法律声明

- 本课件包括：演示文稿，示例，代码，题库，视频和声音等，小象学院拥有完全知识产权的权利；只限于善意学习者在本课程使用，不得在课程范围外向任何第三方散播。任何其他人或机构不得盗版、复制、仿造其中的创意，我们将保留一切通过法律手段追究违反者的权利。

- 课程详情请咨询
  - 微信公众号：小象学院
  - 新浪微博：小象AI学院

大纲

- SparkSQL概述及原理
- DataFrame与DataSet
- SparkSQL程序设计

大纲

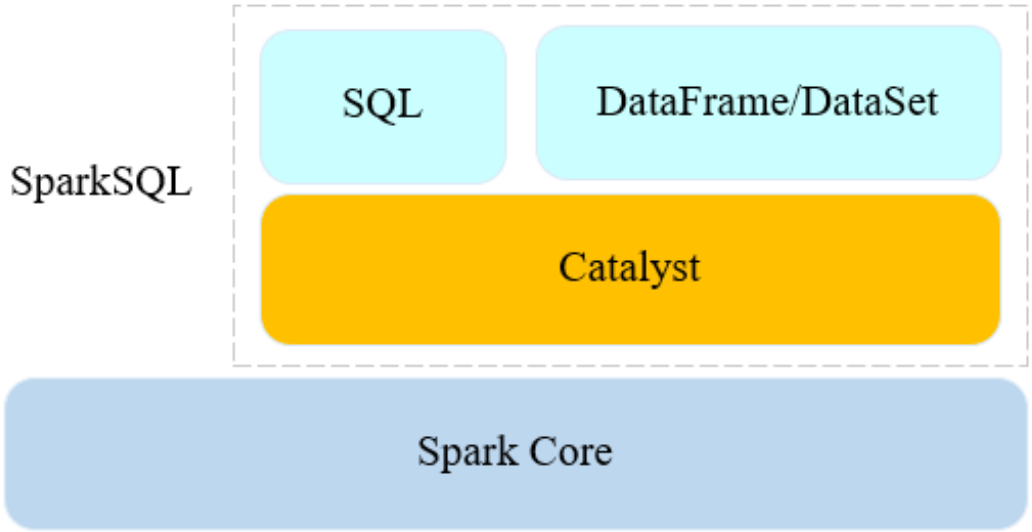
SparkSQL概述及原理

SparkSQL概述

- 从Spark1.0开始，正式成为生态系统的一员
- 专门处理结构化数据的Spark重要组件
- 提供了两种操作数据的方式
  - SQL查询
  - DataFrame和DataSet API
- Spark SQL = Schema + RDD

SparkSQL概述

- 提供了非常丰富的数据源API
  - 如:Text、JSON、Parquet、MySQL等
- 在Spark上实现SQL引擎
  - 提供高伸缩性API：DataFrame和DataSet API
  - 提供高效率的查询优化引擎：Catalyst Optimizer



SparkSQL-DataFrame

- RDD + Schema
  - 以行为单位构成的分布式数据集合，按照列赋予不同的名称
- 对select，filter，aggregation和sort等操作符的抽象
- 在Spark1.3之前，被称为SchemaRDD

引入SparkSQL的目的

- 为什么要引入SparkSQL
  - 写更少的代码
  - 读更少的数据
  - 让优化器自动优化程序
  - 轻松享受Spark高效的性能

SparkSQL-写更少的代码

使用RDD：

```
sc.textFile(inPath)
.flatMap(_.split("\t"))
.map((_,1))
.reduceByKey(_ + _)
.collect
```

使用SQL：

```
select word,count(1)
from words
group by word
```

使用DataFrame：

```
wrodDF.select("userId")
    .groupBy("userId")
    .count()
```

