Reminder: Cost of Simple Nested Loops Join

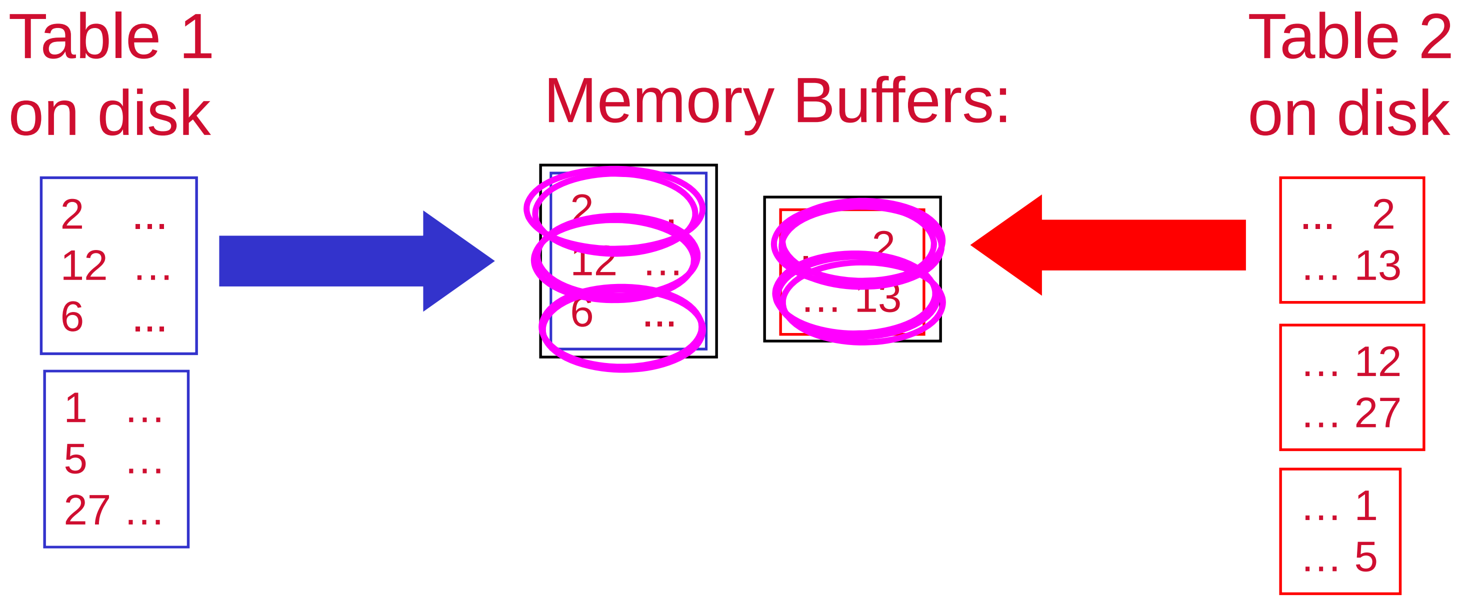
```
Join on ith column of R and jth column of S
foreach tuple r in R do
        foreach tuple s in S do
                if $r_i == s_j$ then add <r, s> to result
```

For each tuple in R, scan the entire relation S.

- Cost: $M + (p_R * M) * N = 1000 + 100*1000*500$ I/Os
- 50,001,000 I/Os ≈ 500,010 seconds ≈ 6 days (If we assume 100 I/Os per second)

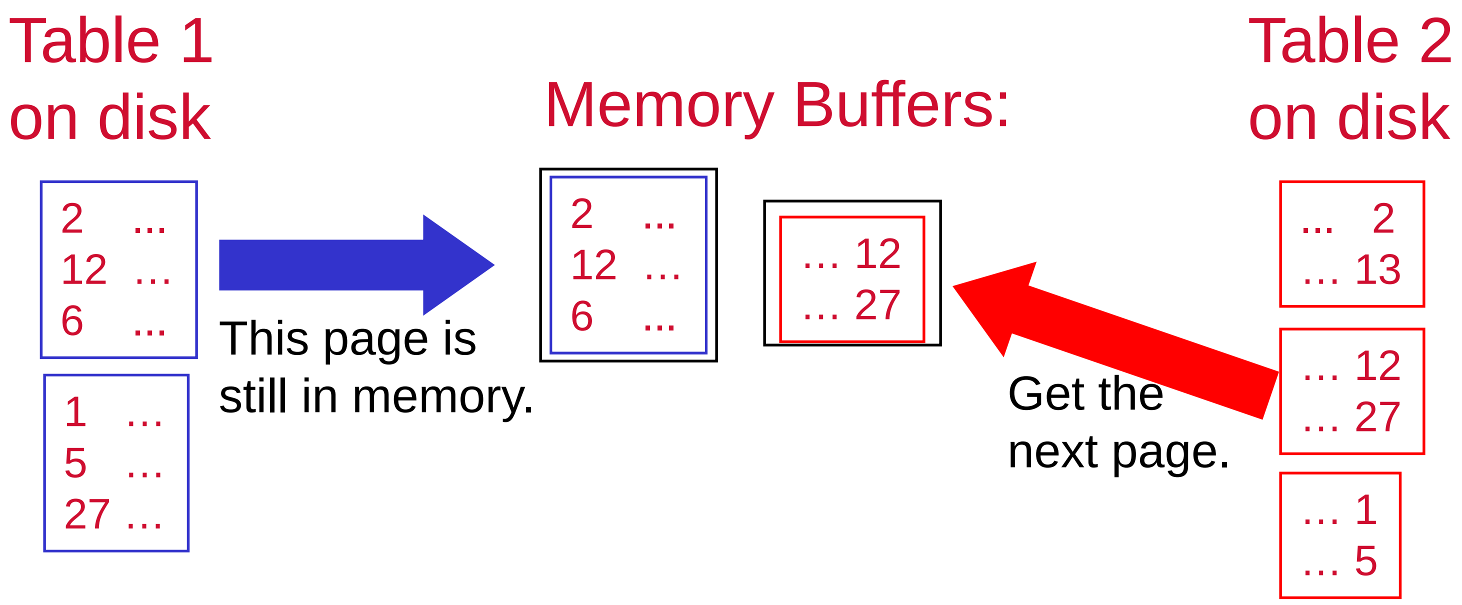
$M = 1000$ pages in R, $p_R = 100$ tuples/page (100,000 Reserves)
$N = 500$ pages in S, $p_S = 80$ tuples per page (40,000 Sailors)

Better: Page-Oriented Nested Loops Join



Once we’ve got these two pages in memory, check every combination from one page to the other page!

Page-Oriented Nested Loops Join (cont.)



Do the same thing…compare all combinations in memory - between these two pages!

Cost of Page-oriented Nested Loops Join

```
for each page of tuples r in R do
        for each page of tuples s in S do
                (match all combinations in memory)
                if ri == sj  then add <r, s> to result
```

For each *page* of R, get each *page* of S, write out matching pairs of tuples <r, s>.

Cost: M + M*N = 1000 + 1000*500 = 501,000 (R outer)

Cost: N + N*M = 500 + 500*1000 = 500,500 (S outer)

Therefore, use smaller relation as outer relation.

500,000 I/Os ≈ 1.4 hours (assume 100 I/Os/second)

Compare with simple nested loops:

$M + (p_R * M) * N = 1000 + 100*1000*500 = 50{,}001{,}000$

Page-oriented Nested Loops Join

```
for each page of tuples r in R do
          for each page of tuples s in S do
                    (match all combinations in memory)
                    if ri == sj  then add <r, s> to result
```

For each *page* of R, get each *page* of S, write out matching pairs of tuples <r, s>.

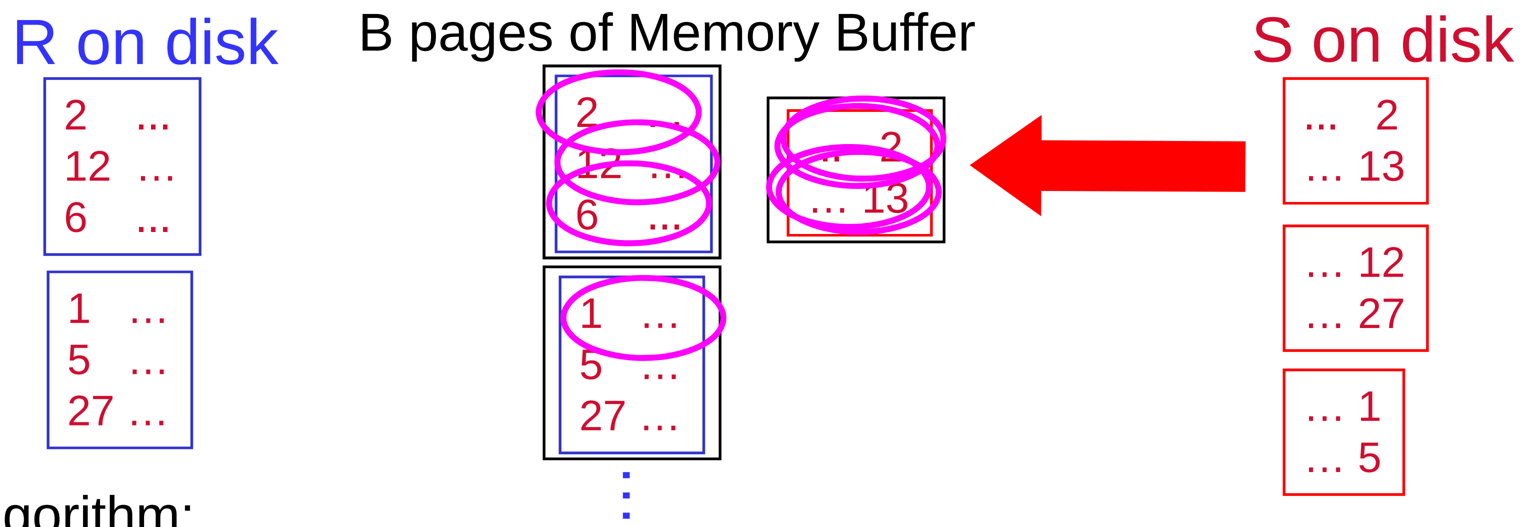
Cost: M + M*N = 1000 + 1000*500 = 501,000 (R outer)

Cost: N + N*M = 500 + 500*1000 = 500,500 (S outer)

Therefore – typically use smaller relation as outer relation.

