

21121350

Database System

Lecture 9: Query Processing

Lu Chen (陈璐)

College of Computer Science

Zhejiang University

Spring & Summer 2023

luchen@zju.edu.cn/18868818726

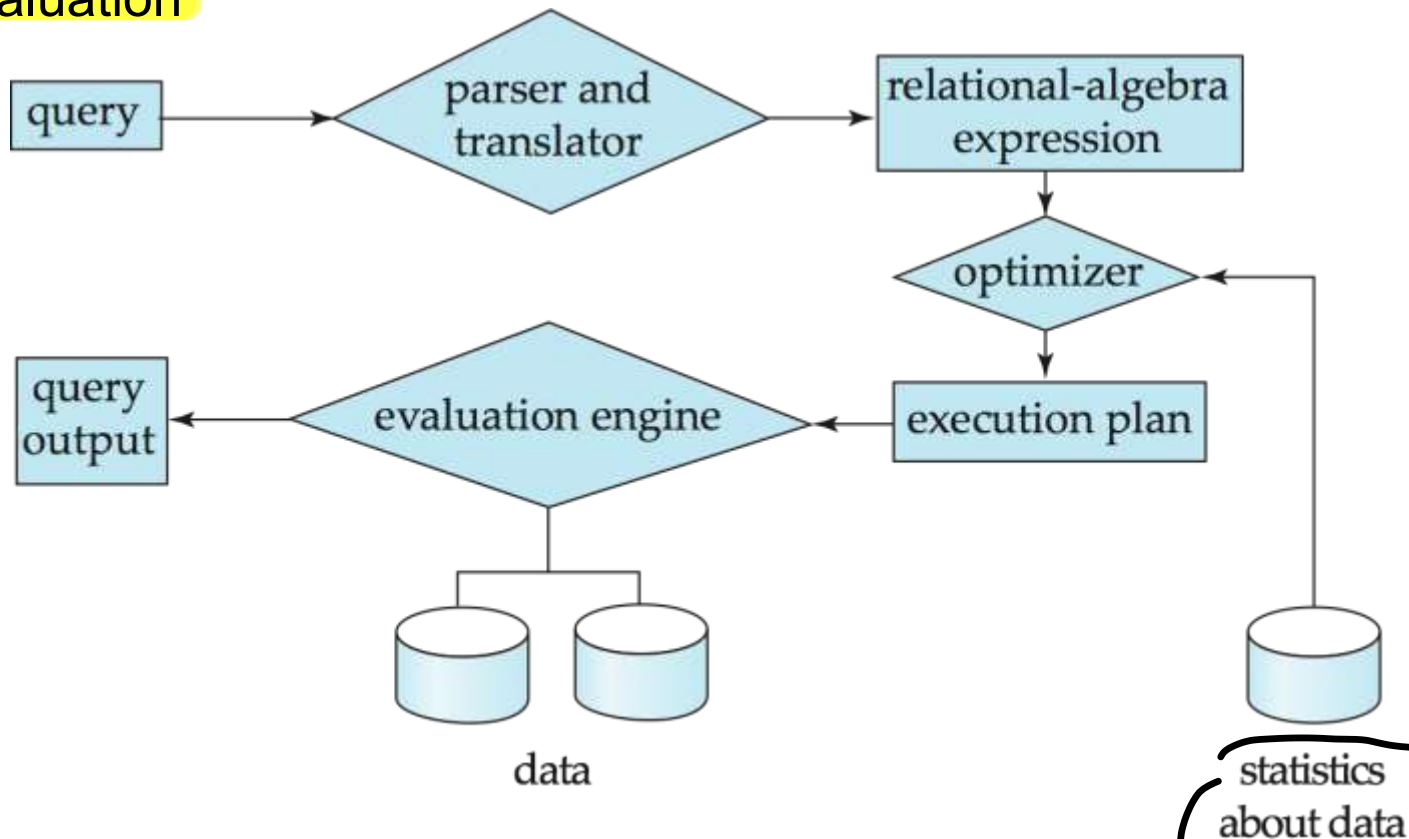
Outline

- ❑ Overview
- ❑ Measures of Query Cost
- ❑ Selection Operation
- ❑ Sorting
- ❑ Join Operation
- ❑ *Other Operations
- ❑ *Evaluation of Expressions



Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation

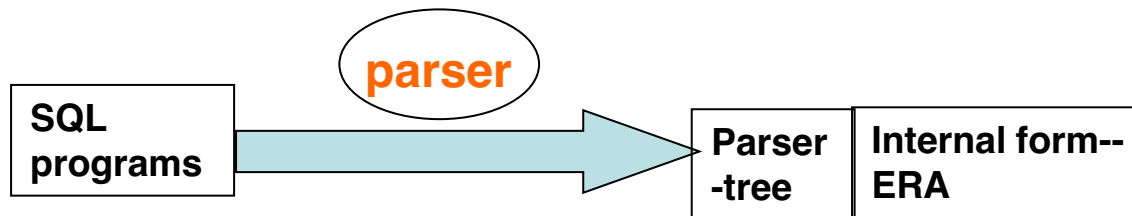


Basic Steps in Query Processing (Cont.)

略, 仅了解

❑ Parsing and translation (语法分析与翻译) vs. compiler

- Parser checks **syntax**, **verifies relations**
- Translate the query into its internal form--- **extended relational algebra (ERA)**.



❑ Evaluation

- The query-execution engine takes a **query-evaluation plan**, executes that plan, and returns the answers to the query.

❑ An **evaluation plan** defines exactly what algorithm is used for each operation, and how the execution of the operations is coordinated.

Basic Steps in Query Processing (Cont.)

□ Optimization ----- why ?

- (1) For a given SQL query, may have many **equivalent relational algebra expressions**.

E.g.: `Select balance from account where balance > 2500;`

(a) $\sigma_{balance > 2500}(\Pi_{balance}(account))$

(b) $\Pi_{balance}(\sigma_{balance > 2500}(account))$

先做投影: table有很多个attribute
先做select: 满足select condition的很少

- (2) Each relational algebra operation can be evaluated using one of several **different algorithms**.
 - E.g., For above (b), how to do the σ operation, use **linear scan**, or **index** ?
- Correspondingly, a relational-algebra expression can be evaluated in many ways.
- Annotated expression specifying detailed evaluation strategy is called an **evaluation-plan**.

Basic Steps: Optimization(Cont.)

- ❑ (1) Find out various equivalent relational algebra expressions. (chp. 16)
- ❑ (2) A sequence of primitive operations which specify detailed evaluation strategy is called a **query-execution plan** or **query evaluation-plan**.

E.g1:

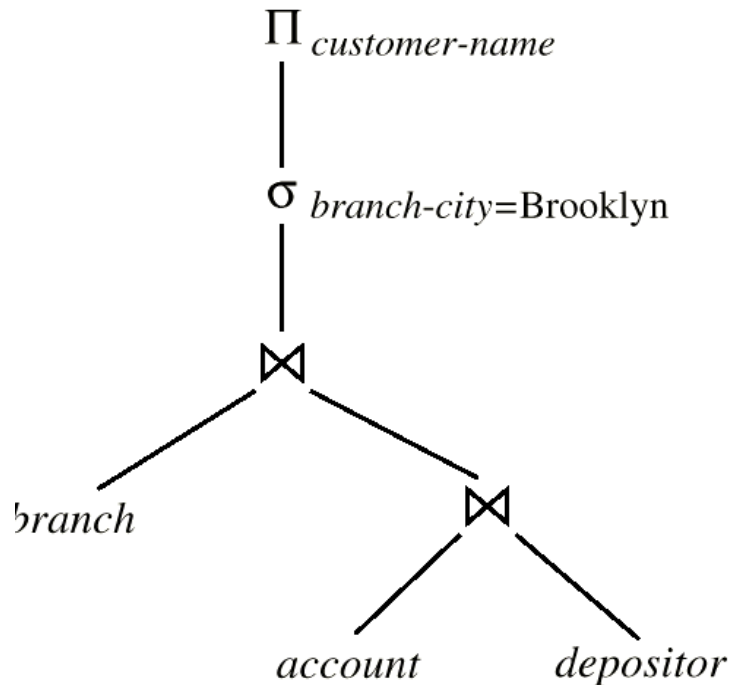
(1) **use an index on *balance*** to find accounts with $\text{balance} > 2500$

Π_{balance}
|
 $\sigma_{\text{balance} > 2500}$, **using index 1**
|
account

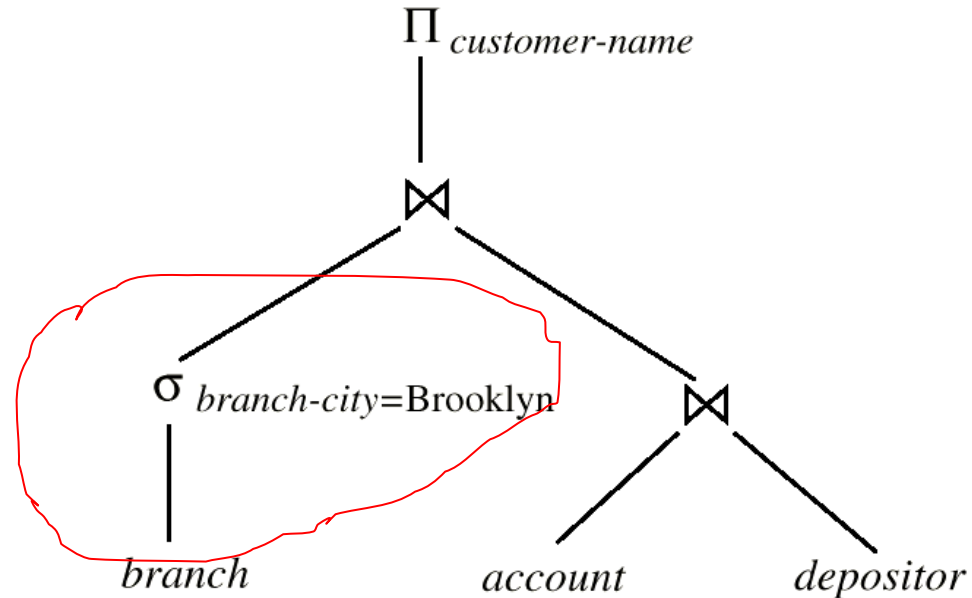
query-execution plan

E.g2: $\Pi_{customer-name} (\sigma_{branch-city='Brooklyn'} (branch \bowtie account \bowtie depositor))$

先做过滤, 减少参与join的data数



(a) Initial Expression Tree



(b) Transformed Expression Tree

Some equivalent relational algebra expressions
(Partial query-execution plan)

Basic Steps: Optimization (Cont.)

