mseaspring

首页 新随笔 联系 订阅 管理

spark结构化数据处理：Spark SQL、DataFrame和Dataset

本文讲解Spark的结构化数据处理，主要包括：Spark SQL、DataFrame、Dataset以及Spark SQL服务等相关内容。本文主要讲解Spark 1.6.x的结构化数据处理相关东东，但因Spark发展迅速(本文的写作时值Spark 1.6.2发布之际，并且Spark 2.0的预览版本也已发布许久)，因此请随时关注Spark SQL官方文档以了解最新信息。

文中使用Scala对Spark SQL进行讲解，并且代码大多都能在spark-shell中运行，关于这点请知晓。

概述

相比于Spark RDD API，Spark SQL包含了对结构化数据和在其上的运算的更多信息，Spark SQL使用这些信息进行了额外的优化，使对结构化数据的操作更加高效和方便。

有多种方式去使用Spark SQL，包括SQL、DataFrames API和Datasets API。但无论是哪种API或者是编程语言，它们都是基于同样的执行引擎，因此你可以在不同的API之间随意切换，它们各有各的特点，看你喜欢。

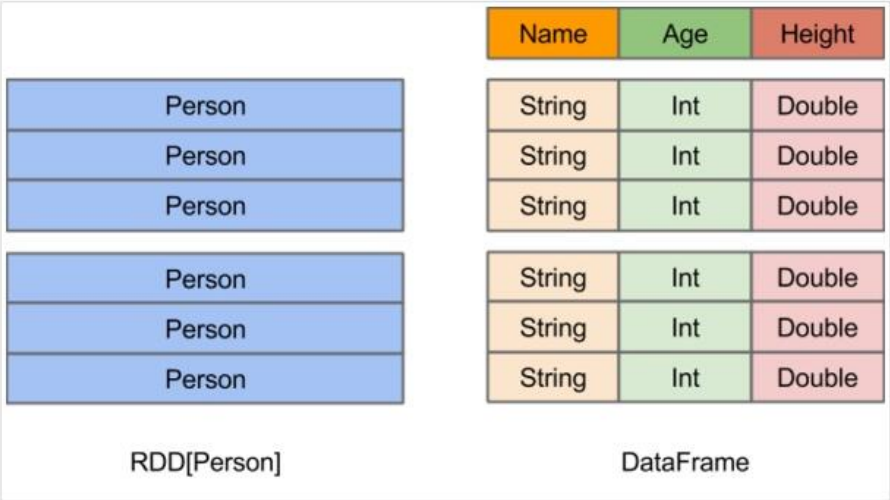
SQL

使用Spark SQL的一种方式就是通过SQL语句来执行SQL查询。当在编程语言中使用SQL时，其返回结果将被封装为一个DataFrame。

DataFrame

DataFrame是一个分布式集合，其中数据被组织为命名的列。它概念上等价于关系数据库中的表，但底层做了更多的优化。DataFrame可以从很多数据源构建，比如：已经存在的RDD、结构化文件、外部数据库、Hive表。

DataFrame的前身是SchemaRDD，从Spark 1.3.0开始SchemaRDD更名为DataFrame。与SchemaRDD的主要区别是：DataFrame不再直接继承自RDD，而是自己实现了RDD的绝大多数功能。你仍旧可以在DataFrame上调用.rdd方法将其转换为一个RDD。RDD可看作是分布式的对象的集合，Spark并不知道对象的详细模式信息，DataFrame可看作是分布式的Row对象的集合，其提供了由列组成的详细模式信息，使得Spark SQL可以进行某些形式的执行优化。DataFrame和普通的RDD的逻辑框架区别如下所示：



DataFrame不仅比RDD有更加丰富的算子，更重要的是它可以进行执行计划优化(得益于Catalyst SQL解析器)，另外Tungsten项目给DataFrame的执行效率带来了很大提升(不过Tungsten优化也可能在后续开发中加入到RDD API中)。

