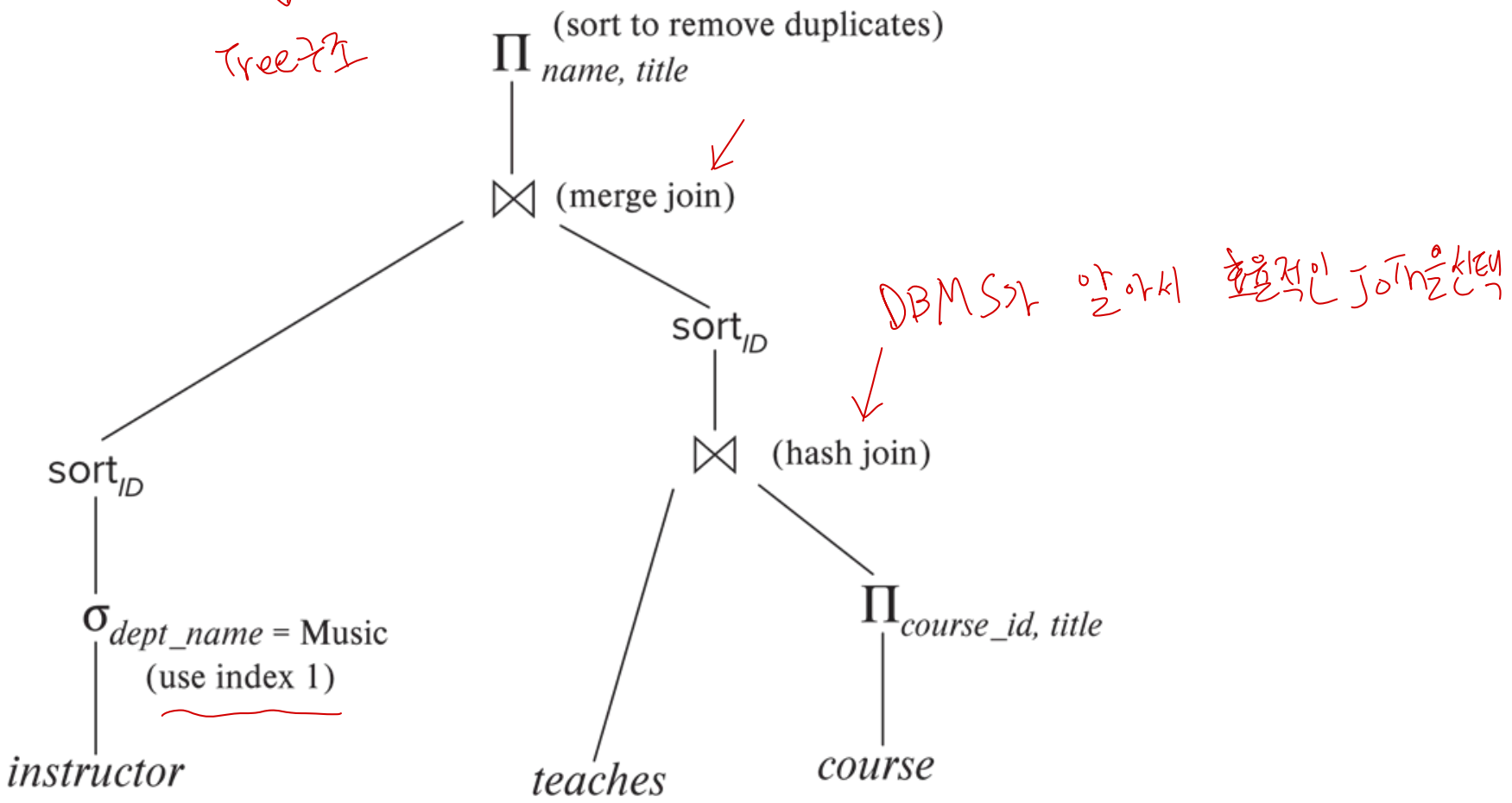
Database Systems
Lecture15 – Chapter 16: Query Optimization

