Chapter 16: Query Optimization

Sistemas de Bases de Dados 2020/21

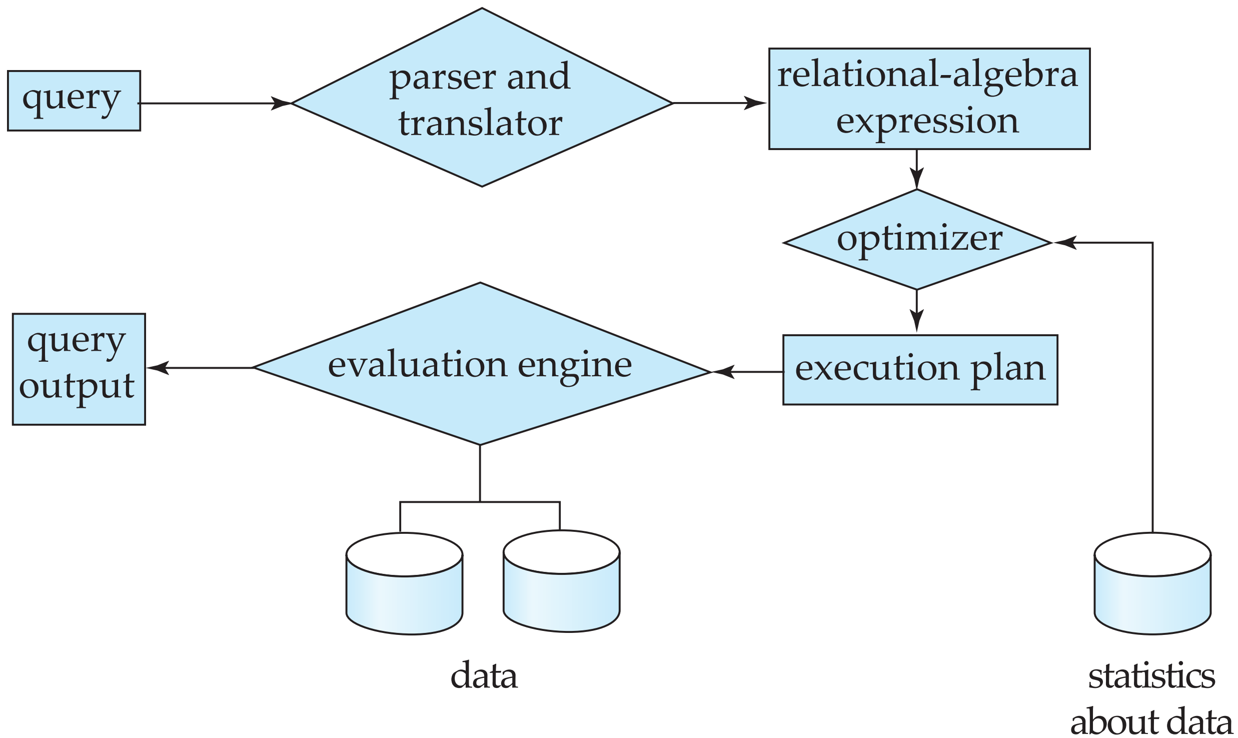
Capítulo refere-se a: Database System Concepts, 7th Ed

Outline

- Introduction
- Transformation of Relational Expressions
- Catalog Information for Cost Estimation
- Statistical Information for Cost Estimation
- Cost-based optimization
- Dynamic Programming for Choosing Evaluation Plans
- Join minimization, Materialized views and nested subqueries

