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

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)！》**
- **《【Spark重点难点06】SparkSQL YYDS(中)！》**

本篇是Spark SQL的加餐篇，篇幅可能不是很长。希望大家喜欢。

Spark发展到今天，Spark SQL的方式已经是官方推荐的开发方式了。在今年的Spark 3.0大版本发布中，Spark SQL的优化占比将近50%；而像PySpark、Mllib 和 Streaming的优化占比都不超过10%，Graph的占比几乎可以忽略不计。

这代表着Spark SQL基本取代了Spark Core成为最新一代的引擎内核。

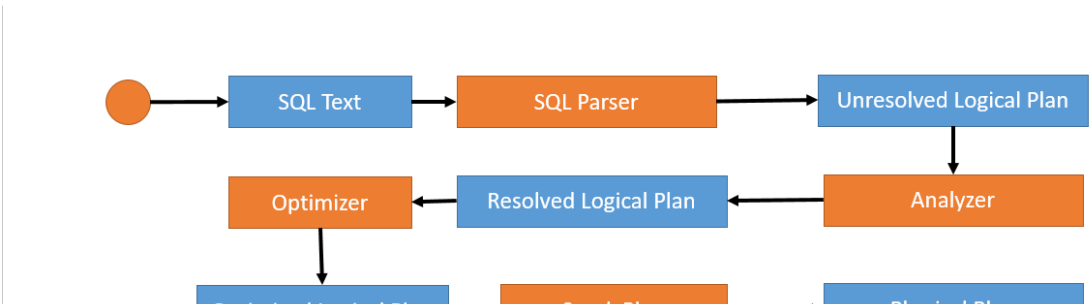
加上整个Spark社区转向了数据分析、算法方向，**我个人估计Spark Streaming和Structured Streaming可能在未来2-3年会慢慢退淡出我们的视野**。

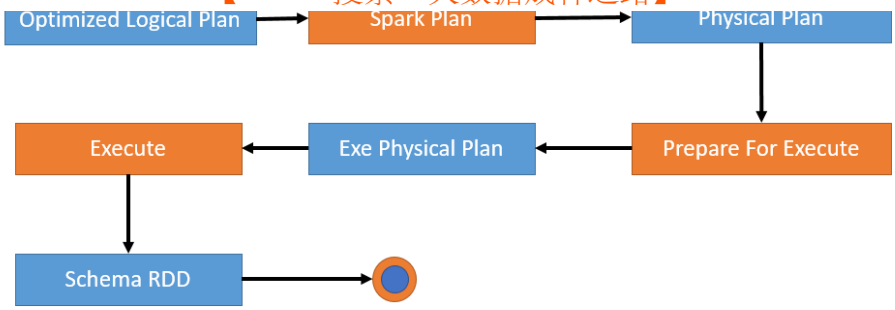
这也是我在这个系列中闭口不提Spark Streaming/Structured Streaming的主要原因。

这节课为加餐环节，我们详细的展开讲解一下Spark Catalyst优化和Tungsten优化。

Catalyst优化

我们在之前的课中已经讲解了SparkSQL Catalyst优化器的主要作用和核心工作流程。





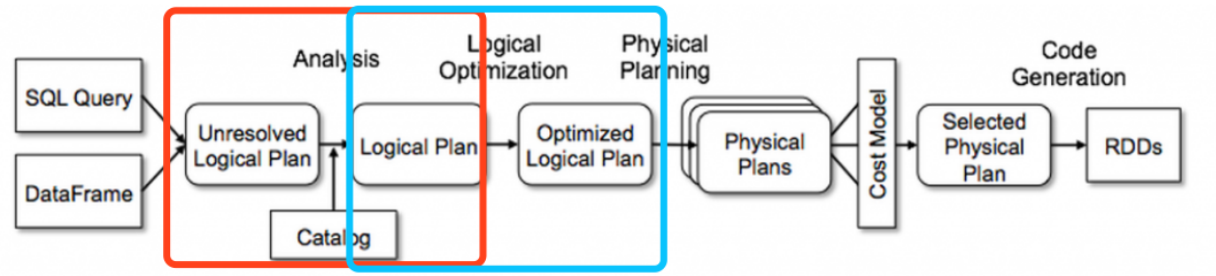
总体来说，Catalyst优化器包含**逻辑优化(Logical Planning)** 和**物理优化(Physical Planning)** 两个阶段。

