THE HODS DATA STORAGE MANAGEMENT SYSTEM

Allen Reiter, T.F. Chang, Alan Clute

and Doug Cox


Most of the work described in this document was done at the Lockheed Missiles and Space Company at Palo Alto, Calif. supported under the Lockheed Independent Research Program. The senior author wishes to acknowledge the support of the National Science Foundation (Postdoctoral Fellowship Number 49073) which made possible the writing of this paper.
THE HODS DATA STORAGE MANAGEMENT SYSTEM

Abstract: A generalized data management system which is useful for both on-line and batch processing on small and medium size computers is described. The concept of self-descriptive data allows powerful primitives to be defined and supported at the systems programs level. The standard data structure chosen is that of a tree. Use is made of a peripheral random-access device in presenting the user with a single-level virtual memory with a symbolic address space. Data groups and items may be of completely variable length. Novel features of HODS include the automatic dynamic segmentation of data into pages or records, economy in accesses to peripheral storage by avoiding the use of pointers, and the close cooperation between the monitor and the data management system for core storage management.
The HODS Data Storage Management System

Allen Reiter (#), T.F. Chang ($) , Alan Clute ($) , Doug Cox ($) 

I. Introduction

The management of items of information available to a computer for processing—
their physical representation, the allocation of memory for their storage, their
movement among different levels of a computer memory, the definition of global
processes over the items which facilitate the writing of application programs—
is one of the central problems in computing today. The problem arises in
essentially the same form in entirely different areas of computer applications,
for example in business data processing, in the implementation of time sharing
systems for on-line programming, in string-manipulative processes associated
with work in pattern recognition and artificial intelligence, etc.

This paper describes an approach to a solution of this problem. Although
the specific context for the described implementation was an on-line information
management system, its usefulness (it is hoped) extends to other areas as well.

The concepts presented are machine-independent and can be implemented on any
modern computer equipped with a large secondary random-access storage device.
The presence of flexible addressing hardware such as base registers simplifies
the usage of the described system, but is not essential to the implementation.
The orientation is chiefly toward the small-to-medium computer which is not
equipped with paging hardware or automatic address-mapping schemes; however

*) The work described in this paper was largely performed at the Lockheed
Missiles and Space Company Palo Alto Research Laboratory under the Lockheed
Independent Research Program. This paper was written while the senior author
was a National Science Foundation Postdoctoral Fellow, Fellowship Number 49073.
#) Technion Computer Science Department, Haifa, Israel

$) Information Sciences Laboratory, Lockheed Research Laboratory, Palo Alto,
California.
there are many similarities between HODS and the MULTICS file system [1], which has been implemented on a large (CE 645) computer equipped with hardware-assisted paging. There is a time-shared-use influence evident in two facets: the machinery which allows sharing of data among simultaneous independent users while guarding against simultaneous update attempts, and the absence of parallelism in the sequential execution of a process. (*)

(*) Compare the IBM system/360 OS "queued access methods"

(The philosophy is to let the time sharing monitor overlap input-output operations with processing by queueing requests from different simultaneous processes, instead of trying for parallelism within a single process). At the same time, the page-turning algorithms associated with the unlimited virtual memory addressing are equally useful to a single user whose memory requirements exceed the fast memory available, and the directory-building algorithms lead to a data base which is organized for efficient searches for the batch as well as the time-sharing user.

HODS (for Hierarchically Organized Data Structures) was developed as part of the ALERT project [3] at the Information Sciences Laboratory of Lockheed, and first implemented on the IBM 360 Model 40 computer (*)

(*) The system described below differs in minor points from the implemented version.

equipped with IBM 2311 discs and a 2321 Data Cell. In its first version, HODS operates under a special time-sharing monitor [4] SYSEX which itself operates in a partition of a DOS-partitioned computer and manages the partition resources among several tasks.
HODS relies on SYSEX for the dynamic core storage management. Conversely, the basic SYSEX concept relies on data segmented along the lines described below, and on having a systems program serve as an interface between a user and his data. Projects currently under way at the Technion will

a) modify SYSEX (and hence HODS) to run under OS;

b) implement HODS to be usable as a stand-alone package by programs written in a number of IBM-supported source languages and run under OS.

The present paper does not concern itself with the very difficult problem of managing a multi-level store with more than two levels of storage; see for example[5]. No algorithms are incorporated for moving (for instance) data between the data cell and the discs. HODS is therefore strictly a two-level system. Whether or not HODS can be extended to a multi-level system is an open question.

Some of the ideas incorporated into HODS, in particular the concept of relocating and splitting records, originated with the ALLSTAR[6] data management system developed at Lockheed for the LACONIQ[7] time-sharing monitor.

A number of operational systems today utilize the concept of self-descriptive data; the reader is referred to [8], [9] and [10] for bibliographies and comparative evaluations. The novel features of HODS include the automatic directory creation which results from the splitting algorithms and the close cooperation between the paging strategy in the operating system and the data management system, unusual on a small computer.
Sections II and III of the paper are concerned with discussing and justifying the concepts of self-descriptive data and the choice of a tree as the standard data structure. Section IV talks about the physical representation of the data, while Section V presents HODS from the point of view of a user. Section VI discusses in detail the automatic splitting algorithms and the resulting directory structure; the basic idea is not critically dependent on the rest of HODS and can be adapted for automatic directory generation with other data structures. Section VII contains some conclusions.

II. Self-Descriptive Data

The major concept in HODS is that of self-descriptive data. Instead of representing data in storage as an unstructured string of bits and assigning semantic meaning to the strings within the processing application programs, HODS adopts a representation which, with reference to the global HODS subroutines, allows the meaning of the data to be stored along with the data. By taking away the user's "right" to choose data representations no flexibility is sacrificed; at worst, some processing time and storage overhead is incurred in the encoding-decoding process which might have been avoided by an application-oriented representation. The gains however are so great (and apparently, are not yet so widely recognized) that it is worth while to enumerate some of them.
1. Ease of use. The savings in applications programming effort will more than offset the (possible) extra operating cost.