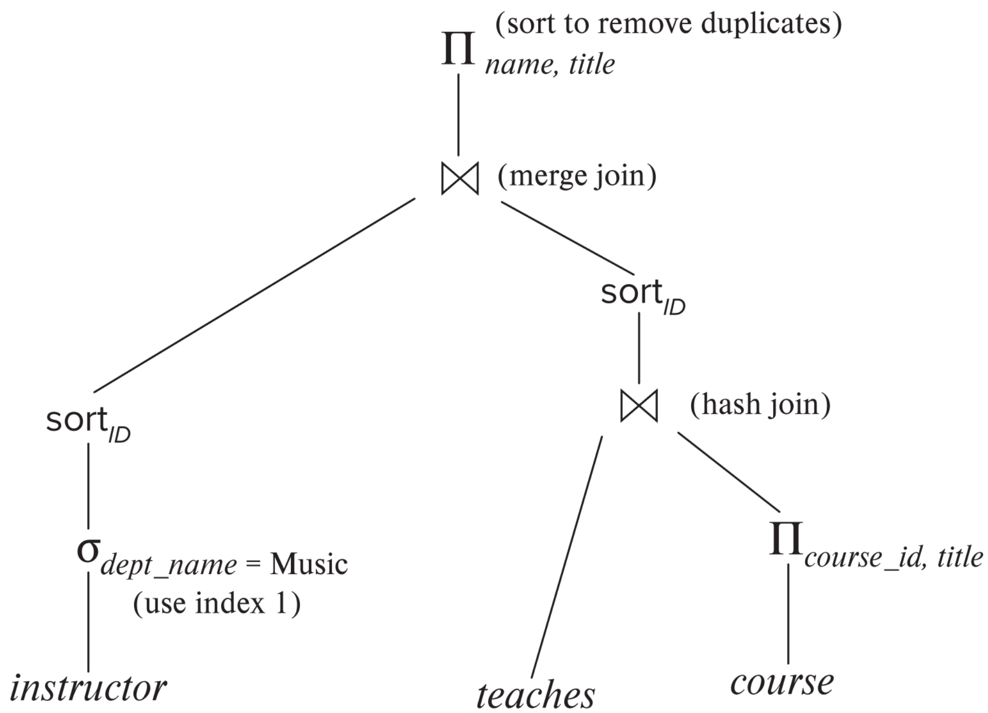
Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation



Introduction

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



Introduction (Cont.)

- 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**
  1. Generate logically equivalent expressions using **equivalence rules**
  2. Annotate resultant expressions to get alternative query plans
  3. Choose the cheapest plan based on **estimated cost**
- Estimation of plan cost based on:
  - Statistical information about relations. Examples:
    - number of tuples, number of distinct values for an attribute
  - Statistics estimation for intermediate results
    - to compute cost of complex expressions
  - Cost formulae for algorithms, computed using statistics

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 the computed and stored temporary relation (in case no pipelining is used) is smaller

- Query optimizers use equivalence rules to **systematically** generate expressions equivalent to the given expression
- Must consider the interaction of evaluation techniques when choosing evaluation plans
  - choosing the cheapest algorithm for each operation independently may not yield the best overall algorithm. E.g.
    - merge-join may be costlier than hash-join but may provide a sorted output which reduces the cost for an outer level aggregation (or a following join)
    - nested-loop join may provide opportunity for pipelining

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

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

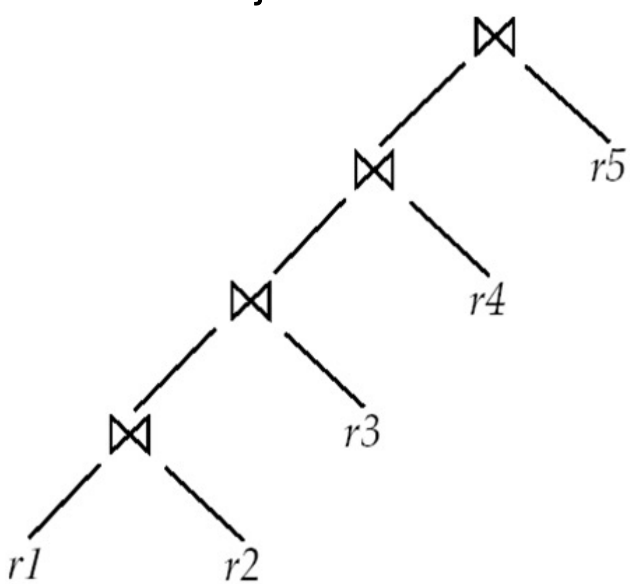
  but the result of the first join is likely to be a large relation.
- Only a small fraction of the university's instructors are likely to be from the Music department
  - it is better to compute

