A FRESH APPROACH TO DATA BASE MANAGEMENT SYSTEMS
PART I: CONCEPTS, CONSIDERATIONS AND JUSTIFICATIONS

Wm. Orchard-Hays

December 1978 WP-78-64

2361
Laxenburg
Austria

International Institute for Applied Systems Analysis

A FRESH APPROACH TO DATA BASE MANAGEMENT SYSTEMS

Part I: Concepts, Considerations and Justifications

Wm. Orchard-Hays

FOREWORD

This paper is in two parts. In this part, some of the concepts and considerations in the design of a data base management system are set forth, together with justifications for some of the decisions made. In Part II, a particular system is described, essentially in the form of a tutorial users manual. Some further justifications are given there as well as a couple of more discussions of a conceptual nature. Although the two parts are intended to form a set, they can each stand largely alone. A few definitions given in Part I are used in Part II without restatement.

INTRODUCTION

One of the difficulties — probably the greatest one — in designing and implementing a data base management system is the abstract quality of data. It is hard to avoid the feeling or concept that one is dealing with real things of some kind. Thus, for example, one tends to think of a personnel "record" as a concise entity "belonging to" — in the sense of being identified with — a real individual. The personnel files certainly exist because of real individuals and the records for one person or a group of persons can be taken as a surrogate for the person or persons in the sense used, for example, in mathematical models. Furthermore, there are real data items — albeit electronically recorded and manipulated — which go to make up a data base and the records can be presented visually or on hard copy. However, in using a data base, one is interested in much more than retrieving and printing the input records.

One way to approach the problem — probably the only way — is through formal, abstract definitions which define items implicitly. Actually, this is at best an approach to only part of the problem. In addition to the conceptual structure of the data, as such, there are three other considerations:

(1) the "physical" structure of data in the sense of computer-stored files;

(2) the means of specifying and referencing data by a human which requires some kind of language which, in turn, involves further abstractions; and

(3) the relational structure imputed to the body of data which permits using the data for new or unanticipated purposes.

The last is at best a vague idea even though this is often averred to be the main purpose of a data base. It is the thesis here that whatever relational structure exists is a consequence of the designed conceptual structure and human beings' ability to use it effectively and cleverly. Nevertheless, a well-designed system can, in a sense, create new information or at least aid in its creation.

The physical layout of computerized entities and the routines to manipulate them present tough problems in programming. Furthermore, massive data bases may so overtax physical capacities as to lead to a number of practical problems which have very little to do with the initial purposes but which must be anticipated in any practical design. Although, for an initial conceptual approach, one would like to ignore all but the most obvious of the problems arising from a massive amount of data — after all, current data processing systems are powerful and have large capacities — it may not be possible for both practical and theoretical reasons. A basic dichotomy appears justified separating large bibliographic data bases, and more highly structured systems of refined data used in analyses and for input to other computerized procedures, such as models. The emphasis here, and definitely in Part II, is toward the latter. There is also a third kind of data bank, so-called, which involves records of a huge number of measurements, such as telemetered "data" from a satellite. No pretense is made at addressing this sort of records.

As to the means for interfacing the system with human users, two fundamentally different viewpoints are held, at least superficially. One is that a natural-language-like formalism should be provided to make it as easy as possible for almost anyone to use the data base. The other is that a formal, specialized language is much better suited to the task and that it is to the users' advantage to take the trouble to learn it. Those that are unable to should not attempt to work with something they don't understand. Both these positions are extreme and, in practice, both are modified considerably. It is, in fact, impossible to use a natural language in full generality; only narrowly constricted, specialized subsets of a natural language are really programmable. Furthermore, facility in such a sublanguage does very little to clarify difficult concepts which must somehow be phrased. On the other hand, a formal language is seldom used in its pure form by ordinary users. Various simplifications

and combinations are usually defined and given reasonable-looking labels. Indeed, with either approach, a substantial part of the working language consists in words which were defined by the user, using the basic capabilities, however expressed. Natural-language proponents admit that an underlying formal language is necessary and the natural language capabilities are a superstructure. Any such superstructure will be ignored here. Nevertheless, the language presented in Part II is not strictly a formal language. It looks about like the command-style English found in mathematical papers but is rigorously defined.

So much work has been done in this area for such a long time that it is perhaps presumptuous to speak of a "fresh approach", particularly in a short paper. The term can be interpreted in the sense of "let's try again" or "back to first principles". Sometimes this can bring new clarity without actually inventing radically new concepts or mechanisms.

