Lecture 11: External Sorting
Announcements

1. Midterm Review: This Friday!
2. Project Part #2 is out. Implement CLOCK!
3. Midterm Material: Everything up to Buffer management.
4. Today's lecture is no fair game. Do not forget to go over the activities as well!

Our usual Wednesday update...
Lecture 11: External Sorting
What you will learn about in this section

1. External Merge (of sorted files)

2. External Merge - Sort
1. External Merge
Challenge: Merging Big Files with Small Memory

How do we efficiently merge two sorted files when both are much larger than our main memory buffer?
External Merge Algorithm

• **Input**: 2 sorted lists of length $M$ and $N$

• **Output**: 1 sorted list of length $M + N$

• **Required**: At least 3 Buffer Pages

• **IOs**: $2(M+N)$
Key (Simple) Idea

To find an element that is no larger than all elements in two lists, one only needs to compare minimum elements from each list.

If:

\[
A_1 \leq A_2 \leq \cdots \leq A_N \\
B_1 \leq B_2 \leq \cdots \leq B_M
\]

Then:

\[
\text{Min}(A_1, B_1) \leq A_i \\
\text{Min}(A_1, B_1) \leq B_j
\]

for \( i=1 \ldots N \) and \( j=1 \ldots M \)
External Merge Algorithm

Input:
Two sorted files

Output:
One merged sorted file

Disk

Main Memory
Buffer

F_1
1,5
7,11
20,31

F_2
2,22
23,24
25,30

One merged sorted file

Lecture 11 > Section 1 > External Merge
**External Merge Algorithm**

Input: Two sorted files
Output: One *merged* sorted file

Diagram:
- Disk with files $F_1$ and $F_2$
  - $F_1$: 7,11, 20,31
  - $F_2$: 23,24, 25,30
- Main Memory
  - Buffer: 1,5, 2,22
- Disk
- Input: Two sorted files
- Output: One *merged* sorted file
External Merge Algorithm

Input: Two sorted files
Output: One merged sorted file

Disk

Main Memory
Buffer

$F_1$:
- 7,11
- 20,31

$F_2$:
- 23,24
- 25,30

Input:
Two sorted files
Output:
One merged sorted file
External Merge Algorithm

Input: Two sorted files
Output: One merged sorted file

Disk

Main Memory
Buffer

F_1
7,11
20,31

F_2
23,24
25,30

1,2

5
22
External Merge Algorithm

Input: Two sorted files
Output: One merged sorted file

This is all the algorithm “sees”... Which file to load a page from next?
External Merge Algorithm

Input: Two sorted files

Output: One merged sorted file

Disk

Main Memory

Buffer

7,11 20,31
23,24 25,30
1,2

We know that F₂ only contains values ≥ 22... so we should load from F₁!
External Merge Algorithm

Input: Two sorted files
Output: One merged sorted file

Disk

Main Memory

Buffer

F₁

20, 31

23, 24

25, 30

F₂

1, 2

Input:

Output:
External Merge Algorithm

Input:
Two sorted files

F₁
20,31

F₂
23,24
25,30

Output:
One *merged* sorted file

1,2

Disk

Main Memory

Buffer

11
22
5,7
External Merge Algorithm

Input: Two sorted files
Output: One merged sorted file

F<sub>1</sub>: 20, 31
F<sub>2</sub>: 23, 24, 25, 30

Disk

Main Memory
Buffer
11, 22

1, 2, 5, 7
External Merge Algorithm

Input: Two sorted files

F\_1
20,31
F\_2
23,24 25,30

Output: One merged sorted file

1,2 5,7

And so on...
We can merge 2 lists of arbitrary length with only 3 buffer pages.

If lists of size M and N, then

**Cost:** $2(M+N)$ IOs
Each page is read once, written once

With B+1 buffer pages, can merge B lists. How?
2. External Merge Sort
What you will learn about in this section

1. External merge sort (2-way sort)
2. External merge sort on larger files
3. Optimizations for sorting
External Merge Algorithm

• Suppose we want to merge two sorted files both much larger than main memory (i.e. the buffer)

• We can use the external merge algorithm to merge files of arbitrary length in $2*(N+M)$ IO operations with only 3 buffer pages!

Our first example of an “IO aware” algorithm / cost model
Why are Sort Algorithms Important?

• Data requested from DB in sorted order is **extremely common**
  • e.g., find students in increasing GPA order

• Why not just use quicksort in main memory??
  • What about if we need to sort 1TB of data with 1GB of RAM...

A classic problem in computer science!
More reasons to sort...

• Sorting useful for eliminating *duplicate copies* in a collection of records (Why?)

• Sorting is first step in *bulk loading* B+ tree index.

• *Sort-merge* join algorithm involves sorting
Do people care?

http://sortbenchmark.org

Sort benchmark bears his name
External Merge Sort
So how do we sort big files?