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

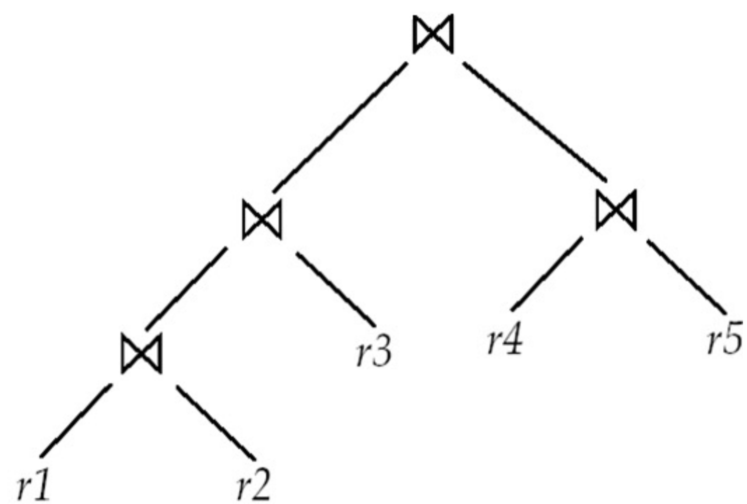
    first.

Dynamic Programming & Left Deep Join Trees

- To deal with the high combinatoric, Dynamic Programming may be used
- To trim the combinatoric use **left-deep join trees,** where 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

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.

- Local search (e.g. hill-climbing and genetic algorithms) may also be used for optimisation

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
- Even with the use of heuristics, cost-based query optimisation imposes a substantial overhead.
  - But is worth it for expensive queries in large datasets
  - Optimisers often use simple heuristics for very cheap queries, and perform exhaustive enumeration for more expensive queries
  - **The cost of optimisation is a function of the size of the query, whilst the gains are a functions of the size of the dataset**

Statistics for Cost Estimation

Sistemas de Bases de Dados 2020/21

Capítulo refere-se a: Database System Concepts, 7th Ed

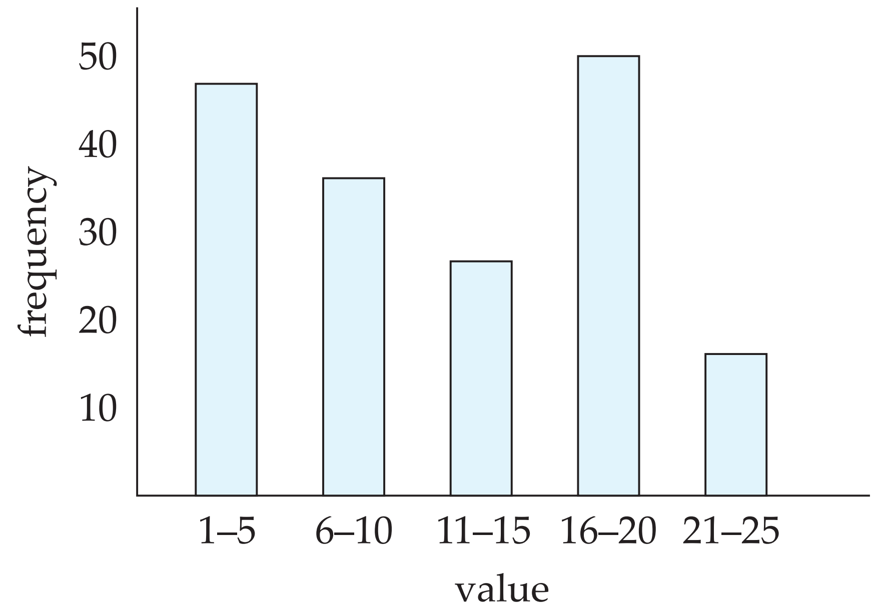
Statistical Information for Cost Estimation

- 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 are usually computed based on a **random sample**
- Statistics may be out of date
  - Some database require an **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

Estimating Statistical Information

- Statistical information about exiting tables (and indices) can be used to estimate the cost of the algorithms for relational algebra operators
  - (we've seen that working in practice)

- But what if an operator in an execution plan takes as input the result of another operation (rather than a table that directly belongs to the database)?
  - In this case, we need to have statistics about the "intermediate table"
  - How can we obtain such statistics?

