12. SparkSQL自定义函数

UDF函数

用户自定义函数

```
import org.apache.spark.sql.{DataFrame, SparkSession}
import org.apache.spark.{SparkConf}

object SparkSQL {
  def main(args:Array[String]):Unit = {
  //创建SparkConf()并设置App名称
    val conf = new SparkConf().setAppName("SparkSQLDemo").setMaster("local")
    val spark = SparkSession.builder().config(conf).getOrCreate()
    val df: DataFrame = spark.read.json("dir/people.json")
    //注册函数,在整个应用中可以使用
    val addName = spark.udf.register("addName", (x: String) => "Name:" + x)
    df.createOrReplaceTempView("people")
    spark.sql("Select addName(name), age from people").show()
    spark.stop()
  }
}
```

UDAF函数

用户自定义聚合函数

12.1 UDAF函数支持DataFrame(弱类型)

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。

ps:弱类型指的是在编译阶段是无法确定数据类型的,而是在运行阶段才能创建类型

```
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
//自定义UDAF函数
class MyAverage extends UserDefinedAggregateFunction {
  // 输入数据
  def inputSchema: StructType = StructType(List(StructField("Salary",DoubleType,true)))
  // 每一个分区中的 共享变量 存储记录的值
  def bufferSchema: StructType = {
    //                    工资的总和                    工资的总数
    StructType(StructField("sum", DoubleType):: StructField("count", DoubleType)  :: Nil)
  }
  // 返回值的数据类型表示UDAF函数的输出类型
  def dataType: DataType = DoubleType
```

```
  //如果有相同的输入,那么是否UDAF函数有相同的输出,有true 否则false
  //UDAF函数中如果输入的数据掺杂着时间,不同时间得到的结果可能是不一样的所以这个值可以设置为false
  //若不掺杂时间,这个值可以输入为true
  def deterministic: Boolean = true

  // 初始化对Buffer中的属性初始化即初始化分区中每一个共享变量
  def initialize(buffer: MutableAggregationBuffer): Unit = {
    // 存工资的总额
    buffer(0) = 0.0//取得就是sum
    // 存工资的个数
    buffer(1) = 0.0//取得就是count
  }
  // 相同Execute间的数据合并,合并小聚合中的数据即每一个分区中的每一条数据聚合的时候需要调用的方法
  /*
   第一个参数buffer还是共享变量
   第二个参数是一行数据即读取到的数据是以一行来看待的
   */
  def update(buffer: MutableAggregationBuffer, input: Row): Unit = {
    if (!input.isNullAt(0)) {
      //获取这一行中的工资,然后将工资添加到该方法中
      buffer(0) = buffer.getDouble(0) + input.getDouble(0)
      //将工资的个数进行加1操作最终是为了计算所有的工资的个数
     buffer(1) = buffer.getDouble(1) + 1
    }
  }
  // 不同Execute间的数据合并,合并大数据中的数即将每一个区分的输出合并形成最后的数据
  def merge(buffer1: MutableAggregationBuffer, buffer2: Row): Unit = {
    //合并总的工资
    buffer1(0) = buffer1.getDouble(0) + buffer2.getDouble(0)
    //合并总的工资个数
    buffer1(1) = buffer1.getDouble(1) + buffer2.getDouble(1)
  }
  // 计算最终结果
  def evaluate(buffer: Row): Double = buffer.getDouble(0) / buffer.getDouble(1)


}

object MyAverage{
  def main(args: Array[String]): Unit = {
    val spark = SparkSession.builder().appName("MyAverage").master("local[*]").getOrCreate()
    // 注册函数
    spark.udf.register("myAverage",new MyAverage)

    val df = spark.read.json("dir/employees.json")
    df.createOrReplaceTempView("employees")
    df.show()
    //虽然没有使用groupby那么会将整个数据作为一个组
    val result = spark.sql("SELECT myAverage(salary) as average_salary FROM employees")
    result.show()


  }
```

```
}
```

12.2.UDAF函数支持DataSet(强类型)

