

数据库原理与技术



An Introduction to Database System

第九章 关系查询处理和查询优化

安徽农业大学信息与计算机学院



第九章 关系系统及其查询优化



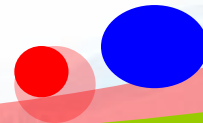
9.1 关系数据库系统的查询处理

9.2 关系数据库系统的查询优化

9.3 代数优化

9.4 物理优化

9.5 小结



关系系统及其查询优化（续）



❖ 本章目的：

- RDBMS的查询处理步骤
- 查询优化的概念
- 基本方法和技术

❖ 查询优化分类：

- 代数优化
- 物理优化



9.1 关系数据库系统的查询处理



❖ 9.1.1 查询处理步骤

❖ 9.1.2 实现查询操作的算法示例

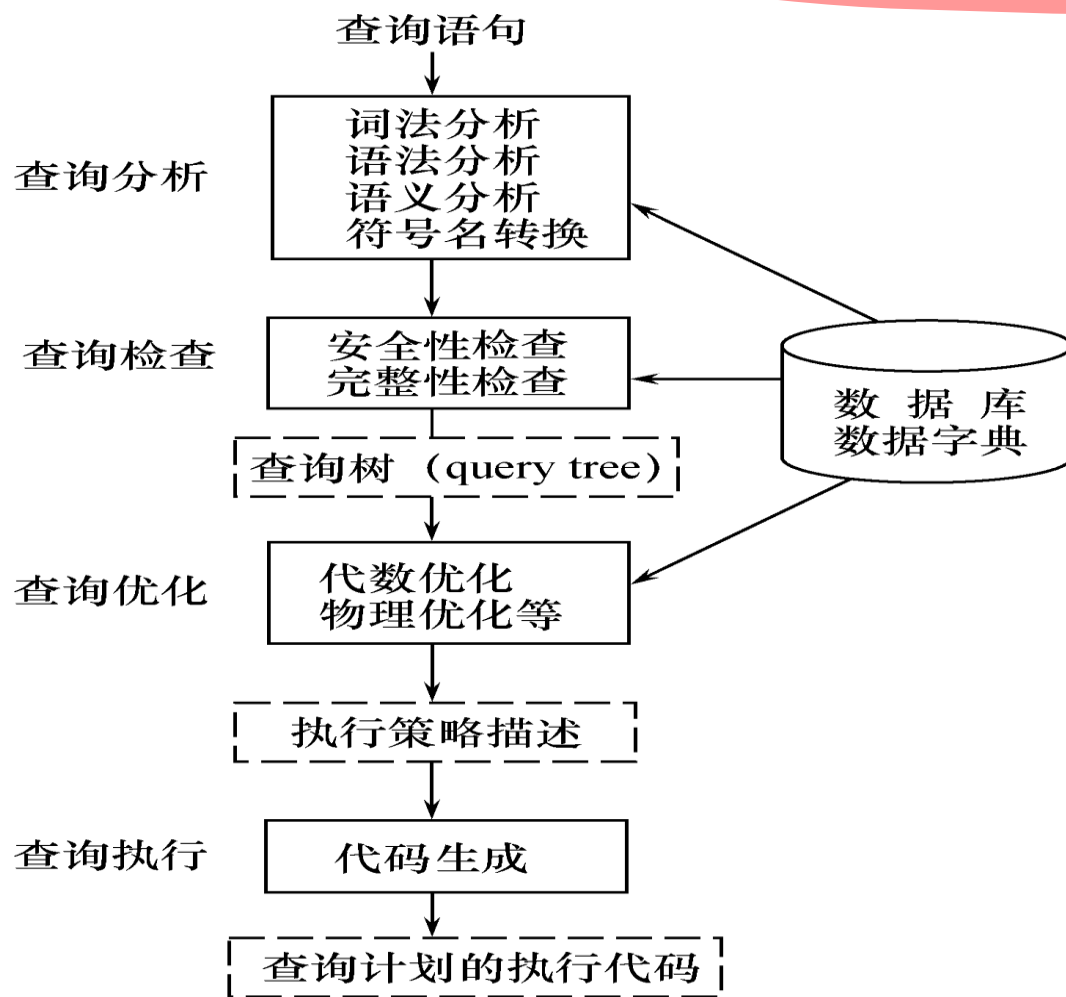
9.1.1 查询处理步骤

❖ RDBMS查询处理阶段：

1. 查询分析
2. 查询检查
3. 查询优化
4. 查询执行



查询处理步骤（续）



1. 查询分析

- ❖ 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
- ❖ 进行语法检查和语法分析



2. 查询检查



- ❖ 根据数据字典对合法的查询语句进行语义检查
- ❖ 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
- ❖ 检查通过后把SQL查询语句转换成等价的关系代数表达式
- ❖ RDBMS一般都用查询树(语法分析树)来表示扩展的关系代数表达式
- ❖ 把数据库对象的外部名称转换为内部表示

3. 查询优化

❖ 查询优化：选择一个高效执行的查询处理策略



❖ 查询优化分类：

- 代数优化：指关系代数表达式的优化
- 物理优化：指存取路径和底层操作算法的选择

❖ 查询优化方法选择的依据：

- 基于规则(rule based)
- 基于代价(cost based)
- 基于语义(semantic based)

4. 查询执行

- ❖ 依据优化器得到的执行策略生成查询计划
- ❖ 代码生成器(code generator)生成执行查询计划的代码



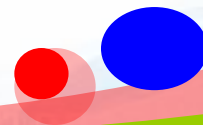


9.1 关系数据库系统的查询处理



❖ 9.1.1 查询处理步骤

❖ 9.1.2 实现查询操作的算法示例



9.1.2 实现查询操作的算法示例

- ❖ 一、选择操作的实现
- ❖ 二、连接操作的实现



一、选择操作的实现

❖ [例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='200215121'`;