- Estimate those statistics, based on the operator and the statistics stored for the original tables.
- I.e. besides estimating the costs, for each operator estimate also:
  - Number of tuples of the result of the operation
  - Size of tuples, number of distinct value per attribute, histograms, etc, of the result of the operation

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 = $n_r \cdot \dfrac{v - \min(A,r)}{\max(A,r) - \min(A,r)}$
  - 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 $\bowtie$ *takes*

Catalog information for join examples:

- $n_{student}$ = 5,000.
- $f_{student}$ = 50, which implies that
  $b_{student}$ =5000/50 = 100.
- n_{takes} = 10000.
- f_{takes} = 25, which implies that
  b_{takes} = 10000/25 = 400.
- *V(ID, takes)* = 2500, which implies that on average, each student who has taken a course has taken 4 courses.
  - Attribute *ID* in *takes* is a foreign key referencing *student.*
  - *V*(*ID, student*) = 5000 (*primary key!*)

Estimation of the Size of Joins

- 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 $R \cap S$ *in* S is a foreign key in S referencing R, then the number of tuples in $r \bowtie s$ is exactly the same as the number of tuples in s.
  - The case for $R \cap S$ being a foreign key referencing S is symmetric.
- In the example query *student* $\bowtie$ *takes*, *ID* in *takes* is a foreign key referencing *student*
  - hence, the 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 $R \bowtie S$, 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.
- We can improve on above if histograms are available
  - Use formula like 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 of Outer Joins

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

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 $r - s = r$.
    - <u>All the three estimates may be quite inaccurate but provide upper bounds on the sizes</u>.

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 $\prod_{A}(r)$ 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*)

