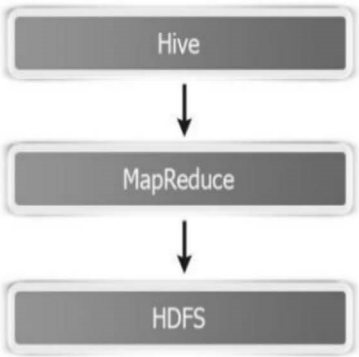
第15章 Spark SQL精华及与Hive的集成

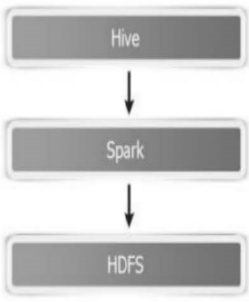
• Objective(本课目标)

- [x] Spark SQL API介绍 (重点)
- [x] Spark SQL优化器 – Catalyst Optimizer
- [x] DataFrame与Dataset, 及其操作运算(重点)
- [x] Spark自定义函数
- [x] Spark性能优化综述 (重点)
- [x] Spark SQL与Hive的集成

> Spark SQL 是 Spark 用于处理结构化数据的一个模块。不同于基础的 Spark RDD API，Spark SQL 提供的接口提供了更多关于数据和执行的计算任务的结构信息。Spark SQL 内部使用这些额外的信息来执行一些额外的优化操作。有几种方式可以与 Spark SQL 进 行交互，其中包括 SQL 和 Dataset API。当计算一个结果时 Spark SQL 使用的执行引擎是一样的， 它跟你使用哪种 API 或编程语 言来表达计算无关。这种统一意味着开发人员可以很容易地在不同的 API 之间来回切换，基于哪种 API 能够提供一种最自然的方式来表 达一个给定的变换（transformation）

• Spark SQL History(历史)



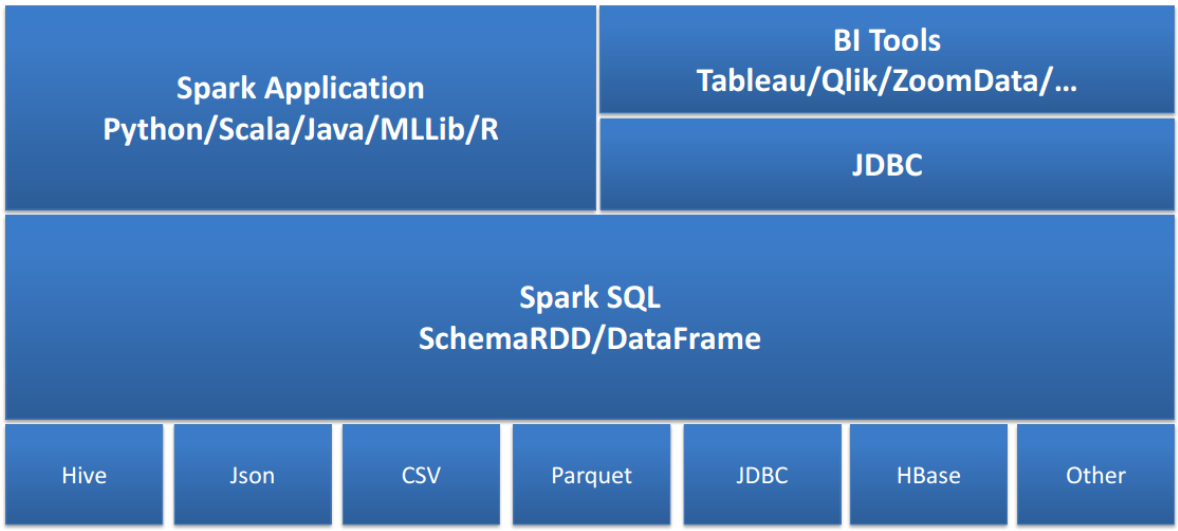


Shark, Spark SQL, Hive on Spark, and the future of SQL onSpark

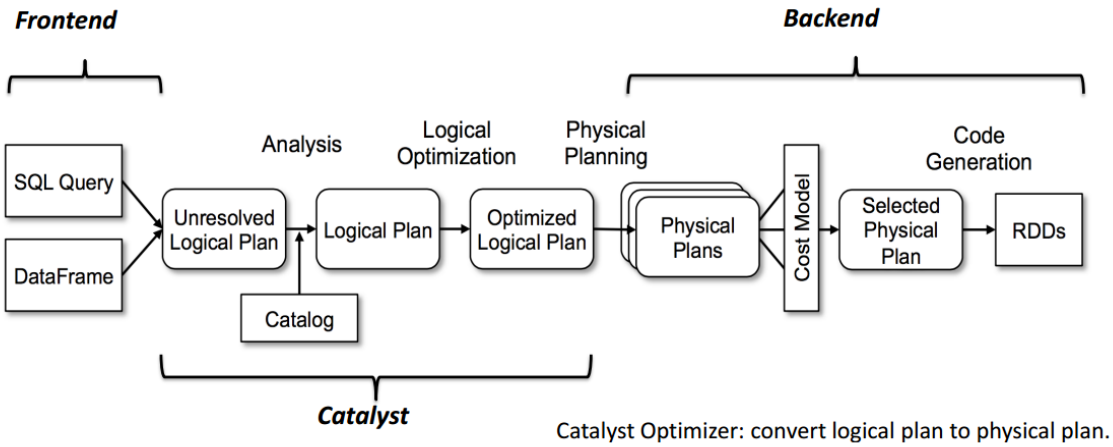
- Hive
  - 底层基于MapReduce，而MapReduce的shuffle又是基于磁盘的 => 性能异常低下。进场出现复杂的SQL ETL，要运行数个小时，甚至数十个小时的情况
- Shark
  - 依赖了Hive的语法解析器、查询优化器等组件
  - 修改了内存管理、物理计划、执行三个模块,底层使用Spark的基于内存的计算模型
- Spark SQL
  - 支持多种数据源：Hive、RDD、Parquet、JSON、JDBC等。
  - 多种性能优化技术：in-memory columnar storage、byte-code generation、cost model动态评估等。
  - 组件扩展性：对于SQL的语法解析器、分析器以及优化器，用户都可以自己重新开发，并且动态扩展
- 发展历程

```
Spark 0.x Shark
Spark 1.0.X SparkSQL Spark开始成为顶级项目
Spark 1.3.X 提出来DataFrame的核心抽象
Spark 1.6.X 提出来DataSet的核心抽象，属于测试阶段
Spark 2.X DataSet属于正式阶段
```

- **Apache Spark SQL Stack(技术栈)**



- **Spark SQL – How It Works(工作原理)**



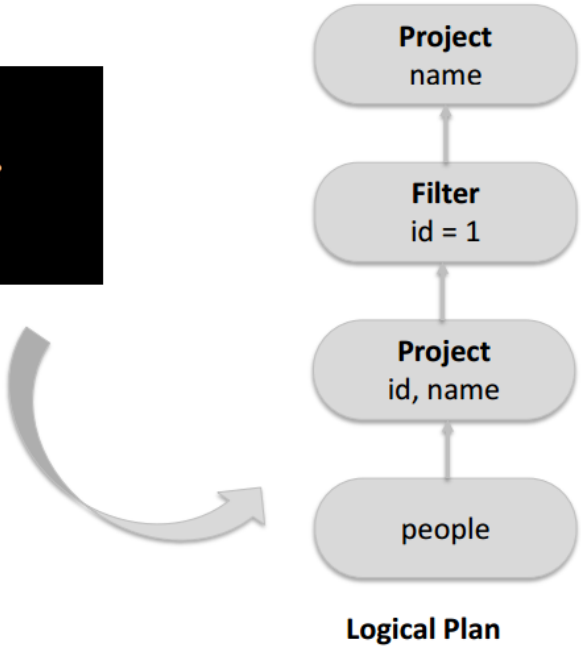
Catalyst Optimizer: convert logical plan to physical plan.

