CS 5/7330

Query Optimization

Query Optimization

- Step of query optimization
- Given an SQL query
  1. Convert it to (enhanced) relational algebra
  2. Build multiple parse trees
     - Note: Multiple ways to decorate a tree
     - Each decorated tree is also called a ***plan***
  3. Calculate the cost of each plan
  4. Pick the best one

Convert to relational algebra

- In a SELECT …. FROM …. WHERE …. SQL query
  - SELECT … corresponds to π
  - WHERE … correspond to σ
  - FROM … correspond to $\bowtie$ (if corresponding WHERE course exists), $\times$ if not
  - UNION, INTERSECT, EXCEPT correspond to $\cup$, $\cap$, -
  - GROUP BY, HAVING and aggregate does not appear in relational algebra, so have extra operators added to accommodate it.
  - We'll deal with nested query later

Convert to relational algebra

- From the relational algebra, one can build a parse tree
- Parse tree specifies the order of execution of the opeartions
  - Post-order traversal of the tree
- Notice that, for joins, we put the “outer loop” (if we want to do nested loop) on the left

Convert to relational algebra

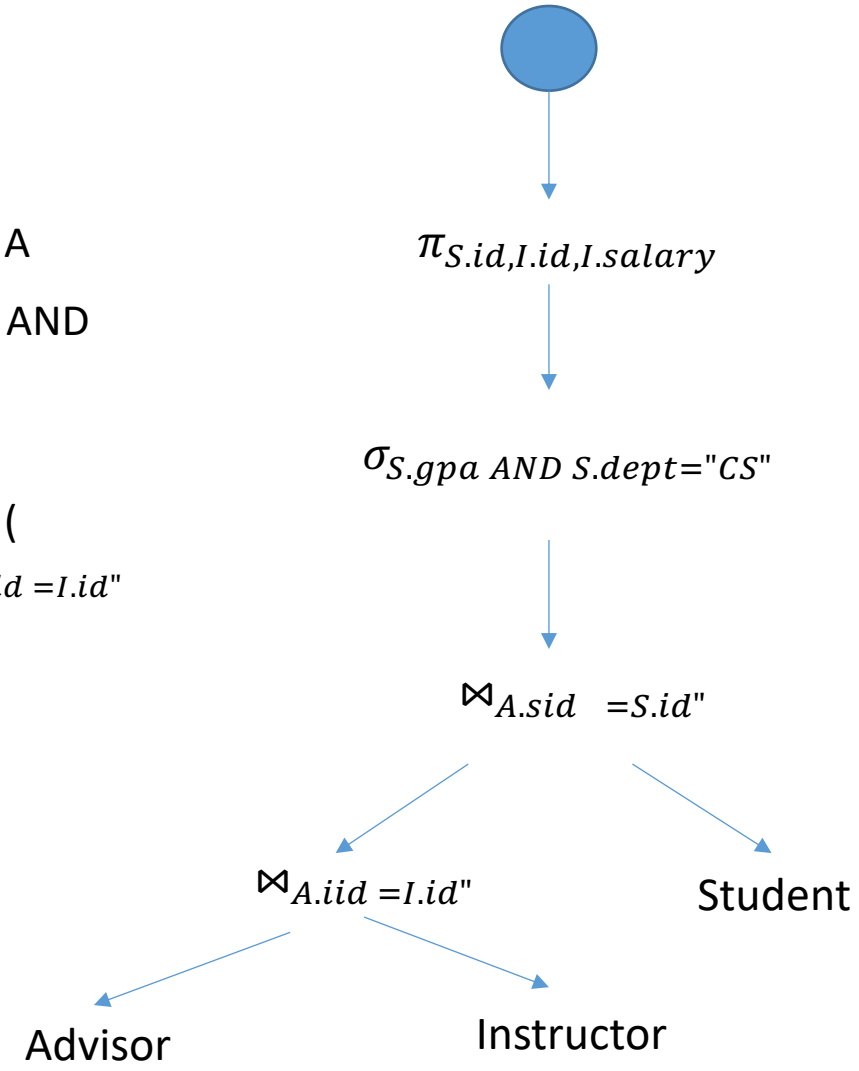
- Example

SELECT S.id, I.id, I.salary

FROM Student S, Instructor I, Advise A

WHERE S.id = A.s_id AND I.id = A.i_id AND S.dept="CS" AND S.gpa >= 3.5

$\pi_{S.id,I.id,I.salary}(\sigma_{S.gpa\ AND\ S.dept="CS"}($
$Student \bowtie_{A.sid\ =S.id"}\ Advisor \bowtie_{A.iid\ =I.id"}$
))



Generate multiple parse trees

- However, there are multiple options of parse trees
- For relational algebra
  - Joins are associative
  - Selection are commutative
  - Same as projection
  - Also there are some distributive properties

Transformation of Relational Expressions

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

Equivalence Rules

1. Conjunctive selection operations can be deconstructed into a sequence of individual selections.
   $$\sigma_{\theta_1 \wedge \theta_2}(E) \equiv \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$
2. Selection operations are commutative.
   $$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) \equiv \sigma_{\theta_2}(\sigma_{\theta_1}(E))$$
3. Only the last in a sequence of projection operations is needed, the others can be omitted.
   $$\Pi_{L_1}(\Pi_{L_2}(\ldots(\Pi_{L_n}(E))\ldots)) \equiv \Pi_{L_1}(E)$$
   where $L_1 \subseteq L_2 \ldots \subseteq L_n$
4. Selections can be combined with Cartesian products and theta joins.
   a. $\sigma_{\theta}(E_1 \times E_2) \equiv E_1 \bowtie_{\theta} E_2$
   b. $\sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) \equiv E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$

Equivalence Rules (Cont.)

