打造价值40万Offer的朋友圈
数据人的宝藏朋友圈
欢迎邀请你的同事同学参观

公众号：import_bigdata
知　乎：https://www.zhihu.com/people/wang-zhi-wu-66
CSDN ：https://blog.csdn.net/u013411339
Github：https://github.com/wangzhiwubigdata/God-of-Bigdata

本文已经加入「大数据成神之路PDF版」中提供下载，你可以关注公众号，后台回复：「**PDF**」即可获取。

更多PDF下载可以参考：**《重磅,大数据成神之路PDF可以分类下载啦!》**

Spark重点难点系列：

- **《【Spark重点难点01】你从未深入理解的RDD和关键角色》**
- **《【Spark重点难点02】你以为的Shuffle和真正的Shuffle》**
- **《【Spark重点难点03】你的数据存在哪了?》**
- **《【Spark重点难点04】你的代码跑起来谁说了算？(内存管理)》**
- **《【Spark重点难点05】SparkSQL YYDS(上)！》**

在上节课中我们讲解了Spark SQL的来源，Spark DataFrame创建的方式以及常用的算子。

Spark SQL的关联

你大概从茫茫多的网上博客中可以看到Spark SQL支持的Join有哪几种？比如NLJ(Nested Loop Join)、SMJ(Sort Merge Join)和 HJ(Hash Join)，一会又是 Shuffle Join、Broadcast Join。

好了，你是不是已经懵逼了。

下面我来告诉大家这些是怎么分类的：

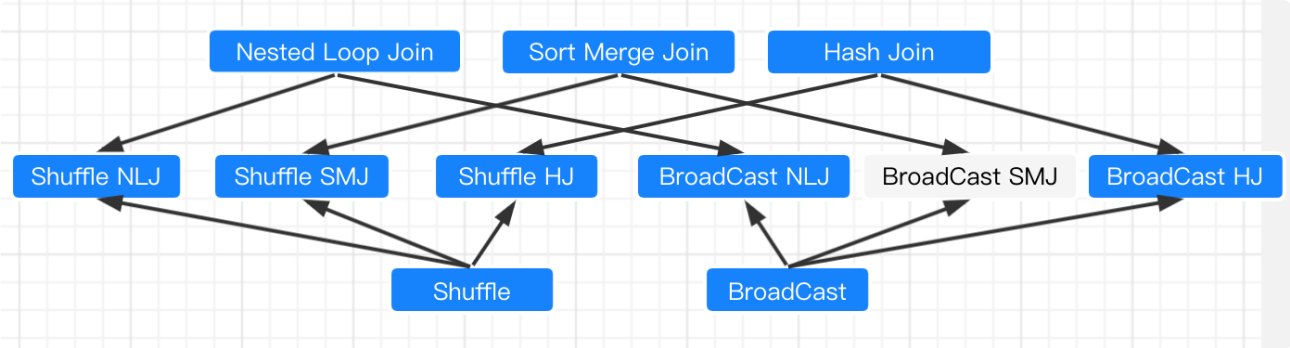
在分布式环境中，Spark支持两类数据分发模式:**Shuffle**和**Broadcast**。

因此，**从数据分发模式的角度出发**，数据关联可以分为**Shuffle Join**和**Broadcast Join**这两大类。

从实现机制来看，Join又可以分为**NLJ(Nested Loop Join)**、**SMJ(Sort Merge Join)和HJ(Hash Join)**。

> 上面的2种分发模式和3种实现机制的笛卡尔积，就构成了Spark支持的5种Join策略。(图中白色BroadCast SMJ不支持)。

如图所示：



这五种关联机制，Spark会怎么选择呢?

- 等值关联：Broadcast HJ > Shuffle SMJ > Shuffle HJ
- 不等值关联：Broadcast NLJ > Shuffle NLJ

我们重点看看关联机制

Nested Loop Join

在探讨关联机制的时候，我们又常常把左表称作是"驱动表"，而把右表称为"基表"。一般来说，驱动表的体量往往较大，在实现关联的过程中，驱动表是主动扫描数据的那一方。

Nested Loop Join会使用外、内两个嵌套的for循环，来依次扫描驱动表与基表中的数据记录。

假设驱动表有M行数据，而基表有N行数据，那么NLJ算法的计算复杂度是O(M * N)。尽管 NLJ 的实现方式简单、直观、易懂，但它的执行效率显然很差。

Sort Merge Join

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行hash join，但需要在join前将**数据排序**。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。

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，而是即用即取即丢弃，从而大大提升了大数据量下sql join的稳定性。

Sort Merge Join算法的计算复杂度为O(M + N)。

Hash Join

HJ 的设计初衷是以空间换时间，力图将基表扫描的计算复杂度降低至 O(1)。

HJ 的计算分为两个阶段，分别是 Build 阶段和 Probe 阶段。在 Build 阶段，在基表之上，算法使用既定的哈希函数构建哈希表。哈希表中的 Key 是 id 字段应用哈希函数之后的哈希值，而哈希表的Value同时包含了原始的Join Key和Payload。

在Probe阶段，算法依次遍历驱动表的每一条数据记录。首先使用同样的哈希函数，以动态的方式计算 Join Key 的哈希值。然后，算法再用哈希值去查询刚刚在 Build 阶段创建好的哈希表。如果查询失败，则说明该条记录与基表中的数据不存在关联关系；相反，如果查询成功，则继续对比两边的 Join Key。如果 Join Key 一致，就把两边的记录进行拼接并输出，从而完成数据关联。

SparkSQL的优化器

Catalyst优化器

Catalyst优化器，主要用来创建并优化执行计划，包含：创建语法树并生成执行计划、逻辑阶段优化和物理阶段优化。

Catalyst优化器的核心工作流程包括：

1. 解析SQL,并且生成AST(抽象语法树)
2. 把元数据信息（列的标识和类型）添加到AST(抽象语法树)中
3. 对已经加入元数据的AST,输入优化器,进行优化

这里的优化包括：

- 谓词下推 Predicate Pushdown, 将 Filter 这种可以减小数据集的操作下推, 放在 Scan（表） 的位置, 这样可以减少操作时候的数据量
- 列值裁剪 Column Pruning, 在谓词下推后,可以把表中没有用到的列裁剪掉, 这样可以减少处理的数据量, 从而优化处理速度

4. 由逻辑执行计划生成物理计划,从而生成RDD来运行

Tungsten

有一段时间，Tungsten被称为Spark有史以来最大的改动。其致力于提升Spark程序对内存和CPU的利用率，使性能达到硬件的极限，主要包含以下三个方面:

1. **Memory Management and Binary Processing**: off-heap管理内存，降低对象的开销和消除JVM GC带来的延时。
2. **Cache-aware computation**: 优化存储，提升CPU L1/ L2/L3缓存命中率。
3. **Code generation**: 优化Spark SQL的代码生成部分，提升CPU利用率。

Tungsten设计并实现了一种叫做**Unsafe Row**的二进制数据结构。**Unsafe Row本质上是字节数组，它以极其紧凑的格式来存储DataFrame的每一条数据记录，大幅削减存储开销，从而提升数据的存储与访问效率**。

与默认的Java Object相比，二进制的Unsafe Row以更加紧凑的方式来存储数据记录，大幅提升了数据的存储与访问效率。