CS 4604: Introduction to Database Management Systems

Hashing and Sorting

Virginia Tech CS 4604 Sprint 2021
Instructor: Yinlin Chen
Today’s Topics

• Hashing
  – Static Hashing
  – Extendible Hashing
  – Linear Hashing

• Sorting
  – Two-way merge sort
  – External merge sort
  – Fine-tunings
  – B+ trees for sorting
Hashing

• Many times, we don’t require order
  – Problem: “find EMP record with ssn=123”

• Static Hashing

• Dynamic hashing techniques:
  – Extendible Hashing
  – Linear Hashing
(Static) Hashing

- Each hash bucket has one **primary** page and possibly additional **overflow** pages
- Page holds many records
- hash function: $h(key) = \text{slot-id}$
(Static) Hashing

• Insert:
  1. hash function: \( h(\text{key}) \) find the correct bucket
  2.1 There is a space, insert a data there
  2.2 There is no space
     
     step 1. allocate a new overflow page and then insert a data there
     step 2. add that page to the overflow chain of the bucket
(Static) Hashing

- Delete:
  1. hash function: \( h(key) \) find the correct bucket
  2. Locate the data then remove it
  2.1 Last item in an overflow page? overflow page
Cost of (Static) Hashing

• Search: One disk I/O
• Insert and Delete: Two disk I/O
• Many overflow pages → poor performance
Problem with static hashing

- The number of bucket is fixed
- Underflow:
  - A lot of space is wasted (underutilization)
- Overflow:
  - Poor performance
- Better solution: **Dynamic hashing**
Extendible hashing

- Idea:
  - Use a directory of pointers to buckets
  - Double the directory
  - Double the size of the number of buckets
  - Splitting the bucket that overflowed
Extendible hashing

GLOBAL DEPTH

LOCAL DEPTH

DIRECTORY

DATA PAGES

Search 5: 101
Extendible hashing

Insert 13: 1101
Extendible hashing

GLOBAL DEPTH

DIRECTORY

LOCAL DEPTH

DATA PAGES

Insert 20: 10100

Full!
Extendible hashing

Insert 20: 10100

Step 1: **split** the bucket
Step 2: redistribute the contents by last *three* bits of h(r)

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>1100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>100000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>10000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>10100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extendible hashing

Insert 20: 10100

Step 3: **double** the directory
Extendible hashing

Insert 9: 1001

Full!
Extendible hashing

Insert 9: 1001

Step 1: split the bucket
Step 2: redistribute the contents by last three bits of h(r)
Step 3: no need to double the directory

How to know if we need to double a directory?
• Global and local depth are the same (Double!)
Cost of Extendible Hashing

• Search: One disk I/O or (worse) two I/Os (and rare)
• Insert and Delete: Two disk I/O
• Better performance
• Special case: collisions, or data entries with the same hash value.
  – Need overflow pages
Linear hashing

- It does not require a directory
- Deal naturally with collisions
- Still need overflow pages and chains
- Utilizes a family of hash functions: $h_0, h_1, h_2, \ldots$
  - $h_0$: $M$ buckets
  - $h_1$: $2M$ buckets
  - $h_2$: $4M$ buckets
  - $\ldots$
Linear hashing

- Number of $N$ buckets ($N = 4$)
- $d_0$ is the number of bits needed to represent $N$ ($d_0 = 2$)
- $h_0$ is $h \mod 4$: 0 to 3
- $d_1 = d_0 + 1 = 3$
- $h_1$ is $h \mod (2 \times 4)$: 0 to 7
Linear hashing

- $h(x) = x \mod N$ ($N = 4$)
- Assume capacity: 4 records per bucket
- Insert key 43 (101011)

Bucket ID

<table>
<thead>
<tr>
<th>Level = 0</th>
<th>Next = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
</tr>
<tr>
<td>32 44</td>
<td>25 9</td>
</tr>
<tr>
<td>36</td>
<td>5</td>
</tr>
</tbody>
</table>

Full!
Linear hashing – split first

• $h(x) = x \mod N$ ($N = 4$)
• Assume capacity: 4 records per bucket
• Insert key 43 (101011)

Next = 1

<table>
<thead>
<tr>
<th>Bucket ID</th>
<th>Level = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>32 5</td>
</tr>
<tr>
<td>01</td>
<td>25 9</td>
</tr>
<tr>
<td>10</td>
<td>14 18</td>
</tr>
<tr>
<td>11</td>
<td>31 35</td>
</tr>
<tr>
<td>100</td>
<td>44 36</td>
</tr>
</tbody>
</table>
Linear hashing – after split

- \( h_0(x) = x \mod N \) (\( N = 4 \))
- \( h_1(x) = x \mod (2*N) \)
- Insert key 43 (101011)

Bucket ID

000 01 10 11 100

Level = 0

32 25 9 14 18 31 35 44 36

Next = 1

Overflow page

Insert key 43 (101011)
Linear hashing

- $h_0(x) = x \text{ mod } N$ ($N = 4$)
- $h_1(x) = x \text{ mod } (2 \times N)$
- Insert key $37$ (100101)

