Spark SQL程序设计基础：第二部分

董西成

2017年04月

主要内容

1. Spark SQL程序设计基础
2. 常用DF/DS的operation介绍
3. Spark SQL中的SQL
4. Spark SQL调优
5. 应用案例：篮球运动员评估系统

主要内容

1 Spark SQL程序设计基础

2 常用DF/DS的operation介绍

3 Spark SQL中的SQL

4 Spark SQL调优

5 应用案例：篮球运动员评估系统

Spark SQL程序设计：编写流程

- 创建SparkSession对象
  - 封装了spark sql执行环境信息，是所有Spark SQL程序的唯一入口
- 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SparkSession

SparkSession中包含：

spark.sparkContext
spark.sqlContext

```
val spark = SparkSession.builder.
    master("local")
    .appName("spark session example")
    .getOrCreate()


//  注意，后面所有程序片段总的spark变量均值SparkSession
//  将RDD隐式转换为DataFrame
import spark.implicits._
```

步骤2：创建DataFrame或Dataset

提供了读写各种格式数据的API

Built-In

- { JSON }
- JDBC
- Parquet
- HIVE
- MySQL
- PostgreSQL
- HDFS
- amazon web services S3
- H2

External

- AVRO
- CSV
- dBase
- APACHE HBASE
- elasticsearch.
- cassandra
- amazon web services Amazon Redshift

