Sorting and Hashing

See R&G Chapters:
9.1, 13.1-13.3, 13.4.2

Why Sort?

- Rendezvous
  - Eliminating duplicates
  - Summarizing groups of items
- Ordering
  - Sometimes, output must be ordered
  - e.g., return results in decreasing order of relevance
- Upcoming fundamentals:
  - *Sort-merge join* algorithm involves sorting (rendezvous)
  - First step in bulk loading *tree indexes* (ordering)
- Problem: sort 100GB of data with 1GB of RAM.
  - why not virtual memory?

But First…

- Important to know a little something about disks

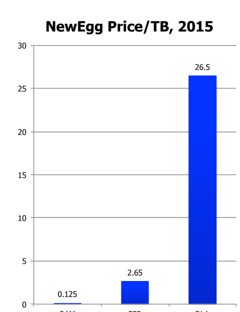
Disks and Files

- A lot of databases still use magnetic disks.
  - Disks are a mechanical anachronism!
- Major implications!
  - No "pointer derefs". Instead, an API:
    - READ: transfer "page" of data from disk to RAM.
    - WRITE: transfer "page" of data from RAM to disk.
  - Both API calls are expensive
    - Plan carefully!
  - An explicit API can be a good thing
    - Minimizes the kind of pointer errors you see in C

Economics

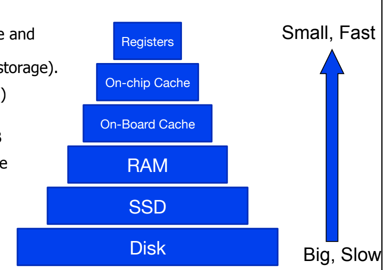
For $1000, NewEgg offers:
- ~0.125TB of RAM
- ~2.65TB of Solid State Disk
- ~26.5TB of Magnetic Disk

(desktop grade disks)

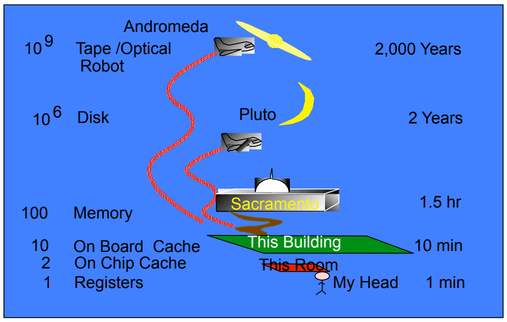


The Storage Hierarchy

- Main memory (RAM) for currently used data.
- Disk for main database and backups/logs (secondary & tertiary storage).
- The role of Flash (SSD) varies by deployment
  - Sometimes the DB
  - Sometimes a cache

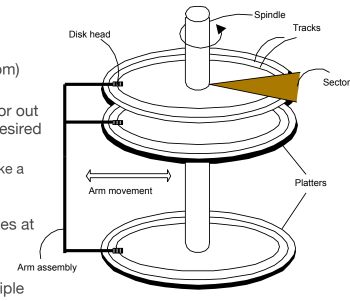


Jim Gray's Latency Analogy: How Far Away is the Data?



Components of a Disk

- Platters spin (say 7200 rpm)
- Arm assembly moved in or out to position a head on a desired track.
  - Tracks under heads make a cylinder (imaginary)
- Only one head reads/writes at any one time
- Block/page size is a multiple of (fixed) sector size



Accessing a Disk Page

- Time to access (read/write) a disk block:
  - seek time (moving arms to position disk head on track)
    - ~8-10msec on average
  - rotational delay (waiting for block to rotate under head)
    - ~2-4msec (at 7200 RPM one rotation is 4.2msec)
  - transfer time (actually moving data to/from disk surface)
    - ~0.3 msec per 64KB page
- Key to lower I/O cost: reduce seek/rotation delays!
  - Hardware vs. software solutions?

Arranging Pages on Disk

- 'Next' block concept:
  - blocks on same track, followed by
  - blocks on same cylinder, followed by
  - blocks on adjacent cylinder
- Arrange file pages sequentially on disk
  - minimize seek and rotational delay.
- For a sequential scan, pre-fetch
  - several pages at a time!

