SparkSQL DataFrame

目录

Spark SQL 背景介绍

为什么在大数据领域仍使用SQL

- SQL能够跟现有系统进行很好集成
- 跟现有的JDBC/ODBC BI系统兼容
- 很多工程师习惯使用SQL
- 相比于MapReduce，SQL更容易表达

Spark SQL是什么

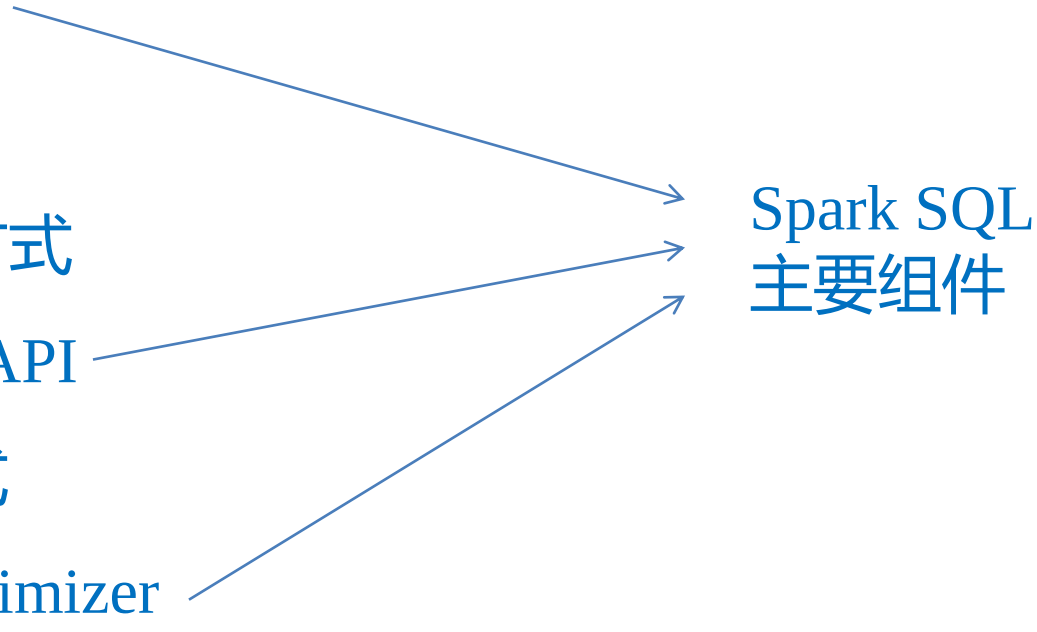
- 从Spark 1.0开始，成为Spark生态系统一员
- 专门处理结构化数据（比如DB, Json）的Spark组件
- 提供了三种操作数据的方式
  - SQL Queries
  - DataFrames API
  - Datasets API
- Spark SQL = Schema + RDD

Spark SQL引入的主要动机

- 更快地编写和运行Spark程序
  - 编写更少的代码
  - 读取更少的数据
  - 让优化器自动优化程序，释放程序员的工作

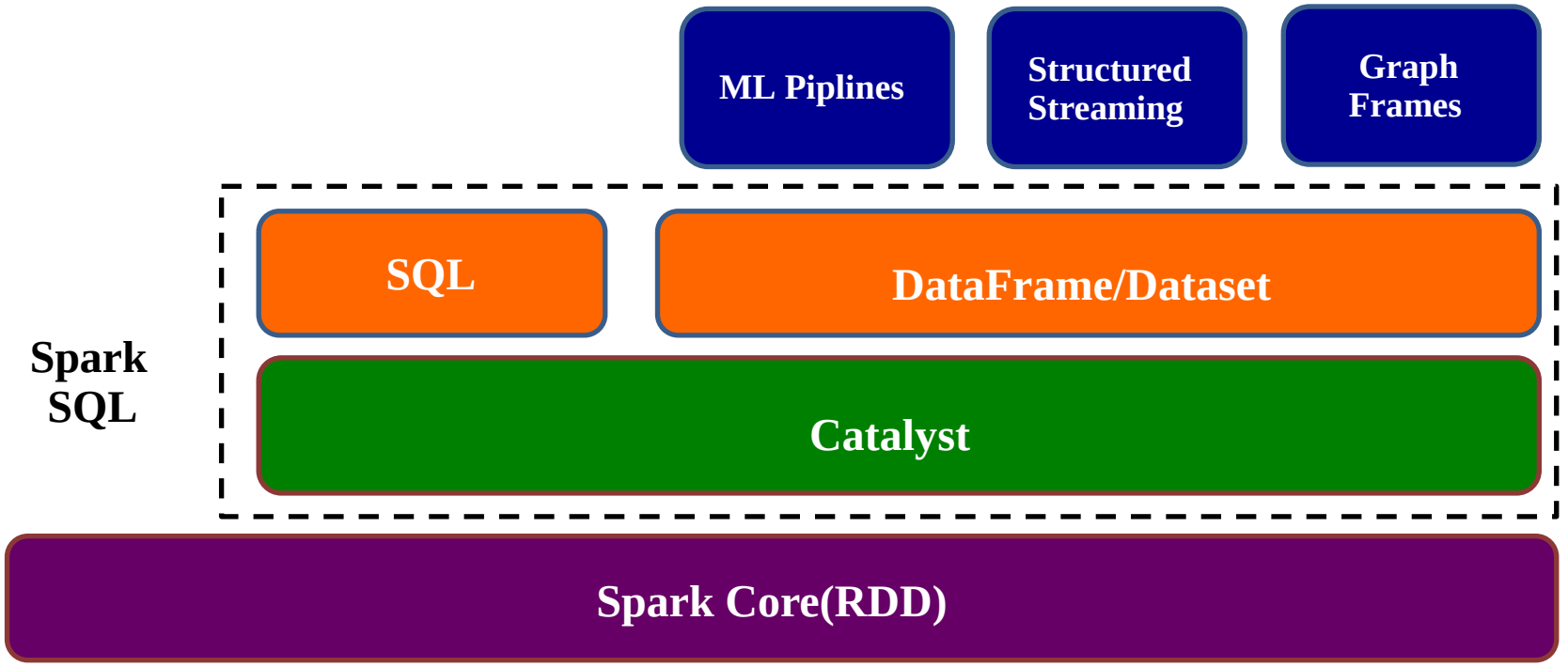
Spark SQL：不仅仅是“SQL”

- SQL能够对各种数据源进行ETL操作
  - 解决方案：Data Source API
- 在Spark上实现SQL引擎
  - 通过一种具备高伸缩性的方式
    - 解决方案：DataFrame API
  - 通过一种具备高效率的方式
    - 解决方案：Catalyst Optimizer