and more…

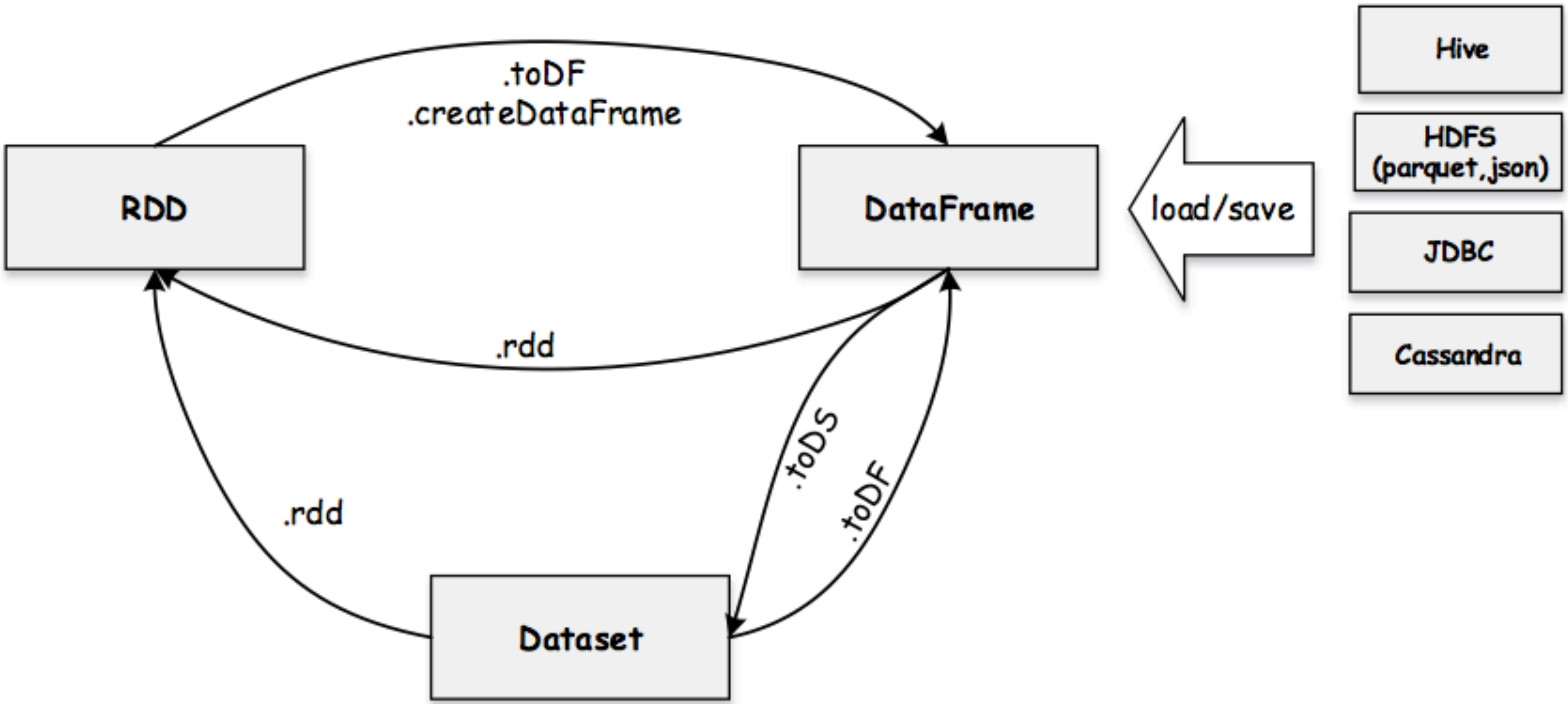
步骤3：在DataFrame或Dataset之上进行operation

Untyped transformations (DF -> DF)	Typed transformations (DS -> DS)	Actions (DF/DS -> console/output)
agg	map	collect
col	select	count
cube	filter	first
drop	flatMap	foreach
groupBy	mapPartitions	reduce
join	join	take
rollup	groupByKey	…
select	interset	
withColumn	repartition	
…	where	
	sort	
	…	
For DataFrame & Dataset	For Dataset	For DataFrame

DataFrame与Dataset

- **DataFrame = Dataset[Row]**
  - **Row表示一行数据，比如Row=[“a”, 12, 123]**
  - **RDD、DataFrame与Dataset之间可以相互转化**
- **DataFrame**
  - 内部数据无类型，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

RDD、DataFrame与Dataset的关系



val df = spark.read.parquet("...") // DataFrame
val ds = df.as[Person] // DataFrame → Dataset
val df2 = ds.toDF() / Dataset → DataFrame
val rdd1 = ds.rdd // Dataset → RDD
val rdd2 = df.rdd // DataFrame → RDD
val newDs = Seq(Person("Andy", 32)).toDS() // Seq → DS

主要内容

1. Spark SQL程序设计基础
2. 常用DF/DS的operation介绍
3. Spark SQL中的SQL
4. Spark SQL调优
5. 应用案例：篮球运动员评估系统

DataFrame Operations(>= 2.x)

Actions (DS/DF → console/output)	Typed transformations (DS → DS)	Untyped transformations (DF → DF)	Basic Function
collect	map	agg	cache
Count	select	col	persist
first	filter	cube	printSchema
head	flatMap	drop	toDF
show	mapPartitions	groupBy	unpersist
take	join	join	…
…	groupByKey	rollup	
	interset	select	
	repartition	withColumn	
	where	…	
	sort		
	…		

准备数据

1. Json数据

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

2. 读取Json数据

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

3. 生成Json数据

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

查看DF：show, toJSON & printSchema

```
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)
```

Action: collect, first, take & head

```
scala> userDF.collect
res35: Array[org.apache.spark.sql.Row] = Array([1,F,10,1,48067], [56,M,16,2,70072],
[25,M,15,3,55117], [45,M,7,4,02460], [25,M,20,5,55455])
```

```
scala> userDF.first
res36: org.apache.spark.sql.Row = [1,F,10,1,48067]
```

```
scala> userDF.take(2)
res37: Array[org.apache.spark.sql.Row] = Array([1,F,10,1,48067], [56,M,16,2,70072])
```

```
scala> userDF.head(2)
res38: Array[org.apache.spark.sql.Row] = Array([1,F,10,1,48067], [56,M,16,2,70072])
```

Transformation: select

```
scala> userDF.select("userID", "age").show
```

```
+------+---+
|userID|age|
+------+---+
|     1|  1|
|     2| 56|
+------+---+
```

```
scala> userDF.selectExpr("userID", "ceil(age/10) as newAge").show(2)
```

```
+------+------+
|userID|newAge|
+------+------+
|     1|   1.0|
|     2|   6.0|
+------+------+
```

```
scala> userDF.select(max('age), min('age), avg('age)).show
```

```
+--------+--------+-----------------+
|max(age)|min(age)|         avg(age)|
+--------+--------+-----------------+
|      56|       1|30.63923841059602|
+--------+--------+-----------------+
```

