

University of Michigan

21 Query Planning – Part II



Database Management Systems
EECS 484
Fall 2024



Lin Ma
Computer Science and
Engineering Division

QUERY OPTIMIZATION

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.



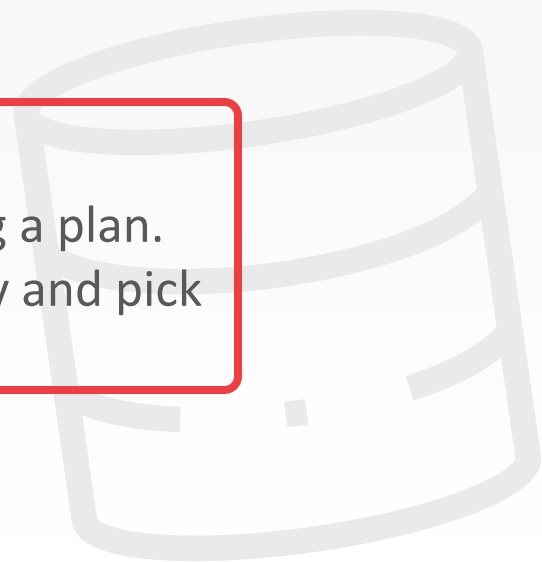
QUERY OPTIMIZATION

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



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

$$A: N_R / V(A, R)$$



DERIVABLE STATISTICS

The selection cardinality $SC(A, R)$ is the average number of records with a value for an attribute

$$A: N_R / V(A, R)$$

Note that this formula assumes *data uniformity* where every value has the same frequency as all other values.

→ Example: 10,000 students, 10 departments – how many students in EECS?



LOGICAL COSTS

Estimate the result size of every operator in the query plan

Use the estimated result size from the child operator as the input size for the parent's cost estimation



RESULT SIZE ESTIMATION

Equality predicates on unique keys are easy to estimate.



RESULT SIZE ESTIMATION

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)  
);
```

RESULT SIZE ESTIMATION

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)  
);
```

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

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

```
SELECT * FROM people  
WHERE age = 30  
  AND status = 'yes'  
  AND age+id IN (1,2,3)
```

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



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



SELECTIONS – COMPLEX PREDICATES

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

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

→ $\text{sel}(A = \text{constant}) = SC(A, R) / N_R$

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



SELECTIONS – COMPLEX PREDICATES

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

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

→ $\text{sel}(A = \text{constant}) = SC(A, R) / N_R$

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

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



SELECTIONS – COMPLEX PREDICATES

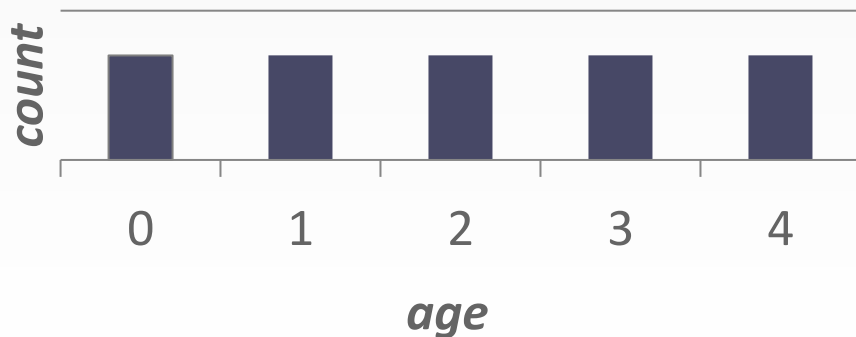
Assume that $V(\text{age}, \text{people})$ has five distinct values (0–4) and $N_R = 5$

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

→ $\text{sel}(A = \text{constant}) = SC(A, R) / N_R$

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

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



SELECTIONS – COMPLEX PREDICATES

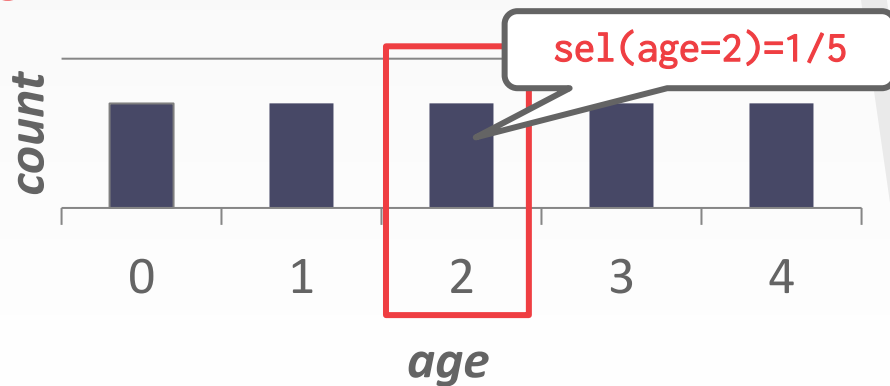
Assume that $V(\text{age}, \text{people})$ has five distinct values (0–4) and $N_R = 5$

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

→ $\text{sel}(A = \text{constant}) = SC(A, R) / N_R$

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

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



SELECTIONS – COMPLEX PREDICATES

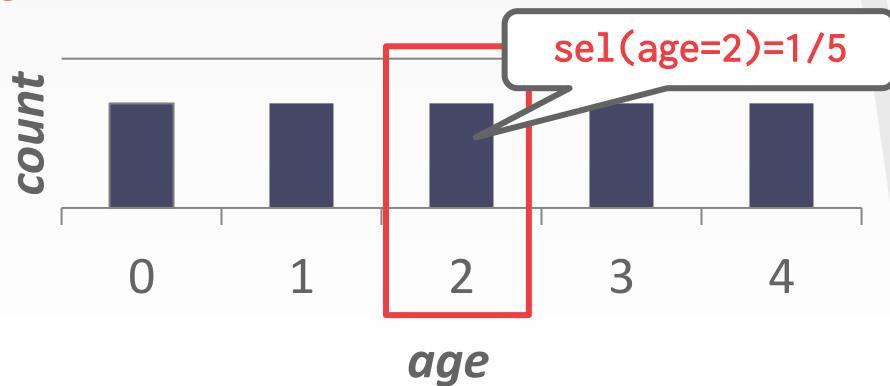
Assume that $V(\text{age}, \text{people})$ has five distinct values (0–4) and $N_R = 5$

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

→ $\text{sel}(A = \text{constant}) = SC(A, R) / N_R$

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

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



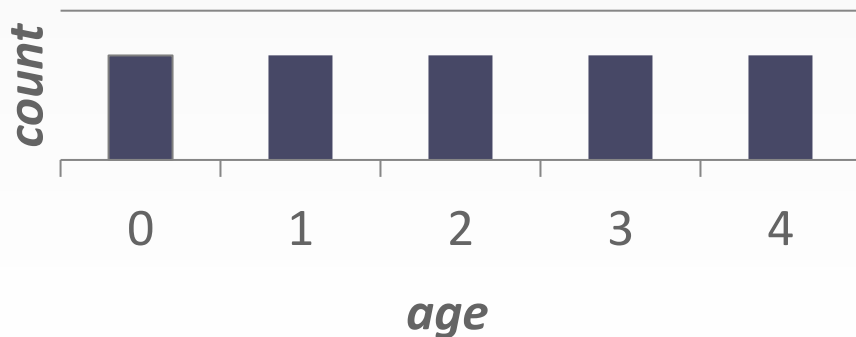
SELECTIONS – COMPLEX PREDICATES

Range Predicate:

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

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

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



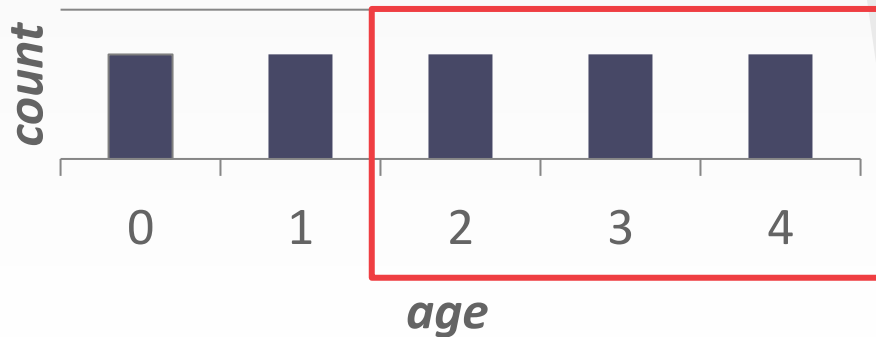
SELECTIONS – COMPLEX PREDICATES

Range Predicate:

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

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

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



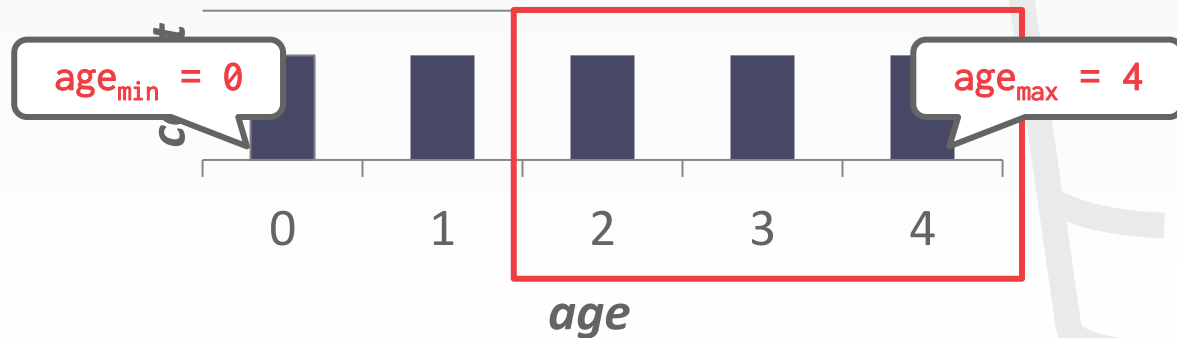
SELECTIONS – COMPLEX PREDICATES

Range Predicate:

