# Database Systems
# Lecture15 – Chapter 15: Query Processing

SUNG KYUN KWAN UNIVERSITY

Beomseok Nam (남범석)
bnam@skku.edu

# Sorting

- Sorting plays an important role in DBMS.

- For relations that fit in memory, techniques like quicksort can be used. For relations that don't fit in memory, **external sort-merge** is a good choice.

# 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:

# Example: External Sorting Using Sort-Merge

M=4

**initial relation**

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

create runs

**runs**

| | |
|---|---|
| a | 19 |
| c | 33 |
| d | 31 |
| g | 24 |

| | |
|---|---|
| b | 14 |
| d | 21 |
| e | 16 |
| r | 16 |

| | |
|---|---|
| a | 14 |
| d | 7 |
| m | 3 |
| p | 2 |

merge pass?1

**sorted output**

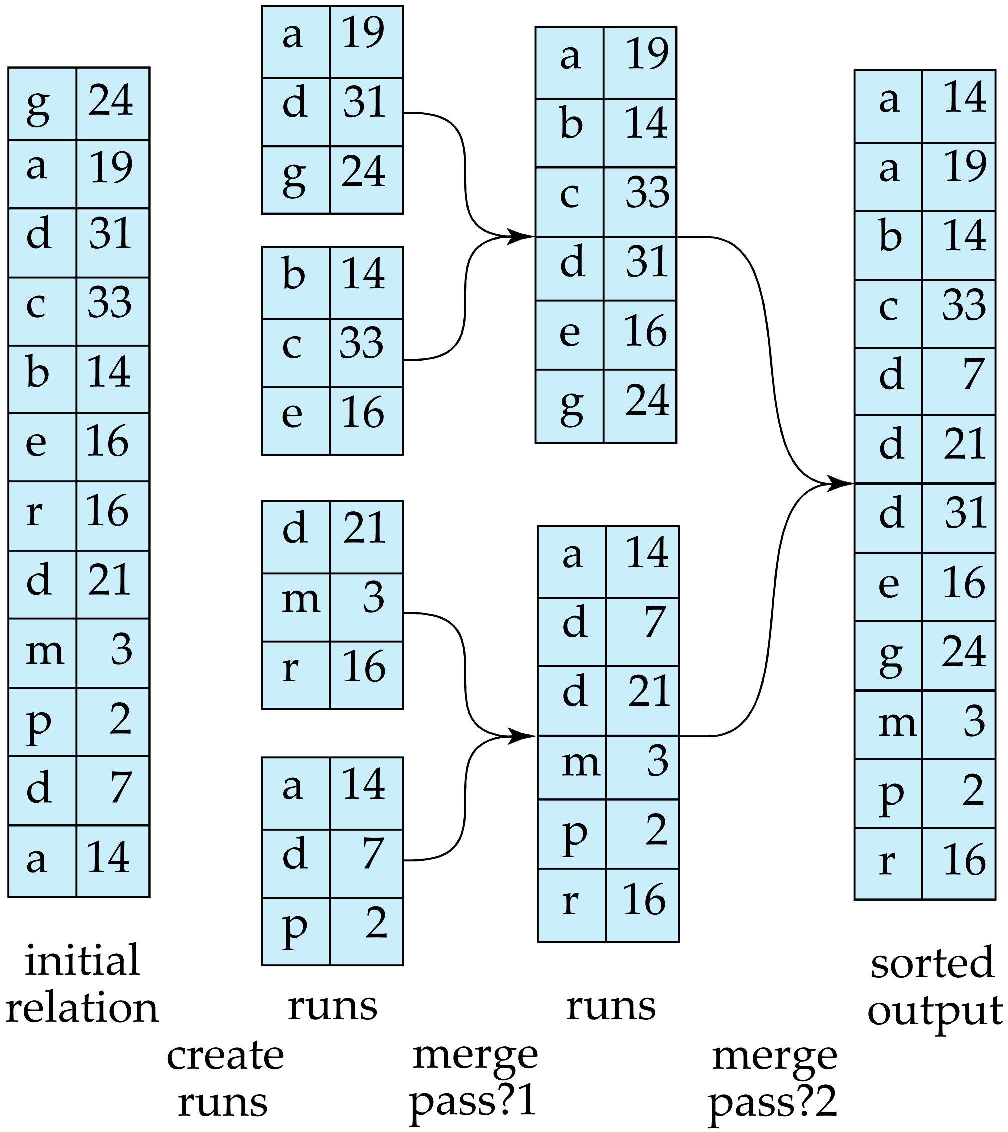
| | |
|---|---|
| a | 14 |
| a | 19 |
| b | 14 |
| c | 33 |
| d | 7 |
| d | 21 |
| d | 31 |
| e | 16 |
| g | 24 |
| m | 3 |
| p | 2 |
| r | 16 |

# 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 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

M=3



# External Merge Sort – Cost analysis

- Let $b_r$ denote the number of blocks containing records of relation r.
- The initial run creation needs $2b_r$ block transfer
- The initial number of runs is $\lceil b_r/M \rceil$.
- The number of runs decreases by a factor of M-1.
  - Total number of merge passes required: $\lceil \log_{M-1}(b_r/M) \rceil$.
- The number of block transfers for each pass is $2b_r$.
  - for final pass, we don't count write cost since the output of an operation is sent to the parent operation without being written to disk
- Thus total number of block transfers for external sorting:

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

- 1 block per run leads to too many seeks during merge
  - Instead use $b_b$ buffer blocks per run

    ➔ read/write $b_b$ blocks at a time

  ➔Replace M in the above equation with $\lfloor M/b_b \rfloor$

# External Merge Sort – Cost analysis

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

# Join Operation

- Several different algorithms to implement joins
  - Nested-loop join
  - Block nested-loop join
  - Indexed nested-loop join
  - Merge-join
  - Hash-join
- Choice based on cost estimate
- Examples use the following information
  - Number of records of *student*: 5,000
  - Number of records of *takes*: 10,000
  - Number of blocks of *student*: 100
  - Number of blocks of *takes*: 400

# Nested-Loop Join

- 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

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

- If the smaller relation fits entirely in memory, use that as the inner relation.
  - Reduces cost to $b_r + b_s$ block transfers and 2 seeks
- Assuming worst case,
  - with *student* as outer relation:
    - 5000 $*$ 400 + 100 = 2,000,100 block transfers,
    - 5000 + 100 = 5100 seeks
  - with *takes* as the outer relation
    - 10000 $*$ 100 + 400 = 1,000,400 block transfers and 10,400 seeks
- If smaller relation (*student*) fits entirely in memory, the cost estimate will be 500 block transfers.
- Block nested-loops algorithm (next slide) is preferable.

# 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 $t_r \bullet t_s$ 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
- Best case: $b_r + b_s$ block transfers + 2 seeks.

- 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
    - reduces the number of scans of inner relation from $b_r$ to $\lceil b_r / (M-2) \rceil$
    - Cost = $\lceil b_r / (M-2) \rceil * b_s + b_r$ block transfers + $2 \lceil b_r / (M-2) \rceil$ seeks