500,000 I/Os ≈ 1.4 hours

The best loops-based join algorithm: Block Nested-Loops Join (use a block of buffers)



- Algorithm:
  - One page is assigned to be the output buffer (not shown on this slide)
  - One page assigned to input from S, B-2 pages assigned to input from R

```
Until all of R has been read {
    Read in B-2 pages of R
    For each page in S {
        Read in the single S page
        Check pairs of tuples in memory and output if they match } }
```

Cost: M + (M/(B-2))*N.

For B=35, cost is 1000 + 1000*500/33 = 16,000 I/Os ≈ 3 minutes

Comparing the nested loops join algorithms

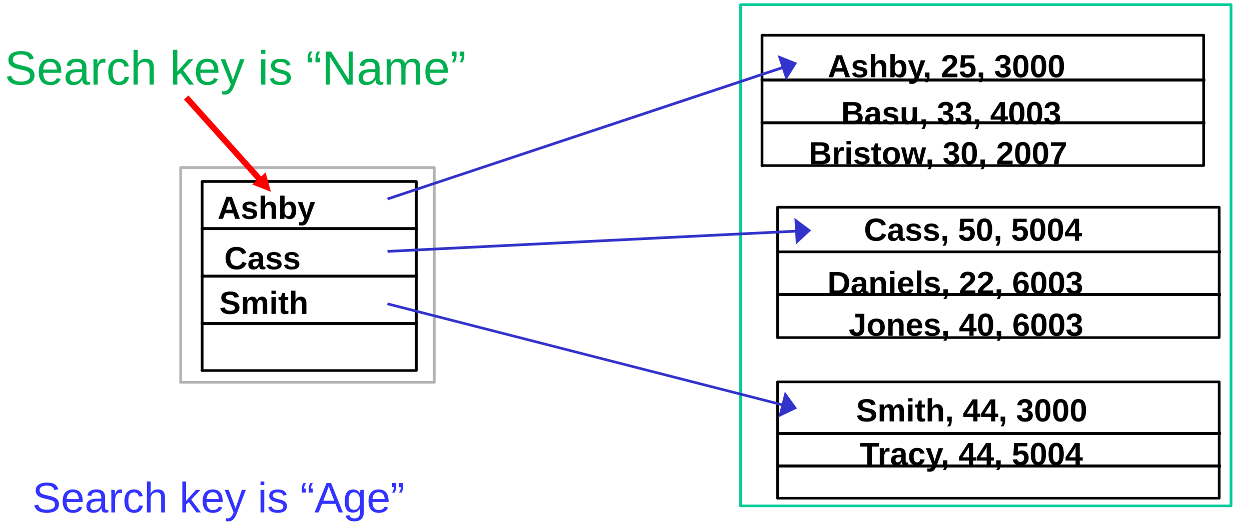
Algorithm	Number of times to read inner table	Cost formula (with example M = 1000, p_r = 100, b = 52, N = 500)
simple nested loops join	Once for each row in outer table	M + $(M*p_r)$ * N 1000 + (1000*100) * 500 1000 + 100,000 * 500 1000 + 50,000,000
page-oriented nested loops join	Once for each PAGE of rows in outer table	M + M*N 1000 + 1000 * 500 1000 + 500,000
block nested loops join	Once for each b-2 pages of rows in outer table	M + (M/(b-2) * N 1000 + (1000/50) * 500 1000 + 20 * 500 1000 + 10,000

A few comments about costs

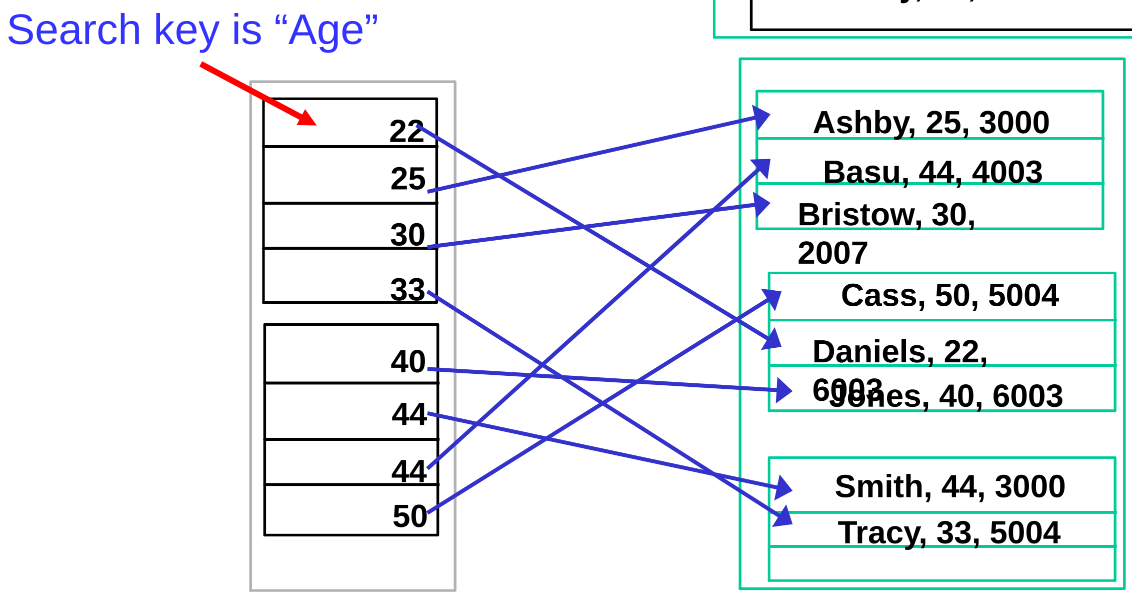
These cost formulas are much simpler that any real cost formulas, for a number of reasons.

- Real formulas take other costs into account (including CPU time)
- Real systems use buffers and caches to assist with I/Os. (So what we estimate as independent I/Os aren’t necessarily actually I/Os; they might be “reading” data from memory or from a disk cache.)
- Note: you can’t control/know what happens … e.g., when you read data from a table or query answer using a cursor. (The OS/DBMS handles these details.)

Clustered vs. Unclustered Index (reminder)



Records are sorted by "Name" in the file; clustered!



Records are sorted by "Name" in the file; but since the index is on Age, this index is unclustered!

Sorting using indices

- With a clustered index, you read the file (in sorted order) by reading blocks as directed by the leaf level of the index. Cost = M (with a couple of extra I/Os to work through the index, the first time). I/Os are close to sequential. GOOD IDEA
- With an unclustered index, you read the rows (one at a time) in sorted order using the index. But Cost = M*pr and the I/Os are scattered over the disk. BAD IDEA

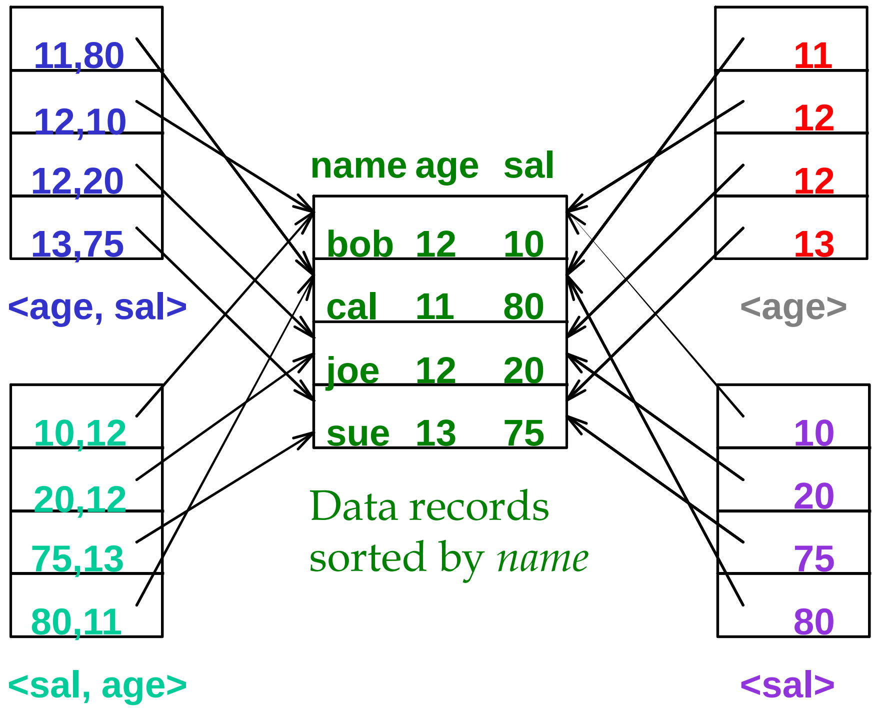
Unclustered Index

- Good for getting one (or a few) records based on the search key
- Can be very much worse for getting a large number of records (in a range search or a sort).
- Question: what else is an unclustered index good for?

