MDA104 Introduction to Databases

# 5. Query Processing

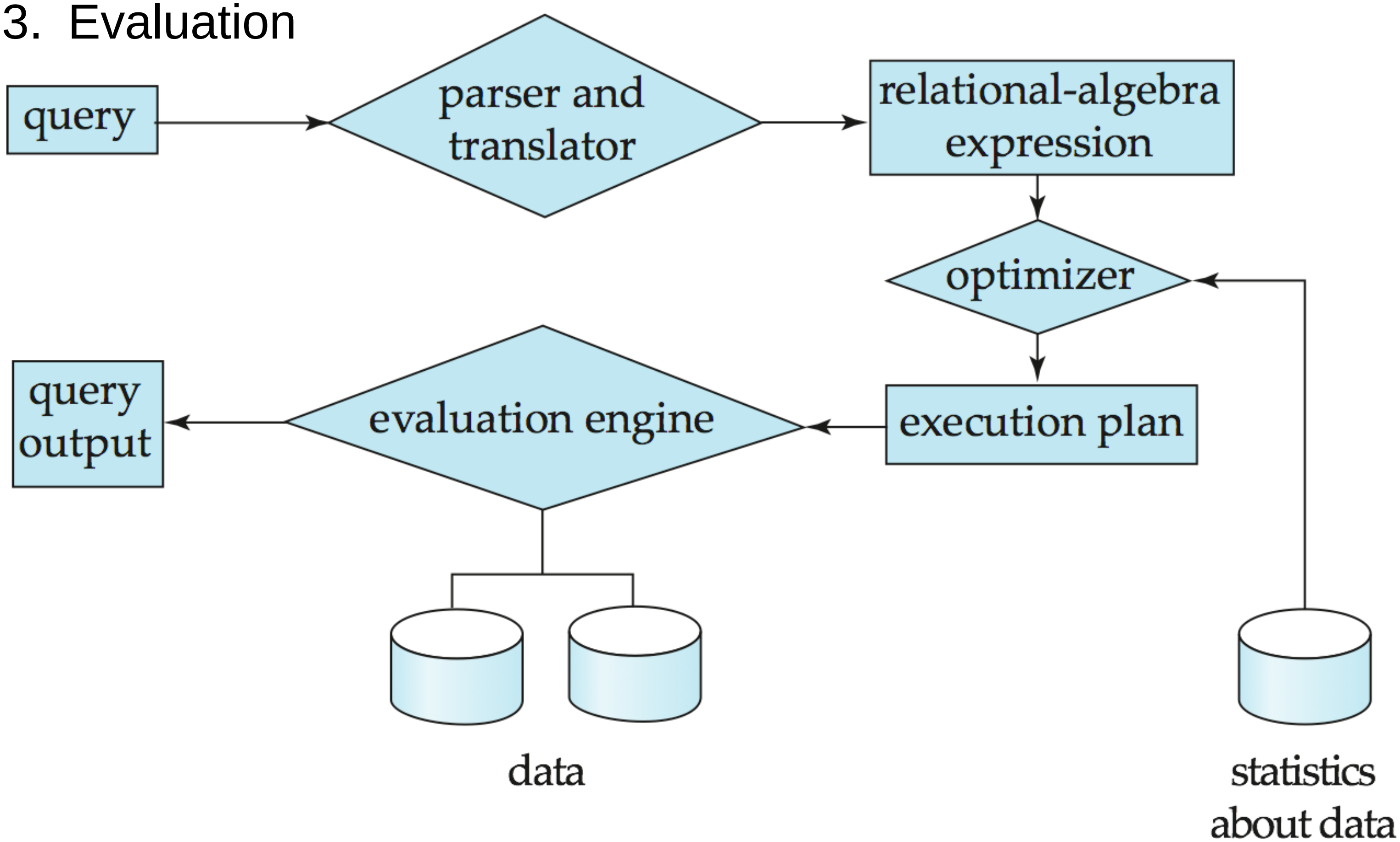
Vlastislav Dohnal

# Query Processing

- Overview
  - Evaluation of Expressions
  - Measures of Query Cost
- Evaluation algorithms
  - Sorting
  - Join Operation

# Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation



# Basic Steps in Query Processing (Cont.)

- Parsing and translation
  - Translate the SQL query into its internal form.
    - This is then translated into relational algebra.
  - Parser checks syntax, verifies relations
- Optimization
  - Generate a query-evaluation plan and choose algorithms for evaluating individual operations
- Evaluation
  - The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.