SparkSQL-写更少的代码

读/写复杂的数据结构更加便捷

- 读JSON文件

  val josnDF = spark.read.json("/tmp/user_json")
- 读Parquet

  val parquetDF = spark.read.parquet("/tmp/user_parquet")
- 写JSON文件

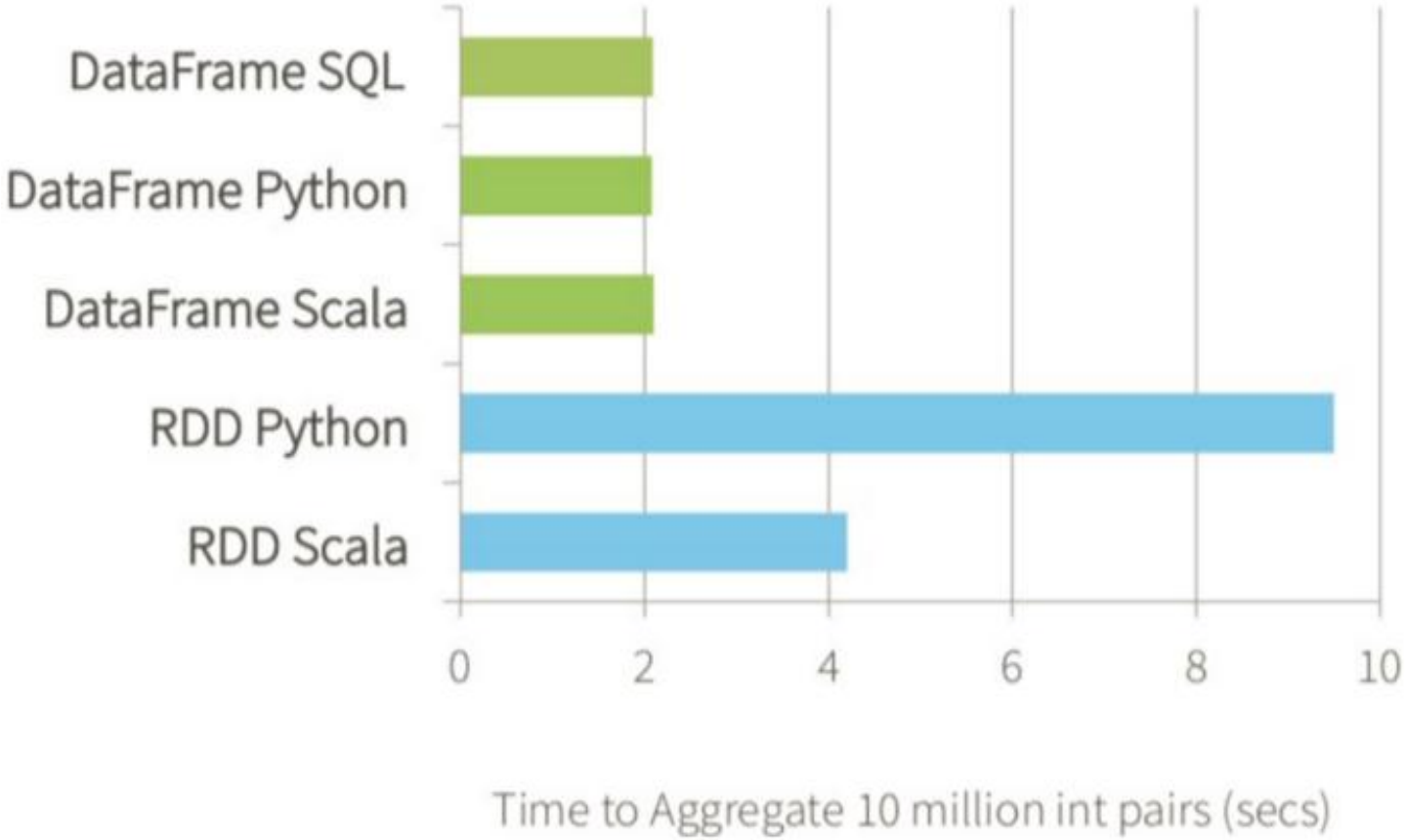
  userCoreDF2.write.json("tmp/user_json")
- 写Parquet

  userCoreDF2.write.parquet("/tmp/user_parquet")

