SETL Newsletter 129

J. Schwartz
May 20, 1974

More on SETL in a data-base environment

1. Introduction.

This newsletter

i. takes up the discussion initiated in newsletter 90;

ii. tries to put the discussion on a more reasoned semantic basis;

iii. arrives at a proposal differing substantially from that of NL. 90.

The discussion which follows is much influenced by the proposals made by George Weinberger in his thesis-in-progress.

The basic issues which must be faced in adapting SETL to a data base environment are as follows:

(a) The data collections with which one will deal are very big. In large part, they reside on secondary memory, which can only be addressed on terms very different from, and much less favorable than, the terms of central memory access. This makes it quite essential to be able to search a data-base for particular items without examing more than a fraction of the total data present.

(b) The data base may be thought of as storing vectors, whose components are either object attributes (numaric, character set, etc) or further vectors of attributes. The family of attributes associated with an object will often be quite large and miscellaneous, and will inevitably evolve if a data base is kept in use over substantial periods of time. For this reason it is inappropriate to access attributes by explicit position; a considerably more powerful and flexible scheme of attribute accessing is required.

Such a scheme will aim to stabilise the access process vis-a-vis variations in vector layout. The essential notion in scheme of this type is that of *vector structure* defined by imposing a pattern of named selectors on the vector; this notion is used explicitly in almost every known data management scheme (some schemes inherit their structure notion from the host language in which they are intended to operate.) In the present newsletter, we shall suppose such a structuring concept to be available, and will only refer to details of the assumed structuring scheme as this becomes necessary.

(c) The data-bearing 'vectors' described under (b) generally behave like physical objects, or like 'index cards' or 'dossiers' representing physical objects, in that they have a continuing identity independent of the changing values of their attributes. For this reason, we shall assume that each such vector contains a unique, system-issued serial number defining its continuing identity; and shall call such vectors *records*.

(d) Unlike the data environments of individual programs, which are transient and exist only as long as their associated programs, a data base continues to exist and evolve through the execution and termination of many programs. However, a program working on a data base may create large 'scratch' data objects, which should live no longer than the program. This makes necessary some reasonably efficient mechanism for dividing a data base into subparts of definably different *persistance*.

(e) A large data base will often be more valuable than any of the programs which manipulate it. For this reason, data base *integrity* will often be a very important issue. To guard data base integrity, one must

(ei) aim to prevent inadvertent errors;

(eii) aim to make the willful destruction of parts of a data base difficult;

(eiii) provide backup and recovery procedures which can be used in case of disaster. This last may involve systematic checkpointing and journaling. To detect error situations, utility routines which examine a data-base for consistency may be necessary.

(f) Some parts of a data base may be confidential, so that certain access operations may require security locks. Cf. also point (eii) above.

(g) In an on-line environment, a data base will often be addressed by many query and update processes at the same time. For this reason, mechanisms for coordinating the use of the data base by processes acting in parallel may be required.

(h) In the present newsletter, we shall be interested in explicitly procedural use of a data base. However, a non-procedural set-theoretic query language can be a considerably more natural user tool for much data base usage. For this reason, it is important in designing a family of procedural data-base primitives to ensure that typical queries can be translated into efficient procedural sequences.

2. Design considerations

To design mechanisms which adequately reflect all the desiderata that have just been noted is a formidable task. This full task will by no means be attempted in the present newsletter, in which we will only attempt to discuss issues (a), (b), (c), (d) and (ei), ignoring the others.

Were the sets implicit in typical data-base manipulations not large, and were not the operations to be applied potentially very costly, it might be best not to introduce any SETL dictions specifically for data base manipulation, since the

ordinary SETL dictions (supplemented by appropriate 'structure' dictions, cf. (b) above) are appropriate and would be adequate. This remark serves to remind us that in adding data-base primitives to SETL we aim specifically to optimise the handling of large sets. The issue of continuous identity raised in (c) above could be handled by the SETL newat mechanism, i.e. by associating with each 'record' a blank atom generated to identify it.

In data-base SETL ('DBSETL') we will want a 'record' notion. A *record* is a vector (to be accessed through a *form* defining its layout) for which a unique serial number (or *record identifier*) is issued when the record is created. Much of the data contained in a data base will be resident in its records. We assume that the physical starting point of a record can be reached efficiently if the record identifier is known. From this starting section, all other parts of the record can be reached easily; however, for reasons which will be explained later, we shall not assume that all parts of a record are physically contiguous. One possible implementation technique is to use a master hash table which is keyed by record identifiers and which stores the physical position of every record. An advantage of this scheme is that it allows record accesses to be counted individually, and thus makes possible some form of automatic residence level control.

Nothing prevents us from adding 'record' as a new object type in SETL, and from allowing records to be set members and vector components, as long as our sets and vectors remain 'small' i.e., central-memory storable. This makes available maps from records to records, two-parameter inter-record maps, etc., as long as everthing remains 'small'.

New problems of a specifically 'data base' character arise in dealing with sets and maps which are large. Maps M of this character will typically have some large collection of records either as their domain or as their range. In preparing to deal with such maps we aim to define mechanisms which possess reasonably efficient implementations and which allow set-theoretic semantics, which is our guiding ideal, to be mimicked reasonably well.

The four following cases typify the ways in which we expect to use maps having large sets of records either as their domain or as their range.

(a) Single valued record ⟶ record maps, as for example maps serving to associate subsidiary information with a record without making it necessary to duplicate the record in which this subsidiary information is found.

(b) Multiple-valued record ⟶ record maps. These might either serve the purpose just noted in (a), or might be used to optimise searches by mapping a given record r into the set of all records r' which need to be examined when we seek records standing in some particular relationship to r.

(c) Multiple-values value ⟶ record maps, which serve as 'secondary indices' used to optimise searches by mapping a value v into the set of all records having some attribute equal to v.

(d) (As a rarer case) Single - or multiple values maps f(r,x,y,...) of several parameters. These parameters can be records or values, but at least one must be a record. An example might be a map recording certain 'joint' activities of two persons, each of whom is separately represented by a record.