# Basic Steps in Query Processing (Cont.)

- Example of query:
  - List salary of all instructors that earn less than $75,000.

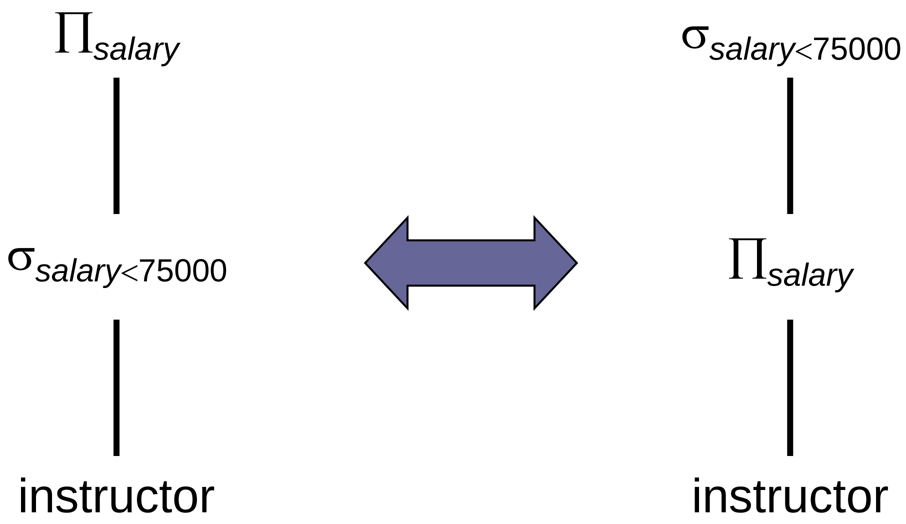
- SQL query
  - SELECT salary FROM instructor WHERE salary < 75000

- Conversion to rel. algebra
  - $\Pi_{salary}(\sigma_{salary<75000}(instructor))$

# Basic Steps: Optimization

- A relational-algebra expression may have many equivalent expressions:
  - $\Pi_{salary}(\sigma_{salary<75000}(instructor))$
  - $\sigma_{salary<75000}(\Pi_{salary}(instructor))$

