

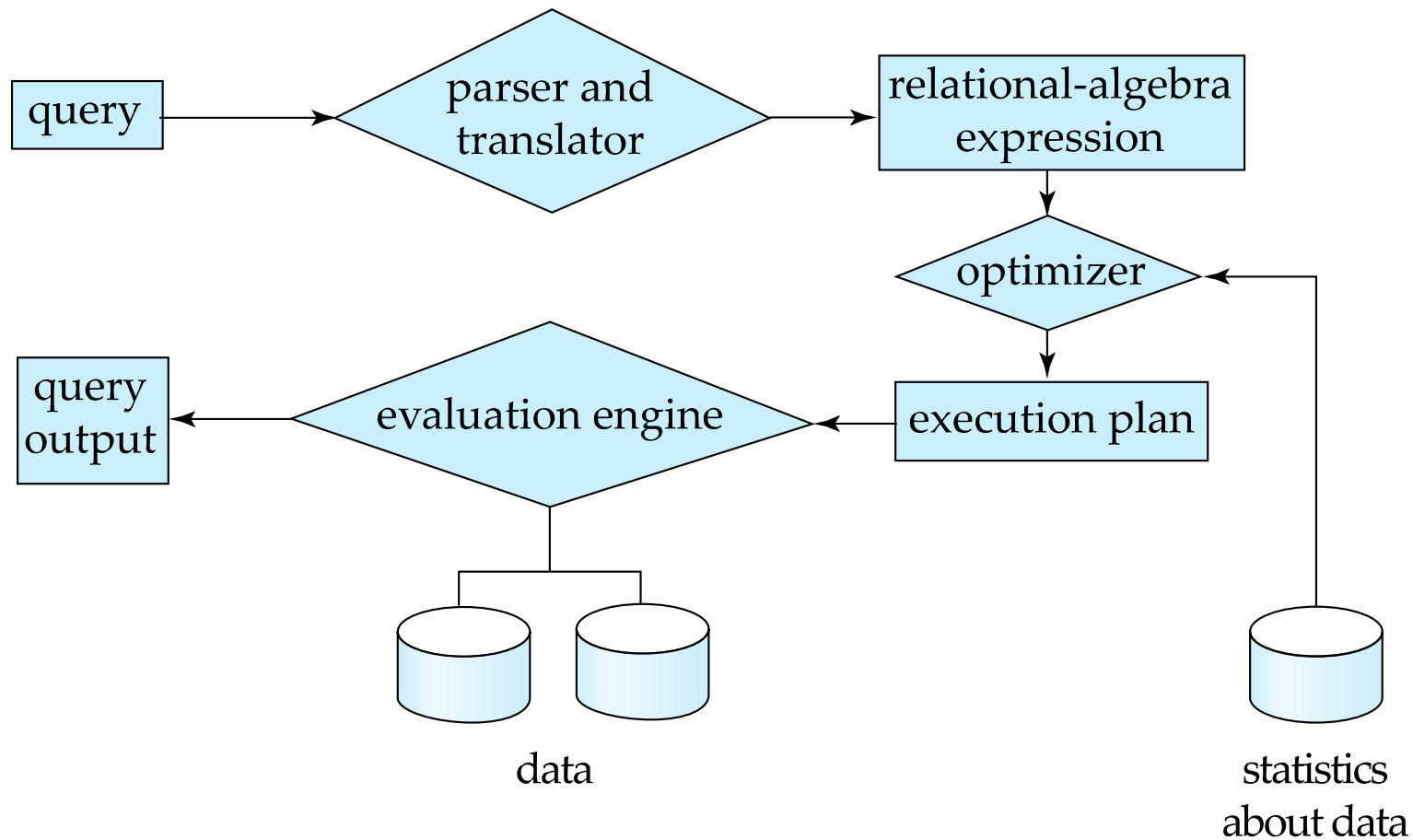
Query Processing

Basic Steps in Query Processing

1. Parsing and translation

2. Optimization

3. Evaluation



Basic Steps in Query Processing (Cont.)