- ❑ (3) Query Optimization: Among all equivalent evaluation plans choose the one with lowest cost. Two factors are considered for cost:
 - Cost is dependent on the algorithms of the executions.
 - Cost is also estimated **using statistical information** from the **database catalog**.
 - E.g., the number of tuples in each relation, the size of tuples, etc.
- ❑ In this chapter we study
 - How to measure query costs:
 - Algorithms for evaluating relational algebra operations: **select, join**
 - How to **combine algorithms** for individual operations in order to evaluate a complete expression
- ❑ In Chapter 16
 - We study how to optimize queries, that is, how to **find an evaluation plan with lowest estimated cost**

Outline

- ❑ Overview
- ❑ Measures of Query Cost
- ❑ Selection Operation
- ❑ Sorting
- ❑ Join Operation
- ❑ *Other Operations
- ❑ *Evaluation of Expressions



Three Points

- ❑ CPU can't manipulate the data in disk directly.
- ❑ The growth rate of the disk data read/write (transfer) speed is much slower than that of the disk size.
 - In 20 years, disk size: 1000 times; read/write speed: 40 times
- ❑ The growth rate of the disk seek speed is much slower than that of the disk data transfer speed.

Measures of Query Cost

- ❑ Cost is generally measured as total elapsed time for answering query.
 - Many factors contribute to time cost
 - Disk access + CPU + network communication
- ❑ Typically disk access is the predominant cost, and is relatively easy to estimate. Measured by taking into account:
 - Number of seek operations performed
 - Number of blocks read \times average-block-read-cost
 - Number of blocks written \times average-block-write-cost
 - Cost to write a block is greater than cost to read a block
 - Data is read back after being written to ensure that the write was successful

Measures of Query Cost (Cont.)

- ❑ For simplicity, we just use the number of block transfers from disk and the number of seeks as the cost measures:

- t_T – time to transfer one block. ($\approx 0.1\text{ms}$) *不分 read & write*
- t_S – time for one seek. ($\approx 4\text{ms}$)
- Cost for b block transfers plus S seeks
 $b * t_T + S * t_S$

- ❑ We ignore CPU costs for simplicity.

- But real systems do take CPU cost into account

- ❑ We do not include cost to writing the final result back to disk in our cost formulae.

Measures of Query Cost (Cont.)

- ❑ Costs depends on the size of the buffer in main memory
 - Having more memory, reduces need for disk access
 - Amount of real memory available to buffer depends on other concurrent OS processes, and hard to determine ahead of actual execution
 - We often use worst case estimates, assuming only the minimum amount of memory needed for the operation is available, and the best case estimates.
 - Required data may be buffer resident already, avoiding disk I/O. But hard to take into account for cost estimation

Outline

- ❑ Overview
- ❑ Measures of Query Cost
- ❑ **Selection Operation**
- ❑ Sorting
- ❑ Join Operation
- ❑ *Other Operations
- ❑ *Evaluation of Expressions



Selection Operation: (1) Basic algorithms

- ❑ **File scan** – search algorithms that locate and retrieve records that fulfill a selection condition, **do not use index**. 把block一个个读入内存中来
- ❑ **Algorithm A1 (linear search)**. Scan each file block and test all records to see whether they satisfy the selection condition. 看是否满足
 - **Cost estimate = b_r block transfers + 1 seek** 文件都存在一起: 只要寻址一次, 接着顺序往下走就行
 - b_r denotes number of blocks containing records from relation r
 - If selection is on a key attribute, can stop on finding record
 - **cost = $(b_r/2)$ block transfers + 1 seek** 最好=1, 最坏: b_r , 平均: $\frac{b_r}{2}$
 - Linear search can be applied regardless of
 - selection condition or
 - ordering of records in the file, or
 - availability of indices
- ❑ **Note: binary search generally does not make sense since data is not stored consecutively**
 - Except when there is an index available,
 - And binary search requires more seeks than index search

Basic algorithms

❑ **A2 (binary search).** Applicable if selection is an equality comparison on the attribute on which file is ordered.

➤ Assume that the blocks of a relation are stored contiguously

selection on a key attribute ➤ Cost = $\lceil \log_2(b_r) \rceil$ block transfer + $\lceil \log_2(b_r) \rceil$ seek – cost of locating the first tuple by a binary search on the blocks.

(time cost = $\lceil \log_2(b_r) \rceil * (t_s + t_r)$)

要重新定位寻址(访问几个block, seek几次)

➤ If selection is not on a key attribute, need plus the number of blocks containing records that satisfy selection condition. 使用binary search 默认block已排好序

block transfer = $\lceil \log_2(b_r) \rceil + \lceil \text{sc}(A, r) / f_r \rceil - 1$, 其中 $\text{sc}(A, r)$ 是满足选则条件的记录数, f_r 是每个block的记录数 (See chpt16)

表示满足条件的block数

= -1 : 找到的第一条记录所在block
在 $\lceil \log_2(b_r) \rceil$ 和 $\lceil \text{sc}(A, r) / f_r \rceil$ 中都算了一遍

binary search 相比 linear search: block transfer ↓
但 seek ↑, 在记录数很多的时候体现优势

A-217	Brighton	750	
A-101	Downtown	500	
A-110	Downtown	600	
A-215	Mianus	700	
A-102	Perryridge	400	
A-201	Perryridge	900	
A-218	Perryridge	700	
A-222	Redwood	700	
A-305	Round Hill	350	



account(account_number, branch_name, balance)

Assume: account is sequentially ordered on **search-key branch_name**.
 Select * from account where branch_name = 'Downtown';
 May return 100 tuples.

(2) Selections Using Indices and equality

❑ **Index scan** — search algorithms that use an index

- Selection condition must be on search-key of index.

❑ **A3 (primary index, equality on key).** Retrieve a single record that satisfies the corresponding equality condition

- $Cost = (h_i + 1) * (t_T + t_S)$, 其中 h_i 为索引树高 → 访问节点数
叶节点存的是指向真实 record 的 pointer

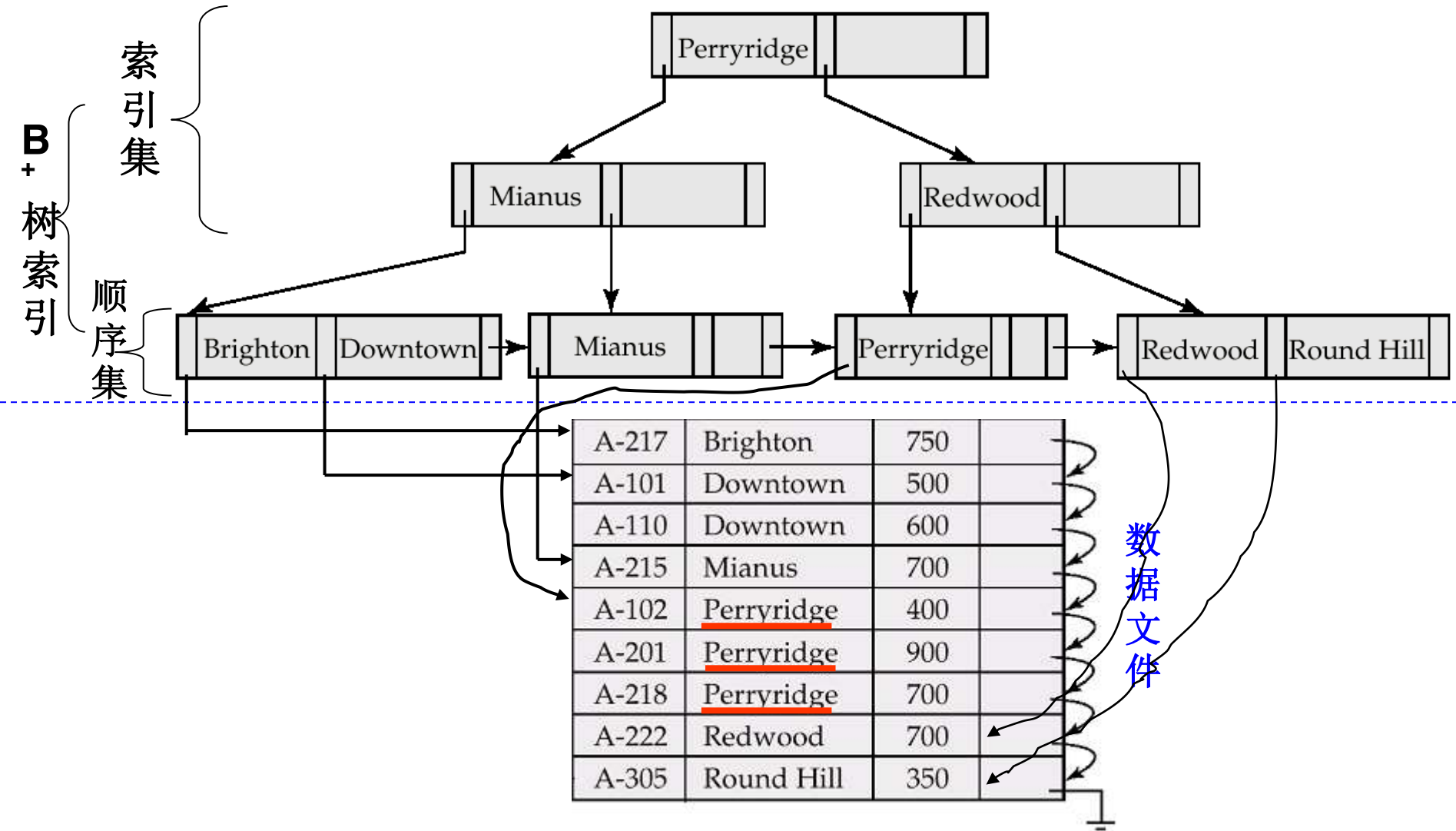
❑ **A4 (primary index, equality on nonkey)** Retrieve multiple records.