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

ps:在编译阶段就确定了数据类型

```
package Day01

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
//自定义UDAF函数
// 既然是强类型，可能有case类
case class Employee(name: String, salary: Double)
case class Average(var sum: Double, var count: Double)
//依次配置输入,共享变量,输出的类型,需要使用到case class
class MyAverage extends Aggregator[Employee, Average, Double] {
  // 初始化方法 初始化每一个分区中的 共享变量即定义一个数据结构，保存工资总数和工资总个数，初始都为0
  def zero: Average = Average(0.0, 0.0)
  //每一个分区中的每一条数据聚合的时候需要调用该方法
  def reduce(buffer: Average, employee: Employee): Average = {
    buffer.sum += employee.salary
    buffer.count += 1
    buffer
  }
  //将每一个分区的输出 合并 形成最后的数据
  def merge(b1: Average, b2: Average): Average = {
    b1.sum += b2.sum
    b1.count += b2.count
    b1
  }
  // 给出计算结果
  def finish(reduction: Average): Double = reduction.sum / 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
}

object MyAverage{
  def main(args: Array[String]): Unit = {
    val spark = SparkSession.builder().appName("MyAverage").master("local[*]").getOrCreate()
    import spark.implicits._
    val ds = spark.read.json("dir/employees.json").as[Employee]
    ds.show()
    val averageSalary = new MyAverage().toColumn.name("average_salary")

    val result = ds.select(averageSalary)
```

```
    result.show()
  }

}
```

13. 内置函数

内置函数

内置函数包含了五大基本类型：
1、聚合函数，例如countDistinct、sumDistinct等；
2、集合函数，例如sort_array、explode等
3、日期、时间函数，例如hour、quarter、next_day
4、数学函数，例如asin、atan、sqrt、tan、round等；
5、开窗函数，例如rowNumber等
6、字符串函数，concat、format_number、rexexp_extract
7、其它函数，isNaN、sha、randn、callUDF

```
import org.apache.spark.sql.types.{DoubleType, StringType, StructField, StructType}
import org.apache.spark.sql.{Row, SparkSession}

object DataFrameCreate {
  def main(args: Array[String]): Unit = {
    val sparkSession = SparkSession.builder()
      .appName("sparksql1")
      .master("local[2]")
      .getOrCreate()
    //val df = ssc.read.text("D:\\wc.txt")
    import sparkSession.implicits._
    // 模拟数据
    val userSaleLog = Array("2015-10-01,55.05,1122",
      "2015-10-01,23.15,1133",
      "2015-10-01,15.20,",
      "2015-10-02,56.05,1144",
      "2015-10-02,78.87,1155",
      "2015-10-02,113.02,1123")
    val userSaleLogRDD = sparkSession.sparkContext.parallelize(userSaleLog, 5)
    // 进行有效销售日志的过滤
    val filteredUserSaleLogRDD = userSaleLogRDD
      .filter { log => if (log.split(",").length == 3) true else false }
    val userSaleLogRowRDD = filteredUserSaleLogRDD
      .map { log => Row(log.split(",")(0), log.split(",")(1).toDouble) }
    val structType = StructType(Array(
      StructField("date", StringType, true),
      StructField("sale_amount", DoubleType, true)))
    val userSaleLogDF = sparkSession.createDataFrame(userSaleLogRowRDD, structType)
    // 开始进行每日销售额的统计
    import org.apache.spark.sql.functions._

    userSaleLogDF.groupBy("date")
      .agg(sum("sale_amount"))
```

```
      .collect()
      .foreach(println)
  }
}
```

