# Lecture 20: Cost-Based Query Optimization

CREATING THE NEXT®

# Today's Agenda



- S4 + L4
- 1 week

# Recap

# Query Optimization

$Q \to Q' \to Q''$

- **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.

Apache Calcite

# 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

Complex

"QO + QE"

Width

# 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?

```
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
```

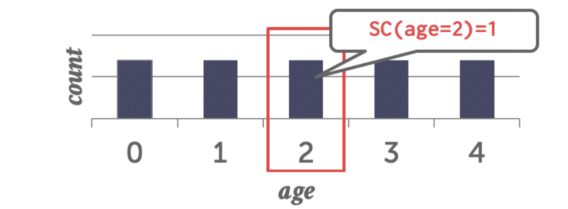
# Complex Predicates

- 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 $V$(age,people) has five distinct values (0–4) and $N_R = 5$
- **Equality Predicate:** A=constant
  - *sel*(A=constant) = $SC(P)$ / $N_R$
  - Example: sel(age=2) = 1/5

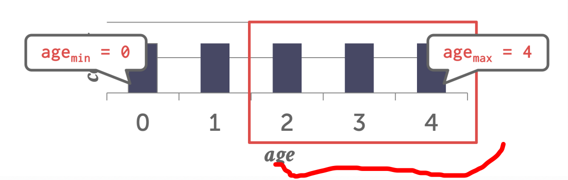
```
SELECT * FROM people  WHERE age = 2
```



# Selection – Complex Predicates

- **Range Predicate:**
  - sel(A>=a) = $(A_{max}$ – a) / $(A_{max} – A_{min})$
  - Example: sel(age>=2) ≈ (4 – 2) / (4 – 0) ≈ 1/2

```
SELECT * FROM people  WHERE age >= 2
```

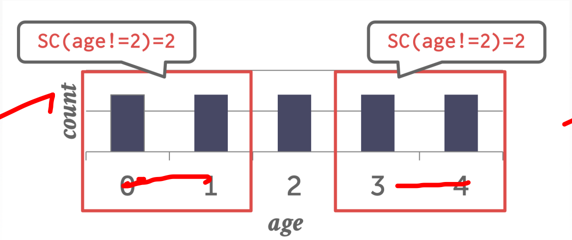


# Selection – Complex Predicates

- **Negation Query:**
  - sel(not P) = 1 – sel(P)
  - Example: sel(age != 2) = 1 – (1/5) = 4/5
- **Observation:** Selectivity ≈ Probability

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

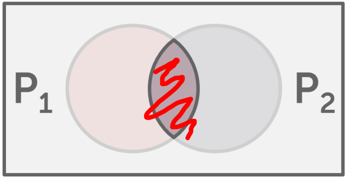
2 x 2
/ 5



# Selection – Complex Predicates

- **Conjunction**:
  - sel(P1 $\wedge$ P2) = sel(P1) $\times$ sel(P2)
  - sel(age=2 $\wedge$ name LIKE 'A%')
- This assumes that the predicates are **independent**.
- Not always true in practice!

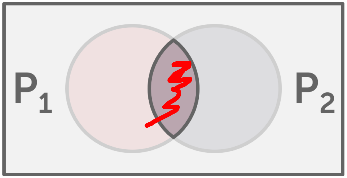
```
SELECT * FROM people  WHERE age = 2 AND name LIKE 'A%'
```



# Selection – Complex Predicates

- **Disjunction:**
  - $sel(P1 \vee P2) = sel(P1) + sel(P2) - sel(P1 \wedge P2) = sel(P1) + sel(P2) - sel(P1) \times sel(P2)$
  - sel(age=2 OR name LIKE 'A%')
- This again assumes that the selectivities are independent.

```
SELECT * FROM people  WHERE age = 2 OR name LIKE 'A%'
```



# Selection Cardinality

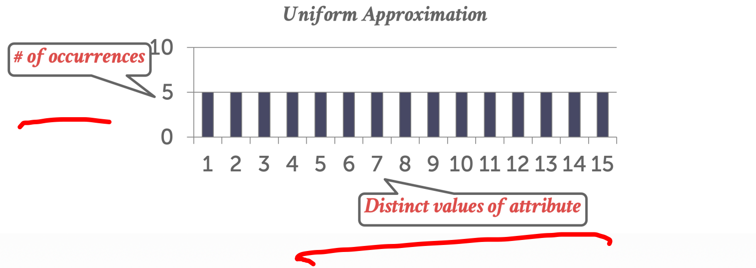
- **Assumption 1: Uniform Data**
  - The distribution of values (except for the heavy hitters) is the same.
- **Assumption 2: Independent Predicates**
  - The predicates on attributes are independent
- **Assumption 3: Inclusion Principle**
  - The domain of join keys overlap such that each key in the 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: $(make = "Honda" AND model = "Accord")$
- 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 $1/100 = 0.01$

