External Sorting

Chapter 13

Why Sort?

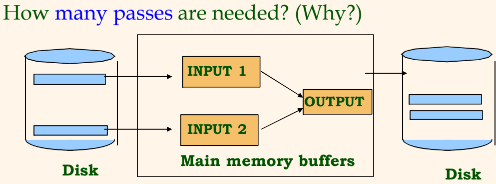
- A classic problem in computer science!
- Data requested in sorted order
  - e.g., find students in increasing *gpa* order
- Sorting is the first step in *bulk loading of* B+ tree index.
- Sorting is useful for eliminating *duplicate copies* in a collection of records (Why?)
- *Sort-merge* join algorithm involves sorting.
- Problem: sort 100Gb of data with 1Gb of RAM.
  - why not virtual memory?
- Take a look at sortbenchmark.com
- Take a look at main memory sort algos at www.sorting-algorithms.com

Sorting a file in RAM

- Three steps:
  - Read the entire file from disk into RAM
  - Sort the records using a standard sorting procedure, such as Shell sort, heap sort, bubble sort, …
  - Write the file back to disk
- How can we do the above when the data size is 100 or 1000 times that of available RAM size?
- And keep I/O to a minimum!
  - Effective use of buffers
  - Merge as a way of sorting
  - Overlap processing and I/O (e.g., heapsort)

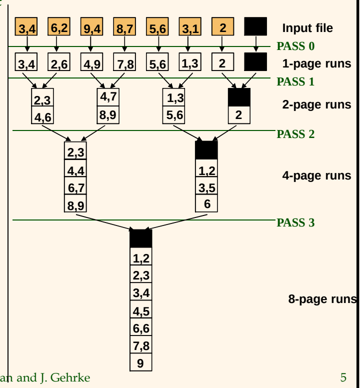
2-Way Sort of N pages

- Requires Minimum of 3 Buffers
- Pass 0: Read a page, sort it, write it.
  - only one buffer page is used
  - How many I/O's does it take?
- Pass 1, 2, 3, …, etc.:
  - Minimum three buffer pages are needed! (Why?)
  - How many i/o's are needed in each pass? (Why?)
  - How many passes are needed? (Why?)



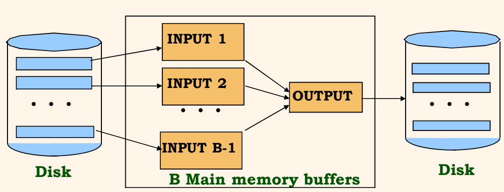
Two-Way External Merge Sort

- Each pass we read + write each page in file.
- N pages in the file => the number of passes
  $= \lceil \log_2 N \rceil + 1$
- So total cost is:
  $$2N\left(\lceil \log_2 N \rceil + 1\right)$$
- *Idea:* ***Divide and conquer:*** sort subfiles and merge
- Can we improve upon this? How?



General External Merge Sort

- Suppose we can have more than 3 buffers! Can we use them effectively?
- 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 2, …, etc.: merge *B-1* runs.



Cost of External Merge Sort

- Number of passes: $1 + \lceil \log_{B-1} \lceil N / B \rceil \rceil$
- Cost = 2N * (# of passes)
- E.g., with 5 buffer pages, to sort 108 page file:
  - 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
- Note that with 3 buffers, initial can be of 3-page runs (not 1)
  - Cost is: 1 + $2N(\lceil \log_2 N/3 \rceil + 1)$

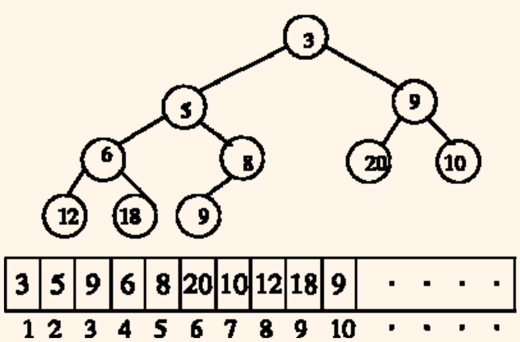
Number of Passes of External Sort

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

Internal (main memory) Sort Algorithm

- Quicksort is a fast way to sort in memory.
  - A divide-and-conquer algorithm
  - Partition initial array into 2 (preferably equal size) with some property for each partition
  - Sort each partition recursively
  - In-place sort algorithm
  - Sorts a fixed size input to generate a fixed-size output!
- An alternative is "tournament sort" (aka "heapsort")
  - You build a max- or min-heap (binary tree with some property)
    - A node's key >= its children's keys
    - Can be implemented using an array
  - Can HEAPIFY a HEAP to insert a new value!

