Data Base Management

What's It All About?

By
Mike O'Connell

Digital Equipment Corporation

Table of Contents

Preface

This book is intended for the technically oriented user who wishes to have a little light shed on the subjects of File Management and Data Base Management.

Before the magic words "data base" appeared on the scene, everyone seemed to understand what files were all about. But now, with common use of true Data Base Management systems on the horizon, confusion has replaced understanding. Many products on the market have been labelled "Data Base Management," when in fact they consist of nothing more than minor features thrown into the same old File Management systems that have been around since the early 1950s.

Part of the problem is terminology. Data base technology hasn't been around long enough for the industry to settle on a set of common terms that we all understand the same way. This book introduces some new terms and defines them in the manner that is becoming accepted in the industry.

The other major factor in the problem is the mystique of data base. No user wants to be told that his problem is simple enough to be handled by File Management techniques, so many software vendors have labelled pseudo data base systems as true data base systems, and the user is delighted to find that his problem is indeed a serious one, which can be solved by only the very newest technology. Data base is exciting; file management is old hat.

If you finish this book with the ability to discern the difference between a jazzy file system and a true data base system, and can also determine which type of system is required to solve a given user problem, then this book will have achieved its goal.

Please note that, although the author has been associated with Codasyl since 1968, the material in this book does not state the opinion of, nor imply support of, Codasyl.

Chapter

1 File Management

Although this book assumes that you know quite a bit about File Management, we'll review some of the basics just to provide a proper perspective for the more advanced material to follow.

File Management is the software that provides the user with the ability to manipulate files. Now, what are files? A file is a collection of data records related to each other in some way known to the user. For example, a file of inventory master records contains all information needed to describe the parts stocked by the user's company. But of course the File Management system doesn't have any idea of how the records in the file are related to each other or to other files in the installation. (Keep that in mind, because it is one of the characteristics that sets File Management apart from Data Base Management.)

With a File Management system, the user must ask the software to retrieve a record by providing it either with the location of the record in the file or with the unique contents of some pre-defined field in that record. When the user has retrieved the desired record, the job of retrieving the next desired record is no easier–the File Management system must again be given the same kind and same amount of information. File Management systems simply don't have the ability to recognize that some data records are logically related to other data records (possibly in a different file) as seen by the user.

We all recognize that the data records in the user's files represent some real events that have taken place in the conduct of the user's business. The user, whether a manager who uses output reports or a programmer, knows that various events in the conduct of the business are related to other events in the conduct of the business. For example, the user knows that a file of payroll master records is related to a file of time worked records and also to a file of labor distribution records.

But where must the user imbed the knowledge of those real world relationships? In the application program, that's where. The application program must contain the proper read statements and compare statements to make sure the right records from the right files are available to the program before it begins to make calculations and produce reports for management. The responsibility for insuring that the proper records contribute to the proper calculations is in the application program. Any other application program that must use those same records must have the same algorithm coded into it to insure that the same records are used. That process is time consuming and error prone, but we've lived with it for a long time. (Data Base Management overcomes that problem, among others.)

Let's look now at the three possible methods of organizing a file: sequential, relative, and keyed. A file's organization is specified at the time the file is initially created.

SEQUENTIALLY ORGANIZED FILES

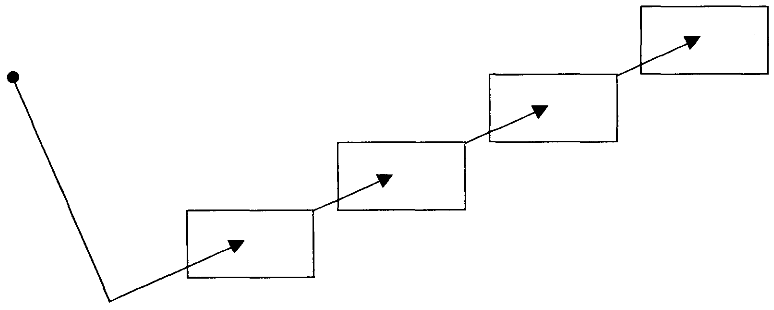
Sequentially organized files (Figure 1) are the oldest and simplest in the industry. Most devices–card readers, printers, magnetic tape, and terminals–support only sequentially organized files. A sequentially organized file is one in which each record is retrieved by successively retrieving all the records that physically precede it. Records can only be added to the end of a file, or deleted from the end of a file, because the physical location of each record is fixed in relationship to the record preceding and succeeding it.

Figure 1

SEQUENTIAL FILES

EACH RECORD IS RETRIEVED IN ORDER BY ITS PHYSICAL LOCATION.

EXAMPLE: GET THE NEXT RECORD.



RELATIVELY ORGANIZED FILES

Slightly more complex and powerful, relatively organized files (Figure 2) permit random access to their records. For this reason, these files are normally stored on a random access device, such as a disk. Each record in a relatively organized file is identified by its position in the file, relative to the beginning of the file. Each user request to retrieve a record must specify the relative record number of the desired record.

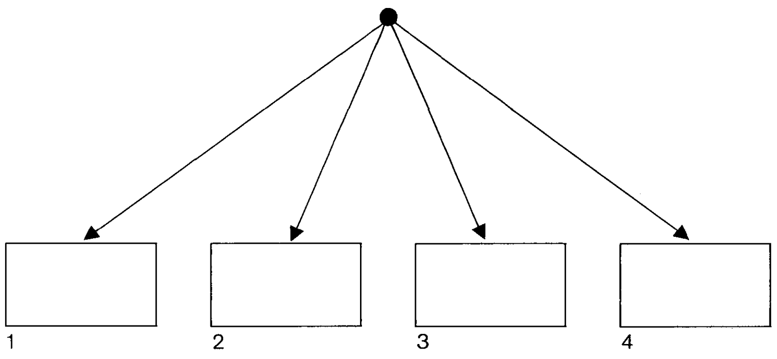
Like sequential files, relatively organized files depend on the physical placement of records for retrieval. Like sequential files, the records in a relative file may also be retrieved sequentially, by successively asking for record number one, record number two, etc.

Figure 2

RELATIVE FILES

EACH RECORD IS RETRIEVED RANDOMLY BY ITS LOCATION RELATIVE TO THE BEGINNING OF THE FILE.

EXAMPLE: GET THE 47th RECORD



KEYED FILES

Neither sequentially nor relatively organized files permit their records to be retrieved by any method other than physical location of the record. That isn't too handy for an application where the user needs the record for its customer Nabisco, but hasn't the foggiest notion where that record might exist in the file. Even if the file is in order by customer name, there is no relationship between customer name and the physical location of the customer's record. The user is stuck with reading the entire file sequentially until the right record is found, or, if the file is a relative one with the records stored in customer name order, using the hunt and peck method. Or a binary search could be used to try to zero in on the right record.

Users recognized long ago that neither sequential nor relative methods were too fruitful, so the third file organization, keyed, was invented. In a keyed file, each record is retrieved based on the contents of some field, called a key field, in the record. When the file is initially created, the field to be used as the key must be pre-defined and must remain the key field for the life of the file. This is the only file organization that allows information useful in the user's business, like an employee number or customer name, to be given to the File Management system with a request to find the matching record.

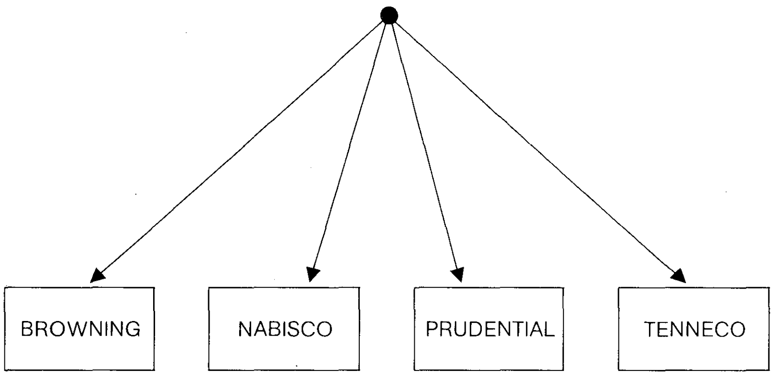
Keyed files (Figure 3) are implemented in two different ways, hashing and indexing, each having its strong and weak points. Let's look now at those two ways.

Figure 3

KEYED FILES

EACH RECORD IS RETRIEVED RANDOMLY BY THE VALUE OF A FIELD IN THE RECORD (THE KEY).

EXAMPLE: GET THE RECORD IN WHICH CUSTOMER NAME = 'NABISCO'



Hashing

Hashing is a term used to describe a software algorithm that takes some piece of user information and churns it into a relative record number (Figure 4). There are a lot of different algorithms for doing this conversion; the pages of the ACM publications are full of them. For our purposes, just believe that there are methods of transforming the name "Nabisco" into the relative record number "1369" or something comparable. From there on, the File Management system just treats the file as a relatively organized one, and retrieves the desired record by getting the right relative location.

