A Fast Retrieval Method for Local or Distributed Data

Rafael O. Fontao, Member IEEE (*)

Abstract—In this paper, we propose an improvement to an approach to data retrieval which is performed in only one access to a bucket hash table or file. The idea behind it, is to let the system assign one digit to the record key so that the hashed new record key is "forced " to fall in a bucket according to some practical criteria. From a user point of view this forced hash procedure could be thought of as a “user-system cooperating code assignment”, since the user is free to code an object to be retrieved but the system may append s a digit to that code. For one access retrieval purposes, the new code key-digit is used to find its address. However, should the digit is not known, the retrieval process will find the key in its surrounding, provided it exists. In this approach it is unnecessary a bucket overflow area of any kind, since this method allows a high load factor for practical use. In the event of the hash table is nearly full, a simple procedure could be ran to extend the table size either by keeping the original digit or assigning new ones.
For distributed data sets this methodology shows an appealing performance in real life and simulation results.

Index terms: Distributed data searching, Hashing, one access retrieval, system cooperating code assignment, key management.

(*) Departamento de Ingeniería Eléctrica, Universidad Nacional del Sur,Bahía Blanca - ARGENTINA
e-mail: fontao@ieee.org

Introduction

Searching in data bases is always a worthy subject of research. Fast retrieval of records with a given key is essential to any data processing. Even thought the speed and store capacity of individual computers are increasing every day, searching on local or distributed data base on a network, as the Web, still poses a problem [3],[5].

Among the searching methods, the hashing techniques are well known and applied everywhere. The technique proposed here is an improvement to an existing method [2] and may deserve such qualifications. The underlining idea came from the use of a **checking digit** attached to some type of unique key (for example in Social Security, Payroll Personal Number, Unique Identifying Key for Tax Purposes, even the University Student ID.)

The **checking digit** was inherited from the batch processing era when data were typed more than once for validation purposes. However, in on line data processing, where as soon as the key is typed the user can visually check for the correct entry, the checking digit already implemented in existing information systems could be used for other purposes. For instance, it can be used to provide additional information to "**force**" a given key to fall down in a bucket (or bucket file) , so that bucket overflow will only occur when the available space is almost full. This allows to give the buckets a size as large as a disk sector, so the retrieval of a record key will take only one disk access. For distributed data among several files, the bucket itself is a file, and the one access characteristic is made on it.

Simulation results show that this technique may allow a load factor grater than 0.93 in the worst cases. Henceforth we do not need to reserve large hash table space as in the standard hash bucket technique to prevent the overflow [3].

Therefore we may implement insertion, deletion and searching of a record without taking care of the overflow problem, thereafter in maintenance period, keep control of the load factor, and eventually rehash the table.

The Standard Bucket Hash Methodology.

Let us have a set of at most N data records with a unique identifying key field named **RecordKey** of length **LRecordKey**. The problem of searching a record as fast as possible given its **RecordKey** value, is in the core of all information systems. One of the approaches to deal with this problem is to use hashing techniques. Hash functions map the set of data records into a set of integers { 1,2,3,…., **MaxHashTable** } which are the entries in a Hash Table that contains information where the data record is physically located. Perfect hash functions map any two different **RecordKey's** into different values.

Additionally a perfect hash function is called **minimal** when every entry is just filled only once (i.e. **MaxHashTable** =N). Unfortunatelly, such functions are hard to find for dynamic data sets (if possible at all), so the practical hash function we can recur to, will distribute as evenly as possible the data record key among the entries { 1,2, ,…., **MaxHashTable**}.

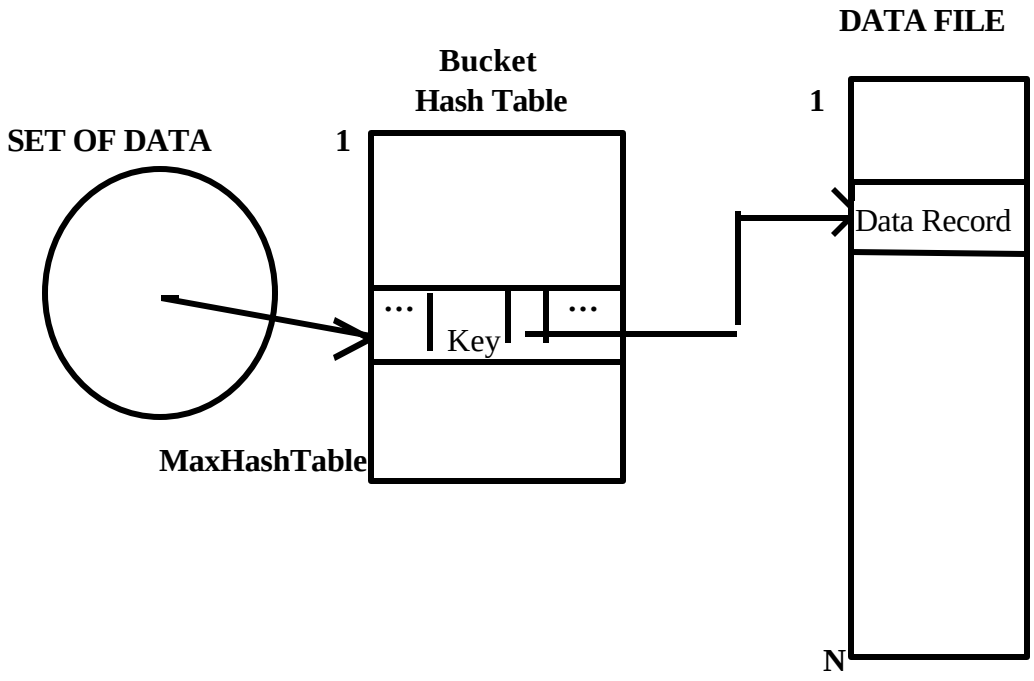
Eventually, there may be two or more **Record Key** which will map to the same entry, giving rise to the **collision problem**.

