CS460: Intro to Database Systems

Class 21: Relational Query Optimization

Instructor: Manos Athanassoulis

https://midas.bu.edu/classes/CS460/

Query Optimization

Overview

Readings: Chapter 12.4

Query optimization

Cost estimation

Plan enumeration and costing

System R strategy

Review of Query Processing

Implementation of single Relational Operations

Choices depend on indexes, memory, stats,…

Joins

- Blocked nested loops:
  - simple, exploits extra memory
- Indexed nested loops:
  - best if one relation small and one indexed
- Sort/Merge Join
  - good with small amount of memory, bad with duplicates
- Hash Join
  - fast (enough memory), bad with skewed data

Query Optimization

Typically many methods of executing a given query, all giving same answer

Cost of alternative methods often varies enormously

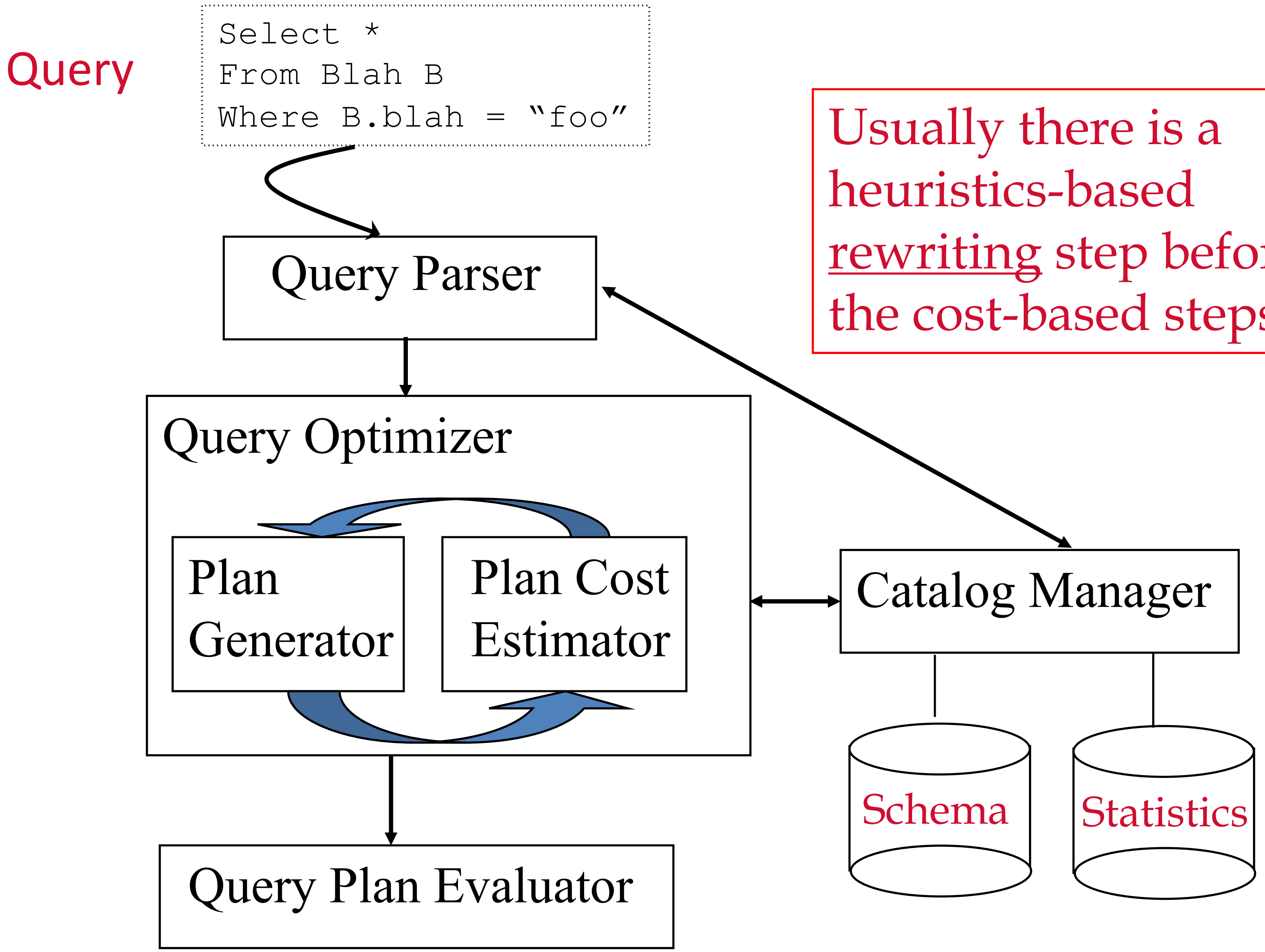
Desirable to find a low-cost execution strategy

We will cover:

- Relational algebra equivalences
- Cost estimation
  - Result size estimation and reduction factors
  - Statistics and Catalogs
- Enumerating alternative plans

Will focus on "System R"-style optimizers

Refresh: Query execution



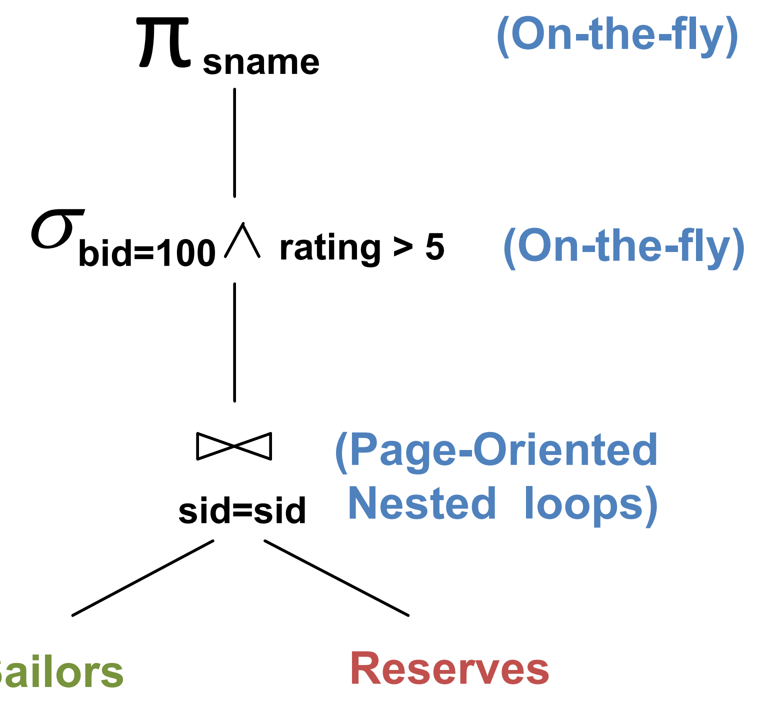
Usually there is a heuristics-based rewriting step before the cost-based steps.

Query Plans

A tree, with relational algebra operators as nodes

Each operator labeled with choice of algorithm

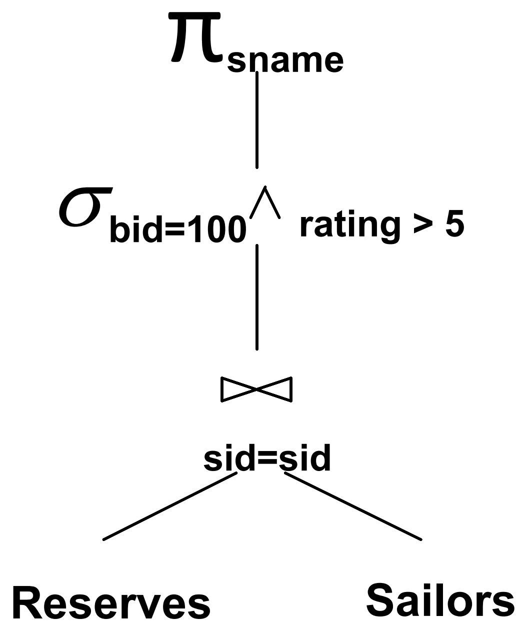
Plan:


π_{sname}
(On-the-fly)
$\sigma_{\text{bid=100} \wedge \text{rating > 5}}$
(On-the-fly)
$\bowtie_{\text{sid=sid}}$
(Page-Oriented Nested loops)
Sailors
Reserves

By convention, *outer* is on *left*.

Iterator Interface

A note on implementation:



Relational operators at nodes support uniform *iterator* interface:

open( ), get_next( ), close( )

Unary Operators – On open() call open() on child

Binary Operators – call open() on left child then on right

Query Optimization Overview

A Query:

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

To optimize:

1. Query first broken into "blocks"
2. Each block converted to relational algebra
3. Then, for each block, several alternative query plans are considered
4. Plan with lowest estimated cost is selected

A Familiar Schema for Examples

Sailors (*sid*: integer, *sname*: string, *rating*: integer, *age*: real)
Reserves (*sid*: integer, *bid*: integer, *day*: dates, *rname*: string)
Boats (*bid*: integer, *bname*: string, *color*: string)

