

第10章：查询优化

Query Optimization

李东博

哈尔滨工业大学

计算学部

物联网与泛在智能研究中心

电子邮件: ldb@hit.edu.cn

2023年春

教学内容¹

1 Overview of Query Optimization

2 Logical Query Optimization

- Cost-based Query Optimization
- Plan Enumeration
- Relational Algebra Equivalences
- Cardinality Estimation
- Join Ordering

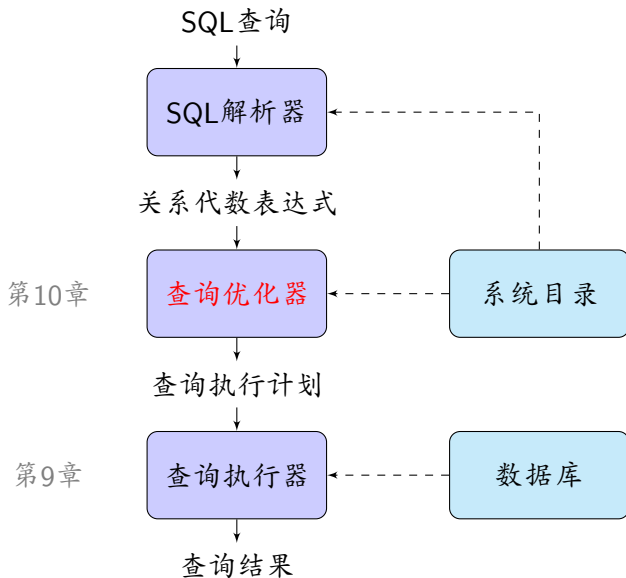
3 Physical Query Optimization

- Determining Selection Algorithms
- Determining Join Algorithms
- Determining Execution Model
- Physical Query Plan Enumeration

¹课件更新于2023年2月16日

Overview of Query Optimization

查询处理(Query Processing)的基本过程



查询优化(Query Optimization)

查询优化: 将一个关系代数表达式转换成一个可以快速执行的**查询执行计划(query execution plan)**的过程

- 查询优化是一个**NP难**问题

查询优化器(query optimizer): DBMS中负责查询优化的组件

- 查询优化器是DBMS中**最难设计**的组件之一

查询优化的两个阶段

- **逻辑查询优化(logical query optimization)**
- **物理查询优化(physical query optimization)**
- 实际上DBMS并不严格区分逻辑查询优化和物理查询优化

逻辑查询优化(Logical Query Optimization)

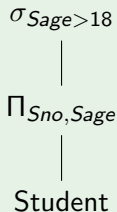
将一个关系代数表达式转换成另一个可以更快执行的关系代数表达式

- 逻辑查询计划(logical query plan): 关系代数表达式

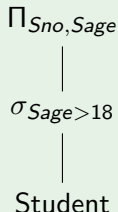
Example (逻辑查询优化)

SELECT Sno, Sage FROM Student WHERE Sage > 18;

初始逻辑查询计划



更好的逻辑查询计划



物理查询优化(Physical Query Optimization)

基于选定的逻辑查询计划，生成一个优化的物理查询计划(physical query plan)，使DBMS按照该物理查询计划执行可以快速得到查询结果

- 物理查询计划: 带有“如何执行”注释的关系代数表达式

Example (物理查询优化)

逻辑查询计划

$\Pi_{Sno, Sage}$

|

$\sigma_{Sage > 18}$

|

Student

\Rightarrow

物理查询计划

$\Pi_{Sno, Sage}$

|

$\sigma_{Sage > 18}$; 使用索引

|

Student

查询优化技术的产生

20世纪70年代，IBM System R最早实现了DBMS的查询优化器

- System R查询优化器的很多概念和技术仍被现在的DBMS使用

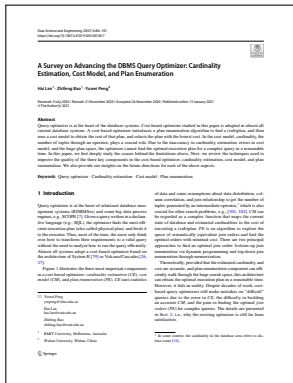


Patricia Selinger

世界上首个关系数据库管理系统System R的主要团队成员

基于代价的查询优化(cost-based query optimization)技术的创造者

DBMS的查询优化器通常需要40–50人年的研发



H. Lan, Z. Bao, Y. Peng. **A Survey on Advancing the DBMS Query Optimizer: Cardinality Estimation, Cost Model, and Plan Enumeration.** *Data Science and Engineering*, 6:86-101, 2021.

查询优化技术的未来

研究人员(包括我在内)希望借助机器学习(machine learning)技术来提高查询优化器的准确率和效率, 即AI4DB



X. Zhou, C. Chai, G. Li, J. Sun. **Database Meets Artificial Intelligence: A Survey.** *IEEE Transactions on Knowledge and Data Engineering*, 34(3):1096–1116, 2022.

Section 2.2 “Learning-based Database Optimization”

Logical Query Optimization

Logical Query Optimization

Cost-based Query Optimization

查询执行计划的代价(Cost)

查询执行计划的代价为该计划的所有操作的执行代价之和

- 代价是反映查询计划执行时间长短的数
- 查询计划的代价越低，其执行时间越短
- 查询计划的代价与实际执行时间没有直接的函数关系
- 比较不同查询的查询计划的代价没有意义

Example (PostgreSQL 查询优化器的代价常量)

- 一次顺序磁盘页读取的代价 $\text{seq_page_cost} = 1.0$
- 一次随机磁盘页读取的代价 $\text{random_page_cost} = 4.0$
- CPU处理一条元组的代价 $\text{cpu_tuple_cost} = 0.01$
- CPU处理一个索引项的代价 $\text{cpu_index_tuple_cost} = 0.005$
- CPU处理一个操作或函数的代价 $\text{cpu_operator_cost} = 0.0025$
- ...

基于代价的查询优化(Cost-based Query Optimization)

所有DBMS都采用基于代价的查询优化去找代价最低的查询执行计划

计划枚举(plan enumeration)

- 从初始查询计划 P 开始，生成与 P 等价的查询计划 P'

代价计算(cost evaluation)

- 对每个生成的查询计划，计算该计划的代价

Logical Query Optimization

Plan Enumeration

查询计划枚举(Query Plan Enumeration)

关系代数表达式的等价变换

- 将一个关系代数表达式转换为等价的关系代数表达式

连接顺序(join order)优化

- 确定连接操作的最优执行顺序

Logical Query Optimization

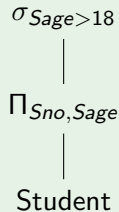
Relational Algebra Equivalences

等价关系代数表达式

如果两个关系代数表达式在任意数据库实例上的结果都相同，则这两个关系代数表达式等价(equivalent)

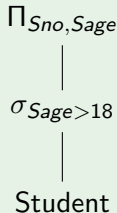
Example (等价关系代数表达式)

初始关系代数表达式树



≡

等价关系代数表达式树



关系代数表达式的等价变换规则

既满足交换律又满足结合律的关系代数操作

- $R \times S = S \times R$
 $(R \times S) \times T = R \times (S \times T)$
- $R \bowtie S = S \bowtie R$
 $(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$
- $R \cap S = S \cap R$
 $(R \cap S) \cap T = R \cap (S \cap T)$
- $R \cup S = S \cup R$
 $(R \cup S) \cup T = R \cup (S \cup T)$

θ 连接满足交换律，但不满足结合律(见作业1)

- $(R(a, b) \bowtie_{R.b > S.b} S(b, c)) \bowtie_{a < d} T(c, d) \neq R(a, b) \bowtie_{R.b > S.b} (S(b, c) \bowtie_{a < d} T(c, d))$

有关选择的等价变换规则

The splitting laws

- $\sigma_{\theta_1 \wedge \theta_2}(R) = \sigma_{\theta_1}(\sigma_{\theta_2}(R)) = \sigma_{\theta_2}(\sigma_{\theta_1}(R))$
- $\sigma_{\theta_1 \vee \theta_2}(R) = \sigma_{\theta_1}(R) \cup \sigma_{\theta_2}(R)$

Pushing a selection through a union

- $\sigma_{\theta}(R \cup S) = \sigma_{\theta}(R) \cup \sigma_{\theta}(S)$

Pushing a selection through a difference

- $\sigma_{\theta}(R - S) = \sigma_{\theta}(R) - S = \sigma_{\theta}(R) - \sigma_{\theta}(S)$

Pushing a selection through an intersection

- $\sigma_{\theta}(R \cap S) = \sigma_{\theta}(R) \cap S = R \cap \sigma_{\theta}(S) = \sigma_{\theta}(R) \cap \sigma_{\theta}(S)$

有关选择的等价变换规则(续)

