+++
+ + +
+ Burroughs + "PRELIMINARY" +
+ Application + Technical +
+ Development + Information +
+ Aids + Paper +
+ + +
+++

Date: November 24, 1983

Author: Fred Trout
TSC-West

Product: DMSII - ALL SYSTEMS
Subject: FOUNDATIONS OF A DATABASE MANAGEMENT SYSTEM

This paper is organized into two general topics. The first, DBMS PREPARATIONS, is a discussion on why an organization should implement a Data Base Management System and the attitudes required for a sucessful beginning. The second topic, CHOOSING A BASIC DESIGN PHILOSOPHY, deals with the design of data and structure relationships. It gives some of the advantages and disavantages of several design models and suggests criteria for a selection.

A small bibliography is included, not for reasons of proof or reference, but as suggested cover-to-cover reading as a part of the DBMS learning curve.

Printed in U.S. of America TIP # 1114238-020

BLANK PAGE FOR FORMATTING

DBMS PREPARATIONS

Jumping on the Data Base Management Systems bandwagon should be a considered leap. The attitudes and commitments that motivate this decision should be carefully prepared. Technological and product learning curves need to be initiated. A new administration will apply new problem-solving techniques and new software products on the user interface. In fact, there is a long list of preparations required simply to provide the foundation that will eventually fulfill the promises implied by a Data Base Management System.

WHOSE IDEA WAS THIS, ANYWAY ?

Whether to implement a Database Management System is a critical decision facing most users in the 1980s. There are increasing pressures from the industry prophets who foretell that a database environment will be a requirement to do business in the next decade. There are pressures from mainframe vendors as they progress through the advancing technologies replacing older products with new features and new products. Application software products are also using advanced environmental and generative software to implement database concepts.

These motivations, however real, are often not enough to insure the commitment of your user community. The attitude of the users that is required for cooperative development is their perception of a compelling need to improve the control and productivity of the company's data and personnel resources. Without this internally generated need, there will be an additional requirement to sell the database concepts and the attendant changes to a status quo user community. There will be changes, and users will view an unrequested change with the same enthusiasm as a wage freeze. Without considerable promotion, the database may be viewed as doing something to, rather than for, the users.

Oversell is often mistakenly used to gain the user's cooperation. Promises of unlimited inquiry, immediate response, and fantastic new features result in disappointed users and complex, unwieldy systems. An understanding that a proper foundation must be built over one or two years and evolved toward those exotic features within the envelope of the physical resources is more realistic. The systems concepts and software technology is available to conceive these systems, but the hardware is not yet capable of handling high volumes in that environment.

CAN WE CHEW WHAT WE HAVE BITTEN ?

The development of a database and the use of a Database Management System is often included with a concept of an on-line, real-time, transaction-oriented system. Transaction processing is the more likely goal, i.e. that concept where a small unit of change affects a only small portion of the database, and that change is immune from the kinds of failures that plague current systems. Transaction processing systems will effect everything from user views and screen formats, datacomm systems, message control systems, to the transaction processors and database design. But more importantly, it will change the way you perceive and solve data processing problems, and may have an effect upon a user environment equal to the introduction of Data Communications, or EDP itself.

Any data management system that is available today is expensive in the consumption of hardware resources. It also requires a shift of human resources from production and maintenance technologies toward design and analysis technologies. End-users may be required to review their requirements and usage of data. Operational procedures become more rigorous and disciplined. A fully developed system is likely to encompass all of the critical information flow and automated decision-making for a whole company. The final effect of such a powerful software system is likely to reach far beyond the current scope of data processing departments. Although the DP department is likely to assume technical leadership, a decision of this magnitude should become a company commitment rather than a departmental alternative.

DIRECTION FORWARD - WHICH RIGHT FOOT ?

Recognizing that Data Management is still low on the technology curve is an important input to the decision of how and how much of the company's resources are to be committed. The industry has, and will continue to be inundated by publications on the philosophies and technologies of Data Management. The logical conclusion derived from this flood of information is that the technology, hardware and software for DMS and for data comm systems is in a state of rapid change. Therefore, a systems design strategy that uses small, simple, flexible, and expandable components is best suited to assimilate future technologies as they evolve.