2. Improved resource utilization. The adoption of global resource management strategies supported by appropriate allocation algorithms can lead to great savings; this is usually difficult to achieve if attempted in the context of specific applications. For example, the use of variable length records (without sacrifice in retrieval time due to many accesses to peripheral storage) leads to very compact data storage and more than offsets the storage overhead due to the required self-descriptions.

3. No unnecessary duplication in programs. Since the data storage and retrieval functions are centrally located, user programs can be very compact. This is especially important in a time sharing system on a small computer, since the HODS routines are all reentrant and can be shared by several users.

4. Sharing a single copy of a datum among several users. This feature of HODS is of special importance in a time-sharing environment because the pyramid-like structure of the data base makes simultaneous usage of higher-level directories a very common occurrence.
5. Implementability of global data security and data validity procedures, and of procedures for recovery failures. Although these features were implemented in HODS version 1 in a rudimentary form, the adoption of uniform representations left open the possibility of extending these features later without requiring the rewriting of existing programs.

6. Implementability of general-purpose data-structure-dependent subroutines. For example, the ALERT system contains a DISPLAY primitive, which causes a specified subset of the data base to be displayed on the current user-display device. The calling program does not have to specify the data format(s) or the characteristics of the output device. If the resulting DISPLAY would occupy more than one screen on a CRT-display device, the necessary pagination* is accomplished automatically by the system in an application-independent manner. The addition of new display devices or of new data types could be handled very easily by additions to the DISPLAY routine, without requiring modification to the application programs.
7. Transferability of programs. By interposing system programs between the physical data representation and the processing application programs, there arises a possibility of transferring data from one computer configuration to a second one with an incompatible hardware data representation without having to rewrite the application programs. Of course, to avoid instruction incompatibility, the programs would have to be written in a higher-level language.

There are many parallels between general data manipulation systems such as HODS and higher-level language processors, and arguments similar to the ones advanced above have been used to justify the (possible) inefficiencies introduced by compilers. (Actually, general data manipulation systems necessarily operate in a mode more closely related to that of interpreters.)

HODS is designed to minimize accesses to peripheral storage during the retrieval of information. Because in many HODS applications the computer tends to be input/output bound, CPU time was in general considered expendable. An important goal was to achieve compact data storage, partly because compaction can somewhat reduce the number of accesses to peripheral storage required, but also because the cost of random-access on-line storage tends to be very high and such storage is a crucial resource in data-base applications.
III. Logical structure of the data.

All of the data within the system are represented logically as part of a tree. The user (here and in the sequel user denotes an application process) addresses data by specifying the name of the desired sub-node relative to the (current) parent node. Thus, the user sees a virtual single-level memory, with a hierarchical symbolic address space.

The address of an item* may be DATABASE1.STUDENTFILE. A347.AGE

* In addition to using tree-theoretic terminology, we sometimes adopt the terminology given in [10]. Thus, we shall say group for any node in the tree (possibly empty) which consists of data other than atomic; we shall say item for a leaf or atom of the tree.

(see Figure 1) which may denote the field containing the age of student no. A347 in the "student file" in the part of the data base known as DATABASE 1.

No restrictions exist on branch names. Instead of specifying a name, a sequence number may also be specified for addressing the sub-group; thus, if "studentfile" is the second group in DATABASE1, the address DATABASE1.2.A347.AGE is equivalent to the one above.

In the actual use of the system, the user does not supply the full address all at once, but rather repeatedly issues the LOCATE command specifying the desired branch. This facilitates the writing of tree-traversal algorithms, and in general makes possible "context" addressing determined by the (current) parent node.

Only one access path exists to any data group or item: the one specified by the sequence of branch identifiers.
The user may not specify a physical address (somehow known to him), or use any other shortcuts in getting to his data. Since some applications could conceivably run more efficiently had they been allowed other access paths to data, some justification of the choice of a tree as the standard data structure is in order.

The primary reason for using trees lies in the use of variable-length data. In HODS, all items or groups of data may change in size as a function of time. Essentially no restriction exists on the potential size of a datum. When this fact is combined with the HODS objective of efficient utilization of space within a random-access memory, a consequence is the dynamic relocatability of all data; that is, the system may change the location of a datum at its own convenience. Now, it is impractical to "remember" along with a datum all of the places where it is referenced. Hence, if multiple access paths to a datum are to be allowed, a consequence is "indirect addressing", i.e. system of directories which have to be consulted in order to translate the reference address into a current physical address. Since these directories are typically very voluminous, such indirect addressing would require one or more accesses to peripheral memory for each access to a datum, thus slowing down retrieval of information by a factor of two or more. Instead, HODS uses a direct addressing scheme, made possible by the uniqueness of access paths. Each time a datum is relocated, its current address is changed in its parent (the parent is always known to the system without being explicitly pointed to by the child; the system "remembers" access paths during execution).
Thus, by depriving the user of (possible) shortcuts which might be effective in the context of a specific application, a generally efficient access scheme can be implemented.

There are a number of other benefits which come about because of the choice of a tree as the basic data structure. Here is a brief list:

i) The recognition problem, i.e. recognizing that different calls for data are in reality an instance of a multiple use of the same datum, is trivially solved: as the system knows only path names, two different things are really different.

ii) Data security problems can be solved very easily and without much overhead, since any branch may have guards associated with it. Small subsets of data can be protected as easily as large ones. Hierarchies of protection levels can be easily handled. Passwords or other caveats can be stored along with the data independently of the procedures which operate on them.

iii) Semaphores to prevent simultaneous update attempts can also be implemented easily. There is no need to lock unduly large subsets of data, since small subtrees can be locked as easily as large ones. (Note that in more conventional systems, e.g. OS data management services [2], a file is the smallest unit of data over which the system retains access control and which can be locked or unlocked).

These are the reasons why, at the lowest level of system support, the tree is used. Other data structures can be superimposed at higher levels.
For example, the information retrieval applications of HODS utilize the concept of inverted files. The "pointer" to a data group is however precisely the path name.

