MC SYLLABUS 46.2

COLLOQUIUM DATABANKORGANISATIE

DEEL 2

P.M.G. APERS (red.)

MATHEMATISCH CENTRUM AMSTERDAM 1981

1980 Mathematics subject classification: 68-06, 68A05, 68B20, 68H05

ACM-Computing Reviews-category: 4.33

ISBN 90 6196 231 5

INHOUD

x

IX. ONTWERPASPECTEN VAN DISTRIBUTED DATABASE MANAGEMENT SYSTEMS door R.A.C. THOMAS

VOORWOORD

De onderhavige syllabus vormt het tweede deel van het colloquium databank-organisatie. Een achttal sprekers hielden een lezing.

In de twee bijdragen van Litwin wordt het gebruik van hashingtechnieken om gegevens op extern geheugen te benaderen, onderzocht. Zowel de loadfactor als het aantal diskaccesses worden bekeken.

Remmen beschouwt in zijn bijdrage een optimale logische gegevensstructuur voor een informatiesysteem. Onder optimaal wordt verstaan een minimaal aantal relaties. Onder bepaalde voorwaarden blijkt zo'n structuur uniek te zijn.

Nijssen presenteert in zijn bijdrage een kader voor informatiesystemen. Het doel is programma-efficiëntie, correctheid van informatiebank en extern-intern transformaties te isoleren, zodat ze als onafhankelijke problemen kunnen worden opgelost.

In de bijdrage van Engmann wordt het ontwerp en de implementatie van ERIS, een Experimenteel Relationeel Informatie-Systeem, besproken. De taal SIMULA 67 is vanwege zijn structureringsfaciliteiten gebruikt voor de implementatie.

Het relationele database-management systeem VIDEBAS wordt behandeld in de bijdrage van Blanken. De architectuur is gebaseerd op een goedkoop extern geheugen en een grote rekencapaciteit.

Apers behandelt in zijn bijdrage het probleem van het zo optimaal mogelijk plaatsen in een netwerk van de data van een gespreide database. Twee kostenfuncties worden beschouwd: het totale netwerkverkeer en de gemiddelde responstijd.

De bijdrage van Kersten geeft een beschrijving van de interactieve database-faciliteiten van PLAIN (Programming LAnguage for INteraction). De realiseerbaarheid van toegangsprotectie in PLAIN wordt onderzocht.

Thomas geeft in zijn bijdrage een classificatie van gespreide database-systemen. Een paar specifieke distributie-aspecten worden aan de hand van D-INGRES, een gespreide versie van INGRES, behandeld.

Dit colloquium, dat in samenwerking met het Mathematisch Centrum werd georganiseerd, stond onder leiding van drs. P.M.G. Apers (Vrije Universiteit), prof. J.M. van Oorschot (PTT en Vrije Universiteit), prof.dr. J.A. van der Pool (IBM en TH Twente), drs. F. Remmen (TH Eindhoven) en prof.dr. R.P. van de Riet (Vrije Universiteit).

Wij danken het Mathematisch Centrum, en met name dr. J.C. van Vliet, voor de goede samenwerking.

P.M.G. Apers.

LINEAR HASHING: A NEW TOOL FOR FILE AND TABLE ADDRESSING*

W. LITWIN
I.N.R.I.A.
78150 Le Chesnay, France

0. ABSTRACT

Linear hashing is a hashing in which the address space may grow or shrink dynamically. A file or a table may then support *any* number of insertions or deletions without access or memory load performance deterioration. A record in the file is, in general, found in one access, while the load may stay practically constant up to 90%. A record in a table is found in a mean of 1.7 accesses, while the load is constantly 80%. No other algorithms attaining such a performance are known.

1. INTRODUCTION

The most fundamental data structures are files and tables of records identified by a primary key. Hashing and trees (B-tree, binary tree,..) are the basic addressing techniques for those files and tables, thousands of publications dealed with this subject. If a file or a table is almost static, hashing allows a record to be found in general in one access. A tree always requires several accesses. However, when the file or the table is, as usually, dynamic, then a tree still works reasonably well, while the performance of hashing may become very bad. It may even become necessary to rehash all records into a new file.

We have shown, however, in LITWIN [14] that hashing may be a tool for dynamic files, if the hashing function is dynamically modified in the course of insertions or deletions. We have called this new type of hashing *virtual hashing* (VH), in contrast to the well known hashing with a static function,

* Republished, with permission of IEEE, from Proceedings, Sixth International Conference on Very Large Data Bases, Oct. 1-3, 1980, Montreal, Canada.

which we will refer to as *classical*. Through an algorithm called VH0 (LITWIN [14]), we have shown that a record in a dynamic file may typically be found in two accesses, while the load stays close to 70% (LITWIN [15]. Another algorithm, called VH1 (LITWIN [16,17]), has shown that a record may even be found typically in one access, while the load during insertions oscillates between 45% and 90% and is 67.5% on average. It showed also that the average load during insertions may be always greater than 63% and almost always greater than 85%, if we accept that the average successful search requires 1.6 accesses (LITWIN [19]); Finally, a generalisation of VH1, called VH2, has shown that for a similar load, the average successful search requires very close to one access (LITWIN [19]). Two other algorithms similar to VH0 have been proposed, Dynamic Hashing (DH) (LARSON [12]) and Extendible Hashing (EH) (FAGIN, NIEVERGELT, PIPPENGER & STRONG [5]). Since trees typically lead to more than 3 or 4 accesses per search and to a load close to 70% (KNUTH [11], COMER [4]), all these VH algorithms offer better access performance for similar or higher load factors.

VH0, DH and EH require at least two accesses per search because the data structure which represents the dynamically created hashing functions must be on the disk. VH1 and VH2 are faster, because the functions are represented by a bit table which, depending on file parameters, needs 1 kbyte of core (main storage), per file for 7000 to more that 1500000 records. In this paper, we present the algorithm which goes further, only a few bytes of core suffice now for a file of any size. For any number of insertions or deletions, the load of a file may therefore be high and a record may be found, in general, in one access. No other algorithms attaining such a performance are known.

The algorithm is called Linear Virtual Hashing or *Linear Hashing* in short (LH). The choice of file parameters may lead to a mean number of accesses per successful search not greater than 1.03, while the load stays close to 60%. It may also lead to a load staying equal to 90% while the successful search requires 1.35 accesses in the average. Even if the buffer in core may contain only one record, a search in the file needs 1.7 access in the average while the load remains at 80%. This property makes LH probably the best performing tool for dynamic tables as well.

The next section describes the principles of LH. We first show the basic schema for hashing. We then discuss the computing of the physical addresses of buckets, when the storage for them is allocated in a non-

contiguous manner. Finally, we present some variants of the basic schema.

Section 3 shows performance of the Linear Hashing. First, we show access and memoryload performance of the basic schema. Next, the performances are analysed for a variant with a socalled load control.

Section 4 concludes the paper. We sum up the advantages which Linear Hashing brings, we show some application areas and, finally, we indicate directions for further research.

2. PRINCIPLES OF THE LINEAR HASHING

2.1. Basic schema

We recall that hashing is a technique which addresses records provided with an identifier called *primary key*, or, simply, key. The key, let it be c, is usually a non-negative integer, and, in a work on the addressing by primary key, we may disregard the rest of the record. A simple pseudo-random function, let it be h, called a *hashing function*, assigns to c the memory cell identified by the value h(c). The *hashing by division* c/ → c mod M; M = 2,3,.. ; is an example of a hashing function. The cells are called *buckets* and may contain b records, b = 1,2,.. . The record is inserted into the bucket h(c), called *primary* for c, unless the bucket is already full. The search for c always starts with the access to the bucket h(c).

If the bucket is full when c should be stored, we speak about a *collision*. An algorithm called *collision resolution method* (CRM) is then applied which, typically, stores c in a bucket m such that m ≠ h(c). c then becomes an *overflow record* and the bucket m is called overflow bucket for c. If (i) overflow buckets are not primary for any c, (ii) each of them is devoted to only one h(c) and (iii) a new overflow bucket for an h(c) is chained to the existing ones, then we have a *bucket chaining* CRM. If, in particular, the capacity b' of an overflow bucket is b' = 1, we call this CRM *separate chaining* (KNUTH [11]).

A search for an overflow record requires for at least two accesses. If all collisions are resolved only by overflow record creations, as it was assumed until recently LITWIN [14], then access performance must rapidly deteriorate when primary buckets become full. If the insertion of c leads to a collision and no records already stored in the bucket h(c) should become overflow records, then c may be stored in its primary bucket only if a new hashing function is chosen. The new function, let it be h', should

assign new addresses to some of the records hashed with h on h(c) and the file should be reorganized in consequence. If h = h' for all other records, the regorganizing needs to move only a few records and so may be performed dynamically. The new function is then called by us *dynamically created hashing function* or, shortly, a *dynamic hashing function*. The modification to the hashing function and to the file is called a *split*, h(c) is, under the circumstances, *split address*. The idea in VH in general and so, in particular, in LH is to use splits in order to avoid the accumulation of overflow records. Splits are typically performed during some insertions. All splits result from the application of *split functions*. For LH, as well as for VH0 and for VH1, the basic split functions are defined as follows (LITWIN [18, 20]):

Let C be the key space. Let $h_0 : C \to \{0,1,..,N-1\}$ be the function that is used to load the file. The functions $h_1,h_2,..,h_i,..$ are called split functions for h_0 if they obey the following requirements:

(1)

$$h_i : C \to \{0,1,..,2^iN-1\}.$$

(2)

For any c either:

(2.1) $$h_i(c) = h_{i-1}(c)$$

or:

(2.2) $$h_i(c) = h_{i-1}(c) + 2^{i-1}N.$$

We assume that, typically, each h_i; i = 0,1,.. ; hashes randomly. This means that the probability that c is mapped by h_i to a given address is $1/2^iN$. This also means that (2.1) and (2.2) are equiprobable events.

Fig. 1 illustrates the use of split functions. The file is created with h0 : c → c mod N, where N = 100. The bucket capacity is b = 5 records. For split functions we choose the hashing by division, namely we put:

$$h_i : c \not\to c \bmod 2^iN.$$

This choice respects (2) since, obviously, for any non-negative integers k, L, either:

$$k \bmod 2L = k \bmod L$$

or:

$$k \bmod 2L = k \bmod L + L.$$

We assume that a collision occurs during the insertion of c = 4900. Instead of simply storing c as an overflow record, we change h_0 to the following h:

$$h(c) = h_1(c) \quad \text{if } h_0(c) = 0$$
$$h(c) = h_0(c) \quad \text{otherwise.}$$

We then reorganize the file. We thus have applied h_1 as the split function and we have performed the split for the address 0. The hashing function h results from the split and is a dynamic hashing function.

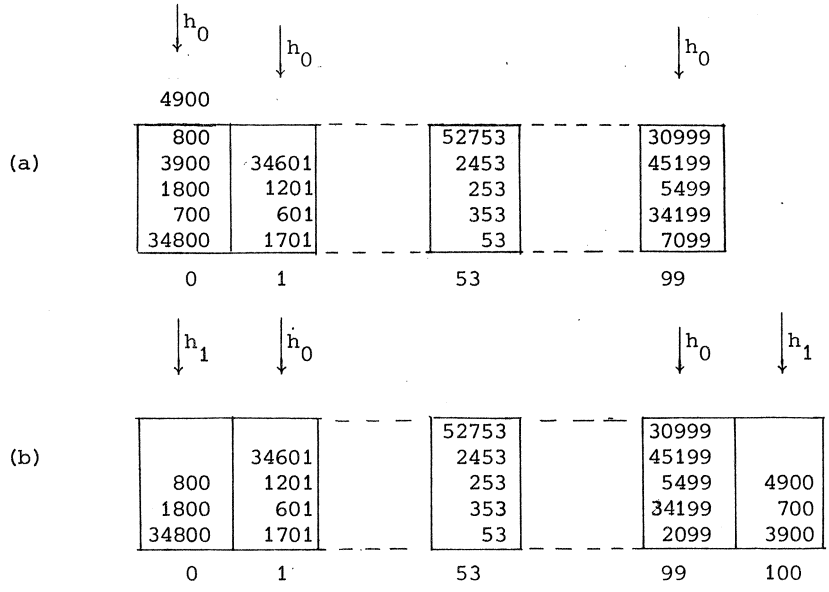


Fig. 1. The use of a split for a collision resolution.
(a) - a collision occurs for the bucket 0.
(b) - the collision is resolved without creating an overflow record and the address space is extended.

Fig. 1.b shows the new state of the file. Since $h = h_0$ for each address except 0, no records other than these hashed to 0 have been moved. Since $h = h_1$ for the records which h_0 hashed to the address 0, for approximatively half the number of these records the address has been changed. It followed from (2) that all these records had the same new address which, under the

circumstances, had to be 100. A new bucket has therefore been appended to the last bucket of the file to which all the records have been moved *in one access*. Since for all these records the bucket 100 is henceforward the primary bucket, they are all accessible in *one* access. In particular, this is also the case of the new record, i.e. 4900. On the other hand, the records which remained in the bucket 0 continue to be accessible in one access. In contrast to what could be done if a classical hashing was used, the split has resolved the collision *without creating an overflow record* and *without access performance deterioration.*

Let us now assume that the file addressed with h0 undergoes a sequence of insertions which did not yet lead to overflow records. Furthermore, let us assume that a split is performed iff a collision occurs. The natural idea would be to split the bucket which undergoes the collision, this was implicit for all algorithms for VH. However, split addresses must then be random and this must lead to dynamic hashing functions using tables. Dynamic hashing functions which do not need tables may be obtained only if the split addresses are chosen in a predefined order. To perform splits in some predefined order, instead of split the bucket which undergoes the collision, is the main new idea in the linear hashing.

Let m be the address of a collision. Let n be the address of a split to be performed in the course of the resolution of this collision. Since the values of m are random while these of n are predefined, usually $n \neq m$. If so, we assume that the new record is stored as an overflow record from the bucket m through a classical CRM, bucket chaining for instance. Next, we assume that n is given by a pointer which thus indicate the bucket to be split. For the first N collisions, the buckets are pointed in the linear order 0,1,2,..,N-1 and all splits use h_1. (2.2) implies then, that the file becomes progressively larger, including one after another the buckets N+1, N+2,...,2N-1. A record to be inserted undergoes a split usually not when it leads to the collision, but with some delay. The delay corresponds to the number of buckets which has to be pointed while the pointer travels up, from the address indicated in the moment of the collision, to the address of this collision.

With this mechanics, no matter what is the address, let it be m1, of the first collision, LH performs the first split using h_1 and for the address 0 are then randomly distributed between the bucket 0 and a new bucket N, while, unless $m_1 = 0$, an overflow record is created for the bucket m_1.

The second collision, no matter what is its address, let us say m_2, leads to an analogous result, except that, first, it splits for the address 1 and appends the bucket N+1. Next, it may constitute the delayed split for the first collision, suppressing therefore the corresponding overflow record. This process continues for each of the N first collisions, moving thus the pointer, step by step, up to the bucket N-1. Sooner or later, the pointer points to each m and the splits, despite of being delayed, move thus most of the overflow records to the primary buckets. We may therefore reasonably expect that, for any $n < N$, only a few overflow records exist.

After N collisions, we have $h = h_1$. (1) implies then that, instead of the hashing on N addresses, we now hash on 2N addresses. (2) implies that h_2 has on the hashing with h_1, the action analogous to that of h_1 on the hashing with $h = h_0$, except that it hashes on 4N addresses. We therefore assume that $n = 0$ again, that now we split with h_2 and that the upper bound on n is now 2N-1. For further insertions, we use $h_3, h_4, .., h_j, ..$ while the pointer travels each time from 0 to $2^{j-1}N$.

It results from the above principles that, first, the address space increases *linearly* and is as large as needed. Next, for any number of insertions, most of overflow records is moved to the primary buckets by the delayed splits. On one hand, we may thus reasonably expect that the rate of overflow records remains always small. If $b \gg 1$, the rate should even be neglectably small. Thus, we may expect the linear hashing to find a record usually in one access to the bucket, no matter how few buckets were provided when the file was created and how high the number of insertions finally is.

The highest index of a split function currently used, let it be j; $j = 0,1,..$; is called *file level*. If $n = 0$, we always have $h = h_j$ for some j. Otherwise, first, $h = h_{j-1}$ for the buckets not yet split with h_j, i.e. n, $n+1,...,2^{j-1}N$. Next $h = h_j$ for all the others. The algorithm computing the primary address of a c is therefore trivial:

```
(A1)
     if n = 0 : m ← h_j(c)
     else
          m ← h_{j-1}(c)
          if m < n  : m ← h_j(c) endif
     endif
endA1
```

2.2. Physical address computing

The address given by hashing must be transformed into the physical address of the bucket in the memory. The memory of files is usually divided into quanta of let it be q buckets; q = 1,2,.. . Quanta may be all of the same size or different sizes may be available. It then may be particularly worthwile to use sizes which are 2^i of a certain minimal q (buddy system KNUTH [11].

When the file is loaded some quanta are statically allocated. Then, if a file increases dynamically, quanta are sometimes added. If all quanta for a file are contiguous, then the physical address of a bucket m is as follows:

$$m'(m) = m'(0) + dm \tag{3}$$

where d is the number of memory elements, i.e., bytes, words or sectors,.. per bucket. Thus the advantage of a contiguous allocation is that only the address of the first quantum is needed and that the computing of the physical address is trivial.

However, if several dynamic files should share a memory, it may be better to allocate non-contiguous quanta. For each file, the addresses of these quanta must then be collected. The address of the i-th quantum may be the value T(i) of a table T. For the quanta of a fixed size, we then have the following formula:

$$\begin{aligned} i(m) &= \mathrm{INT}(m/q) \\ m'(m) &= T(i(m)) + d(m - i(m)q) \end{aligned} \tag{4}$$

where INT denotes the integer part. In particular, if q = 1, i.e., if the allocation is totally distributed, then we have simply:

$$m'(m) = T(m). \tag{5}$$

For the quanta of different sizes, we particularly recommend the following schema:

- let K be a parameter: K = 1,2,.. . The sizes $q_0, q_1, \ldots$ of successive quanta of the file should be:

$$
\begin{aligned}
& q_0 = N \\
& q_1 = q_2 = \ldots = q_k = q_0/K \\
& \ldots \\
& q_{lk+1} = q_{lk+2} = \ldots = q_{(l+1)k} = 2^l q_1
\end{aligned}
\tag{6}
$$

where l = 0,1,.. . For instance, if N = 10 and K = 4, then the sizes of the quanta dynamically allocated are 5, 5, 5, 5, 10, 10, 10, 10, 20, 20,.. . Dynamic allocations take thus place when LH starts to use the addresses 20, 25, 30, 35, 40, 50, 60, 70, 80, 100,.. .Higher is the value of K, smaller is the drop in memory load when a new quantum is allocated, but T is larger. The practical values of K are between 1 and 10.

Let m_i be the smallest logical address in the i-th quantum. Next, let it be:

$$m'' = m - m_{i(m)}.$$

Therefore we have:

$$m'(m) = T(i(m)) + dm''. \tag{7}$$

In the case of (6), i(m) and m'' may then be computed by the following obvious algorithm:

```
(A2)
    if m < N : i ← 0 ; m'' ← m
    else
         i ← j-1; M ← 2^i N
         while M > m and i > 0:
               M ← M/2 ; i ← i-1
         endwhile
         i' ← M/k ; m'' ← m-M
         i ← INT(m''/i') + ik + 1
         m'' ← m'' mod i'
    endif
endA2
```

2.3. Variants of the basic schema

2.3.1. Split control

Splits that are performed iff a collosion occurs are called *uncontrolled*. Splits are called *controlelled* if they also depend on other conditions or are performed even if there is no collision. A particularly useful control is called *load control*. Under this control, a split is performed when a collision occurs, but only of the load factor is superior to some treshold. This may concern the load factor, let it be $\hat{a}$ defined as usual (KNUTH [11]):

(8) $$\hat{a} = x/bM$$

where x is the number of records in the file, b is the bucket capacity and M is the number of primary buckets. The control may also take in to account the overflow buckets in which case the load factor, let it be $\hat{a}$, is defined as:

(9) $$\hat{a}' = x/(bM + b'M')$$

where M' is the number of overflow buckets and b' is the overflow bucket capacity.

In what follows the tresholds are denoted as g and g', respectively. We will show that, when the file undergoes insertions, the load control usually keeps the load factor almost equal to the chosen treshold. A similar control may keep the load factor greater or almost equal to a treshold, when the file undergoes deletions. Each time a deletion brings the load below this treshold, we may simply perform an operation called *grouping* which is inverse to splitting. A grouping moves thus the pointer one address backward and so decreases M. If the treshold for deletions is equal to the one for insertions, *the load of a* LH *file usually stays almost constant.*

2.3.2. Pointer independent address computing

It may surprise, but primary address may be computed in fact without the knowledge of the value of n. The following algorithm, analogous to that of VH1 (LITWIN [17]), proves it:

(A3)

$m \leftarrow h_j(c)$

if $m >= M : m \leftarrow h_{j-1}(c)$ endif

endA3

If n = 0, then it is obvious that (A3) works. If $n \neq 0$ then, if:

$$h(c) = h_j(c),$$

then:

$$h(c) < n < M$$

or:

$$2^{j-1} < n < M.$$

Thus (A3) terminates correctly. Else, we have:

$$h(c) = h_{j-1}(c).$$

Then, if:

$$h_{j-1}(c) = h_j(c)$$

then:

$$h(c) < 2^{j-1}N < M.$$

Thus (A3) also terminates correctly. Else:

$$h_j(c) >= M,$$

since the bucket h(c) is not yet split. Therefore, in this last case, the algorithm terminates correctly as well.

In particular, we may assume N = 1. The file level j is then a function of M. It follows, first, that the size of address space may be the only one parameter which LH needs for addresses computing. It follows, next, that the parameters of a classical hashing may suffice in order to construct a linear one. For instance, the knowledge of the number of the addresses for the hashing function and of the fact that a hashing by division is used, suffice in both cases.

2.3.3. Other split functions

Let $b_1,b_2,\ldots,b_i,\ldots$ be a sequence of randomly generated bits, with equiprobability of $b_i = 0$ and of $b_i = 1$. Such a sequence may be obtained using a random number generator. Let B_i be the integer with binary representation $b_i,b_{i-1},\ldots,b_1$. The functions h_i defined as follows:

(10) $\qquad h_i : c \mapsto h_0(c) + B_iN$

are, obviously, split functions for *any* h_0. Thus, LH may be constructed not only for a hashing by division, but for any usual hashing function.

Split functions with B_i given by a random number generator may be particularly interesting for N = 1. For this value each h_i hashes on 2^i addresses. If the hashing by division is applied, the address of c is, simply, the i least significant bits of c. The hashing by division may then be sometimes rather non-random, while a random number generator may still perform well. The choice of N = 1 is particularly useful since, first, there is no more problem to choose among several possible h_0. Next, LH covers all possible address space sizes. Finally, since the file may then be constituted even from only one bucket, a good load factor may be provided even for very few records.

B_i may also result from the *multiplication function* (KNUTH [11]), let it be h'. If w is the word size and A is an integer relatively prime to w, then the hashing with h' on 2^i addresses is defined as follows:

(11) $\qquad h'_i(c) = \mathrm{INT}(2^i((Ac/w) \bmod 1)).$

B_i is in this case constituted from the bits of $h'_i(c)$, taken in the reverse order. In other terms, the most significant bit of h'_i becomes the least significant in B_i etc.

Knuth shows that a particularly good choice for A is A = 6 125 423 371 . He also shows an algorithm computing (11) in only four instructions of the MIX assembler. Finally, he shows that his algorithm is usually faster than the hashing by division. To compute B_i through the multiplication function may thus be faster than through a random number generator.

In particular, Knuth shows that h' is a *scrambling function*. This means, first, that its partial result, let it be f(c);f(c) = Ac mod w; is such that if $c' \neq c''$, then $f(c') \neq f(c'')$. Next, this means that the transformation

$c \rightarrow f(c)$ tenders to randomaize the keys. Therefore, the following split functions may be constructed:

$$h_i(c) = f(c) \bmod 2^i N \qquad (12)$$

which may perform better than the direct hashing by division.

Finally, split functions may also be constructed for alphabetic or variable-length keys. In particular, the individual words of such a key may be simply combined into a single word, to which any of the previously discussed functions may then be applied. Any of the combinations suggested by Knuth may be used, the addition mod w for instance.

2.3.4. General definition of split functions

LH may be seen as VH1 in which split addresses have been predefined. VH1 may be generalized into an algorithm called VH2.Furthermore, it may be assumed that split addresses are, in fact, predefined for VH2. The conditions (1) and (2) may than be generalized as follows:

- let K be a parameter which value is fixed when the file is created; $K = 1,2,\ldots$. Let it be $k_i = K + i \bmod K$. Let N be an integer, $N > 1$. The hashing functions h_i; $i = 1,2,\ldots$; are split functions for a hashing functions h_0, if the following conditions are respected:

- for $i = 0,1,\ldots$:

$$h_i : C \rightarrow \{0,1,\ldots,N_i-1\} \qquad (13)$$

$$N0 = KN$$

$$N_{i+1} = N_i + N_i/k_i;$$

- for any c, either:

$$h_i(c) = h_{i-1}(c) \qquad (14)$$

or:

$$h_i(c) = N_{i-1} + \mathrm{INT}(h_{i-1}(c) \,/\, k_i).$$

For example, if N = 5 and K = 4, then the successive N_is are: 20, 25, 30, 35, 40, 50, 60, 70, 80, 100, ... (note the similarity to (6)).

As previously, we assume that each h_i should hash randomly. It follows that the probability that a key changes the address after a split, let it be p, is now $p_k = 1/(k_i+1)$. If k = 1, then (13) and (14) are, simply, (1) and (2) and $p_k = 0.5$. For greater K, p_k decreases and the distribution of records within LH file becomes more uniform. First, higher load factor obviously results for uncontrolled splits. Next, when the treshold increases, load control with K > 1 should lead to a better search performance. However, it is easy to see that for such K, to perform a split, usually needs more accesses. herefore, insertions and deletions will be more costly than for K = 1 as well.

K > 1 implies that the address space doubles not after one, but after K trips of the pointer. Each of K trips may then be called a *partial expansion* of the address space. Partial expansions may result from formula others than the above, these introduced by LARSON [13] in particular. Performance resulting from (13) and (14) for K > 1 being quite similar to these of the Larson's schema, only the case of K = 1 is discussed in what follows.

3. PERFORMANCE ANALYSIS

3.1. Address computing

If the allocation is contiguous, LH is obviously almost as simple and fast as the classical hashing. If the allocation is non-contiguous and (3) or (4) are used, then this is also the case, as long as T may be entirely in core. The use of (6) needs few more instructions, but the computing of (A2) is also very fast; it is quite clear that, for any j, the "while" loop is in the average executed at most twice.

A four byte word allows that values of n and of j to go up to 2^{32}, i.e., allows the LH file to grow up to more than four billions buckets. For any number of insertions, (A1) enters thus even a very small core. If the allocation is contiguous, since (3) is used, the computing of the physical address also needs only a few bytes. If the allocation is non-contiguous, the core is mainly needed for T. If (A2) is used, the size of T is, obviously, not greater than jk+1. For k = 10 which is largely sufficient in practice, 301 words are then sufficient for a file which increases even a billion times. Thus, in practice, no matter if the allocation is contiguous or not, no matter how small is the core and how high is the number of insertions, the computing of an address resulting from LH, *never requires a disk access*.

However, the disk storage is usually needed if the allocation is totally distributed. (5) shows then clearly that any address is computed in no more than one access. On the other hand, since the disk is required only for T and since T contains only the pointers to buckets, the disk storage required then by LH is the minimal one. It is usually much smaller than that of the index of a VSAM file, since the index contains keys and internal pointers and since its load factor is lower. It is also several times smaller than the storage required by the tables of VH0, DH and EH, either because of their much lower load factor (VH0, EH) or because of the internal pointers (DH).

3.2. Uncontrolled split

3.2.1. Access performance

We now assume that overflow records are addressed through separate chaining and that the file is created by x insertions; x = 0,1,.. . We also assume that each h_i hashes randomly. Finally, we assume (and we have now right to do it) that the computing of an address never requires a disk access. By s', s'', s''' we denote the mean number of accesses per successful search, per unsuccessful search and per insertion. By $\hat{s}$ we denote the mean number of accesses per split. These coefficients will be called *costs*.

Fig. 2 shows curves of s'(x) obtained through simulations for bucket capacities b = 1,5,10,50. For b = 1,5 we have not shown the first splits, since they correspond to x < 10 and to s' practically equal to 1. For b = 5, we have also shown the evolution of the file level j. The curves describe the the file which is created with only one bucket and which undergoes $x > 2^{13}b$ insertions. At the end, the number of records in the file is thus more than 8000 times greater than the number of records which could be found in one access in the initial one. It does not even make sence to compute what would be the deterioration of access performance, if the classical hashing would have been used.

For each b, when x increases, the curve is first irregular and therefore displays the existence of a *transient state*. After a few insertions, the file reaches a *stable state*, where the curve is a periodic function of $\log_2 x$. On one hand, it means that in the stable state s' does not depend on x, but on the relative position of the pointer. On the other hand, it means that s' does not increase for whatever the number of insertions could be. We therefore may conclude that for any bucket capacity and any number of insertions, the mean number of accesses per successful search stays close

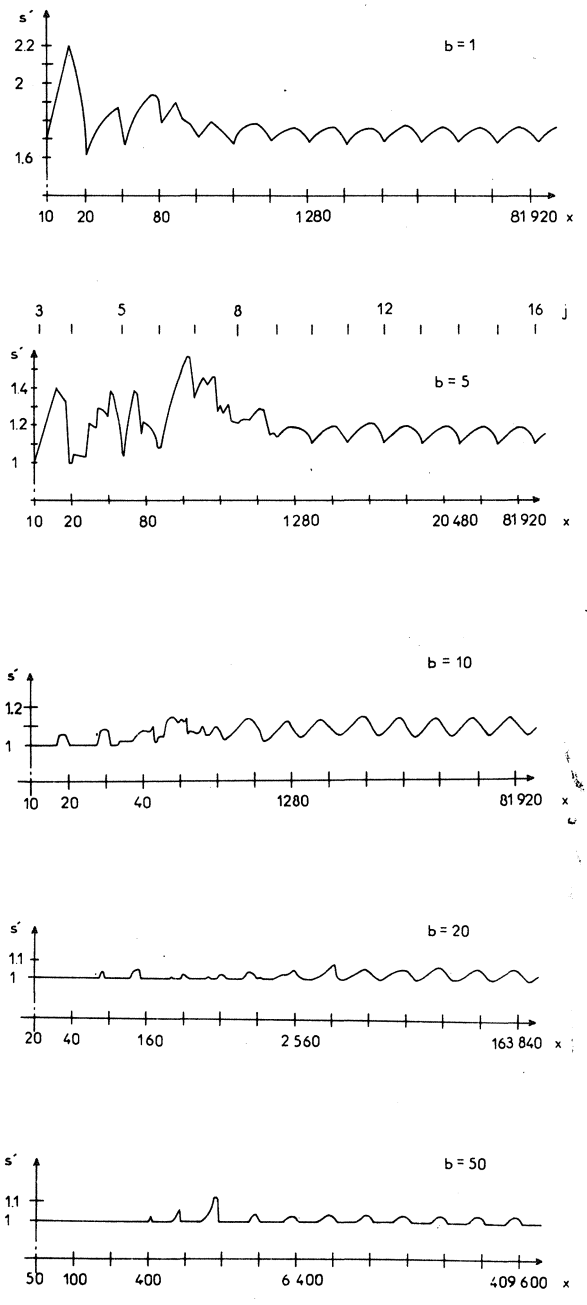


Fig. 2. Mean number of accesses per successful search for linear hashing with uncontrolled split.

to 1. For large buckets we may even consider that we have always s' = 1! LH is thus a very important algorithm, since, first, *no other algorithms attaining such an almost ideal performance are known*. Next, it performs several times better than trees. We recall that B-trees need in general 4,5 accesses while binary trees need typically more than 10, since $s'(x) \hat{=} \log_2 x$ (KNUTH [11]).

The transient state may be disregarded, since the performance is good and x is neglectably small. Since the curves in the stable state are periodical, they may be characterized by the values of s'_{min}, s'_{max} and of s'_{ave}, which is the mean value of s' over one period. The values corresponding to the curves are displayed in table 1 (a).

	b	1	5	10	20	50
(a)	s'_{ave}	1.73	1.16	1.07	1.02	1.00
	s'_{max}	1.77	1.20	1.11	1.06	1.03
	s'_{min}	1.68	1.10	1.02	1.00	1.00
(b)	s''_{ave}	1.62	1.28	1.19	1.12	1.06
	s''_{max}	1.63	1.32	1.26	1.24	1.23
	s''_{min}	1.6	1.20	1.08	1.02	1.00
(c)	s'''_{ave}	7.91	3.97	3.14	2.67	2.35
	s'''_{max}	8.92	4.07	3.45	3.09	2.71
	s'''_{min}	6.45	3.34	2.48	2.11	2.00
(d)	$\hat{s}_{ave}$	6.94	6.09	5.99	5.98	5.98
	$\hat{s}_{min}$	5.80	5.10	5.00	5.00	5.00
	$\hat{s}_{max}$	8.62	7.45	8.10	8.85	10.85
(e)	$\hat{a}_{ave}$	1.30	0.66	0.61	0.59	0.59
	$\hat{a}_{max}$	1.30	0.67	0.63	0.65	0.70
	$\hat{a}_{min}$	1.30	0.65	0.59	0.55	0.51
	$\hat{a}'_{ave}$	0.8	0.63	0.59	0.59	0.59

Table 1. Performance of linear hashing with uncontrolled splits:

(a) successful search
(b) unsuccessful search
(c) insertions
(d) split
(e) lead factor.

The analysis of s'' through simulations and through modelling (LITWIN [18]) shows also a transient state and a stable state. Both states correspond of course with these of s'. The performance of unsuccessful search with LH in the stable state are displayed in the table 1.(b). As in the case of classical hashing using the separate chaining (see KNUTH [11]), they are better than the performance of the successful search for b = 1 and poorer for b > 1. However, as before, for any bucket capacity and any number of insertions s'' stays close to 1. The limit value for s''_{max} when b increases is 1.24 (LITWIN [18]).

Table 1.(d) shows the characteristics of the split cost. Unless split is performed for the address of the collision, this cost is at least five accesses (two accesses in order to store the overflow record for the bucket m, three accesses in order to split the bucket n). The split needs more accesses if the bucket n overflows. This case is obviously the most frequent for b = 1; that is why $\hat{s}_{ave}$ is maximal. The number of overflow records on n is obviously the shortest for n close to 0, $\hat{s} = s_{min}$ corresponds to such values of n. Inversely, the longest chains exist for n close to 2^i, $\hat{s} = s_{max}$ corresponds thus to the end of a trip of the pointer. $\hat{s}_{max}$ increases with b, since the chains become longer while the use of the separate chaining implies one access per every record in the chain. However, $\hat{s}_{ave}$ reveals practically independent of b.

Finally, table 1.(c) lists s''', i.e., the cost of an insertion. For b = 1 almost 8 accesses are needed, since split cost is the highest and since a split results from almost any insertion. For larger b, s''' falls down quickly, since the proportion of insertions leading to a split decreases and the others need typically 2 to 3 accesses. If b $\gg$ 20, which is a typical value for files, s''' oscillates between 2 and 3. Thus, first, as it was the case of the other costs, insertion cost of LH may also stay always close to its theoretical minimum. Next, even in the worst case, i.e., for b = 1, the insertion cost is still typically much smaller than for trees, since, for instance, for $x = 10^5$, a binary tree leads to s''' > 16 (KNUTH [11]).

3.2.2. Load factor

The characteristics of load factors $\hat{a}$ and $\hat{a}'$ are shown in the table 1.(e). For b = 1 the load factor is constantly equal to 80%. This conjunction of such a good load and of the previously shown access performance makes LH probably the best known tool for dynamic tables. For higher values

of b, the load is going down to the average value of almost 60% and so the load of LH with uncontrolled split may be almost 10% worse than the load of a B-tree. However, better access performance is usually preferrred to a slightly better use of the increasingly cheaper disk space.

3.3. Controlled split

If the load is controlled and the threshold g is greater than $\hat{a}_{max}$, then the load is practically equal to g. It is obvious that the higher g is, the worse must be the access performance, since the ratio of overflow records increases. Simulation studies show, however, that substantial increases to the load factor may be achieved while the value of the access performance still stays excellent.

For instance g = 0.75 and b = 5 leads to a load which is almost 10% higher than the one for the uncontrolled load. The corresponding access performance is still very good, since $s'_{ave} = 1.25$, $s''_{ave} = 1.43$ and $s'''_{ave} = 3.84$. For b = 50, the same g leads to a 16% improvement, while the access performance becomes: $s'_{ave} = 1.28$, $s''_{ave} = 2.38$ and $s'''_{ave} = 3.46$. For many applications the above trade off may be significant.

Higher thresholds increase the length of overflow record chains. On one hand, the storage occupied by overflow buckets is then no more negligible. On the other hand, larger overflow buckets must lead to better access performance. For higher thresholds, more stable load factor results thus from the control on $\hat{a}'$ and it is better to choose b' > 1.

The threshold corresponding to the load control on $\hat{a}'$ is denoted g'. For given values of b and of g', access performance depends on b'. If g' is higher than $\hat{a}_{max}$ of uncontrolled split, then obviously, neither b' = 1 nor b' >> 1 can provide the best access performance. Therefore, they are b's which are the optimal ones. Fig. 3 shows curves of s' corresponding to the minimal s'_{ave} for b = 10,20,50 while g' = 0.75,0.9. It also indicates the corresponding optimal b's. Table 2 displays the performance of the corresponding stable states and the performance for g' = 0.85. All these results are obtained through simulations.

It first becomes apparent form the figure, that if g' = 0.75, then s' is always almost 1. It is also apparent that s' stays close to one even for g' = 0.9! In other words, LH does not only find a record in general in one access independently of the number of insertions, but may also use almost the minimal storage! In particular LE achieves not only a much better access performance than B-trees but also saves more than 20% in storage!

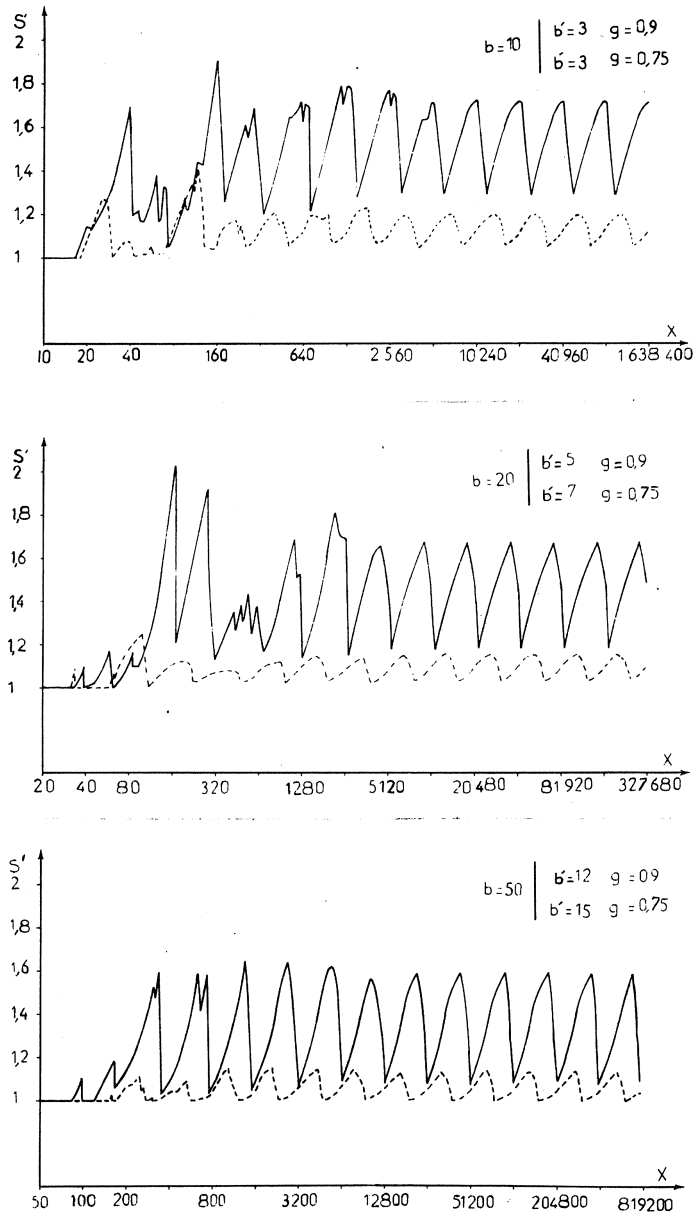


Fig. 3. Mean number of accesses per successful search for linear hashing with the load kept equal to 75% (- -) and to 90% (——).

Furthermore, with respect to table 1, table 2 shows that it is rather worthwile to choose a high threshold, even if one seriously cares about performance other than s'. For g' = 0.9, load control improves the load factor up to 31% in mean and up to 40% with respect to the worst value. The price to pay for such a significant improvement in load seems rather low, since, first, s''_{ave} increases only by 1.3 accesses. Next, s'''_{ave} increases only by 1.5 accesses. Only $\hat{s}_{ave}$ deteriorates more substantially, since it increases by 3.4 accesses. However, for b = 50 and g' = 0.85, this deterioration stays small, since it not exceeds 1.3 accesses. Finally, $\hat{s}_{max}$ may do not deteriorate at all, being, even for g' = 0.75, better for all each b. For b = 50 the gain is quite important, since it reaches 3.4 accesses. These gains are obviouwly due to b' > 1.

	b	10			20			50		
	b'	4	3	3	7	6	5	15	14	12
	$\hat{a}'$	0.75	0.85	0.90	0.75	0.85	0.90	0.75	0.85	0.90
	s'_{ave}	1.14	1.33	1.57	1.08	1.24	1.44	1.05	1.20	1.35
(a)	s'_{max}	1.22	1.47	1.73	1.15	1.35	1.65	1.13	1.35	1.59
	s'_{min}	1.06	1.15	1.31	1.02	1.07	1.17	1.00	1.02	1.09
	s''_{ave}	1.37	1.99	2.48	1.29	1.80	2.45	1.27	1.78	2.37
(b)	s''_{max}	1.48	2.15	2.75	1.43	2.11	2.87	1.49	2.19	2.95
	s''_{min}	1.21	1.52	2.06	1.10	1.38	1.85	1.02	1.18	1.66
	s'''_{ave}	3.42	4.05	4.68	2.91	3.42	4.17	2.62	3.10	3.73
(c)	s'''_{max}	3.67	4.71	5.43	3.11	3.60	4.43	2.68	3.38	4.15
	s'''_{min}	2.91	3.29	3.73	2.51	2.82	3.25	2.27	2.43	2.91
	$\hat{s}_{min}$	5.05	5.60	6.15	5.05	5.75	5.9	5.00	5.45	6.20
(d)	$\hat{s}_{ave}$	6.34	7.82	9.48	6.24	7.6	9.47	6.24	7.27	9.02
	$\hat{s}_{max}$	7.10	9.50	11.1	7.35	8.75	11.6	7.15	8.5	10.50

Table 2. Performance of linear hashing with controlled load:

(a) successful search
(b) unsuccessful search
(c) insertion
(d) split cost.

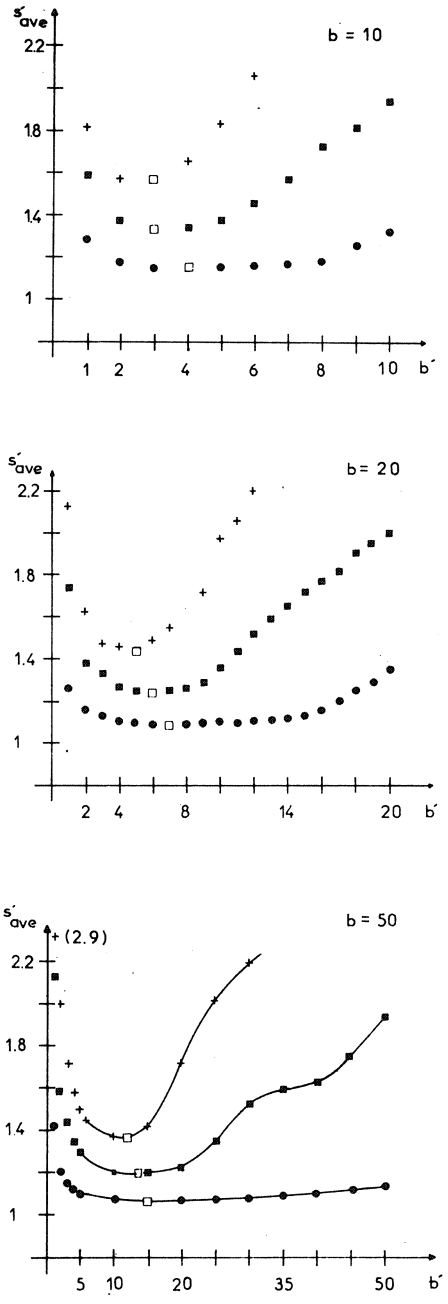


Fig. 4. Mean number of accesses per successful search as a function of the size of the overflow bucket.

Fig. 4 displays s'_{ave} in function of b' with values of b and of g' from table 2 as parameters. It appears that for a file loaded up to 75%, access performances are almost the same for a large number of values of b'. For these values, performances are, in addition, almost independent of b. For example, s'_{ave} is smaller than 1.2 accesses, for all b' between 2 and 8 when b = 10, for all b' between 2 and 16 when b = 20 and for all b' between 2 and 50 when b = 50! Since practical constraints may frequently impose bucket capacities which are not the optimal ones, this stability of excellent access performance is one more important property of LH.

The figure shows, however, that when the load becomes higher, b' should be kept closer to the optimal one. The practical rule which appears is then:

$$b/5 \leq b' \leq b/3.$$

For b > 20, even if the lower bound is b/7, we stay under a mean of 1.5 accesses.

It also becomes apparent that the access performance deteriorates less when g' increases from 75% to 85%, as it deteriorates when g' increases from 85% to 90%. In other terms, the last 5% are the most expensive ones and it is not recommended to further increase g'.

4. CONCLUSIONS

If a file or a table addressed with LH is static, then the performance is simply that of classical hashing, i.e., the best known. If they are dynamic, then the mean number of accesses per search stays close to 1 independently of the number of insertions and for load factors reaching 90%. If the bucket capacity is greater than 10 records, then almost any record is found in one access. Finally, address computing is almost as simple and rapid as for classical hashing. The comparison of these performances with those of the classical hashing, of trees and even of other algorithms for virtual hashing, shows that for the search by the primary key, *Linear Hashing is the best performing technique known*.

High and constant load factor means that LH store records always in an almost minimal storage. A sequential search scans thus an almost minimal number of buckets, i.e., is almost as fast as possible. If the classical hashing is used, the number of primary buckets is fixed when the file is created. If the number of records is then i times less than expected, a

sequential search with LH is almost i times faster. This property of LH is also important, since sequential searches are quite frequent.

With respect to trees, in addition to much faster search, LH provides much simpler algorithmic. This is, first, the case of the algorithms for a search, for an insertion and, especially, for a deletion. This is also the case of the algorithms for concurrency control, since only the key and the pointer must be locked, instead of a path in the tree. Also, there is not a problem of an inconsistency which may occur in a tree because of keys duplicated between the file and its index. Thus trees stay more advantageous only when the file must be searched in one order.

LH is, of course, primarily devoted to applications where the file may heavily grow or shrink or where the number of records is unknown when the file is created. It may thus be very useful for compilers and text processing systems. It may avoid the painful estimations of the file sizes in a DBMS, from which files the delected records are, usually, not physically removed. It may also avoid performance deterioration for such files, rendering thus the very annoying reorganizations of the whole database (SCHNEIDERMAN [24]) unnecessary or less frequent. It is a good tool for the management of working spaces for queries to a DBMS, since the number of records retrieved by a query to a working space is, unknown in advance. It may also be used for a virtual management. On the other hand, since it works very small cores, LH renders dynamic files usable on micro-computers. Clearly, the applications of the Linear Hashing are very numerous.

Research on LH has just started and many possibilities are still open. Other criteria for control may be useful, SCHOLL [25] for instance shows that the performance may be excellent if we simply split one time for any gb insertions. M. Girault (Institut de Programmation) has suggested to consider splits and groupings as operations which are, finally, completely out of the algorithmic for a record insertion or deletion. He suggests further to leave splits and groupings to the competence of a dedicated processor which task would thus be to take care of performance of all files. This idea obviously leads to a new and interesting type of an associative memory.

Furthermore, methods other than bucket chaining should be explored for overflow record addressing. The first investigations of open addressing show that it may work pretty well (KARLSSON [9]). Also, the properties of split functions should be investigated. Finally, much work is needed in modelling, since classical methods do not apply to dynamically created hashing functions. Especially, there are no models for the transient state and for a file with small buckets.

ACKNOWLEDGEMENTS

This work was sponsored by project SIRIUS.

REFERENCES

[1] AHO, A.V., J.E. HOPCROFT & J.D. ULLMAN, *The design and analysis of computer algorithms*, Addison-Wesley, Reading Mass. 1975.

[2] BLAKE, I.F. & A.G. KONHEIN, *Big buckets are (are not) better!* Journal ACM 24, 4 (Oct 1977), 591-606.

[3] CARTER, B., *The reallocation of hash-coded tables*. Com. ACM 16, 1 (Jan 1973), 11-14.

[4] COMER, D. *The ubiquous* B-*trees*. ACM Comp. Surv., 11, 2, (Jun 1979), 121-138.

[5] FAGIN, R., J. NIEVERGELT, N. PIPPENGER & H.R. STRONG. *Extendible hashing - a fast access method for dynamic files*, IBM Res. Rep. RJ2305, (Jul 31, 1978).

[6] GHOST, S.P. & V.Y. LUM, *Analysis of collisions when hashing by division*. Information Systems, 1-B (1975), 15-22.

[7] GUIHO, G., *Sur l'etude de collisions dans les methodes de hash-coding*, CRAS 274 (Feb 14, 1972).

[8] GUIHO, G., *Organisation des memoires, Influence d'une structure et etude d'une optimisation*. These de Doctorat d'Etat. Univ. Paris VI, (Jun 1973), 278.

[9] KARLSSON, K., *Resolution de collisions du hachage virtuel lineaire par une methode du type adressage ouvert*. Rap. D.E.A. Inf. Institut de Programmation, (Jun 1979), 81.

[10] KNOTT, G.D., *Expandable open addressing hash table storage and retrieval*. SIGFIDET Workshop on Data Description, Access and Control, ACM (1971), 186-206.

[11] KNUTH, D.E., *The Art of Computer Programming*, Vol. 3. Addison-Wesley, Reading Mass. 1974.

[12] LARSON, P. *Dynamic hashing*. BIT 18 (1978), 184-201.

[13] LARSON, P., *Linear hashing with partial expansions*. Proc. 6-th Conf. on Very Large Databases, Montreal (Oct 1980).

[14] LITWIN, W., *Auto-structuration d'un fichier: methodologie, organisation d'acces, extension du has-coding*. Res. Rep 77/11, Institut de Programmation, Paris, (Avr 1977), 102.

[15] LITWIN, W., *Methode d'access par hash-coding virtuel (VHAM): Modelisation, Application a la gestion de memoire*. Res. Rep. 77/16, Institut de Programmation, Paris, (Nov 1977), 50.

[16] LITWIN, W., *Une nouvelle methode d'acces par codage decoupe a un fichier*. Compte-rendus de l'Academie des Sciences, Paris, t. 286, (Avr. 1978), 695,698.

[17] LITWIN, W. *Virtual hashing: a dynamically changing hashing*. Proc. 4-th Conf. on Very Large Databases, Berlin, (Sep 1978), 517-523.

[18] LITWIN, W., *Linear virtual hashing: a new tool for files and tables implementation*. Res. Rep. MAP-I-021, I.R.I.A., (Jan 1979), 24.

[19] LITWIN, W., *Hachage Virtuel: une nouvelle technique d'adressage de memoires*. These de Doctorat d'Etat. Univ. Paris VI, (Mar 1979) 248.

[20] LITWIN, W., *Linear hashing:a new algorithm for files and tables addressing*. Proc. Int. Conf. On Data Bases, Aberdeen, (Jul 1980).

[21] LUM, V.Y., *General performance analysis of key to address transformation methods*. Com. ACM 16, (Oct 1973).

[22] MARTIN, J., Computer Datebase Organization. Prentice Hall, Inc. Englewood Cliffs, New Jersey, 1977.

[23] ROSENBERG, A.L. & L.J. STOCKMAYER, *Hashing schemes for extendible arrays*. JACM 24, 2, (Apr. 1977), 199-221.

[24] SCHNEIDERMAN, B., *Optimum data reorganization points*, Com ACM 16, 6 (Jun 1973), 23-28.

[25] SCHOLL, M., *Performance analysis of new file organizations based on dynamic hash-coding*. Res. Rep 347, I.R.I.A. - Laboria, (Mar 1979), 28.

[26] SPRUGNOLI, R., *Perfect hashing functions : a single probe retrieving method for static sets*. Com ACM 20, 11 (Nov 1977), 841-850.

TRIE HASHING

W. LITWIN
I.N.R.I.A.
78150 Le Chesnay, France

0. ABSTRACT

We propose a new algorithm for hashing. Contrary to the usual hashing, ours stores the record in order. Furthermore, the file may be highly dynamic, even may be constituted entirely with insertions. The load factor is typically about 70%. The search for a record is performed in only one disk access, for files attaining millions of records. No other algorithms providing such a fast search in an ordered file are known.

1. INTRODUCTION

Hashing is a well-known addressing technique. "The idea in hashing is to chop off some aspects of the key and to use this partial information as the basis for searching" (KNUTH [11]). Since longtime, hashing is considered as the best performing technique for a key search in a reasonably static file. Recently, it was proven that this is also true for dynamic files, even entirely constituted by an unknown number of insertions. In particular, the algorithm called linear hashing allows to find a record typically in one disk access, no matter how high the number of insertions may reveal (LITWIN [18], LARSON [13]). The load factor may be kept constant, up to 90%. No other technique attaining such performance is known.

The usual algorithms for hashing store records in disorder. This is also the case of the linear hashing. If the order is needed, the best actual technique for files are B-trees (BAYER & McCREIGHT [2], BAYER [3], KNUTH [11], COMER [4]). A record with a given key is then found typically in 3-5 accesses. The load factor is usually about 70% (KNUTH [11]).

However, several very interesting proposals have been made for hashing functions which store the records in order (KNOTT [10], DEUTCHER [5], GUIHO

[9], SPRUGNOLI [20], MALY & KAMPA [19],...). Access and memory occupation performance of these functions are worse than these of unordered hashing, but may be better than these of B-trees. Unfortunately, each proposal presents also some drawbacks. The computation of the hashing function is typically very complex. Acceptable access performance lead to a weak load factor and vice versa. Finally, few insertions may lead to the need for a rehashing. That is probably why such hashings are not widely used.

The algorithm which we propose in this paper, definitely proves that hashing may be an excellent technique also for ordered files. As the linear hashing, the hashing which we define is a virtual hashing (LITWIN [14,15,16, 17,18]). This means that the hashing function is dynamically modified in the course of some insertions or deletions. In this sense, it is also a kind of dynamic hashing (LARSON [12]) and of extendible hashing (FAGIN, NIEVERGELT, PIPPENGER & STRONG [6]).