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

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

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



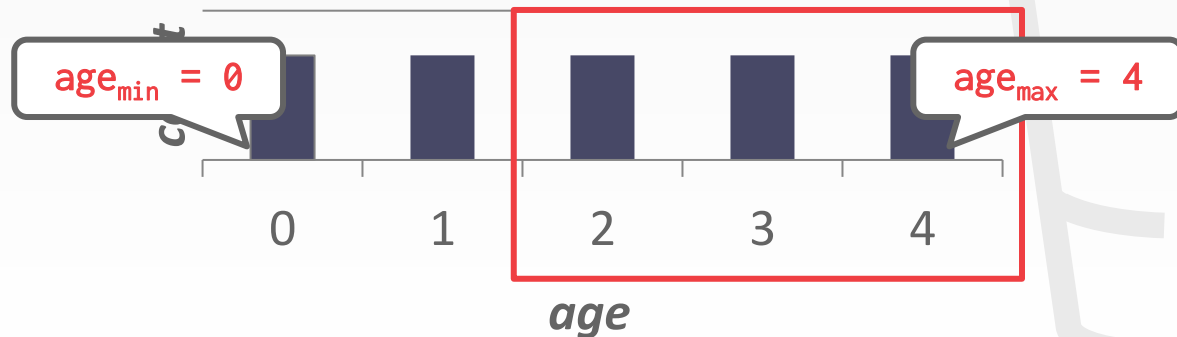
SELECTIONS – COMPLEX PREDICATES

Range Predicate:

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

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

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



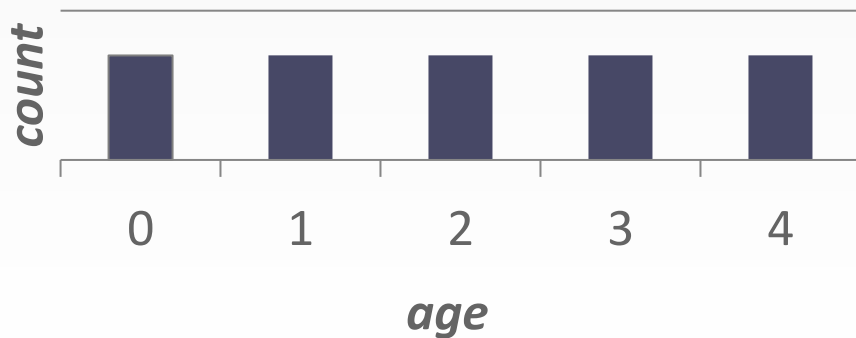
SELECTIONS – COMPLEX PREDICATES

Negation Query:

→ $\text{sel}(\text{not } P) = 1 - \text{sel}(P)$

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

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



SELECTIONS – COMPLEX PREDICATES

Negation Query:

→ $\text{sel}(\text{not } P) = 1 - \text{sel}(P)$

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

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



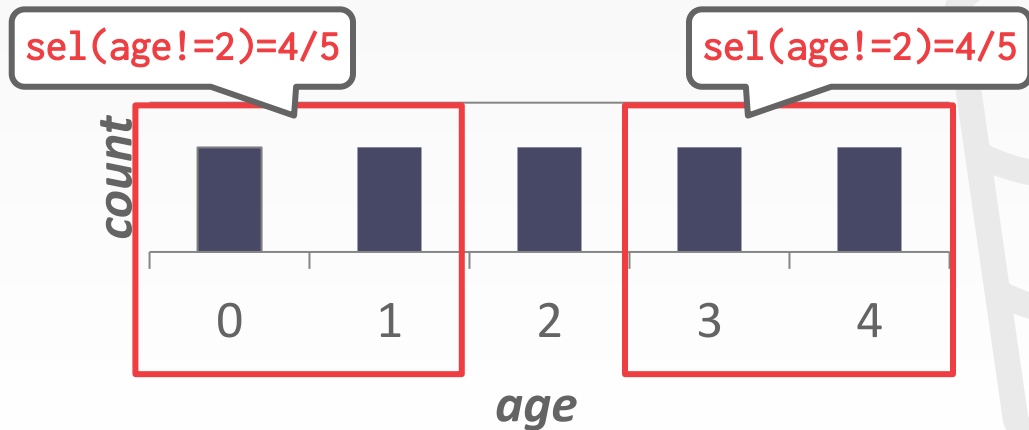
SELECTIONS – COMPLEX PREDICATES

Negation Query:

→ $\text{sel}(\text{not } P) = 1 - \text{sel}(P)$

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

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



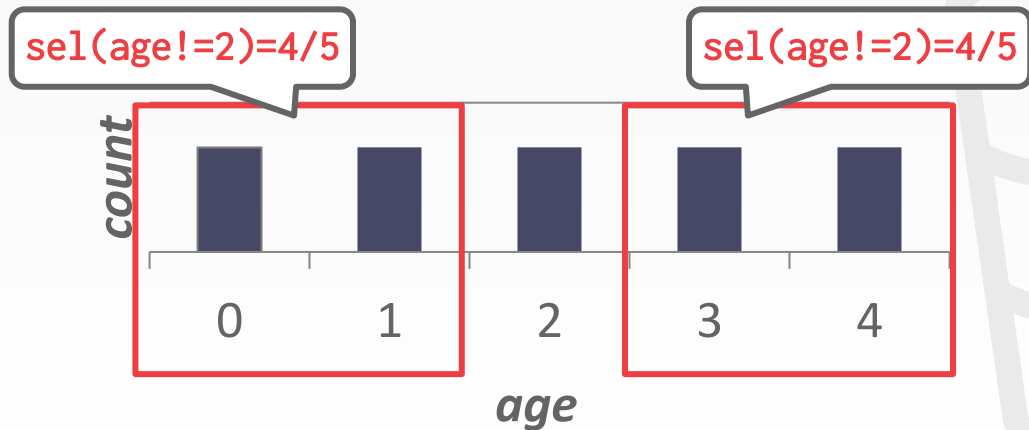
SELECTIONS – COMPLEX PREDICATES

Negation Query:

→ $\text{sel}(\text{not } P) = 1 - \text{sel}(P)$

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

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



SELECTIONS – COMPLEX PREDICATES

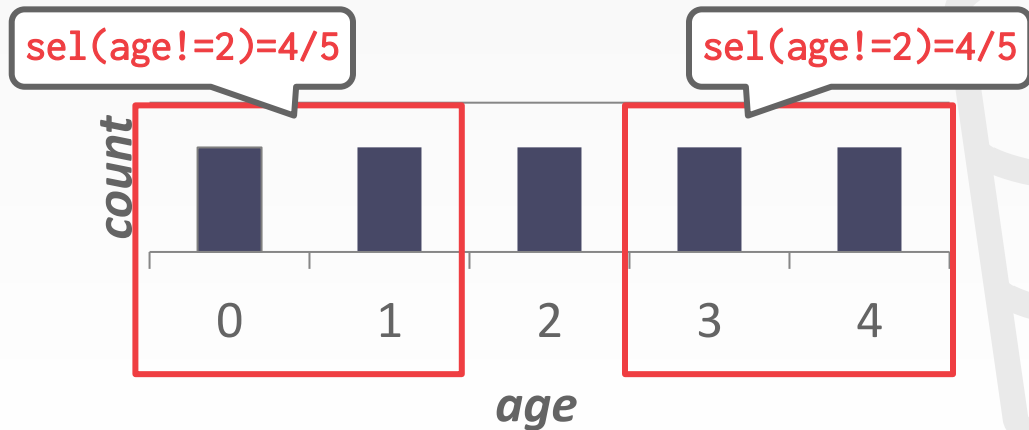
Negation Query:

→ $\text{sel}(\text{not } P) = 1 - \text{sel}(P)$

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

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

Observation: Selectivity \approx Probability



SELECTIONS – COMPLEX PREDICATES

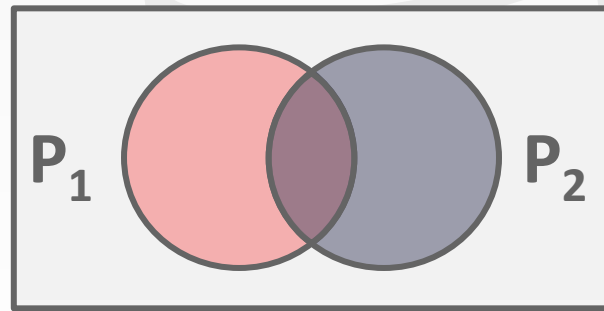
Conjunction:

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

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

This assumes that the predicates are independent.

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



SELECTIONS – COMPLEX PREDICATES

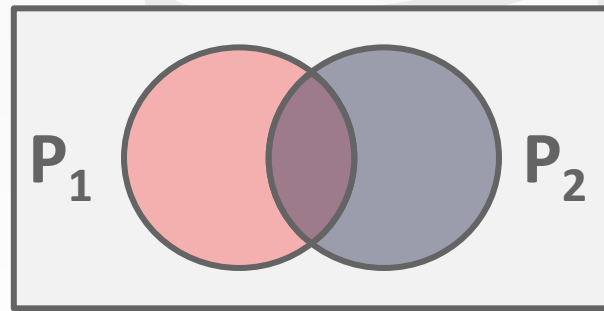
Conjunction:

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

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

This assumes that the predicates are independent.

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



SELECTIONS – COMPLEX PREDICATES

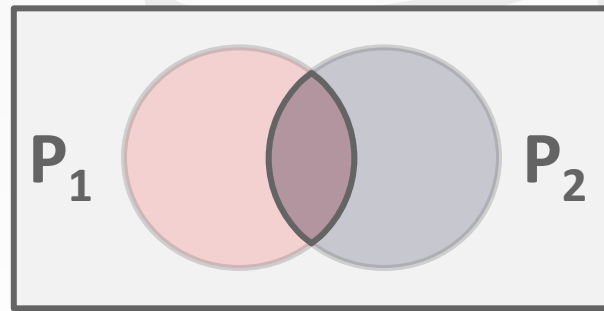
Conjunction:

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

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

This assumes that the predicates are independent.

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



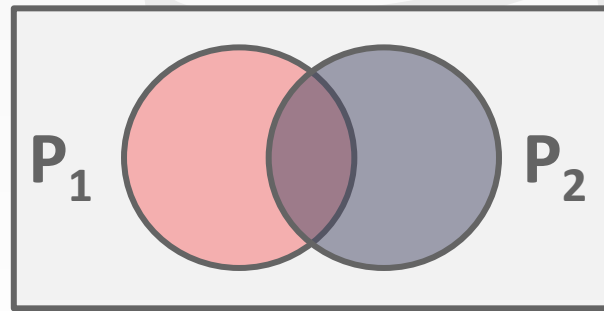
SELECTIONS – COMPLEX PREDICATES

Disjunction:

→ $\text{sel}(P1 \vee P2)$
= $\text{sel}(P1) + \text{sel}(P2) - \text{sel}(P1 \wedge P2)$
= $\text{sel}(P1) + \text{sel}(P2) - \text{sel}(P1) \cdot \text{sel}(P2)$
→ $\text{sel}(\text{age}=2 \text{ OR name LIKE 'A\%'})$

This again assumes that the selectivities are independent.