- Parsing and translation
 - translate the query into its internal form. This is then translated into relational algebra.
 - Parser checks syntax, verifies relations
- 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 : Optimization

- A relational algebra expression may have many equivalent expressions
 - E.g., $\sigma_{salary < 75000}(\Pi_{salary}(instructor))$ is equivalent to $\Pi_{salary}(\sigma_{salary < 75000}(instructor))$
- 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 **evaluation-plan**.
 - E.g., can use an index on *salary* to find instructors with salary < 75000,
 - or can perform complete relation scan and discard instructors with salary ≥ 75000

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.
- In this chapter 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
- In Chapter 14
 - We study how to optimize queries, that is, how to find an evaluation plan with lowest estimated cost

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 IO 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

- Algorithm **A1** (**linear search**). 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 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.
- **A2 (primary index, equality on key)**. Retrieve a single record that satisfies the corresponding equality condition
 - $Cost = (h_i + 1) * (t_T + t_S)$
- **A3 (primary index, equality on nonkey)** Retrieve multiple records.
 - Records will be on consecutive blocks
 - Let b = number of blocks containing matching records
 - $Cost = h_i * (t_T + t_S) + t_S + t_T * b$

Selections Using Indices

- **A4 (secondary index, equality on nonkey).**
 - 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 $\sigma_{A \geq v}(r)$ by using
 - a linear file scan,
 - or by using indices in the following ways:
- **A5 (primary index, comparison)**. (Relation is sorted on A)
 - For $\sigma_{A \geq v}(r)$ use index to find first tuple $\geq v$ and scan relation sequentially from there
 - For $\sigma_{A \leq v}(r)$ just scan relation sequentially till first tuple $> v$; do not use index
- **A6 (secondary index, comparison)**.
 - For $\sigma_{A \geq v}(r)$ use index to find first index entry $\geq v$ and scan index sequentially from there, to find pointers to records.
 - For $\sigma_{A \leq v}(r)$ just scan leaf pages of index finding pointers to records, till first entry $> v$
 - In either case, retrieve records that are pointed to
 - requires an I/O for each record
 - Linear file scan may be cheaper

Implementation of Complex Selections

- **Conjunction:** $\sigma_{\theta_1 \wedge \theta_2 \wedge \dots \wedge \theta_n}(r)$
- **A7 (conjunctive selection using one index).**
 - Select a combination of θ_i and algorithms A1 through A7 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

