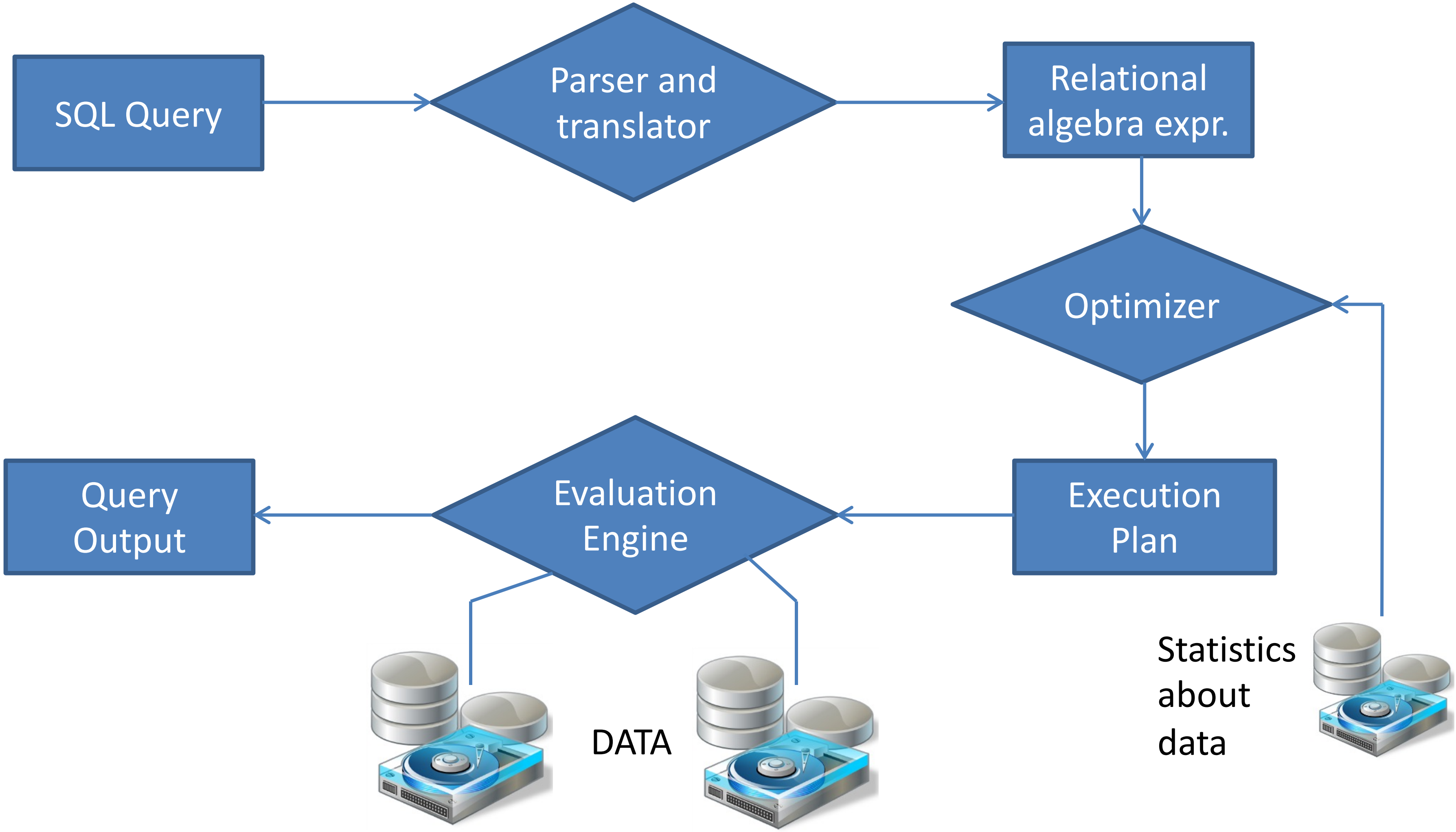
CSC 553
Advanced Database Concepts
Lecture 6

Alexander Rasin
College of CDM, DePaul University

May 2^{nd}, 2022

Query Processing Steps



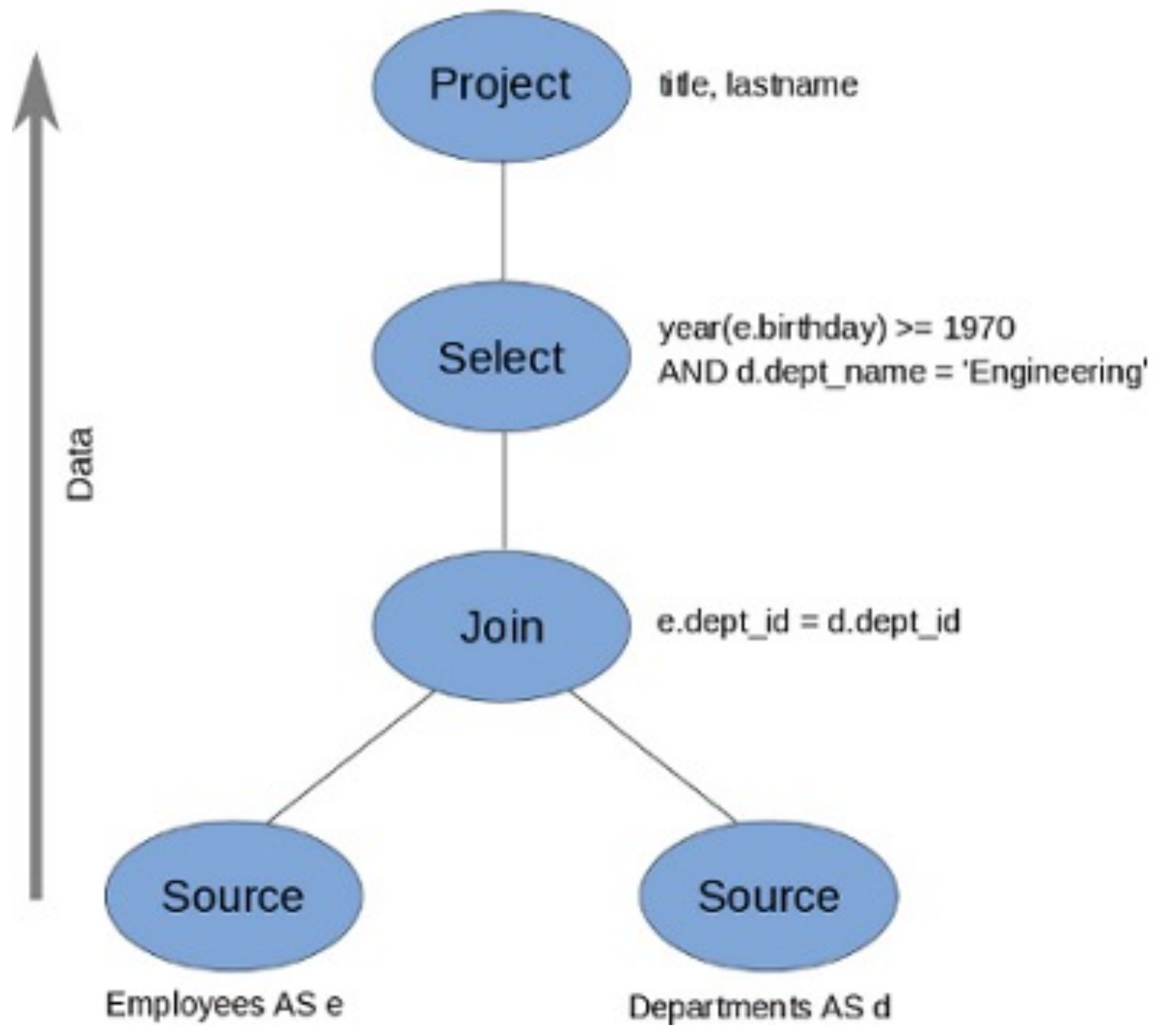
Optimization Fundamentals

- Relational algebra expressions can be substituted for other equivalent expressions
  - E.g., $\sigma_{salary<75000}(\pi_{salary}(instructor))$ is equivalent to $\pi_{salary}(\sigma_{salary<75000}(instructor))$
- Each relational algebra operation (such as $\sigma_{salary<75000}$) can be evaluated using several different algorithms
  - Therefore, a full relational-algebra expression can be evaluated in many ways