SparkSQL-读更少的数据

- 采用更高效的数据格式保存数据
- 使用列式存储格式（比如parquet）
- 使用统计数据自动跳过数据（如：min、max）
- 使用分区
- 查询下推：将谓词下推到存储系统执行

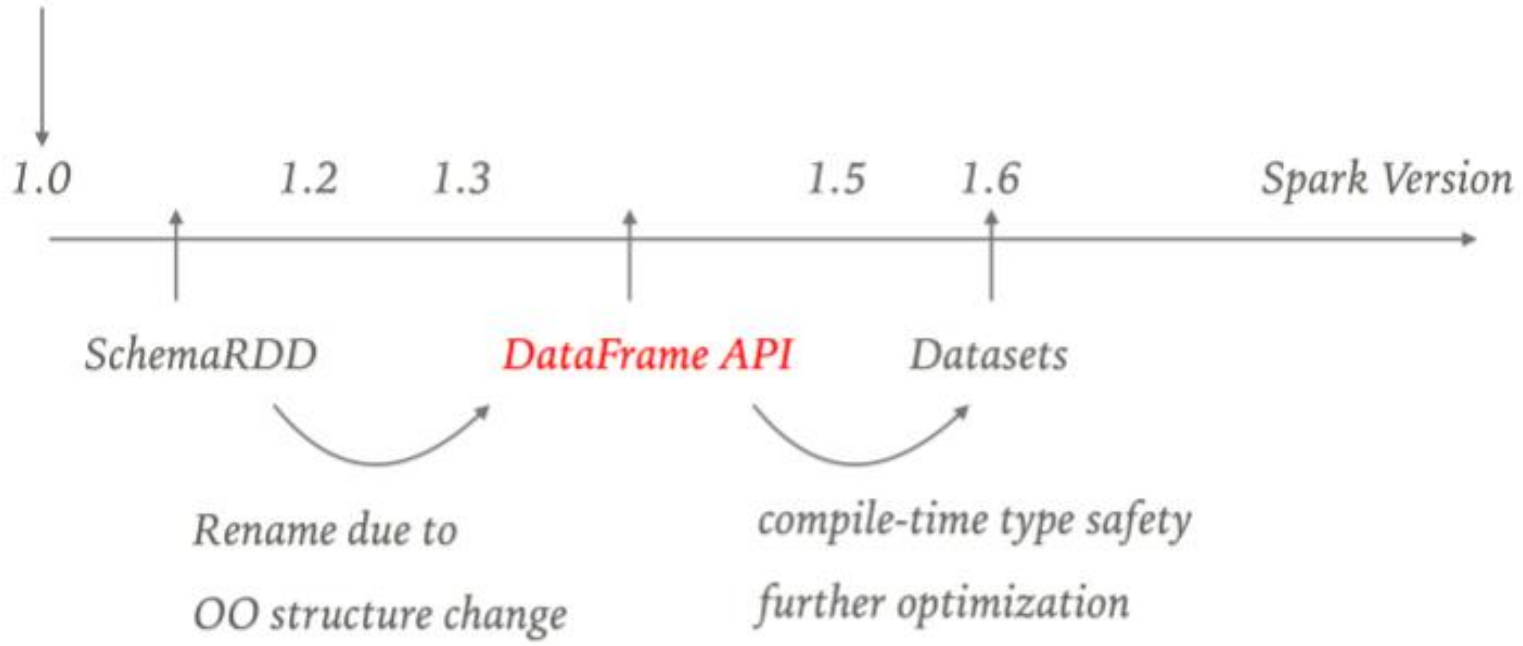
SparkSQL-更高效的性能



大纲

DataFrame与DataSet

SparkSQL API演化



RDD API

- JVM对象组成的弹性分布式数据集
- 不可变且具有容错能力
- 可处理结构化和非结构化数据
- Transformation和Action算子

RDD API的局限性

- 没有Schema
- 用户自己优化程序
- 从不同的数据源读取数据非常复杂
- 合并多个数据源的数据非常困难

DataFrame

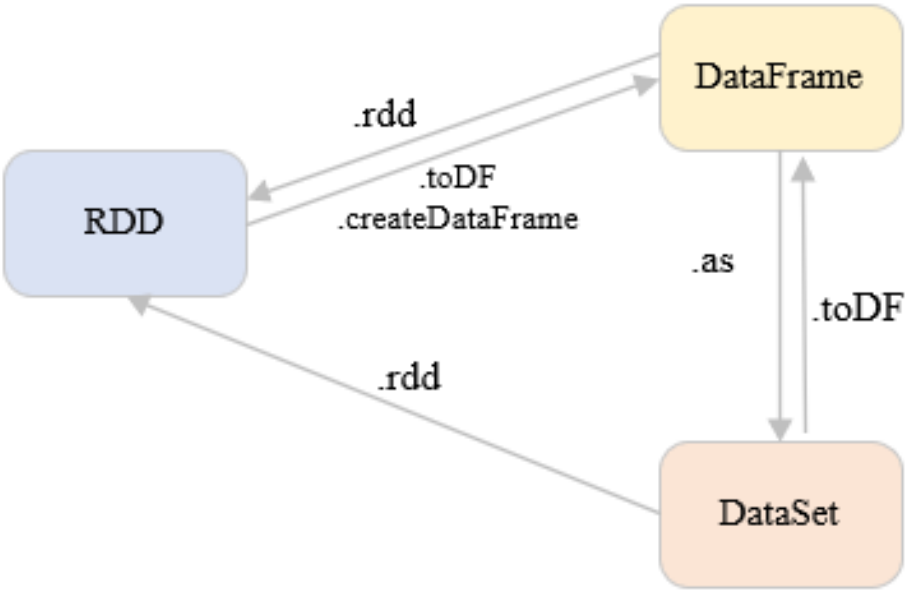
- Row对象组成的分布式数据集合
- 不可变且具有容错能力
- 处理结构化数据
- 内置的优化器可自动优化程序
- 丰富的数据源API
- 局限性：运行时检查

DataSet

- 扩展自DataFrame API，提供编译时类型安全，面向对象风格的API
- DataSet API
  - 类型安全
  - 高效：代码生成编解码器，序列化更高效
  - 协作：DataSet与DataFrame可互相转换
    - ✓ DataFrame = DataSet[Row]

RDD、DataFrame、DataSet的关系

- val parquetDF = spark.read.parquet(inpath) // parquet -> dataframe
- val ds = parquetDF.as[UserCore] // dataframe -> dataset
- val df = ds.toDF() // dataset -> dataframe
- val dsRdd = ds.rdd // dataset -> rdd
