Master File

A SYSTEMATIC APPROACH TO THE DESIGN OF COMPLEX SYSTEMS: APPLICATION TO DBMS DESIGN AND EVALUATION.

R. C. Andreu
S. E. Madnick

March, 1977

REPORT CISR 32

Sloan WP 920-77

Center for Information Systems Research
Alfred P. Sloan School of Management
Massachusetts Institute of Technology
50 Memorial Drive
Cambridge, Massachusetts, 02139

A SYSTEMATIC APPROACH TO THE DESIGN OF COMPLEX SYSTEMS: APPLICATION TO DBMS DESIGN AND EVALUATION.

R. C. Andreu

S. E. Madnick

ABSTRACT

Software systems produced to support complex applications are often found to be costly, unreliable, difficult to repair or modify, and not particularly responsive to user requirements. Such problems, although detected in latter phases of the system development process, reflect the lack of an appropriate methodology for earlier phases. The need for structuring these phases around a framework consistent with the system requirements suggests that the traditional "requirements analysis" development phase could be extended to infer a system structure that can be used to guide the design process. Basically, we propose to investigate the possibility of isolating groups of requirements whose elements are strongly interdependent, and infer, from them, design subproblems, thus decomposing the design of the overall system into those of more manageable subsystems. The emphasis is on a methodology to identify such subsystems; in the past, this activity has only been approached in an ad-hoc manner. Our proposed structuring framework can also be used as a basis for constructing system evaluation models.

A SYSTEMATIC APPROACH TO THE DESIGN OF COMPLEX SYSTEMS: APPLICATION TO DBMS DESIGN AND EVALUATION.

1.- Overview.

The fact that software systems produced to support complex applications are typically found to be costly, unreliable, difficult to repair or modify, and not particularly responsive to user requirements has triggered increasing concern about what can be done to avoid such inconveniences. The purpose of this paper is to suggest an approach to complex software systems design explicitly cognizant of these problems, in an effort to deal with them effectively.

It is our contention that these problems, typically detected in latter phases of the system development process, result from more fundamental flaws at earlier phases, and that there is a need for a strategy aimed at structuring and organizing design activities in a well defined way from the start.

Basically, we suggest that the design of a complex system in general should be organized in a technology independent framework (so as to avoid technology biases that often "force the problem to fit the solution"), in whose context the design decisions can be conveniently conceptualized. We will argue that this framework should allow the designer to make explicit his knowledge about the system of interest, in such a way that the trade - offs among both system requirements and alternative

implementation techniques can be easily identified and resolved. In the realm of complex systems, this framework should also permit the "decomposition" of the global design problem --often so large that it becomes intractable-- into more tractable subproblems of moderate size. This decomposition idea results in a hierarchically structured framework.

The main thrust of this paper is to emphasize the process of constructing such a framework in a systematic way. As we will see, the idea of decomposing a complex system so as to simplify its design is not new and has been used in the past. However, since the decompositions employed have been either intuitively generated or implied by specific implementation techniques chosen a priori, the resulting decomposition is often difficult to justify and not always appropriate. In contrast, we will focus on a methodology to derive a system framework that facilitates viewing it as a collection of subsystems whose designs can be approached as independently as possible of one another.

The methodology that we propose to accomplish this is centered around the concept of design interdependencies among system requirements. Its application requires specifying a set of system requirements (stating how it is to behave) that is then structured by assessing interdependencies between pairs of requirements. These interdependencies are meant to make explicit design trade - offs that could otherwise be overlooked, because there are so many of them. That set can then be decomposed into subsets which point out groups of

requirements so interdependent that it makes sense to take care of them at the same time in a "design subproblem". This decomposition activity can be performed by way of solving a graph decomposition problem. The resulting "design subproblems" effectively define a collection of subsystems, organized as dictated by the interdependencies among requirements belonging to different subsets, thus pointing out how the several subsystems' designs should be coordinated in the design of the complete system.

We also propose to explore the possibility of constructing a performance evaluation model for each subsystem, then to be combined into a performance evaluation model for the entire system.

For concreteness and practical reasons, we propose to investigate the appropriateness of this strategy for a specific case. Since an increasing number of data processing applications require the manipulation (i.e., storage, retrieval, processing) of large amounts of related data items, and Data Base Management Systems (DBMSs) have evolved as a response to such requirements, we have chosen DBMSs as an instance of complex software systems characterized by many of the inconveniences outlined above.

Although several DBMSs have been developed and effectively used in recent years, thus generating alternative approaches to system organization and identifying a number of implementation techniques, those DBMSs have, by and large, resulted from ad - hoc solutions to specific data manipulation problems. As is the

case with other software systems, little attention has been paid to the identification of a systematic approach to DBMS design in general.

We believe that it is precisely the lack of such an approach what is responsible for many of the typical drawbacks found in existing DBMSs. In particular:

- There is no agreed upon framework in whose context the design decisions can be coordinated.

- System adaptiveness to changes in operational needs is made very difficult and time consuming by the fact that such changes often impact the entire system.

- The incorporation of new, potentially appropriate technology (both hardware and software technology) into an existing system is cumbersome, because there is no systematic way of analyzing how that new technology would affect the system operation if adopted.

- System performance evaluation may require an enormous model to represent the entire system, so that problems regarding adequate modeling methodology often arise.

In the latter sections of this paper we exemplify how the strategy in this proposal can contribute to easing these problems.

The paper is organized as follows:

Section 2 analyzes the complex system design problem from a general standpoint and attempts to identify its roots. The need for a simplifying scheme is discussed and the decomposition idea motivated. A set of properties that a system

decomposition strategy should possess is then introduced, and the advantages that are likely to result from this strategy are outlined.

Section 3 is a brief literature review showing that the decomposition characteristics motivated in section 2 have in fact been explored in the past, but never concurrently.

Section 4 is intended to introduce a proposed methodology for system decomposition whose output is to be a system framework in which the design process can be structured. It gives rise to several research activities that should be undertaken in order to make that methodology operational.

The remaining sections are devoted to exemplify how the proposed methodology can be applied to the DBMS case:

Section 5 contains background information about the nature of DBMSs, presents a series of typical DBMS drawbacks that motivate the need for a better design strategy, and describes an example in which the methodology is actually applied, resulting in a DBMS framework useful for design.

Section 6, finally, discusses how design activities can be organized in the context of this framework, and illustrates a possible coordination of design subproblems.

2.- The need for a design framework.

Software systems produced to support complex applications are often found to be costly, unreliable, difficult to repair or modify, and not particularly responsive to user requirements. These problems have been explicitly recognized in recent years. Software development costs are on the increase, and frequently there are additional costs derived from software development delays. As pointed out in [Brooks 75], increasing manpower to solve these problems is not always appropriate, and often it is even counterproductive.

Focussing on the software development process, we believe that those problems can be alleviated if this process is organized around a meaningful framework, in whose context the design activities can be coordinated from the early phases on. Our contention is that problems appearing during later phases in that process (e.g., implementation or maintenance) are often due to more fundamental flaws at earlier phases, particularly requirements analysis and the so called "preliminary design".

In this section we examine the need for such a framework, make an attempt to identify the properties it should possess, and discuss how it can help to avoid the problems mentioned above.

2.1.- The complex system design problem.

From a general standpoint, the process of system design is concerned with meeting a series of system requirements (i.e., specifications stating how the system is supposed to behave), by means of appropriately combining available technology. This implies resolving the trade - offs that exist among system requirements (e.g., a low priority requirement may be sacrified in order to meet a higher priority one to a satisfactory extent), as well as those among alternative implementation techniques (i.e., alternative technologies; for example, a given storage device can provide such a quick access that the software needed to achieve the desired response time can be simplified --but using such a device can mean higher cost).

When the system under consideration is not complex (meaning that there are not too many requirements, and only a few, well defined technology alternatives for its implementation), the associated design problem is of moderate size, so that formulating and solving it in its entirety, at once, may be possible.

When system complexity increases, however, such a global design approach is no longer appropriate. System requirements and alternative implementation techniques become very numerous, and, consequently, trade - offs among them are difficult to formulate and consider in their entirety. In a sense, the designer is faced with a cognitive, or perception, problem: it is very hard to keep all relationships or trade - offs among

design variables in mind, at once, in order to conceptualize the structural characteristics of the design problem at hand. Some kind of "problem formulation framework" is needed to explicitly lay out the designer's perception of the problem. In addition, a "solving procedure" is also needed because the problem can be so large that available "computational" techniques may prove to be insufficient. As an analogy, the situation is not dissimilar from that arising with large scale mathematical programming problems: formulating them in an optimization framework is an important step, but the limitations of available computing facilities often require having to "be clever" about the solving procedures, for a straightforward application of traditional techniques may result in a hopelessly time consuming process. Exploiting what is often called the "problem structure" results in a series of decomposition techniques that improve the efficiency of available computational facilities considerably. This analogy points out an important feature with which we will be concerned later, namely, the fact that effective solving procedures have been devised by means of exploiting the problem structure, as opposed to adapting traditional solutions to the problem at hand.

At the root of the complex system design problem, thus, there is a need for simplification, in the form of a formulation framework and a solving procedure. Problem simplification in this sense has been employed, explicitly or implicitly, in the design of any complex system, for otherwise

it couldn't possibly have been designed. The issue becomes more one of identifying a satisfactory simplification scheme. Little attention has been paid to the explicit consideration of this issue: it is often taken care of in an ad - hoc fashion. For instance, it is not uncommon to take a given, predefined implementation technique for granted and organize the design around it, so that, to some extent, the problem is forced to fit the solution. A more systematic approach is needed to avoid this kind of unjustified, ad - hoc design strategy.

2.2.- A simplification scheme: The decomposition concept.

Since, in order to identify a "formulation framework", we need to focus on a simplification approach allowing the exploitation of any special structure found in the design problem, the decomposition concept comes to mind. Briefly stated, this concept involves the following idea: Is there a way of breaking the system of interest down into parts or subsystems such that they can be attacked almost independently of one another for design purposes and which, once designed, can be combined in a well defined manner to form the overall system? In other words, is there a way of decomposing the design problem into smaller, more tractable subproblems whose solutions can then be put back together to generate the solution to the overall problem? This decomposition idea should be carefully considered in its own right, for not any arbitrary system break - down will work. In particular, the decomposition process should end up with a framework consisting of a collection of subsystems such that they:

(a) Are "loosely coupled" among themselves,

(b) Are internally coherent,

(c) Display the intrinsic overall system structure as perceived by the designer, and

(d) Are independent of any specific technology or implementation technique.

Condition (a) must be met if we wish to be able of attacking the design of each subsystem as independently as

possible of others. Condition (b) is important in order to obtain "self contained" design problems for each subsystem. Condition (c) is needed to achieve meaningful coordination among the designs of the different subsystems. Condition (d), finally, is central to avoid technology biases, to avoid overlooking potentially relevant technologies, and to be capable of considering the incorporation of new technologies.

Section 4 focuses on this decomposition concept and proposes a methodology cognizant of the conditions just outlined. These conditions will determine not only the "shape" of such a methodology, but also the basic rules for its application; in particular, condition (d) is determinant of the point in time, in the design process, in which it should be applied --in sharp contrast with traditional approaches.

A decomposition strategy usually results in a hierarchical framework for the system under consideration, derived from the way in which system parts are combined to form the overall system (Fig. 1). It is not clear whether hierarchical structure is intrinsic to most complex systems or it is just a convenient scheme that helps our cognitive abilities in order to understand them better; this philosophical question is not of concern to us here. More pragmatically, it is interesting to note that hiererchical structure has been employed effectively to cope with the complexity problem in a variety of settings (see [Pattee 73], [Simon 67], [Mesarovic et al. 70]); more importantly, it has proven effective with several <u>software</u> systems (see [Madnick and Alsop 69], [Dijkstra 68], [Madnick

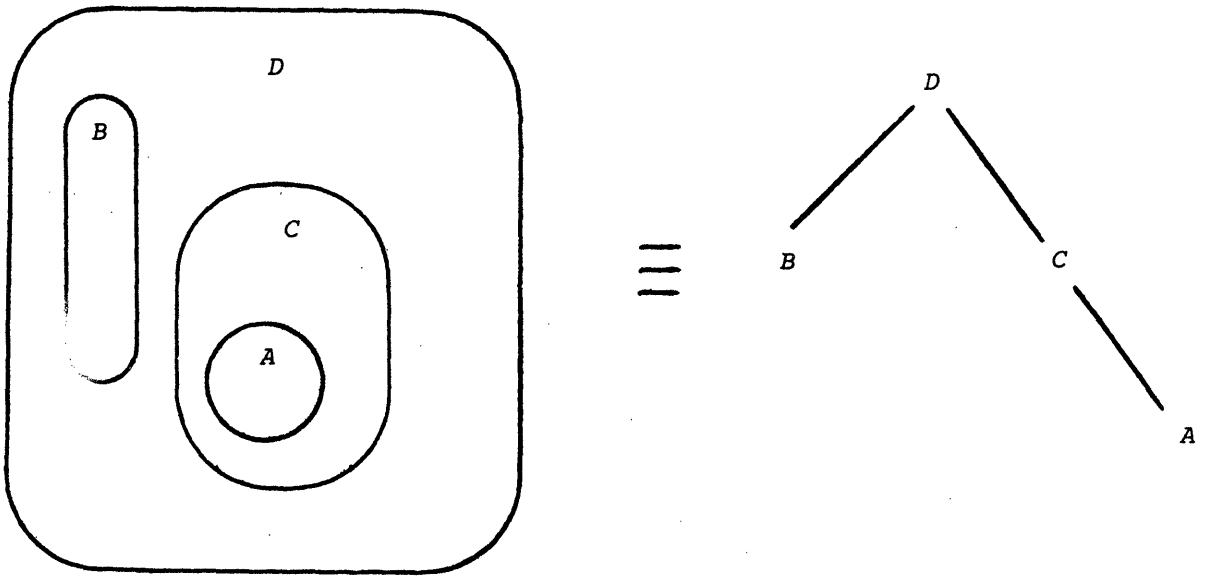


Fig. 1

and Donovan 74], [Madnick 76]). However, the nature of the hierarchy has not always been explicitly recognized ([Parnas 74]).

2.3.- Advantages of a hierarchical decomposition scheme.

Organizing the software system development process around a hierarchical framework with the characteristics discussed above can be an effective strategy to cope with the type of problems noted in section 2. In particular, software development costs can be more effectively controlled if good coordination is achieved among the design decisions that must be made, so that they can be approached as independently of one another as possible and only the truly critical interdependencies considered in the overall design; the difficulties found in system repair or modification can be decreased if the impact of such activities are confined to well defined parts of the system; being more responsive to user requirements is facilitated if the system can be made really compatible with others that might meet specific requirements well. These ideas are discussed in some detail below.

(1) The goal of achieving good coordination among different design decisions is facilitated when such a framework is available. Currently, achieving this goal is made difficult by the fact that although several in - depth analyses of, say, alternative implementation techniques may exist (see [Cardenas 75], [Lum et al. 71], [Rothnie 72], [Severance 75], for examples in the DBMS field), they tend to focus only on parts of the design problem, so that some way of putting them together in the context of the overall design is needed. Of course, a possible strategy to do so is to analyze all the

relevant design decisions at the same time in a unique analysis. Unfortunately, this is not practical with complex systems, both for the reasons discussed above and because available analysis methodologies often pose constraints to the scope of the problems that can be attacked: if an analytical method is used, a single model including all the relevant system "parameters" is likely to be mathematically intractable (see [Sekino 72]); if a simulation approach is taken, attempting to simulate operations with very different time scales becomes prohibitively expensive (see [Blum 66]). Alternatively, a hierarchy of models has proven to be a sound solution to this problem ([Sekino 72], [Hax 75]). Typically, each hierarchy level is analyzed separately and the different analyses coordinated by using the results of one of them as input to the next one down the hierarchy. A hierarchical framework should allow the coordination of different design subproblems in a similar fashion.

