Introduction to Databases
《数据库引论》

Lecture 9: Query Processing
第9讲：查询处理

周水庚 / Shuigeng Zhou

邮件: sgzhou@fudan.edu.cn 网址：admis.fudan.edu.cn/sgzhou

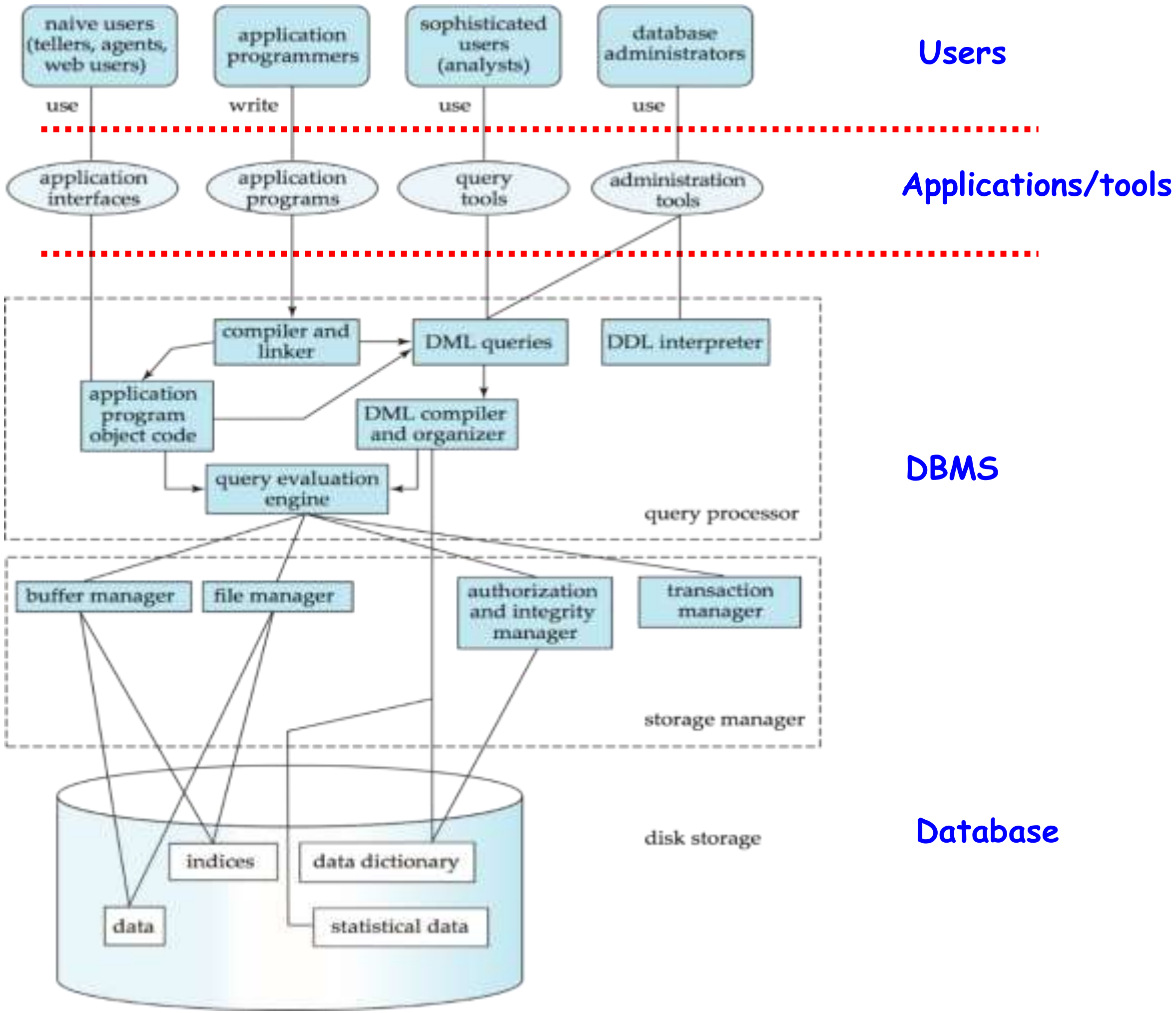
复旦大学计算机科学技术学院

Outline of the Course

- **Part 0: Overview**
  - Lect. 1 (Feb. 29) - Ch1: Introduction
- **Part 1 Relational Databases**
  - Lect. 2 (Mar. 7) - Ch2: Relational model (data model, relational algebra)
  - Lect. 3 (Mar. 14) - Ch3: SQL (Introduction)
  - Lect. 4 (Mar. 21) – Ch4/5: Intermediate and Advanced SQL
- **Part 2 Database Design**
  - Lect. 5 (Mar. 28) - Ch6: Database design based on E-R model
  - **Apr. 4 (Tomb-Sweeping Day): no course**
  - Lect. 6 (Apr. 11/18) - Ch7: Relational database design
- **Midterm exam: Apr. 25**
  - 13: 00-15: 00, H3109
- **Part 3 Data Storage & Indexing**
  - Lect. 7 (May 2 -> Apr. 28) - Ch12/13: Storage systems & structures
  - Lect. 8 (May 10) - Ch14: Indexing and Hashing
- ☞ **Part 4 Query Processing & Optimization**
  - Lect. 9 (May 17) - Ch15: Query processing
