# Spark day05

1. Shark

Shark是基于Spark计算框架之上且兼容Hive语法的SQL执行引擎，由于底层的计算采用了Spark，性能比MapReduce的Hive普遍快2倍以上，当数据全部load在内存的话，将快10倍以上，因此Shark可以作为交互式查询应用服务来使用。除了基于Spark的特性外，Shark是完全兼容Hive的语法，表结构以及UDF函数等，已有的HiveSql可以直接进行迁移至Shark上Shark底层依赖于Hive的解析器，查询优化器，但正是由于SHark的整体设计架构对Hive的依赖性太强，难以支持其长远发展，比如不能和Spark的其他组件进行很好的集成，无法满足Spark的一栈式解决大数据处理的需求。

1. SparkSQL
2. SparkSQL介绍

Hive是Shark的前身，Shark是SparkSQL的前身,SparkSQL产生的根本原因是其完全脱离了Hive的限制。

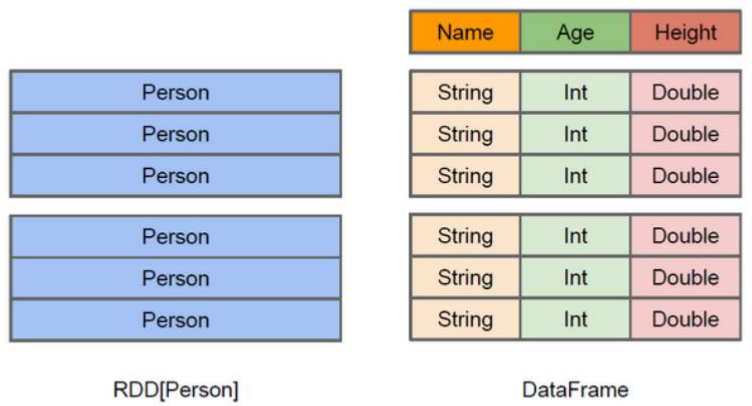
* SparkSQL支持查询原生的RDD。 RDD是Spark平台的核心概念，是Spark能够高效的处理大数据的各种场景的基础。
* 能够在Scala中写SQL语句。支持简单的SQL语法检查，能够在Scala中写Hive语句访问Hive数据，并将结果取回作为RDD使用。

1. Spark on Hive和Hive on Spark

Spark on Hive： Hive只作为储存角色，Spark负责sql解析优化，执行。

Hive on Spark：Hive即作为存储又负责sql的解析优化，Spark负责执行。

1. DataFrame

DataFrame也是一个分布式数据容器。与RDD类似，然而DataFrame更像传统数据库的二维表格，除了数据以外，还掌握数据的结构信息，即schema。同时，与Hive类似，DataFrame也支持嵌套数据类型（struct、array和map）。从API易用性的角度上 看， DataFrame API提供的是一套高层的关系操作，比函数式的RDD API要更加友好，门槛更低。

DataFrame的底层封装的是RDD，只不过RDD的泛型是Row类型。

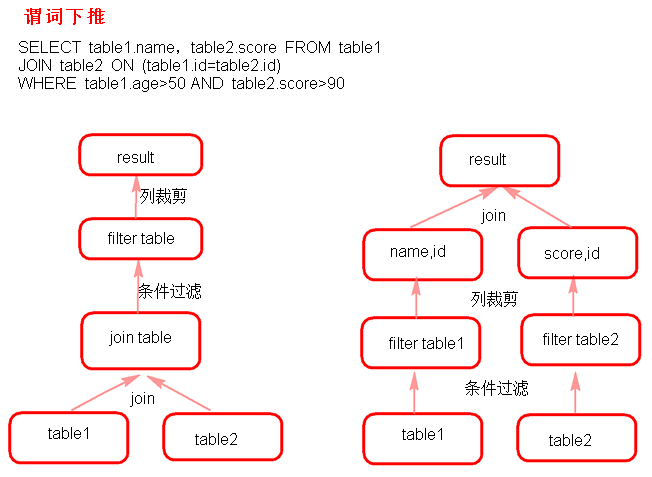
1. SparkSQL的数据源

SparkSQL的数据源可以是JSON类型的字符串，JDBC,Parquent,Hive，HDFS等。

1. SparkSQL底层架构

首先拿到sql后解析一批未被解决的逻辑计划，再经过分析得到分析后的逻辑计划，再经过一批优化规则转换成一批最佳优化的逻辑计划，再经过SparkPlanner的策略转化成一批物理计划，随后经过消费模型转换成一个个的Spark任务执行。

