**SparkSQL应用解析**

**教案**

**文档修订记录**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 文件状态：  [√ ]草稿  [ ]正式发布 | | **当前版本：** | | V1.1 | | | |
| **作 者：** | | 武玉飞 | | | |
| **审 核 人：** | |  | | | |
| **发布日期：** | | 20170518 | | | |
| **编制日期** | **版本** | | **状态** | **简要说明** | **作者** | **审核者** | **审核日期** |
| 20170518 | V1.1 | | A |  |  |  |  |
|  |  | |  |  |  |  |  |
|  |  | |  |  |  |  |  |
|  |  | |  |  |  |  |  |
|  |  | |  |  |  |  |  |
|  |  | |  |  |  |  |  |

说明：

1. 按修改时间先后倒序排列，最新修改的排在第一行。
2. 版本栏中填入版本编号或者更改记录编号。
3. 状态分为三种状态：A­——增加；M——修改；D——删除。
4. 在简要说明栏中填写变更的内容和变更的范围。
5. 表中所有日期格式为：YYYYMMDD。

**目 录**

[第1章 Spark SQL概述 4](#_Toc499563800)

[1.1 什么是Spark SQL 4](#_Toc499563801)

[1.2 RDD vs DataFrames vs DataSet 6](#_Toc499563802)

[1.2.1 RDD 6](#_Toc499563803)

[1.2.2 Dataframe 7](#_Toc499563804)

[1.2.3 Dataset 9](#_Toc499563805)

[1.2.4 三者的共性 10](#_Toc499563806)

[1.2.5 三者的区别 11](#_Toc499563807)

[第2章 执行SparkSQL查询 13](#_Toc499563808)

[2.1 命令行查询流程 13](#_Toc499563809)

[2.2 IDEA创建SparkSQL程序 14](#_Toc499563810)

[第3章 SparkSQL解析 15](#_Toc499563811)

[3.1 新的起始点SparkSession 15](#_Toc499563812)

[3.2 创建DataFrames 16](#_Toc499563813)

[3.3 DataFrame常用操作 17](#_Toc499563814)

[3.3.1 DSL风格语法 17](#_Toc499563815)

[3.3.2 SQL风格语法 19](#_Toc499563816)

[3.4 创建DataSet 20](#_Toc499563817)

[3.5 Dataset和RDD互操作 20](#_Toc499563818)

[3.5.1 通过反射获取Scheam 21](#_Toc499563819)

[3.5.2 通过编程设置Schema 22](#_Toc499563820)

[3.6 类型之间的转换总结 23](#_Toc499563821)

[3.7 用户自定义函数 24](#_Toc499563822)

[3.7.1 用户自定义UDF函数 24](#_Toc499563823)

[3.7.2 用户自定义聚合函数 25](#_Toc499563824)

[第4章 SparkSQL数据源 28](#_Toc499563825)

[4.1 通用加载/保存方法 28](#_Toc499563826)

[4.1.1 手动指定选项 28](#_Toc499563827)

[4.1.2 文件保存选项 30](#_Toc499563828)

[4.2 Parquet文件 30](#_Toc499563829)

[4.2.1 Parquet读写 30](#_Toc499563830)

[4.2.2 解析分区信息 31](#_Toc499563831)

[4.2.3 Schema合并 33](#_Toc499563832)

[4.3 Hive数据库 34](#_Toc499563833)

[4.3.1 内嵌Hive应用 36](#_Toc499563834)

[4.3.2 外部Hive应用 36](#_Toc499563835)

[4.4 JSON数据集 37](#_Toc499563836)

[4.5 JDBC 38](#_Toc499563837)

[第5章 JDBC/ODBC服务器 39](#_Toc499563838)

[第6章 运行Spark SQL CLI 40](#_Toc499563839)

[第7章 Spark SQL 的运行原理（了解） 41](#_Toc499563840)

[7.1 Spark SQL运行架构 41](#_Toc499563841)

[7.1.1 TreeNode 41](#_Toc499563842)

[7.1.2 Rule & RuleExecutor 41](#_Toc499563843)

[7.2 Spark SQL运行原理 42](#_Toc499563844)

[7.2.1 使用SessionCatalog保存元数据 42](#_Toc499563845)

[7.2.2 解析SQL,使用ANTLR生成未绑定的逻辑计划 43](#_Toc499563846)

[7.2.3 使用分析器Analyzer绑定逻辑计划 43](#_Toc499563847)

[7.2.4 使用优化器Optimizer优化逻辑计划 43](#_Toc499563848)

[7.2.5 使用SparkPlanner生成物理计划 43](#_Toc499563849)

[7.2.6 使用QueryExecution执行物理计划 43](#_Toc499563850)

[第8章 数据仓库的建设（了解） 43](#_Toc499563851)

[8.1 什么是数据仓库 43](#_Toc499563852)

[8.2 数据仓库的特点 44](#_Toc499563853)

[8.3 数据仓库发展历程 46](#_Toc499563854)

[8.4 数据库与数据仓库的区别 46](#_Toc499563855)

[8.5 数据仓库架构分层 48](#_Toc499563856)

[8.6 数据质量检查 49](#_Toc499563857)

[8.7 元数据介绍 50](#_Toc499563858)

[8.8 什么是数据模型 52](#_Toc499563859)

[8.9 为什么需要数据仓库模型 53](#_Toc499563860)

[8.10 如何建设数据仓库模型 54](#_Toc499563861)

[8.10.1 数据仓库数据模型架构 54](#_Toc499563862)

[8.10.2 数据仓库建模阶段划分 55](#_Toc499563863)

[8.10.3 数据仓库建模方法 57](#_Toc499563864)

[8.11 维度建模 61](#_Toc499563865)

[8.11.1 概述 61](#_Toc499563866)

[8.11.2 使用选择 62](#_Toc499563867)

[8.11.3 缓慢变化维 64](#_Toc499563868)

[第9章 Spark SQL实战 65](#_Toc499563869)

[9.1 数据说明 65](#_Toc499563870)

[9.2 加载数据 65](#_Toc499563871)

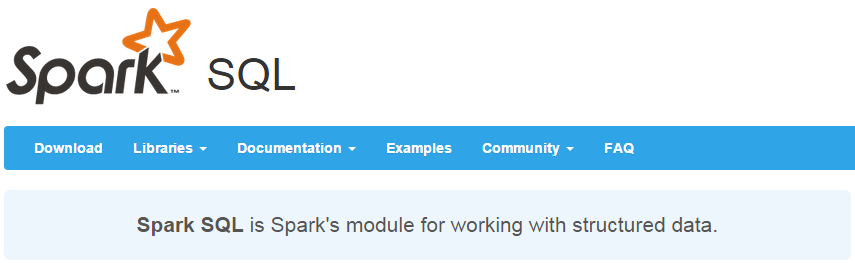
[9.3 计算所有订单中每年的销售单数、销售总额 69](#_Toc499563872)

[9.4 计算所有订单每年最大金额订单的销售额 70](#_Toc499563873)

[9.5 计算所有订单中每年最畅销货品 71](#_Toc499563874)

# Spark SQL概述

## 什么是Spark SQL



Spark SQL是Spark用来处理结构化数据的一个模块，它提供了一个编程抽象叫做DataFrame并且作为分布式SQL查询引擎的作用。

我们已经学习了Hive，它是将Hive SQL转换成MapReduce然后提交到集群上执行，大大简化了编写MapReduce的程序的复杂性，由于MapReduce这种计算模型执行效率比较慢。所有Spark SQL的应运而生，它是将Spark SQL转换成RDD，然后提交到集群执行，执行效率非常快！

1.易整合



1. 统一的数据访问方式

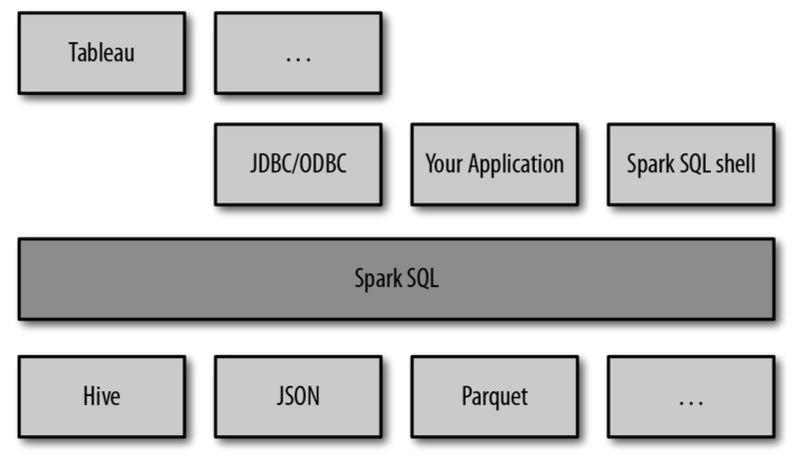


1. 兼容Hive



1. 标准的数据连接





SparkSQL可以看做是一个转换层，向下对接各种不同的结构化数据源，向上提供不同的数据访问方式。

## RDD vs DataFrames vs DataSet



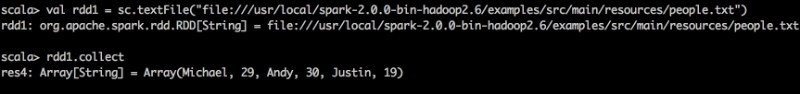
在SparkSQL中Spark为我们提供了两个新的抽象，分别是DataFrame和DataSet。他们和RDD有什么区别呢？首先从版本的产生上来看：  
RDD (Spark1.0) —> Dataframe(Spark1.3) —> Dataset(Spark1.6)

如果同样的数据都给到这三个数据结构，他们分别计算之后，都会给出相同的结果。不同是的他们的执行效率和执行方式。

在后期的Spark版本中，DataSet会逐步取代RDD和DataFrame成为唯一的API接口。

### RDD

* RDD是一个懒执行的不可变的可以支持Lambda表达式的并行数据集合。
* RDD的最大好处就是简单，API的人性化程度很高。
* RDD的劣势是性能限制，它是一个JVM驻内存对象，这也就决定了存在GC的限制和数据增加时Java序列化成本的升高。

RDD 例子如下:

### Dataframe

与RDD类似，DataFrame也是一个分布式数据容器。然而DataFrame更像传统数据库的二维表格，除了数据以外，还记录数据的结构信息，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。由于与R和Pandas的DataFrame类似，Spark DataFrame很好地继承了传统单机数据分析的开发体验。



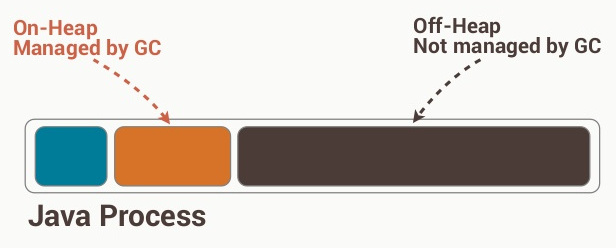
上图直观地体现了DataFrame和RDD的区别。左侧的RDD[Person]虽然以Person为类型参数，但Spark框架本身不了解Person类的内部结构。而右侧的DataFrame却提供了详细的结构信息，使得Spark SQL可以清楚地知道该数据集中包含哪些列，每列的名称和类型各是什么。DataFrame多了数据的结构信息，即schema。RDD是分布式的Java对象的集合。DataFrame是分布式的Row对象的集合。DataFrame除了提供了比RDD更丰富的算子以外，更重要的特点是提升执行效率、减少数据读取以及执行计划的优化，比如filter下推、裁剪等。

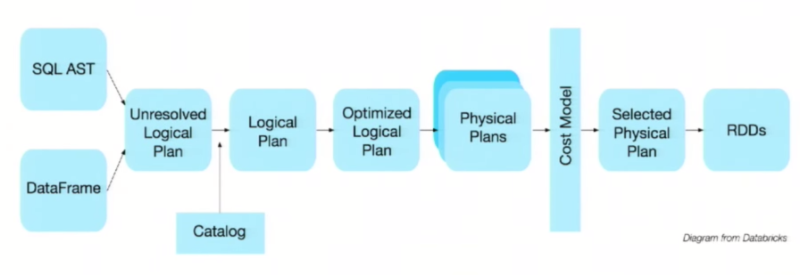
DataFrame是为数据提供了Schema的视图。可以把它当做数据库中的一张表来对待

DataFrame也是懒执行的。

性能上比RDD要高，主要有两方面原因：

*定制化内存管理*  
 数据以二进制的方式存在于非堆内存，节省了大量空间之外，还摆脱了GC的限制。



*优化的执行计划*  
查询计划通过Spark catalyst optimiser进行优化.   


比如下面一个例子：



为了说明查询优化，我们来看上图展示的人口数据分析的示例。图中构造了两个DataFrame，将它们join之后又做了一次filter操作。如果原封不动地执行这个执行计划，最终的执行效率是不高的。因为join是一个代价较大的操作，也可能会产生一个较大的数据集。如果我们能将filter下推到 join下方，先对DataFrame进行过滤，再join过滤后的较小的结果集，便可以有效缩短执行时间。而Spark SQL的查询优化器正是这样做的。简而言之，逻辑查询计划优化就是一个利用基于关系代数的等价变换，将高成本的操作替换为低成本操作的过程。

得到的优化执行计划在转换成物 理执行计划的过程中，还可以根据具体的数据源的特性将过滤条件下推至数据源内。最右侧的物理执行计划中Filter之所以消失不见，就是因为溶入了用于执行最终的读取操作的表扫描节点内。

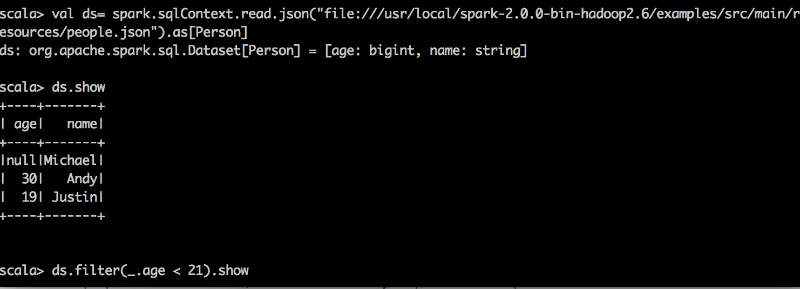
对于普通开发者而言，查询优化器的意义在于，即便是经验并不丰富的程序员写出的次优的查询，也可以被尽量转换为高效的形式予以执行。

Dataframe的劣势在于在编译期缺少类型安全检查，导致运行时出错.

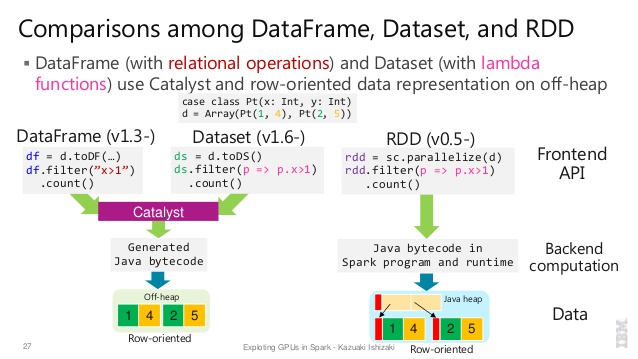
### Dataset

1. 是Dataframe API的一个扩展，是Spark最新的数据抽象
2. 用户友好的API风格，既具有类型安全检查也具有Dataframe的查询优化特性。
3. Dataset支持编解码器，当需要访问非堆上的数据时可以避免反序列化整个对象，提高了效率。
4. 样例类被用来在Dataset中定义数据的结构信息，样例类中每个属性的名称直接映射到DataSet中的字段名称。
5. Dataframe是Dataset的特列，DataFrame=Dataset[Row] ，所以可以通过as方法将Dataframe转换为Dataset。Row是一个类型，跟Car、Person这些的类型一样，所有的表结构信息我都用Row来表示。
6. DataSet是强类型的。比如可以有Dataset[Car]，Dataset[Person].