Another important function at this level of decision making is to define short and long range goals. Short range goals must be consistent with the company's resources and the capability of the selected software tools. Long range goals are required to set the direction and provide the resources for the necessary learning curves of these new technologies and systems. Some of that learning curve may include the management, for these decisions require considered and learned judgement at the highest levels.

Once a reasonable set of goals and time-frames have been established, the theoretical systems development would follow this scenario: data analysis and structuring, database design and modeling, systems design, implementation, simulation, optimization, and production. This scenario works very well for new applications, but for entrenched applications one must consider the emotional inertia of users and programmers and the management's concept of sunk cost.

If a project's starting point includes a considerable investment in application programs, then there is a strong tendency to think in terms of converting to a database. However, "converting to a database" may be akin to turning lead into gold and require the services of an alchemist rather than a DBA. Having a "considerable investment in application programs" is another term for being stuck with a bunch of old programs with old designs and data relationships. Certainly there is the capability to "convert" to a Data Base Management System, but recognize that here the major change is the the replacement of the filing system. The old implementation request, via MCP, is to perform a specific I/O with a narrow interface. The latter request, via DMSII, is much broader, generalized interface with higher levels of capabilities and guarantees. The resultant comparison of cost/performance between the two systems have provided unwarranted disappointment of this unfortunately popular technique of "converting" to a database. These conversions tend to incur most of the cost of a DBMS and little of the benefits.

There are other forms of conversion that seem innocuous at first glance. They include applications using FORTE, FORTE2, DMSI, or even other vendors DMS products. It could even include older implementations of DMSII that are being converted from Burroughs Small or Large Systems. Their major flaw is the probable lack of data analysis and structuring and the outdated database designs used with older systems.

Just a few years ago the state of the art in database design and DMS systems was the capability to physically implement multiple levels of hierarchical and network relationships. This implementation greatly reduced or eliminated most forms of redundant data which was important in the days of limited and expensive disk storage. It involved the use of embedded structures, i.e. a master record would own other records or a list of records. The industry was quite proud of this technological advance and embraced the concept enthusiastically.

The experience gained with those earlier databases showed embedded structure relationships to be extremely inflexible and cumbersome for the database operations relative to reorganization and recovery. In today's systems, the response to change and failure is critical. Embedded structures also have severe limitation in access and maintainence functions.

Todays designs are almost exclusively flat (not embedded) with limited data redundancy for the purpose of developing symbolic relationships between data structures. Reorganization and recovery of flat structures affect only the specified disjoint structures, which is likely to be a smaller subset of the database. Embedded structures, if used at all, would be limited to special cases of specific optimization usage. One to one conversion of hierarchical and network structure designs will retain these logical and operational anomalies. They should take the path of punch cards and teletypes.

WHO IS IN CHARGE OF THIS CHARGE ?

Due to its generative nature, much of the effectiveness of the DBMS lies within the control of the individual site. It is the site's definition of its data relationships and structures; the site's definition of physical file attributes, DMSII options and features; the site's application program's specific usage requests, and the site's actual volumes and populations that make each environment unique. It is the design and definition of this environment and the physical resources available that have the greatest effect on performance.

The Burroughs environmental software tool, DMSII, is used to develop a user tool, a database. The database is a component of a processing system. Using a building as an analogy, DMSII would be the building material, the two-by-fours of your house. The design and function of the building is up to the architect. The features and fixtures are determined by the end-user. The characteristics of the building material is developed by the vendor, and as the materials will affect to some degree the functional design of a building, so will the features of a DBMS affect the processing system design.

The definition, design, and usage of the DBMS and, most likely, the higher levels of the other parts of the system should be coordinated and controlled by a single administration. This control is established in the function of the Data Base Administrator. It is the DBA who must acquire expertise in the specific applications and usage of data, in database design technologies, and in the DMSII features and capabilities. The DBA must then develop specific knowledge of how these interrelated criteria apply to the requirements and resources of the organization.