1. 谓词下推（predicate Pushdown）



1. 创建DataFrame的几种方式
2. 读取json格式的文件创建DataFrame

注意：

* json文件中的json数据不能嵌套json格式数据。
* DataFrame是一个一个Row类型的RDD，df.rdd()/df.javaRdd()。
* 可以两种方式读取json格式的文件。
* df.show()默认显示前20行数据。
* DataFrame原生API可以操作DataFrame（不方便）。
* 注册成临时表时，表中的列默认按ascii顺序显示列。

java:

|  |
| --- |
| SparkConf conf = **new** SparkConf();  conf.setMaster("local").setAppName("jsonfile");  SparkContext sc = **new** SparkContext(conf);    //创建sqlContext  SQLContext sqlContext = **new** SQLContext(sc);    /\*\*  \* DataFrame的底层是一个一个的RDD RDD的泛型是Row类型。  \* 以下两种方式都可以读取json格式的文件  \*/  **DataFrame df = sqlContext.read().format("json").load("sparksql/json");**  **// DataFrame df2 = sqlContext.read().json("sparksql/json.txt");**  // df2.show();  /\*\*  \* DataFrame转换成RDD  \*/  **RDD<Row> rdd = df.rdd();**  /\*\*  \* 显示 DataFrame中的内容，默认显示前20行。如果现实多行要指定多少行show(行数)  \* 注意：当有多个列时，显示的列先后顺序是按列的ascii码先后显示。  \*/  // df.show();  /\*\*  \* 树形的形式显示schema信息  \*/  **df.printSchema();**    /\*\*  \* dataFram自带的API 操作DataFrame  \*/  //select name from table  // df.select("name").show();  //select name age+10 as addage from table  df.select(df.col("name"),df.col("age").plus(10).alias("addage")).show();  //select name ,age from table where age>19  df.select(df.col("name"),df.col("age")).where(df.col("age").gt(19)).show();  //select count(\*) from table group by age  df.groupBy(df.col("age")).count().show();    /\*\*  \* 将DataFrame注册成临时的一张表，这张表临时注册到内存中，是逻辑上的表，不会雾化到磁盘  \*/  **df.registerTempTable("jtable");**    DataFrame sql = sqlContext.sql("select age,count(1) from jtable group by age");  DataFrame sql2 = sqlContext.sql("select \* from jtable");    sc.stop(); |

scala:

|  |
| --- |
| **val** conf = **new** SparkConf()  conf.setMaster("local").setAppName("jsonfile")  **val** sc = **new** SparkContext(conf)  **val** sqlContext = **new** SQLContext(sc)  **val df = sqlContext.read.json("sparksql/json")**  **//val df1 = sqlContext.read.format("json").load("sparksql/json")**  **df.show()**  **df.printSchema()**  //select \* from table  df.select(df.col("name")).show()  //select name from table where age>19  df.select(df.col("name"),df.col("age")).where(df.col("age").gt(19)).show()  //select count(\*) from table group by age  df.groupBy(df.col("age")).count().show();    /\*\*  \* 注册临时表  \*/  **df.registerTempTable("jtable")**  **val** result = sqlContext.sql("select \* from jtable")  result.show()  sc.stop() |

1. 通过json格式的RDD创建DataFrame

java:

|  |
| --- |
| SparkConf conf = **new** SparkConf();  conf.setMaster("local").setAppName("jsonRDD");  JavaSparkContext sc = **new** JavaSparkContext(conf);  SQLContext sqlContext = **new** SQLContext(sc);  JavaRDD<String> nameRDD = sc.parallelize(Arrays.*asList*(  "{\"name\":\"zhangsan\",\"age\":\"18\"}",  "{\"name\":\"lisi\",\"age\":\"19\"}",  "{\"name\":\"wangwu\",\"age\":\"20\"}"  ));  JavaRDD<String> scoreRDD = sc.parallelize(Arrays.*asList*(  "{\"name\":\"zhangsan\",\"score\":\"100\"}",  "{\"name\":\"lisi\",\"score\":\"200\"}",  "{\"name\":\"wangwu\",\"score\":\"300\"}"  ));  **DataFrame namedf = sqlContext.read().json(nameRDD);**  **DataFrame scoredf = sqlContext.read().json(scoreRDD);**  namedf.registerTempTable("name");  scoredf.registerTempTable("score");  DataFrame result = sqlContext.sql("select name.name,name.age,score.score from name,score where name.name = score.name");  result.show();  sc.stop(); |

scala:

|  |
| --- |
| **val** conf = **new** SparkConf()  conf.setMaster("local").setAppName("jsonrdd")  **val** sc = **new** SparkContext(conf)  **val** sqlContext = **new** SQLContext(sc)  **val** nameRDD = sc.makeRDD(Array(  "{\"name\":\"zhangsan\",\"age\":18}",  "{\"name\":\"lisi\",\"age\":19}",  "{\"name\":\"wangwu\",\"age\":20}"  ))  **val** scoreRDD = sc.makeRDD(Array(  "{\"name\":\"zhangsan\",\"score\":100}",  "{\"name\":\"lisi\",\"score\":200}",  "{\"name\":\"wangwu\",\"score\":300}"  ))  **val nameDF = sqlContext.read.json(nameRDD)**  **val scoreDF = sqlContext.read.json(scoreRDD)**  nameDF.registerTempTable("name")  scoreDF.registerTempTable("score")  **val** result = sqlContext.sql("select name.name,name.age,score.score from name,score where name.name = score.name")  result.show()  sc.stop() |