From the user's point of view, therefore, HODS can be regarded as a computer whose basic architecture involves not a linear string of fixed-length cells but a tree with variable-length terminal leafs. This HODS computer can do anything the more usual computers can do, but can do some things much more conveniently.

IV. The Physical Data Representation

Most systems which support non-linear structures represent the parent-child relationship (B is a member of A) by a pointer from A to B (sometimes a two-way pointer). When a hierarchy of memories are involved in the data storage this frequently means that paging activity is involved in going from A to B; i.e. an access to random-access storage has to be made for almost every change in the nesting level of the data*.

* Some systems, most notably GE's Integrated Data Store[11], attempt to cut down on accesses by allowing the user to direct the placing of his data in the peripheral store. IDS however is a fixed-field-length system, and data are not subject to dynamic relocation.

In order to avoid excessive accesses, HODS attempts to represent logical containment relations by actual physical set containment. Thus, if the node B is a child of the node A, then the physical representation of A in storage as a linear string will include as a subset the linear string which represents B.
This is possible because of the choice of a tree as the basic structure, so that B can be the son of at most one A. In other words, whenever A is brought into core, B is necessarily also brought into core at the same time.

We say "attempts"; clearly this strategy would break down for all but the smallest of databases, since the physical length of linear strings is limited severely by core considerations. Long linear strings must be broken into pieces called records or pages (we will use the two terms interchangeably) when a string is about to become too long, automatic splitting algorithms (described in section VI) break it into two records. The splitting involves replacing the actual occurrence of a child node in the parent by a pointer to the child. Occasionally the child is broken into a number of pieces, each of which is represented (together with an appropriate key) by a pointer in the parent; retrieval then proceeds directly from the parent to the appropriate piece without having to scan the other pieces of the child.

The actual occurrences of indirectness (replacement by pointers) are however quite small; in going from the root node of the tree to the remotest leaf, it is estimated that at most four or five pointers are encountered even for very large databases ($10^9$ characters). The actual number of accesses to peripheral storage will typically be fewer than the number of pointers encountered, since there is a high probability of finding one or more of the required pages already present in core.

The physical representation of the typical node (field) is shown in Figure 2. The structure is recursive, so that all imbedded nodes have the same structure, while this node in turn is imbedded in a similar higher-level node.
For the sake of consistent terminology, the entire structure depicted in Figure 2 is known as a field, while the same structure without the leading key field is known as the node.

The first part of the field is its key; this is the branch name (in its parent) leading to this node, and defines the (logical) address. Keys may be of variable length, up to 255 characters. No duplicate keys may appear within a node, and all nodes have their subfields ordered on the keys. It is possible to omit the key (i.e. to have a key length of zero); in that case, all other subfields of the parent must not have any keys, and the ordering of the subfields is chronological, i.e. new subfields are added at the end. (This is the HODS analogue of a sequentially organized file.)

Unless the parent node is of type indirect (see below), the next part of the field is a descriptor known as type. On the 360, this is normally one byte long, but may have length up to 128 bytes. (The first bit, if zero, means that the next seven bits are the length of the type part; if one, means that the length is precisely 1.) The first byte is reserved for HODS use, but the rest provide an open-ended means for representing immediate data within the node in addition to the descendants. Thus, access information, various output information, etc., can be represented there. One of the bits in type is reserved to denote that this node is indirect, i.e. that all of the descendant fields have been replaced by pointers. This is discussed more fully in Section VI. Another bit denotes that this is terminal atomic data (leaf, or item) and has no further structure from the point of view of HODS.
If this node is not atomic, it necessarily contains zero or more subfields (zero subfields is not treated as a special case in HODS). The next part of the node contains the subfield count.

In order to be able to search very efficiently among subfields of varying lengths, the addresses of the subfields are represented tabularly; this is an alternative to specifying the lengths of the subfields. Since the addresses themselves are fixed-length information, searches on sequence number are implemented by direct fetches of the subfield address, while searches on key utilize a binary search technique. The next part of the node contains the starting addresses of each subfield; these are given as relative displacements from the start of the node.

With all of this structural information out of the way, the actual substantive data content of the node (i.e. its subfields) now follow. The length of each subfield is not represented explicitly within the data base, but is computed from the relative addresses when the node is accessed.

We have described the typical subfield which is itself imbedded in some node. Such fields are known as secondaries. When a node is split off and set up in a separate record, it appears as a primary. A physical record contains exactly one primary node. The structure of physical records, their core buffers, and of the primary is shown in Figure 3. Note that a primary node is not preceded by a key part; the path name appears in the parent along with the pointer to this physical record, so that it is necessarily known by the time this record is accessed.
(If we should somehow get this record by itself, we have no way of knowing its relationship to the rest of the data base without searching the latter in its entirety).

Figure 3 shows three "slack" areas. These come about because of quantification of both peripheral and core storage, as follows.

A storage quantum for peripheral storage is determined by the topography of the storage device; for the 2311 disc, the quanta were full-track, half-track, quarter-track, and one-eighth-track (respectively 3625, 1739, 829 and 383 bytes). Whenever storage for a new record was required, the smallest quantum of sufficient size was allocated. As the record grew, it would be relocated to the next larger size, until it exceeded the largest quantum; thereupon, it was split.

The reason for two slack areas can be stated very succinctly: the block-move (MVC) instruction on the 360 works very well in shifting a storage area towards lower core addresses, but runs the risk of propagating data characters when shifting toward higher addresses. Execution of the second category of shifts can thus be quite time-consuming. For this reason, HODS attempts whenever possible to have right-to-left shifts. Slack 1 is set up automatically as "room for growth" when a new record is set up; Slack 2 occurs as a result of deletions. This, incidentally, is one of the few places in HODS where an attempt was made to save processing time at the expense of static storage space (two extra bytes, to represent the relative starting address of the primary node, are required).

