# 查询处理
# Query Processing

李文根/Wengen Li
Email: lwengen@tongji.edu.cn
先进数据与机器智能系统实验室
Advanced Data and Machine Intelligence Systems (ADMIS) Lab
https://admis-tongji.github.io
同济大学 计算机科学与技术学院
2026年05月

# 课程概要

同济大学 TONGJI UNIVERSITY

- **Part 0: Overview**
  - Ch1: Introduction
- **Part 1 Relational Languages**
  - Ch2: Relational model
  - Ch3: Introduction to SQL
  - Ch4: Intermediate SQL
  - Ch5: Advanced SQL
- **Part 2 Database Design**
  - Ch6: Database design via E-R model
  - Ch7: Relational database design
- **Part 3 Application Design & Development**
  - Ch8: Complex data types
  - Ch9: Application development
- **Part 4 Big Data Analytics**
  - Ch10: Big data
  - Ch11: Data analytics
- **Part 5 Storage Management & Indexing**
  - Ch12: Physical storage systems
  - Ch13: Data storage structures
  - Ch14: Indexing
- **Part 6 Query Processing & Optimization**
  - **Ch15: Query processing**
  - Ch16: Query optimization
- **Part 7 Transaction Management**
  - Ch17: Transactions
  - Ch18: Concurrency control
  - Ch19: Recovery system
- **Part 8 Parallel & Distributed Database**
  - Ch20: Database system architecture
  - Ch21-23: Parallel & distributed storage, query processing & transaction processing
- **Advanced topics**
  - DB Platform: **OceanBase**, MongoDB, Neo4J
  - RAG, Multimodal retrieval, …

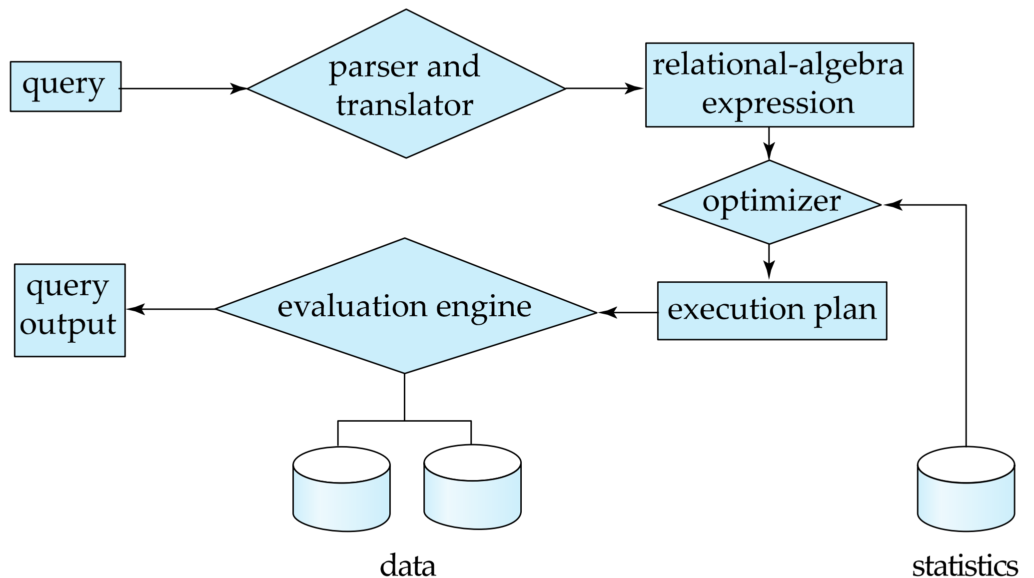
# 目录



- 概述
- 查询代价度量
- 选择操作
- 排序操作
- 连接操作
- 其他操作
- 表达式执行

# 查询处理的基本步骤

- **语法分析与翻译/Parsing and translation**
  - Translate the query into the internal form which is then translated into relational algebra
- **优化/Optimization**
  - Generate the optimal execution plan (执行计划)
- **执行/Execution**
  - The query execution engine executes the execution plan, and returns the answers to the query