但是在有些情况下RDD可以表达的逻辑用DataFrame无法表达，所以后续提出了Dataset API，Dataset结合了RDD和DataFrame的好处。

关于Tungsten优化可以参见：Project Tungsten：让Spark将硬件性能压榨到极限

Dataset

Dataset是Spark 1.6新添加的一个实验性接口，其目的是想结合RDD的好处(强类型(这意味着可以在编译时进行类型安全检查)、可以使用强大的lambda函数)和Spark SQL的优化执行引擎的好处。可以从JVM对象构造出Dataset，然后使用类似于RDD的函数式转换算子(map/flatMap/filter等)对其进行操作。

Dataset通过Encoder实现了自定义的序列化格式，使得某些操作可以在无需解序列化的情况下直接进行。另外Dataset还进行了包括Tungsten优化在内的很多性能方面的优化。

实际上Dataset是包含了DataFrame的功能的，这样二者就出现了很大的冗余，故Spark 2.0将二者统一：保留Dataset API，把DataFrame表示为Dataset[Row]，即Dataset的子集。

API进化

Spark在迅速的发展，从原始的RDD API，再到DataFrame API，再到Dataset的出现，速度可谓惊人，执行性能上也有了很大提升。

我们在使用API时，应该优先选择DataFrame & Dataset，因为它的性能很好，而且以后的优化它都可以享受到，但是为了兼容早期版本的程序，RDD API也会一直保留着。后续Spark上层的库将全部会用 DataFrame & Dataset，比如MLlib、Streaming、Graphx等。

关于这三种API的更详细的讨论以及选择参见：Apache Spark: RDD, DataFrame or Dataset?

入门

起点之SQLContext

要想使用Spark SQL，首先你得创建一个SQLContext对象，在这之前你只需要创建一个SparkContext就行了，如下：

```
val sqlContext = new org.apache.spark.sql.SQLContext(sc)

// 这个东东用来将RDD隐式转换为DataFrame
import sqlContext.implicits._
```

另外，你也可以使用HiveContext，它是SQLContext的超集，提供了一些额外的功能：使用HiveQL解析器、访问Hive用户定义函数、从Hive表读取数据。并且，你不需要安装Hive就可以使用HiveContext。不过将来版本的Spark SQL可能会逐步缩小SQLContext和HiveContext之间的差距。

对于SQLContext，目前只有一个简单的SQL语法解析器sql，而对于HiveContext，则可以使用hiveql和sql两个解析器，默认是hiveql，我们可以通过如下语句来修改默认解析器：

```
SQLContext.setConf("spark.sql.dialect", "sql")
```

不过就目前来说，HiveQL解析器更加完善，因此推荐使用HiveQL。

创建并使用DataFrame

通过SQLContext，应用程序可以从已经存在的RDD、结构化文件、外部数据库以及Hive表中创建出DataFrame。如下代码从JSON文件(该文件可以从Spark发行包中找到)创建出一个DataFrame：

```
val df = sqlContext.read.json("examples/src/main/resources/people.json")
```

DataFrame提供了一个领域特定语言(DSL)以方便操作结构化数据。下面是一些使用示例，更多更全的DataFrame操作参见Spark API文档中的org.apache.spark.sql.DataFrame：

文章分类(57)

HBASE(13)
HDFS(7)
java(5)
lucene(4)
Parquet(4)
solr(10)
spark(10)
计算机基础知识(4)

优秀博客

JAVA高性能
kafka高手文章
阿里技术团队
虾皮工作室

最新评论

1. Re:NXLog中文文档（1）：简介

博主，可否进一步说下这个nxlog的使用和配置？

--谷仁儿

2. Re:NXLog中文文档(3):配置

亲，能稍微讲下运行逻辑和原理吗，这个nxlog整体的适用场景和常见配置，从实用的角度出发，谢谢啦！

--谷仁儿

3. Re:关于Snmp的Trap代码开发之坑

V3版本遇到过，v2版本没有遇到。

--XGogo

4. Re:关于Snmp的Trap代码开发之坑

2c版本，在高频率发送情况下，接收端会出现部分包接收无响应的情况，是监听性能问题吗？

--funying

5. Re:solr中时区处理

没看懂你的逻辑，solr.in.sh起什么作用？安装文件没有也不影响数据导入和查询 配置它根本不起作用 tomcat/bin/setenv.sh确定tomcati/bin下有setenv.sh吗.....