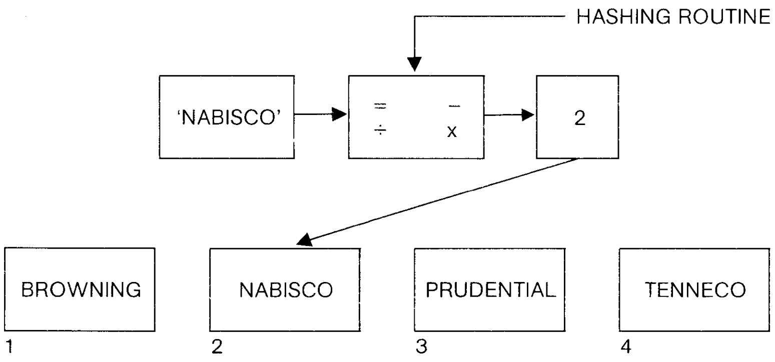
Hashing can be a very good method for rapidly retrieving records, because in most cases only one disk read must be performed to find the right record. However, complications in the hashing process can lead to many disk reads. It is not possible to guarantee in any hashing algorithm that two different inputs might not produce the same output. In other words, both "Nabisco" and "Prudential" might produce record number 1369, and both records cannot possibly be stored in the 1369th relative record position in the file. When this happens, we have what is known as the synonym problem.

The most common method of resolving synonyms when storing new records is to see if the desired record location is empty; that is, determine if some earlier record with a different key has already been stored in the spot. If it has, the File Management system searches for the first available location to put the new record. When retrieving a record, the record at the computed relative location is retrieved, and the File Management system determines if the contents of that record's key field matches the key asked for by the user. If they don't match, the File Management system searches the next available locations until the "matching" record is found. When synonyms occur often, the performance of the system suffers because so many disk accesses are required to store new records and to retrieve old ones.

Figure 4
HASHING

HASHING USES SOME MATHEMATICAL METHOD OF CONVERTING THE USER'S KEY VALUE INTO A RELATIVE RECORD NUMBER. THEN THE RELATIVE ACCESS METHOD IS USED.

EXAMPLE: GET THE RECORD IN WHICH CUSTOMER NAME = 'NABISCO'



Indexing
Indexing is a method similar to that used by book publishers. By definition, an index (Figure 5) is an ordered list of key words, each entry of which is associated with the location of the data in the book. If you have a history book and wish to look up information about Napoleon, you look in the index under "N" to get a page reference. Then, you can go directly to the right page and read about Napoleon.

An indexed file uses the same principle: the data records are arranged in a random physical order, but an index is maintained in order by key contents (Figure 6). Each entry in the index references a relative record number instead of a book page, but the idea is the same. If you want to know about your customer Nabisco, you ask the File Management system to retrieve the record in which the customer name field (the key field) is equal to "Nabisco." The File Management system consults the index, which is in order by customer name, then, after finding the right index record, uses the relative record number found there to retrieve the right data record. The data records themselves are, in effect, in a relatively organized file.

Note that random access to keyed files (both hashed files and indexed files) really uses relatively organized files. The only difference with keyed files is that the File Management system uses some additional facilities to get at the proper relative location in the file, instead of asking the user to know the relative location. Thus, the user can use values familiar to the running of the business, like "Nabisco" or "BOLT-93," instead of having to use relative record numbers like "1569" or "145."

One of indexing's weak points is that each request for a record means a disk access, probably more, to find the right index record, then one or more accesses to retrieve the desired record from the data file. This usually cuts down on the performance of indexed systems compared to hashed systems, but the indexed systems offset this by providing more power to the user.

One of the most useful functions available in an indexed file but not available with a hashed file is sequential access to the data records. By using the index, which is ordered by the values in the key field, it is possible to retrieve the data records in that same order. The File Management system simply reads one index record, turns the corresponding data record over to the user, then reads the next physical record in the index, which points to the next logical record in the data file. Thus, indexed files provide sequential access for the user, even though the File Management system is accessing the data records in a physically random order. Without the presence of an index, this would not be possible. For this reason, indexed files are usually referred to as Indexed Sequential files, or ISAM files. (ISAM means Indexed Sequential Access Method.)

You may have seen advertisements for hashed systems that claimed that sequential access was supported. What they mean is that you can read through the hashed file in physical sequence. That is obvious when you realize that the hashed file is really just a relatively organized file with some software to hash key values and generate relative record numbers. But retrieving data records by physical sequence is not the same capability as retrieving them in logical order by the contents of the key field. That capability exists only if an index exists for the file.

Figure 5
INDEXING

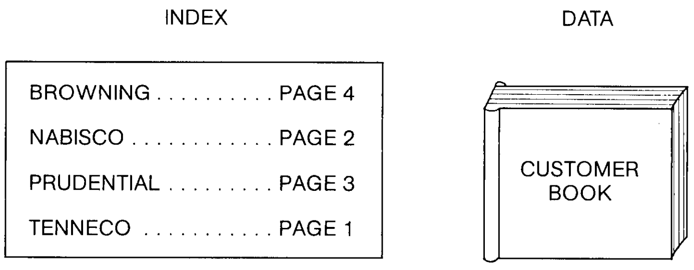
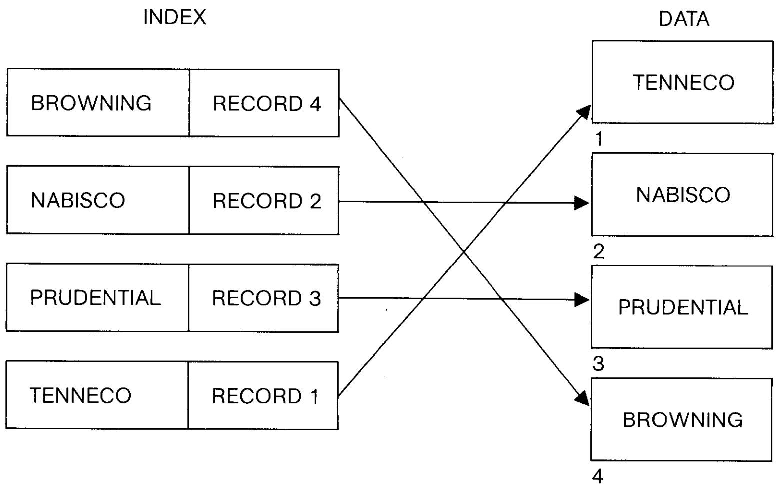

INDEX
DATA
BROWNING . . . . PAGE 4
NABISCO . . . . PAGE 2
PRUDENTIAL . . . . PAGE 3
TENNECO . . . . PAGE 1
CUSTOMER BOOK

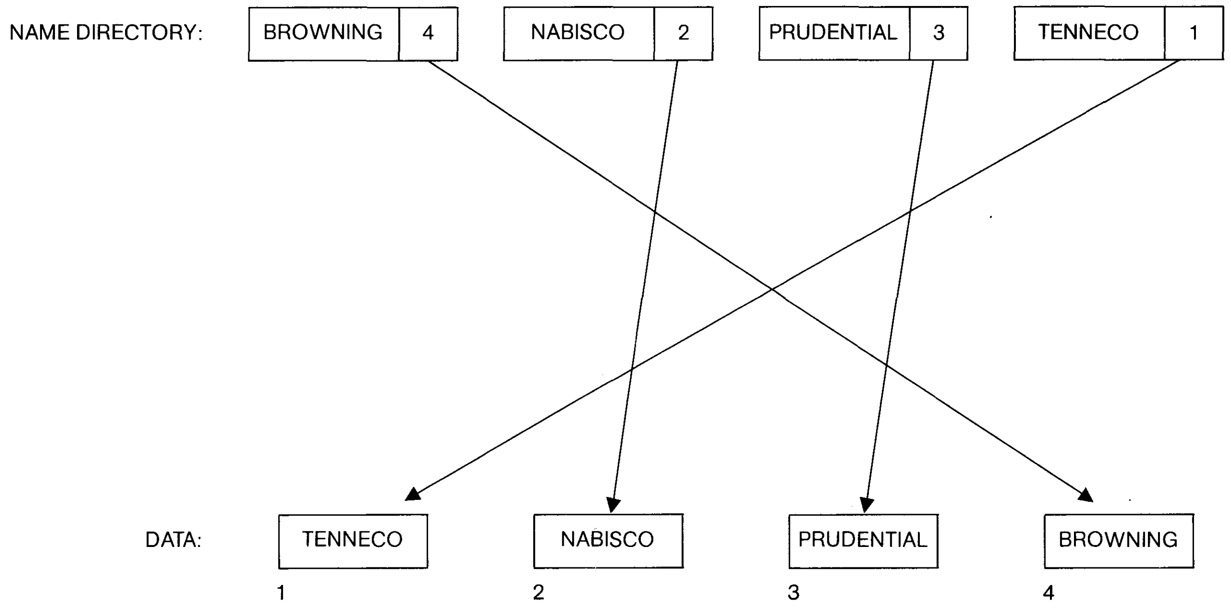
Figure 6
INDEXING BY KEY CONTENTS



SINGLE KEY ISAM

Now let's talk about some of the recent improvements that have been made in providing services to the user with ISAM files. In the beginning, there were ISAM files in which the user specified one key field. When the file was created and as it was maintained, the File Management system kept all the values contained in that key field in the index. But only one field in the data records could be the key field. It was somewhat like having a book index that referenced only people's names, but no place names or event names. To reuse the earlier example, if you wanted to look into your history book for references to Napoleon, the index solved your problem. However, if it was an index containing only people's names, any desire to look up references to Waterloo or insanity would be doomed to failure, unless you were willing to scan each page of the book looking for the references.

