Query Optimization

CS 317/387

1

Query Evaluation

- ***Problem***: An SQL query is declarative –does not specify a query execution plan.
- A relational algebra expression is procedural
  - there is an associated query execution plan.
- ***Solution*:**Convert SQL query to an equivalent relational algebra and evaluate it using the associated query execution plan.
  - *But which equivalent expression is best?*

2

- Select Distinct *targetlist* from *R1,…,Rn* Where *condition*

Is equivalent to:

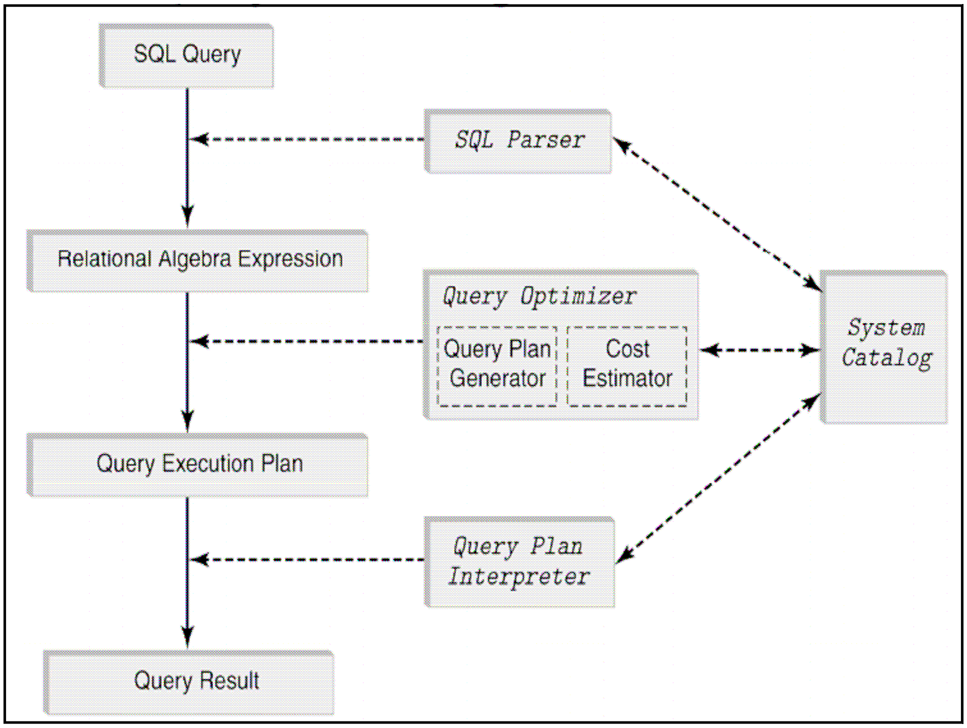
$$\pi_{TargetList}(\sigma_{Condition}(R1 \times R2 \times \ldots \times RN))$$

Example

$$\pi_{Name}(\sigma_{Id=ProfId \wedge CrsCode='CS532'}(\text{Professor} \times \text{Teaching}))$$

- Result can be < 100 bytes
- But if each Relation is 50k then we end up computing an intermediate result Professor x Teaching that is over 1G

Problem: Find an *equivalent* relational algebra expression that can be evaluated "*efficiently*".



Query Optimizer

- Uses heuristic algorithms to evaluate relational algebra expressions. This involves:
  - estimating the cost of a relational algebra expression
  - transforming one relational algebra expression to an equivalent one
  - choosing access paths for evaluating the subexpressions
- Query optimizers do not "optimize"– just try to find "reasonably good"evaluation strategies

6

Equivalence Preserving Transformation

- To transform a relational expression into another equivalent expression we need transformation rules that preserve equivalence

- Each transformation rule
  - Is provably correct (ie, does preserve equivalence)
  - Has a heuristic associated with it

7

Selection and Projection Rules

- Break complex selection into simpler ones:
- $\sigma_{Cond1 \wedge Cond2}(R) \equiv \sigma_{Cond1}(\sigma_{Cond2}(R))$

- Break projection into stages:
- $\Pi_{attr}(R) \equiv \pi_{attr}(\pi_{attr'}(R))$, if $attr \subseteq attr'$

- Commute projection and selection:
- $\Pi_{attr}(\sigma_{Cond}(R)) \equiv \sigma_{Cond}(\pi_{attr}(R))$,
  - if $attr \supseteq$ all attributes in *Cond*

