Database Systems
(資料庫系統)

December 19/20, 2006

Lecture #11

Announcement

- Assignment #4 is due tomorrow.
- Assignment #5 will be out on the course homepage next monday.
  - It is two weeks from tomorrow.
- Forum on "Technology, Human, and Life"on 1/6
  - http://www.nightmarket.org/2007/forum0106.htm

External Sorting

Chapter 13

Why learn sorting again?

- O (n*n): bubble, insertion, selection, … sorts
- O (n log n): heap, merge, quick, … sorts
- Sorting huge dataset (say 10 GB)
- CPU time complexity may mean little on practical systems
- Why?

"External" Sorting Defined

- Refer to sorting methods when the data is too large to fit in main memory.
  - E.g., sort 10 GB of data in 100 MB of main memory.
- During sorting, some intermediate steps may require data to be stored externally on disk.
- Disk I/O cost is much greater than CPU instruction cost
  - Average disk page I/O cost: 10 ms vs. 4 GHz CPU clock: 0.25 ns.
  - Minimize the disk I/Os (rather than number of comparisons).

5

Outline (easy chapter)

- Why does a DMBS sort data?
- Simple 2-way merge sort
- Generalize B-way merge sort
- Optimization
  - Replacement sort
  - Blocked I/O optimization
  - Double buffering
- Using an existing B+ tree index vs. external sorting

6

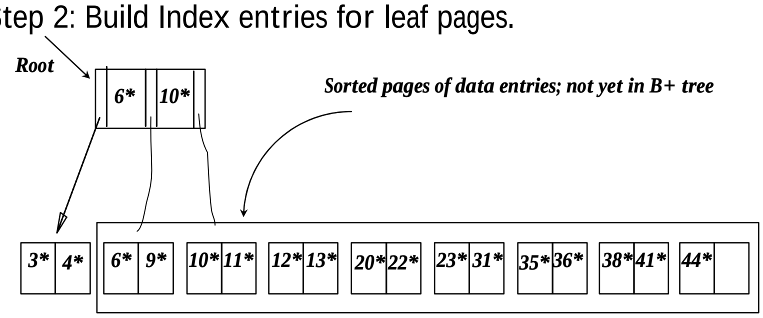
When does a DBMS sort data?

- Users may want answers to query in some order
  - E.g., students sorted by increasing age
- Sorting is the first step in bulk loading a B+ tree index
- Sorting is used for eliminating duplicate copies
- Join requires a sorting step.
  - Sort-join algorithm requires sorting.

7

Bulk Loading of a B+ Tree

- Step 1: Sort data entries. Insert pointer to first (leaf) page in a new (root) page.
- Step 2: Build Index entries for leaf pages.



8

Example of Sort-Merge Join

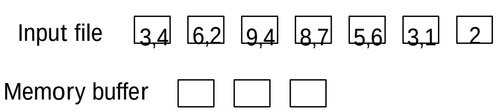
sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

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

9

A Simple Two-Way Merge Sort

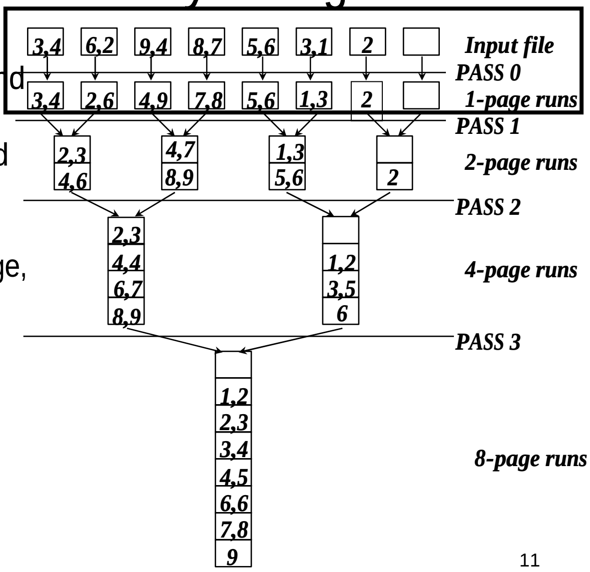
- External sorting: data >> memory size
- Say you only have 3 (memory) buffer pages.
  - You have 7 pages of data to sort.
  - Output is a sorted file of 7 pages.
- How would you do it?
- How would you do it if you have 4 buffer pages?
- How would you do it if you have n buffer pages?



