Why Sort?

- select ... order by
  - e.g., find students in increasing gpa order
- bulk loading B+ tree index.
- duplicate elimination (select distinct)
- select ... group by
- Sort-merge join algorithm involves sorting.
Outline

- two-way merge sort
- external merge sort
- fine-tunings
- B+ trees for sorting

2-Way Sort: Requires 3 Buffers

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

Two-Way External Merge Sort

- Each pass we read + write each page in file.
Two-Way External Merge Sort

- Each pass we read + write each page in file.

Input file

<table>
<thead>
<tr>
<th>1-page runs</th>
<th>2-page runs</th>
</tr>
</thead>
<tbody>
<tr>
<td>PASS 0</td>
<td>PASS 1</td>
</tr>
<tr>
<td>3,4</td>
<td>5,6</td>
</tr>
<tr>
<td>6,2</td>
<td>2,6</td>
</tr>
<tr>
<td>9,4</td>
<td>4,9</td>
</tr>
<tr>
<td>8,7</td>
<td>7,8</td>
</tr>
<tr>
<td>5,6</td>
<td>1,3</td>
</tr>
<tr>
<td>3,1</td>
<td>2</td>
</tr>
</tbody>
</table>

1-page runs

<table>
<thead>
<tr>
<th>2-page runs</th>
</tr>
</thead>
<tbody>
<tr>
<td>PASS 2</td>
</tr>
<tr>
<td>2,3</td>
</tr>
<tr>
<td>4,6</td>
</tr>
<tr>
<td>4,7</td>
</tr>
<tr>
<td>8,9</td>
</tr>
<tr>
<td>1,3</td>
</tr>
<tr>
<td>5,6</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

2-page runs

<table>
<thead>
<tr>
<th>4-page runs</th>
</tr>
</thead>
<tbody>
<tr>
<td>PASS 3</td>
</tr>
<tr>
<td>2,3</td>
</tr>
<tr>
<td>4,4</td>
</tr>
<tr>
<td>6,7</td>
</tr>
<tr>
<td>8,9</td>
</tr>
<tr>
<td>1,2</td>
</tr>
<tr>
<td>3,5</td>
</tr>
<tr>
<td>6</td>
</tr>
</tbody>
</table>

4-page runs

<table>
<thead>
<tr>
<th>8-page runs</th>
</tr>
</thead>
<tbody>
<tr>
<td>PASS 0</td>
</tr>
<tr>
<td>1,2</td>
</tr>
<tr>
<td>2,3</td>
</tr>
<tr>
<td>3,4</td>
</tr>
<tr>
<td>4,5</td>
</tr>
<tr>
<td>6,6</td>
</tr>
<tr>
<td>7,8</td>
</tr>
</tbody>
</table>

8-page runs
Two-Way External Merge Sort

- Each pass we read + write each page in file.
- N pages in the file $\Rightarrow$ $N = \lceil \log_2 N \rceil + 1$
- So total cost is: $2N \left( \lceil \log_2 N \rceil + 1 \right)$

Idea: Divide and conquer: sort subfiles and merge

Outline
- two-way merge sort
- external merge sort
- fine-tunings
- B+ trees for sorting

External merge sort

B > 3 buffers
- Q1: how to sort?
- Q2: cost?
General External Merge Sort

B > 3 buffer pages. How to sort a file with N pages?

- Pass 0: use B buffer pages. Produce \( \frac{N}{B} \) sorted runs of \( B \) pages each.
- Pass 1, 2, ..., etc.: merge \( B-1 \) runs.

External merge sort

B > 3 buffers

- Q1: how to sort?
- Q2: cost?
Sorting

– create sorted runs of size B (how many?)
– merge them (how?)

B

... 

I

I

....

Sorting

– create sorted runs of size B
– merge first B-1 runs into a sorted run of
  (B-1) *B, ...

B

... 

I

I

....

Sorting

– How many steps we need to do?
  ‘i’, where  \( B \times (B-1)^i > N \)
– How many reads/writes per step? N+N
Cost of External Merge Sort

- Number of passes: \(1 + \lceil \log_{B} \left( \frac{N}{B} \right) \rceil\)
- Cost = \(2N \times \text{(number of passes)}\)

Cost of External Merge Sort

- E.g., with 5 buffer pages, to sort 108 page file:
  - Pass 0: \(\lceil \frac{108}{5} \rceil = 22\) sorted runs of 5 pages each (last run is only 3 pages)
  - Pass 1: \(\lceil \frac{22}{4} \rceil = 6\) 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: \(\log_{4} 22 \approx 3 \ldots + 1 \rightarrow 4\) passes

Number of Passes of External Sort

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

<table>
<thead>
<tr>
<th>(N)</th>
<th>(B=3)</th>
<th>(B=5)</th>
<th>(B=9)</th>
<th>(B=129)</th>
<th>(B=257)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1,000</td>
<td>10</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>10,000</td>
<td>13</td>
<td>7</td>
<td>5</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>100,000</td>
<td>17</td>
<td>9</td>
<td>6</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1,000,000</td>
<td>20</td>
<td>10</td>
<td>7</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>10,000,000</td>
<td>23</td>
<td>12</td>
<td>8</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>100,000,000</td>
<td>26</td>
<td>14</td>
<td>9</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>30</td>
<td>15</td>
<td>10</td>
<td>8</td>
<td>5</td>
</tr>
</tbody>
</table>
Outline

- two-way merge sort
- external merge sort
  - fine-tunings
  - B+ trees for sorting

Internal Sort Algorithm

- Quicksort is a fast way to sort in memory.
- But: we get B buffers, and produce 1 run of length B.
- Can we produce longer runs than that?
Internal Sort Algorithm

- Quicksort is a fast way to sort in memory.
- But: we get B buffers, and produce 1 run of length B.
- Can we produce longer runs than that?

