Query Execution

Werner Nutt

Introduction to Databases

Free University of Bozen-Bolzano

Example Database

Our example queries will be based on the relations `Sailors` and `Reserves`

- `Sailors`:
  Each tuple 50 bytes long
  80 tuples per page
  500 pages
- `Reserves`:
  Each tuple 40 bytes long
  100 tuples per page
  1000 pages

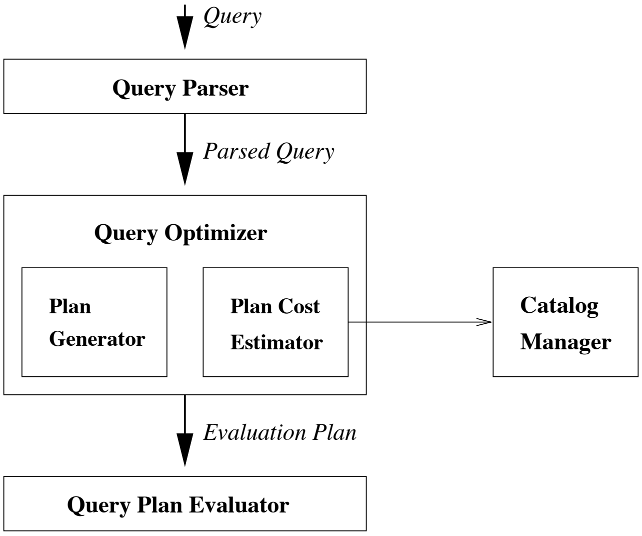
$S =$

`sid`	`sname`	`rating`	`age`
22	Dustin	7	45.0
31	Lubber	8	55.5
58	Rusty	10	35.0

$R =$

`sid`	`bid`	`day`	`rcode`
22	101	10/10/96	Hoho
58	103	11/12/96	007

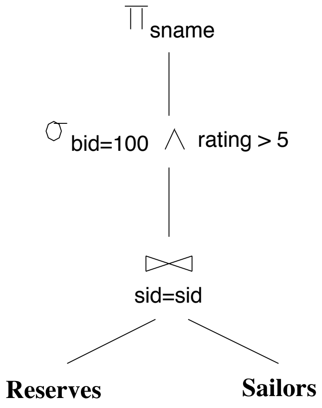
Query Processor: Architecture



Queries are parsed, optimized, evaluated

Query Parser

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



Parser creates relational algebra expression of the form

$$\pi_{\mathit{Attributes}}(\sigma_{\mathit{Conditions}}(R_1 \bowtie \cdots \bowtie R_n))$$

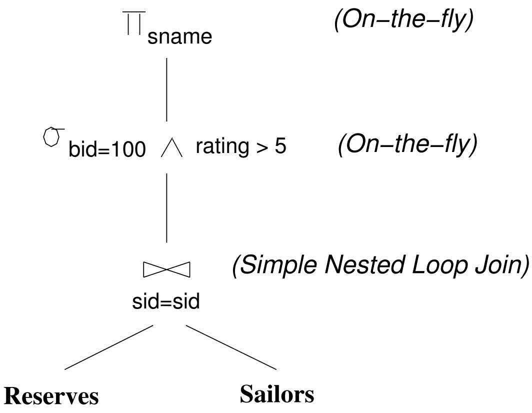
i.e., first join, then select, then project

Plan Generator

The *Plan Generator*

- generates a set of *equivalent* algebra expressions
- *annotates* the operators with *procedures* to compute them.

Example:



π_{sname} *(On–the–fly)*

$\sigma_{\text{bid=100} \wedge \text{rating > 5}}$ *(On–the–fly)*

$\bowtie_{\text{sid=sid}}$ *(Simple Nested Loop Join)*

Reserves **Sailors**

The Cost of Plans

The optimizer

- estimates for each generated plan the cost,
- then chooses the cheapest plan

Important: Avoid the worst plans!

We will study

1. first, implementations of operators,
2. then, plans that combine operator implementations.