(2) If the selection of the framework's hierarchical levels allows the isolation of design and operating problems into specific subsystems, their impact in the entire system will be avoided: The affected subsystems can be identified, so that only those need to be reconsidered in order to respond to shifts in operational needs and/or study the impact of adopting new technology. Since the strategy proposed in section 4 below is centered around decomposing a set of system requirements which reflects the operational and design needs, we have reasons to believe that it will generate a framework with those

properties.

(3) Designing systems compatible with one another is usually a matter of concern, too. When systems are organized as a hierarchical arrangement of subsystems, a "family" of compatible systems can be defined ([Parnas 76], [Madnick 76-b]). Different "members" of such a family are easily conceptualized: While the basic structure is the same for all of them (i.e., that dictated by the framework), the specific subsystems that are combined to form a particular member can vary from one member to another. In general, they may vary in terms of(i) functionality, (ii) performance, or (iii) existence. In the DBMS field, for example, a number of alternative schemes can be employed to organize the physical files' layout on a storage device (sequential, indexed, hashed, etc.). For any given organization (e.g., indexed), there are a variety of possible algorithms (e.g., ISAM, B-trees) that may be used and which vary in terms of performance (e.g., execution speed, amount of extra storage required, etc.). Also, it may be the case that particular subsystems are not needed at all: the corresponding family member is then made smaller and simpler.

3.- Previous related work.

The system decomposition concept sketched above is not new and has, in fact, been used rather extensively in software engineering (see, for instance, [Myers 75]). The "modularity" characteristic of most large scale software systems is really a version of the same concept at the implementation stage. However, the decomposition strategies employed have failed, in general, to formally consider the conditions outlined in section 2.2.

Myers ([Myers 75]) has been cognizant of these conditions, but only at the implementation stage: he has suggested the terms "module strength" and "module coupling" to characterize, respectively, conditions (b) and (a), and has proposed qualitative measures to evaluate them while devising software modules. Delaying the explicit consideration of those two conditions until the system implementation phase, however, leaves condition (d) unsatisfied, as the extent to which the former are met depends strongly upon the kind of technology employed: indeed, the techniques suggested by Myers to achieve "loose coupling" and "internal module strength" take the form of implementation strategies.

Condition (d), technology independence, has been emphasized in the early phases of system design by means of focusing on system requirements, postponing the consideration of any implementation techniques. This is consistent with our goals, but it has been employed only to ensure that the set of

requirements specified for the system at hand is "complete" (i.e., to avoid overlooking requirements) and "consistent" (i.e., to detect apparent contradictions in the requirements' set): The so called "Problem statement languages" (see [Teichroew 70]) are used precisely for these purposes. Our view is that something else should be done with such sets of requirements, namely, inferring from them a system structure upon which a system decomposition can be identified.

Decomposition for design has also been employed in the past. It is not uncommon to describe the system under analysis as a collection of different parts. Such parts, however, are ususally identified by way of superimposing a preconceived structure on the system. Typically, this structure has to do with the physical organization of the system (e.g., programs, files, etc. --see [Rhodes 72]--), or with existing implementation techniques. No attempt is made to justify the superimposed structure in any way. Consequently, the result is often an "artificial" system decomposition, where the intrinsic system structure is distorted to match that of the predefined, arbitrary and superimposed structure.

Arranging and coordinating design decisions or performance analyses in a hierarchical framework has also been suggested (see [Nunamaker 71], [Sekino 72]), but, again, the hierarchy employed has been arbitrarely chosen. The same basic idea has been used in the design of non - software systems (see [Hax 75], [Gabbay 75]), the hierarchical framework being identified more or less intuitively, so as to reflect "traditional" or

"natural" system structure. Such approaches are not particularly well suited to our problem: traditional software systems' structure has the drawback of being typically biased around specific implementation techniques, while "natural" system structure, that can be a strong guideline for well established systems (e.g., production planning systems) is weaker in our case because most software systems are so new that they have not generated a "classical" system organization of that nature yet.

It is apparent, thus, that several of the conditions for decomposition set forth in section 2.2 above have been employed at some point in the past. They have not been used concurrently, though, but rather on a one at a time basis. This probably reflects a traditional view of the software system development process, in which three main steps are identified ([Young and Kent 74]): "analysis", "programming" and "coding". Analysis is concerned with determining "what is to be done" (i.e., identifying requirements); programming is concerned with "how to do it" (i.e., algorithm design, thus bringing technology into play); coding with "translating the programming output into machine language". The direct passage from analysis to programming in the above sense causes implementation techniques to be brought into play prior to making any attempt to infer, at the analysis stage, the intrinsic structure of the system at hand: the result is that the eventual system structure is determined to a large extent by the implementation techniques employed.

In a sense, the strategy proposed in section 4 suggests postponing the programming step --as defined above-- by introducing an intermediate one aimed at inferring system structure from the set of system requirements; i.e., at explicitly organizing the design problem in a form consistent with the designer's perception of the system requirements and trade - offs.

4.- A strategy for system decomposition.

As discussed in section 2.2, our goal is to devise a methodology for system decomposition meeting the conditions specified there.

Condition (d), technology independence, calls for postponing any consideration regarding possible implementation techniques, in order to avoid technology biases. A way of attaining this goal is to restrict that methodology to the exclusive consideration of system requirements (i.e., specifications stating how the system is to behave, but independently of how this behavior is to be achieved).

Alexander ([Alexander 64]) has proposed an approach whose main emphasis is consistent with that goal. The idea is to work with a set of system requirements, which is given structure by means of assessing interdependencies among its elements. These interdependencies aim at reflecting the designer's perception of requirements' trade - offs, in the following sense: two requirements are said to be related when the designer can think of any way in which (i) the two can be met simultaneously, or (ii) doing something to meet one is likely to jeopardize the extent to which the other can be met, or vice versa. More intuitively, such interdependencies make explicit the designer's "view" of the system of interest: they show what requirements ought to be considered at the same time for design purposes, if we are to avoid unbalanced designs.

Once this kind of structure is given to the requirements'

set, a system framework can be derived by decomposing that set into subsets whose elements are strongly related within a given subset, while the interdependencies among the elements of different subsets are kept to a minimum, thus satisfying conditions (b) and (a) of section 2.2: The requirements in each one of these subsets will define a subsystem; the interdependencies among subsets will point out how these subsystems interact in order to perform the desired system functions.

Partitioning the requirements' set in this way can be formulated as a graph decomposition problem, where nodes correspond to requirements and links to interdependencies. In the graph decomposition problem, conditions (a) and (b) of section 2.2 can be explicitly formalized, so that the resulting subgraphs(corresponding to subsets of requirements) are both loosely coupled and internally coherent. If no explicit assumptions regarding technology are made while establishing requirements nor while assessing interdependencies, condition (d) will also be met. Meeting condition (c) becomes a matter of interpreting the eventual partition (i.e., of giving intuitive meaning to each of the subsets). This may imply a reformulation of the initial graph in the case that the obtained partition points out any inconsistency which can be corrected: in this sense, the process will become iterative in nature and will allow interaction on the part of the designer, so that his intuition and/or previous experience can play an important role.

4.1.- Research activities.

In order to make operational the decomposition strategy outlined above, several research activities must be undertaken:

(1) Identify a set of technology independent requirements that faithfully represent what is expected from the system under study.

(2) Investigate a systematic way of assessing interdependencies among pairs of requirements in that set.

(3) Formulate the decomposition problem as a graph decomposition one. Identify appropriate graph decomposition techniques allowing the explicit formalization of conditions (a) and (b) described insection 2.2, with emphasis on "robust" techniques that avoid drastically different decompositions when applied to slightly different graphs.

(4) Solve the graph decomposition problem. Analyze the solution in the context of the original requirements' set; i.e., give an interpretation to the obtained subgraphs, whose nodes will represent subsets of requirements defining subsystems and associated design subproblems. The output of this activity will be the required system framework.

(5) Analyze the design of each subsystem identified in (4) and investigate solving procedures for the associated design subproblems; study the coordination of these subproblems in the context of the overall framework.

An additional step may be taken or at least explored:

(6) Analyze each subsystem from a performance evaluation

viewpoint. Propose models for each of them and study their combination into a global performance evaluation model.

These activities are discussed in more detail below.

4.1.1.- Set of DBMS requirements.

The decomposition methodology must be robust regarding changes in the basic requirements' set and interdependencies. Thus, it is not necessary, at the outset, to define a unique set of requirements: different sets may be decomposed and the results compared in order to draw conclusions regarding both the "best" (i.e., complete and consistent) set and the most appropriate decomposition technique.

For a particular system, sets of requirements can be found in the literature. These can be taken as a starting point for our purposes.

4.1.2.- Interdependencies' assessment.

The interdependencies' assessment activity, as originally proposed by Alexander, was to be a purely subjective one, on the designer's part. We plan to devote a considerable amount of effort to identifying a less subjective, more structured assessment procedure. One possibility is to view requirements as design specifications involving a number of system "attributes" or "characteristics" (for example, "storage cost" and "processing cost" could be attributes, while the

requirement "minimize system cost" would involve those two attributes, and possibly some others): a measure of the interdependency between two requirements can then be put in terms of the number of common attributes. This would represent a significant improvement over the approach proposed by Alexander, on two counts: (i) the assessment process is more structured, and (ii) different interdependencies can have different importance, i.e., different links can have different "strength" in the graph formulation. Alexander's graph link structure was such that all links had the same strength, which is not realistic.

4.1.3.- Graph decomposition techniques.

The graph decomposition technique eventually used must allow us to explicitly formalize conditions (a) and (b) of section 2.2. Alexander proposed one such decomposition technique; however, it requires making a series of assumptions regarding the basic nature of the graph. It is not always easy to show that a given graph possesses the properties implied by these assumptions. Furthermore, the technique is only appropriate for graphs with <u>one</u> type of links (i.e., all links must have the same strength). We plan to work on the identification of a more general decomposition technique that can be applied to graphs with more than one link strength and which doesn't require making such strong assumptions.

Note that although we may have an intuitive feeling for

what conditions (a) and (b) of section 2.2 mean in the context of a graph (for example, the graph of Fig. 2-a should probably be partitioned as indicated in Fig. 2-b, assuming that all the

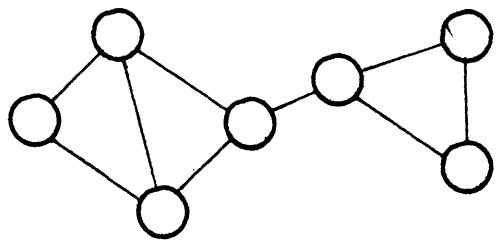


Fig. 2-a

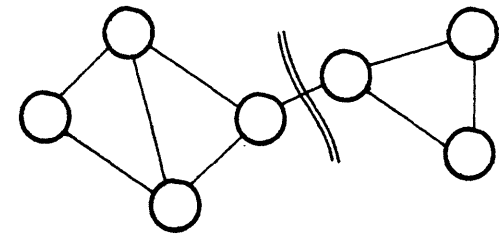


Fig. 2-b

links have the same strength), intuition is not enough when the graph is more complicated, as it will be exemplified in section 5.3.3.

4.1.4.- Identification of a DBMS framework for design purposes.

Once a graph decomposition technique of the characteristics suggested above is available, its application to a representative set of system requirements will result in a decomposition of that set into subsets that will define a system framework: The identified subsets will specify the main components of the system under analysis, while the relationships among them (in terms of links joining the subgraphs) will provide insight as to how the different subsystems interact. It is apparent that this activity, which is responsible for an "intuitive interpretation" of the

decomposition obtained, may involve interaction with previous ones, particularly with those in 4.1.1 and 4.1.2: The partition can point out inconsistencies in the initial requirements' set that may have to be corrected, so that the process (activities (1), (2) and (4) of section 4.1) will be repeated.

4.1.5.- Design subproblems; definition and coordination.

The overall design problem can at this point be analyzed in the context of the framework. The design of each subsystem will generate a design subproblem. These subproblems should be analyzed, and solving procedures investigated and coordinated as dictated by the framework. Interaction with preceding activities may also be required: it is conceivable that the available analysis techniques can suggest minor modifications on the framework, in order to facilitate their use.

4.1.6.- Performance evaluation.

System performance evaluation can conceivably be approached also in the context of the framework identified as discussed above. The idea is to investigate the possibility of coordinating performance evaluation models for the different subsystems in order to obtain an overall performance evaluation model. The outcome would be a hierarchy of models, each focusing on a specific subsystem. This has the advantage that the most appropriate modeling technique can be chosen for each

subsystem (e.g., simulation can be very convenient for some of them, while an analytical approach can be more adequate for others). For reasons similar to those in the preceding point, this activity can also involve interactions with the preceding ones.

* * *

It is the presence of interactions among these activities what makes our approach iterative and interactive on the designer's part. The basic nature of these interactions among research activities is summarized in Fig. 3.

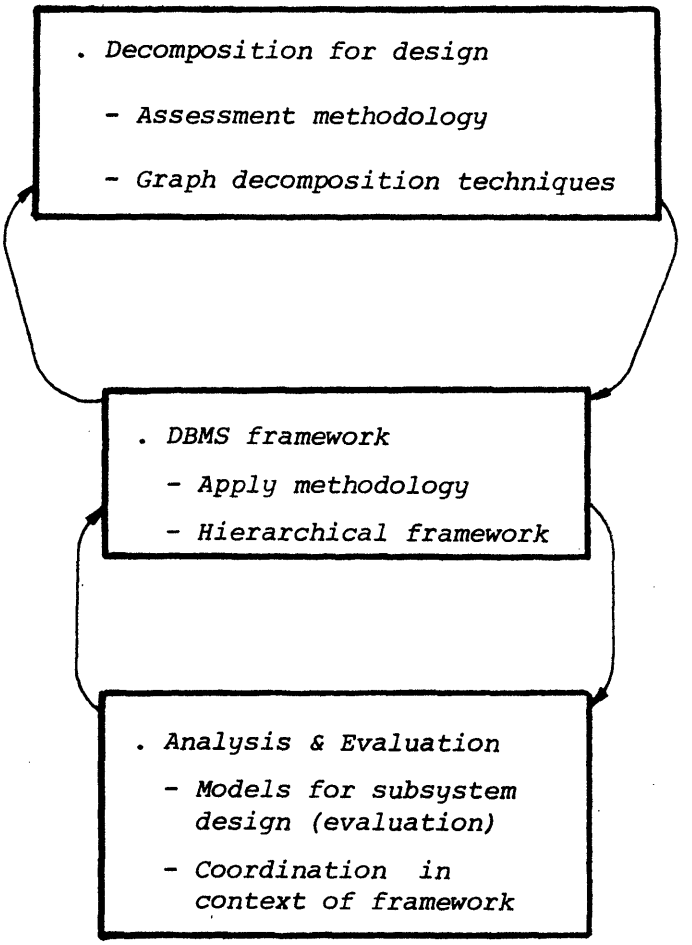

. Decomposition for design
- Assessment methodology
- Graph decomposition techniques
. DBMS framework
- Apply methodology
- Hierarchical framework
. Analysis & Evaluation
- Models for subsystem design (evaluation)
- Coordination in context of framework

Fig. 3

5.- The DBMS case.

We now illustrate how the system decomposition strategy proposed above can be applied to a specific case. Since Data Base Management Systems (DBMSs) are currently becoming increasingly important for most data processing applications, we choose them as a representative example of complex systems for which such a strategy can be very beneficial. Illustrative in nature, the discussion below does not pretend to be a definitive analysis; rather, its purpose is merely to show that the proposed approach is feasible and very promising.

