Hash-Based Indexes

Chapter 11
Sina Meraji
Announcement

• Project page is updated, check course webpage
  – [http://www.teach.cs.toronto.edu/~csc443h/fall/project.shtml](http://www.teach.cs.toronto.edu/~csc443h/fall/project.shtml)

• Pick a topic by Oct 6th
  – Markus
  – Here: [https://docs.google.com/spreadsheets/d/11WRAncin_1L0-Vyp3gQ1PzzehSCBEQztEaVwfg1oHg/edit#gid=0](https://docs.google.com/spreadsheets/d/11WRAncin_1L0-Vyp3gQ1PzzehSCBEQztEaVwfg1oHg/edit#gid=0)
  – First come first serve

• Want your own topic?
  – Write to me by Friday Sept 29th
Introduction

• As for any index, 3 alternatives for data entries $k^*$:
  – Data record with key value $k$
  – $<k, \text{rid of data record with search key value } k>$
  – $<k, \text{list of rids of data records with search key } k>$
  – Choice orthogonal to the indexing technique

• **Hash-based** indexes are best for *equality selections*. **Cannot** support range searches.

• Static and dynamic hashing techniques exist;
Static Hashing

• # primary pages fixed, allocated sequentially, never de-allocated; overflow pages if needed.

• $h(k) \mod N$ = bucket to which data entry with key $k$ belongs. ($N =$ # of buckets)
Static Hashing (Contd.)

• Buckets contain *data entries*.
• Hash fn works on *search key* field of record \( r \). Must distribute values over range 0 ... M-1.
  – \( h(key) = (a \times key + b) \) usually works well.
  – \( a \) and \( b \) are constants; lots known about how to tune \( h \).
• **Long overflow chains** can develop and degrade performance.
  – *Extendible* and *Linear Hashing*: Dynamic techniques to fix this problem.
Extendible Hashing

• Situation: Bucket (primary page) becomes full. Why not re-organize file by doubling # of buckets?
  – Reading and writing all pages is expensive!
  – **Idea:** Use directory of pointers to buckets, double # of buckets by doubling the directory, splitting just the bucket that overflowed!
  – Directory much smaller than file, so doubling it is much cheaper. Only one page of data entries is split. *No overflow page!*
  – Trick lies in how hash function is adjusted!
Example

- Directory is array of size 4.
- To find bucket for \( r \), take last `global depth` # bits of \( h(r) \); we denote \( r \) by \( h(r) \).
  - If \( h(r) = 5 = \) binary 101, it is in bucket pointed to by 01.

- **Insert**: If bucket is full, *split* it (allocate new page, re-distribute).
- *If necessary*, double the directory. (As we will see, splitting a bucket does not always require doubling; we can tell by comparing *global depth* with *local depth* for the split bucket.)
Insert $h(r)=20$ (Causes Doubling)
Points to Note

• 20 = binary 10100. Last 2 bits (00) tell us \( r \) belongs in A or A2. Last 3 bits needed to tell which.
  – *Global depth of directory*: Max # of bits needed to tell which bucket an entry belongs to.
  – *Local depth of a bucket*: # of bits used to determine if an entry belongs to this bucket.

• When does bucket split cause directory doubling?
  – Before insert, *local depth* of bucket = *global depth*. Insert causes *local depth* to become > *global depth*; directory is doubled by *copying it over* and `fixing’ pointer to split image page. (Use of least significant bits enables efficient doubling via copying of directory!)
Directory Doubling

Why use least significant bits in directory?
⇔ Allows for doubling via copying!

6 = 110

Least Significant vs. Most Significant
Comments on Extendible Hashing

• If directory fits in memory, equality search answered with one disk access; else two.
  – 100MB file, 100 bytes/rec, 4K pages contains 1,000,000 records (as data entries) and 25,000 directory elements; chances are high that directory will fit in memory.
  – Directory grows in spurts, and, if the distribution of hash values is skewed, directory can grow large.
  – Multiple entries with same hash value cause problems!

