Database Applications (15-415)

DBMS Internals- Part X
Lecture 21, April 7, 2015

Mohammad Hammoud
Today...

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

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

- **Announcements:**
  - PS4 is due on Sunday, April 12 by midnight
  - Quiz II is on Thursday, April 9th (all concepts covered after the midterm are included)
DBMS Layers

Queries

Query Optimization and Execution

Relational Operators

Files and Access Methods

Buffer Management

Disk Space Management

DB

Transaction Manager

Lock Manager

Recovery Manager

Continue…
Outline

A Brief Primer on Query Optimization

Query Evaluation Plans

Relational Algebra Equivalences

Estimating Plan Costs

Enumerating Plans

Nested Sub-Queries

Last Session
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 set 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:

\[
\text{SELECT attribute list} \\
\text{FROM } R_1, R_2, \ldots, R_n \\
\text{WHERE } \text{term 1 AND ... AND term k}
\]

- What is the *maximum* number of tuples generated by \( QB \)?
  - \( \text{NTuples}(R_1) \times \text{NTuples}(R_2) \times \ldots \times \text{NTuples}(R_n) \)

- Every term in the WHERE clause, however, eliminates some of the possible resultant tuples
  - A *reduction factor* can be associated with each term
Consider a query block, \(QB\), of the form:

\[
\text{SELECT attribute list}
\text{FROM } R_1, R_2, \ldots, R_n
\text{WHERE } \text{term 1 AND } \ldots \text{ AND term k}
\]

The \textit{reduction factor} (RF) associated with each \textit{term} reflects the impact of the \textit{term} in reducing the result size.

Final (\textit{estimated}) result cardinality = \([\text{NTuples}(R_1) \times \ldots \times \text{NTuples}(R_n)] \times [\text{RF(term 1)} \times \ldots \times \text{RF(term k)}] \]

\textit{Implicit assumptions: terms are independent and distribution is uniform!}

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

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

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

![Diagram of grade distribution and NKeys(I)]
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 < 1/2

\[ \text{E.g., grade } \geq \text{‘C’} \]
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.

**Distribution D**

What is the result size of term value > 13?

9 tuples
Improved Statistics: Histograms

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

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

\[
\frac{1}{15} \times 45 = 3 \text{ tuples}
\]

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**

<table>
<thead>
<tr>
<th>Bucket</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>15</td>
</tr>
</tbody>
</table>

**Equidepth histogram**

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

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 × 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!*

**Equidepth histogram**

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

Equal # of tuples per a bucket
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 $Q = A \Join B \Join C \Join D$

- 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 allow 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
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]

1. **S** → **R** → **B**
2. **S** → **R** → **B** (heap scan)
3. **S** → **R** → **B** (heap scan)
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
  - CASE II: Multiple-Relation Queries

- **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 I: Single-Relation Queries - An Example

- Consider the following SQL query \( Q \):

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

- \( Q \) 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?

**Sailors**

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

\[ \text{(Scan; Write to Temp T1)} \]

\[ \text{GROUP BY} \quad \text{rating} \]

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

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

\[ \text{NPages(Sailors)} \]

\[ + \]

\[ \times \]

\[ \text{Size of T1 tuple/Size of Sailors tuple} \]

\[ \times \]

\[ \text{Reduction Factor (RF) of S.rating} \]

\[ \times \]

\[ \text{Reduction Factor (RF) of S.age} \]
CASE I: Single-Relation Queries - An Example

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

\[
\text{TERM of Form 4 (default } < 1/2) + \left( \frac{\text{NPages(Sailors)}}{\text{Size of T1 tuple/Size of Sailors tuple}} \right) \times \left( \left( \frac{\text{Reduction Factor (RF) of S.rating}}{\text{Reduction Factor (RF) of S.age}} \right) \right)
\]
**CASE I: Single-Relation Queries - An Example**