Spark SQL 主要组件

Spark SQL总体架构



Spark SQL

- 使用SQL
  - 如果你非常熟悉SQL语法，则使用SQL
- 使用DataFrame/Dataset
  - DSL(Domain Specific Language)
  - 采用更通用的语言（Scala，Python）表达你的查询需求
  - 使用DataFrame更快的捕获错误

SQL DataFrame		
Syntax error example	“SELECT id FROM table”	df.select(“id”)
Caught at	RunDme	Compile Time

Spark SQL：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Spark 1.3之前，被称为SchemaRDD

DataFrame：编写更少的代码（input & output）

- 提供了读写各种格式数据的API

DataFrame：编写更少的代码（input & output）

- 提供了读写各种格式数据的API

```
df = sqlContext.read \
  .format("json") \
  .option("samplingRatio", "0.1")  \
.load("/home/michael/data.json")
```

```
df.write  \
  .format("parquet") \
  .mode("append") \
  .partitionBy("year") \
  .saveAsTable("fasterData")
```

Read and write

Spark 2.x 融入了sparkSession

```
df = spark.read \
  .format("json") \
  .option("samplingRatio", "0.1")  \
.load("/home/michael/data.json")
```

```
df.write  \
  .format("parquet") \
  .mode("append") \
  .partitionBy("year") \
  .saveAsTable("fasterData")
```

DataFrame：编写更少的代码（input & output）

- 提供了读写各种格式数据的API

```
df = sqlContext.read \
.format("json") \
.option("samplingRatio", "0.1")  \
.load("/home/michael/data.json")
```

```
df.write  \
.format("parquet") \
.mode("append") \
.partitionBy("year") \
.saveAsTable("fasterData")
```

Builder methods are used to specify:

- **Format**
- **Partitioning**
- **Handling of existing data**
- **and more**

DataFrame：编写更少的代码（input & output）

- 提供了读写各种格式数据的API

```
df = sqlContext.read \
.format("json") \
.option("samplingRatio", "0.1")   \
.load("/home/michael/data.json")
```

```
df.write  \
  .format("parquet") \
  .mode("append") \
  .partitionBy("year") \
  .saveAsTable("fasterData")
```

load(…), save(…) or saveAsTable(…) functions create new builders for doing I/O

DataFrame：编写更少的代码（writeless）

```
| 20|    eng|  Andy|   98|
| 20|chinese|  Andy|   93|
| 20|   math|  Andy|   88|
| 19|   math|Justin|   82|
| 19|    eng|Justin|   91|
| 19|chinese|Justin|   73|
| 19|   math|  jams|   88|
| 19|    eng|  jams|   90|
| 19|chinese|  jams|   71|
```

```
val rdd = studentDF.rdd
// rdd.take(2)
val r = rdd.map{line =>
  (line(2).toString,(line(3).toString.toInt, 1))
}.reduceByKey((x,y) => (x._1 + y._1, x._2 + y._2)).map(x => (x._1,x._2._1 / x._2._2))
r.collect foreach println
```

Using SparkSQL

```
val resultDF = sqlContext
  .sql("select name, avg(score) from student group by name")
```

Using DataFrame

```
val resultDF = studentDF.groupBy("name")
  .agg(Map("score" -> "avg"))
```

DataFrame：读取更少的数据

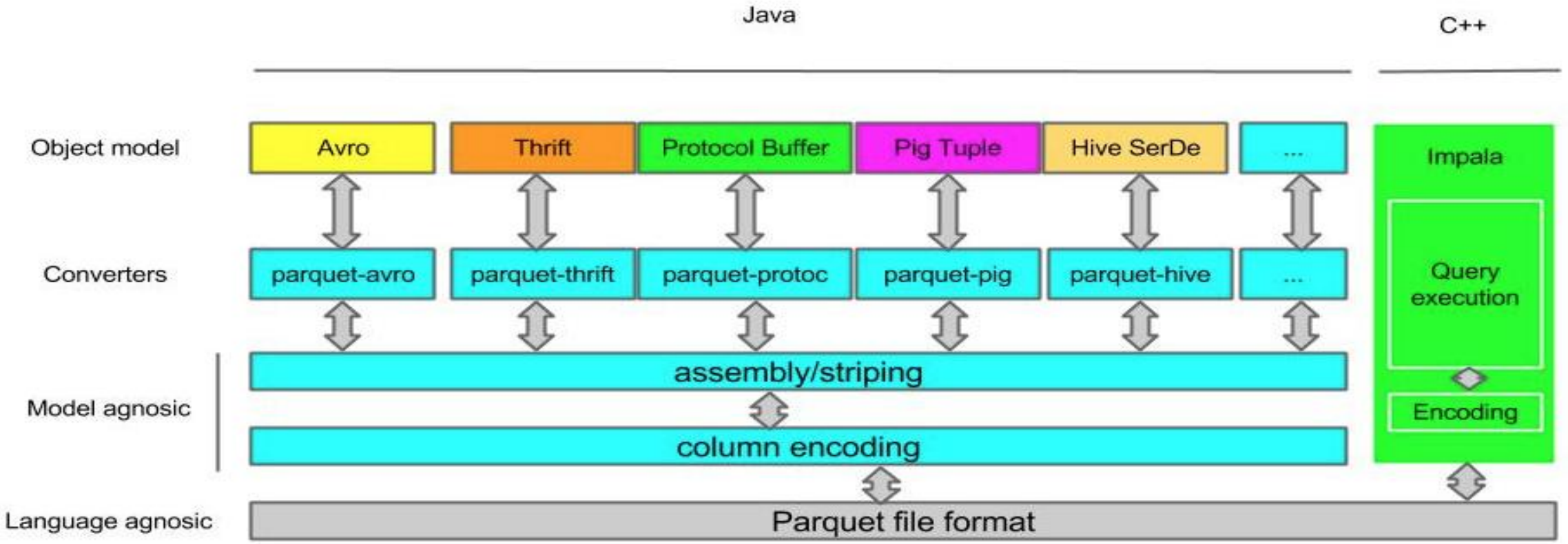
- 采用更高效的数据格式保存数据
- 使用列式存储格式（比如parquet）
- 使用分区（比如/year=2014/month=02/…）
- 使用统计数据自动跳过数据（比如min、max）
- 查询下推：将谓词下推到存储系统执行

