# PG数据库查询优化深度探索---理论篇

## 概述

PG数据库是世界上最陷阱的开源关系数据库，是数据库从业人员研究数据库的宝贵财富，我们不打算再复述PG数据库的历史及概况，而是直入主题，看一下世界上最先进的开源数据库中的一个模块---查询优化器的实现方法。

## 查询优化的简介

查询优化是数据库管理系统中承上启下的一个模块，它接收来自语法分析模块传递过来的查询树(query = parse\_analyze(parsetree))， 在这个查询树的基础上进行了逻辑的等价变换、物理执行路径的筛选，并且把选择出的最优的执行路径传递给数据库的执行器模块。简而言之，一个查询优化器它的输入是查询树，输出是查询计划。

parsetree\_list = pg\_parse\_query(query\_string);  
 raw\_parsetree\_list = raw\_parser(query\_string); // 词法和语法分析入口  
 scanner\_init() // 词法分析  
 base\_yyparse() // 语法分析  
List \* pg\_analyze\_and\_rewrite(RawStmt \*parsetree)  
 query = parse\_analyze(parsetree); // 语义分析  
 querytree\_list = pg\_rewrite\_query(query); // 查询重写--逻辑优化 [input]  
 return querytree\_list；  
plantree\_list = pg\_plan\_queries(querytree\_list); // 查询优化--物理优化, 生成最优查询计划 [output]  
PortalDefineQuery(portal, plantree\_list); // 转换成物理执行路径--portal  
(void) PortalRun(portal, receiver); // 交给执行器执行, 并返回结果给receiver

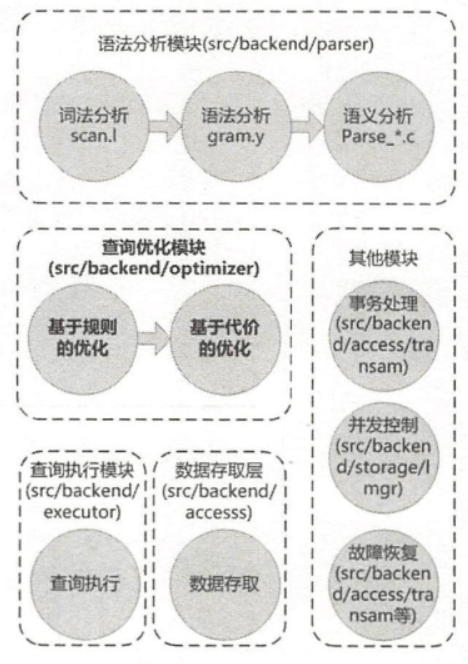


图 PG数据库的整体架构

数据库开发人员很难面面俱到地写出"极好的"语句,而查询优化器相对于数据库应用开发人员而言,具有一些独特的优势:

* 查询优化器和数据库用户之间的信息不对称,查询优化器在优化过程中会参考数据库统计模块自动产生的统计信息,这些统计信息从各个角度来描述数据的分布情况,查询优化器会综合考虑统计信息中的各种数据从而得到一个好的执行方案,而数据库用户一方面无法全面地了解数据的分布情况,另一方面即使数据库用户获得了所有的统计数据,人脑也很难构建一个精确的代价计算模型来对执行方案进行筛选.
* 查询优化器和数据库用户之间的时效性不同,数据库中的数据瞬息万变,一个在A时间点执行性能很高的执行计划,在B时间点由于数据内容发生了变化,它的性能可能就很低,查询优化器随时能根据数据的变化调整执行计划,而数据库用户则只能手动更改执行方案,和查询优化器相比,它的时效性比较低.
* 查询优化器和数据库用户之间的计算能力不同,目前计算机的计算能力已经大幅提高,在执行数值计算方面和人脑相比具有巨大的优势,查询优化器对一个语句进行优化时,可以从几百种执行方案中选出一个最优的方案,而人脑要全面地计算这几百种方案,需要的时间远远要长于计算机.

因此,查询优化器是提升查询效率非常重要的一个手段,虽然一些数据库也提供了人工干预生成查询计划的方法,但是通常而言查询优化器的优化过程对数据库开发人员是透明的,它自动地进行逻辑上的等价变换、自动地进行物理路径的筛选，极大地解放了数据库应用开发人员的“生产力”。

通常数据库的查询优化方法分为两个层次：

* 基于代价的查询优化（逻辑优化， Rule Based Optimization， 简称RBO）。
* 基于代价的查询优化（物理优化， Cost Based Optimization， 简称CBO）。

逻辑优化是建立在关系代数基础上的优化，关系代数中有一些等价的逻辑变换规则，通过对关系代数表达式进行逻辑上的等价变换，可能会获得执行性能比较好的等式，这样就能提高查询的性能；而物理优化则是建立在物理执行路径的过程中进行优化，关系代数中虽然指定了两个关系如何进行连接操作，但是这时的连接操作符属于逻辑运算符，它没有指定以何种方式实现这种逻辑连接操作，而查询执行器是不“认识”关系代数中的逻辑连接操作的，我们需要生成多个物理连接路径来实现关系代数中的逻辑连接操作，并且根据查询执行器的执行步骤，建立代价计算模型，通过计算所有的物理连接路径的代价，从中选择出“最优”的路径。

## 逻辑优化

逻辑优化时建立在关系代数基础之上的优化形式,下面通过介绍关系模型的理论知识来认识逻辑优化.

