Query Optimization
ADMINISTRIVIA

Homework #4 is due TODAY @ 11:59pm

Mid-term Exam is on Wednesday October 18th (in class)

Project #2 is due Wednesday October 25th @ 11:59am
QUERY OPTIMIZATION

Remember that SQL is declarative.
→ User tells the DBMS what answer they want, not how to get the answer.

There can be a big difference in performance based on plan is used:
→ See last week: 1.3 hours vs. 0.45 seconds
IBM SYSTEM R

First implementation of a query optimizer.
People argued that the DBMS could never choose a query plan better than what a human could write.
A lot of the concepts from System R’s optimizer are still used today.
QUERY OPTIMIZATION

Heuristics / Rules
→ Rewrite the query to remove stupid / inefficient things.
→ Does not require a cost model.

Cost-based Search
→ Use a cost model to evaluate multiple equivalent plans and pick the one with the lowest cost.
TODAY'S AGENDA

Relational Algebra Equivalences
Plan Cost Estimation
Plan Enumeration
Nested Sub-queries
Mid-Term
RELATIONAL ALGEBRA EQUIVALENCES

Two relational algebra expressions are equivalent if they generate the same set of tuples.

The DBMS can identify better query plans without a cost model. This is often called query rewriting.
SELECT s.name, e.cid  
FROM student AS s, enrolled AS e  
WHERE s.sid = e.sid  
AND e.grade = 'A'
RELATIONAL ALGEBRA EQUIVALENCES

\[
\pi_{\text{name}, \text{cid}}(\sigma_{\text{grade}=\text{A}}(\text{student} \bowtie \text{enrolled}))
\]

\[
= \pi_{\text{name}, \text{cid}}(\text{student} \bowtie (\sigma_{\text{grade}=\text{A}}(\text{enrolled})))
\]
RELATIONAL ALGEBRA EQUIVALENCES

Selections:
→ Perform filters as early as possible
→ Break a complex predicate, and push down
\[ \sigma_{p_1 \land p_2 \land \ldots \land p_n}(R) = \sigma_{p_1}(\sigma_{p_2}(\ldots \sigma_{p_n}(R))) \]

Simplify a complex predicate
→ \((X = Y \text{ AND } Y = 3) \rightarrow X = 3 \text{ AND } Y = 3\)
RELATIONAL ALGEBRA EQUVALENCE

Projections:
→ Perform them early to create smaller tuples and reduce intermediate results (if duplicates are eliminated)
→ Project out all attributes except the ones requested or required (e.g., joining attr.)

This is not important for a column store...
SELECT s.name, e.cid 
FROM student AS s, enrolled AS e 
WHERE s.sid = e.sid 
AND e.grade = 'A'
MORE EXAMPLES

Impossible / Unnecessary Predicates

\[
\text{SELECT } \ast \text{ FROM table WHERE } 1 = 0
\]

\[
\text{SELECT } \ast \text{ FROM table WHERE } 1 = 1
\]

Join Elimination

\[
\text{SELECT } A1.\ast \\
\text{ FROM } A \text{ AS A1 JOIN } A \text{ AS A2} \\
\text{ ON A1.id = A2.id}
\]
MORE EXAMPLES

Ignoring Projections

```
SELECT * FROM A AS A1
WHERE EXISTS(SELECT * FROM A AS A2
             WHERE A1.id = A2.id)
```

Merging Predicates

```
SELECT * FROM A
WHERE val BETWEEN 1 AND 100
     AND val BETWEEN 50 AND 150
```
RELATIONAL ALGEBRA EQUIVALENCES

Joins:
→ Commutative, associative

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

How many different orderings are there for an \( n \)-way join?
How many different orderings are there for an $n$-way join?

**Catalan number** $\approx 4^n$  
→ Exhaustive enumeration will be too slow.

We’ll see in a second how an optimizer limits the search space...
COST ESTIMATION

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

How many tuples will be read/written?
What statistics do we need to keep?
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
→ MySQL: ANALYZE TABLE
STATISTICS

For each relation $R$, the DBMS maintains the following information:

$\rightarrow N_R \rightarrow \#$ tuples

$\rightarrow V(A, R) \rightarrow \#$ of distinct values of attribute $A$
DERIVABLE STATISTICS

The **selection cardinality** \((SC(A, R))\) is the average number of records with a value for an attribute \(A\) given \(N_R / V(A, R)\)

Note that this assumes **data uniformity**

\[ \rightarrow \text{10,000 students, 10 colleges – how many students in SCS?} \]
SELECTION STATISTICS

Equality predicates on unique keys are easy to estimate.

