

Relational Query Optimization II: Costing and Searching

Alvin Cheung
Fall 2022

Reading: R & G Chapter 15



Announcements



- Project 3
- Edstem etiquette
- Final date: Dec 14 11:30am-2:30pm

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

A Naïve Query Optimizer



- Given an input query Q:
 1. Enumerate all possible plans for Q
 - Too many plans to consider!
 2. Estimate the cost of each plan
 - Hard to estimate cost accurately given caches etc
 3. Pick plan with the lowest cost
 - How? Keep all plans in memory?

Query plan space

```
Select o_year,  
sum(case  
when nation = 'BRAZIL' then volume  
else 0  
end) / sum(volume)  
from  
(  
select YEAR(O_ORDERDATE) as o_year,  
L_EXTENDEDPRICE * (1 - L_DISCOUNT) as volume,  
n2.N_NAME as nation  
from PART, SUPPLIER, LINEITEM, ORDERS, CUSTOMER, NATION n1,  
NATION n2, REGION  
where  
P_PARTKEY = L_PARTKEY and S_SUPPKEY = L_SUPPKEY  
and L_ORDERKEY = O_ORDERKEY and O_CUSTKEY = C_CUSTKEY  
and C_NATIONKEY = n1.N_NATIONKEY and n1.N_REGIONKEY = R_REGIONKEY  
and R_NAME = 'AMERICA' and S_NATIONKEY = n2.N_NATIONKEY  
and O_ORDERDATE between '1995-01-01' and '1996-12-31'  
and P_TYPE = 'ECONOMY ANODIZED STEEL'  
and S_ACCTBAL <= constant-1  
and L_EXTENDEDPRICE <= constant-2  
) as all_nations  
group by o_year order by o_year
```



There about 22 million
alternative ways of executing
this query!



Slide from D Dewitt

Big Picture of System R Optimizer



- Plan Space
 - Many plans have the same high cost subtree that can be pruned
 - Heuristics (aka tricks that usually work):
 - Consider only left-deep plans
 - Avoid Cartesian products
 - Don't optimize the entire query at once
- Cost estimation
 - Inexact is fine as long as we can compare plans
 - Better estimators have been developed
- Search Algorithm
 - Dynamic Programming

Query Optimization



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

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”
 - Recall relational algebra lecture

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

Outer block

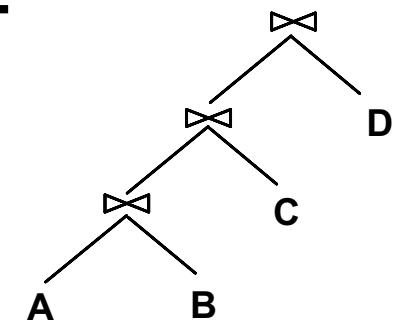
```
(SELECT MAX (S2.age)  
      FROM   Sailors S2  
     GROUP BY S2.rating)
```

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

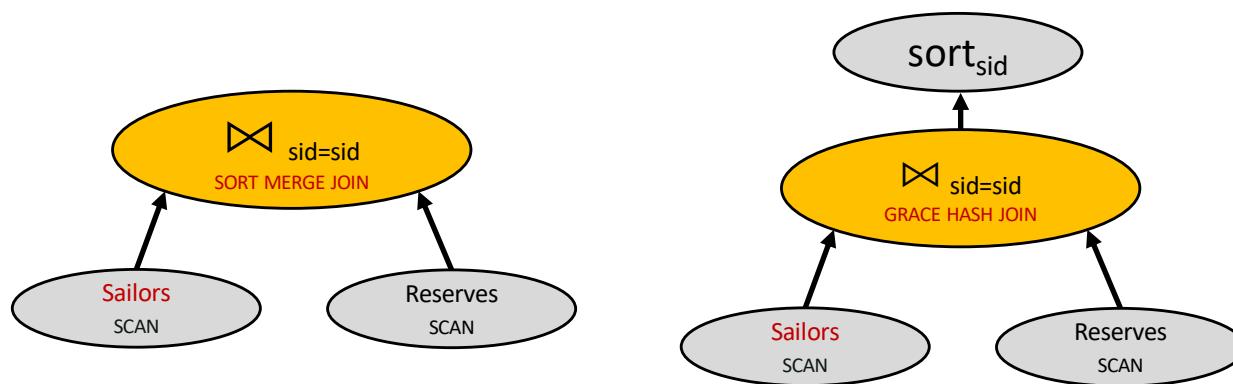


- Two common “physical” properties of an output:
 - Sort order
 - Hash Grouping
- Certain operators produce these properties in output
 - E.g. Index scan (result is sorted)
 - E.g. Sort (result is sorted)
 - E.g. Hash (result is grouped)
- Certain operators require these properties at input
 - E.g. MergeJoin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