Pushing a selection through a product

- $\sigma_{\theta}(R \times S) = R \bowtie_{\theta} S$
- 如果 R 包含 θ 中使用的全部属性, 而 S 没有
 $\sigma_{\theta}(R \times S) = \sigma_{\theta}(R) \times S$

Pushing a selection through a theta-join

- $\sigma_{\theta_1}(R \bowtie_{\theta_2} S) = R \bowtie_{\theta_1 \wedge \theta_2} S$
- 如果 R 包含 θ_1 中使用的全部属性, 而 S 没有
 $\sigma_{\theta_1}(R \bowtie_{\theta_2} S) = \sigma_{\theta_1}(R) \bowtie_{\theta_2} S$

Pushing a selection through a natural join

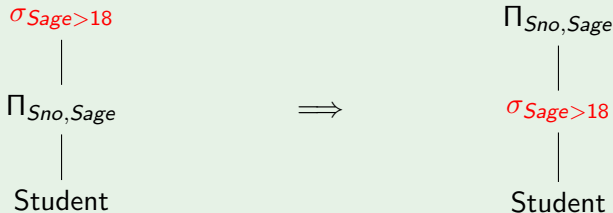
- 如果 R 包含 θ 中使用的全部属性, 而 S 没有
 $\sigma_{\theta}(R \bowtie S) = \sigma_{\theta}(R) \bowtie S$
- 如果 R 和 S 均包含 θ 中使用的全部属性
 $\sigma_{\theta}(R \bowtie S) = \sigma_{\theta}(R) \bowtie S = R \bowtie \sigma_{\theta}(S) = \sigma_{\theta}(R) \bowtie \sigma_{\theta}(S)$

选择下推(Selection Pushdown)

将关系代数表达式树中的选择操作**向下推(push down)**通常可以提高查询的执行效率

- 选择下推可以尽早过滤掉与结果无关的元组

Example (选择下推)

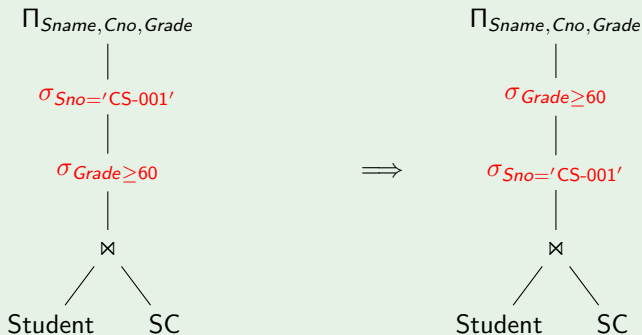


选择下推(续)

选择度高的选择操作优先做

- 选择度(selectivity): 满足选择条件的元组所占的比例(比例越低, 选择度越高)
- 尽早过滤掉更多与结果无关的元组

Example (选择下推)

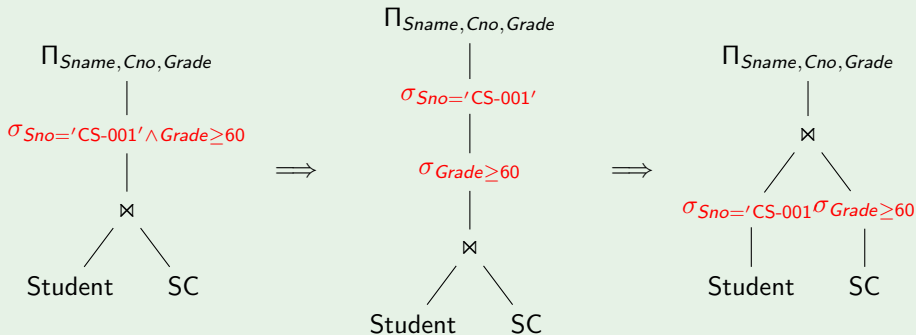


选择下推(续)

将复杂的选择条件进行分解，然后再进行选择下推

- $\sigma_{\theta_1 \wedge \theta_2}(R) = \sigma_{\theta_1}(\sigma_{\theta_2}(R))$

Example (选择下推)

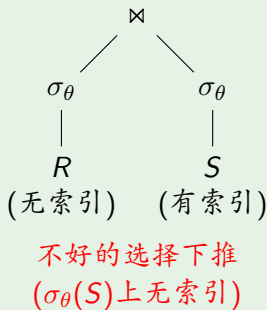
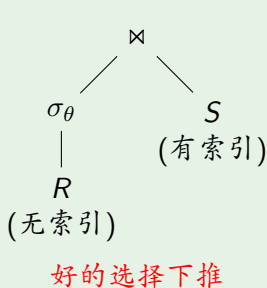
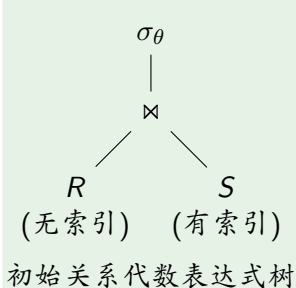


选择下推(续)

当选择操作可以向关系代数表达式树的多个分支下推时，需要考虑向哪个分支下推更合适

- 索引会影响选择下推的方案
- 选择操作的结果上没有索引

Example (索引对选择下推的影响)



有关投影的等价变换规则

Pushing a projection through a projection (the absorbing law)

- $\Pi_{L_1}(\Pi_{L_2}(R)) = \Pi_{L_1}(R) \quad (L_1 \subseteq L_2)$

Pushing a projection through a selection

- $\Pi_L(\sigma_\theta(R)) = \Pi_L(\sigma_\theta(\Pi_M(R)))$
M中既包含L中的属性，又包含 θ 中使用的属性

Pushing a projection through a union

- $\Pi_L(R \cup S) = \Pi_L(R) \cup \Pi_L(S)$

有关投影的等价变换规则(续)

Pushing a projection through a product

- $\Pi_L(R \times S) = \Pi_L(\Pi_M(R) \times \Pi_N(S))$

M 中包含既在 R 中又在 L 中的属性

N 中包含既在 S 中又在 L 中的属性

Pushing a projection through a natural join

- $\Pi_L(R \bowtie S) = \Pi_L(\Pi_M(R) \bowtie \Pi_N(S))$

M 中包含既在 R 中又在 L 中的属性, 以及 R 中的连接属性

N 中包含既在 S 中又在 L 中的属性, 以及 S 中的连接属性

Pushing a projection through a theta-join

- $\Pi_L(R \bowtie_{\theta} S) = \Pi_L(\Pi_M(R) \bowtie_{\theta} \Pi_N(S))$

M 中包含既在 R 中又在 L 中的属性, 以及 R 中的连接属性

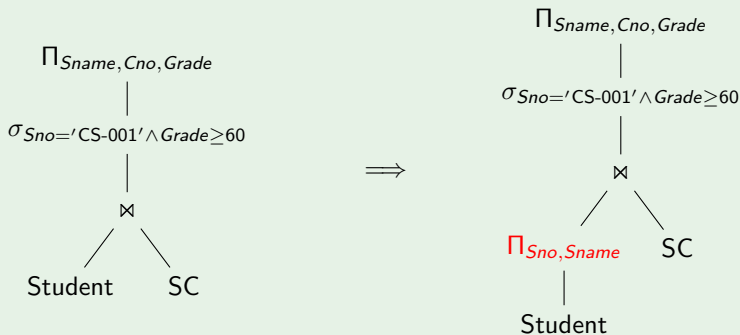
N 中包含既在 S 中又在 L 中的属性, 以及 S 中的连接属性

投影下推(Projection Pushdown)

将关系代数表达式树中的投影操作向下推通常可以提高查询执行效率

- 投影下推可以降低元组的大小

Example (投影下推)

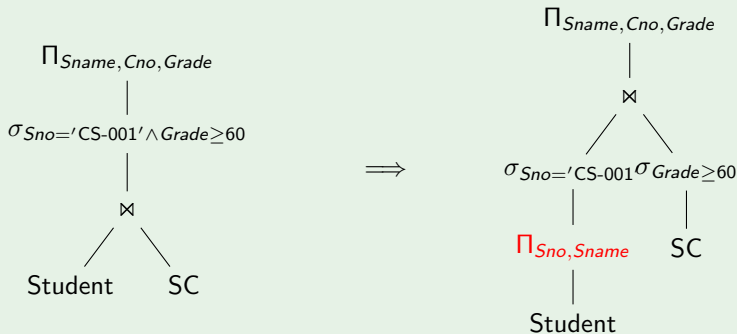


投影下推(续)

有些情况下, 投影操作下推会浪费查询优化的机会

- 投影操作的结果上没有索引

Example (投影下推)



- $\Pi_{Sno, Sname}(Student)$ 上没有索引
- $\sigma_{Sno=CS-001}$ 只能顺序扫描 $\Pi_{Sno, Sname}(Student)$

