第15讲查询处理

(第15章)

关继宏教授

电子邮件:jhguan@tongji.edu.cn

计算机科学与技术系

同济大学

课程大纲

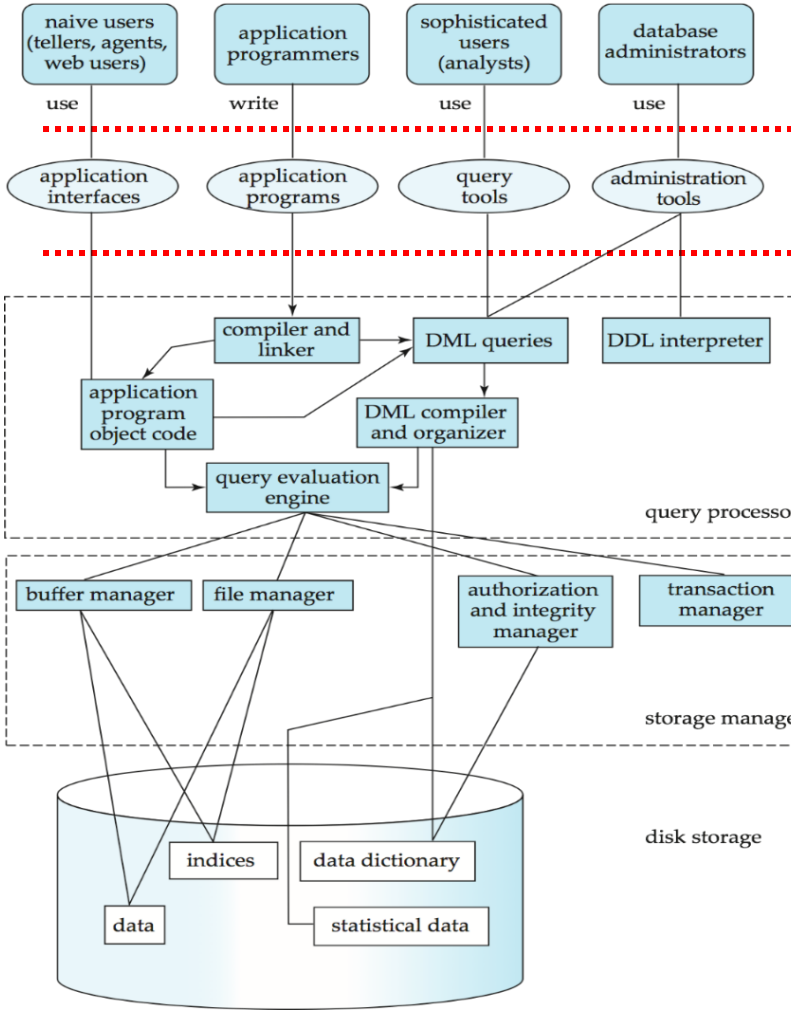
- **第0部分:概述**
  - Ch1:介绍
- **Part 1关系数据库**
  - Ch2:关系模型(数据模型，关系代数)
  - Ch3&4: SQL(结构化查询语言)
  - Ch5:高级SQL
- **第二部分数据库设计**
  - Ch6:基于E-R模型的数据库设计
  - Ch7:关系型数据库设计
- **第三部分:应用程序设计与开发**
  - Ch8:复杂数据类型
  - Ch9:应用开发
- **Part 4大数据分析**
  - Ch10:大数据
  - Ch11:数据分析
- **第5部分:数据存储和索引**
  - Ch12:物理存储系统
  - Ch13:数据存储结构
  - Ch14:索引

第6部分:查询处理与优化

- **Ch15:查询处理**
- Ch16:查询优化

- **第7部分事务管理**
  - Ch17:交易
  - Ch18:并发控制
  - Ch19:恢复系统
- **第8部分:并行和分布式数据库**
  - Ch20:数据库系统架构
  - Ch21-23:并行和分布式存储，查询处理和事务处理
- **第9部分**
  - **DB平台:OceanBase、MongoDB、Neo4J**

