Query Evaluation

EECS 339

Lecture 13

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 <*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.
  - E.g., 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.
  - 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 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, up to 10000 I/Os!

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

Projection

- 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!

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

Join: Index Nested Loops

```
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)
  - M=#pages of R, p_R=# R tuples per page
- 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.

Sophisticated Joins

- Until now, we have used nested loops for joining data
  - This is slow, n^2 comparisons
- How can we do better?
  - Sorting
  - Divide & conquer
- Trade-off in I/O and CPU time for each algo

Sort Merge Join

Equi-join of two tables S & R

|S| = Pages in S; {S} = Tuples in S

|S| ≥ |R|

M pages of memory; M > sqrt(|S|)

Algorithm:

– Partition S and R into memory sized sorted runs, write out to disk
– Merge all runs simultaneously

Total I/O cost: Read |R| and |S| twice, write once

3(|R| + |S|) I/Os

Example

R=1,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R1 = 1,3,4

S1 = 2,3,7

R2 = 6,9,14

S2 = 8,9,12

R3 = 1,7,11

S3 = 4,6,15

R1	R2	R3	S1	S2	S3
1	6	1	2	8	4
3	9	7	3	9	6
4	14	11	7	12	15

OUTPUT

Need enough memory to keep 1 page of each run in memory at a time

Example

R=1,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R1 = 1,3,4 R2 = 6,9,14 R3 = 1,7,11

S1 = 2,3,7 S2 = 8,9,12 S3 = 4,6,15

R1	R2	R3	S1	S2	S3
1	6	1	2	8	4
3	9	7	3	9	6
4	14	11	7	12	15

OUTPUT

Example

R=1,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R1 = 1,3,4 R2 = 6,9,14 R3 = 1,7,11

S1 = 2,3,7 S2 = 8,9,12 S3 = 4,6,15

R1	R2	R3	S1	S2	S3
1	6	1	2	8	4
3	9	7	3	9	6
4	14	11	7	12	15

OUTPUT

Example

R=1,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R1 = 1,3,4 R2 = 6,9,14 R3 = 1,7,11

S1 = 2,3,7 S2 = 8,9,12 S3 = 4,6,15

R1	R2	R3	S1	S2	S3
1	6	1	2	8	4
3	9	7	3	9	6
4	14	11	7	12	15

OUTPUT
(3,3)

Example

R=1,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R1 = 1,3,4 R2 = 6,9,14 R3 = 1,7,11

S1 = 2,3,7 S2 = 8,9,12 S3 = 4,6,15

R1	R2	R3	S1	S2	S3
1	6	1	2	8	4
3	9	7	3	9	6
4	14	11	7	12	15

OUTPUT
(3,3)
(4,4)

Example

R=1,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R1 = 1,3,4　　R2 = 6,9,14　　R3 = 1,7,11

S1 = 2,3,7　　S2 = 8,9,12　　S3 = 4,6,15

R1	R2	R3	S1	S2	S3
1	6	1	2	8	4
3	9	7	3	9	6
4	14	11	7	12	15

OUTPUT
(3,3)
(4,4)

Example

R=1,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R1 = 1,3,4 R2 = 6,9,14 R3 = 1,7,11

S1 = 2,3,7 S2 = 8,9,12 S3 = 4,6,15

R1	R2	R3	S1	S2	S3
1	6	1	2	8	4
3	9	7	3	9	6
4	14	11	7	12	15

OUTPUT
(3,3)
(4,4)
(6,6)

Example

R=1,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R1 = 1,3,4 R2 = 6,9,14 R3 = 1,7,11

S1 = 2,3,7 S2 = 8,9,12 S3 = 4,6,15

R1	R2	R3	S1	S2	S3
1	6 ⇐	1	2	8 ⇐	4
3	9	7 ⇐	3	9	6 ⇐
4 ⇐	14	11	7 ⇐	12	15

OUTPUT
(3,3)
(4,4)
(6,6)
(7,7)

Output in sorted order!

...

Study Break: Sort-Merge Join

- Say we are joining tables:

A=8,20,19,20,3,13,20,18,6,5,4,5

B=15,1,3,13,13,10,19,6,8,15,16,2

If our in-memory runs are of length 4, what will the sorted runs be for A and B?

What is the join output? Walk through the steps of the merge.

Study Break Solution

- A: (8,19,20,20) (3,13,18,20)(4,5,5,6)
- B: (1,3,13,15)(6,10,13,19)(2,8,15,16)

Output: (3,3)(6,6) (8,8)(13,13)(13,13)(19,19)
(output in sorted order)

Simple Hash

Algorithm:

```
Given hash function H(x) → [0,…,P-1]
    where P is number of partitions
for i in [0,…,P-1]:
    for each r in R:
        if H(r)=i, add r to in-memory hash
        otherwise, write r back to disk in R’
    for each s in S:
        if H(s)=i, lookup s in hash, output matches
        otherwise, write s back to disk in S’
    replace R with R’, S with S’
```

Simple Hash I/O Analysis

Suppose P=2, and hash uniformly maps tuples to partitions

Read	|R| + |S|
Write	1/2 (|R| + |S|)
Read	1/2 (|R| + |S|) = 2 (|R| + |S|)

P=3

Read	|R| + |S|
Write	2/3 (|R| + |S|)
Read	2/3 (|R| + |S|)
Write	1/3 (|R| + |S|)
Read	1/3 (|R| + |S|) = 3 (|R| + |S|)

P=4

$$|R| + |S| + 2 * (3/4 (|R| + |S|)) + 2 * (2/4 (|R| + |S|)) + 2 * (1/4 (|R| + |S|))$$

$$= 4 (|R| + |S|)$$

➔ P = n ; n * (|R| + |S|) I/Os

Grace Hash

Algorithm:

Partition:

Suppose we have P partitions, and H(x) → [0…P-1]

Choose P = |S| / M ➔ P ≤ sqrt(|S|) //may need to leave a little slop for imperfect hashing

Allocate P 1-page output buffers, and P output files for R

For each r in R:

  Write r into buffer H(r)

  If buffer full, append to file H(r)

Allocate P output files for S

For each s in S:

  Write s into buffer H(s)

  if buffer full, append to file H(s)

Join:

For i in [0,…,P-1]

  Read file i of R, build hash table

  Scan file i of S, probing into hash table and outputting matches

Total I/O cost: Read |R| and |S| twice, write once

3(|R| + |S|) I/Os

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2

P output buffers

F0	F1	F2

P output files

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2
		5

F0	F1	F2

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2
	4	5

F0	F1	F2

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2
3	4	5

F0	F1	F2

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2
3	4	5
6		

F0	F1	F2

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2
3	4	5
6		

Need to flush R0 to F0!

F0	F1	F2

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2
	4	5

F0	F1	F2
3		
6		

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2
9	4	5

F0	F1	F2
3		
6		

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2
9	4	5
		14

F0	F1	F2
3		
6		

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2
9	4	5
	1	14

F0	F1	F2
3		
6		

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2
9	4	5
	1	14

F0	F1	F2
3		
6		

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2
9		5
		14

F0	F1	F2
3	4	
6	1	

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2
