第 9 章 关系查询处理和查询优化

本章介绍关系数据库的**查询处理**（query processing）和**查询优化**（query optimization）技术。首先介绍关系数据库管理系统的查询处理步骤，然后介绍查询优化技术。查询优化一般可分为**代数优化**（也称为**逻辑优化**）和**物理优化**（也称为**非代数优化**）。代数优化是指关系代数表达式的优化，物理优化则是指通过存取路径和底层操作算法的选择进行的优化。本章讲解实现查询操作的主要算法思想，目的是使读者初步了解关系数据库管理系统查询处理的基本步骤，及查询优化的概念、基本方法和技术，为数据库应用开发中利用查询优化技术提高查询效率和系统性能打下基础。

9.1 关系数据库系统的查询处理

查询处理是关系数据库管理系统执行查询语句的过程，其任务是把用户提交给关系数据库管理系统的查询语句转换为高效的**查询执行计划**。

9.1.1 查询处理步骤

关系数据库管理系统查询处理可以分为 4 个阶段：查询分析、查询检查、查询优化和查询执行，如图 9.1 所示。

1. 查询分析

首先对查询语句进行扫描、词法分析和语法分析。从查询语句中识别出语言符号，如 SQL 关键字、属性名和关系名等，进行语法检查和语法分析，即判断查询语句是否符合 SQL 语法规则。如果没有语法错误就转入下步处理，否则便报告语句中出现的语法错误。

2. 查询检查

对合法的查询语句进行语义检查，即根据数据字典中有关的模式定义检查语句中的数据库对象，如关系名、属性名是否存在和有效。如果是对视图的操作，则要用视图消解方法把对视图的操作转换成对基本表的操作。还要根据数据字典中的用户权限和完整性约束

定义对用户的存取权限进行检查。如果该用户没有相应的访问权限或违反了完整性约束，就拒绝执行该查询。当然，这时的完整性检查是初步的、静态的检查。检查通过后便把 SQL 查询语句转换成内部表示，即等价的**关系代数表达式**。这个过程中要把数据库对象的外部名称转换为内部表示。关系数据库管理系统一般都用**查询树**（query tree），也称为**语法分析树**（syntax tree）来表示扩展的关系代数表达式。

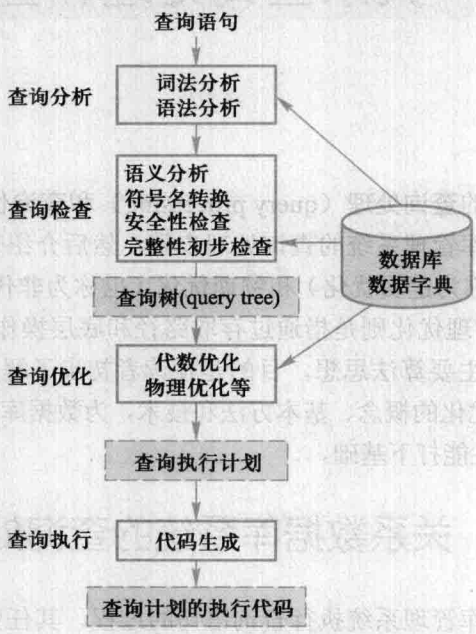

图 9.1 查询处理步骤

3. 查询优化

每个查询都会有许多可供选择的执行策略和操作算法，查询优化就是选择一个高效执行的查询处理策略。查询优化有多种方法。按照优化的层次一般可将查询优化分为代数优化和物理优化。代数优化是指关系代数表达式的优化，即按照一定的规则，通过对关系代数表达式进行等价变换，改变代数表达式中操作的次序和组合，使查询执行更高效；物理优化则是指存取路径和底层操作算法的选择。选择的依据可以是基于规则（rule based）的，也可以是基于代价（cost based）的，还可以是基于语义（semantic based）的。

实际关系数据库管理系统中的查询优化器都综合运用了这些优化技术，以获得最好的查询优化效果。

4. 查询执行

依据优化器得到的执行策略生成查询执行计划，由**代码生成器**（code generator）生成执行这个查询计划的代码，然后加以执行，回送查询结果。

9.1.2 实现查询操作的算法示例

本节简单介绍选择操作和连接操作的实现算法，确切地说是算法思想。每一种操作有多种执行的算法，这里仅仅介绍最主要的几个算法，对于其他重要操作的详细实现算法，有兴趣的读者请参考有关关系数据库管理系统实现技术的书。

1. 选择操作的实现

第3章中已经介绍了SELECT语句的强大功能，SELECT语句有许多选项，因此实现的算法和优化策略也很复杂。不失一般性，下面以简单的选择操作为例介绍典型的实现方法。

［例9.1］ SELECT * FROM Student WHERE <条件表达式>;

考虑<条件表达式>的几种情况：

C1: 无条件;

C2: Sno='201215121';

C3: Sage>20;

C4: Sdept='CS' AND Sage>20;

选择操作只涉及一个关系，一般采用全表扫描或者基于索引的算法。

（1）简单的全表扫描算法（table scan）

假设可以使用的内存为M块，全表扫描的算法思想如下：

① 按照物理次序读Student的M块到内存。

② 检查内存的每个元组t，如果t满足选择条件，则输出t。

③ 如果Student还有其他块未被处理，重复①和②。

全表扫描算法只需要很少的内存（最少为1块）就可以运行，而且控制简单。对于规模小的表，这种算法简单有效。对于规模大的表进行顺序扫描，当选择率（即满足条件的元组数占全表的比例）较低时，这个算法效率很低。