1. SQL Queary(速度稍微快一点点，但是弱类型的，执行才会发现错误)
2. DataFrame(强制类型的，性能稍微弱一点点)
3. Catalyst会对Logical Plan进行优化，生成物理计划，然后选择最佳模型，执行计划，生成RDD

Note：Relational Data Processing with SQL in Spark(在Spark中使用SQL进行关系数据处理)

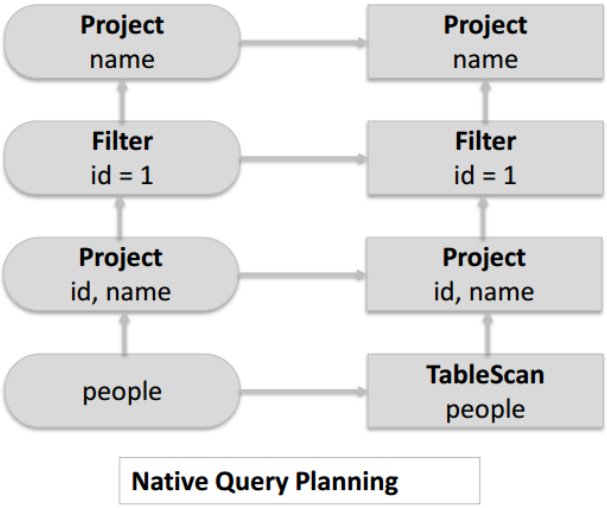
- **Catalyst Optimizer – Logical Plan(逻辑计划)**

```
SELECT name FROM
(
   SELECT id, name FROM people
) p
WHERE p.id = 1
```



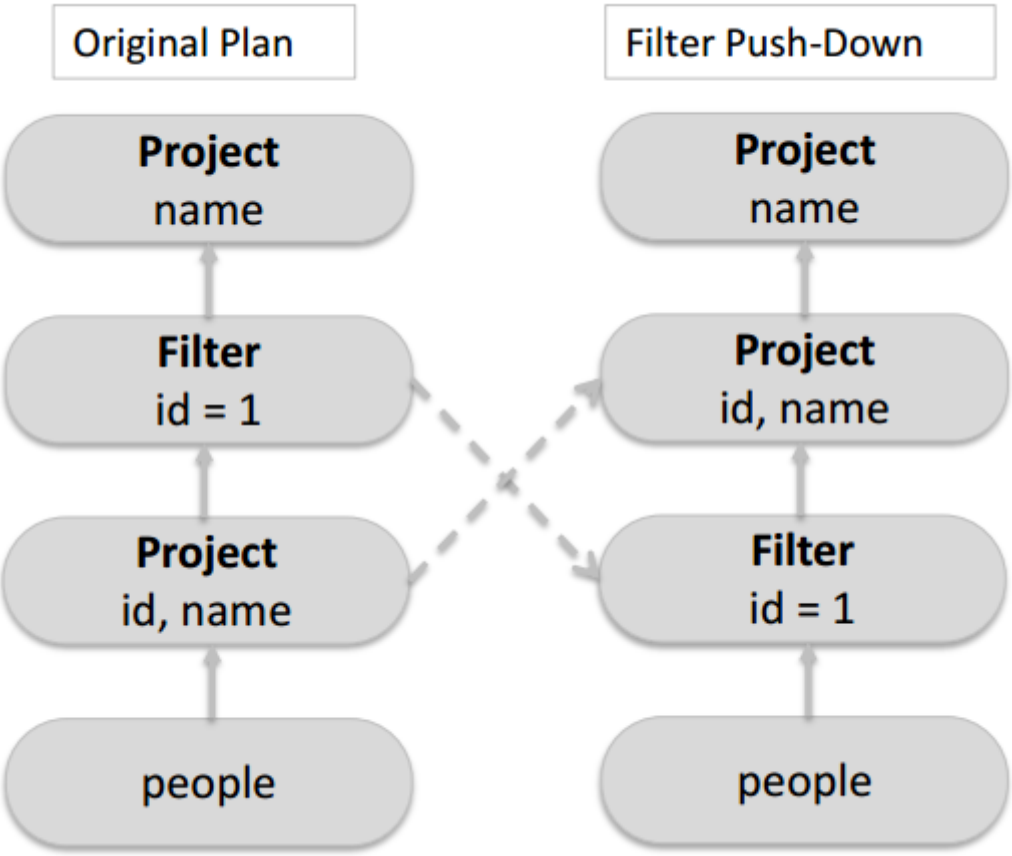
- **Catalyst Optimizer – Native Query**

```
SELECT name FROM
(
   SELECT
      id, name
   FROM people
) p
WHERE p.id = 1
```

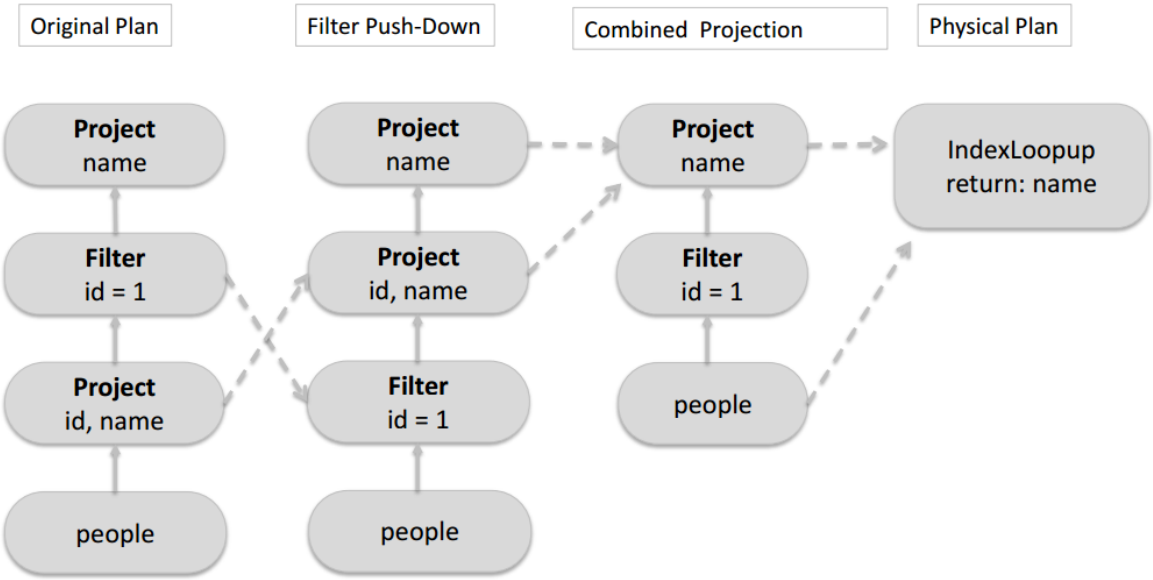


- **Catalyst Optimizer - Optimization(优化)**

  - Find filters on top of projections(找到过滤器)
  - Check if the filter can be evaluated before the projection(s) (评估过滤器是否能够提前)
  - If yes, switch the operations (如果是，请切换操作)



- Catalyst Optimizer – Physical Plan(物理计划)



- 请思考:

```
#案例
spark.sql(SELECT * FROM emp e WHERE e.sal<( SELECT sal FROM emp_2 WHERE
ename='ALLEN'))

1. 在driver上触发。
2. 在 workerNode上需要触发 SELECT sal FROM emp_2 WHERE ename='ALLEN'，做不到，因为无法
sc.broadcast(sc)
3. 解决方案?
```

SparkSQL程序入口

- 程序入口：Spark1.x
  - val sqlContext = new SQLContext(sc)
  - val hiveContext = new HiveContext(sc)
- 程序入口：Spark2.x
  - val spark = SparkSession

> SparkSession是Spark-2.0引如的新概念。SparkSession为用户提供了统一的切入点，来让用户学习Spark的各项功能。在Spark的早期版本中，SparkContext是Spark的主要切入点，由于RDD是主要的API，我们通过sparkContext来创建和操作RDD。对于每个其他的API，我们需要使用不同的context。SparkSession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。

```
//SparkSQL程序入口之Spark1.X
SQLContext:
    val sc: SparkContext // An existing SparkContext.
    val sqlContext = new org.apache.spark.sql.SQLContext(sc)
    // this is used to implicitly convert an RDD to a DataFrame.
    import sqlContext.implicits._

HiveContext
// sc is an existing SparkContext.
    val sqlContext = new org.apache.spark.sql.hive.HiveContext(sc)
    sqlContext.sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING)")
    sqlContext.sql("LOAD DATA LOCAL INPATH
'examples/src/main/resources/kv1.txt'INTO TABLE src")

//案例1 -> job1
SparkSQL程序入口之Spark2.X
import org.apache.spark.sql.SparkSession
def main(args: Array[String]): Unit = {
    val spark = SparkSession
      .builder()
```

```
      .appName("Spark SQL basic example")
      .getOrCreate()
    println("run success")
  }
```

