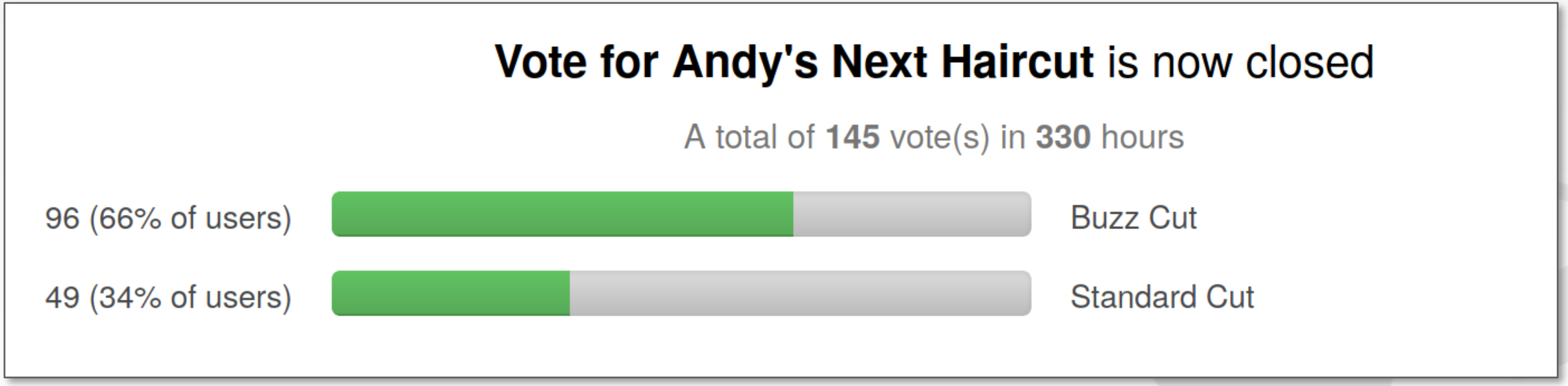
Carnegie Mellon University

15 Query Planning – Part II

Intro to Database Systems
15-445/15-645
Fall 2020

AP
Andy Pavlo
Computer Science
Carnegie Mellon University

HAIRCUT


Vote for Andy's Next Haircut is now closed
A total of 145 vote(s) in 330 hours
96 (66% of users)
Buzz Cut
49 (34% of users)
Standard Cut

ADMINISTRIVIA

Project #2 – C2 is due Sun Nov 1st @ 11:59pm

Project #3 will be released this week.
It is due Sun Nov 22nd @ 11:59pm.

Homework #4 will be released next week.
It is due Sun Nov 8th @ 11:59pm.

UPCOMING DATABASE TALKS

Datometry
→ Monday Oct 26th @ 5pm ET

MySQL Query Optimizer
→ Monday Nov 2nd @ 5pm ET

EraDB "Magical Indexes"
→ Monday Nov 9th @ 5pm ET

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

Cost Estimation

Plan Enumeration

COST-BASED QUERY PLANNING

Generate an estimate of the cost of executing a particular query plan for the current state of the database.

→ Estimates are only meaningful internally.

This is independent of the search strategies that we talked about last class.

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.

DISK-BASED DBMS COST MODEL

The number of disk accesses will always dominate the execution time of a query.

→ CPU costs are negligible.
→ Must consider sequential vs. random I/O.

This is easier to model if the DBMS has full control over buffer management.

→ We will know the replacement strategy, pinning, and assume exclusive access to disk.

POSTGRES COST MODEL

Uses a combination of CPU and I/O costs that are weighted by "magic" constant factors.

Default settings are obviously for a disk-resident database without a lot of memory:

→ Processing a tuple in memory is **400x** faster than reading a tuple from disk.
→ Sequential I/O is **4x** faster than random I/O.

19.7.2. Planner Cost Constants

The *cost* variables described in this section are measured on an arbitrary scale. Only their relative values matter, hence scaling them all up or down by the same factor will result in no change in the planner's choices. By default, these cost variables are based on the cost of sequential page fetches; that is, `seq_page_cost` is conventionally set to `1.0` and the other cost variables are set with reference to that. But you can use a different scale if you prefer, such as actual execution times in milliseconds on a particular machine.

> **Note:** Unfortunately, there is no well-defined method for determining ideal values for the cost variables. They are best treated as averages over the entire mix of queries that a particular installation will receive. This means that changing them on the basis of just a few experiments is very risky.

`seq_page_cost (floating point)`

Sets the planner's estimate of the cost of a disk page fetch that is part of a series of sequential fetches. The default is 1.0. This value can be overridden for tables and indexes in a particular tablespace by setting the tablespace parameter of the same name (see ALTER TABLESPACE).

`random_page_cost (floating point)`

IBM DB2 COST MODEL

Database characteristics in system catalogs

Hardware environment (microbenchmarks)

Storage device characteristics (microbenchmarks)

Communications bandwidth (distributed only)

Memory resources (buffer pools, sort heaps)

Concurrency Environment

→ Average number of users

→ Isolation level / blocking

→ Number of available locks

Source: Guy Lohman

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` 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?

SELECTION STATISTICS

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 selectivity of complex predicates is more difficult…

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

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

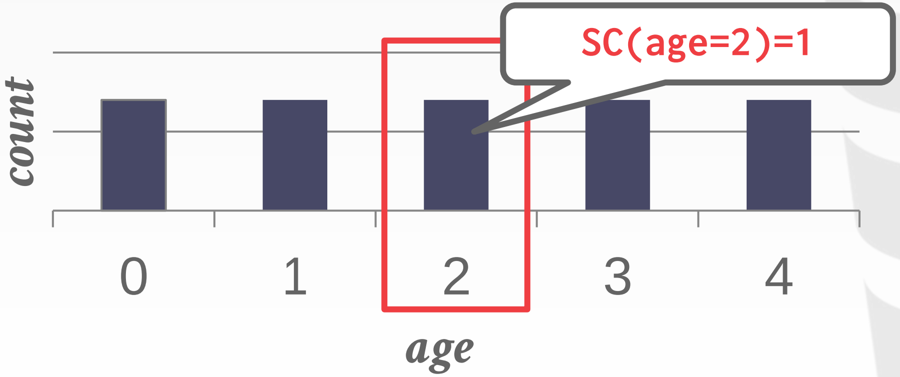
Assume that V(age,people) has five distinct values (0–4) and N_R = 5

Equality Predicate: A=constant

→ sel(A=constant) = SC(P) / N_R

→ Example: sel(age=2) = 1/5

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



SELECTIONS – COMPLEX PREDICATES

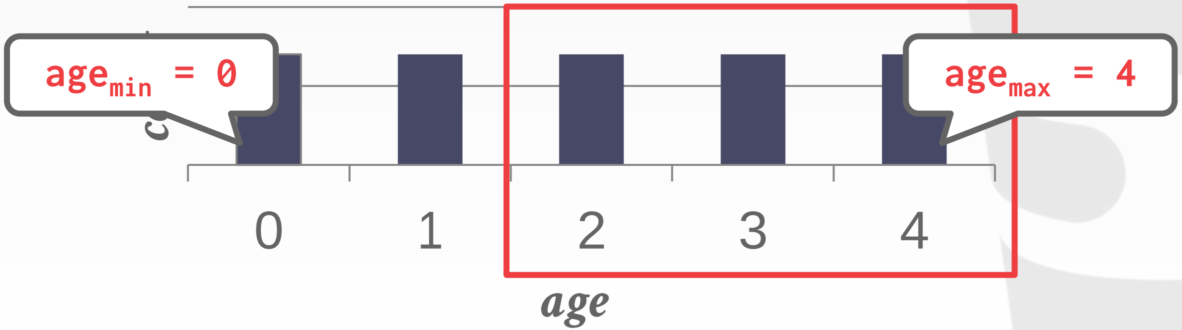
Range Predicate:

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

→ Example: sel(age>=2) ≈ (4−2+1) / (4−0+1)

≈ 3/5

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



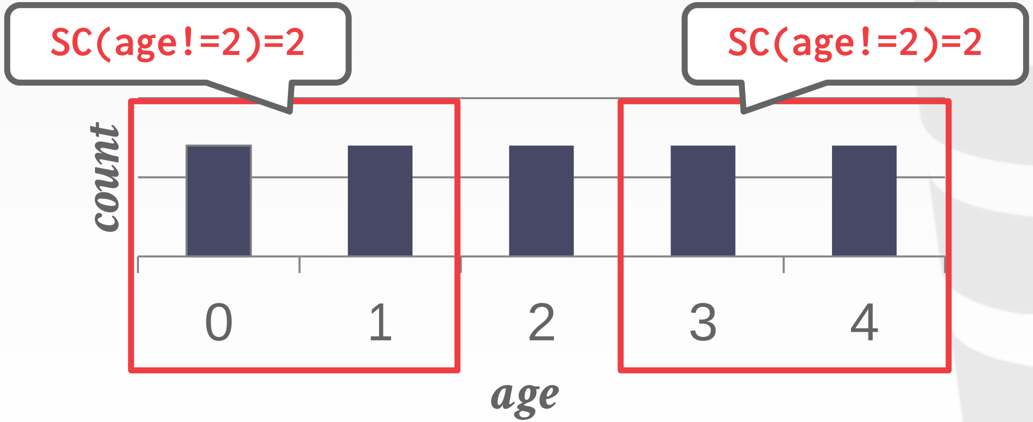
SELECTIONS – COMPLEX PREDICATES

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
```



SELECTIONS – COMPLEX PREDICATES

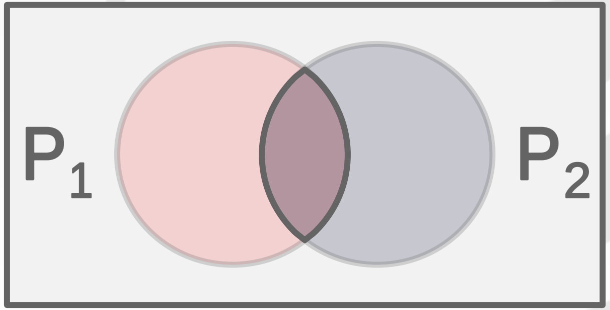
Conjunction:

→ sel(P1 ∧ P2) = sel(P1) • sel(P2)

→ sel(age=2 ∧ name LIKE 'A%')

This assumes that the predicates are **independent**.

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



SELECTIONS – COMPLEX PREDICATES

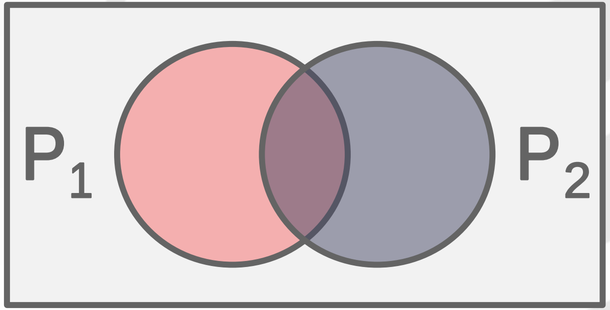
Disjunction:

→ sel(P1 ⋁ P2)
  = sel(P1) + sel(P2) - sel(P1⋀P2)
  = sel(P1) + sel(P2) - sel(P1) • 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%'
```



SELECTIONS – COMPLEX PREDICATES

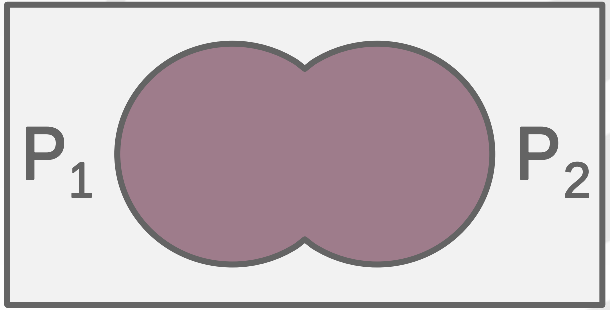
Disjunction:

→ sel(P1 ⋁ P2)
 = sel(P1) + sel(P2) - sel(P1⋀P2)
 = sel(P1) + sel(P2) - sel(P1) • 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:
→ # of Makes = 10, # of Models = 100

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

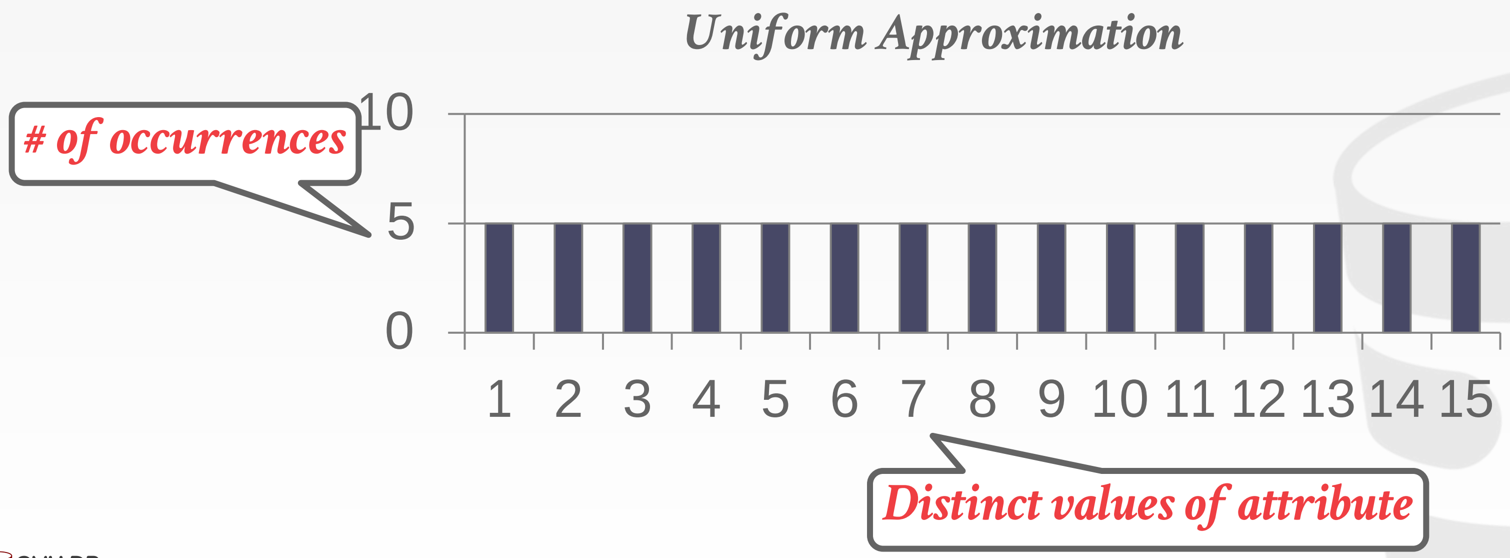
With the independence and uniformity assumptions, the selectivity is:
→ $1/10 \times 1/100 = 0.001$