BASIC CONCEPTS AND DEFINITIONS

Of what or in what does data consist? This is not a trivial question in spite of our intuitive feeling that we know what data is. The question here is not whether some set of data is correct but in what sense a number, symbol, or other representation has meaning. There is a dual nature to data: its meaning and its representation. It is almost impossible to fully specify representations; we always rely on a great amount of cultural and technical background. How can one define ab initio an alphabet, the arabic numerals, word structure, floating point representation, graphs, etc., not to speak of bits, bytes, words, records, files, etc., in a computer system? Whole books are written on such subjects. Particular characters, such as the greek letter pi or the plus sign, are understood to have meaning on the basis of an extensive and almost universal cultural background. Representational structures are an important aspect of data but they are not its conceptual structure even though there may be some correspondence. In some cases, the representational structure may even appear to be part of the concept — a matrix, for instance. One of our most fundamental concepts is the distinction between right-hand and left-hand. It is natural to use representations which somehow picture such concepts. It is a trap, however, to suppose that any graphic representation really defines what it stands for, even though mental images of the representation may greatly facilitate thought processes, such as calculation or puzzle-solving.

Although it is even hard to decide what constitutes the basic elements of representations, let alone data units, they are specifiable through various conventions which are widely or universally accepted by people with any reasonable degree of education for the purpose at hand. It would be merely pedantic, and futile, to try to explain, justify, change or define in more basic terms those conventions which are recognized by most people with an interest in what is being discussed.

The conceptual structure of data is more subtle and less direct or intuitive. A representation which may appear understandable, even pronounceable or with exact numeric value, is not a piece of data in isolation. The representations

PRODUCTION $2.135 * 10^6$

are readable but have no meaning standing alone. We will term a representation like either of the above, or any of several other possible forms, a datum. In a workable system, the allowable forms for a datum must be specified, of course, but that is a subject for Part II.

We encounter a deficiency in natural language here and will take care of it immediately. What is the plural of "datum"? A datum is not in and of itself a piece of data. Hence the rather infantile-sounding "datums" will be used for the plural. Data (which we construe as a collective singular) is represented by datums but has an abstract structure which gives it meaning over and above representations.

In order for a set of datums to constitute a piece of data, two things are necessary. At least some of the datums must represent values of different attributes and the set of attributes represented must be logically related to some conceptual item. We define these ideas more precisely as follows:

An attribute is an abstract quality or characteristic which can be measured by and only by a finite number of discrete and distinguishable values, represented by datums. The attribute must have a unique identifier in context but its allowable values need not be unique independently, i.e., among different attributes.

An item is a member of a set defined by a particular set of attributes. An item is defined implicitly by a particular set of values for the attributes defining the set to which it belongs. At least one combination of attributes must have unique sets of values over the set of items if identification of items is to be possible.

A data structure is an organized set of datums to which meaning is imputed, first by its rules of organization, and second by the means of accessing it.

An inferior is a substructure which is accessible only through a superstructure, called its superior.

A peer group is a set of structures all of which have the same superior. If all members of the peer group are actually hooked to the same superior, the group is called an echelon. If a superior has only one hook to the inferior group, all members of the group have the same structure, and they are connected to each other (including the one hooked to the superior) by some ordering principle, the group is called an inferior set. Any member of a peer group may have its own inferior group which is a peer group. The inferior groups of a peer group are said to be at the same level regardless of the connecting rules.

We interrupt to comment on the two forms of peer groups. It might be supposed that the strictly hierarchical echelons are simpler, more natural and hence more efficient and useful. This is not the case, however. The organization presented in Part II is based on inferior sets which have advantages both for storage and searches.

A data bank is a collection of data structures which themselves form a grand data structure. One and only one data structure, called the root, is a superior without a superior and without peers in the context of the data bank. The root is regarded as level zero.

A data base management system (DBMS) is a system for creating, modifying, manipulating and using a particular style of data banks. The DBMS consists of a system of computer routines, one or more languages for controlling these routines and for specifying various forms of datums, and manuals for explaining and documenting all parts of the DBMS including themselves. However, the DBMS should be distinguished from any particular data bank and related material which is implemented using the DBMS, except possibly an illustrative example or embryonic structures common to all uses of the DBMS.

