Lecture 20 (Chapter 11)
An Overview of Query Optimization

Lecturer: Sael Lee

Slide adapted from the author’s, Peter Bailis’s and Dr. Ilchul Yoon’s slides.
Query Evaluation

- **Problem**
  - An SQL query is declarative – does not specify a query execution plan.
  - A relational algebra expression is procedural and 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?*
Naive Conversion

```
SELECT DISTINCT TargetList
FROM R1, R2, ..., RN
WHERE Condition
```

is equivalent to: \( \pi_{\text{TargetList}}(\sigma_{\text{Condition}}(R_1 \times R_2 \times ... \times R_N)) \)

but this may imply a very inefficient query execution plan.

**Example:** \( \pi_{\text{Name}}(\sigma_{\text{Id=ProfId}} \land \text{CrsCode='CS532'})(\text{Professor} \times \text{Teaching}) \)

- Result can be < 100 bytes
- But if each relation is 50K then we end up computing an intermediate result \( \text{Professor} \times \text{Teaching} \) of size 500M before shrinking it down to just a few bytes.

**Problem statement:**

Find an *equivalent* relational algebra expression that can be evaluated “efficiently”. 

Query Processing Architecture

- SQL Query
  - SQL Parser
  - Relational Algebra Expression
    - Logical query plan
  - Query Optimizer
    - Query Plan Generator
    - Cost Estimator
  - System Catalog
  - Query Plan Interpreter
    - Physical query plan
  - Query Execution Plan
  - Query Result
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 sub-expressions

- Query optimizers do not “optimize” – just try to find “reasonably good” evaluation strategies
Example: SQL query

SELECT title
FROM StarsIn
WHERE starName IN ( 
    SELECT name 
    FROM MovieStar 
    WHERE birthdate LIKE '%1960'
);

(Find the movies with stars born in 1960)
Example: 1 Parse Tree

```
SELECT <SelList>     FROM    <FromList>     WHERE     <Condition>
   <Attribute>              <RelName>                 <Tuple>  IN  <Query>
      title                       StarsIn               <Attribute>      (  <Query>  )
               starName       <SFW>

SELECT      <SelList>    FROM     <FromList>     WHERE     <Condition>
   <Attribute>           <RelName                  <Attribute>  LIKE  <Pattern>
      name                 MovieStar              birthDate            '%1960'
```
Example: 2 Generating Relational Algebra

\[ \Pi_{\text{title}} \ \sigma_{\text{birthdate LIKE '1960'}} \ \text{IN} \ \Pi_{\text{name}} \ \text{StarSlave} \ \text{IN} \ \Pi_{\text{name}} \ \text{MovieStar} \ \text{starName} \]

Fig. An expression using a two-argument \( \sigma \), midway between a parse tree and relational algebra
Example: 3 Logical Query Plan

\[ \Pi_{\text{title}} \sigma_{\text{starName}=\text{name}} \times \Pi_{\text{name}} \sigma_{\text{birthdate LIKE } \%1960\%} \]

Fig. Applying the rule for IN conditions
**Example: 4 Improve Logical Query Plan**

\[ \Pi_{\text{title}} \]

\[ \text{starName}=\text{name} \]

\[ \text{StarsIn} \]

\[ \Pi_{\text{name}} \]

\[ \sigma \text{birthdate LIKE ‘% 1960’} \]

MovieStar

**Fig. 7.20: An improvement on fig. 7.18.**

**Question:**

Push project to StarsIn n?
Example: Estimate Result Sizes

Need expected size

StarsIn

\[ \Pi \sigma \]
Example: One Physical Plan

Hash join

SEQ scan
StarsIn

index scan
MovieStar

Parameters: join order, memory size, project attributes,...

