COS 597A:
Principles of
Database and Information Systems

Indexing, Part II

1

Dynamic hashing

- Have talked about static hash
  - Pick a hash function and bucket organization and keep it
  - Assume (hope) inserts/deletes balance out
  - Use overflow pages as necessary
- What if database growing?
  - Overflow pages may get too plentiful
  - Reorganize hash buckets to eliminate overflow buckets
    - Can't completely eliminate

2

Family of hash functions

- Static hashing:
  choose one good hash function *h*
  - What is "good"?
- Dynamic hashing:
  chose a family of good hash functions
  - $h_0, h_1, h_2, h_3, \ldots h_k$
  - h_{i+1} refines h_i :
    if $h_{i+1}(x)= h_{i+1}(y)$ then $h_i(x)=h_i(y)$

3

A particular hash function family

- Commonly used: integers mod 2^i
  - Easy: low order i bits
- Base hash function can be any *h* mapping hash field values to positive integers
- $h_0(x)= h(x) \bmod 2^b$ for a chosen *b*
  - 2^b buckets initially
- $h_i(x)= h(x) \bmod 2^{b+i}$
  - *Double buckets each refinement*
- *If x integer, h(x)= x sometimes used*
  - ➢What does this assume for h_0 to be *good?*

4

Specifics of dynamic hashing

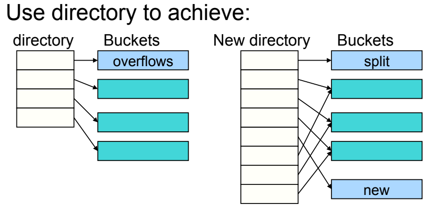
- Conceptually double # buckets when reorganize
- Implementationally don't want to allocate space may not need
  - One bucket overflows, double all buckets? NO!

Solution?
Extendable hashing
- Reorganize when and where need

5

Extendable hashing

- When a bucket overflows,
  - actually split that bucket in two
  - Conceptually split all buckets in two
- Use directory to achieve:

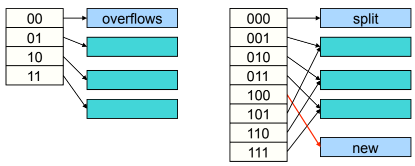


6

Extendable hashing details

- Indexing directory with $h_i(x) = h(x) \bmod 2^{b+i}$
- On overflow, index directory with

  $h_{i+1}(x) = h(x) \bmod 2^{b+i+1}$
- Directory size doubles
- Add one bucket

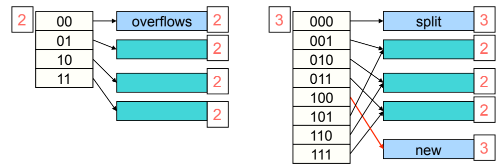


7

- What did we do?
  - Split overflowing bucket m
    - Allocate new bucket
  - Copy directory
  - Change pointer of directory entry $m+2^{b+i}$

Keep track of how many bits actually using

- depth of directory: global depth
- depth of each bucket: local depth (WHY KEEP?)



8

Rule of bucket splitting

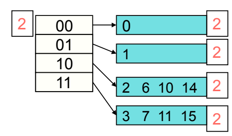
- On bucket m overflow:
  - If depth(directory) > depth(bucket m)
    - Split bucket m into bucket m and bucket $m+2^{depth(m)}$
    - Update depth buckets m and $m+2^{depth(m)}$
    - Update pointers for all directory entries pointing to m
  - If depth(directory) = depth(bucket m)
    - Split bucket m into bucket m and bucket $m+2^{depth(m)}$
    - Update depth buckets m and $m+2^{depth(m)}$
    - Copy directory and update depth(directory)
    - Change pointer of directory entry $m+2^{depth(m)}$

9

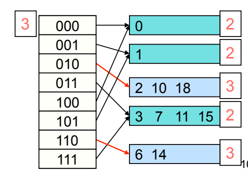
Example

Buckets: max 4 keys and data per bucket
Start with 4 buckets: depth(directory)=2

Insert records with hash values h(r) = 0, 1, 2, 3, 6, 10, 14, 7, 11, 15:



Then insert h(r) = 18
bucket '10' overflows
=> split

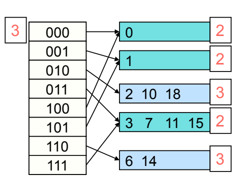


10

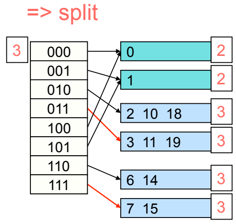
Example continued

Buckets: max 4 keys and data per bucket

After inserted h(r)=18:



Then insert h(r) = 19
bucket '11' overflows
=> split



11

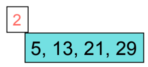
Extendable hashing observations

- Splitting bucket does not always evenly distribute contents
  - $h_i(x)$ may equal $h_{i+1}(x)$, $h_{i+2}(x)$, …
- May need to split bucket several times
  - NOT: global depth – min(local depth) = 1
- Can accept some overflow pages or split aggressively
- Almost no overflow pages with good hash function and aggressive splitting.
- If h(x) = h(y) always same bucket
  - cannot avoid overflow if too many of these!

12

Example bad bucket overflow

Bucket:



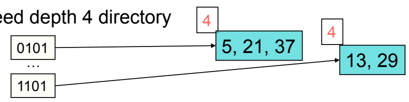
h(key) mod 2^2 = 1

h(key) mod 2^3 = 5

If add new entry with h(key)= 37 then h(key) mod 2^3 = 5

=>splitting once not enough

Need depth 4 directory



Index Operation Costs

Extendable Hashing Costs

Assume: One page per bucket; no overflow pages

- Look up: # pages read = 1 + 1
  - Assumes directory on disk
- Insert without overflow

  = look-up cost + 1 to write page of bucket
- Insert with overflow - splitting once:

  = look-up cost + 1 to write page of original bucket

  + 1 to write page of new bucket

  + 2 * (# disk pages of directory) to copy
- Splitting once may not be enough

Extendable Hashing Costs

One page per bucket; use some overflow pages

- Look up: add (# overflow pages) worst case
- Insert without splitting: add 1 **if** add new overflow page
- Insert with splitting once:

  add (# overflow pages) **always** to look-up cost

  add (# overflow pages) to write cost worst case
  - must read overflow pages to split
  - adding 1 new bucket (page), so end up with # overflow pages within 1 of number had before

B+ tree costs: preliminaries

- height of B+ tree = length of path: root → leaf

  $\leq \lceil \log_{d+1} (N) \rceil + 1$
  - N is number of leaves of tree
  - d+1 is min fanout of interior nodes except root
  - + 1 is for root
- typically root kept in memory
  - keep as many levels of tree as can in memory
  - buffer replacement algorithm may do, or pin

B+ tree costs: What is N?

- B+ tree file organization:
  - each leaf holds records

    N ≥ ⌈(# records in file / # records fit in a page)⌉

    N ≤ 2* ⌈(# records in file / # records fit in a page)⌉

    assuming no duplicate search key values
- B+ tree primary index on sequential file:
  - each leaf holds pointers to file pages
    - can be sparse index
      - one key value (smallest) for each file page
    - (key value, pointer) pairs in leaves
      - assume fit between d and 2d in leaf

    ⌈(# pages in file) / 2d)⌉ ≤ N ≤ ⌈(# pages in file) / d)⌉
    - assumes no key value spans multiple pages

B+ tree costs: What is N?

- B+ tree secondary index:
  - each leaf holds pointers to page of pointers
    - indirection: pointers in point to records
    - must be dense
    - (key value, pointer) pairs in leaves
      - assume fit between d and 2d in leaf

$N \leq \lceil(\text{\# key values in file}) / d)\rceil$
$N \geq \lceil(\text{\# key values in file}) / 2d)\rceil$

19

B+ tree costs: retrieval

- retrieving single record
  \# of pages accessed =
  height of B+-tree
  + 1 for root if on disk
  + { 1 if leaves pt to records
  2 if leaves pt to page of pointers to records

  $\leq \lceil \log_{d+1}(N) \rceil + 3$

- typical height?

20

Indexing summary

- dynamic search tree: B+ trees
- dynamic hash table: extendable hashing
- size of index depends on parameters
  - dense or sparse?
  - storing records? pointers to records? pointers to pages of pointers to records?
- disk I/O cost same order as "in core" running time.
  - hash constant time
  - search tree as log(N)

21