INDEXING – HASHING
Selection Queries

B+-tree is perfect, but....

to answer a selection query (ssn=10) needs to traverse a full path.

In practice, 3-4 block accesses (depending on the height of the tree, buffering)

Any better approach?

Yes! Hashing

- static hashing
- dynamic hashing
Hashing

- **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.
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 \( r \). Use its value MOD N to distribute values over range 0 ... N-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.
- Bucket for record \( r \) has entry with index = \( \text{``global depth''} \) 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.

\[ h(r) \]

we denote \( r \) by \( h(r) \).
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, then 19, 15

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

GLOBAL DEPTH

LOCAL DEPTH

DIRECTORY

DATA PAGES

Bucket A
- 4*
- 12*
- 32*
- 16*

Bucket B
- 2*
- 1*
- 5*
- 21*
- 13*

Bucket C
- 2*
- 10*

Bucket D
- 2*
- 7*
- 19*
- 15*
Insert $h(r)=20$ (Causes Doubling)

Bucket A

- LOCAL DEPTH: 3
- GLOBAL DEPTH: 3
- Elements: $32*16^*$

Bucket B

- LOCAL DEPTH: 2
- GLOBAL DEPTH: 2
- Elements: $1^* 5^* 21^*13^*$

Bucket C

- LOCAL DEPTH: 2
- GLOBAL DEPTH: 2
- Elements: $10^*$

Bucket D

- LOCAL DEPTH: 2
- GLOBAL DEPTH: 2
- Elements: $15^* 7^* 19^*$

Bucket A2 (split image of Bucket A)

- LOCAL DEPTH: 3
- GLOBAL DEPTH: 3
- Elements: $4^* 12^* 20^*$
Points to Note

- 20 = binary 10100. Last 2 bits (00) tell us \( r \) belongs 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 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.
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 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 $h_0, h_1, h_2, ...$
  - $h_i(key) = h(key) \mod (2^iN)$
    - $N = \text{initial } \# \text{ buckets}$
    - $h$ is some hash function
  - $h_{i+1}$ doubles the range of $h_i$ (similar to directory doubling)
Linear Hashing (Contd.)

- Algorithm proceeds in `rounds’. Current round number is “Level”.
- There are $N_{Level} (= N \times 2^{Level})$ buckets at the beginning of a round.
- Buckets 0 to $Next-1$ have been split; $Next$ to $N_{Level}$ have not been split yet this round.
- Round ends when all initial buckets have been split (i.e. $Next = N_{Level}$).
- To start next round:
  
  Level++;
  Next = 0;
Linear Hashing - Insert

- Find appropriate bucket
- If bucket to insert into is full:
  - 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!
Overview of LH File

- In the middle of a round.

Buckets that existed at the beginning of this round: this is the range of $h_{\text{Level}}$

Bucket to be split

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.

'split image' buckets: created (through splitting of other buckets) in this round
Example: Insert 43 (101011)

Level=0, N=4

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

(This info is for illustration only!)

**PRIMARY PAGES**

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

(Next=0)

```
32* 44* 36*
9* 25* 5*
14* 18* 10* 30*
31* 35* 7* 11*
```

Level=0

Next=1

```
<table>
<thead>
<tr>
<th>h</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>000</td>
<td>00</td>
</tr>
<tr>
<td>001</td>
<td>01</td>
</tr>
<tr>
<td>010</td>
<td>10</td>
</tr>
<tr>
<td>100</td>
<td>00</td>
</tr>
</tbody>
</table>
```

**PRIMARY PAGES**

```
32*
9* 25* 5*
14* 18* 10* 30*
31* 35* 7* 11*
```

**OVERFLOW PAGES**

```
43*
```

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

#### Insert 50 (110010)

**Level=0, Next = 3**

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

**Level=1, Next = 0**

<table>
<thead>
<tr>
<th>h1</th>
<th>h0</th>
<th>PRIMARY PAGES</th>
<th>OVERFLOW PAGES</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 00</td>
<td>66* 18* 10* 34* 50*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001 01</td>
<td>9* 25*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>010 10</td>
<td>43* 35* 11*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>011 11</td>
<td>44* 36*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100 00</td>
<td>5* 37* 29*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>101 11</td>
<td>14* 30* 22*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>110 11</td>
<td>31<em>7</em></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
LH Search Algorithm

- To find bucket for data entry $r$, find $h_{\text{Level}}(r)$:
  - 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.
  - 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.
Example: Search 44 (11100), 9 (01001)

Level=0, Next=0, N=4

(This info is for illustration only!)
Example: Search 44 (11100), 9 (01001)

Level=0, Next = 1, N=4

(This info is for illustration only!)

<table>
<thead>
<tr>
<th>h</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>000</td>
<td>00</td>
</tr>
<tr>
<td>001</td>
<td>01</td>
</tr>
<tr>
<td>010</td>
<td>10</td>
</tr>
<tr>
<td>011</td>
<td>11</td>
</tr>
<tr>
<td>100</td>
<td>00</td>
</tr>
</tbody>
</table>

PRIMARY PAGES

OVERFLOW PAGES

32*
9* 25* 5*
14* 18* 10* 30*
31* 35* 7* 11*
44* 36*

Example: Search 44 (11100), 9 (01001)