数据库系统结构



用户

应用程序/工具

数据库管理系统

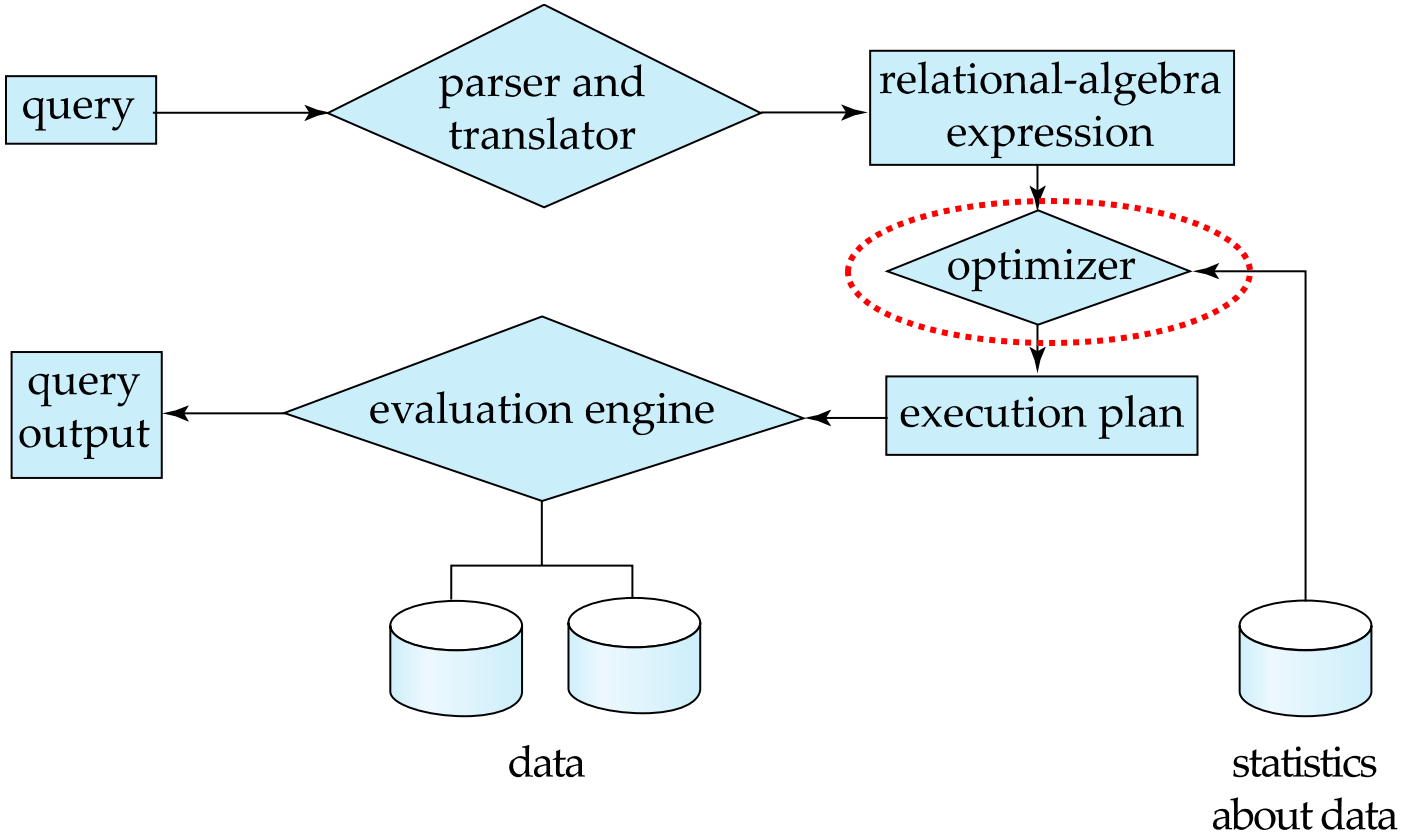
数据库

大纲

☞概述

- 查询成本的衡量标准
- 选择操作
- 排序
- 连接操作
- 其他操作
- 表达式求值

查询处理的基本步骤



1. 解析与翻译
2. 优化
3. 评价

查询处理的基本步骤

- **解析和翻译**
  - 将查询转换为内部形式，然后转换为关系代数
- **优化**
  - 生成最优执行计划()
- **执行**
  - 查询执行引擎执行求值计划，并返回查询的答案

查询优化

选择工资
从教练
哪里工资< 75000

给出对应的关系代数表达式

$$\sigma_{salary<75000}(\Pi_{salary}(instructor))$$

$$\Pi_{salary}(\sigma_{salary<75000}(instructor))$$

查询优化

- A relational algebra expression may have many **equivalent expressions**
- Annotated expression specifying detailed execution strategy is called an **execution-plan**
  - can use an index on *instructor* to find instructors with *salary* < *75000*, or
  - perform complete relation scan and discard instructors with *salary* ≥ *75000*

查询优化(续)

- **查询优化**
  - 在所有等效的评估方案中，选择成本最低的方案
  - 使用数据库目录中的统计信息估算成本
- **这节课**
  - 如何衡量查询成本
  - 评估关系代数运算的算法
  - 将单个运算的算法组合起来计算一个完整的表达式
- **下节课**
  - 如何找到一个估算成本最低的执行计划

大纲

- 概述

<s:1>查询成本的度量

- 选择操作
- 排序
- 连接操作
- 其他操作
- 表达式求值

查询成本的度量

- 成本通常以回答查询的总运行时间来衡量
  - 磁盘访问，CPU，甚至网络通信
- 通常，磁盘访问是主要的成本，也相对容易估算
- **磁盘访问是通过考虑来衡量的**
  - **寻道次数**
  - **读取的块数**
  - **写入的块数**
    - 写一个块的代价大于读一个块的代价
    - 数据写完后再回读，以确保写成功

查询成本的度量(续)

- For simplicity, use **the number of block transfers from disk** and **the number of seeks** as the cost measure
- Cost for b block transfers plus S seeks

$$b * t_T + S * t_S$$

  - t_T – time to transfer one block, ≈0.1ms
  - t_S – time for one seek, ≈4ms
- Cost also depends on **the size of the buffer** in main memory
  - Large buffer reduces the need for disk access
  - Often use **worst case estimates**, assuming only the minimum amount of buffer storage is available

大纲

- 概述
- 查询成本的度量

<s:1>选择操作

- 排序
- 连接操作
- 其他操作
- 表达式求值

选择操作

- **文件扫描(英文)**
  - 查找和检索满足选择条件的记录的搜索算法
- **索引扫描()**
  - 使用索引的搜索算法
  - 选择条件必须在索引的搜索关键字上

选择操作

- **Algorithm A1 (linear search，线性搜索)**
  - Cost estimate = b_r **block transfers + 1 seek (前提：文件块顺序存放)**
    - b_r denotes number of blocks containing records from relation r
  - If selection is on a key attribute, can stop on finding record
    - average **cost = (b_r /2) block transfers + 1 seek**
  - Linear search can be applied regardless of
    - selection condition or
    - ordering of records in the file, or
    - availability of indices

选择操作(control .)