Using Composite Search Keys (revisited)

Note: ALL of these indices are dense/unclustered.

- age = 12 use <age> (best case), use all except <sal,age>
- age = 12 and sal = 20 use <age,sal> or <sal,age> (best case) use all of these indices
- age=12 and sal > 10 use <age,sal> (best case), use <age>, <sal,age> or <sal> may not help
- age > 12 and sal > 30 all could be used, but they may not help



Range queries, particularly when the range of records you need is large, are not very well supported by unclustered indices.

Index Nested Loops Join

```
foreach tuple r in R do
        foreach tuple s in S where $r_i == s_j$
                Use the Index to find s do
                    add <r, s> to result
```

If there is an index on the search key s_j then can use the index on the inner table - get matching tuples!

Cost: M + ((M*p_R) * cost of finding matching S tuples)

M + ((M*p_R) * (I/Os to find index + 1 to get the data)

= 500 + (500*80*4) = 160,500 ≈ 1/2 hour (Reserves as inner)

= 1000 + (1000*100*3) = 301,000 ≈ 1 hour (Sailor as inner)

These could be smaller – if top levels of B+ tree are in memory.

Could be 0… if entire index is in memory.

For each R tuple, cost of probing S index is about 2-4 for B^+ tree.

80,500 (Reserves as inner) 15 min.; 151,00 (Sailor as inner) 30 min.

Index nested loops join

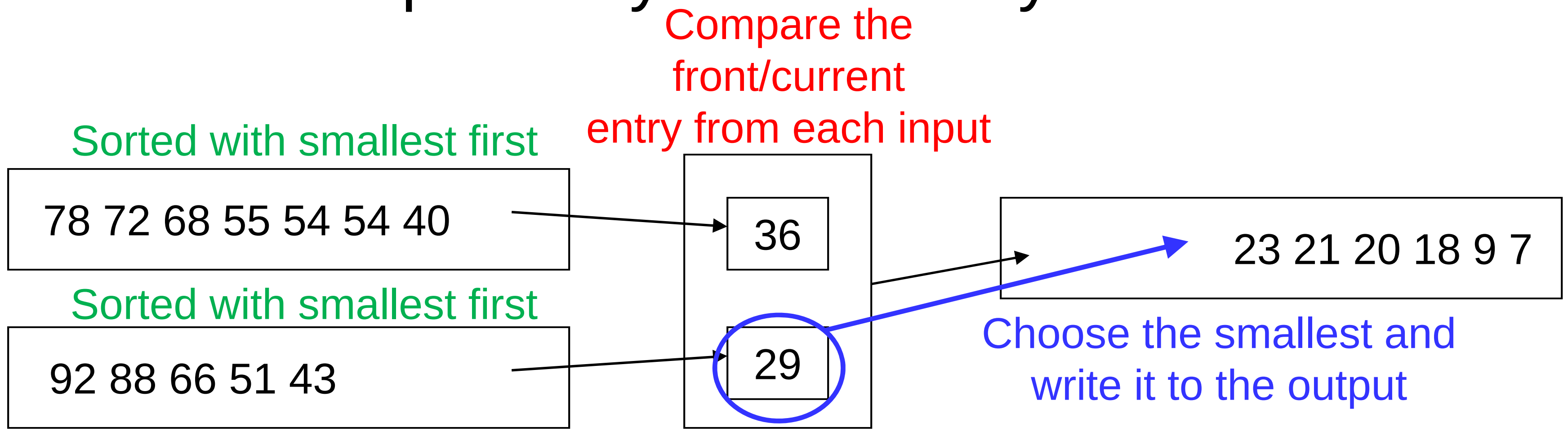
- Advantage: access exactly the right records in the inner loop (for equi-join).
- You can use a clustered or an unclustered index for an index nested loops join.
- Disadvantage: (necessarily) you have one I/O per row (in the table in the outer loop) rather than one I/O per page (of rows in the outer loop).

N-Way External Sorting

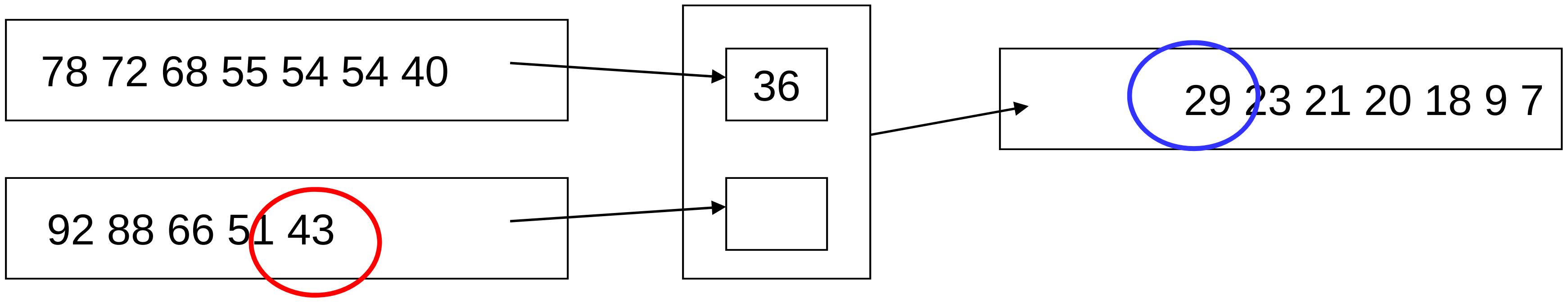
- First pass: read until memory is full, sort it, write it back out to disk to create one "sorted run"/small sorted files. Repeat until you've read the entire input. Costs 2*M.
- Do an n-way merge rather than a 2-way merge
- Each pass does 2*M I/Os (where M is number of pages in the table). Read & Write entire file in each pass.
- Number of passes depends on how many pages of memory are devoted to sorting
  - #Passes = Ceiling (Log $_{B-1}$ (M/B)) M – pages, B - buffers
  - Can sort 100 million pages in 4 passes with 129 pages of memory space
- Can sort M pages using B memory pages in 2 passes if sqrt(M) <= B (this condition is satisfied often)

External Sorting – merge phase

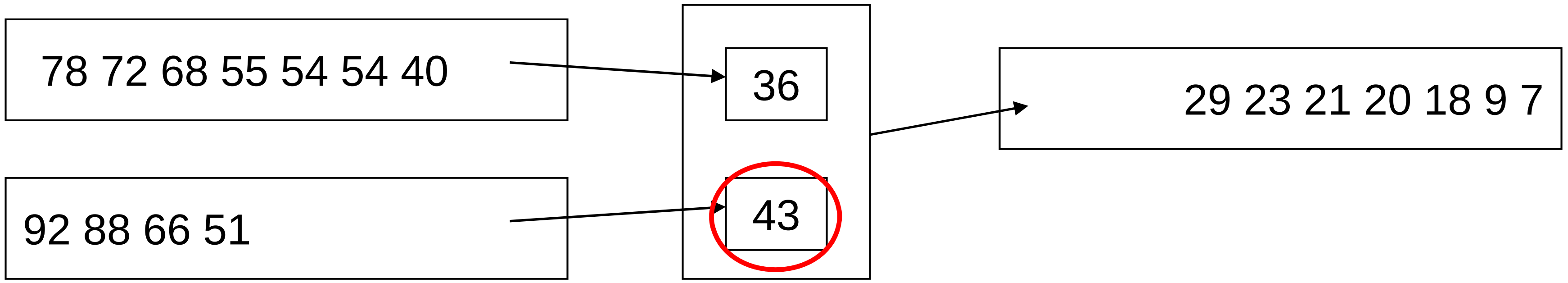
- Various relational operator algorithms require sorting a table
- Issue: table won’t fit in memory
- Approach: Use merge-sort where sorted runs can be read sequentially into memory


Compare the front/current entry from each input
Sorted with smallest first
78 72 68 55 54 54 40
Sorted with smallest first
92 88 66 51 43
36
29
23 21 20 18 9 7
Choose the smallest and write it to the output

External Sorting – merge phase (cont.)

