Key Space Compression and Hashing in PRECI

D. A. Bell
School of Computer Science, Ulster Polytechnic, Shore Road, Newtownabbey, Co. Antrim BT37 0QB, UK
S. M. Deen
Department of Computing Science, University of Aberdeen, Aberdeen AB9 2UB, UK

The hash trees method is classified as an external hashing scheme and its features are compared with those of other members of this class. Attention is focused on the key space compression algorithms which are an essential component of hash trees. Results of experiments to assess the performance of the algorithms on three relations needed in actual applications are given. The performance is found to depend upon the distribution of the keys. The storage utilization and sequential and direct access performance obtained make this technique a useful addition to the database designer's tool-kit.

1. INTRODUCTION

Methods of storing data on secondary storage devices so as to facilitate subsequent access in order to service queries are fundamental components of any database system. Recently several important contributions have been made in this area, e.g. Refs 1–5, all of which claim advantages for some mode of access or some particular distribution of primary keys. We refer to this class of methods as external hashing schemes and describe it in more detail in Section 2.

In a study of several more traditional storage and access methods, Lum *et al.*[6] showed that although one well-known hashing algorithm—the division method—was a good 'safe' choice in all cases considered, it was not always the best choice for a given application. Their conclusion was that 'horses for courses' should be the rule; the selector of a hashing method should take pains to exploit the structure of the key space and the distribution of the key occurrences within it when opting for a strategy, in order to optimize performance indices such as response time, storage utilization or throughput.

It is likely that a similar conclusion would be drawn from an analogous comparison between external hashing schemes, so there is unlikely to exist such a scheme which is the best choice irrespective of application or key distribution. This observation supplies the motivation for seeking novel techniques for the storage and access of data, and for investigating their performance for different applications and different key distributions, in order to determine their places in the taxonomy of such schemes. The purpose of this paper is to examine the performance of the hash trees external hashing method.[1]

The PRECI (Prototype of a Relational Canonical Model with local Interfaces) project was set up primarily as a research vehicle to carry out studies such as this and now has collaborators throughout the UK. PRECI is a generalized database system based on a canonical data model capable of providing interfaces to other major models through local schemas and host languages or a query language. The relational interface has been implemented and the Codasyl interface is expected this year. The structure of PRECI is modular and flexible with an open-ended approach that allows easy incorporation of variations and upgrades.

A major feature of the model is run-time efficiency and the novel external hashing method, hash trees, was designed to support this. While providing direct access to tuples it enhances sequential access and access on secondary keys and non-key attributes by storing the data in fixed data locations each indicated by a surrogate. An important step in this hashing procedure is to compress the key space so that sparse key spaces do not cause excessive degradation of storage utilization, while preserving the key sequence of stored tuples, at the time of loading at least.

In this paper one approach to compressing the key space is described and the results of experiments carried out using relations obtained from collaborating establishments are presented. As expected these empirical results show that the advantages of this method depend on the applications and key distributions.

In the next section we survey some notable contributions to the collection of available external hashing methods and in Section 3 the hash trees method is outlined. Section 4 presents the key compression algorithms and in Section 5 the experimental results are given. The conclusions appear in Section 6.

2. EXTERNAL HASHING METHODS

A hashing function is defined as a transformation which maps an identifier attribute (key) to an address location for storage and retrieval of the key and its associated data (tuple). We refer throughout this paper to a general model for data storage and retrieval systems, described in Fig. 1 and by constraints 1 to 4 below.

In Fig. 1 the *key space* of a relation is a collection of elements where each element represents a possible key of the relation. The *initial key space* is the subset of the key space representing the range of key values existing in the relation as it stands at database load time. These terms are considered synonymous for the purpose of this paper. The collection of available secondary storage locations' addresses in which the tuples are to be physically stored constitutes the *address space* of a relation.