Beomseok Nam (남범석)
bnam@skku.edu

Query Evaluation Plan

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

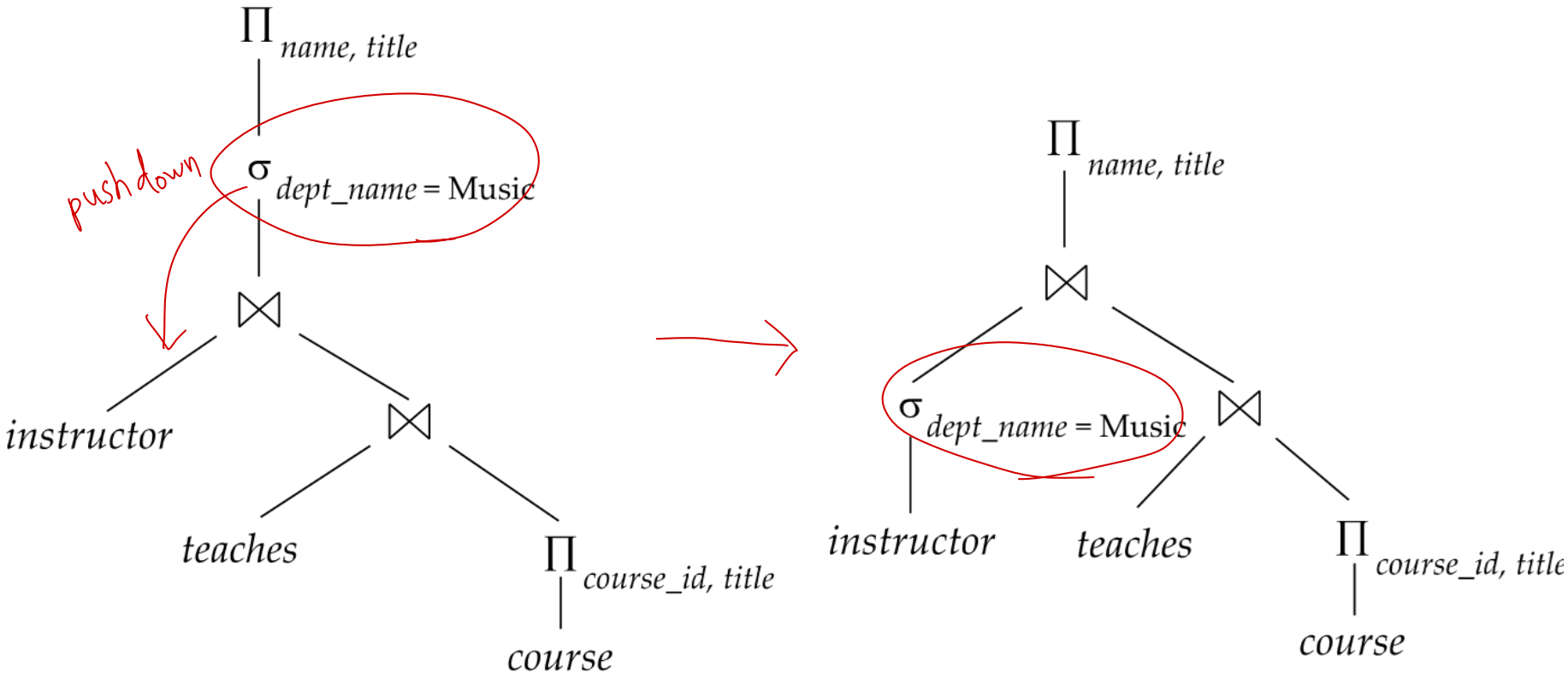


Viewing Query Evaluation Plans

- Most database support **explain** <query>
  - Some syntax variations between databases
  - Oracle: **explain plan for** <query>
  - SQL Server: **set showplan_text on** <query>
  - MySQL: **explain** <query>
  - PostgreSQL: **explain analyse** <query>
    - E.g., **explain analyse** select * from table
    - Shows runtime statistics, in addition to the plan
    - PostgreSQL shows cost as *F..L*
      - *F:* is the cost of delivering first tuple
      - *L:* is cost of delivering all results

Equivalent Query Evaluation Plans

- Different but equivalent expressions
- Different algorithms for each operation



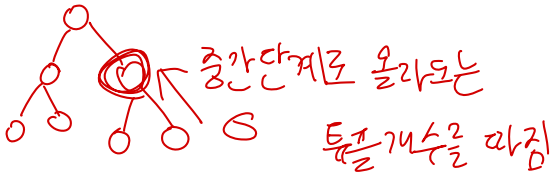
(a) Initial expression tree

(b) Transformed expression tree

Cost-based Query Optimization

- Steps in **cost-based query optimization**
  1. Generate logically equivalent expressions using **equivalence rules** *(next slides)*
  2. Annotate resultant expressions to get alternative query plans
  3. Choose the cheapest plan based on **estimated cost**