The function developped by the algorithm may be represented by data structure called trie (KNUTH [11]). Our trie grows gracefully with the insertions. The file also grows gracefully through the mechanism of splits of overloaded buckets, similar to that of B-trees. The algorithm uses thus the ideas in tree searching (especially in prefix B-tree searching), in digital (trie) searching and in hashing; It may be seen as a synthesis of these three techniques.

If the key values of records which present for storage are drawn from the key space randomly, trie hashing typically leads to the load factor of 70%, i.e. to this of a B-tree. The size of the trie grows with the file, but stays typically small enough for keeping the trie in core. For instance 3.5 Kbytes suffice for the trie of a file of 35000 to 140000 records, while 64 Kbytes suffice for a file of 780000 to 2200000 records. When the trie is in core, any successful or unsuccessful search for a record in the file is performed in exactly one disk access. The average insertion cost is then almost two accesses.

Section 1 presents the algorithm. Section 2 discusses performance. Section 3 concludes the paper.

2. THE ALGORITHM

2.1. Main concepts

We consider that each record is identified with a value called *primary key* or, shortly, *key*. Social security numbers, unique names etc. are examples of keys. Key space for a file is finite. The portion of the record which is not the key is unimportant for a work on the addressing.

A key is represented with *digits*. A digit may be:

- binary i.e. a bit or two consecutive bits or some consecutive bits,
- numerical i.e. a number 0,1,...,9 or a number 00, 01,..,99 etc.,
- alphabetical i.e. _, A, B,... or __, A_, AB,... , where '_' means space,
- etc.

The digits of a key are numbered starting from the most important ones. The i-th digit; i=0,1,.. ; will be denoted c_i. Digit representation of a key c will be denoted $c_0c_1...c_k$. Keys values and digit values are totally ordered according to a certain law. It thus makes sense to compare any two keys or digit digits, they must be either equal or one is smaller than the other. The maximal digit value will be denoted ':'.

The files are stored on a secondary storage, called *disk*. A cell of the disk is called *bucket*. The buckets of a file are numbered 0,1,... . The number of a bucket is called *bucket address*. Each bucket may contain the same number of records which is called *bucket capacity*. Bucket capacity is denoted b and we assume that b >> 1, b >= 4 for instance. An access to the disk, shortly an *access*, brings to the main memory, called core, one bucket.

A search for a record with a given key, shortly *a search*, consists of:

- address computing through an algorithm usually called *key-to-address transform*,
- some accesses bringing to the core keys for examination.

Algorithms computing a hashing function or traversing a tree are key-to-address transforms. Key-to-address transform may cost some accesses, this is typically the case of trees. It does not cost accesses when all data needed for computing are in core, this is typically the case of hashing. If the the access to the bucket with the computed address and, eventually, the accesses to the overflow buckets find the record, then the search is qualified of *successful*. Else we are in the case of an *unsuccessful search*.

The records in a bucket are stored in order of their keys. This means that if $c_1 < c_2$, then the record with c_1 precedes in the bucket the record with c_2. We will speak about an ordered file or about records stored in order iff for any two buckets all keys in one bucket are greater than all keys in the other. To traverse a file in order will mean that all keys are examined in the ascending order. If we can store in the core only one bucket at the time, then the ordered file may be traversed accessing any bucket once. To traverse a disordered file, like the one resulting from the usual hashing, requires typically much more accesses. In practice, a disordered file must be sorted prior to the traversing.

A *collision* occurs when a key presents for storage in a bucket which is already full. The key which position in the ordered sequence of b+1 keys is the nearest one to (b+1)/2, will be called the middle key. For instance, if b = 4, then the middle key is the third one. The middle key will be denoted c'. It may be either one of the keys on the bucket or the new key.

The algorithm dynamically creates a particular hashing function. The storage representation of this function is a kind of a *trie* (KNUTH [11]). Our trie is composed from *nodes* , numbered 0,1,.. . The number of a node will be called *node address*. A node contains the following data (fig. 1):

- two pointers qualified respectively of *upper* and of *lower* (UP,LP). Each pointer may be either external or internal. The value of an internal pointer is a node address. The value of an external pointer is either the address of a bucket or a special value denoted 'nil'. This last value means that the pointer does not indicate any bucket.
- a field called *digit field* (DF). Each digit field contains a digit value and a digit number (DV, DN).

We assume that if a pointer is an internal pointer to a node, let it be m, then its value is -m.

2.2. File constitution

For the purpose of this section we assume that the file is constituted entirely with insertions. We also assume that digits are alphabetical. Finally, we assume that keys for storage are drawn randomly from the key space. We will constitute an example file containing the most used english words (KNUTH [11], vol2, 489, fig. 32). The words are inserted in decreasing order of frequency (fig. 6). We assume that bucket capacity is b = 4.

Trie hashing constitutes a file according to the following scheme:

(1) - The bucket 0 is allocated.

We thus provide the storage only for b records, i.e. only for 4 records.

(2) - It is assumed that the file is provided with the hashing function which attributes the address 0 to all keys.

The first b records are thus inserted to the bucket 0 (fig. 2a).

(3) - The first collision occurs when the record b+1 presents for storage (fig. 2a). The following operations are performed (fig. 2):

(3.1) - We select the shortest sequence of digits $c'_0c'_1\ldots c'_K$ such that for some of the b+1 keys we have:

(3.1.1) $\quad c_0c_1\ldots c_K > c'_0c'_1\ldots c'_K$

We say that these digits split the set of b+1 records. For the first split, K = 0 typically suffice, since c_0 may take many values. In occurrence we have c'_0 = 'o' (fig. 2b).

The atypical case of K > 0 is discussed in the next section. For K = 0 we proceed as follows:

(3.2) - Let M' be the number of nodes already created. We create the node M'. Let m be the address of the bucket for which the collision occurs. Let i be the number of the selected digit. We set up the values within the node M' as follows:

LP = m ; UP = M ; DV = c'_i DN = i.

In occurrence we have (fig. 2b):

LP = 0, UP = 1, DV = c'_i, DN = 1.

(3.3) - Let M be the number of buckets of the file. We allocate the bucket M.

In occurrence M = 1. We thus extend the file with the bucket 1.

(3.4) - All keys which respect the condition (3.1.1) are inserted to the bucket M. All others are reinserted to the bucket m.

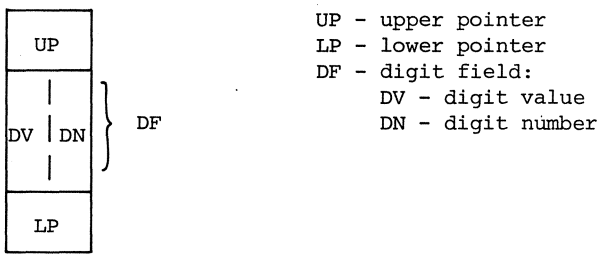
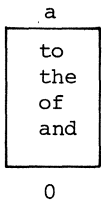
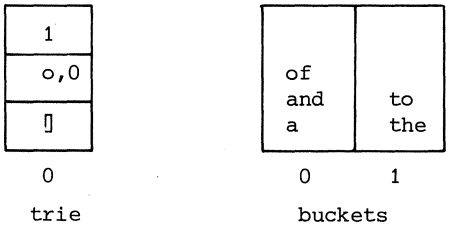


Fig. 1. Node of the trie

(a)



(b)



(c)

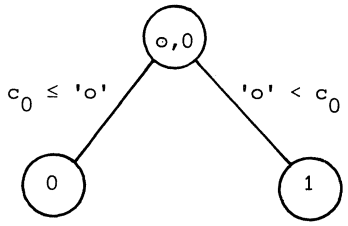


Fig. 2. First split in the example file (a) the collision, (b) file after the split, (c) graph of the trie.

In occurrence (3.1.1) leads to the comparison on co. Within b+1 records, all records with c_0 =< 'o' keep the old address. All other records are now in the bucket 1. Instead of b records, the file may now accomodate 2b records.

(4) - We consider, henceforward, that any new record is inserted to the bucket in which it would be, if it would undergo all already performed splits.

This means that the hashing function was dynamically modified. It now provides the addresses 0 and 1, instead of 0 only. Henceforward, a record is inserted according to the comparison to DV in the node. If $c_0 > c'_0$, then the record goes to the bucket pointed by UP. In occurrence (fig. 2b) it thus goes to bucket 1. Else the record goes to the bucket pointed by LP. In occurrence, it thus goes to bucket 0. It is clear that the resulting file is ordered.

The above rules may be seen as the definition of a graph like this of the fig. 2c. The root of the graph contains c'_0. The left branch corresponds to the choice when c_0 =< c'_0. The right branch corresponds to the inverse result. The values in the leaves (the external nodes others than the root), are the actual addresses of the buckets of the file. In order to traverse the file, it is necessary and it suffices to follow the leaves from the left to the right.

The graph of this kind is called trie (KNUTH [11]). To use a trie allows us to "chop off some aspects of the key", in occurrence the first digits, "using this partial information as the basis for searching" (KNUTH [8]). Thus in our case, the trie represents a hashing function which, in addition, is dynamically defined. We speak about trie hashing because of the first property. Our trie hashing is a virtual hashing because of the second one.

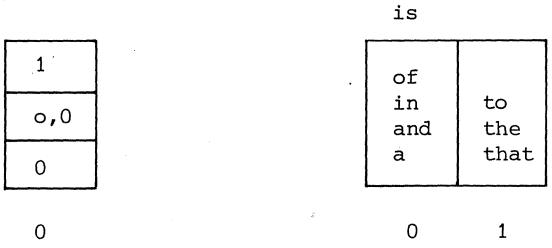
After some more insertions, either on the bucket 0 or on the bucket 1 a collision occurs. On the fig. 3a, this is the case of the bucket 0. The following operations are then performed:

(5.1) - Let m' be the address of the node containing the pointer to the overloaded bucket. This pointer is replaced by the internal pointer to the node M', i.e. the new node.

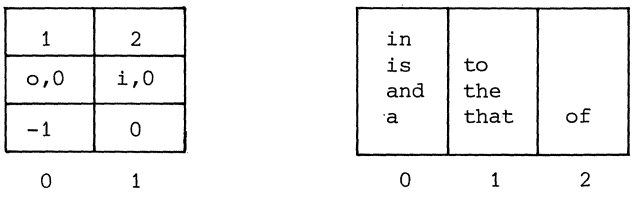
In occurrence it leads to LP = -1 in the node 0 (fig. 3b).

(5.2) - The step (3.1) is applied.

(a)



(b)



(c)

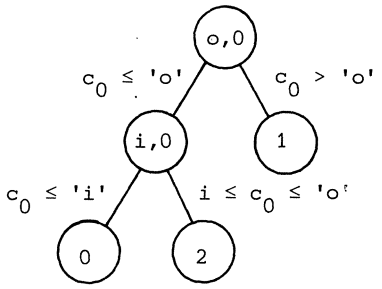


Fig. 3. Second split. (a) File when the collision occurs, (b) file after the split, (c) graph of the new trie.

Again, typically, c'0 provides the split. In occurrence, the selected digit is 'i'. The case of K > 0 is discussed later.

(5.3) - The step (3.2) is applied, i.e. the new node is created and filled up with values.

In occurrence the node 1 is created with values as follows (fig. 3b):

LP = 0, UP = 2, DV = 'i', DN = 0.

(5.4) - Step (3.3) is applied i.e. the new bucket is allocated.

In occurrence, this is the case of the bucket 2.

(5.5) - Step (3.4) is applied, i.e. the records are inserted according to their new addresses.

Fig. 3b shows the file after this split. The file grew up to three buckets. buckets. As previously, the file may be traversed in order. The new hashing function corresponds to the trie with the graph obtained as follows (fig. 3c):

- The address in the leave corresponding to the bucket which overflows is replaced with the content of the digit field of the new node.
- Two new leaves are appended to this node. The left one indicates the address of the collision. The right one corresponds to the new bucket.

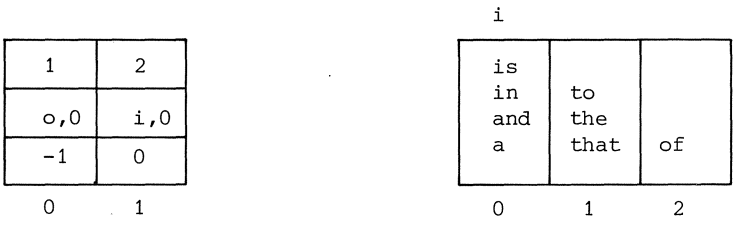
(6) - The rule (4) is applied to any new insertion.

This means that, henceforward, the address of a record is computed as follows:

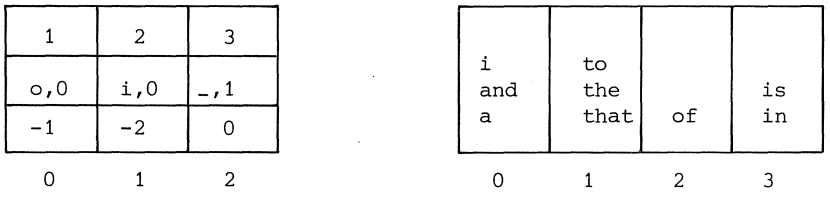
- The computing starts with the node 0.
- If $c_0 > c'_0$, then the upper pointer is chosen. It this pointer is external, as on the fig 3c, it indicates the address of the record.
- Else the lower pointer is chosen. If this pointer is external, it also indicates the address of the record.
- If however the selected pointer is internal, it means that the node indicated by this pointer should be examined.

In occurrence, this is the node 1.

(a)



(b)



(c)

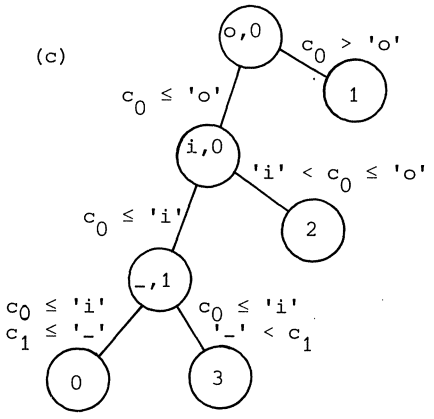


Fig. 4. Third split, (a) the file when the collision occurs, (b) the file after the split, (c) graph of the new trie.

- The key is compared to the content of the digit field in this node, following the previous rules. Since both pointers are now external, an address is chosen.

In occurrence, if c_0 > 'i' (but smaller than 'o'), the address is 2. Else the address is 0. For instance the record 'this' should be stored in the bucket 1, the record 'i' in the bucket 0, while the record 'not' should go to the bucket 2.

If insertions continue, further collisions occur. The application of the above rules allows to extend gracefully the file with some buckets, i.e. the buckets 3,4,... . A consequence of splitting process is that for each bucket undergoing splits, the number of values possible for c_0 decreases. Sooner or later, for a bucket, the following situation must occur:

(i) either for all keys greater than the middle one we have $c_0 = c'_0$,
(ii) or we have $c_0 = c'_0$ for all keys.

The case (i) corresponds to the fig 4a. We have c'_0 = 'i' and this digit does not provide the split. The case (ii) appears on the fig. 6. It corresponds to the 8-th split (node 7), which concerns the bucket 7, as LP in this bucket indicates. It also corresponds to the 9-th split (node 8), which concerns the bucket 0. In the bucket 7, all keys started with 'h'. In the bucket 0, they all started with 'a'.

In this situation we basically proceed as follows (some refinements are discussed in section 2.4):

(7.1) - the step (3.1) is applied.

Now, it typically results that K = 1 (the case of K > 1 is discussed in section 2.3). In occurrence, the selected digits $c'_0c'_1$ are respectively: 'i_', 'he', 'ar'.

(7.2) - We compute the address for $c'_0c'_1$. We note all c'_i for which we had during the computing i = DN and c'_i = DV.

In occurrence, this is the case of c'_0 for all three splits.

(7.3) - We apply the step (5.1).

In this case of 'i_', it results that LP of the node 1 points now at the node 2 (fig. 4b). In the case of 'he', UP of the node 6 points now at the node 7, while it pointed at the bucket 7 when the collision occurred for the

record 'had'. Finally, in the case of 'ar', LP of the node 3 points now at the node 8, while it pointed at the bucket 0 in the moment of the collision caused by the record 'at'.

(7.4) - We consider that the selected digits are the digits which were not met in a node when (7.2) was performed. If only one digit is selected, we apply the step (3.2).

In occurrence, we create, respectively, the node 2, the node 7 and the node 8. The case of more than one selected digit will be discussed later. Note that this definition of selected digits applies to the case when c'_0 suffices for splitting.

(7.5) - We apply the steps (3.3) and (3.4).

In occurrence, some records leave the buckets 0, 7 and 0 respectively, moving, also respectively, to the buckets 2, 8 and 9.

(8) - As previously, we assume that the address for a new record results from the rule (4).

In the case of the split with 'i_' for instance, (8) implies the choice among 4 addresses (fig. 4c), according to the following algorithm:

```
if c0 > 'o' : m = 1
else
   if c0 > 'i' : m = 2
   else
      if c0c1 > 'i_' : m = 3
      else
         m = 0
      endif
   endif
endif
```

For further insertions, we continue to apply the above rules. It might occur that despite of using c'_i for a bucket, we again use for this bucket c'_j with $j < i$. This is for instance the case of the 4-th split for the bucket 0, where c'_0 = 'a' provides the split.

However, the rules lead to the progressive passage for all buckets on which collisions occur, to $c_1, c_2, \ldots$. A digit c_{i+1} is used when c_i does not provide a split anymore and the rules allow the file to grow gracefully as much as needed. The trie also extends gracefully, with one new node per split, as long as the basic schema is applied. The records are always stored in order. The increasing order of key value corresponds to the leaves of the trie from the left to the right. Any collision leads to a split, thus the file has no overflow records. Therefore, as long as the trie is entirely in core, a successful search, as well as an unsuccessful one, costs exactly one disk access.

2.3. Full algorithm for splitting

The basic schema assumes that for any split it suffices to select one new digit. This is the typical case under the assumption of random choice of key values. However, it may occur that more than one digit is selected. For instance, this is the case when all first b+1 records start with the same letter, let it be m (fig. 5a). The middle key is 'mile', the selected digits are 'mi'.

In such a case we proceed as follows:

(9.1) - We apply the step (3.2) to the node m' unless the first selected digit is c'_0.

(9.2) - For each selected digit $c'_I, c'_{I+1}, \ldots, c'_{K-1}$; I = 0,1,.. ; we create the node M' with the values:

$$UP = \text{'nil'} \; ; \; LP = -M' \; ; \; DV = c'_i \; DN = i$$

i = I,..,K.

We thus create K = I nodes containing successively all selected digits, except the last one. Each node points at the next one through LP. All upper pointers do not indicate any address since for all of them UP = 'nil'. Therefore, there are no allocated buckets. On the fig. 5b, (9.2) creates one node which is the node 0.

(9.3) - For the last selected digit we apply the step (3.2).

In occurrence, we create the node 1.

(9.4) - We apply the steps (3.3), (3.4) and the rule (4).

(9.3) and (9.4) mean simply that for the last digit we come back to the basic schema. As it is the case of the basic schema, the split leads thus to the allocation of only one bucket which is in occurrence the bucket 1 (fig. 5b). However, it creates more than one node, K-I+1 in general, two in occurrence .

In fact, the whole step (9) is simply a recursive application of the basic schema. We apply this schema to each selected digit, from left to right. If the collision subsists, we avoid to allocate an empty bucket and we re-apply the basic schema to the new trie. We assume of course that if a further insertion finds 'nil', the bucket M is allocated and the pointer in the node is updated (fig. 5d).

The integration of all above rules leads to the following algorithm:

- Let p be the pointer selected during the address computing, i.e. either p = LP(m') or p = UP(m'). We assume that the file is created with M' = -1 and M = 0.

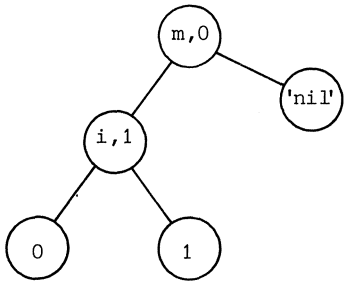
A1 : Trie hashing split algorithm.

```
If M' >= 0 : p(m') <-- -M' endif
compute c'0c'1...c'K
compute I
for i = I,K-1 :
   UP(M') <-- 'nil'
   LP(M') <-- -(M'+1)
   DN(M') <-- i
   DV(M') <-- c'i
   M' <-- M' + 1
endfor
UP(M') <-- M
LP(M') <-- m
DN(M') <-- K
DV(M') <--c'K
M' <-- M' + 1
allocate bucket M
split the records among the buckets m and M.
Rewrite the bucket m and write the bucket M.
M <-- M + 1
endA1
```

(a)

(b)

(c)



(d)

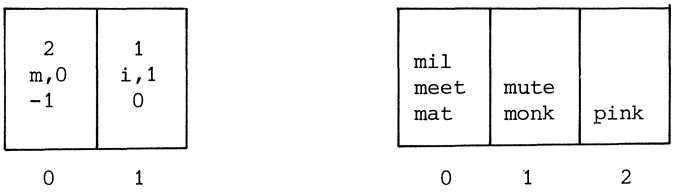


Fig. 5. Split creating more then one node. (a) the collision, (b) the file after the split, (c) the graph, (d) the file after the insertion of a record with c_0 > 'm'.

2.4. Refinement of the split algorithm

In the cases (i) and (ii) the basic schema creates more than one node. The following rule allows to create only one in the case (i):

(10) - Select the greatest key, let it be c'', leading to the selection of only one digit. Apply A1 using c'' instead of c'.

c'' always exists in the case (i) and we have c'' < c'. If we apply (10) to the case of the third split in the example file, than the selected digit is c''_0 = 'a', instead of $c'_0 c'_1$ = 'i_'. The keys moved to the bucket 2 are now: 'i','in' and 'is'.

The rule (10) may be generalized as follows:

(11) - Select the key, let it be C, leading to the minimal value of K. Apply A1 using C instead of c'.

The above rules are particularly useful for binary digits.

A split which moves half or about half of b+1 records may be called even. Otherwise, we may speak about a split which is uneven. For b = 4, a split may thus be qualified of even when 2 or 3 records change the address. The split with (o,0) is therefore even, while the split with (i,0) reveals uneven (fig. 2 and fig. 3).

If it reveals worthwile to keep the splits even, the following rule may be applied:

(12) - Select the key, let it be C', providing an even split. Apply A1 using C'.

C' may be c' or C if one of these keys provides an even split. Else, another key is chosen. In occurrence, instead of splitting using c'_0 = 'i', we would split using C'_0 = 'a'. Three keys would move : 'in', 'is' and 'of'. Thus the split would become even.

The rules (10) to (12) may be combined into more complex rules, providing compromises between the value of K and the evenness of the splits.

2.5. Key-to-address transform

Each hashing function dynamically created by trie hashing leads, for each bucket, to a maximal bound on the keys which may be inserted to this

(a) The, of, and, to, a, in, that, is, i, it, for, as, with, was, his, he, be, not, by, but, have, you, which, are, on, or, her, had, at, from, this.

(b)

are and a	to this the that	or on of not	it is in	by but be	you with which was	i	her he have had	his	at as	from for
0	1	2	3	4	5	6	7	8	9	10

(c)

a	is	i_	as	was	be	by	had	at	from
-4	2	3	-5	5	6	-7	8	9	10
o,0	i,0	_,1	a,0	t,0	h,0	f,0	e,1	r,1	b,0
-1	-2	-3	-8	1	-6	-9	7	0	4
0	1	2	3	4	5	6	7	8	9

(d)

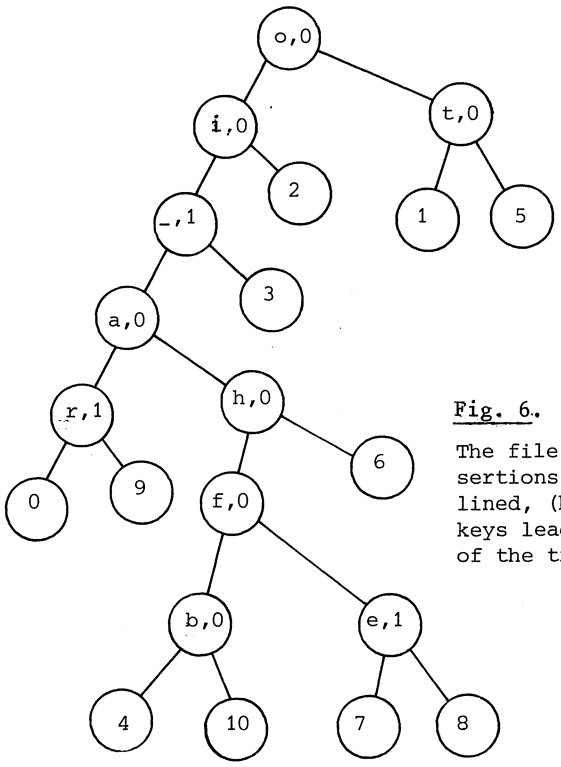


Fig. 6.

The file after 10 splits, (a) the insertions in order, with collisions underlined, (b) the buckets, (c) the trie with keys leading to each split, (d) the graph of the trie.

bucket. This bound will be called maximal key and will be denoted C. Before the the first split, C is the maximal key in the key space. This may be the maximal value resulting from the number of bytes chosen for the key representation.

After the first split, the values of C depend on the trie. If j is the number of splits supported by a bucket, then for the bucket 0 we have in occurrence (fig. 6):

```
C = 'o:::...'        j = 1
  = 'i:::...'        j = 2
  = 'i_:::...'       j = 3
  = 'a:::...'        j = 4
  = 'ar:::...'       j = 5
```

Each split decreases C of the overloaded bucket. The rules for splitting imply that c keeps the same address iff:

(13) $c =< c'_0 c'_1 \ldots c'_K :::\ldots$

(the same being valid for c'', C and C'). The rule (4) leads therefore to the following algorithm:

- Let c be the searched key. Let m be the address for c.

A2 : Trie hashing key-to-address transform

```
          C <-- ':::...:'
(1)       m <-- -0
(2)       if M = 0 : return endif
(3)       while sign(m) = -1 :
(3.1)        m <-- -m
(3.2)        i <-- DN(m)
(3.3)        C_i <-- DV(m)
(3.4)        For n = i+1,i'
                C_n <-- ':'
             endfor
(3.5)        if c =< C : m <-- LP(m) else m <-- UP(m) endif
(3.6)        if m >= 0 : return endif
             i' <-- i
```

```
endwhile
endA2
```

(1) sets up m to the value pointing at the internal node 0. (2) corresponds to the file with bucket 0 only. (3) realizes the loop which ends up when A2 finds the external pointer (sign(m) < 0 iff the number is negative). (3.1) determines the node corresponding to the internal pointer. (3.2), (3.3) and (3.4) actualize the value of C. (3.4) is needed only if DN in the current node is smaller than DN in the previous one. For instance, (3.4) is performed for bucket 0 when j = 4, since i' = 1 while i = 0. (3.5) tests (13) and so applies the principle (4). Finally, (3.6) tests if the pointer found is external.

2.6. Trie representations

2.6.1. Standard representation

We call standard representation of the trie the representation defined as follows:

(14) - The i-th node of the trie; i = 0,1,.. ; is the i-th element of a table, let it be T. Each T(i) occupies 6 bytes: 2 for each pointer, 1 for DN and 1 for DV.

Standard representation seems the most practical one. It allows to use binary, numerical or alphabetical digits. The file may extend to 32K buckets since among 64K values for pointer representation, 32K-1 values are negative and one is 'nil'. We will show in the next section, that such a file may typically contain more than 2.2 millions of records. This seems enough for most applications.

2.6.2. Refinements

Standard representation is far from the minimal representations for a given type of a file. If T is too large, for instance T cannot be entirely in core, more compact representations may be used.

On one hand, less than 2 bytes may suffice for DF:

- If the digit is a bit, DV is not necessary since we always have DV = 0. DF may therefore use only 1 byte. Furthermore, frequently the key does not exceed 16 bits. DF needs then only 4 bits. Therefore 5 or even 4.5 bytes per node suffice for such files.

- 4 bits per DV suffice for numerical digits. It is unusual for numerical keys to exceed 16 digits, 4 other bits typically suffice thus for DN. 5 bytes per node suffice thus for such files.
  On the other hand, we may optimize pointer representation:
- If the file cannot exceed 2k buckets, then k+1 bits suffice for each pointer. If a file cannot exceed 2K buckets, 3 bytes suffice for LP and UP, instead of 4. Such a file may typically contain more than 100000 records, as we will show in the next section.

Finally the trie may be represented with data sturctures other than a table. Data structure which do not use internal pointers are particularly compact. Fig. 7 shows such a representation for the trie of the example file.

To each node of the trie corresponds one node in the data structure (NDS) and vice versa. Each NDS contains DV, UP and a bit called *leaf bit* (B), equal to zero iff the node is a leaf. The value of DM is called *node level*. All NDSs of level i; i = 0,1,.. ; precede in the data structure all NDSs of level i+1. The NDSs of the same level are ordered as follows:

- For the level i = 0, the order of the NDSs respects the order of DVs.

In occurrence, the order of the NDSs corresponds to the alphabetical order on the selected c'_0s (fig. 7a).

- Let c'_0 be DV of j-th node of level 0; j = 0,1,.. . All c'_1 such that $c'_0c'_1$ was used for a split, compose a sublevel. C'_0 is the root of this sublevel. Two sublevels are consecutive iff their roots are consecutive. Within a sublevel, the order of NDSs respects the order of DVs.
- For each c'_1, all c'_2 such that $c'_1c'_2$ was used for a split, form a sublevel within the level 2. Again, (i) within each sublevel, the order of NDSs follows the order of DVs, (ii) two sublevels are consecutive iff their roots are consecutive.
- etc.

For each sublevel and for the level 0 there is also a counter of the number of nodes (NC).

This representation leads to node sizes as follows:

- For alphabetic digits, we may choose 1 byte for DV and 2 bytes for UP and the leaf bit. A node is then represented using 3 bytes and the file may grow up to 32K buckets.

- For numerical digits, we may choose 4 bits for DV and 11 bits for UP. We thus use 2 bytes per NDS and the file may grow up to 2048 buckets.
- For binary digits, there is no DV and the count occupies 1 bit (fig. 8). 32K bucket file needs only 2 bytes per node, i.e. three times less than for the standard representation.

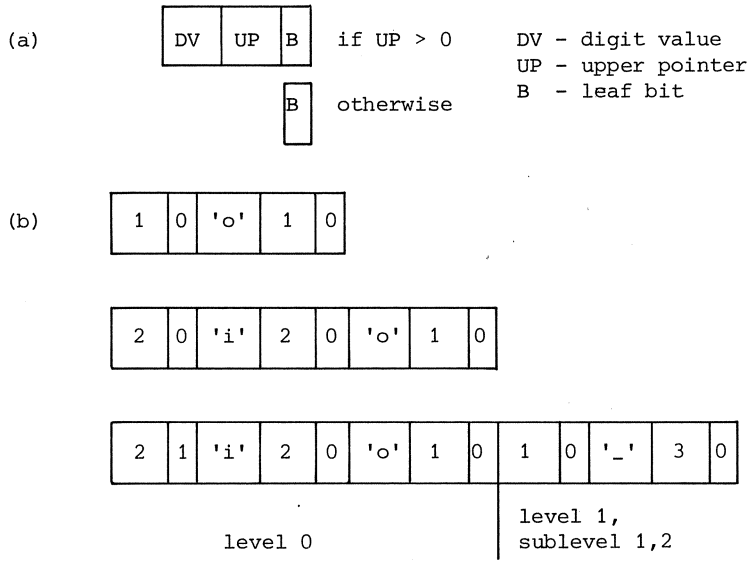


Fig. 7. Sequential representation of the trie. (a) the node (NDS), (b) data structure after 1-st, 2-nd and 3-rd split (see fig. 2,3,4).

This last conclusion means that 2 bytes per node may suffice for a 32K bucket file also for alphabetic or numerical digits. The reason is that the usual codes of these digits respect the order of binary numbers.

The above representations will be called *sequential representations*. They need less storage than the standard representation, but long bit strings must be moved when a new NDS is created. The equivalents of A1 and of A2 are trivial.

The sequential representations seem very close to minimal representation for each type of digits. In particular for binary digits we store almost only the addresses of the buckets which undergo splits and 1 bit per such a bucket for the order. Disordered virtual hashings which share all principles of the trie hashing except the order must store at least addresses of such buckets (FAGIN, NIEVERGELT, PIPPENGER & STRONG [6], LARSON [12], LITWIN [15]). Thus the order costs us one bit per split and, given the principle of splitting, it does not seem that one may use less.

3. PERFORMANCE

3.1. Load factor

Each digit may usually take several values. The split following (3.1) or (10) to (12), is thus typically even. The load factor of the trie hashing is therefore the one of a B-tree, constituted under the same assumptions. If, as typically, key values are chosen randomly, the load factor is thus 70% (COMER [4]). If keys present for storage in ascending order, the load factor is 50%.

3.2. Trie representation size

For the numerical and alphabetical digits, the case of more than one node per split is obviously exceptional. If x is the number of records in the file, we thus typically have:

$$M' = x/(0.7b).$$

B-trees use large values of b, b = 100 is a typical one. Standard representation leads therefore to following performance:

- Let M'' be the byte size of a representation. For the standard representation we thus have M'' = 6M'. Therefore, in particular:

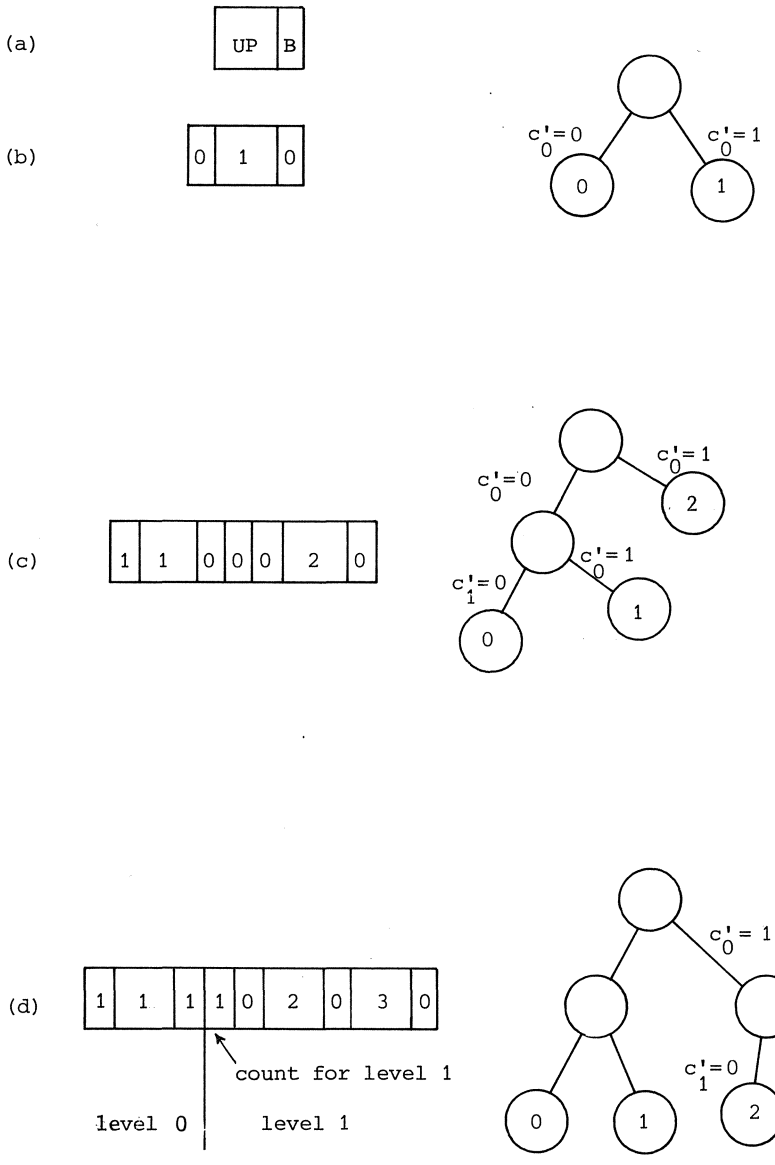


Fig. 8. Sequential representation for binary digits
(a) the node, (b) data structure after 1-st split,
(c) after 2-nd split, if $c'_0=0$,
(d) after 3-nd split, if $c'_0=1$.

- M'' = 3K allows the file to grow up from 1 to 512 buckets. Such a file may typically suffice for 35000 records.
- M'' = 12K allows files of 2048 buckets, i.e. files which may typically suffice for more than 140000 records.
- M'' = 64K allows files with about 760000 records.

Files of about 800000 records characterize rather huge applications which thus use larger computers. It is rather unlikely that an application on a small computer exceeds 35000 records per file. The size of the core of a larger computer exceeds typically Mbytes. On micros the core size is typically 32 - 64 Kbytes. For most of files, T may thus be entirely in core.

The refinements to the standard representation improve the performance. They lead in particualr to the following values:

- For binary digits, 3 byte node allows 512 buckets. The file of 35000 needs then 1.5 Kbytes, instead of 3 Kbytes.
- For numerical digits, 2K buckets file may need nodes of 4 bytes. 8 Kbytes suffice now for 140000 records, instead of 12 Kbytes.

The size of the representation decreases further when sequential representations are used. Since the storage for NCs is neglectable with respect to that for NDSs, the trie needs two or three times less storage. Thus, in particular, the following performance result:

- 64 Kbytes suffice for the file of 1500000 to 2200000 records,
- 3.5 Kbytes suffice for 140000 records.

3.3. Access performance

The above data mean that the trie may typically be entirely in core. Therefore, key-to-address transform typically does not cost disk accesses. On the other hand, there is no overflow records. This means that any successful or unsuccessful search for a record in a typical file, is performed to one disk access.

The insertion cost is in these conditions:

- two accesses when there is no a collision,
- four accesses otherwise.

The number of splits is M. The average cost of an insertion is then:

$$I = (2(x - M) + 4M)/x = 2 + 2/(0.7b) = 2 + 2.86/b, \tag{18}$$

since x = 0.7bM. The value of I is thus 2.29 accesses for b = 10 and 2.03 accesses for b = 100. Thus, practically, average insertion cost of trie hashing is 2 accesses.

3.4. Comparison to other algorithms

3.4.1. B-tree

The average number of accesses per search in a basic B-tree is $\lceil \log_b x \rceil$, where $\lceil\ \rceil$ means the smallest integer greater than or equal to the value inside. For the discussed files, we have 3 - 4 accesses. Thus a search with trie hashing is typically three to four times faster than a search in a basic B-tree.

Insertion cost for a basic B-tree is therefore also higher than the one of trie hashing. For the discussed files, the average cost is 4 - 5 accesses instead of 2. The maximal cost is 10 - 11 accesses (since, in a B-tree, all nodes on the access path may need a split), instead of 4 accesses.

Among numerous B-tree variants, some are faster than the basic one. Probably the fastest one is the variant called prefix B-tree by BAYER & UNTERAUER [3] and prefix B^+-tree by COMER [4]. The basic B^+-tree leads to access costs higher than these of the basic B-tree since all keys are in leaves. However, prefix B-trees require 20-27% fewer accesses than basic B^+-trees, for the discussed sizes of files [3]. We thus still need 2-3 accesses per average search.

3.4.2. Other algorithms for hashing

The average successful and unsuccessful search with hashing typically need more than 1 access. The difference may be neglectible, but means that trie hashing is typically faster than the known algorithm for hashing. In addition, it provides the advantages of the ordered file.

However, it should not be forgotten that this is true as long as the trie is in core. If the core is small or the file is very large and the order is not needed, usual hashing seems preferable. Linear hashing in particular, provides then better access performance and higher load factor.

4. CONCLUSIONS

Trie hashing is a new algorithm for hashing, mainly devoted to dynamic files. Contrary to the usual hashing, trie hashing keeps the file ordered. If keys are random, the load factor is about 70%. Any search is performed in one disk access for files attaining millions of records. These performance actually make trie hashing probably the best and at least the fastest technique for the ordered files. Consequently they make hashing the best technique not only for the disordered files, but for the ordered ones as well.

REFERENCES

[1] AHO, A.V., J.E. HOPCROFT & J.D. ULLMAN, *The design and Analysis of Computer Algorithms*. Addison-Wesley, Reading Mass., 1975.

[2] BAYER, R. & E. McCREIGHT, *Organization and maintenance of large ordered indexes*. Acta Informatica 1, 3 (1972), 173-189.

[3] BAYER, R. & K. UNTERAUER, *Prefix* B-*trees*. ACM Transactions on Database Systems, Vol. 2, 1, (March 1977), 11-26.

[4] COMER, D. *The ubiquitous* B-*tree*. ACM Comp. Surv., 11, 2, (June 1979), 121-138.

[5] DEUTCHER, R.F., *Distribution dependent hashing functions and their characteristics*.

[6] FAGIN, R., J. NIEVERGELT, N. PIPPENGER & H.R. STRONG, *Extendible hashing - a fast access method for dynamic files*. ACM Transactions on Database Systems Vol. 4, 3, (Sep, 1979), 315-344.

[7] GHOST, S.P. & V.Y. LUM, *Analysis of collisions when hashing by division*. Information Systems, 1-B (1975), 15-22.

[8] GUIHO, G., *Sur l'etude de collisions dans les methodes de hash-coding*, CRAS 274 (Feb 14, 1972).

[9] GUIHO, G. *Organisation des memoires, Influence d'une structure et etude d'une optimisation*. These de Doctorat d'Etat. Univ. Paris VI, (Jun 1973), 278.

[10] KNOTT, G.D., *Expandable open addressing hash table storage and retrieval*. SIGFIDET Workshop on Data Description, Access and Control, ACM (1971) 186-206.

[11] KNUTH, D.E., *The art of Computer Programming,* Vol. 3, Addison-Wesley, Reading Mass. 1974.

[12] LARSON, P., *Dynamic hashing.* BIT 18 (1978), 184-201.

[13] LARSON, P., *Linear hashing with partial expansions.* Procs 6-th Conf. on Very Large Databases, Montreal (Oct 1980).

[14] LITWIN, W., *Auto-structuration d'un fichier: methodologie, organisation d'acces, extension du hash-coding.* Res. Rep 77/11, Institut de Programmation, Paris, (Avr 1977), 102.

[15] LITWIN, W., *Une nouvelle methode d'acces per codage decoupe a un fichier.* Compte-rendus de l'Academie des Sciences, Paris, t. 286, (Avr 1978), 695,698.

[16] LITWIN, W., *Virtual hashing: a dynamically changing hashing.* Procs 4-th Conf. on Very Large Databases, Berlin (Sep 1978), 517-523.

[17] LITWIN, W., *Hachage Virtuel: une nouvelle technique d'adressage de de memoires.* These de Doctorat d'Etat. Univ. Paris VI (Mar 1979) 248.

[18] LITWIN, W., *Linear Hashing: a new tool for file and table addressing.* Procs 6-th Conf. on Very Large Data Bases, Montreal (Oct 1980).

[19] MALY, K. & L. KAMPA, H-*trees.* IFIP-80, Tokyo (Oct 1980), 445-450.

[20] SPRUGNOLI, R., *Perfect hashing functions: a single probe retrieving method for static sets.* Com ACM 20, 11 (Nov 1977), 841-850.

NAAR EEN UNIEKE DATABASESTRUCTUUR

F. REMMEN
Technische Hogeschool Eindhoven

1. INLEIDING

Sinds het onstaan van het relationele model is regelmatig de vraag opgeworpen of een zodanige formele aanpak kan worden gevolgd dat deze leidt tot een optimale logische gegevensstructuur voor elk informatiesysteem. Optimaal dan te verstaan in de zin van: minimaal aantal relaties , onder voorwaarde dat voldaan is aan bepaalde eisen. Een van deze eisen is dan derde normaalvorm (3NF) voor elke relatie. Het is een bekend feit dat deze eis van 3NF in ieder geval niet voldoende is om een optimale logische structuur in juist vermelde zin te garanderen. Dit probleem vindt men o.a. besproken bij Falkenberg en Bernstein. De eerste ziet geen mogelijkheid voor optimalisering, (FALK [7]), de tweede wel (BERNSTEIN [1]). Als het dan al mogelijk is om tot een optimale structuur te komen, dan is de volgende vraag of deze optimale structuur uniek is. Bij mijn weten is deze vraag, tenminste in de literatuur, niet of nauwelijks aan bod gekomen. Toch lijkt mij een bevestigend antwoord interessant genoeg, om te onderzoeken of dit mogelijk is. Immers, in dit geval van een unieke structuur, zal bij verschil in structuur kunnen worden geconcludeerd tot verschil in betekenis van de gegevens.

Wij zullen zien dat deze uniciteit kan worden bereikt onder bepaalde voorwaarden. Hierbij is belangrijk op te merken dat deze voorwaarden alle te maken hebben met de betekenis van de gegevens, en dus niet zomaar worden ingevoerd. In de loop der jaren zijn diverse modellen voorgesteld die ieder voor zich een ander uitgangspunt vormen voor de ontwikkeling van het formalisme ter vastlegging van gegevensstructuren. Zo zijn te noemen, het object-role model van Falkenberg, het binaire model van(o.a.)Brachi, het entity-set model van Senko. In deze eerste paragraaf zullen wij de voor ons geldende uitgangspunten weergeven. Hierbij spelen de begrippen objectsoort en kenmerk een centrale rol. Om misverstand te voorkomen zij opgemerkt dat

deze begrippen niet behoren tot het formele gedeelte, dat in verdere paragrafen zal worden ontwikkeld. Wij zullen genoemde begrippen aan de hand van voorbeelden toelichten. Zo kunnen wij spreken over objectsoorten, die relevant zijn voor een onderwijsorganisatie, bijv. de objectsoort "student" en de objectsoort "vak". Relevante kenmerken van de objectsoort "student" zijn bijv. rnr(registratienummer), nm(naam), adr(adres), wpl(woonplaats), gbd(geboortedatum). Een individueel object van het objectsoort student is bijv. de student met registratienummer 115026. Zo kunnen er bij elke objectsoort vele individuele objecten voorkomen. In plaats van de benaming "individueel object" wordt ook veel gebruikt "object-occurrence" of kortweg "occurrence". In plaats van "objectsoort" wordt dikwijls de benaming" "object-type" gebruikt. Deze laatste benaming zullen wij vermijden, omdat wij het begrip type in een andere (formele) betekenis zullen gebruiken. Eigenlijk is "student met registratienummer 115026" geen object-occurrence, maar de representatie van een object-occurrence. De occurrence zelf is de betreffende student "van vlees en bloed".

Wij zijn nu toe aan het formuleren van twee uitgangspunten. Het eerste is: informatie over een individueel object van een objectsoort O is uitsluitend mogelijk via waarden van bij O behorende kenmerken.
Dit uitgangspunt zien wij duidelijk terug in de wijze waarop informatiebehoefte wordt weergegeven, nl. met uitdrukkingen in de vorm van <kenmerk> van <de representatie van een object-occurrence>.
Voorbeelden van dergelijke uitdrukkingen zijn:
- naam van de student met registratienummer 115026;
- naam van elke student die voor 1960 is geboren.

Min of meer samenhangend met het eerste is ons tweede uitgangspunt: Een objectsoort kan nooit optreden als kenmerk en een kenmerk nooit als objectsoort. Dit is in lijnrechte tegenspraak met hetgeen o.a. Smith en Smith hierover beweren middels het door hun ingevoerde begrip objectrelativiteit (SMITH & SMITH [9], pg. 42). In hun opvatting moet nl. de rol van een objectsoort nogal relatief "afhankelijk van de omstandigheden", worden gezien. Zo kan, naar hun mening, een objectsoort ook eventueel als kenmerk optreden en omgekeerd. Ik meen echter dat deze opvatting en dus het (op zich aardig klinkend) begrip "objectrelativiteit" een goede formele behandeling van gegevensstructuren ernstig bemoeilijkt. Dit begrip is m.i. te danken, of liever te wijten, aan het feit dat in de natuurlijke taal toegestane maar onnauwkeurige benamingen zonder meer worden overgenomen in de formele beschrijving van gegevensstructuren. Een voorbeeld moge een en

ander verduidelijken.
Stel gegeven is de objectsoort "auto" met de volgende kenmerken (tussen haakjes): auto(kenteken, bouwjaar, kleur, eigenaar). Het is zeer wel denkbaar dat in de organisatie, waarin "auto" een relevante objectsoort is, ditzelfde ook geldt voor de objectsoort "eigenaar", waarvan hierbij enkele kenmerken worden gegeven:
eigenaar(naam, adres, woonplaats, telefoonnummer). Ogenschijnlijk treedt eigenaar nu op als kenmerk maar ook als objectsoort. In feite zal men echter met het kenmerk eigenaar bij de objectsoort auto een kenmerk bedoelen zoals "naam van de eigenaar". Dit zal ook duidelijk blijken als men een waarde van het kenmerk eigenaar wil representeren.

In het voorgaande zit tevens opgesloten, dat informatie over verband tussen objectsoorten alleen mogelijk is via kenmerken.

Na bovenstaande "beginselverklaringen" kunnen wij nu een begin maken met de formele opzet voor de representatie van objectsoorten en kenmerken.

In paragraaf 2 zullen enkele wiskundige basisbegrippen aan bod komen. Dan kunnen in paragraaf 3 de begrippen objectkarakterisering, objectvariabele en objectgroepvariabele worden geintroduceerd.

In paragraaf 4 wordt het afhankelijkheidsbegrip, met zijn verschillende verfijningen behandeld.

Normalisatie is het onderwerp van paragraaf 5.

In paragraaf 6 worden de begrippen database, databasesoort, databasevariabele en externe sleutel ingevoerd.

Het begrip afleidbaarheid wordt geintroduceerd in paragraaf 7.

Tot slot wordt in paragraaf 8 geschetst, hoe, onder bepaalde voorwaarden, een unieke representatie van de gegevensstructuur ontstaat.

2. ENKELE WISKUNDIGE BASISBEGRIPPEN

begrip verzameling, evanals de operaties vereniging, doorsnede en verschil (notatie $\cup$, $\cap$ en $\setminus$) kunnen wij gevoeglijk bekend veronderstellen.

Voor onze opzet is verder van essentiëel belang het begrip machtsverzameling (powerset).

Definitie 2.1: De machtsverzameling van een verzameling V is de verzameling van alle deelverzamelingen van V (inclusief de lege verzameling ϕ en V zelf). □

Notatie P(V). Uit deze definitie volgt:

$$X \in P(V) \Leftrightarrow X \subset V$$

Wij zijn nu toe aan het bekende begrip afbeelding.

Definitie 2.2: Een afbeelding f van een verzameling V in een verzameling W is een deelverzameling van

$$\{(v,w) \mid v \in V \wedge w \in W\}$$

met de eigenschap

$$\forall_{v \in V} \exists!_{w \in W} [(v,w) \in f] \ . \qquad \square$$

notatie: $f: V \to W$

Uit de definitie volgt dat een afbeelding een verzameling is, en wel een verzameling van paren. We zeggen ook wel dat f een functie is op V met waarden in W. De verzameling V noemen we het domein van f (notatie: dom(f)). Als $(v,w) \in f$, dan heet w het beeld van v onder f; notatie: $w = f(v)$. Als $V_1 \subset V$, dan heet de verzameling $W_1 = \{f(v) \mid v \in V_1\}$ het beeld van V_1 onder f; notatie: $W_1 = f[V_1]$. Opmerking: wij gebruiken vierkante resp. ronde haken voor het beeld van een deelverzameling resp. element van het domein.

Het beeld van V wordt ook wel genoemd de bij f behorende waardenverzameling.

$f: V \to W$ heet een afbeelding op W, als $f[V] = W$.

Een bijzondere afbeelding is die, waarbij elk beeld een verzameling is, zo'n afbeelding noemen wij een verzamelingsafbeelding (set function).

Voorbeeld 2.1:

$$F = \{(a,\{1,2,3,4\}) , (b,\{x,y,z\}) , (c,\{1,\ell,x,f\}\}.$$

In dit voorbeeld is dom(F) = {a,b,c}, F(a) = {1,2,3,4} en
F[{a,c}] = {F(a) , F(c)} = {{1,2,3,4} , {1,ℓ,x,f}} . □

Nu kunnen wij tenslotte het begrip productverzameling definiëren.

Definitie 2.3: De productverzameling van een verzamelingsafbeelding F van V in W is de verzameling

$\{f \mid f \text{ is een afbeelding van } V \text{ in } \bigcup_{w\in W} w \wedge \forall_{v\in V}[f(v) \in F(v)]\}$.

notatie: PROD(F). □

Voorbeeld 2.2: De productverzameling van F uit voorbeeld 2.1 bestaat uit 48 elementen. Een van deze elementen is bijv.:

$$\{(a,1)(b,y),(c,x)\} .$$ □

Een deelverzameling van een productverzameling kan geschikt worden weergegeven met behulp van een zogeheten tabel. Voorbeeld 2.3 bevat een tabel voor een deelverzameling van de productverzameling van F uit voorbeeld 2.1.

Voorbeeld 2.3: Deelverzameling van PROD(F) in tabelvorm.

a	b	c
1	y	x
1	z	x
4	x	x
3	z	ℓ
1	z	1

□

Wil men een afbeelding "beperken" tot een deelverzameling van het domein dan is de volgende definitie van belang.

Definitie 2.4: Is f een afbeelding van V in W en $X \subset V$, dan heet de verzameling

$$\{(v,w) \mid (v,w) \in f \wedge v \in X\}$$

de restrictie van f op X.
Notatie: $f|_X$. □

En voor een soortgelijke "beperking" voor elk element van een deelverzameling van een productverzameling:

Definitie 2.5: Is F een verzamelingsafbeelding van V in W en $X \subset V$, $Y \subset PROD(F)$, dan heet de verzameling

$\{f|_X \mid f \in Y\}$ de projectie van Y op X.
Notatie: $Y\|_X$. □

Voorbeeld 2.4: Zij F de verzamelingsafbeelding van voorbeeld 2.1, f het element van voorbeeld 2.2 en D de deelverzameling van PROD(F) in voorbeeld 2.3.
Dan is

$$f|_{\{a,b\}} = \{(a,1),(b,y)\} .$$

$$f|_{\{a\}} = \{(a,1)\} .$$

$$D_{\|\{a,b\}} = \Big\{\{(a,1),(b,y)\},\{(a,1),(b,z)\},$$
$$\{(a,4),(b,x)\},\{(a,3),(b,z)\}\Big\} .$$

$$D_{\|\{a\}} = \Big\{\{(a,1)\},\{(a,4)\},\{(a,3)\}\Big\} .$$

3. OBJECT, OBJECTKARAKTERISERING, OBJECTVARIABELE, OBJECTGROEPVARIABELE.

Wij beginnen in het formele vlak met het (ongedefiniëerde) begrip object. Bij elk object hoort een verzamelingsfunctie

Definitie 3.1: De bij een object X behorende verzamelingsfunctie KX heet de objectkarakterisering van X. Elk element van het domein van KX heet een

attribuut van X en het beeld van een attribuut heet het type van dit attribuut. Elk element van de bijbehorende produktverzameling PROD(KX) heet een tupel. □

In onderstaand lijstje is weergegeven hoe men, op voor de hand liggende wijze, bovengenoemde begrippen kan zien als formele representanten van in paragraaf 1 genoemde niet-formele begrippen.

formeel	representant van	niet-formeel
object		objectsoort
attribuut		kenmerk
type van een attribuut		verzameling mogelijke waarden van een kenmerk
tupel		object-occurrence

Voorbeeld 3.1: Van het object "art" luidt de karakterisering "K-art" als volgt:

$$\text{K-art} = \{(\text{ano}, \{1...100\}), (\text{anm}, \text{charstring } 20), (\text{pr}, \{10...500\}), (\text{gew}, \{1...30\}), (\text{kl}, \{\text{rd,w,bl}\})\} .$$

Hierbij is charstring 20 het type dat bestaat uit de verzameling van alle combinaties van maximaal 20 letters, cijfers of andere toegestane tekens. □

Het domein van K-art bestaat dus uit de attributenverzameling {ano,anm,pr, gew,kl}. Het type van het attribuut kleur bestaat uit de verzameling {rd, w,bl}.