DataFrame只是知道字段，但是不知道字段的类型，所以在执行这些操作的时候是没办法在编译的时候检查是否类型失败的，比如你可以对一个String进行减法操作，在执行的时候才报错，而DataSet不仅仅知道字段，而且知道字段类型，所以有更严格的错误检查。就跟JSON对象和类对象之间的类比。

**

RDD让我们能够决定怎么做，而DataFrame和DataSet让我们决定做什么，控制的粒度不一样。



### 三者的共性

1、RDD、DataFrame、Dataset全都是spark平台下的分布式弹性数据集，为处理超大型数据提供便利

2、三者都有惰性机制，在进行创建、转换，如map方法时，不会立即执行，只有在遇到Action如foreach时，三者才会开始遍历运算，极端情况下，如果代码里面有创建、转换，但是后面没有在Action中使用对应的结果，在执行时会被直接跳过.

|  |
| --- |
| val sparkconf = new SparkConf().setMaster("local").setAppName("test").set("spark.port.maxRetries","1000")  val spark = SparkSession.builder().config(sparkconf).getOrCreate()  val rdd=spark.sparkContext.parallelize(Seq(("a", 1), ("b", 1), ("a", 1)))  // map不运行  rdd.map{line=>    println("运行")    line.\_1  } |

3、三者都会根据spark的内存情况自动缓存运算，这样即使数据量很大，也不用担心会内存溢出

4、三者都有partition的概念

5、三者有许多共同的函数，如filter，排序等

6、在对DataFrame和Dataset进行操作许多操作都需要这个包进行支持

|  |
| --- |
| import spark.implicits.\_ |

7、DataFrame和Dataset均可使用模式匹配获取各个字段的值和类型

DataFrame:

|  |
| --- |
| testDF.map{        case Row(col1:String,col2:Int)=>          println(col1);println(col2)          col1        case \_=>          ""      } |

Dataset:

|  |
| --- |
| case class Coltest(col1:String,col2:Int)extends Serializable //定义字段名和类型      testDS.map{        case Coltest(col1:String,col2:Int)=>          println(col1);println(col2)          col1        case \_=>          ""      } |

### 三者的区别

**RDD:**

1、RDD一般和spark mlib同时使用

2、RDD不支持sparksql操作

**DataFrame:**

1、与RDD和Dataset不同，DataFrame每一行的类型固定为Row，只有通过解析才能获取各个字段的值，如

|  |
| --- |
| testDF.foreach{    line =>      val col1=line.getAs[String]("col1")      val col2=line.getAs[String]("col2")  } |

每一列的值没法直接访问

2、DataFrame与Dataset一般不与spark ml同时使用

3、DataFrame与Dataset均支持sparksql的操作，比如select，groupby之类，还能注册临时表/视窗，进行sql语句操作，如

|  |
| --- |
| dataDF.createOrReplaceTempView("tmp")  spark.sql("select  ROW,DATE from tmp where DATE is not null order by DATE").show(100,false) |

4、DataFrame与Dataset支持一些特别方便的保存方式，比如保存成csv，可以带上表头，这样每一列的字段名一目了然

|  |
| --- |
| //保存  val saveoptions = Map("header" -> "true", "delimiter" -> "\t", "path" -> "hdfs://master01:9000/test")  datawDF.write.format("com.atguigu.spark.csv").mode(SaveMode.Overwrite).options(saveoptions).save()  //读取  val options = Map("header" -> "true", "delimiter" -> "\t", "path" -> "hdfs://master01:9000/test")  val datarDF= spark.read.options(options).format("com.atguigu.spark.csv").load() |

利用这样的保存方式，可以方便的获得字段名和列的对应，而且分隔符（delimiter）可以自由指定。

**Dataset:**

Dataset和DataFrame拥有完全相同的成员函数，区别只是每一行的数据类型不同。

DataFrame也可以叫Dataset[Row],每一行的类型是Row，不解析，每一行究竟有哪些字段，各个字段又是什么类型都无从得知，只能用上面提到的getAS方法或者共性中的第七条提到的模式匹配拿出特定字段

而Dataset中，每一行是什么类型是不一定的，在自定义了case class之后可以很自由的获得每一行的信息

|  |
| --- |
| case class Coltest(col1:String,col2:Int)extends Serializable //定义字段名和类型  /\*\*   rdd  ("a", 1)   ("b", 1)   ("a", 1)  \*\*/  val test: Dataset[Coltest]=rdd.map{line=>        Coltest(line.\_1,line.\_2)      }.toDS  test.map{        line=>          println(line.col1)          println(line.col2)      } |

可以看出，Dataset在需要访问列中的某个字段时是非常方便的，然而，如果要写一些适配性很强的函数时，如果使用Dataset，行的类型又不确定，可能是各种case class，无法实现适配，这时候用DataFrame即Dataset[Row]就能比较好的解决问题

# 执行SparkSQL查询

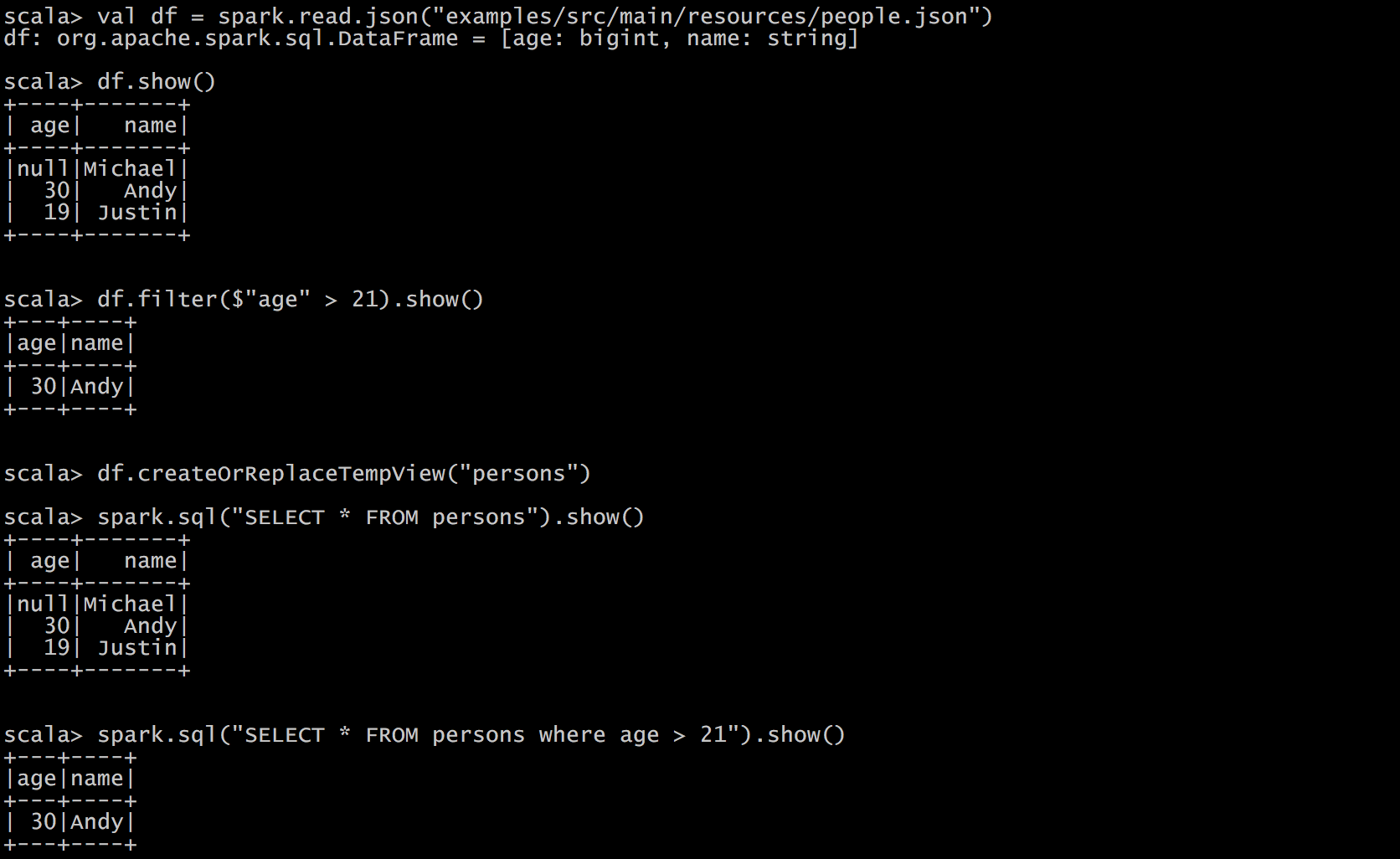
## 命令行查询流程

打开Spark shell

例子：查询大于30岁的用户

创建如下JSON文件，注意JSON的格式：

|  |
| --- |
| {**"name"**:**"Michael"**} {**"name"**:**"Andy"**, **"age"**:30} {**"name"**:**"Justin"**, **"age"**:19} |



## IDEA创建SparkSQL程序

IDEA中程序的打包和运行方式都和SparkCore类似，Maven依赖中需要添加新的依赖项：

<**dependency**>  
 <**groupId**>org.apache.spark</**groupId**>  
 <**artifactId**>spark-sql\_2.11</**artifactId**>  
 <**version**>${spark.version}</**version**>  
 <**scope**>provided</**scope**>  
</**dependency**>

程序如下：

|  |
| --- |
| **package** com.atguigu.sparksql  **import** org.apache.spark.sql.SparkSession **import** org.apache.spark.{SparkConf, SparkContext} **import** org.slf4j.LoggerFactory   */\*\*  \* Created by wuyufei on 31/07/2017.  \*/* **object** HelloWorld {   **val** *logger* = LoggerFactory.*getLogger*(HelloWorld.getClass)   **def** main(args: Array[String]) {  *//创建SparkConf()并设置App名称* **val** spark = SparkSession  .*builder*()  .appName(**"Spark SQL basic example"**)  .config(**"spark.some.config.option"**, **"some-value"**)  .getOrCreate()   *// For implicit conversions like converting RDDs to DataFrames* **import** spark.implicits.\_   **val** df = spark.read.json(**"examples/src/main/resources/people.json"**)   *// Displays the content of the DataFrame to stdout* df.show()   df.filter(**$"age"** > 21).show()   df.createOrReplaceTempView(**"persons"**)   spark.sql(**"SELECT \* FROM persons where age > 21"**).show()   spark.stop()  }  } |

# SparkSQL解析

## 新的起始点SparkSession

在老的版本中，SparkSQL提供两种SQL查询起始点，一个叫SQLContext，用于Spark自己提供的SQL查询，一个叫HiveContext，用于连接Hive的查询，SparkSession是Spark最新的SQL查询起始点，实质上是SQLContext和HiveContext的组合，所以在SQLContext和HiveContext上可用的API在SparkSession上同样是可以使用的。SparkSession内部封装了sparkContext，所以计算实际上是由sparkContext完成的。

|  |
| --- |
| **import** org.apache.spark.sql.SparkSession  **val** spark = SparkSession .builder() .appName(**"Spark SQL basic example"**) .config(**"spark.some.config.option"**, **"some-value"**) .getOrCreate()  *// For implicit conversions like converting RDDs to DataFrames* **import** spark.implicits.\_ |

SparkSession.builder 用于创建一个SparkSession。

import spark.implicits.\_的引入是用于将DataFrames隐式转换成RDD，使df能够使用RDD中的方法。

如果需要Hive支持，则需要以下创建语句：

|  |
| --- |
| **import** org.apache.spark.sql.SparkSession  **val** spark = SparkSession .builder() .appName(**"Spark SQL basic example"**) .config(**"spark.some.config.option"**, **"some-value"**) .enableHiveSupport() .getOrCreate()  *// For implicit conversions like converting RDDs to DataFrames* **import** spark.implicits.\_ |

## 创建DataFrames

 在Spark SQL中SparkSession是创建DataFrames和执行SQL的入口，创建DataFrames有三种方式，一种是可以从一个存在的RDD进行转换，还可以从Hive Table进行查询返回，或者通过Spark的数据源进行创建。

从Spark数据源进行创建：

|  |
| --- |
| **val** df = spark.read.json(**"examples/src/main/resources/people.json"**)  *// Displays the content of the DataFrame to stdout* df.show() *// +----+-------+ // | age| name| // +----+-------+ // |null|Michael| // | 30| Andy| // | 19| Justin| // +----+-------+* |

从RDD进行转换：

|  |
| --- |
| **/\*\***  **Michael, 29**  **Andy, 30**  **Justin, 19**  **\*\*/**  **scala> val peopleRdd = sc.textFile("examples/src/main/resources/people.txt")**  **peopleRdd: org.apache.spark.rdd.RDD[String] = examples/src/main/resources/people.txt MapPartitionsRDD[18] at textFile at <console>:24**  **scala> val peopleDF3 = peopleRdd.map(\_.split(",")).map(paras => (paras(0),paras(1).trim().toInt)).toDF("name","age")**  **peopleDF3: org.apache.spark.sql.DataFrame = [name: string, age: int]**  **scala> peopleDF.show()**  **+-------+---+**  **| name|age|**  **+-------+---+**  **|Michael| 29|**  **| Andy| 30|**  **| Justin| 19|**  **+-------+---+** |

Hive我们在数据源章节介绍

## DataFrame常用操作

### DSL风格语法

|  |
| --- |
| *// This import is needed to use the $-notation* **import** spark.implicits.\_ *// Print the schema in a tree format* df.printSchema() *// root // |-- age: long (nullable = true) // |-- name: string (nullable = true)  // Select only the "name" column* df.select(**"name"**).show() *// +-------+ // | name| // +-------+ // |Michael| // | Andy| // | Justin| // +-------+  // Select everybody, but increment the age by 1* df.select(**$"name"**, **$"age"** + 1).show() *// +-------+---------+ // | name|(age + 1)| // +-------+---------+ // |Michael| null| // | Andy| 31| // | Justin| 20| // +-------+---------+  // Select people older than 21* df.filter(**$"age"** > 21).show() *// +---+----+ // |age|name| // +---+----+ // | 30|Andy| // +---+----+  // Count people by age* df.groupBy(**"age"**).count().show() *// +----+-----+ // | age|count| // +----+-----+ // | 19| 1| // |null| 1| // | 30| 1| // +----+-----+* |

### SQL风格语法

|  |
| --- |
| *// Register the DataFrame as a SQL temporary view* df.createOrReplaceTempView(**"people"**)  **val** sqlDF = spark.sql(**"SELECT \* FROM people"**) sqlDF.show() *// +----+-------+ // | age| name| // +----+-------+ // |null|Michael| // | 30| Andy| // | 19| Justin| // +----+-------+   // Register the DataFrame as a global temporary view* df.createGlobalTempView(**"people"**)  *// Global temporary view is tied to a system preserved database `global\_temp`* spark.sql(**"SELECT \* FROM global\_temp.people"**).show() *// +----+-------+ // | age| name| // +----+-------+ // |null|Michael| // | 30| Andy| // | 19| Justin| // +----+-------+  // Global temporary view is cross-session* spark.newSession().sql(**"SELECT \* FROM global\_temp.people"**).show() *// +----+-------+ // | age| name| // +----+-------+ // |null|Michael| // | 30| Andy| // | 19| Justin| // +----+-------+* |

临时表是Session范围内的，Session退出后，表就失效了。如果想应用范围内有效，可以使用全局表。注意使用全局表时需要全路径访问，如：global\_temp.people

## 创建DataSet

Dataset是具有强类型的数据集合，需要提供对应的类型信息。

|  |
| --- |
| *// Note: Case classes in Scala 2.10 can support only up to 22 fields. To work around this limit, // you can use custom classes that implement the Product interface* **case class** Person(name: String, age: Long)  *// Encoders are created for case classes* **val** caseClassDS = Seq(Person(**"Andy"**, 32)).toDS() caseClassDS.show() *// +----+---+ // |name|age| // +----+---+ // |Andy| 32| // +----+---+  // Encoders for most common types are automatically provided by importing spark.implicits.\_* **val** primitiveDS = Seq(1, 2, 3).toDS() primitiveDS.map(\_ + 1).collect() *// Returns: Array(2, 3, 4)  // DataFrames can be converted to a Dataset by providing a class. Mapping will be done by name* **val** path = **"examples/src/main/resources/people.json" val** peopleDS = spark.read.json(path).as[Person] peopleDS.show() *// +----+-------+ // | age| name| // +----+-------+ // |null|Michael| // | 30| Andy| // | 19| Justin| // +----+-------+* |

## Dataset和RDD互操作

Spark SQL支持通过两种方式将存在的RDD转换为Dataset，转换的过程中需要让Dataset获取RDD中的Schema信息，主要有两种方式，一种是通过反射来获取RDD中的Schema信息。这种方式适合于列名已知的情况下。第二种是通过编程接口的方式将Schema信息应用于RDD，这种方式可以处理那种在运行时才能知道列的方式。

### 通过反射获取Scheam

SparkSQL能够自动将包含有case类的RDD转换成DataFrame，case类定义了table的结构，case类属性通过反射变成了表的列名。Case类可以包含诸如Seqs或者Array等复杂的结构。

|  |
| --- |
| *// For implicit conversions from RDDs to DataFrames* **import** spark.implicits.\_  *// Create an RDD of Person objects from a text file, convert it to a Dataframe* **val** peopleDF = spark.sparkContext .textFile(**"examples/src/main/resources/people.txt"**) .map(\_.split(**","**)) .map(attributes => Person(attributes(0), attributes(1).trim.toInt)) .toDF()  *// Register the DataFrame as a temporary view* peopleDF.createOrReplaceTempView(**"people"**)  *// SQL statements can be run by using the sql methods provided by Spark* **val** teenagersDF = spark.sql(**"SELECT name, age FROM people WHERE age BETWEEN 13 AND 19"**)  *// The columns of a row in the result can be accessed by field index ROW object* teenagersDF.map(teenager => **"Name: "** + teenager(0)).show() *// +------------+ // | value| // +------------+ // |Name: Justin| // +------------+  // or by field name* teenagersDF.map(teenager => **"Name: "** + teenager.getAs[String](**"name"**)).show() *// +------------+ // | value| // +------------+ // |Name: Justin| // +------------+  // No pre-defined encoders for Dataset[Map[K,V]], define explicitly* **implicit val** mapEncoder = org.apache.spark.sql.Encoders.kryo[Map[String, Any]] *// Primitive types and case classes can be also defined as // implicit val stringIntMapEncoder: Encoder[Map[String, Any]] = ExpressionEncoder()  // row.getValuesMap[T] retrieves multiple columns at once into a Map[String, T]* teenagersDF.map(teenager => teenager.getValuesMap[Any](List(**"name"**, **"age"**))).collect() *// Array(Map("name" -> "Justin", "age" -> 19))* |

### 通过编程设置Schema

如果case类不能够提前定义，可以通过下面三个步骤定义一个DataFrame

创建一个多行结构的RDD;

创建用StructType来表示的行结构信息。

通过SparkSession提供的createDataFrame方法来应用Schema .