We may take it that, for given x, f(x,x,y,...) is defined only for a small set of <x,y,...>; indeed, the contrary assumption would make the object f 'quadratically' large, which should be a rare case.

Records have named fields, and in what follows we shall take it that (under certain restrictions to be descussed later) additional fields can be added in a dynamic and efficient way. If n names a field and r is a record, let r(n) access the field n in the record r. Then a map f of type (a) can be handled by treating f(r) as a synonym for r(n); so that maps of this kind are relatively unproblematical. A similar remark applies to maps f(r,x,y,...) in the 'typical' case noted above under (d); we can treat these by associating f with a field n, and treating f(r,x,y,...) as a synonym for r(n)(x,y,...), where we take it that the value of r(n) is a ('small') set. Much the same remark applies to the case (b) above.

Different problems arise in the treatment of value-to-record maps, case (b) above. Here we can no longer assume that the collection of records having some common attribute is necessarily small. For example, if records are grouped by some coarse notion of type, large sets will certainly result. This suggests developing some implementation level mechanism supporting something close to the semantics of the SETL 'set of ordered pairs'(in the large set case). Such a set, which for emphasis we call an *index*, should be capable of serving as the basis for an iteration, and should also serve for the efficient location of a record if one is given a suitable, i.e. index-associated, attribute of the record. Note that we may well wish to avoid storing record attribute values within an index, since the attribute values which would be stored are available anyhow in the indexed records, and double storage might be unnecessarily wasteful of memory.

Note also that in storing values in secondary memory we will often prefer 'explicit copy' to 'pointer' techniques, since secondary memory garbage collection may not be feasible, and it may be important to limit the proliferation of pointers.

Two techniques for the storage of indices suggest themselves. The first involves ordering, the second employs a hashing technique. In both techniques, indices consist of collections of pointers to records. In the first technique, the pointers are kept in a logically sorted order, possibly as the nodes of a balanced multi-way tree, and records with a desired attribute value are located by a binary search process. In the second technique, the pointers are kept in a hash table, and records with a given attribute value are located by hashing. We shall call these two possible approaches the *ordering technique* and the *hashing technique* respectively. The hashing technique can permit a record of known key to be accessed via an index in a number of probes independent of the size of the index. It has in addition the advantage of being strongly consonant with other aspects of the SETL approach. If the ordering technique is used, then to access a record of known key k within a collection of n records one will generally need to make log n probes. However, a rather strong argument can be made, even at an abstract level, in favor of the ordering technique. This argument rests on the observation that *coordinated processing* is an important file processing technique. More specifically, when two or more files f_1, f_2,... are to interact logically in order to produce some result (e.g., another file) it will often be the case that each record of f_1 needs to interact only with some few records of f_2,... In such situations, it is often possible and can be quite useful to arrange all the files f_1, f_2,... in an order chosen so that if r_1 and $\bar{r}_1$

are records of f_1, if r_1 preceeds $\bar{r}_1$, and if r_j and $\bar{r}_j$ are records of f_j with which r_1 and $\bar{r}_1$ respectively need to interact, then r_j preceeds $\bar{r}_j$. When such an arrangement has been set up, processing can move efficiently down the files f_1, f_2,... in their established order. This general observation accounts for the importance of sorting in the processing of data-files: sorting can be regarded as a process of concentrated (and optimised) ordering which brings large data collections into an order permitting the coordinated processing of several files. If only a hashing technique and no ordering technique is used within a data-base system, use of sort-like techniques will become clumsy. Appropriately chosen value-to-record maps can substitute for orderings, but not always easily, especially when the orderings appropriate to a particular application are based not on particular attributes but on subtler implicit record properties.

Note also that in dealing with large objects having structural properties S that a programmer may aim to exploit, it is important to provide differential modification proceduces which preserve the properties S. For example a system in which ordered sets of records are supported must allow records to be inserted and deleted with preservation of order. Pointer rather than value semantics will be appropriate for large objects, since object copying is bound to be a very expensive operation.

Related issues enter if one assumed that the secondary memory on which the bulk of a data-base resides is a disc or drum type of memory in which a substantial inital penalty attaches to each access. In using memories of this type, it is natural to move data by 'pages, i.e., to bring several hundred adjacent words of data from secondary memory each time a program calls for as much as one word.

In such an environment, physical data contiguity becomes an important issue. Here again semantic concepts based on order are found to posess advantage, since a principle of contiguity grows naturally out of a notion of order, but not out of a hash-oriented technique. The question of physical contiguity has also an intra-record aspect. It is clear that, if all the sections of a record are contiguous, the number of records which are packed onto a physical page when records are grouped together is inversely proportional to the size of a record. For this and other reasons it may be desirable to allow rarely consulted, relatively 'passive' sections of a record to be placed remotely from record sections which are more actively consulted. A mechanism of just this kind will be described in the pages which follow.

It should be noted that questions of physical contiguity become far less compelling if one envisions an environment in which bulk data is stored on purely electronic rather than on electromechanical memories. In such an environment, some of the record-order related proposals made in the following pages may become superfluous.

3. Persistence; the notion of 'area'.

The programmer using a data-base system will sometimes wish to create large temporary files, and if this is to be possible the system will have to be able to recognise the temporary character of these files. File structures interconnected in complex ways by inter-file mappings may be created temporarily and require erasure. Because of the large size of the data sets which a garbage collector would have to search, garbage collection may not be a feasible storage recovery technique, and one may have to fall back on a semantic mechanism involving explicit allocation and deallocation calls.

It is this problem that is addressed by the 'area' mechanism proposed below. An area is a dynamically growing collection of memory pages within which the abstract objects of a data base may be stored. We allow the data-base-SETL user to create indefinitely many areas. An area may be deleted; when this is done, all the objects stored within the area are also destroyed.

