Hash Trees Versus B-Trees

D. A. Bell
School of Computer Science, Ulster Polytechnic, UK

S. M. Deen
Department of Computing Science, University of Aberdeen, UK

The hash trees method of external hashing is known to have advantages for certain types of primary key distribution. In this paper the value of the method as a general indexing technique—for secondary keys as well as primary keys—is assessed, and a comparison with the B-trees method is presented.

1. INTRODUCTION

The problem of organizing large files of data on secondary storage devices has been recognized since the earliest attempts at applying computers to handle large-scale information systems. Retrieval efficiency can only be attained by proper and judicious organization of the records in the file. Most of the early solutions to emerge involved fairly crude methods of hashing, indexing, linking and overflowing to deal with direct access to data items and growth of data volumes. Sequential access on some key was achieved generally by contiguous placement. Only recently—within the last decade—have attempts been made to adapt methods traditionally used only in primary storage for data access. Much of the motivation for this has come from the advent of database management systems (DBMS), which allow a single conceptual view of a database consisting of many integrated data files which are potentially related on many keys rather than a single one. Traversal between the files constituting the database typically require more complex access paths than are supported by traditional methods—for example, direct linking on several keys rather than just one, and sequential access on more than one key. Also the control software has to keep track of the available space to be left for insertions and freed by deletions on many different files. The inflexibility of the crude traditional access structures and strategies forces recourse to even cruder reorganization measures when the cost of high occupancy levels of the originally assigned data accommodation is intolerably high in performance terms. Also, the basic tree structures for direct access are not very well balanced in the sense that much more probing is required at some parts of the file than at others.

The methods of Refs 1–3 all claim advantages over traditional methods for some mode of access to data or some particular distribution of primary and secondary keys. These methods are collectively called *external hashing schemes*, and their features have been compared elsewhere[4] and in Table 1.

Two particular methods from the repertoire of external hashing methods are compared in this paper. B-trees, which can now be considered to be a classical tree method, first became an official contender in 1972 when, Bayer and McCreight[5] proposed the method and claimed relatively inexpensive costs for insert, delete and direct access operations.

The method of hash trees, H-trees, was first introduced in publication in 1981 by Deen *et al.*,[6] although it appeared as early as 1978 in internal reports, and the method is claimed to support the insert, delete and direct and sequential access operations, but also to promote immunity of indexing to changes in data addresses.[7]

In the next section we outline the features of external hashing methods, and in Sections 3 and 4, describe briefly the H-tree and B-tree methods. In Section 5 we examine their performances for direct and sequential access. A summary of their strengths, weaknesses and relative appropriateness appears in Section 6.

Table 1. Comparison of external hashing schemes

	Sequentiality F	Sequentiality P	Graceful overflow	Secondary indexing facilitated	Direct access	Load factor
D-hash	×	×	✓	×	1–2	69%
X-hash	×	✓ (weak)	✓	×	1–2	70%
T-hash	✓	✓	✓	×	1–2	70%
B^+-trees	✓	✓	✓	×	3–5	70%
H-trees	✓	✓ (weak)	✓ (weak)	✓	1.11 (10%O/F) 1.33 (30%O/F)	85–90%

N.B. (i) *F* and *P* are as in Figure 1. (ii) Other contenders are either very complex, have a weak load factor or require re-hashing for collision-handling.

2. EXTERNAL HASHING METHODS

A *hashing function* is a transformation which maps an identifier attribute, or key, to an address location. Often this transformation is to an intermediate directory, as in Fig. 1, instead of directly to the address space. This is particularly true of the enquiries made on the file involving predicates on several different attributes—sometimes referred to as primary and secondary keys.

In Fig. 1 the *key space* of a file for a particular key is a collection of elements each representing a possible value of the key. The collection of available secondary storage location addresses in which the records are actually stored are referred to as the *address space* of the file, or the *data pages*. A *directory* provides a level of indirection between the key and address spaces, and can facilitate direct and sequential access. So when the 'logical' address has been determined by hashing the key, the 'physical' address can be found by inspecting the contents of the

