A model for a generalized data access method*

by RANDALL L. FRANK
The University of Utah
Salt Lake City, Utah
and
KOICHI YAMAGUCHI
The University of Michigan
Ann Arbor, Michigan

INTRODUCTION

The proliferation of the methods used in modern operating systems to access data is apparent. In the operating system OS/360 alone there exist multiple ways of accessing sequential data (BSAM, QSAM, BPAM), indexed sequential data (BISAM, QISAM) and directly addressable data (BDAM). If one adds to this the variations of the above used by various systems that run under the control of OS/360, such as IBM's data base management system IMS/360, there exists almost a countless number of ways to access and store data within a computer system.

This proliferation of data access methods has left persons involved in the design, implementation or evaluation of new or existing access methods with an almost hopeless task. In order to better understand the nature of data access methods, a model has been developed in whose terms existing and proposed data access methods can be stated. This paper will discuss the components of such a generalized data access method and give examples of its use in modeling existing data access methods. Parts of the model to be presented are at a further state of development than others, and, therefore, at times a formal discussion of parts of the model may be replaced by a more informal functional description.

A discussion of existing data access methods

Since the need for such a generalized data access method is in part based on the fact that we lack a common basis with which to discuss data access methods, it is difficult to offer very many global remarks about existing data access methods. Nevertheless, several general observations about data access methods will be offered.

Computing hardware, in general, tends to be rather inhospitable to the average users of a computing system. In particular, input/output hardware and its associated protocol tend at times even to make system's programmers cringe. Thus, one of the more important functions of data access methods is to provide a cleaner user interface between the user and the underlying hardware.

While the hardware on a given computing system remains fairly constant, user requirements and needs change constantly. Thus, another important function of data access methods is to take a fixed hardware/input-output environment and provide a virtual environment which more nearly matches the environment desired for a particular application. However, any time an operating system designer attempts to provide a fixed number of such virtual applications oriented environments, users will come along whose requirements are not met by the access methods provided. Thus, many operating systems provide for "escape" access methods which allow the user (and, indeed, require him) to interact with the intricacies of the input/output hardware, providing him with only basic support. An example of such a facility is the EXCP (Execute Channel Program) access method in OS/360.

Clearly, it would be desirable to provide for tailored accessing methods without forcing a user to spend undue amounts of time learning the details of a particular hardware system. One purpose of the generalized data access method model to be presented here is to allow for such data access methods tailoring at a fairly high level. Thus, the parameters to such a model must be in the form of user-oriented, and where possible, machine independent languages. The user should be able to state his access method requirements in languages natural to his application, rather than those convenient to a particular machine.

Goals of this research

Several of the goals of this research effort have been alluded to above. Four primary goals to be set forth for this

---

* Research sponsored in part by the Air Force Office of Scientific Research, Air Force Systems Command, USAF, under Grant No. AFOSR-72-2219. The United States Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon.
effort are:

(1) the ability to model existing and proposed access methods.
(2) the ability to compare different data access methods in a common terminology.
(3) to provide a facility with which one can easily create and try out new access methods tailored to a particular application.
(4) to enable one to read and write files which are foreign to a particular hardware/software system.

The first two goals above are highly interrelated, and amount to providing a common language with which to discuss data access methods. Attempts to perform comparative analysis of file organizations and access methods (see, for example, Severance) have found that before any analysis could be performed, a common model had to be developed. While models have been developed to facilitate the analysis of specialized aspects of data access methods, for example, performance aspects, little work has been done previously in developing a model with applicability to a wide range of problems.

As was mentioned earlier, when the access methods provided by a vendor are not ideal for a particular application, one is forced to either make do with an existing access method, or design one from the ground up, and specify it usually at the machine level. It is interesting to note that even computer vendors have found their own provided access methods unsuitable for some of the applications software they provide. For example, IBM, in designing IMS/360, decided not to use the standard access methods of OS/360, but instead to develop new ones particular to IMS, such as HISAM. Even programs such as the IBM Mathematical Programming System/360 have ignored operating system provided access methods in favor of specialized accessing techniques using the above mentioned EXCP facility. While these above two major software efforts could afford the time and expense needed to develop new access methods from scratch, the normal user, finding himself in a situation where existing access methods lead to sub-optimal results, often cannot afford the time or money, or lacks the expertise, to design and program a new access method. Thus, a high-level access method design facility is called for.

