Relational Query Optimization

CS 375, Fall 2014

It is safer to accept any chance that offers itself, and extemporize a procedure to fit it, than to get a good plan matured, and wait for a chance of using it.

Thomas Hardy (1874)
in *Far from the Madding Crowd*

Overview of Query Evaluation

- ***Plan:*** ***Tree of R.A. ops, with choice of alg for each op.***
  - Each operator typically implemented using a `pull' interface: when an operator is `pulled' for the next output tuples, it `pulls' on its inputs and computes them.
- **Two main issues in query optimization:**
  - For a given query, what plans are considered?
    - Algorithm to search plan space for cheapest (estimated) plan.
  - How is the cost of a plan estimated?
- **Ideally: Want to find best plan.**
- **Practically: Avoid worst plans!**

Some Common Techniques

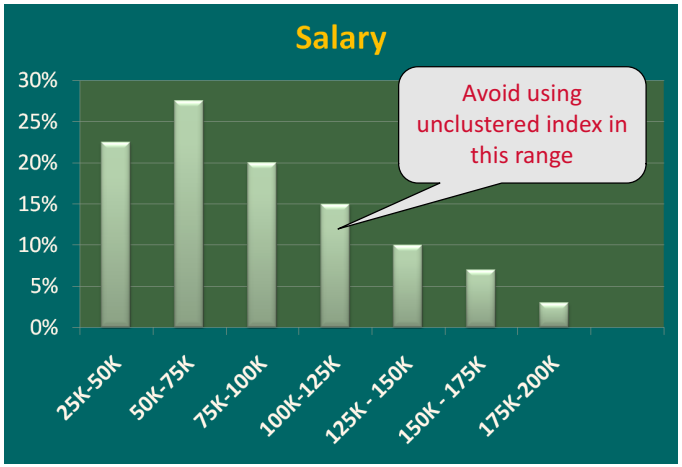
- **Algorithms for evaluating relational operators use some simple ideas extensively:**
  - Indexing: Can use WHERE conditions to retrieve small set of tuples (selections, joins)
  - Iteration: Sometimes, faster to scan all tuples even if there is an index. (And sometimes, we can scan the data entries in an index instead of the table itself.)
  - Partitioning: By using sorting or hashing, we can partition the input tuples and replace an expensive operation by similar operations on smaller inputs.

Watch for these techniques as we discuss query evaluation!

Statistics and Catalogs

- **Need information about the relations and indexes involved.** ***Catalogs*** **typically contain at least:**
  - # tuples (NTuples) and # pages (NPages) for each relation.
  - # distinct key values (NKeys) and NPages for each index.
  - Index height, low/high key values (Low/High) for each tree index.
- **More detailed information (e.g., histograms of the values in some field) are sometimes stored.**
- **Catalogs updated periodically.**
  - Updating whenever data changes is too expensive; lots of approximation anyway, so slight inconsistency ok.

Histogram



Access Paths

- **An access path is a method of retrieving tuples:**
  - File scan, or index that matches a selection (in the query)
- **A tree index *matches* (a conjunction of) terms that involve only attributes in a *prefix* of the search key.**
  - Example: Tree index on *<a, b, c>* matches the selection *a=5 AND b=3*, and *a=5 AND b>6*, but not *b=3*.
- **A hash index *matches* (a conjunction of) terms that has a term *attribute = value* for every attribute in the search key of the index.**
  - Example: Hash index on *<a, b, c>* matches *a=5 AND b=3 AND c=5*; but it does not match *b=3, or a=5 AND b=3, or a>5 AND b=3 AND c=5*.

A Note on Complex Selections

(day<8/9/94 AND rname='Paul') OR bid=5 OR sid=3

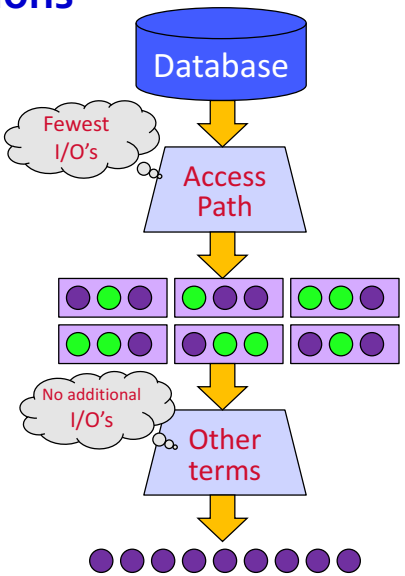
- **Selection conditions are first converted to *conjunctive normal form* (CNF):**

  (day<8/9/94 OR bid=5 OR sid=3) AND (rname= 'Paul' OR bid=5 OR sid=3)

- **We only discuss case with no ORs; see text if you are curious about the general case.**

One Approach to Selections

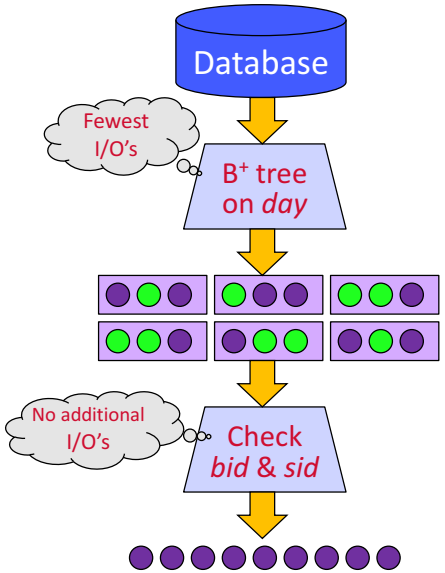
- **Find the *most selective access path*, retrieve tuples using it, and apply any remaining terms that don't match the index:**
  - *Most selective access path:* An index or file scan that we estimate will require the fewest page I/Os.
  - Terms that match this index reduce the number of tuples *retrieved*; other terms are used to discard some retrieved tuples, but do not affect number of tuples/pages fetched.



One Approach to Selections - Example

Consider *day<8/9/94 AND bid=5 AND sid=3*.

- A B+ tree index on *day* can be used; then, *bid=5* and *sid=3* must be checked for each retrieved tuple.
- Similarly, a hash index on *<bid, sid>* could be used; *day<8/9/94* must then be checked.



Using an Index for Selections

- **Cost depends on #qualifying tuples, and clustering.**

Cost = Cost(finding qualifying data entries) + Cost(retrieving record)

Typically small

Could be large w/o clustering

Example: Assuming uniform distribution of names, about 5% of tuples qualify (say 100 pages, 10,000 tuples).

