CS 525: Advanced Database Organization

05: Hashing and More

Boris Glavic

Slides: adapted from a course taught by Hector Garcia-Molina, Stanford InfoLab
Hashing

key → h(key)

Buckets (typically 1 disk block)
Two alternatives

(1) key $\rightarrow h(key)$
Two alternatives

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

(2) key $\rightarrow$ h(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.
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:

\begin{align*}
  h(a) &= 1 \\
  h(b) &= 2 \\
  h(c) &= 1 \\
  h(d) &= 0
\end{align*}
EXAMPLE  2 records/bucket

INSERT:
\[ h(a) = 1 \]
\[ h(b) = 2 \]
\[ h(c) = 1 \]
\[ h(d) = 0 \]
\[ h(e) = 1 \]
EXAMPLE 2 records/bucket

INSERT:

\[
\begin{align*}
    h(a) &= 1 \\
    h(b) &= 2 \\
    h(c) &= 1 \\
    h(d) &= 0 \\
    h(e) &= 1
\end{align*}
\]
EXAMPLE: deletion

Delete:

\[ \begin{array}{c}
e \\
f \\
\end{array} \]

\[ \begin{array}{c}
a \\
b \\
c \\
e \\
f \\
g \\
\end{array} \]

\[ \begin{array}{c}
d \\
\end{array} \]
EXAMPLE: deletion

Delete:

e
f
c

\begin{itemize}
\item 0: a
\item 1: b, c
\item 2: e
\item 3: f, g
\end{itemize}

\begin{itemize}
\item maybe move “g” up
\end{itemize}
EXAMPLE: deletion

Delete:

e
f
c

c 1
  b
  c  d
  e
  f
  g

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}}
Rule of thumb:

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

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

• If < 50%, wasting space
• If > 80%, overflows significant
  depends on how good hash function is & on \# \text{ keys/bucket}
How do we cope with growth?

- Overflows and reorganizations
- Dynamic hashing
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

$h(K) \rightarrow \begin{array}{c} 00110101 \end{array}$

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

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

$i = \begin{array}{c}
1 \\
0001 \\
1 \\
1001 \\
1100 \\
\end{array}$

Insert 1010
Example: $h(k)$ is 4 bits; 2 keys/bucket

Insert 1010
Example: \( h(k) \) is 4 bits; 2 keys/bucket

Insert 1010

New directory
Example continued

Insert:

0111
0000

CS 525
Notes 5 - Hashing
Example continued

Insert:
0111
0000

i = 2
00
01
10
11
Example continued

Insert:

0111

0000
Example continued

Insert: 1001

\[ i = 2 \]

0000 2

0001

0111 2

1001 2

1010

1100 2
Example continued

Insert: 1001
Example continued

Insert:

1001

0000 2
0001 2
0111 2
1001 3
1001 3
1100 2

i = 3

000
001
010
011
100
101
110
111
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!
Note: Still need overflow chains

• Example: many records with duplicate keys

insert 1100

if we split:

\[ \begin{array}{c}
1 \\
1101 \\
1100 \\
\end{array} \quad \begin{array}{c}
2 \\
1100 \\
1100 \\
\end{array} \]
Solution: overflow chains

insert 1100

add overflow block:
Summary
Extensible hashing

① Can handle growing files
  - with less wasted space
  - with no full reorganizations
Summary 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

```
01110101
```

$b$

grows$\longleftarrow i$
Linear hashing

- Another dynamic hashing scheme

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

(b) File grows linearly
Example  $b=4$ bits,  $i =2$,  2 keys/bucket

<table>
<thead>
<tr>
<th>0000</th>
<th>0101</th>
<th>1111</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>1010</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Future growth buckets

$m = 01$ (max used block)
Example \( b=4 \) bits, \( i=2 \), 2 keys/bucket

\[
\begin{array}{cccc}
0000 & 0101 & \text{Future growth buckets} & \\
1010 & 1111 & & \\
00 & 01 & 10 & 11
\end{array}
\]

\( m = 01 \) (max used block)

**Rule**  
If \( h(k)[i] \leq m \), 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 0101

<table>
<thead>
<tr>
<th>0000</th>
<th>0101</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1010</td>
<td>1111</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\( m = 01 \) (max used block)

Rule

If \( h(k)[i] \leq m \), 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 0101
- can have overflow chains!

$m = 01$ (max used block)

Rule
If $h(k)[i] \leq m$, then look at bucket $h(k)[i]$ else, look at bucket $h(k)[i] - 2^{i-1}$
Note

- In textbook, $n$ is used instead of $m$
- $n = m + 1$

$m = 01$ (max used block)

$n = 10$

Future growth buckets
Example $b=4$ bits, $i=2$, 2 keys/bucket

$m = 01$ (max used block)
Example  \( b=4 \) bits,  \( i = 2 \), 2 keys/bucket

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

\( m = 01 \) (max used block)
Example  $b=4$ bits,  $i=2$,  2 keys/bucket

- insert 0101

0000 0101 1010 1111

00 01 10 11

$m = 01$ (max used block)
Example \( b=4 \) bits, \( i=2 \), 2 keys/bucket

- insert 0101

Future growth buckets

\[ m = 01 \] (max used block)
Example  \( b=4 \) bits,  \( i=2 \), 2 keys/bucket

- Insert 0101

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

\( m = 01 \) (max used block)
Example Continued: How to grow beyond this?

