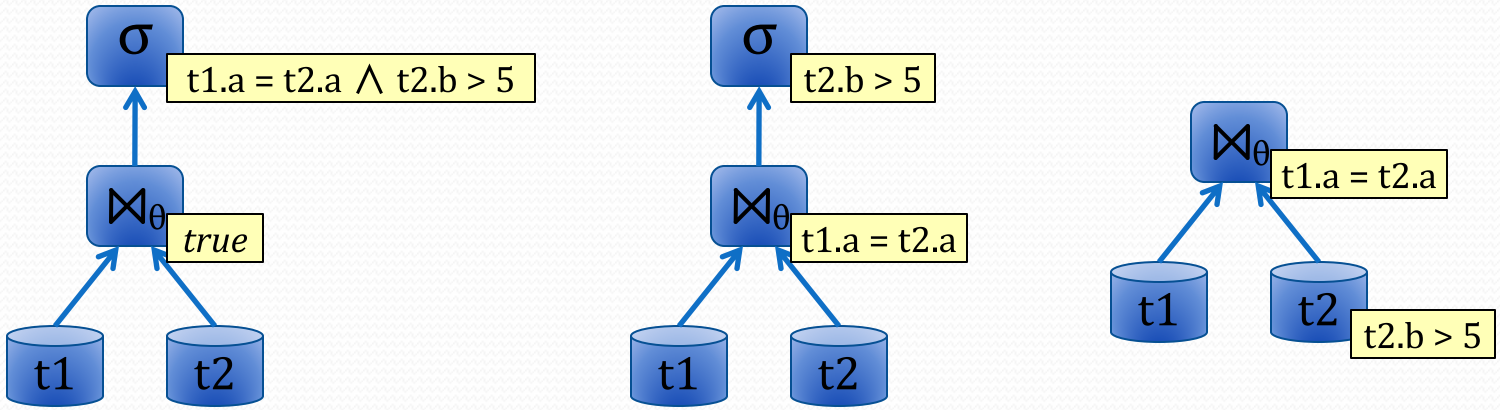
# Relational Database System Implementation

CS122 – Lecture 9

Winter Term, 2017-2018

# Equivalent Plans?

- Previously had this query:
  - SELECT * FROM t1, t2 WHERE t1.a = t2.a AND t2.b > 5;




- How do we know these plans are actually equivalent?

# Equivalent Plans

- Two plans are *equivalent* if they produce the same results for every legal database instance
  - A “legal” database instance satisfies all constraints
- Generally, the order of tuples is irrelevant
  - If sorting is not specified on results, two equivalent plans may generate results in different orders
- *Equivalence rules* specify different forms of an expression that are equivalent
  - Can prove that these rules hold for all legal databases
  - Can use them to transform query plans into equivalent (but hopefully faster) plans

# Simple Equivalence Rules

- Cascade of σ:
  - $\sigma_{\theta 1 \wedge \theta 2}(E) = \sigma_{\theta 1}(\sigma_{\theta 2}(E))$
- σ is commutative:
  - $\sigma_{\theta 1}(\sigma_{\theta 2}(E)) = \sigma_{\theta 2}(\sigma_{\theta 1}(E))$
- Selections, Cartesian products, and theta-joins:
  - $\sigma_{\theta}(E1 \times E2) = E1 \bowtie_{\theta} E2$
  - $\sigma_{\theta 1}(E1 \bowtie_{\theta 2} E2) = E1 \bowtie_{\theta 1 \wedge \theta 2} E2$
- Theta-joins are commutative:
  - $E1 \bowtie_{\theta} E2 = E2 \bowtie_{\theta} E1$

# Theta Join Equivalence Rules

- Natural joins are associative:
  - $(E1 \bowtie E2) \bowtie E3 = E1 \bowtie (E2 \bowtie E3)$
- Theta-joins are also associative, but it's a bit trickier:
  - $(E1 \bowtie_{\theta 1} E2) \bowtie_{\theta 2 \wedge \theta 3} E3 = E1 \bowtie_{\theta 1 \wedge \theta 3} (E2 \bowtie_{\theta 2} E3)$
  - θ1 only refers to attributes in E1 and/or E2
  - θ2 only refers to attributes in E2 and/or E3
  - θ3 only refers to attributes in E1 and/or E3
  - Any of these conditions might also simply be *true*