- With a clustered index, cost is little more than 100 I/Os;
- if unclustered, up to 10,000 I/Os!

```
SELECT  *
FROM    Reserves R
WHERE   R.rname LIKE 'C%'
```

Duplicates Elimination Using Sorting

- **The expensive part is removing duplicates.**

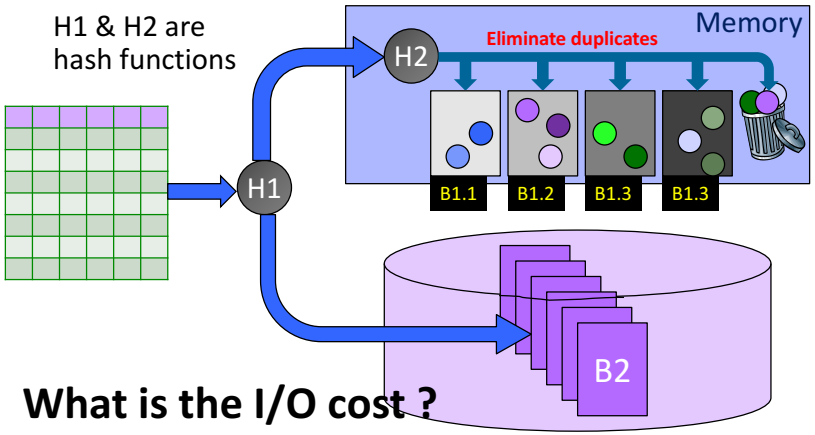
  SQL systems don't remove duplicates unless the keyword DISTINCT is specified in a query.

```
SELECT  DISTINCT  R.sid, R.bid
FROM    Reserves R
```

- **Sorting Approach: Sort on <sid, bid> and remove duplicates. (Can optimize this by dropping unwanted attributes while sorting.)**
- **Sorting Data Entries: If there is an index with both R.sid and R.bid in the search key, may be cheaper to sort data entries!**

Duplicates Elimination Using Hashing

```
SELECT  DISTINCT  R.sid, R.bid
FROM    Reserves R
```

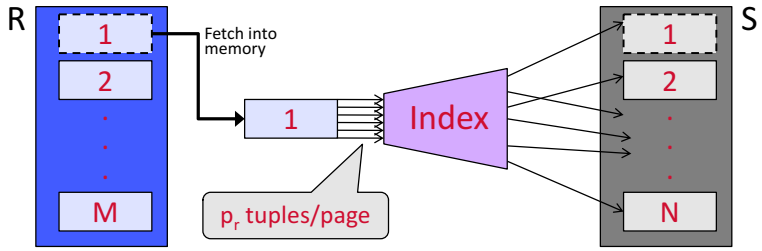


What is the I/O cost ?

Join: Index Nested Loops (1)

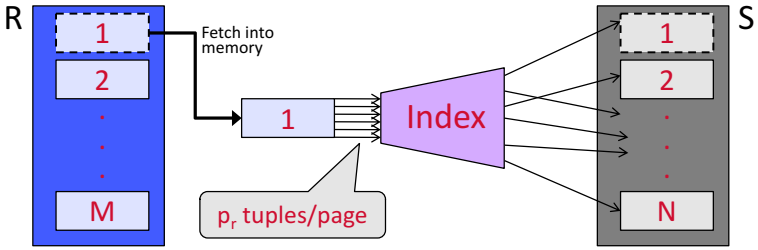
foreach tuple *r* in *R* do
    foreach tuple *s* in *S* where r_i == s_j do
        add <*r*, *s*> to result

- **If there is an index on the join column of one relation (say *S*), can make it the inner and exploit the index.**
  - Cost: M + ((M*p_R) × cost of finding matching S tuples)



Join: Index Nested Loops (2)

- **For each *R* tuple, cost of probing *S* index (i.e., finding data entry) is about 1.2 for hash index, 2-4 for B+ tree.**
- **Cost of then finding S tuples (assuming Alt. (2) or (3) for data entries) depends on clustering.**
  - Clustered index: 1 I/O (typical),
  - Unclustered: upto 1 I/O per matching S tuple.



Examples of Index Nested Loops (1)

- **Hash-index (Alt. 2) on *sid* of Sailors (as inner):**
  - Scan Reserves: 1,000 page I/Os, 100*1,000 tuples.
  - For each Reserves tuple: 1.2 I/Os to get data entry in index, plus 1 I/O to get (the exactly one) matching Sailors tuple.
  - Total: 1,000 + (100×1,000) × (1.2 +1) ≈ 220,000 I/Os.

Load *Reserves* pages
of *Reserves tuples*
Cost of finding all matching *Sailors tuples*
Find the matching *Sailors* tuple

Examples of Index Nested Loops

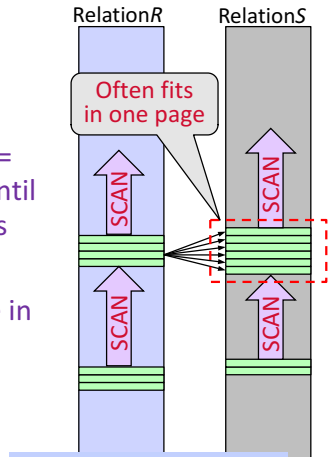
- **Hash-index (Alt. 2) on *sid* of Reserves (as inner):**
  - Scan Sailors: 500 page I/Os, 80×500 tuples.
  - Assuming uniform distribution, 2.5 reservations per sailor, there are (80×500)×2.5 = 100,000 matching *Reserves* tuples.
  - For each Sailors tuple: 1.2 I/Os to find index page with data entries, plus cost of retrieving matching *Reserves* tuples.
  - COST = 500 + (80×500)×1.2 + Cost(Fetching 100,000 matching *Reserves* tuples)

Load *Sailors* pages
Find matching data entries
Cost of finding each match is 1/2.5 or 1 I/Os depending on whether the index is clustered

Join: Sort-Merge ($R \bowtie_{i=j} S$)

Sort *R* and *S* on the join column, then scan them to do a ``merge'' (on join col.), and output result tuples.

- Advance scan of *R* until current *R*-tuple >= current *S* tuple, then advance scan of S until current *S*-tuple >= current *R* tuple; do this until current *R* tuple = current *S* tuple.
- At this point, all *R* tuples with same value in R_i (*current R group*) and all *S* tuples with same value in S_j (*current S group*) match; output <*r*, *s*> for all pairs of such tuples.
- Then resume scanning *R* and *S*.



COST: *R* is scanned once; each *S* group is scanned once per matching *R* tuple.