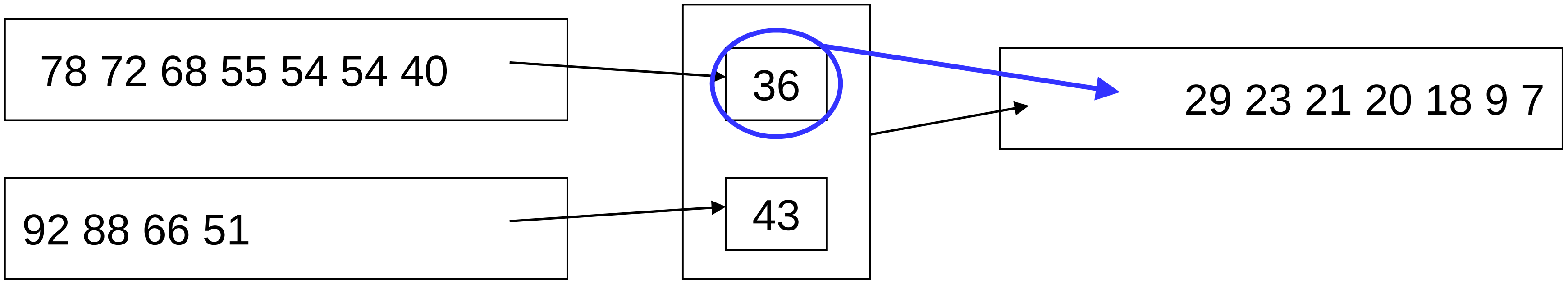


Read the next input from the appropriate file

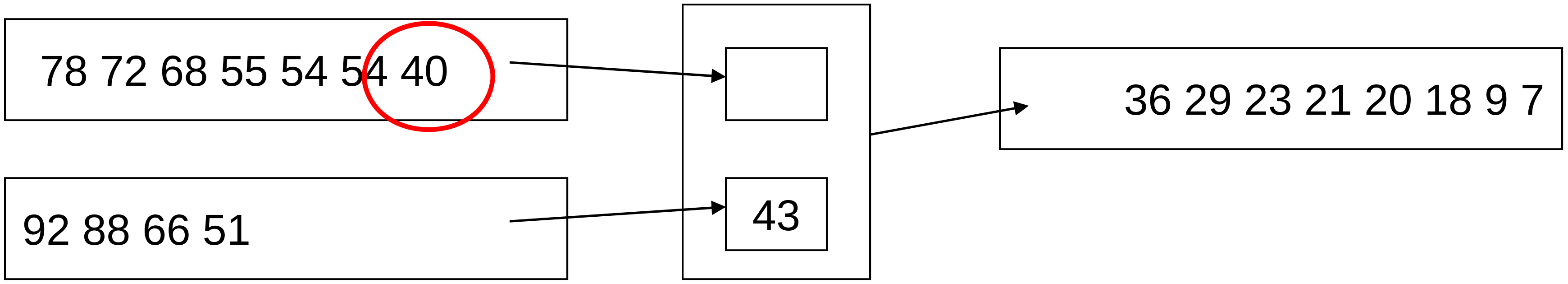


Repeat the process; compare the two numbers, choose the smallest

External Sorting – merge phase (cont.)



Repeat the process; compare the two numbers, choose the smallest



Read the next input from the appropriate file

N-Way External Sorting (repeated)

- First pass: read until memory is full, sort it, write it back out to disk to create one “sorted run”/small sorted files. Repeat until you’ve read the entire input.
- Do an n-way merge rather than a 2-way merge
- Each pass does 2*M I/Os (where M is number of pages in the table). Read & Write entire file in each pass.
- Number of passes depends on how many pages of memory are devoted to sorting
  - #Passes = Ceiling (Log $_{B-1}$ (M/B)) M – pages, B - buffers
  - Can sort 100 million pages in 4 passes with 129 pages of memory space
- Can sort M pages using B memory pages in 2 passes if sqrt(M) <= B (this condition is satisfied often)

Sort-Merge Join

1. Sort R on join attribute
2. Sort S on join attribute
3. Merge R and S
   - 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 Ri (*current R group*) and all S tuples with same value in Sj (*current S group*) *match*; output <r, s> 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. Depends on the size of the group! If the matching group is small - matching is in memory.

Best case: cost is: Cost to sort R + Cost to sort S + (M+N) assuming all matches fit in memory

Worst case: R and S all have the same value - thus the matching group is the entire relation, for R and for S. Cost is: Cost to sort R + Cost to sort S + (M*N)

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: (cost to sort R) + (cost to sort S) + (M+N) (in memory matches)

With 35 buffers, Reserves and Sailors can each be sorted in 2 passes

Cost is: 4 * 1000 + 4 * 500 + 1000 + 500 = 7500

(we multiply by 4 because there are 2 passes, and we read and write each page, each pass)

Sort Merge Join

- Best case: cost of merge is M + N
  This happens when there are small numbers of “collisions” where a join value from the left matches MANY join values from the right table. Best case: just ONE matching record.
- Worst case: cost of merge is M * N
  This happens when there are large “collisions”. Worst case: all records have same join value. This requires “cross product”.

Cost of Sort-Merge

(cost to sort R)+(cost to sort S)+(cost to merge)

Cost to sort M pages in 2 passes = 4*M. Why?

Cost to merge is typically M+N. Why?

If both R and S can be sorted in 2 passes, then

Cost is: 4M+4N+(M+N) = 5*(M+N)

There is an optimization (page 462 in our text) that improves this to 3*(M+N)

Thus the cost of joining Sailors and Reservations, assuming there are enough buffers to sort each table in two passes and best case for merge, is

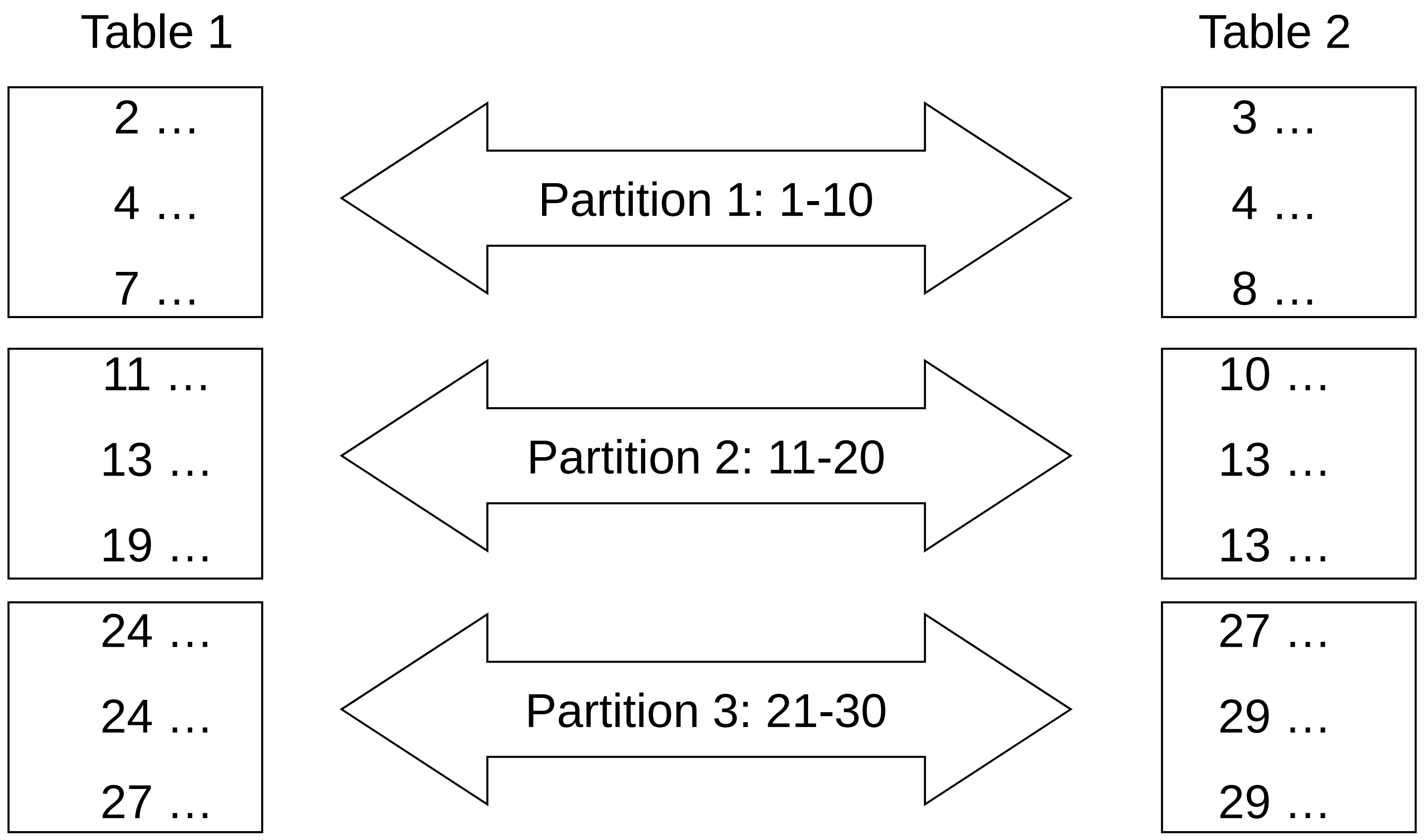
5*(M+N) = 7500 Pages

Hash Join

Simple case – S fits in main memory
- Build an in-memory hash index for S
- Proceed as for index nested-loops join

Harder case – neither R nor S fits in memory
- Divide them both in the same way (1 pass) so that each partition of S fits in memory
- Do in-memory matching on each pair of matching partitions

Partitioning



Use Hash Function Instead of Ranges

- No guarantee we can find ranges of values that will divide Table 2 into roughly equal-sized partitions
- Apply hash function h to join value

  Partition 1: h(val) = 1

  Partition 2: h(val) = 2

  Partition 3: h(val) = 3

Hash Join Cost

- Cost to partition R: 2M (read & write)
- Cost to partition S: 2N (read & write)
- Cost to join partitions: M+N
- Total: 3*(M+N), same as sort-merge with the optimization.

Comparison of Approximate Costs of Joining R and S, assuming 100 I/Os/second

Algorithm	I/Os	Time
Simple Nested Loops	50,000,000	6 days
Page Nested Loops	500,000	1.4 hours
Block Nested Loops*	16000	3 minutes
Index Nested Loops	160,500	½ hour
Sort-Merge**	4,500	1 minute
Hash join**	4,500	1 minute

*Assuming 35 buffer pages

**Assuming appropriate files, M, satisfy sqrt(M) < pages of buffer and assuming best case behavior (match groups fit in memory).

All operators have associated algorithms

- Select, project
- Set operators (UNION, UNION ALL, etc.)
- Grouping and aggregate functions
- Sorting (ORDER BY)

Algorithms for other operators

- Table scan
- Index retrieval (select operator)
- Index-only scan (project operator)

- What might a simple algorithm be for eliminating duplicates (e.g., for DISTINCT or for UNIOIN)?

