Chapter 13: Query Optimization

Database System Concepts, 6th Ed.

Chapter 13: Query Optimization

- Introduction
- Transformation of Relational Expressions
- Catalog Information for Cost Estimation
- Statistical Information for Cost Estimation
- Cost-based optimization
- Dynamic Programming for Choosing Evaluation Plans
- Materialized views

Introduction

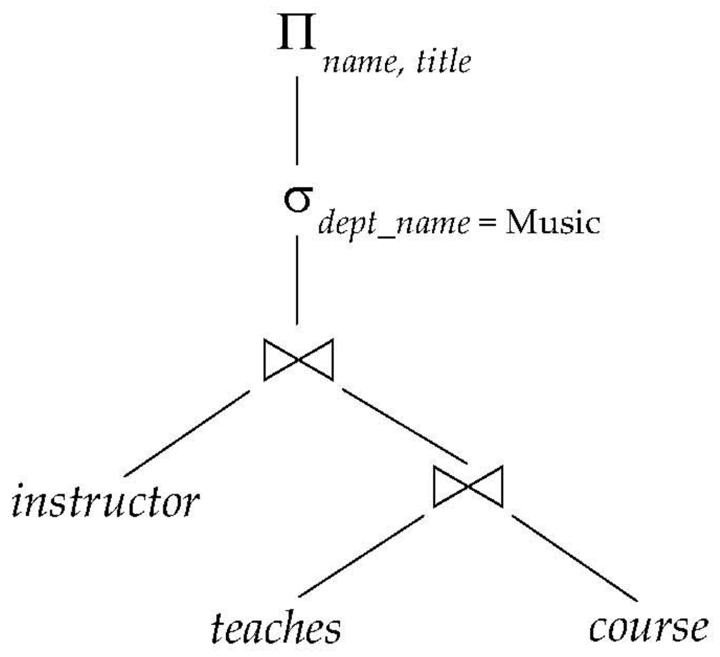
- Query optimization is the process of selecting the most efficient query-evaluation plan from among the many strategies usually possible for processing a given query, specially if the query is complex.
- Two aspects of optimization:

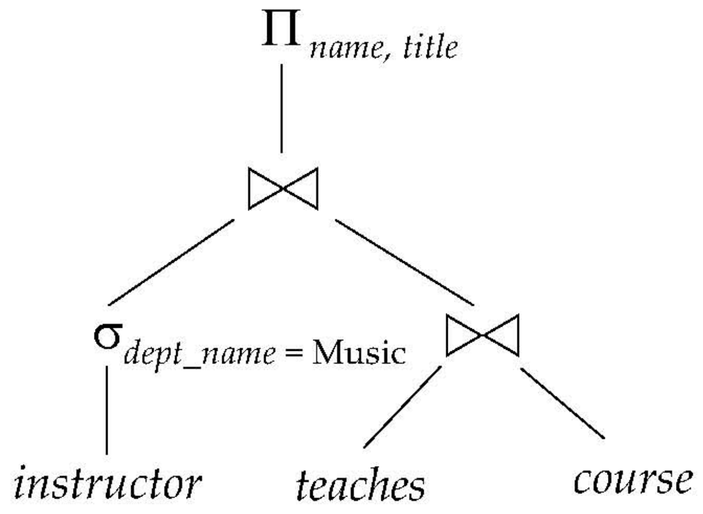
  1. Equivalent expressions: At the relational algebra-level, where the system attempts to find an expression which is equivalent to the given expression, but more efficient to execute.

  2. Different algorithms for each operation: Selecting a detailed strategy for processing the query, such as choosing the algorithm to use for executing an operation, choosing the specific indices to use, and so on.

Introduction (Cont.)

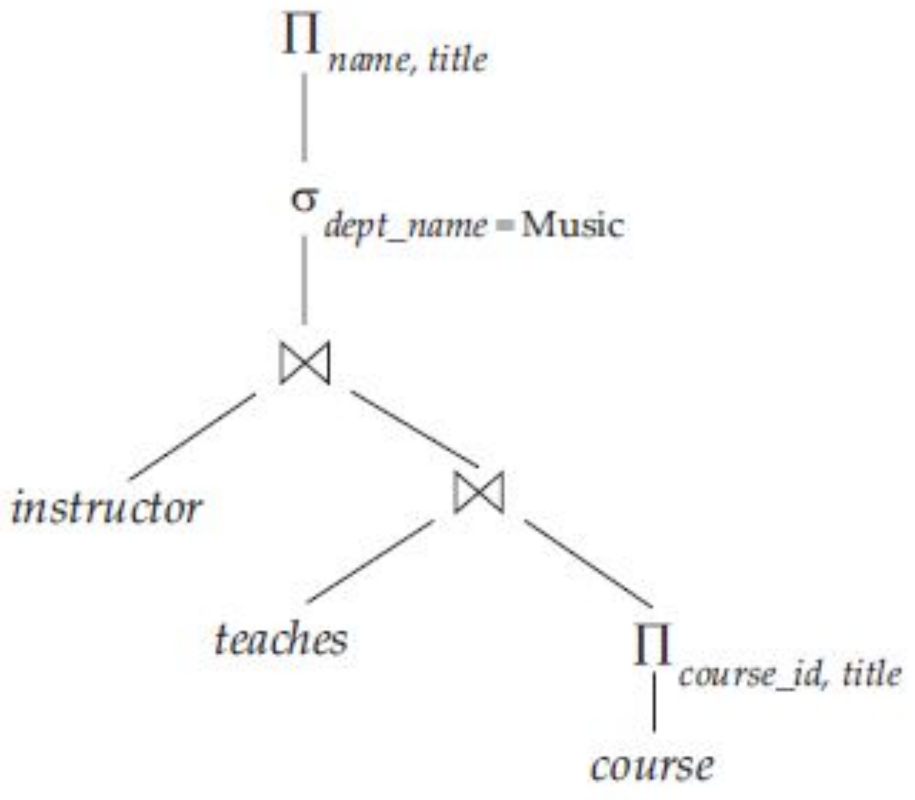
- Relations generated by two equivalent expressions have the same set of attributes and contain the same set of tuples although their tuples/attributes may be ordered differently.
- "Find the names of all instructors in the Music department together with the course title of all the courses that the instructors teach."





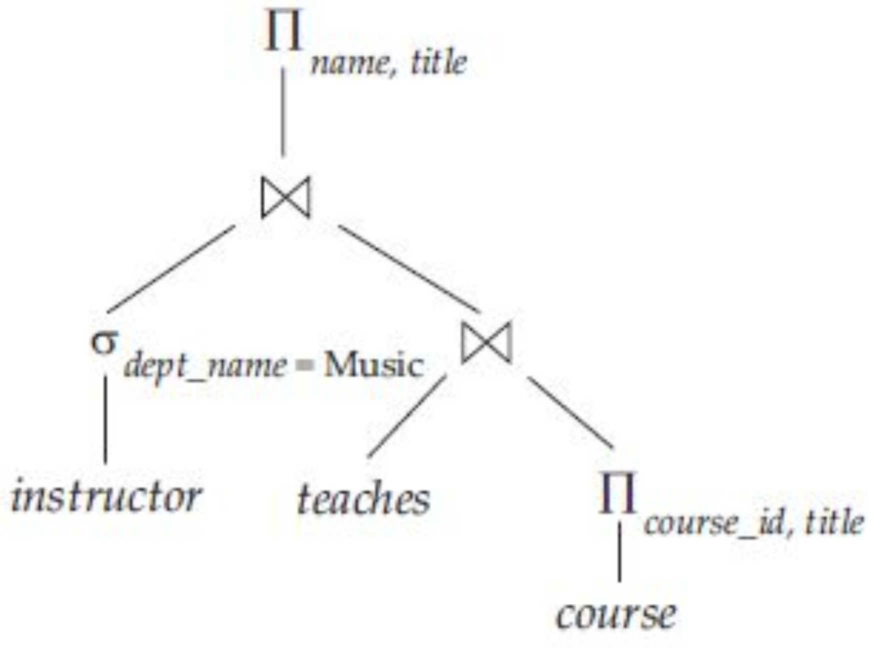
Introduction (Cont.)

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



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

(a) Initial expression tree

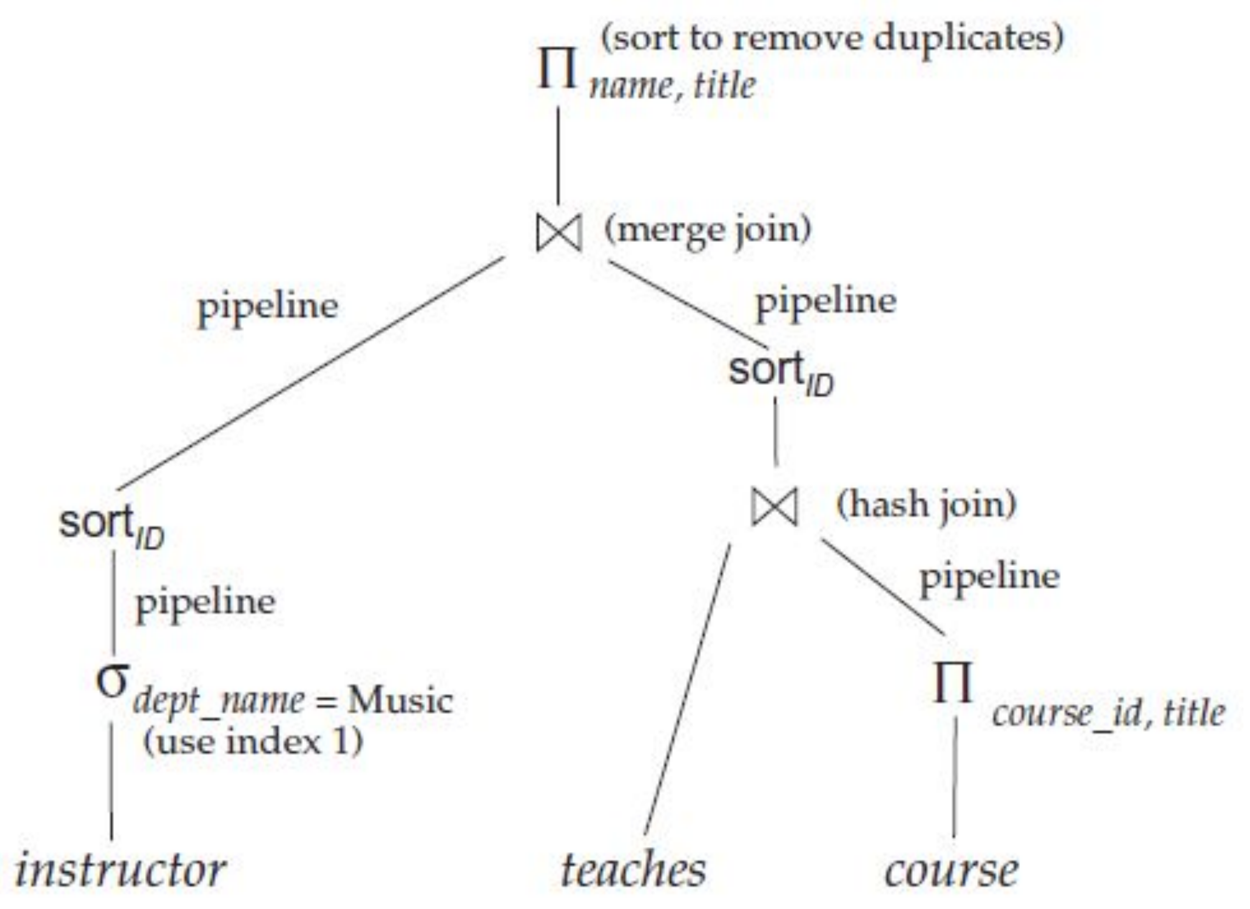


$\Pi_{name,\ title}$
$\bowtie$
$\sigma_{dept\_name = \text{Music}}$
$\bowtie$
instructor
teaches
$\Pi_{course\_id,\ title}$
course

(b) Transformed expression tree

Introduction (Cont.)