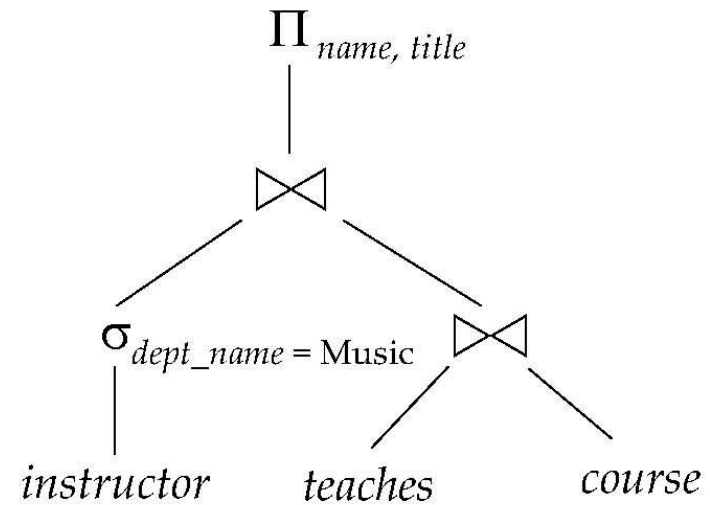
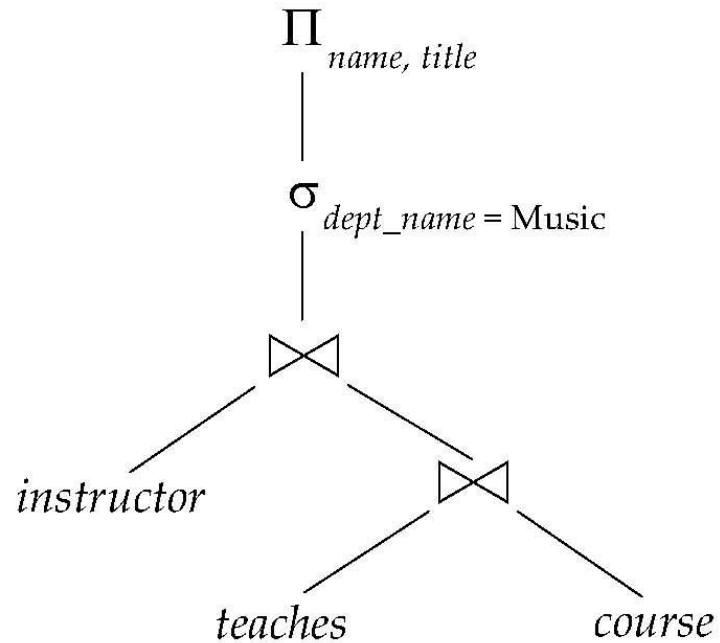
- **Disjunction:** $\sigma_{\theta_1 \vee \theta_2 \vee \dots \vee \theta_n}(r)$.
- **A10 (disjunctive selection by union of identifiers).**
 - Applicable if *all* conditions have available indices.
 - Otherwise use linear scan.
 - Use corresponding index for each condition, and take union of all the 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 Optimization

Introduction

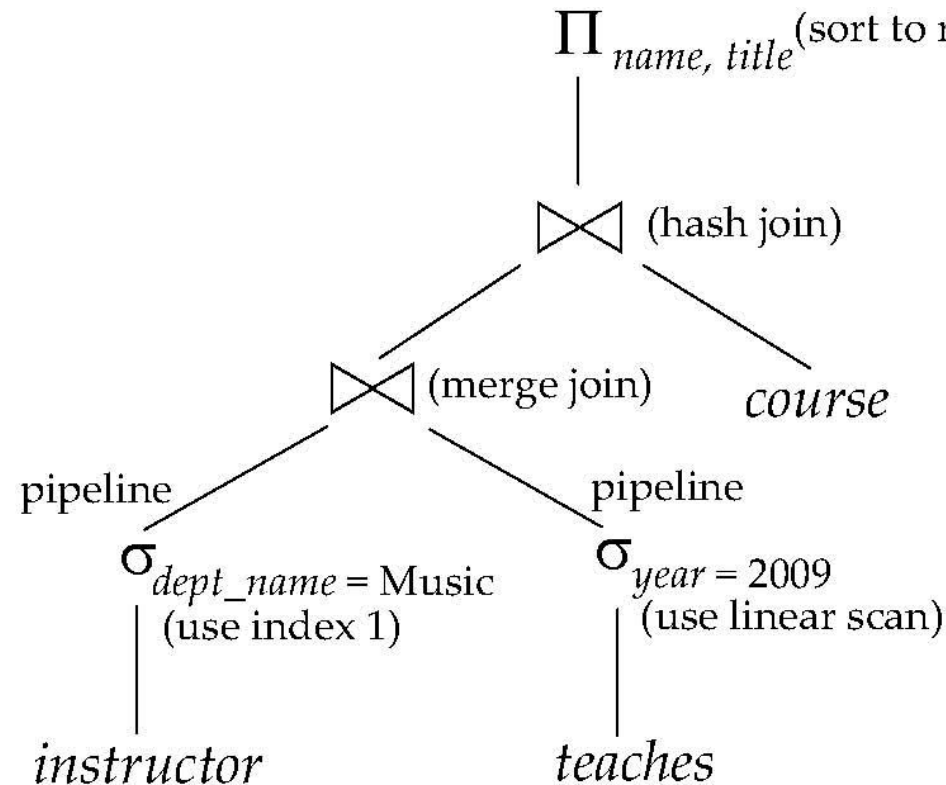
Alternative ways of evaluating a given query

- Equivalent expressions
- Different algorithms for each operation



Introduction (Cont.)

