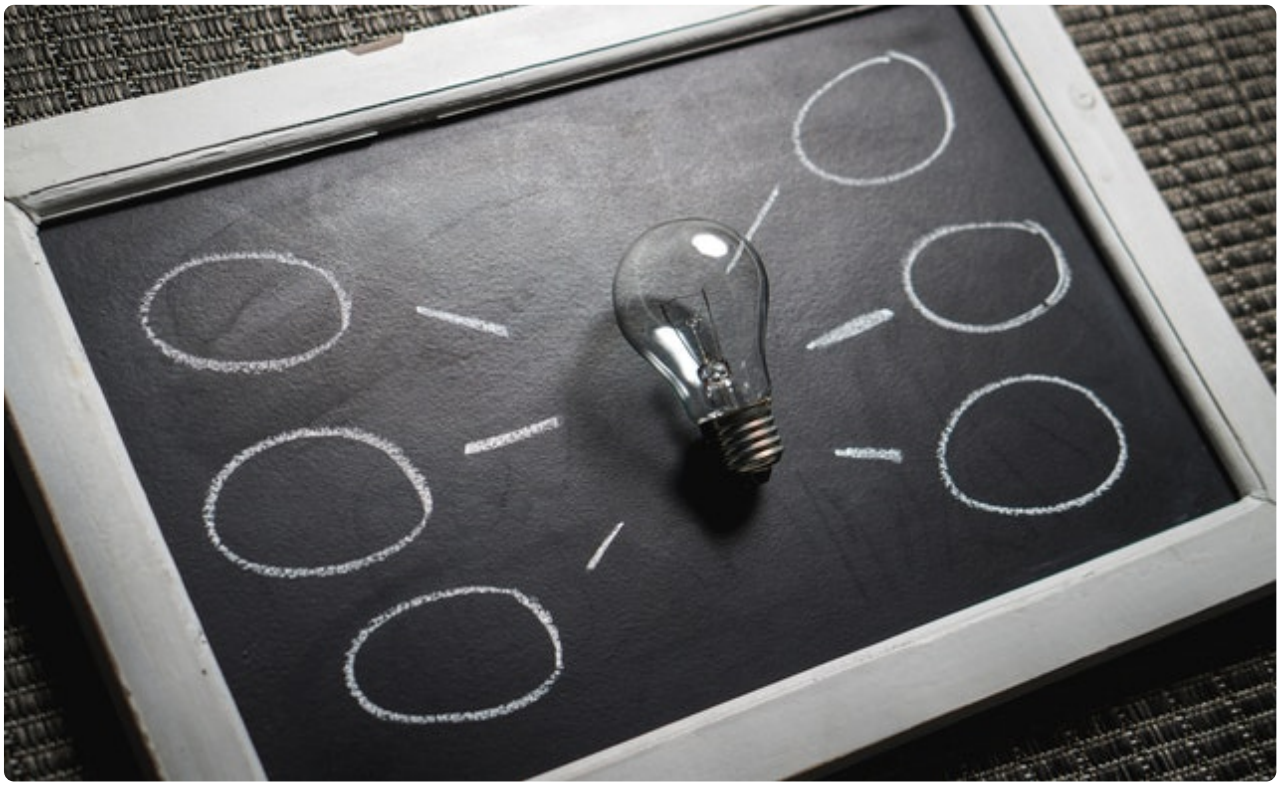
30 SQL 查询优化器的实现原理又是什么样的呢？

更新时间：2020-05-15 10:56:08



> 最聪明的人是最不愿浪费时间的人。——但丁

查询优化器是数据库中用于把关系表达式转换成执行计划的核心组件，很大程度上决定了一个系统的性能。但是，由于优化器的工作过程“隐藏”在 MySQL 系统中的底层，导致很多人对它会比较陌生。这一节里，我们首先去宽泛的看一看优化器的概述，之后再去理解它的两项核心工作，即逻辑与物理查询优化。

1. 查询优化器概述

之前在讲解 MySQL 的逻辑架构中曾经提到，**查询过程主要涉及三个组件：分析器、优化器和执行器**。可见，优化器位于逻辑架构的中部，同时，它也是整个 SQL 运行过程中的神经中枢。接下来，我们就去看一看优化器的定义以及分类。

1.1 查询优化器的定义

MySQL 查询优化器的主要功能是完成 SELECT 语句的执行（因为需要依赖优化器去生成执行计划），除了能够保证生成正确的执行计划之外，还有一个重要的功能，就是利用关系代数、代价模型等技术，提高 SQL 语句的执行效率。

根据定义，我们可以知道，查询优化器实际上负责两项工作。但是，由于生成执行计划的工作基本上是对语法树（分析器语法分析的输出）进行的逻辑读取，所以，通常就直接忽略了。同样，在接下来的内容中，我们也是只针对“优化”功能进行讲解。

1.2 查询优化器的分类

目前，有两种代价模型广泛的应用于查询优化器，那么，根据代价模型的不同，优化器可以分为两类：基于规则的优化器（Rule-Based Optimizer，RBO）和基于成本的优化器（Cost-Based Optimizer，CBO）。