### 关系模型

关系数据库采用了关系模型来描述数据库，每个数据库是一个“关系”的集合，这个“关系”就是我们通常所谓的表,其形态类似于一个二维数组,我们称其中的一行为一个"N-元组",通常简称为"元组",其中的一列代表的就是一个"属性",所有属性的值最终组成了"域".

例如有如下的关系:

student(sno, sname, ssex);

course(cno, cname, tno);

score(sno, cno, degree);

teacher(tno, tname, tsex);

每个关系的实例如下图所示:

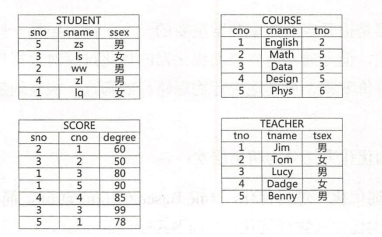


图 关系实体的数据

其中student代表了一个关系, 而sno、sname、ssex则代表的是这个关系的属性，student关系中共包含5个元组。

在关系模型中,为了对关系、元组、属性等进行操作，定义了两种形式化的语言，分别是关系代数和关系演算。关系代数从逻辑的角度定义了对数据进行操作的方法，而关系演算则是描述性的，它准确刻画需要获得的结果而不关心获得结果的过程。

关系代数的操作主要包含5个基本操作符，分别是选择（）、投影（）、笛卡尔积（）、并集（）、差集（），其中并集操作、差集操作、笛卡尔积操作来自集合论，选择和投影操作则是关系代数所特有的操作，这些基本操作是关系代数的基石，缺一不可，通过这些基本操作还可以衍生出一些其他比较重要的操作（可以用上述的5个基本操作将其表达出来），其中最重要的是交集操作（）和连接操作（）。如下图所示分别是投影、选择和笛卡尔积操作的示例。

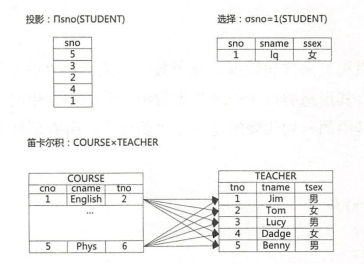


图 投影、选择和笛卡尔积的示例

另外，结合数据库的实际使用情况，通常数据库会对关系代数的逻辑运算符做一些扩展，例如外连接操作（左外连接、右外连接、全外连接、半连接、反半连接）、聚集和分组操作等，这些都不符合经典的关系代数理论，但是它们在数据库应用开发过程中却经常被用到，扩展操作的示例分别如下图所示。



图 扩展连接操作的示例

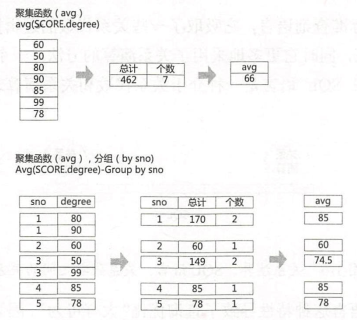


图 聚集和分组操作的示例

连接操作中左侧的表通常可以称为外表、左表或LHS（Left-Hand Side）表，右侧的表通常称为内表、右表或RHS（Right-Hand Side）表，而在外连接中还可以根据是否补NULL值进行区分，比如左连接，它的外表无须补NULL值，因此是Nonnullable-side的表，而左连接的内表需要补NULL值，因此是Nullable-side的表。

关系演算和关系代数不同，它从更高的层次来描述计算结果，完全不关心计算的过程，其中给予元组的关系演算可以称为元组关系演算，基于属性的关系演算称为域关系演算，元组关系演算包含的操作符如下：

* 存在量词（）和全称量词（）.
* 比较操作符（>, >=, <, <=, =, !=).
* 逻辑操作符 （ =>）

关系演算的表达能力和关系代数是等价的,我们用元组关系演算的方式来实现选择、投影等几个关系代数表达式，如下表所示：

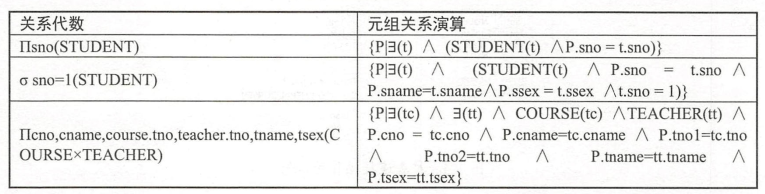


表 关系代数和关系演算的等价对照表

SQL作为数据库的标准查询语言，它吸收了一些关系代数的逻辑操作符，但是放弃了关系代数中“过程化”的特点，同时它更多地采用了关系演算的方法，一个SQL语句通常是对执行结果的描述，因此我们说SQL语言是一种介于关系代数和关系演算之间的描述性语言，如下图所示。