To begin accessing a data base, a program must gain initial access to certain objects of the data base, which then function as indices and templates putting the rest of the stored data within reach. The conventions which make this possible form the nexus between a data management language and the operating system which this language presupposes. These conventions can of course be set up in various ways. One possibility is the following:

a) The objects externally cataloged will be areas, each of which will be known by some 'catalog name' (a character string) to the operating system (which maintains externally cataloged objects between program runs.) A program desiring to access an area a will evaluate the function

(1) open *catname*,

where *catname*, a character string, is the catalog name of a. The value returned by (1) will be a pair, whose first component is a, and whose second component is a (formed) record r called the *prime record* of a. The record r has various standardised attributes,which are essentially indices to the remaining information in a.

b) To catalog an area a, one executes

(2) close <a, r, *catname*>.

Here *r* is a (formed) record, which becomes the prime record of the cataloged area, and where *catname* is a character string which becomes its catalog name.

c) Areas created by a program but not cataloged before its termination are automatically deleted by the data base operating system.

4. Integrity: automatic maintainance of secondary indices.

The complicated question of data-base integrity can be approached at several levels. At an elementary level, one may note the importance of preventing implausible values from being stored in the fields of a record. These errors will doubtless tend to be very common in practice; by preventing it one can also bring more subtle errors to light. Errors of this type are easily handled by building appropriate mechanisms into the record access interpreter; all that is required is a suitable generalisation of the notion of record form.

When secondary indices are used to speed up access to the records of some primary file, any operation which modifies, inserts, or deletes a record may require a compensating adjustment in an index. It is annoying and error inducing for a programmer intending some relatively straightforward data modification to have to remember and explicitly insert all these subsidiary operations. For this reason, some data base systems designers (notably the DBTG design group) have tried to specify declaration-based record indexing schemes that remove the maintainance of secondary indices from the ken of the system user, making this maintainance largely automatic.

In such schemes, one can for example declare that all records of a given type are to be 'mandatory' members of some ordered file, and are to be located by some pre-stated search strategy. Then any modification of data in the record automatically triggers appropriate index-update action. The problem with the schemes of this type that have been proposed to date is that they are not foolproof, but impose what may be a complex requirement of cooperation on their user, who if he misses something, can cause important 'automatically maintained' indices to pass into an unanticipated and undesirable condition. This problem will be particularly acute in systems maintaining numerous and miscellaneous collections of secondary indices, systems in which location of the right secondary index wherein to find a desired record becomes a matter that is at least partly procedural.

Not knowing how to solve this problem, we evade it, omit any automatic index maintainance features from the proposals which follow, and leave it to the programmer to handle index updating procedurally. By this evasion we in effect imply that it is impossibly difficult to provide foolproof automatic index maintainance in an environment allowing procedural file manipulation. In this view, automatic index maintainance can only be provided in a still higher level, non-procedural,'query' or 'transaction' oriented language, relative to which a procedural data-base language is an implementation underpinning. However, we will suggest a number of syntactic forms which facilitate the programming of index updating operations.

5. Error handling

In the manipulation of files, error situations will often arise: e.g. one may attempt to insert a record into an inappropriate index, try to read a nonexistent field in some record, try to write bad data into the fields of a record, etc. A data-base oriented language should provide some adequate means for treating these errors. It is important for the programmer to be able to centralise whatever set of error handling procedures he supplies, so that error tests and error routine calls do not proliferate in his code. A semantic primitive good for this purpose is the PL/1-like 'ON' construct. SETL lacks this feature, and for data base application should probably be extended to include it, but in a form restricted to avoid undesirable implications for the analysibility of DBSETL programs.

6. Proposals.

Guided by the general semantic discussion which preceeds we now come to make a number of specific syntactic and semantic proposeas. We propose to extend SETL to DBSETL by adding the following semantic object classes: *records*, *indices*, *files*, *areas*, and *forms*. Concerning these, we make the following introductory heuristic observations.

1. *Records* are essentially tuples for which a system of named 'selectors' have been defined, i.e., 'formed vectors' in the sense of O.P. Item 20, section B. In addition, each record has a system-issued unique serial number, which gives it 'continuing identity', and through which the body of the vector can be located efficiently. In this sense, records have 'pointer' semantics. The components of a record are accessible only through the record itself.

ii. *Indices* behave rather like 'large' SETL sets of ordered pairs, sets of pairs in which the second component of each pair is a record *r*, the first component of each pair being a quantity calculated from specified fields of *r*. Like SETL sets of pairs, indices are accessed by a hash technique. However, the hash table used contains only record identifiers, i.e. only second and not first components of stored pairs. Each index *inx* possesses a *lead record*, r_0, whose single attribute is a function ϕ which (normally) maps each record *r* of the range of *inx* into the quantity *a* whose *inx*-image is *r*. The function ϕ will be called the *identifying function* of *inx*. Each index *inx* also possesses a system-issued identifying number which identifies *inx* uniquely, giving *inx* continuous identify and pointer semantics, and through which *inx* can be located physically. Indices *inx* support the following operations:

(a) The diction

(1) r = inx(a);

designates the record *r* into which *inx* maps a, if this record exists and is unique. Otherwise, (1) has the value Ω. Note that in using the diction (1) to retrieve *r*, the quantity *a* hashed in the normal SETL manner and the resulting hash index is used as the starting point of a search. In this search, one uses the identifying function ϕ associated with *inx*, and looks for one or more records *r'* such that $\phi(r')$ eq *a*. If exactly one such record is found, this record is the value of (1); if no such record, or more than one such record is found, then the value of (1) is Ω. Note that we do not assume that indices are automatically maintained by the DBSETL system. That is, changes in the attribute values of a record *r* do not automatically trigger compensating changes in every one of the indices in whose range *r* occurs.

Invocation of the re-indexing operations needed to compensate for changes in attribute values is left to the DBSETL user. Suppose then that *inx(a)* initially evalnates to *r*, and that an assignment *r(attr) = expn* changes the value of some attribute of *r*, but that no compensating modification of *inx* is made. Then *inx* may 'lose track' of the record *r*, in the sense that *inx(ϕ(r))* and *r* lose the relationship which they should always have in a well-formed index. In such case, *r* will differ from every value *inx(a')*, even though *inx* will contain a pointer to *r*. Such situations are of course undersirable, but (because to do so would require an expensive dynamic mechanism) we do not furnish DBSETL with semantic mechanisms which automatically and universally present such situations from arising.