Logical Query Optimization

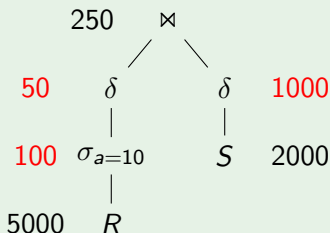
Cardinality Estimation

逻辑查询计划的代价模型(Cost Model)

逻辑查询计划的代价用执行计划过程中产生的中间结果的元组数来度量

- 查询计划的执行时间 $\leftarrow^{estimate}$ 执行查询计划时产生的I/O次数
- I/O次数 $\leftarrow^{estimate}$ 查询计划执行过程中产生的中间结果的大小
- 中间结果的大小 $\leftarrow^{estimate}$ 中间结果的基数(cardinality)，即元组数

Example (逻辑查询计划的代价)



该逻辑查询计划的代价 = 50 + 100 + 1000 = 1150

逻辑查询计划的代价模型vs物理查询计划的代价模型

物理查询计划中包含一些注释

- 关系代数操作的执行算法
- 中间结果的传递方式(物化、流水线)
- 中间结果是否排序

可以根据注释分析执行每个操作的I/O代价和CPU计算代价

- 物理查询计划的代价比逻辑查询计划的代价更可靠
- DBMS实际上直接枚举物理查询计划，而不区分逻辑查询优化和物理查询优化

无论如何，代价模型一定需要估计每个操作的结果元组数，即基数估计(cardinality estimation)

基数估计(Cardinality Estimation)

基数估计: 估计查询结果的元组数

查询优化器对基数估计方法的要求

- 准确
- 易计算
- 逻辑一致(logically consistent)

逻辑一致性(logical consistency)

- 单调性: 一个操作的输入越大, 操作结果的基数估计值越大
- 顺序无关性: 在多个关系上执行同一种满足交换律和结合律的操作时(如 \bowtie , \times , \cup , \cap), 最终结果的基数估计值与操作的执行顺序无关

基数估计所需的数据库统计信息

系统目录(system catalog)中记录着基数估计所需的数据库统计信息

- $T(R)$: 关系 R 的元组数
- $V(R, A)$: 关系 R 的属性集 A 的不同值的个数

基本假设(实际通常不成立)

- 关系中每个属性的取值均服从均匀分布
- 关系的所有属性相互独立

收集数据库统计信息

DBMS自动收集

用户手动收集

- Postgres/SQLite: **ANALYZE**
- Oracle/MySQL: **ANALYZE TABLE**
- SQL Server: **UPDATE STATISTICS**
- DB2: **RUNSTATS**

笛卡尔积操作的基数估计

$$T(R \times S) = T(R)T(S)$$

投影操作的基数估计

不含去重的投影操作

- $T(\Pi_L(R)) = T(R)$

含去重的投影操作

- $T(\Pi_L(R)) = V(R, L)$

选择操作的基数估计

$$S = \sigma_{A=c}(R)$$

- $T(S) = T(R)/V(R, A)$

$$S = \sigma_{A \neq c}(R)$$

- $T(S) = T(R)$ 或
- $T(S) = T(R) - T(R)/V(R, A)$

$$S = \sigma_{A > c}(R) \text{ or } S = \sigma_{A < c}(R)$$

- $T(S) = T(R)/3$ (《数据库系统实现(第2版)》) 或
- $T(S) = T(R)/2$ (《数据库系统概念(第6版)》)
- 如果系统目录中记录了 A 属性的最大值 $hi(A)$ 和最小值 $lo(A)$,
则 $T(\sigma_{A > c}(R)) = T(R) \frac{hi(A) - c}{hi(A) - lo(A)}$

选择操作的基数估计(续)

$$S = \sigma_{\theta_1 \wedge \theta_2}(R)$$

- $S = \sigma_{\theta_1 \wedge \theta_2}(R) = \sigma_{\theta_1}(\sigma_{\theta_2}(R)) = \sigma_{\theta_2}(\sigma_{\theta_1}(R))$
- $T(S) = T(R)f_1f_2$ (假设 θ_1 和 θ_2 独立)

$$f_i = \begin{cases} 1/V(R, A) & \text{if } \theta_i \text{ is of the form } A = c, \\ 1 - 1/V(R, A) & \text{if } \theta_i \text{ is of the form } A \neq c, \\ 1/3 & \text{if } \theta_i \text{ is of the form } A < c \text{ or } A > c, \end{cases}$$

$$S = \sigma_{\theta_1 \vee \theta_2}(R)$$

- $S = \sigma_{\theta_1}(R) \cup \sigma_{\theta_2}(R) = R - \sigma_{\neg\theta_1 \wedge \neg\theta_2}(R)$
- $T(S) = T(R)(1 - (1 - f_1)(1 - f_2))$

$$S = \sigma_{\neg\theta}(R)$$

- $S = R - \sigma_{\theta}(R)$
- $T(S) = T(R) - T(\sigma_{\theta}(R))$

二路自然连接(2-Way Natural Join)的基数估计

考虑两个关系 R 和 S 的自然连接 $R \bowtie S$

基本假设

- **连接属性值集合包含假设(Containment of Value Sets)**
对于连接属性 K ，如果 $V(R, K) \subseteq V(S, K)$ ，则 $R.K$ 的属性值集合是 $S.K$ 的属性值集合的子集
- **非连接属性值集合保留假设(Preservation of Value Sets)**
对于 R 中任意非连接属性 A ，有 $V(R \bowtie S, A) = V(R, A)$

二路自然连接的基数估计(续)

情况1: R 和 S 只有一个连接属性 Y

$$T(R \bowtie S) = \frac{T(R)T(S)}{\max(V(R, Y), V(S, Y))}$$

Example (二路连接的基数估计)

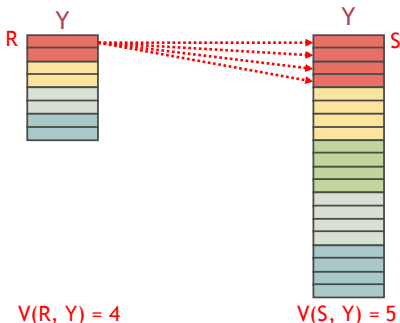
$R(a, b)$	$S(b, c)$	$U(c, d)$
$T(R) = 1000$	$T(S) = 2000$	$T(U) = 5000$
$V(R, b) = 20$	$V(S, b) = 50$	
	$V(S, c) = 100$	$V(U, c) = 500$

- $T((R \bowtie S) \bowtie U) = \frac{\frac{1000 \times 2000}{50} \times 5000}{500} = 400000$
- $T(R \bowtie (S \bowtie U)) = \frac{1000 \times \frac{2000 \times 5000}{500}}{50} = 400000$

证明

不失一般性，假设 $V(R, Y) \leq V(S, Y)$

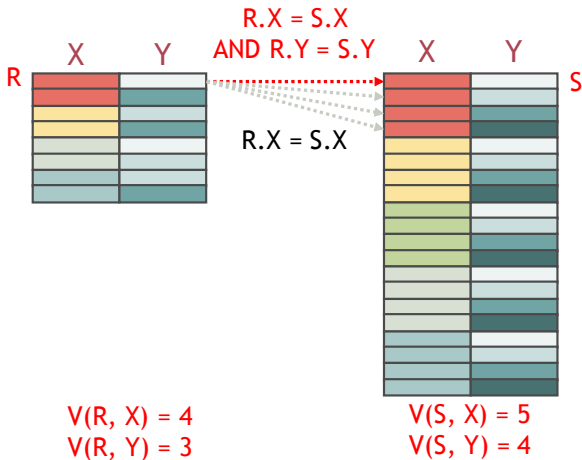
- 根据属性值包含假设， $R.Y$ 的属性值集合是 $S.Y$ 的属性值集合的子集，因此 R 的每个元组都可以和 S 的一些元组进行连接
- 根据属性值均匀分布假设， R 的每个元组都可以和 S 中 $\frac{T(S)}{V(S, Y)}$ 个元组进行连接
- 因此， $R \bowtie S$ 的结果包含 $\frac{T(R)T(S)}{V(S, Y)}$ 个元组



二路自然连接的基数估计(续)

情况2: R 和 S 有2个连接属性 X 和 Y

$$T(R \bowtie S) = \frac{T(R)T(S)}{\max(V(R, X), V(S, X)) \max(V(R, Y), V(S, Y))}$$



