CS460: Intro to Database Systems

Class 13: Hash Indexing

Instructor: Manos Athanassoulis

https://bu-disc.github.io/CS460/
Last time: B⁺ Trees

“It could be said that the world’s information is at our fingertips because of B-trees”

Other forms of indexing?
Hash Indexing

Static Hashing

Extendible Hashing

Linear Hashing
Reminder: Alternatives of Data Entries

1. \(<k, \text{ entire data record}>\>
2. \(<k, \text{ rid of exactly-one-at-a-time matching data record}>\>
3. \(<k, \text{ list of rids of matching data records}>\>

Choice is orthogonal to the indexing technique

*Hash-based* indexes \(\Rightarrow\) *equality selections*
*Cannot* support range searches

*Static* and *dynamic* hashing techniques exist
Hash function

A function that maps a search key to an index between \([0 \ldots M-1]\)

where \(M\) is the number of buckets (pages) available to our index

- Ideally a hash function maps the search keys uniformly in \([0, \ldots, M-1]\)
- In practice simple hash functions are used (fast to compute)
- Different keys might be mapped to the same bucket
Static Hashing

#primary bucket pages fixed, allocated sequentially, never de-allocated; overflow pages if needed

\( h(k) \mod M \) = bucket to insert data entry with key \( k \) (\( M: \#baskets \))

![Diagram of Static Hashing](attachment:image)
Static Hashing (Contd.)

Buckets contain data entries

Hash function on search key field of record r

Must distribute values over range 0 ... M-1

What is a good hash function?

\[ h(key) = (a \times key + b) \] usually works well

a and b are constants; lots known about how to tune h
Static Hashing – Problems?

What does that do to performance?

Instead of $O(1)$ we may go as bad as $O(N)$
Static Hashing – Solutions

Long overflow chains can develop and degrade performance

Ways to solve?
– Reorganization (re-hashing) is expensive and may block queries
– *Extendible* and *Linear Hashing*: Dynamic techniques to fix this problem
Hash Indexing

Static Hashing

Extendible Hashing

Linear Hashing
Let’s start from Static Hashing

What else we can do instead of adding an overflow page?

$h(k) \mod M = \text{bucket to insert data entry with key } k \ (M: \#\text{buckets})$

![Diagram showing hash function and bucket allocation]

- $h(k) \mod M$ determines the bucket to insert a data entry with key $k$.
- M: number of buckets.

The diagram illustrates the function $h(key) \mod M$ mapping keys to buckets. There are primary bucket pages and overflow pages.
Extendible Hashing

Why not **double** the **number** of buckets?

Note that reading and writing all pages is expensive!

**Idea:**

Use **directory of pointers** to buckets

On overflow, **double only the directory** (not the # of buckets)

Why does this help?

  - **Directory is much smaller** than the entire index file
  - Only **one page of data entries is split**
    - *No overflow page!* (caveat: duplicates w.r.t. the hash function)

Trick lies in how the hash function is adjusted!
Extendible Hashing

Directory: an array
Search for $k$:
- Apply hash function $h(k)$
- Take last **global depth** # bits of $h(k)$

Insert:
- If the bucket has space, insert, done
- If the bucket if full, **split** it, re-distribute – If necessary, double the directory
Example

what is the hash function?

the last two bits! so: \( k \mod 4 \)
Example: Insert 6

global depth: 6* = 0110

h

6* = 0110

directory

data pages
Example: Insert 6

6* = 0110

h

global depth:

directory

00
01
10
11

data pages

2
2
2
2

4* 12*
1* 13*
10*
15* 7*
Example: Insert 6

$6^* = 0110$

Global depth:

```
<table>
<thead>
<tr>
<th>Directory</th>
<th>Data Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>4* 12*</td>
</tr>
<tr>
<td>01</td>
<td>1* 13*</td>
</tr>
<tr>
<td>10</td>
<td>10* 6*</td>
</tr>
<tr>
<td>11</td>
<td>15* 7*</td>
</tr>
</tbody>
</table>
```
Example 2: Insert 9

9* = 1001

h

directory

data pages

4* 12*
1* 13*
10* 6*
15* 7*
Example 2: Insert 9

$9^* = 1001$

Now what??
Example 2: Insert 9

9*=1001

(1) double the directory
Example 2: Insert 9

(1) double the directory
(2) re-distribute the split bucket
Example 2: Insert 9

9\* = 1001

(1) double the directory
(2) re-distribute the split bucket
(3) connect corresponding buckets
Example 2: Insert 9

9* = 1001

h
Example 2: Insert 9

\[9^* = 1001\]

Do we have to re-distribute all?
Example 3: Insert 5

5* = 0101

h

000
001
010
011
100
101
110
111
Example 3: Insert 5

5* = 0101

What happens if we want to insert 17? Do we have to re-distribute all?

[17 → 10001] so, double the dir again!
Example 3: Insert 5

5* = 0101

h

do we have to double the directory every time we split a bucket?

when?
Example 3: Insert 14

14* = 1110

Diagram showing the insertion process.
Example 3: Insert 14

14\ast = 1110
Example 3: Insert 14

14*=1110
Example 3: Insert 14

14* = 1110

h
Notes on Extendible Hashing

How many disk accesses for equality search?
− One if directory fits in memory, else two

Directory grows in spurts, and, if the distribution of hash values is skewed, can grow large
Notes on Extendible Hashing

Do we ever need overflow pages?
- Multiple entries with same hash value cause problems!