  - Lect. 10 (May 24) - Ch16: Query optimization
- **Part 5 Transaction Management**
  - Lect. 11 (May 31) - Ch17: Transactions
  - Lect. 12 (Jun. 7) - Ch18: Concurrency control
  - Lect. 13 (Jun. 14) - Ch19: Recovery system

Final exam: 13:00-15:00, Jun. 26

Database System Structure

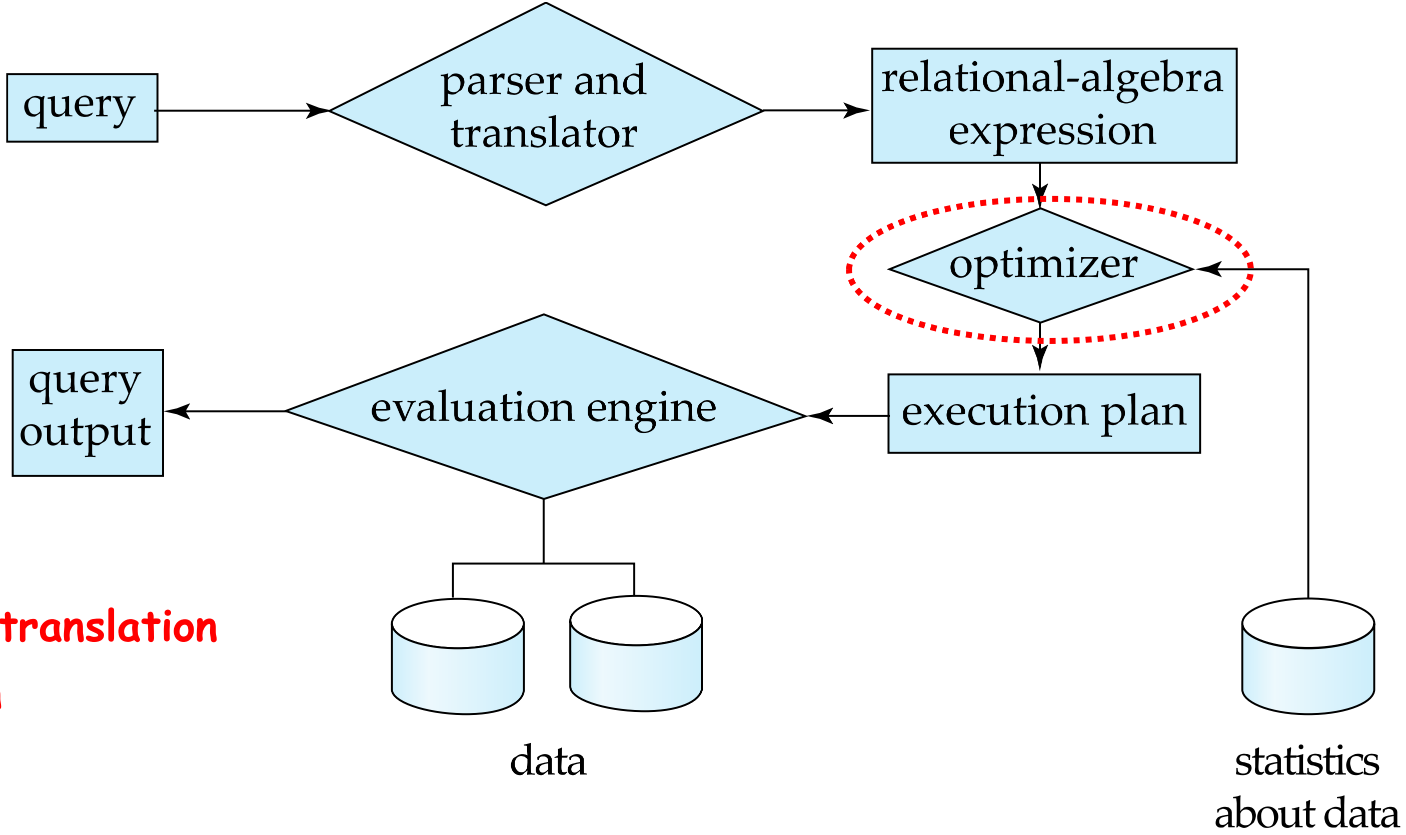


Outline

☞ **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

- **Parsing and translation**
  - translate the query into the internal form which is then translated into relational algebra
- **Optimization**
  - Generate the optimal execution plan (执行计划)
- **Execution**
  - The query execution engine executes the evaluation plan, and returns the answers to the query

Query Optimization

```
select salary
from instructor
where salary < 75000
```

给出对应的关系代数表达式

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

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

Query Optimization

- 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*

Query Optimization (Cont.)