（2）索引扫描算法（index scan）

如果选择条件中的属性上有索引（例如B+树索引或hash索引），可以用索引扫描方法，通过索引先找到满足条件的元组指针，再通过元组指针在查询的基本表中找到元组。

［例9.1-C2］ 以C2为例：Sno='201215121'，并且Sno上有索引，则可以使用索引得到Sno为'201215121'元组的指针，然后通过元组指针在Student表中检索到该学生。

［例9.1-C3］ 以C3为例：Sage>20，并且Sage上有B+树索引，则可以使用B+树索引找到Sage=20的索引项，以此为入口点在B+树的顺序集上得到Sage>20的所有元组指针，然后通过这些元组指针到Student表中检索到所有年龄大于20的学生。

［例9.1-C4］ 以C4为例：Sdept='CS' AND Sage>20，如果Sdept和Sage上都有索引，一种算法是，分别用上面两种方法找到Sdept='CS'的一组元组指针和Sage>20的另一组元

组指针，求这两组指针的交集，再到 Student 表中检索，就得到计算机系年龄大于 20 岁的学生。

另一种算法是，找到 Sdept='CS'的一组元组指针，通过这些元组指针到 Student 表中检索，并对得到的元组检查另一些选择条件（如 Sage>20）是否满足，把满足条件的元组作为结果输出。

一般情况下，当选择率较低时，基于索引的选择算法要优于全表扫描算法。但在某些情况下，例如选择率较高，或者要查找的元组均匀地分布在查找的表中，这时基于索引的选择算法的性能不如全表扫描算法。因为除了对表的扫描操作，还要加上对 *B*+树索引的扫描操作，对每一个检索码，从 *B*+树根结点到叶子结点路径上的每个结点都要执行一次 I/O 操作。

2．连接操作的实现

连接操作是查询处理中最常用也是最耗时的操作之一。人们对它进行了深入的研究，提出了一系列的算法。不失一般性，这里通过例子简单介绍等值连接（或自然连接）最常用的几种算法思想。

［**例 9.2**］ SELECT * FROM Student，SC WHERE Student.Sno=SC.Sno;

（1）嵌套循环算法（nested loop join）

这是最简单可行的算法。对外层循环（Student 表）的每一个元组，检索内层循环（SC 表）中的每一个元组，并检查这两个元组在连接属性（Sno）上是否相等。如果满足连接条件，则串接后作为结果输出，直到外层循环表中的元组处理完为止。这里讲的是算法思想，在实际实现中数据存取是按照数据块读入内存，而不是按照元组进行 I/O 的。嵌套循环算法是最简单最通用的连接算法，可以处理包括非等值连接在内的各种连接操作。

（2）排序-合并算法（sort-merge join 或 merge join）

这是等值连接常用的算法，尤其适合参与连接的诸表已经排好序的情况。

用排序-合并连接算法的步骤是：

① 如果参与连接的表没有排好序，首先对 Student 表和 SC 表按连接属性 Sno 排序。

② 取 Student 表中第一个 Sno，依次扫描 SC 表中具有相同 Sno 的元组，把它们连接起来（如图 9.2 所示）。

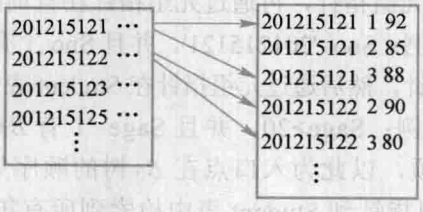

图 9.2 排序-合并连接算法示意图

③ 当扫描到 Sno 不相同的第一个 SC 元组时，返回 Student 表扫描它的下一个元组，再扫描 SC 表中具有相同 Sno 的元组，把它们连接起来。

重复上述步骤直到 Student 表扫描完。

这样 Student 表和 SC 表都只要扫描一遍即可。当然，如果两个表原来无序，执行时间要加上对两个表的排序时间。一般来说，对于大表，先排序后使用排序-合并连接算法执行连接，总的时间一般仍会减少。

（3）索引连接（index join）算法

用索引连接算法的步骤是：

① 在 SC 表上已经建立了属性 Sno 的索引。

② 对 Student 中每一个元组，由 Sno 值通过 SC 的索引查找相应的 SC 元组。

③ 把这些 SC 元组和 Student 元组连接起来。

循环执行②③，直到 Student 表中的元组处理完为止。

（4）hash join 算法

hash join 算法也是处理等值连接的算法。它把连接属性作为 hash 码，用同一个 hash 函数把 Student 表和 SC 表中的元组散列到 hash 表中。第一步，划分阶段（building phase），也称为创建阶段，即创建 hash 表。对包含较少元组的表（如 Student 表）进行一遍处理，把它的元组按 hash 函数(hash 码是连接属性)分散到 hash 表的桶中；第二步，试探阶段(probing phase)，也称为连接阶段（join phase），对另一个表（SC 表）进行一遍处理，把 SC 表的元组也按同一个 hash 函数（hash 码是连接属性）进行散列，找到适当的 hash 桶，并把 SC 元组与桶中来自 Student 表并与之相匹配的元组连接起来。

上面的 hash join 算法假设两个表中较小的表在第一阶段后可以完全放入内存的 hash 桶中。不需要这个前提条件的 hash join 算法以及许多改进的算法请参考本章文献[16]。以上的算法思想可以推广到更加一般的多个表的连接算法上。

9.2 关系数据库系统的查询优化

查询优化在关系数据库系统中有着非常重要的地位。关系数据库系统和非过程化的 SQL 之所以能够取得巨大的成功，关键是得益于查询优化技术的发展。关系查询优化是影响关系数据库管理系统性能的关键因素。