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

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

6. (a) Natural join operations are associative:

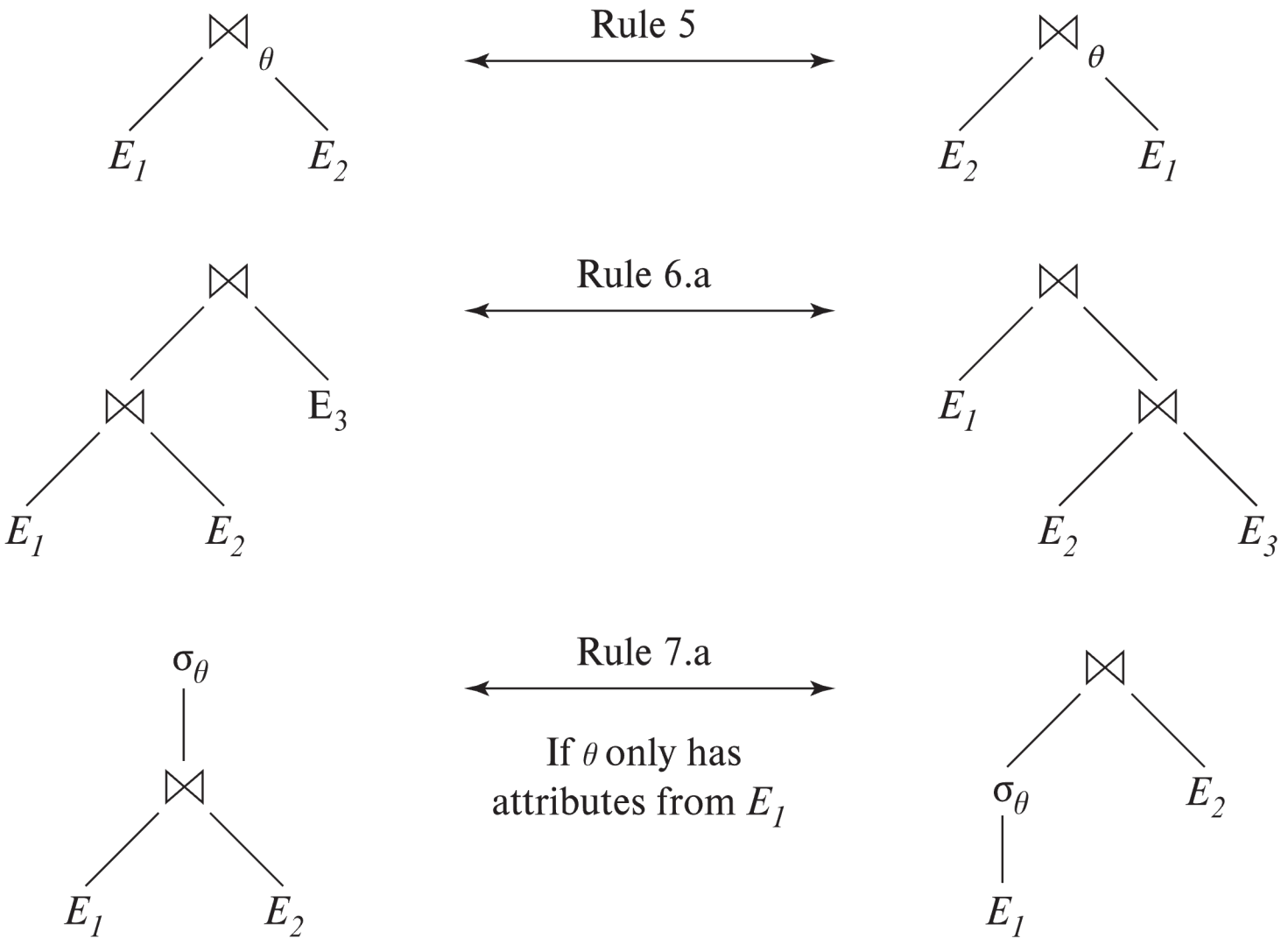
$$(E_1 \bowtie E_2) \bowtie E_3 \equiv E_1 \bowtie (E_2 \bowtie E_3)$$

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

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

where θ_2 involves attributes from only E_2 and E_3.

Pictorial Depiction of Equivalence Rules



Equivalence Rules (Cont.)

7. The selection operation distributes over the theta join operation under the following two conditions:
   (a) When all the attributes in θ_0 involve only the attributes of one
   of the expressions (E_1) being joined.

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

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

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

Equivalence Rules (Cont.)

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

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

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

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

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

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

   Similar equivalences hold for outerjoin operations: $\overset{\frown}{\bowtie}$, $\overset{\frown}{\bowtie}$, and $\overset{\frown}{\bowtie}$

Equivalence Rules (Cont.)