Transformation: filter

```
scala> userDF.filter(userDF("age") > 30).show(2)
```

```
+---+------+----------+------+-------+
|age|gender|occupation|userID|zipcode|
+---+------+----------+------+-------+
| 56|     M|        16|     2|  70072|
| 45|     M|         7|     4|  02460|
+---+------+----------+------+-------+
```

```
scala> userDF.filter("age > 30 and occupation = 10").show
```

```
+---+------+----------+------+-------+
|age|gender|occupation|userID|zipcode|
+---+------+----------+------+-------+
| 35|     M|        10|  4562|  94133|
| 56|     M|        10|  5223|  11361|
+---+------+----------+------+-------+
```

Transformation: 混用select & filter

```
scala> userDF.select("userID", "age").filter("age > 30").show(2)
```

```
+------+---+
|userID|age|
+------+---+
|     2| 56|
|     4| 45|
+------+---+
```

```
scala> userDF.filter("age > 30").select("userID", "age").show(2)
```

```
+------+---+
|userID|age|
+------+---+
|     2| 56|
|     4| 45|
+------+---+
```

Transformation: groupBy

```
scala> userDF.groupBy("age").count().show()
```

```
+---+-----+
|age|count|
+---+-----+
| 50|  496|
| 56|  380|
|  1|  222|
| 18| 1103|
| 25| 2096|
| 35| 1193|
| 45|  550|
+---+-----+
```

```
scala> userDF.groupBy("age").agg(count('gender),countDistinct('occupation)).show()
```

```
+---+-------------+--------------------------+
|age|count(gender)|COUNT(DISTINCT occupation)|
+---+-------------+--------------------------+
| 50|          496|                        20|
| 56|          380|                        20|
|  1|          222|                        13|
| 18|         1103|                        20|
| 25|         2096|                        20|
| 35|         1193|                        21|
| 45|          550|                        20|
+---+-------------+--------------------------+
```

Transformation: groupBy

```
scala> userDF.groupBy("age").agg("gender"->"count","occupation"->"count").show()
```

```
+---+-------------+-----------------+
|age|count(gender)|count(occupation)|
+---+-------------+-----------------+
| 50|          496|              496|
| 56|          380|              380|
|  1|          222|              222|
| 18|         1103|             1103|
| 25|         2096|             2096|
| 35|         1193|             1193|
| 45|          550|              550|
+---+-------------+-----------------+
```

可用的聚集函数:
`avg`, `max`, `min`, `sum`, `count`

Transformation: join

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

```
scala> ratingDataFrame.printSchema
root
 |-- userID: string (nullable = true)
 |-- movieID: string (nullable = true)
 |-- Rating: string (nullable = true)
 |-- Timestamp: string (nullable = true)
```

```
scala> val mergedDataFrame = ratingDataFrame.filter("movieID = 2116").
     |         join(userDataFrame, "userID").
     |         select("gender", "age").
     |         groupBy("gender", "age").
     |         count
mergedDataFrame: org.apache.spark.sql.DataFrame = [gender: string, age: string, count: bigint]
```

Transformation: 更多join

```
val mergedDataFrame2 = ratingDataFrame.filter("movieID = 2116").
   join(userDataFrame, userDataFrame("userID") === ratingDataFrame("userID"), "inner").
   select("gender", "age").
   groupBy("gender", "age").
   count
```

```
scala> mergedDataFrame2.show
+------+---+-----+
|gender|age|count|
+------+---+-----+
|     M| 25|  169|
|     F| 45|    3|
|     F| 50|    3|
|     M| 35|   66|
|     F| 56|    2|
|     F|  1|    4|
|     M| 45|   26|
|     M| 50|   22|
|     M| 56|    8|
|     F| 18|    9|
|     M|  1|   13|
|     F| 25|   28|
|     M| 18|   72|
|     F| 35|   13|
+------+---+-----+
```

Spark SQL支持的Join类型：inner, outer, left_outer, right_outer, semijoin

DataFrame -> 临时表

```
userDataFrame.createOrReplaceTempView("users")
val groupedUsers = spark.sql("select gender, age, count(*) as n from users group by gender, age")
groupedUsers.show()
```

```
+------+---+----+
|gender|age|   n|
+------+---+----+
|     M| 25|1538|
|     F| 45| 189|
|     F| 50| 146|
|     M| 35| 855|
|     F| 56| 102|
|     F|  1|  78|
|     M| 45| 361|
|     M| 50| 350|
|     M| 56| 278|
|     F| 18| 298|
|     M|  1| 144|
|     F| 25| 558|
|     F| 35| 338|
|     M| 18| 805|
+------+---+----+
```