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



$\Pi_{name, title}$ (sort to remove duplicates)
$\bowtie$ (merge join)
pipeline
pipeline
$sort_{ID}$
$sort_{ID}$
$\bowtie$ (hash join)
pipeline
pipeline
$\sigma_{dept\_name = \text{Music}}$ (use index 1)
$\Pi_{course\_id, title}$
instructor
teaches
course

Figure 13.2 An evaluation plan.

Introduction (Cont.)

- Cost difference between evaluation plans for a query can be enormous
  - E.g. seconds vs. days in some cases
- Given a relational-algebra expression, it is the job of the query optimizer to come up with a query-evaluation plan that computes the same result as the given expression, and is the least-costly way of generating the result (or, at least, is not much costlier than the least-costly way).
- To find the least-costly query-evaluation plan, the optimizer needs to generate alternative plans that produce the same result as the given expression, and to choose the least-costly one.

Introduction (Cont.)

- Generation of query-evaluation plans involves three steps:
  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**
- The overall process is called **cost-based query optimization**
- 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
- Materialized views also help to speed up processing of certain queries.

Generating Equivalent Expressions (Equivalence Rules)

Database System Concepts, 6th Ed.

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
  - The optimizer uses equivalence rules to transform expressions into other logically equivalent expressions.

Equivalence Rules

1. Conjunctive selection operations can be deconstructed into a sequence of individual selections. (referred to as a cascade of σ)

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

2. Selection operations are commutative.

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

3. Only the final in a sequence of projection operations is needed, the others can be omitted. (referred to as a cascade of Π)

$$\Pi_{L_1}(\Pi_{L_2}(\ldots(\Pi_{Ln}(E))\ldots)) = \Pi_{L_1}(E)$$

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

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

   a. $\sigma_\theta(E_1 \mathrm{X}\, E_2) = E_1 \bowtie_\theta E_2$ (definition of the theta join)

   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:

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

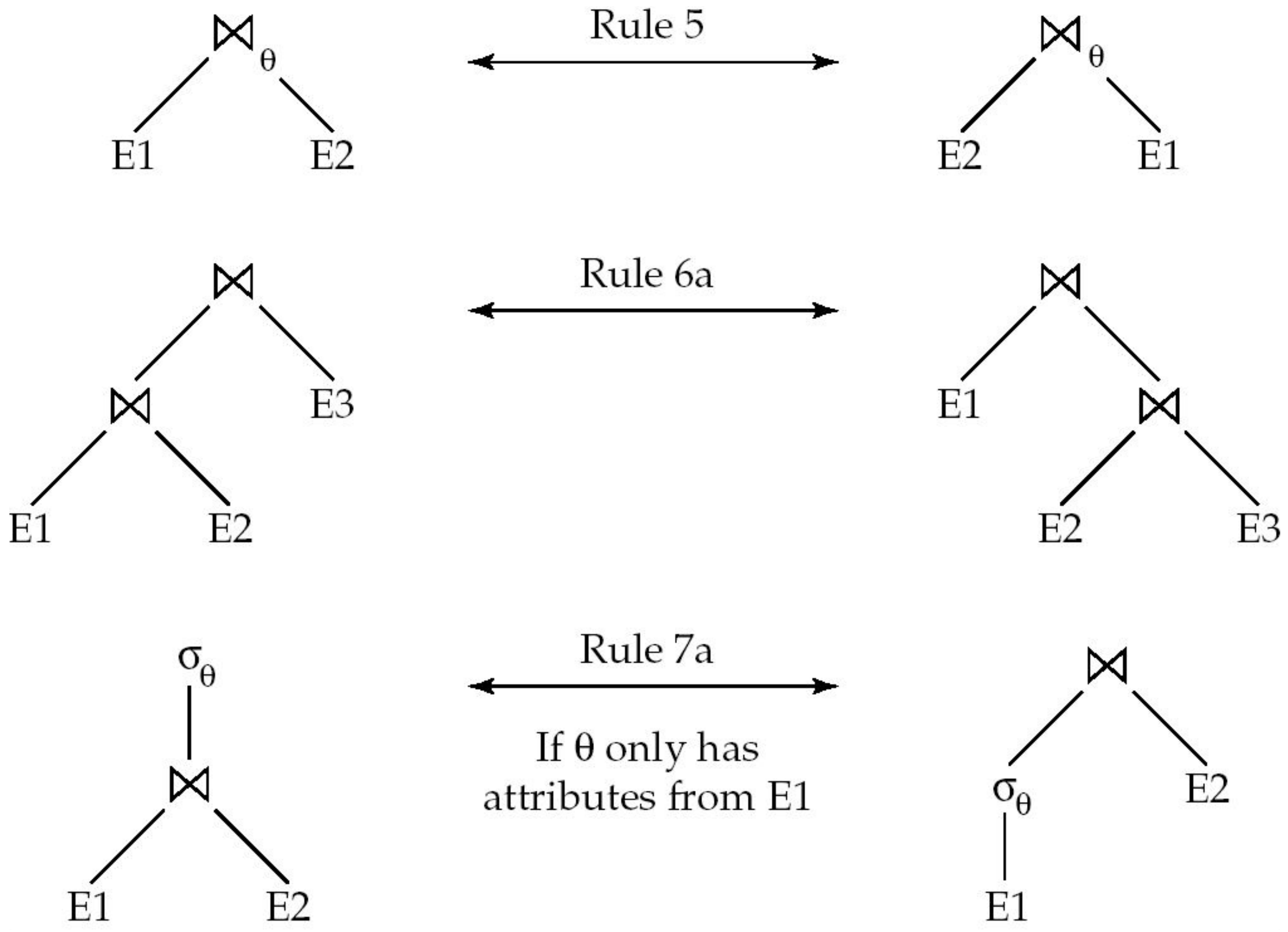
   (b) Theta joins are associative in the following manner:

$$(E_1 \bowtie_{\theta 1} E_2) \bowtie_{\theta 2 \wedge \theta 3} E_3 = E_1 \bowtie_{\theta 1 \wedge \theta 3} (E_2 \bowtie_{\theta 2} E_3)$$

   where θ_2 involves attributes from only E_2 and E_3.

- Any of these conditions may be empty; hence, it follows that the Cartesian product (×) operation is also associative.
- The commutativity and associativity of join operations are important for join reordering in query optimization.

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_1 involve only the attributes of one of the expressions (E_1) being joined.

   $$\sigma_{\theta 1}(E_1 \bowtie_{\theta} E_2) = (\sigma_{\theta 1}(E_1)) \bowtie_{\theta} E_2$$

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

   $$\sigma_{\theta 1} \wedge_{\theta 2} (E_1 \bowtie_{\theta} E_2) = (\sigma_{\theta 1}(E_1)) \bowtie_{\theta} (\sigma_{\theta 2}(E_2))$$

Equivalence Rules (Cont.)

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

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

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

   (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 $L_1 \cup L_2$.

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

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

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

Exercise

- Create equivalence rules involving
  - The group by/aggregation operation
  - Outer joins operation

Equivalence Rules (Cont.)

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

14.a. Full outer-join is commutative:

$$E_1 ⟗ E_2 \equiv E_2 ⟗ E_1$$

b. Left and right outer-join are not commutative

$$E_1 ⟕ E_2 \ {}_{\text{not}}\equiv E_2 ⟕ E_1 , E_1 ⟖ E_2 \ {}_{\text{not}}\equiv E_2 ⟖ E_1$$

, but:

$$E_1 ⟕ E_2 \equiv E_2 ⟖ E_1$$

15. Selection distributes over left and right outer-joins as below, provided θ_1 only involves attributes of one of the expression (say E_1)

a. $\sigma_{\theta 1}(E_1 ⟕_\theta E_2) \equiv (\sigma_{\theta 1}(E_1)) ⟕_\theta E_2$

b. $\sigma_{\theta 1}(E_1 ⟖_\theta E_2) \equiv E_2 ⟕_\theta (\sigma_{\theta 1}(E_1))$

16. Outer joins can be replaced by inner joins under some conditions

a. $\sigma_{\theta 1}(E_1 ⟕_\theta E_2) \equiv \sigma_{\theta 1}(E_1 \bowtie_\theta E_2)$

b. $\sigma_{\theta 1}(E_1 ⟖_\theta E_1) \equiv \sigma_{\theta 1}(E_1 \bowtie_\theta E_2)$

provided θ_1 is null rejecting on E_2

Equivalence Rules (Cont.)