The fourth goal, which was the original motivation behind this work, has to do with the translation of foreign data files. Foreign in this sense implies software and/or hardware incompatibilities. In the course of research in general file translation at The University of Michigan, it became apparent that two of the functional components of a generalized data translator were readers and writers.

Since the basic premise of The University of Michigan research was that data translation could be performed using a very high level stored data description language, it follows that these general readers and writers would likewise have to be driven by statements in this high level language. While the basic design of such general readers and writers is often simpler than that of a completely general data access method, much of the underlying model is the same. For example, in general data translation the question of the data base update function can be ignored while in a general data access method this is a necessary function.

A review of related work

There are several research and/or developmental efforts that are related to our endeavors. However, none of them appear to fully comply with all four of our goals as stated in the previous section. Some of the more interesting research includes work on formulating a model for data description by IBM and by the Stored-Data Definition and Translation Task Group of CODASYL, a development of a description-driven data translator at The University of Michigan.

Data independent architecture model (DIAM)

A group at IBM research in San Jose has formulated a model for describing data structures, as well as data accessing properties of a wide variety of data base systems. The model, called the Data Independent Architecture Model (DIAM), uses a limited number of basic concepts so that it can be useful not only in describing and comparing various aspects of existing data base systems, but also in designing and implementing new data accessing methods.

The DIAM structure is based upon the four hierarchically related models in describing data, starting at the very abstract view of data in terms of entities and relationships among them, down to the encoding on physical devices. The most interesting features of DIAM are seen in the second and the third levels in the structure, the String (Access Path) Model, and the Encoding Model. In the String Model, various access path structures independent of their encoding techniques and physical implementations are described by varying parameter values of the three types of access path (strings) and by specifying simple operations on these strings.

The stored representation of each element of the access path structure thus defined is depicted in the Encoding Model. All the relevant information on encoding each element of the access path structure is collected in a fundamental unit, called the Basic Encoding Unit (BEU), so that variations in encoding can be nicely expressed in terms of the BEU structure and the process of data factoring applied on BEUs. The combination of these three concepts, access path structures, BEUs, and data factoring, provides DIAM with a general capability of characterizing and evaluating existing and proposed data base systems.

Stored-data definition and translation task group (SDDTUG)

In 1970 a task group of the CODASYL Systems Committee, the Stored-Data Definition and Translation Task Group (SDDTUG), was formed to formulate a stored-data definition language, which is a means of explicitly and formally defining data as they appear on physical media, in
order to facilitate data exchange among possibly dissimilar systems. The initial results of the Task Group were reported at the 1970 SIGFIDET (then SICFIDET) Workshop.11

Two major criteria have been set forth by the Task Group in developing a model for stored-data description. One is the generality of the model so that it cannot only be applied to existing structures, but also easily be extended to new structures. The other is the separation of logical aspects of stored-data from its physical representation in storage.

In accordance with this guide line, the Task Group introduced a model at the 1972 SIGFIDET Workshop.12 The model is based on the stored-data description models developed by Smith10 and by Taylor.13 The model currently being revised by taking the concepts presented in DIAM into consideration consists of two independent streams of data description. One stream describes logical aspects of stored-data, and the other describes the structure of storage media on which the data is represented. These two descriptions are combined into one in order to complete the total description of stored-data.

Data translation project

The Data Translation Project at The University of Michigan has expressed as its goal the development of a general methodology for transferring data from one environment to another. With this research goal in mind, the project undertook, as its first year's work, the task of developing a prototype data translator, in order to demonstrate the feasibility of the technical approach (i.e., a language-driven data translator as stated in an earlier section), and to gain a better understanding of the problems associated with data translation.1

The prototype translator developed translates files generated by NIPS, which operates on an IBM/360 computer, into input files for WWDMs, which operates on a Honeywell H-6000 computer.2 Although in a rather restricted manner, the major components of the translator are driven by two forms of high level languages: stored-data definition language and translation definition language. The stored-data definition language is used to describe input and output files and the translation definition language is used to make an association between input and output file descriptions. After a successful accomplishment of the first year's task, the project is currently undertaking the task of developing a more general data translator by relaxing the allowable classes of input and output files, as well as by making the translator more language-driven.