ONCE AGAIN, WHY ARE WE DOING THIS ?

Once the control of data resources is accomplished, the users can expect data that is common, current, secure, and reliable. From this position, the increase in the productivity can be realized. The data can take on additional responsibilities in algorithmic business decisions. Information, heretofore unavailable, is easily derived and presented. Programming responses to changing environments become flexible and timely. The bottomline rewards of responsive programming and a flexible, reliable database which contains coordinated, current information have value; a value that has been determined to outweigh the cost of current, or more likely, future alternatives. What is the cost of missed opportunity? What value can be placed upon a new function not implemented or lost chance to improve productivity because of the time, or cost, or capability? The justification, to some, may be simply to provide increased control and productivity of their programming and data resources. To others, the DBMS becomes the lifeblood of the company's competitive existence.

Once the decision to have a database management system is made, there is a sequence of functions that should be performed before implementation. They include developing a broad perspective and an informed attitude, becoming trained in DMS technologies, analyzing the company's data resources and usage, providing a comprehensive strategy, and finally, developing and modeling the systems and database design. Only after implementation are the simulation, analysis, and optimization functions performed.

Performance of these functions allows the development of a Database Management System which answers the challenge of the '80s and provides the foundation for the technology of the '90s.

CHOOSING A BASIC DESIGN PHILOSOPHY

DMSII is often described as a hierarchical data management system. This is far from the whole truth. DMSII provides the database designer with the tools required to follow several design philosophies: hierarchical, network, flat/relational, or any mixture thereof.

The following sections discuss each of the approaches to database design. But first, the concept of basic data storing and accessing should be understood, and that requires some understanding of the DMSII structures.

DATA AND ACCESSING STRUCTURES

Datasets

The basic database structure is the dataset. The physical form of a DMSII dataset is a disk or pack file with all the typical file attributes: blocksize, arealength, areas, familyname, etc. This DMSII file contains records which are described as a collection of related data items. In one popular abstraction, the dataset describes an entity, the data items are attributes of that entity, and records are instances or occurrences of that entity. In the CODASYL terminology, the dataset would be a schema.

All datasets can be accessed serially (or ordinally) via read first/next type of functions. Records can also added, deleted, locked, etc., via the same concept of physical, ordinal, direct access to the dataset. There are, in fact, many applications where the "old-fashoned", serial access is logically sound and physically the most efficient method, even in the most sophisticated implementations.

Keys

One of the major characteristics of a database system is increased power to access data. Underlying this is the facility to identify data, and since it is organized into records, to identify the record. Each database record should be uniquely identifiable. Modern database designs require that this uniqueness be a function of the data items. That is, the value of a data item or items in a record be unique for a dataset. The item(s) thus identified is called the key in this design philosophy.

The definition of the term, key, has an original meaning of record identification, however, its usage has given it connotations more concerned with record accessing. Over several generations of data design, its meaning has been expanded, narrowed, specified, generalized,

and even syntaxed to the point that it requires adjective or contextual definition for each usage. This is mentioned as a warning not to form a constricted definition of the term, key. It has, for instance, a different connotation when it is used for record accessing. It can identify a single record or a set of records. It can be used for sequencing an access or the physical dataset itself. Part of the key (or all of it) can be used as an argument for a record or data search. There are primary and alternate keys, major and minor keys, simple, complex, and concatenated keys, and a host of other combinations. Rather than attempt to clarify them all, a "soft" definition will be used: a key is specified data items that identify a record and have a usage of record accessing.

Index structures

Serial accessing may need to identify a record, but there is no concept of keyed access. DMSII's capabilities of keyed access can be abstracted into whether or not it requires an index structure. The general form of index structures consist of entries containing a key and a record location pointer, although some have only a pointer and perhaps not even that. The major common characteristic is that physical data is required to develop the access. DMSII uses a separate file to contain this access data; this file is a index structure.