• **Delete:** If removal of data entry makes bucket empty, can be merged with `split image`. If each directory element points to same bucket as its split image, can halve directory.
Linear Hashing

• This is another dynamic hashing scheme, an alternative to Extendible Hashing.

• LH handles the problem of long overflow chains without using a directory, and handles duplicates.

• Idea: Use a family of hash functions $h_0, h_1, h_2, ...$
  – $h_i(key) = h(key) \mod(2^iN)$; $N = \text{initial \# buckets}$
  – $h$ is some hash function (range is not 0 to N-1)
  – If $N = 2^{d_0}$, for some $d_0$, $h_i$ consists of applying $h$ and looking at the last $d_i$ bits, where $d_i = d_0 + i$.
  – $h_{i+1}$ doubles the range of $h_i$ (similar to directory doubling)
Linear Hashing (Contd.)

• Directory avoided in LH by using overflow pages, and choosing bucket to split round-robin.
  – Splitting proceeds in `rounds’. Round ends when all $N_R$ initial (for round $R$) buckets are split. Buckets 0 to $Next - 1$ have been split; $Next$ to $N_R$ yet to be split.
  – Current round number is Level.
  – **Search:** To find bucket for data entry $r$, find $h_{Level}(r)$:
    • If $h_{Level}(r)$ in range `Next to $N_R'$, $r$ belongs here.
    • Else, $r$ could belong to bucket $h_{Level}(r)$ or bucket $h_{Level}(r) + N_R$; must apply $h_{Level+1}(r)$ to find out.
Overview of LH File

• In the middle of a round.

Buckets that existed at the beginning of this round:
this is the range of

Bucket to be split

Next

Buckets split in this round:
If \( h_{\text{Level}} \) (search key value) is in this range, must use \( h_{\text{Level}+1} \) (search key value) to decide if entry is in `split image' bucket.

\( h_{\text{Level}} \)

`split image' buckets:
created (through splitting of other buckets) in this round
Linear Hashing (Contd.)

• **Insert**: Find bucket by applying $h_{\text{Level}} / h_{\text{Level+1}}$:
  – If bucket to insert into is full:
    • Add overflow page and insert data entry.
    • *(Maybe)* Split Next bucket and increment Next.
• Can choose any criterion to ``trigger’ split.
• Since buckets are split round-robin, long overflow chains don’t develop!
• Doubling of directory in Extendible Hashing is similar; switching of hash functions is *implicit* in how the # of bits examined is increased.
Example of Linear Hashing

- On split, $h_{\text{Level}+1}$ is used to re-distribute entries.

Level=0, N=4

<table>
<thead>
<tr>
<th>h</th>
<th>h</th>
<th>PRIMARY PAGES</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>32<em>44</em>36*</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>01</td>
<td>9<em>25</em>5*</td>
</tr>
<tr>
<td>010</td>
<td>10</td>
<td>14<em>18</em>10<em>30</em></td>
</tr>
<tr>
<td>011</td>
<td>11</td>
<td>31<em>35</em>7<em>11</em></td>
</tr>
</tbody>
</table>

Next=0

Level=0

<table>
<thead>
<tr>
<th>h</th>
<th>h</th>
<th>PRIMARY PAGES</th>
<th>OVERFLOW PAGES</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>32*</td>
<td>43*</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>01</td>
<td>9<em>25</em>5*</td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>10</td>
<td>14<em>18</em>10<em>30</em></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>11</td>
<td>31<em>35</em>7<em>11</em></td>
<td></td>
</tr>
</tbody>
</table>

Data entry $r$ with $h(r)=5$

Primary bucket page

(This info is for illustration only!)

