

# Query Optimization

## CS460 Databases for Data Scientists



# Previously on Database Technology

- **Last time**, we have seen different algorithms for query processing
  - plus their best/worst case cost estimates
- **Today**
  - how to come up with an *efficient* query execution plan

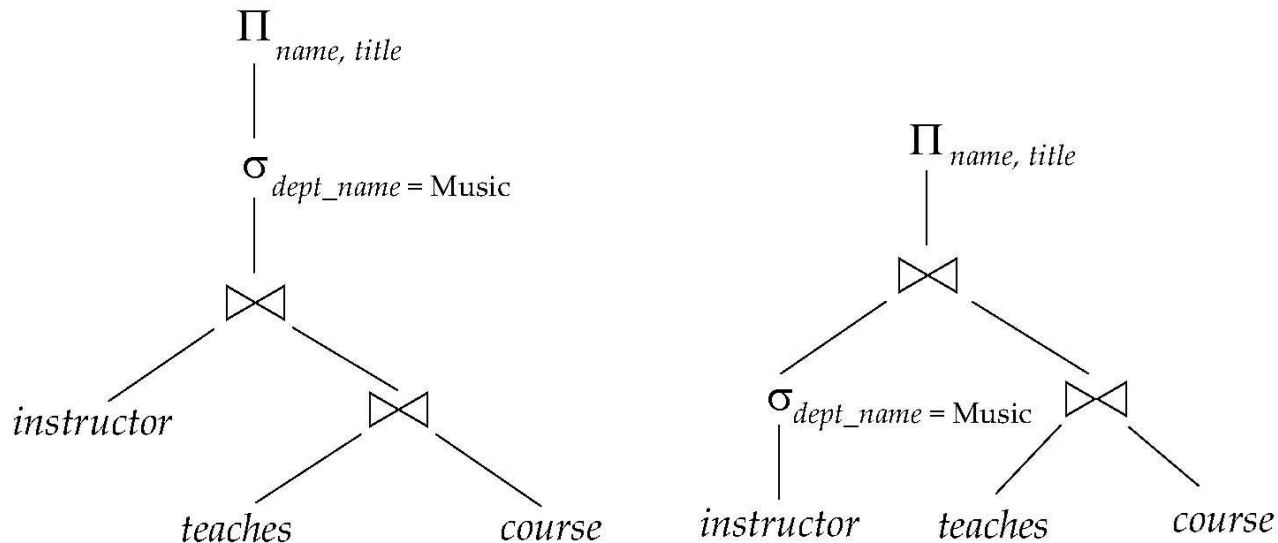


# Agenda

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

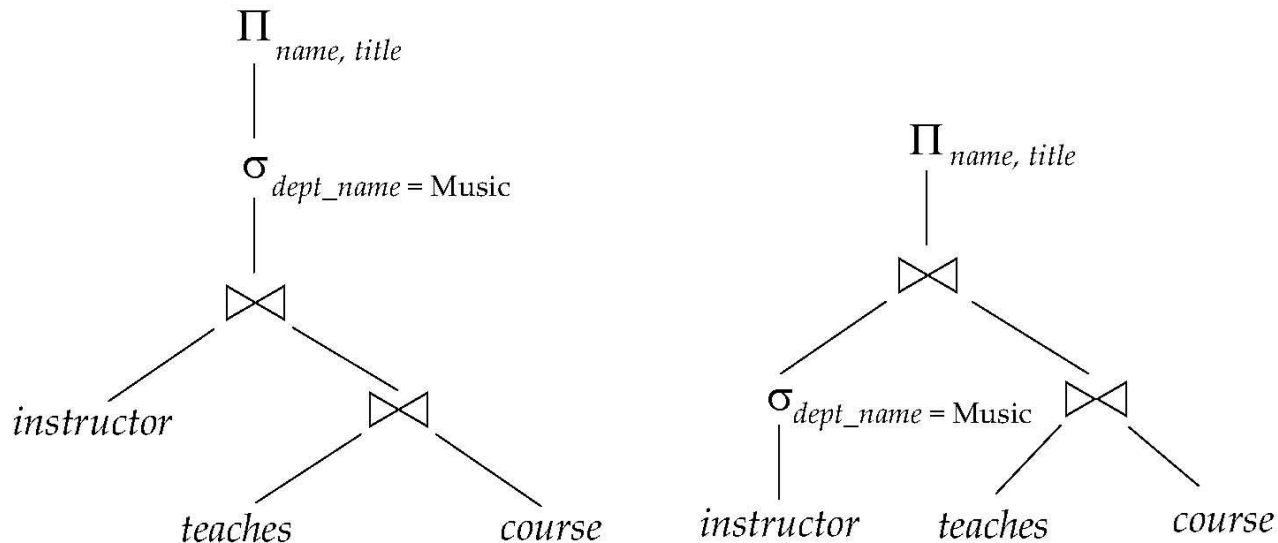
# Introduction

SELECT name, title  
FROM instructor, teaches, course  
WHERE instructor.inst\_ID = teaches.inst\_ID AND  
course.dept\_name = instructor.dept\_name AND  
dept\_name="Music"