Query Optimization Options

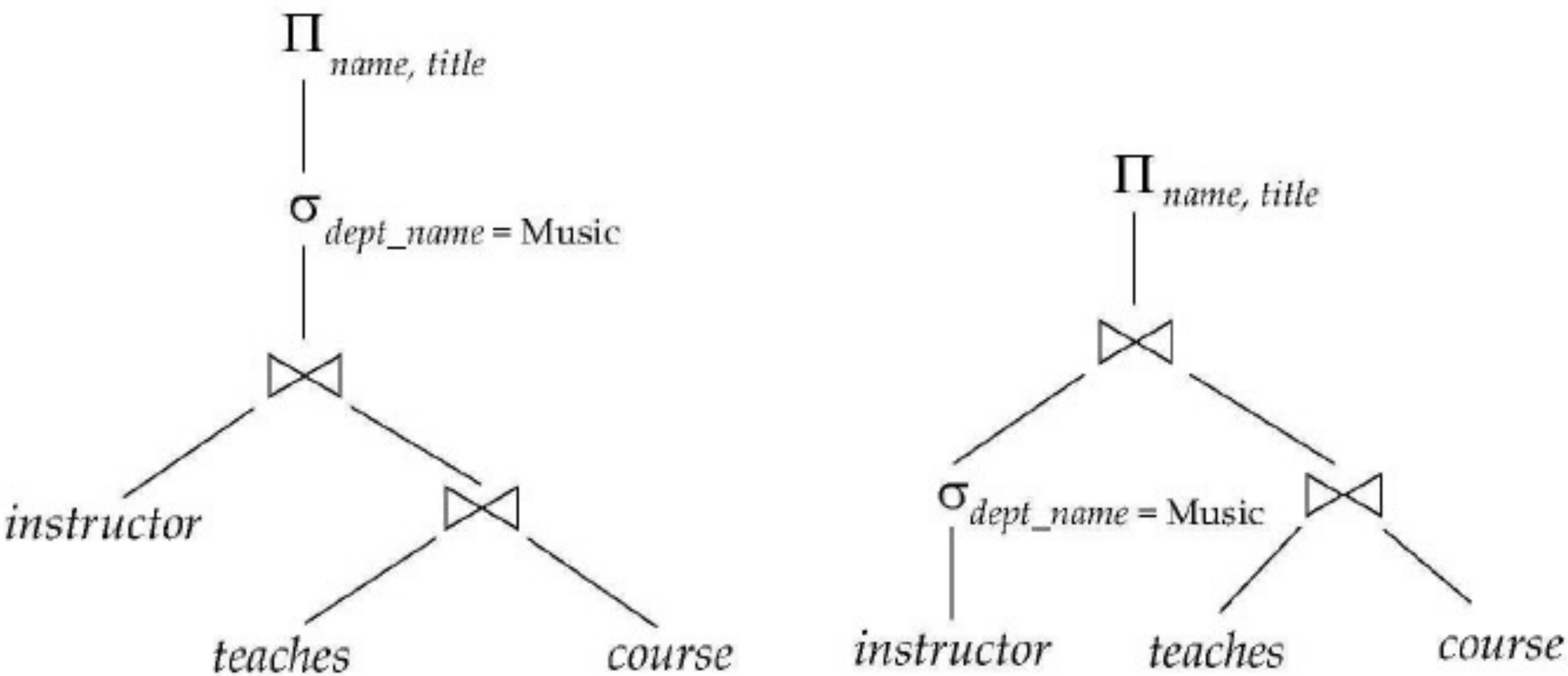
- Consider the relational algebra operator such as $\sigma_{salary<75000}$(instructors) and the evaluation options
  - Use an index on salary (if any) to find instructors who make less than 75000
  - Perform a complete relation scan and discard instructors with salary ≥ 75000
- Annotated expression specifying the evaluation strategy is called <u>evaluation-plan</u>

Query Plans


Data
Project
title, lastname
Select
year(e.birthday) >= 1970
AND d.dept_name = 'Engineering'
Join
e.dept_id = d.dept_id
Source
Source
Employees AS e
Departments AS d

The Intuition

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



Equivalence Rules

- Conjunctive selection operations can be deconstructed into a sequence of individual selections
  - $\sigma_{\text{cond1 and cond2}}(E) = \sigma_{\text{cond1}}(\sigma_{\text{cond2}}(E))$
- Selection operations are commutative
  - $\sigma_{\text{cond1}}(\sigma_{\text{cond2}}(E)) = \sigma_{\text{cond2}}(\sigma_{\text{cond1}}(E))$