A scatter-table or *directory* is generally inserted between these two spaces, to provide a level of indirection and facilitate access. So when a logical slot number is indicated after hashing the key, the address space

CCC–0010–4620/82/0025–0486 $03.50

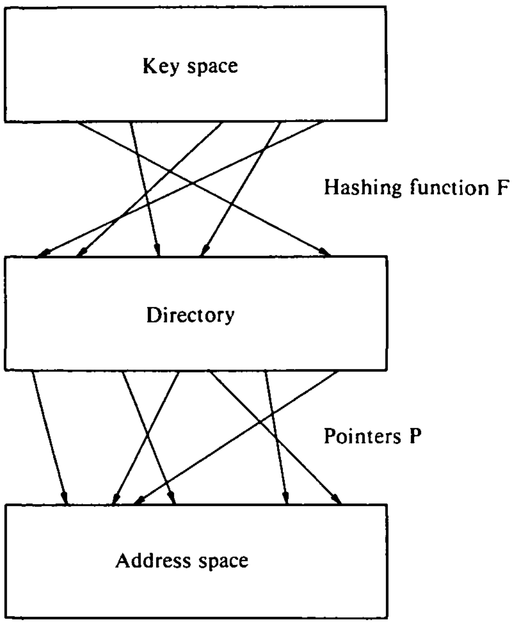

Figure 1. General model of external hashing schemes.

location is found by searching the directory, whose size dynamically reflects the address space size. Once this address is known only 1 access is needed to obtain the stored tuple's data.

In general existing external hashing methods are restricted by the following four constraints:

Constraint 1

The hashing function, F, in Fig. 1 has the negative effect of breaking down the natural sequence of the keys K_i in order to give a uniform distribution of tuples over the directory thus incurring performance penalties in sequential access. That is, they *do not* satisfy the condition:

$$K_1 < K_2 \text{ iff } F(K_1) < F(K_2) \qquad (1)$$

Constraint 2

The pointers, P, by which the directory slots D_i are associated with address space slots (leaf slots) breaks down the sequence of the directory slots, thus compounding the disadvantage due to constraint 1 for sequential access. That is, they *do not* satisfy the condition:

$$D_1 < D_2 \text{ iff } P(D_1) < P(D_2) \qquad (2)$$

Constraint 3

Many synonyms occur under F, and when the number of synonyms reaches a threshold, ω, the corresponding address space slot will not be large enough to hold all the tuples assigned to it, so an overflowing strategy must be invoked with associated access performance degradation. Normally this means that the tuple with key K is associated with a directory slot D where the following condition *does not* hold

$$F(K) = D \qquad (3)$$

Constraint 4

If access to the relation is required via a non-key or secondary key attribute NK, this is normally facilitated using a secondary index (NK_i, A_i) showing for the attribute NK_i its storage address space location, A_i. If this index is held in attribute sequence, sequential access to the tuples by the attribute is also expedited.

However, if the address of the tuple is changed for some reason, then an additional problem of secondary index maintenance is encountered. That is, if the new address of NK_i is A_i^1 the corresponding index entry (NK_i, A_i) *does not* satisfy the condition

$$A_i^1 = A_i \qquad (4)$$

External hashing schemes should aim to eliminate, or at least minimize the effect of, these constraints.

If we restrict our attention to the support of single-relation queries, the above general model is sufficient. However for multi-relation queries an extra dimension is encountered necessitating that the priority of the accessing functions and other load specification parameters must be included in the model. PRECI does provide facilities for this, but these are beyond the scope of this paper.

The external hashing methods described in this section all remove constraint 3 by using a splitting technique and dynamically altering the hashing function to avoid rehashing or drastic reorganization. However none of these methods remove constraint 4 because splitting involves reallocation of tuples to new addresses which implies secondary index reorganization. The directory in Fig. 1 allows the hashing function to remain unchanged. In general they are also constrained by conditions 1 and 2 in that they are not order-preserving, although there are some (partial) exceptions to this rule.

Dynamic hashing[4] allows graceful expansion and contraction as the number of stored tuples increases or decreases, thus eliminating overflow problems. Data can be retrieved in one access if the 'forest of trees' of the directory is held in primary storage. The storage utilization claimed for this method is poor at 69%. When a data set overflows the directory leaf slot pointing to it becomes an internal node pointing to two directory leaf slots. The right one points to a new data slot which is allocated some of the tuples, the left one pointing to the original slot in which the rest of the tuples remain. Shrinkage is catered for analogously. Scholl[3] has applied a method of deferred splitting to this basic method, thereby providing a smaller directory and higher storage utilization without too great a penalty in access time.