Relational Operators

We will consider how to implement:

- **Selection** “σ”**:** selects a subset of rows from relation
- **Projection** “π”**:** deletes unwanted columns from relation
- **Join** “$\bowtie$”**:** allows us to combine two relations

Each operator returns a relation $\leadsto$ operators can be *composed!*

First cover operator, *then* discuss how to optimize queries
formed by composing them.

What is the Cost of an Operator Implementation?

Two parameters:

- **Time:** *How many I/O operations are needed?* Depends on
  - *#pages* of input relations
  - *#records* per page
  - existence of *index* etc.
- **Result Size:** *What is the size of the result?* Factors as above plus
  - *selectivity* of conditions in a selection or join
  - size of attributes *projected out*

  Usually expressed as a *“reduction factor”*

Both are combined to estimate *overall cost* of an evaluation plan

Simple Selections

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

General form $\sigma_{R.A \textit{ op Val}}(R)$

Assumption: M pages of R, p_R tuples per page

- Size of result approximated as: *(size of R) × (reduction factor)*
- **No index**, **unsorted**: Relation scan $\leadsto$ cost is M ($=$ #pages in R)
- **With index** on selection attribute:
  Use index to find qualifying *data entries,*
  then retrieve corresponding *data records.*
  (Hash index useful only for equality selections.)

Using an Index for Selections

Cost depends on **#qualifying tuples,** and **clustering**:

Cost of *finding* qualifying data *entries* *(typically small)*

$+$

Cost of *retrieving records* *(could be large w/o clustering)*

Example: Uniform distribution of code names

$\leadsto$ $\approx$ 10% of tuples qualify (100 pages, 10,000 tuples)

clustered index $\leadsto$ cost $\approx$ $\boxed{\text{100 IO's}}$

unclustered index $\leadsto$ cost up to $\boxed{\text{10,000 IO's}}$!

Using an Index for Selections: Refinement

Important refinement for **unclustered** *indexes:*

1. **Find** qualifying data *entries*
2. **Sort** the *rid's* of the data records to be retrieved
3. **Fetch** *rid's* in order.

This ensures that each data page is looked at just once

(though # of such pages likely to be higher than with clustering).

More General Selection Conditions

```
(day<8/9/94 AND rcode='Hiho') OR bid=5 OR sid=3
```

- Each disjunct (i.e, part connected by `OR`) is processed separately,

  ... then the union is taken of the results.

- An *index* **matches** (a conjunction of) *conditions*

  if they involve only attributes in a **prefix** of the search key, and
  if all, but possibly the last, are involved in equality conditions

  - E.g., *index* on $\langle \mathtt{a}, \mathtt{b}, \mathtt{c} \rangle$

    matches `a=5 AND b=3`

    but not `b=3`

First Approach to General Selection

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/O's.
- Conditions 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.

Example: `day<8/9/94 AND bid=5 AND sid=3`

- *First* B+-tree index on `day`, *then* check `bid=5` and `sid=3`, or
- *First* hash-based index on $\langle \texttt{bid}, \texttt{sid} \rangle$ *then* check `day<8/9/94`

Second Approach: Intersection of Rid's

Applicable if we have 2 or more matching indexes that use Alternatives (2) or (3) for data entries

Using each matching index, **get sets of rid's**
Intersect these sets of rid's (⇝ *How?)*
Retrieve the records and apply any remaining terms

Example: `day < 8/9/94` `AND bid=5 AND` `sid = 3`

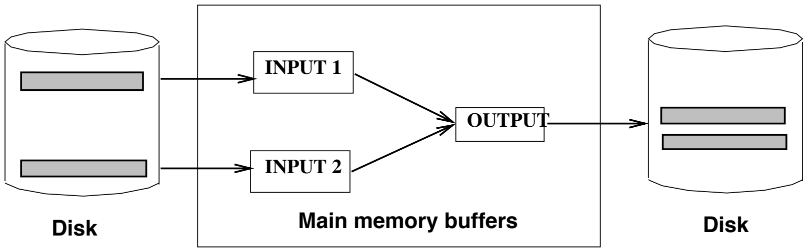
Assumption: *B+-tree* index on `day` and *hash-based index* on `sid` *(both using Alternative (2))*

