Lecture 20: Cost-Based Query Optimization
Recap
Query Optimization

• **Approach 1: Heuristics / Rules**
  ▶ Rewrite the query to remove stupid / inefficient things.
  ▶ These techniques may need to examine catalog, but they do **not** need to examine data.

• **Approach 2: Cost-based Search**
  ▶ Use a model to estimate the cost of executing a plan.
  ▶ Evaluate multiple equivalent plans for a query and pick the one with the lowest cost.
Today’s Agenda

- Plan Cost Estimation
- Plan Enumeration
Plan Cost Estimation
Cost Estimation

• How long will a query take?
  ▶ CPU: Small cost; tough to estimate
  ▶ Disk: Number of block transfers
  ▶ Memory: Amount of DRAM used
  ▶ Network: Number of messages

• How many tuples will be read/written?

• It is too expensive to run every possible plan to determine this information, so the DBMS need a way to derive this information. . .
Statistics

- The DBMS stores internal statistics about tables, attributes, and indexes in its internal catalog.
- Different systems update them at different times.
- Manual invocations:
  - Postgres/SQLite: ANALYZE
  - Oracle/MySQL: ANALYZE TABLE
  - SQL Server: UPDATE STATISTICS
  - DB2: RUNSTATS
Statistics

• For each relation $R$, the DBMS maintains the following information:
  > $N_R$: Number of tuples in $R$.
  > $V(A, R)$: Number of distinct values for attribute $A$. 
Derivable Statistics

- The **selection cardinality** $SC(A, R)$ is the average number of records with a value for an attribute $A$ is given by: $NR / V(A, R)$
- What could go wrong with this estimate?
Derivable Statistics

- The **selection cardinality** $SC(A, R)$ is the average number of records with a value for an attribute $A$ is given by: $NR / V(A, R)$
- Note that this assumes **data uniformity**.
  - 10,000 students, 10 colleges – how many students in SCS?
Selection Statistics

- Equality predicates on unique keys are easy to estimate.
- What about more complex predicates? What is their selectivity?

```sql
CREATE TABLE people (  
id INT PRIMARY KEY,  
val INT NOT NULL,  
age INT NOT NULL,  
status VARCHAR(16)
);

SELECT * FROM people WHERE id = 123  --- Easier
SELECT * FROM people WHERE val > 1000  --- Harder: Range predicate
SELECT * FROM people WHERE age = 30 AND status = 'Lit'  --- Harder:
Complex predicate
```
Complex Predicates

- The **selectivity** (sel) of a predicate $P$ is the fraction of tuples that qualify.
- Formula depends on type of predicate:
  - Equality
  - Range
  - Negation
  - Conjunction
  - Disjunction
Selection – Complex Predicates

• Assume that $V(\text{age,people})$ has five distinct values (0–4) and $N_R = 5$
• Equality Predicate: $A=\text{constant}$
  ▶ $\text{sel}(A=\text{constant}) = \frac{\text{SC}(P)}{N_R}$
  ▶ Example: $\text{sel}(\text{age}=2) = \frac{1}{5}$

```
SELECT * FROM people WHERE age = 2
```
Selection – Complex Predicates

- **Range Predicate:**
  - $\text{sel}(A \geq a) = \frac{(A_{\text{max}} - a)}{(A_{\text{max}} - A_{\text{min}})}$
  - Example: $\text{sel}(\text{age} \geq 2) \approx \frac{(4 - 2)}{(4 - 0)} \approx \frac{1}{2}$

```sql
SELECT * FROM people WHERE age >= 2
```
Selection – Complex Predicates

- **Negation Query:**
  - \( \text{sel}(\neg P) = 1 - \text{sel}(P) \)
  - Example: \( \text{sel}(\text{age} \neq 2) = 1 - (1/5) = 4/5 \)

- **Observation:** Selectivity \( \approx \) Probability

\[
\text{SELECT } * \text{ FROM } \text{people} \text{ WHERE } \text{age} \neq 2
\]
Selection – Complex Predicates