It may or may not be possible to separate a DBMS from a particular type of computer. We do not regard such inseparability as a fault, nor portability at the expense of worthwhile characteristics as a virtue. These are questions of implementation which require different viewpoints and judgments

from those taken here. One cannot completely ignore the style of computer system used, however. Good character and character-string manipulation capability, adequate central and peripheral storage, fast execution speeds and high data transmission rates, and extensive provision for supporting numerous types of perioheral devices and telecommunication lines are all necessary for effective implementation of an elaborate DBMS. Interactive operation is also a requirement. We are not interested in the question of how small or inadequate a computing system can be "supported".

Some readers may feel that the general concept of data banks and DBMSs have already been overly constricted by the foregoing discussions and definitions. As a practical matter, however, it is difficult to see where one can begin with much less bounding of the problem area. When one gets to actual specifications and design, many further limitations have to be imposed.

FURTHER BASIC CONSIDERATIONS

Certain terms used in the definitions of the preceding section were neither intuitively clear nor defined. This was necessary since the intent was to first briefly encompass the range of the subject.

Let us first distinguish between data entities and functions of sets of entities. The definition of an attribute is normally an entity. An item, on the other hand, may or may not be an entity but the set to which it belongs is at any rate a function of a set of attributes. An entity must have some form of identifier by which it can be referenced, and be extractable as a unit when once located. The set of identifiers for all attributes defining a set of items may itself be collected together and given an identifier, in which case it has the formal structure of an attribute. However, regarding it as such involves one in a logical morass. Rather, the identifier for the set of attribute identifiers is effectively the referent to the set of items. This is only the first of many examples where great care must be taken to distinguish between formal structure and conceptual structure.

The following definitions will be useful.

A primitive set is a set of datums deliberately defined for some purpose with fixed formats and specified values, arranged in the form of a set of items embracing or as though embracing one or more attributes. The set of items and, optionally, the attributes are given pre-defined referents which have the status of reserved words, either in a language or the meta-language defining it.

(One of the difficulties with a natural language is that it is its own meta-language and hence cannot be rigorously defined.)

A symbol is a character string restricted in form by convention and used as the name of something. A typical restriction is that the first character be a letter, that only letters, digits and perhaps one or two other characters (such as the period or currency sign) be used, and that total length not exceed eight characters (sometimes six). (Less restrictive conventions are also in use but too much latitude creates problems for both designer and user.)

Special characters are characters which are given special meaning, usually wherever they appear. The set of special characters are usually further differentiated into operators, punctuation, flags, etc. (Their definitions are an instance of primitive sets.)

The use of special characters is unavoidable and no one would want to do away with all of them. However, beyond those universally accepted — such as the arithmetic operators, comma, and such like — there is little agreement on the meaning or desirability of further special characters. Their proliferation is limited by the availability of graphics on various peripheral devices although the number of available graphics has been increasing. Unfortunately there has been little standardization with respect to keyboard positions, internal codes, and local meaning (such as typing controls). Even the standard special characters are often used with non-standard meanings or traditionally have multiple meanings depending on context. The asterisk, for example, may mean multiplication, indicate a comment, be used as a universal character, or, doubled, represent exponentiation. One cannot deny the naturalness of these various uses (except possibly the last) but proliferation of special meanings for characters leads to logical snarls, or at least untidiness. Some languages are built up carefully and almost exclusively from special characters; if suitable to the purpose, they may be highly efficient. However, such an approach seems unsuitable for a DBMS language for the general user. Perhaps the cryptic nature of some formal languages has been the chief motivation for proponents of natural language.

It might be thought useful at this point to introduce the concept of divisions into data banks, in the sense of main functional subsystems, somewhat as in COBOL. One might, for example, define a language division, an operational division, and a data division. (Another main subsystem, the underlying file system, is clearly necessary but it is best kept below the view of the user.) A serious attempt was made to use the idea of divisions — in fact, the three just mentioned — even to the point of trying to base

a preliminary design of the DBMS in Part II on it. It proved to be unworkable when details were examined more closely. It is true that a system has what may be termed dimensions, and three important ones are the command and control language and mechanism, the operations and functions existing in executable code, and the data files and other structures upon which the system operates. However, these are quite dissimilar things, even conceptually. While it may be possible to conceive of some supersystem which embraces the various subsystems as though they were special cases of a unified formalism, this leads one into a number of difficult problems in programming, computer science, logic and probably other areas. At best, the solution of these problems, however elegant in concept, must lead to inordinate complexity in the actual routines which do the work. The practical problems to be dealt with in a DBMS are already severe enough without further burdening the system for the sake of abstruse ideas. Furthermore, it does not appear that the idea of divisions really helps the user or leads him to a more orderly breakdown of his work and material. If anything, on the contrary, it blurs distinctions which ought to be kept clearly in mind.

