SparkSQL 底层执行原理

本文档来自公众号：**五分钟学大数据**

微信扫码关注

目录

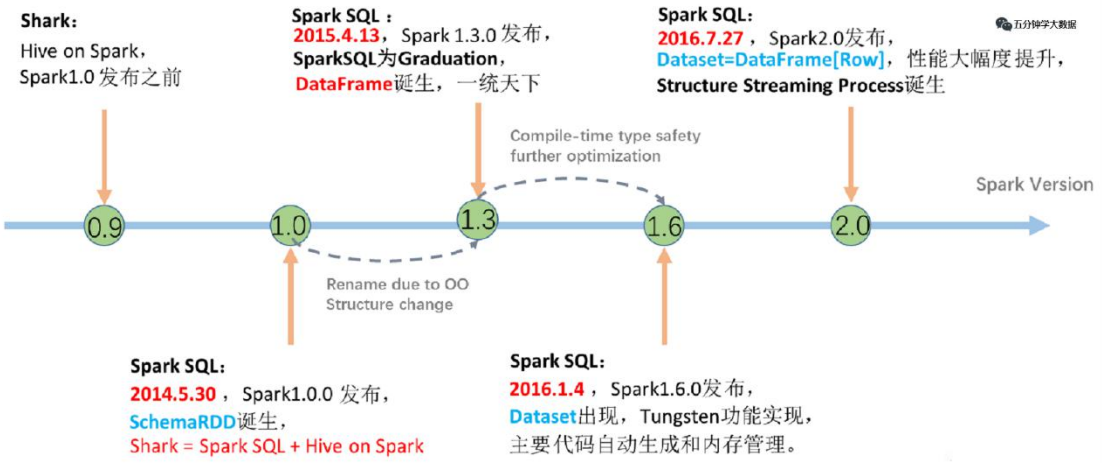
传送门：Hive SQL 底层执行过程详细剖析

一、Apache Spark

Apache Spark 是用于大规模数据处理的统一分析引擎，基于内存计算，提高了在大数据环境下数据处理的实时性，同时保证了高容错性和高可伸缩性，允许用户将 Spark 部署在大量硬件之上，形成集群。

Spark 源码从 1.x 的 40w 行发展到现在的超过 100w 行，有 1400 多位大牛贡献了代码。整个 Spark 框架源码是一个巨大的工程。

二、Spark SQL 发展历程



我们知道 Hive 实现了 SQL on Hadoop，简化了 MapReduce 任务，只需写 SQL 就能进行大规模数据处理，但是 Hive 也有致命缺点，因为底层使用 MapReduce 做计算，查询延迟较高。

1. Shark 的诞生

所以 Spark 在早期版本（1.0 之前）推出了 Shark，这是什么东西呢，Shark 与 Hive 实际上还是紧密关联的，Shark 底层很多东西还是依赖于 Hive，但是修改了内存管理、物理计划、执行三个模块，底层使用 Spark 的基于内存的计算模型，从而让性能比 Hive 提升了数倍到上百倍。

产生了问题：

1. 因为 Shark 执行计划的生成严重依赖 Hive，想要增加新的优化非常困难；

2. Hive 是进程级别的并行，Spark 是线程级别的并行，所以 Hive 中很多线程不安全的代码不适用于 Spark；
3. 由于以上问题，Shark 维护了 Hive 的一个分支，并且无法合并进主线，难以为继；
4. 在 2014 年 7 月 1 日的 Spark Summit 上，Databricks 宣布终止对 Shark 的开发，将重点放到 Spark SQL 上。