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

Source: Guy Lohman

COST ESTIMATIONS

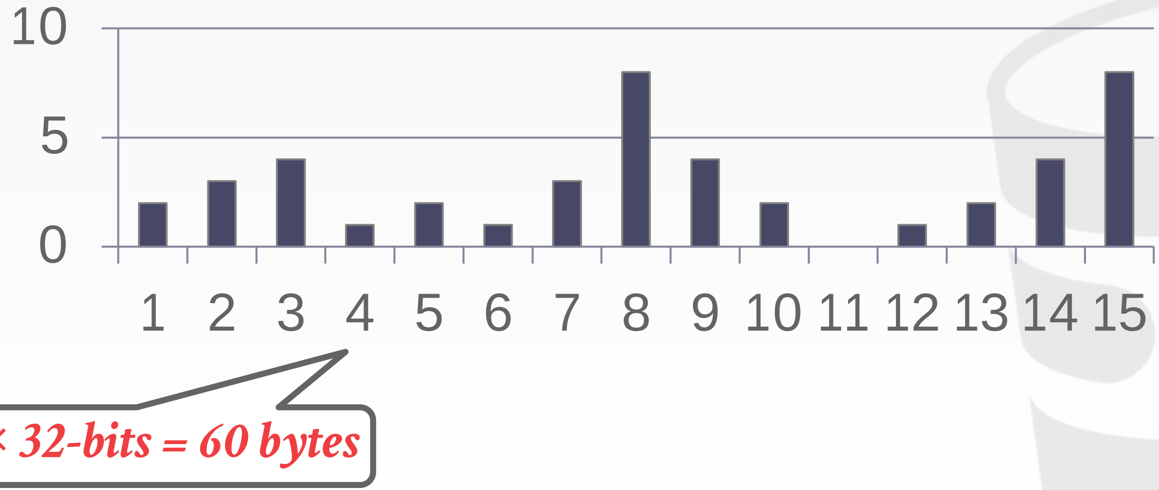
Our formulas are nice, but we assume that data values are uniformly distributed.



COST ESTIMATIONS

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

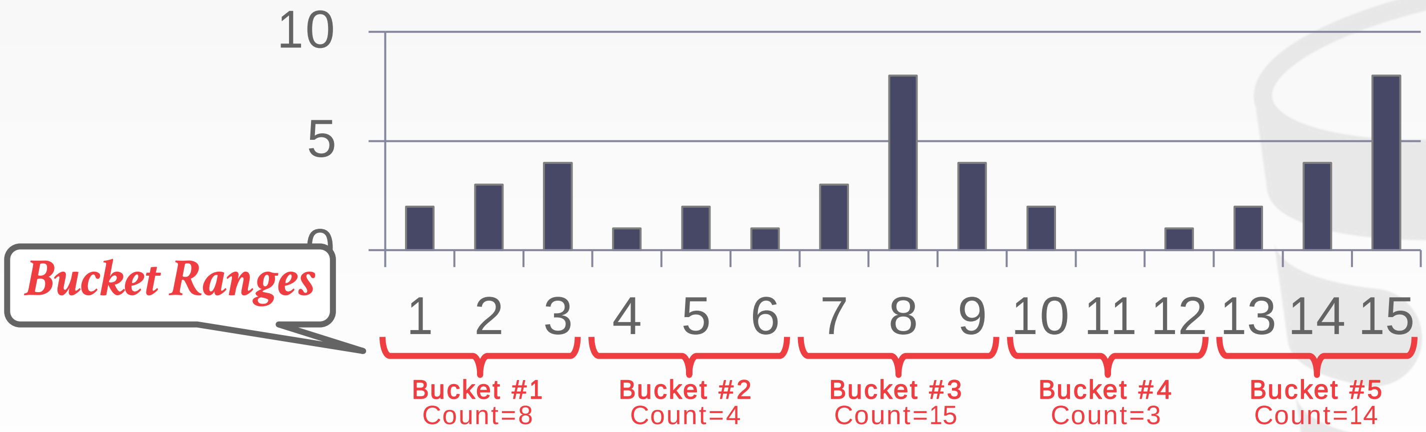
Non-Uniform Approximation



EQUI-WIDTH HISTOGRAM

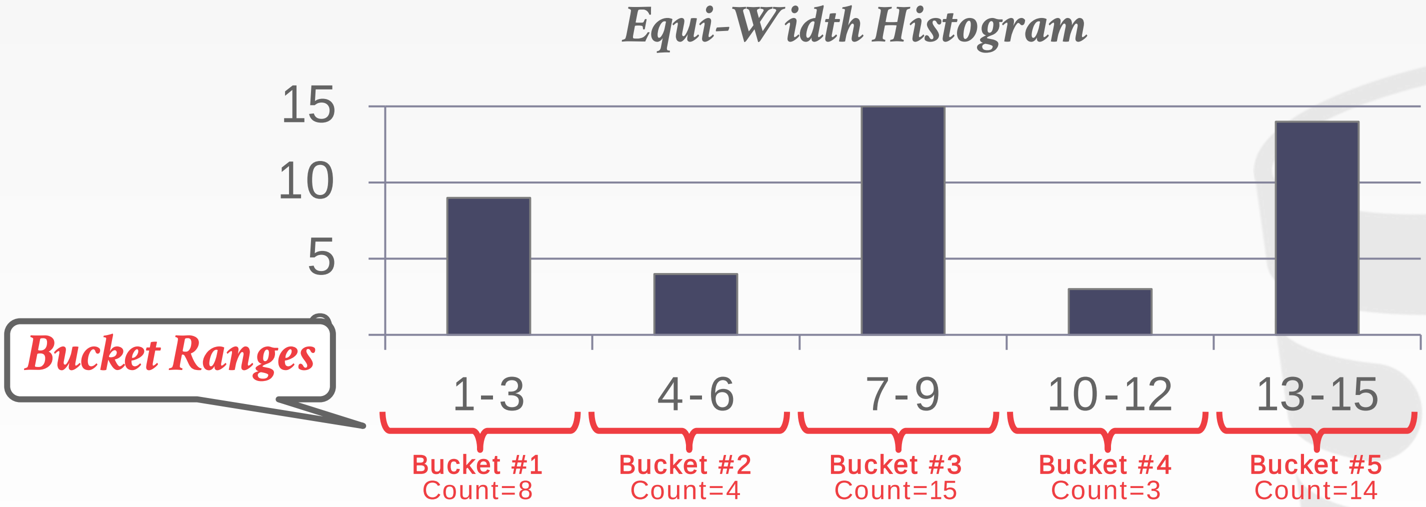
All buckets have the same width (i.e., the same number of values).