--前世张公子

阅读排行榜

1. IDEA15 下运行Scala遇到问题以及解决办法(23343)
2. Apache Curator入门实战(11386)
3. Spark性能优化指南——高级篇(10834)
4. 关于Solr6.0中solrj使用简单例子(9463)
5. Zookeeper运维的一些经验[转](7119)

评论排行榜

1. Alluxio 内存存储系统部署(3)
2. 关于Snmp的Trap代码开发之坑(2)
3. spark结构化数据处理：Spark SQL、DataFrame和Dataset(1)
4. Spark的基本说明(1)
5. solr中时区处理(1)

推荐排行榜

1. Apache Curator入门实战(5)
2. Spark性能优化指南——高级篇(2)
3. Spark性能优化指南——基础篇转(1)

```
// 显示DataFrame的内容
df.show()
// age  name
// null Michael
// 30   Andy
// 19   Justin

// 以树形格式打印schema
df.printSchema()
// root
// |-- age: long (nullable = true)
// |-- name: string (nullable = true)

// 选取“name”列
df.select("name").show()
// name
// Michael
// Andy
// Justin

// 选取每一个人，年龄加1
df.select(df("name"), df("age") + 1).show()
// name    (age + 1)
// Michael null
// Andy    31
// Justin  20

// 选取年龄大于21的人
df.filter(df("age") > 21).show()
// age name
// 30  Andy

// 年龄分组计数
df.groupBy("age").count().show()
// age  count
// null 1
// 19   1
// 30   1
```

另外，org.apache.spark.sql.functions单例对象还包含了一些操作DataFrame的函数，主要有这几个方面：聚集操作、集合操作、字符串处理、排序、日期计算、通用数学函数、校验码操作、窗口操作、用户定义函数支持等等。

在程序中执行SQL查询

我们可以通过在程序中使用SQLContext.sql()来执行SQL查询，结果将作为一个DataFrame返回：

```
val df = sqlContext.sql("SELECT * FROM table")
```

我们还可以将DataFrame注册为一个临时表，然后就可以在其上执行SQL语句进行查询了，临时表的生命周期在与之关联的SQLContext结束生命之后结束。示例如下：

```
val df = sqlContext.read.json("examples/src/main/resources/people.json")
df.registerTempTable("people")
val resultDF = sqlContext.sql("SELECT name, age FROM people WHERE age >= 20")
resultDF.show()
// name age
// Andy 30
```

创建并使用Dataset

Dataset跟RDD相似，但是Dataset并没有使用Java序列化库和Kryo序列化库，而是使用特定Encoder来序列化对象。encoder通常由代码自动产生(在Scala中是通过隐式转换产生)，并且其序列化格式直接允许Spark执行很多操作，比如：filtering、sorting、hashing，而不需要解序列化。

4. Kudu:支持快速分析的新型Hadoop存储系统(1)
5. Java8初体验（二）Stream语法详解(1)

```
// SQLContext.implicits(org.apache.spark.sql.SQLImplicits)提供了大多数类型的encoder的实现
// 如下用到了SQLContext.implicits中的localSeqToDatasetHolder隐式转换和newIntEncoder隐式参数
val ds = Seq(1, 2, 3).toDS()
ds.map(_ + 1).collect()         // 返回: Array(2, 3, 4)

// 样例类的encoder也会被自动创建
case class Person(name: String, age: Long)
val ds = Seq(Person("Andy", 32)).toDS()

// DataFrame也可以转换为Dataset(需提供一个class)，并通过name完成映射
val path = "examples/src/main/resources/people.json"
val people = sqlContext.read.json(path).as[Person]
```

