1.1 多表关联 join

```
    // 获取到一个 sparksession 对象
 val session = SparkSession.builder()
  .master("local[*]")
  .appName(JoinDemo.getClass.getSimpleName)
  .getOrCreate()
import session.implicits._

// 多表的关联查询
/** 第一个表
  * name age fv address
  *
  * 第二个:
  * address  aname
  * hn  河南省
  *
  */
val data1: Dataset[String] = session.createDataset(List("dd 25 100 hn", "nn 26 100 sd",
"tt 18 100 gz", "lz 30 100 heb", "cangls 40 100 jp"))

val data2: Dataset[String] = session.createDataset(List("hn 河南省", "sd 山东省", "heb
黑龙江省", "jp 日本省"))
// 数据的切分
val splitData1: Dataset[(String, Int, Int, String)] = data1.map({
  t =>
    val lines = t.split(" ")
    val name = lines(0)
    val age = lines(1).toInt
    val fv = lines(2).toInt
    val pro = lines(3)
    (name, age, fv, pro)
})

// 指定 shema
val df1: DataFrame = splitData1.toDF("name", "age", "fv", "pro")

val splitData2: Dataset[(String, String)] = data2.map {
  t =>
    val lines = t.split(" ")
    (lines(0), lines(1))
}
```

```scala
val df2: DataFrame = splitData2.toDF("pro1", "pname")
//    要读两个数据进行join1.2
// SQL 语法
// 注册两张表，然后进行join 关联查询
//    df1.createTempView("t_pinfo")
//    df2.createTempView("t_proinfo")

//    session.sql("select * from t_pinfo inner join t_proinfo on t_pinfo.pro =
t_proinfo.pro ")
//    session.sql("select * from t_pinfo right join t_proinfo on t_pinfo.pro =
t_proinfo.pro ")
//    .show()


// DSL 语法风格
// 先进行join  然后再通过where 指定join 的条件
//    df1.join(df2).where(df1("pro") === df2("pro"))

// 如果有相同的字段，可以直接使用该字段名称
//    df1.join(df2,"pro")

// 默认使用的是inner join  我们可以使用自定的
//    inner`, `cross`, `outer`, `full`, `full_outer`, `left`, `left_outer`,
//    *                `right`, `right_outer`, `left_semi`, `left_anti`.
//      df1.join(df2,df1("pro") === df2("pro1"))

// 利用第3 个参数来指定 join 的类型
df1.join(df2, $"pro" === $"pro1", "left")
  .show()
```

案例：求 ip 地址归属地，利用 sparksql 的多表关联实现

```scala
object SparkSqlJoinIp {
  def main(args: Array[String]): Unit = {
    val session = SparkSession.builder()
      .master("local")
      .appName("xxx")
      .getOrCreate()
    session

    import session.implicits._
```

```
    val ipLines:Dataset[String] = session.read.textFile("data/spark/ip/ip.txt")

    val iprules: DataFrame = ipLines.map({
      line =>
        // 按指定分隔符切分数据
        val lines = line.split("[|]")
        val start = lines(2).toLong
        val end = lines(3).toLong
        val province = lines(6)
        (start, end, province)
    }).toDF("start", "end", "province")
    iprules

    val data = session.read.textFile("data/spark/ip/access.log")
    val af = data.map({
      line =>
        // 获取日志数据中的 ip 地址
        val strings = line.split("[|]")
        val ip = strings(1)
        // 把 ip 地址转换为 10 进制，然后去规则库中进行匹配
        val ipNum: Long = IPUtils.ip2Long(ip)
        // 现在只需要返回 ip 地址的十进制数据
        ipNum
    }).toDF("ipnum")
    af

    iprules.createTempView("rules")
    af.createTempView("access")

//    session.sql("select province,count(*) from rules join  access on (ipnum >= start and ipnum <= end ) group by province")
//    .show()

    import org.apache.spark.sql.functions._
    iprules.join(af,$"ipnum" >= $"start" and $"ipnum" <= $"end").groupBy($"province").count().sort($"count" desc)
//    .show()

    iprules.join(af,$"ipnum" between($"start",$"end")).groupBy($"province").count()
    .show()

    session.close()
  }
}
```