9	7	5
		14

F0	F1	F2
3	4	
6	1	

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2
9	7	5
		14

F0	F1	F2
3	4	
6	1	

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2
9	7	

F0	F1	F2
3	4	5
6	1	14

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2
9	7	11

F0	F1	F2
3	4	5
6	1	14

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R0	R1	R2

F0	F1	F2
3	4	5
6	1	14
9	7	11

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R Files

F0	F1	F2
3	4	5
6	1	14
9	7	11

S Files

F0	F1	F2
3	7	2
12	4	8
9		
15		
6		

Example

P = 3; H(x) = x mod P

Matches:

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R Files

F0	F1	F2
3	4	5
6	1	14
9	7	11

Load F0 from R into memory

S Files

F0	F1	F2
3	7	2
12	4	8
9		
15		
6		

Example

P = 3; H(x) = x mod P

Matches:

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R Files

F0	F1	F2
3	4	5
6	1	14
9	7	11

Load F0 from R into memory

S Files

F0	F1	F2
3	7	2
12	4	8
9		
15		
6		

Scan F0 from S

Example

P = 3; H(x) = x mod P

Matches:
3,3

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R Files

F0	F1	F2
3	4	5
6	1	14
9	7	11

Load F0 from R into memory

S Files

F0	F1	F2
3	7	2
12	4	8
9		
15		
6		

Scan F0 from S

Example

P = 3; H(x) = x mod P

Matches:
3,3

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R Files

F0	F1	F2
3	4	5
6	1	14
9	7	11

Load F0 from R into memory

S Files

F0	F1	F2
3	7	2
12	4	8
9		
15		
6		

Scan F0 from S

Example

$P = 3; H(x) = x \bmod P$

Matches:
3,3
9,9

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R Files

F0	F1	F2
3	4	5
6	1	14
9	7	11

Load F0 from R into memory

S Files

F0	F1	F2
3	7	2
12	4	8
9		
15		
6		

Scan F0 from S

Example

P = 3; H(x) = x mod P

Matches:
3,3
9,9

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R Files

F0	F1	F2
3	4	5
6	1	14
9	7	11

Load F0 from R into memory

S Files

F0	F1	F2
3	7	2
12	4	8
9		
15		
6		

Scan F0 from S

Example

$P = 3; H(x) = x \bmod P$

Matches:
3,3
9,9
6,6

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R Files

F0	F1	F2
3	4	5
6	1	14
9	7	11

Load F0 from R into memory

S Files

F0	F1	F2
3	7	2
12	4	8
9		
15		
6		

Scan F0 from S

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

Matches:
3,3
9,9
6,6

R Files

F0	F1	F2
3	4	5
6	1	14
9	7	11

S Files

F0	F1	F2
3	7	2
12	4	8
9		
15		
6		

Example

P = 3; H(x) = x mod P

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

Matches:
3,3
9,9
6,6
7,7
4,4

R Files

F0	F1	F2
3	4	5
6	1	14
9	7	11

S Files

F0	F1	F2
3	7	2
12	4	8
9		
15		
6		

Example