Catalyst逻辑优化

在第一步的逻辑优化阶段，Catalyst又分为两个环节：

- 把"Unresolved Logical Plan"转换为"Analyzed Logical Plan"
- 基于启发式规则(Heuristics Based Rules)，把"Analyzed Logical Plan"转换为"Optimized Logical Plan"。

也就是图中的红框和蓝框部分。



你听起来是不是已经懵逼了。

我们用「人话」来解释一下这两个阶段。

第一个环节「逻辑计划解析环节」 也就是把 `「Unresolved Logical Plan」` 转换为 `「Analyzed Logical Plan」` ，大家注意 `「Unresolved」` 和 `「Analyzed」` 这两个单词， `「Unresolved」` 代表未被解析， `「Analyzed」` 表示被解析过。

那么从「未被解析」到「解析过」是怎么做的呢?

Catalyst就是要结合元数据(例如DataFrame中的Schema信息，Hive Metastore中的信息)的信息，通过一系列的Rules将数据进行解析。

第二个环节「逻辑计划优化」 也就是把 「Analyzed Logical Plan」 转换为 「Optimized Logical Plan」 的过程，同样 「Analyzed」 和 「Optimized」 两个单词已经能说明问题了。

在Spark3.0版本中，Catalyst总共有的优化规则(Rules)多达几十种。大概可以分为下面三类：

- 谓词下推(Predicate Pushdown)
- 列剪枝/剪裁(Column Pruning)
- 常量替换(Constant Folding)

谓词下推 是把过滤表达式(类似where age>18)下推到存储层直接过滤数据，减少传输到计算层的数据量。

列剪裁 就是只读取那些与查询相关的字段，减少数据读取的数量。

常量替换 就更简单了，Catalyst会自动用常量替换一些表达式。

Catalyst物理优化

物理优化从 「Optimized Logical Plan」 开始，分别经过 「Spark Plan」 最终生成 「Physical Plan」 。所以Catalyst物理优化同样可以分为两个两个阶段：生成 「Spark Plan」 和生成 「Physical Plan」 。

第一个阶段 ：Catalyst基于既定的优化策略(Strategies)，把逻辑计划中的关系操作符映射成物理操作符，生成Spark Plan。

其中的一个关键角色就是SparkPlanner，我们来看一下源码：

```
protected[sql] val planner = new SparkPlanner
//包含不同策略的策略来优化物理执行计划
protected[sql] class SparkPlanner extends SparkStrategies {
  val sparkContext: SparkContext = self.sparkContext
  val sqlContext: SQLContext = self
  def codegenEnabled: Boolean = self.conf.codegenEnabled
```

```
  def unsafeEnabled: Boolean = self.conf.unsafeEnabled
  def numPartitions: Int = self.conf.numShufflePartitions
  //把LogicPlan转换成实际的操作，具体操作类在org.apache.spark.sql.execution包下面
  def strategies: Seq[Strategy] =
    experimental.extraStrategies ++ (
    DataSourceStrategy ::
    DDLStrategy ::
    //把limit转换成TakeOrdered操作
    TakeOrdered ::
    //转换聚合操作
    HashAggregation ::
    //left semi join只显示连接条件成立的时候连接左边的表的信息
    // 比如select * from table1 left semi join table2 on(table1.student_no=table2.
    // 它只显示table1中student_no在表二当中的信息，它可以用来替换exist语句
    LeftSemiJoin ::
      //等值连接操作，有些优化的内容，如果表的大小小于spark.sql.autoBroadcastJoinThreshold
      //就自动转换为BroadcastHashJoin，即把表缓存，类似hive的map join（顺序是先判断右表再判
      //这个参数的默认值是10000
      //另外做内连接的时候还会判断左表右表的大小，shuffle取数据大表不动，从小表拉取数据过来计算
    HashJoin ::
    //在内存里面执行select语句进行过滤,会做缓存
    InMemoryScans ::
      //和parquet相关的操作
      ParquetOperations ::
      //基本的操作
      BasicOperators ::
      //没有条件的连接或者内连接做笛卡尔积
      CartesianProduct ::
      //把NestedLoop连接进行广播连接
      BroadcastNestedLoopJoin :: Nil)
  ……
}
```

第二个阶段：Catalyst基于事先定义的Preparation Rules，对Spark Plan 做进一步的完善、生成可执行的Physical Plan。

这里特别需要提醒的一点是：一个逻辑计划(Logical Plan)经过一系列的策略处理之后，可以得到多个物理计划(Physical Plans)。多个物理计划再经过代价模型(Cost Model)得到选择后的物理计划(Selected Physical Plan)。

