Chapter 13: Query Optimization

Database System Concepts, 6th Ed.

Chapter 13: Query Optimization

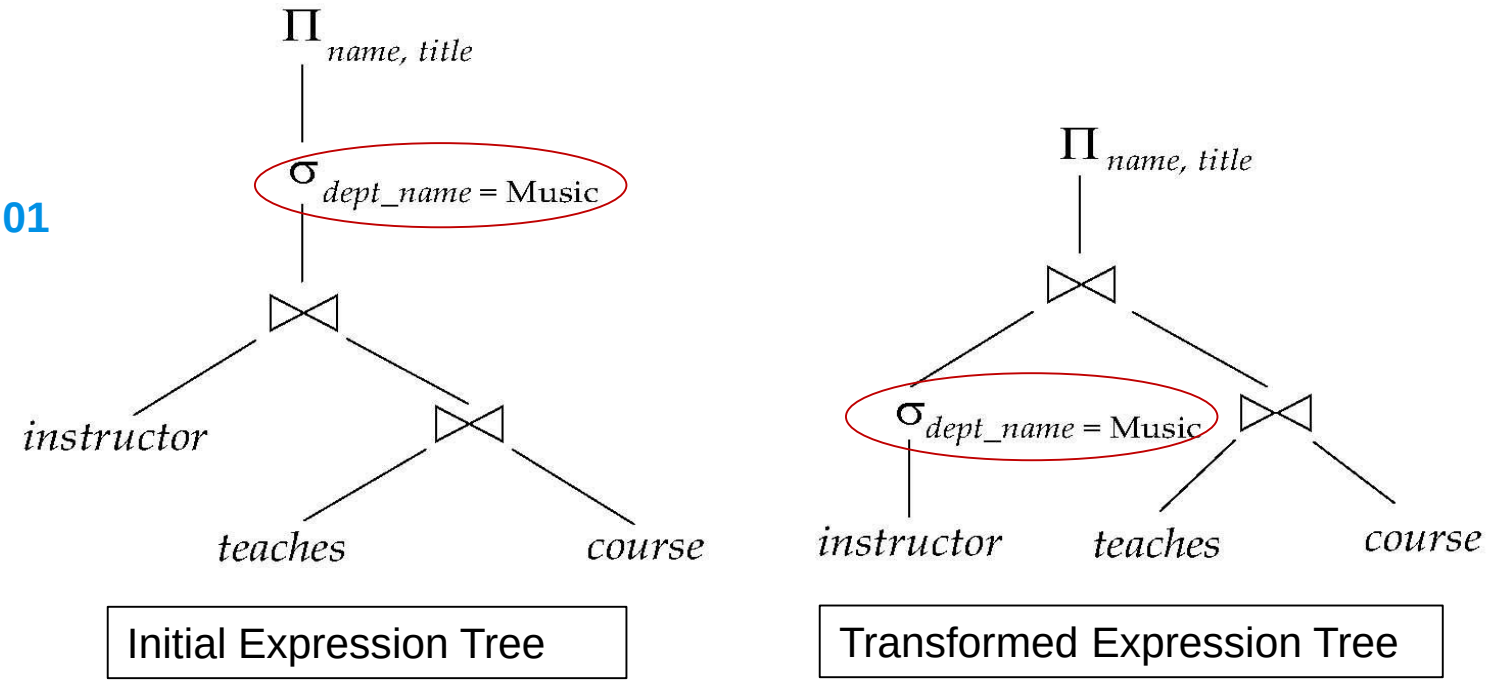
- 13.1 Overview
- 13.2 Transformation of Relational Expressions
- 13.3 Estimating Statistics of Expression
- 13.4 Choice of Evaluation Plans
- 13.5 Materialized views**
- 13.6 Advanced Topics in Query Optimization**

Overview [1/3]

- Alternative ways of evaluating a given query
  - Equivalent expressions
  - Different algorithms for each operation in expression

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

Fig 13.01



Initial Expression Tree

Transformed Expression Tree

** Relations generated by two equivalent expressions have the same set of attributes and contain the same set of tuples, although their attributes may be ordered differently

Overview [2/3]

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

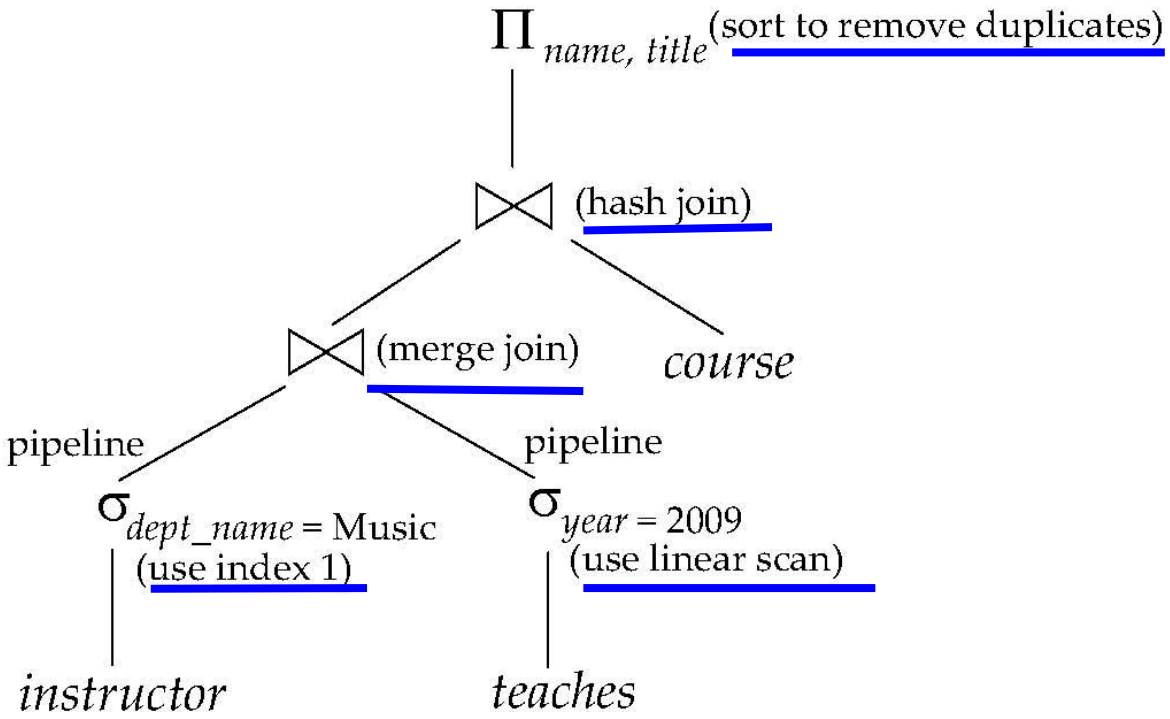


Fig 13.02

- Most commercial DBMS provide a way to view the query execution plan chosen to execute a given query, and the estimated cost for the plan
  - The command like "explain <query>"

Overview [3/3]

- Cost difference between evaluation plans for a query can be enormous
  - E.g. few seconds vs. several 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**

- Estimating cost of query execution plan is 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
    - Ch12에 있는 operation들의 cost formula of disk accesses

Chapter 13: Query Optimization

- 13.1 Overview
- 13.2 Transformation of Relational Expressions
  = Generating Equivalent Expressions
- 13.3 Estimating Statistics of Expression
- 13.4 Choice of Evaluation Plans
- 13.5 Materialized views**
- 13.6 Advanced Topics in Query Optimization**

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
  - Legal database instance에서 같은 결과를 냈던 2개의 euiqvalent한 expression들이 illegal database instance에서 결과를 다르게 낼수도 있다
- 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 [1/5]

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

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

2. Selection operations are commutative

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

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

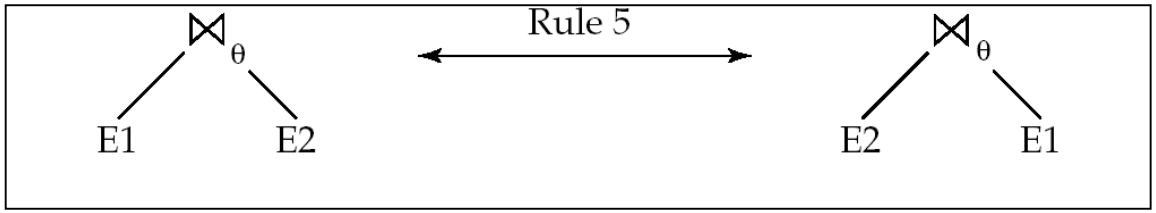
$$\Pi_{L_1}(\Pi_{L_2}(\ldots(\Pi_{Ln}(E))\ldots)) = \Pi_{L_1}(E)$$

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 [2/5]

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

$$E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$$



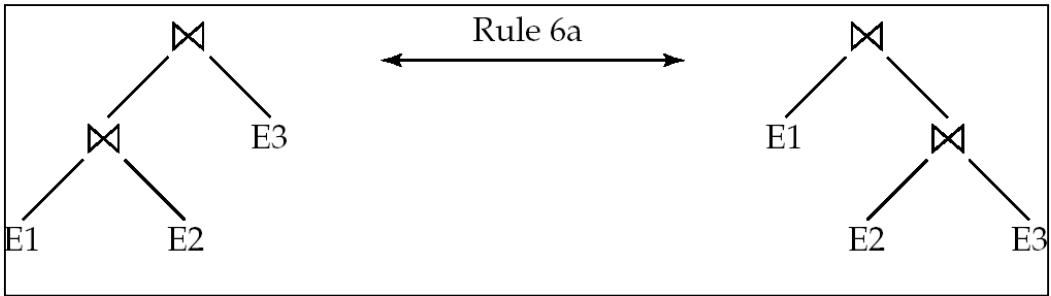
6. (a) Natural join operations are associative:

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

(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.