与RDD互操作

Spark SQL支持两种不同的方法将RDD转换为DataFrame。第一种方法使用反射去推断包含特定类型对象的RDD的模式(schema)，该方法可以使你的代码更加精简，不过要求在你写Spark程序时已经知道模式信息(比如RDD中的对象是自己定义的case class类型)。第二种方法通过一个编程接口，此时你需要构造一个模式并将其应用到一个已存在的RDD以将其转换为DataFrame，该方法适用于在运行时之前还不知道列以及列的类型的情况。

用反射推断模式

Spark SQL的Scala接口支持将包含case class的RDD自动转换为DataFrame。case class定义了表的模式，case class的参数名被反射读取并成为表的列名。case class也可以嵌套或者包含复杂类型(如序列或者数组)。示例如下：

```
// 假设sc是你的SparkContext变量
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
// 这个东东用来将RDD隐式转换为DataFrame
import sqlContext.implicits._

// 用case class定义模式信息
// 注意: Scala 2.10中case class仅支持22个字段，解决办法是在定义类时继承Product接口
case class Person(name: String, age: Int)

// 创建一个包含Person类型对象的RDD并转换为DataFrame(使用到了隐式转换)
val df = sc.textFile("examples/src/main/resources/people.txt")
  .map(_.split(",")).map(p => Person(p(0), p(1).trim.toInt)).toDF()

df.registerTempTable("people")
val resultDF = sqlContext.sql("SELECT name, age FROM people WHERE age >= 20")
```

手动编程指定模式

当case class不能提前定义时(比如记录的结构被编码为字符串，或者当文本数据集被解析时不同用户需要映射不同的字段)，可以通过下面三步来将RDD转换为DataFrame：

从原始RDD创建得到一个包含Row对象的RDD。

创建一个与第1步中Row的结构相匹配的StructType，以表示模式信息。

通过SQLContext.createDataFrame()将模式信息应用到第1步创建的RDD上。

```
// 引入 Row 和 Spark SQL相关数据类型
import org.apache.spark.sql.Row
import org.apache.spark.sql.types.{StructType,StructField,StringType}

// 原始RDD
val people = sc.textFile("examples/src/main/resources/people.txt")

// 模式信息被编码进字符串
val schemaString = "name age"

// 根据上述字符串生成用StructType表示的模式信息
val schema =
  StructType(
    schemaString.split(" ").map(fieldName => StructField(fieldName, StringType, true)))

// 将原始RDD的每条记录转换为Row类型
val rowRDD = people.map(_.split(",")).map(p => Row(p(0), p(1).trim))

// 将模式信息应用到RDD创建出DataFrame
val peopleDataFrame = sqlContext.createDataFrame(rowRDD, schema)

peopleDataFrame.registerTempTable("people")
val resultDF = sqlContext.sql("SELECT name FROM people")
resultDF.map(t => "Name: " + t(0)).collect().foreach(println)
```

数据源

DataFrame可以当作标准RDD进行操作，也可以注册为一个临时表。将DataFrame注册为一个临时表，允许你在其上执行SQL查询。DataFrame接口可以处理多种数据源，Spark SQL也内建支持了若干种极有用的数据源格式(包括json、parquet和jdbc，其中parquet是默认格式)。此外，当你使用SQL查询这些数据源中的数据并且只用到了一部分字段时，Spark SQL可以智能地只扫描这些用到的字段。

通用加载/保存函数

DataFrameReader和DataFrameWriter中包好一些通用的加载和保存函数，所有这些操作都将parquet格式作为默认数据格式。示例如下：

```
val df = sqlContext.read.load("examples/src/main/resources/users.parquet")
df.select("name", "favorite_color").write.save("namesAndFavColors.parquet")
```

手动指定选项

你也可以手动指定数据源和其他任何想要传递给数据源的选项。指定数据源通常需要使用数据源全名(如org.apache.spark.sql.parquet)，但对于内建数据源，你也可以使用它们的短名(json、parquet和jdbc)。并且不同的数据源类型之间都可以相互转换。示例如下：

```
val df = sqlContext.read.format("json").load("examples/src/main/resources/people.json")
df.select("name", "age").write.format("parquet").save("namesAndAges.parquet")
```