What about more complex predicates? What is their selectivity?
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
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
SELECT * FROM people WHERE age = 2

Assume that $V(\text{age, people})$ has 5 distinct values (0–4) and $N_R = 5$

Equality Predicate: $A = \text{constant}$

$\rightarrow \text{sel}(A=\text{constant}) = SC(P) / V(A,R)$
Assume that $V(\text{age}, \text{people})$ has 5 distinct values (0–4) and $N_R = 5$

Equality Predicate: $A=\text{constant}$

$\rightarrow \text{sel}(A=\text{constant}) = \frac{SC(P)}{V(A,R)}$

$\rightarrow$ Example: $\text{sel}(\text{age}=2) =$

```
SELECT * FROM people
WHERE age = 2
```
Assume that $V(\text{age, people})$ has 5 distinct values (0–4) and $N_R = 5$

Equality Predicate: $A=\text{constant}$

$\rightarrow \text{sel}(A=\text{constant}) = \frac{SC(P)}{V(A,R)}$

$\rightarrow$ Example: $\text{sel}(\text{age}=2) =$

```
SELECT * FROM people
WHERE age = 2
```
Assume that $V(\text{age, people})$ has 5 distinct values (0–4) and $N_R = 5$

Equality Predicate: $A=\text{constant}$

$\rightarrow$ $sel(A=\text{constant}) = \frac{SC(P)}{V(A,R)}$

$\rightarrow$ Example: $sel(\text{age}=2) = \frac{1}{5}$
Range Query:

\[ \text{sel}(A \geq a) = \frac{(A_{\text{max}} - a)}{(A_{\text{max}} - A_{\text{min}})} \]

\[ \rightarrow \text{Example: sel(age} \geq 2) \]

```sql
SELECT * FROM people
WHERE age >= 2
```
Range Query:

\[ \text{sel}(A \geq a) = \frac{(A_{\text{max}} - a)}{(A_{\text{max}} - A_{\text{min}})} \]

Example: \( \text{sel}(\text{age} \geq 2) = \frac{(4 - 2)}{(4 - 0)} = \frac{1}{2} \)

```
SELECT * FROM people
WHERE age >= 2
```
Negation Query:
→ sel(not P) = 1 - sel(P)
→ Example: sel(age != 2)

```
SELECT * FROM people
WHERE age != 2
```
**SELECTIONS – COMPLEX PREDICATES**

**Negation Query:**

→ sel(not \(P\)) = 1 - sel(P)

→ Example: \(sel(age \neq 2)\)

**SELECT * FROM people**

**WHERE age \neq 2**
SELECTIONS – COMPLEX PREDICATES

Negation Query:

\[
\rightarrow \text{sel(not } P) = 1 - \text{sel}(P)
\]

\[
\rightarrow \text{Example: sel(age } \neq 2) = 1 - (1/5) = 4/5
\]

Observation: selectivity = probability

\[
\text{SELECT } * \text{ FROM people}
\]

\[
\text{WHERE age } \neq 2
\]
**SELECTIONS – COMPLEX PREDICATES**

Conjunction:

→ \( \text{sel}(P_1 \land P_2) = \text{sel}(P_1) \cdot \text{sel}(P_2) \)

→ \( \text{sel}(\text{age}=2 \land \text{name LIKE } 'A\%') \)

This assumes that the predicates are independent.

```
SELECT * FROM people
WHERE age = 2
AND name LIKE 'A%'
```
Disjunction:
\[ \text{sel}(P_1 \lor P_2) = \text{sel}(P_1) + \text{sel}(P_2) - \text{sel}(P_1 \lor P_2) = \text{sel}(P_1) + \text{sel}(P_2) - \text{sel}(P_1) \cdot \text{sel}(P_2) \]
\[ \rightarrow \text{sel}(\text{age}=2 \text{ OR name LIKE 'A%'}) \]

This again assumes that the selectivities are independent.
Disjunction:

\[ \text{sel}(P_1 \lor P_2) \]
\[ = \text{sel}(P_1) + \text{sel}(P_2) - \text{sel}(P_1 \lor P_2) \]
\[ = \text{sel}(P_1) + \text{sel}(P_2) - \text{sel}(P_1) \cdot \text{sel}(P_2) \]
\[ \rightarrow \text{sel}(\text{age}=2 \ \text{OR} \ \text{name} \ \text{LIKE} \ 'A\%') \]

This again assumes that the selectivities are independent.
RESULT SIZE ESTIMATION FOR JOINS

Given a join of \( R \) and \( S \), what is the range of possible result sizes in \# of tuples?