Extendible hashing[7] also accommodates overflows by splitting. The directory is collapsed into a single level, and is implemented using a 'buddy system' partition. For an address space of up to $2^x - 1$ slots, a directory of 2^x entries, not necessarily distinct, is required. When the depth x of the partition increases (i.e. a 2^xth data slot is required) the directory doubles in size. The x most significant bits of $F(K)$ give the directory slot's address. When a data slot overflows, its occupancy is halved and a new slot is assigned. Conversely if it underflows it is absorbed by its buddy if there is accommodation available. When the data slots are pages this method is attractive. Constraint 2 is removed and this method allows 'weak sequential access' in that updates may be applied efficiently in pseudo-key order.

Virtual hashing[5] does not use a directory and the hashing function changes dynamically for synonyms. Splits are implemented in a similar manner to that above. The sequence of keys is rather badly broken down and storage utilization is poor, but only 1 access is ever needed to get a tuple when its key is known.

3. THE HASH TREES METHOD

This method does not use a directory to provide a level of indirection between the key space and the address space, but it does use a compact directory, the Surrogate Directory (SD),[7] to keep track of available storage locations in data slots, and an index, PINDEX, to give access to tuples in primary key sequence. Other indices allow direct access to tuples by secondary keys or non-key attributes in an unusually efficient manner.

The method centres around the allocation to each tuple of a surrogate, composed of a relation number and an effective key which remains fixed for the lifetime of the tuple, or until reorganization. The effective key is determined by hashing and consultation of SD. The hashing function H allocates tuples to data slots of width ω, in a manner which ensures that all tuples in a data slot D^1 which has a lower address than slot D^{11} have a key value which is less than that of any tuple in D^{11}. The placement of tuples within the slot is carried out after consulting SD to locate the first empty position in the slot—so that on initial loading the keys are in sequence. Overflow slots of reduced width, usually $\omega/2$, are allocated dynamically and exclusively, either in local overflow slots (on the same data page) or in global overflow slots.

To minimize the spread of the address space the primary key space is compressed during the hashing procedure, and the primary aim of this paper is to show how this compression is effected and measure its success.

If a surrogate is not allocated or is deleted the corresponding position remains empty until another tuple is assigned to it via H and SD, but this is only allowed after a suitable integrity check.

The Surrogate Directory, SD, assumed here to be held in compact form in primary storage (as described in Ref. 7) contains an entry for every hash slot showing the current number of tuples C in the slot and a pointer to its overflow slot.

To insert a tuple the associated data slot is determined using H and SD is searched. If $C < \omega$, for this slot, the first empty tuple position in the home slot is allocated to this tuple. This happens even if there is an overflow chain associated with this slot, so that the sequence of keys is partially broken. If $C = \omega$ and there were no previous overflows, an overflow is assigned and the tuple is assigned to its first position. If there is an overflow slot already the first available position is allocated to the tuple.

Access in primary key sequence is effected by using PINDEX, a tree showing the surrogate for each primary key and maintained in primary key sequence. Direct and sequential access to secondary keys can also be supported by additional indices. Because the surrogate is fixed for a tuple's life time, no periodic reorganization of secondary indices is required (unlike other schemes where migration via splitting is allowed).

The initial loading of a relation is in primary key sequence and if it is possible to determine the likely tuple insertions at load time surrogates can be allocated for these. An important feature of the hash trees method is to ensure the sequentiality of the initial load, and this is the subject of later sections.

Table 1 indicates how this method compares with the other external hashing schemes.

Table 1. Taxonomy of external hashing schemes

Method	Condition 1	Condition 2	Condition 3	Condition 4
Extendible	×	✓ (weak)	✓	×
Virtual	×	×	✓	×
Dynamic	×	×	✓	×
Hash trees	✓		✓ (weak)	✓

4. THE PRECI KEY SPACE COLLAPSING METHOD

The PRECI hashing function, H, which provides fast direct access but preserves natural sequence, is based on an extremely simple principle. In the traditional division hashing algorithm the key sequence is broken down in the address space because the remainder is used as $F(K)$ rather than the quotient. Hash trees uses the quotient as $H(K)$, thus ensuring that the key sequence is preserved in that consecutive groups of keys will be assigned to consecutive address slots, so that condition 1 and condition 2 of the general model are satisfied.

However, although this division hashing function (using quotient as $H(K)$ has always been recognized as useful for high density key distributions, it is clearly not suitable for sparse key distributions owing to the large amounts of wasted, expensive address space locations.