用SQL直接查询文件

也可以不用read API加载一个文件到DataFrame然后查询它，而是直接使用SQL语句在文件上查询：

val df = sqlContext.sql(“SELECT * FROM parquet.`examples/src/main/resources/users.parquet`”)

保存到持久表

可以使用DataFrameWriter.saveAsTable()将一个DataFrame保存到一个持久化表，根据不同的设置，表可能被保存为Hive兼容格式，也可能被保存为Spark SQL特定格式，关于这一点请参见API文档。

我们可以通过SQLContext.table()或DataFrameReader.table()来加载一个表并返回一个DataFrame。

Parquet文件

Parquet格式是被很多其他的数据处理系统所支持的列式数据存储格式。它可以高效地存储具有嵌套字段的记录，并支持Spark SQL的全部数据类型。Spark SQL支持在读写Parquet文件时自动地保存原始数据的模式信息。出于兼容性考虑，在写Parquet文件时，所有列将自动转换为nullable。

加载数据

```
import sqlContext.implicits._

case class Person(name: String, age: Int)
val df = sc.textFile("examples/src/main/resources/people.txt")
  .map(_.split(",")).map(p => Person(p(0), p(1).trim.toInt)).toDF()

// 将数据保存到parquet文件
df.write.parquet("people.parquet")

// 从parquet文件中加载数据
val parquetDF = sqlContext.read.parquet("people.parquet")

parquetDF.registerTempTable("people")
val result = sqlContext.sql("SELECT name FROM people WHERE age >= 20")
result.map(t => "Name: " + t(0)).collect().foreach(println)
```

下面是SQL示例：

```
CREATE TEMPORARY TABLE people
USING org.apache.spark.sql.parquet
OPTIONS (
  path "examples/src/main/resources/people.parquet"
)

SELECT * FROM people
```

分区发现

在很多系统中(如Hive)，表分区是一个通用的优化方法。在一个分区的表中，数据通常存储在不同的目录中，列名和列值通常被编码在分区目录名中以区分不同的分区。Parquet数据源能够自动地发现和推断分区信息。 如下是人口分区表目录结构，其中gender和country是分区列：

```
path
└── to
    └── table
        ├── gender=male
        │   ├── ...
        │   │
        │   ├── country=US
        │   │   └── data.parquet
        │   ├── country=CN
        │   │   └── data.parquet
        │   └── ...
        └── gender=female
            ├── ...
            │
            ├── country=US
            │   └── data.parquet
            ├── country=CN
            │   └── data.parquet
            └── ...
```

当使用SQLContext.read.parquet 或 SQLContext.read.load读取path/to/table时，Spark SQL能够自动从路径中提取分区信息，返回的DataFrame的模式信息如下：

```
root
|-- name: string (nullable = true)
|-- age: long (nullable = true)
|-- gender: string (nullable = true)
|-- country: string (nullable = true)
```

上述模式中，分区列的数据类型被自动推断。目前，支持的数据类型有数字类型和字符串类型。如果你不想数据类型被自动推断，可以配置spark.sql.sources.partitionColumnTypeInference.enabled，默认为true，如果设置为false，将禁用自动类型推断并默认使用字符串类型。

从Spark 1.6.0开始，分区发现默认只发现给定路径下的分区。如果用户传递path/to/table/gender=male作为路径读取数据，gender将不被作为一个分区列。你可以在数据源选项中设置basePath来指定分区发现应该开始的基路径。例如，还是将

path/to/table/gender=male作为数据路径，但同时设置basePath为path/to/table/，gender将被作为分区列。

模式合并

就像ProtocolBuffer、Avro和Thrift，Parquet也支持模式演化(schema evolution)。这就意味着你可以向一个简单的模式逐步添加列从而构建一个复杂的模式。这种方式可能导致模式信息分散在不同的Parquet文件中，Parquet数据源能够自动检测到这种情况并且合并所有这些文件中的模式信息。