证明

设 $U = R \bowtie_{R.X=S.X} S$, 则 $T(R \bowtie S) = T(\sigma_{R.Y=S.Y}(U))$

- $T(U) = \frac{T(R)T(S)}{\max(V(R,X), V(S,X))}$
- 根据属性值保留假设,
有 $V(U, R.Y) = V(R, Y)$, $V(U, S.Y) = V(S, Y)$
- 不失一般性, 假设 $V(R, Y) \leq V(S, Y)$, 则 $V(U, R.Y) \leq V(U, S.Y)$
- 根据属性值包含假设, U 的 $R.Y$ 属性值集合是 U 的 $S.Y$ 属性值集合的子集
- 根据属性值均匀分布假设, U 中满足 $R.Y = S.Y$ 的元组占 $1/V(U, S.Y) = 1/V(S, Y)$
- 因此, $T(\sigma_{R.Y=S.Y}(U)) = \frac{T(R)T(S)}{\max(V(R,X), V(S,X)) \cdot V(S, Y)}$

二路自然连接的基数估计(续)

Example (二路连接的基数估计)

$R(a, b)$	$S(b, c)$	$U(c, d)$
$T(R) = 1000$	$T(S) = 2000$	$T(U) = 5000$
$V(R, b) = 20$	$V(S, b) = 50$	
	$V(S, c) = 100$	$V(U, c) = 500$

- $T((R \bowtie S) \bowtie U) = \frac{\frac{1000 \times 2000}{50} \times 5000}{500} = 400000$
- $T(R \bowtie (S \bowtie U)) = \frac{1000 \times \frac{2000 \times 5000}{500}}{50} = 400000$
- $T((R \bowtie U) \bowtie S) = \frac{(1000 \times 5000) \times 2000}{50 \times 500} = 400000$
- 基数估计结果与连接顺序无关

二路自然连接的基数估计(续)

情况3: R 和 S 有2个以上连接属性

证明留作课后习题

多路自然连接(Multi-way Natural Join)的基数估计

设 $S = R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n$

$$T(S) = \frac{T(R_1)T(R_2) \cdots T(R_n)}{\prod_{A \in \{\text{attributes appearing in more than two relations}\}} V(A)}$$

设 $R_{i_1}, R_{i_2}, \dots, R_{i_n}$ 是所有包含属性 A 的关系，
且 $V(R_{i_1}, A) \leq V(R_{i_2}, A) \leq \cdots \leq V(R_{i_n}, A)$

$$V(A) = V(R_{i_2}, A)V(R_{i_3}, A) \cdots V(R_{i_n}, A)$$

自然连接操作基数估计的逻辑一致性

无论是按不同的顺序进行多次二路连接，还是进行一次多路连接，上述基数估计方法得到的结果相同

Example (连接操作的基数估计)

$R(a, b)$	$S(b, c)$	$U(c, d)$
$T(R) = 1000$	$T(S) = 2000$	$T(U) = 5000$
$V(R, b) = 20$	$V(S, b) = 50$	
	$V(S, c) = 100$	$V(U, c) = 500$

- $T((R \bowtie S) \bowtie U) = \frac{\frac{1000 \times 2000}{50} \times 5000}{500} = 400000$
- $T(R \bowtie (S \bowtie U)) = \frac{1000 \times \frac{2000 \times 5000}{50}}{50} = 400000$
- $T((R \bowtie U) \bowtie S) = \frac{(1000 \times 5000) \times 2000}{50 \times 500} = 400000$
- $T(R \bowtie S \bowtie U) = \frac{1000 \times 5000 \times 2000}{50 \times 500} = 400000$

集合操作的基数估计

$$T(R \cup S) = \frac{1}{2}(\max(T(R), T(S)) + T(R) + T(S))$$

- 原因: $\max(T(R), T(S)) \leq T(R \cup S) \leq T(R) + T(S)$

$$T(R - S) = T(R) - T(S)/2$$

- 原因: $T(R) - T(S) \leq T(R - S) \leq T(R)$

$$T(R \cap S) = \min(T(R), T(S))/2$$

- 原因: $0 \leq T(R \cap S) \leq \min(T(R), T(S))$

$$T(R \cap S) = T(R \bowtie S)$$

- 原因: $R \cap S = R \bowtie S$

去重操作的基数估计

$$T(\delta(R)) = (T(R) + 1)/2 \approx T(R)/2$$

- 原因: $1 \leq T(\delta(R)) \leq T(R)$

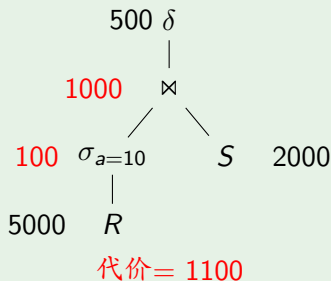
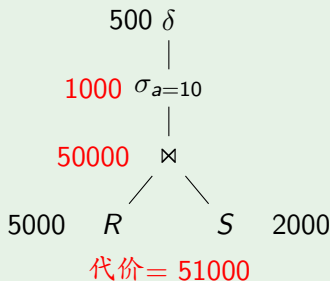
$$T(\delta(R)) = \min((T(R) + 1)/2, V(R, A_1)V(R, A_2) \cdots V(R, A_n))$$

- A_1, A_2, \dots, A_n 是 R 的属性
- 原因: $1 \leq T(\delta(R)) \leq V(R, A_1)V(R, A_2) \cdots V(R, A_n)$

逻辑查询计划的代价估计(Cost Estimation)

Example (逻辑查询计划的代价估计)

- $R(a, b) : T(R) = 5000, V(R, a) = 50, V(R, b) = 100$
- $S(b, c) : T(S) = 2000, V(S, b) = 200, V(S, c) = 100$



属性值分布的精确近似

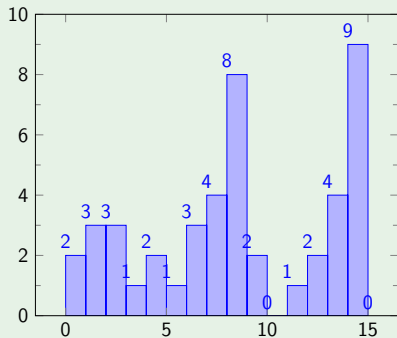
实际数据经常不满足均匀分布假设，导致基数估计的误差较大

Example (均匀分布假设对基数估计的影响)

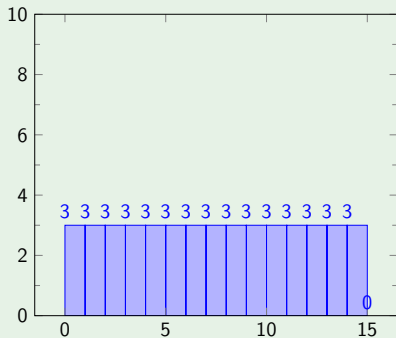
属性值 ≥ 13 的元组个数

- 实际情况: 13个
- 均匀分布假设: 6个

实际分布



均匀分布假设



直方图(Histogram)

DBMS使用直方图来记录属性值在不同区间内的出现频率，用于近似数据分布

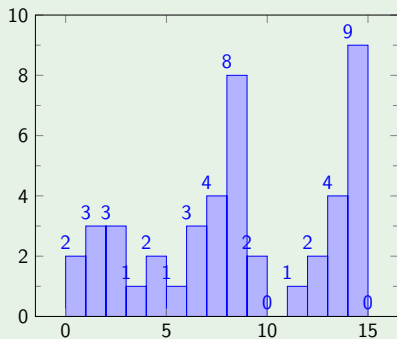
- 等宽直方图(equal-width histogram)
- 等高直方图(equal-height histogram)
- 压缩直方图(compressed histogram)

等宽直方图(Equal-Width Histogram)

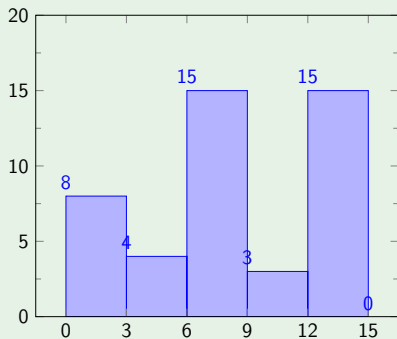
- 属性值各区间的宽度相同
- 各区间内属性值的出现次数不同

Example (等宽直方图)

实际分布



等宽直方图

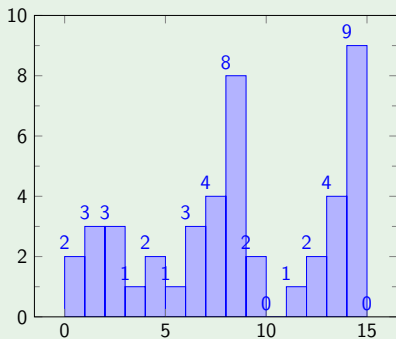


等高直方图(Equal-Height Histogram)