Example of Sort-Merge Join

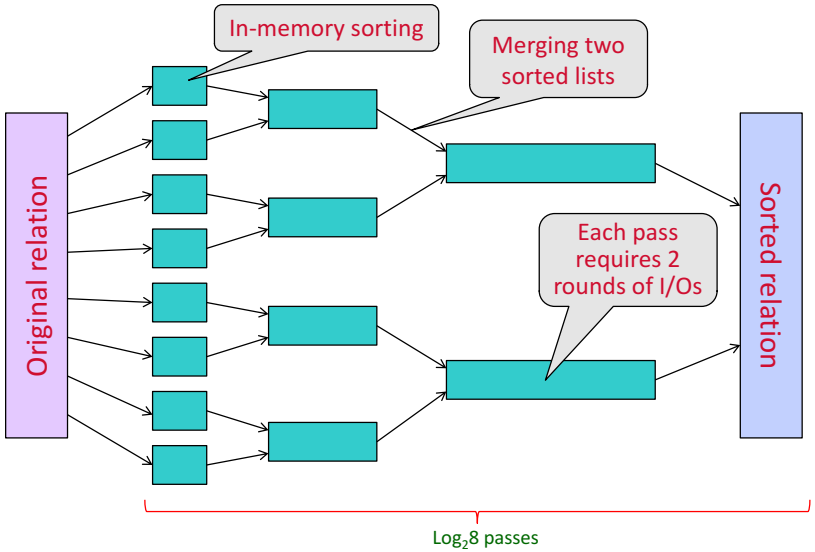
sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

Cost: M log M + N log N + (M+N)

- The cost of scanning, M+N, could be M*N (very unlikely!)

Merger Sort



Highlights of System R Optimizer

- **Impact:**
  - Most widely used currently; works well for < 10 joins.
- **Cost estimation: Approximate art at best.**
  - Statistics, maintained in system catalogs, used to estimate cost of operations and result sizes.
  - Considers combination of CPU and I/O costs.
- **Plan Space: Too large, must be pruned.**
  - Only the space of *left-deep plans* is considered.
    - Left-deep plans allow output of each operator to be *pipelined* into the next operator without storing it in a temporary relation.
  - Cartesian products avoided.

Query Block

- **An SQL query is parsed into a collection of query blocks and then passed on to the query optimizer**
- **A query block is an SQL query with no nesting**

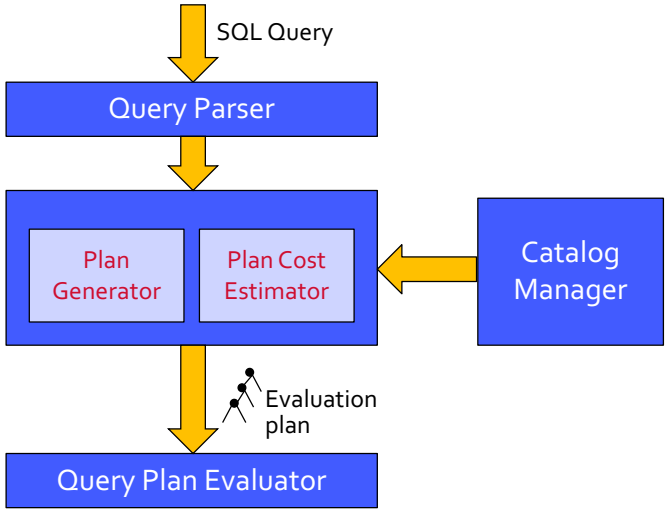
```
SELECT   S.sid, MIN(R.day)
FROM     Sailors S, Reserves R, Boats B
WHERE    S.sid=R.sid AND R.bid-B.bid AND
         B.colr= 'red' AND
         S.rating = (SELECT MAX (S2.rating)
                     FROM Sailors S2 )
GROUP BY S.sid
HAVING   COUNT(*)>1
```

```
SELECT MAX (S2.rating)
FROM   Sailors S2
```

```
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.colr= 'red' AND
         S.rating = Reference to nested block
GROUP BY S.sid
HAVING   COUNT(*)>1
```

The query optimizer considers each query block and chooses a query evaluation plan for that block.

Query Processor



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 also estimate *size of result* for each operation in tree!
  - Use information about the input relations.
  - For selections and joins, assume independence of predicates.

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.**

Result cardinality = Max # tuples × product of all RF's

⇒ Implicit assumption that *terms* are independent!

Size Estimation and Reduction Factors

***Result cardinality* = Max # tuples × product of all RF's**

- **Term *col=value* has RF 1/NKeys(*I*), given index I on *col***

  EXAMPLE: If the number of distinct key values is 100, size(result) = 1%*size(operand relation)

- **Term *col1=col2* has RF 1/MAX(NKeys(*I1*), NKeys(*I2*))**

  OBSERVATION:

  - The smaller the numbers of distinct key values, the bigger the size of the result due to more matches between the two operand relations.
  - If Nkeys(*I1*)=Nkeys(*I2*)=1, RF=1 and size(*R1* ⋈ *R2*) = size(*R1*×*R2*)

- **Term *col>value* has RF (High(*I*)-*value*)/(High(*I*)-Low(*I*))**
  - RF is the percentage of the tuples that satisfy the term

Schema for Examples

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

- **Similar to old schema; *rname* added for variations.**
- ***Reserves*:**
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages.
- ***Sailors*:**
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages.

Plan Generation (1)

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

SELECT clause | WHERE clause | FROM clause

$$\Pi_{sname}(\sigma_{sid=sid \wedge bid=100 \wedge rating>5} (R \times S))$$

$$= \Pi_{sname}(\sigma_{bid=100 \wedge rating>5} (\sigma_{sid=sid} (R \times S)))$$

$$= \Pi_{sname}(\sigma_{bid=100 \wedge rating>5} (R \bowtie S))$$

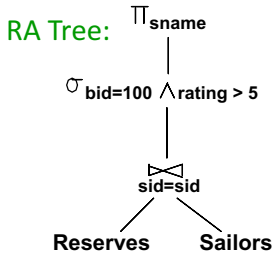
Plan Generation (2)

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

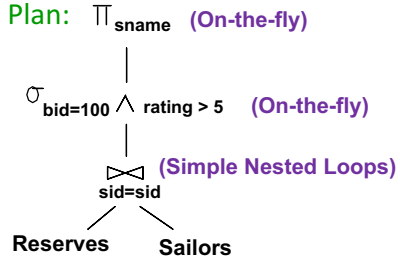
$$\Pi_{sname}(\sigma_{bid=100 \wedge rating>5} (R \bowtie S))$$

On-the-fly ⇒ Save I/O

RA Tree:



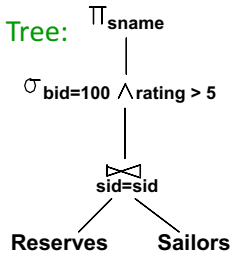
Plan:



Motivating Example

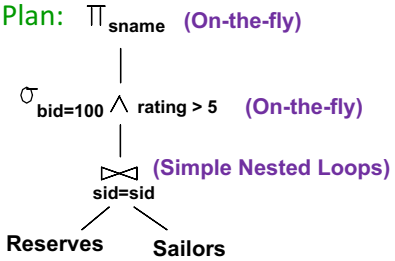
```
SELECT  S.sname
FROM  Reserves R, Sailors S
WHERE  R.sid=S.sid AND
   R.bid=100 AND S.rating>5
```

RA Tree:

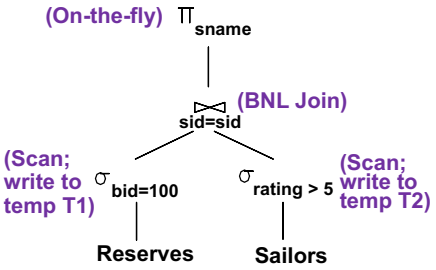


- **Cost: 500 + 500×1000 I/Os**
- **By no means the worst plan!**
- **Misses several opportunities: selections could have been `pushed' earlier, no use is made of any available indexes, etc.**
- ***Goal of optimization:* To find more efficient plans that compute the same answer.**

Plan:



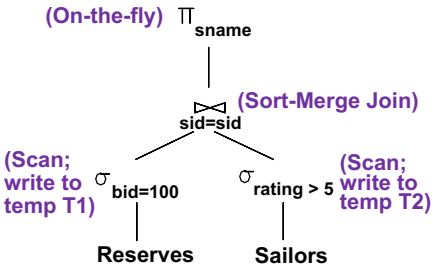
Alternative Plans 1 (No Indexes)



- ***Main difference:* *push selects.***
- **With 5 buffer pages, cost of plan:**
  - TotalCost(using sort-merge join) = 5,060 page I/Os.
- **If we used Block Nested Loop join, join cost = 10 + 4×250, total cost = 2770.**
- **If we `push' projections, T1 has only *sid*, T2 only *sid* and *sname*:**
  - T1 fits in 3 pages, cost of BNL drops to about 250 pages, total < 2000.

Scan T2 for every 3 pages of T1
⇒need to scan T2 ⌈10/3⌉ or four times

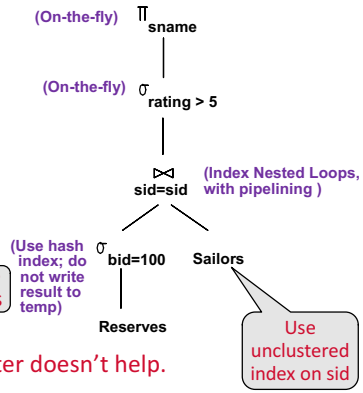
Alternative Plans 1 (No Indexes)



- ***Main difference:* *push selects.***
- **With 5 buffer pages, cost of plan:**
  - Scan Reserves (1000) + write temp T1 (10 pages, if we have 100 boats, uniform distribution ⇒ RF=1%).
  - Scan Sailors (500) + write temp T2 (250 pages, if we have 10 ratings ⇒ RF=5/10).
  - Sort T1 (2×2×10), sort T2 (2×6×250), merge (10+250)
  - Total: 5,060 page I/Os.

Need 6 passes
Each pass requires 2 rounds of I/Os

Alternative Plans 2 with Indexes



- **With clustered index on *bid* of *Reserves*, we get 100,000/100 = 1000 tuples on 1000/100 = 10 pages.**
- **INL with *pipelining* (outer is not materialized, i.e., not written to disk).**
  - Projecting out unnecessary fields from outer doesn't help.

There are 100 boats

- Join column *sid* is a key for Sailors.
  - At most one matching tuple, unclustered index on *sid* OK.
- Decision not to push *rating > 5* before the join is based on availability of *sid* index on Sailors.
- Cost: Selection of Reserves tuples (10 I/Os); for each, must get matching Sailors tuple (1000×1.2); total 1210 I/Os.

Summary

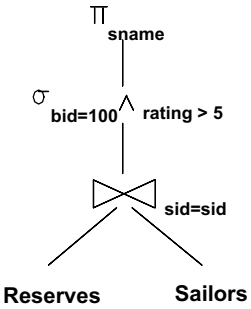
- **There are several alternative evaluation algorithms for each relational operator.**
- **A query is evaluated by converting it to a tree of operators and evaluating the operators in the tree.**
- **Must understand query optimization in order to fully understand the performance impact of a given database design (relations, indexes) on a workload (set of queries).**
- **Two parts to optimizing a query:**
  - Consider a set of alternative plans.
    - Must prune search space; typically, left-deep plans only.
  - Must estimate cost of each plan that is considered.
    - Must estimate size of result and cost for each plan node.
    - *Key issues*: Statistics, indexes, operator implementations.

Query Optimization Overview

- **Query are converted to an intermediate format such as relational algebra**
- **Rel. Algebra converted to tree, joins as branches**
- **Each operator has implementation choices**
- **Operators can also be applied in different order!**

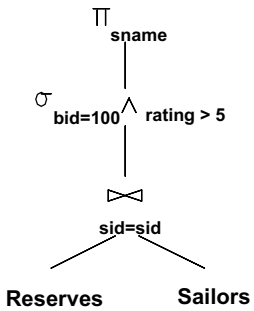
```
SELECT  S.sname
FROM  Reserves R, Sailors S
WHERE  R.sid=S.sid AND
   R.bid=100 AND S.rating>5
```

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



Iterator Interface (pull from the top)

- **Recall:**



•Relational operators at nodes support uniform *iterator* interface:

Open( ), get_next( ), close( )

•Unary Ops – On Open() call Open() on child.

•Binary Ops – call Open() on left child then on right.

•By convention, outer is on left.

Alternative is pipelining (i.e. a "push"-based approach).

Can combine push & pull using special operators.

Query Optimization Overview (cont)

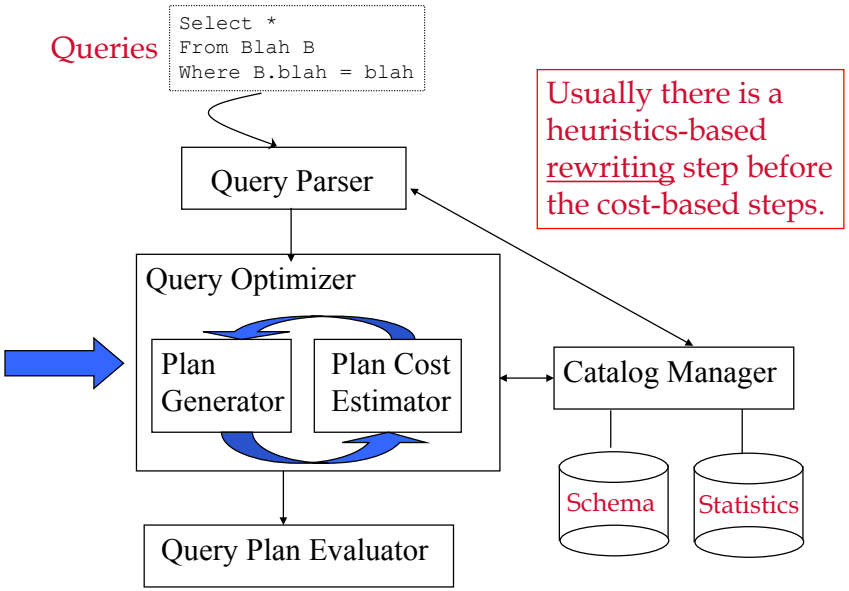
- ***Logical Plan:*** *Tree of R.A. ops*
- ***Physical Plan:*** *Tree of R.A. ops, with choice of algorithm for each operator.*

