CS 4604: Introduction to Database Management Systems

B. Aditya Prakash
Guest Lecture by
Shamimul Hasan and Sorour Amiri
Lecture #9: Hashing contd., and Sorting
Announcement

- Extra office hours till midterm
  - Check Piazza post

- Next class:
  - A bit of query processing
  - And quick review for midterm
This lecture

- Continuation of Hashing
  - Linear Hashing

- Then External Memory algorithms for Sorting
Linear hashing - overview

- Motivation
- main idea
- search algo
- insertion/split algo
- deletion
Linear hashing

- Motivation: ext. hashing needs directory etc etc; which doubles (ouch!)
- Q: can we do something simpler, with smoother growth?
Linear hashing

- Motivation: ext. hashing needs directory etc etc; which doubles (ouch!)
- Q: can we do something simpler, with smoother growth?
- A: split buckets from left to right, regardless of which one overflowed (‘crazy’, but it works well!) - Eg.:
Linear hashing

Initially: $h(x) = x \mod N$  \hspace{1em} (N=4 here)

Assume capacity: 3 records / bucket

Insert key ‘17’

<table>
<thead>
<tr>
<th>bucket-id</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>8</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>13</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11</td>
</tr>
</tbody>
</table>
Initially: $h(x) = x \mod N$ (N=4 here)

overflow of bucket#1
Linear hashing

Initially: $h(x) = x \mod N$  \hspace{1cm} (N=4 here)

overflow of bucket#1

Split #0, anyway!!!
Initially: \( h(x) = x \mod N \) (N=4 here)

Split #0, anyway!!!

Q: But, how?
A: use two h.f.: $h_0(x) = x \mod N$

$h_1(x) = x \mod (2*N)$
Linear hashing - after split:

A: use two h.f.:  
\[ h_0(x) = x \mod N \]
\[ h_1(x) = x \mod (2*N) \]
A: use two h.f.: \( h_0(x) = x \mod N \)
\[ h_1(x) = x \mod (2*N) \]
Linear hashing - after split:

A: use two h.f.:  
\[ h_0(x) = x \mod N \]
\[ h_1(x) = x \mod (2*N) \]
Linear hashing - searching?

\[ h_0(x) = x \mod N \quad \text{(for the un-split buckets)} \]

\[ h_1(x) = x \mod (2*N) \quad \text{(for the split ones)} \]
Linear hashing - searching?

Q1: find key ‘6’?
Q2: find key ‘4’?
Q3: key ‘8’?

<table>
<thead>
<tr>
<th>Bucket Id</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8</td>
<td>5</td>
<td>9</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td></td>
<td></td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>overflow</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>split ptr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Linear hashing - searching?

Algo to find key ‘k’:

• compute $b = h_0(k)$;
  • if $b <$split-ptr, compute $b = h_1(k)$
• search bucket $b$
Linear hashing - insertion?

Algo: insert key ‘k’

• compute appropriate bucket ‘b’

• if the **overflow criterion** is true
  • split the bucket of ‘split-ptr’
  • split-ptr ++ (*)
Linear hashing - insertion?

- notice: overflow criterion is up to us!!
- Q: suggestions?
Linear hashing - insertion?

- notice: overflow criterion is up to us!!
- Q: suggestions?
- A1: space utilization $\geq u\text{-max}$
Linear hashing - insertion?

- notice: overflow criterion is up to us!!
- Q: suggestions?
- A1: space utilization > u-max
- A2: avg length of ovf chains > max-len
- A3: ....
Linear hashing - insertion?

Algo: insert key ‘k’

• compute appropriate bucket ‘b’

• if the **overflow criterion** is true
  • split the bucket of ‘split-ptr’
  • split-ptr ++ (*)

what if we reach the right edge??
Linear hashing - split now?

\[ h_0(x) = x \mod N \quad \text{(for the un-split buckets)} \]
\[ h_1(x) = x \mod (2 \times N) \quad \text{for the split ones} \]
Linear hashing - split now?

