Relational Query Optimization II: Costing and Searching

The image part with relationship ID rId2 was not found in the file.

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

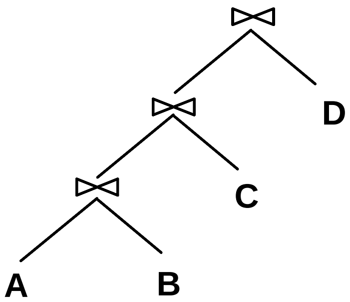
- Works well for up to 10-15 joins.
- **Plan Space**: Too large, must be pruned.
  - Algorithmic insight:
    - Many plans could have the same “overpriced” subtree
    - Ignore all those plans
  - Common heuristic: consider only left-deep plans
  - Common heuristic: avoid Cartesian products
- Cost estimation
  - 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 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”
  - Beyond the scope of CS186!




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

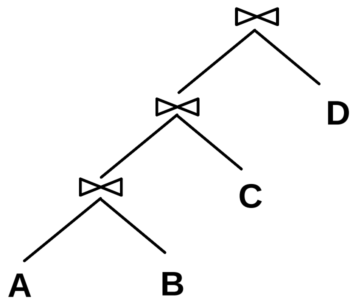
Outer block

```
  (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)
```

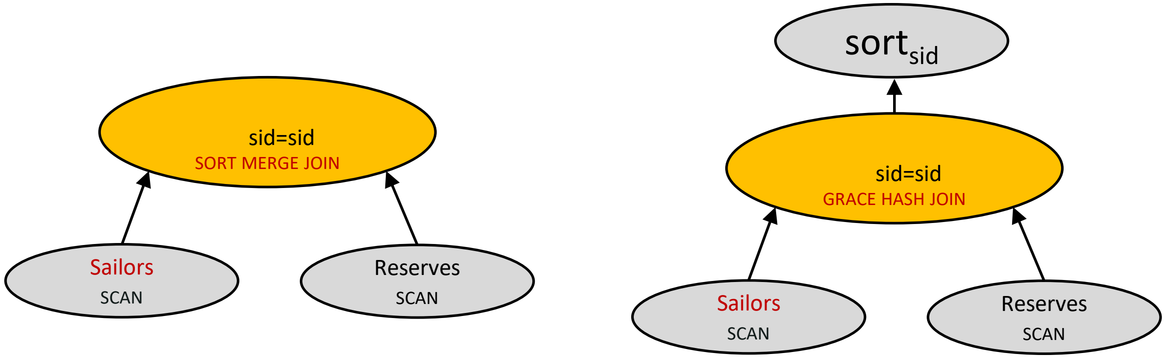
- Reserves:
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages.
  - 100 distinct bids.
- Sailors:
  - Each tuple is 50 bytes long,
  - 80 tuples per page, 500 pages.
  - 10 ratings, 40,000 sids.

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

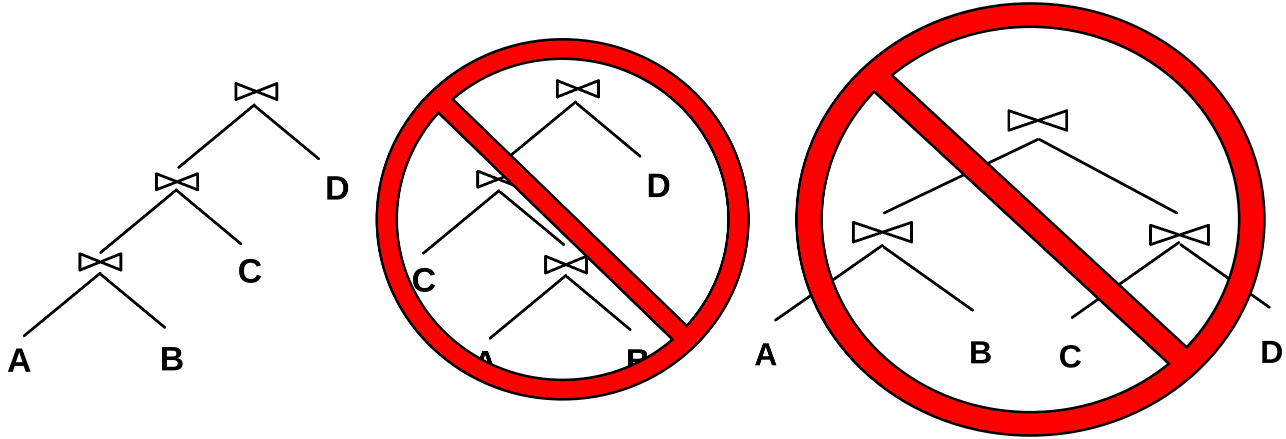
Physically Equivalent Plans

- 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.
    - 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 + ***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.
  - selectivity = |output| / |input|
  - Book calls selectivity “Reduction Factor” (RF)
- Avoid confusion:
  - “highly selective” in common English is opposite of a high selectivity value (|output|/|input| high!)

