External Sorting:

2 passes size $N = B(B-1)$

space $= B \approx \sqrt{N}$

Sort N pages of data in about $\sqrt{N}$ space

Merge:

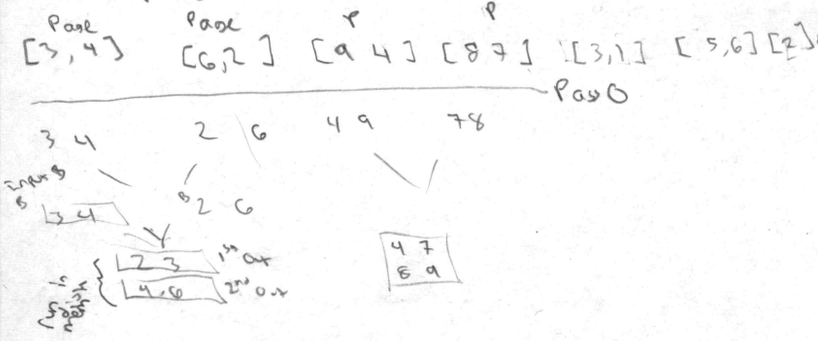
passes

2 way $\lceil \log_2 N \rceil + 1$

General $\lceil \log_{B-1} \lceil \frac{N}{B} \rceil \rceil + 1$ ← $\frac{N}{B}$ comes from pass 0, creates $\lceil \frac{N}{B} \rceil$ runs of size B.

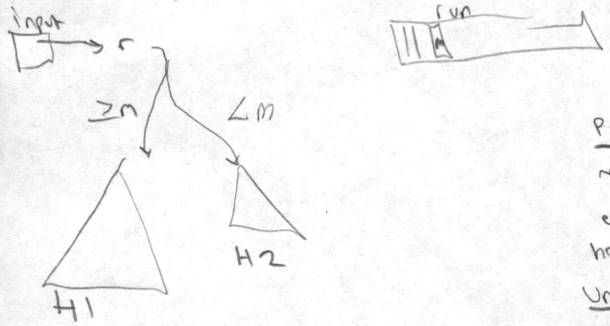
Total cost:

$2N \cdot [\#\text{ passes}]$



Replacement Sort / Tournament / Heapsort:

- load in B-2 pages into "current run" heap H1
- keep H2 initially empty for next run.
- Pop off an m and write to run
- read in r from memory and if $\geq m$ add r to H1, if $< m$ add to H2.
- done when H1 empty. H2 starts next run, H2 guaranteed sized B-2
- on average runs sized 2(B-2) node.



Double Buffering

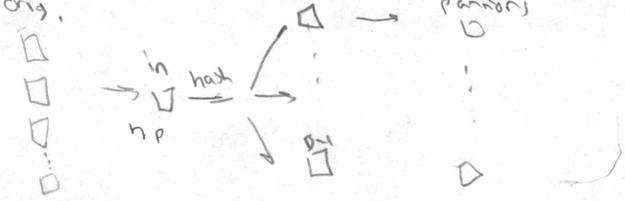
- CPU idle when have to read in input buffers and write out output buffer.
- allocate additional pages to input an output that are used when other set reading or writing so CPU not idle.

Terminology for Tree Index

- Fanout (F) = # of children per non-leaf node
- Order (d): min # of entries per node
- Ni: # of leaf pages

External Hashing: Good for removing duplicates or forming groups

1. Divide stage: use hashing function h_{B-1} to hash stream of incoming data into B-1 output buffers connected to B-1 partitions on disk



2. Rehash/conquer: use hashing funct. to read B-1 partitions created in first stage into RAM hash table. Right out complete hash table to disk

Runtime: 4N

Biggest table in two passes: $B(B-1)$. 1st stage creates B-1 partitions and partitions can't be more than B pages. Recursive partitioning can be used for data that can't be hashed in 2 passes.

B = # pages, D = (Avg.) time to read/write, R = # records/page.

	Heap File	Sorted File	Clustered File
Scan all records	BD	BD	1.5BD
Equality search	$\frac{1}{2}$BD	$(\log_2 B)D$	$(\log_{\lceil 1.5B \rceil} 1.5B+1)D$
Range search	BD	$(\log_2 B + \#\text{match})D$	$(\log_F 1.5B + \#\text{match})D$
Insert	$D+D$	$(\log_2 B + \frac{1}{2}B + \frac{1}{2}B)D$	$(\log_{\lceil 1.5B \rceil} 1.5B+2)D$
Delete	$\frac{1}{2}BD+D$ average!	$(\log_2 B + \frac{1}{2}B + \frac{1}{2}B)D$	$(\log_F 1.5B+2)D$

Page Formats

Packed: store records in 1st N slots (N = records on page): when record deleted, move last record into vacated slot: all empty slots at end. Con: Doesn't work if are external ref. to moved record (rid has slot # which moved)

Unpacked (Bitmap): handle deletions with array of bits locating record takes scanning bit map to find slots where bit at.

Slotted Page: used for variable length records and fixed length

Pro: need to move records on page for reasons other than tracking space freed by deletions

- when record deleted, move records to fill hole so free space contiguous
- maintain directory of slots/page with (offset, length) pair
- delete by setting offset to -1
- keep pointer to free space, when trying to put in record too large move records to reclaim freed space.

