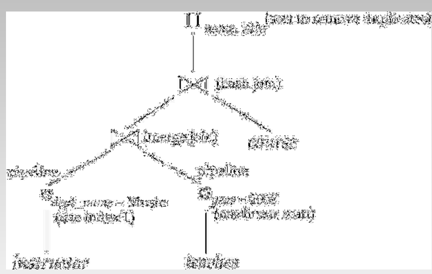
13: Query Optimization

- Cost-based optimization
- Dynamic Programming for Choosing Evaluation Plans
- Rule-Based Query Optimization
- Transformation of Relational Expressions

13.1 Cost-Based Optimization

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



- There are several- sometimes hundreds- of equivalent evaluation plans
- The optimizer evaluates the cost of all and selects the cheapest
- It relies on the ability to estimate both
  - the cost of each operation, and
  - the size of the results in each operation in order to evaluate the follow up operation

Selection σ Size Estimation

- *Case:* $\sigma_{A=v}(r)$

$$size = |\sigma_{A=v}| = \frac{n_r}{V(A,r)}$$

If A is a key, then $V(A,r) = n_r$

$$size = |\sigma_{A=v}| = \frac{n_r}{n_r} = 1$$

Selection σ Size Estimation

- *Case:* $\sigma_{A \leq v}(r)$ **(case of $\sigma_{A \geq v}(r)$ is symmetric)**
  - **If min(A,r) and max(A,r) are available in catalog**

$$size = |\sigma_{A \leq v}| = n_r * \frac{v - \min(A,r)}{\max(A,r) - \min(A,r)}$$

If $v < \min(A,r)$ **then** $size = |\sigma_{A \leq v}| = 0$

  - **If we know the value distribution of A (e.g. histograms) we can get much better estimates.**
  - **In absence of statistical information** *size* **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 tuples in r satisfying θ_i the selectivity of it is s_i / n
- Conjunction result estimate : (*assuming Independence of predicates)*

$$| \sigma_{\theta 1 \wedge \theta 2 \wedge \ldots \wedge \theta n}(r) | = n_r * \frac{s_1 * s_2 * \ldots * s_n}{n_r^n}$$

- Disjunction result estimate:

$$| \sigma_{\theta 1 \vee \theta 2 \vee \ldots \vee \theta n}(r) | = n_r * \left(1 - (1 - \frac{s_1}{n_r}) * (1 - \frac{s_2}{n_r}) * \ldots * (1 - \frac{s_n}{n_r}) \right)$$

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

  $| \sigma_{\neg\theta}(r). | = n_r - size(\sigma_\theta(r))$

324

Size Estimation of Join $n(R \bowtie S)$

$0 \le |n(r \bowtie s)| \le n_r * n_s$

- (0 is when nothing joins and $n_r * n_s$ when everything joins

- if joining attribute is a key of r then $| n(r \bowtie s) | \le n_s$
  each value of s.A would join to at most one value of r.A

- if joining attribute is a key of r and a foreign key of s referencing r.A then
  $| n(r \bowtie s) | = n_s$
  each value of s.A would join to exactly one value of r.A

- if joining attribute is not a key then
  each value of A in r appears $\frac{n_s}{V(A,s)}$ times in s, therefore,
  n_r tuples of r produce: $| n(r \bowtie s) | = \frac{n_r * n_s}{V(A,s)}$

  symmetrically n_s tuples of s produce : $| n(r \bowtie s) | = \frac{n_s * n_r}{V(A,r)}$

  if these two values are different we use: $\min\{ \frac{n_r * n_s}{V(A,s)}, \frac{n_s * n_r}{V(A,r)} \}$

325

Join Example: student $\bowtie$ takes

Catalog stats:

$n_{student}$ = 5,000 $f_{student}$ = 50 $b_{student}$ =5,000/50 = 100

n_{takes} = 10,000 f_{takes} = 25 b_{takes} = 10,000/25 = 400

V(*ID, student*) = 5,000 (*primary key*)
V(*ID, takes*) = 2,500 (*ID* is a foreign key referencing *student*)