The obvious solution to this problem is to use a collapsing technique to remove the empty (and possibly the underfilled) slots, in order to concertina the key space and/or address space. To keep track of these removed empty slots a storage overhead in the form of a table, T, is incurred to support the hashing function.

Each entry in T shows, for a particular value of $H(K)$ the cumulative number of empty slots, e_i, which precede it (see Fig. 2). Clearly not all values of $H(K)$ need have an entry in T; only those corresponding to occupied slots which follow a gap (a series of empty slots) need to be included. The true slot, Z, for a key K is found by seeking $H(K)$ in T and *subtracting* the total number of empty slots e_i preceding it. In this way a key-distribution such as that in Fig. 3(a) is altered to that of Fig. 3(b).

Because the hash divisor (h) may be much greater than ω (see step 3 below), some slots will be allocated more than ω tuples, i.e. $H(K)$ will give a particular value f_i more than ω times. Table T indicates the depth d_i of such overflow, and this allows the allocated tuples to be spread over the $d_i + 1$ slots starting at the true slot, Z. The value

	← Table T Entries		→	
Notation	f_i	d_i	e_i	z
Meaning of entry	Hashed slot number $H(K)$	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).

Figure 2. Composition of table T.

d_i must be subtracted from the e-value of the next T entry. In this way the distribution of Fig. 3(b) is altered to that of Fig. 3(c).

In this section the method is described, mainly in narrative form, and in the next section empirical results are presented to assess the performance of H on some typical relations encountered in collaborating establishments. The key space compression method is described as a number of steps, which may run in parallel to some extent, carried out as the 'initial relation', R, cardinality c, is loaded into physical storage.

Step 1

(a) Find all contenders for each character position of the key (length l characters). This results in a table $\{K_{i_j}\}$ $i = 1, \ldots, l; j = 1, \ldots, M$ where M is maximum number of contenders for any i.

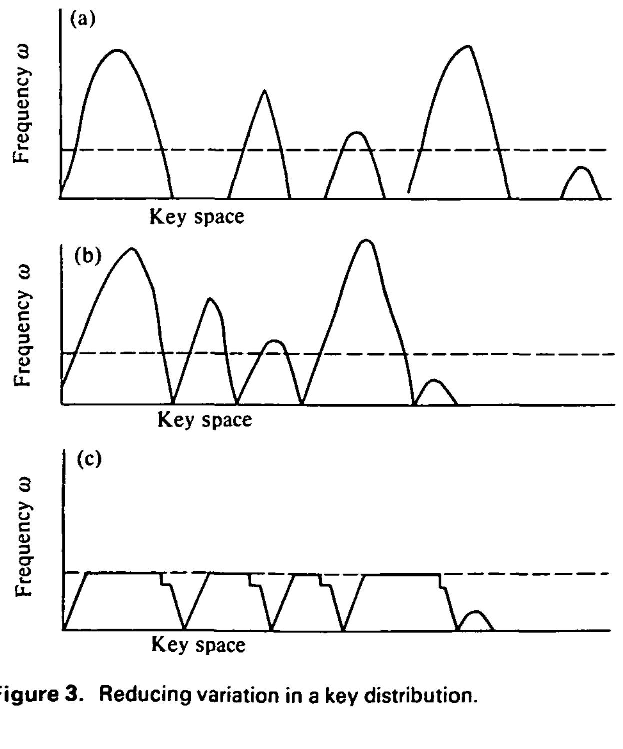

Figure 3. Reducing variation in a key distribution.

(b) Allocate to each character in $\{K_{i_j}\}$ an integer $\{A_{i_j}\}$ corresponding to the ranking of the $\{K_{i_j}\}$ in the computer's character order, for each i.

Note. Often (a) is given information. The DBA is often aware of the possibilities for each character position within the key at data analysis time. Otherwise a simple algorithm may be implemented to examine Q (the initially loaded relation) to determine $\{K_{i_j}\}$. $\{A_{i_j}\}$ is very readily obtained from $\{K_{i_j}\}$.

Step 2

Convert each key to a decimal number X_k. For $k = 1, \ldots, c$ and $i = 1, \ldots, l$ (c is cardinality of relation)

(i) determine the integer value from $\{A_{i_j}\}$ $j = 1, \ldots, M$

(ii) convert this to a decimal number b_i by changing the radix of each character position i^1, $i^1 > i$

(iii) add it to X_k.

