Database Applications (15-415)

DBMS Internals- Part X
Lecture 18, March 26, 2014

Mohammad Hammoud
Today...

- **Last Session:**
  - DBMS Internals- Part VIII
    - Query Optimization

- **Today’s Session:**
  - DBMS Internals- Part IX
    - Query Optimization (Cont’d)

- **Announcements:**
  - Project 3 is due on April 5th
  - Quiz 2 is on Thursday, April 3, at 5:00PM in Room 2051 (*all material covered after the midterm*)
Query Optimization Steps

- **Step 1**: Queries are parsed into internal forms (e.g., parse trees)

- **Step 2**: Internal forms are transformed into ‘canonical forms’ (syntactic query optimization)

- **Step 3**: A subset of alternative plans are enumerated

- **Step 4**: Costs for alternative plans are estimated

- **Step 5**: The query evaluation plan with the least estimated cost is picked
Required Information to Estimate Plan Costs

- For each enumerated plan, we have to estimate its cost.

- To estimate the cost of a query plan, the query optimizer examines the *system catalog* and retrieves:
  - Information about the types and lengths of fields
  - Statistics about the referenced relations
  - Access paths (indexes) available for relations

- In particular, the *Schema* and *Statistics* components in the Catalog Manager are inspected to find a *good enough* query evaluation plan.
Cost-Based Query Sub-System: Revisit

Select * From Blah B Where B.blah = blah

Queries

Query Parser

Query Optimizer

Plan Generator

Plan Cost Estimator

Query Plan Evaluator

Catalog Manager

Schema

Statistics

Usually there is a heuristics-based rewriting step before the cost-based steps.
Catalog Manager: The Schema Component

- What kind of information do we store at the Schema?
  - Information about tables (e.g., table names and integrity constraints) and attributes (e.g., attribute names and types)
  - Information about indices (e.g., index structures)
  - Information about users

- Where do we store such information?
  - In tables; hence, can be queried like any other tables
  - For example: Attribute_Cat (attr_name: `string`, rel_name: `string`; type: `string`; position: `integer`)
Catalog Manager: The Statistics Component

- What would you store at the Statistics component?
  - NTuples(R): # records for table R
  - NPages(R): # pages for R
  - NKeys(I): # distinct key values for index I
  - INPages(I): # pages for index I
  - IHeight(I): # levels for I
  - ILow(I), IHigh(I): range of values for I
  - ...

- Such statistics are important for estimating operation costs and result sizes
Estimating the Cost of a Plan

- The cost of a plan can be estimated by:
  1. Estimating the *cost of each operation* in the plan tree
     - Already covered last week (e.g., costs of various join algorithms)
  2. Estimating the *size of the result of each operation* in the plan tree
     - The output *size* and *order* of a child node affects the cost of its parent node

How can we estimate result sizes?
Estimating Result Sizes

- Consider a query block, \( QB \), of the form:

\[
\text{SELECT attribute list} \\
\text{FROM R1, R2, ..., Rn} \\
\text{WHERE term 1 AND ... AND term k}
\]

- What is the \textit{maximum} number of tuples generated by \( QB \)?
  - \( \text{NTuples (R1)} \times \text{NTuples (R2)} \times \ldots \times \text{NTuples(Rn)} \)

- Every term in the WHERE clause, however, eliminates some of the possible resultant tuples
  - A \textit{reduction factor} can be associated with each term
Estimating Result Sizes (Cont’d)

- Consider a query block, \( QB \), of the form:

```sql
SELECT attribute list 
FROM R1, R2, ..., Rn 
WHERE term 1 AND ... AND term k
```

- The *reduction factor (RF)* associated with each *term* reflects the impact of the *term* in reducing the result size.

- Final *(estimated)* result cardinality = \([\text{NTuples}(R1) \times \ldots \times \text{NTuples}(Rn)] \times [RF(\text{term 1}) \times \ldots \times RF(\text{term k})]\)

  - **Implicit assumptions**: terms are independent and distribution is uniform!

But, how can we compute reduction factors?
Approximating Reduction Factors

- Reduction factors (RFs) can be \textit{approximated} using the statistics available in the DBMS’s catalog.