Because the join attribute ID is a FK
| student $\bowtie$ takes | = n_{takes} = 10,000

If ID was not a primary/foreign key in the above: min($\frac{n_r * n_s}{V(ID, student)}$, $\frac{n_s * n_r}{V(ID, takes)}$ ***)***

$$= min\{ \frac{5{,}000 * 10{,}000}{5000}, \frac{5{,}000 * 10{,}000}{2{,}500} \} = 10{,}000$$

Size Estimation for Other Operations

- **Projection: estimated size of $\prod_A(r)$ = $V(A,r)$**
- **Aggregation : estimated size of $_A g_F(r)$ = $V(A,r)$**
- **Set operations**
  - **For unions/intersections of selections on the same relation: use size estimate for selections**
    - **E.g. $\sigma_{\theta 1}(r) \cup \sigma_{\theta 2}(r)$ can be rewritten as $\sigma_{\theta 1 \vee \theta 2}(r)$**
  - **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.**

The above three estimates may be quite inaccurate, but provide upper bounds on the sizes.

Size Estimation (Cont.)

- Outer joins:
  - Estimated size of r ⟕ s = *size of* $r \bowtie s$ + *size of* r
    - Case of right outer join is symmetric

      = *size of* $r \bowtie s$ + *size of* s
  - Estimated size of r ⟗ s = *size of* $r \bowtie s$ + size of r + size of s

328

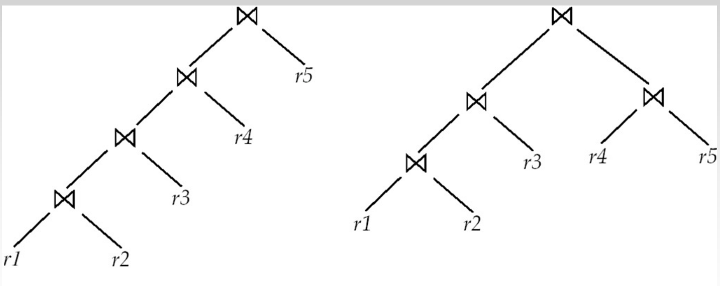
Join-Order Optimization

- 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
  - 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 cached for its use in the algorithm (sub-optimal).

329

Left Deep Join Trees

- In left-deep join trees, the right-hand-side input for each join is a base relation, not the result of an intermediate join
- Preferable because we have indices and better stats on base relations



(a) Left-deep join tree (b) Non-left-deep join tree

Choice of Evaluation Plans

- 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
- Practical query optimizers incorporate elements of the following two broad approaches:
  1. Search all the plans and choose the best plan in a cost-based fashion.
  2. Uses heuristics to choose a plan.

Materialized Views

- A materialized view is a view whose contents are computed and stored
- Consider the view
  create view *department_total_salary*(*dept_name, total_salary*) as
  select *dept_name*, sum(*salary*)
  from *instructor*
  group by *dept_name*
- Materializing the above view would be very useful if the total salary by department is required frequently
  - Saves the effort of finding multiple tuples and adding up their amounts

332

Materialized View Maintenance

- Keeping a materialized view up-to-date with the underlying data
- Materialized views can be maintained by re-computation on every update
- A much better option is to use incremental view maintenance
  - Changes to database relations are used to compute changes to the materialized view, which is then updated
- View maintenance can be done by
  - Manually defining triggers on insert, delete, and update of each relation in the view definition
  - Manually written code to update the view whenever database relations are updated
  - Periodic recomputation (e.g. nightly)
  - Lazy approach- incrementally update the view on demand (when accessed)
  - Some of the above methods are directly supported by many database system

333

Incremental View Maintenance

- The changes (inserts and deletes) to a relation or expressions are referred to as its differential
  - Set of tuples inserted to and deleted from r are denoted i_r and d_r
- To simplify our description, we only consider inserts and deletes
  - We replace updates to a tuple by deletion of the tuple followed by insertion of the update tuple
- We describe how to compute the change to the result of each relational operation, given changes to its inputs- define the ***differential algebra***
- We then outline how to handle relational algebra expressions