Equivalence Rules

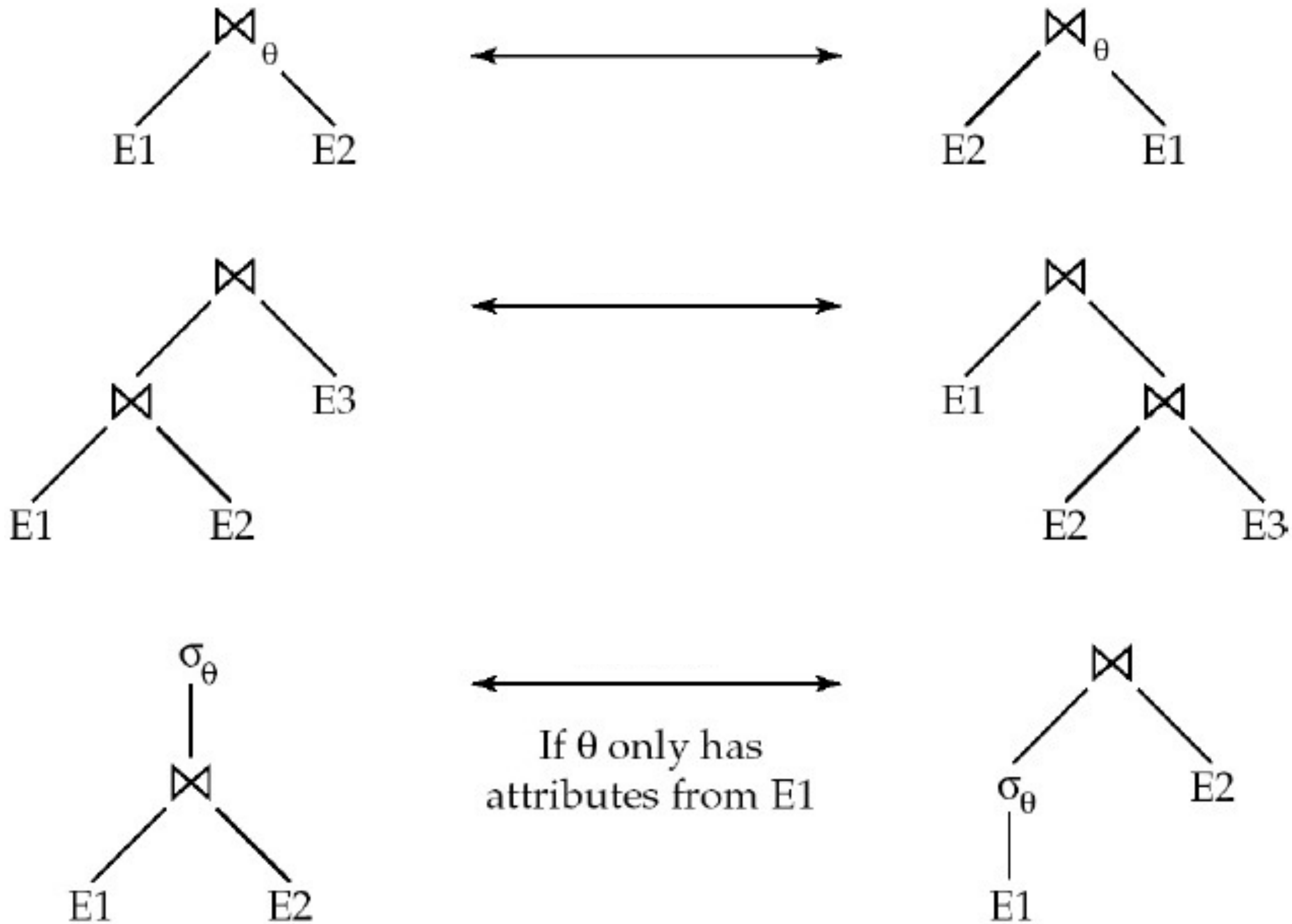
- Only the last in a sequence of projection operations is needed, the others can be omitted
  - $\Pi_{L1}(\Pi_{L2}(\ldots(\Pi_{Ln}(E)))) = \Pi_{L1}(E)$
- Selections can be combined with Cartesian products and joins
  - $\sigma_{cond1}(E_1 \text{ X } E_2) = E_1 \bowtie_{cond1} E_2$
  - $\sigma_{cond1}(E_1 \bowtie_{cond2} E_2) = E_1 \bowtie_{cond1 \text{ and } cond2} E_2$

Equivalence Rules

- Joins are commutative
  - $E_1 \bowtie_{cond1} E_2 = E_2 \bowtie_{cond1} E_1$
- Natural joins are associative
  - $(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$
  - (E1 join E2) join E3 = E1 join (E2 join E3)
- Joins are associative with a condition
  - $(E_1 \bowtie_{cond1} E_2) \bowtie_{cond2\ \mathbf{and}\ cond3} E_3 =$
    $E_1 \bowtie_{cond1\ and\ cond3} (E_2 \bowtie_{cond2} E_3)$

  where $cond_2$ involves attributes from only E2/E3

Equivalence Rules



Equivalence Rules

- 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
- Set union and intersection are associative
  - $(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$
  - $(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$

Equivalence Rules

- Selection operation distributes over U and ∩ and –
  - $\sigma_{cond}(E_1 - E_2) = \sigma_{cond}(E_1) - \sigma_{cond}(E_2)$
  - Similar for U and ∩
- Also
  - $\sigma_{cond}(E_1 - E_2) = \sigma_{cond}(E_1) - E_2$
  - Similar for ∩ in place of –, but not for U
- Projection operation distributes over union
  - $\prod_L(E_1 \cup E_2) = (\prod_L(E_1) \cup \prod_L(E_2))$

Transformation Example: Pushing Selections

- Query: find the names of instructors in the Music department along with the titles of the courses they teach
  - $\Pi_{\text{name, title}}\ (\sigma_{\text{dept\_name=“Music”}}\ (\text{instructor} \bowtie\ (\text{teaches} \bowtie \Pi_{\text{course\_id,title}}\ (\text{course}))))$
- Transformation
  - $\Pi_{\text{name, title}}\ ((\sigma_{\text{dept\_name=“Music”}}\ (\text{instructor}) \bowtie\ (\text{teaches} \bowtie \Pi_{\text{course\_id,title}}\ (\text{course})))$
- Performing the selection as early as possible reduces the size of the relation to be joined

Multiple-Transformation Example

- Find the names of all instructors in the Music department who have taught a course in 2009, along with the titles of the courses they taught
  - $\Pi_{name,\ title}\ (\sigma_{dept\_name=\text{“Music”}\ and\ year\ =\ 2009}$ $(instructor \bowtie (teaches \bowtie \Pi_{course\_id,title}\ (course))))$
- Transform using join associativity
  - $\Pi_{name,\ title}\ (\sigma_{dept\_name=\text{“Music”}\ and\ year\ =\ 2009}$ $((instructor \bowtie teaches) \bowtie \Pi_{course\_id,title}\ (course)))$

Multiple-Transformation Example

- Result of 1st transformation
  - $\Pi_{name,\ title}\ (\sigma_{dept\_name=\text{"Music"} \text{ and } year\ =\ 2009}\ (instructor \bowtie teaches) \bowtie \Pi_{course\_id,title}\ (course)))$
- Now transform the sub-expression
  - $\sigma_{dept\_name=\text{"Music"} \text{ and } year\ =\ 2009}(instructor \bowtie teaches)$

  Into

  - $\sigma_{dept\_name=\text{"Music"}}\ (instructor) \bowtie \sigma_{year\ =\ 2009}\ (teaches)$

  Results in

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

Transformation: Pushing Projections

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

  We obtain a relation with schema

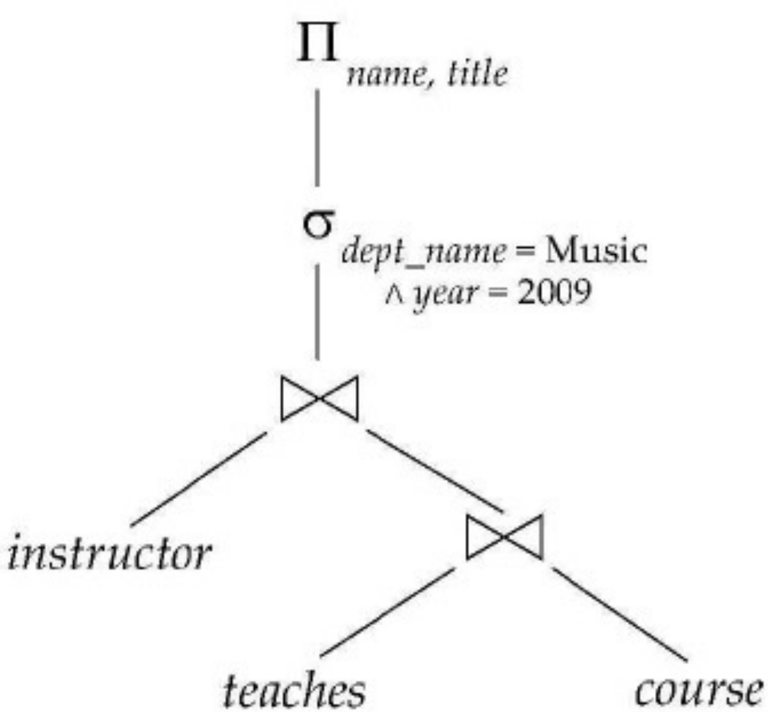
  (ID, name, dept_name, salary, course_id, sec_id, semester, year)