CCC-0010-4620/84/0027-0218$03.50

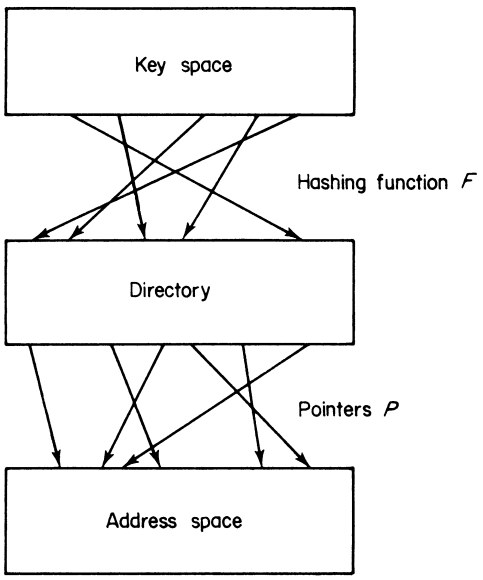

Figure 1. General model of external hashing schemes.

corresponding directory entry. Once this address is known only 1 access is required to get the data associated with the key.

Dynamic hashing (D-hash),[1] extendible hashing (X-hash)[3] and trie hashing (T-hash)[2] allow graceful expansion and contraction without the drastic reorganization due to overflow problems. These all use a form of 'splitting' which involves some method such as the following. When a leaf directory node points to an overflowing data slot it migrates up a level in the directory tree and points to two leaf nodes. The right node points to a new data slot accommodating a share of the tuples from the original data slot, which retains the rest of the tuples itself. The reverse process is applied for dealing with shrinkage of data. Table 1 summarizes the features of these methods.

3. B-TREES

B-trees are fully described by Bayer and McCreight[5] who present algorithms for retrieval, insertion and deletion and give a precise analysis of the costs.

In B-trees the structure between nodes—assumed initially to be placed on a page each—is that of a multi-way tree, with a leaf page having the occupancy (λ) level set at $\frac{1}{2} \leq \lambda \leq 1$ for all pages except possibly the root node. Each index page contains keys and pointers, and if there are $2t$ keys and $2t + 1$ pointers on a page we say that the B-tree is of order t. Within leaf pages the keys are maintained in sequential order.

The basic method for searching proceeds as follows. Choose one of $2t + 1$ paths (pointers) at each node on the basis of the relative position of the search key in the sequence of $2t$ keys. This decision procedure is repeated at each node until either a hit is made or the key is discovered to be not present. This method clearly eliminates the hashing function F in Fig. 1.

The number of keys n at each node can in fact vary, but must be in the range t to $2t$, with $n + 1$ pointers in all cases. This implies that each node is at least half full, which preserves the balance of the tree and avoids catastrophic reorganization through this incremental reorganization: when a node is assigned too many keys, the contents of the node are split into two approximately even groups, which are stored in two sibling pages. One or more of the directory levels above this node then receive an injection of a discriminator between the contents of these nodes. The reverse process occurs when a node's occupancy is reduced to below one-half of its capacity, and a concatenation with a sibling is dispatched.

Many nodes of the B-tree may therefore be considerably underfilled, and thus the height of the tree may be greater than is necessary. However, this excess may be assumed to be only 1 level in practice, which represents the cost of balancing.

B-trees do have a guaranteed worst case number of accesses per retrieval. This number can be reduced further by refinements of the basic method, and more complex redistribution techniques.

In B^+-trees all keys appear in the leaf nodes—the superstructure of the higher nodes being a pure index. This means that non-existent key values may appear in the superstructure e.g. even after a key has been deleted from the leaf nodes. Also since all keys are in the leaf node deletion and insertion complexity is reduced. The leaf nodes are linked together to enhance sequential traversal of the leaves. Because real keys need not be stored in the upper leaves of the tree, the discriminators may be more compact than keys. Prefix B-trees allow the shortest string which discriminates between lower nodes to be stored in the higher index pages, so that space is saved and more discriminators can fit on a node, reducing the height of the tree, and with it the seek time. Most of the analysis and comparison which follows refers to the basic B-tree method, with comments on the effect of refinements as appropriate. In implementations additional information is usually stored in the nodes to show the number of keys residing in the node pages.

The assumption was made at the start of this section that nodes and pages were synonymous. However in the index itself clearly for trees of low order, many index entries may be accommodated on reasonably sized pages. However some embedded free space, in the form of slots which can generally hold many index entries, should be allocated on each page to accommodate splitting. A 'global overflow' area may also be required to supply new slots for nodes which cannot be accommodated on 'home' pages or siblings. In this respect there is a similarity with H-trees described in the next section.

A basic problem of B-trees, in common with many other extendible hashing schemes, is that the splitting of data pages required when more than $2t$ keys are assigned to it, means either migration of records in the data pages, or loss of sequentiality within pages. This problem is overcome by H-trees by using 'impure' surrogates as described in the next section.