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



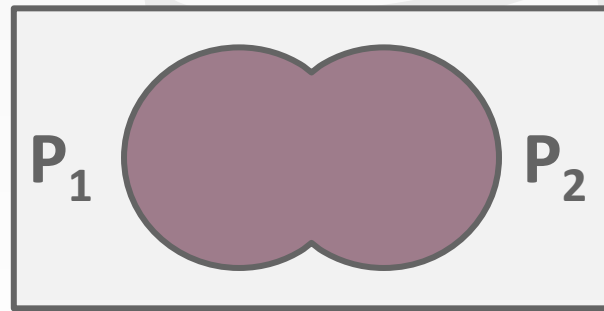
SELECTIONS – COMPLEX PREDICATES

Disjunction:

→ $\text{sel}(P1 \vee P2)$
= $\text{sel}(P1) + \text{sel}(P2) - \text{sel}(P1 \wedge P2)$
= $\text{sel}(P1) + \text{sel}(P2) - \text{sel}(P1) \cdot \text{sel}(P2)$
→ $\text{sel}(\text{age}=2 \text{ OR name LIKE 'A\%'})$

This again assumes that the selectivities are independent.

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

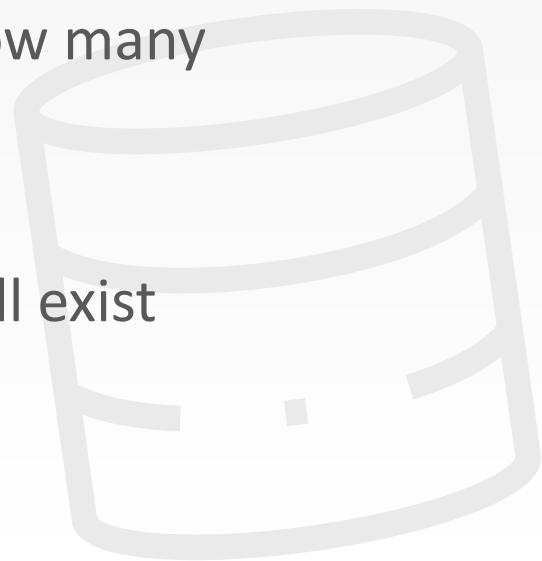


RESULT SIZE ESTIMATION FOR JOINS

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

→ Match each R -tuple with S -tuples:

$$\text{estSize} \approx N_R \cdot SC(A, S) = N_R \cdot N_S / V(A, S)$$

→ Symmetrically, for S :

$$\text{estSize} \approx N_S \cdot SC(A, R) = N_R \cdot N_S / V(A, R)$$

Overall:

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



SELECTIVITY ESTIMATION

Assumption #1: Uniform Data

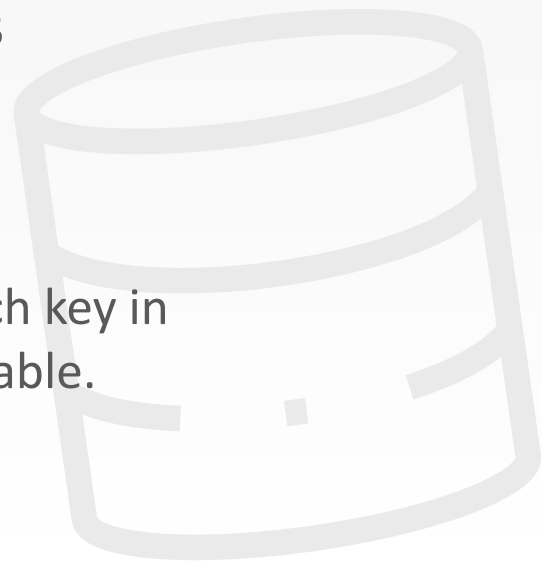
→ The distribution of values is the same.

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")



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:

→ $1/10 \times 1/100 = 0.001$

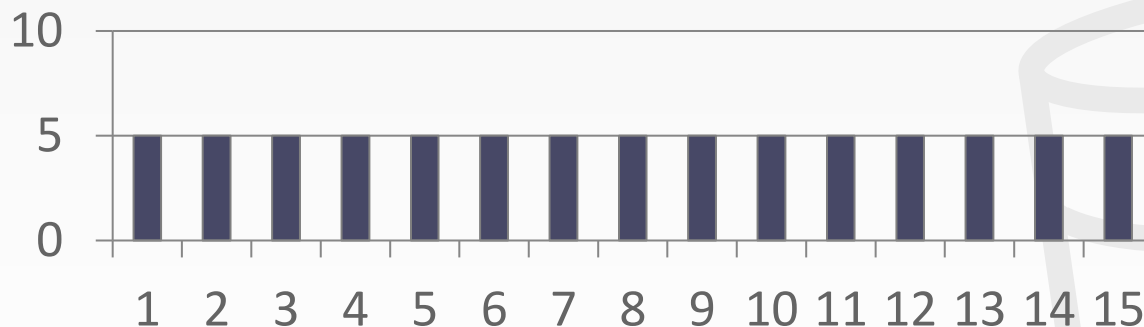
But since only Honda makes Accords the real selectivity is $1/100 = 0.01$