  – Push projections using equivalence rules to eliminate unneeded attributes

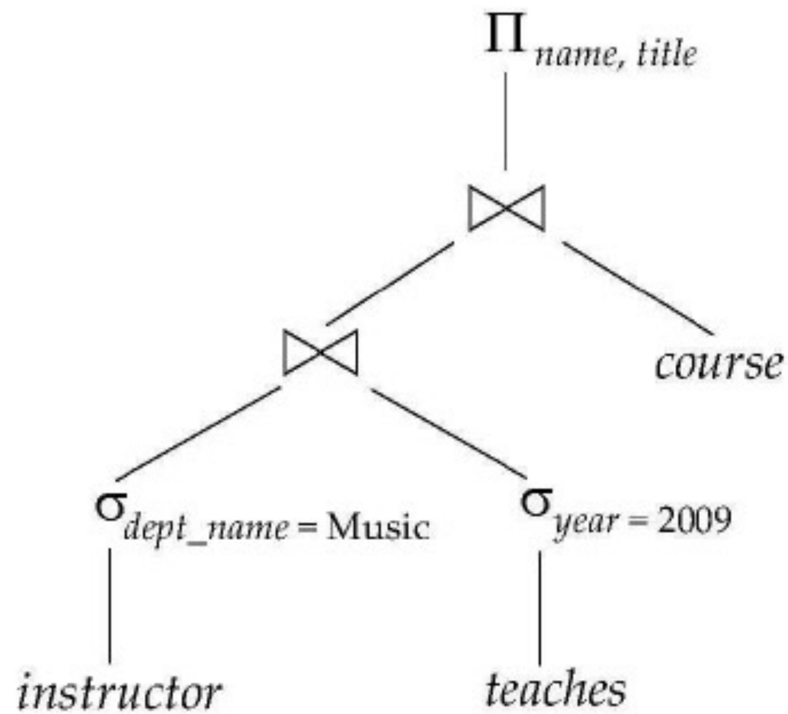
  – $\Pi_{\text{name, title}}\,(\Pi_{\text{name, corse\_id}}\,(\sigma_{\text{dept\_name=“Music”}}\,(\text{instructor}) \bowtie \text{teaches}) \bowtie \Pi_{\text{course\_id,title}}\,(\text{course}))))$

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

Multiple-Transformation Result

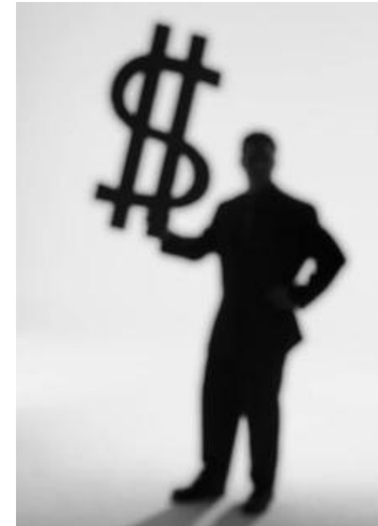


(a) Initial expression tree



(b) Tree after multiple transformations

Statistical Information for Cost Estimation



- n_r: number of tuples in a relation r
- b_r: number of blocks in relation r
- l_r: size of a tuple of r
- f_r: blocking factor of r – i.e. the number of tuples 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 $b_r = \lceil n_r/f_r \rceil$

Estimating Size of a Projection

- R(a, b, c)
  - a, b = 4bytes, c = 100bytes
  - Header = 12 bytes, total of 120 bytes
  - If block is 1024, can fit up to 8 tuples per block
- $\pi_{a+b=>x,c}(R)$
  - 116 bytes instead, still only 8 tuples per block
- $\pi_{a,b}$ (R)
  - 20 bytes, 50 tuples per block

Selection Size Estimation

- $\sigma_{A=v}(r)$
  - $T(R) / V(A,r)$:
    - number of records that will satisfy the selection (assuming uniform distribution)
  - Equality condition on a key attribute
    - size estimate = 1

Selection Size Estimation

- $\sigma_{A \leq v}(r)$ case of ($\sigma_{A \geq v}(r)$ is symmetric)
  - Let c denote the estimated number of tuples satisfying this condition
  - If min(A,r) and max(A, r) are available in catalog
    - c = 0 if v < min(A, r)
    - c = T(R) * ((v-min(A,r)) / max(A, r) – min(A, r))
    - If a histogram is available, refine the above estimate
    - In absence of statistical information c is assumed to be T(R) /3.

Size Estimation of Complex Selections

- The selectivity of a condition $cond_i$ is the probability that the tuple in the relation r satisfies $cond_i$
  - If s_i is the number of satisfying tuples in r, the selectivity of $cond_i$ is given by $s_i / T(R)$
- Conjunction: $\sigma_{cond_1 \text{ and } cond_2 \text{ and } \ldots\ cond_n}(r)$
  - Assuming independence, estimate of tuples in the result is $n_r * (s_1 * s_2 * \ldots s_n) / (T(R))^n$

Size Estimation of Complex Selections

- Disjunction: $\sigma_{\text{cond}_1 \text{ or } \text{cond}_2 \text{ or } \dots \text{ cond}_n}(r)$
  - Estimated number of tuples is:
  - $T(R) * (1 - (1 - s_1/T(R)) * (1 - s_2/ T(R)) * \dots (1 - s_n/ T(R)))$
- Negation: $\sigma_{\text{not cond}_1}(r)$
  - Estimated number of tuples
  - $T(R) - \text{size}(\sigma_{\text{cond}_1}(r))$

Estimating Join Cost

Join Operation Example

student ⋈ enrolled

- Catalog information:
  - $n_{student} = 5000$
  - $f_{student} = 50$ (meaning that $b_{student} = 5000/50 = 100$)
  - $n_{enrolled} = 10000$
  - $f_{enrolled} = 25$ ($b_{enrolled} = 10000/25 = 400$)
  - V(ID, enrolled) = 2500 which implies that on average each student is enrolled in 4 courses
    - Attribute ID is a foreign key referencing student

V(ID, student) = ?

Estimation of the Size of Joins

- The Cartesian product r x s contains T(R)xT(S) tuples; each tuple occupies $s_r + s_s$ bytes
  - R is the set of attributes of r
  - S is the set of attributes for s
- 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

Estimation of the Size of Joins

- If $R \cap S$ is a foreign key in S referencing R, then the number of tuples in $r \bowtie s$ is exactly same as the number of tuples in s
  - The case of $R \cap S$ being a foreign key referencing S is symmetric
- In the example query student $\bowtie$ enrolled, ID in enrolled is a foreign key referencing student
  - Hence, the result has exactly $n_{enrolled}$ tuples which is 10000

Estimation of the Size of Joins

- If R ∩ S = [A] (set of attributes)
- If we assume that every tuple t in r produces tuples in r ⋈ s, the number of tuples in r ⋈ s is estimated to be : $(n_r * n_s) / V(A, s)$

  (If the reverse is true, the estimate is $(n_r*n_s) / V(A, r)$
- Different if V(A, r) ≠ V(A, s) => dangling tuples
  - Lower of two estimates is probably more accurate
- Can improve on above with histograms…
  - Use formula similar to above, for each cell of histograms of the two relations

