External Sorting

R & G Chapter 13

"There it was, hidden in alphabetical order."

Rita Holt

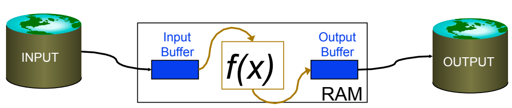
Why Sort?

- A classic problem in computer science!
- Data requested in sorted order
  - e.g., find students in increasing *gpa* order
- First step in *bulk loading* B+ tree index.
- Useful for eliminating *duplicates* (Why?)
- Useful for summarizing groups of tuples
- *Sort-merge* join algorithm involves sorting.

- Problem: sort 100Gb of data with 1Gb of RAM.
  - why not virtual memory?

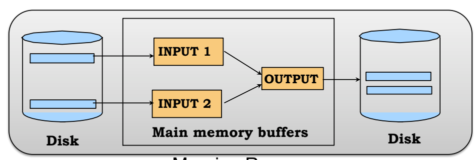
Streaming Data Through RAM

- An important method for sorting & other DB operations
- Simple case:
  - Compute *f(x)* for each record, write out the result
  - Read a page from INPUT to Input Buffer
  - Write *f(x)* for each item to Output Buffer
  - When Input Buffer is consumed, read another page
  - When Output Buffer fills, write it to OUTPUT
- Reads and Writes are *not* coordinated
  - E.g., if f() is Compress(), you read many pages per write.
  - E.g., if f() is DeCompress(), you write many pages per read.



2-Way Sort: Requires 3 Buffers

- Pass 0: Read a page, sort it, write it.
  - only one buffer page is used (as in previous slide)
- Pass 1, 2, 3, …, etc.:
  - requires 3 buffer pages
  - merge pairs of runs into runs twice as long
  - three buffer pages used.



Merging Runs

Two-Way External Merge Sort

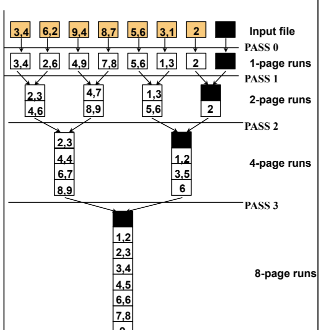
- Each pass we read + write each page in file.
- N pages in the file => the number of passes

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

- So total cost is:

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

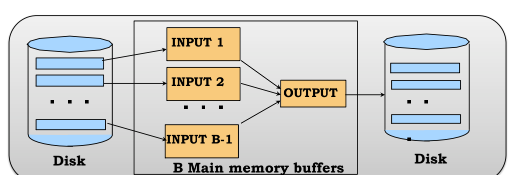
- *Idea: Divide and conquer:* sort subfiles and merge



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: Sorted file of 108 pages

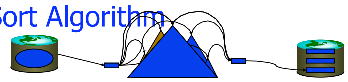
Formula check: $\lceil \log_4 22 \rceil$ = 3 … + 1 → <u>4 passes</u> √

Number 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

Internal Sort Algorithm

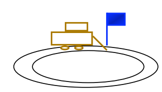


- Quicksort is a fast way to sort in memory.
- Alternative: "tournament sort" (a.k.a. "heapsort", "replacement selection")
- Keep two heaps in memory, H1 and H2

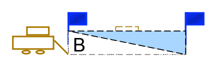
```
read B-2 pages of records, inserting into H1;
 while (records left) {
    m = H1.removemin();  put m in output buffer;
    if (H1 is empty)
        H1 = H2;  H2.reset();
        start new output run;
    else
        read in a new record r (use 1 buffer for
          input pages);
        if (r < m)  H2.insert(r);
        else        H1.insert(r);
 }
 H1.output();  start new run;  H2.output();
```

More on Heapsort

- Fact: average length of a run in heapsort is *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?



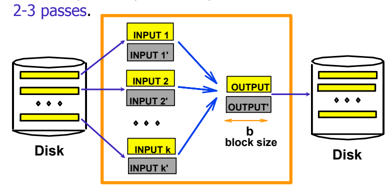
- Quicksort is faster, but ... longer runs often means fewer passes!