优化对关系系统来说既是挑战又是机遇。所谓挑战是指关系系统为了达到用户可接受的性能必须进行查询优化。由于关系表达式的语义级别很高，使关系系统可以从关系表达式中分析查询语义，提供了执行查询优化的可能性。这就为关系系统在性能上接近甚至超过非关系系统提供了机遇。

9.2.1 查询优化概述

关系系统的查询优化既是关系数据库管理系统实现的关键技术，又是关系系统的优点所在。它减轻了用户选择存取路径的负担。用户只要提出“干什么”，而不必指出“怎么干”。对比一下非关系系统中的情况：用户使用过程化的语言表达查询要求，至于执行何种记录级的操作，以及操作的序列是由用户而不是由系统来决定的。因此用户必须了解存取路径，系统要提供用户选择存取路径的手段，查询效率由用户的存取策略决定。如果用户做了不当的选择，系统是无法对此加以改进的。这就要求用户有较高的数据库技术和程序设计水平。

查询优化的优点不仅在于用户不必考虑如何最好地表达查询以获得较高的效率，而且在于系统可以比用户程序的“优化”做得更好。这是因为：

（1）优化器可以从数据字典中获取许多统计信息，例如每个关系表中的元组数、关系中每个属性值的分布情况、哪些属性上已经建立了索引等。优化器可以根据这些信息做出正确的估算，选择高效的执行计划，而用户程序则难以获得这些信息。

（2）如果数据库的物理统计信息改变了，系统可以自动对查询进行重新优化以选择相适应的执行计划。在非关系系统中则必须重写程序，而重写程序在实际应用中往往是不太可能的。

（3）优化器可以考虑数百种不同的执行计划，而程序员一般只能考虑有限的几种可能性。

（4）优化器中包括了很多复杂的优化技术，这些优化技术往往只有最好的程序员才能掌握。系统的自动优化相当于使得所有人都拥有这些优化技术。

目前关系数据库管理系统通过某种代价模型计算出各种查询执行策略的执行代价，然后选取代价最小的执行方案。在集中式数据库中，查询执行开销主要包括磁盘存取块数（I/O 代价）、处理机时间（CPU 代价）以及查询的内存开销。在分布式数据库中还要加上通信代价，即

总代价=I/O 代价+CPU 代价+内存代价+通信代价

由于磁盘 I/O 操作涉及机械动作，需要的时间与内存操作相比要高几个数量级，因此，在计算查询代价时一般用查询处理读写的块数作为衡量单位。

查询优化的总目标是选择有效的策略，求得给定关系表达式的值，使得查询代价较小。因为查询优化的搜索空间有时非常大，实际系统选择的策略不一定是最优的，而是较优的。

9.2.2 一个实例

首先通过一个简单的例子来说明为什么要进行查询优化。

[**例 9.3**] 求选修了 2 号课程的学生姓名。

用 SQL 语句表达：

```
SELECT Student.Sname
FROM Student,SC
WHERE Student.Sno=SC.Sno AND SC.Cno='2';
```

假定学生-课程数据库中有 1 000 个学生记录，10 000 个选课记录，其中选修 2 号课程的选课记录为 50 个。

系统可以用多种等价的关系代数表达式来完成这一查询，但分析下面三种就足以说明问题了：

$$Q_1=\Pi_{Sname}(\sigma_{Student.Sno=SC.Sno \wedge SC.Cno='2'}(Student \times SC))$$

$$Q_2=\Pi_{Sname}(\sigma_{SC.Cno='2'}(Student \bowtie SC))$$

$$Q_3=\Pi_{Sname}(Student \bowtie \sigma_{SC.Cno='2'}(SC))$$

后面将看到由于查询执行的策略不同，查询效率相差很大。

1. 第一种情况

（1）计算广义笛卡儿积

把 Student 和 SC 的每个元组连接起来。一般连接的做法是：在内存中尽可能多地装入某个表（如 Student 表）的若干块，留出一块存放另一个表（如 SC 表）的元组；然后把 SC 中的每个元组和 Student 中每个元组连接，连接后的元组装满一块后就写到中间文件上，再从 SC 中读入一块和内存中的 Student 元组连接，直到 SC 表处理完；这时再一次读入若干块 Student 元组，读入一块 SC 元组，重复上述处理过程，直到把 Student 表处理完。

设一个块能装 10 个 Student 元组或 100 个 SC 元组，在内存中存放 5 块 Student 元组和 1 块 SC 元组，则读取总块数为

$$\frac{1000}{10}+\frac{1000}{10\times5}\times\frac{10000}{100}=100+20\times100=2100 \text{ 块}$$

其中，读 Student 表 100 块，读 SC 表 20 遍，每遍 100 块，则总计要读取 2 100 数据块。

连接后的元组数为 $10^3\times10^4=10^7$。设每块能装 10 个元组，则写出 10^6 块。

（2）作选择操作

依次读入连接后的元组，按照选择条件选取满足要求的记录。假定内存处理时间忽略。这一步读取中间文件花费的时间（同写中间文件一样）需读入 10^6 块。若满足条件的元组假设仅 50 个，均可放在内存。

（3）作投影操作

把第（2）步的结果在 Sname 上作投影输出，得到最终结果。

因此第一种情况下执行查询的总读写数据块=$2100+10^6+10^6$。

2. 第二种情况

（1）计算自然连接

为了执行自然连接，读取 Student 和 SC 表的策略不变，总的读取块数仍为 2 100 块。但自然连接的结果比第一种情况大大减少，连接后的元组数为 10^4 个元组，写出数据块 $=10^3$ 块。