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



- Find out how to view query execution plans on your favorite database

Introduction (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:
 - number of tuples, number of distinct values for an attribute
 - Statistics estimation for intermediate results
 - to compute cost of complex expressions
 - Cost formulae for algorithms, computed using statistics

Generating Equivalent Expressions

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
 - we don't care if they generate different results on databases that violate integrity constraints
- In SQL, inputs and outputs are multisets of tuples
 - Two expressions in the multiset version of the relational algebra are said to be equivalent if the two expressions generate the same multiset of tuples on every legal database instance.
- An **equivalence rule** says that expressions of two forms are equivalent
 - Can replace expression of first form by second, or vice versa

Equivalence Rules

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

$$S_{q_1 \cup q_2}(E) = S_{q_1}(S_{q_2}(E))$$

2. Selection operations are commutative.

$$S_{q_1}(S_{q_2}(E)) = S_{q_2}(S_{q_1}(E))$$

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

$$\Pi_{L_1}(\Pi_{L_2}(\mathbf{K}(\Pi_{L_n}(E))\mathbf{K})) = \Pi_{L_1}(E)$$

4. Selections can be combined with Cartesian products and theta joins.

- a. $\sigma_{\theta}(E_1 \times E_2) = E_1 \bowtie_{\theta} E_2$

- b. $\sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) = E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$

Equivalence Rules (Cont.)

⊗ ⊗

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

$$E_1 \bowtie_{\theta} E_2 = E_2 \bowtie_{\theta} E_1$$

6. (a) Natural join operations are associative:

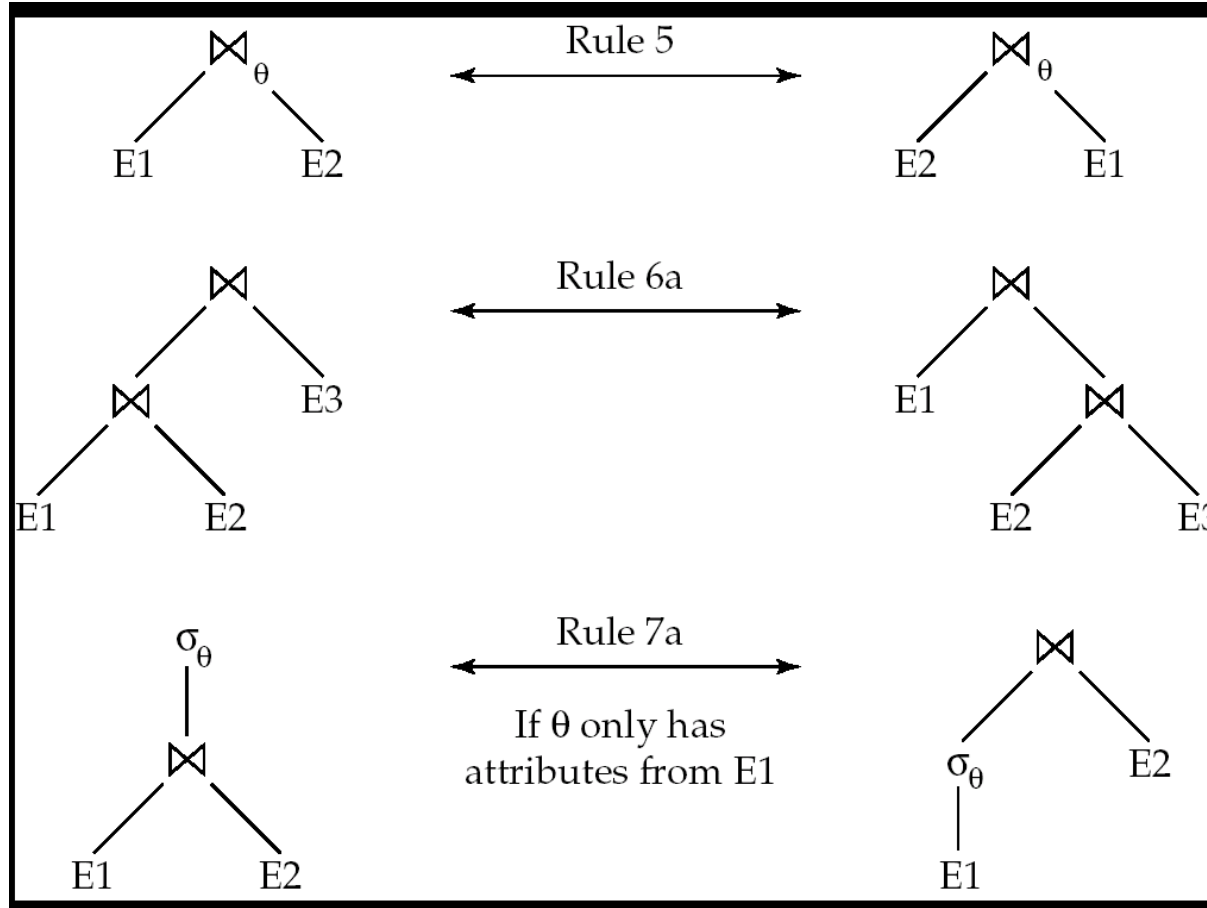
$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$$

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

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

where θ_2 involves attributes from only E_2 and E_3 .

Pictorial Depiction of Equivalence Rules



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.

$$\sigma_{\theta_0}(E_1 \bowtie E_2) = (\sigma_{\theta_0}(E_1)) \bowtie E_2$$

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

$$\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie E_2) = (\sigma_{\theta_1}(E_1)) \bowtie (\sigma_{\theta_2}(E_2))$$

Equivalence Rules (Cont.)

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

(a) if θ involves only attributes from $L_1 \cup L_2$:

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_{\theta} E_2) = (\Pi_{L_1}(E_1)) \bowtie_{\theta} (\Pi_{L_2}(E_2))$$

(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 $L_1 \cup L_2$.

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_{\theta} E_2) = \Pi_{L_1 \cup L_2}((\Pi_{L_1 \cup L_3}(E_1)) \bowtie_{\theta} (\Pi_{L_2 \cup L_4}(E_2)))$$

Equivalence Rules (Cont.)

9. The set operations union and intersection are commutative

$$E_1 \cup E_2 = E_2 \cup E_1$$

$$E_1 \cap E_2 = E_2 \cap E_1$$

■ (set difference is not commutative).

10. Set union and intersection are associative.

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

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

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

$$\sigma_{\theta}(E_1 - E_2) = \sigma_{\theta}(E_1) - \sigma_{\theta}(E_2)$$

and similarly for \cup and \cap in place of $-$

Also:
$$\sigma_{\theta}(E_1 - E_2) = \sigma_{\theta}(E_1) - E_2$$

and similarly for \cap in place of $-$, but not for \cup

12. The projection operation distributes over union

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

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.