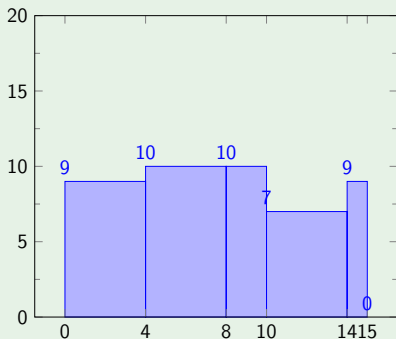
- 属性值各区间的宽度不同
- 各区间内属性值的出现次数基本相同

Example (等高直方图)

实际分布



等高直方图

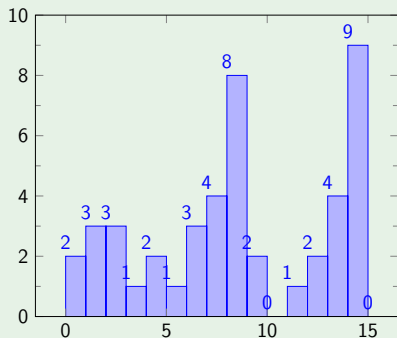


压缩直方图(Compressed Histogram)

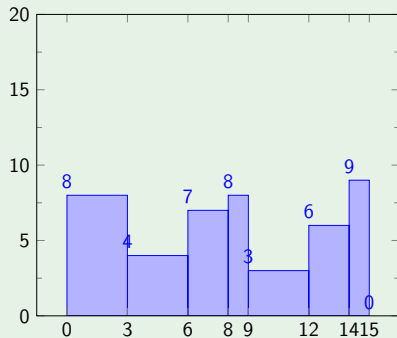
- 单独记录少量频繁出现的属性值
- 用等高或等宽直方图近似其他属性值的分布

Example (压缩直方图)

实际分布



压缩直方图



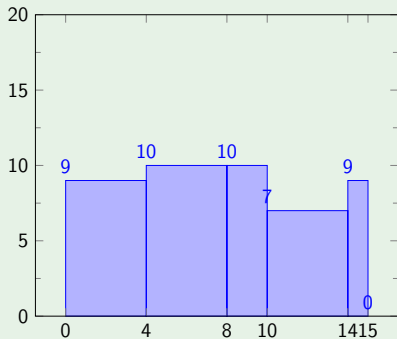
等高直方图vs等宽直方图

大多数DBMS使用等高直方图，而非等宽直方图

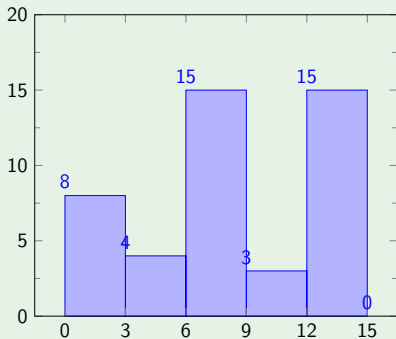
- 包含高频属性值的区间窄，对高频属性值的频率估计更准确(高频属性值对基数估计的误差影响更大)
- 区间边界由数据确定，无需用户确定

Example (等高直方图vs等宽直方图)

等高直方图



等宽直方图

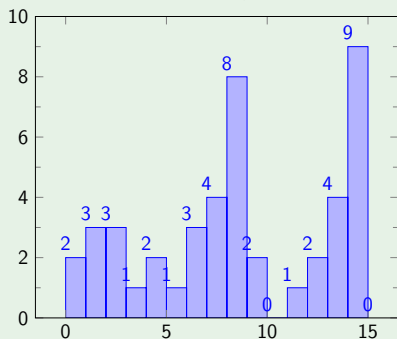


基于直方图的基数估计

假设: 直方图中每个区间内的属性值服从均匀分布

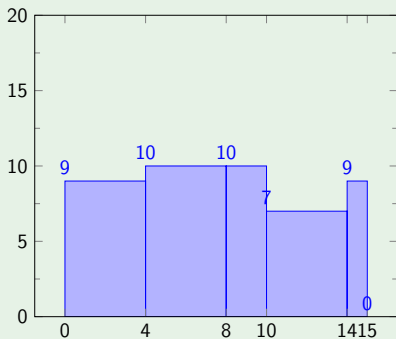
Example (基于直方图的基数估计)

属性值 > 9 的元组的个数
实际分布



真实基数 = $2 + 1 + 2 + 4 + 9 = 18$

等高直方图



估计基数 = $10/2 + 7 + 9 = 21$

基于直方图的连接结果基数估计

Example (基于直方图的连接结果基数估计)

已知关系 $R(a, b)$ 和 $S(b, c)$ 的属性 $R.b$ 和 $S.b$ 的直方图如下

Range	$R.b$	$S.b$
0-9	40	0
10-19	60	0
20-29	80	0
30-39	50	0
40-49	10	5
50-59	5	20
60-69	0	50
70-79	0	100
80-89	0	60
90-99	0	10
Total	245	245

- 使用直方图: $T(R \bowtie S) = 10 \times 5/10 + 5 \times 20/10 = 15$
- 不使用直方图: $T(R \bowtie S) = 245 \times 245/100 = 600$

属性相关性(Attribute Correlation)

当数据不满足属性独立性假设时，上述基数估计方法的误差较大

Example (属性独立性假设存在的问题)

设关系 $R(A, B)$ 中属性 A 和 B 满足约束条件 $B = A + 1$ ，已知 $T(R) = 1000$, $V(R, A) = 10$, $V(R, B) = 10$

- 属性独立:

$$T(\sigma_{A=1 \wedge B=2}(R)) = T(\sigma_{A=1}(\sigma_{B=2}(R))) = 1000 \times \frac{1}{10} \times \frac{1}{10} = 10$$

- 属性相关:

$$T(\sigma_{A=1 \wedge B=2}(R)) = T(\sigma_{A=1}(\sigma_{B=2}(R))) = 1000 \times \frac{1}{10} \times 1 = 100$$

刻画属性相关性的方法

- 多维直方图(multi-dimensional histogram)
- 概率图模型(probabilistic graphical model)
- 神经网络(neural network)
- 核密度估计(kernel density estimation, KDE)
- 和积网络(sum-product network, SPN)
- 自回归模型(autoregressive model)

Data Science and Engineering (2021) 6:86–101
https://doi.org/10.1007/s10339-020-00387-7



A Survey on Advancing the DBMS Query Optimizer: Cardinality Estimation, Cost Model, and Plan Enumeration

Hai Lan¹, Zhiling Bao², Yiyang Peng³

Received: 15 July 2020 / Revised: 27 November 2020 / Accepted: 26 November 2020 / Published online: 11 January 2021
© The Author(s) 2021

Abstract

Query optimization is at the heart of the database systems. Cost-based optimizer studied in this paper is adopted to address all current database systems. A cost-based optimizer simultaneously performs a plan enumeration algorithm to find a candidate plan, and then uses a cost model to evaluate the cost of that plan, and selects the plan with the lowest cost. In the cost model, cardinality, the number of tuples through an operator, plays a crucial role. Due to the inaccuracy in cardinality estimation, errors in cost model, and the large plan space, the optimizer cannot find the optimal execution plan for a complex query in a reasonable time. In this paper, we first deeply study the current database optimization issues. Next, we review the techniques used to improve the quality of the three key components in the cost-based optimizer: cardinality estimation, cost model, and plan enumeration. We also provide our thoughts on the future directions for each of the three main aspects.

Keywords Query optimizer · Cardinality estimation · Cost model · Plan enumeration

1 Introduction

Query optimization is at the heart of relational database management systems (RDBMSs) and some big data processing engines, e.g., Hadoop [1]. Given a query written in a declarative language (e.g., SQL), the optimizer finds the most efficient execution plan (also called physical plan) and feeds it to the executor. Then, based on the plan, the users only think about how to transform their requirements to a valid query without the need to realize how to run the query efficiently. Almost all systems adopt a cost-based optimizer based on the architecture of System R [2] (e.g., Volcano [3] and Oracle [4]).

Figure 1 illustrates the three most important components in a cost-based optimizer: cardinality estimation (CE), cost model (CM), and plan enumeration (PE). CE uses cardinality

of data and some assumptions about data distribution, column correlation, and join relationship to get the number of tuples generated by an intermediate operator¹, which is also used for the other search problems, e.g., [5, 6]. CE can be regarded as a complex function that maps the current state of database and estimated cardinalities to the cost of executing a join plan. PE is an algorithm to explore the space of potentially optimal join orders and find the optimal orders with minimal cost. There are two principal approaches to find an optimal join order: bottom-up join enumeration via dynamic programming and top-down join enumeration through enumeration.

Theoretically, provided that the estimated cardinality and cost are accurate, and plan enumeration component can efficiently walk through the large search space, this architecture can obtain the optimal execution plan in a reasonable time. However, it fails in reality. Despite decades of work, cost-based query optimizers still make mistakes on “difficult” queries due to the errors in CE, the difficulty in building an accurate CM, and the gaps in finding the optimal join orders (PE) for complex queries. The details are presented in Sect. 2, i.e., why the existing optimizers can still be from satisfaction.

