Lecture 10: 查询处理

Database

Author: Forliage

Email: masterforliage@gmail.com

Date: June 5, 2025

College: 计算机科学与技术学院

Abstract

本讲笔记系统地介绍了关系型数据库中查询处理的核心原理与实现方法，内容涵盖三个主要方面：首先，对查询处理的基本步骤进行了概述，包括解析与翻译、优化与评估三个阶段；在此基础上，说明了如何通过等价关系代数表达式转换及成本估算来选择最优评估计划，以降低磁盘 I/O 成本，重点讨论了采用缓冲区大小和块传输次数进行估算的方法（如 $b \times t_T + S \times t_S$）以及优化时的最优/最差情况分析；其次，详细剖析了选择操作的多种实现算法，包括文件扫描（线性扫描和基于排序的二分查找）、利用主索引和辅助索引进行等值及范围选择的策略，以及复杂谓词（合取、析取与否定）的处理方式，分别给出了各种选择方法的成本公式与应用场景；最后，深入讲解了常见的连接操作算法，如嵌套循环连接（及其带块缓冲的改进）、索引嵌套循环连接、排序归并连接（包括混合归并连接）和哈希连接，并对哈希连接的分区与匹配阶段进行了详细描述，分析了哈希分区数的选择、溢出处理（递归分区与块嵌套循环回退策略）及成本估算（包括分区阶段的顺序读写开销和匹配阶段的块读取与寻道次数），帮助读者理解如何在大规模数据环境下实现高效的等值连接操作。通过本讲笔记的学习，读者能够掌握查询执行引擎中的关键算法与优化思路，为进一步研究查询优化与性能调优奠定坚实基础
（该Abstract由ChatGPT-o4-mini-high生成）

Contents

1 概述 . . . . .

1.1 查询处理的基本步骤 2

1.2 基本步骤：优化 3

2 查询成本的度量 . . . . .

3 选择操作 . . . . .

3.1 基本算法 4

3.2 使用索引和相等条件进行选择 5

3.3 涉及比较的选择 5

3.4 复杂选择的实现 5

4 连接操作 . . . . .