Figure 7
CUSTOMER NAME INDEX



MULTIPLE KEY ISAM

The solution to this problem is very simple, but it took years before vendors began to supply it. To see how the problem was solved, look at Figure 7. ISAM is implemented with an index (or directory, if you wish) for each key field. In Figure 7, there is a data file of customer records, with the customer name chosen as the key field. The File Management system builds a directory of names, along with relative record numbers.

That's what we've already seen. Now look at Figure 8. We've decided that we need to retrieve customer records by using the customer number as well as the customer name. Without changing the data file, or even the customer name index, we simply create a second index file which points to the same data file. The second index file contains entries for each customer number, and is kept in customer number order. A user who wants the record for customer number 600 would get the Nabisco record, as would a user who asks for the customer record for the customer name Nabisco.

This same idea can be used to create an index for every field in the data records, so that a user can retrieve a record by knowing the value in any one field of the entire record.

Note that this method, known as Multiple Key ISAM, does not provide the ability to access a record by more than one field for any one user request. For a given request, the user must specify one key field and one value to be retrieved in that field. Of course, the application program can be written to provide for more complex searching algorithms, such as "retrieve a record for customer 606 in state Massachusetts." The application program, using the ISAM file in which customer number is a key field, simply retrieves every record for customer 606 and compares the state field in each record to see which one contains the value "Massachusetts."

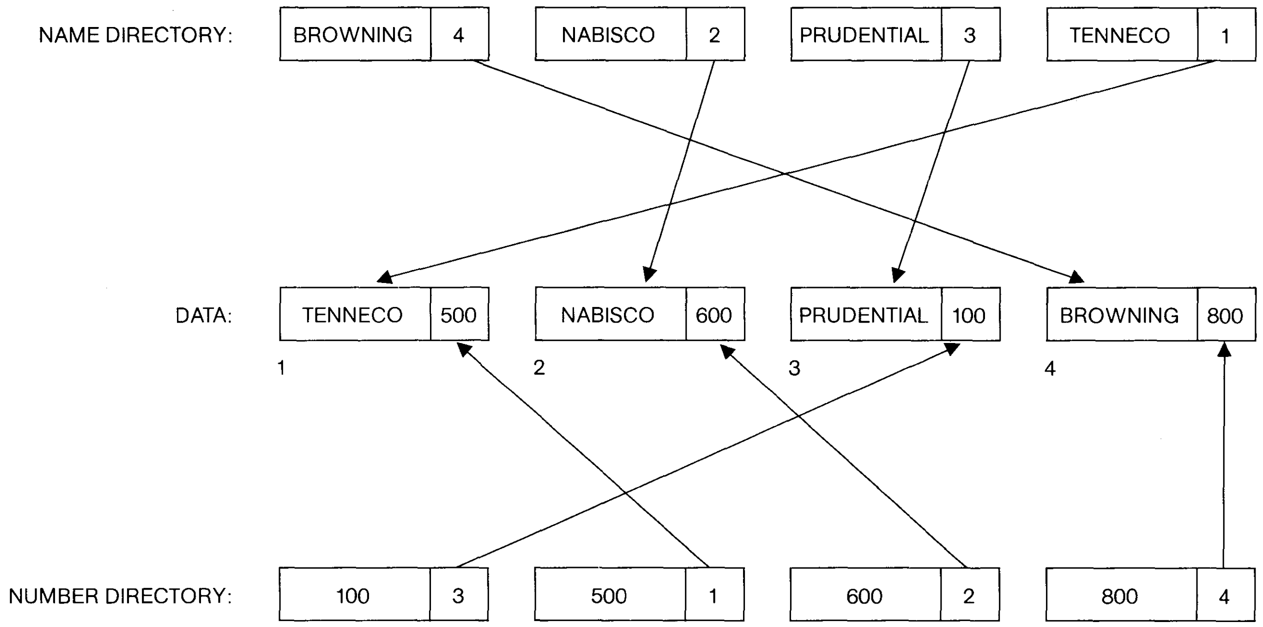
Building an index for a file is known as "file inversion," because the traditional process is to access a record and then discover the contents of its fields. With indexed files, that process is "inverted" in that the user already knows the contents of a field and wants to access the record. If not all of the fields in the data records are designated as keys, the file is said to be "partially inverted," and if all the fields are keys, the file is "fully inverted."

Multiple Key ISAM systems are now in wide use, with many products to choose from. Each product has its own bells and whistles, but they are all basically the same.

GENERIC KEYS

Some applications would find it useful to retrieve records without the program knowing the full contents of the key values. For example, a company could design its part numbers such that the first three digits of the part number represented the vendor from whom the part was purchased. If the inventory master records are stored in an ISAM file with part number as the key, normally the application would have to supply the entire part number to the File Management system in order to retrieve a record. In most cases, that isn't a problem, because most requests include the part number. But what about the application that needs to print a listing of all the parts on hand that were purchased from vendor 854?

Figure 8
CUSTOMER NAME AND NUMBER INDEX



To satisfy this need, File Management systems can permit "generic keys," that is, keys that consist of the value of a group of records, instead of values that identify only one record. To provide this capability, the File Management system accepts the short, or generic, key from the application program, then, using only the high order characters of the index, makes a normal search of the index. The first (lowest) index record that matches on the characters in the generic key value is the one used to retrieve for the user the proper data record.

In the example above, giving the File Management system the generic key value "854" would result in the application being given the record for part number 854000486, or 854000003, or 854098468, or whatever inventory record existed with the lowest part number starting with the digits 854. Once the File Management system has returned a data record to the user, further requests by the user can proceed as if the user had initially supplied the full key value. In this example, the application program, after printing the first record retrieved, could simply request each successive sequential record, until all records beginning with 854 have been retrieved.

APPROXIMATE KEYS

With an index, it is possible that the File Management system could return, to the program, records that contain the key value requested, or the next highest value if the requested value doesn't exist in the file. This is not difficult to provide; it simply means that the File Management system, in searching the index for the requested key value, stops when it finds either the requested value or a value that exceeds the requested value.

The ability to provide generic keys and approximate keys depends upon the presence of an index that is in order by key value. A file that is accessed by a hashing algorithm can provide neither of these capabilities, since the nature of the hashing method depends upon the user supplying a value that can be manipulated to yield a relative record number. Supplying either a value shorter than the one used to store the record, or an approximate value, simply cannot obtain the proper relative record number to retrieve the desired record. The ability to access records sequentially by their key values also depends upon the presence of an index that is built and maintained in that order.

If the File Management system was forced to search the index sequentially to locate the entry corresponding to the desired record, the performance of the system would not be acceptable. To overcome this problem, various methods are used to shorten the search time and the number of disk accesses required to find the proper index record.

The most commonly used method is to build a hierarchy of index records instead of a simple sequential index. The implementation and manipulation of these hierarchies can be quite complex, but the user need never concern himself with how the index is put together since the File Management system is responsible for controlling the index structure. The user simply asks the File Management system for a record containing the proper key, and lets the File Management system worry about how to locate it.

Chapter 2 Popular "Data Base Management Systems"

The last few years have seen the introduction of numerous systems labelled "Data Base Management" by their vendors. With few exceptions, these systems have provided essentially the same capabilities. This chapter will describe the concepts embodied in these systems and will compare them with the File Management techniques discussed in Chapter One.

Let's see what capabilities these systems provide to the user. First, these systems use two kinds of files. We'll call them Master Files and Detail Files. The data of interest to the user is stored in the Detail Files; the Master Files provide a means of gaining access to the Detail Files of interest.

MASTER FILES

A Master File contains records with one key field. Each Master File contains one record (and only one record) for each value that appears in one key field of a Detail File. Each record in a Master File also contains a pointer, which can't be accessed by the user, which points to a record in a Detail File which contains the same key value that the Master File record contains. In Figure 9, we see two Master Files: one called "Name," which contains a record for every customer name in our data, and one called "Number," which contains a record for every customer number in our data.

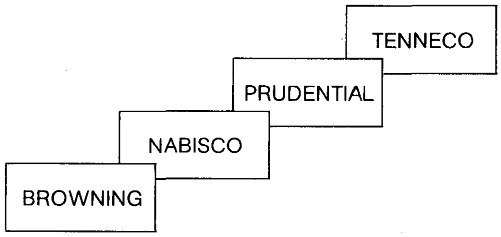
It is also possible for the user to place data in the Master File records. If the user chooses, then, a Master File could be defined to contain one key field, some non-key fields, and "hidden" pointers to Detail Files.