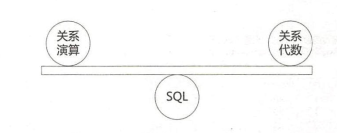


图 关系演算、SQL语言、关系代数之间的关系

SQL语言是描述性语言这种特性导致了查询优化"大有可为",因为它只规定了"WHAT",而没有规定"HOW",不同的获取结果的方法代价相差可能极大,因此数据库的查询优化就变得极为重要了.

既然逻辑优化是建立在关系代数等价变换基础上的优化,下面我们来总结一下关系代数有哪些等价变换的规则， 如下图所示：

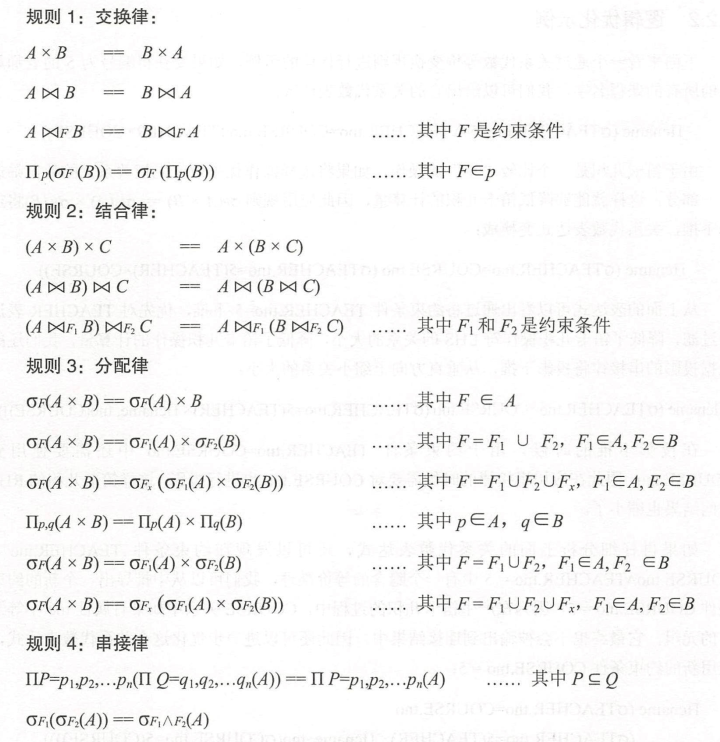


图 关系代数的等价表换规则

上面的规则并不能把所有的情况都列举出来，如果你有集合论和数理逻辑的基础，那么就能灵活地理解和运用这些规则，例如，如果对继续推导，那么就可以获得：



也就是说选择操作满足这样的交换律，集合论和数理逻辑都属于离散数学中的内容，因此感兴趣的读者可以参阅一些离散数学的相关资料。

### 逻辑优化示例

下面来看一个通过关系代数等价变换规则进行优化的示例，如果要获得编号为5的老师承担的所有的课程名字，我们可以给出它的关系代数表达式：



由于笛卡尔积是一个比较“重”的操作，如果将选择操作优先做，先把关系上的数据筛选掉一部分，这样就能够降低笛卡尔积的计算量，因此应用规则将选择下推，关系代数表达式变换成：

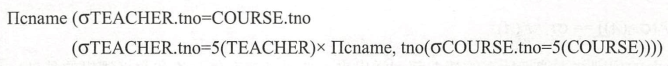


从上面的表达式可以看出通过将约束条件teacher.tno=5下推，优先对teacher表进行过滤，降低了笛卡尔积操作符LHS的关系的大小，降低了笛卡尔积操作的计算量，我们还能根据投影的串接律将投影下推，从垂直方向上缩小关系的大小：



在投影下推的时候，由于约束条件teacher.tno=course.tno中还需要使用到course.tno，因此在应用串接律的时候需要对course.tno也进行投影，这样笛卡尔积RHS中间结构也缩小了。

如果再仔细分析上面的关系代数表达式，还可以发现在约束条件teacher.tno=course.tno  teacher.tno=5中有一个隐含的等价推理,我们可以从中推导出一个新的约束条件course.tno=5，因为在笛卡尔积的过程中，course关系中如果有属性tno不等于5的元组，它最终也不会被输出到连接结果中因此还可以进一步优化这个关系代数表达式，应用新的约束条件course.tno=5：



这样笛卡尔积操作LHS和RHS的关系从水平方向和垂直方向都缩小了。这个关系表达式还可以进一步优化，因为现在经过选择产生的结果关系中tno的值一定是5，而经过选择产生的结果关系中tno的值也一定是5，也就是说经过选择操作下推之后，teacher.tno=course.tno已经隐含是一个永远为TRUE的约束条件，因此关系代数表达式可以变换成：