P = 3; H(x) = x mod P

Matches:
3,3
9,9
6,6
7,7
4,4

R=5,4,3,6,9,14,1,7,11

S=2,3,7,12,9,8,4,15,6

R Files

F0	F1	F2
3	4	5
6	1	14
9	7	11

S Files

F0	F1	F2
3	7	2
12	4	8
9		
15		
6		

Execution Costs

Notation: P partitions / passes over data; assuming hash is O(1)

Sort-Merge	Simple Hash	Grace Hash
I/O: 3 (|R| + |S|) CPU: O(P x {S}/P log {S}/P)	I/O: P (|R| + |S|) CPU: O({R} + {S})	I/O: 3 (|R| + |S|) CPU: O({R} + {S})

Grace hash is generally a safe bet, unless memory is close to size of tables, in which case simple can be preferable

Extra cost of sorting makes sort merge unattractive unless there is a way to access tables in sorted order (e.g., a clustered index), or a need to output data in sorted order (e.g., for a subsequent ORDER BY)

Study Break: Grace Hash Join

- Say we are joining tables:

A=8,20,19,20,3,13,20,18,6,5,4,5

B=15,1,3,13,13,10,19,6,8,15,16,2

If we have four partitions, what will the hash partitions be for A and B?

Walk through the steps for producing this join's output.

Study Break Solution

- Partitions:
  - A0: (8,20,20,20,4) A1: (13,5,5) A2: (18,6) A3: (19,3)
  - B0: (8,16) B1: (1,13,13) B2: (2, 6, 10) B3: (15,3,19,15)
- Execution:
  - Join A0 w/B0 (produces (8,8))
  - Join A1 w/B1 (produces (13,13), (13,13))
  - …

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

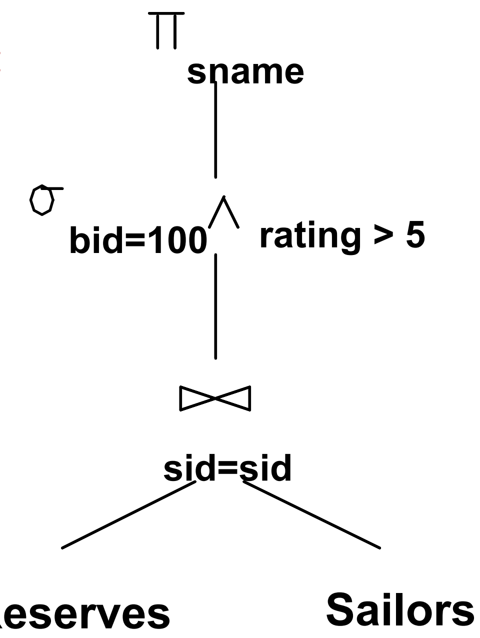
Sailors (*sid*: integer, *sname*: string, *rating*: integer, *age*: real)
Reserves (*sid*: integer, *bid*: integer, *day*: dates, *rname*: string)

- 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

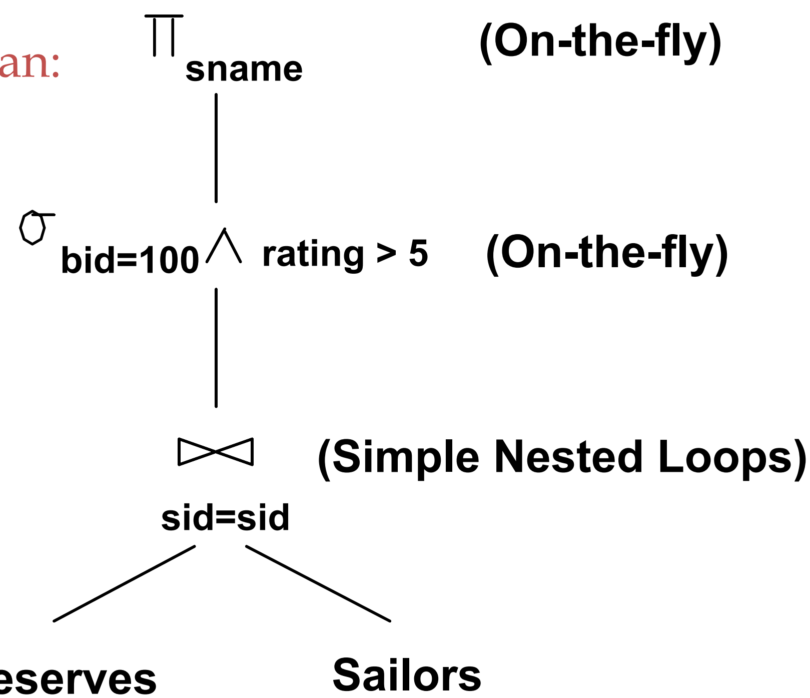
```
SELECT  S.sname
FROM  Reserves R, Sailors S
WHERE  R.sid=S.sid AND
   R.bid=100 AND S.rating>5
```

RA Tree:

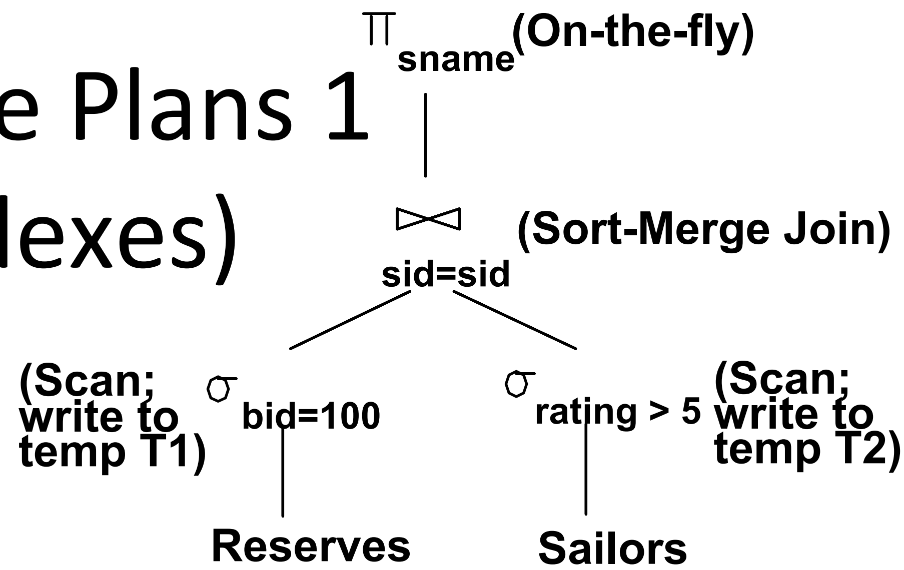


- Cost: 500+500*1000 I/Os
- By no means the worst plan!
- Misses several opportunities: selections could have been \`pushed' earlier, no use is made of any available indexes, etc.
- *Goal of optimization:* To find more efficient plans that compute the same answer.

Plan:

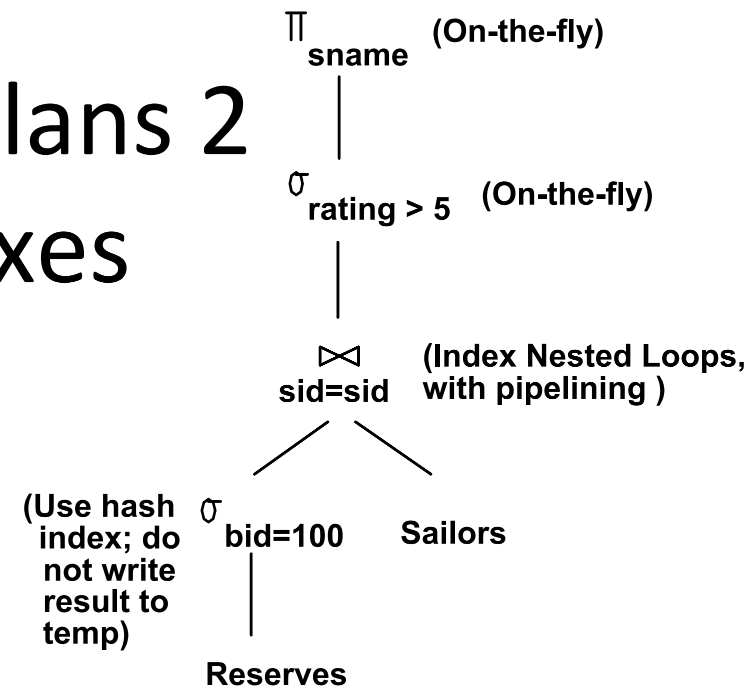


Alternative Plans 1 (No Indexes)



- *Main difference:* <u>*push selects.*</u>
- 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*2*10), sort T2 (2*3*250), merge (10+250)
  - Total: 3560 page I/Os.
- If we used BNL 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 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.
  - Especially for joins!
- 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.