Note that several equivalences that hold for joins do not hold for outer joins

- $\sigma_{\text{year=2017}}(\textit{instructor} ⟕ \textit{teaches}) \not\equiv (\textit{instructor} ⟕ \sigma_{\text{year=2017}}(\textit{teaches}))$
- $\sigma_{\text{year=2017}}(\textit{instructor} ⟕ \textit{teaches}) \equiv \sigma_{\text{year=2017}}(\textit{instructor} \bowtie \textit{teaches})$

- Outer joins are not associative

  $(r ⟕ s) ⟕ t \quad \not\equiv \quad r ⟕ (s ⟕ t)$

  - e.g. with r(A,B) = {(1,1), s(B,C) = { (1,1)}, t(A,C) = { }

Transformation Example: Pushing Selections

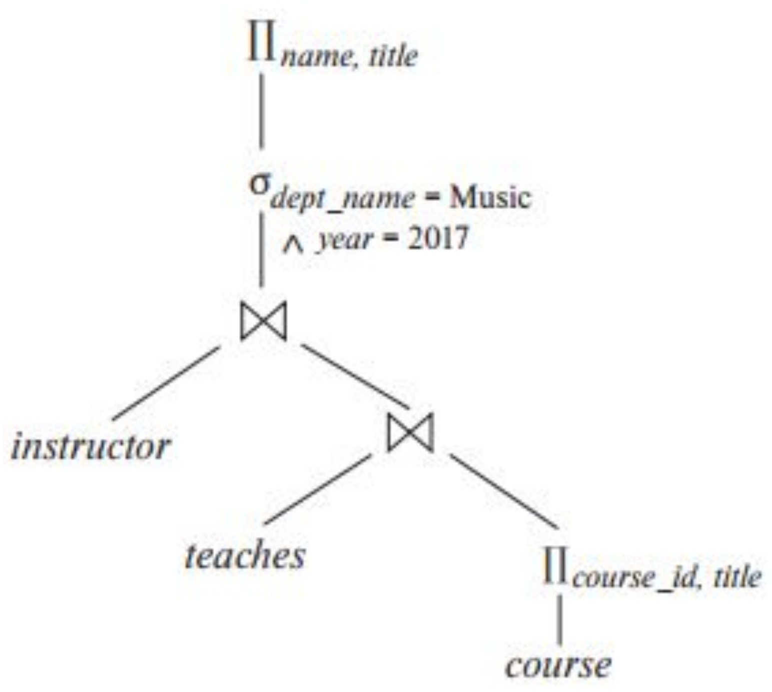
- Query: Find the names of all instructors in the Music department, along with the titles of the courses that they teach
  - $\Pi_{name,\ title}(\sigma_{dept\_name=\text{“Music”}}(instructor \bowtie (teaches \bowtie \Pi_{course\_id,\ title}(course))))$
- Transformation using rule 7a.
  - $\Pi_{name,\ title}((\sigma_{dept\_name=\text{“Music”}}(instructor)) \bowtie (teaches \bowtie \Pi_{course\_id,\ title}(course)))$
- 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

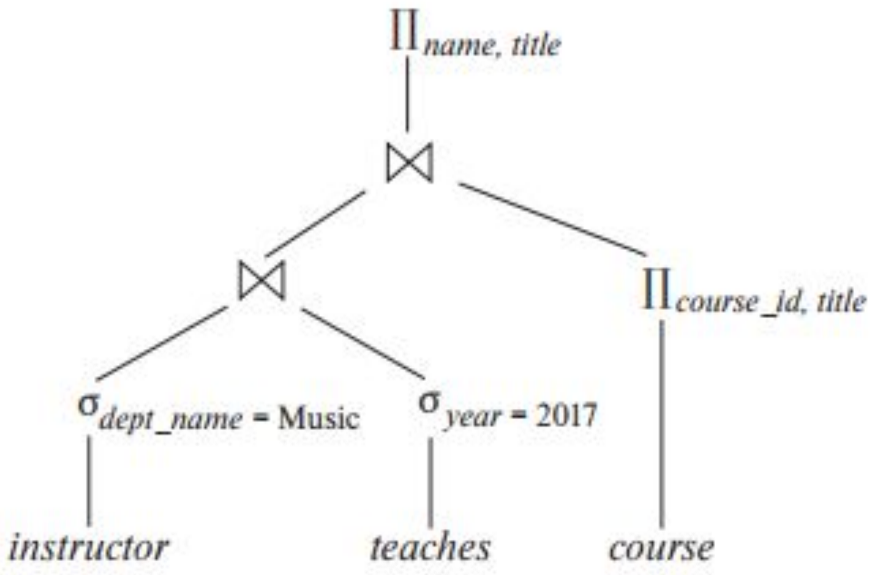
- Query: Find the names of all instructors in the Music department who have taught a course in 2017, along with the titles of the courses that they taught
  - $\Pi_{name,\ title}(\sigma_{dept\_name = \text{“Music”} \wedge year = 2017}(instructor \bowtie (teaches \bowtie \Pi_{course\_id,\ title}(course))))$
- Transformation using join associatively (Rule 6a):
  - $\Pi_{name,\ title}(\sigma_{dept\_name = \text{“Music”} \wedge year = 2017}((instructor \bowtie teaches) \bowtie \Pi_{course\_id,\ title}(course)))$
- Second form provides an opportunity to apply the “perform selections early” rule, resulting in the subexpression

$$\sigma_{dept\_name\ =\ \text{“Music”}}(instructor) \bowtie \sigma_{year\ =\ 2017}(teaches)$$

Multiple Transformations (Cont.)


$\Pi_{name,\ title}$
$\sigma_{dept\_name\ =\ \text{Music} \wedge year\ =\ 2017}$
$\bowtie$
instructor
$\bowtie$
teaches
$\Pi_{course\_id,\ title}$
course

(a) Initial expression tree



(b) Tree after multiple transformations

Figure 16.4 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 rules 8a and 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 ordering of join operations is important for reducing the size of temporary results; hence, most query optimizers pay a lot of attention to the join order.
- As mentioned in equivalence rule 6.a, the natural-join operation is associative.

- 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

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

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

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

    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

Statistics for Cost Estimation

Database System Concepts, 6th Ed.

Estimating Statistics of Expression Results

- The cost of an operation depends on the size and other statistics of its 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
- We first list some statistics about database relations that are stored in database-system catalogs,
- and then show how to use the statistics to estimate statistics on the results of various relational operations.

Catalog Information for Cost Estimation

- 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 *A* is a key for relation *r* , *V(A, r)* is n_r .
- If tuples of *r* are stored together physically in a file, then:

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

- Statistics about indices, such as the heights of B+-tree indices and number of leaf pages in the indices, are also maintained in the catalog.

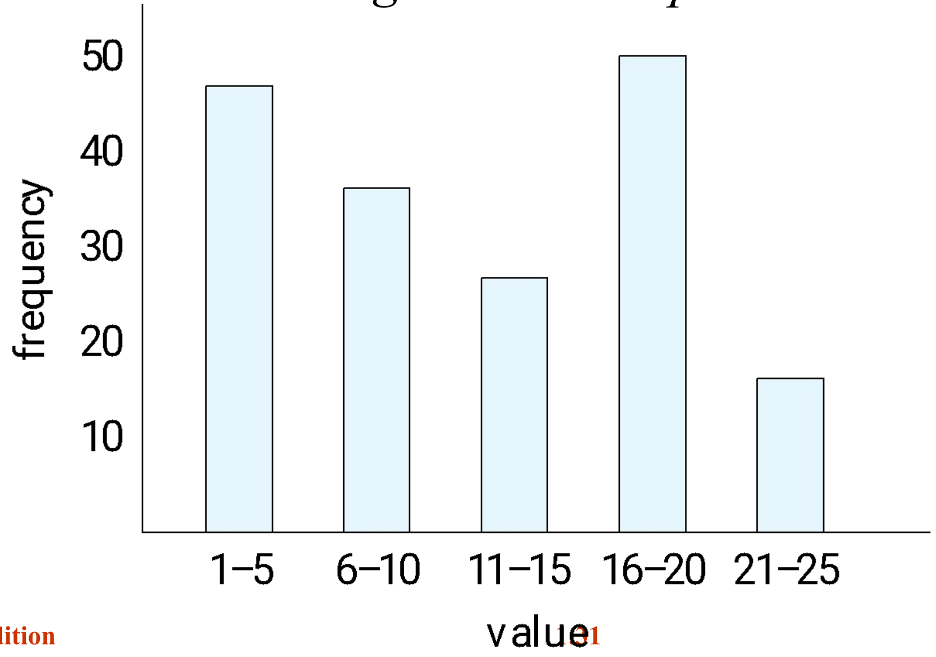
Catalog Information: when to update?

- If we wish to maintain accurate statistics, then every time a relation is modified, we must also update the statistics.
- This update incurs a substantial amount of overhead.

- Therefore, most systems do not update the statistics on every modification. Instead, they update the statistics during periods of light system load.
- As a result, the statistics used for choosing a query-processing strategy may not be completely accurate.
- However, if not too many updates occur in the intervals between the updates of the statistics, the statistics will be sufficiently accurate to provide a good estimation of the relative costs of the different plans.

Histograms

- Real-world optimizers often maintain further statistical information to improve the accuracy of their cost estimates of evaluation plans. Most databases store the distribution of values for each attribute as a **histogram.**
- In a histogram the values for the attribute are divided into a number of ranges, and with each range the histogram associates the number of tuples whose attribute value lies in that range.
- Histogram on attribute *age* of relation *person*



Histograms

- Histograms used in database systems usually record the number of distinct values in each range, in addition to the number of tuples with attribute values in that range.
- Without such histogram information, an optimizer would have to assume that the distribution of values is uniform; that is, each range has the same count.
- A histogram takes up only a little space, so histograms on several different attributes can be stored in the system catalog.
- **Types of histogram :**
- **Equi-width:** divides the range of values into equal-sized ranges
- **Equi-depth:** histogram adjusts the boundaries of the ranges such that each range has the same number of values.

Selection Size Estimation

- **Single equality predicate: $\sigma_{A=v}(r)$**
  - assuming uniform distribution of values, $n_r / V(A,r)$: number of records that will satisfy the selection
  - If histogram is available, frequency count for pertaining range / the number of distinct values that occurs in that range
  - Equality condition on a key attribute: *size estimate* = 1
- **Single comparison predicate: $\sigma_{A \le V}(r)$** (case of $\sigma_{A \geq V}(r)$ is symmetric)
  - Let c denote the estimated number of tuples satisfying the cond.
  - If min(A,r) and max(A,r) are available in catalog
    - c = 0 if v < min(A,r); c = n_r if v >= max(A,r) ; otherwise
    - c =
$$n_r \cdot \frac{v - \min(A,r)}{\max(A,r) - \min(A,r)}$$
  - If histograms are available, can refine above estimate
  - In absence of statistical information *c* is assumed to be $n_r / 2$.

Size Estimation of Complex Selections

- **Combinations of predicates:** 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:
$$n_r * \left(1 - (1 - \frac{s_1}{n_r}) * (1 - \frac{s_2}{n_r}) * \ldots * (1 - \frac{s_n}{n_r})\right)$$
- The probability that the tuple will satisfy the disjunction is 1 minus the probability that it will satisfy *none* of the conditions an then multiply by n_r
- **Negation:** $\sigma_{\neg\theta}(r)$. Estimated number of tuples:
  $n_r - size(\sigma_\theta(r))$

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 only half the students have taken any course (this is unrealistic, but we use it to show that our size estimates are correct even in this case) and 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!*)

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 $R \cap S = \varnothing$, 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 $R \cap S$ forms 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.
  - 4 The case for $R \cap S$ being a foreign key referencing S is symmetric.
- In the example query *student* $\bowtie$ *takes, ID* in *takes* is a foreign key referencing *student*
  - hence, the result has exactly n_{takes} tuples, which is 10000

Estimation of the Size of Joins (Cont.)