8

Commutativity, Associativity of Joins

- Join commutativity: $R \propto S \equiv S \propto R$
  - Used to reduce cost of nested loop evaluation strategies (smaller relation should be in outer loop)
- Join associativity: $R \propto (S \propto T) \equiv (R \propto S) \propto T$
  - used to reduce the size of intermediate relations in computation of multi-relational join
  - first compute the join that yields smaller intermediate result
- N-way join has $T(N) \times N!$ different evaluation plans–
  - $T(N)$ is the number of parenthesized expressions
  - $N!$ is the number of permutations
- Query optimizer cannot look at all plans Hence it does not necessarily produce optimal plan

9

Pushing Selections and Projections

- $\sigma_{Cond}(R \times S) \equiv R \propto_{Cond} S$
  - *Cond* relates attributes of both R and S
  - Reduces size of intermediate relation since rows can be discarded sooner
- $\sigma_{Cond}(R \times S) \equiv \sigma_{Cond}(R) \times S$
  - *Cond* involves only the attributes of R
  - Reduces size of intermediate relation since rows of R are discarded sooner
- $\pi_{attr}(R \times S) \equiv \pi_{attr}(\pi_{attr'}(R) \times S)$,
  - if $attributes(R) \supseteq attr' \supseteq attr$
  - reduces the size of an operand of product

10

Equivalence Example

$\sigma_{C1 \wedge C2 \wedge C3}(R \times S)$
$\equiv \sigma_{C1}(\sigma_{C2}(\sigma_{C3}(R \times S)))$
$\equiv \sigma_{C1}(\sigma_{C2}(R) \times \sigma_{C3}(S))$
$\equiv \sigma_{C2}(R) \propto_{C1} \sigma_{C3}(S)$

Assuming C1 involves attributes of R and S, C2 involves only R and C3 involves only S.

11

Query Tree

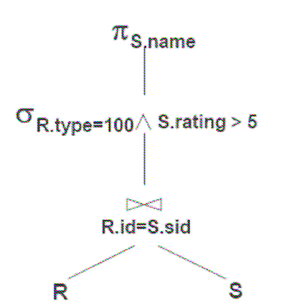
Tree structure that corresponds to a relational algebra expression:

–A leaf node represents an input relation;

–An internal node represents a relation obtained by applying one relational operator to its child nodes

–The root relation represents the answer to the query

–Two query trees are equivalent if their root relations are the same



SELECT DISTINCT S.sname
FROM R, S
WHERE R.id = S.sid
AND R. type = 100
AND S.rating > 5

<u>Left Deep Tree</u>: right child of a join is always a base relation

$\pi_{S.sname}(\sigma_{R.type = 100 \wedge S.rating > 5}(R \bowtie_{R.id=S.sid} S))$

Query Plan

Query Tree with specification of algorithms for each operation.

–A query tree may have different execution plans

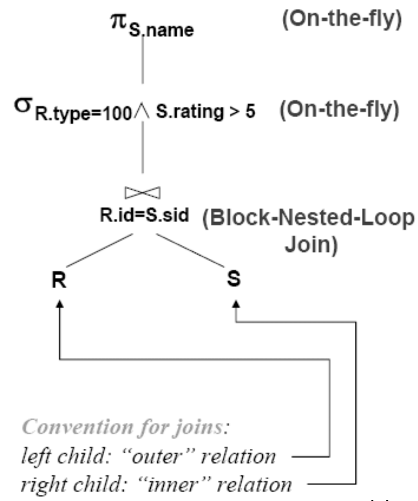
–Some plans are more efficient to execute than others.

•**<u>Two main issues:</u>**

–For a given query, what plans are considered?

–How is the cost of a plan estimated?

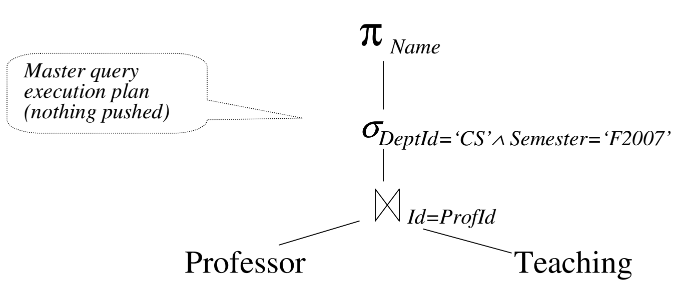
•Ideally: want to find best plan. Practically: avoid worst plans!