Equivalence Rules [3/5]

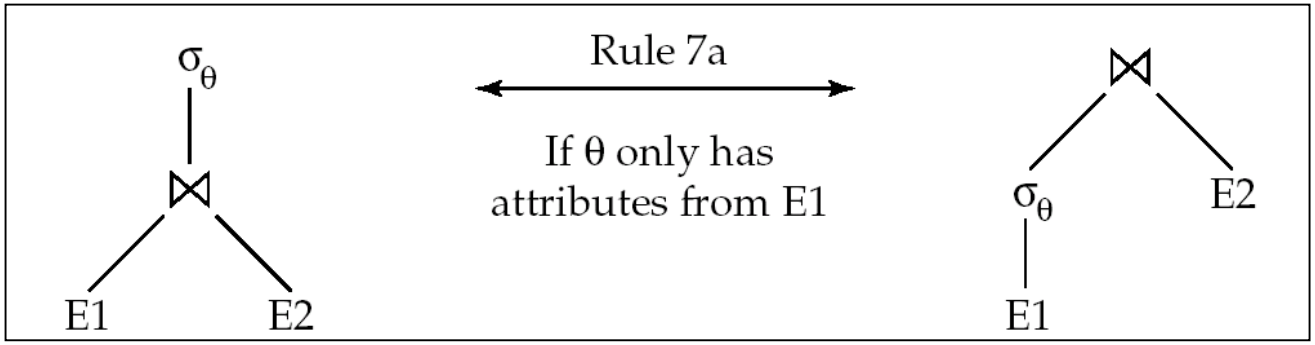
7. The selection operation distributes over the theta join operation under the following two conditions:

(a) When all the attributes in θ_0 involve only the attributes of one of the expressions (E_1) being joined

$$\sigma_{\theta 0}(E_1 \bowtie_{\theta} E_2) = (\sigma_{\theta 0}(E_1)) \bowtie_{\theta} E_2$$

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

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



보조자료

Equivalence Rule 7-a [1/2]

- θ_0의 모든 속성들이 조인되는 expression(E_1) 의 한 쪽의 속성들로만 이루어져 있을 때

1) $\sigma_{\theta 0}(E1 \bowtie_{\theta} E2)$

branch-city	branch-name	assets	loan-number	amount
Brighton	Perryridge	7100000	L-15	1500
Brighton	Perryridge	7100000	L-16	1300

$\sigma_{assets > 2000000}$

branch-city	branch-name	assets	loan-number	amount
Bennington	Pownal	300000	L-23	2000
Horseneck	Mianus	400000	L-11	900
Brighton	Perryridge	7100000	L-15	1500
Brighton	Perryridge	7100000	L-16	1300

$\bowtie$

branch

branch-name	branch-city	assets
Brighton	Perryridge	7100000
Downtown	Brooklyn	9000000
Mianus	Horseneck	400000
North Town	Rye	3700000
Perryridge	Horseneck	1700000
Pownal	Bennington	300000
Redwood	Palo Alto	2100000
Round Hill	Horseneck	8000000

loan

loan-number	branch-name	amount
L-11	Horseneck	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Pownal	2000
L-93	Mianus	500

Equivalence Rule 7-a [2/2]

보조자료

2) $(\sigma_{\theta 0}(E_1)) \bowtie_\theta E_2$

Result

branch-city	branch-name	assets	loan-number	amount
Brighton	Perryridge	7100000	L-15	1500
Brighton	Perryridge	7100000	L-16	1300

Smaller

branch

branch-name	branch-city	assets
Brighton	Perryridge	7100000
Downtown	Brooklyn	9000000
North Town	Rye	3700000
Redwood	Palo Alto	2100000
Round Hill	Horseneck	8000000

$\sigma_{assets > 2000000}$

loan

loan-number	branch-name	amount
L-11	Horseneck	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Pownal	2000
L-93	Mianus	500

branch-name	branch-city	assets
Brighton	Perryridge	7100000
Downtown	Brooklyn	9000000
Mianus	Horseneck	400000
North Town	Rye	3700000
Perryridge	Horseneck	1700000
Pownal	Bennington	300000
Redwood	Palo Alto	2100000
Round Hill	Horseneck	8000000

So, $\sigma_{\theta 0}(E1 \bowtie_\theta E2) = (\sigma_{\theta 0}(E1)) \bowtie_\theta E2$

Equivalence Rule 7-b [1/2]

보조자료

- θ_1이 E_1 의 속성들에만 관여되고, θ_2가 E_2의 속성들에만 관여될 때

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

branch-city	branch-name	assets	loan-number	amount
Brighton	Perryridge	7100000	L-15	1500

$\sigma_{assets > 2000000 \text{ and } amount > 1300}$

branch-city	branch-name	assets	loan-number	amount
Bennington	Pownal	300000	L-23	2000
Horseneck	Mianus	400000	L-11	900
Brighton	Perryridge	7100000	L-15	1500
Brighton	Perryridge	7100000	L-16	1300

branch

branch-name	branch-city	assets
Brighton	Perryridge	7100000
Downtown	Brooklyn	9000000
Mianus	Horseneck	400000
North Town	Rye	3700000
Perryridge	Horseneck	1700000
Pownal	Bennington	300000
Redwood	Palo Alto	2100000
Round Hill	Horseneck	8000000

loan

loan-number	branch-name	amount
L-11	Horseneck	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Pownal	2000
L-93	Mianus	500

Equivalence Rule 7-b [2/2]

보조자료

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

Smaller

branch-name	branch-city	assets
Brighton	Perryridge	7100000
Downtown	Brooklyn	9000000
North Town	Rye	3700000
Redwood	Palo Alto	2100000
Round Hill	Horseneck	8000000

Result

branch-city	branch-name	assets	loan-number	amount
Brighton	Perryridge	7100000	L-15	1500

Smaller

loan-number Horseneck	branch-name	amount
L-14	Downtown	1500
L-15	Perryridge	1500
L-23	Pownal	2000

$\bowtie$

$\sigma_{assets > 2000000}$

$\sigma_{amount > 1500}$

branch

branch-name	branch-city	assets
Brighton	Perryridge	7100000
Downtown	Brooklyn	9000000
Mianus	Horseneck	400000
North Town	Rye	3700000
Perryridge	Horseneck	1700000
Pownal	Bennington	300000
Redwood	Palo Alto	2100000
Round Hill	Horseneck	8000000

loan

loan-number	branch-name	amount
L-11	Horseneck	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Pownal	2000
L-93	Mianus	500

So, $\sigma_{\theta 1 \wedge \theta 2}(E1 \bowtie_\theta E2) = (\sigma_{\theta 1}(E1)) \bowtie_\theta (\sigma_{\theta 2}(E2))$

Equivalence Rules [4/5]

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

(a) Let L_1 and L_2 be sets of attributes from E_1 and E_2, respectively

If θ involves only attributes from $L_1 \cup L_2$:

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

(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 in join condition θ, but are not in $L_1 \cup L_2$
- Let L_4 be attributes of E_2 in join condition θ, but are not in $L_1 \cup L_2$

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

Intuition: Join에 참여하는 attribute와 나중에 Projection에 참여하는 attribute를 이용해서 base relation을 projection으로 미리 smaller relation으로 만들어 처리

The Equivalence Rule 8-a [1/3]

보조자료

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

- E1 = LOAN, E2 = BRANCH
- L1 = {loan-number, branch-name}, L2 = {branch-name, assets}
- θ = BRANCH.assets > 75000
  - In this condition, join condition involves only attributes in $L_1 \cup L_2$

LOAN

loan-number	branch-name	amount
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500

BRANCH

branch-name	branch-city	assets
Brighton	Brooklyn	7100000
Downtown	Brooklyn	9000000
Mianus	Horseneck	400000
North Town	Rye	3700000
Perryridge	Horseneck	1700000
Pownal	Bennington	300000
Redwood	Palo Alto	2100000
Round Hill	Horseneck	8000000

Equivalence Rule 8-a: left-hand side [2/3]

보조자료

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

LOAN

loan-number	*branch-name*	*amount*
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500