- Estimation of plan cost based on:
  - Statistics about relations.
    - 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



Transformation of Relational Expressions

- Two relational algebra expressions are said to be **equivalent** if the two expressions generate the same set of tuples
  - Note: order of tuples is irrelevant

- **equivalence rule** →
  - replace expression of first form by second, or vice versa

Equivalence Rules

1. Conjunctive selection operations → a sequence of 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 projection operation is needed

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

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'd)

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

교환법칙

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

6. (a) Natural join operations are associative:

결합법칙

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

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

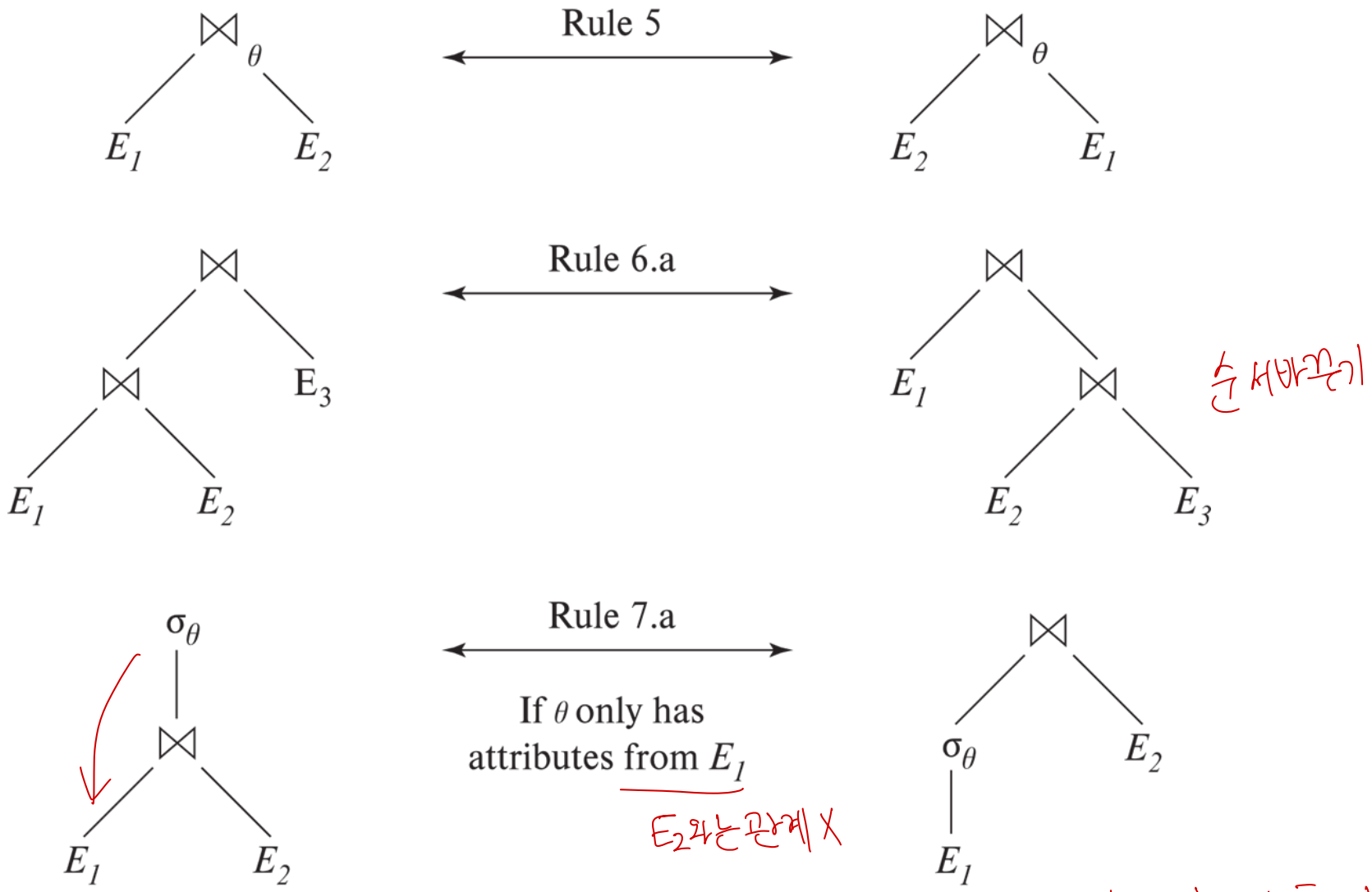
$$(E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \wedge \theta_3} E_3 \equiv E_1 \bowtie_{\theta_1 \wedge \theta_3} (E_2 \bowtie_{\theta_2} E_3)$$