Estimation of the Size of Joins

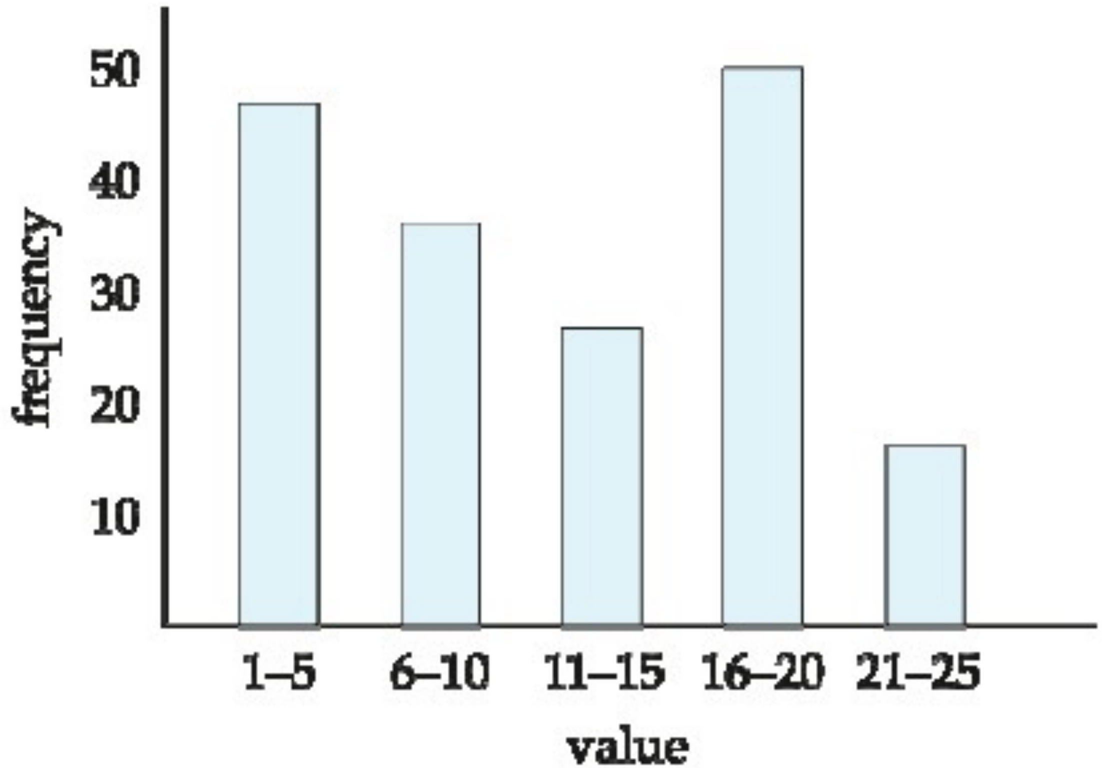
- Compute size estimates for student ⋈ enrolled (without using information about foreign keys)
  - V(ID, enrolled) = 2500 and
  - V(ID, student) = 5000
  - Two estimates are 5000 * 10000/2500 = 20K
  - and 10000* 5000/5000 = 10K
  - We choose the lower estimate, which in this case is the same as our earlier computation using foreign keys

Estimating Size of Set Operations

- Union
  - Can be T(R) and T(R)+T(S)
- Intersection
  - Can be between 0 and T(S)
- Difference
  - Can be between T(R) and T(R)-T(S)
- Duplicate Elimination
  - Smaller of T(R)/2 and product of all V(R, a_i)

Histograms

- Histogram on attribute age of relation person



  - Equi-width histograms
  - Equi-height histograms

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)
- If $r_2 \bowtie r_3$ is 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

- Consider the expression
  - $\Pi_{name,\ title}\ (\sigma_{dept\_name=\text{“Music”}}\ (instructor) \bowtie teaches) \bowtie \Pi_{course\_id,title}\ (course))))$
- Could compute $teaches \bowtie \Pi_{course\_id,title}\ (course)$ and then join result with
  - $\sigma_{dept\_name=\text{“Music”}}\ (instructor)$

  but the result of the first join will likely 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

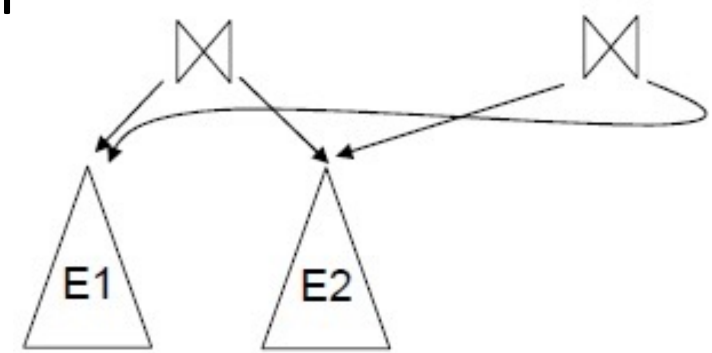
    $\sigma_{dept\_name=\text{“Music”}}\ (instructor) \bowtie teaches$ first

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
  - Apply all applicable equivalence rules on every sub-expression of every equivalent expression found so far
  - Add newly generated expressions to the set of equivalent expressions
  - Until no equivalent expressions are generated
- This approach is very expensive (space and time)

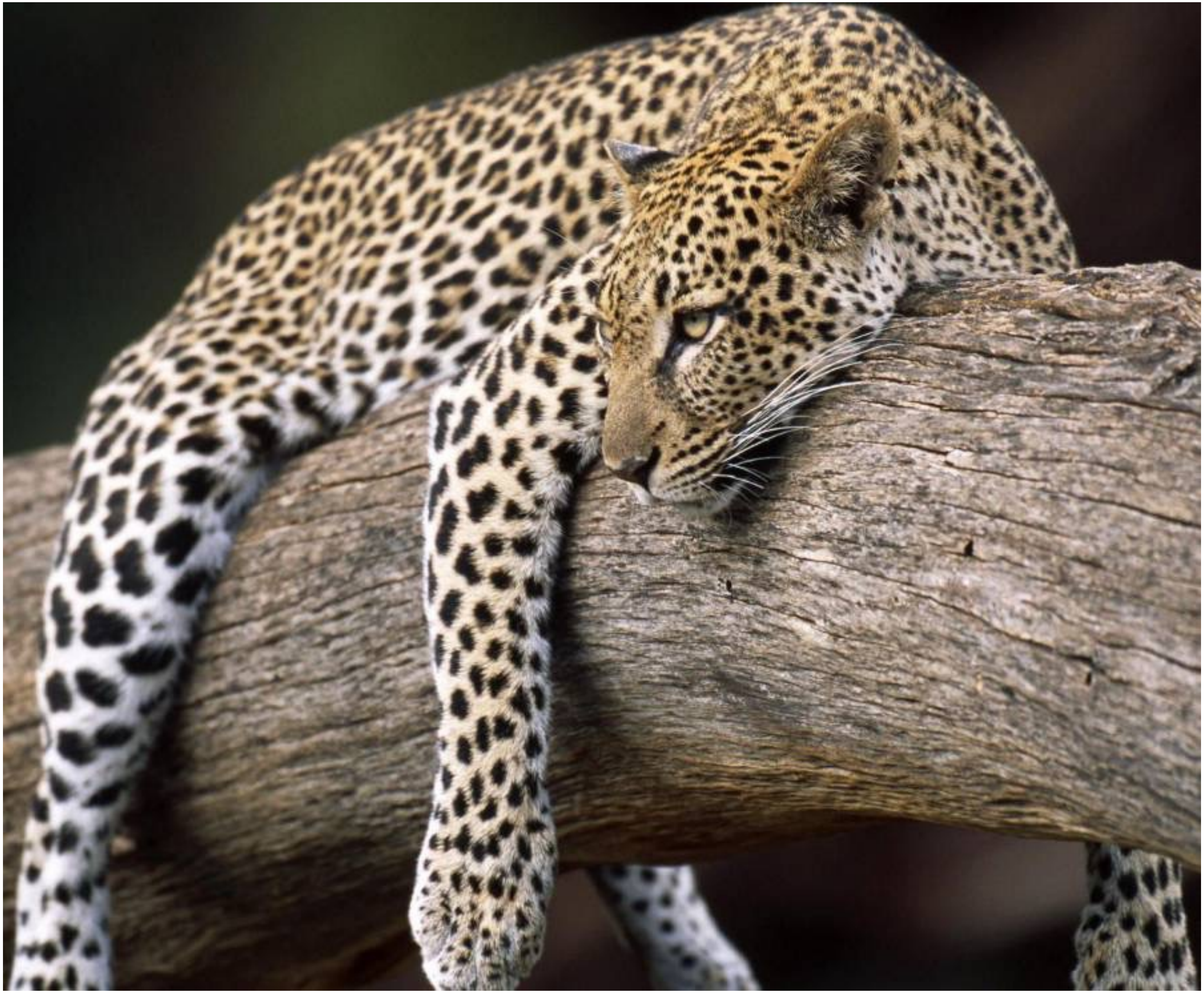
Implementing Transformation Based Optimization