Query Optimization

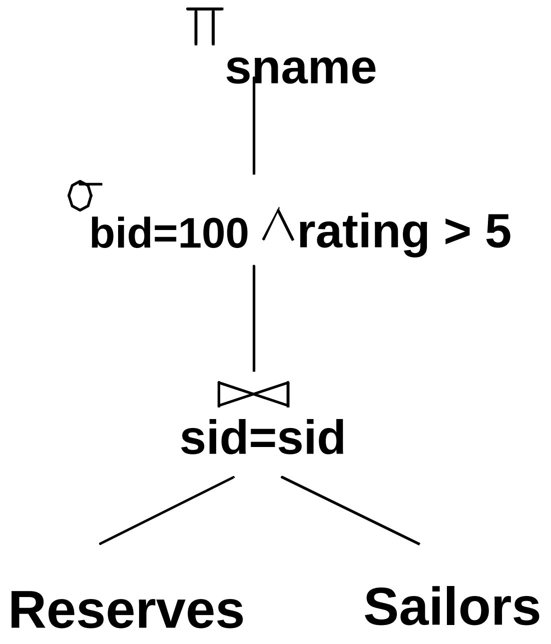
- Translate SQL query into a query tree
  (operators: relational algebra plus a few other ones)
- Generate other, equivalent query trees
  (e.g., using relational algebra equivalences)
- For each possible query tree:
  - select an algorithm for each operator (producing a query *plan*)
  - estimate the cost of the plan
- Choose the plan with lowest estimated cost - of the plans considered (which is not necessarily all possible plans)

Initial Query Tree - Equivalent to SQL
(without any algorithms selected)

SQL Query:

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

Relational Algebra Tree:



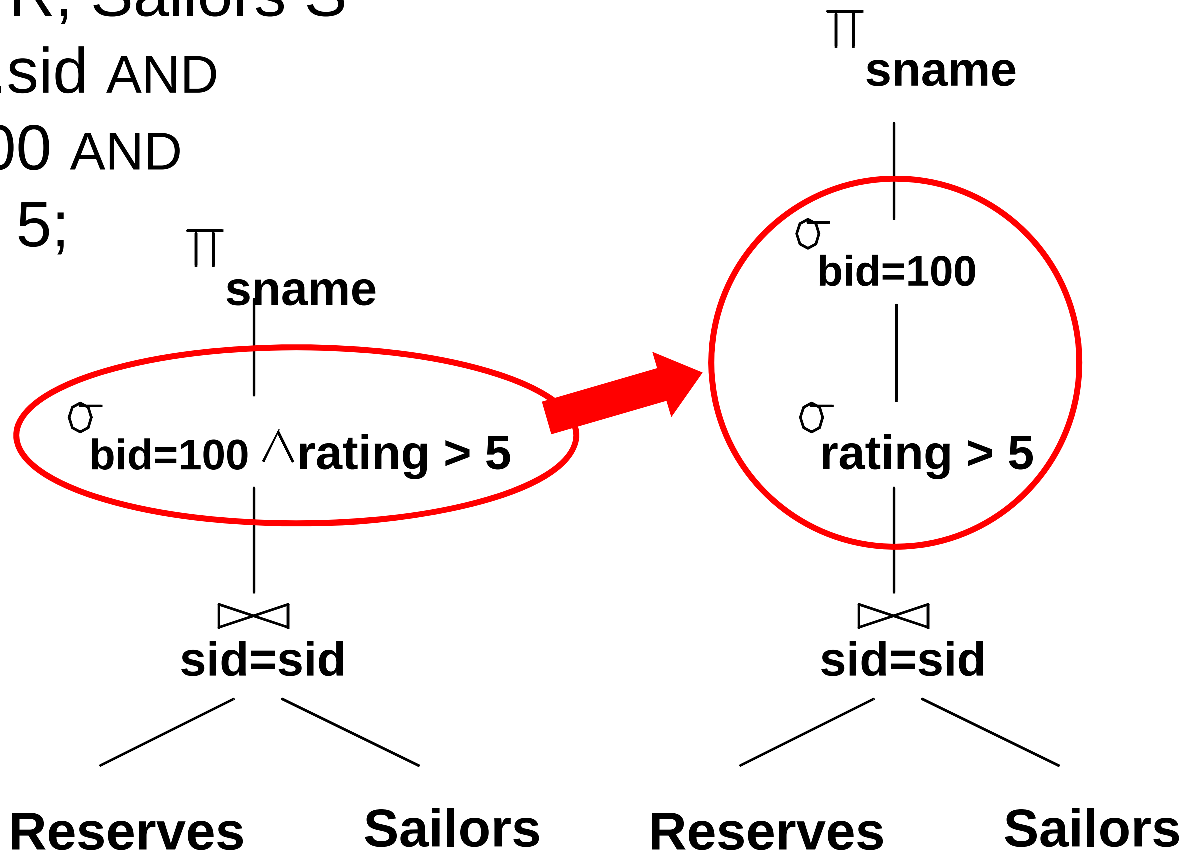
π_{sname}
$\sigma_{\text{bid=100} \wedge \text{rating > 5}}$
$\bowtie_{\text{sid=sid}}$
Reserves
Sailors

Relational Algebra Equivalence: Cascade (of Uncascade) Selects

- $\sigma_{c1 \wedge \ldots \wedge cn}(R) \equiv \sigma_{c1}(\ldots \sigma_{cn}(R))$
- This symbol means equivalence.
- So you can replace $\sigma_{c1}(\ldots \sigma_{cn}(R))$ with $\sigma_{c1 \wedge \ldots \wedge cn}(R)$
- And you can replace $\sigma_{c1 \wedge \ldots \wedge cn}(R)$ with $\sigma_{c1}(\ldots \sigma_{cn}(R))$
- If you have several conditions connected by “AND” in a select operator, then you can apply them one at a time.

Example: $\sigma_{c1 \wedge \ldots \wedge cn}(R) \equiv \sigma_{c1}(\ldots \sigma_{cn}(R))$

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



Relational Algebra Equivalence

$\sigma_c(R \bowtie S) \equiv \sigma_c(R) \bowtie S$

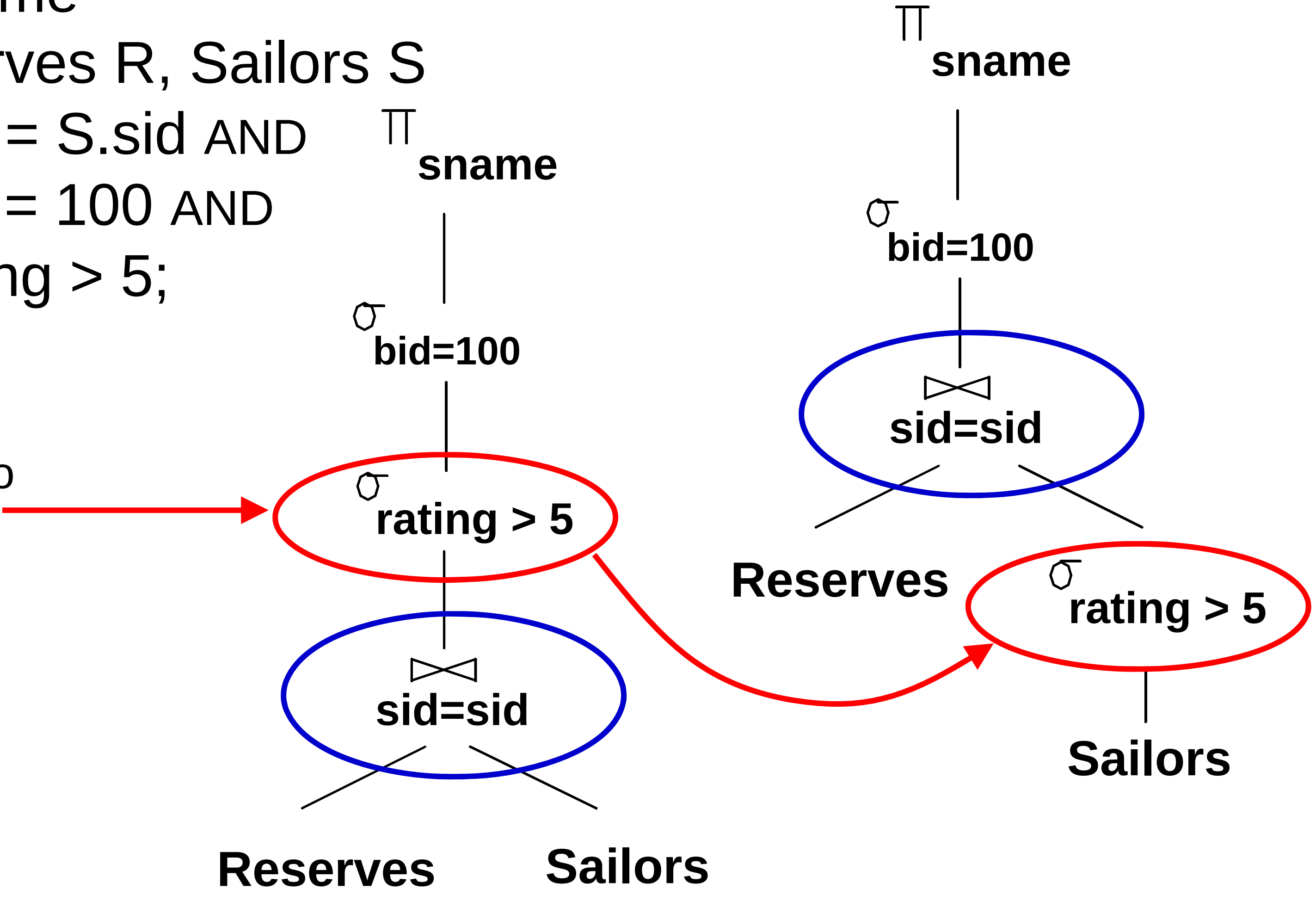
Given a select operation following a join, if the select condition applies ONLY to one of the tables (R, in this example), then you can introduce a new select operator (before the join operator) with that condition.