where θ_2 involves attributes from only E_2 and E_3.

E_1에는 없다

$$\left(\sigma_{\theta_2}(E_1 \bowtie_{\theta_1} E_2)\right) \bowtie_{\theta_3} E_3 \rightarrow \left(E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2\right) \bowtie_{\theta_3} E_3$$

Pictorial Depiction of Equivalence Rules



순서바꾸기

E2와는 관계 X

Filter push down 방법 ⇒ select 먼저해주면 input 데이블사이즈를 줄일수있어 성능↑

Equivalence Rules (Cont'd)

7. The selection operation distributes over the theta join operation as follows:

(a) if all the attributes in θ_0 involve only the attributes of E_1

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

slow fast

(b) if 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'd)

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

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

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

이때만 사용하자

θ가 L_1, L_2에 속하지 않는 속성을 요구할 때

projection을 만지지 않을 때

join attribute를 잃는다면 변환에서는 안됨

(b) In general, consider a join $E_1 \bowtie_\theta E_2$.

- Let L_1 and L_2 be 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 $L_1 \cup L_2$.

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

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

Equivalence Rules (Cont'd)

9. The set operations union and intersection are commutative

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

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

$\rightsquigarrow E_1 - E_2 \neq E_2 - E_1$

(set difference is not commutative).

10. Set union and intersection are associative.

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

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

Equivalence Rules (Cont'd)

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

    a. $\sigma_\theta (E_1 \cup E_2) \equiv \sigma_\theta (E_1) \cup \sigma_\theta(E_2)$ → 성능이 더 좋다

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

    preceding equivalence does not hold for $\cup$

12. The projection operation distributes over union

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

Equivalence Rules (Cont'd)

13. Selection distributes over aggregation as below

$$\sigma_{\theta}({}_{A}g_{f}(E)) \equiv {}_{A}g_{f}(\sigma_{\theta}(E))$$

provided θ only involves attributes in G

14. a. Full outerjoin is commutative:

$$E_1 \rtimes E_2 \equiv E_2 \rtimes E_1$$

b. Left and right outerjoin are not commutative, but:

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

c. Left and right outerjoins are not associative

$$(r \bowtie s) \bowtie t \not\equiv r \bowtie (s \bowtie t)$$

15. Selection distributes over left and right outerjoins as below, provided θ_1 only involves attributes of E_1

a. $\sigma_{\theta_1}(E_1 \rtimes_{\theta} E_2) \equiv (\sigma_{\theta_1}(E_1)) \rtimes_{\theta} E_2$

b. $\sigma_{\theta_1}(E_1 \ltimes_{\theta} E_2) \equiv E_2 \rtimes_{\theta} (\sigma_{\theta_1}(E_1))$