4.1 嵌套循环连接 7

4.2 块嵌套循环连接 7

4.3 索引嵌套循环连接 8

4.4 合并连接（排序归并连接） 8

4.5 哈希连接 9

4.5.1 Hash连接的基本思想 9

4.5.2 Hash-Join的详细步骤 10

4.5.3 Hash-Join优缺点 11

4.5.4 处理溢出（Overflow）的策略 11

4.5.5 Hash-Join成本估计 12

1 概述

1.1 查询处理的基本步骤

1.解析与转换

2.优化

3.评估

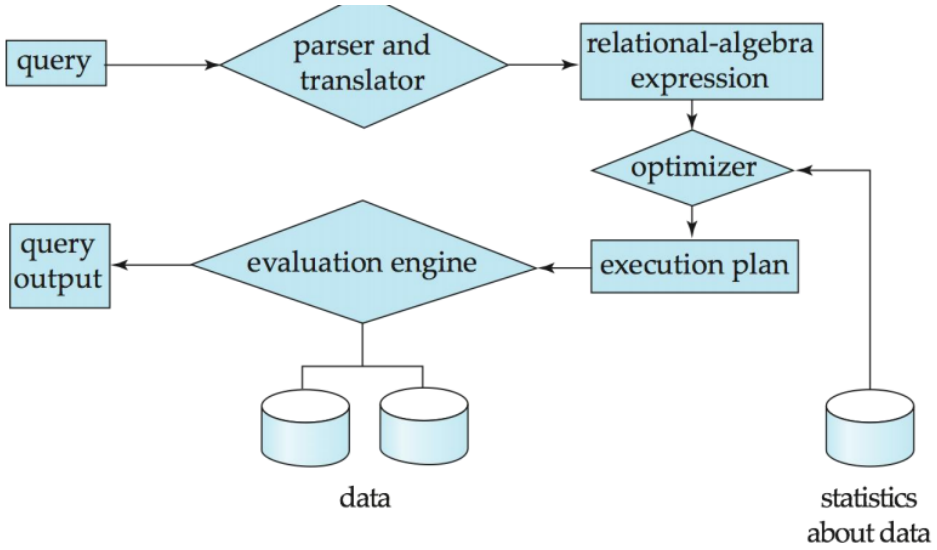

Figure 1. 基本步骤

解析与翻译（语法分析与翻译）与编译器对比：解析器检查语法，验证关系；将查询转换为其内部形式—扩展关系代数（ERA）。

评估：查询执行引擎获取查询评估计划，执行该计划，并返回查询的答案。

评估计划精确地定义了每个操作使用的算法，以及操作的执行如何协调。

优化—为什么？

对于给定的SQL查询，可能有许多等价的关系代数表达式。

例如：`select balance from account where balance > 2500.`

$$\sigma_{balance>2500}(\prod_{balance}(account))$$

$$\prod_{balance}(\sigma_{balance>2500}(account))$$

每个关系代数运算都可以使用几种不同的算法之一进行评估。

相应地，一个关系代数表达式可以通过多种方式进行评估。

指定详细评估策略的带注释表达式称为评估计划。

1.2 基本步骤：优化

(1)找出各种等价的关系代数表达式。

(2)一组指定详细评估策略的基本操作序列称为查询执行计划或查询评估计划。

(3)查询优化：在所有等效的评估计划中，选择成本最低的那个。成本考虑两个因素：成本取决于执行算法，还可以使用数据库目录中的统计信息来估算成本。

2 查询成本的度量

成本通常以回答查询的总耗时来衡量。许多因素会导致时间成本：磁盘访问+CPU+网络通信

通常，磁盘访问是主要主要成本，并且相对容易估算。估算时需考虑以下因素：

- 执行的寻道操作次数
- 读取的块数×平均块读取成本
- 写入的块数×平均块写入成本
- 注：写入一个块的成本大于读取一个块的成本，数据写入后被读回，以确保写入成功

为简单起见，我们仅使用从磁盘进行的块传输次数和寻道次数作为成本度量：

- t_T:传输一个块的时间($\approx 0.1ms$)
- t_S:一次寻道的时间($\approx 4ms$)
- b次块传输加上S次寻道的成本

$$\Longrightarrow b * t_T + S * t_S$$

为了简单起见，我们忽略CPU成本，但实际系统确实会考虑CPU成本。

我们的成本公式中不包括将最终结果写回磁盘的成本。

成本取决于主存中缓冲区的大小：

- 拥有更多内存可减少对磁盘访问的需求
- 用于缓冲的实际可用内存量取决于其它并发的操作系统进程，并且在实际执行前很难确定。
- 我们通常使用最坏情况估计(假设操作仅能使用所需的最小内存量)和最好情况估计。
- 所需数据可能已存在于缓冲区中，从而避免了磁盘I/O。但在成本估算时很难考虑到这一点。

3 选择操作

3.1 基本算法

文件扫描—定位并检索满足选择条件的记录的搜索算法，不使用索引。

算法A1（线性搜索）：扫描每个文件块并测试所有记录，查看它们是否满足选择条件。

- 成本估算= b_r块传输+1寻道：b_r表示包含来自关系r的记录的块数
- 如果选择是基于键属性，则在找到记录时可以停止：成本= $(b_r/2)$块传输+1寻道
- 线性搜索无论：选择条件、文件中记录的排序、索引的可用性

注意：由于数据并非连续存储，二分查找通常没有意义。除非有可用的索引，并且二分查找比索引查找需要更多的查找操作。

算法A2（二分查找）：若选择操作是对文件排序所依据的属性进行相等比较，则适用。

- 假设关系的块是连续存储的
- 成本= $\lceil \log_2(b_r) \rceil$块传输+$\lceil \log_2(b_r) \rceil$通过对块进行二分查找来定位第一个元组的寻道成本（时间成本= $\lceil \log_2(b_r) \rceil * (t_S + t_T)$）
- 如果选择操作不是针对键属性，则需要加上包含满足选择条件记录的块的数量：块传输= $\lceil \log_2(b_r) \rceil + \lceil sc(A,r)/f_r \rceil - 1$，其中$sc(A,r)$是满足选择条件的记录数，$f_r$是每个块的记录数

