AN ARCHITECTURE FOR HIGH-LEVEL LANGUAGE DATABASE EXTENSIONS

by

C.J.Date

IBM General Products Division
1501 California Avenue, CA 94304

December 1975

Abstract

This paper describes an architecture for a set of database extensions to the existing high-level languages. The scheme described forms an architecture in the sense that it is not based on any particular language: its constructs and functions, or some suitable subset of them, may be mapped into the concrete syntax of a number of distinct languages, among them COBOL and PL/I. The architecture includes both the means for specifying the programmer's view of a database (i.e. for defining the external schema) and the means for manipulating that view. A significant feature is that the programmer is provided with the ability to handle all three of the well-known database structures (relational, hierarchical, network), in a single integrated set of language extensions. Another important aspect is that both record- and set-level operations are provided, again in an integrated fashion. The objectives of the architecture are to show that it is possible for relational, hierarchical and network support to co-exist within a single language, and also, by providing a common framework and treating the three structures in a uniform manner, to shed some new light on the continuing debate on the relative merits of each.

The paper is intended as an informal introduction to the architecture, and to this end includes several illustrative examples which make use of a PL/I-based concrete syntax.

Disclaimer

The architecture described herein is the responsibility of the author alone; no particular endorsement or commitment is implied on the part of his employer.

Reprinted with kind permission from the proceedings of the 1976 ACM SIGMOD International Conference on the Management of Data.

1. INTRODUCTION

This paper describes an architecture for extending the existing high-level languages (COBOL, PL/I, ...) to provide direct support for a wide range of database functions. The scheme described forms an architecture in the sense that the extensions proposed are not tailored to any particular language: the general approach taken in the design has been to define, in a language-independent way, the various types of data structure to be supported, together with appropriate operations on these structures, and the detailed problems of mapping these structures and operations into the concrete syntax of individual languages have been deferred to a later time. It must be emphasized that at the time of writing the architecture definition is still incomplete, so that the paper should be regarded as a status report rather than as a final specification.

An important goal for the architecture was to support all three of the well-known data structure classes (relational, hierarchical, network). An equally important and related aim was to provide this support via a single general-purpose set of extensions rather than via three distinct special-purpose sets. A third significant objective was to provide both record- and set-level operations, again in an integrated fashion. Thus the architecture described herein represents an attempt to show that it is possible for relational, hierarchical and network support to co-exist within a single language. So far as the author is aware it represents also the first attempt to deal with all three structure classes in a uniform and consistent manner; it may therefore also serve to cast some new light on the continuing debate on the relative merits of the three approaches.

2. THE PROGRAMMER'S VIEW OF A DATABASE

Throughout this paper we are concerned with the database as it is perceived by the high-level language programmer. To use the terminology of the ANSI/X3/SPARC Study Group [1,2], we are interested in the external model (described by an external schema) - and not the underlying "conceptual" database (described by the conceptual schema), nor the stored data itself (described by the internal schema). From here on we shall use the term "database" as synonymous with "external model".

From the programmer's point of view the significant features of a database are that it is persistent and shared. By "persistent" we mean, broadly speaking, that the data is already in existence at the start of program execution and that it continues to exist after the program has terminated; by "shared" we mean that the programmer must in general be aware that other programs may be referencing and updating the data while his own program is using it. In other respects a database can be considered as simply a new kind of data aggregate. The question arises, how should such an aggregate be presented to the programmer?

Strictly speaking this question is purely syntactic, not architectural: there are obviously several ways of representing a database in terms of concrete syntax. However, since it is axiomatic that the entire syntax will be crucially dependent on the particular

representation chosen, and since in the PL/I-based syntax used in this paper we have selected a slightly unobvious approach, we choose to discuss the issue briefly here. In this syntax a database is represented, not as some new kind of input/output file, but instead simply as part of the program's directly addressable storage area - rather like an array, a queue, or any other "in-core" aggregate - and the programmer is allowed to operate on database data in situ ("direct reference"). Explicit I/O operations are specifically not required. (There is an obvious analogy here with virtual storage systems, in which the programmer is given the illusion of being able to address a large storage area directly and the system handles the necessary I/O operations to make the illusion real.) We present below some reasons for making such a choice.

° The basic point is simply that data in a database is in the system, and from the programmer's point of view it should not be necessary to move it from one place to another in order to process it - he should be able to reference it directly, just as he does "ordinary" (non-persistent, non-shared) or "local" data. A comparative uniformity of reference for local and global data is a great simplifying factor for the user.

° Direct reference automatically provides a great deal of function within existing language. For example, in PL/I, the existing power of operational expressions is immediately available for database data. Thus the database language extensions can be kept to a comparatively manageable size - it is not necessary to have one set of language for database data and one for ordinary data. Retrieval, for example, can be handled by means of an ordinary assignment statement in which the source is a variable in the database.

° As an extension of the previous point, direct reference allows operations of the form

```
X=Y;
```

where X and Y are both references to objects in a database. (Such an operation would require two steps if READ and REWRITE statements were involved.)

° A more specific example:

```
EMPLOYEE.SALARY=EMPLOYEE.SALARY+500;
```

EMPLOYEE is a reference to some database record, SALARY is a field in this record. This example shows how the "update but not see" function - sometimes stated as a security requirement - could be handled in the proposed language.

° The foregoing example also shows that field-level access (a major requirement) can be incorporated very naturally into the direct reference language.

° There is an important semantic distinction between assignment operations and READ/WRITE operations, at least as they apply to source/sink devices, which has been obscured in the past by the fact that READ/WRITE operations have also been used for storage devices such as disks. Typically, once a piece of data has been accessed by means of an input operation, it cannot be accessed again (think of a message from a terminal). An assignment operation, on the

other hand, may be repeated indefinitely and will normally produce the same result every time. A database retrieval operation will also normally produce the same result every time. Similar remarks apply to output: in particular, once a piece of data - say an output message - has been transmitted, it is generally not possible to access it again, whereas it is generally possible to re-access a piece of data after it has been placed in the database. The direct reference proposal is in line with these semantic distinctions. (As a corollary, I/O statements, not direct reference, should be used for source/sink access. This should not be construed to mean that a program's source/sink data cannot be held on a storage or "database" device.)