parquet & orc

- Parquet 和 orc 都是列式存储格式，列式存储具备以下特点：
- 1.可以跳过不符合条件的数据，只读取需要的数据，降低IO数据量。
- 2.压缩编码可以降低磁盘存储空间。由于同一列的数据类型是一样的，可以使用更高效的压缩编码（例如Run Length Encoding和Delta Encoding）进一步节约存储空间。
- 3.只读取需要的列，支持向量运算，具备更好的扫描性能。

parquet & orc

- Parquet是嵌套结构（层级，树状），先通过对每一层的查询，然后在相关叶子上进行数据查询。

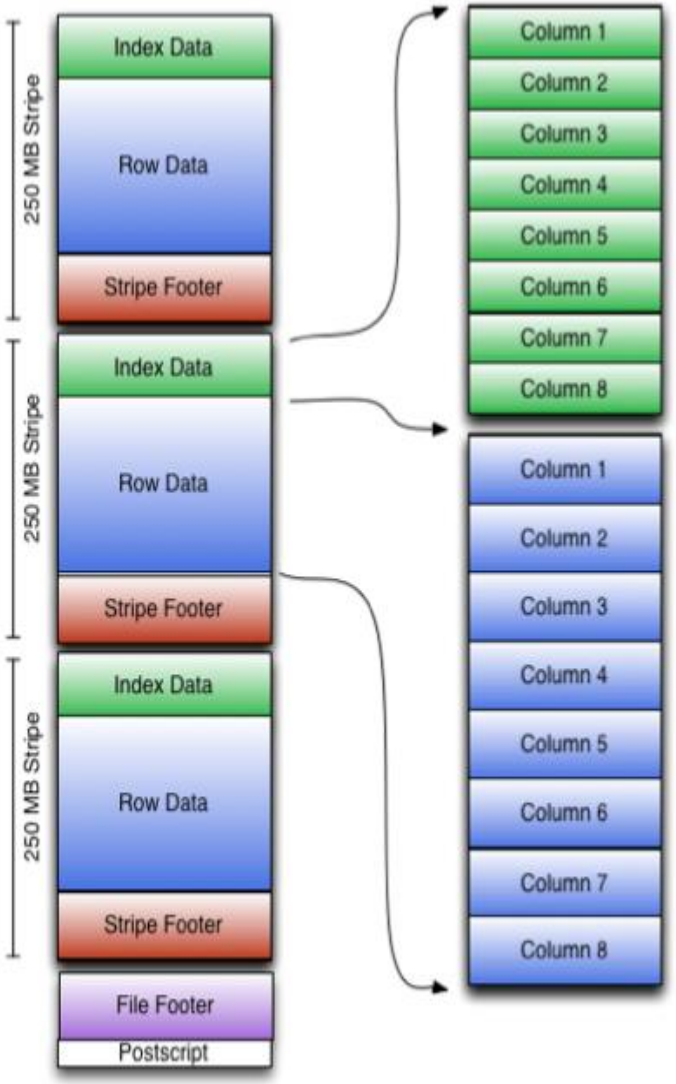


parquet & orc

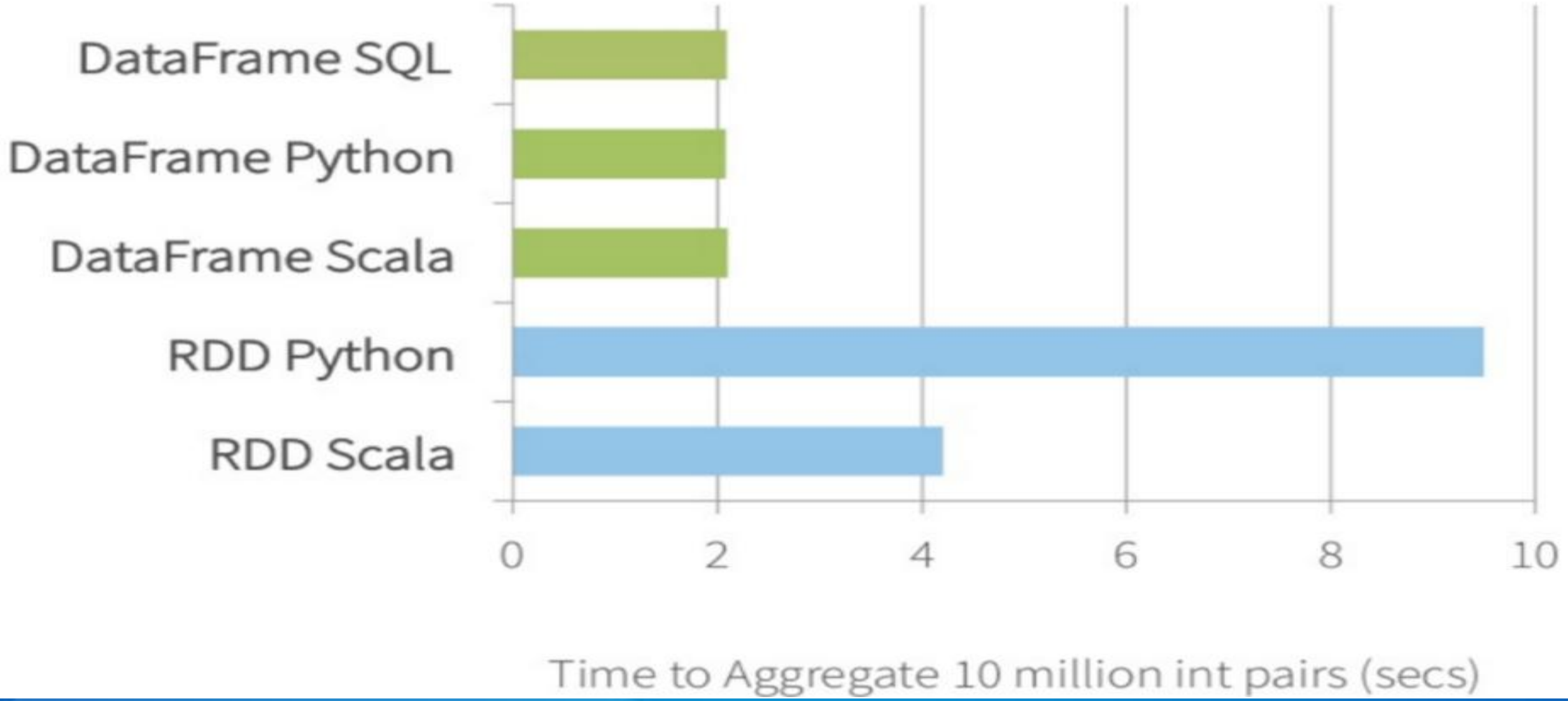
- Parquet 是一个列式存储文件格式，在同一个数据文件中保存一行中的所有数据，以确保在同一个节点上处理时一行的所有列都可用。
- Parquet 设置 HDFS块大小和最大数据文件大小为 1GB，一组行的数据会重新排列，以便第一行所有的值被重组为一个连续的块，然后是第二行的所有值。
- Parquet可以表达嵌套结构，用definition level和repetition level两个值分别表达在整个嵌套格式中，最深嵌套层数，和同一个嵌套层级中第几个值。