（2）读取中间文件块，执行选择操作，读取的数据块$=10^3$ 块。

（3）把第（2）步结果投影输出。

第二种情况下执行查询的总读写数据块$=2100+10^3+10^3$。其执行代价大约是第一种情况的 488 分之一。

3. 第三种情况

（1）先对 SC 表作选择操作，只需读一遍 SC 表，存取块数为 100 块，因为满足条件的元组仅 50 个，不必使用中间文件。

（2）读取 Student 表，把读入的 Student 元组和内存中的 SC 元组作连接。也只需读一遍 Student 表，共 100 块。

（3）把连接结果投影输出。

第三种情况总的读写数据块=100+100。其执行代价大约是第一种情况的万分之一，是第二种情况是 20 分之一。

假如 SC 表的 Cno 字段上有索引，第一步就不必读取所有的 SC 元组而只需读取 Cno='2'的那些元组（50 个）。存取的索引块和 SC 中满足条件的数据块大约共 3～4 块。若 Student 表在 Sno 上也有索引，则第二步也不必读取所有的 Student 元组，因为满足条件的 SC 记录仅 50 个，涉及最多 50 个 Student 记录，因此读取 Student 表的块数也可大大减少。

这个简单的例子充分说明了查询优化的必要性，同时也给出一些查询优化方法的初步概念。例如，读者可能已经发现，在第一种情况下，连接后的元组可以先不立即写出，而是和下面第（2）步的选择操作结合，这样可以省去写出和读入的开销。有选择和连接操作时，应当先做选择操作，例如，把上面的代数表达式 Q_1、Q_2 变换为 Q_3，这样参加连接的元组就可以大大减少，这是代数优化。在 Q_3 中，SC 表的选择操作算法可以采用全表扫描或索引扫描，经过初步估算，索引扫描方法较优。同样对于 Student 和 SC 表的连接，利用 Student 表上的索引，采用索引连接代价也较小，这就是物理优化。

9.3 代数优化

9.1 中已经讲解了 SQL 语句经过查询分析、查询检查后变换为查询树，它是关系代数表达式的内部表示。本节介绍基于关系代数等价变换规则的优化方法，即代数优化。

9.3.1 关系代数表达式等价变换规则

代数优化策略是通过对关系代数表达式的等价变换来提高查询效率。所谓关系代数表达式的等价是指用相同的关系代替两个表达式中相应的关系所得到的结果是相同的。两个关系表达式 E_1 和 E_2 是等价的，可记为 $E_1 \equiv E_2$。

下面是常用的等价变换规则，证明从略。

1. 连接、笛卡儿积的交换律

设 E_1 和 E_2 是关系代数表达式，F 是连接运算的条件，则有

$$E_1 \times E_2 \equiv E_2 \times E_1$$

$$E_1 \bowtie E_2 \equiv E_2 \bowtie E_1$$

$$E_1 \underset{F}{\bowtie} E_2 \equiv E_2 \underset{F}{\bowtie} E_1$$

2. 连接、笛卡儿积的结合律

设 E_1、E_2、E_3 是关系代数表达式，F_1 和 F_2 是连接运算的条件，则有

$$(E_1 \times E_2) \times E_3 \equiv E_1 \times (E_2 \times E_3)$$

$$(E_1 \bowtie E_2) \bowtie E_3 \equiv E_1 \bowtie (E_2 \bowtie E_3)$$

$$(E_1 \underset{F_1}{\bowtie} E_2) \underset{F_2}{\bowtie} E_3 \equiv E_1 \underset{F_1}{\bowtie} (E_2 \underset{F_2}{\bowtie} E_3)$$

3. 投影的串接定律

$$\Pi_{A_1,A_2,\cdots,A_n}(\Pi_{B_1,B_2,\cdots,B_m}(E)) \equiv \Pi_{A_1,A_2,\cdots,A_n}(E)$$

这里，E 是关系代数表达式，$A_i(i=1, 2, \cdots, n)$，$B_j(j=1, 2, \cdots, m)$是属性名，且$\{A_1, A_2, \cdots, A_n\}$构成$\{B_1, B_2, \cdots, B_m\}$的子集。

4. 选择的串接定律

$$\sigma_{F_1}(\sigma_{F_2}(E)) \equiv \sigma_{F_1 \wedge F_2}(E)$$

这里，E 是关系代数表达式，F_1、F_2 是选择条件。选择的串接律说明选择条件可以合并，这样一次就可检查全部条件。

5. 选择与投影操作的交换律

$$\sigma_F(\Pi_{A_1,A_2,\cdots,A_n}(E)) \equiv \Pi_{A_1,A_2,\cdots,A_n}(\sigma_F(E))$$

这里，选择条件 F 只涉及属性 $A_1, \cdots, A_n$。

若 F 中有不属于 $A_1, \cdots, A_n$ 的属性 $B_1, \cdots, B_m$，则有更一般的规则：

$$\Pi_{A_1,A_2,\cdots,A_n}(\sigma_F(E)) \equiv \Pi_{A_1,A_2,\cdots,A_n}(\sigma_F(\Pi_{A_1,A_2,\cdots,A_n,B_1,B_2,\cdots,B_m}(E)))$$

