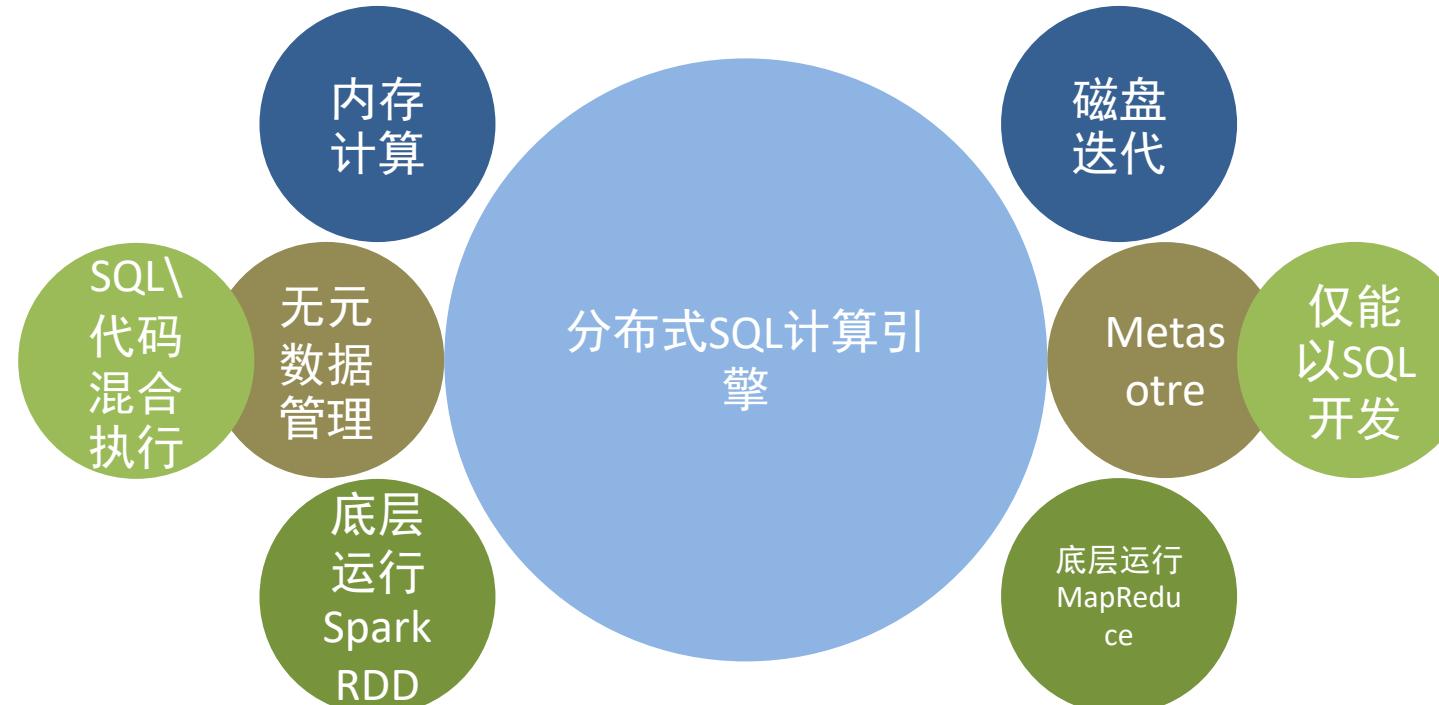


## SparkSQL 概述

- 2.1 SparkSQL和Hive的异同
- 2.2 SparkSQL的数据抽象
- 2.3 SparkSQL数据抽象的发展
- 2.4 DataFrame数据抽象
- 2.6 SparkSession对象
- 2.7 SparkSQL HelloWorld



## 2.1 SparkSQL和Hive的异同



Spark

Hive

都可以运行在YARN之上

**Hive和Spark 均是：“分布式SQL计算引擎”**

**均是构建大规模结构化数据计算的绝佳利器，同时SparkSQL拥有更好的性能。**

**目前，企业中使用Hive仍旧居多，但SparkSQL将会在很近的未来替代Hive成为分布式SQL计算市场的顶级**



## 2.2 SparkSQL的数据抽象

Pandas

• DataFrame

SparkCore

• RDD

SparkSQL

• DataFrame

Pandas - DataFrame

- 二维表数据结构
- 单机（本地）集合

SparkCore - RDD

- 无标准数据结构，存储什么数据均可
- 分布式集合（分区）

SparkSQL - DataFrame

- 二维表数据结构
- 分布式集合（分区）



## 2.2 SparkSQL的数据抽象

SparkSQL  
For JVM

SparkSQL  
For  
Python\R

• Dataset

• DataFrame

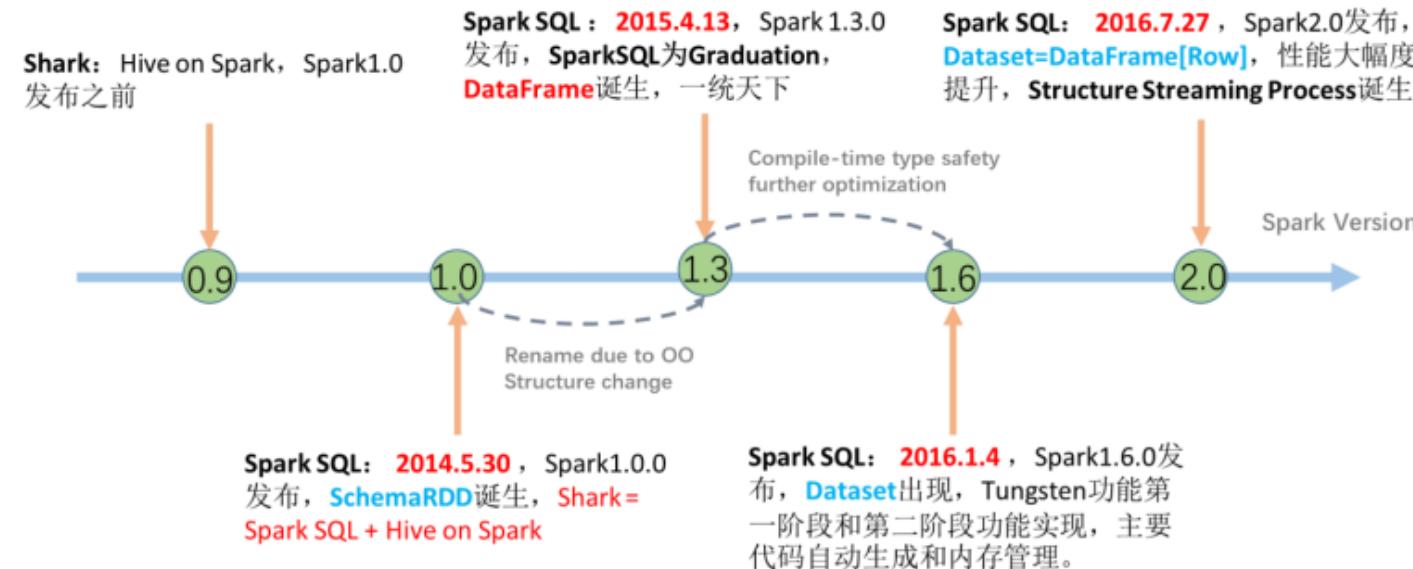
SparkSQL 其实有3类数据抽象对象

- SchemaRDD对象（已废弃）
- DataSet对象：可用于Java、Scala语言
- DataFrame对象：可用于Java、Scala、Python、R

我们以Python开发SparkSQL，主要使用的就是  
DataFrame对象作为核心数据结构



## 2.3 SparkSQL数据抽象的发展

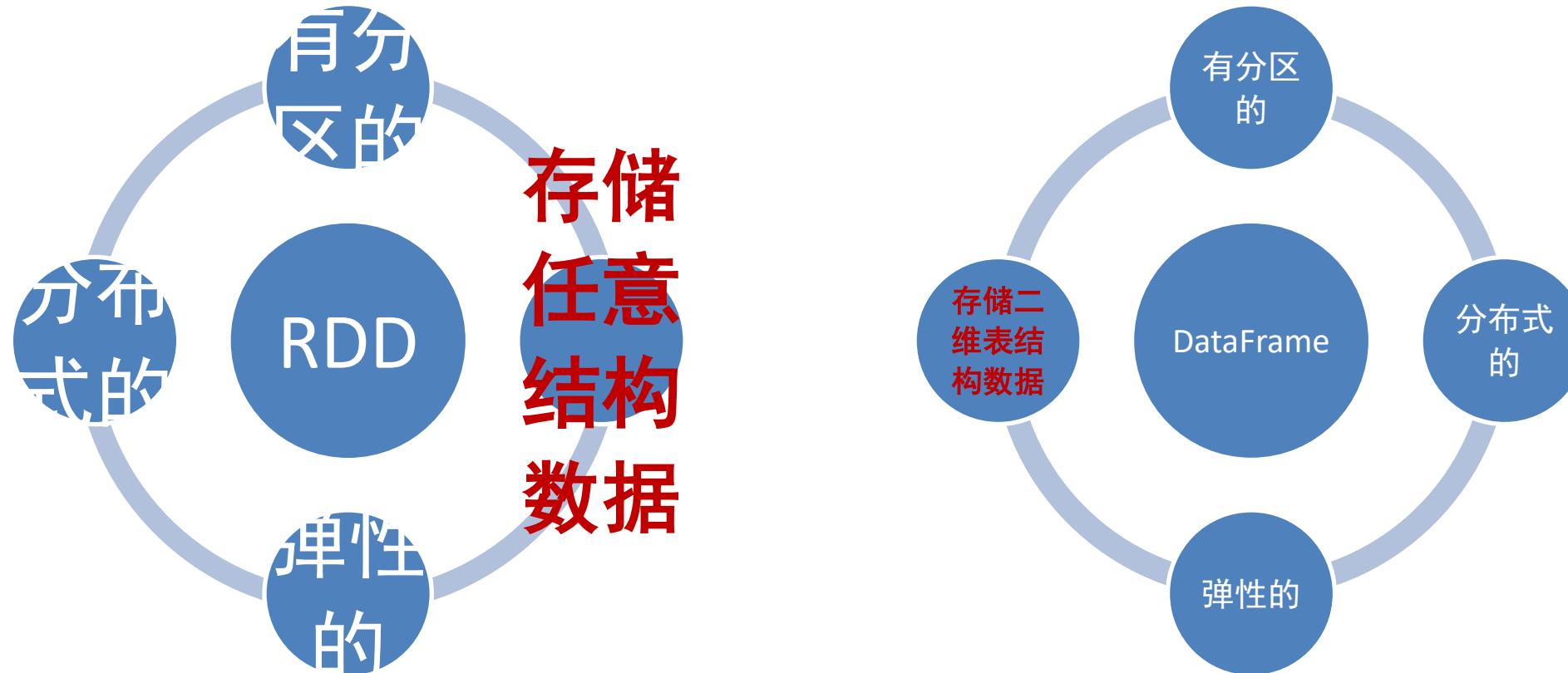


从SparkSQL的发展历史可以看到：

- 14年最早的数据抽象是：SchemaRDD（内部存储二维表数据结构的RDD），SchemaRDD就是魔改的RDD，将RDD支持的存储数据，限定为二维表数据结构用以支持SQL查询。由于是魔改RDD，只是一个过渡产品，现已废弃。
- 15年发布DataFrame对象，基于Pandas的DataFrame（模仿）独立于RDD进行实现，将数据以二维表结构进行存储并支持分布式运行
- 16年发布DataSet对象，在DataFrame之上添加了泛型的支持，用以更好的支持Java和Scala这两个支持泛型的编程语言
- 16年，Spark2.0版本，将DataFrame和DataSet进行合并。其底层均是DataSet对象，但在Python和R语言到用时，显示为DataFrame对象。和老的DataFrame对象没有区别



## 2.4 DataFrame概述



DataFrame和RDD都是：弹性的、分布式的、数据集  
只是，DataFrame存储的数据结构“限定”为：二维表结构化数  
据  
而RDD可以存储的数据则没有任何限制，想处理什么就处理什么



