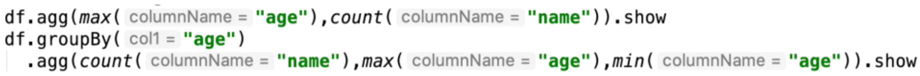
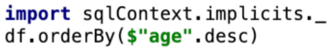
DataFrames 常用操作

1. Actions
   a) collect
   b) count
   c) describe
   d) first
   e) head
   f) show
   g) take
   h) taskAsList
2. Basic DataFrame Functions
   a) cache/persist
   b) columns
   c) dtypes
   d) printSchema
   e) registerTempTable
   f) unpersist
3. Language Integrated Queries
   a) agg



```
df.agg(max( columnName = "age"),count( columnName = "name")).show
df.groupBy( col1 = "age")
  .agg(count( columnName = "name"),max( columnName = "age"),min( columnName = "age")).show
```

   b) apply/col
   c) distinct
   d) drop
   e) dropDuplicates
   f) except
   g) filter
   h) groupBy
   i) intersect
   j) join
   k) limit
   l) orderBy/sort



```
import sqlContext.implicits._
df.orderBy($"age".desc)
```

   m) select
   n) unionAll
   o) where
   p) withColumn
   q) withColumnRenamed

4. Output Operations
    a) write
5. RDD Operations
    a) Rdd

DataFrame 函数

org.apache.spark.sql.functions 中提供了约两百多个函数，大部分函数与 Hive 中类似，除 UDF 函数，均可在 SparkSQL 中直接使用

如果想要用于 Dataframe 和 Dataset，可导入函数

```
import org.apache.spark.sql.functions._
```

其中，大部分支持 Column 的函数也支持 String 类型的列名，这些函数的返回类型基本都是 Column。

函数分类

聚合函数

集合函数

时间函数

数学函数

混杂 misc 函数

其他非聚合函数

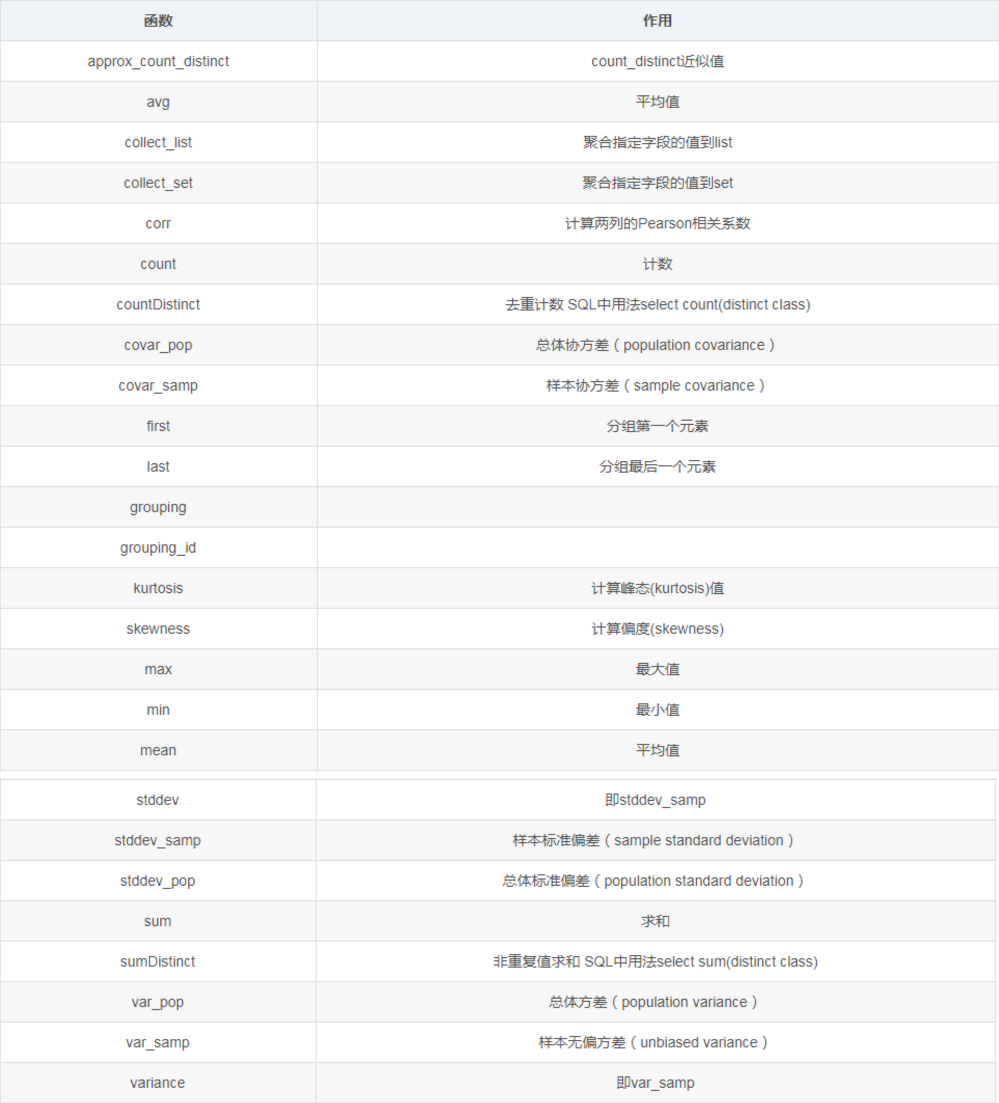
排序函数

字符串函数

UDF 函数

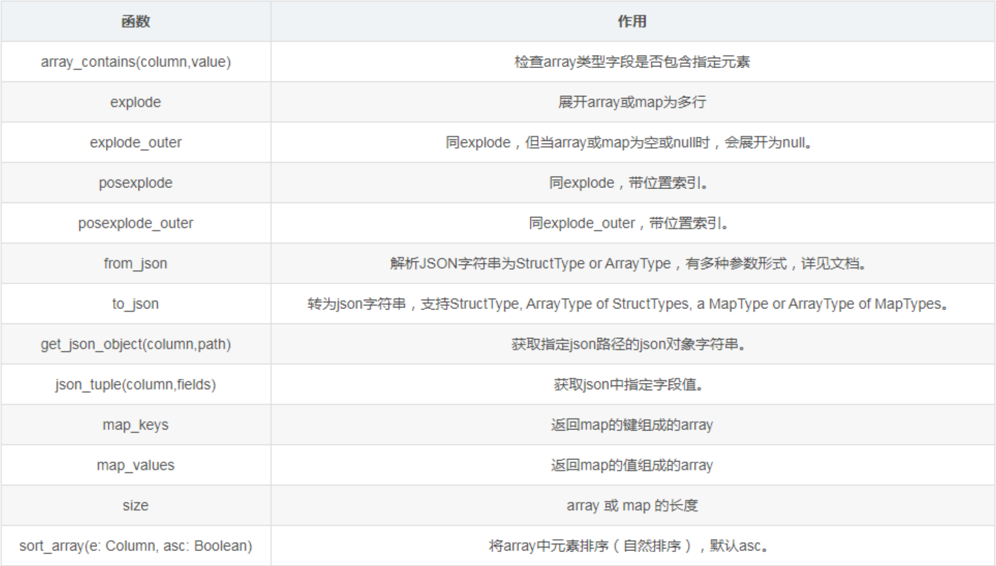
窗口函数

1、聚合函数



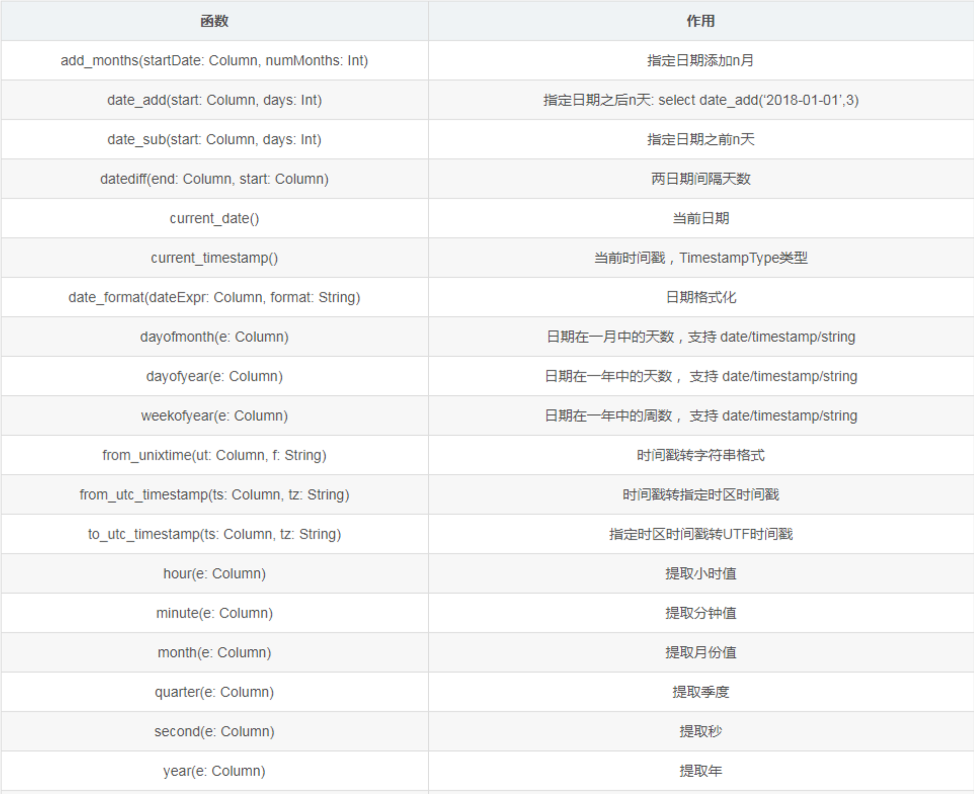
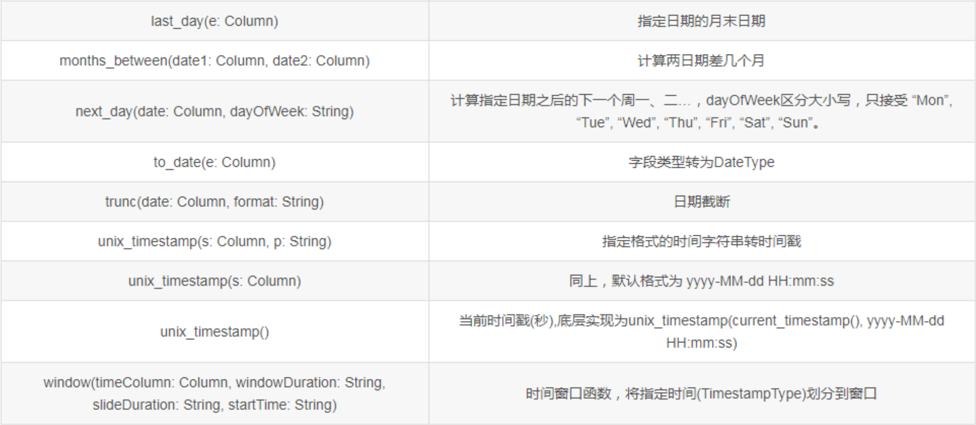
函数	作用
approx_count_distinct	count_distinct近似值
avg	平均值
collect_list	聚合指定字段的值到list
collect_set	聚合指定字段的值到set
corr	计算两列的Pearson相关系数
count	计数
countDistinct	去重计数 SQL中用法select count(distinct class)
covar_pop	总体协方差（population covariance）
covar_samp	样本协方差（sample covariance）
first	分组第一个元素
last	分组最后一个元素
grouping	
grouping_id	
kurtosis	计算峰态(kurtosis)值
skewness	计算偏度(skewness)
max	最大值
min	最小值
mean	平均值
stddev	即stddev_samp
stddev_samp	样本标准偏差（sample standard deviation）
stddev_pop	总体标准偏差（population standard deviation）
sum	求和
sumDistinct	非重复值求和 SQL中用法select sum(distinct class)
var_pop	总体方差（population variance）
var_samp	样本无偏方差（unbiased variance）
variance	即var_samp