Recall: Physically Equivalent Plans



- Same content and same physical properties

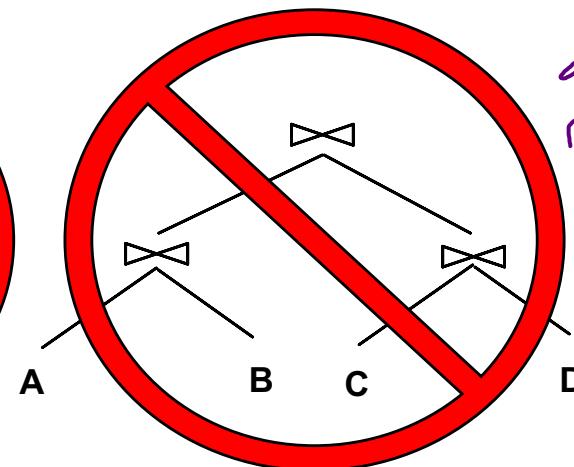
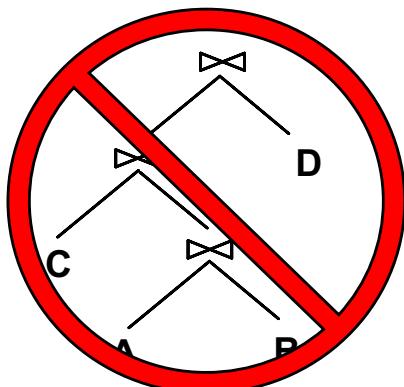
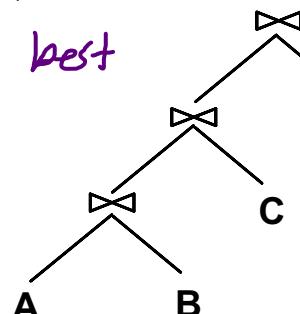


Queries Over Multiple Relations



- 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.
 - i.e., intermediate results not written to temporary files.
 - Not all left-deep trees are fully pipelined (e.g., SM join).

might not
be best
plan



data from
right tables always
available to be
streamed

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 prune this space by:
 - 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

More than counting I/Os



- We can no longer ignore the cost of computation!
- 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 $\#I/O + CPU\text{-factor} * \#tuples$
 - Second term estimate the cost of tuple processing

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 even more
 - Especially 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)
 - Always between 0 and 1 ex:  \Rightarrow \text{sel} = \frac{1}{3}
- 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



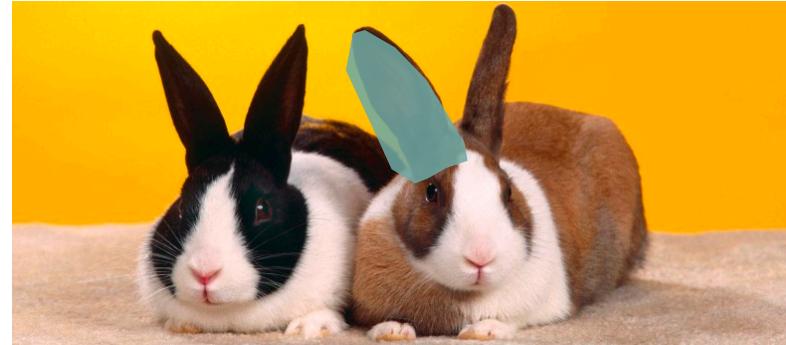
- Result cardinality = Max # tuples * **product** of all selectivities.
- Term col=value (given Nkeys(col) unique values of col)
 - $\text{sel} = 1/\text{NKeys}(\text{col})$
- Term col1=col2 (handy for joins...)
 - $\text{sel} = 1/\text{MAX}(\text{NKeys}(\text{col1}), \text{NKeys}(\text{col2}))$
 - Why MAX?
- Term col>value
 - $\text{sel} = (\text{High}(\text{col})-\text{value})/(\text{High}(\text{col})-\text{Low}(\text{col}) + 1)$
- If no info about actual range of values, assume 1/10!!!