10

A Simple Two-Way Merge Sort

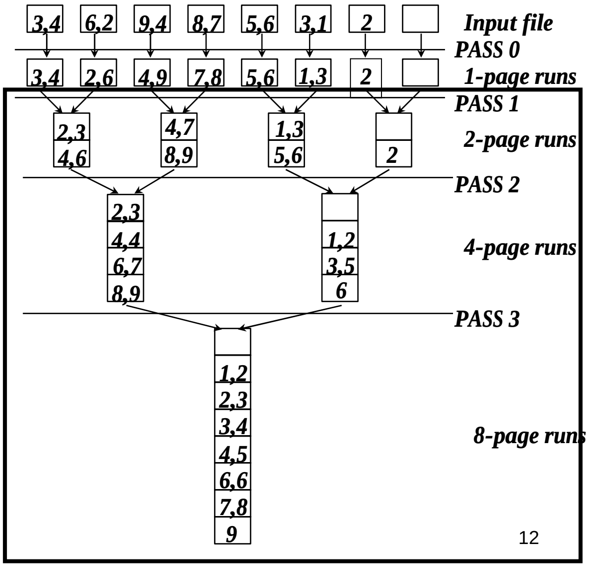
- Basic idea is divide and conquer.
- Sort smaller runs and merge them into bigger runs.
- Pass 0: read each page, sort records in each page, and write the page out to disk. (1 buffer page is used)



11

A Simple Two-Way Merge Sort

- Pass 1: read two pages, merge them, and write them out to disk. (3 buffer pages are used)
- Pass 2-3: repeat above step till one sorted 8-page run.
- Each run is defined as a sorted subfile.



12

2-Way Merge Sort

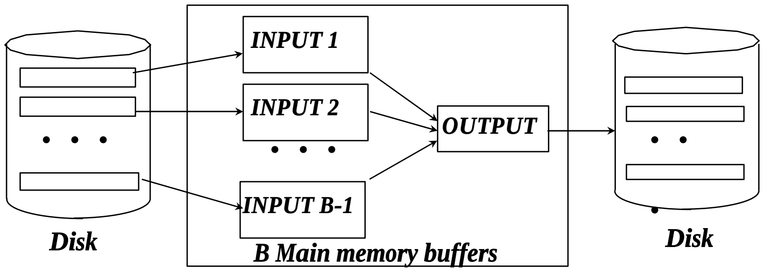
- Say the number of pages in a file is 2^k:
  - Pass 0 produces 2^k sorted runs of one page each
  - Pass 1 produces 2^{k-1} sorted runs of two pages each
  - Pass 2 produces 2^{k-2} sorted runs of four pages each
  - Pass k produces one sorted runs of 2^k pages.
- Each pass requires read + write each page in file: 2*N
- For a N pages file,
  - the number of passes = ceiling ($\log_2$ N) + 1
- So total cost (disk I/Os) is
  - 2*N*(ceiling($\log_2$ N) + 1)

13

General External Merge Sort

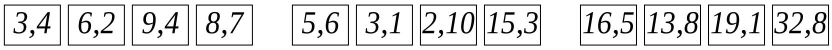
More than 3 buffer pages. How can we utilize them?

- To sort a file with N pages using B buffer pages:
  - Pass 0: use B buffer pages. Produce N / B sorted runs of B pages each.
  - Pass 1..k: use B-1 buffer pages to merge B-1 runs, and use 1 buffer page for output.

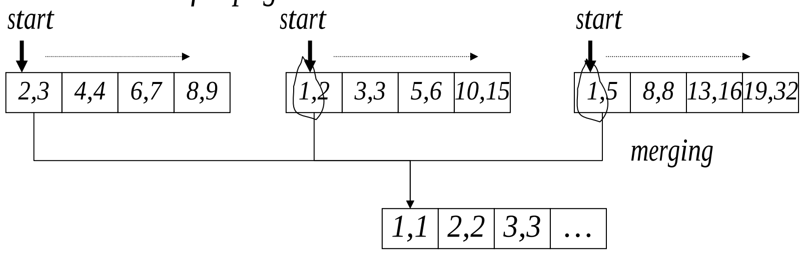


14

General External Merge Sort (B=4)



Pass 0: Read four unsorted pages, sort them, and write them out. Produce 3 runs of 4 pages each.