# 查询优化



***select*** *salary*
***from*** *instructor*
***where*** *salary* < *75000*

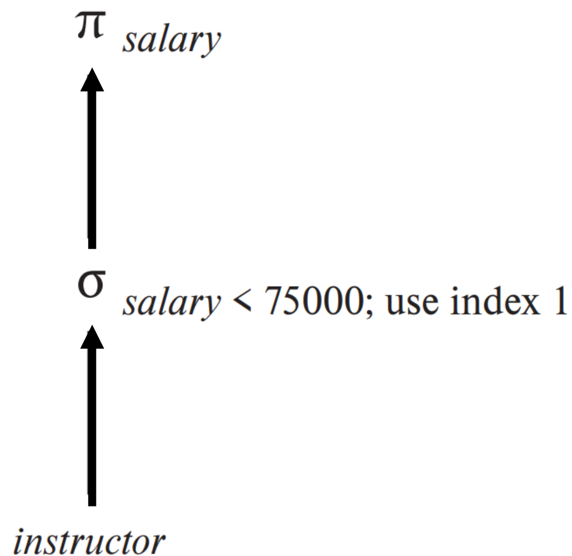
**对应的关系代数表达式**

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

**A relational algebra expression may have many equivalent expressions**

# 查询优化（续）

- **查询执行计划/Query execution plan**
  - Annotated expression specifies detailed execution strategy



# 查询优化（续）



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

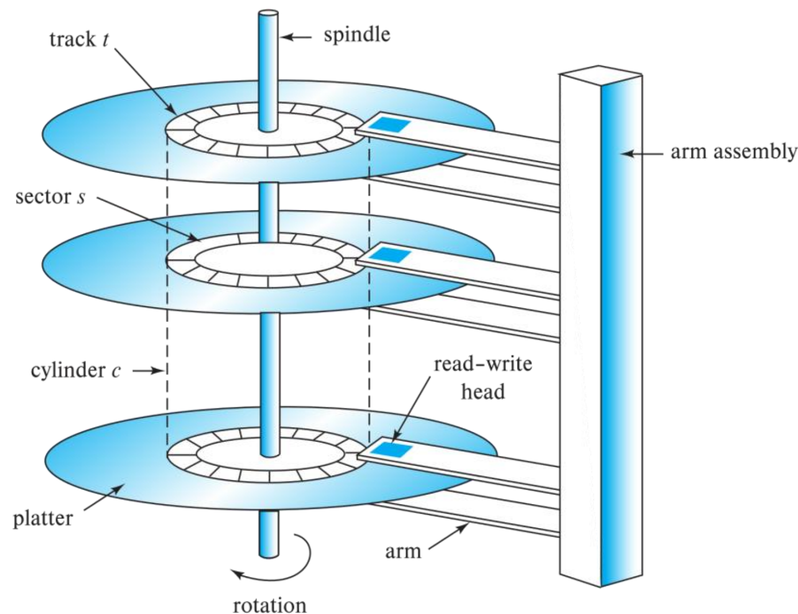
# 目录



- 概述
- 查询代价度量
- 选择操作
- 排序操作
- 连接操作
- 其他操作
- 表达式执行

# 查询代价度量

- Cost is generally measured as the total elapsed time for answering the 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 or written



# 查询代价度量（续）

同濟大學 TONGJI UNIVERSITY

- For simplicity, use the number of block transfers from disk and the number of seeks as the cost measure
- Cost for $b$ block transfers and $s$ 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 the worst case estimates, assuming that only the minimum amount of buffer storage is available