## 2.4 DataFrame概述

假定有如下数据集

id	name	age
1	张三	11
2	李四	11
3	王五	11
4	赵六	9
5	王潇	11

# DataFram

e

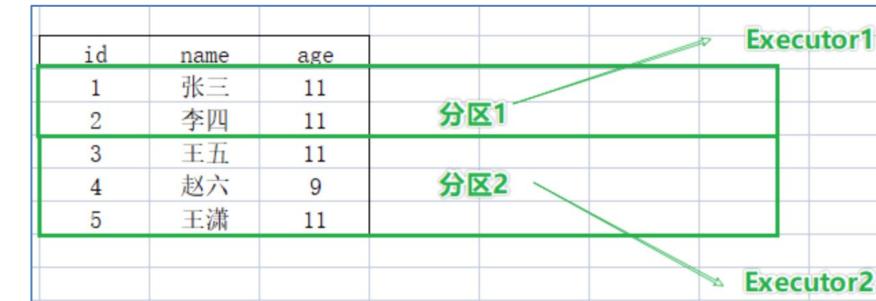
按二维表格存储

# RDD

按数组对象存储

DataFrame 是按照二维表格的形式存储数据

RDD则是存储对象本身





思考

id	name	age	
1	张三	11	
2	李四	11	
3	王五	11	
4	赵六	9	
5	王潇	11	

分区1

分区2

Executor1

Executor2

[1, 张三, 11]	分区1
[2, 李四, 11]	
[3, 王五, 11]	
[4, 赵六, 9]	
[5, 王潇, 11]	分区2

Executor1

Executor2

谁更适合用SQL进行处理？



## 2.5 SparkSession对象

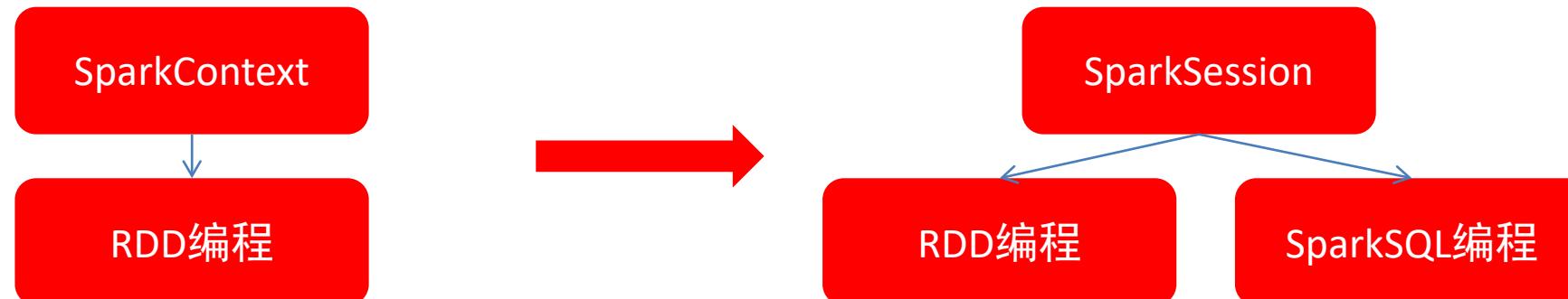
在RDD阶段，程序的执行入口对象是： SparkContext

在Spark 2.0后，推出了SparkSession对象，作为Spark编码的统一入口对象。

SparkSession对象可以：

- 用于SparkSQL编程作为入口对象
- 用于SparkCore编程，可以通过SparkSession对象中获取到SparkContext

所以，我们后续的代码，执行环境入口对象，统一变更为SparkSession对象





## 2.5 SparkSession对象

现在，来体验一下构建执行环境入口对象：SparkSession

构建SparkSession核心代码

```
# coding:utf8

# SparkSQL 中的入口对象是SparkSession对象
from pyspark.sql import SparkSession

if __name__ == '__main__':
    # 构建SparkSession对象, 这个对象是 构建器模式 通过builder方法来构建
    spark = SparkSession.builder.\
        appName("local[*]").\
        config("spark.sql.shuffle.partitions", "4").\
        getOrCreate()
    # appName 设置程序名称, config设置一些常用属性
    # 最后通过getOrCreate()方法 创建SparkSession对象
```



## 2.6 SparkSQL HelloWorld 演示

有如下数据集：列1ID，列2学科，列3分数

```
26,数学,96
27,数学,96
28,数学,96
29,数学,96
30,数学,96
1,英语,99
2,英语,99
3,英语,99
```

数据集文件：

资料\data\sql\stu\_score.txt

注意：右侧代码同学们不需要练习，只是先体验一下SparkSQL的数据处理

需求：读取文件，找出学科为“语文”的数据，并限制输出5条

where subject = '语文' limit 5

代码如下：

```
# coding:utf8
# SparkSQL 中的入口对象是SparkSession对象
from pyspark.sql import SparkSession
if __name__ == '__main__':
    # 构建SparkSession对象, 这个对象是 构建器模式 通过builder方法来构建
    spark = SparkSession.builder.\
        appName("local[*]").\
        config("spark.sql.shuffle.partitions", "4").\
        getOrCreate()
    # appName 设置程序名称, config设置一些常用属性
    # 最后通过getOrCreate()方法 创建SparkSession对象

    df = spark.read.csv('../data/sql/stu_score.txt', sep=',', header=False)
    df2 = df.toDF('id', 'name', 'score')
    df2.printSchema()
    df2.show()
    df2.createTempView("score")

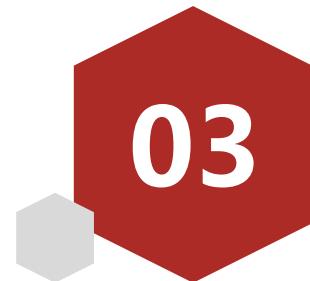
    # SQL 风格
    spark.sql("""
        SELECT * FROM score WHERE name='语文' LIMIT 5
    """).show()

    # DSL 风格
    df2.where("name='语文'").limit(5).show()
```



## 总结

1. SparkSQL 和 Hive同样，都是用于大规模SQL分布式计算的计算框架，均可以运行在YARN之上，在企业中广泛被应用
2. SparkSQL的数据抽象为： SchemaRDD（废弃）、 DataFrame（Python、 R、 Java、 Scala） 、 DataSet（Java、 Scala）。
3. DataFrame同样是分布式数据集，有分区可以并行计算，和RDD不同的是， DataFrame中存储的数据结构是以表格形式组织的，方便进行SQL计算
4. DataFrame对比DataSet基本相同，不同的是DataSet支持泛型特性，可以让Java、 Scala语言更好的利用到。
5. SparkSession是2.0后退出的新执行环境入口对象，可以用于RDD、 SQL等编程



# DataFrame入门

- 3.1 DataFrame的组成
- 3.2 DataFrame的代码构建
- 3.3 DataFrame的入门操作
- 3.4 词频统计案例
- 3.5 电影数据分析
- 3.6 SparkSQL Shuffle 分区数目
- 3.7 SparkSQL 数据清洗API
- 3.8 DataFrame数据写出
- 3.9 DataFrame 通过JDBC读写数据库（MySQL示例）



### 3.1 DataFrame的组成

DataFrame是一个二维表结构，那么表格结构就有无法绕开的三个点：

- 行
- 列
- 表结构描述

比如，在MySQL中的一张表：

- 由许多行组成
- 数据也被分成多个列
- 表也有表结构信息（列、列名、列类型、列约束等）

基于这个前提，DataFrame的组成如下：

在结构层面：

- StructType对象描述整个DataFrame的表结构
- StructField对象描述一个列的信息

在数据层面

- Row对象记录一行数据
- Column对象记录一列数据并包含列的信息



### 3.1 DataFrame的组成



如图，在表结构层面，DataFrame的表结构由：

StructType描述，如下图

```
struct_type = StructType().\n    add("id", IntegerType(), False).\\ ← StructType\n    add("name", StringType(), True).\\ ← StructField\n    add("age", IntegerType(), False) \\ ← StructField
```

一个StructField记录：列名、列类型、列是否运行为空

多个StructField组成一个StructType对象。

一个StructType对象可以描述一个DataFrame：有几个列、每个列的名字和类型  
、每个列是否为空

同时，一行数据描述为Row对象，如Row(1, 张三, 11)

一列数据描述为Column对象，Column对象包含一列数据和列的信息

Row、Column、StructType、StructField的编程我们在后面编码阶段会接触



## 3.2 DataFrame的代码构建 - 基于RDD方式1

DataFrame对象可以从RDD转换而来，都是分布式数据集  
其实就是转换一下内部存储的结构，转换为二维表结构

将RDD转换为DataFrame方式1：

调用spark

```
# 首先构建一个RDD rdd[(name, age), ()]
rdd = sc.textFile("../data/sql/people.txt").\
    map(lambda x: x.split(',')).\
    map(lambda x: [x[0], int(x[1])])          # 需要做类型转换, 因为类型从RDD中探测
# 构建DF方式1
df = spark.createDataFrame(rdd, schema = ['name', 'age'])
```

通过SparkSession对象的createDataFrame方法来将RDD转换为DataFrame

这里只传入列名称，类型从RDD中进行推断，是否允许为空默认为允许（True）

```
# coding:utf8
# 演示DataFrame创建的三种方式
from pyspark.sql import SparkSession
if __name__ == '__main__':
    spark = SparkSession.builder.\
        appName("create df").\
        master("local[*]").\
        getOrCreate()

    sc = spark.sparkContext
    # 首先构建一个RDD rdd[(name, age), ()]
    rdd = sc.textFile("../data/sql/people.txt").\
        map(lambda x: x.split(',')).\
        map(lambda x: [x[0], int(x[1])])          # 需要做类型转换, 因为类型从RDD中探测

    # 构建DF方式1
    df = spark.createDataFrame(rdd, schema = ['name', 'age'])
    # 打印表结构
    df.printSchema()
    # 打印20行数据
    df.show()

    df.createTempView("ttt")
    spark.sql("select * from ttt where age < 30").show()
```



## 3.2 DataFrame的代码构建 - 基于RDD方式2

将RDD转换为DataFrame方式2：

通过StructType对象来定义DataFrame的“表结构” 转换  
RDD

```
# 创建DF，首先创建RDD 将RDD转DF
rdd = sc.textFile("../data/sql/stu_score.txt").\
    map(lambda x:x.split(',')).\
    map(lambda x:(int(x[0]), x[1], int(x[2])))

# StructType 类
# 这个类 可以定义整个DataFrame中的Schema
schema = StructType().\
    add("id", IntegerType(), nullable=False).\
    add("name", StringType(), nullable=True).\
    add("score", IntegerType(), nullable=False)

# 一个add方法 定义一个列的信息,如果有3个列,就写三个add,每一个add代表一个StructField
# add方法: 参数1: 列名称, 参数2: 列类型, 参数3: 是否允许为空
df = spark.createDataFrame(rdd, schema)
```

```
# coding:utf8
# 需求: 基于StructType的方式构建DataFrame 同样是RDD转DF
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType, StringType, IntegerType
if __name__ == '__main__':
    spark = SparkSession.builder.\
        appName("create_df").\
        config("spark.sql.shuffle.partitions", "4").\
        getOrCreate()
    # SparkSession对象也可以获取 SparkContext
    sc = spark.sparkContext
    # 创建DF，首先创建RDD 将RDD转DF
    rdd = sc.textFile("../data/sql/stu_score.txt").\
        map(lambda x:x.split(',')).\
        map(lambda x:(int(x[0]), x[1], int(x[2])))

    # StructType 类
    # 这个类 可以定义整个DataFrame中的Schema
    schema = StructType().\
        add("id", IntegerType(), nullable=False).\
        add("name", StringType(), nullable=True).\
        add("score", IntegerType(), nullable=False)

    # 一个add方法 定义一个列的信息,如果有3个列,就写三个add
    # add方法: 参数1: 列名称, 参数2: 列类型, 参数3: 是否允许为空
    df = spark.createDataFrame(rdd, schema)
    df.printSchema()
    df.show()
```