- Records will be on consecutive blocks
 - Let b = number of blocks containing matching records
- $Cost = h_i * (t_T + t_S) + t_S + t_T * b$, b containing retrieved records

找到第一条符合条件的记录
再寻址一次

$$= \lceil sc(A, r) / f_r \rceil$$

B⁺-tree for *account* file ($n = 3$)



Selections Using Indices

□ A5 (secondary index, equality on nonkey).

- 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
 - Each of n matching records may be on a different block
 - $Cost = (h_i + n) * (t_T + t_S)$
 - Can be very expensive!
 - May be more worse than linear scan.

有几条记录就要 block transfer
和 seek 几次

balance

350	
400	
500	
600	
700	
750	
900	

Index entries



account

A-101	Downtown	500	
A-217	Brighton	750	
A-110	Downtown	600	
A-215	Mianus	700	
A-102	Perryridge	400	
A-201	Perryridge	900	
A-218	Perryridge	700	
A-222	Redwood	700	
A-305	Round Hill	350	

*Acc-num branch-name
balance*

(3) Selections Involving Comparisons

- ❑ 比较符: $>$, \geq , $<$, \leq , \neq , 与等值比较的不同之处在于选择范围大
- ❑ Can implement selections of the form $\sigma_{A \leq v}(r)$ or $\sigma_{A \geq v}(r)$, by using:

- a) A linear file scan;
- b) Binary search (as A2); 二叉索引
- c) Using indices in the following ways:

B+索引 (B+-Tree)

- ❑ **A6 (primary index, comparison, 基于主索引的比较).**

- For $\sigma_{A \geq v}(r)$, use index to find the first tuple $A \geq v$ and scan relation sequentially from there.
- For $\sigma_{A \leq v}(r)$, just scan relation sequentially till first tuple $A > v$; do not use index

- ❑ **A7 (secondary index, comparison, 基于辅助索引的比较)**

- pointer
是跳跃
的
- For $\sigma_{A \geq v}(r)$, use index to find first index entry $\geq v$ and scan index sequentially from there, to find pointers to records.
 - For $\sigma_{A \leq v}(r)$, just scan leaf pages of index finding pointers to records, till first entry $A > v$ is met
 - In either case, retrieve records that are pointed to
 - Requires an I/O for each record
 - Linear file scan may be cheaper

(4) Implementation of Complex Selections

❑ **Conjunction:** $\sigma_{\theta_1 \wedge \theta_2 \wedge \dots \wedge \theta_n}(r)$

❑ **A8 (conjunctive selection using one index).**

- Select a combination of θ_i , and algorithms A1 through A7 that results in the least cost for $\sigma_{\theta_i}(r)$. (第一步从 n 个条件 $\theta_1 \dots \theta_n$ 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他 θ_i .)
- Test other conditions on tuple after fetching it into memory buffer.

❑ **A9 (conjunctive selection using composite index).**

- Use appropriate composite (multiple-key) index if available.

不作要求, 较少用

❑ **A10 (conjunctive selection by intersection of identifiers).**

- Requires indices with record pointers. *pointer 求交集*
- Use corresponding index for each condition, and take intersection of all the obtained sets of record pointers.
- Then fetch records from file
- If some conditions do not have appropriate indices, apply test in memory.

Algorithms for Complex Selections

❑ **Disjunction:** $\sigma_{\theta_1 \vee \theta_2 \vee \dots \vee \theta_n}(r)$.

❑ **A10 (disjunctive selection by union of identifiers).**

- Applicable if *all* conditions have available indices.
 - Otherwise use linear scan.
- Use corresponding index for each condition, and take union of all the obtained sets of record pointers.
- Then fetch records from file

❑ **Negation:** $\sigma_{\neg\theta}(r)$

- 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

use index: 返回记录较少

Bitmap Index Scan

❑ The **bitmap index scan** algorithm of PostgreSQL

➤ Bridges gap between secondary index scan and linear file scan when number of matching records is not known before execution

➤ Bitmap with 1 bit per page in relation

➤ Steps:

- Index scan used to find record ids, and set bit of corresponding page in bitmap
- Linear file scan fetching only pages with bit set to 1

➤ Performance

- Similar to index scan when only a few bits are set
- Similar to linear file scan when most bits are set
- Never behaves very badly compared to best alternative

根据bitmap的情况来选择用哪种

Outline

- ❑ Overview
- ❑ Measures of Query Cost
- ❑ Selection Operation
- ❑ **Sorting**
- ❑ Join Operation
- ❑ *Other Operations
- ❑ *Evaluation of Expressions



Sorting

- ❑ Why: two cases for sorting of data:
 - Output request
 - Join operations can be implemented quickly.
- ❑ We may build an index on the relation, and then use the index to read the relation in sorted order. But this is only logically, not physically ordered the relation, and may lead to one disk block access for each tuple.
- ❑ How:
 - For relations that fit **in memory**, techniques like **quicksort** can be used for sorting.
 - For relations that **don't fit in memory**, **external sort-merge** (外部排序归并) is a good choice.

Example: External Sorting Using Sort-Merge

Assume: $M = 3$
one block for 1
tuple

算法思路类似于
体育比赛:

小组赛 → 大组赛
→ 决赛

g	24
a	19
d	31
c	33
b	14
e	16
r	16
d	21
m	3
p	2
d	7
a	14

Sort
key

initial
relation

$$N_0 = b/M$$

External Sort-Merge

Let M denote memory size (in pages).

1. **Create sorted runs**. Let i be 0 initially.

Repeatedly do the following till the end of the relation:

- (a) Read M blocks of relation into memory
- (b) Sort the in-memory blocks
- (c) Write sorted data to run R_i ; increment i .

Let the final value of i be N

2. **Merge the runs** (next slide)

External Sort-Merge (Cont.)

2. Merge the runs (N-way merge). We assume (for now) that $N < M$.
 1. Use N blocks of memory to buffer input runs, and 1 block to buffer output. Read the first block of each run into its buffer page
 2. Repeat
 1. Select the first record (in sort order) among all buffer pages
 2. Write the record to the output buffer. If the output buffer is full write it to disk.
 3. Delete the record from its input buffer page.
If the buffer page becomes empty then read the next block (if any) of the run into the buffer.
 3. Until all input buffer pages are empty:

External Sort-Merge (Cont.)

□ If $N \geq M$, several merge *passes* are required.

- In each pass, contiguous groups of $M - 1$ runs are merged.
- A pass reduces the number of runs by a factor of $M - 1$, and creates runs longer by the same factor.

- E.g., If $br = 1000$ blocks, $M = 11$, and one pass reduces the number of runs to 91, each 10 times the size of the initial runs

Pass 0: $N_0 = \lceil 1000/11 \rceil = 91$ runs;

Pass 1: $N_1 = \lceil 91/10 \rceil = 10$ runs;

Pass 2: $N_2 = \lceil 10/10 \rceil = 1$ runs.

- Repeated passes are performed till all runs have been merged into one.