Non-Uniform Approximation



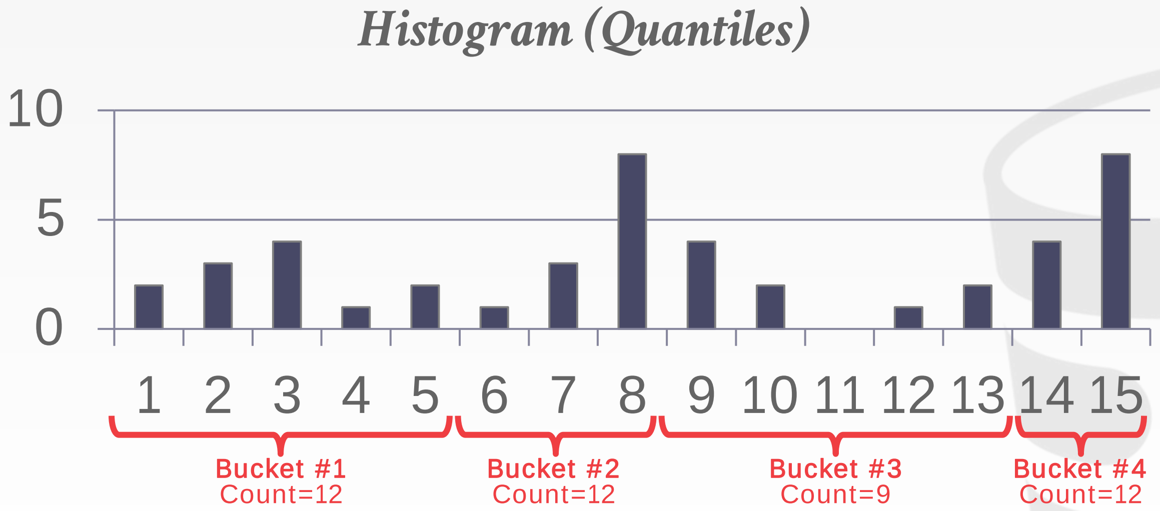
EQUI-WIDTH HISTOGRAM

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



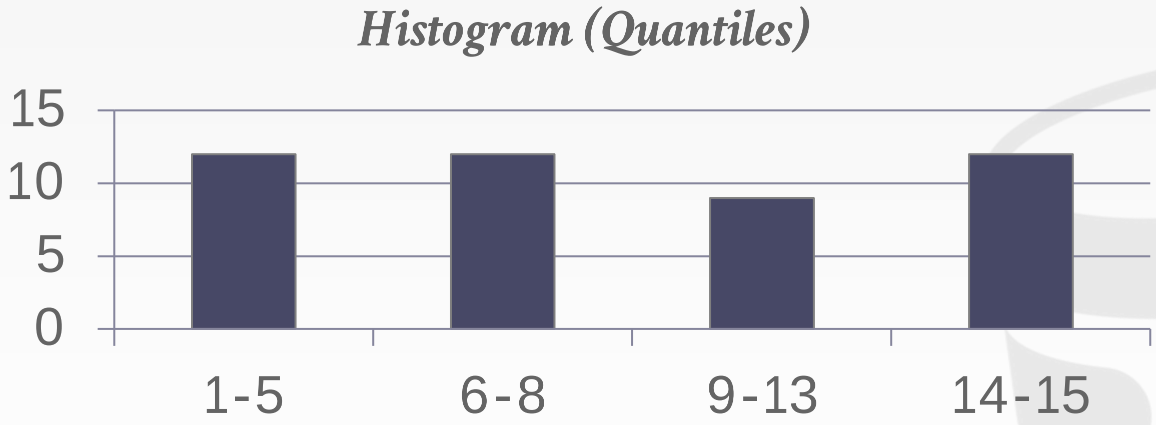
EQUI-DEPTH HISTOGRAMS

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.



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.

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	Obama	59	Rested
1002	Kanye	41	Weird
1003	Tupac	25	Dead
1004	Bieber	26	Crunk
1005	Andy	39	Shaved
1006	TigerKing	57	Jailed

⋮

1 billion tuples

Table Sample

1001	Obama	59	Rested
1003	Tupac	25	Dead
1005	Andy	39	Shaved

sel(age>50) = 1/3

OBSERVATION

Now that we can (roughly) estimate the selectivity of predicates, 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…