OVERVIEW OF THE GENERALIZED DATA ACCESS METHOD MODEL

This section will present and motivate the overall structure of the generalized data access method model. Successive sections will discuss in depth the various components of the model.

The elements of the model to be presented shortly can be divided into two categories, namely language elements and algorithms. The generality of the model is derived from the fact that the algorithms are driven by high-level language descriptions of the desired access method to be modeled. Thus, the algorithms do not as such constitute a particular access method, but merely allow for the simulation of any access method which can be described in the language elements of the model.

A block diagram of the model will be useful for future discussion, and is shown below:

![Diagram of a model for a generalized data access method]

Figure 1—A model for a generalized data access method

The main substantial difference between the above model and those of existing specialized access methods is the addition of the access method description language. However, it is this additional element which enables the remaining components to act in a generalized fashion.

The operating system component is placed in the model based upon the realization that such a generalized access method would not be used in a vacuum. However, the operating system here is meant to represent just the minimal environment necessary to support the generalized access method in a multi-user environment. The operating system assumed here excludes components currently thought of as file systems and specialized access methods. These latter components would be built on top of the generalized access method. Thus, the entire model could be considered to be within the realm of a comprehensive operating system.

The "verbs" of the data manipulation language are, of necessity, somewhat dependent on the access method described. For example, access methods which provide for no direct addressing could not (easily) support a data manipulation command requesting that the system deliver a record whose key has a certain value.

However, this is not meant to imply that there is a fixed data manipulation language. The verbs in the data manipulation language are only restricted by the descriptive capabilities of the access method description language and the algorithms which use that language. Thus, one could define, for an inherently sequential access method, a "find direct" verb. However, the semantics for such a verb in a sequential access method might necessitate a linear search of the data base or file.

Certain data manipulation language constructs, such as "next" relative to a current position in a file, are easily implementable for most access methods. However, its imple-
mentation may have very different specifications for different structures on secondary storage. The definition of "next" would be quite different for a sequentially organized file as opposed to a list structured file. Thus, while the external semantics of "next" are access method invariant, its implementation may vary widely.

Thus, the data manipulation language is only access method independent to the extent that one specifies not only an access method in the access method description language, but also an implementation for all desired data manipulation commands. Precisely which data manipulation commands should be specified for all access methods will not be discussed here, although, through example, certain common data manipulation commands will be presented.

Limitations of the initial research and design effort

In order to limit the scope of the work, several assumptions were made as to the desired level of generality. These limitations have to do primarily with the class of machines usable with a single implementation of the generalized accessing algorithms.

The primary interest here is in studying differing access methods and not in the detailed input/output hardware of computer systems. Therefore, it was decided that description of the access methods would stop at the level above that of the actual input/output instructions of a machine. This restriction limits the usability of an implementation of the generalized accessing algorithms to a given machine architecture. This also implies that the generalized accessing algorithms are responsible for, via "hard code," the translation of access requests in the data manipulation/access method description languages into actual I/O commands for a particular machine.

While many aspects of I/O device architecture are described in the languages of the model, it is assumed that the accessing algorithms know, in a pre-defined sense, how to interpret such descriptions for a given machine architecture. For example, one section of the access method description language discusses the development of secondary storage device addresses. However, the accessing algorithms would be responsible for converting a data request to/from such an address into actual hardware I/O commands. These might include a seek request to a particular cylinder, followed by a transfer request.

An obvious extension of this work would be to provide languages for the description of the structure of the I/O instructions of a machine. Further levels of description might include generalized handling of I/O interrupts and error conditions. While this area is being investigated, it will not be considered further in this paper.