4. HASH TREES

This method has been described in detail in Refs 4, 6 and 7. However in these documents only the implementation of the method in the PRECI system[6] was discussed. The H-trees method can be considered as an alternative to other techniques such as B-trees as a method for accessing data, both sequentially and directly, via indexes.

In its primary mode of use the method centres around the allocation to each record in the file of a surrogate composed of a file number concatenated with a relative address called the effective key. This effective key is determined by using a hashing function on the primary key—the data attributes most frequently used for access—and a compact directory called the SURROGATE DIRECTORY, whose structure is fully described elsewhere.[7] This hashing function is chosen so that records are stored on the data pages in the same sequence as this primary key. To minimize the standard deviation of the page occupancy, σ, the primary key space is compressed during hashing, and this procedure is described and evaluated in Ref. 4.

The surrogate is in fact 'pseudo' in the sense that it is determined by primary key value and so is subject to change if the primary keys change. This contravenes the principle that the surrogate is fixed for the lifetime of a record. However the frequency of change of primary keys may be assumed to be low in most realistic cases for simple practical reasons, and the pseudo surrogate retains the basic character of a record identifier which has permanency.

This permanency rules out the notion of splitting as encountered in B-trees and other methods, where the address may change. Thus overflowing techniques are needed to accommodate file volatility.

In H-trees records are accessed using a combination of hashing and indexing. For primary keys the hashing usually gives direct access by surrogate, but an index must be used for sequential access. The index entries are of the form (key, surrogate) to allow storage independence while allowing fast direct access. The index is organized in a single level, and the position of the slot on an index page to be accessed in order to determine the surrogate is found by hashing. Thus the model of Fig. 1 describes this method well.

Using the division method of hashing, but taking the quotient rather than the remainder as the hash function, F, the sequence of the keys is preserved as they are allocated to slots on the index page. So sequential access to the records by any key is achieved by chaining consecutive index pages, because the pages partition the key-ordered record set.

However while the quotient method is suitable for sequence preservation, it is manifestly expensive in storage for sparse key spaces. Hence a compression technique is employed to concertina the index-address space. A technique for this is to use as the divisor in hashing, a constant number reflecting the density of the key-space occupancy. Then expand the address space for the over-populated areas, and contract it for the under-populated areas. This expansion and contraction is shown in Fig. 2 where (a) shows the occupancy distribution of the key space for a slot size ω. The empty areas are removed in Fig. 2(b) and the dense areas thinned in Fig. 2(c). Doing this necessitates an overhead to keep track of the extent of expansion and contraction so that the address produced by the hashing function F can be reduced or increased. This overhead currently takes the form of a table T with an entry for each hash address which is preceded by an under-occupied slot or which is itself over-occupied. This is illustrated in Table 2. A simple look-up of this table, which can be stored permanently in primary storage in many practical

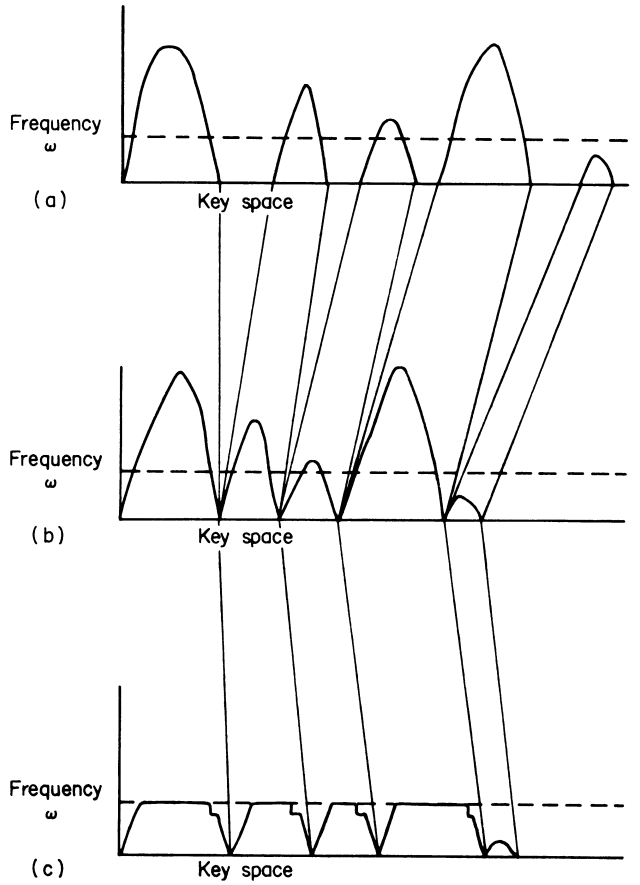

Figure 2. Reducing variation in a key distribution.

situations,[4] allows calculation of the true index slot and hence page for the key.

