Hashing and More

Two alternatives

1. key → h(key) → records

2. key → h(key) → Index

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

- Key = ‘$x_1 \ x_2 \ ... \ x_n$’ $n$ byte character string
- Have $b$ buckets
- $h$: add $x_1 + x_2 + \ldots + x_n$
  - compute sum modulo $b$

This may not be best function ...
_indexes to select a good function

Good hash

Expected number of function:

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:**
- \( h(a) = 1 \)
- \( h(b) = 2 \)
- \( h(c) = 1 \)
- \( h(d) = 0 \)
- \( h(e) = 1 \)

Record may be full tuple or key/pointer combo

EXAMPLE: deletion

**Delete:**
- \( e \)
- \( f \)
- \( c \)

How do we cope with growth?

- Overflows and reorganizations
- Dynamic hashing
  - Extensible
  - Linear

Rule of thumb:
- Try to keep space utilization between 50% and 80%
  \[ \text{Utilization} = \frac{\text{# keys used}}{\text{total # keys that fit}} \]
- If < 50%, wasting space
- If > 80%, overflows significant
  - depends on how good hash function is & on # keys/bucket

---

3
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

Initial directory

New directory

Insert 1010

Example continued

Insert:

0111
0000

0000
0001
0111
0110
Example continued

Deletion example:
- Run thru insert example in reverse!

Extensible hashing: deletion
- No merging of blocks
- Merge blocks and cut directory if possible (Reverse insert procedure)

Note: Still need overflow chains
- Example: many records with duplicate keys
  insert 1100
  if we split:
Solution: overflow chains

insert 1100
add overflow block:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1100</td>
</tr>
<tr>
<td>1</td>
<td>1100</td>
</tr>
</tbody>
</table>

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

(b) File grows linearly

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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
</tr>
<tr>
<td>10</td>
<td>11</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 $h(k)[i] - 2^{i-1}$
**Note**
- In textbook, \( n \) is used instead of \( m \)
- \( n = m + 1 \)

---

**Example**  \( b = 4 \) bits, \( i = 2 \), 2 keys/bucket

- Insert 0101

---

**Example Continued: How to grow beyond this?**

\[ i = 2^3 \]

---

**When do we expand file?**

- Keep track of: \( \frac{\text{# used slots}}{\text{total # of slots}} = U \)
- If \( U > \text{threshold} \) 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

Need to move

m here...

Would waste space...

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., \( \text{SELECT} \ldots \text{FROM} \ R \ \text{WHERE} \ R.A = 5 \)

**Indexing vs Hashing**

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

**Index definition in SQL**

- \( \text{CREATE INDEX} \ \text{name} \ \text{ON} \ \text{rel} (\text{attr}) \)
- \( \text{CREATE UNIQUE INDEX} \ \text{name} \ \text{ON} \ \text{rel} (\text{attr}) \)
  - defines candidate key
- \( \text{DROP INDEX} \ \text{name} \)

**Note**

- CANNOT SPECIFY TYPE OF INDEX (e.g. B-Tree, Hashing, ...)
- OR PARAMETERS (e.g. Load Factor, Size of Hash,...)
  - ... at least in SQL...
Note: \textbf{ATTRIBUTE LIST} $\Rightarrow$ \textbf{MULTIKEY INDEX} (next)  
e.g., \texttt{CREATE INDEX foo ON R(A,B,C)}

\textbf{Multi-key Index}

Motivation: Find records where  
\texttt{DEPT = "Toy" AND SAL > 50k}

\textbf{Strategy I:}
\begin{itemize}
  \item Use one index, say Dept.
  \item Get all Dept = "Toy" records and check their salary
\end{itemize}

\textbf{Strategy II:}
\begin{itemize}
  \item Use 2 Indexes; Manipulate Pointers
  
  Toy $\rightarrow$ \underline{I1} \underline{I1} \underline{I1} \underline{I1} \underline{I1} $\rightarrow$ Sal $>$ 50k
Strategy III:

- Multiple Key Index

One idea:

For which queries is this index good?

- Find RECs Dept = “Sales” \land SAL=20k
- Find RECs Dept = “Sales” \land SAL \geq 20k
- Find RECs Dept = “Sales”
- Find RECs SAL = 20k

Interesting application:

- Geographic Data

Interesting application:

- Geographic Data

DATA:

\langle X_1, Y_1, \text{Attributes} \rangle
\langle X_2, Y_2, \text{Attributes} \rangle
\vdots
Queries:

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

Queries

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

- Many types of geographic index structures have been suggested
  - \( kd \)-Trees (very similar to what we described here)
  - Quad Trees
  - \( R \) Trees
  - ...
Two more types of multi key indexes

- Grid
- Partitioned hash

CLAIM

- Can quickly find records with
  - key 1 = V_i \land Key 2 = X_j
  - key 1 = V_i
  - key 2 = X_j

- And also ranges...
  - E.g., key 1 \geq V_i \land key 2 < X_j

Grid Index

- Key 1
  - V_1
  - V_2
  - V_n

- Key 2
  - X_1 X_2 ...... X_n

To records with key1=V_3, key2=X_2

But there is a catch with Grid Indexes!

- How is Grid Index stored on disk?

  Like Array...

Problem:

- Need regularity so we can compute position of \langle V_i, X_j \rangle entry
Solution: Use Indirection

- Use Indirection
- B1, B2, B3, B4
- B4 contains pointers to buckets

With indirection:
- Grid can be regular without wasting space
- We do have price of indirection

Can also index grid on value ranges

- Salary
  - 0-20K: 1
  - 20K-50K: 2
  - 50K-∞: 3

Grid files

- Good for multiple-key search
- Space, management overhead (nothing is free)
- Need partitioning ranges that evenly split keys
Partitioned hash function

Idea: 010110 1110010

Key1 → h1 → h2 → Key2

EX:

Insert <Fred, toy, 10k>, <Joe, sales, 10k>, <Sally, art, 30k>

• Find Emp. with Dept. = Sales ∧ Sal=40k

• Find Emp. with Sal=30k
\begin{align*}
\text{h1(toy)} &= 0 \quad 000 \quad \text{<Fred>}\\
\text{h1(sales)} &= 1 \quad 001 \quad \text{<Joe><Jan>}\\
\text{h1(art)} &= 1 \quad 010 \quad \text{<Mary>}\\
\text{h2(10k)} &= 01 \quad 100 \quad \text{<Sally>}\\
\text{h2(20k)} &= 11 \quad 101 \quad \text{<Tom><Bill>}\\
\text{h2(30k)} &= 01 \quad 110 \quad \text{<Andy>}\\
\text{h2(40k)} &= 00 \quad 111 \\
\end{align*}

- Find Emp. with Dept. = Sales
  - look here

\section*{Summary}

\textbf{Post hashing discussion:}
- Indexing vs. Hashing
- SQL Index Definition
- Multiple Key Access
  - Multi Key Index
    - Variations: Grid, Geo Data
  - Partitioned Hash

\section*{Reading Chapter 14}

- Skim the following sections:
  - Sections 14.6.6, 14.6.7, 14.6.8
  - Sections 14.7.2, 14.7.3, 14.7.4
- Read the rest

\section*{The BIG picture....}

- Chapter 13: Storage, records, blocks...
- Chapter 14: Access Mechanisms
  - Indexes
  - B-Trees
  - Hashing
  - Multi key
- Chapter 15 & 16: Query Processing

\textbf{NEXT}