PRINCIPLES OF DATA ORGANISATION

Hashing Introduction

MOTIVATION

- Index
  - Give (artificial) ID – get position of the record in the primary file
- Direct access ~ equality query
  - Not for range queries

HASHING

- Also known as direct accessing, randomizing
- Hashing is a technique capable of accessing a record in memory in **O(1)** time by using **hash functions**
  - Maps search keys to (physical or logical) addresses (buckets)
- Hash function is a mapping from the query space to the address space
  $h : K^* \rightarrow \{0, 1, \ldots , M - 1\}$
  - Query space = the space of all possible values of the query key
    - Ex. Name, address, age, …
  - Usually: address space << query space
- ***h*(*k*)** determines the address of a record with a key ***k***

HASH FUNCTION

- A good hash function should have:
  - Uniform distribution: Each bucket should contain keys from all parts of the address space
    - Distributes the values evenly across buckets
    - All buckets are expected to contain a roughly equal number of hash values
    - There are no unused buckets
  - Random distribution: Each bucket should be equally filled regardless of the key value distribution
    - The result should be dependent on all bits of the key
- A good hash function should be:
  - Deterministic: the resulting value is dependent only on the input values
    - For the same key we get the same address
  - Fast: it should take only few instructions to compute the resulting value of the hash function
    - Usually an algorithm evaluates the function

- A bad function
  - would map all the search keys onto the same address
  - search = sequential scan

HASH FUNCTIONS – TRIVIAL

- The numerical representation of the key represents the relative (or absolute) address
  - A small number of values that fit into the (primary memory) address space

- Advantages:
  - fast
  - **perfect** (no collisions)
- Disadvantages:
  - usable only for relatively small domains
  - commonly neither uniform nor random
    - Depends on the distribution of the values of the keys

- Examples:
  - 32-bit integer values – can directly represent the bucket index
  - 26 letters → 3-letter codes can be uniquely mapped into 26^3 = 17576-long array

HASH FUNCTIONS – MODULO

- $h(k) = k \bmod M$
- For $M = 16$ value of $h(k)$ is dependent solely on the 4 low-order (least significant) bits of the key
  - These bits can be poorly distributed, which can lead to poor distribution of the results
    - i.e. lots of collisions
- M is advised to be a prime number

HASH FUNCTIONS – BINNING

- $h(k) = k\ /\ M$

- We need to know the range of the domain
- Can be seen as an inverse to modulo since it looks at the high-order bits
  - If the distribution of the high-order bits is poorly distributed, so will the results
- For $\boldsymbol{M} = 100$ and domain range $< \mathbf{0}; \mathbf{1000} >$
  - values $< \mathbf{0}; \mathbf{99} >$ will go to the first slot
  - values $< \mathbf{100}; \mathbf{199} >$ will go to the second slot
  - …

HASH FUNCTIONS – MID-SQUARE

- Squares the key value, and then takes the middle r bits of the result, giving a value in the range $<0;2^{r-1}>$

- Good to use with integers
- Is not dependent on the distribution of low- or high-order bits – all bits contribute to the final value
  - In the previous two cases, a change of some bits has no impact