In other words, for a given tuple of \( R \), how many tuples of \( S \) will it match?
RESULT SIZE ESTIMATION FOR JOINS

General case: $R_{cols} \cap S_{cols} = \{A\}$ where A is not a key for either table.

→ Match each $R$-tuple with $S$-tuples:
  
  $\text{estSize} \approx N_R \cdot N_S / V(A,S)$

→ Symmetrically, for $S$:
  
  $\text{estSize} \approx N_R \cdot N_S / V(A,R)$

Overall:

→ $\text{estSize} \approx N_R \cdot N_S / \max\{V(A,S), V(A,R)\}$
COST ESTIMATIONS

Our formulas are nice but we assume that data values are uniformly distributed.

**Uniform Approximation**

# of occurrences

Distinct values of attribute
COST ESTIMATIONS

Our formulas are nice but we assume that data values are uniformly distributed.

Non-Uniform Approximation
Our formulas are nice but we assume that data values are uniformly distributed.
HISTOGRAMS WITH QUANTILES

A histogram type wherein the “spread” of each bucket is same.

Equi-width Histogram (Quantiles)
HISTOGRAMS WITH QUANTILES

A histogram type wherein the "spread" of each bucket is same.

Equi-width Histogram (Quantiles)

0  5  10  15

1-5  6-8  9-13  14-15
Modern DBMSs also employ sampling to estimate predicate selectivities.

```sql
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>Obama</td>
<td>56</td>
<td>Rested</td>
</tr>
<tr>
<td>1002</td>
<td>Kanye</td>
<td>40</td>
<td>Weird</td>
</tr>
<tr>
<td>1003</td>
<td>Tupac</td>
<td>25</td>
<td>Dead</td>
</tr>
<tr>
<td>1004</td>
<td>Bieber</td>
<td>23</td>
<td>Crunk</td>
</tr>
<tr>
<td>1005</td>
<td>Andy</td>
<td>36</td>
<td>Lit</td>
</tr>
</tbody>
</table>

1 billion tuples
Modern DBMSs also employ sampling to estimate predicate selectivities.

\[
\text{sel}(\text{age}>50) = \frac{1}{3}
\]

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

Now that we can (roughly) estimate the selectivity of predicates, what can we actually do with them?
QUERY OPTIMIZATION

Bring query in internal form into "canonical form" (syntactic q-opt)
Generate alternative plans.
→ Single relation.
→ Multiple relations.
→ Nested sub-queries.
Estimate cost for each plan.
Pick the best one.
SINGLE-RELATION QUERY PLANNING

Pick the best access method.
→ Sequential Scan
→ Binary Search (clustered indexes)
→ Index Scan

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.
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 plans for each operator
→ Example: Hash, Sort-Merge, Nested Loop...

Enumerate the access paths for each table
→ Example: Index #1, Index #2, Seq Scan...
MULTI-RELATION QUERY PLANNING

Enumerate the orderings
→ Example: Left-deep tree #1, Left-deep tree #2...

Enumerate the plans for each 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.
SELECT * FROM R, S, T
WHERE R.a = S.a
AND S.b = T.b

DYNAMIC PROGRAMMING
SELECT * FROM R, S, T
WHERE R.a = S.a
AND S.b = T.b
DYNAMIC PROGRAMMING

Hash Join
R.a=S.a

Hash Join
T.b=S.b

SELECT * FROM R, S, T
WHERE R.a = S.a
AND S.b = T.b
SELECT * FROM R, S, T
WHERE R.a = S.a
AND S.b = T.b
SELECT * FROM R, S, T
WHERE R.a = S.a
AND S.b = T.b
SELECT * FROM R, S, T
WHERE R.a = S.a
AND S.b = T.b
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...
CANDIDATE PLANS

Step #1: Enumerate relation orderings

Prune plans with cross-products immediately!

SELECT * FROM R, S, T
WHERE R.a = S.a
AND S.b = T.b
Step #1: Enumerate relation orderings

Prune plans with cross-products immediately!

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

Prune plans with cross-products immediately!

CANDIDATE PLANS

Step #1: Enumerate relation orderings
CANDIDATE PLANS

Step #2: Enumerate join algorithm choices

Do this for the other plans.
Step #2: Enumerate join algorithm choices

Do this for the other plans.
SELECT * FROM R, S, T
WHERE R.a = S.a
AND S.b = T.b

Candidate Plans

Step #3: Enumerate access method choices

Do this for the other plans.
NESTED SUB-QUERIES

The DBMS treats nested sub-queries in the where clause as functions that take parameters and return a single value or set of values.

Two Approaches:
→ Rewrite to de-correlate and/or flatten them
→ Decompose nested query and store result to temporary table
SELECT name FROM sailors AS S
WHERE EXISTS (
    SELECT * FROM reserves AS R
    WHERE S.sid = R.sid
    AND R.day = '2017-10-11'
)

SELECT name
FROM sailors AS S, reserves AS R
WHERE S.sid = R.sid
AND R.day = '2017-10-11'
For each sailor with the highest rating (over all sailors) and at least two reservations for red boats, find the sailor id and the earliest date on which the sailor has a reservation for a red boat.
DECOMPOSING QUERIES

For harder queries, the optimizer breaks up queries into blocks and then concentrates on one block at a time.

Sub-queries are written to a temporary table that are discarded after the query finishes.
```sql
SELECT S.sid, MIN(R.day)
FROM sailors S, reserves R, boats B
WHERE S.sid = R.sid
  AND R.bid = B.bid
  AND B.color = 'red'
  AND S.rating = (SELECT MAX(S2.rating)
                  FROM sailors S2)
GROUP BY S.sid
HAVING COUNT(*) > 1
```
SELECT MAX(rating) FROM sailors

SELECT S.sid, MIN(R.day)  
FROM sailors S, reserves R, boats B  
WHERE S.sid = R.sid  
AND R.bid = B.bid  
AND B.color = 'red'  
AND S.rating = (SELECT MAX(S2.rating)  
FROM sailors S2)  
GROUP BY S.sid  
HAVING COUNT(*) > 1

Nested Block
**DECOMPOSED QUERIES**