|  |
| --- |
| **import** org.apache.spark.sql.types.\_  *// Create an RDD* **val** peopleRDD = spark.sparkContext.textFile(**"examples/src/main/resources/people.txt"**)  *// The schema is encoded in a string,应该是动态通过程序生成的* **val** schemaString = **"name age"** *// Generate the schema based on the string of schema Array[StructFiled]* **val** fields = schemaString.split(**" "**) .map(fieldName => StructField(fieldName, StringType, nullable = **true**))  // val filed = schemaString.split(" ").map(filename=> filename match{ case "name"=> StructField(filename,StringType,nullable = true); case "age"=>StructField(filename, IntegerType,nullable = true)} )  **val** schema = StructType(fields)  *// Convert records of the RDD (people) to Rows*  *import org.apache.spark.sql.\_* **val** rowRDD = peopleRDD .map(\_.split(**","**)) .map(attributes => Row(attributes(0), attributes(1).trim))  *// Apply the schema to the RDD* **val** peopleDF = spark.createDataFrame(rowRDD, schema)  *// Creates a temporary view using the DataFrame* peopleDF.createOrReplaceTempView(**"people"**)  *// SQL can be run over a temporary view created using DataFrames* **val** results = spark.sql(**"SELECT name FROM people"**)  *// The results of SQL queries are DataFrames and support all the normal RDD operations // The columns of a row in the result can be accessed by field index or by field name* results.map(attributes => **"Name: "** + attributes(0)).show() *// +-------------+ // | value| // +-------------+ // |Name: Michael| // | Name: Andy| // | Name: Justin| // +-------------+* |

## 类型之间的转换总结

RDD、DataFrame、Dataset三者有许多共性，有各自适用的场景常常需要在三者之间转换

***DataFrame/Dataset转RDD：***

这个转换很简单

|  |
| --- |
| val rdd1=testDF.rdd  val rdd2=testDS.rdd |

***RDD转DataFrame：***

|  |
| --- |
| import spark.implicits.\_  val testDF = rdd.map {line=>        (line.\_1,line.\_2)      }.toDF("col1","col2") |

一般用元组把一行的数据写在一起，然后在toDF中指定字段名

***RDD转Dataset：***

|  |
| --- |
| import spark.implicits.\_  case class Coltest(col1:String,col2:Int)extends Serializable //定义字段名和类型  val testDS = rdd.map {line=>        Coltest(line.\_1,line.\_2)      }.toDS |

可以注意到，定义每一行的类型（case class）时，已经给出了字段名和类型，后面只要往case class里面添加值即可

***Dataset转DataFrame：***

这个也很简单，因为只是把case class封装成Row

|  |
| --- |
| import spark.implicits.\_  val testDF = testDS.toDF |

***DataFrame转Dataset：***

|  |
| --- |
| import spark.implicits.\_  case class Coltest(col1:String,col2:Int)extends Serializable //定义字段名和类型  val testDS = testDF.as[Coltest] |

这种方法就是在给出每一列的类型后，使用as方法，转成Dataset，这在数据类型是DataFrame又需要针对各个字段处理时极为方便。

在使用一些特殊的操作时，一定要加上 import spark.implicits.\_ 不然toDF、toDS无法使用

## 用户自定义函数

通过spark.udf功能用户可以自定义函数。

### 用户自定义UDF函数

|  |
| --- |
| **scala> val df = spark.read.json("examples/src/main/resources/people.json")**  **df: org.apache.spark.sql.DataFrame = [age: bigint, name: string]**  **scala> df.show()**  **+----+-------+**  **| age| name|**  **+----+-------+**  **|null|Michael|**  **| 30| Andy|**  **| 19| Justin|**  **+----+-------+**  **scala> spark.udf.register("addName", (x:String)=> "Name:"+x)**  **res5: org.apache.spark.sql.expressions.UserDefinedFunction = UserDefinedFunction(<function1>,StringType,Some(List(StringType)))**  **scala> df.createOrReplaceTempView("people")**  **scala> spark.sql("Select addName(name), age from people").show()**  **+-----------------+----+**  **|UDF:addName(name)| age|**  **+-----------------+----+**  **| Name:Michael|null|**  **| Name:Andy| 30|**  **| Name:Justin| 19|**  **+-----------------+----+** |

### 用户自定义聚合函数

强类型的Dataset和弱类型的DataFrame都提供了相关的聚合函数， 如 count()，countDistinct()，avg()，max()，min()。除此之外，用户可以设定自己的自定义聚合函数。

#### 弱类型用户自定义聚合函数

通过继承UserDefinedAggregateFunction来实现用户自定义聚合函数。下面展示一个求平均工资的自定义聚合函数。

|  |
| --- |
| **import** org.apache.spark.sql.expressions.MutableAggregationBuffer **import** org.apache.spark.sql.expressions.UserDefinedAggregateFunction **import** org.apache.spark.sql.types.\_ **import** org.apache.spark.sql.Row **import** org.apache.spark.sql.SparkSession  **object** MyAverage **extends** UserDefinedAggregateFunction { *// 聚合函数输入参数的数据类型* **def** inputSchema: StructType = StructType(StructField(**"inputColumn"**, LongType) :: Nil) *// 聚合缓冲区中值得数据类型* **def** bufferSchema: StructType = { StructType(StructField(**"sum"**, LongType) :: StructField(**"count"**, LongType) :: Nil) } *// 返回值的数据类型* **def** dataType: DataType = DoubleType *// 对于相同的输入是否一直返回相同的输出。* **def** deterministic: Boolean = **true** *// 初始化* **def** initialize(buffer: MutableAggregationBuffer): Unit = {  *// 存工资的总额* buffer(0) = 0L  *// 存工资的个数* buffer(1) = 0L } *// 相同Execute间的数据合并。* **def** update(buffer: MutableAggregationBuffer, input: Row): Unit = { **if** (!input.isNullAt(0)) { buffer(0) = buffer.getLong(0) + input.getLong(0) buffer(1) = buffer.getLong(1) + 1 } } *// 不同Execute间的数据合并* **def** merge(buffer1: MutableAggregationBuffer, buffer2: Row): Unit = { buffer1(0) = buffer1.getLong(0) + buffer2.getLong(0) buffer1(1) = buffer1.getLong(1) + buffer2.getLong(1) } *// 计算最终结果*  **def** evaluate(buffer: Row): Double = buffer.getLong(0).toDouble / buffer.getLong(1) }  *// 注册函数* spark.udf.register(**"myAverage"**, MyAverage)  **val** df = spark.read.json(**"examples/src/main/resources/employees.json"**) df.createOrReplaceTempView(**"employees"**) df.show() *// +-------+------+ // | name|salary| // +-------+------+ // |Michael| 3000| // | Andy| 4500| // | Justin| 3500| // | Berta| 4000| // +-------+------+* **val** result = spark.sql(**"SELECT myAverage(salary) as average\_salary FROM employees"**) result.show() *// +--------------+ // |average\_salary| // +--------------+ // | 3750.0| // +--------------+* |

#### 强类型用户自定义聚合函数

通过继承Aggregator来实现强类型自定义聚合函数，同样是求平均工资

|  |
| --- |
| **import** org.apache.spark.sql.expressions.Aggregator **import** org.apache.spark.sql.Encoder **import** org.apache.spark.sql.Encoders **import** org.apache.spark.sql.SparkSession *// 既然是强类型，可能有case类* **case class** Employee(name: String, salary: Long) **case class** Average(**var** sum: Long, **var** count: Long)  **object** MyAverage **extends** Aggregator[Employee, Average, Double] { *// 定义一个数据结构，保存工资总数和工资总个数，初始都为0* **def** zero: Average = Average(0L, 0L) *// Combine two values to produce a new value. For performance, the function may modify `buffer` // and return it instead of constructing a new object* **def** reduce(buffer: Average, employee: Employee): Average = { buffer.sum += employee.salary buffer.count += 1 buffer } *// 聚合不同execute的结果* **def** merge(b1: Average, b2: Average): Average = { b1.sum += b2.sum b1.count += b2.count b1 } *// 计算输出* **def** finish(reduction: Average): Double = reduction.sum.toDouble / reduction.count *// 设定之间值类型的编码器，要转换成case类*  *// Encoders.product是进行scala元组和case类转换的编码器* **def** bufferEncoder: Encoder[Average] = Encoders.product *// 设定最终输出值的编码器* **def** outputEncoder: Encoder[Double] = Encoders.scalaDouble }  import spark.implicits.\_  **val** ds = spark.read.json(**"examples/src/main/resources/employees.json"**).as[Employee] ds.show() *// +-------+------+ // | name|salary| // +-------+------+ // |Michael| 3000| // | Andy| 4500| // | Justin| 3500| // | Berta| 4000| // +-------+------+  // Convert the function to a `TypedColumn` and give it a name* **val** averageSalary = MyAverage.toColumn.name(**"average\_salary"**) **val** result = ds.select(averageSalary) result.show() *// +--------------+ // |average\_salary| // +--------------+ // | 3750.0| // +--------------+* |

# SparkSQL数据源

## 通用加载/保存方法

### 手动指定选项

Spark SQL的DataFrame接口支持多种数据源的操作。一个DataFrame可以进行RDDs方式的操作，也可以被注册为临时表。把DataFrame注册为临时表之后，就可以对该DataFrame执行SQL查询。

Spark SQL的默认数据源为Parquet格式。数据源为Parquet文件时，Spark SQL可以方便的执行所有的操作。修改配置项spark.sql.sources.default，可修改默认数据源格式。

|  |
| --- |
| val df = spark.read.load("examples/src/main/resources/users.parquet") df.select("name", "favorite\_color").write.save("namesAndFavColors.parquet") |

当数据源格式不是parquet格式文件时，需要手动指定数据源的格式。数据源格式需要指定全名（例如：org.apache.spark.sql.parquet），如果数据源格式为内置格式，则只需要指定简称定json, parquet, jdbc, orc, libsvm, csv, text来指定数据的格式。

可以通过SparkSession提供的read.load方法用于通用加载数据，使用write和save保存数据。

|  |
| --- |
| **val** peopleDF = **spark.read.format("json").load("examples/src/main/resources/people.json") peopleDF.write.format("parquet").save("hdfs://master01:9000/namesAndAges.parquet")** |

除此之外，可以直接运行SQL在文件上：

|  |
| --- |
| **val sqlDF = spark.sql("SELECT \* FROM parquet.`hdfs://master01:9000/namesAndAges.parquet`")**  **sqlDF.show()** |

|  |
| --- |
| **scala> val peopleDF = spark.read.format("json").load("examples/src/main/resources/people.json")**  **peopleDF: org.apache.spark.sql.DataFrame = [age: bigint, name: string]**  **scala> peopleDF.write.format("parquet").save("hdfs://master01:9000/namesAndAges.parquet")**  **scala> peopleDF.show()**  **+----+-------+**  **| age| name|**  **+----+-------+**  **|null|Michael|**  **| 30| Andy|**  **| 19| Justin|**  **+----+-------+**  **scala> val sqlDF = spark.sql("SELECT \* FROM parquet.`hdfs://master01:9000/namesAndAges.parquet`")**  **17/09/05 04:21:11 WARN ObjectStore: Failed to get database parquet, returning NoSuchObjectException**  **sqlDF: org.apache.spark.sql.DataFrame = [age: bigint, name: string]**  **scala> sqlDF.show()**  **+----+-------+**  **| age| name|**  **+----+-------+**  **|null|Michael|**  **| 30| Andy|**  **| 19| Justin|**  **+----+-------+** |

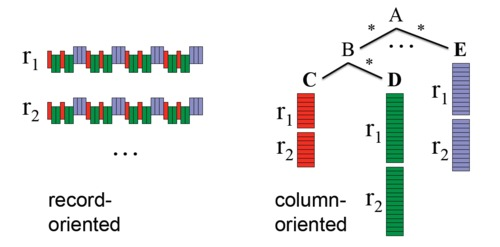
### 文件保存选项

可以采用SaveMode执行存储操作，SaveMode定义了对数据的处理模式。需要注意的是，这些保存模式不使用任何锁定，不是原子操作。此外，当使用Overwrite方式执行时，在输出新数据之前原数据就已经被删除。SaveMode详细介绍如下表：

|  |  |  |
| --- | --- | --- |
| **Scala/Java** | **Any Language** | **Meaning** |
| **SaveMode.ErrorIfExists(default)** | "error"(default) | 如果文件存在，则报错 |
| **SaveMode.Append** | "append" | 追加 |
| **SaveMode.Overwrite** | "overwrite" | 覆写 |
| **SaveMode.Ignore** | "ignore" | 数据存在，则忽略 |

## Parquet文件

Parquet是一种流行的列式存储格式，可以高效地存储具有嵌套字段的记录。



### Parquet读写

Parquet格式经常在Hadoop生态圈中被使用，它也支持Spark SQL的全部数据类型。Spark SQL 提供了直接读取和存储 Parquet 格式文件的方法。

|  |
| --- |
| *// Encoders for most common types are automatically provided by importing spark.implicits.\_* **import** spark.implicits.\_  **val** peopleDF = spark.read.json(**"examples/src/main/resources/people.json"**)  *// DataFrames can be saved as Parquet files, maintaining the schema information* peopleDF.write.parquet(**"hdfs://master01:9000/people.parquet"**)  *// Read in the parquet file created above // Parquet files are self-describing so the schema is preserved // The result of loading a Parquet file is also a DataFrame* **val** parquetFileDF = spark.read.parquet(**"hdfs://master01:9000/people.parquet"**)  *// Parquet files can also be used to create a temporary view and then used in SQL statements* parquetFileDF.createOrReplaceTempView(**"parquetFile"**) **val** namesDF = spark.sql(**"SELECT name FROM parquetFile WHERE age BETWEEN 13 AND 19"**) namesDF.map(attributes => **"Name: "** + attributes(0)).show() *// +------------+ // | value| // +------------+ // |Name: Justin| // +------------+* |

### 解析分区信息

对表进行分区是对数据进行优化的方式之一。在分区的表内，数据通过分区列将数据存储在不同的目录下。Parquet数据源现在能够自动发现并解析分区信息。例如，对人口数据进行分区存储，分区列为gender和country，使用下面的目录结构：

**path**

**└── to**

**└── table**

**├── gender=male**

**│ ├── ...**

**│ │**

**│ ├── country=US**

**│ │ └── data.parquet**

**│ ├── country=CN**

**│ │ └── data.parquet**

**│ └── ...**

**└── gender=female**

**├── ...**

**│**

**├── country=US**

**│ └── data.parquet**

**├── country=CN**

**│ └── data.parquet**

**└── ...**

通过传递path/to/table给 SQLContext.read.parquet或SQLContext.read.load，Spark SQL将自动解析分区信息。返回的DataFrame的Schema如下：

**root**

**|-- name: string (nullable = true)**

**|-- age: long (nullable = true)**

**|-- gender: string (nullable = true)**

**|-- country: string (nullable = true)**

需要注意的是，数据的分区列的数据类型是自动解析的。当前，支持数值类型和字符串类型。自动解析分区类型的参数为：spark.sql.sources.partitionColumnTypeInference.enabled，默认值为true。如果想关闭该功能，直接将该参数设置为disabled。此时，分区列数据格式将被默认设置为string类型，不再进行类型解析。

### Schema合并

像ProtocolBuffer、Avro和Thrift那样，Parquet也支持Schema evolution（Schema演变）。用户可以先定义一个简单的Schema，然后逐渐的向Schema中增加列描述。通过这种方式，用户可以获取多个有不同Schema但相互兼容的Parquet文件。现在Parquet数据源能自动检测这种情况，并合并这些文件的schemas。

因为Schema合并是一个高消耗的操作，在大多数情况下并不需要，所以Spark SQL从1.5.0开始默认关闭了该功能。可以通过下面两种方式开启该功能：

当数据源为Parquet文件时，将数据源选项mergeSchema设置为true

设置全局SQL选项spark.sql.parquet.mergeSchema为true

**示例如下：**

|  |
| --- |
| *// sqlContext from the previous example is used in this example. // This is used to implicitly convert an RDD to a DataFrame.* **import** spark.implicits.\_  *// Create a simple DataFrame, stored into a partition directory* **val** df1 = sc.makeRDD(1 to 5).map(i => (i, i \* 2)).toDF(**"single"**, **"double"**) df1.write.parquet(**"hdfs://master01:9000/data/test\_table/key=1"**)  *// Create another DataFrame in a new partition directory, // adding a new column and dropping an existing column* **val** df2 = sc.makeRDD(6 to 10).map(i => (i, i \* 3)).toDF(**"single"**, **"triple"**) df2.write.parquet(**"hdfs://master01:9000/data/test\_table/key=2"**)  *// Read the partitioned table* **val** df3 = spark.read.option(**"mergeSchema"**, **"true"**).parquet(**"hdfs://master01:9000/data/test\_table"**) df3.printSchema()  *// The final schema consists of all 3 columns in the Parquet files together // with the partitioning column appeared in the partition directory paths. // root // |-- single: int (nullable = true) // |-- double: int (nullable = true) // |-- triple: int (nullable = true) // |-- key : int (nullable = true)* |