Cost - Example 1

SELECT P.*Name*
FROM Professor P, Teaching T
WHERE P.Id = T.*ProfId* -- *join condition*
AND P. *DeptId* = 'CS' AND T.*Semester* = 'F2007'

$$\pi_{Name}(\sigma_{DeptId='CS' \wedge Semester='F2007'}(\text{Professor} \bowtie_{Id=ProfId} \text{Teaching}))$$



14

Metadata on Tables (in system catalogue)

- **Professor (*Id*, *Name*, *DeptId*)**
  - *size*: 200 pages, 1000 rows, 50 departments
  - *indexes*: clustered, 2-level B^{+}tree on *DeptId*, hash on *Id*
- **Teaching (*ProfId*, *CrsCode*, *Semester*)**
  - *size*: 1000 pages, 10,000 rows, 4 semesters
  - *indexes*: clustered, 2-level B^{+}tree on Semester; hash on *ProfId*
- **Definition**: *Weight of an attribute – average number of rows that have a particular value*
  - *weight* of *Id* = 1 (it is a key)
  - *weight* of *Prof Id* = 10 (10,000 classes/1000 professors)

15

Estimating Cost - Example

- *Join* - block-nested loops with 51 page buffer
  - Scanning Professor (outer loop): 200 page transfers, (4 iterations, 50 transfers each)
  - Finding matching rows in Teaching (inner loop): 1000 page transfers <u>*for each iteration*</u> of outer loop
    - 250 professors in each 50 page chunk * 10 matching Teaching tuples per professor = 2500 tuple fetches = 2500 page transfers for Teaching (Why?)
    - By sorting the record Ids of these tuples we can get away with only 1000 page transfers (Why?)
  - total cost = 200+4*1000 = 4200 page transfers

16

Estimating Cost - Example

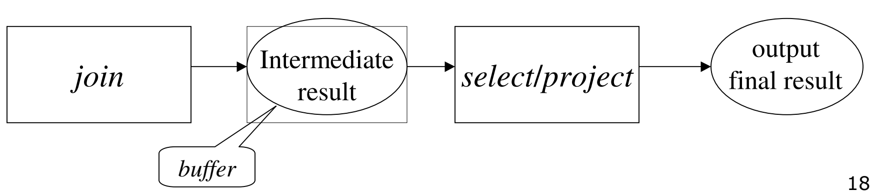
- *Selection* and *projection* – scan rows of intermediate file, discard those that don't satisfy selection, project on those that do, write result when output buffer is full.

- Complete algorithm:
  - do *join*, write result to intermediate file on disk
  - read intermediate file, do *select*/*project*, write final result
  - **Problem**: unnecessary I/O

17

Pipelining

- **Solution**: use *pipelining:*
  - *join* and *select*/*project* act as co-routines, operate as producer/consumer sharing a buffer in main memory.
    - When join fills buffer; *select*/*project* filters it and outputs result
    - Process is repeated until *select*/*project* has processed last output from join
  - Performing select/project adds no additional cost



18

Estimating Cost - Example 1

- Total cost:

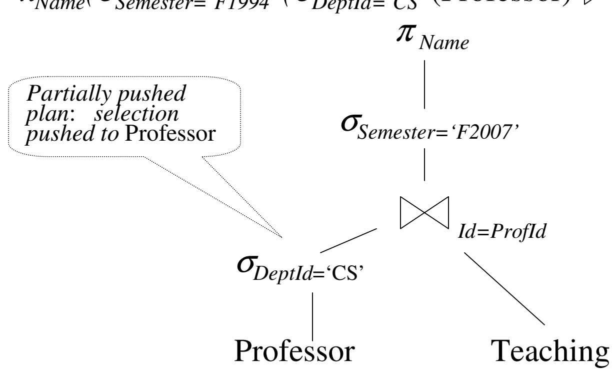
  4200 + (cost of outputting final result)

  - We will *disregard the cost of outputting final result* in comparing with other query evaluation strategies, since this will be same for all

Cost Example 2

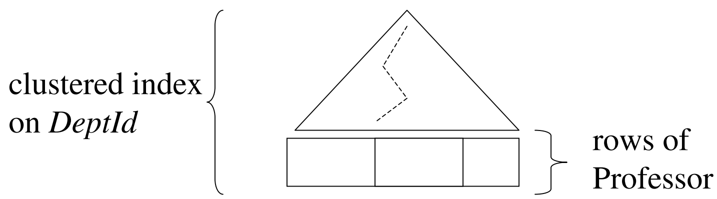
SELECT P.*Name*
FROM Professor P, Teaching T
WHERE P.Id = T.*ProfId* AND
P. *DeptId* = 'CS' AND T.*Semester* = 'F2007'

$$\pi_{Name}(\sigma_{Semester='F1994'}(\sigma_{DeptId='CS'}(\text{Professor}) \bowtie_{Id=ProfId} \text{Teaching}))$$