# 目录




- 概述
- 查询代价度量
- 选择操作
- 排序操作
- 连接操作
- 其他操作
- 表达式执行

# 选择操作（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 conditions must be on the search-key of an index

# 选择操作（续）



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

# 使用索引的选择操作



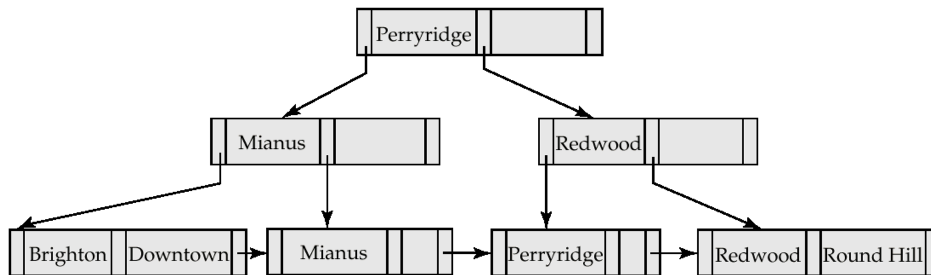
- **A2** (**primary index** on candidate key, equality)
  - Retrieve a single record that satisfies the corresponding equality condition
  - Cost = $(h_i + 1) * (t_T + t_S)$ (B+-tree)

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

- **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
    - cost = $(h_i + n)$ * $(t_T + t_S)$
      - $n$ is the number of records that satisfy the search condition
      - can be very expensive since each record may be on a different block

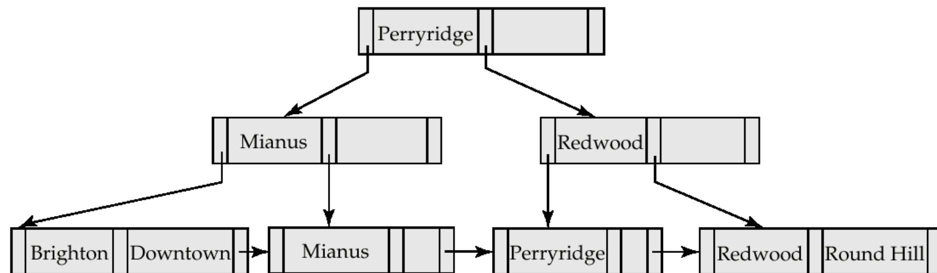
# 包含比较的选择操作

- Selections of the form $\sigma_{A \leq V}(r)$ or $\sigma_{A \geq V}(r)$
  - using a linear file scan or binary search, or
  - using indices in the following ways:
- **A5** (**primary** index, comparison).
  - Relation is sorted on attribute 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 $> V$ and **do not need to use index**



# 使用索引的选择操作（续）

- **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 to find pointers to records, till first entry $> V$
  - In either case
    - requires an I/O for each record
    - linear file scan may be faster if many records are to be fetched!



# 选择操作的代价估计

| | 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. |

# 复杂选择的实现

TONGJI UNIVERSITY

- **Conjunction (合取):** $\boldsymbol{\sigma_{\theta 1 \wedge \theta 2 \wedge \cdots \wedge \theta n}(r)}$
- **A7** (conjunctive selection using one index)
  - select a condition of $\theta_i$ and algorithms A1-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)
  - use the corresponding index for each condition, and take intersection of all the obtained sets of record pointers
  - then fetch records from data file

# 复杂选择的实现（续）

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

- **Negation (取反):** $\boldsymbol{\sigma_{\neg\theta}(r)}$
  - use linear scan on file
  - if few records satisfy $\neg\theta$, and an index is applicable to $\theta$
    - find the required records using index and fetch them from file

# 目录

- 概述
- 查询代价度量
- 选择操作
- 排序操作
- 连接操作
- 其他操作
- 表达式执行

# 排序/Sorting



- 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
  - Quick-sort (快速排序)
- Relations that don’t fit in memory
  - **External sort-merge** (外排序-归并)

# 外排序-归并（External Sort-Merge）

- Let $M$ denote the memory buffer size (in blocks)
- Create sorted runs(归并段)

  $let\ i = 0$
  *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* $R_i$
      $i = i + 1$
  *let the final value of* $i = N$

- Merge the runs (next slide)

# 外排序-归并（续）

- Merge the runs (N-way merge, N路归并) and 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*