## Hive数据库

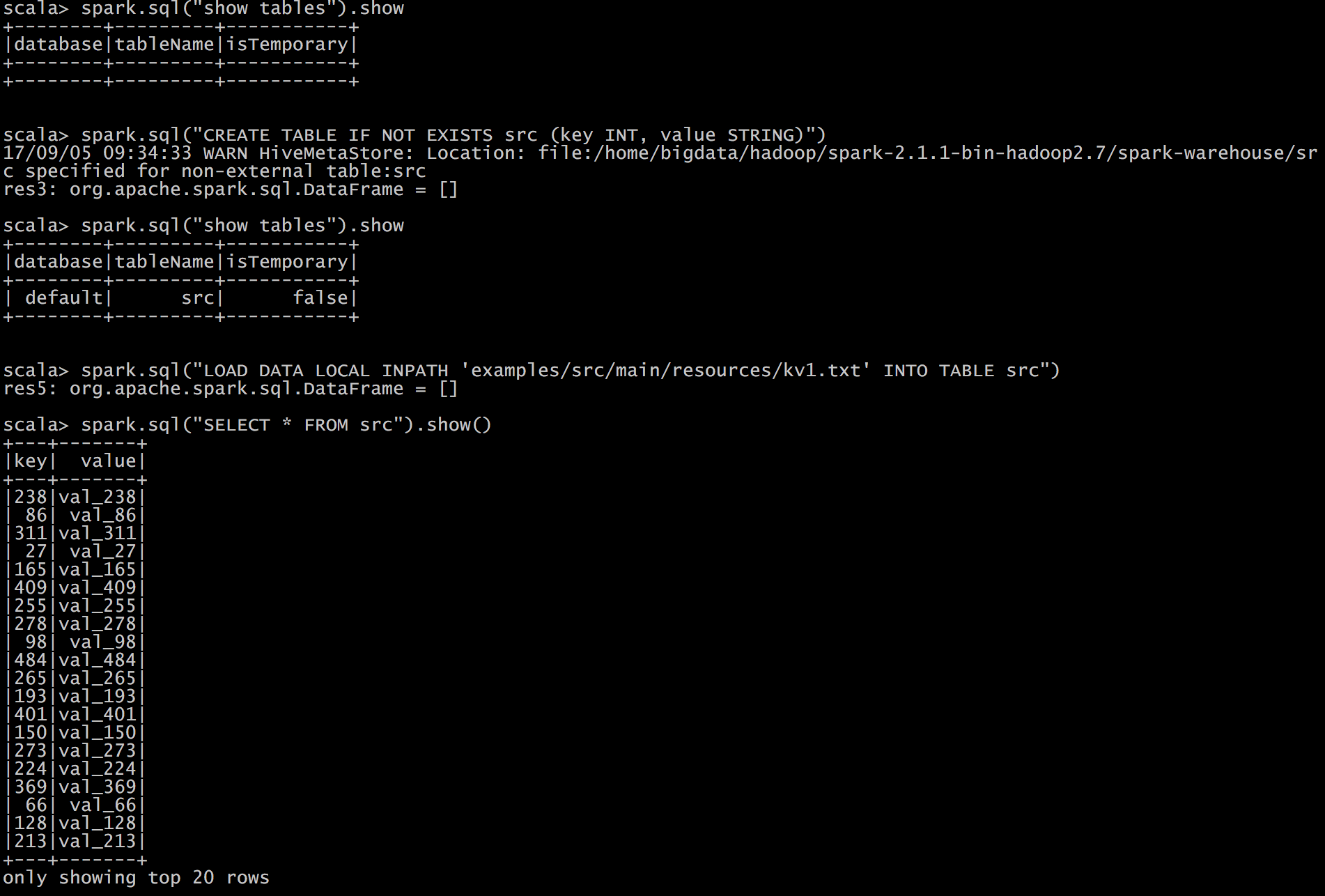
Apache Hive是Hadoop上的SQL引擎，Spark SQL编译时可以包含Hive支持，也可以不包含。包含Hive支持的Spark SQL可以支持Hive表访问、UDF(用户自定义函数)以及 Hive 查询语言(HiveQL/HQL)等。需要强调的 一点是，如果要在Spark SQL中包含Hive的库，并不需要事先安装Hive。一般来说，最好还是在编译Spark SQL时引入Hive支持，这样就可以使用这些特性了。如果你下载的是二进制版本的 Spark，它应该已经在编译时添加了 Hive 支持。

若要把Spark SQL连接到一个部署好的Hive上，你必须把hive-site.xml复制到 Spark的配置文件目录中($SPARK\_HOME/conf)。即使没有部署好Hive，Spark SQL也可以运行。 需要注意的是，如果你没有部署好Hive，Spark SQL会在当前的工作目录中创建出自己的Hive 元数据仓库，叫作 metastore\_db。此外，如果你尝试使用 HiveQL 中的 CREATE TABLE (并非 CREATE EXTERNAL TABLE)语句来创建表，这些表会被放在你默认的文件系统中的 /user/hive/warehouse 目录中(如果你的 classpath 中有配好的 hdfs-site.xml，默认的文件系统就是 HDFS，否则就是本地文件系统)。

|  |
| --- |
| **import** java.io.File  **import** org.apache.spark.sql.Row **import** org.apache.spark.sql.SparkSession  **case class** Record(key: Int, value: String)  *// warehouseLocation points to the default location for managed databases and tables* **val** warehouseLocation = **new** File(**"spark-warehouse"**).getAbsolutePath  **val** spark = SparkSession .builder() .appName(**"Spark Hive Example"**) .config(**"spark.sql.warehouse.dir"**, warehouseLocation) .enableHiveSupport() .getOrCreate()  **import** spark.implicits.\_ **import** spark.sql  sql(**"CREATE TABLE IF NOT EXISTS src (key INT, value STRING)"**) sql(**"LOAD DATA LOCAL INPATH 'examples/src/main/resources/kv1.txt' INTO TABLE src"**)  *// Queries are expressed in HiveQL* sql(**"SELECT \* FROM src"**).show() *// +---+-------+ // |key| value| // +---+-------+ // |238|val\_238| // | 86| val\_86| // |311|val\_311| // ...  // Aggregation queries are also supported.* sql(**"SELECT COUNT(\*) FROM src"**).show() *// +--------+ // |count(1)| // +--------+ // | 500 | // +--------+  // The results of SQL queries are themselves DataFrames and support all normal functions.* **val** sqlDF = sql(**"SELECT key, value FROM src WHERE key < 10 ORDER BY key"**)  *// The items in DataFrames are of type Row, which allows you to access each column by ordinal.* **val** stringsDS = sqlDF.map { **case** Row(key: Int, value: String) => **s"Key: $**key**, Value: $**value**"** } stringsDS.show() *// +--------------------+ // | value| // +--------------------+ // |Key: 0, Value: val\_0| // |Key: 0, Value: val\_0| // |Key: 0, Value: val\_0| // ...  // You can also use DataFrames to create temporary views within a SparkSession.* **val** recordsDF = spark.createDataFrame((1 to 100).map(i => Record(i, **s"val\_$**i**"**))) recordsDF.createOrReplaceTempView(**"records"**)  *// Queries can then join DataFrame data with data stored in Hive.* sql(**"SELECT \* FROM records r JOIN src s ON r.key = s.key"**).show() *// +---+------+---+------+ // |key| value|key| value| // +---+------+---+------+ // | 2| val\_2| 2| val\_2| // | 4| val\_4| 4| val\_4| // | 5| val\_5| 5| val\_5| // ...* |

### 内嵌Hive应用

如果要使用内嵌的Hive，什么都不用做，直接用就可以了。 --conf : spark.sql.warehouse.dir=



**注意：**如果你使用的是内部的Hive，在Spark2.0之后，spark.sql.warehouse.dir用于指定数据仓库的地址，如果你需要是用HDFS作为路径，那么需要将core-site.xml和hdfs-site.xml 加入到Spark conf目录，否则只会创建master节点上的warehouse目录，查询时会出现文件找不到的问题，这是需要向使用HDFS，则需要将metastore删除，重启集群。

### 外部Hive应用

如果想连接外部已经部署好的Hive，需要通过以下几个步骤。

1. 将Hive中的hive-site.xml拷贝或者软连接到Spark安装目录下的conf目录下。
2. 打开spark shell，注意带上访问Hive元数据库的JDBC客户端

|  |
| --- |
| $ bin/spark-shell --master spark://master01:7077 --jars mysql-connector-java-5.1.27-bin.jar |

## JSON数据集

Spark SQL 能够自动推测 JSON数据集的结构，并将它加载为一个Dataset[Row]. 可以通过SparkSession.read.json()去加载一个 Dataset[String]或者一个JSON 文件.注意，这个JSON文件不是一个传统的JSON文件，每一行都得是一个JSON串。

|  |
| --- |
| {**"name"**:**"Michael"**} {**"name"**:**"Andy"**, **"age"**:30} {**"name"**:**"Justin"**, **"age"**:19} |

|  |
| --- |
| *// Primitive types (Int, String, etc) and Product types (case classes) encoders are // supported by importing this when creating a Dataset.* **import** spark.implicits.\_  *// A JSON dataset is pointed to by path. // The path can be either a single text file or a directory storing text files* **val** path = **"examples/src/main/resources/people.json" val** peopleDF = spark.read.json(path)  *// The inferred schema can be visualized using the printSchema() method* peopleDF.printSchema() *// root // |-- age: long (nullable = true) // |-- name: string (nullable = true)  // Creates a temporary view using the DataFrame* peopleDF.createOrReplaceTempView(**"people"**)  *// SQL statements can be run by using the sql methods provided by spark* **val** teenagerNamesDF = spark.sql(**"SELECT name FROM people WHERE age BETWEEN 13 AND 19"**) teenagerNamesDF.show() *// +------+ // | name| // +------+ // |Justin| // +------+  // Alternatively, a DataFrame can be created for a JSON dataset represented by // a Dataset[String] storing one JSON object per string* **val** otherPeopleDataset = spark.createDataset( **"""{"name":"Yin","address":{"city":"Columbus","state":"Ohio"}}"""** :: Nil) **val** otherPeople = spark.read.json(otherPeopleDataset) otherPeople.show() *// +---------------+----+ // | address|name| // +---------------+----+ // |[Columbus,Ohio]| Yin| // +---------------+----+* |

## JDBC

Spark SQL可以通过JDBC从关系型数据库中读取数据的方式创建DataFrame，通过对DataFrame一系列的计算后，还可以将数据再写回关系型数据库中。

注意，需要将相关的数据库驱动放到spark的类路径下。

|  |
| --- |
| $ bin/spark-shell --master spark://master01:7077 --jars mysql-connector-java-5.1.27-bin.jar |
| *// Note: JDBC loading and saving can be achieved via either the load/save or jdbc methods // Loading data from a JDBC source* **val** jdbcDF = spark.read.format(**"jdbc"**).option(**"url"**, **"jdbc:mysql://master01:3306/rdd"**).option(**"dbtable"**, **"** **rddtable"**).option(**"user"**, **"root"**).option(**"password"**, **"hive"**).load()  **val** connectionProperties = **new** Properties() connectionProperties.put(**"user"**, **"root"**) connectionProperties.put(**"password"**, **"hive"**) **val** jdbcDF2 = spark.read .jdbc(**"jdbc:mysql://master01:3306/rdd"**, **"rddtable"**, connectionProperties)  *// Saving data to a JDBC source* jdbcDF.write .format(**"jdbc"**) .option(**"url"**, **"jdbc:mysql://master01:3306/rdd"**) .option(**"dbtable"**, **"rddtable2"**) .option(**"user"**, **"root"**) .option(**"password"**, **"hive"**) .save()  jdbcDF2.write .jdbc(**"jdbc:mysql://master01:3306/mysql"**, **"db"**, connectionProperties)  *// Specifying create table column data types on write* jdbcDF.write .option(**"createTableColumnTypes"**, **"name CHAR(64), comments VARCHAR(1024)"**) .jdbc(**"jdbc:mysql://master01:3306/mysql"**, **"db"**, connectionProperties) |

# JDBC/ODBC服务器

Spark SQL也提供JDBC连接支持，这对于让商业智能(BI)工具连接到Spark集群上以 及在多用户间共享一个集群的场景都非常有用。JDBC 服务器作为一个独立的 Spark 驱动 器程序运行，可以在多用户之间共享。任意一个客户端都可以在内存中缓存数据表，对表 进行查询。集群的资源以及缓存数据都在所有用户之间共享。

Spark SQL的JDBC服务器与Hive中的HiveServer2相一致。由于使用了Thrift通信协议，它也被称为“Thrift server”。

服务器可以通过 Spark 目录中的 sbin/start-thriftserver.sh 启动。这个 脚本接受的参数选项大多与 spark-submit 相同。默认情况下，服务器会在 localhost:10000 上进行监听，我们可以通过环境变量(HIVE\_SERVER2\_THRIFT\_PORT 和 HIVE\_SERVER2\_THRIFT\_BIND\_HOST)修改这些设置，也可以通过 Hive配置选项(hive. server2.thrift.port 和 hive.server2.thrift.bind.host)来修改。你也可以通过命令行参 数--hiveconf property=value来设置Hive选项。

|  |
| --- |
| ./sbin/start-thriftserver.sh \ --hiveconf hive.server2.thrift.port=<listening-port> \ --hiveconf hive.server2.thrift.bind.host=<listening-host> \ --master <**master-uri**> ... ./bin/beeline beeline> !connect jdbc:hive2://master01:10000 |

在 Beeline 客户端中，你可以使用标准的 HiveQL 命令来创建、列举以及查询数据表。

|  |
| --- |
| **[bigdata@master01 spark-2.1.1-bin-hadoop2.7]$ ./sbin/start-thriftserver.sh**  **starting org.apache.spark.sql.hive.thriftserver.HiveThriftServer2, logging to /home/bigdata/hadoop/spark-2.1.1-bin-hadoop2.7/logs/spark-bigdata-org.apache.spark.sql.hive.thriftserver.HiveThriftServer2-1-master01.out**  **[bigdata@master01 spark-2.1.1-bin-hadoop2.7]$ ./bin/beeline**  **Beeline version 1.2.1.spark2 by Apache Hive**  **beeline> !connect jdbc:hive2://master01:10000**  **Connecting to jdbc:hive2://master01:10000**  **Enter username for jdbc:hive2://master01:10000: bigdata**  **Enter password for jdbc:hive2://master01:10000: \*\*\*\*\*\*\***  **log4j:WARN No appenders could be found for logger (org.apache.hive.jdbc.Utils).**  **log4j:WARN Please initialize the log4j system properly.**  **log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.**  **Connected to: Spark SQL (version 2.1.1)**  **Driver: Hive JDBC (version 1.2.1.spark2)**  **Transaction isolation: TRANSACTION\_REPEATABLE\_READ**  **0: jdbc:hive2://master01:10000> show tables;**  **+-----------+------------+--------------+--+**  **| database | tableName | isTemporary |**  **+-----------+------------+--------------+--+**  **| default | src | false |**  **+-----------+------------+--------------+--+**  **1 row selected (0.726 seconds)**  **0: jdbc:hive2://master01:10000>** |

# 运行Spark SQL CLI

Spark SQL CLI可以很方便的在本地运行Hive元数据服务以及从命令行执行查询任务。需要注意的是，Spark SQL CLI不能与Thrift JDBC服务交互。  
在Spark目录下执行如下命令启动Spark SQL CLI：

./bin/spark-sql

配置Hive需要替换 conf/ 下的 hive-site.xml 。

# Spark SQL 的运行原理（了解）

## Spark SQL运行架构

Spark SQL对SQL语句的处理和关系型数据库类似，即词法/语法解析、绑定、优化、执行。Spark SQL会先将SQL语句解析成一棵树，然后使用规则(Rule)对Tree进行绑定、优化等处理过程。Spark SQL由Core、Catalyst、Hive、Hive-ThriftServer四部分构成：