2、集合函数



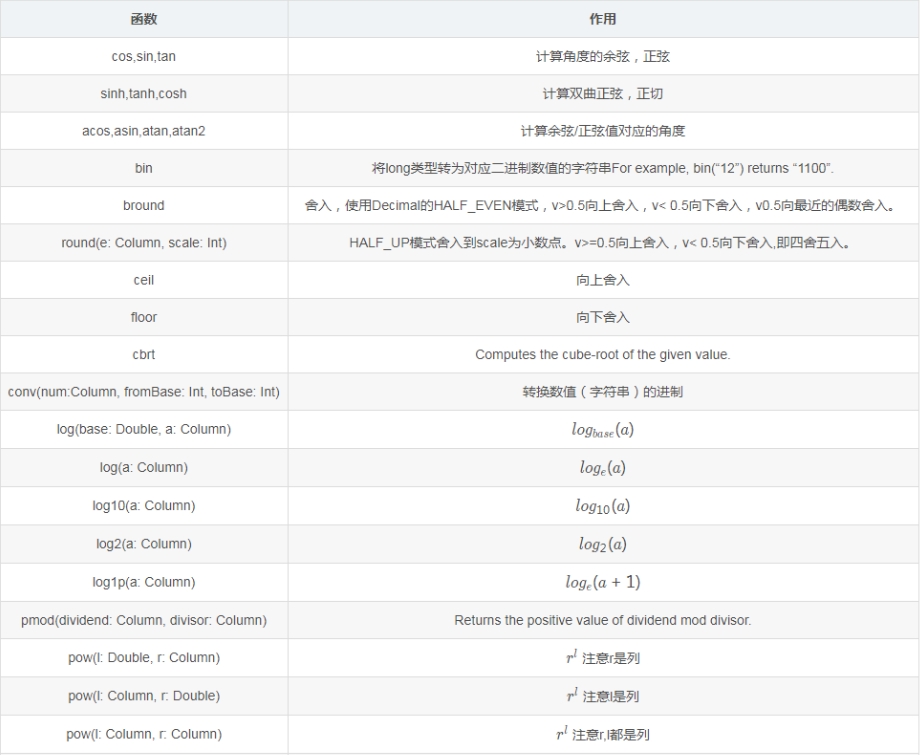
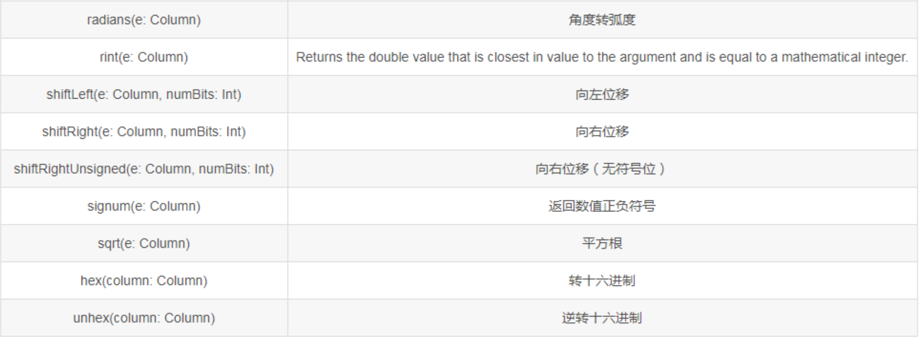
函数	作用
array_contains(column,value)	检查array类型字段是否包含指定元素
explode	展开array或map为多行
explode_outer	同explode，但当array或map为空或null时，会展开为null。
posexplode	同explode，带位置索引。
posexplode_outer	同explode_outer，带位置索引。
from_json	解析JSON字符串为StructType or ArrayType，有多种参数形式，详见文档。
to_json	转为json字符串，支持StructType, ArrayType of StructTypes, a MapType or ArrayType of MapTypes。
get_json_object(column,path)	获取指定json路径的json对象字符串。
json_tuple(column,fields)	获取json中指定字段值。
map_keys	返回map的键组成的array
map_values	返回map的值组成的array
size	array 或 map 的长度
sort_array(e: Column, asc: Boolean)	将array中元素排序（自然排序），默认asc。

3、时间函数

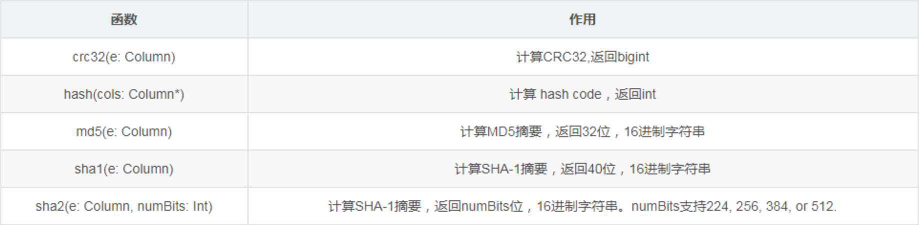
函数	作用
add_months(startDate: Column, numMonths: Int)	指定日期添加n月
date_add(start: Column, days: Int)	指定日期之后n天: select date_add(‘2018-01-01’,3)
date_sub(start: Column, days: Int)	指定日期之前n天
datediff(end: Column, start: Column)	两日期间隔天数
current_date()	当前日期
current_timestamp()	当前时间戳，TimestampType类型
date_format(dateExpr: Column, format: String)	日期格式化
dayofmonth(e: Column)	日期在一月中的天数，支持 date/timestamp/string
dayofyear(e: Column)	日期在一年中的天数， 支持 date/timestamp/string
weekofyear(e: Column)	日期在一年中的周数， 支持 date/timestamp/string
from_unixtime(ut: Column, f: String)	时间戳转字符串格式
from_utc_timestamp(ts: Column, tz: String)	时间戳转指定时区时间戳
to_utc_timestamp(ts: Column, tz: String)	指定时区时间戳转UTF时间戳
hour(e: Column)	提取小时值
minute(e: Column)	提取分钟值
month(e: Column)	提取月份值
quarter(e: Column)	提取季度
second(e: Column)	提取秒
year(e: Column)	提取年
last_day(e: Column)	指定日期的月末日期
months_between(date1: Column, date2: Column)	计算两日期差几个月
next_day(date: Column, dayOfWeek: String)	计算指定日期之后的下一个周一、二…，dayOfWeek区分大小写，只接受 “Mon”, “Tue”, “Wed”, “Thu”, “Fri”, “Sat”, “Sun”。
to_date(e: Column)	字段类型转为DateType
trunc(date: Column, format: String)	日期截断
unix_timestamp(s: Column, p: String)	指定格式的时间字符串转时间戳
unix_timestamp(s: Column)	同上，默认格式为 yyyy-MM-dd HH:mm:ss
unix_timestamp()	当前时间戳(秒),底层实现为unix_timestamp(current_timestamp(), yyyy-MM-dd HH:mm:ss)
window(timeColumn: Column, windowDuration: String, slideDuration: String, startTime: String)	时间窗口函数，将指定时间(TimestampType)划分到窗口