- val dfRdd = df.rdd // dataframe -> rdd



Tungsten计划

- 目标
  - 持续优化CPU和Memory的使用率
  - 让性能逼近现代硬件的极限
  - 让Spark足够快，未来处于领先地位

CPU成为了新的瓶颈

- 网络和磁盘配置提高很多
  - 如今带宽可达到10GB
  - 高带宽的SSD等
- Spark的IO模块已经得到了优化
  - 避免读取不需要的数据
  - 新的shuffle和network模块
- 新型数据存储格式
  - 列式存储格式，如Parquet
  - 序列化和hash是cpu密集型

Tungsten优化

- 运行时代码生成
- 利用cpu cache本地性
- Off-heap内存管理

大纲

SparkSQL程序设计

SparkSQL程序编写流程

- 创建SparkSession对象
  - 封装了spark sql执行环境信息，是Spark SQL程序的唯一入口
- 创建DataFrame或者DataSet
  - Spark SQL支持丰富的的数据源
- 在DataFrame/DataSet之上进行transformation和action
- 返回结果
  - 以不同的格式保存到HDFS
  - 直接打印结果

创建SparkSession

- 创建SparkSession对象

```
val spark = SparkSession
    .builder()
    .master("local[2]")
    .appName("SparkSqlDemo")
    .config(conf)
    .getOrCreate()
```

- SparkSession内部封装了SparkContext

创建DataFrame或DataSet

- 提供了读写各种格式数据的API

DataFrame/DataSet的operation

Actions
(DS/DF → console/output)

- collect
- count
- first
- foreach
- reduce
- take
- ...

