CAS CS 460/660
Introduction to Database Systems
Indexing: Hashing
Introduction

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

- Static and dynamic hashing techniques exist; trade-offs similar to ISAM vs. B+ trees.

- **Recall**, 3 alternatives for data entries $k^*$:
  1. Data record with key value $k$
  2. $<k$, rid of data record with search key value $k$>
  3. $<k$, list of rids of data records w/search key $k$>

Choice is orthogonal to the *indexing technique*
Static Hashing

- # primary pages fixed, allocated sequentially, never de-allocated; overflow pages if needed.
- A simple hash function (for N buckets):
  \[ h(k) = k \mod N \]

  is bucket # where data entry with key \( k \) belongs.
Static Hashing (Contd.)

- Buckets contain *data entries*.
- Hash fn works on *search key* field of record *r*. Use MOD N to distribute values over range 0 ... N-1.
  - $h(key) = key \mod N$ works well for uniformly distributed data.
  - better: $h(key) = (A*key \mod P) \mod N$, where $P$ is a prime number
  - various ways to tune $h$ for non-uniform (checksums, crypto, etc.).

- As with any static structure: **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.
  - Want to avoid overflow pages
- Add more buckets (i.e., increase “N”)?
  - Okay, but need a new hash function!
- Doubling # of buckets makes this easier
  - Say N values are powers of 2: how to do “mod N”? 
  - What happens to hash function when double “N”?
- Problems with Doubling
  - Don’t want to have to double the size of the file.
  - Don’t want to have to move all the data.
Extendible Hashing (cont)

- **Idea**: Add a level of indirection!

- Use *directory of pointers to buckets*,

- Double # of buckets by *doubling the directory*
  - Directory much smaller than file, so doubling it is much cheaper.

- Split only the bucket that just overflowed!
  - *No overflow pages!*
  - Trick lies in how hash function is adjusted!
How it Works

- Directory is array of size 4, so 2 bits needed.
- Bucket for record \( r \) has entry with index = `global depth' least significant bits of \( h(r) \);
  - If \( h(r) = 5 = \text{binary } 101 \), it is in bucket pointed to by 01.
  - If \( h(r) = 7 = \text{binary } 111 \), it is in bucket pointed to by 11.
Handling Inserts

- Find bucket where record belongs.
- If there’s room, put it there.
- Else, if bucket is full, *split* it:
  - increment local depth of original page
  - allocate new page with new local depth
  - re-distribute records from original page.
  - add entry for the new page to the directory
Example: Insert 21, 19, 15

- 21 = 10101
- 19 = 10011
- 15 = 01111

we denote key $r$ by $h(r)$. 
Insert $h(r)=20$ (Causes Doubling)
20 = binary 10100. Last 2 bits (00) tell us r in either 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 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.
Directory Doubling

Why use least significant bits in directory (instead of the most significant ones)?

Allows for doubling by copying the directory and appending the new copy to the original.

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 – a lazy approach

- A dynamic hashing scheme that handles the problem of long overflow chains without using a directory.

- Directory avoided in LH by using temporary overflow pages, and choosing the bucket to split in a round-robin fashion.

- When any bucket overflows split the bucket that is currently pointed to by the “Next” pointer and then increment that pointer to the next bucket.
Linear Hashing – The Main Idea

- Use a family of hash functions $h_0, h_1, h_2, \ldots$

- $h_i(key) = h(key) \mod (2^iN)$
  - $N = \text{initial \# buckets} \ (\text{must be a power of 2})$
  - $h$ is some hash function

- $h_{i+1}$ doubles the range of $h_i$ (similar to directory doubling)
Algorithm proceeds in ‘rounds’. Current round number is “Level”.

There are \( N_{\text{Level}} = N \times 2^{\text{Level}} \) buckets at the beginning of a round (so \( N = N_0 \))

Buckets \( 0 \) to \( \text{Next}-1 \) have been split; \( \text{Next} \) to \( N_{\text{Level}} \) have not been split yet this round.

Round ends when all initial buckets have been split (i.e. \( \text{Next} = N_{\text{Level}} \)).

To start next round:
\[
\text{Level}++; \\
\text{Next} = 0;
\]
LH Search Algorithm

To find bucket for data entry \( r \), find \( h_{\text{Level}}(r) \):

Buckets not yet split this round:

If \( h_{\text{Level}}(r) \geq \text{Next} \) (i.e., \( h_{\text{Level}}(r) \) is a bucket that hasn’t been involved in a split this round) then \( r \) belongs in that bucket for sure.

Buckets split already this round:

Else, \( r \) could belong to bucket \( h_{\text{Level}}(r) \) or bucket \( h_{\text{Level}}(r) + N_{\text{Level}} \) must apply \( h_{\text{Level}+1}(r) \) to find out.
Ex: Search $h(key) = 44 \ (11100), \ h(key) = 9 \ (01001)$

Level=0, Next=0, N=4

$\begin{array}{c|c}
\text{h} & \text{h} \\
1 & 0 \\
000 & 00 \\
001 & 01 \\
010 & 10 \\
011 & 11 \\
\end{array}$

(This info is for illustration only!)