2. SparkSQL-DataFrame 诞生

解决问题：

1. Spark SQL 执行计划和优化交给优化器 Catalyst；
2. 内建了一套简单的 SQL 解析器，可以不使用 HQL；
3. 还引入和 DataFrame 这样的 DSL API，完全可以不依赖任何 Hive 的组件。

新的问题：

对于初期版本的 SparkSQL，依然有挺多问题，例如只能支持 SQL 的使用，不能很好的兼容命令式，入口不够统一等。

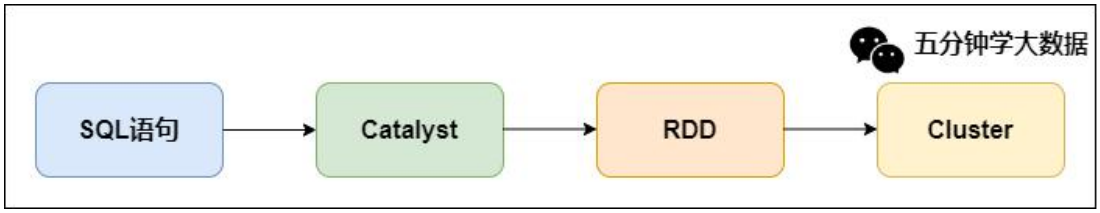
3. SparkSQL-Dataset 诞生

SparkSQL 在 1.6 时代，增加了一个新的 API，叫做 Dataset，Dataset 统一和结合了 SQL 的访问和命令式 API 的使用，这是一个划时代的进步。

在 Dataset 中可以轻易的做到使用 SQL 查询并且筛选数据，然后使用命令式 API 进行探索式分析。

三、Spark SQL 底层执行原理

Spark SQL 底层架构大致如下：



可以看到，我们写的 SQL 语句，经过一个优化器（Catalyst），转化为 RDD，交给集群执行。

SQL 到 RDD 中间经过了一个 Catalyst，它就是 Spark SQL 的核心，是针对 Spark SQL 语句执行过程中的查询优化框架，基于 Scala 函数式编程结构。

我们要了解 Spark SQL 的执行流程，那么理解 Catalyst 的工作流程是非常有必要的。

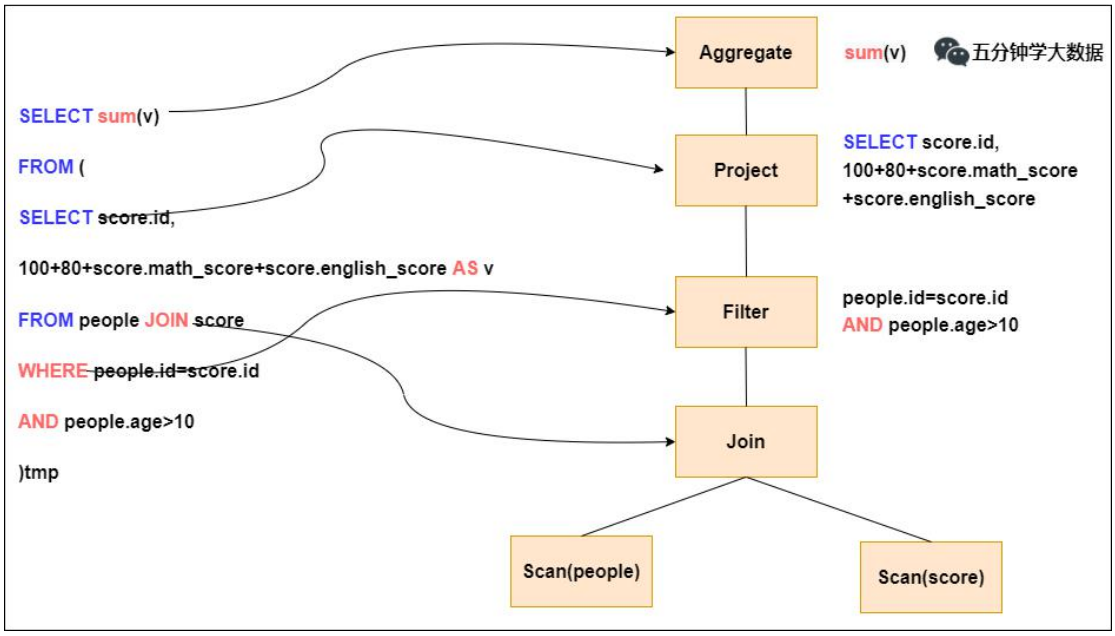
一条 SQL 语句生成执行引擎可识别的程序，就离不开**解析（Parser）、优化（Optimizer）、执行（Execution）** 这三大过程。而 Catalyst 优化器在执行**计划生成**和**优化**的工作时候，它离不开自己内部的五大组件，如下所示：

