Hashing

key → h(key)

Buckets (typically 1 disk block)

Two alternatives

(1) key → h(key)

records

Two alternatives

(2) key → h(key)

Index

• 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

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
- g

Rule of thumb:
- Try to keep space utilization between 50% and 80%
  - Utilization = \# keys used / 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

\[ h(K) \rightarrow 00110101 \]

use $i \rightarrow$ 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

Insert:

0111
0000

Example continued

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!

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 low order bits of hash
    (b) File grows linearly

Example $b=4$ bits, $i=2$, 2 keys/bucket
- Future growth buckets
- $m=01$ (max used block)
- Rule: If $h(k)[i] \leq m$, then look at bucket $h(k)[i] - 2^i - 1$
- can have overflow chains!

Example Continued: How to grow beyond this?
- Future growth buckets
- $m=11$ (max used block)
- Rule: If $h(k)[i] \leq m$, then look at bucket $h(k)[i] - 2^i - 1$
• 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
- Very empty
- 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.,
  \[
  \text{SELECT ...}
  \text{FROM R}
  \text{WHERE R.A = 5}
  \]
**Indexing vs Hashing**

- INDEXING (Including B Trees) good for Range Searches:
  
  e.g.,

  ```sql
  SELECT
  FROM R
  WHERE R.A > 5
  ```

**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 SQL...

**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

 attribute list \( \Rightarrow \) multikey index
  (next)

  e.g., `CREATE INDEX foo ON R(A,B,C)`
Strategy II:
- Use 2 Indexes; Manipulate Pointers

Toy -> Index 1 -> Sal > 50k

Strategy III:
- Multiple Key Index

One idea:

Example Record
Name=Joe
DEPT=Sales
SAL=15k

Example

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

<table>
<thead>
<tr>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

DATA:
- <X1,Y1, Attributes>
- <X2,Y2, Attributes>
- ...

Points

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

Queries

• Find points with Yi > 20
• Find points with Xi < 5
• Find points “close” to i = <12,38>
• Find points “close” to b = <7,24>

Many types of geographic index structures have been suggested
• Quad Trees
• R Trees

Two more types of multi key indexes
• Grid
• Partitioned hash

CLAIM
• Can quickly find records with
  – key 1 = Vi ⋀ Key 2 = Xj
  – key 1 = Vi
  – key 2 = Xj
• And also ranges....
  – E.g., key 1 ≥ Vi ⋀ key 2 < Xj

Grid Index
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 \( <V_i, X_j> \) entry

Solution: Use Indirection

- Grid only 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

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

Partitioned hash function

Idea:

\[
\begin{align*}
\text{Key1} &: \quad h_1 \quad h_2 \quad \text{Key2} \\
010110 &\quad 1110010
\end{align*}
\]
EX:

```
<table>
<thead>
<tr>
<th>h1(toy)</th>
<th>h1(sales)</th>
<th>h1(art)</th>
<th>h2(10k)</th>
<th>h2(20k)</th>
<th>h2(30k)</th>
<th>h2(40k)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 000</td>
<td>0 001</td>
<td>0 010</td>
<td>0 011</td>
<td>01 100</td>
<td>11 101</td>
<td>01 110</td>
</tr>
<tr>
<td>&lt;Fred&gt;</td>
<td></td>
<td></td>
<td></td>
<td>&lt;Joe&gt;</td>
<td>&lt;Sally&gt;</td>
<td></td>
</tr>
</tbody>
</table>
```

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

```
<table>
<thead>
<tr>
<th>h1(toy)</th>
<th>h1(sales)</th>
<th>h1(art)</th>
<th>h2(10k)</th>
<th>h2(20k)</th>
<th>h2(30k)</th>
<th>h2(40k)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 000</td>
<td>0 001</td>
<td>0 010</td>
<td>0 011</td>
<td>01 100</td>
<td>11 101</td>
<td>01 110</td>
</tr>
<tr>
<td>&lt;Fred&gt;</td>
<td></td>
<td></td>
<td></td>
<td>&lt;Sally&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

• Find Emp. with Sal=30k

```
<table>
<thead>
<tr>
<th>h1(toy)</th>
<th>h1(sales)</th>
<th>h1(art)</th>
<th>h2(10k)</th>
<th>h2(20k)</th>
<th>h2(30k)</th>
<th>h2(40k)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 000</td>
<td>0 001</td>
<td>0 010</td>
<td>0 011</td>
<td>01 100</td>
<td>11 101</td>
<td>01 110</td>
</tr>
<tr>
<td>&lt;Fred&gt;</td>
<td></td>
<td></td>
<td></td>
<td>&lt;Sally&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

• Find Emp. with Dept. = Sales

Summary

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

Reading Chapter 14

• Skim the following sections:
  - 14.3.6, 14.3.7, 14.3.8
  - 14.4.2, 14.4.3, 14.4.4
• Read the rest
The BIG picture....

- Chapters 11 & 12: Storage, records, blocks...
- Chapter 13 & 14: Access Mechanisms
  - Indexes
  - B trees
  - Hashing
  - Multi key
- Chapter 15 & 16: Query Processing

NEXT