Because Master Files are keyed files, either a hashing or an indexing approach could be used to access their records. Vendors have chosen hashing in every case, for performance reasons. To access the desired record in a Detail File, the user must first access the proper Master File, in order to find the record there containing the proper key, which contains the pointer to the proper Detail File record. The designers of these systems have chosen not to further add to the number of reads required, which would have occurred if the Master File were an indexed file.

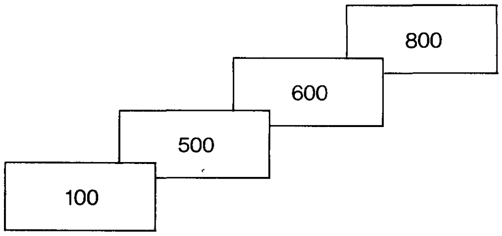
Hashing, then, is used with the hope that the I/O overhead will be less than that required for an indexed file, but if the synonym problems mentioned in the last chapter occur, even the hashed file will suffer performance penalties.

Figure 9
MASTER FILES

- EACH MASTER FILE HAS ONE KEY FIELD.
- EACH RECORD IN A MASTER FILE HAS A UNIQUE KEY VALUE.
- HASHING IS USED TO ACCESS ITS RECORDS.



MASTER FILE: NAME
KEY FIELD: CUSTOMER NAME



MASTER FILE: NUMBER
KEY FIELD: CUSTOMER NUMBER

DETAIL FILES

Detail Files contain the "real" data of interest to the user. Each Detail File can have several key fields. In Figure 10, the Detail File called Customer has two key fields: customer name and customer number. The Detail File is a relatively organized file, with the records stored in random order.

For each key field specified in a Detail File, there must be a Master File with the same key field. If two or more Detail Files have the same key fields, it is possible to build only one Master File for those key values and let each Master File record point to a record in both Detail Files that contain that key value.

Each record in a Detail File also contains a "hidden" pointer, accessible only by the system, for each key field in the record. The pointer points to the next record in that file which contains the same key value in that key field. That means that all the records with the same key value in a given key field are linked together through these pointers, and the Master File record for that key value points to the first Detail File record in the group.

Now let's put the three files in our example together (Figure 11). We have a Detail File (Customer) for which we have specified two key fields: customer name and customer number. We have a Master File (Name) in which each record has as its key a customer name, along with a pointer to the relative record in the Detail File for that customer name. We also have a Master File (Number) for which customer number has been defined to be the key.

By turning over to the system a key value, the user of these systems can request that the system find a desired record in a specific Master File. The system hashes the key value, uses the resulting relative record number to access the Master File, and insures that a synonym is not retrieved. The user can then request that the system retrieve a record in a specific Detail File containing that same key value. The system, using the pointers contained in the retrieved Master File, retrieves and turns over the requested record.

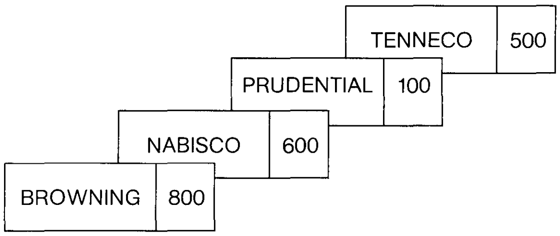
The user can also specify that another Detail File record with the same key value then be retrieved. The system, using the pointers in the Detail File records, can then retrieve every record in the Detail File that contains that value. If the user requests a Detail File record with a different value, the system must again hash the new requested key, locate the proper Master File record, and start over again in the Detail File.

A powerful capability, that's true. But instead of calling these files Master Files and Detail Files, let's change terminology a bit. Let us call the Master File an Index, and call the Detail File a Data File. Let's then call the "Data Base Management System" a new name: Multiple Key ISAM.

We've already seen that Multiple Key ISAM permits data records to be retrieved based on the values in any number of key fields through use of an index. These "Data Base Management Systems" permit data records to be retrieved based on the values in several key fields through use of a Master File. So far, the two kinds of systems are functionally identical, but let's look at the other functions performed by Multiple Key ISAM systems.

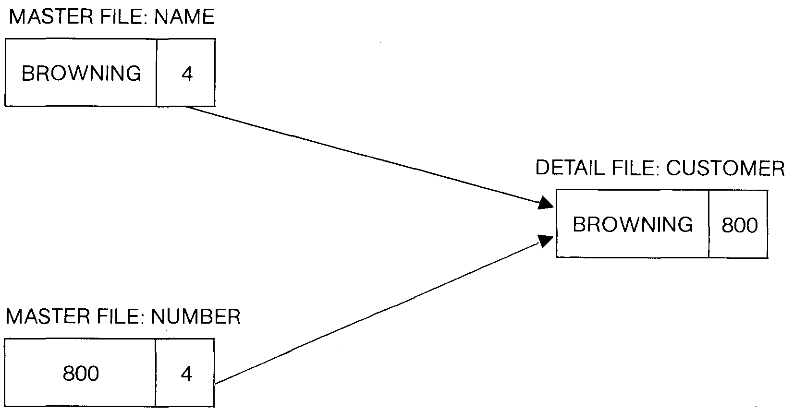
Figure 10
DETAIL FILE

EACH DETAIL FILE CAN HAVE MULTIPLE KEY FIELDS.
KEY VALUES MAY BE DUPLICATED.



DETAIL FILE: CUSTOMER
KEY FIELDS: CUSTOMER NAME
CUSTOMER NUMBER

Figure 11
ALL THREE FILES



With Multiple Key ISAM, the user can retrieve records sequentially based on the ascending values in a key field. With these "Data Base Management Systems," all the data records containing the same key value can be retrieved, but there is no mechanism available to find the next highest key value in the data. ISAM simply accesses the next sequential record in the index; these other systems don't have an index in order by key value. Instead, their equivalent of the index, the Master File, is a hashed file in which the records are in random sequence.

Remember the use of generic keys with ISAM? These "Data Base Management Systems" can't provide generic key capability. The user of these systems must provide the system with the full key value in order for the hashing algorithm to generate the same relative record number that it generated when the Master File record was originally stored in the file.

How about approximate keys, which were no problem with the aid of the ISAM index? Not available with these systems, because the user who provides to the hashing algorithm a key value that doesn't exist in the Master File isn't going to find a record in that Master File that contains the next highest key value.

Aside from functionality, the user's job is usually more complex with these "Data Base Management Systems," because the user must be aware of, and may have to maintain, the Master Files. With ISAM, of course, the File Management system is responsible for knowing about the indexes and using them when necessary.

Well, there you have it. Some vendors have built Multiple Key ISAM File Management Systems, and some vendors have built systems almost as useful and called them "Data Base Management." Given your choice, which do you think stands the better chance of doing a company's job?

So far, all we've discussed have been File Management systems, regardless of what we've called them. This book has promised to talk about real Data Base Management, so let's get to the heart of the matter.

Chapter 3 Data Base Management

The idea of data base management has been around since the early 1960s. Like any other topic with the word "management" in its title, its purpose is to provide a company's management personnel with the ability to better manage the data asset owned by the company. And data is an asset. There has been a growing awareness of that fact over the past decade. Think about it: what does it cost your company to record data, file it, update it, delete it? And what would it cost your company if that data were lost or otherwise tampered with? Would your management be misled into serious management decisions if the data in your computer installation weren't correct or up to date?

That growing awareness led to work in trying to find better ways for management to control and use its data asset. A data base management system then has the following characteristics:

- It provides protection from persons who would tamper with the data (or its definitions).
- It provides a single copy of each piece of data, for the use of the entire installation, instead of multiple copies gathered by multiple means, each used by only one application system.
- It provides a unified description of all the data asset in the company, so that all portions of the company that need to use that asset will have access to it.
- It provides a separation between data and its description, on the one hand, and the application programs that manipulate it, on the other. This provides the ability to change one without changing the other.
- It provides for the definition of the logical relationships which exist between the various records in the data base, so that each and every application program need no longer embody those definitions in the logic of the program.

EVOLUTION FROM FILE MANAGEMENT

Let's look at the conceptual steps in moving from a File Management environment to a Data Base Management environment.

Figure 12 depicts the first step: separating the data definitions from the programs. It is now recognized that one of the worst mistakes we made in the infancy of programming technology was to imbed the definitions of the data inside the definitions of problem-solving algorithms (programs). That mistake was made in the interests of being able to compile programs efficiently by knowing at compilation time what the object time data would look like. There is no logical reason, relating to the nature of problem solving, that requires that the data definitions reside with the program.

The program is a statement of an algorithm for solving some user's problem. We wouldn't write programs if there wasn't a problem to be solved by the program. But the problem solution does not depend upon the format or size of the data. Calculating an employee's gross pay does not depend upon how many decimal places we record on that employee's time card or on how many digits there are in that employee's hourly rate. The calculation is also not altered by whether or not we store those numbers in ASCII, binary, or packed decimal format.

So, for our first step, let's conceptually rip the data definitions apart from the rest, or procedural part, of the programs in the installation.

The next step (Figure 13) is to combine all the installation's data definitions into a machine-readable file called the "Schema." This process will require that all the duplicate data definitions that existed in the many different programs be eliminated so that the Schema contains one and only one definition for each and every data item (field) that is to be in the data base.