OLTP QUERY PLANNING

Query planning for OLTP queries is easy because they are **sargable** (**S**earch **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.

```
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

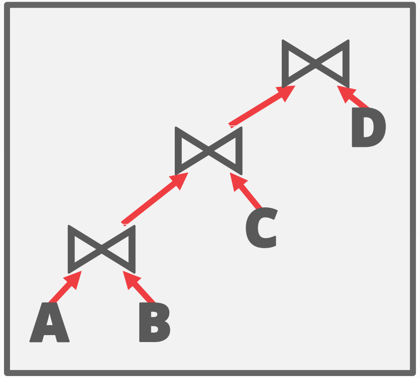
→ We need to restrict search space.

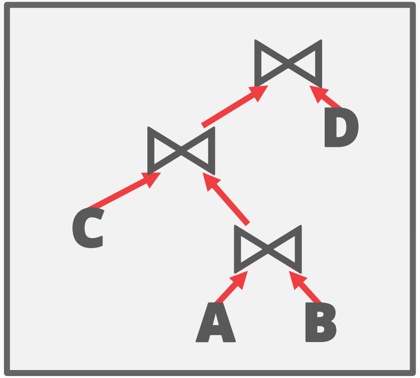
Fundamental decision in **System R**: only left-deep join trees are considered.

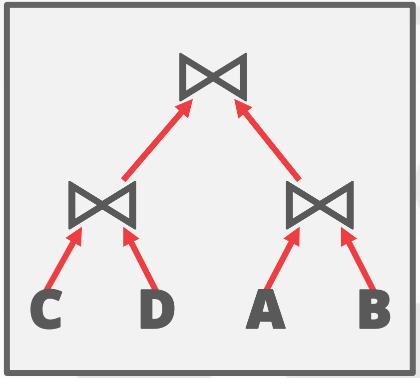
→ Modern DBMSs do not always make this assumption anymore.

MULTI-RELATION QUERY PLANNING

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

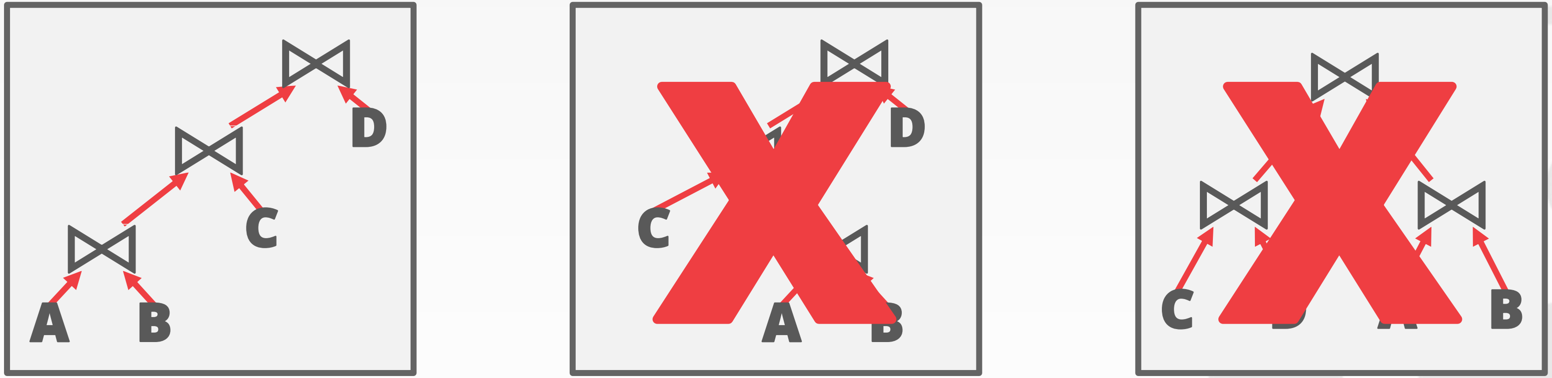






MULTI-RELATION QUERY PLANNING

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



MULTI-RELATION QUERY PLANNING

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.

MULTI-RELATION QUERY PLANNING

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

