Why Sorting?

- users usually want data sorted (**ORDER BY**)
- first step in bulk-loading a B+ tree
- used in duplicate elimination (how?)
- the **sort-merge join** algorithm (later in class) involves sorting as a first step
SORTING IN DATABASES

• Why don’t the standard sorting algorithms work for databases?
  – merge sort
  – quick sort
  – heap sort

• The data typically does not fit in memory!
**Example: Merge Sort**

- Sorting $n$ tuples needs $n \log(n)$ comparisons
- If we do a record-based sorting, we will need $n \log(n)$ I/Os
- **Key idea**: sort based on pages and not records!
The Sorting Problem

- $M$ available memory pages
- a relation $R$ of size $N$ pages (where $N > M$)
- **SORTING**: output a relation $R'$ that is sorted on a given sort key
- **Desiderata**:
  - sort large relations with *small* amounts of memory
  - minimize the number of disk I/Os
  - use sequential I/Os rather than random I/Os
  - Overlap I/O with CPU operations & minimize CPU
Warm Up: 2-Way Sort

- **run**: a sorted sub-file generated in intermediate steps of the sorting algorithm

- **Pass 0**: {requires 1 buffer page}
  - read a page, sort it, write it

- **Pass 1, 2, 3, ...**: {requires 3 buffer pages}
  - read 2 runs, merge them into one run
2-Way Sort: Analysis

• # passes = \[\log_2 N\] + 1

• I/Os per pass = 2N

• Total I/Os = 2N(\[\log_2 N\] + 1)
**Example**

- 1,000,000 records
- each record has 32 bytes
- each page has 8KB
- sort key is 4 bytes
**Can we do better?**

- The 2-way merge algorithm only uses 3 buffer pages
- How can we utilize the fact that we have more available memory?
- **Key idea:** use as much memory as possible in every pass!
  - reducing the number of passes reduces I/O
GENERAL EXTERNAL SORT

• $B$ buffer pages available
• Pass 0:
  – read $B$ buffer pages at a time and sort
  – produces $[N/B]$ runs
• Pass 1, 2, 3, ...:
  – load $B-1$ runs and merge them into one run
General External Sort: Analysis

- \# passes = \left\lceil \log_{B-1} \left( \frac{N}{B} \right) \right\rceil + 1

- I/Os per pass = 2N

- Total I/Os = 2N \left( \left\lceil \log_{B-1} \left( \frac{N}{B} \right) \right\rceil + 1 \right)
EXAMPLE

- 1,000,000 records
- each record has 32 bytes
- each page has 8KB
- sort key is 4 bytes

- Memory has 10 pages available
**Improvement: Replacement Sort**

- used as an alternative for sorting in pass 0
- creates *average runs* of size $2B$
- Algorithm:
  - read $B$-2 pages in memory (keep as sorted heap)
  - move smallest record (that is greater than the largest element in buffer) to output buffer
  - read a new record $r$ and insert into the sorted heap
**Improvement: Blocked I/O**

- reading a block of pages sequentially is faster!
- Make each buffer slot be a block of pages
  - reduces per page I/O cost. Side-effect?

**Analysis**
- Pass 0: creates \([N/2B]\) runs
- can merge \(F = \lfloor B/b \rfloor - 1\), where \(b\) is block size
- # passes: \([\log_F[N/2B]] + 1\)
  - however, less I/O per pass!
**Improvement: Double Buffering**

- So far we have considered only I/O costs
- But CPU may have to wait for I/O!
- **Idea:** keep a second set of buffers so that I/O and CPU overlap
**Using B+ Trees to Sort**

• Can the data be already sorted?
  – yes, if we have created a B+ tree index for the key!
  – the leaves have the entries in sorted order

• There are two possibilities here:
  – clustered B+ tree
  – unclustered B+ tree
SORTING WITH CLUSTERED B+ TREE

• Retrieve the leftmost entry
• Sweep through the leaf pages in order
• For each leaf page, read the data pages
• Cost:
  – If data is in not the index:
    Height + #pages in index + #data pages
  – If data is in the index:
    Height + #pages in index
**Sorting with Unclustered B+ Tree**

- In worst-case, I/Os can be as many as the number of records!
- Even in average case slower than external sorting