\[ i = 2 \]

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

\[ m = 11 \text{ (max used block)} \]
Example Continued: How to grow beyond this?

\[ i = 2^3 \]

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0101</td>
<td>1010</td>
<td>1111</td>
</tr>
<tr>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
</tr>
<tr>
<td>100</td>
<td>101</td>
<td>110</td>
<td>111</td>
</tr>
</tbody>
</table>

\[ m = 11 \text{ (max used block)} \]
Example Continued: How to grow beyond this?

\[ i = 2 \] 

\[
\begin{array}{c|c|c|c|c}
0000 & 0101 & 1010 & 1111 & \ldots \\
0101 & 1010 & 1111 & & \\
000 & 001 & 010 & 011 & 100 \\
010 & 101 & 110 & 111 & \\
010 & 101 & 110 & 111 & \\
100 & & & & \\
\end{array}
\]

\[ m = 11 \text{ (max used block)} \]
Example Continued: How to grow beyond this?

\[ i = 2 \]

\[ m = 11 \text{ (max used block)} \]
When do we expand file?

- Keep track of: \( \frac{\text{# used slots}}{\text{total # of slots}} = U \)
When do we expand file?

- Keep track of: \[
\frac{\text{# used slots}}{\text{total # of slots}} = U
\]

- If \( U > \) threshold then increase \( m \)
  (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

Would waste space...

$m$ here...
Summary

Hashing

- How it works
- Dynamic hashing
  - Extensible
  - Linear
Next:

- Indexing vs Hashing
- Index definition in SQL
- Multiple key access
Indexing vs Hashing

• Hashing good for probes given key
e.g., SELECT ...
   FROM R
   WHERE R.A = 5

-> Point Queries
Indexing vs Hashing

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

-> Range Queries
Index definition in SQL

- **Create** index name on rel (attr)
- **Create unique** index name on rel (attr)

  defines candidate key

- **Drop** INDEX name
Note CANNOT SPECIFY TYPE OF INDEX (e.g. B-tree, Hashing, ...) OR PARAMETERS (e.g. Load Factor, Size of Hash,...) ... at least in standard SQL...

Vendor specific extensions allow that
Note  ATTRIBUTE LIST ⇒ MULTIKEY INDEX

(next)
e.g., CREATE INDEX foo ON R(A,B,C)
Multi-key Index

Motivation: Find records where

DEPT = "Toy" AND SAL > 50k
Strategy I:

- Use one index, say Dept.
- Get all Dept = “Toy” records and check their salary

![Diagram](image)
Strategy II:

- Use 2 Indexes; Manipulate Pointers

Toy $\rightarrow$ | | | | | | Sal $\leftarrow$ | | | | | | > 50k
Strategy III:

- Multiple Key Index

One idea:
Example

<table>
<thead>
<tr>
<th>Dept</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Art</td>
<td></td>
</tr>
<tr>
<td>Sales</td>
<td></td>
</tr>
<tr>
<td>Toy</td>
<td></td>
</tr>
</tbody>
</table>

Example

Record

<table>
<thead>
<tr>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Joe</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DEPT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sales</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>15k</td>
</tr>
</tbody>
</table>

Salary Index

<table>
<thead>
<tr>
<th>10k</th>
</tr>
</thead>
<tbody>
<tr>
<td>15k</td>
</tr>
<tr>
<td>17k</td>
</tr>
<tr>
<td>21k</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>12k</th>
</tr>
</thead>
<tbody>
<tr>
<td>15k</td>
</tr>
<tr>
<td>15k</td>
</tr>
<tr>
<td>19k</td>
</tr>
</tbody>
</table>

12k
15k
15k
19k
For which queries is this index good?

- Find RECs Dept = “Sales” ∧ SAL=20k
- Find RECs Dept = “Sales” ∧ SAL ≥ 20k
- Find RECs Dept = “Sales”
- Find RECs SAL = 20k
Interesting application:

- Geographic Data

DATA:

\(<X_1, Y_1, \text{Attributes}>\)

\(<X_2, Y_2, \text{Attributes}>\)

\vdots
Queries:

- What city is at \(<Xi,Yi>\)?
- What is within 5 miles from \(<Xi,Yi>\)?
- Which is closest point to \(<Xi,Yi>\)?
Example
Example

10  20

\[ \begin{array}{c}
\text{Example}\\
10 & 20
\end{array} \]
Example
Example

The diagram illustrates a hash table with keys and values. The keys are mapped to values using a hash function. The table shows how keys are distributed across different buckets. The diagram includes labels for clarity.
Example

```
Example

```

```
```

```
```
Example

- Search points near f
- Search points near b
Queries

- Find points with $Y_i > 20$
- Find points with $X_i < 5$
- Find points “close” to $i = <12,38>$
- Find points “close” to $b = <7,24>$
Next

- Even more index structures 😊