Bucket hash is among the most useful solutions proposed in the literature. In this technique each entry at the hash table is large enough to hold a set of record keys with the same hash value. The size of the bucket is a design parameter. When a bucket is full (bucket overflow) this technique will take additional considerations to deal with this problem [4],[8].

A bucket hash table could be of the type formed by **BucketCapacity** records, which in turn, are formed by two fields:

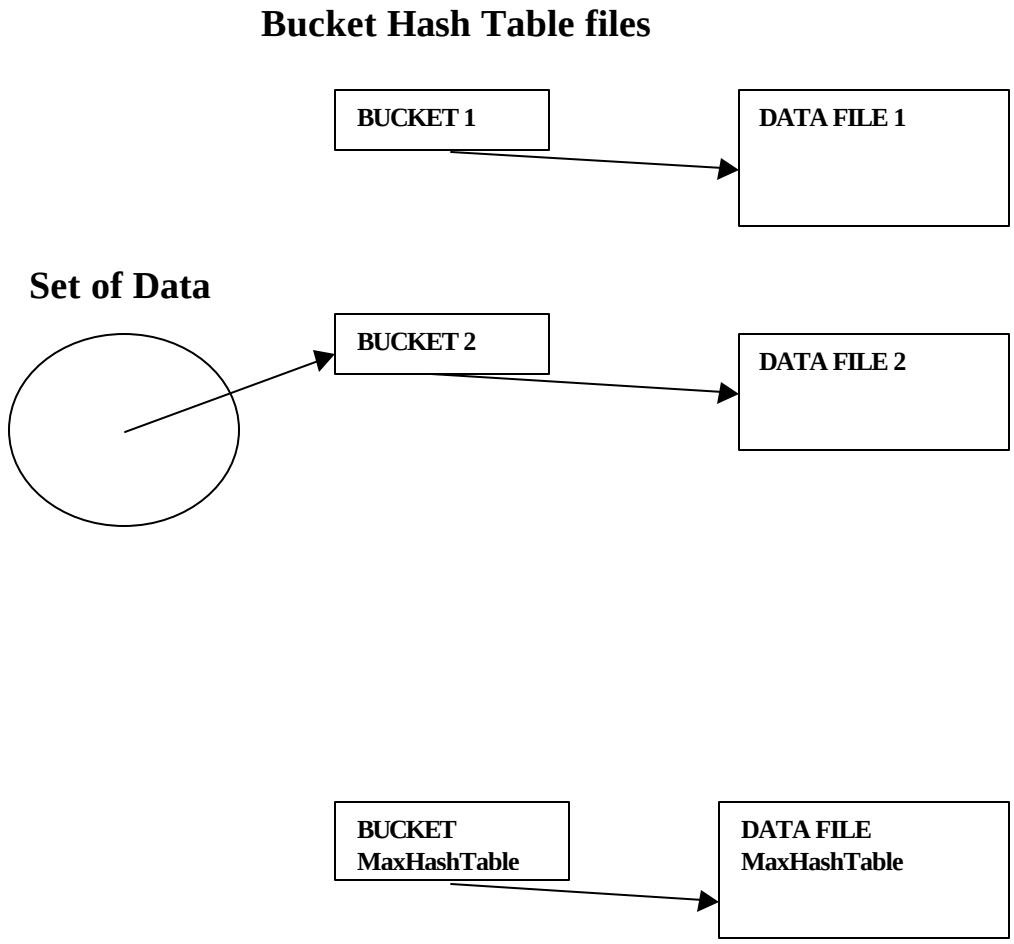
1) **RecordKey** as string of length **LRecordKey** which stores the key identifying the record, and
2) **Pointer** as numeric variable to point the record in the file where the data containing the **RecordKey** is located

The **Figure 1.0** shows how a given data record is hashed into a bucket which (if exists) points to the data record.



Local data Figure 1.0

If the data file were distributed on a set of files either local or on a network then the figure 1.0 would look as figure 1.1



Distributed Data Fig. 1.1

For local data sets the parameter **BucketCapacity** will depend on disk or massive storage device sector length and **LRecordKey**. For instance if **LRecordKey** is 12 bytes, the length to store a pointer takes 4 bytes, then a bucket record of **BucketLenght**=512 bytes will be enough to hold **BucketCapacity** =32 entries ((12+4)*32 = 512) at the bucket hash table.

However for distributed data sets, the **BucketCapacity** will depend on each node (1 to MaxHashTable) storage capacity.

In either case the maximum bucket record **MaxHashTable** will depend of the size of the Data Set, the **BucketCapacity**, N and the load factor. (also named in the literature as the **hash table density**) by the following relation:

$$\textbf{MaxHashTable} >= \frac{\textbf{N}}{\textbf{BucketCapacity} * \textbf{LoadFactor}}$$

Let **hash(Key)** be a hash function which maps the set of record keys into the set { 1,2,3,……,**MaxHashTable** }.

In the search and insertion procedure the standard bucket hash technique tries to search for a given record key in the **hash(key)** bucket. If it is found, then the data record is retrieved from the data file given by the **pointer**. Otherwise the key is inserted in a free entry at the bucket or eventually in an overflow bucket located in an overflow area, usually at the end of the table.

While this standard technique is useful for load factor of less than 0.50 (% 50 full), the management of the overflow problem is very time consuming.
Moreover, searching for a record key located in the overflow area may take several access operations at the bucket table. For heavy a load factor, say more than 0.80 the deletion process is quite complex [7].