- For different \textit{forms} of terms, RF is computed differently:
  - \textbf{Form 1}: \textit{Column} = \textit{Value}
    - RF = \(1/\text{NKeys}(I)\), if there is an index \(I\) on \textit{Column}
    - Otherwise, RF = 1/10

\begin{itemize}
  \item NKeys(I)
  \item count
  \item grade
\end{itemize}

\textit{E.g.}, grade = ‘B’
Approximating Reduction Factors (Cont’d)

- For different forms of terms, RF is computed differently
  - **Form 2**: Column 1 = Column 2
    - RF = 1/\( \text{MAX}(\text{NKeys}(I_1), \text{NKeys}(I_2)) \), if there are indices \( I_1 \) and \( I_2 \) on Column 1 and Column 2, respectively
    - **Or**: RF = 1/\( \text{NKeys}(I) \), if there is only 1 index on Column 1 or Column 2
    - **Or**: RF = 1/10, if neither Column 1 nor Column 2 has an index
  - **Form 3**: Column IN (List of Values)
    - RF equals to RF of “Column = Value” (i.e., Form 1) \( \times \) # of elements in the List of Values
Approximating Reduction Factors (Cont’d)

- For different forms of terms, RF is computed differently
  - **Form 4: Column > Value**
    - RF = \( \frac{\text{High}(I) - \text{Value}}{(\text{High}(I) - \text{Low}(I))} \), if there is an index \( I \) on Column
  - Otherwise, RF equals to any fraction < \( \frac{1}{2} \)

E.g., grade >= ‘C’
Improved Statistics: Histograms

- Estimates can be improved considerably by maintaining more detailed statistics known as histograms.
Improved Statistics: Histograms

- Estimates can be improved considerably by maintaining more detailed statistics known as histograms.

![Histogram of Distribution D]

What is the result size of term value > 13?

8 tuples
Improved Statistics: Histograms

- Estimates can be improved considerably by maintaining more detailed statistics known as histograms.

Uniform Distribution Approximating D

What is the (estimated) result size of term value > 13?

(1/15 \times 44) = \sim 3 tuples

Clearly, this is inaccurate!
Improved Statistics: Histograms

- We can do better if we divide the range of values into *sub-ranges* called *buckets*

*Equiwidth* histogram

*Equidepth* histogram

- Uniform distribution per a bucket
- Equal # of tuples per a bucket
Improved Statistics: Histograms

- We can do better if we divide the range of values into *sub-ranges* called *buckets*

**Equiwidth histogram**

What is the *(estimated)* result size of *term value > 13*?

- The selected range = 1/3 of the range for bucket 5
- Bucket 5 represents a total of 15 tuples
- Estimated size = 1/3 \times 15 = 5 tuples

Better than regular histograms!
Improved Statistics: Histograms

- We can do better if we divide the range of values into *sub-ranges* called *buckets*

*Equidepth* histogram

What is the *(estimated)* result size of *term value > 13*?

- The selected range = 100% of the range for bucket 5
- Bucket 5 represents a total of 9 tuples
- Estimated size = $1 \times 9 = 9$ tuples

Better than *equiwidth* histograms!

*Why?*
Improved Statistics: Histograms

- We can do better if we divide the range of values into sub-ranges called **buckets**

**Equidepth histogram**

- Because, buckets with very frequently occurring values contain fewer slots; hence, the uniform distribution assumption is applied to a smaller range of values!

- What about buckets with mostly infrequent values? *They are approximated less accurately!*

---

**Histogram Details**

<table>
<thead>
<tr>
<th>Bucket</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bucket 1</td>
<td>9</td>
</tr>
<tr>
<td>Bucket 2</td>
<td>10</td>
</tr>
<tr>
<td>Bucket 3</td>
<td>10</td>
</tr>
<tr>
<td>Bucket 4</td>
<td>7</td>
</tr>
<tr>
<td>Bucket 5</td>
<td>9</td>
</tr>
</tbody>
</table>
Outline