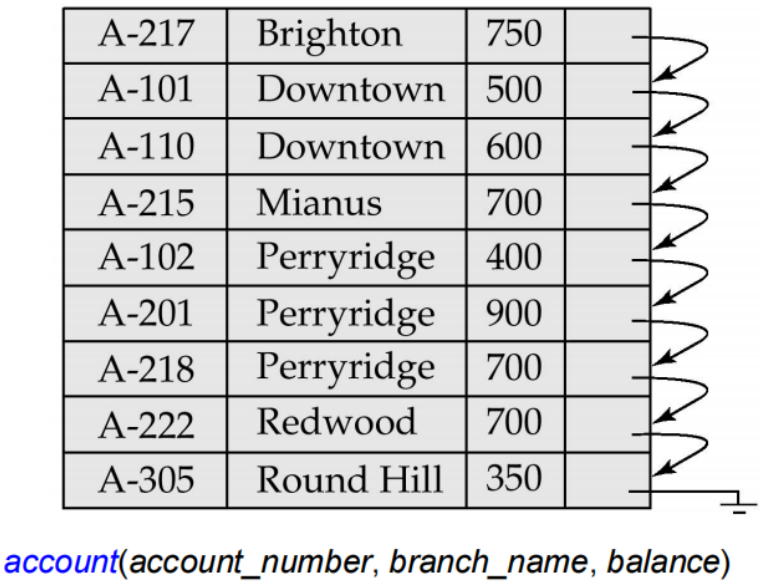

Figure 2. 例

假设：账户按照搜索键分行名称进行排序。`Select * from account where branch_name = 'Downtown';` 可能会返回100个元组。

3.2 使用索引和相等条件进行选择

索引扫描—使用索引的搜索算法：选择条件必须基于索引的搜索键。

算法A3（主索引，键上的相等条件）,检索满足相应相等条件的单条记录。成本= $(h_i + 1) * (t_S + t_T)$，其中h_i为索引树高。

算法A4（主索引，非键上的相等条件）检索多条记录。记录将位于连续的块上，设b =为包含匹配记录的块数，成本= $h_i * (t_S + t_T) + t_S + t_T * b$

算法A5（二级索引，非键上的相等条件）

- 如果搜索键是候选键，则检索单条记录：成本= $(h_i + 1) * (t_T + t_S)$
- 如果搜索键不是候选键，则检索多条记录。每个n匹配记录可能位于不同的块上，成本= $(h_i + n) * (t_T + t_S)$，可能代价非常高，可能比线性扫描更糟糕。

3.3 涉及比较的选择

比较符：$>, \geq, < . \leq, <>$，与等值比较不同之处在于选择范围大。

可以通过以下方式实现$\sigma_{A \leq V}(r)$或$\sigma_{A \geq V}(r)$形式的选择：

a 线性文件扫描；
b 二分查找（如A2）；
c 以下方式使用索引：

算法A6(基于主索引的比较)：

- 对于$\sigma_{A \geq V}(r)$，使用索引查找第一个元组$A \geq v$并从那里开始顺序扫描关系
- 对于$\sigma_{A \leq V}(r)$，只需顺序扫描关系直到找到第一个元组$A \geq v$；不使用索引

算法A7(基于辅助索引的比较)

- 对于$\sigma_{A \geq V}(r)$，使用索引查找第一个索引项$\geq v$并从那里开始顺序扫描索引，以找到指向记录的指针。
- 对于$\sigma_{A \leq V}(r)$，只需扫描索引的叶页以查找指向记录的指针，直到遇到第一个目录$A > v$
- 在这两种情况下，检索所指向的记录：每条记录都需要一个I/O；线性文件扫描可能成本更低。