° The database language extensions also require the introduction of relations [6] as a new kind of local (non-persistent, non-shared) data aggregate. (Relations may also exist in a database.) A local relation may be used, for example, to contain a set of records which are derived in some way (e.g., via a projection operation [6]) from records in the database. It is clearly desirable not to have to use READ/REWRITE statements to access an aggregate which is purely local to the program (a local relation). It is also clearly desirable to be able to access relations uniformly regardless of whether they are local or global (i.e., in a database).

° Direct reference to aggregates (as opposed to individual records) permits the specification of operational expressions whose value is another aggregate of the same type: the important property of closure. To illustrate this point we consider the analogy with simple numeric data. If A, B, C, D are numeric data items, it is the property of closure (under arithmetic operations) which tells us that the value of A+B is also numeric, and hence that this value may be multiplied by C - in a nested expression - or may be assigned to D. Returning to the database language, it is the closure property which allows us to write (for example) an assignment statement setting relation X to contain some join [6] of relations Y and Z, or to write a nested expression defining some projection of such a join. Extending the traditional READ statement would not provide the closure property. This would be true even if the READ operation were made powerful enough to retrieve an entire set of derived records - the essence of READ is that it copies data out of a "file" into something which is not a "file".

The programmer, then, is presented with a view of the database as a storage area containing a large amount of data in the form of records. In addition he is given the ability to reference any record he likes by means of a record reference expression (we assume here that he is authorized to access the record concerned). In principle he could make repeated reference to the same record by merely repeating the same expression - although there would be problems over the effect of update operations, particularly if the updates originated in a concurrent program. In practice, therefore, the normal procedure is for the first reference to a record to be in some cursor-setting operation; the effect of such an operation is to set a nominated cursor to point to the record located by evaluating the specified record reference, and subsequent references to the record are made via the cursor. An example will make this clearer.

```
FIND UNIQUE(EMPLOYEE WHERE EMPLOYEE.EMP#='562170') SET(E);
PUT SKIP LIST(E->EMPLOYEE);
E->EMPLOYEE.SALARY=E->EMPLOYEE.SALARY+500;
```

The first of these three statements locates a particular employee record and sets the cursor E to point to it. The PUT statement then prints this record, and the next statement adds 500 to the SALARY field in this record. In the second and third statements E is being used as a cursor qualifier (cf. pointer qualification in current PL/I). Cursor qualification, like pointer qualification, may frequently be implicit, as subsequent examples will show.

The cursor concept is more important than the above rather trivial example might suggest, however. The most significant point about it is that once a cursor has been set to point to a record, that record becomes locked (unless it was so already), and it remains locked at least until such time as the cursor is set to a new value. Thus in the foregoing example the second and third statements are guaranteed to refer to the same record, and this record is guaranteed not to have been altered by a concurrent program. Neither of these guarantees could have been made if the entire record reference expression appeared (and was therefore re-evaluated) in each of the two accessing statements.

Space precludes detailed discussion of locking in this paper. In general, however, the locking features of the language are as proposed by Engles [13]. Several other reasons for the provision of cursors will also be found in [13].

3.THE APPROACH TO COMMONALITY

As explained in section 1, an objective for the proposed language extensions is commonality of language across the three data structure classes. The approach taken to attaining this objective is based on the realization that, in essence, a hierarchy is merely a special case of a network - a network in which each child record has exactly one parent - and a relation is simply a special case of a hierarchy - a hierarchy consisting of a root only. However, this intuitive statement is far too vague to be useful other than as a broad indication of direction; let us immediately make it more precise by considering each of the three data-structure classes in turn and defining in each case the data constructs that the language extensions will permit.

° In the relational case the only data construct is the relation [6]. Relations are perceived as having an ordering but such ordering cannot be used to carry information "essentially" [10]. (A construct is "essential" if it carries information that would be lost if the construct were removed. Thus, for example, ordering based on field-values is "inessential" - no information is actually lost if the records are shuffled into a different sequence - whereas ordering based on, say, time of arrival is "essential".)

° In the network case there are basically two permitted data constructs, the record-type and the fan set (also called DBTG set [3] or owner-coupled set [10]). Fan sets are usually thought of as coming in two varieties, "singular" and "multiple"; in actuality, however, the two constructs behave very differently; for most of this paper we shall ignore the "singular" case and reserve the term "fan set" for the "multiple" case.

Both record-types and fan sets may be used to carry information essentially. In this context the record-type may be likened to the relation (we shall make this statement more precise later). Fan sets are used to represent certain associations between record-types, associations which in the relational case would be represented by means of a common domain in the record-types concerned. For example, see Fig. 1.

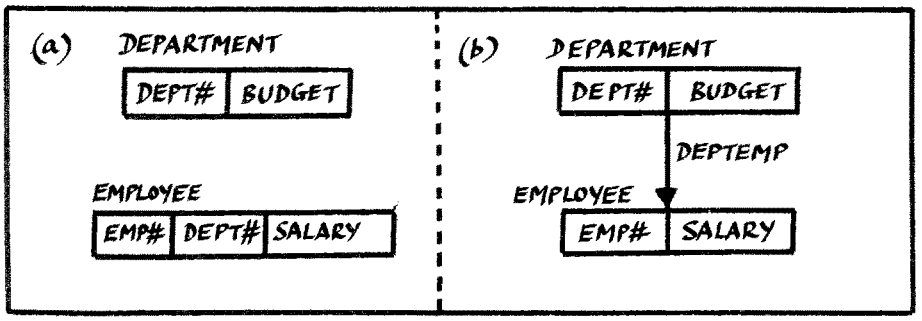

Figure 1: (a) relational representation, and
(b) network representation,
of a department-employee structure

It must be emphasized, however, that in general not all such associations are represented by means of fan sets. For example, in Fig. 1(b), all employees having the same department are associated via the fan set DEPTEMP; but all employees having the same salary are associated, not via a fan set, but via equality of SALARY values in the EMPLOYEE record-type. This latter method of representing associations is the only method available in the relational case. It follows that a network-handling language should be capable of exploiting both methods of representing associations, whereas a relation-handling language need only be capable of handling the second method.

° A hierarchy is merely a special case of a network in which each record-type except one (the "root") is a child in exactly one fan set; the root is not a child in any fan set. The fan sets are conventionally un-named, since no ambiguity can result.