The forced digit modification [2]

Now assume that in the insertion process we not only search for a free place in the **hash(key)** bucket, but also in its neighborhood, **hash(key)+1, hash(key)+2, ……., hash(key)+9**. At this point two approaches can be taken to assign a forced digit:

- *First fit*: choose the **first non empty** bucket
- *Best fit:* choose the **emptiest** bucket.

In either case if **hash(key) + ForcedDigit > MaxHashTable** then the bucket entry may take all the way around and assign bucket:

hash(key) + ForcedDigit - MaxHashTable.

Furthermore, within a bucket the keys could be stored in order so that there is always a quick sorted reference of the Data records (like in merge-sort schema).

The **ForcedDigit** is returned to the user and **attached** to the key for future reference as similar as the checking digit was assigned to every key in a checking digit system.

This technique of forced digits may be thought as a **user-cooperating-search schema** since, for faster access, the user has to provide the key and additional information by means of a single digit. The user provides the key and the system attaches a digit to form a new key (Key plus forced digit).

No claim is made that this schema is for general purpose applications. The keys may exist in contexts where the user has no control over them. For instance, in medicine product names, it should be impossible to assign a digit for every name. However, if the user concern is with the designing of the keys as in stock code items or personal ID numbers, attaching a digit may be taken as part of the coding process.

In searching for a given key, if **ForcedDigit** is known then the key retrieval takes ***only one disk access*** into the bucket **hash(key)+ForcedDigit**. If the key is not there no further search is required; the key is not in the file. Overflow buckets will not be used.

However, should the user forget the forced digit, it may be found by searching into at most 10 (eventually consecutive) buckets, **hash(key), hash(key)+1,……., hash(key)+9.** On distributed data sets the idea of consecutive does not necessarily means geographic proximity.

The fact of using a decimal digit is a matter of human factor. Everybody is aware of the effort to remember some extra digit assigned if it help the systems performance. However, a few people will be glad to remember a large number of digits or characters for the same purpose.

Simulation Results for a local Data Set

The following table summarizes a simulation results assigning by **Best Fit** for values of N from 5000 to 50000 and bucket length of 2,4,8,16 and 32. An entry shows the load density reached at the first overflow (when 10 consecutive buckets are full). For each entry the result is the average of three different runs.

	Bucket Capacity				
N	2	4	8	16	32
5000	0.4556	0.7386	0.8750	0.9390	0.9702
10000	0.4942	0.7020	0.8369	0.9276	0.9702
15000	0.5339	0.7451	0.8609	0.9122	0.9561
20000	0.5678	0.6991	0.7839	0.9135	0.9323
25000	0.5236	0.7218	0.8194	0.9060	0.9592
30000	0.5149	0.6700	0.8340	0.9293	0.9509
35000	0.5595	0.6484	0.7938	0.8833	0.9470
40000	0.4943	0.6657	0.8420	0.8762	0.9433
45000	0.4932	0.7065	0.8246	0.8965	0.9464
50000	0.4682	0.7275	0.8181	0.9054	0.9351

From a practical standpoint this approach may start with a load factor about 0.80 and then allow the data file to grow on its own dynamic of insertion and/or deletion.

However, for a load factor maintenance (say below 90%) from time to time it can be ran an utility to expand, shrink or eventually to rehash the bucket hash table to keep the load factor under control. In case of rehashing, to expand the table, the forced digits already assigned must be preserved.
On the distributed data arena, each node (or archive) may have different storage capacity. However, this fact maybe taken into account to assign forced digits to keep the nodes as balanced as possible.

Besides [2], other approaches to one-access somehow related to this paper can be found in the literature [1],[6],[7].

Future extensions

Instead of a forced digit it may be possible to attach a **forced character**, or some allowed set of characters For instance, by allowing digits plus upper case letters we may extend the consecutive buckets from 10 to 36.

The simulation results for Bucket Capacity = 32 and 36 character allowed, the load factor for N=50000 was 0.9716 and for N=6000 was 0.9950. (only 30 keys out of 6000 did not fit).

Another research direction in the local data arena may be the following: while inserting a new key when an overflow is detected (i.e. 10 consecutive buckets are full) choose a key inside one of these consecutive buckets and rehash it by assigning another ForcedDigit in order to make room for the new key. This characteristic is appealing whenever the user can delay the release of the key codes until all keys are inserted.

Another extension could be in user/password identification. In systems where an unique ID is necessary, a digit may be attached the user/password to speed up the retrieval operation. In this case the user should have to remember one additional digit to his/her password.

Conclusion

A novel approach to one access retrieval has been presented. This method may not be applied everywhere. We just claim that the approach has been highly useful in designing medium size information systems. It is simple to program and shows a robust behavior.

Acknowledgment

The author whishes to thank Prof. Claudio Delrieux and Guillermo Kalocai, from Departamento de Ingeniería Eléctrica, Universidad Nacional del Sur, Bahía Blanca, Argentina, for a critical reading of the manuscript.

Appendix

Some of the principal routines in pseudo (Pascal like) code are given:

Searching for a given **Key (** its **ForcedDigit** may not be given **)**

if ForcedDigit exist **then**

```
  Begin
      Remark: ONLY ONE DISK ACCESS
      If  Key is in the Hash(Key)+ForcedDigit bucket then
            Key is found
      Else
            Key is NOT found;
  End
Else
  Begin
      Remark: the ForcedDigit does not exist ( is not given )
      If there is a ForcedDigit (0 to 9) such that Key is in the
            Hash(Key)+ForcedDigit bucket  then
            Key is found
      Else
            Key is NOT found;

  End ;

If Key is found then
      Return the Data Record pointed by Pointer
Else
      Return KEY NOT FOUND;
```

Insert a new **Key**

```
If Key is NOT FOUND then
  Begin
      Remark: Insertion by Best Fit
      Choose ForcedDigit (0 to 9) so that the bucket Hash(key)+ForcedDigit
      is the emptiest. Then look for a free record in the Data File Pointer,
      fill the entry at the Bucket with Key and this Pointer and return
      the ForcedDigit.
  End
Else
```

Return **KEY EXIST;**

Delete a Key

If Key is found **then**
Begin
Mark as empty the corresponding entry at the Bucket
End ;

References

[1] Cesarini, F., Soda G. A Dynamic hash method with signature. ACM Trans. on Database Systems, Vol 16, 2 (1991).

