第5章

Spark SQL编程

Spark SQL 是用于结构化数据处理的 Spark 模块。不同于 RDD API，Spark SQL 提供了更多有关数据和计算的结构信息，并根据这些信息对计算过程进行优化。Spark SQL 接口包括 SQL 和 Dataset API。但无论哪种接口或开发语言(Scala、Java、Python 或 R 等)，都使用相同的执行引擎，开发人员可以在不同的 API 之间进行切换。

Spark SQL 是 Spark 中最重要的概念之一，Spark SQL 允许用户对组织到数据库中的视图或表执行 SQL 查询。用户还可以使用系统函数或自定义函数分析查询计划，优化工作负载。本章将介绍 Spark SQL 中的核心概念，较少涉及 ANSI-SQL 规范或具体的 SQL 表达式。SQL 规范等请参考相关文档。

Spark SQL 用于执行 SQL 查询，可以从 JDBC/ODBC 数据源或 Hive 等数据仓库中读取数据，可以将查询结果以 Dataset/DataFrame 的形式返回给其他编程语言接口。

数据集 Dataset 是分布式数据集合，是 Spark 1.6 版本中添加的接口，集成了 RDD 的优点(强类型、使用 lambda 函数)和 Spark SQL 引擎优化执行的优点。Dataset 可以从 JVM 对象构造，使用转换函数进行操作。Dataset API 提供 Scala 和 Java 接口。

数据帧 DataFrame 是元素类型为 Row 的 Dataset，概念上等效于关系数据库中的表，但增加了更多的优化。DataFrame 可以从多种数据源构建，如，结构化的数据文件、Hive 中的表、外部数据库或已有 RDD 等。在 Scala API 中，DataFrame 只是 Dataset[Row]的一个别名。

5.1 Spark SQL 基础

5.1.1 概述

易用性是 Spark 流行的原因之一。Spark 提供了一个比 Hadoop MapReduce 更简单的编程模型来处理大数据。与 SQL 开发相比，精通 Spark 核心 API 的开发人员要少很多。

SQL 是用于处理数据的一种 ANSI/ISO 标准语言，不仅可以存储、修改和检索数据，还可以分析数据。相比 Scala、Java 或 Python 等通用编程语言，SQL 更容易学习和使用。SQL 同时具有强大的数据处理能力，是数据分析的主要工具之一。

HiveQL 是一种与 SQL 类似的语言，在 Hadoop 中广泛使用，是 Hadoop MapReduce 的首选接口之一。在 Spark 崛起之前，Hive 是事实上的大数据 SQL 访问层。Hive 最初由 Facebook 开发，后来成为大数据业界非常受欢迎的工具。Spark 最初是 RDD 通用处理引擎，因 sqlContext 接口支持 SQL 子集而快速发展，Spark 1.x 中的 HiveContext 接口支持 Hive 的绝大多数功能。Spark 2.0 版本是 Hive 的超集，其中内置 SQL 解析器，同时支持 ANSI-SQL 和 HiveQL 查询。

Spark SQL 的强大功能表现在多个方面：SQL 分析师可以通过接入 Thrift 服务器或 Spark 的 SQL 接口来利用 Spark 的计算能力；而数据工程师和科学家可以通过任一 Spark 支持的编程语言，使用 Spark SQL 编程接口（如 SparkSession 对象的各方法）进行应用开发；此外，DataFrame 还可以传递给 Spark MLlib（机器学习库）中的各个机器学习算法，进行机器学习应用开发。

Spark SQL 旨在作为 OLAP（联机分析处理）数据库运行，而不是作为 OLTP（联机事务处理）数据库运行，不适合非常低延迟的查询。

5.1.2 Spark SQL 架构

Spark SQL 是基于 Spark 核心执行引擎的库，其架构如图 5-1 所示。

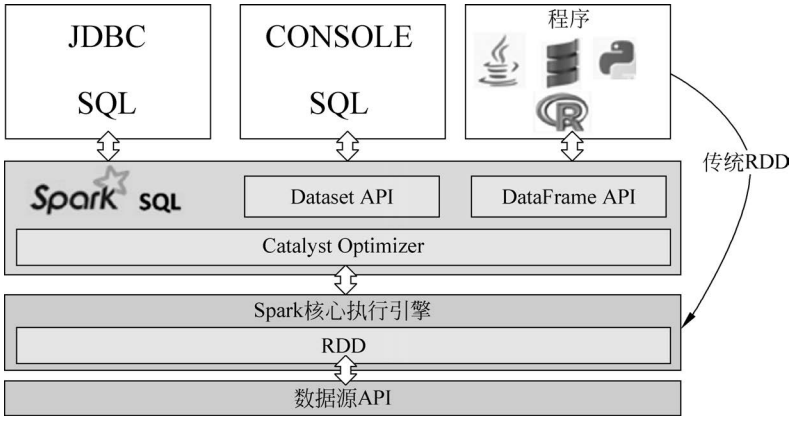

图 5-1 Spark SQL 架构

Spark SQL 是在 Spark Core 执行引擎之上运行的库，提供了比 Spark 核心 API 更高层的抽象来处理结构化数据。结构化数据包括存储在关系数据库、NoSQL 数据库、Parquet、ORC、Avro、JSON、CSV 或任何其他结构化形式的数据。Spark SQL 不仅为 Spark 提供 SQL 接口，使 Spark 更易于使用，还可在提升 Spark 应用程序运行速度的同时，提高开发人员的工作效率。

Spark SQL 可用作 Scala、Java、Python 或 R 应用程序的数据处理开发库，支持多种查询语言，包括 SQL、HiveQL 和语言集成查询。此外，还可以仅通过 SQL/HiveQL 进行交互式分析。

Spark SQL 使用 JDBC/ODBC 为数据仓库应用程序提供 SQL 接口，或通过命令行控制台提供 SQL 交互查询接口。任何商业智能（Business Intelligence，BI）工具都可以连接到 Spark SQL，在内存中执行分析。可基于 API 进行 Java、Scala、Python 或 R 应用程序开发，用户使用数据源（Data Source）API 读写多种数据，创建 Dataset/DataFrame。图 5-1 中也显示了传统的基于 Spark core 和 RDD 进行开发的操作方式。

5.1.3 一个简单的 Spark SQL 开发例子

以下代码展示从 JSON 文件创建 DataFrame,并显示其内容:

```
/**
 * SimpleSparkSqlApp.scala
 *
 * This example illustrates SparkSession / DataFrame
 *
 * @author Rujun Cao
 * @date 2022/10/00
 */
package cn.edu.wzu.SparkExample
// SparkSession 类是所有 Dataset/DataFrame 函数的入口点
import org.apache.spark.sql.SparkSession

object SimpleSparkSqlApp {
  def main(args: Array[String]): Unit = {
// SparkSession.builder()创建 SparkSession
// 在 Spark shell (REPL)中已经创建(无须再次创建),名称为 spark
       val spark = SparkSession.builder()
                .appName("Simple Spark SQL example")
                // 设置特定的配置信息. 此处仅为示例
                .config("spark.some.config.option", "some-value")
                // 代码调用 getOrCreate 方法获取已有或新建的 session 对象
                .getOrCreate()

// 从 JSON 文件创建 DataFrame
    val df = spark.read.json("../tmp/person.json")
// 将 DataFrame 的内容输出(到 stdout)
     df.show()
     spark.stop()
   }
}
```

JSON 文件 person.json 内容示例如下:

```
{"name":"Michael"}
{"name":"Andy", "age":30}
{"name":"Justin", "age":19}
{"name":"Zhaoliu", "age":19, "gender":"male"}
```

在交互式 shell 中加载 JSON 文件并显示内容的执行效果,如图 5-2 所示。

```
val df = spark.read.json("../tmp/person.json")
val df: org.apache.spark.sql.DataFrame = [age: bigint, gender: string ... 1 more
 field]

scala> df.show()
+----+------+-------+
| age|gender|   name|
+----+------+-------+
|null|  null|Michael|
|  30|  null|   Andy|
|  19|  null| Justin|
|  19|  male|Zhaoliu|
+----+------+-------+
```

图 5-2 在 Spark shell 中运行程序

5.2 数据帧 DataFrame

DataFrame 在 Spark 应用程序中非常重要，它通过模式（schema）来包含类型化的数据，并提供功能强大的 API。

作为一个分布式分析引擎，Spark 在某种程度上类似于一个操作系统，提供了构建应用程序和管理资源所需的所有服务（维基百科定义**操作系统**为“管理计算机硬件和软件资源，并为计算机程序提供公共服务的系统软件”）。若要以编程方式使用 Spark，则需要了解其中一些关键的 API。要执行分析和数据操作，Spark 需要逻辑（在应用程序层）存储和物理（在硬件层）存储。在逻辑层，Spark 流行的存储容器是类似于关系表的 DataFrame。