The definitions just given, while capturing the most significant features of the three structure classes, do not cater for certain features of hierarchies and networks as defined in some systems today, specifically IMS [14] and DBTG [3]. The features in question are ones which it is widely felt have no place in a controlled database environment [5,10,12,15,17], though it must be admitted that there is a certain amount of contention on this point. Space does not permit the arguments and counter-arguments to be repeated here. However, we state briefly those characteristics of hierarchical/network structures as described above which distinguish them from the corresponding structures in IMS and DBTG.

° No two records of a given record-type may contain exactly the same values, field for field, and participate as children in exactly the same fans of the same regular fan sets. (A fan set is regular if its children are both NONADOPTABLE and NONORPHANABLE; see section 4 below for an explanation of these terms.)

- Records cannot contain repeating groups.
- Fan sets cannot contain more than one type of child.
- Areas [3]/realms [4] are not supported.
- No ordering is defined between records of different types.
- Ordering across records of the same type cannot be "essential".

Most of these constraints have been imposed on the grounds of intellectual manageability (any one of them could be dropped if sufficient cause were shown, but only at the cost, in each case, of additional language extension). To return to the main argument: if the definitions given earlier for relation, hierarchy and network are accepted, it can be seen that - as stated at the beginning of the section - a relation is a special case of a hierarchy, and a hierarchy is a special case of a network. This observation permits us to define a single set of language extensions encompassing all three structure classes. To be more specific:

- the language extensions required for relational declarations are a subset of those required for hierarchical declarations, which are in turn a subset of those required for network declarations;
- the operators required for relational processing are a subset of those required for hierarchical processing, which are in turn a subset of those required for network processing;
- (for a given operation, as applicable) the operands required for relations are a subset of those required for hierarchies, which are in turn a subset of those required for networks.

The database language extensions thus have an "onion-layer" structure, as illustrated in Fig. 2.

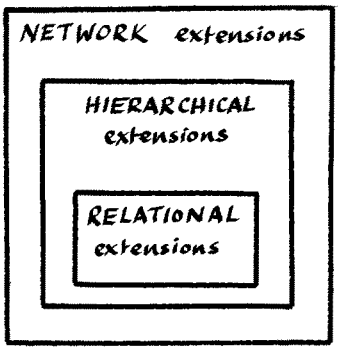

Figure 2: the onion-layer language

4. DECLARATIVE LANGUAGE

As explained in section 2, we are concerned with what is referred to in [2] as an external schema. Although external schemas may in practice be written separately from the program(s) using them, possibly even in a different language [2], a version of the relevant schema must be available to the programmer when he writes his program and to the compiler when the program is compiled; and it is this version which concerns us here. This version of the schema is conceptually part of the source program, and it must therefore be expressed in high-level language terms (very likely it will be INCLUDEd from an appropriate library). Henceforth we shall take "external schema" to refer to this high-level language version.

We are assuming, incidentally, that at some point (normally prior to execution) the object form of the external schema will be bound to the conceptual schema, and also at some point (normally at execution time) the version of the external schema used at compilation time will be checked to match the version currently known to the system. We shall not consider these processes in any detail in this paper.

The basic purpose of the external schema is to define the programmer's view of the database. To achieve this aim the declarative language extensions should permit the specification of as many of the programmer's assumptions about the database as possible. Such specifications will in turn allow a large number of system checks to be applied, many of them at compilation time, and will generally help to prevent programs from executing under false assumptions (and hence from producing incorrect results). In the design of the declarative language extensions, therefore, the ground-rule was: be as explicit as possible.

We present the declarative language by means of an example (based on an example in [11]). An education database contains information about an in-house company training scheme. For each training course the database contains details of all prerequisite courses for that course and all offerings of that course; and for each offering it contains details of all teachers and all students for that offering. A relational schema for this information is shown in Fig. 3 (all relations in third normal form [9]; primary keys [6] shown by underlining; underlying domains [6] not shown). Figures 4 and 5 show, respectively, a hierarchical and a network schema for the same information (underlying domains again not shown; the concepts of third normal form and primary key are inapplicable). Note that two hierarchies (one of them "root only") are required in Fig. 4 if redundancy is to be avoided. (We assume that all fan sets are "essential"; the handling of inessential fan sets is beyond the scope of this paper.)