A Brief Primer on Query Optimization

Query Evaluation Plans

Relational Algebra Equivalences

Estimating Plan Costs

Enumerating Plans

Nested Sub-Queries
Enumerating Execution Plans

- Consider a query \( Q = A \bowtie B \bowtie C \bowtie D \)

- Here are 3 plans that are equivalent:

Left-Deep Tree

Linear Trees

A Bushy Tree
Enumerating Execution Plans

- Consider a query $Q = A \bowtie B \bowtie C \bowtie D$

- Here are 3 plans that are equivalent:

Why?
Enumerating Execution Plans (Cont’d)

- There are two main reasons for concentrating only on left-deep plans:
  - As the number of joins increases, the number of plans increases rapidly; hence, it becomes necessary to prune the space of alternative plans
  - Left-deep trees allows us to generate all fully pipelined plans

- Clearly, by adding details to left-deep trees (e.g., the join algorithm per each join), several query plans can be obtained

- The query optimizer enumerates all possible left-deep plans using typically a dynamic programming approach (later), estimates the cost of each plan, and selects the one with the lowest cost!
Enumerating Execution Plans (Cont’d)

- In particular, the query optimizer enumerates:
  1. All possible left-deep orderings
  2. The different possible ways for evaluating each operator
  3. The different access paths for each relation

- Assume the following query $Q$:

```sql
SELECT S.sname, B.bname, R.day
FROM Sailors S, Reserves R, Boats B
```
Enumerating Execution Plans (Cont’d)

- In particular, the query optimizer enumerates:
  1. All possible left-deep orderings

```
S R B

R S B

B S R

R B S

B R S

S B R
```

Enumerating Execution Plans (Cont’d)

- In particular, the query optimizer enumerates:
  1. All possible left-deep orderings

Prune plans with cross-products immediately!
Enumerating Execution Plans (Cont’d)

- In particular, the query optimizer enumerates:
  1. All possible left-deep orderings
  2. The different possible ways for evaluating each operator
Enumerating Execution Plans (Cont’d)

- In particular, the query optimizer enumerates:
  1. All possible left-deep orderings
  2. The different possible ways for evaluating each operator

+ do same for the 3 other plans
Enumerating Execution Plans (Cont’d)

- In particular, the query optimizer enumerates:
  1. All possible left-deep orderings
  2. The different possible ways for evaluating each operator
  3. The different access paths for each relation

```
[Diagram showing execution plans with NLJ and B operators]
```
Enumerating Execution Plans (Cont’d)

- In particular, the query optimizer enumerates:
  1. All possible left-deep orderings
  2. The different possible ways for evaluating each operator
  3. The different access paths for each relation

+ do same for the 3 other plans
Enumerating Execution Plans (Cont’d)

- In particular, the query optimizer enumerates:
  1. All possible left-deep orderings
  2. The different possible ways for evaluating each operator
  3. The different access paths for each relation

Subsequently, estimate the cost of each plan using statistics collected and stored at the system catalog!

Let us now study a dynamic programming algorithm to effectively enumerate and estimate cost plans.
Towards a Dynamic Programming Algorithm

There are two main cases to consider:

- **CASE I: Single-Relation Queries**
  - Only *selection*, *projection*, *grouping* and *aggregate* operations are involved (i.e., no *joins*)
  - Every available access path is considered and the one with the least estimated cost is selected
  - The different operations are carried out together
    - E.g., if an index is used for a selection, projection can be done for each retrieved tuple, and the resulting tuples can be *pipelined* into an aggregate operation (if any)

- **CASE II: Multiple-Relation Queries**
CASE I: Single-Relation Queries
An Example

- Consider the following SQL query $Q$:

  \[
  \text{SELECT \ } S\text{.rating, COUNT (*)} \\
  \text{FROM \ Sailors } S \\
  \text{WHERE \ } S\text{.rating} > 5 \text{ AND } S\text{.age} = 20 \\
  \text{GROUP BY } S\text{.rating}
  \]

- $Q$ can be expressed in a relational algebra tree as follows:
CASE I: Single-Relation Queries - An Example

