# Spark SQL 架构简介

网址: <https://mp.weixin.qq.com/s/HbUnZcVwusEQQsq8LveeuA>

或者: <https://www.infoq.cn/article/xEwaUj8RN74lvbRpTBa5>

## 口述流程

SparkSQL, 要转化为Physical Plan, 才能被Spark执行.

首先, SparkSQL被Parser成Unresolvered Logical Plan, 这是没有schema的, Analyzer从Hive Matedata中取schema, Analyze成Resolverd Logical Plan.

其次, Optimizer把Resolvered Logical Plan优化, 得到Opitimized Logical Plan.

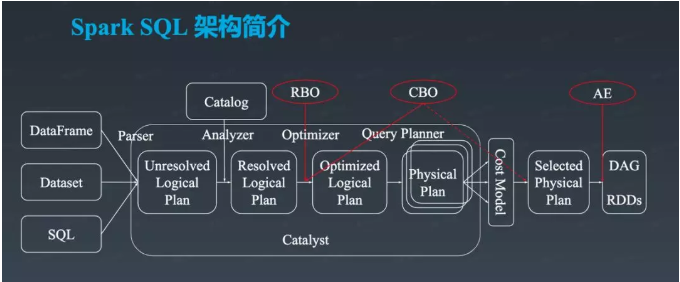
最后, join大概有3种方式, ShuffledHashJoin、BroadcastHashJoin、SortMergeJoin.

有一个称之为CostModel的工具, 将最优的方式, 解析成Phisical Plan, 交由Spark执行.

执行过程中, Phisical Plan, 可能由于其他原因, 不是最优的.

Spark还提供了Adtick Execution去动态优化他的CostModel.

## 流程图



|  |
| --- |
| 一条 SQL 提交之后会被 Parser 解析并转化为 Unresolved Logical Plan。  它的重点是 Logical Plan 也即逻辑计划，它描述了希望做什么样的查询。  Unresolved 是指该查询相关的一些信息未知，比如不知道查询的目标表的 Schema 以及数据位置。  上述信息存于 Catalog 内。  在生产环境中，一般由 Hive Metastore 提供 Catalog 服务。  Analyzer 会结合 Catalog 将 Unresolved Logical Plan 转换为 Resolved Logical Plan。  到这里还不够。不同的人写出来的SQL 不一样，生成的 Resolved Logical Plan 也就不一样，执行效率也不一样。  为了保证无论用户如何写 SQL 都可以高效的执行，Spark SQL 需要对 Resolved Logical Plan 进行优化，这个优化由 Optimizer 完成。  Optimizer 包含了一系列规则，对 Resolved Logical Plan 进行等价转换，最终生成 Optimized Logical Plan。该 Optimized Logical Plan 不能保证是全局最优的，但至少是接近最优的。  上述过程只与 SQL 有关，与查询有关，但是与 Spark 无关，因此无法直接提交给 Spark 执行。  Query Planner 负责将 Optimized Logical Plan 转换为 Physical Plan，进而可以直接由 Spark 执行。  由于同一种逻辑算子可以有多种物理实现。  如 Join 有多种实现，ShuffledHashJoin、BroadcastHashJoin、BroadcastNestedLoopJoin、SortMergeJoin 等。  因此 Optimized Logical Plan 可被 Query Planner 转换为多个 Physical Plan。如何选择最优的 Physical Plan 成为一件非常影响最终执行性能的事情。  一种比较好的方式是，构建一个 Cost Model，并对所有候选的 Physical Plan 应用该 Model 并挑选 Cost 最小的 Physical Plan 作为最终的 Selected Physical Plan。  Physical Plan 可直接转换成 RDD 由 Spark 执行。  我们经常说“计划赶不上变化”，在执行过程中，可能发现原计划不是最优的，后续执行计划如果能根据运行时的统计信息进行调整可能提升整体执行效率。  这部分动态调整由 Adaptive Execution 完成。 |