14 窗口函数

说明:
rank ( ) 跳跃排序，有两个第二名时后边跟着的是第四名
dense_rank() 连续排序，有两个第二名时仍然跟着第三名
over ( ) 开窗函数：
　　在使用聚合函数后，会将多行变成一行，而开窗函数是将一行变成多行；
　　并且在使用聚合函数后，如果要显示其他的列必须将列加入到group by中，
　　而使用开窗函数后，可以不使用group by，直接将所有信息显示出来。
　　开窗函数适用于在每一行的最后一列添加聚合函数的结果。
常用开窗函数：
　1.为每条数据显示聚合信息.(聚合函数() over())
　2.为每条数据提供分组的聚合函数结果(聚合函数() over(partition by 字段) as 别名)
　　--按照字段分组，分组后进行计算
　3.与排名函数一起使用(row number() over(order by 字段) as 别名)
常用分析函数：（最常用的应该是1.2.3 的排序）
　1、row_number() over(partition by ... order by ...)
　2、rank() over(partition by ... order by ...)
　3、dense_rank() over(partition by ... order by ...)
　4、count() over(partition by ... order by ...)
　5、max() over(partition by ... order by ...)
　6、min() over(partition by ... order by ...)
　7、sum() over(partition by ... order by ...)
　8、avg() over(partition by ... order by ...)
　9、first_value() over(partition by ... order by ...)
　10、last_value() over(partition by ... order by ...)
　11、lag() over(partition by ... order by ...)
　12、lead() over(partition by ... order by ...)
lag 和lead 可以 获取结果集中，按一定排序所排列的当前行的上下相邻若干offset 的某个行的某个列(不用结果集的自关联）；
lag ，lead 分别是向前，向后；
lag 和lead 有三个参数，第一个参数是列名，第二个参数是偏移的offset，第三个参数是 超出记录窗口时的默认值

```
import org.apache.spark.sql.SparkSession

//开窗窗函数
object MyAverage  {

  def main(args: Array[String]): Unit = {
    val spark = SparkSession.builder().appName("MyAverage").master("local[*]").getOrCreate()
    import spark.implicits._
    val df = spark.read.json("dir/Score.json")
    df.createOrReplaceTempView("score")
    df.show()

    println("/**************  求每个班最高成绩学生的信息（groupBY）  ***************/")
    //先写的 进行查询  这样可以进行班级分组并得到分数
    spark.sql("select class, max(score) max from score group by class").show()
```

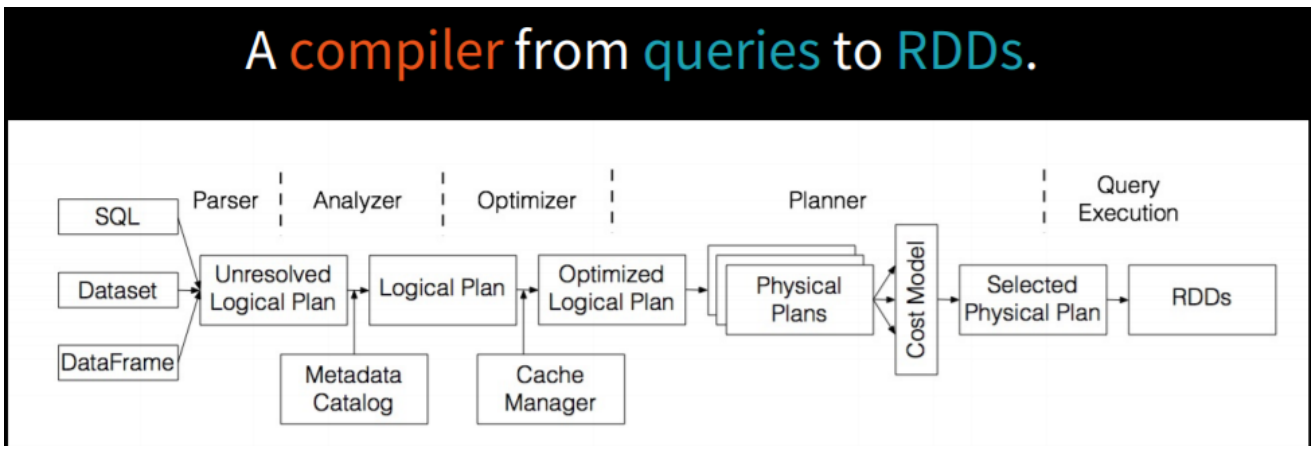
```
    //这句话在集合Sparkcore想一样
    /*这句话select class, max(score) max from score group by class
      相当于key是class  然后执行了group by  求了一个max(score) 得到一个RDD然后和原有RDD执行了一个Join
      直白一些(class相当于key,score相当于value) 然后执行groupbyKey  --> 生成了(class,max) 这个RDD
      然后将(class,max)转了一下 --> (class_max,1)后面的数无所谓
      然就相当于将数据中也做了一个(class_max,score)  -->和之前的RDD -->进行join --> 在进行map
    */
    spark.sql("select a.name, b.class, b.max from score a, " +
      "(select class, max(score) max from score group by class) as b " +
      "where a.score = b.max").show()
      //添加一个函数可以对表中动态添加一列即 以对表中的分数添加排序
      //这里的over就是开窗函数 , row_number是分析函数(排名函数)
    spark.sql("select name,class,score,row_number() over(partition by class order by score desc)
rank from score").show()
        println("    /*******  计算结果的表  *******")
    spark.sql("select * from " +
      "( select name,class,score,rank() over(partition by class order by score desc) rank from
score) " +
      "as t " +
      "where t.rank=1").show()

    println("//***************  求每个班最高成绩学生的信息  ***************/")
    println("    /*******  开窗函数的表  ********/")
    spark.sql("select name,class,score, rank() over(partition by class order by score desc) rank
from score").show()

    println("    /*******  计算结果的表  *******")
    spark.sql("select * from " +
      "( select name,class,score,rank() over(partition by class order by score desc) rank from
score) " +
      "as t " +
      "where t.rank=1").show()
  }

}
```

