CS460: Intro to Database Systems

Class 11: 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. Actual data record (with key value $k$)
2. $<k, \text{rid of matching data record}>$
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 $[0..M-1]$.

Where $M$ is the number of buckets (pages) available to our index.

- Ideally a hash function maps the search keys uniformly in $[0, ..., M-1]$.
- 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

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

> h(key) mod M
> 0
> 1
> ...
> M-1

Primary bucket pages

Overflow pages
Static Hashing (Contd.)

Buckets contain **data entries**

Hash function on *search key* field of record $r$

Must distribute values over range $0 \ldots M-1$

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!)

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?

\[ h(k) \mod M = \text{bucket to insert data entry with key } k \ (M: \#\text{buckets}) \]
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 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 is full, **split** it, re-distribute – If necessary, double the directory
Example

13* = 1101

what is the hash function?
Example: Insert 6

6* = 0110

Global depth: 2

Directory:
- 00: 2
- 01: 2
- 10: 2
- 11: 2

Data pages:
- 00: 4* 12*
- 01: 1* 13*
- 10: 10*
- 11: 15* 7*
Example: Insert 6

6* = 0110

Global depth:

H: 01

00: 2
01: 2
10: 2
11: 2

Directory

Data pages:

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

6* = 0110

Global depth:

```
  | 6* = 0110
  |
  v

00  2
01  1
10  2
11  2

Directory

Data pages:

- 00: 4* 12*
- 01: 1* 13*
- 10: 10* 6*
- 11: 15* 7*
```
Example 2: Insert 9

\[ 9^* = 1001 \]

\begin{align*}
\text{directory} & \\
00 & \rightarrow 2 \\
01 & \rightarrow 2 \\
10 & \rightarrow 2 \\
11 & \rightarrow 2 \\
\text{data pages} & \\
2 & \rightarrow 4* 12* \\
2 & \rightarrow 1* 13* \\
2 & \rightarrow 10* 6* \\
2 & \rightarrow 15* 7*
\end{align*}
Example 2: Insert 9

9* = 1001

directory

data pages

now what??
Example 2: Insert 9

9* = 1001

(1) double the directory
Example 2: Insert 9

9* = 1001

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

- $h(9) = 1001$

(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\star = 1001

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

$5^* = 0101$

$4^* 12^*$

$1^* 9^*$

$10^* 6^*$

$15^* 7^*$

$13^*$

$h$
Example 3: Insert 5

5\ast = 0101

what happens if we want to insert 17?
do we have to re-distribute all?

\[ 17 \rightarrow 10001 \] 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?
Example 3: Insert 14

14* = 1110
Example 3: Insert 14

h

14* = 1110
Example 3: Insert 14

\(14^* = 1110\)
Example 3: Insert 14

$14^* = 1110$

Diagram showing the insertion of 14 into a data structure with a binary search tree-like structure.
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.

\[
\begin{array}{c|c|c}
  & h_0 & h_1 \\
 000 & 00 & 4^* 8^* \\
001 & 01 & 1^* 13^* \\
010 & 10 & 10^* \\
011 & 11 & 15^* 7^* \\
\end{array}
\]

Next bucket to split

what happens when we insert 5?
Example

this for information reasons!
it is not really kept.

<table>
<thead>
<tr>
<th>h₁</th>
<th>h₀</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>

```
4*8*
1*13*
10*
15*7*
```

Next bucket to split

what happens when we insert 5?

(1) 5 goes to an overflow page
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>
<tr>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>

Next bucket to split

what happens when we insert 5?

(1) 5 goes to an overflow page
(2) we split the “next” page
this for information reasons! it is not really kept.

\[
\begin{array}{c|c|c}
\text{h}_1 & \text{h}_0 \\
000 & 00 \\
001 & 01 \\
010 & 10 \\
011 & 11 \\
100 & \\
\end{array}
\]

what happens when we insert 5?

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

this for information reasons!
it is not really kept.

<table>
<thead>
<tr>
<th>h₁</th>
<th>h₀</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
Example: Insert 2

For information reasons, it is not really kept.

Next bucket to split

8*
1* 13*
10* 2*
15* 7*
4*
Example: Insert 3

This for information reasons!
it is not really kept.

what happens when we insert 3?
Example: Insert 3

this for information reasons!
it is not really kept.

Next bucket to split

what happens when we insert 3?

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

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.

<table>
<thead>
<tr>
<th>h₁</th>
<th>h₀</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>
<tr>
<td>101</td>
<td></td>
</tr>
</tbody>
</table>

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
- 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