Om de betekenis van het object "art" en zijn attributen weer te geven, moge onderstaand lijstje dienen. Hierbij dient betekenis te worden opgevat in de zin van namen van objectsoorten en kenmerken.

formeel		niet-formeel (betekenis)	
object:	art	objectsoort	: artikel
attributen:	ano	kenmerken	: artikelnummer

anm	artikelnaam
pr	prijs per stuk
gew	gewicht per stuk
kl	kleur

Opmerking: In de database-literatuur wordt het woord domein gebruikt in de betekenis van waardenverzameling, zoals hierboven met type wordt bedoeld. Zie bijv. (DATE [5], pg. 73). Dit kan uiteraard erg verwarrend zijn. Wij zullen het woord domein dan ook uitsluitend gebruiken in de (wiskundige) betekenis van af te beelden verzameling. □

Er volgen nu nog twee andere voorbeelden van objecten met bijbehorende karakteriseringen. Tussen haakjes wordt hierbij de betekenis vermeld.

Voorbeeld 3.2: Gegeven is het object lev(leverancier) met de attributen lnr(nummer), lnm(naam), ladr(adres), lwpl(woonplaats), ntnr(netnummer), abnr(abonneenummer).
De bijbehorende karakterisering is:

$$K\text{-lev} = \{(lnr, \{1...100\}), (lnm, \text{charstring } 25), (ladr, \text{charstring } 30), (lwpl, \text{charstring } 20), (ntnr, \{10...99\}), (abnr, \{1...10000000\})\}.$$ □

Voorbeeld 3.3: Gegeven is het object order, met de attributen onr(ordernummer), dat(datum), lnr(leveranciernummer), anr(artikelnummer), lnm(leveranciersnaam), wpl(leverancierswoonplaats), arnm(artikelnaam), hoev(aantal stuks), pr(prijs per stuk). De bijbehorende karakterisering is:

$$K\text{-order} = \{(onr, \{1...10000\}), (dat, \{700000...990000\}), (lnr, \{1...100\}), (anr, \{1...100\}), (lnm, \text{charstring } 25), (lwpl, \text{charstring } 20), (anm, \text{charstring } 20), (hoev, \{1...100000\}), (pr, \{10...500\})\}.$$ □

Tenslotte volgt nog een (klein) voorbeeld, waarvan de produktverzameling van de objectkarakterisering ook zal worden weergegeven.

Voorbeeld 3.4: Gegeven is het object meubel met de attributen mnr(meubelnummer), nm(naam) en kl(kleur).
De bijbehorende karakterisering is:

$$K\text{-meubel} = \{(mnr, \{1,2,3\}), (nm, \{tf,st\}), (kl, \{rd,w\})\} .$$

De produktverzameling van K-meubel bestaat uit 12 tupels, zoals hieronder (verkort) weergegeven in de vorm van een tabel. Men bedenke hierbij echter dat elk tupel een afbeelding is van de attributenverzameling in de verzameling typen, zodanig dat het beeld van attribuut a een element is van het type van a. (zie definitie 2.4). Het eerste tupel uit onderstaande tabel is dus (volledig uitgeschreven) gelijk aan: {(mnr,1),(nm,tf), (kl,rd)}.

PROD (K-meubel) =

mnr	nm	kl
1	tf	rd
1	tf	w
1	st	rd
1	st	w
2	tf	rd
2	tf	w
2	st	rd
2	st	w
3	tf	rd
3	tf	w
3	st	rd
3	st	w

□

Wij zullen nu twee speciale soorten variabelen invoeren, nl. objectvariabelen en objectgroepvariabelen.

Wij zullen nu eerst de objectvariabelen behandelen. Bij elk object kan een objectvariabele worden gedeclareerd. De verzameling van mogelijke waarden

van een objectvariabele VX heet het type van VX. Elke waarde van een objectvariabele is een element van de bij het object behorende produktverzameling. Omgekeerd hoeft niet te gelden, en zal in het algemeen ook niet gelden, dat elk element van genoemde produktverzameling als waarde van een objectvariabele mag optreden. Vanwege zogeheten integrity constraints zullen bepaalde tupels niet als waarde van een objectvariabele kunnen optreden. Als bijvoorbeeld (in voorbeeld 3.4), de combinatie "(nm,tf), (kl,rd)" niet is toegestaan, dan kan slechts driekwart van het aantal tupels als waarde van een bij "meubel" behorende objectvariabele optreden.
Voor het type van een objectvariabele geldt dus:
Het type van een objectvariabele VX, behorende bij een object X met karakterisering KX, is een deelverzameling van PROD(KX).

Wij hebben reeds gezien dat een tupel, dus ook (de waarde van) een objectvariabele, kan worden opgevat als de representatie van één object-occurrence ce. Op een bepaald moment zullen van een object-soort echter een aantal object-occurrences bestaan (actueel zijn). Deze occurrences worden door evenzovele tupels uit een produktverzameling gerepresenteerd. Deze verzameling van occurrences en dus ook de verzameling van de overeenkomstige tupels kan in de loop van de tijd veranderen. In de database - literatuur vindt men dit terug onder de kwalificatie "time-varying", zonder hiervoor echter een formeel beschrijvingsgereedschap aan te bieden. Zie bijv. [DATE, ch. 4].

Wij zullen nu, door gebruik te maken van het wiskundige begrip machtsverzameling, het wel mogelijk maken dit "time-varying" aspect in de formele beschrijving op te nemen. Tevens is dan nu de tijd gekomen om de tweede soort variabelen te bespreken.

Bij elk object X kan een objectgroepvariabele worden gedeclareerd. De verzameling mogelijke waarden van een objectgroepvariabele GVX heet het type van GVX. Elke waarde van een objectgroepvariabele is gelijk aan een verzameling tupels, dus een verzameling elementen van het type van een objectvariabele behorende bij object X. Elke waarde van een objectgroepvariabele is dus in feite gelijk aan een element van de machtsverzameling van de bij object X behorende produktverzameling. Omgekeerd hoeft niet te gelden, en zal in het algemeen ook hier weer niet gelden, dat elk element van genoemde machtsverzameling als waarde van een objectgroepvariabele kan optreden. Vanwege allerlei integrity constraints zullen vele elementen van genoemde

machtsverzameling niet als waarde toegelaten zijn (BROCK [2]). Als bijvoorbeeld (zie voorbeeld 3.4) van de machtsverzameling van PROD(K-meubel), alleen die elementen zijn toegestaan waarvoor geldt:

- er zijn geen tweetal tupels met dezelfde waarde voor mnr.
- van nm kan hoogstens tweemaal de waarde "tf" èn hoogstens tweemaal de waarde "st" voorkomen

dan is op een totaal van 2^{12} = 4096 elementen slechts een aantal van 61 toegestaan als mogelijke waarde van een objectgroepvariabele.

Voor het type van een objectgroepvariabele geldt dus:
Het type van een objectgroepvariabele GVX, behorende bij een object X met karakterisering KX is een deelverzameling van de machtsverzameling van PROD(KX).

Onder de karakterisering en attributenverzameling van een objectvariabele resp. objectgroepvariabele zullen wij verstaan de karakterisering en attributenverzameling van het object, waarbij de objectvariabele resp. obgroepvariabele behoort.

4. FUNCTIONELE AFHANKELIJKHEID; SLEUTEL

In het voorgaande is meermalen gesproken over de bij een object behorende attributen. De bedoeling is om het begrip "behorende bij een object" een formele inhoud te geven, zodanig dat binnen het kader van een bepaald informatiesysteem voor elk attribuut ondubbelzinnig vaststaat bij welk object het behoort. Het is de grote verdienste van de ontwerpers van het relationele model, dat zij voor het eerst dit begrip expliciet en systematisch behandeld hebben via de zogeheten normalisatiestappen. Over normalisatie is sinds het eerste artikel van Codd (CODD [3]) een lawine van literatuur verschenen (MOHAN [8]). Hetgeen wij nu verder hier te berde brengen, is dan ook zeker niet in alle opzichten nieuw te noemen. Zoals, hopelijk, uit het voorgaande reeds is gebleken en uit het volgende nog zal blijken, hebben wij getracht het formele gereedschap een betere basis te geven, en daarbij de betekenis van de gegevens niet uit het oog te verliezen. In vele artikelen van de laatste jaren wordt (m.i. terecht) de betekenis van de gegevens (meaning of data) veel sterker dan voorheen benadrukt. Zo bijv. ook door Codd in zijn artikel van eind 1979 (CODD [4]).

Door deze betekenis steeds als uitgangspunt te nemen en te trachten voor de representatie daarvan een adekwaat formeel systeem te vormen, meen ik

gekomen te zijn tot een betere definitie van de eerste normaal-vorm en daarmee tegelijkertijd tot het aantonen van de overbodigheid van de vierde (en hogere?) normaal-vorm. Een en ander moge uit het vervolg blijken.

Voor de rest van deze paragraaf zullen wij aannemen dat TY het type is van objectgroepvariabele Y met attributenverzameling AY en karakterisering KY.

Wij zullen nu de uit de database literatuur bekende begrippen functionele afhankelijkheid, volledige functionele afhankelijkheid en directe functionele afhankelijkheid behandelen.

Definitie 4.1: Als $A_1, A_2 \subset AY$ dan heet A_2 functioneel afhankelijk van A_1 als geldt: $\forall_{W \in TY}$

$$[\{(w|_{A_1}, w|_{A_2}) \mid w \in W\}$$

is een afbeelding van $W\|_{A_1}$ op $W\|_{A_2}$].

Notatie: $A_1 \overset{fa}{\rightarrow} A_2$. □

Opmerkingen:

a) Definitie 4.1 is mogelijk, omdat $\forall w \in W\ [w|_{A_1} \in W\|_{A_1}$ èn $w|_{A_2} \in W\|_{A_2}]$.

b) Het reeds eerder genoemde "time-varying" aspect heeft in definitie 4.1 een formele weergave gekregen in "$\forall W \in TY$ ...".

c) $A_1 \overset{fa}{\Leftrightarrow} A_2$ betekent $A_1 \overset{fa}{\rightarrow} A_2$ èn $A_2 \overset{fa}{\rightarrow} A_1$.

d) Als voor A_1, A_2 geldt $A_1 \overset{fa}{\Leftrightarrow} A_2$ dan noemen wij de attributen verzamelingen A_1 en A_2 equivalent.

Uit definitie 4.1 volgt onmiddellijk

Stelling 4.1: Als $A_1, A_2 \subset AY$ dan geldt:

$$(A_1 \overset{fa}{\rightarrow} A_2) \Leftrightarrow (\forall_{a \in A_2} [A_1 \overset{fa}{\rightarrow} \{a\}]).$$

Het bewijs van deze stelling berust voor het "⇒" gedeelte op het feit dat $w|_{\{a\}} = (w|_{A_2})|_{\{a\}}$ voor elke $a \in A_2$ en voor het "⇐" gedeelte op het feit dat

$$w|_{A_2} = \bigcup_{a \in A_2} w|_{\{a\}} .$$

Funktionele afhankelijkheid is transitief, zoals blijkt uit:

Stelling 4.2: Als $A_1, A_2, A_3 \subset AY$ dan geldt:

$$((A_1 \rightarrow A_2) \wedge (A_2 \rightarrow A_3)) \Rightarrow (A_1 \rightarrow A_3) .$$

Bewijs. Uit definitie 4.1 volgt:

(1) $$A_1 \rightarrow A_2 \Leftrightarrow W \in TY$$
$$\left[\forall_{w_1 \in W} \forall_{w_2 \in W} [(w_1|_{A_1} = w_2|_{A_1}) \Rightarrow (w_1|_{A_2} = w_2|_{A_2})]\right]$$

Stel $A_1 \overset{fa}{\rightarrow} A_2$ èn $A_2 \overset{fa}{\rightarrow} A_3$, dan volgt hieruit met (1):

(2) $$\forall_{W \in TY} \left[\forall_{w_1 \in W} \forall_{w_2 \in W}\right.$$
$$\left[\left((w_1|_{A_1} = w_2|_{A_1}) \Rightarrow (w_1|_{A_2} = w_2|_{A_2})\right)\right.$$
$$\left.\left. \wedge \left((w_1|_{A_2} = w_2|_{A_2}) \Rightarrow (w_1|_{A_3} = w_2|_{A_3})\right)\right]\right] .$$

Uit (2) volgt nu:

(3) $$\forall_{W \in TY} \left[\forall_{w_1 \in W} \forall_{w_2 \in W} \left[\left(w_1|_{A_1} = w_2|_{A_1}\right) \Rightarrow \left(w_1|_{A_3} = w_2|_{A_3}\right) .\right.\right.$$

Uit (1) en (3) volgt tenslotte: $A_1 \overset{fa}{\rightarrow} A_3$. □

Definitie 4.2: Als $A_1, A_2 \subset AY$, dan heet A_2 transitief functioneel afhankelijk van A_1 als geldt:

$$\exists_{A \subset AY} \left[(A_1 \overset{fa}{\rightarrow} A) \wedge (A \overset{fa}{\rightarrow} A_2) \wedge \neg(A_1 \overset{fa}{\Leftrightarrow} A)\right] . \qquad \square$$

Definitie 4.3: Als $A_1, A_2 \subset AY$, dan heet A_2 volledig functioneel afhankelijk van A_1 als geldt:

a) $A_1 \overset{fa}{\rightarrow} A_2$

èn

b) $\forall A \subsetneq A_1\ [\neg(A \overset{fa}{\rightarrow} A_2)]$

Notatie: $A_1 \overset{vfa}{\rightarrow} A_2$. □

Uit definitie 4.3 volgt niet dat als $A_1 \overset{vfa}{\rightarrow} A_2$, ook voor elk attribuut $a \in A_2$ moet gelden: $A_1 \overset{vfa}{\rightarrow} \{a\}$ en zelfs niet dat voor minstens een attribuut $a \in A_2$ moet gelden: $A_1 \overset{vfa}{\rightarrow} \{a\}$. Een tegenvoorbeeld voor dit laatste vinden wij in het nu volgende voorbeeld.

Voorbeeld 4.1:

$$KY = \{(a_1, \{1,2,3\}) , (a_2, \{4,5,6\}) , (a_3, \{7,8,9,10,11\}) ,$$

$$(a_4, \{p,q\} , (a_5, \{r,s,t\})\}$$

TY bestaat uit 2 elementen E_1 en E_2, hieronder weergegeven in tabelvorm

	a_1	a_2	a_3	a_4	a_5
E_1 =	1	4	7	p	s
	1	6	7	p	s
	1	4	9	p	t
	1	5	9	q	t
	3	5	9	p	r
E_2 =	2	5	8	p	s
	2	4	8	q	s
	2	5	10	p	t

Dan geldt: (1) $\{a_1,a_2,a_3\} \overset{vfa}{\rightarrow} \{a_4,a_5\}$

(2) $\{a_1,a_2\} \overset{vfa}{\rightarrow} \{a_4\}$

(3) $\{a_1,a_3\} \overset{vfa}{\rightarrow} \{a_5\}$.

en dus <u>niet</u> $\{a_1,a_2,a_3\} \overset{vfa}{\rightarrow} \{a_4\}$ (tegenspraak met (2))

en ook <u>niet</u> $\{a_1,a_2,a_3\} \overset{vfa}{\rightarrow} \{a_5\}$ (tegenspraak met (3)) □

Wel geldt de volgende stelling:

Stelling 4.3: Als $A_1, A_2 \subset AY$ dan geldt:

$$\left((A_1 \overset{fa}{\rightarrow} A_2) \wedge (\exists_{A_{21} \subset A_2} [A_1 \overset{vfa}{\rightarrow} A_{21})\right) \Rightarrow \left(A_1 \overset{vfa}{\rightarrow} A_2\right) .$$

Bewijs: Stel $\neg(A_1 \overset{vfa}{\rightarrow} A_2)$. Dan is er een $A \subsetneq A_1$ met $A \overset{fa}{\rightarrow} A_2$. Hieruit volgt (met stelling 4.1) $A \overset{fa}{\rightarrow} A_{21}$. Dus (met definitie 4.2): $\neg(A_1 \overset{vfa}{\rightarrow} A_{21})$, hetgeen in tegenspraak is met het gegeven. □

Een direct gevolg van deze stelling is:

Stelling 4.4: Als $A_1, A_{21} \subset AY$ èn $A_1 \overset{vfa}{\rightarrow} A_{21}$ dan geldt:

$$\forall_{A_2 \subset AY} [(A_2 \supset A_{21} \wedge A_1 \overset{fa}{\rightarrow} A_2) \Rightarrow (A_1 \overset{vfa}{\rightarrow} A_2)] .$$ □

Minder formeel weergegeven: Bij "uitbreiding van afhankelijkheid" blijft volledige functionele afhankelijkheid behouden.

Definitie 4.4: Als $A_1, A_2 \subset AY$ dan heet A_2 direct functioneel afhankelijk van A_1 als geldt:

a) $A_1 \overset{fa}{\rightarrow} A_2$

èn

b) $\forall A \subset AY$

$$[(A_1 \overset{fa}{\rightarrow} A \wedge A \overset{fa}{\rightarrow} A_2) \Rightarrow \left(A_1 \overset{fa}{\nrightarrow} A\right)] .$$

Notatie: $A_1 \overset{dfa}{\rightarrow} A_2$.

Uit de definities 4.2 en 4.4 volgt dat "direct functioneel afhankelijk" hetzelfde is als "niet transitief functioneel afhankelijk".

Evenals bij volledig functioneel afhankelijk geldt ook voor direct functioneel afhankelijk dat uit $A_1 \overset{dfa}{\rightarrow} A_2$ niet volgt voor minstens een $a \in A_2$: $A_1 \overset{dfa}{\rightarrow} \{a\}$. Voorbeeld 4.1 kan hier ook weer gebruikt worden als tegenvoorbeeld met dien verstande dat overal "vfa" wordt vervangen door "dfa".

Wel gelden ook hier soortgelijke stellingen voor behoud van $\overset{"dfa"}{\rightarrow}$ bij "uitbreiding".

Stelling 4.5: Als $A_1, A_2 \subset AY$, dan geldt:

$$\left((A_1 \overset{fa}{\rightarrow} A_2) \wedge (\exists_{A_{21} \subset A_2}[A_1 \overset{dfa}{\rightarrow} A_{21}])\right) \Rightarrow \left(A_1 \overset{dfa}{\Leftrightarrow} A_2\right) .$$

Bewijs: Stel $\neg(A_1 \overset{dfa}{\rightarrow} A_2)$. Dan bestaat er een $A \subset A_x$ zò dat

1) $A_1 \overset{fa}{\rightarrow} A \wedge A \overset{fa}{\rightarrow} A_2 \wedge \neg(A_1 \overset{fa}{\Leftrightarrow} A)$.

Uit het gegeven $(A_2 \supset A_{21})$ èn $A \overset{fa}{\rightarrow} A_2$ volgt:

2) $A \overset{fa}{\rightarrow} A_{21}$.

(1) en (2) geeft:

3) $A_1 \overset{fa}{\rightarrow} A \wedge A \overset{fa}{\rightarrow} A_{21} \wedge \neg(A_1 \overset{fa}{\Leftrightarrow} A)$.

Uit (3) volgt $\neg(A \overset{dfa}{\rightarrow} A_{21})$, hetgeen in strijd is met het gegeven. □

Ook hier geldt: bij "uitbreiding van afhankelijkheid" blijft directe functionele afhankelijkheid "behouden".

Formeel:

Stelling 4.6: Als $A_1, A_2 \subset AY$ èn $A_1 \overset{dfa}{\rightarrow} A_2$ dan geldt:

$$\forall_{A \subset AY}[(A \supset A_2 \wedge A_1 \overset{fa}{\rightarrow} A) \Rightarrow (A_1 \overset{dfa}{\rightarrow} A)] . \qquad \square$$

Directe functionele ahhankelijkheid sluit niet zonder meer in volledig functionele afhankelijkheid. Wel echter geldt:

Stelling 4.7: Als $A_1, A_2 \subset AY$ dan geldt:

$$\left((A_1 \overset{dfa}{\rightarrow} A_2) \wedge \forall_{A \subset A_1}[(A_1 \overset{fa}{\Leftrightarrow} A) \Rightarrow (A = A_1)]\right) \Rightarrow \left(A_1 \overset{vfa}{\rightarrow} A_2\right) .$$

Bewijs: Stel $\neg(A_1 \overset{vfa}{\rightarrow} A_2)$. Dan is er een $A \subset A_1$ met $A \neq A_1$, waarvoor geldt $A \overset{fa}{\rightarrow} A_2$. Aangezien $A \subset A_1$, geldt ook $A_1 \overset{fa}{\rightarrow} A$. Dus $A_1 \overset{fa}{\rightarrow} A \wedge A \overset{fa}{\rightarrow} A_2$. Tesamen met het eerste deel van het gegeven $(A_1 \overset{dfa}{\rightarrow} A_2)$ voert dit tot de conclusie: $A_1 \overset{fa}{\Leftrightarrow} A$. Dit is echter in tegenspraak met het tweede deel van het gegeven. □

In "woorden" luidt stelling 4.7: als A_2 direct functioneel afhankelijk is van A_1 èn A_1 is niet equivalent met een van zijn deelverzamelingen,

dan is A_2 ook volledig functioneel afhankelijk is van A_1.

Wij zullen nu het begrip sleutel definiëren.

Definitie 4.5: Als $A \subset AY$ dan heet A een sleutel van objectgroepvariabele Y als geldt:

$$A \overset{vfa}{\rightarrow} AY \qquad \square$$

Met de hier gegeven definitie is de verzameling attributen in een sleutel voldoende èn nodig voor de identificering van de tupels. Deze "minimale" definitie wordt tegenwoordig door verschillende auteurs aangehangen. Zie bijv.: CODD [4], pg. 400)

Er kan uiteraard bij een objectgroepvariabele meer dan een sleutel zijn. Wij spreken dan ook van de sleutelverzameling SY van een objectgroepvariabele Y.

Door genoemde "minimaliteit" in definitie 4.5 is het uitgesloten dat twee verschillende deelverzamelingen van een sleutel equivalent zijn, zoals blijkt uit:

Stelling 4.8: Als SY de sleutelverzameling is van objectgroepvariabele Y, dan geldt:

$$\forall_{S \in SY}\left[\forall_{D_1, D_2 \subset S}\left[(D_1 \overset{fa}{\Leftrightarrow} D_2) \Rightarrow (D_1 = D_2)\right]\right] .$$

Bewijs: Stel:

(1) $D_1 \overset{fa}{\Leftrightarrow} D_2$.

Stel verder $H = D_2 \backslash D_1$ resp $H = D_1 \backslash D_2$ als

$$D_2 \supset D_1 \text{ resp } \neg(D_2 \supset D_1).$$

Dan is

(2) $((D_1 \subset S \backslash H) \wedge (D_2 \cup S \backslash H = S))$ resp

$$((D_2 \subset S \backslash H) \wedge (D_1 \cup S \backslash H = S)).$$

uit (1) en (2) volgt:

(3) $S \backslash H \overset{fa}{\rightarrow} S$.

Aangezien S sleutel is, geldt:

(4) $S \overset{vfa}{\rightarrow} AY$, dus ook

(5) $S \overset{fa}{\rightarrow} AY$.

uit (3), (5) en stelling 4.2 volgt:

(6) $S\backslash H \overset{fa}{\rightarrow} AY$.

En tenslotte uit (4) en (6):

$S\backslash H = S$, dus $H = \phi$, dus $D_1 = D_2$. □

Uit stelling 4.8 volgt in het bijzonder dat geen enkele echte deelverzameling van een sleutel equivalent is met deze sleutel zelf.

Een andere zaak is de equivalentie van twee verschillende echte deelverzamelingen van twee verschillende sleutels van dezelfde objectgroepvariabele. Deze equivalentie wordt door definitie 4.5 niet uitgesloten. Een voorbeeld moge dit verduidelijken.

Voorbeeld 4.2: Stel dat bij objectgroepvariabele Y met sleutelverzameling SY onder meer de volgende attributen behoren: lnr(leveranciersnummer), anr(artikelnummer), datum, lnm(leveranciersnaam), anm(artikelnaam). Stel verder:

$$S_1 = \{lnr,anr,datum\};\ S_2 = \{lnr,anm,datum\};$$

$$S_3 = \{lnm,anr,datum\};\ S_4 = \{lnm,anm,datum\}.$$

Als nu geldt:

$$(1)\ \{lnr\} \overset{fa}{\Leftrightarrow} \{lnm\} \text{ èn } (2)\ \{anr\} \overset{fa}{\Leftrightarrow} \{anm\} \text{ èn } S_1 \in SY$$

dan geldt ook $S_i \in SY$ met $2 \leq i \leq 4$.

Hieruit volgt dan dat elk tweetal van de vier genoemde sleutels onderling equivalente echte deelverzamelingen bevat, zoals

$$\{lnr\} \subset S_1 \text{ en } \{lnm\} \subset S_3 \text{ enz.}$$ □

Bij de definitie van sleutel zou het niet zinvol zijn bovengenoemde equivalentie uit te sluiten. Dit zal wel gebeuren bij normalisatie (zie volgende paragraaf).

Het in de literatuur veel gemaakte onderscheid tussen primaire en kandidaat sleutels (DATE [5], pg. 78) is hier, als niet ter zake doende voor de

logische structuur, vervallen. Bernstein maakt dit onderscheid overigens ook niet (BERNSTEIN [1], pg. 279). Codd daarentegen houdt dit onderscheid ook nog in zijn artikel van 1979 aan (CODD [4], pg. 400). Alvorens enkele voorbeelden te geven, met betrekking tot het afhankelijkheids- en sleutelbegrip willen wij nog het volgende opmerken. Degenen, voor wie het informatiesysteem wordt opgezet, zijn de enige personen die kunnen bepalen welke afhankelijkheden gelden en welke de sleutelverzameling is van een object. Afhankelijkheden en sleutelverzamelingen horen, evenals karakteriseringen tot de invoer van het formele systeem. Wel is het uiteraard mogelijk, met gebruikmaking van het formele systeem, te "wijzen" op de konsekwenties van de gedane keuze. In formele vorm werden enkele van deze konsekwenties reeds naar voren gebracht in de stellingen 4.1 tot en met 4.4.

Voorbeeld 4.3: Uitgaande van de objectgroepvariabelen art, lev en order met karakterisering zoals in de overeenkomstige voorbeelden 3.1, 3.2 en 3.3, zouden de volgende beweringen juist kunnen zijn (veronderstellende dat een woonplaats precies één netnummer heeft).

objectgroepvariabele	beweringen betreffende afhankelijkheden/ sleutels
art	1) {ano} $\overset{fa}{\Leftrightarrow}$ {anm}.
	2) {ano} $\overset{dfa}{\rightarrow}$ {pr,gew,kl}.
	3) sleutelverzameling = {{ano}, {anm}}.
lev	4) {lnm,ladr,lwpl} $\overset{vfa}{\rightarrow}$ {ntnr,abnr}.
	5) $\neg$[{lnm,ladr,lwpl} $\overset{vfa}{\rightarrow}$ {ntnr}].
	6) sleutelverzameling = {{lnr}, {lnm,ladr,lwpl}}.
order	7) {lnr} $\overset{fa}{\rightarrow}$ {lnm}.
	8) {onr} $\overset{fa}{\rightarrow}$ {dat,lnr}.

9) $\{onr\} \overset{fa}{\rightarrow} \{anr\} \overset{fa}{\Leftrightarrow} \{anm\}$.

10) $\neg[\{onr\} \overset{dfa}{\rightarrow} \{anm\}] \wedge \neg[\{onr\} \overset{dfa}{\rightarrow} \{anr\}]$.

11) $\{\{onr\},\{lnr\ anr,dat\}\} \subset$ sleutelverzameling □

5. NORMALISATIE

Door de gekozen definities (met name 2.3,2.4 en 3.1) wordt in een tupel aan elk attribuut precies één waarde toegevoegd uit het bijbehorende type. Dit wil niet zeggen dat in het type geen waarden zouden kunnen voorkomen, die de "schijn" hebben van "meerwaardigheid". Een voorbeeld moge dit verduidelijken. Voor anm(artikelnaam) zou het volgende type denkbaar zijn T1 = {lepel,vork,mes}, maar ook T2 = {lepel,vork,mes,lepel-vork, vork-mes,lepel-vork-mes}. T2 bestaat niet uit 3, maar uit 6 waarden, en is dus niet gelijk aan T1. Zowel T_1 als T_2 kunnen worden gekozen als type voor het attribuut anm, maar als T_1 het type is, dan kunnen niet tegelijk waarden als "lepel-vork" of "vork-mes" voor het attribuut anm optreden. Of een bepaalde waarde wel of niet tot een type behoort, is geheel ter beoordeling van de gebruiker, dus van degene die verantwoordelijk is voor de betekenis van de gegevens. Met andere woorden: vastlegging van typen gebeurt niet door het formele systeem, maar behoort tot de invoer van dit systeem. (zie ook vorige paragraaf).

In de literatuur wordt 1NF (eerste normaal vorm) gedefiniëerd met behulp van het vage begirp "atomaire waarde". Zie bijv. (DATE [5], pg. 157).

In de opzet van dit artikel kan elk van de waarden van een attribuuttype optreden als attribuutwaarde en een andere waarde is niet mogelijk. Hiermee is (de karakterisering van) elke objectgroepvariabele per definitie in 1NF.

Opmerking: Met een andere definitie in plaats van 2.3 (en dan liever ook een ander woord in de plaats van afbeelding) zou uiteraard een andere opzet mogelijk xijn geweest (met bewust toegelaten "meerwaardigheid"). Men denke bijv. aan een definitie in de trant van

$$\{(v,w) \mid v \in V \text{ en } w \in W\}$$

met de eigenschap dat

$$\forall_{v \in V}[\exists_{w \in W}[(v,w) \in f]] \; . \qquad \square$$

Met bovengenoemde aanpak, waarbij voor elk attribuut precies vastligt welke waarden kunnen optreden, meen ik te mogen stellen dat de vierde normaalvorm, zoals geintroduceerd doro Fagin [FAGIN] beter niet gebruikt kan worden en in feite overbodig is: het begrip multi-valued dependency is immers niet nodig, als men van te voren in de karakterisering precies vastlegt welke waarden per attribuut kunnen worden aangenomen. Zie T1 en T2 aan het begin van deze paragraaf. Op deze wijze wordt konsekwent de verantwoordelijkheid voor de betekenis van de gegevens gelegd (en gelaten) waar deze behoort, nl. bij de gebruiker. In dit artikel zal dan ook het begrip vierde normaalvorm niet worden ingevoerd.

Wij kunnen nu overgaan tot de definities van 2^e en 3^e normaalvorm.

Definitie 5.1: Het type van een objectgroepvariabele Y met attributenverzameling AY en sleutelverzameling SY is in tweede normaalvorm (2NF) als geldt:

$$\forall_{a \in AY}\left[\forall_{S \in SY}[(a \notin S) \Rightarrow (S \overset{vfa}{\rightarrow} \{a\})]\right] \; . \qquad \square$$

In "woorden": het type van een objectgroepvariabele is in tweede normaalvorm, als elk attribuut volledig functioneel afhankelijk is van elke sleutel, waarvan het geen deel uitmaakt.

Definitie 5.2: Het type van een objectgroepvariabele met attributenverzameling AY en sleutelverzameling SY, is in derde normaalvorm (3NF) als geldt:

$$\forall_{a \in AY}\left[\forall_{S \in SY}[(a \notin S) \Rightarrow (S \overset{dfa}{\rightarrow} \{a\})]\right] \; . \qquad \square$$

In "woorden": het type van een objectgroepvariabele is in derde normaalvorm, als elk attribuut direct funktioneel afhankelijk is van elke sleutel, waarvan het geen deel uitmaakt.

In de literatuur worden veelal de toch al vage definities van normaalvormen

zo gegeven, dat bij 2NF geeist wordt 1NF en bij 3NF geeist wordt 2NF. In onze definities is dit echter niet nodig, aangezien 2NF wordt omvat door 3NF. Dus geldt:

Stelling 5.1: Als het type van een objectgroepvariabele in 3NF is, dan is het ook in 2NF.

Bewijs: het bewijs volgt onmiddellijk door toepassing van stelling 4.7 (met $A_1 = S$ èn $A_2 = \{a\}$), waarbij dan met stelling 4.8 kan worden geconstateerd dat S aan de "niet-equivalente" voorwaarde voldoet. □

Op grond van stelling 5.1 kunnen we voor de "normalisatie tot en met derde normaalvorm" dus eigenlijk volstaan met èèn definitie (zoals gegeven in definitie 5.2). Om geen onnodige verwarring te scheppen, zullen wij toch de naamgeving 3NF blijven handhaven.

Wij kunnen nu stelling 4.8 uitbreiden en komen daarbij terug op de equivalentie van twee verschillende deelverzamelingen van twee verschillende sleutels (zie voorbeeld 4.2). Als namelijk het type van een objectgroepvariabele in 3NF is, dan kan een deelverzameling D_1 van een sleutel S_1 niet equivalent zijn met een van D_1 verschillende deelverzameling D_2 van een andere sleutel S_2, zoals blijkt uit:

Stelling 5.2: Gegeven is objectgroepvariabele Y met type TY en sleutelverzameling SY. Als TY in 3NF is, dan geldt:

$$\forall_{S_1,S_2 \in SY}\left[\forall_{D_1 \subset S_1} \forall_{D_2 \subseteq S_2} [(D_1 \overset{fa}{\Leftrightarrow} D_2) \Rightarrow (D_1 = D_2)]\right] .$$

Bewijs: Voor $S_1 = S_2$ is stelling 5.2 gelijk aan stelling 4.8. Verder onderscheiden wij de gevallen:

$D_1 \subset S_2$ en $D_1 \not\subset S_2$.

a) $D_1 \subset S_2$: dit is een speciaal geval van stelling 4.8 met $S := S_2$.

b) $D_1 \not\subset S_2$. Dan geldt:

(1) $\exists_{d_1 \in D_1}[d_1 \notin S_2]$

Stel nu:

(2) $D_1 \overset{fa}{\Leftrightarrow} D_2$, dan geldt (met stelling 4.1)

(3) $D_2 \overset{fa}{\rightarrow} \{d_1\}$.

TY is in 3NF, dus volgt uit (1):

(4) $S_2 \overset{dfa}{\rightarrow} \{d_1\}$.

Uit (3) en (4) volgt (met definitie 4.4 en stelling 4.8): $S_2 = D_2$, hetgeen in tegenspraak is met het gegeven. De veronderstelling sub(2) kan dus niet gelden bij $D_1 \not\subset S_2$. □

Als het type van een objectgroepvariabele Y, met attributenverzamelingen AY en karakterisering KY niet in 3NF is, dan zullen een of meer attributen uit AY en dienovereenkomstig een of meer paren uit KY moeten worden "verwijderd", zodat het type van het "overblijvende gedeelte" van Y wel in 3NF is. Terzelfder tijd zullen deze verwijderde attributen en paren dan moeten worden "ondergebracht" in reeds bestaande of nieuw te creëren attributenverzamelingen en karakteriseringen. Dit "verwijderen" en "onderbrengen" dient zó te geschieden, dat uiteindelijk alleen objectvariabelen reusulteren met typen in 3NF. Zodoende kunnen wij spreken van karakteriseringen in 3NF als de typen van de bijbehorende objectvariabelen in 3NF zijn.

Voorbeeld 5.1: Uitgaande van de beweringen in voorbeeld 4.3 zijn de karakteriseringen in de voorbeelden 3.1, 3.2 en 3.3 niet alle in 3NF. Als beweerd wordt dat onderstaande karakteriseringen (K_1^N tot en met K_4^N) in 3NF zijn, dan wordt dit in ieder geval niet weersproken door voorbeeld 4.3. Of het werkelijk zo is, hangt natuurlijk af van alle afhankelijkheden, waarop wij hier verder niet ingaan. Wij zullen bij elke K_i^N alleen het domein noemen, dus de attributenverzameling A_i^N.

K_1^N van objectgroepvariabele "art"

met A_1^N = {ano,anm,pr,gew,kl}.

K_2^N van objectgroepvariabele "lev"

met A_2^N = {lnr,lnm,ladr,lwpl.abnr}.

K_3^N van objectgroepvariabele "order"

met A_3^N = {lnm,ladr,lwpl,anm,onr,dat,hoev}.

K_4^N van (nieuwe) objectgroepvariabele "woonpl"
met A_4^N = {wpl,netnr}.

6. DATABASE; DATABASE VARIABELE; EXTERNE SLEUTEL

Tot nu toe zijn wij in de formele opzet niet verder gekomen dan de behandeling van aparte, los van elkaar staande objectgroepvariabelen. Het ligt echter voor de hand om te veronderstellen dat tussen de waarden van verschillende objectgroepvariabelen ook verbanden bestaan. Zo zou bijv. bij voorbeeld 5.1 de eis kunnen gelden dat een waarde van attribuut amn in een tupel van objectgroepvariabele "order" alleen is toegestaan als op hetzelfde moment objectgroepvariabele "art" een tupel bevat met dezelfde waarde voor het attribuut amn. Het gaat hier dus blijkbaar om integrity constraints van "hogere orde", nl. constraints die niet in het type van één objectgroepvariabele kunnen worden weergegeven. Om deze constraints toch weer te kunnen geven zullen wij een variabele van een "hogere orde" nodig hebben. Zo'n variabele zal zelf weer moeten kunnen stoelen op een "hogere orde" karakterisering.

Wij beginnen dan ook met de invoering van het begrip database, waarvan wij evenals van het begrip object geen definitie geven. Bij elke database hoort een verzamelingsfunctie.

Definitie 6.1: De bij een database U behorende verzamelingsfunctie KU heet de databasekarakterisering van U. Elk element van het domein van KU heet een component van U en het beeld van een component heet het type van deze component. □

Bij elke component van een database hoort een verzamelingsfunctie.

Definitie 6.2: De bij een component C van een database behorende verzamelingsfunctie KC heet de componentkarakterisering van C. Elk element van het domein van KC heet een attribuut van C en het beeld van een attribuut heet het type van dit attribuut. Elk element van de bijbehorende produktverzameling PROD(KC) heet een tupel. □

Een component van een database kan gezien worden als de "vertegenwoordiger" van een objectgroepvariabele "binnen het kader" van genoemde database.

Voorbeeld 6.1: De karakterisering van de database "handel" met componenten

die "overeenkomen" met de objectgroepvariabelen van voorbeeld 5.1, zal er als volgt uitzien (hierbij wordt met "t-art" enz. het type aangegeven van "art" enz.).

K-handel = {(art,t-art),(lev,t-lev),(order,t-order),(wpl,t-wpl)}. □

Bij een database kan een variabele worden gedeclareerd. Zo'n variabele noemen wij een database-variabele. Wij maken de voor de hand liggende afspraak dat met de karakterisering resp. de verzameling componenten van een database-variabele wordt bedoeld de karkaterisering resp. de verzameling componenten van de betreffende database.

Een waarde van een database-variabele Z is een element van de produktverzameling van de karakterisering van Z. Om enig idee te hebben van zo'n waarde, moeten wij nader ingaan op het type van een component. Aangezien een component gezien wordt als "overeenkomend" met een objectvariabele, ligt het voor de hand dat voor het type van een component regels gelden, die analoog zijn aan die voor het type van een objectvariabele. Een waarde van component C met karakterisering KC is dan ook een deelverzameling van PROD(KC). Ook hier zal weer gelden dat niet elke deelverzameling van PROD(KC) mag optreden als waarde van C. Voor het type van een component geldt dan de volgende uitspraak: Het type van een component C met karakterisering KC is een deelverzameling van de machtsverzameling van PROD(KC).

Na bovenstaande moge duidelijk zijn dat de definities resp. stellingen van hoofdstuk 4 en 5 op dezelfde wijze toepasbaar zijn op resp. gelden voor componenten van een database als voor objectgroepvariabelen. Zo zullen wij bijv. spreken van "de sleutelverzameling van een component" en van "het type van een component in 3NF" enz.

Een waarde van een database-variabele met een aantal van n componenten zal bestaan uit n deelverzamelingen van produktverzamelingen (te vergelijken met n bestanden in het niet-formele vlak).

Ook voor een database-variabele Z geldt dat er elementen van de bijbehorende produktverzameling kunnen zijn, die niet als waarde van Z mogen optreden. De verzameling toegelaten waarden van een database-variabele Z noemen wij het type van Z. Voor zo'n type geldt dus de volgende uitspraak: Het type van een database-variabele Z met karakterisering KZ is een deelverzameling van PROD(KZ).

De beperking van genoemd type (verzameling mogelijke waarden) tot een deelverzameling van PROD(KZ) komt voort uit het feit dat wij ook op database-niveau bepaalde beperkingen (integrity constraints) willen laten gelden. (Zie voorbeeld met anm in "order" en "art" aan het begin van deze paragraaf). Daarvoor zijn de "hogere orde" begrippen van deze paragraaf ingevoerd. Door genoemde beperkingen zullen bepaalde elementen van genoemde produktverzameling niet behoren tot de verzameling van toegelaten waarden voor de database-variabele.

Voor ons betoog is èèn bepaalde soort integrity constraint van belang, nl. die welke wordt weergegeven met behulp van een zogeheten externe sleutel. In de literatuur komt men dit (onduidelijk gedefinieërde) begrip tegen onder de naam foreign key [DATE, p. 78].

Ruwweg kan worden gezegd dat een externe sleutel van een database-component bestaat uit een verzameling attributen, die eenduidig verwijst naar een sleutel van een andere database-component. Een en ander zullen wij nu formeel vastleggen met behulp van de volgende definitie.

Definitie 6.2: Zij

Z een database-variabele
TZ type van Z
C_1, C_2 componenten van Z, met $C_1 \neq C_2$.
ES een deelverzameling van de attributenverzameling van C_1
S een sleutel van C_2

dan heet ES een externe sleutel van C_1 ten opzichte van sleutel S van C_2, als er een afbeelding f bestaat met dom(f) = TZ èn $\forall W \in TZ$ [f(W) is een afbeelding van

$$W(C_1)\|_{ES} \text{ in } W(C_2)\|_S] \; . \qquad \square$$

Voorbeeld 6.2: Bij voorbeeld 6.1 (zie ook voorbeelden 4.3 en 5.1), zouden o.a de volgende externe sleutels kunnen worden gedeclareerd:

a) {anm} in A_3^N externe sleutel van "order" t.o.v sleutel {anm} van "art"

b) {lwpl} " A_2^N " " " "lev" " " {wpl} van "woonpl"

c) {lnm,ladr,lwpl} in A_3^N " " "order" " " {lnr} van "lev". $\square$

Opmerkingen:

- Een externe sleutel hoeft niet dezelfde naam te dragen als de sleutel waarnaar hij "verwijst" (voorbeeld 6.2.b). Hij hoeft zelfs niet uit hetzelfde aantal attributen te bestaan (voorbeeld 6.2.c).
- Voor externe sleutels geldt evenals voor sleutels dat deze als zodanig alleen door de gebruikers kunnen worden gespecificeerd.
- Door definitie 6.2 is het niet uitgesloten, dat een externe sleutel ook sleutel is van de "eigen" component , dus dat ES in definitie 6.2 ook sleutel is van C_1. Een (ietwat gekunsteld) voorbeeld moge als toelichting dienen.

Voorbeeld 6.3: Tot een database behoren o.a. de componenten "art" (artikel) en "mag" (magazijn). De attributenverzameling van "art" resp. "mag" is:

A_1 = {anr(artikelnummer), anm(artikelnaam), vr(voorraad), mnr(magazijnnummer)}

resp.

A_2 = {mnr(magazijnnummer), madr(magazijnadres), mc(magazijncapaciteit), anr(artikelnummer)}.

Stel nu dat "mnr" $\in A_1$ resp. "anr" $\in A_2$ externe sleutel zijn van "art" resp. "mag" ten opzichte van sleutel "mnr" van "mag" resp. sleutel "anr" van "art". Het gevolg hiervan is dat "mnr" resp. "anr" sleutel is van "art" resp. "mag". Dit zou dan de representatie kunnen zijn van het (ietwat gekunstelde) feit, dat alle exemplaren van eenzelfde artikel liggen opgeslagen in hetzelfde magazijn èn dat in elk magazijn slechts één exemplaren van eenzelfde artikel liggen opgeslagen. □

Wij kunnen dit soort gevallen waarbij externe sleutel ook "eigen sleutel" is eenvoudig opheffen door de twee betreffende componenten te reduceren tot een component (hetgeen vanwege het "één-op-één verband" tussen de componenten een voor de hand liggende reductie is) met als attributenverzameling de vereniging van beide attributenverzamelingen. In voorbeeld 6.3 zou dit dus leiden tot de reductie van "art" en "mag" tot één component (bijv. "artmag") met attributenverzameling A = {anr,mnr,anm, vr, madr,mc}. Een database (en de bijbehorende database-karakterisering en database-variabele) noemen wij gereduceerd als de database geen tweetallen componenten bevat, die tot één component kunnen worden gereduceerd.

Wij kunnen nu een uitspraak doen over het verband tussen normalisatie en externe sleutels.

Stelling 6.1: Gegeven is een component C van een gereduceerde database. SC resp. ESC zijn de sleutel- resp. externe sleutelverzameling van C.
Dan geldt:

$$\forall_{ES_1 \in ESC} \forall_{ES_2 \in ESC} [(ES_1 \overset{fa}{\Leftrightarrow} ES_2) \Rightarrow (ES_1 = ES_2)] \quad .$$

Bewijs: Stel:

(1) $ES_1 \overset{fa}{\Leftrightarrow} ES_2$.
De database is gereduceerd, dus:
$ES_1 \notin SC$ èn $ES_2 \notin SC$, dus vanwege 3NF geldt voor elke $S \in SC$:

(2) $S \overset{dfa}{\rightarrow} ES_1$ en $S \overset{dfa}{\rightarrow} ES_2$

Uit (1) en (2) volgt dat $ES_1 = ES_2$. □

Voorgaande stelling luidt in "woorden": een component C in 3NF van een gereduceerde database kan geen equivalente externe sleutels bevatten.

Tenslotte leggen wij nog de volgende afspraak vast: een database is in 3NF als elk van zijn componenten in 3NF is.

7. AFLEIDBAARHEID

Bij voorbeeld 6.1 is het denkbaar dat ook een attribuut "bedrag" in de component "order" voorkomt. Dit attribuut is dan bedoeld om weer te geven de waarde van hoev * pr. Hiermee is dan meteen aangegeven dat dit attribuut "bedrag" functioneel afhankelijk is van {hoev,pr}. Wil men nu alleen maar karakteriseringen in 3NF, dan zou, vanwege het attribuut bedrag, een nieuwe component gecreëerd moeten worden met {hoev,pr} als sleutel. Als wij deze functionele afhankelijkheid echter nader bezien, dan blijkt dat hier sprake is van een zogeheten afleidbaar attribuut, d.w.z. een attribuut, waarvan de waarde kan worden afgeleid, zonder gebruik te maken van de attribuut-waarde zelf in de (waarde van de) database-variabele. De afleiding vindt in dit geval plaats met behulp van de formule: bedrag = hoev * pr.

Een ander voorbeeld van een afleidbaar attribuut is het volgende: attribuut

"prg", wederom bij "order". Hierbij veronderstellen wij dat het bestaande attribuut "pr" de prijs in centen en het nieuwe attribuut "prg" de prijs in guldens geeft. Hier is ook weer duidelijk sprake van een afleidbaar attribuut, immes prg = 0,01 * pr.

Tenslotte een derde voorbeeld van een afleidbaar attribuut, stel component "lev" (voorbeeld 6.1) wordt uitgebreid met het attribuut "aord". Met dit attribuut wordt bedoeld het aantal uitstaande orders per leverancier. We veronderstellen hierbij dan dat "order" betrekking heeft op uitstaande orders. Hier is weer duidelijk sprake van een afleidbaar attribuut. Immers voor een bepaalde leverancier met lnr = c is aord = $\#\{x \mid x \in \text{handel(lev)} \wedge x(\text{lnr}) = c\}$. In tegenstelling tot de twee eerste voorbeelden, zal men bij dit laatste voorbeeld bij de afleiding gebruik moeten maken van andere componenten, dan die waarop het afleidbare attribuut betrekking heeft. In feite zijn er talloze afleidbare attributen te bedenken. Men denke slechts aan de informatie, die in queries wordt gevraagd.

Wij zullen nu het begrip afleidbaar attribuut formeel in een definitie vastleggen.

Definitie 7.1: Zij

Z een database-variabele

TZ het type van Z

CZ de verzameling componenten van Z.

$C_i \in CZ$ $(0 \leq i \leq k)$

AC_i de attributenverzameling van C_i

$A_i \subset AC_i$ (A_0 kan leeg zijn)

$a \in AC_0$ èn $a \notin A_0$, dan

heet het attribuut a afleidbaar van $\bigcup_{1 \leq i \leq k} A_i$, als geldt:
er is een procedure P èn

$$\forall_{W \in TZ}\left[\forall_t \in W(C_0)[t(a) = P(\bigcup_{1 \leq i \leq k} W(C_i)\|_{A_i})]\right] . \qquad \square$$

In het tweede voorbeeld van deze paragraaf gold: prg = 0,01 * pr. Dus geldt ook pr = 100 * prg. In zo'n geval spreken wij van equivalent afleidbare attributen.

Definitie 7.2: Als A en A' een verzameling attributen zijn, zodanig dat A minstens twee elementen bevat, dan heet A een verzameling van equivalent afleidbare attributen als geldt:

$$\forall_{a \in A}[a \text{ afleidbaar van } A \backslash \{a\} \cup A'].$$ □

Opmerking: A' kan eventueel de lege verzameling zijn.

Met behulp van bovenstaande definitie kunnen wij de afleidbare attributen in twee klassen onderscheiden, nl.

- de niet-equivalent afleidbare attributen, d.w.z. afleidbare attributen, die niet voorkomen in een verzameling van equivalent afleidbare attributen.
- de equivalent afleidbare attributen, d.w.z afleidbare attributen, die wel voorkomen in een verzameling van equivalent afleidbare attributen.

Evenals voor afhankelijkheden en sleutels, zo geldt ook voor afleidbaarheid dat deze (inclusief de daarvoor benodigde procedure) niet door het formele systeem, maar alleen door de gebruikers kan worden vastgelegd.

8. VOORWAARDEN VOOR EEN UNIEKE DATABASE KARAKTERISERING

Wij zullen nu nagaan onder welke voorwaarden de karakterisering van een database-variabele uniek vastligt, als de betekenis van de gegevens volledig vastligt.

De betekenis van de gegevens ligt volledig vast als bekend zijn

- de database componenten met hun attributen en typen
- de verzamelingen van sleutels en externe sleutels per component
- de afhankelijkheden tussen de attributen
- de afleidbaarheid tussen de attributen.

Hierbij moet worden opgemerkt dat de typen van genoemde componenten voor de volledige weergave van de betekenis niet in 3NF hoeven te zijn, mits de afhankelijkheden duidelijk vastliggen.

Wij gaan nu aan de uiteindleijke database-karakterisering de volgende drie voorwaarden opleggen:

V1: de database moet gereduceerd zijn (dus geen externe sleutel, die ook sleutel van "eigen" component is; zie paragraaf 6)

V2: er mogen geen afleidbare attributen in voorkomen.

V3: de database moet in 3NF zijn, dus elke component moet in 3NF zijn.

Wij zullen nu aantonen dat realisering van achtereenvolgens V1, V2 en V3 leidt tot een unieke database-karakterisering behoudens equivalent afleidbare attributen en equivalente externe sleutels.

Ad V1: Deze voorwaarde is eenvoudig te realiseren door reductie toe te passen als besproken in paragraaf 6. Het resultaat van zo'n reductie is uit de aard der zaak uniek bepaald.

Ad V2: Om te beginnen zullen alle niet-equivalent afleidbare attributen uit de karakteriseringen van de componenten moeten worden verwijderd. Het resultaat van deze verwijdering ligt uiteraard uniek vast. Dit geldt niet ten opzichte van de verwijdering van equivalent afleidbare attributen. Immers van deze klasse afleidbare attributen dienen er zoveel verwijderd te worden tot er geen afleidbare attributen meer in de database voorkomen. Aangezien het hier echter gaat om equivalent afleidbare attributen ligt het resultaat van deze verwijderingsoperatie niet uniek vast. Er is dus sprake van een uniek resultaat behoudens equivalent afleidbare attributen.

Ad V3: Bij de behandeling van V3 zullen wij uitgaan van de volgende notatie:

Z: database-variabele

CZ: verzameling componenten van Z.

C,C': elementen van CZ.

KC,KC': karakteriseringen van C,C'.

SC,SC': sleutelverzamelingen van C,C'.

AC,AC': attributenverzamelingen van C,C'.

Stel het volgende geldt:

(1) $$\exists_{C\in CZ}\exists_{a\in AC}\exists_{S\in SC}[a \notin S \wedge \neg(S \overset{dfa}{\rightarrow} \{a\})] .$$

Zolang er een component C is met attribuut a, waarvoor (1) geldt, is de database niét in 3NF.

Er moet dan voor dit attribuut a het volgende worden ondernomen:

a) het attribuut a moet worden verwijderd uit KC.

b) Uit (1) volgt dat de verzameling:

$$V = \{A \mid A \subset AC \wedge A \overset{dfa}{\rightarrow} \{a\}\}$$

niet leeg is.

Bovendien geldt dat V een verzameling is van onderling equivalente attributenverzamelingen.
Nu kunnen zich twee gevallen voordoen:

b.1) $\exists_{A\in V}\exists_{C'\in CZ}\exists_{S'\in SC'}$[A is externe sleutel van C ten opzichte van S' van C'] .

In dit geval moet attribuut a worden toegevoegd aan KC'.
Als bovendien a equivalent is met A dan moet {a} ook nog worden toegevoegd aan SC'.

b.2) $\forall_{A\in V}$[A is geen externe sleutel] .

In dit geval moet een nieuwe component C'' aan CZ worden toegevoegd met één element van V, zeg A, en het attribuut a als attributen van C''. De sleutelverzameling van C'' zal gelijk zijn aan {A} resp. {A,{a}} als a niet resp. wel equivalent is met A.

Bovenstaande werkwijze moet worden herhaald totdat er geen enkel attribuut meer is waarvoor (1) geldt. Dan is uiteindelijk bereikt dat het type van elke component in 3NF is. Het moge uit bovenstaande ook duidelijk zijn dat deze werkwijze leidt tot een uniek resultaat behoudens equivalente externe sleutels.