15. Spark SQL 深入



16. spark SQL join深入

Join是SQL语句中的常用操作，良好的表结构能够将数据分散在不同的表中，使其符合某种范式，减少表冗余、更新容错等。而建立表和表之间关系的最佳方式就是Join操作。

SparkSQL作为大数据领域的SQL实现，自然也对Join操作做了不少优化，今天主要看一下在SparkSQL中对于Join，常见的3种实现。

SparkSQL的3种Join实现

Broadcast Join

大家知道，在数据库的常见模型中（比如星型模型或者雪花模型），表一般分为两种：事实表和维度表。维度表一般指固定的、变动较少的表，例如联系人、物品种类等，一般数据有限。而事实表一般记录流水，比如销售清单等，通常随着时间的增长不断膨胀。

因为Join操作是对两个表中key值相同的记录进行连接，在SparkSQL中，对两个表做Join最直接的方式是先根据key分区，再在每个分区中把key值相同的记录拿出来做连接操作。但这样就不可避免地涉及到shuffle，而shuffle在Spark中是比较耗时的操作，我们应该尽可能的设计Spark应用使其避免大量的shuffle。

当维度表和事实表进行Join操作时，为了避免shuffle，我们可以将大小有限的维度表的全部数据分发到每个节点上，供事实表使用。executor存储维度表的全部数据，一定程度上牺牲了空间，换取shuffle操作大量的耗时，这在SparkSQL中称作Broadcast Join，如下图所示：