C3: `Sage>20`;

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



选择操作的实现（续）



❖ 选择操作典型实现方法：

■ 1. 简单的全表扫描方法

- 对查询的基本表顺序扫描，逐一检查每个元组是否满足选择条件，把满足条件的元组作为结果输出
- 适合小表，不适合大表

■ 2. 索引(或散列)扫描方法

- 适合选择条件中的属性上有索引(例如B+树索引或Hash索引)
- 通过索引先找到满足条件的元组主码或元组指针，再通过元组指针直接在查询的基本表中找到元组

选择操作的实现（续）

- ❖ [例1-C2] 以C2为例，Sno = ‘200215121’，并且Sno上有索引(或Sno是散列码)
 - 使用索引(或散列)得到Sno为‘200215121’元组的指针
 - 通过元组指针在student表中检索到该学生
- ❖ [例1-C3] 以C3为例，Sage > 20，并且Sage上有B+树索引
 - 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
 - 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。



选择操作的实现（续）

- ❖ [例1-C4] 以C4为例， $Sdept = 'CS' \text{ AND } Sage > 20$ ，如果 $Sdept$ 和 $Sage$ 上都有索引：
- 算法一：分别用上面两种方法分别找到 $Sdept = 'CS'$ 的一组元组指针和 $Sage > 20$ 的另一组元组指针
 - 求这2组指针的交集
 - 到 $student$ 表中检索
 - 得到计算机系年龄大于20的学生
 - 算法二：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
 - 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。



二、连接操作的实现



- ❖ 连接操作是查询处理中最耗时的操作之一
- ❖ 本节只讨论等值连接(或自然连接)最常用的实现算法
- ❖ [例2] `SELECT * FROM Student, SC
WHERE Student.Sno=SC.Sno;`

连接操作的实现（续）



- ❖ 1. 嵌套循环方法(nested loop)
- ❖ 2. 排序-合并方法(sort-merge join 或merge join)
- ❖ 3. 索引连接(index join)方法
- ❖ 4. Hash Join方法