从Spark Plan转换为Physical Plan，需要几组叫做Preparation Rules的规则：

```
//QueryExecution
  protected def prepareForExecution(plan: SparkPlan): SparkPlan = {
    preparations.foldLeft(plan) { case (sp, rule) => rule.apply(sp) }
  }
```

```
protected def preparations: Seq[Rule[SparkPlan]] = Seq(
  PlanSubqueries(sparkSession),//特殊子查询物理计划处理
  EnsureRequirements(sparkSession.sessionState.conf),//确保分区和排序正确
  CollapseCodegenStages(sparkSession.sessionState.conf),//代码生成相关
  ReuseExchange(sparkSession.sessionState.conf),//重用Exchange节点
  ReuseSubquery(sparkSession.sessionState.conf))//重用子查询
```

其中最重要的是： `EnsureRequirements` 。

`EnsureRequirements` 主要作用是确保分区和排序正确，也就是如果输入数据的分区或有序性无法满足当前节点的处理逻辑，则 `EnsureRequirements` 会在物理计划中添加一些Shuffle操作或排序操作来满足要求。

关于 `EnsureRequirements` 的源码实现，我们不做过多展开，大家有兴趣可以去读一读。

Tungsten优化

我们用一句话总结Tungsten对内核引擎的两方面改进： `数据结构设计` 和 `WSCG` 。

数据结构设计

首先，Tungsten设计了一种字节数组：Unsafe Row。这种数据结构能够显著的降低存储开销，仅用一个数组对象就能轻松完成一条数据的封装，大大降低了GC的压力。

其次，Tungsten还推出了基于内存页的内存管理，统一管理堆内与堆外内存。

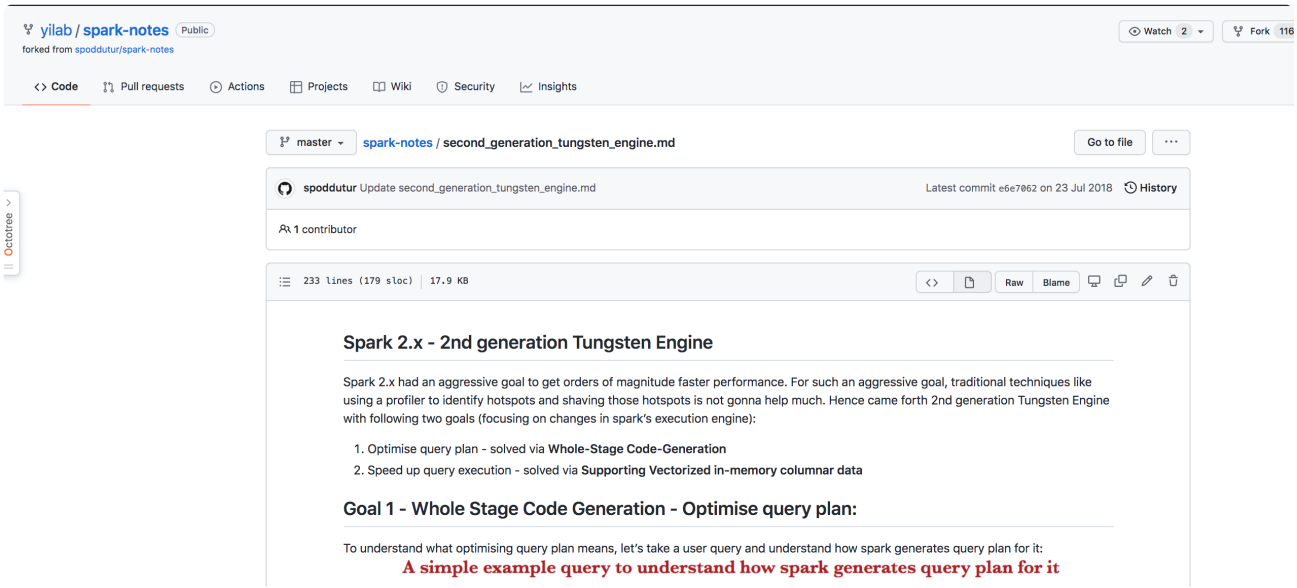
在堆内内存的管理上，基于Tungsten内存地址和内存页的设计机制，相比标准库，Tungsten实现的数据结构(如HashMap)使用连续空间来存储数据条目，连续内存访问有利于提升CPU缓存命中率，从而提升CPU工作效率。