Parameters: Select Condition,...
Example: Estimate costs

L.Q.P

P1 | P2 | .... | Pn

C1 | C2 | .... | Cn

Pick best!
Equivalence Preserving Transformations

- To transform a relational expression into another equivalent expression, we need transformation rules that preserve equivalence.
- Each transformation rule:
  - Is *provably* correct (i.e., does preserve equivalence).
  - Has a heuristic associated with it.
Commutativity and Associativity of Join
(and Cartesian Product as Special Case)

- Join commutativity: \( R \bowtie S \equiv S \bowtie R \)
  - used to reduce cost of nested loop evaluation strategies (smaller relation should be in outer loop)

- Join associativity: \( R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T \)
  - used to reduce the size of intermediate relations in computation of multi-relational join – 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 (might take longer to find an optimal plan than to compute query brute-force). Hence it does not necessarily produce optimal plan
Commutativity and Associativity of Join

Natural joins & cross products & union

\[ R \bowtie S = S \bowtie R \]

\[ (R \bowtie S) \bowtie T = R \bowtie (S \bowtie T) \]

Commutative Law

Associative Law

\[ R \times S = S \times R \]

\[ (R \times S) \times T = R \times (S \times T) \]

\[ R \cup S = S \cup R \]

\[ R \cup (S \cup T) = (R \cup S) \cup T \]
Selection and Projection Rules

- Break complex selection into simpler ones:
  \[ \sigma_{Cond_1 \land Cond_2} (R) \equiv \sigma_{Cond_1} (\sigma_{Cond_2} (R)) \]

- Break projection into stages:
  \[ \pi_{attr} (R) \equiv \pi_{attr} (\pi_{attr'} (R)), \text{ if } attr \subseteq attr' \]

- Commute projection and selection:
  \[ \pi_{attr} (\sigma_{Cond} (R)) \equiv \sigma_{Cond} (\pi_{attr} (R)), \text{ if } attr \supseteq \text{ all attributes in } Cond \]
Laws Involving Selects

Splitting Laws:

\[ \sigma_{p_1 \land p_2}(R) = \sigma_{p_1} \ [ \sigma_{p_2}(R)] \]

\[ \sigma_{p_1 \lor p_2}(R) = [ \sigma_{p_1}(R)] \cup [ \sigma_{p_2}(R)] \]

Since **selections** tend to reduce the size of relations markedly, we want to move the selections down the tree as far as they will go.
Bags vs. Sets

R = \{a,a,b,b,b,c\}
S = \{b,b,c,c,d\}
R \cup S = ?

- **Option 1**  SUM
  
  RUS = \{a,a,b,b,b,b,b,c,c,c,d\}

- **Option 2**  MAX
  
  RUS = \{a,a,b,b,b,c,c,d\}
Laws Involving Project

Let: \( X = \text{set of attributes} \)
\( Y = \text{set of attributes} \)
\( XY = X \cup Y \)

\[ \pi_{xy}(R) = \pi_x[\pi_y(R)] \]

While selections reduce the size of a relation by a large factor, projection keeps the number of tuples the same and only reduce the length of tuples and sometimes increase the length of tuples.
Pushing Selections and Projections

- $\sigma_{Cond}(R \times S) \equiv R \bowtie_{Cond} S$
  - $Cond$ 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 $\text{attributes}(R) \supseteq attr' \supseteq attr \cap \text{attributes}(R)$
  - reduces the size of an operand of product
CSE 305 / CSE532

Lecture 21 (Chapter 11)
An Overview of Query Optimization

Lecturer: Sael Lee

Slide adapted from the author’s and Dr. Ilchul Yoon’s slides.
Let \( p \) = predicate with only R attributes

\( q \) = predicate with only S attributes

\( m \) = predicate with only R, S attributes

**Rules:**

\[
\sigma_p (R \bowtie S) = [\sigma_p (R)] \bowtie S
\]

\[
\sigma_q (R \bowtie S) = R \bowtie [\sigma_q (S)]
\]
Rules: $\sigma + \bowtie$ combined

$\sigma_{p \land q} (R \bowtie S) = [\sigma_p (R)] \bowtie [\sigma_q (S)]$

$\sigma_{p \land q \land m} (R \bowtie S) =
\sigma_m \left[ (\sigma_p R) \bowtie (\sigma_q S) \right]$