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



- 100 bunnies
- 2 distinct LeftEar colors
 - $\{C_1, C_2\}$
- 10 distinct RightEar colors
 - $\{C_1, C_2, \dots, C_{10}\}$
- Independent ears
- What's the probability of matching ears?



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

Postgres 10.0: src/include/utils/selfuncs.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



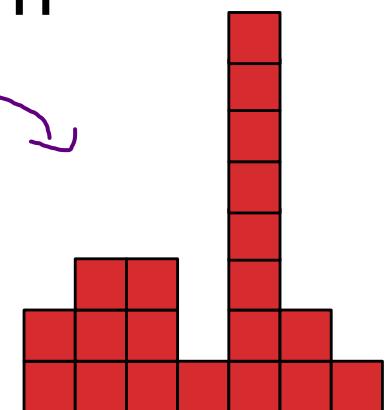
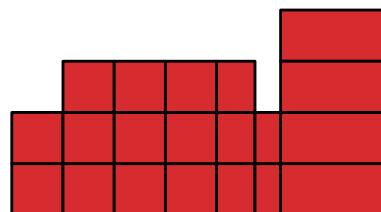
- For better estimation, use a histogram

equiwidth

# values	2	3	3	1	8	2	1
Value	0-0.99	1-1.99	2-2.99	3-3.99	4-4.99	5-5.99	6-6.99

equidepth

# values	2	3	3	3	3	2	4
Value	0-0.99	1-1.99	2-2.99	3-4.05	4.06-4.67	4.68-4.99	5-6.99

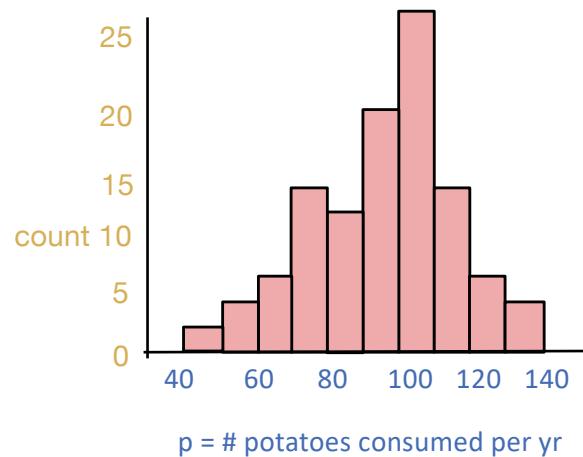


Note: 10-bucket equidepth histogram divides the data into *deciles*
- akin to quantiles, median, etc.

