CS 443
Hash-based Indexes
Chapter 11

Slides adapted from Ramakrishnan & Gerhke pages.cs.wisc.edu/~dbbook/

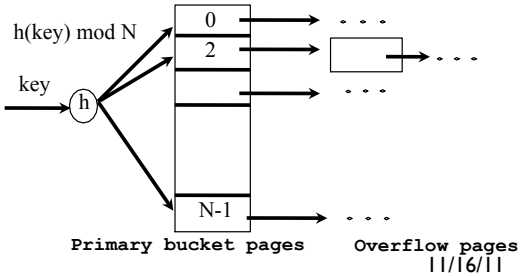
Renée J. Miller

Introduction

- As for any index, 3 alternatives for data entries k*:
  1. Data record with key value k
  2. <k, rid of data record with search key value k>
  3. <k, list of rids of data records with search key k>
  - Choice orthogonal to the indexing technique
- Hash-based indexes are best for equality selections. Cannot support range searches.
- Static and dynamic hashing techniques exist; trade-offs similar to ISAM vs. B+ trees.

Static Hashing

- # primary pages fixed, allocated sequentially, never de-allocated; overflow pages if needed.
- **h**(k) mod M = bucket to which data entry with key k belongs. (M = # of buckets)



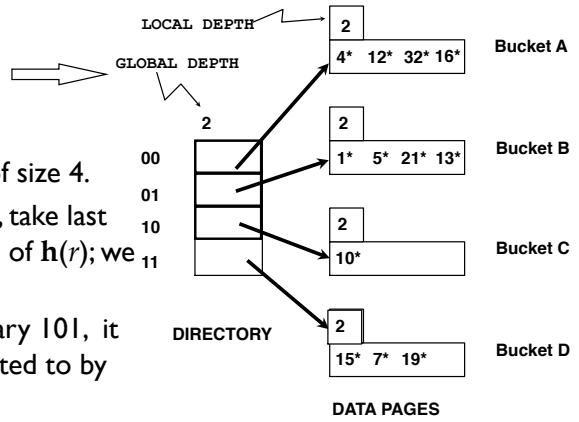
Static Hashing (Contd.)

- Buckets contain *data entries*.
- Hash fn works on *search key* field of record *r*. Must distribute values over range 0 ... M-1.
  - e.g., **h**(*key*) = (a * *key* + b) mod M
  - a and b are constants; lots known about how to tune **h**.
- Long overflow chains can develop and degrade performance.
  - *Extendible* and *Linear Hashing*: Dynamic techniques to fix this problem.

Extendible Hashing

- Situation: Bucket (primary page) becomes full. Why not re-organize file by *doubling* # of buckets?
  - Reading and writing all pages is expensive!
  - *Idea*: Use *directory of pointers to buckets*, double # of buckets by *doubling the directory*, splitting just the bucket that overflowed!
  - Directory much smaller than file, so doubling it is much cheaper. Only one page of data entries is split. *No overflow page*!
  - Trick lies in how hash function is adjusted!

Example

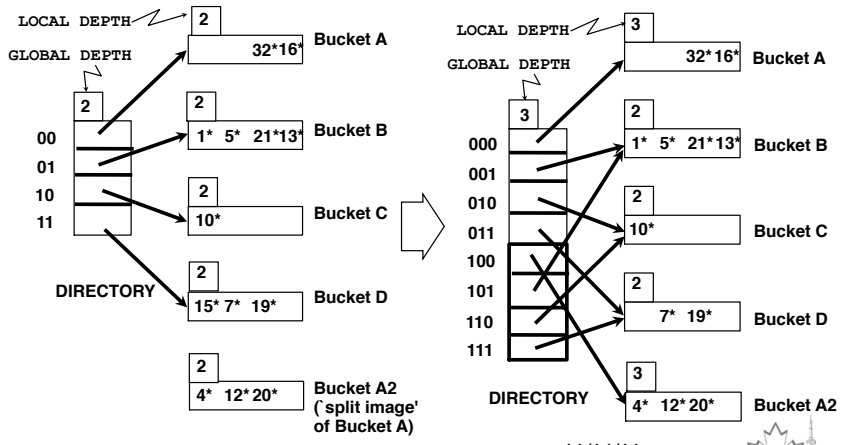


- Directory is array of size 4.
- To find bucket for *r*, take last `*global depth*' # bits of **h**(*r*); we denote *r* by **h**(*r*).
  - If **h**(*r*) = 5 = binary 101, it is in bucket pointed to by 01.