但是由于模式合并是相对昂贵的操作，并且绝大多数情况下不是必须的，因此从Spark 1.5.0开始缺省关闭模式合并。开启方式：在读取Parquet文件时，设置数据源选项mergeSchema为true，或者设置全局的SQL选项spark.sql.parquet.mergeSchema为true。示例如下：

```
import sqlContext.implicits._

val df1 = sc.makeRDD(1 to 5).map(i => (i, i * 2)).toDF("single", "double")
df1.write.parquet("data/test_table/key=1")

val df2 = sc.makeRDD(6 to 10).map(i => (i, i * 3)).toDF("single", "triple")
df2.write.parquet("data/test_table/key=2")

val df3 = sqlContext.read.option("mergeSchema", "true").parquet("data/test_table")
df3.printSchema()
// root
// |-- single: int (nullable = true)
// |-- double: int (nullable = true)
// |-- triple: int (nullable = true)
// |-- key : int (nullable = true)
```

JSON数据

SQLContext.read.json()可以将RDD[String]或者JSON文件加载并转换为一个DataFrame。

有一点需要注意的是：这里用的JSON文件并不是随意的典型的JSON文件，每一行必须是一个有效的JSON对象，如果一个对象跨越多行将导致失败。对于RDD[String]也是一样。

```
// 路径可以是单个文件的路径，也可以是目录路径
val path = "examples/src/main/resources/people.json"
val people = sqlContext.read.json(path)
people.printSchema()
// root
//  |-- age: integer (nullable = true)
//  |-- name: string (nullable = true)

// 从RDD[String]加载数据创建DataFrame，每一个string应该是一个JSON对象
val anotherPeopleRDD = sc.parallelize(
  """{"name":"Yin","address":{"city":"Columbus","state":"Ohio"}}""" :: Nil)
val anotherPeople = sqlContext.read.json(anotherPeopleRDD)
```

下面是SQL示例：

```
CREATE TEMPORARY TABLE people
USING org.apache.spark.sql.json
OPTIONS (
  path "examples/src/main/resources/people.json"
)

SELECT * FROM people
```

数据库

Spark SQL也可以使用JDBC从其他数据库中读取数据，你应该优先使用它而不是JdbcRDD，因为它将返回的数据作为一个DataFrame，所以很方便操作。但请注意：这和Spark SQL JDBC服务是不同的，Spark SQL JDBC服务允许其他应用通过JDBC连接到Spark SQL进行查询。

要在Spark SQL中连接到指定数据库，首先需要通过环境变量SPARK_CLASSPATH设置你的数据库的JDBC驱动的路径。例如在spark-shell中连接MySQL数据库，你可以使用如下命令：

```
SPARK_CLASSPATH=mysql-connector-java-5.1.26-bin.jar bin/spark-shell
```

可以通过SQLContext.read.format("jdbc").options(...).load()或SQLContext.read.jdbc(...)从数据库中加载数据到DataFrame。如下示例，我们从MySQL数据库中加载数据：

```
val jdbcDF = sqlContext.read.format("jdbc").options(
  Map("driver" -> "com.mysql.jdbc.Driver",
      "url" -> "jdbc:mysql://<host>:<port>/<database_name>",
      "dbtable" -> "<table_name>",
      "user" -> "root",
      "password" -> "123456")).load()
```

下面是SQL示例：

```
CREATE TEMPORARY TABLE people
USING org.apache.spark.sql.jdbc
OPTIONS (
  driver "com.mysql.jdbc.Driver",
  url "jdbc:mysql://<host>:<port>/<database_name>",
  dbtable "<table_name>",
  user "root",
  password "123456"
)
```

JDBC驱动需要在所有节点的相同路径下都存在，因为是分布式处理嘛，就像Spark核心一样。

有些数据库需要使用大写来引用相应的名字，好像Oracle中就需要使用大写的表名。

分布式SQL引擎

Spark SQL也可以扮演一个分布式SQL引擎的角色，你可以使用JDBC/ODBC或者Spark SQL命令行接口连接到它，并直接执行交互式SQL查询。

运行Thrift JDBC/ODBC Server

Spark SQL中实现的Thrift JDBC/ODBC Server跟Hive中的HiveServer2相一致。可以使用如下命令开启JDBC/ODBC服务，缺省情况下的服务监听地址为localhost:10000：

```
./sbin/start-thriftserver.sh
```