1.3 自定义函数:

UDF （user defined function）

UDF 输入一行，返回一个结果 一对一 ip2Province(123123111) -> 辽宁省

UDF 使用最普遍的

UDTF 输入一行，返回多行（hive）一对多 spark SQL 中没有 UDTF，spark 中用 flatMap 即可实现该功能

user defined aggregate function

UDAF 输入多行，返回一行 aggregate(聚合) count、sum 这些是 sparkSQL 自带的聚合函数，但是复杂的业务，要自己定义

1.3.1 自定义 UDF 函数

```
val ip = spark.read.textFile("data/spark/ip/ip.txt")
val ipds = ip.map(t => {
  val split = t.split("[|]")
  val start = split(2).toLong
  val end = split(3).toLong
  val province = split(6)
  (start, end, province)
})
// 对规则库的数据进行收集，收集到 driver 端，然后再广播出去
val ipRules: Array[(Long, Long, String)] = ipds.collect()
val bc = spark.sparkContext.broadcast(ipRules)


// 自定义一个 udf 并注册
session.udf.register("udf2Province", (ipnum: Long) => {
  // 获取广播变量数据，并根据 ip 地址对应的十进制查找省份名称
```

```
    val rules = ipRuleBC.value
    val index = IPUtils.binarySearch(rules, ipnum)
    var province = "unknown"
    if (index != -1) {
      province = rules(index)._3
    }
    province
})

如果想把数据写入到 mysql 中
val url = "jdbc:mysql://192.168.8.1:3306/scott?characterEncoding=utf-8"
val tname ="access_log"

// 指定用户名和密码
val p = new Properties()
p.setProperty("user","root")
p.setProperty("password","123")
// 驱动不可少
p.setProperty("driver","com.mysql.jdbc.Driver")

// 写到 mysql 中
sql.write.mode(SaveMode.Overwrite).jdbc(url,tname,p)
```

把任务打包提交到集群运行：

提交任务的命令：

```
[root@hdp-03 ~]# spark-submit --master spark://hdp-01:7077 --jars mysql-connector-java-5.1.3
8.jar --class cn.edu360.spark29.day08.Ip2MysqlBC2 spark29-1.0-SNAPSHOT.jar hdfs://hdp-01:900
0/access.log hdfs://hdp-01:9000/ip.txt
```

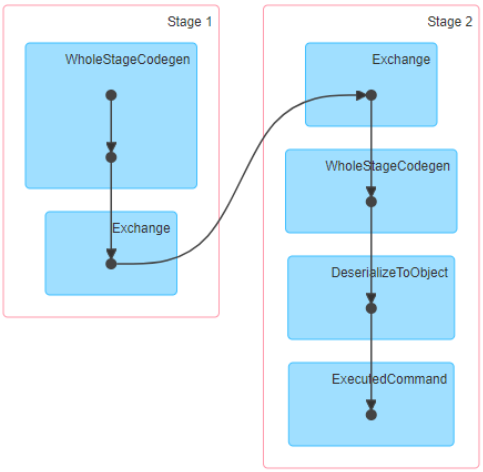
```
ount(*) cnts from v_ip group by province
xception in thread "main" java.sql.SQLException: No suitable driver
        at java.sql.DriverManager.getDriver(DriverManager.java:315)
        at org.apache.spark.sql.execution.datasources.jdbc.JDBCOptions$$anonfun$
ptions.scala:84)
```