# 外排序-归并（续）

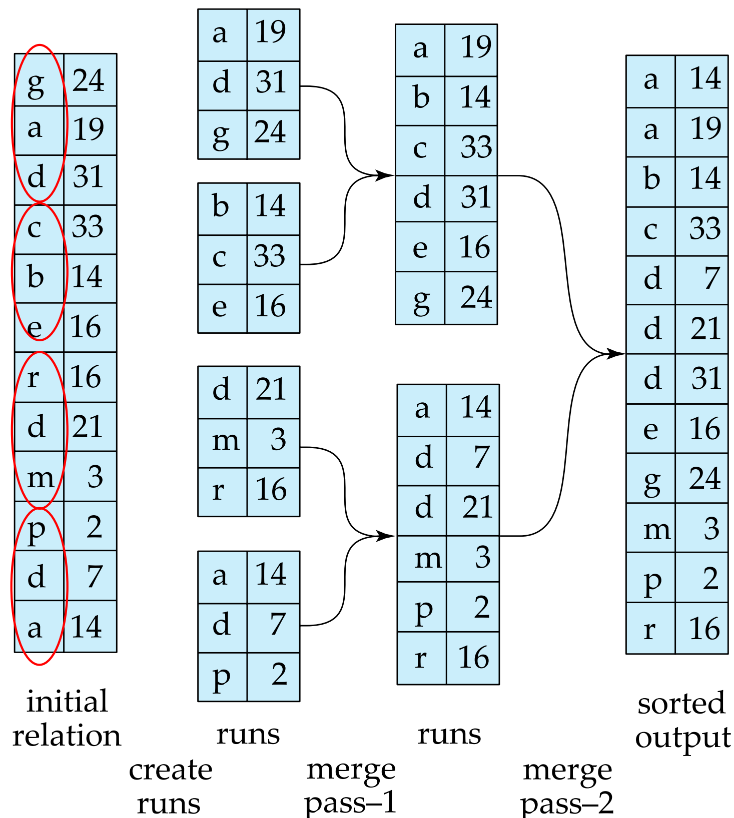
- If $N \geq M$, several **merge passes** (多趟归并) are required
  - In each pass, contiguous groups of M - 1 runs are merged
  - A pass reduces the number of runs by a factor of M -1
    - E.g., if M=11, and there are 90 runs, one pass reduces the number of runs to 9
  - Repeated passes are performed till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

# 外排序-归并（续）

Sort on the first column!

**Assume:**

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



# 外排序-归并（续）



- **Cost analysis**
  - Let $b_r$ denote the number of blocks containing records of relation $r$. The initial number of runs（归并段）is $\lceil b_r/M \rceil$
  - The total number of merge passes required: $\left\lceil log_{\lfloor M/b_b \rfloor -1}(b_r/M) \right\rceil$
    - $b_b$: buffer size for each run
  - Disk accesses for initial run creation as well as in each pass is $2b_r$ (read + write)
    - 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 the total number of disk accesses for external sorting: $b_r(2\left\lceil log_{\lfloor M/b_b \rfloor -1}\left(\frac{b_r}{M}\right) \right\rceil + 1)$
  - E.g., 12*(2*$\log_2$(12 / 3)+1) =60, where $b_b = 1$

# 外排序-归并（续）



- **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 for each run: $b_b$ (read/write $b_b$ blocks for each run 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:
      $2\lceil b_r/M \rceil + \left\lceil \frac{b_r}{b_b} \right\rceil \left(2 \left\lceil log_{\lfloor M/b_b \rfloor - 1}\left(\frac{b_r}{M}\right) \right\rceil - 1\right)$
    - Assume $b_b = 1$, applying the equation to the example, we get: 2*(12/3)+(12/1)(2* $\log_2$ (12 / 3)-1) =8+12*3 = 44 seeks

# 目录

同济大学 TONGJI UNIVERSITY


- 概述
- 查询代价度量
- 选择操作
- 排序操作
- 连接操作
- 其他操作
- 表达式执行

# 连接操作（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
    - student: 5000
    - takes: 10000
  - #blocks
    - student: 100
    - takes: 400

# 嵌套循环连接（Nested-Loop Join）

- 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***

- $r$: outer relation (外层关系), $s$: inner relation (内层关系)
- Require no indices and can be used for any kind of join condition
- **Expensive** since it checks every pair of tuples in the two relations
  - In the worst case, if the memory can only hold one block of each relation, the estimated cost is $b_r + n_r \cdot b_s$ block transfers, plus $n_r + b_r$ seeks
    - $n_r$: number of tuples in $r$
    - $b_s$: number of blocks in $s$
    - $b_r$: number of blocks in $r$
  - If the smaller relation fits entirely in memory, use it as the inner relation
    - Reduce cost to $b_s + b_r$ block transfers and 2 seeks

