**ADMINISTRIVIA**

**Project #3** will be released this week.
It is due Sun Nov 17\textsuperscript{th} @ 11:59pm.

**Homework #3** will be released next week.
It is due Wed Nov 13\textsuperscript{th} @ 11:59pm.
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.

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
Nested Sub-queries
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?

It is too expensive to run every possible plan to determine this information, so the DBMS need a way to derive this information...
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**
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 $A$ given $N_R / V(A, R)$

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.

```
SELECT * FROM people
WHERE id = 123
```

What about more complex predicates? What is their selectivity?

```
SELECT * FROM people
WHERE val > 1000
```

```
SELECT * FROM people
WHERE age = 30
AND status = 'Lit'
```

CREATE TABLE people (
  id INT PRIMARY KEY,
  val INT NOT NULL,
  age INT NOT NULL,
  status VARCHAR(16)
);
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
Assume that $V(\text{age, people})$ has five distinct values (0–4) and $N_R = 5$

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

- $\text{sel}(A=\text{constant}) = SC(P) / N_R$
- Example: $\text{sel}(\text{age}=2) = 1/5$

```
SELECT * FROM people
WHERE age = 2
```
Range Predicate:

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

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

SELECT * FROM people
WHERE age >= 2
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*

```
SELECT * FROM people
WHERE age != 2
```
Conjunction:

\[ \text{sel}(P1 \land P2) = \text{sel}(P1) \cdot \text{sel}(P2) \]

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

This assumes that the predicates are independent.
Disjunction:

\[ \text{sel}(P_1 \lor P_2) = \text{sel}(P_1) + \text{sel}(P_2) - \text{sel}(P_1 \land P_2) = \text{sel}(P_1) + \text{sel}(P_2) - \text{sel}(P_1) \cdot \text{sel}(P_2) \]

\[ \text{sel}(\text{age}=2 \text{ OR name LIKE 'A%'}) \]

This again assumes that the selectivities are independent.
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:
→ # of Makes = 10, # of Models = 100

And the following query:
→ (make="Honda" AND model="Accord")

With the independence and uniformity assumptions, the selectivity is:
→ 1/10 × 1/100 = 0.001

But since only Honda makes Accords the real selectivity is 1/100 = 0.01

Source: Guy Lohman
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

Bucket Ranges

Bucket #1 Count=8
Bucket #2 Count=4
Bucket #3 Count=15
Bucket #4 Count=3
Bucket #5 Count=14
HISTOGRAMS WITH QUANTILES

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

Histogram (Quantiles)
Modern DBMSs also collect samples from tables to estimate selectivities.

Update samples when the underlying tables change significantly.

SEL\(\text{ECET} \ \text{AVG}(\text{age}) \ \text{FROM} \ \text{people} \ \text{WHERE} \ \text{age} > 50\)

**Table Sample**

<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>58</td>
<td>Rested</td>
</tr>
<tr>
<td>1003</td>
<td>Tupac</td>
<td>25</td>
<td>Dead</td>
</tr>
<tr>
<td>1005</td>
<td>Andy</td>
<td>38</td>
<td>Lit</td>
</tr>
</tbody>
</table>

\[\text{sel}(\text{age}>50) = \frac{1}{3}\]
Now that we can (roughly) estimate the selectivity of predicates, what can we actually do with them?
After performing rule-based rewriting, the DBMS will enumerate different plans for the query and estimate their costs.

- Single relation.
- Multiple relations.
- Nested sub-queries.

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.

```
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.

![Diagram showing different join trees and indicating that left-deep join trees are preferred.]
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...

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

Cost: 300
Cost: 400
Cost: 280
Cost: 300
Cost: 400
Cost: 380
Cost: 200
Cost: 450
Cost: 300
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
```
Step #1: Enumerate relation orderings

Prune plans with cross-products immediately!
CANDIDATE PLANS

Step #2: Enumerate join algorithm choices

Do this for the other plans.
CANDIDATE PLANS

Step #3: Enumerate access method choices

Do this for the other plans.
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 # of tables in query is **less** than 12 and switches to GEQO when there are 12 or more.
POSTGRES OPTIMIZER

1st Generation

- Cost: 300

2nd Generation

- Cost: 80

3rd Generation

- Cost: 90

---

Best: 80

1st Generation

- Cost: 300

2nd Generation

- Cost: 80

3rd Generation

- Cost: 90
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
NESTED SUB-QUERIES: REWRITE

```
SELECT name FROM sailors AS S
WHERE EXISTS (  
    SELECT * FROM reserves AS R  
    WHERE S.sid = R.sid  
    AND R.day = '2018-10-15'
  )

SELECT name  
FROM sailors AS S, reserves AS R  
WHERE S.sid = R.sid  
AND R.day = '2018-10-15'
```
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.

```
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
```
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.
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
CONCLUSION

Filter early as possible.

Selectivity estimations
→ Uniformity
→ Independence
→ Histograms
→ Join selectivity

Dynamic programming for join orderings

Rewrite nested queries

Again, query optimization is hard…
EXTRA CREDIT

Each student can earn extra credit if they write an encyclopedia article about a DBMS. → Can be academic/commercial, active/historical. Each article will use a standard taxonomy. → For each feature category, you select predefined options for your DBMS. → You will then need to provide a summary paragraph with citations for that category.
All the articles will be hosted on dbdb.io.
→ I will post registration details on Piazza.

I will post a sign-up sheet for you to pick what DBMS you want to write about.
→ If you choose a widely known DBMS, then the article will need to be comprehensive.
→ If you choose an obscure DBMS, then you will have to do the best you can to find information.
HOW TO DECIDE

Pick a DBMS based on whatever criteria you want:
→ Country of Origin
→ Popularity
→ Programming Language
→ Single-Node vs. Embedded vs. Distributed
→ Disk vs. Memory
→ Row Store vs. Column Store
→ Open-Source vs. Proprietary
This article must be your own writing with your own images. You may not copy text/images directly from papers or other sources that you find on the web.

Plagiarism will not be tolerated. See CMU's Policy on Academic Integrity for additional information.
Transactions!
→ aka the second hardest part about database systems