COST ESTIMATIONS

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

Uniform Approximation



COST ESTIMATIONS

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

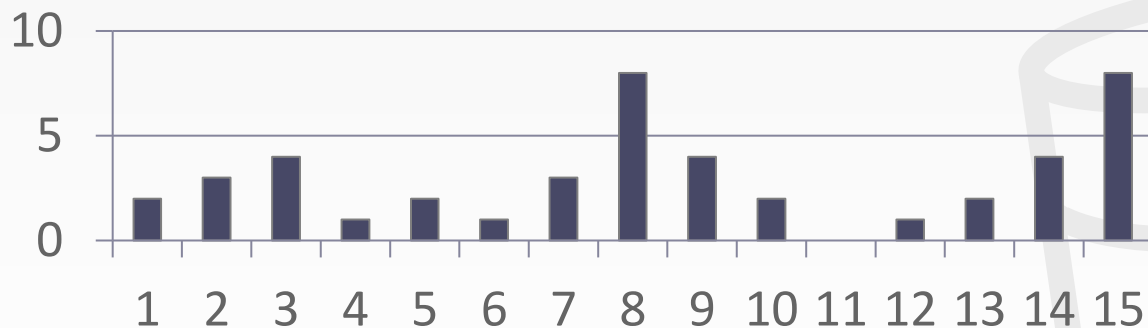
Uniform Approximation



COST ESTIMATIONS

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

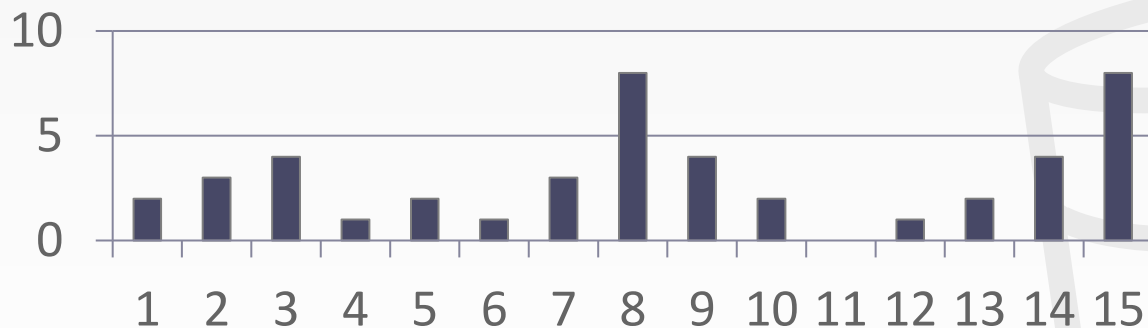
Non-Uniform Approximation



COST ESTIMATIONS

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

Non-Uniform Approximation

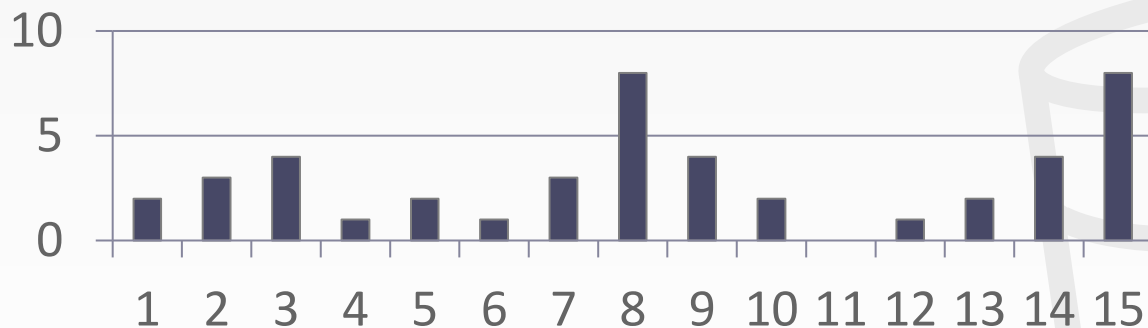


15 Keys \times 32-bits = 60 bytes

EQUI-WIDTH HISTOGRAM

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

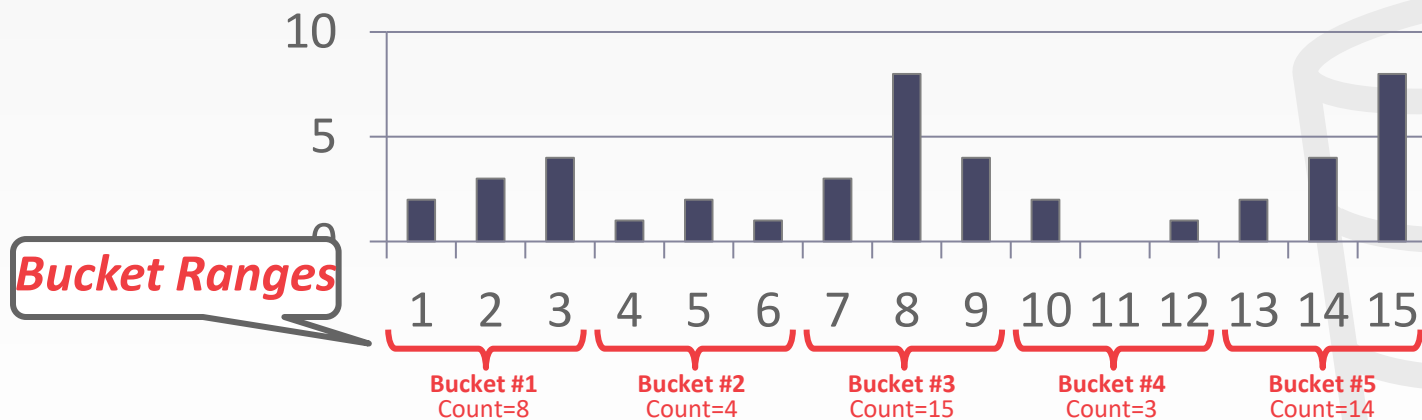
Non-Uniform Approximation



EQUI-WIDTH HISTOGRAM

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

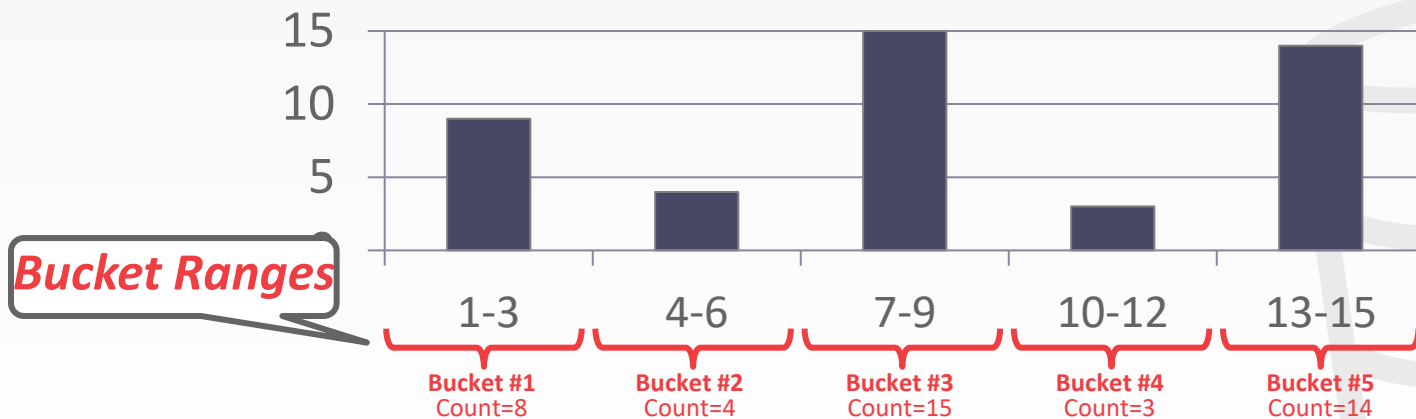
Non-Uniform Approximation



EQUI-WIDTH HISTOGRAM

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

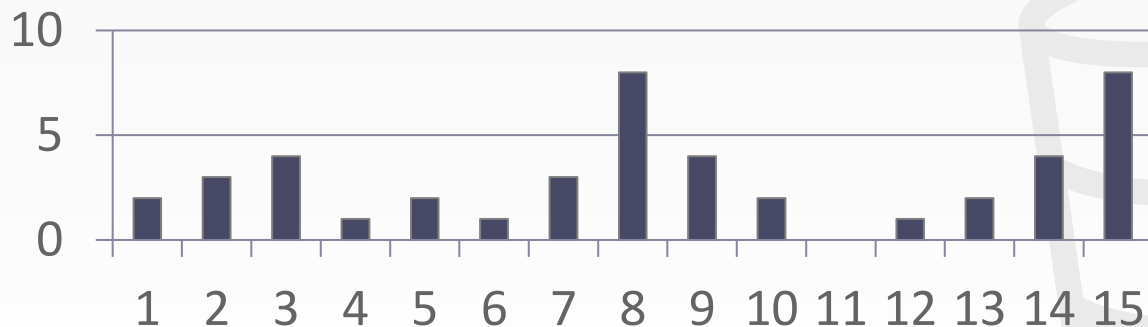
Equi-Width Histogram



EQUI-DEPTH HISTOGRAMS

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

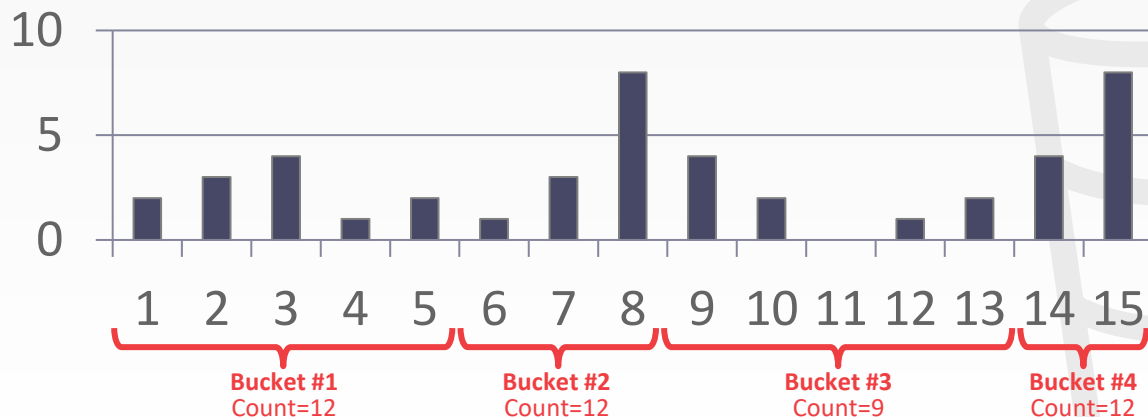
Histogram (Quantiles)



EQUI-DEPTH HISTOGRAMS

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

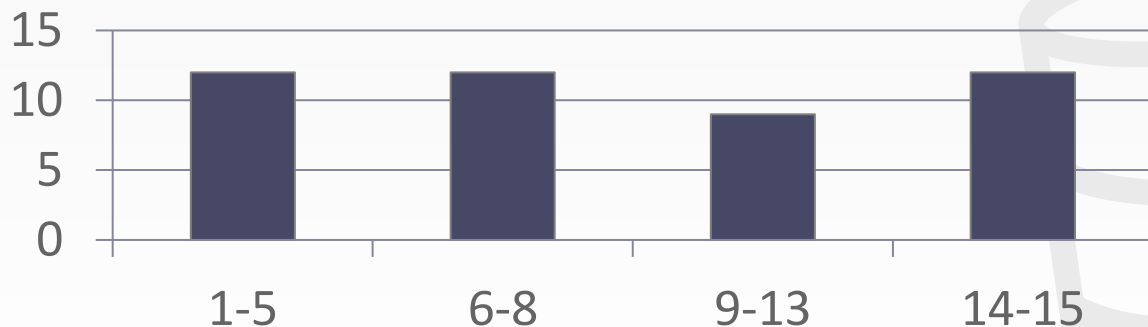
Histogram (Quantiles)



EQUI-DEPTH HISTOGRAMS

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

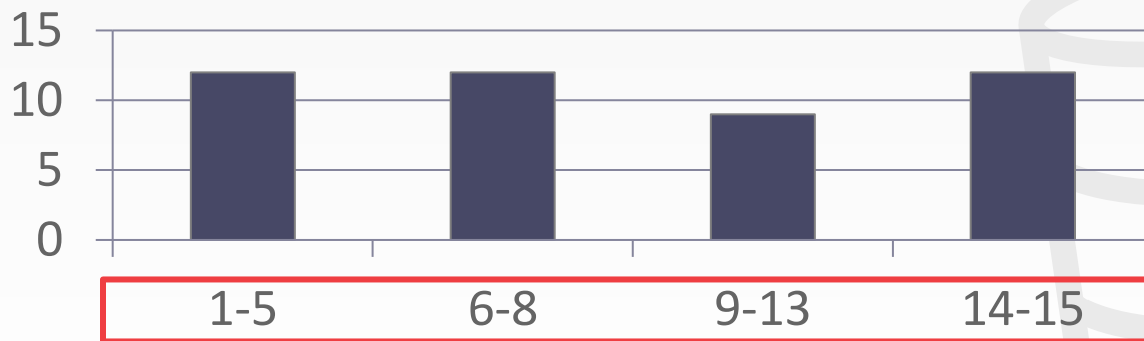
Histogram (Quantiles)



EQUI-DEPTH HISTOGRAMS

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

Histogram (Quantiles)



SAMPLING

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
```

id	name	age	status
1001	Andy	59	yes
1002	Bob	41	no
1003	Cathy	25	yes
1004	Dave	26	no
1005	Eve	39	yes
1006	Frank	57	yes

⋮


1 billion tuples

SAMPLING

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
```



id	name	age	status
1001	Andy	59	yes
1002	Bob	41	no
1003	Cathy	25	yes
1004	Dave	26	no
1005	Eve	39	yes
1006	Frank	57	yes

⋮

1 billion tuples

SAMPLING

Modern DBMSs also collect samples from tables to estimate selectivities.

Update samples when the underlying tables changes significantly.

Table Sample

1001	Andy	59	yes
1003	Cathy	25	yes
1005	Eve	39	yes

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



id	name	age	status
1001	Andy	59	yes
1002	Bob	41	no
1003	Cathy	25	yes
1004	Dave	26	no
1005	Eve	39	yes
1006	Frank	57	yes

⋮

1 billion tuples

SAMPLING

Modern DBMSs also collect samples from tables to estimate selectivities.


Update samples when the underlying tables changes significantly.

Table Sample

1001	Andy	59	yes
1003	Cathy	25	yes
1005	Eve	39	yes

sel(age>50) =

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



id	name	age	status
1001	Andy	59	yes
1002	Bob	41	no
1003	Cathy	25	yes
1004	Dave	26	no
1005	Eve	39	yes
1006	Frank	57	yes

⋮

1 billion tuples

SAMPLING

Modern DBMSs also collect samples from tables to estimate selectivities.


Update samples when the underlying tables changes significantly.

$\text{sel}(\text{age} > 50) =$

Table Sample

1001	Andy	59	yes
1003	Cathy	25	yes
1005	Eve	39	yes

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



id	name	age	status
1001	Andy	59	yes
1002	Bob	41	no
1003	Cathy	25	yes
1004	Dave	26	no
1005	Eve	39	yes
1006	Frank	57	yes

⋮
1 billion tuples

SAMPLING

Modern DBMSs also collect samples from tables to estimate selectivities.


Update samples when the underlying tables changes significantly.

Table Sample

1001	Andy	59	yes
1003	Cathy	25	yes
1005	Eve	39	yes

$$\text{sel}(\text{age} > 50) = 1/3$$

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



id	name	age	status
1001	Andy	59	yes
1002	Bob	41	no
1003	Cathy	25	yes
1004	Dave	26	no
1005	Eve	39	yes
1006	Frank	57	yes

⋮
1 billion tuples

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:

- Count-Min Sketch (1988): Approximate frequency count of elements in a set.
- HyperLogLog (2007): Approximate the number of distinct elements in a set.



OBSERVATION

Now that we can (roughly) estimate the selectivity of predicates, and subsequently the cost of query plans, what can we do with them?