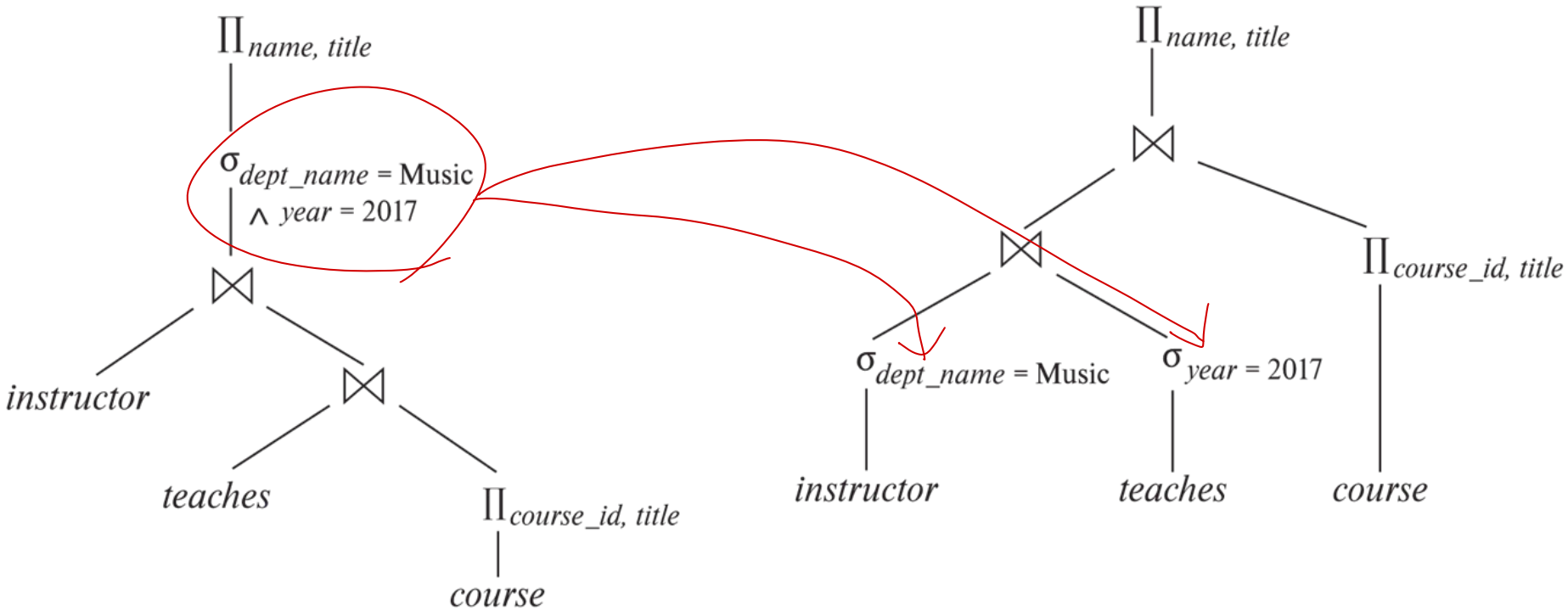
Transformation Example: Filter Pushdown

- 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.

Transformation Example: Filter Pushdown



(a) Initial expression tree

(b) Tree after multiple transformations

Transformation Example: Cascading Join

- (Join Associativity) ⋈

  For all relations r_1, r_2, and r_3,

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

cost가 싸진다 (fast해진다, 성능이 좋아진다)

- 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.

input size 둘다 100 가정

r⋈s의 Join output이 small/large 인지 결정하는 요소?

→ Join되는 속성의 값이 균일하다면 output 100 (균일하게 퍼짐)

→ 〃 극단적 (100개가 둘다 같은 값) output 100X100

같은 속성의

→ 속성의 값이 둘이 다르다면

output < input size

쿼리 최적화의 주요 원칙

intermediate size가 작아지면 (조인하는 테이블) operation을 push down 시킨다.

More Transformation Example

- Consider the expression

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

- Could compute $teaches \bowtie \Pi_{course\_id,\ title}(course)$ first, and join with $\sigma_{dept=\text{"Music"}}(instructor)$

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

- If only a small fraction of the instructors are from the Music dept.
  - 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

```
repeat
    for each equivalent_expression in EquivalentSet
        for each subexpression in equivalent_expression
            apply all applicable equivalence rules
            add newly generated expressions to EquivalentSet
until no new expressions are added
```

parser → E → EquivalentSet

EquivalentSet에 대해 튜플 1개씩 돌며

(E ⋈ ⋯)와 같은 복잡한 표현

→ 추가되는 튜플이 없을 때까지

⟹ exhaustive search 철저한 탐색

- The above method is too expensive in space and time
- Practical query optimizers use the following approaches:
  1. Dynamic programming-like cost-based optimization
  2. Heuristics-based optimization