6. 选择与笛卡儿积的交换律

如果 F 中涉及的属性都是 E_1 中的属性，则

$$\sigma_F(E_1 \times E_2) \equiv \sigma_F(E_1) \times E_2$$

如果 $F=F_1 \wedge F_2$，并且 F_1 只涉及 E_1 中的属性，F_2 只涉及 E_2 中的属性，则由上面的等

价变换规则1、4、6可推出

$$\sigma_F(E_1\times E_2)\equiv\sigma_{F_1}(E_1)\times\sigma_{F_2}(E_2)$$

若F_1只涉及E_1中的属性，F_2涉及E_1和E_2两者的属性，则仍有

$$\sigma_F(E_1\times E_2)\equiv\sigma_{F_2}(\sigma_{F_1}(E_1)\times E_2)$$

它使部分选择在笛卡儿积前先做。

7. 选择与并的分配律

设$E=E_1\cup E_2$，E_1、E_2有相同的属性名，则

$$\sigma_F(E_1\cup E_2)\equiv\sigma_F(E_1)\cup\sigma_F(E_2)$$

8. 选择与差运算的分配律

若E_1与E_2有相同的属性名，则

$$\sigma_F(E_1-E_2)\equiv\sigma_F(E_1)-\sigma_F(E_2)$$

9. 选择对自然连接的分配律

$$\sigma_F(E_1\bowtie E_2)\equiv\sigma_F(E_1)\bowtie\sigma_F(E_2)$$

F只涉及E_1与E_2的公共属性。

10. 投影与笛卡儿积的分配律

设E_1和E_2是两个关系表达式，A_1，…，A_n是E_1的属性，B_1，…，B_m是E_2的属性，则

$$\Pi_{A_1,A_2,\cdots,A_n,B_1,B_2,\cdots,B_m}(E_1\times E_2)\equiv\Pi_{A_1,A_2,\cdots,A_n}(E_1)\times\Pi_{B_1,B_2,\cdots,B_m}(E_2)$$

11. 投影与并的分配律

设E_1和E_2有相同的属性名，则

$$\Pi_{A_1,A_2,\cdots,A_n}(E_1\cup E_2)\equiv\Pi_{A_1,A_2,\cdots,A_n}(E_1)\cup\Pi_{A_1,A_2,\cdots,A_n}(E_2)$$

9.3.2 查询树的启发式优化

本节讨论应用启发式规则（heuristic rules）的代数优化。这是对关系代数表达式的查询树进行优化的方法。典型的启发式规则有：

（1）选择运算应尽可能先做。在优化策略中这是最重要、最基本的一条。它常常可使执行代价节约几个数量级，因为选择运算一般使计算的中间结果大大变小。

（2）把投影运算和选择运算同时进行。如有若干投影和选择运算，并且它们都对同一个关系操作，则可以在扫描此关系的同时完成所有这些运算以避免重复扫描关系。

（3）把投影同其前或后的双目运算结合起来，没有必要为了去掉某些字段而扫描一遍关系。

（4）把某些选择同在它前面要执行的笛卡儿积结合起来成为一个连接运算，连接（特别是等值连接）运算要比同样关系上的笛卡儿积省很多时间。

（5）找出公共子表达式。如果这种重复出现的子表达式的结果不是很大的关系，并且从外存中读入这个关系比计算该子表达式的时间少得多，则先计算一次公共子表达式并把结果写入中间文件是合算的。当查询的是视图时，定义视图的表达式就是公共子表达式的情况。

下面给出遵循这些启发式规则，应用 9.3.1 的等价变换公式来优化关系表达式的算法。

算法：关系表达式的优化。

输入：一个关系表达式的查询树。

输出：优化的查询树。

方法：

（1）利用等价变换规则 4，把形如 $\sigma_{F_1 \wedge F_2 \wedge \cdots \wedge F_n}(E)$ 的表达式变换为 $\sigma_{F_1}(\sigma_{F_2}(\cdots(\sigma_{F_n}(E))\cdots))$。

（2）对每一个选择，利用等价变换规则 4～9 尽可能把它移到树的叶端。

（3）对每一个投影，利用等价变换规则 3、5、10、11 中的一般形式尽可能把它移向树的叶端。

注意：等价变换规则 3 使一些投影消失，而规则 5 把一个投影分裂为两个，其中一个有可能被移向树的叶端。

（4）利用等价变换规则 3～5，把选择和投影的串接合并成单个选择、单个投影或一个选择后跟一个投影，使多个选择或投影能同时执行，或在一次扫描中全部完成，尽管这种变换似乎违背“投影尽可能早做”的原则，但这样做效率更高。

（5）把上述得到的语法树的内结点分组。每一双目运算（×，⋈，∪，-）和它所有的直接祖先为一组（这些直接祖先是（σ，Π 运算）。如果其后代直到叶子全是单目运算，则也将它们并入该组，但当双目运算是笛卡儿积（×），而且后面不是与它组成等值连接的选择时，则不能把选择与这个双目运算组成同一组。把这些单目运算单独分为一组。

［**例 9.4**］ 下面给出例 9.3 中 SQL 语句的代数优化示例。

```
SELECT Student.Sname FROM Student, SC WHERE Student.Sno=SC.Sno AND SC.Cno='2';
```

（1）把 SQL 语句转换成查询树，如图 9.3 所示。

为了使用关系代数表达式的优化法，不妨假设内部表示是关系代数语法树，则上面的查询树如图 9.4 所示。

（2）对查询树进行优化。

利用规则 4、6 把选择 $\sigma_{SC.Cno='2'}$ 移到叶端，图 9.4 查询树便转换成图 9.5 优化的查询树。这就是 9.2.2 节中 Q_3 的查询树表示。前面已经分析了 Q_3 比 Q_1、Q_2 查询效率要高得多。