[2] Fontao, R. O. Forced Hash: A simple One Access retrieval Method. ICIEY2K, UBA (April 2000).

[3] Hilford, V. et al EH* - Extendible Hashing in a Distributed Environment COMPSAC '97 – 21st International Computer Software and Applications Conference (1997).

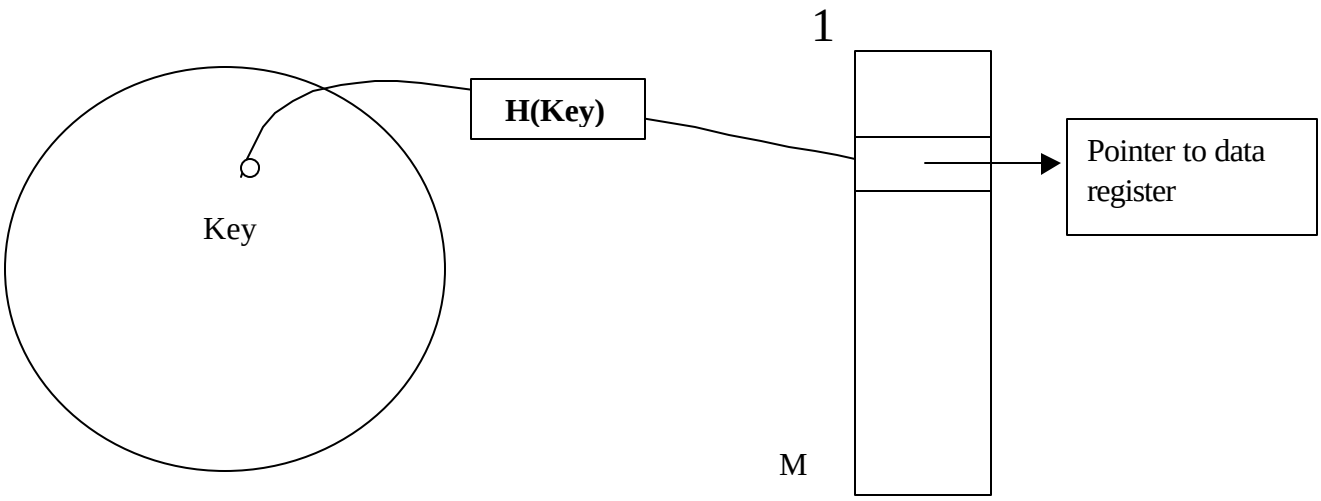
[4] Knuth, The Art of Computer Programming. Vol III. "Sorting and Searching" Addison –Wesley, Reading, Mass (1973)

[5] Kun-Lung, W. and Yu, P. Load Balancing and Hot Spot Relief for Hash Routing among a Collection of Proxy Caches. Proc. IEEE International Conf, on Distributed Computing Systems. (1998).

[6] Larson, P. Kajla, A. File Organization: Implementation of a Method Guaranteeing Retrieval in One-Access. Communications of the ACM 27(7) July (1984).

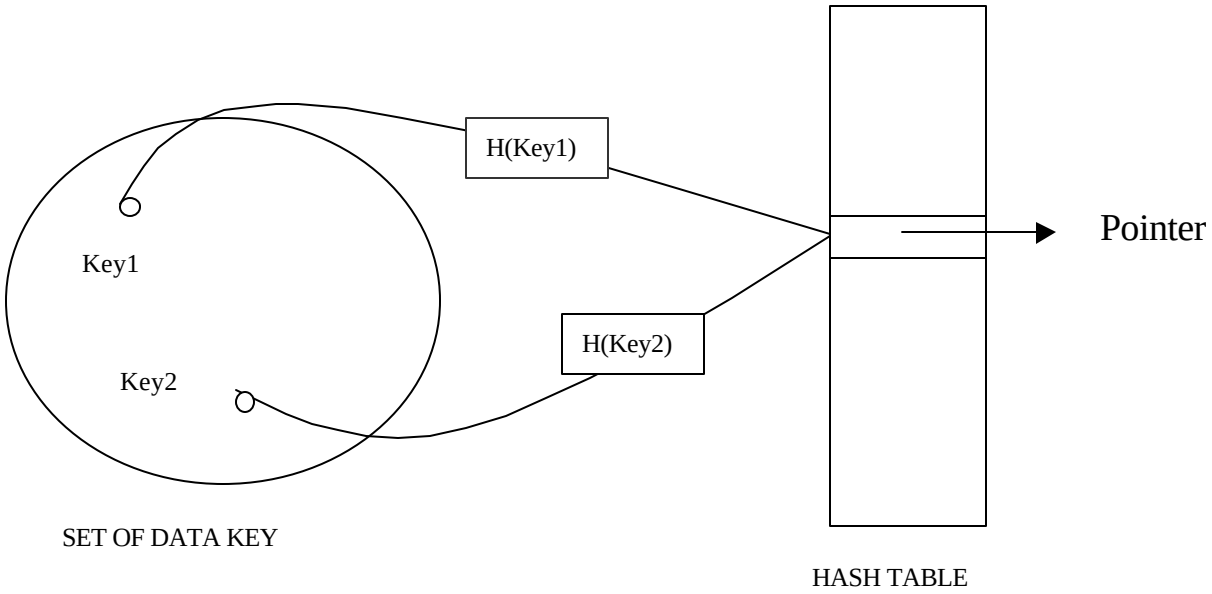
[7] Larson, P. Linear hashing with separators- a dynamic hashing scheme achieving one-access. ACM TDS Vol 13, 3,(1988)