- If $R \cap S = \{A\}$ is not a key for R or S and consider each value appears with equal probability
- 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:

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

  These 2 estimates differ if $V(A,s) \neq V(A,r)$. If this situation occurs, there are likely to be dangling tuples that do not participate in the join. Thus, the **lower** of the two estimates is probably the more accurate on

- Can improve on above if histograms are available
  - Use formula similar to above, for each cell of histograms on the two relations

Estimation of the Size of Joins (Cont.)

- Compute the size estimates for $takes \bowtie student$ 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 $\prod_A(r) = V(A,r)$
- Aggregation : estimated size of ${}_A\mathbf{g}_F(r) = V(A,r)$
- Set operations
  - For unions/intersections/set difference of selections on the same relation: rewrite as disjunction or conjunction or negation and use size estimate for selections
    - 4 E.g. $\sigma_{\theta 1}(r) \cup \sigma_{\theta 2}(r)$ can be rewritten as $\sigma_{\theta 1 \vee \theta 2}(r)$
    - 4 $\sigma_{\theta 1}(r) \cap \sigma_{\theta 2}(r)$ can be rewritten as $\sigma_{\theta 1 \wedge \theta 2}(r)$
    - 4 $\sigma_{\theta 1}(r) - \sigma_{\theta 2}(r)$ can be rewritten as $\sigma_{\neg\theta 2}(r)$
  - We can then use the estimates for selections involving conjunctions, disjunctions, and negation

Size Estimation (Cont.)

- For operations on different relations:
  - 4 estimated size of $r \cup s$ = size of r + size of s.
  - 4 estimated size of $r \cap s$ = minimum size of r and size of s.
  - 4 estimated size of $r - s$ = r.
- <u>All the three estimates may be quite inaccurate, but provide upper bounds on the sizes</u>.

● Outer join:

- Estimated size of r ⟕ s = *size of* r ⋈ s + *size of* r
  - 4 Case of right outer join is symmetric
- Estimated size of r ⟗ s = *size of* r ⋈ s + size of r + size of s

Estimation of Number of Distinct Values

- For selections, the number of distinct values of an attribute (or set of attributes) A in the result of a selection, $V(A,\sigma_\theta(r))$, can be estimated in these ways:
- 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 r,* where *op* com operator
  estimated $V(A,\sigma_\theta(r)) = V(A, r) * s$
  - where s is the selectivity of the selection.
- In all the other cases of selection: we assume that the distribution of A values is independent of the distribution of the values on which selection conditions are specified and use approximate estimate of

  $\min(V(A,r), n_{\sigma\theta(r)})$
  - More accurate estimate can be got using probability theory, but this one works fine generally

Estimation of Distinct Values (Cont.)

Joins: $V(A, r \bowtie s)$, the no of distinct values of an attribute (or set of attributes) A

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

  $\min(V(A1,r)*V(A2 - A1,s), V(A1 - A2,r)*V(A2,s), n_{r \bowtie s})$
  - More accurate estimate can be got using probability theory, but this one works fine generally
- Note that estimation of the number of sizes and the number of distinct values of attributes in an intermediate result E_i helps us estimate the sizes and number of distinct values of attributes in the next level intermediate results that use E_i.

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 (sum, count and average), assume all values are distinct, and use *V*(*G,r*)

Evaluation Plan

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

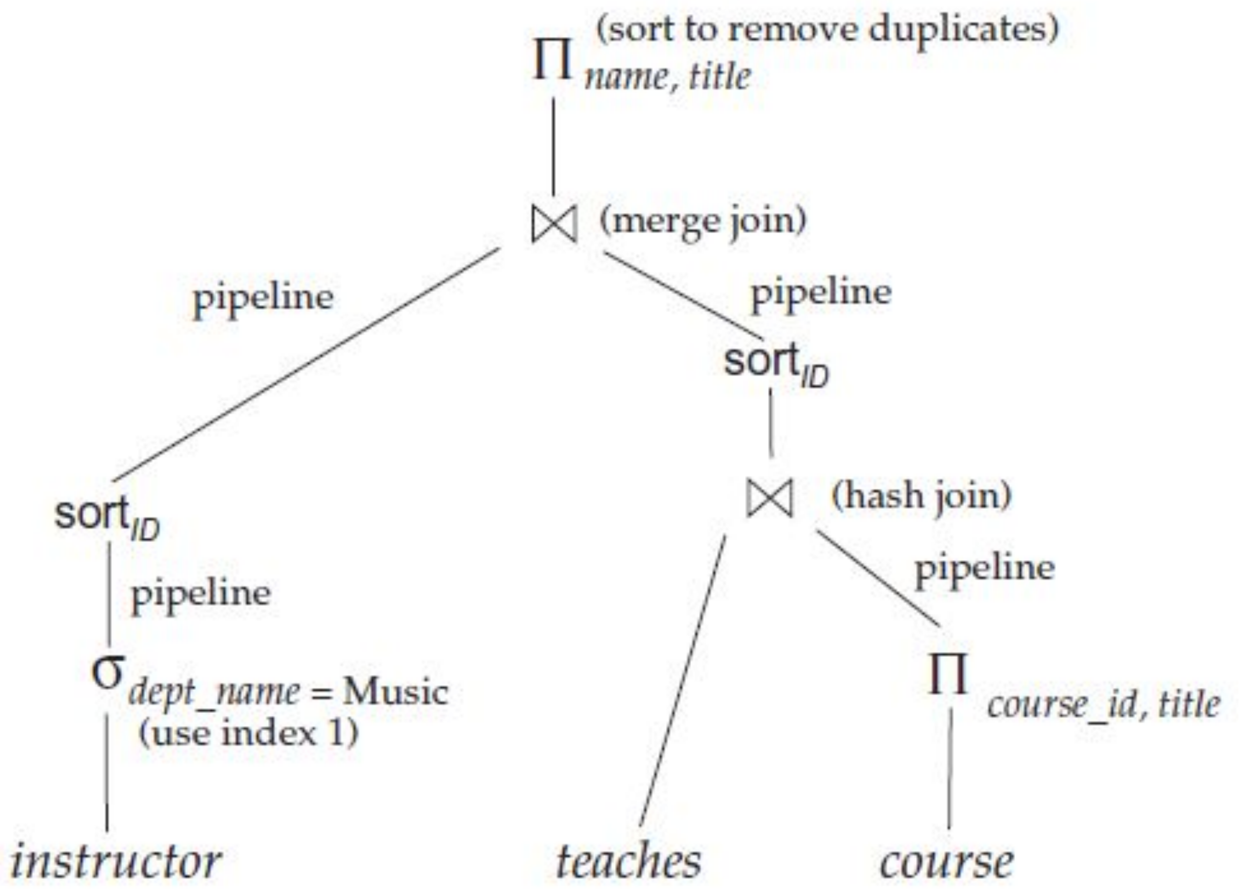


Figure 13.2 An evaluation plan.

Choice of Evaluation Plans

- Generation of expressions is only part of the query-optimization process, since each operation in the expression can be implemented with different algorithms.
- An evaluation plan defines exactly what algorithm should be used for each operation, and how the execution of the operations should be coordinated.
- Given an evaluation plan, we can estimate its cost using statistics estimated by the techniques (just finished) coupled with cost estimates for various algorithms and evaluation methods.
- A **cost-based optimizer** explores the space of all query-evaluation plans that are equivalent to the given query, and chooses the one with the least estimated cost.
- Exploring the space of all possible plans may be too expensive for complex queries. Most optimizers include heuristics to reduce the cost of query optimization, at the potential risk of not finding the optimal plan.

Choice of Evaluation Plans (Cont.)

- 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 the results to the next operation and thus may be useful even it is not the cheapest way of performing the join.
- 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 Join Order Selection

- The most common type of query in SQL consists of a join of a few relations, with join predicates and selections specified in the **where clause.** For a complex join query, the number of different query plans that are equivalent to the query can be large.
- Consider finding the best join-order for $r_1 \bowtie r_2 \bowtie \ldots \bowtie r_n$ where the joins are expressed without any ordering.
- With n = 3, there are 12 different join ordering.

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

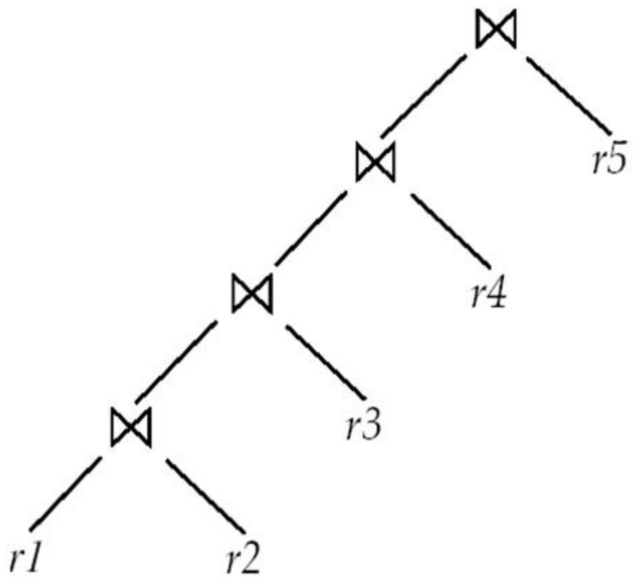
- In general, with n relations, there are $(2(n-1))!/(n-1)!$ different join orders. For joins involving small number of relations, this number is acceptable, however, as n increases, this number rises quickly. With $n = 5$, the number is 1,680, with $n = 7$, the number is 665,280, with $n = 10$, the number is greater than 17.6 billion!

Cost-Based Join Order Selection

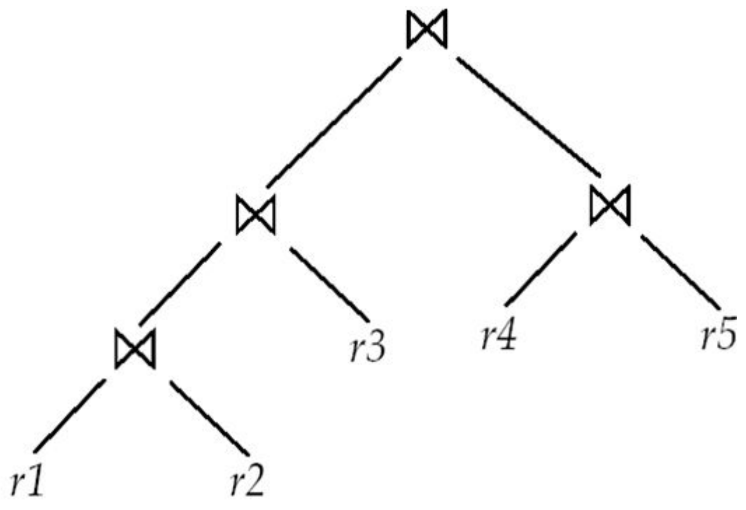
- Luckily, it is not necessary to generate all the expressions equivalent to a given expression. For example, suppose we want to find the best join order of the form: $(r_1 \bowtie r_2 \bowtie r_3) \bowtie r_4 \bowtie r_5$
- There are 12 different join orders for computing $r_1 \bowtie r_2 \bowtie r_3$, 12 orders for computing the join of this result with *r*4 and r5. Thus, there appear to be 144 join orders to examine.
- However, once we have found the best join order for the subset of relations {r1, r2, r3}, we can use that order for further joins with r4 and r5, and can ignore all costlier join orders of $r_1 \bowtie r_2 \bowtie r_3$.
- Thus, instead of 144 choices to examine, we need to examine only 12 + 12 choices.
- Using dynamic programming algorithm, the least-cost (optimal) join order for any subset of $\{r_1, r_2, \ldots r_n\}$ is computed only once and stored for future use; a procedure that can reduce execution time greatly.

