# External Sorting

*CS 564- Fall 2015*

ACKs: *Dan Suciu, Jignesh Patel, AnHai Doan*

# 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 = $\lceil log_2 N \rceil + 1$
- I/Os per pass = $2N$
- Total I/Os = $2N(\lceil log_2 N \rceil + 1)$

# 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 $\lceil N/B \rceil$ runs
- Pass **1**, **2**, **3**, …:
  - load $B$-$1$ runs and merge them into one run

# General External Sort: Analysis

- # passes = $\left\lceil log_{B-1}\lceil N/B \rceil \right\rceil + 1$
- I/Os per pass = $2N$
- Total I/Os = $2N(\left\lceil log_{B-1}\lceil N/B \rceil \right\rceil + 1)$

# 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 $\lceil N/2B \rceil$ runs
  - can merge $F = \lfloor B/b \rfloor - 1$, where $b$ is block size
  - # passes: $\left\lceil log_F \lceil N/2B \rceil \right\rceil + 1$
  - 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