Index structures can be specified as a SET or a SUBSET in DMSII. SETs will have a entry inserted or removed automatically for each record creation or deletion in the referenced dataset. Entries in a SUBSET are either maintained automatically depending upon a condition in the dataset record (automatic subsets), or maintained by specific user function requests (manual subsets). Since a SET and a SUBSET are common in other respects the term, set, is often used for both types of index structures, and the term, spanning set is used to specify a SET. This terminology is less confusing at least.

Sets are organized, logically and physically, to provide some desired characteristics of the access. Random or sequential accessing, or both, or merely presence is one set of criteria. Physical resources consumed, disk, I/O, memory, redundancy is another; recovery, reconstruction, and reorganization yet another. DMSII has many types of organizations for index structures: INDEX SEQUENTIAL, INDEX RANDOM, ORDERED LIST, UNORDERED LIST, and BIT VECTOR. Index sequential is the most versatile and by far the most popular organization. The others are quite useful for special case optimizations.

Datasets and ACCESSes

In DMSII, the most simple form of dataset is the STANDARD dataset. This dataset puts new records at any convenient location, and the record will stay there until it is deleted or reorganized. The deleted record spaces are reused for subsequent record additions, therefore, STANDARD datasets maintain efficient utilization of disk space. Only serial accessing is allowed unless an indexed structure is used. The STANDARD dataset is easily the most popular type of data structure.

DMSII has other types of datasets which can also be accessed via key selection. These datasets will have an associated structure called, unfortunately, an ACCESS. An ACCESS structure, although not a physical file, has many of the same attributes and syntax usage as index structures, including describing the characteristics of the key. Dataset structures which have an ACCESS structure required are DIRECT and RANDOM.

A DIRECT dataset ACCESS limits the key to an 8-digit data item and uses its value as the record location. Both random and sequential accessing and all forms of file maintainence are very efficient. Disk space utilization depends upon the density of the values of the key, which is obviously quite restrictive. Transaction processing systems have increased the need for a fast random accessing function with no requirement for sequential access on that key. As disk space gets cheaper and its I/O time is not showing the same rate of technological advance as other processing components, the tradeoff toward reducing the number of I/Os per request becomes paramount. Therefore, the RANDOM dataset is gaining in popularity. A RANDOM dataset hashes the ACCESS key to a block where the record is found (or chained into overflow) giving an average I/O per request of slightly over one. One I/O versus a typical three I/Os for other types of random access is an extraordinary difference for a high volume, low response time, random processing environment. RANDOM datasets typically require more disk space than other alternatives.

Dataset structures keep the record location constant until the record is deleted or reorganized. Therefore, the referencing index structure pointers need no maintainence for the life of the record. This allows DMSII to provide any number of any type of index structures to reference these datasets. In fact, this is the typical implementation for inverted and interfile relationships. For instance, a RANDOM dataset ACCESS could provide quick specific access, while an INDEX SEQUENTIAL set on another key provides a sequential inversion, and yet another index provides record relationships via partial key access.

Of the several types of index structures and data structures there are three which are especially useful for developing modern database and systems design. RANDOM datasets are used for data with a high volume of single record, random access on a fully specified unique key. For other types of dataset usage, the more generalized STANDARD dataset can be specified. For sequenced access, interfile relationships, and other partial key access, the most versatile index structure is INDEX SEQUENTIAL. Other structure types provide excellent opportunities to optimize accessing or storage for special usage or specific data characteristics. However, for basic database design and for and simple, flexible, and efficient data and index structures only two or three structure types are needed.

DESIGN MODELS

The flat/relational approach

From a few years of experience with the first database systems, it is now recognized that it is less important to accurately implement a database model, and more important to provide a system that can react to change and growth. Finding out that a data management system could handle many small things better than even a few large things, reversed the direction of database design. Smaller files of normalized data, planned redundancy, symbolic rather than physical relationships, and simple, flat, disjoint physical file structure were found to be more effective.