Left Deep Join Trees

- In **left-deep join trees,** the right-hand-side input for each join is a relation, not the result of an intermediate join.



(a) Left-deep join tree



(b) Non-left-deep join tree

Cost of Optimization

- With dynamic programming time complexity of optimization with bushy trees is $O(3^n)$.
  - With $n = 10$, this number is 59000 instead of 17.6 billion!
- Space complexity is $O(2^n)$
- To find best left-deep join tree for a set of n relations:
  - Consider n alternatives with one relation as right-hand side input and the other relations as left-hand side input.
  - Using (recursively computed and stored) least-cost join order for each alternative on left-hand-side, choose the cheapest of the n alternatives.
- If only left-deep join trees are considered, time complexity of finding best join order is $O(n\ 2^n)$
  - Space complexity remains at $O(2^n)$
- Cost-based optimization is expensive, but worthwhile for queries on large datasets (typical queries have small n, generally < 10)

Interesting Sort Orders

- Actually, the order in which the tuples are generated by the join of a set of relations is also important for finding the best overall join order, since it can affect the cost of further join (merge join).
- Consider the expression $(r_1 \bowtie r_2 \bowtie r_3) \bowtie r_4 \bowtie r_5$
- An **interesting sort order** is a particular sort order of tuples that could be useful for a later operation.
  - Generating the result of $r_1 \bowtie r_2 \bowtie r_3$ sorted on the attributes common with r_4 or r_5 may be useful, but generating it sorted on the attributes common only r_1 and r_2 is not useful.
  - Using merge-join to compute $r_1 \bowtie r_2 \bowtie r_3$ may be costlier, but may provide an output sorted in an interesting order.
  - Sort order may also be useful for order by and for grouping
- Not sufficient to find the best join order for each subset of the set of *n* given relations; must find the best join order for each subset, for each interesting sort order
  - Simple extension of earlier dynamic programming algorithms
  - Usually, number of interesting orders is quite small and doesn't affect time/space complexity significantly

Cost Based Optimization with Equivalence Rules

- **Physical equivalence rules** allow logical query plan to be converted to physical query plan specifying what algorithms are used for each operation.
- Efficient optimizer based on equivalent rules depends on
  - A space efficient representation of expressions which avoids making multiple copies of subexpressions
  - Efficient techniques for detecting duplicate derivations of expressions
  - A form of dynamic programming based on **memoization**, which stores the best plan for a subexpression the first time it is optimized, and reuses in on repeated optimization calls on same subexpression
  - Cost-based pruning techniques that avoid generating all plans
- Pioneered by the Volcano project and implemented in the SQL Server optimizer

Heuristic Optimization

- A drawback of cost-based optimization is the cost of optimization itself. Although the cost of query optimization can be reduced by clever algorithms, the number of different evaluation plans for a query can be very large, and finding the optimal plan from this set requires a lot of computational effort.
- Hence, optimizers use **heuristics** to reduce the cost of optimization.
- Heuristic optimization transforms the relational algebra query-tree by using a set of rules that typically (but not in all cases) improve execution performance:
  - Perform selection as early as possible (reduces the no. of tuples)
  - Perform projection early (reduces the no. of attributes)
  - Perform most restrictive selection and join operations (i.e. with smallest result size) before other similar operations.
- It is usually better to perform selections earlier than projections, since selections have the potential to reduce the sizes of relations greatly, and selections enable the use of indices to access tuples.

Steps in Typical Heuristic Optimization

1. Deconstruct conjunctive selections into a sequence of single selection operations (Equiv. rule 1.).
2. Move selection operations down the query tree for the earliest possible execution (Equiv. rules 2, 7a, 7b, 11).
3. Execute first those selection and join operations that will produce the smallest relations (Equiv. rule 6).
4. Replace Cartesian product operations that are followed by a selection condition by join operations (Equiv. rule 4a).
5. Deconstruct and move as far down the tree as possible lists of projection attributes, creating new projections where needed (Equiv. rules 3, 8a, 8b, 12).
6. Identify those subtrees whose operations can be pipelined, and execute them using pipelining.

Structure of Query Optimizers

- Some systems use only heuristics, others combine heuristics with partial cost-based optimization.
- Many optimizers considers only left-deep join orders.
  - Plus heuristics to push selections and projections down the query tree
  - Reduces optimization complexity and generates plans amenable to pipelined evaluation.
- Heuristic optimization used in some versions of Oracle:
  - Repeatedly pick "best" relation to join next
    - Starting from each of n starting points. Pick best among these

Additional Optimization Techniques

- Materialized Views
- Nested Subqueries

Materialized Views**

- When a view is defined, normally the database stores only the query defining the view.
- A **materialized view** is a view whose contents are computed and stored.
- Consider the view
  create view *department_total_salary*(*dept_name, total_salary*) **as**
  select *dept_name*, **sum**(*salary*)
  from *instructor*
  group by *dept_name*
- Materializing the above view would be very useful if the total salary by department is required frequently
  - Saves the effort of finding multiple tuples and adding up their amounts

Materialized View Maintenance

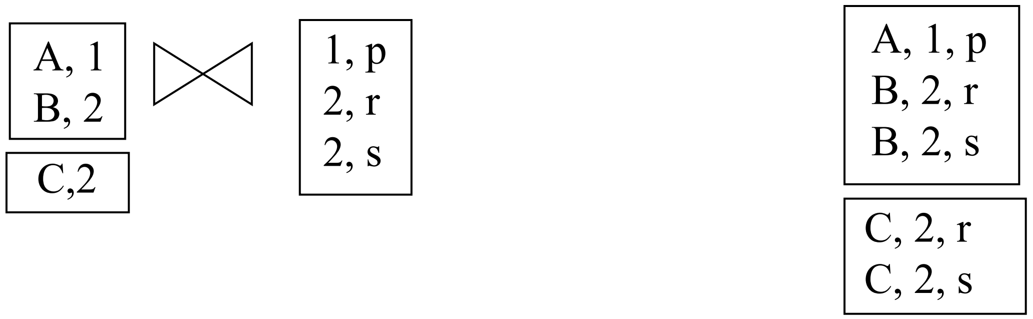
- The task of keeping a materialized view up-to-date with the underlying data is known as **materialized view maintenance**
- Materialized views can be maintained by recomputation on every update
- A better option is to use **incremental view maintenance**
  - **Changes to database relations are used to compute changes to the materialized view, which is then updated**
- View maintenance can be done by
  - Manually defining triggers on insert, delete, and update of each relation in the view definition
  - Manually written code to update the view whenever database relations are updated
  - Periodic recomputation (e.g. nightly)
  - Above methods are directly supported by many database systems
    - Avoids manual effort/correctness issues

Incremental View Maintenance

- The changes (inserts and deletes) to a relation or expressions are referred to as its **differential**
  - Set of tuples inserted to and deleted from r are denoted $\mathbf{i_r}$ and $\mathbf{d_r}$
- To simplify our description, we only consider inserts and deletes
  - We replace updates to a tuple by deletion of the tuple followed by insertion of the update tuple
- We describe how to compute the change to the result of each relational operation, given changes to its inputs
- We then outline how to handle relational algebra expressions

Join Operation

- Consider the materialized view $v = r \bowtie s$ and an update to r
- Let r^{old} and r^{new} denote the old and new states of relation r
- Consider the case of an insert to r:
  - We can write $r^{new} \bowtie s$ as $(r^{old} \cup i_r) \bowtie s$
  - And rewrite the above to $(r^{\text{old}} \bowtie s) \cup (i_r \bowtie s)$
  - But $(r^{\text{old}} \bowtie s)$ is simply the old value of the materialized view, so the incremental change to the view is just $i_r \bowtie s$
- Thus, for inserts $v^{new} = v^{old} \cup (i_r \bowtie s)$
- Similarly for deletes $v^{new} = v^{old} - (d_r \bowtie s)$



Selection and Projection Operations

- Selection: Consider a view $v = \sigma_\theta(r)$.
  - $v^{new} = v^{old} \cup \sigma_\theta(i_r)$
  - $v^{new} = v^{old} - \sigma_\theta(d_r)$
- Projection is a more difficult operation
  - $R = (A,B)$, and r(R) = { $(a,2)$, $(a,3)$}
  - $\prod_A(r)$ has a single tuple (a).
  - If we delete the tuple $(a,2)$ from r, we should not delete the tuple (a) from $\prod_A(r)$, but if we then delete $(a,3)$ as well, we should delete the tuple
- For each tuple in a projection $\prod_A(r)$, we will keep a count of how many times it was derived
  - On insert of a tuple to r, if the resultant tuple is already in $\prod_A(r)$ we increment its count, else we add a new tuple with count = 1
  - On delete of a tuple from r, we decrement the count of the corresponding tuple in $\prod_A(r)$
    - if the count becomes 0, we delete the tuple from $\prod_A(r)$

Aggregation Operations

- count : $v = {}_A\mathbf{g}_{count(B)}{}^{(r)}$.
  - When a set of tuples i_r is inserted
    - For each tuple r in i_r, if the corresponding group is already present in v, we increment its count, else we add a new tuple with count = 1
  - When a set of tuples d_r is deleted
    - for each tuple t in $i_{r.}$ we look for the group *t.A* in *v*, and subtract 1 from the count for the group.
      - If the count becomes 0, we delete from *v* the tuple for the group *t.A*

Aggregate Operations (Cont.)