1.2.1 基于规则的优化器

这一类优化器的核心是“规则”，即根据预先定义的优化规则对关系表达式（可以简单的理解为 SQL 语句）进行转换。转换所表达的语义是：一个关系表达式经过优化规则计算之后，会变成另外一个关系表达式。同时，替换原有的表达式，并再经过简单整理，生成最终的执行计划。

RBO 中最大的特色是“预先定义的优化规则”，它并不会考虑任何环境因素。即对于同样一条 SQL 语句，无论表中的数据如何变化，最后生成的执行计划都是一样的。但同时，这也是 RBO 的最大缺陷，因为往往数据量会与查询性能呈现负相关。另外，对于不同的 SQL 写法也有可能影响最终的执行计划，从而影响查询性能（规则不变，会导致优化过程死板）。

1.2.2 基于成本的优化器

这一类优化器同样会根据规则对关系表达式进行转换，之后生成另外一个关系表达式。但是，原有的表达式会被保留下来，且会经过一系列转换后生成多个执行计划。再之后，CBO 会根据“统计信息”和“代价模型”计算每个执行计划的“代价”，并从中挑选出代价最小的执行计划。所以，CBO 最核心的过程应该就是“代价”的计算，这会影响到最优执行计划选择的合理性。

通常，“统计信息”指的是花费 CPU 的成本；而“代价模型”则指的是 IO 的成本。所以，CBO 总的“代价”计算公式即为：

```
Total cost = CPU cost + IO cost
```

其中，CPU 花费的成本主要是处理返回记录（对结果进行过滤处理）的开销。所以，可以简单的以行记录数作为指标，且认为每处理 5 条记录，需要花费一个 cost。而 IO cost 主要是与索引相关的（从存储引擎中读取数据），索引覆盖性越好，“代价”自然也就越低。

从以上描述可以看出，CBO 是优于 RBO 的，原因也是很明显的：RBO 只认预先定义的规则，对环境并不敏感。而在实际的业务中，数据基本都是在不停的变化中，那么，通过 RBO 生成的执行计划大概率不是最优的。所以，目前各大数据库厂商和大数据计算引擎都更倾向于 CBO。

2. 逻辑查询优化

SQL 优化器的优化过程包含两个部分：逻辑优化和物理优化。这两项工作都要对语法分析树的形态进行修改，把语法树变为查询树。接下来，我们先去看一看逻辑优化的思路以及做法。

2.1 逻辑查询优化思路

找出 SQL 语句的等价变换形式，使得 SQL 执行的更高效，是查询优化器在逻辑优化阶段主要解决的问题。大多数情况下，我们编写的单条 SQL 语句不会很复杂（如果确实很复杂，需要考虑将单条拆分为多条），优化操作则会聚焦于 SQL 本身以及相关数据表的属性（索引和约束）。下面，我们来总结下逻辑优化可能的思路：

- **子句优化**：很多类型的子句都可能被优化，例如针对索引的顺序去优化 WHERE 子句中条件的顺序
- **语义优化**：根据 SQL 表达式的含义和数据表完整性约束进行语义上的优化，一个经典的例子是：使用主键或唯一键查询时，给 SQL 语句加上 LIMIT 约束
- **联接优化**：可以通过形式变化对表关联的语义进行优化，消除外连接和子查询（但是这种可能性比较低）

2.2 查询规则重写

传统的 OLTP（事务型操作）使用基于选择（SELECT）、投影（PROJECT）和连接（JOIN）3种基本操作相结合的查询，这种查询称为 SPJ 查询。优化器在查询优化的过程中，会对这3种基本操作进行优化。方式如下：

- **选择操作**：对应的是限制条件（可以简单的认为是 WHERE 子句后的条件），优化的方式是谓词下推，目的是减少连接操作前的 “选项”，使得中间的临时关系尽量少。以此来减少 IO 和 CPU 的消耗
- **投影操作**：投影指的是 SELECT 的列，优化方式是投影操作下推，减少连接操作前的列数。虽然这样并不能减少 IO，但是可以减少连接后中间关系的列数据大小
- **连接操作**：连接指的是两个或多个表的连接条件，这类查询操作通常比较复杂，优化器会尝试重写表的连接顺序，以获取更高的执行效率

逻辑查询优化将生成 “逻辑查询执行计划”，这里面会将语法树变为关系代数语法树的样式，原先 SQL 语义中的一些谓词变为逻辑代数的操作符样式。之后，查询优化器会进一步对查询树进行物理查询优化。

3. 物理查询优化

生成逻辑查询计划之后，随之会进行物理查询优化。物理优化会对逻辑查询进行改造，改造的内容主要是对连接的顺序进行调整，且最终目标是生成最终的物理查询执行计划（会作为执行器的输入）。