Core: 负责处理数据的输入和输出，如获取数据，查询结果输出成DataFrame等

Catalyst: 负责处理整个查询过程，包括解析、绑定、优化等

Hive: 负责对Hive数据进行处理

Hive-ThriftServer: 主要用于对hive的访问

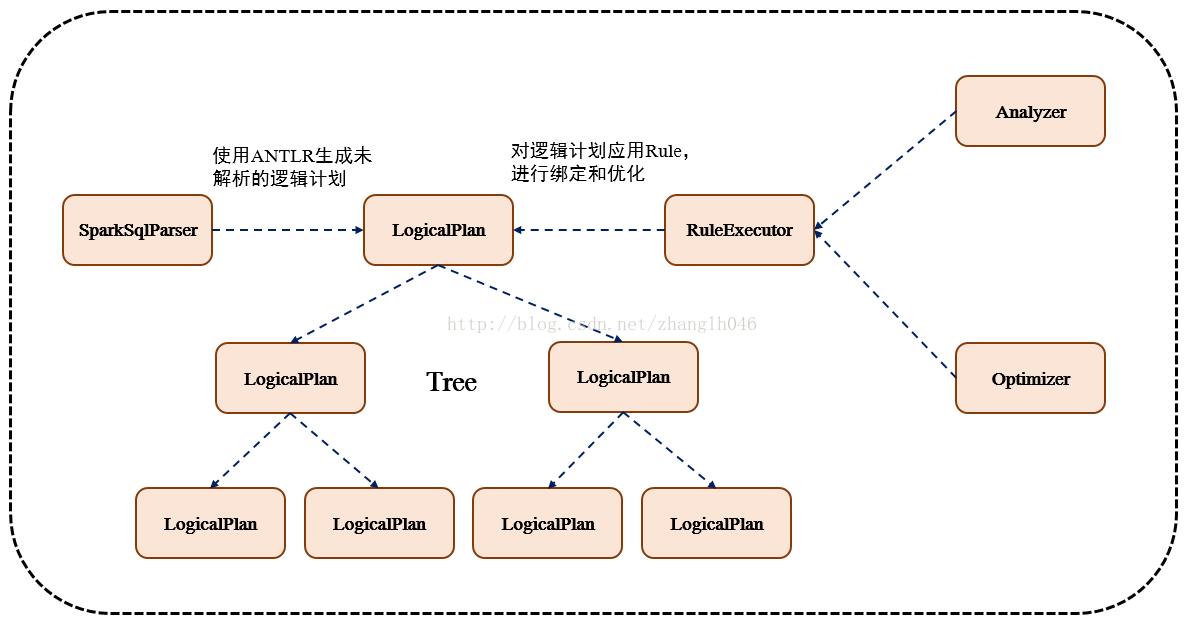
### TreeNode

逻辑计划、表达式等都可以用tree来表示，它只是在内存中维护，并不会进行磁盘的持久化，分析器和优化器对树的修改只是替换已有节点。

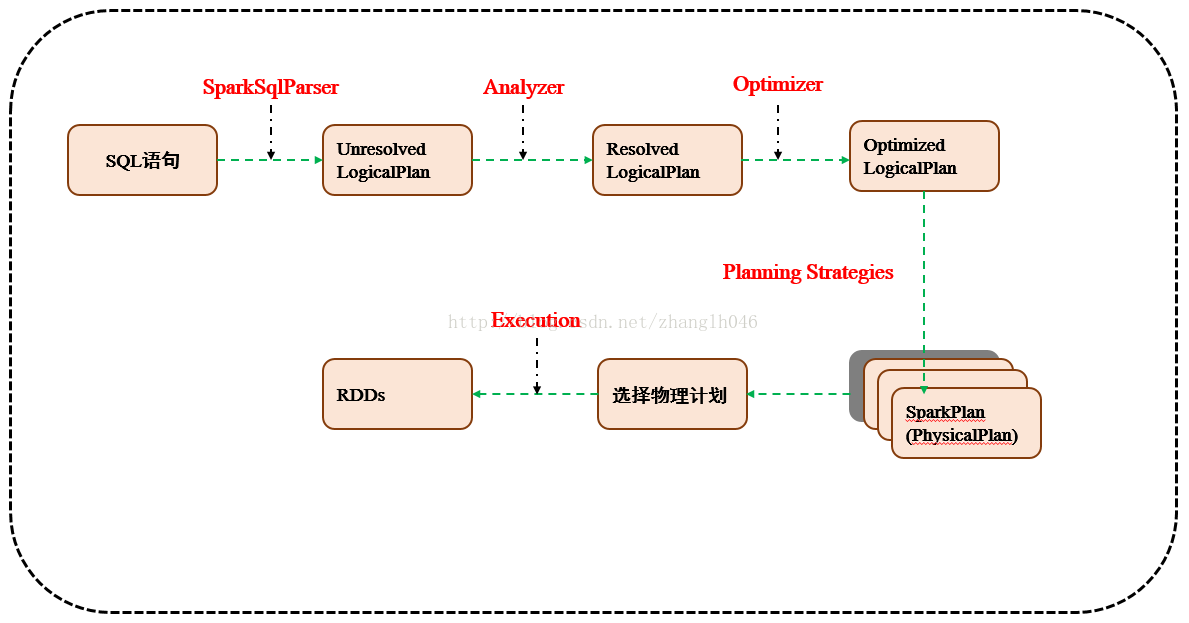
TreeNode有2个直接子类，QueryPlan和Expression。QueryPlam下又有LogicalPlan和SparkPlan. Expression是表达式体系，不需要执行引擎计算而是可以直接处理或者计算的节点，包括投影操作，操作符运算等

### Rule & RuleExecutor

Rule就是指对逻辑计划要应用的规则，以到达绑定和优化。他的实现类就是RuleExecutor。优化器和分析器都需要继承RuleExecutor。每一个子类中都会定义Batch、Once、FixPoint. 其中每一个Batch代表着一套规则，Once表示对树进行一次操作，FixPoint表示对树进行多次的迭代操作。RuleExecutor内部提供一个Seq[Batch]属性，里面定义的是RuleExecutor的处理逻辑，具体的处理逻辑由具体的Rule子类实现。



整个流程架构图：



## Spark SQL运行原理

### 使用SessionCatalog保存元数据

在解析SQL语句之前，会创建SparkSession，或者如果是2.0之前的版本初始化SQLContext，SparkSession只是封装了SparkContext和SQLContext的创建而已。会把元数据保存在SessionCatalog中，涉及到表名，字段名称和字段类型。创建临时表或者视图，其实就会往SessionCatalog注册

### 解析SQL,使用ANTLR生成未绑定的逻辑计划

当调用SparkSession的sql或者SQLContext的sql方法，我们以2.0为准，就会使用SparkSqlParser进行解析SQL. 使用的ANTLR进行词法解析和语法解析。它分为2个步骤来生成Unresolved LogicalPlan：

# 词法分析：Lexical Analysis，负责将token分组成符号类

# 构建一个分析树或者语法树AST

### 使用分析器Analyzer绑定逻辑计划

在该阶段，Analyzer会使用Analyzer Rules，并结合SessionCatalog，对未绑定的逻辑计划进行解析，生成已绑定的逻辑计划。

### 使用优化器Optimizer优化逻辑计划

优化器也是会定义一套Rules，利用这些Rule对逻辑计划和Exepression进行迭代处理，从而使得树的节点进行和并和优化

### 使用SparkPlanner生成物理计划

SparkSpanner使用Planning Strategies，对优化后的逻辑计划进行转换，生成可以执行的物理计划SparkPlan.

### 使用QueryExecution执行物理计划

此时调用SparkPlan的execute方法，底层其实已经再触发JOB了，然后返回RDD

# 数据仓库的建设（了解）

## 什么是数据仓库

数据仓库，英文名称为Data Warehouse，可简写为DW或DWH。数据仓库，是为企业所有级别的决策制定过程，提供所有类型数据支持的战略集合。它出于分析性报告和决策支持目的而创建。 为需要业务智能的企业，提供指导业务流程改进、监视时间、成本、质量以及控制。

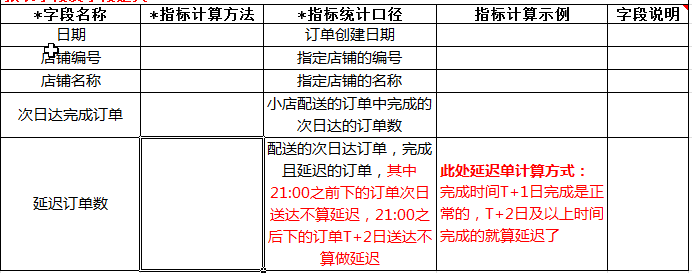
数据仓库能干什么？

1、年度销售目标的指定，需要根据以往的历史报表进行决策，不能拍脑袋。

2、如何优化业务流程

案例1：

一个电商网站订单的完成包括：浏览、下单、支付、物流，其中物流环节可能和中通、申通、韵达等快递公司合作。快递公司每派送一个订单，都会有订单派送的确认时间，可以根据订单派送时间来分析哪个快递公司比较快捷高效，从而选择与哪些快递公司合作，剔除哪些快递公司，增加用户友好型。



案例2：

互联网中国需要对APP进行推广，考核的主要目标是下载安装，有些第三方渠道会对这些数据造假，比如某个渠道在凌晨批量下载，点赞操作，操作步骤一致。通过数据分析，分析出应用的名称和安装时间，来判断一个渠道的是否优质、是否作假。

## 数据仓库的特点

**1．数据仓库的数据是面向主题的**

与传统数据库面向应用进行数据组织的特点相对应，数据仓库中的数据是面向主题进行组织的。什么是主题呢？首先，主题是一个抽象的概念，是较高层次上企业信息系统中的数据综合、归类并进行分析利用的抽象。在逻辑意义上，它是对应企业中某一宏观分析领域所涉及的分析对象。面向主题的数据组织方式，就是在较高层次上对分析对象的数据的一个完整、一致的描述，能完整、统一地刻划各个分析对象所涉及的企业的各项数据，以及数据之间的联系。所谓较高层次是相对面向应用的数据组织方式而言的，是指按照主题进行数据组织的方式具有更高的数据抽象级别。

**2. 数据仓库的数据是集成的**

数据仓库的数据是从原有的分散的数据库数据抽取来的。操作型数据与DSS分析型数据之间差别甚大。第一，数据仓库的每一个主题所对应的源数据在原有的各分散数据库中有许多重复和不一致的地方，且来源于不同的联机系统的数据都和不同的应用逻辑捆绑在一起；第二，数据仓库中的综合数据不能从原有的数据库系统直接得到。因此在数据进入数据仓库之前，必然要经过统一与综合，这一步是数据仓库建设中最关键、最复杂的一步，所要完成的工作有：  
    （1）要统一源数据中所有矛盾之处，如字段的同名异义、异名同义、单位不统一、字长不一致，等等。  
    （2）进行数据综合和计算。数据仓库中的数据综合工作可以在从原有数据库抽取 数据时生成，但许多是在数据仓库内部生成的，即进入数据仓库以后进行综合生成的。

**3. 数据仓库的数据是不可更新的**

数据仓库的数据主要供企业决策分析之用，所涉及的数据操作主要是数据查询，一般情况下并不进行修改操作。数据仓库的数据反映的是一段相当长的时间内历史数据的内容，是不同时点的数据库快照的集合，以及基于这些快照进行统计、综合和重组的导出数据，而不是联机处理的数据。数据库中进行联机处理的数据经过集成输入到数据仓库中，一旦数据仓库存放的数据已经超过数据仓库的数据存储期限，这些数据将从当前的数据仓库中删去。因为数据仓库只进行数据查询操作，所以数据仓库管理系统相比数据库管理系统而言要简单得多。数据库管理系统中许多技术难点，如完整性保护、并发控制等等，在数据仓库的管理中几乎可以省去。但是由于数据仓库的查询数据量往往很大，所以就对数据查询提出了更高的要求，它要求采用各种复杂的索引技术；同时由于数据仓库面向的是商业企业的高层管理者，他们会对数据查询的界面友好性和数据表示提出更高的要求。

**4. 数据仓库的数据是随时间不断变化的**

数据仓库中的数据不可更新是针对应用来说的，也就是说，数据仓库的用户进行分析处理时是不进行数据更新操作的。但并不是说，在从数据集成输入数据仓库开始到最终被删除的整个数据生存周期中，所有的数据仓库数据都是永远不变的。  
    数据仓库的数据是随时间的变化而不断变化的，这是数据仓库数据的第四个特征。这一特征表现在以下3方面：  
    （1）数据仓库随时间变化不断增加新的数据内容。数据仓库系统必须不断捕捉OLTP数据库中变化的数据，追加到数据仓库中去，也就是要不断地生成OLTP数据库的快照，经统一集成后增加到数据仓库中去；但对于确实不再变化的数据库快照，如果捕捉到新的变化数据，则只生成一个新的数据库快照增加进去，而不会对原有的数据库快照进行修改。  
    （2）数据仓库随时间变化不断删去旧的数据内容。数据仓库的数据也有存储期限，一旦超过了这一期限，过期数据就要被删除。只是数据仓库内的数据时限要远远长于操作型环境中的数据时限。在操作型环境中一般只保存有60~90天的数据，而在数据仓库中则需要保存较长时限的数据（如5~10年），以适应DSS进行趋势分析的要求。  
    （3）数据仓库中包含有大量的综合数据，这些综合数据中很多跟时间有关，如数据经常按照时间段进行综合，或隔一定的时间片进行抽样等等。这些数据要随着时间的变化不断地进行重新综合。因此，数据仓库的数据特征都包含时间项，以标明数据的历史时期。

## 数据仓库发展历程

**数据仓库的发展大致经历了这样的三个过程：**

* **简单报表阶段：**这个阶段，系统的主要目标是解决一些日常的工作中业务人员需要的报表，以及生成一些简单的能够帮助领导进行决策所需要的汇总数据。这个阶段的大部分表现形式为数据库和前端报表工具。
* **数据集市阶段：**这个阶段，主要是根据某个业务部门的需要，进行一定的数据的采集，整理，按照业务人员的需要，进行多维报表的展现，能够提供对特定业务指导的数据，并且能够提供特定的领导决策数据。
* **数据仓库阶段：**这个阶段，主要是按照一定的数据模型，对整个企业的数据进行采集，整理，并且能够按照各个业务部门的需要，提供跨部门的，完全一致的业务报表数据，能够通过数据仓库生成对对业务具有指导性的数据，同时，为领导决策提供全面的数据支持。

通过数据仓库建设的发展阶段，我们能够看出，数据仓库的建设和数据集市的建设的重要区别就在于数据模型的支持。因此，数据模型的建设，对于我们数据仓库的建设，有着决定性的意义。

## 数据库与数据仓库的区别

了解数据库与数据仓库的区别之前，首先掌握三个概念。数据库软件、数据库、数据仓库。

数据库软件：是一种软件，可以看得见，可以操作。用来实现数据库逻辑功能。属于物理层。

数据库：是一种逻辑概念，用来存放数据的仓库。通过数据库软件来实现。数据库由很多表组成，表是二维的，一张表里可以有很多字段。字段一字排开，对应的数据就一行一行写入表中。数据库的表，在于能够用二维表现多维关系。目前市面上流行的数据库都是二维数据库。如：Oracle、DB2、MySQL、Sybase、MS SQL Server等。

数据仓库：是数据库概念的升级。从逻辑上理解，数据库和数据仓库没有区别，都是通过数据库软件实现的存放数据的地方，只不过从数据量来说，数据仓库要比数据库更庞大得多。数据仓库主要用于数据挖掘和数据分析，辅助领导做决策。

在IT的架构体系中，数据库是必须存在的。必须要有地方存放数据。比如现在的网购，淘宝，京东等等。物品的存货数量，货品的价格，用户的账户余额之类的。这些数据都是存放在后台数据库中。或者最简单理解，我们现在微博，QQ等账户的用户名和密码。在后台数据库必然有一张user表，字段起码有两个，即用户名和密码，然后我们的数据就一行一行的存在表上面。当我们登录的时候，我们填写了用户名和密码，这些数据就会被传回到后台去，去跟表上面的数据匹配，匹配成功了，你就能登录了。匹配不成功就会报错说密码错误或者没有此用户名等。这个就是数据库，数据库在生产环境就是用来干活的。凡是跟业务应用挂钩的，我们都使用数据库。

数据仓库则是BI下的其中一种技术。由于数据库是跟业务应用挂钩的，所以一个数据库不可能装下一家公司的所有数据。数据库的表设计往往是针对某一个应用进行设计的。比如刚才那个登录的功能，这张user表上就只有这两个字段，没有别的字段了。但是这张表符合应用，没有问题。但是这张表不符合分析。比如我想知道在哪个时间段，用户登录的量最多？哪个用户一年购物最多？诸如此类的指标。那就要重新设计数据库的表结构了。对于数据分析和数据挖掘，我们引入数据仓库概念。数据仓库的表结构是依照分析需求，分析维度，分析指标进行设计的。

数据库与数据仓库的区别实际讲的是OLTP与OLAP的区别。

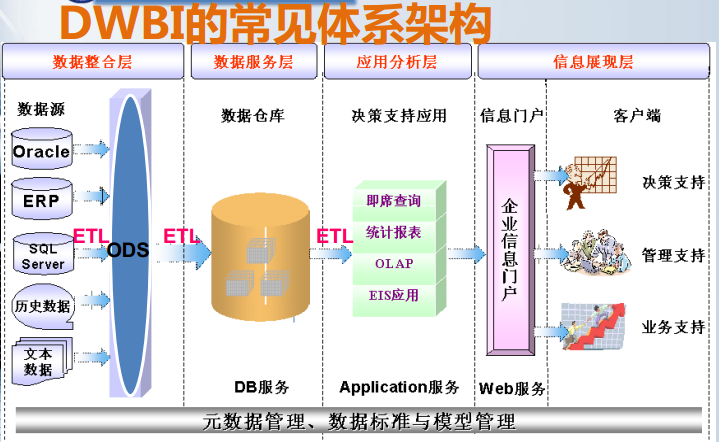
操作型处理，叫联机事务处理OLTP（On-Line Transaction Processing，），也可以称面向交易的处理系统，它是针对具体业务在数据库联机的日常操作，通常对少数记录进行查询、修改。用户较为关心操作的响应时间、数据的安全性、完整性和并发支持的用户数等问题。传统的数据库系统作为数据管理的主要手段，主要用于操作型处理。

分析型处理，叫联机分析处理OLAP（On-Line Analytical Processing）一般针对某些主题的历史数据进行分析，支持管理决策。



## 数据仓库架构分层

数据仓库标准上可以分为四层：ODS（临时存储层）、PDW（数据仓库层）、DM（数据集市层）、APP（应用层）。



ODS层：

为临时存储层，是接口数据的临时存储区域，为后一步的数据处理做准备。一般来说ODS层的数据和源系统的数据是同构的，主要目的是简化后续数据加工处理的工作。从数据粒度上来说ODS层的数据粒度是最细的。ODS层的表通常包括两类，一个用于存储当前需要加载的数据，一个用于存储处理完后的历史数据。历史数据一般保存3-6个月后需要清除，以节省空间。但不同的项目要区别对待，如果源系统的数据量不大，可以保留更长的时间，甚至全量保存；

PDW层：

为数据仓库层，PDW层的数据应该是一致的、准确的、干净的数据，即对源系统数据进行了清洗（去除了杂质）后的数据。这一层的数据一般是遵循数据库第三范式的，其数据粒度通常和ODS的粒度相同。在PDW层会保存BI系统中所有的历史数据，例如保存10年的数据。