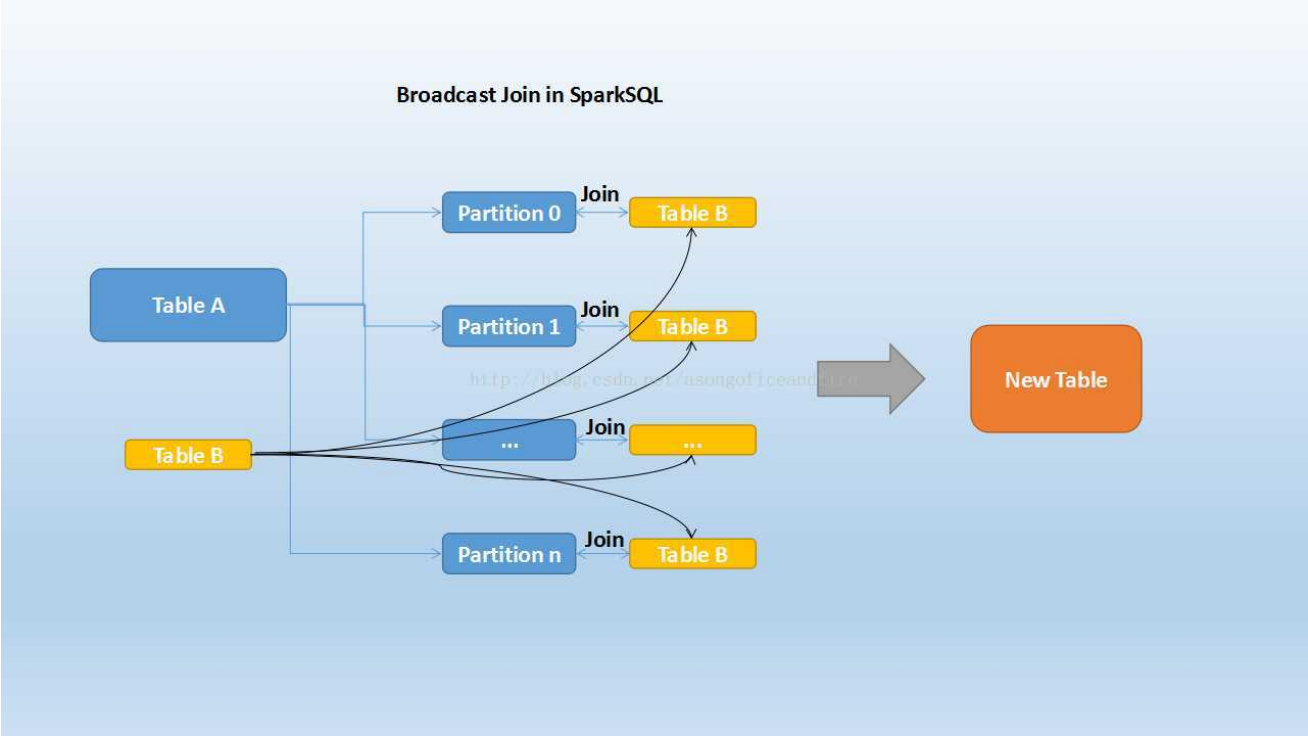


Table B是较小的表，黑色表示将其广播到每个executor节点上，Table A的每个partition会通过block manager取到Table A的数据。根据每条记录的Join Key取到Table B中相对应的记录，根据Join Type进行操作。这个过程比较简单，不做赘述。

Broadcast Join的条件有以下几个：

\1. 被广播的表需要小于spark.sql.autoBroadcastJoinThreshold所配置的值，默认是10M（或者加了broadcast join的hint）

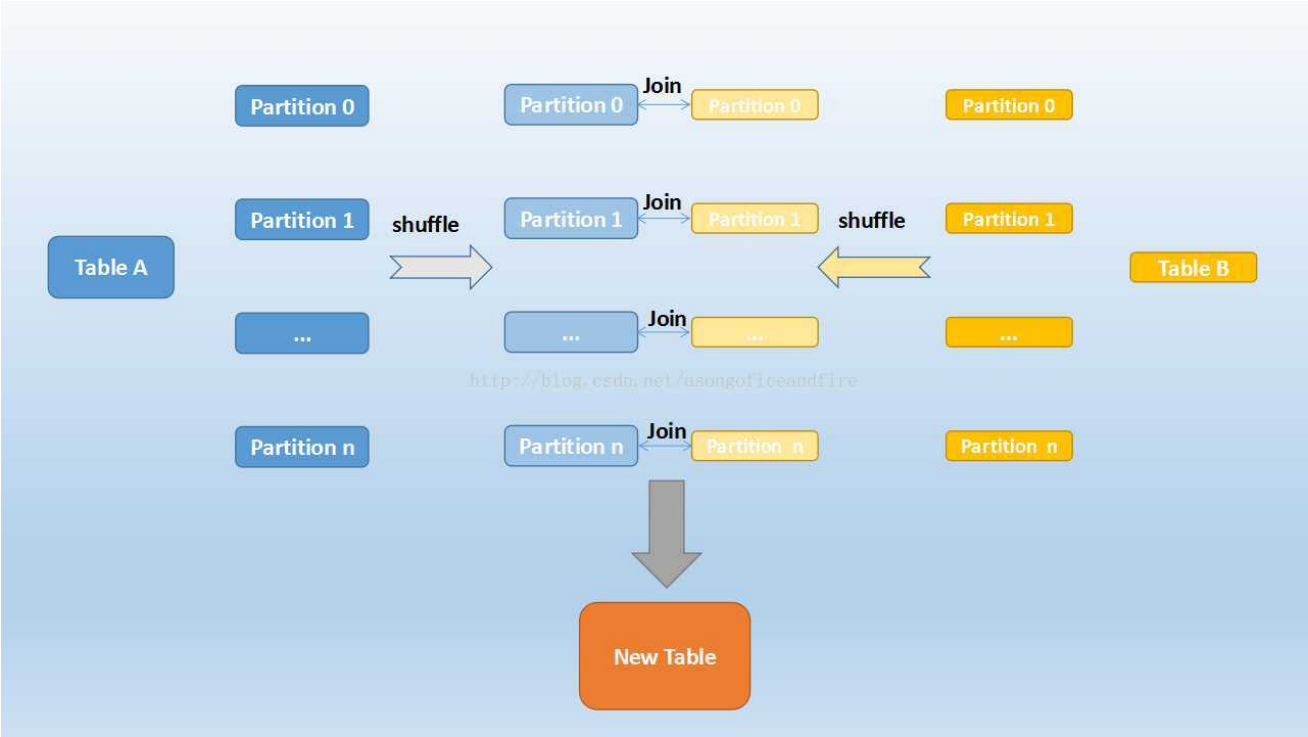
\2. 基表不能被广播，比如left outer join时，只能广播右表

看起来广播是一个比较理想的方案，但它有没有缺点呢？也很明显。这个方案只能用于广播较小的表，否则数据的冗余传输就远大于shuffle的开销；另外，广播时需要将被广播的表现collect到driver端，当频繁有广播出现时，对driver的内存也是一个考验。