Bucket ID:

- Level $= 0$
  - 000: $32$
  - 01: $25, 9, 5, 37$
  - 10: $14, 18, 10, 30$
  - 11: $31, 35, 7, 11$
  - 100: $44, 36$

Next $= 1$

Overflow page:

- 43
Linear hashing

- $h_0(x) = x \mod N$ (N = 4)
- $h_1(x) = x \mod (2 \times N)$
- Insert key 29 (11101)

Bucket ID

Next = 1

Level = 0

Bucket IDs:

- 000: 32, 5, 37
- 01: Full
- 10: 14, 18
- 11: 31, 35
- 100: 44, 36

Overflow page: 43
Linear hashing

- $h_0(x) = x \mod N$ (N = 4)
- $h_1(x) = x \mod (2 \times N)$
- Insert key 29 (11101)
Linear hashing

- $h_0(x) = x \mod N$ (N = 4)
- $h_1(x) = x \mod (2N)$
- Insert key 22 (10110)
Linear hashing

- $h_0(x) = x \mod N$ (N = 4)
- $h_1(x) = x \mod (2 \times N)$
- Insert key 22 (10110)

Bucket ID

<table>
<thead>
<tr>
<th>Level = 0</th>
<th>Bucket ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>32</td>
</tr>
<tr>
<td>001</td>
<td>25 9</td>
</tr>
<tr>
<td>010</td>
<td>18 10</td>
</tr>
<tr>
<td>011</td>
<td>31 35 7 11</td>
</tr>
<tr>
<td>100</td>
<td>44 36</td>
</tr>
<tr>
<td>101</td>
<td>5 37 29</td>
</tr>
<tr>
<td>110</td>
<td>14 30 22</td>
</tr>
</tbody>
</table>

Next = 3

overflow page

VT TECH
Linear hashing

- $h_0(x) = x \mod N$ (N = 4)
- $h_1(x) = x \mod (2 \times N)$
- Insert key 66 (1000010) and 34 (100010)

Next = 3

Bucket ID

<table>
<thead>
<tr>
<th>Level = 0</th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>11</th>
<th>100</th>
<th>101</th>
<th>110</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>32</td>
<td>25 9</td>
<td>18 10</td>
<td>66 34</td>
<td>31 35</td>
<td>44 36</td>
<td>5 37</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7 11</td>
<td></td>
<td>14 30</td>
</tr>
</tbody>
</table>

overflow page

43
Linear hashing

- $h_0(x) = x \text{ mod } N$ ($N = 4$)
- $h_1(x) = x \text{ mod } (2*N)$
- Insert key 50 ($110010$)

Bucket ID

<table>
<thead>
<tr>
<th>Level = 0</th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>11</th>
<th>100</th>
<th>101</th>
<th>110</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>32</td>
<td>25</td>
<td>9</td>
<td>18</td>
<td>10</td>
<td>66</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td></td>
<td>31</td>
<td>35</td>
<td>44</td>
<td>36</td>
<td>5</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
<td>11</td>
<td>44</td>
<td>36</td>
<td>5</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>14</td>
<td>30</td>
<td>14</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>22</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Next = 3

Full!

overflow page

VT

Virginia Tech
Linear hashing

- $h_0(x) = x \mod N$ (N = 4)
- $h_1(x) = x \mod (2 \times N)$
- Insert key 50 (110010)

Bucket ID

Next = 0

Level = 1

overflow page
Cost of Linear Hashing

- Search: One disk I/O or more when having overflow pages (average 1.2 I/Os)
- Insert and Delete: Two disk I/O (unless a split is triggered)
- Better performance
Example: Linear hashing

- $h(x) = x \mod N$ (N = 4)
- Assume capacity: 3 records per bucket
- Insert key 17 (10001)
Example: Linear hashing – after split

- \( h_0(x) = x \mod N \) (\( N = 4 \))
- \( h_1(x) = x \mod (2\times N) \)
- Insert key 17 (10001)
Linear hashing – searching

- $h_0(x) = x \mod N$ (for the un-split buckets)
- $h_1(x) = x \mod (2 \cdot N)$ (for the split ones)
- Q1: find key ‘6’?  Q2: find key ‘4’?  Q3: key ‘8’?

Bucket ID: 000 01 10 11 100

```
<table>
<thead>
<tr>
<th></th>
<th>8</th>
<th>5 9 13</th>
<th>6</th>
<th>7 11</th>
<th>4</th>
</tr>
</thead>
</table>
```

Next = 1

overflow
Hashing Summary

- B-trees and variants: in all DBMSs
- Hash indices: in some DBMSs
  - Hashing is useful for joins
- Hashing performs well on exact match queries
- B+ tree performs well on:
  - Search:
    - exact match queries
    - range queries
    - nearest-neighbor queries
  - Insertion and deletion
  - Smooth growing and shrinking
Sorting

• Two-way merge sort
• External merge sort
• Fine-tunings
• B+ trees for sorting
Why Sort?