DM层：

为数据集市层，这层数据是面向主题来组织数据的，通常是星形或雪花结构的数据。从数据粒度来说，这层的数据是轻度汇总级的数据，已经不存在明细数据了。从数据的时间跨度来说，通常是PDW层的一部分，主要的目的是为了满足用户分析的需求，而从分析的角度来说，用户通常只需要分析近几年（如近三年的数据）的即可。从数据的广度来说，仍然覆盖了所有业务数据。

APP层：

为应用层，这层数据是完全为了满足具体的分析需求而构建的数据，也是星形或雪花结构的数据。从数据粒度来说是高度汇总的数据。从数据的广度来说，则并不一定会覆盖所有业务数据，而是DM层数据的一个真子集，从某种意义上来说是DM层数据的一个重复。从极端情况来说，可以为每一张报表在APP层构建一个模型来支持，达到以空间换时间的目的数据仓库的标准分层只是一个建议性质的标准，实际实施时需要根据实际情况确定数据仓库的分层，不同类型的数据也可能采取不同的分层方法。

为什么要对数据仓库分层：

1用空间换时间，通过大量的预处理来提升应用系统的用户体验（效率），因此数据仓库会存在大量冗余的数据；

2如果不分层的话，如果源业务系统的业务规则发生变化将会影响整个数据清洗过程，工作量巨大

3通过数据分层管理可以简化数据清洗的过程，因为把原来一步的工作分到了多个步骤去完成，相当于把一个复杂的工作拆成了多个简单的工作，把一个大的黑盒变成了一个白盒，每一层的处理逻辑都相对简单和容易理解，这样我们比较容易保证每一个步骤的正确性，当数据发生错误的时候，往往我们只需要局部调整某个步骤即可。

## 数据质量检查

保证报表数据的正确性、稳定性，通过告警机制尽可能快的发现异常、尽可能快的解决问题。

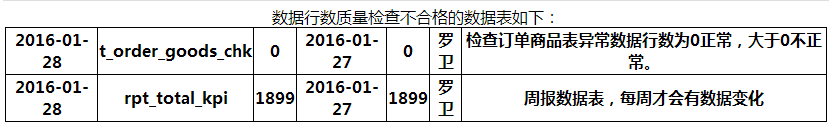
出错的次数太多之后，领导会对你失去信心，该辞职了。

检查方法：

1、数据行数据的比较

2、行数有变化，但是指标有变化。对领导关系的重点指标进行筛选。





在领导发现问题之前，解决问题。

告警

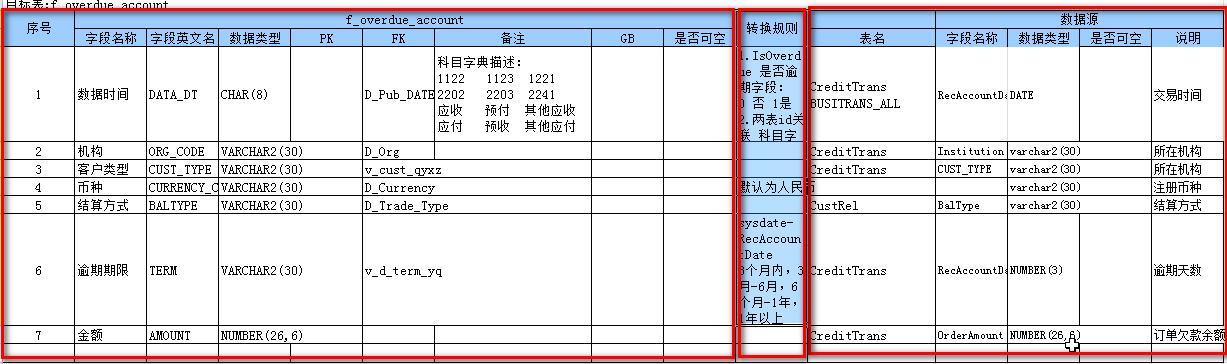


## 元数据介绍

当需要了解某地企业及其提供的服务时，电话黄页的重要性就体现出来了。元数据（Metadata）类似于这样的电话黄页。

1．元数据的定义

数据仓库的元数据是关于数据仓库中数据的数据。它的作用类似于数据库管理系统的数据字典，保存了逻辑数据结构、文件、地址和索引等信息。广义上讲，在数据仓库中，元数据描述了数据仓库内数据的结构和建立方法的数据。



元数据是数据仓库管理系统的重要组成部分，元数据管理器是企业级数据仓库中的关键组件，贯穿数据仓库构建的整个过程，直接影响着数据仓库的构建、使用和维护。

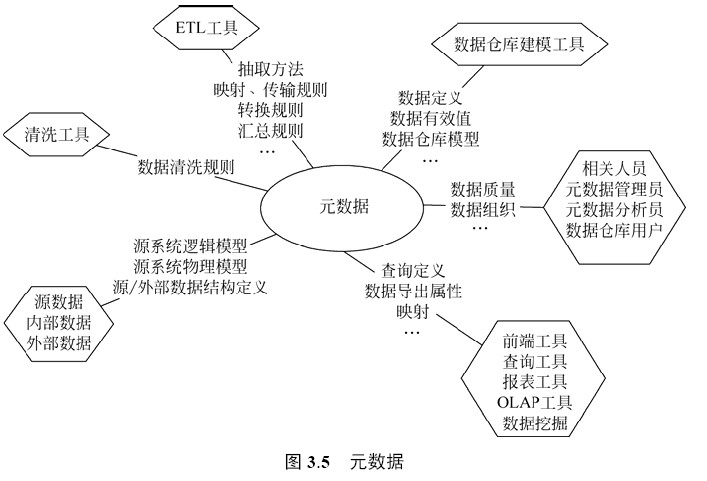
（1）构建数据仓库的主要步骤之一是ETL。这时元数据将发挥重要的作用，它定义了源数据系统到数据仓库的映射、数据转换的规则、数据仓库的逻辑结构、数据更新的规则、数据导入历史记录以及装载周期等相关内容。数据抽取和转换的专家以及数据仓库管理员正是通过元数据高效地构建数据仓库。

（2）用户在使用数据仓库时，通过元数据访问数据，明确数据项的含义以及定制报表。

（3）数据仓库的规模及其复杂性离不开正确的元数据管理，包括增加或移除外部数据源，改变数据清洗方法，控制出错的查询以及安排备份等。

元数据可分为技术元数据和业务元数据。技术元数据为开发和管理数据仓库的IT 人员使用，它描述了与数据仓库开发、管理和维护相关的数据，包括数据源信息、数据转换描述、数据仓库模型、数据清洗与更新规则、数据映射和访问权限等。而业务元数据为管理层和业务分析人员服务，从业务角度描述数据，包括商务术语、数据仓库中有什么数据、数据的位置和数据的可用性等，帮助业务人员更好地理解数据仓库中哪些数据是可用的以及如何使用。

由上可见，元数据不仅定义了数据仓库中数据的模式、来源、抽取和转换规则等，而且是整个数据仓库系统运行的基础，元数据把数据仓库系统中各个松散的组件联系起来，组成了一个有机的整体，如图3.5 所示



2．元数据的存储方式

元数据有两种常见存储方式：一种是以数据集为基础，每一个数据集有对应的元数据文件，每一个元数据文件包含对应数据集的元数据内容；另一种存储方式是以数据库为基础，即元数据库。其中元数据文件由若干项组成，每一项表示元数据的一个要素，每条记录为数据集的元数据内容。上述存储方式各有优缺点，第一种存储方式的优点是调用数据时相应的元数据也作为一个独立的文件被传输，相对数据库有较强的独立性，在对元数据进行检索时可以利用数据库的功能实现，也可以把元数据文件调到其他数据库系统中操作；不足是如果每一数据集都对应一个元数据文档，在规模巨大的数据库中则会有大量的元数据文件，管理不方便。第二种存储方式下，元数据库中只有一个元数据文件，管理比较方便，添加或删除数据集，只要在该文件中添加或删除相应的记录项即可。在获取某数据集的元数据时，因为实际得到的只是关系表格数据的一条记录，所以要求用户系统可以接受这种特定形式的数据。因此推荐使用元数据库的方式。

元数据库用于存储元数据，因此元数据库最好选用主流的关系数据库管理系统。元数据库还包含用于操作和查询元数据的机制。建立元数据库的主要好处是提供统一的数据结构和业务规则，易于把企业内部的多个数据集市有机地集成起来。目前，一些企业倾向建立多个数据集市，而不是一个集中的数据仓库，这时可以考虑在建立数据仓库（或数据集市）之前，先建立一个用于描述数据、服务应用集成的元数据库，做好数据仓库实施的初期支持工作，对后续开发和维护有很大的帮助。元数据库保证了数据仓库数据的一致性和准确性，为企业进行数据质量管理提供基础。

3．元数据的作用

在数据仓库中，元数据的主要作用如下。

（1）描述哪些数据在数据仓库中，帮助决策分析者对数据仓库的内容定位。

（2）定义数据进入数据仓库的方式，作为数据汇总、映射和清洗的指南。

（3）记录业务事件发生而随之进行的数据抽取工作时间安排。

（4）记录并检测系统数据一致性的要求和执行情况。

（5）评估数据质量。

## 什么是数据模型

数据模型是抽象描述现实世界的一种工具和方法，是通过抽象的实体及实体之间联系的形式，来表示现实世界中事务的相互关系的一种映射。在这里，数据模型表现的抽象的是实体和实体之间的关系，通过对实体和实体之间关系的定义和描述，来表达实际的业务中具体的业务关系。

数据仓库模型是数据模型中针对特定的数据仓库应用系统的一种特定的数据模型，一般的来说，我们数据仓库模型分为几下几个层次，如图 2 所示。

**图 2. 数据仓库模型**



通过上面的图形，我们能够很容易的看出在整个数据仓库得建模过程中，我们需要经历一般四个过程：

* **业务建模**，生成业务模型，主要解决业务层面的分解和程序化。
* **领域建模**，生成领域模型，主要是对业务模型进行抽象处理，生成领域概念模型。
* **逻辑建模**，生成逻辑模型，主要是将领域模型的概念实体以及实体之间的关系进行数据库层次的逻辑化。
* **物理建模**，生成物理模型，主要解决，逻辑模型针对不同关系型数据库的物理化以及性能等一些具体的技术问题。

因此，**在整个数据仓库的模型的设计和架构中，既涉及到业务知识，也涉及到了具体的技术，我们既需要了解丰富的行业经验，同时，也需要一定的信息技术来帮助我们实现我们的数据模型，最重要的是，我们还需要一个非常适用的方法论，来指导我们自己针对我们的业务进行抽象，处理，生成各个阶段的模型。**

## 为什么需要数据仓库模型

在数据仓库的建设中，我们一再强调需要数据模型，那么数据模型究竟为什么这么重要呢？首先我们需要了解整个数据仓库的建设的发展史。

**数据仓库的发展大致经历了这样的三个过程：**

* **简单报表阶段：**这个阶段，系统的主要目标是解决一些日常的工作中业务人员需要的报表，以及生成一些简单的能够帮助领导进行决策所需要的汇总数据。这个阶段的大部分表现形式为数据库和前端报表工具。
* **数据集市阶段：**这个阶段，主要是根据某个业务部门的需要，进行一定的数据的采集，整理，按照业务人员的需要，进行多维报表的展现，能够提供对特定业务指导的数据，并且能够提供特定的领导决策数据。
* **数据仓库阶段：**这个阶段，主要是按照一定的数据模型，对整个企业的数据进行采集，整理，并且能够按照各个业务部门的需要，提供跨部门的，完全一致的业务报表数据，能够通过数据仓库生成对对业务具有指导性的数据，同时，为领导决策提供全面的数据支持。

通过数据仓库建设的发展阶段，我们能够看出，数据仓库的建设和数据集市的建设的重要区别就在于数据模型的支持。因此，数据模型的建设，对于我们数据仓库的建设，有着决定性的意义。

**一般来说，数据模型的建设主要能够帮助我们解决以下的一些问题：**

* **进行全面的业务梳理，改进业务流程。**在业务模型建设的阶段，能够帮助我们的企业或者是管理机关对本单位的业务进行全面的梳理。通过业务模型的建设，我们应该能够全面了解该单位的业务架构图和整个业务的运行情况，能够将业务按照特定的规律进行分门别类和程序化，同时，帮助我们进一步的改进业务的流程，提高业务效率，指导我们的业务部门的生产。
* **建立全方位的数据视角，消灭信息孤岛和数据差异。**通过数据仓库的模型建设，能够为企业提供一个整体的数据视角，不再是各个部门只是关注自己的数据，而且通过模型的建设，勾勒出了部门之间内在的联系，帮助消灭各个部门之间的信息孤岛的问题，更为重要的是，通过数据模型的建设，能够保证整个企业的数据的一致性，各个部门之间数据的差异将会得到有效解决。
* **解决业务的变动和数据仓库的灵活性。**通过数据模型的建设，能够很好的分离出底层技术的实现和上层业务的展现。当上层业务发生变化时，通过数据模型，底层的技术实现可以非常轻松的完成业务的变动，从而达到整个数据仓库系统的灵活性。
* **帮助数据仓库系统本身的建设。**通过数据仓库的模型建设，开发人员和业务人员能够很容易的达成系统建设范围的界定，以及长期目标的规划，从而能够使整个项目组明确当前的任务，加快整个系统建设的速度。

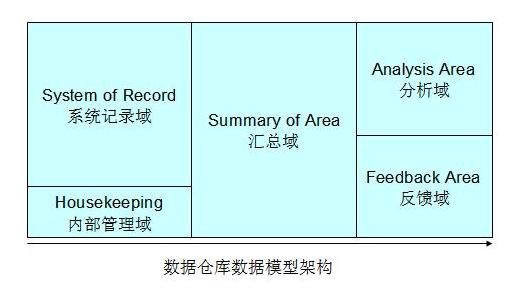
## 如何建设数据仓库模型

建设数据模型既然是整个数据仓库建设中一个非常重要的关键部分，那么，怎么建设我们的数据仓库模型就是我们需要解决的一个问题。这里我们将要详细介绍如何创建适合自己的数据模型。

### 数据仓库数据模型架构

数据仓库的数据模型的架构和数据仓库的整体架构是紧密关联在一起的，我们首先来了解一下整个数据仓库的数据模型应该包含的几个部分。从下图我们可以很清楚地看到，整个数据模型的架构分成 5 大部分，每个部分其实都有其独特的功能。

**图 3. 数据仓库数据模型架构**



从上图我们可以看出，**整个数据仓库的数据模型可以分为大概 5 大部分**：

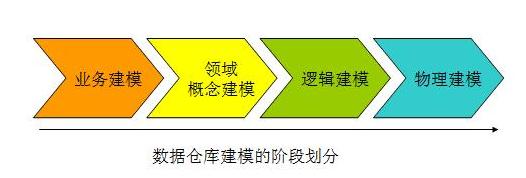
* **系统记录域（System of Record）：**这部分是主要的数据仓库业务数据存储区，数据模型在这里保证了数据的一致性。
* **内部管理域（Housekeeping）：**这部分主要存储数据仓库用于内部管理的元数据，数据模型在这里能够帮助进行统一的元数据的管理。
* **汇总域（Summary of Area）：**这部分数据来自于系统记录域的汇总，数据模型在这里保证了分析域的主题分析的性能，满足了部分的报表查询。
* **分析域（Analysis Area）：**这部分数据模型主要用于各个业务部分的具体的主题业务分析。这部分数据模型可以单独存储在相应的数据集市中。
* **反馈域（Feedback Area）：**可选项，这部分数据模型主要用于相应前端的反馈数据，数据仓库可以视业务的需要设置这一区域。

通过对整个数据仓库模型的数据区域的划分，我们可以了解到，一个好的数据模型，不仅仅是对业务进行抽象划分，而且对实现技术也进行具体的指导，它应该涵盖了从业务到实现技术的各个部分。

### 数据仓库建模阶段划分

我们前面介绍了数据仓库模型的几个层次，下面我们讲一下，针对这几个层次的不同阶段的数据建模的工作的主要内容：

**图 4. 数据仓库建模阶段划分**



从上图我们可以清楚地看出，数据仓库的数据建模大致分为四个阶段：

1.   **业务建模**，这部分建模工作，主要包含以下几个部分：

* 划分整个单位的业务，一般按照业务部门的划分，进行各个部分之间业务工作的界定，理清各业务部门之间的关系。
* 深入了解各个业务部门的内具体业务流程并将其程序化。
* 提出修改和改进业务部门工作流程的方法并程序化。
* 数据建模的范围界定，整个数据仓库项目的目标和阶段划分。

2.   **领域概念建模**，这部分得建模工作，主要包含以下几个部分：

* 抽取关键业务概念，并将之抽象化。
* 将业务概念分组，按照业务主线聚合类似的分组概念。
* 细化分组概念，理清分组概念内的业务流程并抽象化。
* 理清分组概念之间的关联，形成完整的领域概念模型。

3.   **逻辑建模**，这部分的建模工作，主要包含以下几个部分：

* 业务概念实体化，并考虑其具体的属性
* 事件实体化，并考虑其属性内容
* 说明实体化，并考虑其属性内容

4.   **物理建模**，这部分得建模工作，主要包含以下几个部分：

* 针对特定物理化平台，做出相应的技术调整
* 针对模型的性能考虑，对特定平台作出相应的调整
* 针对管理的需要，结合特定的平台，做出相应的调整
* 生成最后的执行脚本，并完善之。