Spark SQL中的表

- **Session范围内的临时表**
  - ✓ **df.createOrReplaceTempView("people")**
  - ✓ **只在session范围内有效，Session结束表自动删除**
- **全局范围内的临时表**
  - ✓ **df.createGlobalTempView("people")**
  - ✓ **所有session共享**

表被放到一个全局临时数据库中

```
df.createGlobalTempView("people")
spark.sql("SELECT * FROM global_temp.people").show()
spark.newSession().sql("SELECT * FROM global_temp.people").show()
```

- **将DataFrame或Dataset持久化到Hive中（需把hive配置放到环境中）**
  - ✓ **df.write.mode("overwrite").saveAsTable("database.tableName")**

DataFrame支持常用的RDD Operation

```
userDataFrame.map { u =>
  (u.getAs[String]("userID").toLong, u.getAs[String]("age").toInt + 1)
}.take(10).foreach(println)
```

```
(1,2)
(2,57)
(3,26)
(4,46)
(5,26)
(6,51)
(7,36)
(8,26)
(9,26)
(10,36)
```

主要内容

1 Spark SQL程序设计基础

2 常用DF/DS的operation介绍

3 Spark SQL中的SQL

4 Spark SQL调优

5 应用案例：篮球运动员评估系统

Spark SQL中的SQL：语法

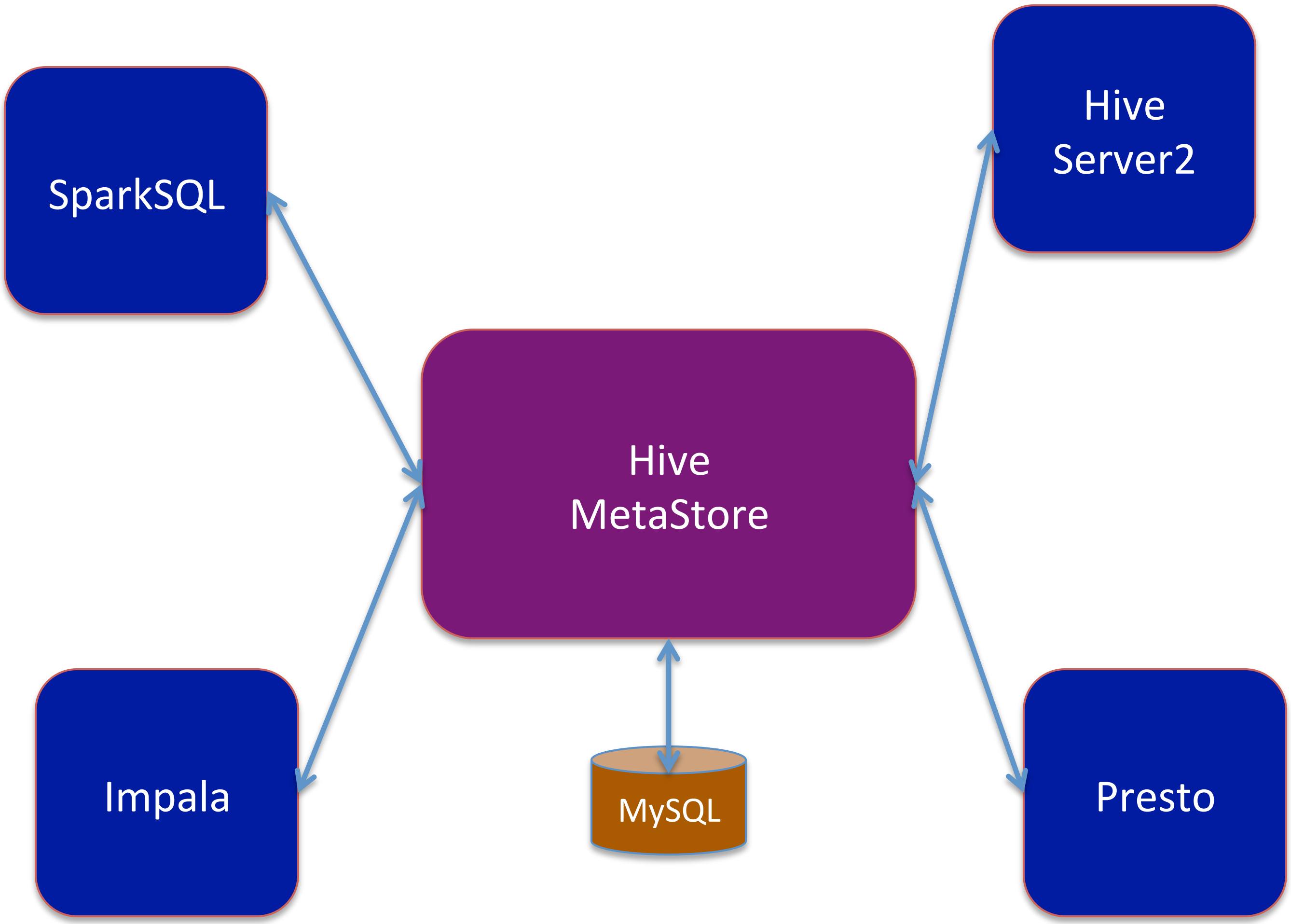
- **根据DataStax给出的Supported Syntax of Spark SQL，指出了Spark SQL支持的语法：**