I/O for External Merge Sort

- Do I/O a page at a time
  - Not one I/O per record
- In fact, read a *block* *(chunk)* of pages sequentially!
- Suggests we should make each buffer (input/output) be a *block* of pages.
  - But this will reduce fan-in during merge passes!
  - In practice, most files still sorted in 2-3 passes.

Double Buffering

- Goal: reduce wait time for I/O requests during merge
- Idea: 2 blocks RAM per run, disk reader fills one while sort merges the other
  - Potentially, more passes; in practice, most files *still* sorted in 2-3 passes.



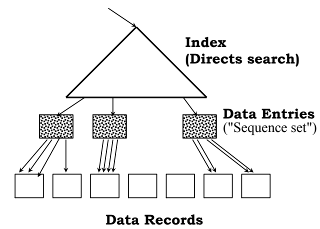
B main memory buffers, k-way merge

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

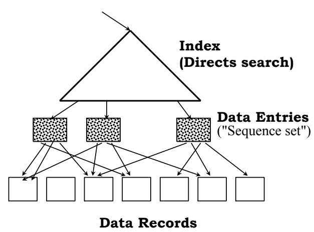
- Cost: root to the left-most leaf, then retrieve all leaf pages (Alternative 1)
- If Alternative 2 is used? Additional cost of retrieving data records: each page fetched just once.



☛ ***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!



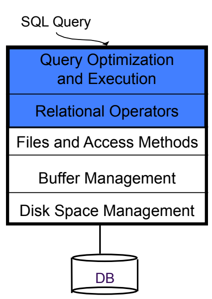
External Sorting vs. Unclustered Index

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.***

Context

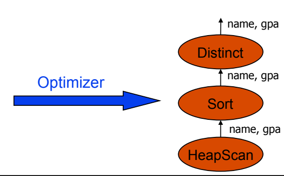
- Recall the stack:



Query Processing Overview

- The *query optimizer* translates SQL to a special internal "language"
  - Query Plans
- The *query executor* is an *interpreter* for query plans
- Think of query plans as "box-and-arrow" *dataflow* diagrams
  - Each box implements a *relational operator*
  - Edges represent a flow of tuples (columns as specified)
  - For single-table queries, these diagrams are straight-line graphs

```
SELECT DISTINCT name, gpa
  FROM Students
```



Iterators

iterator

- The relational operators are all subclasses of the class `iterator`:

```
class iterator {
   void init();
   tuple next();
   void close();
   iterator &inputs[];
   // additional state goes here
}
```

- Note:
  - Edges in the graph are specified by inputs (max 2, usually)
  - Encapsulation: any iterator can be input to any other!
  - When subclassing, different iterators will keep different kinds of state information

Example: Sort

```
class Sort extends iterator {
   void init();
   tuple next();
   void close();
   iterator &inputs[1];
   int numberOfRuns;
   DiskBlock runs[];
   RID nextRID[];
}
```

- `init()`:
  - generate the sorted runs on disk
  - Allocate `runs[]` array and fill in with disk pointers.
  - Initialize numberOfRuns
  - Allocate nextRID array and initialize to NULLs
- `next()`:
  - nextRID array tells us where we're "up to" in each run
  - find the next tuple to return based on nextRID array
  - advance the corresponding nextRID entry
  - return tuple (or EOF -- "End of Fun" -- if no tuples remain)
- `close()`:
  - deallocate the runs and nextRID arrays

Postgres Version

- src/backend/executor/nodeSort.c
  - ExecInitSort (init)
  - ExecSort (next)
  - ExecEndSort (close)
- The encapsulation stuff is hardwired into the Postgres C code
  - Postgres predates even C++!
  - See src/backend/execProcNode.c for the code that "dispatches the methods" explicitly!

Summary

- External sorting is important
- External merge sort minimizes disk I/O cost:
  - Pass 0: Produces sorted ***runs*** of size ***B*** (# 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 runs 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, still improving!
- Clustered B+ tree is good for sorting; unclustered tree is usually very bad.