¹ In some systems, the cardinality in the database can refer to the row count [10].

H. Lan, Z. Bao, Y. Peng. **A Survey on Advancing the DBMS Query Optimizer: Cardinality Estimation, Cost Model, and Plan Enumeration.** *Data Science and Engineering*, 6:86-101, 2021.

Section 3 “Cardinality Estimation”

Logical Query Optimization

Join Ordering

连接顺序优化(Join Ordering)

优化连接操作的执行顺序对于提高查询执行效率至关重要

- 连接操作的执行代价高
- 在数据库上执行几十个关系的连接很常见，如联机分析处理(online analytical processing, OLAP)任务
- 不同的连接顺序的执行代价差异巨大

强制连接执行顺序

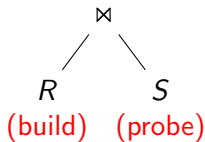
将PostgreSQL配置参数`join_collapse_limit`置为1可以阻止查询优化器对查询语句中的关系连接顺序进行优化(外连接除外)

连接关系的角色(Role)

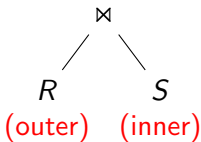
尽管连接操作满足交换律，但在查询计划中连接操作的输入关系具有不同的作用

- 在 $R \bowtie S$ 中， R 是左关系(left relation)， S 是右关系(right relation)
- 如果使用一趟连接(one-pass join)，则左关系是构建关系(build relation)，右关系是探测关系(probe relation)
- 如果使用嵌套循环连接(nested-loop join)，则左关系是外关系(outer relation)，右关系是内关系(inner relation)
- 如果使用索引连接(index-based join)，则右关系是索引关系(indexed relation)

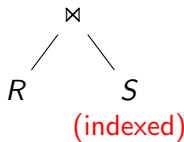
one-pass join



nested-loop join



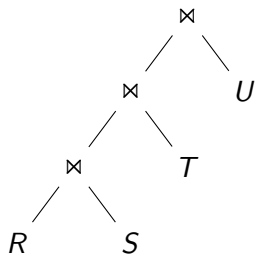
index-join



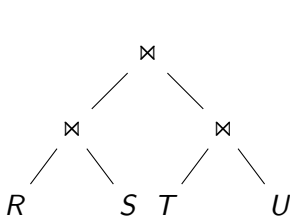
连接树(Join Tree)

一组关系上的连接操作的执行顺序可以用**连接树(join tree)**来表示

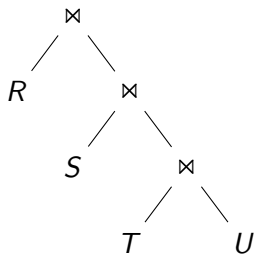
- **左深连接树(left-deep join tree)**: 只有一个关系是左关系, 其他关系都是右关系
- **右深连接树(right-deep join tree)**: 只有一个关系是右关系, 其他关系都是左关系
- **浓密树(bushy tree)**: 除左深连接树和右深连接树以外的其他连接树



左深连接树



浓密树



右深连接树

左深连接树(Left-Deep Join Tree)

System R的查询优化器只考虑左深连接树

优点1: 在给定关系上, 全部左深连接树的数量比全部连接树少得多

- $R_1 \bowtie R_2 \bowtie \dots \bowtie R_n$ 的全部左深连接树的数量为 $n!$
- $R_1 \bowtie R_2 \bowtie \dots \bowtie R_n$ 的全部连接树的数量为 $n!C_n$, 其中 C_n 表示含有 n 个叶子节点的树结构的数量(Catalan数)

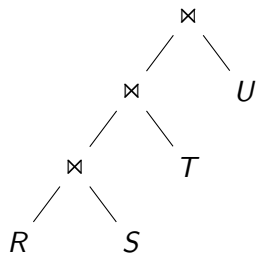
$$C_1 = 1,$$
$$C_n = \sum_{i=1}^{n-1} C_i C_{n-i} \quad \text{if } n > 1$$

优点2: 基于左深连接树的查询计划通常比基于其他类型连接树的查询计划的执行效率更高

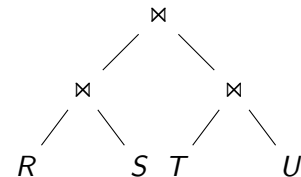
- 可能形成完全通畅的流水线(fully pipelined plans)

System R选择左深连接树的原因

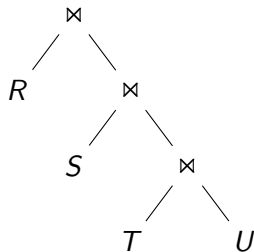
原因1: 如果下列查询计划全部使用一趟连接(one-pass join)算法, 那么在流水线查询执行模型(pipelining model)下, 左深连接树查询计划在任何时刻对内存的需求都比其他查询计划更低



缓冲池中只需存储 R 、 $R \bowtie S$ 或 $R \bowtie S \bowtie T$ 三者之一



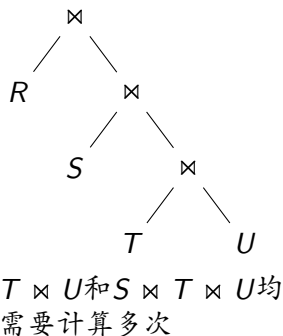
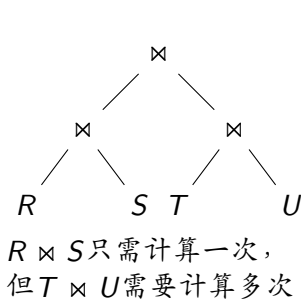
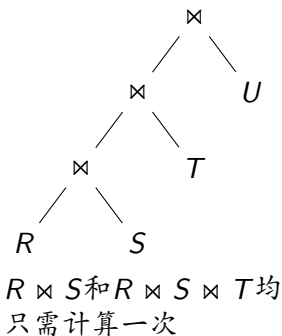
$R \bowtie S$ 和 T 必须同时在缓冲池中



R 、 S 和 T 必须同时在缓冲池中

System R选择左深连接树的原因(续)

原因2: 如果下列查询计划全部使用嵌套循环连接(nested-loop join)算法, 那么在流水线查询执行模型(pipelining model)下, 左深连接树查询计划不需要多次构建中间关系



基于动态规划的连接顺序优化方法

Example (连接顺序优化)

为 $R(a, b) \bowtie S(b, c) \bowtie T(c, d) \bowtie U(d, a)$ 选择最优的连接顺序

$R(a, b)$	$S(b, c)$	$T(c, d)$	$U(d, a)$
$T(R) = 1000$	$T(S) = 1000$	$T(T) = 1000$	$T(U) = 1000$
$V(R, a) = 100$			$V(U, a) = 50$
$V(R, b) = 200$	$V(S, b) = 100$		
	$V(S, c) = 500$	$V(T, c) = 20$	
		$V(T, d) = 50$	$V(U, d) = 1000$

两个关系上的最优连接顺序

	$\{R, S\}$	$\{R, T\}$	$\{R, U\}$	$\{S, T\}$	$\{S, U\}$	$\{T, U\}$
估计基数	5,000	1M	10,000	2,000	1M	1,000
估计代价	0	0	0	0	0	0
最优计划	$R \bowtie S$	$R \bowtie T$	$R \bowtie U$	$S \bowtie T$	$S \bowtie U$	$T \bowtie U$

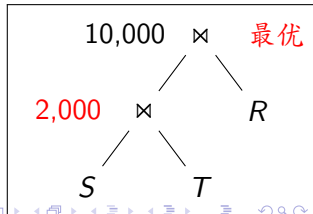
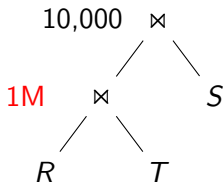
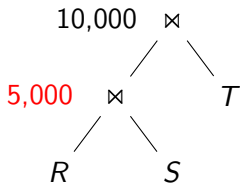
两个关系上的最优连接顺序

	$\{R, S\}$	$\{R, T\}$	$\{R, U\}$	$\{S, T\}$	$\{S, U\}$	$\{T, U\}$
估计基数	5,000	1M	10,000	2,000	1M	1,000
估计代价	0	0	0	0	0	0
最优计划	$R \bowtie S$	$R \bowtie T$	$R \bowtie U$	$S \bowtie T$	$S \bowtie U$	$T \bowtie U$

三个关系上的最优连接顺序

	$\{R, S, T\}$	$\{R, S, U\}$	$\{R, T, U\}$	$\{S, T, U\}$
估计基数	10,000	50,000	10,000	2,000
估计代价	2,000	5,000	1,000	1,000
最优计划	$(S \bowtie T) \bowtie R$	$(R \bowtie S) \bowtie U$	$(T \bowtie U) \bowtie R$	$(T \bowtie U) \bowtie S$

$R \bowtie S \bowtie T$ 的最优连接顺序是如何得到的?