这个脚本不仅可以接受spark-submit命令可以接受的所有选项，还支持–hiveconf 属性=值选项来配置Hive属性。你可以执行./sbin/start-thriftserver.sh –help来查看完整的选项列表。

你可以使用beeline连接到上面已经开启的Spark SQL引擎。命令如下：

```
// 启动beeline
./bin/beeline

// 在beeline中连接到Spark SQL引擎
beeline> !connect jdbc:hive2://localhost:10000
```

连接到beeline需要输入用户名和密码。在非安全模式下，简单地输入你自己的用户名就可以了，密码可以为空。对于安全模式，参见beeline文档。

运行Spark SQL CLI

Spark SQL CLI的引入使得在Spark SQL中可方便地通过Hive metastore对Hive进行查询。目前为止还不能使用Spark SQL CLI与Thrift JDBC/ODBC Server进行交互。这个脚本主要对本地开发比较有用，在共享的集群上，你应该让各用户连接到Thrift JDBC/ODBC Server。

使用Spark SQL CLI前需要注意：

将hive-site.xml配置文件拷贝到$SPARK_HOME/conf目录下。

需要在./conf/spark-env.sh中的SPARK_CLASSPATH添加jdbc驱动的jar包。

启动Spark SQL CLI的命令如下：

```
./bin/spark-sql
```

在启动spark-sql时，如果不指定master，则以local的方式运行，master既可以指定standalone的地址，也可以指定yarn。当设定master为yarn时(spark-sql –master yarn)时，可以通过http://master:8088 页面监控到整个job的执行过程。如

果在./conf/spark-defaults.conf中配置了spark master的话，那么在启动spark-sql时就不需要再指定master了。spark-sql启动之后就可以在其中敲击SQL语句了。

关于Spark SQL CLI的可用命令和参数，请敲击./bin/spark-sql –help以查看。

性能调优

缓存数据在内存中

通过调用sqlContext.cacheTable("tableName")或dataFrame.cache()，Spark SQL可以将表以列式存储缓存在内存中。这样的话，Spark SQL就可以只扫描那些使用到的列，并且将自动压缩数据以减少内存占用和GC开销。你可以调用sqlContext.uncacheTable("tableName")将缓存的表从内存中移除。另外，你也可以在SQL/HiveQL中使用CACHE tablename和UNCACHE tablename来缓存表和移除已缓存的表。

和缓存RDD时的动机一样，如果想在同样的数据上多次运行任务或查询，就应该把这些数据表缓存起来。

将数据缓存在内存中，同样支持一些选项，参见Spark SQL官方文档性能调优部分。

其他调优相关参数

还有其他一些参数可以用于优化查询性能，参见Spark SQL官方文档性能调优部分。

End.

分类: spark

« 上一篇：Spark踩坑记——数据库（Hbase+Mysql）转
» 下一篇：Spark操作Hbase

posted @ 2016-09-01 22:58 XGogo 阅读(2662) 评论(1) 编辑 收藏

评论列表

#1楼 2016-11-19 02:04 暖软大脚丫

你好,请问你这个是2.0.0官方文档翻译过来的???

支持(0) 反对(0)

最新IT新闻:
· 中文在线：与快手战略合作 提高内容吸引力
· 华为回应FB数据泄露受牵连：从没收集或存储FB用户数据
· 腾讯创AsiaSecWest促进中西安全技术交流平台
· 大疆携手安防公司Axon 即将向美国警方出售无人机
· FB与Alphabet今秋将调离科技板块并入电信媒体板块
» 更多新闻...

最新知识库文章:
· 程序员的宇宙时间线
· 突破程序员思维
· 云、雾和霭计算如何一起工作
· 你可以把编程当做一项托付终身的职业
· 评审的艺术——谈谈现实中的代码评审
» 更多知识库文章...

有大数据问题，欢迎加入初级讨论QQ群：大数据初级讨论群 156498981