- **Conjunction:**
  - $\text{sel}(P1 \land P2) = \text{sel}(P1) \times \text{sel}(P2)$
  - $\text{sel}(\text{age} = 2 \land \text{name LIKE 'A%'}$)

- This assumes that the predicates are **independent**.
- Not always true in practice!

```sql
SELECT * FROM people WHERE age = 2 AND name LIKE 'A%'
```
Selection – Complex Predicates

- **Disjunction:**
  - \( \text{sel}(P_1 \lor P_2) = \text{sel}(P_1) + \text{sel}(P_2) - \text{sel}(P_1 \land P_2) = \text{sel}(P_1) + \text{sel}(P_2) - \text{sel}(P_1) \times \text{sel}(P_2) \)
  - \( \text{sel}(\text{age}=2 \lor \text{name LIKE 'A%'}) \)

- This again assumes that the selectivities are independent.

\[
\text{SELECT * FROM people WHERE age = 2 OR name LIKE 'A%'}
\]
Selection Cardinality

- **Assumption 1: Uniform Data**
  - The distribution of values (except for the heavy hitters) is the same.

- **Assumption 2: Independent Predicates**
  - The predicates on attributes are independent

- **Assumption 3: Inclusion Principle**
  - The domain of join keys overlap such that each key in the inner relation will also exist in the outer table.
Correlated Attributes

• Consider a database of automobiles:
  ▶ Number of Makes = 10, Number of Models = 100
• And the following query: \(\text{make} = "Honda" \text{ANDmodel} = "Accord"\)
• With the independence and uniformity assumptions, the selectivity is:
  ▶ \(1/10 \times 1/100 = 0.001\)
• But since only Honda makes Accords, the real selectivity is \(1/100 = 0.01\)
Cost Estimation

- Our formulas are nice, but we assume that data values are uniformly distributed.
Cost Estimation

- Our formulas are nice, but we assume that data values are uniformly distributed.
Cost Estimation

- Our formulas are nice, but we assume that data values are uniformly distributed.
Histories With Quantiles

- Vary the **width of buckets** so that the total number of occurrences for each bucket is roughly the same.
Histograms With Quantiles

- Vary the **width of buckets** so that the total number of occurrences for each bucket is roughly the same.
Sampling

- Modern DBMSs also collect samples from tables to estimate selectivities.
- Update samples when the underlying tables changes significantly.
- Example: 1 billion tuples

```
SELECT AVG(age) FROM people WHERE age > 50
```

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>age</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>Shiyi</td>
<td>58</td>
<td>Senior</td>
</tr>
<tr>
<td>1002</td>
<td>Rahul</td>
<td>41</td>
<td>Sophomore</td>
</tr>
<tr>
<td>1003</td>
<td>Peter</td>
<td>25</td>
<td>Freshman</td>
</tr>
<tr>
<td>1004</td>
<td>Mark</td>
<td>25</td>
<td>Junior</td>
</tr>
<tr>
<td>1005</td>
<td>Alice</td>
<td>38</td>
<td>Senior</td>
</tr>
</tbody>
</table>
Sampling

- Modern DBMSs also collect samples from tables to estimate selectivities.
- Update samples when the underlying tables changes significantly.
- Example: 1 billion tuples
- \( \text{sel}(\text{age}>50) = \frac{1}{3} \)

\[
\text{SELECT AVG(age) FROM people WHERE age > 50}
\]

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>age</th>
<th>status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>Shiyi</td>
<td>58</td>
<td>Senior</td>
</tr>
<tr>
<td>1003</td>
<td>Mark</td>
<td>25</td>
<td>Junior</td>
</tr>
<tr>
<td>1005</td>
<td>Alice</td>
<td>38</td>
<td>Senior</td>
</tr>
</tbody>
</table>
Observation