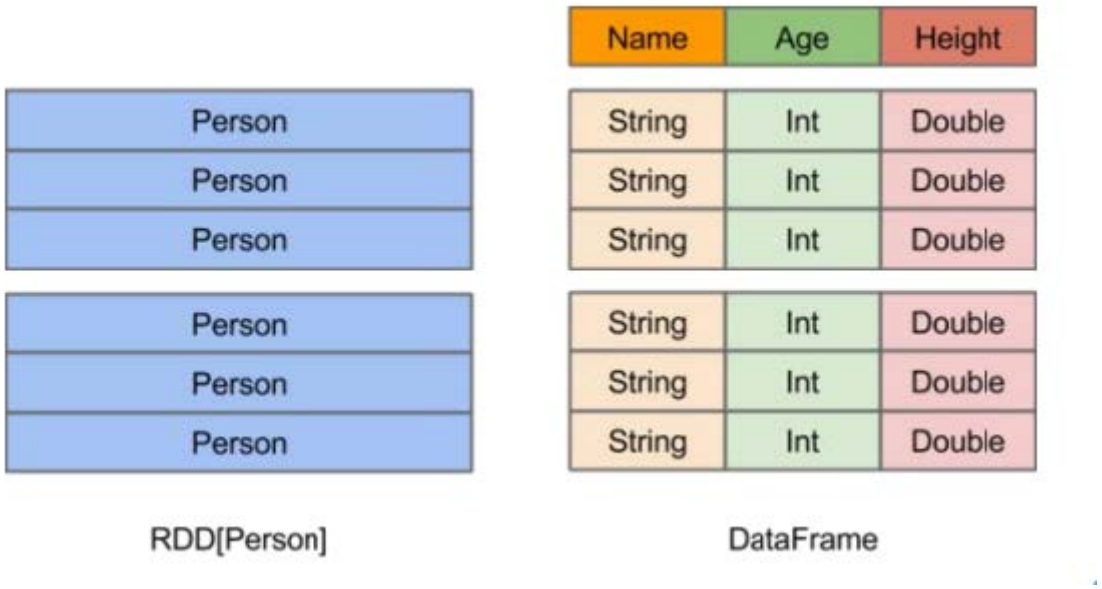
- (1)为用户提供一个统一的切入点使用Spark各项功能
- (2)允许用户通过它调用DataFrame和Dataset相关API来编写程序
- (3)减少了用户需要了解的一些概念，可以很容易的与Spark进行交互
- (4)与Spark交互之时不需要显示的创建SparkConf、SparkContext以及SQlContext，这些对象已经封闭在SparkSession中
- (5)SparkSession提供对Hive特征的内部支持：用HiveQL写SQL语句，访问Hive UDFs，从Hive表中读取数据。

SparkSQL核心抽象-DataFrame

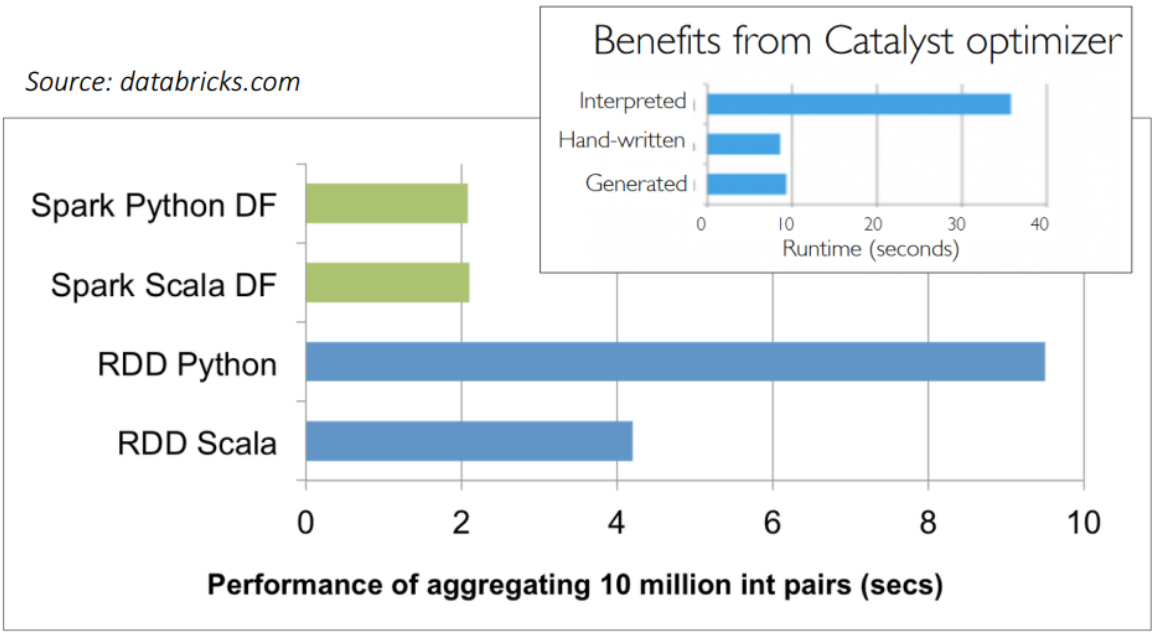
> A DataFrame is a Dataset organized into named columns. It is conceptually equivalent to a table in a relational database or a data frame in R/Python, but with richer optimizations under the hood. DataFrames can be constructed from a wide array of sources such as: structured data files, tables in Hive, external databases, or existing RDDs. The DataFrame API is available in Scala, Java,Python, and R. In Scala and Java, a DataFrame is represented by a Dataset of Rows. In the Scala API, DataFrame is simply a type alias of Dataset[Row]. While,in Java API, users need to use Dataset to represent a DataFrame.
>
> DataFrame是一个组织成指定列的数据集。从概念上说相当于R/Python中的关系数据库中的表或数据帧，但是有更丰富的底层优化。数据帧可以从广泛的源，如:结构化数据文件，Hive表，外部数据库，或现有rdd。DataFrame API有Scala, Java，在Scala和Java中，一个数据帧由一个数据集表示行。在Scala API中，DataFrame只是Dataset[Row]的类型别名。同时,在Java API中，用户需要使用Dataset来表示一个DataFrame。

DataFrame 和 RDD的区别

- Schema on read（DF相当于是 schemaRDD）
- Works with structured and semi-structured data (JSON,XML) (处理结构化和半结构化数据)
- 在Spark中，DataFrame是一种以RDD为基础的分布式数据集，类似于传统数据库中的二维表
- DataFrame与RDD的主要区别在于，前者带有schema元信息，即DataFrame所表示的二维表数据集的每一列都带有名称和类型。
- 使得Spark SQL得以洞察更多的结构信息，从而对藏于DataFrame背后的数据源以及作用于DataFrame之上的变换进行了针对性的优化，最终达到大幅提升运行时效率的目标
- 反观RDD，由于无从得知所存数据元素的具体内部结构，Spark Core只能在stage层面进行简单、通用的流水线优化。



- **DF – Performance(性能比较)**



- **DataFrame案例应用**

  - **DataFrame快速使用**

```
案例2 -> job2

def main(args: Array[String]): Unit = {
      Logger.getLogger("org").setLevel(Level.ERROR)
      val spark = SparkSession
        .builder()
        .master("local")
        .appName("Spark SQL basic example")
        .getOrCreate()
      val df: DataFrame = spark.read.json("person.json")
```

```
      //输出数据到控制台
      df.show()
      // This import is needed to use the $-notation (这个导入需要使用$-表示法)
      import spark.implicits._
      // 输出DataFrame结构化信息
      df.printSchema()
      //指定name列
      df.select("name").show()
      //通过条件查询
      df.select($"name", $"age" + 1).show()
      df.filter($"age" > 21).show()
      df.groupBy("age").count().show()

      // Register the DataFrame as a SQL temporary view  (将DataFrame注册为SQL临时视
图)
      df.createOrReplaceTempView("person")
      val sqlDF = spark.sql("SELECT * FROM person")
      sqlDF.show()
    }
```

Global Temporary View

> Spark SQL中的临时View是有范围域的。如果session停止创建它，那么临时View会消失。如果你想拥有一个临时View，它能在所有session间共享，并且在Spark应用运行期间一直存在，你可以创建一个全局临时View。全局临时View和系统数据库global_temp绑定，我们也需要使用相应的名字来访问它，如SELECT * FROM global_temp.view1

```
Global Temporary View
案例3 -> job3
def main(args: Array[String]): Unit = {
      Logger.getLogger("org").setLevel(Level.ERROR)
      val spark = SparkSession
        .builder()
        .master("local")
        .appName("Spark SQL basic example")
        .getOrCreate()
      val df: DataFrame = spark.read.json("person.json")
      df.createGlobalTempView("people")
      // Global temporary view is tied to a system preserved
database`global_temp` (全局临时视图绑定到系统保留的数据库' global_temp ')
      spark.sql("SELECT * FROM global_temp.people").show()
      // Global temporary view is cross-session  (temporary view是在session中共享
的)
      spark.newSession().sql("SELECT * FROM global_temp.people").show()
    }
```