## 3.2 DataFrame的代码构建 - 基于RDD方式3

将RDD转换为DataFrame方式3：

使用RDD的toDF方法转换RDD

```
# StructType 类
# 这个类 可以定义整个DataFrame中的Schema
schema = StructType().\
    add("id", IntegerType(), nullable=False).\
    add("name", StringType(), nullable=True).\
    add("score", IntegerType(), nullable=False)

# 一个add方法 定义一个列的信息, 如果有3个列, 就写三个add
# add方法: 参数1: 列名称, 参数2: 列类型, 参数3: 是否允许为空

# 方式1: 只传列名, 类型靠推断, 是否允许为空是true
df = rdd.toDF(['id', 'subject', 'score'])

df.printSchema()
df.show()

# 方式2: 传入完整的Schema描述对象StructType
df = rdd.toDF(schema)
df.printSchema()
df.show()
```

```
# coding:utf8
# 需求: 使用toDF方法将RDD转换为DF
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType, StringType, IntegerType
if __name__ == '__main__':
    spark = SparkSession.builder.\
        appName("create_df").\
        config("spark.sql.shuffle.partitions", "4").\
        getOrCreate()
    # SparkSession对象也可以获取 SparkContext
    sc = spark.sparkContext
    # 创建DF ,首先创建RDD 将RDD转DF
    rdd = sc.textFile("../data/sql/stu_score.txt").\
        map(lambda x:x.split(',')).\
        map(lambda x:(int(x[0]), x[1], int(x[2])))

    # StructType 类
    # 这个类 可以定义整个DataFrame中的Schema
    schema = StructType().\
        add("id", IntegerType(), nullable=False).\
        add("name", StringType(), nullable=True).\
        add("score", IntegerType(), nullable=False)

    # 一个add方法 定义一个列的信息, 如果有3个列, 就写三个add
    # add方法: 参数1: 列名称, 参数2: 列类型, 参数3: 是否允许为空
    # 方式1: 只传列名, 类型靠推断, 是否允许为空是true
    df = rdd.toDF(['id', 'subject', 'score'])

    df.printSchema()
    df.show()

    # 方式2: 传入完整的Schema描述对象StructType
    df = rdd.toDF(schema)
    df.printSchema()
    df.show()
```



## 3.2 DataFrame的代码构建 - 基于Pandas的DataFrame

将Pandas的DataFrame对象，转变为分布式的SparkSQL DataFrame对象

```
# 构建Pandas的DF
pdf = pd.DataFrame({
    "id": [1, 2, 3],
    "name": ["张大仙", '王晓晓', '王大锤'],
    "age": [11, 11, 11]
})
# 将Pandas的DF对象转换成Spark的DF
df = spark.createDataFrame(pdf)
```

```
# coding:utf8
# 演示将Panda的DataFrame转换成Spark的DataFrame
from pyspark.sql import SparkSession
# 导入StructType对象
from pyspark.sql.types import StructType, StringType, IntegerType
import pandas as pd
if __name__ == '__main__':
    spark = SparkSession.builder.\n        appName("create df").\n        master("local[*]").\n        getOrCreate()
    sc = spark.sparkContext
    # 构建Pandas的DF
    pdf = pd.DataFrame({
        "id": [1, 2, 3],
        "name": ["张大仙", '王晓晓', '王大锤'],
        "age": [11, 11, 11]
    })
    # 将Pandas的DF对象转换成Spark的DF
    df = spark.createDataFrame(pdf)
    df.printSchema()
    df.show()
```



## 3.2 DataFrame的代码构建 - 读取外部数据

通过SparkSQL的**统一API**进行数据读取构建DataFrame

统一API示例代码：

```
sparksession.read.format("text|csv|json|parquet|orc|avro|jdbc|.....")
    .option("K", "V") # option可选
    .schema(StructType | String) # STRING的语法如.schema("name STRING", "age INT")
    .load("被读取文件的路径, 支持本地文件系统和HDFS")
```



## 3.2 DataFrame的代码构建 - 读取外部数据

读取text数据源

使用format( "text" )读取文本数据

读取到的DataFrame只会有一个列，列名默认称之为：value

示例代码：

```
schema = StructType().add("data", StringType(), nullable=True)
df = spark.read.format("text")\
    .schema(schema)\n    .load("../data/sql/people.txt")
```



## 3.2 DataFrame的代码构建 - 读取外部数据

读取json数据源

使用format( "json" )读取json数据

示例代码：

```
df = spark.read.format("json")\
    .load("../data/sql/people.json")
# JSON 类型一般不用写.schema, json自带, json带有列名 和列类型(字符串和数字)
df.printSchema()
df.show()
```



## 3.2 DataFrame的代码构建 - 读取外部数据

读取csv数据源

使用format( "csv" )读取csv数据

示例代码：

```
df = spark.read.format("csv")\
    .option("sep", ";")\          # 列分隔符
    .option("header", False)\   # 是否有CSV标头
    .option("encoding", "utf-8")\      # 编码
    .schema("name STRING, age INT, job STRING")\      # 指定列名和类型
    .load("../data/sql/people.csv")\        # 路径
df.printSchema()
df.show()
```



## 3.2 DataFrame的代码构建 - 读取外部数据

读取parquet数据源

使用format(“parquet”)读取parquet数据

示例代码：

```
# parquet 自带schema, 直接load啥也不需要了
df = spark.read.format("parquet").\
    load("./data/sql/users.parquet")
df.printSchema()
df.show()
```

parquet: 是Spark中常用的一种列式存储文件格式

和Hive中的ORC差不多, 他俩都是列存储格式

parquet对比普通的文本文件的区别:

- parquet 内置schema (列名\ 列类型\ 是否为空)
- 存储是以列作为存储格式
- 存储是序列化存储在文件中的(有压缩属性体积小)

Parquet文件不能直接打开查看，如果想要查看内容可以在PyCharm中安装如下插件来查看：



A Tool Window for viewing Avro and Parquet files and their schemas

> Change Notes



### 3.3 DataFrame的入门操作

DataFrame支持两种风格进行编程，分别是：

- DSL风格
- SQL风格

#### DSL语法风格

DSL称之为：领域特定语言。

其实是指DataFrame的特有API

DSL风格意思就是以调用API的方式来处理Data

比如：df.where().limit()

#### SQL语法风格

SQL风格就是使用SQL语句处理DataFrame的数据

比如：spark.sql( "SELECT \* FROM xxx")



## 3.3 DataFrame的入门操作

### DSL - show 方法

功能：展示DataFrame中的数据，默认展示20条

语法：

```
df.show(参数1, 参数2)
- 参数1：默认是20，控制展示多少条
- 参数2：是否截断列，默认只输出20个字符的长度，过长不显示，要显示的话，请填入 truncate = True
```

如图，某个df.show后的展示结果

id	name	score
1	语文	99
2	语文	99
3	语文	99
4	语文	99
5	语文	99



### 3.3 DataFrame的入门操作

#### DSL - printSchema方法

功能：打印输出df的schema信息

语法：

```
df.printSchema()
```

```
root
|-- name: string (nullable = true)
|-- age: integer (nullable = true)
|-- job: string (nullable = true)
```



### 3.3 DataFrame的入门操作

#### DSL - select

功能：选择DataFrame中的指定列（通过传入参数进行指定）

语法：

```
self: DataFrame, *cols: Union[Column, str]
self: DataFrame, __cols: Union[List[Column], List[str]]  
  
8     df.select()
```

可传递：

- 可变参数的cols对象，cols对象可以是Column对象来指定列或者字符串列名来指定列
- List[Column]对象或者List[str]对象，用来选择多个列

```
# column 对象的获取
id_column = df['id']
subject_column = df['subject']

# select
# 支持字符串形式传入
df.select(["id", "subject"]).show()
df.select("id", "subject").show()
# 也支持column对象的方式传入
df.select(df['id'], df['subject']).show()
```



### 3.3 DataFrame的入门操作

#### DSL - filter和where

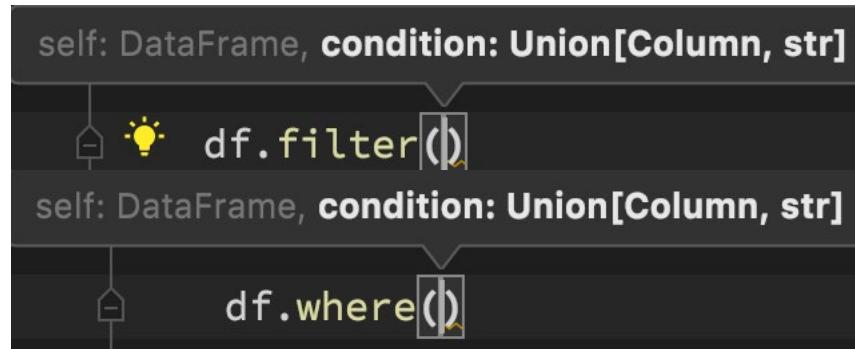
功能：过滤DataFrame内的数据，返回一个过滤后的DataFrame

语法：

df.filter()

df.where()

where和filter功能上是等价的



```
1 # filter
2 # 传字符串的形式
3 df.filter("score < 99").show()
4 # 传column的形式
5 df.filter(df['score'] < 99).show()
6 # where 和 filter等价
7 df.where("score < 99").show()
8 df.where(df['score'] < 99).show()
```



### 3.3 DataFrame的入门操作

#### DSL - groupBy 分组

功能：按照指定的列进行数据的分组， 返回值是GroupedData对象

语法：

df.groupBy()

```
self: DataFrame, *cols: Union[Column, str]
self: DataFrame, __cols: Union[List[Column], List[str]]  
  
💡 df.groupBy()
```

传入参数和select一样，支持多种形式，不管怎么传意思就是告诉spark按照哪个列分组

```
# groupBy
df.groupBy("subject").count().show()
df.groupBy(df['subject']).count().show()
```



## 3.3 DataFrame的入门操作

### GroupedData对象

GroupedData对象是一个特殊的DataFrame数据集

其类全名：<class 'pyspark.sql.group.GroupedData'>

这个对象是经过groupBy后得到的返回值，内部记录了以分组形式存储的数据

GroupedData对象其实也有很多API，比如前面的count方法就是这个对象的内置方法

除此之外，像：min、max、avg、sum、等等许多方法都存在

后续会再次使用它



### 3.3 DataFrame的入门操作

#### SQL风格语法 - 注册DataFrame成为表

DataFrame的一个强大之处就是我们可以将它看作是一个关系型数据表，然后可以通过在程序中使用spark.sql() 来执行SQL语句查询，结果返回一个DataFrame。

如果想使用SQL风格的语法，需要将DataFrame注册成表,采用如下的方式：

```
df.createTempView("score") # 注册一个临时视图(表)
df.createOrReplaceTempView("score") # 注册一个临时表，如果存在进行替换.
df.createGlobalTempView("score")    # 注册一个全局表
```

全局表：跨SparkSession对象使用，在一个程序内的多个SparkSession中均可调用，查询前带上前缀：

**global\_temp.**

临时表：只在当前SparkSession中可用



### 3.3 DataFrame的入门操作

#### SQL风格语法 - 使用SQL查询

注册好表后，可以通过：

`sparksession.sql(sql语句)` 来执行sql查询

返回值是一个新的df

示例：

```
1 # 注册好表后 就可以写sql查询
2 df2 = spark.sql("'''SELECT * FROM score WHERE score < 99'''")
3 df2.show()
```



### 3.3 DataFrame的入门操作

#### pyspark.sql.functions 包

PySpark提供了一个包: pyspark.sql.functions

这个包里面提供了一系列的计算函数供SparkSQL使用

如何用呢？

导包

`from pyspark.sql import functions as F`

然后就可以用F对象调用函数计算了。

这些功能函数, **返回值多数都是Column对象**.

