

Relational Query Optimization II: Costing and Searching

What is needed for query optimization?

- Given: A closed set of operators
 - Relational ops (table in, table out)
 - Physical implementations (of those ops and a few more)
- 1. **Plan space**
 - Based on relational equivalences, different implementations
- 2. **Cost Estimation** based on
 - Cost formulas
 - Size estimation, in turn based on
 - Catalog information on base tables
 - Selectivity (Reduction Factor) estimation
- 3. **A search algorithm**
 - To sift through the plan space and find lowest cost option!

Reminder

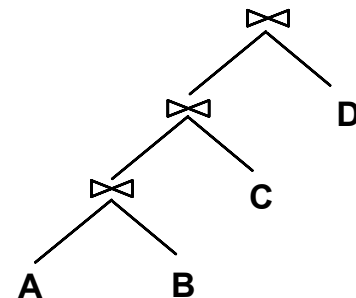
- We're focusing on “System R” (“Selinger”) optimizers
 - Remarkably comprehensive framework
 - Many of the details have been refined over time
 - We'll see some refinements today
 - This remains an area of ongoing research!

Big Picture of System R Optimizer

- Works well for up to 10-15 joins.
- **Plan Space:** Too large, must be pruned.
 - Algorithmic insight:
 - Many plans could have the same “overpriced” subtree
 - Ignore all those plans
 - Common heuristic: consider only left-deep plans
 - Common heuristic: avoid Cartesian products
- Cost estimation
 - Very inexact, but works ok in practice.
 - Stats in system catalogs used to estimate sizes & costs
 - Considers combination of CPU and I/O costs.
 - System R’s scheme has been improved since that time.
- Search Algorithm: Dynamic Programming

Query Blocks: Units of Optimization

- Break query into query blocks
- Optimize one block at a time
- Uncorrelated nested blocks computed once
- Correlated nested blocks are like function calls
 - But sometimes can be “decorrelated”
 - Beyond the scope of CS186!



```
SELECT S.sname  
FROM Sailors S  
WHERE S.age IN
```

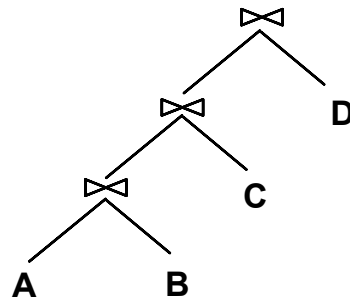
Outer block

```
(SELECT MAX (S2.age)  
FROM Sailors S2  
GROUP BY S2.rating)
```

Nested block

Query Blocks: Units of Optimization Pt 2

- For each block, the plans considered are:
 - All relevant access methods, for each relation in FROM clause.
 - All left-deep join trees
 - right branch always a base table
 - consider all join orders and join methods



```
SELECT S.sname
  FROM Sailors S
 WHERE S.age IN
```

Outer block

```
(SELECT MAX (S2.age)
  FROM Sailors S2
 GROUP BY S2.rating)
```

Nested block

Schema for Examples

Sailors (*sid*: integer, *sname*: text, *rating*: integer,
age: float)

Reserves (*sid*: integer, *bid*: integer, *day*: date,
rname: text)

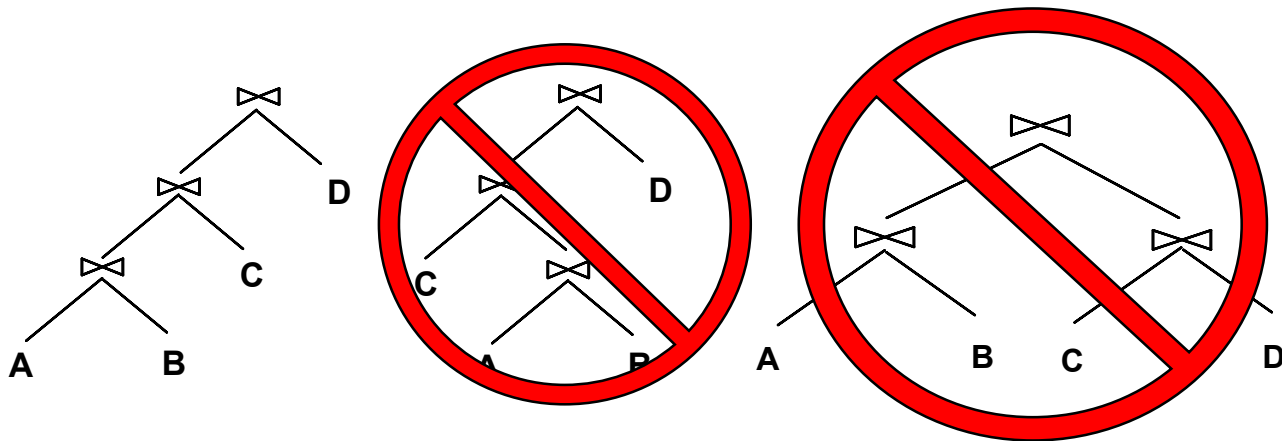
- Reserves:
 - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages.
 - 100 distinct bids.
- Sailors:
 - Each tuple is 50 bytes long,
 - 80 tuples per page, 500 pages.
 - 10 ratings, 40,000 sids.