Computing selectivity with histograms



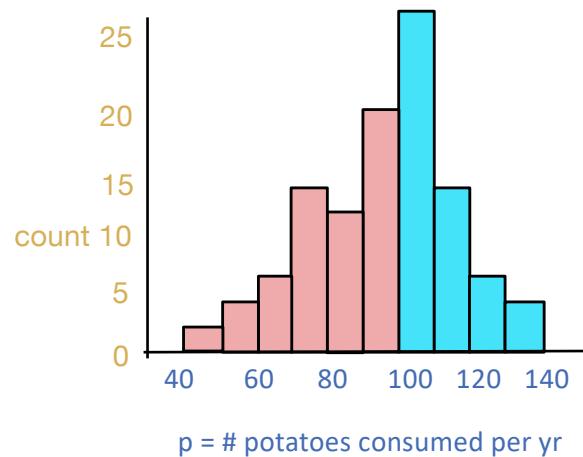
- 100 rows
- $\sigma_p > 99?$



Computing selectivity with histograms



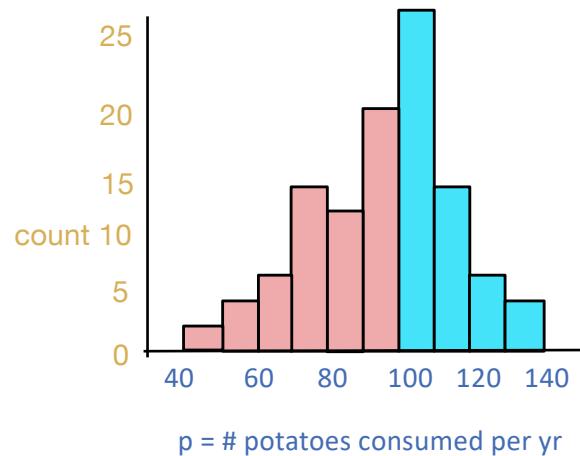
- 100 rows
- $\sigma_p > 99?$



Computing selectivity with histograms



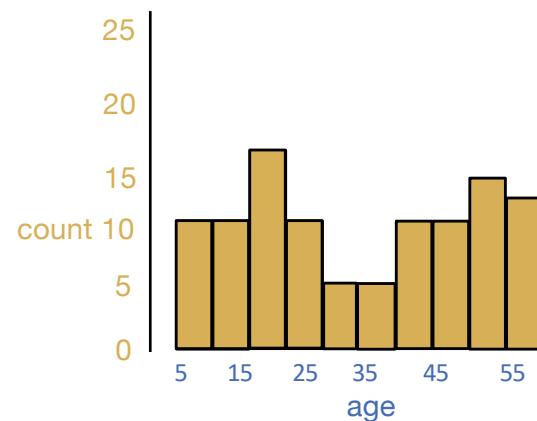
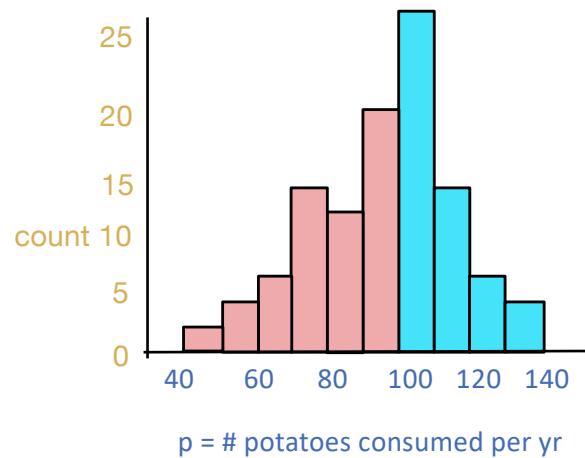
- 100 rows
- $\sigma_{p > 99}?$ $50/100 = 50\%$.



Computing selectivity with histograms



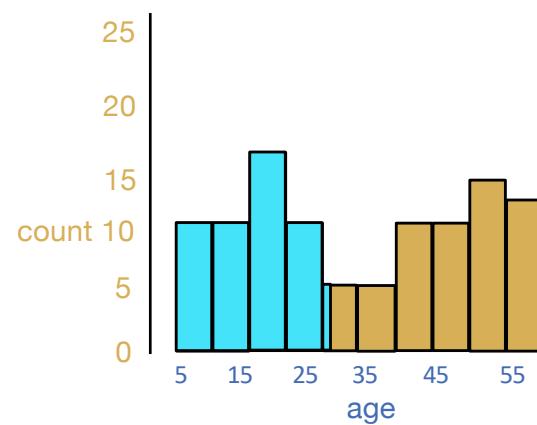
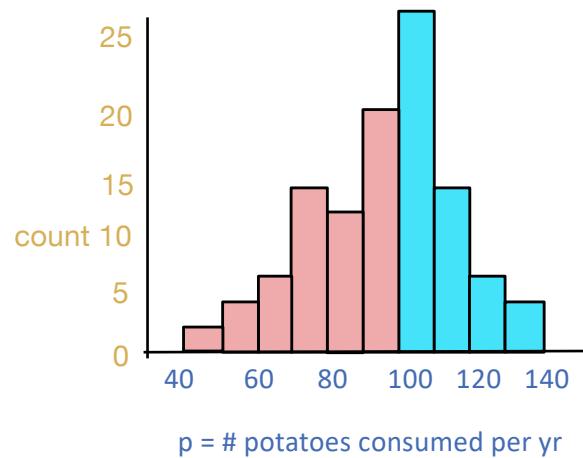
- 100 rows
- $\sigma_{age < 26}?$