The slack in the core buffer arises because the core storage allocation quanta (discussed in [4]) are not (indeed, cannot be) exact multiples of disc quantum sizes.
The resulting slack was useful, since records in active use were allowed to grow beyond their physical record boundaries; disc relocation was invoked only when the record was no longer in active use and had to be rewritten to the disc. (The core relocation was of course invoked whenever required; splitting was invoked either because the largest disk or the largest core quanta were exceeded).

At first glance, it might appear that the physical representation involves a great deal of overhead in storage space. Here, of course, one must ask "Compared to what?" Clearly, fixed-length records can themselves be represented more economically; the unanswerable question is "To what extent could the logical information have been more economically represented by variable-length representations?" Frequently, the systems analysts who set up the data specifications are governed by constraints imposed by antiquated processing methods. It is the authors' conviction that the HODS data representations are not only more convenient but in fact lead to great savings in peripheral storage space due to the increased flexibility in space utilization.

V. Using HODS

This section describes how HODS looks to a user who in writing a program in the 360 assembly language. Although ALERT contains interfaces which allow HODS to be used in two other modes - by a terminal user via direct on-line commands, and by procedures in a simplified macro language called the "Procedure Execution Language" - the assembly-language level is the basic level for which HODS was designed. The other modes do not differ significantly in usage, and in any case utilize the same system primitives.
In the course of using HODS, a process has "windows" into the data base. Each such data window constitutes a single instance of use of data by a process. A process may at any given moment in its execution have any number of such windows active, or it may have none.

A data window can be thought of as a variable which takes on as values pointers to data*

* Actually, the value of a data window is a pointer to a pointer.

Values are assigned by the execution of a LOCATE command, which moves the window around. All HODS commands include the data window specification as one of the parameters.

Instead of pointing directly at data, a data window points at a control element known as a logical access descriptor, or LAD. Each LAD is associated with a single group in the data base which happens to be in current use. A data group may be in multiple use at a given moment (by a single process, or by several independent processes); it nevertheless has but a single LAD describing it. A data group is considered to be in use if either a) it has a data window currently using it, or b) if it is the ancestor of a group which is in use, i.e. if it can be ascended to. In addition to various control information, the LAD contains a pointer to the data group and the length of this data group. See figures 4 and 5.

A node which is currently in use need not necessarily be in core. It is however guaranteed to be in core if a data window is currently pointing at it (indirectly, through the LAD). In such cases, it is said to be in active use.
The totality of all LADS at a given moment in time is known as a LADDER. The LADDER is itself a (core-resident) tree, which corresponds to that subset of the data base tree which is currently in use.

Once a data group is no longer in use, its associated LAD disappears.

The following HODS commands are currently implemented.

(1) CREATEL datawindow. This creates a new data variable by allocating dynamically storage for the control block, and initializing its value. Additional optional include specification of keys for accessing for reading, writing, or execution; specifying an existing data window whose current values and keys are to be copied into the new one; and specifying that the data entity to be manipulated is a temporary storage which is dedicated to this process and which disappears when the process ends.*

* The CREATE command is in some ways not unlike the conventional OPEN command present in many other systems but in HODS. Note however that CREATE constitutes a new instance of use of data in general, and not of any specific file, except that a window created for temporary data is limited to that part of the data base.

(2) ERASE datawindow. This operation is the inverse of CREATE. The datawindow control element is released to free storage. Any LADS formerly in use by this data window are reduced in user count by one, and (possibly) disappear.

(3) LOCATE datawindow, mode=KEY= (or SEQ#=). This command assigns a new value to the datawindow relative to its former value. mode specifies the tree operation desired. Currently recognized values for mode are DOWN, NEXT, BACK, and UP. For DOWN, the current node is searched on the given key or sequence number, and (if successfully found) the corresponding LAD is set up and the data window is made to point to it.
For NEXT, the parent of the current node is searched; beginning with the current field, the next element in sequence (if any) is then produced. BACK operates in similar fashion in the opposite direction. UP simply ascends one node.

Note that all of the required paging is accomplished automatically. If the target node is not currently in core, it is brought in (something else may be paged out, including the formerly current node). For the actual implementation of the core storage allocation, see [4]. If the LOCATE operation is not successful, the value of the data window is unchanged.

(4) INSERT datawindow, mode, KEY# (or SEQ# =). This command creates a new node and inserts it into the data base in a place determined by the current value of the datawindow and by the mode parameter. Mode may have the values DOWN or NEXT which have the same meaning as for LOCATE.*

* NEXT with a key specification is interpreted to mean that the insertion is to be made at the same level as the current. The actual position of the new node relative to the current one is determined by the ordering of the keys.

If a key is specified, the node together with the specified key are inserted into the place determined by the lexicographic ordering of keys. If SEQ# is specified, the node into which the insertion is being made may not have any keyed subfields. In creating non-atomic nodes, an empty "shell" with a count of zero is set up. For atomic nodes, the user must specify the additional parameter ATOMIC= (datalength, dataaddress).
The user must also specify (for all nodes) what to do if a node with this key
or sequence number already exists; the choices are REPLACE or NOTIFY.

Additional optional parameters include provisions for storing access
keys (reading, writing, execution) along with the data; these are given
explicitly and need not be the same as in the current data window. A TYPE
parameter, which is a completely open-ended immediate descriptor of up to
127 bytes in length, may also be specified and stored with the data (the first
three bytes of this will be produced in the LAD whenever the node is in use.)
Note that the data segmentation which may be required as a result of this
command is accomplished automatically by the system. The data window is set to
point to the LAD which is associated with the newly-created node. The paging
which may be required is completely transparent to the user. Note that some of
the former data addresses used by this process for other nodes may no longer be
valid, as data get moved around. Each LAD is however guaranteed to have the
latest current addresses. It is up to the user to reload his addressing
registers from the LADS. Semaphores built into HODS guarantee that the "inserting"
process is the only active user of this data page; hence the addresses used by
one process can never be invalidated by an INSERT operation performed by a
different process.