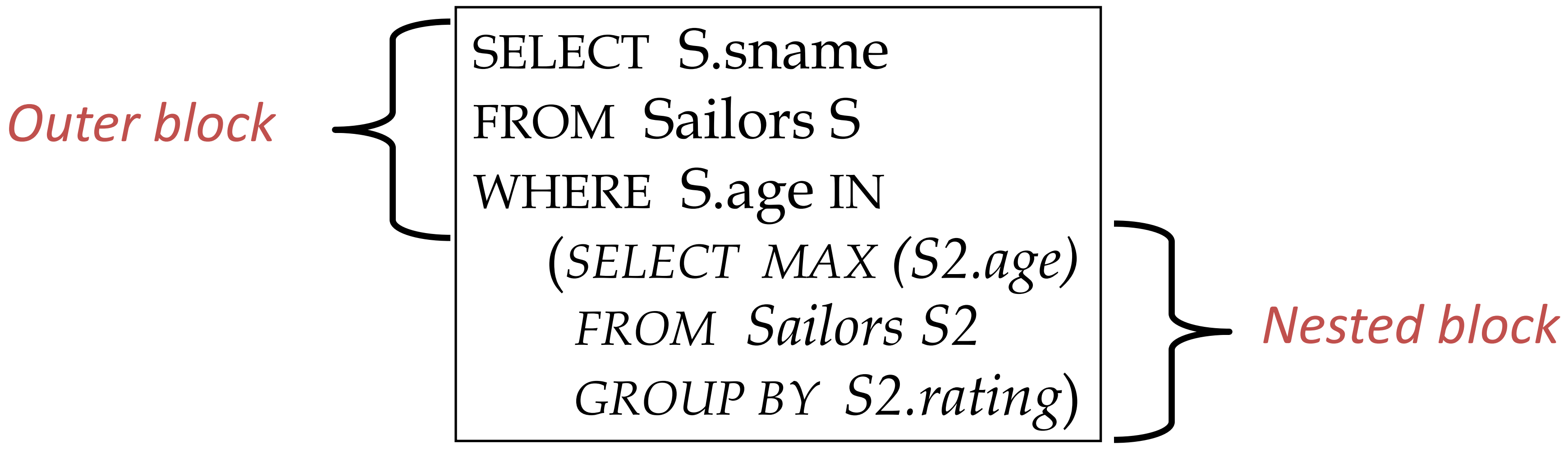
Query Optimization

Step 1: Break query into Query Blocks

Query block = unit of optimization

Nested blocks are usually treated as calls to a subroutine, made once per outer tuple

- (This is an over-simplification, but serves for now)



```
SELECT  S.sname
FROM  Sailors S
WHERE  S.age IN
     (SELECT  MAX (S2.age)
      FROM  Sailors S2
      GROUP BY  S2.rating)
```

Step 2: Converting query block into relational algebra expression

```
SELECT  S.sid
FROM  Sailors S, Reserves R, Boats B
WHERE  S.sid = R.sid AND R.bid = B.bid AND B.color = "red"
```

$$\pi_{S.sid}(\sigma_{B.color = \text{“red”}}(\text{Sailors} \bowtie \text{Reserves} \bowtie \text{Boats}))$$

A Fancier Example …

```
SELECT  S.sid, MIN (R.day)
FROM  Sailors S, Reserves R, Boats B
WHERE  S.sid = R.sid AND R.bid = B.bid AND B.color = “red”
AND S.rating = ( SELECT MAX (S2.rating) FROM Sailors S2)
GROUP BY S.sid
HAVING COUNT (*) >= 2
```

For each sailor with the highest rating (over all sailors), and at least two reservations for red boats, find the sailor id and the earliest date on which the sailor has a reservation for a red boat

Example translated to relational algebra

```
SELECT  S.sid, MIN (R.day)
FROM  Sailors S, Reserves R, Boats B
WHERE  S.sid = R.sid AND R.bid = B.bid AND B.color = “red”
AND S.rating = ( SELECT MAX (S2.rating) FROM Sailors S2)
GROUP BY S.sid
HAVING COUNT (*) >= 2
```

Inner Block

$$\pi_{S.sid,\ MIN(R.day)}\ (\text{HAVING}_{COUNT(*)>2}\ (\text{GROUP BY}_{S.Sid}\ (\sigma_{B.color = \text{“red”} \wedge S.rating = val}(\text{Sailors} \bowtie \text{Reserves} \bowtie \text{Boats}))))$$

Select-Project-Join Optimization

Core of every query is a select-project-join (SPJ) expression

Other aspects, if any, carried out on result of SPJ core:

- Group By (either sort or hash)
- Having (apply filter on-the-fly)
- Aggregation (easy once grouping done)
- Order By (sorting is the name of the game)

Not much room to exploit equivalences on non-SPJ parts

Focus on optimizing SPJ core

Relational Algebra Equivalences

- *Selections*: $\sigma_{c_1 \wedge \cdots \wedge c_n}(R) \equiv \sigma_{c_1}\left(\ldots\left(\sigma_{c_n}(R)\right)\right)$ $(Cascade)$

  $\sigma_{c_1}\left(\sigma_{c_2}(R)\right) \equiv \sigma_{c_2}\left(\sigma_{c_1}(R)\right)$ $(Commute)$

- *Projections:* $\pi_{a_1}(R) \equiv \pi_{a_1}\left(\ldots\left(\pi_{a_n}(R)\right)\right)$ $(Cascade)$

  a_i is a set of attributes of R and $a_i \subseteq a_{i+1}$ for $i = 1 \ldots n-1$

- These equivalences allow us to 'push' selections and projections ahead of joins.

Examples …

$\sigma_{age<18 \land rating>5}$ (Sailors)

$\leftrightarrow \sigma_{age<18} (\sigma_{rating>5}$ (Sailors))

$\leftrightarrow \sigma_{rating>5} (\sigma_{age<18}$ (Sailors))

~~$\pi_{age,rating}$ (Sailors) $\leftrightarrow \pi_{age} (\pi_{rating}$ (Sailors))~~ (??)

$\pi_{age,rating}$ (Sailors) $\leftrightarrow \pi_{age,rating} (\pi_{age,rating,sid}$ (Sailors))

Another Equivalence

A projection commutes with a selection that only uses attributes retained by the projection

$$\pi_{\text{age, rating, sid}} (\sigma_{\text{age}<18 \wedge \text{rating}>5} (\text{Sailors}))$$

$$\leftrightarrow \sigma_{\text{age}<18 \wedge \text{rating}>5} (\pi_{\text{age, rating, sid}} (\text{Sailors}))$$

Equivalences Involving Joins

$$R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T \qquad (Associative)$$

$$(R \bowtie S) \equiv (S \bowtie R) \qquad (Commutative)$$

These equivalences allow us to choose different join orders

Mixing Joins with Selections & Projections

Converting selection + cross-product to join

$\sigma_{S.sid = R.sid}$ (Sailors x Reserves)

$\leftrightarrow$ Sailors $\bowtie_{S.sid = R.sid}$ Reserves

Selection on just attributes of S commutes with R$\bowtie$S

$\sigma_{S.age<18}$ (Sailors $\bowtie_{S.sid = R.sid}$ Reserves)

$\leftrightarrow$ ($\sigma_{S.age<18}$ (Sailors)) $\bowtie_{S.sid = R.sid}$ Reserves

We can also “push down” projection (*but be careful…*)

$\pi_{S.sname}$ (Sailors $\bowtie_{S.sid = R.sid}$ Reserves)

$\leftrightarrow$ $\pi_{S.sname}$ ($\pi_{sname,sid}$(Sailors) $\bowtie_{S.sid = R.sid}$ π_{sid}(Reserves))

What do you think? True or False?

1. $R \times S = S \times R$
2. $(R \times S) \times T = R \times (S \times T)$
3. $\sigma_p(R \cup S) = \sigma_p(R) \cup S$
4. $R \cup S = S \cup R$
5. $\sigma_p(R - S) = R - \sigma_p(S)$
6. $R \cup (S \cup T) = (R \cup S) \cup T$
7. $\sigma_{R.p \vee S.q}(R \bowtie S) =$

$$\left[(\sigma_p R) \bowtie S\right] \cup \left[R \bowtie (\sigma_q S)\right]$$

Think about them and discuss in piazza!!!

Query Rewriting

Modern DBMS's may rewrite queries before the optimizer sees them

Main purpose: de-correlate and/or flatten nested subqueries

De-correlation:

- Convert correlated subquery into uncorrelated subquery

Flattening:

- Convert query with nesting into query w/o nesting

Example: Decorrelating a Query

```
SELECT  S.sid
FROM  Sailors S
WHERE EXISTS
   (SELECT  *
    FROM  Reserves R
    WHERE  R.bid=103
    AND  R.sid=S.sid)
```

Equivalent uncorrelated query:

```
SELECT  S.sid
FROM Sailors S
WHERE  S.sid IN
   (SELECT  R.sid
    FROM  Reserves R
    WHERE  R.bid=103)
```

Advantage: nested block only needs to be executed once (rather than once per S tuple)

Example: “Flattening” a Query

```
SELECT  S.sid
FROM Sailors S
WHERE  S.sid IN
   (SELECT  R.sid
    FROM  Reserves R
    WHERE  R.bid=103)
```

Equivalent non-nested query:

```
SELECT  S.sid
FROM Sailors S, Reserves R
WHERE  S.sid=R.sid
   AND R.bid=103
```

Advantage: can use a join algorithm + optimizer can select among join algorithms & reorder freely