1. Split into chunks small enough to sort in memory ("runs")

2. Merge pairs (or groups) of runs using the external merge algorithm

3. Keep merging the resulting runs (each time = a “pass”) until left with one sorted file!
External Merge Sort Algorithm (2-way sort)

Example:
• 3 Buffer pages
• 6-page file

Disk

Main Memory
Buffer

Orange file = unsorted

1. Split into chunks small enough to sort in memory
External Merge Sort Algorithm (2-way sort)

Example:
• 3 Buffer pages
• 6-page file

Orange file = unsorted

1. Split into chunks small enough to sort in memory
External Merge Sort Algorithm (2-way sort)

Example:
- 3 Buffer pages
- 6-page file

1. Split into chunks small enough to sort in memory
External Merge Sort Algorithm (2-way sort)

Example:
- 3 Buffer pages
- 6-page file

1. Split into chunks small enough to sort in memory
External Merge Sort Algorithm (2-way sort)

Example:
• 3 Buffer pages
• 6-page file

Each sorted file is called a run

1. Split into chunks small enough to sort in memory
External Merge Sort Algorithm (2-way sort)

Example:
- 3 Buffer pages
- 6-page file

2. Now just run the external merge algorithm & we’re done!
Calculating IO Cost

For 3 buffer pages, 6 page file:

1. Split into **two 3-page files** and **sort in memory**
   1. \(= 1 \text{ R} + 1 \text{ W} \) for each file = \(2 \times (3 + 3) = 12\) IO operations

2. **Merge** each pair of sorted chunks **using the external merge algorithm**
   1. \(= 2 \times (3 + 3) = 12\) IO operations

3. Total cost = 24 IO
Running External Merge Sort on Larger Files

Assume we still only have 3 buffer pages (Buffer not pictured)
Running External Merge Sort on Larger Files

1. Split into files small enough to sort in buffer...

Assume we still only have 3 buffer pages (Buffer not pictured)
Running External Merge Sort on Larger Files

1. Split into files small enough to sort in buffer... and sort

Assume we still only have 3 buffer pages (Buffer not pictured)

Call each of these sorted files a run
Running External Merge Sort on Larger Files

Assume we still only have 3 buffer pages (Buffer not pictured)

2. Now merge pairs of (sorted) files... the resulting files will be sorted!
Running External Merge Sort on Larger Files

Assume we still only have 3 buffer pages *(Buffer not pictured)*

3. And repeat...

Call each of these steps a **pass**
Running External Merge Sort on Larger Files

4. And repeat!
Simplified 3-page Buffer Version

Assume for simplicity that we split an N-page file into N single-page runs and sort these; then:

• First pass: Merge $\frac{N}{2}$ pairs of runs each of length 1 page

• Second pass: Merge $\frac{N}{4}$ pairs of runs each of length 2 pages

• In general, for N pages, we do $\lceil \log_2 N \rceil$ passes
  • +1 for the initial split & sort

• Each pass involves reading in & writing out all the pages = $2N$ IO

$\Rightarrow 2N \ast (\lceil \log_2 N \rceil + 1)$ total IO cost!
Using B+1 buffer pages to reduce # of passes

Suppose we have B+1 buffer pages now; we can:

1. **Increase length of initial runs.** Sort B+1 at a time!

At the beginning, we can split the N pages into runs of length B+1 and sort these in memory.

**IO Cost:**

Starting with runs of length 1:

\[ 2N([\log_2 N] + 1) \]

Starting with runs of length \( B+1 \):

\[ 2N\left(\left\lfloor \log_2 \frac{N}{B+1} \right\rfloor + 1\right) \]
Using B+1 buffer pages to reduce # of passes

Suppose we have B+1 buffer pages now; we can:

2. Perform a B-way merge.
On each pass, we can merge groups of \( B \) runs at a time (vs. merging pairs of runs)!

IO Cost:

\[
2N([\log_2 N] + 1) \quad \rightarrow \quad 2N\left(\left\lfloor \log_2 \frac{N}{B+1} \right\rfloor + 1\right) \quad \rightarrow \quad 2N\left(\left\lfloor \log_B \frac{N}{B+1} \right\rfloor + 1\right)
\]

Starting with runs of length 1
Starting with runs of length \( B+1 \)
Performing \( B \)-way merges
Repacking
Repacking for even longer initial runs

• With B+1 buffer pages, we can now start with **B+1-length initial runs** (and use **B-way merges**) to get $2N \left( \left\lceil \log_B \frac{N}{B+1} \right\rceil + 1 \right)$ IO cost...

• Can we reduce this cost more by getting even longer initial runs?

• Use **repacking**- produce longer initial runs by “merging” in buffer as we sort at initial stage
Repacking Example: 3 page buffer

• Start with unsorted single input file, and load 2 pages
Repacking Example: 3 page buffer

- Take the minimum two values, and put in output page

Also keep track of max (last) value in current run...
Repacking Example: 3 page buffer

• Next, *repack*
Repacking Example: 3 page buffer

- Next, **repack**, then load another page and continue!
Repacking Example: 3 page buffer

- Now, however, **the smallest values are less than the largest (last) in the sorted run**...
Repacking Example: 3 page buffer

• Now, however, **the smallest values are less than the largest (last) in the sorted run**...
Repacking Example: 3 page buffer

- Now, however, **the smallest values are less than the largest (last) in the sorted run**...
Repacking Example: 3 page buffer

- Now, however, **the smallest values are less than the largest (last) in the sorted run**...
Repacking Example: 3 page buffer

- Once *all buffer pages have a frozen value*, or input file is empty, start new run with the frozen values
Repacking Example: 3 page buffer

- Once all buffer pages have a frozen value, or input file is empty, start new run with the frozen values
Repacking

• Note that, for buffer with $B+1$ pages:
  • If input file is sorted $\rightarrow$ nothing is frozen $\rightarrow$ we get a **single** run!
  • If input file is reverse sorted (worst case) $\rightarrow$ everything is frozen $\rightarrow$ we get runs of length $B+1$

• In general, with repacking we do **no worse** than without it!

• What if the file is already sorted?

• Engineer’s approximation: runs will have $\sim 2(B+1)$ length

$$\sim 2N\left(\log_B \frac{N}{2(B+1)}\right) + 1$$
Summary

• Basics of IO and buffer management.

• We introduced the IO cost model using sorting.
  • Saw how to do merges with few IOs,
  • Works better than main-memory sort algorithms.

• Described a few optimizations for sorting