4、数学函数

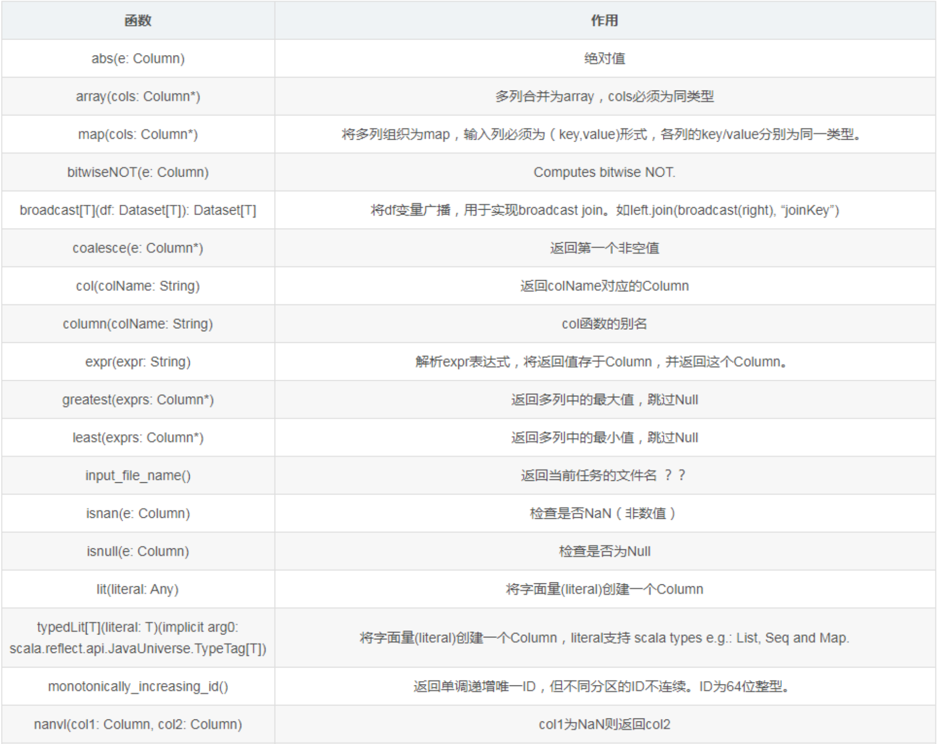
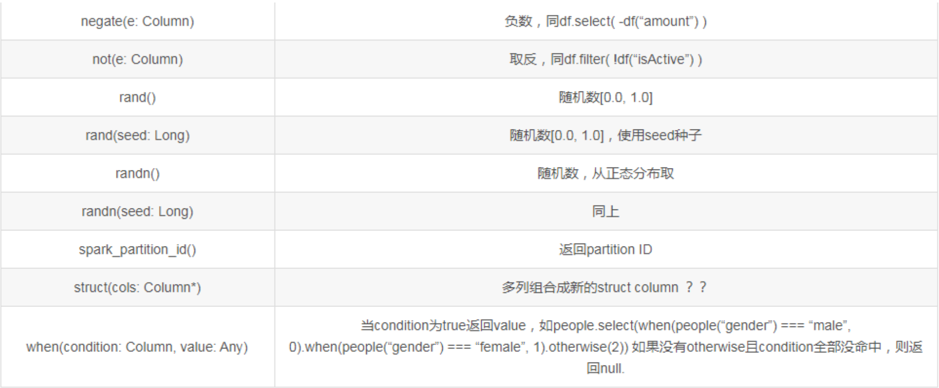
函数	作用
cos,sin,tan	计算角度的余弦，正弦
sinh,tanh,cosh	计算双曲正弦，正切
acos,asin,atan,atan2	计算余弦/正弦值对应的角度
bin	将long类型转为对应二进制数值的字符串For example, bin(“12”) returns “1100”.
bround	舍入，使用Decimal的HALF_EVEN模式，v>0.5向上舍入，v< 0.5向下舍入，v0.5向最近的偶数舍入。
round(e: Column, scale: Int)	HALF_UP模式舍入到scale为小数点。v>=0.5向上舍入，v< 0.5向下舍入,即四舍五入。
ceil	向上舍入
floor	向下舍入
cbrt	Computes the cube-root of the given value.
conv(num:Column, fromBase: Int, toBase: Int)	转换数值（字符串）的进制
log(base: Double, a: Column)	$log_{base}(a)$
log(a: Column)	$log_e(a)$
log10(a: Column)	$log_{10}(a)$
log2(a: Column)	$log_2(a)$
log1p(a: Column)	$log_e(a+1)$
pmod(dividend: Column, divisor: Column)	Returns the positive value of dividend mod divisor.
pow(l: Double, r: Column)	r^l 注意r是列
pow(l: Column, r: Double)	r^l 注意l是列
pow(l: Column, r: Column)	r^l 注意r,l都是列
radians(e: Column)	角度转弧度
rint(e: Column)	Returns the double value that is closest in value to the argument and is equal to a mathematical integer.
shiftLeft(e: Column, numBits: Int)	向左位移
shiftRight(e: Column, numBits: Int)	向右位移
shiftRightUnsigned(e: Column, numBits: Int)	向右位移（无符号位）
signum(e: Column)	返回数值正负符号
sqrt(e: Column)	平方根
hex(column: Column)	转十六进制
unhex(column: Column)	逆转十六进制

5、混杂misc函数



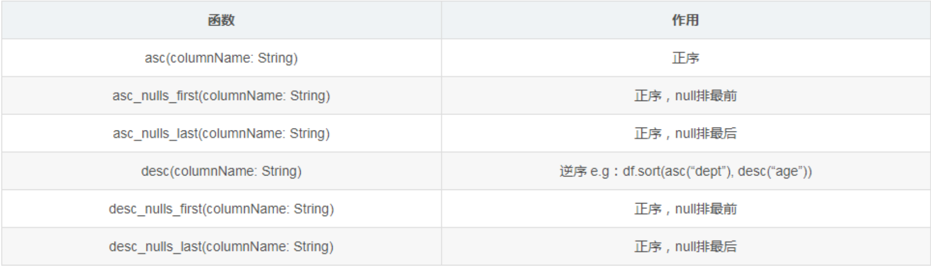
函数	作用
crc32(e: Column)	计算CRC32,返回bigint
hash(cols: Column*)	计算 hash code，返回int
md5(e: Column)	计算MD5摘要，返回32位，16进制字符串
sha1(e: Column)	计算SHA-1摘要，返回40位，16进制字符串
sha2(e: Column, numBits: Int)	计算SHA-1摘要，返回numBits位，16进制字符串。numBits支持224, 256, 384, or 512.

6、非聚合函数

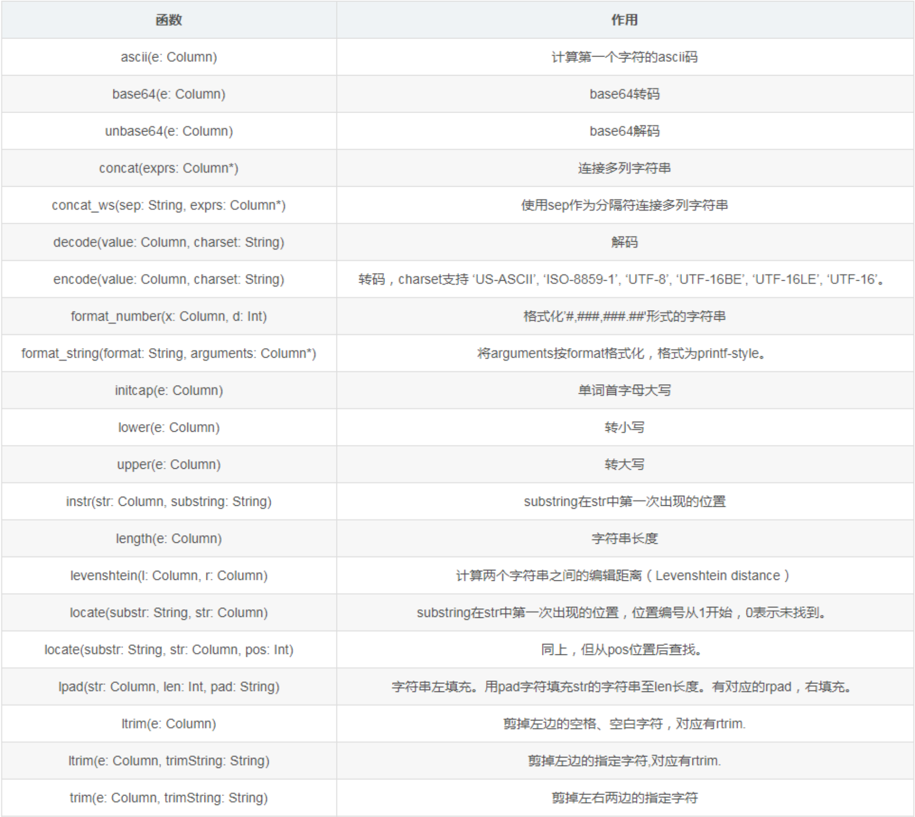
函数	作用
abs(e: Column)	绝对值
array(cols: Column*)	多列合并为array，cols必须为同类型
map(cols: Column*)	将多列组织为map，输入列必须为（key,value)形式，各列的key/value分别为同一类型。
bitwiseNOT(e: Column)	Computes bitwise NOT.
broadcast[T](df: Dataset[T]): Dataset[T]	将df变量广播，用于实现broadcast join。如left.join(broadcast(right), “joinKey”)
coalesce(e: Column*)	返回第一个非空值
col(colName: String)	返回colName对应的Column
column(colName: String)	col函数的别名
expr(expr: String)	解析expr表达式，将返回值存于Column，并返回这个Column。
greatest(exprs: Column*)	返回多列中的最大值，跳过Null
least(exprs: Column*)	返回多列中的最小值，跳过Null
input_file_name()	返回当前任务的文件名 ？？
isnan(e: Column)	检查是否NaN（非数值）
isnull(e: Column)	检查是否为Null
lit(literal: Any)	将字面量(literal)创建一个Column
typedLit[T](literal: T)(implicit arg0: scala.reflect.api.JavaUniverse.TypeTag[T])	将字面量(literal)创建一个Column，literal支持 scala types e.g.: List, Seq and Map.
monotonically_increasing_id()	返回单调递增唯一ID，但不同分区的ID不连续。ID为64位整型。
nanvl(col1: Column, col2: Column)	col1为NaN则返回col2
negate(e: Column)	负数，同df.select(-df(“amount”))
not(e: Column)	取反，同df.filter(!df(“isActive”))
rand()	随机数[0.0, 1.0]
rand(seed: Long)	随机数[0.0, 1.0]，使用seed种子
randn()	随机数，从正态分布取
randn(seed: Long)	同上
spark_partition_id()	返回partition ID
struct(cols: Column*)	多列组合成新的struct column ？？
when(condition: Column, value: Any)	当condition为true返回value，如people.select(when(people(“gender”) === “male”, 0).when(people(“gender”) === “female”, 1).otherwise(2)) 如果没有otherwise且condition全部没命中，则返回null.