Samenvattend kan inderdaad worden gesteld dat realisering van achtereenvolgens V1, V2 en V3 inderdaad leidt tot bovengenoemd unieke resultaat.

Dankwoord Aan de heren E.O. de Brock en M.L. Potters ben ik veel dank verschuldigd voor hun uitgebreid en waardevol commentaar op eerdere versies van dit artikel.

LITERATUUR

[1] BERNSTEIN, P.A., Synthesizing Third Normal Form Relations from Functional Dependencies. ACM Transactions on Database Systems, December

1976, pp. 277-293.

[2] BROCK, E.O. de, Tables, database tables and static integrity constraints, memorandum 80-12, T.H. Eindhoven, Omderafdeling der Wiskunde, 1980.

[3] CODD, E.F., A relational model of Data for Large Shared Data Banks, Communications of the ACM Vol. 13, no. 6, June 1970, pp. 377-387.

[4] CODD, E.F., Extending the database relational model to capture more meaning, ACM Transactions on Database Systems Vol. 4, no. 4, december 1979, pp. 397-434.

[5] DATE, C.F., An introduction to Database Systems, 2nd edition, Addison-Wesley, 1977.

[6] FAGIN, R., Multivated depedencies and a new formal form for relational databases, ACM Transactions on Database Systems, Vol. 2, no. 3, september 1977, pp. 262-278.

[7] FALKENBERG, E., Some aspects of conceptual data modelling, Lecture Notes, Summer School on Data Base Design, 1979, Urbino, Italy.

[8] MOHAN, C., An overview of recent database research, Database, quarterly newsletter of SIGBDP of the ACM Vol. 10, no. 2, 1978.

[9] SMITH, J.M., SMITH, D.C.P., Principles of Database conceptual design. Proc. NYU Symposium on Database design, New York, may 1978, pp. 35-49.

A FRAMEWORK FOR ADVANCED MASS STORAGE APPLICATIONS*

G.M. NIJSSEN
Control Data and University of Brussels

0. ABSTRACT

In this paper we will present a framework for information systems which is considered to be advanced in three aspects.

The first is that this framework is easy to understand for the user, without being first immersed in computer concepts and jargon. The basic axiom of this framework is that an information system can be considered as a simplified natural language communication system, close to Wittgenstein's concept "language game". Users master natural language and consequently it is fairly simple and straightforward for them to get acquainted with an information system's approach which is a subset of a natural language communication.

Concepts like information base, information systems grammar and the distinction between things and names of things are essential in this approach.

The second advanced aspect of this approach is the software architecture of the underlying information base management system which contains a four schema or grammar approach; the most abstract problem description grammar, called in this paper the idea base grammar, and the naming convention grammar, called here bridge base grammar; the combination of idea base grammar and bridge base grammar could be called conceptual grammar; furthermore there is the internal grammar to describe the computer efficiency oriented aspect and the external grammar which describes the programming aspects. In doing so, users can concentrate their efforts on the WHAT of an information system, or on information analysis without having to worry about computer efficiency or programming efficiency aspects, and even within this "WHAT" area they can make a complete problem description without having to worry about naming

* This is an unabridged version of a paper presented at MEDINFO 1980, Tokyo, the World Conference on Medical Informatics.

conventions. Once the WHAT of an information system has been established, the computer efficiency expert, or data base engineer can start to work on HOW to get a certain optimum in terms of computer resources of the internal grammar, while the programming efficiency expert, or application engineer can start to work on HOW to get a certain optimum in terms of programming resources or the external grammar.

The third aspect is that this approach permits building information systems support software in the same way as we build software to be used by users.

Experience with this approach in the last four years has shown a considerable increase in user involvement in the formal definition of the problem, with the consequence of far better user acceptance, better quality of problem description resulting in far lower information systems building and maintenance costs, and higher reliability.

The practical implementation of this framework requires the availability of advanced mass storage equipment. Conversily, the use of advanced mass storage equipment without such a framework seems in many cases questionable.

1. INTRODUCTION

In this paper we will gradually build up a framework for information systems which is understandable to users and at the same time complete. The framework is based upon six axioms which will be introduced and described explicitly. Most, if not all, of these axioms are essentially very simple; experience with teaching these axioms to educated laymen, not being computer experts, has shown that users can understand fully this framework, provided they invest a reasonable amount of learning time.

One of the consequences of this approach is that users and edp experts can cooperate on a more equal basis as opposed to the situation where the edp expert *dictates* the solution.

Experience has shown that both users and edp experts enjoy this form of co-determination and that software development and maintenance costs can be substantially reduced.

2. THE ENALIM AXIOM

The first axiom of our approach to information systems is:

Each permitted communication between a user and an information system can be considered to consist of a set of elementary sentence instances. This axiom has been given the name ENALIM, which is an acronym for Evolving Natural Language Information Model (NIJSSEN [6]). Figure 2.a is a graphical illustration of the ENALIM axiom, which says that the arrows, from the user to the information system and vice versa, can be considered to consist of a set of elementary sentence instances.

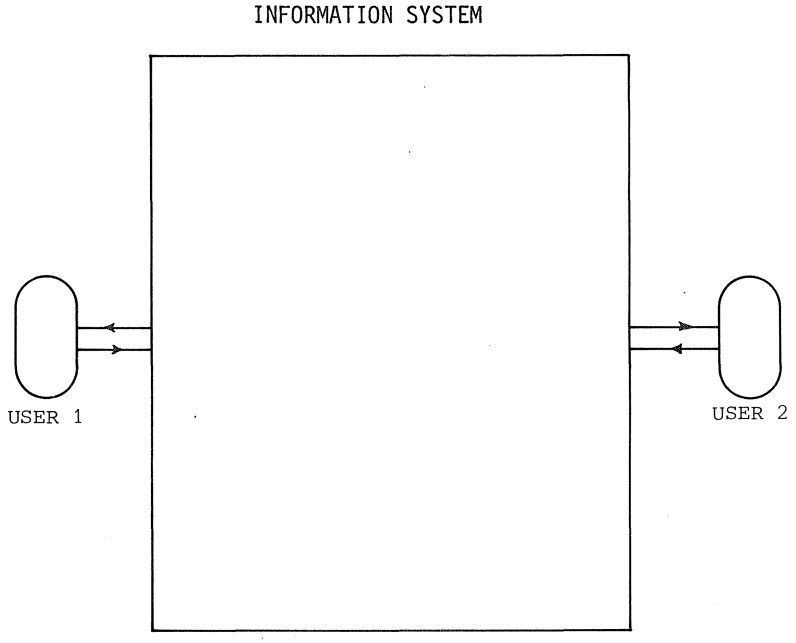


Figure 2.a

The model we use to describe sentences is based on the one used in logic and linguistics, with three extensions: for every object name in the sentence one explicitly mentions the object type (class, category) to which the object, referred to by the object name, belongs and one explicitly mentions the naming convention; furthermore we are only interested in semantically irreducible sentences, which we call elementary sentences.

Examples of such sentences are:

- The employee with employee number E01 works for the department with department number D1.

- The employee with employee number E03 works for the department with department number D2.

- The employee with employee number E01 was born in the country with country code U.K.

The consequence of the ENALIM axiom is that in such an approach there is a need for only two update operators, namely ADD or DELETE elementary sentence.

Another consequence is that if the communication from the user to the information system consists of a set of elementary sentence instances, then the knowledge base inside the information system consists of a set of sentence instances. For now, we call this knowledge base the sentence base (see figure 2.b).

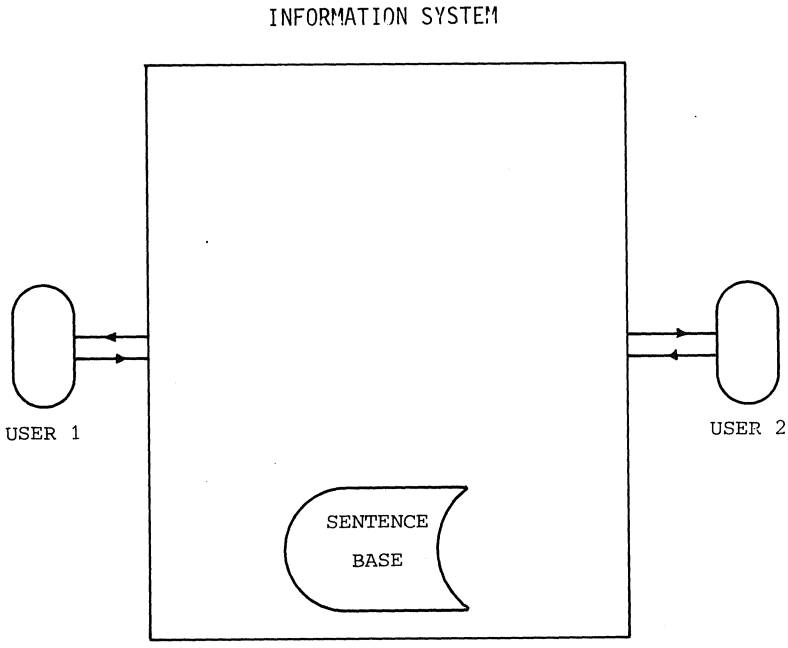


Figure 2.b

3. THE INSYGRAM AXIOM

The second axiom of our approach to information systems is:

Each permitted communication between a user and an information system can be completely prescibed by one single grammar.

This axiom has been given the name INSYGRAM, which is an acronym for INformation SYstems GRAMmar; the single grammar we call sentence base grammar.

An equivalent formulation of the INSYGRAM axiom is:

A sentence base grammar is a set of rules which completely and exclusively prescribes all the permitted sentence base instances and all the permitted sentence base transitions.

Figure 3.a is a graphical illustration extending figure 2.a to include the sentence base grammar.

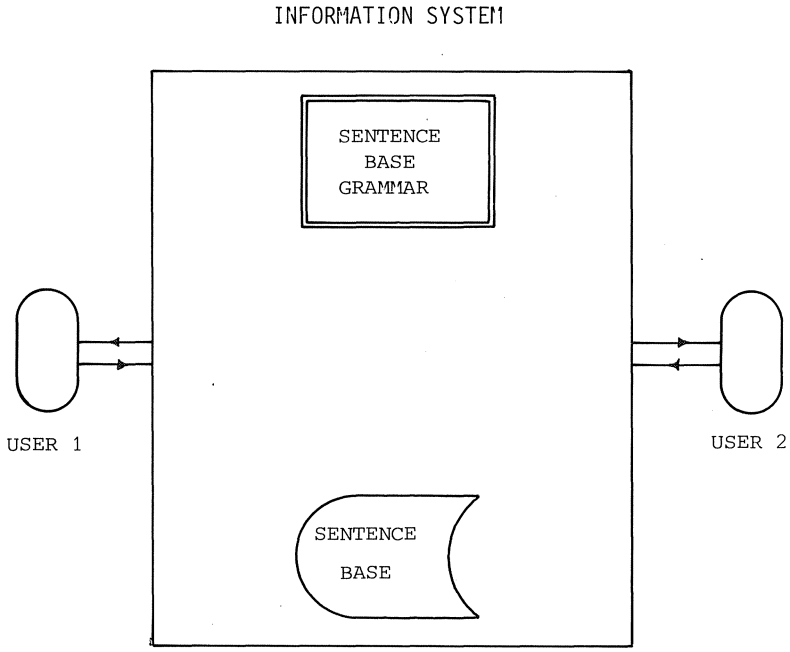


Figure 3.a

Figure 3.b, 3.c, 3.d and 3.e are four sentence base instances, which are permitted by the same sentence base grammar of the next page. Stated informally, this sentence base grammar prescribes that one is interested in sentence bases which contain sentences expressing which president is born in which state and which president died in which state, where president is referred to by president name and state is referred to by state name, and the following three conditions or constraints apply, namely a president can at most have one birth-state, a president can have at most one death-state, and

we only want the sentence in the sentence base, expressing that a certain president died in a certain state, if for that same president there is already the sentence in the sentence base, expressing in which state this president was born.

```
PRESIDENTS-BIRTHSTATE (Kennedy, Massachusetts)
PRESIDENTS-BIRTHSTATE (Nixon, California)
PRESIDENTS-BIRTHSTATE (Adams J, Massachusetts)
PRESIDENTS-DEATHSTATE (Kennedy, Texas)
PRESIDENTS-DEATHSTATE (Adams J, Massachusetts)
```

Figure 3.b

```
PRESIDENTS-BIRTHSTATE (Kennedy, Massachusetts)
PRESIDENTS-BIRTHSTATE (Adams J, Massachusetts)
PRESIDENTS-DEATHSTATE (Kennedy, Texas)
PRESIDENTS-DEATHSTATE (Adams J, Massachusetts)
```

Figure 3.c

```
PRESIDENTS-BIRTHSTATE (Kennedy, Massachusetts)
PRESIDENTS-BIRTHSTATE (Adams J, Massachusetts)
PRESIDENTS-BIRTHSTATE (Eisenhower, Georgia)
PRESIDENTS-DEATHSTATE (Kennedy, Texas)
PRESIDENTS-DEATHSTATE (Adams J, Massachusetts)
```

Figure 3.d

```
PRESIDENTS-BIRTHSTATE (Kennedy, Massachusetts)
PRESIDENTS-BIRTHSTATE (Adams J, Massachusetts)
PRESIDENTS-BIRTHSTATE (Eisenhower, Texas)
PRESIDENTS-DEATHSTATE (Kennedy, Texas)
PRESIDENTS-DEATHSTATE (Adams J, Massachusetts)
```

Figure 3.e

```
SENTENCE BASE GRAMMAR NAME IS EXAMPLE-ONE

SENTENCE TYPE DIVISION

    SENTENCE TYPE NAME IS PRESIDENTS-BIRTHSTATE

        INVOLVED OBJECT TYPE NAME IS PRESIDENT, USING PRESIDENT-NAME,
        IN ROLE WAS-BORN-IN
        INVOLVED OBJECT TYPE NAME IS STATE, USING STATE-NAME,
        IN ROLE IS-BIRTHSTATE-OF

    SENTENCE TYPE NAME IS PRESIDENTS-DEATHSTATE

        INVOLVED OBJECT TYPE NAME IS PRESIDENT, USING PRESIDENT-NAME,
        IN ROLE DIED-IN
        INVOLVED OBJECT TYPE NAME IS STATE, USING STATE-NAME
        IN ROLE IS-DEATHSTATE-OF

CONSTRAINT DIVISION

    CONSTRAINT NAME IS AT-MOST-ONE-BIRTHSTATE-PER-PRESIDENT
        ROLE WAS-BORN-IN IS UNIQUE

    CONSTRAINT NAME IS AT-MOST-ONE-DEATHSTATE-PER-PRESIDENT
        ROLE DIED-IN IS UNIQUE

    CONSTRAINT NAME IS NO-DEATHSTATE-WITHOUT-BIRTHSTATE
        ROLE DIED-IN
                    IS CONTAINED IN
        ROLE WAS-BORN-IN

END SENTENCE BASE GRAMMAR.
```

In figure 3.f there is represented a combination of the major parts of the grammar EXAMPLE-ONE, and a sentence base instance (= contents of figure 3.b) satisfying this grammar. Figure 3.f is an example of a sentence base, represented in semantically extended table format. The advantage of such a representation is that the user can see in the same figure both the important aspects of a sentence base grammar and a sentence base instance.

PRESIDENTS-BIRTHSTATE

PRESIDENT	STATE
PRESIDENT-NAME	STATE-NAME
WAS-BORN-IN	IS-BIRTHSTATE-OF
Kennedy	Massachusetts
Nixon	California
Adams J	Massachusetts

PRESIDENTS-DEATHSTATE

PRESIDENT	STATE
PRESIDENT-NAME	STATE-NAME
DIED-IN	IS-DEATHSTATE-OF
Kennedy	Texas
Adams J	Massachusetts

Figure 3.f

In order to come to a complete conceptual framework for information systems, we need to add two components to figure 3.a.

If the sentence base grammar describes all the permitted sentence base stated and all the permitted sentence base transitions, then the sentence base grammar completely presribes the dynamic behaviour of the sentence base and then there is also a need for an authority which will guarantee that user requests will be honoured only if they are in accordance with the sentence base grammar. This authority we will give the name sentence base grammar enforcer (see figure 3.g).

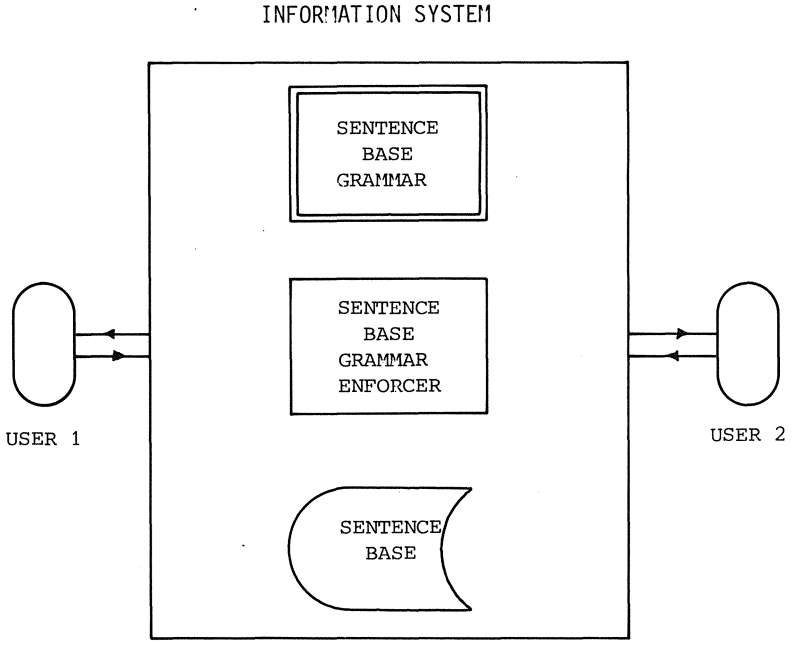


Figure 3.g

If we were to adhere to the information systems framework of figure 3.g, then all users would have to communicate with the sentence base grammar enforcer in its language, and furthermore, the sentence base grammar enforcer should know all the presentation conventions of the different users. This seems too big a burden, and therefore we introduce one other component in the information systems framework, which has the following three jobs:

1. translate the user's request into the language of the sentence base grammar enforcer;
2. present the sentences, which the sentence base grammar enforcer transmits as a response to a user requenst, in a way that a specific user wants to see them (presentation cosmetics, report generation);
3. assemble small requests of users into a larger request which is acceptable for the sentence base grammar enforcer.

This component we will give the name application program (see figure 3.h).

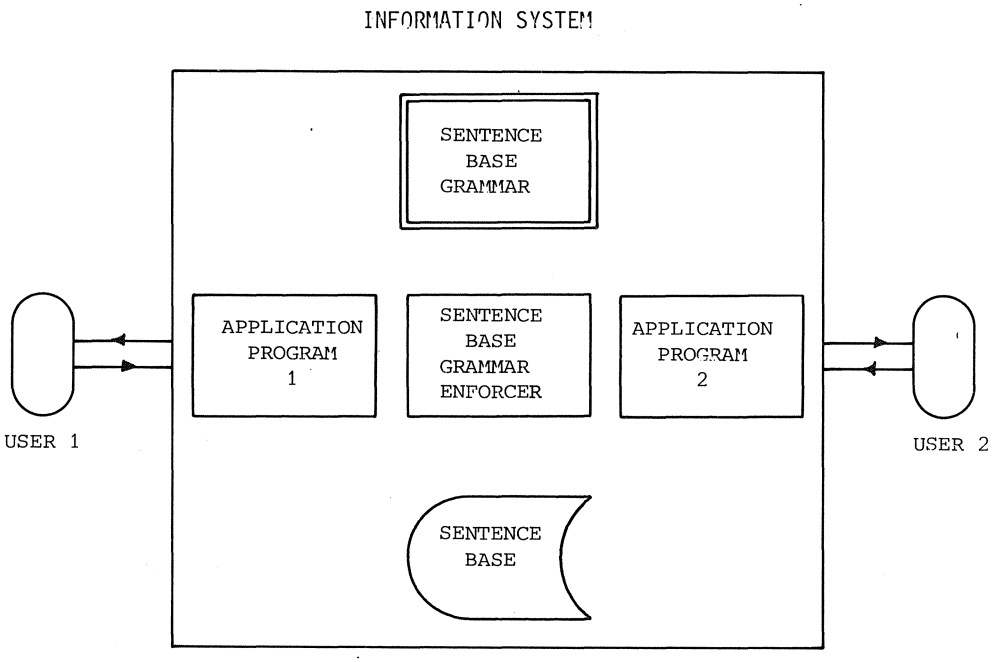


Figure 3.h

Removing the black box surrounding the sentence base grammar, sentence base grammar enforcer, sentence base and application programs and adding arrows to represent the information flow between the components results in figure 3.i. This is the hierarchical decomposition of the framework of figure 2.a.

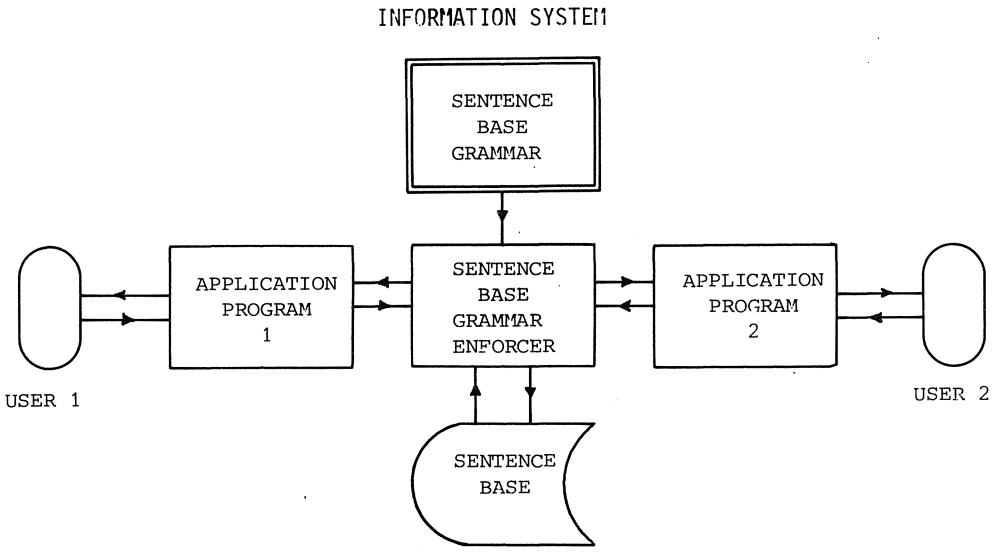


Figure 3.i

4. INSYGRAM BASED INFORMATION SYSTEMS FRAMEWORK AND CONVENTIONAL FRAMEWORKS

The information systems framework as developed in the previous section can be used to explain the various software approaches to information systems in the past, present and extrapolatable future. In the early days of computerised information systems, say until 1960, there was no separate sentence base grammar, nor a sentence base grammar enforcer. In those days the application program contained the entire sentence base grammar (encoded in computer oriented constructs) and was suppose to perform the job of the sentence base grammar enforcer, on top of the tasks of report generation and assembling of user requests.

With the introduction of file management software in the 1960's, we

see the emergence of a so-called file schema. The file schema contained - roughly speaking - some 5% of the sentence base grammar rules while the remaining 95% remained in the application program. In this approach we see the distribution of the responsibility of enforcing the sentence base grammar: 95% of the sentence base grammar had to be enforced by the application program, while 5% had to be enforced by the so-called file manager, a predecessor of the sentence base grammar enforcer (see figure 4.a).

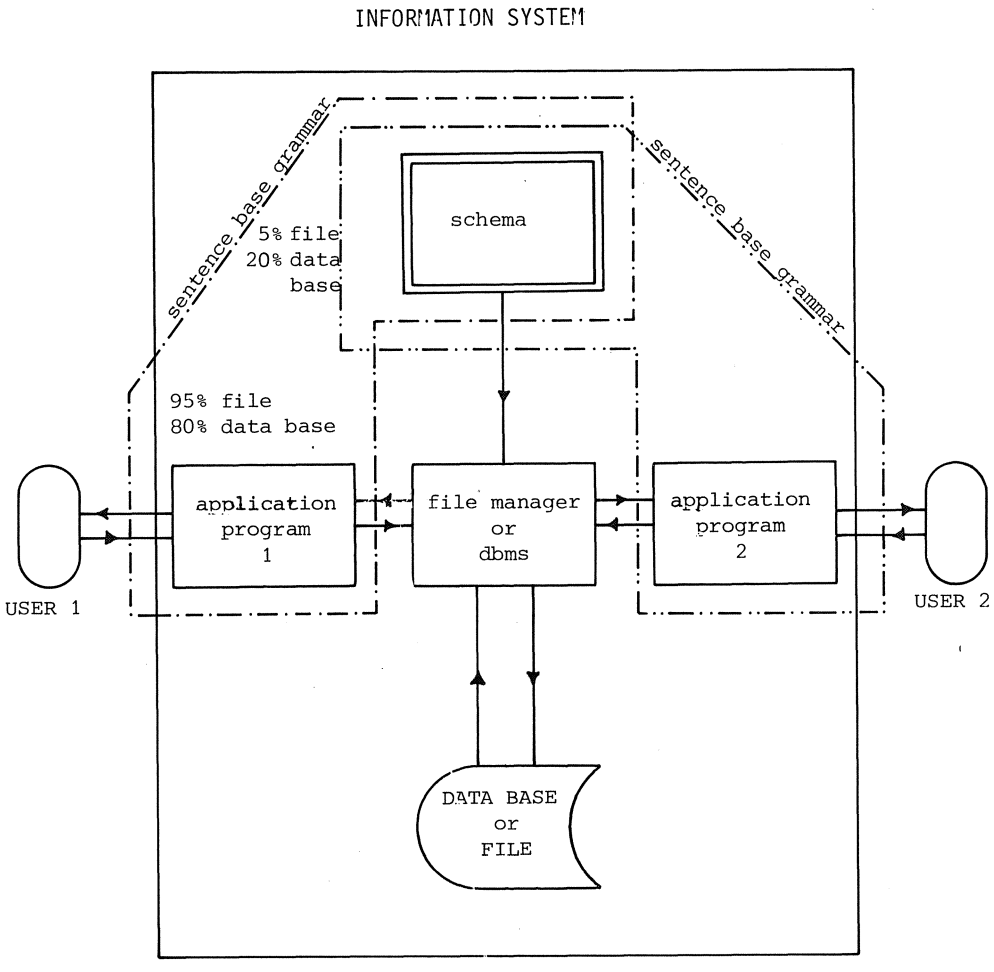


Figure 4.a

From the point of view of central control over enforcement of the sentence base grammar, the introduction of data base management software is a gradual step forward in the correct direction. The schema contains some 20% of the sentence base grammar while the application program contains the remaining 80%. The dbms (data base management system), the successor of the file manager and predecessor of the sentence base grammar enforcer, is responsible for enforcing the 20% of the sentence base grammar, while the application program is responsible for enforcing the remaining 80%.

In our opinion, it is possible that in the near future the INSYGRAM based software will enter the market, and then there will be just one place in which the entire grammar is defined, namely the sentence base grammar, and just one unit responsible for the enforcement of the sentence base grammar as outlined in the previous section. Figure 4.b gives a survey in tabular form of the previously stated historical information.

% of grammar in / period or approach	application program	schema
pre-file management software	100	0
file management software	95	5
data base management software	80	20
INSYGRAM based software	0	100

Figure 4.b

With such an approach to information systems, it is possible to have a much better distribution of work. The software or hardware manufacturer will produce the sentence base grammar enforcer, the users are responsible (possibly with the help of a professional information analyst) to produce

the sentence base grammar (= precise description of the WHAT) and the professional programmers are responsible for writing the application programs (sometimes the user could do this in case there is a very user oriented retrieval and update language available).

5. THE SENE AXIOM

In sections 2 and 3 we have developed a framework for information systems in which we have abstracted from computer efficiency conventions (such as pointers, physical records, files, etc.) and programming efficiency conventions (such as grouping of elementary sentences, etc.) and have dealt with elementary sentences.

The question may be asked: is an elementary sentence a contruct from which one cannot abstract? Before we answer let us look at the following two different sentences:

a. The employee with employee number E01 works for the department with department number D1.

b. The employee with social security number 321 826 754 works for the department with department number D1.

If we now inform the reader that the employee with employee number E01 is the same employee as the employee with social security number 321 826 754, then it is clear that sentence a and b communicate the same *message* or *abstract sentence*, while the naming conventions are different. In short, it is possible to distinguish in a sentence a part which is independent of naming conventions, and a part which is nothing else but naming conventions.

The first part we give the name *idea* or *abstract sentence* and the second part we call *bridge*. Why the latter name? Because a naming convention is a bridge between the world of the objects and the world of lexical or string objects. This was an informal introduction into the third axiom of our approach to information systems.

The deepest structure of each sentence can be considered to consist of a set of ideas about objects, and a set of bridges between objects and lexical objects.

This axiom has been given the name SENE, which is an acronym for Senko Entity Name Entity, in memory of the late Mike Senko, which had done so much to introduce the distinction between entity names and entities into the

data base research community during the seventies (SENKO [8]).

The consequence for the INSYGRAM information systems framework of accepting the SENE axiom is that we can replace the sentence base grammar with a grammar which deals with ideas and bridges. We call this grammar *information base grammar*, or *conceptual grammar* and consequently we call a sentence base, in which the deepest structure of the sentences is clearly visible, an information base, and thus we call the enforcer the *information base grammar enforcer* (see figure 5.a).

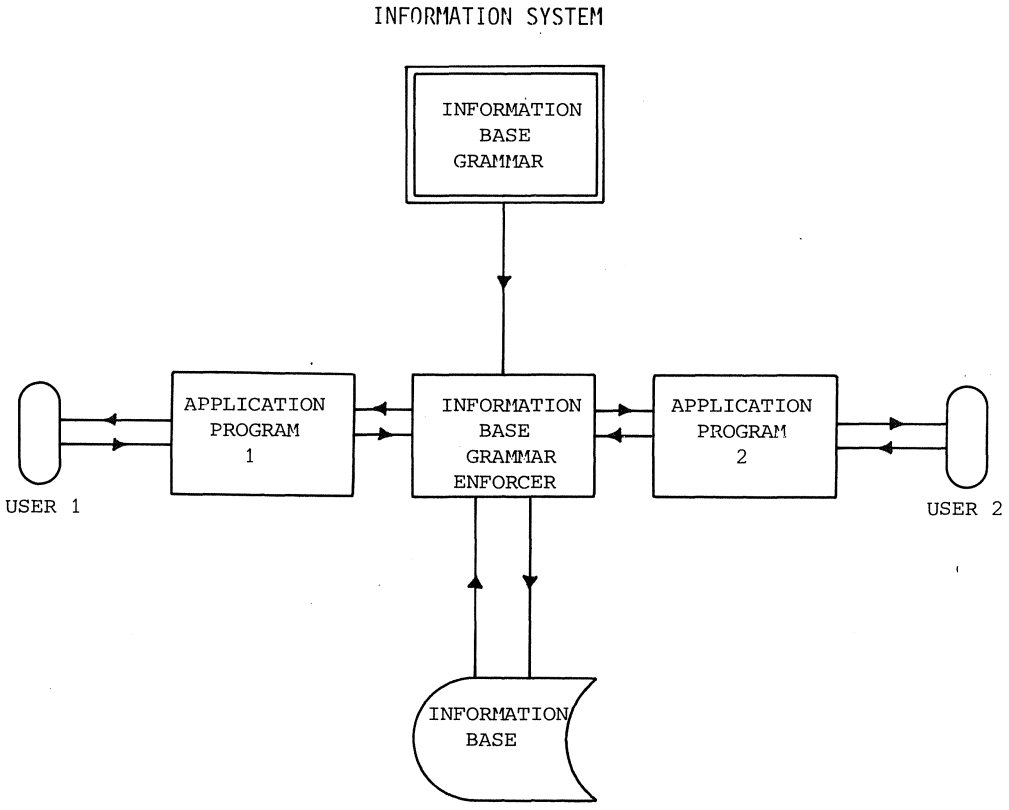


Figure 5.a

6. THE INTERNAL AND EXTERNAL AXIOMS

So far we have developed a framework for information systems in which we have put aside the aspects of computer efficiency and programming efficiency. These two aspects will now be added to the information systems framework of figure 5.a which is based on the ENALIM, INSYGRAM and SENE axioms.

First we introduce the INTERNAL axiom:

The physical representation of each permitted information base can be completely prescribed in one single grammar.

We call this grammar the internal grammar, in which one can prescribe things like physical grouping of ideas and bridges, encoding ideas or bridges in pointer structures, physical access paths or inversions etc. All the internal constructs are special encodings of ideas and/or bridges, and some may be semantically redundant, where both have the aim to minimize computer resources. The conceptual grammar deals with one construct, namely deep structure sentences, in which one can distinguish ideas and bridges. The internal grammar deals with other constructs such as internal records, which group various deep structure sentences together, pointer structures and place-near structures, which both represent a certain deep structure sentence, and access paths, which provide *fast highways* into the information base. Therefore it is necessary to describe how the conceptual constructs (deep structure sentences, or ideas and bridges) are transformed into internal constructs.

In figure 6.a we have represented the internal and the internal-conceptual transformation grammar. (Please note that it is not essential that figure 6.a contains only one user and application program, it is only done for ease of drawing).

The programming efficiency aspects are introduced via the EXTERNAL axiom:

The program view of a subset of each information base can be completely prescribed in one single grammar.

We call this grammar the external grammar, having such constructs as logical records, normalized relations, navigational access path, etc. Just like for the internal grammar, we need a transformation from the conceptual to the external constructs and figure 6.a contains the information systems framework covering problem description aspects or information analysis

(= conceptual grammar), computer efficiency aspects (= internal grammar) and programming aspects (= external grammar).

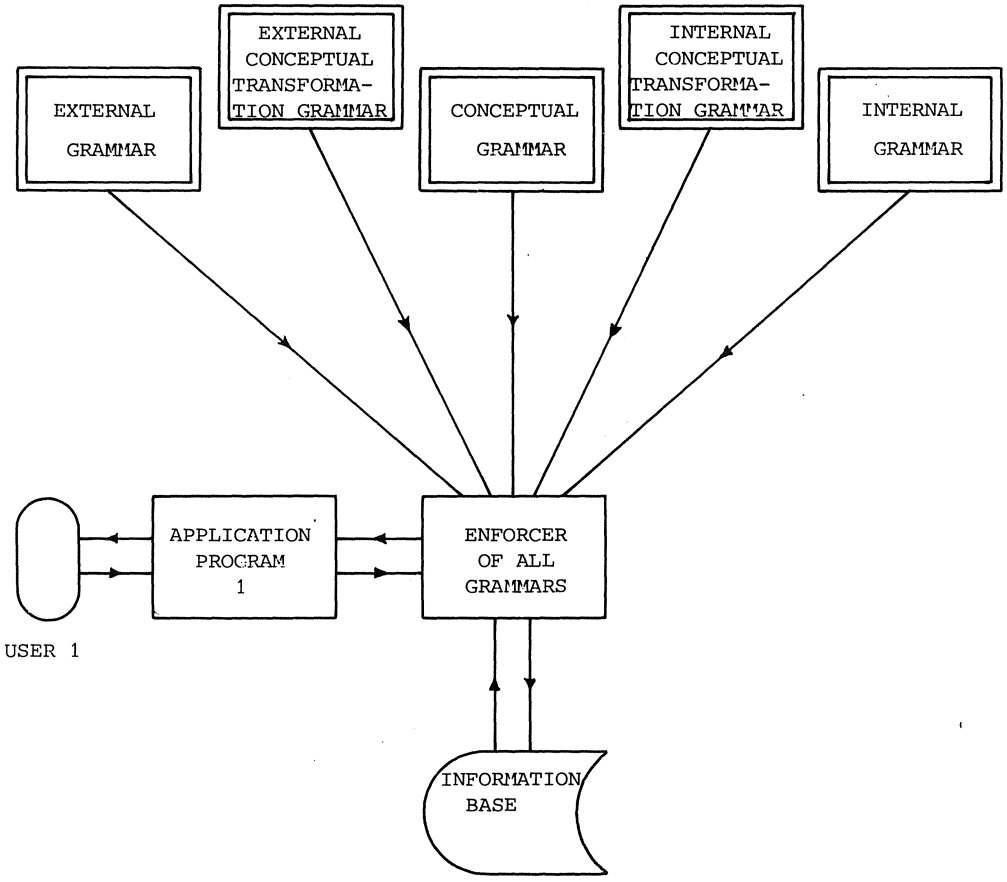


Figure 6.a

It is essential in this approach that the enforcer is solely responsible to keep the information base, the physical information base (called *data base*?) and the part of the information base as presented to a program (called *external base* ?) in accordance with the various grammars. Said otherwise, the application program has no responsibility with respect to these grammars.

In NIJSSEN [6], this framework for information systems has been given the name COEXISTENCE architecture, because it permits the coexistence of various data models, such as CODASYL, Normalized Relations or Binary at the external level.

7. THE META AXIOM AND ISDIS

In this section we will first investigate how the conceptual grammar, or information base grammar can be updated, and how such an update relates to the update of the informationbase, and thereafter we will do the same for all other grammars; this refers to the third aspect mentioned in the summary. Let us start with figure 7.a in which we have represented the same information systems as in figure 6.a, but the grammars associated with the INTERNAL and EXTERNAL axiom have been represented in one grammar symbol.

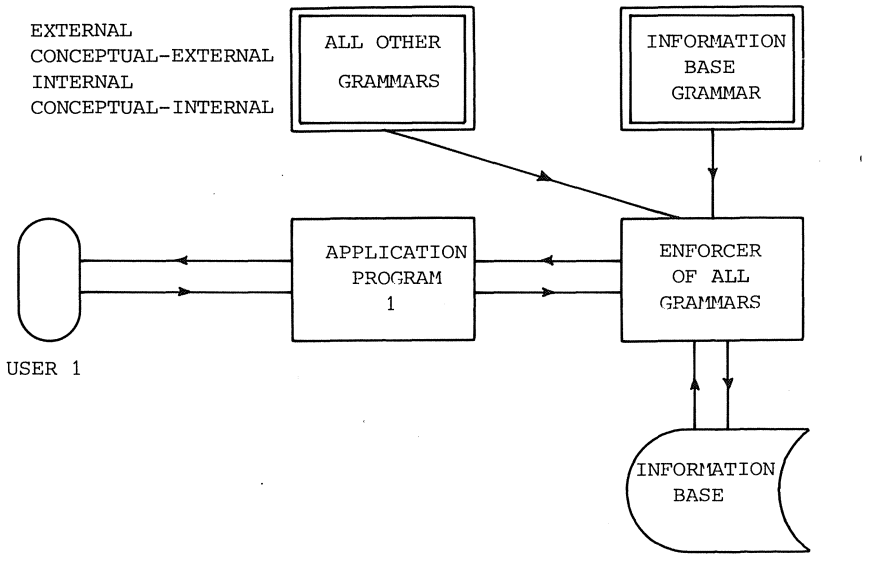


Figure 7.a

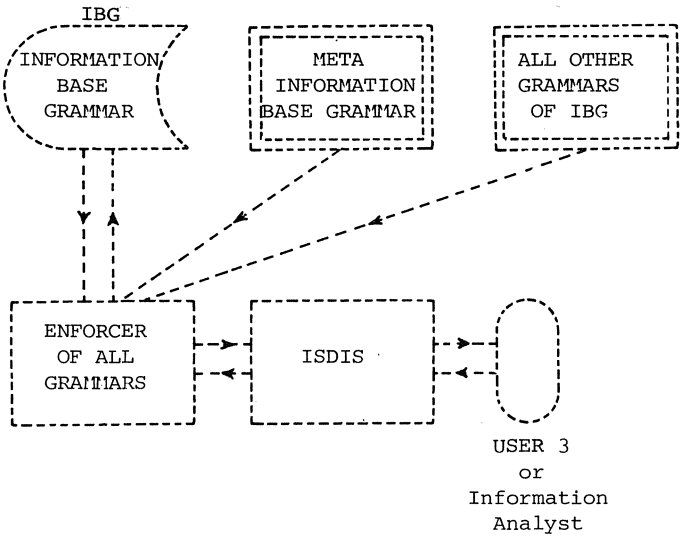


Figure 7.b

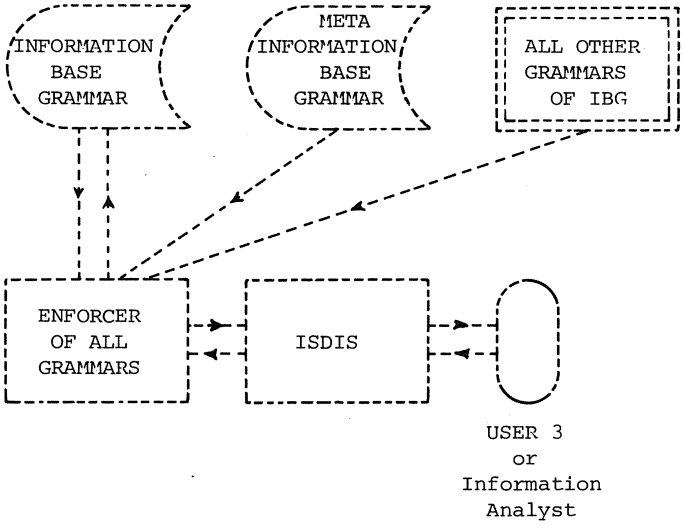


Figure 7.c

One can now pose the question: how can we update the information base grammar?

Before we answer this question, we state the META axiom:

All grammars of the Coexistance information systems framework can be considered as information bases.

If we apply this axiom to the information base grammar, or conceptual grammar, we get a picture like figure 7.b. Here we can see that the user can update and retrieve the information base grammar just as if it were a normal information base. The application program in this case has a special name, ISDIS, which stands for Information Systems Design and Implementation System. One of the functions of this application program is to help the user in the definition of the information base grammar. But according to the INSYGRAM axiom there must be a set of rules which completely and exclusively prescribes all the permitted information base grammar transitions. If we now apply this to the framework of figure 7.b we get figure 7.c.

The information base grammar is sometimes called the meta information base, and consequently, the meta information base grammar is sometimes called the meta meta information base.

It is clear that we now can also apply the META axiom to the other grammars and then we get figure 7.d.

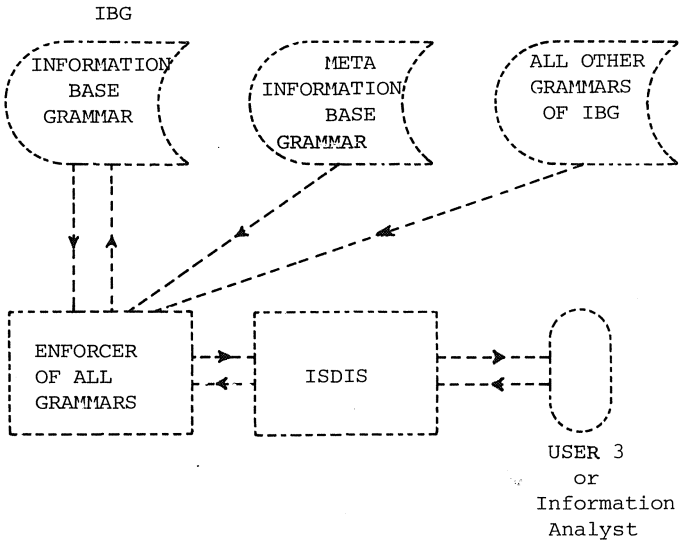


Figure 7.d

For a better understanding, it is important to note that the information base grammar of figure 7.a is the same as the information base grammar of figure 7.b, and the enforcer of figure 7.a is the same as the enforcer of figure 7.b, 7.c and 7.d. Hence we should integrate the information systems represented in figure 7.a and 7.d and then we get figure 7.e.

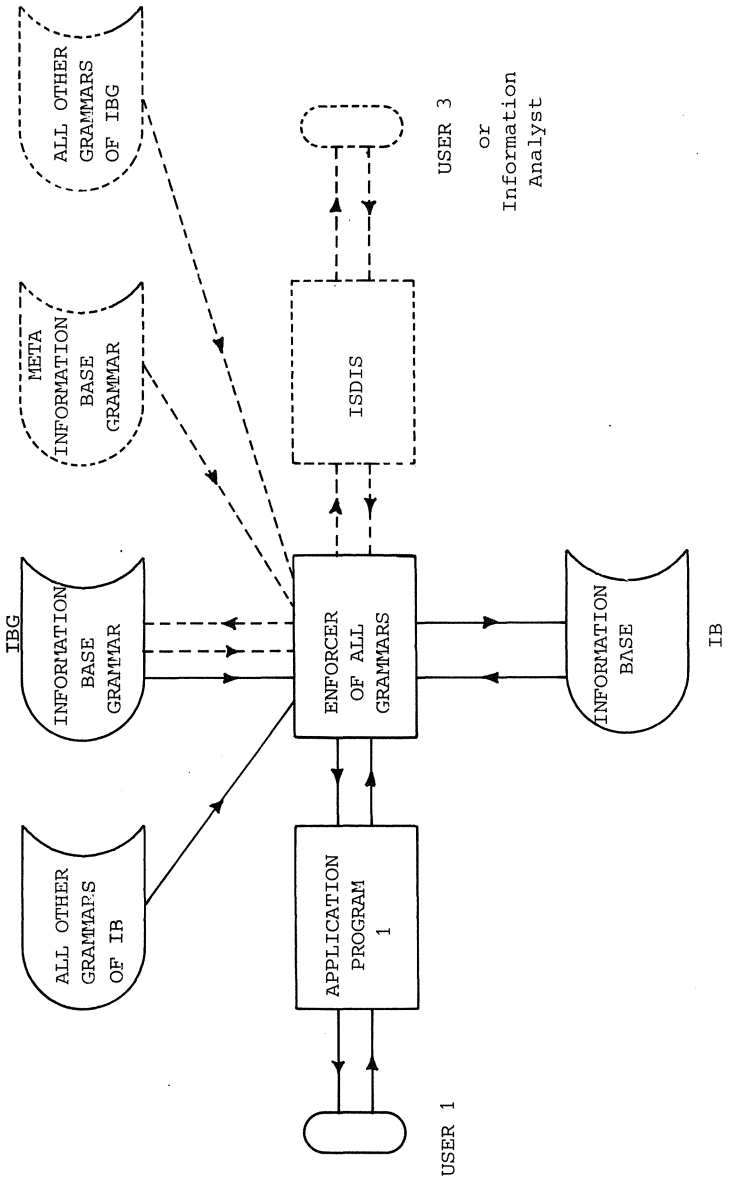


Figure 7.e

Figure 7.e is in our opinion a blue-print for the next generation of integrated data-base-data-dictionary software. This can be seen if we consider that a dbms (data base management system of today) is a predecessor of the enforcer of figure 7.a, while the dd (data dictionary) of today is a predecessor of the ISDIS of figure 7.b.

The consequences of the META axiom is an integration of these two things, the dbms and dd. This way of integration means that the same reasoning that is applied to convince users to go to databases, is now applied to the information systems as used by the experts in the information systems department to perform their own job.

8. WHY ARE APPLICATION PROGRAMS SO COMPLICATED TODAY?

It is almost a general consensus that current application programs are complicated with the consequences of high costs of development and maintenance and poor reliability. Is this understandable and what can we do about it? In the previous sections we have laid down the groundwork to answer these questions.

An application program today is so complicated because of the fact that it is not insulated like in our approach (see figure 6.a) where it can concentrate on its real function. An application program of today is a mixture of several functions; it must guarantee that the information base is in accordance with a large part of the information base grammar (see section 4), it must also deal with computer efficiency aspects, as well as the transformation of external grammar to internal grammar, in addition to its real function. Like in most situations where persons have to tackle a complicated problem, we apply intellectual tools such as abstraction. In the case of current application programs, this abstraction is made impossible because of the fact that current software forces the application programmer to deal with programming efficiency, information base correctness, computer efficiency and external-internal transformations at the same time.

This problem simply does not exist if one applies the information systems framework of figure 7.a.

9. SUMMARY AND CONCLUSIONS

In this paper we have presented an approach to information systems which is based on six axioms, called ENALIM, INSYGRAM, SENE, INTERNAL, EXTERNAL and META. We have indicated that it is our belief that the information systems framework based on these axioms is a good candidate for the next generation of information systems support software. Since 1973 we have built various prototypes of data base and data dictionary software, progressively covering more of these 6 axioms. The results obtained in various practical large scale projects have encouraged us to go on in this direction.

The use of advanced mass storage can often be substantially improved by applying this information systems framework.

ACKNOWLEDGEMENT

The author would like to acknowledge many interesting discussions on the approach described on this paper with E. Falkenberg and M. Senko, the members of IFIP WG 2.6, and the members of the Database Management Research Laboratory at Control Data Brussels, and Consultants from Products Design at Control Data Sunnyvale.

REFERENCES

[1] ANSI, *The* ANSI/X3/SPARC DBMS *Framework,* Report of the Study Group on Data Base Management Systems, edited by D. Tsichritzis and A. Klug; AFIPS Press (1977).

[2] CODASYL, DDL Journal of Development (1978).

[3] FALKENBERG, E., *Significations: The Key to Unify Data Base Management,* In Information Systems vol. 2, nr. 1 (1976), pp. 19-28.

[4] ISO, *Concepts and terminology for the conceptual schema.* ISO TC 97/SC5/WG3 (to be published).

[5] KENT, W., *Data and Reality,* North-Holland Publishing Company, Amsterdam Amsterdam (1978).

[6] NIJSSEN, G.M., *On the Gross Architecture for the Next Generation Database Management Systems,* In: Information Processing 77, Proceedings of the 1977 IFIP Congress, Toronto, Canada, North-Holland Publishing Company, Amsterdam.

[7] NIJSSEN, G.M., *Modelling in Data Base Management Systems,* Euro-IFIP 1979, North-Holland, Amsterdam (1979).

[8] SENKO, M.E., *Conceptual Schemes Abstract Data Structures, Enterprise Descriptions,* In: International Computing Symposium 1977, Proceedings of the International Computing Symposium 1977, Liège, Belgium, April 4-7, published by North-Holland Publishing Company, Amsterdam (1977).

[9] WITTGENSTEIN, L., *Philosophical Investigations.*

ERIS: AN EXPERIMENTAL RELATIONAL INFORMATION SYSTEM

H.M. BLANKEN, O.J.J. BROENINK,
R. ENGMANN, A.H. HAITSMA
Technische Hogeschool Twente

1. INTRODUCTION

Since Codd's original paper on relational data bases (CODD [7]), a number of papers have appeared which describe architectures and implementations of relational data base systems (ASTRAHAN [2], HELD [10], HUTT [11], MCLEOD [13], TODD [17]).

In 1976 a project at the Technological University Twente was started with several aims. Firstly it should be a project in which students could participate and get experience as part of their education in system design and implementation. Secondly the teaching staff should get experience with the management and control of an on-going project in which the main part of the production was carried out by students, who are only a relative short time member of the project team. Thirdly - and now we come to the more technical issues - it should explore some novel ideas about system design and implementation and the possibilities which SIMULA 67 offers to carry out those ideas. Lastly its target should be a relational data base system pursueing the architectural ideas laid down in the so-called Gamma-0 interface (BJORNER [5]) and putting them into practice.

The project has yielded a relational data base management system called ERIS (Experimental Relational Information System) with full query capability based on a relational algebra type language. Techniques for optimisation of algebraic expressions are included in the transformation of user supplied data requests (queries) into lower level programs, interfacing with the stored data base.

The design of the system is based on the principles of hierarchical structuring. In the hierarchical structure succeeding levels represent levels of increasing abstraction of design and implementation details. The SIMULA 67 CLASS concept (BIRTWISTLE [4]) enabled us to apply these design principles rigorously.

The system is primarily a single-user system. Emphasis is placed on data modeling and user guidance with respect to semantic constraints on the data. The system has been implemented on a DECsystem-10.

2. GLOBAL ARCHITECTURE

2.1. The data sublanguage REAL

The main vehicle for communication with the system is the data sublanguage REAL. In this section we will not give a complete overview of the language but only describe its main functional capabilities. An extensive example of REAL is shown in the appendix.

REAL consists of two parts: the data definition language (DDL) and the data manipulation language (DML). The data definition language is primarily meant for the definition of new data bases (see 2.1.) and the modification of existing data base definitions. A schema or data base definition consists of a description of the domains and the n-ary relations defined over those domains, and instructions with respect to the storage representations of domain values and access paths to be built in the system internal model, such as index tables and links.

The data manipulation part of REAL is a relational algebra type language (CODD [8]). Besides the operations of the relational algebra, like selection, projection, join, division and set operations it contains operations which allow the inclusion of data from sequential files external to ERIS data bases into system defined tables. Hence the data manipulation part of REAL allows a high level retrieval and data base maintenance capability.

2.2. ERIS-data bases

An ERIS-data base is a named collection of relations. Many ERIS data bases can coexist at the same time. However only one data base can be accessed by a specific user at the same time. Hence extraction of data belonging to different data bases in one query is not possible. A directory file contains for each data base the necessary information which the control program needs before opening a data base. Access to this file is based upon the data base name. Upon creation of a new data base this information must be supplied by the data base administrator (DBA) and is added to the directory file.

In the context of the operating system an ERIS data base is a random access file with a fixed record length. In ERIS terminology a record of the data base file is named a page. An ERIS data base contains both the user accessible relations (data) and the control information needed by the ERIS control program to access the user data. The control information is also stored in the form of relations, hence access to user data and control information presents a uniform picture.

2.3. Main architectural levels

It has been customary since the appearance of the ANSI/SPARC report (ANSI [1]) to describe the architecture of data base systems in terms of the three level schema architecture or at least to compare it with this proposed architectural framework. In ERIS the three levels, conceptual, internal and external, are represented by the system relational, the system internal and the external data model, respectively.

2.3.1. The system relational data model

The system relational data model is the central view of the data base system. It consists of a collection of relations which are stored as such in the ERIS data base. Relations are defined by means of the data definition part (DDL) of the REAL language. Definition of a relation creates an empty relation. Proper information is entered into the dictionaries and a skeleton relation is built. Next the relation can be loaded using standard REAL operations.

Before a relation can be defined the fundamental domains on which it is defined must be known to the system. Fundamental domains are also defined in the data definition part of REAL. A fundamental domain is considered as a potential set of values which may occur in the corresponding domains of relations. A fundamental domain definition consists of the domain name, type, value, range and format.

The definition of a relation specifies its name, attributes and primary key. The definition of an attribute of a relation specifies the fundamental domain to which the attribute values belong. The attribute name is the role name of the specified fundamental domain. The range and format of an attribute are - if not specified in the attribute definition - identical to those of the corresponding fundamental domain. However, they may be specified to be more limited in the attribute definition, e.g. the range of an attribute may be smaller than the range of its fundamental domain. Hence the definitions of fundamental domain and attribute have an important function in the validation control of data in the system.

The data manipulation part (DML) of REAL enables access to the data stored as relations. It is a relational algebra type language and contains set operations as well as the relational operations projection, selection, join and division. The result of a REAL query statement is a set of tuples (a relation). This relation may be stored as a file with a user-specified name or directly transmitted to the terminal from which the query was issued.

