Project #2 is due tomorrow, Thu Oct 21st @ 11:59pm
→ Check your score again on Gradescope with formatting!

Project #3 will be released today.
It is due Sun Nov 14\(^{th}\) @ 11:59pm.

Homework #4 will be released next week.
It is due Sun Nov 7\(^{th}\) @ 11:59pm.
UPCOMING DATABASE TALK

An Overview of the Starburst Trino Query Optimizer
→ Monday Oct 25th @ 4:30pm ET
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.
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

Moe Cost Estimation (Statistics)
Plan Enumeration
COST MODEL COMPONENTS

Choice #1: Physical Costs
→ Predict CPU cycles, I/O, cache misses, RAM consumption, pre-fetching, etc…
→ Depends heavily on hardware.

Choice #2: Logical Costs
→ Estimate result sizes per operator.
→ Independent of the operator algorithm.
→ Need estimations for operator result sizes.

Choice #3: Algorithmic Costs
→ Complexity of the operator algorithm implementation.
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 $A$. 
The **selection cardinality** $\text{SC}(A, R)$ is the average number of records with a value for an attribute $A$ given $N_R / V(A, R)$.
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 formula assumes **data uniformity** where every value has the same frequency as all other values.

→ Example: 10,000 students, 10 colleges – how many students in SCS?
Equality predicates on unique keys are easy to estimate.
Equality predicates on unique keys are easy to estimate.

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

```
CREATE TABLE people (id INT PRIMARY KEY, val INT NOT NULL, age INT NOT NULL, status VARCHAR(16));
```
Equality predicates on unique keys are easy to estimate.

\[
\text{SELECT * FROM people WHERE id = 123}
\]

Computing the logical cost of complex predicates is more difficult...

\[
\text{SELECT * FROM people WHERE val > 1000}
\]

\[
\text{CREATE TABLE people (}
\text{    id INT PRIMARY KEY,}
\text{    val INT NOT NULL,}
\text{    age INT NOT NULL,}
\text{    status VARCHAR(16) }
\text{)};
\]

\[
\text{SELECT * FROM people WHERE age = 30}
\text{    AND status = 'Lit'}
\text{    AND age+id IN (1,2,3)}
\]
The **selectivity** \((\text{sel})\) of a predicate \(P\) is the fraction of tuples that qualify.

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

Formula depends on type of predicate:

\[\rightarrow \text{Equality} \]
\[\rightarrow \text{Range} \]
\[\rightarrow \text{Negation} \]
\[\rightarrow \text{Conjunction} \]
\[\rightarrow \text{Disjunction} \]
Assume that $V(\text{age}, \text{people})$ has five distinct values (0–4) and $N_R = 5$

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

$\rightarrow \text{sel}(A=\text{constant}) = SC(P) / N_R$

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

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

$\rightarrow$ $\text{sel}(A=\text{constant}) = \frac{\text{SC}(P)}{N_R}$

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

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

**Equality Predicate**: $A=$constant

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

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

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

$\rightarrow \text{sel}(A=\text{constant}) = \frac{SC(P)}{N_R}$

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

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

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

$\rightarrow \text{sel}(A=\text{constant}) = SC(P) / N_R$

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

**SELECTIONS – COMPLEX PREDICATES**

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

→ $\text{sel}(A\geq a) = \frac{(A_{\max} - a + 1)}{(A_{\max} - A_{\min} + 1)}$

→ Example: $\text{sel}(\text{age}\geq 2)$

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

\[ \text{sel}(A \geq a) = \frac{(A_{\text{max}} - a + 1)}{(A_{\text{max}} - A_{\text{min}} + 1)} \]

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

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

\[ \text{sel}(A \geq a) = \frac{(A_{\text{max}} - a + 1)}{(A_{\text{max}} - A_{\text{min}} + 1)} \]

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

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

→ \( \text{sel}(\text{A} \geq a) = \frac{\text{A}_{\text{max}} - a + 1}{\text{A}_{\text{max}} - \text{A}_{\text{min}} + 1} \)

→ Example: \( \text{sel(} \text{age} \geq 2) \approx \frac{4 - 2 + 1}{4 - 0 + 1} \approx \frac{3}{5} \)

SELECT * FROM people
WHERE \text{age} \geq 2
**SELECTIONS – COMPLEX PREDICATES**

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 != 2)

```
SELECT * FROM people
WHERE age != 2
```
Negation Query:

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

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

```
SELECT * FROM people
WHERE age \neq 2
```
SELECTIONS – COMPLEX PREDICATES

Negation Query:

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

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

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

Negation Query:

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

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

*Observation: Selectivity ≈ Probability*

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

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

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

This assumes that the predicates are independent.
Conjunction:
→ sel(P1 \( \land \) P2) = sel(P1) \( \cdot \) sel(P2)
→ sel(age=2 \( \land \) name LIKE 'A%')

This assumes that the predicates are independent.
Conjunction:

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

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

This assumes that the predicates are independent.
Disjunction:

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

This again assumes that the selectivities are independent.
**Disjunction:**

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

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?

Assume each key in the inner relation will exist in the outer table
RESULT SIZE ESTIMATION FOR JOINS

General case: \( R_{cols} \cap S_{cols} = \{A\} \) where \( A \) is not a primary 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:
→ \( \text{estSize} \approx N_R \cdot N_S / \max\{V(A,S), V(A,R)\} \)
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")
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
Our formulas are nice, but we assume that data values are uniformly distributed.

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

**Uniform Approximation**

<table>
<thead>
<tr>
<th>Distinct values of attribute</th>
<th># of occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
</tr>
</tbody>
</table>
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.

**Non-Uniform Approximation**

15 Keys × 32-bits = 60 bytes
EQUI-WIDTH HISTOGRAM

All buckets have the same width (i.e., the same number of values).

Non-Uniform Approximation

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
All buckets have the same width (i.e., the same number of values).

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
All buckets have the same width (i.e., the same number of values).

Equi-Width Histogram

Bucket Ranges

- Bucket #1: Count=8
- Bucket #2: Count=4
- Bucket #3: Count=15
- Bucket #4: Count=3
- Bucket #5: Count=14
Vary the width of buckets so that the total number of occurrences for each bucket is roughly the same.
EQUI-DEPTH HISTOGRAMS

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

Histogram (Quantiles)
EQUI-DEPTH HISTOGRAMS

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

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

**Histogram (Quantiles)**

- 1-5: 10 occurrences
- 6-8: 15 occurrences
- 9-13: 5 occurrences
- 14-15: 10 occurrences
SKETCHES

Probabilistic data structures that generate approximate statistics about a data set. Cost-model can replace histograms with sketches to improve its selectivity estimate accuracy.

Most common examples:
→ HyperLogLog (2007): Approximate the number of distinct elements in a set.
Modern DBMSs also collect samples from tables to estimate selectivities.

Update samples when the underlying tables changes significantly.

```
SELECT AVG(age) 
FROM people 
WHERE age > 50
```
Modern DBMSs also collect samples from tables to estimate selectivities.

Update samples when the underlying tables changes significantly.
Modern DBMSs also collect samples from tables to estimate selectivities.

Update samples when the underlying tables changes significantly.

**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>59</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>39</td>
<td>Shaved</td>
</tr>
</tbody>
</table>

```
SELECT AVG(age)
FROM people
WHERE age > 50
```
Modern DBMSs also collect samples from tables to estimate selectivities.

Update samples when the underlying tables changes significantly.

\[
\text{sel}(\text{age}>50) = \text{AVG(\text{age})}
\]

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>59</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>39</td>
<td>Shaved</td>
</tr>
<tr>
<td>1006</td>
<td>TigerKing</td>
<td>57</td>
<td>Jailed</td>
</tr>
</tbody>
</table>

\[
\text{SELECT AVG(\text{age}) FROM people WHERE age > 50}
\]
Modern DBMSs also collect samples from tables to estimate selectivities.

Update samples when the underlying tables changes significantly.

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

<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>59</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>39</td>
<td>Shaved</td>
</tr>
</tbody>
</table>

Table Sample

**SELECT AVG(age)**
**FROM people**
**WHERE age > 50**
Modern DBMSs also collect samples from tables to estimate selectivities.

Update samples when the underlying tables changes significantly.

