Query Optimization I: Plan Space

The Components

Three beautifully orthogonal concerns:

- Plan Space:
  - for a given query, what plans are considered?
- Cost Estimation:
  - How is the cost of a plan estimated?
- Search strategy
  - How do we "search" in the "plan space"?

The goal

- Optimization Goal:
  - Ideally: Find the plan with the least actual cost.
  - Reality: Find the plan with the least estimated cost.
    - And try to avoid really bad actual plans.

Relational Algebra Equivalences

- Selections:
  - $\sigma_{c1 \wedge \ldots \wedge cn}(R) \equiv \sigma_{c1}(\ldots(\sigma_{cn}(R))\ldots)$ (cascade)
  - $\sigma_{c1}(\sigma_{c2}(R)) \equiv \sigma_{c2}(\sigma_{c1}(R))$ (commute)
- Projections:
  - $\pi_{a1}(R) \equiv \pi_{a1}(\ldots(\pi_{a1,\ldots,an-1}(R))\ldots)$
    - Note: can cascade to a superset of columns of the original projection
- Cartesian Product
  - $R \times (S \times T) \equiv (R \times S) \times T$ (associative)
  - $R \times S \equiv S \times R$ (commutative)

Some Common Heuristics — Relational expressions

Selections

- Selection cascade and pushdown
  - Apply selections as soon as you have the relevant columns
  - Ex:
    - $\pi_{sname}(\sigma_{(bid=100 \wedge rating>5)}(Reserves \bowtie_{sid=sid} Sailors))$
    - $\pi_{sname}(\sigma_{bid=100}(Reserves) \bowtie_{sid=sid} \sigma_{rating>5}(Sailors))$
  - Explain: Applying selections as soon as you can will lower the size of the inputs to the joints and make joints cheaper.
  - Assumption: selection is cheap or even free and join is expensive and so we should always do selection before join projections

Projections

- Projection cascade and pushdown
  - Keep only the columns you need to evaluate downstream operators
  - Ex:
    - $\pi_{sname}\sigma_{bid=100 \wedge rating>5}(Reserves \bowtie_{sid=sid} Sailors)$
    - $\pi_{sname}(\pi_{sid}(\sigma_{bid=100}(Reserves)) \bowtie_{sid=sid} \pi_{sname,sid}(\sigma_{rating>5}(Sailros)))$
    - it makes sure that we only have the columns that necessary in tables that we input (merge-join or a hash table). They'll only take up memory for these columns that we care about.

Avoid Cartesian Products

- Given a choice, do theta-joins rather than cross-products.
- Consider R(a,b), S(b,c), T(c,d)
- Favor $(R \bowtie S) \bowtie T$ over $(R \times T) \bowtie S$
- Exception: If we have two small tables and their cross product is also small, it can sometimes we did good to do that cross product rather than joining one of the tables with a big table and then taking the even bigger output and joining it. So sometimes if you have very small tables, cross products can be good.

Physical Equivalences

- Base Table access,
  - With single-table selections and projections
    - Heap scan
    - Index scan (if available on referenced columns)

- Equijoins
  - Block (Chunk) Nested Loop: simple, exploits extra memory
  - Index Nested Loop: often good if 1 rel small and the other indexed properly
  - Sort-Merge Join: good with small memory, equal-size tables
  - Grace Hash Join: even better than sort with 1 small table
    - Or Hybrid if you have it

Schema for Examples

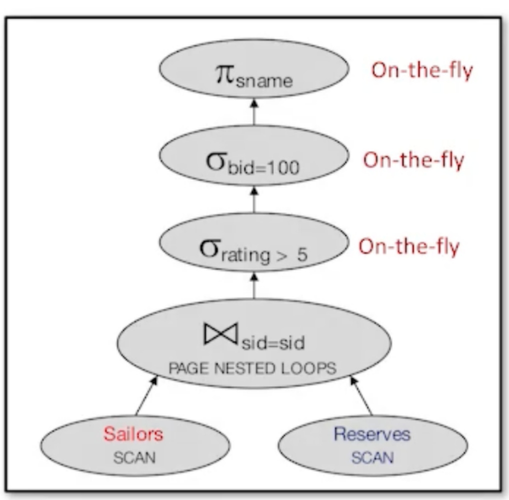
Sailors (sid: integer, sname: text, rating: integer, age: real)

- Assume 10 different ratings and each rating is equally likely
  Reserves (sid: integer, bid: integer, day: date, rname: text)

- Reserves:
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages.
  - Assume there are 100 boats
- Sailors:
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages.
  - Assume there are 10 different ratings
- Assume we have 5 pages to use for joins.

Motivating Example: Plan 1

Here's a reasonable query plan:




```
SELECT S.sname
  FROM Reserves R, Sailors S
 WHERE R.sid=S.sid
   AND R.bid=100
   AND S.rating>5
```