1. 非json格式的RDD创建DataFrame
2. 通过反射的方式将非json格式的RDD转换成DataFrame（不建议使用）

* 自定义类要可序列化
* 自定义类的访问级别是Public
* RDD转成DataFrame后会根据映射将字段按Assci码排序
* 将DataFrame转换成RDD时获取字段两种方式,一种是df.getInt(0)下标获取（不推荐使用），另一种是df.getAs(“列名”)获取（推荐使用）

|  |
| --- |
| /\*\*  \* 注意：  \* 1.自定义类必须是可序列化的  \* 2.自定义类访问级别必须是Public  \* 3.RDD转成DataFrame会把自定义类中字段的名称按assci码排序  \*/  SparkConf conf = **new** SparkConf();  conf.setMaster("local").setAppName("RDD");  JavaSparkContext sc = **new** JavaSparkContext(conf);  SQLContext sqlContext = **new** SQLContext(sc);  JavaRDD<String> lineRDD = sc.textFile("sparksql/person.txt");  JavaRDD<Person> personRDD = lineRDD.map(**new** Function<String, Person>() {  /\*\*  \*  \*/  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  @Override  **public** Person call(String s) **throws** Exception {  Person p = **new** Person();  p.setId(s.split(",")[0]);  p.setName(s.split(",")[1]);  p.setAge(Integer.*valueOf*(s.split(",")[2]));  **return** p;  }  });  /\*\*  \* 传入进去Person.class的时候，sqlContext是通过反射的方式创建DataFrame  \* 在底层通过反射的方式获得Person的所有field，结合RDD本身，就生成了DataFrame  \*/  DataFrame df = sqlContext.createDataFrame(personRDD, Person.**class**);  df.show();  df.registerTempTable("person");  sqlContext.sql("select name from person where id = 2").show();  /\*\*  \* 将DataFrame转成JavaRDD  \* 注意：  \* 1.可以使用row.getInt(0),row.getString(1)...通过下标获取返回Row类型的数据，但是要注意列顺序问题---不常用  \* 2.可以使用row.getAs("列名")来获取对应的列值。  \*  \*/  JavaRDD<Row> javaRDD = df.javaRDD();  JavaRDD<Person> map = javaRDD.map(**new** Function<Row, Person>() {  /\*\*  \*  \*/  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  @Override  **public** Person call(Row row) **throws** Exception {  Person p = **new** Person();  //p.setId(row.getString(1));  //p.setName(row.getString(2));  //p.setAge(row.getInt(0));  p.setId((String)row.getAs("id"));  p.setName((String)row.getAs("name"));  p.setAge((Integer)row.getAs("age"));  **return** p;  }  });  map.foreach(**new** VoidFunction<Person>() {    /\*\*  \*  \*/  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  @Override  **public** **void** call(Person t) **throws** Exception {  System.***out***.println(t);  }  });  sc.stop(); |

scala:

|  |
| --- |
| **val** conf = **new** SparkConf()  conf.setMaster("local").setAppName("rddreflect")  **val** sc = **new** SparkContext(conf)  **val** sqlContext = **new** SQLContext(sc)  **val** lineRDD = sc.textFile("./sparksql/person.txt")  /\*\*  \* 将RDD隐式转换成DataFrame  \*/  **import sqlContext.implicits.\_**  **val** personRDD = lineRDD.map { x => {  **val** person = **Person**(x.split(",")(0),x.split(",")(1),Integer.valueOf(x.split(",")(2)))  person  } }  **val df = personRDD.toDF();**  df.show()  /\*\*  \* 将DataFrame转换成PersonRDD  \*/  **val** rdd = df.rdd  **val** result = rdd.map { x => {  **Person**(x.getAs("id"),x.getAs("name"),x.getAs("age"))  } }  result.foreach { println}  sc.stop() |