THE ACCESS METHOD DESCRIPTION LANGUAGE

Central to the concept of the generalized data access method is that of high level languages for describing access methods. The access method description language is itself made up of several sub-languages. The two most important ones, to be discussed here, are the device description language, and the accessing language.

Since the accessing language is based upon primitives defined in the device language, the device description language will be presented first.

The device description language

The device description language is based on an assumption about the current nature of secondary storage devices. Namely, that such devices can be described in hierarchical terms, with each element of the hierarchy using only elements of lower levels as its components.

For example, one possible hierarchical description of a disk drive consists of storage cells at the lowest level, i.e., positions where bits or characters of data can be placed. At higher levels, records can be viewed as being composed of storage cells, tracks of records, cylinders of tracks, and a disk volume as being composed of cylinders.

It should be noted that this decomposition of a secondary storage device into hierarchical levels is not unique for a particular device, but merely possible. For example, in the above device description, tracks on a single disk surface could be considered to make up the next higher level of the hierarchy. These disk surfaces would then comprise a disk volume.

Thus, the device description language must not contain any assumptions as to the hierarchical breakdown of devices, and must allow for any realistic description. As such, there are no "pre-defined" names for components of devices. Any semantics associated with a device component (such as "record") are given by the user when the accessing of the device is described in the accessing language. The accessing language description is done in terms of the device components specified in the device description language.

Rather than present detailed syntactic and semantic descriptions of statements in the language, a functional description of the language will be presented, followed by possible examples of its use. The language is at a state of development where detailed syntactic decisions have not been completely specified. The examples, therefore, are meant to merely give the flavor of the language.

Elements of the device description language

The two major functions of the device description language are to define the aforementioned device hierarchy, and to describe the scheme used in addressing the device.

As an example device, the IBM 2314 disk drive will be used. In order to minimize the complexity of the example, many low level details associated with such a device will be ignored. Included in this category are problems such as damaged track demarcation and alternate track selection. In a working system based on this model, problems like this would necessarily have to be described and handled.
The device hierarchy is described from the "bottom up" so as to allow for components of one level to be used in the description of higher levels. Keywords in language are underlined. The syntax here is merely illustrative, and should not be viewed as a formal specification.

The first section of the language defines the device hierarchy, and gives attributes of each level of the hierarchy.

DEVICE IBM-2314 DESCRIPTION;
STORAGE CELL IS BYTE;
BYTE CONTAINS 8 BITS;
COMPONENT IS RECORD;
RECORD COMPONENTS ARE BYTE;
LENGTH OF RECORD IS MIN 1 BYTE MAX 7294 BYTE;
COMPONENT IS TRACK;
TRACK COMPONENTS ARE RECORD;
LENGTH OF TRACK IS MIN 1 RECORD MAX 71 RECORD AND MAX 7294 BYTE;
COMPONENT IS CYLINDER;
CYLINDER COMPONENTS ARE TRACK;
LENGTH OF CYLINDER IS 20 RECORD;
COMPONENT IS IBM-2314;
IBM-2314 COMPONENTS ARE
CYLINDER;
LENGTH OF IBM-2314 IS 200 CYLINDER;

In the above description, an IBM-2314 has been described as containing a basic storage cell of an 8 bit byte. The words BYTE, RECORD, TRACK, etc., have no special meaning in the language, but were chosen to make the description more readable. RECORD is defined as comprised of a minimum of 1 BYTE and a maximum of 7294 BYTE. TRACK is composed of RECORD, subject to two length constraints, namely a maximum of 71 RECORD and 7294 BYTE. This dual constraint is needed since the unit RECORD has a variable length, and therefore specifying only the number of RECORD per TRACK is not sufficient.

The remaining elements of the description should be obvious from the above description. The primary purpose of this information is to drive the device dependent accessor, which is described in a later section.

In the second part of the device description language, the addressing structure of the device is defined.