parquet & orc

- ORC文件：保存在文件系统上的普通二进制文件，一个ORC文件中可以包含多个stripe，每一个stripe包含多条记录，这些记录按照列进行独立存储，对应到Parquet中的row group的概念。
- Index Data：一个轻量级的index，默认是每隔1W行做一个索引。这里做的索引应该只是记录某行的各字段在Row Data中的offset，还包括每个Column的max和min值。
- Row Data：存的是具体的数据，先取部分行，然后对这些行按列进行存储。每个列进行了编码，分成多个Stream来存储。
- Stripe Footer：存的是各个Stream的类型，长度等信息。
- 文件级元数据(File Footer)：包括文件的描述信息PostScript、文件meta信息（包括整个文件的统计信息）、所有stripe的信息和文件schema信息。
- stripe：一组行形成一个stripe，每次读取文件是以行组为单位的，一般为HDFS的块大小，保存了每一列的索引和数据。



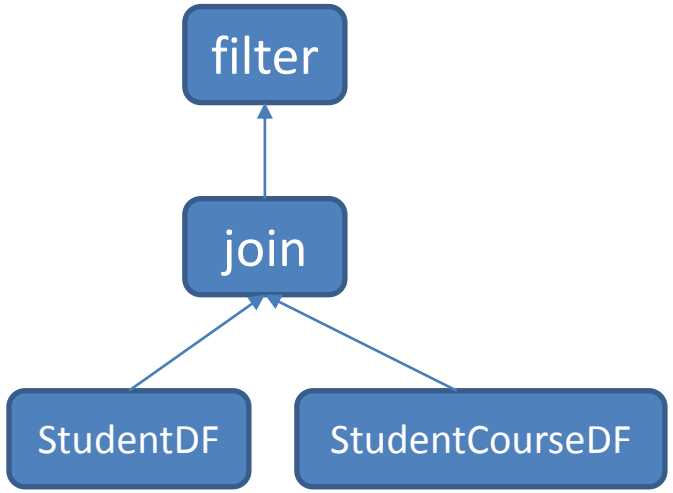
Spark SQL：更高的性能



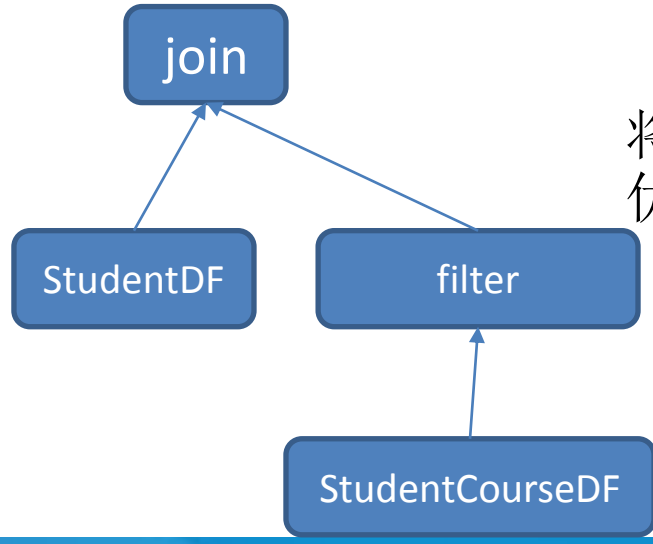
Spark SQL：更高的性能（join）

```
val studentDF = sqlContext.read.format("json").load(root + "student.json")
val studentCourse = sqlContext.read.format("json").load(root + "studentCourse.json")
val result = studentDF.alias("a").join(studentCourse.alias("b"),
  studentDF("course") === studentCourse("course"), "inner")
  .where("a.course = 'eng'").select(studentCourse.col("teacher"))
```

