Management/ Data Base Systems
H. Morgan Editor

A Data Definition and Mapping Language

Edgar H. Sibley
University of Michigan
and
Robert W. Taylor
University of Massachusetts

A data definition language is a declarative computer language for specifying data structures. Most data definition languages concentrate on the declaration of logical data structures with little concern for how these structures are physically realized on a computer system. However, the need for data definition languages which describe both the logical and physical aspects of data is increasingly apparent. Such languages will be a key element in the translation of data between computer systems, as well as in advanced data management systems and distributed data bases.

This paper reviews past work in the data definition language for describing both logical and physical aspects of data. Applications of these "generalized" data definition languages are also discussed.

Key Words and Phrases: data definition language, data and storage structure, data translation, data base management systems, file translation

CR Categories: 3.51, 3.70, 3.73, 4.29, 4.82

Research partially sponsored by the Air Force Office of Scientific Research, Air Force Systems Command, USAS, under Grant No. AFOSR-72-2219. Authors' addresses: Edgar H. Sibley, Department of Information Systems Management, University of Maryland, College Park, MD 20742; Robert W. Taylor, Department of Computer Sciences, University of Massachusetts, Amherst, MA 01002.

1. Introduction

From the earliest days of computing, the concept of formats for data input and output has been commonly used. These formats aided the programmer by simplifying his description of the input and output of data. By using logical device-oriented read and write statements, he was further freed from details specific to particular devices. This meant that the programmer needed to know fewer implementational details of specific media. The resulting device-independence, along with "common" and standard languages, makes it feasible to run a given program on various hardware. Unfortunately, the same cannot be said for data. Only in the most restricted cases is it possible to have a program at one installation process data which was created at another. This stems primarily from a lack of explicit definition of the data to be processed.

Furthermore, the need for generalized data base management systems which can communicate either with similar or with different systems, possibly over computer networks with distributed data bases, again demands an explicit definition of the data.

Transferring data between dissimilar computers is not the only problem; a problem also arises when we transfer between hardware lines supplied by the same manufacturer, and sometimes we find substantial changes in storage and access methodology in different versions of the same operating systems.[1]

The need for explicit definition has led several committees to start considering a language and technology for describing "storage structures." If a language can be developed to describe storage formats in sufficient detail, statements in the language can then be used as input to programs which would reorganize a file or transform the file into a format suitable for use by other hardware or software systems.

This paper will review efforts currently under way in describing file or storage structure languages, provide a basis for continuing such work, and give an example of statements in a language which satisfies some of these needs.

2. Related Work

Several professional groups and individual researchers have made preliminary attempts to define languages for describing the structure and storage of data. Although this section is not exhaustive, it is intended to cover the major contributions in the area.

2.1. ANSI X3 Ad Hoc Committee

In October 1968, an ad hoc committee was formed by the ANSI X3 group, under the direction of John Gosden, to define the scope of a data definition language and to recommend to X3 what standard work should be done on data definition languages. The Gosden

report [1] represents a synthesis of key members' views.[2] Although it has been acknowledged by X3, the report has been neither approved nor rejected by ANSI. Gosden proposed that a formally defined data definition language, describing data on both the logical and physical level, would be an appropriate candidate for standardization. Unfortunately, the X3 ad hoc committee disbanded, implying it was premature to standardize since extensive development was needed.

2.2. European Computer Manufacturers Association

Other related standards work on data definition languages was performed by the members of Task Group 2 (TG2) of Technical Committee 15 of the European Computer Manufacturers Association (ECMA/TC15/TG2). They started to define a Data Description Language (DDL) for formats. Their objective was to formulate a concise method of defining the format of a collection of data to facilitate the interchange of standard defined formats.

In their preliminary draft [3], both an informal and a formal syntax and elements of the semantics of a proposed DDL were presented for describing data format specifications. A format specification is "the definition of a collection of character strings, each of which is a data item to which the format may be applied." The DDL includes the basic operations of set theory: concatenation, intersection, and union. Further, by recursively applying the concatenation operations, tree-like format structures (called hierarchy platforms) could be built. The ECMA/TC15 work represents a good start, but unfortunately it is not continuing because ECMA decided that developmental activity was needed before further standardization could be proposed.

2.3. CODASYL Activities

Within the Conference on Data Systems Languages (CODASYL), several data definition language efforts (with distinctly different objectives) emanate from different committees. In April 1971, the Data Base Task Group (DBTG), a group working under the Programming Language Committee, specified a data definition language to enhance the data structure capabilities of programming languages [4]. COBOL was chosen as the first language to be enhanced. The DDL defined by the DBTG is directed toward specifying network or graph-type data structures not currently specifiable in COBOL and other compiled languages. In addition to extending the domain of the data structures, the DBTG proposal goes further and defines a data manipulation language to access or process the data defined by the DDL. Although the language extends the data structures available to a COBOL programmer, there are no statements to define the storage implementation of data. Hence, the storage structure is left up to the desires of the implementor, which negates use of the DDL for data interchange. Much of the work of the DBTG has now been charged to a new Data Definition Language Committee (DDLC), which has no present plans to implement a storage structure language, though it may incorporate this later.