- For a relational-algebra expression, an **expression tree** is created

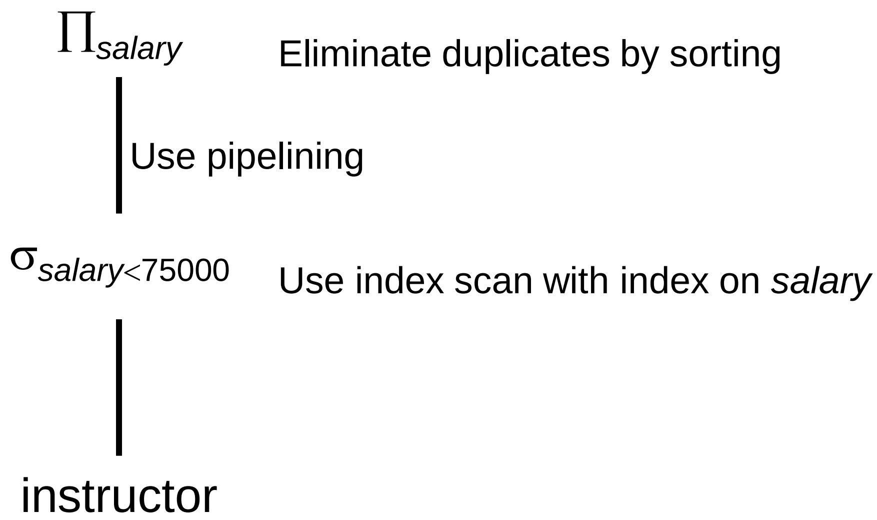


# Basic Steps: Optimization (Cont.)

- Each relational algebra operation can be evaluated using one of several different algorithms
  - Correspondingly, a relational-algebra expression can be evaluated in many ways.
- Annotated expression specifying detailed evaluation strategy is called an **execution-plan** or **evaluation-plan**.
  - E.g., to find instructors with salary < 75000
    - use an index on *salary*, or
    - perform complete relation scan and discard instructors with salary ≥ 75000

# Basic Steps: Optimization (Cont.)

- Example of an **evaluation-plan**



# Basic Steps: Optimization (Cont.)

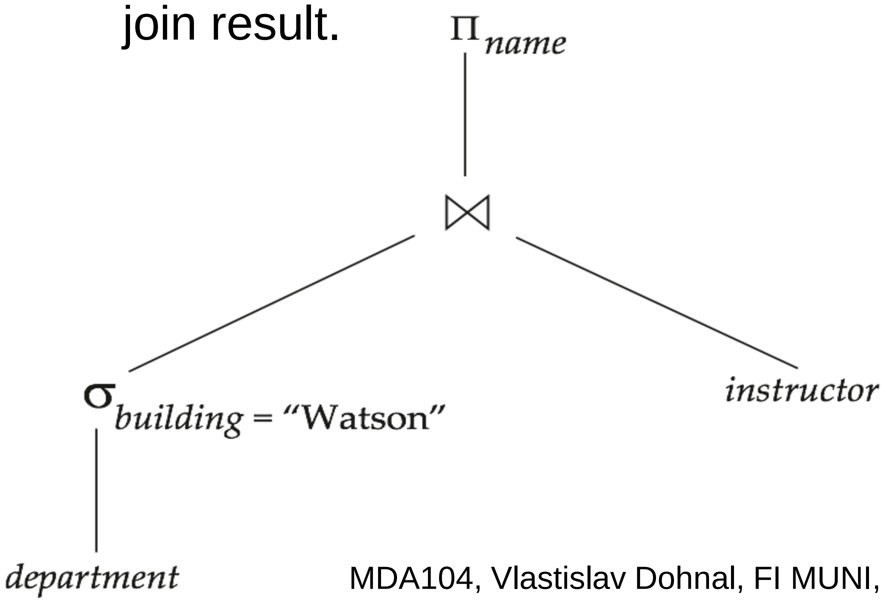
- **Query Optimization**
  - Amongst all equivalent evaluation plans choose the one with lowest cost.
  - Cost is estimated using statistical information from the database catalog
    - E.g. number of tuples in each relation, size of tuples, etc.
- There is a huge number of possible evaluation plans
  - Optimization uses some heuristics
    1. Perform selection early
       - reduce the number of tuples (by using an index, e.g.)
    2. Perform projection early
       - reduce the number of attributes
    3. Perform most restrictive operations early
       - such as join and selection.

# Evaluation of Expressions

- Alternatives for evaluating an entire expression tree
  - **Materialization**
    - Evaluate one operation at a time, starting at the lowest-level.
    - Use intermediate results materialized into temporary relations to evaluate next-level operations.
  - **Pipelining**
    - pass on tuples to parent operations even as an operation is being executed

# Evaluation of Expressions (Cont.)

- **Materialized evaluation**
  - Compute $\sigma_{building=\text{`Watson'}}$(*department*) and store it
  - Then read from stored intermediate result and compute its join with *instructor*, store it
  - Finally read it and compute the projection on *name* and output it.
    - This step can be conveniently evaluated using **pipelining** on join result.