5.1.- The nature of DBMSs.

To bring the subject into focuss, we devote this section to the discussion of the basic nature of DBMSs.

A DBMS can be defined as the software facility that plays the role of intermediary between a computer system and its users, with the goal of providing convenient and efficient data manipulation capabilities. The situation is schematically depicted in Fig. 4.

The term "computer system", here, is intended to mean the combination of hardware and basic software (such as Operating Systems) typically found in a computer installation. "Users" are people who use the computer system only as a tool in their problem solving activities; in particular, they are not interested in the specific techniques required in order to make

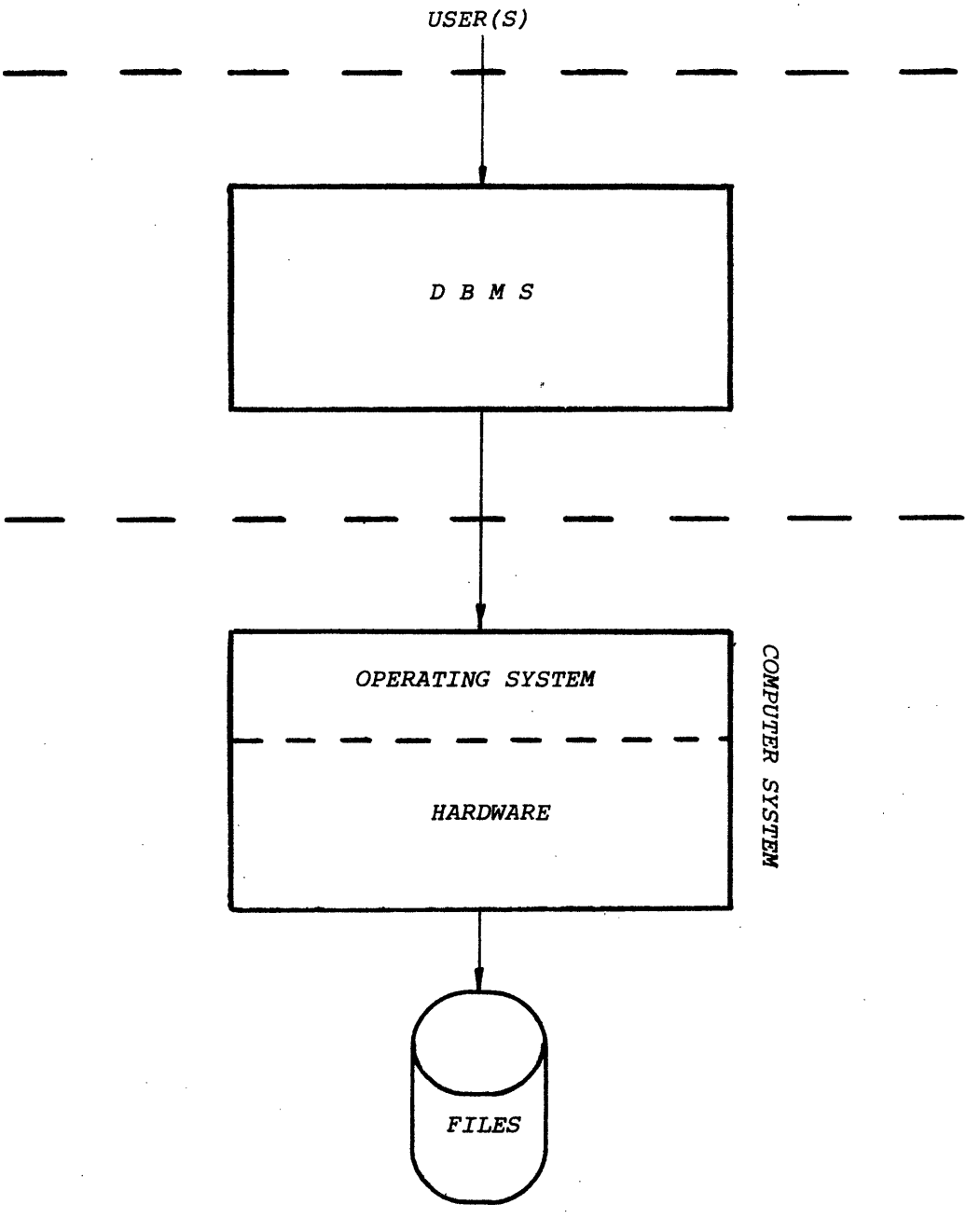

USER(S)
D B M S
OPERATING SYSTEM
HARDWARE
COMPUTER SYSTEM
FILES

Fig. 4

the computer system perform the operations they need. Yet, they are concerned about efficiency (e.g., obtaining results within a reasonable time interval). In this sense, the words "convenient" and "efficient" in the definition above are both relevant. Although Fig. 4 depicts clear separations between the DBMS and either users or the computer system, it should be understood that such boundaries are a function of the specific users, whose degree of sophistication varies across applications, as well as of the capabilities of the specific computer system, different Operating Systems may be available, for instance.

For concreteness, we present a simple example. Assume that a bank operates a computer system where information about accounts and clients is stored as follows: Accounts' information (e.g., account number, type, balance) is stored in one file while clients' information (client name, address, associated accounts' numbers) is stored in a separate file (e.g., as shown in Fig. 5).

If a DBMS is not available, if a user wanted to determine the balance in, say, John Doe's savings account, it would be necessary to interact directly with the computer system. However, this may not be particularly convenient. For example, he may need to write a program that would (see Fig. 5): (a) search the clients' file for the client of interest, (b) retrieve the associated accounts' numbers, (c) search the accounts' file for these accounts, (d) select the savings account, and (e) retrieve the associated balance. This implies

ACCOUNT#	TYPE	BALANCE
1100	Checking	203
2100	Savings	1500
.	.	.
.	.	.
.	.	.
2305	Savings	3400
.	.	.
.	.	.
.	.	.

ACCOUNTS FILE

NAME	ADDRESS	ACCOUNT#'s
Jim Adams	Weston	1100
Sue Black	Newton	2100
.	.	.
.	.	.
.	.	.
John Doe	Boston	2305
.	.	.
.	.	.
.	.	.

CLIENTS FILE

Fig. 5

that the user must employ techniques and procedures which have nothing at all to do with his original problem. Moreover, his solution may not be particularly efficient: For example, he may overlook the fact that the clients' records are stored in alphabetical order in the clients' file, so that a binary search algorithm would speed up processing. Also, another user may have a similar problem and devise his own program, thus incurring duplication of effort.

A DBMS makes the user's task much easier. A typical DBMS would allow him to issue the following "non procedural" command, describing the information he needs, as opposed to writing a program:

```
SELECT BALANCE FROM ACCOUNTS
       WHERE ACCOUNT_NUMBER =
             SELECT ACCOUNT_NUMBER FROM CLIENTS
             WHERE CLIENT_NAME = 'JOHN DOE'
             AND   ACCOUNT_TYPE = 'SAVINGS'
```

A schematic comparison of the two procedures is depicted in Fig. 6:

When a DBMS is not available (left hand side in Fig. 6), the programs written by the user(s) may become inadequate if the files are changed (broken line boxes in Fig. 6), thus requiring appropriate user action.

A DBMS is very convenient: it allows the user to deal exclusively with entities and operations akin to his problem (accounts, balances, clients), instead of with computer oriented ones (files, records, algorithms). Further, the DBMS is made responsible for efficiency: the command above, for instance, specifies only <u>what</u> is to be done; the system will

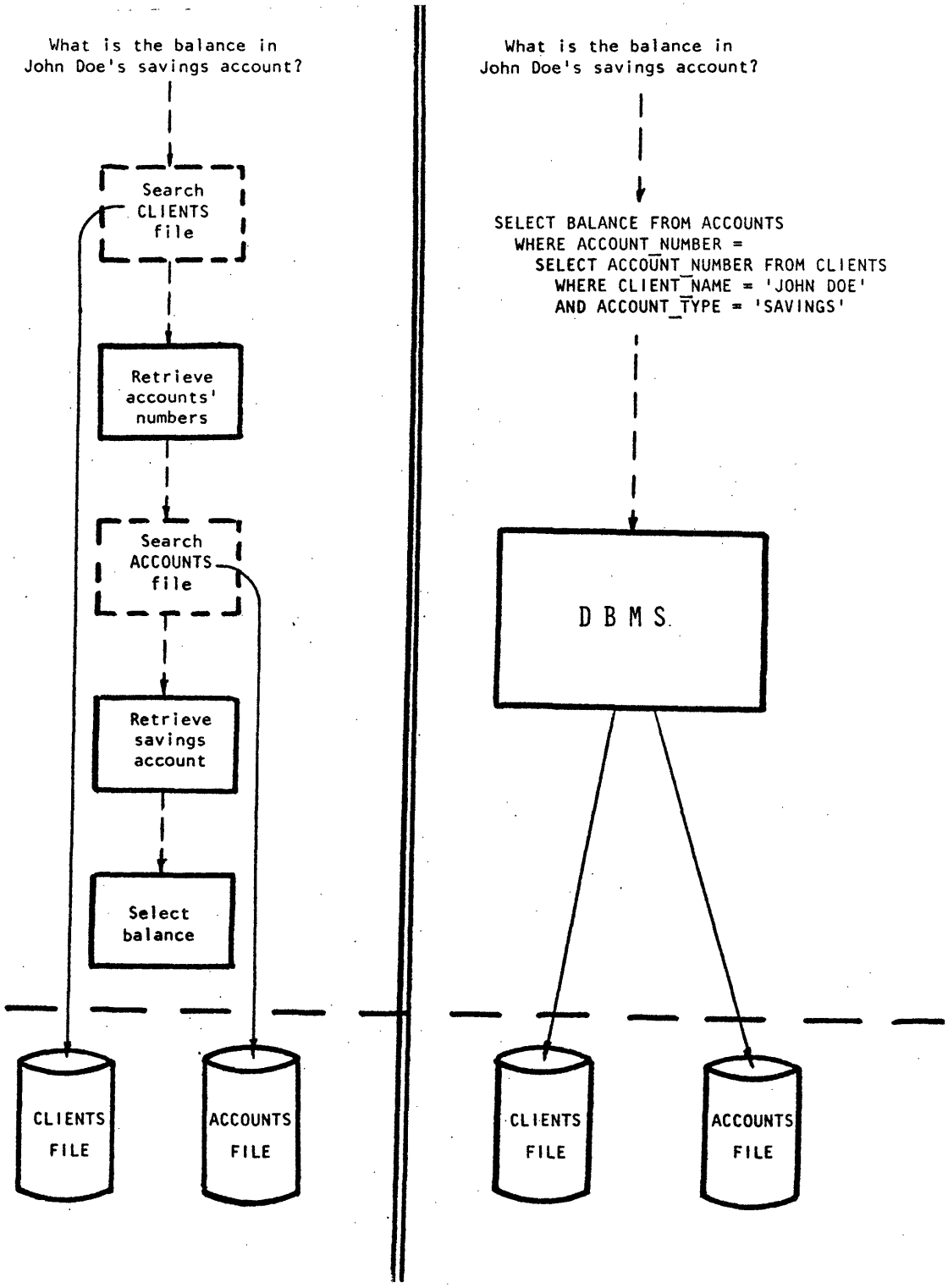

What is the balance in John Doe's savings account?
Search CLIENTS file
Retrieve accounts' numbers
Search ACCOUNTS file
Retrieve savings account
Select balance
CLIENTS FILE
ACCOUNTS FILE
What is the balance in John Doe's savings account?
SELECT BALANCE FROM ACCOUNTS
WHERE ACCOUNT_NUMBER =
SELECT ACCOUNT_NUMBER FROM CLIENTS
WHERE CLIENT_NAME = 'JOHN DOE'
AND ACCOUNT_TYPE = 'SAVINGS'
DBMS
CLIENTS FILE
ACCOUNTS FILE

Fig. 6

decide how to do it. It is possible to make a DBMS take full advantage of any available technique that can improve efficiency (although it is not always trivial to decide which this technique is). Finally, a DBMS can be made available to several users: processing is centralized and thus duplication of effort avoided.

Additional advantages are less apparent: For example, consider what happens if, for whatever reasons (e.g., an increase in the number of clients), the file organization is changed (e.g., binary search is no longer sufficient and the files are indexed). If a DBMS is not used, the users must be informed of the change, and take appropriate action regarding their programs. What is worse, they may have to learn new techniques (e.g., how to process indexed files). On the other hand, if a DBMS exists only it has to be changed to reflect the new file organization (and, under certain circumstances, only parts of it); the users need not even know that the files were reorganized, so that they can continue issuing the same problem oriented queries illustrated above.

This example was extremely simple, but it highlighted why DBMSs have been found to be so useful. In more complex data processing situations, DBMSs are also more complex, but they result in still more advantages. Currently, a number (ranging in the hundreds, see [Palmer 75]) of DBMSs are commercially available and extensively used. However, the fact that they have been developed with specific data manipulation problems in mind and in a rather ad-hoc manner (without any underlying

systematic methodology) often results in deficiencies in actual performance and/or user convenience. In the next section we discuss a few representative instances where this problem is apparent.

5.2.- Some typical DBMS drawbacks.

While many successful DBMS applications have been reported (see, for instance, [Palmer 75], [Nolan 73], [Donovan 75]), there are also several inconveniences that typically characterize DBMS operations. The purpose of this section is to illustrate their basic nature.

Most early DBMSs were ad-hoc, special purpose solutions to specific data manipulation problems found in specific situations. Since such systems proved effective, their application to other settings was encouraged, and in this sense made "general purpose". (For example, the original work that led to the DBTG approach to DBMSs [Bachman 69] was motivated by a bill of materials type of application). More recently, some effort has been devoted to the consideration of DBMSs from a broader perspective (see [Astrahan et al. 76], [Stonebraker et al. 76], [Senko et al. 73], for instance). However, even these broader efforts strongly rely upon the direct application of implementation techniques and/or system organizations whose origin is to be found in those same early ad-hoc DBMSs. In other words, it is often the case that a specific technique is used for its own sake. The result is very likely to be either an over-powered DBMS (with unnecessary overhead that can jeopardize performance significantly), or a DBMS which falls short of the required capabilities (with the consequent user inconvenience). Also, the peculiarities of specific implementation techniques become determinant of the eventual

system structure and functional capabilities, in such a way that apparently unrelated DBMS functions become strongly interdependent. Furthermore, alternative techniques are overlooked. These characteristics are typical of most complex software systems.

Several real life situations in which such characteristics are apparent are described in the DBMS literature. For example:

- The currently operational SEQUEL system ([Chamberlin et al. 74]) was designed to function "on top" of the previously developed systems RM and XRM (see [Lorie 74], [IBM 73], [Andreu 76-a,b]), whose original motivation was not general purpose DBMS processing, and which incorporate several techniques that were intended for a different purpose. As it turns out, the DBMS application makes no use of some of these techniques. Nevertheless, the system incurs the overhead derived from maintaining unneeded control structures.

- IMS ([IBM-a]), an IBM DBMS, incorporates alternative secondary storage access methods. It is up to the user to choose among them. Since there are no specific guidelines to support such a decision, it often becomes a matter of trial and error. It has been reported ([Palmer 75]) that in a specific setting performance improved three-fold by switching from one access method to another. The point here is that while the alternative method was available (and paid for), there was no systematic way of deciding when to use it.

- SEQUEL ([Chamberlin et al. 74]) also incorporates

alternative access methods. However, using one or another affects the functional capabilities of the system (e.g., the range of representable values varies with the access method employed: the maximum representable value is either 2**32 or 2**21). This is a direct consequence of its design being centered around implementation techniques originally developed for other purposes (see [Andreu 76-a,b]).