[8] Martin, J. Computer Data-Base Organization. Prentice-Hall, (1977)



Objetive: Find a function
H:SET OF DATA KEY → {1 .. M}
such that it scatters the SET OF DATA KEY into a pseudo random address 1..M

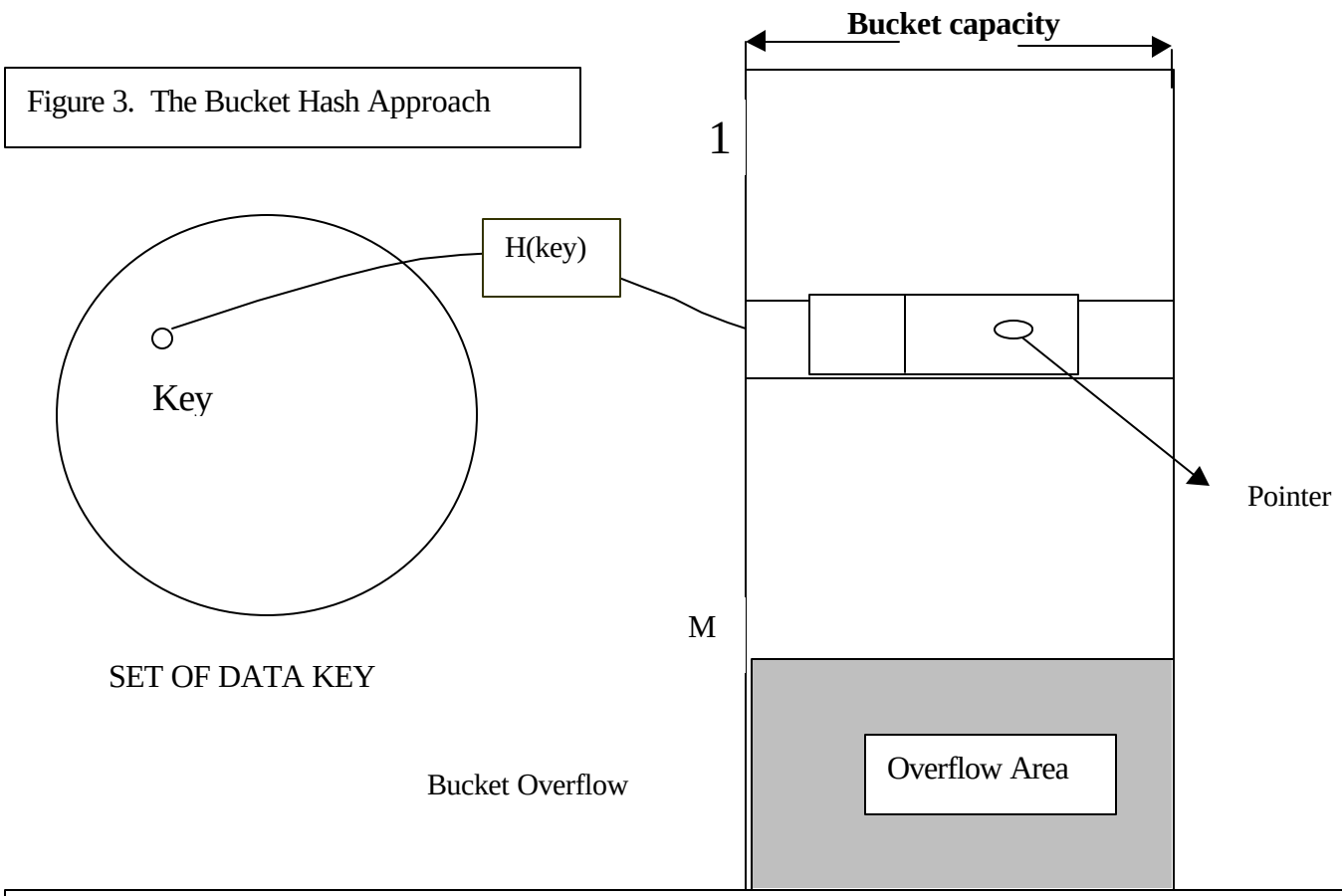
Figure 1. THE HASHING APPROACH