(The writer once designed and implemented a large system for a different but not unrelated application area in which a similar kind of generalization was largely achieved. The system was extremely disappointing in a practical sense because of its inefficiency and continual respecification of what was, in fact, already known. From a programmer's viewpoint it was elegant and flexible but it solved the implementer's problems, not the users'. One can be deceived by the apparent similarity of all coding in the implementation language. The use of standardized techniques and structures in the implementation language is to be recommended, even required, but these often involve complicated formalisms which the user of the system is unfamiliar with and should not be required to understand. However, if these formalisms inhibit the practical and efficient application of the system, the user has a right to complain. The user of a large system, particularly a DBMS, is already dealing with a difficult problem area. The system should assist him with his problems without burdening him with the implementer's problems. Of course, if a clever concept helps both — as occasionally happens — it should by all means be used. It appears that the idea of divisions helps neither.)

We will restrict attention here to what would have been termed the data division, i.e. the data bank proper. Some further consideration of

referents is in order. Even if these are restricted to symbols, one symbol will seldom be sufficient. It is probably impossible and certainly undesirable to maintain uniqueness across all levels and data structures. Both the meaning of a datum and the way it is accessed depend on the relational path by which it is reached. If the data bank is hierarchical, as has already been tacitly assumed, it is possible to record the most direct path from the root node to any physical entry by a chain of referent symbols or some kind of pointers. However, attempting to record these chains would be silly since they would amount to a large set of predefined identifiers, of varying length. Since they would not represent the type of relationship frequently required, even a large set of them would represent only a small fraction of the desired possibilities.

The above difficulty is resolved by recalling the distinction between an attribute name (and form definition) and an attribute value, and by the use of inferior sets rather than echelons. The inferior-set organization endows the data bank with an unambiguous form of hierarchy which distributes values in such a way that only inseparable relationships are recorded. That is, the form of physical paths through the structure is fixed but the possible paths are very large in number and efficient on the average. The attribute definitions are maintained at just the point where they are needed and apply to the most nodes without duplication. This position is at the unique connection of a node to its inferior set. If the idea of a key attribute is introduced here, physical paths are then uniquely defined. One can then search either strings of attributes by name, to locate a set, or strings of values to collect members of an implicit set. The assignment of the key attributes is critical of course and represents a restriction on the generality of structure. However, since each one applies to only one (homogenous) inferior set, the restriction is minimal. This concept is elaborated in detail in Part II and defines the fundamental organization of the data bank.

There is still a large question left as to the order in which a complicated search command should be executed, and to what extent, if any, this should be intermixed with parsing and interpretation of the command. But this problem is close to the surface of the DBMS and can be improved independently, without altering existing data structures.

ON ALLOWABLE KINDS OF DATA AND OPERATIONS ON IT

At first glance it would appear impossible to circumscribe allowable kinds of data and operations on it without seriously reducing the generality

of a DBMS. Yet it seems that this must somehow be accomplished. We begin by dividing the problem into parts, a not very novel idea.

There is, first, the matter of form which has two aspects: external or graphic form, and internal or coded form. Although not completely trivial, this aspect can be taken care of fairly easily.

Second, there is the matter of content. One's first reaction to this is possibly the snip reply that we don't care what the content is as long as there's not too much of it. On more careful thought, however, content must be considered if only in a negative way. Voluminous data or preliminary studies which cannot be abstracted and organized in some fashion to give meaningful "handles", so to speak, cannot be much helped by a DBMS. That is, content does have implications for the third matter of our concern, namely, the induced structure which the DBMS must be adequate for. This conceptual structure becomes a kind of generalized syntax for the material being organized.

Fourth, there is the matter of useful operations. These are not as diverse in practice as might first be thought. The form of data restricts the range of operations. The widest range of operations and functions is for numerical data but this is also the easiest to deal with and to pass to external procedures, if necessary, using standard or easily definable conventions. The DBMS need not accept responsibility for the interpretation or validity (other than for arithmetic and a few standard functions) of numerical transformations. It is sufficient to produce the requested inputs and to re-file the proferred results.