(b) The diction *inx(a)* can be used in sinister position, i.e. in the context

(2) inx(a) = r;

where *r* is a record-valued expression. The assignment (2) has the effect of deleting, from *inx*, all pointers to records *r'* which would otherwise belong to *inx(a)* (see below). After this deletion, a reference to *r* is inserted in appropriate position. Immediately after (2) has been executed, (1) will evaluate to *r*. The special case

(2a) inx(a) = Ω;

of (2) simply deletes form *inx* all pointers to all the members of *inx(a)*.

(c) The diction

(3) inx{a}

designates the set of records r which are the image of a under *inx* and (much as with sets of ordered pairs) is the 'multi-valued' analog of the 'single-valued' diction (1). Wherever-possible, DBSETL will avoid explicit formation of the set (3), and will treat (3) merely as a formal notation useable in iterators like

(4) $\forall\ r \in inx\{a\}$

Note that (4) will iterate over all records r for which $\phi(r)$ eq a and which can be located by a hash chain whose starting position is calculated from a in standard fashion. The same remark applies when in the usual SETL manner the iterator (4) becomes part of a compound operator or other iterator-based construction, as e.g. in

(5) $[+:\ r \in inx\{a\} |\ r(attr_1)\ \underline{eq}\ c_1]\ r(attr_2)$.

(d) The diction

(6) $inx[s]$

designates the set of all records r into which *inx* maps any element a of the set s. DBSETL will wherever possible avoid explicit formation of the set (6) also, and treat (6) as a formal notation occuring in iterators. The related diction

(7) $inx[\ ]$

designates the set of all records r referenced by pointers in *inx*, and is provided principally for use in the iterator

(8) $\forall\ r \in inx[\ ]$

and in related iterator-based constructions.

Note that if ϕ is the mapping associated with the index *inx* (as in (a) above) then

(9) $\{ r \in inx[\,] \mid r \underline{n} \in inx\{\phi(r)\} \}$

is the set of records 'lost' or 'misfiled' within *inx*; unless a DBSETL user is striving for some very special effect, he will normally wish to program so as to keep all sets (9) equal to $n\ell$.

(e) The diction

(10) r **in** inx{a}

requires that a **eq** $\phi(x)$, and given this condition ensures that an immediately subsequent evaluation of the set inx{a} will include r. The diction

(11) r **out** inx{a}

drop r from the collection of records to which *inx* points. The diction

(12) inx{a} = s;

may be regarded as a shorthand for

(13) inx(a) = Ω; (∀r∈s) r **in** inx{a};; .

The diction

(14) # inx

yields the number of record-pointers stored in the index *inx*.

iii. This temporarily completes our account of the semantics of indices, and we now turn to discuss the semantics of *files*. Files behave rather like 'large' SETL tuples whose components are ordered and accessible by numerical position. Each file *fil* possesses a system-issued identifying number which identifies *fil* uniquely, giving *fil* continuous identity and pointer semantics. Files *fil* support the following operations:

(a) The diction

(15) fil(n),

where n is an integer, retrieves the n-th record of *fil*. This diction may also be used in sinister position, i.e. in the context

(16) fil(n) = x;

where x is a record-valued expression. Files are not allowed to have undefined components which preceed defined components of the same file. Thus the integer (15) must satisfy $n \leq \#\,fil + 1$. Note here that the diction

(17) # fil

is available and gives the number of (defined) components of *fil*.

(b) Insertion and removal operations are provided for files. The removal operator has the form

(18) fil(n) = Ω;

The operation (18) has a somewhat different semantics for files that for SETL tuples. When *fil* is a file, (18) both removes the n-th component of *fil* and renumbers its remaining components. The corresponding operation on a SETL tuple *t* is expressed by

(19) t = t(1:n-1) + t(n+1:);

The insertion operator corresponding to (18) has the syntactic form

(20) fil(n) =: r;

where *r* is a record-valued expression. It inserts a new component into *fil*, which becomes *fil's* *n*-th component . The former *n*-th thru final components of *fil* are renumbered and become *fil's* *n* + 1'st thru final component. The corresponding operation on a SETL tuple *t* is expressed by

(21) t = t(1:n-1) + <z> + t(n+1:).

(c) DBSETL provides an iterator over files, which has the syntactic form

(22) ∀r(n) ∈ *fil* starting *E* while C_1 | C_2.

Here, *E* is an integer-valued expression whose value n_o determines the starting component $r(n_o)$ of the iteration (22), C_1 is a boolean expression which terminates the iteration as soon as it becomes false, and C_2 is a boolean expression which serves in the usual way to allow certain cycles of iteration and bypass others when an iterator (22) is invoked. Any of the 'starting E' clause, the 'while C_1', and the '|*C*' clause of (22) may be omitted.

Iterators of the form (22) may in the usual SETL manner become part of a compound operator or other iterator-based construction.

A file, index, or record *obj* can be deleted by executing the command

(23) **delete** *obj*;

iv. Now we come to discuss the semantics of *areas*. As already noted, areas are provided because they make possible certain wholesale data manipulations (especially erasures) and because they provide a modified 'allocate-deallocate' action for use in situations in which garbage collection may be infeasible. The following semantic rules relate to areas:

(a) Every index and file belongs to some area. An area *a* may be deleted by executing the command

(24) **delete** *a*;

when an area is deleted, every object (index, file, or record section, see below) contained within it is also erased.

(b) In a manner whose details will be explained below, every record known to the DBSETL system can be seperated into *sections*. Every record section belongs to some area.

(c) An area is created by a call on the parameterless system function

(25) **newarea**.

(d) *File former* and *index* former dictions are provided in the SETLB system. The index-former diction is

(26) {: *area-expn, fcn-expn: record-expn, iterator*}.

In (26), *area-expn* denotes an area-valued expression, whose value determines the area within which the index (26) will be formed; *fcn-expn* denotes a function-valued expression, whose value determines the identifying function ϕ associated with the index (26). Moreover, still in (26), *iterator* designates any iterator which could appear in a set-former; this iterator controls the addition of records to the index *inx* which (26) forms. Finally, *record-expn* denotes a record-valued expression, whose sucessive values become the records which (26) references. Each record *r* referenced by *inx* is initially a member of $inx(\phi(r))$.