At the end of this procedure X_k is the decimal equivalent of key. When all keys are in decimal the key space will already be partially collapsed—especially if it was originally very sparse.

Step 3

Determine h, the hash divisor for H. Assume that the database designer can state a data slot size, ω tuples per slot. Let r be the range of the X_k, $k = 1, \ldots, c$ values determined in step 2. i.e. $r = X_c - X_1 + 1$, then $h = [(r/c)\omega]$. E.g. consider a relation with 500 tuples in the range 00001 → 10000. Assume slot size = 20, then $r = 10000$, $c = 500$, $\omega = 20$. Therefore $h = 1$.

Get a first estimate, f, of each tuple's directory slot. This is taken to be the quotient obtained when X_k is divided by h

$$f = \left\lfloor \frac{X_k}{h} \right\rfloor$$

Step 4

Scan the key space to model the distribution of empty or 'overflowing' slots in table T.

This step involves producing a table with elements $\{f_i, d_i, e_i\}$, to indicate the cumulative number of empty slots e_i preceding slots with addresses greater than or equal to f_i, and the number of slots d_i needed to accommodate the tuples assigned to f_i (see Fig. 2).

T is expected to be small enough to be held permanently in primary storage. This is supported by the empirical evidence in Table 2.

If the value as calculated obeys the condition $f_i \leq f \leq f_{i+1}$ in T then subtract e_i from f_i to get the slot address for f.

By including the d_i column in T its size is reduced. If ω is much smaller than h, many slots may be allocated more than ω entries in step 4. These 'overflowing' slots are indicated by the d_i value in each T entry. The tuples assigned to f's slot, say slot V, are actually accommodated consecutively, in sequence initially, in data slots V,

$V + 1, \ldots, V + d_i$. If d_i is greater than 0 the resolution of this access technique is not sufficient to provide the address of the single slot to which a tuple belongs, but indicates a $(d_i + 1)$ slot area to which it belongs. A further sequence-preserving hashing algorithm must then be invoked to get a single slot address.

At the present stage of development this secondary hashing algorithm is a simple statistical hashing method, and for the purposes of this study a uniform distribution of keys assigned to the $(d_i + 1)$ slot area is assumed, as this was justified by sampling the observed distributions. This allows the displacement of the tuple's home slot within the slot range to be easily calculated. If there are occasional breaks in the uniformity of the distribution of keys these can be simply incorporated in an extended version of T.

The main contribution of this paper comes from an empirical study of the effect of H upon the primary performance indices, response time and storage utilization. In the next section we present the results of an experiment measuring the size of T and the success of the compression algorithm.

5. RESULTS

An experiment was set up to study the characteristics and performance of this collapsing technique on three relations obtained from databases used in collaborating establishments. The relations described the following entities:

Relation 1 (Cardinality 24848, density (no. of keys/key range) 1/4, key length 6). PARTS used in a manufacturing company.

Relation 2 (Cardinality 20209, density 1/48408, key length 9). BOOKS stored in a university library.

Relation 3 (Cardinality 7390, density 1/250, key length 7). STUDENTS enrolled in a Polytechnic.

All the primary keys—part number, ISBN number (less check digit) and student number—were numeric. In fact all were decimal, which simplified the hashing method as steps 1 and 2, which are straightforward and of little interest in this study, were eliminated.

Since the initial assignment of tuples to slots clearly preserves the key sequence, the primary characteristics of interest in the experiment were the effect on storage utilization and on the response time.

We ran the hashing and table-generation algorithms, which are implemented on ICL 1900 and VAX systems, for three values of the slot size ω, namely 5, 10 and 20 tuples. The percentage of wasted space in the secondary storage devices was calculated for each relation for each slot size, by determining relationships of the total space required to store the tuples with the number of tuple positions actually occupied in each case ($=C$). The variation in slot size can be viewed as a reflection of variations in either the tuple or page sizes, thus assuming that the length of the tuples is fixed, or that the stored tuple size, the slot and page sizes remain fixed.