Alternative: “tournament sort” (a.k.a. “heapsort”, “replacement selection”)
- Produces runs of length ~ 2*B
- Clever, but not implemented, for subtle reasons: tricky memory management on variable length records

Reminder: Heapsort

pick smallest, write to output buffer:
Heapsort:

pick smallest, write to output buffer:

Heapsort:

get next key; put at top and 'sink' it

Heapsort:

get next key; put at top and 'sink' it
Heapsort:

- get next key; put at top and 'sink' it

Heapsort:

- When done, pick top (= smallest) and output it, if 'legal' (i.e., >=10 in our example)
  - This way, we can keep on reading new key values (beyond the B ones of quicksort)

Outline

- two-way merge sort
- external merge sort
- fine-tunings
  - which internal sort for Phase 0?
  - blocked I/O
- B+ trees for sorting
Blocked I/O & double-buffering

- So far, we assumed random disk access
- Cost changes, if we consider that runs are written (and read) sequentially
- What could we do to exploit it?

- A1: Blocked I/O (exchange a few r.d.a for several sequential ones) \([\text{use bigger pages}]\)
- A2: double-buffering \([\text{mask I/O delays with prefetching}]\)

A1: blocked I/O

- Normally, ‘B’ buffers of size (say) 1K
A1: blocked I/O

- Normally, ‘B’ buffers of size (say) 1K
- INSTEAD: \( b \) buffers, of size ‘\( b \)’ Kilobytes

Pros?
- Fewer random d.a. (because some of them -> sequential)
- Cons? Smaller fanout -> maybe more passes
Blocked I/O & double-buffering

- So far, we assumed random disk access
- Cost changes, if we consider that runs are written (and read) sequentially
- What could we do to exploit it?
- A1: Blocked I/O (exchange a few r.d.a for several sequential ones) [use bigger pages]
- A2: double-buffering [mask I/O delays with prefetching]

A2: Double buffering

- Normally, when, say ‘INPUT1’ is exhausted
  - We issue a ‘read’ request and
  - We wait …

\[\text{INPUT } 1 \quad \cdots \quad \text{OUTPUT} \]

\[\text{INPUT } 2 \quad \cdots \]

\[\text{INPUT } \ldots \]

\[\text{B Main memory buffers} \quad \text{Disk} \]

\[\text{Disk} \quad \text{OUTPUT} \]

\[\text{INPUT } 1 \quad \cdots \quad \text{INPUT } \ldots \]

\[\text{INPUT } 1' \quad \cdots \quad \text{OUTPUT' } \]

\[\text{Disk} \quad \text{Disk} \]

\[\text{INPUT } 1 \quad \text{INPUT } k \quad \text{INPUT } 2 \quad \text{INPUT } 1' \quad \text{INPUT } 2' \quad \text{INPUT } k' \]

\[\text{b block size} \]

\[\text{B main memory buffers, k-way merge} \]

A2: Double Buffering

- \(w/\) double buffering, we \textit{prefetch} INPUT1’ into ‘shadow block’
  - When INPUT1 is exhausted, we issue a ‘read’,
  - BUT we proceed with INPUT1’
A2: Double Buffering

- Potentially, more passes; in practice, most files still sorted in 2-3 passes.

Outline

- two-way merge sort
- external merge sort
- fine-tunings
  - B+ trees for sorting

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
  - B+ tree is not clustered
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)
- Data Records
- Index (Directs search)
- Data Entries ("Sequence set")

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!
- Data Records
- Index (Directs search)
- Data Entries ("Sequence set")
### External Sorting vs. Unclustered Index

<table>
<thead>
<tr>
<th>N</th>
<th>Sorting</th>
<th>p=1</th>
<th>p=10</th>
<th>p=100</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>200</td>
<td>100</td>
<td>1,000</td>
<td>10,000</td>
</tr>
<tr>
<td>1,000</td>
<td>2,000</td>
<td>1,000</td>
<td>10,000</td>
<td>100,000</td>
</tr>
<tr>
<td>10,000</td>
<td>40,000</td>
<td>10,000</td>
<td>100,000</td>
<td>1,000,000</td>
</tr>
<tr>
<td>100,000</td>
<td>600,000</td>
<td>100,000</td>
<td>1,000,000</td>
<td>10,000,000</td>
</tr>
<tr>
<td>1,000,000</td>
<td>8,000,000</td>
<td>1,000,000</td>
<td>10,000,000</td>
<td>100,000,000</td>
</tr>
<tr>
<td>10,000,000</td>
<td>80,000,000</td>
<td>10,000,000</td>
<td>100,000,000</td>
<td>1,000,000,000</td>
</tr>
</tbody>
</table>

N: # pages
p: # of records per page
B=1,000 and block size=32 for sorting
p=100 is the more realistic value.

### 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.
- Clustered B+ tree is good for sorting; unclustered tree is usually very bad.