**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>59</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>39</td>
<td>Shaved</td>
</tr>
</tbody>
</table>

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

```
SELECT AVG(age) 
FROM people 
WHERE age > 50
```
Now that we can (roughly) estimate the selectivity of predicates, and subsequently the cost of query plans, what can we 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…
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.
Query planning for OLTP queries is easy because they are **sargable** (Search **Arg**ument **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.
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;
```
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.
Fundamental decision in **System R**: Only consider left-deep join trees.
Fundamental decision in System R: Only consider left-deep join trees.
Fundamental decision in **System R** is to 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.
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
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

Cost: 300

Cost: 400

Cost: 280

Cost: 200
Hash Join
R.a = S.a

Cost: 300

R \Join S
T

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

Cost: 200

Hash Join
T.b = S.b

R \Join S
\Join T

\ldots
DYNAMIC PROGRAMMING

SELECT * FROM R, S, T
WHERE R.a = S.a
AND S.b = T.b
Hash Join
R.a = S.a  Cost: 300

\[ \text{R } \bowtie \text{ S} \]

Hash Join
S.b = T.b  Cost: 380

\[ \text{R } \bowtie \text{ S} \bowtie \text{T} \]

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

\[ \text{R } \bowtie \text{ S} \bowtie \text{T} \]

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

Cost: 300

Cost: 200

Cost: 380
Dynamic Programming

\[
\begin{align*}
R \bowtie S &= T \\
\text{Hash Join} \\
T \bowtie S &= R \\
\text{SortMerge Join} \\
S.a &= R.a \\
\text{Cost: } 300 \\
R \bowtie S \bowtie T
\end{align*}
\]

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

Cost: 200

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
Step #1: Enumerate relation orderings

- Prune plans with cross-products immediately!
Step #1: Enumerate relation orderings

- Prune plans with cross-products immediately!
Step #1: Enumerate relation orderings

Prune plans with cross-products immediately!
CANDIDATE PLANS

Step #2: Enumerate join algorithm choices
Step #2: Enumerate join algorithm choices
Step #2: Enumerate join algorithm choices

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

Do this for the other plans.
CANDIDATE PLANS

Step #3: Enumerate access method choices
CANDIDATE PLANS

Step #3: Enumerate access method choices

Diagram showing different access method choices for tables HJ, R, S, T, including SeqScan and IndexScan(S,b).
Step #3: Enumerate access method choices

Do this for the other plans.
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 GENETIC OPTIMIZER

1st Generation

- NL -> NL
- NL -> T
- R -> S
- T -> R

- NL -> HJ
- HJ -> S
- TR -> T

- HJ -> HJ
- HJ -> T
- S -> R
POSTGRES GENETIC OPTIMIZER

1st Generation

- **Cost:** 300
- **Cost:** 200
- **Cost:** 100
1st Generation

- **Cost: 300**

- **Cost: 200**

- **Cost: 100**

**Best: 100**
POSTGRES GENETIC OPTIMIZER

1st Generation

Cost: 300

Cost: 200

Cost: 100

Best: 100
POSTGRES GENETIC OPTIMIZER

1st Generation

Cost: 300

Cost: 200

Cost: 100

Best: 100
**POSTGRES GENETIC OPTIMIZER**

1st Generation

- Cost: 300
- Cost: 200
- Cost: 100

2nd Generation

- Cost:
- Best: 100
POSTGRES GENETIC OPTIMIZER

1st Generation

- 1st Generation
  - Cost: 300
- 2nd Generation
  - Cost: 200
- 3rd Generation
  - Cost: 100

2nd Generation

- Cost: 80
- Cost: 200
- Cost: 110

Best: 100
POSTGRES GENETIC OPTIMIZER

1st Generation

- Cost: 300
- Cost: 200
- Cost: 100

2nd Generation

- Cost: 80
- Cost: 200
- Cost: 110

Best: 80
POSTGRES GENETIC OPTIMIZER

1st Generation

- Cost: 300
- Cost: 200
- Cost: 100

2nd Generation

- Cost: 80
- Cost: 200
- Cost: 110

Best: 80
POSTGRES GENETIC OPTIMIZER

1st Generation

Cost: 300

Cost: 200

Cost: 100

2nd Generation

Cost: 80

Cost: 200

Cost: 110

3rd Generation

Cost: 90

Cost: 160

Cost: 120

Best: 80
CONCLUSION

Filter early as possible.

Selectivity estimations
→ Uniformity
→ Independence
→ Inclusion
→ Histograms
→ Join selectivity

Dynamic programming for join orderings
Again, query optimization is hard...
Transactions!
→ aka the second hardest part about database systems