“Physical” Properties

- Two common “physical” properties of an output:
 - Sort order
 - Hash Grouping
- Certain operators produce these properties in output
 - E.g. Index scan (result is sorted)
 - E.g. Sort (result is sorted)
 - E.g. Hash (result is grouped)
- Certain operators require these properties at input
 - E.g. MergeJoin requires sorted input
- Certain operators preserve these properties from inputs
 - E.g. MergeJoin preserves sort order of inputs
 - E.g. INLJ preserves sort order of outer (left) input

Queries Over Multiple Relations

- A System R heuristic: only left-deep join trees considered.
 - Restricts the search space
 - Left-deep trees allow us to generate all fully pipelined plans.
 - Intermediate results not written to temporary files.
 - Not all left-deep trees are fully pipelined (e.g., SM join).



Plan Space Review

- For a SQL query, full plan space:
 - All equivalent relational algebra expressions
 - Based on the equivalence rules we learned
 - All mixes of physical implementations of those algebra expressions
- We might prune this space:
 - Selection/Projection pushdown
 - Left-deep trees only
 - Avoid cartesian products
- Along the way we may care about physical properties like sorting
 - Because downstream ops may depend on them
 - And enforcing them later may be expensive

Query Optimization: Cost Estimation

1. Plan Space
- 2. Cost Estimation**
3. Search Algorithm

Cost Estimation

- For each plan considered, must estimate total cost:
 - Must estimate **cost** of each operation in plan tree.
 - Depends on input cardinalities.
 - We've already discussed this for various operators
 - sequential scan, index scan, joins, etc.
 - Must estimate **size of result** for each operation in tree!
 - Because it determines downstream input cardinalities!
 - Use information about the input relations.
 - For selections and joins, assume independence of predicates.
- In System R, cost is boiled down to a single number consisting of $\#I/O + \text{CPU-factor} * \#tuples$

Statistics and Catalogs

- Need info on relations and indexes involved.
- **Catalogs** typically contain at least:

Statistic	Meaning
NTuples	# of tuples in a table (cardinality)
NPages	# of disk pages in a table
Low/High	min/max value in a column
Nkeys	# of distinct values in a column
IHeight	the height of an index
INPages	# of disk pages in an index

- Catalogs updated periodically.
 - Too expensive to do continuously
 - Lots of approximation anyway, so a little slop here is ok.
- Modern systems do more
 - Esp. keep more detailed statistical information on data values
 - e.g., histograms

Size Estimation and Selectivity

- Max output cardinality = product of input cardinalities
- **Selectivity (sel)** associated with each **term**
 - reflects the impact of the term in reducing result size.
 - $\text{selectivity} = |\text{output}| / |\text{input}|$
 - Book calls selectivity “Reduction Factor” (RF)
- Avoid confusion:
 - “highly selective” in common English is opposite of a high selectivity value ($|\text{output}|/|\text{input}|$ high!)

```
SELECT  attribute list
FROM    relation list
WHERE   term1 AND ... AND termk
```

Result Size Estimation

- Result cardinality = Max # tuples * **product** of all selectivities.
- Term col=value (given Nkeys(l) on col)
 - $sel = 1/NKeys(l)$
- Term col1=col2 (handy for joins too...)
 - $sel = 1/MAX(NKeys(l1), NKeys(l2))$
 - Why MAX? See bunnies in 2 slides...
- Term col>value
 - $sel = (High(l)-value)/(High(l)-Low(l) + 1)$
- Note, if missing the needed stats, assume 1/10!!!

Let's dig into selectivity estimation more deeply

- Clarify how some of these estimates came to be
- Refine our stored statistics
- Expose our statistical assumptions

$P(\text{leftEar} = \text{rightEar})$

- 100 bunnies
- 2 distinct LeftEar colors
 - {C1, C2}
- 10 distinct RightEar colors
 - {C1..C10}
- Independent ears
- What's the probability of matching ears?



$$\begin{aligned}P(L = R) &= \sum_i P(C_i, C_i) \\&= P(C_1, C_1) + P(C_2, C_2) + P(C_3, C_3) + \dots \\&= \left(\frac{1}{2} \cdot \frac{1}{10}\right) + \left(\frac{1}{2} \cdot \frac{1}{10}\right) + \left(0 \cdot \frac{1}{10}\right) + \dots \\&= 1/10 \\&= 1/\text{MAX}(2, 10)\end{aligned}$$

Postgres 10.0: src/include/utils/selffuncs.h

```
/* default selectivity estimate for equalities such as "A = b" */
#define DEFAULT_EQ_SEL 0.005

/* default selectivity estimate for inequalities such as "A < b" */
#define DEFAULT_INEQ_SEL 0.3333333333333333

/* default selectivity estimate for range inequalities "A > b AND A < c" */
#define DEFAULT_RANGE_INEQ_SEL 0.005

/* default selectivity estimate for pattern-match operators such as LIKE */
#define DEFAULT_MATCH_SEL 0.005

/* default number of distinct values in a table */
#define DEFAULT_NUM_DISTINCT 200

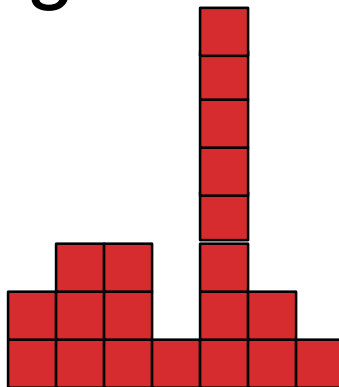
/* default selectivity estimate for boolean and null test nodes */
#define DEFAULT_UNK_SEL 0.005
#define DEFAULT_NOT_UNK_SEL (1.0 - DEFAULT_UNK_SEL)
```