More on Indexing

- height of B+ tree depends on # data entries/size of index entries
- if search key values long not many index entries fit on page, fan-out low height large.
- high fan out more space efficient for duplicates.

Join Algorithms:

M = pages in outer (R)
N = pages in inner(S)
P_r = records/page

Nested Loops Join:

works for continuous stream if stream on outside.
- tuple at a time
- scan R and for each tuple in R scan all of S

total cost = $M + P_r M \cdot N$

Refinement: page at a time; each page R retrieve page S

cost = $M + M \cdot N$

optimize: choose R to be smaller

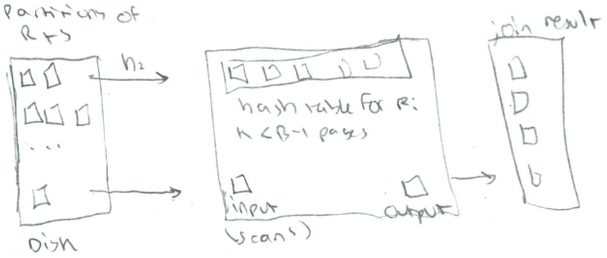
Index-Nested Loop Join

- index on on relations on join attribute; make indexed relation be inner relation.
- compare r only with tuples s in same partition = have same value in join.
- cost of scanning R = M, cost of retrieving matching S depends on kind of index / # of matches.
  1) if index on S in B+, cost 2-4 I/O. In hash 1-2
  2) once find leaf, cost of retrieving depends on clustering. If clustered cost +1, if not could be 1 per match s tuple (on diff. pages in worst case)

cost = $M + (M \cdot P_R)$(cost to leaf) + $M P_R$(cost to retrieve)

Hash-Join: works for equijoin, can't use if can't sort ie. infinite stream

- uses hashing to identify partitions
- Partitioning phase similar to external hashing
- probing phase illustrated:

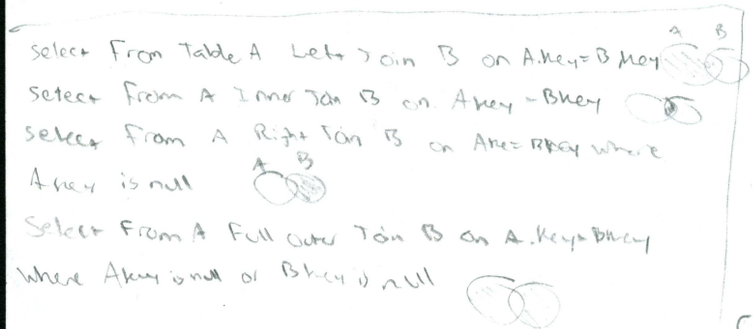


- in partitioning, scan R and S both once and write once ⇒ 2(M+N); 2nd phase scan each partition once for additional M+N

cost = 3(M+N) assuming length(partitions) ≤ B pages

Hybrid Hash-Join: can handle infinite streams

but bad choice
- keep one of hash buckets in memory



Block Nested Loops Join: bad for streams unless outer fits in memory

- suppose enough mem to hold R with 2 extra buffer pages
- read in smaller relation, use buffer scan S.

cost = M+N — if R fits in buffer

refinement: build in memory hash for smaller relation R, minimizes CPU cost.

- when R can't be loaded completely in; load in B-2 pages of R, scan S.

cost = $M + N\lceil \frac{M}{B-2} \rceil$

Sort Merge Join: can't use for duplicates or continuous stream

- sort both relations on join attribute then look for qualifying tuples by merging the two.
- external sorting alg. used to sort, if already sorted then no need.

Assuming no duplicates: cost = cost sort R + cost sort S + M+N

Refinement: combine the merging phase of sorting with merging phase of join.

- produce sorted runs of size B for R and S, $B > \sqrt{L}$ where L is size of larger, # runs per relation $< \sqrt{L}$
- suppose # buffer available for merging at least $2\sqrt{L}$; more than total # of runs of R and S. allocate 1 buffer per run of R and 1 for S.
- merge runs of R, runs of S, and resulting R and S streams.
- increases # of buffers to $2\sqrt{L}$, unless use tech. to produce runs of size 2B, makes it back to $\sqrt{L}$

cost = 3(M+N)

- enough memory for refinement: $B \geq 1 + \frac{R}{B} + \frac{S}{B}$ in 2 passes
  basically N = R+S
- mem requirement using tournament sort $\lceil \frac{S}{2(B-2)} \rceil + \lceil \frac{R}{2(B-2)} \rceil \leq B-1$

Clustered index:

- index data entries stored in approx order by value of search key
- file can be clustered on at most one search key

Pros: good for RANGE queries; potential locality
Cons: more expensive to maintain.

Indexes

Index - disk based data structure for fast lookup by value

Search key: any subset of columns in relation, need not be a key of relation (can be multiple items matching

3 Alternatives

1) Actual data record (with key value k)
2) <k, rid of matching data record)
3) <k, list of rids of matching data records>

CANNOT SORT INFINITE STREAMS