- **Query Optimization**
  - Amongst all equivalent evaluation plans, choose the one with **lowest cost**
  - Cost is estimated using statistical information from the database catalog
- **This lecture**
  - How to measure query costs
  - Algorithms for evaluating relational algebra operations
  - Combine algorithms for individual operations to evaluate a complete expression
- **Next lecture**
  - The way to find an execution plan with the lowest estimated cost

Outline

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

Measures of Query Cost

- Cost is generally measured as total elapsed time for answering query
  - disk accesses, CPU, and even network communication
- Typically disk access is the predominant cost, and is also relatively easy to be estimated
- **Disk access** is measured by taking into account
  - **Number of seeks**
  - **Number of blocks read**
  - **Number of blocks written**
    - The cost to write a block is greater than the 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, 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

Outline

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

Selection Operation

- **File scan (文件扫描)**
  - Search algorithms that locate and retrieve records that satisfy a selection condition

- **Index scan (索引扫描)**
  - Search algorithms that use an index
  - Selection condition must be on search-key of an index

Selection Operation

- **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

Selection Operation (Cont.)

- ***A1' (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
  - Cost estimate (number of disk blocks to be scanned):
    - cost of locating the first tuple by a binary search on the blocks
      - **worst cost $\lceil \log_2(b_r) \rceil * (t_T + t_S)$**
    - If there are multiple records satisfying selection
      - *Add transfer cost of the* number of blocks containing records that satisfy selection condition

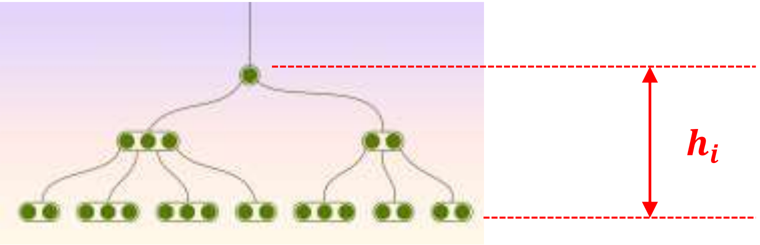
Selections Using Indices

- **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 = $h_i * (t_T + t_S) + t_S + t_T * b$**

Selections Using Indices (Cont.)

- **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

B+ tree



Selections Involving Comparisons

- 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

Selections Involving Comparisons (Cont.)

- **A6 (secondary index, comparison).**
  - 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 $> v$
  - In either case, retrieve records that are pointed to
    - requires an I/O for each record
    - Linear file scan may be cheaper if many records are to be fetched!

Selection Operation Cost Estimation

	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	$h_i * (t_T + t_S) + t_S + t_T * b$	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

Selection Operation Cost Estimation

	Algorithm	Cost	Reason
A1	Linear Search	$t_S + b_r * t_T$	One initial seek plus b_r block transfers, where b_r denotes the number of blocks in the file.
A1	Linear Search, Equality on Key	Average case $t_S + (b_r/2) * t_T$	Since at most one record satisfies the condition, scan can be terminated as soon as the required record is found. In the worst case, b_r block transfers are still required.
A2	Clustering B⁺-tree Index, Equality on Key	$(h_i + 1) * (t_T + t_S)$	(Where h_i denotes the height of the index.) Index lookup traverses the height of the tree plus one I/O to fetch the record; each of these I/O operations requires a seek and a block transfer.
A3	Clustering B⁺-tree Index, Equality on Non-key	$h_i * (t_T + t_S) + t_S + b * t_T$	One seek for each level of the tree, one seek for the first block. Here b is the number of blocks containing records with the specified search key, all of which are read. These blocks are leaf blocks assumed to be stored sequentially (since it is a clustering index) and don't require additional seeks.
A4	Secondary B⁺-tree Index, Equality on Key	$(h_i + 1) * (t_T + t_S)$	This case is similar to clustering index.
A4	Secondary B⁺-tree Index, Equality on Non-key	$(h_i + n) * (t_T + t_S)$	(Where n is the number of records fetched.) Here, cost of index traversal is the same as for A3, but each record may be on a different block, requiring a seek per record. Cost is potentially very high if n is large.
A5	Clustering B⁺-tree Index, Comparison	$h_i * (t_T + t_S) + t_S + b * t_T$	Identical to the case of A3, equality on non-key.
A6	Secondary B⁺-tree Index, Comparison	$(h_i + n) * (t_T + t_S)$	Identical to the case of A4, equality on non-key.