- sum: $v = {}_A\mathbf{g}_{sum\ (B)}^{\ (r)}$
  - We maintain the sum in a manner similar to count, except we add/subtract the B value instead of adding/subtracting 1 for the count
  - Additionally we maintain the count in order to detect groups with no tuples. Such groups are deleted from v
    - Cannot simply test for sum = 0 (why?)
- To handle the case of **avg**, we maintain the **sum** and **count** aggregate values separately, and divide at the end
- **min**, **max**: $v = {}_A\mathbf{g}_{min\ (B)}\,(r)$.
  - Handling insertions on r is straightforward.
  - Maintaining the aggregate values **min** and **max** on deletions may be more expensive. We have to look at the other tuples of *r* that are in the same group to find the new minimum

Other Operations

- Set intersection: $v = r \cap s$
  - when a tuple is inserted in *r* we check if it is present in *s*, and if so we add it to *v*.
  - If the tuple is deleted from r, we delete it from the intersection if it is present.
  - Updates to *s* are symmetric
  - The other set operations, *union* and *set difference* are handled in a similar fashion.
- Outer joins are handled in much the same way as joins but with some extra work
  - we leave details to you.

Handling Expressions

- To handle an entire expression, we derive expressions for computing the incremental change to the result of each sub-expressions, starting from the smallest sub-expressions.
- E.g. consider $E_1 \bowtie E_2$ where each of E_1 and E_2 may be a complex expression
  - Suppose the set of tuples to be inserted into E_1 is given by D_1
    - Computed earlier, since smaller sub-expressions are handled first
  - Then the set of tuples to be inserted into $E_1 \bowtie E_2$ is given by $D_1 \bowtie E_2$
    - This is just the usual way of maintaining joins

Query Optimization and Materialized Views

- Query optimization can be performed by treating materialized views just like regular relations.
- Rewriting queries to use materialized views:
  - A materialized view $v = r \bowtie s$ is available
  - A user submits a query $r \bowtie s \bowtie t$
  - We can rewrite the query as $v \bowtie t$
    - Whether to do so depends on cost estimates for the two alternative
- Replacing a use of a materialized view by the view definition:
  - A materialized view $v = r \bowtie s$ is available, but without any index on it
  - User submits a query $\sigma_{A=10}(v)$.
  - Suppose also that *s* has an index on the common attribute B, and r has an index on attribute A.
  - The best plan for this query may be to replace *v* by $r \bowtie s$, which can lead to the query plan $\sigma_{A=10}(r) \bowtie s$
- Query optimizer should be extended to consider all above alternatives and choose the best overall plan

Materialized View Selection

- **Materialized view selection**: "What is the best set of views to materialize?".
- **Index selection:** "what is the best set of indices to create"
  - closely related, to materialized view selection
    - 4 but simpler
- Materialized view selection and index selection based on typical system **workload** (queries and updates)
  - Typical goal: minimize time to execute workload , subject to constraints on space and time taken for some critical queries/updates
  - One of the steps in database tuning
    - 4 more on tuning in later chapters
- Commercial database systems provide tools (called "tuning assistants" or "wizards") to help the database administrator choose what indices and materialized views to create

Optimizing Nested Subqueries**

- Nested query example:
  select *name*
  from *instructor*
  where exists (**select** *
      from *teaches*
      where *instructor.ID* = *teaches.ID* **and** *teaches.year* = *2007*)
- SQL conceptually treats nested subqueries in the where clause as functions that take parameters and return a single value or set of values
  - Parameters are variables from outer level query that are used in the nested subquery; such variables are called **correlation variables**
- Conceptually, nested subquery is executed once for each tuple in the cross-product generated by the outer level **from** clause
  - Such evaluation is called **correlated evaluation**
  - Note: other conditions in where clause may be used to compute a join (instead of a cross-product) before executing the nested subquery

Optimizing Nested Subqueries (Cont.)

- Correlated evaluation may be quite inefficient since
  - a large number of calls may be made to the nested query
  - there may be unnecessary random I/O as a result
- SQL optimizers attempt to transform nested subqueries to joins where possible, enabling use of efficient join techniques
- E.g.: earlier nested query can be rewritten as
  select *name*
  from *instructor, teaches*
  where *instructor.ID = teaches.ID* **and** *teaches.year = 2007*
  - Note: the two queries generate different numbers of duplicates (why?)
    - teaches can have duplicate IDs
    - Can be modified to handle duplicates correctly as we will see
- In general, it is not possible/straightforward to move the entire nested subquery from clause into the outer level query from clause
  - A temporary relation is created instead, and used in body of outer level query

Optimizing Nested Subqueries (Cont.)

In general, SQL queries of the form below can be rewritten as shown

- Rewrite: **select** …
  from L_1
  where P_1 **and exists** (**select** *
  from L_2
  where P_2)
- To: **create table** t_1 as
  select distinct V
  from L_2
  where P_2^1

  select …
  from L_1, t_1
  where P_1 **and** P_2^2

  - P_2^1 contains predicates in P_2 that do not involve any correlation variables
  - P_2^2 reintroduces predicates involving correlation variables, with relations renamed appropriately
  - V contains all attributes used in predicates with correlation variables

Optimizing Nested Subqueries (Cont.)

- In our example, the original nested query would be transformed to

```
create table t1 as
     select distinct ID
     from teaches
     where year = 2007

select name
from instructor, t1
 where t1.ID = instructor.ID
```

- The process of replacing a nested query by a query with a join (possibly with a temporary relation) is called **decorrelation**.
- Decorrelation is more complicated when
  - the nested subquery uses aggregation, or
  - when the result of the nested subquery is used to test for equality, or
  - when the condition linking the nested subquery to the other query is **not exists**,
  - and so on.

Additional Optimization Techniques

Database System Concepts, 6th Ed.

Top-K Queries

- **Top-K queries**

```
select *
from r, s
where r.B = s.B
order by r.A ascending
limit 10
```

  - Alternative 1: Indexed nested loops join with r as outer
  - Alternative 2: estimate highest r.A value in result and add selection (**and** r.A <= H) to where clause
    - If < 10 results, retry with larger H

Optimization of Updates

- **Halloween problem**

  update R **set** A = 5 * A
  where A > 10

  - If index on A is used to find tuples satisfying A > 10, and tuples updated immediately, same tuple may be found (and updated) multiple times
  - Solution 1: *Always defer updates*
    - collect the updates (old and new values of tuples) and update relation and indices in second pass
    - Drawback: extra overhead even if e.g. update is only on R.B, not on attributes in selection condition
  - Solution 2: *Defer only if required*
    - Perform immediate update if update does not affect attributes in where clause, and deferred updates otherwise.

Join Minimization

- **Join minimization**

  select r.A, r.B
  from r, s
  where r.B = s.B

- Check if join with s is redundant, drop it
  - E.g. join condition is on foreign key from r to s, r.B is declared as not null, and no selection on s
  - Other sufficient conditions possible

    select r.A, s2.B
    from r, s **as** s1, s **as** s2
    where r.B=s1.B **and** r.B = s2.B **and** s1.A < 20 **and** s2.A < 10

    - join with s1 is redundant and can be dropped (along with selection on s1)
  - Lots of research in this area since 70s/80s!

Multiquery Optimization

- Example

  Q1: **select** * **from** (r **natural join** t) **natural join** s

  Q2: **select** * **from** (r **natural join** u) **natural join** s

  - Both queries share common subexpression (r natural join s)
  - May be useful to compute (r natural join s) once and use it in both queries
    - But this may be more expensive in some situations
      - e.g. (r natural join s) may be expensive, plans as shown in queries may be cheaper
- **Multiquery optimization**: find best overall plan for a set of queries, expoiting sharing of common subexpressions between queries where it is useful

Multiquery Optimization (Cont.)

- Simple heuristic used in some database systems:
  - optimize each query separately
  - detect and exploiting common subexpressions in the individual optimal query plans
    - May not always give best plan, but is cheap to implement
  - **Shared scans**: widely used special case of multiquery optimization
- Set of materialized views may share common subexpressions
  - As a result, view maintenance plans may share subexpressions
  - Multiquery optimization can be useful in such situations

Parametric Query Optimization

- Example
  select *
  from r **natural join** s
  where r.a < $1
  - value of parameter $1 not known at compile time
    - known only at run time
  - different plans may be optimal for different values of $1
- Solution 1: optimize at run time, each time query is submitted
    - can be expensive
- Solution 2: **Parametric Query Optimization**:
  - optimizer generates a set of plans, optimal for different values of $1
    - Set of optimal plans usually small for 1 to 3 parameters
    - Key issue: how to do find set of optimal plans efficiently
  - best one from this set is chosen at run time when $1 is known
- Solution 3: **Query Plan Caching**
  - If optimizer decides that same plan is likely to be optimal for all parameter values, it caches plan and reuses it, else reoptimize each time
  - Implemented in many database systems

Extra Slides

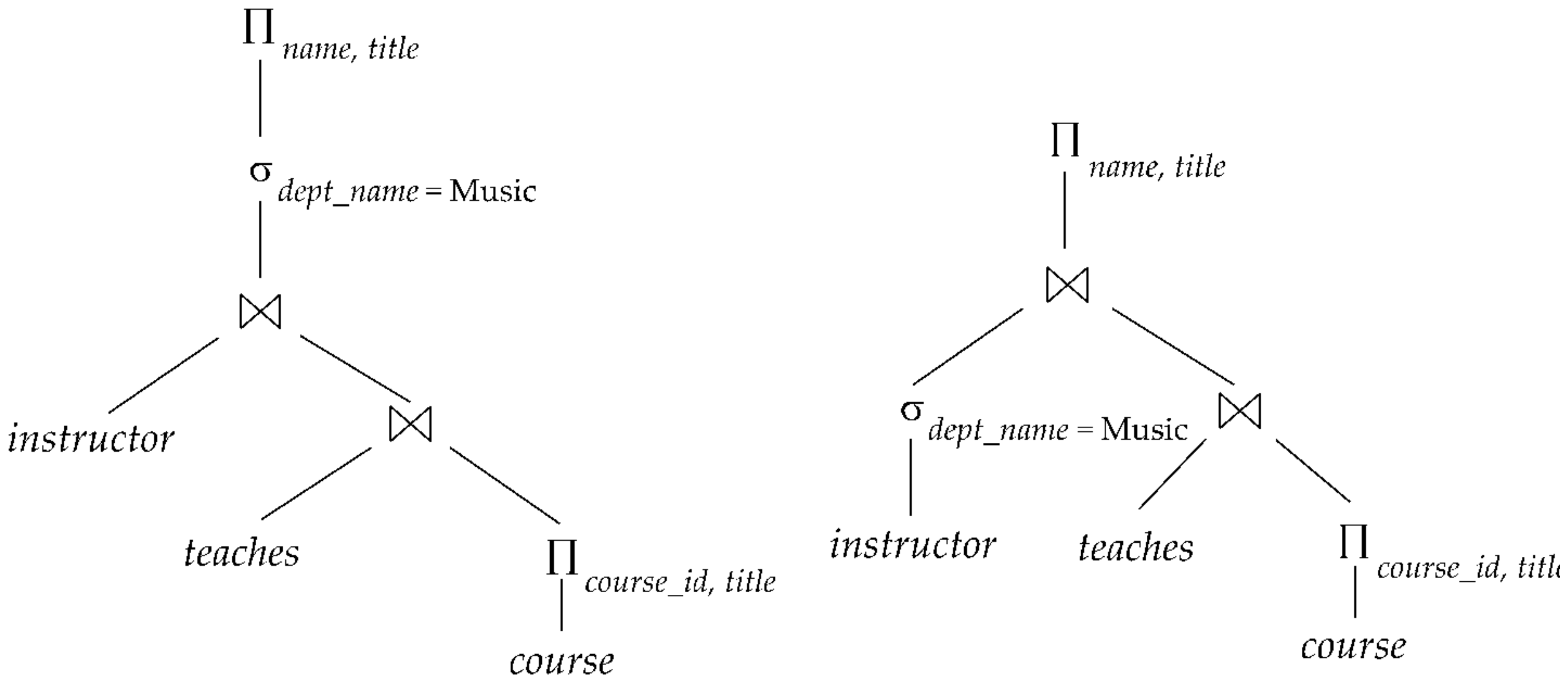
(Not in 6th Edition book)