Notes on Flash (SSD)

- Various technologies, things still evolving
- Read is smallish and fast
  - Single read access time: 0.03 ms
  - 4KB random reads: ~500MB/sec
  - Sequential reads: ~525MB/sex
- Write is slower for random
  - Single write access time: 0.03ms
  - 4KB random writes: ~120MB/sec
  - Sequential writes: ~480MB/sec
- Some concern about write endurance
  - 2K-3K cycle lifetimes?
  - 6-12 months?

http://www.tomshardware.com/charts/ssd-charts-2014/benchmarks,129.html
http://www.storagesearch.com/ssdmyths-endurance.html

Storage Pragmatics & Trends

- Many significant DBs are not that big.
  - Daily weather, round the globe, 1929-2009: 20GB
  - 2000 US Census: 200GB
  - 2009 English Wikipedia: 14GB
- But data sizes grow faster than Moore's Law
- What is the role of disk, flash, RAM?
  - The subject of some debate!

Bottom Line (for now!)

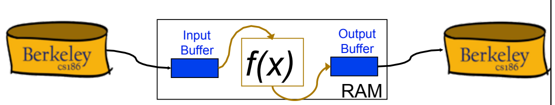
- Very Large DBs: relatively traditional
  - Disk still the best cost/MB by orders of magnitude
  - SSDs improve performance and *performance variance*
- Smaller DB story is changing quickly
  - Entry cost for disk is not cheap, so flash wins at the low end
  - Many interesting databases fit in RAM
- Change brewing on the HW storage tech side
- Lots of uncertainty on the SW/usage side
  - It's Big: Can generate and archive data cheaply and easily
  - It's Small: Many rich data sets have (small) fixed size
- Hmmm…!
- Many people will continue to worry about magnetic disk for some time yet.

Meanwhile…

- Back in the land of out-of-core algs…

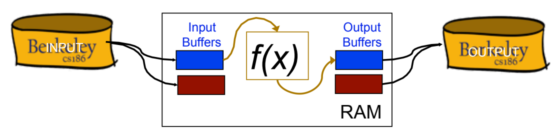
Remember this slide?

- Simple case: "Map".
  - Goal: Compute *f(x)* for each record, write out the result
  - Challenge: minimize RAM, call read/write rarely
- Approach
  - Read a chunk from INPUT to an *Input Buffer*
  - Write *f(x)* for each item to an *Output Buffer*
  - When Input Buffer is consumed, read another chunk
  - When Output Buffer fills, write it to OUTPUT



Better: Double Buffering

- Main thread runs *f(x)* on one pair I/O bufs
- 2nd "I/O thread" fills/drains unused I/O bufs
- Main thread ready for a new buf? Swap!
- Usable in any of the subsequent discussion
  - Assuming you have RAM buffers to spare!
  - But for simplicity we won't bring this up again.

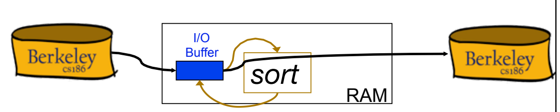


Sorting & Hashing: Formal

- Given:
  - A file *F*:
    - containing a multiset of records *R*
    - consuming **N** blocks of storage
  - Two "scratch" disks
    - each with >> N blocks of free storage
  - A fixed amount of space in RAM
    - memory capacity equivalent to **B** blocks of disk
- Sorting
  - Produce an output file F_S
    - with contents *R* ***stored in order by a given sorting criterion***
- Hashing
  - Produce an output file F_H
    - with contents *R*, ***arranged on disk so that no 2 records that are incomparable*** (i.e. "equal" in sort order) ***are separated by a greater or smaller record.***
    - I.e. matching records are always "stored consecutively" in F_H.

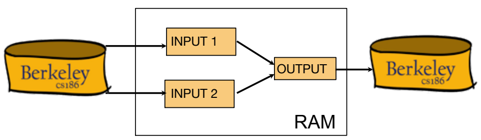
Sorting: 2-Way

- Pass 0 (conquer):
  - read a page, sort it, write it.
  - only one buffer page is used
  - a repeated "batch job"



Sorting: 2-Way

- Pass 0 (conquer):
  - read a page, sort it, write it.
  - only one buffer page is used
  - a repeated "batch job"
- Pass 1, 2, 3, …, etc. (merge):
  - requires **3 buffer pages**
    - note: this has nothing to do with double buffering!
  - *merge pairs* of runs into runs twice as long
  - a streaming algorithm, as in the previous slide!