- IDMS ([IDMS], [Palmer 75]) has been found to perform in a way drastically dependent upon the query "mix" at any point in time. This is probably unavoidable, but it should be properly anticipated (i.e., what if the low performance query mix is the typical mix?).

- In general, different DBMSs are largely incompatible. This becomes an issue, for instance, when two departments in the same organization have been using different DBMSs and wish to integrate their data management applications (there can be significant organizational benefits derived from such a move). As it turns out, switching from one DBMS to another is a very time consuming task. As a consequence, DBMS users are often "locked in" to a specific system that may become inadequate because of shifting operational needs.

These types of problems are precisely what the methodology in this proposal attempts to solve: The availability of an implementation independent DBMS framework, in whose context the design process can be meaningfully organized and performance evaluation models coordinated, contributes significantly to their solution. The following sections illustrate this in some depth.

5.3.- Methodology application.

We now discuss how the methodology proposed in section 4 can be applied to the DBMS case in order to alleviate the typical problems summarized above. In particular, we exemplify the character of the research activities enumerated in section 4 as applied to the design of DBMSs, with emphasis in the identification of the needed DBMS framework. In order to obtain a specific framework that we can use to illustrate its usefulness in the design process, we actually carry out these activities in reduced scale, by means of a simple example.

5.3.1.- Set of DBMS requirements.

Several sets of DBMS requirements have been proposed in the literature (see, for instance, [Patterson 71], [Joyce et al. 74]). They are technology independent because they reflect user requirements (i.e., what is to be achieved is specified, not how). In our actual research, we plan to work with these sets as a starting point. For our purposes here, however, we have chosen a small set of requirements that is listed in Appendix A. To keep its size between reasonable limits, the requirements in this set are very general in scope, but still representative.

5.3.2.- Interdependencies among requirements.

As discussed in 4.1.2 above, we will investigate an assessment procedure at some depth in the future. For our illustrative purposes here, we intuitively assessed interdependencies among pairs of requirements belonging to the set in Appendix A. Pairs of requirements were seen as either related or unrelated, so that in the resulting graph all the links have the same strength. A brief justification for each of the assessments is presented in Appendix B.

5.3.3.- Graph decomposition techniques.

The graph corresponding to the data in Appendices A and B is depicted in Fig. 7. Although it corresponds to a relatively small set of requirements (a more detailed set can easily result in a considerably larger graph, maybe 10 or 20 times larger), there is no obvious "best" partition (as there was in the graph of Fig. 2-a in section 4.1.3). Therefore, a more formal technique is needed. A possibility is summarized in Fig. 8: Thinking in terms of the links that give structure to the requirements' set, a partition of this set can be evaluated by means of the measures called "subset strength" and "subset coupling". Basically, subset strength is a measure of how tightly coupled are the requirements in a given subset, while subset coupling attempts to measure the extent to which two subsets are related to one another. For the former, we can use

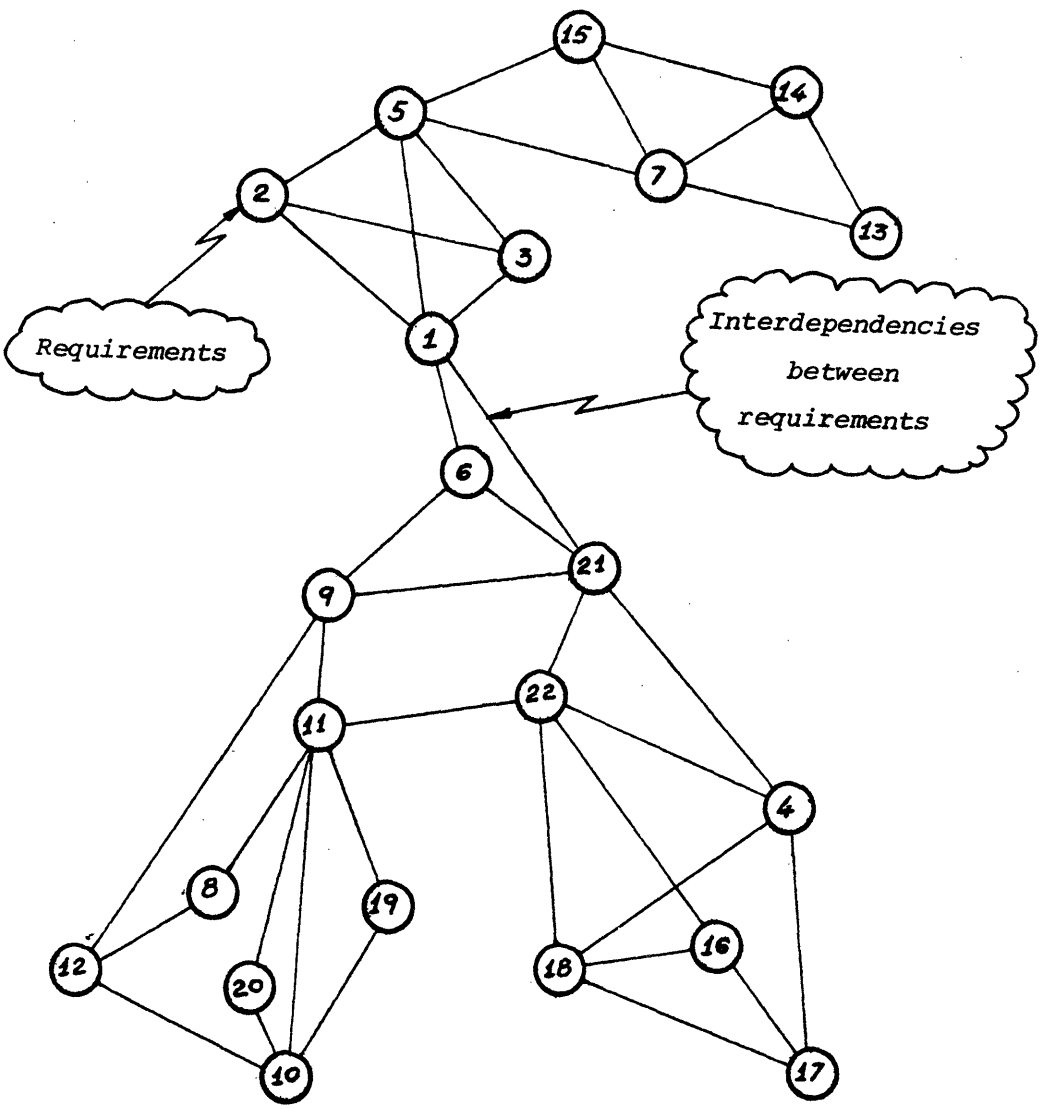

15
14
5
7
2
13
3
Requirements
1
Interdependencies
between
requirements
6
21
9
22
11
4
8
19
16
12
18
20
10
17

Fig. 7

- *Define a graph as a pair (X, L), where:*
  - $X:\{x \mid x = 1, 2, \ldots, n\}$, *the set of (n) nodes, and*
  - $L:\{l_{ij} \mid l_{ij}$ *exists iff a link joints* $\underline{i}$, $\underline{j} \in X$, $i < j\}$, *the set of links.*

- *Define:*
  - $A:\{a_{ij} \mid a_{ij} = 1$ *iff* l_{ij} *exists; 0 otherwise*$\}$.
  - *Strength* S_i *of a subset* X_i *of X,* $X_i \subseteq X$:

$$S_i = \frac{\sum_{k,l \in X_i} a_{kl} - (|X_i| - 1)}{\frac{|X_i| \cdot (|X_i| - 1)}{2}} \qquad \ldots\ldots (*)$$

  - *Coupling* C_{ij} *between two subsets* X_i *and* X_j *of X:*

$$C_{ij} = \frac{\sum_{\substack{k \in X_i \\ l \in X_j}} a_{kl}}{|X_i| \cdot |X_j|}$$

- *Define a partition of X, P as:*

$P:\{X_1, X_2, \ldots, X_p\}$, *with:*

$$\bigcup_{i=1}^{p} X_i = X \quad \text{and} \quad \bigcap_{\substack{i=1 \\ j=1 \\ i \neq j}}^{p} X_i , X_j = \phi$$

- *A partition of X, P can then be evaluated with a measure M:*

$$M = \sum_{i=1}^{p} S_i - \sum_{\substack{i=1 \\ j=1 \\ i \neq j}}^{p} C_{ij} \quad ,$$

to be maximized over all possible P's.

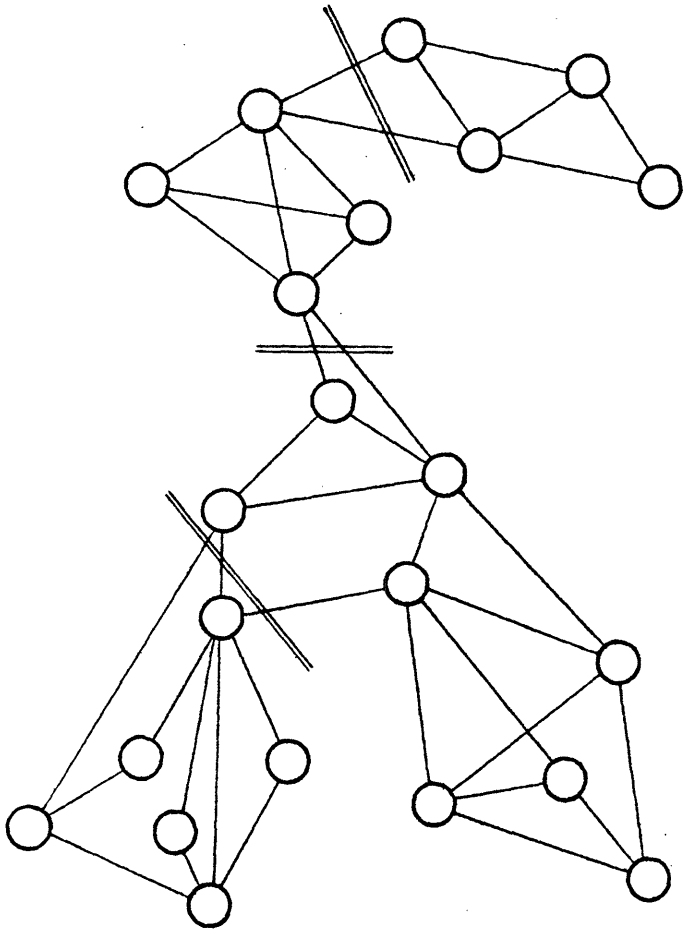
(*) $|X|$ *means the dimension of set X.*

Fig. 8

the number of links joining elements of the same subset, minus n-1 (n being the number of elements in the subset, since n-1 is then the minimum number of links that can form a completely linked subgraph with n nodes), normalized by the factor n(n-1)/2 (i.e., by the maximum number of links that may exist in a subset of dimension n) in order to obtain comparable measures for subsets of different dimension. A similar measure can be used to evaluate subset coupling between a pair of subsets: the number of links actually joining elements of two different subsets, normalized by the factor n*m (where n and m are the dimensions of the two subsets; i.e., normalized by the maximum number of links that may exist among elements of two subsets whose dimensions are n and m). Although these measures apply to graphs with only one type of link strength, they can be easily generalized to the case of links with different strengths. In addition, they don't require us to make any specific assumption regarding the graph itself.

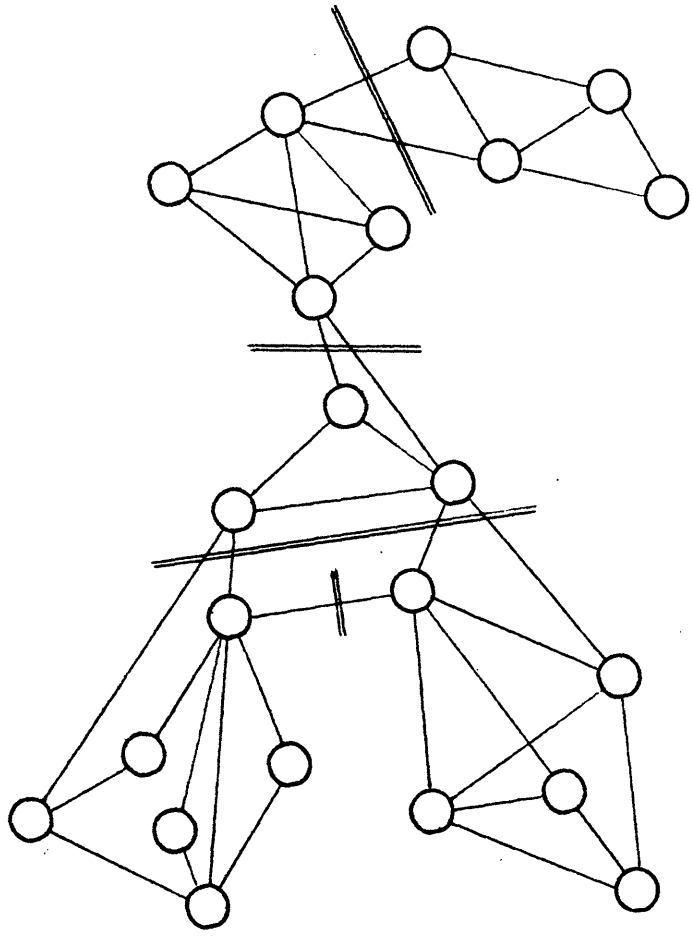
Four possible partitions of the graph in Fig. 7 are illustrated in Figs. 9-a, 9-b, 9-c and 9-d. Although they all seem to be intuitively appropriate (i.e., apparent "clusters" of nodes are put in the same subgraph, and care taken not to leave too many links joining different subgraphs), the associated measures differ significantly, thus reinforcing the need for a formal evaluation approach such as that of Fig. 8.