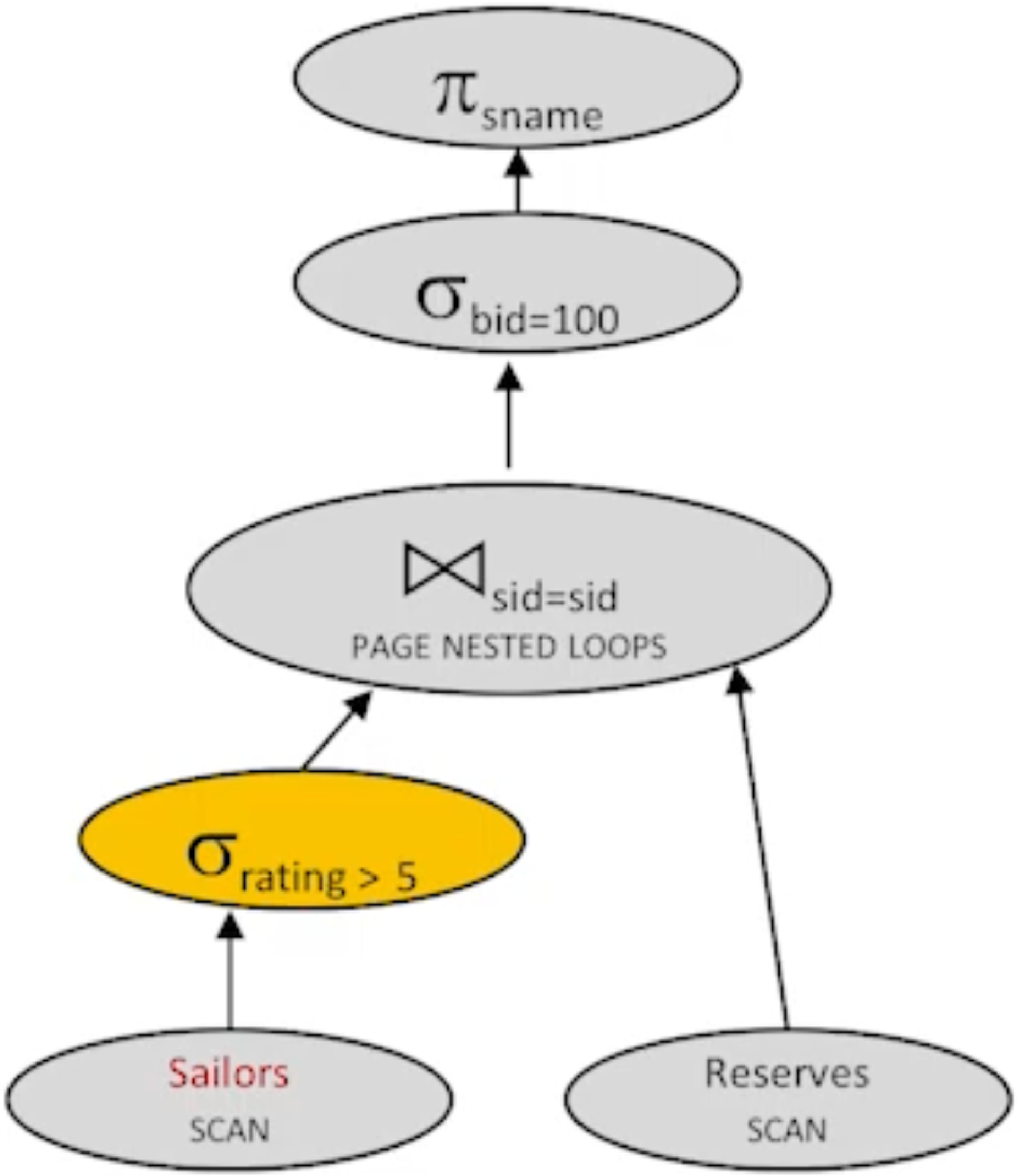
- Estimate the cost:

    - Scan Sailors (500 IOs)
    - For each page of Sailors,
        - Scan Reserves (1000 IOs)
    - Total: 500 + 500*1000 = 500,500 IOs
- By no means the worst plan!
- Misses several opportunities:
  - selections could be 'pushed' down
  - no use made of indexes
  Goal of optimization: Find faster plans that compute the same answer.

Query Plan 2 Cost

- Let's estimate the cost:
- Scan Sailors (500 IOs)

- For each pageful of high-rated Sailors,
    - Scan Reserves (1000 IOs)
- Total: 500 + 0.5*500*1000 = 250,500 IOs
    - Assume uniform distribution (evenly distributed)

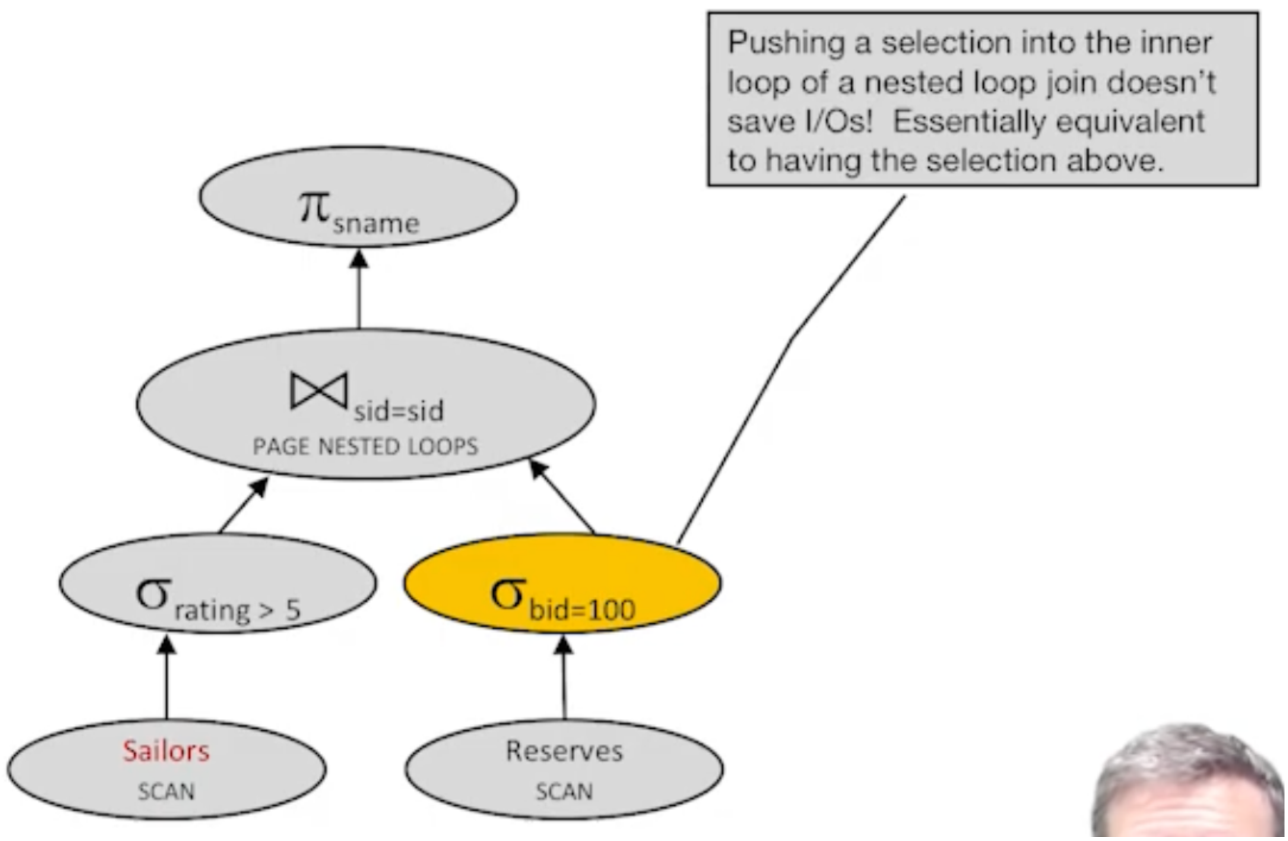


-

Query Plan 3 Cost Analysis

- Let's estimate the cost:
- Scan Sailors (500 IOs)