For DataFrame & Dataset

Typed transformations
(DS → DS)

- map
- select
- filter
- flatMap
- mapPartitions
- join
- groupByKey
- interset
- repartition
- where
- sort
- ...

For Dataset

Untyped transformations
(DF → DF)

- agg
- col
- cube
- drop
- groupBy
- join
- rollup
- select
- withColumn
- ...

For DataFrame

DataFrame与DataSet

- DataFrame = DataSet[Row]
  - Row表示一行数据
  - RDD、DataFrame与DataSet之间可以互相转化
- DataFrame
  - 内部没有数据类型，统一为Row
  - DataFrame是一种特殊类型的DataSet
- DataSet
  - 内部数据有类型，需要由用户定义

通过RDD创建DataFrame方法1

- 定义case class，作为RDD的schema
- 直接通过RDD.toDF将RDD转换为DataFrame

```
case class UserCore(userId : String,age : Int,gender : String,core : Int)
val sc = sparkSession.sparkContext
val userCoresRdd = sc.textFile(inpath)
import spark.implicits._
val userCoreRdd = userCoresRdd.map(_.split("\t")).map(cols =>
UserCore(cols(0),cols(1).toInt,cols(2),cols(3).toInt))
val userCoreDF = userCoreRdd.toDF()
userCoreDF.take(2)
userCoreDF.count
```

通过RDD创建DataFrame方法2

- 使用StructType和StructField定义RDD schema
- 使用SaprkSession的createDataFrame创建DataFrame

```
val userCoreSchema = StructType(
  List(
        StructField("userId",StringType,true),
        StructField("age",IntegerType,true),
        StructField("gender",StringType,true),
        StructField("core",IntegerType,true)
  )
)
val userCoreRdd = userCoresRdd.map(_.split("\t")).map(cols =>
Row(cols(0).trim,cols(1).toInt,cols(2).trim,cols(3).toInt))
val userCoreDF =
sparkSession.createDataFrame(userCoreRdd,userCoreSchema)
userCoreDF.count
```

JSON Source API创建DataFrame

1. 通过SparkSession的read.format(“json”).load(inpath)创建DataFrame

2. 通过SparkSession的read.json(inpath)创建DataFrame

```
val josnDF1 =
spark.read.format("json").load("hdfs://192.168.183.100:9000/tmp/user_json")
println(“1. json dataframe schema ->" + josnDF1.schema)

val josnDF2 = spark.read.json("hdfs://192.168.183.100:9000/tmp/user_json")
println(“2. json dataframe schema ->" + josnDF2.schema)
josnDF2.show()
```

JDBC Source API创建DataFrame

- 方法1

```
val jdbcDF1 = spark.read
    .format("jdbc")
    .option("url", "jdbc:mysql://192.168.183.101:3306")
    .option("dbtable", "hive.TBLS")
    .option("user", "hive")
    .option("password", "hive123")
    .load()
jdbcDF1.show()
```

- 方法2

```
val connectionProperties = new Properties()
connectionProperties.put("user", "hive")
connectionProperties.put("password", "hive123")
val jdbcDF2 = spark.read
  .jdbc("jdbc:mysql://192.168.183.101:3306", "hive.TBLS", connectionProperties)
jdbcDF2.show()
```

DataFrame查询操作

- 通过DSL操作

```
val result = csvDF.filter("age > 20")
                  .select("gender","core")
                  .groupBy("gender")
                  .sum("core")
result.show()
```