Implementation of Complex Selections

- **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

Implementation of Complex Selections (Cont.)

- **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

Outline

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

Sorting

- We can build an index on the relation, and then use the index to read the relation in sorted order.
  - May lead to one disk block access for each tuple (for non-primary indices)
- **Relations that fit in memory**
  - Techniques like **quick-sort (快速排序)** can be used
- **Relations that don't fit in memory**
  - **External sort-merge (外部排序归并)** is a good choice

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（selection sort）：O(n^2)
  - **快速排序（quicksort）**：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 （heapsort）：O(nlogn)
  - 希尔排序 （shell sort）：O(nlogn)
  - 基数排序（radix sort）：O(n·k); 需要 O(n) 额外存储空间 （K为特征个数）

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（insertion sort）：$O(n^2)$
  - 冒泡排序（bubble sort）：$O(n^2)$
  - **归并排序 （merge sort）**：O(n log n); 需要 O(n) 额外存储空间
  - 二叉树排序（Binary tree sort）：O(nlogn); 需要 O(n) 额外存储空间
  - 计数排序 (counting sort) : O(n+k); 需要 O(n+k) 额外存储空间，k为序列中Max-Min+1
  - 桶排序 （bucket sort）：O(n); 需要 O(k) 额外存储空间

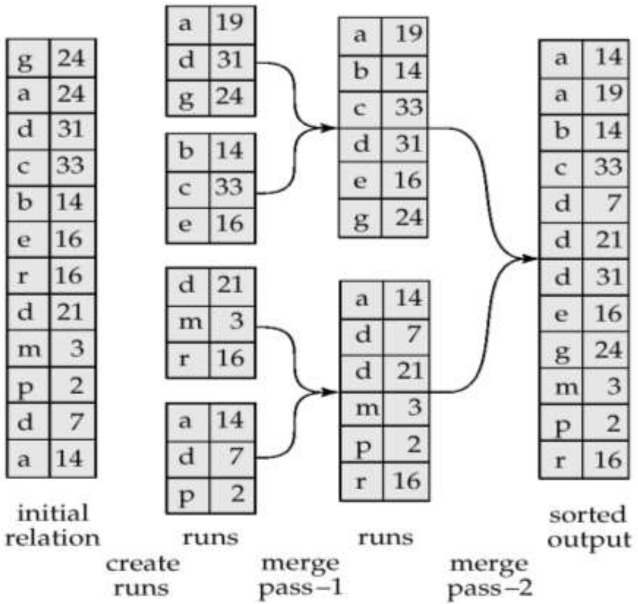
External Sort-Merge (外部排序归并)

