Dynamic Hashing

Introduction

- Can we improve upon logarithmic searching?
- Hashing is a technique that attempts to provide constant time for searching and insertion, i.e O(K)
- The basic idea for both searching and insertion is to apply a hash function to the search field of the record.
- The return value of the hash function is used to reference a location in the hash table

‘

Different approaches

- Create a hash table containing N addressable ‘slots’
- Each “slot’ may contain one record
- Create a hash function that returns a value to be used in insertion and searching
- The value returned by the hash function must be in the correct range, i.e, the address space of the hash table
- If the range of the keys is that of the address space of the table, we can guarantee constant time lookup
- Usually, this is not the case as the address space of the table is much smaller than that of the search field

- With numeric keys can use modulo-division or truncation
- With character keys must first convert to integer value. Can achieve this by multiplying ASCII code of characters together and then applying modulo-division
- Cannot guarantee constant time performance as collisions will occur i.e., two records with different search values being hashed to the same location in the table
- we require a collision resolution policy

Efficiency then depends on the number of collisions. Number of collisions depends mainly on the load factor, λ, of the file:

$$\lambda = \frac{\text{no of records}}{\text{no of slots}}$$

Collision Resolution Policy

- **Chaining:** if location is full, add item to a linked list
- performance degrades if load factor is high.
- lookup time is, on average, 1 + λ (average case)
- **Linear Probing:** if location is full, check in a linear manner for next free space.
- This can degrade to a linear scan: performance:
- if successful: $0.5(1 + \frac{1}{1-\lambda})$
- if unsuccessful: $0.5(1 + \frac{1}{(1-\lambda)^2})$
- one big disadvantage is that this leads to the formation of clusters
- **Quadratic probing:** if location is full, check location $x + 1$, location $x + 4$, ... $(x + n)^2$
- less clustering
- **Double hashing:** if location x occupied, then apply second hash function can help guarantee even distribution (a fairer hash function)

Dynamic Hashing

- Care should be taken in designing hash function. Usually require fair hash function.
- Difficult to guarantee if no/limited information available about the type of data to be stored.
- Often heuristics can be used if domain knowledge available
- Can have both internal (some data structure in memory) or external hashing (to file locations)
- Size of original table or file?

- We considered hashing to an array (in memory).
- In reality, in database systems, we are typically hashing to a disk block (bucket) each of which can contain a fixed number of records.
- If a block is full, then we have a collision.
- Typically dealt with using overflow buckets (chaining).

- The cases we've considered thus far deal with the idea of a fixed hash table; this is referred to as static hashing.
- Problems arise if the database grows larger than planned; too many overflow buckets and performance degrades.
- A more suitable approach is dynamic hashing, where the table/file can be resized as needed.

General Approach

- use a family of hash functions h_0, h_1, h_2, etc.
- h_{i+1} is a refinement of h_i
- For example, $Kmod2^i$
- Develop a base hash function that maps key to a positive integer
- Then use, $h_0(x) = x\text{mod}2^b$ for a chosen b. There will be 2^b buckets initially.
- Can effectively double the size of the table by incrementing b

- Common dynamic hashing approaches: extendible hashing and linear hashing.
- Conceptually double the number of buckets when re-organising. From an implementation perspective, we do not actually double size as it may not be needed.
- Extendible hashing - reorganise buckets when and where needed
- Linear hashing - reorganise buckets when but not where needed.

Extendible Hashing

- When a bucket overflows, split that bucket in two.
- Conceptually, split all the buckets in two A directory (a form of index) is use to achieve this conceptual doubling.

Extendible Hashing

- If a collision or overflow occurs, we don't re-organise the file by doubling the number of buckets; too expensive.
- Instead we maintain a directory of pointers to buckets, we can effectively double the number of buckets by doubling the directory, splitting just the bucket that overflowed.
- As the directory is much smaller than file, so doubling it is much cheaper.

- On overflow, we split the bucket (allocate new bucket and re-distribute contents).
- We double the directory size if necessary.
- For each bucket, we maintain a local depth (effectively the number of bits needed to hash an item here).
- Also maintain a global depth for the directory; the number of bits used in indexing items.
- These values can be used to determine when to split the directory.

- If overflow in bucket with local depth = global depth, then split bucket, re-distribute contents, double the directory.
- If overflow into bucket with local depth < global depth, then split bucket, re-distribute contents. Increase local depth.
- If directory can fit in memory, then retrieval for point queries can be achieved with one disk read.

Linear Hashing

- Another approach to indexing to a dynamic file. Similar idea in that a family of hash functions are used ($h = K \bmod 2^i$), but differs in that no index is needed.
- Initially, create a file of M buckets. $K \bmod M^1$ is a suitable hash function.
- We will use a family of such functions $K \bmod (2^i \times M), i = 0$ initially.
- Can view the hashing as comprising a sequence of phases.
- For phase j, the hash functions $K \bmod 2^j \times M$ and $K \bmod 2^{j+1} \times M$ are used.

- Splitting a bucket means to redistribute the records into two buckets: the original one and a new one.
- In phase j, to determine which ones go into the original while the others go into the new one, we use $h_{j+1}(K) = Kmod2^{j+1} \times M$ to calculate their address.
- Irrespective of the bucket which causes the overflow, we always split the next bucket in a linear order.
- We begin with bucket 0, and keep track of which bucket to split next, p.
- At the end of a phase when p is equal to the number of buckets present at the start of the phase, we reset p and a new phase begins (j incremented).