Hash Index Example
Extendible Hash

Kathleen Durant PhD
Northeastern University
CS 3200

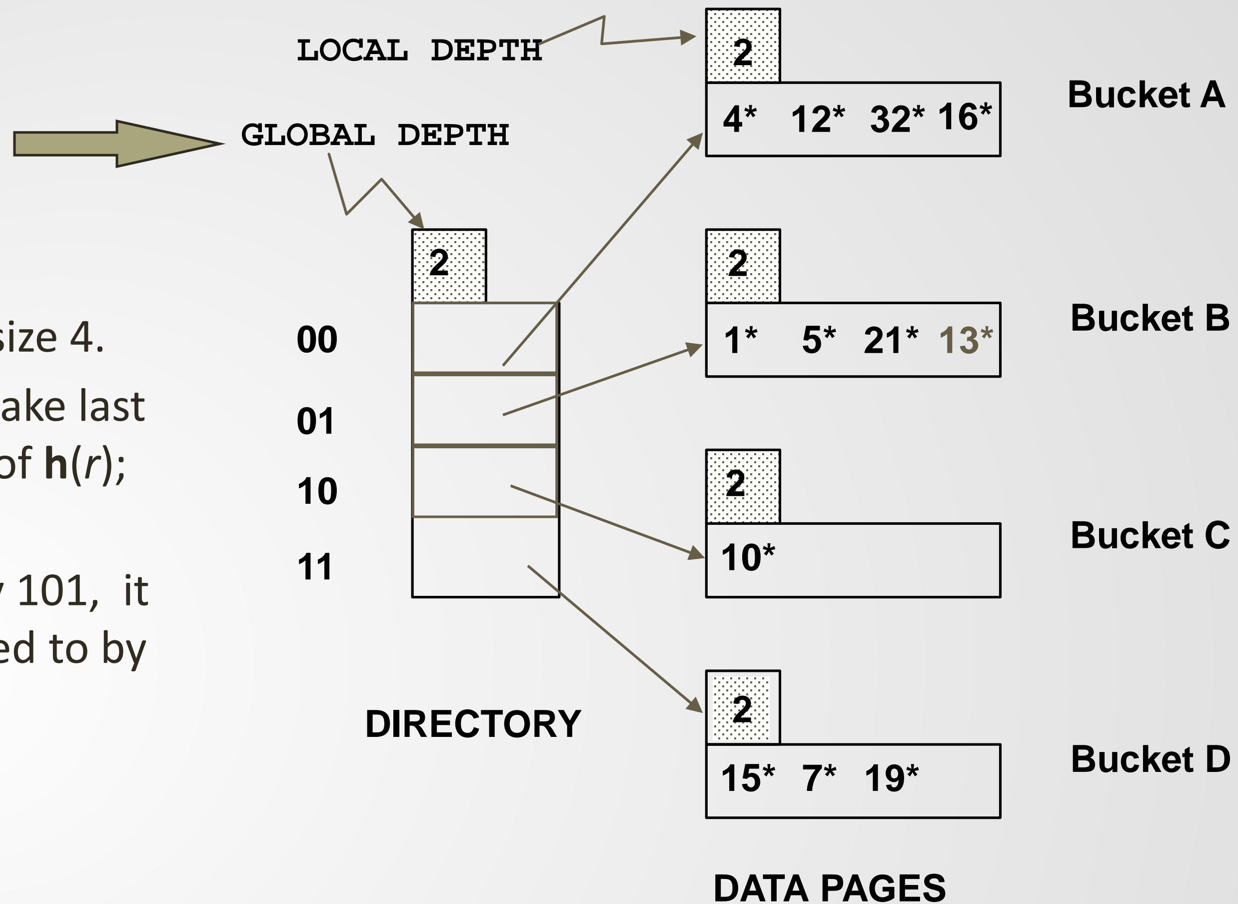
Hashing mechanism

- Your index is a collection of *buckets* (bucket = page)
- Define a hash function, *h*, that maps a key to a bucket.
- Store the corresponding data in that bucket.
- Collisions
  - Multiple keys hash to the same bucket.
  - Store multiple keys in the same bucket.
- What do you do when buckets fill?
  - Chaining: link new pages(overflow pages) off the bucket.