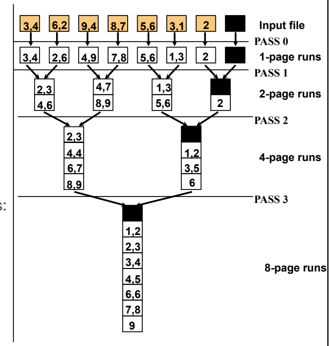
Two-Way External Merge Sort

- ***Conquer and Merge:*** sort subfiles and merge
- Each pass we read + write each page in file.
- N pages in the file. So, the number of passes is:

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

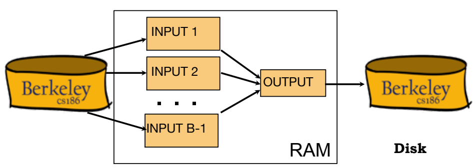
- So total cost is:

$$2N\left(\lceil \log_2 N \rceil + 1\right)$$



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.



Merging 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: 1 run => Sorted file of 108 pages

Formula check: $1 + \lceil \log_4 22 \rceil = 1 + 3$ → 4 passes √

of Passes of External Sort

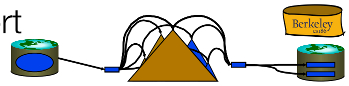
(I/O cost is 2N times number of passes)

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

Memory Requirement for External Sorting

- How big of a table can we sort in two passes?
  - Each "sorted run" after Phase 0 is of size B
  - Can merge up to B-1 sorted runs in Phase 1
- Answer: B(B-1).
  - Sort N pages of data in about $\sqrt{N}$ space

Internal Sort

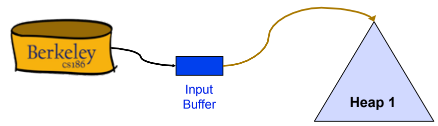


- Quicksort is a fast way to sort in memory.
- Alternative: “tournament sort”
  - a.k.a. “heapsort”, “replacement selection”
- Keep two heaps in memory, **H1** and **H2**

```
read B-2 pages of records, inserting into H1;
 while (records left) {
     m = H1.removemin();  put m in output buffer;
     if (H1 NOT empty)
         read in a new record r (use 1 buffer for
           input pages);
         if (r < m)   H2.insert(r);
         else         H1.insert(r);
     else
         H1 = H2;  H2.reset();
         start new output run;
 }
 H1.output();  start new run;  H2.output();
```

Tournament Sort (Heapsort)

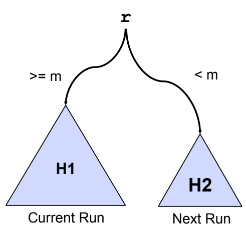
First, load a heap with B-2 pages of records



This is a priority heap, so it is in sorted order
(hence called Heapsort)

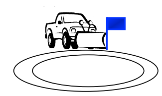
Tournament Sort (2)

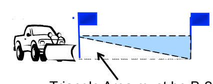
```
while (records left on input) {
        m = H1.removemin(); // get smallest value
        Output(m)           // put m in output buffer;
        if (H1 NOT empty)
            r = InputRecord()
            if (r < m)   H2.insert(r);
            else         H1.insert(r);



        else
            H1 = H2;  H2.reset();
            start new output run;
    }
```



More on Heapsort

- Fact: **average length of a run = 2(B-2)**
  - 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 ... longer runs often means fewer passes!

Alternative: Hashing

- Idea:
  - Many times we don’t require order
  - E.g.: removing duplicates
  - E.g.: forming groups
- Often just need to *rendezvous* matches
- Hashing does this
  - And may be cheaper than sorting! (Hmmm…!)
  - But how to do it out-of-core??