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

Result Size Estimation

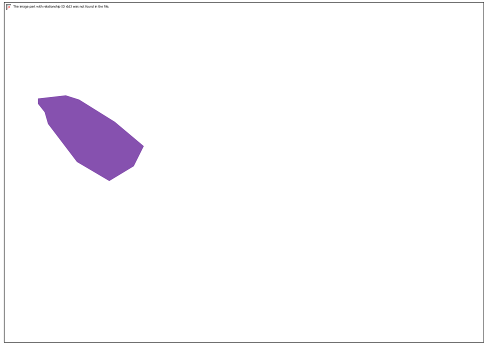
- Result cardinality = Max # tuples * **product** of all selectivities.
- Term col=value (given Nkeys(I) on col)
  - sel = 1/NKeys(I)
- Term col1=col2 (handy for joins too…)
  - sel = 1/MAX(NKeys(I1), NKeys(I2))
  - Why MAX? See bunnies in 2 slides…
- Term col>value
  - sel = (High(I)-value)/(High(I)-Low(I) + 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?

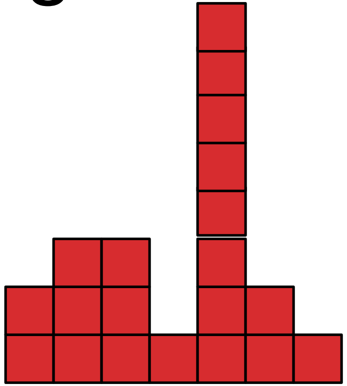


Postgres 10.0: src/include/utils/selfuncs.h

```
/* 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)
```

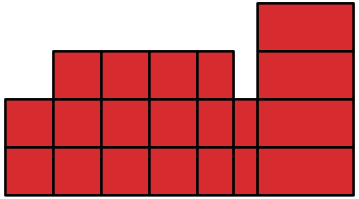
Reduction Factors & Histograms

- For better estimation, use a histogram



equiwidth

equidepth



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

- akin to quantiles, median, etc.

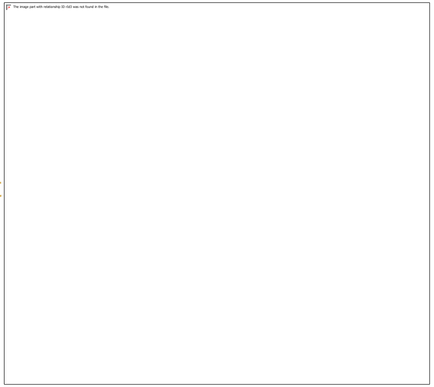
Common trick: “end-biased” histogram

- very frequent values in their own buckets

See also V-Optimal histograms on Wikipedia

Computing selectivity with histograms

- 100 rows
- $\sigma_{p > 99}$?

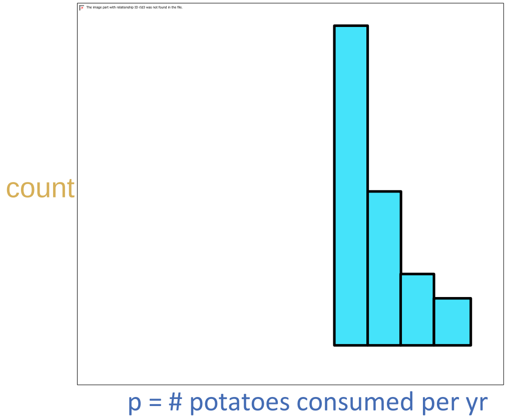


count

p = # potatoes consumed per yr

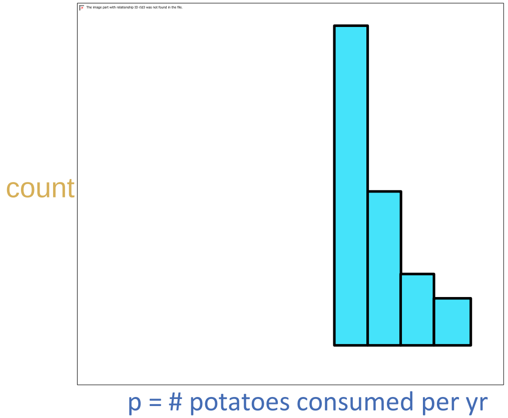
Computing selectivity with histograms, Pt 2

- 100 rows
- $\sigma_{p > 99}$?