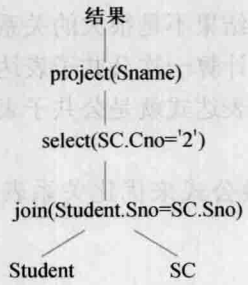

图 9.3 查询树图

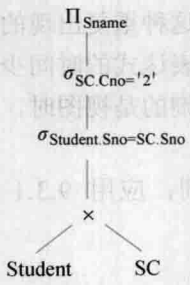

图 9.4 关系代数语法树图

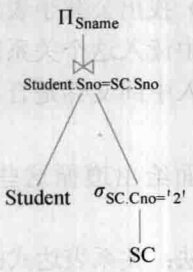

图 9.5 优化后的查询树

9.4 物理优化

代数优化改变查询语句中操作的次序和组合，但不涉及底层的存取路径。9.1.2 小节中已经讲解了对每一种操作有多种执行这个操作的算法，有多条存取路径，因此对于一个查询语句有许多存取方案，它们的执行效率不同，有的会相差很大。因此，仅仅进行代数优化是不够的。物理优化就是要选择高效合理的操作算法或存取路径，求得优化的查询计划，达到查询优化的目标。

选择的方法可以是：

（1）基于规则的启发式优化。启发式规则是指那些在大多数情况下都适用，但不是在每种情况下都是最好的规则。

（2）基于代价估算的优化。使用优化器估算不同执行策略的代价，并选出具有最小代价的执行计划。

（3）两者结合的优化方法。查询优化器通常会把这两种技术结合在一起使用。因为可能的执行策略很多，要穷尽所有的策略进行代价估算往往是不可行的，会造成查询优化本身付出的代价大于获得的益处。为此，常常先使用启发式规则，选取若干较优的候选方案，减少代价估算的工作量；然后分别计算这些候选方案的执行代价，较快地选出最终的优化方案。

9.4.1 基于启发式规则的存取路径选择优化

1. 选择操作的启发式规则

对于小关系，使用全表顺序扫描，即使选择列上有索引。

对于大关系，启发式规则有：

（1）对于选择条件是“主码=值”的查询，查询结果最多是一个元组，可以选择主码索引。一般的关系数据库管理系统会自动建立主码索引。

（2）对于选择条件是“非主属性=值”的查询，并且选择列上有索引，则要估算查询结果的元组数目，如果比例较小（<10%）可以使用索引扫描方法，否则还是使用全表顺序扫描。

（3）对于选择条件是属性上的非等值查询或者范围查询，并且选择列上有索引，同样要估算查询结果的元组数目，如果选择率<10%可以使用索引扫描方法，否则还是使用全表顺序扫描。

（4）对于用 AND 连接的合取选择条件，如果有涉及这些属性的组合索引，则优先采用组合索引扫描方法；如果某些属性上有一般索引，则可以用［例 9.1-C4］中介绍的索引扫描方法，否则使用全表顺序扫描。

（5）对于用 OR 连接的析取选择条件，一般使用全表顺序扫描。

2. 连接操作的启发式规则

（1）如果 2 个表都已经按照连接属性排序，则选用排序-合并算法。

（2）如果一个表在连接属性上有索引，则可以选用索引连接算法。

（3）如果上面 2 个规则都不适用，其中一个表较小，则可以选用 hash join 算法。

（4）最后可以选用嵌套循环算法，并选择其中较小的表，确切地讲是占用的块数（B）较少的表，作为外表（外循环的表）。理由如下：

设连接表 R 与 S 分别占用的块数为 Br 与 Bs，连接操作使用的内存缓冲区块数为 K，分配 K-1 块给外表。如果 R 为外表，则嵌套循环法存取的块数为 Br+ BrBs /(K-1)，显然应该选块数小的表作为外表。

上面列出了一些主要的启发式规则，在实际的关系数据库管理系统中启发式规则要多得多。

9.4.2 基于代价估算的优化

启发式规则优化是定性的选择，比较粗糙，但是实现简单而且优化本身的代价较小，适合解释执行的系统。因为解释执行的系统，其优化开销包含在查询总开销之中。在编译执行的系统中，一次编译优化，多次执行，查询优化和查询执行是分开的。因此，可以采用精细复杂一些的基于代价的优化方法。

1. 统计信息

基于代价的优化方法要计算各种操作算法的执行代价，它与数据库的状态密切相关。为此在数据字典中存储了优化器需要的统计信息（database statistics），主要包括如下几个方面：

（1）对每个基本表，该表的元组总数（N）、元组长度（l）、占用的块数（B）、占用的溢出块数（BO）；

（2）对基本表的每个列，该列不同值的个数（m）、该列最大值、最小值，该列上是否

已经建立了索引，是哪种索引（*B*+树索引、hash 索引、聚集索引）。根据这些统计信息,可以计算出谓词条件的选择率（*f*），如果不同值的分布是均匀的，*f*=1/*m*；如果不同值的分布不均匀，则要计算每个值的选择率，*f*=具有该值的元组数/*N*；

（3）对索引，例如 *B*+树索引，该索引的层数（*L*）、不同索引值的个数、索引的选择基数 *S*（有 *S* 个元组具有某个索引值）、索引的叶结点数（*Y*）；

等等。

2. 代价估算示例

下面给出若干操作算法的执行代价估算。

（1）全表扫描算法的代价估算公式

如果基本表大小为 *B* 块，全表扫描算法的代价 cost=*B*；

如果选择条件是“码=值”，那么平均搜索代价 cost=*B*/2。

（2）索引扫描算法的代价估算公式

如果选择条件是“码=值”，如例 9.1-C2，则采用该表的主索引，若为 *B*+树，层数为 *L*，需要存取 *B*+树中从根结点到叶结点 *L* 块，再加上基本表中该元组所在的那一块，所以 cost=*L*+1。

