SETL Newsletter 90

December 5,1972
J. Schwartz

Preliminary reflections on the use of SETL in a data-base context.

1. Introduction.

A set-theoretic language allows very natural statement of many of the retrieval requests characteristic for the use of data-base systems, and many investigators of such systems currently incline toward a set-theoretic view of their logical structure. However, the use of secondary memory in data base systems presently quite inevitable, and is not clearly consistent with a set-theoretic rather than with a more primitive approach. A language for use in connection with large files requires dictions in aid of certain fundamental optimisations affecting the pattern in which data is moved between secondary and central memory. This newsletter will propose such dictions,giving them a form compatible with our present 'incore' SETL so as to allow SETL to be part of an overall file-manipulation system. The dictions to be proposed give syntactic form to certain semantic concepts and primitive operators. We shall now define these concepts, operators, and dictions, and shall subsequently use them to treat a number of simple prototype problems, thereby illustrating the optimising force of the proposed system.

2. Files.

We define the notion of file as follows. It is a tuple

(1) $\langle$body, length, function-code-mapping$\rangle$

The body of a file is from the logical point of view merely a very long tuple, but one which is in general too long to fit into central memory.

It is of course precisely in view of this last-mentioned circumstance that we introduce the concept of a file, rather than relying exclusively on the logically simpler and more powerful set-theoretic notions amply available in SETL.

The length of a file is merely the length of its body.

Each component of the body of a file is a set of tuples, having the form

(2) { ⟨function-code-1,...⟩, ⟨function-code-2,....⟩,...}.

Each such component is in general small enough for all the data representing it to fit comfortably into central memory or to be handled efficiently using a virtual memory. Thus calculations involving only a single file-body component, and perhaps modifying this component, can be regarded as 'incore' calculations. We shall sometimes refer to a file-body component, as a file item.

The function codes appearing in (2) are small integers. One such code flags each of the attributes of a file item stored in the file component indexed by i.

The system user will think of these attributes as being named (by character strings) rather than as being numbered. The name (i.e., character string) to number correspondence which is necessary in view of this last remark is precisely the function-code-mapping appearing as the third component of (1). Suppose, for example, that a file stores the attributes 'name', 'address', 'telephone number' of its items. Then, for this file, the third component of (1) might be

(3) {⟨'name',1⟩, ⟨'address',2⟩ ⟨ 'telephone number',3 ⟩}.

In this case a particular file component might be

(4) { <1,'JOHN DØE'>, <2, '111 MAIN STREET'>,
<3, 8770664>, <3, 4607381> };

this example shows how a multi-valued attribute would be handled.

It should be noted that when file components are stored on secondary memory, the sets and tuples of which they consist (and any further sets, tuples, strings, etc. which are contained therein) are represented in a self-describing external form, allowing unambiguous reconstruction after re-read.

A file in the sense of a triple (1) is always an incore item; though of course the body of the file will normally be stored on secondary memory, only necessary locators being kept incore as part of (1). Each file will belong to the range of a comprehensive system catalog-of-files. The integer n such that catalog-of-files (n) is equal to a given file is called the file number of the given file.

A file index is logically a pair

(5) <fileno,itemno>

consisting of an item number and a file number. The index is only valid if fileno is no more than the size of the system file catalog (and hence designates a file) and if itemno is no more than the length of the file which fileno designates. File indices will be specially flagged, and should therefore be regarded as belonging to a seperate SETL semantic class (type fileitem) existing equally with strings, booleans, sets,etc. File indices will be treated as atomic, and allowed to be components of vectors, members of sets, and values of mappings; in particular, we allow them to be values of file item attributes .

The following basic operations, added to the presently existing SETL operations, support our proposed file system.

a) Creating a file: x = makefile;

this creates a file of length zero with a null body, and having items without any defined attributes.

b) Destroying a file: destroy x;

this deletes a file and erases its body.

c) Defining initial or additional file attributes.

(6) x = name attrof file;

Here 'name' must be a character string, and 'file' a file. The value x returned is a specially flagged pair consisting of the file number of'file' and the function code of the attribute 'name' such an item may be called a file attribute; file attributes constitute yet another atomic semantic class.