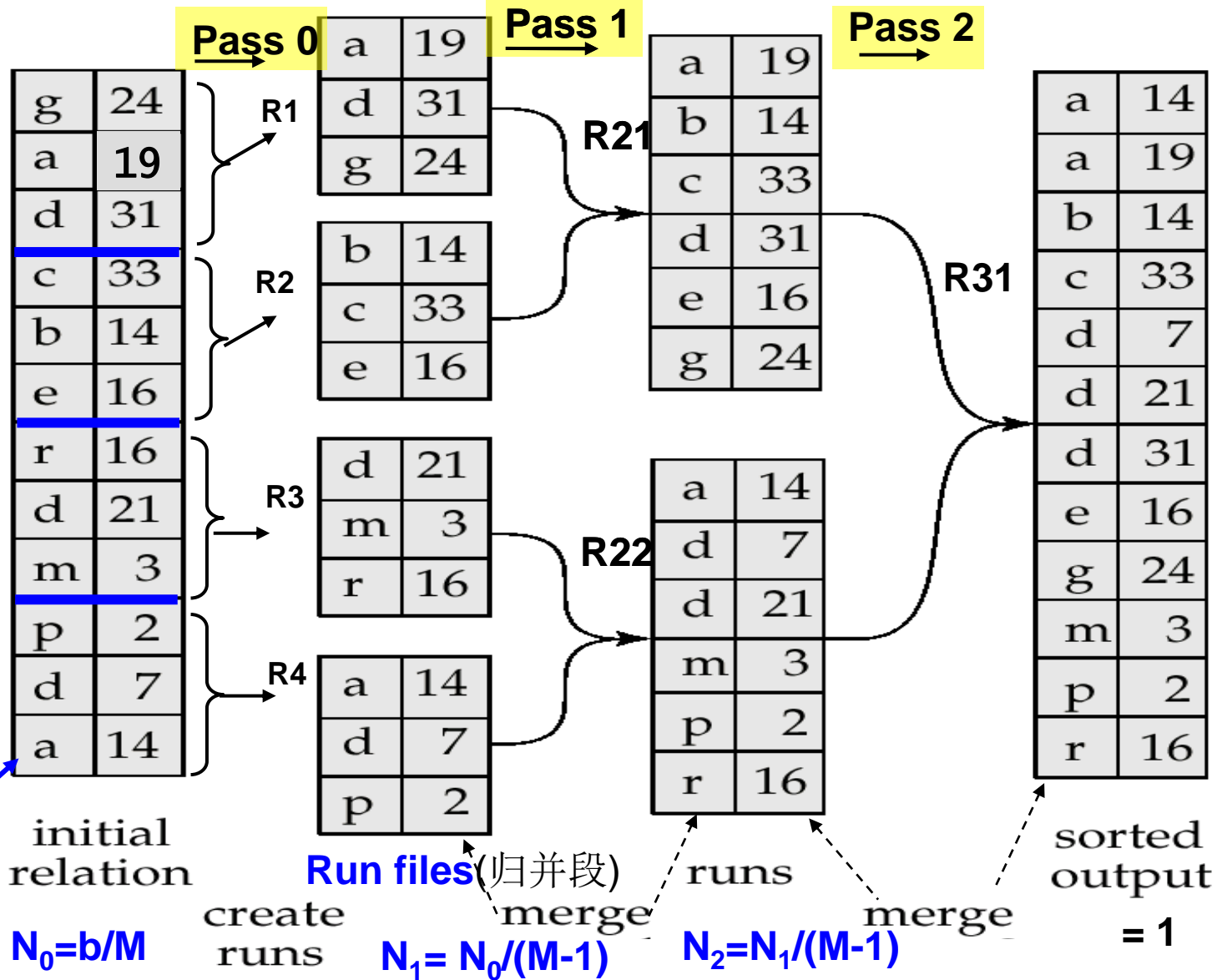
Example: External Sorting Using Sort-Merge

Assume: $M = 3$
one block for 1 tuple

算法思路类似于
体育比赛:

小组赛 → 大组赛
→ 决赛

Sort
key



External Merge Sort (Cont.)

❑ Cost analysis:

- Total number of merge passes required: $\lceil \log_{M-1}(b_r/M) \rceil$.
- Block transfers for initial run creation as well as in each pass is $2b_r$
 - For final pass, we don't count write cost
 - We ignore final write cost for all operations since the output of an operation may be sent to the parent operation without being written to disk
 - Thus total number of block transfers for external sorting:
$$b_r (2 \lceil \log_{M-1}(b_r/M) \rceil + 1)$$
- Seeks: next slide

External Merge Sort (Cont.)

❑ Cost of seeks

- During run generation: one seek to read each run and one seek to write each run

- $2 \lceil b_r / M \rceil$

- During the merge phase

- Buffer size: b_b (read/write b_b blocks at a time)
 - Need $2 \lceil b_r / b_b \rceil$ seeks for each merge pass
 - except the final one which does not require a write

- Total number of seeks:

- $$2 \lceil b_r / M \rceil + \lceil b_r / b_b \rceil (2 \lceil \log_{M-1}(b_r / M) \rceil - 1)$$

Outline

- ☐ Overview
- ☐ Measures of Query Cost
- ☐ Selection Operation
- ☐ Sorting
- ☐ Join Operation
- ☐ *Other Operations
- ☐ *Evaluation of Expressions



Join Operation

depositor ⋈ customer

❑ Several different algorithms to implement joins

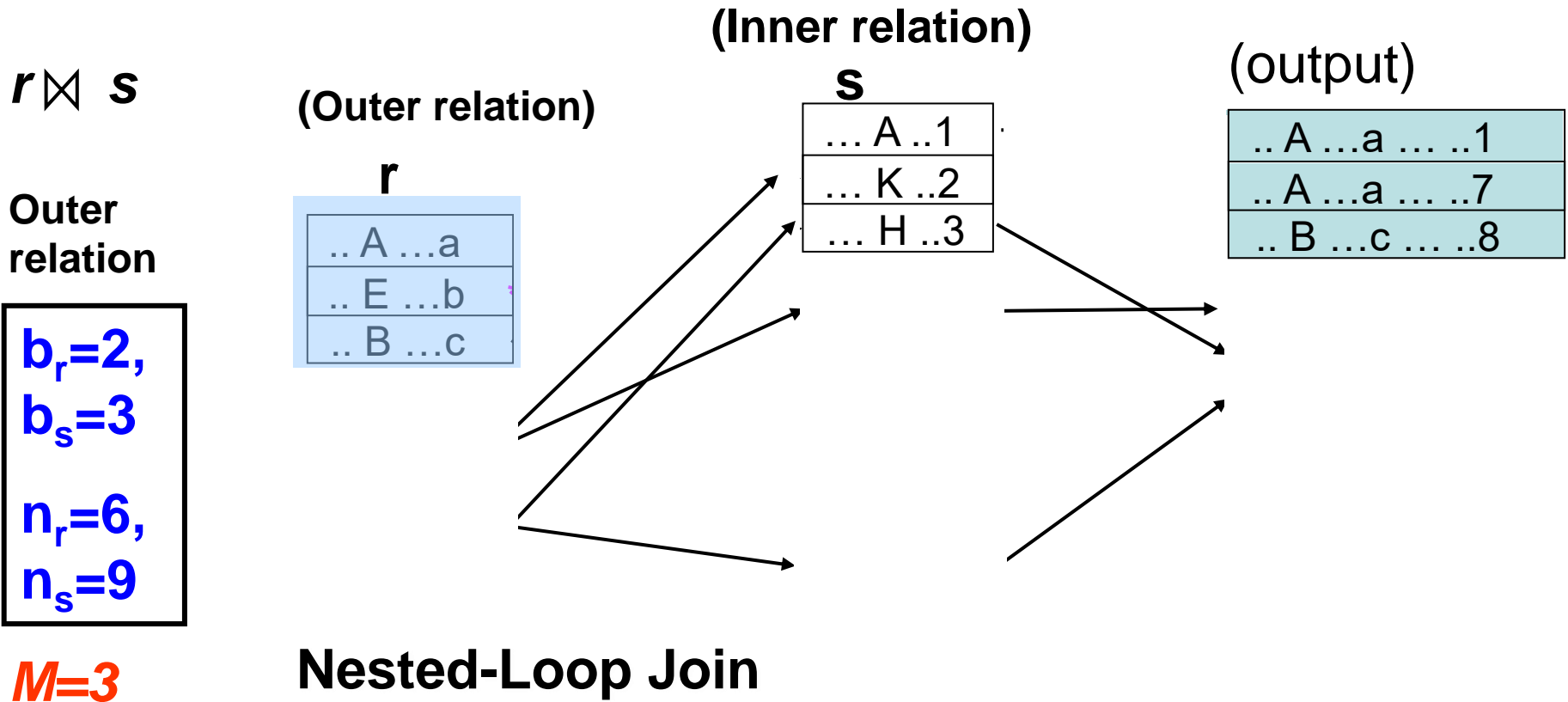
- Nested-loop join
- Block nested-loop join
- Indexed nested-loop join
- Merge-join
- Hash-join

❑ Examples use the following information

Num of records of *customer.n* = 10,000, *depositor.n* = 5000

Num of blocks of *customer.b* = 400 , *depositor.b* = 100

(1) Nested-Loop Join(嵌套循环连接)




The **cost in block transfer in the worst case**: (only one block of each relation can be held in memory) = $6 * 3 + 2 = 20$

Seeks = $6 + 2 = 8$

(1) Nested-Loop Join(嵌套循环连接)