9. The set operations union and intersection are commutative
   $E_1 \cup E_2 \equiv E_2 \cup E_1$
   $E_1 \cap E_2 \equiv E_2 \cap E_1$
   (set difference is not commutative).
10. Set union and intersection are associative.
    $(E_1 \cup E_2) \cup E_3 \equiv E_1 \cup (E_2 \cup E_3)$
    $(E_1 \cap E_2) \cap E_3 \equiv E_1 \cap (E_2 \cap E_3)$
11. The selection operation distributes over $\cup$, $\cap$ and $-$.
    a. $\sigma_\theta (E_1 \cup E_2) \equiv \sigma_\theta (E_1) \cup \sigma_\theta (E_2)$
    b. $\sigma_\theta (E_1 \cap E_2) \equiv \sigma_\theta (E_1) \cap \sigma_\theta (E_2)$
    c. $\sigma_\theta (E_1 - E_2) \equiv \sigma_\theta (E_1) - \sigma_\theta (E_2)$
    d. $\sigma_\theta (E_1 \cap E_2) \equiv \sigma_\theta (E_1) \cap E_2$
    e. $\sigma_\theta (E_1 - E_2) \equiv \sigma_\theta (E_1) - E_2$
    preceding equivalence does not hold for $\cup$
12. The projection operation distributes over union
    $\Pi_L(E_1 \cup E_2) \equiv (\Pi_L(E_1)) \cup (\Pi_L(E_2))$

Equivalence Rules (Cont.)

13. Selection distributes over aggregation as below
    $\sigma_\theta({}_G\gamma_A(E)) \equiv {}_G\gamma_A(\sigma_\theta(E))$
    provided θ only involves attributes in G
14. a. Full outerjoin is commutative:
    $E_1 \mathbin{⟗} E_2 \equiv E_2 \mathbin{⟗} E_1$
    b. Left and right outerjoin are not commutative, but:
    $E_1 \mathbin{⟕} E_2 \equiv E_2 \mathbin{⟖} E_1$
15. Selection distributes over left and right outerjoins as below, provided θ_1
    only involves attributes of E_1
    a. $\sigma_{\theta_1}(E_1 \mathbin{⟕}_\theta E_2) \equiv (\sigma_{\theta_1}(E_1)) \mathbin{⟕}_\theta E_2$
    b. $\sigma_{\theta_1}(E_1 \mathbin{⟖}_\theta E_2) \equiv E_2 \mathbin{⟖}_\theta (\sigma_{\theta_1}(E_1))$
16. Outerjoins can be replaced by inner joins under some conditions
    a. $\sigma_{\theta_1}(E_1 \mathbin{⟕}_\theta E_2) \equiv \sigma_{\theta_1}(E_1 \bowtie_\theta E_2)$
    b. $\sigma_{\theta_1}(E_1 \mathbin{⟖}_\theta E_1) \equiv \sigma_{\theta_1}(E_1 \bowtie_\theta E_2)$
    provided θ_1 is null rejecting on E_2

Equivalence Rules (Cont.)

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

- $\sigma_{\text{year=2017}}(\textit{instructor} ⟕ \textit{teaches}) \not\equiv \sigma_{\text{year=2017}}(\textit{instructor} ⋈ \textit{teaches})$
- Outerjoins are not associative

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

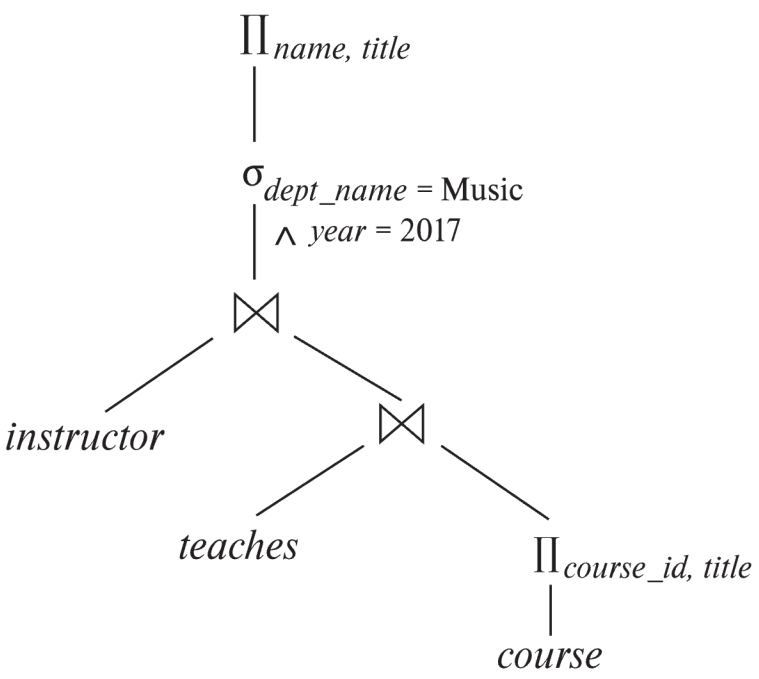
  - e.g. with r(A,B) = {(1,1), s(B,C) = { (1,1)}, t(A,C) = { }