1. 动态创建Schema将非json格式的RDD转换成DataFrame

java：

|  |
| --- |
| SparkConf conf = **new** SparkConf();  conf.setMaster("local").setAppName("rddStruct");  JavaSparkContext sc = **new** JavaSparkContext(conf);  SQLContext sqlContext = **new** SQLContext(sc);  JavaRDD<String> lineRDD = sc.textFile("./sparksql/person.txt");  /\*\*  \* 转换成Row类型的RDD  \*/  JavaRDD<Row> rowRDD = lineRDD.map(**new** Function<String, Row>() {  /\*\*  \*  \*/  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  @Override  **public** Row call(String s) **throws** Exception {  **return** RowFactory.*create*(  String.*valueOf*(s.split(",")[0]),  String.*valueOf*(s.split(",")[1]),  Integer.*valueOf*(s.split(",")[2])  );  }  });  /\*\*  \* 动态构建DataFrame中的元数据，一般来说这里的字段可以来源自字符串，也可以来源于外部数据库  \*/  List<StructField> asList =Arrays.*asList*(  DataTypes.*createStructField*("id", DataTypes.***StringType***, **true**),  DataTypes.*createStructField*("name", DataTypes.***StringType***, **true**),  DataTypes.*createStructField*("age", DataTypes.***IntegerType***, **true**)  );  StructType schema = DataTypes.*createStructType*(asList);  DataFrame df = sqlContext.createDataFrame(rowRDD, schema);  df.show();  sc.stop(); |

scala：

|  |
| --- |
| **val** conf = **new** SparkConf()  conf.setMaster("local").setAppName("rddStruct")  **val** sc = **new** SparkContext(conf)  **val** sqlContext = **new** SQLContext(sc)  **val** lineRDD = sc.textFile("./sparksql/person.txt")  **val** rowRDD = lineRDD.map { x => {  **val** split = x.split(",")  RowFactory.create(split(0),split(1),Integer.valueOf(split(2)))  } }  **val** schema = **StructType**(List(  **StructField**("id",**StringType**,**true**),  **StructField**("name",**StringType**,**true**),  **StructField**("age",**IntegerType**,**true**)  ))  **val** df = sqlContext.createDataFrame(rowRDD, schema)  df.show()  df.printSchema()  sc.stop() |

1. 读取parquet文件创建DataFrame

注意：

* 可以将DataFrame存储成parquet文件。保存成parquet文件的方式有两种

|  |
| --- |
| **df.write().mode(SaveMode.*Overwrite*)format("parquet")**  **.save("./sparksql/parquet");**  **df.write().mode(SaveMode.*Overwrite*).parquet("./sparksql/parquet");** |

* SaveMode指定文件保存时的模式。

Overwrite：覆盖

Append：追加

ErrorIfExists：如果存在就报错

Ignore：如果存在就忽略

java:

|  |
| --- |
| SparkConf conf = **new** SparkConf();  conf.setMaster("local").setAppName("parquet");  JavaSparkContext sc = **new** JavaSparkContext(conf);  SQLContext sqlContext = **new** SQLContext(sc);  JavaRDD<String> jsonRDD = sc.textFile("sparksql/json");  DataFrame df = sqlContext.read().json(jsonRDD);  /\*\*  \* 将DataFrame保存成parquet文件，SaveMode指定存储文件时的保存模式  \* 保存成parquet文件有以下两种方式：  \*/  df.write().mode(SaveMode.***Overwrite***).format("parquet").save("./sparksql/parquet");  df.write().mode(SaveMode.***Overwrite***).parquet("./sparksql/parquet");  df.show();  /\*\*  \* 加载parquet文件成DataFrame  \* 加载parquet文件有以下两种方式：  \*/  DataFrame load = sqlContext.read().format("parquet").load("./sparksql/parquet");  load = sqlContext.read().parquet("./sparksql/parquet");  load.show();  sc.stop(); |

scala：

|  |
| --- |
| **val** conf = **new** SparkConf()  conf.setMaster("local").setAppName("parquet")  **val** sc = **new** SparkContext(conf)  **val** sqlContext = **new** SQLContext(sc)  **val** jsonRDD = sc.textFile("sparksql/json")  **val** df = sqlContext.read.json(jsonRDD)  df.show()  /\*\*  \* 将DF保存为parquet文件  \*/  df.write.mode(SaveMode.Overwrite).format("parquet").save("./sparksql/parquet")  df.write.mode(SaveMode.Overwrite).parquet("./sparksql/parquet")  /\*\*  \* 读取parquet文件  \*/  **var** result = sqlContext.read.parquet("./sparksql/parquet")  result = sqlContext.read.format("parquet").load("./sparksql/parquet")  result.show()  sc.stop() |

1. 读取JDBC中的数据创建DataFrame(MySql为例)

