Hashing?

- Locating the storage block of a record by the hash value $h(k)$ of its key $k$
- Normally really fast
  - records (often) located by a single disk access

Two alternatives

1. Hash value determines the storage block directly
   - to implement a primary index

2. Records located indirectly via index buckets
   - for a secondary index

Example hash function

- Key = ‘$x_1 x_2 ... x_n$’ n byte character string
- Have b buckets
- $h = (x_1 + x_2 + ... + x_n) \mod b$
  - $\in \{0, 1, ..., b-1\}$

Hashing?

- Locating the storage block of a record by the hash value $h(k)$ of its key $k$
- Normally really fast
  - records (often) located by a single disk access
This may not be best function ...

Good hash function: Expected number of keys/bucket is the same for all buckets

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

Next: example to illustrate inserts, overflows, deletes

h(K)

EXAMPLE 2 records/bucket

INSERT:

| h(a) = 1 | 0 | d |
| h(b) = 2 | 1 | a |
| h(c) = 1 | 2 | b |
| h(d) = 0 | 3 | |
| h(e) = 1 | 1 | e |

EXAMPLE: deletion

Delete:

| h(a) = 1 | 0 | g |
| h(b) = 2 | 1 | e |
| h(c) = 1 | 2 | f |
| h(d) = 0 | 3 | c |

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: # of buckets may vary
- Extensible
- Linear
- also others ...

DBMS 2001 Notes 4.2: Hashing 7

DBMS 2001 Notes 4.2: Hashing 8

DBMS 2001 Notes 4.2: Hashing 9

DBMS 2001 Notes 4.2: Hashing 10

DBMS 2001 Notes 4.2: Hashing 11

DBMS 2001 Notes 4.2: Hashing 12
Extensible Hashing: two ideas

(a) Use \( i \) of \( b \) bits output by hash function
\[
h(K) \rightarrow \begin{array}{c}
00110101
\end{array}
\]
For example, \( b=32 \)
\( \text{use } i \rightarrow \text{grows over time...} \)

(b) Use directory
\[
h(K)[i] \rightarrow \text{to bucket}
\]
Directory contains \( 2^i \) pointers to buckets, and stores \( i \).
Each bucket stores \( j \), indicating \#bits used for placing the records in this block \( (j \leq i) \)

Extensible Hashing: Insertion

• If there’s room in bucket \( h(k)[i] \), place record there; Otherwise ...
• If \( j=i \), set \( i=i+1 \) and double the directory
• If \( j<i \), split the block in two, distribute records among them now using \( j+1 \) bits of \( h(k) \);
  (Repeat until some records end up in the new bucket); Update pointers of bucket array
• See the next example

Example: \( h(k) \) is 4 bits; 2 keys/block

Example continued

Example continued
Extensible hashing: deletion

- Reverse insert procedure ...
- Example: Walk thru insert example in reverse!

Extensible hashing:

- Can handle growing files without full reorganizations
- Only one data block examined
- Indirection (Not bad if directory in memory)
- Directory doubles in size (First it fits in memory, then it does not -> sudden performance degradation)

Linear hashing: grow # of buckets by one

Two ideas:
(a) Use i low order bits of hash to address buckets
(b) File grows linearly

-> No bucket directory needed

Linear Hashing: Parameters

- n: number of buckets in use
  - buckets numbered 0...n-1
- i: number of bits of h(k) used to address buckets
  $i = \log_2(n)$
- r: number of records in hash table
  - ratio $r/n$ limited to fit an avg bucket in a block
  - next example: $r \leq 1.7n$, and block holds 2 records
  $\Rightarrow$ AVG bucket occupancy is $\leq 1.7/2 = 0.85$ of a block

Example: 2 keys/block, $b=4$ bits, $n=2$, $i=1$

- insert 0101

$00\quad 01\quad 11\quad 10$

$\downarrow$ now $r=4 > 1.7n$
$\Rightarrow$ get new bucket 10
and distribute keys btw buckets 00 and 10:

$00\quad 01\quad 10$

Note: If $h(k)[i] = \{a_1, a_2\} < n$, then
look at bucket $h(k)[i]$; else
look at bucket $h(k)[i] \cdot 2^{-i} = (0a_1, a_2)$
n=3, i =2; insert 0001:

- can have overflow chains!

Example Continued: How to grow beyond this?

Example Continued: How to grow beyond this?

Summary

Linear Hashing
- Can handle growing files
- No indirection directory of extensible hashing
- Can have overflow chains
  - but probability of long chains can be kept low by controlling the r/n fill ratio (?)
Next:

- Indexing vs Hashing
- Index definition in SQL

Indexing vs Hashing

- Hashing good for probes given key
e.g., \[ \text{SELECT ... FROM R WHERE R.A = 5} \]

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 ...
Oracle and IBM DB2 UDB provide a PCTFREE
  clause to indicate the proportion of B-tree
  blocks initially left unfilled
Oracle: “Hash clusters” with built-in or DBA-
  specified hash function

The BIG picture....

- Chapters 2 & 3: Storage, records, blocks...
- Chapter 4: Access Mechanisms
  - Indexes
    - B trees
    - Hashing
- Chapters 6 & 7: Query Processing