Extendible Hashing

- **Main Idea:** Use a directory of (logical) pointers to bucket pages
- 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

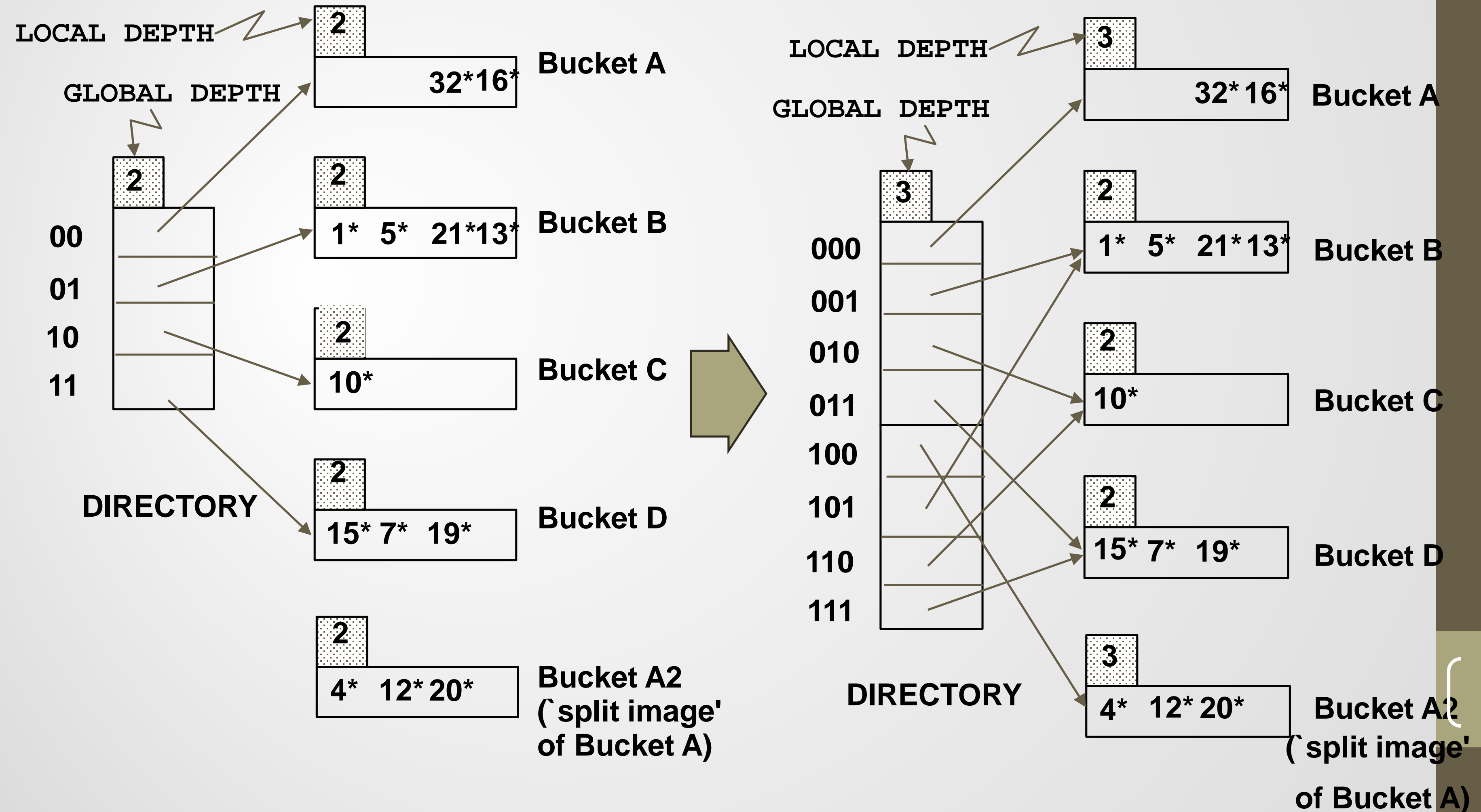


- Directory is array of size 4.
- To find bucket for *r*, take last `*global depth*' # bits of **h**(*r*); we denote *r* by **h**(*r*).
  - If **h**(*r*) = 5 = binary 101, it is in bucket pointed to by 01.