Relational database is now the hot button that promotes this design philosophy. That may be the reason that nearly all the DMS vendors are now using the word, relational, when describing their current, and sometimes not new, DMS product. The following discussion will attempt to clarify the concepts of a relational database, a relational approach to a database, and a flat database as simply as possible and germane to DMSII.

Relational databases

There are several characteristics of the relational model that set it apart from the other methods of modeling data. It offers simple, clear, and understandable components and relationships. It is unfortunate that this has been obscured by a vocabulary of unnecessarily confusing terms surrounding relational software technology. Therefore, the first and perhaps hardest hurdle is the difference in terminology. Relational terms relate fairly well to DMSII and will therefore be obscure for those familiar with DMSII terminology. The following table correlates relational and DMSII terms.

RELATIONAL	DMSII
relation (table)	dataset
tuple (row)	record
domain (column)	data item in a record definition or perhaps, range in VERIFY clause
attribute	instance of a data item
degree	number of data items in a record
cardinality	current population
key (unique id)	key (access definition)

Figure 5 - Relational Terminology vs DMSII Terminology

The foundation of the relational data model is the RELATION or a collection of data. The concept of a relation is a two-dimensional table implemented as a fixed-format file. In the terminology of the relational approach, each record in the file or row in the table is referred to as a TUPLE, and each field in the record is known as an ATTRIBUTE. Tuples are often referred to as n-TUPLES, indicating "n" columns or attributes in the table. DOMAIN is similar to attribute, but there is a significant difference; a domain is a pool or set of values, an attribute is the use of a domain. Domain is often used to describe a column of the two-dimensional table. The DEGREE of a relation is the number of domains that make up the relation. The CARDINALITY of a relation is the number of tuples that exist in the relation. The cardinality of a file would be the number of records in that file. KEYs in the relational concept have nothing to do with accessing, but are defined as attributes that uniquely identify a tuple.

The following diagram presents the tabular view of a relation or dataset. There are "m" tuples (t1-m, cardinality "m") made up "n" attributes (a1-n, degree "n")

Or, in DMSII terms, there are "m" records (population "m") made of "n" data items in each record of the dataset RELATION.

RELATION:

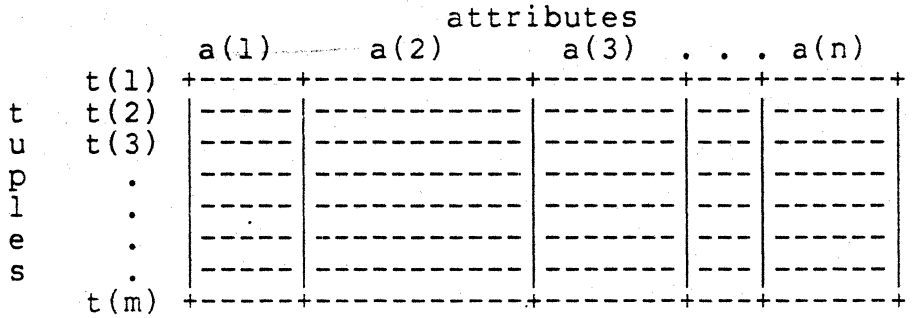

Figure 6 - Tabular View of a Relation

Once the terminology is understood, the more concrete characteristics of the relational database model are:

1) The data must be normalized to at least first normal form. In other words, a flat (non-embedded) logical and physical database is created.

2) A precise user view of the logical and physical database is defined. Relationships between RELATIONS are only developed symbolically via matching ATTRIBUTES.

3) It allows relational algebra and relational calculus operations.

4) It supports the use of a relational query language that forms the query and response as a new relation.

It must be noted that the theoretical relational database has no concept of access except tuple searching (serial access). Any attribute(s) may form the query (new) relation. Implementation then requires a low population, high primary storage environment to provide reasonable response. That is not the typical DMS environments found today, so some further refinements to the concept are required to make viable DMS products.