Plan Stability Across Optimizer Changes

- What if 95% of plans are faster on database/optimizer version N+1 than on N, but 5% are slower?
  - Why should plans be slower on new improved optimizer?
    - Answer: Two wrongs can make a right, fixing one wrong can make things worse!
- Approaches:
  - Allow hints for tuning queries
    - Not practical for migrating large systems with no access to source code
  - Set optimization level, default to version N (Oracle)
    - And migrate one query at a time after testing both plans on new optimizer
  - Save plan from version N, and give it to optimizer version N+1
    - Sybase, XML representation of plans (SQL Server)

End of Chapter

Figure 13.01



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

Figure 13.02

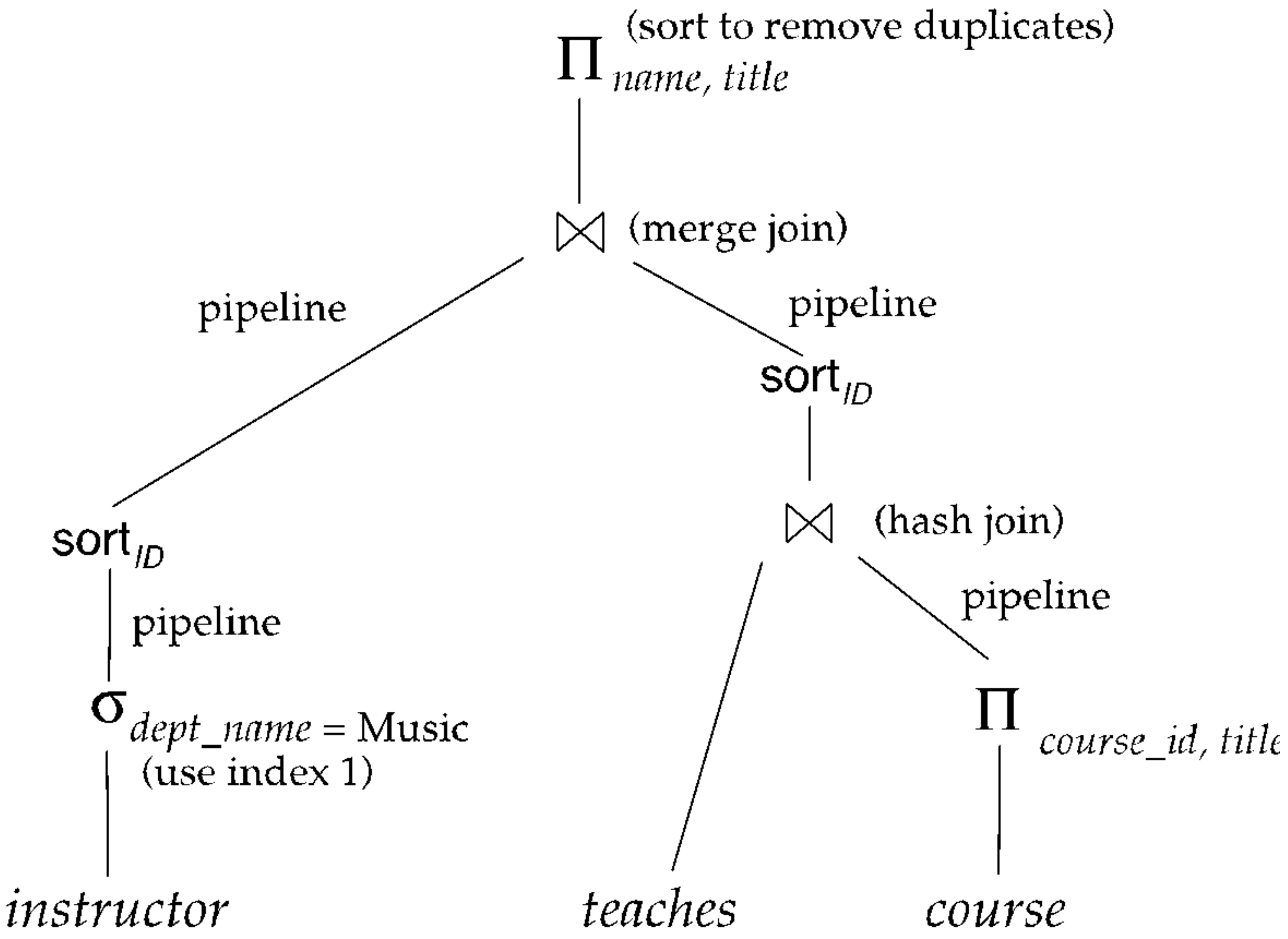
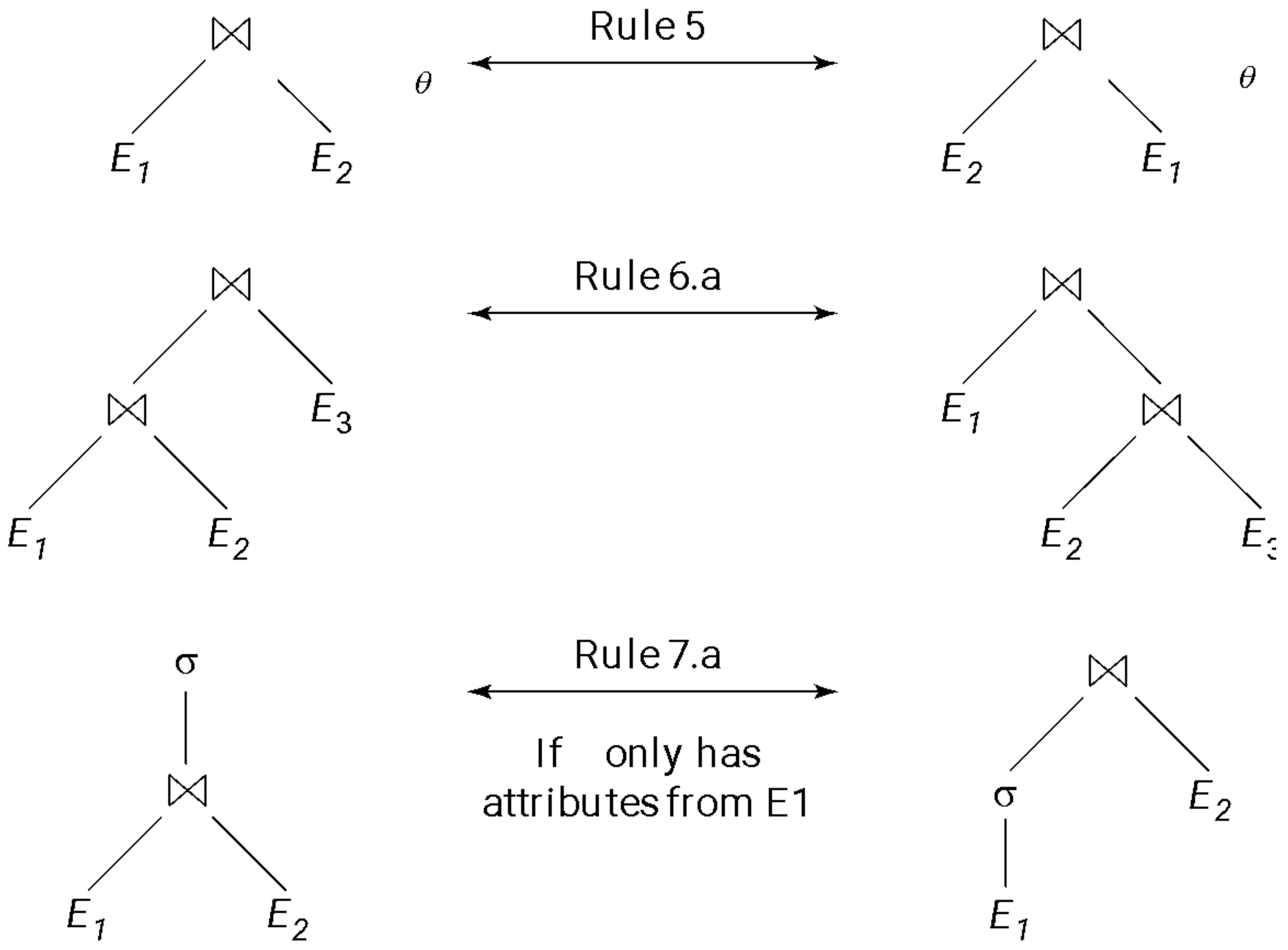
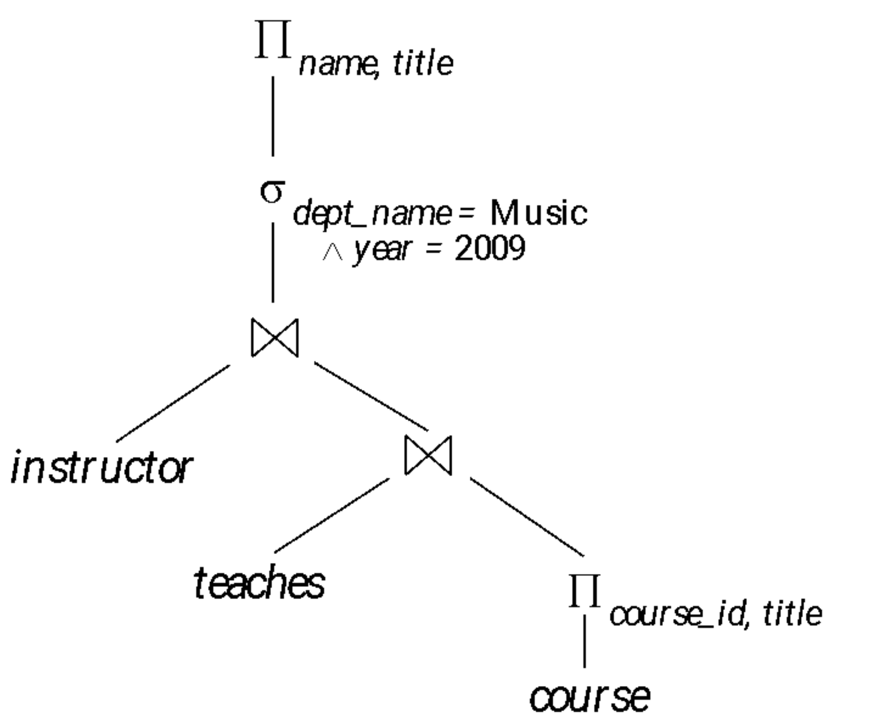