DataFrame 既是数据结构，也是 API，可用于 Spark SQL、Spark Streaming、MLlib（用于机器学习），并可用于操作基于图结构数据的 GraphX。

5.2.1 DataFrame 结构

1. DataFrame 数据组织

DataFrame 是对各列命名的记录集，等效于关系数据库中的表或 Java 中的 ResultSet。数据以分区的形式存储，如图 5-3 所示。

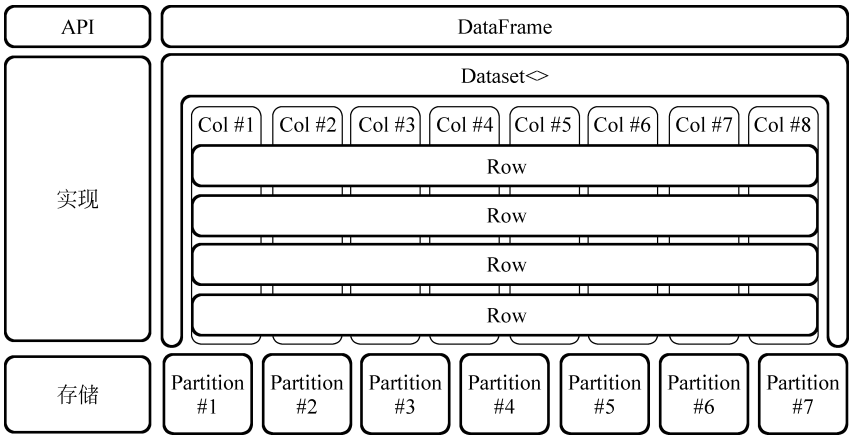

图 5-3 DataFrame 结构

图 5-4 给出了一个 DataFrame 的具体示例：一个带有模式及数据的 DataFrame，命名列描述数据的属性（包含数据类型），数据是行（Row）的集合，存储于分区中。

2. Row

Row（行）是用于表示一行数据的 Spark SQL 抽象。从概念上讲，等效于表中的关系元组或行。Row 对象是将数据传入和传出 Spark 的基本方法，在各种 Spark 开发语言环境中都可以使用。DataFrame 中的每条记录都必须是 Row 类型。Row 构造示例代码如下：

```
spark.range(2).toDF().collect()
```

Spark SQL 提供了用于创建 Row 对象的工厂方法。示例代码如下：

```
import org.apache.spark.sql.Row
val row1 = Row("Joe Biden", "President", "US")
```

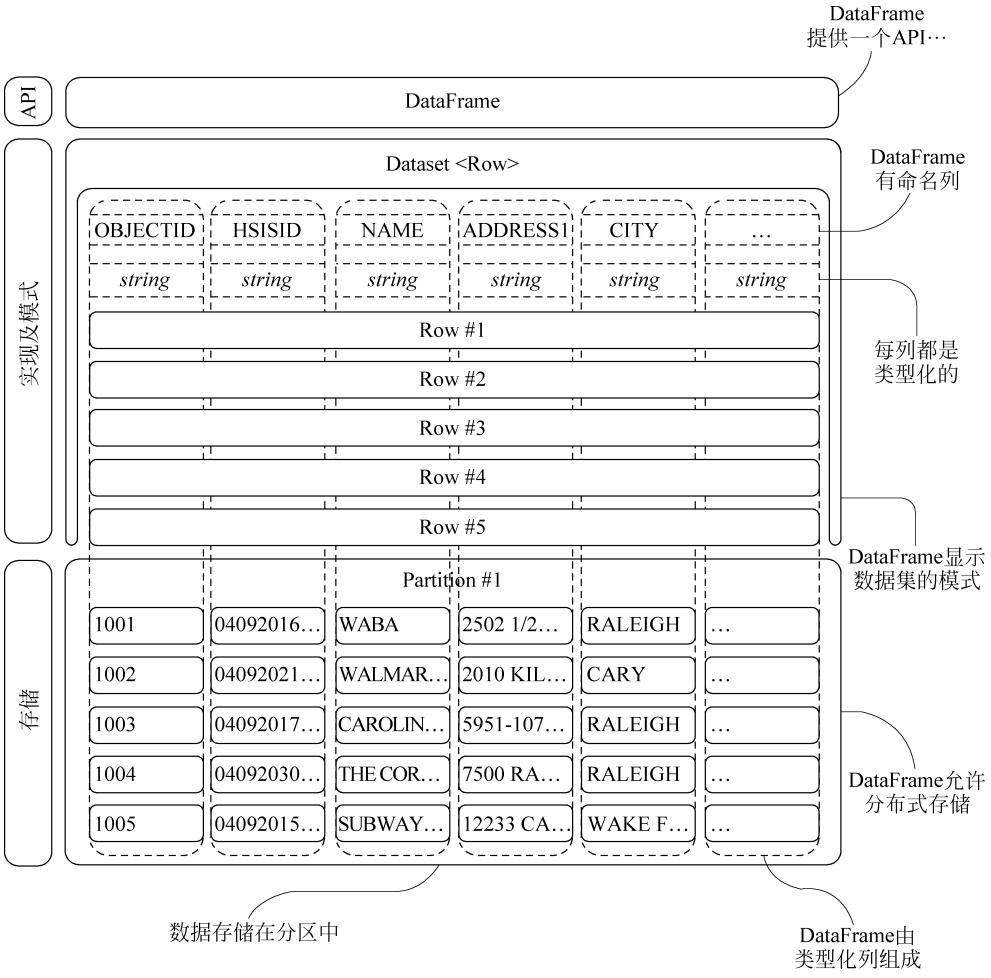

图 5-4 DataFrame 的行与列结构

```
val row2 = Row("Rishi Sunak", "Prime Minister", "UK")
```

当访问行对象的数据时，仅需要指定待访问的位置。由于 Spark 维护自己的内部类型信息，因此在使用时必须手动将其转换为正确的可使用的类型。

```
row1(1)                 // type any
row1.getString(1)       // type String
```

3. Column

Column(列)既可以表示简单数据类型(如整数或字符串)，也可以表示复合类型(如数组或映射)或者空(null)值。Spark 会记录所有这些类型信息，并提供多种列转换方法。

在大多数情况下，可以将 Spark Column 类型视为表中的列。列的计算基于数据帧中的数据，可以从中进行选择、操作和删除列等(这些操作均为表达式)。对列进行操作需要有 Row 对象，而 Row 的存在则以 DataFrame 为基础，也就是说，不能在 DataFrame 之外操作实际的列，只能操作逻辑列的表达式，然后在 DataFrame 上执行该表达式。

Column 构造示例：

```
import org.apache.spark.sql.functions.{col, column}
val col1 = col("column1")
val col2 = column("column2")
$"columnName"                          // Scala 中命名列的简写法
```

可以基于表达式构造列：

```
val col3 = $"a" + 1
```

基于已有 DataFrame 构造或引用列：

```
val df = spark.range(1, 50, 2).toDF()
val id = df("id")
```

Row 对象中列的值可以使用列序号进行访问，如：

```
val presidentName = row1.getString(0)
val country = row2.getString(2)
```

注意：Column 仅仅是表达式，可能存在于数据帧中，也可能不存在。在对列名称与目录中维护的列名称进行比较之前，Spark 不会解析列。

4. Schema

Schema(模式)定义 DataFrame 的列名称和类型。可以直接使用数据源的 Schema，也可以显式自定义 Schema。

Schema 是由多个 StructField(结构字段)组成的 StructType(结构类型)，这些字段具有名称、类型和布尔标志(用于指定该列是否可以包含缺失值或空值)。运行以下命令后结果如图 5-5 所示。

```
spark.read.format("json").load("../tmp/person.json").schema
```

```
scala> spark.read.format("json").load("../tmp/person.json").schema
val res17: org.apache.spark.sql.types.StructType = StructType(StructField(
age,LongType,true),StructField(gender,StringType,true),StructField(name,St
ringType,true))
```

图 5-5　DataFrame 的 Schema 示例

5.2.2　创建 DataFrame

使用 SparkSession，应用程序可以从已有的 RDD、Hive 表或 Spark 数据源创建 DataFrame。

从已有 RDD 创建 DataFrame 主要涉及 SparkSession 类的 createDataFrame 方法；从 Spark 数据源创建则主要是基于 DataFrameReader 类提供的方法，或 DataStreamReader 类的方法，或是 SQL 查询的结果。此外，SparkSession 还提供了创建空数据帧的方法 emptyDataFrame。

1. 从 Dataset 创建 DataFrame

可以直接将 Dataset 转换为 DataFrame，例如，运行以下命令后结果如图 5-6 所示：

```
// spark.range 构造 Dataset，toDF 方法转换为 DataFrame
val df = spark.range(1, 50, 2).toDF()
```

```
scala> val ds = spark.range(1, 50, 2)
     | val df = ds.toDF()
val ds: org.apache.spark.sql.Dataset[Long] = [id: bigint]
val df: org.apache.spark.sql.DataFrame = [id: bigint]
```