2.3.2. The system internal data model storage structures and access paths

For reasons of efficient use of storage two types of relations are distinguished: regular relations and string relations. Regular relations have a fixed number of domains. Each domain has a representation with a fixed length. Hence (variable) length strings with a length exceeding the domain length cannot be stored as domain values in domains of regular relations. To alleviate this problem string relations have been introduced. A string relation is a unary relation whose domain values are text strings. A domain in a user defined relation whose values are of type string is replaced in the corresponding regular relation by a domain whose

values are tuple identifiers, which point to the corresponding strings in the string relation in which the values of the original domain have been stored. Reference to values of type string necessitates accesses to string relations. On the other hand a certain amount of storage economy can be achieved by this storage scheme, because a recorded string value can be pointed to from many individual tuples. This may lead to a storage gain because a tuple identifier is generally shorter in length than a string value.

A page of an ERIS data base only contains tuples of one relation. Tuple ordering within a relation is system defined. An index is always provided on the primary key. When the primary key consists of more than one domain, the index is a so-called combined index. The index is implemented in the form of a B-tree. Each index is stored separately on a set of pages.

The DBA (Data Base Administrator) has the option to define deliberately additional secondary indexes on the domains of a regular relation. These are also implemented as B-trees and stored separately on sets of pages.

Another performance enhancing structure is the LINK. A LINK is a way to speed up access from a tuple in one relation, the parent relation, to a number of tuples in another relation, the child relation. The link criterion is equality of values in a specified domain in the parent relation and a specified domain in the child relation. The link is represented by a chain of tuples linked by tuple identifiers.

2.3.3. The external user data model

The system internal data model is the model created by the DBA to satisfy the information needs of the user community. An individual user may have a data model which is more restricted than the community model. His data model should be a subset of the system relational data model.

In the present version of ERIS the external user data model has not been incorporated, hence an external user interacts in the present version directly with the system relational data model.

3. THE STRUCTURE OF THE ERIS-SYSTEM

The ERIS-system has a layered structure in which on each level constructs are introduced, which are built from building blocks available at a lower level. These constructs are data structures and procedures which embody functions necessary to manipulate the data structures in a desired way. The choice of a consistent set of constructs at a specific level obviates the continued existence at that level of more primitive constructs available at a lower level. In a certain sense a careful leveling of a system can be considered as a means for stepwise abstraction from design and implementation details present at lower levels.

The ERIS system has been implemented in SIMULA 67. SIMULA 67, a derivative of ALGOL 60, has been designed primarily for the description of simulation processes. However it is applicable to a wide variety of problems in different environments. ALGOL 60 with some modifications is available as a sublanguage within SIMULA 67. The language is enhanced by inclusion of the types CHARACTER and TEXT, which allow the processing of alphanumeric data, reference types (pointers), and input/output facilities including those for handling sequential and direct access files.

The main feature present in SIMULA 67 but lacking in languages of comparable strength like ALGOL 60 and PASCAL is the CLASS concept. Together with the concept of the reference type, which allows the declaration and manipulation of references to data objects of various types, the class concept is a powerful tool for the design and implementation of hierarchical program structures (DAHL [9]) and complex data structures.

3.1. The CLASS concept

The class concept can be considered as a generalisation of the block concept of ALGOL 60. Whereas in ALGOL 60 a block instance ceases to exist when control is passed outside its statement part, a block instance in SIMULA 67 (in the form of a class instance) is permitted to outlive its calling statement. Whereas in ALGOL 60 a calling program can only observe the results of the actions of the block which it created at the time of its disappearance by inspection of the parameters passed to it, the attributes

of a class instance remain available to the calling program. Attributes of a class instance are the parameters passed to it and objects (data, procedures, functions and classes) declared within its body. In particular procedures and functions local to a class instance may be invoked from outside the class body.

The concept of a subclass is important for the creation of a leveled structure by means of SIMULA 67 classes. A block or class declaration may be prefixed by the name of a class declared in the same block or its prefix, e.g.

A CLASS B .....;

This clause specifies that an instance of class B is a compound object which has all the attributes of the prefix A in addition to its own attributes according to the declaration of class B. The actions of such an instance are those of A concatenated with those of B. Class B is called a subclass of A.

Attributes of class B possibly inherited from class A may be masked. An attribute declared PROTECTED is only visible inside the class in which it has been specified PROTECTED and inside the body of subclasses of this class or the body of blocks prefixed with this class or a subclass of it.

An attribute may be declared HIDDEN in the same class or a subclass of it if it has been declared PROTECTED. An attribute declared HIDDEN is invisible in subclasses of the class where the HIDDEN specification occurs and in blocks prefixed with this class or subclasses of it. The combined specification HIDDEN PROTECTED thus isolates an attribute and makes it only visible in the very body of the class where this specification occurs.

As an example we consider the following skeletons of class declarations:

```
CLASS L1;
PROTECTED P;
HIDDEN PROTECTED D;
```

```
        BEGIN
            PROCEDURE P;
            .......
        END ;

    L1 CLASS L2;
        PROTECTED  Q;
        HIDDEN P;
        BEGIN
            PROCEDURE Q;
            .......
        END ;

    L2 CLASS L3;
        BEGIN ...
            Q;
            .......
        END;
```

In class L1 D is the name of a data structure on which procedure P operates. In class L2 procedure P can be used as a port of entrance to class L1, but the data on which P operates are invisible within L2. In class L3 procedure Q is used as a port of entrance to class L2. Q is supposed to be implemented with procedure P as a building block. By the HIDDEN specification with respect to P in class L2 this procedure is again invisible within class L3.

In this way a leveled structure can be built in which each higher level is represented as a subclass of the next lower level. By means of the HIDDEN PROTECTED specifications and the scope rules for nested blocks the interfaces between the different levels can be fully controlled. The sequence of prefixes of a chain of class declarations is called the prefix chain. In the example above L1, L2 is the prefix chain of L3.

A hierarchical structure implemented by means of a class concept along the lines sketched above, forms a tree structure whose leaves are (sub)classes which do not appear in the prefix of an other class and the internal nodes classes which appear as prefixes of another class. The root

of the tree is the common base or lowest level of the structure. A level can now be defined as the union of classes with identical prefix chains. For example in fig. 1 the level consisting of L3 and L4 has L1, L2 as its prefix chain.

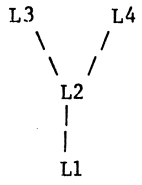


Fig. 1.

In the ERIS system the action part of the class bodies constituting a level is void. A level is represented by a set of procedures and data structures upon which these procedures operate and which can be invoked by the next higher level. Considering the leveled structure as a set of nesting blocks, passing from an outer to an inner block corresponds to passing from a lower to a higher level. The action part of the innermost class body embodies the actions necessary to respond to the signals from outside, such as query requests from users.

The first activity which has to take place in order for the system to become active is the proper initialisation of the different levels, including data structures present at these levels. This is done by an initialisation procedure present at each higher level, which initialises the next lower level, i.e. the class by which it is prefixed. These initialisation procedures successively activate each other and create thus the data base machine, which a user is allowed to percept.

3.2. The main modules of ERIS

The main modules are shown in fig. 2 in a top-down fashion; It starts with the highest, most abstract modules and gradually moves to the lower, more detailed modules.

- The DDL processor.
This module accepts and processes a schema section, which is a sequence of REAL DDL statements containing a definition of a new data base or a modification of an existing data base.

- The DML-processor.
This module accepts and processes a DML section, which is a sequence of REAL DML statements. A DML section describes operations on the stored data and/or the exchange of data between the data base and external files using operations from the relational algebra.

- The internal schema processor.
This level is used for creation, maintenance and inspection of the internal schema, which is a stored description of relations, domains and access paths in the stored data base.

- The lexical scan procedures.
This level contains a set of auxiliary procedures which are used by the DDL and DML processors for reading and interpreting the REAL input text.

- The Gamma-0 interface.
This level realizes the interface for tuple-at-a-time operations on the stored data base. With minor modifications it is an implementation of the proposals laid down in BJORNER [5].

- B-tree management.
Index tables are implemented by means of B-trees. This level contains the necessary operations for the creation, deletion and maintenance of B-trees.

- Page management.
This level maps the data base into a direct file with fixed-size pages. It controls the exchange of data base pages between main storage and disk storage.

- Globals.
This level contains a small collection of system variables which are

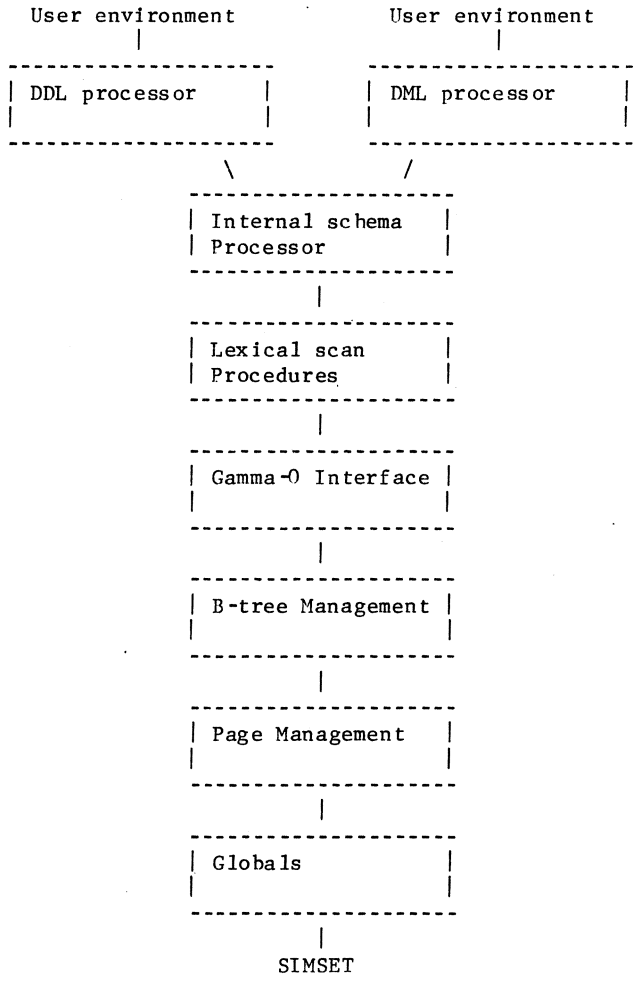


Fig. 2

global for the ERIS system.

The next chapter contains a more detailed description of the components of the data base system.

4. DETAILED DESCRIPTION OF THE MAIN MODULES

4.1. The DDL processor

The processing of a REAL program (i.e. a REAL section) is performed by either the DDL processor in case of a schema (DDL) section, or the DML processor in case of a DML section. A schema section concerns the data definition of a relational data base; a DML section expresses manipulation of the data stored in the data base. In the former case the effect of the processing will be the creation of an internal schema for a new data base or modification of the internal schema of an existing data base. In the latter case processing a DML section implies evaluation of the operations on stored data according to the DML section.

As there is a mutual independence between the processing of schema sections and DML sections the processing of a REAL section has been separated into two functional subprocessors, the DDL processor and the DML processor. An advantage of this separation is that for processing a specific section only a part of the complete system suffices, which relieves limitations on available central storage.

The DDL processor contains the analyzer for a schema section; a schema section concerns the data definition of a relational data base. The input for the DDL processor is a file containing a schema section according to the specifications of the REAL language. The processor checks the input for syntactic and semantic correctness. The text which has been parsed correctly and messages and warnings from the analyzer are stored into an output file.

According to the description of the language REAL there are two aspects of the language:

- the syntax consisting of a set of production rules by which all possible occurrences of the language can be generated; the syntax of REAL is LL(1);
- the semantics denoting precisely what is expressed by the language, including consistency rules which cannot be expressed in the syntax.

These aspects have been implemented by two dedicated sets of procedures: the syntactic procedures (recursive descent), which check for the syntactic

correctness of the DDL section, and the semantic procedures, which examine elements of a syntactically correct schema for consistency with the semantics. For reasons of both modular design and ease of implementation, the procedures have been collected into two separate subsystems, the syntactic analyzer and the semantic analyzer.

The syntactic analyzer contains the syntactical procedures according to the syntactic rules of a schema section. The information provided by the analysis of the schema section is stored in the intermediate data structures, which are contained in the internal schema processor.

The semantic analyzer contains semantic procedures which are called by the corresponding syntactic procedures in order to verify if a specific unit of the schema section, which has been parsed correctly agrees with the semantic rules. The semantic procedures take care of completion of the schema information with default values for items, which have not been specified in the schema section.

The semantic procedures may consult the internal schema by calling appropriate procedures of the internal schema processor. When the analyzer completes successfully the analysis of a schema section it calls the internal schema processor for creating or modifying the internal schema.

4.2. The DML processor

The DML processor contains the analyzer for a DML section and the modules which evaluate DML statements. The input for the DML processor is a file containing a DML section according to the specifications of the REAL language.

The syntactic analyzer contains the syntactical procedures according to the syntactic rules of a DML section. The semantic analyzer contains semantic procedures which are called by the corresponding syntactic procedures in order to verify if a specific part of the DML section, which has been parsed correctly, agrees with the semantic rules. The semantic analysis of DML statements will cause retrieval of data from the internal schema for checking the consistency of DML-statements with the data base schema. The data from the internal schema are provided by the internal

schema processor.

DML-statements may contain relational algebraic expressions which are optimized in order to minimize the number of accesses on secundary storage. The manipulation of relational algebraic expressions has been described by STROET and ENGMANN [16].

DML-statements are evaluated by the DML-processor. Generally evaluation of DML-statements involves consultation of the internal schema, e.g. for checking the existence of access paths and compatibility of input values to be stored into relations with specified types of domains.

DML-statements are evaluated using the facilities offered by the gamma-0 interface. Relational algebraic expressions are evaluated using appropriate algorithms. Selections on relations are calculated according to the methods described by ASTRAHAN and CHAMBERLIN [2]. Algorithms for calculating union, intersection and difference are rather straightforward; essentially the methods UNION1, INTER2 and DIFF2, respectively, as have been described by SMITH and CHANG [14] have been used. For calculating the equijoin method 1 of BLASGEN and ESWARAN [6] has been used, and for the relational division a variation of an algorithm of PECHERER [14].

In order to minimize the number of accesses on secondary storage subsequent operations on the same relations are combined into one step if possible.

4.3. The internal schema processor

The data base is selfdescriptive, such that it contains its own internal schema, which is a description of the relations, fundamental domains and access paths in the data base and the storage structure of data. The function of the internal schema processor is the creation and maintenance of the internal schema and retrieval of data from the internal schema on request of higher levels.

The functions of the internal schema processor are discussed in more detail in the following:

- The DDL processor may order the internal schema processor to create

or to modify the so-called intermediate data structures in main storage, which contain the data resulting from the analysis of a REAL schema section.

- When a REAL schema section has been interpreted correctly the DDL processor will order the internal schema processor to create or modify the internal schema. The relevant data needed for creation or modification of the internal schema are contained in the intermediate data structures which were produced by the DDL-processor during the analysis of a schema section.

The internal schema is represented by a set of permanent relations in the data base. The internal schema contains data about the contents of the data base, such as domain names, relation names, attribute names, access paths, representation and types.

The internal schema processor stores or modifies the internal schema in the data base. In this way access on the internal schema is equivalent to any other access on the data base and is implemented using the gamma-0 commands defined in the gamma-0 interface. The internal schema consists of the following relations:

1. DOMAINS:
   contains a description of the fundamental domains.

2. RELATIONS:
   contains a description of the relations in the data base, including their attributes.

3. LINKS:
   describes the links which have been defined for the data base.

4. STRINGS:
   contains one tuple for each fundamental domain, for which a string relation has been defined. Each tuple contains:
   string relation name,
   relation identifier of string relation.

5. CAT-NAM:
   string relation, containing names of relations, attributes and fundamental domains.

6. SYS-STR:
   string relation, containing restrictions and unit information on fundamental domains and attributes; this string relation is by default also used for storage of fundamental domains of type string which are not stored in a dedicated string relation.

The internal schema doesn't contain explicit data about the existing inversions. However, the internal schema processor can easily acquire such data from the master relation (see section 4.5), which is maintained by the gamma-0 interface. The internal schema processor also provides information on the degree, primary key domains and inversions of a specific relation.

- When an internal schema is created or modified the internal schema processor takes care of the creation or deletion of the declared relations, domains and access paths, e.g. when in the internal schema a new (regular) relation is defined the internal schema processor creates an initially empty relation with the specified name which subsequently may be filled with tuples of the proper format.
- Retrieval of data from the internal schema on request of the DDL processor or the DML processor. Consultation of the internal schema by the DDL and DML processor is necessary in order to enforce semantic consistency of data in a schema section of a DML section with the internal schema, and validation of input data. The processing of REAL ACCESS STATEMENTS (INSERT, UPDATE, DELETE and RETRIEVE) involves checking names of relations, values and types of attributes, uniqueness of key values, etc., in order to keep the contents of the data base consistent with the internal schema.

4.4. The lexical scan procedures

The lexical scan procedures are used by the DDL analyzer and the DML analyzer for scanning and interpreting the input stream which is read from the input file.

4.5. The gamma-0 interface

This level provides the so-called gamma-0 commands to higher levels. These commands include creation and deletion of relations, tuple-oriented operations and scans on relations. The term 'gamma-0' has been adopted from BJORNER [5].

The gamma-0 interface discerns four types of relations:

1. The master relation, which contains descriptions of all relations in the data base, including the description of the master relation itself. In a data base there is one master relation. For each relation in the data base there is one corresponding tuple in the master relation.

2. Regular relations; the degree of regular relations may range from 1 to 99. Items of tuples of regular relations are all of fixed size throughout the whole data base, so the storage representation of the tuples of a specific regular relation has a fixed size.

3. String relations provide facilities for storing domains which contain strings, which are too long to be stored in the fixed size items of regular relations. String relations are unary, i.e. relations with degree equal to 1; the tuples may contain character strings of arbitrary size.

4. Inversion relations or inversions are index tables on attributes of regular or string relations. The items are of fixed size.

The gamma-0 interface discerns three types of tuples:

1. Data tuples: tuples of master, regular and inversion relations, consisting of fixed-size items.

2. Control tuples: each relation contains one control tuple which is the first tuple of the relation. The control tuple contains coding information concerning the contents of the data tuples of

the relation.

3. String relation tuples or strings: tuples of string relations containing a string of characters of arbitrary size.

The gamma-0 interface maps relations onto fixed-size pages. Each page contains only tuples of one relation. Tuples of relations don't have a user-controlled ordering.

Tuples are identified by so-called tuple identifiers (abbreviation: tid). A tid is the address of a tuple in the data base consisting of a page number and a displacement within a page. Relations are identified by relation identifiers (abbreviation: rid). An rid of a relation is the tid of the tuple in the master relation corresponding to that relation.

For retrieving one or more tuples from a relation gamma-0 offers the scan facility. A scan on a relation is a system ordered reading of some or all of its tuples, subject to certain constraints, which are determined by the user. Each scan is identified by a scan identifier. It is possible to define several scans with different constraints on the same relation. All concurrent scans should have different scan identifiers.

The gamma-0 commands have been implemented by a set of procedures with corresponding names. The procedures are listed in the following:

create_relation
: function: creates a regular or string relation, which is initially empty except for the control tuple.

drop_relation
: function: deletes a regular or a string relation with existing inversions.

generate_inversion
: function: generates an inversion on one or more domains of a relation.

drop_inversion

function: deletes an inversion.

insert_tuple

function: inserts a new tuple into a regular or string relation.

delete_tuple

function: removes a tuple from a regular or a string relation.

update_subtuple

function: modifies non-key components of a tuple of a regular relation.

tid_from_string

function: given a string in a string relation this procedure returns the corresponding tuple identifier.

domain_from_tid

function: returns the value of one or more domains of a tuple in a relation.

create_scan

function: creates an instance of a scan on a master, regular or string relation.

set_scan

function: specifies the conditions for an existing scan.

next_subtuple

function: retrieves the values of one or more domains of the next tuple which satisfies the scan conditions.

drop_scan

function: terminates an existing scan.

fetch-mrid

function: when the system is initialized the rid of the master relation provides higher levels with necessary operational data:

the fetch-mrid command provides the initial entry to the data stored in the data base.

4.5.1. Tuple identifiers and relation identifiers

Relations and individual tuples should be mapped into pages of the data base. As several alternatives exist for this mapping and also for the identification of relations and tuples depending upon design conditions, the use and access on the data base was characterized as follows:

- the system should be able to answer non-standard queries,
- the volume of response data is small, i.e. the answer to a query consists of a small amount of data.
- processing of bulk data, e.g. creation and update of relations may be slow.

When a tuple is created a unique tuple identifier (tid) is assigned to it, which will remain unchanged during the tuple's existence. The following alternatives for tuple identifiers have been considered:

1. a tid does't contain information about the address of the tuple,
2. a tid specifies the number of the page where the tuple itself or a pointer to it can be found,
3. a tid specifies the address (page number and displacement) of a tuple.

The first case allows the tuples to be stored in ascending order of one or more attributes as e.g. the primary key, as individual tuples may be displaced when new tuples are inserted. For direct access of individual tuples an index table (tid, address) is needed, which makes direct access and insertions slow and occupies additional storage space.

The second case implies a table of address pointers in each page; an address pointer may specify a displacement of a tuple on the same page or an address of a tuple on another page. This organisation allows displacement of tuples and maintenance of the sequential order of tuples. Direct access may take two page accesses, the pointer tables occupy space on the data base pages and insertions are slow and complicated.

The third case allows fast direct access and is economical in storage space. As it is supposed that for each relation a primary index table is created and maintained sequential access is slow but still possible; however, fast sequential access is not required. This alternative has been chosen because it agrees with the requirements and can be implemented rather easily. Relations are identified by a relation identifier (rid) which in fact is a tuple identifier. An rid identifies a tuple in the master relation, so tuple identifiers and relation identifiers have the same format.

4.5.2. Storage of regular relations and tuples

Relations are stored on pages, which reside either in main or secondary storage. In order to be able to scan a relation with a minimum number of page accesses, each page contains tuples of only one relation. All pages of a specific relation are linked in a circular chain. For each chain of pages of a relation the number of the page in which most recently a tuple has been inserted, will be retained. Searching for free space will start at the same page, as this page is likely to have more free space available. The page number of the last insertion is stored in the corresponding description tuple in the master relation.

Tuples may be deleted by adding the occupied space to the chain of free locations or dummy tuples. In regular relations dummies are chained in arbitrary order and tuples and dummy tuples are distinguished by a tag, which precedes the tuple or dummy tuple. In this way free space is easily found and returned. Also scanning a relation is simple as tuples and dummies can be discriminated by the value of the tag.

4.5.3. Storage of strings

Regular relations accommodate tuples with a fixed format: all domains are represented by a fixed length field. Strings which may exceed in length the size of the domain representation should be stored in special unary relations, the so-called string relations. A tuple of a relation with string domains is represented by a tuple in the corresponding regular relation with tid's in the string domains which refer to the string relation tuples containing the string values. In this way only string

relations contain variable length tuples, which simplifies operations on regular relations. In addition a string in a string relation may be referred to by an unlimited number of tuples in regular relations. A string relation has an index table with entries containing a hash value calculated from the string and a tid. More than one string may correspond to the same hash value. When searching a string this implies that more than one tuple of the string relation may have to be retrieved before the correct string is found. However, this is not a serious drawback of the method.

4.5.4. Storage of string relations

String relations are unary relations which contain alphanumeric strings which may be of variable length. A string or tuple should be preceded by its length and by a reference count. When a string is inserted, which is already present in the relation, its reference count is increased by 1. Deletion of a string results in decreasing the reference count by one. If the reference count becomes 0 the string is effectively deleted and changed into a dummy. Dummies are linked together in a dummy chain; so a dummy should contain its length and the displacement of the next dummy on the same page. Dummy strings which are too small to accommodate these data are considered as fillers of adjacent strings. Each dummy is preceded and followed by a tag which allows a simple method for concatenating adjacent dummy strings. A direct access path to strings stored in string relations is provided by an inversion which is a standard provision. Entries in the inversion table have the format: (hash value of string, address (= tid) of string).

4.6. B-tree management

The B-tree management implements inversions, i.e. index tables on relations. An inversion provides a direct access path: given a specific value of one or more domains it yields the corresponding tuples. Inversions are represented by B-trees (KNUTH [12], WEDEKIND [18]).

The following procedures may be called:

createbtree

function: creates a B-tree using a set of (value,tid) tuples.

deletebtree

function: deletes a B-tree and returns all its nodes to the free_page pool.

insertvalue

function: inserts a new (value,tid) tuple into an existing B-tree.

deletevalue

function: deletes an existing (value,tid) tuple from a B-tree.

retrievetid

function: for a specific value the corresponding tid is retrieved from the B-tree.

nextfrominversion

function: successive calls to this procedure yield a collection of tid's whose value parts range between two given values.

Though inversions may be considered as a kind of relations, only a limited set of operations is permitted. Because the entries of inversions are kept in the order of increasing values of the inverted domains, the storage address of the entries is not stable. For this reason tid's are undefined for inversions.

4.7. Page management

The data base is represented by a direct file, which is stored permanently on disk storage. The direct file contains fixed-size records or pages. The page is the unit of transfer between main storage and secundary storage. The size of the page in characters, page_length, is a constant value for a specific data base. The value of page_length should be given at the time of creation of the data base.

As files in SIMULA usually are text files a page is represented by a character string. The size in characters of a page is equal to the value

of page_length. A small number of pages can be mapped into main storage in an internal buffer; the size of the buffer is expressed in page frames, i.e. the number of pages, that fit in the buffer, such that a page frame may contain one page. The buffer consists of a set of 'no_of_frames' page frames.

Page management contains procedures for mapping pages from secundary storage into the internal buffer and controlling the exchange of data between secundary storage and main storage.

The following procedures are available:

get_page

function: accesses an existing page and transfers it into the internal buffer.

get_from_freepagepool

function: new pages can be obtained from the so-called free-page pool; likewise obsolete pages may be returned to the free-page pool. The procedure get_from_freepagepool delivers a new empty page to the data base and makes it available in a frame in the internal buffer.

return_to_freepagepool

function: returns a page to the free-page pool.

free_page

function: a page is released such that it may be removed from the internal buffer.

setmodifier

A specific page in the internal buffer is marked when its contents is changed. A page which has been modified is rewritten to secundary storage after a free_page operation on it. A page which has not been modified will not be rewritten; the system will overwrite its pageframe, when it is released by a call to free_page.

4.8. Globals

Globals is the lowest level of the ERIS system, which is prefixed by the standard SIMULA class SIMSET. It contains a small collection of global variables which are used by several higher levels:

BOOLEAN new_database:
: has the value FALSE if the present session refers to an existing data base, TRUE if the data base will be created during the present session.

TEXT database_name:
: the name of the data base as it will be stored in the directory file of data bases.

INTEGER page_length:
: the size of the pages of the data base in numbers of characters.

INTEGER pagno_length:
: the number of characters necessary to express the ordinal number of a page in the data base. The value is calculated from the maximum number of pages to be expected in the data base as will be specified by the owner of the data base: pagno_length = ilog(maximum number of pages), where ilog(x)=entier(log x)+1.

INTEGER no_of_frames:
: the number of frames (pages), which fit in the internal buffer.

INTEGER domain_length:
: The number of characters available for a single value in a data base tuple. The value domain_length is uniform for all domains and all tuples in a specific data base.

INTEGER tid_length:
: The number of characters available for a tuple identifier: tid_length = ilog(page_length) + pagno_length.

The data base parameters database_name, page_length, pagno_length, domain_length and tid_length are fixed at creation time of the data base and remain constant during its existence. The session-dependent parameters are new_database and no_of_frames. They have to be specified when the data base is opened.

4.9. The user environment

The ERIS system may be used by an external user program. In this case the body of the user program should be prefixed with the highest ERIS class necessary for a specific application. For processing a schema section containing DDL statements this would be the DDL processor; for processing a DML section this would be the DML processor.

DML-statements may also be incorporated into SIMULA 67 programs. A precompiler has been implemented which accepts such a program and transforms it into a SIMULA 67 program by replacing the DML-statements by procedure calls to the DML processor.

5. REALIZATION

Since 1976 in total about eight man years have been spent on the project. The size of the system is about 500 Kb in executable code. Expressed in SIMULA 67 lines the levels of ERIS have the following sizes:

- globals	20 lines.
- page management	600 lines.
- B-tree management	1150 lines.
- Gamma-0 interface	1900 lines.
- lexical scanner	1000 lines.
- internal schema processor	800 lines.
- DML processor	5100 lines.
- DDL processor	1700 lines.

In total the ERIS system contains about 12000 lines. The comment lines are not included in this figure. On the average about 70% of the lines are comment.

Compared to SYSTEM R (ASTRAHAN [2]) which amounts to about 4Mb of executable code ERIS seems to be small. However, SYSTEM R offers more facilities, so a comparison cannot be made that easy. The ERIS implementation is highly procedural which makes the system easily adaptable. During the execution of ERIS this results in many procedure calls, which reduces the execution speed.

ERIS has been documented extensively and the theses of students could be generated easily from this documentation.

6. CONCLUSIONS

The design and implementation of an experimental relational information system (ERIS) has been described. The project had an educational aim: students should participate in the design and implementation of a sizable system. In total ten students co-operated in the project.

A second goal was that the teaching staff should get experience in the management of a project which is large considering the environment. Five members of the staff worked during some time on the project.

The documentation is very extensive and was good enough to allow students to continue where other students stopped.

It appeared that SIMULA 67 offers good structuring facilities and these facilities contributed highly to the fact that finally a working system resulted.

The last aim (the use of the so-called gamma-0 interface) has been achieved almost completely; only minor modifications to this interface were made.

REFERENCES

[1] ANSI/X3/SPARC Study Group on Data Base Management Systems, Interim report, ACM SIGMOD Bulletin Vol. 7, No. 2, 1975.

[2] ASTRAHAN, M.M., e.a., System R: a relational approach to data base management, ACM Transactions on Data Base Systems Vol. 1, 1976, pp. 97 - 137.

[3] ASTRAHAN, M.M. & D.D. CHAMBERLIN, Implementation of a Structured English Query Language, Communications of the ACM Vol. 18, 1975, pp. 580 - 588.

[4] BIRTWISTLE, G. & J. PALME, DECsystem-10 SIMULA Language Handbook, Swedish National Defense Research Institute, 1974.

[5] BJORNER, D., E.F. CODD, K.L. DECKERT & I.L. TRAIGER, The Gamma-0 n-ary relational data base interface. Specifications of objects and operations, IMB Research Report RJ 1200, 1973.

[6] BLASGEN, M.W. & K.P. ESWARAN, Storage and access in relational data bases, IMB Systems Journal Vol. 16, 1977, pp. 363 - 377.

[7] CODD, E.F. A relational model of data for large shared data banks, Communications of the ACM Vol. 13, 1970, pp. 377 - 387.

[8] CODD, E.F., Relational completeness of data base sublanguages, in R. Rustin, Data Base Systems, Prentice Hall 1972, pp. 65 - 98.

[9] DAHL, O.-J. & C.A.R. HOARE, Hierarchical program structures, in O.-J. Dahl, E.W. Dijkstra & C.A.R. Hoare, Structured programming, Academic Press 1972, pp. 175 - 220.

[10] HELD, G.D., M.R. STONEBRAKER & E. WONG, INGRES a relational data base system, AFIPS Conf. Proc. Vol. 44, 1975, pp. 409 - 416.

[11] HUTT, A.T.F., A relational data base management system. PhD thesis, Southampton University, 1976.

[12] KNUTH, D.E., Sorting and Searching; The Art of Computer Programming, Vol. 3, Addison-Wesley 1973.

[13] MCLEOD, D.J. & M.J. MELDMAN, RISS: A generalized minicomputer relational data base management system, AFIPS Conference Proc. Vol. 44, 1975, pp. 397 - 402.

[14] PECHERER, R.M. Efficient evaluation of expressions in a relational algebra, Proc. ACM Pacific Regional Conf., 1975, pp. 44 - 49.

[15] SMITH, J.M. & P.Y.-T. CHANG, Optimizing the performance of a relational algebra data base interface, Communications of the ACM Vol. 18, 1975, pp. 568 - 579.

[16] STROET, J.W.M. & R. ENGMANN, Manipulation of expressions in a relational algebra, Information Systems Vol. 4, 1979, pp. 195 - 203.

[17] TODD, S.J.P., The Peterlee relational test vehicle - A system overview, IBM Systems Journal Vol. 15, 1976, pp. 285 - 308.

[18] WEDEKIND, H., On the selection of access paths in a data base system, in J.W. Klimbie & K.L. Koffeman, Data Base Management, North Holland 1974.

APPENDIX: Example of REAL

In a data base the following relations with their attributes exist (key attributes have been underlined):

supplier	(supplier-number, supplier-name)
part	(part-number, supplier-number, price, quantity-on-hand)
customer	(customer-number, customer-name, account-balance)
customer-order	(customer-order-number, customer-number, part-number, quantity).

The schema of the data base can be described as shown below. Here the pagelength is equal to 638, the maximum number of pages in the data base 100, the length of a domain 8 and the number of frames in main storage 5.

```
SCHEMA      NEW warehouse, 638, 100, 8, 5;

DOMAIN      number
            VALUE TEXT VARIABLE TO 6;
DOMAIN      name
            VALUE TEXT VARIABLE FROM 3 TO 25;
```

```
DOMAIN      quantity
            VALUE INTEGER FROM 0
            UNIT kg;
DOMAIN      amount
            VALUE REAL
            UNIT hfl
            ALT usd CONV 2.00
            ALT pound CONV 4.52;

RELATION    supplier
            ATTR supplier-number DOMAIN number
            VALUE FIXED 6
            ATTR supplier-name DOMAIN name
            VALUE VARIABLE TO 20
            KEY supplier-number;

RELATION    part
            ATTR part-number DOMAIN number
            VALUE PIC (A9(3))
            ATTR supplier-number DOMAIN number
            VALUE FIXED 6
            ATTR price DOMAIN amount
            VALUE REAL FROM 0.0
            ATTR quantity-on-hand DOMAIN quantity
            ALT ton CONV 1000
            KEY part-number;

RELATION    customer
            ATTR customer-number DOMAIN number
            VALUE PIC (9(6))
            ATTR customer-name DOMAIN name
            ATTR account-balance DOMAIN amount
            KEY customer-number;

RELATION    customer-order
            ATTR customer-order-number DOMAIN number
            VALUE PIC (9(6))
            ATTR customer-number DOMAIN number
```

```
                VALUE PIC (9(6))
                ATTR part-number DOMAIN number
                VALUE PIC (A9(3))
                ATTR quantity
                KEY customer-order-number;

INVERSION       customer-order ON customer-number;

INVERSION       supplier ON supplier-name;

END.
```

Examples of relational expressions

1. Customer-numbers of customers, who have ordered something:

   ```
   customer-order % customer-number
   ```

2. Part-number and supplier-name of suppliers of parts of which the quantity-on-hand is lower than 1000:

   ```
   (part : quantity-on-hand < 1000 * supplier) :
   part@supplier-number = supplier@supplier-number %
   part-number, supplier-name
   ```

3. Customer-numbers of customers who have ordered all parts:

   ```
   customer-order % customer-number, part-number
   [part-number / part-number] part % part-number
   ```

Example of a DML section

```
OPENDB    warehouse, 5;

INSERT    part
```

```
          FILE partfile
          RECORD CHAR(4), CHAR(6), REAL(6), INT(5);

UPDATE    part(part-number, quantity-on-hand)
          FILE transfile
          RECORD CHAR(4), INT(5);

DELETE    part(part-number)
          FILE delfile
          RECORD CHAR(4);

RETRIEVE  customer % customer-name, account-balance
          FILE customfile
          RECORD CHAR(25), REAL(10) UNIT usd;
```

A SIMULATION OF VIDEBAS ON A DEC-SYSTEM10

H.M. BLANKEN,
Technische Hogeschool Twente

1. INTRODUCTION

Data storage and manipulation have been recognised since many years as one of the most important tasks of computer systems. The requirements with respect to storage economy and retrieval speed varied much from one user environment to another, causing many methods to organize, store and retrieve data. After a short description of the relational model, the most important environments in which data(base) management systems operate are characterized in chapter 3. Starting with a simple batch environment we arrive, by increasing the requirements, at an on-line maintenance and retrieval environment. The corresponding data management systems evolve from a batch system to a relational database management system (VIDEBAS). The requirements which have to be satisfied by the hardware to make VIDEBAS feasible are concisely described.

As the required hardware is not present at the THT VIDEBAS is simulated on a DEC-system10. In chapter 4 a description of this (partly) finished system will be given.

In chapter 5 a tool will be described which allows the selection of an approximate optimal storage structure for a database, given the description of the load and the logical database. To implement this tool program modules of the VIDEBAS simulation on the DEC-system10 will be used.

Finally the research, which will be carried out at the THT will be outlined.

2. THE RELATIONAL MODEL

As VIDEBAS is a relational database management system, relations will be introduced right from the beginning. For more information about the relational model, see CODD [1]. A relation is a table in which each row (tuple) describes a real world entity. Each column value contains the value of an attribute of such an entity. Each tuple is unique, which means that no two tuples in the relation have identical values for all attributes. A minimal set of attributes, whose values identify the tuples form the key attributes. Relations will be assumed to be in first normal form, which means that each column of the table represents a simple attribute and is not a relation itself. A relational database is a collection of relations.

Table 1. Sample Database.

PARTS : PARTNO, DESCRIP, QOH, QOO
ORDERS: ORDERNO, PARTNO, SUPPNO, DATE, QTY

In Table 1 a sample database is described. It consists of two relations PARTS and ORDERS. For each relation the attributes are shown. Key attributes are underlined. The PARTS relation contains the description (DESCRIP), quantity-on-hand (QOH) and quantity-on-order (QOO) for a collection of parts identified by their part numbers (PARTNO). The ORDERS relation contains a set of outstanding orders with numbers ORDERNO. Part number (PARTNO), supplier number (SUPPNO), date (DATE) and quantity (QTY) are the other attributes.

Since the advent of the relational model numerous languages have been described for defining operations on databases. CHAMBERLIN, et al. [2] describe one of those languages, viz. SEQUEL, which will be used in this paper to describe data manipulation. To introduce SEQUEL we give a short example. "Find all orders to be supplied by supplier '797'". In SEQUEL this query could be written as:

```
SELECT      *
FROM        ORDERS
WHERE       SUPPNO='797'.
```

The SELECT clause lists the attributes, which define the result. The asterisk indicates that the result should yield all attributes. The FROM clause indicates the relation(s) to be used and the WHERE clause gives the condition which has to be satisfied by the tuples.

3. TAXONOMY OF ENVIRONMENTS

It will be shown how VIDEBAS will cope with each environment described below.

3.1. Relations without relationships

Deferred update with unconditional retrieval. - This is the conventional batch environment. The updates are collected and the relation is updated once per period (week, month etc.). Retrieval normally generates reports which are ordered on a certain criterion. An example of such a report can be formulated as:

```
SELECT      SUPPNO, ORDERNO, PARTNO
FROM        ORDERS
ORDER BY    ORDERNO.
```

The ORDER BY clause defines ORDERNO as the sort criterion. We assume a big, paged memory to store data. A stored relation will be called a file. When the file is sorted on a certain criterion it is in VIDEBAS denoted by SORTFL (sorted file). A SORTFL ordered on key facilitates the periodical batch update. When frequently retrieval of data is required ordered on different attributes then the existence of SORTFLs ordered on those attributes may increase efficiency by preventing sort operations. In VIDEBAS it is allowed to have many SORTFLs for one relation.

Deferred update with conditional retrieval. - If the condition concerns the key, then in VIDEBAS this environment will be handled by offering a SORTFL, ordered on key, with a directory file (DIRFL). This directory offers direct access to the SORTFL and contains a 2-tuple

< attribute value, page number> for each page of the SORTFL. The retrieval of the tuple with ORDERNO='7294' can be formulated in SEQUEL as:

```
SELECT      *
FROM        ORDERS
WHERE       ORDERNO='7294'
```

ORDERNO is key and with the value '7294' we access the DIRFL belonging to the SORTFL, ordered on ORDERNO. Here we find the page number of the required records. When the retrieval of ORDERS records, satisfying a condition on PARTNO is frequently needed then a SORTFL, ordered on PARTNO with a corresponding DIRFL may be advantageous. In this case the following query can be handled fast:

```
SELECT      *
FROM        ORDERS
WHERE       PARTNO='010002'.
```

Suppose in a VIDEBAS database a SORTFL exists for both the attributes SUPPNO and PARTNO. How to handle a query like:

```
SELECT      *
FROM        ORDERS
WHERE       PARTNO='010002'
ORDER BY    SUPPNO.
```

A strategy problem arises: is it the best way to select the ORDERS records which satisfy the PARTNO condition and sort them or to scan the SORTFL ordered on SUPPNO and to neglect all records not satisfying the PARTNO condition?

Immediate update with unconditional retrieval. - Suppose a report has to be derived frequently from a large relation and the report should contain up to date information. In VIDEBAS the updates to a relation are collected in a differential file, see SEVERANCE and LOHMAN [3]. Such a differential file is denoted by DIFFL and contains all updates since the last creation of the corresponding SORTFLs. A DIFFL record consists of a modification code followed by the attribute values. An insertion to a

relation results in a DIFFL record with code 'I', a deletion is defined by the code 'D' and the modification of a tuple results in two records, one which deletes the old contents and the other which inserts the new contents. Suppose the required report has to show the tuples ordered on a certain criterion. This report can be obtained by sorting the DIFFL on the required criterion and merging the resulting records with the SORTFL records.

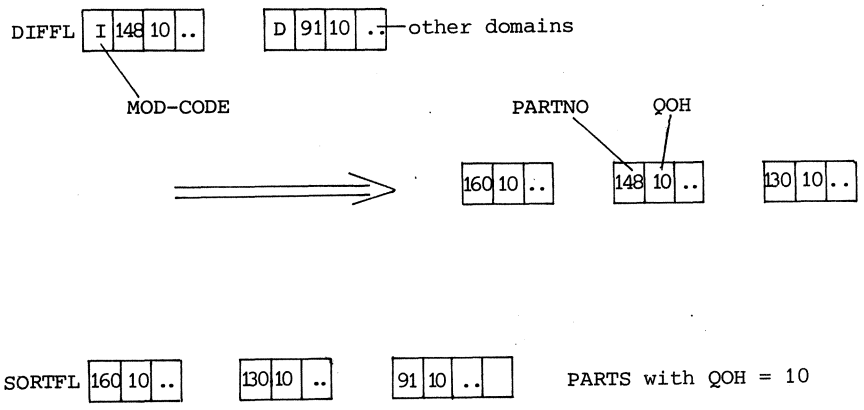


Fig. 1. The merge of SORTFL and DIFFL records.

In Fig. 1 an example of this merge process is given. All PARTS with a QOH value of 10 are given. When several reports are needed, each ordered on a different criterion then in VIDEBAS one DIFFL with several SORTFL files will be kept.

Suppose a new order for part '010002' and quantity '23' is placed with supplier '797'. This results in two update statements, namely an insert and an update.

```
INSERT          INTO ORDERS
'7294 010002 797 791012 23'.
UPDATE          PARTS
SET             QOO= QOO+23
WHERE           PARTNO= '010002'.
```

The INSERT will be handled as follows (it is assumed that no validation has to be done). First create a record for the DIFFL belonging to the ORDERS relation. The attribute values have to be set as requested and the modification code must be I. Store this record in DIFFL. The UPDATE generates a deletion of the old contents and an insertion of the new contents. Read the PARTS record with PARTNO='010002'. Store a record in the DIFFL file of PARTS with the old contents of PARTNO='010002' and a modification code D. Finally store a record in the same DIFFL file with a modification code I and an updated quantity on order Q00.

Immediate update with conditional retrieval. - To handle this environment VIDEBAS offers SORTFLs, DIRFLs for direct access and DIFFLs for immediate updates. When the conditions vary for different retrieval operations then more SORTFLs with corresponding DIRFLs, may be offered to facilitate query handling.

3.2. Relations with relationships

When relationships have to be taken into account then a means to connect relations has to be offered: the join operation. In this paper only equijoins will be considered, see Codd [1]. To explain the equijoin consider the query: "Determine for all orders which have to be supplied by supplier 797 the order number, part number and description". In SEQUEL:

```
SELECT      ORDERNO, PARTNO, DESCRIP
FROM        ORDERS, PARTS
WHERE       ORDERS.PARTNO= PARTS.PARTNO AND
            SUPPNO='797'.
```

It is possible to select all ORDERS with SUPPNO='797' (see above). Each ORDERS tuple has a PARTNO and that number can be used to retrieve the concerning PARTS tuple. So, using the existing VIDEBAS 'access methods' an equijoin is implemented, which 'upgrades' VIDEBAS to a simple relational database management system.

Summarizing: based on a sequential and a direct accessmethod (to scan SORTFLs and to fetch directly a certain page from a SORTFL using a DIRFL)

VIDEBAS can cope with a conditional retrieval environment. DIFFLs are introduced to collect updates and sorting of DIFFLs is needed to offer conditional retrieval in an on-line maintenance environment. To handle relationships an equijoin method is offered which uses the described accessmethods and sorting (of the needed DIFFL files). The result is a simple relational database management system.

An important assumption underlying VIDEBAS is that external memory and processing power become inexpensive. Many SORTFLs for one relation are offered sometimes, implying a large charge on external memory. To achieve a good performance it is important to obtain a fast, 'intelligent' (and small) memory (FASTINT) to store the DIFFL. Roughly stated this intelligence comprises associativity and sorting. For more details see the DIFMAN interface which will be explained below. A simple performance analysis shows, assuming such a memory, some promising results, see BLANKEN [4]. A more detailed analysis, however, has still to be performed. VIDEBAS offers possibly other advantages: backup is easier as only differential files, which are small, have to be backed up. Recovery can be implemented very efficiently caused by the differential file approach. Moreover, concurrency can be achieved to a high degree as FASTINT is fast and intelligent. See for more details BLANKEN [4] and VERHOFSTAD [5].

4. SIMULATION ON DEC-SYSTEM10

Since september 1979 we are implementing a VIDEBAS system on the DEC-system10 of the THT. The goal of this simulation is twofold, namely to get more insight in the problems concerning the implementation of a database system with an architecture like VIDEBAS and to obtain information which can be used to execute a more detailed performance analysis. In Fig. 2 the architecture of the simulated system is given.

4.1. Access to VIDEBAS

The users of VIDEBAS access the database by terminal and can be distinguished into two categories: the 'normal' users and the database

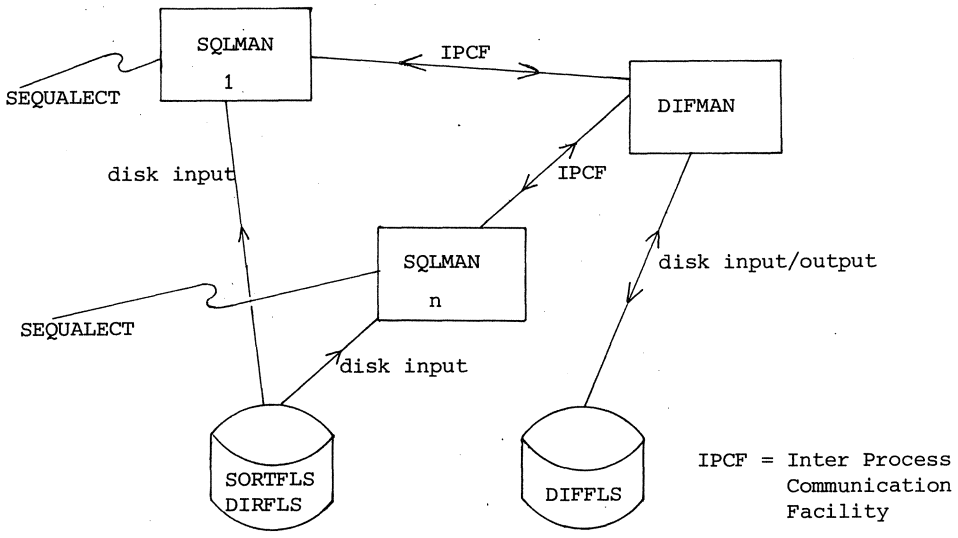


Fig. 2. Architecture of VIDEBAS simulation

administrator. The former category uses SEQUALECT (see appendix 1), which consists of a subset of SEQUEL and the statements DO and UNDO. The subset of SEQUEL offers data manipulation and retrieval. VIDEBAS is a multi-user system, but creates a single user environment for a user, which means that except for the statements DO and UNDO the user is not aware of other users. When a user starts the system, he will be asked whether or not he wants to update the database. In update mode VIDEBAS will look after the locking (automatic locking). All updates to a database are temporary until the user issues a DO statement, which causes the updates to be permanent. By means of an UNDO the temporary updates are destroyed again.

The database administrator is able to create and delete permanent relations and to control the storage structures of a database: adding and deleting SORTFL files, etc. The offered language is called SYSLAN, see appendix 2.

4.2. Mapping of VIDEBAS on the DEC-system10

A VIDEBAS database. - It is possible to implement a VIDEBAS database by one (probably big), direct file. A disadvantage of this approach is that VIDEBAS itself has to keep track of the pages which belong to a

certain file (SORTFL, DIRFL or DIFFL), of the free pages of the database, etc. It is easier to allocate one DEC file to each SORTFL, DIRFL and DIFFL. As PASCAL will be used the files will be character files, so attribute values are character strings. The allowed data types are numeric and alphanumeric character strings.

Timeslicing. - When more users access a database timeslicing is needed. Two possibilities are considered. The first is to offer timeslicing in VIDEBAS itself, which requires much programming (queueing, priority handling etc.), so this alternative has been rejected. The second and selected way uses the facilities as offered by the DEC-system10. Programs started by users to access a VIDEBAS database get timeslices according to the algorithms of the DEC-system10.

Concurrent updating. - How can the database be updated simultaneously? The updates of a VIDEBAS database are stored in DIFFLs and all DIFFLs are allocated to FASTINT. A problem arises as no FASTINT is available on the DEC-system10, requiring simulation of that medium. FASTINT has a memory and a processing (intelligence) function. The memory function is simulated using normal DEC files; intelligence is offered by the DEC-10 processor and some procedures. A DEC file (so a DIFFL) may be read but not updated simultaneously. A solution might be to surround each update operation with an OPEN and a CLOSE statement. In this way the DEC-system10 controls the access to the DIFFL files. This is, however, inefficient. A second way is to introduce a special program which is part of VIDEBAS and which controls the access to the DIFFL files. This special program, the differential file manager DIFMAN, receives instructions from other parts of the VIDEBAS system. The DEC-system10 offers facilities for programs to communicate. The execution of DIFMAN implies a process, which controls the access to a shareable resource: the DIFFL file(s). When n users use the database, then n DEC jobs, which execute the SEQUALECT manager SQLMAN will be present, and only one DIFMAN job (see Fig. 2). SQLMAN receives the user input (=SEQUALECT statement), analyses it using the database catalog, builds an intermediate data structure, calls the optimizer to determine the way of statement handling and uses DIFMAN to execute the statement. See appendix 3 for the definition of the DIFMAN interface.

Locking. - VIDEBAS is a multi-user system which allows concurrent readers and writers. Readers always get immediate access to the database while writers may have to wait sometimes due to locking. All data read or written by a writer is locked automatically by VIDEBAS on a relation at a time basis and these locks are held until a DO or UNDO is issued. These statements cause the release of the locks. The locks are administered by DIFMAN, see Blanken [4].

Maintenance of storage structures. - The database administrator uses the language SYSLAN, which is offered by the program SYSADM (system administrator). SYSADM has to update the database catalog (see below): relations are created and deleted, SORTFLs are added and destroyed, etc. In the beginning of its execution SQLMAN reads parts of the database catalog in internal tables, for instance to check the validity of relation and attribute names. When SYSADM is used the catalog will be updated and no SQLMAN may be active.

4.3. Database catalog

The administration of VIDEBAS is stored in three relations (the 'meta database'). The first relation (AAAREL) stores all information about the relations, the second (AABATT) contains information about the attributes which occur in the database and the last (AACSRD) gives SORTFL information.

AAAREL. - The attributes are the relation name and identifier, which is an internally used shorthand of the name, the tuple length and the number of tuples at the last reorganization of this relation.

AABATT. - Each tuple describes an attribute. The following characteristics are taken into account: the attribute name and identifier, the identifier of the relation to which the attribute belongs, a key indicator, the length of the attribute and the startposition within the tuple; the type (alphanumeric/numeric) and the presence of a SORTFL.

AACSRD. - Each tuple describes a SORTFL. It is possible to have a SORTFL without DIRFL. This is useful for small SORTFLs. Besides the relation identifier, the attribute identifier and a directory indicator,

some statistical data are stored in the tuple. These include the multiplicity, which gives the average number of tuples in which a certain attribute value occurs and a small histogram of the adopted values. Based on these data the optimizer (see below) selects the strategy to handle a statement. Only when a relation is reorganised and new SORTFLs are created these data are updated.

4.4. Optimizer

An important part of each database system is the optimizer. Dependent on the statement and the database characteristics this module has to determine the strategy to obtain the result. In SEQUALECT an INSERT is one tuple at a time, while the DELETE and UPDATE statements have a WHERE clause qualifying the touched tuples, see appendix 1. It appears that the handling of a query is the central part of the optimizer as it contains among others a WHERE-clause. First the optimization criterion will be treated, followed by the handling of one-relation queries and equijoins.

The optimization criterion. - How to handle a query when the storage structure is given. The optimizer will concentrate on needed CPU, disc and FASTINT time. When a query has to be handled, the needed CPU time consists of several parts, e.g.: time to analyze the query, to optimize it, to fetch pages and to select records from pages. The time to analyze and optimize a query is independent of the storage structures. When the optimizer chooses to minimize the number of page accesses then the CPU time is taken, to a certain degree, into account as the CPU time to fetch pages is minimized. A DIRFL contains an entry `attribute value, page number'' for each page in a SORTFL. When an entry occupies 20 bytes and a page 4000 bytes then the extra memory to store a DIRFL is negligible. It can be shown easily that in normal situations the directory is two levels deep. The root page is small enough to be kept in internal memory. From now on it is assumed that each SORTFL has a DIRFL, which is two levels deep and has the root page in internal memory. DIFFLs are small and stored in the fast FASTINT memory and the accesses to DIFFL will be neglected. So the optimization criterion is to minimize the number of accesses to disc memory.

One-relation query. - Three subjects have to be treated: how to handle conditions, nesting and GROUP BY/ORDER BY clauses.

a. A condition can be written in conjunctive normal form, which means: as a conjunct of some disjuncts. To contribute to the final result a tuple has to satisfy each disjunct. A disjunct is an OR-chain of boolean primaries and for each boolean primary a selectivity factor (sf) can be computed. This factor is an estimate of the fraction of the tuples which will satisfy the boolean primary. The database catalog contains statistical data about the values which attributes adopt and these data will be used. The multiplicity, the highest and lowest value adopted (not yet implemented) and the histogram will be used together with the cardinality of the relation. The following cases are distinguished.
   - attribute EQ constant: sf= multiplicity / (cardinality of the relation).
   - attribute NE constant: sf= 1-(sf for equality).
   - attribute GT constant: Use the histogram to compute sf. The same holds for LT, LE and GE.
   - attribute BETWEEN value1 AND value2: Again use the histogram to compute sf.
   - attribute IN (list of values): sf= (number of values in list) * (sf for equality).
   - attribute IN subquery: Compute the expected cardinality of result of subquery. Now apply the previous case.
   - in all other cases: sf=1/3. Of course this is only a very rough estimate. An example of this case is the boolean primary 'attribute1 GE attribute2'.

