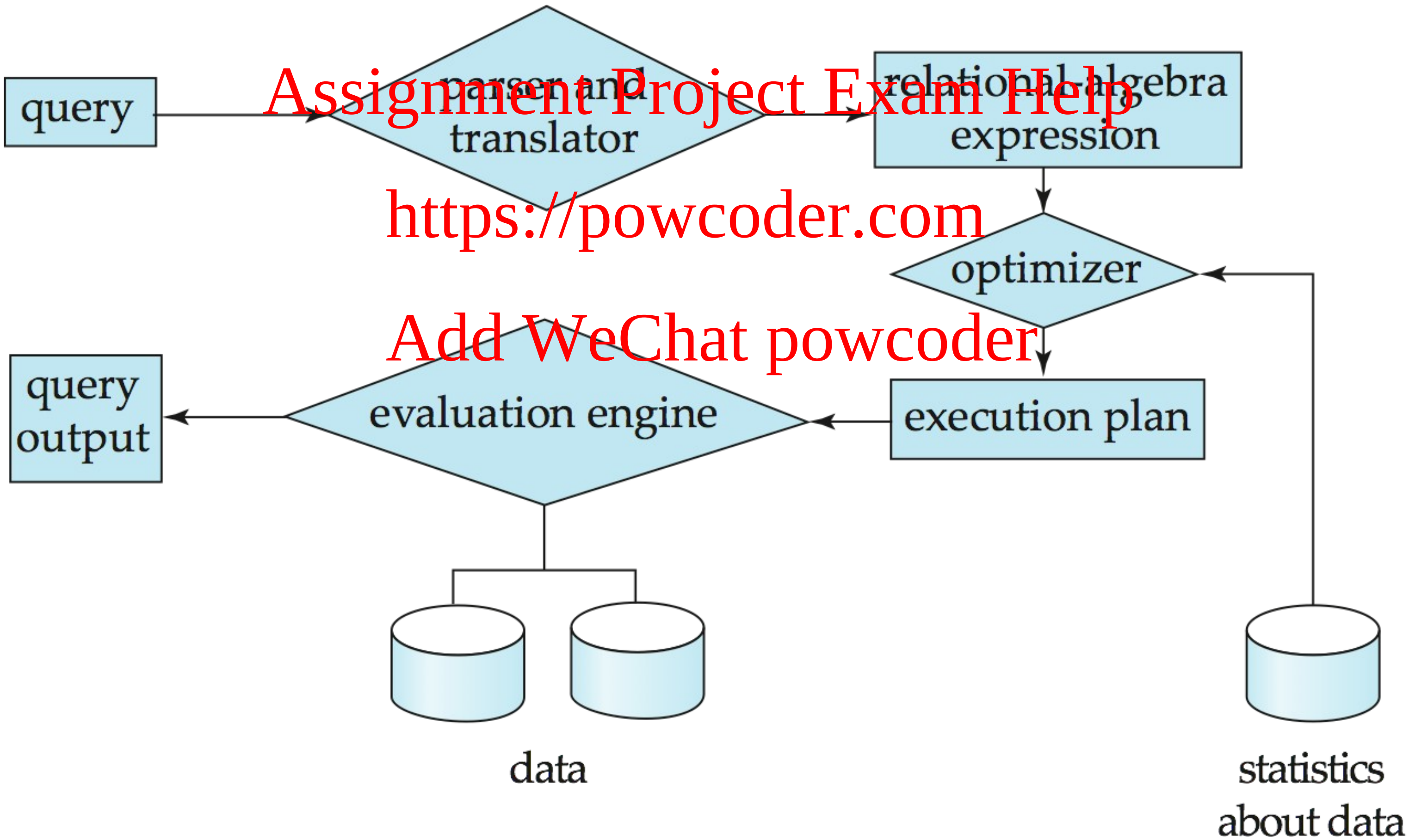
Query Processing and Optimization

- **Overview**
- Measures of Query Cost
- Selection Operation
- Sort Operation
- Join Operation
  - nested-loop, block nested-loop, indexed nested-loop, merge-join and hash join
- Other Operations
- Evaluation of Expressions
- Transformation of Relational Expressions
- Statistics Estimation
- Choices of Evaluation Plans

Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation



Basic Steps in Query Processing (Cont.)

- **Parsing and translation**
  - Parser checks syntax, verifies relations
  - Translator translates the query (e.g., SQL) into its internal form (e.g., relational algebra)

- A SQL query can be translated into ***several relational algebra expressions***.

  - E.g., **select** *salary* **from** *instructor* **where** *salary* < 75000; →
  - $\sigma_{salary<75000}(\Pi_{salary}(instructor))$, or
  - $\Pi_{salary}(\sigma_{salary<75000}(instructor))$

Basic Steps in Query Processing (Cont.)

- Each relational algebra operation can be evaluated using one of several different algorithms
  - E.g., can search every tuple in *instructor* to find tuples with salary < 75000, or
  - can use an index on *salary* to find instructors with salary < 75000
- Correspondingly, a relational algebra expression can be evaluated in many ways.
- Annotated expression specifying detailed evaluation strategy is called an **evaluation plan** (or execution plan).
- **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.)

- **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.
- In this lecture, we study
  - how to measure query costs
  - algorithms for evaluating relational algebra operations
  - how to combine algorithms for individual operations in order to evaluate a complete expression
  - how to optimize queries, that is, how to find an evaluation plan with lowest estimated cost

Query Processing and Optimization

- Overview
- **Measures of Query Cost**
- Selection Operation
- Sort Operation
- Join Operation
  - nested-loop, block nested-loop, indexed nested-loop, merge-join and hash join
- Other Operations
- Evaluation of Expressions
- Transformation of Relational Expressions
- Statistics Estimation
- Choices of Evaluation Plans

Measures of Query Cost

- Cost can be measured based on **response time**, i.e., total elapsed time for answering query
  - Many factors contribute to time cost
    - *disk access, CPU* time, …
- Typically disk access is the predominant cost, and is also relatively easy to estimate.
  - Real systems do take CPU cost into account
- Disk cost can be estimated as **number of block transfers** *from disk and* **number of seeks**
    - t_T – time to transfer one block
    - t_S – time for one seek
    - Cost for *b* block transfers plus *S* seeks

      $b * t_T + S * t_S$

Measures of Query Cost (Cont.)

- We do not include cost of writing output to disk in our cost formulae
  - The output of an operation may be sent to the next operation without being written to disk.
- Worst case estimates
  - No data is initially in buffer, i.e., data must be read from disk initially
  - Only the minimum amount of memory needed for the operation is available
    - 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

Query Processing and Optimization

- Overview
- Measures of Query Cost
- **Selection Operation**
- Sort Operation
- Join Operation
  - nested-loop, block nested-loop, indexed nested-loop, merge-join and hash join
- Other Operations
- Evaluation of Expressions
- Transformation of Relational Expressions
- Statistics Estimation
- Choices of Evaluation Plans

Selection Operation

- **File scan**
- Algorithm **A1** (**linear search**). Scan each file block (stored contiguously) and test all records to see whether they satisfy the selection condition.

  - Cost estimate = $t_S + b_r * t_T$

    - b_r denotes number of blocks in relation r

  - If selection is on a key attribute, can stop on finding record
    - average cost = $t_S + (b_r/2)* t_T$
  - Linear search can be applied regardless of
    - selection condition, or
    - ordering of records in the file, or
    - availability of indices

Selections Using Indices

- **Index scan** – search algorithms that use an index (B^+-tree)
  - selection condition must be on ***search-key of index***.
- **A2** (**primary index, equality on key**). Retrieve a single record that satisfies the corresponding equality condition
  - $Cost = (h_i + 1) * (t_T + t_S)$
    - h_i = height of the ***B^+-tree index***
  - traverse the height of the tree + one I/O to fetch the record; each requires a seek and a block transfer
- **A3** (**primary index, equality on nonkey**). Retrieve multiple records on consecutive blocks.
  - $Cost = h_i * (t_T + t_S) + t_S + b * t_T$
    - b = number of blocks containing matching records

Selections Using Indices (Cont.)

- **A4** (**secondary index, equality**).
  - 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!

Selections Involving Comparisons

- Can implement selections of the form $\sigma_{A \leq V}(r)$ or $\sigma_{A \; V}(r)$ by using
  - a linear file scan,
  - or by using indices in the following ways:
- **A5** (**primary index, comparison**). (Relation is sorted on *A*)
  - For $\sigma_{A \; V}(r)$, ***use index*** to find first tuple *V* and ***scan relation*** sequentially from there (cost estimate is identical to A3)
  - For $\sigma_{A \leq V}(r)$, use ***file scan*** till first tuple > *V*; ***do not use index***

- **A6** (**secondary index, comparison**).
  - For $\sigma_{A \; V}(r)$, ***use index*** to find first index entry *V* and ***scan leaf index nodes*** sequentially from there, to find pointers to records (cost estimate is identical to A4, equality on nonkey)
  - For $\sigma_{A \leq V}(r)$, just ***scan leaf index nodes*** 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

Implementation of Complex Selections

- **Conjunction:** $\sigma_{\theta 1} \wedge_{\theta 2} \wedge \ldots_{\theta n}(r)$
- **A7** (**conjunctive selection using one index**).
  - Select a combination of θ_i and algorithms A1 through A6 that results in the least cost for $\sigma_{\theta i}(r)$.
  - Test other conditions on tuple after fetching it into memory buffer.
- **A8** (**conjunctive selection using composite 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.
  - If some conditions do not have appropriate indices, apply test in memory.