如果选择条件涉及非码属性，如例 9.1-C3，若为 *B*+树索引，选择条件是相等比较，*S* 是索引的选择基数(有 *S* 个元组满足条件)。因为满足条件的元组可能会保存在不同的块上，所以（最坏的情况）cost=*L*+*S*。

如果比较条件是>，>=，<，<=操作，假设有一半的元组满足条件，那么就要存取一半的叶结点，并通过索引访问一半的表存储块。所以 cost=*L*+*Y*/2+*B*/2。如果可以获得更准确的选择基数，可以进一步修正 *Y*/2 与 *B*/2。

（3）嵌套循环连接算法的代价估算公式

9.4.1 中已经讨论过了嵌套循环连接算法的代价 cost=Br+BrBs/(*K*-1)。如果需要把连接结果写回磁盘，则 cost=Br+BrBs/(*K*-1)+(Frs*Nr*Ns)/Mrs。其中 Frs 为连接选择率（join selectivity），表示连接结果元组数的比例，Mrs 是存放连接结果的块因子，表示每块中可以存放的结果元组数目。

（4）排序-合并连接算法的代价估算公式

如果连接表已经按照连接属性排好序，则 cost=Br+Bs+(Frs*Nr*Ns)/Mrs。

如果必须对文件排序，那么还需要在代价函数中加上排序的代价。对于包含 *B* 个块的文件排序的代价大约是(2*B)+(2*B*$\log_2 B$)。

上面仅仅列出了少数操作算法的代价估算示例。在实际的关系数据库管理系统中代价估算公式要多得多，也复杂得多。

前面还提到一种优化的方法，称为语义优化。这种技术根据数据库的语义约束，把原先的查询转换成另一个执行效率更高的查询。本章不对这种方法进行详细讨论，只用一个

简单的例子来说明它。考虑例 9.1 的 SQL 查询：

```
SELECT * FROM Student WHERE Sdept='CS' AND Sage>200;
```

显然，用户在写年龄值 Sage 时，误把 20 写成 200 了。假设数据库模式上定义了一个约束，要求学生年龄在 15—55 岁之间。一旦查询优化器检查到了这条约束，它就知道上面查询的结果为空，所以根本不用执行这个查询。

*9.5 查询计划的执行

查询优化完成后，关系数据库管理系统为用户查询生成了一个查询计划。该查询计划的执行可以分为自顶向下和自底向上两种执行方法。

在自顶向下的执行方式中，系统反复向查询计划顶端的操作符发出需要查询结果元组的请求，操作符收到请求后，就试图计算下一个（几个）元组并返回这些元组。在计算时，如果操作符的输入缓冲区为空，它就会向其孩子操作符发送需求元组的请求……这种需求元组的请求会一直传到叶子结点，启动叶子操作符运行，并返回其父操作符一个（几个）元组，父操作符再计算自己的输出返回给上层操作符，直至顶端操作符。重复这一过程，直到处理完整个关系。

在自底向上的执行方式中，查询计划从叶结点开始执行，叶结点操作符不断地产生元组并将它们放入其输出缓冲区中，直到缓冲区填满为止，这时它必须等待其父操作符将元组从该缓冲区中取走才能继续执行。然后其父结点操作符开始执行，利用下层的输入元组来产生它自己的输出元组，直到其输出缓冲区满为止。这个过程不断重复，直到产生所有的输出元组。

显然，自顶向下的执行方式是一种被动的、需求驱动的执行方式。而自底向上的执行方式是一种主动的执行方式。详细的介绍请参阅关系数据库管理系统实现的有关文献。

9.6 小 结

查询处理是关系数据库管理系统的核心，而查询优化技术又是查询处理的关键技术。本章仅关注查询（Query）语句，它是关系数据库管理系统语言处理中最重要、最复杂的部分。更一般的数据库语言（包括数据定义语言、数据操纵语言、数据控制语言）处理技术可参阅关系数据库管理系统实现的有关文献。

本章讲解了启发式代数优化、基于规则的存取路径优化和基于代价估算的优化等方法，实际系统的优化方法是综合的，优化器是十分复杂的。

本章不是要求读者掌握关系数据库管理系统查询处理和查询优化的内部实现技术，因此没有详细讲解技术细节。本章的目的是希望读者掌握查询优化方法的概念和技术；通过

本章实验，进一步了解具体的查询计划表示，能够利用它分析查询的实际执行方案和查询代价，进而通过建立索引或者修改 SQL 语句来降低查询代价，达到优化系统性能的目标。

对于比较复杂的查询，尤其是涉及连接和嵌套的查询，不要把优化的任务全部放在关系数据库管理系统上，应该找出关系数据库管理系统的优化规律，以写出适合关系数据库管理系统自动优化的 SQL 语句。对于关系数据库管理系统不能优化的查询需要重写查询语句，进行手工调整以优化性能。

习 题

1．试述查询优化在关系数据库系统中的重要性和可能性。

2．假设关系 $R(A, B)$和 $S(B, C, D)$情况如下：R 有 20 000 个元组，S 有 1 200 个元组，一个块能装 40 个 R 的元组，能装 30 个 S 的元组，估算下列操作需要多少次磁盘块读写。

（1）R 上没有索引，select * from R;

（2）R 中 A 为主码，A 有 3 层 B+树索引，select * from R where $A = 10$;