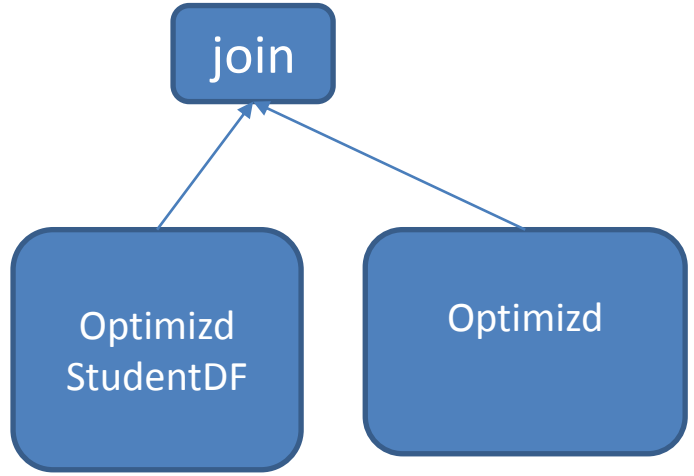
逻辑计划



物理计划

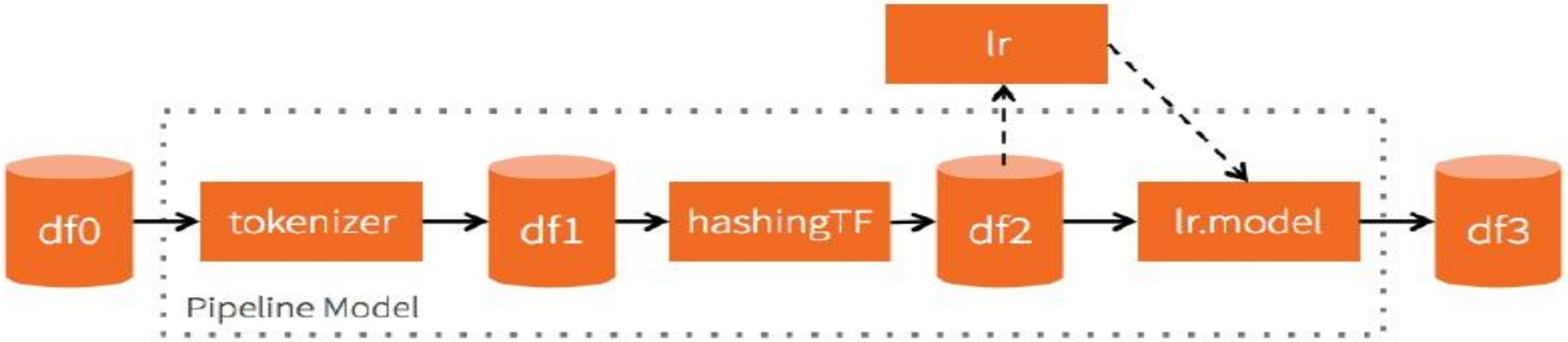


将两次调用
优化为一次



Spark SQL：更高的性能

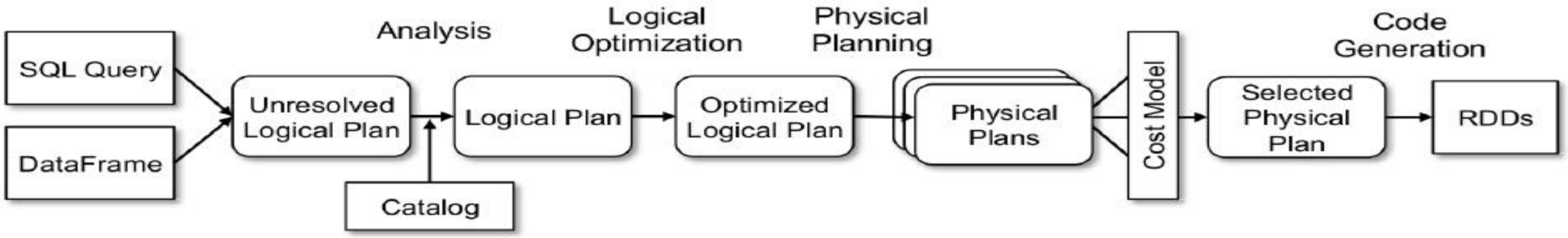
```
tokenizer = Tokenizer(inputCol="text", outputCol="words")
hashingTF = HashingTF(inputCol="words", outputCol="features")
lr = LogisticRegression(maxIter=10, regParam=0.01)
pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])

df = sqlCtx.load("/path/to/data")
model = pipeline.fit(df)
```



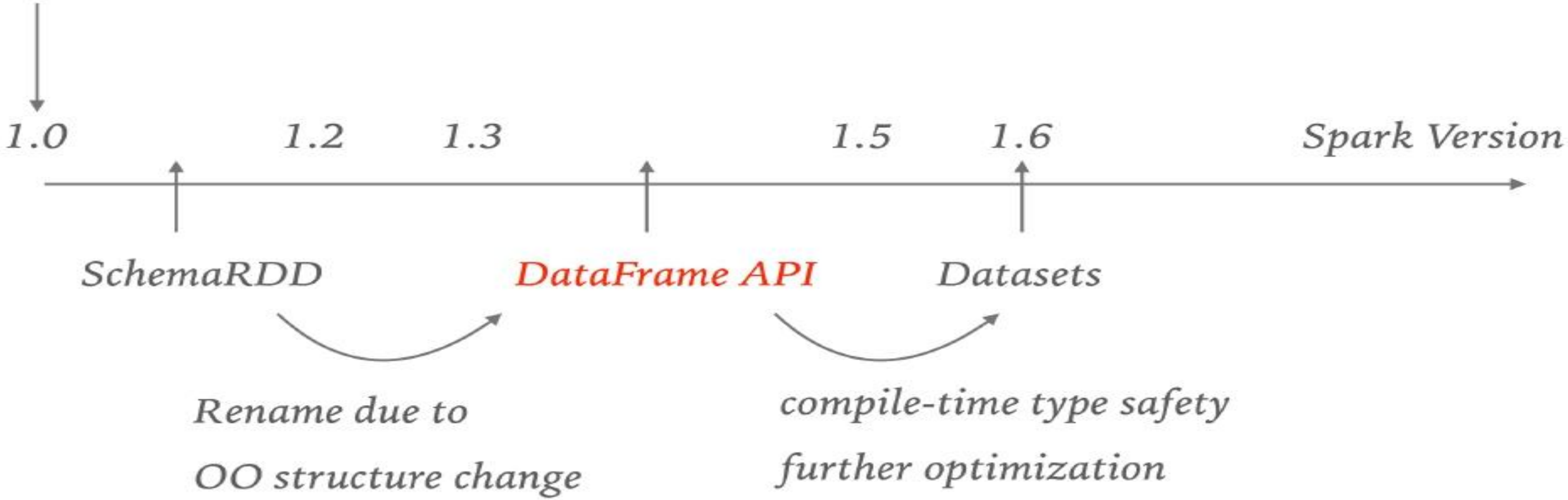

Spark SQL：更高的性能
How dose this all work?

Spark SQL： 核心优化器：Catalyst

- Plan OpAmizaAon & ExecuAon



Spark SQL API演化



RDD API（2011）

- JVM对象组成的分布式数据集合
- 不可变且具有容错能力
- 可处理结构化与非结构化数据
- 函数式转换

RDD API 的局限性

- 无Schema
- 用户自己优化程序
- 从不同的数据源读取数据非常困难
- 合并多个数据源中的数据也非常困难

DataFrame API（2013）

- 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
- 自带优化器Catalyst，可自动优化程序
- Data source API