详细的函数在后续开发中学习

#### 示例

split功能函数

功能: 切分字符串

语法:

`F.split(被切分的列, 切分字符串)`

返回值: Column对象

explode功能函数

功能: 数组转列

语法:

`F.explode(被转换的列)`

返回值: Column对象



### 3.4 词频统计案例练习

我们来完成一个单词计数需求，使用DSL和SQL两种风格来实现。

在实现的过程中，会出现新的API，边写边学习新API，代码中红色部分即新API

```
# coding:utf8
# 演示sparksql wordcount

from pyspark.sql import SparkSession
# 导入StructType对象
from pyspark.sql.types import StructType, StringType, IntegerType
import pandas as pd
from pyspark.sql import functions as F

if __name__ == '__main__':
    spark = SparkSession.builder.\
        appName("create df").\
        master("local[*]").\
        getOrCreate()
    sc = spark.sparkContext

    # TODO 1: SQL风格处理, 以RDD为基础做数据加载
    rdd = sc.textFile("hdfs://node1:8020/input/words.txt").\
        flatMap(lambda x: x.split(" ")).\
        map(lambda x: [x])
    # 转换RDD到df
    df = rdd.toDF(["word"])
```

```
# 注册df为表
df.createTempView("words")
# 使用sql语句处理df注册的表
spark.sql("""SELECT word, COUNT(*) AS cnt FROM words GROUP BY word ORDER BY cnt DESC""").show()

# TODO 2: DSL风格处理, 纯sparksql api做数据加载
# df当前只有一个列 叫做value
df = spark.read.format("text").load("hdfs://node1:8020/input/words.txt")
# df.select(F.explode(F.split(df['value'], " "))).show()
# 通过withColumn方法 对一个列进行操作
# 方法功能: 对老列执行操作, 返回一个全新列, 如果列名一样就替换, 不一样就拓展一个列
df2 = df.withColumn("value", F.explode(F.split(df['value'], " ")))
df2.groupBy("value").\
    count().\
    withColumnRenamed("count", "cnt").\
    orderBy('cnt', ascending=False).\
    show()
```



## 3.5 电影评分数据分析案例

MovieLens数据集

MovieLens数据集包含多个用户对多部电影的评级数据，也包括电影元数据信息和用户属性信息。

- 下载地址

<http://files.grouplens.org/datasets/movielens/>

- 介绍

下面以ml-100k数据集为例进行介绍：

下载u.data文件

u.data — 由943个用户对1682个电影的10000条评分组成。每个用户至少评分20部电影。用户和电影从1号开始连续编号。数据是随机排序的。

用户 ID	电影 ID	评分	时间
196	242	3	881250949
186	302	3	891717742
244	51	2	880606923
136	340	1	886397596
298	474	4	884182806
115	265	2	881171488
253	465	5	891628467
305	451	3	886324817
6	86	3	883603013
62	257	2	879372434
286	1014	5	879781125
200	222	5	876042340
210	40	3	891035994
224	29	3	888104457
303	785	3	879485318
122	387	5	879270459
194	274	2	879539794
291	1042	4	874834944
234	1184	2	892079237
119	392	4	886176814
167	486	4	892738452

### 需求

1. 查询用户平均分
2. 查询电影平均分
3. 查询大于平均分的电影的数量
4. 查询高分电影中(>3)打分次数最多的用户，并求出此人打的平均分
5. 查询每个用户的平均打分，最低打分，最高打分
6. 查询被评分超过100次的电影，的平均分 排名 TOP10



## 3.5 电影评分数据分析案例

```
# coding:utf8
# 演示sparksql 电影评分数据
import time

from pyspark.sql import SparkSession
# 导入StructType对象
from pyspark.sql.types import StructType, StringType, IntegerType
import pandas as pd
from pyspark.sql import functions as F

if __name__ == '__main__':
    spark = SparkSession.builder.\
        appName("create df").\
        master("local[*]").\
        config("spark.sql.shuffle.partitions", "2").\
        getOrCreate()
    sc = spark.sparkContext
    ...
    # 需求
    1.查询用户平均分
    2.查询电影平均分
    3.查询大于平均分的电影的数量
    4.查询高分电影中(>3)打分次数最多的用户，并求出此人打的平均分
    5.查询每个用户的平均打分，最低打分，最高打分
    6.查询被评分超过100次的电影，的平均分 排名 TOP10
    ...
    # 1. 读取数据集
    schema = StructType().add("user_id", StringType(), nullable=True).\
        add("movie_id", IntegerType(), nullable=True).\
        add("rank", IntegerType(), nullable=True).\
        add("ts", StringType(), nullable=True)
    df = spark.read.format("csv").\
        option("sep", "\t").\
        option("header", False).\
        option("encoding", "utf-8").\
        schema(schema).\
        load("../data/sql/u.data")

    # 2. 注册成一个临时表, 方便sq处理
    df.createTempView("movie")
```

```
# 需求1: 用户平均分
df.groupBy("user_id").\
    avg("rank").\
    withColumnRenamed("avg(rank)", "avg_rank").\
    withColumn("avg_rank", F.round("avg_rank", 2)).\
    orderBy("avg_rank", ascending=False).\
    show()

# 需求2: 查询电影平均分
spark.sql("""
    SELECT movie_id, ROUND(AVG(rank), 2) AS avg_rank FROM movie GROUP BY movie_id
    ORDER BY avg_rank DESC
""").show()

# 需求3: 查询大于平均分的电影的数量
# print(df.select(F.avg(df['rank']).first()['avg(rank)']))
print("大于平均分的电影数量: ", df.where(df['rank'] >
df.select(F.avg(df['rank']).first()['avg(rank)']).count()))

# 需求4: 查询高分电影中(>3)打分次数最多的用户，并求出此人打的平均分
# 先找出这个人
user_id = df.where("rank > 3").\
    groupBy("user_id").\
    count().\
    withColumnRenamed("count", "cnt").\
    orderBy("cnt", ascending=False).\
    limit(1).\
    first()['user_id']

# 计算这个人的平均分
df.filter(df['user_id'] == user_id).\
    select(F.round(F.avg('rank'), 2)).show()
# select round(avg(rank), 2) from biao where userid=450;
```

```
# 需求5: 查询每个用户的平均打分, 最低打分, 最高打分
# select user_id, min(rank) as avg_rank, max(rank) as
max_rank, avg(rank) from table group by user_id;
df.groupBy("user_id").\
    agg(
        F.round(F.avg('rank'), 2).alias("avg_rank"),
        F.min('rank').alias("min_rank"),
        F.max('rank').alias("max_rank")
    ).show()

# 需求6: 查询被评分超过100次的电影, 的平均分 排名 TOP10
# select movie_id, count(movie_id) as cnt, avg(rank) as
avg_rank from table group by movie_id having cnt > 100
# order by avg_rank desc limit 10;
df.groupBy("movie_id").\
    agg(
        F.count("movie_id").alias("cnt"),
        F.round(F.avg("rank"), 2).alias("avg_rank")
    ).where("cnt > 100").\
    orderBy("avg_rank", ascending=False).\
    limit(10).show()

time.sleep(600)
```



### 3.6 SparkSQL Shuffle 分区数目

运行上述程序时，查看WEB UI监控页面发现，某个Stage中有200个Task任务，也就是说RDD有200分区Partition。

#### - Completed Stages (3)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
2	show at SparkTop10Movie.scala:85 +details	2020/04/19 22:57:14	0.2 s	1/1			2.2 KB	
1	show at SparkTop10Movie.scala:85 +details	2020/04/19 22:57:10	4 s	200/200		323.6 KB	2.2 KB	
0	show at SparkTop10Movie.scala:85 +details	2020/04/19 22:57:05	5 s	4/4	23.7 ME			323.6 KB

原因：在SparkSQL中当Job中产生Shuffle时，默认的分区数（`spark.sql.shuffle.partitions`）为200，在实际项目中要合理的设置。可以设置在：

1. 配置文件: conf/spark-defaults.conf: `spark.sql.shuffle.partitions 100`
2. 在客户端提交参数中: bin/spark-submit --conf "spark.sql.shuffle.partitions=100"
3. 在代码中可以设置: `spark = SparkSession.builder.\`

```
appName("create df").\
master("local[*]"). \
config("spark.sql.shuffle.partitions", "2").\
getOrCreate()
```



## 3.7 SparkSQL 数据清洗API

前面我们处理的数据实际上都是已经被处理好的规整数据，但是在大数据整个生产过程中，需要先对数据进行数据清洗，将杂乱无章的数据整理为符合后面处理要求的规整数据。

### 去重方法 dropDuplicates

功能：对DF的数据进行去重，如果重复数据有多条，取第一条

```
1 # 去重API dropDuplicates，无参数是对数据进行整体去重
2 df.dropDuplicates().show()
3 # API 同样可以针对字段进行去重，如下传入age字段，表示只要年龄一样 就认为你是重复数据
4 df.dropDuplicates(['age', 'job']).show()
```



## 3.7 SparkSQL 数据清洗API

### 删除有缺失值的行方法 dropna

功能: 如果数据中包含null, 通过dropna来进行判断, 符合条件就删除这一行数据

Python | 复制代码

```
1 # 如果有缺失, 进行数据删除
2 # 无参数 为 how=any执行, 只要有一个列是null 数据整行删除, 如果填入how='all' 表示全部列为空 才会删除, how参数默认是a
3 df.dropna().show()
4 # 指定阀值进行删除,thresh=3表示, 有效的列最少有3个, 这行数据才保留
5 # 设定thresh后, how参数无效了
6 df.dropna(thresh=3).show()
7 # 可以指定阀值 以及配合指定列进行工作
8 # thresh=2, subset=['name', 'age'] 表示 针对这2个列, 有效列最少为2个才保留数据
9 df.dropna(thresh=2, subset=['name', 'age']).show()
```



## 3.7 SparkSQL 数据清洗API

### 填充缺失值数据 `fillna`

功能: 根据参数的规则, 来进行 `null` 的替换

```
1 # 将所有的空, 按照你指定的值进行填充, 不理会列的 任何空都被填充
2 df.fillna("loss").show()
3 # 指定列进行填充
4 df.fillna("loss", subset=['job']).show()
5 # 给定字典 设定各个列的填充规则
6 df.fillna({"name": "未知姓名", "age": 1, "job": "worker"}).show()
```

Python | 复制代码



## 3.8 DataFrame数据写出

### SparkSQL 统一API写出DataFrame数据

统一API语法:

Python | 复制代码

```
1 df.write.mode().format().option(K, V).save(PATH)
2 # mode, 传入模式字符串可选: append 追加, overwrite 覆盖, ignore 忽略, error 重复就报异常(默认的)
3 # format, 传入格式字符串, 可选: text, csv, json, parquet, orc, avro, jdbc
4 # 注意text源只支持单列df写出
5 # option 设置属性, 如: .option("sep", ",")r
6 # save 写出的路径, 支持本地文件和HDFS
```



## 3.8 DataFrame数据写出

常见的源写出：

```
1 # Write text 写出, 只能写出一个单列数据
2 df.select(F.concat_ws("----", "user_id", "movie_id", "rank", "ts")).\
3     write.\
4     mode("overwrite").\
5     format("text").\
6     save("../data/output/sql/text")
7 # Write CSV 写出
8 df.write.mode("overwrite").\
9     format("csv").\
10    option("sep", ",").\
11    option("header", True).\
12    save("../data/output/sql/csv")
13 # Write Json 写出
14 df.write.mode("overwrite").\
15     format("json").\
16     save("../data/output/sql/json")
17 # Write Parquet 写出
18 df.write.mode("overwrite").\
19     format("parquet").\
20     save("../data/output/sql/parquet")
21 # 不给format, 默认以 parquet写出
22 df.write.mode("overwrite").save("../data/output/sql/default")
```

Python | 复制代码



## 3.9 DataFrame 通过JDBC读写数据库 (MySQL示例)

读取JDBC 是需要有驱动的，我们读取的是 `jdbc:mysql://` 这个协议，也就是读取的是mysql的数据

既然如此，就需要有mysql的驱动jar包给spark程序用。

如果不给驱动jar包，会提示：`No suitable Driver`

驱动包在资料中：

 mysql-connector-java-5.1.41-bin.jar	给mysql5版本用	2020-07-05 星期日 2...	Executab
 mysql-connector-java-8.0.13.jar	给mysql8版本用	2020-07-05 星期日 2...	Executab

对于windows系统(使用本地解释器)(以Anaconda环境演示)

将jar包放在：`Anaconda3的安装路径下\envs\虚拟环境\Lib\site-packages\pyspark\jars`

对于Linux系统(使用远程解释器执行)(以Anaconda环境演示)

将jar包放在：`Anaconda3的安装路径下/envs/虚拟环境/lib/python3.8/site-packages/pyspark/jars`



## 3.9 DataFrame 通过JDBC读写数据库（MySQL示例）

写出：

```
1 # 写DF通过JDBC到数据库中
2 df.write.mode("overwrite").\
3     format("jdbc").\
4     option("url", "jdbc:mysql://node1:3306/test?useSSL=false&useUnicode=true").\
5     option("dbtable", "u_data").\
6     option("user", "root").\
7     option("password", "123456").\
8     save()
```