两种方式创建DataFrame

java:

|  |
| --- |
| SparkConf conf = **new** SparkConf();  conf.setMaster("local").setAppName("mysql");  JavaSparkContext sc = **new** JavaSparkContext(conf);  SQLContext sqlContext = **new** SQLContext(sc);  /\*\*  \* 第一种方式读取MySql数据库表，加载为DataFrame  \*/  **Map<String, String> options = new HashMap<String,String>();**  **options.put("url", "jdbc:mysql://192.168.179.4:3306/spark");**  **options.put("driver", "com.mysql.jdbc.Driver");**  **options.put("user", "root");**  **options.put("password", "123456");**  **options.put("dbtable", "person");**  **DataFrame person = sqlContext.read().format("jdbc").options(options).load();**  person.show();  person.registerTempTable("person");  /\*\*  \* 第二种方式读取MySql数据表加载为DataFrame  \*/  **DataFrameReader reader = sqlContext.read().format("jdbc");**  **reader.option("url", "jdbc:mysql://192.168.179.4:3306/spark");**  **reader.option("driver", "com.mysql.jdbc.Driver");**  **reader.option("user", "root");**  **reader.option("password", "123456");**  **reader.option("dbtable", "score");**  **DataFrame score = reader.load();**  score.show();  score.registerTempTable("score");  DataFrame result =  sqlContext.sql("select person.id,person.name,score.score from person,score where person.name = score.name");  result.show();  /\*\*  \* 将DataFrame结果保存到Mysql中  \*/  Properties properties = **new** Properties();  properties.setProperty("user", "root");  properties.setProperty("password", "123456");  result.write().mode(SaveMode.***Overwrite***).jdbc("jdbc:mysql://192.168.179.4:3306/spark", "result", properties);  sc.stop(); |

scala:

|  |
| --- |
| **val** conf = **new** SparkConf()  conf.setMaster("local").setAppName("mysql")  **val** sc = **new** SparkContext(conf)  **val** sqlContext = **new** SQLContext(sc)  /\*\*  \* 第一种方式读取Mysql数据库表创建DF  \*/  **val options = new HashMap[*String*,*String*]();**  **options.put("url", "jdbc:mysql://192.168.179.4:3306/spark")**  **options.put("driver","com.mysql.jdbc.Driver")**  **options.put("user","root")**  **options.put("password", "123456")**  **options.put("dbtable","person")**  **val person = sqlContext.read.format("jdbc").options(options).load()**  person.show()  person.registerTempTable("person")  /\*\*  \* 第二种方式读取Mysql数据库表创建DF  \*/  **val reader = sqlContext.read.format("jdbc")**  **reader.option("url", "jdbc:mysql://192.168.179.4:3306/spark")**  **reader.option("driver","com.mysql.jdbc.Driver")**  **reader.option("user","root")**  **reader.option("password","123456")**  **reader.option("dbtable", "score")**  **val score = reader.load()**  score.show()  score.registerTempTable("score")  **val** result = sqlContext.sql("select person.id,person.name,score.score from person,score where person.name = score.name")  result.show()  /\*\*  \* 将数据写入到Mysql表中  \*/  **val** properties = **new** Properties()  properties.setProperty("user", "root")  properties.setProperty("password", "123456")  result.write.mode(SaveMode.Append).jdbc("jdbc:mysql://192.168.179.4:3306/spark", "result", properties)  sc.stop() |

1. 读取Hive中的数据加载成DataFrame

* HiveContext是SQLContext的子类，连接Hive建议使用HiveContext。
* 由于本地没有Hive环境，要提交到集群运行，提交命令：

|  |
| --- |
| ./spark-submit  --master spark://node1:7077,node2:7077  --executor-cores 1  --executor-memory 2G  --total-executor-cores 1  --class com.bjsxt.sparksql.dataframe.CreateDFFromHive  /root/test/HiveTest.jar |

java:

|  |
| --- |
| SparkConf conf = **new** SparkConf();  conf.setAppName("hive");  JavaSparkContext sc = **new** JavaSparkContext(conf);  //HiveContext是SQLContext的子类。  HiveContext hiveContext = **new** HiveContext(sc);  hiveContext.sql("USE spark");  hiveContext.sql("DROP TABLE IF EXISTS student\_infos");  //在hive中创建student\_infos表  hiveContext.sql("CREATE TABLE IF NOT EXISTS student\_infos (name STRING,age INT) row format delimited fields terminated by '\t' ");  hiveContext.sql("load data local inpath '/root/test/student\_infos' into table student\_infos");  hiveContext.sql("DROP TABLE IF EXISTS student\_scores");  hiveContext.sql("CREATE TABLE IF NOT EXISTS student\_scores (name STRING, score INT) row format delimited fields terminated by '\t'");  hiveContext.sql("LOAD DATA "  + "LOCAL INPATH '/root/test/student\_scores'"  + "INTO TABLE student\_scores");  /\*\*  \* 查询表生成DataFrame  \*/  DataFrame goodStudentsDF = hiveContext.sql("SELECT si.name, si.age, ss.score "  + "FROM student\_infos si "  + "JOIN student\_scores ss "  + "ON si.name=ss.name "  + "WHERE ss.score>=80");  hiveContext.sql("DROP TABLE IF EXISTS good\_student\_infos");  goodStudentsDF.registerTempTable("goodstudent");  DataFrame result = hiveContext.sql("select \* from goodstudent");  result.show();  /\*\*  \* 将结果保存到hive表 good\_student\_infos  \*/  **goodStudentsDF.write().mode(SaveMode.*Overwrite*).saveAsTable("good\_student\_infos");**  Row[] goodStudentRows = hiveContext.table("good\_student\_infos").collect();  **for**(Row goodStudentRow : goodStudentRows) {  System.***out***.println(goodStudentRow);  }  sc.stop(); |

