An Overview of Query Optimization

Chapter 14

Query Evaluation

- **Problem**: An SQL query is declarative - it 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?*

Trivial Conversion

SELECT DISTINCT *TargetList*
FROM R1, R2, … RN
WHERE *Condition*

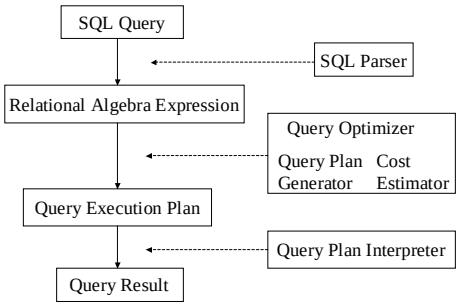
is equivalent to

$\Pi_{TargetList}(\sigma_{Condition}(R1 \times R2 \times ... \times RN))$

but this may imply a very inefficient query execution plan.

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

Query Processing Architecture



Query Optimizer

- Uses a heuristic algorithm that involves
  - estimating the cost of a relational algebra expression
  - transforming one relational algebra expression to an equivalent one.

Equivalency Preserving Transformations

- $\sigma_{Cond1 \wedge Cond2}(R) \equiv \sigma_{Cond1}(\sigma_{Cond2}(R))$
- $\Pi_{attr}(R) \equiv \Pi_{attr}(\Pi_{attr'}(R))$ if $attr \subseteq attr'$
- $\Pi_{attr}(\sigma_{Cond}(R)) \equiv \sigma_{Cond}(\Pi_{attr}(R))$ if *attr* includes all attributes in *Cond*

Commutativity and Associativity of Join (and Cartesian Product)

- $R \bowtie S \equiv S \bowtie R$
  - used to reduce cost of nested loop evaluation strategies (smaller relation should be in outer loop
- $R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T$
  - used to reduce the size of intermediate relations in computation of multi-relational join
- N-way join has *T(N)× N!* different evaluation plans
  - *T(N)* is the number of parenthesized expressions
  - *N!* is the number of permutations
- Query optimizer can't look at all plans => doesn't necessarily produce optimal plan

7

Pushing

- $\sigma_{Cond}(R \times S) \equiv R \bowtie_{Cond} S$
  - *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

8

Equivalence Example

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

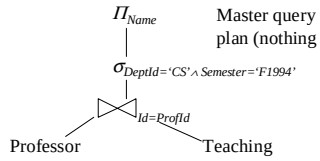
assuming *C2* involves only attributes of R,
C3 involves only attributes of S,
and *C1* relates attributes of R and S

9

Cost - Example 1

```
SELECT P.Name
FROM Professor P, Teaching T
WHERE P.Id = T.ProfId AND
      P. DeptId = 'CS' AND T.Semester = 'F1994'
```

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



Master query execution plan (nothing pushed)

10

Metadata on Tables

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

11

Estimating Cost - Example 1

- Join - block-nested loops with ~50 page buffer
  - Scanning Professor (outer loop): 200 page transfers, (4 iterations, 50 transfers each)
  - Finding matching rows in Teaching (inner loop): 1000 page transfers for each iteration of outer loop
  - total cost = 4200 page transfers

12

Estimating Cost - Example 1

- Selection and projection - scan rows as they are formed, 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 out final result
  - **Problem**: unnecessary I/O

13

Pipelining

- **Solution**: use *pipelining:*
  - join and select/project act as coroutines, operating as producer/consuimer 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
  - Peforming select/project adds no additional cost for I/O to intermediate file

14

Estimating Cost - Example 1

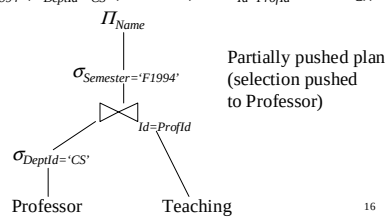
- Total cost =4200 + (cost of outputting final result).
  - Ignore output cost in comparing with other query evaluation strategies since this will be same for all

15

Cost Example 2

SELECT P.Name
FROM Professor P, Teaching T
WHERE P.Id = T.ProfId AND
P. DeptId = 'CS' AND T.Semester = 'F1994'