# Measures of Query Cost

- Cost is generally measured as total elapsed time for answering query
  - Many factors contribute to time cost
    - *disk accesses, CPU*, or even network *communication*
- Typically disk access is the predominant cost and is also relatively easy to estimate. Measured by taking into account
  - Number of seeks * average-seek-cost
  - Number of blocks read * average-block-read-cost
  - Number of blocks written * 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
  - $t_S$ – time for one seek
  - Cost for $b$ block transfers plus $S$ seeks

    $$b * t_T + S * t_S$$
- We ignore CPU costs for simplicity
  - Real systems do take CPU cost into account
- We do not include cost to writing output to disk in our cost formulae

# Measures of Query Cost (Cont.)

- Several algorithms can reduce disk I/O by using extra buffer space
  - Amount of real memory available to buffer depends on other concurrent queries and OS processes, known only during execution
    - We often use worst case estimates, assuming only the minimum amount of memory needed for the operation is available
- Required data may be buffer resident already, avoiding disk I/O
  - But hard to take into account for cost estimation

# Selection Operation

- **File scan** (table / sequential scan) – no index structure is necessary
  - Scan each file block and test all records to see whether they satisfy the selection condition.
  - Cost estimate = $b_r$ block transfers + 1 seek
    - $b_r$ denotes number of blocks containing records from relation $r$
  - If selection is on a key attribute, can stop on finding matching record
    - 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
- 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

# Selections Using Indices

- **Index scan** – search algorithms that use an index
  - selection condition must be on search-key of index

- Now, assume the sequential file is ordered by this key:
- **Algorithm for primary index & equality on primary key**
  - Retrieve a single record that satisfies the corresponding equality condition
  - $Cost = (h_i + 1) * (t_S + t_T)$
    - $h_i$ – height of index $i$ (for hashing $h_i$ =1)
    - +1 – for reading the actual record
- **Algorithm for primary index & equality on non-primary key**
  - Retrieve multiple records.
  - Records will be on consecutive blocks
    - Let $b$ = number of blocks containing all $n$ matching records
  - $Cost = h_i * (t_S + t_T) + t_S + t_T * b$

# Selections Using Indices

- **Algorithm for secondary index & equality on non-primary key**
  - Sequential file is not ordered by this search key!
  - Retrieve a single record if the search-key is a candidate key
    - $Cost = (h_i + 1) * (t_S + t_T)$
  - Retrieve multiple records if search-key is not a candidate key
    - Each of *n* matching records may be on a different block.
    - $Cost = (h_i + n) * (t_S + t_T)$
      - Can be very expensive!

# Sorting Relations

- We may build an index on the relation, and then use the index to read the relation in the sorted order.
  - May lead to one disk block access for each tuple.
- Use a sorting algorithm
  - For relations that fit in memory, techniques like quick-sort 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/blocks):

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.)

