CSE462/562: Database Systems (Spring 23)

Lecture 7: Hash Index

2/21/2023
Hash-based index

• Hash-based index are best for equality searches
  • Does not support range searches

• Difference from in-memory hash table
  • Page oriented: multiple data entries per hash bucket
  • (Usually) open hashing
    • Probing needed to physically delete a data entry with closed hashing
  • Rehashing is very expensive! (reading + writing all the pages)

• Static vs dynamic hashing techniques
  • Static: fixed hash value domain $[m]$
    • Easy to implement, no rehashing overhead
    • Inefficient if number of records is large
  • Dynamic: grow hash value domain over time
    • Sometimes needs to rehash
      • How to amortize cost?
    • Scales gracefully with number of records if the choice of hash function is good
Static hashing

• # primary pages fixed, allocated sequentially, never de-allocated; overflow pages if needed.
• \( h(k) \mod M \) = bucket to which data entry with key k belongs. (\( M = \# \) of buckets)
Static hashing

- Buckets contain *data entries*.
- Hash function works on *search key* field(s) of record $r$.
  - Use its value MOD $N$ to distribute values over range 0 ... $N-1$.
- **Long overflow chains** can develop over time and degrade performance.
  - *Extendible* and *Linear Hashing*: dynamic techniques to fix this problem.

![Diagram of Static Hashing](image-url)
Extendible hashing

• When the primary page of a bucket gets full,
  • why not doubling the number of buckets and rehash?
  • reading and writing all the pages are very expensive!

• Idea: use directory of pointers to buckets (these pointers are page numbers)
  • To double number of buckets, only need to double the directory size
  • Only split the bucket about to overflow. No overflow page!

• Why this works?
  • Directory is much smaller than the data files
  • Uses a family of hash functions $h_D: U \rightarrow [2^D ]$
    • Trick is how to switch from $h_D$ to $h_{D+1}$ without rehashing for doubling number of buckets
Extendible hashing example

- Hash function $h: U \rightarrow [2^{32}]$ (or $[2^{64}]$ depending on the type of hash value)
  - Define $h_D(k) = h(k) \mod 2^D$ -- therefore $h_D: U \rightarrow [2^D]$
  - Essentially taking the lowest $i$ bits of the key hash as the hash value

- Directory: an array of pointers (page numbers) of size $2^D$
  - $D$: global depth
  - Each points to a bucket $p_i$ (*not necessarily unique ones*)
    - A data entry with key $k$ is in $p_i$ iff $h_D(k) = i$

- Each bucket has a local depth $d_i$
  - Can be used to determine whether this bucket is currently shared by two hash values
    *A bucket is not shared iff $D = d_i$*
  - $h_D(k)$ may be different in a bucket
    - Question: what’s in common?
    - $h_{d_i}(k)$ are always the same

**Diagram:**

- **GLOBAL DEPTH**
  - $2$, $4^*$, $12^*$, $32^*$, $16^*$
  - **LOCAL DEPTH**
  - $2$, $1^*$, $5^*$, $7^*$, $13^*$

- **DIRECTORY**
  - $00$, $01$, $10^*$, $11$**
Handling inserts in extendible hashing

- Find the bucket $p_i$ where the insertion belongs
- If there’s room, insert it into $p_i$
- If not, split $p_i$ before insertion
  - increment the local depth $d_i$
  - allocate a new page with the same (new) local depth
  - redistribute the data entries with the new page
  - double the global depth if local depth is now larger than global depth
    - also duplicate the old directory if global depth is doubled
  - set the pointer for the new page in the directory

```
2
1
0

00 01 10 11

DIRECTORY

10* 11* 12* 13* 32* 4*

2
1

Bucket A
Bucket B
Bucket C

CSE462/562 (Spring 2023): Lecture 7
7
```
Example: inserting $21^*, 19^*, 15^*$

- $21 = (10101)_2$
- $19 = (10011)_2$
- $15 = (01111)_2$

```
21 * 19 * 15 = 10101
10011
01111
```

```
\[
h(k) \mod 2^2 = 1
\]
```

```
\[
h(k) \mod 2^2 = 3
\]
```
Example: inserting $20^*$ (causing doubling)

- $20 = (10100)_2$
- Last 2 bits (00) tell us $20^*$ belongs to A or E
  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.
Notes on extendible hashing

• If directory fits in memory, equality search answered with one disk access; else two.
  • If the distribution of hash values is skewed, directory can grow large.
  • Multiple entries with same hash value cause problems!
    • What if we still don’t have space after split?

• 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

• Another dynamic hashing scheme that handles long overflow chains without using a directory.