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

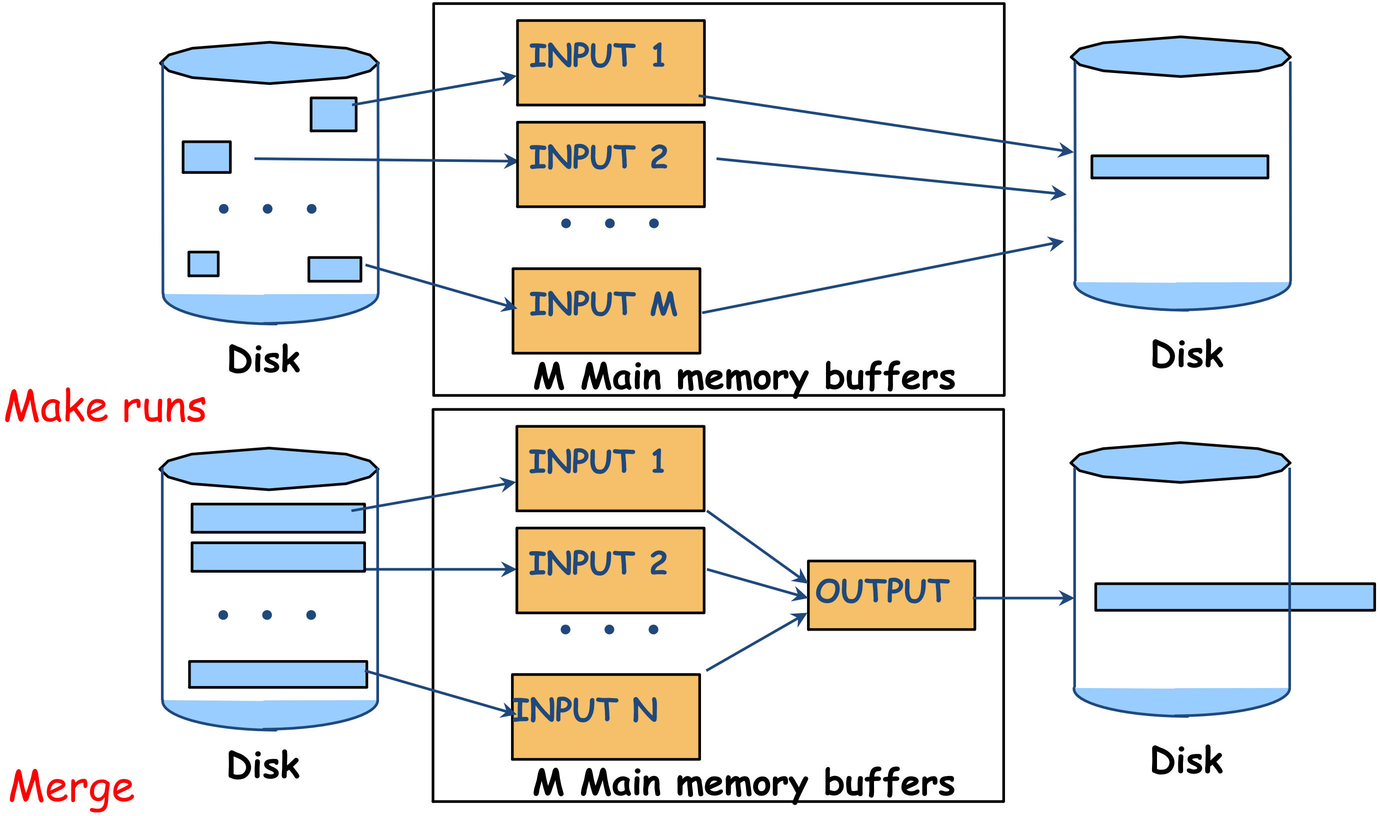
  let $i = 0$ *initially*
  repeatedly do the following till the end of the relation:
  read M blocks of relation into memory
  sort the in-memory blocks
  write sorted data to run R_i
  increment i
  let the final value of $i = N$ (N-way merge)
- **Merge the runs** (next slide)

Assume:

1. Only one tuple fits in a block
2. Memory holds at most 3 blocks, 2 for input and 1 for output



External Sort-Merge



External Sort-Merge (cont.)

- **Merge the runs (N-way merge，N路归并).** We assume N < M
  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

  repeat

  select the first record (in sort order) among all buffer blocks

  write the record to the output buffer block. If the output buffer is full, write it to disk

  delete the record from the input buffer block
  If the buffer block becomes empty then
  read the next block of the run into the buffer

  ***until** all input buffer blocks are empty*

External Sort-Merge (Cont.)

- If $N \geq M$, **several merge passes (多趟归并)** are required
  - In each pass, contiguous groups of $M - 1$ runs are merged.
  - A pass reduces the number of runs by a factor of $M - 1$.
    - E.g. If M=11, and there are 90 runs, one pass reduces the number of runs to 9, each 10 times the size of the initial runs
  - Repeated passes are performed till all runs have been merged into one

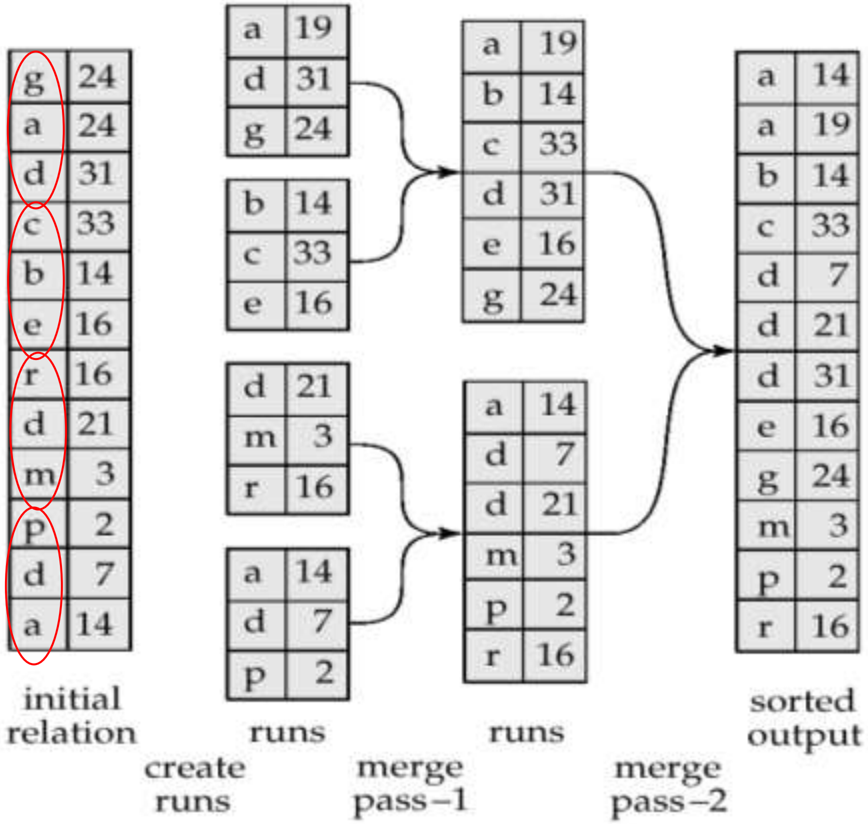
Example: External Sorting using Sort-Merge