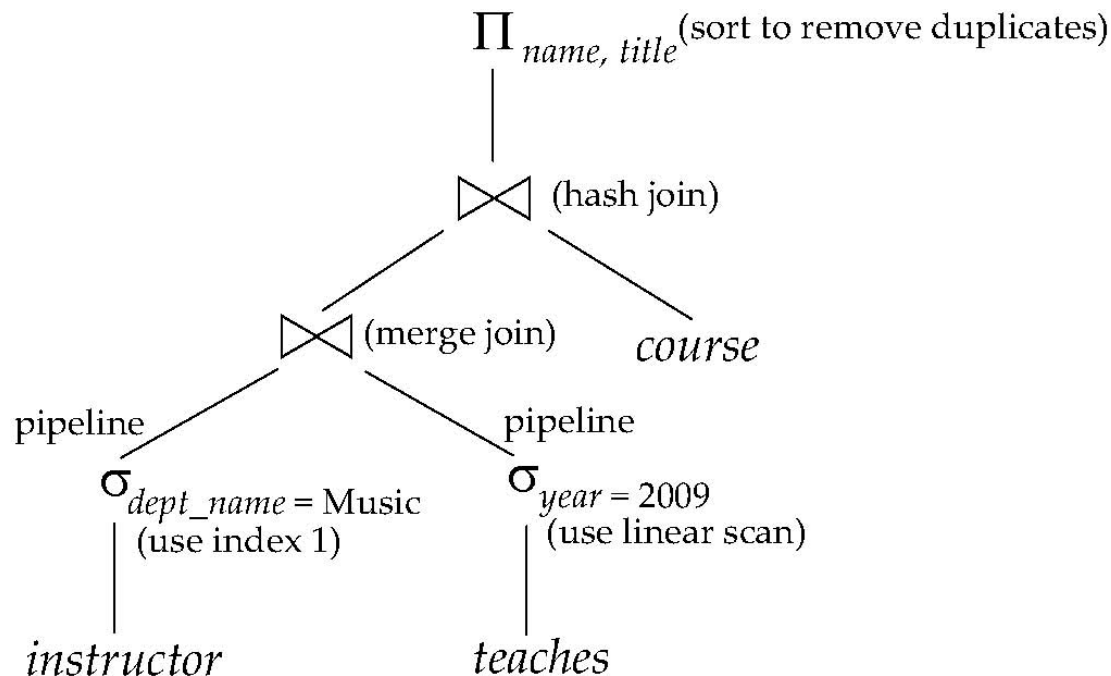
# Introduction

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



# Evaluation Plans

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



# Cost-based Query Optimization

- Cost difference between evaluation plans for a query can be enormous
  - e.g., seconds vs. days in some cases
- Steps in cost-based query optimization
  - Generate logically equivalent expressions using equivalence rules
  - Annotate resulting expressions to get alternative query plans
  - Choose the cheapest plan based on estimated cost
- Estimation of plan cost based on:
  - Statistical information about relations, e.g.:
    - 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

# Equivalence of Relational Algebra 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
  - order of tuples is irrelevant
  - they may yield different results on databases that violate integrity constraints
- Equivalent results must not be a result of chance, e.g.
  - SELECT name FROM employee WHERE id="12345"  
→ "Smith"
  - SELECT name FROM employee WHERE birthday="30.10.1974"  
→ "Smith"
- Those results could be different on a different database instance



# Equivalence of Relational Algebra Expressions

- In SQL, inputs and outputs are multisets of tuples
  - i.e., they may contain duplicates
  - 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
    - for each duplicate result, the same number of duplicates must appear in the results
- An **equivalence rule** says that expressions of two forms are equivalent
  - Can replace expression of first form by second, or vice versa
  - Successive application of equivalence rules generates alternative query formulations

# Equivalence Rules

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

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

```
SELECT name, title  
FROM instructor  
WHERE dept_name="Music" AND  
       salary>50000
```

```
SELECT name,title  
FROM (  
    SELECT name,title  
    FROM instructor  
    WHERE dept_name="Music")  
WHERE salary>50000
```

Note: SQL statements in these slides are solely illustrative!

# Equivalence Rules

2. Selection operations are commutative.

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

```
SELECT name,title
FROM (
  SELECT name,title
  FROM instructor
  WHERE dept_name="Music")
WHERE salary>50000
```

```
SELECT name,title
FROM (
  SELECT name,title
  FROM instructor
  WHERE salary>50000)
WHERE dept_name="Music"
```

# Equivalence Rules

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

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

```
SELECT name, title
FROM (
  SELECT name,title,salary,dept_name
  FROM instructor
  WHERE dept_name="Music" AND
        salary>50000
)
```

```
SELECT name,title
FROM instructor
WHERE dept_name="Music" AND
      salary>50000
```

# Equivalence Rules

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

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

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

```
SELECT name, building
FROM (
  SELECT name, building
  FROM instructor, department
  WHERE instructor_dept_name =
        department.dept_name)
WHERE SALARY > 50000
```

```
SELECT name, building
FROM instructor, department
WHERE instructor.dept_name =
      department.dept_name AND
      salary > 50000
```

# Equivalence Rules

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

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

```
SELECT name, building
FROM instructor, department
WHERE instructor.dept_name =
      department.dept_name AND
      salary > 50000
```

```
SELECT name, building
FROM department, instructor
WHERE department.dept_name =
      instructor.dept_name AND
      salary > 50000
```

# Equivalence Rules (ctd.)

## 6. A) Natural join operations are associative

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

```
SELECT *  
FROM instructor, (  
  SELECT *  
  FROM teaches, course  
  WHERE teaches.course_ID =  
         course.course_ID  
) AS joined  
WHERE instructor.inst_ID = joined.inst_ID
```

```
SELECT *  
FROM course, (  
  SELECT *  
  FROM instructor,teaches  
  WHERE instructor.inst_ID =  
         teaches.inst_ID  
) AS joined  
WHERE course.course_ID = joined.course_ID
```

# Equivalence Rules (ctd.)

6. 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  $\theta_2$  involves attributes from only E2 and E3

```
SELECT *
FROM instructor, (
  SELECT *
  FROM teaches, course
  WHERE teaches.course_ID =
         course.course_ID
) AS joined
WHERE instructor.inst_ID = joined.inst_ID
      AND salary > 50000
```

```
SELECT *
FROM course, (
  SELECT *
  FROM instructor,teaches
  WHERE instructor.inst_ID =
         teaches.inst_ID
) AS joined
WHERE course.course_ID = joined.course_ID
      AND salary > 50000
```