Pass 1: Read three pages, one page from each of 3 runs, merge them, and write them out. Produce 1 run of 12 pages.

15

Cost of External Merge Sort

- # of passes: 1 + ceiling(log $_{B-1}$ ceiling(N/B))
- Disk I/O Cost = 2*N*(# of passes)
- E.g., with 5 buffer pages, to sort 108 page file:
  - Pass 0: ceiling(108/5) = 22 sorted runs of length 5 pages each (last run is only 3 pages)
  - Pass 1: ceiling(22/4) = 6 sorted runs of length 20 pages each (last run is only 8 pages)
  - Pass 2: 2 sorted runs, of length 80 pages and 28 pages
  - Pass 3: Sorted file of 108 pages
  - # of passes = 1 + ceiling (log $_4$ ceiling(108/5)) = 4
  - Disk I/O costs = 2*108*4 = 864

16

Passes of External 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*

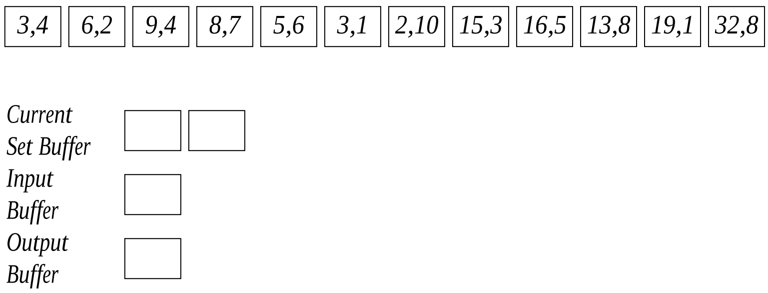
17

Further Optimization Possible?

- Opportunity #1: create bigger length run in pass 0
  - First sorted run has length B.
  - Is it possible to create bigger length in the first sorted runs?
- Opportunity #2: consider block I/Os
  - Block I/O: reading & writing consecutive blocks
  - Can the merge passes use block I/O?
- Opportunity #3: minimize CPU/disk idle time
  - How to keep both CPU & disks busy at the same time?

18

Opportunity 1: create bigger runs on the 1st pass



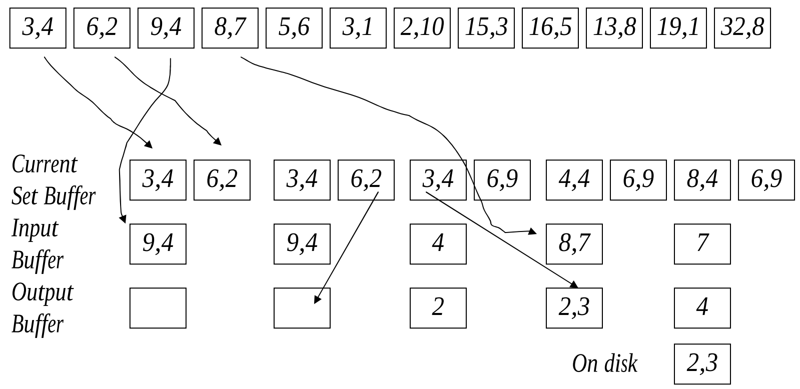
19

Replacement Sort (Optimize Merge Sort)

- Pass 0 can output approx. 2B sorted pages on average. How?
- Divide B buffer pages into 3 parts:
  – Current set buffer (B-2): unsorted or unmerged pages.
  – Input buffer (1 page): one unsorted page.
  – Output buffer (1 page): output sorted page.
- Algorithm:
  – Pick the tuple in the current set with the smallest k value > largest value in output buffer.
  – Append k to output buffer.
  – This creates a hole in current set, so move a tuple from input buffer to current set buffer.
  – When the input buffer is empty of tuples, read in a new unsorted page.

20

Replacement Sort Example (B=4)



When do you start a new run?

All tuple values in the current set < the last tuple value in output buffer

Minimizing I/O Cost vs. Number of I/Os

- So far, the cost metric is the number of disk I/Os.
- This is inaccurate for two reasons:
  - (1) Block I/O is a much cheaper (per I/O request) than equal number of individual I/O requests.
    - Block I/O: read/write several consecutive pages at the same time.
  - (2) CPU cost may be significant.
    - Keep CPU busy while we wait for disk I/Os.
    - Double Buffering Technique.