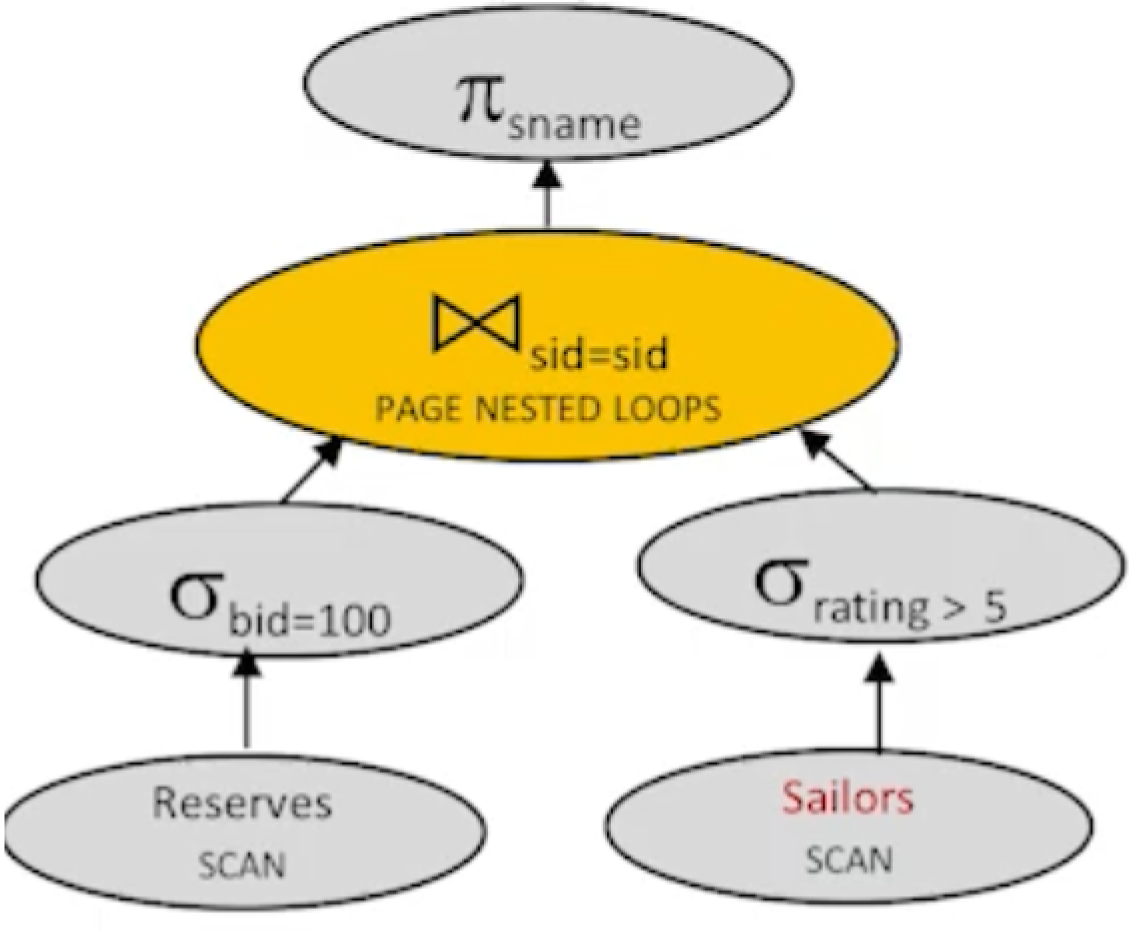
- For each pageful of high-rated Sailors,
  - Do what? (??? IOs)
- Total: 500 + 250*1000
- Filtering happens on the fly, which means that for each scan of reserves it will reduce tuples, but we're still doing the scan.
- For every page full of high rated sailors we are still scanning reserves although we're throwing away tuples as we scan
- It does not save IOs because the selection is happening on the fly after the i/o passed the reserves each time we loop through it.



Query Plan 4 Cost

- Let's estimate the cost:
- Scan Reserves (1000 IOs)
- For each pageful of Reserves for bid 100,
  - Assume reservations were equally distributed (10 pages with bid 100, 1000 tuples per page)
  - Scan Sailors (500 IOs)