  SELECT [DISTINCT] [column names]|[wildcard]
  FROM [kesypace name.]table name
  [JOIN clause table name ON join condition]
  [WHERE condition]
  [GROUP BY column name]
  [HAVING conditions]
  [ORDER BY column names [ASC | DSC]]

- 如果使用**join**进行查询，则支持的语法为**:**

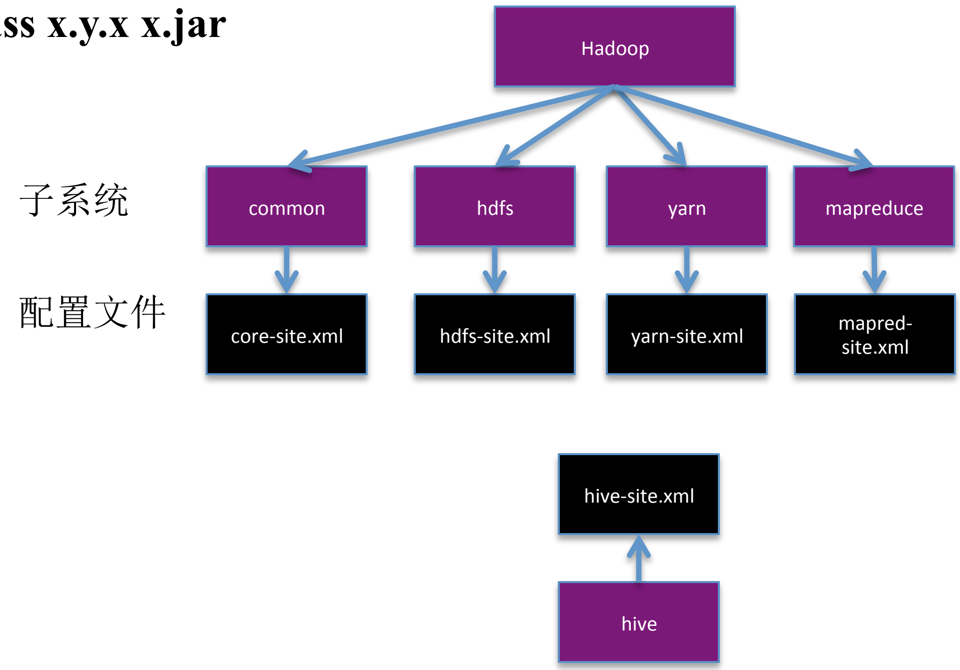
  SELECT statement
  FROM statement
  [JOIN | INNER JOIN | LEFT JOIN | LEFT SEMI JOIN | LEFT OUTER JOIN | RIGHT JOIN | RIGHT OUTER JOIN | FULL JOIN | FULL OUTER JOIN]
  ON join condition

Spark SQL中的SQL：部署架构



Spark SQL中的SQL：与Hive Metastore结合

- 将core-site.xml、hdfs-site.xml和hive-site.xml拷入spark安装目录下的conf/中
- Spark SQL与Hive Metastore结合：直接使用spark.sql(“SELECT … FROM table WHERE …”)
  - ✓ spark-shell --master local
  - ✓ spark-shell --master yarn-client
  - ✓ spark-submit --master yarn-cluster –class x.y.x x.jar
    - ✓ 需将hive-site.xml打包到x.jar中
  - ✓ 使用CLI
    - ✓ ./bin/spark-sql