Query transformations: Summary

Before optimizations, queries are flattened and de-correlated

Queries are first broken into blocks

Blocks are converted to relational algebra expressions

Equivalence transformations are used to push down selections and projections

Query Optimization

Overview

Query optimization

Cost estimation

Readings: Chapter 15.2

Plan enumeration and costing

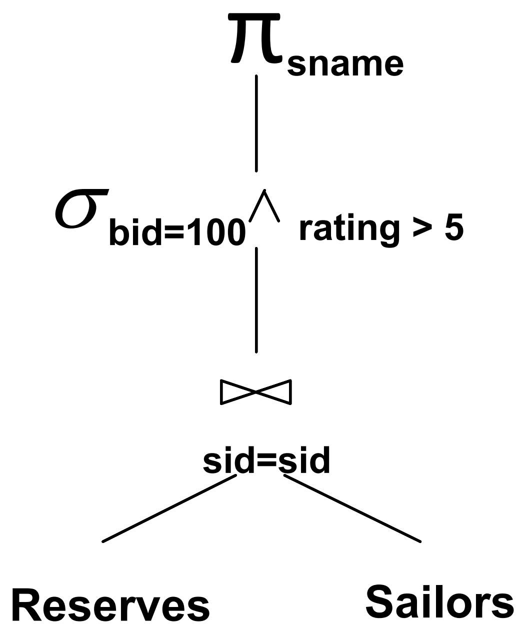
System R strategy

Recall: Query Optimization Overview

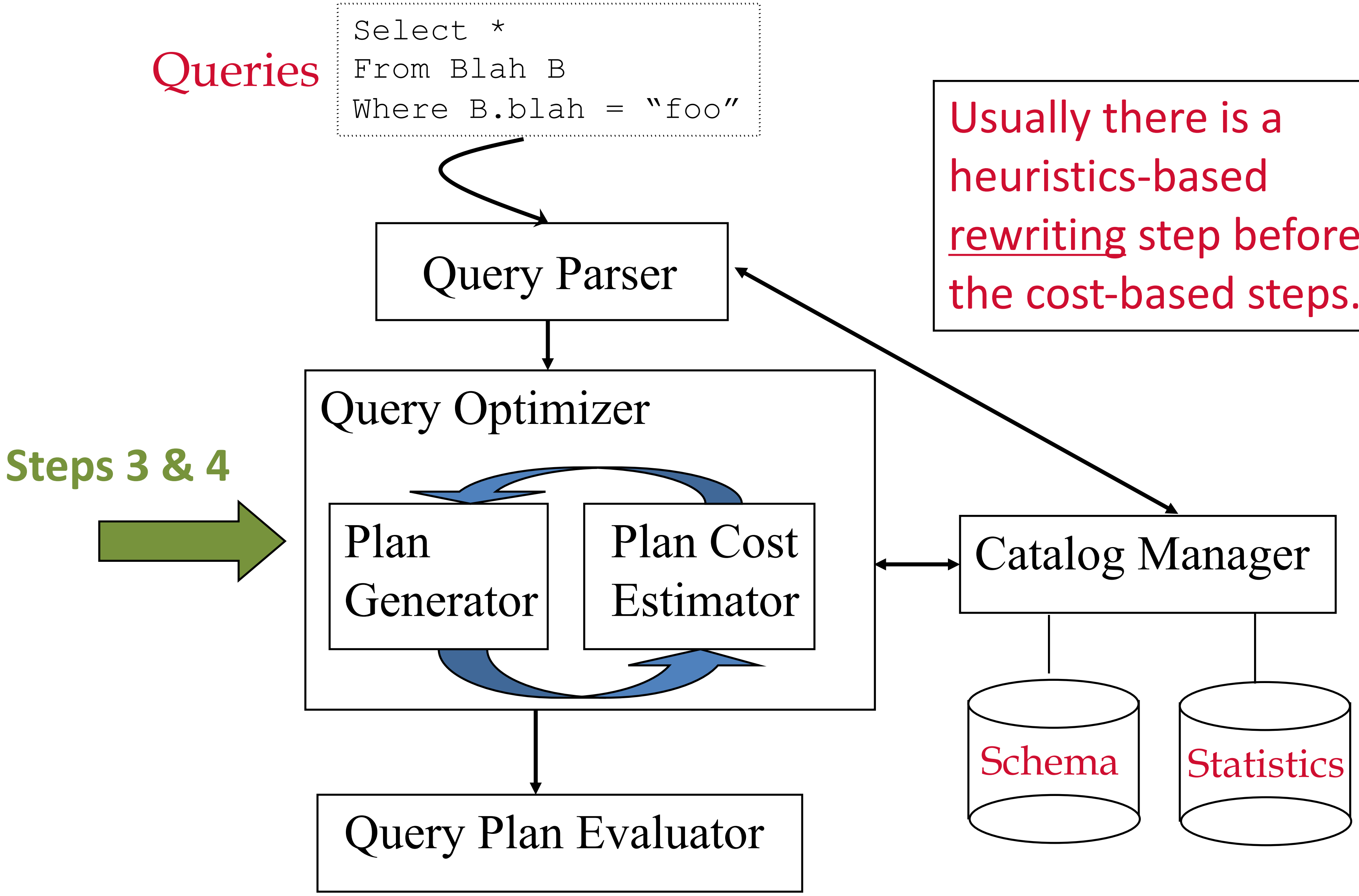
1. Query first broken into "blocks"
2. Each block converted to relational algebra
3. Then, for each block, several alternative query plans are considered
4. Plan with lowest estimated cost is selected

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

$\pi_{(sname)}\sigma_{(bid=100 \wedge rating > 5)}$ (Reserves $\bowtie$ Sailors)



Cost-based Query Sub-System



Usually there is a heuristics-based rewriting step before the cost-based steps.

Two Main Issues

1. For a given query, what plans are considered?

   Algorithm to search plan space for cheapest (estimated) plan.

2. How is the cost of a plan estimated?

Ideally: Want to find best plan.

Reality: Avoid worst plans!

Highlights of System R Optimizer

Impact:

- Most widely used currently; works well for < 10 joins

Cost estimation:

- Very inexact, but works okay in practice
- Statistics, maintained in system catalogs, used to estimate cost of operations and result sizes
- Considers combination of CPU and I/O costs
- More sophisticated techniques known now

Plan Space: Too large, must be pruned

- Only the space of *left-deep plans* is considered
- Cross products are avoided

Schema for Examples

Sailors (*sid*: integer, *sname*: string, *rating*: integer, *age*: real)
Reserves (*sid*: integer, *bid*: integer, *day*: dates, *rname*: string)

Reserves:

- tuple size is 40 bytes, 100 tuples per page, 1000 pages, 100 distinct bids

Sailors:

- tuple size is 50 bytes, 80 tuples per page, 500 pages, 10 Ratings, 40,000 sids

Cost Estimation

For each plan considered, must estimate cost:

- Must estimate *cost* of each operation in plan tree.
  - Depends on input cardinalities
  - We've already discussed how to estimate the cost of operations (sequential scan, index scan, joins, etc.)
- Must estimate *size of result* for each operation in tree!
  - 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 + *factor* * #CPU instructions

Statistics and Catalogs

Need information about the relations and indexes involved. *Catalogs* typically contain at least:

- # tuples (**NTuples**) and # pages (**NPages**) per relation
- # distinct key values (**NKeys**) for each index
- low/high key values (**Low/High**) for each index
- Index height (**IHeight**) for each tree index
- # index pages (**INPages**) for each index

Statistics in catalogs are updated periodically

- Updating whenever data changes is too expensive; lots of approximation anyway, so slight inconsistency is OK

More detailed information (e.g., histograms of the values in some field) are sometimes stored

Size Estimation and Reduction Factors

Consider a query block:

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

Maximum # tuples in result is the product of the cardinalities of relations in the FROM clause

Reduction factor (RF) associated with each *term* reflects the impact of the *term* in reducing result size

RF is usually called "selectivity"

Result Size Estimation for Selections

Result cardinality = Max # tuples * product of all RF's
(Implicit assumption that values are uniformly distributed and *terms* are independent!)

Term *col=value* (given index I on *col*)

RF = *1/NKeys(I)*

Term *col>value*

RF = *(High(I)-value)/(High(I)-Low(I))*

Note: if missing indexes, assume RF = 1/10

Result Size Estimation for Joins

Q: Given a join of R and S, what is the range of possible result sizes (in #of tuples)?

- Hint: what if R_cols ⋂ S_cols = ∅?
- R_cols ⋂ S_cols is a key for R (and a Foreign Key in S)?

Result Size Estimation for Joins

General case: R_cols ⋂ S_cols = {A} (and A is key for neither)

- If NKeys(A,**S**) > NKeys(A,**R**)
  - Assume S values are a superset of R values, so each R value finds a matching value in S
  - Estimate each tuple r of R generates NTuples(S)/NKeys(A,S) result tuples, so…

  est_size = NTuples(R) * NTuples(S)/NKeys(A,**S**)

- Else, if NKeys(A,**R**) > NKeys(A,**S**) … symmetric argument, yielding:

  est_size = NTuples(R) * NTuples(S)/NKeys(A,**R**)

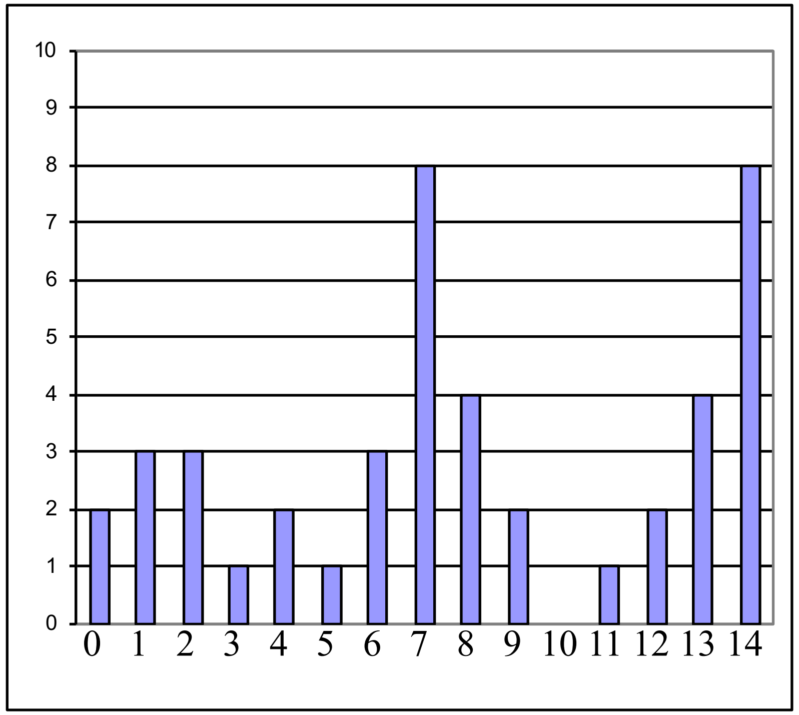
- Overall:

  est_size = NTuples(R)*NTuples(S)/MAX{NKeys(A,**S**), NKeys(A,**R**)}

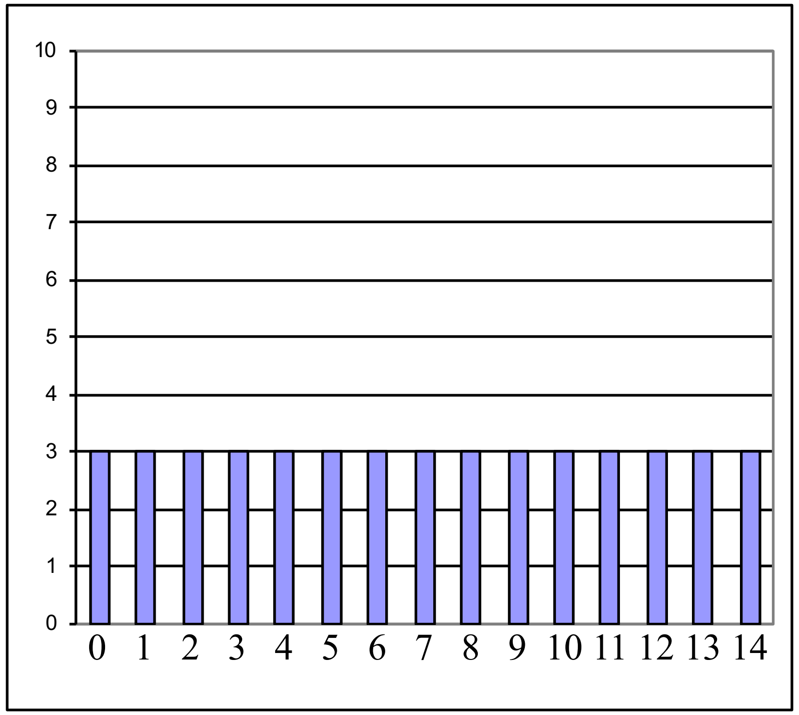
On the Uniform Distribution Assumption

Assuming uniform distribution is rather crude

Distribution D



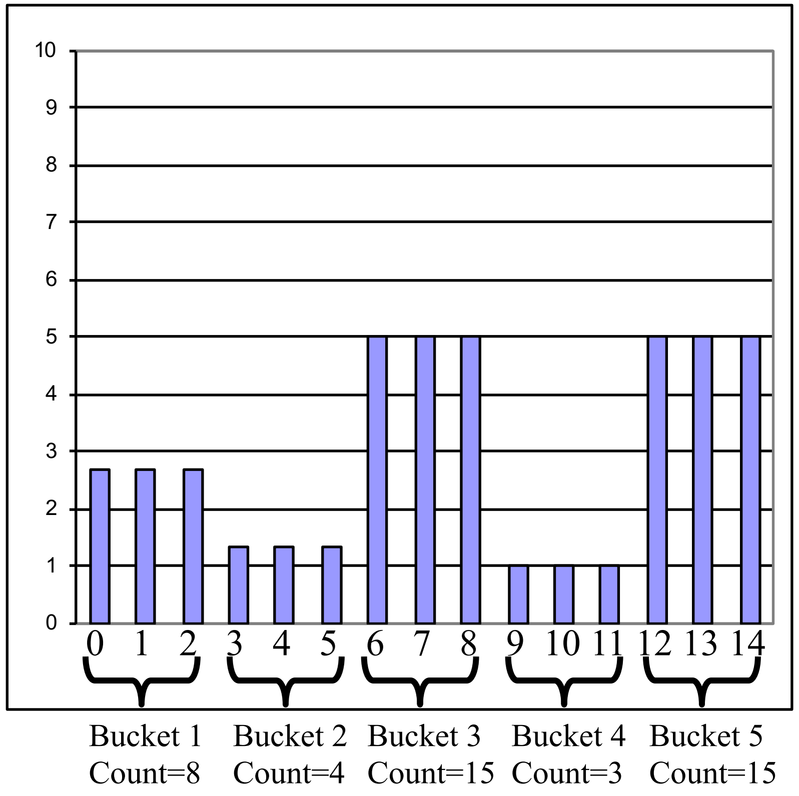
Uniform distribution approximating D



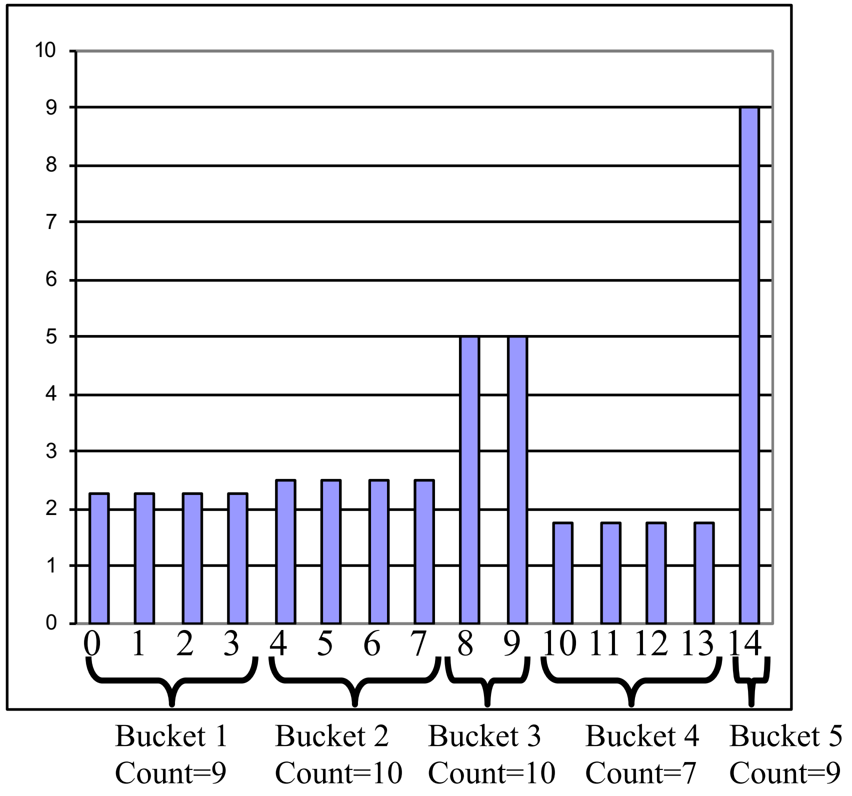
Histograms

For better estimation, use a *histogram*

Equiwidth histogram



Equidepth histogram



Cost estimation: Summary

The costs of possible strategies vary widely

Estimate result sizes using statistics

Estimate costs of each operator

Focus on optimizing select-project-join (SPJ) blocks

Query Optimization

Overview

Query optimization

Cost estimation

Plan enumeration and costing

Readings: Chapter 15.4

System R strategy

Enumeration of Alternative Plans

There are two main cases:

- Single-relation plans
- Multiple-relation plans

For queries over a single relation:

- Each available access path (file scan / index) is considered, and the one with the least estimated cost is chosen
- The different operations are essentially carried out together (e.g., if an index is used for a selection, projection is done for each retrieved tuple)

Cost Estimates for Single-Relation Plans

Index I on primary key matches selection:

- *Cost is Height(I)+1 for a B+ tree, about 2.2 for hash index*

Clustered index I matching one or more selects:

- *(NPages(I)+NPages(R)) * product of RF's of matching selects.*

Non-clustered index I matching one or more selects:

- *(NPages(I)+NTuples(R)) * product of RF's of matching selects*

Sequential scan of file:

- *NPages(R)*

- ***Note:*** *Must also charge for duplicate elimination if required*

Example

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

Reminder: Sailors has 500 pages, 40000 tuples

If we have an index on *rating*:

- Cardinality: (1/NKeys(I)) * NTuples(S) = (1/10)*40000 tuples retrieved
- Clustered index: cost = (1/NKeys(I)) * (NPages(I)+NPages(S)) = (1/10) * (50+500) = 55 pages retrieved.
- Unclustered index: cost = (1/NKeys(I)) * (NPages(I)+NTuples(S)) = (1/10) * (50+40000) = 4005 pages 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)

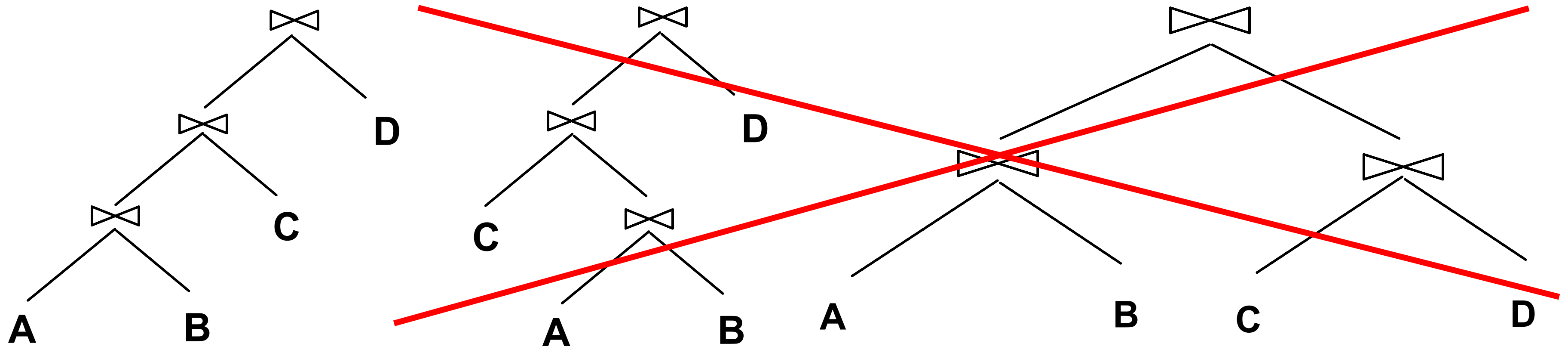
Queries Over Multiple Relations

As number of joins increases, number of alternative plans grows rapidly → *need to restrict search space*

Fundamental decision in System R:

only left-deep join trees are considered

- Left-deep trees allow us to generate all *fully pipelined* plans
  - Intermediate results are not written to temporary files
  - Not all left-deep trees are fully pipelined (e.g., SM join)



Plan Enumeration – The Hard Way

1. Select order of relations (the only degree of freedom for left-deep plans)
   - maximum possible orderings = N! (but no X-products)
2. For each join, select join algorithm
3. For each input relation, select access method

Q: How many plans for a query over N relations?