IF Key1 not equal Key2 AND h(key1)=h(key2) then **Collision hash**

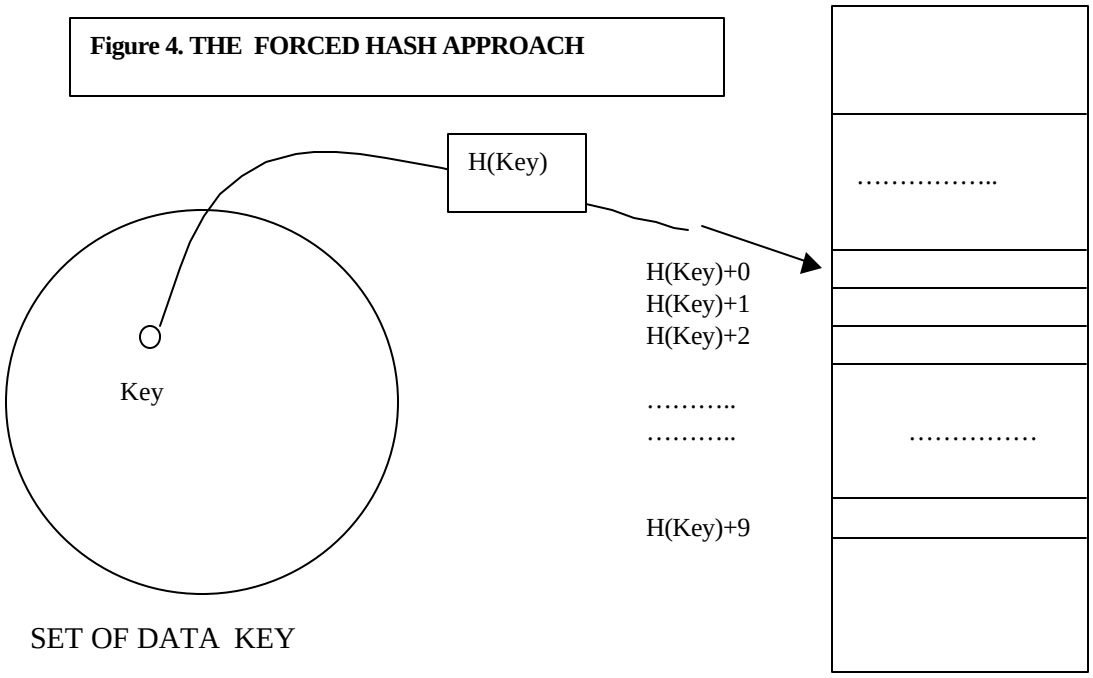
Figure 2. THE COLLISION PROBLEM

Figure 3. The Bucket Hash Approach



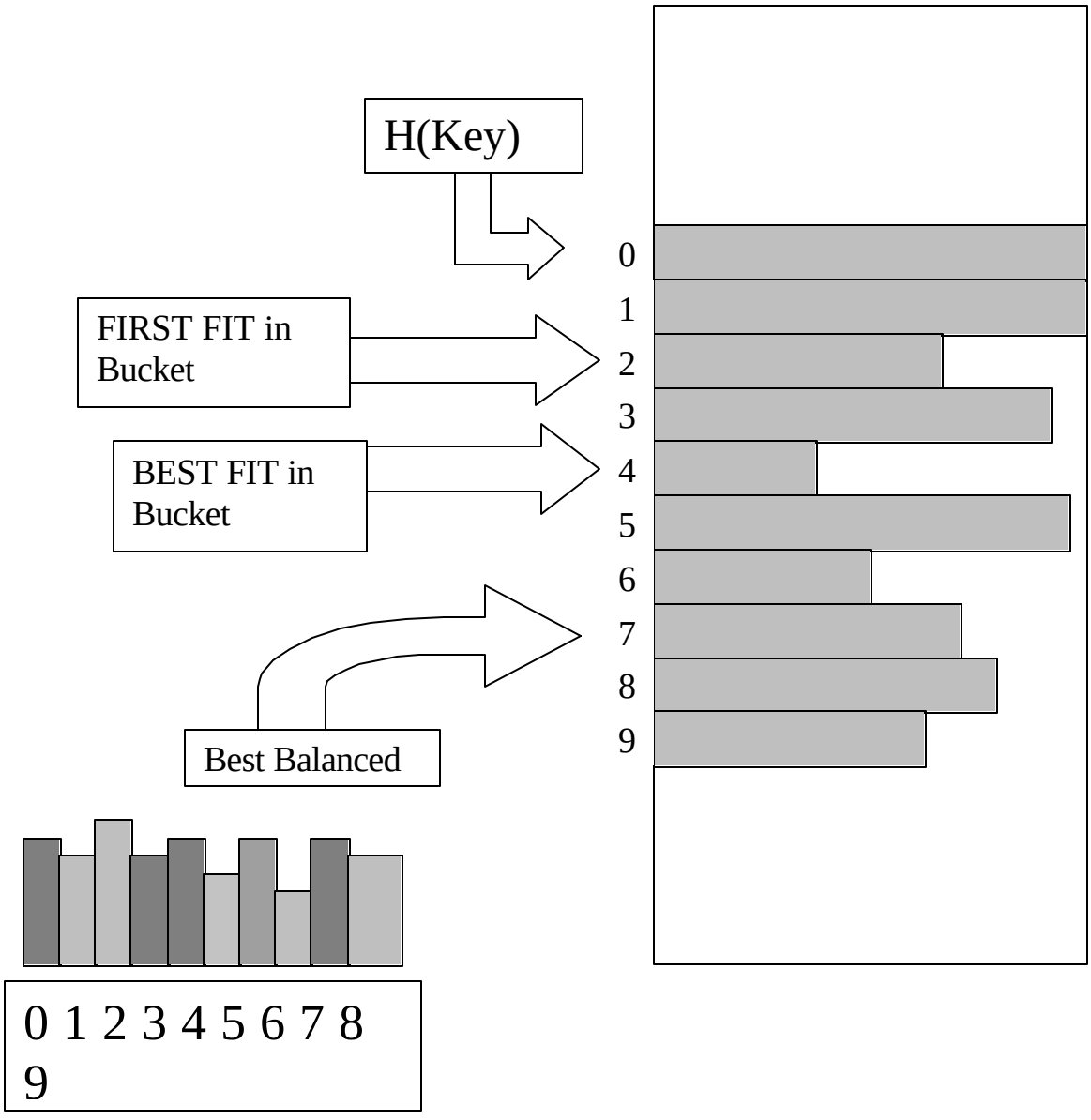
When the buckets become nearly full, its necessary an Overflow Area, but…**Insertion and deletions** are complicated**.**