BRANCH

branch-name	*branch-city*	*assets*
Brighton	Brooklyn	7100000
Downtown	Brooklyn	9000000
Mianus	Horseneck	400000
North Town	Rye	3700000
Perryridge	Horseneck	1700000
Pownal	Bennington	300000
Redwood	Palo Alto	2100000
Round Hill	Horseneck	8000000

$(E_1 \bowtie_\theta E_2)$

loan-number	branch-name	amount	branch-city	assets
L-11	Round-Hill	900	Horse-neck	8000000
L-14	Downtown	1500	Brook-lyn	9000000
L-17	Downtown	1000	Brook-lyn	9000000

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

loan-number	branch-name	assets
L-11	Round-Hill	8000000
L-14	Downtown	9000000
L-17	Downtown	9000000

Equivalence Rule 8-a: right-hand side [3/3]

보조자료

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

$(\Pi_{L_1}(E_1))$

Smaller

loan-number	branch-name
L-11	Round Hill
L-14	Downtown
L-15	Perryridge
L-16	Perryridge
L-17	Downtown
L-23	Redwood
L-93	Mianus

$(\Pi_{L_2}(E_2))$

Smaller

branch-name	assets
Brighton	7100000
Downtown	9000000
Mianus	400000
North Town	3700000
Perryridge	1700000
Pownal	300000
Redwood	2100000
Round Hill	8000000

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

loan-number	branch-name	assets
L-11	Round-Hill	8000000
L-14	Downtown	9000000
L-17	Downtown	9000000

Equivalence Rule 8-b [1/3]

보조자료

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

- E1 = LOAN, E2 = BRANCH
- L1 = {branch-name}, L2 = {branch-name}
- θ = LOAN.amount > 950 and BRANCH.branch-city = 'Brooklyn'
- L3 = {amount}, L4 = {branch-city}

LOAN

loan-number	*branch-name*	*amount*
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500

BRANCH

branch-name	*branch-city*	*assets*
Brighton	Brooklyn	7100000
Downtown	Brooklyn	9000000
Mianus	Horseneck	400000
North Town	Rye	3700000
Perryridge	Horseneck	1700000
Pownal	Bennington	300000
Redwood	Palo Alto	2100000
Round Hill	Horseneck	8000000

Equivalence Rule 8-b: left-hand side [2/3]

보조자료

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

$(E_1 \bowtie_\theta E_2)$

loan-number	branch-name	amount	branch-city	assets
L-14	Downtown	1500	Brook-lyn	9000000
L-17	Downtown	1000	Brook-lyn	9000000

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

branch-name
Downtown

Equivalence Rule 8-b: right-hand side [3/3]

보조자료

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

$(\Pi_{L_1 \cup L_3}(E_1))$

Smaller

branch-name	*amount*
Round Hill	900
Downtown	1500
Perryridge	1500
Perryridge	1300
Downtown	1000
Redwood	2000
Mianus	500

$(\Pi_{L_2 \cup L_4}(E_2))$

Smaller

branch-name	*branch-city*
Brighton	Brooklyn
Downtown	Brooklyn
Mianus	Horseneck
North Town	Rye
Perryridge	Horseneck
Pownal	Bennington
Redwood	Palo Alto
Round Hill	Horseneck

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

branch-name	amount	branch-city
Downtown	1500	Brook-lyn
Downtown	1000	Brook-lyn

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

branch-name
Downtown

Equivalence Rules [5/5]

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
  - $\Pi_{name,\ title}((\sigma_{dept\_name = \text{“Music”}}(instructor)) \bowtie$
    $(teaches \bowtie \Pi_{course\_id,\ title}\ (course)))$
- Point: Performing the selection as early as possible reduces the size of the relation to be joined

Example: Multiple Transformations [1/2]

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

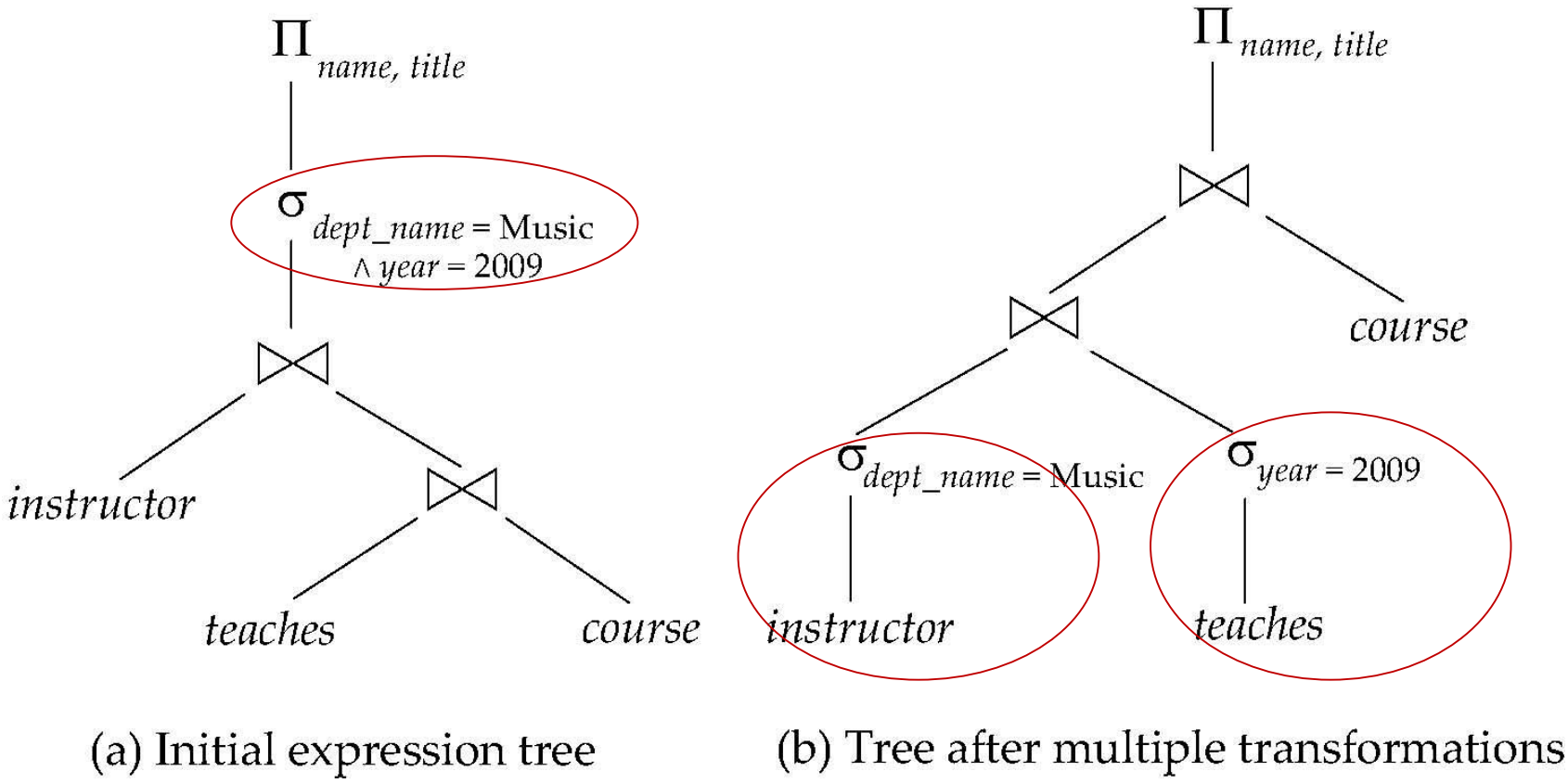
- Transformation using join associatively (Rule 6a):
  - $\Pi_{name,\ title}(\sigma_{dept\_name = \text{“Music”} \wedge year = 2009} ((instructor \bowtie teaches) \bowtie \Pi_{course\_id,\ title} (course)))$

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

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

EXAMPLE: Multiple Transformations [2/2]

$\Pi_{name,\ title}(\sigma_{dept\_name = \text{“Music”} \wedge gear = 2009}$
$(instructor \bowtie (teaches \bowtie \Pi_{course\_id,\ title}(course))))$



(a) Initial expression tree

(b) Tree after multiple transformations

Fig 13.04

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:

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

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

Join Ordering

- For all relations r_1, r_2, and r_3,

$$(r_1 \bowtie r_2) \bowtie r_3 = r_1 \bowtie (r_2 \bowtie r_3) \quad \text{(Join Associativity)}$$

- 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

- 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

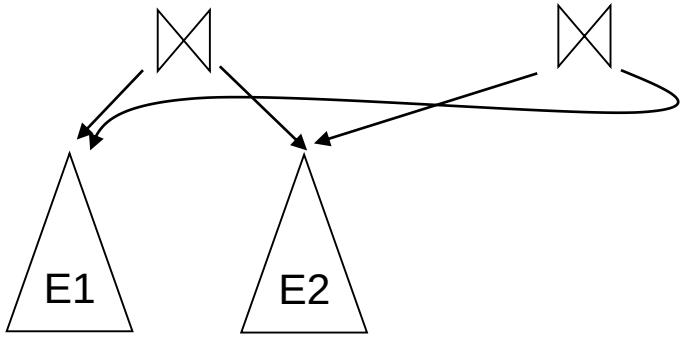
- Query optimizers use equivalence rules to **systematically** generate expressions equivalent to the given expression
- Can generate all equivalent expressions as follows:
  - Repeat
    - apply all applicable equivalence rules on every subexpression of every equivalent expression found so far
    - 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