# Example: Join Ordering

- For all relations  $r_1, r_2$ , and  $r_3$ ,

$$(r_1 \bowtie r_2) \bowtie r_3 = r_1 \bowtie (r_2 \bowtie r_3) \quad (\text{Rule 6a})$$

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

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

so that we compute and store a smaller temporary relation

## Example: Join Ordering (ctd.)

- Consider the expression

```
SELECT name, title
FROM instructor, teaches, course
WHERE instructor.inst_ID = teaches.inst_ID AND
      teaches.course_ID = course.course_ID AND
      instructor.dept_name = "Music"
```

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

## Equivalence Rules (ctd.)

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

(a) If all the attributes in  $\theta_0$  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) If  $\theta_1$  involves only the attributes of  $E_1$  and  $\theta_2$  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 (ctd.)

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

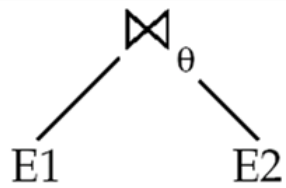
(a) If all the attributes in  $\theta_0$  involve only the attributes of one of the expressions ( $E_1$ ) being joined

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

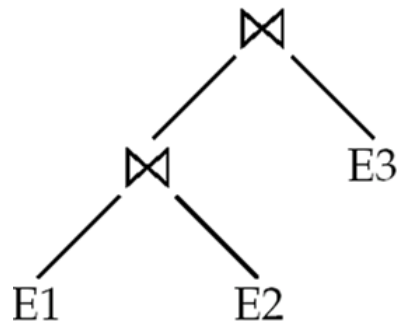
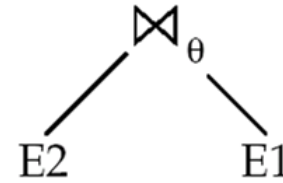
```
SELECT *  
FROM instructor, department  
WHERE instructor.dept_name =  
       department.dept_name AND  
       salary>50000
```

```
SELECT *  
FROM  
  (SELECT * FROM instructor WHERE salary>50000)  
  AS R1,  
  (SELECT * FROM department) AS R2  
WHERE R1.dept_name = R2.dept_name
```

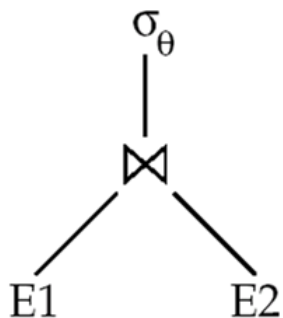
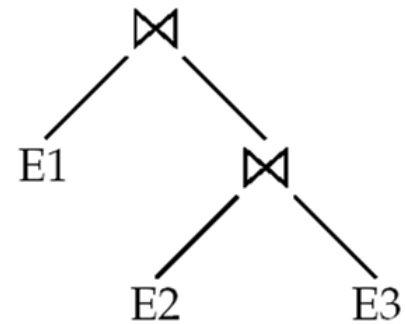
# Equivalence Rules: Graphical Visualization



Rule 5

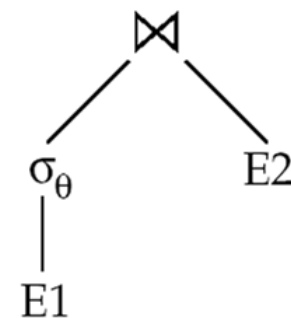


Rule 6a



Rule 7a

If  $\theta$  only has  
attributes from E1



# Example: Pushing Selection

- 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

Note the closing brackets

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

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

# Example with Multiple Expressions

- 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

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

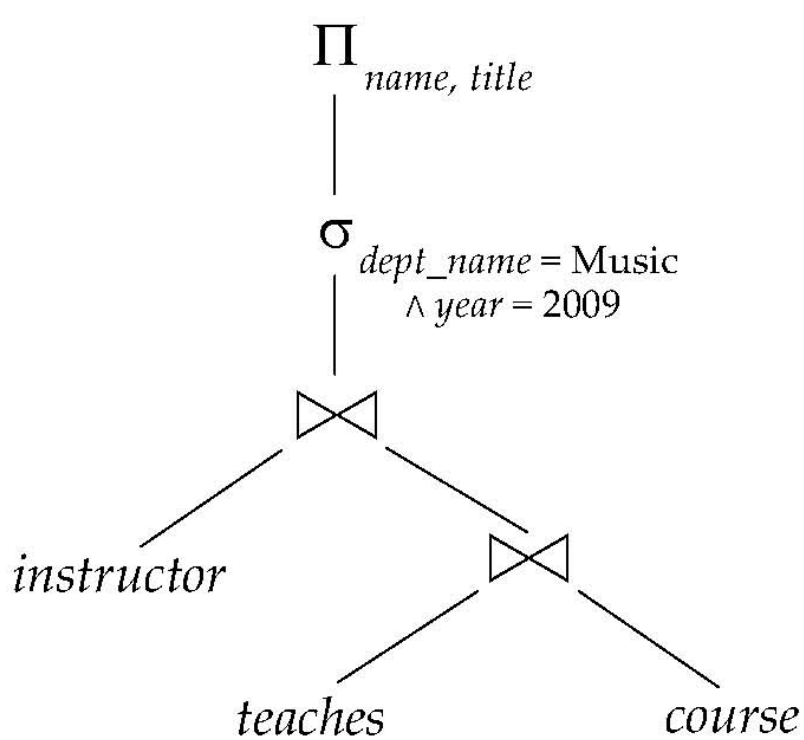
- Transformation using join associatively (Rule 6a):