Python | 复制代码

注意：

- jdbc连接字符串中，建议使用 useSSL=false 确保连接可以正常连接( 不使用SSL安全协议进行连接 )
- jdbc连接字符串中，建议使用 useUnicode=true 来确保传输中不出现乱码
- save() 不要填参数，没有路径，是写出数据库
- dbtable属性：指定写出的表名



### 3.9 DataFrame 通过JDBC读写数据库 (MySQL示例)

读取:

Python | 复制代码

```
1 df = spark.read.format("jdbc"). \
2     option("url", "jdbc:mysql://node1:3306/test?useSSL=false&useUnicode=true"). \
3     option("dbtable", "u_data"). \
4     option("user", "root"). \
5     option("password", "123456"). \
6     load()
```

注意:

- 读出来是自带schema, 不需要设置schema, 因为数据库就有schema
- load()不需要加参数, 没有路径, 从数据库中读取的啊
- dbtable:是指定读取的表名



## 总结



1. DataFrame 在结构层面上由StructField组成列描述，由 StructType构造表描述。在数据层面上，Column对象记录列数据，Row对象记录行数据
2. DataFrame可以从RDD转换、Pandas DF转换、读取文件、读取 JDBC等方法构建
3. spark.read.format()和df.write.format() 是DataFrame读取和写出的统一化标准API
4. SparkSQL默认在Shuffle阶段200个分区，可以修改参数获得最好性能
5. dropDuplicates可以去重、dropna可以删除缺失值、fillna可以填充缺失值
6. SparkSQL支持JDBC读写，可用标准API对数据库进行读写操作



## SparkSQL函数定义

- 4.1 SparkSQL 定义UDF函数
- 4.2 SparkSQL 使用窗口函数



## 4.1 SparkSQL 定义UDF函数

无论Hive还是SparkSQL分析处理数据时，往往需要使用函数，SparkSQL模块本身自带很多实现公共功能的函数，在`pyspark.sql.functions`中。SparkSQL与Hive一样支持定义函数：UDF和UDAF，尤其是UDF函数在实际项目中使用最为广泛。

回顾Hive中自定义函数有三种类型：

- **第一种：UDF (User-Defined-Function) 函数**
  - 一对一的关系，输入一个值经过函数以后输出一个值；
  - 在Hive中继承UDF类，方法名称为evaluate，返回值不能为void，其实就是实现一个方法；
- **第二种：UDAF (User-Defined Aggregation Function) 聚合函数**
  - 多对一的关系，输入多个值输出一个值，通常与groupBy联合使用；
- **第三种：UDTF (User-Defined Table-Generating Functions) 函数**
  - 一对多的关系，输入一个值输出多个值（一行变为多行）；
  - 用户自定义生成函数，有点像flatMap；



## 4.1 SparkSQL 定义UDF函数

目前来说Spark 框架各个版本及各种语言对自定义函数的支持：

Apache Spark Version	Spark SQL UDF (Python, Java, Scala)	Spark SQL UDAF (Java, Scala)	Spark SQL UDF (R)	Hive UDF, UDAF, UDTF
1.1-1.4	✓			✓
1.5	✓	experimental		✓
1.6	✓	✓		✓
2.0	✓	✓	✓	✓

在SparkSQL中，目前仅仅支持UDF函数和UDAF函数，目前Python仅支持UDF



## 4.1 SparkSQL 定义UDF函数

定义方式有2种：

1. sparksession.udf.register()

注册的UDF可以用于DSL和SQL

返回值用于DSL风格，传参内给的名字用于SQL风格

2. pyspark.sql.functions.udf

仅能用于DSL风格

方式1语法：

udf对象 = sparksession.udf.register(参数1, 参数2, 参数3)

参数1：UDF名称，可用于SQL风格

参数2：被注册成UDF的方法名

参数3：声明UDF的返回值类型

udf对象：返回值对象，是一个UDF对象，可用于DSL风格

方式2语法：

udf对象 = F.udf(参数1, 参数2)

参数1：被注册成UDF的方法名

参数2：声明UDF的返回值类型

udf对象：返回值对象，是一个UDF对象，可用于DSL风格

其中F是：

from pyspark.sql import functions as F

其中，被注册成UDF的方法名是指具体的计算方法，如：

def add(x, y): x + y

add就是将要被注册成UDF的方法名



## 4.1 SparkSQL 定义UDF函数

演示如下，构建一个Integer返回值类型的UDF

```
1 # TODO 方式1注册
2 # 注册UDF， 功能： 将数字都乘以10
3 def num_ride_10(num):
4     return num * 10
5 # 返回值用于DSL风格    内部注册的名称用于SQL(字符串表达式)风格
6 # 参数1： UDF名称(可用于SQL风格)， 参数2： UDF的本体方法(处理逻辑)， 参数3： 声明返回值类型
7 # 返回值可用于DSL
8 udf2 = spark.udf.register("udf1", num_ride_10, IntegerType())
9 df.select(udf2(df['num'])).show()
10 # select udf1(num)
11 df.selectExpr("udf1(num)").show()
12 # TODO 方式2注册， 仅能用DSL风格
13 # 参数1： UDF的本体方法(处理逻辑)， 参数2： 声明返回值类型
14 udf3 = F.udf(num_ride_10, IntegerType())
15 df.select(udf3(df['num'])).show()
```



## 4.1 SparkSQL 定义UDF函数

注册一个Float返回值类型：

Python | 复制代码

```
1 rdd = sc.parallelize([1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7]).map(lambda x: [x])
2 df = rdd.toDF(["num"])
3 # TODO 方式1注册
4 # 注册UDF，功能：将数字都乘以10
5 def num_ride_10(num):
6     return num * 10
7 # 返回值用于DSL风格 内部注册的名称用于SQL(字符串表达式)风格
8 udf2 = spark.udf.register("udf1", num_ride_10, FloatType())
9 df.select(udf2(df['num'])).show()
10 # select udf1(num)
11 df.selectExpr("udf1(num)").show()
12 # TODO 方式2注册，仅能用DSL风格
13 udf3 = F.udf(num_ride_10, FloatType())
14 df.select(udf3(df['num'])).show()
```



## 4.1 SparkSQL 定义UDF函数

注册一个ArrayType(数字\list)类型的返回值UDF

Python | 复制代码

```
1 rdd = sc.parallelize([["hadoop spark flink"], ["hadoop flink java"]])
2 df = rdd.toDF(["line"])
3 # TODO 方式1注册
4 # 注册UDF，功能：将数字都乘以10
5 def split_line(line):
6     return line.split(" ")
7 # 返回值用于DSL风格 内部注册的名称用于SQL(字符串表达式)风格
8 udf2 = spark.udf.register("udf1", split_line, ArrayType(StringType()))
9 df.select(udf2(df['line'])).show()
10 # select udf1(num)
11 df.selectExpr("udf1(line)").show()
12 # TODO 方式2注册，仅能用DSL风格
13 udf3 = F.udf(split_line, ArrayType(StringType()))
14 df.select(udf3(df['line'])).show(truncate=False)
```

注意: 数组或者list类型, 可以使用spark的 ArrayType来描述即可.

注意: 声明ArrayType要类似这样: `ArrayType(StringType())`, 在ArrayType中传入数组内的数据类型



## 4.1 SparkSQL 定义UDF函数

注册一个字典类型的返回值的UDF

```
1 rdd = sc.parallelize([[1], [2], [3]])
2 df = rdd.toDF(["num"])
3 # TODO 方式1注册
4 # 注册UDF，功能：将数字都乘以10
5 def split_line(num):
6     return {"num": num, "letter_str": string.ascii_letters[num]}
7 struct_type = StructType().add("num", IntegerType(), nullable=True).\
8     add("letter_str", StringType(), nullable=True)
9 # 返回值用于DSL风格 内部注册的名称用于SQL(字符串表达式)风格
10 udf2 = spark.udf.register("udf1", split_line, struct_type)
11 df.select(udf2(df['num'])).show()
12 # select udf1(num)
13 df.selectExpr("udf1(num)").show()
14 # TODO 方式2注册，仅能用DSL风格
15 udf3 = F.udf(split_line, struct_type)
16 df.select(udf3(df['num'])).show(truncate=False)
```

Python

复制代码

注意: 字典类型返回值, 可以用StructType来进行描述

StructType是一个普通的Spark支持的结构化类型

只是可以用在:

- DF中用于描述Schema
- UDF中用于描述返回值是字典的数据



## 4.1 SparkSQL 定义UDF函数

### 注意

使用UDF两种方式的注册均可以.

唯一需要注意的就是：返回值类型一定要有合适的类型来声明

返回int 可以用IntegerType

返回值小数，可以用 FloatType 或者DoubleType

返回数组 list可用 ArrayType描述

返回字典 可用StructType描述

.....

这些Spark内置的数据类型均存储在:

`pyspark.sql.types` 包中.



## 4.2 SparkSQL 使用窗口函数

### 开窗函数

#### ●介绍

开窗函数的引入是为了既显示聚集前的数据，又显示聚集后的数据。即在每一行的最后一列添加聚合函数的结果。

开窗用于为行定义一个窗口(这里的窗口是指运算将要操作的行的集合)，它对一组值进行操作，不需要使用 GROUP BY 子句对数据进行分组，能够在同一行中同时返回基础行的列和聚合列。

#### ●聚合函数和开窗函数

聚合函数是将多行变成一行，count,avg....

开窗函数是将一行变成多行；

聚合函数如果要显示其他的列必须将列加入到group by中

开窗函数可以不使用group by，直接将所有信息显示出来

#### ●开窗函数分类

##### 1.聚合开窗函数

聚合函数(列) OVER(选项)，这里的选项可以是PARTITION BY 子句，但不可以是 ORDER BY 子句。

##### 2.排序开窗函数

排序函数(列) OVER(选项)，这里的选项可以是ORDER BY 子句，也可以是 OVER(PARTITION BY 子句 ORDER BY 子句)，但不可以是 PARTITION BY 子句。

##### 3.分区类型NTILE的窗口函数



## 4.2 SparkSQL 使用窗口函数

窗口函数的语法：

Python | 复制代码

```
1 # 聚合类型 SUM\MIN\MAX\AVG\COUNT
2 sum() OVER([PARTITION BY xxx][ORDER BY xxx [DESC]])
3
4 # 排序类型：ROW_NUMBER|RANK|DENSE_RANK
5 ROW_NUMBER() OVER([PARTITION BY xxx][ORDER BY xxx [DESC]])
6
7 # 分区类型：NTILE
8 NTILE(number) OVER([PARTITION BY xxx][ORDER BY xxx [DESC]])
```



## 4.2 SparkSQL 使用窗口函数

```
# coding:utf8
# 演示sparksql 窗口函数(开窗函数)
import string

from pyspark.sql import SparkSession
# 导入StructType对象
from pyspark.sql.types import ArrayType, StringType, StructType, IntegerType
import pandas as pd
from pyspark.sql import functions as F

if __name__ == '__main__':
    spark = SparkSession.builder.\
        appName("create df").\
        master("local[*]").\
        config("spark.sql.shuffle.partitions", "2").\
        getOrCreate()
    sc = spark.sparkContext

    rdd = sc.parallelize([
        ('张三', 'class_1', 99),
        ('王五', 'class_2', 35),
        ('王三', 'class_3', 57),
        ('王久', 'class_4', 12),
        ('王丽', 'class_5', 99),
        ('王娟', 'class_1', 90),
        ('王军', 'class_2', 91),
        ('王俊', 'class_3', 33),
        ('王君', 'class_4', 55),
        ('王琪', 'class_5', 66),
        ('郑颖', 'class_1', 11),
        ('郑晖', 'class_2', 33),
        ('张丽', 'class_3', 36),
        ('张强', 'class_4', 79),
        ('黄凯', 'class_5', 90),
        ('黄开', 'class_1', 90),
        ('黄恺', 'class_2', 90),
        ('王凯', 'class_3', 11),
        ('王凯杰', 'class_1', 11),
        ('王开杰', 'class_2', 3),
        ('王景亮', 'class_3', 99)
    ])
```

```
schema = StructType().add("name", StringType()).\
    add("class", StringType()).\
    add("score", IntegerType())
df = rdd.toDF(schema)

# 窗口函数只用于SQL风格, 所以注册表先
df.createTempView("stu")

# TODO 聚合窗口
spark.sql("""
    SELECT *, AVG(score) OVER() AS avg_score FROM stu
""").show()
# SELECT *, AVG(score) OVER() AS avg_score FROM stu 等同于
# SELECT * FROM stu
# SELECT AVG(score) FROM stu
# 两个SQL的结果集进行整合而来

spark.sql("""
    SELECT *, AVG(score) OVER(PARTITION BY class) AS avg_score FROM stu
""").show()
# SELECT *, AVG(score) OVER(PARTITION BY class) AS avg_score FROM stu 等同于
# SELECT * FROM stu
# SELECT AVG(score) FROM stu GROUP BY class
# 两个SQL的结果集进行整合而来

# TODO 排序窗口
spark.sql("""
    SELECT *, ROW_NUMBER() OVER(ORDER BY score DESC) AS row_number_rank,
    DENSE_RANK() OVER(PARTITION BY class ORDER BY score DESC) AS dense_rank,
    RANK() OVER(ORDER BY score) AS rank
    FROM stu
""").show()
# TODO NTILE
spark.sql("""
    SELECT *, NTILE(6) OVER(ORDER BY score DESC) FROM stu
""").show()
```