비슷한 아이디어

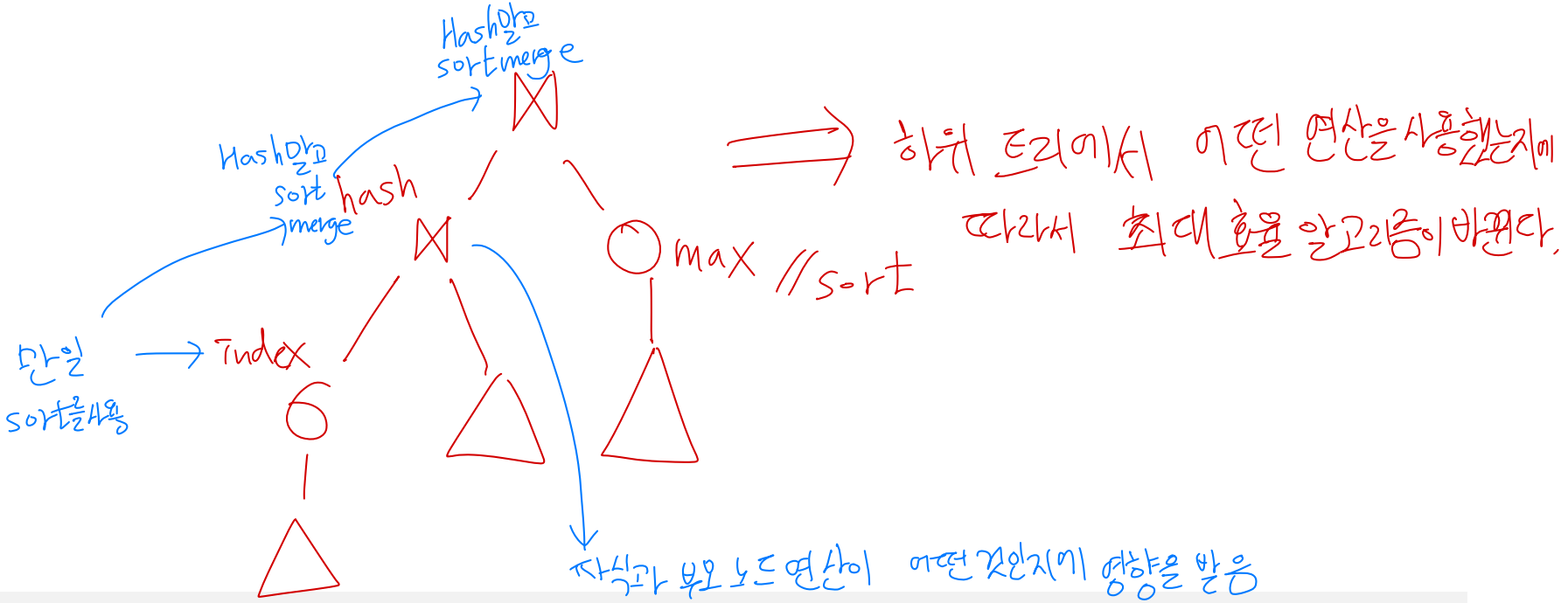
Cost Estimation of Each Operator

- Estimation of each operator cost needs statistics of input relations
  - E.g., number of tuples, sizes of tuples
- Inputs are often 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

key distribution

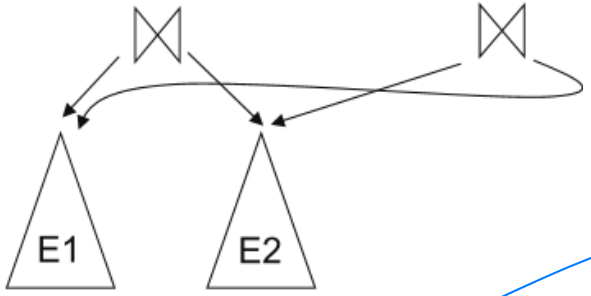
Interaction of Query Processing Algorithms

- Choosing the cheapest algorithm for each operation may not yield best overall algorithm.
- E.g.
  - merge-join provides a sorted output which reduces the cost for an aggregation
  - nested-loop join may provide opportunity for pipelining



Enumeration while Reusing Common Expressions

- Complexity 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



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

E_2의 값은 계산되어 있음
→ 다시 계산하지 X

→ 같은 하위 표현식은 어딘가에 저장해 둔 다음 다시 계산하지 않고 재사용함

  - 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

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. 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 $\{r_1, r_2, \ldots r_n\}$ is computed only once and stored for future use.