- RDD VS DataFrame编程的区别：

```
案例4 -> job4
object Job4 {
    def main(args: Array[String]): Unit = {
      Logger.getLogger("org").setLevel(Level.ERROR)
      val conf = new SparkConf().setAppName("sparksql").setMaster("local")
      val spark = SparkSession.builder().config(conf).getOrCreate()
      val personRDD = spark.sparkContext.textFile("person4.txt").map(line =>
line.split(",")).map(p => Person(p(0),p(1).trim().toLong))
      println(personRDD)
      personRDD.foreach(println(_))

      //spark.read.json
      val personDataFrame = spark.read.json("person.json")
      //显示dataframe里面的每一条数据，row表示一行数据
      personDataFrame.foreach(row => println(row.get(0)))
      personDataFrame.foreach(row => println(row.getString(1)))
      personDataFrame.foreach(row => println(row.getAs[String]("name")))


    }
}
case class Person(name:String,age:Long)

Note: RDD里面的每一条数据是一个object，那么DataFrame里面的每一条数据是什么呢?
```

- DataSet案例应用

```
案例5 -> job5
// 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|
// +----+-------+
```

- **RDD转DataFrame**

```
案例6 -> job6
/**
* RDD 转化成为DataFrame
*/
case class Person1(var name:String,var age:Int)
object Job6 {
    def main(args: Array[String]): Unit = {
      Logger.getLogger("org").setLevel(Level.ERROR)
      val spark = SparkSession
        .builder()
        .master("local")
        .appName("Spark SQL basic example")
        .getOrCreate()

      val rdd1 = spark.sparkContext.textFile("person4.txt")
      import spark.implicits._
      val rdd2 = rdd1.map(line => Person1(line.split(",")(0), line.split(",")
(1).trim.toInt))
      val dataFrame1 = rdd2.toDF()
      dataFrame1.createOrReplaceTempView("person")
      spark.sql("select count(*) from person").show()
    }
}

案例7 -> job7
Programmatically Specifying the Schema
def main(args: Array[String]): Unit = {
      Logger.getLogger("org").setLevel(Level.ERROR)
      val spark = SparkSession
        .builder()
        .master("local")
        .appName("Spark SQL basic example")
        .getOrCreate()

      val rdd1 = spark.sparkContext.textFile("person4.txt")
      import spark.implicits._
      val rowRDD: RDD[Row] = rdd1.map(line => {
```

```
        val fields = line.split(",")
        Row(fields(0), fields(1).trim.toInt)
      })

      val structType = StructType(
        StructField("name", StringType, true) ::
        StructField("age", IntegerType, true) :: Nil
      )

      val df: DataFrame = spark.createDataFrame(rowRDD,structType)
      df.createOrReplaceTempView("people")
      spark.sql("select name from people").show()
    }

Note:
    1.第一种方式需要提前知道数据的结构，然后设计对应的caseClass
```

• 数据源

– Load/Save

```
案例8 -> job8
import org.apache.spark.sql.{DataFrame, SparkSession}
def main(args: Array[String]): Unit = {
      Logger.getLogger("org").setLevel(Level.ERROR)
      val spark = SparkSession
        .builder()
        .master("local")
        .appName("Spark SQL basic example")
        .getOrCreate()

      import spark.implicits._
      //格式一: parquet
      val df: DataFrame = spark.read.load("users.parquet")
      df.createOrReplaceTempView("users")
      spark.sql("select name from users").show()

      //格式二: json
      val df1 = spark.read.format("json").load("people.json")
      df1.createOrReplaceTempView("people1")
      spark.sql("select * from people1").show()

      //格式三: CSV
      val df2 =
spark.read.format("csv").option("header","true").option("delimiter",";").load("pe
ople.csv")
      df2.createOrReplaceTempView("people2")
      spark.sql("select * from people2").show()
    }
```

```
案例9 -> job9
def main(args: Array[String]): Unit = {
      Logger.getLogger("org").setLevel(Level.ERROR)
      val spark = SparkSession
        .builder()
        .master("local")
        .appName("Spark SQL basic example")
        .getOrCreate()
      //格式一: parquet
      val df: DataFrame = spark.read.load("users.parquet")
      df.write.save("xxx")//parquet

      val df1 = spark.read.format("json").load("people.json")
      df1.createOrReplaceTempView("people1")
      spark.sql("select name from
people1").write.format("json").save("test.json")
    }
```

Scala/Java	Any Language	Meaning
SaveMode.ErrorIfExists (default)	"error" (default)	When saving a DataFrame to a data source, if data already exists, an exception is expected to be thrown.
SaveMode.Append	"append"	When saving a DataFrame to a data source, if data/table already exists, contents of the DataFrame are expected to be appended to existing data.
SaveMode.Overwrite	"overwrite"	Overwrite mode means that when saving a DataFrame to a data source, if data/table already exists, existing data is expected to be overwritten by the contents of the DataFrame.
SaveMode.Overwrite	"overwrite"	Overwrite mode means that when saving a DataFrame to a data source, if data/table already exists, existing data is expected to be overwritten by the contents of the DataFrame.
SaveMode.Ignore	"ignore"	Ignore mode means that when saving a DataFrame to a data source, if data already exists, the save operation is expected to not save the contents of the DataFrame and to not change the existing data. This is similar to a CREATE TABLE IF NOT EXISTS in SQL.

Mysql

```
案例10 -> job10
/**
* (1) 方式一:
* 在spark-env.sh里
* SPARK_CLASSPATH=$SPARK_CLASSPATH/xxx.jar
*
* (2) 方式二
* spark-shell --driver-class-path xxx.jar
```

```
*
*/
def main(args: Array[String]): Unit = {
      Logger.getLogger("org").setLevel(Level.ERROR)
      val spark = SparkSession
        .builder()
        .master("local")
        .appName("Spark SQL basic example")
        .getOrCreate()
      val url="jdbc:mysql://192.168.134.130:3306/hivetest?serverTimezone=GMT";
      val properties = new Properties()
      properties.put("url",url)
      properties.put("user","root")
      properties.put("password","root")
      val df: DataFrame = spark.read.jdbc(url,"people",properties)
      df.show()
    }
```

- Hive

```
(1)添加驱动
spark-shell --driver-class-path /work/hadoop/hive-2.3.4/lib/mysql-connectorjava-
5.1.38.jar
(2)创建支持Hive的程序入口
import org.apache.spark.sql.{DataFrame, SparkSession}
val spark = SparkSession.builder().appName("Spark
HiveExample").enableHiveSupport().getOrCreate()
(3)操作Hive数据库
spark.sql("show databases").show
```

自定义函数

- UDF

```
import org.apache.spark.sql.{DataFrame, SparkSession}
def main(args: Array[String]): Unit = {
      Logger.getLogger("org").setLevel(Level.ERROR)
      val spark = SparkSession
        .builder()
        .master("local")
        .appName("Spark SQL basic example")
        .getOrCreate()
      spark.udf.register("strLen",(str:String) =>{
        if(str != null){
          str.length
        }else{
          0
        }
```

```
    })
    val df: DataFrame = spark.read.load("users.parquet")
    df.createOrReplaceTempView("users")
    spark.sql("select strLen(name) from users").show()
  }
```