❖ **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)

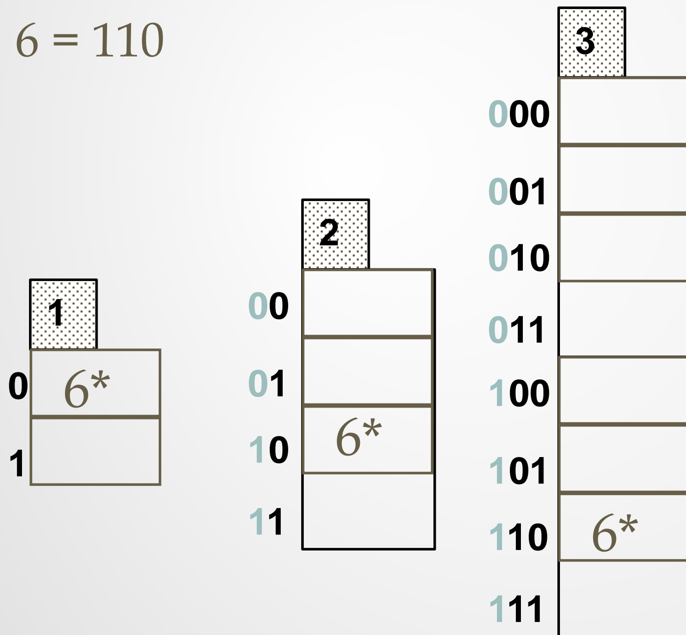


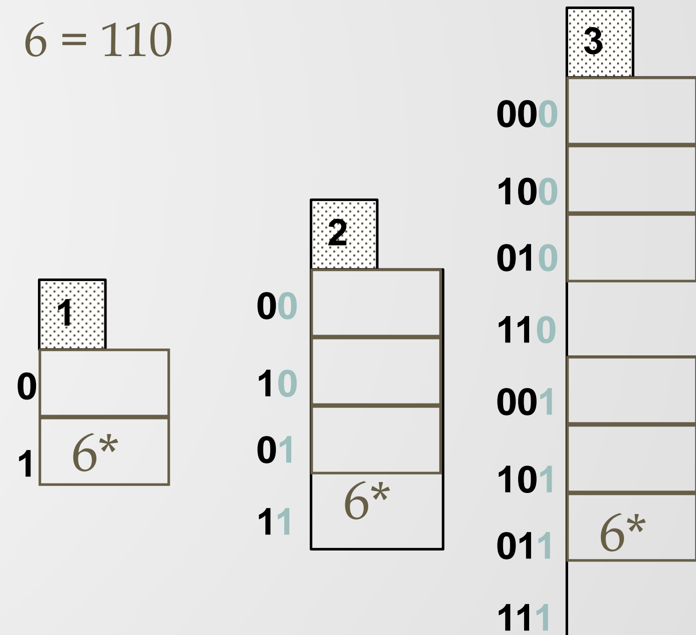
Points to Note

- 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!)

Directory Doubling

Why use least significant bits in directory?
⇔ Allows for doubling via copying!





Least Significant vs. Most Significant

Comments on 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
    - Need a decent hash function
- **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.

Hash index limitations

- They are used only for equality comparisons
  - They cannot be used for comparison operators such as < that find a range of values.
- The optimizer cannot use a hash index to speed up ORDER BY operations. (This type of index cannot be used to search for the next entry in order.)
- MySQL cannot determine approximately how many rows there are between two values (this is used by the range optimizer to decide which index to use).
- Only whole keys can be used to search for a row. (With a B+-tree index, any leftmost prefix of the key can be used to find rows.)