It should be pointed out, at this point, that alternative approaches exist to evaluate graph partitions. In particular, there are several "cluster analysis" techniques ([Hartigan 75])



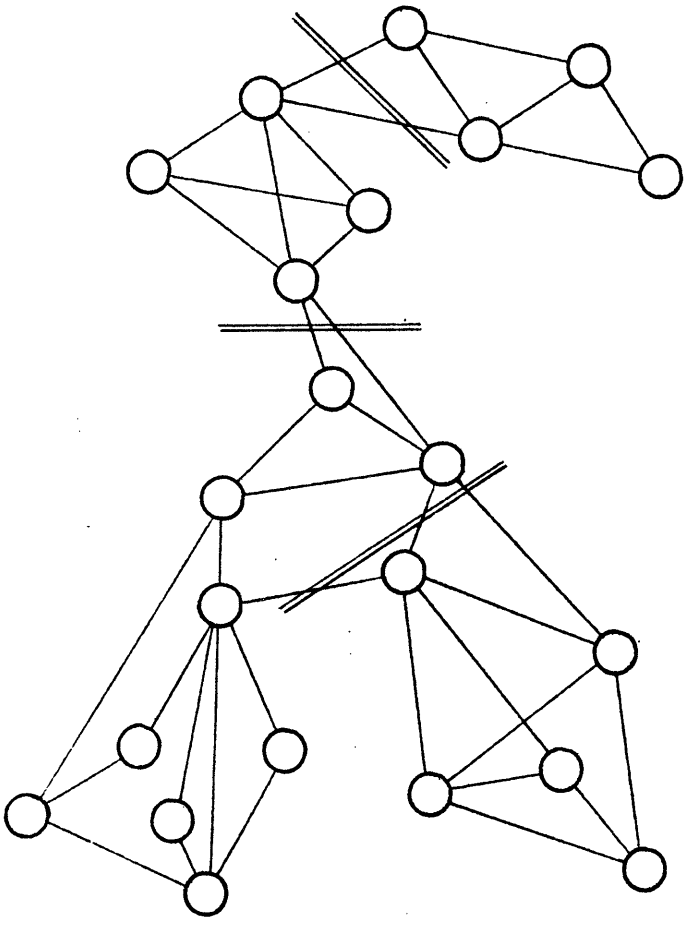
Strength: 1.211
Coupling: 0.250
Measure : 0.961

Fig. 9-a



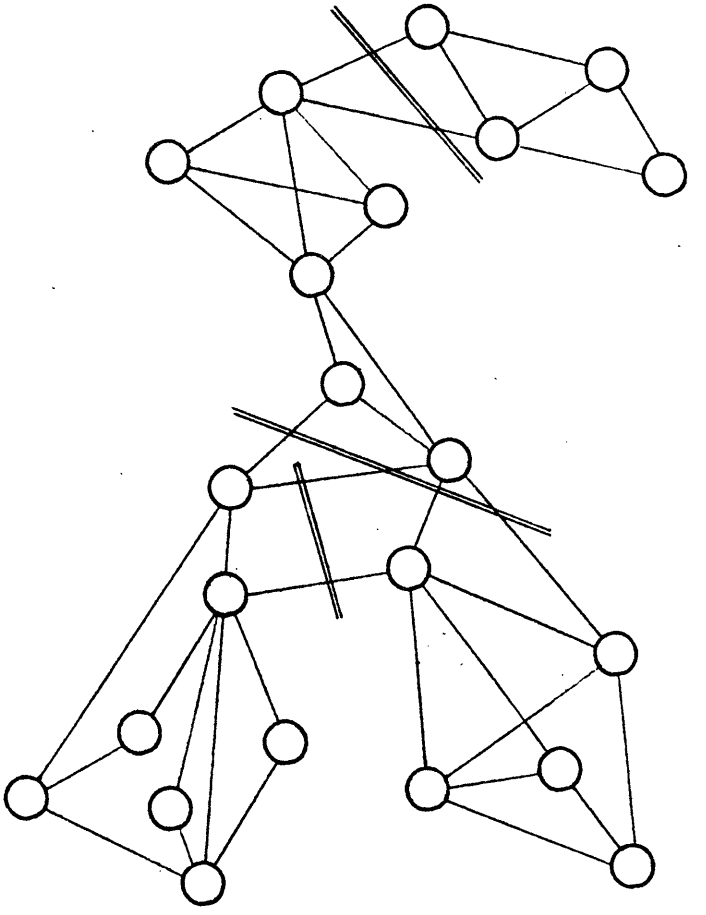
Strength: 1.666
Coupling: 0.569
Measure : 1.097

Fig. 9-b



Strength: 1.138
Coupling: 0.236
Measure : 0.902

Fig. 9-c



Strength: 1.090
Coupling: 0.242
Measure : 0.864

Fig. 9-d

that may prove appropriate in cases where the graph links don't have all the same strength, and that should thus be investigated in conjunction with the link assessment procedure suggested in 4.1.2 above. For example, the so called "leader algorithm" identifies clusters of elements of a given set such that the "distances" from all the elements belonging to a specific cluster to a cluster member known as the "leader" are less than some threshold value T; this algorithm views T as a parameter, that in our case can be used to study the sensitivity of the method to changes in the initial graph, so as to determine its robustness in the sense of section 4.1.1.

5.3.4.- Best graph decomposition. A DBMS framework.

The best partition of the graph in Fig. 7 according to the measures in Fig. 8 turns out to be that in Fig. 9-b. Looking back at Appendix A for the meaning of the requirements that ended up in each subgraph, each of these is seen to be associated with a main DBMS component; the subgraphs have been accordingly labelled in Fig. 10, that thus suggests a first DBMS framework.

In order to discuss this framework in some detail and to show its appropriateness, it is useful to redraw it as in Fig. 11, where its hierarchical structure is made apparent.

In what follows, we analyze the hierarchical levels of Fig. 11 from the point of view of the DBMS designer; in particular, we discuss: (a) the function of each level, (b) the

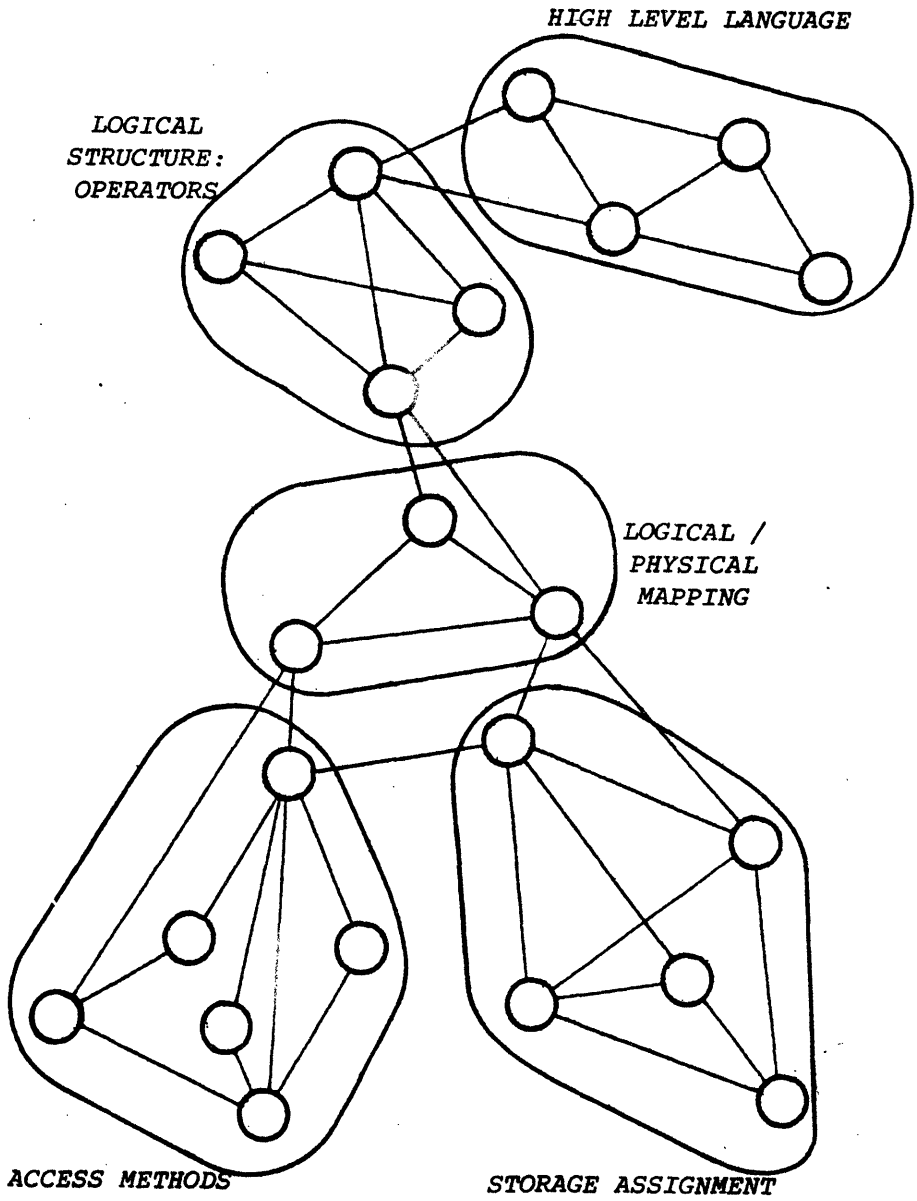

HIGH LEVEL LANGUAGE
LOGICAL
STRUCTURE:
OPERATORS
LOGICAL /
PHYSICAL
MAPPING
ACCESS METHODS
STORAGE ASSIGNMENT

Fig. 10

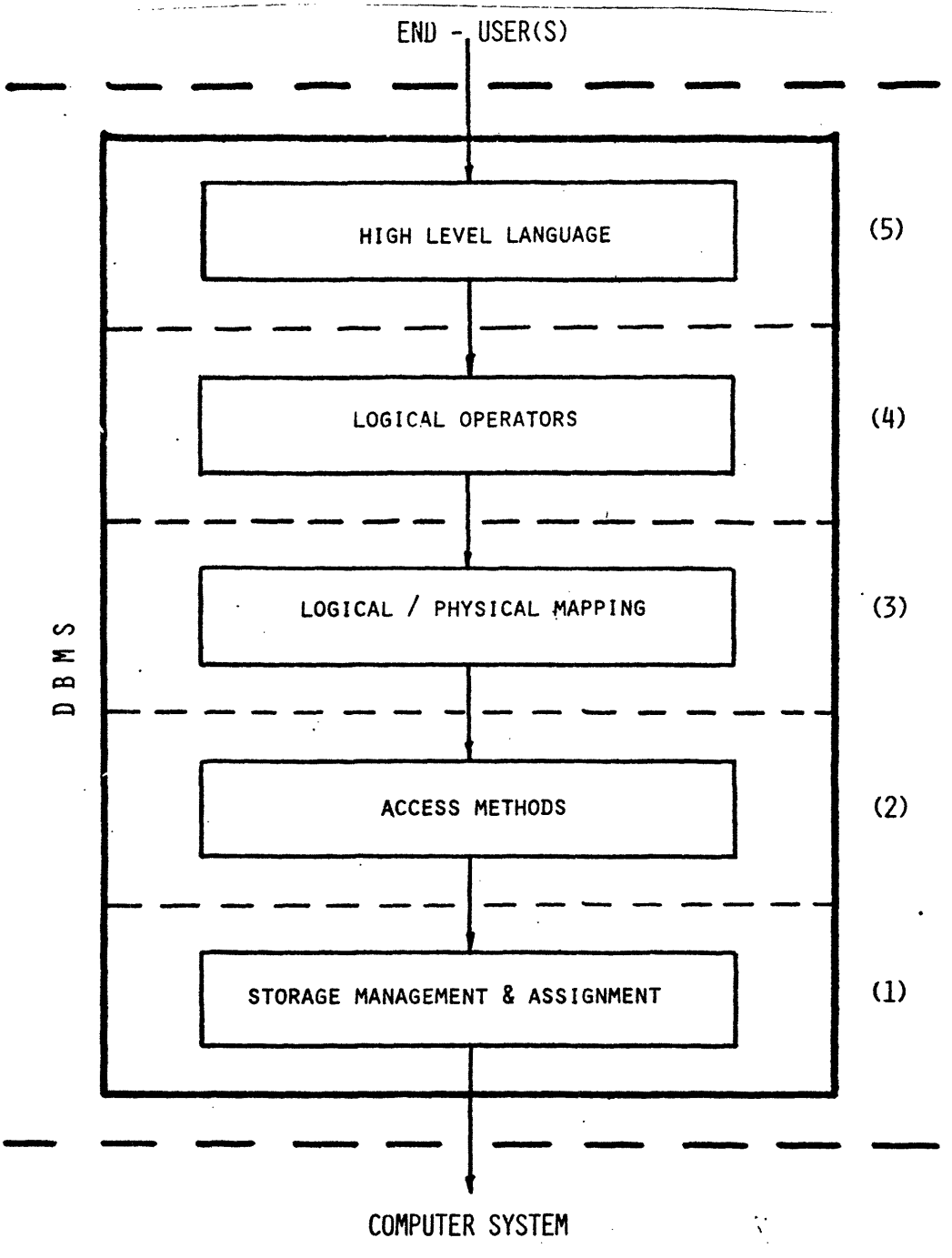

END - USER(S)
HIGH LEVEL LANGUAGE
(5)
LOGICAL OPERATORS
(4)
LOGICAL / PHYSICAL MAPPING
(3)
ACCESS METHODS
(2)
STORAGE MANAGEMENT & ASSIGNMENT
(1)
DBMS
COMPUTER SYSTEM

Fig. 11

interaction between functions, and (c) possible implementation techniques for each function, which need to be brought into play when designing the subsystems corresponding to each function. (An operational view of the framework is briefly presented in the next subsection, where the processing of a simple DBMS command through a system with the structure in Fig. 11 is described).

(1) The lowest hierarchy level is concerned with storage management and assignment (data representation in storage media). Since there are only a few alternative ways of representing real world entities in a computer, the most fundamental function performed by a DBMS is that of "encoding" those entities in some computer tractable form. In particular, the function at this level must take care of assigning storage space to the entities' representations. Once this is done, the location - oriented computer operations (e.g., read, write, etc.) can be used to manipulate such representations. For example, the real world entity "John Doe" (a person) might be represented as the character string 'JOHN DOE', and stored away as a string of bytes. Such a simple implementation scheme, however, may require a great amount of storage space if that entity is to appear many times in the data base. To cope with this kind of problems, several techniques have been developed. As an example, it is possible to store the character string only once, assign an internal "identifier" to it which requires less storage space, e.g., an integer, and represent the

remaining instances of that entity by means of this identifier (often called an "id"). Of course, this complicates this level's function since some mechanism to maintain the correspondence between an id and its associated character string must be provided (e.g., a function to transform the id into the address in storage where the character string is kept): in terms of implementation, thus, there is a trade - off, at this level, between required storage and processing speed.

This level must also take care of grouping entities' representations into storage areas (e.g., files), since the basic data manipulation operations provided by a typical computer system deal with such physical storage areas. There are also several techniques to implement such groupings: for example, entities' representations can be assigned consecutive locations in a storage area (Fig. 12-a), or they may be organized as a linked list within that area (Fig. 12-b). Alternative implementations of this kind achieve different degrees of efficiency for this grouping activity: For instance, if we are to delete entities from the sequential arrangement of Fig. 12-a, holes will begin to form in it, thus jeopardizing the contiguous property of both "free" and "used" storage space and so complicating their management; this problem is avoided with the linked list approach. However, the latter requires more storage space. Also, different storage areas may allow different accessing speeds (e.g., they may belong to different types of storage devices).

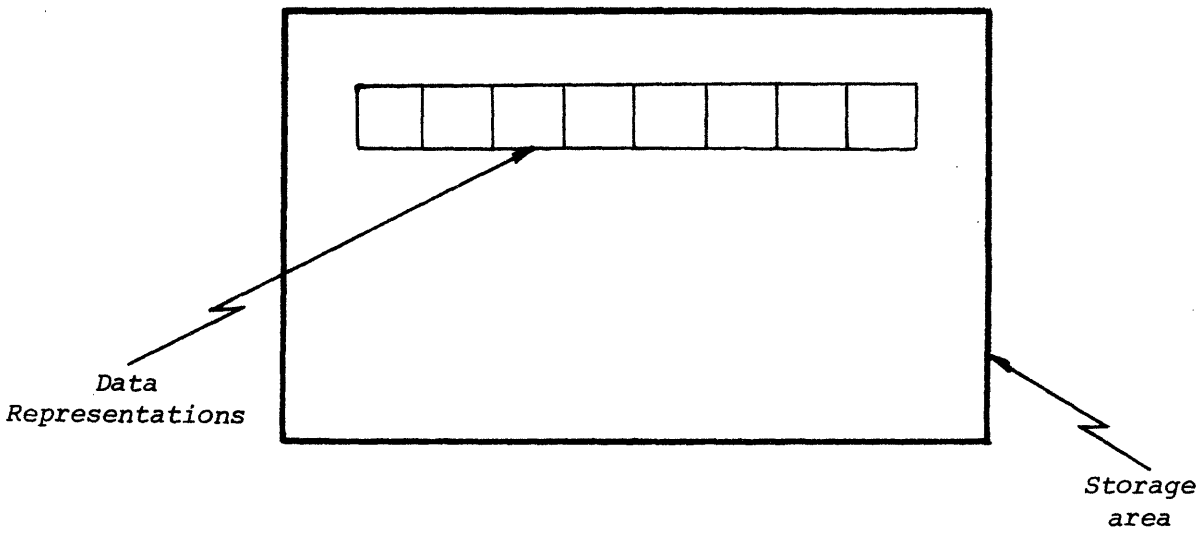


Fig. 12-a.