7、排序函数

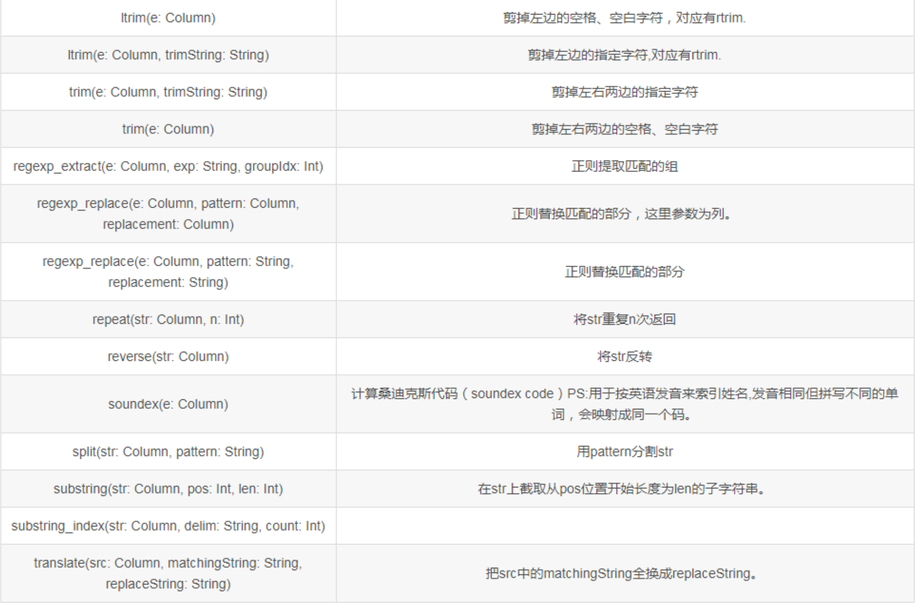


函数	作用
asc(columnName: String)	正序
asc_nulls_first(columnName: String)	正序，null排最前
asc_nulls_last(columnName: String)	正序，null排最后
desc(columnName: String)	逆序 e.g：df.sort(asc(“dept”), desc(“age”))
desc_nulls_first(columnName: String)	正序，null排最前
desc_nulls_last(columnName: String)	正序，null排最后

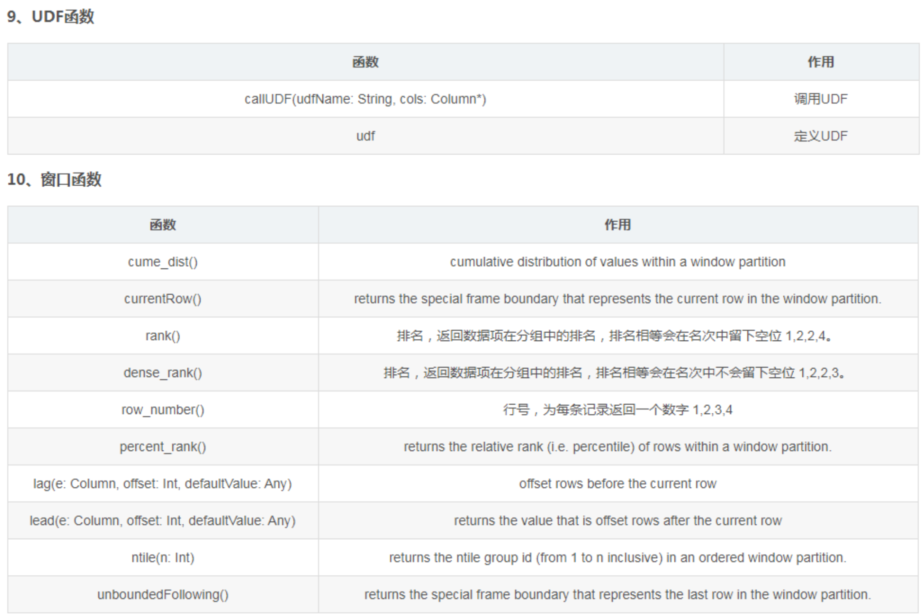
8、字符串函数



函数	作用
ascii(e: Column)	计算第一个字符的ascii码
base64(e: Column)	base64转码
unbase64(e: Column)	base64解码
concat(exprs: Column*)	连接多列字符串
concat_ws(sep: String, exprs: Column*)	使用sep作为分隔符连接多列字符串
decode(value: Column, charset: String)	解码
encode(value: Column, charset: String)	转码，charset支持 'US-ASCII', 'ISO-8859-1', 'UTF-8', 'UTF-16BE', 'UTF-16LE', 'UTF-16'。
format_number(x: Column, d: Int)	格式化'#,###,###.##'形式的字符串
format_string(format: String, arguments: Column*)	将arguments按format格式化，格式为printf-style。
initcap(e: Column)	单词首字母大写
lower(e: Column)	转小写
upper(e: Column)	转大写
instr(str: Column, substring: String)	substring在str中第一次出现的位置
length(e: Column)	字符串长度
levenshtein(l: Column, r: Column)	计算两个字符串之间的编辑距离（Levenshtein distance）
locate(substr: String, str: Column)	substring在str中第一次出现的位置，位置编号从1开始，0表示未找到。
locate(substr: String, str: Column, pos: Int)	同上，但从pos位置后查找。
lpad(str: Column, len: Int, pad: String)	字符串左填充。用pad字符填充str的字符串至len长度。有对应的rpad，右填充。
ltrim(e: Column)	剪掉左边的空格、空白字符，对应有rtrim.
ltrim(e: Column, trimString: String)	剪掉左边的指定字符,对应有rtrim.
trim(e: Column, trimString: String)	剪掉左右两边的指定字符



函数	作用
ltrim(e: Column)	剪掉左边的空格、空白字符，对应有rtrim.
ltrim(e: Column, trimString: String)	剪掉左边的指定字符,对应有rtrim.
trim(e: Column, trimString: String)	剪掉左右两边的指定字符
trim(e: Column)	剪掉左右两边的空格、空白字符
regexp_extract(e: Column, exp: String, groupIdx: Int)	正则提取匹配的组
regexp_replace(e: Column, pattern: Column, replacement: Column)	正则替换匹配的部分，这里参数为列。
regexp_replace(e: Column, pattern: String, replacement: String)	正则替换匹配的部分
repeat(str: Column, n: Int)	将str重复n次返回
reverse(str: Column)	将str反转
soundex(e: Column)	计算桑迪克斯代码（soundex code）PS:用于按英语发音来索引姓名,发音相同但拼写不同的单词，会映射成同一个码。
split(str: Column, pattern: String)	用pattern分割str
substring(str: Column, pos: Int, len: Int)	在str上截取从pos位置开始长度为len的子字符串。
substring_index(str: Column, delim: String, count: Int)	
translate(src: Column, matchingString: String, replaceString: String)	把src中的matchingString全换成replaceString。



9、UDF函数

函数	作用
callUDF(udfName: String, cols: Column*)	调用UDF
udf	定义UDF

10、窗口函数

函数	作用
cume_dist()	cumulative distribution of values within a window partition
currentRow()	returns the special frame boundary that represents the current row in the window partition.
rank()	排名，返回数据项在分组中的排名，排名相等会在名次中留下空位 1,2,2,4。
dense_rank()	排名，返回数据项在分组中的排名，排名相等会在名次中不会留下空位 1,2,2,3。
row_number()	行号，为每条记录返回一个数字 1,2,3,4
percent_rank()	returns the relative rank (i.e. percentile) of rows within a window partition.
lag(e: Column, offset: Int, defaultValue: Any)	offset rows before the current row
lead(e: Column, offset: Int, defaultValue: Any)	returns the value that is offset rows after the current row
ntile(n: Int)	returns the ntile group id (from 1 to n inclusive) in an ordered window partition.
unboundedFollowing()	returns the special frame boundary that represents the last row in the window partition.

- ## SQL 编程

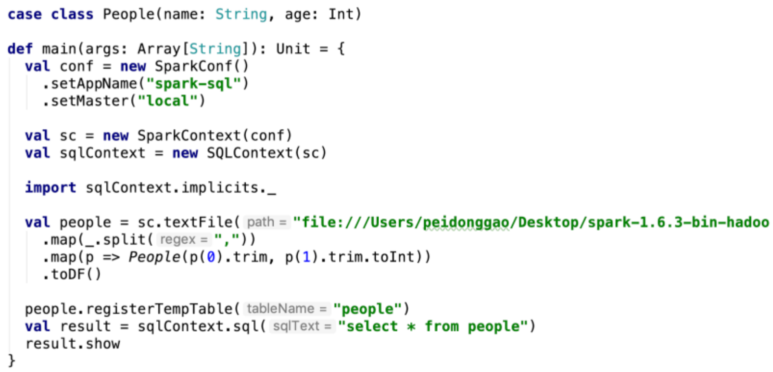
The sql function on a SQLContext enables applications to run SQL queries programmatically and returns the result as a DataFrame.

RDD 转 DataFrame

Spark SQL supports two different methods for converting existing RDDs into DataFrames. The first method uses reflection to infer the schema of an RDD that contains specific types of objects. This reflection based approach leads to more concise code and works well when you already know the schema while writing your Spark application.

The second method for creating DataFrames is through a programmatic interface that allows you to construct a schema and then apply it to an existing RDD. While this method is more verbose, it allows you to construct DataFrames when the columns and their types are not known until runtime.

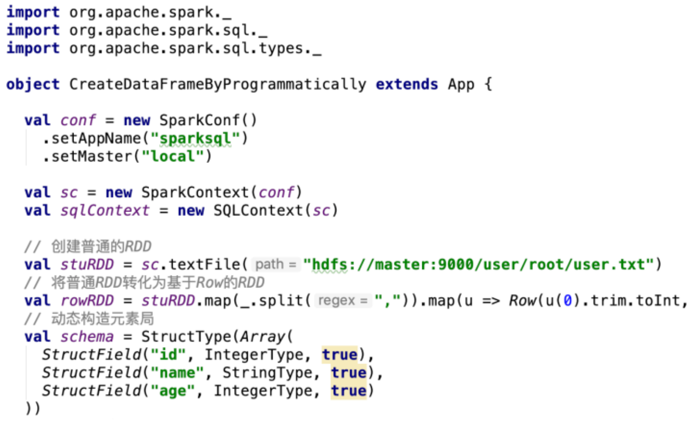
1. 反射机制转换



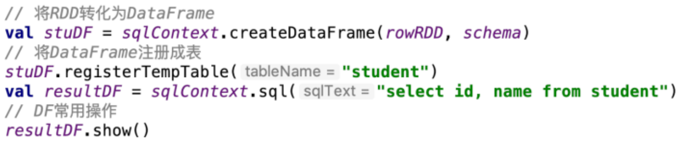
```
case class People(name: String, age: Int)

def main(args: Array[String]): Unit = {
  val conf = new SparkConf()
    .setAppName("spark-sql")
    .setMaster("local")

  val sc = new SparkContext(conf)
  val sqlContext = new SQLContext(sc)

  import sqlContext.implicits._

  val people = sc.textFile( path = "file:///Users/peidonggao/Desktop/spark-1.6.3-bin-hadoo
    .map(_.split( regex = ","))
    .map(p => People(p(0).trim, p(1).trim.toInt))
    .toDF()

  people.registerTempTable( tableName = "people")
  val result = sqlContext.sql( sqlText = "select * from people")
  result.show
}
```