- Consider the following SQL query $Q$:

```sql
SELECT S.rating, COUNT(*)
FROM Sailors S
WHERE S.rating > 5 AND S.age = 20
GROUP BY S.rating
```

- How can $Q$ be evaluated?
  - Apply CASE I:
    - Every available access path for Sailors is considered and the one with the least estimated cost is selected
    - The selection and projection operations are carried out together
CASE I: Single-Relation Queries - An Example

- Consider the following SQL query $Q$:

```sql
SELECT S.rating, COUNT(*)
FROM Sailors S
WHERE S.rating > 5 AND S.age = 20
GROUP BY S.rating
```

- What would be the cost of we assume a file scan for sailors?
CASE I: Single-Relation Queries - An Example

- What would be the cost of we assume a file scan for sailors?

```plaintext
\( \sigma_{\text{rating} > 5 \land \text{age} = 20} \) (Scan; Write to Temp T1)

\( \text{GROUP BY} \) \( \text{rating} \) (External Sorting)

\( \text{rating, COUNT}(\ast) \) (on-the-fly)

NPages(Sailors)

\[ \text{Reduction Factor (RF) of S.rating} \times \text{Size of T1 tuple/Size of Sailors tuple} \times \text{Reduction Factor (RF) of S.age} \]

\[ \text{NPages(Sailors)} + \text{NPages(Sailors)} \times \text{Size of T1 tuple/Size of Sailors tuple} \times \text{Reduction Factor (RF) of S.rating} \times \text{Reduction Factor (RF) of S.age} \]
```
CASE I: Single-Relation Queries - An Example

- What would be the cost of we assume a file scan for sailors?

```
Sailors
rating > 5, age = 20 (Scan; Write to Temp T1)
GROUP BY rating (External Sorting)

rating, COUNT(*) (on-the-fly)
```

Term of Form 4 (default < 1/2)
Term of Form 1 (default = 1/10)

\[ \text{NPages(Sailors)} \times \frac{\text{Size of T1 tuple}}{\text{Size of Sailors tuple}} \times \text{RF of S.rating} \times \text{RF of S.age} + \text{NPages(Sailors)} \]
**CASE I: Single-Relation Queries - An Example**

- What would be the cost of we assume a file scan for sailors?

```
CASE I: Single-Relation Queries - An Example

- What would be the cost of we assume a file scan for sailors?

```

```
rating, COUNT(*) (on-the-fly)
GROUP BY rating (External Sorting)
rating (on-the-fly)

rating > 5
age = 20 (Scan; Write to Temp T1)

Sailors

Term of Form 4 (default < 1/2)
Term of Form 1 (default = 1/10)

NPages(Sailors) = 500 I/Os

+ NPages(Sailors) = 500 I/Os

× Size of T1 tuple/Size of Sailors tuple = 0.25

× Reduction Factor (RF) of S.rating = 0.2

× Reduction Factor (RF) of S.age = 0.1

= 502.5 I/Os
```
CASE I: Single-Relation Queries - An Example

- What would be the cost of we assume a **file scan** for sailors?

\[ \text{rating, COUNT(*) (on-the-fly)} \]

\[ \text{GROUP BY rating} \]

\[ \text{(External Sorting)} \]

\[ 3 \times \text{NPages(T1)} = 3 \times 2.5 = 7.5 \text{ I/Os} \]
CASE I: Single-Relation Queries- An Example

- What would be the cost of we assume a file scan for sailors?

```
Sailors
rating > 5 ∧ age = 20

rating, COUNT(*) (on-the-fly)
GROUP BY rating
(External Sorting)
7.5 I/Os

rating (on-the-fly)
(Scan; Write to Temp T1)
502.5 I/Os

= 510 I/Os
```
CASE I: Single-Relation Queries-An Example

- What would be the cost of we assume a clustered index on rating with A(1)?