- Space requirements are reduced by sharing common sub expressions
  - When E1 is generated from E2 by an equivalence rule, usually on top level is different, while sub-trees below be shared using pointers
    - E.g., when applying join commutativity



- Same sub-expressions may appear multiple times
  - Detect duplicate sub-expressions and share one copy

A Break!



Cost Estimation

- Cost of each operator computed
  - Need statistics of input relations
    - E.g., number of tuples, sizes of attributes
- Inputs can be results of sub-expressions
  - Need to estimate statistics of expressions results
  - To do so, we require additional statistics
    - E.g., number of distinct values for an attribute

Choice of Evaluation Plans

- Must consider the interaction of evaluation techniques when choosing evaluation plans
  - Choosing the cheapest algorithm for each algorithm 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
- Query optimizers incorporate elements of the following general approaches
  - Search all the plans and choose the best plan in a cost-based fashion
  - Use heuristics to choose a plan

Cost-Based Optimization

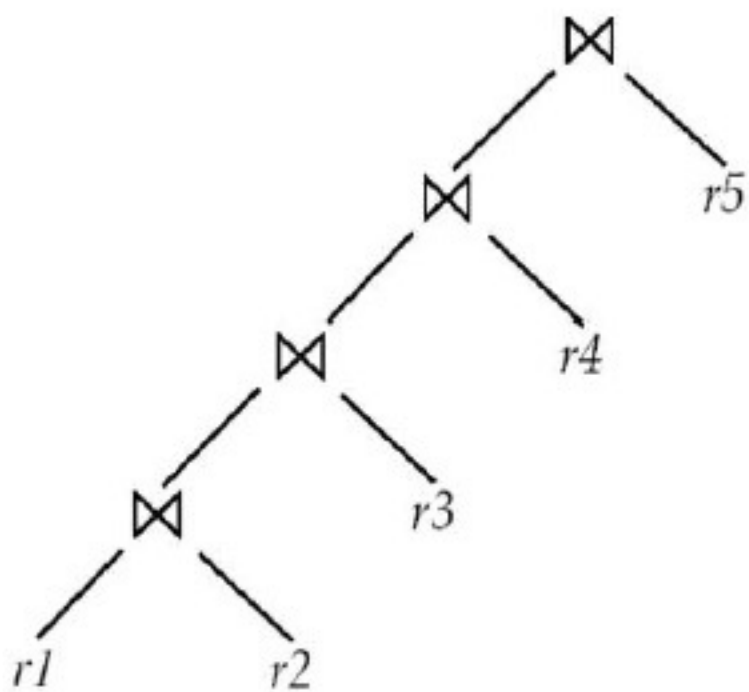
- Consider finding the best join-order for $r_1 \bowtie r_2 \bowtie r_3 \bowtie \ldots r_n$
- There are (2(n-1))!/(n-1)! Different join orders for above expression
  - With n = 7, the number is 665,280
  - With n=10, the number is greater than 176 billion!
- No need to generate all the join orders:
  - Using dynamic programming, the least-cost of join order for any subset of (r_1, r_2, …r_n} is computed only once and stored for future use

Dynamic Programming Optimization

- To find the best join tree for a set of n relations
  - Consider all possible plans of the form S_1 join (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 of R_i
  - When plan for any subset is computed, store it and reuse it when it is required again, instead of re-computing it
    - Dynamic programming

Left Deep Join Trees

- In <u>left-deep join trees</u>, the right-hand-side input for each join is a relation, rather than a result of another (intermediate) join



(a) Left-deep join tree

(b) Non-left-deep join tree

Interesting Sort Orders

- Consider the expression $(r_1 \bowtie r_2) \bowtie r_3$
  - Using common attribute A
- An <u>interesting sort order</u> is a particular sort order of tuples that could be useful for a later operation
  - Using merge-join to compute $r_1 \bowtie r_2$ may be costlier than hash join but generates result sorted on A
  - Which, in turn, may make merge-join with r_3 cheaper, which may reduce cost of join with r_3 and minimizing the overall cost
  - Sort order may also be useful for order by and for group by

Interesting Sort Orders

- Expression $(r_1 \bowtie r_2) \bowtie r_3$
  - At least 2 interesting sort orders, “none” and “sorted on A”
- 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

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 always) improve execution performance
  - Perform selection early (reduces number of tuples)
  - Perform projection early (reduces the number of attributes)
  - Perform most restriction selection and join operations (i.e. smallest result size) before other similar operations
  - Some systems use only heuristics, other combine heuristics with partial cost-based optimization

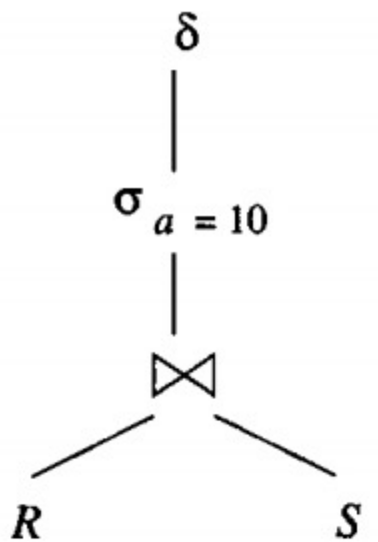
Structure of Query Optimizers

- Many optimizers consider only left-deep join orders
  - Plus heuristics to push selections and projections down the query tree
  - Reduces optimization complexity and generates plans amenable to pipelined evaluation
- 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
  - Nested sub-queries