   For each disjunct a selectivity factor sf can be computed using the assumption that values of different attributes are independent: sf(bp1 OR bp2)= sf(bp1) + sf(bp2) - sf(bp1) * sf(bp2). Here bp is a boolean primary. This formula can be generalised. An important class of boolean primaries has the form 'attribute comparator constant'. Such a primary is called an interesting primary. When a SORTFL (with DIRFL) exists for the attribute then the SORTFL can be used to handle the boolean primary, as the ordering criterion is the concerned attribute. When all boolean primaries of a disjunct are interesting then the disjunct is called interesting and the corresponding SORTFLs can be used to fetch the qualifying records. Accesses to the SORTFLs via the DIRFLs are needed and also sorting of the records on a common attribute to be

able to delete duplicates. The number of page accesses can be computed. When more interesting disjuncts exist then the disjunct which needs the least page accesses will be selected and compared with the scanning of an arbitrarily SORTFL. Now the strategy with the least number of page accesses is selected.

Using the assumption about the independence of attribute values a selectivity factor for a condition as a whole can be computed: sf(condition)= product of sf(disjuncts). This will be used in the handling of equijoins.

b. Nesting. Each subquery must involve only one relation. Example: "Give the ORDERNOs of the orders for parts which have a QOH¨7000". In SEQUEL:

```
SELECT      ORDERNO
FROM        ORDERS
WHERE       PARTNO IN
            (SELECT PARTNO
             FROM   PARTS
             WHERE  QOH¨7000).
```

For the innermost subquery the optimal way of obtaining the result is determined and the needed number of accesses computed. Using this result the innermost but one subquery can be treated, etc.

c. GROUP BY/ORDER BY. When until now no SORTFL has been selected to handle the query, then the attribute mentioned in the GROUP BY or ORDER BY clause determines the access path. When such a clause is not present, then an arbitrary access path will be selected.

Equijoin. - Many methods to implement an equijoin exist. Extensive studies have shown that no method is always superior, see Blasgen and Eswaran [6]. The ordering of files is an important aspect of VIDEBAS, so it is reasonable to choose an equijoin method based on sorting.

a. When more than two relations contribute to the final result, determine first the equijoin of two relations which gives the smallest intermediate result. This result is equijoined with that relation which minimizes the next intermediate result and so on. Suppose A, B and C have to be equijoined and we can start with equijoining A and B, but also with B and C. Then the sizes of the two intermediate results are computed using sf(condition) and the sizes of the required attributes.

The equijoin with the smaller result is chosen.

b. Consider an equijoin of A and B on attribute e. External memory is less important than reponse time, so the equijoin method has to be fast when all needed SORTFLs are available, and has to work otherwise. The selected equijoin method is called the 'scan-direct' method as one relation will be scanned in the order of the equijoin attribute e, while the other will be accessed directly. First there has to be decided whether A or B should be scanned. The criterion is the number of times needed to access the other relation directly. The WHERE-clause in the equijoin can be brought in conjunctive normal form. The disjuncts which concern A attributes only are together called cond(A). Compute the number of tuples of A which satisfy cond(A) and determine the number of different e-values edif(A) which are adopted by those tuples. Compute also edif(B) and suppose that edif(A) is smaller than edif(B). Then A will be scanned in order of attribute e and B will be accessed directly. This implies that before applying the equijoin operation it must be made possible to scan A in order of e and to access B directly with an e-value. Often the available SORTFLs and DIRFLs will allow this; sometimes, however, a sort operation is needed beforehand.

Suppose S(A) is the set of A attributes for which a SORTFL exists and I(A) is the set of interesting disjuncts which apply to A attributes only. The following cases are distinguished.

- e in S(A), S(B) and I(A), I(B) not empty.
  Compute the number of page accesses to scan SORTFL(e) of relation A. Moreover, compute for each disjunct in I(A) the number of page accesses to fetch the A tuples using the SORTFLs with corresponding DIRFLs, to sort the intermediate result on e and to scan the sorted final result. The strategy with the least page accesses is chosen. For relation B an analogous reasoning holds: a known number (say n) of direct accesses have to be performed on B. The number of page accesses needed to access B n times directly using DIRFL(e) and SORTFL(e) can be computed. I(B) is not empty; compute for each interesting disjunct the number of page accesses to fetch the tuples of B satisfying cond(B), to sort those tuples on e, to create a directory and to access n times the sorted resulting file. The strategy with the lowest number of accesses will be selected.
- When for A or B holds that either I is empty or e is not in S then only one alternative remains.

- When I is empty and e not in S then a sorted file on e with directory will be created: scan the relation, neglect the tuples not satisfying the condition while taking into account only the required attributes (projection) and sort the result on e. Now we can apply the 'scan-direct' method. Blasgen and Eswaran [6] have shown that in this situation the described method offers a good performance.

c. GROUP BY/ORDER BY clause. The result until now is ordered on the last equijoin criterion. So, when this clause is present in the equijoin statement sorting is necessary unless the criterion is this last equijoin criterion.

4.5. The status of the simulation

DIFMAN and SYSADM have been implemented now. To realize SQLMAN a parser generator has been used. SQLMAN is ready, except the procedure to handle DIRFLs and the optimizer. The size of DIFMAN is about 1800 PASCAL lines, SYSADM is 2000 lines, while SQLMAN is the largest: 4000 lines. Those three parts have, however, large parts in common, which makes the total size of VIDEBAS about 6000 lines of PASCAL code. In total about three man years have been spent. At present we are implementing VIDEBAS more completely and efficiently.

5. SELECTION OF STORAGE STRUCTURES

One of the tasks of the database administrator is to select the storage structures of a database. The optimal storage structure depends on database characteristics: number of tuples per relation, length of the tuples etc. Another important factor is the load: a list of SEQUALECT statements with their frequencies. In general, finding an optimal or nearly optimal storage structure given the load and database characteristics is difficult. One is, however not really interested in finding the optimal solution since the input to the problem is almost always inaccurate. The description of the load is often an approximation of reality as the frequencies are mostly only rough estimates. The same holds for the description of the database contents. Finally the model to

estimate the number of page accesses needed to handle a statement is only exact under certain assumptions: independence of attribute values, assumed distributions of attribute values, etc. In general we must be content to find a storage structure for the database which is reasonably good.

In VIDEBAS the storage structure is determined by the set of SORTFLs, DIRFLs and DIFFLs. As the considered environment for VIDEBAS is the on-line maintenance and retrieval environment immediate updates are to be expected for all relations, implying a DIFFL for each relation. In the DEC-system10 simulation DIRFLs (each SORTFL has one) are allocated to disc memory. In a real VIDEBAS implementation, however, an allocation to FASTINT is also possible. To get an impression of the difference in performance it is required that the tool allows the allocation of the DIRFLs to disc memory or to FASTINT.

How to implement this tool? A load description gives the frequencies with which SEQUALECT statements have to be executed. So the syntax of SEQUALECT has to be changed only slightly to define a load language, causing a minor modification in the SQLMAN analyser. The VIDEBAS optimizer can be used to predict the number of page accesses for the handling of a statement for a given storage structure. A study of the algorithms used by the optimizer learns that the load caused by a statement can be split up over the relations which contribute to the result. Two observations make this statement reasonable: when more than two relations have to be equijoined then the sequence of equijoining will be determined independent of the storage structures. The same holds for the determination of the relation which has to be scanned, if an equijoin of two relations has to be performed. For each relation the total load can be determined once and for all, meaning that the storage optimization can be done on a relation at a time basis. This observation makes life easy. Suppose that for a certain relation the storage structure which minimizes the number of page accesses has been selected. Now the database administrator can decide to delete a SORTFL and then the loss in speed can be computed. It is up to the database administrator to make the deletion permanent or not. He has to made the comparison external memory occupation versus page accesses. After making these decisions for all relations the "optimal" storage structure remains.

6. FUTURE RESEARCH

The future research in our group will be focussed on two areas: the realization of an efficient VIDEBAS simulation on the DEC-system10 (as far as possible) and an extensive comparison of the performance of a VIDEBAS database system with SYSTEM-R.

6.1. An efficient DEC-system10 implementation

Such an implementation includes at least the following subjects. The optimizer has to be written and perhaps improved. Next the DIFFLs which are now allocated to normal DEC-files will be placed in virtual memory. This means that FASTINT is realized using RAM and magnetic discs, media which will decrease in price drastically in the near future. The maximum size of the DIFFLs will depend on the maximum address space which is not so high on the DEC-system10 (about 1.25 Mb). This will improve performance as the PASCAL character input/output to DEC files is, of course, slow. The third point is, to use special instructions of the DEC-system10 processor. Attribute values are character strings and to sort or select tuples, operations on character strings are needed. In the current VIDEBAS simulation this is performed in PASCAL by comparing attribute values character by character. The DEC-system10, however, offers string operations like 'compare string', 'move string', 'convert string' and so on. These instructions are much faster.

6.2. Comparison with SYSTEM-R

To perform this comparison first the tool described in the previous chapter has to be implemented. Given a load and logical database description the tool will determine how many page accesses are needed to handle the load, given a (nearly) optimal storage structure. When the same data are available for SYSTEM-R databases then a comparison can be made.

7. CONCLUSION

The architecture of VIDEBAS, a relational database management system has been outlined. Arguments for a DEC-system10 simulation of VIDEBAS have been given, together with the reasons behind the decisions concerning this simulation. Planned parts have been described. Future research amounts to an extensive performance analysis and a faster and more complete realization of VIDEBAS.

Acknowledgement

The students S. Wiering and B. van Hoof are acknowledged for defining SEQUALECT and SYSLAN, and for implementing large parts of the VIDEBAS simulation on the DEC-system10.

8. REFERENCES

[1] CODD, E.F., A Relational Model for Large Shared Data Banks, comm, of the ACM, vol. 13, no. 6, pp 377-387, 1970.

[2] CHAMBERLIN, D.D. et al., SEQUEL 2: A Unified Approach to Data Definition, Manipulation and Control, IBM Journal of Research and Development, 20, 6, nov 76, pp 560-575.

[3] SEVERANCE, D.G. and G.M. LOHMAN, Differential Files: Their application to the Maintenance of Large databases ACM Transactions on Database Systems, vol 1, no 3, sept 76, pp 256-267.

[4] BLANKEN, H.M., The architecture of VIDEBAS, a Relational Database management system, Int. Conference on systems Architecture, march/april 1981, London.

[5] VERHOFSTAD, J., Recovery based on Types, IFIP Working Conference on Database Architecture, Venice, june 1979, pp 125-140.

[6] BLASGEN, M.W. and K.P. ESWARRAN, Storage and Access in Relational Data Bases, IBM Systems Journal, 1977, pp 363-377.

In the following appendices three interfaces are described using a notation called BTF. This notation has been chosen as the parser generator requires a LL/1 syntax in BTF as input.
The symbols "(", ")", ".", ":", ";", and "," and the words option, chain, list, pack, and sequence are part of the metalanguage. A ";" separates a syntactical unit from its definition. The symbols "(" and ")" enclose a syntactical unit in a definition. The ";" indicates an alternative, and "," means 'followed by'. A "." finishes a definition. Option indicates that the preceding syntactical unit may be present or not. A syntactical unit followed by sequence means that this unit may be repeated one or more times. Suppose that A and B are syntactical units.
A pack : left parentheses symbol, A,
right parentheses symbol,
A list : A, (comma symbol, A) sequence option.
A chain : A, (B, A) sequence option.
When the name of a syntactical unit finishes with "symbol", then the unit is a terminal. Their representations are for briefness' sake not given here. Also the syntactical units "letter", "integer" and "character" are not given.

Appendix 1: SEQUALECT

Sometimes the syntax has an artificial nature. This may be caused by the required LL/1 property. The names of some terminals are written deliberately in capitals to correspond to the SEQUEL keywords, see Chamberlin [2]. Explanation is sometimes given in the form of comment, which starts with "(*" and finishes with "*)".

```
SEQUALECT statement
        : (retrieval;  dml), semicolon symbol.
retrieval
        : query;  assignment;  deassignment.
query   : queryblock, (ORDER BY symbol, selint, DESC symbol option) option,
          (*
            Selint is a positive integer and indicates the  selint'th  item
            in the selexpr in the queryblock.  Ascending order is default.
          *)
queryblock
        : SELECT symbol, UNIQUE symbol option,
          (selexpr list; asterisk symbol), FROM symbol,    relation
          name, querytype.
          (*
            The asterisk is used as a shorthand for all the  attributes  of
            the relation.  An equijoin of more than two relations has to be
            defined with help of an ASSIGN statement!
          *)
selexpr : attribexpr ;  fun, attribexpr pack;
          COUNT symbol, ((UNIQUE symbol, attrib name);
          asterisk symbol) .pack,
fun     : AVG symbol;  SUM symbol;  MIN symbol;  MAX symbol.
          (*
            AVG and SUM demand numerical arguments;  MIN and MAX  may  have
            alphanumeric  arguments.   The  type  of the result will be the
            same as the type of the operand.
          *)
querytype
        : comma symbol, relation name, WHERE symbol, attribspec, eq symbol,
          attribspec,
```

```
          (AND symbol, simplebool) option;
          (WHERE symbol, boolean) option,        (GROUP BY symbol, attrib
          name) option.
          (*
            Equijoins and one-relational queries are possible. In the
            latter case the allowed condition is more general than in the
            former. Equijoins between more than two relations have to be
            defined using the ASSIGN statement, see furtheron. So the
            relation name may be the name of a temporary relation defined
            by an ASSIGN.
          *)
attribexpr
        : ((attribexpr pack; attribspec; integer)     chain multop symbol)
          chain addop symbol.
          (*
            When the multop or addop is used, the attribspec may only refer
            to numeric attributes.
          *)
boolean : ((NOT symbol option,          predicate) chain AND symbol) chain
          OR symbol.
predicate
        : comparand, comparator, comparand; lsquarebr symbol, boolean,
          rsquarebr symbol.
          (*
            The left- and rightcomparand must both be numeric or
            alphanumeric.
          *)
comparator
        : alphnumcomp symbol; setcomp.
          (*
            When a alphanumeric or numerical comparator (alphnumcomp) is
            used in a predicate the left- and rightcomparand must have
            exactly one element (obtained by a simple expr or a select,
            selecting just one element).
          *)
alphnumcomp
        : ne symbol; eq symbol; lt symbol; le symbol; gt symbol; ge
          symbol.
```

```
setcomp : NOT symbol option,
          (IN symbol; setequal symbol, TO symbol option).
          (*
            IN indicates setmembership and setinclusion. In the latter
            case it considers left- and rightcomparand as set of zero
            (select with "empty" result) or more tuples. (note:
            empty-set IN any-set is always TRUE).
          *)
comparand
        : expr;  SELECT symbol, selexpr, FROM symbol, relation name,
          (WHERE symbol, predicate) option;              lbrace symbol,
          constant list, rbrace symbol.
          (*
            When the second possibility is chosen we have nesting. It is
            possible that in the predicate (after WHERE) a reference is
            made to relation names from an outer SELECT-block.
          *)
expr    : attribexpr; quotedstr.
constant: integer; quotedstr.
simplebool
        : ((expr, lexnumcomp symbol, expr;  lsquarebr symbol, simplebool,
          rsquarebr symbol)
          chain AND symbol) chain OR symbol.
assignment
        : ASSIGN symbol, query, TO symbol, destination.
destination
        : (file symbol, filename);   (rel symbol, relation name, (attrib
          name list) pack.
          (*
            The destination can be a DEC-system10 file or a temporary
            relation. This relation may be used in the SEQUALECT
            statements of this session. The names of the attributes of the
            temporary relation are given in the list. No materialization
            is necessary until output is required!
          *)
deassignment
        : DEASSIGN symbol, relation name.
          (*
```

```
            From now on the previously defined temporary relation is not
            defined anymore.
          *)
dml     : insertion; deletion; update; do; undo.
          (*
            Only permanent relations (not those created by ASSIGN) may be
            changed.
          *)
insertion
        : INSERT INTO symbol, relation name, colon symbol langlebr symbol,
          constant list, ranglebr symbol.
deletion: DELETE FROM symbol, relation name,
          (WHERE symbol,simplebool) option.
          (*
            Without WHERE clause the relation will be emptied.
          *)
update  : UPDATE symbol, relation name, SET symbol (attrib name, eq symbol,
          expr) list (WHERE symbol, simple booleaN) option.
attribspec
        : relationorattrib name, (period symbol, attrib name) option,
          (*
            The construct 'relationorattrib' is needed to make the language
            LL/1.
          *)
quotedstr
        : quote symbol, letter sequence, quote symbol.
do      : do symbol.
undo    : undo symbol.
... name: letter sequence
```

Appendix 2: SYSLAN

Successively the following statements are treated. CREate RELation, DELete RELation, ADD SORtfile, DELete SORtfile, ADD DIRectory, DELete DIRrectory and REORGanize. The database administrator has to know the conventions used in VIDEBAS to construct the file names. From these names VIDEBAS can derive the relation, the file kind (SORTFL, DIRFL or DIFFL)

and, in case of SORTFL the criterion on which the file is ordered.

```
SYSLAN statement
        : (crerel; delrel; addsor; delsor; adddir; deldir; reorg;
          endses), semicolon symbol.
crerel  : crerel symbol, relation name, comma symbol, attribute decs list.
attribute desc
        : attribute name, comma symbol, attribute length, attribute type,
          key indicator.
attribute length
        : length symbol, integer.
attribute type
        : type symbol, (alpha symbol; numer symbol).
key indicator
        : key symbol, integer.
delrel  : delrel symbol, relation name.
addsor  : addsor symbol, sortfile name.
delsor  : delsor symbol, sortfile name.
adddir  : adddir symbol, dirfile name.
deldir  : deldir symbol, dirfile name.
reorg   : reorg symbol, reorg objekt.
reorg objekt
        : relation name; diffile name.
          (*
            With the parameter 'relation name' the whole relation will be
            reorganized, which means that new SORTFLs will be created, and
            that the DIFFL will be emptied. If the parameter is a 'diffile
            name' then only a new DIFFL will be created. This means that
            for each updated tuple only the newest version will be kept (in
            VIDEBAS more than one version of a tuple may coexist to offer
            the reader processes a single-user environment, see BLANKEN
            [4]).
          *)
endses  : endses symbol.
```

Appendix 3: DIFMAN interface

In essence the following syntax describes the interface which is used in VIDEBAS. There are however minor differences.
In SQLMAN and SYSADM the conditions are brought in conjunctive normal form and offered to DIFMAN in reversed polish form. With rid is meant the internally used relation identifier, while aid stands for attribute identifier. The identification of a program which issues the instruction is delivered by the communication routines to DIFMAN. Together these routines are called the Inter Process Communication Facilty. So DIFMAN knows whom to respond the return code and possibly (in case of 'read tup') the ordered (sub)tuples.

```
instruction
        : (enter db; read tup; delete tup;
          insert tup; do upd; undo upd; delete dif;
          create dif; reorg dif; empty dif; leave db).
enter db: enter db symbol, (read symbol; write symbol).
read tup: read symbol, rid, condition option, output,
          orderby symbol, aid,
          (ascending symbol; descending symbol).
          (*
            The orderby clause is always needed as
            a merge has to be performed by the calling
            routine.
          *)
condition
        : boolean; boolprim.
boolean : condition, condition, bool operator.
bool operator
        : and symbol; or symbol.
boolprim: term. term, reloprtor.
term    : aritm expr; operand.
aritm expr
        : term, term, aroprtor.
operand : literal; aid.
reloprtor
        : ne symbol; eq symbol;
```

```
          lt symbol; le symbol; gt symbol; ge symbol.
aroprtor: mult symbol; div symbol; add symbol; sub symbol.
output  : aid sequence; asterisk symbol.
delete tup
        : delete symbol, rid, tuple.
insert tup
        : insert symbol, rid, tuple.
tuple   : literal sequence,
literal : (integer; character sequence) sequence.
do upd  : do symbol.
undo upd: undo symbol.
.... dif: .... dif symbol, rid.
leave db: leave db symbol.
```

DATA-ALLOCATIE IN EEN GESPREIDE DATABASE

P.M.G. APERS
Vrije Universiteit, Amsterdam

1. INLEIDING

Het concurrency control mechanisme, de query verwerkingsalgoritme en de data allocatie zullen gezamenlijk de efficiëntie van een gespreide database bepalen. Deze efficiëntie zal in hoge mate de acceptatie van gespreide databases beinvloeden. Wat precies geoptimaliseerd moet worden om deze efficiëntie te bereiken, is afhankelijk van het type netwerk. In ARPA-achtige netwerken vormt de communicatie de bottleneck en zal het minimaliseren van het totale netwerkverkeer een gunstige invloed hebben op de verwerking van de queries. In lokale netwerken, bestaande uit een snelle verbinding en eén aantal microprocessoren, is het wellicht beter om de responstijd te minimaliseren.

In dit artikel zullen we gereedschappen aandragen om de data allocatie zo efficient mogelijk te maken. Wie deze gereedschappen zal gebruiken hangt af van het soort database. Voor een min of meer centraal beheerde database zal de database administrator er mee onderzoeken hoe efficient de huidige allocatie is en er adviezen van krijgen om zo nodig wijzigingen aan te brengen. In databases die gekenmerkt worden door een decentraal beheer, zal het database management systeem (dbms) zelf beslissingen moeten kunnen nemen over de distributie van de data en ook over de wijze van opslaan. Belangrijk in dit soort databases is dat de gebruiker en/of eigenaar van de data niet zelf de distributie of wijze van opslaan kan veranderen, omdat hij totaal geen overzicht heeft over het gebruik van zijn data in relatie tot andermans data. Ten grondslag aan het gebruik van de allocatie gereedschappen ligt het verzamelen van statistische gegevens over de queries die gesteld worden.

De indeling van dit artikel is als volgt. In hoofdstuk 2 wordt het data en operatie allocatie probleem uit de doeken gedaan. In hoofdstuk 3 bekijken we de verwerkingsstrategieën geproduceerd door de query verwerkingsalgoritmen en hoe deze gerepresenteerd kunnen worden. In de hoofdstukken 4 en 5 worden twee algoritmen gepresenteerd, één om het totale netwerkverkeer te minimaliseren en één om de gemiddelde responstijd te minimaliseren. Tenslotte, zal in hoofdstuk 6 ingegaan worden op de algemene problematiek van data allocatie in gespreide databases.

2. DATA EN OPERATIE ALLOCATIE PROBLEEM

Een database wordt voorgesteld door 3 niveaus: extern schema, conceptueel schema en intern schema. In een gecentraliseerde database zal alle informatie betreffende opslagstructuren, secondary indices, etc. alleen voorkomen in het interne schema. De distributie van de data in een gespreide database is te vergelijken met de opslagstructuren in een gecentraliseerde database en daarom zal informatie daarover slechts voorkomen in het interne schema. De gebruiker van een gespreide database behoeft niets van de distributie te weten. Voor hem mag het, wat betreft zijn vraagstelling, geen verschil maken of hij op een gecentraliseerde of gespreide database werkt. De wijze waarop de data gedistribueerd en/of opgeslagen is, behoeft alleen maar bekend te zijn aan het query verwerkingsalgoritme, omdat deze de verwerkingsstrategieën berekent.

Verschillende aspecten van de data allocatie zijn: efficiëntie, beschikbaarheid en security. Onder beschikbaarheid van de data wordt verstaan de kans dat gebruikers kunnen doorwerken, indien verbindingen in het netwerk zijn verbroken of computers niet beschikbaar zijn. Door meer redundantie toe te staan, zal een hogere beschikbaarheid verkregen worden, maar dit moet wel afgewogen worden tegen het verhoogde netwerkverkeer om de verschillende copieën van de data consistent te houden. Het security aspect zal meestal tot uitdrukking komen in beperkingen van de mogelijke allocaties, zoals bijv.: relatie R mag alleen maar op computer C voorkomen en alle operaties op R moeten ook door C verwerkt worden. De efficiëntie van een allocatie is een nogal veel omvattend begrip. We zullen het hier definiëren als het zo optimaal mogelijk gebruik maken van de middelen, computers en communicatie verbindingen, met het doel een zo groot mogelijke throughput te verkrijgen, wat betreft queries en updates.

In dit artikel zullen we alleen de efficiëntie van de allocatie beschouwen en deze zal gemeten worden door een kostenfunctie. Het data en operatie allocatie probleem is, gegeven de queries en updates en de frequenties van gebruik, het vinden van een allocatie van de fragmenten van relaties en de er op uit te voeren operaties, z.d.d. de kostenfunctie wordt geminimaliseerd. De twee kostenfuncties die worden bekeken zijn: het totale netwerkverkeer en de gemiddelde responstijd.

Het data en operatie allocatie probleem is een generalisatie van het file allocatie probleem, wat betreft de verwerkingsstrategieën van de queries en updates en het beschouwen van diverse kostenfuncties. Het file allocatie probleem is het vinden van een file allocatie z.d.d. het totale netwerkverkeer wordt geminimaliseerd. De toegestane verwerkingsstrategieën zijn erg simpel. Indien men data van verschillende files wil combineren, moet iedere file apart benaderd worden en moet zijn data naar de computer van de gebruiker gestuurd worden, alvorens overgegaan kan worden tot het produceren van het resultaat. De wijze waarop de data wordt verwerkt in een gespreide database is echter veel gecompliceerder. In het hoofdstuk over query verwerking komen we hierop terug.

Ook behoeft er in het file allocatie probleem geen rekening gehouden te worden met de allocatie van de operaties (zie ook CHU, HOLLOWAY, MINTSUNG LAN en EFE[12]) op de data omdat deze volledig bepaald wordt door de data allocatie. In een gespreide database zijn er echter operaties die werken op tussenresultaten en daarom niet plaats gebonden zijn. Een voorbeeld van zo'n operatie is een join die uitgevoerd wordt op de twee operanden, die verkregen zijn d.m.v. een selectie. In het hoofdstuk over het minimaliseren van de gemiddelde responstijd wordt het probleem van operatie allocatie besproken.

In CHU[9,10] werd waarschijnlijk voor het eerst het file allocatie probleem beschreven. Voor een iets ander model werden in CASEY[8] en GRAPA en BELFORD[18] stellingen gegeven om de zoekboom te beperken. In ESWARAN[17] is aangetoond dat het probleem polynomiaal volledig is. Het probleem van file allocatie en het bepalen van de capaciteiten van de communicatielijnen werd onderzocht in MAHMOUD en RIORDON[27]. Levin behandelt in LEVIN[25] en LEVIN en MORGAN[26] het alloceren van files en programma's in een netwerk. Een vertaling van het file allocatie probleem naar een gespreide database kan men vinden in RAMAMOORTHLY en WAH[31]. Echter de

toegestane verwerkingsstrategieën verschillen niet van die in het file allocatie probleem.

3. QUERY VERWERKING IN EEN GESPREIDE DATA BASE

3.1. Query verwerkingsalgoritmen

Query verwerking in een gespreide database wordt gekenmerkt door data transmissies tussen computers in het netwerk. In de meeste literatuur wordt verondersteld dat de kosten van deze transmissies een orde van grootte duurder zijn dan het lokaal verwerken van data; dit laatste is inclusief het lezen van en schrijven op disk. Dus de totale verwerkingskosten van een query worden volledig bepaald door de transmissiekosten. Een consequentie van deze aanname is dat er geen rekening gehouden behoeft te worden met het verwerken van subqueries op één computer. Een andere aanname in de literatuur is dat de kosten van data transmissies tussen elk tweetal computers even duur is.

Query verwerking in gespreide databases krijgt veel aandacht in de huidige literatuur. In WONG[34] wordt een algoritme voorgesteld, dat op het moment geimplementeerd wordt in het SDD-1 systeem (ROTHNIE en GOODMAN[32]). In EPSTEIN, STONEBRAKER en WONG [14] wordt een algoritme gepresenteerd, dat deel uit gaat maken van een gespreid INGRES data base management systeem (STONEBRAKER en NEUHOLD[33]). Verbeteringen hierop zijn te vinden in EPSTEIN en STONEBRAKER[15] en EPSTEIN [16]. Onderzoek dat gebaseerd is op het berekenen van semi-joins: HEVNER en YAO [20], HEVNER[21], APERS[1], APERS, HEVNER en YAO[4]. Algoritmen die dynamisch hun strategieën bepalen, worden onderzocht in NGUYEN GIA TOAN[28,29]. Resultaten van ander onderzoek op dit gebied kan men vinden in: BALDISSERA, BRACCHI en CERI[7], IN-SUP PAIK en DELOBEL[23], PELAGATTI en SCHREIBER[30] en CHU en HURLEY[11].

Nadat een query van een gebruiker aan het systeem is aangeboden, wordt er eerst gecontroleerd of de relaties en attributen in de query wel bestaan, op de juiste wijze gebruikt worden en of de gebruiker deze query wel mag stellen. Daarna wordt door de query verwerkingsalgoritme m.b.v. de data dictionary, die gegevens bevat zoals locatie, grootte, etc., een strategie bepaald. Deze strategie omvat de taken van en de data transmissies tussen de verschillende computers. Een voorbeeld van zo'n taak is: voer

een selectie en projectie uit op relatie R en stuur het resultaat naar computer C. Wat de achterliggende ideeën zijn van een algoritme en hoe deze de verwerkingsstrategieën berekent, is wat betreft de data en operatie allocatie niet interessant. Het gaat er alleen maar om dat er verwerkingsstrategieën kunnen worden berekend.

Voorbeeld

Gegeven een database met cursussen van een internationale summerschool, bestaande uit de volgende relaties:

```
prof(pnaam,land,... )
student(snaam,land,... )
inschrijving(snaam,cursus)
```

die elk op een andere computer voorkomen. De query, gesteld door een gebruiker op een andere computer dan de relaties, zoals bijv.

```
SELECT pnaam, snaam
FROM prof, student
WHERE prof.land = student.land
```

kan als volgt verwerkt worden:

- projecteer relatie !prof' op de attributen pnaam en land,
- stuur het resultaat naar de computer waar 'student' voorkomt,
- projecteer relatie 'student' op de attributen snaam en land,
- bereken de join met als conditie prof.land = student.land,
- stuur het resultaat naar de computer van de gebruiker.

Fig. 1 laat de data transmissies zien om de query te verwerken.

3.2. Verwerkingsstrategieën graaf

Bij de allocatie van de fragmenten van relaties in een netwerk hebben we niet met één query te maken, maar met een heleboel. Het probleem dat we hier hebben is, dat we enerzijds niet in staat zijn om de verwerkingsstrategieën te berekenen omdat de data allocatie nog niet bekend is en dat we anderzijds niet in staat zijn de fragmenten optimaal te alloceren omdat de verwerkingsstrategieën nog niet bekend zijn. Om dit op te lossen zullen we een hypothetische allocatie veronderstellen. Elk fragment wordt aan een

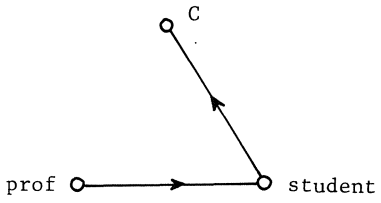


Fig. 1: Data transmissies.

eigen virtuele computer geassigneerd. Deze virtuele computers hebben niets te maken met de fysieke computers in het netwerk. Nu wordt voor elke query de verwerkingsstrategie berekend; ook de operaties waarvan de verwerking niet gebonden is aan de virtuele computer van een fragment, zullen geassigneerd worden aan een eigen virtuele computer. Deze strategieën en de boodschappen die nodig zijn om de benodigde fragmenten te locken zullen worden gerepresenteerd door een verwerkingsstrategieën graaf. Zo'n graaf bestaat uit:

1) C-punten, voor computers,

2) VS-punt voor fragmenten en operaties.

Deze punten zijn verbonden door gerichte lijnen, die gelabeld zijn met de hoeveelheid data die van het ene aangrenzende punt naar het andere wordt getransporteerd per query of per eenheid van tijd. Fig. 2 geeft een voorbeeld van zo'n graaf zonder de concurrency control boodschappen.

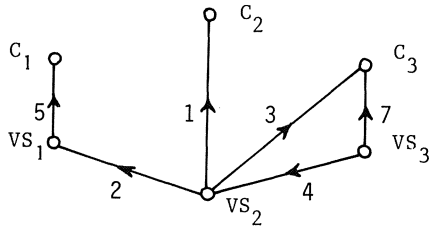


Fig. 2: Verwerkingsstrategieën graaf

M.b.v. een verwerkingsstrategieën graaf kunnen niet-bestaande allocaties worden voorgesteld. Het uiteindelijke doel van het data en operatie allocatie probleem is alle virtuele computers te identificeren met fysieke computers in het netwerk.

4. MINIMALISEREN VAN HET TOTALE NETWERKVERKEER

In dit hoofdstuk zullen we conform de literatuur van query verwerking, veronderstellen dat de transmissie van data vele malen duurder is dan het lokaal verwerken ervan. Om de throughput van queries en updates zo groot mogelijk te maken, zullen we het totale netwerkverkeer minimaliseren. De door de meeste query verwerkingsalgoritmen geproduceerde strategieën omvatten alleen maar data transmissies. De lokale operaties worden impliciet verondersteld, en tellen niet mee in de kosten van een strategie. In de verwerkingsstrategieën graaf zullen deze operaties aan dezelfde virtuele computer geassigneerd worden als de fragmenten waarop ze werken.

4.1. Minimaliseren van het aantal data transmissies

Alvorens we het minimaliseren van het totale netwerkverkeer behandelen, zullen we eerst een eenvoudiger model bekijken. Veronderstel dat de kosten van het transporteren van data onafhankelijk is van de hoeveelheid data. D.w.z. bij het bepalen van de kosten telt alleen maar het aantal data transmissies. De lijnen in de verwerkingsstrategieën graaf zullen dan gelabeld worden met het aantal data transmissies per eenheid van tijd. De som van het aantal data transmissies moet nu geminimaliseerd worden.

Bij de introductie van de verwerkingsstrategieën graaf werd niets gezegd over de veranderingen in de strategieën als de nog niet-bestaande allocatie wijzigde. Als we naar de query verwerkingsalgoritmen kijken, blijkt dat een verwerkingsstrategie soms drastische wijzigingen ondergaat. Bijv. de verwerking van een join wijzigt, indien operanden (fragmenten) aan een zelfde computer worden geassigneerd. In het eenvoudige model van deze sectie is het mogelijk de uiteindelijke verwerking van joins in het midden te laten, door deze in de verwerkingsstrategieën graaf te representeren door een joingraaf. De joingraaf in fig. 3 laat de volgende verwerkingsstrategieën toe:

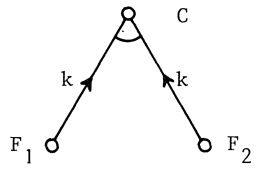


Fig. 3: Joingraaf.

- F_1 en F_2 parallel naar C,
- F_1 naar F_2 en het resultaat van de join naar C,
- F_2 naar F_1 en het resultaat van de join naar C.

Indien F_1 en F_2 in een allocatie aan dezelfde computer geassigneerd zijn, dan zal alleen het resultaat van de join van F_1 en F_2 naar C wordt gestuurd, tenzij dit resultaat groter is dan de som van F_1 en F_2. Ook indien F_1 al in de resultaat computer geassigneerd is, zal alleen F_2 naar C worden gestuurd. De verwerking van de join wordt dus bepaald door de allocatie. Essentiëel bij een joingraaf is dat de labels van de lijnen gelijk zijn.

De constructie van de verwerkingsstrategieën graaf is als volgt. Veronderstel dat elk van de fragmenten van de database aan een eigen virtuele computer zijn geassigneerd. Nu worden de queries aangeboden aan de query verwerkingsalgoritme om de strategieën te bepalen. De lijnen tussen de virtuele computers geven de strategieën weer. Indien de uiteindelijke verwerkingsstrategie van een join in het midden gelaten wordt, wordt gebruik gemaakt van de joingraaf.

Om deze niet-bestaande allocatie om te zetten in een echte allocatie zal de algoritme fragmenten van verschillende virtuele computers samennemen op één virtuele computer en elke virtuele computer identificeren met een fysieke computer. Door dit soort veranderingen wijzigt de verwerkingsstrategieën graaf. Als fragmenten van twee virtuele computers worden samengenomen, verdwijnen alle lijnen tussen die twee computers in de verwerkingsstrategieën graaf. En als door samenvoeging van twee virtuele

computers twee parallelle lijnen ontstaan in een joingraaf, wordt er één verwijderd.

Alvorens we de algoritme geven nog een paar definities. $LINK_{ij}$ is het aantal data transmissies dat verdwijnt als VS_i en VS_j worden samengenomen. DT_{iC} is het aantal data transmissies dat verdwijnt als VS_i geidentificeerd wordt met computer C.

Onderstaand algoritme verandert de ingevoerde verwerkingsstrategieën graaf, die een niet-bestaande allocatie representeert, door lokale optimalisatie. Uiteindelijk zal elke virtuele computer geidentificeerd zijn met een fysieke.

identificeer de virtuele computers VS_i met een computer waarvoor geldt dat DT_{iC} is maximaal;
maak een lijst van paren (VS_i, VS_j) en bereken hun $LINK_{ij}$;
while deze lijst is nog niet leeg
do
    neem het paar (VS_i, VS_j) met de grootste $LINK_{ij}$ en verwijder het van de lijst;
    if $\max_C DT_{iC} + \max_C DT_{jC} - (LINK_{ij} + \max_C DT_{(i,j)C}) \leq 0$
        then
            neem VS_i en VS_j samen in de verwerkingsstrategieën graaf en noem het nieuwe punt $VS_{(i,j)}$
    fi
od

De conditie op grond waarvan VS_i en VS_j worden samengenomen, geeft de verandering aan in het totale netwerkverkeer. Om VS_i en VS_j samen te nemen moeten ze eerst worden gesepareerd van de computer waarmee ze nu geidentificeerd zijn. Dit verhoogt het netwerkverkeer met

$$\max_C DT_{iC} + \max_C DT_{jC}.$$

Samenvoeging van VS_i en VS_j en opnieuw identificeren verlaagt het netwerkverkeer met:

$$LINK_{ij} + \max_C DT_{(i,j)C}.$$

Indien het verschil van deze twee expressies niet positief is, zal het netwerkverkeer niet toenemen en is samenvoeging van VS_i en VS_j niet nadelig.

De vraag die we ons nu kunnen stellen is, hoe goed zijn de allocaties verkregen door bovenstaand algoritme. Voordat we een stelling betreffende de optimaliteit zullen geven voeren we een paar definities in.

Twee virtuele computers VS_i en VS_j zijn direkt verbonden als er een lijn is tussen VS_i en VS_j of als VS_i en VS_j in dezelfde joingraaf voorkomen. VS_i en VS_j komen in dezelfde cluster voor als er een pad is van VS_i naar VS_j, zeg $VS_i=VS_{k1}$, VS_{k2}, ..., $VS_{km}=VS_j$, z.d.d. elk opeenvolgend tweetal direkt verbonden is. Een cluster heet eenvoudig als voor elk tweetal relaties VS_i en VS_j, die direkt verbonden zijn, geldt dat de verwijdering van de lijnen tussen VS_i en VS_j en het verwijderen van de lijnen van de joingrafen waarin VS_i en VS_j samen voorkomen, leidt tot een splitsing van de cluster. We definiëren een eenvoudige verwerkingsstrategieën graaf als een verwerkingsstrategieën graaf met eenvoudige clusters en elke computer heeft maar één query per cluster.

Stelling De data allocatie algoritme bepaalt een allocatie met minimaal totaal netwerkverkeer voor eenvoudige verwerkingsstrategie grafen.

Bewijs zie APERS[2].

De hierboven gestelde eisen aan de verwerkingsstrategieën graaf om de optimaliteit van de algoritme te bewijzen zijn nogal stringent. Daarom is het interessant om te zien hoe goed de data allocaties zijn, verkregen uit random gegenereerde grafen. Het berekenen van een optimale data allocatie is nogal tijdrovend en daarom zijn slechts resultaten bekend voor kleine grafen. In APERS[2] laten we zien dat de data allocatie algoritme in alle testruns, met wisselende parameters betreffende de query opbouw, altijd een allocatie vond met minimaal aantal data transmissies.

4.2. Minimaliseren van het totale netwerkverkeer met redundante allocatie

Een realistischer aanname betreffende de kosten van data transmissies dan in de vorige sectie, is te veronderstellen dat deze kosten evenredig zijn met het aantal te versturen bytes. Een gevolg van deze generalisatie is dat de joingraaf niet meer is toegestaan, omdat in het algemeen de labels van de lijnen verschillend zijn. Om toch een zekere flexibiliteit in de verwerkingsstrategieën te behouden worden in sectie 4.4 enige variaties op de data allocatie algoritme voorgesteld.

In APERS[3] is aangetoond dat het bepalen van een allocatie met minimaal totaal netwerkverkeer NP-volledig is. Dit wil zeggen dat er geen algoritme bestaat dat in polynomiale tijd de optimale allocatie berekent. Oftewel, voor grote netwerken en grote aantallen relaties of fragmenten zal het tijdrovend zijn om zo'n allocatie te vinden. Het probleem is echter wel te vertalen naar een bekend niet-lineair geheeltallig programmeer probleem: quadratic assignment probleem (ELAM[13]). Voor de oplossing hiervan kunnen standaard lineaire en niet-lineaire programmeer technieken worden gebruikt. Toepassingen van cluster analyse technieken op het gebied van fysieke opslag (HOFFER en SEVERANCE[22]) hebben echter laten zien dat ook deze tijdrovend zijn. Verder blijven we ook met het probleem zitten dat de strategieën a priori bepaald moeten worden en er totaal geen veranderingen mogelijk zijn. We verwachten daarom dat het beter is heuristische algoritmen te gebruiken, die tussentijdse veranderingen van verwerkingsstrategieën toestaan.

In de vorige sectie werd slechts een niet-redundante allocatie beschouwd. Dat was ook de reden waarom updates niet ter sprake kwamen. Het toestaan van meerdere copieën van een fragment kan enerzijds het totale netwerkverkeer voor individuele queries verminderen, maar anderzijds het totale netwerkverkeer verhogen om de verschillende copieën consistent te houden. Om deze veranderingen in het netwerkverkeer tegen elkaar af te wegen, willen we een representatie hebben z.d.d. een data allocatie algoritme niet alleen de allocatie van de fragmenten bepaalt, maar ook het aantal copieën.

Een update transactie bestaat uit een query gedeelte dat bepaalt welke tuples kandidaat zijn om veranderd te worden. En pas daarna worden ze echt veranderd. We geven hier een representatie van het primary copy principe. Dit principe houdt in dat een query elke willekeurige copie van het

fragment kan gebruiken, maar dat alle updates slechts een uitverkoren copie, de primary copy, kunnen gebruiken. Elke verandering van de primary copy zal bekend worden gemaakt aan alle andere copieën om consistentie te waarborgen. Fig. 4 laat de data transmissies zien die nodig zijn om de primary copy te veranderen en de overige copieën consistent te houden.

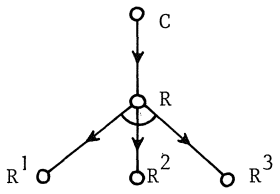


Fig. 4: Een updategraaf

Deze updategraaf is qua vorm gelijk aan de joingraaf van de vorige sectie, omdat veranderingen in de data allocatie dezelfde effecten hebben op de updategraaf.

De constructie van de verwerkingsstrategieën graaf is nu als volgt. Geef elke query zijn eigen copieën van de benodigde fragmenten. Assigneer elk van deze copieën aan zijn eigen virtuele computer. Bepaal voor elke query apart m.b.v. de query verwerkingsalgoritme hoe de strategie zal zijn. Indien een fragment door updates veranderd wordt, voer dan een primary copy in en laat elke update transactie daarop werken. Verbind de virtuele computer van de primary copy met alle andere virtuele computers die copieën hebben en label de lijnen met de som van de updates per eenheid van tijd. Veranderingen in de verwerkingsstrategieën graaf als gevolg van wijzigingen in de data allocatie zijn het zelfde als in de vorige sectie.

Als data allocatie algoritme kan dezelfde algoritme gebruikt worden als geïntroduceerd in de vorige sectie. De algoritme bepaalt dan niet alleen de locatie van de copieën om het totale netwerkverkeer te minimaliseren, doch ook het aantal benodigde copieën.

4.3. Experimentele resultaten

De data allocatie algoritme is weer toegepast op random gegenereerde verwerkingsstrategieën grafen. Ook voor dit algemenere model bereikte de algoritme tijdens alle testruns de optimale niet-redundante allocatie. Een vergelijking voor redundante allocaties is practisch gezien onmogelijk door het grote aantal mogelijkheden. Ook zijn een aantal eigenschappen, als functie van de query/update ratio, van de door de algoritme geproduceerde allocaties onderzocht. Indien er alleen maar updates zijn zal er van elk fragment maar één copie in het netwerk aanwezig zijn, oftewel er is sprake van een niet-redundante allocatie. Indien we geleidelijk aan de updates vervangen door queries zien we dat er langzaam aan wordt overgegaan op een redundante allocatie. In het andere extreme geval dat er alleen maar queries zijn, is het duidelijk dat er totaal geen netwerkverkeer is. Immers elke computer heeft een copie van elke fragment dus query verwerking kost niets en er is ook geen netwerkverkeer nodig om de verschillende copieën consistent te houden. Door het overgaan op een redundante allocatie bedraagt het query verkeer nooit meer dan 50% van het totale netwerkverkeer. Anders gezegd, het grootste gedeelte van het totale netwerkverkeer is nodig om updates te verwerken en om copieën consistent te houden.

Verder is bekeken de uitsplitsing van zowel het query als update verkeer naar verkeer tussen fragmenten onderling en tussen de fragmenten en de resultaat computer. Daaruit bleek dat de algoritme de fragmenten zoveel mogelijk samenvoegde, want het verkeer tussen fragmenten onderling is maar betrekkelijk klein. Dit wetende is het interessant om te zien over hoeveel computers de relaties zijn verdeeld per transactie. De gegenereerde verwerkingsstrategieën grafen bevatten gemiddeld 4 fragmenten per transactie. Het resultaat was dat er in meer dan 70% van de transacties niet meer dan 2 computers waren betrokken bij de verwerking van de transacties, onafhankelijk van de query/update ratio.

4.4. Variaties op allocatie algoritme

We zullen nu een paar variaties op de algoritme bespreken. In de versie van sectie 4.1 worden alleen fragmenten van paren virtuele computers beschouwd om samengenomen te worden. Er zijn echter voorbeelden te geven waarbij het samennemen van 3 of meer virtuele computers beter is dan ze

apart te laten, maar dat het samennemen van fragmenten van een willekeurig tweetal tot verhoging van het netwerkverkeer leidt. Daarom zal de kwaliteit van de oplossing beter zijn als de volgende strategie wordt toegepast: beschouw eerst paren virtuele computers; nadat dit niet meer tot verbetering leidt, beschouw dan drietallen van virtuele computers, etc. Om de polynomialiteit van de algoritme te handhaven, mag dit maar een eindig aantal keren herhaald worden.

Een andere variatie om de flexibiliteit van query verwerking toe te staan, is om nadat de algoritme een beslissing heeft genomen de fragmenten van een groep virtuele computers bij elkaar te nemen, de strategieën van de betrokken queries op nieuw te berekenen, met het doel de verwerkingsstrategieën graaf aan te passen. Dit zal kostbaar zijn, maar de uiteindelijke allocatie zal beter rekening houden met de grillige wijzigingen van verwerkingsstrategieën. Een verdere uitbreiding van deze variatie is dat de lijnen in de verwerkingsstrategieën graaf niet meer overeen behoeven te komen met echte transmissies doch aangeven wat de vermindering is van het totale netwerkverkeer indien de aangrenzende punten samengenomen worden.

Om toch enigszins de aanname dat lokaal data verwerken niets kost, te verzwakken is het mogelijk om m.b.v. een simpel wachttijd model te controleren of een computer niet een te lange wachttijd krijgt, indien de fragmenten van twee virtuele computers worden samengenomen (zie ook GYLYS en EDWARDS[19]).

Al deze variaties maken de algoritme duurder, maar zullen de kwaliteit van de geproduceerde allocaties verbeteren. De winst verkregen door de toepassing van de variaties zal moeten blijken in de praktijk.

5. MINIMALISEREN VAN GEMIDDELDE RESPONSTIJD

Tot dusver hebben we alleen maar het minimaliseren van het totale netwerkverkeer bekeken. Dit zal vooral belangrijk zijn voor netwerken waar de transmissies van data van de ene computer naar de andere vele malen duurder zijn dan het lokaal verwerken van data. Indien dit niet het geval is, zoals bijv. voor lokale netwerken, zal het minimaliseren van de responstijd van de queries veel belangrijker zijn. Uiteindelijk heeft een gebruiker slechts te maken met de tijd die het duurt om zijn query op te lossen. Het gebruik van de responstijd als kostenfunctie is echter niet beperkt tot lokale netwerken, maar is ook toepasbaar voor algemenere

netwerken.

Om in staat te zijn de responstijden van queries te berekenen, moeten we een wachttijd model hebben om de gemiddelde wachttijden voor executie op de verschillende computers te bepalen. We nemen een simpel model zodat voor de verschillende grootheden analytisch formules zijn af te leiden. De voorgestelde algoritmen zijn volledig onafhankelijk van de wijze waarop de wachttijden berekend worden en zijn dus ook toepasbaar als meer geavanceerde wachttijd modellen worden gebruikt.

5.1. Wachttijd model

We veronderstellen dat het aanbod van de i-de query beschreven kan worden d.m.v. een exponentiële verdeling met als gemiddeld aanbod, λ_i. Een query zal uit een aantal operaties op verschillende computers bestaan, die soms op elkaar moeten wachten. Ondanks deze afhankelijkheden zal in navolging van JACKSON[24] verondersteld worden, dat het aanbod van operaties van een specifieke query op de verschillende computers als onafhankelijk kan worden beschouwd en dat dat aanbod voor die verschillende computers ook weer beschreven kan worden door een exponentiële verdeling. Het gemiddelde aanbod van een computer is dan de som van de λ_i's van de aangeboden operaties. Merk op dat de som van 2 exponentiële verdelingen weer een exponentiële verdeling is.

De verwerkingstijd van een operatie wordt bepaald door de hoeveelheid I/O die er gedaan moet worden en de hoeveelheid CPU tijd die nodig is om de bewerking uit te voeren. Omdat deze tijd sterk afhankelijk is van de hoeveelheid te verwerken data, zullen we hier een willekeurige verdeling veronderstellen. In wachttijd theorie termen hebben we hier te maken met het M/G/1 model en de resultaten hiervan worden toegepast om voor iedere computer de gemiddelde wachttijd te bepalen.

$$W = \frac{\lambda \overline{x^2}(1+C_b^2)}{2(1-\lambda \overline{x})}, \qquad C_b^2 = \frac{\overline{x^2}-\overline{x}^2}{\overline{x}^2}$$

W is de gemiddelde wachttijd,

λ is het gemiddelde aanbod,
$\overline{x}$ is de gemiddelde verwerkingstijd,
$\overline{x^2}$ is het 2de moment van de verwerkingsverdeling.

De verwerkingsverdelingen van de computers zijn niet bekend en daarom moeten de waarden van $\overline{x}$ en $\overline{x^2}$ bepaald worden door schatters. In APERS[6] wordt uitvoeriger ingegaan op dit model en zal een formule voor de wachttijd worden afgeleid, die meer rekening houdt met het afwijkende aanbod van operaties als meerdere fragmenten aan dezelfde computer zijn geassigneerd.

De responstijd van een operatie is nu de wachttijd van de computer waar hij verwerkt wordt plus de verwerkingstijd. De responstijd van een query volgt nu van zelf door ook de transport tijden mee te tellen en rekening te houden met het feit dat sommige bewerkingen en transporten parallel plaats vinden. Ook voor communicatie lijnen moet eigenlijk een wachttijd model worden toegepast om de wachttijd voor transport te bepalen. Hier zullen we er echter vanuit gaan dat de tijd die nodig is om X bytes over een lijn tussen twee willekeurige computers te versturen gelijk is aan $T(X) = T_0 + T_1.X$, waar T_0 de wachttijd weergeeft en T_1 bepaald wordt door de baud-rate. Het doel in dit hoofdstuk is de gemiddelde responstijd

$$\frac{\lambda_1.RT_1 + \lambda_2.RT_2 + \ldots + \lambda_k.RT_k}{\lambda_1 + \lambda_2 + \ldots + \lambda_k}$$

te minimaliseren, waarin RT_i de responstijd is van de i-de query.

5.2. Operatie allocatie

In de verwerkingsstrategieën graaf kunnen aan VS-punten niet alleen fragmenten maar ook operaties geassigneerd worden. Bij het minimaliseren van het totale netwerkverkeer werden deze operaties aan dezelfde virtuele computers geassigneerd als de fragmenten waarop ze werken. Wat betreft de selectie en projectie is dat ook bij het minimaliseren van de gemiddelde responstijd geen bezwaar. Want stel dat een selectie niet wordt uitgevoerd op de computer waaraan het fragment is geassigneerd, dan zal het hele fragment van disk gelezen worden zonder gebruikt te maken van (secondary) indices en overgestuurd worden naar een andere computer, alvorens de selectie wordt uitgevoerd. Dit is beduidend duurder. Anders ligt het met een

join. Deze werkt over het algemeen op operanden, die resultaten zijn van eerdere operaties. Het is niet a priori duidelijk of deze join uitgevoerd moet worden door de computer waaraan één van de operanden is geassigneerd of door een derde computer. Punt is hier dat een data allocatie niet volledig de operatie allocatie vast legt. We zullen hier een heuristisch algoritme geven die een operatie allocatie berekent met het doel de gemiddelde responstijd te minimaliseren. In APERS[6] gaan we verder in op de kwaliteit van de geproduceerde operatie allocaties.

Een operatie is gebonden als reeds bepaald is op welke computer hij verwerkt zal worden. Van een aantal operaties is, gegeven de data allocatie, meteen bekend dat ze gebonden zijn. Voor de vrije, dus niet gebonden, operaties moet nog een locatie gevonden worden. Voor het gemak veronderstellen we dat de query een boomstructuur heeft. Een vrije operatie is kandidaat om gebonden te worden, indien alle inputs van deze operatie reeds gebonden zijn. Uit de verzameling van kandidaten wordt er één gekozen, om gebonden te worden.

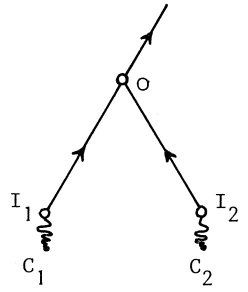


Fig. 5: Operatie boom.

Deze binding gaat als volgt. Veronderstel de situatie van fig. 5: operatie O is gekozen om gebonden te worden, d.w.z. I_1 en I_2 zijn al gebonden. Het doel van de binding van O is het minimaliseren van zijn responstijd. De mogelijke bindingen zijn:

- O wordt gebonden aan C_1, waar I_1 reeds gebonden is.

- O wordt gebonden aan C_2, waar I_2 reeds gebonden is.

- O wordt gebonden aan een andere computer dan C_1 en C_2.