- **A1'(二分查找)。**
  - 如果选择是对文件排序的属性进行相等比较，则适用。
  - 假设一个关系的块是连续存储的
  - 成本估算(需要扫描的磁盘块数量):
    - 通过对块进行二分搜索来定位第一个元组的成本
      - 最坏代价⌈log 2 (b r)⌉* (t t + t S)
    - 如果有多条记录满足选择
      - *将包含满足选择条件的记录的块数的传输代价相加*

使用索引的选择

- **A2 (primary index on candidate key, equality)**
  - Retrieve a single record that satisfies the corresponding equality condition
  - **Cost = $(h_i + 1) * (t_T + t_S)$** **(B^+-tree)**

- **A3 (primary index on non-key, equality) Retrieve multiple records**
  - Records will be on consecutive blocks
    - Let b = number of blocks containing matching records
  - **Cost = $h_i * (t_T + t_S) + t_S + t_T * b$**

使用索引进行选择(续)

- **A4 (equality on search-key of secondary index).**
  - Retrieve a single record if the search-key is a candidate key
    - **Cost = $(h_i + 1) * (t_T + t_S)$**
  - Retrieve multiple records if search-key is not a candidate key
    - Assume that n records satisfy the search condition
    - **Cost = $(h_i + n) * (t_T + t_S)$**
      - Can be very expensive!
    - **Each record may be on a different block**
      - one block access for each retrieved record

涉及比较的选择

- Implement selections of the form $\sigma_{A \leq V}(r)$ or $\sigma_{A \geq V}(r)$ by
  - using a **linear file scan** or **binary search**, or
  - using **indices** in the following ways:
- **A5 (primary index, comparison).**
  - Relation is sorted on *A*
  - For $\sigma_{A \geq V}(r)$ use index to find first tuple $\geq v$ and scan relation sequentially from there
  - For $\sigma_{A \leq V}(r)$ just scan relation sequentially till first tuple $> v$; do not use index

涉及比较的选择(续)

- **A6(二级索引，比较)。**
  - 对于A V (r)使用索引查找第一个索引条目V，从那里依次扫描索引，查找指向记录的指针。
  - 对于A≤V (r)只需扫描索引的页查找到记录的指针，直到第一个条目> V
  - 在任何一种情况下，检索被指向的记录
    - 每条记录都需要一个I/O
    - 如果要获取许多记录，线性文件扫描可能会更便宜!

选择操作成本估算

	算　法	开　销	原　因
A1	线性搜索	$t_s + b_r * t_T$	一次初始搜索加上 b_r 个块传输，b_r 表示在文件中的块数量
A1	线性搜索，码属性等值比较	平均情形 $t_s + (b_r/2) * t_T$	因为最多一条记录满足条件，所以只要找到所需的记录，扫描就可以终止。在最坏的情形下，仍需要 b_r 个块传输
A2	B⁺树主索引，码属性等值比较	$(h_i + 1) * (t_T + t_s)$	（其中 h_i 表示索引的高度）。索引查找穿越树的高度，再加上一次 *I/O* 来取记录；每个这样的 *I/O* 操作需要一次搜索和一次块传输
A3	B⁺树主索引，非码属性等值比较	$h_i * (t_T + t_S) + t_S + t_T * b$	树的每层一次搜索，第一个块一次搜索。b 是包含具有指定搜索码记录的块数。假定这些块是顺序存储（因为是主索引）的叶子块并且不需要额外搜索
A4	B⁺树辅助索引，码属性等值比较	$(h_i + 1) * (t_T + t_s)$	这种情形和主索引相似
A4	B⁺树辅助索引，非码属性等值比较	$(h_i + n) * (t_T + t_s)$	（其中 n 是所取记录数。）索引查找的代价和 *A3* 相似，但是每条记录可能在不同的块上，这需要每条记录一次搜索。如果 n 值比较大，代价可能会非常高
A5	B⁺树主索引，比较	$h_i * (t_T + t_s) + b * t_T$	和 *A3*，非码属性等值比较情形一样
A6	B⁺树辅助索引，比较	$(h_i + n) * (t_T + t_s)$	和 *A4*，非码属性等值比较情形一样

复杂选择的实现

- **Conjunction (合取):** $\sigma_{\theta 1 \wedge \theta 2 \wedge \cdots \wedge \theta n}(r)$
- **A7 (conjunctive selection using one index)**
  - Select a condition of θ_i and algorithms **A1** through **A6** that results in the least cost for $\sigma_{\theta i}(r)$
  - Test other conditions on the tuples after fetching them into memory buffer
- **A8 (conjunctive selection using multiple-key index)**
  - Use appropriate composite (multiple-key) index if available
- **A9 (conjunctive selection by intersection of identifiers)**
  - Requires indices with record pointers
  - Use corresponding index for each condition, and take intersection of all the obtained sets of record pointers
  - Then fetch records from file

复杂选择的实现(续)

- **Disjunction (析取):** $\sigma_{\theta 1 \vee \theta 2 \vee \cdots \vee \theta n}(r)$
- **A10 (disjunctive selection by union of identifiers).**
  - Applicable if all conditions have available indices
    - Otherwise use linear scan
  - Use the corresponding index for each condition, and take union of all the obtained sets of record pointers.
  - Then fetch records from file

- **Negation (取反):** $\sigma_{\neg\theta}(r)$
  - Use linear scan on file
  - If very few records satisfy $\neg\theta$, and an index is applicable to θ
    - Find satisfying records using index and fetch from file

大纲

- 概述
- 查询成本的度量
- 选择操作

分拣

- 连接操作
- 其他操作
- 表达式求值

分拣

- 我们可以在关系上建立索引，然后使用索引按照排序顺序读取关系。
  - 可能导致对每个元组进行一次磁盘块访问(对于非主索引)
- **适合内存的关系**
  - 可以使用快速排序之类的技术
- **不适合内存的关系**
  - **外部排序合并()是一个不错的选择**

