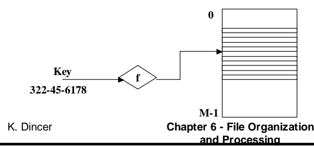
Chapter 6 - Hashing

Record for	SSN	→ f →	Slot address
Al	322-45-6178		178
Joe	123-45-6284		284
Mary	036-23-0373		373
Pete	901-23-4784		284

A randomizing transformation for a personnel file uses SSN as the key.

We assume that the value of the low-order digits of these numbers are evenly (uniformly) distributed.

We have space for 500 employees (M=500 slots)m.

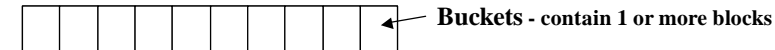


Terminology

Hash or Key-to-Address-Translation (KAT) function is a means of calculating the disk address of a block containing a given record from the value of its key.

Key → f → Address

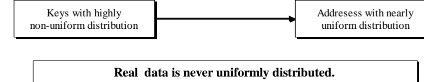
Buckets



Collisions.

We have a collision when the function yields the same number for two different keys.

Hash functions never preserve order.



Real data is never uniformly distributed.

When the bucket fills up and another record is entered whose hash value is the same as the records in the full bucket, we add address of a new bucket in the overflow area.

Chaining. Adding address of an overflow bucket to a primary-area bucket which is full.

Separate Chaining. The case where there is a separate chain of overflow buckets for each primary-area bucket.

When a bucket overflows, a link or address is added referring to an overflow block - called "*Hashing with buckets and separate chaining*."

See Figure 6.1

$h(x) = x \bmod 5$

Bkfr (bucket factor) = 3

Load Factor.

If we allocate exactly the amount of space needed for the file to the primary area, there will be many chains.

Hash functions give nearly random distributions. Primary area should be 70-90% full only.

Load factor Lf = (# records in file) / (# places for records in primary area)
= n / (M x Bkfr)

where Bkfr is Bucketing factor.

Fetching Using Buckets and Chains.

If the distribution of values is even and original area chosen is big enough,

Hashing is very efficient for fetching (~1 access)

$T_F = s + r + dtt$

Effects of load factor and bucket factor ?

Ways of handling overflow?

Time Analysis

T_F (successful) $= s + r + dtt + (x/2)(s + r + dtt)$

T_F (unsuccessful) $= s + r + dtt + x(s + r + dtt)$

where x is the average chain length.

$T_D = T_F$ (unsuccessful) $+ 2r$

$T_I = T_F$ (unsuccessful) $+ 2r$

$T_X = n\, T_F$

A Mix of Operations

- CA income tax file of 6 million 400-byte records.
- Every month we make 10,000 random individual record fetches, and we have ten small-range queries, each for about 3% of the file.
- We need to choose one:
  - Use hash table for individual-record fetches, and read sequentially through the file for the range queries.
    Lfr = 70%, Bkfr = 50 (Each fetch is about one disk access)
  - Use the same hash table for the individual-record fetches, but keep a secondary B+-tree index for the small range queries.
    Tree has block addresses for each record, all nodes are 2400 bytes long.
  - Use a primary B+-tree for both the individual-record fetches and the small-range searches.
    Leaf node size is chosen to make this mix of operations optimal.
    Internal node size is 2400 bytes, and we have two-disk access method.

Hint: T_x for CA file when it is organized as a pile file is 14 minutes.

Intersection File and Hash Partitioning

We have a MA and a BCBS file. BCBS file has an associated hash-table. Both files have 100,000 400-byte records, of which 70,000 are common. (Remember sorting and comparing took 2.5 minutes)

Case 1: Sort and compare : 2.5 minutes

Case 2: hash and no overflow

- Divide hash table into 6 equal parts (partitions).
- Use the same hash function on the same field on the MA file to divide MA file into 6 partitions.

Linear Hashing

LH maintains a constant load factor, even when many new records are added to the file.

- It does so by incrementally adding new buckets to the primary area.

<u>Fetching a Record</u>

The last binary bits in the hash number are used for placing the record.

key ---> hash fcn ---> large number + extract k binary digits

When we expand the table, we split an existing bucket which holds all the records which end in a particular k digits into two buckets using the k+1 last digits of the hash number.

010 ---> 0010 and 1010

For fetching, we can tell whether we need to look at k+1 digits or k digits by checking whether or not the last k digits are smaller than a given value, the ***<u>boundary value</u>***. (kept in the file header)