Delete: Reverse of inserts
- Can merge with split image
- Can shrink the directory by half. When?
  - Each directory element points to same bucket as its split image
- Is shrinking/merging a good idea?
Hash Indexing

Static Hashing

Extendible Hashing

Linear Hashing
Linear Hashing

another dynamic hashing scheme

LH handles overflow chains without a directory

Idea: Use overflow pages, and split pages in a round-robin fashion
Example

this for information reasons!
it is not really kept.

<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>

<table>
<thead>
<tr>
<th></th>
<th>$4^* 8^*$</th>
<th>$1^* 13^*$</th>
<th>$10^*$</th>
<th>$15^* 7^*$</th>
</tr>
</thead>
</table>

Next bucket to split

what happens when we insert 5?

$h_0(5) = 01$
Example

this for information reasons!
it is not really kept.

\[
\begin{array}{cc}
  h_1 & h_0 \\
  000 & 00 \\
  001 & 01 \\
  010 & 10 \\
  011 & 11 \\
\end{array}
\]

what happens when we insert 5?

(1) 5 goes to an overflow page
Example

this for information reasons! it is not really kept.

\[
\begin{array}{ll}
  h_1 & h_0 \\
  000 & 00 \\
  001 & 01 \\
  010 & 10 \\
  011 & 11 \\
  100 & \\
\end{array}
\]

what happens when we insert 5?

(1) 5 goes to an overflow page
(2) we split the "next" page
Example

this for information reasons!
it is not really kept.

<table>
<thead>
<tr>
<th></th>
<th>h_1</th>
<th>h_0</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>001</td>
<td>01</td>
<td></td>
</tr>
<tr>
<td>010</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

what happens when we insert 5?

(1) 5 goes to an overflow page
(2) we split the “next” page
(3) we move the “next” pointer
Example: Insert 2

this for information reasons!

it is not really kept.

\[ h_1 \quad h_0 \]

\[
\begin{array}{c|c}
000 & 00 \\
001 & 01 \\
010 & 10 \\
011 & 11 \\
100 & \\
\end{array}
\]

Next bucket to split

\[ h_0(2) = 10 \]
Example: Insert 2

This for information reasons!
it is not really kept.

\[
\begin{array}{cc}
  h_1 & h_0 \\
  000 & 00 \\
  001 & 01 \\
  010 & 10 \\
  011 & 11 \\
  100 & \\
\end{array}
\]

\[
\begin{array}{c}
  8^* \\
  1^* 13^* \\
  10^* 2^* \\
  15^* 7^* \\
  4^* \\
\end{array}
\]

Next bucket to split
Example: Insert 3

This for information reasons!
it is not really kept.

<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>
<tr>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>

Next bucket to split

what happens when we insert 3? $h_0(3) = 11$
Example: Insert 3

this for information reasons!
it is not really kept.

$$h_1 \quad h_0$$

<table>
<thead>
<tr>
<th>000</th>
<th>00</th>
</tr>
</thead>
<tbody>
<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></td>
</tr>
</tbody>
</table>

$$\begin{array}{c}
8* \\
1* 13* \\
10* 2* \\
15* 7* \\
4*
\end{array}$$

Next bucket to split

what happens when we insert 3?

(1) 3 goes to an overflow page
Example: Insert 3

this for information reasons!
it is not really kept.

\[
\begin{array}{c|c|c|c|c|c|c|c}
    & h_1 & h_0 & & & & & \\
\hline
000 & 00 & & & & & & \\
001 & 01 & & & & & & \\
010 & 10 & & & & & & \\
011 & 11 & & & & & & \\
100 & & & & & & & \\
101 & & & & & & & \\
\end{array}
\]

Next bucket to split

what happens when we insert 3?

(1) 3 goes to an overflow page
(2) we split the “next” page
Example: Insert 3

this for information reasons!
it is not really kept.

\begin{array}{ll}
000 & 00 \\
001 & 01 \\
010 & 10 \\
011 & 11 \\
100 & \\
101 & \\
\end{array}

Next bucket to split

what happens when we insert 3?

(1) 3 goes to an overflow page
(2) we split the ”next” page
(3) we move the ”next” pointer
Linear Hashing

$h_0, h_1, h_2 \ldots$ can be more general hash functions

when $h_0$ hits on a split buffer we employ $h_1$ and we have to look in both buffers

if the second is also split we use $h_2$ and so on

Benefit: buckets are split round-robin
  → no long chains
Hash Indexing

Hash indexes: best for equality searches

*Static Hashing* can lead to long overflow chains

*Extendible Hashing*

  avoids overflow pages by splitting a bucket when full directory to keep track of buckets

**BUT**

dir. can get too large (>memory) when data is skewed

*Linear Hashing*

  avoids directory by splitting buckets round-robin
  uses overflow pages
  overflow pages not likely to be long