Algorithms for Complex Selections

- **Disjunction:** $\sigma_{\theta 1} \vee {}_{\theta 2} \vee \ldots {}_{\theta n} (r)$.
- **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.

Query Processing and Optimization

- Overview
- Measures of Query Cost
- Selection Operation
- **Sort Operation**
- Join Operation
  - nested-loop, block nested-loop, indexed nested-loop, merge-join and hash join
- Other Operations
- Evaluation of Expressions
- Transformation of Relational Expressions
- Statistics Estimation
- Choices of Evaluation Plans

Sorting

- Sorting is important because
  - SQL queries can specify that the output be sorted, and,
  - for query processing, several of the relational operations, such as joins, can be implemented efficiently if the input relations are first sorted.
- 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=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. 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.)

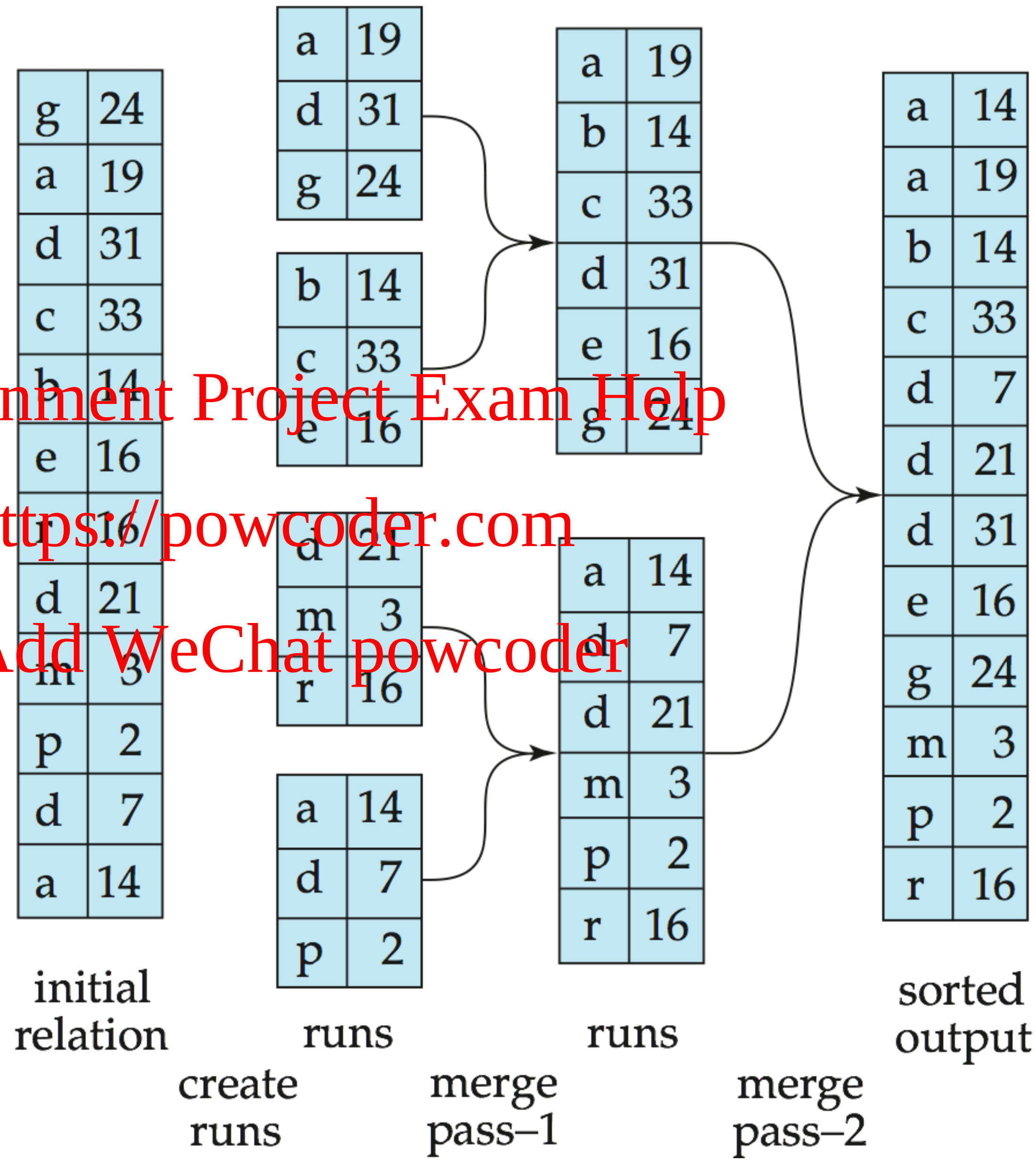
- If *N* *M*, several ***merge passes*** are required.
  - In each pass, contiguous groups of *M* - 1 runs are merged to get a single run for the next pass.
  - A pass reduces the number of runs by a factor of *M* -1, and creates runs longer by the same factor.
    - E.g. If *M*=11, and there are 90 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

- memory size: 3 blocks
- block size: one tuple

For merging

- two input blocks
- one output block



External Merge Sort (Cont.)

- Cost analysis:
  - Initial number of runs: $\lceil b_r / M \rceil$
  - Using b_b buffer blocks per run during merge, i.e., read/write b_b blocks at a time, can merge $\lfloor M / b_b \rfloor$ - 1 runs in one pass
  - Total number of merge passes required: $\lceil \log_{\lfloor M / b_b \rfloor - 1} (b_r / M) \rceil$.
  - Block transfers for initial run creation and in each pass is $2b_r$
    - for final pass, we don't count write cost
      - *Reminder*: ignore final write cost
    - Thus total number of block transfers for external sorting:

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

  - What is the number of block transfers in the example shown on the last slide?
  - *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

    - 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:

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

  - What is the cost of seeks in the example shown on the last slide?

Query Processing and Optimization

- Overview
- Measures of Query Cost
- Selection Operation
- Sort Operation
- **Join Operation**
  - **nested-loop, block nested-loop, indexed nested-loop, merge-join and hash join**
- Other Operations
- Evaluation of Expressions
- Transformation of Relational Expressions
- Statistics Estimation
- Choices of Evaluation Plans

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 (n) of *student*: 5,000 *takes*: 10,000
  - Number of blocks (b) of *student*: 100 *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 θ
          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, we have to perform ***a complete scan on s for each record in r*** and the estimated cost is

  $(n_r * b_s + b_r)$ block transfers + $(n_r + b_r)$ seeks

  where n: no. of tuples, b: no. of blocks
- 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
  - Assume worst case memory availability, cost estimate is

    - 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 and 2 seeks
- *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

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
  - More efficient to use the smaller relation as the outer relation
- Best case: $b_r + b_s$ block transfers + 2 seeks.
- Example
  - Assuming worst case memory availability, cost estimate is
    - with *student* as outer relation:
      - 100 $*$ 400 + 100 = 40,100 block transfers,
      - 2 $*$ 100 = 200 seeks
  - The best-case cost remains the same.

Indexed Nested-Loop Join

- Index lookups can replace file scans if an index is available on the inner relation's join attribute.
- 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* and one block of the index.
- 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.

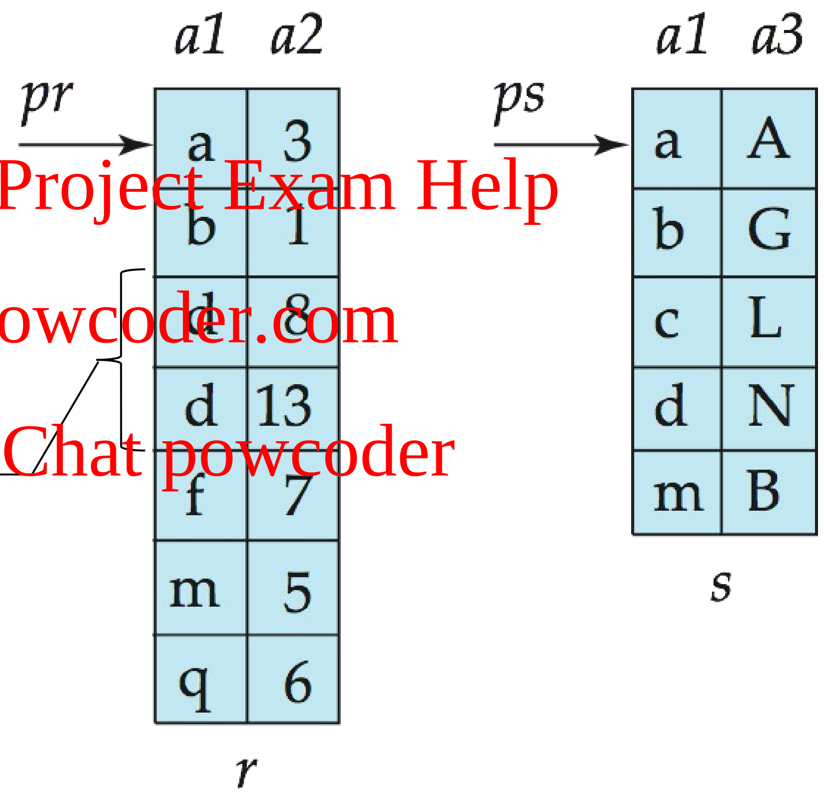
Example of Nested-Loop Join Costs

- Compute *student* $\bowtie$ *takes,* with *student* as the outer relation.
- Let *takes* have a primary B$^+$-tree index on the attribute *ID,* which contains 20 entries in each index node.
- Since *takes* has 10,000 tuples, the height of the tree is 4, and one more access is needed to find the actual data
- *student* has 5000 tuples
- Cost of block nested loops join
  - 100 * 400 + 100 = 40,100 block transfers,
  - 2 * 100 = 200 seeks
- Cost of indexed nested loops join
  - 100 + 5000 * 5 = 25,100 block transfers and seeks.