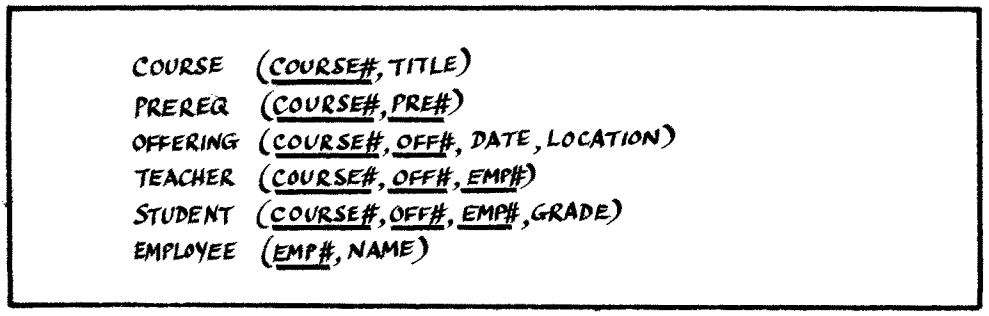

Figure 3: relational schema for the education database

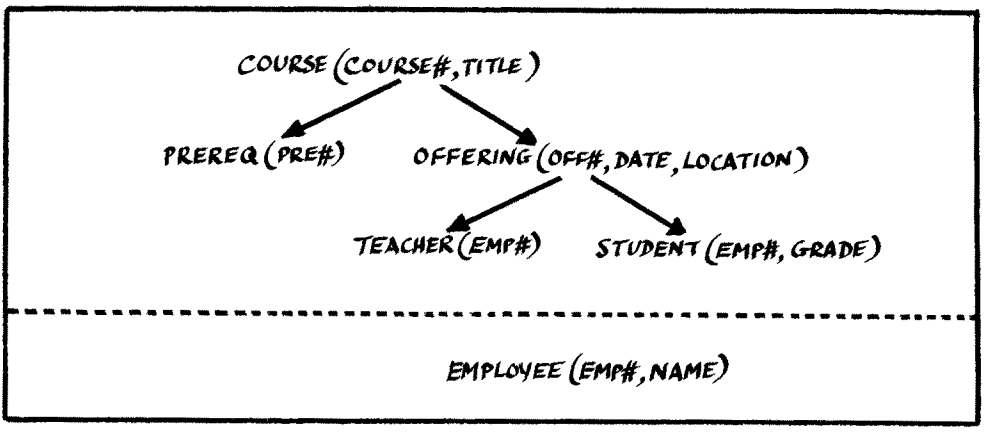

Figure 4: hierarchical schema for the education database

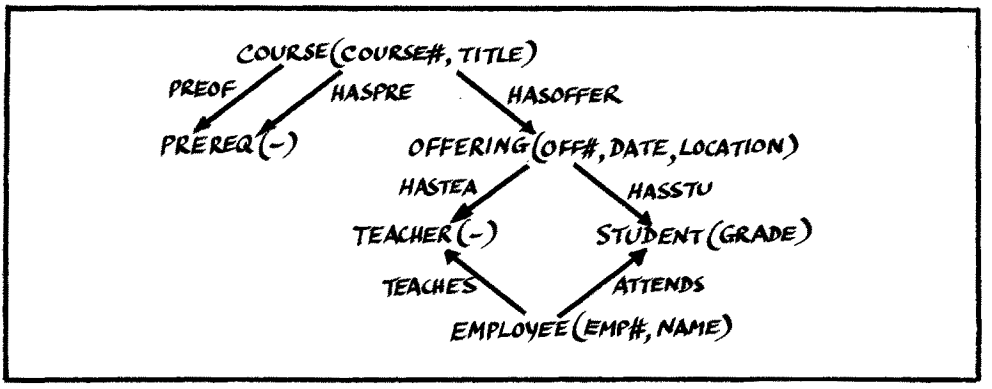

Figure 5: network schema for the education database

In the external schema for this example we first define the record-types by means of appropriate RECORD declarations. (A record is basically the same as a level-one structure in PL/I today, except that the attribute RECORD effectively gives it a new storage class [18].)

```
     DCL  1    COURSE    RECORD(C),
               2    COURSE#   CHAR(3),
               2    TITLE     CHAR(33);

     DCL  1    PREREQ    RECORD(P),
••             2    COURSE#   CHAR(3),
•              2    PRE#      CHAR(3);

     DCL  1    OFFERING  RECORD(O),
••             2    COURSE#   CHAR(3),
               2    OFF#      CHAR(3),
               2    DATE      CHAR(6),
               2    LOCATION  CHAR(12);

     DCL  1    TEACHER   RECORD(T),
••             2    COURSE#   CHAR(3),
••             2    OFF#      CHAR(3),
•              2    EMP#      CHAR(6);

     DCL  1    STUDENT   RECORD(S),
••             2    COURSE#   CHAR(3),
••             2    OFF#      CHAR(3),
•              2    EMP#      CHAR(6),
               2    GRADE     CHAR(1);

     DCL  1    EMPLOYEE  RECORD(E),
               2    EMP#      CHAR(6),
               2    NAME      CHAR(18);
```

Lines marked with a single bullet would be omitted for the network case (Fig. 5); lines marked with a double bullet would be omitted for both hierarchical and network cases (Figs. 4 and 5). These omissions are possible because the relevant information is carried by the hierarchical/network structuring, instead of by fields defined over a common domain as in the relational case. (It would be possible not to omit these fields in the hierarchical and network schemas, but then the fan sets would become inessential. See [10].)

Each record-type has a cursor associated with it. C, for example, is a cursor which will be used to point to individual COURSEs (only). Also, C serves as the default cursor for implicitly-qualified references to COURSE in the processing part of the program. Further cursors may be defined for a record-type by means of explicit cursor declarations - for example:

```
     DCL  C1   CURSOR(COURSE);
```

Every individual cursor is constrained to a single record-type.

Now we can define the database.

```
DCL  EDUCATION DATABASE
     BASESET
     (RECORD(COURSE)      UNIQUE(COURSE#),
      RECORD(PREREQ)      UNIQUE,
      RECORD(OFFERING)    UNIQUE((COURSE#,OFF#)),
      RECORD(TEACHER)     UNIQUE,
      RECORD(STUDENT)     UNIQUE((COURSE#,OFF#,EMP#)),
      RECORD(EMPLOYEE)    UNIQUE(EMP#));
```

This declaration defines the EDUCATION database as containing six "base sets". A base set is an important special case of a construct that recurs throughout database structures, viz. the record set. A record set is simply a homogeneous (single record-type) collection of records, usually but not always with some defined ordering. A base set is that particular (ordered) record set which consists of all occurrences of some given type of record.

In the example the base sets are un-named (note that the names COURSE, PREREQ etc. are the names of the constituent record-types, not of the base sets themselves), but in general record sets may be given a name if desired. With the declarations as shown each of the six base sets is in fact a relation, and this is all that is required for the relational case. For a hierarchical or network structure additional entries will be required (to be explained below), and in the particular example under consideration UNIQUE would be specified for COURSE and EMPLOYEE only - the other four base sets would not be relations in this case. The meaning of UNIQUE is that each record in the base set has a unique value for the indicated field combination.

Any or all of the six base sets could be defined to have a value-controlled ordering: for example, we could specify ORDER(UP COURSE#) for courses (and such an entry would imply UNIQUE(COURSE#) unless NONUNIQUE were specified in the ORDER entry). For simplicity we have assumed default (system-defined) ordering in every case.

To impose a hierarchical or network structure on the database, the declaration must include a FANSET specification as well as the BASESET specification already shown. For the hierarchical structure of Fig. 4 this could be as follows.

```
FANSET
(RECORD(PREREQ)      UNDER(COURSE)      ORDER(UP PRE#),
 RECORD(OFFERING)    UNDER(COURSE)      ORDER(UP OFF#),
 RECORD(TEACHER)     UNDER(OFFERING)    ORDER(UP EMP#),
 RECORD(STUDENT)     UNDER(OFFERING)    ORDER(UP EMP#))
```

The syntax here has been chosen to emphasize the fact that the important thing about a fan is the corresponding set of children, rather than the set of children plus the parent. For example, the scope of a DO-loop - see section 7 - is typically a set of children, not an entire fan. The four fan sets shown above are un-named, though there is no reason why they should not be given a name if desired. Within each fan of each fan set the child records are ordered as indicated. Again each ORDER entry implies a corresponding UNIQUE entry.