QUERY OPTIMIZATION

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...



MULTI-RELATION QUERY PLANNING

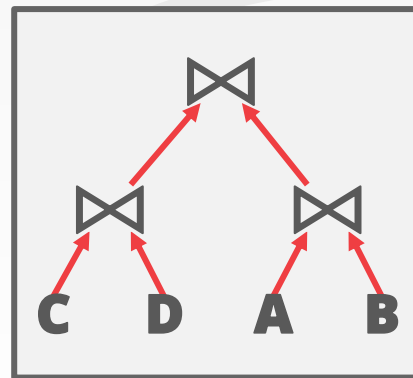
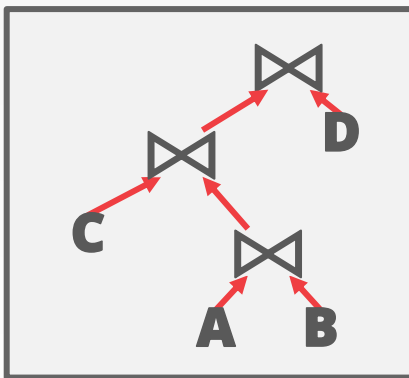
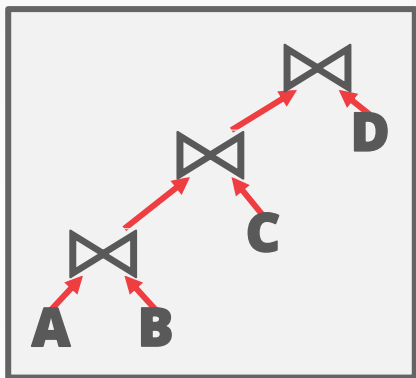
As number of joins increases, number of alternative plans grows rapidly
→ We need to restrict search space.



MULTI-RELATION QUERY PLANNING

Fundamental decision in **System R**: Only consider left-deep join trees.

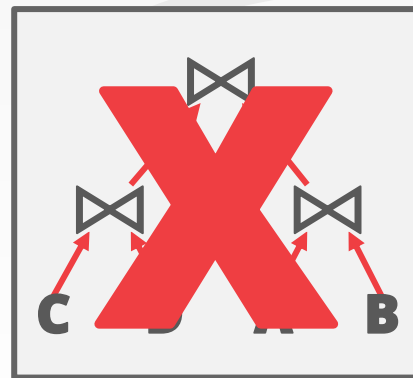
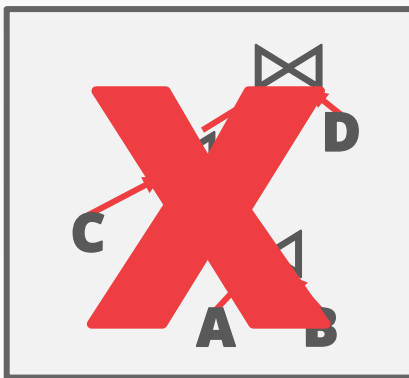
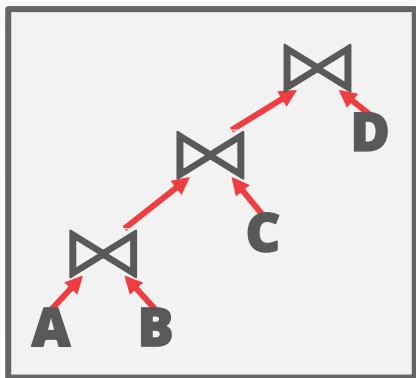
→ Many modern DBMSs still make this assumption.



MULTI-RELATION QUERY PLANNING

Fundamental decision in **System R**: Only consider left-deep join trees.

→ Many modern DBMSs still make this assumption.



BRUTE-FORCE SEARCH

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...

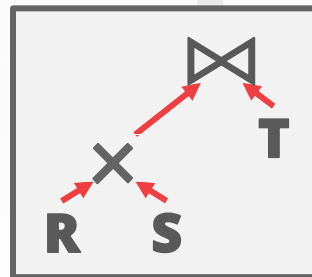
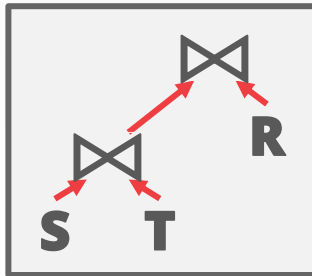
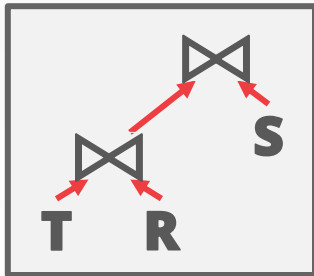
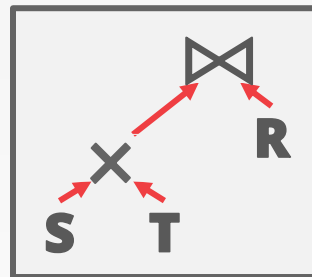
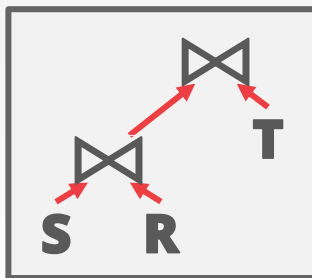
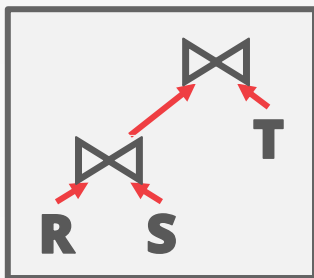
Estimate the cost of every possible plan and
return the best



BRUTE-FORCE

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

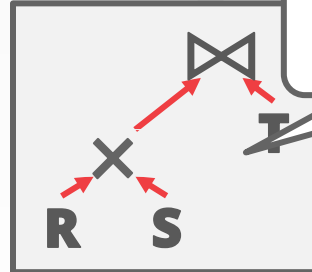
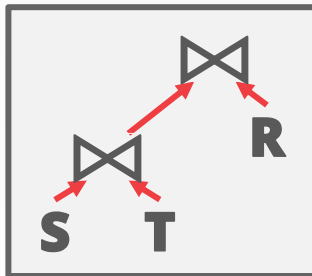
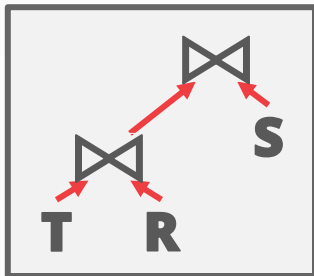
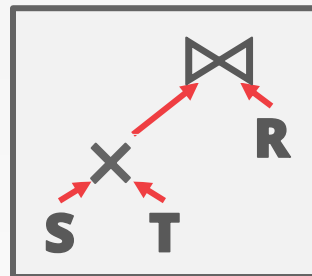
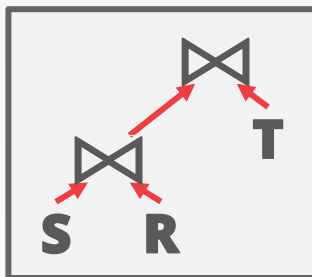
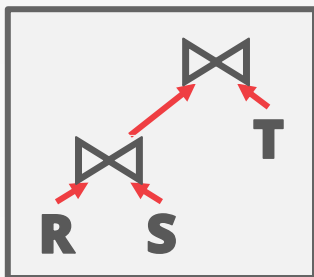
Step #1: Enumerate relation orderings



BRUTE-FORCE

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

Step #1: Enumerate relation orderings

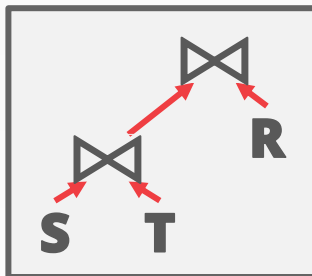
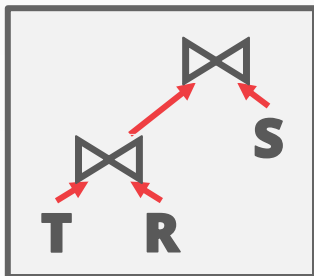
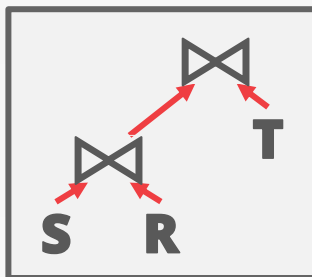
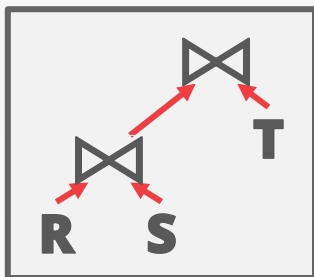


*Prune plans with
cross-products
immediately!*

BRUTE-FORCE

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

Step #1: Enumerate relation orderings

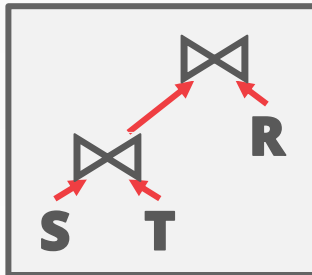
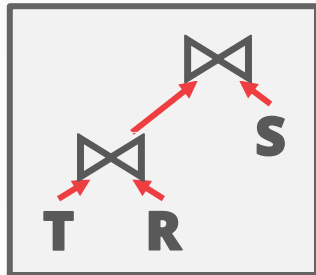
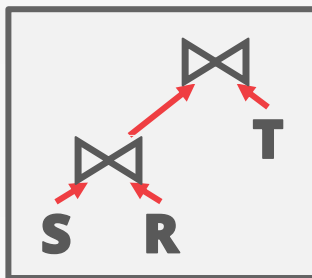
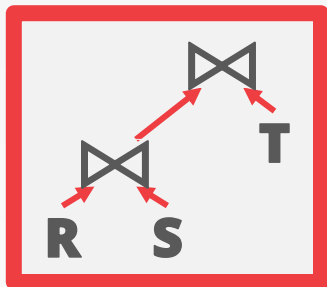