DataFrame API 的局限性

- 运行时类型检查
- 不能直接操作domain对象
- 函数式编程风格

DataFrame API 的局限性

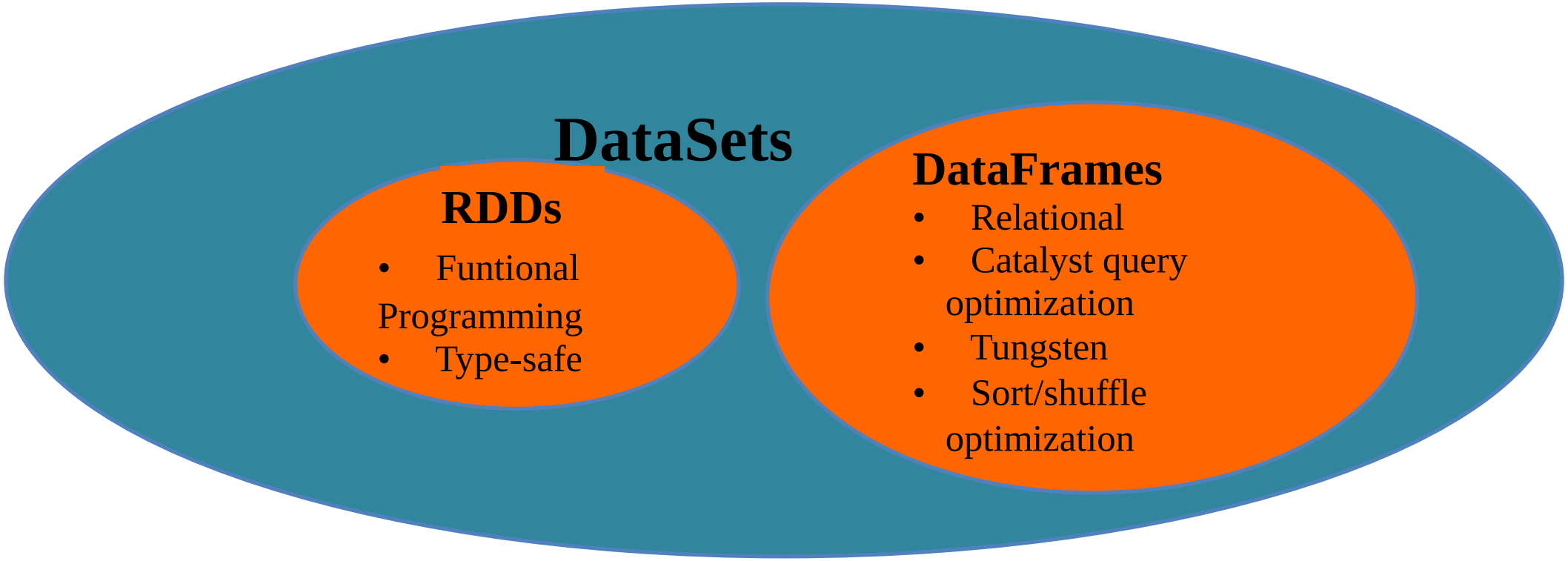
```
val dataframe = sqlContext.read.json("people.json”)
dataframe.filter("salary > 1000").show()

Throws Runtime exception
org.apache.spark.sql.AnalysisException: cannot resolve 'salary' given input columns age,
name;
```

```
 //Create RDD[Person]
val personRDD = sc.makeRDD(Seq(Person("A",10), Person("B",20)))

 //Create dataframe from a RDD[Person]
val personDF = sqlContext.createDataFrame(personRDD)

personDF.rdd
//We get back RDD[Row] and not RDD[Person]          Dataset 转换回来泛型会是person
```

Dataset

- 什么是Dataset
  - 扩展自DataFrame API，提供了编译时类型安全，面向对象风格的API
- Dataset AP
  - 类型安全：可直接作用在domain对象上 Dataset[Person]
  - 高效：代码生成编解码器，序列化更高效 //还没有提供具体实现
  - 协作：Dataset与Dataframe可相互转换

Dataset


DataSets
RDDs
• Funtional Programming
• Type-safe
DataFrames
• Relational
• Catalyst query optimization
• Tungsten
• Sort/shuffle optimization

Dataset ：编译时类型检查

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


val dataframe = sqlContext.read.json("people.json")

val ds : Dataset[Person] = dataframe.as[Person]
ds.filter(p => p.age > 25)

ds.filter(p => p.salary > 12500)

//error: value salary is not a member of Person
```

Dataset ： 作用在domain对象

```
//Create  RDD[Person]
val  personRDD  =  sc.makeRDD(Seq(Person("A",10),  Person("B",20)))

//Create  Dataset  from  a  RDD
val  personDS  =  sqlContext.createDataset(personRDD)

personDS.rdd
//We  get  back  RDD[Person]  and  not  RDD[Row]  in  Dataframe
```

Dataset ： 面向对象编程风格

```
case class Person(name: String, age: Int)
val dataframe = sqlContext.read.json("people.json")
val ds : Dataset[Person] = dataframe.as[Person]
// Compute histogram of age by name
val hist = ds.groupBy(_.name).mapGroups({
case (name, people) => {
  val list = people.map(_.age).toList
  (name, list)
}})
})
```

Dataset API : data sources

- GeneraAc Load/Save FunAons

```
val usersDF = spark.read.load(“/data/users.parquet")
val peopleDF = spark.read.format(“json”).load(“/data/people.json")
```

- Specific Data Sources
  - Parquet:

```
val parquetFileDF = spark.read.parquet("/data/users.parquet”)
parquetFileDF.write.parquet("/data/output.parquet”)
```

  - Json:

```
val jsonFileDF = spark.read.json(“/data/users.json”)
jsonFileDF .write.mode('append').json(“/data/output.json”)
```

- JDBC:

```
val jdbcDF = spark.read.format("jdbc").opAons(
Map("url" - - - > "jdbc:postgresql:dbserver",
"dbtable" - - - > "schema.tablename")).load()
```

Dataset API：operaAons

Actions

collect
count
first
foreach
reduce
take
…

Typed transformations

map
select
filter
flatMap
mapParAAons
join
groupByKey
interset
reparAAon
where
sort
…

Untyped transformations

agg
col
cube
drop
groupBy
join
rollup
select
withColumn
…

RDD/DataFrame/Dataset

Dataset will be stable in spark 2.0

Please use dataframe now until

spark 2.0 is stable！


Spark SQL 应用

数据集准备

- 数据集

  MovieLens 1M Dataset
- 相关数据文件
  - users.dat

    UserID::Gender:: Age::Occupation::Zip-code
  - movies.dat

    MovieID:: Title::Genres
  - ratings.dat

    UserID::MovieID::Rating::Timestamp

SparkSQL 命令行访问

- 准备数据

  将:: 分隔符转为 “ , ” 方便建立hive 表

  (1) 编写代码

  (2) shell 命令 cat ml-1m/users.dat | tr -s "::" ',' >> /tmp/data/users.dat

- ./bin/spark-sql

```
CREATE EXTERNAL TABLE user (
userid INT,
gender STRING,
age INT,
occupation STRING,
zipcode INT
)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ","
STORED AS TEXTFILE
LOCATION '/tmp/data';
```

测试一下表

```
SELECT * from USER limit 10;
```

SparkSQL 命令行访问hive表

- 使用SparkSQL 处理Hive Metastore 中的表
- 将hive-site.xml拷贝到Spark安装的conf 目录下
- 执行spark-sql

SparkSQL 程序设计

- 创建 SQLContext 对象

  封装了 spark sql 执行环境信息

- 创建DataFrame或Dataset

  Spark SQL 支持各种数据源

- 在 DataFrame 或 Dataset 之上进行转换和 action

  Spark SQL 提供了多种转换和 action 函数

- 返回结果

  保存到 HDFS 中，或直接打印出来

步骤1，创建SQLContext

- val sc: SparkContext // 已创建好的 SparkContext
- val sqlContext = new org.apache.spark.sql.SQLContext(sc)
- 将RDD隐式转换为 DataFrame