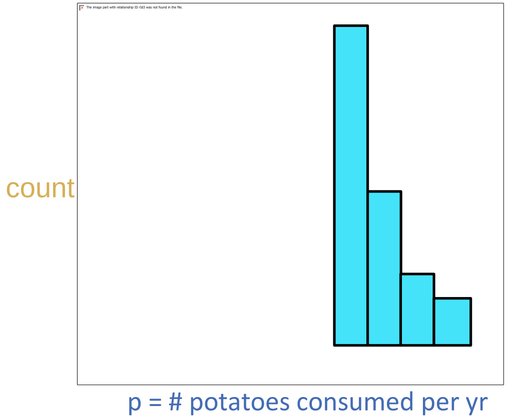
Computing selectivity with histograms, Pt 3

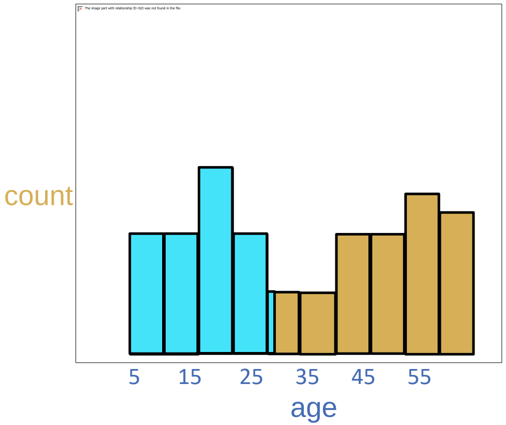
- 100 rows
- $\sigma_{p > 99}$? 50/100 = 50%.



Computing selectivity with histograms, Pt 4

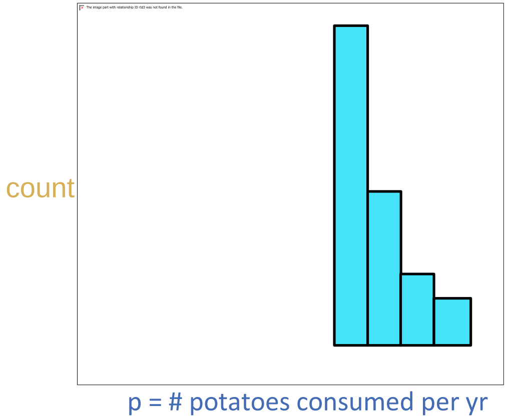
- 100 rows
- $\sigma_{age < 26}$?

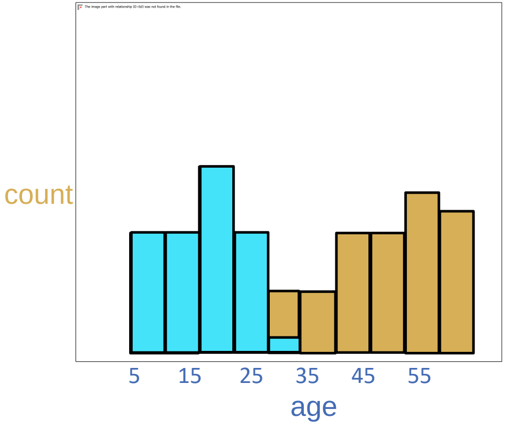




Computing selectivity with histograms, Pt 5

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

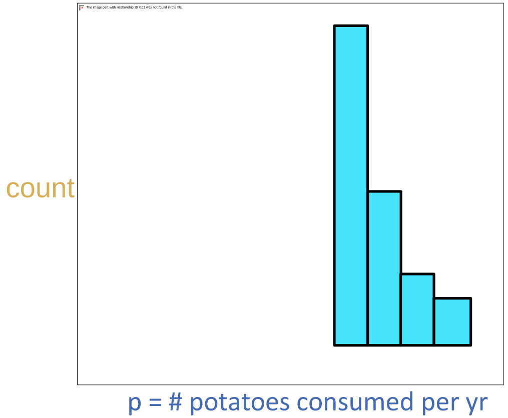


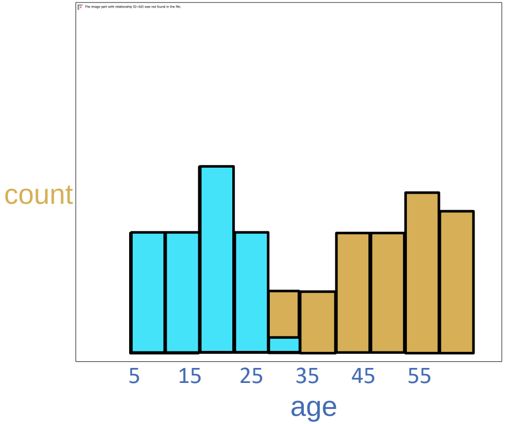


Computing selectivity with histograms, Part 6

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

- **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 + (⅕ * 5) = 46/100 **= 46%**