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:** $12(2*\log_2(12\ /\ 3)+1)\ =60$



External Merge Sort (Cont.)

- Cost analysis:
  - Let b_r denote the number of blocks containing records of relation r
  - The initial number of runs is $\lceil b_r/M \rceil$
  - Total number of merge passes required: $\lceil \log_{M-1}(b_r/M) \rceil$.
  - Disk accesses for initial run creation as well as in each pass is $2b_r$ (read in + write out)
    - 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.
  - Each pass (except the final pass) reads every block once and writes out once. Thus total number of disk accesses for external sorting:
    - $b_r (2 \lceil \log_{M-1}(b_r / M) \rceil + 1)$:

    Example: $12(2*\log_2(12 / 3)+1) =60$

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:

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

    - Applying the equation to the example, we get:

$$2*(12/3)+(12/1)(2* \log_2 (12 / 3)-1) =8+12*3 = 44 \text{ seeks}$$

Outline

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

Join Operation

- Algorithms to implement joins
  - Nested-loop join (嵌套循环连接)
  - Block nested-loop join (块嵌套循环连接)
  - Indexed nested-loop join (索引嵌套循环连接)
  - Merge-join (归并连接)
  - Hash-join (散列连接)
- Examples use the following information
  - #records
    - customer: 10000
    - depositor: 5000
  - #blocks
    - customer: 400
    - depositor: 100

Nested-Loop Join (嵌套循环连接)

- 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

#records
- customer: 10000
- depositor: 5000

#blocks
- customer: 400
- depositor: 100

Nested-Loop Join (Cont.)

- Given the worst case memory availability, the cost estimate is

  $n_r * b_s + b_r$ **block transfers** **plus** $n_r + b_r$ **seeks**

  - 5000 * 400 + 100 = 2,000,100 disk accesses with depositor as outer relation, and 5000 + 100 = 5100 seeks
  - 10000 * 100 + 400 = 1,000,400 disk accesses with customer as the outer relation, and 10,400 seeks
  - **较小的关系做内层更优**
  - If smaller relation (depositor) fits entirely in memory, the cost estimate will be 500 disk accesses，这时较小的关系做内层更优

#records

customer: 10000

depositor: 5000

#blocks

customer: 400

depositor: 100

Block Nested-Loop Join (块嵌套循环连接)

- 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

Block Nested-Loop Join (Cont.)

- **Worst case estimate:** $b_r * b_s + b_r$ **block transfers** + $2 * b_r$ **seeks**
  - 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)
  - 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
- Eg. Cost of block nested loops join
  - 100*400 +100 = 40,100 block transfers + 2 * 100 =200 seeks
- **Best case(内存能容纳内层关系，较小的关系做内层):** $b_r + b_s$ **block transfers + 2 seeks**

#records

customer: 10000

depositor: 5000

#blocks

customer: 400

depositor: 100

Block Nested-Loop Join (Cont.)

- **Improvements to nested loop and block nested loop algorithms:**
  - In block nested-loop, 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
    - Cost = $\lceil b_r / (M-2) \rceil * b_s + b_r$ block transfers + $2 \lceil b_r / (M-2) \rceil$ seeks
  - If equi-join attribute forms a key of inner relation, stop inner loop on first match
  - Scan inner relation forward and backward alternately, to make use of the blocks remaining in buffer (with LRU replacement)
  - Use index on inner relation if available (next slide)

Indexed Nested-Loop Join (索引嵌套循环连接)

- 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 Indexed Nested-Loop Join Costs

- **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

#records

customer: 10000

depositor: 5000

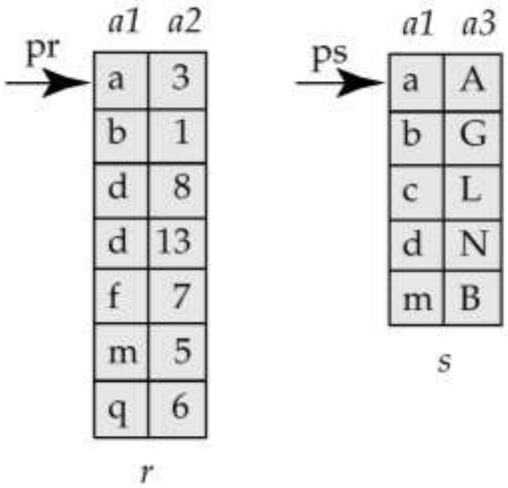
#blocks

customer: 400

depositor: 100

Merge-Join* (归并连接)