This process also must result in an objective definition of each data item, instead of the subjective definitions found in application programs. For example, the same magnetic tape file may be read by two different programs. One of those programs may contain a definition of a data item on that file that is called DATE and is six characters in length. The other program may contain a definition for the same character positions on the file but may describe those six characters as a two character data item called MONTH, another two character data item called DAY, and another two character data item called YEAR. In this example, it's clear that each program contains definitions that are consistent with the other, but other cases may not be so obvious.

The point is that programs contain data definitions that are colored to suit that particular program's viewpoint of the data. Nowhere in the File Management environment is there a true, objective, and uncolored definition of the data. In the Data Base environment, the Schema contains the objective data definition for each data item in the installation's data base.

Figure 12
SEPARATING DATA DEFINITIONS FROM PROGRAMS

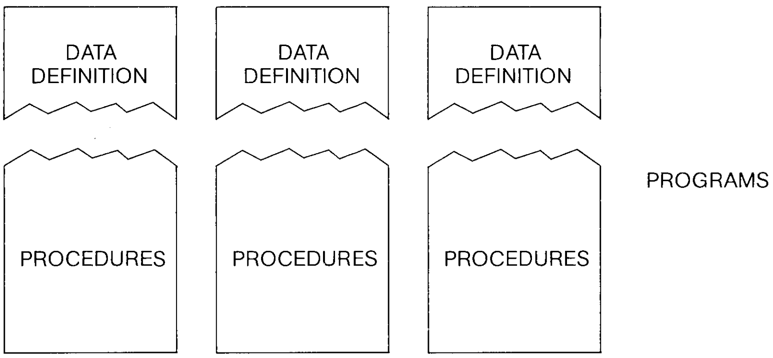


Figure 13
CONSOLIDATING THE DATA DEFINITIONS INTO A "SCHEMA"

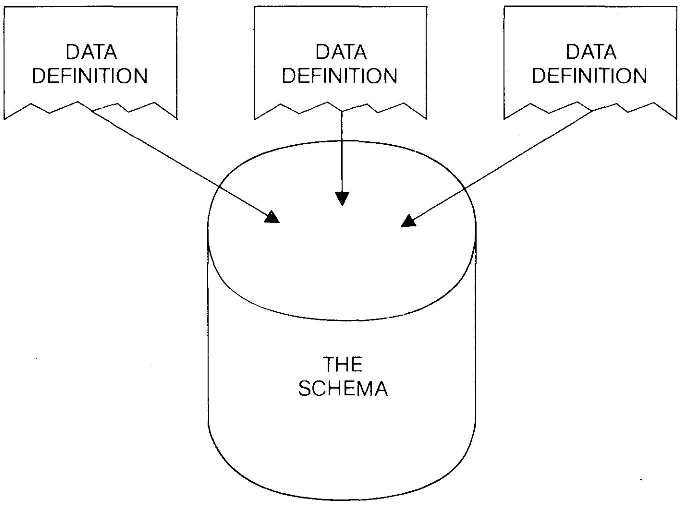
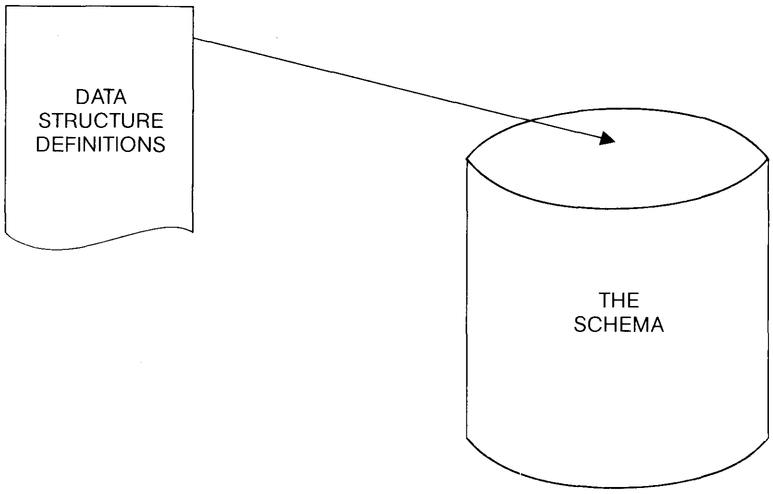


Figure 14 shows the next step in our evolution: the creation of definitions of the relationships that exist between the various records defined in Schema. Remember that we pointed out earlier that the real world relationships existing between the events represented by the data were not known to the File Management systems. One of our goals in a data base environment is to define for the Data Base Management system just what those relationships are, so that we don't have to explain them over and over to application programmers, and the application programmers don't need to code the statements to see that the proper records are used in conjunction with each other.

The data structure definitions that we put into the Schema complete the Schema. The Schema now contains both the definitions of the content of the data base and the definition of the structure of the data base. Using the Schema, the installation can generate dictionaries containing all the information known about the company's data asset. Through use of these dictionaries, systems analysts and users can determine what data is available in the data base before designing new applications. Thus duplicate effort involved in collecting the same data more than once for more than one application system is eliminated.

Figure 14
ADDING DATA STRUCTURE DEFINITIONS TO THE SCHEMA

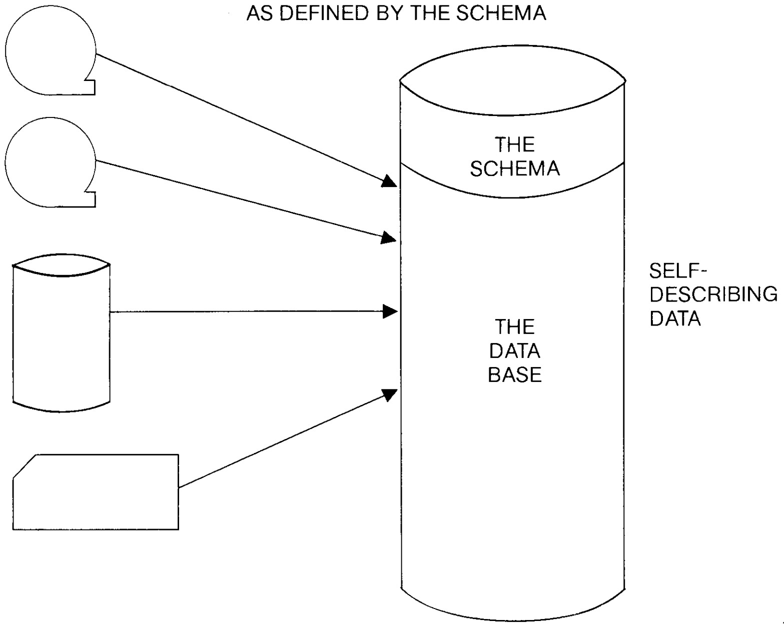


We can now create the data base, as shown in Figure 15. As the files are moved into the data base, the system uses the Schema to determine how the data is to be stored and what relationships exist between the various records going into the data base. The Data Base Management system must construct some internal method of guaranteeing that the records that are a part of a relationship are bound together for use by the applications. The system thus builds whatever kinds of internal tables, pointers, links, or indexes it needs to implement the defined relationships. These internal aspects of the system are never seen by the application.

The process of moving files into the data base is not a simple copying operation. Just as each program had its own data definition, each application had its own files containing some of the same data contained in files belonging to other applications. Our goal is to eliminate redundant data, so we must determine which copy of each data item goes into the data base. That can be a long and tedious process, similar in nature to the older problem of converting old applications to new ones. Questions like the timing of the installation of the new system and the installation of the new programs must be carefully coordinated.

Figure 15

CONSOLIDATING THE DATA FILES INTO A DATA BASE, AS DEFINED BY THE SCHEMA



Now let's turn our attention back to the procedural parts of the programs that we left hanging a few steps ago. By removing the data definitions from the programs, we removed the ability of those programs to access any external data. We must now restore that capability, which we do by defining Sub-schemas (Figure 16).

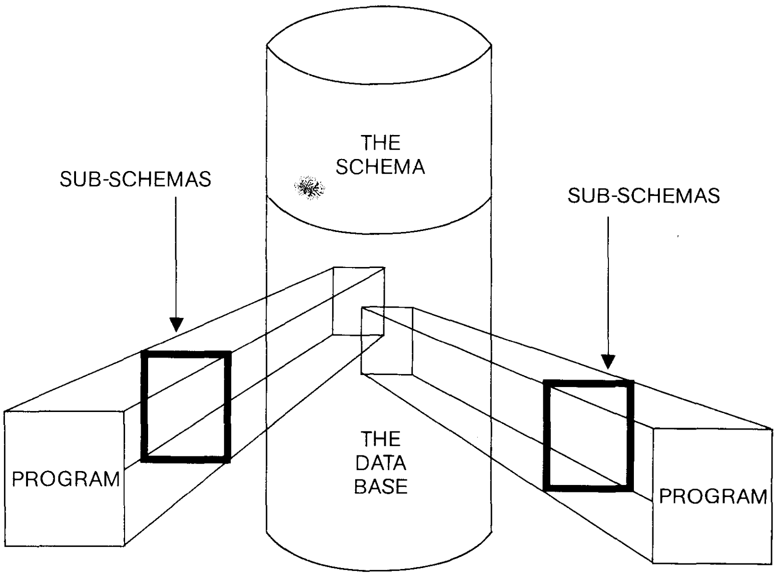
A Sub-schema defines the parts of the data base that are to be accessible to a program, just as the data definitions that were in the program defined what part of the installation's files were to be accessible by that program. The major difference is that the Sub-schema is not a part of the program, but is written separately.