Back-of-envelope calculation:

- With 3 join algorithms, I indexes per relation:
  # plans ≈ $[N!] * [3^{(N-1)}] * [(I + 1)^N]$
- Suppose N = 3, I = 2: # plans ≈ $3! * 3^2 * 3^3 = 1458$ plans

For each candidate plan, must estimate cost

Query optimization is NP-complete

Plan Enumeration Example

```
SELECT  S.sname, B.bname, R.day
FROM  Sailors S, Reserves R, Boats B
WHERE  S.sid = R.sid AND R.bid = B.bid
```

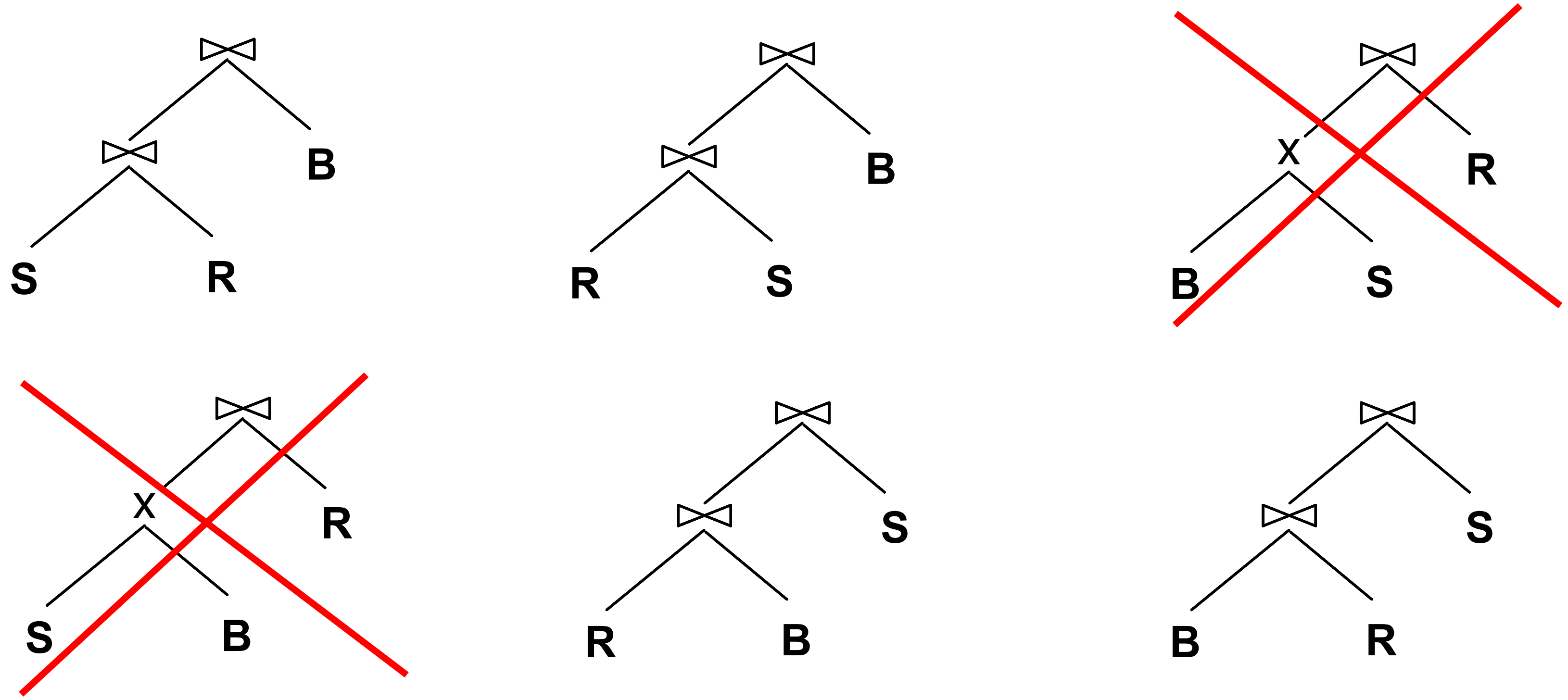
Let’s assume:

- Two join algorithms to choose from:
  - Hash-Join / NL-Join (page-oriented or Index-NL-Join)
- Unneeded columns removed at each stage
- Non-clustered B+Tree index on R.sid; no other indexes
- R.sid index has 50 pages
- S has 500 pages, 80 tuples/page
- R has 1000 pages, 100 tuples/page
- B has 10 pages
- 100 $R \bowtie S$ tuples fit on a page

Candidate Plans

```
SELECT S.sname, B.bname, R.day
FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid
```

1. Enumerate relation orderings:

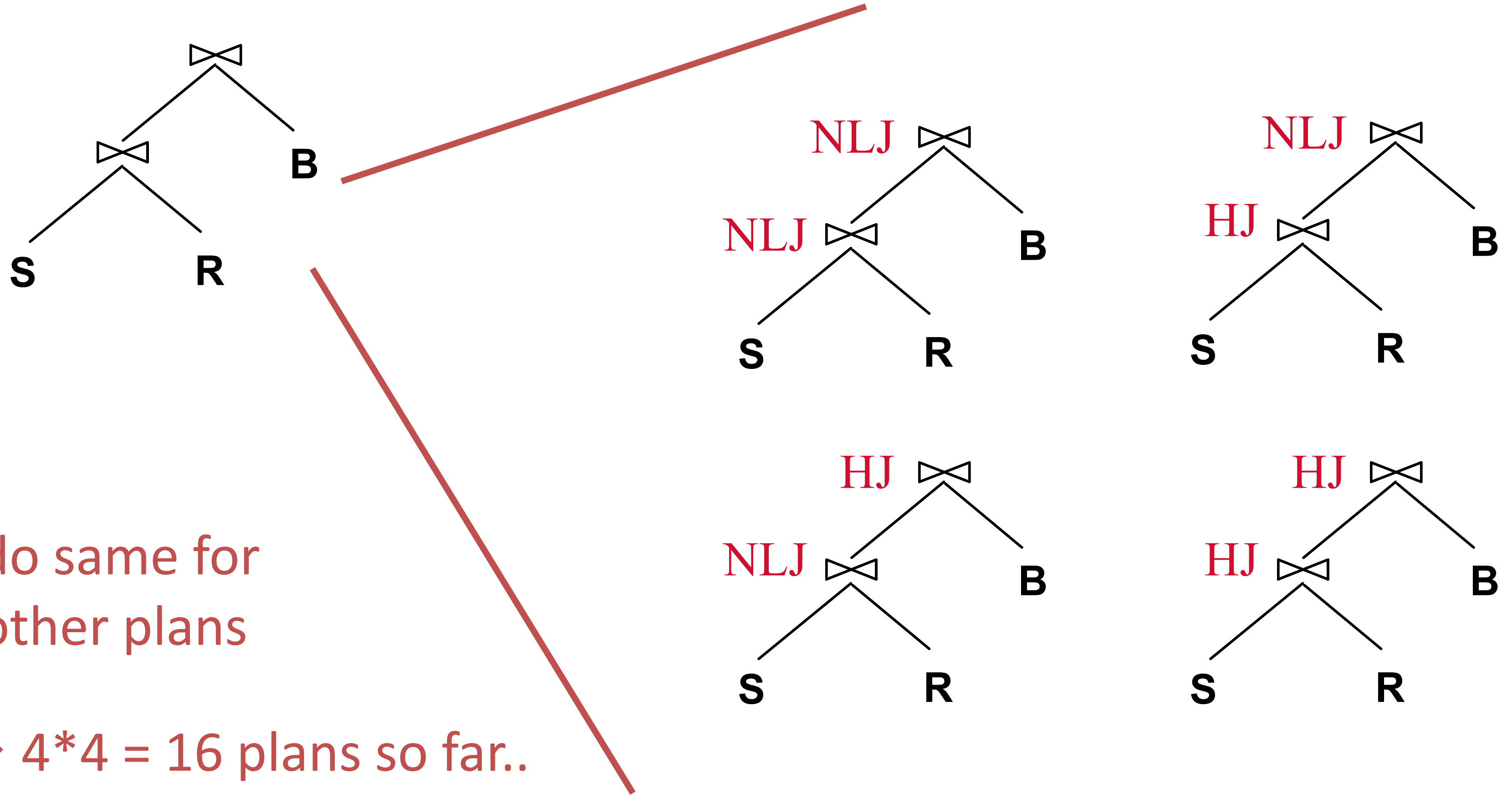


Prune plans with cross-products immediately!