For the network structure of Fig. 5 the FANSET entry could be as follows. Notice that here the fan sets have been named, although there is no reason why they should not remain un-named if no name is required. The details of exactly when a name is required are somewhat

complex and will not be discussed in this paper.

```
FANSET
(PREOF     RECORD(PREREQ)    UNDER(COURSE),
 HASPRE    RECORD(PREREQ)    UNDER(COURSE),
 HASOFFER RECORD(OFFERING) UNDER(COURSE) ORDER(UP OFF#),
 HASTEA    RECORD(TEACHER)   UNDER(OFFERING),
 HASSTU    RECORD(STUDENT)   UNDER(OFFERING),
 TEACHES   RECORD(TEACHER)   UNDER(EMPLOYEE),
 ATTENDS   RECORD(STUDENT)   UNDER(EMPLOYEE))
```

However, the FANSET entries as shown (for both hierarchic and network cases) are not complete. For any given fan set, in general, the child record-type may be (a) TRANSFERABLE or not, (b) ADOPTABLE or not, and (c) ORPHANABLE or not, with respect to that fan set. Each fan set declaration should include a specification of the programmer's assumptions with respect to transferability, adoptability and orphanability, so that appropriate compile- and bind-time checks may be made. Transferability applies to both hierarchies and networks; adoptability and orphanability apply to networks only. In the context of essential fan sets (the only kind we are discussing here), a child record-type is TRANSFERABLE if a RECONNECT statement may be used to move an individual child from one fan to another in the fan set; it is ADOPTABLE if a CONNECT statement must be used to perform the initial linking of an individual child to its first fan in the fan set; and it is ORPHANABLE if a DISCONNECT statement may be used to remove an individual child from one fan without at the same time linking it to another fan in the fan set. (Note that adoptability is not the same as "connectability". A better term would be desirable. For those readers familiar with DBTG, adoptability corresponds to MANUAL membership; orphanability corresponds to OPTIONAL membership [3]. There is at present no DBTG equivalent to transferability.)

5.CURSORS AND CURSOR STATES

A cursor is an object whose primary function is to designate some individual record by means of that record's record identifier (RID). Each record has a unique RID. Before explaining the cursor concept in any more detail, it is first necessary to amplify the associated notion of ordered record set, since the operation of setting a cursor usually involves the selection of a record from such a set.

An ordered record set is simply a set of records with a total ordering imposed on them. A relation is one example; a base set is another (base sets are not necessarily relations). The set of children in a single fan is a third example. The set of all children in all fans of a given fan set is an example of a record set which is only partially ordered (note, however, that exactly the same collection of records may constitute another record set which does have a total ordering).

A given record may participate in any number of ordered record sets simultaneously (every record participates in at least one such set, viz. the relevant base set). We may model this situation as follows. An ordered record set S may be thought of as a closed, directed loop. The loop has a unique point of discontinuity called the zero position (P0), which may be thought of as both the beginning and the end of the loop. At any given time the loop holds a set of n+1 objects (n

greater than or equal to zero), located at n+1 discrete positions P0, P1, P2, ..., Pn on the loop. These n+1 positions are such that, with respect to the loop direction, P0 precedes P1, P1 precedes P2, ..., Pn precedes P0. The object at Pi (i in the range 1 to n) is the RID of the record which is ith in the ordering of the record set being modelled. The object at P0 is the "zero RID", i.e. a dummy record identifier which is considered as identifying a fictitious "zero record" (R0), distinct from all real records. Note that R0 appears in every ordered record set. We shall refer to R0 as the zeroth record of such a set, and use 1st, 2nd, 3rd, ... to refer to the 1st, 2nd, 3rd, ... real record of such a set. See Fig. 6.

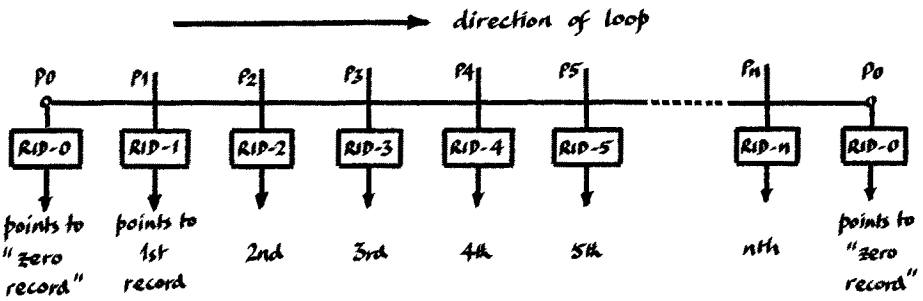

Figure 6: the loop model of an ordered record set

Now let C be a cursor constrained to records of type R (the record-type involved in set S). Then at any given instant C will be either "selecting" or "preselecting" some record of type R (possibly the zero record, which is considered to be of every known record-type).

A cursor that is selecting a given record has the RID of that record as its value. For example, if C has RID-3 as its value, it is selecting the third record in set S. Since records may simultaneously belong to several distinct sets, this record may also be the 10th in set S', the 6th in set S'', and so on. The record concerned may be accessed via a reference of the form C->R, unless it is the zero record: an attempt to access R0 is an addressing error. A cursor which is selecting a record is said to point to that record.

If cursor C currently has the value RID-i (identifying the ith record in S), then it may be stepped forward 1, 2, ... or n-i positions within S to select some other record; however, it cannot be stepped as far as (or past) R0 at the end of the set. Similarly, it may be stepped backward 1, 2, ... or i-1 positions within S to select some other record; however, it cannot be stepped as far as (or past) R0 at the beginning of the set. In general a cursor which is selecting a record can be set to select another record by stepping it any number of positions (providing the zero position is not reached), in either direction, in any ordered set containing the record it is currently selecting.