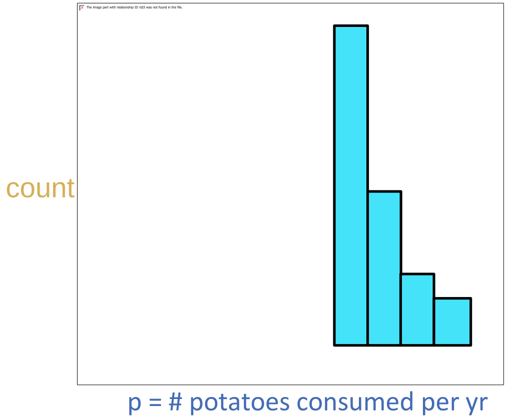


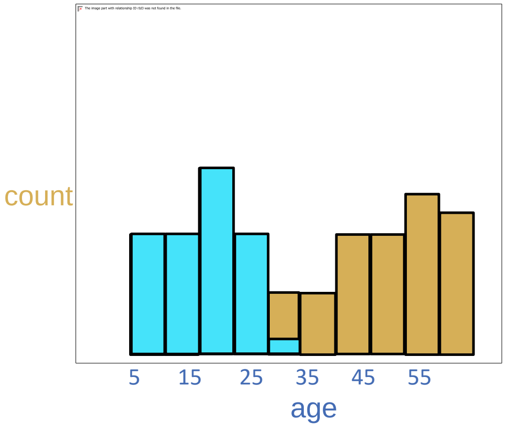


Selectivity of Conjunction

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

  50% 46%



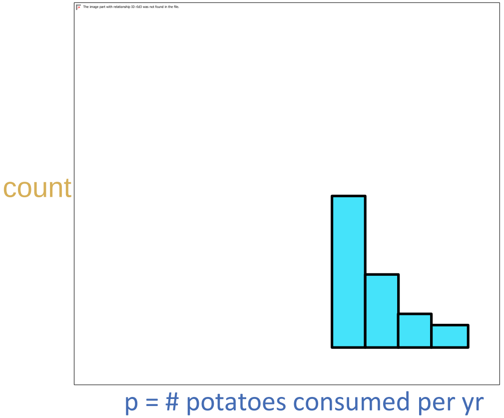


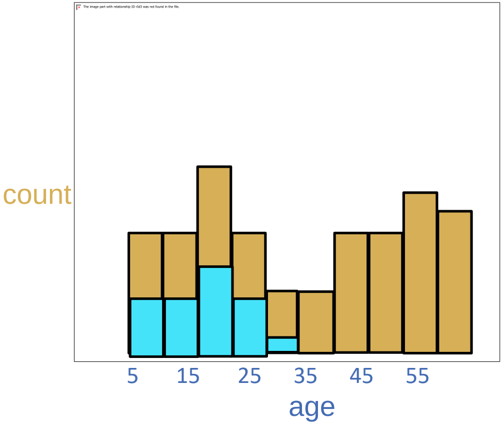
Selectivity of Conjunction, cont

- 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%.





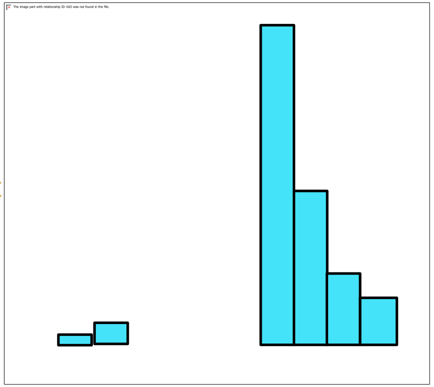
Selectivity: 50% × 46% = **23%**

Selectivity of Disjunction

- 100 rows
- $\sigma_{p\ >\ 99\ \vee\ p\ <\ 60}$?

  50%　　　3%

count



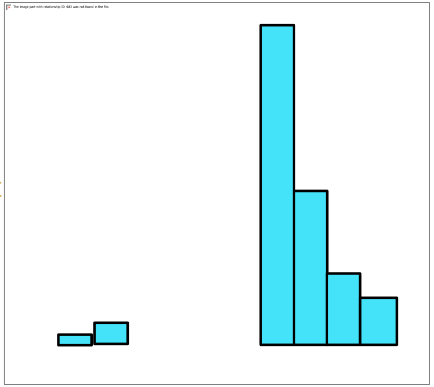
p = # potatoes consumed per yr

Selectivity of Disjunction, Part 2

- 100 rows
- $\sigma_{p > 99 \vee p < 60}$?