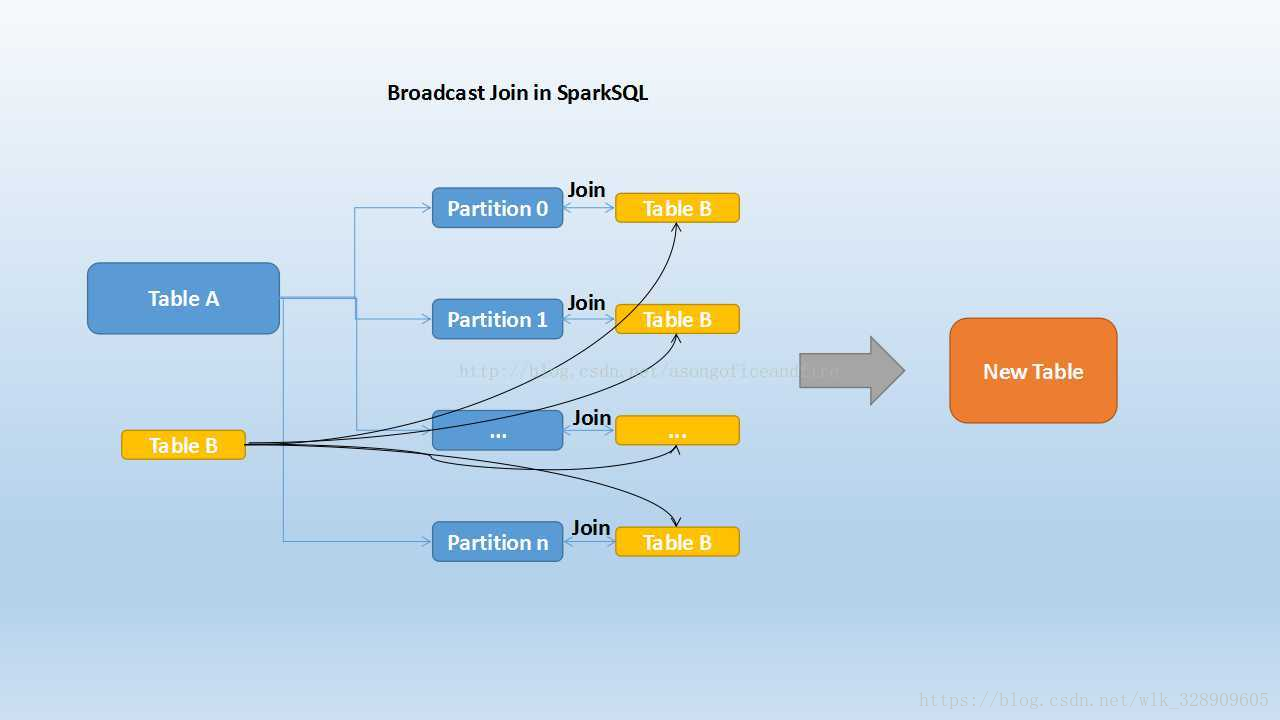
# SparkSQL的三种Join及具体实现(BroadcastJoin、ShuffleHashJoin、SortMergeJoin)