(5) DELETE datawindow. The current node is deleted from the data base. If the
node is non-terminal, all filial nodes are also deleted. If physical records
are no longer used, they are made available to the system for reallocation.
Again note that other nodes may change location; again the LADS contain the
latest information.
(6) CONCATENATE datawindow, newstring, length. For atomic data, it is quite awkward to extend the stored data by concatenating a string at the end if the INSERT command is to be used. For this reason, the CONCATENATE operation is provided. The data field pointed to by the current data window is extended by the string stored at newstring and of the indicated length.

Again note the automatic segmentation, and the fact that the data may be moved.

(7) CHANGE datawindow, DATA or TYPE, displacement, newdata, length. The CHANGE operation is intended for those modifications to the current node which affect the content but not the structural form of the data. Unlike the other operations, CHANGE does not do something for the user which he could not do himself more conveniently; rather, is required because (to avoid errors or conflicts) the user may not modify the data base directly. Either the data portion of an atomic node or the "type" portion of any node, may be changed, in place. This command does not cause paging-segmentation activity, and cannot cause data to change location.

For all operations, in addition to returning results in the datawindow, the system returns the result as a condition code, in one of three conditions:

CC=0 Operation successful; datawindow contains result.

CC=1 Operation valid but not successful; datawindow value is unchanged. This might occur for example during an unsuccessful search for a non-existent key in LOCATE.
CC=2 Operation invalid; data window remains unchanged. This might occur for example if a LOCATE DOWN is attempted from an atomic node.

Access key violations cause an immediate abortion of the task.

VI. The HODS data segmentation algorithms.

At the heart of the HODS design philosophy is the contention that the partitioning of the data base into small manageable units (records) each of which is the subject of a single input-output transaction between the central processor and random-access peripheral storage does not need to be done by the applications programmer but can be accomplished automatically by the system. This section discusses how this is handled in HODS. The concept as presented is also applicable to the automatic creation of directories in systems which do not otherwise utilize the concept of self-descriptive data.

A directory can be generally defined as a (long) list of ordered pairs \((I_n, P_n)\) where \(I_n\) is called an identifier and \(P_n\) a pointer. (The semantics of these terms need not concern us here.) It has the property that each entry is quite short relative to the total length of the list. The actual HODS algorithms are a generalization of the directory-building scheme which is first described. Directory in HODS is not used in the conventional sense of translating indirect reference to physical addresses, but is used as in MULTICS [1] in the sense of a non-terminal node. The justification for this terminology is that the physical representation in HODS of a large collection of keyed entries is in fact similar to the proposed directory scheme.
Many different techniques for representing and searching directories have of course been designed; by far the most common technique involves using the internal representation of the identifier to compute a (not necessarily unique) storage location. A common version of this method is known as hash-coding and is widely used in implementing compilers.

It is sometimes desirable to store the directory entries in some predetermined order, for example in the collating sequence order of the identifiers. This is especially desirable in the context of on-line information retrieval systems, if a terminal user is to be able to approach the system with identifiers known to him only approximately, or if a "browsing mode" is to be implemented, or if all occurrences within certain bounds are required without searching the entire collection. Since it is impossible to devise (without additional information as to the exact distribution of the actual identifiers in the space of all possible identifiers) an order-preserving hash coding scheme, other methods have to be tried. The scheme described below has the properties

a) it is order-preserving;

b) it is completely automatic and requires no prior knowledge whatever as to the expected distribution of identifiers;

c) it leads to rapid searches;

d) very compact storage utilization is achieved.

The scheme involves building up many levels of directories. A single record (prime index) is the master index into the next level; this secondary level consists of records each of which is an index to the tertiary level, etc.
Eventually, at the bottom level, the desired identifier-pointer pair is reached. The records at the higher levels also contain identifier-pointer pairs, which are in fact in the same identical format as the bottom-level entries; the pointer is a pointer to a next-level record, while the identifier is the same as the greatest identifier in the record pointed to (assuming identifiers are in ascending order). See Figure 6. It is important that all levels have the same data format and be processable by the same routines, except that the last level must be recognizable as such. No distinction between the other levels needs to be made.

At the very beginning, when the directory is about to be created, it consists of one empty record*.

* It is in practice more convenient to set up the initial record with a single entry consisting of an identifier larger than any admissible ones, and with a dummy pointer.

As entries are presented, they are inserted in their proper place into the (one) record, until no more room remains. The one record is then split into two more-or-less-equal parts. These two records are both at the secondary level. A new primary-level record is set up, which now contains pointers to the two secondary pieces, together with the keys which are the largest in the respective pieces. See Figure 7. As more identifiers are added, they are placed into the appropriate bottom-level records. When one of these becomes too big, it is split into two pieces, and another entry is inserted into the primary level.
This way, the secondary level keeps growing in records. The process of adding an entry to the primary record is completely identical to the insertions into the bottom level.

When the primary fills up, it is again split and dropped to the secondary level (just as it was before), thereby making the (former) secondary-level records tertiary. The process now continues in a completely recursive fashion; as many levels as are required will be built up. Because the higher-level records are not treated by different machinery that the lower-level ones, the programming is quite simple.

If only a single record size is used in allocating disc storage, the expected packing density for randomly-distributed identifier arrivals is about 75%; this drops to about 50% if the identifiers happen to arrive in their proper order. The situation can be remedied by allocating records in quanta of which the smallest quantum is slightly bigger than $\frac{1}{2}$ of the largest, and by relocating records to the smallest sufficient record size until they exceed the largest quantum. Excellent space utilization can then be achieved, especially if the storage pool used for storing the directory is also available for allocation for other purposes.

Note that the directory is balanced in that the same number of accesses are required to get to any bottom-level identifier. If the largest record size holds (on the average) $N$ identifier-pointer pairs, and if the total number of entries in the directory is $M$, then approximately $[\log_N(M)]$ accesses are required; i.e. this is an $N$-ary search technique with respect to disc accesses. For searching through an individual record in core, a conventional
binary search technique can be used.