- Techniques for reducing enumeration of equivalent Expressions
  - Sharing common subexpressions
  - Dynamic Programming

Reducing Enumeration of Equivalent Expressions

- 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

Chapter 13: Query Optimization

- 13.1 Overview
- 13.2 Transformation of Relational Expressions
- 13.3 Estimating Statistics of Expression
  = Statistics for Cost Estimation
- 13.4 Choice of Evaluation Plans
- 13.5 Materialized views**
- 13.6 Advanced Topics in Query Optimization**

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

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

- $SC(A, r)$: selection cardinality of attribute A of relation r
  - average number of records that satisfy equality on A
- f_i: average fan-out of internal nodes of index i, for B+-trees
- HT_i: number of levels in index i (i.e., the height of i & on attribute A of relation r)
  - For a B+-tree $HT_i = \lceil \log_{fi}(V(A,r)) \rceil$
  - For a hash index, $HT_i = 1$
- LB_i: number of lowest-level index blocks in i (i.e, the # of blocks at the leaf level)

Histograms

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

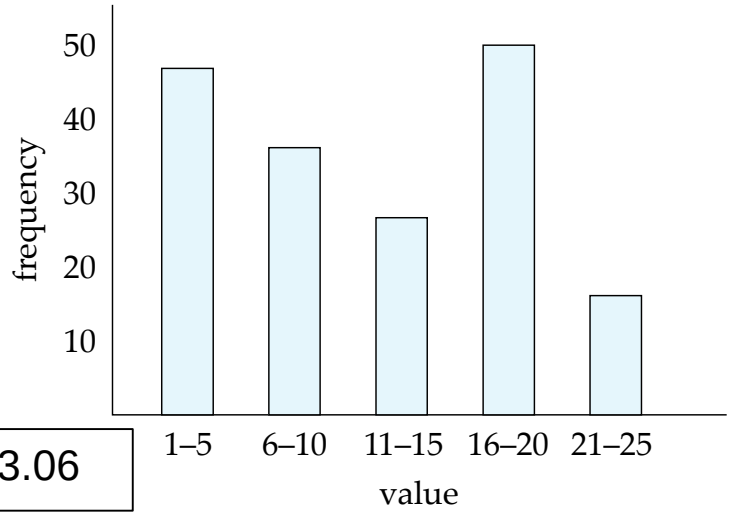
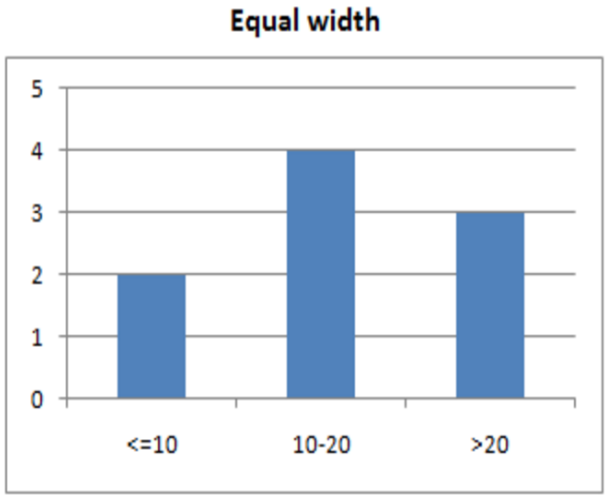
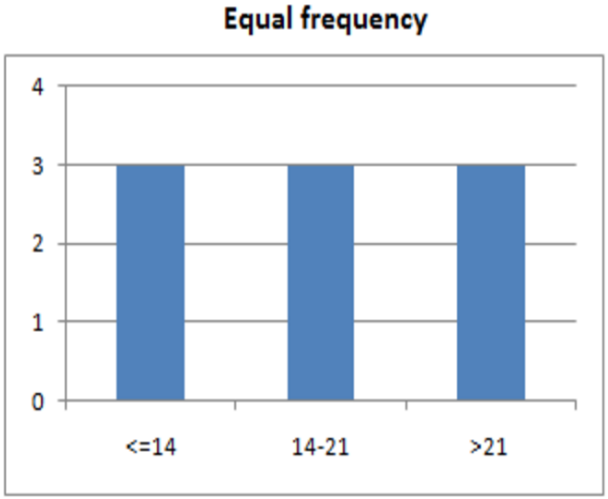


Fig 13.06

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





Size Estimating of Selection Operation

- **Equality selection** $\sigma_{A=V}(r)$
  - $n_r / V(A,r)$: number of records that will satisfy the selection
  - Equality condition on a key attribute: *size estimate* = 1

- $\sigma_{A \le V}(r)$ (case of $\sigma_{A \ge V}(r)$ 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, the above estimate can be refined
  - In absence of statistical information *c* is assumed to be $n_r / 2$

Size Estimation of Complex Selections

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

- **Conjunction:** $\sigma_{\theta 1 \wedge \theta 2 \wedge \ldots \wedge \theta n}(r)$.

  Assuming independence, estimate of tuples in the result is:

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

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

  Estimated number of tuples:

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

- **Negation:** $\sigma_{\neg\theta}(r)$.

  Estimated number of tuples: $n_r - size(\sigma_\theta(r))$

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!*)

Size Estimation of Joins [1/2]

- The Cartesian product r x s contains $n_r . n_s$ tuples; each tuple occupies $s_r + s_s$ bytes
- If $R \cap S = \varnothing$, then $r \bowtie s$ is the same as r x s.
- If $R \cap S$ is a key for R, then a tuple of s will join with at most one tuple from r
  - Then, the number of tuples in $r \bowtie s$ is no greater than the number of tuples in s.
- If $R \cap S$ *in* S is a foreign key in S referencing R, then the number of tuples in $r \bowtie s$ is exactly the same as the number of tuples in s.
    - 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 join result has exactly n_{takes} tuples, which is 10000

Size Estimation of Joins [2/2]

- If $R \cap S = \{A\}$ is not a key for R or S,
  (If we assume that every tuple t in R produces tuples in $R \bowtie S$),
  the number of tuples in $R \bowtie S$ is estimated to be: $\frac{n_r * n_s}{V(A,s)}$

  If the reverse is true, the estimate obtained will be: $\frac{n_r * n_s}{V(A,r)}$

  The lower of these two estimates is probably the more accurate one.
- Can improve on above if histograms are available
- Example: Compute the size estimates for *student* $\bowtie$ *takes* without using information about foreign keys:
  - $n_{student}$ = 5,000 n_{takes} = 10000.
  - *V(ID, takes)* = 2500, and *V(ID, student)* = 5000
  - The two estimates: 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 $\Pi_A(r)$ = $V(A,r)$
- Aggregation : estimated size of ${}_A\boldsymbol{g}_F(r)$ = $V(A,r)$
- Set operations
  - For unions/intersections of selections on the same relation: rewrite and use size estimate for selections
    - E.g. $\sigma_{\theta 1}(r) \cup \sigma_{\theta 2}(r)$ can be rewritten as $\sigma_{\theta 1}\,\sigma_{\theta 2}(r)$
  - For operations on different relations:
    - estimated size of $r \cup s$ = size of r + size of s
    - estimated size of $r \cap s$ = minimum size of r and size of s
    - estimated size of $r - s$ = r
    - All the three estimates may be quite inaccurate, but provide upper bounds on the sizes.
- Outer join:
  - Estimated size of $r \mathbin{⟕} s$ = *size of* $r \bowtie s$ + *size of* r
    - Case of right outer join is symmetric
  - Estimated size of $r \mathbin{⟗} s$ = *size of* $r \bowtie s$ + size of r + size of s

Estimating Distinct Values [1/2]

Selections: $\sigma_\theta(r)$

- If θ forces A to take a specified value: $V(A,\sigma_\theta(r)) = 1$.
  - e.g., $A = 3$
- If θ forces A to take on one of a specified set of values:
  $V(A,\sigma_\theta(r))$ = number of specified values.
  - (e.g., ($A = 1$ *V* $A = 3$ *V* $A = 4$)),
- If the selection condition θ is of the form *A op r*
  estimated $V(A,\sigma_\theta(r)) = V(A.r) * s$
  - where *s* is the selectivity of the selection.
- In all the other cases: use approximate estimate of
  $\min(V(A,r), n_{\sigma\theta(r)})$
  - More accurate estimate can be got using probability theory, but this one works fine generally

Estimating Distinct Values [2/2]

Joins: $r \bowtie s$

- If all attributes in A are from r
  estimated $V(A, r \bowtie s) = \min(V(A,r), n_{r \bowtie s})$
- If A contains attributes $A1$ from r and $A2$ from s, then estimated
  $V(A,r \bowtie s) = \min(V(A1,r)*V(A2 - A1,s), V(A1 - A2,r)*V(A2,s), n_{r \bowtie s})$
  - More accurate estimate can be got using probability theory, but this one works fine generally

Projection: (Π) Estimation of distinct values are straightforward for projections.