注:case class 放在函数外部

2. 以编程方式动态指定元数据

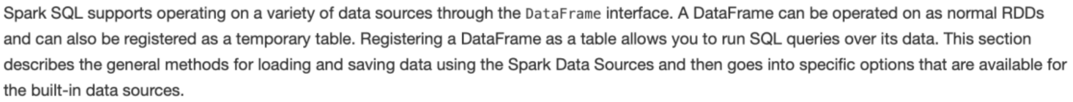


```
import org.apache.spark._
import org.apache.spark.sql._
import org.apache.spark.sql.types._

object CreateDataFrameByProgrammatically extends App {

  val conf = new SparkConf()
    .setAppName("sparksql")
    .setMaster("local")

  val sc = new SparkContext(conf)
  val sqlContext = new SQLContext(sc)

  // 创建普通的RDD
  val stuRDD = sc.textFile( path = "hdfs://master:9000/user/root/user.txt")
  // 将普通RDD转化为基于Row的RDD
  val rowRDD = stuRDD.map(_.split( regex = ",")).map(u => Row(u(0).trim.toInt,
  // 动态构造元素局
  val schema = StructType(Array(
    StructField("id", IntegerType, true),
    StructField("name", StringType, true),
    StructField("age", IntegerType, true)
  ))
```



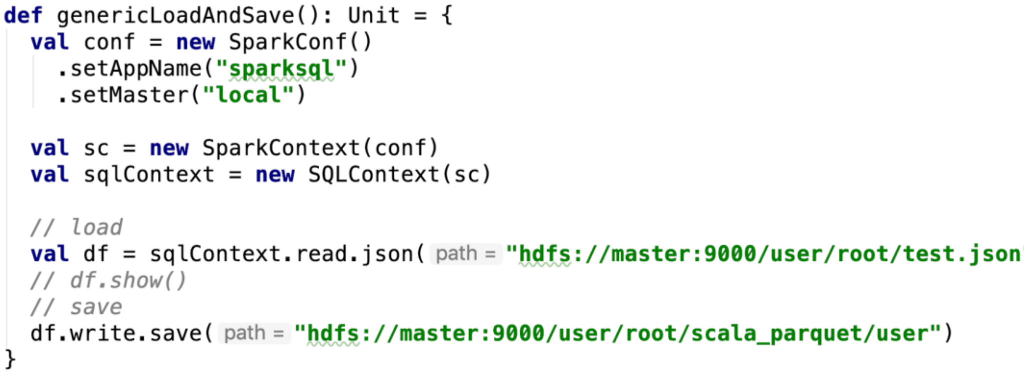
```
// 将RDD转化为DataFrame
val stuDF = sqlContext.createDataFrame(rowRDD, schema)
// 将DataFrame注册成表
stuDF.registerTempTable( tableName = "student")
val resultDF = sqlContext.sql( sqlText = "select id, name from student")
// DF常用操作
resultDF.show()
```

数据源



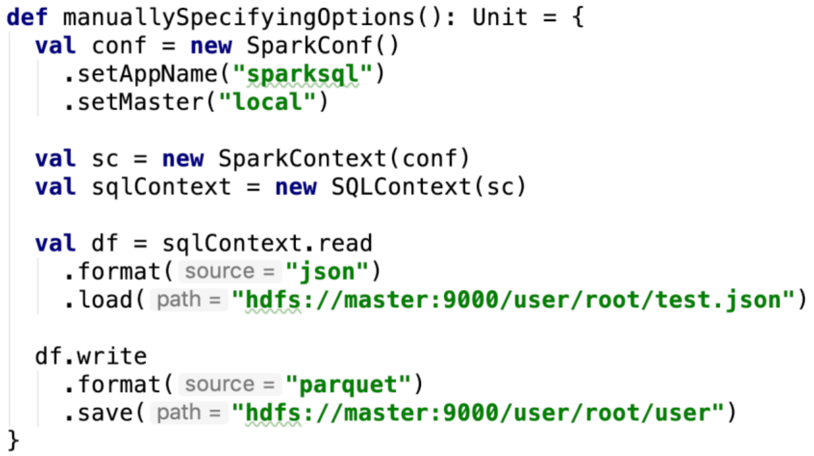
Spark SQL supports operating on a variety of data sources through the `DataFrame` interface. A DataFrame can be operated on as normal RDDs and can also be registered as a temporary table. Registering a DataFrame as a table allows you to run SQL queries over its data. This section describes the general methods for loading and saving data using the Spark Data Sources and then goes into specific options that are available for the built-in data sources.

通用 Load/Save 函数



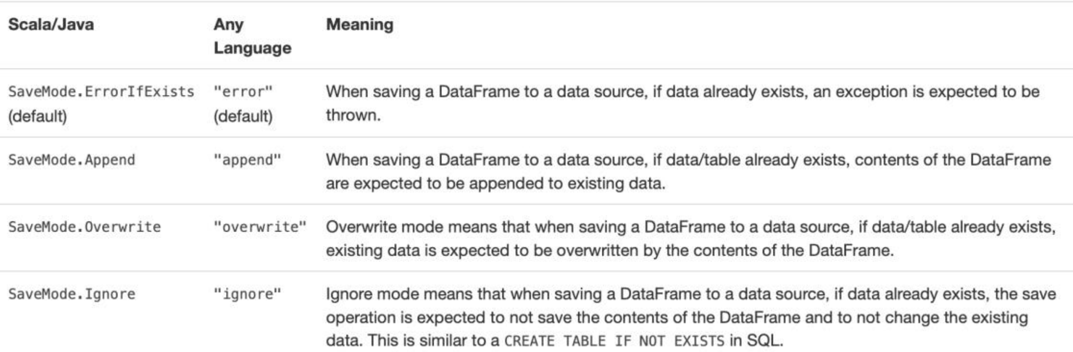
```
def genericLoadAndSave(): Unit = {
  val conf = new SparkConf()
    .setAppName("sparksql")
    .setMaster("local")

  val sc = new SparkContext(conf)
  val sqlContext = new SQLContext(sc)

  // load
  val df = sqlContext.read.json( path = "hdfs://master:9000/user/root/test.json
  // df.show()
  // save
  df.write.save( path = "hdfs://master:9000/user/root/scala_parquet/user")
}
```

手动指定选项

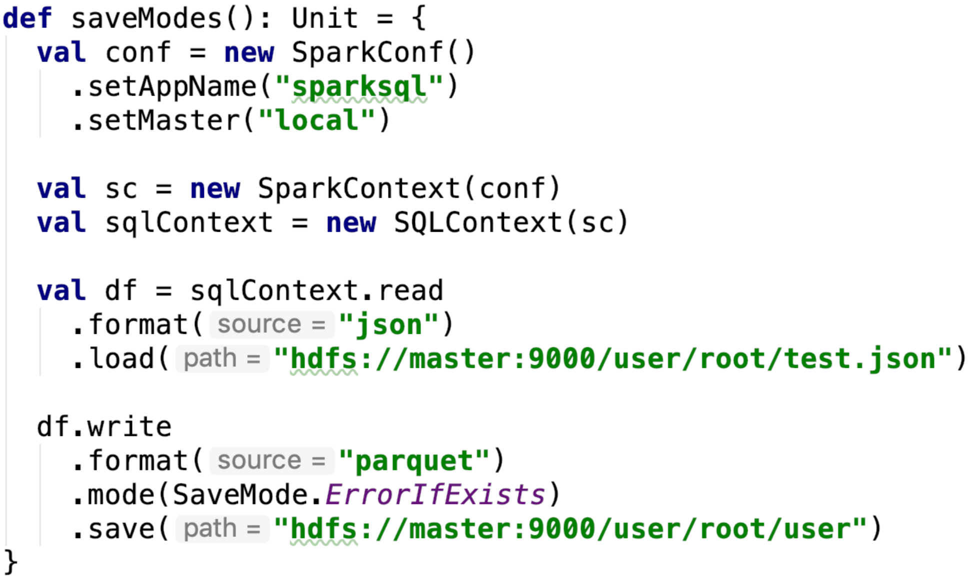


```
def manuallySpecifyingOptions(): Unit = {
  val conf = new SparkConf()
    .setAppName("sparksql")
    .setMaster("local")

  val sc = new SparkContext(conf)
  val sqlContext = new SQLContext(sc)

  val df = sqlContext.read
    .format( source = "json")
    .load( path = "hdfs://master:9000/user/root/test.json")

  df.write
    .format( source = "parquet")
    .save( path = "hdfs://master:9000/user/root/user")
}
```

Save Modes



Scala/Java	Any Language	Meaning
`SaveMode.ErrorIfExists` (default)	`"error"` (default)	When saving a DataFrame to a data source, if data already exists, an exception is expected to be thrown.
`SaveMode.Append`	`"append"`	When saving a DataFrame to a data source, if data/table already exists, contents of the DataFrame are expected to be appended to existing data.
`SaveMode.Overwrite`	`"overwrite"`	Overwrite mode means that when saving a DataFrame to a data source, if data/table already exists, existing data is expected to be overwritten by the contents of the DataFrame.
`SaveMode.Ignore`	`"ignore"`	Ignore mode means that when saving a DataFrame to a data source, if data already exists, the save operation is expected to not save the contents of the DataFrame and to not change the existing data. This is similar to a `CREATE TABLE IF NOT EXISTS` in SQL.