1. **Parser 模块**：将 SparkSql 字符串解析为一个抽象语法树/AST。
2. **Analyzer 模块**：该模块会遍历整个 AST，并对 AST 上的每个节点进行数据类型的绑定以及函数绑定，然后根据元数据信息 Catalog 对数据表中的字段进行解析。
3. **Optimizer 模块**：该模块是 Catalyst 的核心，主要分为 RBO 和 CBO 两种优化策略，其中 **RBO 是基于规则优化，CBO 是基于代价优化**。
4. **SparkPlanner 模块**：优化后的逻辑执行计划 OptimizedLogicalPlan 依然是逻辑的，并不能被 Spark 系统理解，此时需要将 OptimizedLogicalPlan 转换成 **physical plan（物理计划）** 。
5. **CostModel 模块**：主要根据过去的性能统计数据，选择最佳的物理执行计划。这个过程的优化就是 CBO（基于代价优化）。

为了更好的对整个过程进行理解，下面通过简单的实例进行解释。

步骤 1. Parser 阶段：未解析的逻辑计划

Parser 简单说就是将 SQL 字符串切分成一个一个的 Token，再根据一定语义规则解析成一颗语法树。Parser 模块目前都是使用第三方类库 **ANTLR** 进行实现的，包括我们熟悉的 Hive、Presto、SparkSQL 等都是由 **ANTLR** 实现的。

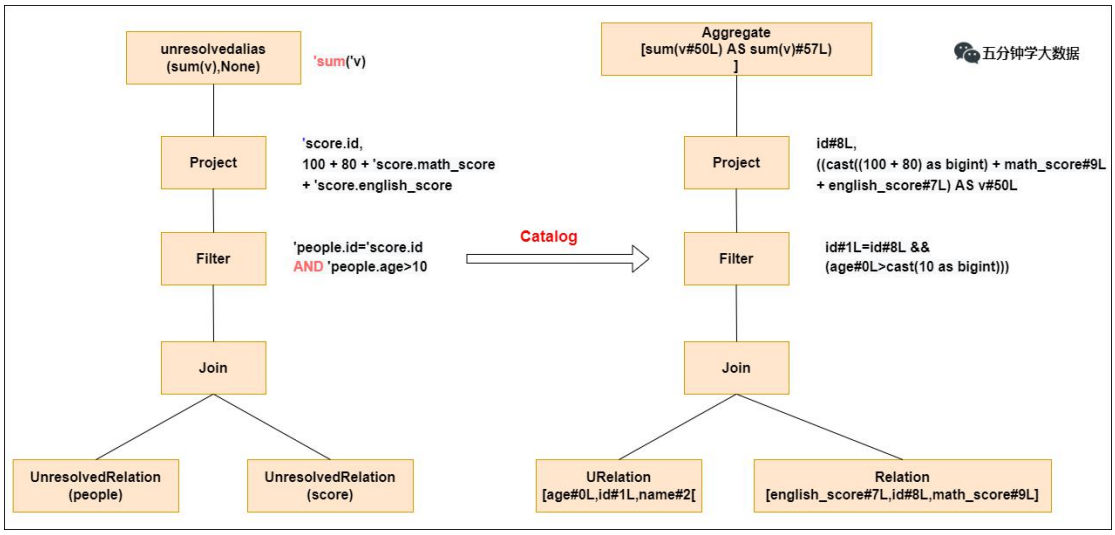


在这个过程中，会判断 SQL 语句是否符合规范，比如 select from where 等这些关键字是否写对。当然此阶段不会对表名，表字段进行检查。

步骤 2. Analyzer 阶段：解析后的逻辑计划

通过解析后的逻辑计划基本有了骨架，此时需要基本的元数据信息来表达这些词素，最重要的元数据信息主要包括两部分：表的 Scheme 和基本函数信息，表的 Scheme 主要包括表的基本定义（列名、数据类型）、表的数据格式（Json、Text）、表的物理位置等，基本函数主要指类信息。