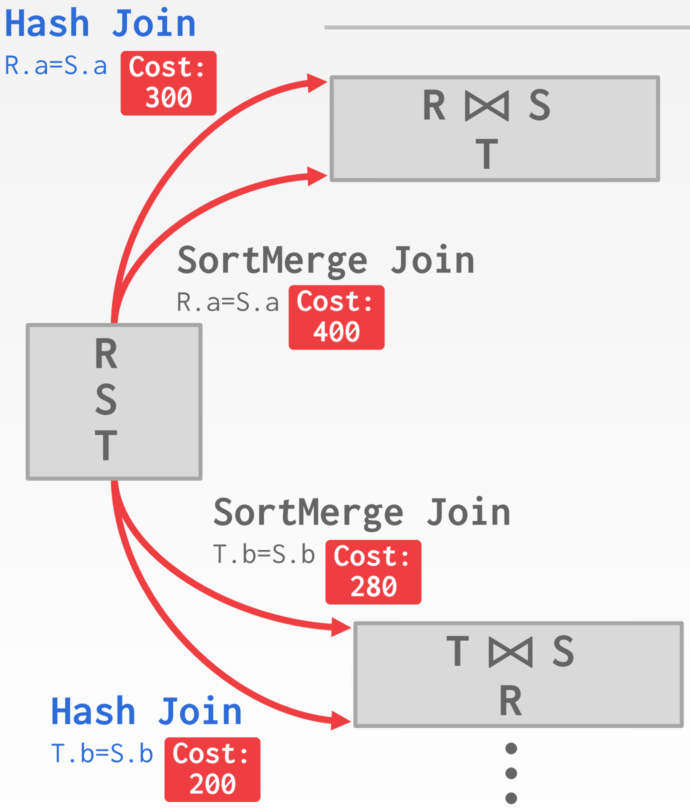
Enumerate the plans for each operator
→ Example: Hash, Sort-Merge, Nested Loop…

Enumerate the access paths for each table
→ Example: Index #1, Index #2, Seq Scan…

Use **dynamic programming** to reduce the number of cost estimations.

DYNAMIC PROGRAMMING

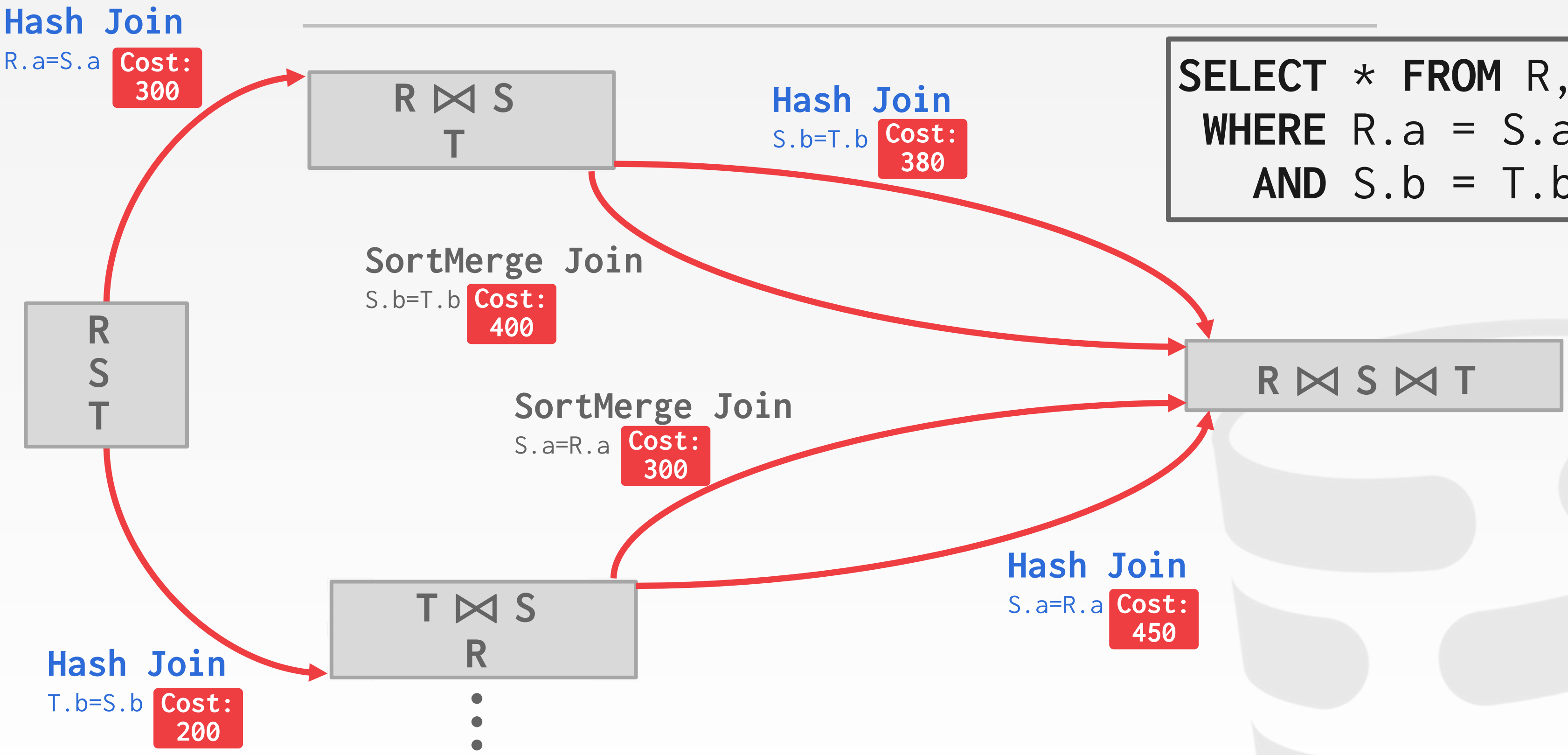
```
SELECT * FROM R, S, T
 WHERE R.a = S.a
   AND S.b = T.b
```



R ⋈ S ⋈ T

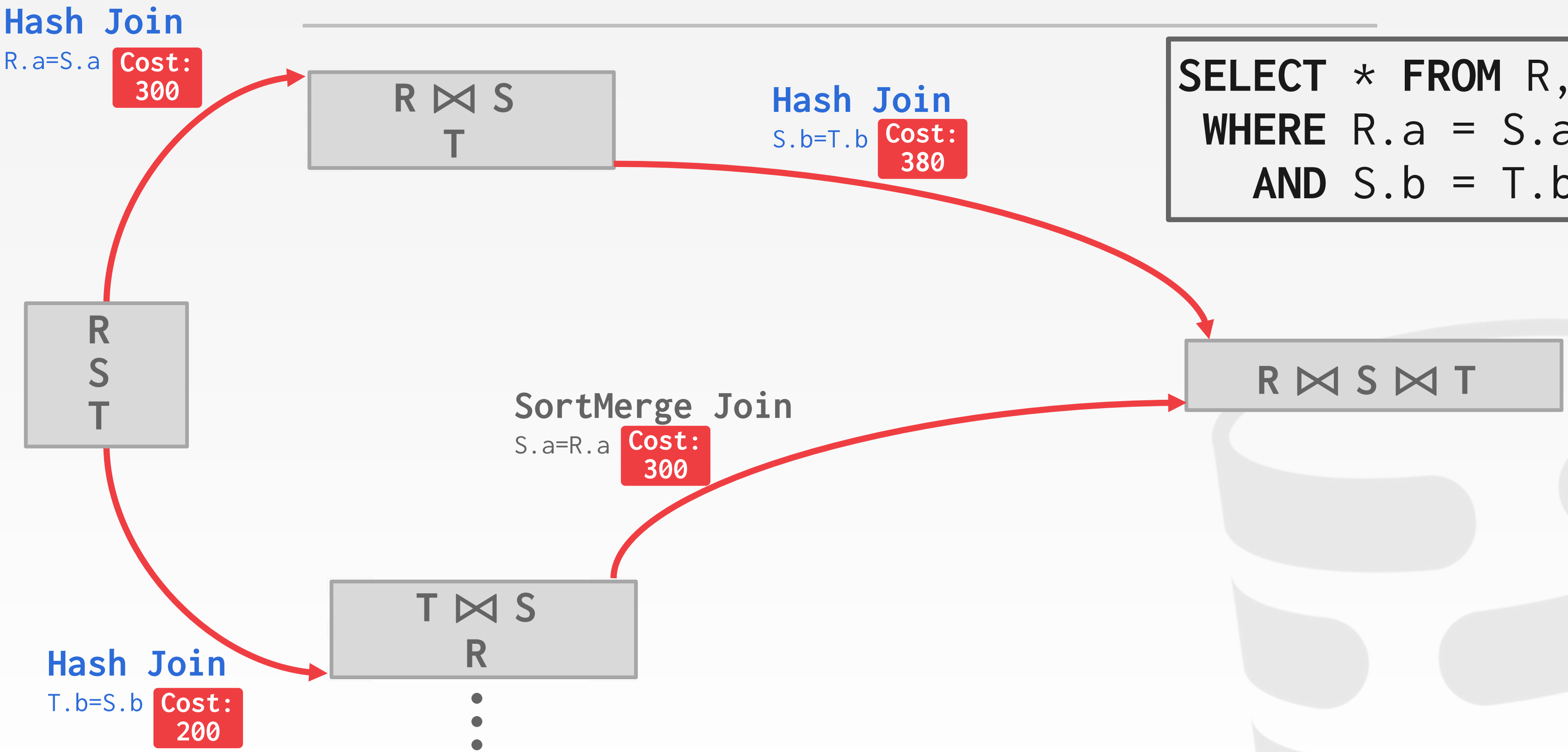
DYNAMIC PROGRAMMING

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



DYNAMIC PROGRAMMING

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



DYNAMIC PROGRAMMING

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

R ⋈ S
T

R
S
T

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

R ⋈ S ⋈ T

T ⋈ S
R

⋮

Hash Join
T.b=S.b Cost: 200

CANDIDATE PLAN EXAMPLE

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