Candidate Plans

```
SELECT  S.sname, B.bname, R.day
FROM  Sailors S, Reserves R, Boats B
WHERE  S.sid = R.sid AND R.bid = B.bid
```

2. Enumerate join algorithm choices:



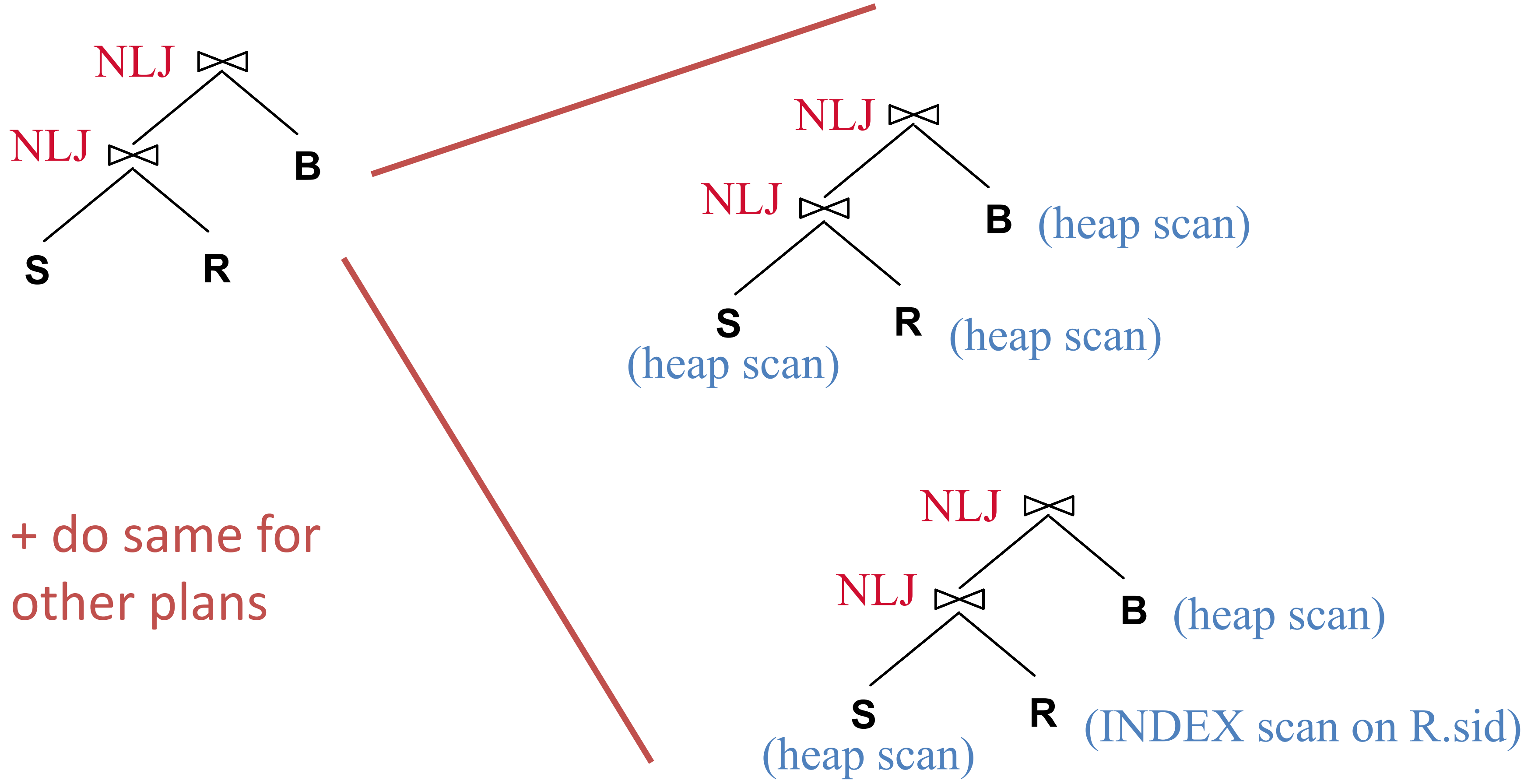
+ do same for
3 other plans

→ 4*4 = 16 plans so far..

Candidate Plans

```
SELECT  S.sname, B.bname, R.day
FROM  Sailors S, Reserves R, Boats B
WHERE  S.sid = R.sid AND R.bid = B.bid
```

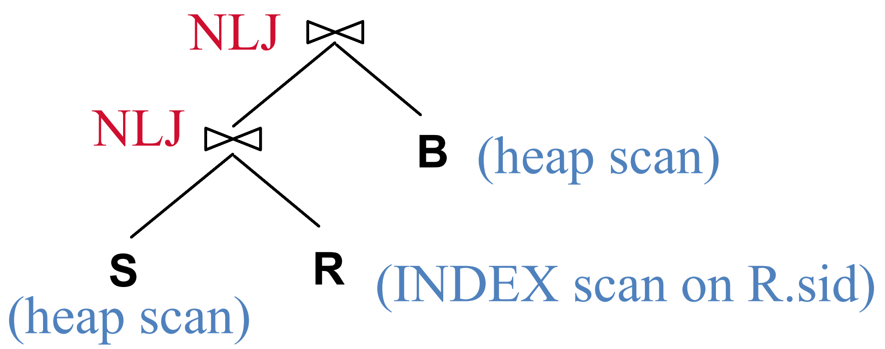
3. Enumerate access method choices:



+ do same for other plans

Now estimate the cost of each plan

Example:



R.sid index has 50 pages
|S|= 500 pg, 80 tuples/pg
|R|= 1000 pg, 100 tuples/pg
|B|= 10 pages
100 R ⋈ S tuples fit on a page
There are 40000 sids

Cost to join S with R

|S| + ((|S|*p_s) * cost of finding matching R tuples)

500 + 500*80 * (1/40000)(50[idx]+100,000) = 100,050

Size of S ⋈ R = NTuples(S)*NTuples(R)/distinct keys(sid) =100,000 tuples; 100,000/100 = 1000 pages

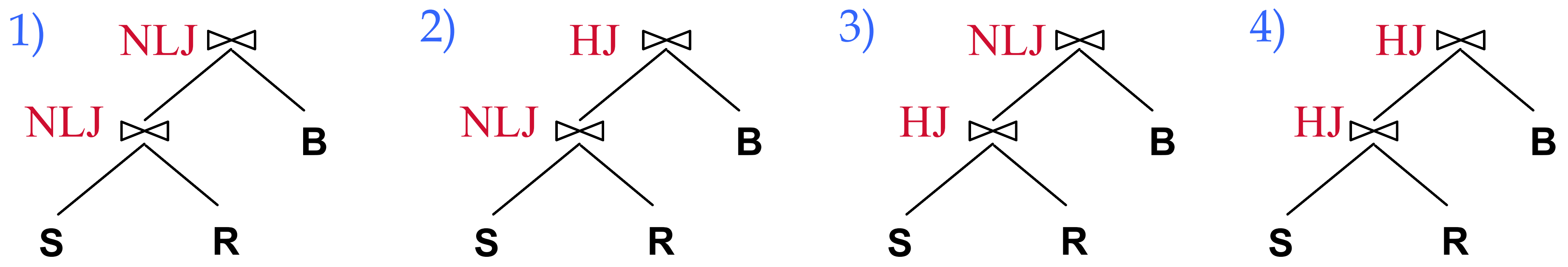
Cost to NL join with B = 1000 * 10 = 10000 (pipelined)

→ Total estimated cost = 500 + 100,050 + 10000 = 110,550

Now You Try …

S = Sailors
R = Reserves
B = Boats

Estimate the cost of each of these plans:



Relevant stats:

- S has 500 pages, 80 tuples/page
- R has 1000 pages, 100 tuples/page
- B has 10 pages
- 100 S $\bowtie$ R tuples fit on a page

Join algorithms:

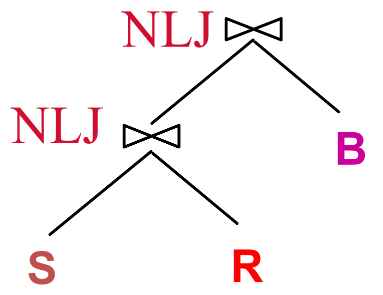
NLJ = page-oriented NL Join

- Scan left input + scan right input once per page in left input

HJ = hash-join (assume 2 passes)

- Scan both inputs + write both inputs in buckets + read all buckets

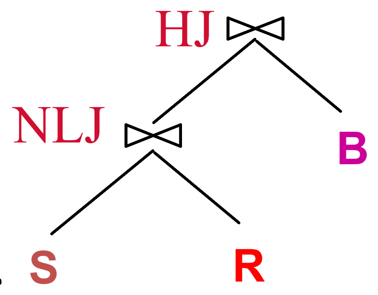
Answers …



Plan 1:

S ⋈ R size = 100,000 tuples; 1000 pages

Estimated cost = $\underbrace{500}_{\text{scan S}} + \underbrace{500(1000)}_{\text{join w/R}} + \underbrace{1000(10)}_{\text{join w/B}} = 510{,}500$

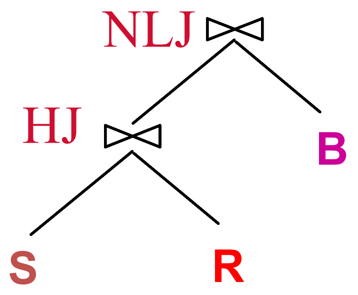


Plan 2:

S ⋈ R size = 100,000 tuples; 1000 pages

Estimated cost = $\underbrace{500}_{\text{scan S}} + \underbrace{500(1000)}_{\text{join w/R}} + \underbrace{2*1000 + 3*10}_{\text{join w/B}} = 502{,}530$

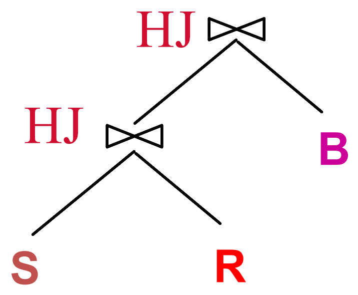
Answers …



Plan 3:

S ⋈ R size = 100,000 tuples; 1000 pages

Cost = $\underbrace{500}_{\text{scan S}} + \underbrace{2*500 + 3*1000}_{\text{join w/R}} + \underbrace{1000(10)}_{\text{join w/B}} = 14500$