A Sub-schema may be viewed as a "window" through which an application program sees the data base. A copy of a Sub-schema listing must be given to the application programmer so that he can see what data and what relationships are known to the program being written.

The Sub-schema also performs another major function. It provides subjective data definitions removed by building the Schema. The Sub-schema permits not only the definition of what is to be available, but also its format. For example, an application may prefer to group the data items in a record in a way different from the way they appear in the data base and are described in the Schema. By writing the Sub-schema to show the application-desired grouping, the responsibility for rearranging the data items as they are read and written is assumed by the Data Base Management system.

Figure 16

DEFINING EACH PROGRAM'S VIEWPOINT
OF THE DATA BASE IN "SUB-SCHEMAS"



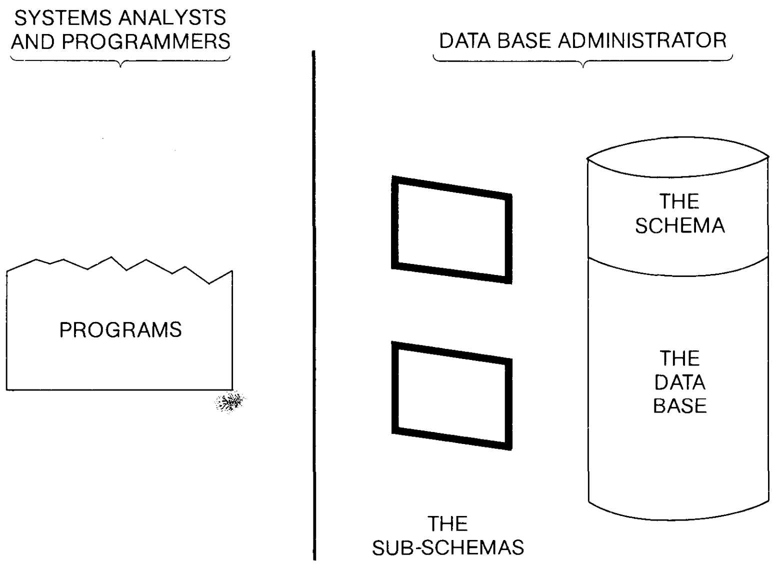
Now we're down to the essence of data base management: the ability of a company to manage its data asset similar to the way it manages its cash asset or its capital assets. By providing the necessary software, a Data Base Management system provides the tool necessary for management to institute a program of data management (Figure 17).

By separating programs (that is, the statement of problem solutions) from the definition of the data operated on by those programs, we can separate the responsibility for the creation and maintenance of both. This requires the creation of a new function in the data processing world, that of the Data Base Administrator.

The Data Base Administrator is responsible for the care of the company's data asset, in the same sense that the company's treasurer is responsible for the care of the company's cash asset. The Data Base Administrator creates the Schema, the Sub-schemas, the data dictionaries, and the data base itself. The Data Base Administrator determines which users can access data, change data, and delete data. Of course, this can't be done without input from the systems analysts and programmers, just as the treasurer can't manage cash assets without input from those who have a need to use the cash.

Figure 17

MANAGING THE DATA AND ITS DEFINITION INDEPENDENT OF PROGRAMS



INDUSTRYWIDE DEVELOPMENT

When the idea of data base began to catch on in advanced circles a decade ago, it lit a lot of fires under software designers and developers in lots of companies. Like any new idea, it was exciting. It led to lots of experimentation to see how to apply this new idea to solving real business problems. Unfortunately, like all new ideas, it led to a lot of terrible implementations and a general confusion about what was good and what was bad. Let's compare what has happened over the last decade with what happened to programming languages over the preceding decade.

When programming languages were in their infancy, literally thousands of them were invented and implemented. Most of them were never heard of outside their immediate area of birth, but many of them got broader use. Gradually, the computing industry saw that we had built our own Tower of Babel; we were prevented from performing to our potential because of the proliferation of languages. As a result, the less useful languages died out and the survivors were standardized, so that we were all talking the same dialect of the same few languages. Today, COBOL and FORTRAN have been standards for years, and both PL/I and BASIC are in the latter stages of becoming standard languages.

In the world of data base facilities, we are seeing the history of programming languages repeat itself. The early proliferation of languages and data structures is dying out, and the industry is zeroing in on a single, generally useful facility. The work being done to bring this about is in an organization called The Conference on Data Systems Languages, or to use the acronym, Codasyl.

Codasyl Activity

Codasyl was formed in 1959, when the Pentagon called together some computer manufacturers and computer users for a two-day conference. Their goal was to determine the feasibility of designing a programming language that would permit the writing of business programs once and then translate them to many different computers. Today, the idea is so commonplace that we forget that it was revolutionary in its day. The result of the initial meeting was an informal organization, Codasyl, consisting of those companies who volunteered their time, money, and personnel to work toward the definition of a language that could achieve that goal. COBOL was the language that emerged from that work.

COBOL has been around now since 1960. So has Codasyl, but Codasyl devoted its attention to more than just COBOL over these years. Although the COBOL Committee of Codasyl meets every six weeks, other committees responsible for other aspects of computing systems are meeting just as often and making the same kind of contributions to the industry.

Codasyl's data base activity started in 1965, when the COBOL Committee formed its Data Base Task Group (DBTG), and chartered it to develop language facilities to bring data base capabilities to the programming world. Now, more than a decade later, Codasyl has given its stamp of approval to a set of languages and data structures designed to bring a single system to the marketplace that would serve the need of all users, just as COBOL serves the need of business programmers.

Codasyl Implementations

Prior to final approval by Codasyl, however, there were a number of draft documents published so that the general computing public could follow its progress. Figure 18 shows the chronology of Codasyl's data base work. It also shows the initial availability dates for some widely used Data Base Management systems.

As you can see, Codasyl published drafts of its work in both 1969 and 1971. The Schema facilities were finally approved in 1973, and the Sub-schema facilities and the extensions to COBOL for accessing the data base were approved in 1975.

Now look below the time line. The systems marked with the asterisk are those systems designed in accordance with the Codasyl specifications. As you can see, since the Codasyl specifications began to take shape in the early 1970s, no vendor has seen fit to invest major amounts in the design of data base systems that are not based on the Codasyl work. In fact, quite a few of the major vendors have seen fit to introduce new products based on that work. Figure 19 shows more detail of the Codasyl-based systems.

No one, least of all the developers at Codasyl, thinks that the Codasyl work is perfect. But it's the only set of specifications that has been researched for twelve years and implemented by more than one vendor. In other words, it's the only game in town, and the industry is working to constantly improve the facilities defined in that game. Digital Equipment Corporation is a part of that activity; the other vendors are Cincom Systems, Control Data, Honeywell, IBM, International Computers Ltd., Univac, Cullinane Corporation, Prime Computers, CII-Honeywell Bull, Computer Sciences Corporation, and the University of Florida.

The Schema

The Schema, as we have seen, is the definition of the content and the structure of the data base. The Schema is written by the Data Base Administrator in a language defined by Codasyl. This language is totally independent of the programming languages that might be used later to access the resulting data base.

We won't go into the syntax of the language, but it looks somewhat like the data definition statements in COBOL or PL/I. The Schema consists of record definitions, data item definitions within those record definitions, and the new concept, the definition of the structure of the records (the record interrelationships).

It's time now to introduce the term "set." A set is a collection of records that are related in some way that is useful to an application program. For a payroll application, a useful set of records might be a payroll master record, a time-worked record, a year-to-date earnings record, and a deduction record. For a labor distribution application, a useful set might be the same payroll master record, a contract master record, and a labor distribution record.

The relationships between records, as defined in the Schema, are defined in terms of sets. The Schema, therefore, consists of a series of record definitions and a series of set definitions.