  50% 3%
- Selectivity: 50% + 3% = **53%**

count



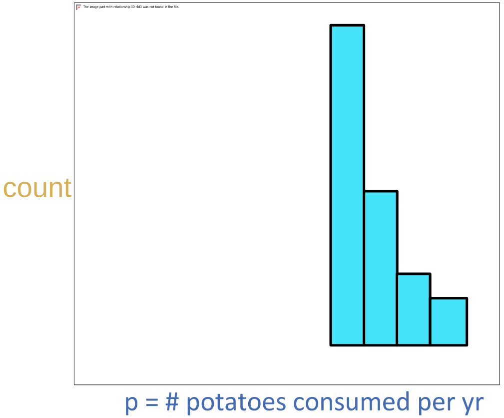
p = # potatoes consumed per yr

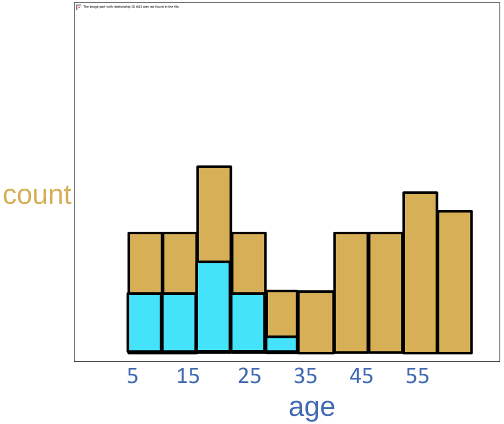
Selectivity of Disjunction, Part 3

- 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%**

The image part with relationship ID rId2 was not found in the file.

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.

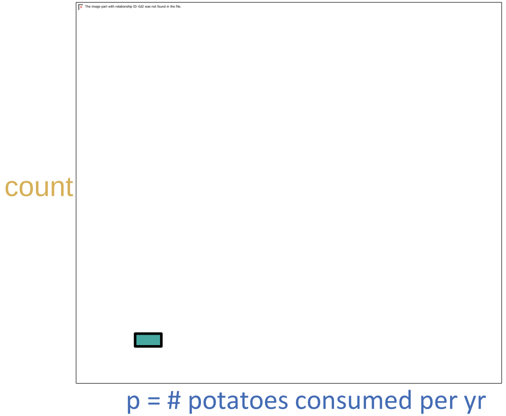
s = 0

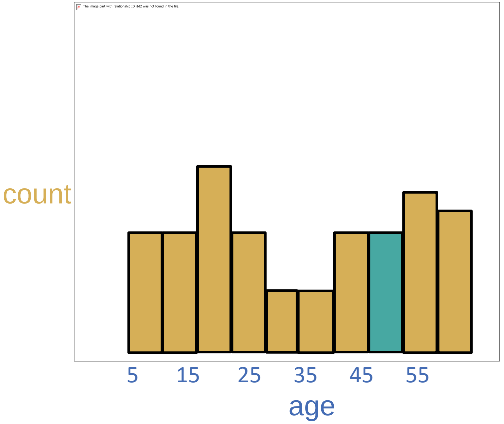
For each value v covered in either histogram:

    // uniformity assumption within bins:

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

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

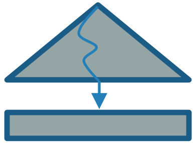
1. Plan Space
2. Cost Estimation
3. Search Algorithm

Enumeration of Alternative Plans

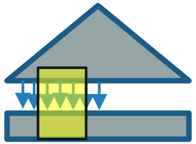
- There are two main cases:
  - **Single-table plans (base case)**
  - **Multiple-table plans (induction)**
- Single-table queries include selects, projects, and groupBy/agg:
  - Consider each available access path (file scan / index)
    - Choose the one with the least estimated cost
  - Selection/Projection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

Cost Estimates for Single-Relation Plans

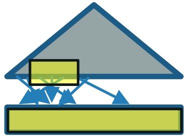
- Index I on primary key matches selection:
  - Cost is (Height(I) + 1) + 1 for a B+ tree.



- Clustered index I matching selection:
  - (NPages(I)+**NPages**(R)) * selectivity.



- Non-clustered index I matching selection:
  - (NPages(I)+**NTuples**(R)) * selectivity.