❖ **Insert**: If bucket is full, *split* it (*allocate new page, re-distribute*).

❖ *If necessary*, double the directory. (As we will see, splitting a bucket does not always require doubling; we can tell by comparing *global depth* with *local depth* for the split bucket.)

Insert *h*(r)=20 (Causes Doubling)



Points to Note

❖ 20 = binary 10100. Last 2 bits (00) tell us *r* belongs in A or A2. Last 3 bits needed to tell which.
  - *Global depth of directory*: Max # of bits needed to tell which bucket an entry belongs to.
  - *Local depth of a bucket*: # of bits used to determine if an entry belongs to this bucket.

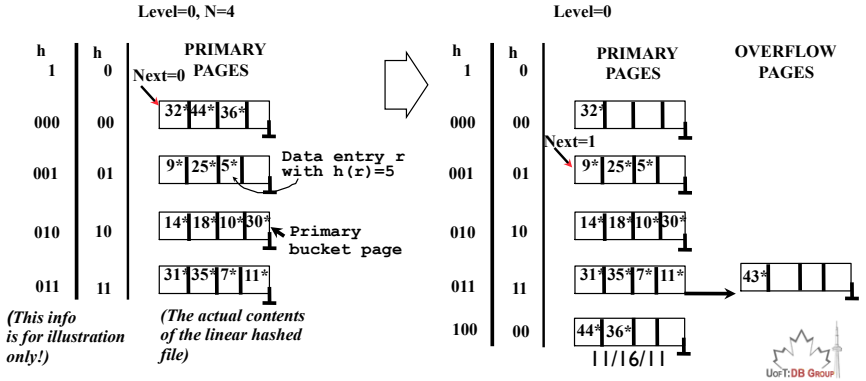
❖ When does bucket split cause directory doubling?
  - Before insert, *local depth* of bucket = *global depth*. Insert causes *local depth* to become > *global depth*; directory is doubled by *copying it over* and `fixing' pointer to split image page. (Use of least significant bits enables efficient doubling via copying of directory!)

Comments: Extendible Hashing

- If directory fits in memory, equality search answered with one disk access; else two.
  - 100MB file, 100 bytes/rec, 4K pages contains 1,000,000 records (as data entries) and 25,000 directory elements; chances are high that directory will fit in memory.
  - Directory grows in spurts, and, if the distribution *of hash values* is skewed, directory can grow large.
  - Multiple entries with same hash value cause problems!
- **Delete**: If removal of data entry makes bucket empty, can be merged with `split image'. If each directory element points to same bucket as its split image, can halve directory.

Example of Linear Hashing

- On split, $\mathbf{h}_{Level+1}$ is used to re-distribute entries.
- Insert 43.