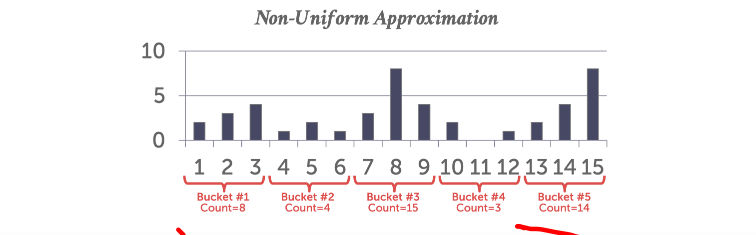
# Cost Estimation

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



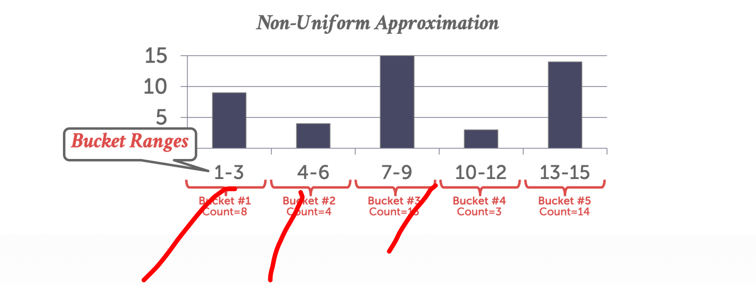
# Cost Estimation

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



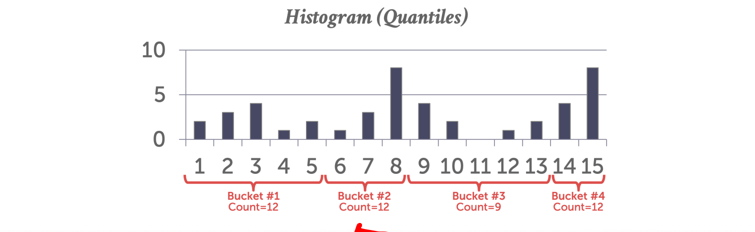
# Cost Estimation

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



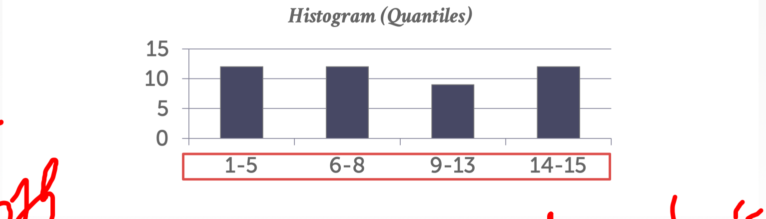
# Histograms With Quantiles

- 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.




equi-depth vs equi-width histogram

# Sampling

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

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

| id | name | age | status |
|---|---|---|---|
| 1001 | Shiyi | 58 | Senior |
| 1002 | Rahul | 41 | Sophomore |
| 1003 | Peter | 25 | Freshman |
| 1004 | Mark | 25 | Junior |
| 1005 | Alice | 38 | Senior |

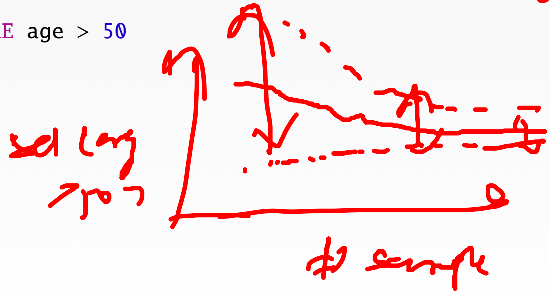
# Sampling

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

AQP

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

| id | name | age | status |
|---|---|---|---|
| 1001 | Shiyi | 58 | Senior |
| 1003 | Mark | 25 | Junior |
| 1005 | Alice | 38 | Senior |



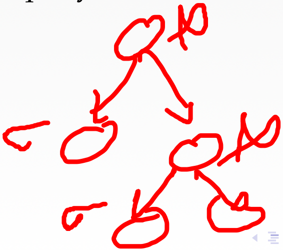
# Observation

- 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 **<u>some timeout</u>**.