## 总结

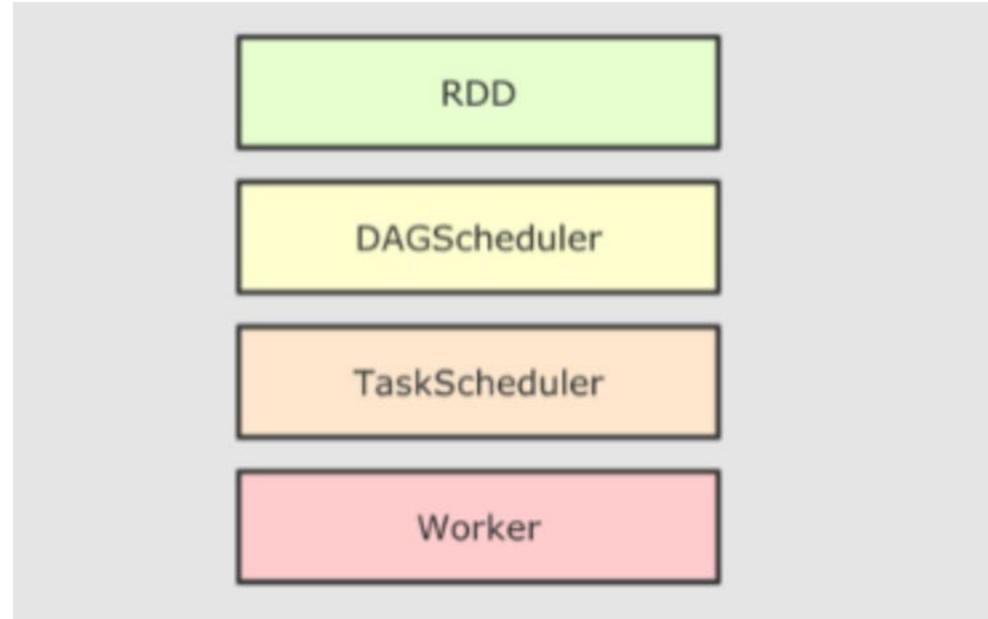
1. SparkSQL支持UDF和UDAF定义，但在Python中，暂时只能定义UDF
2. UDF定义支持2种方式，1:使用SparkSession对象构建。2: 使用functions包中提供的UDF API构建。要注意，方式1可用DSL和SQL风格，方式2仅可用于DSL风格
3. SparkSQL支持窗口函数使用，常用SQL中的窗口函数均支持，如聚合窗口\排序窗口\NTILE分组窗口等



## SparkSQL的运行流程

- 5.1 SparkRDD的执行流程回顾
- 5.2 SparkSQL的自动优化
- 5.3 Catalyst优化器
- 5.4 SparkSQL的执行流程

## 5.1 RDD的执行流程回顾



代码 -> DAG调度器逻辑任务 -> Task调度器任务分配和管理监控 -> Worker干活



## 5.2 SparkSQL的自动优化

RDD的运行会完全按照开发者的代码执行，如果开发者水平有限，RDD的执行效率也会受到影响。

而SparkSQL会对写完的代码，执行“自动优化”，以提升代码运行效率，避免开发者水平影响到代码执行效率。



为什么SparkSQL可以自动优化  
而RDD不可以？

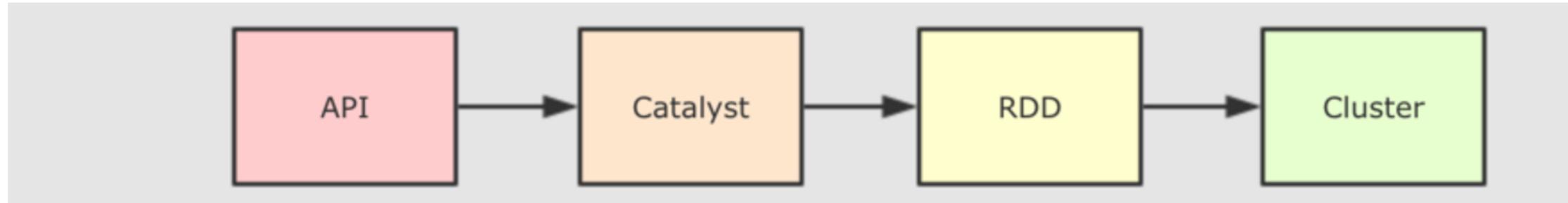
**RDD：内含数据类型不限格式和结构**  
**DataFrame：100% 是二维表结构，可以被针对**

**SparkSQL的自动优化，依赖于：Catalyst优化器**



## 5.3 Catalyst优化器

为了解决过多依赖 Hive 的问题, SparkSQL 使用了一个新的 SQL 优化器替代 Hive 中的优化器, 这个优化器就是 Catalyst, 整个 SparkSQL 的架构大致如下:

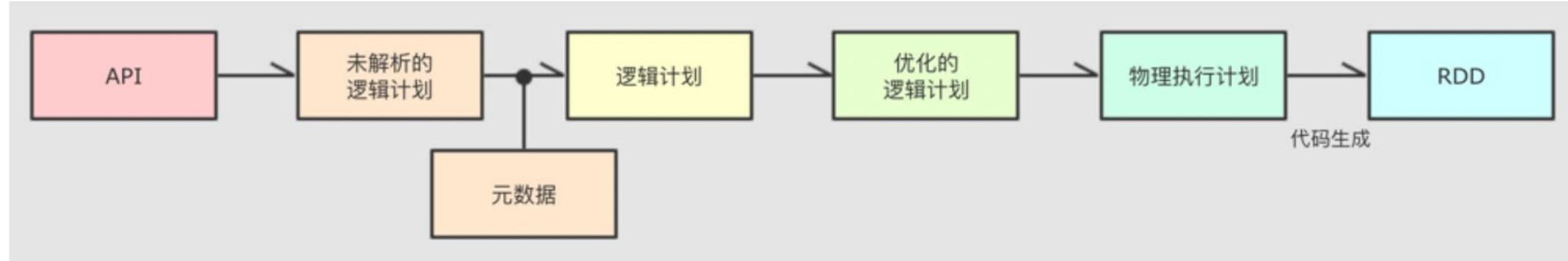


- 1.API 层简单的说就是 Spark 会通过一些 API 接受 SQL 语句
- 2.收到 SQL 语句以后, 将其交给 Catalyst, Catalyst 负责解析 SQL, 生成执行计划等
- 3.Catalyst 的输出应该是 RDD 的执行计划
- 4.最终交由集群运行

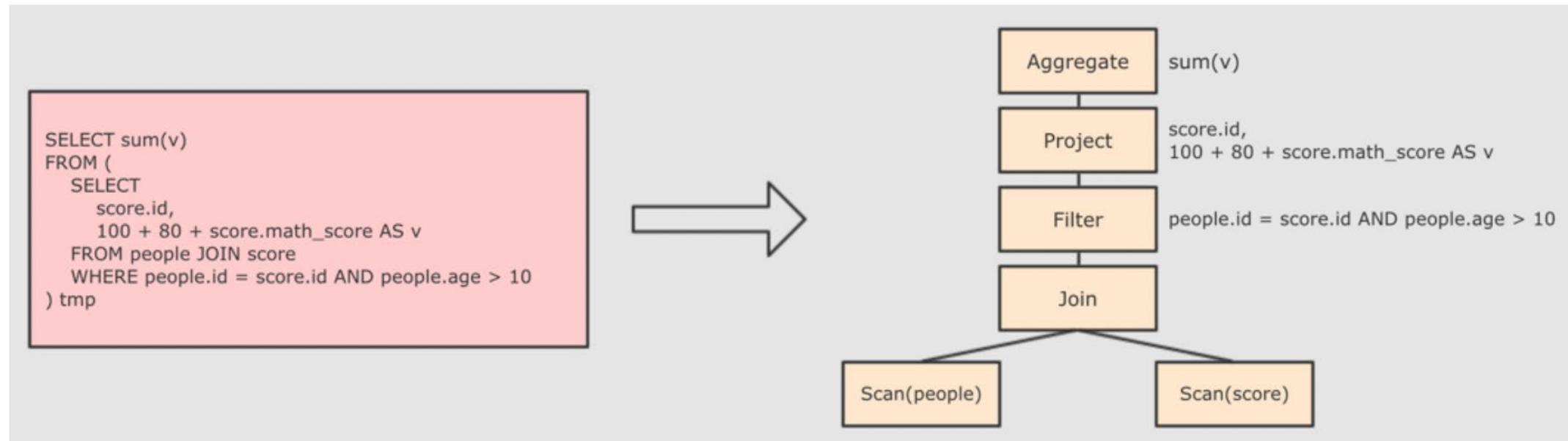


## 5.3 Catalyst优化器

具体流程：



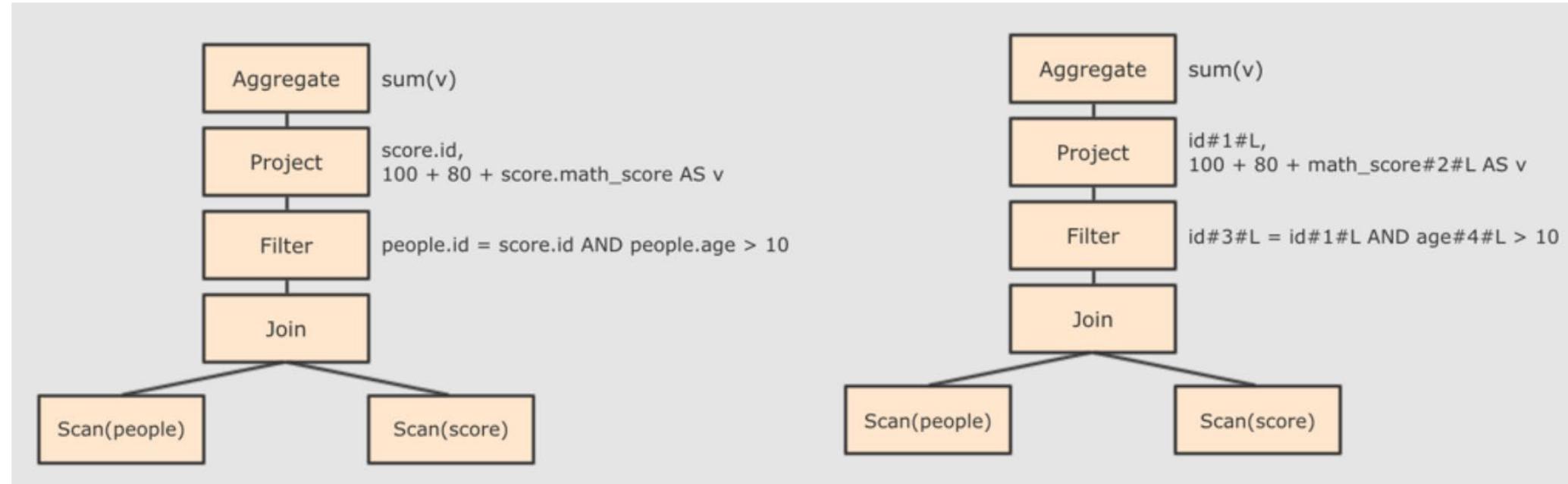
Step 1：解析 SQL, 并且生成 AST (抽象语法树)