Structure of Query Optimizers

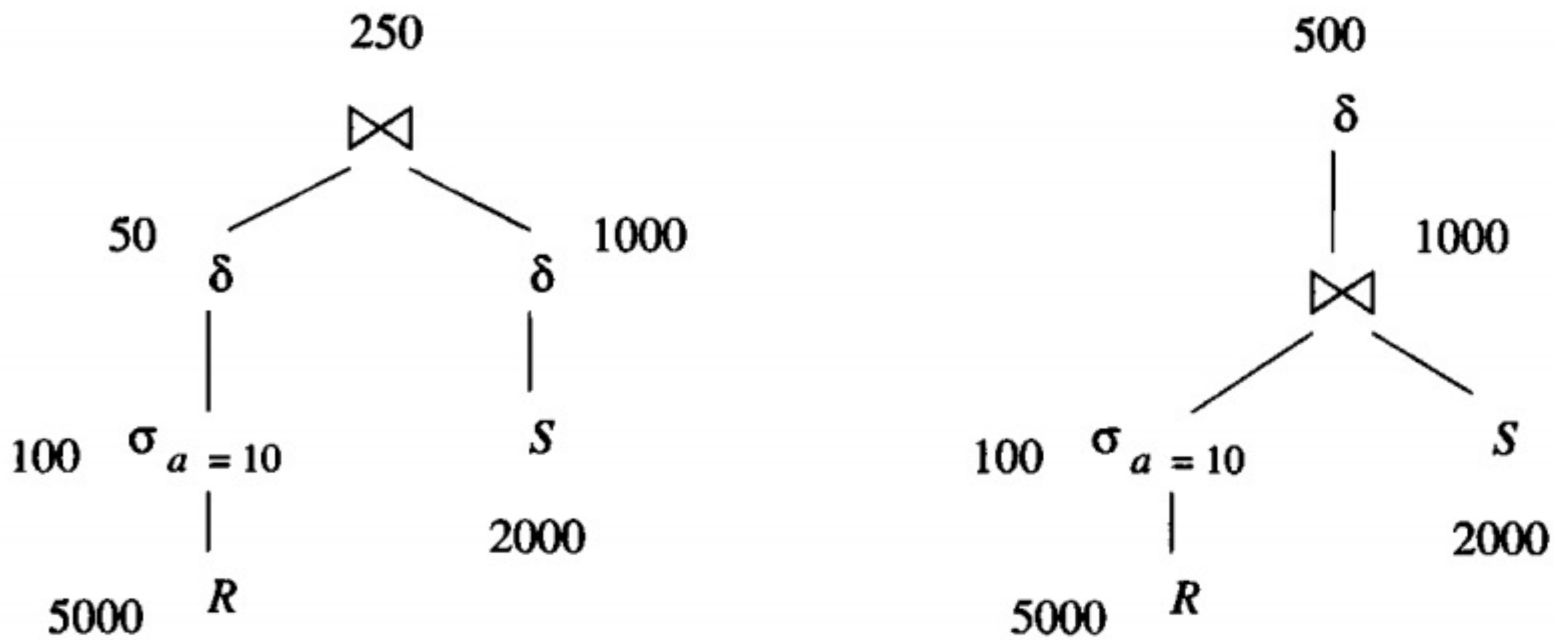
- Some query optimizers integrate heuristics selection and the generation of alternative access plans
  - Frequently used approach
    - Heuristic rewriting of nested block structure and aggregation
    - Followed by cost-based join-order optimization for each block
  - <u>Optimization cost budget</u> to stop optimization early (if cost of plan is less than cost of optimization)
  - <u>Plan caching</u> to reuse previous 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
  - Typically worth it for expensive queries
  - Optimizers often use simple heuristics for very cheap queries, and perform exhaustive enumeration for more expensive queries.

Example Query



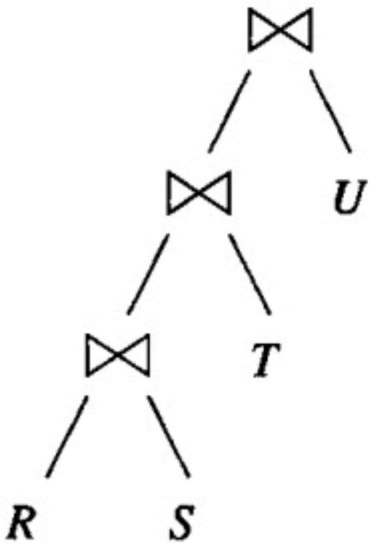
$R(a,b)$	$S(b,c)$
$T(R) = 5000$	$T(S) = 2000$
$V(R,a) = 50$	
$V(R,b) = 100$	$V(S,b) = 200$
	$V(S,c) = 100$

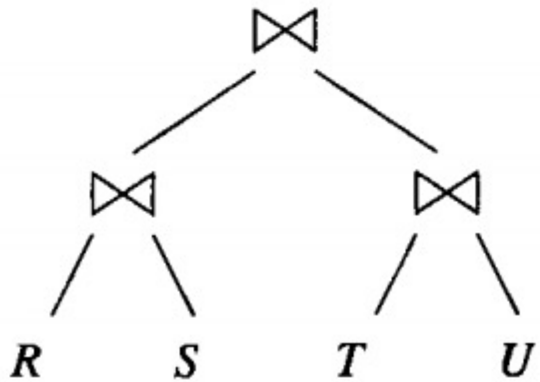
Two Query Plans

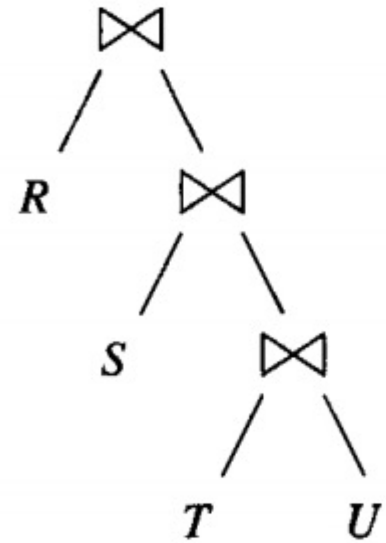


(a) (b)

Possible Joins for R,S,T,U







Stats and Singleton Sets

$R(a,b)$	$S(b,c)$	$T(c,d)$	$U(d,a)$
$V(R,a)=100$			$V(U,a)=50$
$V(R,b)=200$	$V(S,b)=100$		
	$V(S,c)=500$	$V(T,c)=20$	
		$V(T,d)=50$	$V(U,d)=1000$

	$\{R\}$	$\{S\}$	$\{T\}$	$\{U\}$
Size	1000	1000	1000	1000
Cost	0	0	0	0
Best plan	R	S	T	U

Pairs and Triples of Relations

	$\{R,S\}$	$\{R,T\}$	$\{R,U\}$	$\{S,T\}$	$\{S,U\}$	$\{T,U\}$
Size	5000	1,000,000	10,000	2000	1,000,000	1000
Cost	0	0	0	0	0	0
Best plan	$R \bowtie S$	$R \bowtie T$	$R \bowtie U$	$S \bowtie T$	$S \bowtie U$	$T \bowtie U$

	$\{R,S,T\}$	$\{R,S,U\}$	$\{R,T,U\}$	$\{S,T,U\}$
Size	10,000	50,000	10,000	2,000
Cost	2,000	5,000	1,000	1,000
Best plan	$(S \bowtie T) \bowtie R$	$(R \bowtie S) \bowtie U$	$(T \bowtie U) \bowtie R$	$(T \bowtie U) \bowtie S$

Grouping	Cost
$((S \bowtie T) \bowtie R) \bowtie U$	12,000
$((R \bowtie S) \bowtie U) \bowtie T$	55,000
$((T \bowtie U) \bowtie R) \bowtie S$	11,000
$((T \bowtie U) \bowtie S) \bowtie R$	3,000
$(T \bowtie U) \bowtie (R \bowtie S)$	6,000
$(R \bowtie T) \bowtie (S \bowtie U)$	2,000,000
$(S \bowtie T) \bowtie (R \bowtie U)$	12,000

Greedy Join & Join Selectivity

- Choose the smallest join
- $T \bowtie U \Rightarrow (T \bowtie U) \bowtie S \Rightarrow ((T \bowtie U) \bowtie S) \bowtie R$
- Selectivity = ratio between input and output
- Greedy approach picks the smallest selectivity

Physical Query Plan

- Choose algorithms to access data
  - Index scan/table scan
  - Join one pass/sort-join/INLJ/Hash join
- Materialized vs Pipelined
  - Buffer space

Long Indexes

select name, department

from employees

where age **in** (64, 65) **and** salary < 75000
and gender='female' **and** performance > 5;

- Build a composite index on everything
  - (gender, age, performance, salary)
- …what about "**or** performance > 5"
- Downsides to the composite index approach?