□ To 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  $\theta$ 
    if they do, add  $t_r \cdot t_s$  to the result.
  end
end
```

➤ r is called the outer relation and s the inner relation of the join.

- **Requires no indices** and can be used with 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

$$n_r * b_s + b_r \text{ block transfers, plus } n_r + b_r \text{ disk seeks}$$

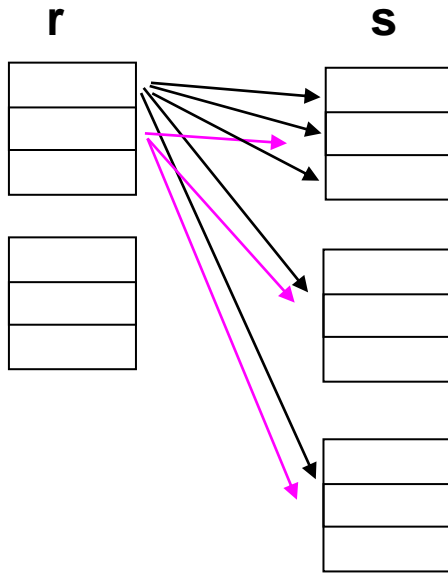
- ❑ If the smaller relation fits entirely in memory, use that as the inner relation:

- In the best case:

$$\text{cost} = b_r + b_s \text{ block transfers plus 2 seeks.}$$

- ❑ Assuming worst case memory availability, cost estimate is
 - $5000 * 400 + 100 = 2,000,100$ block transfers and $5000 + 100 = 5100$ seeks with *depositor* as outer relation, and
 - $10000 * 100 + 400 = 1,000,400$ block transfers and $10000 + 400 = 10,400$ seeks with *customer* as the outer relation.
- ❑ If smaller relation (*depositor*) fits entirely in memory, the cost estimate will be 500 block transfers plus 2 seeks.
- ❑ Block nested-loops algorithm (next slide) is preferable.

(2) Block Nested-Loop Join



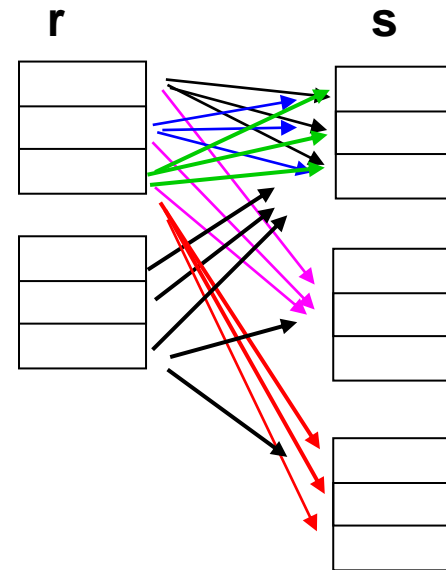
Nested-Loop Join

BT Cost = $6 * 3 + 2 = 20$

Seeks = $6 + 2 = 8$

outer relation

Inner relation



$M=3$

Block Nested-Loop Join

Block transfer cost = $2 * 3 + 2 = 8$

Seeks = 4

Each block in the inner relation *s* is read once for each *block* in the outer relation (instead of once for each tuple in the outer relation)

(2) 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 $t_r \cdot t_s$ to the result.


end

end

end

end

在内存
中处理



Block Nested-Loop Join (Cont.)

- ❑ Worst case: Block transfer cost = $b_r * b_s + b_r$
 - (设内存中只有3个block) plus $2 * b_r$ seeks
 - E.g., $100 * 400 + 100 = 40100$ block transfers, plus 200 seeks;
or $400 * 100 + 400 = 40400$ block transfer, 800seeks
 - Let the relation with smaller number of blocks be outer relation
- ❑ Best case: $b_r + b_s$ block accesses
 - (s全部且始终在内存) . plus 2 seeks
 - Let the relation with smaller number of blocks be inner relation

Block Nested-Loop Join (Cont.)

❑ Improvements :

- 1) when $M > 3$, but $M < b_s$ and $M < b_r$
use $M - 2$ disk blocks as blocking unit for outer relations, where M = memory size in blocks; use remaining two blocks to buffer inner relation and output。--充分利用内存buffer, 增大buffer中外层关系的block数（从而减少内层关系读入批数), 有利于减少代价（减少约M-2倍）。

Data Transfer cost = $\lceil b_r / (M-2) \rceil * b_s + b_r$

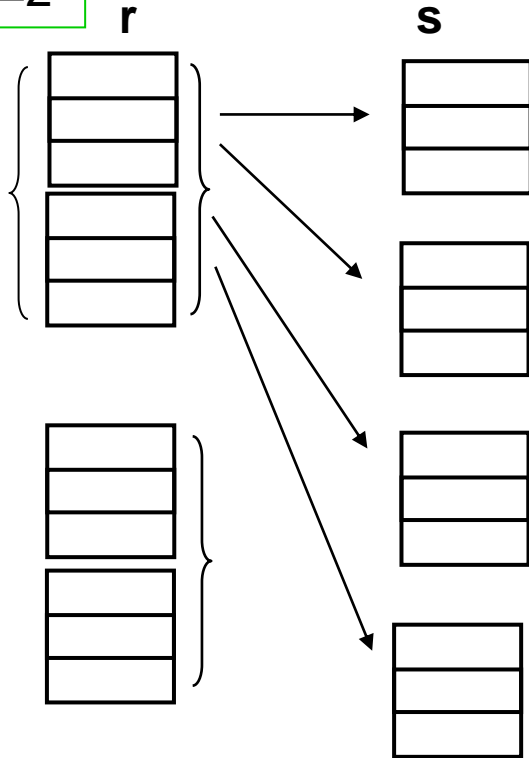
Seeks times = $2 \lceil b_r / (M-2) \rceil$

E.g. $M=10$, DT cost = $\lceil 400 / (10-2) \rceil * 100 + 400 = 5400$, seeks = $2 * 50$

How about $M-2$ blocks for inner relation ?

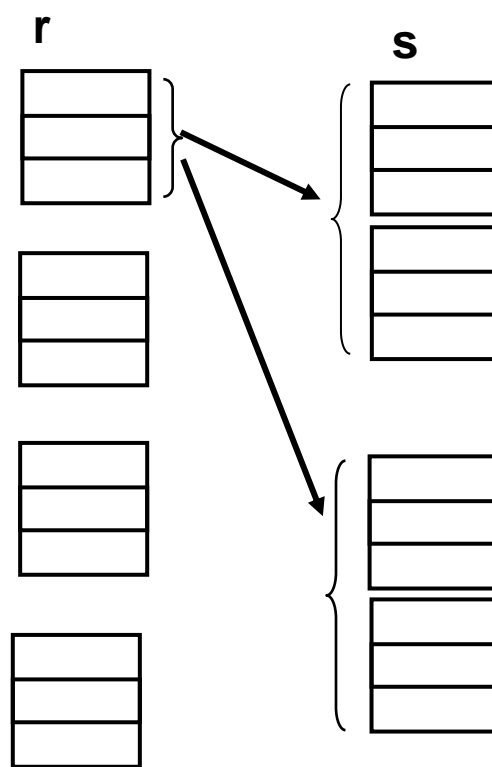
M = 4

M-2=2



BT Cost= 2 x 4 + 4 = 12

Seeks= 2 * 2 = 4



BT Cost= 4 x 4 + 4 = 20

Seeks=2 * 4 =8

Block Nested-Loop Join (Cont.)

❑ Improvements:

- 2) If equi-join(等值连接) attribute forms a key on inner relation, stop inner loop on first match
- 3) Scan inner loop forward and backward alternately, to make use of the blocks remaining in buffer (when LRU replacement strategy is used)
- 4) Use index on inner relation if available (next slide)

(3) Indexed Nested-Loop Join

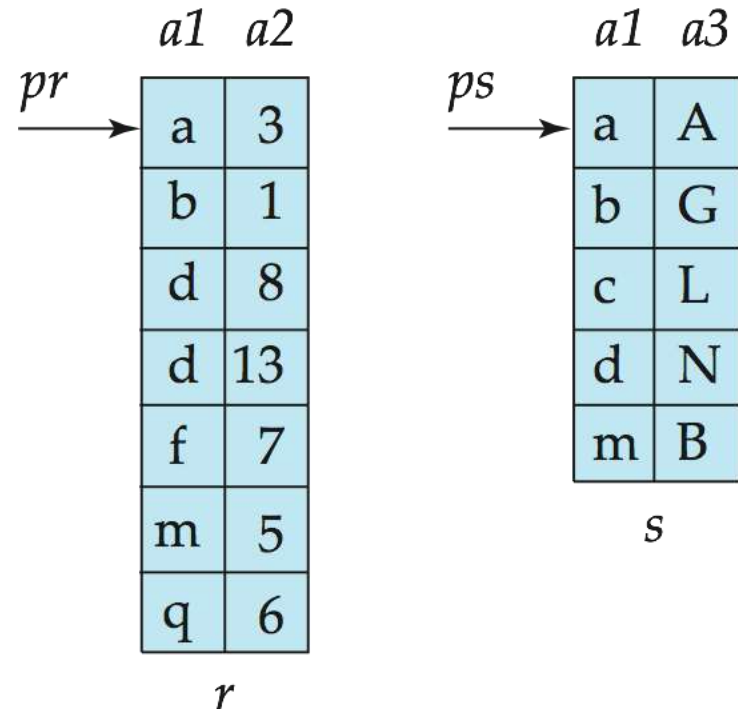
- ❑ Index lookups can replace file scans if
 - Join is an equi-join or natural join and
 - An index is available on the inner relation's join attribute
 - Can construct an index just to compute a join.
- ❑ For each tuple t_r in the outer relation r , use the index to look up tuples in s that satisfy the join condition with tuple t_r .
- ❑ Worst case: buffer has space for only one page of r , and, for each tuple in r , we perform an index lookup on s .
- ❑ Cost of the join: $b_r(t_T + t_S) + n_r * c$
 - Where c is the cost of traversing index and fetching all matching s tuples for one tuple of r
 - c can be estimated as cost of a single selection on s using the join condition.
- ❑ If indices are available on join attributes of both r and s , use the relation with fewer tuples as the outer relation.

Example of Nested-Loop Join Costs

- ❑ Compute *depositor* ⋈ *customer*, with *depositor* as the outer relation.
- ❑ Let *customer* have a primary B⁺-tree index on the attribute *ID*, which contains 20 entries in each index node.
- ❑ Since *customer* has 10,000 tuples, the height of the tree is 4, and one more access is needed to find the actual data
- ❑ *depositor* has 5000 tuples
- ❑ Cost of block nested loops join
 - $400 * 100 + 100 = 40,100$ block transfers + $2 * 100 = 200$ seeks
 - Assuming worst case memory
 - 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.

(4) Merge-Join (sort-merge join, 排序归并连接)

1. Sort both relations on their join attribute (if not already sorted on the join attributes).
2. Merge the sorted relations to join them
 1. Join step is similar to the merge stage of the sort-merge algorithm.
 2. Main difference is handling of duplicate values in join attribute — every pair with same value on join attribute must be matched
 3. Detailed algorithm in book



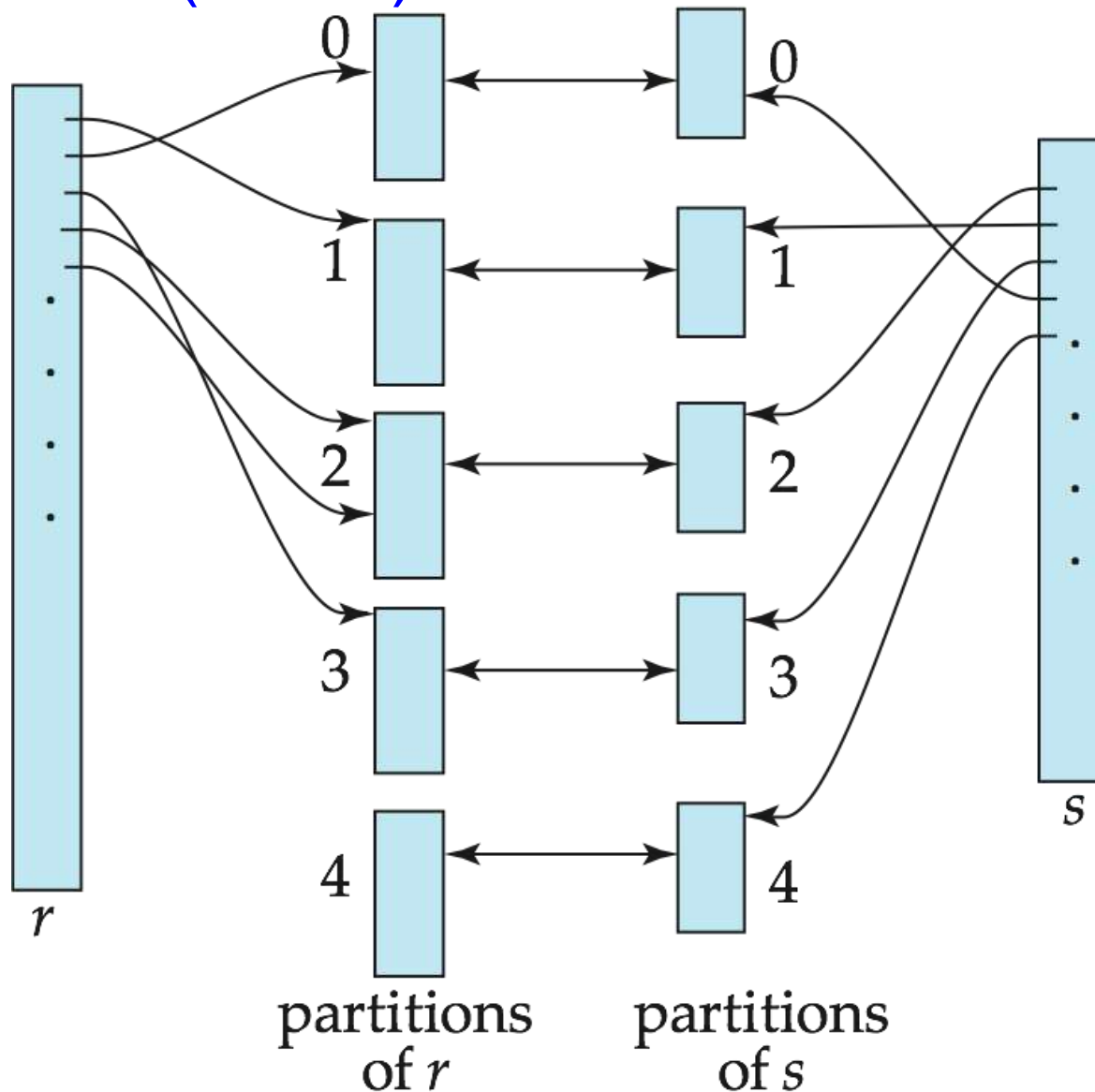
Merge-Join (Cont.)

- ❑ Can be used only for equi-joins and natural joins