排序的稳定性和复杂度

- 插入排序、选择排序、冒泡排序、快速排序、堆排序、归并排序、希尔排序、二叉树排序、计数排序、桶排序、基数排序 ...
- 不稳定：
  - (选择排序):O(n 2)
  - 快速排序(快速排序):O (nlogn)平均时间,O (n 2)最坏情况;对于大的、乱序串列一般认为是最快的已知排序
  - (堆排序):O(nlogn)
  - (shell sort): O(nlogn)
  - (基数排序):O(n·k);(K)

排序的稳定性和复杂度

- 插入排序、选择排序、冒泡排序、快速排序、堆排序、归并排序、希尔排序、二叉树排序、计数排序、桶排序、基数排序 ...
- 稳定：
  - (插入排序):O(n 2)
  - 菜籽油(冒泡排序):O(n 2)
  - **(归并排序):O(n log n);O(n)**
  - 二叉树排序(二叉树排序):O(nlogn);O(n)
  - (计数排序):O(n+k);Max-Min+1
  - (桶排序):O(n);O(k)

外部排序合并(排序合并)

- **Relations that don't fit in memory**
- Let M denote memory **buffer size** (in blocks)
- **Create sorted runs(归并段)**

  let $i = 0$ *initially*

  repeatedly do the following till the end of the relation:

  read M blocks of relation into memory

  sort the in-memory blocks

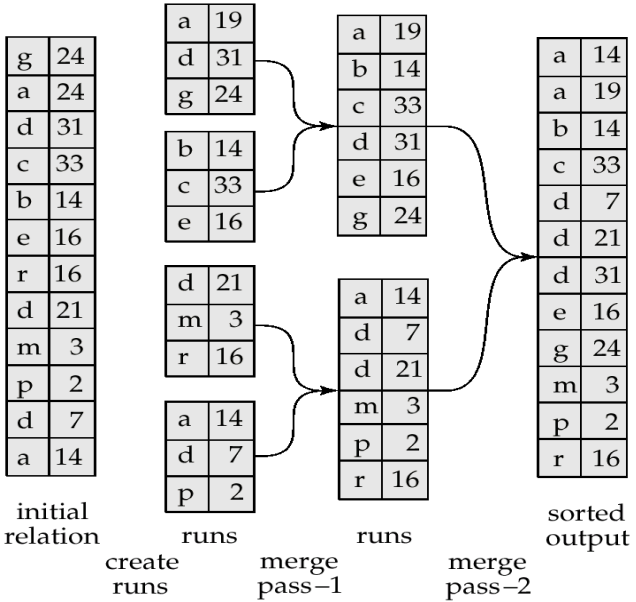
  write sorted data to run R_i

  increment i

  let the final value of $i = N$ (N-way merge)
- **Merge the runs** (next slide)

假设：

1. 一个块中只适合一个元组
2. 内存最多容纳3个块，2个用于输入，1个用于输出



外部排序合并(续)

- **合并运行(N-way Merge，N)。我们假设N < M**

  用N块内存来缓冲输入运行，用1块内存来缓冲输出。将每次运行的第一个块读入它的缓冲页

  重复

  在所有缓冲块中选择第一条记录(按排序顺序)

  将记录写入输出缓冲块。如果输出缓冲区已满，则将其写入磁盘

  从输入缓冲区块中删除记录，如果缓冲区块变为空，则将运行的下一个块读入缓冲区

  直到所有的输入缓冲块都为空

外部排序合并(续)

- 如果是N - M，则需要进行多次合并()
  - 在每一次归并中，合并连续的M - 1组。
  - 一次通过将运行次数减少M -1倍。
    - 例如，如果M= 11，并且有90次运行，一次通过将运行次数减少到9次，每次运行次数是初始运行次数的10倍
  - 重复的传递被执行，直到所有的运行被合并为一个

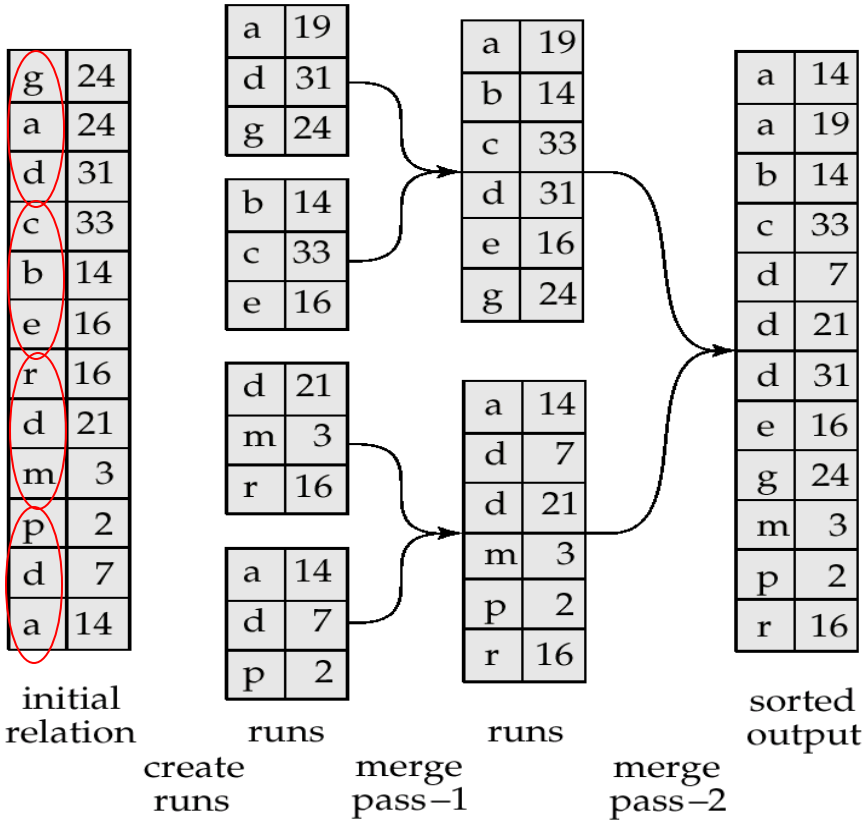
示例:使用排序合并的外部分拣

Sort on the first column!