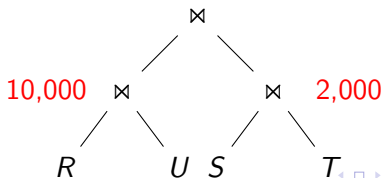
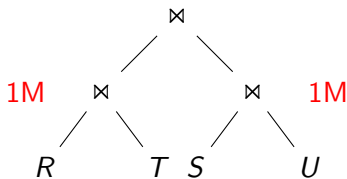
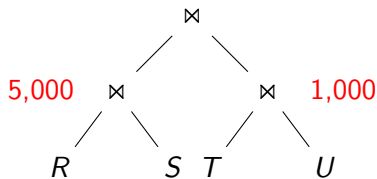


$R \bowtie S \bowtie T \bowtie U$ 的最优连接顺序是如何得到的?

情况1: 连接顺序形如 $(* \bowtie *) \bowtie (* \bowtie *)$

两个关系上的最优连接顺序

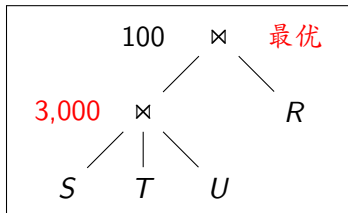
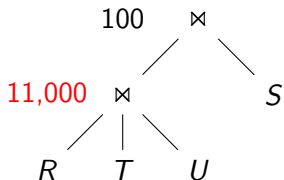
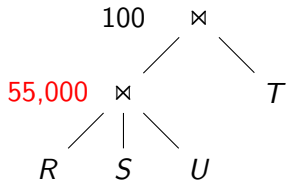
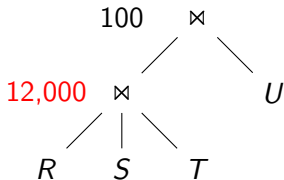
	$\{R, S\}$	$\{R, T\}$	$\{R, U\}$	$\{S, T\}$	$\{S, U\}$	$\{T, U\}$
估计基数	5,000	1M	10,000	2,000	1M	1,000
估计代价	0	0	0	0	0	0
最优计划	$R \bowtie S$	$R \bowtie T$	$R \bowtie U$	$S \bowtie T$	$S \bowtie U$	$T \bowtie U$



情况2: 连接顺序形如 $(* \bowtie * \bowtie *) \bowtie *$

三个关系上的最优连接顺序

	$\{R, S, T\}$	$\{R, S, U\}$	$\{R, T, U\}$	$\{S, T, U\}$
估计基数	10,000	50,000	10,000	2,000
估计代价	2,000	5,000	1,000	1,000
最优计划	$(S \bowtie T) \bowtie R$	$(R \bowtie S) \bowtie U$	$(T \bowtie U) \bowtie R$	$(T \bowtie U) \bowtie S$



Physical Query Optimization

物理查询优化(Physical Query Optimization)

物理查询计划枚举(physical query plan enumeration)

- **任务1:** 为逻辑查询计划中每个操作选择合适的物理**执行算法**
- **任务2:** 当把一个操作的结果传递给下一个操作时，为查询计划选择合适的**执行模型**

Physical Query Optimization

Determining Selection Algorithms

确定选择操作的物理执行算法

为选择操作确定物理执行算法的本质是为选择操作确定最高效的存取路径(access path)

- 存取路径1: 索引扫描(index scan)+过滤(filtering)
- 存取路径2: 多索引扫描+求交集(intersection)²
- 存取路径3: 仅用索引(index-only)
- 存取路径4: 顺序扫描(sequential scan)

²MySQL中称作索引合并(index merge)

索引扫描(Index Scan)+过滤(Filtering)

$\sigma_{\theta_1 \wedge \theta_2}(R)$, 其中 θ_1 和 θ_2 为两个选择条件

适用条件

- 关系 R 上的索引 I 支持条件 θ_1 上的查找

存取路径

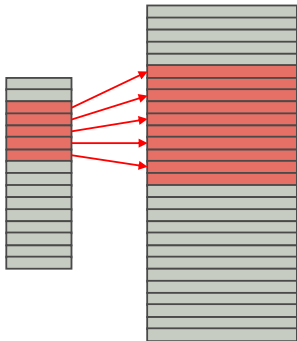
- 在索引 I 上查找满足条件 θ_1 的元组的ID
- 根据元组ID, 从 R 的文件中取出元组
- 使用条件 θ_2 对元组进行过滤

索引扫描+过滤(续)

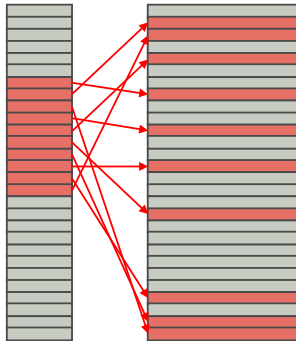
在以下情况下，“索引扫描+过滤”的存取路径非常有效

- 索引 I 是聚簇索引(clustered index)
- 索引 I 是非聚簇索引(nonclustered index)，但只有少量元组满足条件 θ_1

Clustered Index Database File



Nonclustered Index Database File



多索引扫描+求交集(Intersection)

$\sigma_{\theta_1 \wedge \theta_2}(R)$, 其中 θ_1 和 θ_2 为两个选择条件

适用条件

- 关系 R 上的索引 I_1 支持条件 θ_1 上的查找
- 关系 R 上的索引 I_2 支持条件 θ_2 上的查找

存取路径

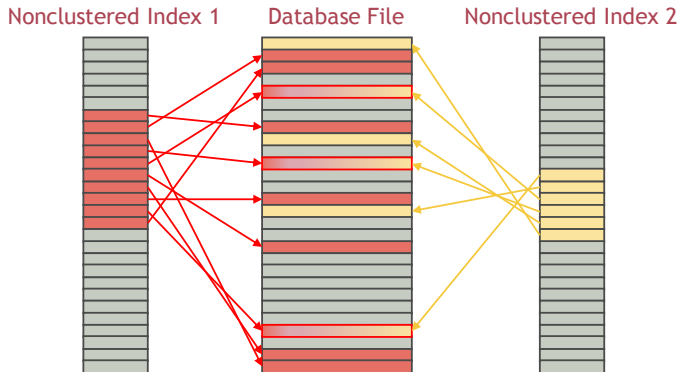
- 在索引 I_1 上查找满足条件 θ_1 的元组的ID
- 在索引 I_2 上查找满足条件 θ_2 的元组的ID
- 计算两个索引上得到的元组ID集合的交集
- 根据交集元组ID, 从 R 的文件中取出元组

该存取路径在MySQL中称作索引合并(index merge)

多索引扫描+求交集(续)

在以下情况下，“多索引扫描+求交集”的存取路径非常有效

- 索引 I_1 和 I_2 都是非聚簇索引
- 满足条件 θ_1 的元组和满足条件 θ_2 的元组均较多
- 但同时满足条件 θ_1 和 θ_2 的元组较少



仅用索引(Index-Only)

$$\sigma_{\theta}(R)$$

- θ : 选择条件
- L : 查询计划中该选择操作的后续操作所涉及的 R 的属性集合

适用条件

- 关系 R 上的索引 I 支持条件 θ 上的查找
- 索引 I 是覆盖索引(covering index)，其中包含 L 中的所有属性

存取路径

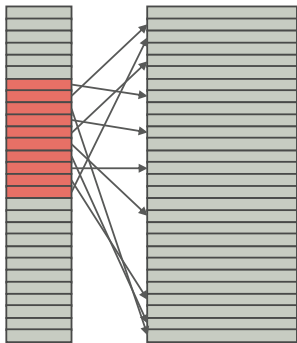
- 在索引 I 上查找满足条件 θ 的索引项
- 直接从索引项中取出该元组向 L 的投影

仅用索引(续)

在以下情况下，“仅用索引”的存取路径非常有效

- 关系 R 上的聚簇索引不支持选择条件 θ
- 索引 I 是覆盖索引

Nonclustered Index Database File



顺序扫描(Sequential Scan)

$\sigma_{\theta}(R)$, 其中 θ 为选择条件

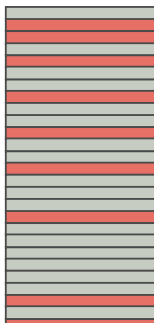
适用条件

- 关系 R 上没有索引可以支持条件 θ 上的查找

存取路径

- 顺序扫描 R 中每条元组
- 使用条件 θ 对元组进行过滤

Database File



Physical Query Optimization

Determining Join Algorithms

确定连接操作的执行算法

- 一趟连接(One-Pass Join)
- 索引连接(Index Join)
- 排序归并连接(Sort-Merge Join)
- 哈希连接(Hash Join)
- 嵌套循环连接(Nested-Loop Join)