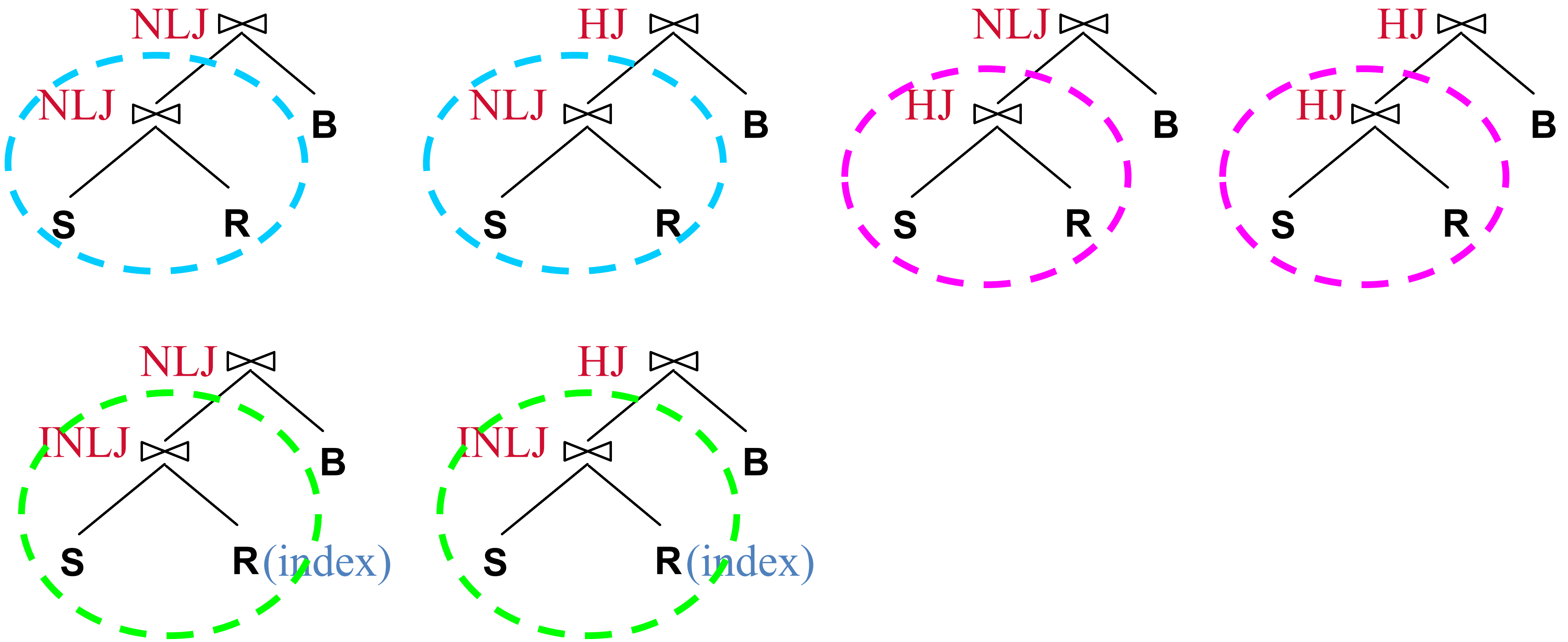


Plan 4:

S ⋈ R size = 100,000 tuples; 1000 pages

Cost = $\underbrace{500}_{\text{scan S}} + \underbrace{2*500 + 3*1000}_{\text{join w/R}} + \underbrace{2*1000 + 3*10}_{\text{join w/B}} = 6530$

Enumerated Plans (just the S-R-B ones)



Observe that many plans share common sub-plans (i.e., only upper part differs)

Notice Anything?

Much of the computation is redundant

Idea: when we estimate costs & result sizes of sub-plans, remember them.

Query Optimization

Overview

Query optimization

Cost estimation

Plan enumeration and costing

System R strategy

Readings: Chapter 15.6

Improved Strategy (used in System R)

Shared sub-plan observation suggests a better strategy:

Enumerate plans using N passes (N = # relations joined):

- Pass 1: Find best 1-relation plans for each relation
- Pass 2: Find best ways to join result of each 1-relation plan as outer to another relation *(All 2-relation plans.)*
- Pass N: Find best ways to join result of a (N-1)-relation plan as outer to the Nth relation *(All N-relation plans.)*

For each subset of relations, retain only:

- Cheapest subplan overall (possibly unordered), plus
- Cheapest subplan for each *interesting order* of the tuples

For each subplan retained, remember cost and result size estimates

A Note on ”Interesting Orders”

An intermediate result has an “interesting order” if it is sorted by any of:

- ORDER BY attributes
- GROUP BY attributes
- Join attributes of other joins

System R Plan Enumeration

A N-1 way plan is not combined with an additional relation unless there is a join condition between them (unless all predicates in WHERE have been used up)

- i.e., avoid Cartesian products if possible

Always push all selections & projections as far down in the plans as possible

- Usually a good strategy, as long as these operations are cheap

System R Plan Enumeration Example

```
SELECT S.sname, B.bname, R.day
FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid
```

This time let's assume:

- Two join algorithms to choose from:
  - Sort-Merge-Join / NL-Join (page-oriented or Index-NL-Join)
- Clustered B+Tree on S.sid (height=3; 500 leaf pages)
- S has 10,000 pages, 5 tuples/page
- R has 10 pages, 10 tuples/page
- B has 10 pages, 20 tuples/page
- 10 $R \bowtie S$ tuples fit on a page
- 10 $R \bowtie B$ tuples fit on a page

Pass 1 (single-relation subplans)

S: (a) heap scan or (b) scan index on S.sid

a) heap scan cost = 10,000

b) index scan cost = 500 + 10,000 = 10,500

Retain both, since (b) has “interesting order” by sid

R: heap scan only option

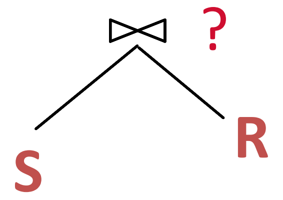
Cost = 10

B: heap scan only option

Cost = 10

Pass 2 (2-relation subplans)

Starting with S as outer



Heap scan-S as outer:

a) NL-Join with R, cost = 10,000 + 10,000(10) = 110,000

b) SM-Join with R, cost = 10,000 + 2*10,000 + 3*10 = 30,030

Index scan-S as outer:

c) NL-Join with R, cost = 10,500 + 10,000(10) = 110,500

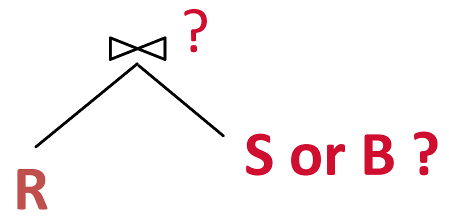
d) SM-Join with R, cost = 10,500 + 3*10 = 10,530

Retain (d) only

Note: best S$\bowtie$ R plan exploits "interesting order" of non-optimal subplan !

Pass 2 (continued)

Starting with R as outer



Join with S:

a) NL-Join with S, cost = 10 + 10(10,000) = 100,010

b) Index-NL-Join with Index-S, cost = 10 + 100*4 = 410

c) SM-Join with S, cost = 10 + 2*10 + 3*10,000 = 30,030

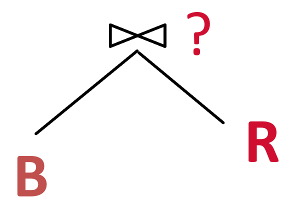
Join with B:

a) NL-Join with B, cost = 10 + 10(10) = 110

b) SM-Join with B, cost = 10 + 2*10 + 3*10 = 60

Pass 2 (continued)