An alternative method of storing the table T has been designed to trade (primary) storage space for table inspection time. In the method just outlined the search is carried out using some method such as binary search in order to minimize the internal computer time. However since processing time in primary storage is usually deemed negligible in comparison to seek, transfer and latency times for accessing data in secondary storage, it is worthwhile to consider techniques for reducing the size

Table 2. Composition of table T

	Table T Entries			
Notation	f_i	d_i	e_i	Z
Meaning of entry	Hashed slot number $F(K)$	Number of additional slots needed to hold all tuples hashed to this slot (f_i)	Cumulative number of empty slots with addresses less than f_i	Addresses of slots actually holding tuples assigned to f_i
Example entries	53	0	25	28
	59	2	30	29, 30, 31
	62	0	30	32

Note: the example entries show two sparse areas in the primary key distribution (namely slots 54–58, and 60–61 which had zero occupancy), and a dense area (2 additional slots, or 3 slots altogether, were needed to accommodate all tuples hashing to slot 59).

of T so that it has a higher probability of fitting into the available primary storage space—even if this may adversely affect the performance of the algorithms to handle it.

The basic idea of this alternative technique is to divide the space taken by T into fixed zones, each subdivided into a set of double-byte entries. Each entry has two components, RS_i showing a number of occupied slots and ES_i showing a following number of empty slots. For example consider a series of entries as in Table 3. This Table shows that there is no gap in the first 256 slots (entry 1), then slot 257 is followed by 250 empty slots, 100 occupied slots and 150 empty slots.

If an entry lies in zone k of T, whose first slot number is b_k (information held in a header in T), then the slot corresponding to entry j in k is

$$X_j^k = \sum_{i=1}^{j} (RS_i + ES_i) + b_k$$

This indicates the method of stepwise sequential table look-up of the pertinent zone in T, and the true slot can be obtained by subtracting $\sum_{i=1}^{j} ES_i$ from X_j^k. This method is attractive if gaps are generally large and few.

An alternative method of key comparisons may be possible when the key distribution is generally dense but has forbidden character contributions. The Appendix to this paper indicates an approach to this which removes the need for Table T.

The key values are compressed and hashed to find an index slot where each key value and its surrogate are stored. Each index slot has a fixed hash width (HW) and each index page has perhaps 30% of its space reserved for local overflow slots to reduce probes. Global overflow index pages may also be declared to accommodate page overflows. Owing to the presence of local overflows, only 1 in 10 requests might be assumed to result in accessing the global overflows. These may be reduced by the periodic reorganization of the H-tree. Each slot contains a header followed by a set of key values—surrogate pairs (K_i, S_i). Each of these pairs has associated an optional number M_i which shows the number of other records in the database which have this K_i value as a foreign key. This allows an integrity check; M_i must be equal to zero before this associated record can be deleted. The header is a triple (N, P, C) where $N(P)$ is the next (prior) index slot (home or overflow) in key sequence, and C is the current number of values in this slot.

The entries are held in key sequence with insertions being made in sequence. Overflow slots are allocated dynamically and exclusively as required. Splitting (and concatenation) of slots is allowed within the overflowing and overflow slots within the index to promote balance.

5. COMPARISON OF H-TREES AND B-TREES

Direct access

It is easily shown that if there are between t and $2t$ keys/node in a B-tree the number of probes (levels of tree) required is

$$1 + \log_{t+1}\left(\frac{n+1}{2}\right) \text{ for } n \text{ keys} \qquad (1)$$

Table 3

Zone k =	1		2		3	
	256	0	1	250	100	150
	RS_1	ES_1	RS_2	ES_2	RS_2	ES_3

If the root level of the B-tree is held in primary storage then a 2-level B-tree has superior access characteristics to H-trees because only 1 probe is required to get the address compared to up to 1.10 probes for H-trees.

However, owing to the above formula for B-trees, the number of levels of the tree quickly increases with the number of entries n, and for more than 2 levels the H-tree performance, which remains at the 1.10 probe level irrespective of the size of n, is attractive—i.e. for large files. This is because the index super-structure is replaced by a hash algorithm.

A special attraction of H-trees is that for one key, namely the primary key, direct access can be optimized in that the directory addresses are actually the surrogate pages themselves. That is, no intermediate directory access is needed to find surrogates—so that a probe is eliminated.