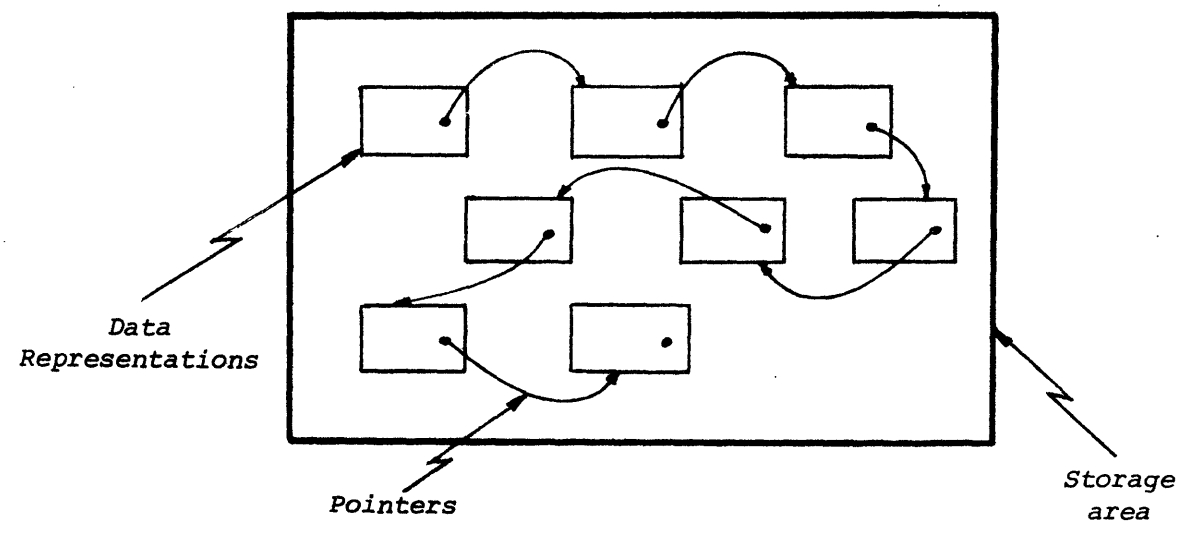


Fig. 12-b.

(2) Level 1, as described above, permits the manipulation of entities' representations by means of storage location - oriented operations (e.g., the "physical address" of the storage location containing a given entity must be specified in order to access that entity). The function at this level is concerned with providing more convenient accessing mechanisms. Since users tend to refer to entities by specifying some of their attributes (i.e., their values), this kind of entity references must be transformed by the function at this level (that we generically call "access methods") into storage location - oriented ones that can then be resolved by using the functions available at level 1.

There are a number of techniques to implement such a translation. Perhaps the simplest is an algorithm that sequentially scans the stored instances of a given entity (e.g., accounts) and selects those whose attributes match the specified values. Obviously, this algorithm can be very time consuming in a large data base. An alternative is to build an "index", as shown in Fig. 13. The index can then be scanned for matching attribute values, and information obtained about the storage location(s) containing the corresponding entities. Execution time is improved with this approach, but a penalty paid in terms of the storage space needed to keep the index, and also in updating time (i.e., each time an entity is changed the index must be updated accordingly).

In summary, the function at level 2 is responsible for transforming attribute - oriented references to entities into

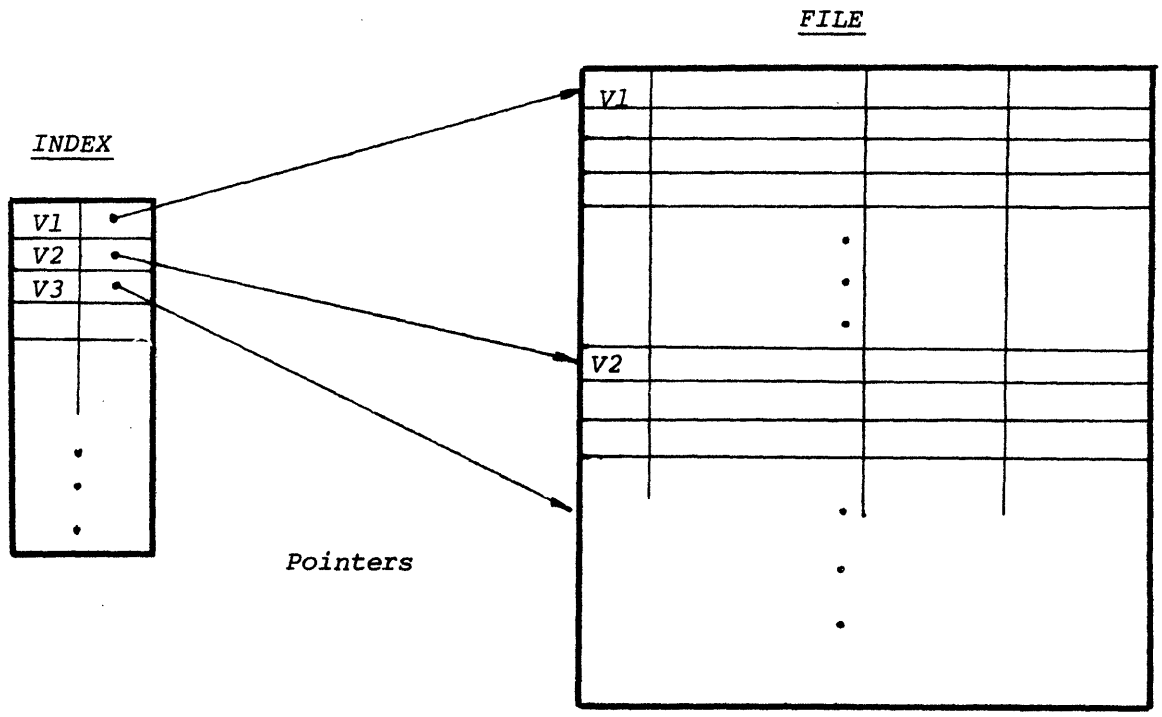


Fig. 13

storage location - oriented ones. Trade - offs among possible implementation techniques must be resolved for each DBMS. The algorithm eventually employed for this function will be "transparent" to any upper level functions; which means that it can be changed without disrupting the functions in upper levels.

(3) While level 2 allows attribute - oriented references to entities, the storage area in which the entities of interest are stored must still be specified (e.g., an index is built on a specific file). Level 3 removes any physical connotation from

entities' references. It allows to "see" entities as the user best conceptualizes them. In other words, this level focuses on "mapping" the "logical" entities' structure into their physical structure. For example, consider the entity "account" of section 2. A user may logically view this entity as depicted in Fig. 14-a (i.e., each account has a set of attributes). In storage, however, accounts' representations need not correspond to that same view. For instance, a subset of their attributes can be kept in a storage area, and the rest in a separate one (Fig. 14-b), in order to improve accessing speed to the most often referenced attributes. Of course, this means that the correspondence between the two areas must be maintained and that references to specific attributes have to be directed to the appropriate area. Level 3 is responsible for this. In the case that a unique logical entity view is adequate for all users and it can be directly represented in storage, the mapping function of this level is greatly simplified; level 3 can be even ommitted in such a case, and the resulting DBMS made simpler (see section 2.3).

(4) With the support of level 3, entities are made to "appear" as the users perceive them, regardless of any particular storage representation. However, the users need to manipulate these logical entities in certain ways. For example, a set union may be required to group the entities obtained in several retrieval operations, or an attribute selection needed to get just the attributes of interest. These operations are algorithmic in nature: a set of "standard"

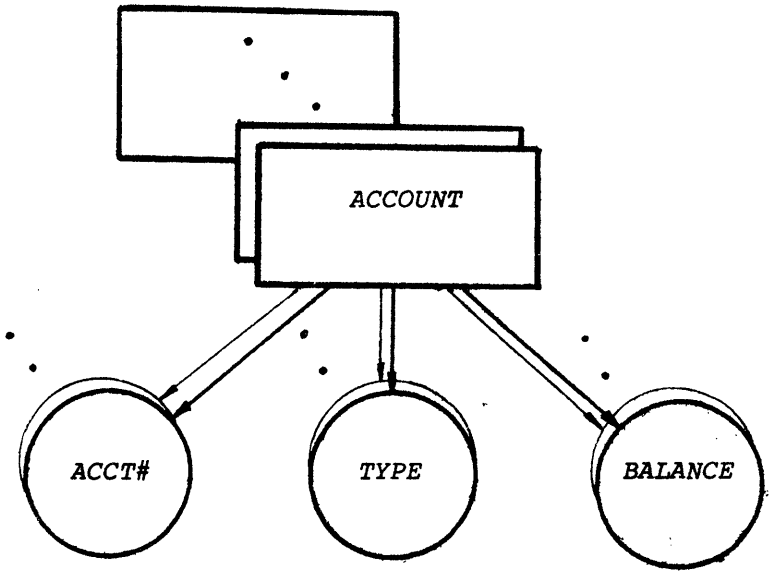


Fig. 14-a

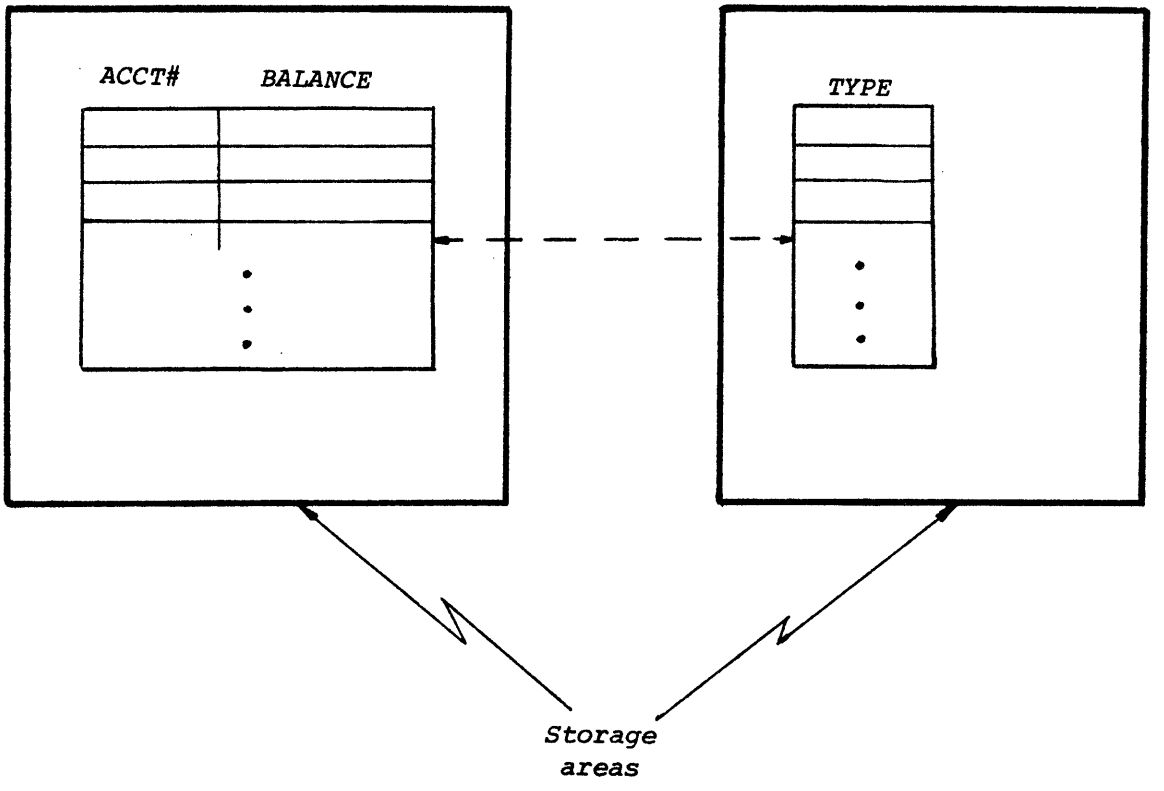


Fig. 14-b

algorithms may be provided to perform them, thus freeing the user from having to explicitly "program" them. Level 4 is made responsible for these algorithms. At the top of level 4, it is thus possible to issue DBMS commands that perform "logical" operations upon logical entities' structures. The particular algorithms employed to implement such functions need not be known at all by the users: In particular, they may be improved, by incorporating a new, more efficient algorithmic scheme, for instance, without changing the way in which users interact with the DBMS.

(5) The top level allows system interactions to be in the form of "property descriptions" (as opposed to sequences of logical operations), expressed as English - like statements such as that illustrated in section 2. Level 5 is responsible for the appropriate translation.

5.3.5.- The coordination of DBMS subsystems.

At this point, it is useful to consider a simple operational example in the context of the framework in Fig. 11. We use the same example discussed in section 5.1, and explain how the DBMS command:

```
SELECT BALANCE FROM ACCOUNTS
       WHERE ACCOUNT_NUMBER =
             SELECT ACCOUNT_NUMBER FROM CLIENTS
             WHERE CLIENT_NAME = 'JOHN DOE'
             AND ACCOUNT_TYPE = 'SAVINGS'
```

can be processed through the hierarchically organized subsystems of Fig. 11. The command is issued to the highest

hierarchy level, that translates it into a semantically equivalent sequence of steps involving logical operators (such as set definition or attribute selection), available at level 4. For example, the following sequence might be used:

1.- Compute set A:

A:(a/a ε CLIENTS, NAME(a)='JOHN DOE',
ACCOUNT_TYPE(a)='SAVINGS')

2.- Compute set B:

B:(b/b = ACCOUNT_NUMBER(a), a ε A)

3.- Compute set C:

C:(c/c ε ACCOUNTS, ACCOUNT_NUMBER(c) ε B), and

4.- Compute set D (the answer set):

D:(d/d = BALANCE(c), c ε C).

Once this sequence has been identified, the lower level subsystems can be employed to actually perform the computations. For example, the following level 3 command may be issued to compute set A:

```
RETRIEVE CLIENTS RECORDS WHERE NAME = 'JOHN DOE'
                         AND ACCOUNT_TYPE = 'SAVINGS'
```

Level 3 is then responsible for mapping the logical set "CLIENTS" into its physical representation. Assume, for the purpose of illustration, that this logical set is stored physically in, say, file number 3 as shown in Fig. 15 (i.e., a sequential file where records' fields correspond to clients' attributes). If this is the case, level 3 will translate the command above into a level 2 command, such as:

```
RETRIEVE FILE 3 RECORDS WITH FIELD1 = 'JOHN DOE'
                        AND FIELD4 = 'SAVINGS' ,
```

FILE #3 (CLIENTS)

(NAME) *FIELD1*	*(ADDRESS)* *FIELD2*	*(ACCT#)* *FIELD3*	*(TYPE)* *FIELD4*
	. . .		
JOHN DOE	*BOSTON, MA.*	*1102*	*Savings*
	. . .		

Fig. 15

and pass such command along to level 2.

By using the available access methods, level 2 would then transform the specification

FIELD1 = 'JOHN DOE' AND FIELD4 = 'SAVINGS'

into a collection of file 3's record numbers whose contents satisfy that condition. These record numbers can then be used to issue level 1 commands to retrieve the corresponding records. The result of this retrieval operation would subsequently be passed back to the top level for further processing.

6.- The design process in the context of the framework.

Once a framework such as that in Fig. 11 has been identified, the remaining design activities can be organized and coordinated in its context, thus bringing more structure into the design process. Fig. 16, discussed below, depicts how this can be done.

Consider again Fig. 11. In a top - down approach to DBMS design, the first problem is to identify a "data model" in which the users' data manipulation problems can be conveniently formulated, and to select a "data manipulation language" associated with it.