Example with Multiple Transformations

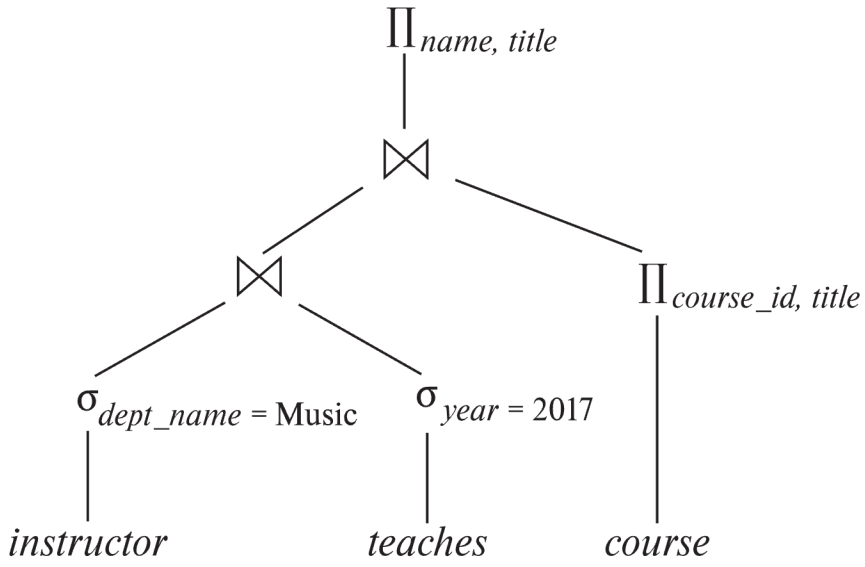
- Query: Find the names of all instructors in the Music department who have taught a course in 2017, along with the titles of the courses that they taught
  - $\Pi_{name,\ title}(\sigma_{dept\_name=\text{"Music"} \wedge year = 2017}$
    $(instructor \bowtie (teaches \bowtie \Pi_{course\_id,\ title}\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

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

Multiple Transformations (Cont.)



(a) Initial expression tree



(b) Tree after multiple transformations

Transformation Example: Pushing Projections

- Consider: $\Pi_{name,\ title}(\sigma_{dept\_name=\ \text{“Music”}}(instructor) \bowtie teaches) \bowtie \Pi_{course\_id,\ title}(course))))$
- When we compute

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

  we obtain a relation whose schema is:
  (*ID, name, dept_name, salary, course_id, sec_id, semester, year)*
- Push projections using equivalence rules 8a and 8b; eliminate unneeded attributes from intermediate results to get:

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

Join Ordering Example

- For all relations r_1, r_2, and r_3,

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

  (Join Associativity) $\bowtie$

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

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

  so that we compute and store a smaller temporary relation.

Join Ordering Example (Cont.)

- Consider the expression

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

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

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

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

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

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

first.

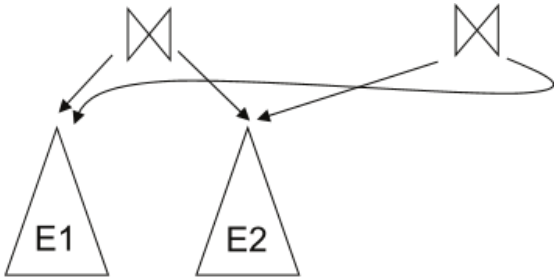
Enumeration of Equivalent Expressions

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

    Until no new equivalent expressions are generated above
- The above approach is very expensive in space and time
  - Two approaches
    - Optimized plan generation based on transformation rules
    - Special case approach for queries with only selections, projections and joins

Implementing Transformation Based Optimization

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



  - Same sub-expression may get generated multiple times
    - Detect duplicate sub-expressions and share one copy
- Time requirements are reduced by not generating all expressions
  - Dynamic programming is an option

Decorating parse trees

- Once the parse trees are build, need to decorate them with algorithm for each step
- Start with looking at the possible algorithms for each operators
- Other consideration
  - Materialization vs. Pipelining
  - Combining operators

Materialization vs. Pipelining

- Consider the following query

$$\sigma_{gpa>3.0}(Student \bowtie Department)$$

- We can push selection in

$$\sigma_{gpa>3.0}(Student) \bowtie Department$$

- Suppose we use nested loop
  - Either table can be in the inner loop

Materialization vs. Pipelining

$$\sigma_{gpa>3.0}(Student) \bowtie Department$$

- Now suppose Student is in the inner loop
- We first apply the selection: $\sigma_{gpa>3.0}(Student)$
- However, since that is in the inner loop, it will have to be read multiple times
- So the result of the selection need to be writing to secondary storage -- ***materialization***

Materialization vs. Pipelining

$$\sigma_{gpa>3.0}(Student) \bowtie Department$$

- Now suppose Student is in the outer loop
- We first apply the selection: $\sigma_{gpa>3.0}(Student)$
- However, since that is in the outer loop, it will have to be read only once
- So we can apply the following algorithm
  - Allocate some amount of main memory buffers for the result of the selection (the amount one allocate for the outer loop of the join)
  - Start executing the selection
  - Pause when the buffer is filled up
  - Then execute the join for those tuples in the buffers only
    - Notice that involve reading the inner loop (Department) once
  - After that is done, those tuples in the buffer from the selection is *no longer needed* (why?)
  - Discard those tuples, resume the selection until the buffer is filled
  - Repeat the process
- This is known as ***pipelining***

Materialization (Cont.)

- Materialized evaluation is always applicable
- Cost of writing results to disk and reading them back can be quite high
  - Our cost formulas for operations ignore cost of writing results to disk, so
    - Overall cost = Sum of costs of individual operations + cost of writing intermediate results to disk
- **Double buffering**: use two output buffers for each operation, when one is full write it to disk while the other is getting filled
  - Allows overlap of disk writes with computation and reduces execution time

Pipelining

- **Pipelined evaluation**: evaluate several operations simultaneously, passing the results of one operation on to the next.
- E.g., in previous expression tree, don't store result of

$$\sigma_{building="\text{Watson}"}(department)$$

  - instead, pass tuples directly to the join.. Similarly, don't store result of join, pass tuples directly to projection.
- Much cheaper than materialization: no need to store a temporary relation to disk.
- Pipelining may not always be possible – e.g., sort, hash-join.
  - Key observation: pipelining is possible if and only if the data that is pipelined into the next operations is read only once for that operation
- For pipelining to be effective, use evaluation algorithms that generate output tuples even as tuples are received for inputs to the operation.
- Pipelines can be executed in two ways: **demand driven** and **producer driven**

Pipelining (Cont.)