$\sigma_{pvq} (R \bowtie S) =
\left[ (\sigma_p R) \bowtie S \right] \cup \left[ R \bowtie (\sigma_q S) \right]$
Rules: $\pi, \sigma$ combined

Let $x$ = subset of $R$ attributes
$z$ = attributes in predicate $P$
(subset of $R$ attributes)

$$\pi_x[\sigma_p (R)] = \{\sigma_p \left[ \pi_x (R) \right]\}$$
**Rules: \( \pi, \sigma \) combined**

Let \( x \) = subset of \( R \) attributes

\( z \) = attributes in predicate \( P \)

(subset of \( R \) attributes)

\[
\pi_x[\sigma_p(R)] = \pi_x\left\{\sigma_p[\pi_x(R)]\right\}
\]
Rules: $\pi$, $\bowtie$ combined

Let $x =$ subset of $R$ attributes
$y =$ subset of $S$ attributes
$z =$ intersection of $R,S$ attributes

$$\pi_{xy}(R \bowtie S) =$$

$$\pi_{xy}\{[\pi_{xz}(R)] \bowtie [\pi_{yz}(S)]\}$$
\[ \pi_{xy} \{ \sigma_p (R \bowtie S) \} = \]

\[ \pi_{xy} \{ \sigma_p [\pi_{xz'} (R) \bowtie \pi_{yz'} (S)] \} \]

\[ z' = z \cup \{ \text{attributes used in } P \} \]
Equivalence Example

- \( \sigma_{C1 \land C2 \land C3} (R \times S) \)
  \[ \equiv \sigma_{C1} (\sigma_{C2} (\sigma_{C3} (R \times S))) \]
  \[ \equiv \sigma_{C1} (\sigma_{C2} (R) \times \sigma_{C3} (S)) \]
  \[ \equiv \sigma_{C2} (R) \bigotimes_{C1} \sigma_{C3} (S) \]

- assuming that
  - \( C2 \) involves only attributes of \( R \),
  - \( C3 \) involves only attributes of \( S \), and
  - \( C1 \) relates attributes of \( R \) and \( S \)
**Rules** $\sigma, U$ combined:

\[ \sigma_p(R \cup S) = \sigma_p(R) \cup \sigma_p(S) \]

\[ \sigma_p(R - S) = \sigma_p(R) - S = \sigma_p(R) - \sigma_p(S) \]
Which are “good” transformations?

- $\sigma_{p_1 \land p_2} (R) \rightarrow \sigma_{\sigma(p_2(R))}(R)$

- $\sigma_p (R \bowtie S) \rightarrow [\sigma_p (R)] \bowtie S$

- $R \bowtie S \rightarrow S \bowtie R$

- $\pi_x [\sigma_p (R)] \rightarrow \pi_x \{\sigma_p [\pi_{xz}(R)]\}$
Conventional wisdom:  

do projects early

Example:  \( R(A,B,C,D,E) \)  \( x=\{E\} \)

\( P: (A=3) \land (B=“cat”) \)

\( \pi_x \{ \sigma_P (R) \} \)  \( \text{vs.} \)  \( \pi_E \{ \sigma_P \{ \pi_{ABE}(R) \} \} \)
What if we have A, B indexes?

But

B = “cat”

Intersect pointers to get
pointers to matching tuples

A = 3
Bottom line:

- No transformation is always good
- Usually good: early selections
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 = 'F1994'

π Name(σ DeptId='CS' ∧ Semester='F1994' (Professor ⋈ Id=ProfId Teaching))

Master query execution plan (nothing pushed)
Metadata on Tables (in system catalogue)

- **Professor** (Id, Name, DeptId)
  - size: 200 pages, 1000 rows, 50 departments (5 tuples/page)
  - indices: clustered 2-level B⁺ tree on DeptId, hash on Id

- **Teaching** (ProflId, CrsCode, Semester)
  - size: 1000 pages, 10,000 rows, 4 semesters, (10 tuples/page)
  - indices: clustered 2-level B⁺ tree on Semester; hash on ProflId

- **Definition**: Weight of an attribute – *average number of rows that have a particular value*
  - weight of Id = 1 (it is a key)
  - weight of ProflId = 10 (10,000 classes/1000 professors)
Estimating Cost - Example 1

- **Assumption**
  - 52 page buffer is available for evaluating join
  - Small amount of additional memory is available for aux. info.