- Given the worst case memory availability, the cost estimate is
  - 5,000 * 400 + 100 = 2,000,100 disk accesses with **student** as outer relation, and 5,000 + 100 = 5,100 seeks
  - 10,000 * 100 + 400 = 1,000,400 disk accesses with **takes** as the outer relation, and 10,000+400 = 10,400 seeks
  - 较小的关系在外层更优
  - If smaller relation (student) fits entirely in memory, the cost estimate will be 500 disk accesses，这时较小的关系在内层更优

**#records**
- student: 5000
- takes: 10000

**#blocks**
- student: 100
- takes: 400

# 块嵌套循环连接（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**

# 块嵌套循环连接（续）

- Worst case cost estimate: $b_r \cdot b_s + b_r$ 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)
  - 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
    - E.g., 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**

- student: 5000
- takes: 10000

**#blocks**

- student: 100
- takes: 400

# 块嵌套循环连接（续）

- **Improvements to nested loop and block nested loop algorithms**
  - If equi-join (等值连接) attribute forms a key of inner relation, stop inner loop on first match
  - In block nested-loop, use $(M - 2)$ disk blocks as blocking unit for outer relations, where $M$ = memory buffer 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
  - 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)

# 索引嵌套循环连接（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 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 block of $r$. 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 tuples in $s$ 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 (why?)

# 索引嵌套循环连接示例



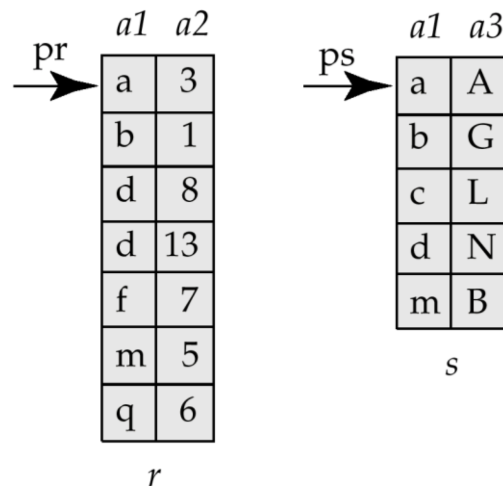
- **Compute student ⋈ takes**
  - Let takes have a primary B⁺-tree index on the join attribute student ID, which contains 20 entries in each index node
  - takes has 10,000 tuples (400 blocks), the height of the tree is 4, and one more access to find the actual data
  - student has 5000 tuples -> 100 blocks
- **Cost of block nested loops join**
  - 100*400 +100 = 40,100 block transfers + 2 * 100 =200 seeks
    - assuming the 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
  - Less block transfer but more seeks

**#records**
- student: 5000
- takes: 10000

**#blocks**
- student: 100
- takes: 400

# 归并连接（Merge-Join*）

- 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

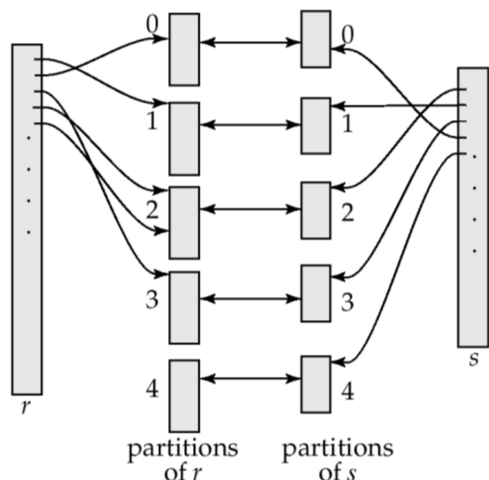


# 归并连接（续）

- Each block needs to be read only once (assuming all tuples for any given value of the join attributes fit in memory)
- Thus the 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

# 散列连接（Hash-Join*）

- 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（续）



- 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)**

# 散列连接（续）

- 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 the 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 $r$

# 溢出的处理



- 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（构造阶段）
  - 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

# 散列连接的Cost

- 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 $s$. It is best to choose the smaller relation as the build relation, and the 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 $b_r + b_s$

# 散列连接示例



***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 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: 3(100 + 400) = 1500 block transfers
  - ignores cost of writing partially filled blocks

# 混合散列连接（Hybrid Hash–Join）

TONGJI UNIVERSITY

- Useful when memory sizes 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 five 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.