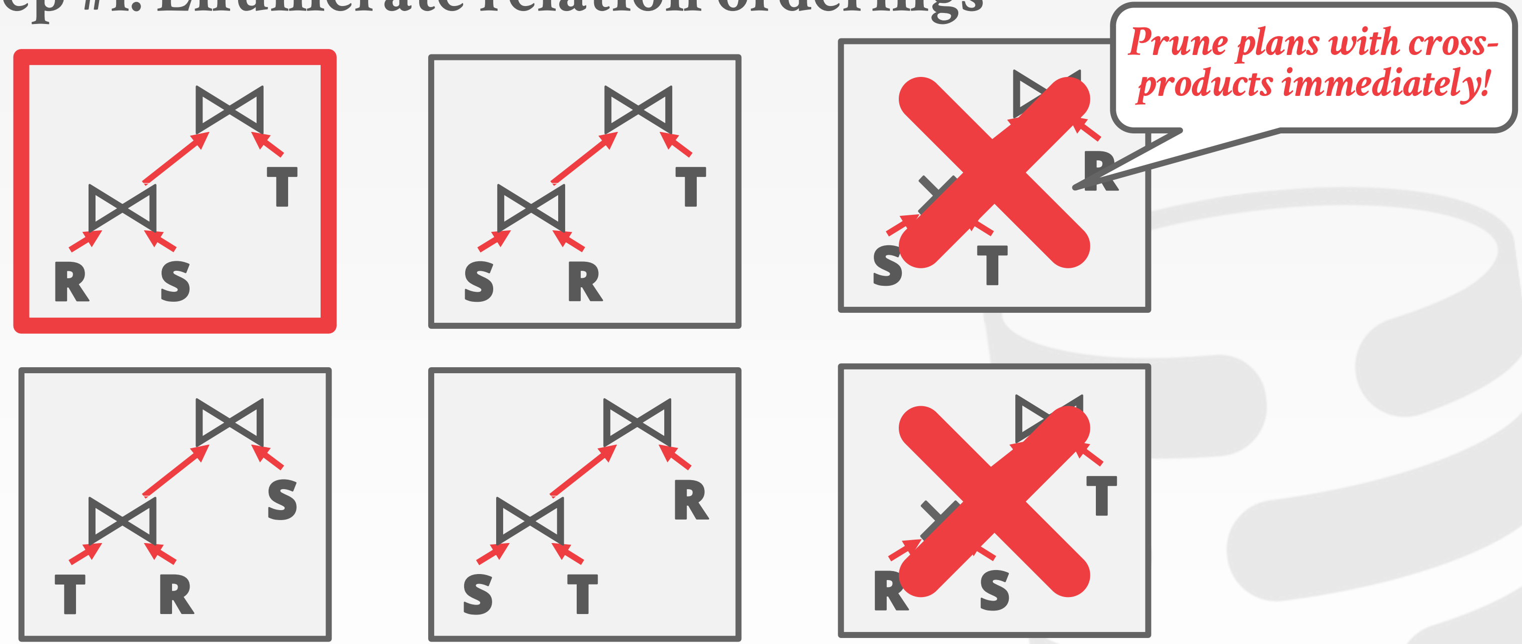
How to generate plans for search algorithm:

→ Enumerate relation orderings
→ Enumerate join algorithm choices
→ Enumerate access method choices

No real DBMSs does it this way.
It's actually more messy…

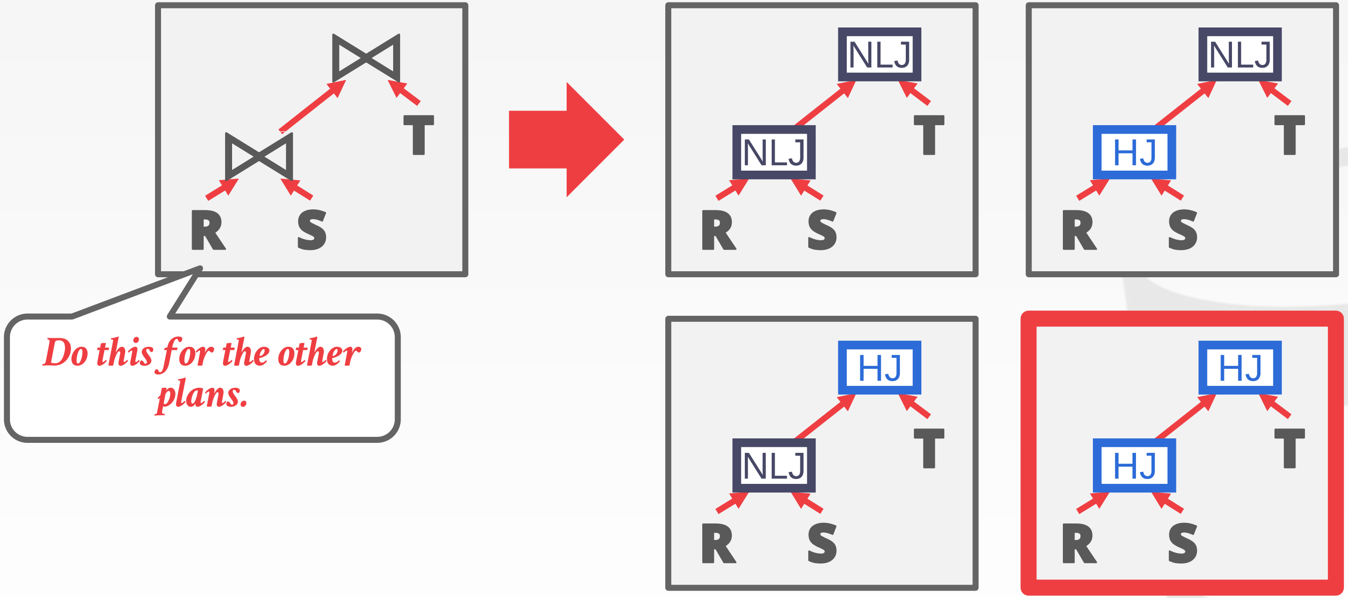
CANDIDATE PLANS

Step #1: Enumerate relation orderings



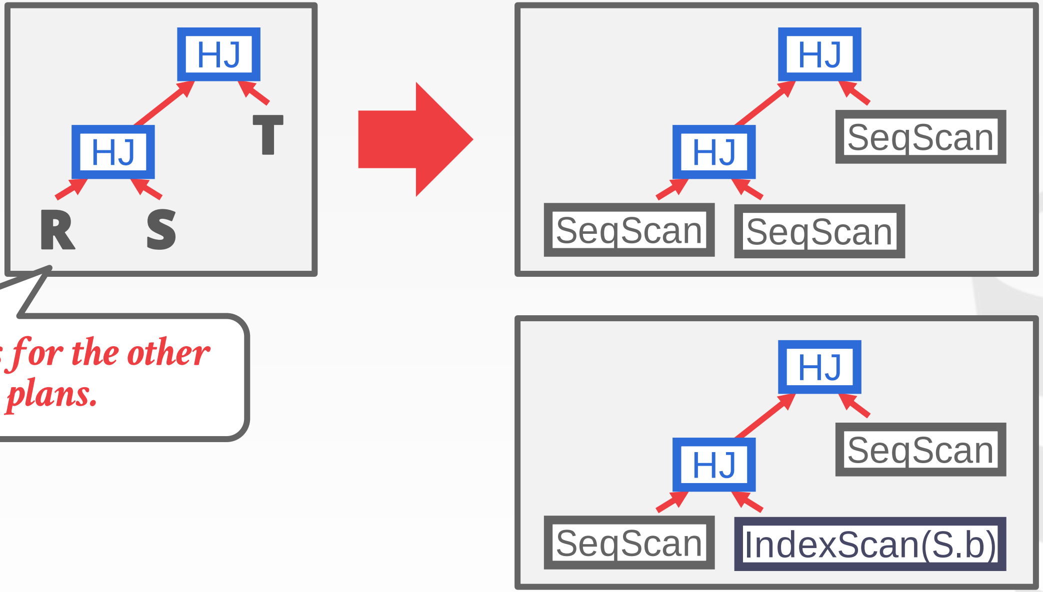
CANDIDATE PLANS

Step #2: Enumerate join algorithm choices



CANDIDATE PLANS

Step #3: Enumerate access method choices



Do this for the other plans.

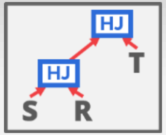
POSTGRES OPTIMIZER

Examines all types of join trees
→ Left-deep, Right-deep, bushy

Two optimizer implementations:
→ Traditional Dynamic Programming Approach
→ Genetic Query Optimizer (GEQO)

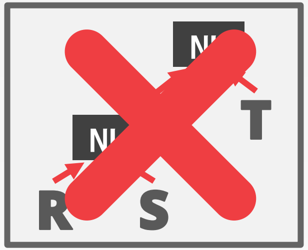
Postgres uses the traditional algorithm when # of tables in query is **less** than 12 and switches to GEQO when there are 12 or more.