The relational approach

The relational approach then is a spectrum of implementations which relax the very precise definitions of the theoretical relational model and provide some pragmatic capabilities. The most obvious requirement is the definition of specific accessing capability. The least useful characteristics are dropped. They are the more exotic mathematical operations developed for two-dimensional tables, and to a lesser degree, the form of query and response.

So vendors have, in varying degrees, implied relational approach capability. All that is required is the ability to build a two-dimensional table. The rest, the vocabulary, normalization, symbolic relationships, and perhaps even query are methodologies and abstractions outside of the DMS implementation, but typically included within the relational approach.

To qualify as a relational approach, a DMS system should include most of the relational attributes. Certainly a strong normalization to a two-dimensional table concept is required. Some of the algebraic table manipulations, query, and terminology would be included.

Rarely does any fully implemented model or methodology address the real world complexity. Just as the relational model ignores the reality of accessing, the normalization methodology ignores access usage resource requirements and language usage requirements.

The real world of processor and I/O speed, memory limits, and COBOL language usage coupled with high volume, high population environments finds the relational approach wanting in a significant number of specific cases.

The flat database

The essence of the relational approach is the flattening of the structures and use of only symbolic relationships. The first tends to make several smaller pieces of a larger piece, and the second eliminates physical implementation of interfile relationships. Given just those two criteria, a data management model will produce most of the advantages of a relational model, and yet allow the flexibility to implement more pragmatic features.

A flat database is another loosely defined concept that has the literal meaning of simply no embedded structures. Perhaps a more useful definition would be a relational approach as described above without some of the restrictions implied by the methodologies and strict model features. There are several data item features, such as grouping and occurring, that are quite handy for COBOL manipulation but beyond the scope of normalization. Flat databases use the essence of relational concepts with considerations of the user environment which includes the DMS product features, the application language, the application usage and population, and the site hardware.

Normalization

It is sometimes hard to distinguish what form or feature belongs to the relational model and what belongs to the normalization methodology. One of the effects, if not the purpose, of normalization is to form two-dimensional tables on which the relational model depends. This structure provides some good news and some bad news, typically, good news for function A but bad news for function B.

The criteria for normalizing data can be described in another loose definition:

> "in each row (tuple, record), each column (attribute, data item) must depend upon the key, the whole key, and nothing but the key".

Key, in this case, is used for identification of the record. Data structures formed in this manner will be independent definitions of a single entity. The process of normalizing will likely form more and smaller entities with fewer maintainence anomalies as it progresses from unnormalized to third or fourth normal form.

The independence of these structures makes the functions of reorganization, recovery, and future conversions more reasonable. The users view is also more precise and therefore, better understood.

The fact that there are more entities increases the number of structures, not only for the dataset, but also for the probable index structure(s). This will increase the fixed overhead, but it may be more efficient overall due to more precise invocation and other usage. The same ambiguity can be recognized for the I/O. Smaller records are more efficient unless the application transactions require several entities to acquire the necessary data. Smaller entities will recover and reorganize quicker, and if they have errors, the failures will affect less of the database.

In the final analysis, the effectiveness of the normalized structures depends upon how well they fit the application usage. An even more definitive analysis may be how well the application usage fits the normalized structures. This leads to the conclusion that data analysis and structure design is the foundation of a flexible, responsive, and effective database system. Data usage, application programs, and system features are then built upon that foundation.

OLDER DESIGN MODELS

The database world did not, or perhaps could not, immediately convert all the existing systems this new design. At the present time there are many production programs whose database design is of an older genera. Two database design models emerged during the early years, hierarchical and network. Both are direct implementations of models that describe real data and entity relationships. Both use a relationship form that DMSII calls embedded.