UDAF

```
/**
*
* 思考： 聚合函数 -》 功能就是求平均工资
* 方式:
* 平均工资=所有人的工资/总人数
* 所以我们求出来平均工资，需要得出两个值:
* 1: 所有人的工资
* total
* 2: 总人数
* count
*
*/
object UDAFDemo extends UserDefinedAggregateFunction{
/**
* 定义输入的数据类型
* @return
*/
override def inputSchema: StructType = StructType(
StructField("salary",DoubleType,true) :: Nil
)
/**
* 定义输出的数据类型
* @return
*/
override def dataType: DataType = DoubleType
/**
* 定义辅助字段:
* 辅助字段一 total:
* 用来记录总工资
* 辅助字段二: count:
* 用来记录总人数
* @return
*/
override def bufferSchema: StructType = StructType(
StructField("total",DoubleType,true) ::
StructField("count",IntegerType,true) ::
Nil
)
/**
* 最后的计算的目标函数
* @param buffer
* @return
```

```
*/
override def evaluate(buffer: Row): Any = {
val total = buffer.getDouble(0)
val count = buffer.getInt(1)
//平均工资
total/count
}
/**
* 初始化辅助字段
*
* @param buffer
*/
override def initialize(buffer: MutableAggregationBuffer): Unit = {
buffer.update(0,0.0)
buffer.update(1,0)
}
/**
* 更新辅助字段的值
* @param buffer
* @param input
* 局部
*/
override def update(buffer: MutableAggregationBuffer, input: Row): Unit = {
val lastTotal = buffer.getDouble(0)
val lastCount = buffer.getInt(1)
val currentSalary = input.getDouble(0)
buffer.update(0,lastTotal + currentSalary)
buffer.update(1,lastCount+1)
}
/**
* 全局的
* @param buffer1
* @param buffer2
*/
override def merge(buffer1: MutableAggregationBuffer, buffer2: Row): Unit = {
val total1 = buffer1.getDouble(0)
val count1 = buffer1.getInt(1)
val total2 = buffer2.getDouble(0)
val count2 = buffer2.getInt(1)
buffer1.update(0,total1 + total2)
buffer1.update(1,count1 + count2)
}
override def deterministic: Boolean = true
}
object SalaryAvgTest {
def main(args: Array[String]): Unit = {
val spark = SparkSession
.builder()
.appName("Spark SQL basic example")
.getOrCreate()
spark.udf.register("avg_salary",UDAFDemo)
spark.sql("select avg_salary(salary) from mydb.worker").show
```

```
}
}
```

- **窗口函数**

```
import org.apache.spark.sql.SparkSession
object WindowFunctionOperation {
def main(args: Array[String]): Unit = {
val spark = SparkSession
.builder()
.master("local")
.enableHiveSupport()
.appName("Spark SQL basic example")
.getOrCreate()
spark.sql("user nx")
val sql1=
"""
create table result
(ipdaddress string,category string, product string, price int)
row format delimited fields terminated by '\t'
""".stripMargin
spark.sql(sql1)
val sql2=
"""
load data local inpath 'xx/result.txt' into table result
""".stripMargin
spark.sql(sql2)
val sql3=
"""
select
ipaddress,category,product,price
from
(select ipaddress,category,product,price,row_number() OVER
(partition by category order by price desc) rank from result)
tmp_result
where
tmp_result.rank <= 3
""".stripMargin
val topNDF = spark.sql(sql3)
spark.sql("drop create if exists top3_result")
topNDF.write.saveAsTable("top3_result")
}
}
```

- **Distributed SQL Engine**

```
export HIVE_SERVER2_THRIFT_PORT=10015
export HIVE_SERVER2_THRIFT_BIND_HOST=10.148.11.45
start-thriftserver.sh \
--master yarn \
--driver-memory 20g \ 这儿调整的就是Driver的堆内存，建议有条件的话，这个值给大一点。
--conf spark.driver.maxResultSize=5g \
--queue root.spark.dp
```

- spark-sql

```
Configuration of Hive is done by placing your hive-site.xml, core-site.xml and
hdfs-site.xml files in conf/. You may run ./bin/spark-sql --help for a complete
list of all available options.
spark-sql --driver-class-path /work/hadoop/hive-2.3.4/lib/mysql-connector-java-
5.1.38.jar
```

- **从file生成DF**

```
//案例1
val dfCustomers =
spark.read.format("csv").option("header","false").option("delimiter",",").load("/
spark_data/spark_sql/customers.csv")

dfCustomers.show()

//转换成表
dfCustomers.createOrReplaceTempView("customers")

//查询
spark.sql("select * from customers limit 10").show
spark.sql("select * from customers limit 20").show(15)

//显示结构信息
df.printSchema

//案例2   withColumnRenamed和withColumn的使用
import org.apache.spark.sql.functions._
import org.apache.spark.sql.types._
val dfProduct =
spark.read.format("csv").option("header","false").option("delimiter",",").option(
"escape","\\").load("/spark_data/spark_sql/products.csv")

.withColumn("product_id",col("_c0").cast(IntegerType))
.withColumn("product_category_id",col("_c1").cast(IntegerType))
.withColumnRenamed("_c2","product_name")
.withColumnRenamed("_c3","product_description")
.withColumn("product_price",col("_c4").cast(DoubleType))
.withColumnRenamed("_c5","product_image")
```

```
dfProduct.printSchema

//案例3.常用属性
delimiter：分隔符，默认为逗号，
nullValue：指定一个字符串代表 null 值
quote：引号字符，默认为双引号"
header：第一行不作为数据内容，作为标题
inferSchema：自动推测字段类型

#基本数数据
id|name|age
1| darren |19
2|anne|20
3|"test"|16
4|'test2'|18

#输出结果
val result = spark.read.format("csv")
      .option("delimiter", "|")
      .option("header", "true")
      .option("quote", "'")
      .option("nullValue", "18")
      .option("inferSchema", "true")
      .load("....csv")
+---+--------+----+
| id|    name| age|
+---+--------+----+
|  1| darren |  19|
|  2|    anne|  20|
|  3|  "test"|  16|
|  4|   test2|null|
+---+--------+----+


//案例4 加载json数据
val conf = new SparkConf().setAppName("sparksql").setMaster("local")
val spark = SparkSession.builder().config(conf).getOrCreate()
val df = spark.read.json("example1.json")
df.printSchema()
df.createOrReplaceTempView("example1")
spark.sql("select dict['key'] from example1").show()
```

- **通过RDD生成DF**

```
//案例1，自动转换
val rddProduct =
spark.sparkContext.textFile("/spark_data/spark_sql/products.csv").map(x =>
x.split(";").map(y => if(y != null) y.replace("\"","") else ""))
rddProduct.toDF


//案例2，手动转换
//定义schema
val schema =
StructType(Array(StructField("product_id",IntegerType,false),StructField("product
_category_id",IntegerType,false),StructField("product_name",StringType,false),Str
uctField("product_description",StringType,true),StructField("product_price",Doubl
eType,false),StructField("product_image",StringType,false)))

import org.apache.spark.sql.Row

val df = spark.createDataFrame(rddProduct2.map(r =>
Row(r(0).toInt,r(1).toInt,r(2),r(3),r(4).toDouble,r(5))),schema)
```

• DataFrame operators（常用操作函数）

- DataFrame operators
  - map, flatMap
  - sample, filter
  - sort
  - pipe –（pipe()方法可以让我们使用任意一种语言实现Spark作业中的部分逻辑）
  - groupBy, groupByKey, cogroup
  - reduce, reduceByKey, fold
  - partitionBy
  - zip, union
  - join, crossJoin, leftOuterJoin, rightOuterJoin
  - count, save
  - first, take

• DF Operator - 函数用法

```
//案例1 select
val df1: DataFrame = session.sqlContext.read.format("csv").load("example3.txt")
df1.printSchema()
df1.show()
val df2 = df1.select("_c0","_c1")
df2.printSchema()
```

```
//案例2 sort
Import org.apache.spark.sql.types._
Import org.apache.spark.sql.functions._
Import org.apache.spark.sql.Row
val ufc = spark.sparkContext.parallelize(Seq(("zhangsan", 123), ("lisi", 81)))
val schema = StructType(Array(StructField("name", StringType, false),
StructField("friend_count", IntegerType, false)))
val df = spark.createDataFrame(ufc.map(x => Row(x._1, x._2)), schema)
//sort
df.sort(col("name").desc,$"friend_count".asc).show


//案例3 cogroup
cogroup


//案例4 zip
val rdd1 = sc.parallelize(List('a','b','c','d'))
val rdd2 = sc.parallelize(List(100,200,300,400))
rdd1.zip(rdd2).foreach(println(_))
```