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



Partially pushed plan (selection pushed to Professor)

16

Cost Example 2

- Compute $\sigma_{DeptId='CS'}$(Professor) (to reduce the size of one element of the join) 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

17

Cost Example 2

- Index-nested loops join using hash index on ProfId of Teaching and looping on the selected professors-
  - Since selection on Semester was not pushed, hash index on Teaching can be used (advantage of not using a fully pushed query execution plan)
  - Each professor in selection result matches ~10 Teaching rows. Index entries for these rows are in same bucket. Assume ~1.2 I/Os to search a bucket.
    - Cost = 1.2 ×20 (to fetch index entries from bucket for 20 professors) + 200 (to fetch Teaching rows since index is unclustered) = 224

18

Cost Example 2

- 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 + 224 = 230
- Comparison: 4200 vs 230 !!!

19

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
  - Values(R.A): number of distinct values of A
  - MaxVal(R.A): maximum value of A
  - MinVal(R.A): minimum value of A

20

Estimating Output Size

- For the query:

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

  - reduction factor is $\frac{Tuples \text{ (result set)}}{Tuples(R_1) \times \ldots \times Tuples(R_n)}$
  - assume:

    reduction(Query) =
    reduction(TargetList) × *reduction(Condition)*

21

Reduction From *Condition*

- *reduction* $(R_i.A = val) = \frac{1}{Values(R.A)}$
- *reduction* $(R_i.A = R_j.B) = \frac{1}{max(Values(R_i.A), Values(R_j.B))}$
  - since we have assumed that values are uniformly distributed, $Tuples(R_i) < Tuples(R_j)$, and every row of R_i matches a row of R_j , the number of tuples that satisfy =

    $Values(R_i.A) \times (Tuples(R_i.A)/Values(R_i.A)) \times (Tuples(R_j.A)/Values(R_j.A))$
- *reduction* $(R_i.A > val) = \frac{MaxVal(R_i.A) - val}{Values(R_i.A)}$

22

Reduction From *TargetList*

- *reduction(TargetList) =*

$$\frac{\text{Number-of-attributes (}TargetList\text{)}}{\text{Number-of-attributes } (R_i)}$$

23

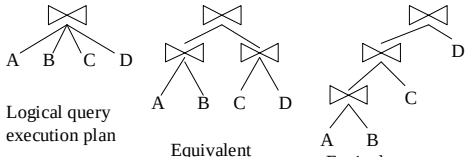
Choosing a Plan

- **Step 1**: Choose *logical* query execution plans: a fully pushed tree and several nearly fully pushed trees.

24

Choosing a Plan

- **Step 2**: Deal with associativity of binary operators (join, union, …)



Logical query execution plan

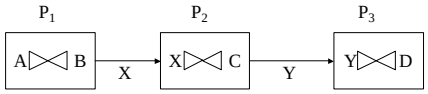
Equivalent query tree

Equivalent query tree: *left deep query tree*

25

Dealing With Associativity

- Too many trees to evaluate.
- Choose left-deep tree since it allows pipelining:

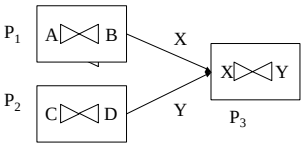


  - property: once a row of X has been output by P_1, it need not be output again (but C may have to be processed several times in P_2 for successive portions of X)
  - Advantage: none of the intermediate relations (X, Y) have to be completely materialized. This can be important if one such relation is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

26

Dealing With Associativity

- consider the alternative: if we use the association ((A ⋈ B) ⋈ (C ⋈ D))



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

27

Dealing|With Commutativity

- The choice of left-deep trees still leaves open many options:
  - (((A ⋈ B) ⋈ C) ⋈ D),
  - (((C ⋈ A) ⋈ D) ⋈ B), …..
- A heuristic (often dynamic programming based) algorithm is used to get a 'good' plan.

28

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 these attributes can be satisfied using only the index - no reference to the 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**: Can't create a second clustered index on CrsCode
  - **Solution**: Create an unclustered index on (CrsCode, Grade)
    - Keep in mind, however, the overhead in maintaining extra indices

29