# Theta Join Equivalence Rules (2)

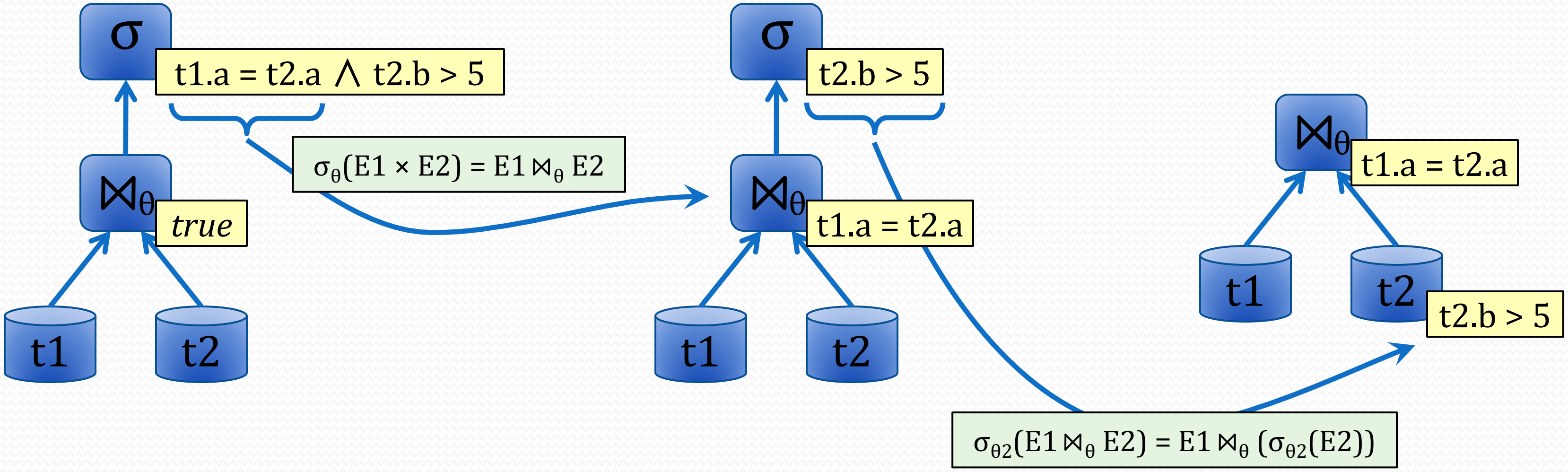
- Can sometimes distribute selects over theta-joins:
  - $\sigma_{\theta 1}(E1 \bowtie_{\theta} E2) = \sigma_{\theta 1}(E1) \bowtie_{\theta} E2$
    - θ1 only refers to attributes in E1
  - $\sigma_{\theta 1 \wedge \theta 2}(E1 \bowtie_{\theta} E2) = \sigma_{\theta 1}(E1) \bowtie_{\theta} \sigma_{\theta 2}(E2)$
    - θ1 only refers to attributes in E1
    - θ2 only refers to attributes in E2

# Equivalence Rules

- Many other equivalence rules besides these
  - Cover grouping, projects, outer joins, set operations, duplicate elimination, sorting, etc.
- Grouping: $\sigma_\theta({}_A\mathcal{G}_F(E))$ is equivalent to ${}_A\mathcal{G}_F(\sigma_\theta(E))$
  - …as long as θ only involves attributes in A!
- Outer joins: $\sigma_\theta(E1 ⟕ E2)$ is equivalent to $\sigma_\theta(E1) ⟕ E2$
  - θ only involves attributes in E1

# Equivalence Rules

- Equivalence rules allow us to transform plans, and know the results will not change:



# Outer Join Transformations

- Need to be very careful transforming outer joins:
  - Obviously correct equivalences for natural joins / theta joins don't necessarily hold for outer joins!
- Is $\sigma_\theta$(E1 ⟕ E2) equivalent to E1 ⟕ $\sigma_\theta$(E2)?
  - θ only uses attributes in E2
  - These are <u>not</u> equivalent. Example:
    - r(A, B) with one row { (1, 2) }
    - s(B, C) with one row { (2, 3) }
    - θ is C = 1
    - $\sigma_{C=1}$(r ⟕ s) = { } *(empty relation)*, but r ⟕ $\sigma_{C=1}$(s) = { (1, 2, *null*) }

# Outer Join Transformations (2)

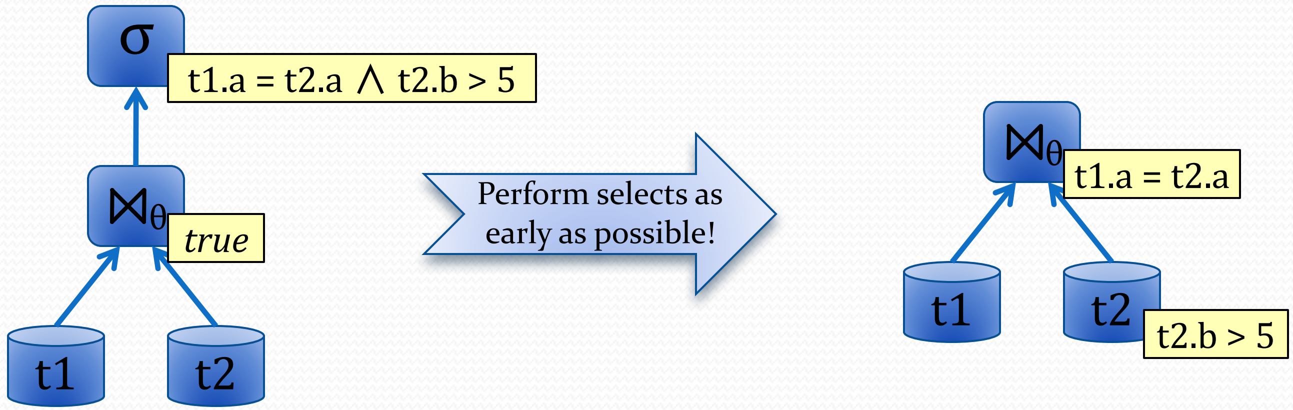
- Need to be very careful transforming outer joins:
  - Obviously correct equivalences for natural joins / theta joins don't necessarily hold for outer joins!
- Is (E1 ⟕ E2) ⟕ E3 equivalent to E1 ⟕ (E2 ⟕ E3)?
  - These are not equivalent. Example:
    - r(A, B) with one row { (1, 2) }
    - s(A, C) with one row { (2, 3) }
    - t(A, D) with one row { (1, 4) }
    - (r ⟕ s) ⟕ t = { (1, 2, *null*) } ⟕ t = { (1, 2, *null*, 4) }
    - r ⟕ (s ⟕ t) = r ⟕ { (2, 3, *null*) } = { (1, 2, *null*, *null*) }

# Query Plan Optimization

- Generally understand how to map SQL queries to plans
  - Ignoring subqueries in SELECT and WHERE clauses for the time being…
- Understand how to implement basic plan nodes
  - Still a lot of optimizations to cover though…
- A query can be evaluated by many different plans…
- How do we find an *optimal* plan to evaluate a query?
  - Many different approaches
  - <u>All</u> depend on equivalence rules to guide generation of equivalent plans