Let ***M*** denote memory **buffer size**

Assume:

1. Only one tuple fits in a block
2. Memory holds at most **3** blocks, **2** for input and **1** for output
3. **Cost:** $b_r (2 \lceil \log_{M-1}(b_r / M) \rceil + 1)$
4. **Total:** $12(2*\log_2(12 / 3)+1) = 60$



外部合并排序(续)

- 成本分析：
  - *设b r表示包含关系r记录的块的数量*
  - 初始运行次数为⌈b r /M⌉
  - 所需的合并总次数:⌈log M -1 (b r /M)⌉。
  - 初始运行创建以及每次通过的磁盘访问是2 b r(读入+写出)
    - 对于最后一次传递，我们不计算写入成本。我们忽略所有操作的最终写成本，因为操作的输出可能会被发送到父操作而不被写入磁盘。
  - 每一次(除了最后一次)读取每个块一次，写入一次。因此，外部排序的磁盘访问总数：
    - *br(2⌈log M -1 (br / M)⌉+ 1):*

    示例:12(2*log 2(12 / 3)+1) =60

外部合并排序(Cont.)

- 查找的代价
  - 在运行生成过程中:每次运行一次seek读取，每次运行一次seek写入
    - *2⌈b r / M⌉*
  - 合并阶段
    - 缓冲区大小:b b(一次读/写b b块)
    - 每次合并通过需要2次⌈b r / b b⌉寻道
      - 除了最后一个不需要写的
    - 总寻数:2⌈b r / M⌉+⌈b r / b b⌉(2⌈log M -1 (b r / M)⌉-1)
    - 将等式应用到例子中，我们得到:

      2 *(12/3) +(12/1)(2 *日志2(12 / 3)1)= 8 + 12 * 3 = 44寻求

大纲

- 概述
- 查询成本的度量
- 选择操作
- 分拣

→联接操作

- 其他操作
- 表达式求值

连接操作

- 实现连接的算法
  - 嵌套循环连接()
  - 块嵌套环连接()
  - 索引嵌套环连接()
  - Merge-join ()
  - Hash-join ()
- 示例使用以下信息
  - #记录
    - 顾客:10000
    - 储户:5000
  - #块
    - 顾客:400
    - 储户:100

嵌套回路联接()

- Compute the theta join $r \bowtie_\theta S$

```
for each tuple t_r in r do begin
    for each tuple t_s in s do begin
        test pair (t_r, t_s) to see if they satisfy the join condition θ
            if they do, add t_r · t_s to the result.
    end
end
```

- r is called the **outer relation (外层关系)** and s is called the **inner relation (内层关系)**
- Require no indices and can be used for any kind of join condition
- **Expensive** since it examines every pair of tuples in the two relations

Nested-Loop Join (Cont.)

- In the worst case, if there is enough memory only to hold one block of each relation, the estimated cost is $n_r * b_s + b_r$ block transfers, plus $n_r + b_r$ seeks (r:outer relation (外层关系) s: inner relation (内层关系))
- If two or the smaller relation(s) fit(s) entirely in memory, use that as the inner relation.
  - Reduces cost to $b_r + b_s$ block transfers and 2 seeks
- If smaller relation (*depositor*) fits entirely in memory, the cost estimate will be 500 disk accesses
- Block nested-loops algorithm is preferable

#记录

顾客:10000

储户:5000

#块

顾客:400

储户:100

Nested-Loop Join (Cont.)

- 给定最坏情况下的内存可用性，成本估计为

  N_r或$b_s + b_r$块传输加上$N_r + b_r$查找

  - 5000 * 400 + 100 = 2,000,100次以存款人为外关系的磁盘访问，5000 + 100 = 5100次查找
  - 10000 * 100 + 400 = 100,400次以客户为外关系的磁盘访问，10,400次查找
  - **较小的关系做内层更优**
  - 如果较小的关系(存款人)完全适合内存，则成本估计将为500次磁盘访问

#记录

顾客:10000

储户:5000

#块

顾客:400

储户:100

Block Nested-Loop Join ()

- Variant of nested-loop join in which every block of inner relation is paired with every block of outer relation.

```
for each block B_r of r do begin
  for each block B_s of s do begin
      for each tuple t_r in B_r do begin
         for each tuple t_s in B_s do begin
            check if (t_r, t_s) satisfy the join condition
            if they do, add t_r · t_s to the result.
         end
           end
    end
end
```

块嵌套循环连接(连续)

- 最坏情况估计:b r或b s + b r块传输+ 2 * b r查找
  - 内部关系s中的每个块对外部关系中的每个块读取一次(而不是对外部关系中的每个元组读取一次)
  - 注：如内存不能容纳任何一个关系，则用较小的关系作为外层关系更有效
- 如。块嵌套循环连接的代价
  - 100*400 +100 = 40,100块传输+ 2 * 100 =200次寻道