Embedded structures

The concept of embedded is one of ownership. In this case, logical ownership directly and physically implemented by embedding one structure in another. It is a way of describing a one to many relationship. Every instance (record) of the owner structure may own instances of another structure. The former records are called masters, owners, parents, or ancestors. The latter are called slaves, members, children, or descendants. If the embedded structure is a dataset, then each master owns related records in the slave dataset. If an index structure is is embedded, then each owner record owns entries in the index structure which point to records in the referenced dataset. These referenced records are often called members of the relationship, however, their existence does not depend upon the relationship. Embedded datasets are used in the hierarchical model, while the network model uses the embedded index, or in DMSII terms, an embedded manual subset.

A disadvantage of this approach is that it uses physical record and block pointers as the only reference of a relationship. The index can be corrupted by any number of failures or errors of hardware, software, and even the user. This corruption could spell doom for a user who may not be able to reconstruct the destroyed relationships and "orphan" records. This problem is compounded when it involves highly populated relationships.

The hierarchical model

A hierarchical database is one whose relationships are implemented via tree-structured series of data sets. The root (master, parent, ancestor) may be described as data records which include in their description varying occurrences of other data records. In DASDL, the descriptions of these embedded datasets are included at the same level as a data item. A reasonable abstraction is an occurring group of data items implemented in a separate dataset. Each of these branches (slave, child, descendant) is considered to be embedded within the root dataset record. Each branch may in turn have its own branches. There is no restriction on the number of branches at any level or the number of levels. Each entity of the hierarchy is maintained in a dataset. The branch dataset is implemented as a series of incongruous blocks belonging to master records in the master dataset. The only means of accessing a record in an embedded dataset is through its master dataset

record.

Insert, delete, and update anomalies are abundant in this design. This approach also introduces unnecessary complications for the user with respect to programming and inquiry. There are true hierarchical structures in the real world and for these cases, the hierarchical model describes them nicely, but the direct, physical implementation of the model is not the most effective structures for the computer environment.

It is inherent to the hierarchical model that a record may not exist in an embedded dataset unless a master record exists. This is a major drawback in the hierarchical approach: accessing and maintaining the slave records is extremely difficult because it must be done only through the existence and access of a master record.

Although the hierarchical model solves some data redundancy problems, it can also create them. If a common instance is found at lower levels of a relationship that must be be duplicated for each occurrence, e.g., nuts and bolts in a parts description. Changing an attribute of a nut would require accessing every master in order to find every nut usage plus finding and making the necessary changes each of the nut records.

Many to many relationships, e.g., classes have many students :: students have many classes, are not reasonable to describe in this model. As an answer to these and other inherent problems, the network approach to data modeling was the next advance in database design.

The network model

A network model database consists of disjoint data sets where index structures are used to indicate the entity relationships. The owner structure may be described as common format data records which include in their description varying occurrences of reference pointers to data records in other disjoint data sets. Only this index structure (its relevant entries) are embedded in the owner dataset (record), the referenced dataset records are independent of the owner. Any number of owner records from the same or different data sets may make reference to a member record. A member record may also be a owner record in another or even the same relationship. If this sounds confusing, it is confusing and points out that clarity is not one of the better attributes of the network model.

By allowing all data sets to exist on the disjoint level, the problem of accessing and maintaining subordinate records is solved; all data sets may be accessed directly. The embedded relationship is still maintained through the use of manual subsets, thus preserving the physically nature of the implementation. Manual subsets do, however, require the application program to maintain that relationship, and therefore open the door for user error. The many to many relationship was easily solved by having each dataset defined with a list of related records in the other dataset. However, relationships of this type typically have attributes that belong to the relationship itself, and therefore, require a dataset anyway.

The network approach allows the modeling of multiple "n to m" relationships as it allows any referenced record to have multiple owners. But this adds complexity to the design, implementation, and user understanding. Records may be accessed concurrently from many different relationships which may (and at times, does) produce unpredictable results. Even though many of the insertion, deletion, and update anomalies that existed in the hierarchical model have been eliminated, new problems have been introduced to the deletion process.

A similar problem exists with the network design as was found to exist in the hierarchical approach, i.e., the occurrences of pointer corruption, the lack of recovery from severe faults, and the physical dependencies between large portions of the database.