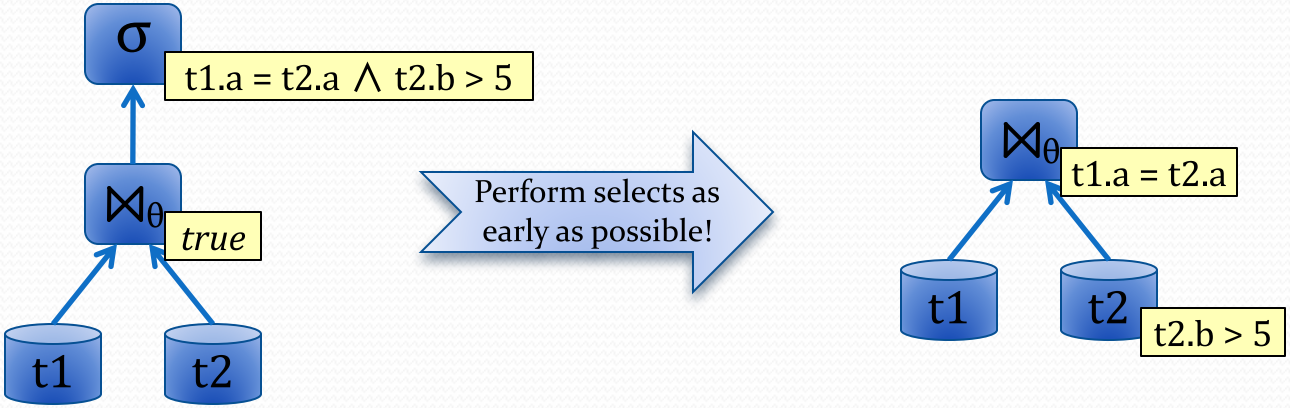
# Heuristic Plan Optimization

- Can transform plans purely based on heuristics
  - Guidelines for what plans will generally be "better"
  - Uses equivalence rules, but no plan costing!
- Example: "Perform selects as early as possible!"
- Would properly handle our previous example:
  - Push predicates down the plan-tree as far as possible



# Heuristic Plan Optimization (2)

- Unfortunately, heuristics don't always work




- Scenario:
  - t1 is a small table
  - t2 is very large, and has an index on a, but *no* index on b!
  - If t2.b > 5 is applied first, join can't use t2's index to find rows
    - Would *greatly* improve join performance in this case
  - Would likely be faster to perform $\sigma_{t2.b>5}(\ldots)$ last, in this case!

# Cost-Based Plan Optimization

- Clearly gain a benefit from estimating a plan's cost
  - Gives us feedback about whether an alternative is actually likely to be better
- *Cost-based optimizers* explore query plan space, and choose the "best" one based on the estimated cost
- Could exhaustively enumerate <u>all</u> equivalent plans…
  - Assign each plan a cost, and choose the best one!
- Unfortunately, plan space is often extremely large
  - Just picking a join ordering produces *many* options…

# Example: Join Ordering

- Given n relations to join: $r_1, r_2, \ldots, r_n$
  - Join is a binary operation
  - $r_1 \bowtie r_2$ may have a different cost than $r_2 \bowtie r_1$
  - Produces (2(n – 1))! / (n – 1)! different orderings!
    - (See Practice Exercise 13.10 in textbook for details.)

| | |
|---|---|
| n = 3 | 12 orderings |
| n = 4 | 120 orderings |
| n = 5 | 1,680 orderings! |
| n = 6 | 30,240 orderings!! |
| n = 7 | 665,280 orderings!!! |

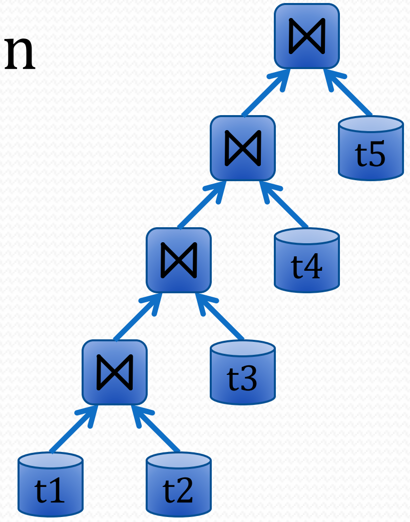
# Exhaustive Plan Enumeration

- Pursuing this strategy requires careful implementation
- Must represent plans in a very space-efficient manner
  - E.g. memoize subplans, so that common subplans are represented in memory only once