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



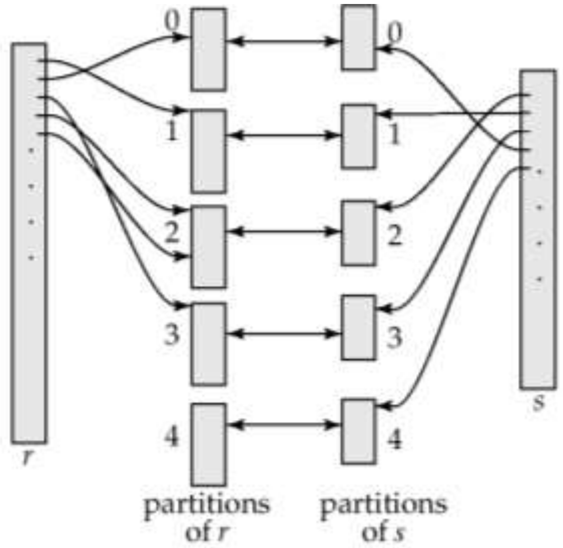
Merge-Join (Cont.)

- 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 number of block accesses for **merge-join** is

  $b_r + b_s$ **+ the cost of sorting** if relations are unsorted.
- **Hybrid merge-join (combining indices with merge-join):** 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, the result file contains tuples from the sorted file and the addresses from the unsorted file
  - Sort the result file 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

Hash-Join* (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**

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:

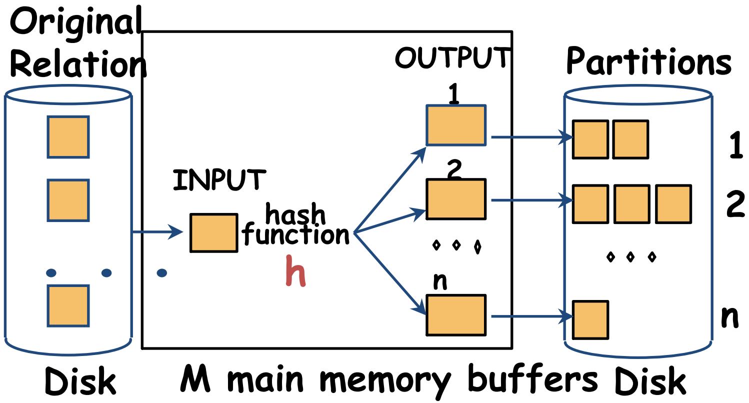
   (a) Load s_i into memory and build an in-memory hash index on it using the join attribute. This hash index uses a different hash function than the earlier one h.

   (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.

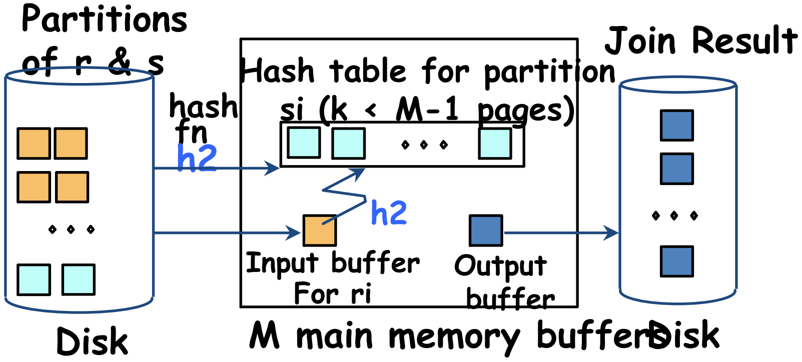
Relation s is called the build input(构造用输入) and r is called the probe input(探查用输入)

Hash-Join Algorithm

- Partition both relations using hash function h: r tuples in partition i will only match s tuples in partition i



- Read in a partition of s, hash it using **h2 (≠ h!)**. Scan matching partition of r, search for matches



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"(避让因子), typically around 1.2
  - The probe relation partitions r_i **need not fit in memory**

- **Recursive partitioning required if number of partitions *n* is greater than 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*

Handling of Overflows

- **Hash-table overflow occurs in partition s_i if s_i does not fit in memory. Reasons could be**
  - Many tuples in s with same value for join attributes
  - Bad hash function
- **Overflow resolution(溢出分解) can be done in build phase**
  - Partition s_i is further partitioned using different hash function.
  - Partition r_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
- **Both approaches fail with large numbers of duplicates**(大量元组链接属性相同)
  - Fallback option: **use block nested loops join** on overflowed partitions