- 最好的情况下(内存能容纳内层关系,较小的关系做内层):b r + b s + 2寻求的块传输

#记录
顾客:10000
储户:5000
#块
顾客:400
储户:100

块嵌套循环连接(连续)

- **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)个磁盘块作为外部关系的块单元，其中M =以块为单位的内存大小;使用剩余的两个块缓冲内部关系和输出
    - 成本=⌈b r / (M-2)⌉phd b s + b r块传输+ 2⌈b r / (M-2)⌉查找
  - 如果equi -join属性形成了一个内部关系的键，在第一次匹配时停止内部循环
  - 交替向前和向后扫描内部关系，以利用缓冲区中剩余的块(使用LRU替换)
  - 在可用的情况下对内部关系使用索引(下一张幻灯片)

索引嵌套循环连接()

- 索引查找可以代替文件扫描，如果
  - Join是一个对等连接或自然连接和
  - 索引可以在内部关系的join属性上使用
    - 可以构造一个索引只是为了计算一个连接。
- 对于外部关系r中的每个元组t r，使用索引查找s中满足元组t r连接条件的元组。
- 最坏的情况:缓冲区只有r的一页空间，并且，对于r中的每个元组，我们在s上执行索引查找。
- 连接的代价:b r (t t + t S) + n r±c
  - 其中c是遍历索引并为r的一个元组获取所有匹配的s个元组的代价
  - *C可以估计为使用连接条件对s进行一次选择的成本。*
- 如果r和s的连接属性上都有索引可用，则使用元组较少的关系作为外部关系。

索引嵌套循环连接成本示例

- **Compute depositor ⋈ customer**
  - Let *customer* have a primary B^+-tree index on the join attribute *customer-name*, which contains 20 entries in each index node
  - *customer* has 10,000 tuples (400 blocks), the height of the tree is 4, and one more access to find the actual data
  - *depositor* has 5000 tuples -> 100 blocks
- **Cost of block nested loops join**
  - 100*400 +100 = 40,100 block transfers + 2 * 100 =200 seeks
    - assuming worst case memory(较小的关系做外层更好)
    - may be significantly less with more memory
- **Cost of indexed nested loops join**
  - 100 + 5000 * 5 = 25,100 block transfers and seeks.
  - CPU cost likely to be less than that for block nested loops join
  - 均有索引，元组较少的做外层关系较好。
  - Less block transfer but more seeks

#记录

顾客:10000

储户:5000

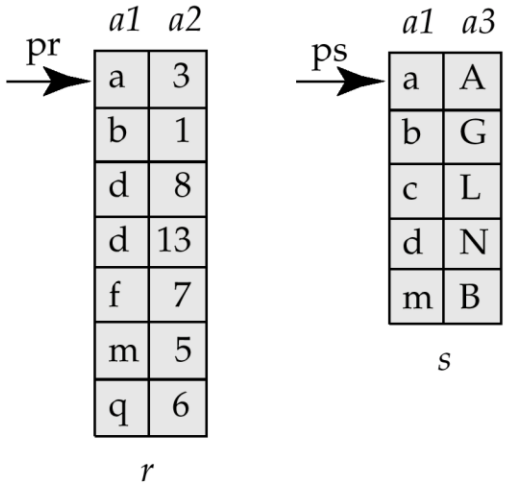
#块

顾客:400

储户:100

Merge-Join* ()

- 对两个关系的连接属性进行排序(如果还没有对连接属性进行排序)
- 合并已排序的关系以连接它们
  - Join步骤类似于排序-归并算法的归并阶段
  - 主要区别在于join属性中重复值的处理——join属性中具有相同值的每一对必须匹配



合并-连接(连续)

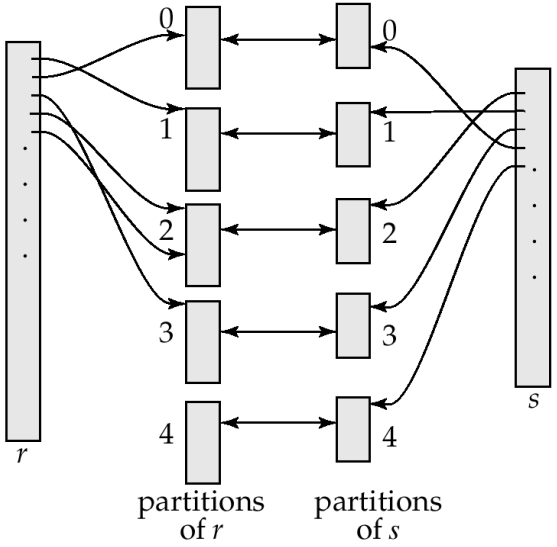
- 只能用于equi -join和natural -join
- 每个块只需要读取一次(假设连接属性的任意给定值的所有元组都适合内存)
- 因此，合并连接的块访问次数是b r + b s +如果关系未排序的排序成本。

- **混合merge-join(将索引与merge-join结合):如果一个关系被排序，而另一个关系在join属性上有一个二级B +树索引**
  - 将排序后的关系与B + -树的叶表项合并，结果文件包含排序后文件的元组和未排序文件的地址
  - 根据未排序关系元组的地址对结果文件进行排序
  - 按物理地址顺序扫描未排序的关系，并与之前的结果合并，用实际的元组替换地址

散列连接*

- A hash function h is used to partition tuples of both relations
  - h maps JoinAttrs values to $\{0, 1, ..., n\}$
  - $r_0, r_1, . . ., r_n$ denote partitions of r tuples
  - $s_0, s_1, . . ., s_n$ denotes partitions of s tuples