(sort to remove duplicates)
Π name, title
⋈ (merge join)
pipeline
pipeline
sort ID
sort ID
⋈ (hash join)
pipeline
pipeline
σ dept_name = Music
(use index 1)
Π course_id, title
instructor
teaches
course

Figure 13.03



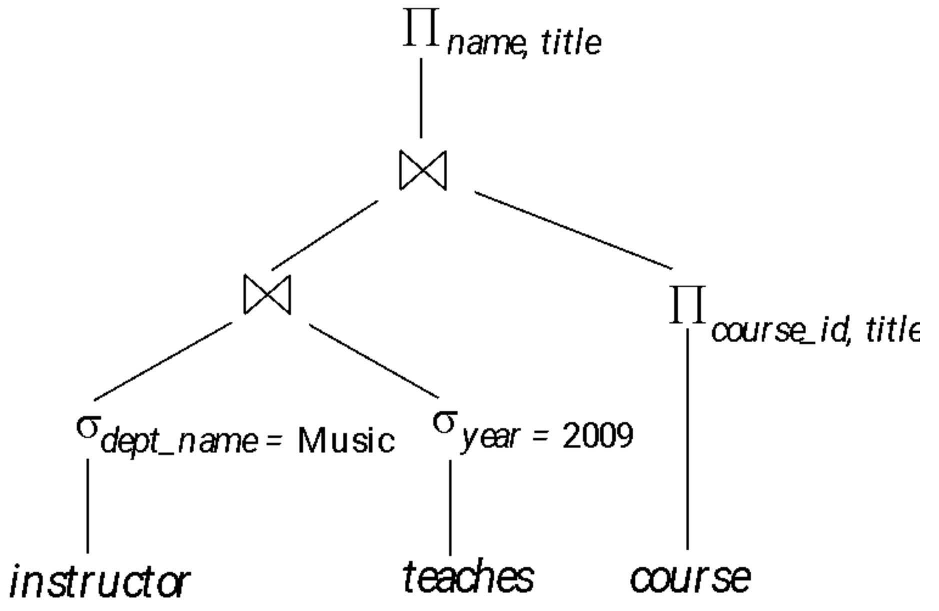
Rule 5
θ
θ
E1
E2
E2
E1
Rule 6.a
E1
E2
E3
E1
E2
E3
σ
Rule 7.a
If only has attributes from E1
σ
E1
E2
E1
E2

Figure 13.04



$\Pi_{name, title}$
$\sigma_{dept\_name = \text{Music} \wedge year = 2009}$
$\bowtie$
instructor
$\bowtie$
teaches
$\Pi_{course\_id, title}$
course

(a) Initial expression tree



(b) Tree after multiple transformations

Figure 13.06

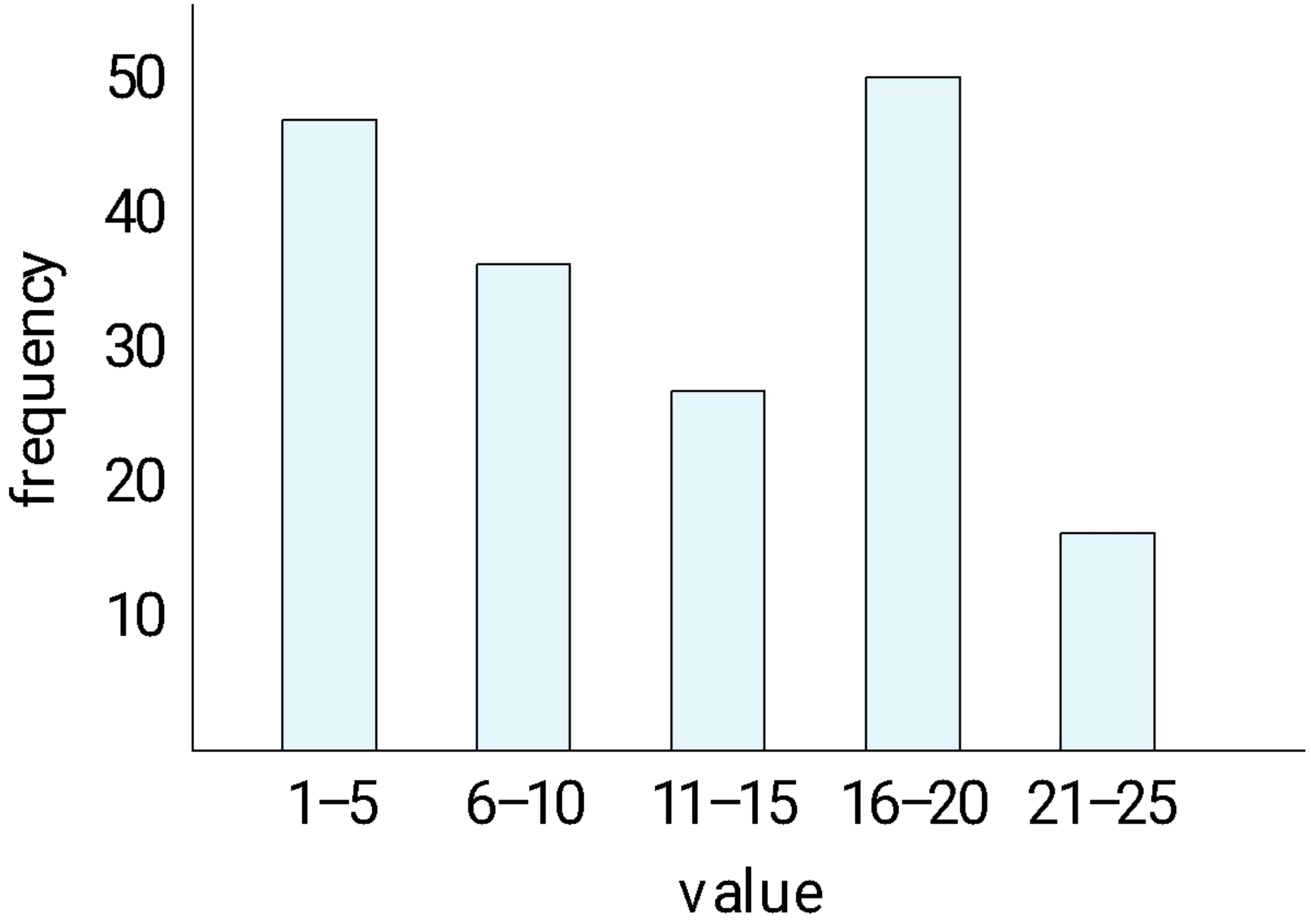
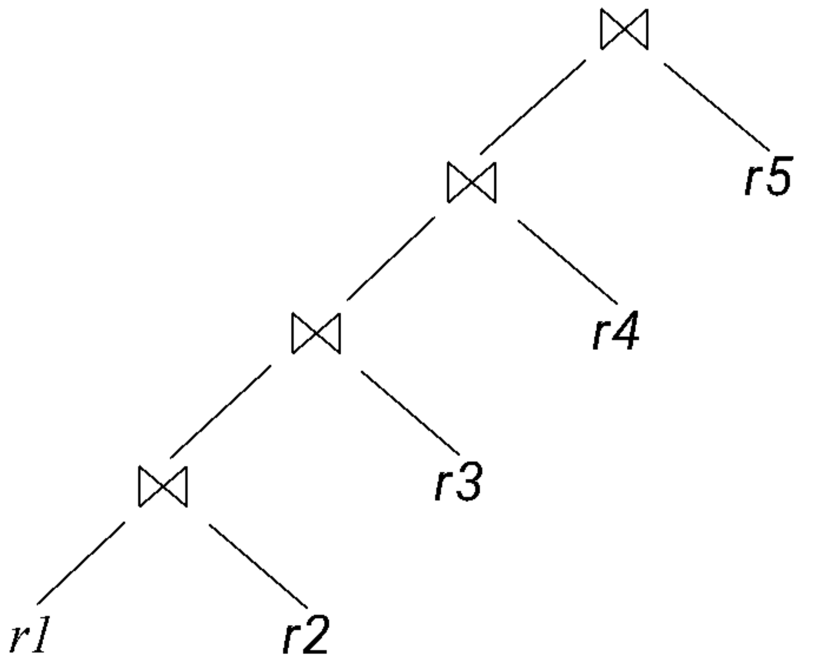
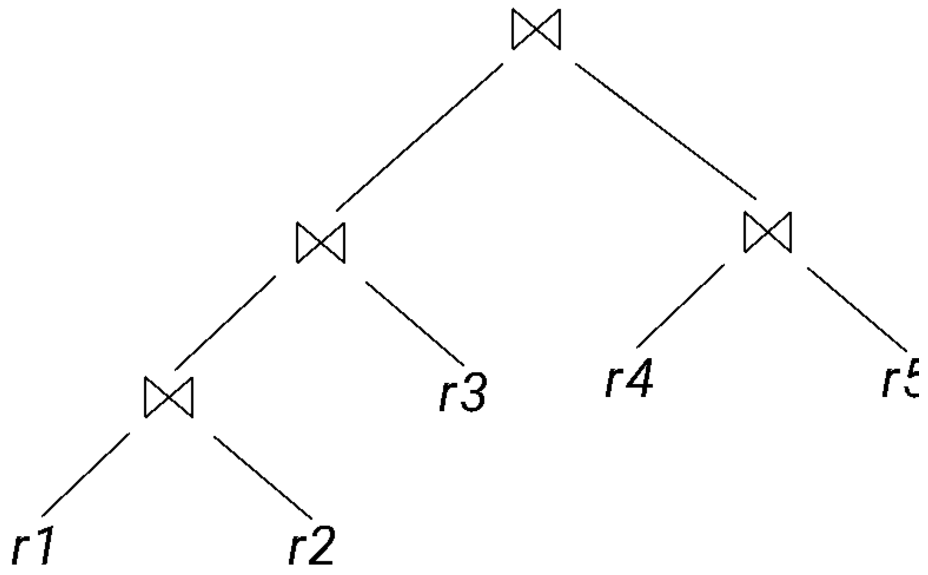


Figure 13.08



(a) Left-deep join tree



(b) Non-left-deep join tree