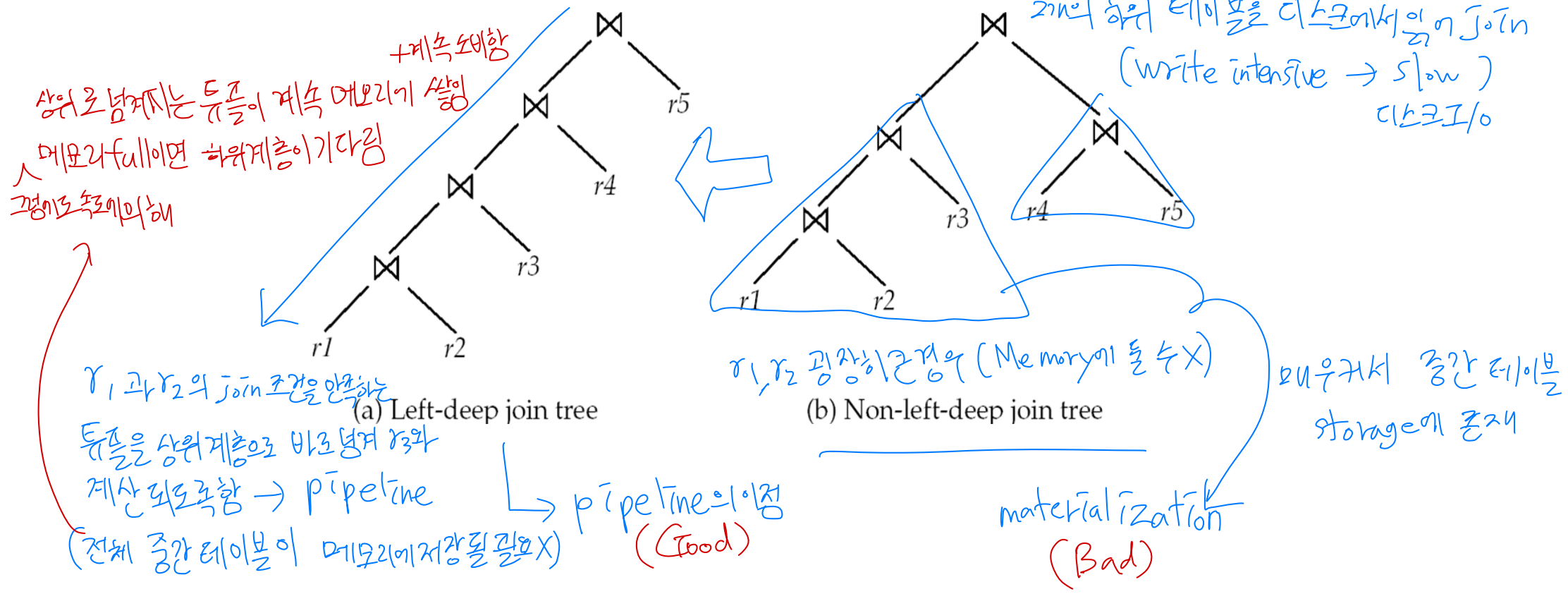
Join Order Optimization Algorithm

```
procedure findbestplan(S)
  if (bestplan[S].cost ≠ ∞)
        return bestplan[S]
  // else bestplan[S] has not been computed earlier, compute it now
  if (S contains only 1 relation)
     set bestplan[S].plan and bestplan[S].cost based on the best way
     of accessing S  using selections on S and indices (if any) on S else
  for each non-empty subset S1 of S such that S1 ≠ S
        P1= findbestplan(S1)
        P2= findbestplan(S - S1)
        for each algorithm A for joining results of P1 and P2
          … compute plan and cost of using A ..
            if cost < bestplan[S].cost
                  bestplan[S].cost = cost
                  bestplan[S].plan = plan;
  return bestplan[S]
```

Cost-based optimization is expensive, even with dynamic programming.

Join Order Optimization: Left Deep Join Trees

- In **left-deep join trees,** the right-hand-side input for each join is a relation, and the left-hand-side input is the result of an intermediate join.