Cost Example 2 -- *selection*

- Compute $\sigma_{DeptId=\text{'CS'}}$ (Professor) (to reduce size of one join table) using clustered, 2-level B^+ tree on *DeptId*.
  - 50 departments and 1000 professors; hence *weight* of DeptId is 20 (roughly 20 CS professors). These rows are in ~4 consecutive pages in Professor.
    - Cost = 4 (to get rows) + 2 (to search index) = 6
    - keep resulting 4 pages in memory and pipe to next step



21

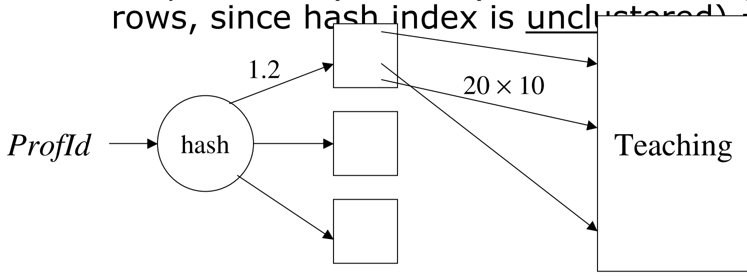
Cost Example 2 -- *join*

- Index-nested loops join using hash index on *ProfId* of Teaching and looping on the selected professors (computed on previous slide)
  - Since selection on *Semester* was not pushed, hash index on *ProfId* of Teaching can be used
  - *Note*: if selection on *Semester* were pushed, the index on *ProfId* would have been lost – an advantage of ***not*** using a fully pushed query execution plan

22

Cost Example 2 – *join* (cont'd)

- Each professor matches ~10 Teaching rows. Since 20 CS professors, hence 200 teaching records.
- All index entries for a particular *ProfId* are in same bucket. Assume ~1.2 I/Os to get a bucket.
  - Cost = 1.2 $\times$ 20 (to fetch index entries for 20 CS professors) + 200 (to fetch Teaching rows, since hash index is <u>unclustered</u>) = 224



23

Cost Example 2 – *select/project*

- Pipe result of join to *select* (on *Semester*) and *project* (on *Name*) at no I/O cost
- Cost of output same as for Example 1
- Total cost:

  6 (select on Professor) + 224 (join) = 230
- Comparison:

  4200 (example 1) vs. 230 (example 2) !!!

24

Estimating Output Size

- It is important to estimate the size of the output of a relational expression – size serves as input to the next stage and affects the choice of how the next stage will be evaluated.
- Size estimation uses the following measures on a particular instance of R:
  - *Tuples*(R): number of tuples
  - *Blocks*(R): number of blocks
  - *Values*(R.*A*): number of distinct values of *A*
  - *MaxVal*(R.*A*): maximum value of *A*
  - *MinVal*(R.*A*): minimum value of *A*

25

Estimating Output Size

- For the query:

  SELECT *TargetList*
  FROM $R_1, R_2, \ldots, R_n$
  WHERE *Condition*

  - *Reduction factor* is

$$\frac{\textit{Blocks}\text{ (result set)}}{\textit{Blocks}(R_1) \times \ldots \times \textit{Blocks}(R_n)}$$

    - Estimates by how much query result is smaller than input

26

Estimation of Reduction Factor

- Assume that reduction factors due to target list and query condition are independent
- Thus:
  reduction(Query) =
  reduction(TargetList) $\times$ r*eduction(Condition)*

27

Reduction Due to *Condition*

- *reduction* ($R_i.A=val$) = $\frac{1}{Values(R.A)}$
- *reduction* ($R_i.A=R_j.B$) = $\frac{1}{max(Values(R_i.A), Values(R_j.B))}$
- *reduction* ($R_i.A > val$) = $\frac{MaxVal(R_i.A) - val}{Values(R_i.A)}$

28

Reduction Due to *TargetList*

- *reduction(TargetList)* =

$$\frac{\textit{number-of-attributes}\ (\textit{TargetList})}{\textit{number-of-attributes}\ (R_i)}$$

29

Estimating Weight of Attribute

weight(R.*A*) =

Tuples(R) $\times$ *reduction*(R.*A*=*value*)

30

Choosing Query Execution Plan

- Step 1: Choose a *logical* plan
- Step 2: Reduce search space
- Step 3: Use a heuristic search to further reduce complexity