By "data model" we mean (see [Date 74]) a well defined logical structure capable of accurately representing the relationships among data items relevant to a specific situation. Several such models have been proposed (for example, the relational [Codd 70], network [Bachman 69], entity-set [Senko et al. 73], entity - relationship [Chen 76] data models). Selecting the appropriate data model is an activity that can hardly be modeled, one of its objectives being to match what has been called "intrinsic information structure" ([Lefkowitz 69]) which is something very difficult to formalize. This activity thus becomes, at least for the time being, a matter of the users' personal preferences (although it can be effected by practical reasons such as model availability). Once a data model has been selected, a set of logical operators (designed to manipulate the data structures

supported by the model) is effectively selected as well, since they are strongly model - dependent. Typically, however, the users don't interact with the DBMS directly through these operators --instead, an English - like query language is provided that allows them to specify their queries in a more "natural" way. Such languages attempt to match the users' cognitive characteristics and sophistication, and thus more than one can be associated to a given data model (for the relational model, for instance, the languages SEQUEL [Chamberlin et al. 74], SQUARE [Boyce et al. 73], QBX [Zloof 75], QUEL [Stonebraker et al. 76] have been proposed). Some effort has been devoted to support the language selection process ([Thomas et al. 75], [Reisner et al. 75]), but it still remains largely a matter of personal preference, too.

This first DBMS design problem, therefore, falls well outside the capabilities of any formal analysis. Its solution, however, determines the next design problem (see Fig. 16).

The second design problem focuses on how to translate English - like query expressions into semantically equivalent sequences of logical operations. This problem is better defined than the preceding one. Although it has not been the subject of extensive mathematical analysis, several heuristics have been proposed for its solution (see [Rothnie 74], [Wong et al. 76]). The main thrust of such heuristics is to avoid combinatorial growth in the resulting sequences of logical operations. Central to our discussion here is that these heuristics tend to be independent of lower level issues, since they are derived

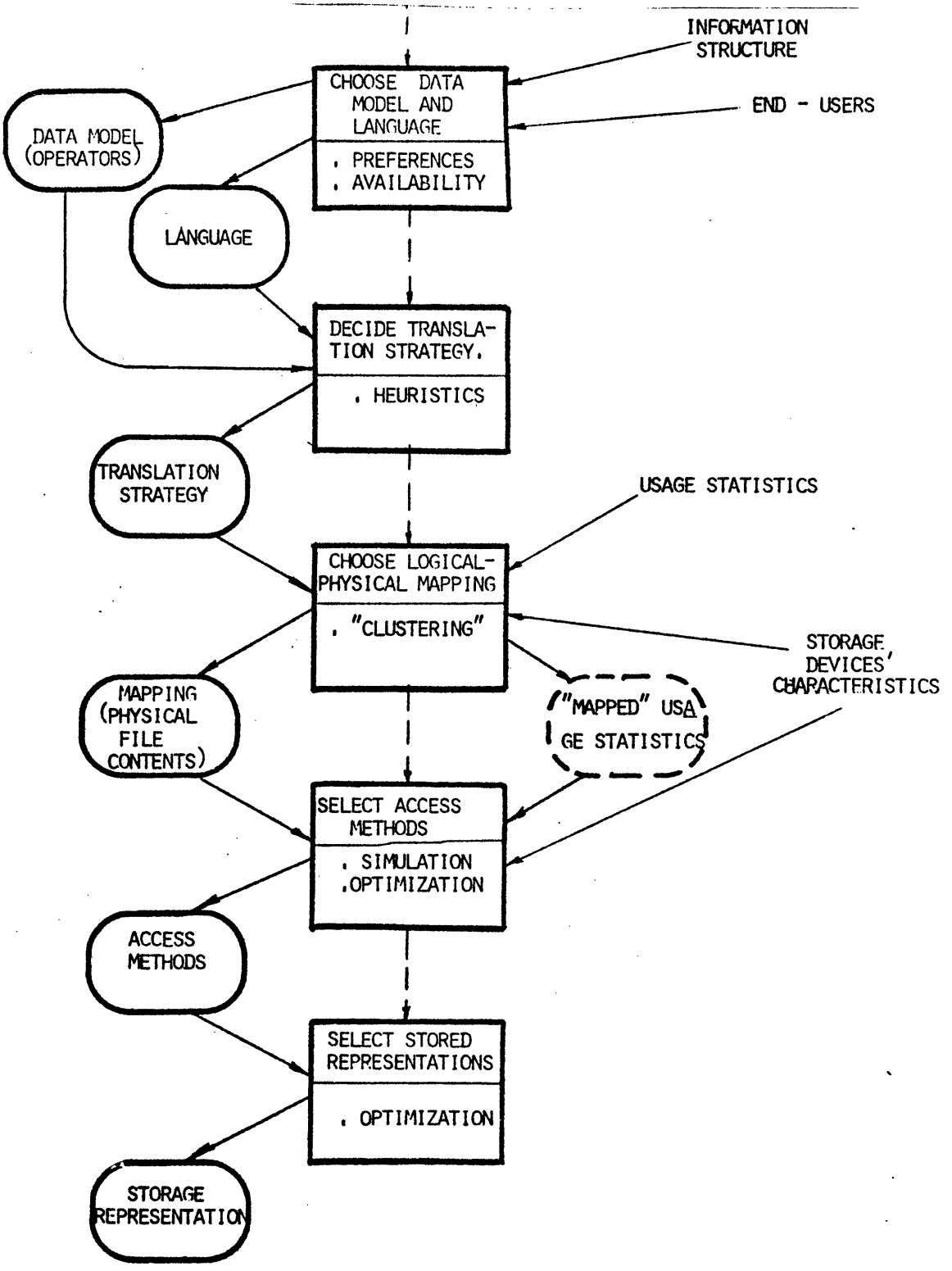

INFORMATION STRUCTURE
CHOOSE DATA MODEL AND LANGUAGE
. PREFERENCES
. AVAILABILITY
END - USERS
DATA MODEL (OPERATORS)
LANGUAGE
DECIDE TRANSLATION STRATEGY.
. HEURISTICS
TRANSLATION STRATEGY
USAGE STATISTICS
CHOOSE LOGICAL-PHYSICAL MAPPING
. "CLUSTERING"
STORAGE DEVICES' CHARACTERISTICS
MAPPING (PHYSICAL FILE CONTENTS)
"MAPPED" USAGE STATISTICS
SELECT ACCESS METHODS
. SIMULATION
.OPTIMIZATION
ACCESS METHODS
SELECT STORED REPRESENTATIONS
. OPTIMIZATION
STORAGE REPRESENTATION

Fig. 16

from intrinsic properties of the logical operations of interest (e.g., whether they are commutative, etc.). By means of these heuristics, a strategy for the translation problem can be identified. Once this is done, the remaining design problems become more structured.

At the next level down in Figs. 11 and 16, the design problem is to decide how logical entities are going to be grouped physically for storage purposes, and to identify the resulting logical / physical mapping. The techniques proposed for solving this problem typically require information regarding data base usage statistics or forecasts (see [Severance et al. 75], [Schmid et al. 75]), as well as some aggregate information about typical storage devices' performance. The former can be generated if the solution to the two preceding problems is known, and forecasts made of the overall data volume and the typical query mix at the user level. The solution to this problem takes the form of a mapping from logical entities onto physiscal files, and specifies the contents of the latter.

The next problem is that of access methods selection. Several approaches proposed to solve it ([Cardenas 75], [Lum 71], [Severance et al. 74]) take as input information the contents of physical files, which is the output of the preceding analysis, plus their usage statistics (that can be obtained from the frequency of queries against logical entities by means of the mapping identified above). The generated result takes the form of recommendations regarding which access method

seems appropriate for the items in each file.

Lastly, the problem of data representation in stored form must be solved. This is basically a trade - off problem between required storage space and encoding/decoding time. For example, a data item may be not stored at all, if it can be computed from other stored data items (see [Folinus et al. 74]), but this means that time will be spent in computing it when needed. Most aspects of this problem may be formulated as a mathematical programming problem, with constraints generated by the solution to the preceding problem (for instance, if an index is to be maintained over a given data item, it must be stored explicitly).

6.1.- An example.

For concreteness, a specific example of the coordination of design activities as described above is presented in this section. The discussion focuses on the last three (from top to bottom) design problems depicted in Fig. 16:

(1) One of the techniques proposed to solve the problem of choosing a logical / physical mapping is described in [Severance et al. 76]. The problem can be described as follows:

Given a logical group of data items (e.g., the attributes of certain entities, as seen by the users):

$$D: \{ d(1),...,d(i),...,d(m)\} \quad ,$$

and a set of retrieval operations (users):

$$U: \{ u(1),...,u(j),...,u(n)\}$$

that manipulate the data items in D, decompose D into two subsets, say D1 and D2, to be assigned to different storage areas, so that the need for "expensive" storage space is minimized and "user convenience" is "maximized".

The rationale behind this formulation is as follows: Storage space is seen as divided into two areas or segments, one of which allows quicker references to data items than the other, but is also more expensive (for example, the two segements could reside in separate storage devices with different speeds and costs, or they could reside in the same one but have different associated access methods, etc.; at this level it is not specified how the two areas actually differ, it is only known that one of them provides better "service").

Storing all the data items in the superior segment is very efficient but expensive. Doing the opposite is cheap but may result in unacceptable performance for critical queries. To formalize this trade - off, the following is assumed:

- Associated with each $d(i) \in D$, there is a measure, $w(i) > 0$, of the storage space needed to represent it (since no decision has yet been made as to how, specifically, each data item is to be encoded, w(i) can be a reasonable upper bound).

- Associated with each $u(j) \in U$, there is a set, $S(j) \subseteq D$ of data items manipulated by user u(j), and a measure, $v(j) > 0$ of the "importance" of user u(j) relative to the rest of the users in U.

The meaning of w(i) and S(j) is clear. As for v(j), a surrogate may be the relative frequency in which u(j)'s operations occur; this can be derived from the frequency distribution of forecasted users' queries.

The problem can then be stated as:

$$\text{Min} \quad k \sum_{i:d(i) \in D1} w(i) - \sum_{j:u(j) \in U-U1} v(j)$$

s.t.:

$D1 \subseteq D$, where:

- $U1 = \{ u(j)/u(j) \in U \text{ and } S(j) \subset D1 \}$,
- k is a conversion factor.

The specific methodology employed to solve this problem is of no particular concern to us here. We will only say that an alternative objective function (in terms of query frequencies and processing costs) that avoids the need for the conversion

factor k has been also proposed and used in [Severance et al. 76], thus making the input data more realistic. What is important to notice is that the formulation above makes no detailed assumptions about lower level parameters (in the context of Fig. 16). In particular:

- No specific access speeds for the two storage segments are considered.
- No assumption is made as to how many instances of each d(i) in D are going to be present in the data base.
- User convenience is taken into account explicitly.

When this problem is solved, it is known that the data items in D1 must be stored in a segment whose accessing speed is greater than that for the segment containing the items in D2. Some lower bounds for these speeds can be assumed. Also, the procedure just described can be applied to different D sets, thus obtaining a collection of data items' subsets to go to quick storage areas and another to slower areas, maybe with different speed lower bounds.

(2) The problem in (1) above decides what data items are to be stored together, basically. Now the issue becomes how to provide the access speed needed for each collection of data items. Approaches to attack this problem have been proposed by [Rothnie 72] and many others. Since there are many access methods available from which we can choose, it is difficult to analyze them all at once; their implementations differ so drastically that parameters describing one method are completely irrelevant for others. This is another reason for

decomposing the design problem in a form as that of Fig. 16: For each design, the most appropriate access methods' analysis can be selected, at the corresponding level. Assume we choose Rothnie's approach. He proposes a methodology for choosing between the access methods called "Multiple Key Hashing" (MKH) and "Inversion".

Let:

- D: {d(1),...,d(i),...,d(m)} be the set of data items that we decided to store together (i.e., a generic "record" in a "file", in the traditional sense),
- N be the number of D instances to be stored (i.e., the number of records in the file),
- NV(i) (i=1,...,m) be the number of different values taken by d(i), NV(i) < N ∀ i,
- P(i) (i=1,...,m) be the probability that d(i) ε D will be used as "key" (i.e., the frequency in which d(i) is involved in an "attribute specification" statement of the form d(i) = value),
- A(i) (i=1,...,m) be a set of binary decision variables, =1 if d(i) is accessed via MKH, 0 if via inversion,
- H(i) be the hashing function applied to d(i) if MKH is used for it, and
- NH(i) (i=1,...,m) be the range of function H(i), if it exists.

The problem can then be stated as a minimization problem, for the expected number of I/O operations (between secondary storage and core memory), as follows:

$$\text{Min} \sum_{i=1}^{m} A(i)*P(i)\{ \prod_{\substack{j=1 \\ j\neq i}}^{m} NH(j) + A + \sum_{i=1}^{m} (1-A(i))*P(i)*(B + N/NV(i))$$

s.t.:

$$NH(i),\ A(i) \geq 0\ ,\ i = 1, \ldots, m$$

$$A(i) \leq 1\ ,\ i = 1, \ldots, m\ ,\ \text{where}$$

A and B are known overhead constants for the two access methods.

Again, the particular procedure employed to solve this problem is not central to our discussion here. What is important is to realize that the information needed in order to formulate the problem can be obtained from higher levels:

- D is part of the solution for the problem in (1),
- The distribution of d(i)'s values can be obtained from the users,
- The probabilities P(i) are similar to the values v(j) in the previous problem, but somewhat more concrete: they specify not only what d(i)'s are used, but also how are so (i.e., as keys).

When a solution for this problem is identified, D has been decomposed into three subsets:

$D = D1 \cup D2 \cup D3,\ \ Di \cap Dj = \phi\ ,i,j=1,2,3,\ i \neq j,$

such that:

- The items in D1 are to be accessed via MKH,
- The items in D2 are to be accessed via inversion, and
- The items in D3 are never accessed directly (i.e., the

corresponding P(i)'s are zero).

(3) The next problem focuses on deciding upon stored representations for the items in sets D. This can be formulated as a linear programming problem to choose among available encoding techniques.

For example, let:

- D: {d(1),...,d(i),...,d(m)} be the set of data items for which encodings have to be identified,

- $X(j) \subset D$, $j=1,\ldots,n$ ($n < m$) be subsets of data items, of dimensions DIM(j) ($\sum_{j=1}^{n} DIM(j) = m$), in each of which K(j) elements (K(j) < DIM(j)) can be algorithmically derived from the remaining DIM(j) - K(j) (i.e., K(j) elements in each subset can be virtually maintained),

- A(i), B(i) be the costs of maintainig data item d(i) in stored form or virtual form, respectively,

- E(i) = 1 if an access method is to be implemented over item d(i), =0 otherwise (i.e., the solution of the problem in (2) above), and

- H(i) be a set of binary decision variables, set to 1 if data item d(i) is explicitly stored, to 0 if it is virtually kept.

Then the problem becomes:

$$\text{Min} \sum_{i=1}^{m} H(i)*A(i) + \sum_{i=1}^{m} (1-H(i))*B(i)$$

s.t.:

$$H(i) \geq E(i),\ i = 1, \ldots, m$$

$$\sum_{d:d(i)\in X(j)} H(i) \geq DIM(j) - K(j),\ j = 1, \ldots, n$$

$1 \geq H(i) \geq 0$, $H(i)$ integers, $i = 1, \ldots, m$.

As in the two problems above, we see that this problem requires information that has been identified in higher levels.

This example illustrates the kind of coordination of analysis techniques that we had in mind in the discussion of section 6.

At this point, it is apparent that a single top - down pass for the design process may not always suffice. For example, we assumed bounds in the variables employed in high level design problems; exact values for these variables are determined in lower level analyses. The possibility of being able to improve a first pass design significantly by reconsidering some of the high level problems once better bounds for the relevant variables are available should be investigated (i.e., the identification of possible feedback loops in the design process schematized in Fig. 16).