- **Two main issues:**
  - For a given query, what plans are considered?
    - Algorithm to search plan space for cheapest (estimated) plan.
  - How is the cost of a plan estimated?

- **Ideally: Want to find best plan.**

- **Reality: Avoid worst plans!**

Cost-based Query Sub-System



Schema for Examples

Sailors (*sid*: integer, *sname*: string, *rating*: integer, *age*: real)

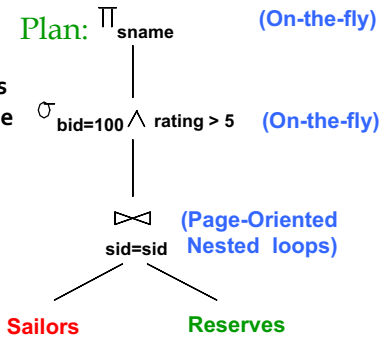
Reserves (*sid*: integer, *bid*: integer, *day*: dates, *rname*: string)

- **As seen in previous lectures…**
- **Reserves:**
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages.
  - Let's say there are 100 boats.
- **Sailors:**
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages.
  - Let's say there are 10 different ratings.
- **Assume we have 5 pages in our buffer pool.**

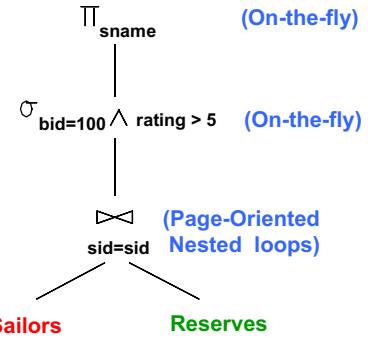
Motivating Example

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

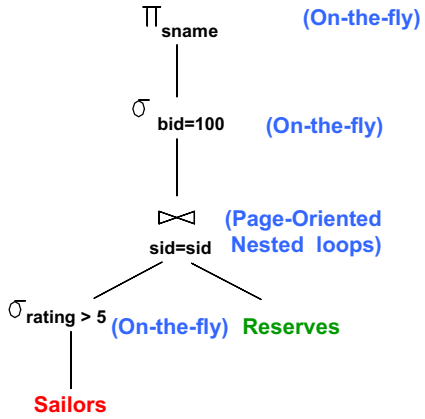
- **Cost: 500+500*1000 I/Os**
- **By no means the worst plan!**
- **Misses several opportunities: selections could have been `pushed' earlier, no use is made of any available indexes, etc.**
- ***Goal of optimization:* To find more efficient plans that compute the same answer.**



Alternative Plans – Push Selects (No Indexes)

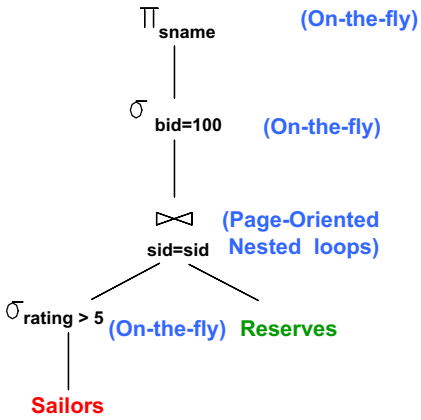


500,500 IOs

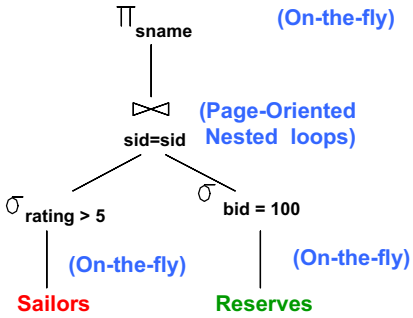


250,500 IOs

Alternative Plans – Push Selects (No Indexes)

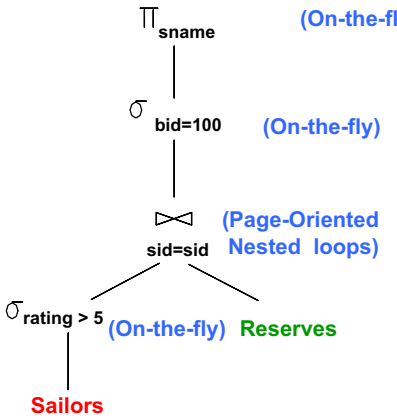


250,500 IOs

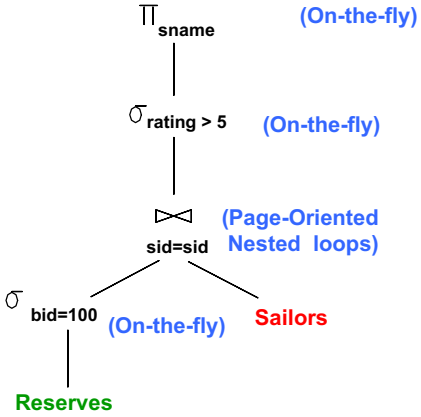


250,500 IOs

Alternative Plans – Push Selects (No Indexes)

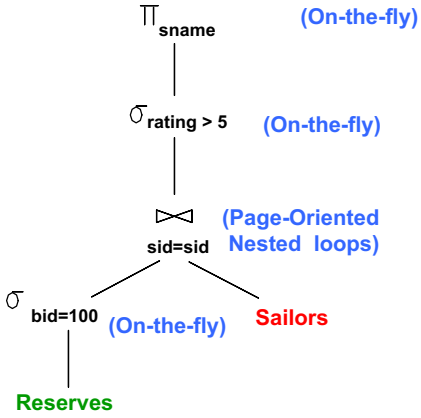


250,500 IOs

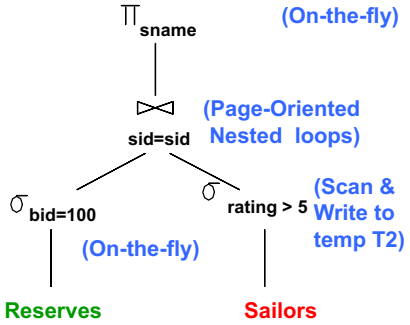


6000 IOs

Alternative Plans – Push Selects (No Indexes)