Reduction Factors & Histograms

- For better estimation, use a histogram

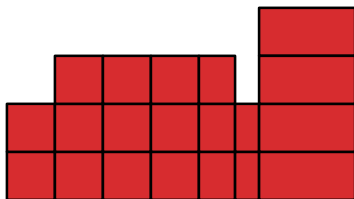
equiwidth

No. of Values	2	3	3	1	8	2	1
Value	0-.99	1-1.99	2-2.99	3-3.99	4-4.99	5-5.99	6-6.99



equidepth

No. of Values	2	3	3	3	3	2	4
Value	0-.99	1-1.99	2-2.99	3-4.05	4.06-4.67	4.68-4.99	5-6.99



Note: 10-bucket equidepth histogram divides the data into *deciles*

- akin to quantiles, median, etc.

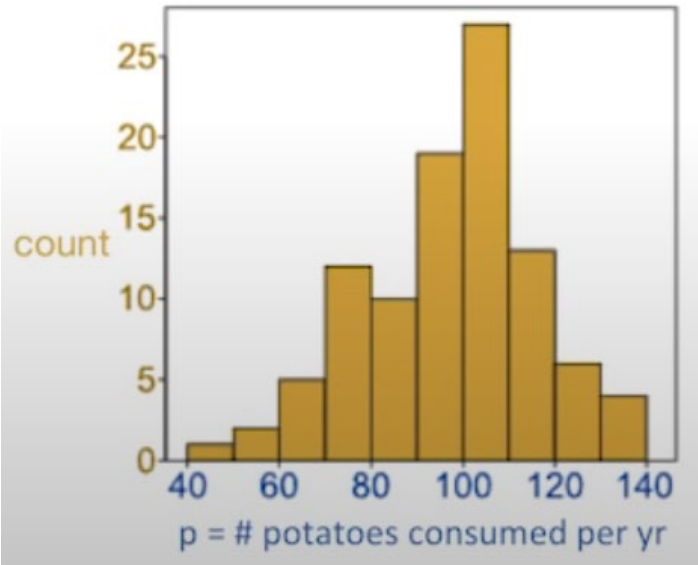
Common trick: “end-biased” histogram

- very frequent values in their own buckets

See also [V-Optimal histograms](#) on Wikipedia

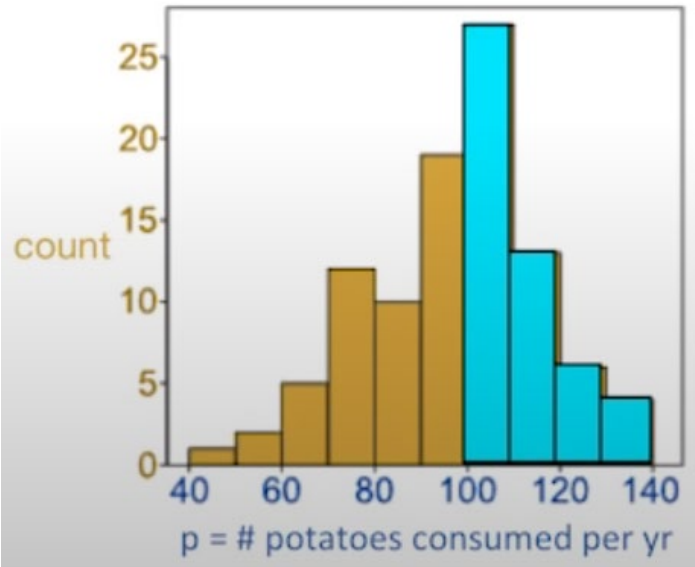
Computing selectivity with histograms

- 100 rows
- $\sigma_p > 99$?



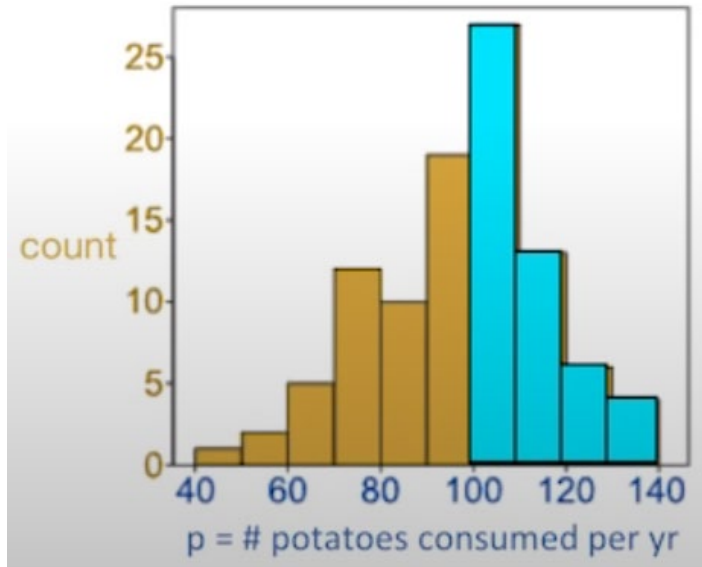
Computing selectivity with histograms, Pt 2

- 100 rows
- $\sigma_p > 99$?