- using the B+-tree, get rid's of records satisfying `day<8/9/94`
- using the hash-based index, get rid's satisfying `sid=3`
- intersect, retrieve records and check `bid=5`

A Useful Technique: External Sorting

Example: *2-Way External Sorting with 3 Buffers*

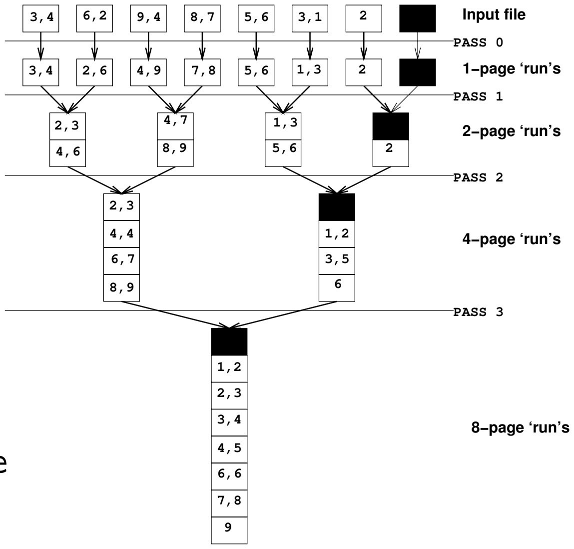
- Pass 0: Read a page, sort it, write it
  - only one buffer page is used
- Pass 1, 2, 3, . . . , etc.
  - three buffer pages used.



Generalisations use more buffers

2-Way External Sorting: Example

- Each pass we read and write each page in file
- M pages in the file $\Rightarrow$ number of passes $\approx \log_2 M$
- Total cost is $\approx M \times \log_2 M$
- *Idea:* **Divide and conquer** i.e., sort subfiles and merge

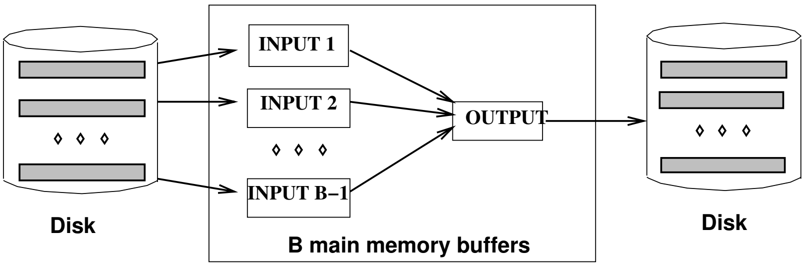


General External Merge Sort

More than 3 buffer pages. How can we utilize them?

To sort a file with N **pages** using B **buffer pages**:

- *Pass 0:* use B buffer pages;
  produce $\lceil N/B \rceil$ sorted runs of B pages each
- *Pass 1, 2,. . . , etc.:* merge $B - 1$ runs



Cost of External Merge Sort

- Number of passes: $1 + \left\lceil \log_{B-1} \lceil N/B \rceil \right\rceil$
- Cost $= 2N \times$ #passes

Example: Sort 108 page file with 5 buffer pages

Pass 0: $\lceil 108/5 \rceil = 22$ sorted runs of 5 pages each
(last run is only 3 pages)

Pass 1: $\lceil 22/4 \rceil = 6$ sorted runs of 20 pages each
(last run is only 8 pages)

Pass 2: 2 sorted runs, 80 pages and 28 pages

Pass 3: Sorted file of 108 pages

Number of Passes of External Merge Sort

N	$B = 3$	$B = 5$	$B = 9$	$B = 17$	$B = 129$	$B = 257$
100	7	4	3	2	1	1
1,000	10	5	4	3	2	2
10,000	13	7	5	4	2	2
100,000	17	9	6	5	3	3
1,000,000	20	10	7	5	3	3
10,000,000	23	12	8	6	4	3
100,000,000	26	14	9	7	4	4
1,000,000,000	30	15	10	8	5	4

Sorting: Summary

- External sorting is important:

  DBMS may dedicate part of **buffer pool** for sorting!