*Prune plans with
cross-products
immediately!*

BRUTE-FORCE

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

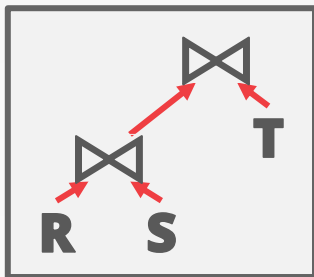
Step #1: Enumerate relation orderings



*Prune plans with
cross-products
immediately!*

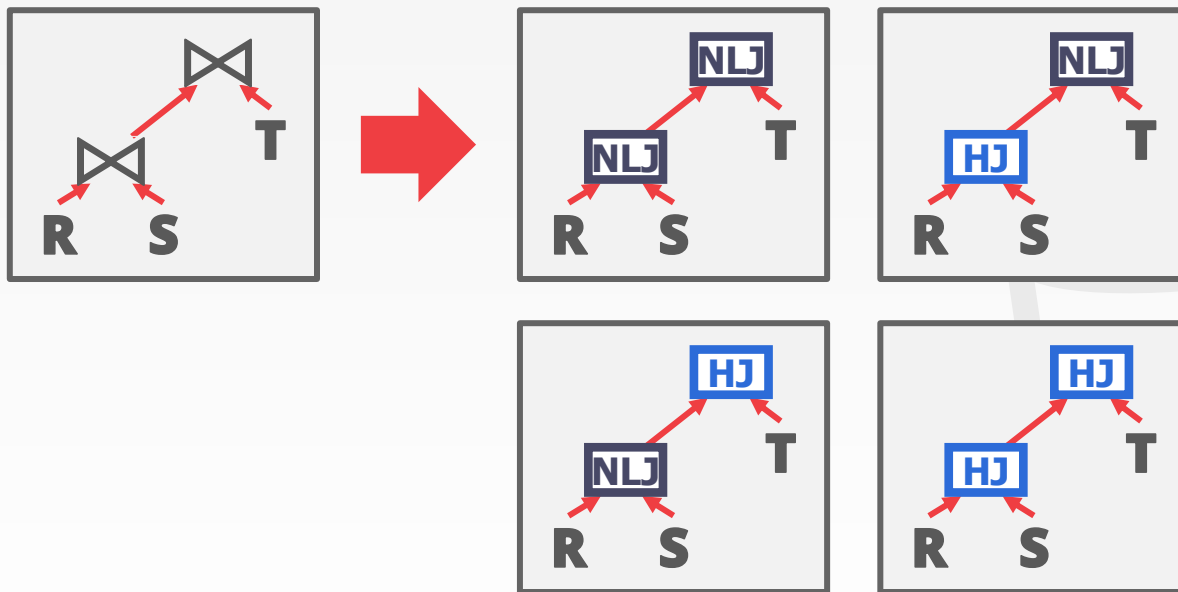
BRUTE-FORCE

Step #2: Enumerate join algorithm choices



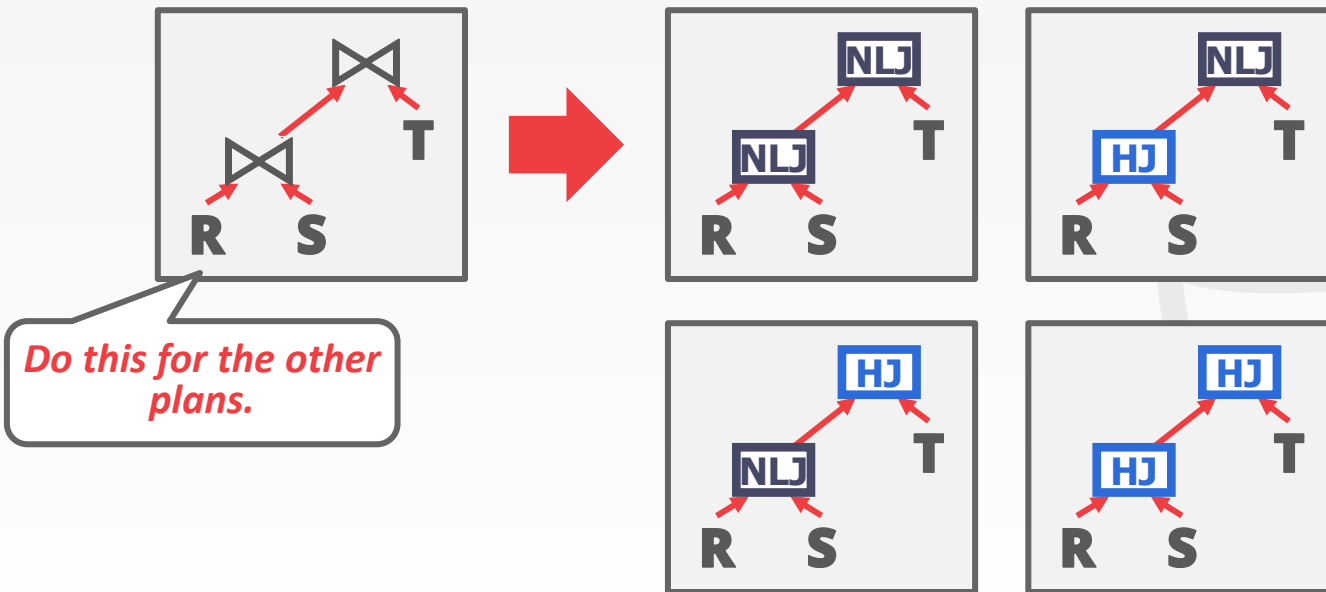
BRUTE-FORCE

Step #2: Enumerate join algorithm choices



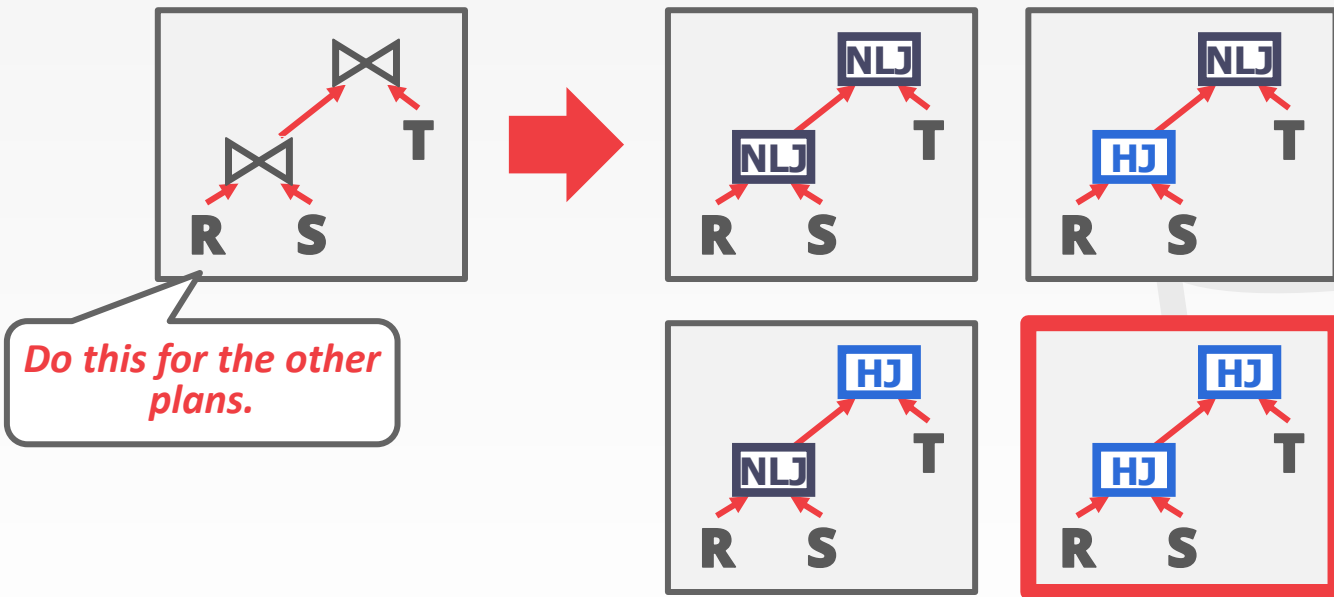
BRUTE-FORCE

Step #2: Enumerate join algorithm choices



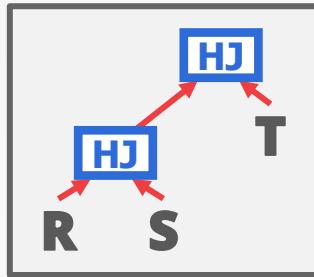
BRUTE-FORCE

Step #2: Enumerate join algorithm choices



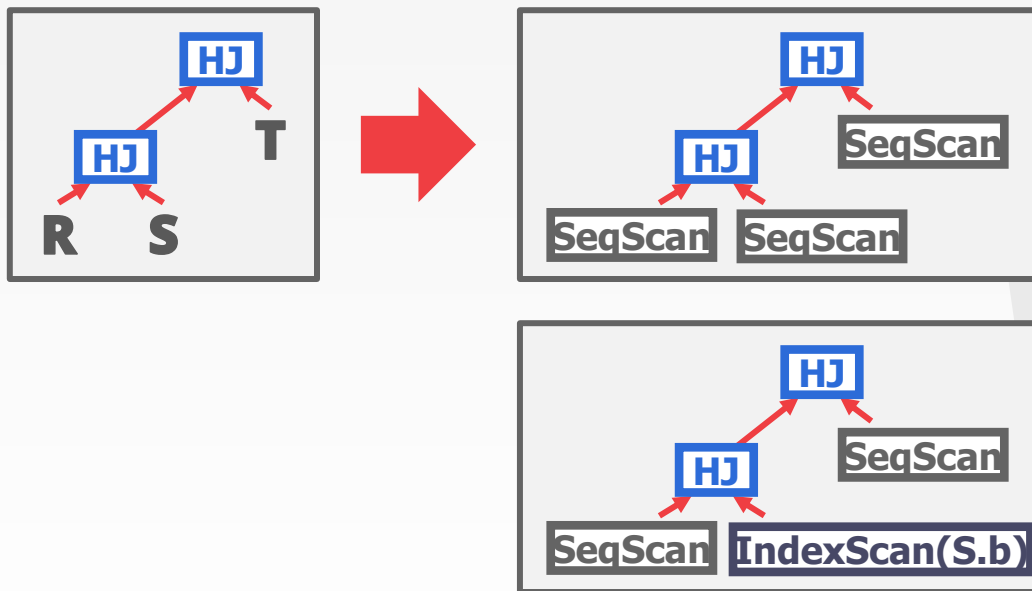
BRUTE-FORCE

Step #3: Enumerate access method choices



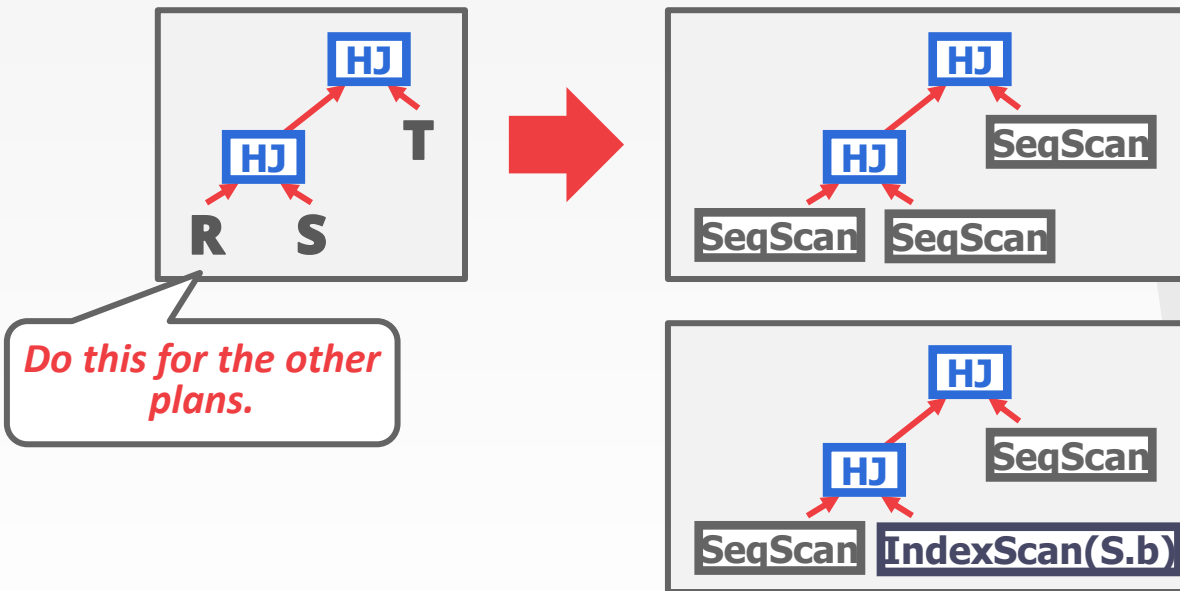
BRUTE-FORCE

Step #3: Enumerate access method choices



BRUTE-FORCE

Step #3: Enumerate access method choices



DYNAMIC PROGRAMMING

$$\begin{array}{c} R \bowtie S \\ T \end{array}$$

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

$$\begin{array}{c} R \\ S \\ T \end{array}$$

$$\begin{array}{c} T \bowtie S \\ R \\ \vdots \end{array}$$

$$R \bowtie S \bowtie T$$

DYNAMIC PROGRAMMING

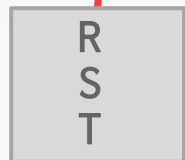
Hash Join

$R.a = S.a$



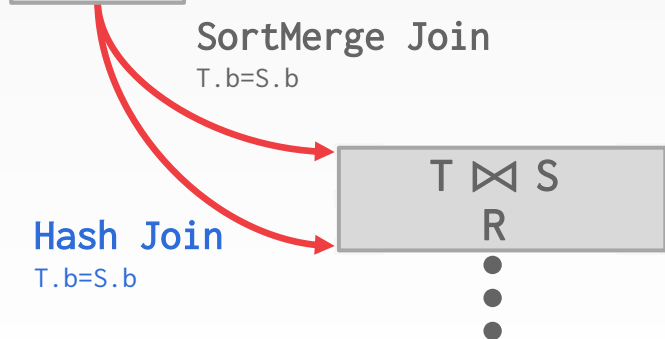
SortMerge Join

$R.a = S.a$



SortMerge Join

$T.b = S.b$



Hash Join

$T.b = S.b$

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



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:
280

Hash Join

T.b=S.b Cost:
200

R ⋈ S
T

R
S
T

T ⋈ S
R
⋮

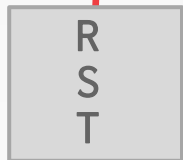
```