- r tuples in r_i need only to be compared with s tuples in s_i
  - an r tuple and an s tuple that satisfy the join condition will **have the same hash value for the join attributes**



散列连接算法

- The hash-join of r and s is computed as follows

1. 使用哈希函数h划分关系s。划分关系时，为每个分区保留一块内存作为输出缓冲区。
2. 分区r类似。
3. 对于每个i:

   (a)将i加载到内存中，并使用join属性在其上建立一个内存哈希索引。这个哈希索引使用的哈希函数与之前的h不同。

   (b)从磁盘中逐一读取r i中的元组。对于每个元组t r，使用内存中的哈希索引找到si中每个匹配的元组t s。输出它们的属性的连接。

Relation s is called the build input(构造用输入) and r is called the probe input(探查用输入)

散列连接算法(连续)

- 选择值n和哈希函数h，使得每个s i都应该适合内存。
  - 通常选择n为□b s /M□* f，其中f是一个"蒙混因子"，通常在1.2左右
  - 探测关系分区r i不需要放在内存中
- 如果分区数n大于内存页数M，则需要递归分区。
  - 与其划分n个分区，不如为s使用M - 1个分区
  - 使用不同的哈希函数进一步划分M - 1个分区
  - 在r上使用相同的分区方法

处理溢出

- 如果分区s i不适合内存，则会发生哈希表溢出。原因可能是
  - 许多元组在s中具有相同的连接属性值
  - 糟糕的哈希函数
- 溢出解析()可以在构建阶段完成
  - 分区i使用不同的散列函数进一步分区。
  - 分区ri也必须进行类似的分区。
- 避免溢出(溢出避免)执行分区仔细避免溢出在构建阶段
  - 例如，将分区构建关系分成许多分区，然后将它们组合起来
- 这两种方法都会因为大量的重复而失败。
  - 备用选项:在溢出分区上使用块嵌套循环连接

哈希连接的成本

- 如果不需要递归分区:哈希连接的代价是2(b r + b s) +(b r + b s) + 4n

- 如果需要递归分区，则分区s所需的传递次数为⌈log M - 1 (b s) - 1⌉。
- 对r进行分区所需的通过次数也与s相同。
- 因此，最好选择较小的关系作为构建关系。
- 总成本估算为:

  2 (b r + b s)⌈log M - 1 (b s) - 1⌉+ b r + b s
- 如果整个构建输入可以保存在主存中，则可以将n设置为0，并且算法不会将关系划分到临时文件中。成本估计下降到b r + b s。

哈希连接成本的例子

$$customer \bowtie depositor$$

- 假设内存大小为20块
- Bdepositor = 100, b customer = 400。
- 存款人将被用作构建输入。将其划分为5个分区，每个分区大小为20块。这个分区可以一次完成。
- 同样，将客户分区为5个分区，每个分区大小为80。这也是1遍完成的。
- 因此总成本:3(100 + 400)= 1500块转移
  - 忽略了写入部分填充的块的成本

混合的散列连接

- 当内存大小相对较大，构建输入大于内存时非常有用。
- 混合哈希连接的主要特点:将构建关系的第一个分区保留在内存中。
- 例如，在内存大小为25块的情况下，存款人可以被划分为5个分区，每个分区大小为20块。
- 内存划分:
  - 第一个分区占用20块内存。
  - 1块用于输入，其余4个分区各1块用于缓冲。

混合的散列连接

- customer is similarly partitioned into 5 partitions each of size 80; the first is used right away for probing, instead of being written out and read back.
- Cost of 3(80 + 320) + 20 +80 = 1300 block transfers for hybrid hash join, instead of 1500 with plain hash-join.
- Hybrid hash-join most useful if $M \gg \sqrt{b_s}$

复杂的连接

- **Join with a conjunctive condition(合取):** $r \bowtie_{\theta_1 \wedge \theta_2 \wedge \cdots \wedge \theta_n} s$
  - Either use nested loops/block nested loops, or
  - Compute the result of one of the simpler joins $r \bowtie_{\theta i} s$
  - final result comprises those tuples in the intermediate result that satisfy the remaining conditions $\theta_1 \wedge \cdots \wedge \theta_{i-1} \wedge \theta_{i+1} \wedge \cdots \wedge \theta_n$

- **Join with a disjunctive condition(析取):** $r \bowtie_{\theta_1 \vee \theta_2 \vee \cdots \vee \theta_n} s$
  - Either use nested loops/block nested loops, or
  - Compute as the union of the records in individual joins $r \bowtie_{\theta_i} s$:

$$(r \bowtie_{\theta_1} s) \cup (r \bowtie_{\theta_2} s) \cdots \cup (r \bowtie_{\theta_n} s)$$

复杂的连接

- **Join involving three relations: loan ⋈ depositor ⋈ customer**
  - Strategy 1: Compute depositor ⋈ customer; use result to compute loan ⋈ (depositor ⋈ customer)
  - Strategy 2: Computer loan ⋈ depositor first, and then join the result with customer.
  - **Strategy 3:** Perform the pair of joins at once. Build an index on loan for loan-number, and on customer for customer-name.
    - For each tuple *t* in depositor, look up the corresponding tuples in customer and the corresponding tuples in loan.
    - Each tuple of depositor is examined exactly once
    - Strategy 3 combines two operations into one special-purpose operation that is more efficient than implementing two joins of two relations.