Table 2. Experimental results

Relation	Slot size (tuples)	% Waste in home slots	T size Entries	T size Bytes	Max. d_i in T	Average d_i in T
1	5	17.1	1806	5.5K	3	1.27
	10	20.4	920	2.5K	3	1.05
	20	23.3	570	1.5K	3	0.98
2	5	4.9	376	1K	183	15.60
	10	7.4	186	550	103	11.36
	20	9.6	137	411	98	9.17
3	5	6.6	165	495	22	8.74
	10	8.6	92	286	22	7.63
	20	11.6	47	141	22	6.52

5.1 Storage utilization

To apply the simple (division-quotient) hashing algorithm H of this method appears at first sight to invoke a penalty of poor storage utilization because of wasted space corresponding to sparse areas of the key space. The results of storage utilization measurement are encouraging as can be seen in column 3 of Table 2. They show that the hashing function H is successful in compressing the key space while maintaining the sequence of the primary keys. The most striking result here is the remarkable performance on the extremely sparse ISBN relation, Relation 2. The maximum waste percentage is 9.6 which must be considered excellent for a key space in which only 1 in 48408 keys are actually used. If the division algorithm, using the quotient upon division by ω as the slot address to preserve the sequence, were used, without using Table T and the associated spreading of clusters and elimination of unused slots, the percentage waste for this relation would be about 500000!

The performance of H on Relation 3 was also very impressive. This key space was characterized by a small number of fairly sizeable gaps in the key distribution occurring, for example, between the last student number in one faculty and the first number in the next. The wasted space here was extremely low for a distribution which is not so sparse as Relation 2, but presents a challenge when attempting to randomize it uniformly and also maintain the key sequence. The wasted space in Relation 1 was higher than expected for a relation whose key space was originally more densely and uniformly occupied than either of the other two relations. The gaps in the key sequence were shorter and more frequent than in the other cases. However, after compression there was a much greater assignment of tuples to their home slots than in either of the other cases, so there were many more partially full home slots in this relation.

The proportion of non-full slots in the three relations was approximately

Relation 1 : Relation 2 : Relation 3 :: 3 : 13 : 8

The implication is that dramatically non-uniform key distributions, characterized by large gaps and very dense clusters, give rise to better storage utilization under H than less dramatically non-uniform distributions.

5.2 Time and space overhead due to table T

The cost of the good compression of the key space is that an image of the initial key distribution must be stored in

T. Therefore it is of interest to determine the size of T for each slot size and each relation so as to gain insights into this overhead, both in terms of space requirements and degradation of response time. The results of these measurements are presented in columns 4 and 5 of Table 2.

The entries in T can be compressed in order to minimize its negative impact on performance. With minimal packing of entries we found that in the worst cases the maximum space required to hold T was about 5½K bytes for Relation 1, 1K bytes for Relation 2 and reduced to about 141 bytes for Relation 3.

The results indicate that large slots are best for response time given realistic buffer page sizes. This bears out the results found by simulation in Ref. 7. If T can be accommodated in primary storage on a fairly permanent basis which is possible and justified in some enquiry systems, then access time will not be affected, as the table look-up using up to 11 comparisons in a binary search procedure, will cause little degradation. However, it is likely that, in a multi-relation, multi-user database environment, T would have to be stored externally so that an extra probe would be required. If the page size of the system were less than required to hold T, even more probing would be needed with corresponding access degradation. However, the effectiveness of this method for the relations considered in this study, although dependent on the record and page sizes, was not adversely affected by T size.

5.3 Allocation of tuples to hash slots

Another measure is important in assessing the usefulness of H, namely the depth of overflows, d_i in T. If this is large frequently, then the response time may be adversely affected. As may be seen from the last two columns of Table 2 a worst-case depth of 183 was encountered for relation 2. This value is a measure of the clustering of values in the key space and means that enough tuples to fill 184 slots are assigned by H to a single slot. This would be a very alarming result if it meant that up to 184 slots would have to be scanned to find a particular tuple's address—a totally unacceptable searching exercise. However the algorithm for H takes this into account. As indicated in Section 4, a secondary hashing function H^1 is invoked in these circumstances to find the unique home slot for the tuple (for example from the 184 slots storing tuples with one $H(k)$ value). The hashing function H^1 must give a dense sequential distribution and so the list of contenders is short. The data used in this study was fortuitously very uniformly distributed across the assigned slots, and only a very simple *displacement function* was required to locate a tuple's home slot or at worst its immediate neighbour. Exceptions were detected in Relation 2 where the uniform distribution was found to be interspersed with some large gaps. These were dealt with efficiently by incorporating the descriptions in table T, which increased T for Relation 2 by about 20%.