Min-Heap



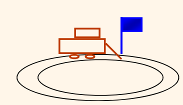
http://www.sorting-algorithms.com/random-initial-order

Internal (main memory) Sort Algorithm

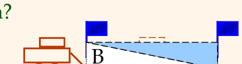
- Given B buffers, Use 1 input, B-2 current set, and 1 output buffer
- Use heapsort on the current set and output the smallest to output buffer
- Insert new record into current set and output the smallest from current set which is greater than the largest in the output (for ascending sort)
- Terminating condition
  - when all values in the current set is smaller than the output, start a new run
- Instead of discreet sort, we are doing a continuous sort (snow plow example)

More on Heapsort

- Fact: average length of a run in heapsort is *2B*
  - The "snowplow" analogy
- Worst-Case:
  - What is min length of a run?
  - How does this arise?



- Best-Case:
  - What is max length of a run?
  - How does this arise?



- Quicksort is faster, but our aim is to reduce the number of initial runs and hence reduce the number of passes!!

I/O for External Merge Sort

- … longer runs often means fewer passes!
- We are assuming that I/O is done one page at a time
- In fact, can read a *block* of pages sequentially!
  - Much faster/cheaper than reading pages of the block individually
- Suggests we should make each buffer (input /output) be a *block* of pages.
  - But this will reduce fan-out during merge passes! Why?
  - In practice, most files still sorted in 2-3 passes.
- Minimizes I/O cost, not the # of I/O's
- Also, double buffering. What does this reduce?

I/O for External Merge Sort (2)

- Blocked I/O
  - Suppose a block is b pages
  - We need b buffer pages for output (1 block)
  - We can only merge ceiling ((B-b)/b) runs (instead of B-1 runs when we read 1 page at a time)
  - If we have 10 buffer pages, we can
    - Either merge nine runs without using blocks, or
    - Four runs if we assume 2 page blocks
  - This tradeoff between using blocks vs. the number of runs needs to be taken into account for external merge sort!
  - The good news is that with greater memory, both block sizes and #runs can be kept to a decent value

Number of Passes of Optimized Sort

N	B=1,000	B=5,000	B=10,000
100	1	1	1
1,000	1	1	1
10,000	2	2	1
100,000	3	2	2
1,000,000	3	2	2
10,000,000	4	3	3
100,000,000	5	3	3
1,000,000,000	5	4	3

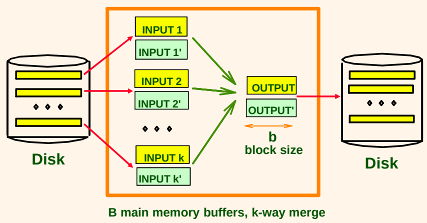
Block size = 32, initial pass produces runs of size 2B.

Blocked I/O

- Let b be the units of read and write
- Given B buffers, # of runs that can be merged is floor ((B-b)/b)
- If we have 10 buffers, we can
  - Merge 9 runs at a time with 1 page buffer, or
  - Merge 4 runs at a time with 2 page input (for each block) and output buffer blocks
- How does it reduce I/O cost?

Double Buffering

- To reduce wait time for I/O request to complete, can *prefetch* into `shadow block'.
  - Tradeoff between buffers and passes; in practice, most files *still* sorted in 2-3 passes.



Sorting Records! (http://sortbenchmark.org/)

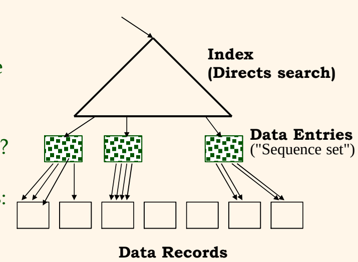
Gray	2013, 1.42 TB/min **Hadoop** 102.5 TB in 4,328 seconds 2100 nodes x (2 2.3Ghz hexcore Xeon E5-2630, 64 GB memory, 12x3TB disks) Thomas Graves Yahoo! Inc.	2013, 1.42 TB/min **Hadoop** 102.5 TB in 4,328 seconds 2100 nodes x (2 2.3Ghz hexcore Xeon E5-2630, 64 GB memory, 12x3TB disks) Thomas Graves Yahoo! Inc.
Penny	2011, 286 GB **psort** 2.7 Ghz AMD Sempron, 4 GB RAM, 5x320 GB 7200 RPM Samsung SpinPoint F4 HD332GJ, Linux Paolo Bertasi, Federica Bogo, Marco Bressan and Enoch Peserico Univ. Padova, Italy	2011, 334 GB **psort** 2.7 Ghz AMD Sempron, 4 GB RAM, 5x320 GB 7200 RPM Samsung SpinPoint F4 HD332GJ, Linux Paolo Bertasi, Federica Bogo, Marco Bressan and Enoch Peserico Univ. Padova, Italy

Using B+ Trees for Sorting

- Scenario: 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
    - ***Good idea!***
  - B+ tree is not clustered
    - ***Could be a very bad idea!***

Clustered B+ Tree Used for Sorting

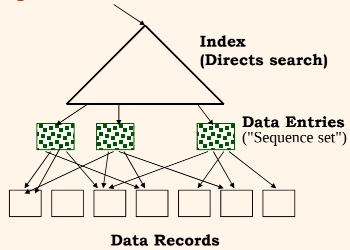
- Cost: root to the left-most leaf, then retrieve all leaf pages (Alternative 1)
- If Alternative 2 is used? Additional cost of retrieving data records: each page fetched just once.