- Some query planners use exhaustive plan enumeration
  - Volcano and Cascades projects used this approach
  - SQLServer’s optimizer is based on these projects

# Guided Plan Enumeration

- Most query planners are satisfied with any good plan
  - *"Don't let the perfect become the enemy of the good."*
- Constrain the plan search-space in various ways
- E.g. some planners only consider *left-deep join trees*
  - For *n* tables, only have *n*! join orders to consider
  - Is also very friendly to pipelined evaluation
    - e.g. nested loops don't have a whole subplan to evaluate over and over, for inner relation
- Rely more on higher-level heuristics
  - Don't just repeatedly apply fine-grained equivalence rules



# Guided Plan Enumeration (2)

- Many queries involve joins of multiple tables
  - (Also, subqueries in SELECT and WHERE can often be transformed into joins.)
- A common (non-exhaustive) optimization strategy:
  - Perform high-level transformations at SQL AST level
    - Flattening subqueries into a larger top-level query with joins
  - Apply heuristics based on strategies that generally improve query performance
  - Focus specifically on choosing a good join order
  - Use plan costing to evaluate whether alternatives are actually better!

# Bottom-Up: Dynamic Programming

- Can enumerate plans in a *bottom-up* approach, or a *top-down* approach
- Example: bottom-up approach
  - Use dynamic programming to search the plan-space
  - Decompose plan into smallest subplans; choose "best" implementation for each subplan, and record its cost
  - When building up larger subplans, reuse earlier work: simply choose "best" way to combine earlier subplans
  - Usually produces *very* good plans but not always the best
    - Can't take advantage of higher-level, whole-plan optimizations

# Top-Down: Branch and Bound

- Example: top-down approach
  - Use branch-and-bound strategy
  - Generate a "good" query plan using heuristics, then compute its cost C
  - Use C as an upper bound for plans we will consider
    - When applying transformations, immediately discard any plan with a cost larger than C
    - If we find a plan with a lower cost, lower C to the new cost
- Upper-bound cost C can guide when to stop optimizing
  - If C is still really large, keep looking for better plans…
  - If C is small, additional effort is probably unnecessary

# Optimizing Join Order

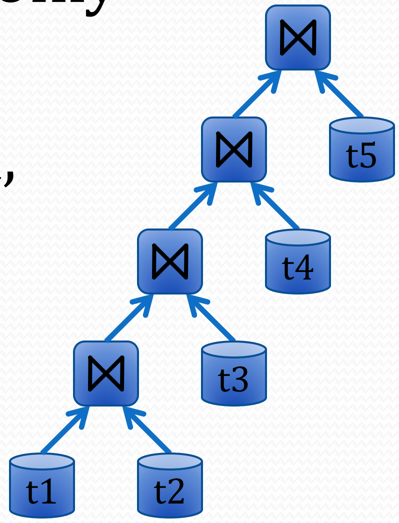
- Given: r1 ⋈ r2 ⋈ r3 ⋈ r4 ⋈ r5
  - Need to devise the optimal join order (along with the optimal join algorithms, access paths, etc.)
  - For n = 5, there are 1680 different join orderings
- Assume we know the optimal join order for r1 ⋈ r2 ⋈ r3
- Want to know optimal order for (r1 ⋈ r2 ⋈ r3) ⋈ r4 ⋈ r5
  - Really don't need to keep figuring out the optimal order for r1 ⋈ r2 ⋈ r3 over and over again…
  - Just reuse the subplan and associated cost already computed for r1 ⋈ r2 ⋈ r3, when trying orders with r4, r5

# Bottom-Up Join Optimizer

- Finding join order with dynamic programming:
- Step 1:
  - Determine optimal way to access each relation directly (including index optimizations based on predicates, etc.)
  - Compute a cost for each access
- Step 2:
  - Determine optimal way to join each pair of relations, using results computed in step 1, along with the computed costs
- Step 3…N:
  - Repeat, adding another relation at each step, reusing earlier results, until optimal way to join all N relations is found