Additional Optimisation Techniques

Join Minimisation

- **Join minimization**

```
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 is declared as not null, and no selection on s
  - Other sufficient conditions possible

    ```
    select r.A, s2.B
    from r, s as s1, s as 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)

Top-K Queries

- **Top-K queries**

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

  - Alternative 1: Indexed nested loops join with r as outer
  - Alternative 2: estimate highest r.A value in result and add selection (**and** r.A <= H) to where clause
    - If < 10 results, retry with larger H

Optimizing Nested Subqueries

- Nested query example:

  select *name*
  from *instructor*
  where exists (**select** *
  from *teaches*
  where *instructor.ID* = *teaches.ID* **and** *teaches.year = 2021*)

- 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
  select *instructor.name*
  from *instructor, teaches*
  where *instructor.ID = teaches.ID* **and** *teaches.year = 2021*
- In general, it is not possible/straightforward to move the entire nested subquery into the outer level query
  - A view is created instead, and used in the body of the outer level query

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: **with** t_1 as
  (**select distinct** V
  from $s_1, s_2, \ldots, s_m$
  where P_2^1)
  select …
  from $r_1, r_2, \ldots, r_n, t_1$
  where P_1 **and** P_2^2
  - P_2^1 contains predicates that do not involve any correlation variables
  - P_2^2 contains predicates involving correlation variables
  - V contains all attributes used in predicates with correlation variables

Optimizing Nested Subqueries (Cont.)

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

  with t_1 **as**
  (**select distinct** *ID*
  from *teaches*
  where *year = 2021)*
  select *name*
  from *instructor*, t_1
  where t_1.*ID = instructor.ID*

- The process of replacing a nested query by a query with a join (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 group-by/aggregation in some cases. E.g.

- **select** *name*
  from *instructor*
  where 1 < (**select count**(*)
  from *teaches*
  where *instructor.ID = teaches.ID*
  and *teaches.year = 2021*)

can be transformed into

- **with** *t* **as**
  (**select** *ID, count(*)* **as** *cnt*
  from *teaches*
  where *teaches.year = 2021*
  group by *ID*)
  select *name*
  from *instructor, t*
  where *instructor.ID = t.ID* **and** *cnt > 1*)

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

Materialized View Maintenance

- The task of keeping a materialized view up-to-date with the underlying data is known as **materialized view maintenance**
- Materialized views can be maintained by recomputation on every update
- A 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)
  - Incremental maintenance supported by many database systems
    - Avoids manual effort/correctness issues

Incremental View Maintenance

- The changes (inserts and deletes) to a relation or expressions are referred to as its **differential**
  - Set of tuples inserted to and deleted from r are denoted $\mathbf{i_r}$ and $\mathbf{d_r}$
- To simplify, 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
- We then outline how to handle relational algebra expressions

Join Operation

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

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

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
  - $R = (A,B)$, and r(R) = { (a,2), (a,3)}
  - $\prod_A(r)$ has a single tuple (a).
  - If we delete the tuple (a,2) from r, we should not delete the tuple (a) from $\prod_A(r)$, but if we then delete (a,3) as well, we should delete the tuple
- For each tuple in a projection $\prod_A(r)$, we keep a count of how many times it was derived
  - On insert of a tuple to r, if the resultant tuple is already in $\prod_A(r)$ we increment its count, else we add a new tuple with count = 1
  - On delete of a tuple from r, we decrement the count of the corresponding tuple in $\prod_A(r)$
    - if the count becomes 0, we delete the tuple from $\prod_A(r)$

Aggregation Operations

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

Aggregate Operations (Cont.)

- **Avg**:
  - Maintain **sum** and **count** separately, and divide at the end
- **min**, **max**: $v = {}_{A}\gamma_{\ min\ (B)}\ (r)$.
  - 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

Other Operations

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

Handling Expressions

- To handle an entire expression, we derive expressions for computing the incremental change to the result of each sub-expressions, starting from the smallest sub-expressions.
- E.g., consider $E_1 \bowtie E_2$ where each of E_1 and E_2 may be a complex expression
  - Suppose the set of tuples to be inserted into E_1 is given by D_1
    - Computed earlier, since smaller sub-expressions are handled first
  - Then the set of tuples to be inserted into $E_1 \bowtie E_2$ is given by $D_1 \bowtie E_2$
    - This is just the usual way of maintaining joins

Query Optimization and Materialized Views

- Rewriting queries to use materialized views:
  - A materialized view $v = r \bowtie s$ is available
  - A user submits a query $r \bowtie s \bowtie t$
  - We can rewrite the query as $v \bowtie t$
    - Whether to do so depends on cost estimates for the two alternative
- Replacing a use of a materialized view by the view definition:
  - A materialized view $v = r \bowtie s$ is available, but without any index on it
  - User submits a query $\sigma_{A=10}(v)$.
  - Suppose also that s has an index on the common attribute B, and r has an index on attribute A.
  - The best plan for this query may be to replace v by $r \bowtie s$, which can lead to the query plan $\sigma_{A=10}(r) \bowtie s$
- Query optimizer should be extended to consider all above alternatives 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 should be 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
    - more on tuning, later
- Commercial database systems provide tools (called “tuning assistants” or “wizards”) to help the database administrator choose what indices and materialized views to create

Multiquery Optimization

- Example

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

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

  - Both queries share common subexpression (r natural join s)
  - May be useful to compute (r natural join s) once and use it in both queries
    - But this may be more expensive in some situations
      - e.g. (r natural join s) may be expensive, plans as shown in queries may be cheaper
- **Multiquery optimization**: find best overall plan for a set of queries, exploiting sharing of common subexpressions between queries where it is useful

Multiquery Optimization (Cont.)

- Simple heuristic used in some database systems:
  - optimize each query separately
  - detect and exploiting common subexpressions in the individual optimal query plans
    - May not always give best plan, but is cheap to implement
  - **Shared scans**: widely used special case of multiquery optimization
- Set of materialized views may share common subexpressions
  - As a result, view maintenance plans may share subexpressions
  - Multiquery optimization can be useful in such situations

Parametric Query Optimization

- Example
  select *
  from r **natural join** s
  where r.a < $1
  - value of parameter $1 not known at compile time
    - known only at run time
  - different plans may be optimal for different values of $1
- Solution 1: optimize at run time, each time the 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 find a 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 optimizer decides that same plan is likely to be optimal for all parameter values, it caches plan and reuses it, else reoptimize each time
  - Implemented in many database systems

Plan Stability Across Optimizer Changes

- What if 95% of plans are faster on database/optimizer version N+1 than on N, but 5% are slower?
  - Why should plans be slower on new improved optimizer?
    - Answer: Two wrongs can make a right, fixing one wrong can make things worse!
- Approaches:
  - Allow hints for tuning queries
    - Not practical for migrating large systems with no access to source code
  - Set optimization level, default to version N (Oracle)
    - And migrate one query at a time after testing both plans on new optimizer
  - Save plan from version N, and give it to optimizer version N+1
    - Sybase, XML representation of plans (SQL Server)

Adaptive Query Processing

- Some systems support adaptive operators that change execution algorithm on the fly
  - E.g., (indexed) nested loops join or hash join chosen at run time, depending on size of outer input
- Other systems allow monitoring of behavior of plan at run time and adapt plan
  - E.g., if statistics such as number of rows is found to be very different in reality from what optimizer estimated
  - Can stop execution, compute fresh plan, and restart
    - But must avoid too many such restarts