Sequential access

B-trees. Processing the 'get next' operation is not really catered for in the basic B-tree organization method. It may require tracing a path through several nodes before reaching the desired key. Assume there are t keys/node; let the number of tuples be n $(= h + g)$ where h is the number in home slots and g is the number in overflow slots; let the average leaf node have l keys. Then a 'level traversal', of 2 probes, is required every l tuples, as the relation is scanned in sequence, i.e. there are $2n/l$ such probes needed in total. In addition there are another 2 probes every $t + 1$ of these $2n/l$ probes, i.e. $2n/(t + 1)l$ more probes *and* another probe for every $t + 1$ of these, i.e. $(2n/(t + 1)^2 l)$ probes and so on. . . .

Assume, for simplicity, that there is a k such that $(t + 1)^k = n/l$. Then total of probes is

$$2\frac{n}{l}\left(1 + \frac{1}{t+1} + \frac{1}{(t+1)^2} + \cdots + \frac{1}{(t+1)^k}\right) = 2\frac{n}{l}\frac{((t+1)^{k+1} - 1)}{t(t+1)^k}$$

Therefore

$$\text{probes/tuple} = \frac{2n}{lt}\left(t + 1 - \frac{l}{n}\right) \qquad (2)$$

Hash-trees. Assume that hash slots are placed in their own sequence, that the page size is p, that the number of key values on home pages is h and that the number of key values on global overflow is g on s slots. We expect $g \leq h/10$. Normal buffer replacement requires h/p probes (assuming that all home pages are full). Overflow access

requires g/s probes. Therefore the total number of probes required to get $h + g$ tuples is $(h/p) + (g/s)$. So

$$\text{probes/tuple} = \frac{hs + pg}{ps\,(h + g)} \qquad (3)$$

Comparison. Compared with the basic B-trees method the H-tree method is much superior—especially if it is possible to estimate the volatility (expansions and contractions) of the file accurately at design time. This will mean that g is small in formula (3) above and so will minimize probing.

However the B^+ variant of B-trees allows direct pointers between leaf nodes for sequential access, and so no higher level probes are required when all the tuples corresponding to a leaf node have been accessed in sequence. This will give similar sequential access to that of H-trees in most cases because the handling of irregular overflows is balanced out in B^+-trees.

Insertion and Deletion

B-trees. These operations typically require access to secondary storage in addition to the simple direct access operation. The *worst-case* penalty incurred due to reverse traversal of the tree is to double the access time so that

$$2\left(\log_{t+1}\left(\frac{n+1}{2}\right) + 1\right) \text{ probes are needed} \qquad (4)$$

Similarly for deletion.

H-trees. To insert a tuple in the H-trees method, the key value is compressed and hashed to get the slot for the index entry.

If the 'home' index slot is not full, the K_i, S_i, M_i values must be rearranged to accommodate it. Even if it is full the (half-sized) local overflows on the same page may still accommodate it.

Only when a global overflow is necessitated is the penalty of an additional probe incurred.

An approximate measure of this penalty is (using notation introduced earlier) h/g for $(h + g)$ tuples. Therefore

$$\text{probes/tuple} = \frac{h}{g\,(h + g)} \qquad (5)$$

A similar penalty is incurred for deletion. There are also overheads due to maintenance of the table T used in hashing.

Comparison. The basic difference here is that related (in key sequence) tuples are indexed in close proximity in H-trees. They are on the same page unless 'heavy' overflow causes a global overflow slot to be required. Housekeeping within a page is cheap—several slots may be adjusted for insertions and deletions without probing penalty.

B-trees vs. H-trees at leaf page level

A final comparison involves the average number of leaf pages. Yao[8] has shown, using the Bernoulli model, that if all $n!$ orders of insertions are equally likely, the average number of leaf pages in a B-tree is $(n \log e)/2t$ asymptotically. In an H-tree organization, if the average occupancy of leaf pages (data pages) is ρ, then the average number of leaf pages is $n/\omega\rho$ where ω is the total capacity (in tuples) of a page.

6. SUMMARY AND CONCLUSIONS

It is a truism that, given current technology, there does not exist a single storage and access strategy which will be the best choice for all applications and environments. The inescapable conclusion is that 'horses for courses' must be the rule for choosing between such methods for a particular set of circumstances. Although the B-trees method and its variants have been regarded for some time as good safe choices for many systems, this should not prevent additions to the physical database designer's repertoire being sought. A taxonomy of some of the major current methods was given in Ref. 4, but in this paper we have shown that the H-trees method has advantages for certain purposes.