- **External merge sort** minimizes disk I/O cost:
  - *Pass 0:* produces sorted **runs** of size B (= #buffer pages).
    Later passes: **merge** runs.
  - #runs merged at a time depends on B
  - In practice, #passes rarely more than 2 or 3

The Projection Operator

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

Approach based on *sorting*

- *Modify Pass 0* of external sort to **eliminate unwanted fields**

  $\leadsto$ tuples in runs are smaller than input tuples

- *Modify merging passes* to **eliminate duplicates**

  $\leadsto$ number of result tuples smaller than input

- Cost
  - Pass 0: read original relation (size M pages),
    write out same number of smaller tuples
  - In merging passes: fewer tuples written out in each pass

Discussion of Projection

- Sort-based approach is the standard

  . . . *but there are also hash-based techniques*

- If an index contains all wanted *attributes* in its *search key*, do an **index-only scan**.
  - Apply projection techniques to data entries (much smaller!)

- If a *tree-based* (i.e., ordered) *index* contains all wanted attributes as **prefix** of search key, do even better:
  - **Retrieve** data entries in order (index-only scan),
  - **Discard** unwanted fields,
  - **Compare** adjacent tuples to check for duplicates.

Equality Joins With One Join Column

```
SELECT *
FROM   Reserves R, Sailors S
WHERE  R.sid = S.sid
```

- In algebra: $R \bowtie S$. Common! Must be carefully optimized
  $R \times S$ is large $\rightsquigarrow$ $R \times S$ followed by selection is inefficient
- Assume: M pages of R, p_R tuples per page,
  N pages of S, p_S tuples per page.
- In examples, R is `Reserves` and S is `Sailors`
- Cost metric: # of I/O's

Simple Nested Loops Join

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

- For each tuple in the *outer* relation R
  we scan the entire *inner* relation S
  - **Cost:** $M + p_R \times M \times N = 1000 + 100 \times 1000 \times 500$ I/O's.

Page-oriented Nested Loops join:

- For each page of R, get each page of S,
  and write out matching pairs of tuples $\langle r, s \rangle$
  where r is in R-page and s is in S-page.
  - **Cost:** $M + M \times N = 1000 + 1000 \times 500$ I/O's.

Index Nested Loops Join: Idea

$$R \bowtie_{R.i\,=\,S.j} S \quad !$$

Suppose, there is an **index** on **attribute** j **of** S

$\rightsquigarrow$ make S **inner relation** of nested loops join

$\rightsquigarrow$ exploit index!

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

Cost of Index Nested Loops Join

Overall cost is

$$\boxed{M + (M \times p_R \times \text{cost of finding matching tuples in } S)}$$

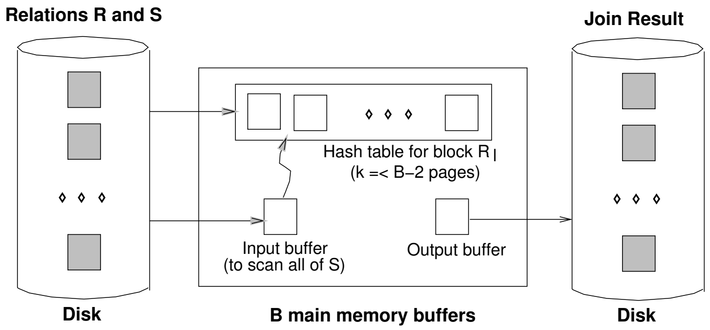
What is the *"cost of finding matching tuples in S"?*

- For each tuple in R, **probe** into S-index
  - hash index: $\boxed{\approx 1.2 \text{ I/O}}$
  - B+-tree: $\boxed{\text{2–4 I/O's}}$
- Then, **retrieve** all matching S-tuples
  - clustered index: $\boxed{\text{1 I/O}}$ typically
  - unclustered: up to $\boxed{\text{1 I/O per tuple}}$

Block Nested Loops Join

Why keep only one page of R in buffer? Better:

- one page as **input buffer** for scanning the inner S
- one page as the **output buffer**
- all remaining pages hold **block** of outer R



For each matching tuple r in R-block, s in S-page,
add $\langle r, s \rangle$ to result.