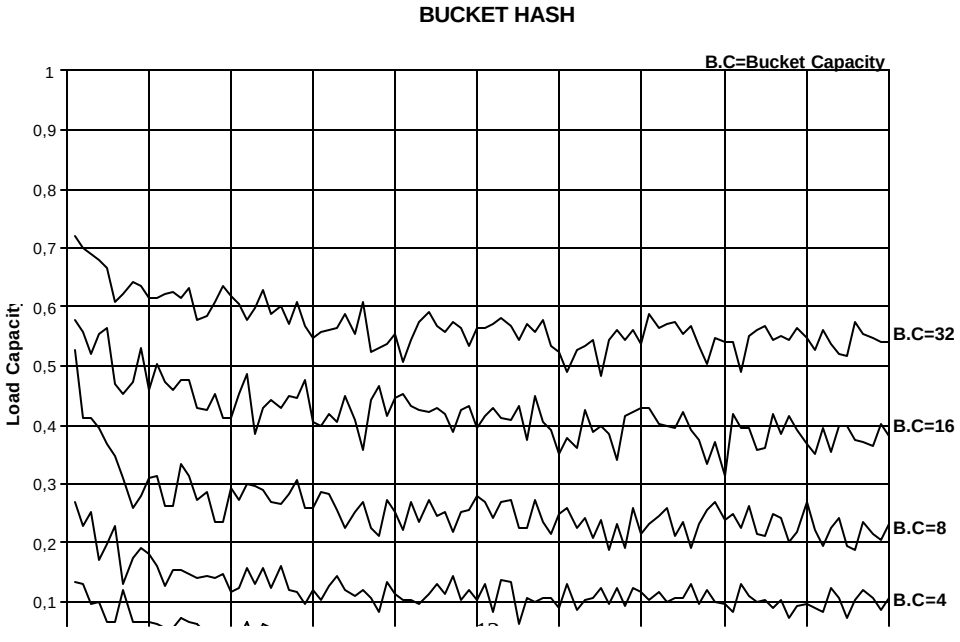
Figure 4. THE FORCED HASH APPROACH



For each Key associate a digit (forced digit) to form a new Key-Digit key so to find in only one table access then data record pointer.