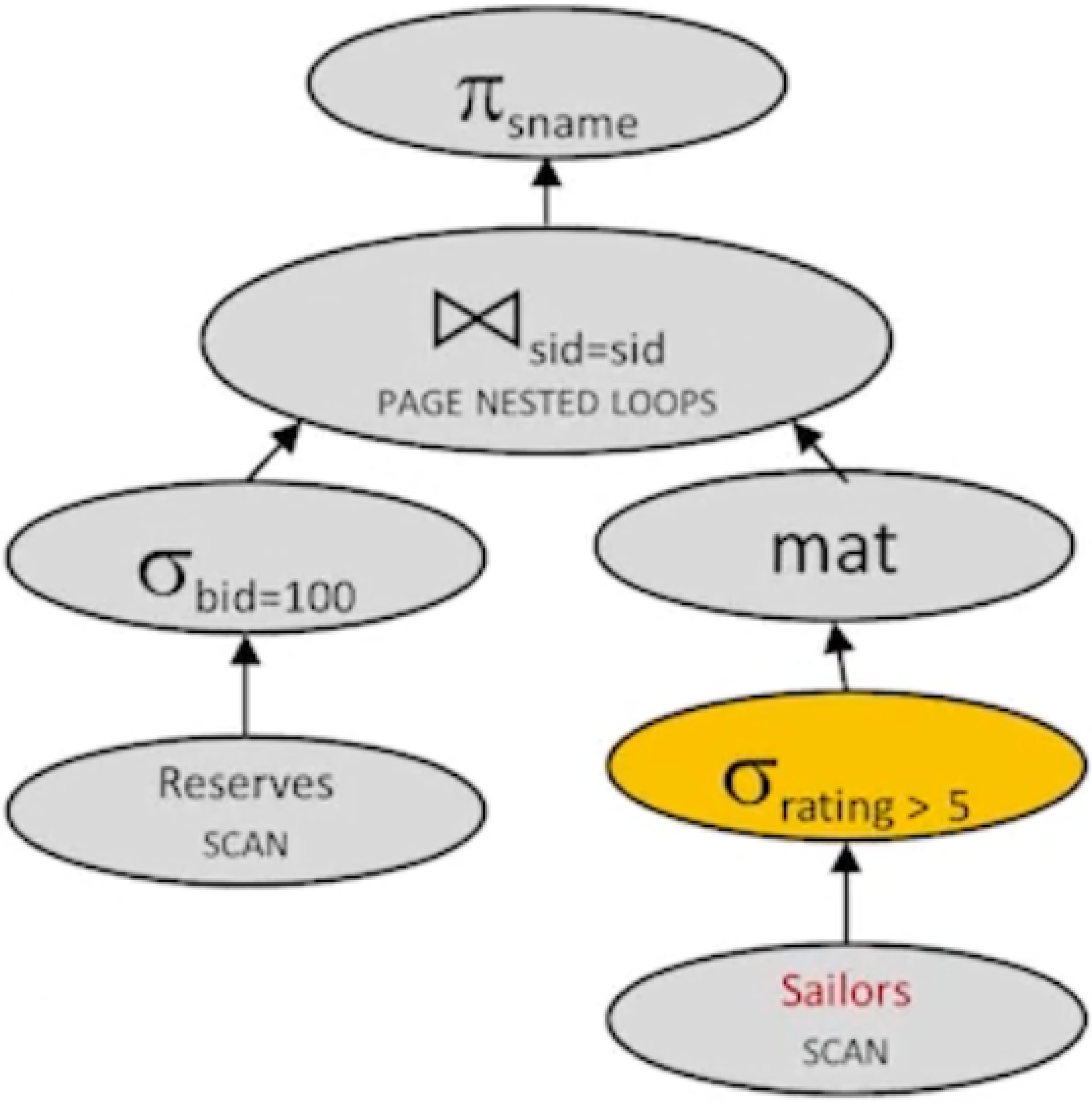
- Total: 1000+10*500 = 6000



Plan 5 Cost Analysis — Materializing Inner Loops, cont

- Let's estimate the cost:
- Scan Reserves (1000 IOs)
- Scan Sailors (500 IOs)
- Materialize Temp table T1 (250 IOs)
- For each pageful of Reserves for bid 100,
    - Scan T1 (250 IOs)

- Total: 1000 + 500 + 250 + 10*250 = 4250 IOs

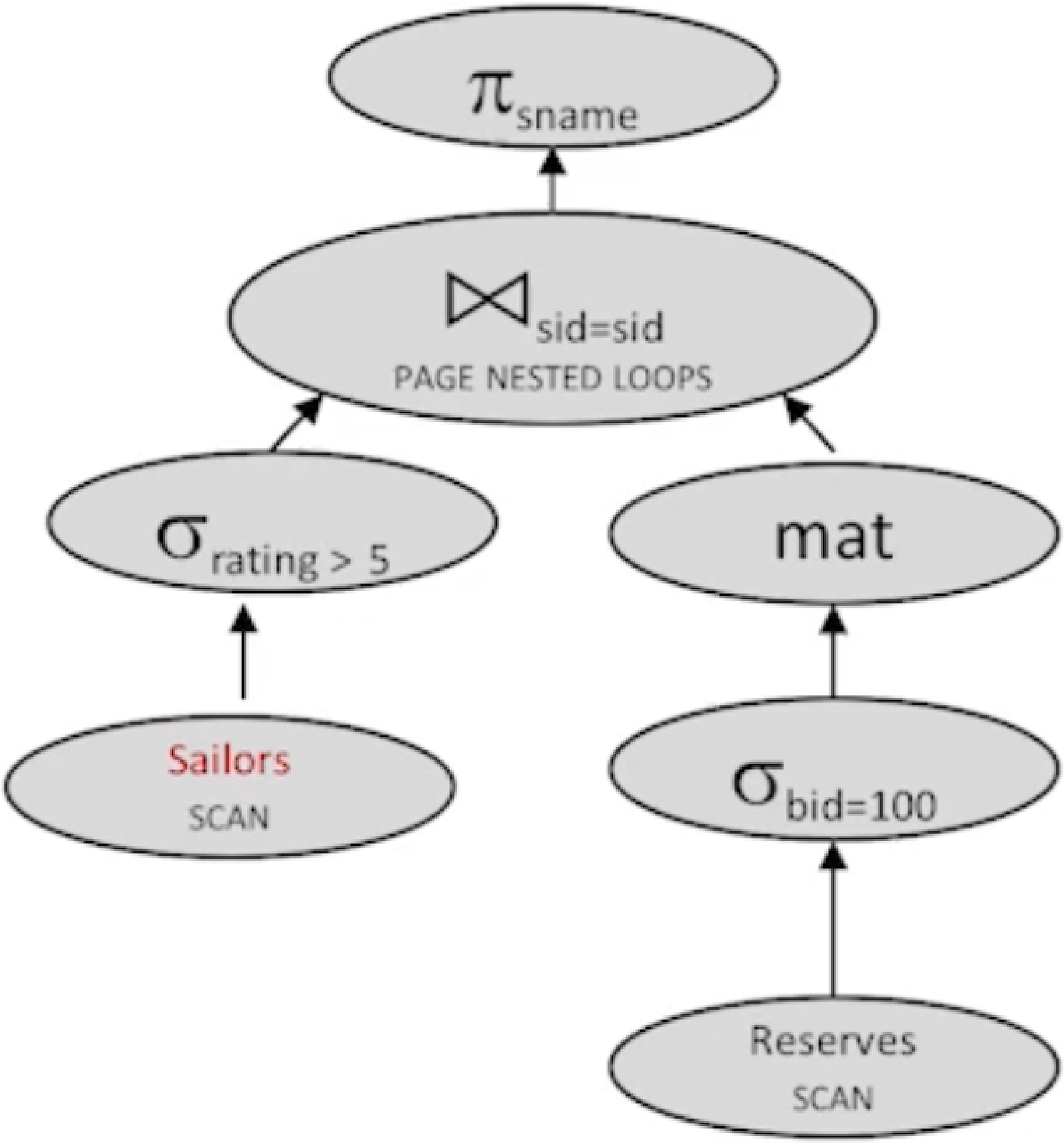


Plan 6 — Join Ordering Again, Cont

Let's estimate the cost:

- Scan Sailors (500 IOs)
- Scan Reserves (1000 IOs)
- Materialize Temp table T1 (10 IOs)
- For each pageful of high-rated Sailors (10 pages in total),
    - Scan T1 (1000 IOs)

