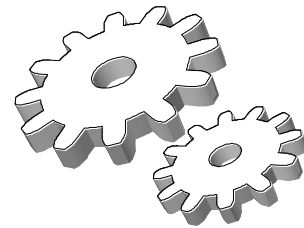


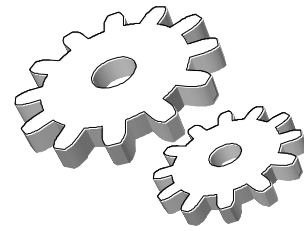
Overview of Query Evaluation

Chapter 12



Overview of Query Evaluation

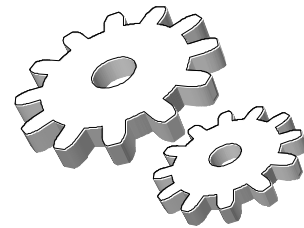
- ❖ Plan: Tree of R.A. ops, with choice of alg for each op.
 - Each operator typically implemented using a 'pull' interface: when an operator is 'pulled' for the next output tuples, it 'pulls' on its inputs and computes them.
- ❖ Two main issues in query optimization:
 - For a given query, **what plans 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!
- ❖ We will study the System R approach.



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.

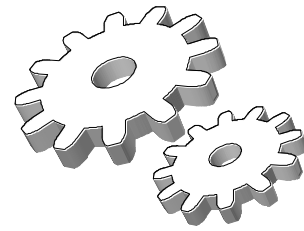
** Watch for these techniques as we discuss query evaluation!*



Statistics and Catalogs

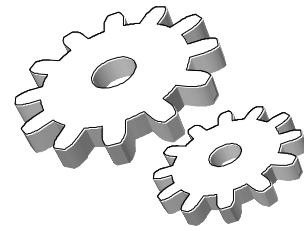
- ❖ Need information about the relations and indexes involved. *Catalogs* typically contain at least:
 - # tuples (NTuples) and # pages (NPages) for each relation.
 - # distinct key values (NKeys) and NPages for each index.
 - Index height, low/high key values (Low/High) for each tree index.
- ❖ Catalogs 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.

Access Paths



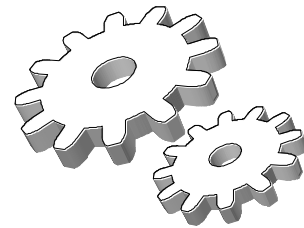
- ❖ An access path is a method of retrieving tuples:
 - **File scan**, or **index** that **matches** a selection (in the query)
- ❖ A tree index matches (a conjunction of) terms that involve only attributes in a *prefix* of the search key.
 - E.g., Tree index on $\langle a, b, c \rangle$ **matches** 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.
 - E.g., Hash index on $\langle a, b, c \rangle$ **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

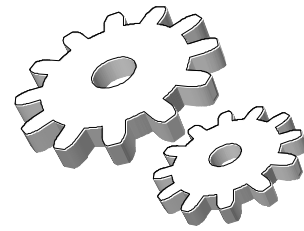
- ❖ Selection conditions are first converted to conjunctive normal form (CNF):
*(day<8/9/94 OR bid=5 OR sid=3) AND
(rname='Paul' OR bid=5 OR sid=3)*
- ❖ We only discuss case with no ORs; see text if you are curious about the general case.



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.
 - 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 $\langle bid, sid \rangle$ could be used; *day<8/9/94* must then be checked.

Using an Index for Selections

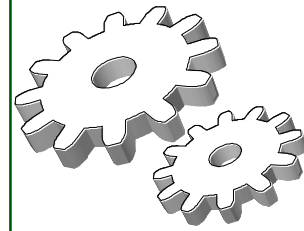


- ❖ Cost depends on #qualifying tuples, and clustering.
 - Cost of finding qualifying data entries (typically small) plus cost of retrieving records (could be large w/o clustering).
 - In example, assuming uniform distribution of names, about 10% of tuples qualify (100 pages, 10000 tuples). With a clustered index, cost is little more than 100 I/Os; if unclustered, upto 10000 I/Os!

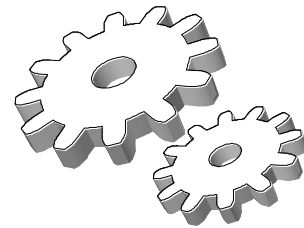
```
SELECT *  
FROM   Reserves R  
WHERE  R.rname < 'C%'
```


Projection

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



- ❖ The expensive part is removing duplicates.
 - SQL systems don't remove duplicates unless the keyword **DISTINCT** is specified in a query.
- ❖ Sorting Approach: Sort on <sid, bid> and remove duplicates. (Can optimize this by dropping unwanted information while sorting.)
- ❖ Hashing Approach: Hash on <sid, bid> to create partitions. Load partitions into memory one at a time, build in-memory hash structure, and eliminate duplicates.
- ❖ If there is an index with both R.sid and R.bid in the search key, may be cheaper to sort data entries!