```
SELECT MAX(rating) FROM sailors

SELECT S.sid, MIN(R.day)
    FROM sailors S, reserves R, boats B
WHERE S.sid = R.sid
    AND R.bid = B.bid
    AND B.color = 'red'
    AND S.rating = ###
GROUP BY S.sid
HAVING COUNT(*) > 1
```
**DECOMPOSING QUERIES**

```
SELECT MAX(rating) FROM sailors

SELECT S.sid, MIN(R.day)
FROM sailors S, reserves R, boats B
WHERE S.sid = R.sid
    AND R.bid = B.bid
    AND B.color = 'red'
    AND S.rating = ###
GROUP BY S.sid
HAVING COUNT(*) > 1
```
CONCLUSION

Filter early as possible.

Selectivity estimations
→ Uniformity
→ Independence
→ Histograms
→ Join selectivity

Dynamic programming for join orderings

Rewrite nested queries

Query optimization is really hard...
Midterm Exam

Who: You
What: Midterm Exam
When: Wed Oct 18th 12:00pm - 1:20pm
Where: Scaife Hall 125
Why: https://youtu.be/xgMialPxS1c
MIDTERM

What to bring:
→ CMU ID
→ Calculator
→ One 8.5x11" page of notes (double-sided)

What not to bring:
→ Live animals
MIDTERM

Covers up to Query Optimization (inclusive).
→ Closed book, one sheet of notes (double-sided)
→ Please email Andy if you need special accommodations.

http://cmudb.io/f17-midterm
RELATIONAL MODEL

Integrity Constraints
Relation Algebra
Basic operations:
→ SELECT / INSERT / UPDATE / DELETE
→ WHERE predicates
→ Output control

More complex operations:
→ Joins
→ Aggregates
→ Common Table Expressions
STORAGE

Buffer Management Policies
→ LRU / MRU / CLOCK

On-Disk File Organization
→ Heaps
→ Linked Lists

Understand high-level trade-offs of different approaches.
HASHING

Extendible Hashing
→ Global Depth vs. Local Depth
→ Overflow Chains

Linear Hashing
→ Insertion / Splitting
→ Overflow Chains

Comparison with B+Trees
TREE INDEXES

B+Tree
→ Insertions / Deletions
→ Splits / Merges
→ Difference with B-Tree

Radix Trees
Skip Lists
SORTING

Two-way External Merge Sort
General External Merge Sort
Cost to sort different data sets with different number of buffers.
QUERY PROCESSING

Processing Models
→ Advantages / Disadvantages

Join Algorithms
→ Nested Loop
→ Sort-Merge
→ Hash

Query Optimization & Planning
NEXT CLASS

Parallel Query Execution