Then read next R-block, scan S, etc.

Sort-Merge Join (1)

$$R \bowtie_{R.i\,=\,S.j} S \quad !$$

Idea: **Sort** R on $R.i$ and S on $S.j$
then **scan** R and S to do a "merge" on join colums
... and **output** result tuples

After sorting, how do we find the next pair of matching tuples?

```
while ($R.i \neq S.j$)
    {while ($R.i < S.j$)
        advance scan of $R$;
     while ($R.i > S.j$)
        advance scan of $S$;}
```

Under which assumption is this code correct?

Sort-Merge Join (2)

At this point: $(R.i = S.j)$

From here on,

- all R tuples with the same value in $R.i$ *(the current R group)*
- and all S tuples with same value in $S.j$ *(the current S group)*

match!

$\rightsquigarrow$ **output** $\langle r, s\rangle$ for all pairs of such tuples!

Then resume scanning R and S

Total cost: *sorting*(R) + *sorting*$(S) + M + N$

Hash Join: Principles

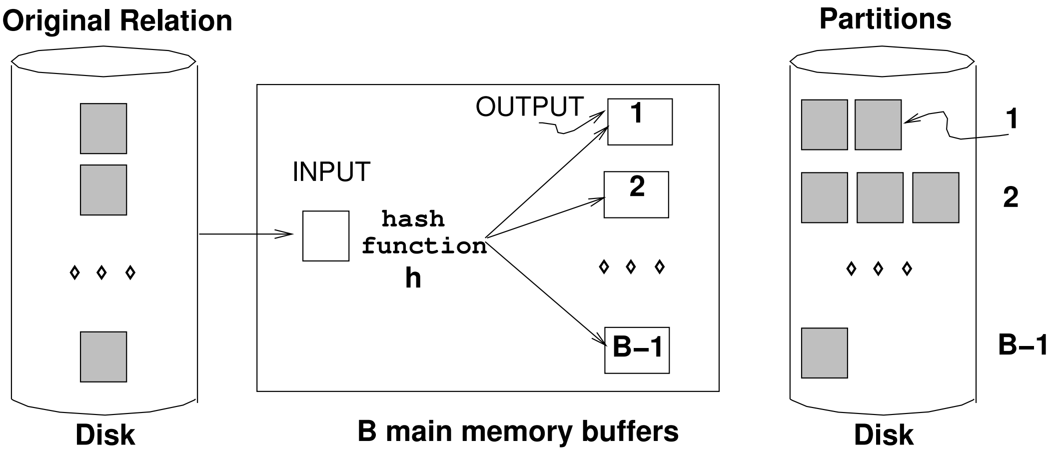
Two phases

- **Partitioning** (or "building"): Each of R and S are divided into *partitions* $R_1, \ldots, R_k$ and $S_1, \ldots, S_k$, using a *hash function* h
- **Probing** (or "matching"): Tuples in R_i and S_i are matched using a *different* hash function h_2