Another effort is directed toward defining data as it has been previously stored by a computing system. The Stored Data Definition and Translation Task Group (SDDTTG) was formed under the aegis of the CODASYL Systems Committee to address a current problem apparent from its study of Generalized Data Base Management Systems (GDBMS): the lack of data transferability or the incompatibility of data. The purpose of the SDDTTG is to develop a method for concisely defining commonly used existing storage structures. Interim reports of its work have been given at the SIGFIDET (Special Interest Group in File Description and Translation) November workshops in 1970 and 1972 [5, 6].

2.4. Other Research

In a recent Ph.D. dissertation at the University of Pennsylvania [7], Diane Smith developed a DDL oriented toward defining data as it exists on various secondary storage media and devices commonly in use. The approach taken is similar, in certain ways, to the approach presented here; we briefly summarize this work.

Smith divides the data definition process into several parts. In one series of language statements, she defines the conceptual record structure and the conceptual file structure of the data. These statements also give encoding characteristics for the record and file structures. Another series of statements defines device characteristics and the hierarchic structures which are presented by the various media. There are also assignments of encoded data to media.

In addition, Smith defines a criterion production system (cps) which can be used to specify when two conceptual record structures can be considered as being related. In this way, Smith provides the ability to specify any relational structure independent of whether it is ultimately represented via chaining, concatenation, or other mechanism. The cps can also be used to specify the set of legal values which a particular data item may take.

The statements of Smith's languages resemble assembly language macro calls in the sense that a number of parameters are collected together, by position, with a single basic data definition statement. As such, the language is difficult to read and to write. On the other hand, it is a highly flexible language.

3. The Need for Data Definition and General Approach

Until recently, the problem of file translation tended to be a personal affair; files were much smaller and had relatively simple data structures. Even in cases where

[1] In one release of a current operating system, Fortran-generated output could not be read using the same format statement in a program compiled on the following release.

[2] Gosden has also discussed these issues in [2].

many users had compiled large volumes of data stored on secondary devices, these files tended to be disparate—each file was accessed by a small community of users with relatively few programs. As a result, file conversion was normally achieved by "dumping" the file onto formatted card or tape records which could be read into the new system with relatively little loss in time and efficiency. Each user saw only his comparatively small data translation cost, even though the total may have been a large drain on the data processing funds.

With the advent of large and integrated data bases using multiple storage media and more complex data structures but constrained to operate within specified data management and operating systems, the problem of conversion became more apparent. The Department of Defense has found that some system conversions cost more for data transfer than for rewriting all of the system and application programs in a new assembly or compiler language.[3]

Even in cases where the hardware, data base management system, and application programs are expected to remain essentially constant, the cost of restructuring a data base (i.e. changing relationships and structures of the data) is found to be prohibitively high. One proposed solution is to isolate the program (algorithms) from considerations of the data which it processes. Such a program is said to be "data independent" [8]. While certain management systems allow parts of the data structure to change without impacting user programs, the technology of data independence is not sufficiently advanced to allow extensive restructuring.

With the implementation of new GDBMSs with hierarchies of storage, the migration of data from higher to lower speed devices means dynamic management of storage structures and dynamic mapping of the logical data structure onto these storage structures. A centralized explicit description of both the logical and physical aspects of the data thus becomes a necessity for these management routines.

Furthermore, with the advent of networks of computers which may have one processor retrieving (or causing retrieval of) data from a distributed data base, the processor should request data at a logical level, and retrieval will then depend on the accessing of self-defining data and storage mapping structures. Once again, this need implies a method of defining not only the logical data structure but a more complete data definition involving also the storage media description and the mapping of data into a storage structure.

The prime requirement of a data definition and mapping language is that it will be able to describe accurately a wide variety of files. Although it is necessary, ultimately, to define a syntax for any language, it is possible to provide a conceptual framework from which any reasonable syntax can be developed. The purpose of this section is not, therefore, to discuss a specific syntax, but rather to show the necessary parts and semantics of such a language. However, a syntactic definition of such a language has been developed; examples of the use of this language are presented in Section 4.

3.1. Scope of the Language

The issue of separating data names and logical structures from physical considerations, such as addresses and the encoding of values, is of prime importance. In compilers, this issue is referred to as "binding," and in generalized data base management systems, it is referred to as the separation of "data structure" and "storage structure."

The trade-offs available for binding in programming languages are well known. The greater the degree of separation between a datum and its type, physical storage address, etc., the greater the flexibility of the language. For example, deferred binding of names to storage addresses allows dynamically growing structures with easy reclamation of the assigned space when the structures are no longer needed. Of course, this increased flexibility has its cost in processing time. Processors of languages with deferred binding are often interpreters or, if compiled, require a large run-time support library to keep track of the current, but changeable bindings.

On the other hand, a more restricted binding of data structure and storage structure often leads to increased efficiency. A good systems programmer can achieve startling improvements in program timing by knowing the way in which the data are stored; he can take advantage of this knowledge in providing the best capabilities for both storage and retrieval of data.