Now suppose cursor C is currently pointing to record Ri (i nonzero) in set S, and suppose that Ri is the object of an operation which removes it from S or changes its position within S. Examples of such operations are DESTROY; DISCONNECT; RECONNECT; and ordinary field update operations, if the field concerned is an ordering field. In order that the programmer may retain his position within S for use in subsequent operations, DISCONNECT and similar statements allow the specification of an ADV (advance) option. Any cursor named in the ADV option is advanced to preselect the record (possibly R0) which immediately follows Ri's old position in the ordering of the set concerned (which is specified either by the operation or in the ADV option itself). A cursor that is preselecting a given record has the RID of that record, together with a flag, as its value; the flag is set to show that the record is being preselected, not selected.

An attempt to access a record via a reference of the form C->R will fail (addressing error) if C is in the preselecting state. However, such a cursor may be stepped either forward or backward, just as if it were actually selecting the record, in any set containing the preselected record; the only differences are that (a) such operations turn the flag off, and (b) a request to step the cursor m places forward will actually step it forward only m-1 places. Thus if C has been advanced to preselect Ri's successor in set S, a request to step it forward one position (in S) will set it to select Ri's successor (in S); a request to step it backward one position (in S) will set it to select Ri's predecessor (in S).

Any cursor which pointed to Ri before the operation not nominated in the ADV option remains unchanged (i.e. it now points to Ri in its new position), except in the case of DESTROY, where such a cursor is automatically advanced to preselect Ri's successor in the appropriate base set.

6.FREE CURSORS

The importance of the record set construct should be clear from the previous section. The language must allow the programmer to write expressions to denote record sets - not only pre-declared ones, such as the set of all EMPLOYEE record occurrences, but also dynamically generated ones, such as that subset of EMPLOYEE record occurrences where the jobname is 'PROGRAMMER' (for the sake of the example we suppose that EMPLOYEE records include a JOBNAME field).

Formally, we need to be able to write set-defining expressions such as:

X WHERE X is an EMPLOYEE record & X.JOBNAME='PROGRAMMER'

Conceptually this may be thought of as follows: "out of the entire universe of objects X, select just those which satisfy the predicate (the condition following the WHERE)"; and the predicate says "we are interested only in those objects X which are EMPLOYEE records, and the JOBNAME field in these records must have the value PROGRAMMER". X here is a "free variable". In the language we represent free variables by means of cursor-qualified record references - for example:

```
E->EMPLOYEE WHERE E->EMPLOYEE.JOBNAME='PROGRAMMER'
```

(the condition "X is an EMPLOYEE record" is implied). If E is the default cursor for EMPLOYEE, and if JOBNAME is unambiguous, this expression can be simplified to:

```
EMPLOYEE WHERE JOBNAME='PROGRAMMER'
```

(a more intuitively obvious form). Note that a cursor-qualified reference that represents a free variable is semantically different from a syntactically identical reference in other contexts. Specifically, it is not a reference to the record selected by the indicated cursor. In fact the current value of the cursor is irrelevant (and is not changed); the cursor merely serves as a syntactic device to link references to fields in the predicate to the free variable. As another example, the expression:

```
E->EMPLOYEE WHERE E->EMPLOYEE.JOBNAME = F->EMPLOYEE.JOBNAME
```

is a reference to the set of employees having the same jobname as the employee selected by cursor F. Cursor F here is performing its normal selection function; cursor E is a "free cursor" and is being used as part of a free variable reference.

To simplify repeated references to the same set of records, it is possible to CONNECT the records resulting from the evaluation of a set-defining expression to an empty, named "window set". This effectively provides a method of giving a name to a dynamically generated set of records. Space precludes detailed discussion of window sets in this paper.

7. MANIPULATIVE LANGUAGE

In this section we present some of the major features of the manipulative language extensions, primarily by means of examples.

We start with the most important cursor-setting operation, viz. the FIND statement - syntax:

```
FIND record-reference  SET(cursor-name) ;
```

Example (using the relational schema of Fig. 3):

```
FIND FIRST(PREREQ WHERE PREREQ.PRE#='322') SET(P);
```

In this example PREREQ WHERE ... is a set reference, i.e. an expression denoting a record set (actually a subset of the PREREQ base set, with an ordering inherited from this base set), and FIRST is a "built-in reference" which selects the first record in this ordered set. Cursor P is set to point to this record. Note that P is also acting as the (implicit) free cursor in the references to PREREQ (because P is the cursor named in the RECORD declaration for PREREQ). The phrase SET(P) could have been omitted: if no SET option appears in a given FIND statement the appropriate default cursor is assumed.

We can now refer to the selected PREREQ record, and to fields within

it, using P as a (possibly implicit) cursor-qualifier.

Examples:

```
LOCALCOURSE#=P->PREREQ.COURSE#;    /*field retrieval */
PREREQ.COURSE#=LOCALCOURSE#;       /*field update    */
LOCALPREREQ=PREREQ;                /*record retrieval*/
PREREQ=LOCALPREREQ;                /*record update   */
IF PREREQ.COURSE#='860' THEN ...
PUT SKIP LIST(PREREQ);
```

The FIND statement may optionally include either a FOUND or a NOTFOUND specification. These options nominate a flag which is to be set if the FIND is successful or unsuccessful (as applicable).

Example (following the previous FIND statement):

```
FIND NEXT(PREREQ WHERE PREREQ.PRE#='322') NOTFOUND(DONE);
```

This statement will step P along to the next PREREQ with PRE# 322. ("Next" here refers to the ordering of the PREREQ base set.) If no more such PREREQs exist the flag DONE will be set. Thus to loop through all such PREREQs:

```
     P=ZERO; /*set P to point to record zero*/
LOOP:FIND NEXT(PREREQ WHERE PREREQ.PRE#='322') NOTFOUND(DONE);
     IF DONE THEN GO TO EXIT;
      .....
     GO TO LOOP;
EXIT: .....
```

Example (the same loop, using a DO statement):

```
DO PREREQ WHERE PREREQ.PRE#='322' SET(P);
     .....
END;
```

This code is defined to be semantically identical to the loop code given above (except that no explicit flag-setting occurs). Again the SET option may be omitted, in which case the cursor assumed is that defined in the relevant RECORD declaration.

Two more important built-in references are UNIQUE and PARENT. PARENT is applicable to hierarchies and networks only.

Example (UNIQUE):

```
FIND UNIQUE(COURSE WHERE COURSE.COURSE#='860');
```

UNIQUE selects the single record in a single-record set; it is an error if the set does not contain exactly one record.