POSTGRES GENETIC OPTIMIZER

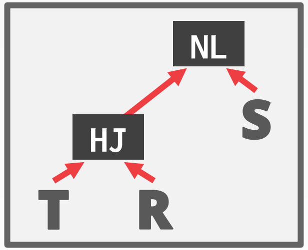


Best:100

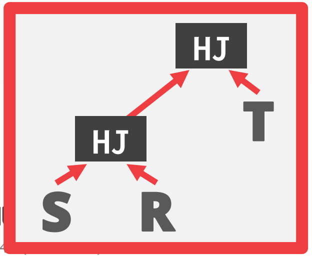
1st Generation



Cost: 300

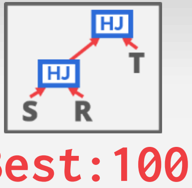


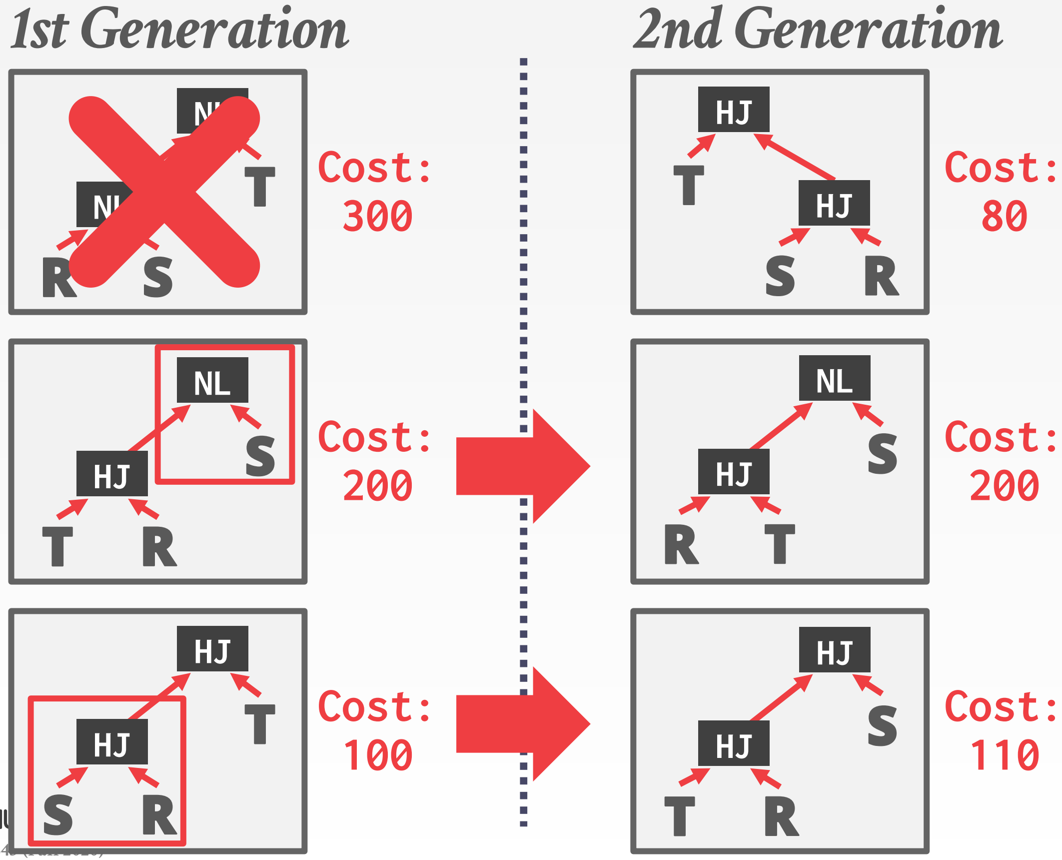
Cost: 200



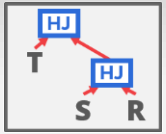
Cost: 100

POSTGRES GENETIC OPTIMIZER





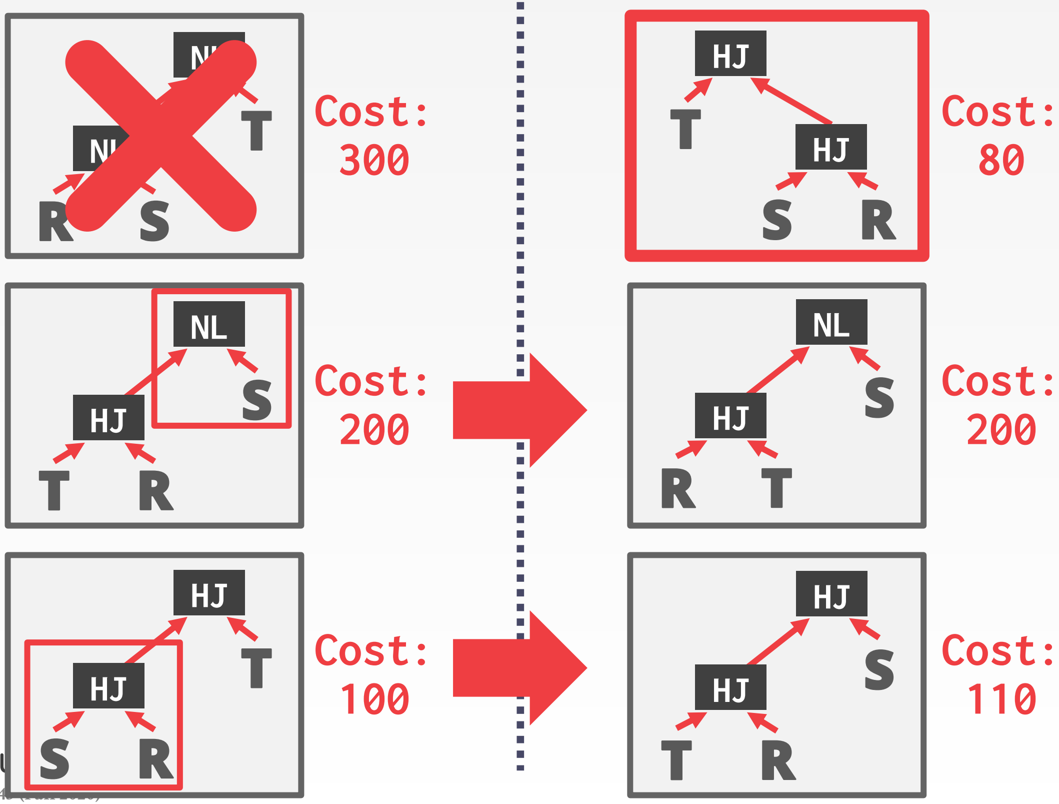
POSTGRES GENETIC OPTIMIZER



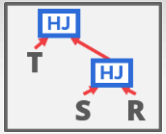
Best:80

1st Generation

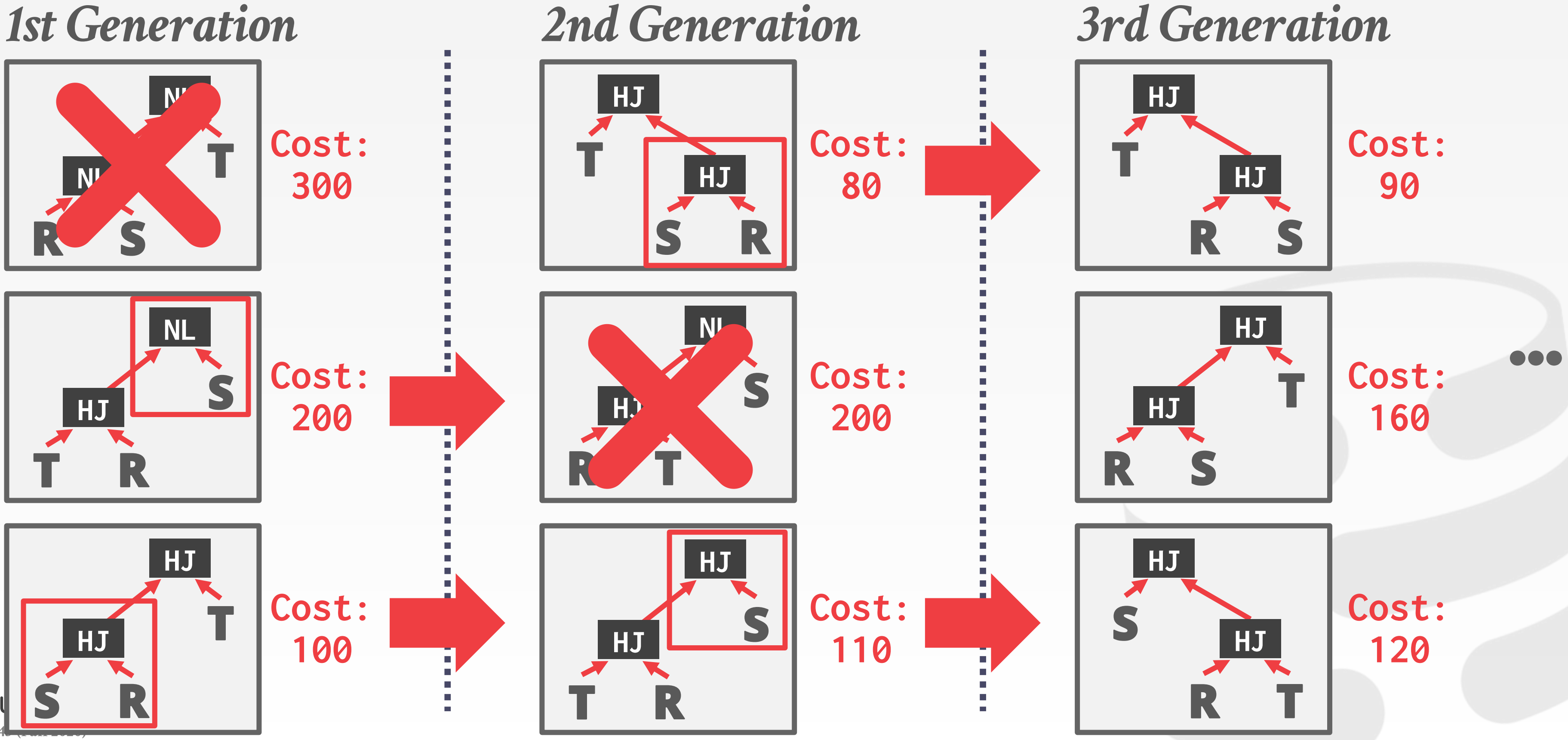
2nd Generation



POSTGRES GENETIC OPTIMIZER



Best:80



CONCLUSION

Filter early as possible.

Selectivity estimations
→ Uniformity
→ Independence
→ Histograms
→ Join selectivity

Dynamic programming for join orderings

Again, query optimization is hard…

NEXT CLASS

Transactions!

→ aka the second hardest part about database systems