- If only left-deep trees are considered, time complexity of finding best join order is decreased from $O(3^n)$ to $O(n\ 2^n)$

Interesting Sort Orders

- An **interesting sort order** is a sort order that could make a later operation (join/group by/order by) cheaper
  - An interesting sort order allows a query plan to be locally sub-optimal but globally optimal by producing useful sorted output for later operations.
- Consider $(r_1 \bowtie r_2) \bowtie r_3$ with common attribute A
  - Merge-join on r1 ⋈ r2 may cost more than hash join
  - However, it generates result sorted on A
  - This sorted output makes the next merge-join with r_3 cheaper
  - As a result, the overall join cost can be minimized

Heuristic Optimization

- Reduce the number of choices in cost-based optimization.
- Heuristic method
  - not guaranteed to be optimal
  - But often effective for quick decision making
- Heuristic optimizations
  - **Early Selection:**
    Apply selections as soon as possible to reduce tuple counts.
  - **Early Projection:**
    Project early to limit the number of attributes.
  - **Prioritize Small Results:**
    Perform the most restrictive selections and joins first (i.e., operations yielding smallest intermediate results).

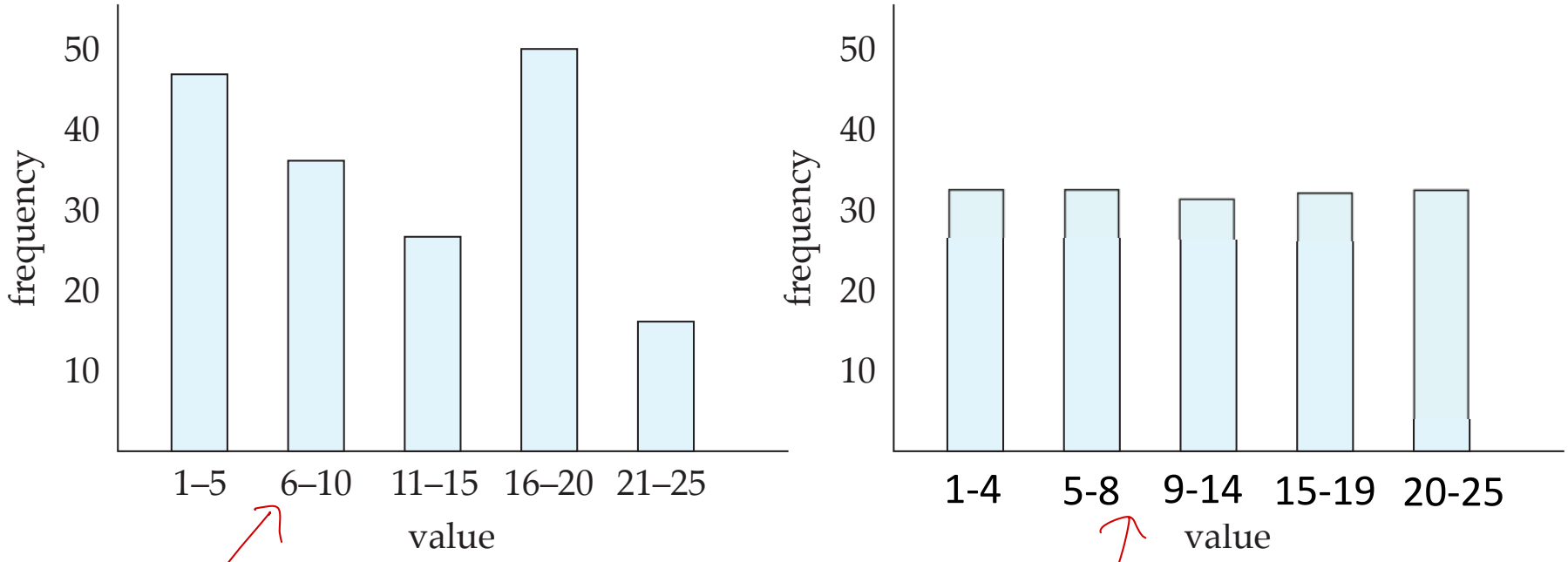
Statistical Information for Cost Estimation

- 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.* (도메인에 대한 정보 포함)
- 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:

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

Statistical Information: Histograms

- Histogram on attribute *age* of relation *person*



- **Equi-width** histograms
- **Equi-depth** histograms break up range such that each range has (approximately) the same number of tuples
- Many databases also store *n* **most-frequent values** and their counts
  - Histogram is built on remaining values only

Statistical Information: Histograms

- Histograms and other statistics usually computed based on a **random sample**
- Statistics may be out of date
  - Some database require a **analyze** command to update statistics
  - Others automatically recompute statistics
    - e.g., when number of tuples in a relation changes by some percentage

일부 데이터로 추정한다.