The major scenario in which H-trees provides significant advantages over B-trees is when there is a high traffic of insertions and/or deletions. In this case methods which split data pages as described earlier are at a distinct disadvantage compared to the H-trees method which uses fixed surrogates, owing to the high level of secondary index reorganization and maintenance required.

Also for direct access to data records H-trees outperform B-trees for realistic depths of B-trees. Although B-trees have modest logarithmic cost in terms of probes as a file grows, once the number of levels gets beyond 2 then H-trees become advantageous. The number of levels depends on the order of the tree as well as the file cardinality, but for node cardinality up to about 50, the depth exceeds 2 rapidly. It should be noted that the H-trees method carries an overhead of the range–compression table which requires both storage and computation time to consult. It was shown empirically in Ref. 4 that for some real files its size was such that it could reasonably be held in each memory during its usage. It was also shown in the same study that for certain key distributions, large numbers of records could be assigned to a single hash slot by the hashing function F. In such cases, unless there is available a simple secondary hashing function, such as the displacement function used in the reported study, to get to the unique data slot, additional probes will be required. Hence there is no guaranteed (worst-case) performance, unlike for B-trees. The key compression algorithm currently in use will, it is hoped, be improved in the future, but was shown to be very useful for key distributions which have long densely—almost fully—occupied segments interspersed with relatively few, but sizeable, empty segments. Such distributions occur very frequently in many real applications—e.g. student numbers allocated consecutively from a different boundary figure—perhaps several thousand keys apart—for different disciplines in a university.

Thus for many indexing applications, H-trees are a worthy alternative to B-trees.

Acknowledgements

The authors thank the members of the PRECI collaboration who commented on this study, which was funded by the Science and Engineering Research Council.

REFERENCES

1. P. A. Larson, Dynamic hashing. *BIT* **18**, 184–201 (1970).
2. W. Litwin, Trie hashing. *Proceedings of ACM-SIGMOD* (1981).
3. R. Fagin, J. Nievergelt, N. Pippenger and H. R. Strong, Extendible hashing—a fast access method for dynamic files. *ACM TODS* **4** (3), 315–344 (1979).
4. D. A. Bell and S. M. Deen, Key space compression and hashing in PRECI. *The Computer Journal* **25** (4), 486–492 (1982).
5. R. Bayer and C. McCreight, Organisation and maintenance of large ordered indexes. *Acta Informatica* **1** (3), 173–189 (1972).
6. S. M. Deen, D. Nikodem and A. Vashishta, The design of a canonical database (PRECI). *The Computer Journal* **24** (3), 200–209 (1981).
7. S. M. Deen, An implementation of impure surrogates. *Proceedings VLDB* (1982).
8. A. C-C. Yao, Random 3-2 trees. *Acta Informatica* **9**, 159–170 (1978).

Received May 1983

APPENDIX

Key space compression using forbidden zones

An alternative method of key compression may be of interest in applications where the key distribution is dense enough not to warrant compression in the manner of section 4 except for specific regions which can be identified by *a priori* knowledge. The method described and analysed in this section is aimed at such a situation where the regions or gaps in the key sequence are derived from the designer's knowledge that particular character combinations cannot occur. For example, the knowledge that if a part number has a character E or F in character position 1, then the part coding rules mean that there can neither be a character 3, 4 or 5 in character position 7, *nor* characters X, Y, Z in position 8 or 9, indicates that there will be considerable gaps in the key sequence.

In such cases these rules may be represented in a compact table, called the forbidden table (FT) and an algorithm can replace the costly Table T in the previous method. This section presents preliminary algorithms to deal with this situation. Similar methods have been devised for knowledge of more complex 'null-areas', where (for example) triples rather than pairs of characters in given positions are forbidden, but the more complex algorithms for these more general key-coding rules are not considered here.