一趟连接(One-Pass Join)

适用条件: 左关系可以全部读入缓冲池的可用页面

索引连接(Index Join)

适用条件

- 左关系较小
- 右关系在连接属性上建有索引

排序归并连接(Sort-Merge Join)

适用条件

- 至少有一个关系已经按连接属性排序
- 多个关系在相同连接属性上做多路连接也适合使用排序归并连接，如 $R(a, b) \bowtie S(a, c) \bowtie T(a, d)$

哈希连接(Hash Join)

适用条件: 在一趟连接、排序归并连接、索引连接都不适用的情况下，哈希连接总是好的选择

嵌套循环连接(Nested-Loop Join)

适用条件: 当内存缓冲区的可用页面特别少时, 可使用嵌套循环连接

Physical Query Optimization

Determining Execution Model

查询计划的执行模型

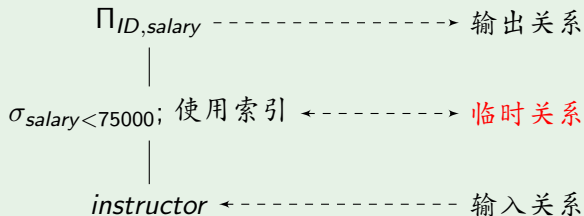
缓冲池的可用页数影响着查询计划执行模型的选择

- **模型1:** 物化执行(materialization)
- **模型2:** 流水线执行(pipelining)/火山模型(the volcano model)

物化执行(Materialization)

- 自底向上执行查询计划中的操作
- 每个中间操作的执行结果写入临时关系文件，作为后续操作的输入

Example (物化执行)



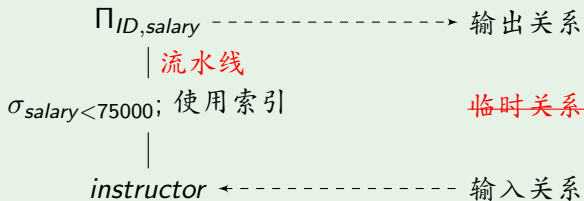
流水线执行(Piplining)/火山模型(The Volcano Model)

火山模型(The Volcano Model)将查询计划中若干操作组成流水线(pipeline)，一个操作的结果直接传给流水线中下一个操作

- 避免产生一些临时关系，避免了读写这些临时关系文件的I/O开销
- 用户能够尽早得到查询结果

几乎所有DBMS都使用流水线执行查询计划

Example (流水线执行)

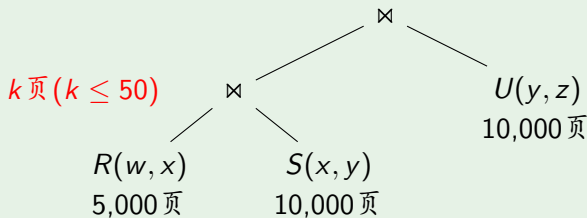


Physical Query Optimization

Physical Query Plan Enumeration

物理查询计划生成

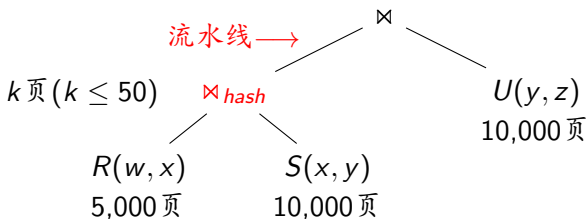
Example (物理查询计划生成1)



逻辑查询计划

- 缓冲池中有 $M = 101$ 个可用页
- R, S, U 上均无索引且未按连接属性排序
- $R \bowtie S$ 的结果占 $k \leq 50$ 页

第1步



使用哈希连接(hash join)执行 $R \bowtie S$

- 哈希分桶阶段使用101页内存

输入缓冲 1 页

100个桶

100 页

- 逐桶连接阶段使用51页内存(不计输出缓冲)

S 的缓冲 1 页

R 的桶

50 页

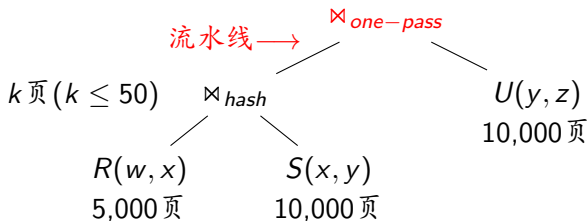
输出缓冲

50 页

- I/O代价: $3B(R) + 3B(S) = 45000$

- 因为 $k \leq 50$, 所以 $R \bowtie S$ 的结果可以保留在输出缓冲区中, 以流水线的形式输入给下一个连接操作

第2步



使用一趟连接(one-pass join)执行 $(R \bowtie S) \bowtie U$

- 一趟连接使用 $k + 1$ 页内存(不计输出缓冲)

U 的缓冲 1 页

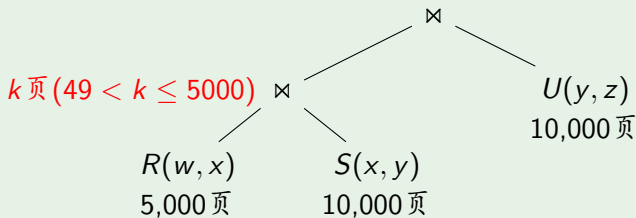
$R \bowtie S$ 的结果 k 页(已在缓冲池中)

输出缓冲 $100 - k$ 页

- I/O 代价: $B(U) = 10000$ ($R \bowtie S$ 的结果已在内存中, 无需 I/O)

物理查询计划生成

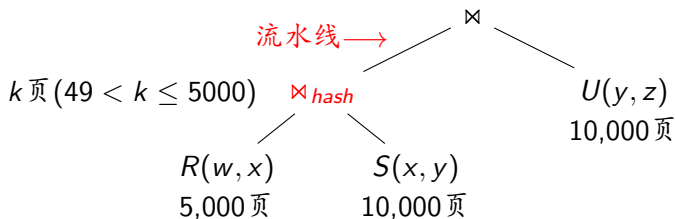
Example (物理查询计划生成2)



逻辑查询计划

- 缓冲池中有 $M = 101$ 个可用页
- R, S, U 上均无索引且未按连接属性排序
- $R \bowtie S$ 的结果占 $49 < k \leq 5000$ 块

第1步

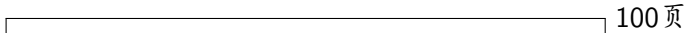


使用哈希连接(hash join)执行 $R \bowtie S$

- 哈希分桶阶段使用101页内存

输入缓冲 ☐ 1 页

100 个桶



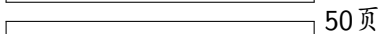
- 逐桶连接阶段使用51页内存(不计输出缓冲)

S 的缓冲 ☐ 1 页

R 的桶



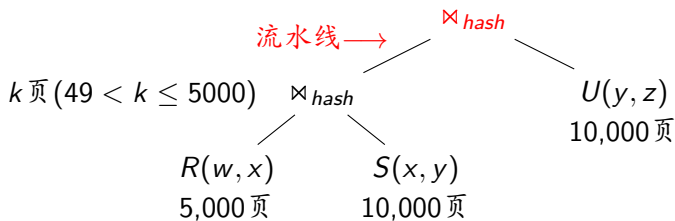
输出缓冲



- I/O代价: $3B(R) + 3B(S) = 45000$

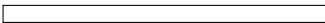
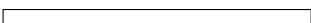
- 因为 $k > 50$, 所以 $R \bowtie S$ 的结果无法全部保留在输出缓冲区中, 但仍以流水线形式输入给下一个连接操作

第2步





使用哈希连接(hash join)执行 $(R \bowtie S) \bowtie U$

- $R \bowtie S$ 的结果哈希分桶阶段使用50页内存

执行 $R \bowtie S$  51 页
50 个桶  50 页

- U 的哈希分桶阶段使用51页内存

U 的缓冲  1 页
50 个桶  50 页

- 逐桶连接阶段使用101页内存(不计输出缓冲)

U 的缓冲  1 页
 $R \bowtie S$ 的桶  100 页

- I/O代价: $2B(R \bowtie S) + 3B(S) = 2k + 30000$

Summary

1 Overview of Query Optimization

2 Logical Query Optimization

- Cost-based Query Optimization
- Plan Enumeration
- Relational Algebra Equivalences
- Cardinality Estimation
- Join Ordering

3 Physical Query Optimization

- Determining Selection Algorithms
- Determining Join Algorithms
- Determining Execution Model
- Physical Query Plan Enumeration