- Total: 500+1000+10+250*10 = 4010 IOs

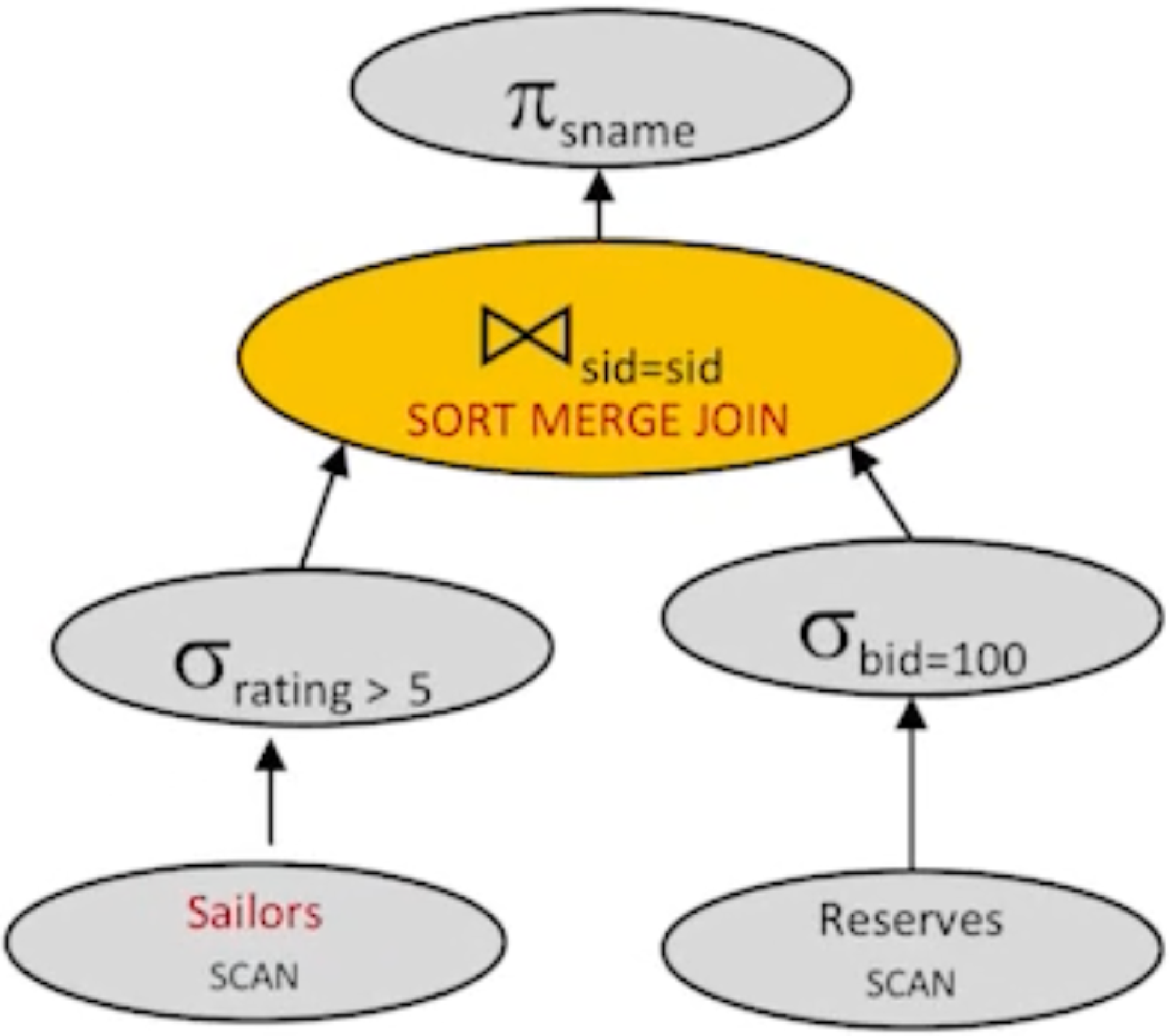


Plan 7 — Join Algorithm, cont. — Sort Merge Join

Sort Merge Join

- With 5 buffers, cost of plan:
- Scan Reserves (1000)
- Scan Sailors (500)
- Sort high-rated sailors (???)
    - Note: pass 0 doesn't do read I/O, just gets input from select.

- Sort reservations for boat 100 (???)
  - Note: pass 0 doesn't do read I/O, just gets input from select.
- How many passes for each sort with log_4?
  - Sort
    - 2 passes for reserves: pass0 = 10 to write, pass1 = 2*10 to read/write
    - 4 passes for sailors: pass0 = 250 to write, pass1,2,3 = 2*250 to read/write
- Merge (10+250) = 260
- Total: 1000+500+sort reserves(10+2*10)+sort sailors(250+3*2*250)+merge(10+250)=3630