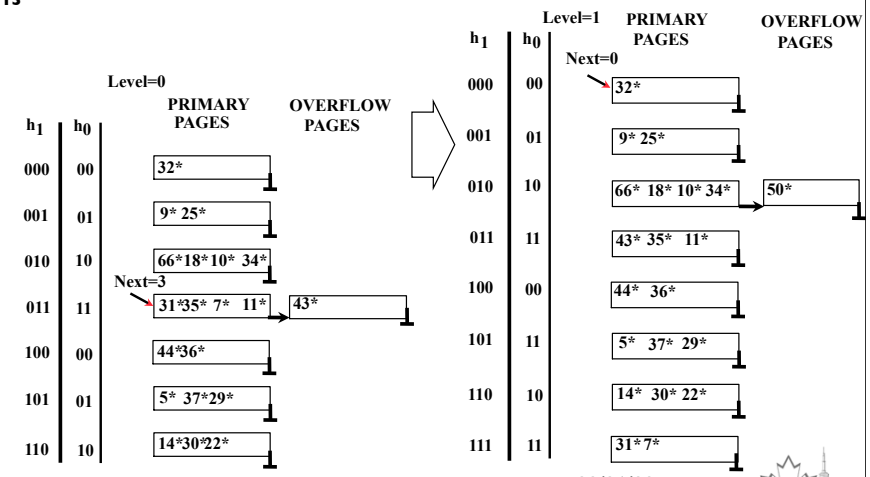
Linear Hashing

- Directory avoided in LH by using overflow pages, and choosing bucket to split round-robin.
  - Splitting proceeds in `rounds'. Round ends when all N_R initial (for round R) buckets are split. Buckets 0 to *Next-1* have been split; *Next* to N_R yet to be split.
  - Current round number is *Level*.
  - **Search:** To find bucket for data entry r, find $\mathbf{h}_{Level}(r)$:
    - If $\mathbf{h}_{Level}(r)$ in range `*Next* to N_R', r belongs here.
    - Else, r could belong to bucket $\mathbf{h}_{Level}(r)$ or bucket $\mathbf{h}_{Level}(r) + N_R$; must apply $\mathbf{h}_{Level+1}(r)$ to find out.

Linear Hashing (Contd.)

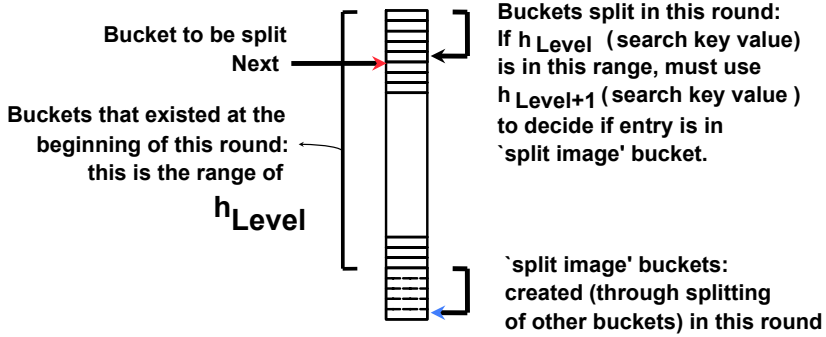
- **Insert**: Find bucket by applying $\mathbf{h}_{Level}$ / $\mathbf{h}_{Level+1}$:
  - If bucket to insert into is full:
    - Add overflow page and insert data entry.
    - (*Maybe*) Split *Next* bucket and increment *Next*.
- Can choose any criterion to `trigger' split.
- Since buckets are split round-robin, long overflow chains don't develop!
- Doubling of directory in Extendible Hashing is similar; switching of hash functions is *implicit* in how the # of bits examined is increased.

Example: End of a Round



Overview of LH File

- In the middle of a round.



LH Described as a Variant of EH

- The two schemes are actually quite similar:
  - Begin with an EH index where directory has N elements.
  - Use overflow pages, split buckets round-robin.
  - First split is at bucket 0. (Imagine directory being doubled at this point.) But elements <1,N+1>, <2,N+2>, ... are the same. So, need only create directory element N, which differs from 0, now.
    - When bucket 1 splits, create directory element N+1, etc.
- So, directory can double gradually. Also, primary bucket pages are created in order. If they are *allocated* in sequence too (so that finding i'th is easy), we actually don't need a directory! Voila, LH.

Summary

- Hash-based indexes: best for equality searches, cannot support range searches.
- Static Hashing can lead to long overflow chains.
- Extendible Hashing avoids overflow pages by splitting a full bucket when a new data entry is to be added to it.
  - Directory to keep track of buckets, doubles periodically.
  - Can get large with skewed data; additional I/O if this does not fit in main memory.

Summary (Contd.)

- Linear Hashing avoids directory by splitting buckets round-robin, and using overflow pages.
  - Overflow pages not likely to be long.
  - Space utilization could be lower than Extendible Hashing, since splits not concentrated on `dense' data areas.
    - Can tune criterion for triggering splits to trade-off slightly longer chains for better space utilization.
- For hash-based indexes, a *skewed* data distribution is one in which the *hash values* of data entries are not uniformly distributed!