3.4 复杂选择的实现

合取：$\sigma_{\theta_1} \wedge \sigma_{\theta_2} \wedge ... \sigma_{\theta_n}(r)$

算法A8（使用一个索引进行合取选择）：

- 选择θ_l的组合，以及算法A1到A7，使得θ_l的代价最小(第一步从n个条件$\theta_1...\theta_n$中选择代价最小的θ_i先执行，返回元组放内存，然后第二步对这些元组施行其它θ_j)
- 将元组提取到内存缓冲区后，对其进行其它条件测试。

算法A9（使用复合索引进行连接选择）：若可用，使用适当的复合（多键）索引。

算法10（通过标识符交集进行连接选择）：

- 需要带有记录指针的索引。
- 为每个条件使用相应的索引，并取所有的交集获取的记录指针集。
- 然后从文件中提取记录
- 如果某些条件没有合适的索引，则在内存中进行测试。

析取：$\sigma_{\theta_1} \vee \sigma_{\theta_2} \vee ...\sigma_{\theta_n}(r)$

算法10（通过标识符的并集进行析取选择）

- 如果所有条件都有可用索引，则适用。否则使用线性扫描。
- 为每个条件使用相应的索引，并取所有获得的记录指针集合的并集。
- 然后从文件中提取记录。

否定：$\sigma_{\neg\theta}(r)$

- 对文件使用线性扫描
- 如果只有极少数记录满足$\neg\theta$，并且索引适用于$\theta-$使用索引查找满足条件的记录并从文件中提取

4 连接操作

几种不同的连接实现算法：

- 嵌套循环连接
- 块嵌套循环连接
- 索引嵌套循环连接
- 合并连接
- 哈希连接

示例使用以下信息：

- 选课表的记录数：$n = 10,000$，学生表：$n = 5000$
- 选课表的块数：$b = 400$，学生表：$b = 100$

4.1 嵌套循环连接

为了计算θ连接$r \bowtie_\theta s$

```
for each tuple tr in r do begin
   for each tuple ts in s de begin
      test pair(tr, ts) to see if they satisfy the join condition
         theta
      if they do, add tr*ts to the result
    end
end
```

r称为连接的外关系，s称为连接的内关系。

不需要索引，可用于任何类型的连接条件。

开销大，因为它会检查两个关系的每一对元组。

在最坏的情况下，如果内存仅够容纳每个关系的一个块，则估计成本为$n_r * b_s + b_r$ block transfers+$n_r + b_r$ seeks

如果较小的关系能完全放入内存，则将其用作内关系：将成本降低至$b_r + b_s$次块传输和2次寻道

假设在最坏的内存可用情况下，成本估计为：

- 以学生表作为外关系：5000*400+100=2,000,100块传输，5000+100=5100查找
- 以外层关系作为输入：10000*100+400=1,000,400块传输和10,400次查找

如果较小的关系（学生）能完全放入内存，成本估算将为500次块传输。

4.2 块嵌套循环连接

是嵌套循环连接的一种变体，其中内关系的每个块都与外关系的每个块配对。

```
for each block Br of r de begin
   for each block Bs of s de begin
      for each tuple tr in Br do begin
         for each tuple ts in Bs do begin
            Check if (tr, ts) satisfy the join condition
            if they do, add tr*ts to the result
         end
      end
   end
end
```

最坏情况估计：$b_r * b_s + b_r$次块传输+$2 * b_r$次寻道。外层关系中的每个块都会读取一次内层关系中的s块。

最佳情况：$b_r + b_s$次块传输+2次寻道

对嵌套循环和块嵌套循环算法的改进：

- 在块嵌套循环中，使用$M-2$个磁盘块作为外层关系的块处理单元，其中$M=$为以块为单位的内存大小；使用剩余的两个块来缓冲内层关系和输出：成本$=\lceil b_r/(M-2)\rceil * b_s +$ b_r块传输+$2\lceil b_r/(M-2)\rceil$寻道
- 如果等值连接属性构成键或内部关系，则在首此匹配时停止内循环
- 交替向前和向后扫描内循环，以利用缓冲区中剩余的块（采用LRU），若可用，使用内部关系上的索引