If no attribute 'name' of the items of 'file' initially exists, one will be created when (6) is called.

To test for the existence of an attribute, a direct reference to the third component of (1), possibly in the form

(7) if file (3) (name) ne Ω then ...

can be used, of course, it might be useful to have a standard for the boolean condition 'file (3) (name) __ Ω' .

d) Retrieving and setting the value of a file attribute.

Suppose that f is a file attribute and x a file index referencing the same file. Then we write f(x) for the quantity retrieved by the following rule. Let f = ⟨fileno, fatcode⟩, and let x = ⟨fileno, n ⟩. Then

(8) z = (catalog-of-files (fileno)) (n) (fatcode).

Similarly, f {x} is an abbreviation for

(9) x = (catalog-of-files (fileno)) (n) {fatcode},

while such generalised notations as f(x,k) are available as abbreviations for

(10) f = (catalog-of-files (fileno)) (n) (fatcode, k),

etc. These syntactic constructs are also useable in left-hand forms. Thus, for example, f (x) = z is an abbreviation for

(11) (catalog-of-files (fileno)) (n) (fatcode) = z,

etc. If x is a file index and f a file attribute, both referencing the same file, then

$$\langle x, u_1, \ldots, u_n \rangle \ \underline{\text{in}} \ f$$

may be used to add the n-tuple $\langle u_1, \ldots, u_n \rangle$ to the set f {x}. This mechanism is of course useful for adding additional values of multi-valued attributes.

An integer n may be converted into file index of the form (5) by use of the operation

n indexto file

this produces a pair ⟨fileno, n⟩ (cf.(5) in which'fileno' is the file number of 'file'

e) Creating, revising,and nulling file items.

Let s be a set of the form (2), i.e. a set which can be a component of some particular file (this requires that the first component of every tuple of (2) should be a function code valid for the file r). Then we may write

(12) x = s into r;

This modifies r, adding s as a new component(concatenated to the body of r.) The value of x returned is a file index, which of course references this newly added item. If x is a file index (referencing r) we allow the diction

(13) r(x) = s,

which modifies all the information stored in the file body component referenced by x. The value s must be a set, and this set must have the form (2),i.e., must consist of tuples each of which has a first component which is an integer valid as a function code for the file r. The elements of the set s will be checked for validity at the time that the assignment (13) is made.

The special assignment

(14) $$r(x) = \underline{nl};$$

is of course allowed. This recovers the space normally required to store the information associated with the x'th entry of the file x. A 'null entry' is retained in r, so that x remains a valid index to r even after the 'null assignment' (14) is executed.

f) Iterator over files, file former Number of components:

If f is a file, then we allow various dictions of the form

(15) $$(\forall\ x \in f)\ \text{block};$$

which specify that a given process is to be repeated for all the components of f. Iteration is in the serial order of file components, a vital fact which the programmer is encouraged to exploit. More generally, we allow

(16) $$(\forall x \in f \mid\ C(x))\ \text{block};$$

which,as should be clear from the sematics of the corresponding construction in SETL, bypasses each iteration for which the Boolean expression C(x) has the value'false'. Still more generally, we allow compound iterations

(17) $$(\forall\ x \in f,\ u_1 \in s_1(x),\ u_2 \in s_2(x_1u_1),\ldots \mid\ C(x,u_1,\ldots,u_n))\ \text{block};$$

of essentially the same compound form as is familiar from prior material on SETL. However, at most one file is allowed in the sequence of range restrictions occuring in (17). This restriction is imposed for reasons of efficiency. If more than one file were involved in an iterator like(17), very extensive input-output sequences might thoughtlessly be called for. One will normally strive energetically to avoid double iterations over files, for this purpose passing either to the use of presorted files (cf. the discussion of the sort primitive given below) or to the use of procedurally coded special loops. A few examples of such loops are given in a later section.