Fifth, as just implied, there is a distinction which should never be forgotten between valid handling of data and valid interpretation. We contend that no mechanistic system can impute meaning to data or deduce interpretations. Data means at most what the user says it means (often less). The DBMS need not be concerned with meaning or interpretation but only with formal relationships. This introduces a large divisor between the possible range of applications and the necessary range of processes which must be carried out.

Sixth, there is the matter of the style of language, which has implications in restricting the range of operations. We are prepared to be quite arbitrary, though hopefully consistent, with regard to language style. There are two kinds of restrictions: those that prevent unnecessarily complicated or difficult-to-execute statements, and those that deprive the

user of desired capabilities. Within reason, the latter restrictions should be avoided. The former seem allowable; it is impossible to satisfy everyone's tastes anyway and legitimate restrictions should not be shied away from. It is always possible to build superstructures for convenience on a clean language.

Seventh, and finally, the possibility of special versions should be provided for. Almost any large application is likely to have special requirements for which special provision should be made in the basic programs. This is not a suggestion that everyone tinker with the system or that it should be necessary in general. But large, complex applications will almost surely have a life of many years and the expense of a special version may be very low when amortized over its lifetime. This depends, however, on clean design in the first place.

In the following sections, the above aspects of the problem will be dealt with in more detail though not strictly organized as numbered above.

DATA CONTENT

Ignoring for the moment the distinctions between mere datums and data to which some meaning can be imputed, how varied can data content be? If one approaches this question from the standpoint of subject matter, there are virtually no bounds. Subject matter, per se, however, is of no consequence to the design of a general DBMS. Nevertheless, some subjects normally have a semantic content which is more readily organized than others. Stastistical data, for example, is more manageable than textual reference material though one subject may involve both. Textual material is itself quite diverse; it might be reasonable to put a handbook for a scientific area in a data bank but hardly a history book. At least some kind of indexing based on key words and phrases must be possible and even then inclusion of an entire document may be impractical and unnecessary. References to encyclopaedias might be suitable content in a data bank but not the encyclooaedias themselves. It must be admitted that some degree of arbitrariness appears involved here. It is not inconceivable that someone might find it useful to have an encyclopaedia computerized, but a special system would be more suitable for such a purpose.

The problem of abstracting material from scientific and scholarly material is a very difficult one. Even professionals in this area don't seem to always do a very effective job which no doubt reflects more the difficulty of the task than on their competence. It would seem foolish

to attempt to make a contribution to so difficult an area in a DBMS. The most that seems feasible is to mechanize the techniques which abstractors would find useful in their work or use in presenting their results. Thus a request to a DBMS to find all references to, say, "energy supply" in the entire data bank would be a very inefficient use of the system unless references under such a heading, perhaps under several superior headings, had previously been created. It is true that computer programs exist — for example, context editors — which will quickly find all occurrences of any string in a body of material, but the volume of data through which they search is comparatively small, usually no more than a few thousand characters, and the organization is simple. At around 100,000 characters, perhaps the equivalent of 50 typewritten pages, search time begins to be quite noticeable.

Should the user then be required to separate his material into distinct classes, with clearly stated hierarchical organization? Most DBMSs require this and there are several advantages as well as the apparent near-necessity. A possible disadvantage is that it may make it difficult for others than the developer to use the data bank but, first, this can be overcome by the ability to display organizational structure, and second, what other approach would make it easier for an unfamiliar user. A very important advantage is that the developer of a complex data bank must organize the material if the result is to be worthwhile. The DBMS should aid in the process but not be required to give good service without it.

The user must, in fact, organize his material in two ways: the conceptual structure which only he can create though with assistance from the DBMS, and the separation of different forms of material which require different handling and which the DBMS can enforce. A discussion of forms will clarify the latter point.

DATA FORMS

As previously stated, data must be considered in both external and internal forms. Externally, it must also be further differentiated in part according to whether it is input or output. A graph, for example, can not be input and stored as such but can be output. Some output data appears the way it does by virtue of a peripheral device, such as a plotter, and is not properly an output of the DBMS itself. However, it is a waste of time to be too picayune about such distinctions.