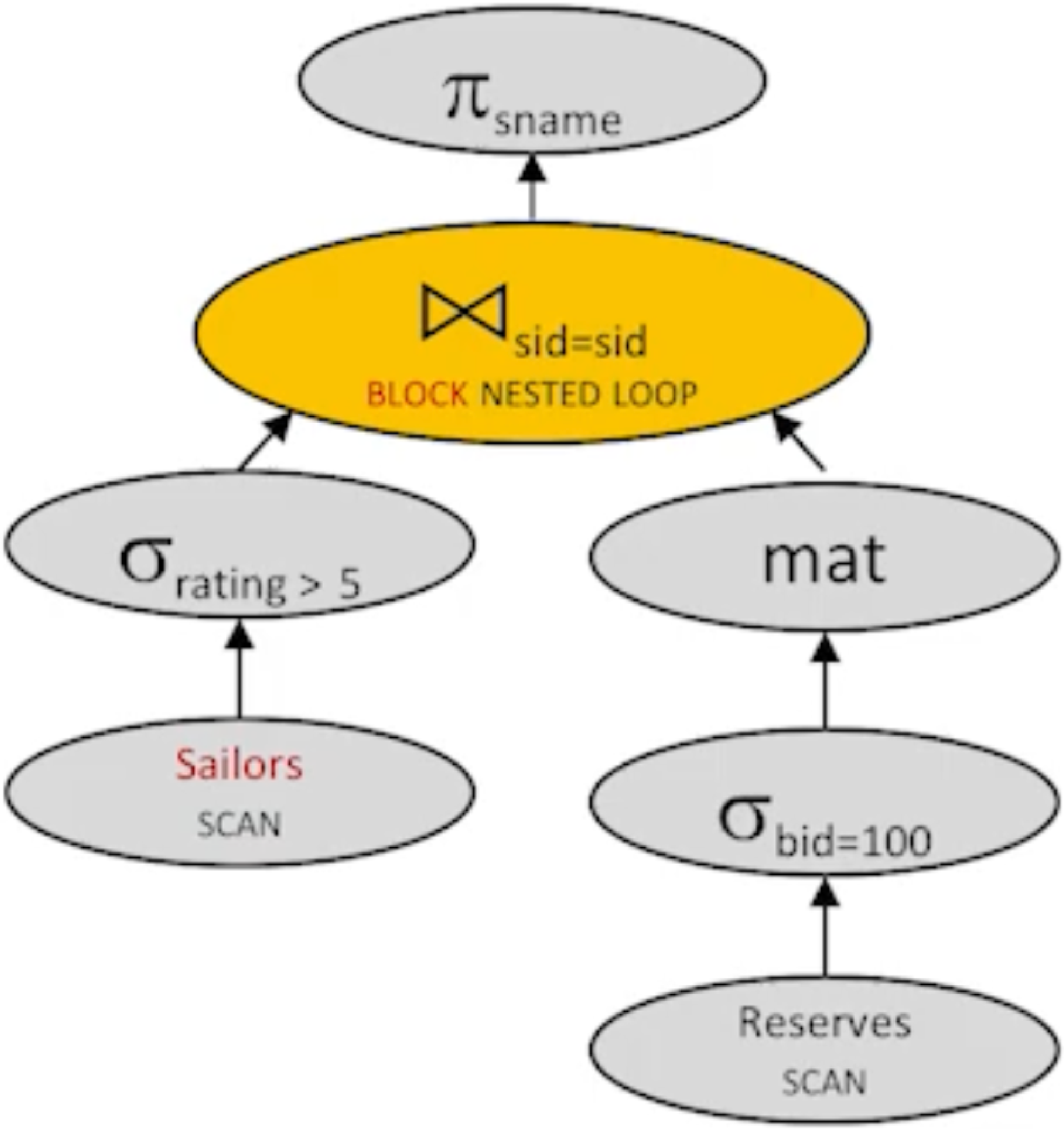


Query Plan 9 — BLOCK NESTED LOOP

• With 5 buffers, cost of plan:

Scan Sailors (500)

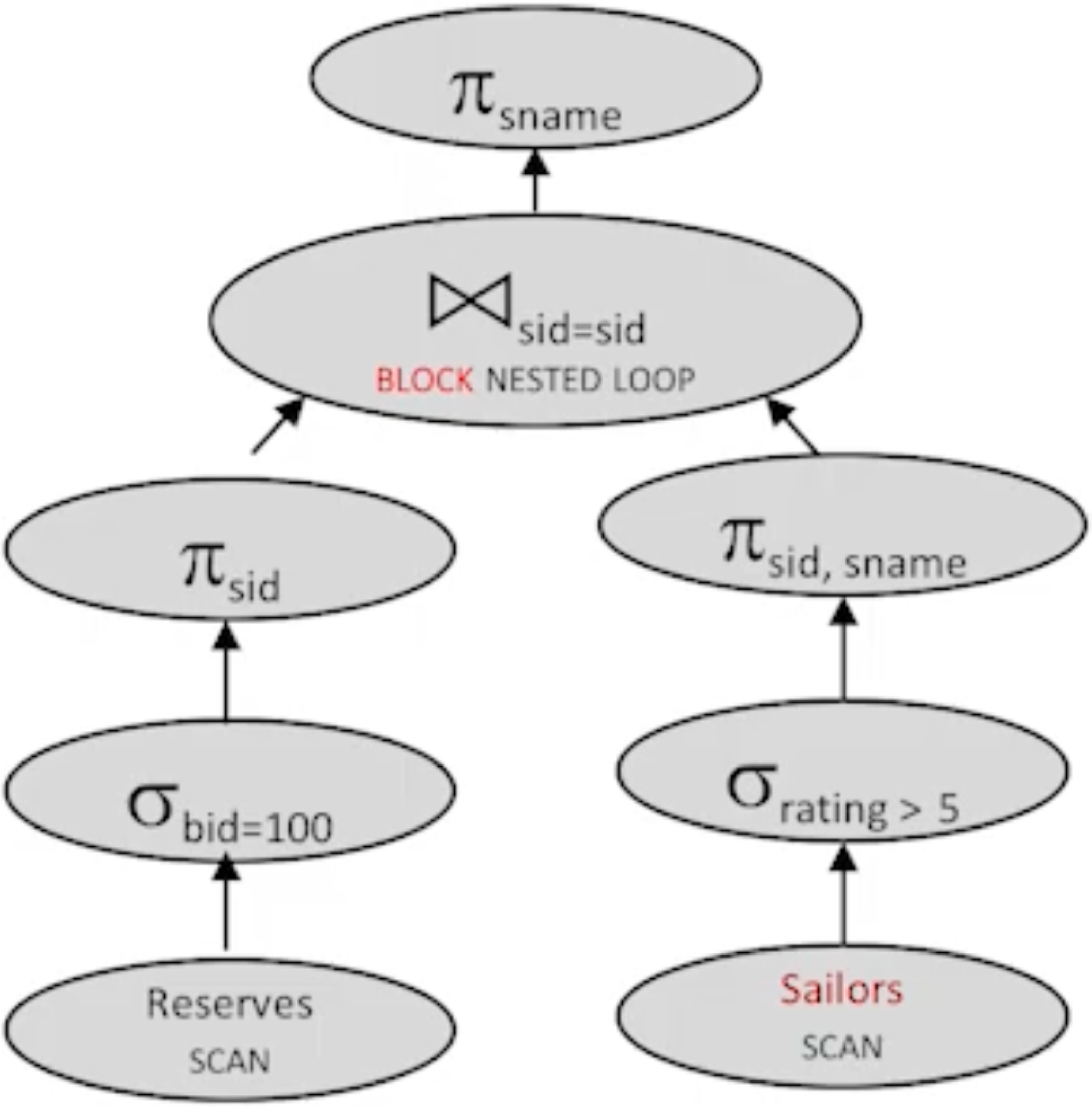
- Scan Reserves (1000)
- Write Temp T1 (10)
- For each blockful of high-rated sailors
- Loop on T1 (ceil(250/5) * 10)
- Total: 500+1000+10+(ceil(250/4)*10)=500+1000+10+(63*10)=2140 IOs



Plan 11 — Projection Cascade & Pushdown, cont

- With 5 buffers, cost of plan:
- Scan Reserves (1000)
- For each blockful of sids that rented boat 100

- (recall Reserve tuple is 40 bytes, assume sid is 4 bytes)
- Loop on Sailors ((1000/100/(40/4)) * 500=500)
- Total: 1000+500=1500



Indexex

- Indexes:
- Reserves.bid clustered
- Sailors.sid unclustered
    - Column sid is a primary key for Sailors.
    - <=1 matching tuple, unclustered index on sid OK

- Assume indexes fit in memory

“Pasted image 20240506003125.png” could not be found.