$r = 2, k = 4567 \rightarrow 4567\text{^}2 = 20857489 \rightarrow h(k) = 57$

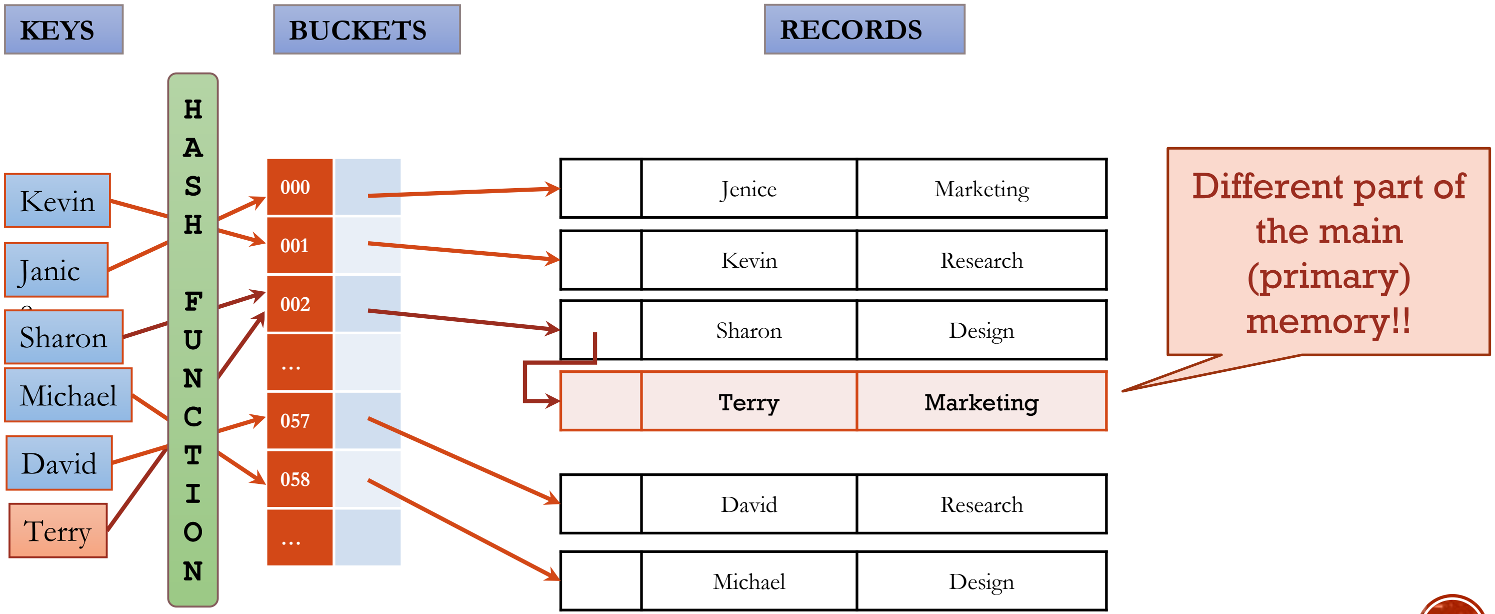
```
       4567
       4567
   --------
      31969
     27402
    22835
   18268
   --------
   20857489
```

INTERNAL HASHING

- Hashing structure fits in main memory ~ limited space
- Each bucket contains one record
  - Basically associative array
- Hash table utilises a hash function (map) to match the keys with their associated values
- If multiple keys are mapped to the same position ~ collision
- Hash tables vary in collision handling
  I. Separate chaining/hashing
  II. Open addressing
  III. Coalesced chaining/hashing
  IV. Cuckoo hashing

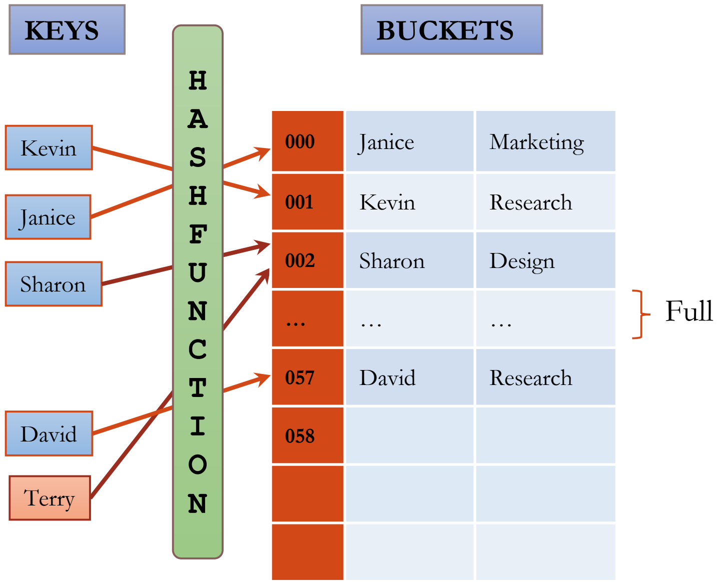
I. SEPARATE CHAINING

Buckets contain links to chains of collided records



II. OPEN ADDRESSING

Collided record is inserted into the next free bucket (basic version)



Searching for a record with key K:

1. compute the address *A* from the query key *K* using the hash function
2. if no record is present at *A*, the searched record is not in the table
3. Otherwise, scan (see the following slides) the table **until either record with key *K* is found (record found) or an empty slot is encountered (record not present)**

Example:

- hash(Terry) = 002 → collision
- Use for Terry the next free bucket: 058
- hash(Michael) = 058 → collision
- Use for <u>Michael</u> the next free bucket