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

- Overview: break file into smaller subfiles, sort each subfile, and merge
- Utilizes only three (buffer) pages of main memory
- Pass 0: Read a page, sort it, write out
  - only one buffer page is used (a sorted run)
  - In-memory sorting technique. E.g., Quicksort
- Pass 1, 2, 3, ...k: Requires 3 buffer pages
  - merge pairs of runs into runs twice as long
  - three buffer pages used.
- Cost: $2N(\lceil \log_2 N \rceil + 1)$ I/Os
Two-Way Merge Sort

- Cost: $2N(\lceil \log_2 N \rceil + 1)$ I/Os
- $N = 8, \ 2 \times 8 \times (3+1) = 64$ I/Os

Binary uses base 2:

- $2^0 = 1 \quad \log_2(1) = 0$
- $2^1 = 2 \quad \log_2(2) = 1$
- $2^2 = 4 \quad \log_2(4) = 2$
- $2^3 = 8 \quad \log_2(8) = 3$
- $2^4 = 16 \quad \log_2(16) = 4$
- $2^5 = 32 \quad \log_2(32) = 5$
Two-Way Merge Sort

• Each pass we read and write each page in file
Two-Way Merge Sort

- Each pass we read and write each page in file
Two-Way Merge Sort

- Each pass we read and write each page in file
Two-Way Merge Sort

- Each pass we read and write each page in file
Two-Way Merge Sort

- Each pass we read and write each page in file
- $N$ pages in the file: $\lceil \log_2 N \rceil + 1$
- Total cost: $2N(\lfloor \log_2 N \rfloor + 1)$ I/Os
- *Divide and conquer:* sort subfiles and merge
External Merge Sort

- Two-Way Merge Sort: We have more than three buffer pages available in main memory, we just use three. (underutilize)
External Merge Sort

- A large file with $N$ pages needs to be sorted
- $B$ buffer pages in memory
- 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
External merge sort

- Number of passes:

\[ 1 + \left\lceil \log_{B-1} \left\lceil \frac{N}{B} \right\rceil \right\rceil = 1 + \left\lceil \log_{B-1} N1 \right\rceil, \quad N1 = \left\lfloor \frac{N}{B} \right\rfloor \]

- Cost = \(2N \times \) (# of passes)
Cost of External Merge Sort

- Example: we have 5 buffer pages and want to sort a file with 108 pages
- Pass 0: \( \left\lceil \frac{108}{5} \right\rceil = 22 \) sorted runs of 5 pages each
- Pass 1: \( \left\lceil \frac{22}{4} \right\rceil = 6 \) sorted runs of 20 pages
- Pass 2: \( \left\lfloor \frac{6}{4} \right\rfloor = 2 \) sorted runs, one run with 80 pages and one run with 28 pages
- Pass 3: Sorted file of 108 pages
- Formula check: \( \left\lfloor \log_4 22 \right\rfloor = 3 \ldots + 1 \rightarrow 4 \) passes
Cost of External Merge Sort

- Each pass we read and write 108 pages
- Total cost: $2 \times 108 \times 4 = 864$ I/Os
- $N_1 = \left\lceil \frac{N}{B} \right\rceil = \left\lceil \frac{108}{5} \right\rceil = 22$
- $B = 5$
- $2N \times (1 + \left\lceil \log_{B-1} N_1 \right\rceil) = 2 \times 108 \times 4$
## 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>
Memory Requirement for External Sorting

• How big of a table can we sort in two passes?
  – Each “sorted run” after Phase 0 is of size $B$
  – Can merge up to $B-1$ sorted runs in Phase 1

• Answer: $B(B-1)$.
  – Sort $N$ pages of data in about $B = \sqrt{N}$ space
Cost Metric

• We assumed random disk access (# of page I/Os)
• Blocked I/O: a single request to read (or write) sequentially
• Also, double buffering: Keep the CPU busy while an I/O op is in progress
## Blocked I/O

- \( \left\lceil \frac{B-b}{b} \right\rceil \) runs
- 10 buffer pages:
  - 9 runs (one buffer blocks)
  - 4 runs (two buffer blocks)

<table>
<thead>
<tr>
<th>( N )</th>
<th>( B = 1000 )</th>
<th>( B = 5000 )</th>
<th>( B = 10,000 )</th>
<th>( B = 50,000 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1000</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10,000</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>100,000</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1,000,000</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>10,000,000</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>100,000,000</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Number of Passes of External Merge Sort with Block Size \( b = 32 \)
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

• Quicksort is a fast way to sort in memory
• 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 page
  - Use alternative 1: Actual data record (with key value \( k \))

Always better than external sorting!
Unclustered B+ Tree Used for Sorting

- Use alternative (2) for data entries \(<k, \text{rid of matching data record}>\)
- Each data entry contains \(\text{rid}\) of a data record. In general, one I/O per data record!
### 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>

*p*: # of records per page

*B*=1,000 and block size=32 for sorting

*p*=100 is the more realistic value.
Sorting 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 B+ tree is usually very bad
Reading and Next Class

- Hashing and Sorting: Ch 11, Ch 13
- Next: Query Processing: Ch 12, Ch 14