- With clustered index on bid of Reserves, we access how many pages of Reserves?:
- 100,000/100 = 1000 tuples on 1000/100 = 10 pages.
- for each Reserves tuple 1000
    - get matching Sailors tuple (1 10)
    - (recall: 100 Reserves per page, 1000 pages)
- 10 + 1000*1
- Cost: Selection of Reserves tuples (10 I/Os); then, for each, must get matching Sailors tuple (1000); total 10101/Os.

Query Optimization II: Costing and Searching

Introduction

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**
    1. Based on relational equivalences, different implementations
2. **Cost Estimation** based on
    1. Cost formulas
    2. Size estimation, in turn based on
        1. Catalog information on base tables
        2. Selectivity (Reduction Factor) estimation
3. **A search algorithm**
    1. To sift through the plan space and find the lowest cost option!

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 AlgorithmL: Dynamic Programming

Query Blocks and Physical Properties

Query Block: 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"
  - Try to flatten these query blocks into a single block when possible before giving it to the optimizer

```sql
-- Outer Block
SELECT S.sname
FROM Sailors S
WHERE S.age IN
-- Nested Block
(SELECT MAX (S2.age)
 FROM Sailors S2
 GROUP BY S2.rating
)
```

- 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

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.

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. NestLoop Join preserves sort order of outer (left) input

Plan Space

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 experessions
- 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

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 the **size of result** for each operation in tree!
    - Because it determines downstream input cardinalities!
      - A strong assumption for simplifying: For selections and joints, the predicates that they use are statistically independent of each other.
    - 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 $\#I/O + CPU factor * \#tuples$
  - CPU-factor is a kind of scaling factor

Statistics and Catalogs

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

Statistics	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
  - selectivity = $|ouput|/|input|$
  - Book calls selectivity "Reduction Factor" (RF)
- Avoid confusion:
  - "High selectivity" in common English is opposite of a high selectivity value ($|ouput|/|input|$ high!)
  - the size of the output over the size of the input is big which means that it lets almost everybody through

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

Result Size Estimation

- Result cardinality = Max # tuples * products of all selectivities
- Term col = value (given Nkeys(I) on col)

    - $sel = 1/NKeys(I)$
- Term col1=col2 (handy for joins too)
    - $sel = 1/MAX(NKeys(I1), NKeys(I2))$
    - Why MAX? See bunnies in 2 slides
- Term col>value
    - $sel = (High(I) - value)/(High(I) - Low(I) + 1)$
- 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(leftEar = rightEar)

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

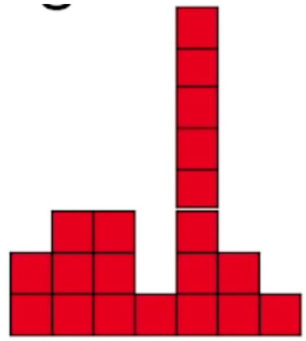
$$\begin{aligned} P(L=R) = \sum_i P(C_i, C_i) \\ = P(C1, C1) + P(C2, C2) + P(C3 + C3) + \ldots \\ = (1/2 * 1/10) + (1/2 * 1/10) + 0 + \ldots \\ = \sum_{k \in L} 1/|L| * 1/|R| \\ = 1/|R| \\ = 1/10 \\ = 1/MAX(2, 10) \end{aligned}$$

Using Histograms for Selectivity Estimation

- 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

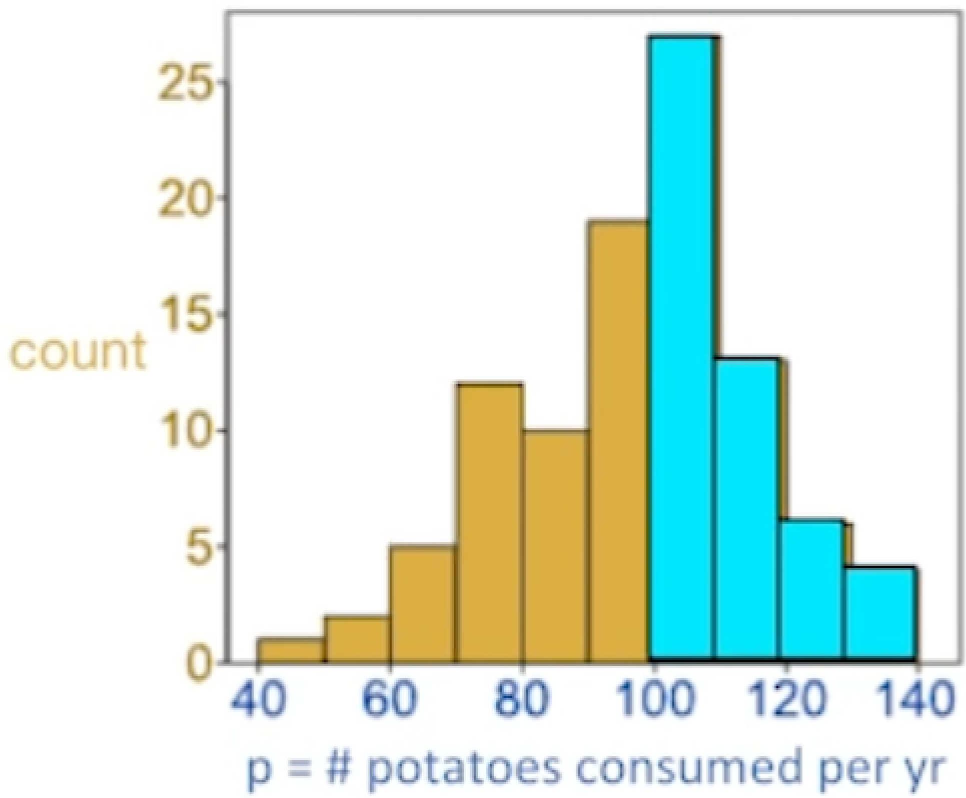
See also V-Optimal histograms on Wikipedia