SELECT * FROM R, S, T
WHERE R.a = S.a
AND S.b = T.b
```

R ⋈ S ⋈ T

DYNAMIC PROGRAMMING

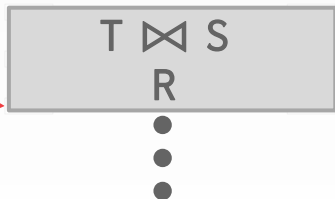
Hash Join

$R.a = S.a$ Cost: 300

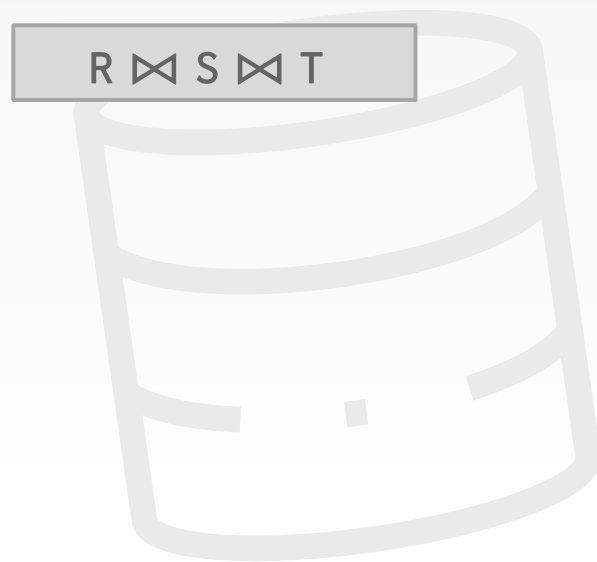


Hash Join

$T.b = S.b$ Cost: 200



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



DYNAMIC PROGRAMMING

Hash Join

R.a=S.a Cost: 300

Hash Join

S.b=T.b Cost: 380

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

SortMerge Join

S.b=T.b Cost: 400

SortMerge Join

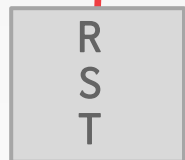
S.a=R.a Cost: 300

Hash Join

S.a=R.a Cost: 450

Hash Join

T.b=S.b Cost: 200



DYNAMIC PROGRAMMING

Hash Join

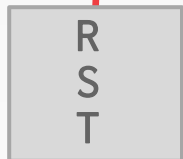
$R.a=S.a$ Cost: 300



Hash Join

$S.b=T.b$ Cost: 380

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



SortMerge Join

$S.a=R.a$ Cost: 300



Hash Join

$T.b=S.b$ Cost: 200

$R \bowtie S \bowtie T$

DYNAMIC PROGRAMMING

$$R \bowtie S$$

$$T$$

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

$$R$$

$$S$$

$$T$$

Hash Join

$T.b = S.b$

Cost:
200

SortMerge Join

$S.a = R.a$

Cost:
300

$$T \bowtie S$$

$$R$$

⋮

$$R \bowtie S \bowtie T$$

POSTGRES OPTIMIZER

Examines all types of join trees

→ Left-deep, Right-deep, bushy

Two optimizer implementations:

→ Traditional Dynamic Programming Approach

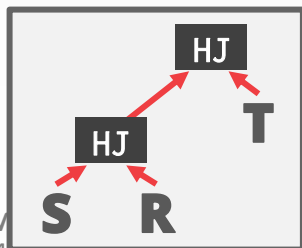
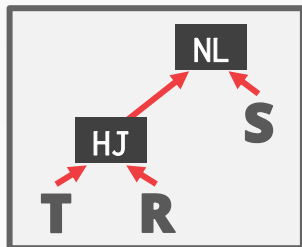
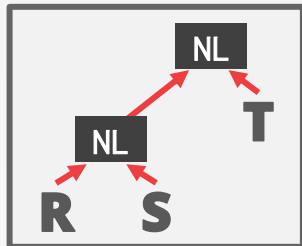
→ Genetic Query Optimizer (GEQO)

Postgres uses the traditional algorithm when # of tables in query is less than 12 and switches to GEQO when there are 12 or more.