Example (PARENT, using the hierarchical schema of Fig. 4):

```
FIND FIRST(STUDENT WHERE .....);
FIND PARENT(STUDENT);
```

Using the network schema of Fig. 5:

```
FIND FIRST(STUDENT WHERE .....);
FIND PARENT(STUDENT,HASSTU);
```

In both these examples S is set to point at the first STUDENT (with respect to the ordering of the STUDENT base set) satisfying some criterion, and O is set to point at its parent OFFERING. In the network case a fan set name is required within the PARENT reference, since otherwise it would be ambiguous.

For clarity we allow the built-in reference

```
PARENT(X)
```

to be written

```
Y OVER X
```

wh is the declared parent record-type for X.
Similarly,

```
PARENT(PARENT(X))
```

may be written

```
Z OVER X
```

where Z is the declared parent of Y, and so on. (Similar simplifications may also be made where X is a child in more than one fan set - i.e. where Y is not unique - under most circumstances, but the details are beyond the scope of this paper.)

Example (using the hierarchical schema, to loop through all children of a fan):

```
FIND UNIQUE(COURSE WHERE COURSE.COURSE#='860');
DO OFFERING WHERE SAME(PARENT(OFFERING),COURSE);
   .....
END;
```

SAME(record-reference,record-reference) is a built-in function whose value is true if the two record references denote the same record, false otherwise. Record set references of the form

```
X WHERE SAME(PARENT(X),Y)
```

are so common that the following is permitted as a shorthand:

```
X UNDER Y
```

Similarly,

```
X WHERE SAME(PARENT(PARENT(X)),Z)
```

can be abbreviated to:

```
X UNDER Z
```

and so on. (Again, similar simplifications may usually be made where X is a child in more than one fan set, but no further details will be given here.)

The DO statement above can thus be reduced to:

```
DO OFFERING UNDER COURSE;
```

The order in which the OFFERINGs are processed in this example is determined by the order of the OFFERING base set.

Example (as previous example, but using the network schema):

```
FIND UNIQUE(COURSE WHERE COURSE.COURSE#='860');
DO OFFERING IN HASOFFER WHERE SAME(PARENT(OFFERING,HASOFFER),COURSE);
    .....
END;
```

In the DO statement OFFERING IN HASOFFER defines a partially ordered record set - the set of all OFFERINGs appearing as children in the fan set HASOFFER - but the WHERE clause restricts this to a totally ordered subset (it would be an error if it did not). Default rules similar to those already mentioned allow expressions of the form

```
X IN F WHERE SAME(PARENT(X,F),Y)
```

to be abbreviated (in most cases) to

```
X UNDER Y
```

Thus, as in the hierarchical case, the DO statement above may be reduced to

```
DO OFFERING UNDER COURSE;
```

So far all examples have been basically one-record-at-a-time. We now present an example dealing with an entire record set as a single operand.

Problem: retrieve all legal course-number/teacher-name pairs
(a pair is "legal" if a teacher with the indicated name teaches at least one offering of the indicated course).

We need a local relation to hold the result.

```
DCL 1 CT RECORD(CTC),
      2 COURSE# CHAR(3),
      2 NAME    CHAR(18);

DCL CTR BASESET(RECORD(CT) UNIQUE);
```

Using the relational schema:

```
CTR=(TEACHER.COURSE#,EMPLOYEE.NAME)
        WHERE EMPLOYEE.EMP#=TEACHER.EMP#;
```

Using the hierarchical schema:

```
CTR=(COURSE.COURSE#,EMPLOYEE.NAME)
        WHERE EMPLOYEE.EMP# EQSOME
              (TEACHER.EMP# WHERE
                 SAME(PARENT(PARENT(TEACHER)),COURSE);
```

The parenthesized expression "(TEACHER.EMP# ...)" can be unambiguously abbreviated to

```
          (TEACHER.EMP# UNDER COURSE)
```

The value of this expression for a given course is the corresponding set of teacher employee numbers. The expression "X EQSOME S" has the value true if the set S contains a member with value equal to that of X.

Using the network schema:

```
CTR=(COURSE.COURSE#,EMPLOYEE.NAME)
        WHERE EMPLOYEE EQSOME
             (PARENT(TEACHER WHERE
                SAME(PARENT(PARENT(TEACHER,HASTEA),HASOFFER),COURSE),
                                                    TEACHES));
```

Notice the use of PARENT with an aggregate argument (and hence acting as an aggregate reference). Once again we may unambiguously simplify the expression following EQSOME, this time to:

```
                (EMPLOYEE OVER (TEACHER UNDER COURSE))
```

Since EMPLOYEE also appears (with a different denotation) before the EQSOME, however, the two references must have different free cursors to distinguish them, and hence at least one must include an explicit free cursor in the complete simplified expression - for example:

```
CTR=(COURSE.COURSE#,EMPLOYEE.NAME)
        WHERE EMPLOYEE EQSOME
             (F->EMPLOYEE OVER (TEACHER UNDER COURSE));
```

(where F, like E, is a cursor over EMPLOYEEs).

In all three cases the result of the retrieval is in local relation CTR; records in this result can be accessed via record references and cursors, just like records in a database.

Example (updating): change the date and location of offering number 3 of course 860 to 21st July 1975, Helsinki.

Using the relational schema:

```
FIND UNIQUE(OFFERING WHERE OFFERING.COURSE#='860'
                         & OFFERING.OFF#   ='  3');
DATE='750721';
LOCATION='HELSINKI';
```

If a program intends to update several fields of a record, as in this example, it may be preferable to retrieve a copy of the record, make the changes to the copy, and then update the original record by means of an assignment from the copy. For example, the sequence:

```
FIND UNIQUE(OFFERING etc.);
TEMPOFFERING=OFFERING;
TEMPOFFERING.DATE='750721';
TEMPOFFERING.LOCATION='HELSINKI';
OFFERING=TEMPOFFERING;
```

may be preferable to the previous code. The advantage is basically one of performance: it is probably more efficient to perform a single (record) update than two or more separate (field) updates. This is particularly likely to be so if the implementation applies integrity checks on each update; moreover, "spurious" errors may be signalled if at some given time the program has updated one field and not another. On the other hand some programs may not be bothered by such problems, and for them code sequences such as the first of the two above are simpler and more natural.

