**Spark难点解析：Join实现原理**

**Join背景介绍**

SQL的所有操作，可以分为简单操作（如过滤where、限制次数limit等）和聚合操作（groupBy，join等）。

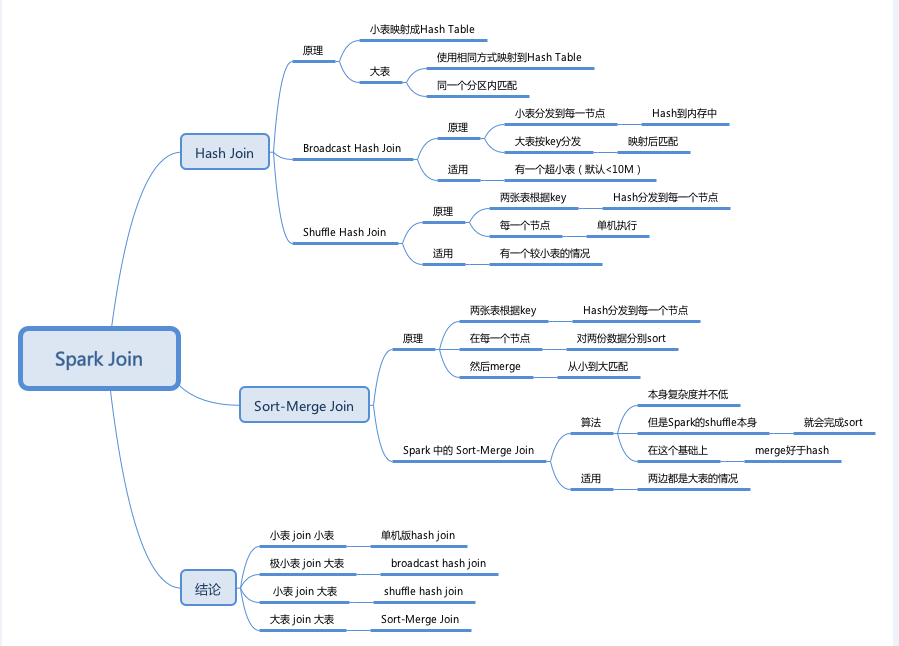
其中，join操作是最复杂、代价最大的操作类型，是大部分业务场景的性能瓶颈所在；所以，今天我们基于SparkSQL，来简要的聊一下SparkSQL所支持的几种常见的Join算法以及其适用场景。

首先，我们需要知道数仓中表格的分类：按照是否会经常涉及到Join操作，可以简单分为**低层次表和高层次表**。

低层次表：直接导入数仓的表，列数少，与其他表存在外键依赖，查询起来经常会用到大量Join算法，查询效率较低

高层次表：由低层次表加工而来，使用SQL将需要join的表预先合并，形成“宽表”。宽表上查询不需要大量Join，因此效率较高。但是，相对的是，宽表的数据存在大量冗余，同时生成滞后，查询不及时。

**Join使用的结论**



**Join常见分类&实现机制**

当前SparkSQL支持三种Join算法－**shuffle hash join、broadcast hash join以及sort merge join**。其中前两者归根到底都属于hash join，只不过在hash join之前需要先shuffle还是先broadcast。所以，首先我们来看一下内核hash join的机制。

**Hash Join**

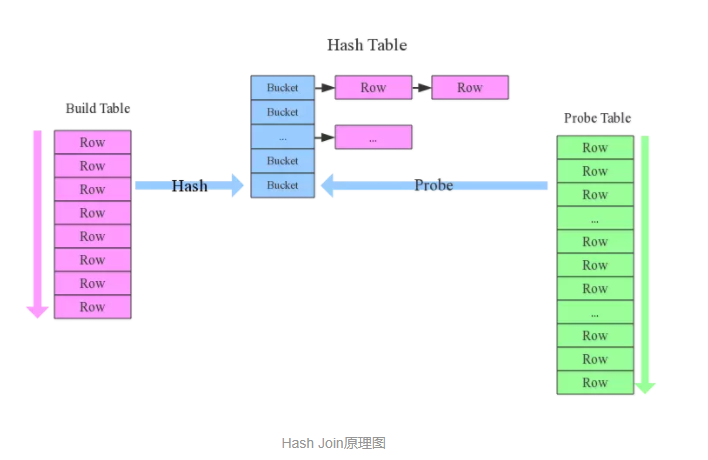
先来看一个简单的SQL：select \* from order,item where item.id = order.id