图 5-6 将 Dataset 转换为 DataFrame

2. 从数据源创建 DataFrame

可以从各结构化的数据源直接创建 DataFrame，例如，基于 JSON 文件内容创建：

```
val df = spark.read.json("../tmp/person.json")
// 或者
val df = spark.read.format("json").load("../tmp/person.json")
// 创建视图以支持 SQL 查询
df.createOrReplaceTempView("dfTable")
```

从更多数据源创建 DataFrame 等内容，请参考本章后续内容。

3. 从 RDD 创建 DataFrame

为了将已有的 RDD 转换为 DataFrame，Spark SQL 提供了两类不同方法。第一类方法使用反射(reflection)来推断包含特定类型对象的 RDD 模式。如果在编写 Spark 应用程序时模式已知，则基于反射的方法代码更简洁。第二类方法使用编程接口构造模式，再将模式应用于 RDD。这种方法比较烦琐，但在模式(列及其类型)未知时仍可以构造数据集。

这两类方法所使用的具体方法名称分别是 toDF 和 createDataFrame。

(1) 基于反射推断 RDD 模式。Spark SQL 的 Scala 接口支持将包含样例类(case class)的 RDD 自动转换为 DataFrame。样例类定义其模式，类参数名称映射为列名称。样例类也可以嵌套或包含复合类型，如 Seq 或 Array。RDD 可以隐式转换为 DataFrame，然后注册为表(table)。表可以在后续的 SQL 语句中使用。例如，运行以下代码后结果如图 5-7 所示。

```
// RDD 到 DataFrame 隐式转换依赖包
import spark.implicits._
// 样例类
case class Person(name: String, age: Long)
// 从文本文件创建 RDD(Person 对象集合)，转换为 DataFrame
val peopleDF = spark.sparkContext
   .textFile("../tmp/people.txt")
   .map(ln => {val p = ln.split(","); Person(p(0), p(1).trim.toInt)})
   .toDF()
// 将 DataFrame 注册为临时视图，以支持 SQL 操作
peopleDF.createOrReplaceTempView("people")
```

(2) 编程定义 RDD 模式。当无法预先定义样例类时(如，记录的结构被编码为字符串，将解析文本数据集，或者不同用户以不同的方式对字段进行投影等)，可以用编程方式创建 DataFrame。主要包括 3 个步骤：

① 从原始 RDD 创建 Row(行)RDD；

② 创建由结构类型 StructType 表示的 Schema，与 Row 结构相匹配；

③ 通过 SparkSession 提供的 createDataFrame 方法将 Schema 应用于 Row RDD。运

```
scala> // RDD到DataFrame隐式转换依赖包
     | import spark.implicits._
     |
     | case class Person(name: String, age: Long)
     |
     | // 从文本文件创建RDD（Person 对象集合），转换为Dataframe
     | val peopleDF = spark.sparkContext
     |   .textFile("../tmp/people.txt")
     |   .map(ln=> {val p = ln.split(","); Person(p(0), p(1).trim.toInt)})
     |   .toDF()
     | // 将DataFrame注册为临时视图，以支持SQL操作
     | peopleDF.createOrReplaceTempView("people")
     | peopleDF.show(5)
+-------+---+
|   name|age|
+-------+---+
|Michael| 29|
|   Andy| 30|
| Justin| 19|
+-------+---+

import spark.implicits._
class Person
val peopleDF: org.apache.spark.sql.DataFrame = [name: string, age: bigint]
```

图 5-7　反射推理 RDD 模式示例

行以下示例代码结果如图 5-8 所示。

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

// 创建一个 RDD，将其元素转换为 Row
val rowRDD = spark.sparkContext.textFile("../tmp/people.txt")
   .map(ln=>{val p = ln.split(","); Row(p(0), p(1).trim)})

// schema 是编码字符串
val schemaString = "name age"
// 基于字符串生成 schema
val fields = schemaString.split(" ")
   .map(field=>StructField(field, StringType, nullable=true))
val schema = StructType(fields)
// 将模式应用于 RDD
val peopleDF = spark.createDataFrame(rowRDD, schema)
// 将 DataFrame 注册为临时视图，以支持 SQL 操作
peopleDF.createOrReplaceTempView("people")
// SQL 测试：姓名、年龄查询
val results = spark.sql("SELECT name, age FROM people")
results.show(5)
```

提示：是否需要使用自定义 Schema 读取数据，取决于应用场景。将 Spark 用于 ETL 时，最好手动定义 Schema，尤其是在使用 CSV、JSON 等非类型化数据源时，因为 Schema 推断可能因所读取的数据类型而变化。

5.2.3　DataFrame 常用操作

与 RDD 类似，DataFrame 支持许多操作，以下给出部分常用操作。

```
import import org.apache.spark.sql.Row
     | import org.apache.spark.sql.types._
     |
     | // 创建一个RDD
     | // 将其元素转换为Row
     | val rowRDD = spark.sparkContext.textFile("../tmp/people.txt")
     |   .map(ln=> {val p = ln.split(","); Row(p(0), p(1).trim)})
     |
     | // schema是编码字符串
     | val schemaString = "name age"
     | // 基于字符串生成schema
     | val fields = schemaString.split(" ")
     |   .map(field=> StructField(field, StringType, nullable=true))
     | val schema = StructType(fields)
     |
     | // Apply the schema to the RDD
     | val peopleDF = spark.createDataFrame(rowRDD, schema)
     |
     | // 将DataFrame注册为临时视图，以支持SQL操作
     | peopleDF.createOrReplaceTempView("people")
     |
     | // SQL测试：姓名、年龄查询
     | val results = spark.sql("SELECT name,age FROM people")
     | results.show(5)
+-------+---+
|   name|age|
+-------+---+
|Michael| 29|
|   Andy| 30|
| Justin| 19|
|Zhaoliu| 19|
+-------+---+
```

图 5-8 自定义 RDD 模式示例

1. select/selectExpr

select/selectExpr 对 DataFrame 执行相当于 SQL 的数据查询，允许操作数据帧中的列。最简单的方法是将列名字符串作为 select 方法的参数。运行以下示例代码结果如图 5-9 所示。

```
val df = spark.read.format("json").load("../tmp/person.json")
df.select("name", "age").show(5)
```

```
scala> val df = spark.read.format("json").load("../tmp/person.json")
     | df.select("name", "age").show(5)
+-------+----+
|   name| age|
+-------+----+
|Michael|null|
|   Andy|  30|
| Justin|  19|
|Zhaoliu|  19|
+-------+----+
```

图 5-9 select 操作

可以使用不同的方式引用列，给列表达式应用别名等，如：

```
df.select(df.col("name"), col("gender"), expr("age") + 2).show(5)
```

select 后跟 expr 的用法，简写为 **selectExpr**。以下两条语句等价，运行以下示例代码结果如图 5-10 所示。

```
df.select( $ "name", expr("age + 2 AS age_after_two_years")).show(5)
df.selectExpr("name", "age + 2 AS age_after_two_years").show(5)
```

```
scala> df.select($"name", expr("age+2 AS age_after_two_years")).show(5)
+-------+------------------+
|   name|age_after_two_years|
+-------+------------------+
|Michael|              null|
|   Andy|                32|
| Justin|                21|
|Zhaoliu|                21|
+-------+------------------+
```

图 5-10　selectExpr 操作

可以在列表达式中应用聚合函数，运行以下示例代码结果如图 5-11 所示。

```
df.selectExpr("avg(age + 2)", "count(distinct(gender))").show()
```

```
scala> df.selectExpr("avg(age+2)","count(distinct(gender))").show()
+------------------+----------------------+
|    avg((age + 2))|count(DISTINCT gender)|
+------------------+----------------------+
|24.666666666666668|                     1|
+------------------+----------------------+
```

图 5-11　selectExpr 操作中的聚合函数

注意：不要在 select 中混合列对象与列名字符串。

2．withColumn

withColumn 方法添加新列或替换源数据帧中的列，返回新的 DataFrame。该方法需要两个参数：第一个参数是新列的名称，第二个参数是用于生成新列值的表达式。如果列名称参数与已有列名称相同，则替换原有列。运行以下示例代码结果如图 5-12 所示。

```
val df2 = df.select(expr(" * "), lit(1).alias("One"))
df2.withColumn("age_1", $ "age" + $ "One").show(5)
```