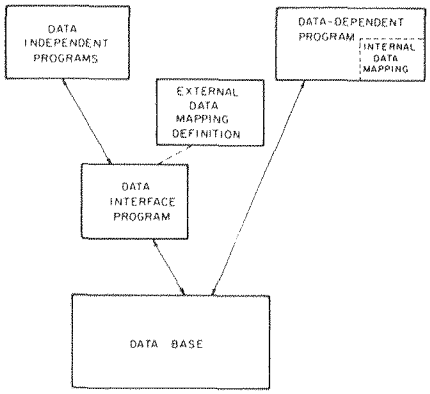
The latest tendency in large scale information processing systems involves the multiple use of common data files. In such systems, no one user has control over the structure of a particular file (except by primogeniture). The role of the *data base administrator* has recently been described [9] as defining "the rules which control the access to the data and determine the manner in which the data will be stored" and further that he controls the physical mappings which "refer to access strategies to be used in manipulating physical data."

There are three tacit assumptions for such functions: (1) that the data base administrator makes decisions on both logical and mapping structures based on the total needs of many users; (2) that data independence allows him to change physical mappings to improve system overall efficiency without affecting the programs of those users who call normal system-accessing routines; and (3) that the system has an interpretive or late binding characteristic to make these roles possible.

However, if descriptions of the logical data structure *and* mapping techniques are available within the system, there is no reason why the sophisticated user should not reference them to improve his program speed and efficiency. Such an action would, however, be de-

[3] Although this is a "well known" fact, to the best of the authors' knowledge, it has not been documented in the public domain. Several inquiries on our part have elicited no specific reference, but a hearty concurrence with the sentiments.

Fig. 1. Data flow independent and data dependent programs. Note: If the data base is restructured, rewriting of data dependent program is necessary; however, it avoids a level of interpretations.



liberate on his part, with the full consideration that a change on the part of the data base administrator could invalidate the program, and hence cause program rewrites. Obviously, this is another of the space/speed and development cost trade-offs so common in large scale systems design.

The difference between such data dependent programs and possible future systems is illustrated in Figure 1.

Fundamentally, the issue is one of user convenience and efficiency. Common data base systems have many classes of users; efficiency and convenience are defined differently by the various classes. To some, run-time efficiency outweighs the risk of possible rewrites. Others demand data base independence and are willing to pay the cost of interpretive mechanisms. The point is that if data base management systems of the future are to offer a range of bindings between logical and physical structures, then an explicit statement of logical structure and physical representation must exist and must be available to users who are permitted access to it.

The second issue which a data definition language must address is that of data structure class. In traditional business data processing, one usually finds only relatively simple file structures. Generally, such files consist of formatted repetition of information for different members of a particular file. Thus, a file may consist of a set of personnel records, one for each of the employees in the corporation, each record of which contains the same number of characters of information (such as name, address, date of birth, present salary, present supervisor, job category). Normally the data structure, which in such a file is nothing much more than a format in FORTRAN or a simple structure in the data division of COBOL, is unknown except to the program or programs that reference the file, as suggested by the internal data mapping of Figure 1. Generalized data base management systems have extended the concept of simple files, and normally allow substantially more complicated data structures than the business data processing systems just described. However, all generalized data base management systems do not allow the same degree of flexibility. It is possible to define various classes of data structure with different degrees of complexity, capability, and overhead. The proliferation of different generalized data base management systems with varying capabilities and with different degrees of complexity of their data structure leads the authors to the conclusion that it is very necessary to define classes of data structure into which the various implementations can naturally be classified. Such a classification provides an explicit definition of differences in the data structures offered by different generalized data base management systems.

Third, it is necessary to be able to describe all phases of data to storage structure transformation. This means that it must be possible to describe the specific data structure (within the data structure classes just discussed), the particular classes of media and secondary storage devices under consideration, the transformation of data structure instances into user working areas, and the method of placing these records into the secondary storage using the predefined access methods.

3.2. Principal Parts of the Language

There are three principal parts of a data definition and mapping language. These are

1. A definition of the data structure.
2. A definition of the target or storage space.
3. A definition of the mapping between data structure and target space.

The data structure section has two functions: the specification of the class of data structure capable of being described within the system under consideration, and the definition of the specific data structure within that class (often termed the *data structure schema*).

The target or storage structure consists of a definition of the different classes of devices, such as high speed memory and secondary storage, which form the basis upon which all data structure instances are mapped.

The mapping language has two functions: to describe those different types of mappings which the overall system can make between a data structure and a target space, and also to provide a means for linking a particular data structure to its specific mapping policy.

As an example, if we were describing a COBOL file, the records of which are to be stored in indexed-sequential fashion on a disk drive, then the total description would be as follows:

1. *Data structure class.* Define the class of data structure available within the COBOL language.
2. *Data structure schema.* Define the data structure schema by naming elementary items, giving their size, and grouping them into levels, etc.
3. *Mapping the data structure instances.*

a. Define both the user working area which is used by the specific implementation of COBOL on the specific machine, as well as the general format of the disk physical record. It might be necessary at this time to dis-

cuss control items stored with the disk record (track and head addresses, record number, etc.).

b. Define the method by which an instance of the COBOL record is assembled into a user working area. This may be extremely simple for this example, but in general, it will require quite complicated statements, e.g. when complex linkages to overflows must be described.

c. Define the format of the auxiliary data used in the indexed sequential access method for the implemented system. This will involve an explanation of the index tables, as well as the relations between the items in it and those in the physical record.

d. Define the mapping between the COBOL specific definition and the user working area, followed by the mapping between the user working area and the indexed sequential file.