- ❑ Each block needs to be read only once (assuming all tuples for any given value of the join attributes fit in memory)
- ❑ Thus the cost of merge join is:
$$b_r + b_s \text{ block transfers} + \lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil \text{ seeks}$$
 - + the cost of sorting if relations are unsorted.
- ❑ **Hybrid merge-join** (混合归并连接) If one relation is sorted, and the other has a secondary B⁺-tree index on the join attribute
 - Merge the sorted relation with the leaf entries of the B⁺-tree.
 - Sort the result on the addresses of the unsorted relation's tuples
 - Scan the unsorted relation in physical address order and merge with previous result, to replace addresses by the actual tuples
 - Sequential scan more efficient than random lookup

(5) Hash-Join (哈希连接)

- ❑ Applicable for equi-joins and natural joins.
- ❑ A hash function h is used to partition tuples of both relations
- ❑ h maps $JoinAttrs$ values to $\{0, 1, \dots, n\}$, where $JoinAttrs$ denotes the common attributes of r and s used in the natural join.
 - r_0, r_1, \dots, r_n denote partitions of r tuples
 - Each tuple $t_r \in r$ is put in partition r_i where $i = h(t_r[JoinAttrs])$.
 - s_0, s_1, \dots, s_n denotes partitions of s tuples
 - Each tuple $t_s \in s$ is put in partition s_i , where $i = h(t_s[JoinAttrs])$.
- ❑ Note: In book, r_i is denoted as H_{ri} , s_i is denoted as H_{si} and n is denoted as n_h .

Hash-Join (Cont.)



Hash-Join (Cont.)

- ❑ r tuples in r_i need only to be compared with s tuples in s_i
Need not be compared with s tuples in any other partition, since:
 - An r tuple and an s tuple that satisfy the join condition will have the same value for the join attributes.
 - If that value is hashed to some value i , the r tuple has to be in r_i and the s tuple in s_i .

Hash-Join Algorithm

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

1. Partition the relation s using hashing function h . When partitioning a relation, one block of memory is reserved as the output buffer for each partition.
2. Partition r similarly.
3. For each i :
 - (a) Load s_i into memory and build an in-memory hash index on it using the join attribute. This hash index uses a different hash function than the earlier one h .
 - (b) Read the tuples in r_i from the disk one by one. For each tuple t_r locate each matching tuple t_s in s_i using the in-memory hash index. Output the concatenation of their attributes.

Relation s is called the **build input**, and r is called the **probe input**.

Hash-Join algorithm (Cont.)

- ❑ The value n and the hash function h is chosen such that each s_i should fit in memory.
 - Typically n is chosen as $\lceil b_s/M \rceil * f$ where f is a “fudge factor”, typically around 1.2
 - The probe relation partitions r_i need not fit in memory

- ❑ **Recursive partitioning** required if the number of partitions n is greater than the number of pages M of memory.
 - instead of partitioning n ways, use $M - 1$ partitions for s
 - Further partition the $M - 1$ partitions using a different hash function
 - Use same partitioning method on r
 - Rarely required: e.g., with block size of 4 KB, recursive partitioning not needed for relations of < 1GB with memory size of 2MB, or relations of < 36 GB with memory of 12 MB

*Handling of Overflows

- ❑ Hash-table overflow occurs in partition Hs_i , if Hs_i does not fit in memory. Reasons could be
 - Many tuples in s with same value for join attributes (分区不均匀)
 - Bad hash function
- ❑ Partitioning is said to be skewed(有偏的) if some partitions have significantly more tuples than some others.
- ❑ Solutions:
 - Typically n_h is chosen as $\lceil b_s/M \rceil * f$, where f is a “fudge factor”, typically around 1.2
 - **Overflow resolution(溢出分解)** can be done in build phase
 - Partition Hs_i is further partitioned using different hash function.
 - Partition Hr_i must be similarly partitioned.
 - **Overflow avoidance(溢出避免)** performs partitioning carefully to avoid overflows during build phase
 - E.g., partition build relation into many partitions, then combine them
 - If both approaches fail with large numbers of duplicates
 - Fallback option: use block nested loops join on overflowed

Cost of Hash-Join

- ❑ If recursive partitioning is not required: cost of hash join is

$$3(b_r + b_s) + 4 * n_h \text{ block transfers}$$

(其中分区代价 $2(b_r+b_s)$, 连接代价 b_r+b_s , 连接不考虑写操作

考虑到每个分区块不满, 多算一块, 从而每个表读写多 $2n_h$, 共 $4n_h$)

$$2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil) // \text{分区} + 2 n_h \text{ seeks} // \text{连接}$$

- ❑ If recursive partitioning required:

- number of passes required for partitioning build relation s is $\lceil \log_{M-1}(b_s) - 1 \rceil$
- best to choose the smaller relation as the build relation.
- Total cost estimate is:

$$2(b_r + b_s) \lceil \log_{M-1}(b_s) - 1 \rceil + b_r + b_s \text{ block transfers} + \\ 2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil) \lceil \log_{M-1}(b_s) - 1 \rceil + 2 n_h \text{ seeks}$$

- ❑ If the entire build input can be kept in main memory no partitioning is required

- Cost estimate goes down to $b_r + b_s$. – best case