# Left-Deep Join Orders

- Some databases limit join ordering to left-deep orders
  - Reduces total number of join orders down to *n*!
  - Facilitates pipelining (particularly if stuck with nested-loops join)
- Easy to constrain bottom-up algorithm to only explore left-deep join orders:
  - When adding another relation to a subplan, always add it on right side of the new join operation, with subplan the on left side



# Top-Down Join Optimizer

- Another version of the same algorithm, written in a more "top-down" style: (Database System Concepts, 6ed, p.600)

```
/* S is a set of relations to join */
procedure FindBestPlan(S)
    if (bestplan[S].cost ≠ ∞)                /* best plan is already computed */
        return bestplan[S]

    if (S contains only 1 relation)
        set bestplan[S].plan, bestplan[S].cost based on best way of accessing S
    else
        set bestplan[S].cost = ∞
        for each non-empty proper subset S1 of 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 = join P1 and P2 using algorithm A

    return bestplan[S]
```

# Top-Down Join Optimizer (2)

- Can constrain this to only produce left-deep join trees
  - Instead of enumerating all subsets of *S*, choose one relation *r* for right subplan, and *S* – *r* for left subplan
- Enumerating subsets at each level is repetitive and uses extra memory
  - Could modify the implementation to memoize results

# Improving the Join Optimizer…

- This optimization approach doesn't always produce the best join order
  - At each step, we only keep the *optimal* solution we find
  - The lowest-cost solution to a subproblem may force more costly operations higher up in the plan-tree
- Selinger-style plan optimization:
  - Besides keeping lowest-cost solution for each problem, also keep solutions that produce "interesting orders"
  - Sometimes, a higher-level operation can use the slightly costlier ordered result to reduce overall costs

# Selinger-Style Optimization

- Selinger-style plan optimization
  - Uses dynamic programming to generate plans…
  - Also keep more expensive subplans that produce results in "interesting orders"
    - Subplans that are slower than the fastest one found, but that produce results in possibly useful orderings
    - e.g. subplans whose results are ordered on the same attributes as a higher-level ORDER BY operation
    - e.g. subplans whose results are ordered on the same attributes as a higher-level join operation
  - Can take advantage of higher-level optimizations than simple dynamic programming

# Selinger-Style Optimization (2)

- Named after Pamela Selinger
  - Worked on planner/optimizer for System R
  - Helped to develop many of the plan-costing approaches used in most databases today
- System R was an early relational database research project at IBM
- Many critical accomplishments:
  - System R's SEQUEL language is the basis of our SQL
  - Use of plan-costing estimates and dynamic programming in plan optimization
  - Demonstrated the feasibility of transaction processing

# System-R Join Optimizer

- Slightly altered version of bottom-up approach:
- For each available ordering of results, record the optimal plan that produces that result-order
- Also record optimal plan that produces unordered results
  - …unless an ordered result's cost is already lower than this!
- Step 1:
  - For each relation:
    - Examine indexes to determine what result-orderings are available
    - For each possible result-ordering, determine the optimal plan for accessing the relation (also applying relevant predicates, etc.)
    - Also determine optimal plan for unordered access. If this is costlier than some ordered result, discard this plan.

# System-R Join Optimizer (2)

- Step 2:
  - Again, compute optimal plans to join pairs of relations
  - Given two relations r1 and r2:
    - Consider all plans for joining r1 and r2, based on Step 1 results
    - Some of these plans will also produce ordered results
    - Partition plans into groups based on their join orderings, and save optimal plan for each join order
    - Discard the unordered plan if some ordered plan is more efficient
- Continue this process until all N relations are joined
  - Join planner may produce multiple ways to join input tables
  - Query planner can choose a join-plan based on overall query requirements (e.g. top-level ORDER BY or GROUP BY clause)

# System-R Join Optimizer (3)

- Usually aren't *that* many interesting orders to consider
  - Tables might have 1-3 indexes, only a few of which are relevant for each join operation being considered
  - Considering result-orderings doesn't add substantial memory overhead to the join optimizer
  - Can really improve optimizer results in cases where the result-ordering can be leveraged for faster queries