# 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. . .

$P_1$ $P_2$ $P_3$

Sel. 0.7 0.2 0.005

# 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;
```

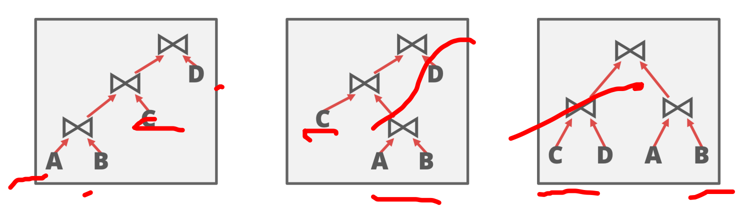
# Multi-Relation Query Planning

- 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.

Georgia Tech

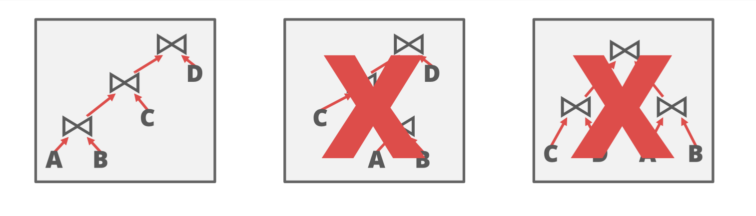
# Multi-Relation Query Planning

- 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**.




Pat Selinger

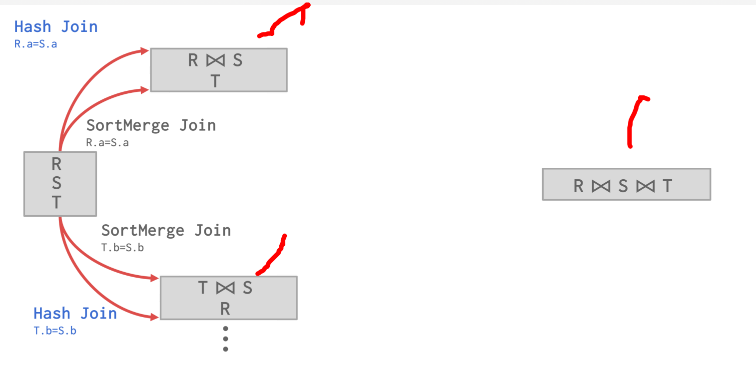
# Multi-Relation Query Planning

- 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.