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

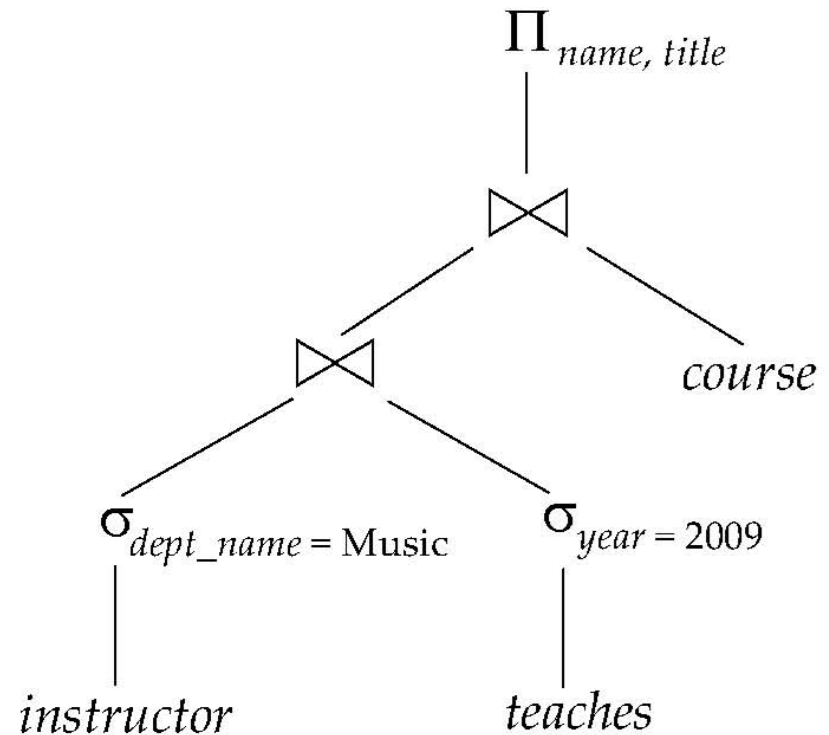
- Transformed expression provides an opportunity to apply the “perform selections early” rule (7b), resulting in

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

# Example with Multiple Expressions



(a) Initial expression tree



(b) Tree after multiple transformations



## Equivalence Rules (ctd.)

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

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

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

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

- Let  $L_1$  and  $L_2$  be sets of attributes from  $E_1$  and  $E_2$ , respectively
- Let  $L_3$  be attributes of  $E_1$  that are involved in join condition  $\theta$ , but are not in  $L_1 \cup L_2$ , and
- Let  $L_4$  be attributes of  $E_2$  that are involved in join condition  $\theta$ , but are not in  $L_1 \cup L_2$

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

# Equivalence Rules (ctd.)

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

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

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

$L_3$  contains instructor.dept\_name  
 $L_4$  contains department.dept\_name

```
SELECT name,salary,building
FROM instructor, department
WHERE instructor.dept_name =
      department.dept_name
```

```
SELECT name,salary,building
FROM
  (SELECT name,salary,dept_name FROM instructor) AS R1,
  (SELECT building,dept_name FROM department) AS R2
WHERE R1.dept_name = R2.dept_name
```

# Example: Pushing Projections

- Consider the query:

```
SELECT name, title
FROM instructor, teaches, course
WHERE instructor.inst_ID = teaches.inst_ID AND
      teaches.course_ID = course.course_ID AND
      instructor.dept_name = "Music"
```

- If we compute

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

we obtain a relation whose schema is:

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

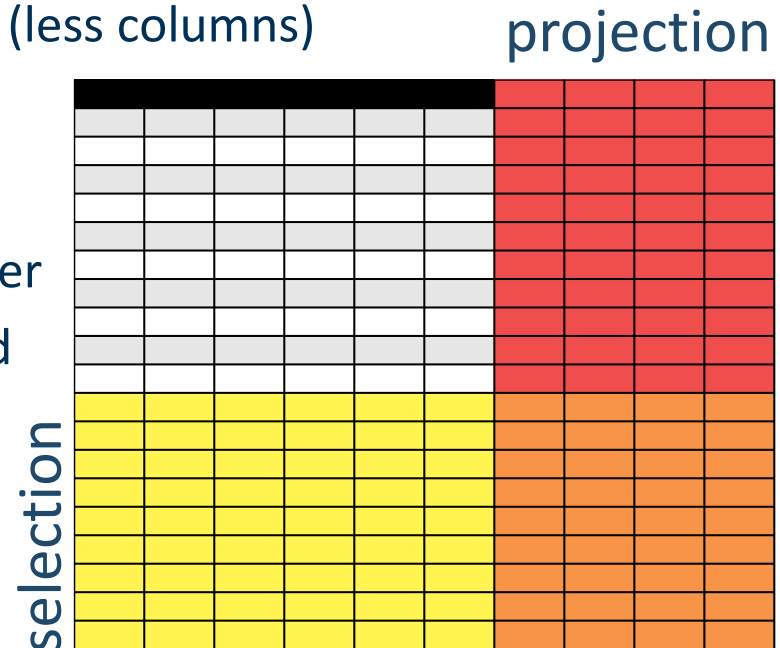
- Push projections using equivalence rules 8a and 8b;  
eliminate unneeded attributes from intermediate results to get:

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

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

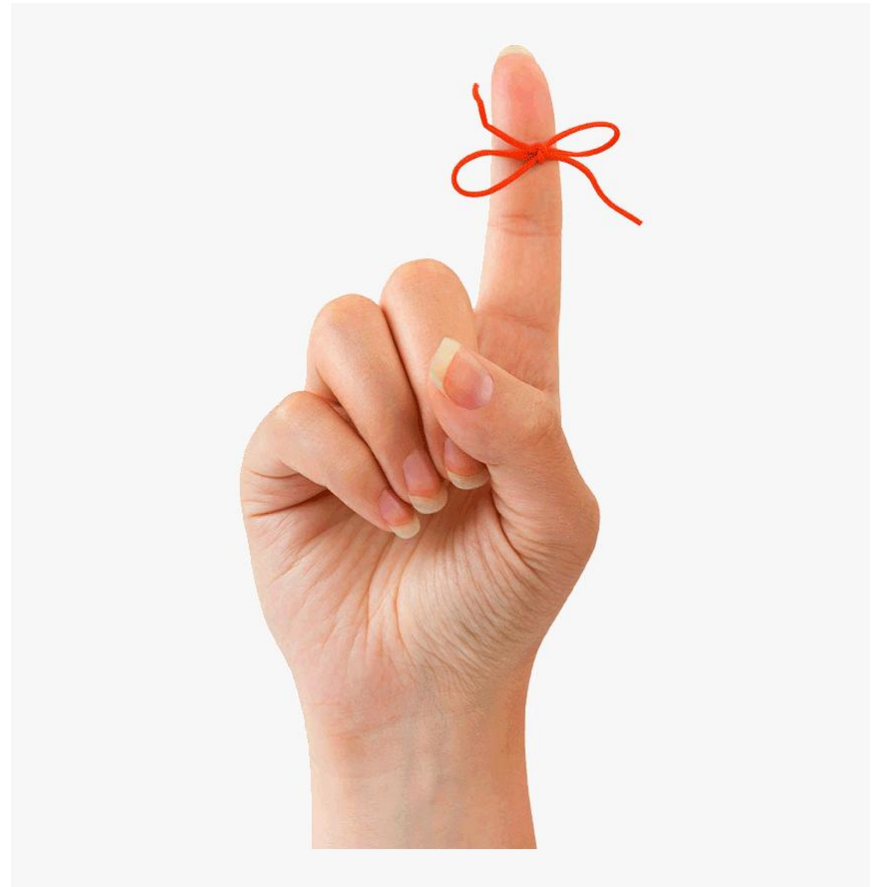
# Pushing Selection and Projection

- Pushing selection to earlier steps
  - Leads to joining *shorter* tables (less rows)
- Pushing projection to earlier steps
  - Leads to joining *narrower* tables (less columns)
- In each case
  - Make intermediate results smaller
  - Reduce amount of cache needed
  - Make subsequent steps faster



# From Equivalence Rules to Query Plans

- Equivalence rules
  - Find alternative query plans
- Query optimization
  - Pick an optimal query plan
  - (or at least a good one)

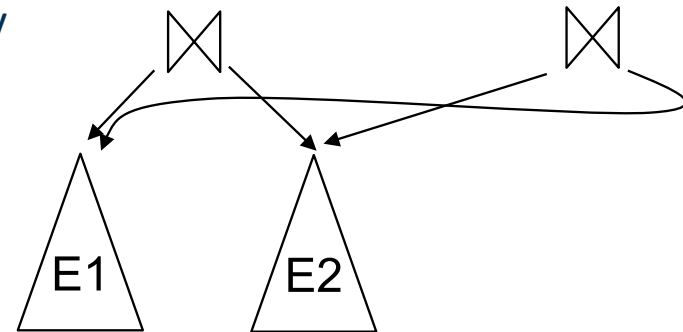


# Enumeration of Equivalent Expressions

- 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

# Transformation based Optimization

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



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

# Cost Estimation for Execution Plans

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



# Cost-based Optimization

- Consider finding the best join ordering for  $r_1 \bowtie r_2 \bowtie \dots \bowtie r_n$
- Join is commutative and associative. For  $n=3$ , we have
  - $(r_1, r_2), r_3 ; (r_2, r_1), r_3 ; r_3, (r_1, r_2) ; r_3, (r_2, r_1) ;$   
 $(r_1, r_3), r_2 ; (r_3, r_1), r_2 ; r_2, (r_1, r_3) ; r_2, (r_3, r_1) ;$   
 $(r_3, r_2), r_1 ; (r_2, r_3), r_1 ; r_1, (r_3, r_2) ; r_1, (r_2, r_3) .$
- In general, the number is very large
  - Mathematically:  $\frac{(2(n-1))!}{(n-1)!}$
- No need to generate all the join orders
  - Dynamic programming:  
compute least-cost join order for any subset of  $\{r_1, r_2, \dots, r_n\}$
  - reduces complexity to  $O(3^n)$

Note: factorial complexity ( $O(n!)$ ) is even worse than exponential!

$n = 5 \rightarrow 1,680$

$n = 10 \rightarrow >17 \text{ billion!}$

# Choosing a Good Execution Plan

- Naively: for each operation, pick the cheapest algorithm
  - given the statistics
  - caution: may not yield best overall algorithm!
- **Example 1:** merge-join may be costlier than hash-join
  - but may provide a sorted output  
which reduces the cost for an outer level aggregation
- **Example 2:** nested-loop join may be a costly variant
  - but provides opportunity for pipelining
- Practical query optimizers incorporate elements of the following two broad approaches
  - Search all the plans and choose the best plan in a cost-based fashion
  - Uses heuristics to choose a plan