Merge-Join

1. Sort both relations on their join attribute (if not already sorted on the join attributes).
2. Merge the sorted relations to join them
   - Similar to the merge stage of the sort-merge algorithm.
   - A group of tuples of one relation with the same join-attribute values is read into a set
   - Skip tuples of another relation with the join-attribute values smaller than the current join-attribute value of those tuples in the set
   - Join every tuple in the set with the tuples of another relation with the same join-attribute values
   - *Detailed algorithm in book*

Merge-Join



A group of tuples of one relation with the same value on the join attributes is read into a set before matching.

Merge-Join (Cont.)

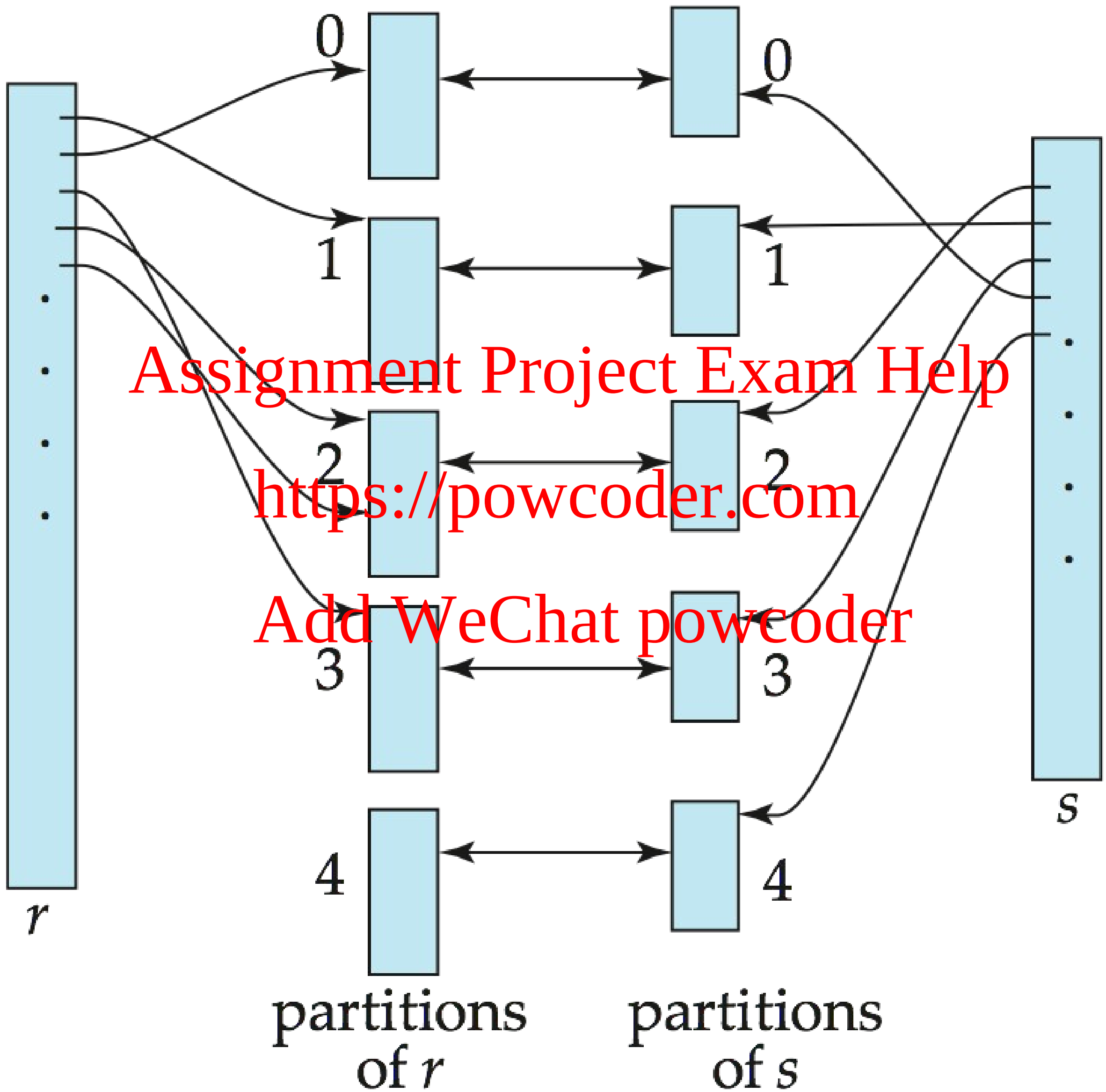
- Can be used only for ***equi-joins*** and ***natural joins***
- Each block needs to be read only once
- Thus the cost of merge join is (assuming b_b buffer blocks are allocated to each relation):
  - $b_r + b_s$ block transfers $+ \lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil$ seeks + the cost of sorting if relations are unsorted.
- Example
  - Assuming worst case memory availability and the relations are already sorted on the join attribute, cost estimate
    - 400 + 100 = 500 block transfers,
    - 400 + 100 = 500 seeks (assuming b_b =1)

Hash-Join

- Applicable for ***equi-joins*** and ***natural joins***.
- A hash function h is used to partition tuples of both relations into sets that have the same hash value on the join attributes.
- h maps *JoinAttrs* values to $\{0, 1, ..., n\text{-}1\}$, where *JoinAttrs* denotes the common attributes of r and s used in the natural join.

  - $r_0, r_1, \ldots, r_{n-1}$ denote partitions of r tuples
    - Each tuple $t_r \in r$ is put in partition r_i where $i = h(t_r[JoinAttrs])$.
  - $s_0, s_1, \ldots, s_{n-1}$ denote partitions of s tuples
    - Each tuple $t_s \in s$ is put in partition s_i where $i = h(t_s[JoinAttrs])$.

Hash-Join (Cont.)



Hash-Join (Cont.)

- 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.

- As a result, *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.

Hash-Join Algorithm

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

1. Partition the relation *s* using hash function *h* on the join attributes. 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:* // perform indexed nested-loop join on each partition

   (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 block by block. 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 for build and probe.
  - The probe relation partitions need not fit in memory
  - Typically, $M > \lceil b_s/n \rceil \rightarrow n > \lceil b_s/M \rceil$
  - Note that $n < M$ for partitioning

Cost of Hash-Join

- Cost of hash join is
  - for partitioning, a complete reading and a subsequent writing back of both relations:

    $$2(b_r + b_s) \text{ block transfers } +$$
    $$2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil) \text{ seeks}$$

  - for build and probe, a reading of each of the partitions once:

    $$(b_r + b_s) \text{ block transfers } + 2 * n \text{ seeks}$$

  - total:

    $$3(b_r + b_s) \text{ block transfers } +$$
    $$2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil) + 2 * n \text{ seeks}$$

- If the entire build input can be kept in main memory, no partitioning is required
  - Cost estimate goes down to $b_r + b_s$ block transfers + 2 seeks

Example of Cost of Hash-Join

takes ⋈ *student*

- Assume that memory size is 22 blocks
- $b_{student}$ = 100 and b_{takes} = 400.

- *student* is to be used as build input. Partition it into five partitions, each of size 20 blocks.

- Similarly, partition *takes* into five partitions, each of size 80.

- Therefore total cost, assuming 3 blocks are allocated to the input and each of the 5 outputs during partitioning (b_b=3):
  - 3(100 + 400) = 1500 block transfers +
    $2(\lceil 100/3 \rceil + \lceil 400/3 \rceil) + 2 \times 5 = 346$ seeks

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 $r \bowtie_{\theta_i} s$:

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

Query Processing and Optimization

- Overview
- Measures of Query Cost
- Selection Operation
- Sort Operation
- Join Operation
  - nested-loop, block nested-loop, indexed nested-loop, merge-join and hash join
- **Other Operations**
- Evaluation of Expressions
- Transformation of Relational Expressions
- Statistics Estimation
- Choices of Evaluation Plans

Other Operations: Duplicate elimination

- **Duplicate elimination** can be implemented via hashing or sorting.
  - On sorting, duplicates will come adjacent to each other, and all but one copy can be deleted
    - In external sort merge, duplicates can be deleted during run generation and at intermediate merge steps.
  - On hashing, duplicates will come into the same bucket
    - relation is partitioned on the basis of a hash function on the whole tuple
    - read in each partition, construct an in-memory hash index and insert a tuple only if it is not already present, otherwise, discard it.
  - Worst-case cost estimate is the same as that for sorting or hash join.
  - Because of the relatively high cost, SQL requires an explicit request to remove duplicates.