Computing selectivity with histograms



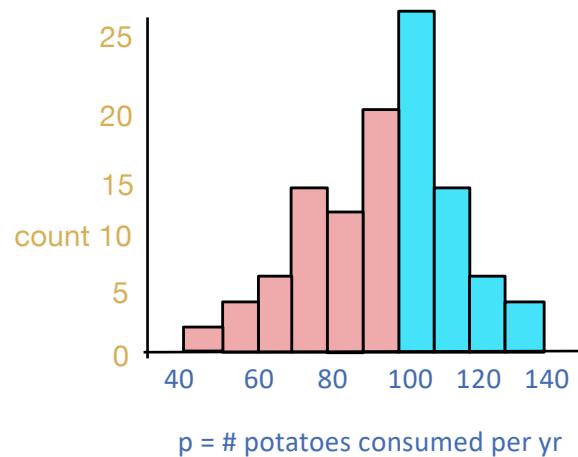
- 100 rows
- $\sigma_{age < 26}?$



Computing selectivity with histograms



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



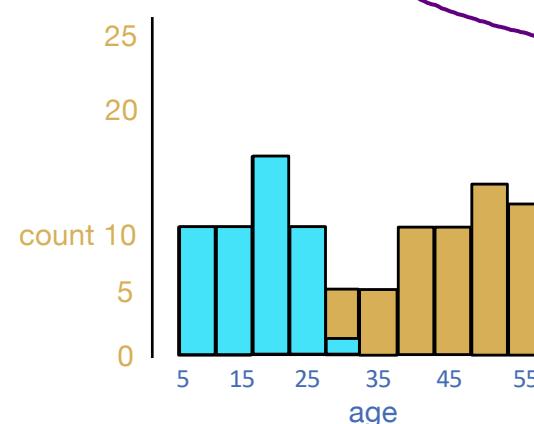
- **Uniformity assumption:**

Uniform distribution within each bin

Each vertical slice the same

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

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



Range = [25, 30]

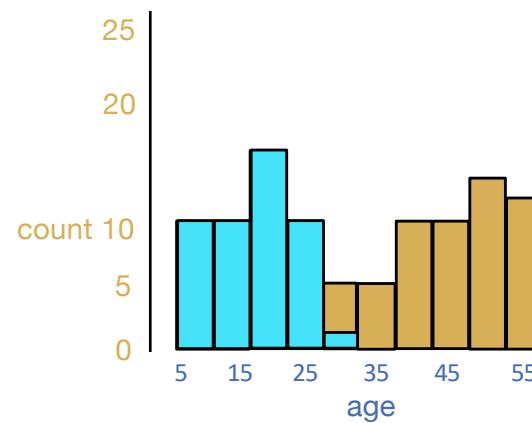
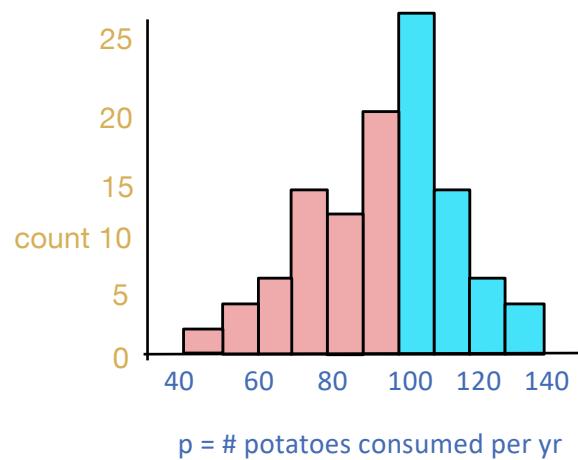
$$\frac{1}{5} = \frac{26-25}{30-25}$$

Selectivity of Conjunction



- 100 rows
- $\sigma_{p > 99 \wedge \text{age} < 26}$?

50% 46%

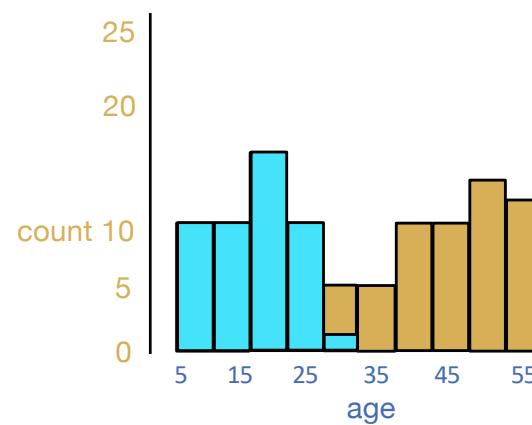
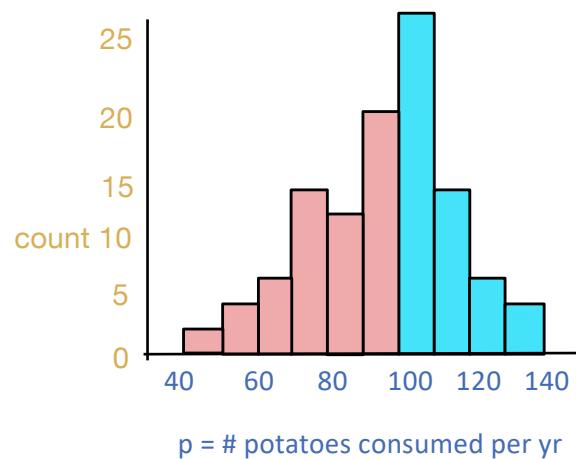