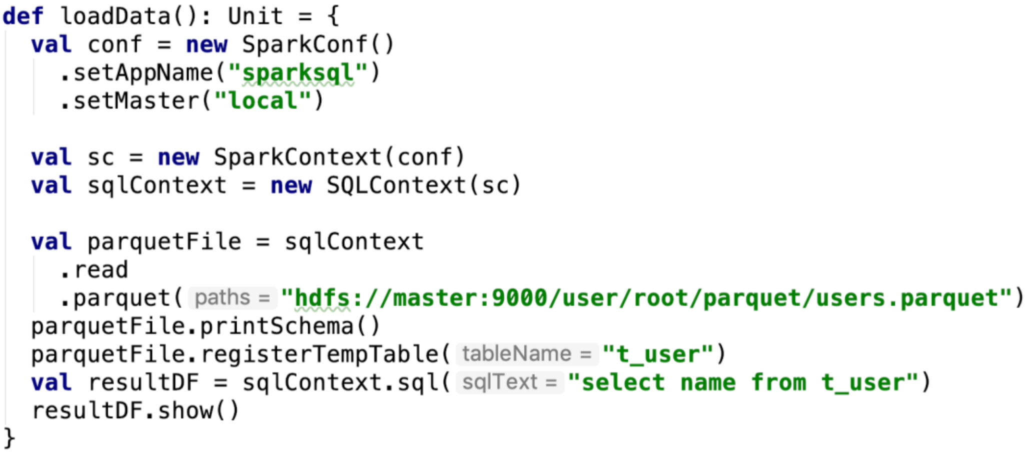


```scala
def saveModes(): Unit = {
  val conf = new SparkConf()
    .setAppName("sparksql")
    .setMaster("local")

  val sc = new SparkContext(conf)
  val sqlContext = new SQLContext(sc)

  val df = sqlContext.read
    .format( source = "json")
    .load( path = "hdfs://master:9000/user/root/test.json")

  df.write
    .format( source = "parquet")
    .mode(SaveMode.ErrorIfExists)
    .save( path = "hdfs://master:9000/user/root/user")
}
```

Parquet File Programmatically

https://en.wikipedia.org/wiki/Column-oriented_DBMS#Column-oriented_systems

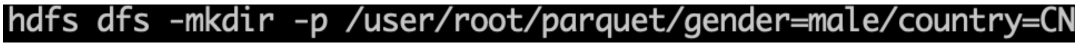
Parquet 数据加载



```scala
def loadData(): Unit = {
  val conf = new SparkConf()
    .setAppName("sparksql")
    .setMaster("local")

  val sc = new SparkContext(conf)
  val sqlContext = new SQLContext(sc)

  val parquetFile = sqlContext
    .read
    .parquet( paths = "hdfs://master:9000/user/root/parquet/users.parquet")
  parquetFile.printSchema()
  parquetFile.registerTempTable( tableName = "t_user")
  val resultDF = sqlContext.sql( sqlText = "select name from t_user")
  resultDF.show()
}
```

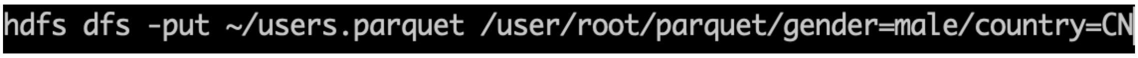
Parquet 分区推断

创建分区目录



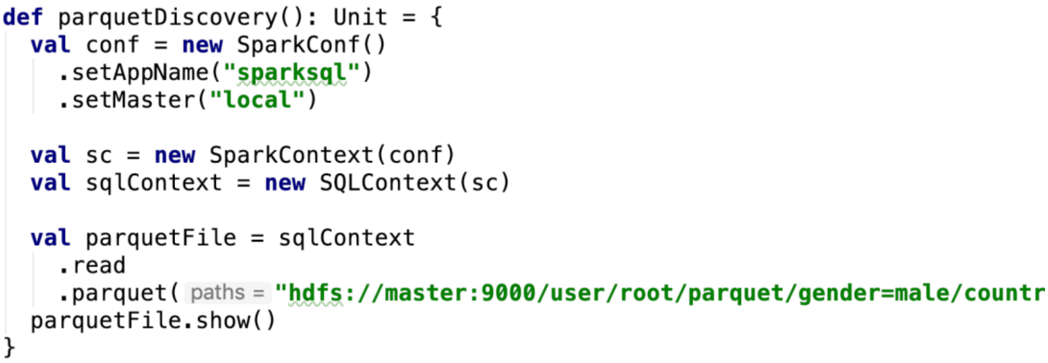
```
hdfs dfs -mkdir -p /user/root/parquet/gender=male/country=CN
```

上传数据到该目录



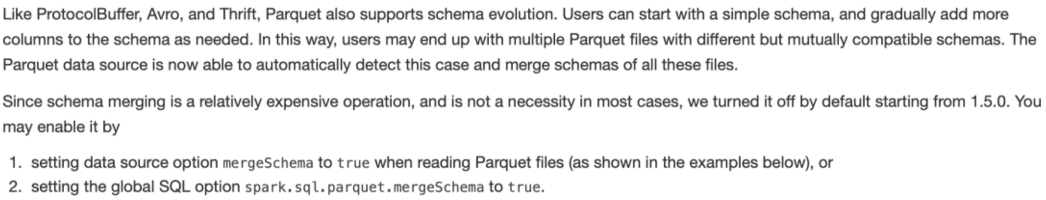
```
hdfs dfs -put ~/users.parquet /user/root/parquet/gender=male/country=CN
```

测试



```
def parquetDiscovery(): Unit = {
  val conf = new SparkConf()
    .setAppName("sparksql")
    .setMaster("local")

  val sc = new SparkContext(conf)
  val sqlContext = new SQLContext(sc)

  val parquetFile = sqlContext
    .read
    .parquet( paths = "hdfs://master:9000/user/root/parquet/gender=male/countr
  parquetFile.show()
}
```

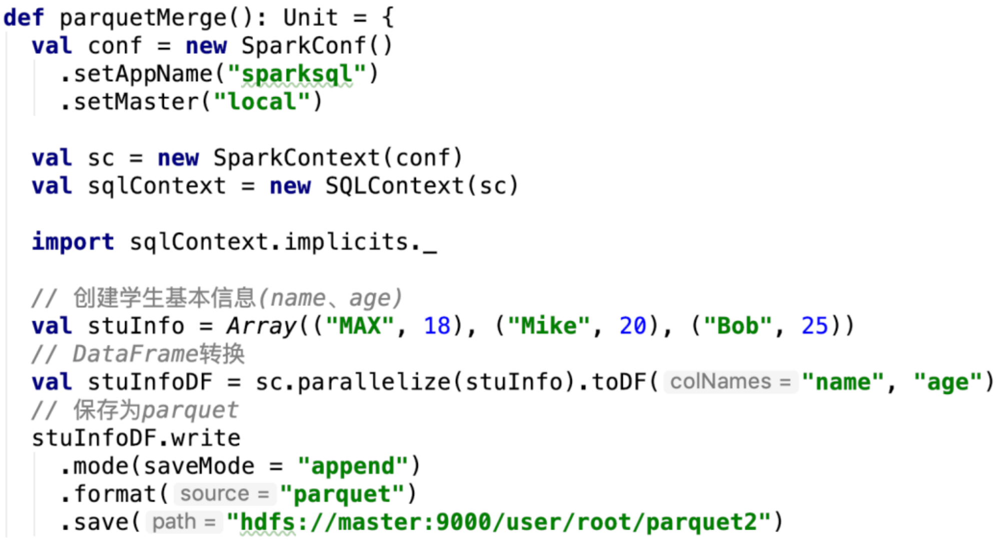
Parquet 合并元数据



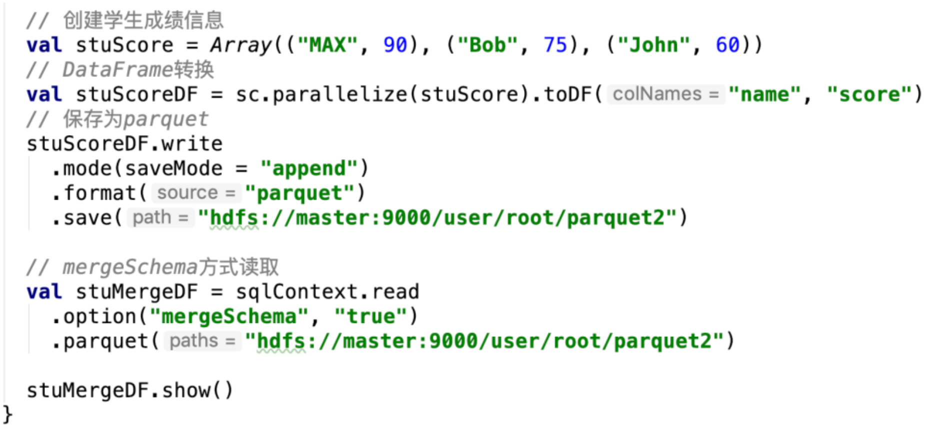
Like ProtocolBuffer, Avro, and Thrift, Parquet also supports schema evolution. Users can start with a simple schema, and gradually add more columns to the schema as needed. In this way, users may end up with multiple Parquet files with different but mutually compatible schemas. The Parquet data source is now able to automatically detect this case and merge schemas of all these files.

Since schema merging is a relatively expensive operation, and is not a necessity in most cases, we turned it off by default starting from 1.5.0. You may enable it by

1. setting data source option `mergeSchema` to `true` when reading Parquet files (as shown in the examples below), or
2. setting the global SQL option `spark.sql.parquet.mergeSchema` to `true`.

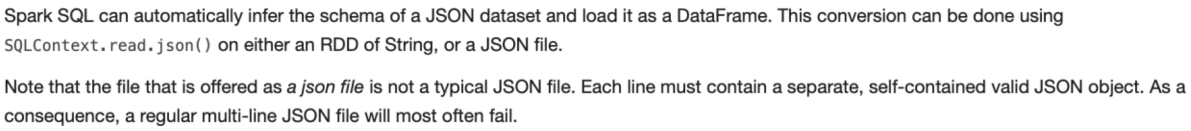


```
def parquetMerge(): Unit = {
  val conf = new SparkConf()
    .setAppName("sparksql")
    .setMaster("local")

  val sc = new SparkContext(conf)
  val sqlContext = new SQLContext(sc)

  import sqlContext.implicits._

  // 创建学生基本信息(name、age)
  val stuInfo = Array(("MAX", 18), ("Mike", 20), ("Bob", 25))
  // DataFrame转换
  val stuInfoDF = sc.parallelize(stuInfo).toDF( colNames = "name", "age")
  // 保存为parquet
  stuInfoDF.write
    .mode(saveMode = "append")
    .format( source = "parquet")
    .save( path = "hdfs://master:9000/user/root/parquet2")
```



```
  // 创建学生成绩信息
  val stuScore = Array(("MAX", 90), ("Bob", 75), ("John", 60))
  // DataFrame转换
  val stuScoreDF = sc.parallelize(stuScore).toDF( colNames = "name", "score")
  // 保存为parquet
  stuScoreDF.write
    .mode(saveMode = "append")
    .format( source = "parquet")
    .save( path = "hdfs://master:9000/user/root/parquet2")

  // mergeSchema方式读取
  val stuMergeDF = sqlContext.read
    .option("mergeSchema", "true")
    .parquet( paths = "hdfs://master:9000/user/root/parquet2")

  stuMergeDF.show()
}
```