Each of the iterator dictions (15), (16), and (17) implies a corresponding compound operator diction as well as set-former, existential and universal quantifier dictions. Thus, if f is a file, we allow

(18)
```
                [max: x ∈ f] salary(x);
            [+: x ∈ f | dept(x) eq 13] salary(x);
        ∃x ∈ f, y ∈ others | dept(x) eq dept (y)
                and salary (x) eq (salary(y) + 1000)
    if  ∀x ∈ f | salary (x) lt 10000 then ....
        {x ∈ f | salary (x) gt 40000}
```

etc. As generally in SETL, sequences of range restrictions are allowed in expressions of the form (18); we continue to impose the restriction that at most one of these range restrictions can imply iteration over a file.

Files being large, we will generally wish to extend iterations over as small a part of a file as possible. To make this convenient, we allow starting and 'while' modifiers to be attached to a file iterator. A starting modifier, if present, defines the initial file component from which an iteration is to begin (if no such modifier is present, iteration over a file begins with the first component of the file.) The syntax of a starting modifier is as follows:

(19) (∀ x ∈ f starting xfirst...),

Here, xfirst must be a variable (or, more generally, an expression) whose value is an index to the file f; in the iteration (19) the initial value of x is xfirst, and iteration continues onward from this component.

A 'while' modifier, if present, states a boolean condition which must be true if iteration is to continue. If a while modifier is present in an iterator and has the value false, the iteration is terminated. The syntax of a while modifier is as follows:

(20) (∀ x ∈ f starting xfirst while C(x)...)

Modified iterators of the form (20) are allowed to appear in such compounds as

(21) (∀ x ∈ f starting xfirst while C(x),yes | D(x,y))

etc., under restrictions which will readily be deduced from the preceeding discussion. Modified iterator-over-files are also allowed in compound operators, set formers, and in existential and universal quantifiers. Thus, for f a file, we allow

(22) [max: x∈f starting xfirst while name(x) eg a] salary(x);
∃ x∈f starting xfirst while dept(x) eq 13,y ∈others |
salary (x) ge (salary (y) + 1000)
{x ∈ f starting xfirst while dept(x) eq 13 |
salary(x) gt 40000},

etc.

In addition to the set former construct of standard SETL, we provide a file former. The value of a file former expression is a file. The syntax of a file former is

(23) {:attribute-name$_1$: ⟨expression$_1$⟩ :attribute-name$_2$: ⟨expression$_2$⟩ :...
:attribute-name$_k$: ⟨expression$_k$⟩ , ⟨iterator⟩ | ⟨condition⟩ }.

An example of the use of a file former is

(24) subfile = {:'name':name (m):'salary':salary(m), m∈ personell
| dept (m) eq 13 and salary (m) le 10000};

In (23), attribute-name$_j$ must be a SETL variable whose value is a string; each such attribute-name value comes to be an attribute of the file created by (23), precisely as if

(25) attribute-name$_j$ = attribute-name$_j$ attrof file;

((6)) had been executed. The semantic significance of (23) is defined by agreeing that the file formed by (23) is identical with that formed by the following procedure.

(26)
```
file = makefile;

attribute-name1 = attribute-name1 attrof file;
                    ...
attribute-namek = attribute-namek attrof file;
 (iterator) newitem = nl into file;
  attribute-name1 (newitem) = expression1;
                    ...
attribute-namek (newitem) = expressionk; end;/ * of iterator scope */
```

When several files must be processed 'against each other' we will normally wish to sort each of these files into a desirable order (cf. below) and to process them using some specially contrived, handwritten loop. To faulitate this, the function

(27) fnext x

is provided. If x is a file index, the value of (27) is the index of the next file component after x.

The length of a file is the number of its components. As might be expected, this is evaluated by the expression

(28) # file.

g. A sorting primitive. Sorting is a process of concentrated (and optimised) regrouping which brings specified data items into significant relationships of physical and sequential proximity. In the SETL file system we propose, sorting is accomplished by writing

(29) newfile = compareop sort file;

In (29), 'compareop' is a binary, boolean valued function which applies to a pair of file indices. The value of compareop(a,b) is to be true if and only if component a is to preceed component b in the sorted order desired. The sorting process invokved by (29) will make optimal use of the secondary storage device on which files are maintained.