- They are the same in $\Pi_{A\,(r)}$ as in r.

Aggregation: The same holds for grouping attributes of aggregation.

- For aggregated values
  - For min(A) and max(A), the number of distinct values can be estimated as min(V(A,r), V(G,r)) where G denotes grouping attributes
  - For other aggregates, assume all values are distinct, and use V(G,r)

Estimating Distinct Values in Join [1/2]

보조자료

A의 모든 속성이 r에 있을때의 estimated $V(A, r \bowtie s) = \min(V(A,r), n_{r \bowtie s})$

❖ Example A = (B)

r

K	B	C
A1	B1	C1
A1	B2	C2
A2	B3	C1
A2	B1	C1
A3	B5	C4

s (case 1)

C	D
C1	D1
C2	D1
C3	D2
C4	D3
C5	D4

s (case 2)

C	D
C1	D1
C5	D2
C12	D3
C5	D4
C6	D5

$r \bowtie s$ (case 1)

K	B	C	D
A1	B1	C1	D1
A2	B3	C1	D1
A2	B1	C1	D1
A1	B2	C2	D1

$\Pi_B(r \bowtie s)$ (case 1)

B
B1
B3
B2

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

$r \bowtie s$ (case 2)

K	B	C	D
A1	B1	C1	D1
A2	B3	C1	D1

$\Pi_B(r \bowtie s)$ (case 2)

B
B1
B3

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

Estimating Distinct Values in Join [2/2]

보조자료

A의 속성 A1은 r에, A2는 s에 있을때의 estimated $V(A, r \bowtie s)$ = min($V(A1,r) * V(A2 - A1,s)$, $V(A1 - A2,r) * V(A2,s)$, $n_{r \bowtie s}$)

❖ Example A = (B, D), 처음 두 개의 경우는 대칭적이므로 처음과 세번째 경우만 고려

r

K	B	C
A1	B1	C1
A1	B2	C2
A2	B2	C1
A2	B1	C1
A3	B2	C4

s (case 1)

C	D
C1	D1
C1	D1
C2	D2
C2	D1

s (case 3)

C	D
C2	D1
C5	D2
C12	D3
C5	D4
C6	D5

$r \bowtie s$ (case 1)

K	B	C	D
A1	B1	C1	D1
…			
A2	B1	C1	D1
A1	B2	C2	D1

$\Pi_A(r \bowtie s)$ (case 1)

B	D
B1	D1
B1	D2
B2	D1
B2	D2

$V(A, r \bowtie s) = V(A1,r) * V(A2-A1,s)$
(4 = 2 X 2)
(Join의 결과가 큰 경우)

$r \bowtie s$ (case 3)

K	B	C	D
A1	B2	C2	D1

$\Pi_A(r \bowtie s)$ (case 3)

B	D
B2	D1

$V(A, r \bowtie s) = n_{r \bowtie s}$
(1 = 1)

Chapter 13: Query Optimization

- 13.1 Overview
- 13.2 Transformation of Relational Expressions
- 13.3 Estimating Statistics of Expression
- 13.4 Choice of Evaluation Plans
- 13.5 Materialized views**
- 13.6 Advanced Topics in Query Optimization**

Cost-based Query Optimizer

- Cost Estimation
  - Chapter 12: disk access cost formula of each operator
  - Chapter 13.3: statistics data (number of tuples, sizes of tuples,..)

- Choosing the cheapest algorithm for each operation independently may not yield the 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 2 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

- Consider finding the best join-order for $r_1 \bowtie r_2 \; . \; . \; . \bowtie r_n$.
- The number of relations: (n), The number of joins: (n-1)
- The number of full binary trees with n leaves (with (n-1) internal nodes)

  1/n X 2(n-1) C (n-1) ➔ 1/n X ((2(n-1))! / (n-1)! * (n-1)!) ➔ (2(n-1))! / (n! * (n-1)!)
- Multiply this by n! (permutations of n leaves)
- Then the number of different join orders: $(2(n-1))! / (n-1)!$
  - 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

- 참고: ** Full binary tree: every node가 children을 0 or 2만을 가지는 binary tree

  ** $(2(n-1))! / (n-1)!$ Is much much bigger than 2^n

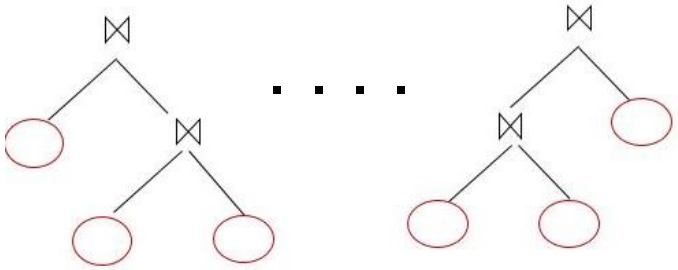
Example: $r_1 \bowtie r_2 \bowtie r_3$

N = 3, $(2(n-1))! / (n-1)!$ ➔ 12

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

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

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



- $(r_1 \bowtie r_2 \bowtie r_3) \bowtie r_4 \bowtie r_5$
  - $(r_1 \bowtie r_2 \bowtie r_3)$: 12 possible orders
  - (result of $(r_1 \bowtie r_2 \bowtie r_3)) \bowtie r_4 \bowtie r_5$: 12 possible orders
  - Therefore, totally 144 possible orders **(Exhaustive Search)**

- The idea of dynamic programming
  - (1) Find the best order of $\{r_1, r_2, r_3\}$ first
  - (2) Find the best order of {result of (1), r_4, r_5}
  - In this case, totally 12+12 = 24 possible orders

이단계에서 결정

Algorithm: Finding the Best Join Order (Dynamic Programming Version)

```
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 on S */
    else for each non-empty subset S1 of S such that S1 ≠ S
            P1= findbestplan(S1)
            P2= findbestplan(S - S1)
            A = best algorithm for joining results of P1 and P2
            cost = P1.cost + P2.cost + cost of A
            if (cost < bestplan[S].cost )
                bestplan[S].cost = cost
                bestplan[S].plan = " execute P1.plan;
                                   execute P2.plan;
                                   join results of P1 and P2 using A "
    return bestplan[S]
```

* Some modifications to allow indexed nested loops joins on relations that have selections (see book)

Dynamic Programming in Join Order Optimization

- To find best join tree for a set of *n* relations:
  - To find best plan for a set *S* of *n* relations, consider all possible plans of the form: $S_1 \bowtie (S - S_1)$ where S_1 is any non-empty subset of *S*
  - Recursively compute costs for joining subsets of *S* to find the cost of each plan
    - $nC1 + nC2 + \ldots nCn = 2^n$
    - Choose the cheapest among the $2^n - 2$ alternatives
    - Space overhead for storing 2^n *costs* ➔ $O(2^n)$
  - Base case for recursion: single relation access plan
    - Apply all selections on R_i using best choice of indices on R_i
  - When plan for any subset is computed, store it and reuse it when it is required again, instead of recomputing it
    - Time complexity due to Dynamic programming ➔ $O(3^n)$
      - $O(3^n)$ by solving the recurrence relation of the next slide
      - much better than $(2(n-1))! / (n-1)!$

Dynamic Programming of Best Join Order $O(3^n)$

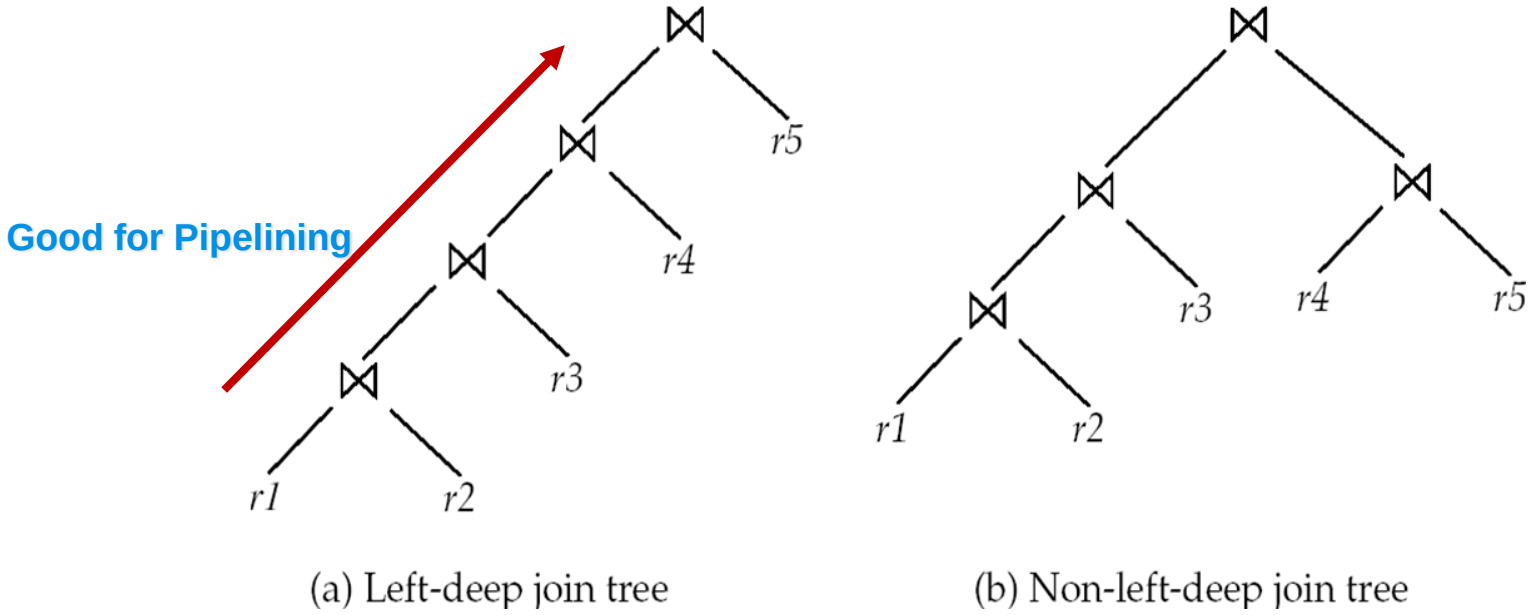
- (Proof)