```
scala> val df2 = df.select(expr("*"), lit(1).alias("One"))
     | df2.withColumn("age_1", $"age"+$"One").show(5)
+----+------+-------+---+-----+
| age|gender|   name|One|age_1|
+----+------+-------+---+-----+
|null|  null|Michael|  1| null|
|  30|  null|   Andy|  1|   31|
|  19|  null| Justin|  1|   20|
|  19|  male|Zhaoliu|  1|   20|
```

图 5-12　withColumn 操作

3．withColumnRenamed

withColumnRenamed 方法重命名列。如，

```
df2.withColumnRenamed("age_plus_1", "age_1")
```

提示：使用 withColumn 方法也可以重命名列。

4．drop

drop 方法删除列。如果同时删除多个列，用逗号分隔待删除的列名称。删除单个列时，可以使用列名称，也可以使用列对象。代码示例如下：

```
df2.drop( $ "One")
df2.drop("age_plus_1", "One")
```

5. printSchema

printSchema 方法在控制台上以树状形式打印 DataFrame 的模式 schema。如果要控制输出树的深度,可传递树层次参数。不指定层数时,输出整个树结构。运行以下示例代码结果如图 5-13 所示。

```
df.printSchema(2)
```

```
scala> df2.printSchema(2)
root
 |-- age: long (nullable = true)
 |-- gender: string (nullable = true)
 |-- name: string (nullable = true)
 |-- One: integer (nullable = false)
```

图 5-13 printSchema 操作

6. createTempView/createOrReplaceTempView

对 DataFrame 使用 spark.sql 函数执行 SQL 查询前,需要将其转换为数据表或视图。

createTempView 使用给定名称创建本地临时视图。此临时视图的生存期与用于创建此数据集的 SparkSession 相关联。本地临时视图是会话范围的,其生存期是创建它的会话的生存期,会话终止时将自动删除。本地临时视图不绑定到任何数据库,不能使用 db1.view1 来引用本地临时视图。

当临时视图已存在时,createOrReplaceTempView 更新本地临时视图。运行以下示例代码结果如图 5-14 所示。

```
df2.createOrReplaceTempView("person")
spark.sql("select * from person order by age desc").show(5)
```

```
scala> df2.createOrReplaceTempView("person")
     | spark.sql("select * from person order by age desc").show(5)
+----+------+-------+---+
| age|gender|   name|One|
+----+------+-------+---+
|  30|  null|   Andy|  1|
|  19|  null| Justin|  1|
|  19|  male|Zhaoliu|  1|
|null|  null|Michael|  1|
```

图 5-14 createOrReplaceTempView 操作

7. createGlobalTempView/createOrReplaceGlobalTempView

createGlobalTempView 使用给定名称创建全局临时视图。全局临时视图的生存期与 Spark 应用程序相关联。全局临时视图是跨会话的,其生存期是 Spark 应用程序的生存期,应用程序终止时自动删除。全局临时视图与系统保留数据库 global_temp 相关联,必须使用限定名称来引用全局临时视图(如,SELECT * FROM global_temp.view1)。

createOrReplaceGlobalTempView 当全局临时视图已存在时更新之,否则创建。运行以下示例代码结果如图 5-15 所示。

```
df2.createOrReplaceGlobalTempView("people")
spark.sql("SELECT * FROM global_temp.people").show(5)
```

```
scala> df2.createOrReplaceGlobalTempView("people")
     | spark.sql("SELECT * FROM global_temp.people").show(5)
+----+------+-------+---+
| age|gender|   name|One|
+----+------+-------+---+
|null|  null|Michael|  1|
|  30|  null|   Andy|  1|
|  19|  null| Justin|  1|
|  19|  male|Zhaoliu|  1|
```

图 5-15　createOrReplaceGlobalTempView 操作

8．filter/where

对 DataFrame 的行进行过滤时，可以使用布尔表达式，或直接指定条件字符串。如，

```
// 以下 3 行语句等价
df2.filter("age < 20")
df2.filter(col("age") < 20)
df2.where( $ "age" < 20)
```

提示：当使用 AND 连接多个过滤条件时，除了在同一表达式中指定多个条件外，也可以使用 Spark 的链式操作(Spark 引擎可以更有效地对查询进行优化)，即，类似以下的查询，优先选择后一种方式：

```
df2.filter( $ "age" < 20 && $ "name" <=> "Zhaoliu")
df2.filter( $ "age" < 20).filter( $ "name" <=> "Zhaoliu")
```

9．union

union 返回两个 DataFrame 的并集。执行 union 操作的两个 DataFrame 结构必须相同：

```
df2.union(df.withColumn("Two", lit(2))).show(5)
```

10．distinct

distinct 去掉 DataFrame 中的重复行。运行以下示例代码结果如图 5-16 所示。

```
df2.union(df.withColumn("One", lit(1))).distinct().show(5)
```

```
scala> df2.union(df.withColumn("Two", lit(2))).show(5)
     | df2.union(df.withColumn("One", lit(1))).distinct().show(5)
+----+------+-------+---+
| age|gender|   name|One|
+----+------+-------+---+
|null|  null|Michael|  1|
|  30|  null|   Andy|  1|
|  19|  null| Justin|  1|
|  19|  male|Zhaoliu|  1|
|null|  null|Michael|  2|
```

图 5-16　distinct 操作

11．sort/orderBy

对 DataFrame 的行进行排序，可以使用 sort/orderBy 等方法。运行以下示例代码结果如图 5-17 所示。

```
df2.sort("age", "name")                    // 与 orderBy 等效
```

可以使用 asc 和 desc 函数指定排序方向：

```
val df3 = df2.union(df.withColumn("Two", lit(2)))
```

```
df3.orderBy(desc("age"), asc("One")).show(9)
```

```
scala> val df3 = df2.union(df.withColumn("Two", lit(2)))
     | df3.orderBy(desc("age"), asc("One")).show(9)
+----+------+--------+---+
| age|gender|    name|One|
+----+------+--------+---+
|  30|  null|    Andy|  1|
|  30|  null|    Andy|  2|
|  19|  null| Justin|  1|
|  19|  male|Zhaoliu|  1|
|  19|  null| Justin|  2|
|  19|  male|Zhaoliu|  2|
```

图 5-17 orderBy 操作

12. groupBy

groupBy 方法使用指定的列参数对 DataFrame 的行进行分组，返回的结果可用于数据聚合。运行以下示例代码结果如图 5-18 所示。

```
val genderGroup = df.groupBy("gender")
genderGroup.count().show()
```

```
scala> val genderGroup = df.groupBy("gender")
     | genderGroup.count().show()
+------+-----+
|gender|count|
+------+-----+
|  null|    3|
|  male|    1|
```

图 5-18 groupBy 操作

注意：groupBy 方法返回的结果是 RelationalGroupedDataset，而不是 DataFrame。类似地，cube/rollup/pivot 方法也返回 RelationalGroupedDataset。该类主要用于数据聚合(agg 函数)、统计(avg/mean/sum)等。

13. agg

agg 方法对源中 DataFrame 的一列或多列执行指定的聚合，返回结果 DataFrame。运行以下示例代码结果如图 5-19 所示。

```
df2.agg(max("age"), count("gender")).show(1)
```

```
scala> df2.agg(max("age"), count("gender")).show(1)
+--------+-------------+
|max(age)|count(gender)|
+--------+-------------+
|      30|            1|
```

图 5-19 agg 操作

14. describe/summary

describe 方法可用于探索性数据分析。它返回源 DataFrame 中列的统计摘要信息，包括计数、最小值、最大值、平均值和标准差。方法的输入参数为列名称，它采用一列或多列的名称作为参数。如果未指定参数，则计算所有列。运行以下示例代码结果如图 5-20 所示。

```
df2.describe().show()
```

类似地，summary 方法计算数值列和字符串列的统计信息。可用统计量包括 count、

```
scala> df2.describe().show()
+-------+------------------+------+-------+---+
|summary|               age|gender|   name|One|
+-------+------------------+------+-------+---+
|  count|                 3|     1|      4|  4|
|   mean|22.666666666666668|  null|   null|1.0|
| stddev|6.3508529610858835|  null|   null|0.0|
|    min|                19|  male|   Andy|  1|
|    max|                30|  male|Zhaoliu|  1|
+-------+------------------+------+-------+---+
```

图 5-20　describe 操作

mean、stddev、min、max、分位数、count_distinct、approx_count_distinct 等。如果未指定参数，则计算 count/mean/stddev/min/quartiles（percentiles at 25%，50%，and 75%）/max。运行以下示例代码结果如图 5-21 所示。