The software machinery required for the implementation of this scheme is minimal; a disc storage allocator and a record-splitting routine are all that is required. If the directory is subject to many deletion, an off-line utility program which periodically rebuilds the directory may be useful; note however that the utility may improve space utilization but will not normally significantly affect the retrieval time.

So far, the discussion has not involved the HODS implementation. In HODS the situation is somewhat complicated by the fact that data groups can have extreme variability in length. Although identifiers (i.e. keys) are all in the length range 0-255, the "pointers" may vary between being actual 3-byte pointers to being imbedded subgroups almost as long as the parent group.

In general, the following two categories of splitting have to be distinguished:

1. A record consists of a large number of very small things. Solution: split the record into two pieces which are at the same level of indirectness, and add a pointer to wherever the original pointer to the record was present. (This is the same as the directory splitting).

2. A record contains, among other things, a very large subgroup. Set up the subgroup as a separate record, and replace its occurrence in the original record by a pointer. The rest of the original record is unaffected. (To facilitate later splitting of the subgroup just removed, its occurrence in the original record is actually replaced by a list of identifier-pointer pairs; this list initially contains but a single pointer, with the identifier greater than the largest admissible identifier value. On the 360, this was taken as X'FFFFFFFF'.)
Note that the distinction is a heuristic one, involving, as it does, concepts of "large number" and "large or small things". The definitions which follow are an attempt to translate these concepts into algorithms. The algorithms always work, in that the data segmentation function is always performed. "Bad" heuristics in this case lead to sub-optimum space utilization and increased retrieval time. Not enough experience is yet available with large data bases to determine whether these particular heuristics are good or bad.

Recall (from section IV) that a primary node is the outermost node contained in a physical record, while a secondary node is one imbedded in another node.

**Definition:** A primary node is big if its length exceeds that containable in the largest size.

**Definition:** A secondary node is big if all the following hold:

1) It is contained in a big node;
2) Its field length is at least half as long as its parent node;
3) Its field length exceeds a certain minimum.*

* This minimum must be greater than the longest possible "list of pointers with one entry" which has to replace a too big secondary. If key lengths of up to 255 bytes are allowed, this minimum is precisely 268 bytes.

**Definition:** (Maximality Condition) A node is said to be too big if both of the following hold:

1) It is big;
2) It does not contain any big node.
From these definitions, there easily follows the proposition "If a primary node exceeds the maximum record size, it contains exactly one too big node".

The splitting algorithms involve finding the too big node. They then proceed as follows.

If a primary node is too big:

1. If it is atomic, an error condition exists, and the INSERT or CONCATENATE operation is not allowed. Atomic data may not normally exceed certain bounds on their length (about 3600 bytes for the ALERT implementation).*

   * Provisions are included for specifying during the INSERT operation that the datum is "long"; in that case, the datum is immediately set up in a separate record, and a pointer to it is introduced into the parent. "Long" data are subject to different space restrictions, related to available core; typically, 10,000 bytes is a reasonable maximum. In the applications, the only need for "long" data arose in connection with programs in the program library; without paging hardware, it is impractical to segment machine code automatically.

2. If it is not atomic, this is a case of the directory splitting discussed earlier. The primary node necessarily consists of at least two subnodes; it is split along subnode boundaries into two pieces of more or less equal size. The parent of the too big node (located in a different physical record) contains a list of pointers; insert another pointer (to the new piece) into it. If the subfields of the too big node are keyed, attach to the new pointer the key of the last subgroup in the first half; otherwise insert the new pointer without a key (See Figure 8).
3. If a secondary node is too big, remove it to another physical record (thereby making it into a primary). Replace its occurrence in the parent by a list of pointers, which initially contains only one entry. (This list of pointers is itself a so-called indirect node.) See Figure 9. If the removed node contains keyed subnodes, add the key X'FFFFFFF' to the pointer entry (for further splitting later); otherwise, add a key of length zero.

If the effect of deletions is ignored, the reader can convince himself that the product of a split is always two records of "reasonable" size, and that spurious accesses to peripheral storage do not occur. Deletions of course can change the picture considerably; if, after removing a too big secondary and replacing it by a pointer, it should happen that the original physical record shrinks in size and can be recombined with its removed child, the pointer does indeed represent a spurious access. Instead of attempting garbage collection *

* "Return of the prodigal" seems to be more appropriate terminology for this context.

of this sort on-line, it is left to an off-line utility; it seems unlikely that performance can be seriously affected even after a long sequence of insertion-deletion operations. (As mentioned earlier, the effect of such operations on space utilizations is more pronounced).

It can happen that even after the split one of the pieces is still too big. For example, consider a primary node A which occupies the maximum allowable space.
A secondary node B is inserted, A becomes big, and B is found to be too big; B is then set up as a separate record, and the appropriate indirect node is inserted into A in the stead of B. However, A is still big because of the indirect node B added to it; the splitting process is then repeated. It should be clear by now that the piece to be split off need have no connection with the piece just inserted.

The splitting routines were the biggest single subsystem within HODS. Because they are invoked relatively infrequently, they were not part of the core-resident nucleus of programs, but (along with the disc storage management routines) were brought in from the disc library whenever required.

VII. Conclusions

The work described here represents a genre of software which is still rather new, although the need for it has been recognized for at least ten years. HODS must be evaluated on three different planes; the ease and convenience of use (and hence its power), the implications of the systems concepts on the operating efficiency of the user programs and other resource utilization questions, and also on the feasibility of the implementation, i.e. was it worth it.

Of the three, the middle one is the hardest to evaluate, in part because HODS has yet to be used in conjunction with a "live" large data base. Throughout this paper, the authors have advanced the argument that what is sacrificed in efficiency in the context of a specific application-oriented task is regained in the increased flexibility in the logical organization of the data system and in the fact that changes and additions are easy to implement.
Thus, instead of comparing HODS efficiency to what a "clean" applications-oriented data base program might do, one should really compare a HODS-implemented application to a real-life system after the latter has undergone several generations of changes. These arguments are necessarily subjective, and it is unlikely that an objective comparative evaluation is possible.