ADDRESS ENCODING IS DISK-ADDR;
WIDTH IS 40 BITS;
REPRESENTATION IS HEX;
ADDRESS FIELDS ARE CYLINDER-NUM, TRACK-NUM, RECORD-NUM;
CYLINDER-NUM FIELD IS 16 BITS;
POSITION IN DISK-ADDR IS 0 THRU 15;
RANGE IS 0 THRU 199;
REPRESENTS CYLINDER WITHIN IBM-2314;
TRACK-NUM FIELD IS 16 BITS;
POSITION IN DISK-ADDR IS 16 THRU 31;
RANGE IS 0 THRU 19;
REPRESENTS TRACK WITHIN CYLINDER;
RECORD-NUM FIELD IS 8 BITS;
POSITION IN DISK-ADDR IS 32 THRU 39;
RANGE IS 1 THRU 71;
REPRESENTS RECORD WITHIN TRACK;

Here a 40 bit disk address named DISK-ADDR is described. An address is divided into fields, and the structure of each field, and what it represents, is specified. A permissible range of values for each field is given in the RANGE statement. Where the particular field is located within the address, is specified in the POSITION statement. The REPRESENTS statement relates the address field to the device hierarchy.

The accessing language

It is envisioned that a single specification of a device would be sufficient for any access method implemented on that device. The above description of the IBM-2314 could support many very different access methods for files on such a device.

The accessing language, on the other hand, describes how data on a device is to be accessed. This is done primarily by defining accessing primitives, which would then be invoked by a user to access the data. Once again, there are no predefined primitives. Any primitives which are needed must be defined in the accessing language in terms of access paths to a device.

Elements of the accessing language

Once again, this language will be presented in terms of an example. The example presented here borders on the trivial, but hopefully will at least give a flavor of the language, and how it is used to describe an accessing technique.

The access method presented here is for a sequential read of an existing file on the previously defined IBM-2314. In this example only the most basic functions are presented in order to minimize the complexity of the example.

ACCESS METHOD SEQUENTIAL-READ;
DEVICE IS IBM-2314;
STATUS INFORMATION IS CURRENT-ADDRESS;
FORMAT CURRENT-ADDRESS IS DISK-ADDR;
SPECIAL CONDITIONS ARE END-OF-TRACK, END-OF-CYLINDER, END-OF-FILE;
CONDITION END-OF-CYLINDER;
RECOGNITION ADDRESS FIELD TRACK-NUM IN CURRENT-ADDRESS GREATER THAN 19
ACTION SET ADDRESS FIELD CYLINDER-NUM IN CURRENT-ADDRESS TO 0;
INCREMENT ADDRESS FIELD CYLINDER-NUM IN CURRENT-ADDRESS BY 1;
CONDITION END-OF-TRACK;
RECOGNITION HARDWARE;
ACTION SET ADDRESS HARDWARE;
RECOGNITION HARDWARE;
ACTION RETURN ENDFILE STATUS;

In this first section of the language the format of certain status information maintained by the accessing algorithms is described. In this case status information named CURRENT-ADDRESS is being kept. The format of this status information is the same as the previously defined DISK-ADDR.

The remainder of this section defines certain exceptional conditions that can occur during the processing of the data. The criteria for the recognition of the special condition are specified, as well as the action that is to be taken when the special condition occurs. In some cases the recognition of the special condition is assumed to be performed by the hardware, while in other cases the accessing algorithms themselves must recognize the special conditions.

The final part of the accessing language defines the accessing primitives, and the semantics associated with each. Here only very simple semantics have been specified.

ACCESS PRIMITIVES ARE OPEN, READ, BACKSPACE, SKIP;
ACCESS PRIMITIVE OPEN;
PARAMETERS ARE FILE-ADDRESS;
FORMAT FILE-ADDRESS IS DISK-ADDR;
PROCEDURE SET CURRENT-ADDRESS TO FILE-ADDRESS;
ACCESS PRIMITIVE READ;
PARAMETERS ARE BUFFER-ADDRESS;
FORMAT BUFFER-ADDRESS IS PRIMARY MEMORY ADDRESS;
LOGICAL ACCESS UNIT IS RECORD;
PROCEDURE TRANSFER FROM CURRENT-ADDRESS INTO BUFFER-ADDRESS;
INCREMENT ADDRESS FIELD RECORD-NUM IN CURRENT-ADDRESS BY 1;
ACCESS PRIMITIVE BACKSPACE;
PROCEDURE DECREMENT ADDRESS FIELD RECORD-NUM IN CURRENT-ADDRESS BY 1;
ACCESS PRIMITIVE SKIP;
PROCEDURE INCREMENT ADDRESS FIELD RECORD-NUM IN CURRENT-ADDRESS BY 1;