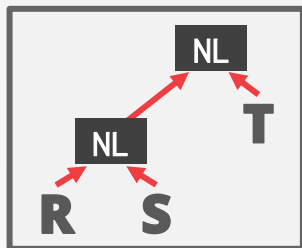
POSTGRES GENETIC OPTIMIZER

1st Generation

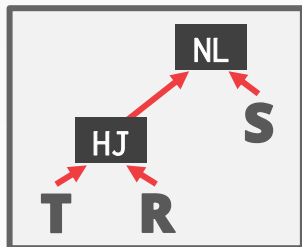


POSTGRES GENETIC OPTIMIZER

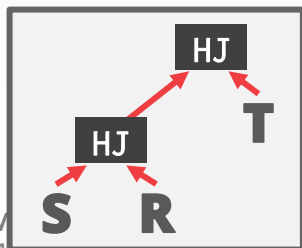
1st Generation



Cost:
300



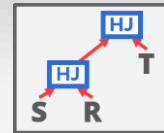
Cost:
200



Cost:
100

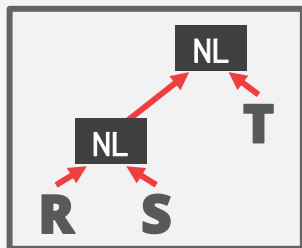


POSTGRES GENETIC OPTIMIZER

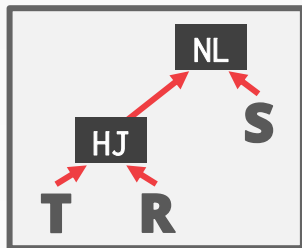


Best: 100

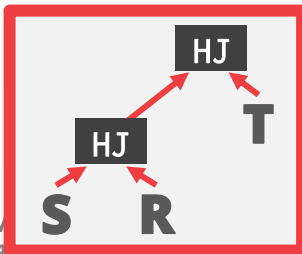
1st Generation



Cost:
300



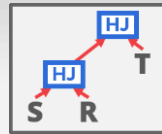
Cost:
200



Cost:
100

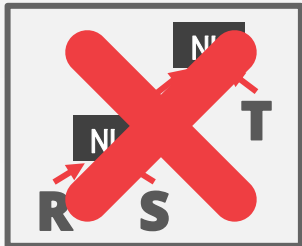


POSTGRES GENETIC OPTIMIZER

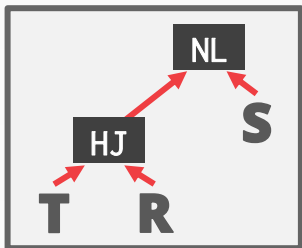


Best: 100

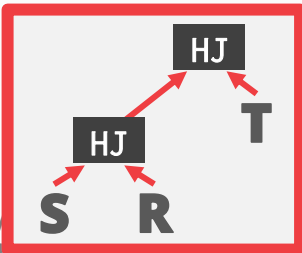
1st Generation



Cost:
300



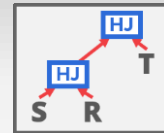
Cost:
200



Cost:
100

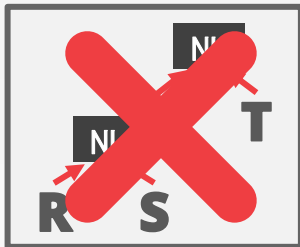


POSTGRES GENETIC OPTIMIZER

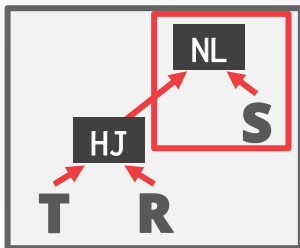


Best: 100

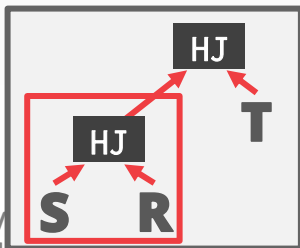
1st Generation



Cost:
300



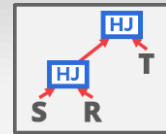
Cost:
200



Cost:
100

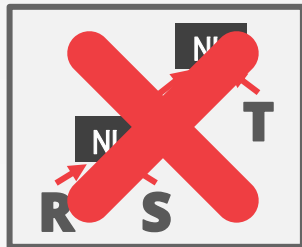


POSTGRES GENETIC OPTIMIZER

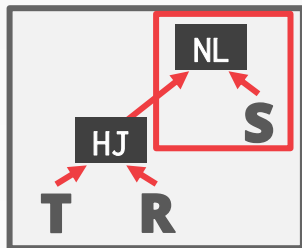


Best: 100

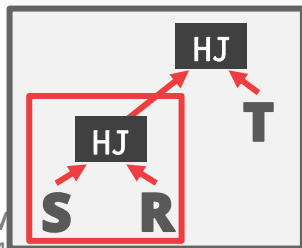
1st Generation



Cost:
300

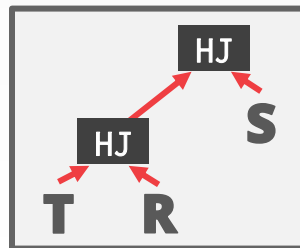
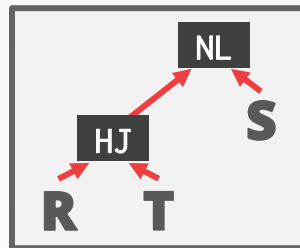
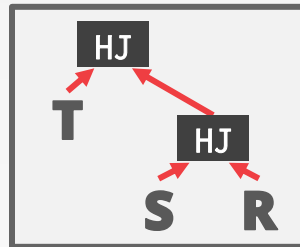


Cost:
200

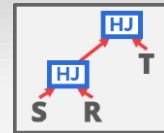


Cost:
100

2nd Generation

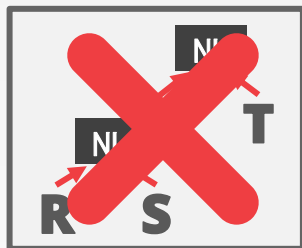


POSTGRES GENETIC OPTIMIZER

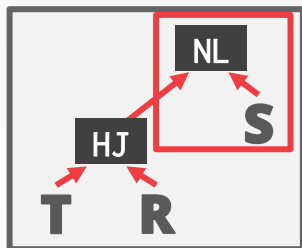


Best: 100

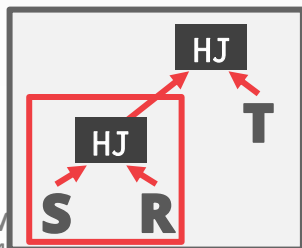
1st Generation



Cost:
300

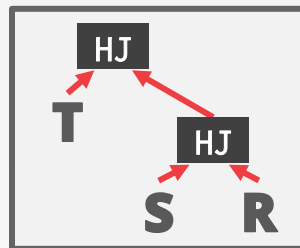


Cost:
200

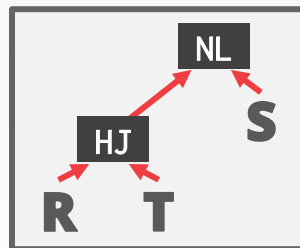


Cost:
100

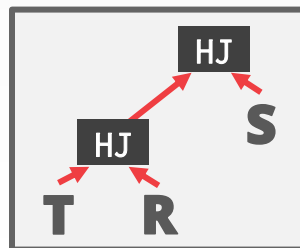
2nd Generation



Cost:
80



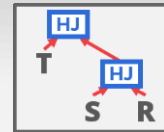
Cost:
200



Cost:
110

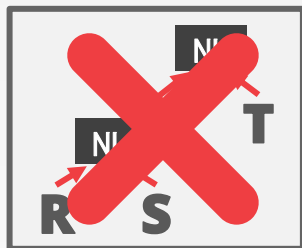


POSTGRES GENETIC OPTIMIZER

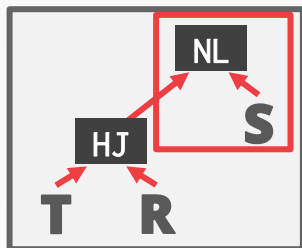


Best: 80

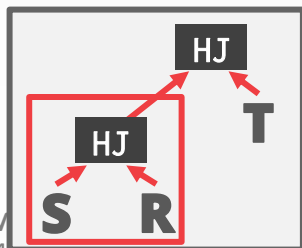
1st Generation



Cost:
300

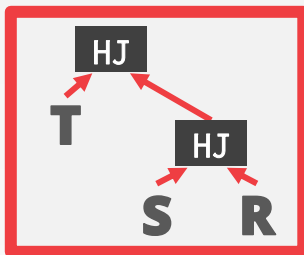


Cost:
200

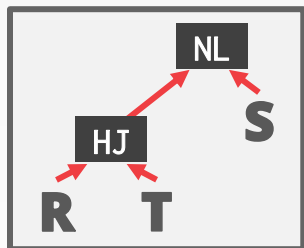


Cost:
100

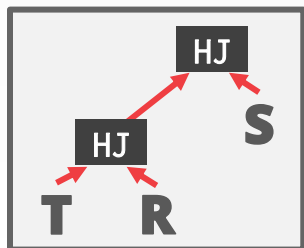
2nd Generation



Cost:
80



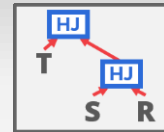
Cost:
200



Cost:
110

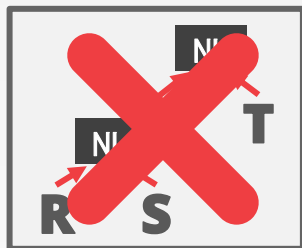


POSTGRES GENETIC OPTIMIZER

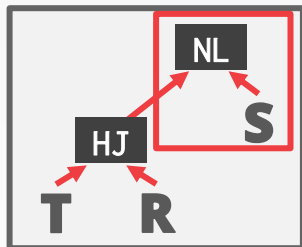


Best: 80

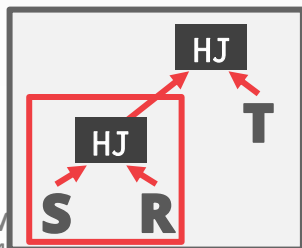
1st Generation



Cost:
300

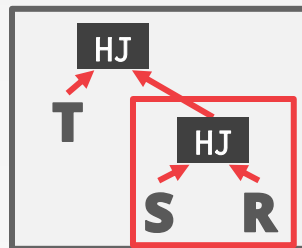


Cost:
200

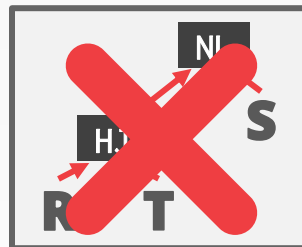


Cost:
100

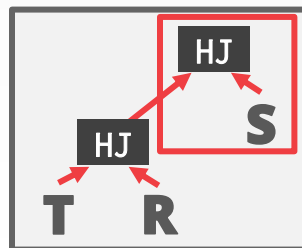
2nd Generation



Cost:
80

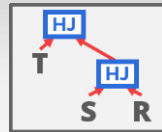


Cost:
200



Cost:
110

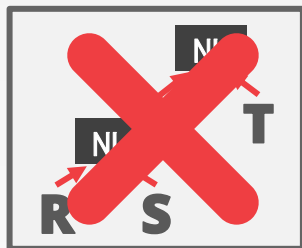
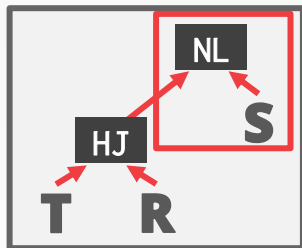
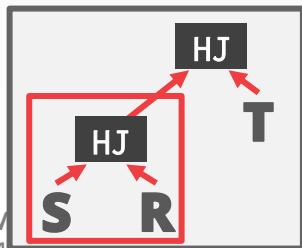




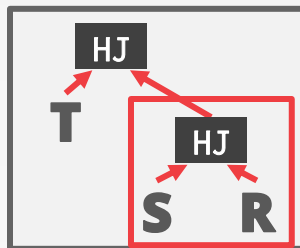
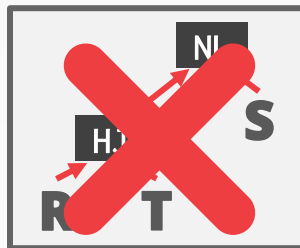
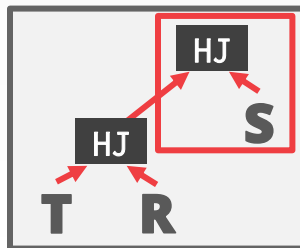
Best: 80

POSTGRES GENETIC OPTIMIZER

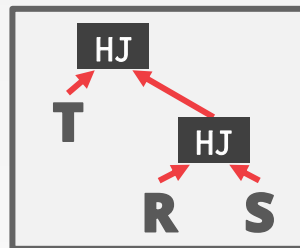
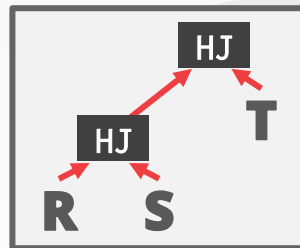
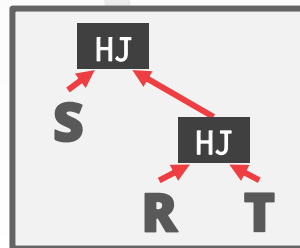
1st Generation

Cost:
300Cost:
200Cost:
100

2nd Generation

Cost:
80Cost:
200Cost:
110

3rd Generation

Cost:
90Cost:
160Cost:
120

CONCLUSION

Filter early as possible.

Selectivity estimations

- Uniformity
- Independence
- Inclusion
- Histograms
- Join selectivity

Dynamic programming for join orderings



NEXT CLASS

Transactions!