# Interesting Sort Orders

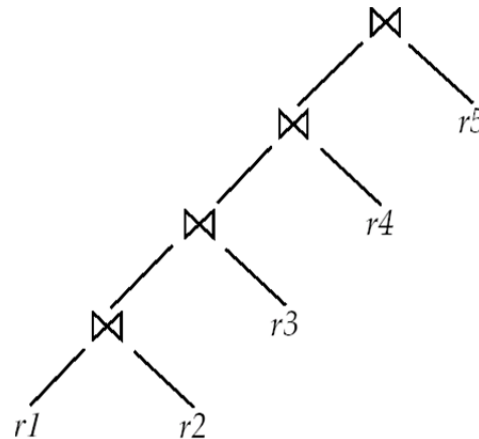
- Consider the expression  $(r_1 \bowtie r_2) \bowtie r_3$  (with A as common attribute)
- An **interesting sort order** is a particular sort order of tuples that could be useful for a later operation
  - Using merge-join to compute  $r_1 \bowtie r_2$  may be costlier than hash join
    - but generates result sorted 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 result ordering and aggregation
- 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**
    - extension of dynamic programming algorithms
  - Usually, number of interesting orders is quite small
    - does not affect time/space complexity significantly

# Heuristic Optimization

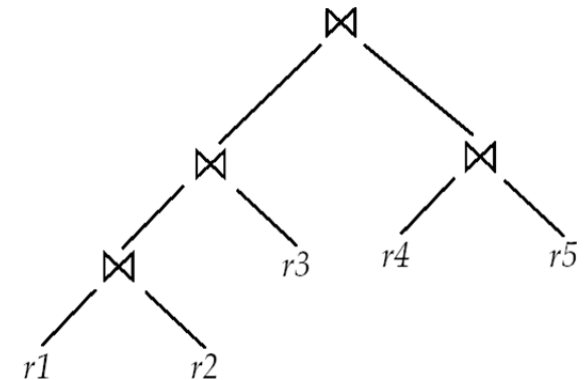
- Cost-based optimization is expensive, even with dynamic programming
- Alternative: use *heuristics* to reduce the number of choices that must be made in a cost-based fashion
  - may miss the *best* solution, but yields a *good* solution
- Heuristic optimization transforms the query tree by using a set of rules that typically 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

# Practical Query Optimizers

- Many optimizers consider 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
- Intricacies of SQL complicate query optimization
  - e.g. nested subqueries



(a) Left-deep join tree



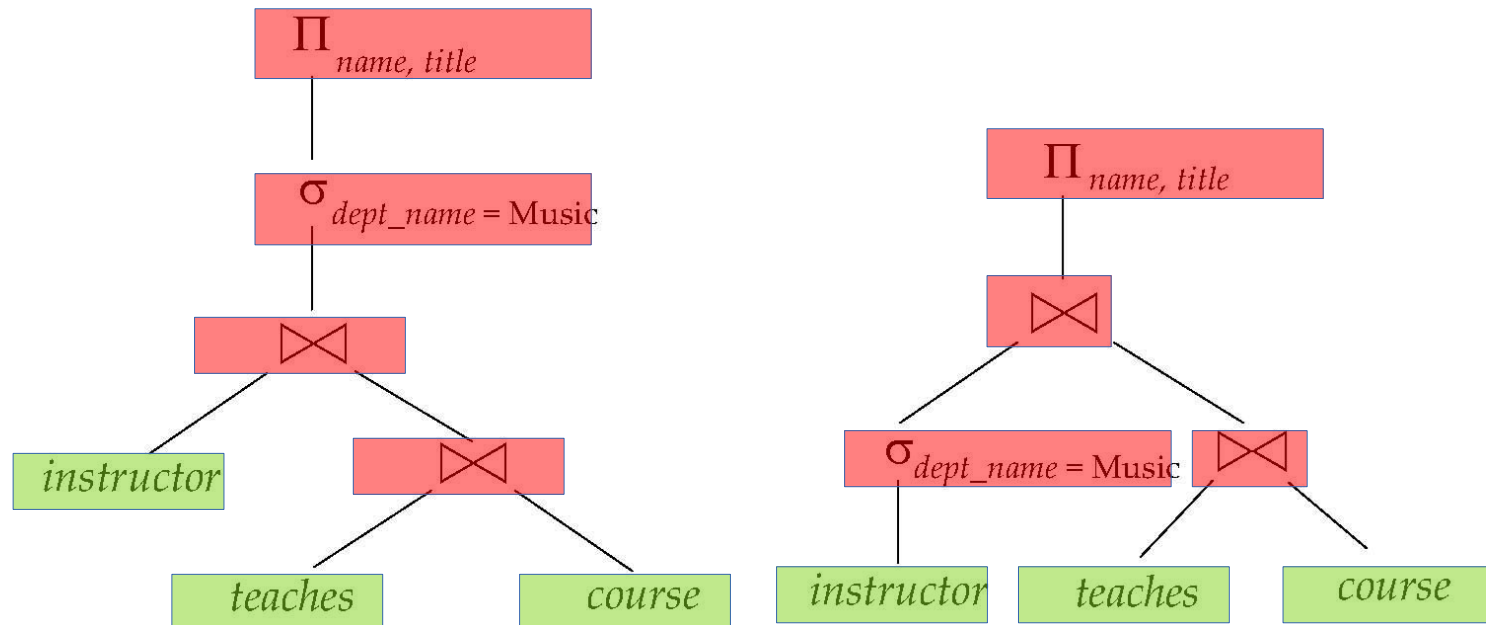
(b) Non-left-deep join tree

# Practical Query Optimizers

- Savings vs. overhead
  - Large search space can lead to severe overhead
- Mixed approach: heuristics for cheap queries,  
exhaustive search for expensive query
- Strategies of practical optimizers (e.g., MS SQL Server) include
  - Optimization cost budget to stop optimization early
    - e.g.: found a plan with cost less than cost of optimization
  - Plan caching to reuse previously computed plan  
if query is resubmitted
    - Even with different constants in query

# Cost Estimation

- We know the size of input tables
  - Not those of intermediate results
  - However, they might be interesting for the optimization
  - Hence, we have to *estimate* them