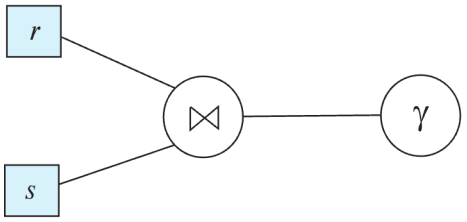
- In **demand driven** or **lazy** evaluation
  - system repeatedly requests next tuple from top level operation
  - Each operation requests next tuple from children operations as required, in order to output its next tuple
  - In between calls, operation has to maintain "**state**" so it knows what to return next
- In **producer-driven** or **eager** pipelining
  - Operators produce tuples eagerly and pass them up to their parents
    - Buffer maintained between operators, child puts tuples in buffer, parent removes tuples from buffer
    - if buffer is full, child waits till there is space in the buffer, and then generates more tuples
  - System schedules operations that have space in output buffer and can process more input tuples
- Alternative name: **pull** and **push** models of pipelining

Pipelining (Cont.)

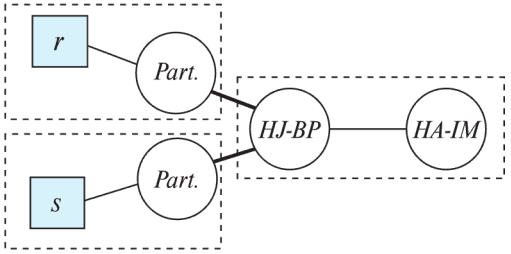
- Implementation of demand-driven pipelining
  - Each operation is implemented as an **iterator** implementing the following operations
    - **open()**
      - E.g., file scan: initialize file scan
        - state: pointer to beginning of file
      - E.g., merge join: sort relations;
        - state: pointers to beginning of sorted relations
    - **next()**
      - E.g., for file scan: Output next tuple, and advance and store file pointer
      - E.g., for merge join: continue with merge from earlier state till next output tuple is found. Save pointers as iterator state.
    - **close()**

Blocking Operations

- **Blocking operations**: cannot generate any output until all input is consumed
  - E.g., sorting, aggregation, …
- But can often consume inputs from a pipeline, or produce outputs to a pipeline
- Key idea: blocking operations often have two suboperations
  - E.g., for sort: run generation (during sort) and merge
  - For hash join: partitioning and build-probe (during the final nested loop)
- Treat them as separate operations



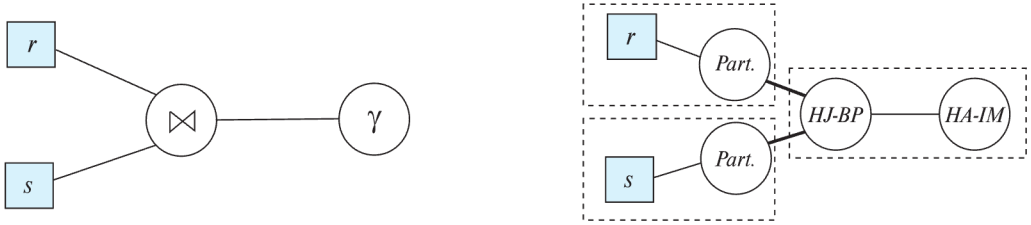
(a) Logical Query



(b) Pipelined Plan

Pipeline Stages

- **Pipeline stages**:
  - All operations in a stage run concurrently
  - A stage can start only after preceding stages have completed execution



(a) Logical Query

(b) Pipelined Plan

Materialization vs. Pipelining: Cost estimation

$$\sigma_{gpa>3.0}(Student) \bowtie Department$$

- Suppose
  - 1000 pages for Department
  - 5000 pages for Student
  - $\sigma_{gpa>3.0}(Student)$ return 2500 pages
  - Suppose 100 buffers, split 50 each
- Consider Student in the inner loop (no pipeline)
  - Cost (page read) = 5000 (read Student for σ) + 2500 (write Student for σ) + 1000 (outer loop for Department) + 1000 * 2500 / 50 (inner loop for $\sigma_{gpa>3.0}(Student)$) = 58500

Materialization vs. Pipelining: Cost estimation

$$\sigma_{gpa>3.0}(Student) \bowtie Department$$

- Suppose
  - 1000 pages for Department
  - 5000 pages for Student
  - $\sigma_{gpa>3.0}(Student)$ return 2500 pages
  - Suppose 100 buffers, split 50 each
- Consider Student in the outer loop (no pipeline)
  - Cost (page read) = 5000 (read Student for σ, outerloop) + 1000 * 2500 / 50 (inner loop for $Department$) = 55000

Index-only query

- Consider: $\sigma_{gpa>3.0}(Student)$
- Now suppose we have a secondary index on gpa
  - Probably not useful if a lot of tuples satisfies the query
- However, consider $\pi_{gpa}(\sigma_{gpa>3.0}(Student))$
  - Now only the gpa attribute is needed
  - All the information needed is in the secondary index
    - No need to go to the main table
    - Efficient regardless of number of tuples retrieved
- This is very useful for multi-attribute indices
  - Index where keys are a combination of attributes (att1, att2, att3 …)
  - Order is based on att1, if tied then att2, if tied again att3 etc.