- Now that we can (roughly) estimate the **selectivity of predicates**, what can we actually do with them?
Plan Enumeration
Query Optimization

- After performing rule-based rewriting, the DBMS will enumerate different plans for the query and estimate their costs.
  - Single relation
  - Multiple relations
- It chooses the best plan it has seen for the query after exhausting all plans or some timeout.
Single-Relation Query Planning

- Pick the best access method.
  - Sequential Scan
  - Binary Search (clustered indexes)
  - Index Scan
- Predicate evaluation ordering.
- Simple heuristics are often good enough for this.
- OLTP queries are especially easy...
OLTP Query Planning

- Query planning for OLTP queries is easy because they are **sargable** (Search Argument Able).
  - It is usually just picking the best index.
  - Joins are almost always on foreign key relationships with a small cardinality.
  - Can be implemented with simple heuristics.

```sql
CREATE TABLE people (
    id INT PRIMARY KEY,
    val INT NOT NULL,
);

SELECT * FROM people WHERE id = 123;
```
Multi-Relation Query Planning

• As number of joins increases, number of alternative plans grows rapidly
  ▶ We need to restrict search space.
• Fundamental decision in System R: only left-deep join trees are considered.
  ▶ Modern DBMSs do not always make this assumption anymore.
Multi-Relation Query Planning

- Fundamental decision in System R: Only consider left-deep join trees.
Multi-Relation Query Planning

- Fundamental decision in System R: Only consider **left-deep join trees**.
Multi-Relation Query Planning

- Fundamental decision in System R: Only consider left-deep join trees.
- Allows for **fully pipelined** plans where intermediate results are not written to temp files.
  - Not all left-deep trees are fully pipelined.
Multi-Relation Query Planning

- Enumerate the orderings
  - Example: Left-deep tree 1, Left-deep tree 2 . . .
- Enumerate the physical join operator for each logical join operator
  - Example: Hash, Sort-Merge, Nested Loop . . .
- Enumerate the access paths for each table
  - Example: Index 1, Index 2, Seq Scan . . .
- Use dynamic programming to reduce the number of cost estimations.
Dynamic Programming
Dynamic Programming

Hash Join
R.a=S.a
Cost: 300

SortMerge Join
R.a=S.a
Cost: 400

SortMerge Join
T.b=S.b
Cost: 200

Hash Join
T.b=S.b
Cost: 200

R ⊘ S ⊘ T
Dynamic Programming
Dynamic Programming
Dynamic Programming
Candidate Plan Example

- How to generate plans for search algorithm:
  - Enumerate relation orderings
  - Enumerate join algorithm choices
  - Enumerate access method choices

- No real DBMSs does it this way. It’s actually more messy...

```
SELECT * FROM R, S, T
WHERE R.a = S.a AND S.b = T.b
```
Candidate Plans

- Step 1: Enumerate relation orderings

Step #3: Enumerate access method choices

Do this for the other plans.
Candidate Plans

- Step 2: Enumerate join algorithm choices

Step #2: Enumerate join algorithm choices

Do this for the other plans.
Candidate Plans

- Step 3: Enumerate access method choices

Step #1: Enumerate relation orderings

- Prune plans with cross-products immediately!
Postgres Optimizer

- Examines all types of join trees
  - Left-deep, Right-deep, bushy
- Two optimizer implementations:
  - Traditional Dynamic Programming Approach
  - Genetic Query Optimizer (GEQO)
- Postgres uses the traditional algorithm when number of tables in query is less than 12 and switches to GEQO when there are 12 or more.
Postgres Optimizer
Postgres Optimizer

Cost-Based Query Optimization

Plan Enumeration
Postgres Optimizer

1st Generation

2nd Generation

3rd Generation
Conclusion
Parting Thoughts

- Selectivity estimations
- Key assumptions in query optimization
  - Uniformity
  - Independence
  - Histograms
  - Join selectivity
- Dynamic programming for join orderings
Next Class

• Design Decisions in Query Optimization