ICS 421 Spring 2010

Indexing (2)

Asst. Prof. Lipyeow Lim
Information & Computer Science Department
University of Hawaii at Manoa
Hash Indexes

• 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; trade-offs similar to ISAM vs. B+ trees.
The Hashing Idea (i)

What if we want to use an array of 7 slots?

Essential idea: get an array index/address directly from the key field

<table>
<thead>
<tr>
<th>DOW Number</th>
<th>DOW String</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Monday</td>
</tr>
<tr>
<td>2</td>
<td>Tuesday</td>
</tr>
<tr>
<td>3</td>
<td>Wednesday</td>
</tr>
<tr>
<td>4</td>
<td>Thursday</td>
</tr>
<tr>
<td>5</td>
<td>Friday</td>
</tr>
<tr>
<td>6</td>
<td>Saturday</td>
</tr>
<tr>
<td>7</td>
<td>Sunday</td>
</tr>
</tbody>
</table>


Print ( “Day 4 of the week is “ + DOWstring[4] );
The Hashing Idea (ii)

What do we do if two strings map to the same hash value?

<table>
<thead>
<tr>
<th>DOW Number</th>
<th>DOW String</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Monday</td>
</tr>
<tr>
<td>2</td>
<td>Tuesday</td>
</tr>
<tr>
<td>3</td>
<td>Wednesday</td>
</tr>
<tr>
<td>4</td>
<td>Thursday</td>
</tr>
<tr>
<td>5</td>
<td>Friday</td>
</tr>
<tr>
<td>6</td>
<td>Saturday</td>
</tr>
<tr>
<td>7</td>
<td>Sunday</td>
</tr>
</tbody>
</table>

How do we get the numeric day of the week (DOW) from the DOW string?

```
Int hashfn(string day) {
    foreach i do sum += day[i]
    return sum % 7;
}

String DOWnum[7];

DOWnum[hashfn("Monday")] = 1;
DOWnum[hashfn("Tuesday")] = 2;
...
Print ( "Monday is day " +
    DOWnum[hashfn("Monday")] + " of the week");
```

What do we do if two strings map to the same hash value?
Hash Indexes in Databases

- Conceptually an array of pages or buckets
- \( h(k) \mod M = \text{bucket ID for key } k \)
- \( M \) is the number of buckets in array
- **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 Hashing

- **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 \( h \).

- Overflow buckets used when primary buckets are full
Example: Static Hashing

- Search key is Sailors.age
- Hash fn $h(k) = k \mod 8$
- Each bucket/page can hold 2 entries
- Insert data entries
  - (8,r1)
  - (9,r3)
  - (15,r2)
  - (7,r6)
  - (23,r9)
  - (31,r7)
  - (39,r8)

How many page IOs do I need to find RIDs of sailors aged 8?

How many page IOs do I need to find RIDs of sailors aged 31?
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: Extendible Hashing

- Directory is array of size 4.
- Each bucket holds 4 entries.
- To find bucket for $r$, take last `global depth' # bits of $h(r)$; we denote $r$ by $h(r)$.
  - If $h(r) = 5 = \text{binary } 101$, 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.

$h(k) = k$
**Example: Insert key 20**

\[ h(k) = k \]

\[ h(20) = 20 = 10100 \]
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!)

• If directory fits in memory, equality search answered with one disk access; else two.
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 $h_0, h_1, h_2, ...$
  - $h_i(key) = h(key) \mod(2^iN)$; $N =$ initial # buckets
  - $h$ is some hash function (range is not 0 to N-1)
  - If $N = 2^{d0}$, for some $d0$, $h_i$ consists of applying $h$ and looking at the last $d_i$ bits, where $d_i = d0 + i$.
  - $h_{i+1}$ doubles the range of $h_i$ (similar to directory doubling)
**Example: Linear Hashing**

- **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.

- Since buckets are split round-robin, long overflow chains don’t develop!

<table>
<thead>
<tr>
<th>$h_0(k)$ = $k \mod 4$</th>
<th>$h_1(k)$ = $k \mod 8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 00</td>
<td>32* 44* 36*</td>
</tr>
<tr>
<td>001 01</td>
<td>9* 25* 5*</td>
</tr>
<tr>
<td>010 10</td>
<td>14* 18* 10* 30*</td>
</tr>
<tr>
<td>011 11</td>
<td>31* 35* 7* 11*</td>
</tr>
<tr>
<td>100</td>
<td>44* 36*</td>
</tr>
<tr>
<td>101</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td></td>
</tr>
<tr>
<td>111</td>
<td><strong>Overflow Buckets</strong></td>
</tr>
</tbody>
</table>

**Note:**
- $h_0(k) = k \mod 4$
- $h_1(k) = k \mod 8$

**Diagram:**
- Primary Buckets
- Overflow Buckets
- Split bucket pointed by next!
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. \textit{(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 (Cont.)

• 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!