How many external forms for input are reasonable? Actually, not many,

and some distinctions are mere technical details, such as fixed-point versus floating-point numbers. From one point of view, the following list covers all reasonable possibilities:

1. Statements typed at a terminal which are structured but may contain commands, numbers and some amount of text.
2. Computerized input files (tapes, cards and disks primarily) containing arrays of numbers, possibly with some symbolic indicators.
3. Computerized input files containing larger amounts of text, perhaps with some editing or indicative information.
4. Mass transfers from another data bank.

There are variants of some of these, such as input via remote telecommunication, possibly use of a light pen or an optical scanner, or the output of another program, but these are not fundamental distinctions. If programs, per se, are to be dealt with, there are two possibilities: source code to be put in the data bank itself, and executable code for extending operational capabilities of the DBMS. The first does not pose any problem different in kind from other textual data; it is probably even simpler to organize appropriate reference relationships. Executable code, on the other hand, does pose technical difficulties but, hopefully, these will not be insurmountable though certain restrictions may be required. In this connection, one must distinguish between macros in the language of the DBMS itself, and linkage by the DBMS to an executable library of routines.

On the output side, the four types of input mentioned above all have equivalences and, in addition, forms such as graphic displays, plotted material, and possibly others are desirable. However, most of the additional forms are in the nature of post-processing of data produced by the DBMS proper, that is, they can be added to the basic system as required. General format control for printed output should be in the DBMS itself.

Internally, the major dichotomy is between rather stereotyped structures used in the command language and the varied forms in the data proper. These should have an obvious relationship, however, and some overlap in form. The first requirement for the data is the ability to store items which are entities but which have a mixed-mode substructure. This implies at least two more capabilities: some way to describe the format of the substructure, and some way of relating the items to subject matter. These obviously have implications for external syntax as well — there must be some way to specify these things and the way should seem natural. It is probably to be recommended that stereotyped format descriptions be used (except for report generation). This turns out to be natural for the definitions of attribute forms once the allowable datum forms are defined

and given labels. No practical limitation need be made. A relatively few forms can accomodate almost any reasonable kind of datum. There are some limitations of a different kind, however. Many items will have associated data which may be fairly voluminous, such as a matrix, a table or actual text. The case of actual text is the most difficult since it is uncertain how to store an unstructured and unknown amount of input. It can be done but it would seem reasonable to impose a fairly low upper limit on any one such body of text. Searching should be done on abbreviated relational and indicative data, with associated pointers to voluminous data, the latter being stored separately and retrievable only in large units. As an analogy, it is usually better to consult a card index in a library rather than wandering through the book stacks. However, the ability to do some amount of browsing may be desirable.

The question of packing sparse data is also important. For example, suppose ten different attributes are embraced by a set of items but, on the average, six of them are void. Without packing, 60% of the storage space is wasted (though packing eats up some fraction of the saving). Packing in itself is not so much of a problem but both updating packed data and searching it can be awkward and inefficient. Perhaps the best that can be done is to allow the user an option as to whether packing is reasonable or not, at least for arrays attached to an entity. If identifying attributes are often void, it is questionable whether the defining set of them is properly defined itself.

These are some of the considerations which must be taken into account in detailed design. We must pass on now to other aspects.

INDUCED STRUCTURES AND OPERATIONS ON DATA

To illustrate the concept and problems of induced structure, let us consider a small example. Suppose a personnel file consists of items, which are entities, with the following twelve fields:

name, exmplyee no., social security no., sex, data of birth, department, title, salary, fixed deductions, dependents.

(Several more would be required in a real personnel file of course, but the above is sufficient for illustration.) First we note that different formats are required for the fields. Down through salary, these can be fixed for all employees although "name" would then require maximum length which would be needed for only a very few. (The wastage could easily average 10 characters per person.) But "fixed deductions" and "dependents" require substructures of their own unless only totals are recorded; this might be

sufficient for "dependents" but hardly for the other. So already there is a problem of storage organization. There are, of course, many solutions to it but they all complicate searching of items, especially for particular attribute values. If one is concerned only with the personnel file, some "best" organization can be determined statistically, but how can this be done for a general data bank and how many rules can a DBMS support?

Second, only the second and third fields are absolutely unique to the real individual, and hence to the file; even names may duplicate. At the same time, no field will have consecutive values over items no matter how the latter are sequenced.