Other Operations : Projection and Aggregation

- **Projection**:
  - perform projection on each tuple
  - followed by duplicate elimination.

- **Aggregation** can be implemented in a manner similar to duplicate elimination, but based on the grouping attributes.

  - 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.

  - The cost estimate is the same as that of duplicate elimination.

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
  2. Process each partition i as follows.
     1. Using a different hashing function, build an in-memory hash index on r_i.
     2. Process s_i as follows (next slide)

Other Operations : Set Operations (Cont.)

2. Process s_i as follows
   - $r \cup s$:
     1. Add tuples in s_i to the hash index if they are not already in it.
     2. At end of s_i, add the tuples in the hash index to the result.
   - $r \cap s$:
     1. Output tuples in s_i to the result if they are already there in the hash index
   - $r - s$:
     1. For each tuple in s_i, if it is there in the hash index, delete it from the index.
     2. At end of s_i, add remaining tuples in the hash index to the result.

Query Processing and Optimization

- Overview
- Measures of Query Cost
- Selection Operation
- Sort Operation
- Join Operation
  - nested-loop, block nested-loop, indexed nested-loop, merge-join and hash join
- Other Operations
- **Evaluation of Expressions**
- Transformation of Relational Expressions
- Statistics Estimation
- Choices of Evaluation Plans

Evaluation of Expressions

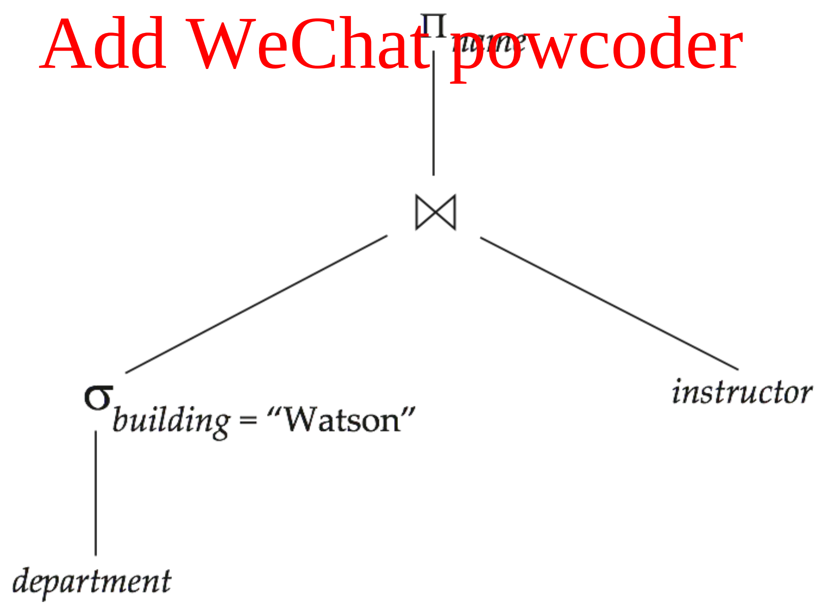
- So far: we have seen algorithms for individual operations
- Alternatives for evaluating an entire expression
  - **Materialization**
    - evaluate one operation at a time in an appropriate order
    - **materialize** (store) the result of each evaluation in a temporary relation for subsequent use
  - **Pipelining**
    - evaluate several operations simultaneously in a **pipeline**
    - pass on the results of one operation to the next, without the need to store a temporary relation

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., the following ***operator tree*** computes and stores

$$\sigma_{building="Watson"}(department)$$

then computes and stores its join with *instructor,* and finally computes the projection on *name.*



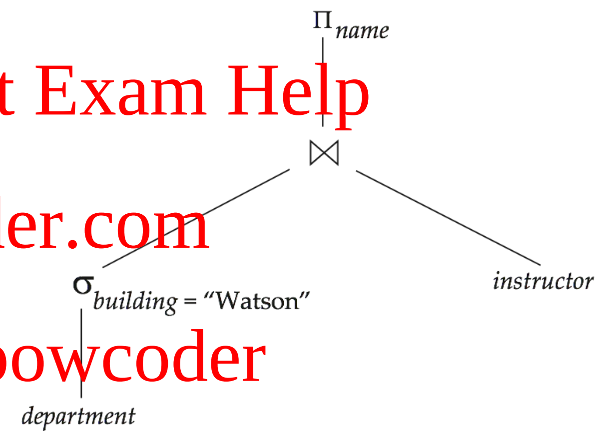
Materialization (Cont.)

- 👍 Materialized evaluation is always applicable
- 👎 Cost of writing results to disk and reading them back can be quite high
  - 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

Pipelining

- **Pipelined evaluation** : evaluate several operations simultaneously, passing the results of one operation on to the next.
- E.g., in previous operator tree, don't store result of

$$\sigma_{building="Watson"}(department)$$



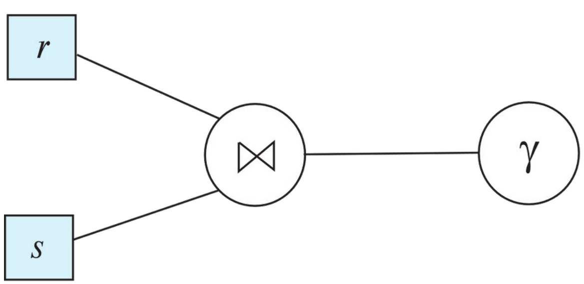
  - instead, pass tuples directly to the join. Similarly, don't store result of join, pass tuples directly to projection.
- ♘ Much cheaper than materialization: no need to store a temporary relation to disk.

Pipelining (Cont.)

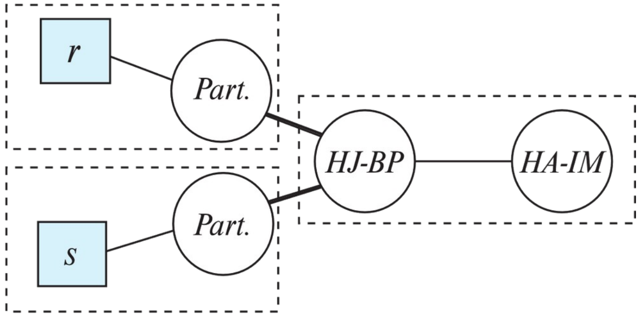
- However, pipelining may not always be possible.
  - Some **blocking operations**, e.g., sort, may not be able to output any results until all tuples from their inputs have been examined.
  - Other operations, such as join, are not inherently blocking but specific algorithms may be blocking.
    - Hash-join requires both its inputs to be fully partitioned before it outputs any tuples.
    - Indexed nested-loop join can output result tuples as it gets tuples for the outer relation (pipelined on its outer relation).
    - Merge join can be pipelined if both inputs are sorted on the join attribute and the join condition is an equi-join.
- For pipelining to be effective, use evaluation algorithms that generate output tuples even as tuples are received for inputs to the operation.

Blocking Operations

- **Blocking operations**: cannot generate any output until all inputs are consumed
  - E.g. sorting, aggregation, …
- But, some blocking operators can often consume inputs from a pipeline, or produce outputs to a pipeline
  - Such operations actually execute in stages and blocking actually happens between two stages of the operation.
  - For sort: run generation and merge
  - For hash join: partitioning and build-probe



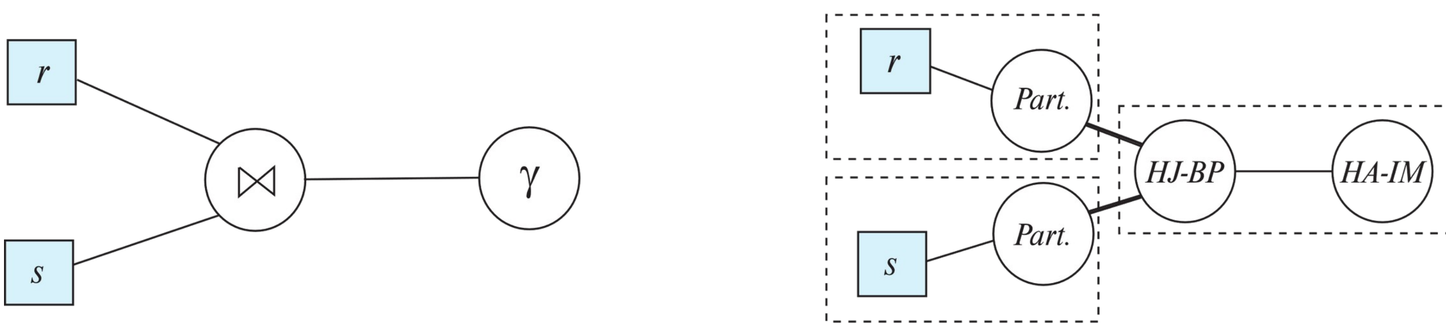
(a) Logical Query



(b) Pipelined Plan

Pipeline Stages

- **Pipeline stages**:
  - All operations in a stage run concurrently
  - A stage can start only after preceding stages have completed execution
- Example: Hash-join is a blocking operation since it requires both its inputs to be fully partitioned before it outputs any tuples.
  - The partitioning step for each input can be pipelined with its input
  - The build-probe step can be pipelined with its output
  - The build-probe step can start only after partitioning has been completed on both inputs