Further Optimization Possible?

- Opportunity #1: create bigger length run in the 1st round
  - First sorted run has length B.
  - Is it possible to create bigger length in the first sorted runs?
- Opportunity #2: consider block I/Os
  - Block I/O: reading & writing consecutive blocks is much faster than separate blocks
  - Can the merge passes use block I/O?
- Opportunity #3: minimize CPU/disk idle time
  - How to keep both CPU & disks busy at the same time?

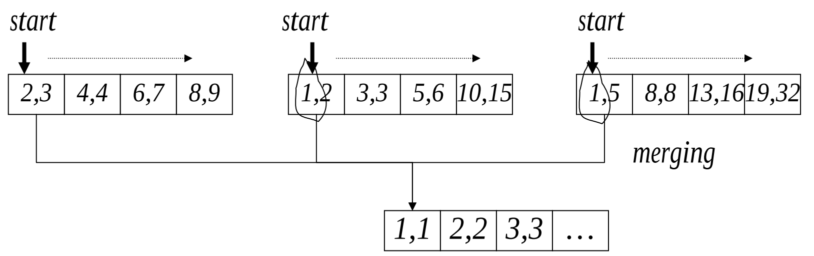
23

General External Merge Sort (B=4)

How to change it to Block I/O (block = 2 pages)?

3,4	6,2	9,4	8,7		5,6	3,1	2,10	15,3		16,5	13,8	19,1	32,8

Pass 0: Read four unsorted pages, sort them, and write them out. Produce 3 runs of 4 pages each.



Pass 1: Read three pages, one page from each of 3 runs, merge them, and write them out. Produce 1 run of 12 pages.

24

Block I/O

- Block access: read/write b pages as a unit.
- Assume the buffer pool has B pages, and file has N pages.
- Look at cost of external merge-sort (with replacement optimization) using Block I/O:
  - Block I/O has little affect on pass 0.
    - Pass 0 produces initial N' (= N/2B) runs of length 2B pages.
  - Pass 1..k, we can merge F = B/b – 1 runs.
  - The total number of passes (to create one run of N pages) is $1 + \log_F (N')$.

25

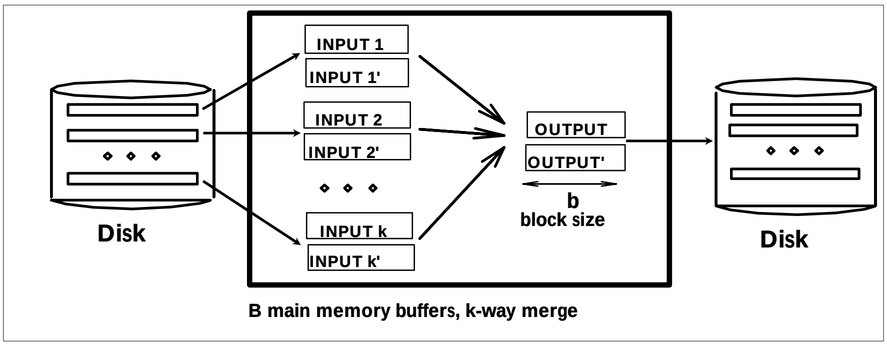
Further Optimization Possible?

- Opportunity #1: create bigger length run in the 1st round
  - First sorted run has length B.
  - Is it possible to create bigger length in the first sorted runs?
- Opportunity #2: consider block I/Os
  - Block I/O: reading & writing consecutive blocks
  - Can the merge passes use block I/O?
- Opportunity #3: minimize CPU/disk idle time
  - While CPU is busy sorting or merging, disk is idle.
  - How to keep both CPU & disks busy at the same time?

26

Double Buffering

- Keep CPU busy, minimizes waiting for I/O requests.
  - While the CPU is working on the current run, start to prefetch data for the next run (called shadow blocks).
- Potentially, more passes; in practice, most files still sorted in 2-3 passes.