```
import sqlContext.implicits._
```

步骤2，创建 DataFrame 或Dataset

- 提供了读写各种格式数据的 API

步骤3，DataFrame 或Dataset 上进行 operation

- 提供了读写各种格式数据的 API

Untyped transformations
(DF -> DF)

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

Typed transformations
(DS -> DS)

- map
- select
- filter
- flatMap
- mapParKKons
- join
- groupByKey
- interset
- reparKKon
- where
- sort
- …

Actions
(DF/DS -> console/output)

- collect
- count
- first
- foreach
- reduce
- take
- …

DataFrame数据源

- RDD

  通过反射方式

  通过自定义schema方式

- json
- parquet
- Jdbc
- orc

DataFrame 反射方式

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

```
import org.apache.spark.sql.SQLContext
import org.apache.spark.sql.Row
case class User(userID: Long, gender: String, age: Int, occupation: String, zipcode: Int)
val usersRdd = sc.textFile("/tmp/ml-1m/users.dat")
val userRDD = usersRdd.map(_.split("::")).map(p => User(p(0).toLong, p(1).trim,
p(2).toInt, p(3), p(4).toInt))
val userDataFrame = userRDD.toDF()
userDataFrame.take(10)
userDataFrame.count()
```

RDD -> DataFrame： 显式注入 Schema

- 定义RDD schema（由StructField/StructType构成）
- 使用SQLContext.createDataFrame生成DF

```
import org.apache.spark.sql.{SaveMode, SQLContext, Row}
import org.apache.spark.sql.types.{StringType, StructField, StructType}
val schemaString = "userID gender age occupation zipcode"
val schema = StructType(schemaString.split(" ").map(fieldName =>
StructField(fieldName, StringType, true)))
val userRDD2 = usersRdd.map(_.split("::")).map(p => Row(p(0), p(1).trim, p(2).trim,
p(3).trim, p(4).trim))
val userDataFrame2 = sqlContext.createDataFrame(userRDD2, schema)
userDataFrame2.take(10)
userDataFrame2.count()
userDataFrame2.write.mode(SaveMode.Overwrite).json("/tmp/user.json")
userDataFrame2.write.mode(SaveMode.Overwrite).parquet("/tmp/user.parquet")
```

json -> DataFrame

- sqlContext.read.format("json").load(...)
- sqlContext.read.json(...)
- SQL

```
import org.apache.spark.sql.{SaveMode, SQLContext, Row}
import org.apache.spark.sql.types.{StringType, StructField, StructType}
val schemaString = "userID gender age occupation zipcode"
val schema = StructType(schemaString.split(" ").map(fieldName =>
StructField(fieldName, StringType, true)))
val userRDD2 = usersRdd.map(_.split("::")).map(p => Row(p(0), p(1).trim, p(2).trim,
p(3).trim, p(4).trim))
val userDataFrame2 = sqlContext.createDataFrame(userRDD2, schema)
userDataFrame2.take(10)
userDataFrame2.count()
userDataFrame2.write.mode(SaveMode.Overwrite).json("/tmp/user.json")
userDataFrame2.write.mode(SaveMode.Overwrite).parquet("/tmp/user.parquet")
```

parquet -> DataFrame

- sqlContext.read.format("json").load(...)
- sqlContext.read.json(...)
- SQL

```
val userParquetDF  = sqlContext.read.format("parquet").load("/tmp/user.parquet")
userParquetDF.take(10)
val userParquetDF2 = sqlContext.read.parquet("/tmp/user.parquet")
userParquetDF2.take(10)

CREATE TABLE user USING parquet
OPTIONS
(path “/tmp/user.parquet”)
```

jdbc -> DataFrame

- read.format("jdbc").options(…)
- SQL

```
export SPARK_CLASSPATH=<mysql-connector-java-5.1.26.jar>
val  jdbcDF =  sqlContext.read.format("jdbc").options(
val jdbcDF = sqlContext.read.format("jdbc").options(
Map(
“url” -> “jdbc:mysql://mysql_hostname:mysql_port/testDB",
"dbtable" -> "testTable")).load()
 CREATE TABLE user USING jdbc
 OPTIONS
(“jdbc:mysql://mysql_hostname:mysql_port/testDB”, “dbtable” -> “testTable”)
```

jdbc -> DataFrame

- read.format("jdbc").options(…)
- SQL

```
export SPARK_CLASSPATH=<mysql-connector-java-5.1.26.jar>
val  jdbcDF =  sqlContext.read.format("jdbc").options(
val jdbcDF = sqlContext.read.format("jdbc").options(
Map(
“url” -> “jdbc:mysql://mysql_hostname:mysql_port/testDB",
"dbtable" -> "testTable")).load()
 CREATE TABLE user USING jdbc
 OPTIONS
(“jdbc:mysql://mysql_hostname:mysql_port/testDB”, “dbtable” -> “testTable”)
```

csv -> DataFrame

- Github: https://github.com/databricks/spark-csv
- Maven: com.databricks:spark-csv_2.10:1.2.0

```
val  userCsvDF = sqlContext.read
        .format("com.databricks.spark.csv")
        .load(”/tmp/user.csv")
        .toDF(”userID", ”age")
```

Avro -> DataFrame

- Github: https://github.com/databricks/spark-avro
- Maven: com.databricks:spark-avro_2.10:2.0.1

```
val df = sqlContext.read
        .format("com.databricks.spark.avro")
        .load(”/tmp/user.avro")
```

Avro -> DataFrame

- Github: https://github.com/databricks/spark-avro
- Maven: com.databricks:spark-avro_2.10:2.0.1

```
val df = sqlContext.read
        .format("com.databricks.spark.avro")
        .load(“/tmp/user.avro”)
```

准备数据

- Json 数据

{"age":"45","gender":"M","occupation":"7","userID":"4","zipcode":"02460"}
{"age":"1","gender":"F","occupation":"10","userID":"1","zipcode":"48067"}

- 读取Json数据

```
scala> val userDF = sqlContext.read.json("/tmp/user.json")
userDF: org.apache.spark.sql.DataFrame = [age: string, gender: string, occupation: string, userID:
string, zipcode: string]
```

- 生成Json数据

```
scala> userDF.limit(5).write.mode("overwrite").json("/tmp/user2.json")
```

准备数据

- Json 数据

{"age":"45","gender":"M","occupation":"7","userID":"4","zipcode":"02460"}
{"age":"1","gender":"F","occupation":"10","userID":"1","zipcode":"48067"}

- 读取Json数据