## 5.3 Catalyst优化器

Step 2：在 AST 中加入元数据信息，做这一步主要是为了一些优化，例如 `col = col` 这样的条件，下图是一个简略图，便于理解

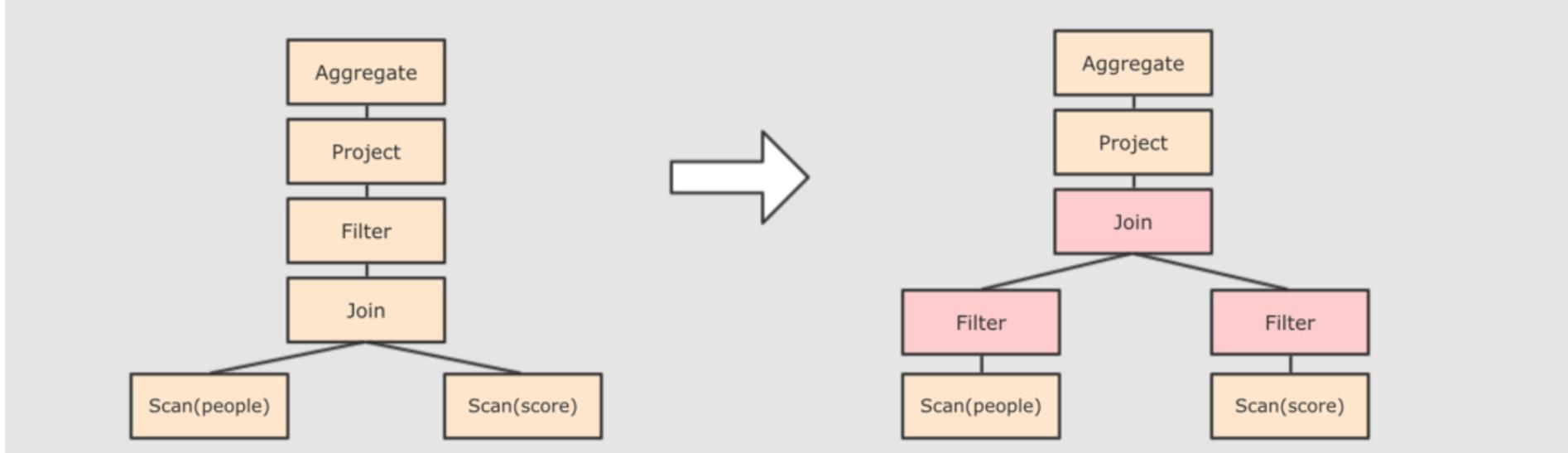


- `score.id → id#1#L` 为 `score.id` 生成 id 为 1, 类型是 Long
- `score.math_score → math_score#2#L` 为 `score.math_score` 生成 id 为 2, 类型为 Long
- `people.id → id#3#L` 为 `people.id` 生成 id 为 3, 类型为 Long
- `people.age → age#4#L` 为 `people.age` 生成 id 为 4, 类型为 Long



## 5.3 Catalyst优化器

Step 3：对已经加入元数据的 AST，输入优化器，进行优化，从两种常见的优化开始，简单介绍：



- 断言下推 Predicate Pushdown, 将 Filter 这种可以减小数据集的操作下推, 放在 Scan 的位置, 这样可以减少操作时候的数据量。

```
SELECT sum(v)
FROM (
  SELECT
    score.id,
    100 + 80 + score.math_score AS v
  FROM people JOIN score
  WHERE people.id = score.id AND people.age > 10
) tmp
```

如这个代码, 正常流程是先JOIN 然后做WHERE

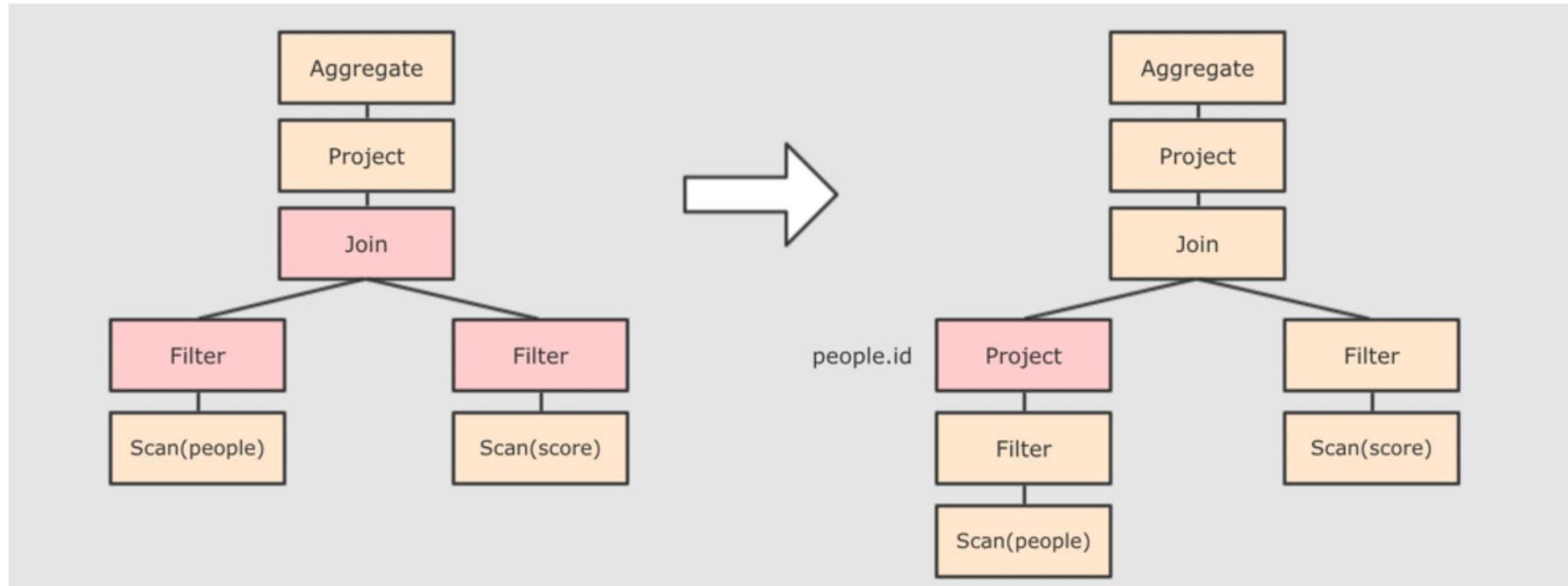
断言下推后, 会先过滤age, 然后在JOIN, 减少JOIN的数据量提高性能.

- 列值裁剪 Column Pruning, 在断言下推后执行裁剪, 由于people 表之上的操作只用到了 id 列, 所以可以把其它列裁剪掉, 这样可以减少处理的数据量, 从而优化处理速度



## 5.3 Catalyst优化器

如下图, 在scan前又加入了Filter, 作为列裁剪用



- 还有其余很多优化点, 大概一二百种, 随着 SparkSQL 的发展, 还会越来越多, 感兴趣的同学可以继续通过源码了解, 源码在 `org.apache.spark.sql.catalyst.optimizer.Optimizer`



## 5.3 Catalyst优化器

Step 4：上面的过程生成的 AST 其实最终还没办法直接运行，这个 AST 叫做 逻辑计划，结束后，需要生成 物理计划，从而生成 RDD 来运行

在生成‘物理计划’的时候，会经过‘成本模型’对整棵树再次执行优化，选择一个更好的计划

在生成‘物理计划’以后，因为考虑到性能，所以会使用代码生成，在机器中运行

可以使用 `queryExecution` 方法查看逻辑执行计划，使用 `explain` 方法查看物理执行计划

```
>>> spark.sql("select name,age from people where age>19").explain(True)
== Parsed Logical Plan ==
'Project ['name', 'age']
+- 'Filter ('age > 19)
  +- 'UnresolvedRelation [people], [], false

== Analyzed Logical Plan ==
name: string, age: bigint
Project [name#8, age#7L]
+- Filter (age#7L > cast(19 as bigint))
  +- SubqueryAlias people
    +- Project [name#8, age#7L]
      +- Relation[age#7L,name#8] json

== Optimized Logical Plan ==
Project [name#8, age#7L]
+- Filter (isnotnull(age#7L) AND (age#7L > 19))
  +- Relation[age#7L,name#8] json

== Physical Plan ==
*(1) Project [name#8, age#7L]
+- *(1) Filter (isnotnull(age#7L) AND (age#7L > 19))
  +- FileScan json [age#7L,name#8] Batched: false, DataFilters: [isnotnull(age#7L), (age#7L > 19)], Format: JSON, Location: InMemoryFileIndex@dfs://node1:9820/pydata/people.json, PartitionFilters: [], PushedFilters: [IsNotNull(age)], ReadSchema: struct<name:string,age:int>
```

```
>>> spark.sql("select name,age from people where age>19").explain()
== Physical Plan ==
*(1) Project [name#8, age#7L]
+- *(1) Filter (isnotnull(age#7L) AND (age#7L > 19))
  +- FileScan json [age#7L,name#8] Batched: false, DataFilters: [isnotnull(age#7L), (age#7L > 19)], Format: JSON, Location: InMemoryFileIndex@dfs://node1:9820/pydata/people.json, PartitionFilters: [], PushedFilters: [IsNotNull(age)], ReadSchema: struct<name:string,age:int>
```



## 5.3 Catalyst优化器 总结

catalyst的各种优化细节非常多, 大方面的优化点有2个:

- 谓词下推(Predicate Pushdown) \ 断言下推: 将逻辑判断 提前到前面, 以减少shuffle阶段的数据量.
- 列值裁剪(Column Pruning): 将加载的列进行裁剪, 尽量减少被处理数据的宽度

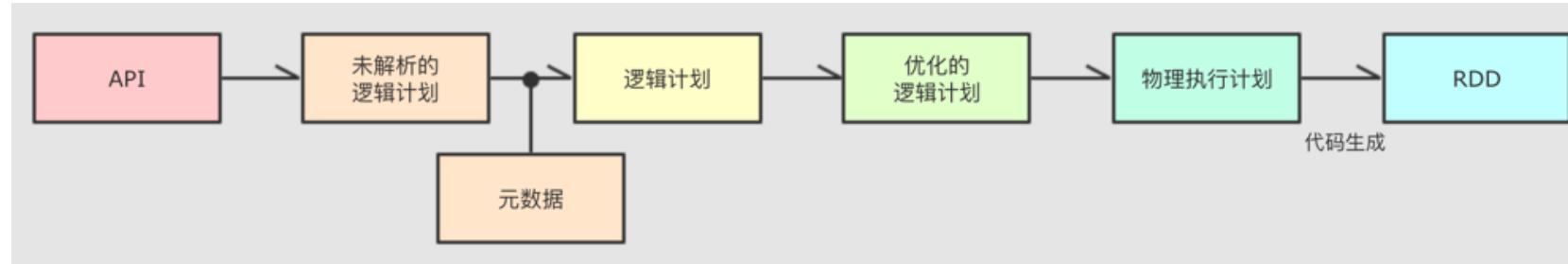
大白话:

- 行过滤, 提前执行where
- 列过滤, 提前规划select的字段数量

思考: 列值裁剪, 有一种非常合适的存储系统: parquet



## 5.4 SparkSQL的执行流程



1. 提交SparkSQL代码
2. catalyst优化
  - a. 生成原始AST语法树
  - b. 标记AST元数据
  - c. 进行断言下推和列值裁剪 以及其它方面的优化作用在AST上
  - d. 将最终AST得到, 生成执行计划
  - e. 将执行计划翻译为RDD代码
3. Driver执行环境入口构建 (SparkSession)
4. DAG 调度器规划逻辑任务
5. TASK 调度区分配逻辑任务到具体Executor上工作并监控管理任务
6. Worker干活.