Number of parse tree/plans

- Each decorated parse tree is known as a plan
- The number of plans can grow very quickly
- E.g. a query with a N-way joins can have (n-1)! ways to order it. (why?)
  - This does not even consider the various option (e.g. outer vs. inner loops)
- Thus we need to limit the number of parse tree generated
- Also after chosen a parse tree one will need to find the best way to decorate the tree

Number of parse tree/plans

- Also after chosen a parse tree one will need to find the best way to decorate the tree
- Must consider the interaction of evaluation techniques when choosing evaluation plans
  - choosing the cheapest algorithm for each operation independently may not yield best overall algorithm. E.g.
    - merge-join may be costlier than hash-join, but may provide a sorted output which reduces the cost for an outer level aggregation.
    - nested-loop join may provide opportunity for pipelining

Cost-Based Optimization

- Consider finding the best join-order for $r_1 \bowtie r_2 \bowtie \ldots \bowtie r_n$.
- There are $(2(n-1))!/(n-1)!$ different join orders for above expression. With $n = 7$, the number is 665280, with $n = 10$, the number is greater than 176 billion!
- No need to generate all the join orders. Using dynamic programming, the least-cost join order for any subset of $\{r_1, r_2, \ldots r_n\}$ is computed only once and stored for future use.

Dynamic Programming in 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. Choose the cheapest of the $2^n - 2$ alternatives.
  - 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
    - Dynamic programming

Join Order Optimization Algorithm

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

Join Order Optimization Algorithm (cont.)

for each algorithm A for joining results of *P*1 and *P*2
// For indexed-nested loops join, the outer could be *P1* or *P2*
// Similarly for hash-join, the build relation could be *P1* or *P2*
// We assume the alternatives are considered as separate algorithms

if algorithm *A* is indexed nested loops
Let P_i and P_o denote inner and outer inputs
if P_i has a single relation r_i and r_i has an index on the join attribute

*plan = "*execute P_o*.plan*; join results of P_o and r_i using *A*",
with any selection conditions on P_i performed as part of the join condition
cost = P_o*.cost* + cost of *A*

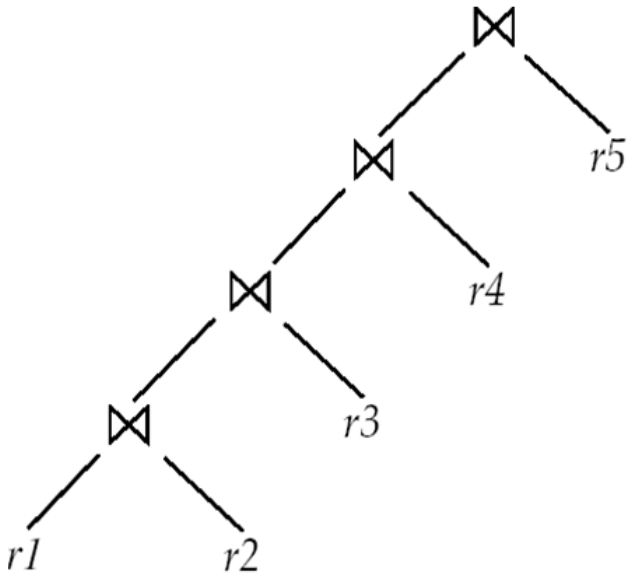
else *cost* = ∞; /* cannot use indexed nested loops join */

else
plan = "execute *P1.plan*; execute *P2.plan*;
join results of *P1* and *P2* using *A*;"
cost = *P1.cost* + *P2.cost* + cost of *A*

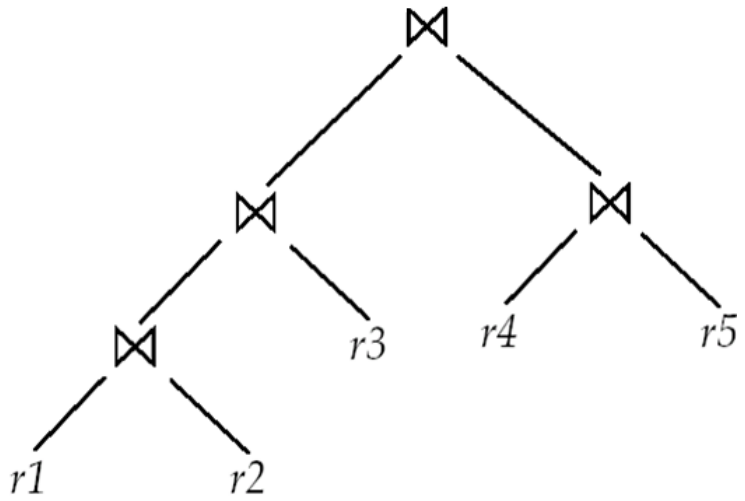
…. See previous page

Left Deep Join Trees

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



(a) Left-deep join tree



(b) Non-left-deep join tree

Cost of Optimization

- With dynamic programming time complexity of optimization with bushy trees is $O(3^n)$.
  - With $n = 10$, this number is 59000 instead of 176 billion!
- Space complexity is $O(2^n)$
- To find best left-deep join tree for a set of n relations:
  - Consider n alternatives with one relation as right-hand side input and the other relations as left-hand side input.
  - Modify optimization algorithm:
    - 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 .
- If only left-deep trees are considered, time complexity of finding best join order is $O(n\ 2^n)$
  - Space complexity remains at $O(2^n)$
- Cost-based optimization is expensive, but worthwhile for queries on large datasets (typical queries have small n, generally < 10)

Interesting Sort Orders

- 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 make a later operation (join/group by/order by) cheaper
  - 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
- Not sufficient to find the best join order for each subset of the set of *n* given relations
  - must find the best join order for each subset, **for each interesting sort order**
  - Simple extension of earlier dynamic programming algorithms
  - Usually, number of interesting orders is quite small and doesn't affect time/space complexity significantly

Cost Based Optimization with Equivalence Rules

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

Heuristic Optimization

- 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.