连接操作的实现（续）



1. 嵌套循环方法(nested loop)

- 对外层循环(Student)的每一个元组(s)，检索内层循环(SC)中的每一个元组(sc)
- 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

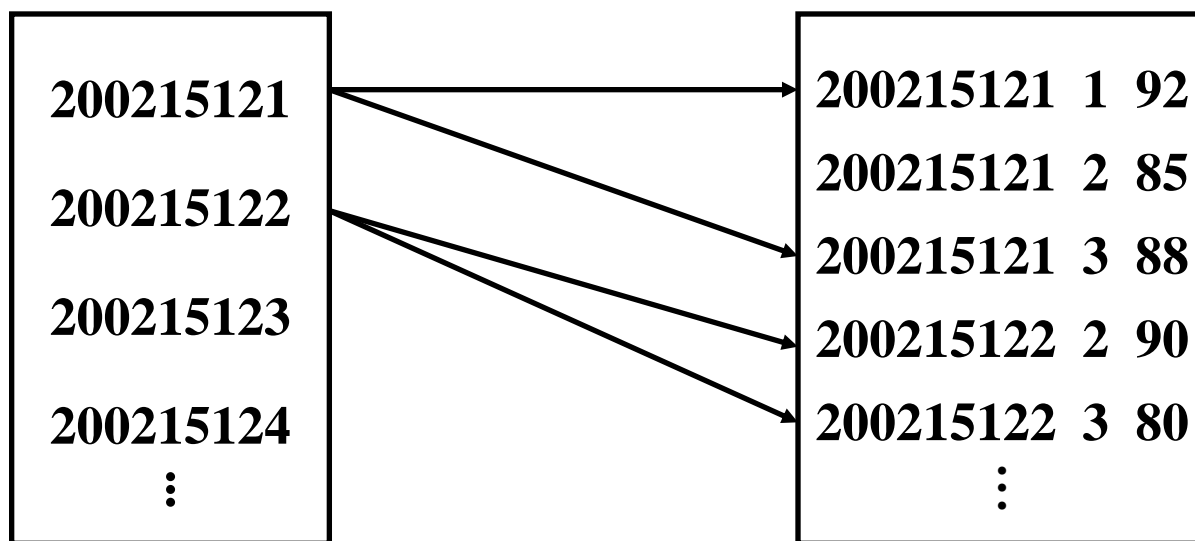
连接操作的实现（续）

2. 排序-合并方法(sort-merge join 或merge join)



- 适合连接的诸表已经排好序的情况
- 排序—合并连接方法的步骤：
 - 如果连接的表没有排好序，先对Student表和SC表按连接属性Sno排序
 - 取Student表中第一个Sno，依次扫描SC表中具有相同Sno的元组

连接操作的实现（续）



排序-合并连接方法示意图

连接操作的实现（续）



- 排序—合并连接方法的步骤（续）：
 - 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

连接操作的实现（续）



- ❖ Student表和SC表都只要扫描一遍
- ❖ 如果2个表原来无序，执行时间要加上对两个表的排序时间
- ❖ 对于2个大表，先排序后使用sort-merge join方法执行连接，总的时间一般仍会大大减少

连接操作的实现（续）

3. 索引连接(index join)方法

❖ 步骤：

- ① 在SC表上建立属性Sno的索引，如果原来没有该索引
 - 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



连接操作的实现（续）



4. Hash Join方法

- 把连接属性作为hash码，用同一个hash函数把R和S中的元组散列到同一个hash文件中
- 步骤：
 - 划分阶段(partitioning phase):
 - 对包含较少元组的表(比如R)进行一遍处理
 - 把它的元组按hash函数分散到hash表的桶中
 - 试探阶段(probing phase): 也称为连接阶段(join phase)
 - 对另一个表(S)进行一遍处理
 - 把S的元组散列到适当的hash桶中
 - 把元组与桶中所有来自R并与之相匹配的元组连接起来