Spark SQL中的SQL：JDBC/ODBC和CLI

- 启动thrift server

```
export HIVE_SERVER2_THRIFT_PORT=<listening-port>
export HIVE_SERVER2_THRIFT_BIND_HOST=<listening-host>
./sbin/start-thriftserver.sh \
  --master <master-uri> \
  ...
```

或者

```
./sbin/start-thriftserver.sh \
  --hiveconf hive.server2.thrift.port=<listening-port> \
  --hiveconf hive.server2.thrift.bind.host=<listening-host> \
  --master <master-uri>
  ...
```

- 使用beeline访问

```
beeline> !connect jdbc:hive2://<host>:<port>/<database>?
hive.server2.transport.mode=http;hive.server2.thrift.http.path=<http_endpoint>
```

主要内容

1. Spark SQL程序设计基础
2. 常用DF/DS的operation介绍
3. Spark SQL中的SQL
4. **Spark SQL调优**
5. 应用案例：篮球运动员评估系统

Spark SQL调优

➢ DataFrame缓存

- ✓ spark.sqlContext.cacheTable(“tableName”)
- ✓ dataFrame.cache()

Property Name	Default	Meaning
spark.sql.inMemoryColumnarStorage.compressed	true	当设置为true时，Spark SQL将为基于数据统计信息的每列自动选择一个压缩算法。
spark.sql.inMemoryColumnarStorage.batchSize	10000	柱状缓存的批数据大小。更大的批数据可以提高内存的利用率以及压缩效率，但有OOMs的风险

Spark SQL调优

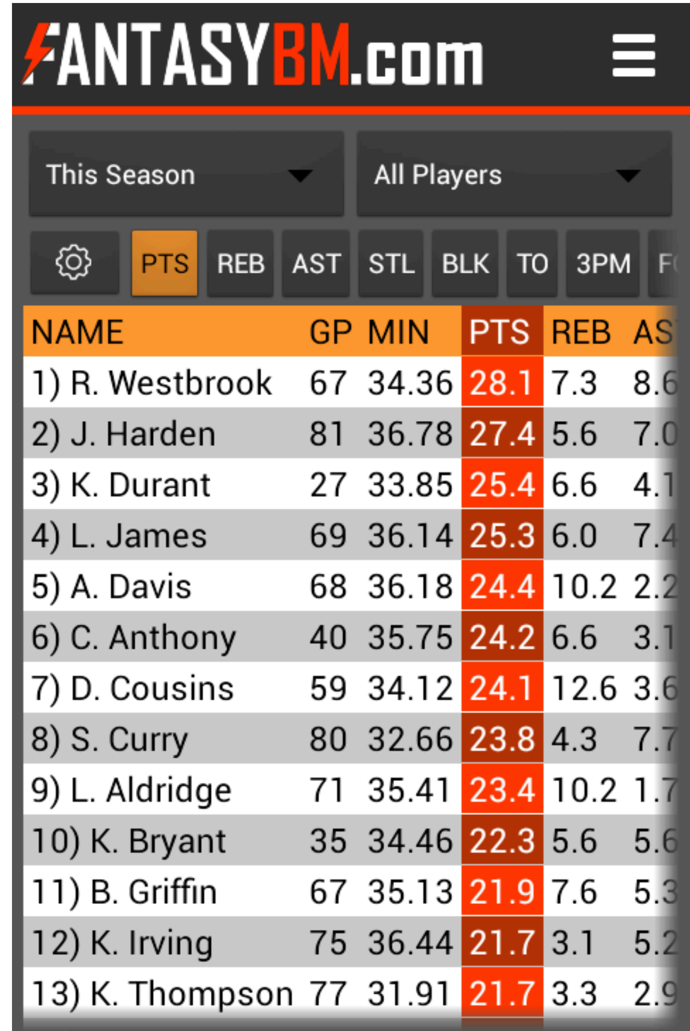
- 参数调优
  - ✓ **Reduce task数目：spark.sql.shuffle.partitions （默认是200）**
  - ✓ 读数据时每个**Partition**大小：**spark.sql.files.maxPartitionBytes （**默认**128MB）**
  - ✓ 小文件合并读：**spark.sql.files.openCostInBytes （**默认是**4194304 (4 MB) ）**
  - ✓ 广播小表大小：**spark.sql.autoBroadcastJoinThreshold （**默认是**10485760 (10 MB)）**