参与join的两张表是item和order，join key分别是item.id以及order.id，假设这个Join采用的是hash join算法，整个过程会经历三步：

1. 确定Build Table（映射表、小表）以及Probe Table（探查表、大表）。其中Build Table用于构建Hash Table，而Probe会遍历自身所有key，映射到所生成的Hash Table上去匹配。

2. Build Table构建Hash Table。依次读取Build Table（item）的数据，对于每一行数据根据join key（item.id）进行hash，hash到对应的Bucket，生成hash table中的一条记录。数据缓存在内存中，如果内存放不下需要dump到外存。

3. Probe Table探测。依次扫描Probe Table（order）的数据，使用相同的hash函数映射Hash Table中的记录，映射成功之后再检查join条件（item.id= order.i\_id），如果匹配成功就可以将两者join在一起。



两点补充：

1 hash join的性能。从上面的原理图可以看出，hash join对两张表基本只扫描一次，**算法效率是o(a+b)**，比起蛮力的笛卡尔积算法的a\*b快了很多数量级。

2 为什么说Build Table要尽量选择小表呢？从原理上也看到了，构建的Hash Table是需要被频繁访问的，所以**Hash Table最好能全部加载到内存**里，这也决定了hash join只适合至少一个小表join的场景。

看完了hash join的内核，我们来看一下这种单机的算法，在大数据分布式情况下，应该如何去做。目前成熟的有两套算法：broadcast hash join和shuffler hash join。

**Broadcast Hash Join**

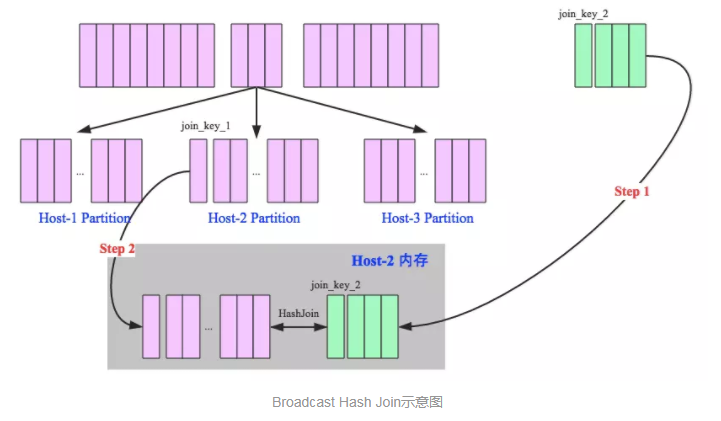
broadcast hash join是将其中一张小表广播分发到另一张大表所在的分区节点上，分别并发地与其上的分区记录进行hash join。broadcast适用于小表很小，可以直接广播的场景。

在执行上，主要可以分为以下两步：

1. broadcast阶段：将小表广播分发到大表所在的所有主机。分发方式可以有driver分发，或者采用p2p方式。

2. hash join阶段：在每个executor上执行单机版hash join，小表映射，大表试探；

需要注意的是，Spark中对于可以广播的小表，默认限制是10M以下。（参数是**spark.sql.autoBroadcastJoinThreshold**）



**Shuffle Hash Join**

当join的一张表很小的时候，使用broadcast hash join，无疑效率最高。但是随着小表逐渐变大，广播所需内存、带宽等资源必然就会太大，所以才会有默认10M的资源限制。