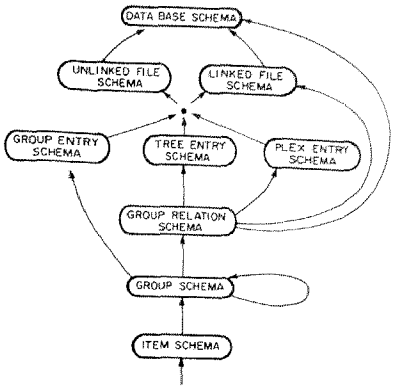
It will be seen that there are certain portions of this description which are purely system dependent, and other parts which are specific to a given file. It is possible, therefore, to consider different levels of detail for a data definition and mapping language. One of these levels could be used to show details such as those in the example above. It would be expected to give a total definition of the system, even though the system does not change from one implementation to another. A second level of detail would assume certain concepts to have been predefined, for example, as macro definitions, giving the entire working of the operating system. Naturally, this would assume that a macro statement like ISAM would convey the total target space and mapping mechanism between a buffer and the secondary storage device. Such a "definition library" would mean that, in many cases, a definer could specify his data structure by merely giving the COBOL definition.

3.3. Summarized Goals of a Total Data Definition Language

This section has introduced some reasons for developing a total data definition language, and used these reasons to set goals for such a language. Basically, such a language is intended to allow description of the logical and physical structuring of a wide variety of stored data. In order to do this, it must meet the following goals.

1. The language must carefully delimit both the logical structure and the physical structure and mapping parts to allow modularity of the language, division of responsibility of the users for various parts of the definition, and ease of implementation of generators or interpreters of the language.
2. It must allow for definition of the richest of data structure classes, but still be able to fractionate the use of simpler classes of data structures. Naturally, this requires that the language provides a means for specifying the class of data structure of the defined system.
3. As much of the language as possible should be nonprocedural since a declarative structure is generally more understandable for complex file definition.
4. The data base administrator should be able to make macro definitions in the language for general use in defining new files or extending old ones.

Fig. 2. Taxonomy of data structure classes in GDBMSs.



4. Elements of a Data Definition and Mapping Language

This section will present examples using a data definition and mapping language which has been defined to satisfy the design goals presented in Section 3. The discussion here is intended to highlight particular needs and, hence, features of the language. The examples chosen have been simplified in order to present a lucid description of particular language features. The definition of real files is, of course, considerably more complex.

One of the design goals just presented is to allow definition of rich data structure classes. The language presented here deals primarily with generalized data base management systems. Though it is the authors' contention that these present as rich a set of data structure classes as is needed to prove the effectiveness of the sample language, it may be necessary to add new concepts for more dynamic data structures such as may be found in artificial intelligence applications.

The CODASYL Systems Committee in its May 1971 *Feature Analysis of Generalized Data Base Management Systems* [10] presents a taxonomy of data structures. This taxonomy is used in the language since it represents a systematic, consistent characterization of data structure classes across a reasonable variety of systems. As such, it provides some assurance that the language defined here will be general in scope. In addition, use of this taxonomy allows definition of data structure classes at a reasonably "high level" using the keywords of the language.

4.1. Review of the CODASYL Data Structure Taxonomy

A full explanation of data structure concepts as they appear in GDBMSs is given in [10] and cannot be attempted here. A brief summary of salient data structure characteristics is presented for completeness.

Fig. 3. Major sections of the language.

```
DATA_STRUCTURE;
      CLASS-SPECIFICATIONS:
            .
            .
            .
      SPECIFIC;
            .
            .
            .
TARGET_SPACE;
            .
            .
            .
MAPPING;
      GLOBAL-ATTRIBUTES
            .
            .
            .
      ITEM_SECTION;
            .
            .
            .
      GROUP_SECTION;
            .
            .
            .
      GROUP_RELATION_SECTION;
            .
            .
            .
      ENTRY_SECTION;
            .
            .
            .
      FILE_SECTION;
            .
            .
            .
      END;
```

The taxonomy can be summarized as shown in Figure 2. The report recognizes six generic levels of data structure in GDBMSS: items, groups, group relations, entries, files, and data bases. Using the data definition language of a particular system, a user will declare a *data base schema*—a "schematic" definition of the data base structure which is independent of data *instances*. A data base schema is declared by giving the item, group, group relation, entry, and file schemas which are its components.

The item (elementary item, data element, atom) is the lowest generic level of data structure. Most GDBMSS offer a variety of *item types* (e.g. name, address, salary) and each item schema (e.g. numeric, character, date) carries one of the available types. For a given GDBMS, the set of item types is a fixed, finite set often characterized by the fact that no operators exist which will select components of an item.[4]

The grouping operation associates a name, a group type, and sometimes other attributes with a previously defined set of items and groups. The items which are immediate constituents of a given group, i.e. not constituents of any other group which is a member of the given group, are called *principal items*. If all components of a group are principal items, the group is called *simple*; otherwise, it is *compound*.