Join Operation

- Consider the materialized view $v = r \bowtie s$

r		s		v
A, 1 B, 2	$\bowtie$	1, p 2, r 2, s		A, 1, p B, 2, r B, 2, s

- Consider the case of an insert to r:

r		s		v
A, 1 B, 2	$\bowtie$	1, p 2, r 2, s		A, 1, p B, 2, r B, 2, s
C,2				C, 2, r C, 2, s

- Let r^{old} and r^{new} denote the old and new states of relation r
  - We can write $r^{new} \bowtie s$ as $(r^{old} \cup i_r) \bowtie s$
  - And rewrite the above to $(r^{old} \bowtie s) \cup (i_r \bowtie s)$
  - But $(r^{old} \bowtie s)$ is simply the old value of the materialized view, so the incremental change to the view is just $i_r \bowtie s$
- Thus, for inserts $v^{new} = v^{old} \cup (i_r \bowtie s)$
- Similarly for deletes $v^{new} = v^{old} - (d_r \bowtie s)$

Selection and Projection Operations

- Selection: Consider a view $v = \sigma_\theta(r)$.
  - $v^{new} = v^{old} \cup \sigma_\theta(i_r)$
  - $v^{new} = v^{old} - \sigma_\theta(d_r)$

- Projection is a more difficult operation
  - *Example:* $R = (A,B)$, and r(R) =

a, 2
a, 3

$\Pi_A(r)$ has a single tuple (a).

If we delete the tuple (a,2) from r, we should not delete the tuple (a) from $\Pi_A(r)$, but if we then delete (a,3) as well, we should delete the tuple

  - For each tuple in a projection $\Pi_A(r)$, keep a count of how many times it was derived
  - On insert of a tuple to r, if the resultant tuple is already in $\Pi_A(r)$ we increment its count, else we add a new tuple with count = 1
  - On delete of a tuple from r, we decrement the count of the corresponding tuple in $\Pi_A(r)$
    - if the count becomes 0, we delete the tuple from $\Pi_A(r)$

336

© Nick Roussopoulos

Aggregation Operations

- count : $v = {}_A g_{count(B)}(r)$.
  - When a set of tuples i_r is inserted, for each tuple r in i_r, if the corresponding group is already present in v, we increment its count, else we add a new tuple with count = 1
  - When a set of tuples d_r is deleted, for each tuple t in i_r we look for the group *t.A* in *v*, and subtract 1 from the count for the group.
    - When the count becomes 0, we delete the tuple from *v* for the group *t.A*

- sum: $v = {}_A g_{sum\,(B)}(r)$
  - We maintain the sum in a manner similar to count, except we add/subtract the B value instead of adding/subtracting 1 for the count
  - Additionally we maintain the count in order to detect groups with no tuples. Such groups are deleted from v
    - Cannot simply test for sum = 0 (why?)

- sum: $v = {}_A g_{avg(B)}(r)$
  - we maintain the sum and count aggregate values separately, and divide at the end

337

© Nick Roussopoulos

Aggregate Operations (Cont.)

- Standard deviation: $v = {}_{A}g_{std(B)}(r)$
  - we maintain the sum of squares and the count
- min, max: $v = {}_{A}g_{min(B)}(r)$.
  - Handling insertions on r is straightforward.
  - Maintaining the aggregate values min and max on deletions may be more expensive. We have to look at the other tuples of *r* that are in the same group to find the new minimum
- Percentiles and other non-distributive computations the problems are more difficult.

338

Other Operations

- Set intersection: $v = r \cap s$
  - when a tuple is inserted in *r* we check if it is present in *s*, and if so we add it to *v*.
  - If the tuple is deleted from r, we delete it from the intersection if it is present.
  - Updates to *s* are symmetric
  - The other set operations, *union* and *set difference* are handled in a similar fashion.
- Outer joins are handled in much the same way as joins but with some extra work *(bookkeeping)*