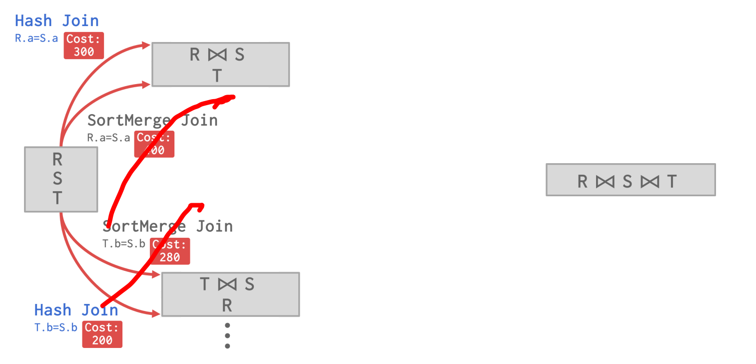
# Multi-Relation Query Planning

- 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.

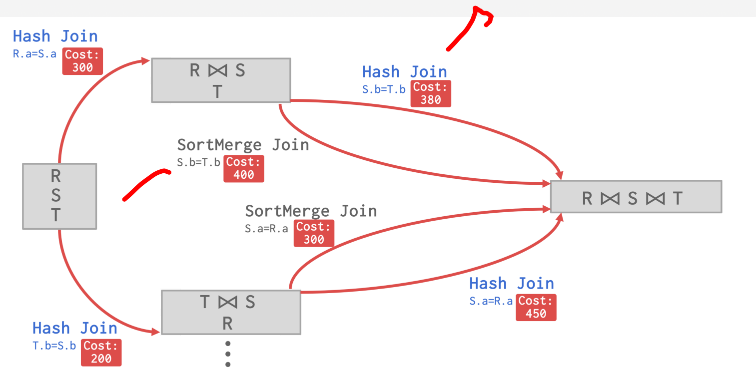
# Dynamic Programming



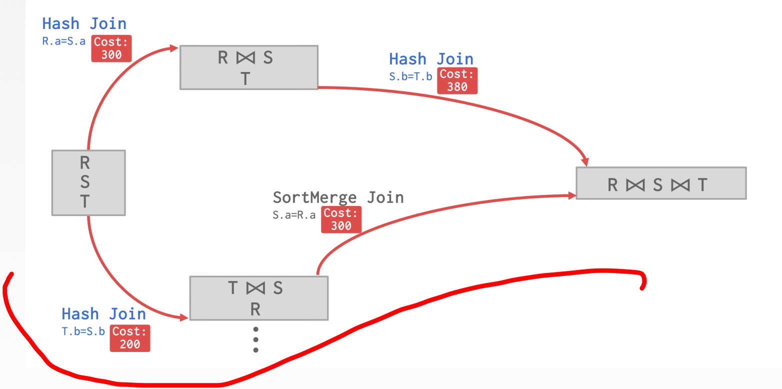
# Dynamic Programming



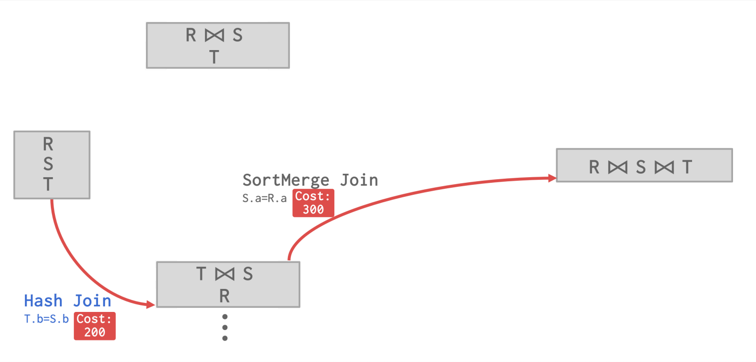
# Dynamic Programming



# Dynamic Programming



# Dynamic Programming



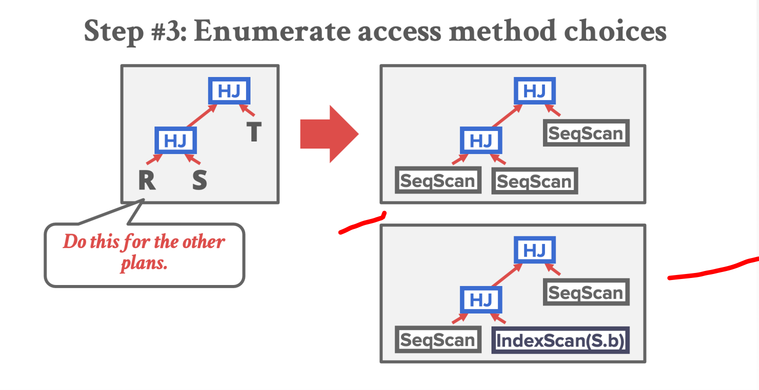
# Candidate Plan Example

- 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
```