Shuffle Hash Join

当一侧的表比较小时，我们选择将其广播出去以避免shuffle，提高性能。但因为被广播的表首先被collect到driver段，然后被冗余分发到每个executor上，所以当表比较大时，采用broadcast join会对driver端和executor端造成较大的压力。

但由于Spark是一个分布式的计算引擎，可以通过分区的形式将大批量的数据划分成n份较小的数据集进行并行计算。这种思想应用到Join上便是Shuffle Hash Join了。利用key相同必然分区相同的这个原理，SparkSQL将较大表的join分而治之，先将表划分成n个分区，再对两个表中相对应分区的数据分别进行Hash Join，这样即在一定程度上减少了driver广播一侧表的压力，也减少了executor端取整张被广播表的内存消耗。其原理如下图：



Shuffle Hash Join分为两步：

\1. 对两张表分别按照join keys进行重分区，即shuffle，目的是为了让有相同join keys值的记录分到对应的分区中

\2. 对对应分区中的数据进行join，此处先将小表分区构造为一张hash表，然后根据大表分区中记录的join keys值拿出来进行匹配

Shuffle Hash Join的条件有以下几个：

\1. 分区的平均大小不超过spark.sql.autoBroadcastJoinThreshold所配置的值，默认是10M

\2. 基表不能被广播，比如left outer join时，只能广播右表

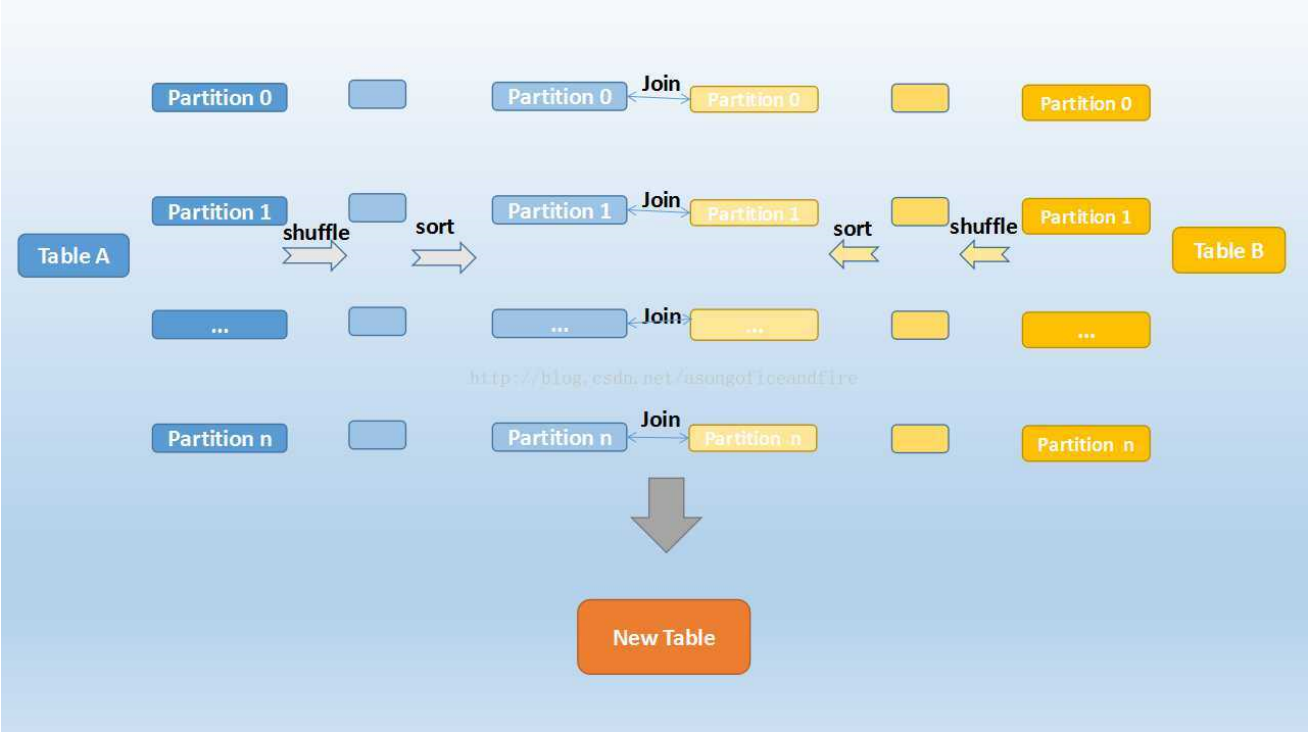
\3. 一侧的表要明显小于另外一侧，小的一侧将被广播（明显小于的定义为3倍小，此处为经验值）