4.3 索引嵌套循环连接

如果满足以下条件，索引查找可以替代文件扫描：

- 连接是等值连接或自然连接，并且
- 内部关系的连接属性上有可用的索引—可以仅为计算连接而构建的一个索引。

对于外部关系r中的每个元组t_r，使用索引在s中查找与元组t_r满足连接条件的元组。

最坏情况：缓冲区仅能容纳r的一个页面，并且对于r中的每个元组，我们都要在s上执行一次索引查找。

连接成本：$b_r(t_T + t_S) + n_r * c$。其中c是遍历索引并为一个元组获取所有匹配的s元组的成本，或者是r。c可以估计为使用连接条件对s进行单次选择的成本。

如果r和s的连接属性上都有索引，则使用元组较少的关系作为外部关系。

4.4 合并连接（排序归并连接）

在连接属性上对两个关系进行排序（如果尚未在连接属性上排序）。

合并已排序的关系以进行连接：

- 连接步骤类似于排序归并算法的合并阶段。
- 主要区别在于处理连接属性中的重复值—连接属性上具有相同值的每一对都必须匹配

仅可用于等值连接和自然连接。

每个块只需读取一次（假设连接属性的任何给定值的所有元组都能放入内存）。

因此，合并连接的成本为：$b_r + b_s$块传输+$\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil$寻道（+如果关系未排序，则为排序成本）

混合归并连接（混合归并连接）如果一个关系已排序，而另一个关系在连接属性上有辅助B+树索引。

- 将已排序的关系与B+树的叶节点条目合并
- 按未排序关系元组的地址对结果进行排序
- 按物理地址顺序扫描未排序关系，并与之前的结果合并，用实际元组替换地址（顺序扫描比随机查找更高效）

4.5 哈希连接

4.5.1 Hash连接的基本思想

1.使用场景：

Hash-Join 主要用于等值连接（Equi-Join）和自然连接（Natural Join），也就是连接条件形式为 r.某属性 = s.某属性的情形。

2.总流程overview：

假设要对关系（表）R 与 S 进行连接，且它们连接条件为相同的属性（例如 R.A = S.A）。Hash-Join 的基本思路可以分为两个阶段：

- 分区阶段：
  1. 选定一个哈希函数h，将R中每个元组tr(tuple)按$h(tr[A])$的哈希值划分到$R_0, R_1, ..., R_n$共n+1个分区中
  2. 同理，将S中每个元组ts按相同的哈希函数h，把ts[A]映射到$S_0, S_1, ..., S_n$
  3. 这样，所有R中哈希值为i的元组都进入同一个分区R_i，所有S中哈希值为i的元组都进入分区S_i。只需要保证：若R中某个元组tr和S中某个元组ts能够满足$tr[A] = ts[A]$，那么它们必然在同一个哈希分区i中。
- 匹配阶段：对每个分区$i = 0, ..., n$:
  1. 将较小的分区—假设是S_i（称为"build"输入）—全部加载到内存中，并基于连接属性再建立一个内存哈希表（使用另一种哈希函数h'，仅对这一小块数据进行索引）
  2. 然后对R_i（称为"probe"输入）中的每个元组tr，依次读取并在内存哈希表中查找所有匹配的ts。只要找到了，就输出$(tr \bowtie ts)$的连接结果。

这样，R_i仅与S_i做内部连接运算，无需与其它$S_j (j \neq i)$分区做比较，因为哈希保证了等值连接的所有候选都落在同一分区。

4.5.2 Hash-Join的详细步骤

假设R有b_r个磁盘块，S有b_s个磁盘块。同时假设可用内存能容纳M个数据页

1.选择分区数n以及哈希函数h。

通常，$n = \lceil b_s/M \rceil \times f$，其中$f$是调节因子，通常取1.2.这样保证每个$S_i$分区大概只有$M$个页面左右，可以一次性装入内存。 R_i分区不一定要完全装入内存，但S_i一定要能放进内存，才能构建哈希表。