339

Query Optimization w/ Materialized Views

- Rewriting queries to use materialized views:
  - A user submits a query $r \bowtie s \bowtie t$
  - A materialized view $v = r \bowtie s$ is available
  - We can rewrite the query as $v \bowtie t$
- Alternatives
  - Incremental update of the view and use
  - Recompute the view from its definition
  - Ignore the view and work with its base relations
- Query optimizer have been extended to consider all above alternatives and choose the best overall plan

Materialized View Selection

- Materialized view selection: "What is the best set of views to materialize?"
- Index selection: "what is the best set of indices to create"
  - closely related, to materialized view selection but simpler
- Materialized view selection and index selection based on typical system workload (queries and updates)
  - Typical goal: minimize time to execute workload , subject to constraints on space and time taken for some critical queries/updates
  - One of the steps in database tuning
- Commercial database systems provide tools (called "tuning assistants" or "wizards") to help the database administrator choose what indices and materialized views to create

Top-K Query Optimization

- Queries may generate more than what is needed in a to-K query (i.e. all joinable tuples) but you only need 10

__Example__

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

- Alternative 1: use sort-merge join and stop at 10
- Alternative 2: make an estimate on the highest r.A value *H* in the 10 result tuples and modify the query to:

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

  - If > 10 results, discard the extra (keep the 10)
  - If < 10 results, retry with larger *H*

342

Join Minimization

- Join minimization r(A,B), s(B,C,D)

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

- Check if join with s is redundant, drop it
  - E.g. join condition is on foreign key from r to s, r.B and no selection on s
    - select r.A, s2.B
      from r, s s1, s s2
      where r.B=s1.B and r.B = s2.B and s1.A < 20 and s2.A < 10
  - join with s1 is redundant and can be dropped (along with selection on s1)
  - Lots of research in this area 70s/80s!

343

Multiquery Optimization

- Example
  Q1: select * from (r natural join t) natural join s
  Q2: select * from (r natural join u) natural join s
  - Queries share common subexpression (r natural join s)
  - May be useful to compute (r natural join s) once and use it in both queries (factor out)
- Multiquery optimization: find best overall plan for a set of queries, exploiting sharing of common subexpressions between queries where it is useful
- Implies the queries are synchronous

Materialized Views Optimization

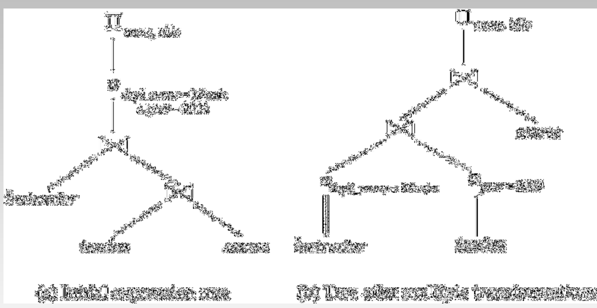
- Set of materialized views may share common subexpressions
  - view maintenance cost is shared
- The best approach is to maintain a Logical Access Path Schema (LAP schema) that captures all the relationships amongst the views
- Materialized view optimization/maintenance is amortized over a long period of time

Parametric Query Optimization

- **Example**
  select *
  from r natural join s
  where r.a < $1
  - value of parameter $1 not known at compile time - only at run time
  - different plans may be optimal for different values of $1
- **Solution 1: optimize at run time, each time query is submitted**
  - can be expensive
- **Solution 2: Parametric Query Optimization:**
  - optimizer generates a set of plans, optimal for different values of $1
    - Set of optimal plans usually small for 1 to 3 parameters
    - Key issue: how to do find set of optimal plans efficiently
  - best one from this set is chosen at run time when $1 is known
- **Solution 3: Query Plan Caching**
  - If some plan is likely to be optimal for all parameter values, the optimizer caches the plan and reuses it, else reoptimizes each time
  - Implemented in many database systems

13.2 Rule-Based Optimization

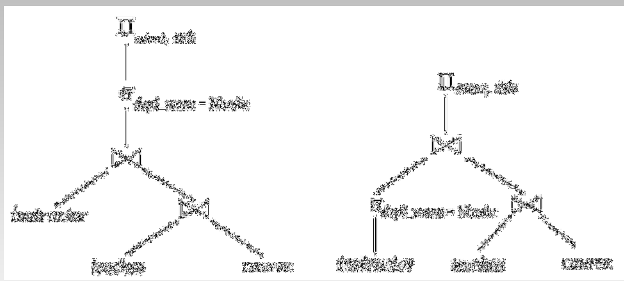
- **Query: Find the names of all instructors in the Music department, along with the titles of the courses that they taught in 2009.**