It is interesting to evaluate these results briefly against the familiar tree-based access methods. In the indexed-sequential access methodology for example the tuples stored in each bucket are indicated by an entry in an index, and higher levels of index such as a seek area index are used to enhance efficiency. The results for the hash trees method as given above show improvements in access times (e.g. for retrieval of a stored tuple by key) of several slot (or bucket) accesses, because hashing followed by look-up of the small table T is used rather than navigation through a tree. In addition the size of T taken along with the Surrogate Directory and PINDEX overheads is much less than for the best of the familiar tree-based packages available.

This improvement is at the cost of embedded waste in the address space, which is usually (at least partially) absorbed eventually, and of more complex algorithms for searching and update.

6. CONCLUSIONS

It is clear from the results that, as in the case of the traditional hashing methods described by Lum *et al.*,[6] the hash trees external hashing method gives performance which varies depending on the initial key distribution. It is suitable for applications requiring sequential access for report generation or updating operations. The address space utilization of the algorithm was pleasingly high in the test cases, the tuples are dispersed in a uniform manner over the address space, and if Table T can be accommodated in primary storage, direct access to a tuple's data slot, or in a small minority of cases, an immediately neighbouring slot, in 1 access.

However, T was found to be of non-trivial size for some key distributions, notably one with a slot occupancy distribution which alternates rapidly between empty and fairly mildly overflowing levels. To use a small slot width in such circumstances incurs a moderate penalty in terms of secondary storage device space but a heavy penalty in access time if the page size is much smaller than is needed to accommodate T. There is however a negative correlation between T size and storage wastage, so that care must be taken in selecting the hash width ω, as both measures vary considerably with ω. For the relations considered it is clear that if direct access is of greater importance than storage utilization then slot widths should be kept high in order to limit overflow depth.

This method is therefore a useful addition to the toolkit of the database designer who must deal with a variety of access profiles and key distributions. It is particularly suitable for distributions which are predominantly uniform but have a significant number of substantial gaps. Such distributions are commonly encountered in practice. For example, if a range of employee numbers are allocated to each department, as these are normally allocated to employees on a serial basis there are generally sizeable gaps between the last employee number allocated to one department and the first employee number allocated to the next. Another example of such a distribution is where a range of part numbers is allocated to each project, and some of each range remains unused.

Extensions and improvements of the method are being researched in several areas. The size of T, and hence the access time can be reduced by considering underflowing slots, for example, those with occupancy less than 20% of ω, to be empty and allocating tuples assigned to such a slot to its immediate predecessor. Candidates for the secondary hashing function are being investigated, so that dense, sequence-maintaining allocations can be ensured for non-uniform distributions irrespective of the distribution of the key over the group of slots. Finally the applicability of H as the preliminary hashing function F in other external hashing methods, in particular extendible hashing, is being investigated.

Acknowledgements

The authors wish to thank Professor P. Stocker of the University of East Anglia for useful suggestions and comments on this study, which was partially funded by the Science and Engineering Research Council.

REFERENCES

1. S. M. Deen, D. Nikodem and A. Vashista, The design of a canonical database (PRECI). *Computer Journal* **24**, 200–209 (1981).
2. P. A. Larson, Dynamic hashing. *BIT* **18**, 184–201 (1978).
3. M. Scholl, New file organisations based on dynamic hashing. *ACM TODS* **6**, 194–211 (1981).
4. R. Fagin, J. Nievergelt, N. Pippenger and H. R. Strong, Extendible hashing—a fast access method for dynamic files. *ACM TODS* **4**, 315–344 (1979).
5. W. Litwin, Virtual hashing—a dynamically changing hashing. *Proceedings of the VLDB Conference*, Berlin, 517–573 (1978).
6. V. Y. Lum, P. S. T. Yuen and M. Dodd, Key to address transform techniques. *Communications of the ACM* **14**, 228–239 (1971).
7. S. M. Deen, An implementation of impure surrogates. Proceedings of 1982 Conference on VLDB, Mexico (1982).

Received May 1982

Book Reviews

Continued from p. 494

E. W. MARTIN AND W. C. PERKINS
FORTRAN for Business Students: A Programmed Instruction Approach
Wiley, Chichester, 1981. 811 pp. £12.15.

This book suffers from serious defects both in the material presented and in the manner of presentation.