Equiwidth Histogram

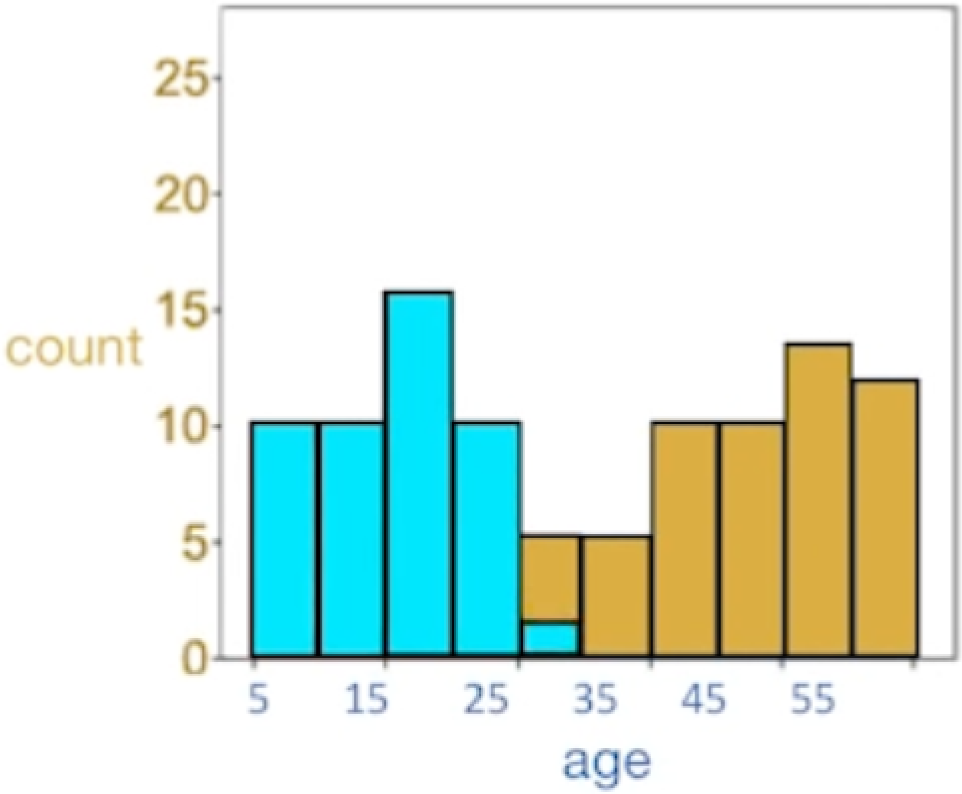
Equidepth Histogram: in the equidepth histogram, we divided the data up into ranges of equal count

V-Optimal hisotgrams

Computing Selectivity with histograms



- 100 rows
- $\sigma_p > 99$?
  - 50/100=50%



- 100 rows
- $\sigma_{age} < 26$?
- assume: the width of that slice is the same proportion as that proportion of the tuples. It's the same as the height of that slice proportionally.
- Uniformity assumption:
  - Uniform distribution within each bin Each vertical slice the same
  - Hence ⅕ of the population of bin [25,30) has age < 26.
  - 10 + 10 + 15 + 10 + (1/5 * 5) = 46/100 =46%

Selectivity of Conjunction

- 100 rows
- $\sigma_{p>99 \wedge age<26}$?
  - 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% × 46% = 23%

- 100 rows
- $\sigma_{p>99 \vee age<26}$?
  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% × 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| × |S|!

- Total rows: $s_p \times |R| \times |S|$

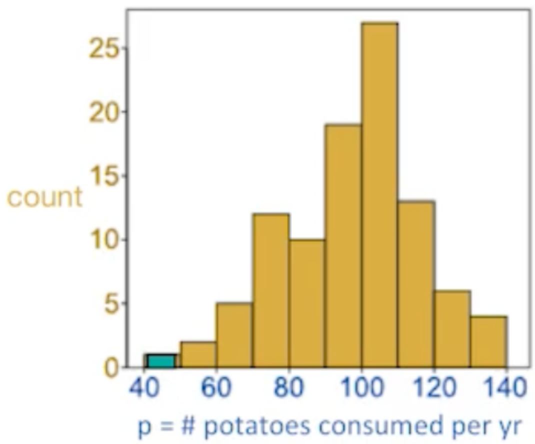
Selectivity for our ealier query

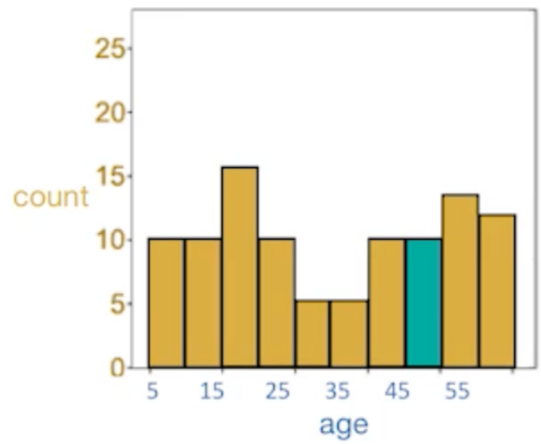
- Recall from algebraic equivalences
  - $R \bowtie_p \sigma_q(S) \equiv \sigma_p(R \times \sigma_q(S)) \equiv \sigma_{p \wedge q}(R \times S)$
  - Hence selectivity just $s_p s_q$
    - Applied to |R| x |S|!

- 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:
    - $P(T.p = v) = height(binp(v))/n * 1/width(binp(v))$
    - $P(T.age = v) = height(binage(v))/n * 1/width(binage(v))$
    - 



    - // 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 versione

- 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 mutiny by the product o of the table sizes

Search Algorithm

Enumeration of Alternative Plans

- There are two main cases:
    - Single-table plans (base table)
    - 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 (Height(I)+1)+1 for a B+ tree.
- Clustered index I matching one or more selects:
    - (NPages(I)+NPages(R))* products of sel's of matching selecs.
- Non-clustered index I matching one or more selects:
    - (NPages(I)+NTuples(R))* products of sel's of matching selects.
- Sequential scan of file:
    - NPages(R).
- Recall: Must also charge for duplicate elimination if required

Example

- If we have an index on rating:
    - Cardinality = (1/NKeys(I)) *NTuples(R) = (1/10)* 40000 = 4000 tuples
    - Clustered index: (1/NKeys(I)) *(NPages(I)+NPages(R))) = (1/10)* (50+500)= 55 pages are retrieved. (This is the cost.)

        - Unclustered index: (1/NKeys(I) *(NPares(I))+NTuples(R)) = (1/10)* (50+40000) = 4005 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
- Let's try to avoid exhaustively enumerating all such plans!
    - By observing that many of them share common subplans.
    - Can we save work by remembering decisions on subplans?

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 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 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

Subset of tables in FROM clause	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

Subset of tables in FROM clause	Interesting order columns	Best Plan	Cost
{A, B}	<none>	hashjoin(A,B)	1000
{A, B}	<A.x, B.y>	sortmerge(A,B)	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 Query

- Sailors:
  - Hash, B+ tree indexes on sid
- Reserves:
  - Clustered B+ tree on bid
  - B+ on sid
- Boats
  - B+ on color

```
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
```

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

Subset of tables in FROM clause	Interesting: order columns	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)
```

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

Subset of tables in FROM clause	Interesting: order columns	Best plan	Cost
{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