- **Independence assumption:**
 - Age and potato consumption are independent
 - Selectivity: $\underline{50\%} \times \underline{46\%} = \underline{\underline{23\%}}$

Selectivity of Disjunction



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



- Answer tuples satisfy one or both predicates
- By independence assumption:
 - Satisfy the first predicate: 50%
 - Satisfy the second predicate: 46%
 - Satisfy both: $50\% \times 46\%$
 - **Don't double-count!**
- Selectivity:
$$50\% + 46\% - (50\% \times 46\%) = \underline{\underline{73\%}}$$

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 algebraic equivalence: $R \bowtie_p S \equiv \sigma_p(R \times S)$
- 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

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

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

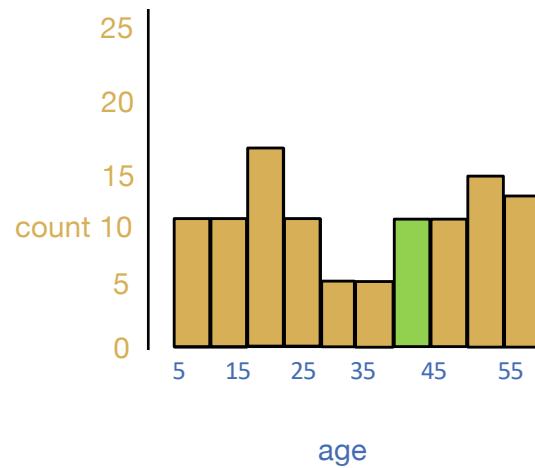
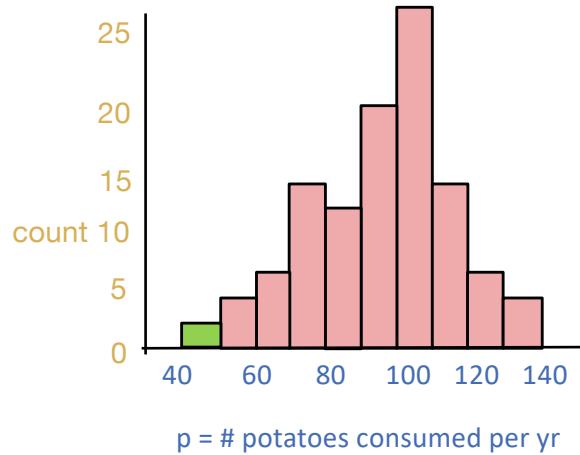
- Total rows: $s_p s_q |R| |S|$

Column Equality?



$T.p = T.age ??$

Idea: scan over all values of p and age, and check when they are equal

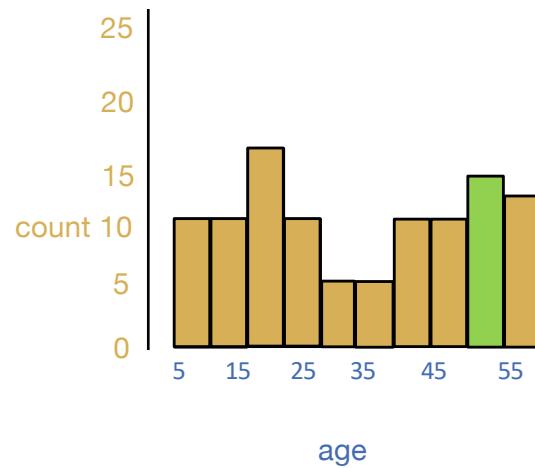
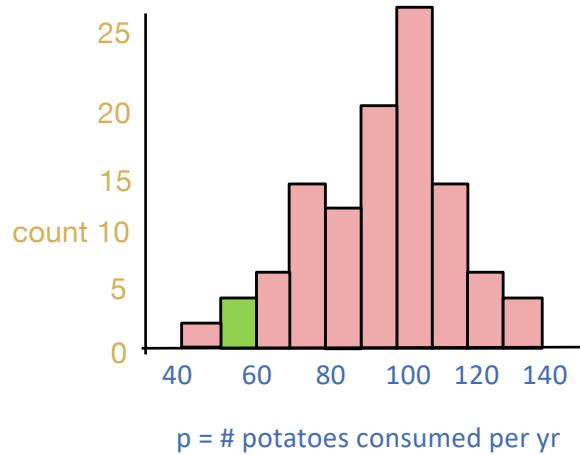


Column Equality?