(a) Logical Query (b) Pipelined Plan

Query Optimization

- *Reminder:*
  - **Query Optimization**: Amongst all equivalent evaluation plans, choose the one with lowest cost.
  - Annotated expression specifying detailed evaluation strategy is called an **evaluation plan** (or **execution plan**).
- Alternative ways of evaluating a given query
  - Different algorithms for each operation
  - Equivalent expressions

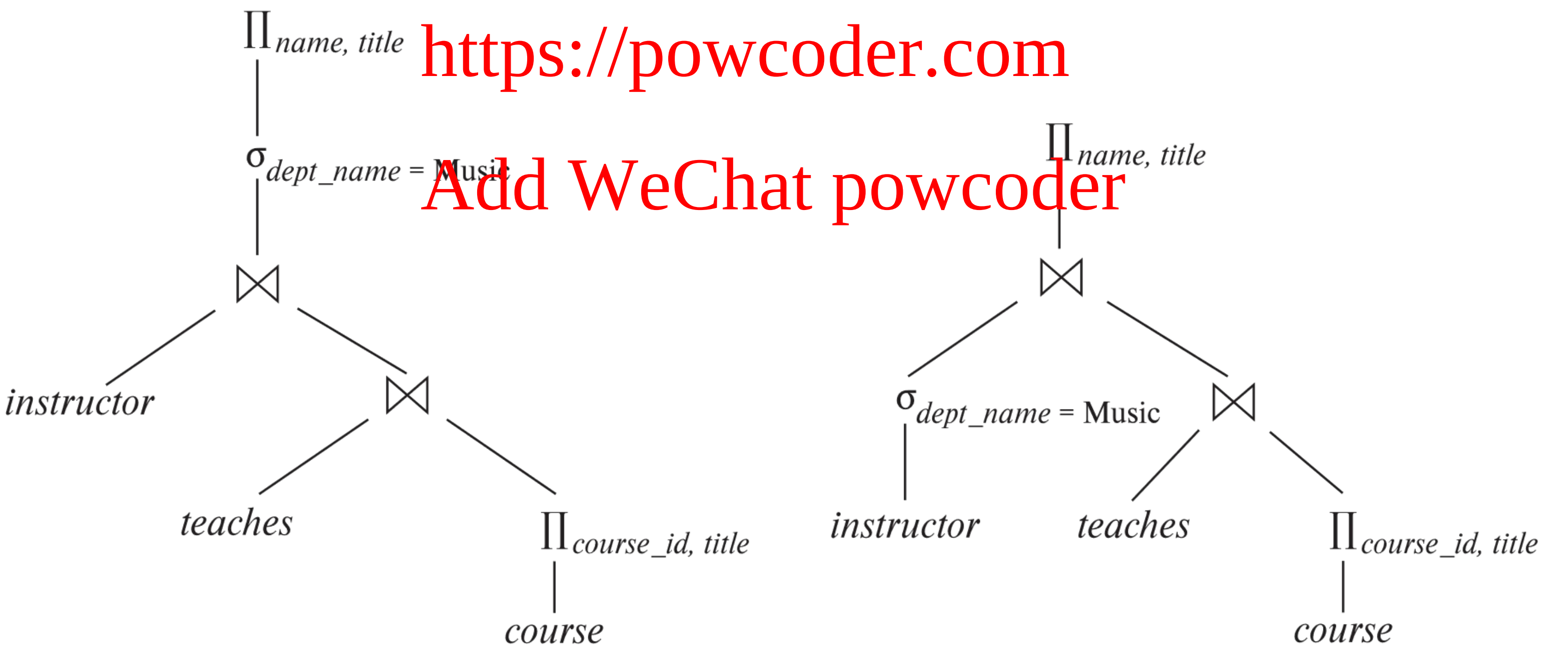
Query Optimization

- Example: Find the names of all instructors in the Music department together with the course title of all the courses that the instructors teach.

instructor(*ID*, *name*, *dept_name*, *salary*)

teaches(*ID*, *course_id*, *sec_id*, *semester*, *year*)

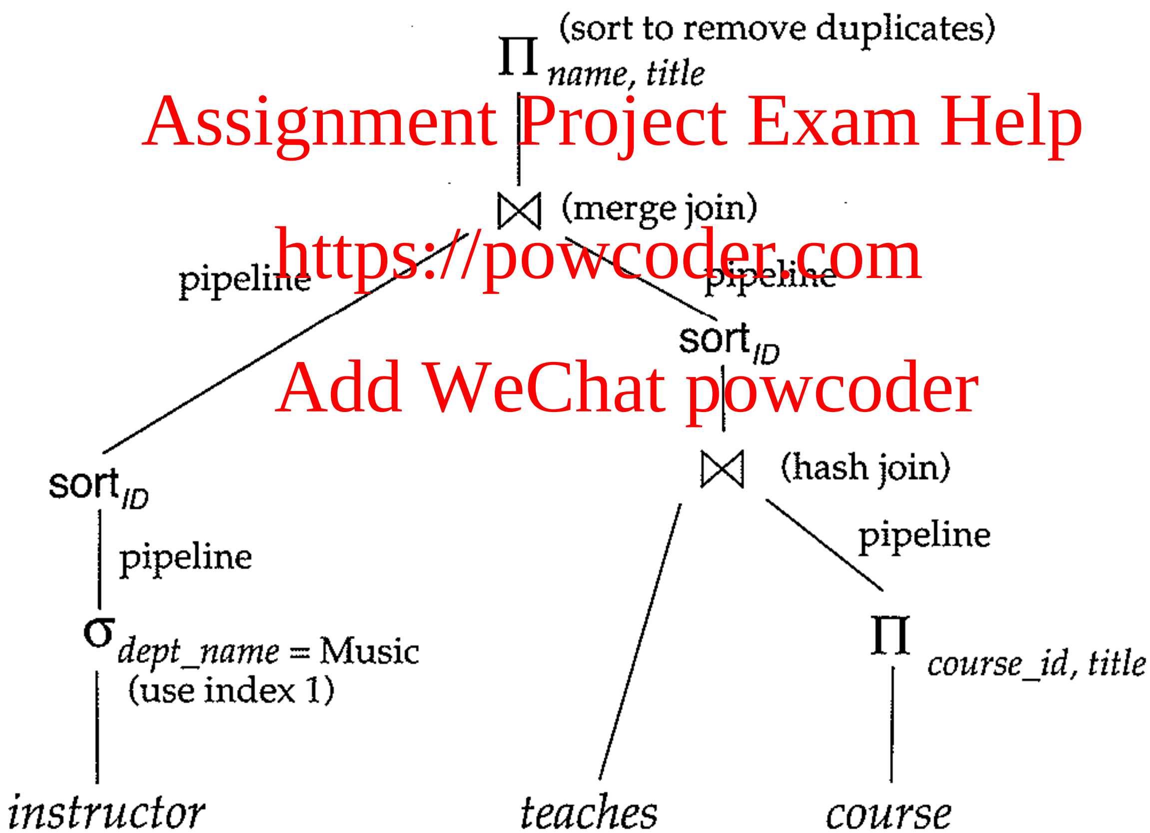
course(*course_id*, *title*, *dept_name*, *credits*)



(a) Initial expression tree (b) Transformed expression tree

Query Optimization (Cont.)

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



Query Optimization (Cont.)

- Cost difference between evaluation plans for a query can be enormous
  - E.g. seconds vs. days in some cases
- Steps in **cost-based query optimization**
  1. Generate logically equivalent expressions using **equivalence rules**
  2. Annotate resultant expressions to get alternative query plans
  3. Choose the cheapest plan based on **estimated cost**
- Estimation of plan cost based on:
  - Statistical information about relations.
    - Examples: no. of tuples, index depths
  - Cost formulae for algorithms, computed using statistics
  - Statistics estimation for intermediate results
    - to compute cost of complex expressions

Query Processing and Optimization

- Overview
- Measures of Query Cost
- Selection Operation
- Sort Operation
- Join Operation
  - nested-loop, block nested-loop, indexed nested-loop, merge-join and hash join
- Other Operations
- Evaluation of Expressions
- **Transformation of Relational Expressions**
- Statistics Estimation
- Choices of Evaluation Plans

Transformation of Relational Expressions

- Two relational algebra expressions are said to be **equivalent** if the two expressions generate the same set of tuples on every *legal* database instance
  - Note: order of tuples is irrelevant
- An **equivalence rule** says that expressions of two forms are equivalent
  - Can replace expression of first form by second, or vice versa
- However, the rules do not say that one is better than the other.
- Notation:
  - θ : predicate
  - L : list of attributes
  - E : relation or relational-algebra expression

Equivalence Rules

1. Conjunctive selection operations can be deconstructed into a sequence of individual selections.

$$\sigma_{\theta_1 \wedge \theta_2}(E) \equiv \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$

2. Selection operations are commutative.

$$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) \equiv \sigma_{\theta_2}(\sigma_{\theta_1}(E))$$

3. Only the last in a sequence of projection operations is needed, the others can be omitted.

$$\prod_{L_1}(\prod_{L_2}(\ldots(\prod_{L_n}(E))\ldots)) \equiv \prod_{L_1}(E)$$

where $L_1 \subseteq L_2 \ldots \subseteq L_n$

4. Selections can be combined with Cartesian products and theta joins.
   a. $\sigma_{\theta}(E_1 \times E_2) \equiv E_1 \bowtie_{\theta} E_2$
   b. $\sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) \equiv E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$

Equivalence Rules (Cont.)