Structure of Query Optimizers

- Many optimizers considers only left-deep join orders.
  - Plus heuristics to push selections and projections down the query tree
  - Reduces optimization complexity and generates plans amenable to pipelined evaluation.
- Heuristic optimization used in some versions of Oracle:
  - Repeatedly pick "best" relation to join next
    - Starting from each of n starting points. Pick best among these
- Intricacies of SQL complicate query optimization
  - E.g., nested subqueries

Structure of Query Optimizers (Cont.)

- Some query optimizers integrate heuristic selection and the generation of alternative access plans.
  - Frequently used approach
    - heuristic rewriting of nested block structure and aggregation
    - followed by cost-based join-order optimization for each block
  - Some optimizers (e.g. SQL Server) apply transformations to entire query and do not depend on block structure
  - **Optimization cost budget** to stop optimization early (if cost of plan is less than cost of optimization)
  - **Plan caching** to reuse previously computed plan if query is resubmitted
    - Even with different constants in query
- Even with the use of heuristics, cost-based query optimization imposes a substantial overhead.
  - But is worth it for expensive queries
  - Optimizers often use simple heuristics for very cheap queries, and perform exhaustive enumeration for more expensive queries

Cost estimation

- Recall the basic problem of query execution (slightly modified)
  - The best way to execute a query (operation) depends on the size of its results, which you don't know until you execute it
- Thus need to estimate the tuples return from an operation

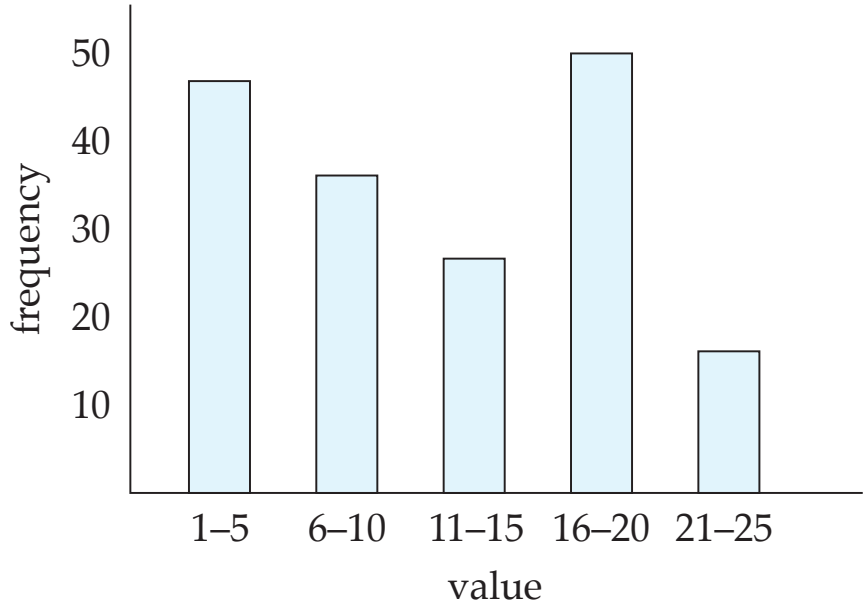
Statistical Information for Cost Estimation

- n_r: number of tuples in a relation *r.*
- b_r: number of blocks containing tuples of *r.*
- l_r: size of a tuple of *r.*
- f_r: blocking factor of *r* — i.e., the number of tuples of *r* that fit into one block.
- *V(A, r):* number of distinct values that appear in *r* for attribute *A;* same as the size of $\prod_A(r)$.
- If tuples of *r* are stored together physically in a file, then:

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

Histograms

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



- **Equi-width** histograms
- **Equi-depth** histograms break up range such that each range has (approximately) the same number of tuples
  - E.g. (4, 8, 14, 19)
- Many databases also store *n* **most-frequent values** and their counts
  - Histogram is built on remaining values only

Histograms (cont.)

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

Selection Size Estimation

- $\sigma_{A=v}(r)$
  - $n_r / V(A,r)$: number of records that will satisfy the selection
  - Equality condition on a key attribute: *size estimate* = 1
- $\sigma_{A \leq V}(r)$ (case of $\sigma_{A \geq V}(r)$ is symmetric)
  - Let c denote the estimated number of tuples satisfying the condition.
  - If min(A,r) and max(A,r) are available in catalog
    - c = 0 if v < min(A,r)
    - c = $n_r \cdot \frac{v - \min(A,r)}{\max(A,r) - \min(A,r)}$
  - If histograms available, can refine above estimate
  - In absence of statistical information *c* is assumed to be $n_r / 2$.