We allow the *fcn-expn* in (26) to be omitted, in which case the 'trivial' function ϕ identically equal to nl is used as the identifying function of *inx*. This albreviated construction can be used to form 'degenerate' indices which serve to represent 'large' collections of records. We also allow this fcn-expn to consist of the sign '-' followed by a qualified name *nm* (or vector of qualified names $<nm_1,\ldots,nm_k>$) in which case the identifying function of *inx* is understood to return the value $r(nm)$ (resp. $<r(nm_1),\ldots,r(nm_k)>$ when applied to a record *r*.

The file-former diction is

(27) <: *area-expn, record-expn, iterator* >.

The syntactic parts *area-expn*, *record-expn*, and *iterator* appearing here have the same definition as they had in (26);

note again that the value of *area-expn* determines the area within which the file (27) will be formed. An empty file may be formed within a given area by invoking the prefix operator

(28) **nulf** *area-expn*.

A null index with specified identifying fuction can be formed by writing

(29) **nuli** <*area-expn*, *fcn-expn*>,

and a null index with trivial identifying function can be formed by writing

(30) **nuli** *area-expn*.

For indices *inx*, the dictions

(31) *x* **in** inx and *x* **out** *inx*

are abbreviations for

(32) *x* **in** *inx*(**nl**) and *x* **out** *inx*(**nl**)

respectively.

(e) An index (or file) *i* may be deleted by writing

(33) **delete** i;

(f) The operation of *sorting* produces a file f out of an index or file *i*; the components of f will be arranged in increasing order of some programmer-specified binary function *comp-fcn* whose arguments are records.

To produce such a sorted f within a given area, we invoke the prefix operator

(34) sorted <*index-or-file, area-expn, comp-fcn*>.

Here *index-or-file* is an expression whose value is the index or file whose sorted form is the value of (34).

One will often want sort records into an order depending not only on the integers, character strings, etc. which they contain, but also on such quantities as record and file identifiers. To make this possible, we allow the operators le, lt, ge, gt to be used to compare record, file, index, and area identifiers, and also blank atoms. These operators will return results which, although implementation-dependent, obey the expected transitivity rules..

v. Now we turn to describe the semantics of *forms*. Forms serve to make the attributes of a record, which will often be numerous and miscellaneous, accessible through a rational, structured family of *qualified names*. The system of forms which we propose to use in DBSETL is essentially that described in O.P., Item 20.B, but extended so as to allow the specification of seperate areas of residence for seperate *sections* of a record. The basic semantic notions of the scheme to be described are *record*, *form*, and *formed record*, the latter two of which are closely comparable to the notions *form* and *formed tuple* of Item 20.B. A formed record *fr* is a SETL pair whose first component is a record *r* and whose second component *f* is a *form* which serves to guide all access operations which address *r*. The record *r* is called the *body* of *fr*; in addition to the components which are given names by *f*, *r* will also contain auxiliary components, not accessible at the DBSETL user level, which help in locating user-level components of *fr*.

However, these issues are of system and not of user concern, since the DBSETL user will always access the components of a record by attaching a form to r, following which r's components will be addressed by name and not by position.

When reference is made to a component of a formed record fr, a system-level *access interpreter* will be invoked. The parameters passed to this interpreter are as follows: the body and the form of fr, the name of the component to be accessed, and, in case this component is to be modified, the new value to be established. The *form* component of f serves as a kind of program during the action of the form interpreter.

It will sometimes be necessary to attach more than one form to a given body r; for example, information stored within r itself, and accessible through some 'provisional' form, may signify the form which r has. To make bodies accessible through more than one form, one requires a *compatibility rule* determining the cases in which a component of given name will be sought in a fixed position within a record r, even if different forms f_1 and f_2 are used to govern two accesses to r. For this purpose we adopt the compatibility rule described in Item 20.B, which is based on the fact that the component names supported by a form f always constitute an ordered sequence. For convenience sake, we restate the compatibility rule (cf. Item 20.B): if two forms f_1 and f_2 both support a given component name n, and if in the sequences of names supported by f_1 and f_2 respectively all names preceeding n are identical, then the n component of a record can be accessed using either f_1 or f_2.

Much of the remaining semantics of forms and formed records is taken from Item 20.B with little change, and we shall not describe this semantic scheme in detail; prefering to refer to that Item, we shall only give details where an extension to or modification of the scheme outlined in 20.B is necessary.

In brief, the semantic facilities outlined in Item 20.B may be enumerated as follows: Forms can be created by writing

(35) newf (formexpn),

where *formexpn* denotes a *form expression*. A form expression can be either

a. A SETL name, which then names an attribute;
b. A list of names seperated by colons, which then become synonymous attribute names;
c. A construct of the syntactic structure

(36) *namelist (formexpn).*

As explained in O.P. Item 20.A and 20.B, the use of such constructs allows us to define systems of structured attribute names, which have the appearance $n_1. n_2 \ldots n_k$, where each n_j is a simple name.

d. The comma may be used as a *form concatenation* operator, and the operator or may be used as a *form alternation* operator. The information necessary to indicate which of two alternative forms f_1, f_2 actually applies to a record r of the form f_1 or f_2 is maintained by the access interpreter and stored in an auxiliary component of r.

e. A form expression may be a *tuple expression* of the structure

(37) * (*formexpn*).

As previously, tuple expressions of this kind are used to indicate that some particular attribute of an object is a tuple from which subattributes may be retrieved by indexing.

f. As in Item 20.B, the construct

(38) like *expn*

may be used both to include parts of previously defined forms into a new form being constructed by a call (35), and to allow the full collection of SETL dictions to be used in the construction of forms.

g. 'Conditional' form expressions, having the syntax

(39) if $boolex_1$ then $formexpn_1$ else ... else if $boolex_k$ then $formexpn_k$

and 'calculated' form expressions having the syntax

(40) *form-vector* (asf $formexpn$, *(qualname)*)

are both provided, and have the semantics described in Item 20.B.

h. If r is a record and f a form, then

(41) r asf f

denotes the formed record whose body is r and whose form is f. The diction (41) may also be used in case r is a formed record, in which case it denotes the formed record whose body is the same as the body of r and whose form is f.