# 总结

1. DataFrame因为存储的是二维表数据结构，可以被针对，所以可以自动优化执行流程。
2. 自动优化依赖Catalyst优化器
3. 自动优化2个大的优化项是：1. 断言（谓词）下推（行过滤） 2. 列值裁剪（列过滤）
4. DataFrame代码在被优化后，最终还是被转换成RDD去执行



## Spark On Hive

- 6.1 原理
- 6.2 配置
- 6.3 在代码中集成



## 6.1 原理

### 回顾Hive的组件

对于Hive来说，就2东西：

1. SQL优化翻译器（执行引擎），翻译SQL到MapReduce并提交到YARN执行
2. MetaStore 元数据管理中心



## 6.1 原理

### Spark On Hive

对于Spark来说，自身是一个 执行引擎

但是 Spark自己没有元数据管理功能，当我们执行：

`SELECT * FROM person WHERE age > 10` 的时候，Spark完全有能力将SQL变成RDD提交

但是问题是，Person的数据在哪？Person有哪些字段？字段啥类型？Spark完全不知道了

不知道这些东西，如何翻译RDD运行。

在SparkSQL代码中 可以写SQL 那是因为，表是来自DataFrame注册的。

DataFrame中有数据，有字段，有类型，足够Spark用来翻译RDD用。

如果以不写代码的角度来看，`SELECT * FROM person WHERE age > 10` spark无法翻译，因为没有 元数据



## 6.1 原理

解决方案

Spark提供执行引擎能力

Hive的MetaStore 提供元数据管理功能.

让Spark和Metastore连接起来, 那么:

Spark On Hive 就有了:

1. 引擎: spark
2. 元数据管理: metastore

总结:

Spark On Hive 就是把Hive的MetaStore服务拿过来 给Spark做元数据管理用而已.

市面上元数据管理的框架很多, 为什么非要用Hive内置的MetaStore

为了割Hive用户的韭菜.



## 6.2 配置

根据原理，就是Spark能够连接上Hive的MetaStore就可以了。

所以：

1. MetaStore需要存在并开机
2. Spark知道MetaStore在哪里( IP 端口号)



## 6.2 配置

步骤1:

在 `spark` 的 `conf` 目录中, 创建 `hive-site.xml`

内容如下:

```
1 <?xml version="1.0"?>
2 <?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
3 <configuration>
4   <!-- 告知Spark创建表存到哪里 -->
5   <property>
6     <name>hive.metastore.warehouse.dir</name>
7     <value>/user/hive/warehouse</value>
8   </property>
9   <property>
10    <name>hive.metastore.local</name>
11    <value>false</value>
12  </property>
13  <!-- 告知Spark Hive的MetaStore在哪 -->
14  <property>
15    <name>hive.metastore.uris</name>
16    <value>thrift://node1:9083</value>
17  </property>
18 </configuration>
```

XML

复制代码



## 6.2 配置

步骤2:

将mysql的驱动jar包放入spark的jars目录

因为要连接元数据，会有部分功能连接到mysql库，需要mysql驱动包

步骤3:

确保Hive 配置了MetaStore相关的服务

检查 `hive` 配置文件目录内的: `hive-site.xml`

确保有如下配置:

```
1 <configuration>
2   <property>
3     <name>hive.metastore.uris</name>
4     <value>thrift://node1:9083</value>
5   </property>
6 </configuration>
```

XML

复制代码



## 6.2 配置

步骤4:

启动hive的MetaStore服务:

```
1 nohup /export/server/hive/bin/hive --service metastore 2>&1 >> /var/log/metastore.log &
```

Shell

复制代码

nohup : 后台启动程序的命令,使用

- `nohup xxx命令 &` 将命令后台执行,日志输出到当前目录的nohup.out中
- `nohup xxx命令 2>&1 >> 某路径下的日志文件 &`, 将命令后台执行, 将日志输出到你指定的路径中

测试:

bin/pyspark: 在里面直接写spark.sql("sql语句").show() 即可

或者:

bin/spark-sql: 可以直接写sql语句



## 6.3 在代码中集成Spark On Hive

前提：确保MetaStore服务是启动好的

Python | 复制代码

```
1 spark = SparkSession.builder.\n2   appName("create df").\\n3     master("local[*]"). \\n4     config("spark.sql.shuffle.partitions", "4"). \\n5     config("spark.sql.warehouse.dir", "hdfs://node1:8020/user/hive/warehouse"). \\n6     config("hive.metastore.uris", "thrift://node1:9083"). \\n7     enableHiveSupport().\\n8     getOrCreate()\n9\n10 spark.sql("""SELECT * FROM test.student""").show()\n11 # 如上加入3条语句:\n12 # 告知Spark 默认创建表存到哪里\n13 config("spark.sql.warehouse.dir", "hdfs://node1:8020/user/hive/warehouse"). \\n14 # 告知Spark Hive的MetaStore在哪\n15 config("hive.metastore.uris", "thrift://node1:9083"). \\n16 # 告知Spark 开启对Hive的支持\n17 enableHiveSupport().\\
```



## 总结

Spark On Hive 就是因为Spark自身没有元数据管理功能，所以使用Hive的Metastore服务作为元数据管理服务。计算由Spark执行。



## 分布式SQL执行引擎

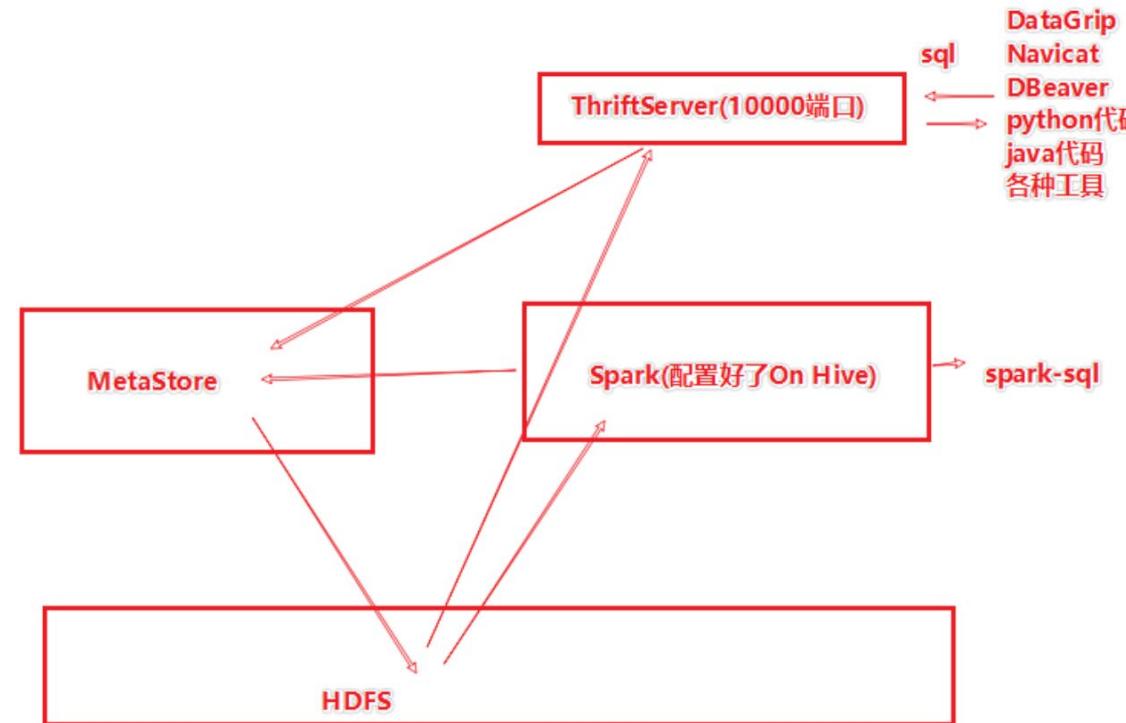
- 7.1 概念
- 7.2 客户端工具连接
- 7.3 代码JDBC连接



## 7.1 概念

Spark中有一个服务叫做: ThriftServer服务, 可以启动并监听在 10000 端口

这个服务对外提供功能, 我们可以用数据库工具或者代码连接上来 直接写SQL即可操作spark



当使用ThriftServer后, 相当于是一个持续性的Spark On Hive集成模式. 它提供10000端口, 持续对外提供服务, 外部可以通过这个端口连接上来, 写SQL, 让Spark运行.



## 7.2 配置

1. 确保已经配置好了Spark On Hive
2. 启动ThriftServer即可

```
1 # 如果是老师提供的虚拟机，请执行如下操作:  
2  
3 # 授权  
4 chmod -R 777 /export/server/spark  
5 # 切换到hadoop账户  
6 su - hadoop  
7 # 启动  
8 $SPARK_HOME/sbin/start-thriftserver.sh \  
9 --hiveconf hive.server2.thrift.port=10000 \  
10 --hiveconf hive.server2.thrift.bind.host=node1 \  
11 --master local[2]  
12 # master选择local，每一条sql都是local进程执行  
13 # master选择yarn，每一条sql 都是在YARN集群中执行
```

Shell | 复制代码

```
1 # 如果是你们自己的虚拟机  
2 # 直接在root账户下启动即可  
3 $SPARK_HOME/sbin/start-thriftserver.sh \  
4 --hiveconf hive.server2.thrift.port=10000 \  
5 --hiveconf hive.server2.thrift.bind.host=node1 \  
6 --master local[2]  
7 # master选择local，每一条sql都是local进程执行  
8 # master选择yarn，每一条sql 都是在YARN集群中执行
```

Shell | 复制代码



## 7.3 测试

### 客户端工具测试

DBeaver 21.1.0 - test

文件(F) 编辑(E) 导航(N) 搜索(A) SQL 编辑器 数据库(D) 窗口(W) 帮助(H)

数据库导航 项目 \* <node1> Script-2 test sales

输入表格名称的一部分

node1 - node1:10000

default

- 表
  - pos\_sales\_data
  - sales
    - 列
    - 唯一键
    - 外键
    - 引用
  - test
    - 列
    - 唯一键
    - 外键
    - 引用
  - xxx
  - 视图
  - 存储过程
  - 数据类型
- test
- global\_temp

test

输入一个 SQL 表达式来过滤结果 (使

id	name
1	哈哈
2	美美



## 7.3 测试

### 代码测试

我们课堂上使用的远程的Python解释器，同时要使用pyhive的包来操作。

为了安装pyhive包需要安装一堆linux软件，执行如下命令进行linux软件安装：

```
yum install zlib-devel bzip2-devel openssl-devel ncurses-devel sqlite-devel readline-devel tk-devel libffi-devel gcc make gcc-c++ python-devel cyrus-sasl-devel cyrus-sasl-plain cyrus-sasl-gssapi -y
```

安装好前置依赖软件后，安装pyhive包

```
/export/server/anaconda3/envs/pyspark/bin/python -m pip install -i https://pypi.tuna.tsinghua.edu.cn/simple pyhive pymysql sasl thrift thrift_sasl
```



## 7.3 测试

### 代码测试

Python | 复制代码

```
1 # coding:utf8
2 # 以JDBC模式 连接分布式Spark引擎
3 # 这个包是用来 以python代码 连接hive使用
4 from pyhive import hive
5
6 if __name__ == '__main__':
7     # 获取到Hive(Spark ThriftServer)的连接
8     conn = hive.Connection(host="node1", port=10000, username='hadoop')
9     # 获取一个游标对象，用来执行sql
10    cursor = conn.cursor()
11
12    # 执行sql 使用executor API
13    cursor.execute("SELECT * FROM test")
14    # 执行后，使用fetchall API 获取全部的返回值，返回值是一个List对象
15    result = cursor.fetchall()
16
17    # 打印输出
18    print(result)
```



## 总结

分布式SQL执行引擎就是使用Spark提供的ThriftServer服务，以“后台进程”的模式持续运行，对外提供端口。

可以通过客户端工具或者代码，以JDBC协议连接使用。

SQL提交后，底层运行的就是Spark任务。

相当于构建了一个以MetaStore服务为元数据，Spark为执行引擎的数据库服务，像操作数据库那样方便的操作SparkSQL进行分布式的SQL计算。



黑马程序员  
[www.itheima.com](http://www.itheima.com)

传智教育旗下高端IT教育品牌