• Page to split is chosen in a round-robin fashion, not where it will overflow
  • LH allows using temporary overflow pages
  • If the hash values are reasonably uniform -- overflows will be resolved quickly
Handling insertion in linear hashing

- Also uses a family of hash functions
  - \( h_i(k) = h(k) \mod (2^i N) \)
  - Initial size \( N \) does not need to be power of 2

- Proceeds in “rounds”.
  Current round number is called level \( l \geq 0 \)

- There are \( N_l = N \times 2^l \) buckets at the beginning
  - \( Next \) initially set to 0
  - Invariant:
    - Buckets \([0, Next)\) has been split in this round
    - Buckets \([Next, N_l)\) are to be split in this round

- On insert
  - If the bucket for insertion is full
    - Add an overflow page and insert the data entry
    - Split \( Next \) bucket and increment \( Next \)
    - Use \( h_{l+1} \) to redistribute entries for a split bucket

- Round ends when \( Next = N_l \)
  - Start a new round, \( Next \leftarrow 0, l \leftarrow l + 1 \)

<table>
<thead>
<tr>
<th>( h_1 )</th>
<th>( h_0 )</th>
</tr>
</thead>
<tbody>
<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>

Primary pages

\( N = 4, l = 0 \)
\( Next = 0 \)

```latex
\begin{tabular}{|c|c|c|c|}
\hline
00 & 00 & 32\* & 44\* & 36\* \\
01 & 01 & 09\* & 25\* & 05\* \\
10 & 10 & 014\* & 18\* & 010\* & 030\*  \\
11 & 11 & 031\* & 035\* & 007\* & 011\*  \\
\hline
\end{tabular}
```
Example: insert $43^*$ $(101011)_2$

\[ N = 4, l = 0 \]

Next = 1

<table>
<thead>
<tr>
<th>$h_1$</th>
<th>$h_0$</th>
<th><strong>Primary pages</strong></th>
<th><strong>Overflow pages</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>00</td>
<td>32*</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>01</td>
<td>9* 25* 5*</td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>10</td>
<td>14* 18* 10* 30*</td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>11</td>
<td>31* 35* 7* 11*</td>
<td>43*</td>
</tr>
<tr>
<td>100</td>
<td>00</td>
<td>44* 36*</td>
<td></td>
</tr>
</tbody>
</table>
Example: end of a round

Insert $50^* (110010)_2$

$$N = 4, l = 0$$

$Next = 3$

<table>
<thead>
<tr>
<th>$h_1$</th>
<th>$h_0$</th>
<th>Primary pages</th>
<th>Overflow pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>01</td>
<td>9* 25*</td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>10</td>
<td>66* 18* 10* 34*</td>
<td>43*</td>
</tr>
<tr>
<td>011</td>
<td>11</td>
<td>31* 35* 7* 11*</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>00</td>
<td>44* 36*</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>01</td>
<td>5* 37* 29*</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>10</td>
<td>14* 30* 22*</td>
<td></td>
</tr>
</tbody>
</table>
**Example: end of a round (cont’d)**

Insert $50^* (110010)_2$

\[ N = 4, \ l = 1 \]

\[ \text{Next} = 0 \]

<table>
<thead>
<tr>
<th>$h_1$</th>
<th>Primary pages</th>
<th>Overflow pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>32*</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>9* 25*</td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>66* 18* 10* 34*</td>
<td>50*</td>
</tr>
<tr>
<td>011</td>
<td>43* 35* 11*</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>44* 36*</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>5* 37* 29*</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>14* 30* 22*</td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>31* 7*</td>
<td></td>
</tr>
</tbody>
</table>

CSE462/562 (Spring 2023): Lecture 7
Linear Hashing Search Algorithm

- To find the bucket for a data entry $k^*$
  - Compute $h_l(k) = h(k) \mod (2^l N)$
    - If $h_l(k) \geq \text{Next}$
      - Bucket $h_l(k)$ is the bucket for $k^*$ (because it hasn’t been split in this round)
    - Otherwise,
      - $k^*$ could belong to either bucket $h_l(k)$ or bucket $h_l(k) + 2^l N$
      - Compute $h_{l+1}(k)$ to find out

$$N = 4, l = 0$$
$$\text{Next} = 1$$

<table>
<thead>
<tr>
<th></th>
<th>$h_1$</th>
<th>$h_0$</th>
<th>Primary pages</th>
<th>Overflow pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Find 32*?</td>
<td>000</td>
<td>00</td>
<td>32*</td>
<td>43*</td>
</tr>
<tr>
<td>Find 14*?</td>
<td>001</td>
<td>01</td>
<td>9* 25* 5*</td>
<td></td>
</tr>
<tr>
<td>Find 43*?</td>
<td>010</td>
<td>10</td>
<td>14* 18* 10* 30*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>011</td>
<td>11</td>
<td>31* 35* 7* 11*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>00</td>
<td>44* 36*</td>
<td></td>
</tr>
</tbody>
</table>
Notes on linear hashing

• If hash values are skewed
  • because of key skew or bad hash function
  • then will still have long overflow chains

• Delete: the reverse of insertion algorithm
Composite keys in hash index

• Composite key: multiple fields as the key (f1, f2, ..., fk)

• How to handle composite keys in hash index?
  • Combine the hash values of each field together
  • Many libs available, e.g., boost::hash_combine, absl::Hash::combine(), etc ...
    e.g., in boost:
    seed ^= hash_value + 0x9e3779b9 + (seed<<6) + (seed>>2);

• Search with composite keys
  • Must specify all the fields, equality search only
  • Can’t perform partial key search
    • e.g., hash index on (sid, login)
      • may be used for predicate sid = 12345 AND login = ‘alice’
      • but not sid = 12345
      • nor login = ‘Alice’
Summary

• This lecture
  • Static hashing
  • Dynamic hashing
    • Extendible hashing
    • Linear hashing
  • Composite keys in hash index

• Next two lectures
  • Tree indexes