# 混合散列连接（续）

- customer is similarly partitioned into five 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 is most useful if $M \gg \sqrt{b_s}$

# 复杂连接

- **Join with a conjunctive condition:** $\boldsymbol{r \bowtie_{\theta_1 \wedge \theta_2 \wedge \cdots \wedge \theta_n} s}$
  - use nested loops/block nested loops
  - compute the result of one of the simple joins $r \bowtie_{\theta i} s$, and check the result to 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:** $\boldsymbol{r \bowtie_{\theta_1 \vee \theta_2 \vee \cdots \vee \theta_n} s}$
  - use nested loops/block nested loops
  - compute as the union of the records in individual joins $r \bowtie_{\theta_i} s$:

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

# 复杂连接（续）



- **Join involving three relations: loan ⋈ depositor ⋈ customer**
  - **Strategy 1**: Compute depositor ⋈ customer, and use result to compute loan (depositor ⋈ customer)
  - **Strategy 2**: Compute 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.

# 目录



- 概述
- 查询代价度量
- 选择操作
- 排序操作
- 连接操作
- 其他操作
- 表达式执行

# 去重复与投影



- **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

# 聚合



- **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

# 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 $s_1, s_2, \ldots, s_n$
  - 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
    - $r \cup s$: 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 $r ⟕ s$
  - 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

# 目录



- 概述
- 查询代价度量
- 选择操作
- 排序操作
- 连接操作
- 其他操作
- 表达式执行

# 表达式的执行

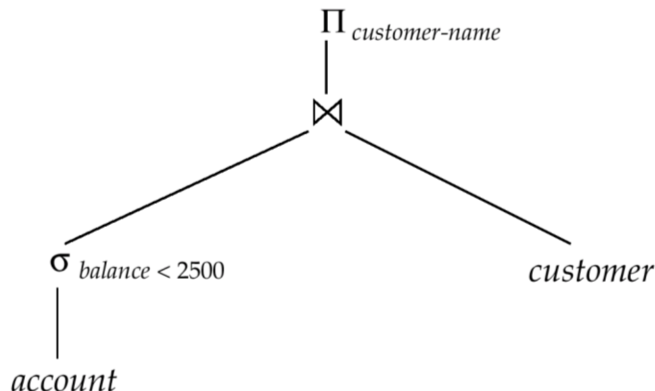
- Alternative ways for evaluating an entire expression tree
  - **Materialization (物化)**: generate the results of an expression and materialize (store) the results on disk
  - **Pipelining (流水线)**: pass on tuples to parent operations even as an operation is being executed

# 物化（Materialization）

- Materialized evaluation (物化计算) : evaluate one operation at a time, starting from 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.



# 物化（续）



- **Materialized evaluation** (物化计算) is always applicable
- The cost of writing the 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

# 流水线（Pipelining）



- **Pipelined evaluation** (流水线计算)
  - Evaluate several operations simultaneously, and pass the results of one operation to the next
  - E.g., in previous expression tree, don’t store the result of $\sigma_{balance<2500}(account)$ and pass tuples directly to the join. Similarly, don’t store the results of join, pass tuples directly to projection
  - Much cheaper than materialization but may not be possible – e.g., sort, hash-join
- Pipelines can be executed in two ways:
  - **demand driven (需求驱动)**
  - **producer driven (生产者驱动)**

# 流水线（续）

同 濟 大 學 TONGJI UNIVERSITY

- **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
- **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, and parent reads tuples from buffer
    - If buffer is full, child waits till there is space in the buffer, and then generates more tuples