该错误是因为：缺少 Driver 的类，需要指定 Driver

```
count(*) cnts from v_ip group by province
Exception in thread "main" org.apache.spark.sql.AnalysisException: Table or view 'access_log
3' already exists. SaveMode: ErrorIfExists.;
        at org.apache.spark.sql.execution.datasources.jdbc.JdbcRelationProvider.createRelati
on(JdbcRelationProvider.scala:81)
        at org.apache.spark.sql.execution.datasources.DataSource.write(DataSource.scala:472)
        at org.apache.spark.sql.execution.datasources.SaveIntoDataSourceCommand.run(SaveInto
```

```
// 执行入库
result.write.mode(SaveMode.Append).jdbc(url,tname,conn)
spark.stop()
```

运行的 DAG 图:



1.3.2自定义 UDAF 函数

简单几何平均数:

$$G_n = \sqrt[n]{\prod_{i=1}^{n} x_i} = \sqrt[n]{x_1 x_2 x_3 \cdots x_n}$$

```
// 自定义一个 udaf 的类
class GeoMean extends UserDefinedAggregateFunction {
  //输入数据的类型
  override def inputSchema: StructType = StructType(List(
    StructField("id", DoubleType)
  ))
```

```scala
//产生中间结果的数据类型
override def bufferSchema: StructType = StructType(List(
  //相乘之后返回的积
  StructField("product", DoubleType),
  //参与运算数字的个数
  StructField("counts", LongType)
))
//最终返回的结果类型
override def dataType: DataType = DoubleType

//确保一致性 一般用 true
override def deterministic: Boolean = true

//指定初始值
override def initialize(buffer: MutableAggregationBuffer): Unit = {
  //相乘的初始值
  buffer(0) = 1.0
  //参与运算数字的个数的初始值
  buffer(1) = 0L
}

//每有一条数据参与运算就更新一下中间结果(update 相当于在每一个分区中的运算)
override def update(buffer: MutableAggregationBuffer, input: Row): Unit = {
  //每有一个数字参与运算就进行相乘（包含中间结果）
  buffer(0) = buffer.getDouble(0) * input.getDouble(0)
  //参与运算数据的个数也有更新
  buffer(1) = buffer.getLong(1) + 1L
}

//全局聚合
override def merge(buffer1: MutableAggregationBuffer, buffer2: Row): Unit = {
  //每个分区计算的结果进行相乘
  buffer1(0) = buffer1.getDouble(0) * buffer2.getDouble(0)
  //每个分区参与预算的中间结果进行相加
  buffer1(1) = buffer1.getLong(1) + buffer2.getLong(1)
}

//计算最终的结果
override def evaluate(buffer: Row): Double = {
  math.pow(buffer.getDouble(0), 1.toDouble / buffer.getLong(1))
}
```

```
//  使用

// 定义值
    val range: Dataset[lang.Long] = session.range(1,11) //  左闭右开
    range.show()
    range.printSchema() // 默认列名是 id  可以用 toDF 自定义
    val gm = new GeoMean
    // 注册函数
    session.udf.register("gm", gm)
    range.createTempView("v_udaf")

    session.sql("select gm(id) from v_udaf")
      //    session.udf.register(GeometricMean)
//        .show()

    import session.implicits._
    range.select("id").agg(gm($"id"))
    .show()
    session.close()
```

1.4 数据源操作

常见的数据源有普通文件，json 文件，parquet 文件，数据库，csv 文件。

1.4.1 普通文件

```
object DataSourceFile {
  def main(args: Array[String]): Unit = {
    val session = SparkSession.builder()
      .master("local")
      .appName(IPsql.getClass.getSimpleName)
      .getOrCreate()

    import session.implicits._
    val file: Dataset[String] = session.read.textFile("ip.txt")

    file.map({
      t =>
```

```
            val lines = t.split("[|]")
            val start = lines(2)
            val end = lines(3)
            val province = lines(6)
            (start,end,province)
        }).show()
    // text 方式保存文件只能有一列，相当于是保存字符串
    file2.write.text("xx2")


import session.implicits._
// text 方法读取到的数据类型是 DataFrame
val file2: DataFrame = session.read.text("ip.txt")
println(file2.printSchema())

val ds= file2.map({
  t =>
    val data: String = t.getString(0)
    println(data)
    val lines = data.split("[|]")
    lines
})
ds.show()
  }
}
```