2.第一遍—分区阶段：对S做哈希分区

```
// Reserve a memory output buffer for each partition output_buf[i]
for (each tuple ts in S read) do:
    idx = h(ts[A])
    write ts to output_buf[idx]
    // When output_buf[idx] is one page full, flush (write) it to
       the partitioned file S_idx on disk and clear the buffer
end for
// Flush all buffers that are not yet full and write the remaining
    tuples to the corresponding S_i file
```

3.分区阶段：对R做哈希分区

```
for (each tuple tr in R read) do:
    idx = h(tr[A])
    Write tr to output_buf[idx] of R
    // Write to R_idx file on disk when buffer is full
end for
//  Flush remaining buffers to each R_i partition file
```

4.都每个分区$i = 0, ..., n$：执行Build & probe

```
for i in 0...n do:
    // 4.1  Load S_i into memory first to build the hash table
    load the entire S_i partition into memory
    Create in-memory hash index HSi = <value $\to$ list of records
       > by hashing each tuple in S_i with join attribute A

    // 4.2 Probe R_i tuple by tuple.
    for (each tuple tr in R_i read) do
        key = tr[A]
        Find a list of all matching tuples for key in HSi L = HSi[
```

```
            key]
        for (each ts in L) do:
            Output connection results
        end for
    end for

    // 4.3 Release HSi, go to next partition i+1
end for
```

4.5.3 Hash-Join优缺点

1.优点：

大多数 IO 都是顺序写+顺序读，而非随机访问，磁盘吞吐量高。

只要分区合理，每个S_i足够小都能驻留内存，Probe 阶段查找非常快。

如果 R 与 S 都是一次性顺序扫描分区，再顺序读R_i，磁盘寻道次数少。

2.缺点：

需要一次完整地对 R、S 做哈希分区，要写出 n 个分区文件、再读回。对于大表，分区和合并会带来开销。

查询 $R \bowtie S$ 时，要先完成所有分区并写入磁盘，再做 Build & Probe。适合批量连接，不适合低延迟单次查询。

如果哈希分区不均匀（Skew），导致某些分区S_i超大，会出现“分区倾斜”，或者哈希表溢出，需要额外再递归拆分。

4.5.4 处理溢出（Overflow）的策略

1.可能原因

某个哈希分区 HSi 过大，无法整体装入内存。

原因可能是连接键的分布不均匀（很多元组的 key 落在同一个桶），或者哈希函数选择不当。

2.解决方案

- 递归分区（Overflow Resolution）:对那个超大的分区S_i，再用第二个哈希函数h_2 将其细分成更小的子分区。同时对对应的R_i也要用同一h_2做分区。这样便可保证“最终子分区”能够满足内存大小。
- 预先避免（Overflow Avoidance）:在最外层分区时，就尽量让 n 取得更大,使得每个分区平均更小。如果预计某些 key 会非常集中，可以先对 S 做一次非常细的分区，然后再

合并分区（分层合并）。

- 极端场景退化:如果分区之后仍然无法保证某个构建分区在内存里，那么可以对该分区执行块嵌套循环连接（Block Nested Loop Join），对重数据量进行笨拙但安全的直接连接。

4.5.5 Hash-Join成本估计

设不需要递归分区（所有S_i都能一次性装进 M 页内存），则：

分区阶段：

- 对S做一次顺序读和分区写$\rightarrow 2 \times b_s$个块传输
- 对R做一次顺序读和分区写$\rightarrow 2 \times b_r$个块传输加上为每个分区写空缓存页的开销（约n_h块）
- 总计：$3 \times (b_r + b_s) + 4 \times n_h$块传输

匹配阶段I/O：

- 对S_i直到装入内存已在分区阶段完成，分区读无需额外开销；
- 对每个分区R_i顺序读一次:b_r块
- 每次分区查找结束后，不需要再写出连接结束到磁盘。
- 再加上每读一个分区要一次磁盘寻道，共n_h次。