主要内容

1. Spark SQL程序设计基础
2. 常用DF/DS的operation介绍
3. Spark SQL中的SQL
4. Spark SQL调优
5. **应用案例：篮球运动员评估系统**

梦幻篮球：背景

- 将梦想和现实结合的游戏
- 自己组建球队，根据现实中的对手制定你自己的先发阵容，可以管理球员，包括任何阵容上的调整，包括裁员、签入、交易等等。



NAME	GP	MIN	PTS	REB	AS
1) R. Westbrook	67	34.36	28.1	7.3	8.6
2) J. Harden	81	36.78	27.4	5.6	7.0
3) K. Durant	27	33.85	25.4	6.6	4.1
4) L. James	69	36.14	25.3	6.0	7.4
5) A. Davis	68	36.18	24.4	10.2	2.2
6) C. Anthony	40	35.75	24.2	6.6	3.1
7) D. Cousins	59	34.12	24.1	12.6	3.6
8) S. Curry	80	32.66	23.8	4.3	7.7
9) L. Aldridge	71	35.41	23.4	10.2	1.7
10) K. Bryant	35	34.46	22.3	5.6	5.6
11) B. Griffin	67	35.13	21.9	7.6	5.3
12) K. Irving	75	36.44	21.7	3.1	5.2
13) K. Thompson	77	31.91	21.7	3.3	2.9

梦幻篮球：任务与数据

- 任务
  - ✓ 分析球员技能，为组建最强球队作数据支撑
- 数据
  - ✓ http://www.basketball-reference.com/leagues/NBA_2016_per_game.html
  - ✓ 1970-2016 NBA联赛球员数据：
    https://github.com/jordanvolz/BasketballStats/tree/master/data

梦幻篮球：任务与数据

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

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

梦幻篮球：数据分析

➢ 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
- ✓ 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：

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

➢ 选用指标

- ✓ **Standard-nine：Field Goal Percentage (FG%), Free Throw Percentage (FT%), Three Pointers Made (3P), Total Rebounds (TRB), Assists (AST), Steals (STL), Blocks (BLK), Turnovers (TOV), and Points (PTS)**

数据分析: (1) 数据预处理

- 添加日期：数据文件中没有时间信息

```
val DATA_PATH = "/user/cloudera/"
//process files so that each line includes the year
for (i <- 1980 to 2016) {
  println(i)
  val yearStats = sc.textFile(s"${DATA_PATH}/BasketballStats/leagues_NBA_
$i*").repartition(sc.defaultParallelism)
  yearStats.filter(x => x.contains(",")).map(x => (i, x)).saveAsTextFile(s"/user/cloudera/
BasketballStatsWithYear/$i/")
}
```

数据分析：(2) 计算z-score

➢ 步骤2.1：缓存数据以加快数据处理

```
val stats = sc.textFile(s"${DATA_PATH}/BasketballStatsWithYear/*/
*").repartition(sc.defaultParallelism)

//filter out junk rows, clean up data entry errors as well
val filteredStats = stats.filter(x => !x.contains("FG%")).filter(x => x.contains(","))
    .map(x => x.replace("*", "").replace(",,", ",0,"))
filteredStats.cache()
```

数据分析：(2) 计算z-score

- 步骤2.2：计算统计值
  - ✓ 均值、方差、最大值、最小值、出现次数

```
//process stats and save as map
val txtStat = Array("FG", "FGA", "FG%", "3P", "3PA", "3P%", "2P", "2PA", "2P%", "eFG%",
"FT",
  "FTA", "FT%", "ORB", "DRB", "TRB", "AST", "STL", "BLK", "TOV", "PF", "PTS")
val aggStats = processStats(filteredStats, txtStat).collectAsMap

//collect rdd into map and broadcast
val broadcastStats = sc.broadcast(aggStats)
```

数据分析：(2) 计算z-score

➢ 步骤2.3：计算z-score

```
val txtStatZ = Array("FG", "FT", "3P", "TRB", "AST", "STL", "BLK", "TOV", "PTS")
val zStats = processStats(filteredStats, txtStatZ, broadcastStats.value).collectAsMap

//collect rdd into map and broadcast
val zBroadcastStats = sc.broadcast(zStats)

//parse stats, now normalizing
val nStats = filteredStats.map(x => bbParse(x, broadcastStats.value, zBroadcastStats.value))
```