Third, there are several possible orderings of the items which are "sensible": alphabetically by name, monotonically increasing by employee no., major by grade, major by department, minor by name within department, and so on. Only ordering by employee no. (if assigned sequentially over time) will guarantee that no inserts need be made, although deletions will be required.

Such problems are familiar enough to anyone who has had to deal with such files. But our personnel file is essentially the simplest kind possible; it is quite homogeneous with only a couple of possible exceptions. Suppose it is stored in order on employee no. but we chain together all employees in the same department and also with the same grade. First we must decide on the order of chaining. Perhaps we want a department chained alphabetically by name, and grade by department. We have then induced two more subattributes, "after" and "before", to each of the attributes "grade" and "department" and fields must be provided for each in at least one direction. Furthermore, anchors for these chains, for each department and each grade, must be provided. Already the overall structure, both conceptually and mechanistically, has taken a quantum jump in complexity.

Of course, the above requirements could be satisfied in other ways such as searching and sorting. Indeed, the ordinary user would probably not consider chaining but would simply request, for example, "all employees in department xyz, alphabetically by name". If chaining had not been done, the implied search and sort would be the only way to get the information. Suppose, however, that chaining had been done but this user didn't know it. Should the DBMS be able to take advantage of the chaining automatically? This might not be unreasonable if the attributes and any subattributes were properly recorded internally. What would be too much to ask is that the DBMS be able to do the chaining automatically when the items were stored,

that is, without specific instruction.

A department manager might want merely an up-to-date listing of his employees but, more likely, some executive would want a distribution by department of salaries and perhaps other attributes like sex and race (or whatever circumlocation is in vogue). This leads us to the question of what standard functions should be provided. Should "distribution by ..." be available or must someone program it from more basic operations? In the latter case, if the request were frequent, it would be desirable to store the program as a new operation. In one situation, a decision could be made but if we turn to other kinds of data, an entirely different type of operation is needed. For example, if one is dealing with sets of encodings which must be combined in various ways, operations like concatenation, masking and symbolic incrementing are needed. Again, for some kinds of data, one wants means, standard deviations and other statistical tools. The four arithmetic operations, square root, exponentials and logarithms, and similar basic tools are frequently needed.

The criteria for primitive operations should probably be generality and difficulty of programming with any others built up as storable macros or accessible in a subroutine library. With this approach, the content of the basic routines in the DBMS can be minimized and specialized, providing robustness without excessive size. The basic and general operations and functions usually require more polished implementation than more comprehensive functions. For example, square root can be programmed from the arithmetic operations (indeed must be at some level) but an efficient and robust square root routine is not trivial to program. Even the arithmetic operations for mixed modes and precisions are rather complicated. (Compilers hide all this from the average user of a higher-level language.) However, given general summation, counting and square root, the mean of a set of values is trivial to program. In the case of symbolic operations, such things as masking, concatenation and symbolic incrementing are tricky to code but, given such primitive capabilities, quite complicated symbolic functions are readily programmed. The set of primitive operations should be much larger than those mentioned above, of course, but not excessive. Rather than trying to be too comprehensive, optional inclusion of both user-defined macros and standard or user-provided subroutines should be made easy and workable.

Desirable as it might seem to be able to extend the set of primitive operations, that is, by the user, this not only leads to severe implemen-

tation problems but requires the user to be aware of and take into account numerous considerations with which he is most probably not familiar, and should not be required to be. As previously remarked, when special situations justify special versions of the DBMS, they should be created by those who are familiar with internal intricacies and can guarantee reliable use.

EPILOGUE TO PART I

Clearly much more could be said on any of various parts of the subject raised in the foregoing sections. Equally clearly, not all readers will be satisfied with what has been said and this includes the writer. Still, no purpose is served by endless discussion. At some point, the principles and suggestions elicited by discussion (albeit a monologue) must be tried and the results evaluated. It seems that that point has been reached.

In Part II, a design for a DBMS will be presented. Some concepts presented vaguely or by mere reference in this part will be given greater precision as a matter of course. Other features will appear which have not been discussed. Some will be justified, others will be left to the reader's own evaluation.

Order of presentation is itself a problem. No justification or explanation for the order used will be given in Part II. The writer has followed what seemed to him a logical sequence which minimizes the number of terms used which, at any point, have not previously been explained. It does not seem possible to find any order in which this can be completely avoided. When serious misunderstanding appears possible, forward references or interim definitions are used.