5. Theta-join operations (and natural joins) are commutative.

$$E_1 \bowtie E_2 \equiv E_2 \bowtie E_1$$

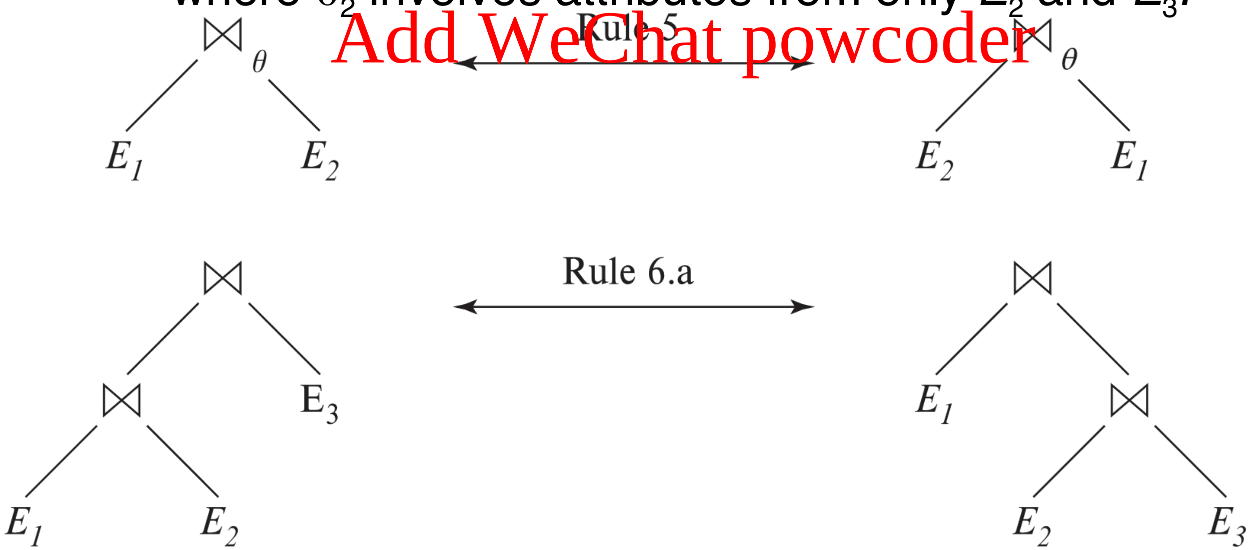
6. (a) Natural join operations are associative:

$$(E_1 \bowtie E_2) \bowtie E_3 \equiv E_1 \bowtie (E_2 \bowtie E_3)$$

(b) Theta joins are associative in the following manner:

$$(E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \wedge \theta_3} E_3 \equiv E_1 \bowtie_{\theta_1 \wedge \theta_3} (E_2 \bowtie_{\theta_2} E_3)$$

where θ_2 involves attributes from only E_2 and E_3.



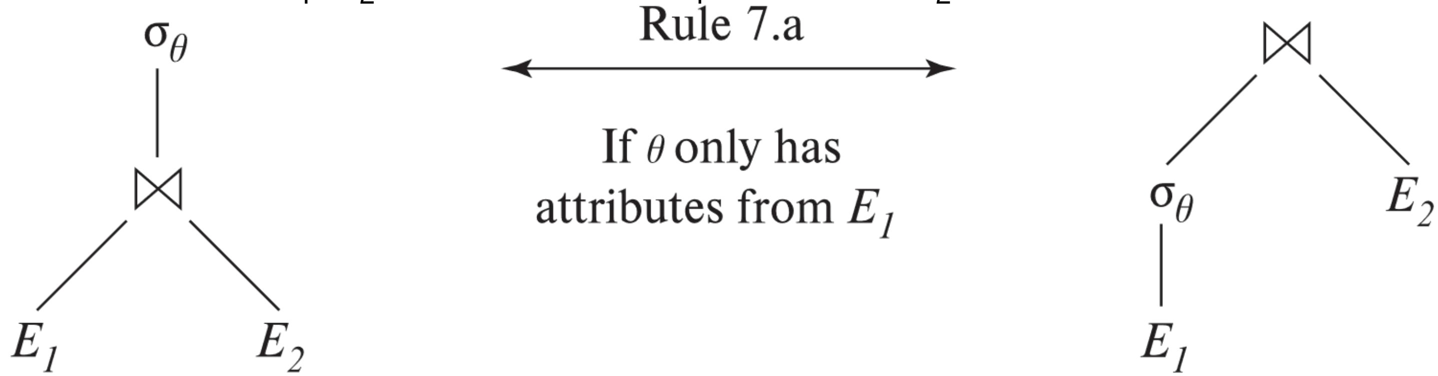
Equivalence Rules (Cont.)

7. The selection operation distributes over the theta join operation under the following two conditions:
(a) When all the attributes in θ_0 involve only the attributes of one of the expressions (E_1) being joined.

$$\sigma_{\theta_0}(E_1 \bowtie_\theta E_2) \equiv (\sigma_{\theta_0}(E_1)) \bowtie_\theta E_2$$

(b) When θ_1 involves only the attributes of E_1 and θ_2 involves only the attributes of E_2.

$$\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_\theta E_2) \equiv (\sigma_{\theta_1}(E_1)) \bowtie_\theta (\sigma_{\theta_2}(E_2))$$



Equivalence Rules (Cont.)

8. The projection operation distributes over the theta join operation as follows:

   (a) Let L_1 and L_2 be sets of attributes from E_1 and E_2, respectively, if θ involves only attributes from $L_1 \cup L_2$:

$$\prod_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) \equiv \prod_{L_1}(E_1) \bowtie_\theta \prod_{L_2}(E_2)$$

   (b) Consider a join $E_1 \bowtie_\theta E_2$.

   - Let L_1 and L_2 be sets of attributes from E_1 and E_2, respectively.

   - Let L_3 be attributes of E_1 that are involved in join condition θ, but are not in $L_1 \cup L_2$, and
   - let L_4 be attributes of E_2 that are involved in join condition θ, but are not in $L_1 \cup L_2$.

$$\prod_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) \equiv \prod_{L_1 \cup L_2}(\prod_{L_1 \cup L_3}(E_1) \bowtie_\theta \prod_{L_2 \cup L_4}(E_2))$$

Similar equivalences hold for outerjoin operations: ⟗, ⟕, and ⟖

Equivalence Rules (Cont.)

9. The set operations union and intersection are commutative

$$E_1 \cup E_2 \equiv E_2 \cup E_1$$

$$E_1 \cap E_2 \equiv E_2 \cap E_1$$

- (set difference is not commutative).

10. Set union and intersection are associative.

$$(E_1 \cup E_2) \cup E_3 \equiv E_1 \cup (E_2 \cup E_3)$$

$$(E_1 \cap E_2) \cap E_3 \equiv E_1 \cap (E_2 \cap E_3)$$

11. The selection operation distributes over $\cup$, $\cap$ and $-$.

a. $\sigma_\theta (E_1 \cup E_2) \equiv \sigma_\theta (E_1) \cup \sigma_\theta(E_2)$

b. $\sigma_\theta (E_1 \cap E_2) \equiv \sigma_\theta (E_1) \cap \sigma_\theta(E_2)$

c. $\sigma_\theta (E_1 - E_2) \equiv \sigma_\theta (E_1) - \sigma_\theta(E_2)$

d. $\sigma_\theta (E_1 \cap E_2) \equiv \sigma_\theta(E_1) \cap E_2$

e. $\sigma_\theta (E_1 - E_2) \equiv \sigma_\theta(E_1) - E_2$

Equivalence Rules (Cont.)

12. The projection operation distributes over union

$$\Pi_L(E_1 \cup E_2) \equiv (\Pi_L(E_1)) \cup (\Pi_L(E_2))$$

13. Selection distributes over aggregation as below

$$\sigma_\theta({}_G\gamma_A(E)) \equiv {}_G\gamma_A(\sigma_\theta(E))$$

provided θ only involves attributes in G

Transformation Example: Pushing Selections

- Query: Find the names of all instructors in the Music department, along with the titles of the courses that they teach
  - $\Pi_{name,\ title}(\sigma_{dept\_name=\text{"Music"}}$

$$(instructor \bowtie (teaches \bowtie \Pi_{course\_id,\ title}\ (course))))$$

- Transformation using rule 7a.

$$\Pi_{name,\ title}((\sigma_{dept\_name=\text{"Music"}}(instructor)) \bowtie$$

$$(teaches \bowtie \Pi_{course\_id,\ title}\ (course)))$$

- ***Performing the selection as early as possible*** reduces the size of the relation to be joined.

instructor(*ID*, *name*, *dept_name*, *salary*)
teaches(*ID*, *course_id*, *sec_id*, *semester*, *year*)
course(*course_id*, *title*, *dept_name*, *credits*)