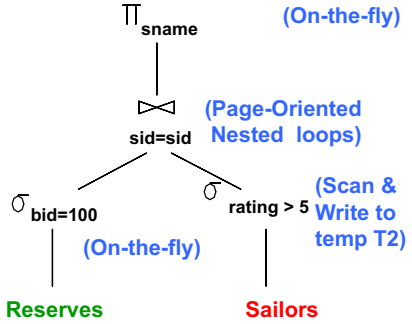


6000 IOs

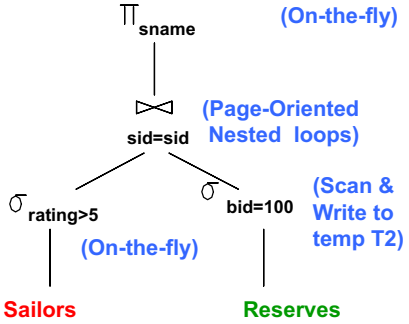


4250 IOs
1000 + 500+ 250 + (10 * 250)

Alternative Plans – Push Selects (No Indexes)

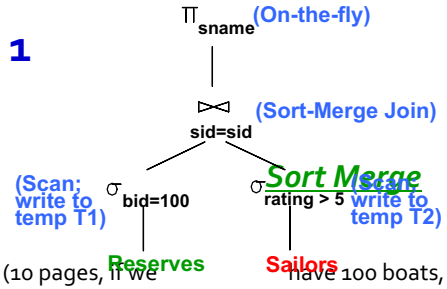


4250 IOs



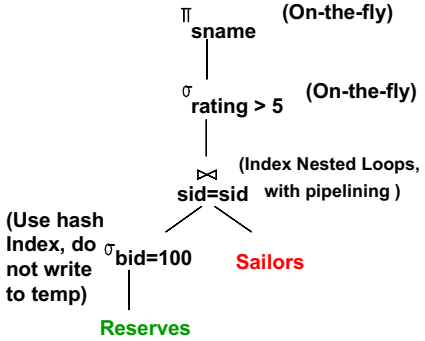
4010 IOs
500 + 1000 +10 +(250 *10)

Alternative Plans 1 (No Indexes)



- *Main difference:* ***Join***
- **With 5 buffers, cost of plan:**
  - Scan Reserves (1000) + write temp T1 (10 pages, if we have 100 boats, uniform distribution).
  - Scan Sailors (500) + write temp T2 (250 pages, if have 10 ratings).
  - Sort T1 (2*2*10), sort T2 (2*4*250), merge (10+250)
  - Total: 4060 page I/Os. (note: T2 sort takes 4 passes with B=5)
- **If use BNL join, join = 10+4*250, total cost = 2770.**
- **Can also `push' projections, but must be careful!**
  - T1 has only *sid*, T2 only *sid*, *sname*:
  - T1 fits in 3 pgs, cost of BNL under 250 pgs, total < 2000.

Alt Plan 2: Indexes



- With clustered hash index on *bid* of Reserves, we get 100,000/100 = 1000 tuples on 1000/100 = 10 pages.
- INL with outer not materialized.
  - Projecting out unnecessary fields from outer doesn't help.

- Join column *sid* is a key for Sailors.
  At most one matching tuple, unclustered index on *sid* OK.
- Decision not to push *rating>5* before the join is based on availability of *sid* index on Sailors.
- Cost: Selection of Reserves tuples (10 I/Os); then, for each, must get matching Sailors tuple (1000*1.2); total 1210 I/Os.

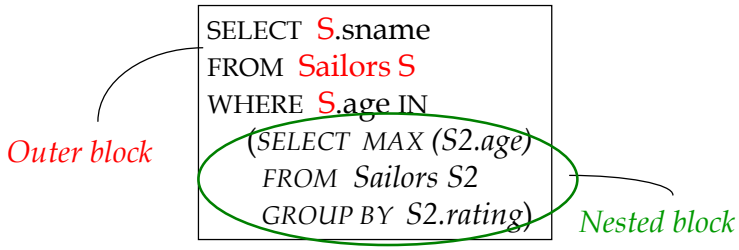
What is needed for optimization?

- **Iterator Interface**
- **Cost Estimation**
- **Statistics and Catalogs**
- **Size Estimation and Reduction Factors**

Summary so far

- **Query optimization is an important task in a relational DBMS.**
- **Must understand optimization in order to understand the performance impact of a given database design (relations, indexes) on a workload (set of queries).**
- **Two parts to optimizing a query:**
  1. Consider a set of alternative plans.
     - Must prune search space; typically, left-deep plans only.
  2. Must estimate cost of each plan that is considered.
     - Must estimate size of result and cost for each plan node.
     - *Key issues*: Statistics, indexes, operator implementations.

Query Blocks: Units of Optimization



- An SQL query is parsed into a collection of *query blocks*, and these are optimized one block at a time.

- Inner blocks are usually treated as subroutines
- Computed:
  - once per query (for uncorrelated sub-queries)
  - or once per outer tuple (for correlated sub-queries)

Translating SQL 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
```

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.

Translating SQL 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

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

Relational Algebra Equivalences

- **Allow us to choose different operator orders and to `push' selections and projections ahead of joins.**
- ***Selections:***

$$\sigma_{c1 \wedge \ldots \wedge cn}(R) \equiv \sigma_{c1}(\ldots \sigma_{cn}(R)) \quad \text{(Cascade)}$$

$$\sigma_{c1}(\sigma_{c2}(R)) \equiv \sigma_{c2}(\sigma_{c1}(R)) \quad \text{(Commute)}$$

- *Projections:*

$$\pi_{a1}(R) \equiv \pi_{a1}(\ldots(\pi_{an}(R))) \quad \text{(Cascade)}$$

(if a_n in a_1)

- *Joins:*

$$R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T \quad \text{(Associative)}$$

$$(R \bowtie S) \equiv (S \bowtie R) \quad \text{(Commute)}$$

These two mean we can do joins in any order.

More Equivalences

- **A projection commutes with a selection that only uses attributes retained by the projection.**
- **Selection between attributes of the two arguments of a cross-product converts cross-product to a join.**
- **Selection Push: selection on R attrs commutes with R ⋈ S:** $\sigma(R \bowtie S) \equiv \sigma(R) \bowtie S$
- **Projection Push: A projection applied to R ⋈ S can be pushed before the join by retaining only attributes of R (and S) that are needed for the join or are kept by the projection.**

The "System R" Query Optimizer

- **Impact:**
  - Inspired most optimizers in use today
  - Works well for small-med complexity queries (< 10 joins)
- **Cost estimation:**
  - Very inexact, but works ok in practice.
  - Statistics, maintained in system catalogs, used to estimate cost of operations and result sizes.
  - Considers a simple 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.
  - Cartesian products avoided.

Cost Estimation

- **To estimate cost of a plan:**
  - Must estimate *cost* of each operation in plan tree and sum them up.
    - Depends on input cardinalities.
  - So, 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 ops + *factor* * #CPU instructions**

Q: How does "cost" relate to estimated "run time"?

Statistics and Catalogs

- **Need information about the relations and indexes involved.** *Catalogs* **typically contain at least:**
  - # tuples (**NTuples**) and # pages (**NPages**) per rel'n.
  - # 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.
- **Stats in catalogs updated periodically.**
  - Updating whenever data changes is too expensive; lots of approximation anyway, so slight inconsistency 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
```