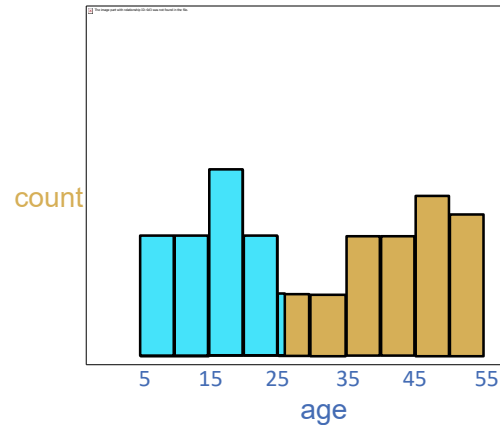
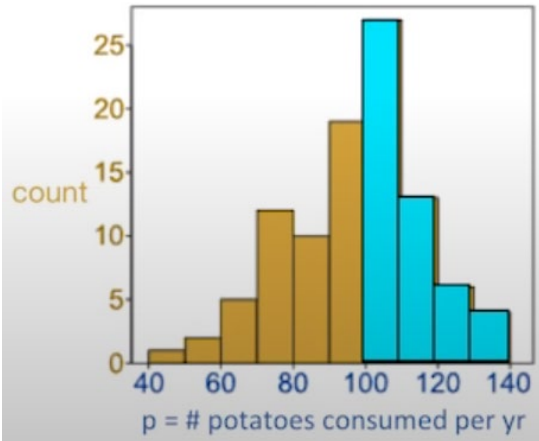
Computing selectivity with histograms, Pt 3

- 100 rows
- $\sigma_p >_{99}$? $50/100 = 50\%$.



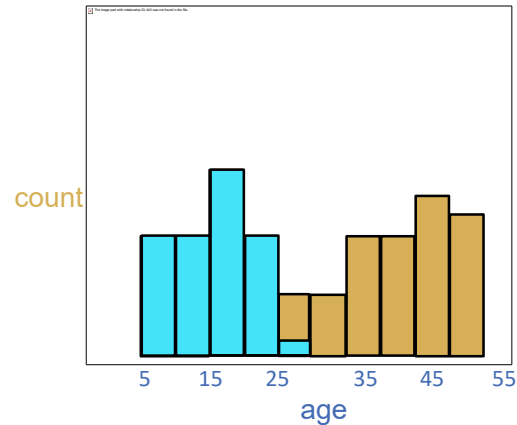
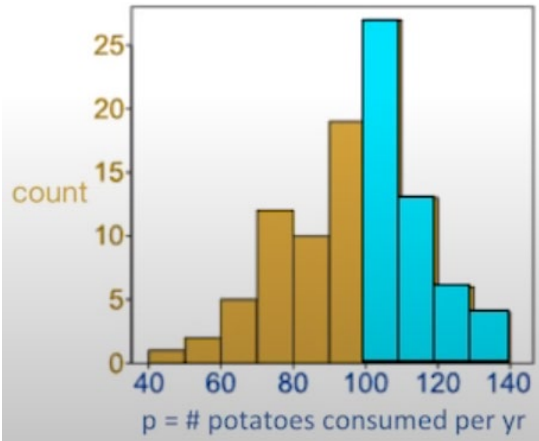
Computing selectivity with histograms, Pt 4

- 100 rows
- $\sigma_{\text{age} < 26}$?



Computing selectivity with histograms, Pt 5

- 100 rows
- $\sigma_{\text{age} < 26}$?



Computing selectivity with histograms, Part 6

- 100 rows
- $\sigma_{\text{age} < 26}$?

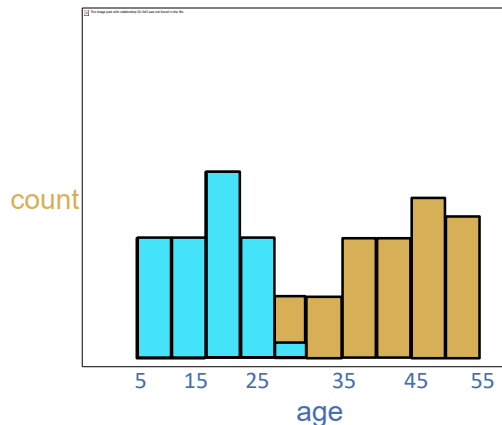
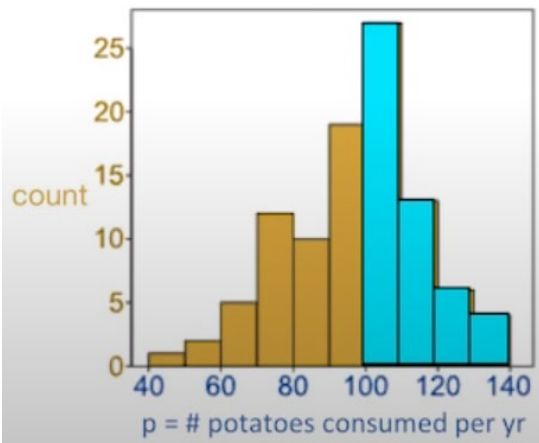
- **Uniformity assumption:**