$T.p = T.age ??$

Idea: scan over all values of p and age, and check when they are equal



Column Equality?

$T.p = T.age ??$

Idea: scan over all values of p and age, and check when they are equal



$T.p = T.age$

$= (T.p = 40 \wedge T.age = 40) \vee (T.p = 41 \wedge T.age = 41) \vee (T.p = 42 \wedge T.age = 42) \dots$

$= (T.p = 40 \wedge T.age = 40) + (T.p = 41 \wedge T.age = 41) + (T.p = 42 \wedge T.age = 42) \dots$

$= (T.p = 40 * T.age = 40) + (T.p = 41 * T.age = 41) + (T.p = 42 * T.age = 42) \dots$

Independence assumption

$(T.p = 40)$

$$= \frac{\text{height}(\text{binp}(40))}{\text{width}(\text{binp}(40)) * \underbrace{\text{np}}_{\text{bin range}}}$$

$(T.age = 40)$

$$= \frac{\text{height}(\text{binage}(40))}{\text{width}(\text{binage}(40)) * \underbrace{\text{nage}}_{\text{bin range}}}$$

Uniform assumption

Just add up all the values...

What you need to know



- 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

$$\boxed{\text{ }} = \boxed{\text{ }}$$

$$\sigma \Delta_p b \\ = s_p |a| |b|, \quad s_p = \text{selectivity of } \sigma_p$$

Summary: Selectivity Estimation



- We need a way to estimate the size of the intermediate tables
 - Recall cost of each operator = I/Os (to bring in input) + *CPU-factor* * # tuples processed
- Output size = input size * operator selectivity

Summary: Selectivity Estimation



System R

- col=value
 - $1/\text{uniq-keys}(\text{col})$
- col1=col2
 - $1/\text{MAX}(\text{uniq-keys}(\text{col1}), \text{uniq-keys}(\text{col2}))$
- col>value
$$\frac{\text{High}(\text{col}) - \text{value}}{\text{High}(\text{col}) - \text{Low}(\text{col}) + 1}$$

Histogram

- col=value
$$\frac{\text{bar height containing value}}{(\# \text{ values contained in bar}) * \# \text{ rows}}$$
- col1=col2
 - Breakdown into
$$(\text{col1} = v1 \wedge \text{col2} = v1) \vee (\text{col1} = v2 \wedge \text{col2} = v2) \vee \dots$$
- col>value
$$\frac{\text{sum of bar heights } >\text{value}}{\text{total number of rows}}$$

See discussion for floating-point-valued columns!

Summary: Selectivity Estimation



- In both cases, for more complex predicates:
 - $p_1 \wedge p_2$
 - $\text{selectivity}(p_1) * \text{selectivity}(p_2)$
 - $p_1 \vee p_2$
 - $\text{selectivity}(p_1) + \text{selectivity}(p_2) - (\text{selectivity}(p_1) * \text{selectivity}(p_2))$
 - Last term is 0 if p_1 and p_2 are non-overlapping
(e.g., $\text{age} > 60$ OR $\text{age} < 21$)
 - $\text{not } p_1 = 1 - \text{selectivity}(p_1)$
- This stems from our **independence assumption**

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



Recall: Cost of Operations Using Index

	Heap File	Sorted File	Clustered Index
Scan all records	$B*D$	$B*D$	$3/2 * B * D$
Equality Search	$0.5*B*D$	$(\log_2 B)*D$	$(\log_F(BR/E)+2)*D$
Range Search	$B*D$	$((\log_2 B)+\text{pages})*D$	$(\log_F(BR/E)+3*\text{pages})*D$
Insert	$2*D$	$((\log_2 B) + B)*D$	$(\log_F(BR/E)+4)*D$
Delete	$(0.5*B+1)*D$	$((\log_2 B) + B)*D$	$(\log_F(BR/E)+4)*D$

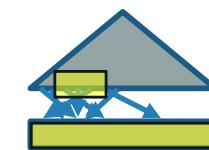
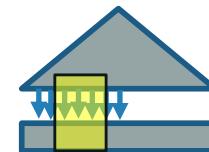
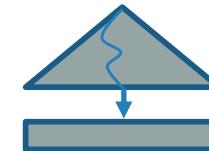
- **B:** Number of data blocks
- **R:** Number of records per block
- **D:** Average time to read/write disk block
- **F:** Average internal node fanout
- **E:** Average # data entries per leaf

Review

Cost Estimates for Single-Relation Plans

I/Os

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



Example

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.