数据分析：(3)分析z-score

➢ 3.1 注册成临时表，便于分析

```
 //create schema for the data frame
val schemaN = StructType(
    StructField("name", StringType, true) ::
…
… :: Nil
)

//create data frame
val dfPlayersT = spark.createDataFrame(nPlayer,schemaN)

//save all stats as a temp table
dfPlayersT.createOrReplaceTempView("tPlayers")

//calculate experience levels
vval dfPlayers=spark.sql("select age-min_age as exp,tPlayers.* from tPlayers join (select
name,min(age)as min_age from tPlayers group by name) as t1 on tPlayers.name=t1.name order
by tPlayers.name, exp ")

//save as table
```

数据分析：(3)分析z-score

➢ 3.2 分析自1980年以来每个年龄段参赛数目

scala> dfPlayers.groupBy("age").count.sort("age").show(100)

```
+---+-----+
|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|
+---+-----+
```

数据分析：(3)分析z-score

➢ 3.3 查看2016年z-score排名

```
scala> spark.sql("Select name, zTot from Players where year=2016 order by zTot desc").take(10).foreach(println)

[Stephen Curry,19.766248304312754]
[Kevin Durant,15.323017389251323]
[Anthony Davis,13.186429940875069]
[Kawhi Leonard,13.18181904233336]
[James Harden,12.622408009920706]
[Russell Westbrook,12.26014043592826]
[Kyle Lowry,11.634357073733122]
[Paul Millsap,11.28903998833887]
[Chris Paul,10.843486407063033]
[Jimmy Butler,10.475908301410975]
```

数据分析：(3)分析z-score

➢ 3.4 查看2016年正则化z-score排名

```
scala> spark.sql("Select name, nTot from Players where year=2016 order by nTot desc").take(10).foreach(println)

[Stephen Curry,3.911865399387443]
[Kevin Durant,2.8729484855957916]
[Kawhi Leonard,2.7288580160780636]
[Anthony Davis,2.599364621997217]
[Russell Westbrook,2.4555072670169955]
[Paul Millsap,2.3037685595490816]
[LeBron James,2.1939606667616527]
[Kyle Lowry,2.151090172022115]
[Chris Paul,2.1331243771597586]
[James Harden,2.0788749389912686]
```

数据分析：(3)分析z-score

➢ 3.5 查看Curry所有数据

```
scala> spark.sql("Select * from Players where year=2016 and name='Stephen Curry'").collect.foreach(println)

[6,Stephen Curry,2016,27,PG,GSW,
42,42,33.9,10.0,19.5,0.51,4.9,10.8,0.451,5.1,8.7,0.583,0.635,5.3,5.9,0.911,0.8,4.6,5.4,6.6,2.1,0.1,3.4,2.0,30.1,
3.2822803060371077,3.4693236141930193,6.056007802613955,0.7550168635135219,2.695650422425284,
3.076598009775487,-0.6534242946356899,-2.7596520748161213,3.8444476552061824,19.7662483043127
47,0.8113439563367003,0.42770109585695865,1.0,0.15840862722238425,0.4898747874478281,0.7843137
254901962,-0.08887924083607253,-0.6708975521305531,1.0,3.911865399387443]
```

数据分析：(3)分析z-score

➢ 3.6 查看Curry 三分球得分排名

```
scala > spark.sql("select name, 3p, z3p from Players  where year=2016 order by z3p desc").take(10).foreach(println)

[Stephen Curry,4.9,6.056007802613957]
[Klay Thompson,3.2,3.6363158210435227]
[Damian Lillard,3.1,3.493980998598203]
[Paul George,2.9,3.2093113537075637]
[Kyle Lowry,2.7,2.9246417088169245]
[J.J. Redick,2.7,2.9246417088169245]
[James Harden,2.7,2.9246417088169245]
[Eric Gordon,2.5,2.6399720639262854]
[Wesley Matthews,2.5,2.6399720639262854]
[C.J. McCollum,2.5,2.6399720639262854]
```

总结

- Spark SQL程序设计思路与Spark类似
- Spark SQL支持各种数据源
  - json， parquet, jdbc, hbase ….
- DataFrame提供了丰富的operation函数
  - Transformation
  - Action
  - 转换为临时表，用SQL查询
  - RDD operation

hadoop123：董西成的微信公众号

专注于**Hadoop/spark**等大数据相关技术的分享

小象科技
ChinaHadoop.cn

联系我们：

- 新浪微博：ChinaHadoop
- 微信公号：ChinaHadoop

让你的数据产生价值！