Example with Multiple Transformations

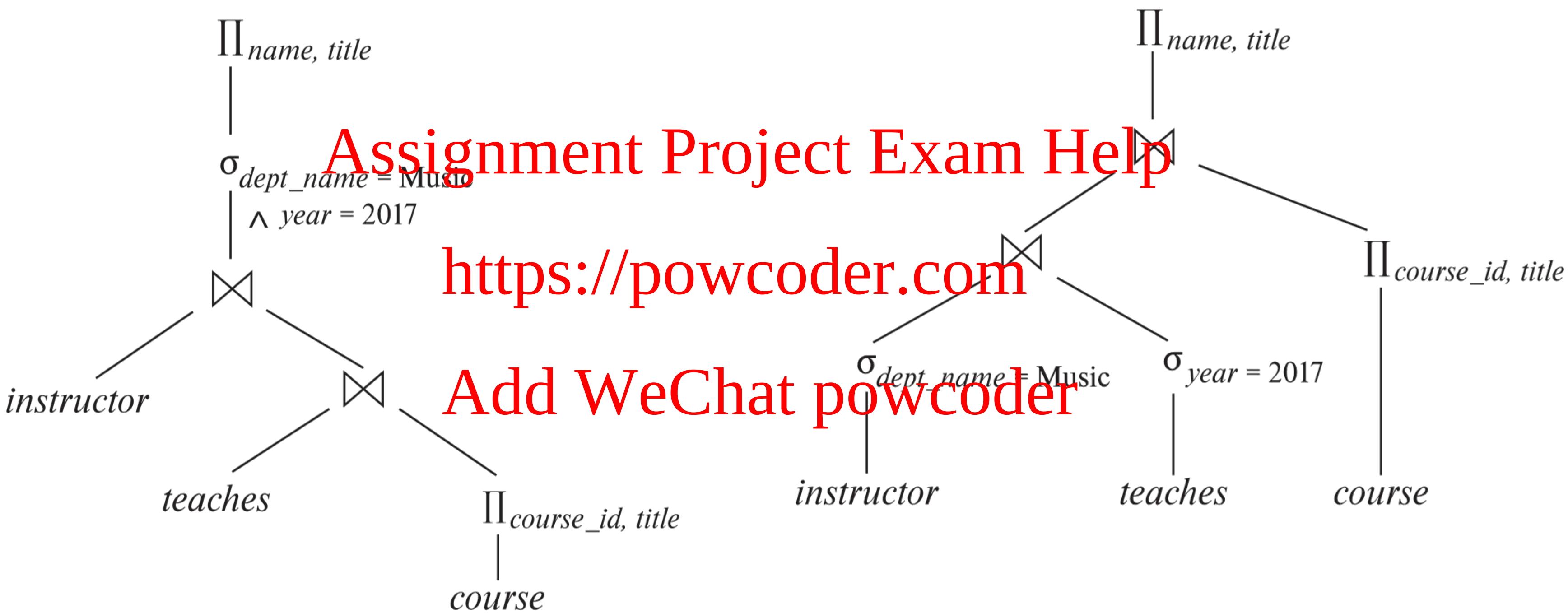
- Query: Find the names of all instructors in the Music department who have taught a course in 2009, along with the titles of the courses that they taught
  - $\Pi_{name,\ title}(\sigma_{dept\_name=\text{"Music"} \wedge year = 2009}$
    $(instructor \bowtie (teaches \bowtie \Pi_{course\_id,\ title}(course))))$

- Transformation using join associativity (Rule 6a):

  - $\Pi_{name,\ title}(\sigma_{dept\_name=\text{"Music"} \wedge year = 2009}$
    $((instructor \bowtie teaches) \bowtie \Pi_{course\_id,\ title}(course)))$
- Second form provides an opportunity to apply the "perform selections early" rule (Rule 7b), resulting in the subexpression

$$\sigma_{dept\_name = \text{"Music"}}(instructor) \bowtie \sigma_{year = 2009}(teaches)$$

Multiple Transformations (Cont.)



(a) Initial expression tree

(b) Tree after multiple transformations

Transformation Example: Pushing Projections

- Consider: $\Pi_{name,\ title}((\sigma_{dept\_name=\text{"Music"}}(instructor) \bowtie teaches)$

  $\bowtie \Pi_{course\_id,\ title}(course))$

- When we compute

  $(\sigma_{dept\_name = \text{"Music"}}(instructor) \bowtie teaches)$

  we obtain a relation whose schema is:

  (ID, $name$, $dept\_name$, $salary$, $course\_id$, sec_id, $semester$, $year$)

- Push projections using equivalence rule 8b; eliminate unneeded attributes from intermediate results to get:

  $\Pi_{name,\ title}((\Pi_{name,\ course\_id}$

  $((\sigma_{dept\_name=\text{"Music"}}(instructor)) \bowtie teaches))$

  $\bowtie \Pi_{course\_id,\ title}(course))$

- ***Performing the projection as early as possible*** reduces the size of the relation to be joined.

Join Ordering Example

- A good join ordering is important for reducing the size of temporary results.
- For all relations r_1, r_2, and r_3,

$$(r_1 \bowtie r_2) \bowtie r_3 = r_1 \bowtie (r_2 \bowtie r_3)$$

(Join Associativity) (Rule 6a)

- If $r_2 \bowtie r_3$ is quite large and $r_1 \bowtie r_2$ is small, we choose

$$(r_1 \bowtie r_2) \bowtie r_3$$

so that we compute and store a smaller temporary relation.

Join Ordering Example (Cont.)

- Consider the expression

$$\Pi_{name,\ title}((\sigma_{dept\_name=\text{“Music”}}(instructor)) \bowtie teaches \bowtie \Pi_{course\_id,\ title}(course))$$

- Could compute

$$teaches \bowtie \Pi_{course\_id,\ title}(course)$$

first but the result is likely to be a large relation because it contains one tuple for every course.

- Only a small fraction of the university's instructors are likely to be from the Music department, it is better to compute

$$\sigma_{dept\_name=\text{“Music”}}(instructor) \bowtie teaches$$

first.

Query Processing and Optimization

- Overview
- Measures of Query Cost
- Selection Operation
- Sort Operation
- Join Operation
  - nested-loop, block nested-loop, indexed nested-loop, merge-join and hash join
- Other Operations
- Evaluation of Expressions
- Transformation of Relational Expressions
- **Statistics Estimation**
- Choices of Evaluation Plans

Statistics Estimation

- *Reminder*: Cost of each operator needs statistics of input relations
- The database-system catalog stores the following statistical information about database relations.
  - n_r : number of tuples in a relation *r.*
  - b_r : number of blocks containing tuples of *r.*
  - l_r : size of a tuple of *r.*
  - f_r : blocking factor of *r*, i.e., the number of tuples of *r* that fit into one block.
  - $V(A, r)$: number of distinct values that appear in *r* for attribute *A;* same as the size of $\prod_A(r)$.
  - If tuples of *r* are stored together physically in a file, then:

$$b_r = \left\lceil \frac{n_r}{f_r} \right\rceil$$

  - Others: heights of B^+-tree indices, no. of leaf pages in the indices
- However, inputs can be results of sub-expressions
  - Need to estimate statistics of expression results

Statistics Estimation (Cont.)

- A query-evaluation plan that has the lowest *estimated* execution cost may NOT have the lowest *actual* execution cost because the estimates are based on assumptions that may not hold exactly.
- However, real-world experience has shown that the plans with the lowest estimated costs usually have actual execution costs that are either *the lowest*, or are *close to the lowest*.

Selection Size Estimation

- $\sigma_{A=V}(r)$
  - Assume uniform distribution (though may not be correct)
  - $n_r / V(A,r)$: number of records that will satisfy the selection
  - Equality condition on a key attribute: *size estimate* = 1

- $\sigma_{A \leq V}(r)$ (case of $\sigma_{A \geq V}(r)$ is symmetric)
  - Let c denote the estimated number of tuples satisfying the condition.
  - If $\min(A,r)$ and $\max(A,r)$ are available in catalog
    - $c = 0$ if $v < \min(A,r)$
    - $c = n_r$ if $v \geq \max(A,r)$
    - $c = n_r \cdot \dfrac{v - \min(A,r)}{\max(A,r) - \min(A,r)}$, otherwise
  - If the value v is not available, c is assumed to be $n_r / 2$.

Size Estimation of Complex Selections

- The **selectivity** of a condition θ_i is the probability that a tuple in the relation r satisfies θ_i .
  - If s_i is the number of satisfying tuples in r, the selectivity of θ_i is given by s_i / n_r.

- **Conjunction:** $\sigma_{\theta 1 \wedge \theta 2 \wedge \ldots \wedge \theta n}(r)$. Assuming *independence*, estimate of tuples in the result is:

$$n_r * \frac{s_1 * s_2 * \ldots * s_n}{n_r^n}$$

- **Disjunction:** $\sigma_{\theta 1 \vee \theta 2 \vee \ldots \vee \theta n}(r)$. Estimated number of tuples:

$$n_r * \left(1 - (1 - \frac{s_1}{n_r}) * (1 - \frac{s_2}{n_r}) * \ldots * (1 - \frac{s_n}{n_r})\right)$$

- **Negation:** $\sigma_{\neg\theta}(r)$. Estimated number of tuples:

$$n_r - size(\sigma_\theta(r))$$

Estimation of the Size of Joins

- The Cartesian product $r \times s$ contains $n_r * n_s$ tuples; each tuple occupies $l_r + l_s$ bytes.
- If $R \cap S = \emptyset$, then $r \bowtie s$ is the same as $r \times s$.
- If $R \cap S$ is a key for R, then a tuple of s will join with at most one tuple from r
  - therefore, the number of tuples in $r \bowtie s$ is ***no greater than*** the number of tuples in s.
- If $R \cap S$ in S is a foreign key in S referencing R, then the number of tuples in $r \bowtie s$ is ***exactly the same*** as the number of tuples in s.