31

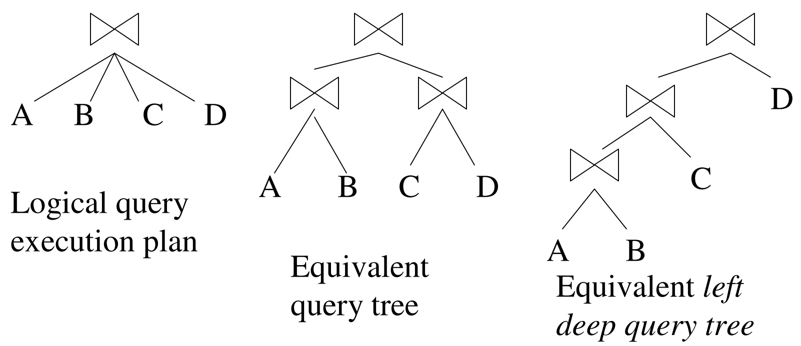
Step 1: Choosing a Logical Plan

- Involves choosing a query tree, which indicates the order in which algebraic operations are applied
- *Heuristic*: Pushed trees are good, but sometimes "nearly fully pushed" trees are better due to indexing (as we saw in the example)
- **So**: Take the initial "masterplan" tree and produce a *fully pushed* tree plus several *nearly fully pushed* trees.

32

Step 2: Reduce Search Space

- Deal with *associativity* of binary operators (join, union, …)



33

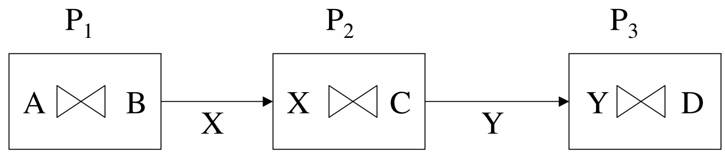
Step 2 (cont'd)

- Two issues:
  - Choose a particular *shape* of a tree (like in the previous slide)
    - Equals the number of ways to parenthesize N-way join – grows very rapidly
  - Choose a particular permutation of the leaves
    - E.g., 4! permutations of the leaves A, B, C, D

34

Step 2: Dealing With Associativity

- Too many trees to evaluate: settle on a particular shape: *left-deep tree.*
  - Used because it allows *pipelining*:

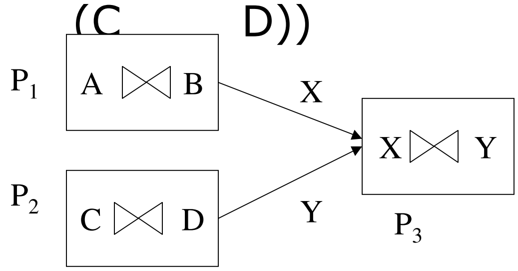


  - *Property*: once a row of X has been output by P_1, it need not be output again (but C may have to be processed several times in P_2 for successive portions of X)
  - *Advantage*: none of the intermediate relations (X, Y) have to be completely materialized and saved on disk.
    - Important if one such relation is very large, but the final result is small

35

Step 2: Dealing with Associativity

- consider the alternative: if we use the association $((A \bowtie B) \bowtie (C \bowtie D))$



Each row of X must be processed against all of Y. Hence all of Y (can be very large) must be stored in P_3, or P_2 has to recompute it several times.

36

Step 3: Heuristic Search

- The choice of left-deep trees still leaves open too many options (N! permutations):
  - $(((A \bowtie B) \bowtie C) \bowtie D)$,
  - $(((C \bowtie A) \bowtie D) \bowtie B)$, …..
- A heuristic (often dynamic programming based) algorithm is used to get a ‘good’ plan

37

Step 3: Dynamic Programming Algorithm

- Just an idea – see book
- To compute a join of E_1, E_2, …, E_N in a left-deep manner:
  - Start with 1-relation expressions (can involve σ, π)
  - Choose the best and “nearly best” plans for each (a plan is considered nearly best if its out put has some “interesting” form, e.g., is sorted)
  - Combine these 1-relation plans into 2-relation expressions. Retain only the best and nearly best 2-relation plans
  - Do same for 3-relation expressions, etc.

38

Index-Only Queries

- A B^+ tree index with search key attributes A_1, A_2, …, A_n has stored in it the values of these attributes for each row in the table.
  - Queries involving a prefix of the attribute list A_1, A_2, .., A_n can be satisfied using *only the index* – no access to the actual table is required.
- **Example**: Transcript has a clustered B^+ tree index on *StudId*. A frequently asked query is one that requests all grades for a given *CrsCode*.
  - ***Problem***: Already have a clustered index on StudId – cannot create another one (on *CrsCode)*
  - ***Solution***: Create an unclustered index on (*CrsCode*, *Grade*)

39