我们可以看到，在一定大小的表中，SparkSQL从时空结合的角度来看，将两个表进行重新分区，并且对小表中的分区进行hash化，从而完成join。在保持一定复杂度的基础上，尽量减少driver和executor的内存压力，提升了计算时的稳定性。

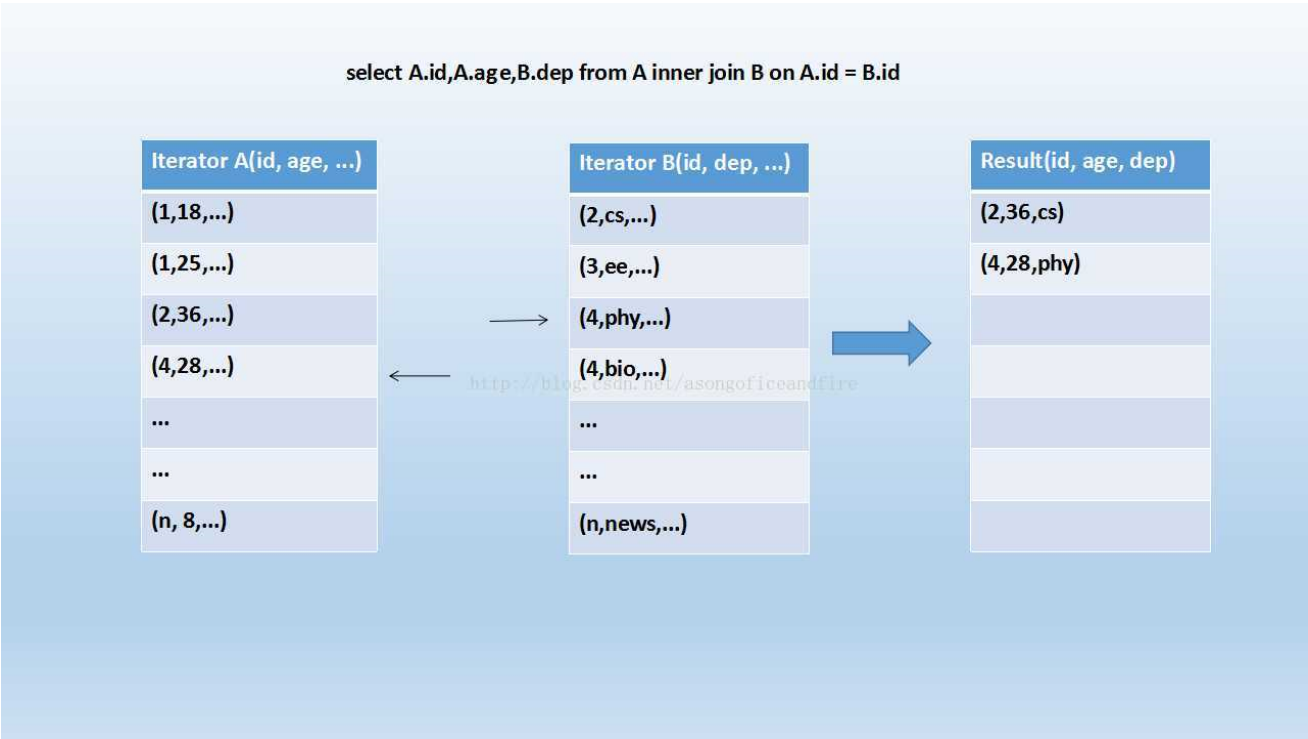
Sort Merge Join

上面介绍的两种实现对于一定大小的表比较适用，但当两个表都非常大时，显然无论适用哪种都会对计算内存造成很大压力。这是因为join时两者采取的都是hash join，是将一侧的数据完全加载到内存中，使用hash code取join keys值相等的记录进行连接。