text 写文件，只支持一列

```
18/03/10 14:56:40 INFO FileSourceScanExec: Pushed Filters:
Exception in thread "main" org.apache.spark.sql.AnalysisException: Text data source supports only a single column, and you have 2 columns.;
    at org.apache.spark.sql.execution.datasources.text.TextFileFormat.verifySchema(TextFileFormat.scala:46)
    at org.apache.spark.sql.execution.datasources.text.TextFileFormat.prepareWrite(TextFileFormat.scala:66)
    at org.apache.spark.sql.execution.datasources.FileFormatWriter$.write(FileFormatWriter.scala:142)
    at org.apache.spark.sql.execution.datasources.InsertIntoHadoopFsRelationCommand.run(InsertIntoHadoopFsRelationCommand.scala:145)
```

1.4.2 JDBC 数据源：

```
def main(args: Array[String]): Unit = {
  val session: SparkSession = SparkSession.builder()
    .master("local[*]")
    .appName(DataSourceDemo1.getClass.getName)
    .getOrCreate()
  //    import session.implicits._
```

```
        // 连接 mysql 数据库 设置参数 url driver dbtable user password
        val empData: DataFrame = session.read.format("jdbc").options(
          Map("url" -> "jdbc:mysql://localhost:3306/scott?characterEncoding=utf-8",
            "driver" -> "com.mysql.jdbc.Driver",
            "dbtable" -> "emp",
            "user" -> "root",
            "password" -> "123")).load()

    val filtered: Dataset[Row] = empData.where("sal > 1600")
    val props = new Properties()
    props.put("user","root")
    props.put("password","123")
    props.put("driver", "com.mysql.jdbc.Driver")

    filtered.write.mode("append").jdbc("jdbc:mysql://localhost:3306/scott","emp2",props)
    empData.show()

        // 连接 mysql 数据库方法二
       import session.implicits._
      val url = "jdbc:mysql://localhost:3306/scott?characterEncoding=utf-8"
      val table ="salgrade"
      val p = new Properties()
      p.setProperty("user","root")
      p.setProperty("password","123")
      p.setProperty("driver"," com.mysql.jdbc.Driver ")
      val result = session.read.jdbc(url,table,p)
        result.where($"losal" > 1000 and $"hisal" < 3000)

//       .show()
      // 不能直接保存为文本文件，除非单列的字符串类型
//     filtered.write.text("data/t2") // Text data source supports only a single column, and you have 3 columns.;
        filter.write.jdbc(url,"emp3",pro)


        // 连接 mysql 数据库方法三
      session.read.format("jdbc").jdbc(url,table,p)


    session.close()
}
```

操作 mysql 时，需要导入 jar 包，maven 工程需要设置 maven 依赖

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.38</version>
</dependency>
```

1.4.3 json 数据源：

```
  def main(args: Array[String]): Unit = {
    val session: SparkSession = SparkSession.builder()
      .master("local[*]")
      .appName(DataSourceDemo1.getClass.getName)
      .getOrCreate()

    import session.implicits._
    // 读普通文本，textFile 和 text 方法都可以，只是返回值类型不同
    //    val lines: Dataset[String] = session.read.textFile("person.txt")
    val lines: DataFrame = session.read.text("person.txt")
//    lines.show()
    // 读取 json 数据
    val json: DataFrame = session.read.json("test.json")
    val res: Dataset[Row] = json.select("name","age").where("age > 13")
//    res.show()

    // 会报错，因为数据是 json 格式的，有两列，而普通文本文件只支持一列
//    res.write.text("resjson.txt")

    // 如果用普通文件的方式去读取 json 格式的数据，会把所有列当成一行，
//    session.read.text("test.json").show()

    // 所以 json 数据格式需要使用 json 方式写
//    res.write.json("jsonout")