- Sequential scan of file:
  - NPages(R).
- Recall: Must also charge for duplicate elimination if required

Example

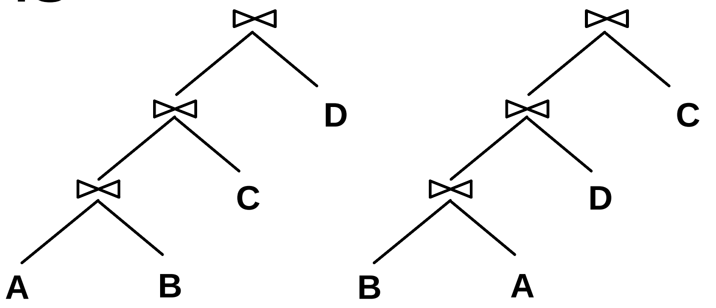
```
SELECT S.sid
  FROM Sailors S
 WHERE S.rating=8
```

- 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) = **55 pages are retrieved**. (This is the cost.)
  - **Unclustered index**: (1/NKeys(I)) * (NPages(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).

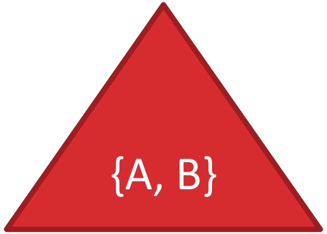
Enumeration of Left-Deep Plans



- Left-deep plans differ in
  - the order of relations
  - the access method for each leaf operator
  - the join method for each join operator
- 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

- 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) ⋈ C
  - (The best plan for joining A, C) ⋈ B
  - (The best plan for joining B, C) ⋈ A
- This is great!
  - When optimizing a subplan (e.g. A ⋈ 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 ⋈ B ⋈ C) we can reuse the best results of subroutines (e.g. A ⋈ B)!



Dynamic Programming Algorithm for System R

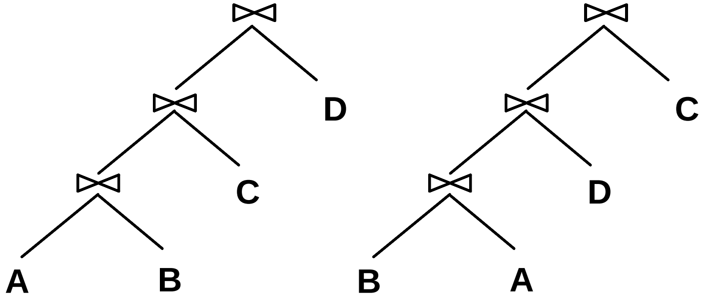
- Principle of optimality allows us to build best subplans "bottom up"
  - Pass 1: Find best plans of height 1 (base table accesses), and record them in a table
  - Pass 2: Find best plans of height 2 (joins of base tables) by combining plans of height 1, record them in a table
  - …
  - Pass i: Find 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

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



- 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.

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

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

Table keyed on concatenation of 1st two columns

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

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	
{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

Physical DB Design

R&G 20

The image part with relationship ID rId2 was not found in the file.

Physical DB Design

- Query optimizer does what it can to use indices, clustering etc.
- DataBase Administrator (DBA)
  - expected to set up physical design well
- Good DBAs understand query optimizers very well

One Key Decision: Indexes

- Which tables
- Which field(s) should be the search key?
- Multiple indexes?
- Clustering?

Index Selection

- A greedy approach:
  - Consider most important queries in turn.
  - Consider best plan using the current indexes
  - See if better plan is possible with an additional index.
  - If so, create it.

- But consider impact on updates!
  - Indexes can make queries go faster, updates slower.
  - Require disk space, too.

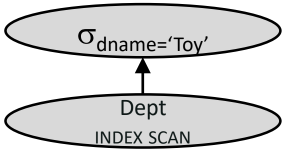
Issues to Consider in Index Selection

- Attributes mentioned in a WHERE clause are candidates for index search keys.
  - Range conditions are sensitive to clustering
  - Exact match conditions don't require clustering
    - Or do they????
    - What if you have a lot of duplicate values? Then just like range search!
- Choose indexes that benefit many queries
- NOTE: only one index can be clustered per relation!
  - So choose it wisely!

Example 1, Part 1

```
SELECT E.ename, D.mgr
  FROM Emp E, Dept D
 WHERE E.dno=D.dno
   AND D.dname='Toy'
```

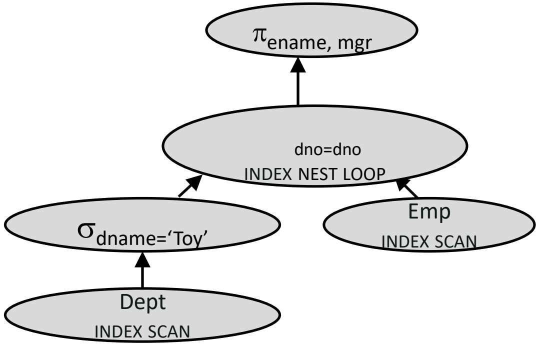
- B+ tree index on D.dname supports 'Toy' selection.
  - Given this, index on D.dno isn't important.



