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

Recall: Data Entries

• Recall, 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 w/search key k>
Choice is orthogonal to the indexing technique

• Hash-based indexes are best for equality selections. Cannot support range searches.

• Static and dynamic hashing techniques exist; tradeoffs similar to ISAM vs. B+ trees.

Static Hashing

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

h(search key) yields which bucket

key ➔ h(key) ➔ \( \begin{array}{c} 0 \\ 1 \\ \vdots \\ N-1 \end{array} \) ➔ \( \begin{array}{c} \ldots \\ \ldots \end{array} \)

Primary bucket pages  Overflow pages
Hashing Functions

- Hash function works on search key field(s) of record
- What would be good properties for a hash function?

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
- \( h(key) = key \mod N \) works well for uniformly distributed integer data
- Can tune \( h \) to achieve uniformity (e.g., cryptographic techniques)

Static Hashing

- Example:
  - Suppose creating index over integer search key, e.g., SID or age
  - # buckets \( N = 4 \) → Multiple of 2, use 2 bits to differentiate

  \[
  \begin{align*}
  00 &= 0 \\
  01 &= 9^* \\
  10 &= 2 \\
  11 &= 3 \\
  \end{align*}
  \]
  - Insert 9*
  - 9 \mod 4 = 1
  - \( 1001 \& 0011 = 0001 \)

  Try Exercise 2

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 a new hash function!
  - Doubling # of buckets makes this easier, use one more bit to get 2N buckets
    - Any potential issues with doubling the number of buckets?!

- Two dynamic approaches:
  - Extendible hashing
  - Linear hashing

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

This mask lets us inspect only the last two bits of 9
Handling Inserts

• Find bucket where data entry belongs.
• If there’s room, put it 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 (before picture)
Insert 20* (10100): Causes Doubling

Local vs. Global Depth

Extendible 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.

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.

Illustration only, this info not actually stored anywhere

000 00
001 01
010 10
011 11
100 00

Insert 3*?

Linear Hashing – The Main Idea

- Use a family of hash functions $h_0, h_1, h_2, \ldots$
- $h_i(key) = key \mod (2^N)$
  - $N$ = 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 hash functions: one function for buckets that have been split vs. ones that haven’t

State of bucket splits:
Buckets 0 to Next – 1 are split
Buckets Next to $N_{level}$ – 1 are not

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;

Next=1

Linear Hashing Search Algorithm

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

Then:
If $h_{level}(k) >= Next$ (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 \( h_{level+1} \) to re-distribute entries.

- Since buckets are split round-robin, long overflow chains don’t develop!

Example: Search 44 (11100), 9 (01001)

\[ h_{level}(key) = key \mod (2^{level}N) \]
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