Outline/summary

• Conventional Indexes
  • Sparse vs. dense
  • Primary vs. secondary

• B trees
  • B+trees vs. indexed sequential

• Hashing schemes  -->  Next
Hashing

key $\rightarrow$ h(key)

Buckets (typically 1 disk block)
Two alternatives

(1) key $\rightarrow h(\text{key}) \quad \rightarrow \quad \text{records}$
Two alternatives

(2) key $\rightarrow h(\text{key})$

- Alt (2) for “secondary” search key
Example hash function

• Key = ‘x₁ x₂ … xₙ’  n byte character string

• Have b buckets

• h: add x₁ + x₂ + ..... xₙ
  – compute sum modulo b
This may not be best function ...

Read Knuth Vol. 3 if you really need to select a good function.

Good hash function:

Expected number of keys/bucket is the same for all buckets
Within a bucket:

- Do we keep keys sorted?
  - Yes, if CPU time critical
    & Inserts/Deletes not too frequent
Next: example to illustrate inserts, overflows, deletes

\[ h(K) \]
EXAMPLE  2 records/bucket

INSERT:

h(a) = 1
h(b) = 2
h(c) = 1
h(d) = 0
h(e) = 1
EXAMPLE: deletion

Delete:

e
f
c

<table>
<thead>
<tr>
<th>0</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>c (\rightarrow) d</td>
</tr>
<tr>
<td>3</td>
<td>e (\rightarrow) f (\rightarrow) g</td>
</tr>
</tbody>
</table>

maybe move “g” up
Rule of thumb:

• Try to keep space utilization between 50% and 80%

  Utilization = \frac{\# \text{ keys used}}{\text{total \# keys that fit}}

• If < 50%, wasting space

• If > 80%, overflows significant depends on how good hash function is & on \# keys/bucket
How do we cope with growth?

- Overflows and reorganizations
- Dynamic hashing
  - Extensible
  - Linear
Extensible hashing: two ideas

(a) Use $i$ of $b$ bits output by hash function $h(K)$ → 00110101

use $i$ → grows over time....
(b) Use directory

\[ h(K)[i \ ] \rightarrow \text{to bucket} \]
Example: \( h(k) \) is 4 bits; 2 keys/bucket

Insert 1010

New directory
Example continued

i = 2

Insert:
0111
0000
Example continued

Insert: 1001

1010

1001

1100
Extensible hashing: deletion

- No merging of blocks
- Merge blocks and cut directory if possible
  (Reverse insert procedure)
Deletion example:

- Run thru insert example in reverse!
Extensible hashing

+ Can handle growing files
  - with less wasted space
  - with no full reorganizations

- Indirection
  (Not bad if directory in memory)

- Directory doubles in size
  (Now it fits, now it does not)
Linear hashing

- Another dynamic hashing scheme

Two ideas:
(a) Use \( i \) low order bits of hash

(b) Number of buckets in use grows linearly

Constraint: \( 2^{i-1} \leq n+1 < 2^i \)

(We take \( n \) to be the id of the largest bucket in use, starting at 0.)
Example  \(b=4\) bits,  \(i=2\),  2 keys/bucket

\[
\begin{array}{cccc}
0000 & 0101 & \text{• insert 0101} & \\
1010 & 1111 & \text{• can have overflow chains!} & \\
00 & 01 & 10 & 11
\end{array}
\]

\(n = 01\) (number of last bucket in use)

**Rule**  If \(h(k)[i] \leq n\), then

- look at bucket \(h(k)[i]\)
- else, look at bucket \(h(k)[i] - 2^{i-1}\)
Example \( b=4 \) bits, \( i=2 \), \( 2 \) keys/bucket

- Insert 1110
- Bucket \( h(k)[i] - 2^{i-1} \) is the bucket whose \( i \)th bit is flipped in binary

\[
\begin{array}{c|c|c|c}
0000 & 0101 & 1111 & \\
1010 & 1111 & & \\
00 & 01 & 10 & 11 \\
\end{array}
\]

\( n = 01 \) (number of last bucket in use)

Rule
If \( h(k)[i] \leq n \), then
look at bucket \( h(k)[i] \)
else, look at bucket \( h(k)[i] - 2^{i-1} \)
Example  \( b=4 \text{ bits, } i =2, \text{ 2 keys/bucket} \)

- insert 0101

\[
\begin{array}{c|c|c|c|c}
0000 & 0101 & 1010 & 1111 \\
1010 & 0101 & 1111 & 1111 \\
\hline
00 & 01 & 10 & 11 \\
\end{array}
\]

\[ n = 01 \]

Future growth buckets

Rule

If \( h(k)[i] \leq n \), then look at bucket \( h(k)[i] \)

else, look at bucket \( h(k)[i] - 2^{i-1} \)
Example Continued:
How to grow beyond this?

\[ i = 2^3 \]

Constraint: \(2^{i-1} \leq n+1 < 2^i\)

<table>
<thead>
<tr>
<th>0000</th>
<th>0101</th>
<th>1010</th>
<th>1111</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>011</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ n = 11 \]

Rule

If \( h(k)[i] \leq n \), then

look at bucket \( h(k)[i] \)

else, look at bucket \( h(k)[i] - 2^{i-1} \)
When do we expand file?

- Keep track of: \( \frac{\# \text{ records}}{\# \text{ buckets}} - U \)

- If \( U > \text{threshold} \) then increase \( n \) (and maybe \( i \))
Summary

Linear Hashing

+ Can handle growing files
  - with less wasted space
  - with no full reorganizations

+ No indirection like extensible hashing

- Can still have overflow chains
Example: BAD CASE

Very full

Very empty

Need to move $n$ here...

Would waste space...
Summary

Hashing
- How it works
- Dynamic hashing
  - Extensible
- Linear
B+trees vs Hashing

- Hashing good for probes given key
  e.g.,
  
  ```
  SELECT ...
  FROM R
  WHERE R.A = 5
  ```
B+Trees vs Hashing

- INDEXING (Including B Trees) good for
  Range Searches:
  e.g., SELECT
  FROM R
  WHERE R.A > 5