\[
\begin{align*}
\sigma_{\text{rating} > 5 \land \text{age} = 20} \\
\text{Sailors} \\
\end{align*}
\]

\[
\begin{align*}
\text{rating, COUNT(\ast)} \text{ (on-the-fly)} \\
\text{GROUP BY rating} \text{ (External Sorting)} \\
\text{rating} \text{ (on-the-fly)} \\
\end{align*}
\]

\[
\begin{align*}
\text{Cost of retrieving the index entries} + \\
\text{Cost of retrieving the corresponding Sailors tuples} + \\
\text{Cost of writing out T1}
\end{align*}
\]
CASE I: Single-Relation Queries - An Example

- What would be the cost of if we assume a clustered index on rating with A(1)?

\[ \text{Term of Form 4} \]
\[
\text{RF} = \frac{\text{High}(I) - \text{Value}}{\text{High}(I) - \text{Low}(I)} = \frac{10 - 5}{10} = 0.5
\]

\[ \text{Term of Form 1. Can be applied to each retrieved tuple.} \]

\[ \text{Cost of retrieving the index entries} + \]
\[ \text{Cost of retrieving the corresponding Sailors tuples} + \]
\[ \text{Cost of writing out T1} \]
CASE I: Single-Relation Queries - An Example

- What would be the cost of we assume *a clustered index on rating with A(1)*?

\[
\text{Term of Form 4} \quad \text{RF} = \frac{\text{High}(I) - \text{Value}}{\text{High}(I) - \text{Low}(I)} = \frac{10 - 5}{10} = 0.5
\]

\[
\text{Term of Form 1. Can be applied to each retrieved tuple.}
\]

\[
\begin{align*}
&\text{Cost of retrieving the index entries} \\
&\text{+ Cost of retrieving the corresponding Sailors tuples} \\
&= 0.5 \times 0.1 \times \text{NPages}(I) \\
&= 0.5 \times 0.1 \times 600 \\
&= 30 \text{ I/Os}
\end{align*}
\]
CASE I: Single-Relation Queries - An Example

- What would be the cost of we assume *a clustered index on rating with A(1)*?

\[ \text{Cost of retrieving the index entries} + \text{Cost of retrieving the corresponding Sailors tuples} + \text{Cost of writing out T1} = 2 \times 30 = 60 \text{ I/Os} \]

Term of *Form 4*

Term of *Form 1*. Can be applied to each retrieved tuple.

\[ \text{RF} = \frac{\text{High}(I) - \text{Value}}{\text{High}(I) - \text{Low}(I)} = \frac{10 - 5}{10} = 0.5 \]
CASE I: Single-Relation Queries - An Example

- What would be the cost of we assume a clustered index on rating with A(1)?

\[
\sum_{\text{rating} > 5 \land \text{age} = 20} \sigma_{\text{Sailors}}
\]

\[\text{rating, COUNT(*) (on-the-fly)}\]

\[\text{(External Sorting)}\]

\[\approx 7.5 \text{ I/Os}\]

\[2 \times 30 = 60 \text{ I/Os}\]

67.5 I/Os (as opposed to 510 I/Os with a file scan)
Towards a Dynamic Programming Algorithm

- There are two main cases to consider:
  - CASE I: Single-Relation Queries
  - CASE II: Multiple-Relation Queries

- CASE II: Multiple-Relation Queries
  - Only consider left-deep plans
  - Apply a dynamic programming algorithm
Enumeration of Left-Deep Plans Using Dynamic Programming

Enumerate using $N$ passes (if $N$ relations joined):

- **Pass 1:**
  - For each relation, enumerate all plans (all 1-relation plans)
  - Retain the cheapest plan per each relation

- **Pass 2:**
  - Enumerate all 2-relation plans by considering each 1-relation plan retained in **Pass 1** (as outer) and successively every other relation (as inner)
  - Retain the cheapest plan per each 1-relation plan

- **Pass N:**
  - Enumerate all $N$-relation plans by considering each ($N-1$)-relation plan retained in **Pass N-1** (as outer) and successively every other relation (as inner)
  - Retain the cheapest plan per each ($N-1$)-relation plan

- **Pick the cheapest N-relation plan**
An \(N-1\) way plan is not combined with an additional relation unless:
- There is a join condition between them
- All predicates in the WHERE clause have been used up

ORDER BY, GROUP BY, and aggregate functions are handled as a final step, using either an `interestingly ordered` plan or an additional sorting operator