从下图1和图2中可以看出，应用关系代数的等价转换之后，关系代数表达式的计算量降低了。

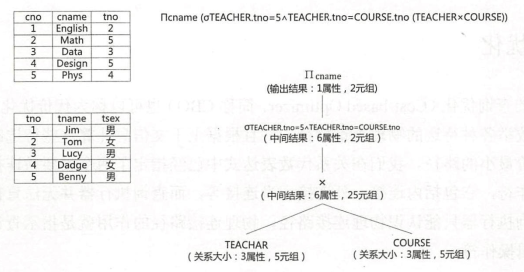


图1 关系代数实例，优化前的关系代数表达式

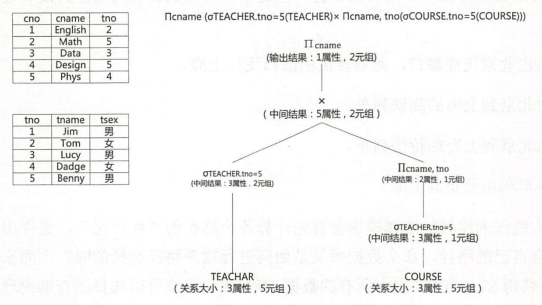


图2 关系代数实例，优化后的关系代数表达式

基于上述示例中的关系代数的等价变换，我们可以获得两个启发式的规则，它们分别从水平方向和垂直方向上尽早地缩小笛卡尔积的中间结果：

* 尽量将选择操作下推到下层节点来做。
* 尽量在叶子节点上使用投影缩小中间结果。

上面的示例看似完美是因为它总能将过滤条件一推到底,然而在数据库的实现过程中却要面临很多障碍,比如数据库除了基本的关系代数操作之外还扩展出了外连接、聚集操作，在内连接中能够进行下推的约束条件，如果换成外连接就不一定下推，基于内连接能够做的等价推理换成外连接也不一定等价，这极大地增加了数据库查询优化的难度。

## 物理优化

基于代价的查询优化（Cost-based Optimizer， 简称CBO）也可以称为代价优化、物理优化，其主要流程是枚举各种待选的物理查询路径，并且根据上下文信息计算这些待选择路径的代价，进而选择出代价最小的路径。我们在关系代数表达式中已经指定了两个表要做连接，这种连接操作是逻辑操作符，它包括内连接、外连接、外连接等，而查询执行器并无法直接执行这些逻辑操作符，查询执行器只能认识物理连接路径，物理连接路径的作用就是指示查询执行器以何种方式实现逻辑操作符。

生活中不乏类似物理优化的例子，正所谓条条大路通罗马。例如公司员工需要由北京去上海出差，去上海出差就好比是一个逻辑操作符，但是以何种方式去上海出差并没有设定，假如待选的路径如下：

A：乘飞机由北京飞往海口，然后转机由海口飞往上海。

B：乘坐由北京到上海的高铁列车。

C：乘坐由北京到上海的特快列车。

D：骑共享单车由北京到上海。

该员工的大脑作为路径的选择模块会首先计算各个路径的“性价比”，选择出“性价比”最高的、最适合自己的路径，那么数据库又是如何进行物理路径的选择呢？下面来看一个物理路径的例子，要获得student表中所有的数据，查询优化模块可以选择的查询物理路径如下。

A：扫描表的全部数据页面来获得所有的元组，并在所有元组上进行选择和投影。

B：如果表上有索引，并且约束条件满足索引的要求，可以尝试扫描索引，并在索引扫描产生的结果上做选择和投影。

C：如果有特殊的约束条件，还可以尝试位图扫描或TID扫描，并在扫描的结果上做选择和投影。

数据库无法从感性的角度来衡量哪条物理路径的代价低，因此它需要构建一个量化的模型，这个代价模型需要从两个方面来衡量路径的代价：

**执行代价 = IO代价 + CPU代价**

产生IO代价的原因是因为数据时保存在磁盘上,要对数据进行处理,需要将数据从磁盘加载到主存,另外在数据需要排序、建立hash表、物化的时候还可能将处理后的数据写入磁盘，这些都是IO代价，数据库要计算一个查询的IO代价存在一些困难：

* 磁盘IO到底是什么样的代价基准，由于磁盘种类不同，它的读写效率不同，如果有些数据挂载在机械磁盘上，而有些数据挂载在SSD磁盘上，那么不同磁盘上的数据的IO效率相差非常大，数据库如何区分这种区别呢？
* 数据库本身是有缓存系统的，假如某个表上有一些数据已经保存在缓存中了，这些数据在对表进行扫描的时候就不会产生IO，因此要想计算准确的IO代价，数据库还需要知道一个表中有多少页在缓存中，有多少页没有在缓存中，但是缓存中的页面可能随时地换入换出，数据库是否有能力实时地记录这种变化呢？
* 磁盘本身也有磁盘的缓存系统，在磁盘上随机读写和顺序读写的效率也不同，那么顺序读写和随机读写的效率差别如何量化呢？机械磁盘上顺序读写和随机读写的性能差别可能差距比较大，而SSD磁盘上的顺序读写和随机读写的性能差距则相对较小，数据库如何量化这种区别呢？