# Statistical Information for Cost Estimation

- Important numbers:
  - $n_r$ : number of tuples in a relation  $r$
  - $V(A, r)$ : number of **distinct values** that appear in  $r$  for attribute  $A$ ; same as the size of  $\Pi_A(r)$

- Running example: student  $\bowtie$  takes

- $n_{\text{student}} = 5,000$
- $n_{\text{takes}} = 10,000$
- $V(\text{ID}, \text{takes}) = 2500$



- on avg., each student who has taken a course has taken four courses
- Attribute student\_id in takes is a foreign key referencing student
- $V(\text{ID}, \text{student}) = 5000$ 
  - Because ID in student relation is a primary key



# Selection Size Estimates

- Given a selection criterion,  
 $c$  is the estimated number of matching tuples
- $\sigma_{A=v}(r)$ 
  - if  $A$  is a key attribute:  $c = 1$
  - if  $A$  is a non-key attribute:  $c = \frac{n_r}{V(A,r)}$
- $\sigma_{A \leq v}(r)$  (case of  $\sigma_{A \geq v}(r)$  is symmetric)
  - In absence of statistical information:  $c = \frac{n_r}{2}$
  - If  $\min(A,r)$  and  $\max(A,r)$  are available in catalog
    - $c = 0$  if  $v < \min(A,r)$
    - $c = n_r * \frac{v - \min(A,r)}{\max(A,r) - \min(A,r)}$
  - Further refinement possible using histograms

# Example for Selection Size Estimation

- Estimate for temperature  $\leq 12$

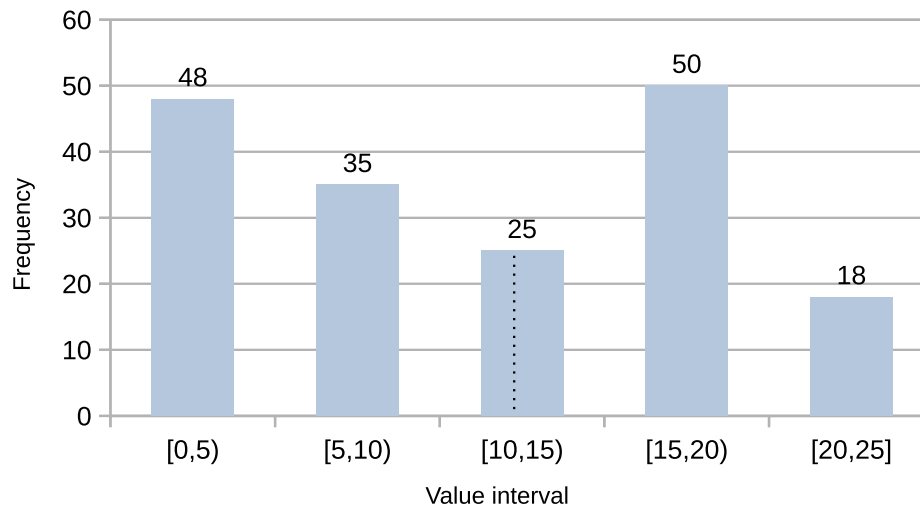
- without statistics ( $n_r = 176$ ):  $c = \frac{n_r}{2} = 88$

- with  $\text{min}=0$ ,  $\text{max}=25$ :  $c = n_r * \frac{v - \text{min}(A,r)}{\text{max}(A,r) - \text{min}(A,r)} = 84.5$

- using histogram:  $c = 48 + 35 + 25 * 2/5 = 93$

assumes uniform distribution

assumes uniform distribution within histogram bars



# Size Estimation for Complex Selections

- $s_i$  is the number of satisfying tuples of a condition  $\theta_i$  in  $r$

- **Conjunction:**  $\sigma_{\theta_1 \wedge \theta_2 \wedge \dots \wedge \theta_n}(r)$

*Assuming independence*, estimate of tuples in the result is:

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

Number of conjunctions

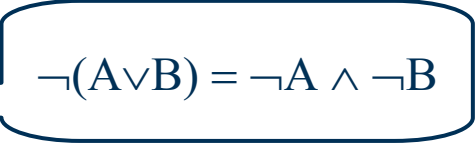
- **Negation:**  $\sigma_{\neg\theta}(r)$ .

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

# Size Estimation for Complex Selections

- **Disjunction:**  $\sigma_{\theta_1 \vee \theta_2 \vee \dots \vee \theta_n}(r)$ .
- Let's use negation and disjunction:

$$- \sigma_{\theta_1 \vee \theta_2 \vee \dots \vee \theta_n}(r) = \sigma_{\neg(\neg\theta_1 \wedge \neg\theta_2 \wedge \dots \wedge \neg\theta_n)}(r)$$


$$\neg(A \vee B) = \neg A \wedge \neg B$$

$$= n_r - (n_r - s_1) * (n_r - s_2) * \dots * (n_r - s_n)$$

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

# Estimating the Size of Joins

- Let  $r(R)$  and  $s(S)$  be relations ( $R$  and  $S$  are the attributes)
- The Cartesian product  $r \times s$  contains  $n_r * n_s$  tuples
  - each tuple occupies  $s_r + s_s$  bytes.
- If  $R \cap S = \emptyset$ , then  $r \bowtie s$  is the same as  $r \times s$
- If  $R \cap S$  is a key for  $r$ , then a tuple of  $s$  will join with at most one tuple from  $r$ 
  - therefore, the number of tuples in  $r \bowtie s$  is no greater than the number of tuples in  $s$
- If  $R \cap S$  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  $student \bowtie takes$ ,  $student\_id$  in  $takes$  is a foreign key referencing  $student$
  - hence, the result has exactly  $n_{takes}$  tuples, which is 10,000