```
df3.summary().show()
```

```
scala> df3.summary().show()
+-------+------------------+------+-------+------------------+
|summary|               age|gender|   name|               One|
+-------+------------------+------+-------+------------------+
|  count|                 6|     2|      8|                 8|
|   mean|22.666666666666668|  null|   null|               1.5|
| stddev| 5.680375574437544|  null|   null|0.5345224838248488|
|    min|                19|  male|   Andy|                 1|
|    25%|                19|  null|   null|                 1|
|    50%|                19|  null|   null|                 1|
|    75%|                30|  null|   null|                 2|
|    max|                30|  male|Zhaoliu|                 2|
+-------+------------------+------+-------+------------------+
```

图 5-21　summary 操作

注意：describe/summary 函数仅用于探索性数据分析，不保证所生成的结果数据集 schema 的向后兼容性。如果要以编程方式计算汇总统计信息，则使用 agg 函数。

15. na/空值数据处理

na 方法返回的 DataFrameNaFunctions 可用于 DataFrame 中的空值数据处理，如，填充、删除等。运行以下示例代码结果如图 5-22 所示。

```
df2.na.fill("female", Array("gender")).show()
df2.na.drop(Array("age")).show()
```

```
scala> df2.na.fill("female", Array("gender")).show()
     | df2.na.drop(Array("age")).show()
+----+------+--------+---+
| age|gender|    name|One|
+----+------+--------+---+
|null|female|Michael|  1|
|  30|female|    Andy|  1|
|  19|female| Justin|  1|
|  19|  male|Zhaoliu|  1|
```

图 5-22　na 操作

分别填充多个字段的示例代码如下：

```
val valuesToFill = Map("age" -> 22, "gender" -> "X")
df2.na.fill(valuesToFill).show()
```

5.2.4 保存 DataFrame

Spark SQL 使用统一接口，将 DataFrame 写入关系数据库、NoSQL 数据库或各种格式的文件。可以使用 DataFrame 的 write 方法将 DataFrame 保存到各种存储系统。

write 方法返回 DataFrameWriter 实例，该类提供多种将 DataFrame 内容保存到数据源的方法。DataFrameWriter 类的 builder 方法用于指定数据保存的不同选项，如格式、分区及数据处理方式等。write 方法的一般用法类似如下：

```
df.write.format("output-data-source-format").save()
```

- 如果输出为 JSON 文件，则 write 方法用法如下：

```
df.write.format("json").save("../tmp/json/path")
//或者 df.write.json("../tmp/json/path")
```

- 如果输出为 CSV 文件，则 write 方法的用法如下：

```
df3.write.format("csv").option("header", true).save("path")
```

5.3 数据集 Dataset

Dataset 是特定域(domain-specific)的强类型集合(strongly typed collection)，可以使用函数或关系操作并行转换这些对象。Dataset 有一个无类型视图(untyped view)，即 DataFrame，是行(Row)数据集。

在 Spark 2.0 中，DataFrame 只是 Scala 和 Java API 中的行数据集，其定义为：

```
type DataFrame = Dataset[Row]
```

定义在 DataFrame 上的操作也称为“非类型转换”(untyped transformation)，作为对比，强类型(Scala/Java)Dataset 上定义的操作称为“类型转换”(typed transformation)。

Dataset 上可用的操作分为转换和动作。转换生成新的数据集，动作触发计算并返回结果。转换包括 map、filter、select 和 aggregate(如 groupBy)等，动作操作包括 count、show 或写入文件系统等。

Dataset 操作是“惰性的”，即仅在调用动作时触发计算。在 Spark 内部，Dataset 表示描述生成数据所需的逻辑计划。执行动作调用时，Spark 的查询优化器会优化该逻辑计划，并生成物理计划，以便以并行和分布式方式高效执行计划。想要了解逻辑计划以及优化后的物理计划，可以使用 explain 函数。

为有效支持特定域的对象，需要编码器(encoder)。编码器将特定域的类型映射到 Spark 的内部类型系统。例如，给定一个具有两个字段 name (string)和 age (int)的类 Person，编码器会告诉 Spark 在运行时生成代码，将 Person 对象序列化为二进制结构。这种二进制结构通常只需占用较少的内存，并且针对数据处理进行过优化(如，列式存取)。可以使用 schema 函数了解数据的内部二进制表示形式。

Dataset 与 RDD 类似，但不使用 Java 序列化(serialization 或 Kryo)，而是使用特定的编码器来序列化对象，以便通过网络进行传输或处理。虽然编码和标准序列化都负责将对象转换为字节，但编码器动态生成代码，允许 Spark 执行许多操作，而无须将字节反序列化为对象。

5.3.1 创建 Dataset

创建 Dataset 有两种常用方法。最常见的方法是用 SparkSession 的 read 方法读取存储系统中的文件。运行以下创建 Dataset 的示例代码，结果如图 5-23 所示。

```
case class Person(name: String, age: Int, gender: String)

// 样例类编码器
val caseDS = Seq(Person("Wangwu", 22, "male")).toDS()
// 导入 spark.implicits._，为许多常用类型自动提供编码器
val ds2 = Range(1, 5).toDS()
spark.range(2).toDS()

// 通过名称映射将 DataFrame 转换为 Dataset
val jsonDS = spark.read
   .schema("name String, age Int, gender String")
   .json("../tmp/person.json").as[Person]
jsonDS.show(5,false)
```

```
scala> val jsonDS = spark.read.schema("name String, age Int, gender String")
     | .json("../tmp/person.json").as[Person]
     | jsonDS.show(5, false)
+-------+----+------+
|name   |age |gender|
+-------+----+------+
|Michael|null|null  |
|Andy   |30  |null  |
|Justin |19  |null  |
|Zhaoliu|19  |male  |
```

图 5-23　创建 Dataset

5.3.2 Dataset 常用方法

Dataset 的方法可用于 DataFrame，以下给出了部分常用操作。

1. as

as 方法将 Dataset 的记录映射为指定的类型。列映射方法取决于编码器 U 的类型：

(1) U 是 class，该类的字段将映射到同名的列(spark.sql.caseSensitive 区分字母大小写)；

(2) U 是元组 tuple，列将按序数映射(即将第一列分配给_1)；

(3) U 是基本类型(字符串、整数等)，使用 DataFrame 的第一列。

如果 Dataset 的 schema 与所需的 Encoder 类型不匹配，则可根据需要进行选择，使用别名、重新排列或者重命名。

注意：as 仅更改传递到类型化操作(如 map)中的数据视图，不移除指定类中不存在的任何列。

2. cache/persist

cache/persist 方法持久化 Dataset。cache 及不带参数的 persist 使用默认存储类别(MEMORY_AND_DISK)。

3. checkpoint

checkpoint 方法返回此数据集的检查点版本信息。在计算量可能呈指数增长的迭代算

法中，checkpoint 方法尤其有用。checkpoint 被保存到 SparkContext＃setCheckpointDir 设置的目录中。

5.4 数据源

通过 DataFrame 接口，Spark SQL 支持对多种数据源的操作。可以使用关系转换、创建临时视图等对 DataFrame 进行操作。视图支持 SQL 查询。本节介绍 Spark 数据源的加载和保存方法，以及可用于内置数据源的特定选项。

Spark 中的数据读取接口在 DataFrameReader 中定义，通过 SparkSession 对象的 read 属性访问；数据保存接口在 DataFrameWriter 中定义，通过 Dataset 对象的 write 属性访问。

5.4.1 通用 load/save 函数

Spark SQL 中的默认数据源是 parquet 格式（由 spark.sql.sources.default 设置）。示例代码如下：

```
val userDF = spark.read.load("../tmp/users.parquet")
userDF.write.save("../tmp/users-parquet-dir")
```

1. 常用选项

数据源类型由其全名称（如 org.apache.spark.sql.parquet）指定，但对内置数据源，可以使用短名称（如 json、parquet、jdbc、orc、libsvm、csv、text 等）。在加载或保存数据源时，可以指定相应的参数或选项。具体的内置数据源及其相关参数或选项，请参考 API 文档。

加载的数据源可以转换为其他数据源。例如，将 json 文件保存为 csv 格式：

```
val jsonDF = spark.read.format("json").load("people.json")
jsonDF.write.format("csv").save("../tmp/people-csv-dir")
```

2. read mode

以下代码示例使用特定选项加载 csv 文件：

```
val csvDF = spark.read.format("csv")
            .option("header", "true")
            .option("inferSchema", "true")
            .option("sep", ";")
            .load("people.csv")
```

读取数据源时，通常会指定 format（格式）、schema（模式）、read mode（读取模式）、option（选项）以及 path（路径）。至少必须提供格式和路径参数。

读取模式设置 Spark 遇到格式错误的记录时的处理方式，包括：

（1）permissive 在遇到损坏的记录时将所有字段设置为 null（默认值）；

（2）dropMalformed 删除错误行；

（3）failFast 遇到格式错误的记录时立即失败（抛出异常）。

3. save mode

以下代码示例使用特定选项将数据集保存为 csv 格式：