连接操作的实现（续）

- ❖ 上面hash join算法前提：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
- ❖ 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



第九章 关系系统及其查询优化



9.1 关系数据库系统的查询处理

9.2 关系数据库系统的查询优化

9.3 代数优化

9.4 物理优化

9.5 小结

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
- ❖ 关系查询优化是影响RDBMS性能的关键因素
- ❖ 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

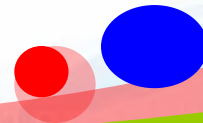


9.2 关系数据库系统的查询优化



❖ 9.2.1 查询优化概述

❖ 9.2.2 一个实例



9.2.1 查询优化概述

❖ 关系系统的查询优化

❖ 非关系系统



查询优化概述（续）

- ❖ 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好的效率，而且在于系统可以比用户程序的“优化”做得更好
 - (1) 优化器可以从数据字典中获取许多统计信息，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重新优化以选择相适应的执行计划。在非关系系统中必须重写程序，而重写程序在实际应用中往往是不太可能的。

查询优化概述（续）

- (3) 优化器可以考虑数百种不同的执行计划，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



查询优化概述（续）

❖ **RDBMS**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

- 集中式数据库

- 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)

- 查询的内存开销

- I/O代价是最主要的

- 分布式数据库

- 总代价=I/O代价+CPU代价+内存代价+通信代价

查询优化概述（续）

❖ 查询优化的总目标：

- 选择有效的策略
- 求得给定关系表达式的值
- 使得查询代价最小(实际上是较小)





9.2 关系数据库系统的查询优化

❖ 9.2.1 查询优化概述

❖ 9.2.2 一个实例



9.2.2 一个实例

[例3] 求选修了2号课程的学生姓名。用SQL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1000个学生记录，10000个选课记录
- 其中选修2号课程的选课记录为50个

一个实例（续）

❖ 系统可以用多种等价的关系代数表达式来完成这一查询

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

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

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



一个实例（续）

❖ 一、第一种情况

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

1. 计算广义笛卡尔积

❖ 把Student和SC的每个元组连接起来的做法：

- 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。
- 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
- 从SC中读入一块和内存中的Student元组连接，直到SC表处理完。
- 再读入若干块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



一个实例（续）

- ❖ 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

$$\frac{1000}{10} + \frac{1000}{10 \times 5} \times \frac{10000}{100} = 100 + 20 \times 100 = 2100 \text{ 块}$$

- ❖ 其中，读Student表100块。读SC表20遍，每遍100块。若每秒读写20块，则总计要花105s
- ❖ 连接后的元组数为 $10^3 \times 10^4 = 10^7$ 。设每块能装10个元组，则写出这些块要用 $10^6 / 20 = 5 \times 10^4 \text{ s}$



一个实例（续）

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。读取中间文件花费的时间(同写中间文件一样)需 $5 \times 10^4 \text{s}$
- 满足条件的元组假设仅50个，均可放在内存



一个实例（续）



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
- 第一种情况下执行查询的总时间 $\approx 105 + 2 \times 5 \times 10^4 \approx 10^5 \text{s}$
- 所有内存处理时间均忽略不计

一个实例（续）

❖ 二、第二种情况

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

1. 计算自然连接

- 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2100块花费105 s
- 自然连接的结果比第一种情况大大减少，为 10^4 个
- 写出这些元组时间为 $10^4/10/20=50s$ ，为第一种情况的千分之一

2. 读取中间文件块，执行选择运算，花费时间也为50s。

3. 把第2步结果投影输出。

第二种情况总的执行时间 $\approx 105+50+50 \approx 205s$



一个实例（续）

❖ 三、第三种情况



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

1. 先对**SC**表作选择运算，只需读一遍**SC**表，存取100块，花费时间为**5s**，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，花费时间为**5s**。
3. 把连接结果投影输出