Answer: Consider the dynamic programming algorithm given in Section 13.4. For each subset having $k+1$ relations, the optimal join order can be computed in time 2^{k+1}. That is because for one particular pair of subsets A and B, we need constant time and there are at most $2^{k+1}-2$ different subsets that A can denote. Thus, over all the $\binom{n}{k+1}$ subsets of size $k+1$, this cost is $\binom{n}{k+1}2^{k+1}$. Summing over all k from 1 to $n-1$ gives the binomial expansion of $((1+x)^n - x)$ with $x = 2$. Thus the total cost is less than 3^n.

Heuristics: 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
- Left-deep join orders are convenient for pipelined evaluation: the right operand is a stored relation and only one input to each join is pipelined
- Figure (a) would enjoy pipelining while pipelining may not be possible in Figure (b)
- The numbe of all left-deep join orders < The numbe of All join orders

Figure 13.08



(a) Left-deep join tree

(b) Non-left-deep join tree

Cost Comparison of Join Order Optimizations

- Join order optimization with exhaustive search
  - Time complexity is $O((2(n-1))!/(n-1)!)$
- Join order optimization with dynamic programming
  - Time complexity is $O(3^n)$
    - With $n = 10$, this number is 59000 instead of 176 billion!
  - Space complexity is $O(2^n)$
- Join order optimization with only left-deep join tree
  - Consider n alternatives with one relation as right-hand side input and the other relations as left-hand side input
  - Modify the join order optimization algorithm (dynamic programming version):
    - Replace "**for each** non-empty subset $S1$ of S such that $S1 \neq S$"
    - By: "**for each** relation r in S
let S1 = S – r "
  - Time complexity 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 Order Property

- Consider the expression $(r_1 \bowtie r_2) \bowtie r_3$ (with A as common attribute)
- An **interesting sort order** is a particular sort order of tuples that could be useful for a later operation
  - Using merge-join to compute $r_1 \bowtie r_2$ may be costlier than hash join, but generates sorted result on A
  - Which in turn may make merge-join with r_3 cheaper, which may reduce cost of join with r_3 and minimizing overall cost
  - Sort order may also be useful for order by and for grouping

- Hence it is 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 with the **interesting sort order**
  - Simple extension of earlier dynamic programming algorithms $O(3^n)$
  - Usually, number of interesting orders is quite small and doesn't affect time/space complexity significantly

Cost Based Optimization with Equivalence Rules

- Ch13.2.4 shows **logical equivalence rules** on relational operations
- The logical equivalence rules are complemented with **physical equivalence rules**
  - **Physical equivalence rules** allow logical query plan to be converted to physical query plan specifying what algorithms are used for each operation
  - A logical operation "Join" is transformed to physical joins (hash-join, merge-join)

- Furthermore, **efficient query 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

- IBM DB2: Starburst Project (by Loura Haas, et al. 1989)
- SYBASE SQL Server: Volcano project

Heuristics in Optimization

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

- Finding the best join order is a big mission in query optimization!
  - We saw the heuristics "left-deep join order"

Structure of Modern Query Optimizer

- Modern commercial query optimizers has a lot of features
  - Integrate the heuristics and the cost-based query optimization
    - Optimal join-order heuristics (only left-deep join orders)
    - Heuristics to push selections and projections down the query tree
    - Reduces optimization complexity and generates plans amenable to pipelined evaluation
  - Use Optimization cost budget to stop optimization early (if cost of plan is less than cost of optimization)
  - Use Query-plan caching to reuse previously computed plan if query is resubmitted
    - Even with different constants in query (parametric query optimization)
  - Use the mixed policy: use simple heuristics for very cheap queries, and perform exhaustive enumeration for more expensive queries
  - Various techniques for special queries
    - Heuristics rewriting of nested queries and aggregation
    - Top K query
    - Holloween query

Optimizing Nested Subqueries** [1/4]

- Intricacies of SQL (such as nested subqueries) complicate query optimization
- 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
  - Input parameters are **correlation variables** from outer level query that are used in the nested subquery **(such as** *instructor.ID)*
- 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 which is inefficient**
  - Note: other conditions in where clause may be used to compute a join (instead of a cross-product) before executing the nested subquery

Intricacy: 복잡, 얽히고 설킴

Optimizing Nested Subqueries [2/4]

- Correlated evaluation may be quite inefficient since
  - a large number of function calls may be made to the nested query
  - there may be unnecessary random I/O as a result
- Hence, 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

- However, 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 [3/4]

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

- Rewrite:

```
select …
 from L1
 where P1 and exists (select *
                      from L2
                      where P2)
```

- To:

```
create table t1 as
        select distinct V
        from L2
        where P2^1

 select …
 from L1, t1
 where P1 and P2^2
```

  - 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 [4/4]

- 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**
- Manu query optimizer provides a limited amount of 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**,
- 사용자는 가급적 복잡한 nested query의 구사를 피해야 한다

Chapter 13: Query Optimization

- 13.1 Overview
- 13.2 Transformation of Relational Expressions
- 13.3 Estimating Statistics of Expression
- 13.4 Choice of Evaluation Plans
- 13.5 Materialized views**
- 13.6 Advanced Topics in Query Optimization**

Materialized Views**

- A **materialized view** is a view whose contents are computed and stored
- Consider the view
  c**reate 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

- The task of keeping a materialized view up-to-date with the underlying data is known as **materialized view maintenance**

- **Materialized view maintenance를 efficient하게 하는것이 query optimization문제이고, materialized view maintenance가 잘되어야 추후에 들어오는 query들의 optimization에 기여할수 있다**

Materialized View Maintenance

- Materialized views can contribute enormously to query performance
- Materialized view maintenance can be done by
  - Manually written code to update the view whenever database relations are updated (error prone, not recommended)
  - Manually defining triggers on insert, delete, and update of each relation in the view definition (uncontrollable, not recommended)
  - Recomputation on every update: immediate update
  - Periodic recomputation (e.g. nightly): deferred update
- Materialized View Update Schemes
  - **Completely recomputed the materialized view**
  - **Incremental view maintenance (modify only the affected part)**
- Modern-day DBMSs have functionality of **incremental view maintenance facility**
    - Avoids manual effort/correctness issues

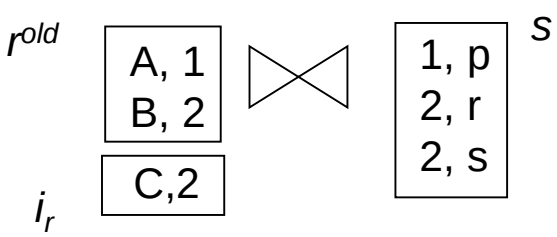
Incremental View Maintenance

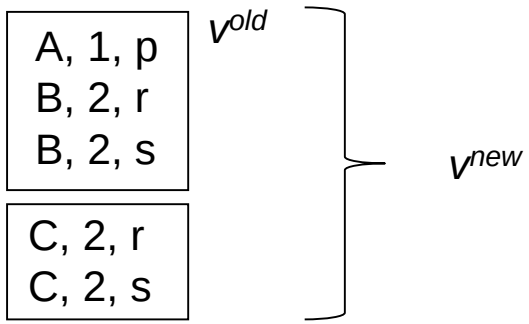
- The changes (inserts and deletes) to a relation or expressions are referred to as its **differential**
  - Set of tuples inserted to and deleted from r are denoted $\mathbf{i_r}$ and $\mathbf{d_r}$
- To simplify our description, we only consider inserts and deletes
  - We replace updates to a tuple 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, in incremental view update
- We then outline how to handle relational algebra expressions in incremental view update