- **Join** - index-nested loops with 50 page buffers
  - 50 pages – input for Professor,
  - 5 profs per page and average 10 classes per each prof
  - Cost to scan **Professor** relation
    - 200 page transfers
  - Cost to find matching tuples in **Teaching**
Cost to find matching tuples in Teaching

Max. 2500 tuples (50 pages x 5 faculty/page x avg 10 classes/faculty) in Teaching could be matched. (i.e., max. page transfers could be 2500.) for loaded Professor pages

However, by sorting record ids of the Teaching pointed by the 2500 tuples, this can be done in 1000 page transfers = size(Teaching)

Repeating 4 times (200 pages/50 buffer) makes 4000 page transfers from Teaching
Estimating Cost - Example 1 (cont’d)

- 50 pages – input for Professor,
- 5 profs per page and average 10 classes per each prof

Cost to search index of Teaching (p.Id=t.ProfId)

- ProfID is hash-indexed.
- 1.2 I/O per index search, assuming good hash function (1.2)
- If all matching tuples are stored in a single bucket (10 on average), indices for the 10 tuples can be retrieved in one I/O operation.
- There are 10000 tuples in Teaching. This requires 1000 I/Os makes 1200 page transfers

So... the total cost is 200 + 4000 + 1200 = 5400 page transfers
Estimating Cost - Example 1 (cont’d)

- **Join** - block-nested loops with 52 page buffers
  - 50 pages – input for Professor,
  - 1 page – input for Teaching,
  - 1 – output page

- 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 = 200 + 4*1000 = 4200 page transfers
Estimating Cost - Example 1 (cont’d)

- **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 (big) intermediate file, do **select/project**, write final result

- **Problem: unnecessary I/O**

\[ \pi_{Name} \]
\[ \sigma_{\text{DeptId}='CS' \land \text{Semester}='F1994'} \]

4200 page

Professor

Teaching

Id=ProflId
Pipelining

Solution: use pipelining:

- join and select/project act as co-routines, operate as producer/consumer sharing a buffer in main memory.
- Output of one relational operator is “piped” to the input of the next operator without saving the intermediate result on disk.
  - 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
Estimating Cost - Example 1 (cont’d)

- I/O operations required for storing data will be reduced

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

\[ \pi_{\text{Name}}(\sigma_{\text{Semester}=\text{F1994}}(\sigma_{\text{DeptId}=\text{CS}}(\text{Professor}) \bowtie_{\text{Id}=\text{ProfId}} \text{Teaching})) \]
```

**Partially pushed plan:**

*selection pushed to Professor*
Cost Example 2 -- selection

- Compute $\sigma_{\text{DeptId}=\text{'CS'}}$ (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

![Diagram of clustered index on DeptId and rows of Professor]
Cost Example 2 – *join* (cont’d)

- Each professor matches ~ roughly 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.
  - Index fetch cost: $1.2 \times 20$ (to fetch index entries for 20 CS professors)

Total Cost

- $24 + 200$ (to fetch Teaching rows, since hash index is unclustered) = 224
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:
  
  \[6 \text{ (select on Professor)} + 224 \text{ (join)} = 230\]
- Comparison:
  
  4200 (example 1) vs. 230 (example 2) !!!
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
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.
  - Avoid exponential complexity problem by grouping consecutive binary operators of the same kind into one node.

- **So:** Take the initial “master plan” tree and produce a **fully pushed** tree plus several **nearly fully pushed** trees.
Step 1: Choosing a Logical Plan (cont’d)
Step 2: Reduce Search Space

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

Logical query execution plan

Equivalent query tree

Equivalent *left deep query tree*