Analyzer 会再次遍历整个语法树，对树上的每个节点进行数据类型绑定及函数绑定，比如 people 词素会根据元数据表信息解析为包含 age、id 以及 name 三列的表，people.age 会被解析为数据类型的 int 的变量，sum 被解析为特定的聚合函数。

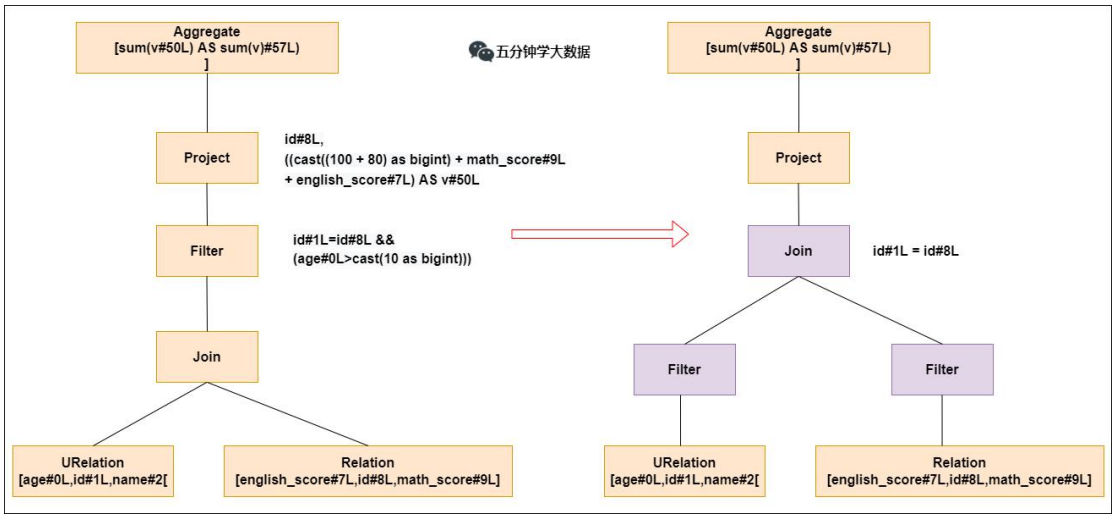


此过程就会判断 SQL 语句的表名，字段名是否真的在元数据库里存在。

步骤 3. Optimizer 模块：优化过的逻辑计划

Optimizer 优化模块是整个 Catalyst 的核心，上面提到优化器分为基于规则的优化（RBO）和基于代价优化（CBO）两种。基于规则的优化策略实际上就是对语法树进行一次遍历，模式匹配能够满足特定规则的节点，在进行相应的等价转换。下面介绍三种常见的规则：谓词下推(Predicate Pushdown) 、常量累加(Constant Folding) 、列值裁剪(Column Pruning) 。

- 谓词下推(Predicate Pushdown)