Simple records, rather than formed records, are written to secondary memory (by being inserted into indices or files using the operations described in paragraphs ii and iii above). If a formed record appears in a secondary memory write operation which expects a simple record, the form of the record is stripped off and its body written. This convention avoids repeated recopying of forms.

This completes our reprise of the form-related semantic facilities which are available both for formed tuples in ordinary SETL and for formed vectors in DBSETL. We shall now describe a few additional features which are available only in connection with formed vectors. The first of these is a construct

(42) *namelist* [*formexpn*]

which resembles (36) in its syntax and semantics. In (42), as in (36), *namelist* is a colon-seperated list of names, which become synonymous references to the part p of a formed record which (42) describes. However, the use of (42) rather than (36) indicates that p, which is of course part of a larger record r, constitutes *a record section*, which among other things implies that p may be stored in a different *area* from the remainder of r.

As already stated, forms are created by invoking the operator (35), and serve as specialised programs determining the actions of a system-level access interpreter routine which is used whenever a field of a stored record must be either retrieved or modidied. To give additional flexibility and power to the access interpreter, we allow up to four information records to be associated with a form f. These records contain information determining:

α) the pattern in which subsections of a formed record with form *f* are assigned to areas;

β) 'initial' field values, actually values to be used when one accesses an attribute that has never been explicitly set either during the creation or since the creation of the record possessing it;

δ) any restrictions on values which can be stored in particular fields of *r*;

δ) any locks restricting access to particular fields of *r*.

These records can be retrieved by writing

(43a) areainf *f*
(43b) initinf *f*
(43c) protinf *f*
(43d) and lockinf *f*

respectively. The latter three of these records have the form *f* itself (but consist of a single record section). The record areainf *f* has a form *f'* derived from *f* by deleting each subpart of *f* which contains no occurence of the construct

(44) *namelist* [*formexpn*]

and then by replacing surviving occurences of (44) in the form expression defining *f* by a corresponding occurence either of

(45) *namelist* (residencearea)

(if the *formexpn* of (44) contains no further suboccurence of the construct (44)) or of

(46) *namelist* (residencearea, *formexpn'*),

(if the *formexpn* of (44) contains further suboccurences of (44), and where *formexpn'* is devived from *formexpn* by recursive application of the rule of replacement just stated.) In both (45) and (46), *residencearea* is a reserved literal. Note, as an example of the rule just stated that if *f* is defined by the form expression

(47) part1, part2 (outer [field], field2,
inner [first [field1, field2],second])

then *f'* is defined by the form expression

(48) part2 (outer(residencearea),
inner (residencearea,first(residencearea))).

Fields in the records (43a-d) associated with *f* can be set by writing

(49a) areainf *f (qualname)* = *val*;
(49b) protinf *f (qualname)* = *val*;

etc. Here *qualname* is a qualified name appropriate to the formed records areainf *f*, protinf *f*, etc., and of course *val* represents a value to be assigned.

The semantic significance of the several records (43a-d) is as follows. Let *qn* designate a qualified name specifying an attribute of records having the form *f*. Then *p* = (protinf *f*)(*qn*) can be either Ω, a SETL type, a pair of integers, a pair of reals, a set, or a boolean-valued function of one parameter. If *p* is not equal to Ω, then when the field *r(qn)* of a formed record *r* having the form *f*

is modified, one of the following checks will be made, and suitable error action taken if the check fails;

i. if p is a SETL type, the type of the new value $r(qn)$ must be p;

ii. if p is a pair $\langle x_1, x_2 \rangle$ of integers or reals, then $r(qn)$ must satisfy $r(qn)$ le x_2 and $r(qn)$ ge x_1;

iii. if p is a set, then $r(qn)$ must belong to it;

iv. if p is a function, then $p(r(qn))$ must have the value true.

If one creates a formed record r with form f, and then, without ever setting the value of some particular attribute $r(qn)$, accesses $r(qn)$, the value returned will be (initinf f)(qn). Thus the record initinf f stores a family of 'default values' that can be used with any other record of form f.

Similarly, p = (lockinf f)(qn) determines the system of security locks which will be applied when the field $r(qn)$ of a record with structure f is accessed or modified. Since we do not wish to enter into an extended discussion of protection-related issues, we shall not supply any details concerning the internal structure or semantic treatment of the quantity (lockinf f)(qn)

Next, let qn designate a qualified name specifying an attribute of records which have the form f' derived from f by transforming each occurence of (44) into an occurence of (45) or (46) in the manner described above. Then qn addresses both a field of the formed record areainf f and a section of each record r having f as its form. The quantity p = (areainf f) (qn) can be either Ω or an area.

Let *r* be a formed record with form *f*. If *p* is Ω, then the section *s* of *r* addressed by the qualified name *qn* is placed within the same area as the 'main' or 'initial' section *s'* of *r* when *s* is created (by a first assignment to one of the fields of *s*). (The area in which *s'* is placed is determined, in a manner to be described shortly, when *r* itself is created). If *p* is different from Ω, then when *s* is created it is placed within the area *p*.

It would be quite easy to use mechanisms like those described in the predeeding paragraphs to associate encoding/decoding actions with modifications/retrievals of record attributes. We do not provide such a feature because relatively concrete and strictly efficiency-oriented considerations of this sort are somewhat foreign to the SETL spirit.

To create a record with form *f*, one invokes the function

(50) newrecord (*f*, *area*).

Here *area* is an area-valued expression; the newly created record is made resident in *area*. Records *r* created by a call (50) have all their attributes initially undefined. (Consequently, (initinf f)(*qn*) will be returned if *r* is created by a call (50) and *r(qn)* is accessed immediately after *r* is created.) To allow creation of a record *r* of form *f*, some of whose fields are set to values different from (initinf *f*)(*qn*), we provide a variant of (50), having the syntax

(51) newrecord (*f*, *area*, (*initialised-attributes-list*), *initial-values-tuple*).