In spite of pruning plan space, this approach is still exponential in the # of tables
CASE II: Multiple-Relation Queries - An Example

- Consider the following relational algebra tree:

```
  ┌───┬───┐
  │   │   │
  └───┴───┘
     □  □
    sid=sid

  ┌ ┌─┐
  │ │ │
  └───┘
     ⊕
    σ bid=100

  ┌─┐
  │ │
  └───┘
     ⊕
    σ rating > 5

Reserves   Sailors
```

- Assume the following:

  - Sailors:
    - B+ tree on rating
    - Hash on sid
  - Reserves:
    - B+ tree on bid
CASE II: Multiple-Relation Queries - An Example

- **Pass 1:**
  - **Sailors:**
    - B+ tree matches rating > 5, and is *probably* the cheapest
    - If this selection is expected to retrieve a lot of tuples, and the index is un-clustered, file scan might be cheaper!
  - **Reserves:** B+ tree on *bid* matches *bid*=500; *probably* the cheapest

- **Sailors:**
  - B+ tree on *rating*
  - Hash on *sid*
- **Reserves:**
  - B+ tree on *bid*
CASE II: Multiple-Relation Queries - An Example

- **Pass 2:**
  - Consider each plan retained from **Pass 1** as the outer, and join it effectively with every other relation
  
- E.g., Reserves as outer:
  
  - Hash index can be used to get Sailors tuples that satisfy sid = outer tuple’s sid value

- Sailors:
  - B+ tree on rating
  - Hash on sid
- Reserves:
  - B+ tree on bid
Outline

A Brief Primer on Query Optimization

Query Evaluation Plans

Relational Algebra Equivalences

Estimating Plan Costs

Enumerating Plans

Nested Sub-Queries
Nested Sub-queries

- Consider the following nested query $Q_1$:

  $$\begin{align*}
  &\text{SELECT } S.\text{sname} \\
  &\text{FROM Sailors } S \\
  &\text{WHERE } S.\text{rating} = \\
  &\quad (\text{SELECT } \text{MAX} (S2.\text{rating}) \\
  &\quad \text{FROM Sailors } S2)
  \end{align*}$$

- The nested sub-query can be evaluated *just once*, yielding a single value $V$

- $V$ can be incorporated into the top-level query as if it had been part of the original statement of $Q_1$
Nested Sub-queries

- Now, consider the following nested query $Q2$:

```
SELECT S.sname
FROM Sailors S
WHERE EXISTS
  (SELECT R.sid
   FROM Reserves R
   WHERE R.bid=103)
```

- The nested sub-query can still be evaluated *just once*, but it will yield a collection of *sids*

- Every *sid* value in Sailors must be checked whether it exists in the collection of *sids* returned by the nested sub-query
  - This entails a join, and the full range of join methods can be explored!
Nested Sub-queries

- Now, consider another nested query **Q3**: 

  ```sql
  SELECT S.sname
  FROM Sailors S
  WHERE EXISTS
    (SELECT *
     FROM Reserves R
     WHERE R.bid=103
     AND R.sid=S.sid)
  ```

- **Q3 is correlated**; hence, we “cannot” evaluate the sub-query just once!

- In this case, the typical evaluation strategy is to evaluate the nested sub-query **for each tuple** of Sailors 

The common approach, indeed, is to **always** do nested loops join!
Summary

- Query optimization is a crucial task in a relational DBMSs

- We must understand query optimization in order to understand the performance impact of a given database design (relations, indexes) on a workload (set of queries)

- Two parts to optimizing a query:
  1. Consider a set of alternative plans (e.g., using dynamic programming)
     - Apply selections/projections as early as possible
     - Prune search space; typically, keep left-deep plans only
  2. Estimate the cost of each plan that is considered
     - Must estimate size of result and cost of each tree node
     - Key issues: Statistics, indexes, operator implementations
Next Class

Queries

- Query Optimization and Execution
- Relational Operators
- Files and Access Methods
- Buffer Management
- Disk Space Management

Transaction Manager
Lock Manager

Recovery Manager

DB