A group relation is a set of ordered pairs of group schemas. This mechanism is used when groups are *related* to one another, but neither is *included within* the other, as may be the case with the grouping operation.

The basic concept of the entry level data structure is that the group or set of related groups corresponds to an outside world entity which is being modeled. Thus, entry is the term corresponding to the more common term "record"; this word was not used since it may be confused with storage structure concepts and methodologies. Different systems allow different degrees of complexity in their entry structures. To some systems, the entry is defined by the outermost level of grouping—the *group entry*. To others, distinct groups can be related so long as the relation is a tree, singly rooted with no interlevel connections—the TREE ENTRY. Still others allow more complex relations between groups in an entry—the *plex entry*.

The file level data structure is defined to be a set of entry schemas (record types). If group relations exist between different entries within a file, the file is called *linked*, or else it is *unlinked*. Similarly, a data base schema is a set of file schemas, possibly linked together.

It should be clear that, for example, group entries related through linked files may be equivalent to, for instance, plex entries in unlinked files. Both involve the grouping operation followed by a single set of group relations. Thus, the taxonomy reflects the systems as they exist, rather than some logically minimal structure.

4.2. Structure of the Language

In order to clearly separate the issue of logical structure from physical realization, and thus meet a design goal of Section 3, the language is divided into three major sections—DATA_STRUCTURE, TARGET_SPACE, and MAPPING—with DATA_STRUCTURE and MAPPING having further subdivisions as illustrated in Figure 3. In the DATA_STRUCTURE section, the declarer will state the salient characteristics of the logical structure he wishes to view. This is accomplished in two steps. First, the declarer characterizes the data structure class in which the particular data structure schema is included. This is necessary in order to take account of the fact that the data structure class will vary from one system to another. Having characterized the data structure class, the declarer will specify, in the SPECIFIC subdivision, the particular data structure schema which is a member of that class and to which data instances conform. This specification involves a capability to state names, nestings, group relations, sort keys, etc. Thus the SPECIFIC subdivision resembles most closely what is usually called the data definition language of any given system.

The TARGET_SPACE section of the language serves to define the space of structures in which data structure instances will be represented. Clearly, there exists in computer systems today a tremendous variety of such structures, ranging from the structure of a "naked machine" to the structures offered by any processor on that machine (e.g. an ALGOL machine). Thus, to be completely general, the statements in the TARGET_SPACE

[4] The RCA UL/I system as described in [10] is an exception to this. It may be treated as a predefined group type.

section must be capable of defining any space of structures for any universal processing system. This is presently too ambitious a goal, however, and the class of target spaces to be defined must be restricted. One candidate which would serve well, from a practical point of view, would be the space of structures that can be defined using cards, tapes, disks, etc. Efforts are under way to include capabilities for defining such a class of structures [6]. At present, however, the level of capability in the storage structure definition is limited to the structure of a "virtual memory machine." That is, the definable storage structures are unbounded finite strings of *basis elements* (either bit, byte, or word) such that each basis element is uniquely identified by a nonnegative integer.

The MAPPING section of the language relates the data structures defined in the DATA–STRUCTURE section to the structures defined in the TARGET_SPACE section. The statements which define this process are broken down by data structure level—item, group, group relation, etc. Several interesting features appear in the mapping process. For example, it is frequently the case that the structure represented in the target space involves data that were never declared in the SPECIFIC subdivision. Tape labels and date stamps are typical examples of non-user-declared data. At other times, the target space requirements may be such that it is convenient to create intermediate structures to simplify the specification of the mapping process. In both these cases it is necessary to create variables and structures which are local to the mapping. The facilities provided by the mapping language for creating these structures are exactly those of the DATA_STRUCTURE section; thus a block structured, recursively processed language results.

The procedural issue has also influenced the design of the language. As discussed in Sections 1 and 3, the implicit definition of storage structures via procedures has been a major reason for the lack of flexibility in translation between various data and storage structures. Thus it would appear that the data definition and mapping language should be as declarative and nonprocedural as possible. Existing systems show that this is possible when dealing with logical structures, and is usually possible in describing the mapping to physical structures. However, the ability to escape to procedures seems mandatory within the MAPPING section. The following examples indicate why this is the case. In storing an entry instance, certain GDBMSs will use list structuring techniques if the size of the entry exceeds certain target space parameters, such as the size of a disk track, but use sequential techniques for smaller entries. Another example is found in entries which are stored using hashing techniques with some method for solving the multiple-hit problem. Cases such as this seem to demand a procedural language in their definition. Thus, at various points, statements in the MAPPING section refer to mapping procedures. These procedures would presumably be written in a conventional procedure-oriented language. However, we hasten to point out that, while necessary in the general case, the use of procedures is not necessary to describe files as they exist in many generalized data base management systems.

4.3. Selected Examples of Language Statements

This section will present sample language statements from each of the three sections in order to illustrate further relevant concepts of data definition languages, as they have been discussed in Section 3. The sample statements will be illustrated by declaring the structure and mapping of a simple file.