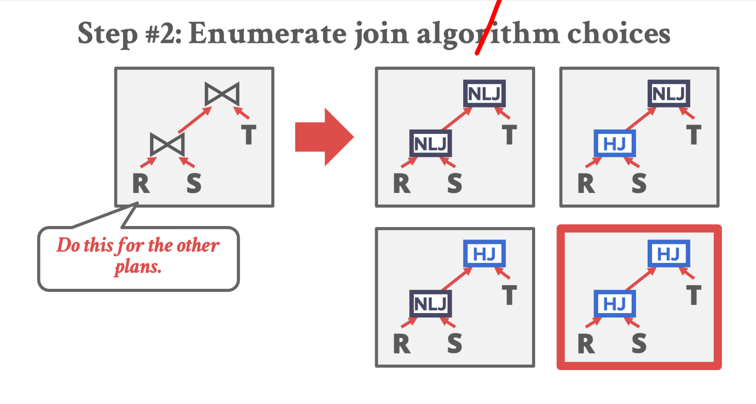
# Candidate Plans

- Step 1: Enumerate relation orderings



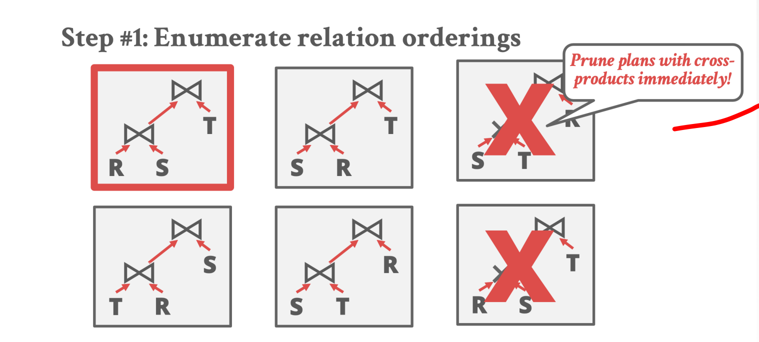
# Candidate Plans

- Step 2: Enumerate join algorithm choices



# Candidate Plans

- Step 3: Enumerate access method choices

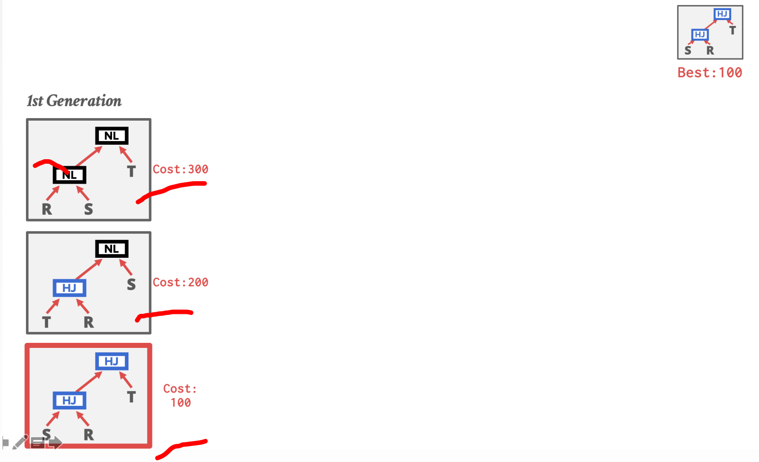


# Postgres Optimizer

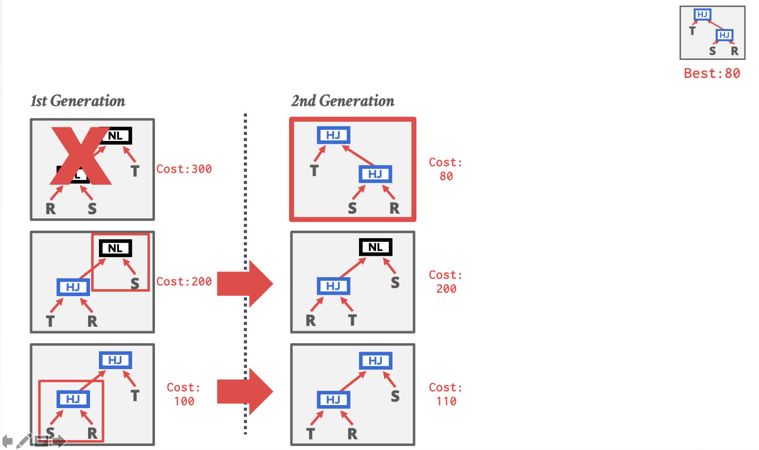
- Examines all types of join trees
  - Left-deep, Right-deep, bushy
- Two **optimizer implementations**:
  - Traditional Dynamic Programming Approach
  - Genetic Query Optimizer (GEQO)
- Postgres uses the traditional algorithm when **number of tables** in query is less than 12 and switches to GEQO when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

Georgia Tech

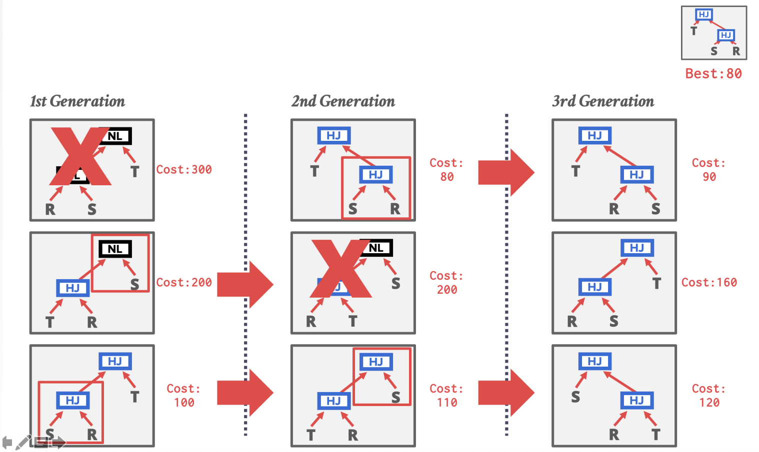
# Postgres Optimizer



# Postgres Optimizer



# Postgres Optimizer



# 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