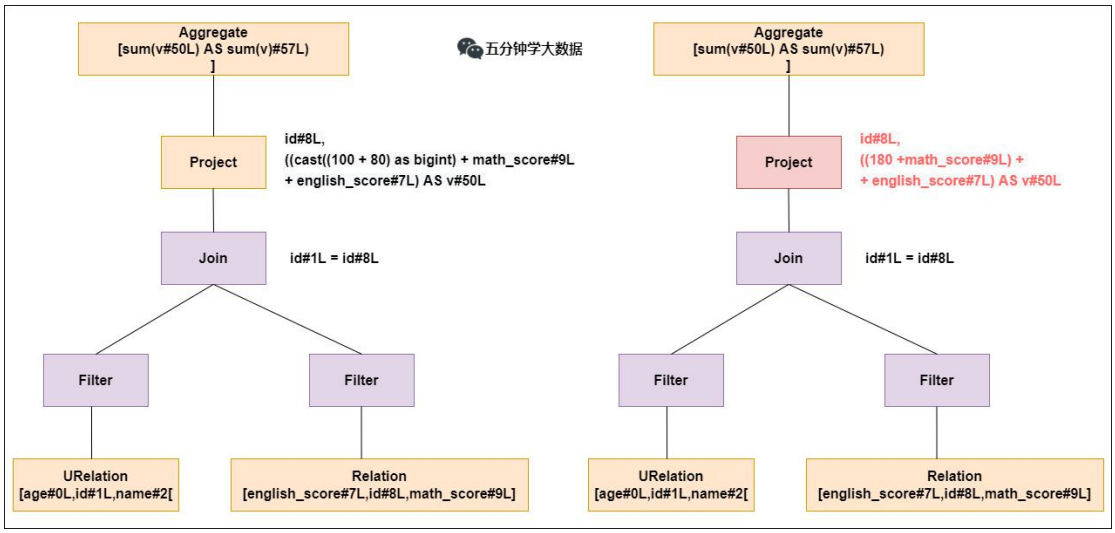


上图左边是经过解析后的语法树，语法树中两个表先做 join，之后在使用 age>10 进行 filter。join 算子是一个非常耗时的算子，耗时多少一般取决于参与 join

的两个表的大小，如果能够减少参与 join 两表的大小，就可以大大降低 join 算子所需的时间。

谓词下推就是将过滤操作下推到 join 之前进行，之后再进行 join 的时候，数据量将会得到显著的减少，join 耗时必然降低。

- 常量累加(Constant Folding)



常量累加就是比如计算 `x+(100+80)->x+180`，虽然是一个很小的改动，但是意义巨大。如果没有进行优化的话，每一条结果都需要执行一次 `100+80` 的操作，然后再与结果相加。优化后就不需要再次执行 `100+80` 操作。

- 列值裁剪(Column Pruning)

列值裁剪是当用到一个表时，不需要扫描它的所有列值，而是扫描只需要的 id，不需要的裁剪掉。这一优化一方面大幅度减少了网络、内存数据量消耗，另一方面对于列式存储数据库来说大大提高了扫描效率。

步骤 4. SparkPlanner 模块：转化为物理执行计划

根据上面的步骤，逻辑执行计划已经得到了比较完善的优化，然而，逻辑执行计划依然没办法真正执行，他们只是逻辑上可行，实际上 Spark 并不知道如何去执行这个东西。比如 join 是一个抽象概念，代表两个表根据相同的 id 进行合并，然而具体怎么实现合并，逻辑执行计划并没有说明。

此时就需要将逻辑执行计划转化为物理执行计划，也就是将逻辑上可行的执行计划变为 Spark 可以真正执行的计划。比如 join 算子，Spark 根据不同场景为该

算子制定了不同的算法策略，有 BroadcastHashJoin、ShuffleHashJoin 以及 SortMergejoin 等，物理执行计划实际上就是在这些具体实现中挑选一个耗时最小的算法实现，怎么挑选，下面简单说下：