Recall from Section 3 that the DATA_STRUCTURE section accomplishes two tasks—the declaration of a data structure class from which the data structure schema will be drawn, and the definition of the data structure schema itself. In this way, the wide class of structures available in the full language can be specialized to a class of structures particular to a given GDBMS.

The declaration of data structure class is given in the CLASS_SPECIFICATIONS section. Statements in this section fall into three categories. The first allows definition of the item, group, entry, and file types available to a definer of a data structure schema. The second allows declaration of attributes which will be carried with each type in a schema declaration. The third allows declaration of structure checks which the data base schema must satisfy. All of these statements are necessary: the first, because different systems allow different item, group, etc., structures; the second, because attributes of items, group, etc., vary; the third, because of the finite restrictions of various sorts imposed by the various systems. The sufficiency of statements in CLASS_SPECIFICATIONS can be measured against the systems in the CODASYL Systems Committee's *Feature Analysis of Generalized Data Base Management Systems* [10]. At present, a series of statements exist which seem sufficient for the systems analyzed there. As the class of structures in GDBMSs evolves, so statements in CLASS_SPECIFICATIONS will also evolve.

The statements within CLASS_SPECIFICATIONS are illustrated by the following example. Consider a data structure class with items of type integer, real, and character string, where each character string item schema must carry a length specification. There may be at most 256 item schema definitions in a data base schema. Items can be grouped arbitrarily so long as the nesting depth is less than 16. Statements to define this data structure class are shown in Figure 4.

The DEFINE statement declares item and group types which will be allowed in a schema declaration. The GROUP entry and LINKED file portions of the DEFINE statement are fixed keywords of the language which restrict the data structure classes to be GROUP entries only, with LINKED file structures permitted. Then, a statement declares that item schemas of type CHAR must have a LENGTH attribute given as an integer (or list of integers). The NESTING_MAX= 16 statement declares that a check be made for depth of nesting of items and groups

Fig. 4. Data structure class specifications.

```
CLASS_SPECIFICATIONS;
        DEFINE (INTEGER, REAL, CHAR) ITEM,
            (SIMPLE, COMPOUND) GROUP
            (GROUP) ENTRY,
            (LINKED) FILE;
        ITEM_TYPE CHAR, ATTRIBUTE = LENGTH (INT_LIST);
        NESTING_MAX = 16
        ITEM_MAX = 256
```

Fig. 5. Sample data structure schema definition.

```
SPECIFIC;
    FILE SAMPLE, TYPE = LINKED
        ENTRY REC_ONE, TYPE = GROUP
            GROUP PERSON, TYPE = COMPOUND;
                ITEM ID_NO, TYPE = INTEGER
                ITEM NAME, TYPE = CHAR, LENGTH = 20;
                ITEM AGE, TYPE = INTEGER;
                    GROUP BIRTH, TYPE = SIMPLE;
                        ITEM DAY, TYPE = INTEGER;
                        ITEM MONTH, TYPE = INTEGER;
                        ITEM YEAR, TYPE = INTEGER;
                    END BIRTH;
                END PERSON;
            END REC_ONE;

            ENTRY REC_TWO, TYPE = GROUP:
                GROUP JOB, TYPE = SIMPLE;
                    ITEM JOB_CODE, TYPE = INTEGER;
                    ITEM SALARY, TYPE = REAL;
                END JOB;
            END REC_TWO;
        RELATIONSHIP WORKS_AT IS (PERSON, JOB);
END SAMPLE;
```

Fig. 6. A target space definition.

```
TARGET_SPACE;
        BASIS = BYTE;
        CLASS POINTER, LENGTH = 4
                    START = MODULO 4 BYTES
        CLASS TWOS_COMP_INT, LENGTH = 2
                    START = MODULO 2 BYTES;
        CLASS FLOATING, LENGTH = 4,
                    START = MODULO 4 BYTES;
```

within groups. The maximum nesting depth allowed is 16. Similarly, ITEM_MAX = 256 declares that the total number of item schemas declared should not be greater than 256. In general, there will be a number of these structural checks that will apply for a given GDBMS.

Figure 5 presents a sample schema definition which conforms to the data structure class of Figure 4. Two group entries —PERSON and JOB —are defined with items of various types included. The item of type character string has a LENGTH attribute specification, as required. A relationship named WORKS_AT is then declared to exist between entry instances of PERSON and JOB. Thus, the statements of the SPECIFIC section look most like the usual data definition language statements, with the exception that the particular attribute keywords will both vary depending on what was declared in the CLASS_SPECIFICATIONS.

Although not shown in Figure 5, the language also contains facilities for the definition of tree entries and plex entries and for the definition of group structures which are arbitrary constructs of lower level nesting and repetition operators. Certain predefined attributes are also included to allow for multiple group instances occurring within a group (the so-called repeating group) and to allow certain items to have derived values (e.g. the "count item" often associated with repeating groups).