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

```
Sailors
  rating > 5 {Scan; Write to Temp T1} (on-the-fly)
  age = 20 (on-the-fly)

GROUP BY rating

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

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

\[
\text{NPages(Sailors)} = \frac{500}{\text{I/Os}}
\]

\[
+ \quad \frac{500}{\text{I/Os}} \times \frac{\text{Size of T1 tuple}}{\text{Size of Sailors tuple}} \times \left( \text{Reduction Factor (RF) of S.rating} = 0.2 \right) \times \left( \text{Reduction Factor (RF) of S.age} = 0.1 \right)
\]

\[
= 502.5 \text{ I/Os}
\]
CASE I: Single-Relation Queries - An Example

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

```
Sailors
  age = 20
  rating > 5

rating, COUNT(*) (on-the-fly)
GROUP BY rating
(External Sorting) --> 3 \times \text{NPages(T1)} = 3 \times 2.5 = 7.5 \text{ I/Os}
```

(Scan; Write to Temp T1)
**CASE I: Single-Relation Queries**

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

*Sailors*

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

- **(Scan; Write to Temp T1)**
  - **(External Sorting) on-the-fly**
  - 7.5 I/Os

- **(Scan; Write to Temp T1)**
  - 502.5 I/Os

**Total:** 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)?

\[
\sigma_{\text{rating} > 5 \land \text{age} = 20} \left( \pi_{\text{rating}, \text{COUNT}(\ast)} (\text{on-the-fly}) \right) \]

<table>
<thead>
<tr>
<th>(Index; Write to Temp T1)</th>
</tr>
</thead>
</table>

- Cost of retrieving the index entries
- Cost of retrieving the corresponding Sailors tuples
- 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)?

Term of Form 4

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

RF = (High(I) – Value)/(High(I) – Low(I)) = (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)?

\[
\text{Term of Form 4} \quad \text{Term of Form 1. Can be applied to each retrieved tuple.}
\]

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

\[
\text{Cost of retrieving the index entries} + \text{Cost of retrieving the corresponding Sailors tuples} = 0.5 \times 0.1 \times \text{NPages(I)}
\]

\[
= 0.5 \times 0.1 \times 600
\]

\[
= 30 \text{ I/Os}
\]
CASE I: Single-Relation Queries - An Example

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

Term of Form 4

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

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

**Sailors**

\( \text{age} = 20 \)

\( \text{rating} > 5 \)

\( \text{rating} \) (on-the-fly)

\( \text{COUNT}(*) \) (on-the-fly)

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

(External Sorting)

Cost of retrieving the index entries

+ 

Cost of retrieving the corresponding Sailors tuples

+ 

Cost of writing out T1

= 

\( 2 \times 30 = 60 \text{ I/Os} \)
CASE I: Single-Relation Queries - An Example

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

\[
\begin{align*}
\text{\textbf{Sailors}} & \quad \text{\textbf{rating > 5 \land age = 20}} \\
& \quad \text{\textbf{rating \land COUNT(* \text{(on-the-fly))}}} \\
& \quad \text{\textbf{GROUP BY rating \text{(on-the-fly)}}} \\
& \quad \text{\textbf{rating \text{(on-the-fly)}}} \\
\end{align*}
\]

\( \implies 67.5 \text{ I/Os (as opposed to 510 I/Os with a file scan)} \)
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**
Enumeration of Left-Deep Plans Using Dynamic Programming (Cont’d)

- 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

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

- Consider the following relational algebra tree:

\[
\begin{align*}
\Pi_{\text{snames}} \\
\bowtie_{\text{sid} = \text{sid}} \\
\sigma_{\text{bid} = 100} \\
\sigma_{\text{rating} > 5}
\end{align*}
\]

- 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
**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 $Q1$:

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

- 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 $Q1$
Nested Sub-queries

- Now, consider the following nested query $Q_2$:

```
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 $Q_3$:

  $$
  \begin{align*}
  \text{SELECT} & \quad \text{sname} \\
  \text{FROM} & \quad \text{Sailors S} \\
  \text{WHERE} & \quad \text{EXISTS} \\
  & \quad (\text{SELECT} \quad * \\
  & \quad \text{FROM} \quad \text{Reserves R} \\
  & \quad \text{WHERE} \quad \text{R.bid=103} \\
  & \quad \text{AND} \quad \text{R.sid=S.sid})
  \end{align*}
  $$

- $Q_3$ 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.
Summary

- Query optimization is a crucial task in 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

DB

Transaction Manager

Lock Manager

Recovery Manager