We use the following notation when specifying the kth rule in the 'forbidden table' as specified by the database designer. Subscript l takes the values 1 and 2, and F'_{k_l} shows the lth character in the forbidden pair; P_{k_l} shows the lth character position. Thus each entry in FT has four components specifying the forbidden combination and its positions. Throughout this Appendix we use the following notation:

C	Cardinality of character set
D	Semi-compressed key as decimal—collapsed w.r.t. $F_{k_{1,2}}$
E	Current surrogate (E' developing)
$F'_{k_{1,2}}$	$k = 1, 20$ are first, second characters, rule k
$F_{k_{1,2}}$	$k = 1, 20$ are first, second s-c digits, rule k
J	Current semi-compressed key
K	Current P-key
$L_{i,j}$	jth contender for ith character position
M	No. of forbidden character combinations (≤ 20)
N	No. of characters in key
$P_{k_{1,2}}$	$k = 1, 20$ first and second digit positions for rule k
Q_i	No. of contenders for ith character position (≤ 0)
R_i	ith character in current P-key ($R_{i,j}$ for key j)
S_i	ith digit in current S-key ($S_{i,j}$ for key j)
T_k	Total forbidden S-keys < current S-key
$U_{p,j}$	Decimal value of P digits to left of jth in S-key
W	Boolean variable
X_j	Decimal value of jth S-key
Z	Total P-keys
$\delta_{a,b}$	Increment between successive S-keys ($V_{a,b+1} - V_{a,b}$)
δ	1 if $S_x > F_k$; 0 otherwise
δ'	1 if $S_x = F_k$; 0 otherwise
δ''	1 if $S_y > F_k$; 0 otherwise

Definition 1. An *S-key* is a primary key expressed in 'semi-compressed' form, i.e. with its ith character, R_i, replaced by the j subscript of the matching entry in the contender-table $L_{i,j}$, which shows the jth character contending for the ith character position.

Narrative description of algorithm to collapse P-key K.

1. Read table of contenders for each character position $1 \leq i \leq N$. Store as $L_{i,j} : 1 \leq j \leq Q_i$.
2. Read table of illegal character combinations (F'_{k_l}) and their character positions (P_{kl}): $l = 1, 2$; $k = 1, 2, \ldots, 20$.
3. Use j-subscript of $L_{i,j}$ entry matching the ith character in K, namely R_i, as character i in semi-compressed key j.
4. Calculate corresponding semi-compressed key values of the F'_{k_l}, $k = 1$, and store as F_{k_l} (as in step 3 above).
5. Calculate corresponding total number T_k of semi-compressed keys which are less than the current semi-compressed key (see below).
6. Subtract T_k reduced by portions previously encountered in an F-rule from the current semi-compressed key, J (expressed in decimal form as D), thereby collapsing it further.
7. Repeat steps 3–6 increasing k by 1 until $k = M$. In each case discard those portions of T_k which have previously been taken into account (i.e. as a result of applying a previous F-rule).
8. Use the final value of D as surrogate E, for allocation to slots.

Formula used to calculate T_k in steps 5, 6 above

Let $x = P_{k_1}$; $y = P_{k_2}$

Definition. The *upper of p and j* ($U_{p,j}$) for a semi-compressed key J is defined as the decimal value of the semi-compressed number represented by the digits from the pth digit of J to the jth digit of J, i.e.

$$U_{p,j} = \sum_{i=p}^{j-1} (S_i\, Q_{i+1}\, Q_{i+2} \ldots Q_j) + S_j$$

Definition. A *cycle* for the kth forbidden-rule table entry is (intuitively) defined as an iteration which increases the decimal value corresponding to the the high-order digits of J (i.e. those digits to the left of P_{k_1}) by 1.

Such an incrementation occurs as the result of each complete enumeration of all semi-compressed keys which are less than a value of J with a '1' in character position $P_{k_1} - 1$, and zeros in all the remaining character positions. More concisely: a *cycle* for rule k is any complete iteration through consecutive semi-compressed keys which causes the value of $U_{1,x-1}$ to increase by unity.

Theorem 1.

$$T_k = 0, \quad \text{if } F_{k_1} > U_{1,x} \tag{6}$$

$$T_k = [U_{1,x-1} + \delta]\frac{\prod_{x+1}^{N} Q_i}{Q_y} + \delta'\left[U_{x+1,y-1i} + \delta''\right]\prod_{y+1}^{N} Q \quad \text{if } F_{k_1} \le U_{1,x} \tag{7}$$

where $\delta = 1$ if $S_x > F_{k_1}$; 0 otherwise

$\delta' = 1$ if $S_x = F_{k_1}$; 0 otherwise

$\delta'' = 1$ if $S_y > F_{k_2}$; 0 otherwise

Proof. The truth of equation (6) is easily established. If $F_{k_1} > U_{1,x}$ then, by definition, there have not previously occurred any cycles. Therefore $T_k = 0$.