Estimation of the Size of Joins (Cont.)

- If $R \cap S = \{A\}$ is not a key for R or S.
  If we assume that every tuple t in R produces tuples in $R \bowtie S$, the number of tuples in $R \bowtie S$ is estimated to be:

$$\frac{n_r * n_s}{V(A,s)}$$

  If the reverse is true, the estimate obtained will be:

$$\frac{n_r * n_s}{V(A,r)}$$

  The lower of these two estimates is probably the more accurate one.

Join Operation Example

Running example:

student ⋈ *takes*

Catalog information for join examples:

- $n_{student}$ = 5,000.
- $f_{student}$ = 50, which implies that $b_{student}$ =5000/50 = 100.
- n_{takes} = 10000.
- f_{takes} = 25, which implies that b_{takes} = 10000/25 = 400.
- *V*(*ID, student*) = 5000 (*primary key!*)
- *V(ID, takes)* = 2500, which implies that on average, each student who has taken a course has taken 4 courses.
  - Attribute *ID* in *takes* is a foreign key referencing *student.*

Join Operation Example (Cont.)

- Example: *student* $\bowtie$ *takes, ID* in *takes* is a foreign key referencing *student*
  - hence, the result has exactly n_{takes} tuples, which is 10000
- Example: compute the size estimates for *student* $\bowtie$ *takes* without using information about foreign keys:
  - $V(ID, takes) = 2500$, and $V(ID, student) = 5000$
  - The two estimates are 5000 * 10000/2500 = 20,000 and 5000 * 10000/5000 = 10000
  - We choose the lower estimate, which in this case, is the same as our earlier computation using foreign keys.

Size Estimation for Other Operations

- Projection: estimated size of $\prod_A(r) = V(A,r)$, duplicates eliminated
- Aggregation: estimated size of ${}_G\gamma_F(r) = V(G,r)$, one tuple in ${}_G\gamma_F(r)$ for each distinct value of G.
- Set operations
  - For unions/intersections of selections on the ***same*** relation: rewrite and use size estimate for selections
    - E.g. $\sigma_{\theta 1}(r) \cup \sigma_{\theta 2}(r)$ can be rewritten as $\sigma_{\theta 1 \vee \theta 2}(r)$
  - For operations on ***different*** relations:
    - estimated size of $r \cup s$ = size of r + size of s.
    - estimated size of $r \cap s$ = minimum size of r and size of s.
    - estimated size of $r - s = r$.
    - All the three estimates may be quite inaccurate, but provide upper bounds on the sizes.

Estimation of Number of Distinct Values

Selections: $\sigma_\theta(r)$

- If θ forces A to take a specified value:

  $V(A,\sigma_\theta(r)) = 1.$

  - e.g., $A = 3$

- If θ forces A to take on one of a specified set of values:

  $V(A,\sigma_\theta(r))$ = number of specified values.

  - e.g., $(A = 1 \vee A = 3 \vee A = 4)$

- If the selection condition θ is of the form A *op* v, *where op* is a comparison operator:

  estimated $V(A,\sigma_\theta(r)) = V(A, r) * s$

  where s is the selectivity of the selection.

- In all the other cases: use approximate estimate of

  $\min(V(A,r), n_{\sigma\theta(r)})$

Estimation of Distinct Values (Cont.)

Joins: $r \bowtie s$

- If all attributes in A are from r

  estimated $V(A, r \bowtie s) = \min(V(A, r), n_{r \bowtie s})$

- If A contains attributes $A1$ from r and $A2$ from s, then

  estimated $V(A, r \bowtie s) =$

  $\min(V(A1, r) * V(A2 - A1, s), V(A1 - A2, r) * V(A2, s), n_{r \bowtie s})$

Estimation of Distinct Values (Cont.)

- Estimation of distinct values are straightforward for projections.
  - They are the same in $\prod_A(r)$ as in r.
- The same holds for grouping attributes of aggregation.
- For aggregated values
  - For min(A) and max(A), the number of distinct values can be estimated as min($V(A,r)$, $V(G,r)$) where G denotes grouping attributes
  - For other aggregates, assume all values are distinct, and use $V(G,r)$

Query Processing and Optimization

- Overview
- Measures of Query Cost
- Selection Operation
- Sort Operation
- Join Operation
  - nested-loop, block nested-loop, indexed nested-loop, merge-join and hash join
- Other Operations
- Evaluation of Expressions
- Transformation of Relational Expressions
- Statistics Estimation
- **Choices of Evaluation Plans**

Choice of Evaluation Plans

- Choosing the cheapest algorithm for each operation independently may not yield best overall algorithm.
  - Must consider the interaction of evaluation techniques when choosing evaluation plans
    - merge-join may be costlier than hash-join, but may provide a sorted output which reduces the cost for an outer level aggregation.
    - nested-loop join may provide opportunity for pipelining
- Practical query optimizers incorporate elements of the following two broad approaches:

1. Search all the plans and choose the best plan in a cost-based fashion.
2. Uses heuristics to choose a plan (at the potential risk of not finding the optimal plan).

Cost-Based Optimization

- Consider finding the best join-order for $r_1 \bowtie r_2 \bowtie \ldots \bowtie r_n$.
- There are $(2(n-1))!/(n-1)!$ different join orders for above expression.
  - $n = 3$, the number is 12
  - $n = 7$, the number is 665280
  - $n = 10$, the number is greater than 176 billion!

- No need to generate all the join orders. Using dynamic programming, the least-cost join order for any subset of $\{r_1, r_2, \ldots r_n\}$ can be computed only once and stored for future use.

  - Example: Find the best join order of the form $(r_1 \bowtie r_2 \bowtie r_3) \bowtie r_4 \bowtie r_5$

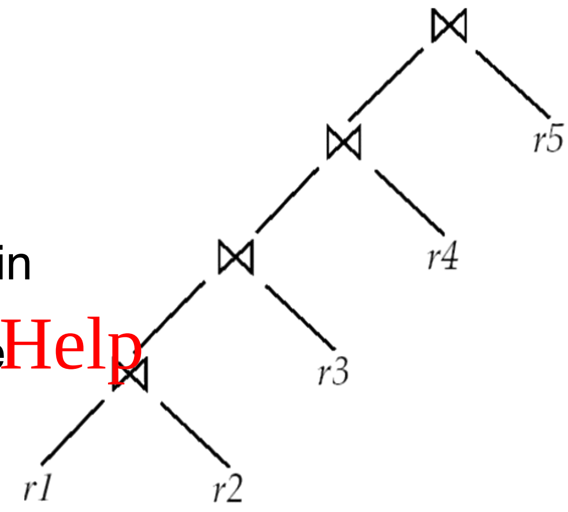
  - There are 12 different join orders for computing $r_1 \bowtie r_2 \bowtie r_3$ and 12 orders for computing the join of this result with r_4 and r_5.
  - We first find the best join order for the subset of relations $\{r_1, r_2, r_3\}$, we can use that order for further joins with r_4 and r_5, and ignore all more expensive join orders of $r_1 \bowtie r_2 \bowtie r_3$.
  - Thus, only 12 + 12 choices, instead of 144 join orders are examined.
- *Read pseudo-code in book for details.*
- Cost-based optimization is expensive, but worthwhile for queries on large datasets (typical queries have small n, generally < 10)

Heuristic Optimization

- Cost-based optimization is expensive.
- Systems may use *heuristics* to reduce the number of choices that must be made in a cost-based fashion.
- Heuristic optimization transforms the query-tree by using a set of rules that typically (but not in all cases) improve execution performance:
  - Perform selection early (reduces the number of tuples)
  - Perform projection early (reduces the number of attributes)
  - Perform most restrictive selection and join operations (i.e. with smallest result size) before other similar operations.

Heuristic Optimization (Cont.)

- Many optimizers consider only ***left-deep join orders*** (instead of all join orders).
  - Right-hand-side input for each join is a relation, not the result of an intermediate join
  - 👍 Convenient for pipelined evaluation since the right operand is a stored relation
  - 👍 Reduces optimization complexity
    - For n=3, there are 6 ways



(a) Left-deep join tree

- Some versions of Oracle, consider *n* evaluation plans in a left-deep join order for an *n*-way join
  - Starting from each of the *n* relations
  - Repeatedly pick "best" relation to join next on the basis of a ranking of the available access paths

Heuristic Optimization (Cont.)

- Search for optimal plan is terminated when the ***optimization cost budget*** is exceeded and the best plan found so far is returned
  - Optimizers usually first apply cheap heuristics to find a plan
  - Then, start a full cost-based optimization with a budget based on the heuristically chosen plan
- ***Plan caching*** reuses previous optimal query plan if query is resubmitted (with different constants in query).
  - Example: a query to find the courses for which a student has registered in a university application.

Concluding Remarks

- Real-life observations
  - The difference in execution time between a good plan and a bad one may be ***huge***.
  - The added ***cost*** of cost-based query optimization is usually ***more than offset by the saving*** at query-execution time, which is dominated by slow disk accesses.
  - The achieved saving is magnified in applications that run on a regular basis, where a query can be ***optimized once***, and the selected query plan can be ***used many times***.