大纲

- 概述
- 查询成本的度量
- 选择操作
- 排序
- 连接操作

<s:1>其他操作

- 表达式求值

重复消除和投影

- **重复消除可以通过哈希或排序来实现**
  - 在排序时，重复项将彼此相邻，并且除了一个副本之外的所有副本都可以删除。
  - 优化:重复项可以在运行生成过程中删除，也可以在外部排序合并的中间合并步骤中删除
  - 哈希是类似的-重复项将进入相同的桶
- **投影是通过在每个元组上执行投影，然后消除重复来实现的**

聚合

- **聚合可以以类似于重复消除的方式实现**
  - 可以使用排序或散列将同一组中的元组聚集在一起，然后可以在每个组上应用聚合函数
  - 优化:通过计算部分聚合值，在运行生成和中间合并期间组合同一组中的元组
    - 对于count, min, max, sum:保留到目前为止在组中找到的元组的聚合值。
    - 对于avg，保留sum和count，并在最后将sum除以count

集合操作

- **Set operations (∪, ∩ and −):** can either use variant of **merge-join after sorting**, or variant of **hash-join**
- E.g., set operations using hashing
  - Partition both relations using the same hash function, thereby creating, r_1, …, r_n and s_1, s_2,…, s_n
  - Process each partition i as follows. Using a different hashing function to build an in-memory hash index on r_i after it is brought into memory
    - $r \cup s$: add tuples in s_i to the hash index if they are not already in it. Finally, add the tuples in the hash index to the result
    - $r \cap s$: output tuples in s_i to the result if they are already there in the hash index
    - $r - s$: for each tuple in s_i, if it is in the hash index, delete it from the index. Finally, add the remaining tuples in the hash index to the result

外连接

- Outer join can be computed either as
  - A join followed by addition of null-padded non-participating tuples
  - by modifying the join algorithms
- Modifying merge join to compute $r \mathbin{⟕} s$
  - In $r \mathbin{⟕} s$, non participating tuples are those in $r - \Pi_R(r \bowtie s)$
  - Modify merge-join to compute $r \mathbin{⟕} s$: During merging, for every tuple t_r from r that do not match any tuple in s, output t_r padded with nulls
  - Right outer-join and full outer-join can be computed similarly

大纲

- 概述
- 查询成本的度量
- 选择操作
- 排序
- 连接操作
- 其他操作

→表达式求值

表情评价

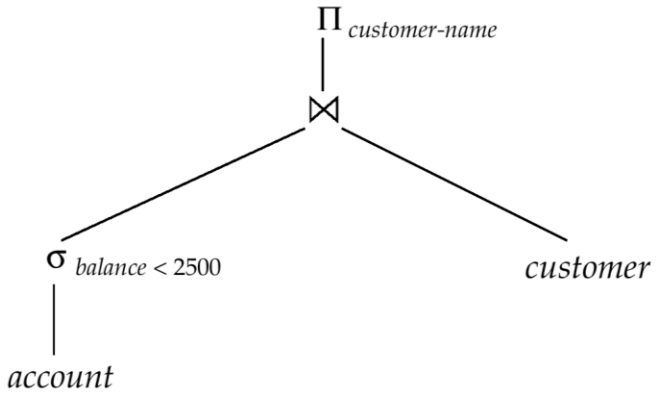
- 我们已经看到了单个操作的算法
- 对整个表达式树求值的替代方法
  - **物化(Materialization):生成输入是关系或已经计算过的表达式的结果，物化(存储)到磁盘上**
  - **流水线(Pipelining):将元组传递给父操作，即使操作正在执行**

物化()

- **Materialized evaluation (物化计算)**：evaluate one operation at a time, starting at the lowest-level.
- E.g., for the figure below, compute and store

$$\sigma_{balance<2500}(account)$$

- then compute and store the previous result' join with customer, and finally compute the projections on customer-name.



实体化(续)。

- **物化评价(物化计算)总是适用的**
- 将结果写入磁盘并将其读取回来的成本可能相当高
  - **总成本=单个操作的成本之和+将中间结果写入磁盘的成本**

- **双缓冲:为每个操作使用两个输出缓冲区，当一个缓冲区已满时将其写入磁盘，而另一个缓冲区将被填满**
  - 减少执行时间

流水线()

- **Pipelined evaluation** (流水线计算): evaluate several operations simultaneously, passing the results of one operation to the next
- E.g., in previous expression tree, don't store the result of

$$\sigma_{balance<2500}(account)$$

  - instead, pass tuples directly to the join. Similarly, don't store result of join, pass tuples directly to projection
- **Much cheaper than materialization**
- Pipelining may not always be possible – e.g., sort, hash-join
- Pipelines can be executed in two ways:
  - **demand driven (需求驱动)** and **producer driven (生产者驱动)**

管道(续)。

- **需求驱动或惰性评估**
  - 系统从顶层操作中反复请求下一个元组
  - 每个操作根据需要从子操作请求下一个元组，以便输出它的下一个元组
  - 在调用之间，操作必须保持"状态"，以便它知道下一步返回什么
  - 每个操作都被实现为实现以下操作的迭代器
    - open ()
      - 例如，文件扫描:初始化文件扫描，将指向文件开头的指针存储为状态
      - 例如，合并连接:排序关系和存储指针的开始排序关系作为状态
    - next ()
      - 例如，对于文件扫描:输出下一个元组，并推进和存储文件指针
      - 例如，对于合并连接:从先前的状态继续合并，直到找到下一个输出元组。将指针保存为迭代器状态。
    - close ()

管道(续)。

- **生产者驱动的或急切的流水线**
  - 操作符急切地生成元组，并将其传递给父节点
    - 操作符之间维护缓冲区，子操作符将元组放入缓冲区，父操作符将元组从缓冲区中移除
    - 如果缓冲区已满，子节点等待直到缓冲区中有空间，然后生成更多的元组
  - 系统调度在输出缓冲区中有空间并且可以处理更多输入元组的操作

第十五讲结束