In this simple implementation of a sequential access method on disk, a very basic OPEN function is specified. Here it is assumed that the accessing program provides the starting disk address for the file. In a more realistic implementation, the name of a file would be provided, and the specification for OPEN would cause a search of a table of contents on the disk volume for the starting address of the file. Needless to say, this would cause a great deal more complexity in the specification of OPEN.

The only primitive here which actually causes the transfer of information is READ. In the description of READ the unit of information transferred between the accessing algorithms is named in the LOGICAL ACCESS UNIT phrase. In this simple case it is assumed that a complete physical disk RECORD is passed to the accessing program. In more realistic descriptions, this simple correspondence between physical units of transfer and logical transfer units would obviously not hold. The procedure for mapping physical transfer units to logical transfer units would also have to be specified in this case.

The access primitive SKIP and BACKSPACE merely update the status information CURRENT-ADDRESS. Thus, on future invocations of READ the order in which records are returned is altered.

GENERALIZED ACCESSING ALGORITHMS

In order to assure the generality of the data access method being developed, the algorithms used in the method must be invariant, regardless of the data to be accessed. In other words, they must not depend on particular characteristics of the data. Although the development of such totally data independent, but practical, algorithms seems improbable, several approaches can be taken toward that direction.

The approach taken here is to identify a set of primitive operations of which algorithms used in various data access methods are comprised. These operations are primitive in the sense that the semantics of each operation must be unambiguous and simple, and parameters to direct each operation must be limited in number and well defined. Then by reducing the process of data accessing into a series of these primitive operations, and by varying parameter values for these operations, the algorithms which can be used for accessing various classes of data are obtained. In the model, these parameters are specified in the form of a high level language as discussed in a previous section.

As mentioned previously, it is not foreseen that a single set of algorithms will be sufficient to cover all cases of stored-data. However, hopefully pursuit of this approach will result in a limited number of sets of algorithms (or a single set of algorithms with multiple entries, if preferred) which cannot only be applied to any existing data access methods, but also easily be extended to new data access methods. Since the development of such algorithms is still in its infancy, presented here are functional descriptions of components of generalized accessing algorithms.
Components of generalized accessing algorithms

In discussing accessing algorithms, it is important to recognize two types of accessing units. One is a unit to which the data base (or operating) system provides a means of addressability. We term this a physical access unit. The other is a unit of information which is subject to access requests (i.e., a basic communication unit between the user and the system). This we term a logical access unit. This dichotomy is the underlying concept of the model.

As depicted in Figure 2, generalized accessing algorithms consist of three components: a Controller, a Device Dependent Accessor, and a Device Independent Accessor. The functional descriptions of these components will not be particularly novel in the sense that they can readily be seen in existing specialized access methods. However, they are different from those in conventional access methods to the extent that functions of each component are totally driven by the explicit descriptions of access methods.

Controller

The Controller is the part of the system which directs the entire operation of data accessing. Besides the normal function of coordinating linkages between various components of the system, the Controller has three additional functions to carry out: parsing of access requests, selection of an access path, and determination of access request fulfillment.

The Controller determines the validity of an access request expressed in a data manipulation language. Legitimacy of the request may include access security considerations. This process is performed by consulting structure mapping description tables which are obtained from the accessing description of the language. Once the validity of the access request is established, the request is transformed into a more convenient form for further processing. The complexity of this process depends on the allowable types of requests in the system.

Given the parsed form of the access request, the Controller then selects an appropriate (logical) access path to fulfill the request. It should be noted that there may exist more than one access path which can be qualified to satisfy the request. Therefore, the process of access path selection includes the determination of all access paths qualified, followed by the selection of the “best” one among them. This selection process is also driven by structure mapping description tables.