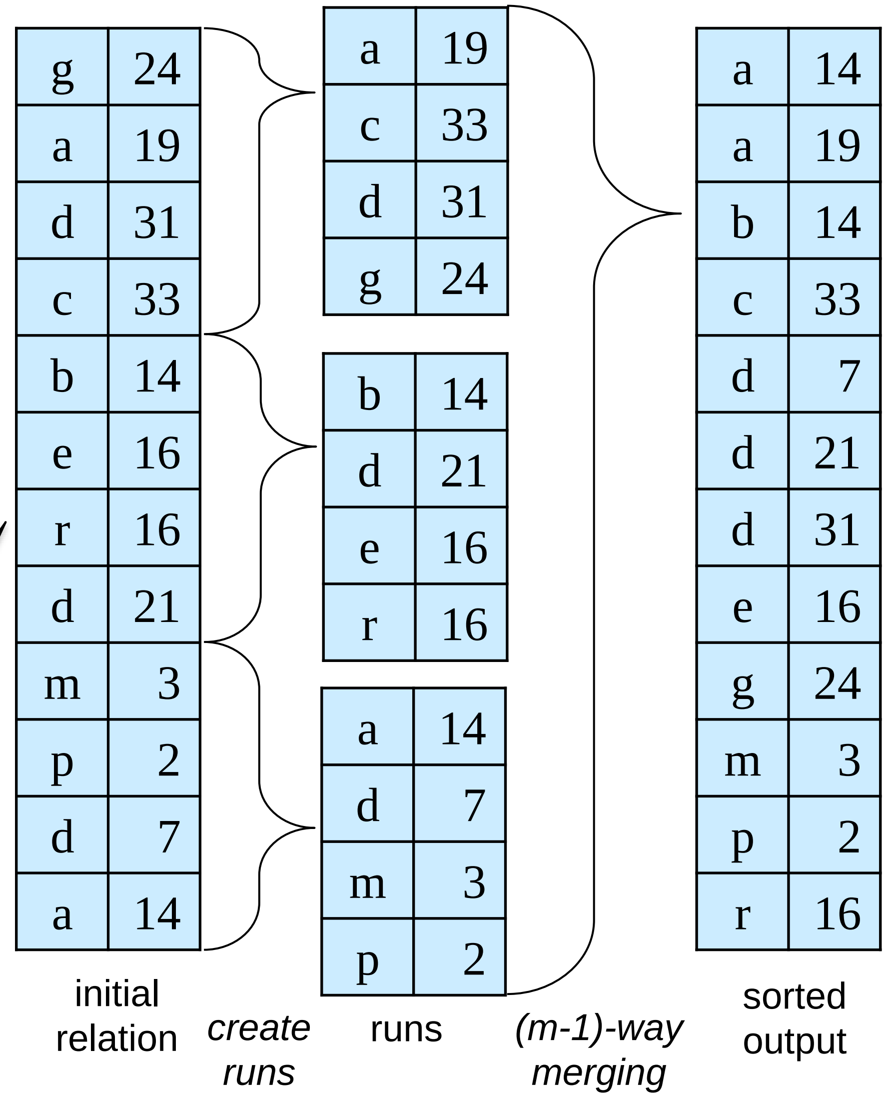
2. **Merge the runs (N-way merge)**.
   We assume (for now) that $N < M$.
   1. Use $N$ blocks of memory to buffer input runs, and 1 block to buffer output.
   2. Read the first block of each run into its buffer page
   3. **repeat**
      1. Select the first record (in sort order) among all buffer pages
      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.
   4. **until** all input buffer pages are empty.

# External Sort-Merge (Cont.)

- If $N \geq M$, several merge *passes* are required.
  - In each pass, continuous 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

Available memory **M=4** blocks.




One record per block

# External Sort-Merge (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

# Join Operation

- Several different algorithms to implement joins
  - Nested-loop join
  - Block nested-loop join
    - Improved nested-loop join by reading records in blocks
  - Indexed nested-loop join
    - Improved by using an index to look up equal records
  - Merge-join
  - Hash-join
- Choice based on cost estimate
  - For each of the variants a cost estimation can be stated.

# Nested-Loop Join

- To compute the join $r \bowtie s$
- **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 equality on shared attributes
  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.
  - $Cost = n_r * (t_S + t_T) * (n_s * (t_S + t_T))$
    - where $n_r$ = number of tuples in *r*

# 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

  - Example on *student* and *takes*
    - *student* (the smaller one) as the outer relation:
      - 5000 ∗ 400 + 100 = 2,000,100 block transfers,
      - 5000 + 100 = 5,100 seeks
    - *takes* (the larger one) as the outer relation
      - 10000 ∗ 100 + 400 = 1,000,400 block transfers and 10,400 seeks

$n_{student}$=5,000
$b_{student}$=100
$n_{takes}$=10,000
$b_{takes}$= 400

- 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
  - Example: *student* fits entirely in memory
    - the cost estimate is 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.