参考: <https://blog.csdn.net/wlk_328909605/article/details/82933552>

## 小表和大表Join

### BroadcastJoin

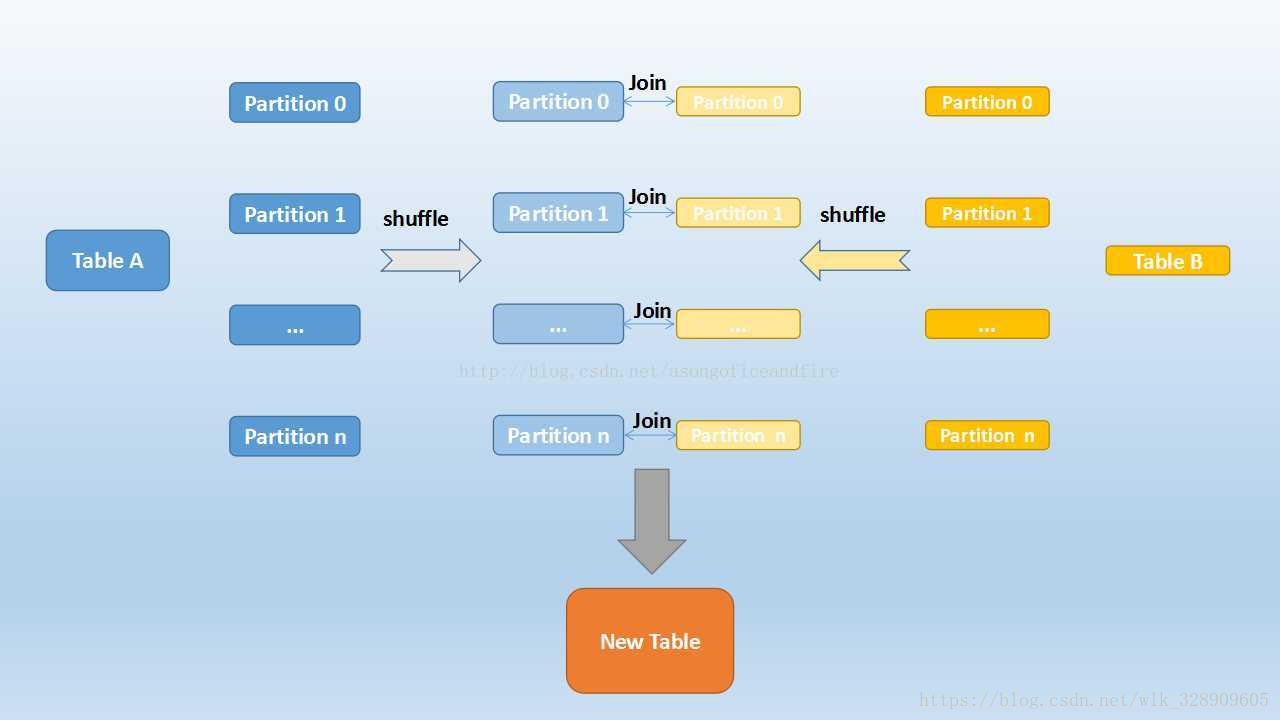
|  |
| --- |
| 介绍:  将小表的数据分发到每个节点上，供大表使用。executor存储小表的全部数据，一定程度上牺牲了空间，换取shuffle操作大量的耗时，这在SparkSQL中称作Broadcast Join  条件:   1. 被广播的表需要小于 spark.sql.autoBroadcastJoinThreshold 所配置的值，默认是10M （或者加了broadcast join的hint） 2. 基表不能被广播，比如 left outer join 时，只能广播右表 |



|  |
| --- |
| 小表足够小, 在spark.sql.autoBroadcastJoinThreshold(默认10M, 经验上最大不超过100M) 还小 |

### Shuffle Hash Join

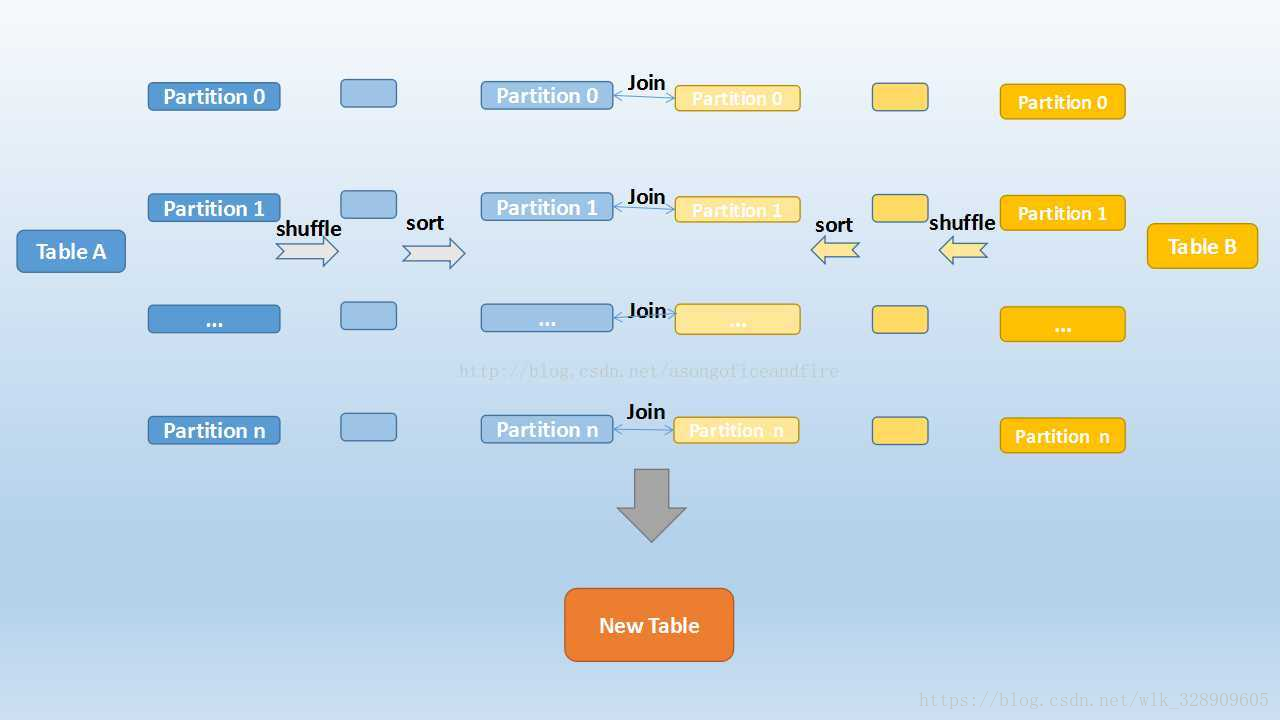
|  |
| --- |
| 因为被广播的表首先被collect到driver段，然后被冗余分发到每个executor上，所以当表比较大时，采用broadcast join会对driver端和executor端造成较大的压力。  spark可以通过分区的形式将大批量的数据划分成n份较小的数据集进行并行计算.  利用key相同必然分区相同的这个原理，SparkSQL将较大表的join分而治之，先将表划分成n个分区，再对两个表中相对应分区的数据分别进行Hash Join，  这样即在一定程度上减少了driver广播一侧表的压力，也减少了executor端取整张被广播表的内存消耗。  \*Shuffle Hash Join分为两步：  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  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：  分区的平均大小不超过spark.sql.autoBroadcastJoinThreshold所配置的值，默认是10M  基表不能被广播，比如left outer join时，只能广播右表  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） |