$\begin{array}{c}
\text{PRIMARY}\text{ PAGES} \\
32^\ast 44^\ast 36^\ast \\
9^\ast 25^\ast 5^\ast \\
14^\ast 18^\ast 10^\ast 30^\ast \\
31^\ast 35^\ast 7^\ast 11^\ast \\
\end{array}$

$h_{\text{Level}}(key) = h(key) \mod (2^{\text{Level}N})$
Linear Hashing - Insert

- Find appropriate bucket, if fits, then DONE.
- else, if no room:
  - Add overflow page and insert data entry.
  - Split $Next$ bucket and increment $Next$.
    - Note: This is likely NOT the bucket being inserted to!!!
    - to split a bucket, create a new bucket and use $h_{\text{Level}+1}$ to re-distribute entries.

- Since buckets are split round-robin, long overflow chains don’t develop!
Example: Insert 43 (101011)

**Level=0, N=4**

<table>
<thead>
<tr>
<th>h</th>
<th>h</th>
<th>h</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>000</td>
<td>00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>01</td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>11</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*(This info is for illustration only!)*

Next=0

<table>
<thead>
<tr>
<th>PRIMARY PAGES</th>
</tr>
</thead>
<tbody>
<tr>
<td>32<em>44</em>36*</td>
</tr>
<tr>
<td>9<em>25</em>5*</td>
</tr>
<tr>
<td>14<em>18</em>10<em>30</em></td>
</tr>
<tr>
<td>31<em>35</em>7<em>11</em></td>
</tr>
</tbody>
</table>

**Level=0**
Next=1

<table>
<thead>
<tr>
<th>OVERFLOW PAGES</th>
</tr>
</thead>
<tbody>
<tr>
<td>32*</td>
</tr>
<tr>
<td>9<em>25</em>5*</td>
</tr>
<tr>
<td>14<em>18</em>10<em>30</em></td>
</tr>
<tr>
<td>31<em>35</em>7<em>11</em></td>
</tr>
<tr>
<td>43*</td>
</tr>
<tr>
<td>44<em>36</em></td>
</tr>
</tbody>
</table>

*(This info is for illustration only!)*
Example: Search 9 (010001) 44 (11100)

Level=0, Next = 1, N=4

(This info is for illustration only!)
Example: End of a Round

Insert 50 (110010)

<table>
<thead>
<tr>
<th>Level=0, Next = 3</th>
<th>PRIMARY PAGES</th>
<th>OVERFLOW PAGES</th>
</tr>
</thead>
<tbody>
<tr>
<td>h₁ h₀</td>
<td>000 00</td>
<td>32*</td>
</tr>
<tr>
<td></td>
<td>001 01</td>
<td>9* 25*</td>
</tr>
<tr>
<td></td>
<td>010 10</td>
<td>66<em>18</em>10* 34*</td>
</tr>
<tr>
<td></td>
<td>011 11</td>
<td>31<em>35</em> 7* 11*</td>
</tr>
<tr>
<td></td>
<td>100 00</td>
<td>44<em>36</em></td>
</tr>
<tr>
<td></td>
<td>101 01</td>
<td>5* 37<em>29</em></td>
</tr>
<tr>
<td></td>
<td>110 10</td>
<td>14<em>30</em>22*</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Level=1, Next = 0</th>
<th>PRIMARY PAGES</th>
<th>OVERFLOW PAGES</th>
</tr>
</thead>
<tbody>
<tr>
<td>h₁ h₀</td>
<td>000 00</td>
<td>32*</td>
</tr>
<tr>
<td></td>
<td>001 01</td>
<td>9* 25*</td>
</tr>
<tr>
<td></td>
<td>010 10</td>
<td>66<em>18</em>10* 34*</td>
</tr>
<tr>
<td></td>
<td>011 11</td>
<td>43* 35* 11*</td>
</tr>
<tr>
<td></td>
<td>100 00</td>
<td>44<em>36</em></td>
</tr>
<tr>
<td></td>
<td>101 11</td>
<td>5* 37<em>29</em></td>
</tr>
<tr>
<td></td>
<td>110 10</td>
<td>14<em>30</em>22*</td>
</tr>
<tr>
<td></td>
<td>111 11</td>
<td>31<em>7</em></td>
</tr>
</tbody>
</table>

Insert 50 (110010)
LH Described as a Variant of EH

The two schemes are actually quite similar:

- Begin with an EH index where directory has $N$ elements.
- Use overflow pages, split buckets round-robin.
- First split is at bucket 0. (Imagine directory being doubled at this point.) But elements $<1,N+1>$, $<2,N+2>$, ... are the same. So, need only create directory element $N$, which differs from 0, now.
  - When bucket 1 splits, create directory element $N+1$, etc.

So, “directory” can double gradually. Also, primary bucket pages are created in order. If they are allocated in sequence too (so that finding i’th is easy), we actually don’t need a directory! Voila, LH.
Summary

- Hash-based indexes: best for equality searches, cannot support range searches.
- Static Hashing can have 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.
Linear Hashing avoids directory by splitting buckets round-robin & using overflow pages.

- Overflow pages not likely to be long.
- 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 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!