- 注册临时视图，通过SQL查询

```
csvDF.createOrReplaceTempView("user_core")
spark.sql("select gender,sum(core) " +
                  "from user_core " +
                  "where age > 20 " +
                  "group by gender")
                  .show()
```

DataFrame写出操作

- dataframe将数据保存成json文件到hdfs

```
userCoreDF2.write
  .mode(SaveMode.Overwrite)
  .json("hdfs://192.168.183.100:9000/tmp/user_json")
```

- dataframe将数据保存成parquet文件到hdfs

```
userCoreDF2.write
  .mode(SaveMode.Overwrite)
  .parquet("hdfs://192.168.183.100:9000/tmp/user_parquet")
```

- dataframe将数据保存成csv文件到hdfs

```
userCoreDF2.write
  .mode(SaveMode.Overwrite)
  .csv("hdfs://192.168.183.100:9000/tmp/user_csv")
```

DataFrame写出操作

- dataframe将数据写入到MySQL表

```
val connectionProperties = new Properties()
connectionProperties.put("user", "hive")
connectionProperties.put("password", "hive123")
userCoreDF2.write
  .mode(SaveMode.Overwrite)
  .jdbc("jdbc:mysql://192.168.183.101:3306", "hive.user_core",
connectionProperties)
```

SparkSQL与Hive结合

- 使用Spark SQL访问Hive
  - 将hive安装包中conf/hive-site.xml配置文件拷贝到spark安装包的conf目录下
  - 将mysql驱动jar包拷贝到spark安装包的jars目录下
  - 启动：spark安装包下bin/spark-sql

分布式SQL查询引擎

- 配置HiveServer2 Thrift服务的访问地址和端口号
  - 方法1：在hive-site.xml文件中添加hiveserver2的配置信息

```
<property>
  <name>hive.server2.thrift.port</name>
  <value>10010</value>
</property>
<property>
  <name>hive.server2.thrift.bind.host</name>
  <value>192.168.183.100</value>
</property>
```

  - 方法2：在环境变量中配置

```
export HIVE_SERVER2_THRIFT_PORT=10010
export HIVE_SERVER2_THRIFT_BIND_HOST=192.168.183.100
```

  - 方法3：在启动Spark Thrift Server的时候以参数的形式指定

```
--hiveconf hive.server2.thrift.port=10010
--hiveconf hive.server2.thrift.bind.host=192.168.183.100
```

分布式SQL查询引擎

- 启动Spark Thrift Server
  - Local模式启动

```
sbin/start-thriftserver.sh \
--hiveconf hive.server2.thrift.port=10010 \
--hiveconf hive.server2.thrift.bind.host=192.168.183.100
```

  - yarn-client模式启动

```
sbin/start-thriftserver.sh \
--hiveconf hive.server2.thrift.port=10010 \
--hiveconf hive.server2.thrift.bind.host=192.168.183.100 \
--master yarn \
--deploy-mode client \
--executor-memory 3g \
--executor-cores 1 \
--num-executors 2 \
--driver-cores 1 \
--driver-memory 1g
```

连接分布式SQL查询引擎

- 通过bin/beeline使用JDBC访问

./bin/beeline
beeline> !connect jdbc:hive2://192.168.183.100:10010

- 通过Java Api使用JDBC访问

```
Class.forName("org.apache.hive.jdbc.HiveDriver")
val (url,username,userpasswd)=("jdbc:hive2://192.168.183.100:10010/rel","hadoop","hadoop")
val conn = DriverManager.getConnection(url,username,userpasswd)
val sql="select * from rel.user_core_info"
val stat = conn.prepareStatement(sql)
val rs = stat.executeQuery()
while(rs.next()){
  println("userId -> " + rs.getString(1))
  println("name -> " + rs.getString(2))
}
rs.close()
stat.close()
conn.close()
```

疑问

- 小象问答官网
  - http://wenda.chinahadoop.cn

联系我们

小象学院：互联网新技术在线教育领航者

- 微信公众号：小象学院
- 新浪微博：小象AI学院