- *Reduction factor (RF)* associated with each *term* reflects the impact of the *term* in reducing result size.
- *RF is usually called "selectivity".*
- How to predict size of output?
  - Need to know/estimate input size
  - Need to know/estimate RFs
  - Need to know/assume how terms are related

Result Size Estimation for Selections

- *Result cardinality (for conjunctive terms)* = **# input tuples * product of all RF's.**

  Assumptions:
  1. Values are uniformly distributed and *terms* are independent!
  2. In System R, stats only tracked for indexed columns (modern systems have removed this restriction)
- Term *col=value*
  RF = *1/NKeys(I)*
- Term *col1=col2* **(This is handy for joins too…)**
  RF = *1/MAX(NKeys(I1), NKeys(I2))*
- Term *col>value*
  RF = *(High(I)-value)/(High(I)-Low(I))*

- *Note, In System R, if missing indexes, assume **1/10**!!!*

Reduction Factors & Histograms

- **For better RF estimation, many systems use histograms:**

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

equiwidth

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

equidepth

Result Size estimation for joins

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

  (R,S = schema for reln R,S; i.e, attributes of R,S)
  - Hint: what if attr R∩S = ∅?
  - R∩S is a key for R (and a Foreign Key in S)?

- **General case: R∩S = {A} (and A is key for neither)**
  - estimate each tuple r of R generates NTuples(S)/NKeys(A,S) result tuples, so…

    NTuples(R) * NTuples(S)/NKeys(A,**S**)
  - but can also consider it starting with S, yielding: NTuples(R) * NTuples(S)/NKeys(A,**R**)
  - If these two estimates differ, take the lower one!
    - Q: Why?

Enumeration of Alternative Plans

- **There are two main cases:**
  - Single-relation plans (unary ops) and Multiple-relation plans
- **For unary operators:**
  - For a scan, each available access path (file scan / index) is considered, and the one with the least estimated cost is chosen.
  - consecutive **Scan, Select, Project** and **Aggregate** operations can be essentially carried out together

(e.g., if an index is used for a selection, projection is done for each retrieved tuple, and the resulting tuples are *pipelined* into the aggregate computation).

I/O Cost Estimates for Single-Relation Plans

- **Index I on primary key matches selection:**
  - *Cost is Height(I)+1 for a B+ tree, about 1.2 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 requried*

Schema for Examples

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

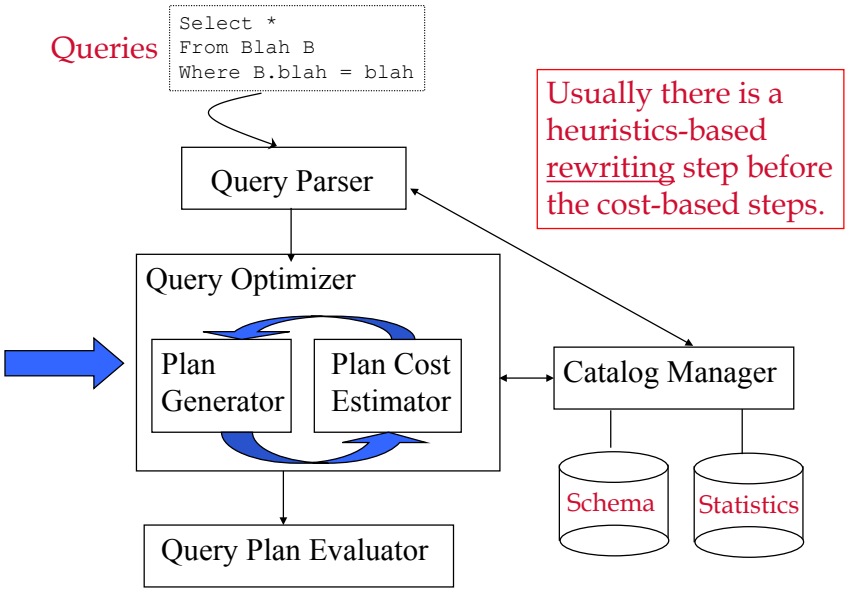
- **Reserves:**
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages. 100 distinct bids.
- **Sailors:**
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages. 10 Ratings, 40,000 sids.

Example

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

- **If we have an index on *rating*:**
  - Cardinality: (1/NKeys(I)) * NTuples(S) = (1/10) * 40000 tuples retrieved.
  - Clustered index: (1/NKeys(I)) * (NPages(I)+NPages(S)) = (1/10) * (50+500) = 55 pages are retrieved.
  - Unclustered index: (1/NKeys(I)) * (NPages(I)+NTuples(S)) = (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. No reason to use this index! (see below)
- **Doing a file scan:**
  - We retrieve all file pages (500).

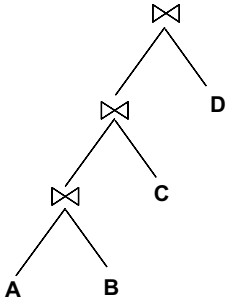
Cost-based Query Sub-System



System R - Plans to Consider

For each block, plans considered are:

- All available access methods, for each relation in FROM clause.
- All *left-deep join trees*
  - i.e., all ways to join the relations one-at-a-time, considering all relation permutations and join methods.
  (note: system R originally only had NL and Sort Merge)

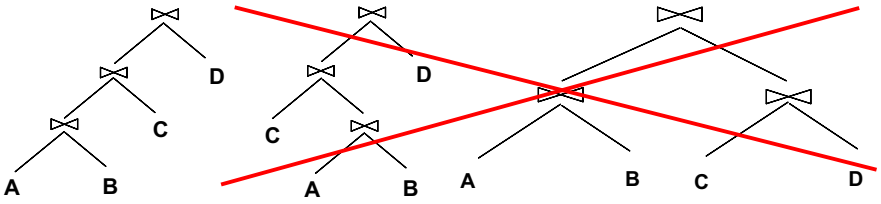


Highlights of System R Optimizer

- **Impact:**
  - Most widely used currently; works well for < 10 joins.
- **Cost estimation:**
  - Very inexact, but works ok in practice.
  - Statistics, maintained in system catalogs, used to estimate cost of operations and result sizes.
  - Considers combination of CPU and I/O costs.
    - For simplicity we ignore CPU costs in this discussion
  - More sophisticated techniques known now.
- **Plan Space: Too large, must be pruned.**
  - Only the space of *left-deep plans* is considered.
  - Cartesian products avoided.