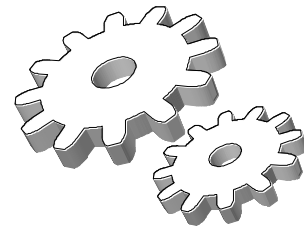


Join: Index Nested Loops

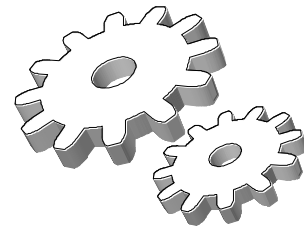
```
foreach tuple r in R do
    foreach tuple s in S where  $r_i == s_j$  do
        add  $\langle r, s \rangle$  to result
```

- ❖ If there is an index on the join column of one relation (say S), can make it the inner and exploit the index.
 - Cost: $M + (M * p_R) * \text{cost of finding matching S tuples}$
- ❖ For each R tuple, cost of probing S index 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



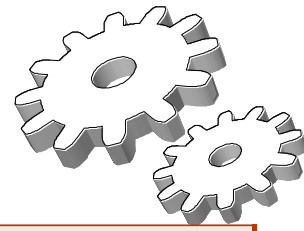
- ❖ Hash-index (Alt. 2) on *sid* of Sailors (as inner):
 - Scan Reserves: 1000 page I/Os, 100×1000 tuples.
 - For each Reserves tuple: 1.2 I/Os to get data entry in index, plus 1 I/O to get (the exactly one) matching Sailors tuple. Total: 220,000 I/Os.
- ❖ Hash-index (Alt. 2) on *sid* of Reserves (as inner):
 - Scan Sailors: 500 page I/Os, 80×500 tuples.
 - For each Sailors tuple: 1.2 I/Os to find index page with data entries, plus cost of retrieving matching Reserves tuples. Assuming uniform distribution, 2.5 reservations per sailor ($100,000 / 40,000$). Cost of retrieving them is 1 or 2.5 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 \geq current S tuple, then advance scan of S until current S-tuple \geq 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 $\langle r, s \rangle$ for all pairs of such tuples.
 - Then resume scanning R and S.
- ❖ R is scanned once; each S group is scanned once per matching R tuple. (Multiple scans of an S group are likely to find needed pages in buffer.)

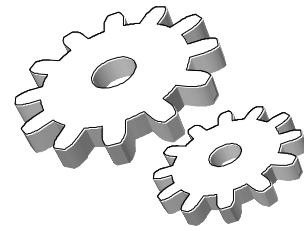
Example of Sort-Merge Join



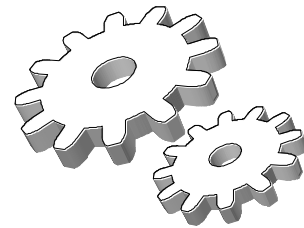
<u>sid</u>	sname	rating	age	<u>sid</u>	<u>bid</u>	<u>day</u>	rname
22	dustin	7	45.0	28	103	12/4/96	guppy
28	yuppy	9	35.0	28	103	11/3/96	yuppy
31	lubber	8	55.5	31	101	10/10/96	dustin
44	guppy	5	35.0	31	102	10/12/96	lubber
58	rusty	10	35.0	31	101	10/11/96	lubber
				58	103	11/12/96	dustin

- ❖ **Cost:** $M \log M + N \log N + (M+N)$
 - The cost of scanning, $M+N$, could be $M*N$ (very unlikely!)
- ❖ With 35, 100 or 300 buffer pages, both Reserves and Sailors can be sorted in 2 passes; total join cost: 7500.

Highlights of System R Optimizer

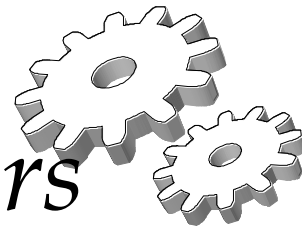


- ❖ Impact:
 - Most widely used currently; works well for < 10 joins.
- ❖ **Cost estimation:** Approximate art at best.
 - Statistics, maintained in system catalogs, used to estimate cost of operations and result sizes.
 - Considers combination of CPU and I/O costs.
- ❖ **Plan Space:** Too large, must 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.