Uniform distribution within each bin

Each vertical slice the same

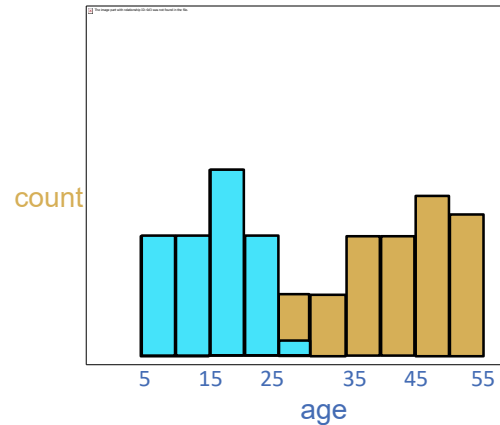
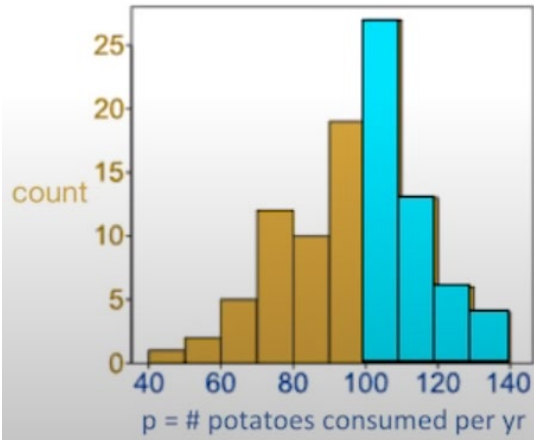
Hence $\frac{1}{5}$ of the population of bin [25,30) has age < 26.

$$10 + 10 + 15 + 10 + (\frac{1}{5} * 5) = 46/100 = \mathbf{46\%}$$



Selectivity of Conjunction

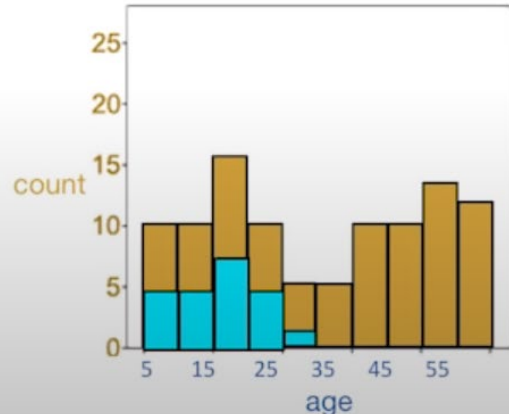
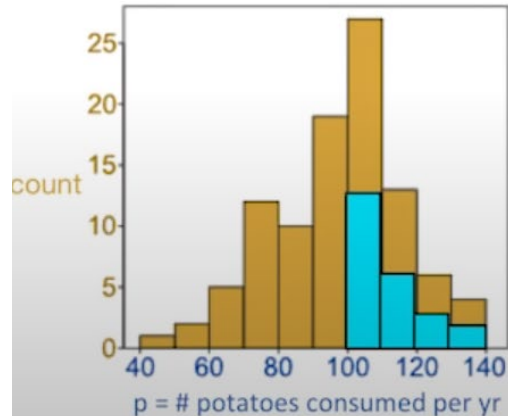
- 100 rows
- $\sigma_p > 99 \wedge \text{age} < 26$?
50% 46%



Selectivity of Conjunction, cont

- 100 rows
- $\sigma_p > 99 \wedge \text{age} < 26$?
50% 46%

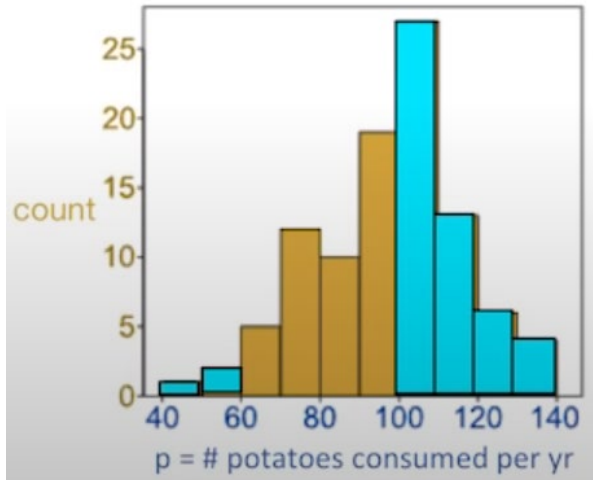
- **Independence assumption:**
 - Age and potato consumption are independent
 - Hence p bins all shrink by 46%.
 - Hence age bins all shrink by 50%.