scala:

|  |
| --- |
| **val** conf = **new** SparkConf()  conf.setAppName("HiveSource")  **val** sc = **new** SparkContext(conf)  /\*\*  \* HiveContext是SQLContext的子类。  \*/  **val** hiveContext = **new** HiveContext(sc)  hiveContext.sql("use spark")  hiveContext.sql("drop table if exists student\_infos")  hiveContext.sql("create table if not exists student\_infos (name string,age int) row format delimited fields terminated by '\t'")  hiveContext.sql("load data local inpath '/root/test/student\_infos' into table student\_infos")    hiveContext.sql("drop table if exists student\_scores")  hiveContext.sql("create table if not exists student\_scores (name string,score int) row format delimited fields terminated by '\t'")  hiveContext.sql("load data local inpath '/root/test/student\_scores' into table student\_scores")    **val** df = hiveContext.sql("select si.name,si.age,ss.score from student\_infos si,student\_scores ss where si.name = ss.name")  hiveContext.sql("drop table if exists good\_student\_infos")  /\*\*  \* 将结果写入到hive表中  \*/  df.write.mode(SaveMode.Overwrite).saveAsTable("good\_student\_infos")    sc.stop() |

1. Spark On Hive的配置
2. 在Spark客户端配置Hive On Spark

在Spark客户端安装包下spark-1.6.0/conf中创建文件hive-site.xml：

配置hive的metastore路径

|  |
| --- |
| **<configuration>**  **<property>**  **<name>hive.metastore.uris</name>**  **<value>thrift://node1:9083</value>**  **</property>**  **</configuration>** |

1. 启动Hive的metastore服务

|  |
| --- |
| hive --service metastore |

1. 启动zookeeper集群，启动HDFS集群。
2. 启动SparkShell 读取Hive中的表总数，对比hive中查询同一表查询总数测试时间。

|  |
| --- |
| ./spark-shell  --master spark://node1:7077,node2:7077  --executor-cores 1  --executor-memory 1g  --total-executor-cores 1  import org.apache.spark.sql.hive.HiveContext  val hc = new HiveContext(sc)  hc.sql("show databases").show  hc.sql("user default").show  hc.sql("select count(\*) from jizhan").show |

* 注意：

如果使用Spark on Hive 查询数据时，出现错误：



找不到HDFS集群路径，要在客户端机器conf/spark-env.sh中设置HDFS的路径：

1. 序列化问题。
2. 储存DataFrame
3. 将DataFrame存储为parquet文件。
4. 将DataFrame存储到JDBC数据库。
5. 将DataFrame存储到Hive表。
6. 自定义函数UDF和UDAF
7. UDF:用户自定义函数。

可以自定义类实现UDFX接口。

java:

|  |
| --- |
| SparkConf conf = **new** SparkConf();  conf.setMaster("local");  conf.setAppName("udf");  JavaSparkContext sc = **new** JavaSparkContext(conf);  SQLContext sqlContext = **new** SQLContext(sc);  JavaRDD<String> parallelize = sc.parallelize(Arrays.*asList*("zhansan","lisi","wangwu"));  JavaRDD<Row> rowRDD = parallelize.map(**new** Function<String, Row>() {  /\*\*  \*  \*/  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  @Override  **public** Row call(String s) **throws** Exception {  **return** RowFactory.*create*(s);  }  });  List<StructField> fields = **new** ArrayList<StructField>();  fields.add(DataTypes.*createStructField*("name", DataTypes.***StringType***,**true**));  StructType schema = DataTypes.*createStructType*(fields);  DataFrame df = sqlContext.createDataFrame(rowRDD,schema);  df.registerTempTable("user");  /\*\*  \* 根据UDF函数参数的个数来决定是实现哪一个UDF UDF1，UDF2。。。。UDF1xxx  \*/  sqlContext.udf().register("StrLen", **new** UDF1<String,Integer>() {  /\*\*  \*  \*/  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  @Override  **public** Integer call(String t1) **throws** Exception {  **return** t1.length();  }  }, DataTypes.***IntegerType***);  sqlContext.sql("select name ,StrLen(name) as length from user").show();  //sqlContext.udf().register("StrLen",new UDF2<String, Integer, Integer>() {  //  // /\*\*  // \*  // \*/  // private static final long serialVersionUID = 1L;  //  // @Override  // public Integer call(String t1, Integer t2) throws Exception {  //return t1.length()+t2;  // }  //} ,DataTypes.IntegerType );  //sqlContext.sql("select name ,StrLen(name,10) as length from user").show();  sc.stop(); |