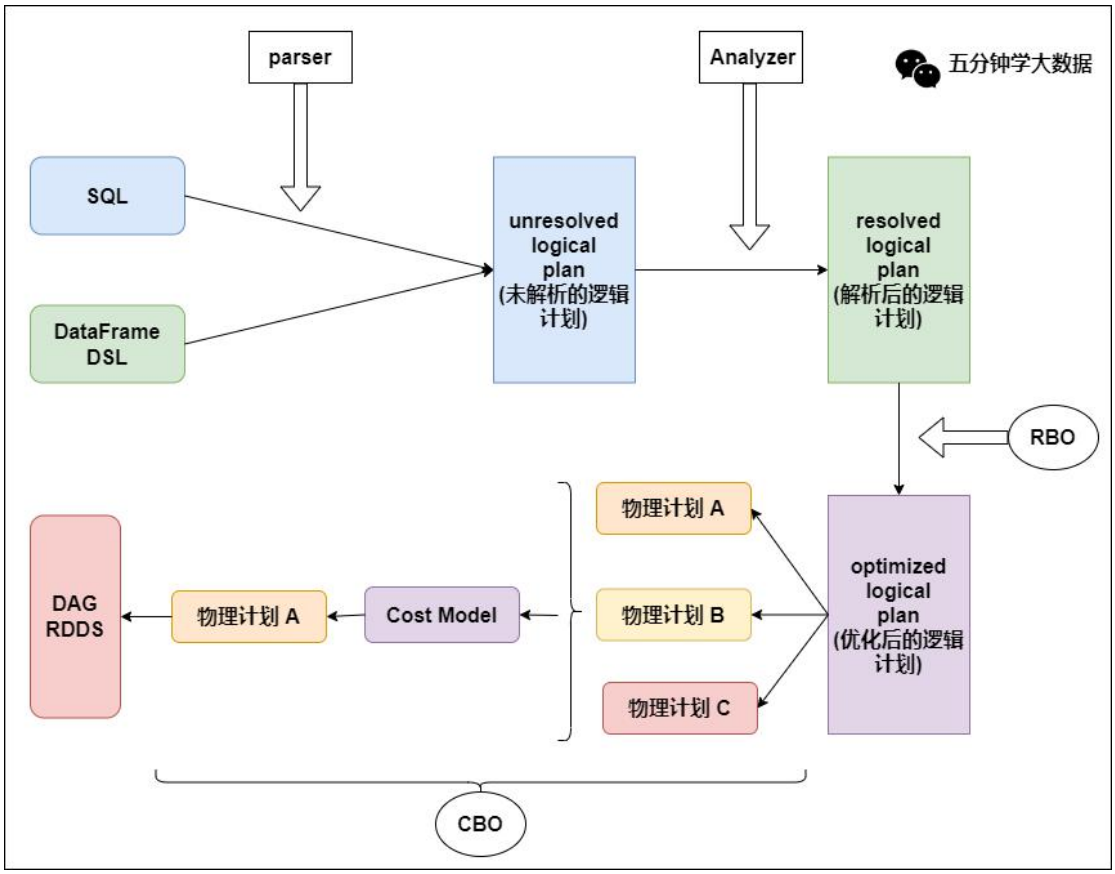
- 实际上 SparkPlanner 对优化后的逻辑计划进行转换，是生成了**多个可以执行的物理计划 Physical Plan**；
- 接着 **CBO（基于代价优化）**优化策略会根据 **Cost Model** 算出每个 Physical Plan 的代价，并选取代价最小的 Physical Plan 作为最终的 Physical Plan。

以上 2、3、4 步骤合起来，就是 Catalyst 优化器！

步骤 5. 执行物理计划

最后依据最优的物理执行计划，生成 java 字节码，将 SQL 转化为 DAG，以 RDD 形式进行操作。

总结：整体执行流程图



四、Catalyst 的两大优化

这里在总结下 Catalyst 优化器的两个重要的优化。

1. RBO：基于规则的优化

优化的点比如：谓词下推、列裁剪、常量累加等。

- **谓词下推案例：**

```
select
*
from
table1 a
join
table2 b
on a.id=b.id
where a.age>20 and b.cid=1
```

上面的语句会自动优化为如下所示：

```
select
*
from
(select * from table1 where age>20) a
join
(select * from table2 where cid=1) b
on a.id=b.id
```

就是在子查询阶段就提前将数据进行过滤，后期 join 的 shuffle 数据量就大大减少。

- **列裁剪案例：**

```
select
a.name, a.age, b.cid
from
(select * from table1 where age>20) a
join
(select * from table2 where cid=1) b
on a.id=b.id
```

上面的语句会自动优化为如下所示：

```
select
a.name, a.age, b.cid
from
(select name, age, id from table1 where age>20) a
join
(select id, cid from table2 where cid=1) b
on a.id=b.id
```

就是提前将需要的列查询出来，其他不需要的列裁剪掉。

- 常量累加：

```
select 1+1 as id from table1
```

上面的语句会自动优化为如下所示：

```
select 2 as id from table1
```

就是会提前将 1+1 计算成 2，再赋给 id 列的每行，不用每次都计算一次 1+1。

2. CBO：基于代价的优化

就是在 SparkPlanner 对优化后的逻辑计划生成了多个可以执行的物理计划 Physical Plan 之后，多个物理执行计划基于 Cost Model 选取最优的执行耗时最少的那个物理计划。