3.1 物理查询优化的工作内容

查询优化器在物理优化阶段，主要的工作内容有以下三点：

- 从可选的单表扫描方式中，挑选出最优的解
- 对于两张表的连接，选择最优的方式
- 对于多张表的连接，选择最优的连接顺序

可以看到，物理优化阶段的每一项工作都很复杂，且这里关于 “最优” 有一个定量的问题，即如何判断当前的选择是最优解 ？ 自此，也就引入了基于 “代价” 的查询优化（估算模型）方法。

3.2 代价估算模型

“代价模型” 是物理优化的重中之重，它所表达的语义是：给定一个 SQL，确定最优的执行计划，使得查询效率最高（需要注意，这是查询优化器认为的最高）。很明显，这其实就是 CBO 的核心思想。

除了之前对 CBO 代价计算的介绍之外，这里，对查询优化器 “代价模型” 计算进行总结（影响代价的方面）：

- 聚簇索引扫描代价为索引页面的总数量，效率最高，代价最小
- 覆盖索引扫描代价较小
- 非覆盖索引扫描代价较大
- 全表扫描代价巨大

另外，“代价” 的计算还需要统计信息的支持，例如：行记录数、索引长度、列值宽度等等。最终，针对于（相同的）SQL 的每一个执行计划，都可以计算出一个 “代价”，代价越小，效率越高。

3.3 索引选择优化

我们的数据表中可能定义了很多索引，且对于一个查询来说，可以使用的索引不止一个。那么，选择哪一个索引使得查询效率更高，也是优化器的一项重点工作。例如：

```
mysql> EXPLAIN SELECT * FROM worker WHERE city = '宿州市';
+----+-------------+--------+------------+------+-----------------------------------------+-----------------------+---------+-------+------+----------+-------+
| id | select_type | table  | partitions | type | possible_keys                           | key                   | key_len | ref   | rows | filtered | Extra |
+----+-------------+--------+------------+------+-----------------------------------------+-----------------------+---------+-------+------+----------+-------+
|  1 | SIMPLE      | worker | NULL       | ref  | city_nam_type_salary_idx,city_salary_idx | city_nam_type_salary_idx | 192     | const |    1 |   100.00 | NULL  |
+----+-------------+--------+------------+------+-----------------------------------------+-----------------------+---------+-------+------+----------+-------+
1 row in set, 1 warning (0.00 sec)
```

这里，EXPLAIN 命令的执行结果其实就是索引优化的过程和结果。其中，possible_keys 是可以被使用的索引，而 key 则是最终选择的索引。

3.4 表连接算法优化

表（两张表或多张表）连接会使得查询性能急剧下降，但是它又会时不时的出现在我们的业务场景中。MySQL 中定义了两种表连接算法：Nested-Loop Join 和 Block Nested-Loop join。

Nested-Loop Join 是 MySQL 基本的表连接算法，也就是嵌套循环算法。一个简单的嵌套循环连接（NLJ）会一次一个循环地从第一个表中读取行，将每一行传递给一个嵌套循环，之后该循环处理连接中的下一个表。官方文档对于 NLJ 给出了一个伪代码：

```
for each row in t1 matching range {
  for each row in t2 matching reference key {
    for each row in t3 {
      if row satisfies join conditions, send to client
    }
  }
}
```

t1、t2、t3 分别使用 range、ref 和 all 来进行连接，其中，t3 会扫描全表，之后依次去读取 t1 和 t2 上的数据，t3 则被称作是驱动表。查询优化器可以智能选择结果集最小的表作为驱动表，根据伪代码来看，结果集较小的驱动表确实可以使循环次数减少，达到优化的目的。

除了 NLJ 之外，MySQL 还提供了块循环算法（Block Nested-Loop join），它将外层循环的数据存在 join buffer 中，内层循环中的表会和 buffer 中的数据进行对比，从而减少循环次数。官方也给出了一个表达式，用于计算循环次数：

```
# S 表示 t1，t2 组合在缓存中的大小，C 是这些组合在 buffer 中的数量，所以，结果就是 t3 被扫描的次数
(S * C) / join_buffer_size + 1
```

可以看出，join_buffer_size 越大，扫描的次数越小。但是这个优化有上限，当 join_buffer_size 大到能够缓存所有之前的行组合，那么就是性能最好的时候，再增大这个值，也就没有优化效果了。

4. 总结

查询优化器技术的实现，基本上可以分为逻辑优化和物理优化两个阶段。但是，从以上的讲解可以看出，它们之间的界限并没有那么清晰。经过这两个阶段之后，执行器依据物理查询计划，逐步调用相关算法实现查询。

5. 问题

你理解查询优化器的优化思想吗？能用自己的语言简单表述出来吗？

除了查询优化器帮我们做的优化，你还能对 SQL 查询做哪些优化呢？

6. 参考资料

《高性能 MySQL（第三版）》

《数据库技术丛书·数据库查询优化器的艺术：原理解析与SQL性能优化》

MySQL 官方文档：Join Types Index

MySQL 官方文档：Optimization

}

29 你知道 SQL 分析器的实现原理是什么吗？

31 存储是本质，理解 InnoDB 存储引擎也就理所当然