Cost Estimation

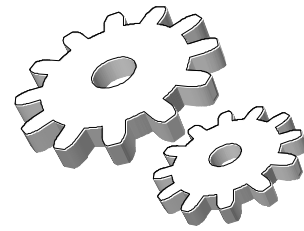
- ❖ For each plan considered, must estimate cost:
 - Must *estimate cost* of each operation in plan tree.
 - Depends on input cardinalities.
 - We've already discussed how to estimate the cost of operations (sequential scan, index scan, joins, etc.)
 - Must also *estimate size of result* for each operation in tree!
 - Use information about the input relations.
 - For selections and joins, assume independence of predicates.



Size Estimation and Reduction Factors

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

- ❖ Consider a query block:
- ❖ 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!
 - Term *col=value* has RF $1/NKeys(I)$, given index *I* on *col*
 - Term *col1=col2* has RF $1/MAX(NKeys(I1), NKeys(I2))$
 - Term *col>value* has RF $(High(I)-value)/(High(I)-Low(I))$



Schema for Examples

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

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

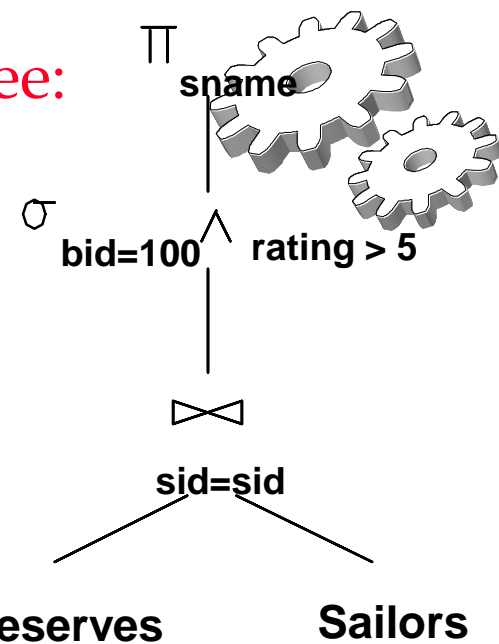
- ❖ Similar to old schema; *rname* added for variations.
- ❖ Reserves:
 - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages.
- ❖ Sailors:
 - Each tuple is 50 bytes long, 80 tuples per page, 500 pages.

Motivating Example

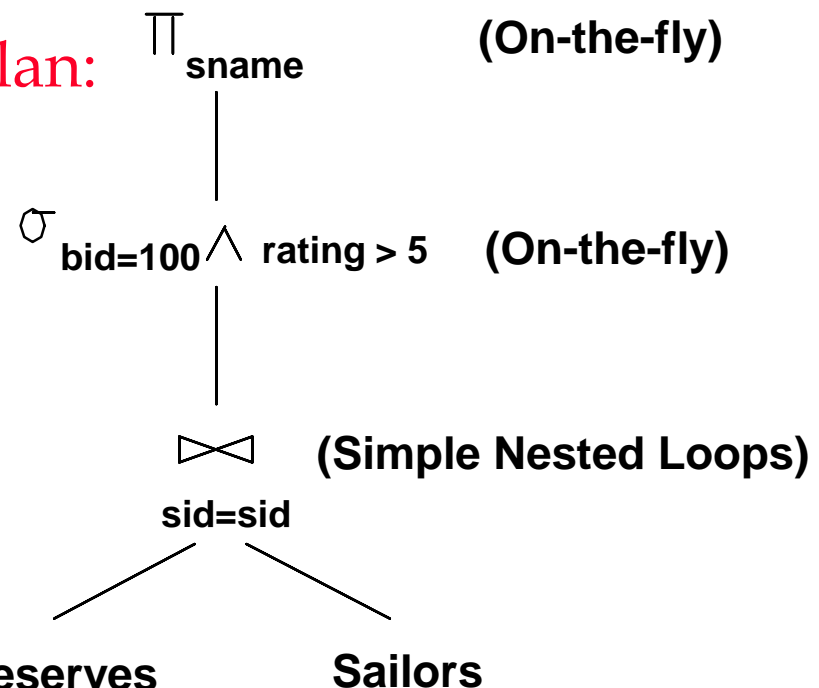
```
SELECT S.sname
FROM Reserves R, Sailors S
WHERE R.sid=S.sid AND
      R.bid=100 AND S.rating>5
```

- ❖ **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.