```
scala> val userDF = sqlContext.read.json("/tmp/user.json")
userDF: org.apache.spark.sql.DataFrame = [age: string, gender: string, occupation: string, userID:
string, zipcode: string]
```

- 生成Json数据

```
scala> userDF.limit(5).write.mode("overwrite").json("/tmp/user2.json")
```

查看结果

```
scala> userDF.show(4)
```

```
+---+------+----------+------+-------+
|age|gender|occupation|userID|zipcode|
+---+------+----------+------+-------+
|  1|     F|        10|     1|  48067|
| 56|     M|        16|     2|  70072|
| 25|     M|        15|     3|  55117|
| 45|     M|         7|     4|  02460|
+---+------+----------+------+-------+
```

```
scala> userDF.limit(2).toJSON.foreach(println)
{"age":"1","gender":"F","occupation":"10","userID":"1","zipcode":"48067"}
{"age":"56","gender":"M","occupation":"16","userID":"2","zipcode":"70072"}
```

```
scala> userDF.printSchema
root
|-- age: string (nullable = true)
|-- gender: string (nullable = true)
|-- occupation: string (nullable = true)
|-- userID: string (nullable = true)
|-- zipcode: string (nullable = true)
```

查看结果

```
scala> userDF.show(4)
```

```
+---+------+----------+------+-------+
|age|gender|occupation|userID|zipcode|
+---+------+----------+------+-------+
|  1|     F|        10|     1|  48067|
| 56|     M|        16|     2|  70072|
| 25|     M|        15|     3|  55117|
| 45|     M|         7|     4|  02460|
+---+------+----------+------+-------+
```

```
scala> userDF.limit(2).toJSON.foreach(println)
{"age":"1","gender":"F","occupation":"10","userID":"1","zipcode":"48067"}
{"age":"56","gender":"M","occupation":"16","userID":"2","zipcode":"70072"}
```

```
scala> userDF.printSchema
root
|-- age: string (nullable = true)
|-- gender: string (nullable = true)
|-- occupation: string (nullable = true)
|-- userID: string (nullable = true)
|-- zipcode: string (nullable = true)
```

课后习题

- 1. 分析篮球运动员数据
- 数据集为NBA 1970 ~ 2016年球员的相关技术参数

Player	Pos	Age	Tm	G	GS	MP	FG	FGA	FG%
Kareem Abdul-Jabbar*	C	32	LAL	82		38.3	10.2	16.9	0.604
Tom Abernethy	PF	25	GSW	67		18.2	2.3	4.7	0.481
Alvan Adams	C	25	PHO	75		28.9	6.2	11.7	0.531
Tiny Archibald*	PG	31	BOS	80	80	35.8	4.8	9.9	0.482
Dennis Awtrey	C	31	CHI	26		21.5	1	2.3	0.45
Gus Bailey	SG	28	WSB	20		9	0.8	1.8	0.457
James Bailey	PF	22	SEA	67		10.8	1.8	4	0.45
Greg Ballard	SF	25	WSB	82		29.7	6.6	13.4	0.495
Mike Bantom	SF	28	IND	77		30.3	5	9.9	0.505
Marvin Barnes	PF	27	SDC	20		14.4	1.2	3	0.4
Rick Barry*	SF	35	HOU	72		25.2	4.5	10.7	0.422
Tim Bassett	PF	28	TOT	12		13.7	1	2.8	0.353
Billy Ray Bates	SG	23	POR	16		14.7	4.5	9.1	0.493
Ron Behagen	PF	29	WSB	6		10.7	1.5	3.8	0.391
Kent Benson	C	25	TOT	73		25.9	4.1	8.5	0.484
Del Beshore	PG	23	CHI	68		12.8	1.3	3.7	0.352
Henry Bibby	PG	30	PHI	82		24.8	3.1	7.6	0.401

课后习题

- 1. 分析篮球运动员数据

篮球数据缩写说明												
GP	出场次数		GS	首发次数			ORB	前场篮板		ORPG	场均前板
MP	总上场时间		MPG	场均上场时间			DRB	后场篮板		DRPG	场均后板
FG	投篮命中	FGA	投篮出手	FG%	投篮命中率		TRB	篮板球		RPG	场均篮板
3P	三分命中	3PA	三分出手	3P%	三分命中率		AST	助攻		APG	场均助攻
2P	两分命中	2PA	两分出手	2P%	两分命中率		STL	抢断		SPG	场均抢断
FT	罚球命中	FTA	罚球出手	FT%	罚球命中率		BLK	盖帽		BPG	场均盖帽
TOV	失误	PF	犯规	粗体	最高纪录		PTS	得分		PPG	场均得分

课后习题

- 评价球员水平的指标
  - ✓ **Z-score** $statZ_{(i,j)} = \frac{(stat_{(i,j)} - \mu_i)}{\sigma_i}$
  - ✓ μ表示平均值， σ 表示stat数据的标准差 $\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$。
  - ✓ 比如John Doe 在某年的每场比赛篮板球平均数目为7.1， 而当年所有球员μ=4.5， σ=1.3， 则该球员z-score得分为：

$$statZ_{(TRB,John\ Doe)} = \frac{(stat_{(TRB,John\ Doe)} - \mu)}{\sigma} = \frac{(7.1-4.5)}{1.3} = \frac{2.6}{1.3} = 2$$

- 用来分析的指标①FG% : 投篮命中率，② FT% :罚球命中率 ③ 3P: 三分球命中率
- ④ TRB: 篮板球 ⑤ age： 年龄

课后习题

- (1) 分析2016年 ①②③④ 属性 z-score 排名
- (2) 分析自1980年以来每个年龄段参赛的数目

```
+---+-----+
|age|count|
+---+-----+
| 18|   12|
| 19|   93|
| 20|  238|
| 21|  450|
| 22| 1137|
| 23| 1623|
| 24| 1626|
| 25| 1455|
| 26| 1356|
| 27| 1236|
| 28| 1077|
| 29|  980|
| 30|  883|
| 31|  745|
| 32|  619|
| 33|  487|
| 34|  362|
| 35|  251|
| 36|  166|
| 37|  111|
| 38|   73|
| 39|   40|
| 40|   15|
| 41|    4|
| 42|    3|
| 43|    1|
| 44|    1|
+---+-----+
```


联系我们
如何联系我们...

数据 引领变革

缔造新型 价值

联系我们

地址： 沈阳市和平区三好街84-8号易购大厦319A

电话： 024-88507865

邮箱： horizon@syhc.com.cn

公司网站： http://www.syhc.com.cn

THANKS