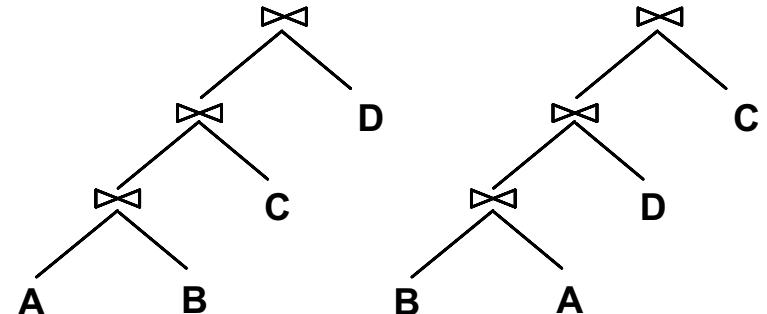
- If we have an index on rating:
 - **Cardinality** = $(1/\text{NKeys}(I)) * \text{NTuples}(R) = (1/10) * 40000$ tuples
 - **Clustered index:** $(1/\text{NKeys}(I)) * (\text{NPages}(I)+\text{NPages}(R)) = (1/10) * (50+500) = \textbf{55 pages are retrieved.}$
 - **Unclustered index:** $(1/\text{NKeys}(I)) * (\text{NPages}(I)+\text{NTuples}(R)) = (1/10) * (50+40000) = \textbf{4005 pages are retrieved.}$
 - (costs on indexes are approximate as we might not need to retrieve all index pages)
- If we have an index on sid:
 - Would have to (roughly) 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).

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

Joins: Enumeration of Left-Deep Plans



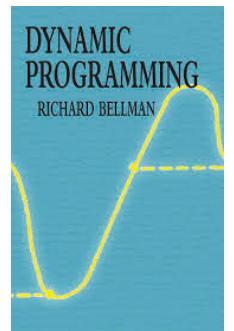
- Left-deep plans differ in
 - the order of relations
 - the access method for each leaf operator
 - the join method for each join operator
- Enumerated using N passes (if N relations are joined):
 - **Pass 1:** Find best 1-relation plan for each relation (*types of scan*)
 - **Pass i:** Find best way 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



- Bellman '57 (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)!



} Dynamic programming

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



Programming table is keyed on 1st column (tables in the FROM clause)

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

R

hash, 1000	None	None
None	merge, 2000	hash, 900
hash, 700	None	merge, 300

S

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 (i.e., “downstream” op):
 - ORDER BY attributes
 - GROUP BY attributes
 - Join attributes of yet-to-be-added joins
 - subsequent merge join might be good
- 
- A hand-drawn blue curly brace on the right side of the slide, spanning from the word "downstream" in the list to the word "downstream" in the explanatory text above. The text "downstream" is written in blue cursive.

The Dynamic Programming Table



Table keyed on concatenation of first 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

← Higher cost, but may lead to global optimal plan!

TINSTAFL!!



Enumeration of Plans



- First figure out the scans and joins (select-project-join) using dynamic programming
 - **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 dynamic programming 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:

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 (output sorted on bid)
- Also B+ tree on Reserves.sid as interesting order (output sorted on sid)
- Also B+ tree on Sailors.sid as interesting order (output sorted on sid)
- Boats: B+ tree on color as interesting order (output sorted on color)

Best plans after pass 1



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

Best plans after pass 2



<u>Subset of tables in FROM clause</u>	<u>Interesting-order columns</u>	Best plan	Cost
{Sailors}	n/a	filescan	...
{Reserves}	n/a	Filescan	...
{Boats}	n/a	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 group-by/aggregate:
 - This is the cost to sort the result by sid, *unless it has already been sorted by a previous operator.*
- Finally, choose the cheapest plan

Now you understand the optimizer!



- Benefit #1: You could build one.
 - And you will in project 3!
- 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