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

- $\Pi_{name, title}(\sigma_{dept_name = \text{“Music”} \wedge year = 2009} (instructor \text{ teaches } \Pi_{course_id, title} (course))))$

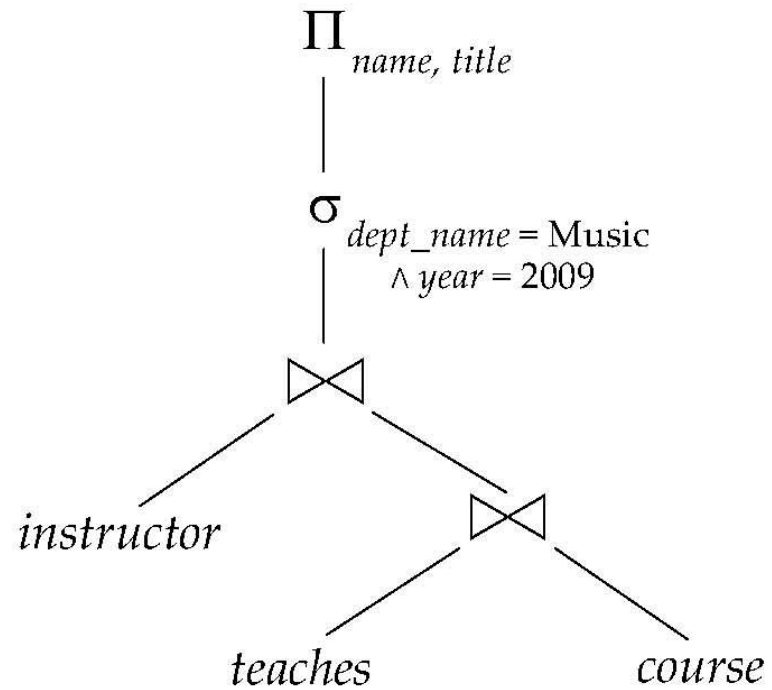
- Transformation using join associatively (Rule 6a):

- $\Pi_{name, title}(\sigma_{dept_name = \text{“Music”} \wedge year = 2009} ((instructor \text{ teaches}) \Pi_{course_id, title} (course))))$

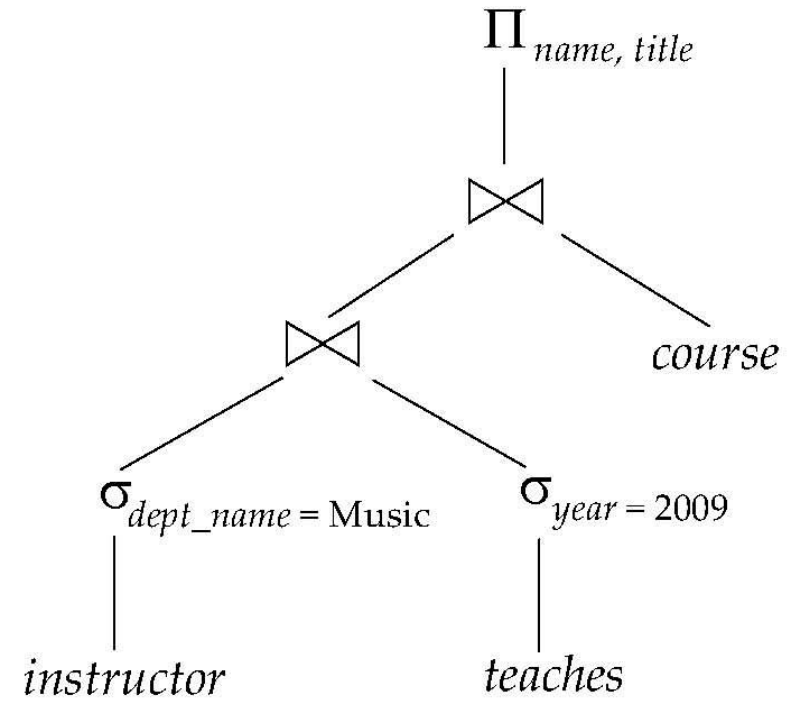
- Second form provides an opportunity to apply the “perform selections early” rule, resulting in the subexpression

$$\sigma_{dept_name = \text{“Music”}} (instructor) \quad \sigma_{year = 2009} (teaches)$$

Multiple Transformations (Cont.)