The statements of the TARGET_SPACE section are quite straightforward, reflecting the fact that the class of target spaces —finite strings —is not difficult to describe. Figure 6 illustrates a target space definition. The elements from which the strings are built are declared in the BASIS statement. Strings of BITS, BYTES, or WORDS are possible, where each is a primitive concept in the language. Each primitive in the string is assumed to be addressable. In addition, one level of structure can be declared by giving each construct a name, a length, and possibly restrictions on where in the target space the construct may begin. Thus in Figure 6, TWOS_COMP_INT and FLOATING are defined. In addition, two predefined functions exist over the target space: SEQUENTIAL (n, target space constructs) and LIST (n, target space constructs). The first defines a string of n target space constructs where each construct also includes the predefined target space element POINTER, such that a one way list of the target space constructs is established.

Finally, MAPPING relates structures in the DATA_STRUCTURE section to those in the TARGET_SPACE. Sample mapping statements are written as shown in Figure 7. The GLOBAL_ATTRIBUTES section defines certain policies concerning padding, justification and the representation of null values. The other sections of the language are then built up from the item level to define how each construct of the defined structure is to be represented. Thus in the ITEM_SECTION, the correspondence is made between data structure items and their representation in the target space.

The representation of groups can then be considered a collection of representations of the constituent items. Following Olle [11] the keywords of mapping at the group level reflect the fact that group mappings in GDBMSs are either concatenations of the constituent items (shown in Figure 7), or else constituents identified by position, label, or by an indirect addressing mechanism. Thus, the language contains facilities for each of these mappings at the group level. The group level mapping statements also illustrate that mapping policies can be declared by data structure type or data structure

name, with specification by name overriding specification by type.

Group relations are usually represented by pointer mechanisms. Thus, the language has facilities for specifying a wide variety of chaining techniques, one of which is shown in Figure 7. Note that there are two phases to defining the representation of a group relation when pointer techniques are used. First, it is necessary to specify which of a possible family of pointers is to be used for a particular element in a group relation instance. For example, certain systems [12] have pointers which carry encoded information telling whether a pointer points to a header element or a preceding member element. Other systems distinguish between a pointer used in the header element and one used in the member element. Once a particular pointer has been chosen, the chaining policy (e.g. one-way or two-way lists) must be specified. Of course, group relations may not be represented by pointers at all, but rather by matching values of particular data items in the groups, for example. The language has facilities for specifying a variety of such representations.

Entry representation is similar in spirit to group representation. In Figure 7, we have complicated the example by including a created structure—a deletion marker—which will be carried by every entry instance. The means for declaring these auxiliary structures is through recursion. Each section of MAPPING may contain a recursion back to the DATA_STRUCTURE section in order to declare and map those auxiliary structures needed in the realization of the user's data structure. The facility for recursion is typically used to create deletion bytes and other structures which are usually transparent to a user program.

Finally in the FILE_SECTION, our sample file is defined to be partitioned by entry type and ordered by a certain field within each entry. The two partitions are then to be concatenated; thus the file is considered to be sequential in the target space. The FILE_SECTION also has facilities for partitioning files into fixed-length segments (for possible assignment to fixed-length or blocked media) and for the definition of indices which are to be constructed. An index is considered to be a file of derived values. The strategy is to declare the index as an auxiliary structure and define how the index itself is to be represented. An example is given in [13].

Although the sample definition may seem long, we reiterate that not all parts of the definition would be written by a single person for each data base schema. Rather, the CLASS_SPECIFICATIONS section would be written once for a given GDBMS. The MAPPING section, to the extent that mappings are specified by data type and not by data name, can also remain constant across all the files in a given system. TARGET_SPACE definitions, even as they become more complex, tend to remain fixed for a given device or file on a device. Thus the only section that is highly dependent on a given data base schema is the SPECIFIC section in which the schema is de-

Fig. 7. Sample mapping statements.

```
MAPPING;
    GLOBAL_ATTRIBUTES;
        PAD_CHAR = '40'X;
        LEFT_JUSTIFIED;
        NULL_CHAR = '00'X
        NULL_POINTER = '00000000'X
    ITEM_SECTION;
        TYPE INTEGER, REPRESENTATION = TWOS_COMP_INT;
        TYPE REAL, REPRESENTATION = FLOATING;
        TYPE CHAR, REPRESENTATION = SEQUENTIAL (LENGTH,BYTE);
    GROUP_SECTON;
        TYPE COMPOUND, REPRESENTATION = ORDERED (CONSTITUENTS);
        GROUP JOB, REPRESENTATION = ORDERED (SALARY, JOB_CODE);
    GROUP_RELATION_SECTION;
        PARENT OF WORKS, REPRESENTATION = POINTER;
        DEPENDENT OF WORKS, REPRESENTATION = POINTER;
        RELATION WORKS, REPRESENTATION = CHAIN;
    ENTRY_SECTION;
        DATA_STRUCTURE;
            CLASS_SPECIFICATIONS;
                DEFINE(CHAR) ITEM
            SPECIFIC;
                ITEM DELETE_MARK TYPE = CHAR, LENGTH = 1;
    MAPPING;
                ITEM_SECTION;
                    ITEM DELETE_MARK, REPRESENTATION = BYTE;
    END;
    ENTRY PERSON, REPRESENTATION = ORDERED (DELETE_MARK
                                    PARENT_RELATIONS,
                                    CONSTITUENTS);
    ENTRY JOB, REPRESENTATION = ORDERED (DELETE MARK,
                                        DEPENDENT_RELATIONS,
                                        CONSTITUENTS);
FILE_SECTION:
    ORDER PERSONORDER IS ASCENDING BY VALUE OF NAME;
    ORDER JOBORDER IS ASCENDING BY VALUE OF JOB_CODE
    PARTITION REC_ONE_PARTITION IS UNBOUNDED ENTRIES
        OF TYPE REC_ONE;
    PARTITION REC_TWO_PARTITION IS UNBOUNDED ENTRIES OF
        TYPE REC_TWO;
    PARTITION FILE SAMPLE BY ENTRY TYPE
    ORDER PARTITION REC_ONE_PARTITION USING PERSONORDER;
    ORDER PARTITION REC_TWO_PARTITION USING JOB_CODE;
    FILE SAMPLE, REPRESENTATION =
                ORDERED (REC_ONE_PARTITION,
                        REC_TWO_PARTITION);
END;
```