Since there are a number of different system resources which have to be considered (CPU time, response time or total throughput time, accesses to peripheral storage, static on-line storage space in random access memory), any meaningful discussion of efficiency must be conducted within a framework which describes the tradeoffs to be made is resolving conflicts in resource utilization. Thus a system inefficient from one point of view may be extremely efficient when a different stand is taken on the relative costs or values of the various resources. The basic HODS philosophy is as follows.

The single most vital resource in the system is access to peripheral storage. It has become widely recognized that for time shared systems in particular a bottleneck develops in trying to access random access memory (see for example [12], [13]); it follows that a system which minimizes this bottleneck is somehow more "efficient".

Static on-line storage is also considered to be very crucial, although less so than the dynamic access to it. This fact, well known to managers of computer installations, is for some reason frequently ignored by system software designers in deciding on tradeoffs. Especially in the context of time shared systems, the power of a computer for on-line use is directly related to the
number of different data bases which are accessible from a terminal, which in turn is frequently limited by the number of disc drives. (For batch processing, the equivalent consideration is the amount of time wasted in mounting and dismounting volumes; a consideration perhaps less important but nevertheless significant.)

CPU time was considered to be the least important resource. One reason for this is the presumed frequent bottleneck at the peripheral device, which implies the presence of CPU idle time. A more subtle consideration was the desire to have a system upgradable in a more-or-less continuous fashion. If better performance is desired in a system which is compute-bound, today's technology allows one to buy processors with ever faster arithmetic and core memory, without requiring changes to one's basic configuration. If, however, the limiting factor in system performance is the rate of access to the mass storage devices (determined by the combined effects of arm movements, rotational delays, and data transfer rates), it is difficult if not impossible to improve performance by hardware changes; hardware costs rise much more steeply than system performance. Because of this, it was felt desirable to shift as much of the load as possible on the CPU.

No direct effort was made to achieve good response or throughput times. For reason discussed in [4] it was felt that the way to achieve a good response time was to optimize system resource utilization; this means that on the whole good response times will be achieved, but in specific cases under more or less unusual conditions a job may take an "inordinately" long time to complete.
Thus, the question of the efficiency of HODS and similar systems is necessarily rather inconclusive; you either believe in it or you don't. More concrete answers do exist on the other two points.

HODS is extremely easy to use, given the fact that it can be used today only from within an assembly-language program (no vehicle for using it in a higher-level-language program exists). Fairly sophisticated user-interface programs (a query language [14], a text editor, and a number of other components of ALERT) have been implemented using HODS in very short time. Some of this feedback has resulted in modifications in the usage of HODS (as mentioned previously, the implemented version differs slightly from the one described here). The power and convenience come not only from the actual HODS-implemented commands described in Section V, but also from general systems support functions (e.g. the DISPLAY primitive) which fall outside the realm of HODS proper but which could not have been written without the concepts of uniformly structured and self-descriptive data imposed by HODS.

Is HODS useful outside the context of information retrieval applications? Again, not enough experience is yet available. As of the time of this writing, a project currently under way at Lockheed is using HODS in solving a problem in combinatorial analysis. This is done not because the tree data structure is particularly suitable for the solution of the problem, but simply because the problem exceeds the limits of available core, and HODS offers the simplest and quickest means for getting the problem into the machine. Perhaps the feedback from this application will lead to further changes in how HODS looks to the user.
Finally, the question of implementation. The initial version of HODS, in spite of several mid-stream changes and an unfortunate choice of standard internal representations (different from the one shown in Fig. 2) which led to much unnecessary complexity in the programs, took less than 1 1/2 man-years to implement. The version described here can be implemented by a reasonably experienced programmer in about six months. Even with various "bells and whistles" in the areas of automatic recovery and data security, at most one man-year of effort is required.

Given that so little effort (relative to the advantages gained) is required for implementation, it is remarkable that so few generalized data storage management systems seem to have been developed to date. Those that have been attempted have apparently been attempted on a very large scale, and had been beset by implementation problems (e.g. MULTICS). A quote from [9]: "As with other systems of comparable complexity, implementation is not proceeding as fast as the designers may have hoped." is substituted by the present authors not so much to protect the identity of the system under discussion, as because the sentence does indeed fit many systems attempted to date. The effort involved in a small-scale undertaking is however not prohibitive, and should in the authors' opinion be undertaken more often than it is.

Perhaps the situation today in the area of data storage systems is not too different from the comparable situation with respect to compilers (again) in the middle and late 1950's. Then, writing a compiler was indeed a major undertaking involving an effort of five man-years or more. Today, one can find many a
university computer science course in which students have to write compilers as term projects; also, using syntax-directed compilation and related techniques, several commercial compilers have been automatically generated by computers. It is to be hoped that the state-of-the-art for data-driven systems will in the future attain a similar level.

Acknowledgments

Many of the ideas present in this paper originated in discussions with Dr. A.J. Nichols, now with the Novar Corporation, Palo Alto, Ca.; the authors are also grateful to Professor J. Feldman of Stanford for some very cogent criticisms which led to major revisions of the concepts. Neither one of them should however be assumed to subscribe to the philosophy embodied in HODS. Finally, thanks are due to Dr. C. E. Duncan, manager of the Information Sciences Laboratory of Lockheed, for his unstinting support of the ALERT project.
References