当两个表都非常大时，SparkSQL采用了一种全新的方案来对表进行Join，即Sort Merge Join。这种实现方式不用将一侧数据全部加载后再进星hash join，但需要在join前将数据排序，如下图所示：



可以看到，首先将两张表按照join keys进行了重新shuffle，保证join keys值相同的记录会被分在相应的分区。分区后对每个分区内的数据进行排序，排序后再对相应的分区内的记录进行连接，如下图示：



看着很眼熟吧？也很简单，因为两个序列都是有序的，从头遍历，碰到key相同的就输出；如果不同，左边小就继续取左边，反之取右边。

可以看出，无论分区有多大，Sort Merge Join都不用把某一侧的数据全部加载到内存中，而是即用即取即丢，从而大大提升了大数据量下sql join的稳定性。

本文介绍了SparkSQL中的3中Join实现，其实这也不是什么新鲜玩意儿。传统DB也有这也的玩法儿，SparkSQL只是将其做成分布式的实现。

本文仅仅从大的理论方面介绍了这几种实现，具体到每个join type是怎么遍历、没有join keys时应该怎么做、这些实现对join keys有什么具体的需求，这些细节都没有展现出来。感兴趣的话，可以去翻翻源码。

```
import org.apache.spark.sql.{DataFrame, Dataset, SparkSession}

object JoinTest {

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

    val spark = SparkSession
      .builder()
      .appName("JoinTest")
      .master("local[*]")
      .getOrCreate()

    import spark.implicits._
    val lines: Dataset[String] = spark.createDataset(List("1,zhangsan,china", "2,lisi,usa",
"3,wangwu,jp"))
    //对数据进行整理

    val tpDs: Dataset[(Long, String, String)] = lines.map(line => {
      val fields = line.split(",")
      val id = fields(0).toLong
      val name = fields(1)
      val nationCode = fields(2)
      (id, name, nationCode)
    })
    val df1 = tpDs.toDF("id", "name", "nation")

    val nations: Dataset[String] = spark.createDataset(List("china,中国", "usa,美国"))
    //对数据进行整理
    val ndataset: Dataset[(String, String)] = nations.map(l => {
      val fields = l.split(",")
      val ename = fields(0)
      val cname = fields(1)
      (ename, cname)
    })

    val df2 = ndataset.toDF("ename","cname")


    df2.count()

    //第一种，创建视图
    //df1.createTempView("v_users")
    //df2.createTempView("v_nations")

    //val r: DataFrame = spark.sql("SELECT name, cname FROM v_users JOIN v_nations ON nation =
```

```
ename")


    //import org.apache.spark.sql.functions._
    val r = df1.join(df2, $"nation" === $"ename", "left_outer")

    //

   r.show()

    spark.stop()

  }
}
```

方式2：

```
import org.apache.spark.sql.{DataFrame, SparkSession}

/**
  * Created by zx on 2017/10/16.
  */
object JoinTest {

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


    val spark = SparkSession.builder().appName("CsvDataSource")
      .master("local[*]")
      .getOrCreate()

    import spark.implicits._
    //import org.apache.spark.sql.functions._

    //spark.sql.autoBroadcastJoinThreshold=-1
    spark.conf.set("spark.sql.autoBroadcastJoinThreshold", -1)
    //spark.conf.set("spark.sql.join.preferSortMergeJoin", true)

    //println(spark.conf.get("spark.sql.autoBroadcastJoinThreshold"))

    val df1 = Seq(
      (0, "playing"),
      (1, "with"),
      (2, "join")
    ).toDF("id", "token")

    val df2 = Seq(
      (0, "P"),
      (1, "W"),

      (2, "S")
```

```
    ).toDF("aid", "atoken")


    df2.repartition()

    //df1.cache().count()

    val result: DataFrame = df1.join(df2, $"id" === $"aid")

    //查看执行计划
    result.explain()

    result.show()

    spark.stop()


  }

}
```