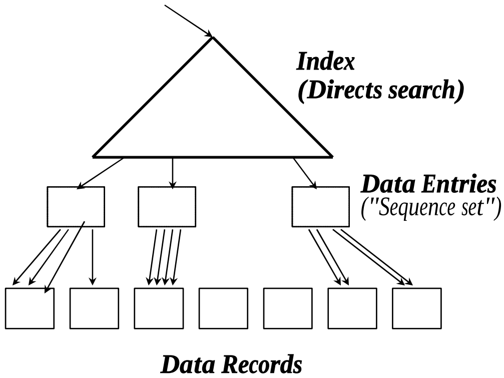
Using B+ Trees for Sorting

- Assumption: Table to be sorted has B+ tree index on sorting column(s).
- Idea: Can retrieve records in order by traversing leaf pages.
- Is this a good idea?
- Cases to consider:
  - B+ tree is clustered
  - B+ tree is not clustered

28

Clustered B+ Tree Used for Sorting

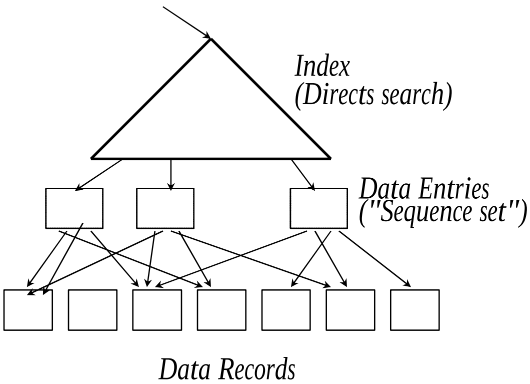
- Cost: root to the left-most leaf, then retrieve all leaf pages (Alternative 1)
- If Alternative 2 is used? Additional cost of retrieving data records: each page fetched just once.
- Cost better than external sorting?



29

Unclustered B+ Tree Used for Sorting

- Alternative (2) for data entries; each data entry contains rid of a data record. In general, one I/O per data record.



30

External Sorting vs. Unclustered Index

N	Sorting	p=1	p=10	p=100
100	200	100	1,000	10,000
1,000	2,000	1,000	10,000	100,000
10,000	40,000	10,000	100,000	1,000,000
100,000	600,000	100,000	1,000,000	10,000,000
1,000,000	8,000,000	1,000,000	10,000,000	100,000,000
10,000,000	80,000,000	10,000,000	100,000,000	1,000,000,000

* *p: # of records per page*
* *B=1,000 and block size=32 for sorting*
* *p=100 is the more realistic value.*

31

We are done with Chapter 13

Chapter 14 (only section 14.4)

32

Equality Joins With One Join Column

SELECT *
FROM Reserves R1, Sailors S1
WHERE R1.sid=S1.sid

$\bowtie$

- R ∞ S is very common, so must be carefully optimized.
- R X S is large; so, R X S followed by a selection is inefficient.
- Assume: M pages in R, p_R tuples per page, N pages in S, p_S tuples per page.
  - In our examples, R is Reserves and S is Sailors.
- We will consider more complex join conditions later.

33

Two Classes of Algorithms to Implement Join Operation

- Algorithms in class 1 require enumerating all tuples in the cross-product and discard tuples that do not meet the join condition.
  - Simple Nested Loops Join
  - Blocked Nested Loops Join
- Algorithms in class 2 avoid enumerating the cross-product.
  - Index Nested Loops Join
  - Sort-Merge Join
  - Hash Join

34

Simple Nested Loops Join

foreach tuple r in R do
  foreach tuple s in S do
    if ri == sj then add <r, s> to result

- For each tuple in the *outer* relation R, scan the entire *inner* relation S (scan S total of p_R * M times!).
  - Cost: M + p_R * M * N = 1000 + 100*1000*500 I/Os => very huge.
- How can we improve simple nested loops join?

35

Page Oriented Nested Loops Join

foreach page of R do
  foreach page of S do
    for all matching tuples r in R-block and
    s in S-page, add <r, s> to result

- Cost: M + M*N = 1000 + 1000*500 = 501,000 => still huge.
- If smaller relation (S) is outer, cost = 500 + 500*1000 = 500,500

36

Block Nested Loops Join

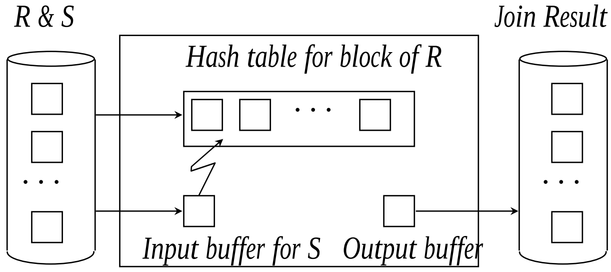
foreach block of B-2 pages of R do
  foreach page of S do
    for all matching tuples r in R-block and
    s in S-page, add <r, s> to result

- Use one page as an input buffer for scanning the inner S, one page as the output buffer, and use all remaining pages to hold ``block'' of outer R.
  - For each matching tuple r in R-block, s in S-page, add <r, s> to result. Then read next R-block, scan S, etc.