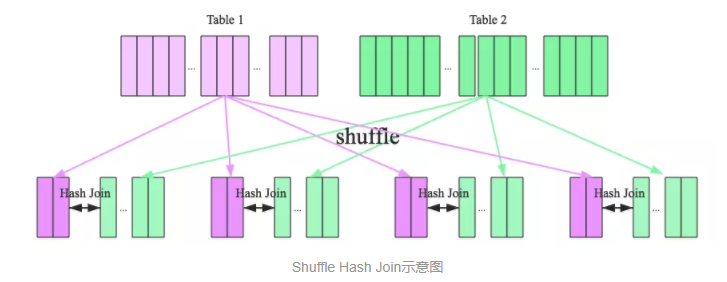
所以，当小表逐渐变大时，就需要采用另一种Hash Join来处理：Shuffle Hash Join。

Shuffle Hash Join按照join key进行分区，根据key相同必然分区相同的原理，将大表join分而治之，划分为小表的join，充分利用集群资源并行化执行。

在执行上，主要可以分为以下两步：

1. shuffle阶段：分别将两个表按照join key进行分区，将相同join key的记录重分布到同一节点，两张表的数据会被重分布到集群中所有节点。

2. hash join阶段：每个分区节点上的数据单独执行单机hash join算法。



刚才也说过，Hash Join适合至少有一个小表的情况，那如果两个大表需要Join呢？这时候就需要Sort-Merge Join了。

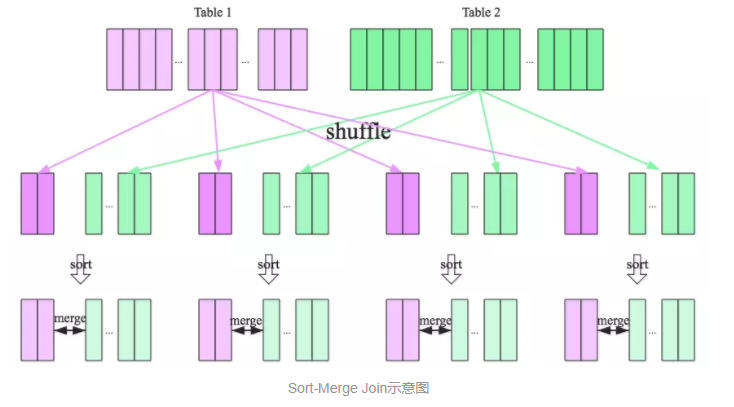
**Sort-Merge Join**

SparkSQL对两张大表join采用了全新的算法－sort-merge join，整个过程分为三个步骤：

1. shuffle阶段：将两张大表根据join key进行重新分区，两张表数据会分布到整个集群，以便分布式并行处理

2. sort阶段：对单个分区节点的两表数据，分别进行排序

3. merge阶段：对排好序的两张分区表数据执行join操作。join操作很简单，分别遍历两个有序序列，碰到相同join key就merge输出，否则继续取更小一边的key。

仔细分析的话会发现，sort-merge join的代价并不比shuffle hash join小，反而是多了很多。那为什么SparkSQL还会在两张大表的场景下选择使用sort-merge join算法呢？

这和Spark的shuffle实现有关，目前spark的shuffle实现都适用sort-based shuffle算法，因此在经过shuffle之后partition数据都是按照key排序的。因此理论上可以认为数据经过shuffle之后是不需要sort的，可以直接merge。

**结论：如何优化**

经过上文的分析，可以明确每种Join算法都有自己的适用场景。在优化的时候，除了要根据业务场景选择合适的join算法之外，还要注意以下几点：

1 数据仓库设计时最好避免大表与大表的join查询。

2 SparkSQL也可以根据内存资源、带宽资源适量将参数spark.sql.autoBroadcastJoinThreshold调大，让更多join实际执行为broadcast hash join。