产生CPU代价的原因是选择、投影、连接都需要进行大量的运算，尤其是像聚集函数这样CPU密集型的操作符，而CPU代价的计算和IO代价一样也面临一些问题：

* 和IO代价类似，CPU也拥有很多型号，各种型号间的性能不同，是否需要量化一个CPU代价的基准单位？
* CPU为了加快执行速度，它有多级的cache，这些cache比数据库的缓存（内存）效率更高，在计算CPU代价的时候是否需要考虑这些cache带来的性能优化呢？
* 不同的表达式在执行的时候产生的代价不同，有些表达式的代价大，有些表达式的代价小，数据库如何量化这些表达式产生的代价呢？

要想计算物理路径的代价，数据库还需要对数据的分布情况有一个了解，因为无论是IO代价还是CPU代价，都是建立在对数据处理的基础之上的，数据的分布情况也会从很大程度上对代价产生影响：

* 相同的数据在不同的分布下锁带来的开销不同，例如数据在有序的情况下和无需的情况下，如果要执行一个排序的操作，那么可能就一个需要排序，而另一个不需要排序，开销肯定是不同的，再例如同样一份数据，它在磁盘上的存储是稀疏的还是紧凑的对IO代价的影响也非常大。
* 相同的数据在面临不同的选择操作时，它的开销也不同，比如选择操作要处理数据中的高频值，相对而言它需要的计算就多一些，因此代价也会高一些。

总之，数据库很难给出一个“准确”的代价模型来描述所有的情况，计算代价的目的是在物理路径之间进行挑选，它只需要能够用于比较物理路径的优劣就够了，虽然大部分数据库都采用了IO代价和CPU代价来衡量物理路径的代价，但具体的实现细节则千差万别，一个数据库的代价模型需要不断地“修炼”才能接近完美。

### 物理优化的4个“法宝”

关系的本身可以视为一个集合或者包，这拽数据结构对数据的分布没有设定，为了 提升计算的性能，我们需要借助一些数据结构或算法来对数据的分布做一些预处理，在物理优化的过程中有4个非常重要的数据结构或算法贯穿其中，下面简单介绍一下这些方法。

#### B+树

如果要查询一个表中的数据，最简单的办法自然是将表中的数据全部遍历一遍，但是随着当前数据量越来越大，遍历表中数据的代价也越来越高，而B+树就称为了我们高效地遍历数据的有力武器。

1970年，R.Bayer和E.mccreight提出了一种适用于外查找的树，它是一种平衡的多叉树，称为B树，B树就是在表的数据上建立一个“目录”，类似于书籍的目录，这样就能快速地定位到要查找的数据。

B树具有如下性质：

* 除了根节点之外，每个节点至少有m/2 (注: m为树的高度)个子节点，也就是说每个节点上最少半满，至多全满，多余全满会发生节点的分裂，少于半满会发生节点的合并，这种半满和全满的属性保证了在增加或者删除叶子结点的过程中，不会频繁地合并和分裂。
* 所有的叶节点都在同一层上，也就是说查找所有叶子节点的复杂度是相同的，都等于树高，由于B树是一颗多叉树，通常树的高度不会超过4，因此查找一个叶子节点的复杂度不高。
* 有k颗子树的分支节点则存在k-1个关键码，关键码按照递增次序进行排列，也就是说树中的在同一层上的节点是有序的，也就是说叶子节点也是有序的。

PG的数据库使用的是基于Lehman和Yao的论文进行改进的B+树，它在原来B树的基础上增加了“下一个节点的指针”和“页内最大值”，这样能提高B树的使用效率。

B+树作为一种数据结构和查询优化器本身没有直接的关系，但是数据库通常会建立基于B+树的索引，而在查询优化的过程中，索引扫描、位图扫描都会涉及这种B+树类型的索引，在索引扫描的过程中会根据B+树的情况计算代价，不同的B+树索引扫描的代价是不同的。例如，student表上有一个基于B+树的主键索引，因此通过sno=1这样的约束条件查询B+树，它的代价等于B+树的树高，而顺序扫描则需要遍历整个student表，因此B树对单值的查询、基于范围的约束条件通常都有比较好的查询效率。

#### Hash表

Hash表也是一种对数据进行预处理的方法，PG数据在多个地方使用了Hash表或借用了Hash表的思想，它在查询优化器中有如下的使用方式：

1. 借用Hash表可以实现分组操作，因为Hash表天然就有对数据分类的功能。
2. 借用Hash可以建立Hash索引，这种索引适用于等值的约束条件。
3. 物理连接路径中Hash Join是非常重要的一条路径，它对内表建立Hash表，外表的元组在Hash表中进行探测。