Figure 1. Logical organization of a typical data base in HODS. Boxes indicate the names of the nodes, not their content. For non-atomic nodes, the content is usually precisely the filial set. Content of atomic nodes is not shown. Not shown also is the lexicographic ordering on name within each node.
<table>
<thead>
<tr>
<th>Key Length (1 byte)</th>
<th>Key (0 to 255 bytes)</th>
<th>Type (1 to 127 bytes)</th>
<th>Count of Subfields N (2 bytes)</th>
<th>Table of relative subfield addresses (2N bytes)</th>
<th>Subfield 1</th>
<th>Subfield 2</th>
<th>...</th>
<th>Subfield N</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data (if type is atomic)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pointer, 3 bytes (if parent is of type indirect)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2. The structure of a HODS field. All subfields have the identical structure.
Figure 3. Physical records, or pages, as they appear on the disc and in core.
<table>
<thead>
<tr>
<th>Byte 0</th>
<th>Bytes 1-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flags</td>
<td>Pointer to LAD</td>
</tr>
<tr>
<td>Byte 4</td>
<td>Bytes 5-7</td>
</tr>
<tr>
<td>Access key</td>
<td>Pointer to Task Control Block for this task</td>
</tr>
<tr>
<td>for reading</td>
<td></td>
</tr>
<tr>
<td>Byte 8</td>
<td>Bytes 9-11</td>
</tr>
<tr>
<td>Access key</td>
<td>Link to other data windows for this LAD</td>
</tr>
<tr>
<td>for writing</td>
<td></td>
</tr>
<tr>
<td>Byte 12</td>
<td>Bytes 13-15</td>
</tr>
<tr>
<td>Access key</td>
<td>Link to other data windows for this task</td>
</tr>
<tr>
<td>for execution</td>
<td></td>
</tr>
</tbody>
</table>

Bytes 16-31 reserved for system use

Figure 4. The Data Window Control Block for the 360. 32 bytes was a standard control-block length for the entire system.
<table>
<thead>
<tr>
<th>Length, in Bytes</th>
<th>MNEMONIC</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>CURFIELD</td>
<td>Address of start of field (if in core)</td>
</tr>
<tr>
<td>3</td>
<td>CURNODE</td>
<td>Address of start of node (if in core)</td>
</tr>
<tr>
<td>3</td>
<td>NODETYPE</td>
<td>TYPE of node (useful if node is not currently in core)</td>
</tr>
<tr>
<td>2</td>
<td>LENGTH</td>
<td>Length of the current node</td>
</tr>
<tr>
<td>1</td>
<td>LEVELCNT</td>
<td>Logical nesting level (relative to root node)</td>
</tr>
<tr>
<td>2</td>
<td>SEQUENCE</td>
<td>Sequence number of this field in parent node</td>
</tr>
<tr>
<td>1</td>
<td>FLAGS</td>
<td>Various status flags</td>
</tr>
<tr>
<td>3</td>
<td>UPLINK</td>
<td>Link to LAD for the parent node</td>
</tr>
<tr>
<td>3</td>
<td>DOWNLINK</td>
<td>Link to the first in-use descendantant LAD</td>
</tr>
<tr>
<td>3</td>
<td>RTLINK</td>
<td>Link to the next in-use LAD of a brother node</td>
</tr>
<tr>
<td>3</td>
<td>WINDLIST</td>
<td>Link to the first data window currently pointing to this LAD</td>
</tr>
<tr>
<td>1</td>
<td>USERCNT</td>
<td>Number of instances of use of this node</td>
</tr>
<tr>
<td>3</td>
<td>DASDKEY</td>
<td>Address of this physical record in random-access storage for primary-field LADS only</td>
</tr>
<tr>
<td>1</td>
<td>ACTUSERS</td>
<td>Number of active users of this record</td>
</tr>
</tbody>
</table>

32 bytes total

**Figure 5. The Logical Access Description (LAD)**
Figure 6. Balanced multi-level directory, shown with three levels.
Figure 7. Very young directory, shown just before and just after first split.
Physical record in which parent is imbedded

### Parent, before split

<table>
<thead>
<tr>
<th>Key:</th>
<th>Type:</th>
<th>Subfield</th>
<th>Subfield</th>
<th>Key:</th>
<th>Pointer</th>
<th>Key:</th>
<th>Pointer</th>
<th>...</th>
<th>Key</th>
<th>Pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>indirect</td>
<td>count: N</td>
<td>Addresses</td>
<td>B1</td>
<td>to R1</td>
<td>B2</td>
<td>to R2</td>
<td></td>
<td>BN</td>
<td>to RN</td>
</tr>
</tbody>
</table>

### Record R2 primary node, before split (exceeds maximum size)

<table>
<thead>
<tr>
<th>Type:</th>
<th>Subfield</th>
<th>Subfield</th>
<th>Key</th>
<th>Subnode</th>
<th>...</th>
<th>Key</th>
<th>Subnode</th>
<th>Key</th>
<th>Subnode</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>count: K</td>
<td>Addresses</td>
<td></td>
<td>B11</td>
<td>11</td>
<td></td>
<td>B15</td>
<td>25</td>
<td>B2</td>
</tr>
</tbody>
</table>

### Parent, after child primary is split

<table>
<thead>
<tr>
<th>Key</th>
<th>Type:</th>
<th>Subfield</th>
<th>Subfield</th>
<th>Key</th>
<th>Pointer</th>
<th>Key</th>
<th>Pointer</th>
<th>...</th>
<th>Key</th>
<th>Pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>indir.</td>
<td>count: N+1</td>
<td>addresses</td>
<td>B1</td>
<td>to B1</td>
<td>B15</td>
<td>to R2'</td>
<td></td>
<td>B2</td>
<td>to R2''</td>
</tr>
</tbody>
</table>

### New primary R2', first half of former R2

<table>
<thead>
<tr>
<th>Type:</th>
<th>Subfield</th>
<th>Subfield</th>
<th>Key</th>
<th>Subnode</th>
<th>...</th>
<th>Key</th>
<th>Subnode</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>count: J</td>
<td>Addresses</td>
<td></td>
<td>B11</td>
<td>11</td>
<td></td>
<td>B15</td>
</tr>
</tbody>
</table>

### New primary R2'', second half of former R2

<table>
<thead>
<tr>
<th>Type:</th>
<th>Subfield</th>
<th>Subfield</th>
<th>...</th>
<th>Key</th>
<th>Subnode</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>count: K-J</td>
<td>addresses</td>
<td>...</td>
<td>B2</td>
<td>2</td>
</tr>
</tbody>
</table>

**Figure 8.** Parent (in a different record) and primary, before and after the primary is split.