Divide

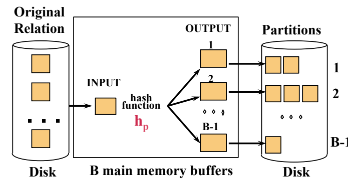
- Streaming Partition (divide):
  Use a hash function h_p to stream records to disk partitions
  - All matches rendezvous in the same partition.
  - *Streaming* algorithm to create partitions on disk:
    - “Spill” partitions to disk via output buffers

Divide & Conquer

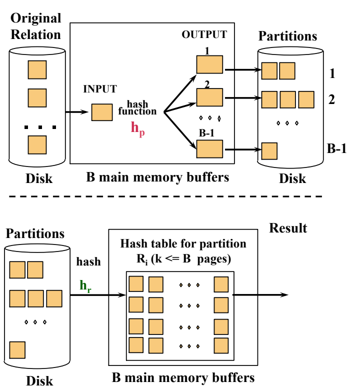
- Streaming Partition (divide):
  Use a hash function h_p to stream records to disk partitions
  - All matches rendezvous in the same partition.
  - *Streaming* algorithm to create partitions on disk:
    - "Spill" partitions to disk via output buffers
- ReHash (conquer):
  Read partitions into RAM hash table one at a time, using different hash h_r
  - Then go through each bucket of this hash table to achieve rendezvous in RAM
    (rendezvous in the bucket)

Two Phases

- Partition:
  (Divide)

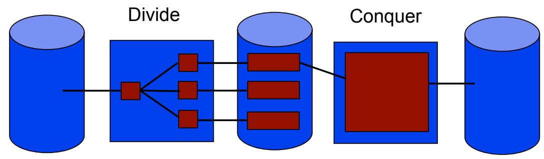


Two Phases



- Partition:
  (Divide)
- Rehash:
  (Conquer)

Cost of External Hashing



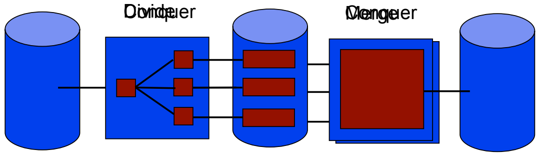
cost = 4*N IO's

Memory Requirement

- How big of a table can we hash in two passes?
  - B-1 "partitions" result from Pass 1
  - Each should be no more than B pages in size
  - Answer: B(B-1).
    - We can hash a table of size N pages in about $\sqrt{N}$ space
  - Note: assumes hash function distributes records evenly!
- Have a bigger table? Recursive partitioning!

How does this compare with external sorting?

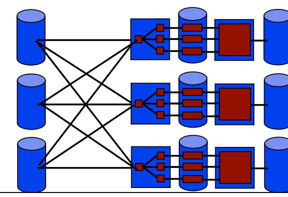
Cost of External Sorting/Hashing



cost = 4*N IO's

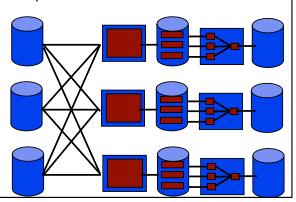
Parallelize me! Hashing

- Phase 1: partition data across machines
  - *streaming* out to network as it is scanned
  - which machine for this record?
    use (yet another) independent hash function h_n
- Receivers proceed with phase 1
  as data streams in
  - from local disk
    and network



Parallelize me! Sorting

- Pass 0: partition data across machines
  - *streaming* out to network as it is scanned
  - which machine for this record?
    check value range (e.g. [-∞,10], [11,100], [101, ∞]).
- Receivers proceed with pass 0
  as data streams in
- A Wrinkle: How to ensure ranges are the same size?!
  - i.e. avoid data skew?



So which is better ??

- Simplest analysis:
  - Same memory requirement for 2 passes
  - Same I/O cost
  - But we can dig a bit deeper…
- Sorting pros:
  - Great if input already sorted (or almost sorted) w/heapsort
  - Great if need output to be sorted anyway
  - Not sensitive to "data skew" or "bad" hash functions
- Hashing pros:
  - For duplicate elimination, scales with # of values
    - Not # of items! We'll see this again.
  - Can simply conquer sometimes! (Think about that)

Summary

- Sort/Hash Duality
  - Hashing is Divide & Conquer
  - Sorting is Conquer & Merge
- Sorting is overkill for rendezvous
  - But sometimes a win anyhow
- Sorting sensitive to internal sort alg
  - Quicksort vs. HeapSort
  - In practice, QuickSort tends to win
- Don't forget double buffering