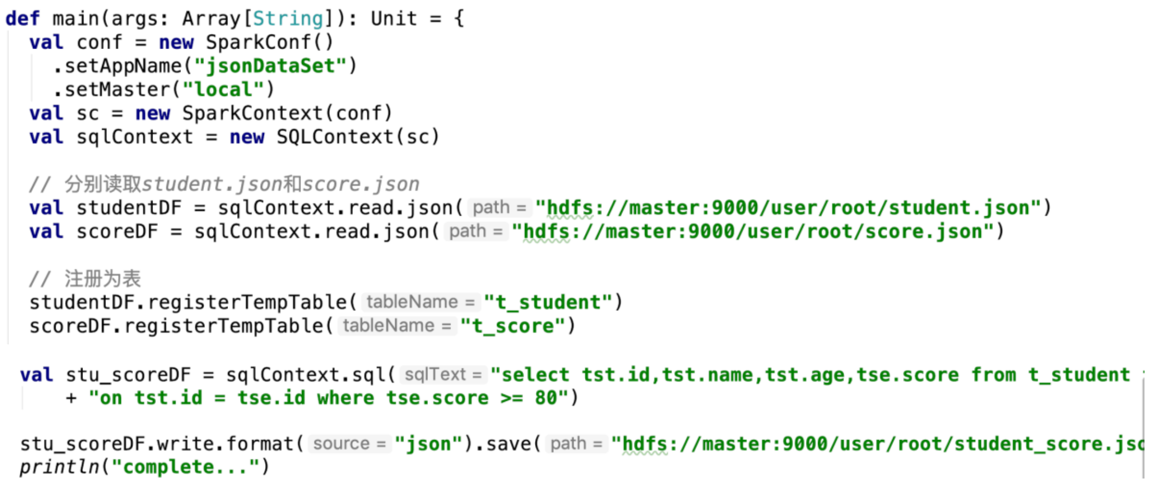
JSON 数据源



Spark SQL can automatically infer the schema of a JSON dataset and load it as a DataFrame. This conversion can be done using `SQLContext.read.json()` on either an RDD of String, or a JSON file.

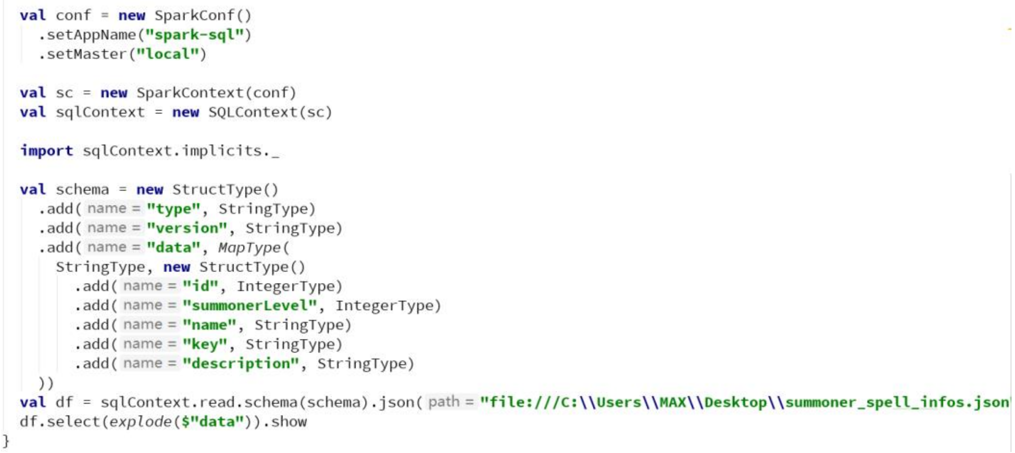
Note that the file that is offered as *a json file* is not a typical JSON file. Each line must contain a separate, self-contained valid JSON object. As a consequence, a regular multi-line JSON file will most often fail.

查询成绩为 80 分以上的学生的基本信息与成绩信息



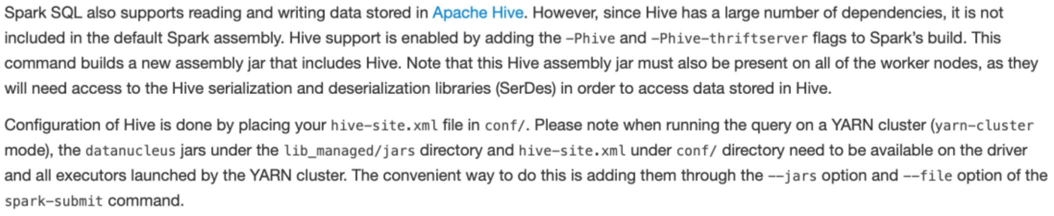
```
def main(args: Array[String]): Unit = {
  val conf = new SparkConf()
    .setAppName("jsonDataSet")
    .setMaster("local")
  val sc = new SparkContext(conf)
  val sqlContext = new SQLContext(sc)

  // 分别读取student.json和score.json
  val studentDF = sqlContext.read.json( path = "hdfs://master:9000/user/root/student.json")
  val scoreDF = sqlContext.read.json( path = "hdfs://master:9000/user/root/score.json")

  // 注册为表
  studentDF.registerTempTable( tableName = "t_student")
  scoreDF.registerTempTable( tableName = "t_score")

 val stu_scoreDF = sqlContext.sql( sqlText = "select tst.id,tst.name,tst.age,tse.score from t_student
     + "on tst.id = tse.id where tse.score >= 80")

 stu_scoreDF.write.format( source = "json").save( path = "hdfs://master:9000/user/root/student_score.jsc
 println("complete...")
```

复杂 JSON 格式处理



```
  val conf = new SparkConf()
    .setAppName("spark-sql")
    .setMaster("local")

  val sc = new SparkContext(conf)
  val sqlContext = new SQLContext(sc)

  import sqlContext.implicits._

  val schema = new StructType()
    .add( name = "type", StringType)
    .add( name = "version", StringType)
    .add( name = "data", MapType(
      StringType, new StructType()
        .add( name = "id", IntegerType)
        .add( name = "summonerLevel", IntegerType)
        .add( name = "name", StringType)
        .add( name = "key", StringType)
        .add( name = "description", StringType)
    ))
  val df = sqlContext.read.schema(schema).json( path = "file:///C:\\Users\\MAX\\Desktop\\summoner_spell_infos.json
  df.select(explode($"data")).show
}
```

Hive 数据源



Spark SQL also supports reading and writing data stored in Apache Hive. However, since Hive has a large number of dependencies, it is not included in the default Spark assembly. Hive support is enabled by adding the `-Phive` and `-Phive-thriftserver` flags to Spark's build. This command builds a new assembly jar that includes Hive. Note that this Hive assembly jar must also be present on all of the worker nodes, as they will need access to the Hive serialization and deserialization libraries (SerDes) in order to access data stored in Hive.

Configuration of Hive is done by placing your `hive-site.xml` file in `conf/`. Please note when running the query on a YARN cluster (`yarn-cluster` mode), the `datanucleus` jars under the `lib_managed/jars` directory and `hive-site.xml` under `conf/` directory need to be available on the driver and all executors launched by the YARN cluster. The convenient way to do this is adding them through the `--jars` option and `--file` option of the `spark-submit` command.

1. 配置 Hive metaStore Service

```
<property>
        <name>hive.metastore.uris</name>
        <value>thrift://master:9083</value>
</property>
```

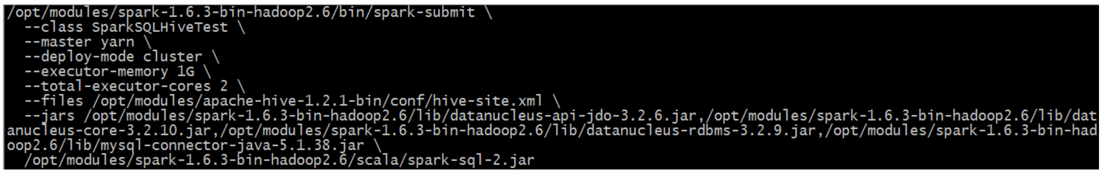
2. 开启 Hive metaStore Service

```
 bin/hive  - service metastore
```

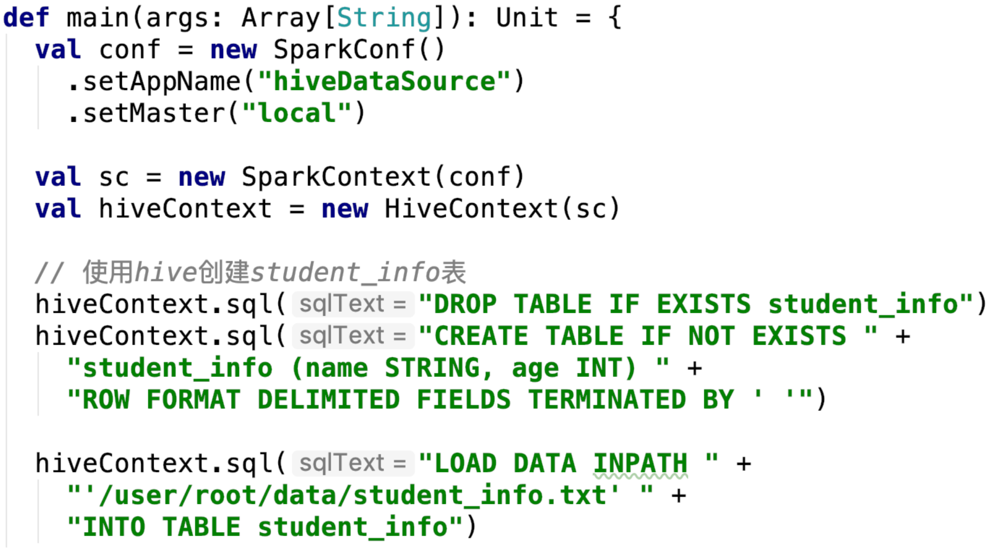
3. 拷贝 Hive conf/hive-site.xml 到 Spark conf 目录下