Figure 18
DATA BASE CHRONOLOGY (CODASYL)

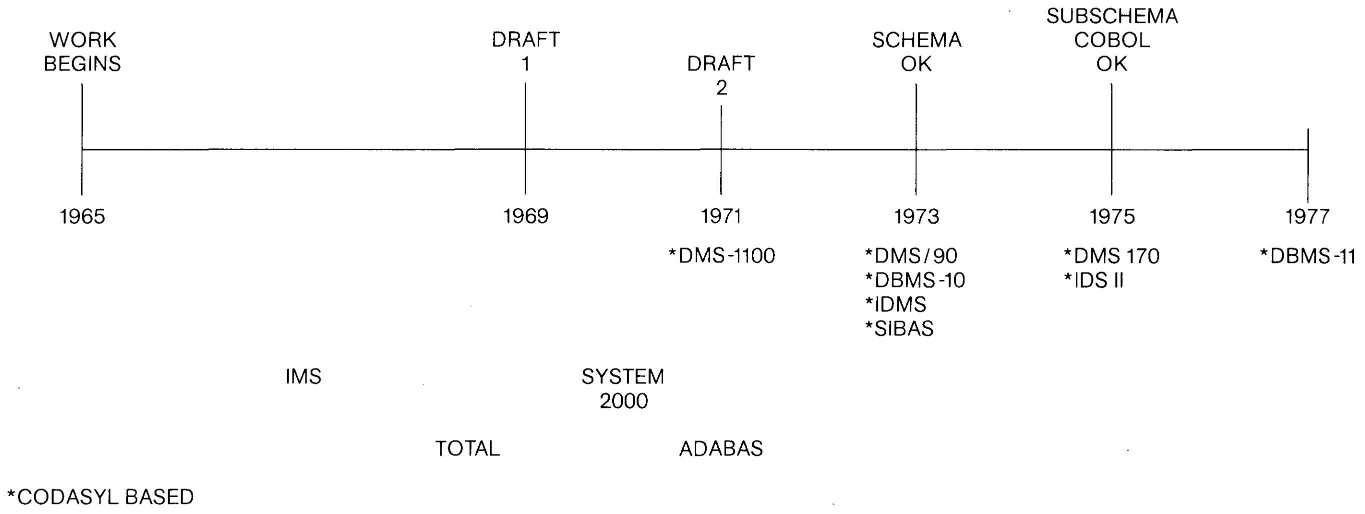


Figure 19
CODASYL DATA BASE SYSTEMS

NAME	RUNS ON	AVAILABLE
DMS-1100	UNIVAC 1100 SERIES	1971
DMS/90	UNIVAC 9000 SERIES	1973
DBMS-10	DEC SYSTEM-10	1973
IDMS	IBM 360/370 SERIES	1973
	ICL 2900 SERIES	1975
SIBAS	IBM 360/370 SERIES	1973
	UNIVAC 1100 SERIES	1973
	DEC SYSTEM-10	1973
	NORD	1973
DMS 170	CONTROL DATA CYBER 170	1975
IDS II	HONEYWELL 6000 SERIES	1975
DBMS-11	DEC PDP-11 SERIES	1977

The Sub-Schema

The purpose of the Sub-schema is to specify what part of the total data base is to be available to an application program. To do that, the Sub-schema consists of a series of record and set definitions, just as the Schema did. However, the Sub-schema may list only records and sets that have been defined to exist in the Schema. In other words, the Sub-schema cannot be used to define new kinds of records or sets.

The Sub-schema for a given application program will not list all the records in the data base, will not list all the sets in the data base, and might not even list all the data items that exist in the records that it does list. The Data Base Administrator, in writing the Sub-schema, has the ability to specify that a certain kind of record only has, say, five data items in it, rather than the seven data items that the same record in the data base has. This capability means that the Data Base Management system must "filter" the record as it is retrieved, so that the application asking for the record will only get that part of the record that its Sub-schema says it is to get.

The language used to write the Sub-schema also looks much like the data definition parts of a COBOL or PL/I program.

DATA BASE STRUCTURES

There are three types of classic data base structures (Figure 20). In that figure, each small circle represents a type of record, such as payroll master or customer record. The lines between the various types of record represent the logical relationships between them.

The simplest is the sequential structure, also called the chain or ring. In this structure, each record type is related to two other types of records. Note the similarity between this type of structure and the sequentially organized files with which we're familiar.

The next most complex is the tree or hierarchical structure. In this structure, each record type may have following it more than one other record type. The tree gets its name from the picture, which looks like the roots of a tree. Note here that, as in tree roots, once a branch is taken, there is no way to get to the records on the other side of the branch without returning to the branch and taking the other path.

The most complex form of structure is the network. Here, any record may be related to any other record, without the restrictions mentioned above. The network is a superset of the tree. Trees are usually imbedded within networks, just as chains are usually imbedded within trees.

One or more of these three structures are associated with all Data Base Management systems.

Figure 20
DATA BASE STRUCTURES

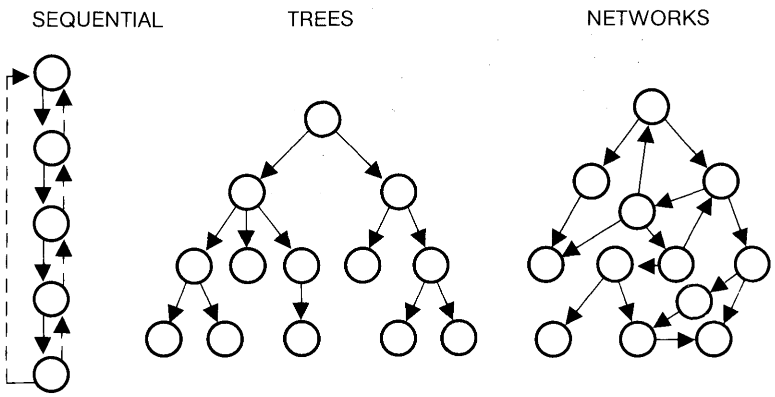


Figure 21 shows a typical tree data structure, with each lettered box representing a specific type of record. If the problem to be solved involves data that has a natural hierarchical structure, this form is very good.

Because a tree doesn't support a network, there are some very definite rules that must be obeyed by the Data Base Administrator when designing the data base. Referring again to Figure 21, if an application needs to use both the record K and the record A as branches subordinate to record B, the Data Base Administrator would have to find some way to change the entire configuration of the tree, because the desired structure is not legal, given the tree as presently defined. For that reason, the Data Base Administrator in a tree installation must draw a map of the data base structure, much like the one in Figure 21, before actually beginning to code the Schema. This will insure the legality of the required structures.

Figure 21
TREE

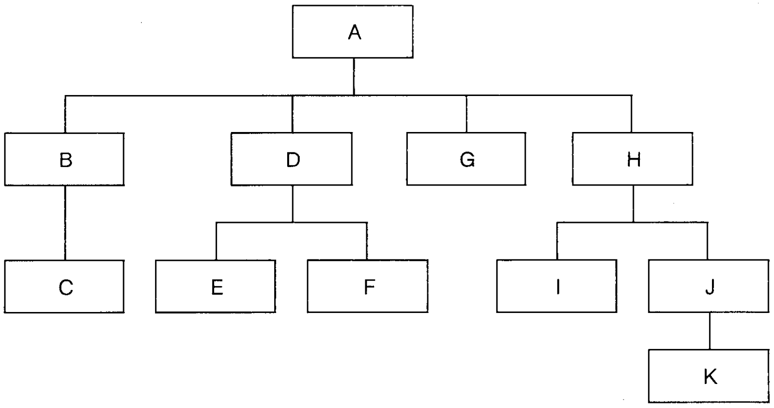
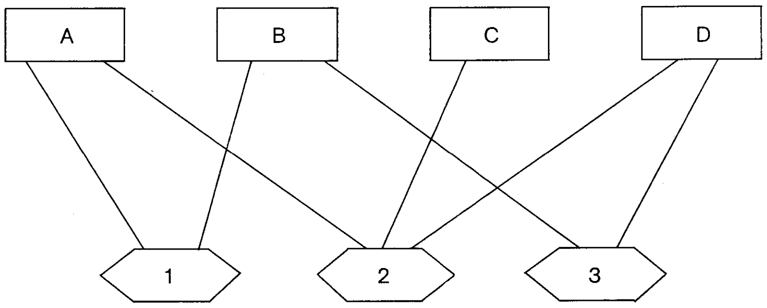


Figure 22 shows a typical network as supported by several popular systems. Again, each box represents a type of record and is labelled with a record name. The "master" records are rectangles labelled with letters; the "detail" records are hexagons labelled with digits. In this example, record A is related to both record 1 and record 2; record 2 is related to record A, record C, and record D.

Figure 22
NETWORK



Remember, though, the analogy to Multiple Key ISAM. With these systems, it is not possible to define a relationship between two master files. In other words, if an application would like to see a set consisting of records 2 and 3, it could not be defined. The application would have to use record D as the common link between the records desired.

With these networks, as with trees, the Data Base Administrator must map the data base before beginning the job of coding the Schema, because of the restrictions that exist in not being able to define any possible relationship.

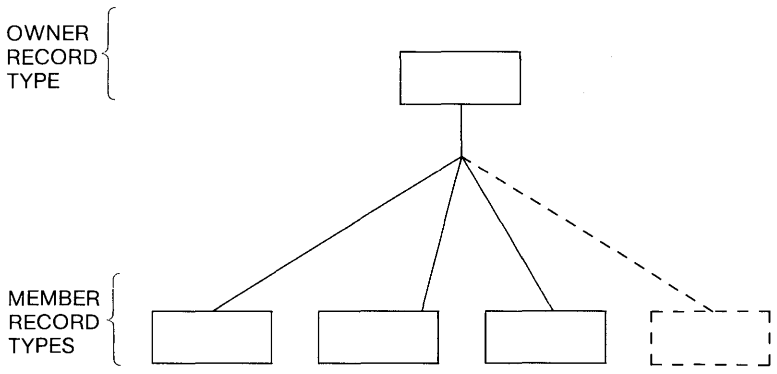
All the pre-Codasyl data base management systems had this sort of restricted capability to define record relationships. Even worse was the mess that occurred when a change had to be made to the structure. To overcome both these problems, Codasyl specified the set as the basic building block from which all complex relationships would grow.