Example (creating a new record): local structure STUDENTAREA contains a new student record to be inserted into the database.

Using the relational schema:

```
CREATE STUDENT FROM STUDENTAREA;
```

Using the hierarchical schema:

```
CREATE STUDENT FROM STUDENTAREA
     CONNECT(UNDER OFFERING);
```

(We asume here that cursor O already identifies the OFFERING which is the new student's parent-to-be.)

Using the network schema:

```
CREATE STUDENT FROM STUDENTAREA
     CONNECT(UNDER OFFERING VIA HASSTU,
             UNDER EMPLOYEE VIA ATTENDS);
```

(We assume here that cursors O and E have already been set appropriately. An UNDER entry must be specified for each fan set in which STUDENT is NONADOPTABLE. VIA options are used to indicate the relevant fan sets; they may be omitted if no ambiguity results.)

Example (destroying an existing record):

```
DESTROY OFFERING;
```

We assume here that cursor 0 has been set to identify the record to be destroyed. In general the operand may be any record reference, or any set reference if it is desired to destroy several records at once. The record(s) specified are destroyed, together with all their NONORPHANABLE children. All cursors which previously identified any of these records are advanced within the OFFERING base set.

Fan set operations

The fan set operations are CONNECT, DISCONNECT, RECONNECT. Their syntax is as shown below (no examples will be given). VIA options (and the FROM option in the case of DISCONNECT) may be omitted if no ambiguity results. The ADV option mentioned in Section 5 is not shown.

```
CONNECT record-reference UNDER record-reference [VIA fan-set-name];

RECONNECT record-reference UNDER record-reference [VIA fan-set-name];

DISCONNECT record-reference [FROM fan-set-name];
```

Once again a set reference (rather than a record reference) may appear as the first operand, if it is required to deal with several records at once.

8. SUMMARY

We have presented some major features of the proposed database language extensions. We may summarize the highlights of the proposal as follows.

- All three data structure classes are supported.
- The database is represented as an extension of the program's storage area.
- The programmer can retain multiple explicit positions in the database.
- Record selection (via cursor) and record access (via cursor-qualified reference) are separable functions.
- Record references permit a wide variety of "navigational" operations (UNIQUE, FIRST, PARENT, plus others not discussed in this paper).
- The programmer can deal with entire record sets as single operands.
- Record set expressions have the generality and "completeness" [8] of predicate calculus, without involving predicate calculus notation.

In conclusion we list some other features of the proposal which we do not have room to discuss in this paper.

- Locking [13].
- Error-handling and feedback (on-conditions).
- "Window sets" and associated CONNECT/DISCONNECT operations. A window set is a record set which is wholly contained within some underlying set; it acts as a window into the underlying set, through which some subset of the underlying records may be seen. The records may be viewed in a different sequence through such a window. "Singular fan sets" are handled by this mechanism.
- Built-in functions such as COUNT and TOTAL [7].
- Set comparisons such as SUBSETOF.
- Null data values.
- Fields, records etc. as arguments and parameters.

ACKNOWLEDGEMENTS

The writer is pleased to acknowledge useful discussions and communications with a large number of colleagues, especially Bob Engles, Chris Paradine and John Roskell. Acknowlegements are also due to Rita Summers, Charlie Coleman and Eduardo Fernandez, who independently developed language extensions very similar to the relational portions of the present proposal [19].

REFERENCES

1. C.W.Bachman: "ANSI/X3/SPARC Study Group on Data Base Systems: Summary of Current Work" (January 1974).
2. ANSI/X3/SPARC Study Group on Data Base Systems: Interim Report (8 Feb 1975).
3. Data Base Task Group of CODASYL Programming Language Committee: Final Report (April 1971).
4. Data Base Language Task Group of CODASYL Programming Language Committee: COBOL Data Base Facility Proposal (March 1973).
5. Proceedings of IFIP TC-2 Special Working Conference: A Technical In-Depth Evaluation of the DDL (Namur, Belgium, 13-17 Jan 1975). North-Holland (1975).
6. E.F.Codd: "A Relational Model of Data for Large Shared Data Banks". CACM 13,6 (June 1970).

7. E.F.Codd: "A Data Base Sublanguage Founded on the Relational Calculus". Proc. 1971 ACM SIGFIDET Workshop on Data Description, Access and Control.

8. E.F.Codd: "Relational Completeness of Data Base Sublanguages". In Data Base Systems, Courant Computer Science Symposia Series Vol. 6, Prentice-Hall (1972).

9. E.F.Codd: "Recent Investigations in Relational Data Base Systems". Proc. IFIP Congress 1974, North-Holland.

10. E.F.Codd, C.J.Date: "Interactive Support for Non-Programmers: the Relational and Network Approaches". Proc. 1974 ACM SIGMOD Debate, Ann Arbor, Michigan (May 1974).

11. C.J.Date: "An Introduction to Database Systems". Addison-Wesley (1975).

12. R.W.Engles: "An Analysis of the April 1971 DBTG Report". Proc. 1971 ACM SIGFIDET Workshop on Data Description, Access and Control.

13. R.W.Engles: "Currency and Concurrency in the COBOL Data Base Facility". Proc. IFIP TC-2 Working Conference on Modelling in Data Base Management Systems, Freudenstadt, Germany, January 1976 (to appear).

14. IBM: Information Management System/Virtual Storage General Information Manual (IBM Form No. GH20-1260).

15. G.M.Nijssen: "Data Structuring in DDL and Relational Data Model". In Data Base Management Systems (ed. Klimbie and Koffeman), Proc. IFIP TC-2 Working Conference, Cargese, Corsica 1974 (North-Holland).

16. G.Held, M.Stonebraker: "Networks, Hierarchies and Relations in Data Base Management Systems". Proc. ACM Pacific Conference, San Francisco, April 1975.

17. R.W.Taylor: "Report on IFIP TC-2 Special Working Conference: A Technical In-Depth Evaluation of the DDL (Namur, Belgium, 13-17 Jan 1975)".

18. ANSI/ECMA: BASIS/1-12 (Draft PL/I Standard), July 1974.

19. R.C.Summers, C.D.Coleman and E.B.Fernandez: "A Programming Language Extension for Access to a Shared Data Base." Proc. ACM Pacific Conference, San Francisco, April 1975.