Voor elk van deze 3 mogelijkheden moet de responstijd van O berekend worden. We bekijken het algemene geval dat O gebonden is aan C_3, een computer waaraan zowel I_1 als I_2 niet gebonden zijn. I_1 is aan C_1 gebonden z.d.d. zijn responstijd minimaal is; er wordt echter geen rekening gehouden met de data transmissie van I_1 naar O. Om de responstijd van I_1 plus transmissie te minimaliseren is het wellicht nodig een gedeelte van de query boom onder I_1 te verwijderen van C_1 en te binden met C_3. Omdat de query een boomstructuur heeft, kan in een eindig aantal stappen bepaald worden om welk gedeelte het gaat. Hetzelfde moet voor I_2 gedaan worden. Voor O wordt die binding gekozen die zijn responstijd minimaliseert.

Door dit steeds te herhalen zal uiteindelijk de operatie gekozen worden, die de output verzorgt voor de gebruiker. Van deze operatie is bekend dat hij aan de computer gebonden is waar de query gesteld werd. Dus de verschillende bindingen zoals bij bovenstaande operatie O behoeven niet onderzocht worden. Echter wel de wijzigingen in de bindingen van de input operaties om de responstijd te minimaliseren.

De algoritme ziet er als volgt uit:

```
bind de operaties die data lezen;
while er nog vrije operaties zijn
do
     kies een vrije operatie O, waarvan alle input operaties reeds
     gebonden zijn;
     bepaal aan welke computer O gebonden zal worden z.d.d. zijn res-
     ponstijd wordt geminimaliseerd
od
```

Elke operatie heeft een λ (frequentie waarmee hij uitgevoerd wordt) en een x (executie tijd). Bij het kiezen van O kan rekening gehouden worden met de grootte van λ, z.d.d. "grotere" operaties eerder gekozen worden dan "kleinere" operaties. De reden voor dit is dat bij het bepalen aan welke computer O gebonden moet worden, er ook rekening gehouden moet worden met de operaties die reeds gebonden waren aan die computer. Indien bijv. O gebonden wordt aan computer C zal de gemiddelde wachttijd van C toenemen.

Oftewel, de responstijden van andere operaties zullen toenemen. Daarom zal het voordeel van het binden van O aan C afgewogen moeten worden tegen het nadeel van de reeds aan C gebonden operaties.

5.3. Data allocatie

In de vorige sectie berekenden we de gemiddelde responstijd van de queries gegeven een bepaalde data allocatie. Het uiteindelijke doel is een data allocatie te vinden zodat, nadat ook een operatie allocatie heeft plaatsgevonden, een minimale responstijd wordt bereikt. We zullen echter een iets eenvoudiger probleem bekijken. We nemen aan dat na de data allocatie alle operaties gebonden zijn.

De idee achter de algoritme, die tot doel heeft de gemiddelde responstijd te minimaliseren, is dat elke query een voorstel doet om de data allocatie te veranderen om zijn eigen responstijd te verminderen. Om echter over de responstijd van een query te spreken, moet er wel een data en operatie allocatie zijn. Identificeer daarom de virtuele computers met fysieke computers z.d.d. de wachttijden op de verschillende computers ongeveer hetzelfde zijn. Het doel hiervan is de responstijd van de individuele operaties te minimaliseren. Dit is een goede methode om de responstijd van queries te minimaliseren indien de transmissie tijd betrekkelijk gering is vergeleken bij de executie tijd.

Stel we hebben op bovenstaand beschreven wijze een data allocatie verkregen. Bereken nu voor elke query de responstijd en bepaal ook welk pad (transmissies en executies) in de query boom deze responstijd veroorzaakt. Andere paden behoeven niet beschouwd te worden omdat veranderingen in deze geen invloed hebben op de responstijd. Alle mogelijke wijzigingen in de data allocatie worden voor dit pad beschouwd. De twee basis mogelijkheden zijn: 1) identificeer virtuele computer VS met computer C en 2) voeg de fragmenten van de virtuele computers VS_i en VS_j samen. Van al deze voorgestelde wijzigingen kan berekend worden wat de invloed is op de responstijd van de betrokken query. Nadat de voorgestelde wijzigingen van alle queries verzameld zijn, wordt die wijziging gekozen waarvan de verwachte verbetering van de gemiddelde responstijd het best is. Om te zien of deze wijziging ook echt een verbetering van de responstijd inhoudt, wordt een nieuwe data allocatie berekend inclusief de voorgestelde wijziging. Vermindert de responstijd dan wordt de wijziging geaccepteerd. Is er echter

sprake van een verhoging dan zal de voorgestelde wijziging met de op één na beste verwachte verbetering van de gemiddelde responstijd worden beschouwd. Herhaal dit net zo lang tot dat een wijziging geaccepteerd wordt. Met de dan verkregen allocatie wordt weer van vooraf aan begonnen met het berekenen van de responstijd verminderende voorstellen van de individuele queries. Indien er geen wijziging te vinden is die de gemiddelde responstijd vermindert, is de uiteindelijke allocatie bereikt.

6. ALLOCATIE PROBLEMATIEK IN GESPREIDE DATABASES

In de voorgaande hoofdstukken is een model ingevoerd om de verwerkingsstrategieën te kunnen representeren en tevens zijn algoritmen geïntroduceerd om allocaties te berekenen die kostenfuncties minimaliseren. De vraag is nu, hoe past dit in de algehele allocatie problematiek van een gespreide database. Een aantal o.i. belangrijke problemen zullen we bespreken.

In het model wordt verondersteld dat de fragmenten die gealloceerd moeten worden, van te voren bekend zijn. Dit is natuurlijk niet waar. Op grond van de relaties in het conceptuele schema is het op geen enkele wijze mogelijk om deze fragmenten te bepalen. Een methode is ontwikkeld om gegeven de queries (of alleen maar views) van de gebruikers, de relaties zodanig in stukjes te snijden dat er niet-overlappende fragmenten overblijven (APERS[5]). Deze fragmenten zullen als ondeelbaar beschouwd worden, omdat een verdere verdeling de complexiteit van het allocatie probleem sterk zal vergroten en de winst in efficiëntie marginaal zal zijn. De op deze wijze verkregen fragmenten vormen de invoer van de allocatie algoritmen.

De bepaling van een allocatie is statisch en bij de berekening van een nieuwe allocatie, als bijv. de frequentie van het gebruik van de querie veranderd is, wordt er geen rekening gehouden met een al bestaande allocatie en de kosten die nodig zijn om deze aan te passen. De re-organisatie van een allocatie is een eenmalige investering die afgewogen moet worden tegen een voortdurende verbetering in efficiëntie. Een manier om dit te doen is te eisen dat de winst in efficiëntie binnen een gestelde termijn opweegt tegen de investeringskosten. Als we uitgaan van een bestaande allocatie zullen in de verwerkingsstrategieën graaf de virtuele computers van de fragmenten verbonden worden met de computers waaraan ze in de

huidige allocatie geassigneerd waren. Een nieuwe allocatie kan nu m.b.v. de data allocatie algoritmen berekend worden, en tegelijkertijd wordt er rekening gehouden met de re-organisatiekosten.

Gebruikers kunnen queries stellen waarvan de beantwoording tijdkritisch is. In het file allocatie probleem werden deze "constraints" gewoon toegevoegd aan de overigen die het probleem beschrijven. Het komt er dan op neer dat alle gebruikers mee betalen om voor een klein groep gebruikers een goede responstijd te verzorgen. Binnen één bedrijf is dat misschien nog acceptabel, maar in netwerken waarin databases voorkomen van verschillende bedrijven niet. Een gebruiker die dit soort eisen stelt, zal meer moeten betalen voor de extra belasting van het netwerk en de computers.

Een laatste punt dat we willen bespreken is bij wie of wat de verantwoordelijkheid voor de efficiëntie van de allocatie ligt in een decentraal beheerde database. Er is een tendens om de verantwoordelijkheid te geven aan de eigenaren of gebruikers van de data. Een potentiëel probleem hier is dat de eigenaar om emotionele redenen beslissingen omtrent de opslag of distributie zal nemen, die een nadelig effect hebben op de efficiëntie. Wij pleiten er dan ook voor dat algoritmen, die bij de bepaling van de allocatie zoveel mogelijk rekening houden met de efficiëntie, beschikbaarheid en security, de verantwoordelijkheid hebben over de allocatie en dat alleen om redenen die buiten de skoop vallen van de algoritmen, de eigenaar kan interveniëren.

CONCLUSIE

Het file allocatie probleem is gegeneraliseerd zodat de complexe verwerkingsstrategieën, die gebruikt worden in gespreide databases, gemodelleerd kunnen worden. Voor netwerken waarbij data transmissies de kostenfactor zijn, is een data allocatie algoritme gegeven, die het totale netwerkverkeer minimaliseert. Uit experimenten blijken de geproduceerde allocaties veelal optimaal te zijn. Een ander algoritme is ontwikkeld, die niet alleen rekening houdt met de data allocatie, maar ook met de operatie allocatie, voor het minimaliseren van de gemiddelde responstijd. Tenslotte is beproken hoe deze gereedschappen passen in de algehele problematiek van een gespreide database.

LITERATUUR

[1] APERS, P.M.G., Distributed Query Processing: Minimum Response Time Schedules for Relations, IR 50, maart 1979, Vrije Universiteit, Amsterdam.

[2] APERS, P.M.G., Data Allocation and Distributed Query Processing, Proc. ACM PACIFIC '80, San Francisco, november 1980, pp. 48-54.

[3] APERS, P.M.G., Redundant Allocation of Relations in a Communication Network, Proc. Fifth Berkeley Workshop on Distributed Data Management and Computer Networks, San Francisco, februari 1981, pp.245-258.

[4] APERS, P.M.G., A.R. HEVNER & S.B. YAO, Improved Distributed Query Processing Algorithm GENERAL, ter publicatie aangeboden, 1980.

[5] APERS, P.M.G., Centralized or Decentralized Data Allocation, ter publicatie aangeboden, 1981.

[6] APERS, P.M.G., Data Allocation: Minimizing Response Time, IR, Vrije Universiteit, Amsterdam, verschijnt binnenkort.

[7] BALDISSERA, C., G. BRACCHI & S. CERI, A Query Processing Strategy for Distributed Data Bases, Proc. EURO-IFIP 1979, North-Holland Publ. Co. Amsterdam, 1979, pp.667-677.

[8] CASEY, R.G., Allocation of Copies of Files in an Information Network, Proc. AFIPS 1972 SJCC, AFIPS Press, vol.40, 1972, pp.617-625.

[9] CHU, W.W., Optimal File Allocation in a Multiple-Computer Information System, IEEE Trans. Computers, C-18, 1969, pp. 885-889.

[10] CHU, W.W., Optimal File Allocation in a Computer Networks, Computer Communication Network, Prentice-Hall, Englewood Cliffs N.J., 1973.

[11] CHU, W.W. & P. HURLEY, A Model for Optimal Processing for Distributed Databases, Proc. 18th IEEE COMPCON, Spring 1979, pp. 116-122.

[12] CHU, W.W., L.J. HOLLOWAY, MIN-TSUNG LAN & K. EFE, Task Allocation in Distributed Data Processing, IEEE Computer, november 1980, pp. 57-69.

[13] ELAM, J., A Model for Distributing a Database, Dept. of Decision Sciences, The Wharton School, University of Pennsylvania, Philadelphia, maart 1978,

[14] EPSTEIN, R., M.R. STONEBRAKER & E. WONG, Distributed Query Processing in a Relational Data Base System, Proc. ACM-SIGMOD, mei 1979, pp.169-180.

[15] EPSTEIN, R. & M.R. STONEBRAKER, Analysis of distributed data base

processing strategies, Proc. Sixth Conf. on Very Large Data Bases, Montreal, oktober 1980, pp. 92-101.

[16] EPSTEIN, R., Query Processing Techniques for Distributed Data Base Systems, PhD. Thesis Memorandum No. UCB/ERL M80/9, Univ. Calif. Berkeley, maart 1980.

[17] ESWARAN, K.P., Placement of records in a file and file allocation in a computer network, Information Processing 1974, North-Holland Publ. Co., Amsterdam, 1974, pp.304-307.

[18] GRAPA, E. & G.G. BELFORD, Some Theorems to Aid in Solving the File Allocation Problem, Communications ACM, vol.20, no.11, november 1977, pp.878-882.

[19] GYLYS, V.B. & J.A. EDWARDS, Optimal Partitioning of Workload for Distributed Systems, Digest of Papers, COMPCON Fall 1976, september 1976, pp. 353-357.

[20] HEVNER, A.R. & S.B. YAO, Query Processing in Distributed Database Systems, IEEE Transactions on Software Engineering, vol. SE-5, no.3, mei 1979, pp.177-187.

[21] HEVNER, A.R., The Optimization of Query Processing on Distributed Database Systems, PhD thesis, Purdue University, December 1979.

[22] HOFFER, J.A. & D.G. SEVERANCE, The use of cluster analysis in physical data base design, Proc. First Conf. on Very Large Data Base, Framington, Mass., september 1975, pp. 69-86.

[23] IN-SUP PAIK & C. DELOBEL, A strategy for optimizing the distributed query processing,

[24] JACKSON, J.R., Queueing Systems, Operations Research, vol. 5, 1957, pp. 518-521.

[25] LEVIN, K.D., Organizing Distributed Data Bases in Computer Networks, The Wharton School, University of Pennsylvania, Philadelphia, september 1974.

[26] LEVIN, K.D. & H.L. MORGAN, Optimizing Distributed Databases- A Framework for Research, Proc. 1975 AFIPS NCC, AFIPS Press, vol.44, 1975, pp.473-478.

[27] MAHOUD, S. & J.S. RIORDON, Optimal Allocation of Resources in Distributed Information Networks, ACM Transactions on Database Systems, vol.1, no.1, maart 1976, pp.66-78.

[28] NGUYEN GIA TOAN, A unified method for query decomposition and shared information updating in distributed systems, First Int. Conf. on Distributed Computing Systems, Huntsville (Alabama), oktober

1979, pp. 679-685.

[29] NGUYEN GIA TOAN, Decentralized Dynamic Query Decomposition for Distributed Database Systems, Proc. ACM Pacific '80, San Francisco, november 1980, pp. 55-60.

[30] PELAGATTI, G. & F.A. SCHREIBER, A Model of an Access Strategy in a Distributed Database, IFIP-TC2, Data Base Architecture, Venice, juni 1979.

[31] RAMAMOORTHLY, C.V. & B.W. WAH, The Placement of Relations on a Distributed Relational Database, Proc. of the 1st International Conference on Distributed Computing Systems, Huntsville, oktober 1979, pp.642-650.

[32] ROTHNIE, J.B. & N. GOODMAN, An Overview of the Preliminary Design of SDD-1: A System for Distributed Databases, Proc. of Second Berkeley Workshop on Distributed Data Management and Computer Networks, mei 1977, pp. 39-57.

[33] STONEBRAKER, M.R. & E. NEUHOLD, A Distributed Version of INGRES, Proc. of Second Berkeley Workshop on Distributed Data Management and Computer Networks, mei 1977, pp. 217-235.

[34] WONG, E., Retrieving Dispersed Data from SDD-1: A System for Distributed Data Bases, Proc. Second Berkeley Workshop on Distributed Data Management and Computer Networks, mei 1977, pp.217-235.

PLAIN, DATABANKEN EN PROTECTIE

M.L. KERSTEN
Vrije Universiteit, Amsterdam

1. INLEIDING

In dit artikel wordt een overzicht gegeven van de onderzoeksactiviteiten die plaats vinden in het kader van het PLAIN project. Het onderzoek is geconcentreerd rond de taal PLAIN, een moderne hogere programmeertaal ontwikkeld voor de constructie van interactieve informatiesystemen. Binnen dit onderzoekskader ligt het zwaartepunt op de integratie van databankconcepten met programmeertaalconcepten en in het byzonder op de protectie van een databank tegen ongeoorloofde toegang en inhoud.

In het eerste gedeelte van dit artikel wordt de integratie van het datamodel met de overige taalconcepten in PLAIN gepresenteerd en worden er enige kanttekeningen geplaatst bij de databankmanipulaties. Vervolgens wordt in hoofdstuk 3 de invloed van de taal en het datamodel op de constructie van een relationeel databanksysteem geillustreerd.

In het tweede gedeelte van dit artikel zal worden gedemonstreerd hoe een databank met behulp van de concepten in PLAIN geconstrueerd kan worden. Hoofdstuk 5 bevat een overzicht van de formele modellen voor de beschrijving van de toegangsprotectie en hoofdstuk 6 geeft een inleiding van het in ontwikkeling zijnde SPE model.

In dit artikel worden de aspekten slechts summier belicht, voor uitgebreidere informatie wordt verwezen naar [14,16,22,19].

1.1. PLAIN project.

PLAIN (Programming LAnguage for INteraction) is een algemeen toepasbare, hogere programmeertaal, die wat betreft de structuur en de basisfaciliteiten sterk lijkt op de programmeertaal Pascal [9]. De taal PLAIN verschilt echter van andere moderne programmeertalen, zoals Ada [8], CLU [13] en Euclid [1], in de geboden faciliteiten voor de constructie van interactieve informatiesystemen. Een interactief informatiesysteem kan globaal gekarakteriseerd worden door:

1) Geparameteriseerde systeem/gebruiker interacties.

2) Grammaticale analyse en validatie van (niet) gestructureerde invoer.

3) Grote hoeveelheid van relatief eenvoudige databankmanipulaties.

4) Noodzaak voor flexibele foutenafhandeling.

Men kan bij een interactief informatiesysteem denken aan een ziekenhuis informatiesysteem (ZIS), telefoonservice (008) of een magazijnbeheerssysteem (Wehkamp). De faciliteiten die PLAIN biedt voor deze grote groep van systemen omvat:

- Relationele databankfaciliteiten.

- Mechanismen om patronen te definiëren en te herkennen.

- Exceptieafhandeling op procedurele basis.

- Mechanismen voor zowel procedurele als data-abstractie.

PLAIN is een van de weinige talen waarbij de integratie van deze faciliteiten tot stand gebracht is op het moment van de taaldefinitie. De gebruikelijke methode is een bestaande programmeertaal uit te breiden met (databank) faciliteiten zoals bijvoorbeeld in de CODASYL [5] aanpak en Pascal-R [9]. Het uitbreiden van een taal met databankfaciliteiten levert niet altijd een fraai resultaat op, omdat deze faciliteiten niet goed aansluiten bij de overige taalconcepten, zoals bijvoorbeeld te zien is in EQUEL in INGRES [18] en SQL in SQL/DS [2] of nog duidelijker, SQL gecombineerd met IBM 370 assembler [2]. Van de nieuwe talen met volledige integratie van databankfaciliteiten dienen RIGEL [17] en ASTRAL [3] te worden vermeld.

1.2. Een stukje geschiedenis.

Het ontwerp van de taal PLAIN is gestart in 1977 aan de afdeling Medische Informatica van de Universiteit van Californië te San Francisco door A. Wasserman. De eerste definitie van de taal verscheen in de zomer van 1978 [20]. Vanaf die tijd is onze groep, de Vakgroep Informatica van de Vrije Universiteit, actief betrokken bij zowel de verbetering van de definitie van de taal als de implementatie. Gedurende de afgelopen 3 jaar zijn een aantal ambiguiteiten en inconsistenties in het originele PLAIN rapport verbeterd, wat heeft geresulteerd in een Revised Report [19].

De implementatie- en onderzoeks- taken zijn tussen de twee universiteiten verdeeld. De PLAIN compiler wordt ontwikkeld in San Francisco (en Berkeley) en het onderzoek aldaar richt zich op het ontwerp en de implementatie van programmeer hulpmiddelen, het Module Control System [21]. Er is tot op dit moment echter nog geen werkzame PLAIN compiler beschikbaar, wat een grote belemmering vormt voor de toetsing van de ontwikkelde ideeën.

Aan de Vrije Universiteit is een relationeel databanksysteem in ontwikkeling ter ondersteuning van de taal [12]. Het onderzoek beweegt zich op het terrein van protectie van databanken in PLAIN omgevingen.

2. PLAIN EN HET RELATIONELE DATAMODEL

De databankfaciliteiten van PLAIN zijn gebaseerd op het relationele datamodel [4] en de bijbehorende algebraische datamanipulatietaal. In het relationele datamodel werken we met drie datastructurerings concepten: domeinen, relaties en attributen.

Een domein staat voor een verzameling waarden, bijvoorbeeld: de verzameling integers, de verzameling strings ter lengte 10 of de geordende verzameling (false,true).

Een relatie schema is een bundeling van (attribuut naam, domein naam) paren. Een attribuut naam beschrijft de rol, die het bijbehorende domein in de relatie speelt. De waarde van een relatie

$$R[\ (A_1,D_1),\ldots,(A_k,D_k)\ ]$$

is een deelverzameling van het Cartesiaans product

$$D_1 * D_2 .. D_k .$$

Een element van een relatie wordt wel een tupel genoemd, het aantal

elementen van een relatie heet de kardinaliteit van de relatie en k de rang van de relatie.

2.1. Domeinen.

Traditioneel zijn domeinen geassocieerd met de standaard datatypen van het databanksysteem, zoals integers, char, boolean en float. De reden hiervoor is dat deze datatypen het laagste niveau van abstractie vormen in een databanksysteem. Ook in PLAIN zijn de domeinen geassocieerd met de door de gebruiker te definiëren eenvoudige datatypen (simple types), die eerder genoemde de standaard datatypen omvatten. Daarnaast kunnen strings van variabele en vaste lengte worden gebruikt voor de definitie van nieuwe domeinen. Het mechanisme om nieuwe datatypen te definiëren in PLAIN maakt het mogelijk om op drie manieren nieuwe domeinen te introduceren:

1) door een "scalar" definitie
2) door herbenaming
3) door beperking van het waardenbereik van een datatype.

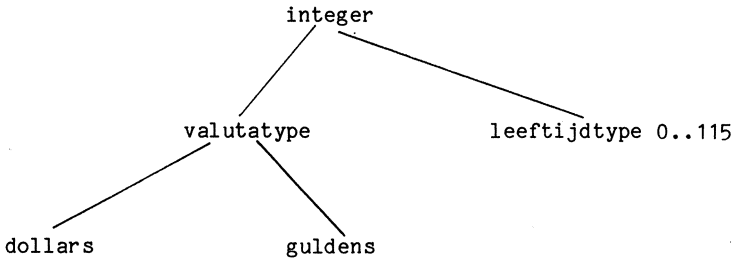
Een "scalar" definitie introduceert een nieuw datatype door opsomming van een (geordende) lijst van waarden, gerepresenteerd door namen.

Herbenaming van een bestaand type introduceert een nieuw type waarvan het waardenbereik gelijk is aan dat van het oude type. Het voornaamste gebruik van herbenaming is de bescherming tegen onbedoelde vergelijking en waardetoekenning (assignment).

Beperking van het waardenbereik wordt mogelijk gemaakt door het "subrange" mechanisme. Dit mechanisme is identiek aan de methode in Pascal, d.w.z. "subranges" kunnen alleen gedefineerd worden op de standaard datatypen. Dit is restrictiever dan de methode in Ada, waarin "subranges" gedefinieerd kunnen worden op elk eenvoudig datatype.

```
type valutatype = integer;
     leeftijdstype  = 0..115;
     dollars        = valutatype;
     guldens        = valutatype;
```

figuur 1 Voorbeeld domain definities.



figuur 2 Voorbeeld datatypenboom.

De in figuur 1 gedefinieerde typen zijn gerepresenteerd in de vorm van een datatypenboom in figuur 2. De datatypen boom wordt als volgt geinterpreteerd; twee objecten zijn vergelijkbaar of assigneerbaar als de bijbehorende datatypen op een pad naar de top van de boom liggen. In ons voorbeeld betekent dit dat dollars niet vergelijkbaar zijn met leeftijden of met guldens, maar wel met integers en valutatypeobjecten.

2.2. Relaties.

De relaties zijn in PLAIN geassocieerd met de datatype definitie en variabele declaratie mechanismen. De algemene vorm van een relatietype definitie wordt geillustreerd door:

```
type werknemerstype = relation
                          naam : string;
                          nummer: integer;
                          chefnr: integer;
                          sal   : guldens;
                          lftijd: leeftijdtype
                      end relation
```

Het relationele model eist dat een relatie een verzameling vormt, dat wil zeggen dat alle tupels in een relatie uniek zijn. In het algemeen hangt dit slechts af van een deelverzameling van de attributen van een relatie. In ons voorbeeld wordt het uniek zijn bepaald door 'nummer', maar waarschijnlijk ook door de combinatie 'naam' en 'sal'. Een van de

deelverzamelingen met deze eigenschap wordt uitverkoren om als primaire sleutel te dienen. De volledige definitie van het bovenstaande relatietype wordt dan:

```
type werknemerstype = relation [ key nummer ] of
                          naam : string;
                          nummer: integer;
                          chefnr: integer;
                          sal   : guldens;
                          lftijd: leeftijdtype
                      end relation

    {creatie van een relatie object}
var werknemer : werknemerstype;
```

Ook de relatietypes kunnen gegroepeerd worden in z.g. datatypen bomen, wat de gebruiker een eenvoudig hulpmiddel biedt ter bescherming van conversie en waarde toekenning.

2.3. Enige kanttekeningen bij de datamanipulatie.

De databankmanipulatieconcepten van PLAIN zijn reeds in een eerder artikel [15] in deze reeks uitgebreid aan de orde geweest. Hier volstaan we met een paar kritische kanttekeningen. De taal PLAIN is ontworpen met een efficiënte en eenvoudige implementatie in het achterhoofd. Deze filosofie heeft enerzijds in een aantal ogenschijnlijk 'overbodige' en anderzijds in een aantal 'beperkende' taalconcepten geresulteerd.

Het belangrijkste hulpmiddel in de taal voor het mogelijk maken van een efficiënt systeem is de introductie van hulpvariabelen, de z.g. markings, in combinatie met een op de relationele algebra georienteerde data-manipulatietaal [14]. Ter illustratie bekijken we hier het concept van tupelindicatoren en de beperkingen opgelegd aan de selectie-operatie.

2.3.1. Tupel indicatoren.

De primaire sleutel van een relatie kan worden gebruikt om een individueel tupel te benaderen op grond van de waarden van de sleutelattributen, wat een z.g. associatieve addressering geeft. Bijvoorbeeld:

```
werknemer[800105]
```

refereert naar een tupel in de relatie werknemers (indien aanwezig), en kan gebruikt worden voor de inspectie en tot wijziging van (niet sleutel) attributen, zoals bijvoorbeeld in:

```
werknemer[800105].sal := werknemer[800105].lftijd * 100
```

Een directe interpretatie van de componenten van deze operatie zou kunnen resulteren in tweemaal het (dure) opzoeken van het tupel werknemer[800105]. Voor dit soort, veel voorkomende, situaties is de "tupel indicator" uitgevonden. Een tupel indicator is korte notatie voor een efficiënte toegang voor herhaaldelijk gebruik van een en het zelfde tupel. Een tupel indicator krijgt een waarde door middel van een z.g. associatief tupelselectie-operatie. Elke relatie variabele naam kan gebruikt worden als een tupel indicator, mits gevolgd door een '%'. Bijvoorbeeld, bovenstaande operatie kan vervangen worden door:

```
werknemer% := werknemer[800105];
werknemer%.sal := werknemer%.lftijd *100;
```

Het weglaten van de taalconstructie voor tuple indicator zou ofwel tot een complex globaal optimalisatie proces leiden, danwel tot een complex databanksysteem ter voorkoming van overbodige zoekoperaties. Een nadeel van de huidige definitie van tuple indicatoren is de dubbele rol die een relatie naam speelt. Enerzijds fungeert de naam als referentie voor een relatie en anderzijds wordt impliciet een variabele gedeclareerd, die een referentie naar een (1) tuple bevat. Bovendien is het concept van een tuple indicator niet doorgevoerd als datatype. Voor de hand zou liggen tuple indicatoren te introduceren als databank pointer variabelen. Dit is echter uit oogpunt van de programmeerrisico's, die aan het onbeperkt gebruik van pointers kleven, niet gedaan.

2.3.2. De selectie-operatie.

Het tweede taalconcept waarbij enige kanttekeningen geplaatst kunnen worden is de selecte-operatie. De syntax voor een selectie-operatie wordt gegeven door:

```
dbselect ::= dbvariable where dbconditie
```

De dbconditie is syntactisch een normale boolean expressie, met de uitzondering dat een dbconditie-factor een attribuut-naam kan bevatten. Echter, de attribuut-naam kan niet gebruikt worden als een index in een array, noch als een parameter in een functieaanroep, noch in patroonherkenning.
Bijvoorbeeld:

```
t:= werknemer where lftijd in [21..23] ¦
                  taxperc[ sal ] >32 ¦
                  maxtax(sal) >5200 ¦
                  naam ?= lowercasepat;
```

is niet toegestaan in PLAIN. De reden voor de beperking van de complexiteit van de selectie operatie is voornamelijk ingegeven door de operaties die de relationele systemen anno 1978 boden. Hierbij moet nog opgemerkt worden dat bij het ontwerp van de taal ervan uit is gegaan dat het databanksysteem los van het PLAIN programma zou draaien. Communicatie tussen PLAIN programma en databanksysteem zou moeten geschieden via interproces communicatie. Als gevolg hiervan eist PLAIN dat alle complexe deelexpressies evalueerbaar zijn voordat de opdracht naar het databanksysteem gestuurd wordt.

Een alternatief zou zijn geweest deze problematiek te verbergen voor de gebruiker door code te genereren voor de equivalente operaties:

```
t:= [];  {Begin met leeg resultaat}
foreach e in werknemer
loop
    if lftijd in [21..23] ¦
         taxperc[ sal ] >32 ¦
         maxtax(sal) >5200 ¦
         naam ?= lowercasepat
    then
        { Voeg tupel aan resultaat toe}
        t:+ [e]
    end if
repeat
```

3. EEN DATABANKSYSTEEM VOOR PLAIN

De eisen die de taal PLAIN stelt aan een databanksysteem gaan uit boven de mogelijkheden die de meeste databanksystemen bieden. Ter illustratie: de meeste systemen missen goede faciliteiten voor de definitie van de domeinen (scalaren) en het manipuleren met een datatypenboom. Een gevolg hiervan is het ontwerp en de implementatie van een relationeel databanksysteem [11], deels toegespitst op de eisen die PLAIN stelt. De primaire taak van het databanksysteem is de concepten te leveren voor relatievariabelen, domeinadministratie en de bijbehorende operatoren.

Het eerste ontwerp van het systeem kwam begin 1979 gereed, waarna met de implementatie werd gestart. Op dit moment is een systeem beschikbaar voor zowel de ondersteuning van PLAIN als voor interactief gebruik.

3.1. Architectuurkeuze en criteria.

Voor het realiseren van een databank zijn er twee fundamenteel verschillende aanpakken mogelijk.

1) Een interpretatief systeem
2) Een compiler georienteerd systeem.

Een interpretatief systeem interpreteert, optimaliseert en voert de datadefinitie en datamanipulatie constructies van de programmeertaal rechtstreeks uit. Het voordeel van een interpretatief systeem is de vrijheid bij het definiëren van de interface. Een nadeel van een interpretatief systeem is het verlies aan efficiëntie door het uitvoeren van alle acties op runtime

niveau. De interpretatiemethode is in verschillende prototypen gehanteerd [18].

In de compiler georienteerde aanpak wordt reeds tijdens de vertaling van het applicatie programma de definitie van de databank gebruikt voor analyse en optimalisatie. Een gevolg hiervan is o.a. dat het databank systeem eenvoudiger van opzet kan zijn. Bovendien leidt deze aanpak tot een vermindering van de runtime overhead aan administratie activiteiten, als ook een vermindering van de overhead ten gevolge van de interpretatie van expressies ten behoeve van de selectie van tupels. Als voorbeeld voor deze aanpak kan SQL/DS [2] aangehaald worden.

De criteria voor onze keuze van de interpretatieve methode kunnen als volgt worden samengevat:

1) Een compileraanpak vereist het bestaan van een werkzame en stabiele compiler voor PLAIN. Deze is op dit moment nog niet beschikbaar.

2) Het ontwerp en de implementatie van de twee subsystemen, compiler- en databanksysteem, werden 9000 km van elkaar uitgevoerd. Daar dit tot communicatieproblemen leidt moet de interface tussen de twee subsystemen eenvoudig zijn.

3) De databankfaciliteiten moesten een zelfstandig subsysteem vormen. Het systeem zou ook vanuit andere programmeertalen, zoals Pascal en C, dan wel rapport generatoren en vraagtalen (query) systemen gebruikt moeten kunnen worden.

4) De databankfaciliteiten moesten de databank beschermen tegen eenvoudige integriteitsfouten, ongeacht de gebruikte interface.

3.2. Invloed van de taal op het databanksysteem.

De taal heeft de architectuur van het databanksysteem aanzienlijk beinvloed, hetgeen geillustreerd kan worden aan de hand van een aantal voorbeelden.

Domeinen

Daar de domeinen in de taal zijn geassocieerd met het mechanisme van datatypen definities, zal het databanksysteem alle informatie over de gebruikte datatypen moeten bewaren. In het bijzonder betekent dit dat de scalaren en het pad naar de top van de typenbomen moet worden bewaard. Bij hergebruik van een relatie moet worden gecontroleerd of de aanwezige informatie niet strijdig is met de definitie van de datatypen in het nieuwe programma.

Foreach-operatie.

De foreach operatie in PLAIN, vergelijkbaar met een for statement in Pascal, vereist een aantal laag niveau operaties om successievelijk alle tupels van de relatie te benaderen. Het concept voor deze lage operaties is een cursor (een variabele die een referentie bevat naar een tupel in een relatie) en 'reset', 'get next' and 'nil test' operaties. Ook de eerder vermelde tuple indicator wordt met behulp van een cursor gerealiseerd. Opgemerkt moet worden dat de cursors geen beveiliging bieden tegen wijziging, of verwijdering van het geselecteerde tupel.

Exceptiemechanisme.

Het foutenafhandelingsmechanisme in PLAIN bestaat uit het moderne exceptiemechanisme (zie [15]). De taal definieert een aantal exceptie namen voor veel voorkomende ongewenste situaties , zoals referentie naar een ongedefinieerd object en deling door nul. Het databanksysteem zal rekening moeten houden met deze ongewenste situaties. Bij interactief gebruik van het databanksysteem is gebleken dat het exceptiemechanisme te weinig informatie biedt. In de huidige versie wordt dan ook extra informatie aan de gebruiker gegeven.

Datamanipulatie

De keuze voor de algebraische datamanipulatiemethode vereenvoudigde de taak van het databanksysteem aanzienlijk, er behoefde geen ingewikkelde optimalisator gebouwd te worden.

3.3. Invloed van het databanksysteem op de taal.

De invloed van het databanksysteem op de taal is tot nu toe beperkt gebleven tot de introductie van een nieuwe exceptie, 'tupel verwijderd'. Tevens geschiedt op dit moment de communicatie tussen een PLAIN programma en het databanksysteem op basis van tekst, hetgeen de implementatie van de compiler heeft vereenvoudigd. In de toekomst zal echter ter verbetering van de communicatie efficiëntie een boodschappenprotocol worden gebruikt. Ook zal de taal beinvloed worden door de definitie van taalconcepten voor het reglementeren van het gelijktijdig gebruik van de databank en de mogelijkheid tot het ongedaan maken van de wijzigingen op de databank.

4. CONSTRUCTIE VAN EEN DATABANK MET PLAIN.

In het voorgaande hoofstuk zijn kort de databankfaciliteiten van PLAIN geillustreerd en de vraag die overblijft is: Hoe construeer je nu een databank met behulp van de concepten beschikbaar in PLAIN ?
In dit hoofdstuk zal slechts een voorlopig antwoord gegeven kunnen worden. Bij de beantwoording wordt ervan uitgegaan dat de ontwerpfase van de databank is voltooid en heeft geresulteerd in een formele beschrijving van de databank in de vorm van een conceptueel schema, de z.g. community view, en externe schema's, de z.g. user views.

De vraag die we ons kunnen stellen is, hoe beelden we de schema's af naar de PLAIN datatypen, data objecten en procedures. Deze afbeelding kan in potentie leiden tot een ongestructureerde, onoverzichtelijke en onbeheersbare verzameling van PLAIN code. Een constructiedisipline ondersteund door zowel de semantiek van de taal, als door de PLAIN compiler en de programma-onderhoudshulpmiddelen is een noodzaak voor een adequate oplossing.

De hier voorgestelde methode voor databankdefinitie is gebaseerd op het gebruik van programmatuurlagen. Elke programmatuurlaag heeft een specifieke taak en resulteert in een nieuwe, abstracte definitie van de databank.

niveau 0 De relaties van de databank worden gedefinieerd met behulp van de datatypen en de variable declaraties in PLAIN.

```
type werknemerstype = relation [ key nummer ] of
                          naam : string;
                          nummer: integer;
                          chefnr: integer;
                          sal   : guldens;
                          lftijd: leeftijdtype
                      end relation

     afdtype = relation [ key afdnr ]
                   afdnr: integer;
                   afnaam: string
               end relation ;

     var werknemers : werknemerstype;
         afdelingen : afdtype
```

niveau 1 Regels ten behoeve van de integriteit van de afzonderlijke databankrelaties, die niet met behulp van de datatypen kunnen worden gedefinieerd, worden gewaarborgd door ze op te nemen in nieuw gedefinieerde (primitieve) operaties (zoals insert en delete), die de basis zullen vormen voor het volgende niveau.
Voorbeelden:
- Alle werknemers hebben een chef.
- Een salaris is altijd groter dan leeftijd * f100,--.

Daarnaast worden enkelvoudige aggregaatfuncties gedefinieerd, d.w.z. virtuele dataelementen, gebaseerd op een (1) onderliggende relatie.
Voorbeelden:
- Het aantal werknemers in een afdeling.
- Het gemiddelde salaris van alle werknemers.

niveau 2 Regels ten behoeve van de semantische integriteit tussen verschillende relaties worden gewaarborgd door de herdefinitie en groepering van de basisoperaties.
Voorbeeld: - De eis dat elke werknemer tot een bestaande afdeling moet behoren resulteert in een nieuwe insertie-operatie op de werknemersrelatie. Deze operatie zal voordat de werknemer toegevoegd wordt, moeten controleren of de gespecificeerde afdeling

voorkomt in de afdelingsrelatie.

Daarnaast worden op dit niveau de meervoudige aggregaatfuncties gedefinieerd.

niveau 3 De operaties van niveau 2 worden gegroepeerd tot atomaire transacties, die de toegestane toestandsovergangen van de databank beschrijven. Als zodanig beschrijft deze laag de voor de gebruiker beschikbare operaties op de databank, waarbij voorzieningen zijn getroffen voor het gebruik van gedeelten van de databank door meerdere gebruikers tegelijkertijd.

niveau 4 Op het hoogste niveau vinden we de applicatieprogramma's.

Elk niveau levert een nieuwe collectie van operaties op de databank op en als zodanig wordt de databank benaderd door een aantal abstracte machines. Ideaal zou zijn als de basisoperatoren op relaties (:+,:-) hergedefinieerd zouden kunnen worden met behoud van de syntax, ook wel het overladen van een operator genoemd. PLAIN biedt deze faciliteit echter niet en daardoor zal een herdefinitie resulteren in een procedure of functie declaratie. Het zal duidelijk zijn dat deze lagen onder de verantwoordelijkheid vallen van de databankbeheerder.

Om deze aanpak te kunnen laten werken moet uit oogpunt van beveiliging aan twee belangrijke eisen worden voldaan:

1) Niveau i+1 mag alleen gebruik maken van de objecten en operaties van niveau i en die van niveau i-1 voorzover deze niet op niveau i hergedefinieerd zijn.

2) Op niveau i moeten mechanismen aanwezig zijn om het gebruik door niveau i+1 van objecten en operaties van niveau i-1 te beperken.

De tweede eis vloeit voort uit het bestaan van externe schema's, welke het gebruik van de databank voor een groep gebruikers beperkt.

Voor de verwezenlijking van deze methode komen in principe drie taalconcepten in aanmerking:

a) Abstract Data Types (ADT).

b) Scoperegels.

c) Programmabibliotheken

Achtereenvolgens zullen deze drie methoden onderzocht worden op de (on)mogelijkheden tot verwezenlijking van de databank met behulp van PLAIN.

4.1. ADT's voor databankdefinitie.

Een Abstract DataType (ADT) definieert een collectie van objecten, die gekarakteriseerd worden door de mogelijke operaties, en waarvan de representatie verborgen is voor de gebruiker. In een eerder in deze reeks verschenen artikel [15] is een uitgebreid voorbeeld opgenomen, dat de mogelijkheden van het gebruik van een ADT in PLAIN illustreert voor de definitie van een eenvoudige databank. Een rechtstreekse uitbreiding van die methode is de definitie van een ADT voor niveau i, die als representatie een of meerdere objecten van niveau i-1 bevat. De aldus verkregen datatype structuur vormt een boom, alle mogelijke operaties op de databank zijn gedefinieerd op het hoogste niveau en voor een databank zal slechts een (1) object van het ADT gedeclareerd behoeven te worden. Elk applicatieprogramma zal van dit object gebruik maken door middel van de zichtbare operaties.

De tweede eis, een opdeling van de operaties over verschillende, mogelijk overlappende groepen kunnen we met behulp van de abstracte datatypen niet bereiken. In het eerder geciteerde artikel wordt voor de protectie gebruik gemaakt van het concept van Userid. Als het gebruik van een operatie beperkt moet worden tot een selecte groep van gebruikers, dan dient er een test op de inhoud van het Userid plaats te vinden in de definitie van de operatie zelf.

Deze methode is echter minder geschikt voor het beperken van de verzameling van mogelijke databankoperaties binnen een applicatie programma. Het biedt de databankbeheerder geen mogelijkheid te voorkomen dat een salarismutatie gecombineerd wordt met een vrachtafhandelings transactie, oftewel de context waarbinnen een transactie plaats kan vinden valt buiten de controle van de databankbeheerder.

4.2. Scoperegels voor databankdefinitie.

Een tweede concept dat we zouden kunnen gebruiken voor de definitie van een databank is gebruik te maken van de mogelijkheden die de scoperegels ons bieden. In PLAIN is de scope van een variabele, functies en procedure gedefinieerd zoals in ALGOL60, dus d.m.v. een geneste blokstructuur. In PLAIN vormen de programma's, procedures, functies en module typen (ADT) echter gesloten scopes, d.w.z dat een variable, procedure en functie expliciet geimporteerd moet worden, d.m.v. een import operatie, alvorens binnen een scope te kunnen worden gebruikt. Tevens dienen alle indirect benaderde variabelen en programma-eenheden geimporteerd te worden. Het voordeel van deze zeer stringente regel is het beschermen van de programmeur tegen mogelijke fouten. Het nadeel is echter een grotere hoeveelheid specificaties.

Deze scope regels stellen de databankbeheerder echter niet in staat op adequate wijze de scheiding tussen de verschillende niveaus tot stand te brengen. Objecten op niveau i zullen gebruik maken van objecten op niveau i-1, deze laatsten zullen dus gedefinieerd moeten worden in de scope die de definitie van niveau i omsluit en eventueel door een import declaratie binnen niveau i beschikbaar gemaakt moeten worden. Hetzelfde verhaal gaat op voor de relatie tussen niveau i-1 en i-2. Maar dit impliceert dat met de geimporteerde niveau i-1 objecten ook de niveau i-2 objecten geimporteerd moeten worden.

De taal biedt de mogelijkheid om het gebruik van variabelen te beperken tot een verzameling scopes, d.m.v. de restrictedto optie in de declaratie van variabelen. Willen we echter dit mechanisme gebruiken voor de definitie van de databank, dan zou deze taalconstructie ook beschikbaar moeten zijn voor de andere programma objecten, zoals procedures en functies, en de betekenis van deze constructie in verhouding tot de scope regels nader moeten worden uitgewerkt.

4.3. Programmabibliotheken voor databankdefinitie.

De tekortkomingen van de ADT's en de scope regels werden veroorzaakt door de reeds bestaande semantiek van deze concepten. Willen we een databank definiëren, die aan de bovengestelde eisen voldoet, kan dat alleen nog geschieden door op het niveau van PLAIN programma's nieuwe concepten te definiëren. Oftewel, wat zijn de relaties tussen PLAIN programma's en hoe

zijn die te manipuleren.

Het gebied van programma of routine bibliotheken is bij de meeste programmeertalen en systemen onderontwikkeld. Een bibliotheek wordt in de meeste gevallen gezien als een ongestructureerde verzameling van programma teksten en/of code. Bescherming van deze bibliotheken tegen ongeoorloofd gebruik wordt uitgevoerd op het niveau van file protectie door het bedrijfssysteem.

Een alternatieve benadering is een PLAIN programma te zien als een definitie van een collectie objecten, zoals datatypen, variabelen, procedures, etc. Normaliter zal een geactiveerd PLAIN programma vertaald en uitgevoerd worden in een voorgedefinieerde omgeving, de systeem omgeving, waarin zich o.a. definities bevinden van de goniometrische functies.

Als we deze gedachte voortzetten, kunnen we elk PLAIN programma P opvatten als de definitie van een nieuwe omgeving. Het zou nu mogelijk moeten zijn om een programma P' te draaien in de omgeving gedefinieerd door P. De objecten die P' uit de omgeving gedefinieerd door P wil gebruiken, moeten door middel van een external declaratie in P' worden gespecificeerd.

Dit mechanisme lijkt veel op het gesloten scope mechanisme. Zouden we echter de semantiek van het scope mechanisme overnemen dan kunnen we niet voorkomen dat objecten van niveau 0 toegankelijk worden op het niveau van applicatie programma's. Een oplossing voor dit probleem is rechtstreeks onze eerste eis over te nemen. D.w.z. dat objecten die met behulp van external zijn gedeclareerd slechts op hogere niveau zichtbaar zijn, voorzover ze niet door een tussenliggend niveau worden gebruikt voor de definitie van nieuwe objecten (bijv. in modules).

Het construeren van de externe schema's is nu te vertalen in de constructie van een aantal programma's, waarin een gedeelte van de operaties op de databank worden gedefinieerd. Een bijkomend voordeel van deze aanpak is de mogelijkheid om op niveau 1 een vraagtaal interpretator te construeren voor de databank. Dit zou voor het geval van ADT's een doorbreking van het verbergen van de representatie betekenen.

4.4. Implicaties voor het PLAIN programmeersysteem.

De bovengeschetste methode heeft verstrekkende gevolgen voor de architectuur van het PLAIN programmeersysteem, de compiler en het runtime systeem. Belangrijkste consequentie is het gebruik van technieken voor partiële compilatie, welke door de relaties tussen de verschillende progamma's leidt tot een complex administratie systeem. Bij de uitvoering van de programma's dient rekening gehouden te worden met het bestaan van extern gedeclareerde data objecten. Deze externe objecten kunnen zowel statische objecten (constante en datatype definities), als ook dynamische objecten (variabelen en files) zijn. Het gebruik van dynamische objecten door meerder programma's tegelijk in een zelfde omgeving moet worden gereguleerd.

Aan de andere kant biedt deze aanpak een handvat voor de introductie van een protectie mechanisme op programmatuur niveau. Dit kan worden geillustreerd door het authorisatie mechanisme van System-R [6] af te beelden naar het PLAIN programmeersysteem. Het PLAIN systeem, nu gezien als beheerder van PLAIN bibliotheken, kent gebruikers en PLAIN programma's. Een gebruiker kan, na een adequate authenticatie, het systeem opdracht geven nieuwe programma's op te nemen in een bepaalde omgeving, progamma's te laten draaien in een bepaalde omgeving, maar ook rechten met betrekking tot deze operaties doorgeven aan andere gebruikers.

Een mogelijke PLAIN programmeersysteem instructie verzameling kan zijn:

```
ADD <program> TO <environment>
REMOVE <program> FROM <environment>
REPLACE <program> IN <environment>

ACTIVATE <program>
RUN <program>

GRANT { ADD      } ON <environment> TO <users> [WITH GRANT OPTION]
      { REMOVE   }
      { REPLACE  }
      { ACTIVATE }
      { RUN      }

REVOKE { ADD      } ON <environment> FROM <users>
       { REMOVE   }
       { REPLACE  }
       { ACTIVATE }
       { RUN      }
```

De RUN operatie op een programma heeft dezelfde uitwerking als het laten draaien van een programma in een conventionele omgeving. De ACTIVATE operatie verschilt in de beeindiging van het programma. Bij een RUN opdracht worden alle door het programma gebruikte objecten verwijderd, terwijl bij een ACTIVATE opdracht de globale variabelen blijven bestaan en er mechanismen worden opgezet om het toekomstig gelijktijdig gebruik te reguleren.

De GRANT instructie biedt de eigenaar van een programma de mogelijkheid de rechten op dat programma (ADD,ACTIVATE,etc) selectief naar gebruikers te distribueren en eventueel met behulp van de REVOKE operatie in te trekken.

5. MODELLEN VOOR TOEGANGSPROTECTIE

Het laatste brengt ons in een volgend onderzoeksgebied, namelijk de bescherming van de databank tegen niet toegestane operaties. De toegangsprotectie vormt slechts een klein onderdeel van de beveiliging van een databank, die bijvoorbeeld ook de fysieke beveiliging en beveiligings

condities op grond van tijd, geschiedenis, context en inhoud zal omvatten.

We concentreren ons hier echter op dit aspekt, omdat er een nauwe relatie bestaat met de beveiligings methoden in bedrijfssystemen en zodoende de nodige kennis en methoden reeds voorhanden zijn.

De toegangsprotectie wordt uitgevoerd met behulp van twee functies, de identificatie van een gebruiker (authenticatie) en de feitelijke beslissing over het al dan niet geven van toegang tot een beschermd object (authorisatie).

In de afgelopen jaren zijn er een aantal modellen ontwikkeld voor de beschrijving en analyse van bescherming in computer systemen [7,10]. Van deze modellen is het model geintroduceerd door Harrison-Ruzzo-Ullman het belangrijkste door zijn eenvoud en verstrekkende theoretische gevolgen. In dit model wordt uitgegaan van een toegangstabel (access matrix), waarin de rechten van de gebruikers (subjecten) ten opzichte van de objecten wordt geadministreerd. Op deze tabel zijn een aantal primitieve operaties gedefinieerd, zoals: voeg een nieuwe gebruiker toe, verwijder een object en voeg een recht toe aan een element van de tabel. Deze primitieve operaties kunnen worden gegroepeerd in de vorm van commando's. Een commando heeft een optionele start conditie, die de vereiste eigenschappen van de toegangstabel inspecteert voordat het commando wordt uitgevoerd.

Met deze concepten wordt aandacht geschonken aan het veiligheidsvraagstuk:

> Kan een commando ooit een recht aan een element van de toegangstabel toevoegen waar het oorspronkelijk nog niet aanwezig was.

Bij de beantwoording van deze vraag gaan ze ervan uit dat de te vertrouwen gebruikers uit de tabel verwijderd zijn. Alleen in deze situatie kan het toevoegen van een recht gezien worden als een inbreuk op de veiligheid.

Het resultaat van hun analyse is:

> Er bestaat geen algoritme dat voor een willekeurige toestand van een protectie systeem (toegangstabel, collectie commando's en rechten) beslist of het veilig is voor een gegeven recht.

Dit resultaat wordt verkregen door het model te associëren met een Turingmachine, waarbij het veiligheidsvraagstuk correspondeert met het stop-

probleem. Dit resultaat sluit niet uit dat er algoritme te ontwerpen is voor het beantwoorden van een veiligheidsvraagstuk voor alle situaties van een specifiek protectie systeem. Een van de protectie systemen waarvoor een specifiek veiligheidsvraagstuk te beantwoorden is, is het capability georienteerde Take-Grant model [10]. Het Take-Grant model verschilt van het Harrison-Ruzzo-Ullman model in de vorm van de primitieve operaties en het veiligheidsvraagstuk. Ook in het Take-Grant model wordt gewerkt met gebruikers, objecten en rechten. De operaties zijn herschrijvingsregels voor een graph, de protectie graph, die de relaties tussen de gebruikers en objecten representeert. Het veiligheidsvraagstuk in het Take-Grant model is of er een reeks van graph herschrijvingen bestaat, zodanig dat een gebruiker U een recht R tot een object O kan verkrijgen. De noodzakelijk en voldoende voorwaarde voor deze vraag is het bestaan van een (ongericht) pad in de protectie graph tussen U en O en het bestaan van het recht R op het object O.

De zwakte van het Take-Grant model is de eenvoud waaronder rechten verworven kunnen worden. Dit heeft geresulteerd in een aantal uitbreidingen op het basismodel [10].

De relatie tussen HRU en Take-Grant model is onderzocht door Weyuker [23]. Zij laat zien dat het Take-Grant model te simuleren is in het HRU model en omgekeerd. Als gevolg hiervan is het algemene veiligheidsvraagstuk ook voor het Take-Grant model onbeslisbaar, terwijl een specifiek Take-Grant veiligheidsvraagstuk te beantwoorden is in lineaire tijd naar het aantal elementen in de protectie graph.

6. HET SPE MODEL.

De in de vorige sectie besproken modellen zijn niet direct geschikt voor het definiëren van de activiteiten in een PLAIN programmeer omgeving. Met als gevolg dat er in het kader van het PLAIN project gezocht wordt naar een nieuw model voor de beschrijving en analyse van de relaties tussen PLAIN programma's, de relaties tussen objecten in PLAIN programma's en de operaties in een PLAIN omgeving.

De basisconcepten in het Secure Programming Environment (SPE) model zijn gebruikers (U), objecten (O) en gebieden (R). Gebruikers zijn eigenaar zijn van gebieden (OWN), waarin objecten zijn zijn gedefinieerd (DEF). Een gebied kan meerdere deelgebieden omvatten en zelf ook deel uitmaken van een

groter gebied, resulterend in een structuur relatie (STR). De communicatie tussen gebieden kan twee vormen aannemen, import en export. De import relatie (IMP) beschrijft het toegankelijk maken van objecten uit een omgeving van een gebied. De export relatie (EXP) beschrijft het in een omgeving toegankelijk maken van een in een gebied gedefinieerd object.

Deze relaties zijn een rechtstreekse afspiegeling van de concepten in de PLAIN omgeving. Gebieden corresponderen met de programma's en de scopes binnen een programma. Objecten zijn alle te definiëren PLAIN objecten, zoals procedures, datatypen, constanten etc. De import relatie beschrijft de external, het import en het parameter mechanisme. De structuur relatie beschrijft de relaties tussen verschillende scopes en programma's.

Naast deze relaties worden primitieve operatoren gedefinieerd, zoals: voeg een nieuw gebied X toe met eigenaar Y, X importeert het object O uit de omgeving Y. Groepering van de primitieve operaties in de vorm van commando's levert een samenvatting van de PLAIN programma's met betrekking tot de protectie.

Binnen dit model kan nu aandacht worden besteed aan de veiligheidsvraagstukken: "kan een object O in een gebied R gebruikt worden ?" en "zijn alle objecten nodig voor het draaien van een programma beschikbaar?".

7. SAMENVATTING

In dit artikel is in het kort geschetst hoe het relationele datamodel met de syntax van de programmeertaal verweven is, waarbij een aantal kanttekeningen zijn geplaatst bij de databank manipulaties. Vervolgens is geillustreerd hoe de taal de constructie van een relationeel databank systeem heeft beinvloed. Een open vraag is wat de syntactische consequenties van de definitie van databank transacties zijn.