第三种情况总的执行时间 $\approx 5 + 5 \approx 10s$

一个实例（续）



- ❖ 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总共3~4块
- ❖ 若Student表在Sno上也有索引
 - 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
 - 读取Student表的块数也可大大减少
- ❖ 总的存取时间将进一步减少到数秒

一个实例（续）



❖ 把代数表达式 Q_1 变换为 Q_2 、 Q_3 ,

- 即有选择和连接操作时，先做选择操作，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表的选择操作算法有全表扫描和索引扫描2种方法，经过初步估算，索引扫描方法较优
- 对于Student和SC表的连接，利用Student表上的索引，采用index join代价也较小，这就是物理优化



第九章 关系系统及其查询优化



9.1 关系数据库系统的查询处理

9.2 关系数据库系统的查询优化

9.3 代数优化

9.4 物理优化

9.5 小 结

9.3 代数优化



❖ 9.3.1 关系代数表达式等价变换规则

❖ 9.3.2 查询树的启发式优化

9.3.1 关系代数表达式等价变换规则

- ❖ 代数优化策略：通过对关系代数表达式的等价变换来提高查询效率
- ❖ 关系代数表达式的等价：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 $E_1 \equiv E_2$



关系代数表达式等价变换规则（续）



❖ 常用的等价变换规则：

1. 连接、笛卡尔积交换律

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

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

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

$$E_1 \bowtie_F E_2 \equiv E_2 \bowtie_F E_1$$

2. 连接、笛卡尔积的结合律

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

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

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

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

关系代数表达式等价变换规则（续）



3. 投影的串接定律

$$\pi_{A_1, A_2, \dots, A_n}(\pi_{B_1, B_2, \dots, B_m}(E)) \equiv \pi_{A_1, A_2, \dots, A_n}(E)$$

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

关系代数表达式等价变换规则（续）



5. 选择与投影操作的交换律

$$\sigma_F(\pi_{A_1, A_2, \dots, A_n}(E)) \equiv \pi_{A_1, A_2, \dots, A_n}(\sigma_F(E))$$

选择条件 F 只涉及属性 A_1, \dots, A_n 。

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

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

关系代数表达式等价变换规则（续）



6. 选择与笛卡尔积的交换律

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

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

它使部分选择在笛卡尔积前先做。

关系代数表达式等价变换规则（续）



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

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

F 只涉及 E_1 与 E_2 的公共属性

关系代数表达式等价变换规则（续）



10. 投影与笛卡尔积的分配律

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

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

11. 投影与并的分配律

设 E_1 和 E_2 有相同的属性名，则

$$\pi_{A_1, A_2, \dots, A_n} (E_1 \cup E_2) \equiv \pi_{A_1, A_2, \dots, A_n} (E_1) \cup \pi_{A_1, A_2, \dots, A_n} (E_2)$$

9.3 代数优化



❖ 9.3.1 关系代数表达式等价变换规则

❖ 9.3.2 查询树的启发式优化

9.3.2 查询树的启发式优化

❖ 典型的启发式规则:



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
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的这些运算以避免重复扫描关系

查询树的启发式优化（续）

3. 把投影同其前或其后的双目运算结合起来
4. 把某些选择同在它前面要执行的笛卡尔积结合起来成为一个连接运算
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



查询树的启发式优化（续）

- ❖ 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 $\sigma_{F1 \wedge F2 \wedge \dots \wedge Fn}(E)$ 变换为 $\sigma_{F1}(\sigma_{F2}(\dots(\sigma_{Fn}(E))\dots))$ 。

(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。



查询树的启发式优化（续）

(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

➤ 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



查询树的启发式优化（续）

(5) 把上述得到的语法树的内节点分组。每一双目运算(\times , \bowtie , \cup , $-$)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尔积(\times)，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