Maintaining Materialized View with Join

- 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 i_r to r:
  - We can write $r^{new} \bowtie s$ as $(r^{old} \cup i_r) \bowtie s$
  - And rewrite the above to $(r^{old} \bowtie s) \cup (i_r \bowtie s)$
  - But $(r^{old} \bowtie s)$ 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 $v^{new} = v^{old} - (d_r \bowtie s)$





Incremental View Maintenance in Materialized View with Join

보조자료

- *r = loan, s = branch*

loan-number	*branch-name*	*amount*
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500

branch-name	*branch-city*	*assets*
Brighton	Brooklyn	7100000
Downtown	Brooklyn	9000000
Mianus	Horseneck	400000
North Town	Rye	3700000
Perryridge	Horseneck	1700000
Pownal	Bennington	300000
Redwood	Palo Alto	2100000
Round Hill	Horseneck	8000000

- The materialized view $v = r \bowtie s$

loan-number	branch-name	amount	branch-city	assets
L-11	Round-Hill	900	Horseneck	8000000
L-14	Downtown	1500	Brooklyn	9000000
L-15	Perryridge	1500	Horseneck	1700000
L-16	Perryridge	1300	Horseneck	1700000
L-17	Downtown	1000	Brooklyn	9000000
L-23	Redwood	2000	Palo Alto	2100000
L-93	Mianus	500	Horseneck	400000

Incremental Deletion in Materialized View with Join

보조자료

- *dr = (L-11, Round Hill, 900)* // *delete a tuple from r*
- *dr* ⋈ *s*

loan-number	branch-name	amount	branch-city	assets
L-11	Round-Hill	900	Horseneck	8000000

- $v_{new} = v_{old} - (dr \bowtie s)$

loan-number	branch-name	amount	branch-city	assets
L-14	Downtown	1500	Brooklyn	9000000
L-15	Perryridge	1500	Horseneck	1700000
L-16	Perryridge	1300	Horseneck	1700000
L-17	Downtown	1000	Brooklyn	9000000
L-23	Redwood	2000	Palo Alto	2100000
L-93	Mianus	500	Horseneck	400000

Incremental Insertion in Materialized View with Join

보조자료

- *ir = (L-11, Round Hill, 900)* *// insert a tuple*
- *ir* ⋈ *s*

loan-number	branch-name	amount	branch-city	assets
L-11	Round-Hill	900	Horseneck	8000000

- $v_{new} = v_{old} \cup (ir \bowtie s)$

loan-number	branch-name	amount	branch-city	assets
L-11	Round-Hill	900	Horseneck	8000000
L-14	Downtown	1500	Brooklyn	9000000
L-15	Perryridge	1500	Horseneck	1700000
L-16	Perryridge	1300	Horseneck	1700000
L-17	Downtown	1000	Brooklyn	9000000
L-23	Redwood	2000	Palo Alto	2100000
L-93	Mianus	500	Horseneck	400000

Maintaining Materialized View with Selection and Projection

- Selection: Consider a view $v = \sigma_\theta(r)$
  - $v^{new} = v^{old} \cup \sigma_\theta(i_r)$ // when a new tuple is inserted into r
  - $v^{new} = v^{old} - \sigma_\theta(d_r)$ // when a tuple is deleted from r

- Projection is a more difficult operation
  - *Suppose 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 $\prod_A(r)$

Maintaining Materialized Views with Aggregation Operation "count" [1/2]

- **count** : $V = {}_{A}g_{count(B)}(r)$
  - 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*

loan

loan-number	*branch-name*	*amount*
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500

branch-name	count
Round Hill	1
Downtown	2
Perryridge	2
Red Wood	1
Mianus	1

$V = $ ***branch-name*** g ***count*** ***(loan-number)*** ***(loan)***

Maintaining Materialized Views with Aggregation Operation "count" [2/2]

- Base relation loan에 tuple (L-30, Round Hill, 1300) 이 삽입된 경우
  - Count: Materialized View Count 에 Round Hill 이 있는가? ➔ Yes
    - Update the tuple (Round Hill, 1) ➔ (Round Hill, 2)

- Base relation loan에 tuple (L-70, Seoul, 1000) 이 삽입된 경우
  - Count: Materialized View Count 에 Seoul 이 있는가? ➔ No
    - Insert a new tuple (Seoul, 1) to Count

- Base relation loan에 tuple (L-11, Round Hill, 900)이 삭제된 경우
  - Count: Attribute 'count' value를 1감소. 0인가?
    - If so, Tuple (Round Hill, 1)을 Count에서 삭제

- Base relation loan에 tuple (L-15, Downtown, 1500) 삭제된 경우
  - Count: Attribute 'count' value를 1감소. 0인가?
    - If not, update (Downtown, 2) ➔ (Downtown, 1)

Maintaining Materialized Views with Aggregation Operation "sum" [1/2]

- **sum**: $V = {}_{A}g_{sum\ (B)}(r)$
  - 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?)

loan

loan-number	*branch-name*	*amount*
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500

branch-name	sum
Round Hill	900
Downtown	2500
Perryridge	2800
Red Wood	2000
Mianus	500

$V = {}_{\textbf{branch-name}}g_{\textbf{sum}(\textbf{amount})}(\textbf{loan})$

Maintaining Materialized Views with Aggregation Operation "sum" [2/2]

- Base relation loan에 tuple (L-30, Round Hill, 1300) 이 삽입된 경우
  - Sum: Materialized View Sum 에 RoundHill 이 있는가? ➔ Yes
    - Update the tuple (RoundHill, 900) ➔ (RoundHill, 2200) // 900+1300

- Base relation loan에 tuple (L-70, Seoul, 1000) 이 삽입된 경우
  - Sum: Materialized View Sum 에 Seoul 이 있는가? ➔ No
    - Insert a new tuple (Seoul, 1000) to Sum

- Base relation loan에 tuple (L-11, Round Hill, 900)이 삭제된 경우
  - Sum: 구룹별 count 에서 1감소. 0인가?
    - If so, Tuple (Round Hill, 900)을 Sum에서 삭제

- Base relation loan에 tuple (L-15, Downtown, 1500) 삭제된 경우
  - Sum: . Group별 count에서 1감소. 0인가?
    - If not, update Tuple (Downtown, 2500) ➔ (Downtown, 1000) // 2500 - 1500

Maintaining Materialized Views with Aggregation Operation "avg, min, max" [1/2]

- To handle the case of **avg**, we maintain the **sum** and **count** aggregate values separately, and divide at the end

loan

loan-number	*branch-name*	*amount*
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500

branch-name	avg
Round Hill	900
Downtown	1250
Perryridge	1400
Red Wood	2000
Mianus	500

$$V = \textit{branch-name}\, g_{\textit{avg}(\textit{amount})}\ (\textit{loan})$$

- **min**, **max**: $v = {}_{A}g_{min\ (B)}\ (r)$.
  - 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

Maintaining Materialized Views with Aggregation Operation "avg, min, max" [2/2]

- Base relation loan에 tuple (L-30, Round Hill, 1300) 이 삽입된 경우
  - Avg: Materialized View Sum 에 RoundHill 이 있는가? ➔ Yes
    - Update the tuple (RoundHill, 900) ➔ (RoundHill, 1100) // (900 + 1300) / 2

- Base relation loan에 tuple (L-70, Seoul, 1000) 이 삽입된 경우
  - Avg: Materialized View Sum 에 Seoul 이 있는가? ➔ No
    - Insert a new tuple (Seoul, 1000) to Sum

- Base relation loan에 tuple (L-11, Round Hill, 900)이 삭제된 경우
  - Avg: 구릅별 count 에서 1감소. 0인가?
    - If so, Tuple (Round Hill, 900)을 Avg에서 삭제

- Base relation loan에 tuple (L-15, Downtown, 1500) 삭제된 경우
  - Avg: Group별 count에서 1감소. 0인가?
    - If not, update Tuple (Downtown, 2500) ➔ (Downtown, 1000) // (2500 – 1500)/ (2 -1)

Maintaining Materialized View with 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 Relational Algebra Expressions in Materialized View Maintenance

- 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 using Materialized Views

- Rewriting queries to use materialized views:
  - A materialized view $v = r \bowtie s$ is available
  - A user submits a query $r \bowtie s \bowtie t$ // the original query
  - We can rewrite the query as $v \bowtie t$ // the rewritten query with v
    - 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 related with materialized views and choose the best overall plan

Materialized View and Index Selection

- **Materialized view selection**: Given a set of relations and a set of views,

  "What is the best set of views to materialize?"

- **Index selection:** Given a set of attributes in a relation,

  "What is the best set of indices to create?"