(a) Initial expression tree



(b) Tree after multiple transformations

Transformation Example: Pushing Projections



- Consider: $\Pi_{name, title}(\sigma_{dept_name = \text{“Music”}}(\Pi_{course_id, title}(instructor \text{ teaches})))$

- When we compute

$$(\sigma_{dept_name = \text{“Music”}}(instructor \text{ teaches}))$$

we obtain a relation whose schema is:

$(ID, name, dept_name, salary, course_id, sec_id, semester, year)$

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



$$\Pi_{name, title}(\Pi_{name, course_id}(\sigma_{dept_name = \text{“Music”}}(\Pi_{course_id, title}(instructor \text{ teaches}))))$$

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

Join Ordering Example

- 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)

- 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

$$\Pi_{name, title}(\sigma_{dept_name = \text{“Music”}}(instructor) \bowtie \Pi_{course_id, title}(course))$$

- Could compute $\Pi_{course_id, title}(course)$ first, and join result with

$$\sigma_{dept_name = \text{“Music”}}(instructor)$$

but the result of the first join is likely to be a large relation.

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

$$\sigma_{dept_name = \text{“Music”}}(instructor) \bowtie$$

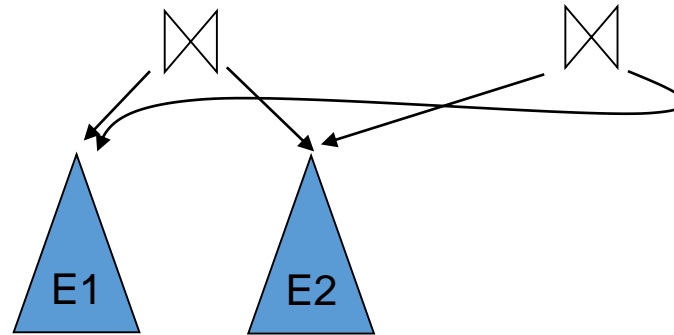
first.

Enumeration of Equivalent Expressions

- Query optimizers use equivalence rules to **systematically** generate expressions equivalent to the given expression
- Can generate all equivalent expressions as follows:
 - Repeat
 - apply all applicable equivalence rules on every subexpression of every equivalent expression found so far
 - add newly generated expressions to the set of equivalent expressions
 - Until no new equivalent expressions are generated above
- The above approach is very expensive in space and time
 - Two approaches
 - Optimized plan generation based on transformation rules
 - Special case approach for queries with only selections, projections and joins

Implementing Transformation Based Optimization

- Space requirements reduced by sharing common sub-expressions:
 - when E1 is generated from E2 by an equivalence rule, usually only the top level of the two are different, subtrees below are the same and can be shared using pointers
 - E.g. when applying join commutativity



- Same sub-expression may get generated multiple times
 - Detect duplicate sub-expressions and share one copy
- Time requirements are reduced by not generating all expressions
 - Dynamic programming
 - We will study only the special case of dynamic programming for join order optimization

Cost Estimation

- Cost of each operator computed as described in Chapter 12
 - Need statistics of input relations
 - E.g. number of tuples, sizes of tuples
- Inputs can be results of sub-expressions
 - Need to estimate statistics of expression results
 - To do so, we require additional statistics
 - E.g. number of distinct values for an attribute
- More on cost estimation later

Choice of Evaluation Plans

- Must consider the interaction of evaluation techniques when choosing evaluation plans
 - choosing the cheapest algorithm for each operation independently may not yield best overall algorithm. E.g.
 - 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.

Cost-Based Optimization



- Consider finding the best join-order for $r_1 \ r_2 \ \dots \ r_n$.
- There are $(2(n - 1))!/(n - 1)!$ different join orders for above expression. With $n = 7$, the number is 665280, with $n = 10$, 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 $\{r_1, r_2, \dots, r_n\}$ is computed only once and stored for future use.