Hash Join: Partitioning

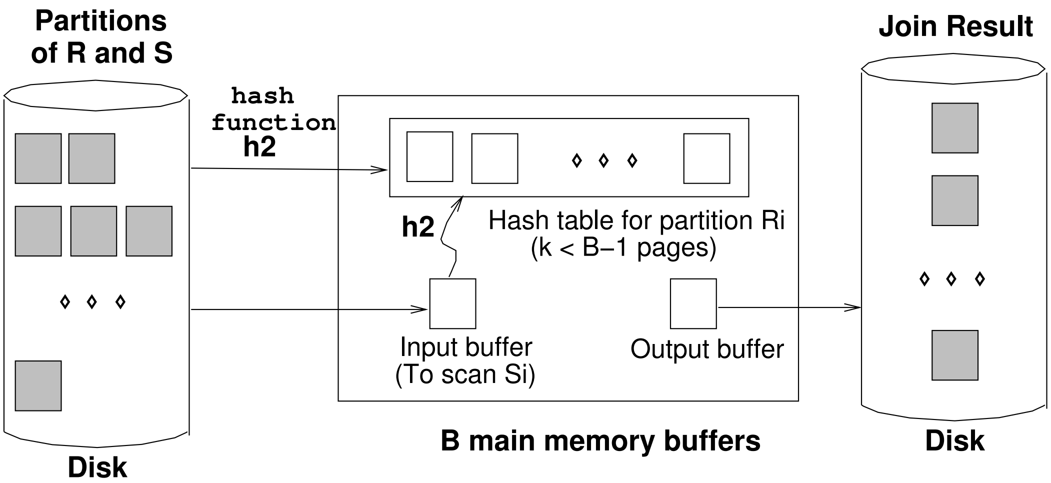
Partition R and S using a hash function h

$\Rightarrow$ R tuples in partition i will only match S tuples in partition i



Hash Join: Probing

- Read in a partition of R, hash it using h_2 ($\neq h$!)
- Scan matching partition of S, search for matches



How Much Buffer Space Do We Need?

Constraints:

- k (= # partitions) $\leq B-1$
- size of largest partition to be held in memory $\leq B-2$

Assumption: all partitions have equal size. Then:

- $k = B-1$ and $M/(B-1) \leq B-2 \quad \Rightarrow \quad \boxed{B \geq \sqrt{M}}$

Optimisation: Use an in-memory hash to compute matching tuples

⇒ more memory is needed

Possible Problem: The hash function does not partition uniformly

⇒ one or more R partitions may not fit into memory

Solution: Apply hash-join technique recursively
to join this R-partition with corresponding S-partition

Hash Join: Analysis

Cost:

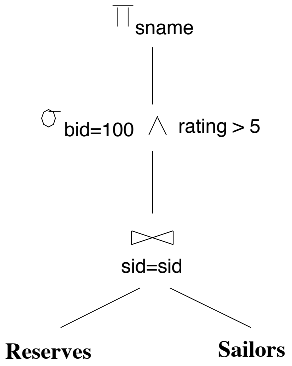
- *Partitioning phase:* read and write both R and $S \Rightarrow 2(M+N)$ I/Os
- *Probing phase:* read both R and $S \Rightarrow M+N$ I/Os
- In the running example: 4500 I/Os in total

Sort-Merge Join vs. Hash Join

- Both have cost of $3(M+N)$ I/Os if sufficient(?) memory is available
- Hash Join is superior if relation sizes differ greatly
  (proof needs some assumptions about internal sorting method)
- Hash Join can be parallelized
- Sort-Merge is less sensitive to data skew

Query Optimization: Example

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

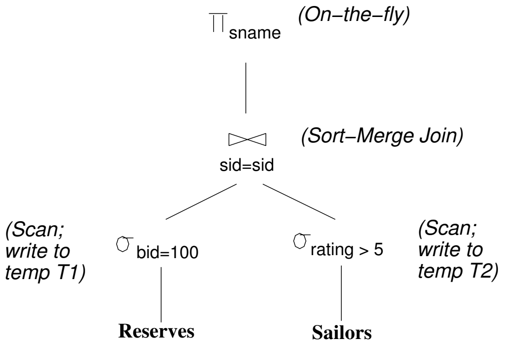


- Cost of this plan: $\boxed{500 + 500 \times 1000 \text{ I/O's}}$
- Missed opportunities:
  - selections have not been *“pushed”*
  - no *indexes* are used

Alternative Plans: No Indexes

Main difference:

selections pushed down



Cost of plan

(with 5 buffers):

scan `Reserves` ($1,000$ pages)