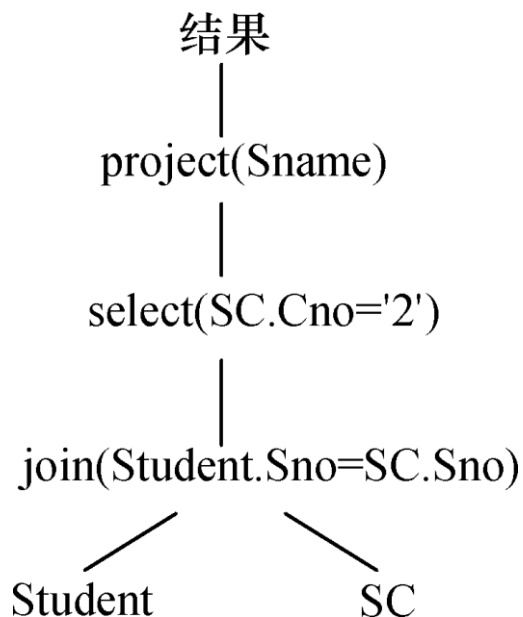


查询树的启发式优化（续）

[例4] 下面给出[例3]中 SQL语句的代数优化示例。



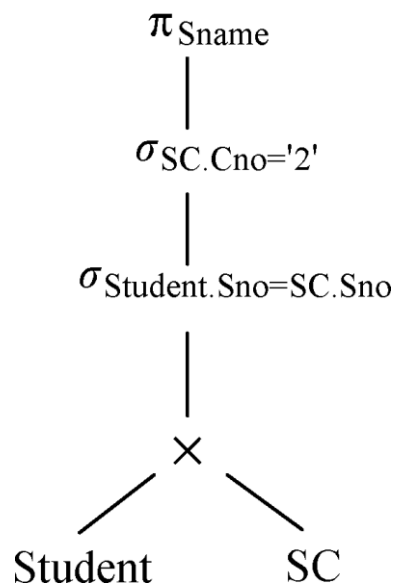
(1) 把SQL语句转换成查询树，如下图所示



查询树

查询树的启发式优化（续）

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

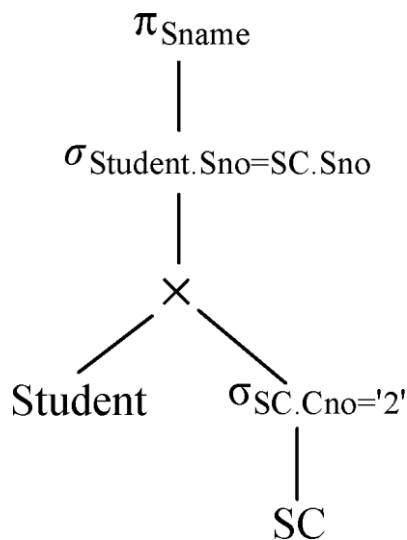


关系代数语法树

查询树的启发式优化（续）

(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，查询树便转换成下图所示的优化的查询树。这就是9.2.2节中 Q_3 的查询树表示



优化后的查询树



第九章 关系系统及其查询优化



9.1 关系数据库系统的查询处理

9.2 关系数据库系统的查询优化

9.3 代数优化

9.4 物理优化

9.5 小 结

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，不涉及底层的存取路径
- ❖ 对于一个查询语句有许多存取方案，它们的执行效率不同，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



物理优化（续）

❖ 选择的方法：

- 基于规则的启发式优化
- 基于代价估算的优化
- 两者结合的优化方法



9.4 物理优化



❖ 9.4.1 基于启发式规则的存取路径选择优化

❖ 9.4.2 基于代价的优化

9.4.1 基于启发式规则的存取路径选择优化



❖ 一、 选择操作的启发式规则

❖ 二、 连接操作的启发式规则

基于启发式规则的存取路径选择优化(续)



❖ 一、选择操作的启发式规则:

1. 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：

2. 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RDBMS会自动建立主码索引。

基于启发式规则的存取路径选择优化(续)

3. 对于选择条件是非主属性=值的查询，并且选择列上有索引

- 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



基于启发式规则的存取路径选择优化(续)