In het tweede gedeelte is getoond hoe met behulp van de concepten in PLAIN een databank kan worden geconstrueerd. Hierbij is gebruik gemaakt van het nog ongedefinieerd zijn van de omgeving van een PLAIN programma. Tevens is getoond hoe protectie mechanismen met dit schema geintegreerd kunnen worden. Tenslotte is deze methode van protectie geplaatst in het kader van formele modellen voor toegangscontrole en is het Secure Programming Environment geintroduceerd.

DANKBETUIGINGEN

M. Kersten heeft dit onderzoek gedaan in dienst van de Nederlandse Organisatie voor Zuiver Wetenschappelijk Onderzoek (ZWO).
W. de Jonge en F. Vrielink worden hartelijk bedankt voor de constructieve opmerkingen ter verbetering van de leesbaarheid van dit stuk.

LITERATUUR

[1] AMBLE, T., K. BRATBERGSENGEN, P. KONGSHAUG, S.H. KVITSAND, R. LARSEN, O. RISNES, AND T. STALHANE, "Report on the Programming Language Euclid," ASTRA-notat 31 (may 1981).

[2] ASTRAHAN, M.M. AND OTHERS, "System-R: Relational Approach to Database Management," Transactions on Database Systems, Vol. 1(2), pp.97-137 (June 1976).

[3] BRATBERGSENGEN, K., O. RISNES, AND TORE AMBLE, "ASTRAL- A Structured Relational Applications Language," Technical Report 6/79, Division of Computing Sciences, University of Trondheim (June 1979).

[4] CODD, E.F., "A Relational Model of Data for Large Shared Data Banks," Comm.ACM, Vol. 13(6) (june 1970).

[5] COMMITTEE, CODASYL SYSTEMS, "Report of the Codasyl Data Description Language Committee," Information Systems, Vol. 3 (1978).

[6] GRIFFITHS, P.P. AND B.W. WADE, "An Authorization Mechanism for a Relational Data Base System," Transactions on Database Systems, Vol. 1(3), pp.242-255 (September 1976).

[7] HARRISON, M.A., W.L. RUZZO, AND J.D. ULLMAN, "Protection in Operating Systems," Comm. ACM, Vol. 19(8), pp.461-470.

[8] (ED), J.D. ICHBIAH, "Preliminary Ada Reference Manual," SIGPLAN Notices, Vol. 14(6 part A) (June 1979).

[9] JENSEN, K. AND N. WIRTH, "Pascal- User Manual and Report," in Lecture Notes in Computer Science 18, Springer-Verlag (1974).

[10] JONES, A.K., "Protection Mechanism Models Their Usefulness," pp. 237-253 in Foundations of Secure Computation, ed. Lipton, Academic Press (1978).

[11] KERSTEN, M.L., A.I. WASSERMAN, AND R.P. VD RIET, The Design of the PLAIN Data Base Handler, To appear as VU Informatica report in 1981.

[12] KERSTEN, M.L. AND A.I. WASSERMAN, "The Architecture of the PLAIN Data Base Handler," Software, Practice & Experience (Feb. 1981).

[13] LISKOV, B. AND OTHERS, "CLU Reference Manual," Technical Report TR-225, MIT Lab. for Computer Science (1980).

[14] RIET, R.P. VAN DE, A.I. WASSERMAN, M.L. KERSTEN, AND W. DE JONGE, "High Level Programming Features for Improving the Efficiency of a Relational Data Base System," Transactions on Database Systems. In press.

[15] RIET, R.P. VAN DE, "Databanken, enkele onderzoeksaspecten," in Colloquium Databankorganisatie, Math.Centrum, Amsterdam (1980).

[16] RIET, R.P. VAN DE, M.L. KERSTEN, AND A.I. WASSERMAN, "A Module Definition Facility for Access Control in Distributed Data Base Systems," IEEE Computer Society Security and Privacy Conference (april 1980).

[17] ROWE, L. AND K. SHOENS, "Data Abstraction, Views and Updates in Rigel," SIGMOD 1979, Dep of Comp Sci U.C. Berkeley.

[18] STONEBRAKER, M.R. AND OTHERS, "The Design and Implementation of INGRES," ACM Tods (Sep. 1977).

[19] WASSERMAN, A.I., D.D. SHERETZ, M.L. KERSTEN, R.P. VAN DE RIET, AND M. DIPPE, "Revised Report on the Programming Language PLAIN," Technical Report #42, Laboratory of Medical Information Science, University of California, San Francisco.

[20] WASSERMAN, A.I., SHERERTZ, D.D., AND HANDA, E.F., "Report on the Programming Language PLAIN," Informatica Rapport IR 37 (june 1978).

[21] WASSERMAN, A.I., "USE: a Methodology for the Design and Development of Interactive Information Systems," pp. 31-50 in Formal models and practical tools in Inf. Sys. Design, ed. Scheider,H.J., North Holland, Amsterdam (1979).

[22] WASSERMAN, A.I., R.P. VAN DE RIET, AND M.L. KERSTEN, "PLAIN: An Algoritmic Language for Interactive Information Systems," IFIP TC2 International Symposium on Algorithmic Languages (October 1981).

[23] WEYUKER, E.J., "Security in Operating Systems: separating the roles of rigths," Technical Report 003, New York University (D).

ONTWERPASPECTEN VAN DISTRIBUTED DATABASE MANAGEMENT SYSTEMS

R.A.C. THOMAS
Vrije Universiteit, Amsterdam

1. INLEIDING

De ontwikkeling van de gedistribueerde database management systemen (DDBMS) staat nog in de kinderschoenen. Door de komst van de computer-netwerken ontstond allereerst de mogelijkheid gegevensverzamelingen elders te raadplegen en al gauw ontstond de wens de inhoud van verschillende gegevensverzamelingen te combineren. Daarnaast werd gewerkt aan de ontwikkeling van nieuwe systemen, zonder de beperkingen van reeds bestaande lokale systemen. Beide stromingen leverden een serie benamingen voor DDBMS's op. In sectie 2 worden de meest voorkomende termen toegelicht.

Het ontwerpen van DDBMS staat centraal in sectie 3, waar naast de specifieke distributie-aspecten ook het toekomstig onderhoud wordt belicht.

In sectie 4 wordt ingegaan op de voordelen van standaardisatie van de subsystemen van DDBMS's, zowel voor de ontwerpfase als voor het onderhoud op langere termijn.

Tenslotte wordt, aan de hand van het D-INGRES systeem, een flexibel mechanisme voor uitbreiding van de gedistribueerde database (DDB) en aanpassing van de distributie getoond.

2. CLASSIFICATIE

In de literatuur worden voor DDBMS's verschillende benamingen gebruikt. De naam slaat meestal terug op één of meerdere specifieke eigenschappen van het DDBMS. Soms heeft de naam betrekking op de verschillen tussen, dan wel de overeenkomsten van, de lokale DBMS's, die tezamen het DDBMS vormen.

In andere gevallen wordt het DDBMS getypeerd op grond van een facet, zoals bijvoorbeeld het gekozen datamodel of de aard van de distributie. De bekendste aanduidingen voor DDBMS zijn:

- heterogeneous/homogeneous
- general purpose/special purpose
- uniform/diversified
- centralized/decentralized
- partitioned
- replicated.

2.1. Heterogeneous en homogeneous DDBMS's

Een DDBMS is homogeen, wanneer de lokale DBMS's gelijk zijn. Soms wordt een verder onderscheid gemaakt en wordt er gesproken van een "fully homogeneous" DDBMS, indien ook de hardware voor lokaties identiek is. DRAFFAN [1] Een heterogene DDMBS is opgebouwd met behulp van verschillende DBMS's. Het bestaan van beide termen benadrukt eens te meer het ontstaan van de DDBMS's uit bestaande systemen. Vanuit die invalshoek is het ook logisch dat ook de hardware in de naamgeving werd betrokken. Immers, het gebruik van verschillende machines kan tal van extra problemen opleveren bij de communicatie tussen de lokale DBMS's, zodat het resulterende heterogene DDBMS aanzienlijk zal verschillen van de homogene oplossing.

Of deze naamgeving in de toekomst zijn waarde zal behouden valt te bezien. Door standaardisering van de communicatie-netwerken en het doorvoeren van een functionele scheiding tussen netwerk-, database- en bedrijfssysteem-aspecten, zullen de aspecten op het juiste niveau worden afgehandeld, en dat is onder het DBMS.

2.2. General purpose/special purpose DDBMS's

Dit onderscheid wordt gemaakt in verband met het uiteindelijke gebruik van de DDBMS. Een onderscheid dat overigens evenzeer voor DBMS's geldt. Bij een special purpose DDBMS ligt het gebruik op voorhand vast. Het systeem zal veelal ongeschikt zijn voor andere toepassingen dan welke aan het ontstaan ten grondslag hebben gelegen. Het klassieke voorbeeld van een special purpose systeem is een vliegtickets reserveringssysteem.

We spreken van een general purpose DDBMS indien de toepassing niet van tevoren vast ligt, en het systeem voor de oplossing van verschillende problemen kan worden aangewend.

2.3. Uniform/diversified DDBMS's

De uniformiteit slaat op het datamodel. Zo zal een "special purpose" DDBMS veelal uniform zijn, omdat het datamodel vast ligt, zodat de lokale DBMS's, gebruik makend van die kennis, geoptimaliseerd kunnen worden. Er is sprake van een diversified DDBMS als niet alle LDB's met hetzelfde datamodel zijn gebouwd; bijvoorbeeld: de ene lokatie bevat een relationele database en een andere een hiërarchische.

2.4. Centralized/decentralized DDBMS's

Een centralized DDBMS wordt gekenmerkt door het bestaan van één besturend proces, een kernel of nucleus. In een decentralized DDBMS is ieder lokaal DBMS (LDBMS) autonoom. De controle- c.q. besturingsfunctie, die al dan niet centraal geregeld is, kan samenhangen met "concurrency control", "distributed query decomposition" en administratie van data-allocatie en toegangsbeveiliging. Een voorbeeld van een decentralized DDBMS is SDD-1. ROTHNIE [5], WONG [19]

2.5. Partitioned DDBMS's

De term partitioned geeft de aard van de datadistributie weer. Over het algemeen wordt er onderscheid gemaakt tussen "partitioned by structure" en "partitioned by occurence". Partitioning by structure duidt op het verdelen van de data per set of relatie over de lokaties: alle records die tot een en dezelfde set behoren, ofwel alle tuples van één relatie worden op een bepaalde computer in het netwerk samengebracht. Bij partitioning by occurence kunnen ook sets en/of relaties over meerdere lokale databases worden verdeeld.

2.6. Replicated DDBMS's

Replicated duidt op het dupliceren van data op verscheidene lokaties. Eén van de voordelen van het werken met netwerken is de verhoogde betrouwbaarheid ten aanzien van beschikbare computerfaciliteiten. Het uitvallen van een machine kan worden opgevangen door de overige computers in het

netwerk. Ook bij database toepassingen kan hiervan gebruik worden gemaakt. In principe komt dit neer op het bijhouden van een "back-up" van iedere LDB. Verschillende implementaties zijn mogelijk. De back-up van een LDB bevindt zich in z'n geheel op een tweede lokatie, als een schaduw-database. Gaat de eerste lokatie "down", dan kan alle verlangde informatie met behulp van de back-up worden verstrekt. Daar de distributie voor gebruikers van DDBMS's verborgen dient te blijven (de gebruiker werkt volgens zijn waarnemingen met een centrale DB), zal het overgaan op de back-up automatisch en buiten de waarneming van de gebruiker moeten geschieden.

Een tweede toepassing wordt gevonden in het op één of meer lokaties bijhouden van een copie van een deel van de database. Daarmee wordt bereikt dat een groter deel van de gebruikers direkt over de gegevens kan beschikken, zonder dat daar netwerkverkeer aan te pas komt. Het "up-to-date" houden van de copieën kost daarentegen het rondsturen van de veranderingen. Aan de hand van een analyse van het verwachte gebruik van de back-up's en de verwachte frequentie waarmee het originele bestand zal wijzigen, kan worden bepaald of een back-up moet worden onderhouden en zo ja, op hoeveel lokaties dat dient te gebeuren.

3. ONTWERPASPECTEN

Bij het ontwerpen van een general purpose DDBMS, wordt uitgegaan van een modelmatige beschrijving van de toepassingsgebieden. In hoge mate kan daarbij worden voortgebouwd op DBMS-research, en zullen met name distributie-afhankelijke grootheden nader moeten worden onderzocht. Het ontwikkelen van special purpose DDBMS's zal nodig zijn wanneer de toepassing buiten het gekozen model van een general purpose DDBMS valt.

In het bedrijfsleven worden (D)DBMS's gezien als onderdeel van informatiesystemen. De meeste ontwikkelingsmethoden voor informatiesystemen onderkennen opeenvolgende en van elkaar afhankelijke fasen. Uitgaande van een functie- en informatie-analyse, ontstaat een beschrijving van de totale behoefte aan informatie binnen een bedrijf of organisatie. Het doel van de daaropvolgende fasen is het, middels korte en bestuurbare projecten, komen tot de formele beschrijving van gegevens en gegevensstructuren en de processen die met deze gegevens zullen werken.

De totale analyse kan in vier etappes worden verdeeld: informatie-analyse, data-analyse, implementatie-analyse en fysieke analyse. VANDENBULCKE 16 Nadat is bepaald welk deel van het formele informatiesysteem geautomatiseerd zal worden, kunnen de desbetreffende processen en gegevensstructuren met behulp van computers worden gerealiseerd. Met de aldus verkregen specificaties is het mogelijk te onderzoeken welk bestaand (D)DBMS bij de implementatie van het informatiesysteem kan worden toegepast. Mocht geen van de (D)DBMS's geschikt blijken, dan zal een passend systeem gemaakt moeten worden. Het hoofddoel blijft het beantwoorden aan de gebruikersspecificaties en wel zò, dat ook na acceptatie flexibel op nieuwe behoeften kan worden ingespeeld.

Bij de ontwikkeling van DBMS's kunnen diverse probleemgebieden worden aangegeven:

- de gebruikers-interface: vraagtaal/applicatietaal
- datamodellen
- functionele afhankelijkheden
- views
- opslagstructuren
- toegangspaden
- query processing
- consistency constraints
- concurrency control
- recovery mechanisme
- beveiliging.

Op alle gebieden is uitvoerig onderzoek verricht. De gevonden oplossingen zijn veelal een direct gevolg van de modellen die aan het onderzoek ten grondslag hebben gelegen. Het is derhalve onmogelijk uit alle onderzoekrapporten betreffende DBMS's, het bij uitstek geschikte DBMS te construeren.

Dat het ontwikkelen van DDBMS's nog veel meer voeten in de aarde heeft, zal niemand verbazen. Naast alle problemen van DBMS's speelt nu de distributiefactor mee. Dit specifieke aspect doet zich voornamelijk gelden bij:

- data-allocatie
- distributed queries
- concurrency control
- consistency
- beveiliging.

Data-allocatie kwam bij de classificatie al ter sprake. De plaatsing van gegevens over het netwerk wordt mede bepaald door de aard van het gebruik en de transportkosten, die bij niet-lokaal gebruik in rekening worden gebracht.

Distributed queries zijn vragen die slechts beantwoord kunnen worden door samenwerking van twee of meer LDBMS's. Problemen die zich hier voordoen zijn o.a.:

- Wordt de beantwoording van de vraag gestuurd door één LDBMS of werken de LDBMS's als autonome eenheden?
- Ligt de beantwoording van de vraag vast, nadat op de query computer, de computer waar de vraag gesteld werd, de aanwezige beschrijving van de data is geraadpleegd (statische decompositie), of vindt er tijdens de decompositie een herberekening plaats op grond van de gegenereerde tussenresultaten (dynamische decompositie)?

Concurrency control voorziet in het garanderen van een consistente DDB, d.w.z. voor iedere gebruiker moeten de verschillende LDB's tijdens het beantwoorden van een distributed query ook ten opzichte van elkaar consistent zijn. Dit is een extra eis bovenop de consistentie van de afzonderlijke LDB's.

Voor de complexiteit van DDBMS's bestaat nog een tweede reden. In feite komen drie probleemgebieden samen:

1. DBMS's
2. Bedrijfssystemen
3. Communicatie-netwerken

Zeker in die gevallen, waarin DDBMS's worden gebouwd met behulp van reeds bestaande LDBMS's en LDB's is het moeilijk deze probleemgebieden onafhankelijk van elkaar te benaderen.

Zoals bij de ontwikkeling van ieder omvangrijk systeem, moet al tijdens het ontwerp rekening worden gehouden met het toekomstige onderhoud.

Zo moeten voorzieningen zijn getroffen voor:

- Back-up/crash recovery. Normaal wordt van iedere LDB geregeld een back-up gemaakt. De frequentie hangt af van de snelheid waarmee gegevens in de LDB verouderen en in mindere mate van de grootte van de LDB en daarmee de tijd die het duurt om een back-up te maken. Het crash recovery mechanisme zorgt voor het herstellen van de consistency in de DDB, nadat één van de LDBMS's is "gecrashed". Mocht de DDB onherstelbaar beschadigd zijn, dan kan door het laden van de lokale back-up's een vroegere consistente staat worden verkregen.

- Nieuwe hardware. De nieuwe hardware kan een nieuwe computer zijn, maar ook speciale apparatuur, zoals bijvoorbeeld associatieve geheugens, die hele functies van het software-pakket kunnen overnemen. Indien een doordachte, functionele scheiding heeft plaatsgevonden, bestaat de kans dat dergelijke aanpassingen kunnen plaatsvinden, zonder het herschrijven van grote delen van het systeem.

- Aanpassingen van de DDB. De totale omvang van de DDB kan variëren in de tijd, bijvoorbeeld door verandering van het aantal lokaties. Derhalve moet het mogelijk zijn de data te realloceren. Over het algemeen worden er speciaal voor de database administrator (DBA) instructies geschapen op het niveau van de user-interface, waarmee de data-allocatie gespecificeerd kan worden.

- Lokale veranderingen. Daarbij valt te denken aan bijvoorbeeld het kiezen van andere opslagstructuren, dan wel het veranderen van de implementatie van een bestaande opslagstructuur. Daar de veranderingen lokaal gelden, hebben we te maken met de normale aanpassingen, die we al van de DBMS's kennen.

4. STANDAARDISATIE

Inmiddels is de kennis over en de ervaring met DBMS's dusdanig toegenomen, dat internationaal naar standaards wordt gezocht. Zo kennen we nu de conceptuele beschrijving van DBMS's volgens de Codasyl Data Base Task Group. Daarbij wordt uitgegaan van een splitsing van het ontwerp in externe, conceptuele en interne schema's. In het conceptuele schema wordt de conceptuele databeschrijving vastgelegd, alsmede de regels waaraan de data zal blijven voldoen (consistency & integrity constraints).

In het externe schema kan de view van een gebruikersgroep op de data of een deel daarvan worden gedefinieerd. Zo kunnen aparte gebruikersgroepen over verschillende externe schema's beschikken, waarmee redelijk flexibel aan gebruikerswensen tegemoet kan worden gekomen, zonder dat andere delen van de DB worden beïnvloed. Evenzo wordt de fysische implementatie afgescheiden van het conceptuele schema door middel van het interne schema. De implementatie van een nieuwe opslagstructuur wordt geheel binnen het interne schema afgedaan en laat de rest van het systeem ongemoeid.

Het model is vrij simpel tot het gedistribueerde geval uit te breiden (zie figuur 1). Duidelijk is te zien, dat netwerk-gebruikers de DDB door het globale conceptuele schema als een simpele DB zien, waardoor de distributie onzichtbaar wordt.

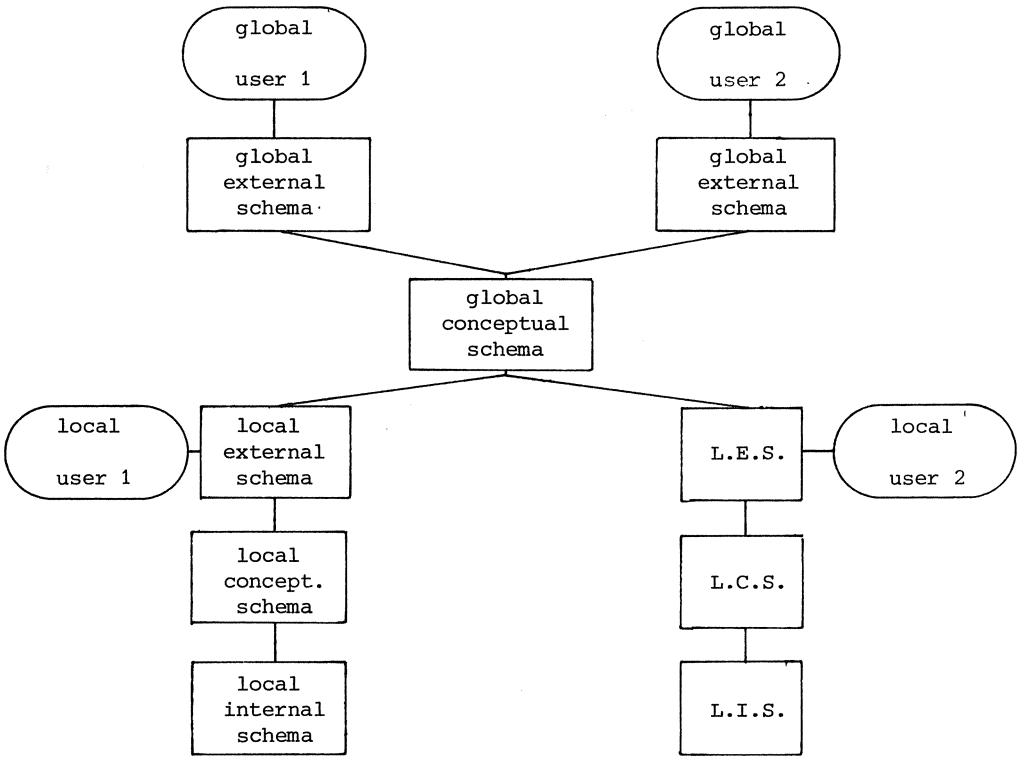


fig. 1: Schematische voorstelling van een DDBMS.

Het nut van deze ontwikkeling is de mogelijkheid die ermee geschapen wordt, het DDBMS in de toekomst te onderhouden, waarbij ook ingrijpende veranderingen realiseerbaar zullen zijn.

Een dergelijke ontwikkeling doet zich ook voor op het gebied van de computernetwerken. Door het ISO (International Standards Organization) werd het OSI-model gelanceerd. Het OSI-model, dat staat voor het model voor "Open Systems Interconnection", is een model bestaande uit zeven "layers" (lagen), die tezamen de verbinding tussen computers weergeven (zie figuur 2). Tussen dezelfde layers van verschillende machines moeten nu protocollen worden afgesproken, zodat op den duur een verzameling protocollen als standaard kan worden aangewezen. Iedere layer zal een zeer specifieke functie uitvoeren, en iedere layer communiceert uitsluitend met de layers direct onder en boven zich, met uitzondering van de "physical layer", die zorgdraagt voor het oversturen van de bits naar de ontvangende machine(s).

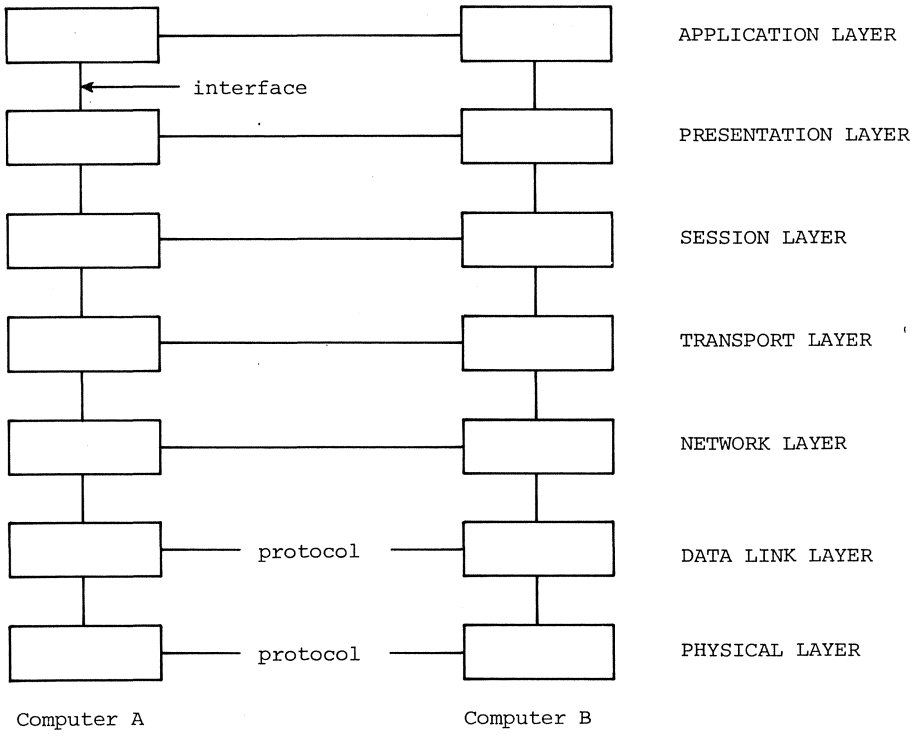


fig. 2: Het OSI-model.

De opsplitsing in zeven layers moet gezien worden als eerste stap in het modulair en gestructureerd ontwerpen van computerverbindingen. De layers op een machine kunnen via interfaces communiceren. Zolang aan de input/output specificatie is voldaan, kan iedere interface naar keuze geïmplementeerd worden. In dit kader zouden LDBMS's door middel van het protocol tussen de "application layers" kunnen converseren. Het doel is te komen tot een internationale standaard.

Eén van de belangrijkste voordelen is de scheiding die zo kan worden aangebracht tussen communicatie-software en database-software. Indien die scheiding stringend wordt doorgevoerd, wordt a) het DDBMS-probleem beperkter en b) de DDBMS-oplossing onafhankelijk van de gekozen implementatie van de computercommunicatie.

5. REALISATIE VAN DE DISTRIBUTIE

Aan de hand van een ontwerp van een gedistribueerd database management systeem, dat aan de VU werd ontwikkeld, wordt aangetoond hoe de distributie van data over de verschillende lokaties bereikt kan worden. Twee uitgangspunten stonden centraal:

1. Het DB-model moet simpel te wijzigen zijn.
2. De distributie van data moet selectief kunnen gebeuren.

5.1. D-Ingres

Het ontwerp van Distributed Ingres (D-Ingres) is gebaseerd op het DBMS-Ingres. Ingres staat voor INteractive Graphics REtrieval System en werd ontworpen aan de Berkeley University in Californië. STONEBRAKER [8] Het Ingres-systeem draait onder het bedrijfssysteem Unix. RITCHIE [3]

Ingres is een relationeel systeem, zodat het datamodel ook voor het DDBMS relationeel zal zijn. Geen wezenlijke beperking, daar de meeste gegevensverzamelingen op een relationeel model zijn af te beelden.

Ingres draait als een verzameling processen op het Unix-bedrijfssysteem en is feitelijk een single user systeem. Gebruikers kunnen wel gelijktijdig aan dezelfde database werken, omdat iedere gebruiker een privé-copie van Ingres krijgt toegewezen.

Daar voor een effectieve concurrency control alle aktiviteiten rond de DB door een centraal proces bestuurd dienen te worden, zijn slechts twee implementaties mogelijk.

De concurrency control zou geregeld kunnen worden door een proces, het CCP, dat voor de eerste gebruiker wordt opgestart en waaraan alle volgende Ingres-systemen worden gekoppeld. De communicatie tussen het CCP en de verschillende Ingres-systemen verloopt via pipes, het interproces-communicatiemiddel onder Unix. Pipes zijn speciale files met een maximale grootte van 4096 bytes, die door het ene proces gelezen en door het andere proces beschreven kunnen worden. In verband met het creëren van deze pipe-verbinding, is het noodzakelijk de verschillende Ingres-systemen door de CCP te laten genereren. Het nadeel van deze implementatie is de hoge frequentie waarmee het CCP lock requests moet verwerken, zodat de vertraging ten gevolge van het gebruik maken van pipes zich extra zwaar doet voelen.

De tweede implementatie, die ook door de ontwerpers van Ingres is gekozen, behelst het toevoegen van een zogenaamd "lock device" aan het Unix-bedrijfssysteem. Ieder lock request resulteert nu in een "system call", hetgeen de verwerkingssnelheid aanzienlijk ten goede komt. Beide implementaties geven in het gedistribueerde geval een hoop extra problemen.

Omdat Ingres is gebaseerd op het Unix-systeem, zullen alle machines waar D-Ingres op zal draaien minstens virtuele Unix-machines moeten zijn. In de naaste toekomst is deze voorwaarde misschien nauwelijks een beperking te noemen, daar Unix op steeds meer machines beschikbaar komt.

De eenheid van distributie is de relatie. Dat betekent, dat alle tuples van een relatie bij elkaar in één LDB worden opgeslagen; met andere woorden: partitioning by structure.

Voor D-Ingres werd het oorspronkelijke systeem uitgebreid met een extra proces, de "Distributor", dat zorgdraagt voor de communicatie met Ingres-systemen op andere lokaties.

Roept een Unix-gebruiker het D-Ingres-systeem aan, dan wordt op de query computer een lokaal D-Ingres opgestart, d.w.z. het Distributor-proces wordt niet opgestart. Zolang de gebruiker vragen stelt betreffende de lokaal aanwezige relaties, blijft hij/zij met een lokaal systeem werken.

Wordt echter een distributed query gesteld, dan wordt het Distributor-proces alsnog geactiveerd. Op alle lokaties, die opgevraagde relaties bevatten, zal vervolgens een extra D-Ingres worden gestart.

Het oorspronkelijke D-Ingres noemen we Master-Ingres (MI) en de bijbehorende D-Ingres-systemen op de overige lokaties noemen we Slave-Ingres (SI). De termen Master en Slave geven slechts aan op welke lokatie de eindgebruiker zich bevindt en duiden niet op een centraal geregelde besturing van alle slaven. Integendeel, tijdens de query decompositie zullen de MI en alle SI's autonoom hun bijdrage leveren. D-Ingres kan dus bestaan uit enkel een MI, indien er sprake is van lokaal gebruik van de DDB, of uit één MI en een verzameling van SI's, indien distributed queries worden gesteld.

In figuur 3 is een netwerk, bestaande uit de machines M1, M2 en M3, geschetst. Op M1 is D-Ingres opgestart voor een gebruiker, die wil werken met database DB1. Na het stellen van een distributed query met betrekking tot relaties, die zich op M2 en M3 bevinden, is op beide machines een Slave-Ingres opgestart.

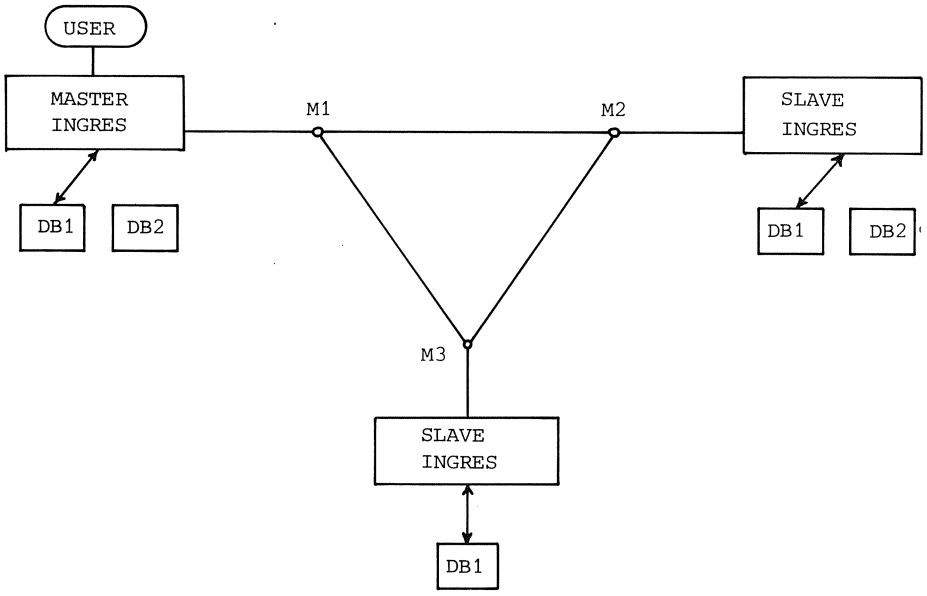


fig. 3: Master-Ingres & Slave-Ingres systemen.

5.2. Administratie

Uitgaande van dit ontwerp werd een uitsplitsing gemaakt van de benodigde gegevens, die tezamen de kennis omtrent de distributie weergeven. Voor D-Ingres onderscheiden wij de volgende administratieve gegevens:

- database identificatie (DI)
- relatie identificatie (RI)
- relatie beschrijving (RB)
- performance informatie (PI)
- concurrency & privacy informatie (CI).

Alleen de CI wordt niet gedistribueerd, zodat ieder lokaal DBMS verantwoordelijk blijft voor het garanderen van de beveiliging van de lokale DB.

De RB bestaat ook in het niet-gedistribueerde systeem. Per relatie worden de volgende gegevens opgeslagen:

- naam
- eigenaar
- aantal tuples
- lengte van een tuple
- opslagstructuur
- aantal attributen
- naam, type en lengte per attribuut.

Meer nog dan in het lokale geval is het van belang over voldoende PI te beschikken, nl. in verband met de optimalisatie van netwerkverkeer tijdens distributed query processing. De aard van de PI hangt derhalve sterk af van de te gebruiken distributed query decompositie algoritme. Daar uiteindelijk alle SI's autonoom een distributed query zullen afhandelen, zal een copie van de PI op iedere lokatie aanwezig zijn. Willen we de PI op iedere lokatie up-to-date houden, dan kan een aanzienlijke netwerkoverhead het gevolg zijn. Immers, lokale veranderingen aan relaties moeten naar de overige lokaties worden doorgegeven. De overhead kunnen wij echter beperken als een geringe afwijking wordt toegelaten. De update kan dan plaatsvinden nadat een nieuwe verandering de discrepantie tussen oude en nieuwe PI over een vooraf vastgestelde grens tilt.

De DI en RI gegevens bevatten de informatie betreffende de distributie. De inhoud en werking van beiden wordt nu toegelicht.

5.3. Database identificatie en relatie identificatie

De DI is uniek voor elke lokale database. Laat gegeven zijn een netwerk bestaande uit drie computers: M1, M2 en M3. Zij verder DDB-A, een DDB bestaande uit LDB-1 en LDB-2 op respectievelijk M1 en M2. Op M1 bevat de DI van LDB-1 een beschrijving van LDB-2 en de DI van LDB-2 op M2 bevat een beschrijving van LDB-1. Het formaat van de DI is schematisch weergegeven in figuur 4.

DB-NAME	LOCATION	LOCAL NAME	RIGHTS
LDB-2	M2	DDB-A	NONE

fig. 4a: Database identificatie van LDB-1.

DB-NAME	LOCATION	LOCAL NAME	RIGHTS
LDB-1	M1	DDB-A	STORE

fig. 4b: Database identificatie van LDB-2.

Daar de DI per LDB uniek is, is het mogelijk voor iedere lokale database een lokaal idee van de totale distributie te geven. Daartoe creëren wij allereerst een nieuwe database DB-3 op M3. Daar DB-3 niet tot een DDB behoort, is de DI leeg. Het toevoegen van DB-3 aan DDB-A geschiedt door het uitwisselen van DI-informatie tussen de verschillende lokale databases. Laten wij aannemen dat LDB-1 en DB-3 DI-informatie naar elkaar versturen. De totale DI-verzameling wordt dan gegeven door figuur 5.

	DB-NAME	LOCATION	LOCAL NAME	RIGHTS
LDB-1	LDB-2	M2	DDB-A	NONE
LDB-1	LDB-3	M3	DB-3	NONE
LDB-2	LDB-1	M1	DDB-A	STORE
DB-3	LDB-1	M1	DDB-A	STORE

fig. 5: De DI na connectie van LDB-1 en DB-3.

Vanuit M1 bezien bestaat de DDB-A uit LDB's op M1, M2 en M3. Vanuit M2 en M3 bezien maken slechts twee lokale DB's deel uit van de DDB.

Merk op dat:

1. de DDB op verschillende machines onder verschillende namen bekend is, nl. DDB-A op M1 en M2 en DB-3 op M3;
2. de benaming van lokale databases in de DI onder het hoofd DB-NAME uniek is, maar binnen iedere LDB vrij te kiezen.

Met andere woorden, als zich bij het verbinden van twee LDB's een naamconflict zou voordoen in de bestaande DI's, dan moeten de betrokken DBA's door het kiezen van een geschikte DB-NAME het conflict opheffen. Indien de benaming niet aan bepaalde schoonheidseisen hoeft te voldoen, kan de naamgeving ook geautomatiseerd worden.

De RI-informatie werkt zoals de DI-informatie, met dit verschil, dat bij het versturen van RI-informatie juist het bestaan van een lokale relatie aan andere LDB's wordt gemeld. Een LDB kan enkel RI-informatie versturen van lokaal opgeslagen relaties. Ontvangen RI-informatie kan dus nooit aan een derde LDB worden doorgegeven. Verder kan slechts RI-informatie worden verstuurd naar LDB's die in de DI voorkomen.

In figuur 5 was tevens sprake van een veld RIGHTS. In RIGHTS staat gespecificeerd welke rechten de verbonden LDB heeft. STORE duidt op het recht van de andere LDB, RI-informatie te zenden. Uit figuur 5 blijkt derhalve dat de lokale DB's op M2 en M3 wel relaties naar M1 mogen distribueren en dat RI-informatie niet uit M1 kan worden verzonden.

Samenvattend, iedere DDB kan uiterst flexibel worden uitgebreid met nieuwe LDB's. Zelfs bestaat de mogelijkheid op deze wijze twee bestaande DDB's te koppelen. Het distribueren van relaties gebeurt zeer selectief,

waarna tevens een extra niveau van beveiliging wordt geïntroduceerd.

LITERATUUR

[1] DRAFFAN, I., Advanced Course on Distributed Data Bases, Sheffield, England, 1979.

[2] HEVNER, A.R., S.B. Yao, Query Processing in Distributed Database Systems, Proc. of the third Berkeley Workshop on Distributed Data Management and Computer Networks, Berkeley, Calif., 1978.

[3] RITCHIE, D.M., K. Thompson, The Unix Time-Sharing System, The Bell System Technical Journal, Vol. 57, No. 6, Part 2, pp. 1905-1929.

[4] ROSENKRANTZ, D.J. et al, A System Level Concurrency Control for Distributed Database Systems, Proc. of the second Berkeley Workshop on Distributed Data Management and Computer Networks, Berkeley, Calif. 1977.

[5] ROTHNIE, J.B., N. Goodman, An Overview of the Preliminary Design of SDD-1: A System for Distributed Data Bases, Proc. of the second Berkeley Workshop on Distributed Data Management and Computer Networks, Berkeley, Calif. 1977.

[6] STONEBRAKER, M., E. Wong, Access Control in a Relational Data Base Management System by Query Modification, Proc. ACM Nat. Conf. 1974, San Diego, Calif. pp. 180-187.

[7] STONEBRAKER, M., Implementation of Integrity Constraints and Views by Query Modification, Proc. 1975 ACM-SIGMOD Conf. on Management of Data, San Jose, Calif., June 1975.

[8] STONEBRAKER, M., E. Wong, P. Kreps & G. Held, The Design and Implementation of Ingres, ACM Transactions on Database Systems, Vol. 1, No. 3, 1976, pp. 189-222.

[9] STONEBRAKER, M., P. Rubenstein, The Ingres Protection System, Proc. 1976 ACM Nat. Conf., Houston, Texas.

[10] STONEBRAKER, M., E. Neuhold, A Distributed Database Version of Ingres, Proc. of the second Berkeley Workshop on Distributed Data Management and Computer Networks, Berkeley, Calif., 1977.

[11] STONEBRAKER, M., Concurrency Control and Consistency of Multiple Copies of Data in Distributed Ingres, Proc. of the third Berkeley Workshop on Distributed Data Management and Computer Networks, Berkeley, Calif., 1978, pp. 235-258.

[12] SUNDGREN, B., Data Base Design in Theory and Practice, Proc. fourth International Conf. on Very Large Data Bases, West-Berlin, Germany, 1978.

[13] THOMAS, R.A.C., A Commentary on Synchronization of the Data Base System Ingres, Dept. of Comp. Sc., Free University, Amsterdam, Report IR-26, 1977.

[14] THOMAS, R.A.C., A Commentary on Storage Structures and Access Method of the Data Base System Ingres-5.1., Dept. of Comp. Sc., Free University, Amsterdam, Report IR-32, 1978.

[15] THOMAS, R.A.C., Process Structure Alternatives towards a Distributed Ingres, Proc. International Symp. on Distributed Data Bases, Paris, March 12-14, 1980, pp. 215-227.

[16] VANDENBULCKE, J.A., Data Base Ontwerp, Colloquium Databankorganisatie Deel 1, Mathematisch Centrum, Amsterdam, 1981.

[17] WIEDERHOLD, G., Database Design, McGraw Hill, 1977.

[18] WONG, E., K. Youssefi, Decomposition - A Strategy for Query Processing, ACM Transactions on Database Systems, Vol. 1, No. 3, 1976, pp. 223-241.

[19] WONG, E., Retrieving Dispersed Data from SDD-1: A System for Distributed Data Bases, Proc. of the second Berkeley Workshop on Distributed Data Management and Computer Networks, Berkeley, Calif., 1977.

UITGAVEN IN DE SERIE MC SYLLABUS

Onderstaande uitgaven zijn verkrijgbaar bij het Mathematisch Centrum, 2e Boerhaavestraat 49 te Amsterdam-1005, tel. 020-947272.

MCS 1.1 F. GÖBEL & J. VAN DE LUNE, *Leergang Besliskunde, deel 1: Wiskundige basiskennis*, 1965. ISBN 90 6196 014 2.

MCS 1.2 J. HEMELRIJK & J. KRIENS, *Leergang Besliskunde, deel 2: Kansberekening*, 1965. ISBN 90 6196 015 0.

MCS 1.3 J. HEMELRIJK & J. KRIENS, *Leergang Besliskunde, deel 3: Statistiek*, 1966. ISBN 90 6196 016 9.

MCS 1.4 G. DE LEVE & W. MOLENAAR, *Leergang Besliskunde, deel 4: Markovketens en wachttijden*, 1966. ISBN 90 6196 017 7.

MCS 1.5 J. KRIENS & G. DE LEVE, *Leergang Besliskunde, deel 5: Inleiding tot de mathematische besliskunde*, 1966. ISBN 90 6196 018 5.

MCS 1.6a B. DORHOUT & J. KRIENS, *Leergang Besliskunde, deel 6a: Wiskundige programmering 1*, 1968. ISBN 90 6196 032 0.

MCS 1.6b B. DORHOUT, J. KRIENS & J.TH. VAN LIESHOUT, *Leergang Besliskunde, deel 6b: Wiskundige programmering 2*, 1977. ISBN 90 6196 150 5.

MCS 1.7a G. DE LEVE, *Leergang Besliskunde, deel 7a: Dynamische programmering 1*, 1968. ISBN 90 6196 033 9.

MCS 1.7b G. DE LEVE & H.C. TIJMS, *Leergang Besliskunde, deel 7b: Dynamische programmering 2*, 1970. ISBN 90 6196 055 X.

MCS 1.7c G. DE LEVE & H.C. TIJMS, *Leergang Besliskunde, deel 7c: Dynamische programmering 3*, 1971. ISBN 90 6196 066 5.

MCS 1.8 J. KRIENS, F. GÖBEL & W. MOLENAAR, *Leergang Besliskunde, deel 8: Minimaxmethode, netwerkplanning, simulatie*, 1968. ISBN 90 6196 034 7.

MCS 2.1 G.J.R. FÖRCH, P.J. VAN DER HOUWEN & R.P. VAN DE RIET, *Colloquium Stabiliteit van differentieschema's, deel 1*, 1967. ISBN 90 6196 023 1.

MCS 2.2 L. DEKKER, T.J. DEKKER, P.J. VAN DER HOUWEN & M.N. SPIJKER, *Colloquium Stabiliteit van differentieschema's, deel 2*, 1968. ISBN 90 6196 035 5.

MCS 3.1 H.A. LAUWERIER, *Randwaardeproblemen, deel 1*, 1967. ISBN 90 6196 024 X.

MCS 3.2 H.A. LAUWERIER, *Randwaardeproblemen, deel 2*, 1968. ISBN 90 6196 036 3.

MCS 3.3 H.A. LAUWERIER, *Randwaardeproblemen, deel 3*, 1968. ISBN 90 6196 043 6.

MCS 4 H.A. LAUWERIER, *Representaties van groepen*, 1968. ISBN 90 6196 037 1.

MCS 5 J.H. VAN LINT, J.J. SEIDEL & P.C. BAAYEN, *Colloquium Discrete wiskunde*, 1968.
ISBN 90 6196 044 4.

MCS 6 K.K. KOKSMA, *Cursus ALGOL 60*, 1969. ISBN 90 6196 045 2.

MCS 7.1 *Colloquium Moderne rekenmachines, deel 1*, 1969. ISBN 90 6196 046 0.

MCS 7.2 *Colloquium Moderne rekenmachines, deel 2*, 1969. ISBN 90 6196 047 9.

MCS 8 H. BAVINCK & J. GRASMAN, *Relaxatietrillingen*, 1969.
ISBN 90 6196 056 8.

MCS 9.1 T.M.T. COOLEN, G.J.R. FÖRCH, E.M. DE JAGER & H.G.J. PIJLS, *Elliptische differentiaalvergelijkingen, deel 1*, 1970.
ISBN 90 6196 048 7.

MCS 9.2 W.P. VAN DEN BRINK, T.M.T. COOLEN, B. DIJKHUIS, P.P.N. DE GROEN, P.J. VAN DER HOUWEN, E.M. DE JAGER, N.M. TEMME & R.J. DE VOGELAERE, *Colloquium Elliptische differentiaalvergelijkingen, deel 2*, 1970.
ISBN 90 6196 049 5.

MCS 10 J. FABIUS & W.R. VAN ZWET, *Grondbegrippen van de waarschijnlijkheidsrekening*, 1970. ISBN 90 6196 057 6.

MCS 11 H. BART, M.A. KAASHOEK, H.G.J. PIJLS, W.J. DE SCHIPPER & J. DE VRIES, *Colloquium Halfalgebra's en positieve operatoren*, 1971.
ISBN 90 6196 067 3.

MCS 12 T.J. DEKKER, *Numerieke algebra*, 1971. ISBN 90 6196 068 1.

MCS 13 F.E.J. KRUSEMAN ARETZ, *Programmeren voor rekenautomaten; De MC ALGOL 60 vertaler voor de EL X8*, 1971. ISBN 90 6196 069 x.

MCS 14 H. BAVINCK, W. GAUTSCHI & G.M. WILLEMS, *Colloquium Approximatietheorie*, 1971. ISBN 90 6196 070 3.

MCS 15.1 T.J. DEKKER, P.W. HEMKER & P.J. VAN DER HOUWEN, *Colloquium Stijve differentiaalvergelijkingen, deel 1*, 1972. ISBN 90 6196 078 9.

MCS 15.2 P.A. BEENTJES, K. DEKKER, H.C. HEMKER, S.P.N. VAN KAMPEN & G.M. WILLEMS, *Colloquium Stijve differentiaalvergelijkingen, deel 2*, 1973. ISBN 90 6196 079 7.

MCS 15.3 P.A. BEENTJES, K. DEKKER, P.W. HEMKER & M. VAN VELDHUIZEN, *Colloquium Stijve differentiaalvergelijkingen, deel 3*, 1975.
ISBN 90 6196 118 1.

MCS 16.1 L. GEURTS, *Cursus Programmeren, deel 1: De elementen van het programmeren*, 1973. ISBN 90 6196 080 0.

MCS 16.2 L. GEURTS, *Cursus Programmeren, deel 2: De programmeertaal ALGOL 60*, 1973. ISBN 90 6196 087 8.

MCS 17.1 P.S. STOBBE, *Lineaire algebra, deel 1*, 1974. ISBN 90 6196 090 8.

MCS 17.2 P.S. STOBBE, *Lineaire algebra, deel 2*, 1974. ISBN 90 6196 091 6.

MCS 17.3 N.M. TEMME, *Lineaire algebra , deel 3*, 1976. ISBN 90 6196 123 8.

MCS 18 F. VAN DER BLIJ, H. FREUDENTHAL, J.J. DE IONGH, J.J. SEIDEL & A. VAN WIJNGAARDEN, *Een kwart eeuw wiskunde 1946-1971, Syllabus van de Vakantiecursus 1971*, 1974. ISBN 90 6196 092 4.

MCS 19 A. HORDIJK, R. POTHARST & J.Th. RUNNENBURG, *Optimaal stoppen van Markovketens*, 1974. ISBN 90 6196 093 2.

MCS 20 T.M.T. COOLEN, P.W. HEMKER, P.J. VAN DER HOUWEN & E. SLAGT, *ALGOL 60 procedures voor begin- en randwaardeproblemen*, 1976. ISBN 90 6196 094 0.

MCS 21 J.W. DE BAKKER (red.), *Colloquium Programmacorrectheid*, 1975. ISBN 90 6196 103 3.

MCS 22 R. HELMERS, F.H. RUYMGAART, M.C.A. VAN ZUYLEN & J. OOSTERHOFF, *Asymptotische methoden in de toetsingstheorie; Toepassingen van naburigheid*, 1976. ISBN 90 6196 104 1.

MCS 23.1 J.W. DE ROEVER (red.), *Colloquium Onderwerpen uit de biomathematica, deel 1*, 1976. ISBN 90 6196 105 X.

MCS 23.2 J.W. DE ROEVER (red.), *Colloquium Onderwerpen uit de biomathematica, deel 2*, 1976. ISBN 90 6196 115 7.

MCS 24.1 P.J. VAN DER HOUWEN, *Numerieke integratie van differentiaalvergelijkingen, deel 1: Eenstapsmethoden*, 1974. ISBN 90 6196 106 8.

MCS 25 *Colloquium Structuur van programmeertalen*, 1976. ISBN 90 6196 116 5.

MCS 26.1 N.M. TEMME (ed.), *Nonlinear analysis, volume 1*, 1976. ISBN 90 6196 117 3.

MCS 26.2 N.M. TEMME (ed.), *Nonlinear analysis, volume 2*, 1976. ISBN 90 6196 121 1.

MCS 27 M. BAKKER, P.W. HEMKER, P.J. VAN DER HOUWEN, S.J. POLAK & M. VAN VELDHUIZEN, *Colloquium Discretiseringsmethoden*, 1976. ISBN 90 6196 124 6.

MCS 28 O. DIEKMANN, N.M. TEMME (EDS), *Nonlinear Diffusion Problems*, 1976. ISBN 90 6196 126 2.

MCS 29.1 J.C.P. BUS (red.), *Colloquium Numerieke programmatuur, deel 1A, deel 1B*, 1976. ISBN 90 6196 128 9.

MCS 29.2 H.J.J. TE RIELE (red.), *Colloquium Numerieke programmatuur, deel 2*, 1976. ISBN 90 6196 144 0

* MCS 30 P. GROENEBOOM, R. HELMERS, J. OOSTERHOFF & R. POTHARST, *Efficiency begrippen in de statistiek*, . ISBN 90 6196 149 1.

MCS 31 J.H. VAN LINT (red.), *Inleiding in de coderingstheorie*, 1976. ISBN 90 6196 136 X.

MCS 32 L. GEURTS (red.), *Colloquium Bedrijfssystemen*, 1976. ISBN 90 6196 137 8.

MCS 33 P.J. VAN DER HOUWEN, *Differentieschema's voor de berekening van waterstanden in zeeën en rivieren*, 1977. ISBN 90 6196 138 6.

MCS 34 J. HEMELRIJK, *Oriënterende cursus mathematische statistiek*, ISBN 90 6196 139 4.

MCS 35 P.J.W. TEN HAGEN (red.), *Colloquium Computer Graphics*, 1977. ISBN 90 6196 142 4.

MCS 36 J.M. AARTS, J. DE VRIES, *Colloquium Topologische Dynamische Systemen*, 1977. ISBN 90 6196 143 2.

MCS 37 J.C. van Vliet (red.), *Colloquium Capita Datastructuren*, 1978. ISBN 90 6196 159 9.

MCS 38.1 T.H. KOORNWINDER (ED.), *Representations of locally compact groups with applications*, 1979. ISBN 90 6196 161 0.

MCS 38.2 T.H. KOORNWINDER (ED.), *Representations of locally compact groups with applications*, 1979. ISBN 90 6196 181 5.

MCS 39 O.J. VRIEZE & G.L. WANROOIJ, *Colloquium Stochastische Spelen*, 1979. ISBN 90 6196 167 X.

MCS 40 J. VAN TIEL, *Convexe Analyse*, 1979. ISBN 90 6196 187 4.

MCS 41 H.J.J. TE RIELE (ED.), *Colloquium Numerical Treatment of Integral Equations*, 1979. ISBN 90 6196 189 0.

MCS 42 J.C. VAN VLIET (RED.), *Colloquium Capita Implementatie van Programmeertalen*, 1980. ISBN 90 6196 191 2.

MCS 43 A.M. COHEN & H.A. WILBRINK, *Eindige groepen (Een inleidende cursus)*, 1980. ISBN 90 6196 203 X.

MCS 44 J.G. VERWER (ED.), *Numerical solution of partial differential equations*, 1980. ISBN 90 6196 205 6.

MCS 45 P. KLINT (RED.), *Colloquium hogere programmeertalen en computerarchitectuur*, 1980. ISBN 90 6196 206 4.

MCS 46.1 P.M.G. APERS (RED.), *Colloquium Databankorganisatie*, 1981. ISBN 90 6196 212 9.

MCS 46.2 P.M.G. APERS (RED.), *Colloquium Databankorganisatie*, 1981. ISBN 90 6196 232 3.

MCS 47.1 P.W. HEMKER (ED.), *NUMAL, numerical procedures in ALGOL 60*, Part I: *General information and indices*, 1981. ISBN 90 6196 217 X.

MCS 47.2 P.W. HEMKER (ED.), *NUMAL, numerical procedures in ALGOL 60*, Part II: *Elementary procedures, algebraic evaluations*, 1981. ISBN 90 6196 217 X.

MCS 47.3 P.W. HEMKER (ED.), *NUMAL, numerical procedures in ALGOL 60*, Part III: *Linear algebra, part I*, 1981. ISBN 90 6196 217 X.

MCS 47.4 P.W. HEMKER (ED.), *NUMAL, numerical procedures in ALGOL 60*, Part IV: *Linear algebra, part II*, 1981. ISBN 90 6196 217 X.

MCS 47.5 P.W. HEMKER (ED.), *NUMAL, numerical procedures in ALGOL 60*, Part V: *Analytical evaluations, analytical problems, part I*, 1981. ISBN 90 6196 217 X.

MCS 47.6 P.W. HEMKER (ED.), *NUMAL, numerical procedures in ALGOL 60*, Part VI: *Analytical problems, part II*, 1981. ISBN 90 6196 217 X.

MCS 47.7 P.W. HEMKER (ED.), *NUMAL, numerical procedures in ALGOL 60*, Part VII: *Special functions and constants, interpolation and approximation*, 1981. ISBN 90 6196 217 X.

De met een * gemerkte uitgaven moeten nog verschijnen.