A DATABASE DESIGN ALTERNATIVE

Both the hierarchical and network model are very reasonable approaches to forming the relationships between entities for many of the real world situations. They can both be used to form a complex structure relationship. Their implementations are sometimes performance effective for a particular usage. But for the general case of basic database design and for implementation of physical structures, they have been found to be severely limited and inflexible. The model may be useful for user views when more relational models are not appropriate. At the opposite end of the development, embedded structure's performance characteristics may fit a limited usage where the performance of the strict physical organization provides an expansion of a critical bottleneck.

The simple forms of the relational approach provide the database environment with the most adaptable and reliable structures. However, to be effective the usage of the data must be compatible with the structures. Changing data usage is a harder sale in that it affects the capabilities of the end-user. Even with user acceptance the rather low thresholds of volume, population, and response time lessen the feasibility of the relational approach.

Of the several approaches discussed so far, the flat design is the best basic design philosophy. The flat design approach eliminates the problems of uncorrectable physical pointers by using symbolic pointers (keys). It also helps simplify many of the complexities that are inherent to the use of embedded structures. A flat design has the freedom to deviate from the rigors and discipline of extreme methodologies found in the relational models.

Start with the best materials

Data analysis and structuring is the foundation of any information system. These beginnings should develop simple, two-dimensional tables that reflect the attributes of precisely indentified entities. The relationships between entities are described by data items in the referenced dataset. These enities are implemented by RANDOM datasets

where the access usage, disk, and I/O response tradeoffs are appropriate, otherwise, by STANDARD datasets. INDEX SEQUENTIAL spanning sets and automatic subsets are used to form the interfile relationships. These are also the structure types that best implement intrafile relationships.

Scratch only where it itches

Now develop the usage requirements. If possible, limit the usage to functions that effectively access the data. If the number of accesses are excessive for an unnegotiable usage, then it is possible to "unnormalize". This would mean collapsing structures or parts of structures into reasonable form for the accessing criteria. The possible anomalies created by this technique would need to be resolved. In fact, the whole capability/cost situation may need re-evaluation. Be careful not to over-optimize at this point. After real environment modeling has been analyzed, some further structure manipulations may be required, but do not give away future flexibility too easily.

Once the live environment is tested, there may be additional deviations from the normalized, and even the flat, design. Real live work giving not so live response times can lead to desperate measures (embedded structures). Hopefully, they would be limited to resolve specific bottlenecks, and their negative aspects fully understood.

If it feels good, do it

There are many other features, structures, and capabilities in a fully-featured system like DMSII. Some, like GROUP items and OCCURS, are oriented to the host language. Others may trade DBA control and DMSII general implementation for application program specific implementation and speed. Still others allow for physically tuning the file attributes for greater control or efficiency. The door ought to be left open to take advantage of any of the special features that fit a specific situation. Every site, application, usage, volume, population, hardware, etc. provides a different opportunity. Anything that restricts a solution to that problem must be carefully evaluated.

Use clay, not granite

Everything is changing. Users, vendors, hardware, software, design technology, it will change before the next reorganization. Database management systems are low on a steep technology curve. So, prudent users will think of the future in both short term and long term criteria. In the areas of design, and usage, and even optimization some rather broad outlooks must be considered. It must be recognized that today's concrete decision may be the subject of tomorrow's reorganization. Design in small, discrete parts; choose currently reasonable file attributes; optimize only where necessary; and hopefully, keep pressure upon any requirement/resource that causes a deviation from good design philosophy.

BIBLIOGRAPHY

1) Martin, James; Principles of Database Management; Prentice-Hall, Inc.; 1976;

2) Date, C.J.; An Introduction to Database Systems, Second Edition; Addison-Wesley Publishing Company, 1977

3) Martin, James; Computer Database Organization; Prentice-Hall, Inc.; 1977; Chapters 13-16

4) Barnhardt, Robert S.; "Implementing Relational Data Bases"; DATAMATION, October 1980; pp. 161-172

<this page left blank for formatting purposes>

TABLE OF CONTENTS