Cost of Hash-Join

- If recursive partitioning is not required: cost of hash join is
  $$2(b_r + b_s) + (b_r + b_s) + 4n$$
- If recursive partitioning required, number of passes required for partitioning s is $\lceil log_{M-1}(b_s) - 1 \rceil$.
- the number of passes for partitioning of r is also the same as for s.
- Therefore **it is best to choose the smaller relation as the build relation.**
- Total cost estimate is:
  $$2(b_r + b_s) \lceil log_{M-1}(b_s) - 1 \rceil + b_r + b_s$$
- If the entire build input can be kept in main memory, n can be set to 0 and the algorithm does not partition the relations into temporary files. Cost estimate goes down to $b_r + b_s$.

Example of Cost of Hash-Join

customer ⋈ depositor

- Assume that memory size is 20 blocks
- $b_{depositor}$= 100 and $b_{customer}$ = 400.
- depositor is to be used as build input. Partition it into 5 partitions, each of size 20 blocks. This partitioning can be done in 1 pass.
- Similarly, partition customer into 5 partitions, each of size 80. This is also done in 1 pass.
- Therefore total cost: 3(100 + 400) = 1500 block transfers
  - ignores cost of writing partially filled blocks

Hybrid Hash-Join

- Useful when memory size 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, depositor can be partitioned into 5 partitions, each of size 20 blocks.
- Division of memory:
  - The first partition occupies 20 blocks of memory（无需递归划分）
  - 1 block is used for input, and 1 block each for buffering the other 4 partitions.

Hybrid Hash-Join

- 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 $M \gg \sqrt{b_s}$

Complex Joins

- **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 $r \bowtie_{\theta_i} s$:

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

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: Computer loan $\bowtie$ 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.

Outline

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

Duplicate Elimination & Projection

- Duplicate elimination can be implemented via hashing or sorting
  - On sorting duplicates will come adjacent to each other, and all but one copy 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 is implemented by performing projection on each tuple followed by duplicate elimination

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.
    - For **avg**, keep sum and count, and divide sum by count at the end

Set Operations

- **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, $r_1, \ldots, r_n$ and $s_1, s_2, \ldots, s_n$
  - 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
    - $\boldsymbol{r \cup s}$: 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
    - $\boldsymbol{r \cap s}$: output tuples in s_i to the result if they are already there in the hash index
    - $\boldsymbol{r - s}$: 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

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 $r ⟕ s$
  - In $r ⟕ s$, non participating tuples are those in $r - \Pi_R(r \bowtie s)$
  - Modify merge-join to compute $r ⟕ s$: 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

Outline

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

Evaluation of Expressions

- We have seen algorithms for individual operations

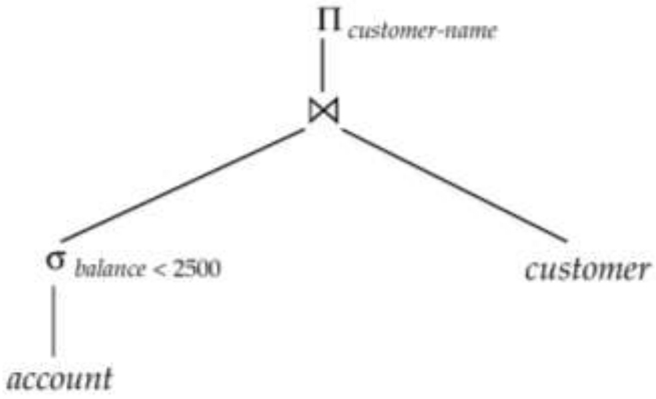
- Alternatives for evaluating an entire expression tree
  - **Materialization (物化)**: generate the results of an expression whose inputs are relations or are already computed, materialize (store) it on disk
  - **Pipelining (流水线)**: pass on tuples to parent operations even as an operation is being executed

Materialization (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.



Materialization (Cont.)

- **Materialized evaluation** (物化计算) is always applicable
- Cost of writing results to disk and reading them back can be quite high
  - **overall cost = sum of costs of individual operations + cost of writing intermediate results to disk**

- **Double buffering (双缓冲):** use two output buffers for each operation, when one is full write it to disk while the other is getting filled
  - Reduce the execution time

Pipelining (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)**

Pipelining (Cont.)

- **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
  - Between calls, operation has to maintain "state" so it knows what to return next
  - Each operation is implemented as an iterator implementing the following operations
    - 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()

Pipelining (Cont.)

- **producer-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
  - System schedule operations that have space in output buffer and can process more input tuples

Assignments

- Practice exercises: 15.3, 15.6
- Exercises: 15.17

- Submission DDL: 12:59pm, May 22

End of Lecture 9