#### 排序

排序也是一种对数据进行预处理的方法，它主要用在如下几个方面。

1. 借用排序可以实现分组操作，因为经过排序之后，相同的数据都聚集在一起，因此它可以用来实现分组。
2. B树索引的建立需要借助排序来实现，PG数据采用堆的方式对数据进行存储，而B树索引的叶子结点是有序的，因此需要先将数据进行排序，而后在有序的数据上建立B树索引。
3. 物理连接路径MergeJoin路径需要借助排序实现，MergeJoin需要先对连接操作中的内表和外表进行排序，然后才能进行“归并”。
4. 数据库中的Order By操作需要借助排序实现，Order By本身的语义就是对数据进行排序。

在数据量比较小时，数据可以全部加载到内存，这时使用内排序就能完成排序的工作，而当数据量比较大时，则需要使用外排序才能完成排序的工作，因此在计算排序的代价时需要根据数据量的大小以及可使用的内存的大小来决定排序的代价。

#### 物化

物化就是将扫描操作或者连接操作的结果保存起来，这种保存是有代价的，因为如果扫描操作或者连接操作产生的中间结果比较大，就可能需要将中间结果写入外存，这会产生IO代价。

物化的优点是如果数据可以一次产生多次使用，那么就可以将这个中间结果保存下来多次利用，例如对student表和score表做连接，如果score表经过扫描之后，只有5%的数据作为中间结果，其他95%的数据都被过滤掉了，那么就可以考虑将这%5的数据物化起来，这样student表的每条元组只和这5%的数据进行连接就可以了。

中间结果是否物化主要取决于计算的模型，通常物理优化生成的物理路径时会比较物化和不物化两条路径的代价，最终选择代价较低的一个。

### 物理路径的生成过程

数据库中的物理路径大体上可以分为扫描路径和连接路径,扫描路径是针对单个关系的,他可以用来实现关系代数中的选择和投影,而连接路径对应的是两个表做连接操作,因此它可以用来实现关系代数中的笛卡尔积.

#### 物理路径的分类

如果要获得一个关系中的数据,最基础的方法就是将关系中的所有数据都遍历一遍,从中挑选出符合条件的数据,如下图所示,这种方式就是顺序扫描路径,顺序扫描路径的优点是具有广泛的实用性,各种"关系"都可以用这种方法,它的缺点是代价通常比较高,因为要把所有的数据都遍历一遍.



图 顺序扫描示意图

如果将数据做一些预处理,比如建立一个索引,如果要想获得一个表的数据,可以通过扫描索引获得所需数据的“地址”，然后通过地址将需要的数据获取出来，尤其是在选择操作带有约束的情况下，在索引和约束条件的共同作用下，关系中的有些数据就不用再遍历了，因为通过索引就很容易知道这些数据是不符合约束条件的，更有甚者，因为索引上也保存了数据，它的数据和关系中的数据是一致的，因此如果索引上的数据能满足要求，只需要扫描索引就可以获得所需数据了，也就是说在扫描路径中还可以有索引扫描路径和快速索引扫描路径两种方式，如下图所示：



图 索引扫描和快速索引扫描示意图

索引扫描路径带来一个问题，它可能会带来大量的随机读，因为索引中记录的是数据元组的地址，索引扫描通过扫描索引获得元组地址，然后通过元组地址访问数据，索引中保存的“有序”的地址，到数据中就可能是随机的了。为了解决这个问题，又增加了位图扫描，它通过位图将地址保存起来，把地址收集起来之后，让地址变得有序，这样就通过中间的位图把随机读消解掉了，如下图所示。

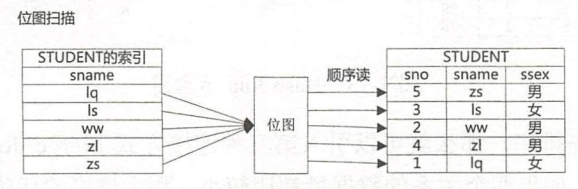


图 位图扫描示意图

当然，扫描的过程中还会结合一些特殊的情况使用一些非常高效的扫描路径，比如TID扫描路径，TID实际上是元组在磁盘上的存储地址，我们能够根据TID直接获得元组，这样查询的效率就非常高了。

扫描路径通常是执行计划中的叶子节点，也就是在最底层对表进行扫描的节点（也可能扫描节点的下层又是一个子执行计划），扫描路径就是为连接路径做准备的，扫描出来的数据就可以给连接路径来实现连接操作了。

要对两个关系做连接，受笛卡尔积的启发，可以用一个算法复杂度是O（m\*n)的方法来实现，我们叫它Nestlooped Join方法，这种方法虽然复杂度比较高，但是和顺序扫描一样，胜在具有普适性。如果Nestlooped Join的内表的路径是一个索引扫描路径，那么算法的复杂度就会降下来，索引扫描的算法复杂度是O（logn），因此如果Nestlooped Join的内表是一个索引扫描，它的整体复杂度就变成了O（mlogn），看上去这样也是可以接受的，如下图所示。