37

Block Nested Loops Join: Efficient Matching Pairs

- If B is large, it may be slow to find matching pairs between tuples in S-page and R-block (R-block has B-2 pages).
- The solution is to build a main-memory hash table for R-block.



38

Examples of Block Nested Loops

- Cost: Scan of outer + #outer blocks * scan of inner
  - #outer blocks = ceiling (# of pages of outer / blocksize)
- With Reserves (R) as outer, and 102 buffer pages:
  - Cost of scanning R is 1000 I/Os; a total of 10 *blocks*.
  - Per block of R, scan Sailors (S); 10*500 I/Os.
  - Total cost = 1000 + 10 * 500 = 6000 page I/Os => huge improvement over page-oriented nested loops join.
- With 100-page block of Sailors as outer:
  - Cost of scanning S is 500 I/Os; a total of 5 blocks.
  - Per block of S, we scan Reserves; 5*1000 I/Os.
  - Total cost = 500 + 5*1000 = 5500 page I/Os
- For blocked access (block I/Os are more efficient), it may be best to divide buffers evenly between R and S.

39

Index Nested Loops Join

foreach tuple r in R do
  foreach tuple s in S where ri == sj 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)
- For each R tuple, cost of probing S index is about 1.2 for hash index, 2-4 for B+ tree. Cost of finding S tuples (assume Alt. (2) or (3) for data entries) depends on clustering.
  - Clustered index: 1 I/O (typical)
  - Unclustered index: up to 1 I/O per matching S tuple.

40

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: 100 0+ 100,000 * 2.2 = 221,000 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.
  - Assume 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.
  - Total (Clustered): 500 + 40,000 * 2.2 = 88,500 I/Os.
- Given choices, put the relation with higher # tuples as inner loop.
- Index Nested Loop performs better than simple nested loop.

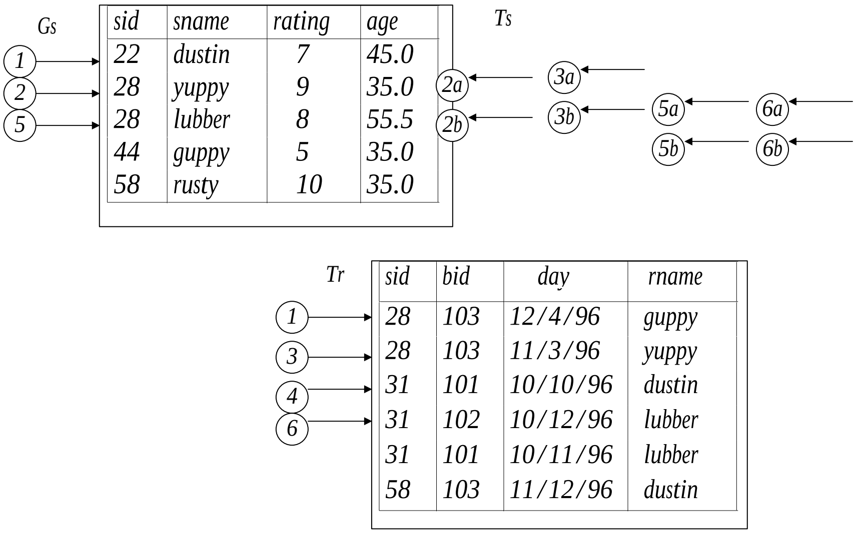
41

Sort-Merge Join

- Sort R and S on the join column [merge-sort], then scan them to do a ``merge'' (on join col.) [scan-merge], and output result tuples.
- Scan-merge:
  - 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.