scala:

|  |
| --- |
| **val** conf = **new** SparkConf()  conf.setMaster("local").setAppName("udf")  **val** sc = **new** SparkContext(conf)  **val** sqlContext = **new** SQLContext(sc);  **val** rdd = sc.makeRDD(Array("zhansan","lisi","wangwu"))  **val** rowRDD = rdd.map { x => {  RowFactory.create(x)  } }  **val** schema = DataTypes.createStructType(Array(**StructField**("name",**StringType**,**true**)))  **val** df = sqlContext.createDataFrame(rowRDD, schema)  df.registerTempTable("user")  //sqlContext.udf.register("StrLen",(s : String)=>{s.length()})  //sqlContext.sql("select name ,StrLen(name) as length from user").show  sqlContext.udf.register("StrLen",(s : *String*,i:Int)=>{s.length()+i})  sqlContext.sql("select name ,StrLen(name,10) as length from user").show  sc.stop() |

1. UDAF:用户自定义聚合函数。

* 实现UDAF函数如果要自定义类要继承UserDefinedAggregateFunction类

java:

|  |
| --- |
| SparkConf conf = **new** SparkConf();  conf.setMaster("local").setAppName("udaf");  JavaSparkContext sc = **new** JavaSparkContext(conf);  SQLContext sqlContext = **new** SQLContext(sc);  JavaRDD<String> parallelize = sc.parallelize(Arrays.*asList*("zhansan","lisi","wangwu","zhangsan","zhangsan","lisi"));  JavaRDD<Row> rowRDD = parallelize.map(**new** Function<String, Row>() {  /\*\*  \*  \*/  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  @Override  **public** Row call(String s) **throws** Exception {  **return** RowFactory.*create*(s);  }  });  List<StructField> fields = **new** ArrayList<StructField>();  fields.add(DataTypes.*createStructField*("name", DataTypes.***StringType***, **true**));  StructType schema = DataTypes.*createStructType*(fields);  DataFrame df = sqlContext.createDataFrame(rowRDD, schema);  df.registerTempTable("user");  /\*\*  \* 注册一个UDAF函数,实现统计相同值得个数  \* 注意：这里可以自定义一个类继承UserDefinedAggregateFunction类也是可以的  \*/  sqlContext.udf().register("StringCount", **new** UserDefinedAggregateFunction() {    /\*\*  \*  \*/  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  /\*\*  \* 更新 可以认为一个一个地将组内的字段值传递进来 实现拼接的逻辑  \* buffer.getInt(0)获取的是上一次聚合后的值  \* 相当于map端的combiner，combiner就是对每一个map task的处理结果进行一次小聚合  \* 大聚和发生在reduce端.  \* 这里即是:在进行聚合的时候，每当有新的值进来，对分组后的聚合如何进行计算  \*/  @Override  **public** **void** update(MutableAggregationBuffer buffer, Row arg1) {  buffer.update(0, buffer.getInt(0)+1);  }  /\*\*  \* 合并 update操作，可能是针对一个分组内的部分数据，在某个节点上发生的 但是可能一个分组内的数据，会分布在多个节点上处理  \* 此时就要用merge操作，将各个节点上分布式拼接好的串，合并起来  \* buffer1.getInt(0) : 大聚和的时候 上一次聚合后的值  \* buffer2.getInt(0) : 这次计算传入进来的update的结果  \* 这里即是：最后在分布式节点完成后需要进行全局级别的Merge操作  \*/  @Override  **public** **void** merge(MutableAggregationBuffer buffer1, Row buffer2) {  buffer1.update(0, buffer1.getInt(0) + buffer2.getInt(0));  }  /\*\*  \* 指定输入字段的字段及类型  \*/  @Override  **public** StructType inputSchema() {  **return** DataTypes.*createStructType*(  Arrays.*asList*(DataTypes.*createStructField*("name",  DataTypes.***StringType***, **true**)));  }  /\*\*  \* 初始化一个内部的自己定义的值,在Aggregate之前每组数据的初始化结果  \*/  @Override  **public** **void** initialize(MutableAggregationBuffer buffer) {  buffer.update(0, 0);  }  /\*\*  \* 最后返回一个和DataType的类型要一致的类型，返回UDAF最后的计算结果  \*/  @Override  **public** Object evaluate(Row row) {  **return** row.getInt(0);  }    @Override  **public** **boolean** deterministic() {  //设置为true  **return** **true**;  }  /\*\*  \* 指定UDAF函数计算后返回的结果类型  \*/  @Override  **public** DataType dataType() {  **return** DataTypes.***IntegerType***;  }  /\*\*  \* 在进行聚合操作的时候所要处理的数据的结果的类型  \*/  @Override  **public** StructType bufferSchema() {  **return**  DataTypes.*createStructType*(  Arrays.*asList*(DataTypes.*createStructField*("bf", DataTypes.***IntegerType***,  **true**)));  }    });  sqlContext.sql("select name ,StringCount(name) from user group by name").show();  sc.stop(); |