```
csvDF.write.format("csv")
            .option("mode", "OVERWRITE")
            .option("dateFormat", "yyyy-MM-dd")
            .save("path/to/file")
```

使用 DataFrameWritier 保存数据时，通常会指定 format（格式）、save mode（写入模式）、option（选项）以及 path（路径），至少必须提供路径参数。

save mode 指定如果在输出位置已存在数据时的处理方式，包括：

（1）append 追加数据；

（2）overwrite 覆盖现有数据；

（3）errorIfExists 中止操作（默认值，抛出异常，或写作 error）；

（4）ignore 忽略（无操作）。

4. SQL 直连文件

除了使用读取 API 将文件加载到 DataFrame 再执行查询外，还可以直接使用 SQL 查询文件。运行以下示例代码，结果如图 5-24 所示。

```
val sqlDF = spark.sql("SELECT * FROM parquet.`../tmp/users.parquet`")
sqlDF.show()
```

```
scala> val sqlDF = spark.sql("SELECT * FROM parquet.`../tmp/users.parquet`")
     | sqlDF.show()
+------+--------------+----------------+
|  name|favorite_color|favorite_numbers|
+------+--------------+----------------+
|Alyssa|          null|  [3, 9, 15, 20]|
|   Ben|           red|              []|
```

图 5-24　直接使用 SQL 查询文件

5. 将数据保存到表

使用 saveAsTable 方法可以将 DataFrame 持久保存到 Hive metastore 的表中。与 createOrReplaceTempView 命令不同，saveAsTable 存储的是数据帧的内容，并在 Hive metastore 中存储元数据，即使重启 Spark，持久表仍然存在。运行以下示例代码，结果如图 5-25 所示。

```
sqlDF.write.mode("overwrite").saveAsTable("t_user")
sql("select * from t_user").show(5)
```

```
scala> sqlDF.write.mode("overwrite").saveAsTable("t_user")
     | sql("select * from t_user").show(5)
+------+--------------+----------------+
|  name|favorite_color|favorite_numbers|
+------+--------------+----------------+
|Alyssa|          null|  [3, 9, 15, 20]|
|   Ben|           red|              []|
```

图 5-25　saveAsTable 方法保存数据到表

当模式为追加时，如存在同名表，则使用已有表的格式和选项。DataFrame 的 schema 中的列的顺序不需要与现有表的列顺序相同，saveAsTable 将使用列名来查找正确的位置。

注意：此功能不需要部署 Hive。Spark 使用 Derby 创建一个默认的本地 Hive metastore。

5.4.2 文件数据源

1. 文件数据源选项

通用的文件数据源选项包括以下几种。

(1) 忽略损坏的文件。当 spark.sql.files.ignoreCorruptFiles 设置为 true 时,Spark 任务在遇到被损坏的文件时继续运行,并且仍会返回已读取的内容,示例代码如下:

```
// enable ignore corrupt files
spark.sql("set spark.sql.files.ignoreCorruptFiles = true")
// 路径中非 parquet 文件被忽略
val testCorruptDF = spark.read.parquet(
          "examples/src/main/resources/dir1/",
          "examples/src/main/resources/dir1/dir2/")
```

(2) 忽略缺失文件。Missing File(缺失的文件)是指在构造数据帧 DataFrame 之后,目录中已删除的文件。当 spark.sql.files.ignoreMissingFiles 设置为 true 时,Spark 从文件中读取数据时忽略缺失的文件,Spark 任务在遇到文件缺失时继续运行,并返回已读取的内容。

(3) 文件过滤器。pathGlobFilter 用于仅包含与模式匹配的文件,用法同 org.apache.hadoop.fs.GlobFilter,不改变发现分区的方式。示例代码如下:

```
// 仅加载 parquet 格式文件,滤出 json 等其他格式文件
spark.read.format("parquet")
        .option("pathGlobFilter", "*.parquet")
        .load("examples/src/main/resources/dir1")
```

(4) 递归文件查找。recursiveFileLookup 用于递归加载文件,并禁用分区推理。默认值为 false,如果数据源在 recursiveFileLookup 为 true 时显式指定了 partitionSpec,则会引发异常。示例代码如下:

```
// 递归加载指定路径(及子目录)下的文件
spark.read.format("parquet")
        .option("recursiveFileLookup", "true")
        .load("examples/src/main/resources/dir1")
```

(5) 路径修改时间过滤器。modifiedBefore 和 modifiedAfter 可以同时使用或单独使用,以实现对所加载数据的更细粒度的控制(Structured Streaming 文件源不支持)。

modifiedBefore/modifiedAfter 为可选时间戳,用于仅包含修改时间早于/晚于指定时间的文件。时间戳格式为:YYYY-MM-DDTHH:mm:ss(如,2022-11-02T11:02:02)。如果未提供时区选项,那么将根据 Spark 会话时区(spark.sql.session.timeZone)解释时间戳。运行以下示例代码,结果如图 5-26 所示。

```
// 加载限定时间范围内的文件
spark.read.format("parquet")
        .option("pathGlobFilter", "*.parquet")
        .option("modifiedBefore", "2022-11-02T11:02:02")
        .option("modifiedAfter", "2000-01-01T00:00:00")
        .load("examples/src/main/resources/dir1")
        .show(5)
```

```
scala> spark.read.format("parquet")
     | .option("pathGlobFilter", "*.parquet")
     | .option("modifiedBefore", "2022-11-02T11:02:02")
     | .option("modifiedAfter",  "2000-01-01T00:00:00")
     | .load("examples/src/main/resources/dir1")
     | .show(5)
+-------------+
|         file|
+-------------+
|file1.parquet|
```

图 5-26　数据加载过滤器示例

2. CSV 文件数据源

CSV(Comma-Separated Value)是一种常见的文本文件格式，其中每行表示由多个列组成的单个记录。

CSV 文件虽然看起来结构良好，但实际处理比较复杂，因为在实际生产中无法对其包含的内容或结构方式做出太多假设。因此，CSV reader 具有较多的选项，用于处理诸如字符转义问题。

Spark SQL 提供 spark.read().csv("file_name")将 CSV 格式的文件或文件目录读取到 DataFrame 中，提供 dataframe.write().csv("path")方法写入 CSV 文件。函数 option()可用于自定义读取或写入的行为，例如标题头、分隔符、字符集等。例如，

```
// Read a csv with delimiter and a header
spark.read.option("delimiter", ";")
         .option("header", "true").csv("path-to-csv")
```

CSV 数据源常用选项如表 5-1 所示。

表 5-1　CSV 文件源常用选项

选　项　名	含　　义	默　认　值	读/写
sep/delimiter	列分隔符	,	RW
encoding	文件编码方式	UTF-8	RW
quote	设置用于转义单个字符的引号。引号内的分隔符作为普通字符	“	RW
escape	转义符(单字符)，用以转义引号内的引号符(作为普通字符)	\	RW
escapeQuotes	是否将包含引号的值括在引号中	true	W
header	对于读，使用第一行作为列的名称 对于写，将列名称写入第一行	false	RW
dateFormat	日期格式	yyyy-MM-dd	RW
nanValue	Not-a-Number 的字符串表示形式	NaN	R
ignoreLeadingWhiteSpace	忽略前导空格符	false	RW
ignoreTrailingWhiteSpace	忽略尾部空格符	false	RW
mode	解析期间处理损坏记录的模式	PERMISSIVE	R
locale	区域语言标记，IETF BCP 47 格式	en-US	R
compression	压缩编解码器，none、bzip2、gzip、lz4、snappy 和 deflate		W

3. JSON 数据源

JSON(JavaScript Object Notation)是 JavaScript 的常用数据格式。在 Spark 中使用的 JSON 文件,是行分隔(line-delimited)的 JSON 文件(每行包含一个单独的、有效 JSON 对象),区别于具有大型 JSON 对象或数组的文件。行分隔的 JSON 是一种更稳定的格式,可以方便地将新记录添加到文件中(而不必读取整个文件然后再写),更容易使用。JSON 对象具有结构,JavaScript(JSON 所基于的)具有类型,因此,Spark 可以对行分隔 JSON 作更多假设(选项比 CSV 少)。普通的多行 JSON 文件需要将 multiLine 选项设置为 true。

Spark SQL 可自动推断 JSON 数据集的模式,将其加载为 DataFrame(即 Dataset[Row])。SparkSession.read.json()方法既可以加载 JSON 文件,也可以转换 Dataset[String]。例如(运行结果见图 5-27):

```
// 读 JSON 文件,参数可以是文件名也可以是目录
spark.read.option("mode", "FAILFAST")
        .json("../tmp/person.json")
        .show(5)
// 从 JSON 数据集(Dataset[String])创建 DataFrame
val personDS = spark.createDataset("""
    {"name":"Cao","addr":{"city":"Wenzhou","state":"Zhejiang"}}""" ::Nil)
val personDF = spark.read.json(personDS)
```