Example of Cost of Hash-Join

Depositor ⋈ *customer*

- ❑ Assume that memory size is 20 + 2 blocks
- ❑ $b_{\text{depositor}} = 100$ and $b_{\text{customer}} = 400$.
- ❑ *Depositor* is to be used as build input. Partition it into five partitions, each of size 20 blocks. This partitioning can be done in one pass.
- ❑ Similarly, partition *customer* into five partitions, each of size 80. This is also done in one pass.
- ❑ Therefore total cost, ignoring cost of writing partially filled blocks:
 - $3(100 + 400) = 1500$ block transfers +
 $2(\lceil 100/3 \rceil + \lceil 400/3 \rceil) + 2 * 5 = 346$ seeks

* Hybrid Hash–Join

- ❑ Useful when memory sized are relatively large, and the build input is bigger than memory.
- ❑ **Main feature of hybrid hash join:**
 - Keep the first partition of the build relation in memory.**
- ❑ E.g., With memory size of 25 blocks, *instructor* can be partitioned into five partitions, each of size 20 blocks.
 - Division of memory:
 - The first partition occupies 20 blocks of memory
 - 1 block is used for input, and 1 block each for buffering the other 4 partitions.
- ❑ *teaches* is similarly partitioned into five partitions each of size 80
 - the first is used right away for probing, instead of being written out
- ❑ 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 $M \gg \sqrt{b_s}$

(6) Complex Joins

❑ Join with a conjunctive condition:

$$r \bowtie_{\theta_1 \wedge \theta_2 \wedge \dots \wedge \theta_n} s$$

- 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

$$\theta_1 \wedge \dots \wedge \theta_{i-1} \wedge \theta_{i+1} \wedge \dots \wedge \theta_n$$

❑ Join with a disjunctive condition

$$r \bowtie_{\theta_1 \vee \theta_2 \vee \dots \vee \theta_n} s$$

- Either use nested loops/block nested loops, or
- Compute as the union of the records in individual joins $r \bowtie_{\theta_i} s$:

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

Outline

- ❑ Overview
- ❑ Measures of Query Cost
- ❑ Selection Operation
- ❑ Sorting
- ❑ Join Operation
- ❑ *Other Operations
- ❑ *Evaluation of Expressions



*Other Operations

❑ **Duplicate elimination** can be implemented via hashing or sorting.

- On sorting duplicates will come adjacent to each other, and all but one set of duplicates can be deleted.
- *Optimization*: duplicates can be deleted during run generation as well as at intermediate merge steps in external sort-merge.
- Hashing is similar – duplicates will come into the same bucket.

❑ **Projection**:

- Perform projection on each tuple
- Followed by duplicate elimination.

Other Operations: Aggregation

❑ **Aggregation** can be implemented in a manner similar to duplicate elimination.

- Sorting or hashing can be used to bring tuples in the same group together, and then the aggregate functions can be applied on each group.
- *Optimization*: combine tuples in the same group during run generation and intermediate merges, by computing partial aggregate values
 - For count, min, max, sum: keep aggregate values on tuples found so far in the group.
 - When combining partial aggregate for count, add up the aggregates
 - For avg, keep sum and count, and divide sum by count at the end

Other Operations : Set Operations

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

Other Operations: Outer Join

- ❑ 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 $r \bowtie_{\text{outer}} s$
 - In $r \bowtie_{\text{outer}} s$, non participating tuples are those in $r - \Pi_R(r \bowtie s)$
 - Modify merge-join to compute $r \bowtie_{\text{outer}} s$: 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.
- ❑ Modifying hash join to compute $r \bowtie_{\text{outer}} s$
 - If r is probe relation, output non-matching r tuples padded with nulls
 - If r is build relation, when probing keep track of which r tuples matched s tuples. At end of s_i output non-matched r tuples padded with nulls

Outline

- ❑ Overview
- ❑ Measures of Query Cost
- ❑ Selection Operation
- ❑ *Sorting
- ❑ Join Operation
- ❑ *Other Operations
- ❑ *Evaluation of Expressions



Evaluation of Expressions

- ❑ So far: we have seen algorithms for individual operations. Now consider multi operations.

- $\Pi_{customer-name} ((\sigma_{balance < 2500}(account)) \bowtie depositor)$

- ❑ Alternatives for evaluating an entire expression tree

- Materialization(实体化): generate the results of an expression, and store the results to disk.
 - Pipelining (流水线) : pass on tuples to parent operations even as an operation is being executed

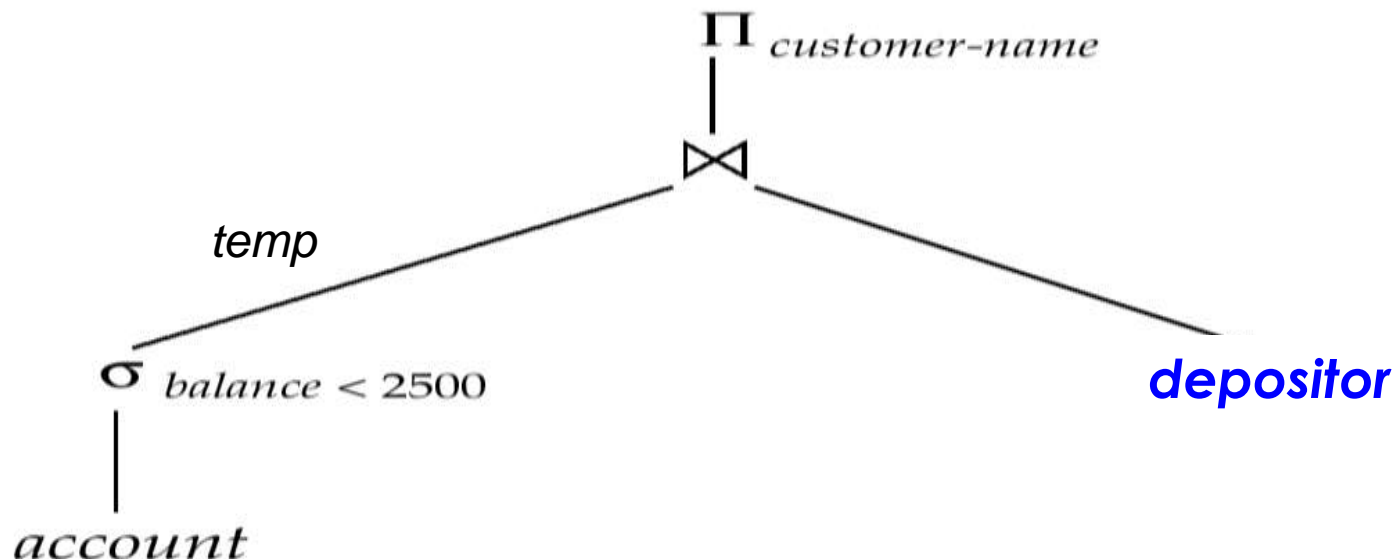
- ❑ We study above alternatives in more detail

(1) Materialization

- ❑ **Materialized evaluation:** evaluate one operation at a time, starting at the lowest-level. Use intermediate results materialized into temporary relations to evaluate next-level operations.
- ❑ E.g., in figure below, compute and store

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

then get back the stored result and join with *depositor*, and finally compute the projections on *customer-name*.

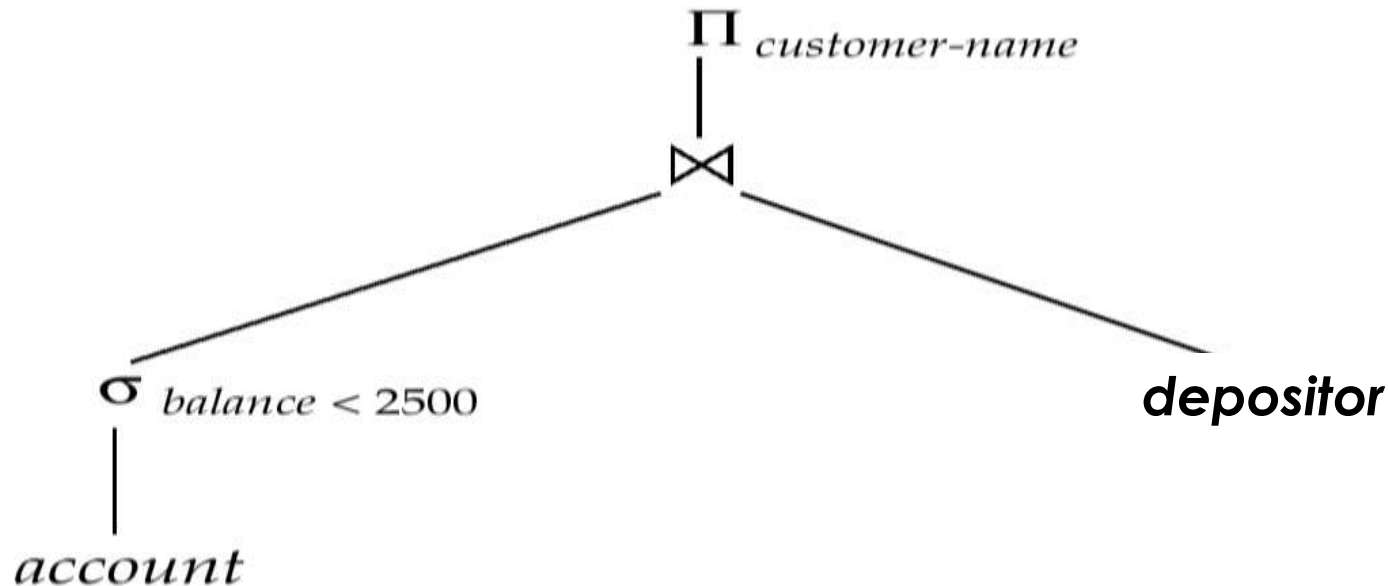


Materialization (Cont.)

- ❑ Materialized evaluation is always applicable