clared. We therefore anticipate that a definition library facility incorporated into a data definition language processor will greatly alleviate the labor involved in producing a complete logical and physical data definition.

5. Conclusions

This paper has introduced the concept of a total data definition language which can describe not only the logical relationships between items of data but also the physical mapping of instances of the data onto secondary storage devices.

There are several reasons why such a language is needed, and several purposes to which such definition could be put. These include:

1. The formalization of the description of the storage structure of modern information processing and generalized data base management systems can be used for better communication of methodology. Presently this description is verbalized in user manuals. The description is usually fractionated throughout the manuals and sometimes is incomplete. A description in an exact and formal fashion would aid the systems programmers who must maintain these large and expensive systems.
2. A formal stored description could be used by the *system itself* for many processes which are now difficult or impossible. Some examples are: the process of reorganizing the data base (i.e. the collection of useless or redundant—garbage—space); the process of restructuring the data base (i.e. the data base administrator's decision to change either the logical relationships or their mappings without affecting the users' programs); and the ability of a system to *generate* its own access mechanisms based on a knowledge of the logical access paths and their physical realization.
3. A formal description of data could be used by *other systems* for intercomputer communication. The use of computer networks, possibly with distributed data bases, means that one computer system may need to utilize data stored in another (possibly "foreign") computer system. The data may be made available in one of several ways from total reading, with translation, of the foreign base, to generation of query language requests in the foreign language. However, in all cases there is a need for an augmented DDL.

The discussion of requirements of such a DDL has led to the definition of a new set of language capabilities in the following areas.

1. There is need for a DDL to define the specific data structures for a given instance of the data base. Such a language exists in all generalized data base management systems, though the capability differs due to restricted classes of data structures allowed by different systems.
2. The data structure class specification should be made in an explicit fashion. This has been achieved by using the taxonomy of GDBMS; though this is somewhat restrictive, it is a good step toward total generality.
3. The effect of storage devices on the class of items stored and on the mapping functions has been investigated, and a first-round design of a language has been produced to satisfy the requirements.
4. A mapping language between these somewhat generalized classes of data structure and storage device has been defined and used to document several real systems.

Our investigations have also revealed the need for investigations into several aspects of total data definition languages. For example, there is probably a need for a storage accessing and retrieval definition language to complement the static DDL definitions now given. Such a language would deal with the descriptions of free storage management, garbage collection, and migration of data. As all of these fields develop, the capability will emerge to exert increased control over the behavior of a GDBMS. It is this increased control which will make GDBMSs truly flexible.

Received March 1972; revised March 1973

References

1. Gosden, J.A. Report to S3 on data definition languages. SIGFIDET *1*, 2 (Dec., 1969).
2. Gosden, J.A. Software compatibility: What was promised, what we have, what we need. Proc. AFIPS 1968 FJCC Vol. 33, AFIPS Press, Montvale, N.J., pp. 81–87.
3. ECMA/TC15/69/15. First preliminary draft report on the ECMA data description language.
4. CODASYL Data Base Task Group. ACM, New York, April, 1971.
5. SIGFIDET, Proc. of a Workshop on File Description and Access, Houston, Texas, Nov., 1970.
6. SIGFIDET, Proc. of a Workshop on File Description and Access, Denver, Nov., 1972.
7. Smith, Diane P. An approach to data description and conversion. Ph.D. diss. The Moore School of Electrical Engineering, U. of Pennsylvania, Philadelphia, 1971.
8. Codd, E. F. A relational model of data for large, shared data banks. *Comm. ACM 13*, 6 (June 1970), 377–387.
9. Joint Guide-Share Data Base Requirements Group. Data base systems requirements. Nov., 1970.
10. CODASYL Systems Committee. Feature Analysis of Generalized Data Base Management Systems. ACM, New York, 1971.
11. Olle, T.W. Data structures and storage structures for generalized file processing. Proc. FILE 68 Internat. Seminar on File Organization, Copenhagen. 1968, pp. 285–294.
12. Roberts, L.G. Graphical communication and control languages. Second Cong. on Inform. Syst. Sci., Spartan Books, Baltimore, Maryland, 1973.
13. Taylor, R.W. Generalized data base management system data stuctures and their mapping to physical storage. Ph.D. diss., U. of Michigan, Ann Arbor, 1971.