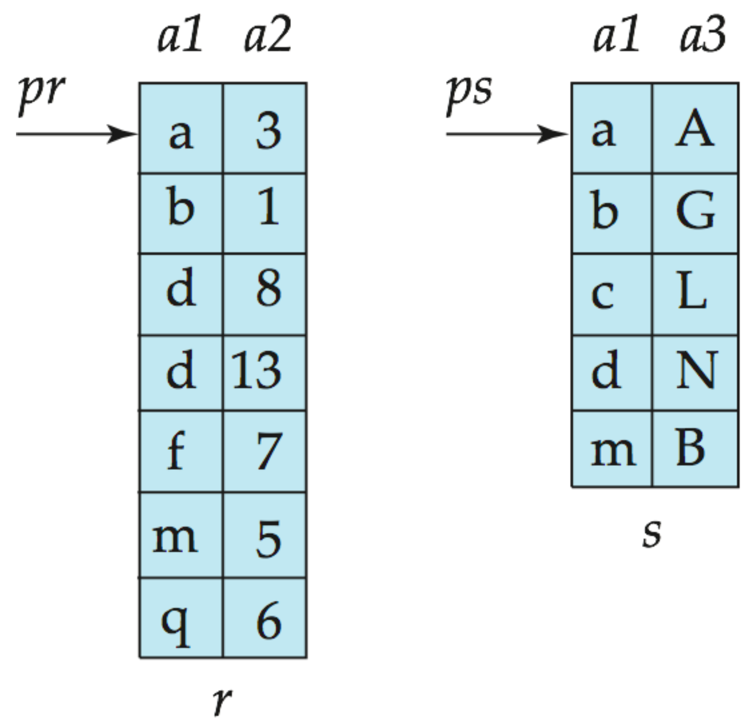
**for each** block $B_r$ **of** *r* **do begin**
  **for each** block $B_s$ **of *s* do begin**
    **for each** tuple $t_r$ **in** $B_r$ **do begin**
      **for each** tuple $t_s$ **in** $B_s$ **do begin**
        Check if $(t_r, t_s)$ satisfy the join condition
        if they do, add $t_r \bullet t_s$ to the result.
      **end**
    **end**
  **end**
**end**

- Cost: $b_r * (1+b_s)$ blocks; $b_r * (1+1)$ seeks
  - For *student* (outer) and *takes* (inner):
    - 100 + 100 * 400 = 40,100 block transfers
    - 100 + 100 seeks

$n_{student}$=5,000
$b_{student}$=100
$n_{takes}$=10,000
$b_{takes}$= 400

# Merge-Join

1. Sort both relations on their join attributes
   - If not already sorted.
2. 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