scala:

|  |
| --- |
| **class** MyUDAF **extends** UserDefinedAggregateFunction {  // 聚合操作时，所处理的数据的类型  **def** bufferSchema: **StructType** = {  DataTypes.createStructType(Array(DataTypes.createStructField("aaa", **IntegerType**, **true**)))  }  // 最终函数返回值的类型  **def** dataType: DataType = {  DataTypes.IntegerType  }  **def** deterministic: Boolean = {  **true**  }  // 最后返回一个最终的聚合值 要和dataType的类型一一对应  **def** evaluate(buffer: Row): Any = {  buffer.getAs[Int](0)  }  // 为每个分组的数据执行初始化值  **def** initialize(buffer: MutableAggregationBuffer): Unit = {  buffer(0) = 0  }  //输入数据的类型  **def** inputSchema: **StructType** = {  DataTypes.createStructType(Array(DataTypes.createStructField("input", **StringType**, **true**)))  }  // 最后merger的时候，在各个节点上的聚合值，要进行merge，也就是合并  **def** merge(buffer1: MutableAggregationBuffer, buffer2: Row): Unit = {  buffer1(0) = buffer1.getAs[Int](0)+buffer2.getAs[Int](0)  }  // 每个组，有新的值进来的时候，进行分组对应的聚合值的计算  **def** update(buffer: MutableAggregationBuffer, input: Row): Unit = {  buffer(0) = buffer.getAs[Int](0)+1  }  }  **object** UDAF {  **def** main(args: Array[*String*]): Unit = {  **val** conf = **new** SparkConf()  conf.setMaster("local").setAppName("udaf")  **val** sc = **new** SparkContext(conf)  **val** sqlContext = **new** SQLContext(sc)  **val** rdd = sc.makeRDD(Array("zhangsan","lisi","wangwu","zhangsan","lisi"))  **val** rowRDD = rdd.map { x => {RowFactory.create(x)} }    **val** schema = DataTypes.createStructType(Array(DataTypes.createStructField("name", **StringType**, **true**)))  **val** df = sqlContext.createDataFrame(rowRDD, schema)  df.show()  df.registerTempTable("user")  /\*\*  \* 注册一个udaf函数  \*/  sqlContext.udf.register("StringCount", **new** MyUDAF())  sqlContext.sql("select name ,StringCount(name) from user group by name").show()  sc.stop()  }  } |

1. 开窗函数

注意：

row\_number() 开窗函数是按照某个字段分组，然后取另一字段的前几个的值，相当于 分组取topN

如果SQL语句里面使用到了开窗函数，那么这个SQL语句必须使用HiveContext来执行，HiveContext默认情况下在本地无法创建。在MySql8之后也增加了开窗函数。

开窗函数格式：

**row\_number() over (partitin by XXX order by XXX)**

java:

|  |
| --- |
| SparkConf conf = **new** SparkConf();  conf.setAppName("windowfun");  JavaSparkContext sc = **new** JavaSparkContext(conf);  HiveContext hiveContext = **new** HiveContext(sc);  hiveContext.sql("use spark");  hiveContext.sql("drop table if exists sales");  hiveContext.sql("create table if not exists sales (riqi string,leibie string,jine Int) "  + "row format delimited fields terminated by '\t'");  hiveContext.sql("load data local inpath '/root/test/sales' into table sales");  /\*\*  \* 开窗函数格式：  \* 【 rou\_number() over (partitin by **XXX** order by **XXX**) 】  \*/  DataFrame result = hiveContext.sql("select riqi,leibie,jine "  + "from ("  + "select riqi,leibie,jine,"  + "row\_number() over (partition by leibie order by jine desc) rank "  + "from sales) t "  + "where t.rank<=3");  result.show();  sc.stop(); |

scala:

|  |
| --- |
| **val** conf = **new** SparkConf()  conf.setAppName("windowfun")  **val** sc = **new** SparkContext(conf)  **val** hiveContext = **new** HiveContext(sc)  hiveContext.sql("use spark");  hiveContext.sql("drop table if exists sales");  hiveContext.sql("create table if not exists sales (riqi string,leibie string,jine Int) "  + "row format delimited fields terminated by '\t'");  hiveContext.sql("load data local inpath '/root/test/sales' into table sales");  /\*\*  \* 开窗函数格式：  \* 【 rou\_number() over (partitin by **XXX** order by **XXX**) 】  \*/  **val** result = hiveContext.sql("select riqi,leibie,jine "  + "from ("  + "select riqi,leibie,jine,"  + "row\_number() over (partition by leibie order by jine desc) rank "  + "from sales) t "  + "where t.rank<=3");  result.show();  sc.stop() |