由于内存页本质上是 Java Object，内存页管理机制往往能够大幅削减存储数据所需的对象数量，因此对GC非常友好的。

WSCG

WSCG是 `WholeStageCodeGeneration` 的英文简称，翻译过来就是：全周期代码生成。

我在Gihub找到了下面这份文档：



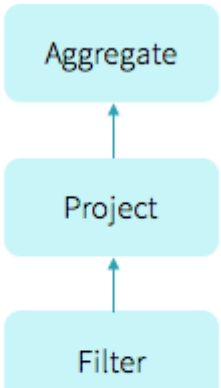
> https://github.com/yilab/spark-notes/blob/master/second_generation_tungsten_engine.md

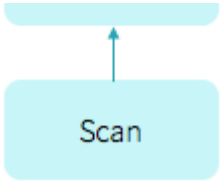
如果你的英文足够好，应该能看懂。

在Spark1.x时代，Spark SQL使用「火山迭代模型」。我们举个例子：

```
select count(*) from table where name='test';
```

要执行这个查询，Spark 1.x会使用一种最流行、最经典的查询求值策略，也就是： `Volcano Iterator Model` 。如下图：





在这种模型中，一个查询会包含多个operator，每个operator都会实现一个接口，提供一个next()方法，该方法返回operator tree中的下一个operator。

举例来说，上面那个查询中的filter operator的代码大致如下所示：

```
class Filter(child:Operator, predicate:(Row => Boolean))extends Operator{
  def next():Row ={
    var current = child.next()
    while(current == null || predicate(current)) {
      current = child.next()
    }
    return current
  }
}
```

让每一个operator都实现一个iterator接口，可以让查询引擎优雅的组装任意operator在一起。而不需要查询引擎去考虑每个operator具体的一些处理逻辑，比如数据类型等。

Vocano Iterator Model也因此成为了数据库SQL执行引擎领域内过去30年中最流行的一种标准。而且Spark SQL最初的SQL执行引擎也是基于这个思想来实现的。

对于上面的那个查询，如果我们通过代码来手工编写一段代码实现那个功能，代码大致如下所示：

```
def function() {
  var count = 0
  for(ss_item_sk in store_sales) {
    if(ss_item_sk == 1000) {
      count += 1
    }
  }
}
```

手写代码的性能比Volcano Iterator Model高了一整个数量级，而这其中的原因包含以下几点：

- 避免了virtual function dispatch：在Volcano Iterator Model中，至少需要调用一次next()函数来获取下一个operator。这些函数调用在操作系统层面，会被编译为virtual function dispatch。而手写代码中，没有任何的函数调用逻辑。虽然说，现代的编译器已经对虚函数调用进行了大量的优化，但是该操作还是会执行多个CPU指令，并且执行速度较慢，尤其是当需要成百上千次地执行虚函数调用时。
- 通过CPU Register存取中间数据，而不是内存缓冲：在Volcano Iterator Model中，每次一个operator将数据交给下一个operator，都需要将数据写入内存缓冲中。然而在手写代码中，JVM JIT编译器会将这些数据写入CPU Register。CPU从内存缓冲种读写数据的性能比直接从CPU Register中读写数据，要低了一个数量级。
- Loop Unrolling和SIMD：现代的编译器和CPU在编译和执行简单的for循环时，性能非常地高。编译器通常可以自动对for循环进行unrolling，并且还会生成SIMD指令以在每次CPU指令执行时处理多条数据。CPU也包含一些特性，比如pipelining，prefetching，指令reordering，可以让for循环的执行性能更高。然而这些优化特性都无法在复杂的函数调用场景中施展，比如Volcano Iterator Model。

如果要对Spark进行性能优化，一个思路就是在运行时动态生成代码，以避免使用Volcano模型，转而使用性能更高的代码方式。要实现上述目的，就引出了Spark第二代Tungsten引擎的新技术：`WholeStageCodeGeneration(WSCG)` 。

通过该技术，SQL语句编译后的operator-treee中，每个operator执行时就不是自己来执行逻辑了，而是通过whole-stage code generation技术，动态生成代码，生成的代码中会尽量将所有的操作打包到一个函数中，然后再执行动态生成的代码。

正如吴磊老师所说：WSCG生成的手写代码解决了操作符之间频繁的虚函数调用，以及操作符之间数据交换引入的内存随机访问。手写代码中的每一条指令都是明确的，可以顺序加载到 CPU 寄存器，源数据也可以顺序地加载到 CPU 的各级缓存中，因此，CPU 的缓存命中率和工作效率都会得到大幅提升。