Always better than external sorting!

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!



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.***

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*** *or more* (# buffer pages). Later passes: ***merge*** runs.
  - # of runs merged at a time depends on ***B***, and ***block size***.
  - Larger block size means less I/O cost per page.
  - Larger block size means smaller # runs merged.
  - In practice, # of passes rarely more than 2 or 3.

Summary, cont.

- Choice of internal sort algorithm may matter:
  - Quicksort: Quick!
  - Heap/tournament sort: slower (2x), longer runs
- The best sorts are wildly fast:
  - Despite 40+ years of research, we're still improving!
- Clustered B+ tree is good for sorting
- Unclustered B+ tree is usually very bad.

Sort-Merge Join $(R \bowtie_{i=j} S)$ (p. 460, 3rd ed.)

- Sort R and S on the join column, then scan them to do a ``merge'' (on join col.), and output result tuples.
  - 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. (Multiple scans of an S group are likely to find needed pages in buffer!). Depends upon buffer management policy!!

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: M log M + N log N + (M+N)
  - The cost of scanning, M+N, could be M*N (very unlikely!)
- With 35, 100 or 300 buffer pages, both Reserves and Sailors can be sorted in 2 passes; total join cost: 7500.

(BNL cost: 2500 to 15000 I/Os)

Refinement-1 of Sort-Merge Join

- We can combine the merging phase of *sorting* of R and S with the merging phase of join.
  - Let L be the size (in pages) of the larger relation
  - In order to manage L/B runs in pass 1, you need at least
    - L/B + 1 buffers
  - Hence, B > L/B or B^2 > L or B > $\sqrt{L}$
  - If the # of buffers available for the merge phase is 2 $\sqrt{L}$, that is, more than the number of runs of R and S
    - We allocate one buffer for each run of R and one for each run of S
    - We then merge the runs of R and S streams as they are generated. we apply the join condition and discard tuples if they do not join.

Refinement-1 of Sort-Merge Join (Contd.)

- Cost: read+write each relation in Pass 0 + (only) read each relation in merging pass (+ writing of result tuples).
  - 3 * (M+N)
- In example, cost goes down from 7500 to 4500 I/Os
  - 3 * (1000+500) = 4500
- In practice, cost of sort-merge join, like the cost of external sorting, can be *linear*.

Refinement-2 of Sort-Merge Join

- This changes the number of buffers required to $\sqrt{2*L}$
- We apply the heapsort optimization to produce runs of size 2*B.
- Hence, we will have L /2*B runs of each relation, given the assumption that we have B buffers.
- Thus the number of buffers is B > L/2*b +1, or
- B > $\sqrt{L/2}$
- Hence we only need B > $\sqrt{2L}$ buffers instead of 2*$\sqrt{L}$ with this optimization.

Spatial Indexes

- **Spatial indices** are used by spatial databases (databases which store information related to objects in space) to optimize spatial queries. Conventional index types do not efficiently handle spatial queries such as how far two points differ, or whether points fall within a spatial area of interest. Common spatial index methods include:
  - Grid (spatial index)
  - Z-order (curve)
  - Quadtree
  - Octree

Spatial Indices

- R-tree: Typically the preferred method for indexing spatial data. Objects (shapes, lines and points) are grouped using the minimum bounding rectangle (MBR). Objects are added to an MBR within the index that will lead to the smallest increase in its size.
  - R+ tree
  - R* tree
  - Hilbert R-tree
  - kd-tree

Others

- Bit map index
  - A **bitmap index** is a special kind of database index that uses bitmaps. Bitmap indexes have traditionally been considered to work well for *low-cardinality columns*, which have a modest number of distinct values, either absolutely, or relative to the number of records that contain the data.
- Bloom filters
  - The purpose of a bloom filter is to indicate, with some chance of error, whether an element belongs to a set. This error refers to the fact that it is possible that the bloom filter indicates some element *is* in the set, when it in fact *is not* in the set (false positive). The reverse, however, is not possible – if some element *is* in the set, the bloom filter cannot indicate that it *is not* in the set (false negative).