CONCLUSIONS

The need for a new, more structured approach to complex software systems design is a subject of increasing concern due to several problems in the software development process. We believe that a technology independent system framework is needed to organize and coordinate the design activities in such a way that: (a) designs biased by existing technology are avoided, (b) design subproblems that have been analyzed and solved can be meaningfully coordinated, (c) the impact of shifts in operational needs and/or the appearance of new technology is reduced to affecting only well defined subsystems,(d) compatible systems are naturally obtained, and (e) modelling activities for performance evaluation purposes are simplified.

A methodology aimed at the synthesis of such a framework was proposed and its potential investigated in the context of a representative instance of complex software systems: DBMSs.

A number of research activities must be undertaken in order to make that methodology operational. They constitute the core of our present research activities.

APPENDIX A

SET OF DBMS REQUIREMENTS

1.- The collection of data items that is to be supported is perceived as logically organized in more that one way.

2.- In these logical organizations, data items are seen as forming logical groups, of special meaning to the user(s).

3.- Relationships exist among data items, meaningful to the user(s).

4.- Some of the relationships among data items are algorithmic in nature.

5.- There is a collection of logical operations involving groups and relationships that must be supported, since they define the types of data manipulations required.

6.- Data items are to be organized physically in a unique way.

7.- There are a number of specific queries to be supported.

8.- Query frequency is not uniform (there are "critical" queries).

In a query, references to data items are by:

9.- Logical group membership, and

10.- Value (i.e., their value is specified).

11.- The expected time spent in locating the data items appearing in a given query should be minimized.

12.- The distribution of data items accross queries (i.e., data items appearing in a query) is far from uniform, in general.

13.- Queries are to be expressed in an English - like language.

14.- The query language should be unambiguous.

15.- Query expressions should be non - procedural (descriptive).

16.- Different types of data items must be supported (e.g., integers, character strings).

17.- Data items of the same type can be combined by means of well defined operations (e.g., addition for integers; concatenation for character strings).

18.- Alternative data types may be employed, if necessary, for certain data items.

19.- Each data item takes values in a specific range of its data type.

20.- Data items do not necessarily take all values in their value range.

21.- Data redundancy should be avoided.

22.- Storage cost should be minimized.

APPENDIX B

ASSESSMENT OF INTERDEPENDENCIES BETWEEN PAIRS OF REQUIREMENTS

*Requirement 1 is related to:
- 2.- Logical views defined in terms of logical groups.
- 3.- Logical views defined in terms of logical relationships.
- 5.- Logical operations possible in context of logical view.
- 6.- Different logical views derivable from unique physical organization.
-21.- A data item in more than a logical view could be redundantly represented.

*Requirement 2 is related to:
- 1.- See requirement 1.
- 3.- Relationships among groups possible.
- 5.- Logical operations possible with groups.

*Requirement 3 is related to:
- 1.- See requirement 1.
- 2.- See requirement 2.
- 5.- Logical operations possible with relationships.

*Requirement 4 is related to:
-17.- Algorithmic relationships consistent with operations defined on associated data types.
-18.- As above.
-21.- Algorithmic relationships can help to avoid redundancy.
-22.- Algorithmic relationships can help to reduce storage cost.

*Requirement 5 is related to:
- 1.- See requirement 1.
- 2.- See requirement 2.
- 3.- See requirement 3.
- 7.- Queries to be supported must be computable through logical operations.
-15.- Non procedural expressions should correspond to (at least one) combination(s) of logical operations.

*Requirement 6 is related to:
- 1.- See requirement 1.
- 9.- Logical group membership should unambiguously correspond to membership in some part of the unique physical organization.
-21.- A unique physical organization favors non - redundancy.

*Requirement 7 is related to:
- 5.- See requirement 5.
-13.- All queries should be expressable.
-14.- Expressions for all queries should be unambiguous.
-15.- No procedural expression should be allowed in the statement of any query.

*Requirement 8 is related to:
-11.- Frequency must be taken into account for expected response time.
-12.- Frequency of queries and frequency of data items in queries determine frequency of data items' references: possibility of getting very bad response time for some unfrequent queries should be avoided.

*Requirement 9 is related to:
- 6.- See requirement 6.
-11.- Efficiency of mechanism for locating a data item given its membership in a specific group.
-12.- Take into account the overall importance of each group as indicated by the frequency in which their data items appear in queries, to decide upon the mechanism above.
-21.- Representing every logical group physically is an alternative that goes against avoiding redundancy.

*Requirement 10 is related to:
-11.- Efficiency of mechanism for locating a data item given its value.
-12.- Take into account overall importance of each data item (in terms of queries where it appears) when choosing the mechanism above.
-19.- If it is known that a specific value is not taken by a given data item, a reference specifying that value can be easily resolved.
-20.- Similar to above.

*Requirement 11 is related to:
- 8.- See requirement 8.
- 9.- See requirement 9.
-10.- See requirement 10.
-19.- See relationship between requirements 10 and 19.
-20.- See relationship between requirements 10 and 20.
-22.- General trade - off response time - storage space.

*Requirement 12 is related to:
- 8.- See requirement 8.
- 9.- See requirement 9.
-10.- See requirement 10.

*Requirement 13 is related to:
- 7.- See requirement 7.
-14.- English - like goes against unambiguousity.

*Requirement 14 is related to:
- 7.- See requirement 7.
-13.- See requirement 13.
-15.- Non procedurality can bias the language towards being ambiguous.

*Requirement 15 is related to:

- 5.- See requirement 5.
- 7.- See requirement 7.
-14.- See requirement 14.

*Requirement 16 is related to:
-17.- Operations consistent with data item using corresponding data type.
-18.- Similar to above.
-22.- Certain data types may be stored more efficiently than others.

*Requirement 17 is related to:
- 4.- See requirement 4.
-16.- See requirement 16.
-18.- If a data type is changed, how about operations associated with corresponding data item?

*Requirement 18 is related to:
- 4.- See requirement 4.
-16.- See requirement 16.
-17.- See requirement 17.
-22.- An alternative data type may be convenient to reduce storage.

*Requirement 19 is related to:
-10.- See requirement 10.
-11.- See requirement 11.

*Requirement 20 is related to:
-10.- See requirement 10.
-11.- See requirement 11.

*Requirement 21 is related to:
- 1.- See requirement 1.
- 4.- See requirement 4.
- 6.- See requirement 6.
- 9.- See requirement 9.
-22.- Avoiding redundancy agrees with reducing storage.

*Requirement 22 is related to:
- 4.- See requirement 4.
-11.- See requirement 11.
-16.- See requirement 16.
-18.- See requirement 18.
-21.- See requirement 21.

REFERENCES

Key to abbreviations employed:

ACM - Association for Computing Machinery.
AFIPS - American Federation of Information Processing Societies.
CACM - Communications of the ACM.
CVLDB - Conference on Very Large Data Bases.
IEEE - Institute of Electrical and Electronic Engineers.
IFIP - International Federation of Information Processors.
SJCC - Spring Joint Computer Conference.
TODS - Transactions on Data Base Systems.

[Alexander 64]:
Alexander, C.: "Notes on the synthesis of form", Harvard U. Press, 1964.

[Andreu 76-a]:
Andreu R.: "The XRM interface as used by SEQUEL", Internal report,M.I.T. Sloan School, September 1976.

[Andreu 76-b]:
Andreu, R.: "The implementation of XRM on top of RM", internal report, M.I.T. Sloan School, September, 1976.

[Astrahan et al. 76]:
Astrahan, M.M. and others: "System R: Relational approach to Database Management", TODS, vol 1 no. 2, January, 1976.

[Bachman 69]:
Bachman,C.: "Data Structure Diagrams", Procs., File 68 Conference, IFIP occasional publication no. 3, Administrative Data Processing Group (IAG), Swets and Zeitlinger N.V., 1969.

[Blum 66]:
Blum, J.: "Modeling, Simulation and Information System Design", Information Science and Technology, Nov. 1966.

[Boyce et al. 73]:
Boyce,R.F. and others: "Specifying queries as relational expressions: SQUARE", IBM Tech. Report RJ 1291, IBM Res. Lab., San Jose, Ca, October 1973.

[Brooks 75]:
Brooks, F. P.: "The Mythical Man - Month: Essays on Software Engineering", Addison - Wesley, Reading, Mass., 1975.

[Cardenas 75]:
Cardenas, A.: "Analysis and performance of inverted data base structures", CACM, 18 - 5, May 1975.

[Chamberlin et al. 74]:
Chamberlin, D.D. and others: "SEQUEL: A Structured English Query Language", Procs. from the ACM workshop on data description, access and control, May 1974.

[Chen 76]:
Chen, P.: "The entity - relationship model -- Towards a unified view of data", TODS, 1 - 1, March 1976.

[Codd 70]:
Codd, E.F.: "A relational model of data for large shared data banks", CACM, 13 - 6, June 1970.

[Date 74]:
Date, J.: "An introduction to Data Base Systems", Addison Wesley, 1974.

[Dijkstra 68]:
Dijkstra, E.W.: "The structure of T.H.E. multiprogramming system", CACM, 11 - 5, May, 1968.

[Donovan 75]:
Donovan, J.J.: "An application of a Generalized Management Information System to Energy Policy and Decision Making", Procs., AFIPS, 1975.

[Folinus et al. 74]:
Folinus, J.J. and others: "Virtual information in data base systems", M.I.T. Sloan School working paper 723 - 74, 1974.

[Gabbay 75]:
Gabbay,H.: "A hierarchical approach to production planning", M.I.T. O.R. Center TR No. 120, December 1975.

[Hartigan 75]:
Hartigan, J.: "Clustering algorithms", Wiley Interscience, 1975.

[Hax 75]:
Hax,A.C.: "The design of large scale logistics systems: A survey and an approach", in "Logistics Research Conference", W. Marlow, ed.,M.I.T. Press, 1975.

[IDMS]:
"Integrated Data Management System", Cullinane Corp., Boston, Ma.

[IBM 73]:
"Guide for the users of RM - PL/I version", June, 1973.

[IBM-a]:
"Information Management System/360 Version 2. General Information Manual", IBM form No. GH20-0765.

[Joyce et al. 74]:
Joyce, J.D. and others: "Data Management System Requirements", in "Data Base Management Systems", D.A. Jardine, Editor, North Holland, 1974.

[Lefkowitz 69]:
Lefkowitz, D.: "Data management for on - line systems", Hayden Book Co., 1969.

[Lorie 74]:
Lorie,R.A.: "XRM - An extended (n - ary) relational memory", IBM Cambridge Scientific Center Technical Report No. 320-2096, January 1974.

[Lum 71]:
Lum, V.Y. and others: "Key-to-address Transform Techniques: A Fundamental Performance Study on Large Existing Formatted Files", CACM, 14 - 4, April, 1971.

[Madnick and Alsop 69]:
Madnick, S.E. and Alsop, J.W.: "A modular approach to file systems design", Procs. of the Spring Joint Computer Conference, 1969.

[Madnick and Donovan 74]:
Madnick, S.E. and Donovan, J.J.: "Operating Systems", McGraw Hill, 1974.

[Madnick 76-a]:
Madnick,S.E.: "Concepts and Facilities", Report FOS-22, Research Report prepared for the U.S. Navy Electronics Laboratory Center, San Diego, Ca.

[Madnick 76-b]:
Madnick,S.E.: "Proposal for Research on the Design of a Family of Database Systems", preliminary draft, July 1976.

[Mesarovic et al. 70]:
Mesarovic,M.D. and others: "Theory of multilevel, hierarchical systems", Academic Press, N.Y., 1970.

[Myers 75]:
Myers, G.J.: "Reliable software through composite design", Petrocelli / Charter, N.Y., 1975.

[Nolan 73]:
Nolan, R.L.: "Computer Data Bases - The Future is Now", Harvard Business Review, 51 - 5, September - October 1973.

[Nunamaker 71]:
Nunamaker, J.F.: "A Methodology for the Design and Optimization of Information Processing Systems", AFIPS Conference Procs., SJCC, 1971.

[Palmer 75]:
Palmer. I: "Database Systems: A practical reference", Q.E.D. Information Sciences, Wellesley, Ma, 1975.

[Parnas 74]:
Parnas, D.L.: "On a buzzword: Hierarchical structure", Information Processing 74 - IFIP vol. 2, North Holland, 1974.

[Parnas 76]:
Parnas,D.L.: "On the design and development of Program Families", IEEE Transactions on Software Engineering, SE-2-1, March 1976.

[Pattee 73]:
Pattee, H.H.: "Hierarchy theory: The challenge of complex systems", George Brazillier, N.Y., 1973.

[Patterson 71]:
Patterson, A.C.: "Requirements for a generalized data base management system", Procs, FJCC, AFIPS, 1971.

[Reisner et al. 75]:
Reisner, P. and others: "Human factors evaluation of two data base query languages: SQUARE and SEQUEL", Procs., AFIPS, 1975.

[Rhodes 72]:
Rhodes,J.: "Beyond Programming: Practical Steps Towards the Automation of DP System Creation", in "System Analysis Techniques", J.D. Couger and R.W. Knapp, Eds., John Wiley, 1974.

[Rothnie 72]:
Rothnie, J.B.: "The design of generalized Data Management Systems", Ph. D. dissertation, Dept. of Civil Engineering, M.I.T., September, 1972.

[Rothnie 75]:
Rothnie, J.B.: "Evaluating inter - entry retrieval expressions in a relational data base management system", Procs., AFIPS, 1975.

[Schmid et al. 75]:
Schmid, .A. and others: "A multi - level architecture for relational data base systems", Procs., CVLDB, Framingham, Ma, September 1975.

[Senko 75]:
Senko,M.E.: "Information systems: records, relations, sets, entities and things", Information Systems, 1 - 1, Pergamon Press, 1975.

[Senko et al. 73]:

Senko,M.E. and others: "Data structures and accessing in data base systems": I, II, III", IBM Systems Journal, 12 - 1, 1973.

[Sekino 72]:
Sekino,A.: "Performance evaluation of multiprogrammed time shared computer systems", Project M.A.C. report TR - 103, M.I.T., September, 1972.

[Severance et al. 74]:
Severance, D.G. and others: " A practitioner's guide to addressing algorithms: a collection of reference tables and rules of thumb", T.R. No. 240, O.R. Dept., Cornell U., November 1974.

[Severance 75]:
Severance, D.G. and others: "The use of cluster analysis in physical data base design", Procs, CVLDB, Framingham, Ma, September, 1975.

[Severance et al. 76]:
Severance, D.G. and others: "Mathematical techniques for efficient record segmentation in large shared databases", Journal of the ACM, 23 - 4, October 1976.

[Simon 67]:
Simon, H.A: "The architecture of complexity", in "Sciences of the artificial, the M.I.T. Press, Cambridge, Mass, 1967.

[Stonebraker 74]:
Stonebraker, M.: "A functional view of data independence", Procs., ACM workshop on data description, acess and control, May 1974.

[Stonebraker et al. 76]:
Stonebraker, M. and others: "The design and implementation of INGRES", TODS, 1 - 3, September, 1976.

[Teichroew 70]:
Teichroew, D.: "Problem Statement Languages in MIS", Procs., Internaticnal Symposioum of BIFOA, July 1970.

[Thomas et al. 75]:
Thomas, J.C. and others: "A psychological study of query by example", Procs., AFIPS, 1975.

[Wong et al. 76]:
Wong, E. and others: "Decomposition - An strategy for query processing", TODS, 1 - 3, September 1976.

[Young and Kent 74]:
Young, J.W. and Kent, H.K.: "Abstract Formulation of Data Processing Problems", in "System Analysis Techniques", J.D. Couger andR.W. Knapp, Eds., John Wiley, 1974.

[Zloof 75]:
Zloof, M.M.: "Query by example", Procs., AFIPS, 1975.