When a logical access unit is identified by the Device Independent Accessor, the Controller determines if it satisfies the access request. If so, the control is returned to the user. If not, the Controller requests the Device Independent Accessor to provide the “next” logical access unit on the access path selected. This request may, in turn, invoke the Device Dependent Accessor.

Device dependent accessor

Stored-data is an organized collection of physical access units. Its organization usually depends strongly upon the characteristics of a device on which the data is located. It is highly desirable that components of the accessing algorithms function as independently of these salient characteristics of devices and file organizations as possible. The function of this component is to remove such device and organization dependent characteristics from stored-data, so that the Device Independent Accessor can function independent of a particular device used and the addressing mechanism employed in the stored-data.

By examining the way that the access path selected by the Controller is encoded on the device, the Device Dependent Accessor transmits a physical access unit from the secondary storage into the main memory. This process is driven by a set of tables, called device and access description tables, whose contents are derived primarily from the device description. These tables are rather independent of the tables which drive the Device Independent Accessor. Thus, by driving the two components of the model using a set of independent tables, the model provides a very powerful and flexible means of accessing the data. By simply changing tables to be used by each component, the model permits the user to access the data, with different access and mapping strategies, which may reside on various storage devices.

Another function which must be carried out by this component is a problem resulting from the possible differences in elementary data representations between the system which created the data and the system under which the algorithms operate. This problem is a very common one when the
accessing of foreign data files is considered. These architectural differences can also be resolved in the same technical approach (i.e. the description driven approach). However, it is felt that inclusion of problems arising from architectural differences is beyond the scope of our immediate research.

Device independent accessor

Given a physical access unit in a buffer, this component performs two major functions. The first function is to extract and identify a logical access unit from the physical access unit. Here the term “extract” is used to mean to separate one unit from another and the term “identify” to mean to recognize the name of the access unit. It should be noted that the order between the extraction and the identification of a unit is not definite. For example, the unit may first be extracted and then identified. On the other hand, the identification of the unit may be necessary to extract the unit.

The boundaries of physical access units may or may not correspond to the boundaries of logical access units within a particular data base architecture. In other words, the relationships between these two types of access units are in general m:n. Therefore, in order to extract a logical access unit, multiple physical access units may be required. This is accomplished by invoking the Device Dependent Accessor repeatedly.

The second function is, given a logical access unit as a result of the first function, to decompose it into its constituents. This decomposition process is carried out by the use of storage templates constructed from structure and mapping description tables. A storage template is a collection of named elements which schematically represent a logical access unit. One template is created for each type of logical access unit. It should be noted that a complete storage template for certain logical access units may not be constructed solely from the information in structure and mapping description tables. The construction of such a template may have to be deferred until the decomposition process of the access unit is initiated. In other words, some storage templates may be constructed dynamically.

CONCLUSIONS

It has been impossible in this paper to discuss in detail many of the really interesting questions one confronts when discussing access methods. Out of necessary space limitations, only the basic components of the model for generalized data access have been presented.

The question of generalized data access is by no means a solved one. The model presented here is intended as a research tool and not a production model. In the current languages the level of procedurality, particularly in describing access primitives, is much higher than desired. A great deal of emphasis in the design of the languages was placed on the factoring of information common to multiple accessing methods in common places, so that it would not have to be repeated. However, more work must be done in this area.

It is also felt that there are too many “pre-defined” keywords in the languages, which limit the generality of the model. Lower level descriptions of these current keywords are needed.

However, the current model gives us an important base to build upon. An implementation of the current model is planned, and it is expected that this will give us greater insight into additional problems. Even with the basic current model we now have the ability to model many different access methods in a common language. This alone has given us valuable insight into the nature of access methods, as well as having shown us weaknesses in the current model.

REFERENCES

4. IBM Corporation, OS/360 Data Management Services, form #GC26-3746.
5. ———, OS/360 Data Management for Systems Programmers, form #GC28-6550.
7. ———, 3314 Direct Access Storage Facility and 3314 Auxiliary Storage Control, form #A26-3599.