Example 1, Part 2

```
SELECT E.ename, D.mgr
  FROM Emp E, Dept D
 WHERE E.dno=D.dno
   AND D.dname='Toy'
```

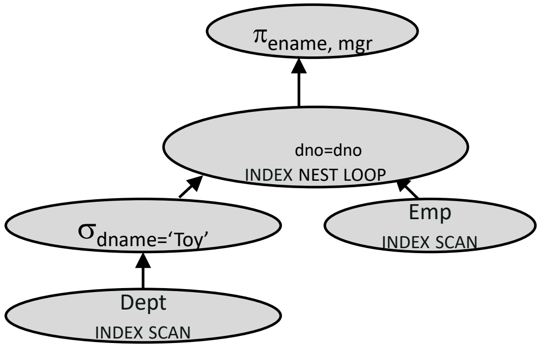
- B+ tree index on D.dname supports 'Toy' selection.
  - Given this, index on D.dno isn't important:
    D is already filtered quite a lot prior to join.
- B+ tree index on E.dno allows us to get matching (inner) Emp tuples for each selected (outer) Dept tuple.

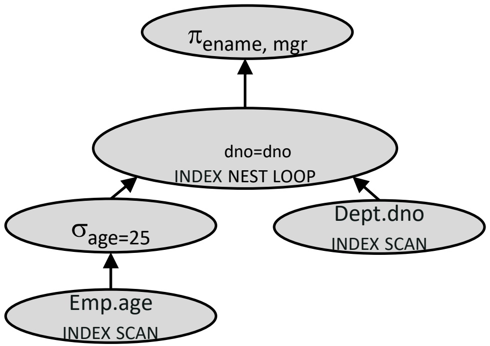


Example 1, Part 3

```
SELECT E.ename, D.mgr
  FROM Emp E, Dept D
 WHERE E.dno=D.dno
   AND D.dname='Toy'
```

- What if WHERE included: `` ... AND E.age=25'' ?
  - Could retrieve Emp tuples using index on Emp.age, then join with Dept tuples satisfying dname selection.
    - Comparable performance to strategy that used E.dno index.
  - So, if Emp.age index is already created, this query provides much less motivation for adding an Emp.dno index.





Example 2

```
SELECT  E.ename, D.mgr
FROM  Emp E, Dept D
WHERE  E.sal BETWEEN 10000 AND 20000
  AND E.hobby='Stamps' AND E.dno=D.dno
```

- All selections are on Emp so it should be the outer relation in any Index NL join.
    - Suggests that we build a B+ tree index on D.dno.
- What index should we build on Emp?
    - B+ tree on E.sal could be used, OR an index on E.hobby could be used.
    - Only one of these is needed, and which is better depends upon the selectivity of the conditions.
        - As a rule of thumb, equality selections more selective than range selections.
- Helps to understand optimizers to get this right!

Examples of Clustering

- B+ tree index on E.age can be used to get qualifying tuples.
  - How selective is the condition?
  - Is the index clustered?
- Consider the GROUP BY query.
  - If many tuples have E.age > 10, using E.age index and sorting the retrieved tuples may be costly.
  - Clustered E.dno index may be better!
- Equality queries and duplicates:
  - Clustering on E.hobby helps!

```
SELECT E.dno
  FROM Emp E
 WHERE E.age>40
```

```
SELECT E.dno,  COUNT (*)
  FROM Emp E
 WHERE E.age>10
GROUP BY E.dno
```

```
SELECT E.dno
  FROM Emp E
 WHERE E.hobby=Stamps
```

Index-Only Plans

Answer query without going to heap file!

```
SELECT D.mgr
  FROM Dept D, Emp E
 WHERE D.dno=E.dno
```

<E.dno>

```
SELECT D.mgr, E.eid
  FROM Dept D, Emp E
 WHERE D.dno=E.dno
```

<E.dno,E.eid>

```
SELECT E.dno, COUNT(*)
  FROM Emp E
 GROUP BY E.dno
```

<E.dno>

```
SELECT E.dno, MIN(E.sal)
  FROM Emp E
 GROUP BY E.dno
```

<E.dno,E.sal>
B-tree trick!

```
SELECT AVG(E.sal)
  FROM Emp E
 WHERE E.age=25 AND
   E.sal BETWEEN 3000 AND 5000
```

<E. age,E.sal>
or
<E.sal, E.age>

Index Tuning “Wizards”

- A number of RDBMSs now have automated index advisors
    - Some info in Section 20.6 of the book
- Basic idea:
    - Train on a workload of queries
        - Possibly based on logging what’s been going on
    - Use the optimizer cost metrics to estimate the cost of the workload over different choices of sets of indexes
    - Enormous # of different choices of sets of indexes:
        - Heuristics to help this go faster

Tuning Queries and Views

- If a query runs slower than expected:
  - check if an index needs to be re-clustered, or if statistics are too old.