# Estimating the Size of Joins

- If  $R \cap S = \{A\}$  is *not* a key for  $r$  or  $s$   
If we assume that every tuple 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,  
i.e., we ultimately use

$$\min\left(\frac{n_r * n_s}{V(A, s)}, \frac{n_r * n_s}{V(A, r)}\right)$$

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

# Estimating the Size of Joins

- If  $R \cap S = \{A\}$  is not a key for  $r$  or  $s$
- *Example:*  
computing student  $\bowtie$  takes without join information

$$\frac{n_{student} * n_{takes}}{V(ID, takes)} = \frac{5,000 * 10,000}{2,500} = 20,000$$

$$\frac{n_{student} * n_{takes}}{V(ID, student)} = \frac{5,000 * 10,000}{5,000} = 10,000$$

→ The minimum of the two is 10,000

# Estimating the Size of Joins

- Left/right outer join:
  - Estimated size of  $r \bowtie s = \text{size of } r \bowtie s + \text{size of } r$
  - Case of right outer join is symmetric
- Full outer join:
  - Estimated size of  $r \bowtie s = \text{size of } r \bowtie s + \text{size of } r + \text{size of } s$
- *Note:* These are pessimistic estimates
  - i.e. upper bounds
- In our example: not all students have to take courses
  - hence, computing student  $\bowtie$  takes make sense
- Estimated upper bound:  $10,000 + 5,000 = 15,000$



# Size Estimation for Projection and Aggregation

- Projection: estimated size of  $\Pi_A(r) = V(A,r)$ 
  - Example: find all students that have taken courses
  - $\Pi_{student\_id}(takes) = V(student\_id, takes) = 2,500$
- Aggregation : estimated size of  ${}_A g_F(r) = V(A,r)$ 
  - Example: compute average courses taken by students
  - For each student, an average is computed
    - Hence:  ${}_{student\_id} g_{avg}(takes) = 2,500$

# Estimating the Number of Distinct Values

Selections:  $\sigma_{\theta}(r)$

- If  $\theta$  forces  $A$  to take a specified value:  $V(A, \sigma_{\theta}(r)) = 1$ .
  - e.g.,  $A = 3$
- If  $\theta$  forces  $A$  to take on one of a specified set of values:  
 $V(A, \sigma_{\theta}(r)) = \text{number of specified values}$ .
  - e.g.,  $(A = 1 \vee A = 3 \vee A = 4)$
- If the selection condition  $\theta$  is of the form  $A < r$   
estimated  $V(A, \sigma_{\theta}(r)) = V(A.r) * s$ 
  - where  $s$  is the selectivity of the selection
- In all 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

analogous for  
>, ≤, ≥

# Estimating the Number of Distinct Values

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  
$$V(A, r \bowtie s) = \min(\begin{aligned} &V(A1,r)*V(A2 - A1,s), \\ &V(A1 - A2,r)*V(A2,s), \\ &n_{r \bowtie s} \end{aligned})$$
- Again:
  - more accurate estimate can be got using probability theory
  - but this one works fine generally

# Optimizing Nested Subqueries

- In the part about SQL, we have learned about nested subqueries
  - A useful tool, but can lead to complex & expensive queries
- Consider:
  - **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
  - Parameters are variables from outer level query, 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**

# Optimizing Nested Subqueries

- Correlated evaluation may be quite inefficient since
  - a large number of calls may be made to the nested query
  - may lead to many additional I/O operations (block seek/transfer) 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

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

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

```
CREATE TABLE  $t_1$  AS
  SELECT DISTINCT ID
  FROM teaches
  WHERE year = 2007

SELECT name
FROM instructor,  $t_1$ 
WHERE  $t_1.ID = instructor.ID$ 
```
- Replacing a nested query by a query with a join (possibly with a temporary relation) is called **decorrelation**.
- Decorrelation is more complicated if
  - the nested subquery uses aggregation
  - the result of the nested subquery is used to test for equality / comparison
  - the condition linking the nested subquery to the other query is **not exists**
  - ...

# A Note on Subqueries

- In the part about SQL, we have learned that there's many variants

```
select distinct course_id  
from section  
where semester = 'Fall' and year = 2009 and  
course_id in (select course_id from section  
where semester = 'Spring' and year = 2010);
```

- vs.

```
select course_id  
from section as S  
where semester = 'Fall' and year = 2009 and  
exists (select *  
from section as T  
where semester = 'Spring' and year = 2010  
and S.course_id = T.course_id);
```

- vs.

```
select course_id  
from section as s1, section as s2  
where s1.semester = 'Fall' and s1.year = 2009  
and s2.semester = 'Fall' and s2.year = 2009 and s1.course_id = s2.course_id
```



# A Note on Subqueries

- For the RDBMS, joins are easier to optimize than subqueries
- Details may differ from RDBMS to RDBMS
- Rule of thumb:
  - if in doubt, use a join rather than a subquery
- That they are equivalent does not mean that they have the same performance!

# Further Approaches in Join Optimization

- Identifying redundant joins
  - e.g., tables from which no attributes are selected
- Special approaches for top-k queries
  - Very often, the top k results are interesting (e.g., for paging)
- Optimizing across queries
  - e.g., deep optimization for frequently used (sub)queries
    - identifying frequent (sub)queries in logs
  - e.g., parametric queries
    - queries with a variable, e.g., read data for student \$name\$

# Summary

- Queries can be expressed in multiple forms
  - equivalent in terms of results
  - but different in terms of performance
- Query Optimization
  - pick best execution plan
  - estimate time/memory consumption for an execution plan
  - based on statistical information
- A widely researched area
  - e.g., exploiting advanced statistics about datasets
  - e.g., exploiting log files and histories
  - etc.

# Questions?