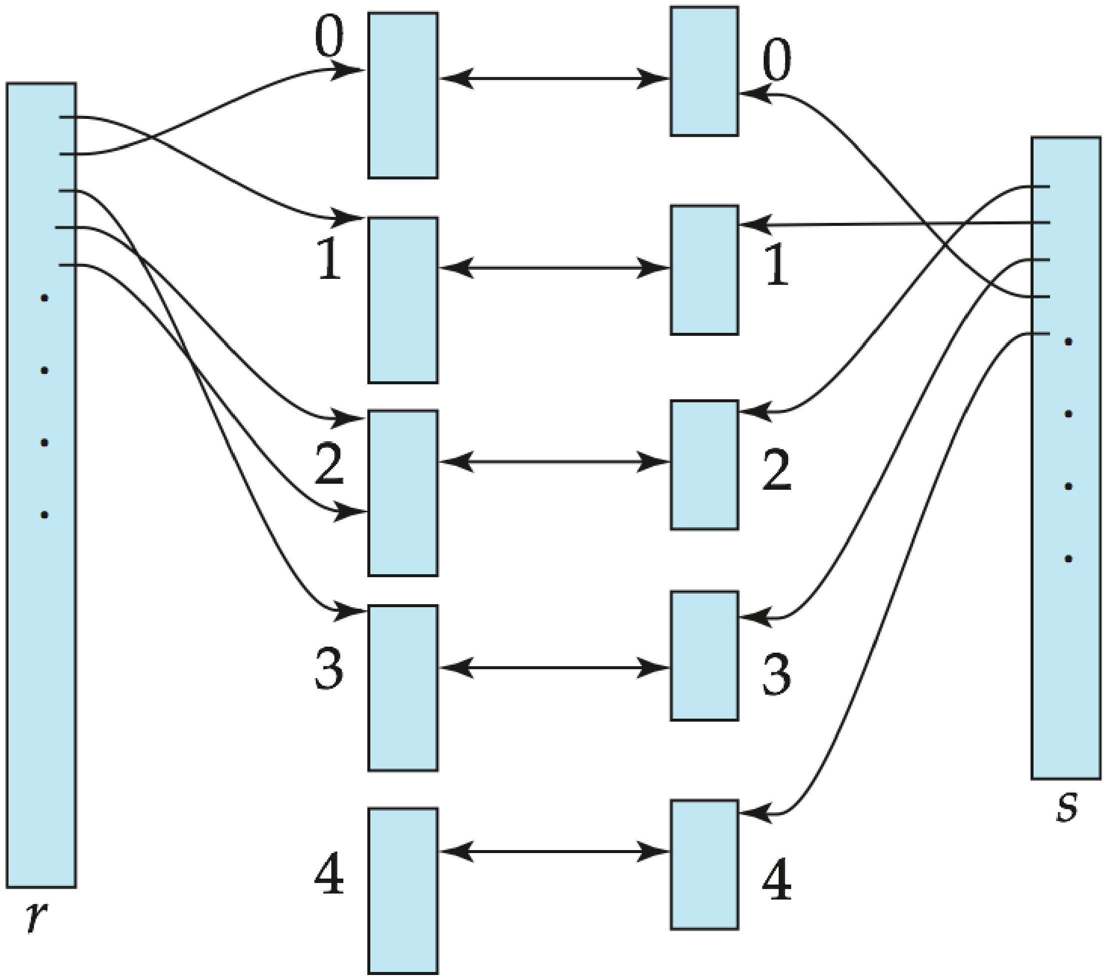
# Merge-Join (Cont.)

- 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 the cost of merge join is:
  - $b_r + b_s$ block transfers, and
  - max. $2 * \lceil b_r / b_b \rceil + 1$ seeks
    - Assuming we read $r$ in runs of $b_b$ blocks
  - + the cost of sorting if relations are unsorted.

# Hash-Join

- A hash function *h* is used to partition tuples of both relations
  - *JoinAttrs* are the common attributes of *r* and *s* used in $r \bowtie s$
- *h* maps *JoinAttrs* values to {0, 1, ..., *n*}
  - $r_0, r_1, \ldots, r_n$ denote buckets of *r*
    - Each tuple $t_r \in r$ is put in bucket $r_i$
      - where $i = h(t_r[JoinAttrs])$.
  - $s_0, s_1, \ldots, s_n$ denotes buckets of *s*
    - Each tuple $t_s \in s$ is put in bucket $s_i$,
      - where $i = h(t_s[JoinAttrs])$.

# Hash-Join (Cont.)




buckets $r_i$ of $r$ buckets $s_i$ of $s$

# Hash-Join (Cont.)

- Tuples in $r_i$ need only to be compared with tuples in $s_i$
  - Need not be compared with *s* tuples in any other bucket, since:
  - a tuple of *r* and a tuple of *s* that satisfy the join condition will have the same value for the join attributes.
  - If that value is hashed to some value *i*, the tuple of *r* has to be in $r_i$ and the tuple of *s* in $s_i$.
- Cost of hash join is $3(b_r + b_s)$ block transfers
  - 3*(100+400) for *student* ⋈ *takes*

$n_{student}$=5,000
$b_{student}$=100
$n_{takes}$=10,000
$b_{takes}$= 400

# Summary – Takeaways

- Steps in query processing
- Idea of query optimization
  - expression transformations (in parse tree)
  - selection of algorithm to evaluate an operator
    - index vs table scan
- Algorithms
  - Sorting large relations (exceeding RAM allocation)
  - Joining relations
    - nested loops, merge join, hash join