Selectivity: $50\% \times 46\% = 23\%$

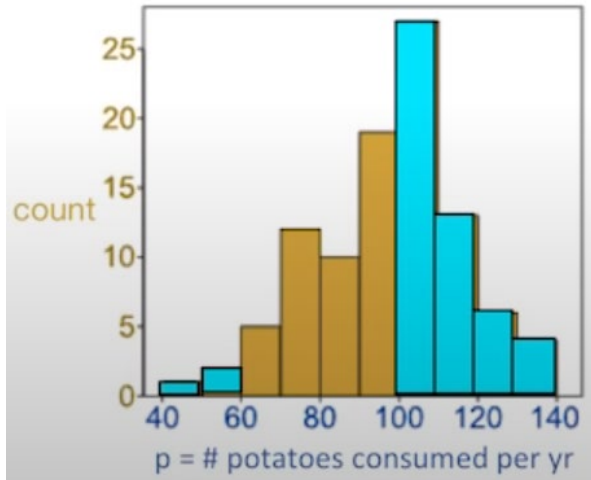
Selectivity of Disjunction

- 100 rows
- $\sigma_p > 99 \vee p < 60$?
50% 3%



Selectivity of Disjunction, Part 2

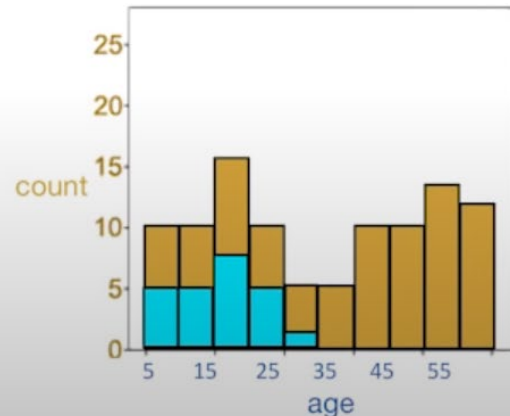
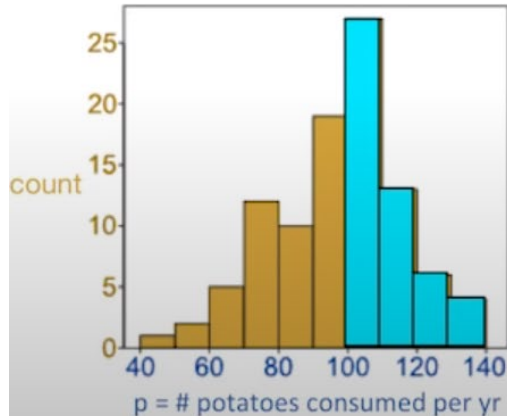
- 100 rows
- $\sigma_p > 99 \vee p < 60$?
- 50% 3%
- Selectivity: 50% + 3% = 53%



Selectivity of Disjunction, Part 3

- 100 rows
- $\sigma_{p > 99 \vee \text{age} < 26}$?
50% 46%

- Answer tuples satisfy one or both predicates
- By independence assumption:
 - Satisfy the first predicate: 50%
 - Satisfy the second predicate: 46%
 - Satisfy both: 50% × 46%
 - **Don't double-count!**



Selectivity:
 $50\% + 46\% - (50\% \times 46\%) = 73\%$

Selectivity for more complicated queries?

- $R \bowtie_p \sigma_q(S)$
 - Selectivity of join predicate p is s_p
 - Selectivity of selection predicate q is s_q
 - How to think about overall selectivity?

Join Selectivity

- Recall from algebraic equivalences: $R \bowtie_p S \equiv \sigma_p(R \times S)$
- Hence join selectivity is “just” selectivity s_p
 - Over a big input: $|R| \times |S|$
- Total rows: $s_p \times |R| \times |S|$

Selectivity for our earlier query?

- Recall from algebraic equivalences

$$R \bowtie_p \sigma_q(S) \equiv \sigma_p(R \times \sigma_q(S)) \equiv \sigma_{p \wedge q}(R \times S)$$

- Hence selectivity just $s_p s_q$
 - Applied to $|R| \times |S|$!
- Total rows: $s_p s_q |R| |S|$

Column Equality?

T.p = T.age ??

Intuition: similar to bunny ears, but weighted by the histogram bins.

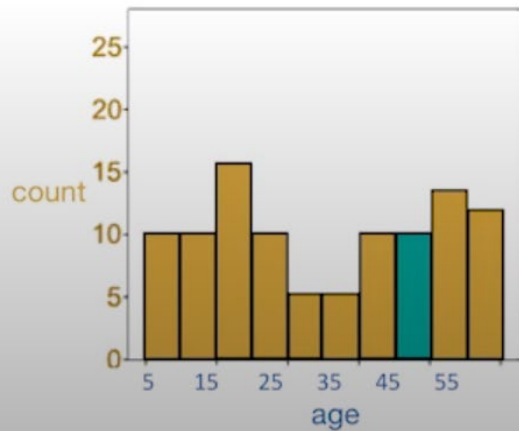
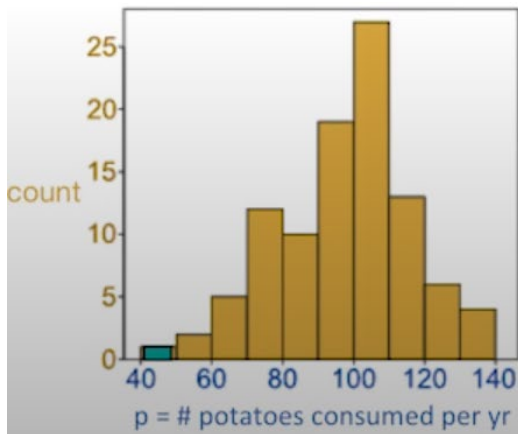
s = 0

For each value v covered in either histogram:

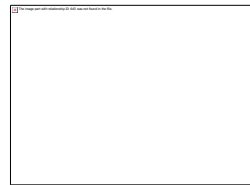
// uniformity assumption within bins:

// $P(T.p = v) = \text{height}(\text{binp}(v))/n * 1/\text{width}(\text{binp}(v))$

// $P(T.age = v) = \text{height}(\text{binage}(v))/n * 1/\text{width}(\text{binage}(v))$



Column Equality?



T.p = T.age ??

