CS 133: Databases

Fall 2018
Lec 6 – 09/25
Hash-based Indexes
Prof. Beth Trushkowsky

Goals for Today

• Learn how hash-based indexes are constructed

• Understand how operations work on static and dynamic hash indexes, and the impact on cost in I/Os

• Reason about the tradeoffs between approaches to dynamic hash indexes

Anatomy of an Index: Hash-based

• Apply a hash function hash to search key to determine which data entries bucket
  – N number of buckets, find bucket as hash(k) MOD N

• Note: unlike tree, no index entries necessary

Hashing Functions

• Hash function works on search key field(s) of record

• Desirable properties for hash function:
  – Uniform distribution: the same number of search key values map to each bucket, for all possible values
  – Random distribution: at any given point in time, each bucket has the same number of search key values

• In practice
  – Typically operate on a binary representation of the data
  – Can tune hash to achieve uniformity (e.g., cryptographic techniques)

We’ll use integer keys in our examples
Bucket # = hash(key) MOD N
Static Hashing

- Number of primary bucket pages fixed
  - Allocated sequentially
  - Never de-allocated; chain of overflow pages if needed.

MOAR Buckets

- Situation: Bucket (primary page) becomes full.
  - Want to avoid chains of overflow pages

- Solution: add more buckets (i.e., increase “N”)?
  - Okay, but need to rehash everything!
  - Doubling # of buckets makes rehashing easier, just use one more bit to differentiate 2N buckets

- Two dynamic approaches:
  - Extendible hashing
  - Linear hashing

Classroom Example

- Example:
  - Let \( h(key) \) represent \( \text{hash(key)} \mod N \)
  - \( h(key) \) is already converted to integer data
  - # buckets \( N = 4 \)

Try Exercise 2

Extendible Hashing

- Idea: add level of indirection!
- Use a directory to point to buckets
- “Double” # of buckets by doubling the directory
  - Directory much smaller than file, so doubling it is much cheaper (might fit in RAM)
  - When want to “split” a bucket, double the directory
  - Allocate new page only for the split bucket

Any potential issues with doubling the number of buckets?!
Handling Inserts

- Use **global depth** to look up bucket in directory
- If there’s room, put data entry there.

- Else, if bucket is full, **split** it:
  - increment **local depth** of original page
  - allocate new page with new **local depth**
  - re-distribute records from original page
  - double directory if necessary (when local > global)
  - add entry for the new page to the directory

**Example: Insert 21*, 19*, 15***

- 21 = 10101
- 19 = 10011
- 15 = 01111

**Example: Insert 20* (10100): Causes Doubling**

- 20 = 10100
Insert 20* (10100): Causes Doubling

Extended Hashing: Comments

- If directory fits in memory, equality search answered with **one disk access**; else two
- Avoids overflow pages (*besides those needed for duplicates/collisions*)

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.

Local vs. Global Depth

- **Local depth:** # bits that entries in this bucket share
- **Global depth:** max # bits to determine bucket

Local < Global: When this bucket splits, it will not cause directory to double

Linear Hashing – a Lazier Approach

- Issues with Extendible
  - Completion of an insertion can take a while if it caused a split... **have to move data around**

- Linear Hashing:
  - Idea: decouple **what is split** from the **action that triggers a split**
  - A dynamic hashing scheme that handles the problem of long overflow chains **without using a directory**
**Linear Hashing Example**

- Avoids directory by:
  - using temporary overflow pages and choosing the bucket that is split in a *round-robin* fashion.
  - For example, when *any* bucket overflows:
    split the bucket that is currently pointed to by the “Next” pointer and then increment that pointer to the next bucket.

**Linear Hashing – The Main Idea**

- Use a family of functions $h_0, h_1, h_2, ...$
- $h_i(key) = hashed~key~mod(2^N)$
  - $N =$ initial # buckets (a power of 2)
  - $h_{i+1}$ doubles the range of $h_i$ (similar to directory doubling in extendible hashing)
- *Note:* at a given time, could be “using” two functions: one function for buckets that have been split vs. ones that haven’t

**Linear Hashing (Contd.)**

- Algorithm proceeds in *rounds*. Current round number is *Level*
  - There are $N_{Level} = N \times 2^{Level}$ buckets at the beginning of a round (so $N = N_0$)
  - Round ends when all *initial* buckets have been split (i.e. Next = $N_{Level}$).
  - The level determines which hash function to use

  - To start next round:
    *Level*++;
    Next = 0;

**Linear Hashing Search Algorithm**

To find bucket for data entry $k$, first find $h_{Level}(k)$.

Then:

- If $h_{Level}(k) \geq$ Next (i.e., $h_{Level}(k)$ is a bucket that hasn’t been split this round) then $k$ belongs in that bucket for sure.

- Else, $k$ could belong to bucket $h_{Level}(k)$ or bucket $h_{Level}(k) + N_{Level}$, must apply $h_{Level+1}(k)$ to find out
Linear Hashing - Insert

- Find appropriate bucket, if fits, then DONE.
- Else, if no room:
  - Add overflow page and insert data entry.
  - Split Next bucket and increment Next.
    - This is likely NOT the bucket being inserted to!
    - To split a bucket, create a new bucket and use $h_{\text{level+1}}$ to re-distribute entries.

- Since buckets are split round-robin, long overflow chains don’t develop!
Example: End of a Round

Insert 50 (110010)

Level=0, Next = 3

Level=1, Next = 0

Extendible vs. Linear

• Extendible
  – Directory grows in spurts, and, if the distribution of hash values is skewed, directory can grow large

• Linear
  – 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