This is called *pushing down* a SELECT.

Example: $\sigma_c(R \bowtie S) \equiv \sigma_c(R) \bowtie S$

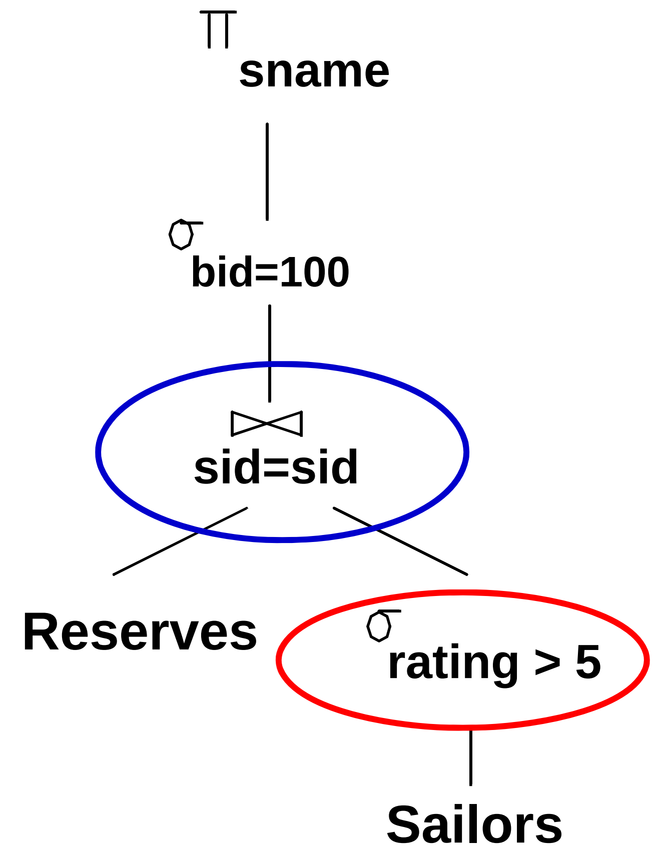
SELECT S.sname
FROM Reserves R, Sailors S
WHERE R.sid = S.sid AND
R.bid = 100 AND
S.rating > 5;

This applies only to the Sailors table!



Example: $\sigma_c(R \bowtie S) \equiv \sigma_c(R) \bowtie S$ (cont.)

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



What are the advantages of "pushing" a select past a join operator?

What are the disadvantages of "pushing" a select past a join operator?

Relational Algebra Equivalences

- Selections:

  $\sigma_{c1 \wedge \dots \wedge cn}(R) \equiv \sigma_{c1}(\dots \sigma_{cn}(R))$ Selects *Cascade*

  $\sigma_{c1}(\sigma_{c2}(R)) \equiv \sigma_{c2}(\sigma_{c1}(R))$ Selects *Commute*

- Projections:

  $\pi_a (R) \equiv \pi_a (\pi_{a1} (\dots \pi_{an}(R)))$ If each a_i contains a. Only last project matters

- Joins:

  $R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T$ Joins are *Associative*

  $R \bowtie S \equiv S \bowtie R$ Joins *Commute*

Try to prove that: $R \bowtie (S \bowtie T) \equiv (T \bowtie R) \bowtie S$

Some Rel. Algebra Equivalences have Constraints

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

Is this always true? No matter what the conditions are … no matter what table R we use?

What about this one?

$\pi_a(\sigma_c(R)) \equiv \sigma_c(\pi_a(R))$

Is this always true? No matter what the condition is? No matter what project list is used?

Equivalences with 2 or More Operations

- Projection commutes with a selection
  PROVIDED that the selection uses attributes that are retained by the projection (*c*'s *attrs* $\subseteq$ *a*):

  $$\pi_a(\sigma_c(R)) \equiv \sigma_c(\pi_a(R))$$

- A cross-product can be converted to a join
  PROVIDED that the selection condition involves attributes of the tables involved in a cross-product

  $$\sigma_c(R \times S) \equiv R \bowtie_c S$$

- A selection can be pushed past a cross product (or join)
  PROVIDED the select uses just attributes of R

  $$\sigma_c(R \bowtie S) \equiv \sigma_c(R) \bowtie S$$

Equivalences with 2 or More Operations

- Join distributes over the various set operators (union, intersection, difference. For example, with union:

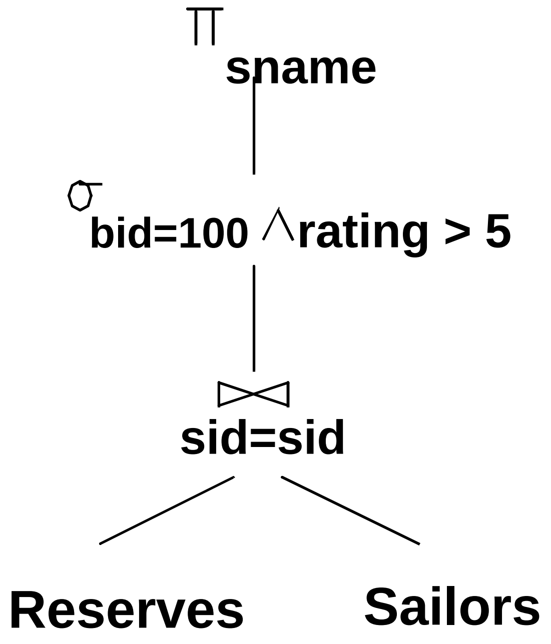
$$(Q \bowtie S) \cup (R \bowtie S) \equiv (Q \cup R) \bowtie S$$

Original Query Tree
(without any algorithms selected)

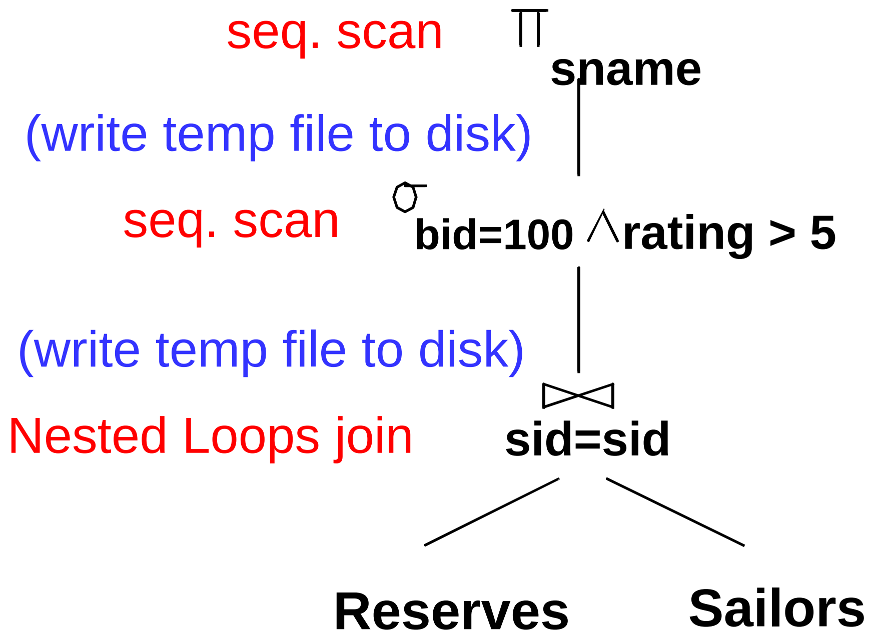
SQL Query:

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

RA Tree:



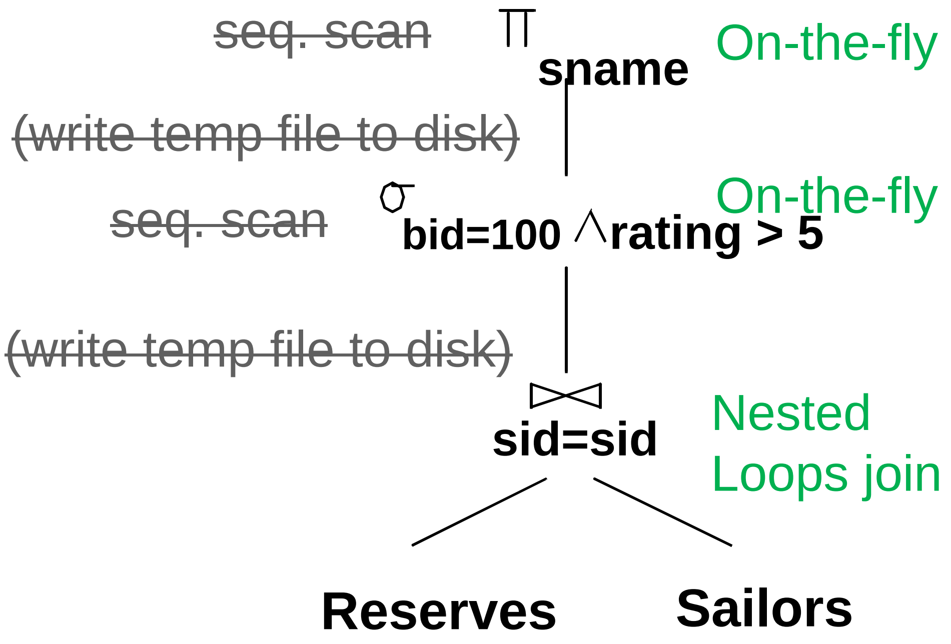
A Plan for the Original Query Tree: How shall we run the query plan?


seq. scan
π_{sname}
(write temp file to disk)
seq. scan
$\sigma_{\text{bid=100} \wedge \text{rating > 5}}$
(write temp file to disk)
Nested Loops join
$\bowtie_{\text{sid=sid}}$
Reserves
Sailors

One way to execute this query is to perform each operator (starting from the bottom) and always write the intermediate results out to disk.