    // 还可以使用 mode
//    res.write.mode(SaveMode.Overwrite)

    res.write.mode("overwrite").json("jsonout3")

import session.implicits._
val cd = session.createDataset(List("abc 25 99"))
val map = cd.map({
```

```
  t =>
    val s = t.split(" ")
    (s(0),s(1),s(2))
}).toDF("name","age","fv")
  map.write.mode(SaveMode.Overwrite).json("jsonout1")

    session.close()
  }

// 指定 json 数据的 schema 信息时，需要注意的时候，必须按照 schema 的顺序来修改，而不是按照原始的数据的顺序

json.toDF("p1","p2","a1").show()
```

嵌套 json 操作：

```
  // session 实例
    val session: SparkSession = SparkSession.builder()
      .master("local[*]")
      .appName(this.getClass.getSimpleName)
      .getOrCreate()

    // 读取普通文件的 2 个 API
    val json: DataFrame = session.read.json("jsonlog2.json")
    json.printSchema()
    json.show()
    //DSL
    json.select("address.province").show()
    json.createTempView("v_tmp")
    // 使用 sql 语法查询
//    session.sql("select address.city from  v_tmp").show()
    session.close()
```

嵌套 json 可参考：https://blog.csdn.net/qq_21439395/article/details/80710180

1.4.4 csv 数据源：

csv 数据格式，默认是使用，分割符，默认可以使用 excel 打开。

// csv 是机器学习中的一种重要数据格式，可直接使用 excel 打开 csv 格式的文件

```
    // 使用 csv 方式读取数据，没有了 json 读的 name，和 age 列名，而是叫做_c0,_c1 这两列
```

```
 val csv: DataFrame = session.read.csv("csvlog.csv")
 // 使用 toDF 指定新的 schema 信息
val csv1: DataFrame = csv.toDF("name", "age")
 val csvDF = csv.withColumnRenamed("_c0","name").withColumnRenamed("_c1","age")
 val csvRes: Dataset[Row] = csvDF.select("name","age").where("age > 14")
 csvRes.show()
csvRes.write.mode("overwrite")csv("data/csv2")
```

1.4.5 parquet 数据源

```
// 推荐直接把结果数据写入到 parquet 文件中,parquet 文件的数据，不能直接用 text 查看，因为有一些校验文件
csvRes.write.mode("overwrite").parquet("parquetlog")
// 读取 parquet 格式的文件
session.read.parquet("parquetlog").show()
```

Parquet 文件

Apache Parquet 最初的设计动机是存储嵌套式数据，比如 Protocolbuffer，thrift，json 等，将这类数据存储成**列式**格式，以方便对其高效压缩和编码，且使用更少的 IO 操作取出需要的数据

parquet 数据格式的优势？？

列式存储，使用时只需要读取需要的列，支持向量运算，能获得更好的扫描行。

压缩编码可以降低磁盘存储空间，由于同一列的数据类型是一样的，可以使用不同的压缩编码。

可以跳过不符合条件的数据，只读取需要的数据，降低 IO 的数据量

通用的，适配多种计算框架，查询引擎（hive,impala,pig 等），计算框架（mapreduce,spark 等），数据模型（avro,thrift,json 等）

压缩比：12g -→ 1g

1tb 1024Gb 100GB

日志数据，可能就是 parquet 文件格式

缺点：

比如它不支持 update 操作（数据写成后不可修改）。

数据源总结：
根据需求来读取指定的数据格式，
如果要写文件，具体写哪种格式的数据，取决于需求。

日志大小的补充：

```
// 一条日志数据的大小 0.5 KB

// 一天的数据量有多少 （GB，记录，多少条日志）
//    800G
// 用户数量    *  每一个用户大致每一天产生的日志数量 10 条   =  条记录   *  0.5Kb    ====》
1000GB