The quantities *f* and *area* have the same significance in (51) as in (50); *initialised-attributes-list* is a list of qualified names validly addressing attributes of a record *r* of form *f*, and *initial-values-tuple* is an expression evaluating to a tuple

t. This tuple must have as many components as there are qualified names in the *initialised-values-list*; the successive fields named in this list are initialised to the successive values occuring as components of *t*. Note (from D.P. III, Item 20.A) that the construct

(52) $\langle \forall$ *qualname-list*$\rangle$r,

where *qualname-list* is a list of qualified names and r a record, forms a tuple by extracting from the fields named in *qualname-list*. Thus the construction

(53) newrecord(*f*,*area*,*initialised-attributes-list*,$\langle \forall$ *qualname-list*$\rangle$r)

serves to form a new record with certain of its fields initialised with values taken from particular fields of the record *r*.

If a form *f* has the structure

(54) *name* [*formexpn*],

and if in addition a=areainf *f*(*name*) is not Ω (so that a specifies an area) then the *area* parameter of (50) or (51) can be elided and a record of form *f* created by a call

(55) newrecord (f)

or

(56) newrecord(f,(*initialised-attributes-list*),*initial-values-tuple*).

7. Physical contiguity; Space reclamation.

We noted in Section 2 that the bulk of a data base may very well reside on a disc or drum type of memory in which it is natural to move data by 'pages' consisting of several hundred adjacent words, and that in such an environment physical contiguity becomes an important issue. If this is the case iterations extended over whole files will require relatively few page accesses and run relatively rapidly if *all* the data items

which must be accessed during the iteration are physically placed in logically corresponding order. During such an iteration, one will have to access:

(a) the pointers which constitute the file;

(b) the records to which these elements point;

and, as an auxiliary

(c) the implementation-level master catalog which gives position of every record.

This last data structure is necessary since we assume that each record has a serial number giving it continuing identity, and that a record can be located efficiently if this identifier is known. Note that since the size of record attributes is allowed to vary dynamically, records may have to be moved in physical storage; of course, the master catalog entry that locates a record should not move.

It is reasonable to suppose that records created within an area A are issued serial numbers in their order of creation; this convention, which makes deletion of whole areas particularly convenient, implies that a record is fully identified by the area to which it belongs and by its serial number within this area. We may assume that the master catalog for each area is maintained in a physical order corresponding to the increasing order of record numbers. After execution of a delete statement (23), or of the space-recovery operation described below, gaps can appear in the sequence of record numbers recorded in such a catalog. Note that areas will also have identifying serial numbers, recorded in an implementation-level 'master catalog of areas'.

A file, which is essentially a large tuple, can be stored in a balanced multiway tree, the record identifiers which constitute the file being kept physically in the logical order in which they occur within the file. Use of balanced multiway trees makes element insertion and deletion efficient.

The records created within an area can initially be placed in positions corresponding to their order of creation, and each record can be left in its initial position unless it grows so large that it must be moved. For each physical page, information distinguishing'sections in use'from'sections not in use'can be maintained, and the page can be replaced when appropriate. Moreover, account can be kept by separate areas of the total size of'sections not in use', and areas for which this total has grown substantial can be compressed overall.

The conventions which have just been outlined imply that for a newly created file consisting of newly created records, the physical order of records, of record identifiers within the file, and of record locators within the master catalog locating the records will all correspond. Thus iterations extended over such files will require particularly few page accesses and will be particularly efficient. For this reason, if it is anticipated that iteration will often be extended over a file, it may be advantageous to use the file former diction (27) to create a new copy of both the file and its records (or of relevant record portions), even though the copying operation itself is expensive.

As already noted, garbage collection will probably not be

a practical technique for recovery of unused space in large file systems. The *area* construct and the delete statement (24) provide an acceptable means for recoving the space occupied by substantial data sets generated for predictably temporary use. However, this technique cannot be used to recover space occupied by records which are initially made part of several files and subsequently removed from some of them. For use in such cases, it might be appropriate to provide an explicit *purge* statement of the form.

(57) purge *area-list: file-index-list;*

Here we take *area-list* to be a comma-separated list of area-valued expressions, and *file-index-list* to be a list of file- or index-valued expressions. When (57) is executed, every record in any of the areas occuring in *area-list* which is not referenced by a file or index in the *file-index-list* will be deleted, and the space occupied by these records recovered. If a record is deleted from an area, the master catalog entry which locates it will also be eliminated.

Appendix: A Critical and Supplementary Comment on the Foregoing.

By Phillip Shaw

I. Characteristic Problems of "Data Base" Systems

Following general usage, by "data base" we mean systems in which a diverse collection of applications share a common pool of information, as distinguished from single-application systems (even online systems such as airline reservations) and from systems in which the various applications maintain private files (whether on the same or on different systems). In general, the relative merits of data-base versus private-file systems reflect the trade-offs involved in any centralization-decentralization problem, and the disadvantages of data-base systems then form the so-called "data base problem".

The advantages of data-base systems are evident: There is at least potentially a lower per-application cost for the information, and the various applications are consistent in their "facts" (e.g. reports from "payroll" and from "personnel" would agree as to how many "employees" there are).

The drawbacks of data-base systems relate to three areas: conflicting requirements, interference, and the need for automatic recovery operations.

Conflicting requirements

The various applications sharing a data base generally differ substantially in regard to what information is needed, and how it is to be accessed. Private-file systems have quite the advantage here, in that the files can be exactly tailored to application requirements.

In data base systems the stored information is more than any one application needs, and the organization is ideal for at most one of the applications. Other applications must make-do with supplementary link-fields and secondary indexes, so that access is somewhat less efficient and programming considerably more complicated and error-prone.