3. Some examples.

We will now use the dictions proposed in sections 2 and 3 to program various typical retrievals. For the following examples, we suppose that a large 'personell' file is given, and that (at least) the following attributes of 'file' items are stored in the file:

name:	a character string
department:	an integer
manager:	an index to the personell file
accumtaxes:	taxes paid to date (say, in fiscal year)

a) Find all employees named 'a' who work in a department also containing an employee named 'b'.

```
bdepts ← {dept (m),mε personell | name (m) eq b } ;
print    {: val: < a,m, dept (m) > , mε personell | dept (m) ε bdepts
                                    and name (m) eq a } .
```

b) Find all employees named 'a' who work in a department containing more than 25 employees.

```
    count = nl;
( ∀mε personell) count (dept(m))= count (dept(m)) orm 0  +1;;
print {:val: <a,m,dept (m) >,  mε personell
                        | name (m) eq a and count dept(m)) ge 25}
```

In the preceeding x orm y is a function returning the value

if x ne Ω then x else y.

c) Example (a) using an alphabetised index leading from employee names to their indentifying indices in the personell file. This index is assumed to constitute a file called 'index'. In the following code, we assume a routine 'firstn', essentially of search type (though it may even be assisted by having 'condensed version' of 'index' available, possibly even incore.), which locates the first item in 'index' containing a given name.

```
firstb = firstn (b,index);
bdepts = {dept (item (x),  x ϵ index starting firstb
                           while name (x) eq b };
print {:val: <a,item (x),dept(item(x))>,
    xϵ index starting firstn (a,index) while name (x) eq a};
```

Note that the 'index' file assumed in the above could be produced from the personell file by executing the following code:

```
index = namecompare sort  {:name: name(m) :item: m, mϵ personell};
```

The binary boolean function 'namecompare' used in this code is defined in term of a string primitive alphbigger as follows:

```
define namecompare (ma, mb):
return name (ma) alphigger name (mb);
end namecompare;
```

d) Using the same index assumed in (c), find all employees named 'a' who work in a department containing at least one other employee of the same name.

```
count = nl;
(∀xϵ index starting (firstn (a,index) is start)while name (x) eq a)
count   (dept(x)) = count (dept(x)) ormo   + 1 ;;
print {:val: ⟨a,item (m),dept(item(m))⟩,
                    mϵ index starting  start while name item(m) eq  a
                          | count (dept(item(m))) eq  2 };
```

e) Given a file of taxes paid in current period (by employee number, i.e., the index of an employee in the personell file) update the 'accumtaxes' entries in the personell file. We assume that not every employee occurs in the'file of taxes paid', and also that this file can contain several entries for a particular employee. A programmed loop is used.

```
/* sort the file of taxes paid in current period, putting
it into order of increasing employee number */

curtaxfile = comparenumb  sort curtaxfile;
     /* the binary operator 'comparenumb' returns true
          if two items of curtaxfile are in the relative order
              of employee number */

taxitem = 1        ;

( ∀ x ϵ personell )

(While  ( employeeno (taxitem) is curemp) le x doing
                    taxitem = nextf taxitem ) if curemp eq x then

accumtaxes (x) = accumtaxes (x) + tax (taxitem);; end while;
                        end ∀x;
```

f) Find all employees who have the same name as precisely two other employees. This becomes quite procedural.

```
index = {:name: (m) :item:m, m∈ personell};
     namecompare sort index;
     namenow = 0; items = nl;
x file = makefile; set = 'set' attrof xfile;
       (∀ x ∈ index)
iftree
                        toomany ?
                         noopn,        namesame ?
                                countup,          exact?
                                          addto + restart,
                                          restart;

toomany: = (# items) gt 3;
noopn:    noop;
namesame: =  name (x) eq namenow;
countup: if (# names) lt  3 then item (x) in items;;
exact: = (# names) eq  3;
restart: items = {item(x) };
addto: y = nl into xfile; set (y) = items;

                                end iftree;

                                end  V;

                    /* now print xfile */

    (∀ x ∈ xfile)  print name (∋ x), x; end  ∀ x_j
```

4. An important optimisation: jamming.

Iteration over an entire file may be a lengthy and expensive process. It is therefore important to detect cases in which several operations which might otherwise require seperate iterations over the same file can be performed together during a single iteration over the file. This optimisation is the analog for files of the procedure of 'loop combination' or 'jamming' by which the contents of several independent do-loops with identical limits are combined into a single loop to save iteration count overhead. Of course, the corresponding file optimisation can have a much greater advantage than would normally be attained by incore loop combination.