- **DataFrame - 自定义函数**

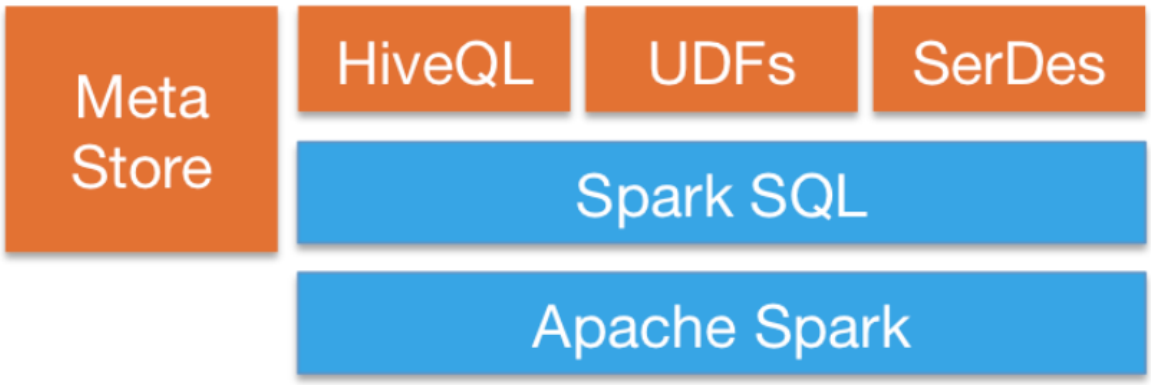
```
//案例1
import org.apache.spark.sql.functions._

//定义业务方法
def getDayOfWeek(s_date:String):String = {
    import java.util.Calendar
    val weeks = Array("星期日", "星期一", "星期二", "星期三", "星期四", "星期五", "星期
六")
    val cal = Calendar.getInstance
    //2020-12-05 00:00:00
    val dt_format = new java.text.SimpleDateFormat("yyyy-MM-dd")
    val date = dt_format.parse(s_date)
    cal.setTime(date)
    var week_index = cal.get(Calendar.DAY_OF_WEEK) - 1
    val pattern = """([0-9]{4})-([0-9]{2})-([0-9]{2}).*""".r
    s_date match {
      case pattern(year,month,day) => weeks(week_index)
      case _ => "Unknown"
    }
  }

//注册
val udfDayOfWeek = udf( (s_date:String) => getDayOfWeek(s_date))

//测试
  def main(args: Array[String]): Unit = {
```

```
    val conf = new SparkConf().setAppName("sparksql").setMaster("local")
    val spark = SparkSession.builder().config(conf).getOrCreate()
    val sc = spark.sparkContext


    //测试
    val str = getDayOfWeek("2020-12-05")
    println(str)

    val dfOrders =
spark.read.format("csv").option("header","false").option("delimiter",",").load("o
rders.csv")
    val dfOrderFinal = dfOrders.withColumnRenamed("_c0","order_id")
      .withColumn("order_week_day",udfDayOfWeek(col("_c1")))
      .withColumnRenamed("_c2","order_customer_id")
      .withColumnRenamed("_c3","order_status")

    dfOrderFinal.createOrReplaceTempView("orders")
    //方式一
    spark.sql("select order_week_day,count(order_week_day) as order_cnt from
orders group by order_week_day order by order_cnt desc ").show()
    //方式二
    dfOrderFinal.select(col("order_week_day")).distinct.show

dfOrderFinal.groupBy(col("order_week_day")).agg(count(col("order_id")).as("order_
count")).sort(col("order_count").desc).select("order_week_day","order_count").sho
w


  }



//案例2 在SQL里面使用
dfOrders.createOrReplaceTempView("orders")
//注册函数
spark.udf.register("udfDayOfWeek",(s_date:String) => getDayOfWeek(s_date))
//使用函数
spark.sql("select _c0 as order_id, udfDayOfWeek(_c1) as week ,_c2 as
order_customer_id,_c3 as order_status from orders limit 10").show
//代码优化写法
s_date match {
    case pattern(year,month,day) =>
scala.util.Try(weeks(week_index)).toOption.getOrElse("Unknown")
    case _ => "Unknown"
}



//案例2 User Defined Functions
import java.util.Calendar
val dfEvents = sparkSession.createDataFrame(rddEvents, schemaEvent)
```

```
val daysAhead = udf {
(dts1: String, dts2: String) => Util.getDays(dts1, dts2)
}

val getAge = udf { (birth_year: Int) => Calendar.getInstance().get(Calendar.YEAR)
- birth_year }

dfEvents.join(dfUsers, dfEvents.col("user_id") === dfUsers.col("id"),
"left_outer")
.withColumn("view_ahead_days", daysAhead(dfUsers.col("invited_date"),
dfEvents("start_date"))
.withColumn("age", getAge(dfUsers.col("birth_year").cast(IntegerType))
select(
"event_id", "event_name",
"user_id", "user_name", "age"
)
```

• Spark SQL with Hive Meta(Spark SQL读取hive数据)

- SparkSQL uses Hive's meta store, but his own version of thrift server(Spark SQL使用Hive Metastore，但是他自己的Thrift Server版本)




```
spark.sql("show databases").show
spark.sql("use default")
spark.sql("show tables").show
spark.sql("select ....").show
spark.table("table_name").show
spark.table("table_name").count
```

- Access in Spark(spark读取hive数据)

```
// 案例1
// beeline -u jdbc:hive2://localhost:10000/hive_db
create database if not exists hive_db;
create table employees(name string, phone_number string, office_address string);
```

```
insert into employees(name, phone_number, office_address) values('zs', '111-222-
333 ', 'wh');
insert into employees(name, phone_number, office_address) values('lisi', '131-
281-312','bj');
insert into employees(name, phone_number, office_address) values('wangwu', '533-
121-732 ', 'sh');
insert into employees(name, ssn, office_address) values('zhaoliu', '823-021-591
','sz');

val df = spark.sqlContext.table("hive_db.employees")
df.printSchema
|-- name: string (nullable = true)
|-- phone_number: string (nullable = true)
|-- office_address: string (nullable = true)

df.filter($"phone_number".startsWith("111")).show
+---------+------------+-----------------------+
|name| phone_number| office_address|
+---------+------------+-----------------------+
| zs |111-222-333 |123 wh |
+---------+------------+-----------------------+

// 案例2 IDE开发
//如果没有hive, spark可以创建一个local hive datawarehouse。enableHiveSupport 是告诉
spark可以直接操作hive
val spark = SparkSession.builder().config("spark.sql.warehouse.dir",
warehouseLocation)
.enableHiveSupport().getOrCreate()
val df = spark.sql("select * from hive_db.employees")
df.filter($"phone_number".startsWith("111")).show
```

- **DF Example – Write to Hive（spark写入数据到hive）**

```
//案例1
val users = spark.read.format("csv").option("header", "true").load("users.csv")
users.printSchema
val s_users = users.select("user_id", "birthyear", "gender", "location")
//save
s_users.write.mode("overwrite").saveAsTable("hive_db.users")
//query
spark.sql("select * from hive_db.users limit 20")
```

- **DF Example – Write to RDB（spark写入数据到RDB）**

  - spark-shell --jars /usr/hdp/current/hive-client/lib /mysql-connector-java.jar (指定mysql驱动)

```
//案例1 写入mysql
val train = spark.read.format("csv").option("header",
"true").load("train.csv").select("user", "event", "invited", "timestamp",
"interested")

import java.util.Properties
val props = new Properties()
props.put("driver", "com.mysql.jdbc.Driver")
props.put("user", "root")
props.put("password", "hadoop")
//save
train.write.mode("append").jdbc("jdbc:mysql://localhost:3306/mysql", "train",
props)
```

• DF Example – Read from RBD（spark从RDB读取数据）

```
//案例
val train = spark.read
.format("jdbc")
.option("url", "jdbc:mysql://localhost:3306/mysql")
.option("user", "root")
.option("password", "hadoop")
.option("dbtable", "train")
.option("driver", "com.mysql.jdbc.Driver")
.load()
train.printSchema
train.show
```

- Spark needs to be launched with mysql driver (spark启动时候需要加载mysql驱动)
- spark-shell --jars /usr/hdp/current/hive-client/lib /mysql-connector-java.jar