Interference

In shared-data systems, a "read" operation must be delayed if an "update" on the same data item is in progress, and an "update" operation must be delayed if either a "read" or an "update" is in progress. Clearly such interference (and the complexity involved in detecting and coordinating such conflicts) is not encountered in private-file systems. Also, in order to achieve adequate levels of utilization and responsiveness, data-base systems generally have to coordinate requests on the basis of smaller units of information for shorter periods of access than has been required in batch or time-sharing systems: locking at the "record" rather than "file" level, and assigning locks to "transactions" rather than to "jobs" or to "sessions". All of which tends to increase the complexity of both the application programs and of the underlying system, over what is required for private-file systems.

Another, more severe, form of interference is caused by incorrect programs and data. The data-base can be (correctly) updated with incorrect information; or, it can be incorrectly updated by an erroneous program; or, it can be partially updated by a program which aborts in themiddle of updating a file or record, leaving the information in an inconsistent and incorrect state. In private-file systems these problems of course affect only the single application, whereas in data-base systems the effect can be considerably more widespread.

Automatic Recovery

"Recovery" refers to the actions which restore a system to good working order after a failure has been detected. This is inherently more difficult in data-base systems than in file-oriented systems, because of the complexity of the data structures and the incremental nature of the update operations.

Data-base systems are also generally operated "online", with transactions entered from remote terminal; and, they tend to be highly integrated into the operation of an enterprise. This, coupled with the very fact that the data-base is shared by numerous applications, means that the impact of failure is both more widespread and more severe than is the case with private-file systems. Recovery operations must therefore proceed at speeds which preclude manual intervention, and so must be pre-planned and programmed.

II. Functional Data Structures

It is possible to completely define a data structure or class of data structures by specifying a collection of procedures which implement the operations which are defined for instances of the data structure. Typically one would provide procedures for operations of create, select, update, perhaps size, etc. Languages which provide such a capability include GEDANKEN, POP2, PPL, SIMULA-67, EL1, and SETL. The primary design variations relate to

a) whether the number and meaning of the "entries" or operations is open-ended and arbitrary (as in SIMULA-67) or fixed and standardized (as in all of the others); and

b) whether the representation, or basic data, of such data structures can be accessed only by the defined procedures (as in GEDANKEN and I believe SETL) or also by other routines (as in the other languages above).

I feel that such an approach, with an open-ended number of procedures and with only the defined procedures able to access the data structure, has very great potential value for data base systems. This is because such an approach makes it possible to centralize certain critical operations.

The operations which one would want to centralize (in the access procedures) would be those which

a) are dependent on the particular organization of the data; or
b) affect the consistency of the data; or
c) require exclusive access to the data.

For example, one might program a functional data structure which would be accessed as if it contained the payroll file sequenced by employee-name; internally, however, the access procedures of that data structure might in fact obtain the required data from several files, perhaps following supplementary link fields or using secondary indexes. All of those details would, however, be known only to the access procedures. Thus, they are the only procedures which would have to be changed if the "real" data structures were changed or re-organized.

As another example, if all updates to the salary fields are performed by calls to such an access procedure, then that procedure can inspect the update value to ensure that it is a positive number, not too big or too small, etc. By validating that access procedure, one can then ensure that the validity (but not of course the correctness) of the salary fields will be maintained, no matter which program actually may call for the updating.

Finally, if all routines which require exclusive access to the date are coded as such access procedures, then one can once and for all be assured that no simultaneous updates will be attempted, thereby relieving the referencing programs of responsibility for issuing locks, etc.

Functional data structures have two disadvantages:

i) every field reference requires a procedure call; and

ii) the initial programming effort (when setting up the data structures) is greater.

I feel that the increase in flexibility (for re-organizing the data) and in reliability are quite sufficient to justify the increased execution cost, and that the programming simplicity after the initial set-up of such data structures would easily justify the higher initial programming cost.

III. The Suitability of On-Units for Data Base Error Handling

PL/I On-conditions provide the facility for a program to specify a routine which should be given control when error or exception conditions subsequently arise. My primary concern is that the ON-unit is specified by, and located within, the executing program, and not with details as to whether the ON-units are static or dynamic, or whether they are invoked by a CALL or a GOTO, etc.

ON-conditions can be classified into those for exceptions, such as ENDFILE, those for fixable errors, such as some CONVERSION conditions, and those for unfixable error such as wild branches or bad data.

There are a variety of acceptable ways of handline ENDFILE-type conditions which do not involve ON-units. I tend to prefer having READ return a null-valued record pointer, which I would test with an explicit IF NULL statement. If, however, implicit branches to endfile labels are desired, I would much prefer to see them specified, as in COBOL, as explicit label arguments in the I-O statements; or, perhaps better, as attributes in the file declaration.

Either such approach keeps the specification of endfile actions closely associated with the specific file, which seems desirable.

The main example of fixing up bad data is PL/1 character input, where blanks are often found in places where zeros are required. An On-unit can then translate the blanks to zeroes, and retry the conversion. I would prefer to see such translations either built-in to the conversion, or specified as an option on the declaration of the specific field.

Thus, I would argue that exception-conditions and fixable errors should be dealt with in the user program, but with much more rudimentary mechanisms than On-units. That leaves unfixable errors.

There are two possible actions for unfixable errors. One is to abort the program, and the other is to skip part of it and carry on. The latter action is appropriate for iterative type programs, where, for example, when one input record is found to have unfixable errors it is possible to record that fact and go on to the next input record.

In abort conditions there are two actions which are required: one is to provide a diagnostic dump of the program, and that can always be implicit; two, is to determine what effect on the data base the program failure implies. For example, the program may have been in the middle of updating a file or record, thus leaving it in an inconsistent state.

In skippable error-conditions there are also two actions required: one is to determine where in the program processing should resume, and two is to signal some outside agent (either a human or a recovery program)

to handle the skipped transaction. It would seem appropriate to handle the first of these actions in exactly the same way one elects to handle ENDFILE-type conditions --- either a label-parameter on the I-O operation, or a label attribute on the file declaration.

The remaining question then is how to handle the recovery actions which are required for abort conditions and for skipped errors. Some form of ON-unit specification may indeed be appropriate for specifying such actions. But for reliability, it seems highly desirable that such specifications not be made by the various application programs, but rather be centralized, perhaps associated with scheduler programs and/or with portions or the data-base itself.