Starting with B as outer

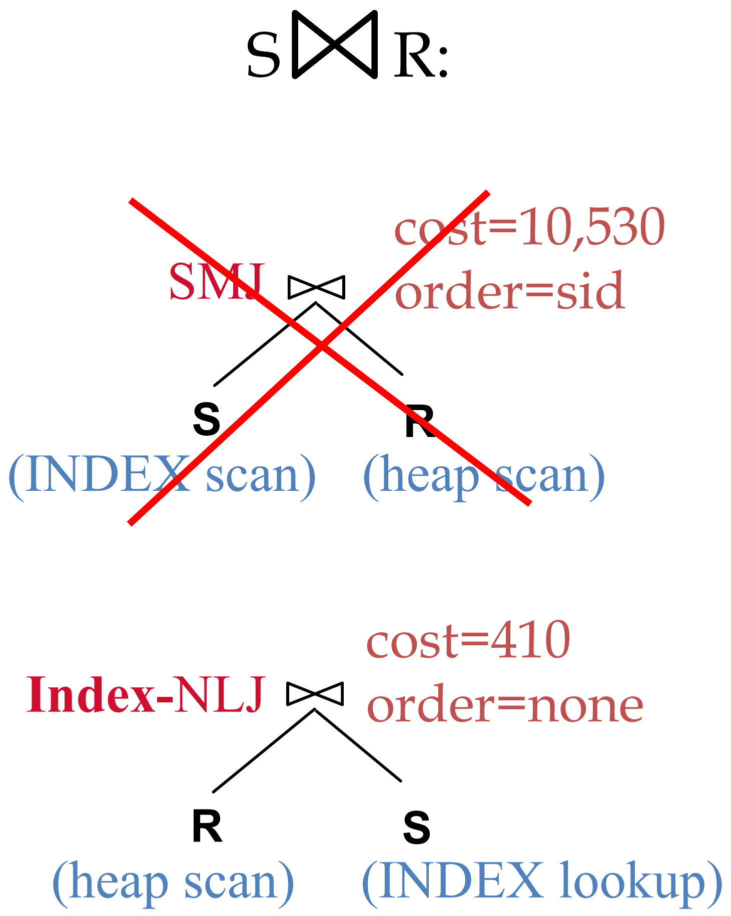


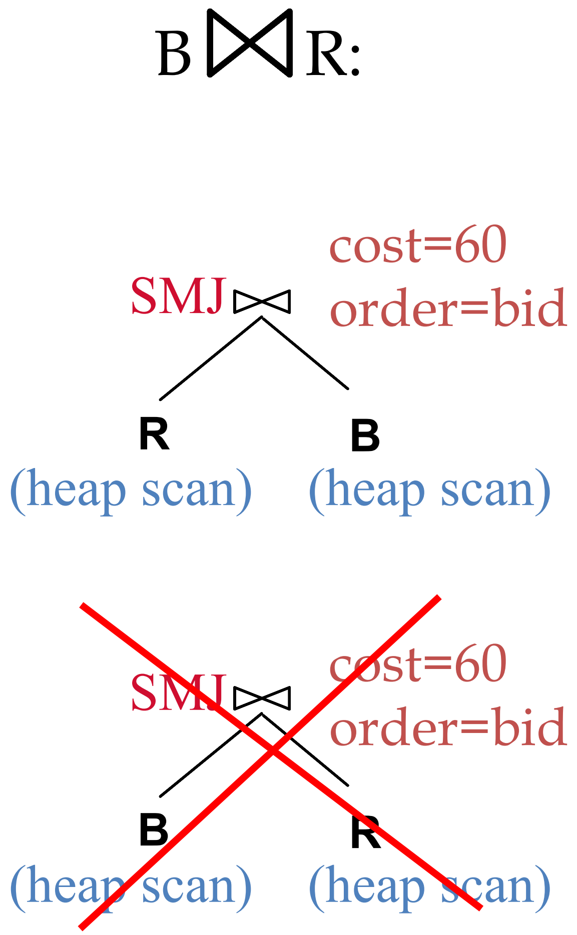
Join with R:

a) NL-Join with R, cost = 10 + 10(10) = 110

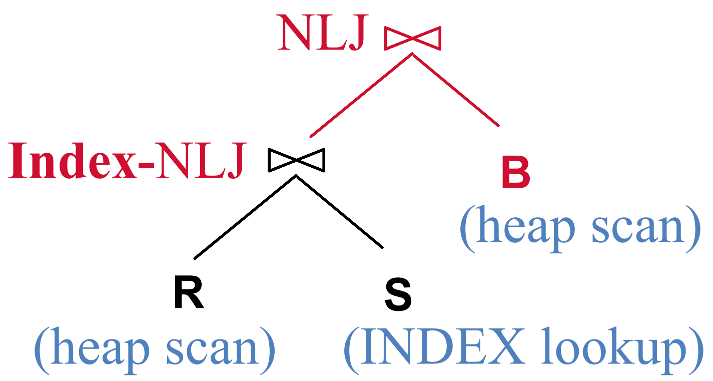
b) SM-Join with R, cost = 10 + 2*10 + 3*10 = 60

Further pruning of 2-relation subplans



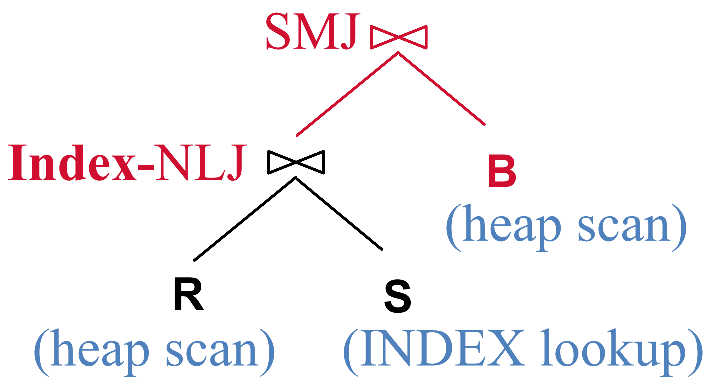


Pass 3 (3-relation subplans)



cost = 410 + 10(10) = 510

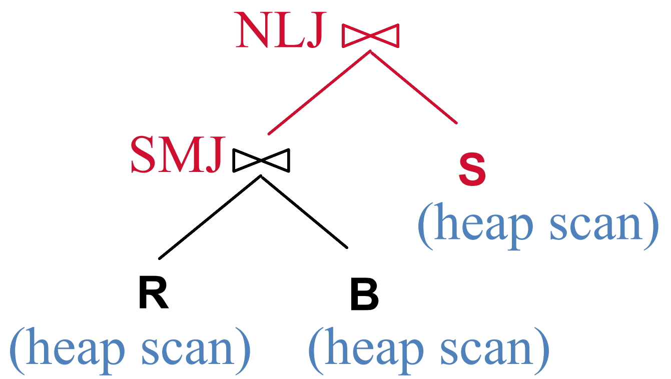
S ⋈ R subplan:
cost=410
order=none
result size = 10 pages



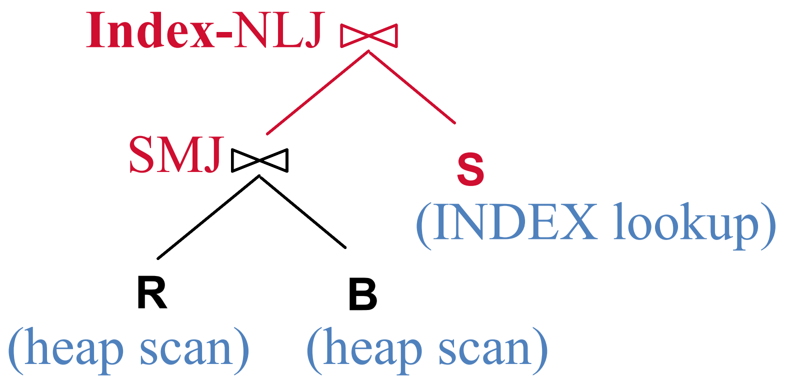
cost = 410 + 2*10 + 3*10 = 460

Pass 3 (continued)

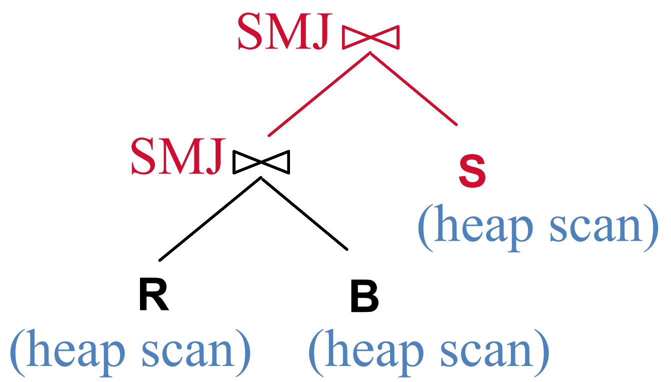
B⋈R subplan:
cost=60, order=bid
result size = 100 tuples (10 pages)



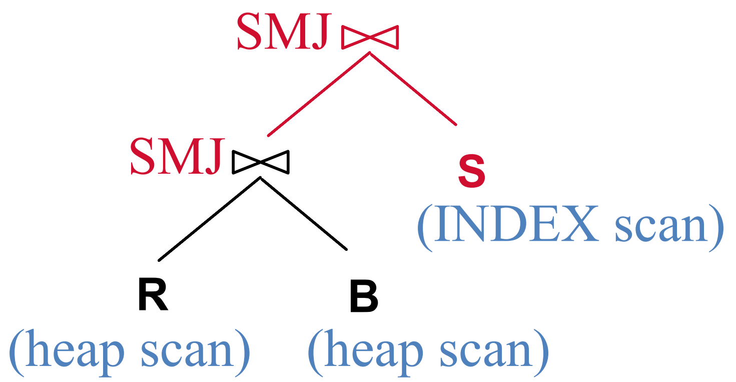
cost = 60 + 10(10,000) = 100,060



cost = 60 + 100*4 = 460

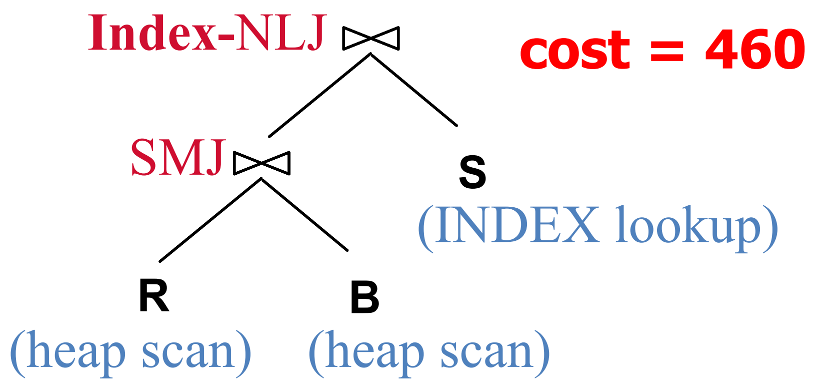


cost = 60 + 10*2 + 3*10,000 = 30,080



cost = 60 + 10*2 + 10,500 = 10,580

And the Winner is …



Observations:

– Best plan mixes join algorithms

– Worst plan had cost > 100,000

(exact cost unknown due to pruning)

Optimization yielded ~ **1000-fold improvement** over worst plan!

Some notes w.r.t. reality…

In spite of pruning plan space, this approach is still exponential in the # of tables

– Rule of thumb: works well for < 10 joins

In real systems, COST considered is:

#IOs + *factor* * #CPU Instructions

System R strategy: Summary

Enumerate plans using N passes (N = # relations joined):

For each subset of relations, retain only:

- Cheapest subplan overall (possibly unordered), plus
- Cheapest subplan for each *interesting order* of the tuples

For each subplan retained, remember cost and result size estimates