（3）嵌套循环连接 $R \bowtie S$;

（4）排序合并连接 $R \bowtie S$，区分 R 与 S 在 B 属性上已经有序和无序两种情况。

3．对学生-课程数据库，查询信息系学生选修了的所有课程名称。

```
SELECT Cname
FROM Student, Course, SC
WHERE Student.Sno=SC.Sno AND SC.Cno=Course.Cno AND Student.Sdept='IS';
```

试画出用关系代数表示的语法树，并用关系代数表达式优化算法对原始的语法树进行优化处理，画出优化后的标准语法树。

4．对于下面的数据库模式

Teacher (Tno, Tname, Tage, Tsex); Department (Dno, Dname, Tno); Work (Tno, Dno, Year, Salary)

假设 Teacher 的 Tno 属性、Department 的 Dno 属性以及 Work 的 Year 属性上有 B+树索引，说明下列查询语句的一种较优的处理方法。

（1）select * from teacher where Tsex = '女'

（2）select * from department where Dno < 301

（3）select * from work where Year <> 2000

（4）select * from work where year > 2000 and salary < 5000

（5）select * from work where year < 2000 or salary < 5000

5．对于题 4 中的数据库模式，有如下的查询：

```
select Tname
from teacher, department, work
```

where teacher.tno = work.tno and department.dno = work.dno and

department.dname = '计算机系' and salary > 5000

画出语法树以及用关系代数表示的语法树，并对关系代数语法树进行优化，画出优化后的语法树。

6．试述关系数据库管理系统查询优化的一般准则。

7．试述关系数据库管理系统查询优化的一般步骤。

实　验

实验 9　数据库监视与性能优化

理解和掌握数据库监视与性能优化的基本原理和方法。掌握数据库性能调优的方法，包括使用 EXPLAIN 命令分析查询执行计划、利用索引优化查询性能、优化 SQL 语句，以及理解和掌握数据库模式规范化设计对查询性能的影响，并能针对给定的数据库模式设计不同的实例验证查询性能优化效果。

本章参考文献

[1]　CODD E F．Relational Database：a Practical Foundation for Productivity．CACM，1982(25):2.

（文献[1]是 CODD E F 获得图灵奖后的演说，阐述了关系数据库系统能极大地提高用户生产率这一重要观点。）

[2]　SMITH J M，CHANG P Y T．Optimizing the Performance of a Relational Algebra Database Interface．CACM，1975(18):10.

（文献［2］研究了关系代数的优化，给出了详细的算法。）

[3]　WONG E，YOUSSEFI K．Decomposition：a Strategy for Query Processing．ACM TODS，1976(1):3.

[4]　YOUSSEFI X，WONG E．Query Processing in a Relational Database Management System．in Proceedings of the 5th International Conference on Very Large Data Bases，1979.

（文献［3］和［4］介绍了 INGRES 采用的查询分解优化方法。）

[5]　AHO A V，SAGIV Y，ULLMAN J D．Efficient Optimization of a Class of Relational Expressions．ACM TODS 1979(4):4.

（文献［5］讨论了关系表达式的优化方法。）

[6]　ASTRAHAN M，et al．System R：A Relational Approach to Data Base Management．ACM TODS，1976(1):2.

（文献［6］介绍了 System R 的基于代价估算的查询优化算法。）

[7]　SELINGER P，et al．Access Path Selection in a Relational Database Management System．in Proceedings of ACM SIGMOD，1979.

（文献［7］讨论了 System R 中多表连接操作的查询优化问题。）

[8] ASTRAHAN M M，SCHKOLNICK M. Performance of the System R Access Path Selection Mechanism. IFIP，1980.

（文献［8］介绍了 System R 的存取路径选择机制及其性能。）

[9] KIM W. On Optimizing an SQL like Nested Query. TODS，1982(3):3.

（文献［9］提出了 SQL 嵌套查询的优化方法。）

[10] DEWITT D，et al. Implementation Techniques for Main Memory Databases. in Proceedings of ACM SIGMOD，1984.

（文献［10］研究了主存数据库上的查询优化问题。）

[11] YAO S B. Optimization of Query Evaluation Algorithms. ACM TODS，1979(4):2.

（文献［11］对多个查询优化算法进行了比较和分析。）

[12] JARKE M，KOCH J. Query Optimization in Database Systems. ACM Computing Surveys，1984(16):2.

（文献［12］系统地综述了查询优化的主要研究成果，并列出了这一研究领域的主要文献。）

[13] KING J. QUIST：a System for Semantic Query Optimization in Relational Databases. In Proceedings of VLDB，1981.

（文献［13］讨论了在数据库中利用语义知识进行查询优化的问题。）

[14] MALLEY C，ZDONICK S. a Knowledge Based Approach to Query Optimization. in Proceedings of the 1st Inter. Conference on Expert Database Systems，1986.

（文献［14］也是讨论在数据库中利用语义知识进行查询优化的问题。）

[15] BECK H，GALA S，NAVATHE S. Classification as a Query Processing Technique in the CANDIDE Semantic Data Model.in Proceedings of IEEE Inter. Conference on Data Engineering，1989.

（文献［15］讨论了一种实现查询优化的分类技术，这种技术适用于基于语义数据模型的数据库系统。）

[16] MOLINA H G, ULLMAN J D, WIDOM J. 数据库管理系统实现. 2 版. 杨冬青，唐世谓，徐其钧，等，译. 北京：机械工业出版社，2010.

（文献［16］详细介绍了关系数据库管理系统的实现技术。）