// 结合具体的业务   parqut  12 : 1   log
```

1.5 Save Modes （保存模式）

Save operations （保存操作）可以选择使用 SaveMode，它指定如何处理现有数据如果存在的话. 重要的是要意识到, 这些 save modes （保存模式）不使用任何 locking （锁定）并且不是 atomic （原子）. 另外, 当执行 Overwrite 时, 数据将在新数据写出之前被删除.

Scala/Java	Any Language	Meaning
SaveMode.ErrorIfExists(default)	"error"(default)	将 DataFrame 保存到 data source （数据源）时, 如果数据已经存在, 则会抛出异常.

SaveMode.Append	"append"	将 DataFrame 保存到 data source （数据源）时，如果 data/table 已存在，则 DataFrame 的内容将被 append （附加）到现有数据中.
SaveMode.Overwrite	"overwrite"	Overwrite mode （覆盖模式）意味着将 DataFrame 保存到 data source （数据源）时，如果 data/table 已经存在，则预期 DataFrame 的内容将 overwritten （覆盖）现有数据.
SaveMode.Ignore	"ignore"	Ignore mode （忽略模式）意味着当将 DataFrame 保存到 data source （数据源）时，如果数据已经存在，则保存操作预期不会保存 DataFrame 的内容，并且不更改现有数据. 这与 SQL 中的 CREATE TABLE IF NOT EXISTS 类似.

1.6 分组 topK

```
object TopK {

  val topK = 2

  def main(args: Array[String]): Unit = {
    val session = SparkSession.builder()
      .master("local")
      .appName(IPsql.getClass.getSimpleName)
      .getOrCreate()

    import session.implicits._

    val file = session.read.textFile("teacher.log")
    val st: DataFrame = file.map({
```

```
        t =>
          val index = t.lastIndexOf("/")
          // 截取字符串
          val tName = t.substring(index + 1)
          // 封装数据为 URL，然后获取 Host 内容
          val uri = new URL(t.substring(0, index))
          val host = uri.getHost

          val hostArray = host.split("[.]") // 要对特殊的字符进行转义
        // 获取学科名称
        val sub = hostArray(0)
          // 返回元组
          (sub, tName)
      }).toDF("subject", "teacher")

      st.createTempView("t_sub_teacher")

      // 该学科下的老师的访问次数
      val sql = session.sql("select subject,teacher,count(*) cnts from t_sub_teacher group By subject,teacher ")
//order by cnts desc
      //      sql
      //      .show()

      // 全局 topK
      //      sql.limit(3).show()
      sql.createTempView("v_tmp")

      // 求分组 topK   分学科的老师 排序
      //      val groupedTop = session.sql("select subject,teacher,cnts,row_number() over(partition by subject order
by cnts desc) sub_order from v_tmp    order by cnts desc")

      // 分学科的排序 取 topK
      //      val groupedTop = session.sql(s"select * from (select subject,teacher,cnts,
      // row_number() over(partition by subject order by cnts desc) sub_order
      // from v_tmp order by cnts desc ) where sub_order <= $topK")

      //  分学科的排序   + 全局排序
      //      val groupedTop = session.sql("select subject,teacher,cnts,
      // row_number() over(partition by subject order by cnts desc) sub_order,
      // row_number() over(order by cnts desc) g_order
      // from v_tmp    order by cnts desc")

      // 分学科的 TopK 排序 + 全局排序   rank 支持并列排序
//      val groupedTop = session.sql("select * from (select subject,teacher,cnts," +
```

```
//         "row_number() over(partition by subject order by cnts desc) sub_order," +
//         "rank() over(order by cnts desc) g_order " +
//        s"from v_tmp     ) where sub_order <=$topK")


    // 全局排序，分学科 topk 排序，选中的全局排序（可使用 row_number over()   rank() over ()   dense_rank())
    val groupedTop = session.sql("select * ,dense_rank() over(order by cnts desc) choose_num from (select subject,teacher,cnts," +
      "row_number() over(partition by subject order by cnts desc) sub_order," +
      "rank() over(order by cnts desc) g_order " +
      s"from v_tmp) where sub_order <=$topK")

    groupedTop.show()

    // 释放资源
    session.close()
  }
}
```