• Spark-SQL Shell

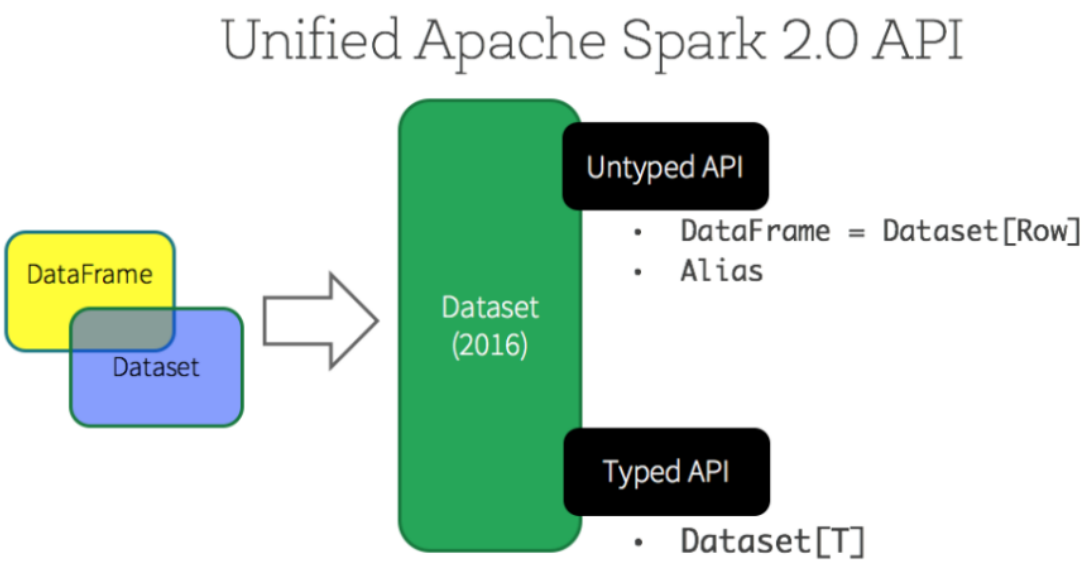
```
# 案例 spark-sql进入交互式环境
show databases;
use db2;
select * from employees limit 10;
create table montreal(name string, phone_number string, office_address string);
insert into montreal values('zs', '111-222-333 ', 'beijing');
select t.name, m.phone_number, t.office_address, m.office_address from employees
t inner
join montreal m on t.phone_number = m.phone_number;
```

• What is DataSet

- A Dataset is a collection of strongly-typed JVM objects and then can be manipulated using functional transformations (map, flatMap, filter, etc) (数据集是强类型JVM对象的集合，然后可以使用功能转换(map，flatMap，filter等)进行操作)
    - Instead of using Java or Kryo serialization, objects in a DataSet are serialized by using a specialized

      Encoder for processing and transmitting over the network.(代替使用Jav a或Kryo序列化，通过使用专用的Encoder对DataSet中的对象进行序列化，以通过网络进行处理和传输)
    - Encoders are code generated dynamically and use a format that allows Spark to perform many(Encoder是动态生成的代码，spark对这种格式的数据可以直接进行很多的操作)
    - operations like filtering, sorting and hashing without de-serializing the bytes back into an object.(类似于filter,sort,hash这些操作不需要反序列化为object)

Unified Spark 2.0 API(统一API)



Dataset from Case Class(生成DS)

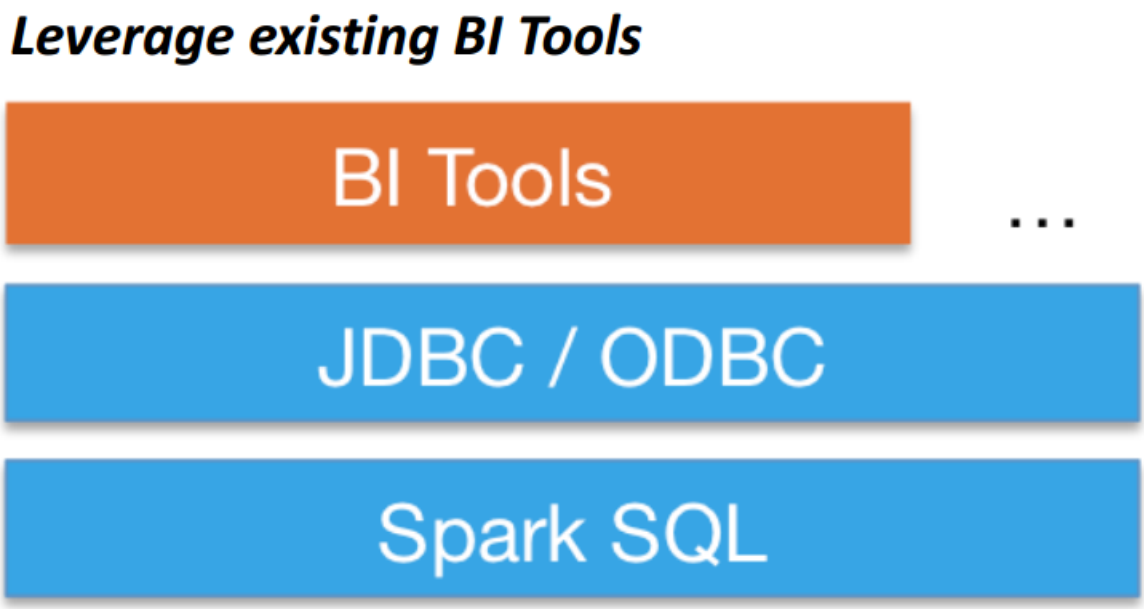
```
//案例1
case class UserFriend(name: String, friend_count: Int)
import org.apache.spark.sql.catalyst.ScalaReflection
val schema =
ScalaReflection.schemaFor[UserFriend].dataType.asInstanceOf[StructType]
val ufs =
spark.sparkContext.textFile("file:///root/events/data/user_friends.csv")
.map(line => line.split(",", -1))
.map(x => UserFriend(x(0), x(1).split(" ", -1).length))
val ds = spark.createDataset(ufs)
ds.show

val ds1 = ds.filter(x => x.friend_count > 500)
```

```
import org.apache.spark.sql.functions._
val ds2 = ds.sort($"friend_count".desc)

//案例2
case class Point(label: String, x: Double, y: Double)
case class Category(id: Long, name: String)
val points = Seq(Point("bar", 3.0, 5.6), Point("foo", -1.0, 3.0)).toDS
val categories = Seq(Category(1, "foo"), Category(2, "bar")).toDS
points.join(categories, points("label") === categories("name"), "inner")
```

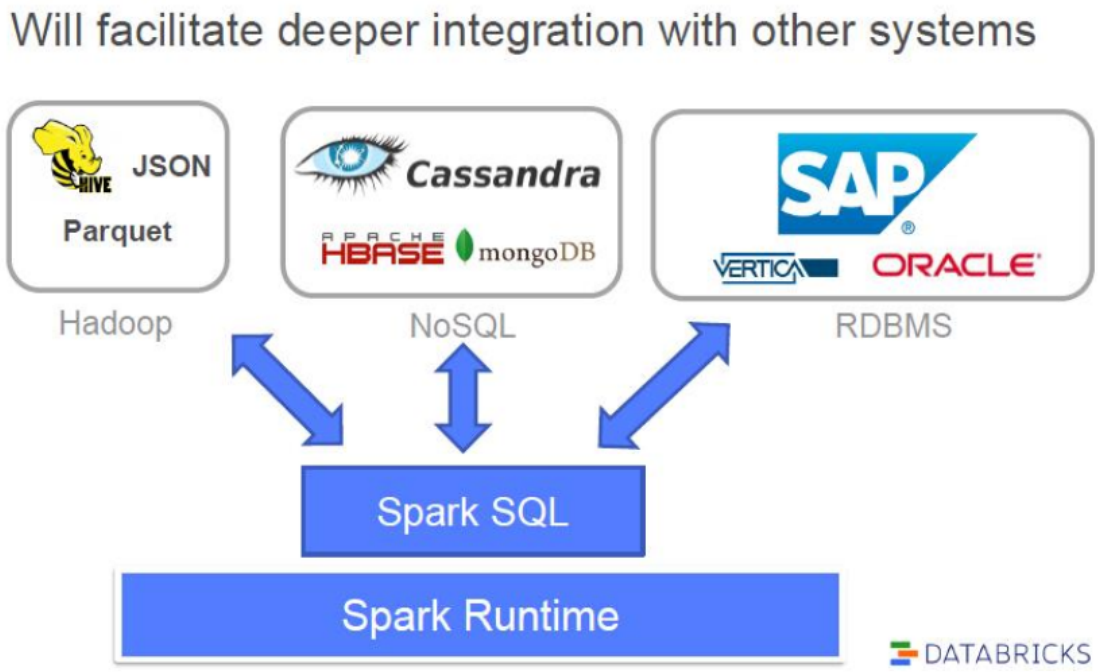
- DataFrame → DataSet(DF转换DS)

```
//案例 DF转换DS
import org.apache.spark.sql.Row
val ufs = sparkSession.sparkContext.textFile("/user/events/user_friends/csv")
.map(line => line.split(",", -1))
.map(uf => uf(1).split(" ", -1).map(f => (uf(0), f)))
.map(r => Row(r(0), r(1)))


import org.apache.spark.sql.types._
val schema = StructType(Array(StructField("user", StringType, false),
StructField("friend", StringType, false)))
val df = sparkSession.createDataFrame(ufs, schema)
case class PersonFriend(user: String, friend: String)
val ds = df.as[PersonFriend]
ds.show
```

- Spark SQL with BI Tools



- Spark SQL with Other Integrations



• RDD vs. DF

- RDD is lazily evaluated immutable parallel collection of objects exposed with lambda functions – transformations & actions(RDD是用lambda函数——转换和操作——来延迟的计算不可变的对象并行集合)
    - Drawback: Garbage Collection & Java (little better with Kryo) Serialization are expensive especially
      when data grows (垃圾回收和Java（使用Kryo更好）序列化非常昂贵，尤其是当数据增长时)
- DataFrame is an abstraction which gives a schema view of data. It offers huge performance improvement over RDDs:(DataFrame是一种提供数据格式视图的抽象。与RDDs相比，它提供了巨大的性能改进)
    - Custom Memory management - Data is stored in off-heap memory in binary format, which avoids
      the java serialization. Thus, there is no garbage collection overhead(自定义内存管理——数据以二进制格式存储在堆外内存中，从而避免了java序列化。因此，没有垃圾收集开销)
    - Optimized Execution Plans - Query plans are created for execution using Spark catalyst optimizer(优化执行计划——使用Spark catalyst optimizer为执行创建查询计划)
    - Drawbacks(缺点)
        - Compile-time type safety is not supported(不支持编译时类型安全检查)
        - Cannot operate on domain Object(无法操作对象)

• When to use RDD?