The Set

The only form of structure that can be defined in a Codasyl system is the set, which is illustrated in Figure 23. Each box represents a type of record. A set always has two levels, with an "owner" type of record defined at the top level and one or more "member" types of records defined at the bottom level. When the Data Base Administrator defines the relationships between the records, in coding the Schema, it is done by defining one or more types of sets.

Figure 23

A SET TYPE IS A TWO-LEVEL STRUCTURE



Set Attributes

It would appear that this simple two-level structure can't possibly define all the complex relationships that exist between records in a large and complex business. But the set has some interesting attributes in addition to its structural simplicity. The most interesting attribute, and the one we'll spend our time discussing, is this:

THE DATA BASE ADMINISTRATOR MAY DEFINE EACH SET WITHOUT REGARD TO WHAT KINDS OF SETS HAVE ALREADY BEEN DEFINED, OR WILL BE DEFINED IN THE FUTURE.

The attitude taken on by the Data Base Administrator in a Codasyl installation is to sit back and let each application designer state what relationships will be useful to his application. The Data Base Administrator then defines the sets required, without concern for their legality. There are no illegal structures. The Data Base Administrator in this environment need never draw a map of the entire data base structure, because the entire structure is immaterial. The data base may consist of chains, rings, trees, or networks, and no one really cares. The Data Base Administrator may examine the interaction of the various sets, from time to time, to see if changing the physical mapping of the data base could be improved to provide better performance. Any such changes do not affect the application programs, except to make them run faster.

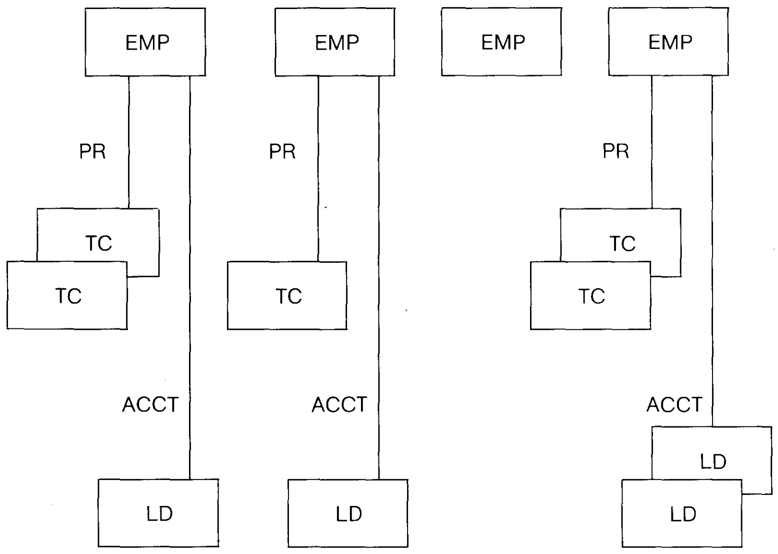
Figure 24 illustrates the relationship between the Schema and the data base defined by that Schema. In this example, the data base consists of three types of records, defined as the employee (EMP) record, the timecard (TC) record, and the labor distribution (LD) record. Subordinate to each record description are the definitions of the data items contained in each record, which aren't shown here in detail. In the data base, we see four EMP records, five TC records, and four LD records.

Figure 24

SCHEMA

```
RECORDS:            SETS:
  01 EMP               PR:OWNER = EMP
  01 TC                  MEMBER = TC
  01 LD              ACCT:OWNER = EMP
                         MEMBER = LD
```

DATA BASE



Now look at the right half of the Schema, where we define the sets of interest to the applications. We have a payroll (PR) set, defined to have the EMP record as its owner and the TC record as its member. We have also defined an ACCT set, consisting of the same EMP record as the owner and the LD record as the member. In the data base, these sets are both illustrated by the lines connecting these records and labelled with the name of the set. Notice that in the data base, there are several occurrences of the PR set and several occurrences of the ACCT set. In each occurrence, the EMP record is connected to its associated TC and LD records.

We can now write a payroll application that accesses EMP records by employee number. Once the application has the EMP record, logic need not be coded to access the proper TC records. The application simply requests the Data Base Management system to retrieve the proper TC records. The system, using its internal knowledge of the data base structure, retrieves the proper associated TC records. It is no longer required that we complicate our application programs with the logic involved to find the right records.

WHAT IT MEANS TO THE USER

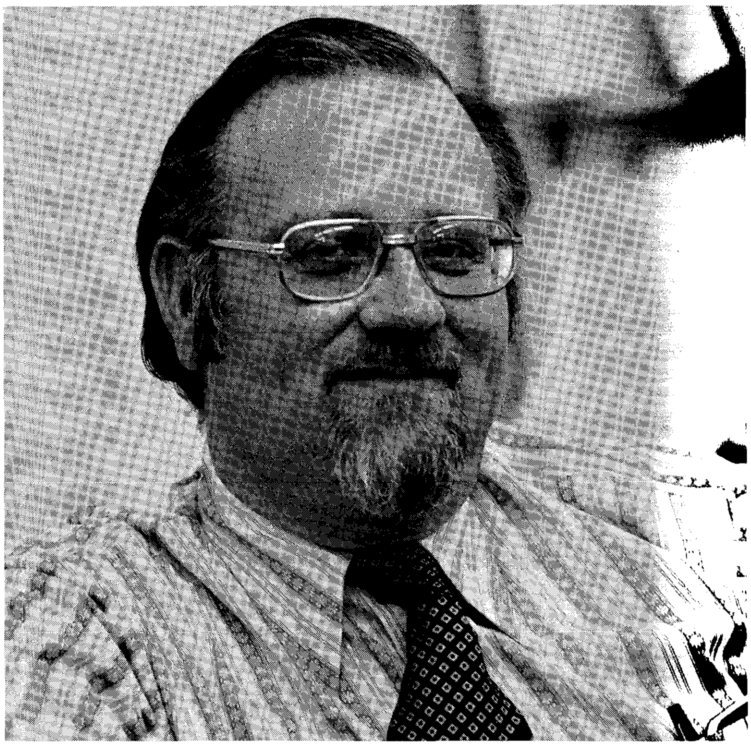
The growing industry movement toward the implementation of Codasyl data base systems is a repetition of the industry's movement to Codasyl's earlier product, COBOL. Today, COBOL is used for almost all business applications, and the most recent information indicates that 83% of all programs running today are written in COBOL. The U.S. Federal Government requires that any computer sold to the government for business processing must be accompanied by a standard COBOL compiler. Representatives within the government have also indicated that the same requirement will exist with data base systems, as soon as the vendors have had a reasonable time to implement such systems. The marketplace already shows such systems.

What do Codasyl systems mean to the user? Well, to begin with, they are the de facto standard for data base definition and processing. The user who already recognizes the advantages of common facilities, such as reduced training and ability to move programs to other computers, will recognize that the same benefits exist for data base systems. Another major benefit to the user is that there are no structural constraints in Codasyl systems. The Data Base Administrator can intersperse chains, rings, trees, and networks in the same data base, and can regularly change the structure without invalidating dozens of programs.

The technical facilities of the Codasyl systems are too detailed and numerous for discussion here, but you could look at it this way: the specifications were written by dozens of people from dozens of companies who spent twelve years in researching the problems and finding solutions that would serve the needs of all kinds of users in all kinds of applications in all kinds of businesses. We believe that those people did a pretty good job, and that the future of data base systems lies with those specifications. To top it off, ANSI has begun to examine standardizing the Codasyl data base work.

Most user problems can be solved by good File Management systems; some require the use of a true Data Base Management system. Mismatching the user's problem with a data solution is always a mistake. If the user needs data base, his job will be much more costly and difficult if he is forced to use a File Management system. If the problem can be solved with File Management, however, the use of a Data Base Management system will introduce needless overhead and complexity to the problem.

We believe it is in the customers' best interests to have a full range of products from which to choose the best tool to solve a given problem. We provide that range of products with our File Management system, RMS-11K, that equals or exceeds the capabilities of any File Management system on the market, and with our DBMS-11, which is in conformance with the Codasyl specifications for Data Base Management systems and provides the most comprehensive data base facilities in the entire computer industry, bar none.



About The Author

Mike O'Connell has been associated with CODASYL since 1968. His work there began with helping to define the COBOL communications specifications. After several years of general work in improving the CODASYL COBOL language specifications, he began to specialize in data base work. Under contract to Bell Labs, he developed the proposal to CODASYL for the COBOL Sub-schema language and the COBOL data base manipulation facility, which CODASYL ultimately approved. He was the first chairman of the CODASYL Data Base Language Task Group, and is now the chairman of the CODASYL Data Description Language Committee (DDLC).

He has been with Digital Equipment since 1973, and is presently Manager of Product Planning for Large Systems for the Business Products Group.