In some cases, an optimiser will be able to determine that it is possible to combine several iterations-over-files into a single iteration. It might also be useful to provide dictions allowing such cases to be signalled by the user. For example, some type of syntactic parenthesisition might be used to mark a group of statements all of which were to be performed during one common iteration over a large file.

Since data base systems will normally be used in a timeshared mode, one may wish not only to combine iterations initiated by a given user, but even to collect independent iterations over the same file initiated by several independent users, and to perform them all at once in a single pass over the file. Efficiency may be substantially enhanced by this procedure.

5. Various important problems which this newsletter evades or ignores.

A data retrieval system will normally be used by persons who wish to enter queries and receive information, but who prefer very strongly to avoid all involvement with programming in its normal procedural sense. From this point of view, it must be admitted that far too many procedural details show through in the style of retrieval programming illustrated in the section 3. Comprehensive removal of this objection probably requires a system which can choose approprate search strategies automatically by the use of some type of dynamic optimisation procedure . It is not at all clear how this should be done; at any rate the preceeding proposal merely evades this very important question. In favor of such evasion, it may of course be argued that the creation of an appropriate set of procedural file oriented dictions is a necessary preliminary to the development of file-search optimisation techniques.

Data base systems will normally be used in timeshared mode, and therefore require a comprehensive set of system protections, which ration and control the ability to create files (especially large or premanent files), and to access or modify files or fields within files. Moreover, mechanisms which 'jam' iterations over files initiated by several seperate users into a smaller number of iterations can be quite important to overall system efficiency. Of course, all these issues, together with the dictional problems they raise, are ignored in the present newsletter.

6. The file concept and file-namepulation techniques from a more general point of view .

The progress of hardware technology is rapidly bringing us to a point at which logic can be associated with memory on a for more lavish basis than that to which past experience has made us regard as normal. In the limit, these technological advances might allow a small computer to be associated with every one of a great many small memory blocks out of which a very large computer memory (say, a 10^{24} bit memory) was put together. It is worth assaying the effect that this might have on the file notions reflected in the prior parts of this newsletter. In such an environment both memory size and computing power in the sense of comparisons performed (lavishly available for parallel operations) will be much less significant comstraints than they are at present. The most crucial constraint is apt to be internal communications bandwidth, i.e., access to the master buses which carry information from one computer section to another, or which 'brodcast' information from one computer section to all or many other sections. In such a situation, a central process is obviously the regrouping of information to bring into physical proximity items of information which must iteract. This last remark underscores the importance of sorting in the handling of large amounts of data; as noted previously, sorting is a process of concentrated (and optimised) regrouping which brings specified data items into proximity. From this point of view, the file notion can be seen as a general scheme for imposing a pattern of physical proximity on the manner in which we store values which may very likely interact in particular, the values of various 'attributes' of a single 'object')

. Additionalaly, files have an overall serial order having exploitable implications for physical proximity of data. The extraction or production of new files from old is undertaken partly for the algorithmic significance (as sets) of these new files, but partly also for the proximity implications of the file-build process.

The set-theoretic dictions which SETL embodies may have an interesting degree of appropriateness to the type of 'active memory' computer configuration which we contemplate. Note in particular that a set of file indices could in principle be represented by numerous disjoint subparts, one in each memory section to which a mincomputer was attached. This might allow us to recognise that certain iterations of the form

(∀ x ∈ file) block;

could be carried out in parallel. The parallel evaluation of various associated expressions of set-former, existential, and compound operator type might then also become possible. One might even hope to design an optimiser which could unravel the inter-memoryblock communications implications of a sequence of SETL statements, and atomatically 'precondition' for execution by generating appropriate sorting operations. This line of thought, though optimistic, raises problems deserving of study.