The pedagogic approach is very stultifying. The book is *not* a programmed instruction text, despite the title. Instead, a sequence of small frames is presented, each containing some text and a question to be answered. However, if a wrong answer is given, the student is not directed to remedial material, but merely reconsiders why he gave that answer (and by now, of course, he has been shown the correct one). The level of the questions is of such utter triviality that the student is defended from boredom only by a mounting indignation against the insult to his intelligence.

The subject matter itself is not convincing. Leaving aside any consideration of the merits of ANSI FORTRAN IV in general, there is little case for its use as a first language for business students. The authors' discussion of files is, in consequence of their use of FORTRAN, inadequate and long deferred, central though the topic is to commercial data processing.

The technology described is out of date, both hardware and software (perhaps because such a massive tome took a decade to write!). It is assumed that the usual computing environment consists of access via punched cards to a batch-processing mainframe, which rather undermines the authors' claim that the book will serve as a text for any student with access to FORTRAN (further subverted by the occasional non-standardness of the FORTRAN they use). There is no attempt to encourage the student to think in terms of the constructs of structured programming, and that most valuable of structural tools, the subprogram, is not introduced until p. 535, where it is allocated only 35 pages.

In summary, this monumental volume is a poor realization of a misguided project.

C. D. F. MILLER
Leeds

R. E. BERRY
Programming Language Translation
Ellis Horwood, Chichester, 1982. 175 pp. £15.00, £6.50 paper.

This is a pleasant little book, which I enjoyed reading. Its title might mislead people into thinking that it is about translating from one high-level language to another, but in fact it is about translating from high-level to low-level, i.e. compiling.

The first six chapters (lexical analysis, syntax definition and syntax analysis, symbol tables—structure and access, the run time environment, semantic processing, run time support) are concerned mainly with the high-level side of the fence; the next three (assemblers, macros, loaders) more with the low-level side. The final two (Pascal S compiler, Pascal S interpreter) are the real meat of the book, describing a compiler in detail with a complete listing of it, written in Pascal (but not in the Pascal S subset which it compiles).

Considerable prior knowledge of both computing in general, and Pascal in particular, are taken for granted, and detailed knowledge of the Pascal S compiler is also assumed before you reach it, particularly in the exercises at the end of each of the first eight chapters. For this reason it is rather a difficult book to read, as the King of Hearts' algorithm ('Begin at the beginning, and go on till you come to the end: then stop') certainly will not do, but no other order is suggested. Probably the only answer is to read it several times.

The Pascal S syntax is given in diagrammatic form, but with too many errors in the diagrams. Other misprints are not too bad: ('relativizer' is a nasty enough word without misprinting it 'relavitizer' though). The printing is unfortunate in that the typefaces of both the main text and the Pascal listings have virtually identical renderings of letter l and figure 1. In some places, I was actually misled by this and it is nasty even where not misleading.

Although there are references throughout the text to a bibliography, I came to the conclusion during my reading that the bibliography itself has been accidentally omitted. I finally came across it, nestling at the end of Chapter 9, but there is no indication anywhere else of where or how to find it.

Why is it that Pacal books so often give reserved words underlining, as here, or boldface, as if it were Algol? There are those, including myself, who believe that a language ought to distinguish such words from identifiers, and those who believe it to be disadvantageous. Whoever is right about this, the fact is that Pascal does not do it, and it is misleading (or unfair to Pascal) to pretend that it has this advantage (or disadvantage) when it does not.

I. D. HILL
Harrow

MICHAEL SHAVE
Data Structures
McGraw-Hill, New York, 1981.

This book serves as a useful self-teaching text introducing a range of concepts. The range brings in on the one hand techniques which may be unfamiliar to a computer scientist and conversely some explanations are approached from the point of view of a computer scientist and may therefore present a somewhat different but interesting aspect of the work to those who are working in a professional computer applications role.

The author sets out the theory underlying a number of algorithms to cover functions. From these functions the choice would have to be made in order to set up a system to handle data, data structure or conventional files. In principle the data may be numerical or text or mixed.

The author covers storage structures including sequential, linked, binary trees, also allocation of space to linked structures, dynamic block allocation, methods of garbage collection and operations on tree structures.

A companion text which leads on to an assessment of the deployment of such algorithms with the various strands of database technology or alternative application systems would be both useful and interesting.

M. M. BARRITT
Edinburgh

Continued on p. 422