## 大表和大表Join

### Sort Merge Join

|  |
| --- |
| 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  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(即用即取即丢) |



# RDD、Dataframe、Dataset转化关系

## RDD转化成Dataframe

|  |  |
| --- | --- |
| 1. 前言: RDD是无schema的数据集, Dataframe是具有结构的数据集, 因此只要给RDD增加schema就可以完成转化. 2. 过程   步骤一: 通过sparkSession创建RDD   |  | | --- | | **val** rdd = spark.sparkContext.makeRDD(*Array*((**"michael"**,17),(**"nika"**,18),(**"gaga"**,19))) |   步骤二: 导入隐式转换  **import** spark.implicits.\_  步骤三: 调用toDF()函数(底层其实调用的是Dataset.toDF())  rdd.toDF(**"name"**,**"age"**) |

## Dataframe转化成RDD

|  |
| --- |
| 1. 前言: Dataframe去掉schema即为RDD. 2. 步骤: 直接调用df.rdd即可 3. Note: Dataframe转化成RDD后, 类型为RDD[Row] |

## RDD转化为Dataset

|  |  |  |
| --- | --- | --- |
| 1. 前言: Dataset是具有对象行为的数据集, 因此RDD转化为Dataset需要一个样例类. 2. 步骤:   步骤一: 通过sparkSession创建RDD   |  | | --- | | **val** rdd = spark.sparkContext.makeRDD(*Array*((**"michael"**,17),(**"nika"**,18),(**"gaga"**,19))) |   步骤二: 创建样例类  **case class** Persion(name:String, age:Int)  步骤三: 使用Map函数, 把数据转化成Person对象.   |  | | --- | | **val** ds = rdd.map(x => {  *Persion*(x.\_1, x.\_2) }).toDS() |  1. Note: 样例类在的位置涉及到访问机制. |

## Dataset转化为RDD

|  |
| --- |
| 1. 前言: Dataset去掉schema即为RDD. 2. 步骤: 直接调用ds.rdd即可 3. Note: Dataset转化成RDD后, 类型为RDD[Persion] |

## Dataframe转化为Dataset

|  |
| --- |
| 1. 前言: Dataframe转化为Dataset需要样例类. 2. 步骤:   步骤一: 样例类  **case class** Persion(name:String, age:Int)  步骤二: 转换  df.as[Persion] |

## Dataset转化为Dataframe

|  |
| --- |
| 1. 前言: Dataset可以直接调用函数转化为Dataframe, 因为Dataset具有对象, 所以Dataframe需要的schema可以直接转化. 2. 步骤: ds.toDF |

# RDD、Dataframe、Dataset的创建方式

## RDD的创建方式

### 从集合中创建RDD

|  |
| --- |
| val rdd = sc.makeRDD(Array(1,2,3)) |

### 从外部存储创建RDD

|  |
| --- |
| val rdd = sc.textFile("hdfs://hadoop102:9000/RELEASE") |

### 从其他RDD转化

|  |
| --- |
| val mapRDD = rdd.map(x=>(x,1)) |

## Dataframe的创建方式

## Dataset的创建方式