- Materialized view selection and index selection are closely related
  - 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 (ch 24.1)
- Commercial database systems provide tools called "tuning assistants" or "wizards"
  - to help DBA choose what indices and materialized views to create
  - Microsoft SQL Server Database Tuning Assistant
  - IBM DB2 Design Advisor
  - Oracle SQL Tunning Wiard

Chapter 13: Query Optimization

- 13.1 Overview
- 13.2 Transformation of Relational Expressions
- 13.3 Estimating Statistics of Expression
- 13.4 Choice of Evaluation Plans
- 13.5 Materialized views**
- 13.6 Advanced Topics in Query Optimization**

Top-K Optimization

- **Top-K query**

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

r

A	B	C
..	..	..

s

I	B	F
..	..	..

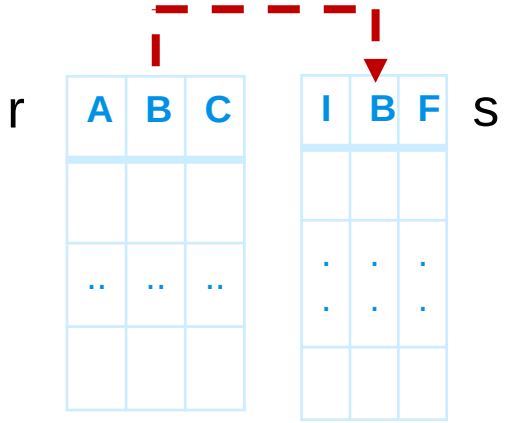
  - Naïve Approach: Generate and sort the entire result, and pick up the top K
  - Alternative 1: Use pipelined plans that can generate the results in sorted order
    - r ⋈ s 에서 tuple생성될때마다 sort로 pipeline (병렬화의 효과)
  - Alternative 2: (r.A ascending ➔ 작은것부터 k개)
    - Estimate the highest r.A value H in result
    - Add a selection predicate (**and** r.A <= H) to where clause
    - If “more than K” results came out, pick K tuples using sort
    - If “less than K” results came out, retry the query with a larger H
    - Sort할 대상물을 줄이는 효과
- Numerous papers on Top-K Query Optimization
  - Carey and Kossman, “Reducing the braking distance of SQL engine”, VLDB1998

Join Minimization

- Check if join and some condition is redundant
  - If join condition is on foreign key from r to s and no selection on s
    - Suppose r.B is a foreign key of s.B

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

    - join with s is redundant and can be dropped



  - Other sufficient conditions can be simplified

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

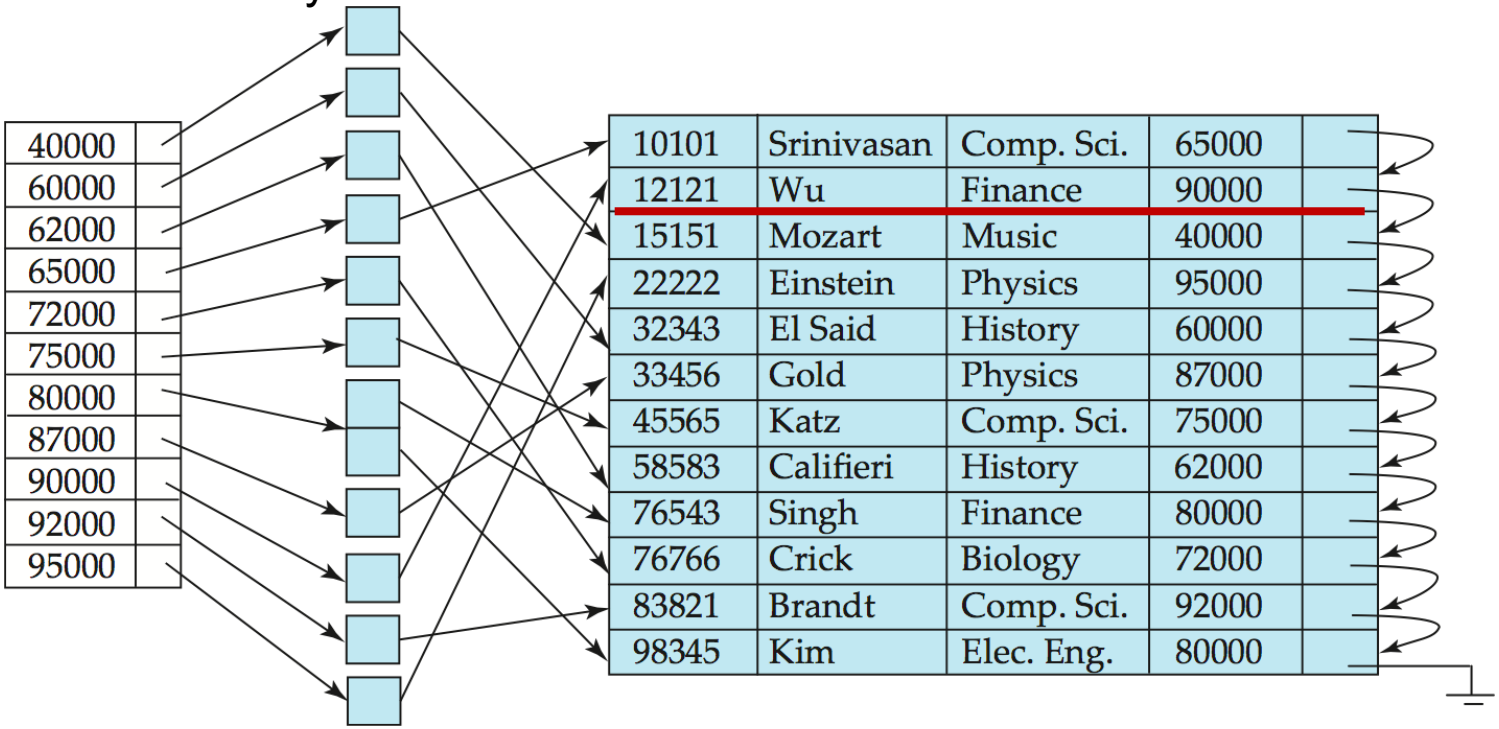
    - join with s2 is redundant and can be dropped (along with selection on s2)

Optimization of Multiple Updates [1/2]

- **Halloween problem**

```
update R set Salary = Salary + Salary * 0.1
where Salary > $80,000
```

  - If index on Salary is used to find tuples satisfying Salary > $80,000, and tuples updated immediately, same tuple may be found (and updated) multiple times
  - 위의 query는 tuple을 한개씩 search & update를 하는 방식으로 수행되는것이 문제
  - Wu' salary: 90000 ➔ 99000 ➔ 108900 ➔



Optimization of Multiple Updates [2/2]

- **update** R **set** Salary = Salary + Salary * 0.1
  where Salary > $80,000
- A naïve solution for Hollween Problem
  - Step1: Collect the updates (old and new values of tuples) into a batch with "Select * from Emp where Salary > $80,000"
  - Step2: Update the batch
    - 2 step processing is expensive
- A better "optimized" solution
  - Perform immediate update in the following situations guaranteeing No Holloween
    - Condition A: if update does not affect attributes in where clause
    - Even violating Condition A, if updates decrease the value and the index is scanned in increasing order
  - Otherwise perform deferred update
    - Naïve solution의 step1과 step2사이에 sorting추가
    - Sort the batch updates in the index order
      - It can reduce random IOs for updating indices

Multiquery Optimization [1/2]

- Multiquery optimization techniques for best overall plan for a set of queries
  - Exploiting sharing of common subexpressions
  - Shared-scan optimization

- Example

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

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

  - Both queries share common subexpression (r ⋈ s)
  - May be useful to compute (r ⋈ s) once and use it in both queries
    - But this may be more expensive in some situations
      - e.g. (r ⋈ s) ⋈ t may be more expensive than (r ⋈ t) ⋈ s

Multiquery Optimization [2/2]

- **Shared scans optimization**
  - Mutiple query들이 a single large relation에 scan을 해야 한다면,
  - 여러 query가 disk로부터 반복적으로 relation을 scan하는 대신,
  - 첫번 query에 대해서 disk로부터 data를 읽고, 다른 query들에게 그 data를 pipelining 형태로 전달하는 방법

- Suppose there are many materialized views
- Maintaining multiple materialized views are similar to multiquery optimization
  - Multiquery optimization techniques can be useful
  - A set of materialized views may share common subexpressions
  - Shared scan optimizations may be useful

Parametric Query Optimization

- Example
  select *
  from r **natural join** s
  where r.a < $1
  - Suppose value of parameter $1 is not known at compile time (known at run time)
  - different plans may be optimal for different values of $1

- Solution 1: **Query-Plan Caching**
  - If optimizer decides that same plan is likely to be optimal for all parameter values, the optimizer caches the query plan and reuses it, else re-optimize each time
  - Query의 constant value에 의해서 영향을 받지 않는다면

- Solution 2: **Parametric Query Optimization**
  - Optimizer generates a set of plans, optimal for different values of $1
  - 변수로 입력 가능한 몇몇 값에 대해서 몇 가지 계획을 생성해 둠
    - The above query: r.a < 10 ➔ r을 scan할때 B+ index 이용, r.a < 1000 ➔ r을 단순 scan
    - Key issue: how to do find a set of optimal plans efficiently
  - Best one from this set is chosen at run time when $1 is known