```
scala> val personDS = spark.createDataset(
     |   """{"name":"Cao","addr":{"city":"Wenzhou","state":"Zhejiang"}}""" ::Nil)
     | val personDF = spark.read.json(personDS)
     | personDF.show(false)
     | personDS.show(false)
+------------------+----+
|addr              |name|
+------------------+----+
|{Wenzhou, Zhejiang}|Cao |
+------------------+----+

+-------------------------------------------------------------+
|value                                                        |
+-------------------------------------------------------------+
|{"name":"Cao","addr":{"city":"Wenzhou","state":"Zhejiang"}}|
+-------------------------------------------------------------+

val personDS: org.apache.spark.sql.Dataset[String] = [value: string]
val personDF: org.apache.spark.sql.DataFrame = [addr: struct<city: string, state:
 string>, name: string]
```

图 5-27 JSON 数据源加载为 DataFrame/Dataset 示例

JSON 数据源常用选项见表 5-2。

表 5-2 JSON 数据源常用选项

选 项 名	含 义	默 认 值	读/写
timeZone	时区	spark.sql.session.timeZone	RW
allowComments	忽略 Java/C++样式注释	false	R
allowSingleQuotes	是否允许使用单引号	false	R
mode	解析期间处理损坏记录的模式	PERMISSIVE	R
dateFormat	日期格式	yyyy-MM-dd	RW
timestampFormat	时间戳格式	yyyy-MM-dd'T'HH:mm:ss[.SSS]	RW

续表

选 项 名	含 义	默 认 值	读/写
multiLine	记录可跨多行	false	R
encoding	文件编码方式	读，自动检测；写，UTF-8	RW
lineSep	行分隔符	读，\r，\n，\r\n；写，\n	RW
dropFieldIfAllNull	忽略全 null，或全空数组/结构的列	false	R
ignoreNullFields	忽略空字段	spark. sql. jsonGenerator. ignoreNullFields	W

4. Parquet 数据源

Apache Parquet 是面向列的开源数据存储格式，提供各种存储优化策略，尤其适用于数据分析。Parquet 格式文件可以通过列压缩节省存储空间，并允许读取单个列而非整个文件。Parquet 是 Spark 的默认文件格式，读 Parquet 文件比 JSON 或 CSV 更高效。Parquet 支持复杂类型，其列数据可以是数组(CSV 文件不支持)、map 或 struct 等。示例代码如下：

```
val userDF = spark.read.parquet("../tmp/users.parquet")
userDF.write.save("../tmp/users-parquet-dir")
```

Parquet 的可选项较少。除压缩方式 compression 外，另一个选项是 mergeSchema，其默认设置是由 spark. sql. parquet. mergeSchema 确定的。有多种数据处理系统支持 Parquet 格式。Spark SQL 支持读取和写入 Parquet 文件时自动保留原始数据的 Schema。读取 Parquet 文件时，出于兼容性考虑，所有列都将自动设置为可为空(nullable)。

类似于 Protocol Buffer、Avro 和 Thrift，Parquet 也支持 schema 演进。用户可以从简单的 schema 开始，根据需要逐渐添加更多列，最终可能会得到多个具有不同 schema 但相互兼容的 Parquet 文件。Parquet 数据源能够自动检测并合并所有这些文件的 schema。

由于模式合并比较耗费资源，且在多数情况下不是必需的，因此其默认状态是关闭的。可以将 spark. sql. parquet. mergeSchema 设置为 true，或是在读取文件时将 mergeSchema 设置为 true。

运行以下示例代码，结果如图 5-28 所示。

```
import spark.implicits._
// 由 RDD 创建 DataFrame(value: Int, square: Int)
val squaredDF = spark.sparkContext.makeRDD(1 to 5)
        .map(i => (i, i * i)).toDF("value", "square")
// 另一 DataFrame(value: Int, cube: Int),cube <-- square
val cubedDF = spark.sparkContext.makeRDD(6 to 10)
        .map(i => (i, i * i * i)).toDF("value", "cube")
// 将两个 DataFrame,存储到不同分区
squaredDF.write.parquet("data/test_table/key=1")
cubedDF.write.parquet("data/test_table/key=2")
// 加载分区数据,schema 合并
val mergedDF = spark.read.option("mergeSchema", "true")
        .parquet("data/test_table")
mergedDF.printSchema()
```

注意：不同版本的 Parquet 文件可能不兼容。使用 Spark 不同版本(尤其是旧版本)写 Parquet 文件时要注意文件格式版本。

```
scala> // 加载分区数据，schema合并
     | val mergedDF = spark.read.option("mergeSchema", "true")
     | .parquet("data/test_table")
     | mergedDF.printSchema()
root
 |-- value: integer (nullable = true)
 |-- square: integer (nullable = true)
 |-- cube: integer (nullable = true)
 |-- key: integer (nullable = true)
```

图 5-28 Parquet 数据源合并

5. ORC 数据源

Apache ORC 是一种列式存储格式(Optimized Row Columnar,ORC)文件,具有 zstd 压缩、布隆过滤器(bloom filter)和列式加密等功能。ORC 是借鉴 Hive 的一种高效文件格式。Spark 了解 ORC 文件格式细节,读数据时仅提供 mergeSchema 选项。ORC 与 Parquet 非常相似,但 Spark 针对 Parquet 会有特殊优化。

Spark 支持两种 ORC 实现(内置实现和 Hive 实现),由 spark.sql.orc.impl 控制,两种实现的大多数功能相同(设计目标不同)。内置实现遵循与 Parquet 一致的 Spark 数据源行为; Hive 实现使用 Hive SerDe,遵循 Hive 规范。

在 Spark 一些历史版本的内置实现中,使用内置 String 处理 CHAR/VARCHAR,而 Hive 实现使用 Hive CHAR/VARCHAR(查询结果不同)。从 Spark 3.1.0 开始,Spark 端支持 CHAR/VARCHAR(差异消除)。

ORC 数据读写示例代码如下:

```
val flights = spark.read.format("orc").load("../tmp/flight.orc")
csvDF.write.format("orc").save("../tmp/csv-to-orc")
```

6. Text 文件数据源

Spark SQL 可以直接读写文本文件,文件内容被解析为一组字符串。读文本文件的方法是 spark.read().text(),可以将文本文件或目录读入 DataFrame。读文本文件时,每行都是 Row 对象,默认情况下,其 value 列的内容是文本行内容。写文件使用 dataframe.write().text()方法。option()方法可以修改默认的行分隔符、设置压缩方式等。示例代码如下:

```
val txt = spark.read.option("lineSep", ",").text("test.txt")
txt.write.option("compression","gzip").text("compressed_txt")
```

option 方法可以修改默认的行分隔符(lineSep,读或写,默认值为"\n"),压缩方式(compression,写,默认无压缩)。

5.4.3 Hive 数据源

Spark SQL 支持 Apache Hive 数据源的读写。但 Hive 有大量依赖项,这些依赖项不包含在默认的 Spark 发行版本中。如果在类搜索路径上可以找到 Hive 依赖项,那么 Spark 将自动加载它们(这些 Hive 依赖项还必须存在于所有的 Worker 节点,因为它们需要访问 Hive 序列化和反序列化库 SerDes 才能访问 Hive 中的数据)。

配置 Hive 主要是通过 conf/目录中的 Hive-site.xml、core-site.xml 及 hdfs-site.xml

来完成。具体配置过程请参考 Hive 的相关文档。

使用 Hive 数据源时，必须使用支持 Hive 的 SparkSession 实例，包括 Hive metastore 连接、Hive serdes 的支持，以及 Hive 用户定义函数的支持。未部署 Hive 时依然可以启用 Hive 支持。未配置 hive-site.xml 时，Spark context 会自动在当前目录下创建 metastore_db 目录，并创建由 spark.sql.warehouse.dir 配置的目录，该目录默认为 Spark 应用程序启动时的当前目录下的 spark-warehouse，相应地，启动 Spark 应用程序的用户需要写权限。

Hive 数据源读写示例代码如下：