- ❑ Cost of writing results to disk and reading them back can be quite expensive.
 - Our cost formulas for operations ignore cost of writing results to disk, so
 - Overall cost = Sum of costs of individual operations + cost of writing intermediate results to disk
- ❑ (改进) Double buffering: use two output buffers for each operation, when one is full write it to disk while the other is getting filled
 - Allows overlap of disk writes with computation and reduces execution time

(2) Pipelining

- ❑ Pipelined evaluation: evaluate several operations simultaneously, passing the results of one operation on to the next.
- ❑ E.g., in previous expression tree, don't store result of instead, pass tuples directly to the join. Similarly, don't store result of join, pass tuples directly to projection.



(2) Pipelining (cont.)

- ❑ Much cheaper than materialization: no need to store a temporary relation to disk.
- ❑ Pipelining may **not** always be possible. (depend on the type of next operations, and whether the output is sorted, etc.)
- ❑ For pipelining to be effective, use evaluation algorithms that generate output tuples even as tuples are received for inputs to the next operation.
- ❑ Pipelines can be executed in two ways: demand driven and producer driven

(需求驱动流水线、生产者驱动流水线)

由上而下吸引

由下而上的推动

*Pipelining (Cont.)

❑ In **demand driven** or **lazy** evaluation

- System repeatedly requests next tuple from top level operation
- Each operation requests next tuple from children operations as required, in order to output its next tuple
- In between calls, operation has to maintain “**state**” so it knows what to return next
- Each operation is implemented as an **iterator** (迭代器) implementing the following operations

*Pipelining (Cont.)

❑ Open()

- E.g., **file scan**: initialize file scan, store pointer to beginning of file as state
- E.g., **merge join**: sort relations and store pointers to beginning of sorted relations as state

❑ Next()

- E.g., **for file scan**: output next tuple, and advance and store file pointer
- E.g., **for merge join**: continue with merge from earlier state till next output tuple is found. Save pointers as iterator state.

❑ Close()

*Pipeline (Cont.)

❑ In **produce-driven** or **eager** pipelining

- Operators produce tuples eagerly and pass them up to their parents
 - Buffer maintained between operators, child puts tuples in buffer, parent removes tuples from buffer
 - If buffer is full, child waits till there is space in the buffer, and then generates more tuples

- Systems schedules operations that have space in output buffer and can process more input tuples

*Evaluation Algorithms for Pipelining

- ❑ Some algorithms are not able to output results even as they get input tuples
 - E.g. merge join, or hash join
 - These results in intermediate results being written to disk and then read back always
- ❑ Algorithm variants are possible to generate(at least some) results on the fly, as input tuples are read in
 - E.g. hybrid hash join generates output tuples even as probe relation tuples in the in-memory partition (partition 0) are read in
 - **Double-pipelined join technique**: Hybrid hash join, modified to buffer partition 0 tuples of both relations in-memory, reading them as they become available, and output results of any matches between partition 0 tuples
 - When a new r_0 tuple is found, match it with existing s_0 tuples, output matches, and save it in r_0
 - Symmetrically for s_0 tuples

*Complex Joins

- ❑ Join involving three relations: $loan \bowtie depositor \bowtie customer$
- ❑ **Strategy 1.** Compute $depositor \bowtie customer$; use result to compute $loan \bowtie (depositor \bowtie customer)$
- ❑ **Strategy 2.** Compute $loan \bowtie depositor$ first, and then join the result with $customer$.
- ❑ **Strategy 3.** Perform the pair of joins at once. Build and index on $loan$ for $loan\text{-}number$, and on $customer$ for $customer\text{-}name$.
 - For each tuple t in $depositor$, look up the corresponding tuples in $customer$ and the corresponding tuples in $loan$.
 - Each tuple of $deposits$ 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.

Example

□ Given tables:

T1(bno, type, title, press, year);

T2(id, bno, date1, date2); // bno is foreign key referencing T1

T1 has 20,000 tuples, T2 has 45,000 tuples. 25 tuples of T1 fit in one block, 30 tuples of T2 fit in one block.

T1 has a primary B+-tree index on primary key attribute bno, which contains maximal 60 entries in each index node(l.t. n=60). Assume the allowed memory is in worst case. Compute the cost of the join operation for T1 and T2:

(1) Block nested-loop join

(2) index nested-loop join

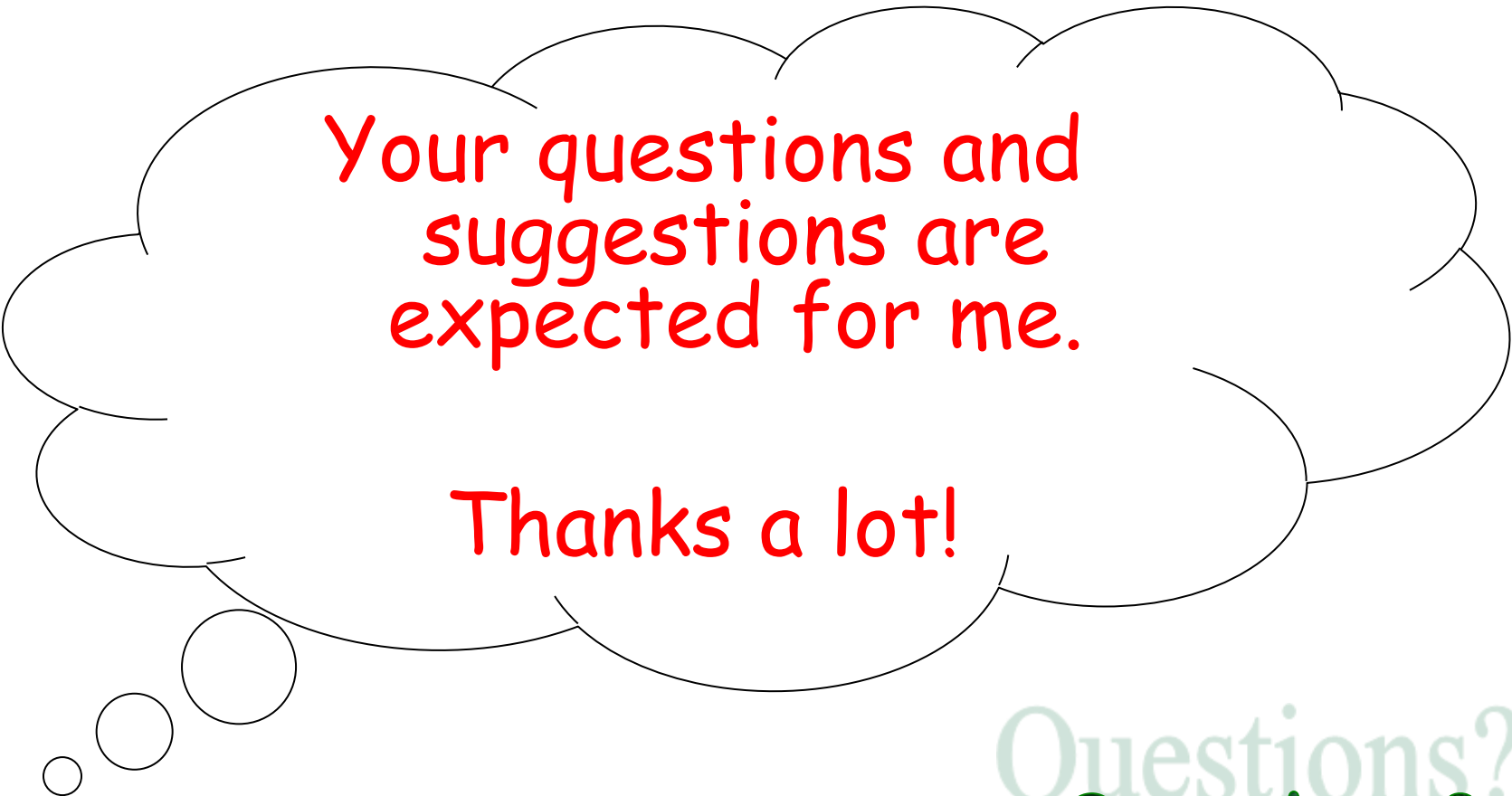
$$b_1 = 20000/25 = 800, b_2 = 45000/30 = 1500, \log_{60/2} 20000 \approx 3$$

$$(1) \text{ BT costs} = 800 \times 1500 + 800 = 1200800, \text{ seeks} = 2 \times 800$$

$$(2) \text{ BT costs} = 45000 \times (3 + 1) + 1500 = 18150, \text{ seeks} = 18150$$

(T2 is outer, T1 is inner relation.)

Q & A



Your questions and
suggestions are
expected for me.

Thanks a lot!

Exercises: 15.2, 15.3, 15. 6, and
15.20 (see Pages 736-739)

Questions?
Questions?