$h_0(x) = x \mod N$  (for the un-split buckets)  $h_1(x) = x \mod (2*N)$  (for the splitted ones)
Linear hashing - split now?

h₀(x) = x mod N  (for the un-split buckets)  h₁(x) = x mod (2*N)  (for the split ones)

split ptr

0  1  2  3  4  5  6  7
Linear hashing - split now?

\[ h_0(x) = x \mod N \quad \text{(for the un-split buckets)} \]
\[ h_1(x) = x \mod (2*N) \quad \text{(for the splitted ones)} \]
Linear hashing - split now?

this state is called ‘full expansion’
In general, at any point of time, we have at most two h.f. active, of the form:

- \( h_n(x) = x \mod (N \times 2^n) \)
- \( h_{n+1}(x) = x \mod (N \times 2^{n+1}) \)

(after a full expansion, we have only one h.f.)
Linear hashing - deletion?

- reverse of insertion:
Linear hashing - deletion?

- reverse of insertion:
- if the underflow criterion is met
  – contract!
Linear hashing - how to contract?

\[ h_0(x) = \text{mod } N \quad \text{(for the un-split buckets)} \]
\[ h_1(x) = \text{mod } (2*N) \quad \text{(for the splitted ones)} \]
**Linear hashing - how to contract?**

\[ h_0(x) = \text{mod } N \quad \text{(for the un-split buckets)} \]

\[ h_1(x) = \text{mod } (2 \times N) \quad \text{(for the split buckets)} \]
Hashing - pros?
Hashing - pros?

- Speed,
  - on exact match queries
  - on the average
B(+)-trees - pros?
B(+)‐trees - pros?

- Speed on search:
  - exact match queries, worst case
  - range queries
  - nearest-neighbor queries
- Speed on insertion + deletion
- Smooth growing and shrinking (no re-org)
Conclusions

- B-trees and variants: in all DBMSs
- hash indices: in some
  - (but hashing in useful for joins...: will see in later lecture)
SORTING
Why Sort?
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.
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.
Two-Way External Merge Sort

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

**Idea:** Divide and conquer: sort subfiles and merge
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?
General External Merge Sort

- Pass 0: use $B$ buffer pages. Produce $\left\lceil \frac{N}{B} \right\rceil$ sorted runs of $B$ pages each.
- Pass 1, 2, ..., etc.: merge $B-1$ runs.
Sorting

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

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

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

- Number of passes: \(1 + \lceil \log_{B-1} \left[ N / B \right] \rceil\)
- Cost = \(2N \times \text{(\# of passes)}\)
Cost of External Merge Sort

- E.g., with 5 buffer pages, to sort 108 page file:
  - Pass 0: \(\left\lceil \frac{108}{5} \right\rceil = 22\) sorted runs of 5 pages each (last run is only 3 pages)
  - Pass 1: \(\left\lceil \frac{22}{4} \right\rceil = 6\) 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: \(\lceil \log_4 22 \rceil = 3 \ldots + 1 \rightarrow 4\) 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=17</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>
<td>1</td>
</tr>
<tr>
<td>1,000</td>
<td>10</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</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>
<td>2</td>
</tr>
<tr>
<td>100,000</td>
<td>17</td>
<td>9</td>
<td>6</td>
<td>5</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>
<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>
<td>3</td>
</tr>
<tr>
<td>100,000,000</td>
<td>26</td>
<td>14</td>
<td>9</td>
<td>7</td>
<td>4</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>
<td>4</td>
</tr>
</tbody>
</table>
Internal Sort Algorithm

- Quicksort is a fast way to sort in memory.
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?
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)
  - A2: double-buffering
Double Buffering

- To reduce wait time for I/O request to complete, can *prefetch* into `shadow block`
  
  - Potentially, more passes; in practice, most files still sorted in 2-3 passes.
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)

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!*

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

### Table

<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>

**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.