\+ write temporary `T1` (10 pages, if #boats $= 100$ and uniform distribution)

scan `Sailors` (500 pages)

\+ write temporary `T2` (250 pages, if #ratings $= 10$)

sort `T1` ($2 \times 2 \times 10$ I/O's) + sort `T2` ($2 \times 4 \times 250$ I/O's)

\+ merge `T1` and `T2` ($10 + 250$ I/O's)

$4,060$ page I/O's

Alternative Plan with Indexes

With **clustered index** on `bid` of `Reserves`:

$100,000/100 = 1,000$ tuples on $1,000/100 =$ $\boxed{\text{10 pages}}$

Index nested loops join with "pipelining"

(i.e., outer relation is not materialized

$\leadsto$ projection doesn't help)

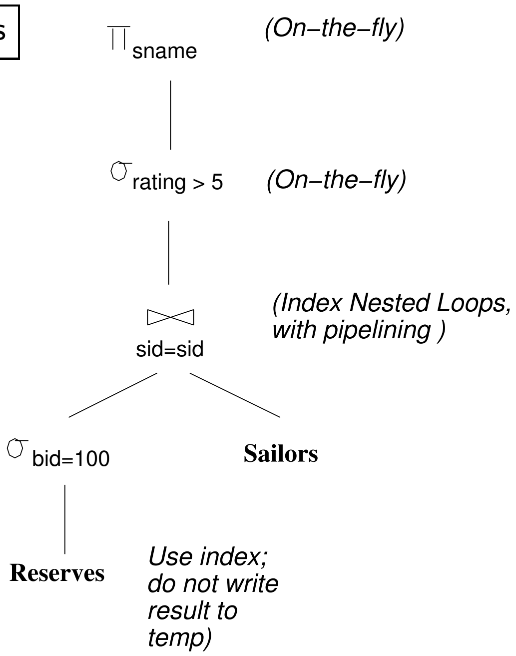
Join attribute `sid` is a **key** for `Sailors`

at most one tuple in `Sailors` matches

$\leadsto$ clustering wouldn't help)

Selection $\sigma_{\texttt{rating}>5}$ is **not pushed**

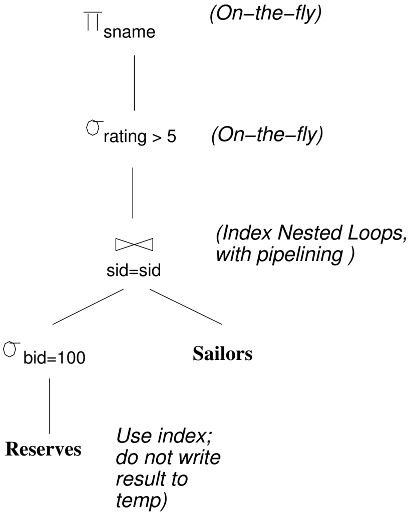
because join is based on index for `sid`



Alternative Plan with Indexes (Cntd.)

Cost:

- Selection of `Reserves` tuples: $\boxed{\text{10 I/O's}}$
- For each, retrieve matching tuples from `Sailors`: $\boxed{1,000 \times 1.2 \text{ I/O's}}$
- Total: $\boxed{1,210 \text{ I/O's}}$



Summary

- Query optimization (QO) is an important task in a relational DBMS
- Understanding of QO is necessary to understand the impact
  ⇝ of a given **database design** (relations, indexes)
  ⇝ on the **workload** (= set of queries)
- QO has two parts:
  - Enumeration of **alternative plans**
    ⇝ pruning of search space: left-deep plans only
  - Estimation of **cost** of enumerated plans
    ⇝ **size** of results
    ⇝ **cost** of each plan node

Key issues: Statistics, indexes, operator implementations

References

These slides are based on Chapters 12, 13, 14, and 15 of the book *Database Management Systems* by R. Ramakrishnan and J. Gehrke, and on slides by the authors published at

`www.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html`