Covering Indexes

select name, department

from employees

where age **in** (64, 65) **and** salary < 75000
and gender='female' **and** performance > 5;

- Use index (gender, age, salary, performance)
  - Lookup the rows positions and sort
  - Table could be really, really large
- Consider the following index
  - (gender, age, salary, performance, name, department)

Covering Indexes

select name, department

from employees

where age **in** (64, 65) **and** salary < 75000
and gender='female' **and** performance > 5;

- Use the index
  - (gender, age, salary, performance, name, department)
  - Do not follow the pointers to the table!
  - Read the requested values from the keys

Composite Indexes

- Composite/covering indexes keys are large
- What is the problem?
- Index might be very large
  - Come back to that
- B+-Tree performance suffers
  - Long keys
  - E.g., 900 byte limit in SQL Server

Included Columns

- MS SQL Server
- Index:
  - (gender, age, salary, performance, name, department)
- Alternatively
  - **Create index** Name **on** Employees (gender, age, salary, performance) **include** (name, department)
  - Benefits of covering index w/out long key!

Prefix Suppression

- Oracle
- Consider our index with a long key
  - gender, age, salary, performance, name, department
  - (6 chars)+(3 digits)+(8 dig) + (2 dig) + (21 char)+(10 char)
  - Every key takes 50 bytes, and given 512-byte page
    - What's the fan-out of the B+-tree?
- What about neighboring keys in the index?
  - Values sorted in the leaves

Index Organized Tables (IOTs)

- Hide the data in the B-Tree!
  - Oracle and MySQL
- Merge the structures
  - Different from regular clustering
  - Share the structure properties

Few-Valued Columns

- Few-valued columns partition (sorted) data into “buckets”
  - E.g., gender column, performance, age
- Few-valued columns also create many opportunities for compression

Using Parts of a Composite Index

- Index: (gender, age, salary, performance)

select name, department
from employees
where age **in** (64, 65) **and** salary < 75000
and gender='male' **and** performance > 5;

select name, department
from employees
where age **in** (64, 65) **and** salary < 75000 **and** gender='male';

select name, department
from employees
where age **in** (64, 65) **and** gender='male';

Using Parts of a Composite Index

- Index: (gender, age, salary, performance)

select name, department
from employees
where age **in** (64, 65) **and** salary < 75000
and gender='male' **and** performance > 5;

select name, department
from employees
where salary < 75000 **and** gender='male' **and** performance > 5;

select name, department
from employees
where salary < 75000 **and** performance > 5;

Skip-Scan

- Oracle
  - Few-valued attributes as prefix
  - Searches every sub-B+-tree to compensate
- SQL Server
  - Adds predicates for few-valued attribute in prefix
  - E.g., index on (year, salary)
  - Convert “year > 2009” into “year in (2010,2011)”
    - Statistics!

Bitmap Indexes

- An index lets us map values to rowIDs
  - Age in (64,65)
  - Row pointers = (3,4,150,200,500,1000)
- What if we just stored the per-value pointers?
  - Age = 18 => (1, 333, 555, 1001)
  - …
  - Age = 64 => (3, 150, 500, 1000)
  - Age = 65 => (4, 200)

Bitmap Indexes

- Only works on few-valued columns
- Oracle and DB2
- Two ways of storing the same data
  - List of RowIDs
  - Dense 0,1 values

Views

- A view is a “virtual table”

  create view dept_tot_salary(dept_name, tot_salary) as
  select dept_name, **sum**(salary)

  from instructor

  group by dept_name

- Does not do much to optimize queries
  - Security, convenience, possibly query optimization
    - Optimizer may reconstruct
- Oracle will even preserve views when instructor table is dropped (how?)

Materialized Views

- Views are nearly cost-free
- Automatically updated (with table changes)
- … but not very useful
- Materialized Views
  - Compute and store the query
  - Use the view directly
  - Space cost
  - Maintenance cost

Dept_Name	SUM(Salary)
Comp. Sci.	10M
Music	2M
Economics	11M
History	1M
Comics Studies	500K
Numerology	750K

Using Materialized Views

- Materialized View (MV) is similar to a pre-computed query
  - Simpler language semantics
- Best case, $Q_a = MV_a$
  - Maintenance and disk space limitations
- Otherwise a lot of work
  - Pre-join
  - Pre-filtered
  - Pre-aggregated

Pre-joined Materialized View

- Consider $MV_a = r \bowtie s$
- Can be used to optimize query $r \bowtie s \bowtie t_1 \bowtie t_2$
  – If query plan costs are lower
    - Can't use indexes on s,r with MV_a
  – MV_a needs the right key for join
- MV_a has indexes of its own
  – In some DBs first (clustered) index materializes
  – All table-indexing considerations apply to MVs

Pre-filtered Materialized View

- Consider $MV_b = \sigma_{salary > 75,000} (r \bowtie s)$
  - Used to optimize query $\sigma_{salary > 75,000} (r \bowtie s)$
  - Or $\sigma_{salary > 100,000} (r \bowtie s)$
  - But not $\sigma_{salary > 71,000} (r \bowtie s)$
- Design a compromise
  - Cannot afford all pre-computation
  - Find the smallest suitable MV
    - E.g., $\sigma_{salary > 70,000} (r \bowtie s)$

Pre-aggregated Materialized View

- $MV_b = {}_{dept\_name,gender}G_{sum(salary)}(instructors)$
  - Optimize query ${}_{dept\_name,gender}G_{sum(salary)}(instr)$
  - Or ${}_{dept\_name}G_{sum(salary)}(instructors)$
  - But not ${}_{dept\_name,position}G_{sum(salary)}(instructors)$
- Design a compromise
  - Find the best shared aggregation
  - Data cubes

Materialized View Maintenance

- The task of keeping a materialized view up-to-date with the underlying data is know as <u>materialized view maintenance</u>
- Materialized views using <u>incremental view maintenance</u>
  - Changes do database relations are used to compute changes to the materialized view, which is then updated

Index Maintenance

- B+tree index
  - Amortized logarithmic cost of update
  - Roughly linear in # of indexes
  - Buffer conflicts
- Clustered B+trees index
  - Restructuring table is expensive
  - Update every secondary index
  - Some DBMSes do not offer support (IOTs)

Materialized View Maintenance

- Materialized Views
  - Simply propagate the row (or delete the row)
  - Update all affected views and indexes
- Pre-filtered MVs
  - Only update the row if it is relevant
  - E.g., MV = $\sigma_{salary > 75,000}$ (r)
  - Only update if new faculty has salary > 75K

Materialized View Maintenance

- Pre-joined MVs
  - Update all affected MVs
  - Execute all necessary joins (may be expensive)
  - Then, propagate the updates
- Pre-aggregated MVs
  - For **sum/avg** can just add new value
  - What about **delete**?
  - What about **max/min**?

Update Batching

- Individual row-inserts are expensive
  - Building a query plan per insert
  - Caching is unreliable
- Batch the inserts
  - B+-trees can be bulk-updated
  - Similarly, MVs can be bulk-updated
  - MyISAM and PostgreSQL batch internally
- Updates periodic in data warehouses