We could choose a join algorithm and then do everything else with a sequential table scan.

But wait … select and project operators operate on just one row at a time


~~seq. scan~~
π_{sname}
On-the-fly
~~(write temp file to disk)~~
~~seq. scan~~
$\sigma_{\text{bid=100} \wedge \text{rating > 5}}$
On-the-fly
~~(write temp file to disk)~~
$\bowtie_{\text{sid=sid}}$
Nested Loops join
Reserves
Sailors

We can do a select (to filter out unwanted rows) and we can do a project (to drop columns) while the row is in memory.

Some previous operator must first read the rows into memory; then … we can do select and project "on the fly."

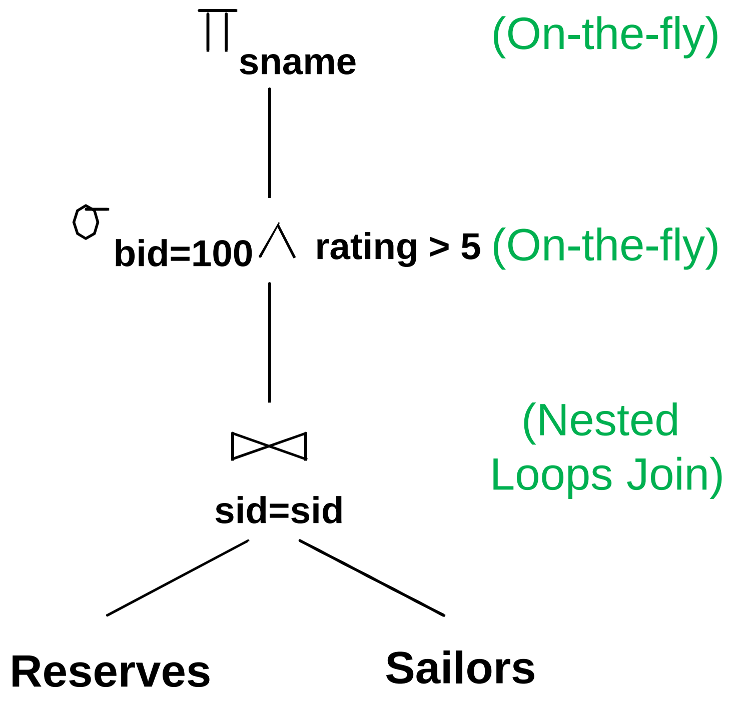
“On the fly” is “free”

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

On the fly means that we evaluate the operator in memory - while we have the tuple available.

“On the fly” induces no I/O cost!

Plan:



Limitations of “On the fly”

- Can only happen if:
  - Computation can be done entirely on tuples in memory
  - Results do not need to be materialized
  - An earlier operator has read the table
- Cannot be used for first table scan!
- Cannot apply to all operations such as join or aggregates, etc.

Cost of plan 1
no index (Sailors – inner loop)

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

M = # of pages in outer table

N = # of pages in inner table

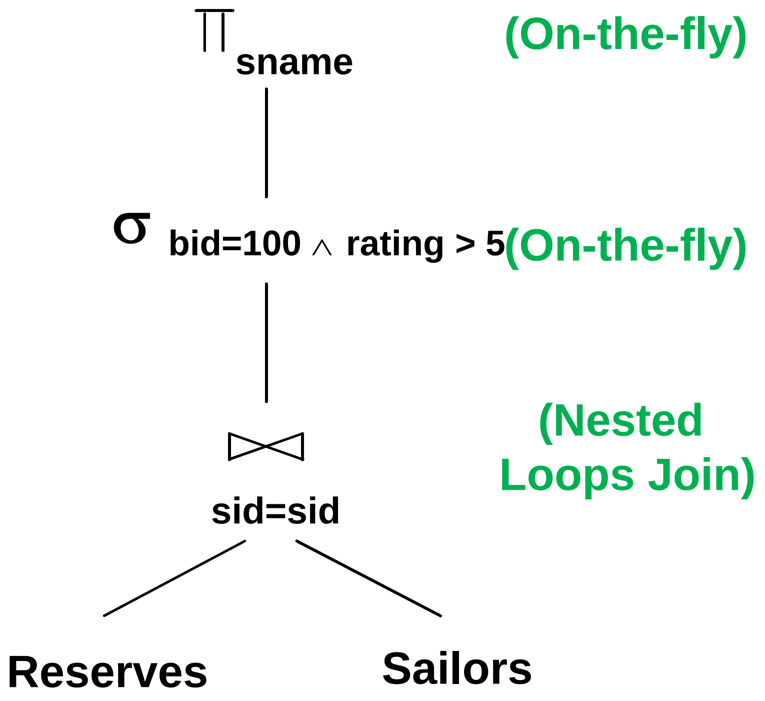
Cost of page-oriented nested loops join is:

M + M * N

1000 + 1000 * 500 = 501,000

And the "on-the-fly" operations have no I/O - so plan cost is 501,000

Plan:



Create other plans

- Use relational algebra equivalences to produce new query trees.
  - Advantage: you are sure that the new tree is equivalent to the original, because equivalences have proofs
- Assign different algorithms to operators

Cost of plan 2
(Reserves on inner loop)

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

Sailors as the outer relation rather than Reserves.

M = # of pages in outer table

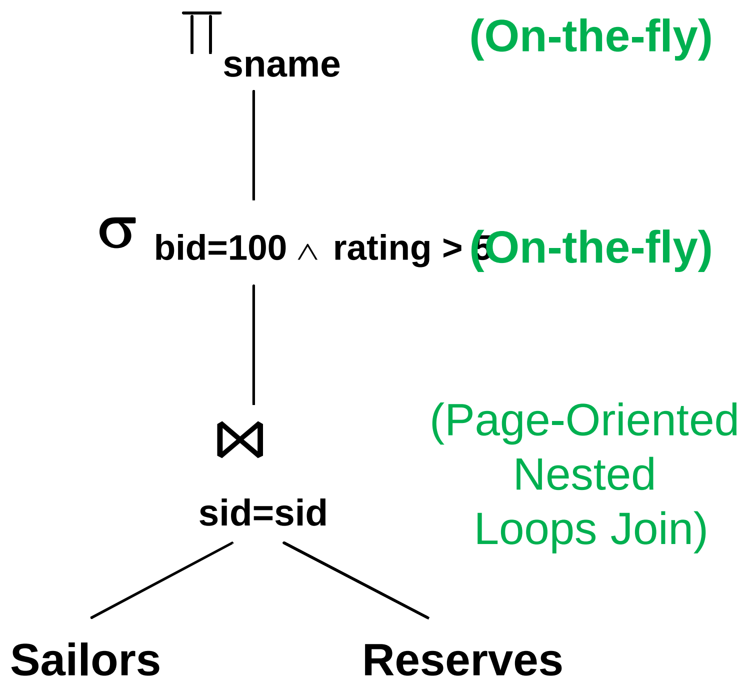
N = # of pages in inner table

Cost of page-oriented nested loops join is:

500 + 500 * 1000 = 500,500

And the “on-the-fly” operations have no I/O - so plan cost is 500,500

Plan:



Cost of plan 3
Index nested loops

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

What is the cost of the plan shown?

M + M * pr * (# of index I/Os + 1)

1000 + (1000*100*(3+1))

or

1000 + (1000 * 100 * (0 + 1))

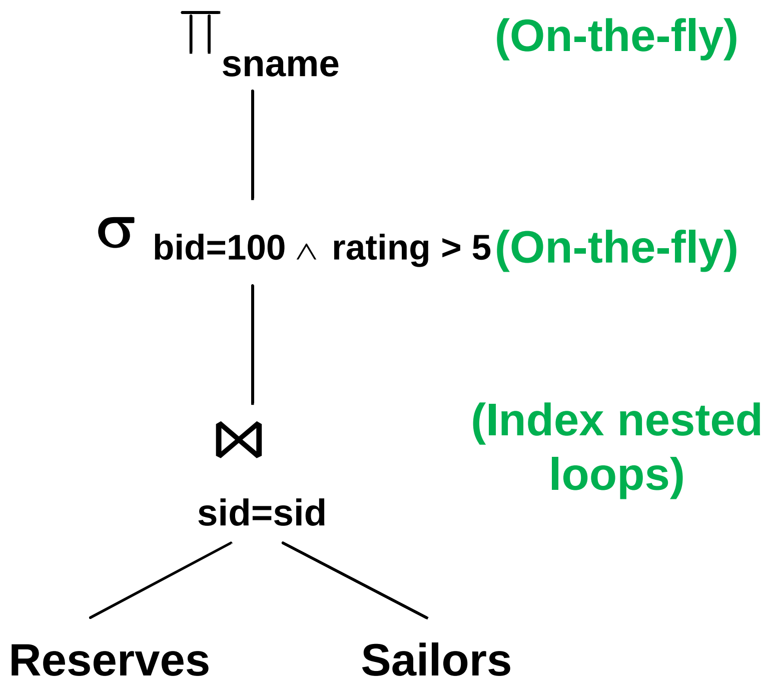
Thus …

1000 + 300,000 = 301,000

or

1000 + 100,000 = 101,000

Plan:



Cost of plan 4

Push down selects

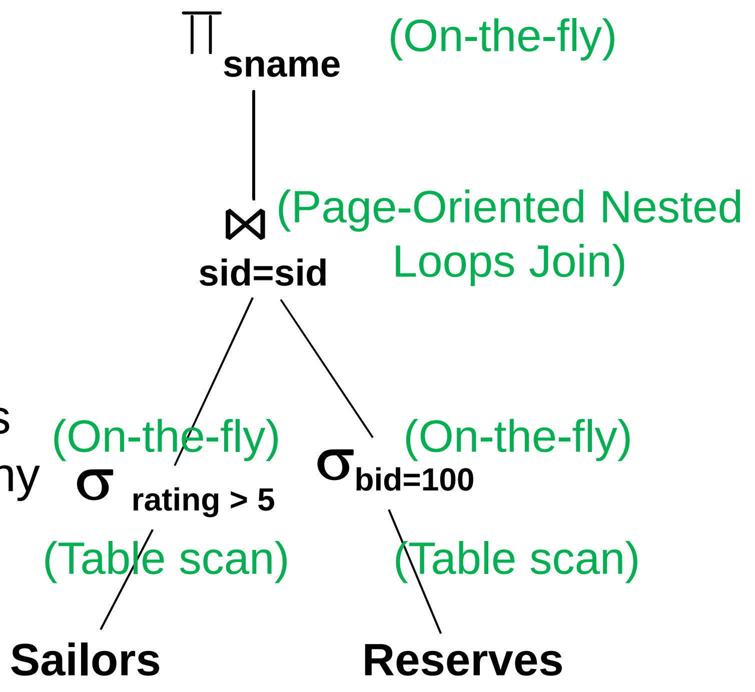
Apply this equivalence:

$$\sigma_c(R \bowtie S) \equiv \sigma_c(R) \bowtie S$$

To the previous query tree to get an equivalent query tree.

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

Plan:



What is the cost of the plan shown?

- Scanning sailors and reserves cost M+N I/Os. But here we read intermediate files.
- What about the cost of the join? It depends on how many reservations there are for boat 100 and how many sailors have a rating >5.
- How would you find this information?
  - Statistics will help.

To estimate cost, we need table sizes

- For all operators beyond the leaf level of the query plan, the input tables are the result of some earlier query.
- Thus, we need to estimate the size of intermediate results!
  (This can be difficult. This is one reason why the cost estimates may not be very good. Estimation errors tend to compound.)
- For example, what information would you need to estimate how many reservations are for bid 100 and how many sailors have a rating >5?

DBMS Usually Maintains Some Statistics in the DB Catalog

- *Catalogs* typically contain at least:
  - # tuples and # pages for each table.
  - # distinct key values and # pages for each index.
  - Index height, low/high key values for each tree index.

- Catalogs are updated periodically - say, once a week or once a month. Perhaps they're updated during the backup.

- Simplest case: assume that all attribute values are uniformly distributed. Thus if gender was an attribute, the optimizer would assume that half of the rows have the male value and other half have the female value. (This might be grossly inaccurate.)

Calculating Selectivities

- Assume that rating values range from 1 to 10, and that bid values range from 1 to 100.
- What percentage of the incoming tuples, to the operator $\sigma_{\mathbf{bid=100}}$, will be output?
- What about $\sigma_{\mathbf{rating > 5}}$ **?**
- $\sigma_{\mathbf{bid=100} \wedge \mathbf{rating > 5}}$ **?**

Doing better than a uniform distribution

- The DBMS might gather more detailed information about how the values of attributes are distributed (e.g., histograms of the values in a field) and store it in the catalog.

  Suppose there was an attribute degree-program with three possible values: “BS CS” “MS CS” “PhD CS”
  - Then the DBMS might count the values and know that there are 428 “BS CS” values, 98 “MS CS” values and 25 “PhD CS” values.
  - This allows much better estimate of the reduction factor.

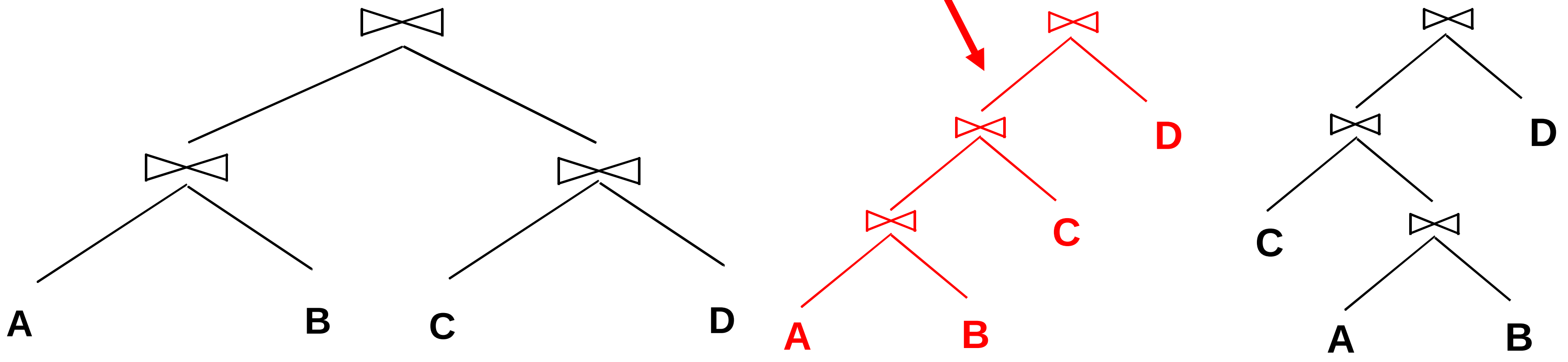
Independence of Reduction Factors

- So far, in our quest to compute costs of plans, we have gathered statistics and shown how to use them, assuming uniform distributions. But what about a select operator with terms separated by AND?
  - Example: $\sigma_{\text{bid=100} \wedge \text{rating > 5}}$
- We assume that all terms are independent!
- Thus, if one attribute is class and the other is number-of-hours - the query optimizer might assume that class is uniformly distributed over {Fresh, Soph, Jun, Sen} and that number-of-hours is uniformly distributed over {0, 1, …, 205}
  - But, we know that class correlates with number-of-hours! Might even be that number-of-hours $\rightarrow$ class.
- What percentage of the incoming tuples, to the operator $\sigma_{\text{bid=100} \wedge \text{rating > 5}}$, will be output?

Enumerating Plans for Multiple Joins

- Back to the problem of generating plans.
- Are we trying to generate as many plans as possible?
  - No, best to generate few, as long as cheap plans are among them.
- In System R: only left-deep join trees are considered.

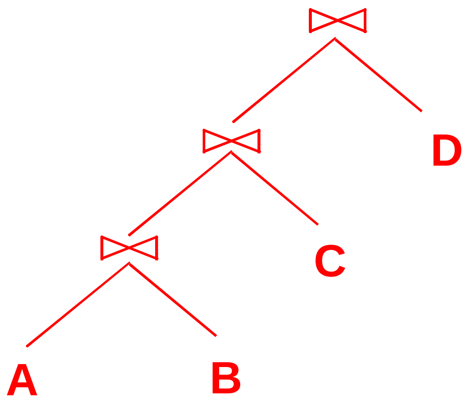
This one is left-deep - the other two are not.



Queries Over Multiple Relations (Joins)

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).



- Using only left-deep plans (obviously) restricts the search space. (So optimizer may not find the optimal plan.)

Enumeration of Left-Deep Plans

- Need to consider all possible left-deep plans.
  - For SPJ queries, these are enumerated by orderings of the tables
- For each ordering, consider the access method for each relation and the join method for each join.
- Enumerated using N passes (if N relations joined):
  - Pass 1: Find best 1-relation plan for each relation.
  - Pass 2: Find best way to join result of each 1-relation plan (as outer) to another relation. (All 2-relation plans.)
  - Pass N: Find best way to join result of a (N-1)-relation plan (as outer) to the N'th relation. (All N-relation plans.)
- At the end of each pass, retain only:
  - Cheapest plan overall, plus
  - Cheapest plan for each interesting order of the tuples.
    - Interesting order: corresponding to a join, ORDER BY or GROUP BY

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' the 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. The optimizer might not find it from the nested version, so you may need to explicitly unnest the query.

```
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  S.sid= outer value
```

Equivalent non-nested query:

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

Summary – Algorithms for Relational Algebra Operators

- A virtue of relational DBMSs: *queries are composed of a few basic operators*; the implementation of these operators can be carefully tuned (and it is important to do this!).
- Many alternative implementation techniques for each operator; no universally superior technique for most operators.
- Must consider available alternatives for each operation in a query and choose best one based on system statistics, etc. This is part of the broader task of optimizing a query composed of several ops.

Query Optimizers Don’t (Always) Find the Best Plan

- There are usually more plans than you can consider, even if only left deep plans are considered.
- The optimizer might not even try to generate all possible plans (it won’t be able to consider all of them anyway).
- Sometimes the optimizer will compare the optimization cost to the estimated execution cost and quit early.
- The optimizer chooses the plan with the lowest ESTIMATED cost. Actual costs may differ.