4. 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

- 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



基于启发式规则的存取路径选择优化(续)



5. 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
 - 优先采用组合索引扫描方法
- 如果某些属性上有一般的索引
 - 则可以用 [例1-C4] 中介绍的索引扫描方法
 - 否则使用全表顺序扫描。

6. 对于用OR连接的析取选择条件，一般使用全表顺序扫描

基于启发式规则的存取路径选择优化(续)



❖ 二、 连接操作的启发式规则：

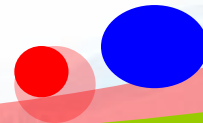
1. 如果2个表都已经按照连接属性排序
 - 选用排序-合并方法
2. 如果一个表在连接属性上有索引
 - 选用索引连接方法
3. 如果上面2个规则都不适用，其中一个表较小
 - 选用Hash join方法

基于启发式规则的存取路径选择优化(续)

4. 可以选用嵌套循环方法，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：

- 设连接表**R**与**S**分别占用的块数为 B_r 与 B_s
- 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 $B_r + (B_r / (K-1)) B_s$
- 显然应该选块数小的表作为外表



9.4 物理优化（续）



❖ 9.4.1 基于启发式规则的存取路径选择优化

❖ 9.4.2 基于代价的优化

9.4.2 基于代价的优化

❖ 启发式规则优化是定性的选择，适合解释执行的系统

- 解释执行的系统，优化开销包含在查询总开销之中

❖ 编译执行的系统中查询优化和查询执行是分开的

- 可以采用精细复杂一些的基于代价的优化方法





基于代价的优化（续）



❖ 一、统计信息

❖ 二、代价估算示例

基于代价的优化（续）



一、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，与数据库的状态密切相关
- ❖ 数据字典中存储的优化器需要的统计信息：
 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)

基于代价的优化（续）

2. 对基表的每个列

- 该列不同值的个数(m)
- 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, 则每个值的选择率=具有该值的元组数/ N
- 该列最大值
- 该列最小值
- 该列上是否已经建立了索引
- 索引类型(B+树索引、Hash索引、聚集索引)



基于代价的优化（续）

3. 对索引(如B+树索引)

- 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)



基于代价的优化（续）



二、代价估算示例

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 $\text{cost} = B$
- 如果选择条件是码=值，那么平均搜索代价 $\text{cost} = B/2$

基于代价的优化（续）



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)
 - 最坏的情况下，满足条件的元组可能会保存在不同的块上，此时， $cost=L+S$

基于代价的优化（续）

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



基于代价的优化（续）



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

$$\text{cost} = B_r + B_s / (K - 1) \quad B_r$$

➤ 如果需要把连接结果写回磁盘，

$$\text{cost} = B_r + B_s / (K - 1) \quad B_r + (F_{rs} * N_r * N_s) / M_{rs}$$

- 其中 F_{rs} 为连接选择性(join selectivity)，表示连接结果元组数的比例
- M_{rs} 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。

基于代价的优化（续）



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
$$\text{cost} = B_r + B_s + (F_{rs} * N_r * N_s) / M_{rs}。$$
- 如果必须对文件排序
 - 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
$$(2*B) + (2*B*\log_2 B)$$



第九章 关系系统及其查询优化



9.1 关系数据库系统的查询处理

9.2 关系数据库系统的查询优化

9.3 代数优化

9.4 物理优化

9.5 小 结

9.5 小 结

- ❖ 查询处理是RDBMS的核心，查询优化技术是查询处理的关键技术
- ❖ 本章讲解的优化方法
 - 启发式代数优化
 - 基于规则的存取路径优化
 - 基于代价的优化
- ❖ 本章的目的：希望读者掌握查询优化方法的概念和技术



小 结（续）

- ❖ 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RDBMS上
 - 应该找出RDBMS的优化规律，以写出适合RDBMS自动优化的SQL语句
- ❖ 对于RDBMS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