Size Estimation of Complex Selections

- The **selectivity** of a condition θ_i is the probability that a tuple in the relation *r* satisfies θ_i .
  - If s_i is the number of satisfying tuples in *r,* the selectivity of θ_i is given by s_i / n_r.
- **Conjunction:** $\sigma_{\theta 1 \wedge \theta 2 \wedge \ldots \wedge \theta n}(r)$. *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 ⋈ *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!*)

Estimation of the Size of Joins

- The Cartesian product r x 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 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
  - therefore, 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* ⋈ *takes, ID* in *takes* is a foreign key referencing *student*
  - hence, the result has exactly n_{takes} tuples, which is 10000

Estimation of the Size of Joins (Cont.)

- If $R \cap S = \{A\}$ is not a key for R or S.
  If we assume that every tuple t in R produces tuples in R 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
  - Use formula similar to above, for each cell of histograms on the two relations

Estimation of the Size of Joins (Cont.)

- Compute the size estimates for *depositor* ⋈ *customer* without using information about foreign keys:
  - *V(ID, takes)* = 2500, and
    V(ID, student) = 5000
  - The two estimates are 5000 * 10000/2500 = 20,000 and 5000 * 10000/5000 = 10000
  - We choose the lower estimate, which in this case, is the same as our earlier computation using foreign keys.

Size Estimation for Other Operations

- Projection: estimated size of $\prod_A(r)$ = $V(A,r)$
- Aggregation : estimated size of ${}_G\gamma_A(r)$ = $V(G,r)$
- 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 $\sigma_{\theta 1 \text{ or } \theta 2}(r)$
  - 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.

Size Estimation (Cont.)

- 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

Estimation of Number of Distinct Values

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

Estimation of Distinct Values (Cont.)

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

Estimation of Distinct Values (Cont.)

- Estimation of distinct values are straightforward for projections.
  - They are the same in $\Pi_{A\,(r)}$ as in *r*.
- The same holds for grouping attributes of aggregation.
- For aggregated values
  - For min(*A*) and max(*A*), the number of distinct values can be estimated as min(V(*A,r*), *V*(*G,r*)) where G denotes grouping attributes
  - For other aggregates, assume all values are distinct, and use *V*(*G,r*)

Optimizing Nested Subqueries**

- Nested query example:

```
select name
from instructor
where exists (select *
                from teaches
                where instructor.ID = teaches.ID and teaches.year = 2019)
```

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

Optimizing Nested Subqueries (Cont.)

- Correlated evaluation may be quite inefficient since
  - a large number of calls may be made to the nested query
  - there may be unnecessary random I/O as a result
- SQL optimizers attempt to transform nested subqueries to joins where possible, enabling use of efficient join techniques
- E.g.,: earlier nested query can be rewritten as $\prod_{name}(instructor \bowtie_{instructor.ID=teaches.ID \wedge teaches.year=2019} teaches)$
- Note: the two queries generate different numbers of duplicates (why?)
  - Can be modified to handle duplicates correctly using semijoins

Optimizing Nested Subqueries (Cont.)

- The **semijoin** operator $\ltimes$ is defined as follows
  - A tuple r_i appears n times in $r \ltimes_\theta s$ if it appears *n* times in *r*, and there is at least one matching tuple s_i in *s*
- E.g.: earlier nested query can be rewritten as $\prod_{name}(instructor \ltimes_{instructor.ID=teaches.ID \wedge teaches.year=2019} teaches)$
  - Or even as: $\prod_{name}(instructor \ltimes_{instructor.ID=teaches.ID} (\sigma_{teaches.year=2019} teaches))$
  - Now the duplicate count is correct!
- The above relational algebra query is also equivalent to

```
from instructor
where ID in (select teaches.ID
                    from teaches
                    where teaches.year = 2019)
```

Optimizing Nested Subqueries (Cont.)

- This could also be written using only joins (in SQL) as

  with t_1 **as**
  (**select distinct** *ID*
  from *teaches*
  where *year = 2019)*
  select *name*
  from *instructor*, t_1
  where t_1.*ID = instructor.ID*

- The query

  select *name*
  from *instructor*
  where not exists (**select** *
  from *teaches*
  where *instructor.ID = teaches.ID* **and** *teaches.year = 2019*)

  can be rewritten using the **anti-semijoin** operation as $\overline{\ltimes}$

Optimizing Nested Subqueries (Cont.)

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

- Rewrite: **select** A
  from $r_1, r_2, \ldots, r_n$
  where P_1 **and exists** (**select** *
  from $s_1, s_2, \ldots, s_m$
  where P_2^1 **and** P_2^2)
- To: $\prod_A(\sigma_{P1}(r_1 \times r_2 \times \ldots \times r_n) \ltimes_{P2^2} \sigma_{P2^1}(s_1 \times s_2 \times \ldots \times s_m)$
  - P_2^1 contains predicates that do not involve any correlation variables
  - P_2^2 contains predicates involving correlation variables
- The process of replacing a nested query by a query with a join/semijoin (possibly with a temporary relation) is called **decorrelation**.
- Decorrelation is more complicated in several cases, e.g.
  - The nested subquery uses aggregation, or
  - The nested subquery is a scalar subquery
  - Correlated evaluation used in these cases

Decorrelation (Cont.)

- Decorrelation of scalar aggregate subqueries can be done using groupby/aggregation in some cases
- **select** *name*
  from *instructor*
  where 1 < (**select count**(*)
  from *teaches*
  where *instructor.ID = teaches.ID*
  and *teaches.year = 2019*)
- $\Pi_{name}(instructor \ltimes_{instructor.ID=TID \wedge 1 < cnt} ($
  $_{ID\ as\ TID}\gamma_{\mathbf{count}(*)\ as\ cnt}(\sigma_{teaches.year=2019}(teaches))))$