II. OPEN ADDRESSING – PROBE FUNCTION

Next bucket is determined by a probe sequence generated by a probe function.
The function should also keep a track of whether it did not get into a cycle.

```
void insert(const Key& k, const Record& r)
{
  int home;                          // Home position for k
  int pos = home = h(k);             // Init probe sequence
  int i = 0;
  while (HT[pos].key() != EMPTYKEY) {
    i++;
    pos = (home + p(k, i)) % M;      // probe function
    if (k == HT[pos].key()) {
      cout << "Duplicates not allowed\n";
      return;
    }
  }
  HT[pos] = r;
}
```

II. OPEN ADDRESSING – PROBE FUNCTION

Clustering

- When sequentially scanning for a next free slot, the probe sequences can collide and thus cause clustering
  - Long sequence for receiving a record
- Optimal probe function should provide each slot with an equal probability of receiving a record
  - It should cycle through all slots in the hash table before returning to the home position.

II. OPEN ADDRESSING – PROBE FUNCTIONS

Linear probing

- $p(k,i) = c * i$
- c and M should share no factors
  - M – the size of address space
- i – the number of failed attempts to find an empty bucket
- c = 1 … try the next bucket

Quadratic probing

- $p(k,i) = (c_1\, i + c_2\, i^2)$
- Wrong choice of constants can prevent from visiting every slot
- There exists a fitting choice of the constants
  - $c_1 = 0, c_2 = 1$
M = prime number
    - At least half slots will be visited
  - $c_1 = ½\ c_2 = ½$
M = power of 2
    - Every slot will be visited

(Pseudo-)random probing

- $p(k,i) = perm[i]$
- *perm* is a pre-defined table with permutations of length M

Double hashing

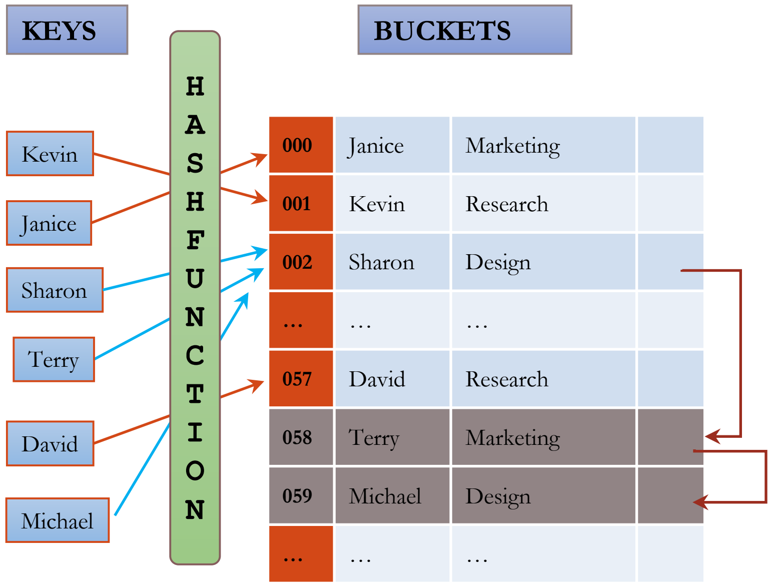
- $p(k,i) = i * g(k)$
- The probe sequence is now different for different keys

III. Coalesced Chaining

- Combines separate chaining and open addressing
  - The chains are stored in the hash table
- When a collision occurs, the new value is stored to the first free bucket from the end of the table
- The end of the chain is connected to this new value
- Collided records are chained to decrease the retrieval time
  - For both insert and query operations
- Two chains never merge (as probe sequences can)

III. Coalesced Chaining

Combines separate chaining and open addressing
Two chains never merge (as probe sequences can)



IV. Cuckoo Hashing

Two hash functions h_1, h_2

- No overflow chains or scanning of the hash table
- If $h_1(k)$ is full, insert the record anyway and kick the residing record (k') into its alternative location $h_2(k')$
  - If $h_2(k')$ is full, repeat the strategy until a new position is found or the process is too long
  - If too long, choose new functions and rebuild (rehash) the structure
- Often implemented by 2 tables each having its own hash function
  - Values move between the tables

IV. Cuckoo Hashing – Example

Insert Z : $h_1\ (Z) = 7, h_2\ (Z) = 0$ (positions in the table)

The graph shows the insertion "chain"

- $Z \longrightarrow W$
- $W \longrightarrow H$
- $H \longrightarrow Z$
- $Z \longrightarrow A$
- $A \longrightarrow B$
- $B \longrightarrow empty$

- Insert:
  - Worst case complexity: O(n)
  - Amortized: O(1)
- Look-up, delete: O(1)

https://programming.guide/cuckoo-hashing.html