- You want low-level transformation and actions and control on your dataset;(需要对数据集进行底层转换、操作和控制;)

- Your data is unstructured, such as media streams or streams of text;(数据是非结构化的，如媒体流或文本流)
- You don't care about imposing a schema, such as columnar format, while processing or accessing data attributes by name or column; (???)
- You want to manipulate your data with functional programming constructs than domain specific expressions;(操作数据是面向函数编程，而不是操作对象的特定API)
- You can forgo some optimization and performance benefits available with DataFrames and Datasets for structured and semi-structured data (对于结构化和半结构化数据，可以放弃DataFrames和DataSet提供的一些优化和性能优势)

DF vs. DataSet

- DataFrame and Dataset APIs are built on top of the Spark SQL engine(DataFrame和Dataset api构建在Spark SQL引擎之上)
    - Both use the Catalyst optimizer to generate an optimized logical and physical query plan.(两者都使用Catalyst optimizer来生成优化的逻辑和物理查询计划)
    - Dataset[T] typed API is optimized for data engineering tasks;(Dataset[T]类型化API针对数据工程任务进行了优化)
    - The untyped Dataset[Row] (an alias of DataFrame) is even faster and suitable for interactive analysis.(非类型化数据集[Row] (DataFrame的别名)甚至更快，更适合于交互式分析)
    - Spark as a compiler understands the Dataset type JVM object, it maps the typespecific JVM object to Tungsten's internal memory representation using Encoders. As a result, Tungsten Encoders can efficiently serialize/deserialize JVM objects as well as generate compact bytecode that can execute at superior speeds(当编译器了解数据集类型的JVM对象时，Spark会使用编码器将特定于类型的JVM对象映射到Tungsten的内部内存表示形式。 结果，钨丝编码器可以有效地对JVM对象进行序列化/反序列化，并生成可以以更高速度执行的紧凑字节码)

When to use DF & DataSet

- If you want rich semantics, high-level abstractions, and domain specific APIs, use DataFrame or Dataset.(如果要使用丰富的语义，高级抽象和特定于域的API，请使用DataFrame或Dataset)
- If your processing demands high-level expressions, filters, maps, aggregation, averages, sum, SQL queries, columnar access and use of lambda functions on semi-structured data, use DataFrame or Dataset.(如果处理需要高级表达式，filters，maps，aggregation，averages，sum，SQL查询，列访问以及对半结构化数据使用lambda函数，请使用DataFrame或Dataset)
- If you want higher degree of type-safety at compile time, want typed JVM objects, take advantage of Catalyst optimization, and benefit from Tungsten's efficient code generation, use Dataset.(希望在编译时获得更高的类型安全性，希望使用类型化的JVM对象，利用Catalyst优化并从Tungsten的高效代码生成中受益，请使用数据集。)

- If you want unification and simplification of APIs across Spark Libraries, use DataFrame or Dataset.(如果要在Spark库中统一和简化API，请使用DataFrame或Dataset)
- If you are a R user, use DataFrames.(如果您是R用户，请使用DataFrames)
- If you are a Python user, use DataFrames and resort back to RDDs if you need more control. (如果您是Python用户，请使用DataFrames，如果需要更底层的操作，请回到RDDs)

spark Performance Tuning (性能调优1)

- Serialization plays an important role in the performance of any distributed application:(序列化在任何分布式应用程序的性能中都扮演着重要的角色)
  - Java Serialization – this is the default serialization for Spark to serialize objects;(Spark默认使用Java序列化接口)
  - Kryo Serialization – about 10x faster than Java serialization, but doesn't support all Serializable types(kryo性能超过java Serialization 10倍，但是不支持所有类型)
    - conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer");
- To register classes with Kyro:
  - conf.registerKryoClasses(Array(classOf[MyClass1], classOf[MyClass2]));
- If custom classes are not registered, Kryo will still work, but it will have to store the full class name with each object, which is wasteful(如果未注册自定义类，则Kryo仍然可以工作，但必须将完整的类名与每个对象一起存储，这很麻烦)

Spark Performance Tuning (性能调优2)

- Try to use array of objects, primitive types, instead of java/scala collection class;(尽量使用对象数组、基本类型，而不是java/scala集合类)
- Avoid nested structure(避免嵌套结构)
- Try to use numeric Keys, avoid use of String Keys (尽量使用数字key，避免使用字符串key)
- Use serialized form for large RDDs for persistence (对大型RDD使用序列化形式以保持持久性)

Spark Performance Tuning (性能调优3)

- When loading from CSV / JSON, only loads fields that required.(从CSV / JSON加载时，仅加载必填字段)
- Try not to use infra schema (尽量不要使用infra schema)
- Only persist intermediate results (RDD/DF/DS) when needed(仅在需要时保留中间结果(RDD / DF / DS))
- Skip un-necessary intermediate results (RDD/DF/DS) generation (跳过不必要的中间结果(RDD / DF / DS)生成)
- DF performs around 3 times faster than DS(DF的执行速度比DS快3倍)

Spark Performance Tuning (性能调优4)

- Customize RDD partitions and spark.default.parallelism:
    - Sometimes, you will get an OutOfMemoryError not because your RDDs don't fit in memory, but because the working set of one of your tasks, such as one of the reduce tasks in groupByKey, was too large. Spark's shuffle operations (sortByKey, groupByKey, reduceByKey, join, etc) build a hash table within each task to perform the grouping, which can often be large(OutOfMemoryError的原因不是因为您的RDD不能容纳在内存中，而是因为执行任务(例如groupByKey中的reduce任务之一)的工作集太大。 Spark的shuffle操作（sortByKey，groupByKey，reduceByKey，join等）会在每个任务中建立一个哈希表来执行分组，通常这可能会很大)
    - The simplest fix here is to increase the level of parallelism, so that each task's input set is smaller. Spark can efficiently support tasks as short as 200 ms, because it reuses one executor JVM across many tasks and it has a low task launching cost, so you can safely increase the level of parallelism to more than the number of cores in your clusters(此处最简单的解决方法是提高并行度，以使每个任务的输入集更小。 Spark可以高效地支持短至200ms的任务，因为它可以在多个任务中重用一个执行器JVM，并且任务启动成本低，因此可以安全地将并行度提高到集群中核心的数量以上)
- Broadcast large variables;(广播大变量)
- Try to process local data and minimize the data transfers across worker nodes(尝试处理本地数据，并尽量减少跨工作节点的数据传输)

Spark SQL Performance Tuning (性能调优5)

- For joining tables: (关联表)

```
select * from t1 join t2 on t1.name = t2.full_name where t1.name = ''mike' and
t2.full_name ='mike'
```

- put the largest table first (把大表放在前面)
- Broadcast the smallest table (广播小表)
- Minimize the number of tables in Join (最小化联接中的表数)

Summary(总结)

- 了解了Spark SQL API
- 分析了Spark SQL优化器 – Catalyst Optimizer的 工作原理
- 深入学习了DataFrame与Dataset, 及其操作运算；
- 练习怎样用Spark装载CSV， JSON数据
- 了解了Spark性能优化
- 学习了Spark SQL与Hive的集成

作业

- 谁是最大买“货”(谁购买的最多，以￥算)
- 哪个产品是最大卖货?(哪个产品销售的最多)
- 找出购买的周分布
- 找出购买力最强地域