Intuition: similar to bunny ears, but weighted by the histogram bins.

s = 0

For each value v covered in either histogram:

```
// uniformity assumption within bins:
```

```
// P(T.p = v) = height(binp(v))/n * 1/width(binp(v))
```

```
// P(T.age = v) = height(binage(v))/n * 1/width(binage(v))
```

```
// independence assumption across columns:
```

```
// P(T.p = v ∧ T.age = v)
```

```
// = P(T.p = v) * P(T.age = v)
```

```
s += height(binp(v))/(n*width(binp(v)))  
    * height(binage(v))/(n*width(binage(v)))
```

Challenge: make this more efficient by iterating over bin boundaries rather than values!

Upshot

- Know how to compute selectivities for basic predicates
 - The original Selinger version
 - The histogram version
- Assumption 1: uniform distribution within histogram bins
 - Within a bin, fraction of range = fraction of count
- Assumption 2: independent predicates
 - Selectivity of AND = product of selectivities of predicates
 - Selectivity of OR = sum of selectivities of predicates - product of selectivities of predicates
 - Selectivity of NOT = 1 – selectivity of predicates
- Joins are not a special case
 - Simply compute the selectivity of all predicates
 - And multiply by the product of the table sizes



Query Optimization

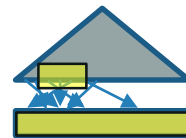
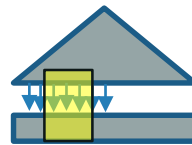
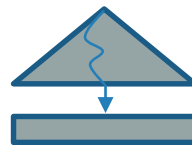
1. Plan Space
2. Cost Estimation
3. Search Algorithm

Enumeration of Alternative Plans

- There are two main cases:
 - **Single-table plans** (base case)
 - **Multiple-table plans** (induction)
- Single-table queries include selects, projects, and groupBy/agg:
 - Consider each available access path (file scan / index)
 - Choose the one with the least estimated cost
 - Selection/Projection done on the fly
 - Result pipelined into grouping/aggregation

Cost Estimates for Single-Relation Plans

- Index I on primary key matches selection:
 - Cost is $(\text{Height}(I) + 1) + 1$ for a B+ tree.
- Clustered index I matching selection:
 - $(\text{NPages}(I) + \mathbf{NPages}(R)) * \text{selectivity}$.
- Non-clustered index I matching selection:
 - $(\text{NPages}(I) + \mathbf{NTuples}(R)) * \text{selectivity}$.
- Sequential scan of file:
 - $\text{NPages}(R)$.
- Recall: Must also charge for duplicate elimination if required



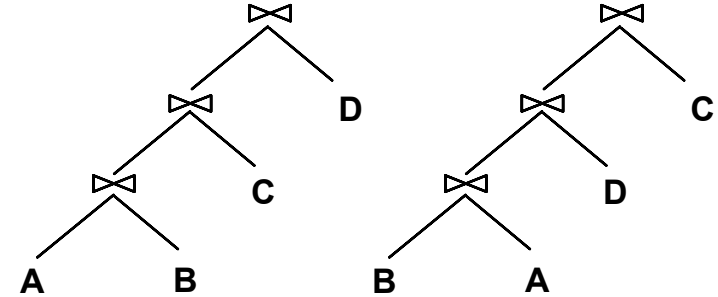
Example

```
SELECT S.sid  
FROM Sailors S  
WHERE S.rating=8
```

- If we have an index on rating:
 - **Cardinality** = $(1/NKeys(I)) * NTuples(R) = (1/10) * 40000$ tuples
 - **Clustered index:** $(1/NKeys(I)) * (NPages(I) + NPages(R))$
 $= (1/10) * (50 + 500) = \mathbf{55 \text{ pages are retrieved.}}$ (This is the cost.)
 - **Unclustered index:** $(1/NKeys(I)) * (NPages(I) + NTuples(R))$
 $= (1/10) * (50 + 40000) = \mathbf{4005 \text{ pages are retrieved.}}$
- If we have an index on sid:
 - Would have to retrieve all tuples/pages. With a clustered index, the cost is 50+500, with unclustered index, 50+40000.
- Doing a file scan:
 - We retrieve all file pages (500).

Enumeration of Left-Deep Plans

- Left-deep plans differ in
 - the order of relations
 - the access method for each leaf operator
 - the join method for each join operator
- Enumerated using N passes (if N relations joined):
 - **Pass 1:** Find best 1-relation plan for each relation
 - **Pass i:** Find best way to join result of an $(i - 1)$ -relation plan (as outer) to the i' th relation. (i between 2 and N.)
- For each subset of relations, retain only:
 - Cheapest plan overall, plus
 - Cheapest plan for each *interesting order* of the tuples.



The Principle of Optimality

- Richard Bellman (slightly adapted to our setting)
- The best overall plan is composed of best decisions on the subplans
 - Optimal result has optimal substructure
- For example, the best left-deep plan to join tables A, B, C is either:
 - (The best plan for joining A, B) \bowtie C
 - (The best plan for joining A, C) \bowtie B
 - (The best plan for joining B, C) \bowtie A
- This is great!
 - When optimizing a subplan (e.g. A \bowtie B), we don't have to think about how it will be used later (e.g. when dealing with C)!
 - When optimizing a higher-level plan (e.g. A \bowtie B \bowtie C) we can reuse the best results of subroutines (e.g. A \bowtie B)!



{A, B}

Dynamic Programming Algorithm for System R

- Principle of optimality allows us to build best subplans “bottom up”
 - Pass 1: Find best plans of height 1 (base table accesses), and record them in a table
 - Pass 2: Find best plans of height 2 (joins of base tables) by combining plans of height 1, record them in a table
 - ...
 - Pass i : Find best plans of height i by combining plans of height $i - 1$ with plans of height 1, record them in a table
 - ...
 - Pass n : Find best plan overall by combining plans of height $n-1$ with plans of height 1.



