CS 133: Databases

Fall 2016
Lec 6 – 9/19

Prof. Beth Trushkowsky

Goals for Today

• Learn how hash-based indexes are constructed

• Understand how operations work on static and dynamic hash indexes, and the impact on cost in I/Os

• Reason about the tradeoffs between approaches to dynamic hash indexes

Recall: Data Entries

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

Choice is orthogonal to the indexing technique

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

• Static and dynamic hashing techniques exist; tradeoffs similar to ISAM vs. B+ trees.

Static Hashing

• Number of primary pages fixed
  – Allocated sequentially
  – Never de-allocated; overflow pages if needed.

$h(search\ key)$ yields which bucket

key $\rightarrow h(key)$

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Primary bucket pages</td>
<td>Overflow pages</td>
<td></td>
</tr>
</tbody>
</table>
Hashing Functions

- Hash function works on search key field(s) of record
- What would be good properties for a hash function?

Desirable properties for hash function:
- Uniform distribution: the same number of search key values map to each bucket, for all possible values
- Random distribution: at any given point in time, each bucket has the same number of search key values

In practice
- Typically operate on a binary representation of the data
- \( h(key) = key \mod N \) works well for uniformly distributed integer data
- Can tune \( h \) to achieve uniformity (e.g., cryptographic techniques)

Static Hashing

- Example:
  - Suppose creating index over integer search key, e.g., SID or age
  - \# buckets \( N = 4 \) → Multiple of 2, use 2 bits to differentiate

<table>
<thead>
<tr>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

  - Insert 9
  - \( 9 \mod 4 = 1 \)
  - \( 1001 \) & \( 0011 \) = \( 0001 \)

  Try Exercise 2

MOAR Buckets

- Situation: Bucket (primary page) becomes full.
  - Want to avoid chains of overflow pages

- Solution: add more buckets (i.e., increase “N”)?
  - Okay, but need a new hash function!
  - Doubling # of buckets makes this easier, use one more bit to get 2N buckets
    - Any potential issues with doubling the number of buckets?!

- Two dynamic approaches:
  - Extendible hashing
  - Linear hashing

Extendible Hashing

- Idea: add level of indirection!

- Use a directory to point to buckets
- “Double” # of buckets by doubling the directory
  - Directory much smaller than file, so doubling it is much cheaper (might fit in RAM)
  - When want to “split” a bucket, double the directory
  - Allocate new page only for the split bucket

\[
\begin{array}{c}
\text{GLOBAL DEPTH} \\
00 & 01 & 10 & 11 \\
0 & 1 & 2 & 3 \\
\end{array}
\]

\[
\begin{array}{c}
\text{LOCAL DEPTH} \\
9^* (1001) \\
11^* (1011) \\
7^* (011) \\
\end{array}
\]
Handling Inserts

- Find bucket where data entry 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
  - double directory if necessary (when local > global)
  - add entry for the new page to the directory

---

Example: Insert 21*, 19*, 15*

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

---

Example: Insert 21*, 19*, 15* (before picture)

- LOCAL DEPTH
- GLOBAL DEPTH

---

Insert 20* (10100): Causes Doubling

- LOCAL DEPTH
- GLOBAL DEPTH

---

Example: Insert 21*, 19*, 15*

- LOCAL DEPTH
- GLOBAL DEPTH

---

Example: Insert 21*, 19*, 15* (before picture)

- LOCAL DEPTH
- GLOBAL DEPTH
Insert 20* (10100): Causes Doubling

Local vs. Global Depth

Extendible Hashing: Comments
• If directory fits in memory, equality search answered with one disk access; else two
• Avoids overflow pages (besides those needed for duplicates/collisions)

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 Lazier Approach
• Issues with Extendible
  – Completion of an insertion can take a while if it caused a split... have to move data around

• Linear Hashing:
  – Idea: decouple what is split from the action that triggers a split
  – A dynamic hashing scheme that handles the problem of long overflow chains without using a directory
Linear Hashing Example

- Avoids directory by:
  - using temporary overflow pages and choosing the bucket that is split in a round-robin fashion.
  - For example, 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 \( h_0, h_1, h_2, \ldots \)
- \( h_i(key) = key \mod(2^N) \)
  - \( N \) = initial # buckets (a power of 2)
  - \( h_{i+1} \) doubles the range of \( h_i \) (similar to directory doubling in extendible hashing)
- Note: at a given time, could be “using” two hash functions: one function for buckets that have been split vs. ones that haven’t

Linear Hashing — Search Algorithm

To find bucket for data entry \( k \), first find \( h_{\text{level}}(k) \). Then:

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

Else, \( k \) could belong to bucket \( h_{\text{level}}(k) \) or bucket \( h_{\text{level}}(k) + N_{\text{level}} \), must apply \( h_{\text{level+1}}(k) \) to find out
**Example: Search 44 (11100), 9 (01001)**

Level=0, Next=0, N=4

\[ h_{\text{Level}}(key) = key \mod (2^{\text{Level}\times N}) \]

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 \( Next \).
    - 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, Next = 0, N=4

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

Level=0, Next = 1, N=4

For 44*, use \( h_1 \)

For 9*, still use \( h_0 \)
Example: End of a Round

Insert 50 (110010)

Level=0, Next = 3

Level=1, Next = 0

Extendible vs. Linear

• Extendible
  – Directory grows in spurts, and, if the distribution of hash values is skewed, directory can grow large

• Linear
  – 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