Queries Over Multiple Relations

- **Fundamental decision in System R: *only left-deep join trees* are considered.**
  - As the number of joins increases, the number of alternative plans grows rapidly; *we need to restrict 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).



Enumeration: Dynamic Programming

- **Plans differ by: order of the N relations, access method for each relation, and the join method for each join.**
  - maximum possible orderings = N! (but delay X-products)
- **Enumerated using N passes**
- **For each subset of relations, retain only:**
  - Cheapest plan overall (possibly unordered), plus
  - Cheapest plan for each *interesting order* of the tuples.

Enumeration: Dynamic Programming

- **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.)*
  consider all possible join methods & inner access paths
- **Pass N: Find best ways to join result of a (N-1)-rel'n plan as outer to the N'th relation.** *(All N-relation plans.)*
  consider all possible join methods & inner access paths

Interesting Orders

- **An intermediate result has an "interesting order" if it is returned in order of any of:**
  - ORDER BY attributes
  - GROUP BY attributes
  - Join attributes of other joins

System R Plan Enumeration (Contd.)

- An 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.
- ORDER BY, GROUP BY, aggregates etc. handled as a final step, using either an `interestingly ordered' plan or an additional sorting operator.
- In spite of pruning plan space, this approach is still exponential in the # of tables.
- COST = **#IOs + (inst_per_IO * *CPU Inst)***

Example (modified from book ch 15)

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

Indexes
Reserves:
Clustered B+ tree on *bid*
Sailors:
Unclust B+ tree on *rating*

Pass1:

Reserves: Clustered B+ tree on *bid* matches *bid=100*, and is cheaper than file scan

Sailors: B+ tree matches *rating>5*, not very selective, and index is unclustered, so file scan w/ select is likely cheaper. Also, Sailors.rating is not an interesting order.

Pass 2:We consider each Pass 1 plan as the outer:

Reserves as outer (B+Tree selection on bid):

Use Sort Merge to join with Sailors as inner

Sailors as outer (File Scan w/select on rating):

Use BNL on result of selection on Reserves.bid

Example (modified from book ch 15)

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

Sailors:
B+ on *sid*
Reserves:
Clustered B+ tree on *bid*
B+ on *sid*
Boats
Clustered Hash on *color*

- **Pass1: Best plan(s) for accessing each relation**
  - Sailors: File Scan; B+ on sid
  - Reserves: File Scan; B+ on bid, B+ on sid
  - Boats: Hash on color

    (note: given selection on color, clustered Hash is likely to be cheaper than file scan, so only it is retained)

Pass 2

- **For each of the plans in pass 1, generate plans joining another relation as the inner (avoiding cross products).**
- **Consider all join methods and every access path for the inner.**
  - File Scan Reserves (outer) with Boats (inner)
  - File Scan Reserves (outer) with Sailors (inner)
  - B+ on Reserves.bid (outer) with Boats (inner)
  - B+ on Reserves.bid (outer) with Sailors (inner)
  - B+ on Reserves.sid (outer) with Boats (inner)
  - B+ on Reserves.sid (outer) with Sailors (inner)
  - File Scan Sailors (outer) with Reserves (inner)
  - B+Tree Sailors.sid (outer) with Reserves (inner)
  - Hash on Boats.color (outer) with Reserves (inner)
- **Retain cheapest plan for each pair of relations plus cheapest plan for each interesting order.**

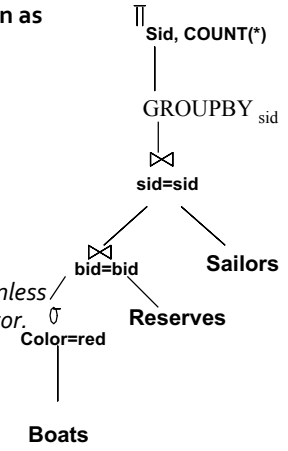
Pass 3

- **For each of the plans retained from Pass 2, taken as the outer, generate plans for the remaining join**
  - e.g.

    Outer= Hash on Boats.color JOIN Reserves

    Inner = Sailors

    Join Method = Index NL using Sailors.sid B+Tree
- **Then, add the cost for doing the group by and 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 overall**



Nested Queries

- **Nested block is optimized independently, with the outer tuple considered as providing a selection condition.**
- **Outer block is optimized with the cost of `calling' nested block computation taken into account.**
- **Implicit ordering of these blocks means that some good strategies are not considered.** ***The non-nested version of the query is typically optimized better.***

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

Nested block to optimize:

```
SELECT  *
FROM  Reserves R
WHERE  R.bid=103
  AND  R.sid= outer value
```

Equivalent non-nested query:

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

Points to Remember

- Must understand optimization in order to understand the performance impact of a given database design (relations, indexes) on a workload (set of queries).
- Two parts to optimizing a query:
  - Consider a set of alternative plans.
    - Must prune search space; typically, left-deep plans only.
  - Must estimate cost of each plan that is considered.
    - Must estimate size of result and cost for each plan node.
    - *Key issues*: Statistics, indexes, operator implementations.

Points to Remember

- **Single-relation queries:**
  - All access paths considered, cheapest is chosen.
  - *Issues*: Selections that *match* index, whether index key has all needed fields and/or provides tuples in a desired order.

More Points to Remember

- **Multiple-relation queries:**
  - All single-relation plans are first enumerated.
    - Selections/projections considered as early as possible.
  - Next, for each 1-relation plan, all ways of joining another relation (as inner) are considered.
  - Next, for each 2-relation plan that is `retained', all ways of joining another relation (as inner) are considered, etc.
  - At each level, for each subset of relations, only best plan for each interesting order of tuples is `retained'.

Summary

- **Performance can be dramatically improved by changing access methods, order of operators.**
- **Iterator interface**
- **Cost estimation**
  - Size estimation and reduction factors
- **Statistics and Catalogs**
- **Relational Algebra Equivalences**
- **Choosing alternate plans**
- **Multiple relation queries**
- **We focused on "System R"-style optimizers**
  - New areas: Rule-based optimizers, random statistical approaches (*eg simulated annealing), adaptive/dynamic optimization.*