Statistics for Cost Estimation

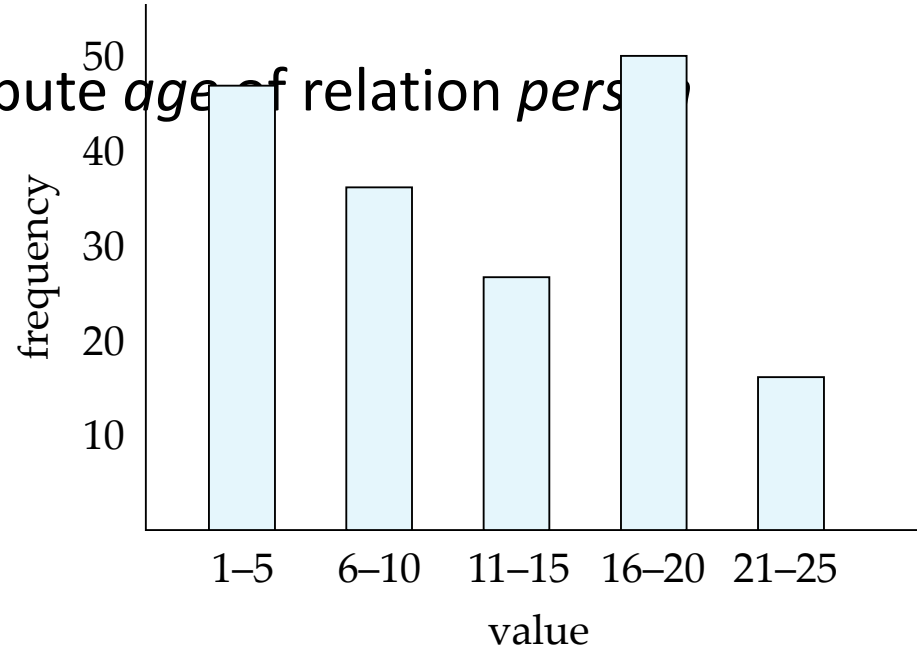
Statistical Information for Cost Estimation

- 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 $\Pi_A(r)$.
- If tuples of r are stored together physically in a file, then:

$$b_r = \frac{n_r}{f_r}$$

Histograms

- Histogram on attribute *age of relation persons*



- **Equi-width** histograms
- **Equi-depth** histograms

Selection Size Estimation

- $\sigma_{A=v}(r)$
 - $n_r / V(A,r)$: 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 $v < \min(A,r)$
 - $c =$
$$n_r \cdot \frac{v - \min(A,r)}{\max(A,r) - \min(A,r)}$$
 - If histograms available, can refine above estimate
 - In absence of statistical information 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:** $\sigma_{\theta_1 \wedge \theta_2 \wedge \dots \wedge \theta_n}(r)$. Assuming independence, estimate of

tuples in the result is:

$$n_r \cdot \frac{s_1}{n_r} \cdot \frac{s_2}{n_r} \cdot \dots \cdot \frac{s_n}{n_r}$$

- Disjunction:** $\sigma_{\theta_1 \vee \theta_2 \vee \dots \vee \theta_n}(r)$. Estimated number of tuples:

$$n_r \cdot \left(1 - \left(1 - \frac{s_1}{n_r}\right) \cdot \left(1 - \frac{s_2}{n_r}\right) \cdot \dots \cdot \left(1 - \frac{s_n}{n_r}\right) \right)$$

Estimated number of tuples:

$$n_r - \text{size}(\sigma_{\theta}(r))$$

Join Operation: Running Example

Running example:

$student \bowtie takes$

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 $b_{takes} = 10000/25 = 400$.
- $V(ID, takes) = 2500$, 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 $student$.
 - $V(ID, student) = 5000$ (primary key!)