- **Guideline: filter results as early as possible**
  - Work with most selective predicates first
  - Push selections ahead of joins
  - Order joins

Equivalence of Relational Expressions

- Two expressions are equivalent if the result is the same (attributes and tuples)
  - even when tuples and attributes are ordered differently



- Which alternative is better?
  - The right one will generate less intermediate results. Is this always better?
    - for a single user environment and a single query, probably
    - In a multi-user multi-query optimization, maybe not

Equivalence of Rel. Expressions and Rules

- Two relational algebra expressions are said to be *equivalent* if on every legal database instance the two expressions generate the same set of tuples
  - Again: order of tuples is irrelevant
- 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 on every legal database instance the two expressions generate the same multiset of tuples
- 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.

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

2. Selection operations are commutative.

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

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

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

4. Selections can be combined with Cartesian products and theta joins.
   a. $\sigma_\theta(E_1 \times E_2) = E_1 \bowtie_\theta E_2$
   b. $\sigma_{\theta 1}(E_1 \bowtie_{\theta 2} E_2) = E_1 \bowtie_{\theta 1 \wedge \theta 2} E_2$

Equivalence Rules (Cont.)

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

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

6. (a) Natural join operations are associative:

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

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

$$(E_1 \bowtie_{\theta 1} E_2) \bowtie_{\theta 2 \wedge \theta 3} E_3 = E_1 \bowtie_{\theta 1 \wedge \theta 3} (E_2 \bowtie_{\theta 2} E_3)$$

where θ_2 involves attributes from only E_2 and E_3.

Equivalence Rules (Cont.)

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

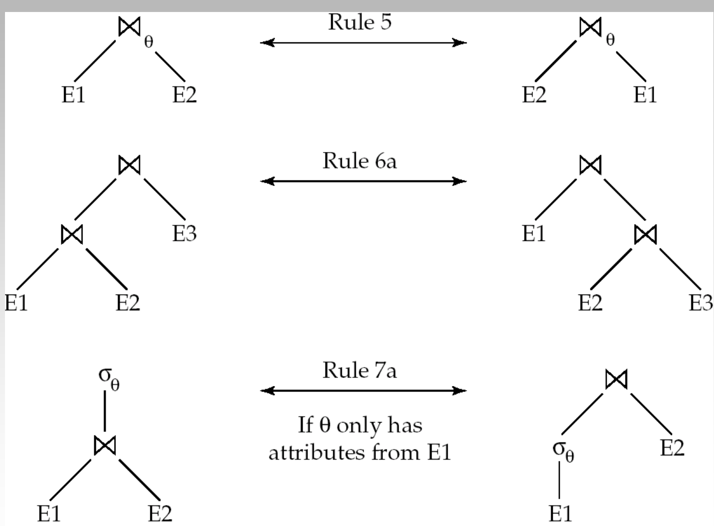
(a) When all the attributes in θ_0 involve only the attributes of one of the expressions (E_1) being joined.

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

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

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

Pictorial Depiction of Equivalence Rules



Equivalence Rules (Cont.)

9. The set operations union and intersection are commutative

$$E_1 \cup E_2 = E_2 \cup E_1$$
$$E_1 \cap E_2 = E_2 \cap E_1$$

- (set difference is not commutative).

10. Set union and intersection are associative.

$$(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$$
$$(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$$

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

$$\sigma_\theta (E_1 - E_2) = \sigma_\theta (E_1) - \sigma_\theta(E_2)$$

and similarly for $\cup$ and $\cap$ in place of $-$

Also: $\sigma_\theta (E_1 - E_2) = \sigma_\theta(E_1) - E_2$

and similarly for $\cap$ in place of $-$, but not for $\cup$

12. The projection operation distributes over union

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

354