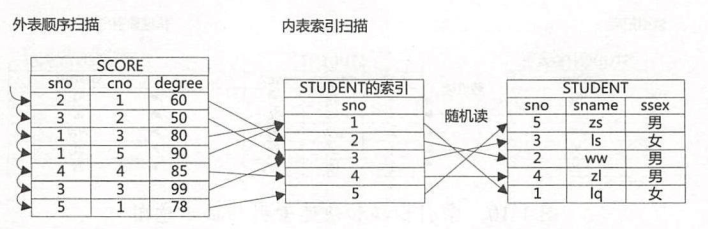


图 Nested Loop Join示意图

如果NestLooped Join的内表上没有索引，那么我们是否可以将内表的数据做一些处理，让算法的复杂度降低下来呢？答案是肯定的，在内表数据量不多的情况下，可以建立一个Hash表，如下图所示，这样由外表驱动在内表的Hash表上探测约束条件，假设Hash表有N个桶，内表数据均匀地分布在各个桶中，那么Hash Join的时间复杂度就是O（m\*n/N）。

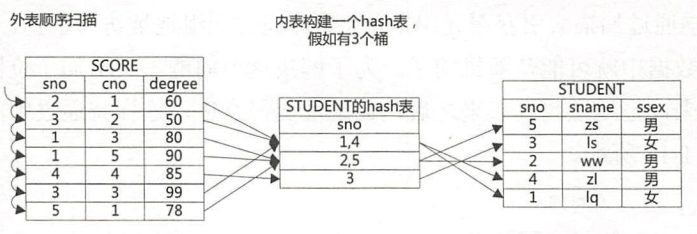


图 Hash Join示意图

如果将两个关系先排序，那么就可以引入第三中连接方式Merge Join，这种连接方式的代价主要浪费在排序上，如果两个关系的数据量都比较小，那么排序的代价是可控的，Merge Join就是适用的，另外如果关系上有有序的索引，那么就可以不用单独排序了，这样也比较适用于Merge Join， 如下图所示。

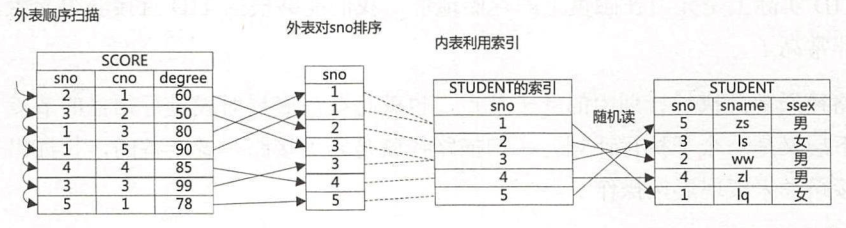


图 Merge Join示意图

综上所述，我们对物理路径有了基本的了解，屋里扫描路径主要有顺序扫描路径、（快速）索引扫描路径、位图扫描路径、TID扫描路径等，而物理连接路径主要有NestLooped Join、Hash Join和Merge Join，这些路径的生成都会或多或少地使用到物理优化的4个“法宝”，因此读者可以将它们结合在一起来理解物理优化路径生成的过程。

#### 路径搜索的方法

物理路径的种类越多，挑选最优路径的难度就越大，例如有3个表要做连接操作，每个表上有3个扫描路径，那么扫描路径的组合就有27种可能，由于表之间的连接顺序可以交换，3个表可能产生的连接顺序有12种情况，每个连接可能的物理连接路径是9种情况，那么最终要生成的3个表的连接路径“树”，共需要计算27x12x9=2916种情况。然后从中选出最优的那个路径“树”，如果再增加新的表参与连接操作，那么物理执行计划的解空间就会以几何级数的方式不断增长，因此物理路径的搜索方法就非常重要了，通常物理路径的搜索有以下几种方法：

1. 物理路径的搜索方法中最常用的是自底向上的一种方法，代表性的就是System-R系统所使用的模型---动态规划方法，这种方法把查找最优执行计划问题划分为子问题，用最优的子问题不断地向上迭代，最终获得最优解。
2. 还有自顶向下的方法，这种方法对逻辑优化和物理优化没有明显的界限，它先通过自顶向下的方式构造逻辑查询树以及物理查询树，和自底向上的方法不同，它不是通过子问题的最优解迭代出整个问题的最优解，而是通过先构建出逻辑的整个查询树，然后再迅速地枚举各种物理路径。
3. 随机搜索的方法也是一种重要的物理路径搜索方法，由于自底向上的方法和自顶向下的方法都需要遍历所有的解空间，因此在参与连接的表比较多的情况下，可以尝试采用随机的搜索算法，例如遗传算法，这种算法的优点是在有限的解空间取得最优的连接路径，它的效率是可控的，而缺点则是最优解是有限解空间的最优解，在整体空间上它可能只是一个局部的最优解。