Equation (7) is proved as follows: if $F_{k_1} \le U_{1,x}$ then there have been $U_{1,x-1}$ previous cycles, and possibly a partially completed (current) cycle. Then T_k is composed of the sum of two subtotals: (i) the number of occurrences of combinations in previous cycles; and (ii) the number of occurrences of combinations in the current cycle if $S_x = F_{k_1}$.

(i) Assume δ is as defined as above. Then subtotal (i) is represented by:

$[U_{1,x-1} + \delta]$ [the number of times F_{k_2} occurs for each F_{k_1} occurrence]

$$= [U_{1,x-1} + \delta][Q_{x+1}\, Q_{x+2} \cdots Q_{y-1}\, Q_{y+1} \cdots Q_N]$$

$$= [U_{1,x-1} + \delta]\frac{\prod_{x+1}^{N} Q_i}{Q_y}$$

(ii) Assume δ', δ'' are as defined above. Then subtotal (ii) is represented by:

δ' (the number of occurrences of combination this cycle)

$$= \delta'\,[U_{x+1,y-1}\, Q_{y+1}\, Q_{y+2} \cdots Q_N + \delta''\, \Pi_{y+1}^{N}\, Qi$$

because the combination has occurred $\Pi_{y+1}^{N} Q_i$ times in the current cycle, *only if* $S_y > F_{k_2}$

Note. The values of δ, δ' and δ'' can be conveniently calculated within a computer implementation of this algorithm by evaluating the following expressions:

$$\delta = \text{INT}\left[\frac{S_x + 10000}{F_{k_1} + 10001}\right]$$

$$\delta' = \text{INT}\left[\frac{S_x + 10000}{F_{k_1} + 10000}\right] - \delta$$

$$\delta'' = \text{INT}\left[\frac{S_y + 10000}{F_{k_2} + 10001}\right]$$

To reduce current T_k value in the light of overlap with previous T_k rules. For convenience we consider F-rules k' and k'' to be ordered in accordance with the following precedence rule: if $(P_{k''_1} < P_{k'_1})$ or $(P_{k''_1} = P_{k'_1}$ and $F_{k''_1} > F_{k'_1})$ or $(P_{k''_1} = P_{k'_1}$, $F_{k''_1} = F_{k'_1}$ and $[(P_{k''_2} < P_{k'_2})$ or $(P_{k''_2} = P_{k'_2}$ and $F_{k''_2} \ge F_{k'_2})]$) then F-rule k' precedes F-rule k'' in the table, i.e. the least 'significant' rules come first, e.g. the rule with E in position 3, F in position 5, precedes the rule with E in position 2, F in position 5, but succeeds the rule with E in position 3, F in position 6.

In order to determine how many of the forbidden keys for this F-rule, T_k, have already been encountered in previous F-rules, the subscripts and characters for each of the preceding rules must be compared in turn with those for the current rule, taking account of overlaps between *sets* of rules.

During each comparison there are four possible cases to be considered.

Case 1. Subscripts identical ($P_{k'_1} = P_{k''_1}$ and $P_{k'_2} = P_{k''_2}$). No overlap.

Case 2. First subscripts identical, second not ($P_{k'_1} = P_{k''_1}$ and $P_{k'_2} \neq P_{k''_2}$). There are 2 subcases:

Subcase (a) First digits not identical ($F_{k'_1} \neq F_{k''_1}$). No overlap.

Subcase (b) First digits identical ($F_{k'_1} = F_{k''_1}$). There are

$$\frac{\prod_{P_{k'_1}+1}^{N} Q_i}{Q_{P_{k'_2}}\, Q_{P_{k''_2}}}$$

duplicates per cycle.

Case 3. Second subscripts identical, first not ($P_{k'_1} \neq P_{k''_1}$ and $P_{k'_2} = P_{k''_2}$) Again there are 2 subcases:

Subcase (a) Second digits not identical ($F_{k'_2} \neq F_{k''_2}$). No overlap.

Subcase (b) Second digits identical ($F_{k'_2} = F_{k''_2}$). There are

$$\frac{\prod_{P_{k''_1}+1}^{N} Q_i}{Q_{P_{k'_1}}\, Q_{P_{k''_2}}}$$

duplicates per cycle.

Case 4. Neither subscripts identical ($P_{k'_1} \neq P_{k''_1}$ and $P_{k'_2} \neq P_{k''_2}$). There are

$$\frac{\prod_{P_{k''_1}+1}^{N} Q_i}{Q_{P_{k'_2}}\, Q_{P_{k''_2}}\, Q_{P_{k'_1}}}$$

duplicates per cycle.

(Overlap with gaps encountered in groups of previous rules must also be identified and accounted for.)