The Basic Dynamic Programming Table

Table keyed on
1st column

<u>Subset of tables in FROM clause</u>	Best plan	Cost
{R, S}	hashjoin(R,S)	1000
{R, T}	mergejoin(R,T)	700

A Wrinkle: Interesting Orders

- Physical properties can break the principle of optimality
 - For example, consider a suboptimal plan p for $A \bowtie B$ that is ordered on column x
 - Suppose we need to join with table C on column x
 - Sort-merge of p with C might be the best overall plan
 - The best plan for $A \bowtie B$ requires us to sort for Sort-Merge join
 - But the suboptimal plan p doesn't require us to sort $A \bowtie B$
- Solution: expand our definition of “optimal substructure”
 - The structure will include both the set of tables and the physical properties (order)
 - But not all orders are “interesting”! We can prune further

A Note on “Interesting Orders”

- Physical property: Order.
When should we care? When is it “interesting”?
- An intermediate result has an “interesting order” if it is sorted by anything we can use later in the query (“downstream” the arrows):
 - ORDER BY attributes
 - GROUP BY attributes
 - Join attributes of yet-to-be-added joins
 - subsequent merge join might be good

The Dynamic Programming Table

Table keyed on
concatenation of 1st
two columns

<u>Subset of tables in FROM clause</u>	<u>Interesting- order columns</u>	Best plan	Cost
{R, S}	<none>	hashjoin(R,S)	1000
{R, S}	<R.a, S.b>	sortmerge(R,S)	1500

Enumeration of Plans (Contd.)

- First figure out the scans and joins (select-project-join) using D.P.
 - **Avoid Cartesian Products** in dynamic programming as follows:
When matching an $i - 1$ way subplan with another table, only consider it if
 - There is a join condition between them, **or**
 - All predicates in WHERE have been “used up” in the $i - 1$ way subplan.
- Then handle ORDER BY, GROUP BY, aggregates etc. as a post-processing step
 - Via “interestingly ordered” plan if chosen (free!)
 - Or via an additional sort/hash operator
- Despite pruning, this System R D.P. algorithm is **exponential** in #tables.

Example

```
SELECT S.sid, COUNT(*) AS number
FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid
AND R.bid = B.bid
AND B.color = "red"
GROUP BY S.sid
```

Sailors:

Hash, B+ tree indexes on *sid*

Reserves:

Clustered B+ tree on *bid*

B+ on *sid*

Boats

B+ on *color*

Pass 1: Best plan(s) for each relation

- Sailors, Reserves: File Scan
- Also B+ tree on Reserves.bid as interesting order
- Also B+ tree on Sailors.sid as interesting order
- Boats: B+ tree on color

<u>Subset of tables in FROM clause</u>	<u>Interesting- order columns</u>	Best plan	Cost
{Sailors}	--	filescan	
{Reserves}	--	Filescan	
{Boats}	--	B-tree on color	
{Reserves}	(bid)	B-tree on bid	
{Sailors}	(sid)	B-tree on sid	

Pass 2

// for each left-deep logical plan

for each plan P in pass 1

for each FROM table T not in P

// for each physical plan

for each access method M on T

for each join method

generate $P \bowtie M(T)$

- File Scan Reserves (outer) with Boats (inner)
- File Scan Reserves (outer) with Sailors (inner)
- Reserves Btree on bid (outer) with Boats (inner)
- Reserves Btree on bid (outer) with Sailors (inner)
- File Scan Sailors (outer) with Boats (inner)
- File Scan Sailors (outer) with Reserves (inner)
- Boats Btree on color with Sailors (inner)
- Boats Btree on color with Reserves (inner)
- Retain cheapest plan for each (pair of relations, order)

<u>Subset of tables in FROM clause</u>	<u>Interesting- order columns</u>	Best plan	Cost
{Sailors}	--	filescan	
{Reserves}	--	Filescan	
{Boats}	--	B-tree on color	
{Reserves}	(bid)	B-tree on bid	
{Sailors}	(sid)	B-tree on sid	
{Boats, Reserves}	(B.bid) (R.bid)	SortMerge(B-tree on Boats.color, filescan Reserves)	
Etc...			

Pass 3 and beyond

- Using **Pass 2 plans** as outer relations, generate plans for the next join in the same way as Pass 2
 - E.g. **{SortMerge(B-tree on Boats.color, filescan Reserves)}** (outer) |
with Sailors (B-tree sid) (inner)
- Then, add cost for groupby/aggregate:
 - This is the cost to sort the result by sid, *unless it has already been sorted by a previous operator.*
- Then, choose the cheapest plan

```
SELECT S.sid, COUNT(*) AS number
FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid
AND R.bid = B.bid
AND B.color = "red"
GROUP BY S.sid
```

Now you understand the optimizer!

So what?!

- Benefit #1: You could build one.
 - And you will!
- Benefit #2: You can influence one
 - People who write non-trivial SQL often get frustrated with the optimizer
 - It picked a crummy plan!
 - It didn't use the index I built!
 - Etc.
 - Understanding the optimizer can lead you to:
 - Design your DB & Indexes better
 - Avoid “weak spots” in your optimizer's implementation
 - Coax your optimizer to do what you want