```
// 创建支持 Hive 的 SparkSession 对象
val spark = SparkSession.builder()
    .appName("Spark Hive Example")
    .config("spark.sql.warehouse.dir", "spark-warehouse")
    .enableHiveSupport()
    .getOrCreate()
// 创建 Hive 数据表，加载数据
spark.sql("create TABLE src(key INT, value STRING) USING hive")
spark.sql("load DATA LOCAL INPATH 'kv1.txt' into TABLE src")
// 读 Hive 数据表，创建 DataFrame
val hiveDF = spark.sql("select * FROM src WHERE key < 100")
// 创建 Hive 表，parquet 存储格式，将 DataFrame 写入表中
sql("create TABLE hive_t(key int, value string) STORED as PARQUET")
hiveDF.write.mode(SaveMode.Overwrite).saveAsTable("hive_t")
```

5.4.4 SQL 数据源

Spark SQL 可以连接到各种 SQL 数据源，如 MySQL、PostgreSQL，或 Oracle、SQLite 等数据库。有别于文件数据源，在如何连接到数据库时需要考虑数据库连接选项，包括身份认证和连接方式（Spark 集群的网络是否连接到数据库网络）等。

Spark SQL 可以使用 JDBC 从其他数据库读取数据，结果以 DataFrame 的形式返回。与 RDD（JdbcRDD）相比，应优先使用此方式，以便 Spark SQL 对数据进行处理，或与其他数据源连接。JDBC 数据源也更易用于 Java 或 Python 环境，因其不需要用户提供 ClassTag（不同于 Spark SQL JDBC 服务器，其他应用程序使用 Spark SQL 运行查询）。

使用 JDBC 连接数据源时，需要在 Spark 类路径中包含特定数据库的 JDBC 驱动程序。例如，要想使用 Spark Shell 连接 PostgreSQL 数据库，需要运行如下命令：

```
./bin/spark-shell --jars ./jars/postgresql-42.5.0.jar
```

1. PostgreSQL 数据源

Spark 支持的 JDBC 选项不区分字母大小写。JDBC 数据源选项通过相应的 option/options 方法设置（DataFrameReader/DataFrameWriter）。

对 JDBC 数据库连接属性，可以在数据源选项中指定，如用户名、密码、数据库服务器链接地址等。读写 PostgreSQL 数据源，需要使用 PostgreSQL JDBC 驱动程序。

运行以下示例代码，结果如图 5-29 所示。

```
// 使用 load 方法加载 PostgreSQL 数据源
// option 方法设置数据源连接属性
```

```
val pgDF = spark.read.format("jdbc")
                .option("url", "jdbc:postgresql:dbserver")
                .option("dbtable", "schema.tablename")
                .option("user", "username")
                .option("password", "password")
                .load()

// 将 DataFrame 写到 PostgreSQL 数据表中
pgDF.write.format("jdbc")
                 .option("url", "jdbc:postgresql:dbserver")
                 .option("dbtable", "write_to_table")
                 .option("user", "username")
                 .option("password", "password")
                 .save()
```

```
scala> val pgDF = spark.read
     |   .format("jdbc")
     |   .option("url", "jdbc:postgresql:hive")
     |   .option("dbtable", "person")
     |   .option("user", "hive")
     |   .option("password", "hive")
     |   .load()
val pgDF: org.apache.spark.sql.DataFrame = [id: int, nam

scala> // Saving data to a JDBC source
     | pgDF.write
     |   .format("jdbc")
     |   .option("url", "jdbc:mysql://localhost/hive")
     |   .option("dbtable", "write_to_person")
     |   .option("user", "root")
     |   .option("password", "Password123#@!")
     |   .save()
```

图 5-29 连接 PostgreSQL 数据源示例

2. MySQL 数据源

MySQL 数据源的读写与 PostgreSQL 类似，连接时需要指定 MySQL 数据库服务器。以下示例使用 java.util.Properties 传递 MySQL 连接信息（需要加载 MySQL JDBC 驱动程序）。

运行以下示例代码，结果如图 5-30 所示。

```
// 使用 load 方法加载 MySQL 数据源
// 使用 Properties 传递数据源连接属性
val connProp = new java.util.Properties()
connProp.put("user", "username")
connProp.put("password", "password")
val myDF = spark.read
        .jdbc("jdbc:mysql:dbserver", "tablename", connProp)

// 类似地，可以将 DataFrame 写到 MySQL 数据表
myDF.write.jdbc("jdbc:mysql:dbserver", "to_table", connProp)
```

```
scala> val connProp = new java.util.Properties()
     | connProp.put("user", "root")
     | connProp.put("password", "Password123#@!")
     | val myDF = spark.read
     | .jdbc("jdbc:mysql://localhost/hive", "write_to_person", connProp)
val connProp: java.util.Properties = {password=Password123#@!, user=root}
val myDF: org.apache.spark.sql.DataFrame = [id: int, name: string]
```

图 5-30　访问 MySQL 数据源

3. JDBC 数据源选项

Spark SQL 连接 JDBC 数据源，除用户名、密码（SQLite 数据库不需要）等属性之外，还支持分区等选项。常用选项见表 5-3。

表 5-3　JDBC 数据源选项

选　项　名	含　　义	示　　例	读/写
url	JDBC URL，形如： jdbc:subprotocol:subname 可以在 URL 中指定特定的连接属性	jdbc:postgresql://localhost/test?user=fry&password=secret	RW
dbtable	数据表名。读数据时可以是子查询。 不能同时指定 dbtable 和 query		RW
query	将数据读入 Spark 的查询。指定的查询被用作 FROM 子句中的子查询。Spark 为子查询子句分配别名	.option("query", "select c1, c2 from t1")	RW
driver	用于连接的 JDBC 驱动程序		RW
partitionColumn	对表进行分区的列名		R
numPartitions	最大分区数（并行读写） 与 JDBC 最大并发连接数有关		RW
queryTimeout	语句执行超时（秒） 0 表示无限制		RW
fetchsize	每次操作读取的行数		R
batchsize	每次写操作要插入的行数		W
truncate	启用 SaveMode.Overwrite 后，此选项会导致 Spark 截断现有表，而不是删除并重新创建它。这可以防止表元数据（例如，索引）被删除。 由于 DBMS 中 TRUNCATE TABLE 的行为不同，使用它并不总是安全的	.option("truncate", false)	W
createTableOptions	此选项允许在创建表时设置特定于数据库的表和分区选项		W
createTableColumnTypes	创建表时要使用的数据库表列数据类型。类型以与 CREATE TABLE 语法相同的格式指定		W
customSchema	读数据时的自定义模式。列名应与 JDBC 表的相应列名相同		R

5.5 安装关系数据库

5.5.1 PostgreSQL

1. 安装 PostgreSQL Server

直接安装,执行如下命令:

```
sudo apt-get update
sudo apt-get -y install postgresql
```

注意:PostgreSQL 安装后,默认权限认证为 peer,即,使用 Linux 用户登录(认证)。但由于 PostgreSQL 新安装时只创建了 postgres 用户,所以会导致数据库登录失败(与 Linux 系统用户账户不一致)。常用的解决办法有两种:一种是切换 Linux 用户为 postgres(su postgres ...)后登录数据库,创建与 Linux 系统账户同名的数据库角色(用户,create role ...);另一种方法是修改 PostgreSQL 的系统配置。

2. 安装 PostgreSQL Client(pgAdmin 4)

可以手动下载安装包进行安装。首先确认认证 key:

```
# Install the public key for the repository (if not done previously):
wget -c https://www.pgadmin.org/static/packages_pgadmin_org.pub | sudo gpg --dearmor -o /usr/share/keyrings/packages-pgadmin-org.gpg
```

再下载安装包进行安装:

```
# Create the repository configuration file:
sudo sh -c 'echo "deb [signed-by=/usr/share/keyrings/packages-pgadmin-org.gpg] https://ftp.postgresql.org/pub/pgadmin/pgadmin4/apt/$(lsb_release -cs) pgadmin4 main" > /etc/apt/sources.list.d/pgadmin4.list && apt update'
```

pgAdmin 4 安装包的下载地址是:

https://ftp.postgresql.org/pub/pgadmin/pgadmin4/apt/jammy/dists/pgadmin4/main/binary-amd64/pgadmin4-desktop_6.15_amd64.deb

5.5.2 MySQL Server

直接安装,执行如下命令(安装 MySQL Server 8.x):

```
sudo apt-get install mysql-server
```

服务验证:

```
systemctl is-active mysql
```

安全认证:

```
sudo mysql_secure_installation
```

注意:Server 安装后需要设置安全认证策略,设置 root 等数据库用户的密码(见图 5-31)。

```
o@o-VirtualBox:~$ sudo mysql_secure_installation
[sudo] password for o:

Securing the MySQL server deployment.

Connecting to MySQL using a blank password.

VALIDATE PASSWORD COMPONENT can be used to test passwords
and improve security. It checks the strength of password
and allows the users to set only those passwords which are
secure enough. Would you like to setup VALIDATE PASSWORD component?

Press y|Y for Yes, any other key for No: y

There are three levels of password validation policy:

LOW    Length >= 8
MEDIUM Length >= 8, numeric, mixed case, and special characters
STRONG Length >= 8, numeric, mixed case, special characters and dictionary

Please enter 0 = LOW, 1 = MEDIUM and 2 = STRONG: 1
Please set the password for root here.

New password:

Re-enter new password:

Estimated strength of the password: 25
Do you wish to continue with the password provided?(Press y|Y for Yes, any other key
```

图 5-31　设置 MySQL 安全认证策略

更多安装、使用的详细信息，请参考 MySQL 用户手册及相关资料。