- Sometimes, the DBMS may not be executing the plan you had in mind.
- Common areas where optimizers are sub-par:
  - Selections involving **null values** (bad selectivity estimates)
  - Selections involving **arithmetic or string expressions** (ditto)
  - Selections involving **OR** conditions (ditto)
  - Complex **subqueries** (lack of flattening)
  - Failed **cost estimation** (a common problem in large queries)
  - **Lack of evaluation features** like index-only strategies or certain join methods.

- Check the plan that is being used!
- Then adjust the choice of indexes or **rewrite the query/view.**
  - E.g. check via SQL `EXPLAIN` command
  - Many systems rewrite for you under the covers (e.g. DB2)
    - Can be confusing and/or helpful!

Forcing the optimizer

- Many DBMSs allow you to override or “hint” the optimizer if you believe it’s messing up
  - MS SQL Server allows hints on specific parts of your query (each Join, Union, etc.).
    - Also allows you to specify an entire plan in a special XML syntax.
  - PostgreSQL folks argue that this is a bad idea, and offer some other controls instead:
    - E.g. limit the number of tables being reordered in the FROM clause during dynamic programming
    - E.g. disable certain physical operators so the optimizer won’t choose them
  - Etc.

More Guidelines for Query Tuning

- Minimize the use of DISTINCT: don’t need it if duplicates are acceptable, or if answer contains a key.
- Minimize the use of GROUP BY and HAVING:

```
SELECT MIN (E.age)
  FROM Employee E
 GROUP BY E.dno
HAVING E.dno=102
```

```
SELECT MIN (E.age)
  FROM Employee E
 WHERE E.dno=102
```

- Consider DBMS use of index when writing math:
  - E.age = 2*D.age might only match index on E.age!
- Of course a good optimizer should do all these things for you!
  - But many optimizers have gaps, esp in open source

Guidelines for Query Tuning (Contd.)

- Avoid using intermediate relations:

```
SELECT E.dno,AVG(E.sal)
  FROM Emp E, Dept D
 WHERE E.dno=D.dno
   AND D.mgrname='Joe'
 GROUP BY E.dno
```

```
SELECT * INTO Temp
  FROM Emp E, Dept D
 WHERE E.dno=D.dno
   AND D.mgrname='Joe'
```

and

```
SELECT T.dno, AVG(T.sal)
  FROM Temp T
 GROUP BY T.dno
```

- Does not materialize the intermediate reln Temp.
- If there is a dense B+ tree index on Emp <*dno, sal*>
  - an index-only plan can be used
    to avoid retrieving Emp tuples in the left query!

Points to Remember

- Want to understand DB design (tables, indexes)?
  - Must understand query optimization
- Three parts to optimizing a query:
  - Plan space
    - E.g., left-deep plans only
    - avoid Cartesian products.
    - Prune plans with interesting orders separate from unordered plans
  - Cost Estimation
    - Output cardinality and cost for each plan node.
    - Key issues: Statistics, indexes, operator implementations.
  - Search Strategy
    - we learned “bottom-up” dynamic programming

Points to Remember, cont

- Single-relation queries:
  - All access paths considered, cheapest is chosen.
  - *Issues*:
    - Selections that *match* index
    - Whether index key has all needed fields
    - Whether index provides tuples in an interesting order.

More Points to Remember

- Multiple-relation queries:
  - All single-relation plans are first enumerated.
    - Selections/projections considered as early as possible.
  - Use best 1-way plans to form 2-way plans. Prune losers.
  - Use best (i-1)-way plans and best 1-way plans to form i-way plans
  - At each level, for each subset of relations, retain:
    - Best plan for each interesting order (including no order)

Summary

- Optimization is the reason for the lasting power of the relational system
- But it is primitive in some SQL databases, and in the Big Data stack
- New areas: many!
  - Smarter statistics (fancy histograms, "sketches")
  - Auto-tuning statistics
  - Adaptive runtime re-optimization (e.g. *Eddies*)
  - Multi-query optimization
  - Parallel scheduling issues