42

Scan-Merge



sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
28	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

43

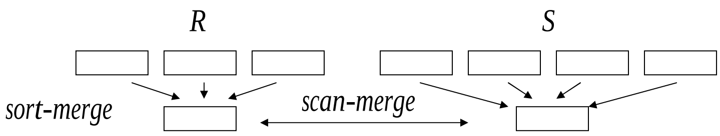
Cost of Merge-Sort Join

- 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.)
- Cost: 2 (M log M + N log N) + (M+N)
  - Assume enough buffer pages to sort both Reserves and Sailors in 2 passes
  - The cost of merge-sorting two relations is 2 M log M + 2 N log N.
  - The cost of scan-merge two sorted relations is M+N.
  - Total join cost: 2*2*1000 + 2*2*500 + 1000 + 500 = 7500 page I/Os.
- Any possible refinement to reduce cost?

44

Refinement of Sort-Merge Join

- Combine the merging phase in *sorting* with the scan-merge for the join.
  - Allocate one buffer space for each run (in the merge pass) in R & S.
  - Buffer size B > squar_root(L), where *L* is the size of the larger relation. Why?
    - # runs = 2(L/2B) = L/B < B [replacement sort]
  - Cost: read+write each relation in Pass 0 + read each relation in (only) merging pass (not counting the writing of result tuples).
  - Cost goes down from 7500 to 4500 I/Os

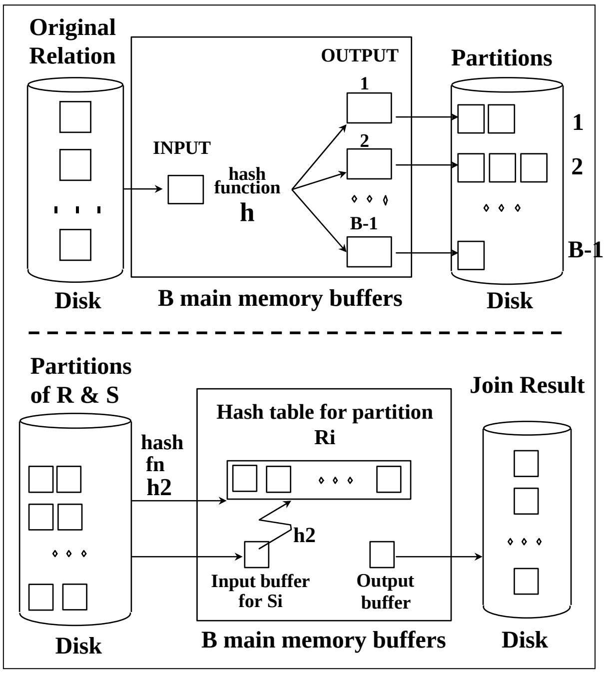


Hash Join

- Hash both relations on the join attribute using the same hash function h.
  - Tuples in R-partition_i (bucket) can only match with tuples in S-partition_i.
- For i=1..k, check for matching pairs in R-partition_i and S-partition_i.
- For efficient matching pairs, apply hashing to tuples of R-partition using another hash function h2.

Hash-Join

- Partition both relations using hash fn **h**: R tuples in partition i will only match S tuples in partition i.
- Read in a partition of R, hash it using **h2 (<> h!)**. Scan matching partition of S, search for matches.



Observations on Hash-Join

- #partitions k < B-1 (need one buffer page for reading), and B-2 > size of largest partition to be held in memory.
- Assume uniformly sized partitions, and maximize k, we get:
  - k= B-1, and B-2 > M/(B-1), i.e., B must be > square_root(M)
- If we build an in-memory hash table to speed up the matching of tuples, a little more memory is needed.
- If the hash function does not partition uniformly, one or more R partitions may not fit in memory. Can apply hash-join technique recursively to do the join of this R-partition with corresponding S-partition.

48

Cost of Hash-Join

- In partitioning phase, read+write both relns; 2(M+N). In matching phase, read both relns; M+N I/Os.
- In our running example, this is a total of 4500 I/Os.
- Sort-Merge Join vs. Hash Join:
  - Same amount of buffer pages.
  - Same cost of 3(M+N) I/Os.
  - Hash Join shown to be highly parallelizable.
  - Sort-Merge less sensitive to data skew; result is sorted.

49

Complex Join Conditions

- So far, we have only discussed single equality join condition.
- How about equalities over several attributes? (e.g., R.sid=S.sid AND R.rname=S.sname):
  - For Index Nested Loops join, build index on <sid, sname> on R (if R is inner); or use existing indexes on sid or sname.
  - For Sort-Merge and Hash Join, sort/partition on combination of the two join columns.

50