Introduction

- As for any index, 3 alternatives for data entries $k^*$:
  - Data record with key value $k$
  - $<$k, rid of data record with search key value $k$>
  - $<$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

- $h(k) \mod N = \text{bucket to which data entry with key } k \text{ belongs. } k_1 \ast k_2 \text{ can lead to the same bucket.}$
- Static: # buckets (N) fixed
  - main pages allocated sequentially, never de-allocated;
  - overflow pages if needed.
Static Hashing (Contd.)

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

Extendible Hashing

- When 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 and double # of buckets by (1) doubling the directory, (2) 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 $N=4$, global depth $D = 2$.
- To find bucket for key:
  - Get $h(key)$,
  - Take last $\lfloor \text{global depth} \rfloor$ # bits of $h(r)$, i.e., mod $2^D$.
  - If $h(key) = 5 = \text{binary 101}$, take last 2 bits, go to bucket pointed to by 10.
- Each bucket has local depth $L$ ($L \leq D$) for splits!
**Inserts**

- If bucket is full, **split** it:
  - allocate new page,
  - re-distribute.
- If necessary, **double** the directory. Decision made by comparing global depth $D$ and local depth $L$ for the split bucket.
  - Split if $D = L$.
  - Otherwise, don’t.

Insert $k^*$ with $h(k)=20$?

---

**Insert $h(k)=20$ (Causes Doubling)**

---

**Points to Note**

- $20 = \text{binary } 10100$. Last 2 bits (00) tell that $k$ belongs in A or A2. Last 3 bits needed to tell exactly which.
  - **Global depth $D$ of directory**: Max # of bits needed to tell which bucket an entry belongs to.
  - **Local depth $L$ of a bucket**: # of bits actually needed to determine if an entry belongs to this bucket.
- When does bucket split cause directory doubling?
  - Before insert, $L$ of bucket = $D$ of directory.
  - 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 (inserting 8*)**

**Least Significant**

```
0 1011 1010 1001 1000 1001 1000 1000
1 0101 0100 0100 0011 0010 0010 0000
```

**Most Significant**

```
0001 0011 0101 0110 1000 1001 1010 1011
```

**Deletes**

- **Delete**: removal of data entry from bucket
  - If bucket is empty, can be merged with ‘split image’.
  - If each directory element points to same bucket as its split image, can halve directory.
  - If assume more inserts than deletes, do nothing…

**Comments on Extendible Hashing**

- **Access cost**: If directory fits in memory, equality search takes one I/O to access the bucket; else two.
  - 100MB file, 100 bytes/record, 4K pages
  - 1,000,000 records (treated as data entries)
  - 40 data entries per page, 25,000 pages/buckets
  - 25,000 elements (pointers) in directory; if 1K pointers per page, 25 pages for directory, likely to fit in memory!
- **Skews**: If the distribution of hash values is skewed, directory can grow large. An example?
- **Duplicates**: Entries with same key value need overflow pages!
Linear Hashing

- Another dynamic hashing scheme, an alternative to Extendible Hashing.
- Advantage: adjusts to inserts/deletes without using a directory.
- Idea: Use a family of hash functions $h_0, h_1, h_2, \ldots$
  - $h_i(key) = h(key) \mod(2^i N)\quad N = \text{initial \# buckets}$
  - $h$ is some hash function (range is not 0 to N-1)
  - $h_0 = h(key) \mod N$
  - $h_{i+1}$ doubles the range of $h_i$ (similar to directory doubling)
  - If $N = 2^{d_0}$, for some $d_0$, $h_i$ consists of applying $h$ and looking at the last $d_i$ bits, where $d_i = d_0 + i$.

Linear Hashing (Contd.)

- LH avoids directory by
  - using temporary overflow pages, and
  - choosing bucket to split round-robin.
- Splitting proceeds in 'rounds'.
  - Round $L$ ends when all $N_L = 2^LN$ initial buckets are split.
  - Next bucket to be split. Buckets 0 to Next-1 have been split; Next to $N_L$ yet to be split.
  - Search: To find bucket for data entry $k^*$, find $h_L(k)$:
    - If $h_L(k)$ in range 'Next' to $N_L$, $k^*$ belongs here.
    - Else, $k^*$ could belong to bucket $h_L(r)$ or bucket $h_L(k) + N_L$; must apply $h_{L+1}(r)$ to find out.

LH File and Searches

- In the $L^{th}$ Round of splitting:
  - Buckets split in this round
  - Search: If $h_L(key)$ is in the split range, must use $h_{L+1}$ to decide if entry is in a split image bucket.

'Split image' buckets: created (through splitting of other buckets) in this round.
**Inserts**

- **Insert**: Find bucket $B$ by applying $h_L / h_{L+1}$:
  - If bucket $B$ to insert into is full:
    - Add overflow page and insert data entry $k$.
    - (Maybe) Split bucket Next (often $B 
eq Next$) using $h_{L+1}$ and increment Next.
- Can choose any criterion to 'trigger' split.
- Since buckets are split round-robin, long overflow chains don't develop!
- Extendible Hashing in comparison:
  - Doubling of directory is similar.
  - Switching of hash functions is implicit in how the $B$ of bits examined is increased. No need to physically double the directory!

**Example of Linear Hashing**

- On split, $h_{L+1}$ is used to redistribute entries.

**Example: End of a Round**

- Insert $50^*$
**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 $(1, N+1), (2, N+2), \ldots$ 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.
- Directory can double gradually.
- 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!

---

**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. *(But duplicates may require overflow pages.)*
  - 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.
  - Duplicates handled easily.
  - Space utilization could be lower than Extendible Hashing, since splits not concentrated on ‘dense’ data areas.
  - Especially true with skewed data.
  - Can tune criterion for triggering splits to trade-off slightly longer chains for better space utilization.
- For hash indexes, a skewed data distribution means ***hash values*** of data entries are not uniformly distributed! Cause problems with EH and LH.