Figure 5: HOW TO ASSIGN FORCED DIGITS





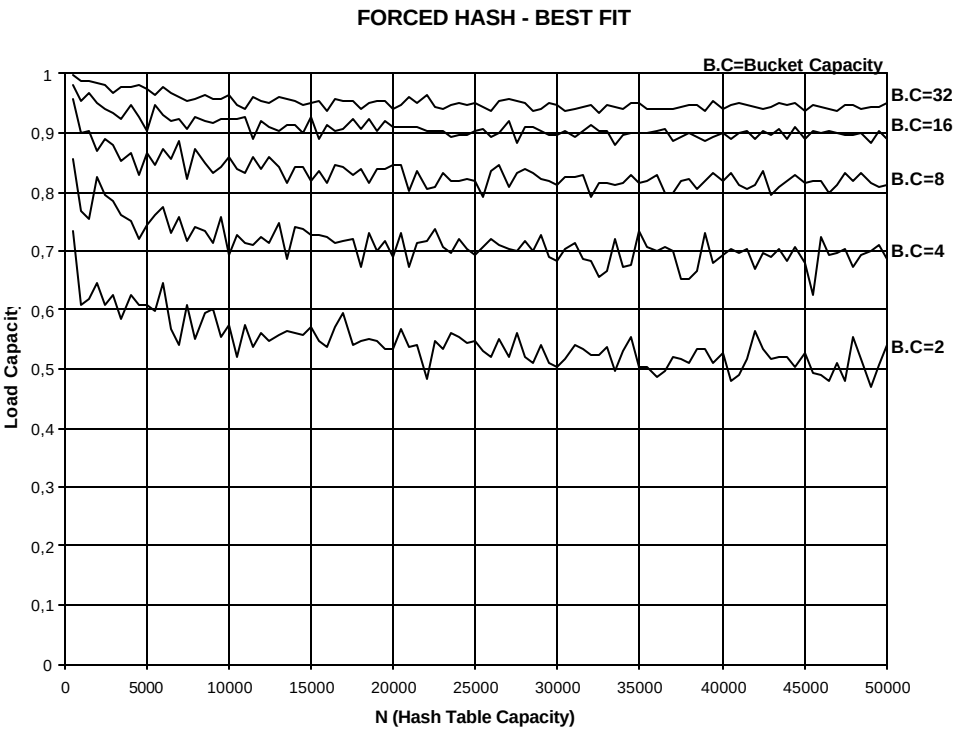


Figure 6

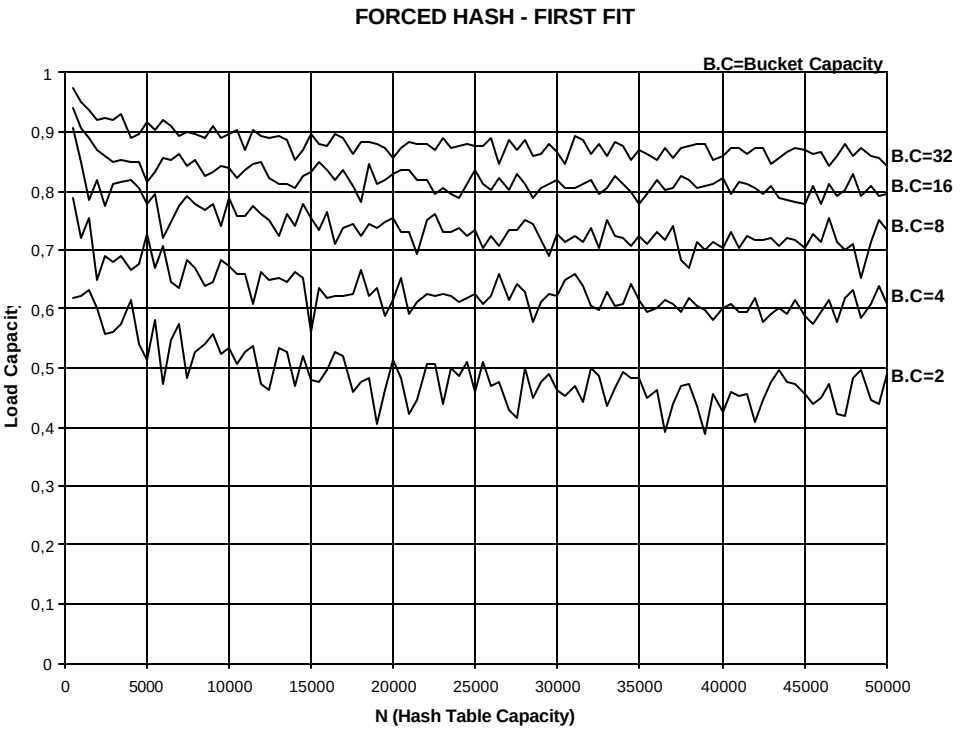


Figure 7

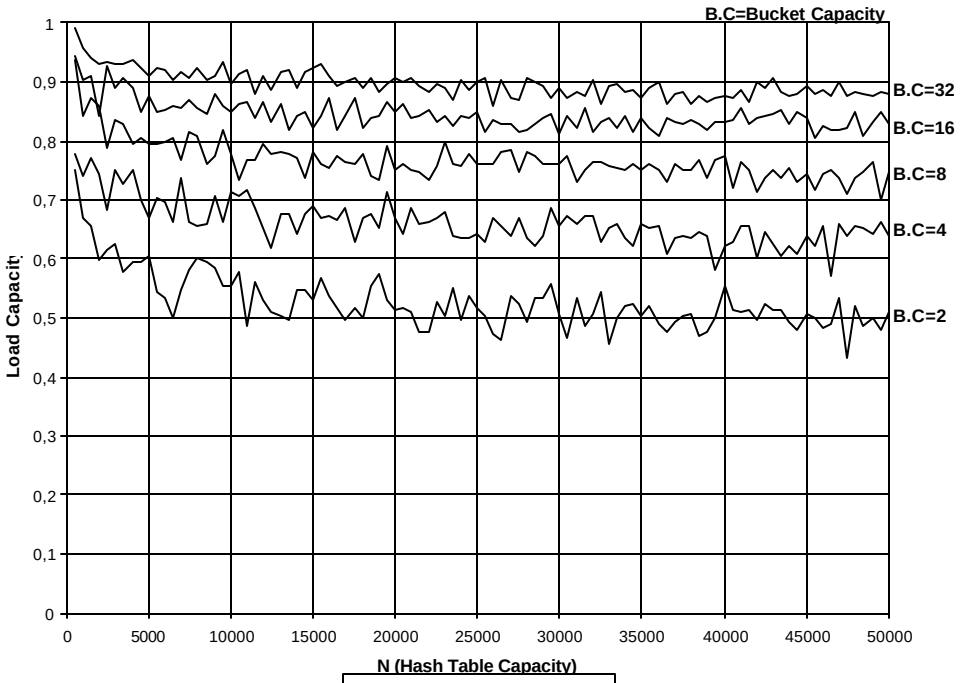


Figure 8

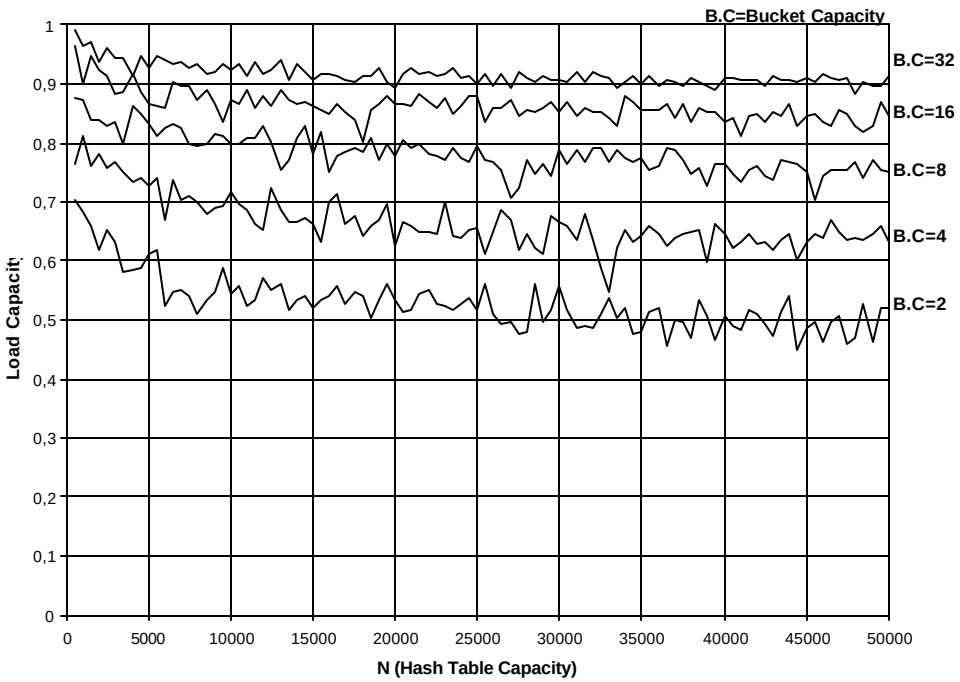


Figure 9