从我们上面对数据仓库的数据建模阶段的各个阶段的划分，我们能够了解到整个数据仓库建模的主要工作和工作量，希望能够对我们在实际的项目建设能够有所帮助。

### 数据仓库建模方法

大千世界，表面看五彩缤纷，实质上，万物都遵循其自有的法则。数据仓库的建模方法同样也有很多种，每一种建模方法其实代表了哲学上的一个观点，代表了一种归纳，概括世界的一种方法。目前业界较为流行的数据仓库的建模方法非常多，这里主要介绍范式建模法，维度建模法，实体建模法等几种方法，每种方法其实从本质上讲就是从不同的角度看我们业务中的问题，不管从技术层面还是业务层面，其实代表的是哲学上的一种世界观。我们下面给大家详细介绍一下这些建模方法。

**1. 范式建模法（Third Normal Form，3NF）**

范式建模法其实是我们在构建数据模型常用的一个方法，该方法的主要由 Inmon 所提倡，主要解决关系型数据库得数据存储，利用的一种技术层面上的方法。目前，我们在关系型数据库中的建模方法，大部分采用的是三范式建模法。

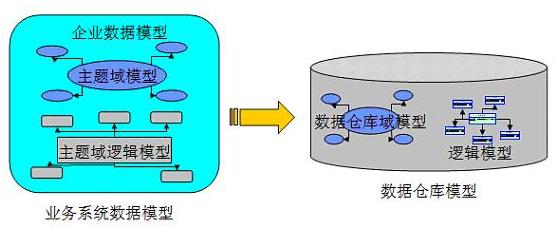
范式是数据库逻辑模型设计的基本理论，一个关系模型可以从第一范式到第五范式进行无损分解，这个过程也可称为规范化。在数据仓库的模型设计中目前一般采用第三范式，它有着严格的数学定义。从其表达的含义来看，**一个符合第三范式的关系必须具有以下三个条件 :**

* 每个属性值唯一，不具有多义性 ;
* 每个非主属性必须完全依赖于整个主键，而非主键的一部分 ;
* 每个非主属性不能依赖于其他关系中的属性，因为这样的话，这种属性应该归到其他关系中去。

由于范式是基于整个关系型数据库的理论基础之上发展而来的，因此，本人在这里不多做介绍，有兴趣的读者可以通过阅读相应的材料来获得这方面的知识。

根据 Inmon 的观点，数据仓库模型得建设方法和业务系统的企业数据模型类似。在业务系统中，企业数据模型决定了数据的来源，而企业数据模型也分为两个层次，即主题域模型和逻辑模型。同样，主题域模型可以看成是业务模型的概念模型，而逻辑模型则是域模型在关系型数据库上的实例。

**图 5. 范式建模法**



从业务数据模型转向数据仓库模型时，同样也需要有数据仓库的域模型，即概念模型，同时也存在域模型的逻辑模型。这里，**业务模型中的数据模型和数据仓库的模型稍微有一些不同。主要区别在于：**

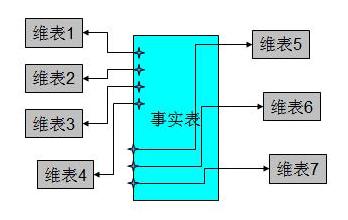
* 数据仓库的域模型应该包含企业数据模型的域模型之间的关系，以及各主题域定义。数据仓库的域模型的概念应该比业务系统的主题域模型范围更加广。
* 在数据仓库的逻辑模型需要从业务系统的数据模型中的逻辑模型中抽象实体，实体的属性，实体的子类，以及实体的关系等。

Inmon 的范式建模法的最大优点就是从关系型数据库的角度出发，结合了业务系统的数据模型，能够比较方便的实现数据仓库的建模。但其缺点也是明显的，由于建模方法限定在关系型数据库之上，在某些时候反而限制了整个数据仓库模型的灵活性，性能等，特别是考虑到数据仓库的底层数据向数据集市的数据进行汇总时，需要进行一定的变通才能满足相应的需求。因此，笔者建议读者们在实际的使用中，参考使用这一建模方式。

**2. 维度建模法**

维度建模法，Kimball 最先提出这一概念。其最简单的描述就是，按照事实表，维表来构建数据仓库，数据集市。这种方法的最被人广泛知晓的名字就是星型模式（Star-schema）。

**图 6. 维度建模法**



上图的这个架构中是典型的星型架构。星型模式之所以广泛被使用，在于针对各个维作了大量的预处理，如按照维进行预先的统计、分类、排序等。通过这些预处理，能够极大的提升数据仓库的处理能力。特别是**针对 3NF 的建模方法，星型模式在性能上占据明显的优势。**

同时，维度建模法的另外一个优点是，维度建模非常直观，紧紧围绕着业务模型，可以直观的反映出业务模型中的业务问题。不需要经过特别的抽象处理，即可以完成维度建模。这一点也是维度建模的优势。

但是，维度建模法的缺点也是非常明显的，由于在构建星型模式之前需要进行大量的数据预处理，因此会导致大量的数据处理工作。而且，当业务发生变化，需要重新进行维度的定义时，往往需要重新进行维度数据的预处理。而在这些与处理过程中，往往会导致大量的数据冗余。

另外一个维度建模法的缺点就是，如果只是依靠单纯的维度建模，不能保证数据来源的一致性和准确性，而且在数据仓库的底层，不是特别适用于维度建模的方法。

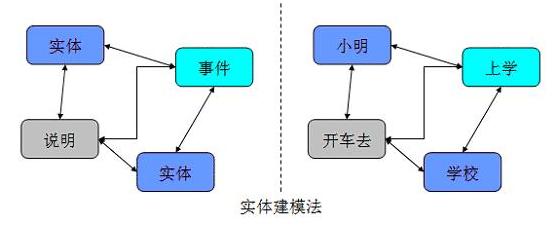
因此以笔者的观点看，维度建模的领域主要适用与数据集市层，它的最大的作用其实是为了解决数据仓库建模中的性能问题。维度建模很难能够提供一个完整地描述真实业务实体之间的复杂关系的抽象方法。

**3. 实体建模法**

实体建模法并不是数据仓库建模中常见的一个方法，它来源于哲学的一个流派。从哲学的意义上说，客观世界应该是可以细分的，客观世界应该可以分成由一个个实体，以及实体与实体之间的关系组成。那么我们在数据仓库的建模过程中完全可以引入这个抽象的方法，将整个业务也可以划分成一个个的实体，而每个实体之间的关系，以及针对这些关系的说明就是我们数据建模需要做的工作。

虽然实体法粗看起来好像有一些抽象，其实理解起来很容易。即我们可以将任何一个业务过程划分成 3 个部分，实体，事件和说明，如下图所示：

**图 7. 实体建模法**



上图表述的是一个抽象的含义，如果我们描述一个简单的事实：“小明开车去学校上学”。以这个业务事实为例，我们可以把“小明”，“学校”看成是一个实体，“上学”描述的是一个业务过程，我们在这里可以抽象为一个具体“事件”，而“开车去”则可以看成是事件“上学”的一个说明。

从上面的举例我们可以了解，我们使用的抽象归纳方法其实很简单，**任何业务可以看成 3 个部分：**

* **实体**，主要指领域模型中特定的概念主体，指发生业务关系的对象。
* **事件**，主要指概念主体之间完成一次业务流程的过程，特指特定的业务过程。
* **说明**，主要是针对实体和事件的特殊说明。

由于实体建模法，能够很轻松的实现业务模型的划分，因此，在业务建模阶段和领域概念建模阶段，实体建模法有着广泛的应用。从笔者的经验来看，再没有现成的行业模型的情况下，我们可以采用实体建模的方法，和客户一起理清整个业务的模型，进行领域概念模型的划分，抽象出具体的业务概念，结合客户的使用特点，完全可以创建出一个符合自己需要的数据仓库模型来。

但是，实体建模法也有着自己先天的缺陷，由于实体说明法只是一种抽象客观世界的方法，因此，注定了该建模方法只能局限在业务建模和领域概念建模阶段。因此，到了逻辑建模阶段和物理建模阶段，则是范式建模和维度建模发挥长处的阶段。

因此，笔者建议读者在创建自己的数据仓库模型的时候，可以参考使用上述的三种数据仓库得建模方法，在各个不同阶段采用不同的方法，从而能够保证整个数据仓库建模的质量。

## 维度建模

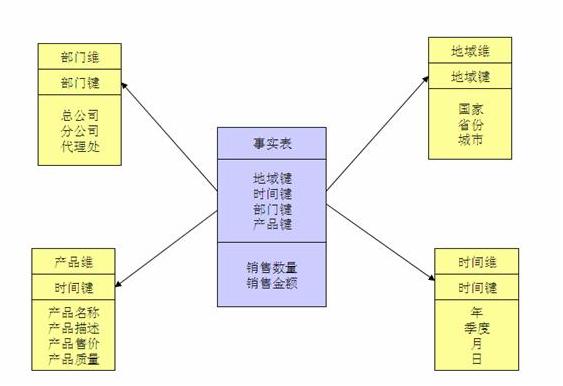
### 概述

在多维分析的商业智能解决方案中，根据事实表和维度表的关系，又可将常见的模型分为星型模型和雪花型模型。在设计逻辑型数据的模型的时候，就应考虑数据是按照星型模型还是雪花型模型进行组织。

当所有维表都直接连接到“ 事实表”上时，整个图解就像星星一样，故将该模型称为星型模型，如图 1 。

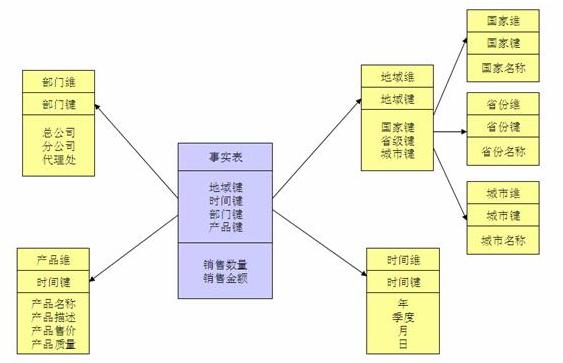
**星型架构是一种非正规化的结构，多维数据集的每一个维度都直接与事实表相连接，不存在渐变维度，所以数据有一定的冗余**，如在地域维度表中，存在国家 A 省 B 的城市 C 以及国家 A 省 B 的城市 D 两条记录，那么国家 A 和省 B 的信息分别存储了两次，即存在冗余。

**图1.** **销售数据仓库中的星型模型**



当有一个或多个维表没有直接连接到事实表上，而是通过其他维表连接到事实表上时，其图解就像多个雪花连接在一起，故称雪花模型。雪花模型是对星型模型的扩展。它对星型模型的维表进一步层次化，原有的各维表可能被扩展为小的事实表，形成一些局部的 " 层次 " 区域，这些被分解的表都连接到主维度表而不是事实表。如图 2，将地域维表又分解为国家，省份，城市等维表。它的优点是 : **通过最大限度地减少数据存储量以及联合较小的维表来改善查询性能。**雪花型结构去除了数据冗余。

**图 2.** **销售数据仓库中的雪花型模型**



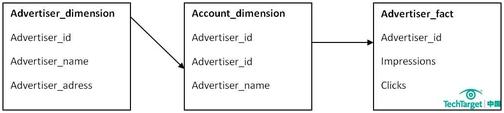
**星型模型因为数据的冗余所以很多统计查询不需要做外部的连接，因此一般情况下效率比雪花型模型要高。**星型结构不用考虑很多正规化的因素，设计与实现都比较简单。雪花型模型由于去除了冗余，有些统计就需要通过表的联接才能产生，所以效率不一定有星型模型高。正规化也是一种比较复杂的过程，相应的数据库结构设计、数据的 ETL、以及后期的维护都要复杂一些。**因此在冗余可以接受的前提下，实际运用中星型模型使用更多，也更有效率。**

### 使用选择

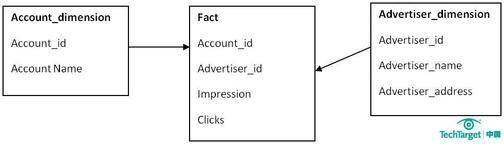
星形模型(Star Schema)和雪花模型(Snowflake Schema)是数据仓库中常用到的两种方式，而它们之间的对比要从四个角度来进行讨论。

#### ****数据优化****

**雪花模型使用的是规范化数据，也就是说数据在数据库内部是组织好的，以便消除冗余，因此它能够有效地减少数据量。**通过引用完整性，其业务层级和维度都将存储在数据模型之中。

  
▲图1 雪花模型

**相比较而言，星形模型实用的是反规范化数据。**在星形模型中，维度直接指的是事实表，业务层级不会通过维度之间的参照完整性来部署。



▲图2 星形模型

#### ****业务模型****

主键是一个单独的唯一键(数据属性)，为特殊数据所选择。在上面的例子中，Advertiser\_ID就将是一个主键。外键(参考属性)仅仅是一个表中的字段，用来匹配其他维度表中的主键。在我们所引用的例子中，Advertiser\_ID将是Account\_dimension的一个外键。

在雪花模型中，数据模型的业务层级是由一个不同维度表主键-外键的关系来代表的。而在星形模型中，所有必要的维度表在事实表中都只拥有外键。

#### ****性能****

第三个区别在于性能的不同。**雪花模型在维度表、事实表之间的连接很多，因此性能方面会比较低。**举个例子，如果你想要知道Advertiser 的详细信息，雪花模型就会请求许多信息，比如Advertiser Name、ID以及那些广告主和客户表的地址需要连接起来，然后再与事实表连接。

而星形模型的连接就少的多，在这个模型中，如果你需要上述信息，你只要将Advertiser的维度表和事实表连接即可。

#### ****ETL****

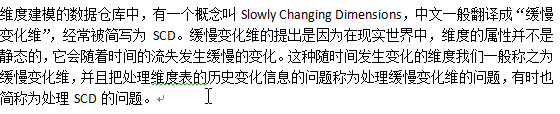
雪花模型加载数据集市，因此ETL操作在设计上更加复杂，而且由于附属模型的限制，不能并行化。

星形模型加载维度表，不需要再维度之间添加附属模型，因此ETL就相对简单，而且可以实现高度的并行化。

#### ****总结****

雪花模型使得维度分析更加容易，比如“针对特定的广告主，有哪些客户或者公司是在线的?”星形模型用来做指标分析更适合，比如“给定的一个客户他们的收入是多少?”

### 缓慢变化维



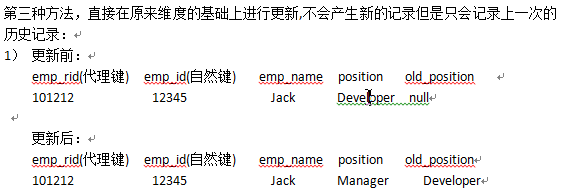
如何解决缓慢变化维带来的影响？



**上图中position有变化**



**上图中多了一条记录**



**多一个字段，用来存放以前的position**

# Spark SQL实战

## 数据说明

数据集是货品交易数据集。



每个订单可能包含多个货品，每个订单可以产生多次交易，不同的货品有不同的单价。

## 加载数据

tbStock：

|  |
| --- |
| **scala> case class tbStock(ordernumber:String,locationid:String,dateid:String) extends Serializable**  **defined class tbStock**  **scala> val tbStockRdd = spark.sparkContext.textFile("tbStock.txt")**  **tbStockRdd: org.apache.spark.rdd.RDD[String] = tbStock.txt MapPartitionsRDD[1] at textFile at <console>:23**  **scala> val tbStockDS = tbStockRdd.map(\_.split(",")).map(attr=>tbStock(attr(0),attr(1),attr(2))).toDS**  **tbStockDS: org.apache.spark.sql.Dataset[tbStock] = [ordernumber: string, locationid: string ... 1 more field]**  **scala> tbStockDS.show()**  **+------------+----------+---------+**  **| ordernumber|locationid| dataid|**  **+------------+----------+---------+**  **|BYSL00000893| ZHAO|2007-8-23|**  **|BYSL00000897| ZHAO|2007-8-24|**  **|BYSL00000898| ZHAO|2007-8-25|**  **|BYSL00000899| ZHAO|2007-8-26|**  **|BYSL00000900| ZHAO|2007-8-26|**  **|BYSL00000901| ZHAO|2007-8-27|**  **|BYSL00000902| ZHAO|2007-8-27|**  **|BYSL00000904| ZHAO|2007-8-28|**  **|BYSL00000905| ZHAO|2007-8-28|**  **|BYSL00000906| ZHAO|2007-8-28|**  **|BYSL00000907| ZHAO|2007-8-29|**  **|BYSL00000908| ZHAO|2007-8-30|**  **|BYSL00000909| ZHAO| 2007-9-1|**  **|BYSL00000910| ZHAO| 2007-9-1|**  **|BYSL00000911| ZHAO|2007-8-31|**  **|BYSL00000912| ZHAO| 2007-9-2|**  **|BYSL00000913| ZHAO| 2007-9-3|**  **|BYSL00000914| ZHAO| 2007-9-3|**  **|BYSL00000915| ZHAO| 2007-9-4|**  **|BYSL00000916| ZHAO| 2007-9-4|**  **+------------+----------+---------+**  **only showing top 20 rows** |

tbStockDetail:

|  |
| --- |
| **scala> case class tbStockDetail(ordernumber:String, rownum:Int, itemid:String, number:Int, price:Double, amount:Double) extends Serializable**  **defined class tbStockDetail**  **scala> val tbStockDetailRdd = spark.sparkContext.textFile("tbStockDetail.txt")**  **tbStockDetailRdd: org.apache.spark.rdd.RDD[String] = tbStockDetail.txt MapPartitionsRDD[13] at textFile at <console>:23**  **scala> val tbStockDetailDS = tbStockDetailRdd.map(\_.split(",")).map(attr=> tbStockDetail(attr(0),attr(1).trim().toInt,attr(2),attr(3).trim().toInt,attr(4).trim().toDouble, attr(5).trim().toDouble)).toDS**  **tbStockDetailDS: org.apache.spark.sql.Dataset[tbStockDetail] = [ordernumber: string, rownum: int ... 4 more fields]**  **scala> tbStockDetailDS.show()**  **+------------+------+--------------+------+-----+------+**  **| ordernumber|rownum| itemid|number|price|amount|**  **+------------+------+--------------+------+-----+------+**  **|BYSL00000893| 0|FS527258160501| -1|268.0|-268.0|**  **|BYSL00000893| 1|FS527258169701| 1|268.0| 268.0|**  **|BYSL00000893| 2|FS527230163001| 1|198.0| 198.0|**  **|BYSL00000893| 3|24627209125406| 1|298.0| 298.0|**  **|BYSL00000893| 4|K9527220210202| 1|120.0| 120.0|**  **|BYSL00000893| 5|01527291670102| 1|268.0| 268.0|**  **|BYSL00000893| 6|QY527271800242| 1|158.0| 158.0|**  **|BYSL00000893| 7|ST040000010000| 8| 0.0| 0.0|**  **|BYSL00000897| 0|04527200711305| 1|198.0| 198.0|**  **|BYSL00000897| 1|MY627234650201| 1|120.0| 120.0|**  **|BYSL00000897| 2|01227111791001| 1|249.0| 249.0|**  **|BYSL00000897| 3|MY627234610402| 1|120.0| 120.0|**  **|BYSL00000897| 4|01527282681202| 1|268.0| 268.0|**  **|BYSL00000897| 5|84126182820102| 1|158.0| 158.0|**  **|BYSL00000897| 6|K9127105010402| 1|239.0| 239.0|**  **|BYSL00000897| 7|QY127175210405| 1|199.0| 199.0|**  **|BYSL00000897| 8|24127151630206| 1|299.0| 299.0|**  **|BYSL00000897| 9|G1126101350002| 1|158.0| 158.0|**  **|BYSL00000897| 10|FS527258160501| 1|198.0| 198.0|**  **|BYSL00000897| 11|ST040000010000| 13| 0.0| 0.0|**  **+------------+------+--------------+------+-----+------+**  **only showing top 20 rows** |

tbDate:

|  |
| --- |
| **scala> case class tbDate(dateid:String, years:Int, theyear:Int, month:Int, day:Int, weekday:Int, week:Int, quarter:Int, period:Int, halfmonth:Int) extends Serializable**  **defined class tbDate**  **scala> val tbDateRdd = spark.sparkContext.textFile("tbDate.txt")**  **tbDateRdd: org.apache.spark.rdd.RDD[String] = tbDate.txt MapPartitionsRDD[20] at textFile at <console>:23**  **scala> val tbDateDS = tbDateRdd.map(\_.split(",")).map(attr=> tbDate(attr(0),attr(1).trim().toInt, attr(2).trim().toInt,attr(3).trim().toInt, attr(4).trim().toInt, attr(5).trim().toInt, attr(6).trim().toInt, attr(7).trim().toInt, attr(8).trim().toInt, attr(9).trim().toInt)).toDS**  **tbDateDS: org.apache.spark.sql.Dataset[tbDate] = [dateid: string, years: int ... 8 more fields]**  **scala> tbDateDS.show()**  **+---------+------+-------+-----+---+-------+----+-------+------+---------+**  **| dateid| years|theyear|month|day|weekday|week|quarter|period|halfmonth|**  **+---------+------+-------+-----+---+-------+----+-------+------+---------+**  **| 2003-1-1|200301| 2003| 1| 1| 3| 1| 1| 1| 1|**  **| 2003-1-2|200301| 2003| 1| 2| 4| 1| 1| 1| 1|**  **| 2003-1-3|200301| 2003| 1| 3| 5| 1| 1| 1| 1|**  **| 2003-1-4|200301| 2003| 1| 4| 6| 1| 1| 1| 1|**  **| 2003-1-5|200301| 2003| 1| 5| 7| 1| 1| 1| 1|**  **| 2003-1-6|200301| 2003| 1| 6| 1| 2| 1| 1| 1|**  **| 2003-1-7|200301| 2003| 1| 7| 2| 2| 1| 1| 1|**  **| 2003-1-8|200301| 2003| 1| 8| 3| 2| 1| 1| 1|**  **| 2003-1-9|200301| 2003| 1| 9| 4| 2| 1| 1| 1|**  **|2003-1-10|200301| 2003| 1| 10| 5| 2| 1| 1| 1|**  **|2003-1-11|200301| 2003| 1| 11| 6| 2| 1| 2| 1|**  **|2003-1-12|200301| 2003| 1| 12| 7| 2| 1| 2| 1|**  **|2003-1-13|200301| 2003| 1| 13| 1| 3| 1| 2| 1|**  **|2003-1-14|200301| 2003| 1| 14| 2| 3| 1| 2| 1|**  **|2003-1-15|200301| 2003| 1| 15| 3| 3| 1| 2| 1|**  **|2003-1-16|200301| 2003| 1| 16| 4| 3| 1| 2| 2|**  **|2003-1-17|200301| 2003| 1| 17| 5| 3| 1| 2| 2|**  **|2003-1-18|200301| 2003| 1| 18| 6| 3| 1| 2| 2|**  **|2003-1-19|200301| 2003| 1| 19| 7| 3| 1| 2| 2|**  **|2003-1-20|200301| 2003| 1| 20| 1| 4| 1| 2| 2|**  **+---------+------+-------+-----+---+-------+----+-------+------+---------+**  **only showing top 20 rows** |

注册表：

|  |
| --- |
| **scala> tbStockDS.createOrReplaceTempView("tbStock")**  **scala> tbDateDS.createOrReplaceTempView("tbDate")**  **scala> tbStockDetailDS.createOrReplaceTempView("tbStockDetail")** |

## 计算所有订单中每年的销售单数、销售总额

*统计所有订单中每年的销售单数、销售总额*

*三个表连接后以count(distinct a.ordernumber)计销售单数，sum(b.amount)计销售总额*



|  |
| --- |
| **SELECT c.theyear, COUNT(DISTINCT a.ordernumber), SUM(b.amount)**  **FROM tbStock a**  **JOIN tbStockDetail b ON a.ordernumber = b.ordernumber**  **JOIN tbDate c ON a.dateid = c.dateid**  **GROUP BY c.theyear**  **ORDER BY c.theyear** |

|  |
| --- |
| **spark.sql("SELECT c.theyear, COUNT(DISTINCT a.ordernumber), SUM(b.amount) FROM tbStock a JOIN tbStockDetail b ON a.ordernumber = b.ordernumber JOIN tbDate c ON a.dateid = c.dateid GROUP BY c.theyear ORDER BY c.theyear").show** |

结果如下：

|  |
| --- |
| **+-------+---------------------------+--------------------+**  **|theyear|count(DISTINCT ordernumber)| sum(amount)|**  **+-------+---------------------------+--------------------+**  **| 2004| 1094| 3268115.499199999|**  **| 2005| 3828|1.3257564149999991E7|**  **| 2006| 3772|1.3680982900000006E7|**  **| 2007| 4885|1.6719354559999993E7|**  **| 2008| 4861| 1.467429530000001E7|**  **| 2009| 2619| 6323697.189999999|**  **| 2010| 94| 210949.65999999997|**  **+-------+---------------------------+--------------------+** |

## 计算所有订单每年最大金额订单的销售额

目标：统计每年最大金额订单的销售额:



1. 统计每年，每个订单一共有多少销售额

|  |
| --- |
| **SELECT a.dateid, a.ordernumber, SUM(b.amount) AS SumOfAmount**  **FROM tbStock a**  **JOIN tbStockDetail b ON a.ordernumber = b.ordernumber**  **GROUP BY a.dateid, a.ordernumber** |

|  |
| --- |
| **spark.sql("SELECT a.dateid, a.ordernumber, SUM(b.amount) AS SumOfAmount FROM tbStock a JOIN tbStockDetail b ON a.ordernumber = b.ordernumber GROUP BY a.dateid, a.ordernumber").show** |

结果如下：

|  |
| --- |
| **+----------+------------+------------------+**  **| dateid| ordernumber| SumOfAmount|**  **+----------+------------+------------------+**  **| 2008-4-9|BYSL00001175| 350.0|**  **| 2008-5-12|BYSL00001214| 592.0|**  **| 2008-7-29|BYSL00011545| 2064.0|**  **| 2008-9-5|DGSL00012056| 1782.0|**  **| 2008-12-1|DGSL00013189| 318.0|**  **|2008-12-18|DGSL00013374| 963.0|**  **| 2009-8-9|DGSL00015223| 4655.0|**  **| 2009-10-5|DGSL00015585| 3445.0|**  **| 2010-1-14|DGSL00016374| 2934.0|**  **| 2006-9-24|GCSL00000673|3556.1000000000004|**  **| 2007-1-26|GCSL00000826| 9375.199999999999|**  **| 2007-5-24|GCSL00001020| 6171.300000000002|**  **| 2008-1-8|GCSL00001217| 7601.6|**  **| 2008-9-16|GCSL00012204| 2018.0|**  **| 2006-7-27|GHSL00000603| 2835.6|**  **|2006-11-15|GHSL00000741| 3951.94|**  **| 2007-6-6|GHSL00001149| 0.0|**  **| 2008-4-18|GHSL00001631| 12.0|**  **| 2008-7-15|GHSL00011367| 578.0|**  **| 2009-5-8|GHSL00014637| 1797.6|**  **+----------+------------+------------------+** |

1. *以上一步查询结果为基础表，和表tbDate使用dateid join，求出每年最大金额订单的销售额*

|  |
| --- |
| **SELECT theyear, MAX(c.SumOfAmount) AS SumOfAmount**  **FROM (SELECT a.dateid, a.ordernumber, SUM(b.amount) AS SumOfAmount**  **FROM tbStock a**  **JOIN tbStockDetail b ON a.ordernumber = b.ordernumber**  **GROUP BY a.dateid, a.ordernumber**  **) c**  **JOIN tbDate d ON c.dateid = d.dateid**  **GROUP BY theyear**  **ORDER BY theyear DESC** |

|  |
| --- |
| **spark.sql("SELECT theyear, MAX(c.SumOfAmount) AS SumOfAmount FROM (SELECT a.dateid, a.ordernumber, SUM(b.amount) AS SumOfAmount FROM tbStock a JOIN tbStockDetail b ON a.ordernumber = b.ordernumber GROUP BY a.dateid, a.ordernumber ) c JOIN tbDate d ON c.dateid = d.dateid GROUP BY theyear ORDER BY theyear DESC").show** |

结果如下：

|  |
| --- |
| **+-------+------------------+**  **|theyear| SumOfAmount|**  **+-------+------------------+**  **| 2010|13065.280000000002|**  **| 2009|25813.200000000008|**  **| 2008| 55828.0|**  **| 2007| 159126.0|**  **| 2006| 36124.0|**  **| 2005|38186.399999999994|**  **| 2004| 23656.79999999997|**  **+-------+------------------+** |

## 计算所有订单中每年最畅销货品

目标：统计每年最畅销货品（哪个货品销售额amount在当年最高，哪个就是最畅销货品）



第一步、求出每年每个货品的销售额

|  |
| --- |
| **SELECT c.theyear, b.itemid, SUM(b.amount) AS SumOfAmount**  **FROM tbStock a**  **JOIN tbStockDetail b ON a.ordernumber = b.ordernumber**  **JOIN tbDate c ON a.dateid = c.dateid**  **GROUP BY c.theyear, b.itemid** |

|  |
| --- |
| **spark.sql("SELECT c.theyear, b.itemid, SUM(b.amount) AS SumOfAmount FROM tbStock a JOIN tbStockDetail b ON a.ordernumber = b.ordernumber JOIN tbDate c ON a.dateid = c.dateid GROUP BY c.theyear, b.itemid").show** |

结果如下：

|  |
| --- |
| **+-------+--------------+------------------+**  **|theyear| itemid| SumOfAmount|**  **+-------+--------------+------------------+**  **| 2004|43824480810202| 4474.72|**  **| 2006|YA214325360101| 556.0|**  **| 2006|BT624202120102| 360.0|**  **| 2007|AK215371910101|24603.639999999992|**  **| 2008|AK216169120201|29144.199999999997|**  **| 2008|YL526228310106|16073.099999999999|**  **| 2009|KM529221590106| 5124.800000000001|**  **| 2004|HT224181030201|2898.6000000000004|**  **| 2004|SG224308320206| 7307.06|**  **| 2007|04426485470201|14468.800000000001|**  **| 2007|84326389100102| 9134.11|**  **| 2007|B4426438020201| 19884.2|**  **| 2008|YL427437320101|12331.799999999997|**  **| 2008|MH215303070101| 8827.0|**  **| 2009|YL629228280106| 12698.4|**  **| 2009|BL529298020602| 2415.8|**  **| 2009|F5127363019006| 614.0|**  **| 2005|24425428180101| 34890.74|**  **| 2007|YA214127270101| 240.0|**  **| 2007|MY127134830105| 11099.92|**  **+-------+--------------+------------------+** |

第二步、在第一步的基础上，统计每年单个货品中的最大金额

|  |
| --- |
| **SELECT d.theyear, MAX(d.SumOfAmount) AS MaxOfAmount**  **FROM (SELECT c.theyear, b.itemid, SUM(b.amount) AS SumOfAmount**  **FROM tbStock a**  **JOIN tbStockDetail b ON a.ordernumber = b.ordernumber**  **JOIN tbDate c ON a.dateid = c.dateid**  **GROUP BY c.theyear, b.itemid**  **) d**  **GROUP BY d.theyear** |

|  |
| --- |
| **spark.sql("SELECT d.theyear, MAX(d.SumOfAmount) AS MaxOfAmount FROM (SELECT c.theyear, b.itemid, SUM(b.amount) AS SumOfAmount FROM tbStock a JOIN tbStockDetail b ON a.ordernumber = b.ordernumber JOIN tbDate c ON a.dateid = c.dateid GROUP BY c.theyear, b.itemid ) d GROUP BY d.theyear").show** |

结果如下：

|  |
| --- |
| **+-------+------------------+**  **|theyear| MaxOfAmount|**  **+-------+------------------+**  **| 2007| 70225.1|**  **| 2006| 113720.6|**  **| 2004|53401.759999999995|**  **| 2009| 30029.2|**  **| 2005|56627.329999999994|**  **| 2010| 4494.0|**  **| 2008| 98003.60000000003|**  **+-------+------------------+** |

第三步、用最大销售额和统计好的每个货品的销售额join，以及用年join，集合得到最畅销货品那一行信息

|  |
| --- |
| **SELECT DISTINCT e.theyear, e.itemid, f.MaxOfAmount**  **FROM (SELECT c.theyear, b.itemid, SUM(b.amount) AS SumOfAmount**  **FROM tbStock a**  **JOIN tbStockDetail b ON a.ordernumber = b.ordernumber**  **JOIN tbDate c ON a.dateid = c.dateid**  **GROUP BY c.theyear, b.itemid**  **) e**  **JOIN (SELECT d.theyear, MAX(d.SumOfAmount) AS MaxOfAmount**  **FROM (SELECT c.theyear, b.itemid, SUM(b.amount) AS SumOfAmount**  **FROM tbStock a**  **JOIN tbStockDetail b ON a.ordernumber = b.ordernumber**  **JOIN tbDate c ON a.dateid = c.dateid**  **GROUP BY c.theyear, b.itemid**  **) d**  **GROUP BY d.theyear**  **) f ON e.theyear = f.theyear**  **AND e.SumOfAmount = f.MaxOfAmount**  **ORDER BY e.theyear** |

|  |
| --- |
| **spark.sql("SELECT DISTINCT e.theyear, e.itemid, f.maxofamount FROM (SELECT c.theyear, b.itemid, SUM(b.amount) AS sumofamount FROM tbStock a JOIN tbStockDetail b ON a.ordernumber = b.ordernumber JOIN tbDate c ON a.dateid = c.dateid GROUP BY c.theyear, b.itemid ) e JOIN (SELECT d.theyear, MAX(d.sumofamount) AS maxofamount FROM (SELECT c.theyear, b.itemid, SUM(b.amount) AS sumofamount FROM tbStock a JOIN tbStockDetail b ON a.ordernumber = b.ordernumber JOIN tbDate c ON a.dateid = c.dateid GROUP BY c.theyear, b.itemid ) d GROUP BY d.theyear ) f ON e.theyear = f.theyear AND e.sumofamount = f.maxofamount ORDER BY e.theyear").show** |

结果如下：

|  |
| --- |
| **+-------+--------------+------------------+**  **|theyear| itemid| maxofamount|**  **+-------+--------------+------------------+**  **| 2004|JY424420810101|53401.759999999995|**  **| 2005|24124118880102|56627.329999999994|**  **| 2006|JY425468460101| 113720.6|**  **| 2007|JY425468460101| 70225.1|**  **| 2008|E2628204040101| 98003.60000000003|**  **| 2009|YL327439080102| 30029.2|**  **| 2010|SQ429425090101| 4494.0|**  **+-------+--------------+------------------+** |