4. 拷贝 mysql-connector-java-5.1.27-bin.jar 到 Spark lib 目录下

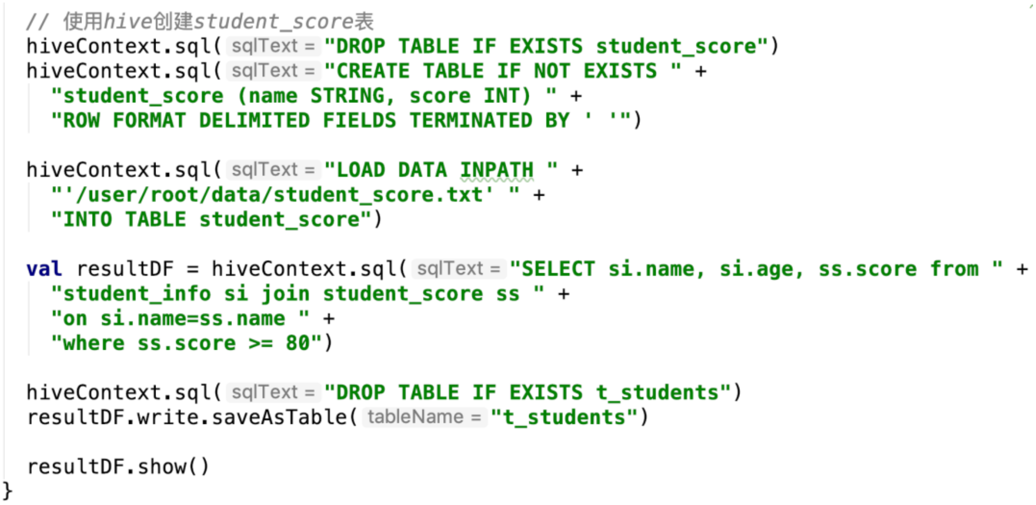
5. 编写脚本



```
/opt/modules/spark-1.6.3-bin-hadoop2.6/bin/spark-submit \
  --class SparkSQLHiveTest \
  --master yarn \
  --deploy-mode cluster \
  --executor-memory 1G \
  --total-executor-cores 2 \
  --files /opt/modules/apache-hive-1.2.1-bin/conf/hive-site.xml \
  --jars /opt/modules/spark-1.6.3-bin-hadoop2.6/lib/datanucleus-api-jdo-3.2.6.jar,/opt/modules/spark-1.6.3-bin-hadoop2.6/lib/dat
anucleus-core-3.2.10.jar,/opt/modules/spark-1.6.3-bin-hadoop2.6/lib/datanucleus-rdbms-3.2.9.jar,/opt/modules/spark-1.6.3-bin-had
oop2.6/lib/mysql-connector-java-5.1.38.jar \
  /opt/modules/spark-1.6.3-bin-hadoop2.6/scala/spark-sql-2.jar
```

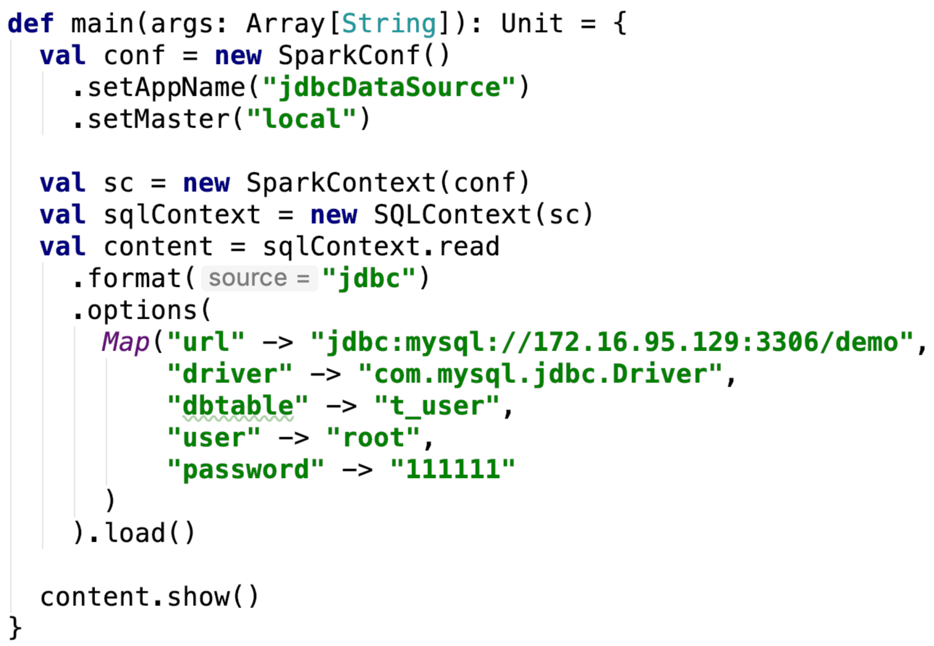


```scala
def main(args: Array[String]): Unit = {
  val conf = new SparkConf()
    .setAppName("hiveDataSource")
    .setMaster("local")

  val sc = new SparkContext(conf)
  val hiveContext = new HiveContext(sc)

  // 使用hive创建student_info表
  hiveContext.sql( sqlText = "DROP TABLE IF EXISTS student_info")
  hiveContext.sql( sqlText = "CREATE TABLE IF NOT EXISTS " +
    "student_info (name STRING, age INT) " +
    "ROW FORMAT DELIMITED FIELDS TERMINATED BY ' '")

  hiveContext.sql( sqlText = "LOAD DATA INPATH " +
    "'/user/root/data/student_info.txt' " +
    "INTO TABLE student_info")
```



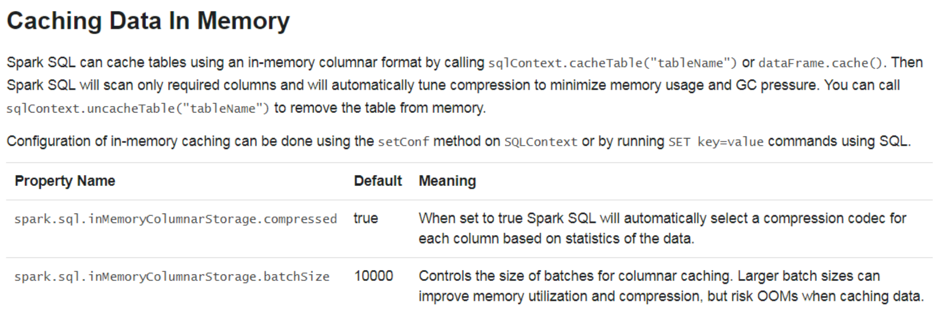
```scala
  // 使用hive创建student_score表
  hiveContext.sql( sqlText = "DROP TABLE IF EXISTS student_score")
  hiveContext.sql( sqlText = "CREATE TABLE IF NOT EXISTS " +
    "student_score (name STRING, score INT) " +
    "ROW FORMAT DELIMITED FIELDS TERMINATED BY ' '")

  hiveContext.sql( sqlText = "LOAD DATA INPATH " +
    "'/user/root/data/student_score.txt' " +
    "INTO TABLE student_score")

  val resultDF = hiveContext.sql( sqlText = "SELECT si.name, si.age, ss.score from " +
    "student_info si join student_score ss " +
    "on si.name=ss.name " +
    "where ss.score >= 80")

  hiveContext.sql( sqlText = "DROP TABLE IF EXISTS t_students")
  resultDF.write.saveAsTable( tableName = "t_students")

  resultDF.show()
}
```

- JDBC 数据源



```scala
def main(args: Array[String]): Unit = {
  val conf = new SparkConf()
    .setAppName("jdbcDataSource")
    .setMaster("local")

  val sc = new SparkContext(conf)
  val sqlContext = new SQLContext(sc)
  val content = sqlContext.read
    .format( source = "jdbc")
    .options(
      Map("url" -> "jdbc:mysql://172.16.95.129:3306/demo",
          "driver" -> "com.mysql.jdbc.Driver",
          "dbtable" -> "t_user",
          "user" -> "root",
          "password" -> "111111"
      )
    ).load()

  content.show()
}
```

- 性能优化

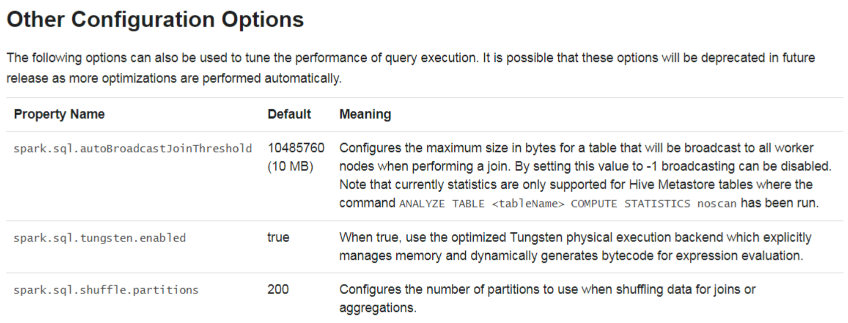


Caching Data In Memory

Spark SQL can cache tables using an in-memory columnar format by calling `sqlContext.cacheTable("tableName")` or `dataFrame.cache()`. Then Spark SQL will scan only required columns and will automatically tune compression to minimize memory usage and GC pressure. You can call `sqlContext.uncacheTable("tableName")` to remove the table from memory.

Configuration of in-memory caching can be done using the `setConf` method on `SQLContext` or by running `SET key=value` commands using SQL.

Property Name	Default	Meaning
`spark.sql.inMemoryColumnarStorage.compressed`	true	When set to true Spark SQL will automatically select a compression codec for each column based on statistics of the data.
`spark.sql.inMemoryColumnarStorage.batchSize`	10000	Controls the size of batches for columnar caching. Larger batch sizes can improve memory utilization and compression, but risk OOMs when caching data.



Other Configuration Options

The following options can also be used to tune the performance of query execution. It is possible that these options will be deprecated in future release as more optimizations are performed automatically.

Property Name	Default	Meaning
`spark.sql.autoBroadcastJoinThreshold`	10485760 (10 MB)	Configures the maximum size in bytes for a table that will be broadcast to all worker nodes when performing a join. By setting this value to -1 broadcasting can be disabled. Note that currently statistics are only supported for Hive Metastore tables where the command `ANALYZE TABLE <tableName> COMPUTE STATISTICS noscan` has been run.
`spark.sql.tungsten.enabled`	true	When true, use the optimized Tungsten physical execution backend which explicitly manages memory and dynamically generates bytecode for expression evaluation.
`spark.sql.shuffle.partitions`	200	Configures the number of partitions to use when shuffling data for joins or aggregations.

https://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-1/

https://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/