PG数据库采用了其中的两种方法，一种是在表的数量比较少的情况下采用基于System-R系统的动态规划方法，另一种是在表的数量比较多的情况下启用遗传算法。对自顶向下的方法感兴趣的读者可以参考Pivotal公司开源的查询优化器ORCA的实现方法。

## 查询优化文件介绍

PG数据库的查询优化的代码在src/backend/optimizer目录下，其中有plan、prep、path、geqo、util共5个子目录，plan是总入口目录，它调用了prep目录进行逻辑优化，调用path、geqo目录进行物理优化，util目录是一些公共函数，供所有目录使用。

如下图所示，plan模块为总调用模块，prep和path被它调用。在执行中，从plan模块入口，先调用prep模块进行预处理，再调用path模块进行优化。path模块中有开关，指示是否启用遗传算法进行优化，如果启用，且连接的表超过11，就调用geqo目录中的遗传算法进行优化。util模块为辅助工具模块，提供其他模块使用的工具函数。

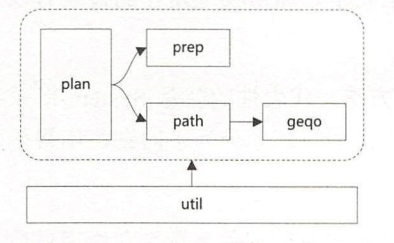


图 PG数据库查询优化文件目录结构图

prep目录主要处理逻辑优化中的逻辑重写的部分，对投影、选择条件、集合操作、连接操作都进行了重写，如下图所示。

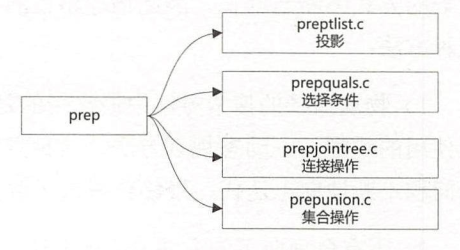


图 PG数据库查询优化prep目录结构图

path目录则主要是生成物理路径的部分，包括生成扫描路径、连接路径等，如下图所示。

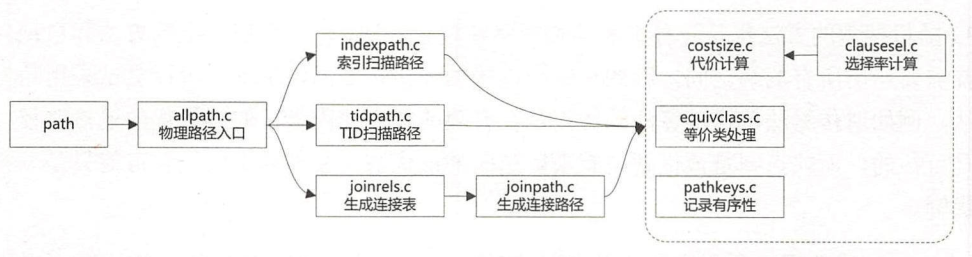


图 PG数据库查询优化path目录结构图

geqo目录主要是实现了一种物理路径的搜索算法---遗传算法，通过这种算法可以处理参与连接的表比较多的情况，utils目录则提供了大量的公共函数，其他各个目录中均可能会调用这些函数。

## 示例的约定

-- student(学号, 姓名, 性别)

create table student(sno int primary key, sname varchar(10), ssex int);

-- course(课程编号, 课程名, 教师编号)

create table course(cno int primary key, cname varchar(10), tno int);

-- score(学号, 课程编号, 分数)

create table score(sno int, cno int, degree int);

-- teacher(教师编号,教师姓名,教师性别)

create table teacher(tno int primary key, tname varchar(10), tsex int);

-- 在物理优化阶段则主要使用test\_a, test\_b, test\_c, test\_d....作为示例, 每个表的列名约定为 -- a, b, c, d....., 下面也给出它们的定义:

create table test\_a(a int, b int, c int, d int);

create table test\_b(a int, b int, c int, d int);

create table test\_c(a int, b int, c int, d int);

create table test\_d(a int, b int, c int, d int);

## 总结

关系数据库的查询优化通常分为逻辑优化和物理优化.逻辑优化是基于关系代数的等价的逻辑变换,关系代数中有大量的逻辑等价规则,可以利用这些规则尝试将选择操作和投影操作尽量下推,缩小查询中的中间结果以提高执行效率;物理优化则是通过代价估算的方式挑选代价比较低的物理路径,根据物理路径的性质又可以分成扫描路径和连接路径,在生成物理路径的过程中通常可以选择动态规划方法等最优路径算法来获得对物理路径进行搜索.

虽然本章中已经介绍了部分查询优化涉及的基础理论,但是这远远不够,有兴趣的读者可以参阅《数据库系统实现》、《数据库系统概念》等理论专著的相关章节来了解查询优化的基础理论,另外也可以查阅相关的论文来了解学术界对查询优化的不断改进.