(The actual contents of the linear hashed file)
Example: End of a Round

Level=0

<table>
<thead>
<tr>
<th>h1</th>
<th>h0</th>
<th>PRIMARY PAGES</th>
<th>OVERFLOW PAGES</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>32*</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>9* 25*</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>66<em>18</em>10* 34*</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>31<em>35</em> 7* 11*</td>
<td>43*</td>
</tr>
<tr>
<td>10</td>
<td>00</td>
<td>44<em>36</em></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>01</td>
<td>5* 37<em>29</em></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>14<em>30</em>22*</td>
<td></td>
</tr>
</tbody>
</table>

Next=3

Level=1

<table>
<thead>
<tr>
<th>h1</th>
<th>h0</th>
<th>PRIMARY PAGES</th>
<th>OVERFLOW PAGES</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>32*</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>9* 25*</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>66<em>18</em>10* 34*</td>
<td>50*</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>43* 35* 11*</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>00</td>
<td>44* 36*</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>5* 37* 29*</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>14* 30* 22*</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>31* 7*</td>
<td></td>
</tr>
</tbody>
</table>
Summary

- Hash-based indexes: best for equality searches, cannot support range searches.
- Static Hashing can lead to long overflow chains.
- Extendible Hashing avoids overflow pages by splitting a full bucket when a new data entry is to be added to it. *(Duplicates may require overflow pages.)*
  - Directory to keep track of buckets, doubles periodically.
  - Can get large with skewed data; additional I/O if this does not fit in main memory.
Summary (Contd.)

• Linear Hashing avoids directory by splitting buckets round-robin, and using overflow pages.
  – Overflow pages not likely to be long.
  – Duplicates handled easily.
  – Space utilization could be lower than Extendible Hashing, since splits not concentrated on `dense’ data areas.
    • Can tune criterion for triggering splits to trade-off slightly longer chains for better space utilization.

• For hash-based indexes, a skewed data distribution is one in which the hash values of data entries are not uniformly distributed!
External Sorting

Chapter 13
Why Sort?

• A classic problem in computer science!
• Data requested in sorted order
  – e.g., find students in increasing *gpa* order
• Sorting is first step in *bulk loading* B+ tree index.
• Sorting useful for eliminating *duplicate copies* in a collection of records (Why?)
• *Sort-merge* join algorithm involves sorting.
• Problem: sort 1Gb of data with 1Mb of RAM.
  – why not virtual memory?
2-Way Sort: Requires 3 Buffers

• Pass 1: Read a page, sort it, write it.
  – only one buffer page is used

• Pass 2, 3, ..., etc.:
  – three buffer pages used.
Two-Way External Merge Sort

- Each pass we read + write each page in file.
- N pages in the file => the number of passes 
  \[= \lceil \log_2 N \rceil + 1\]
- So total cost is: 
  \[2N \left( \lceil \log_2 N \rceil + 1 \right)\]
- Idea: Divide and conquer: sort subfiles and merge
General External Merge Sort

More than 3 buffer pages. How can we utilize them?

• To sort a file with $N$ pages using $B$ buffer pages:
  – Pass 0: use $B$ buffer pages. Produce $\lceil N / B \rceil$ sorted runs of $B$ pages each.
  – Pass 2, ..., etc.: merge $B-1$ runs.
Cost of External Merge Sort

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

<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>
<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>
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

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

• For data entries; each data entry contains \textit{rid} of a data record. In general, \textit{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.
External sorting is important; DBMS may dedicate part of buffer pool for sorting!

External merge sort minimizes disk I/O cost:
- Pass 0: Produces sorted runs of size $B$ (# buffer pages). Later passes: merge runs.
- # of runs merged at a time depends on $B$, and block size.
- Larger block size means less I/O cost per page.
- Larger block size means smaller # runs merged.
- In practice, # of runs rarely more than 2 or 3.
Summary, cont.

• Choice of internal sort algorithm may matter:
  – Quicksort: Quick!
  – Heap/tournament sort: slower (2x), longer runs

• The best sorts are wildly fast:
  – Despite 40+ years of research, we’re still improving!

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