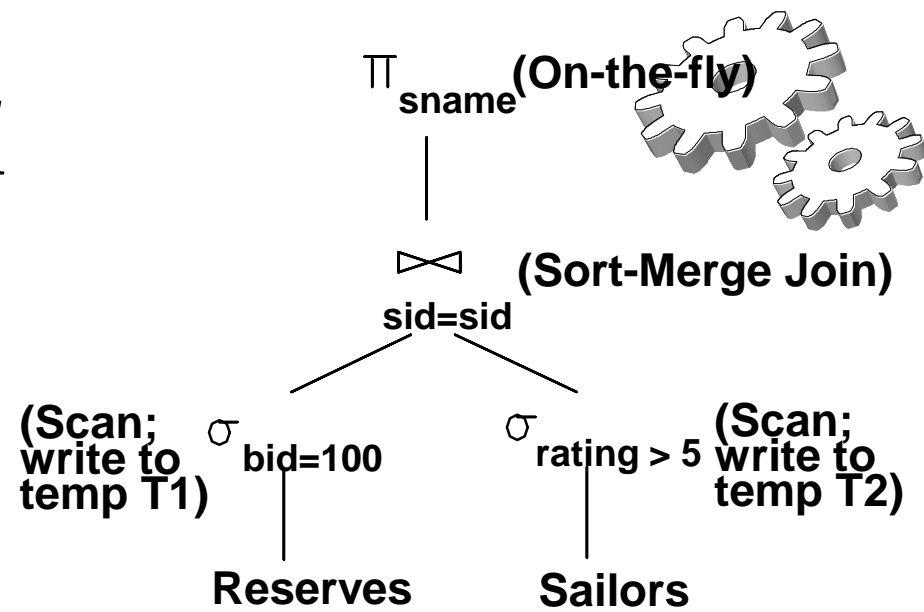
RA Tree:



Plan:



Alternative Plans 1 (No Indexes)



- ❖ **Main difference:** push selects.
- ❖ 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 we have 10 ratings).
 - Sort T1 ($2 \times 2 \times 10$), sort T2 ($2 \times 3 \times 250$), merge ($10 + 250$)
 - **Total: 3560 page I/Os.**
- ❖ If we used BNL join, join cost = $10 + 4 \times 250$, **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 under 250 pages, **total < 2000.**

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).

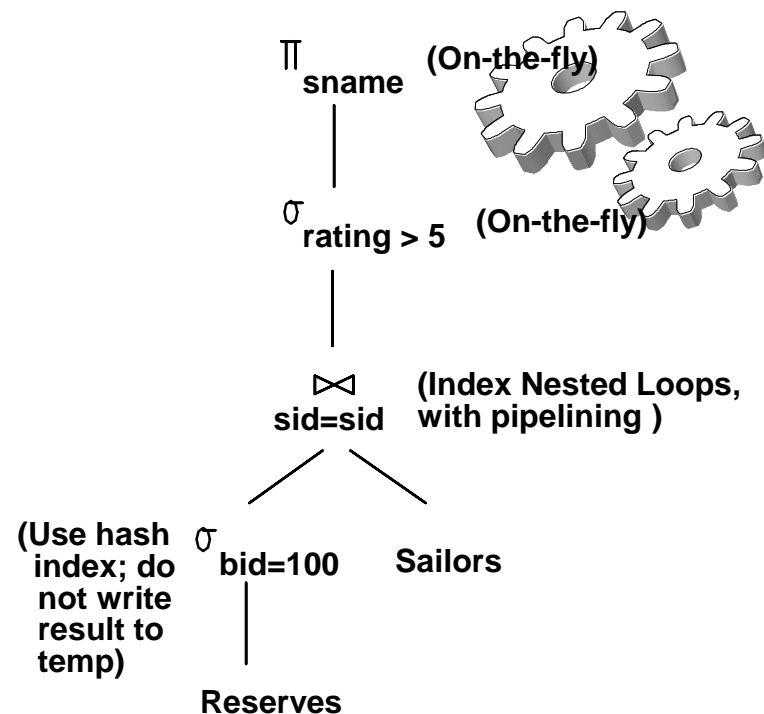
–Projecting out unnecessary fields from outer doesn't help.

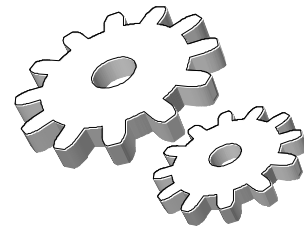
✓ Join column *sid* is a key for Sailors.

–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); 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.