November 1975

AN INTERLISP RELATIONAL DATA BASE SYSTEM

by
Stephen Weyl

Artificial Intelligence Center
Technical Note 116

SRI Projects 1031 and 8721

ABSTRACT

This report describes the file system for the experimental large file management support system currently being implemented at SRI. INTERLISP, an interactive, development-oriented computer programming system, has been augmented to support applications requiring large data bases maintained on secondary store. The data base support programs are separated into two levels: an advanced file system and relational data base management procedures.

The file system allows programmers to make full use of the capabilities of on-line random access devices using problem related symbolic primitives rather than page and word numbers. It also performs several useful data storage functions such as data compression, sequencing, and generation of symbols which are unique for a file.

The data management procedures provide a high-level relational data base facility. Programmers define the logical contents of their data base as a set of relations declared as a formal data base definition, or "schema." This schema is then interpreted by general purpose functions which provide for creation, updating, and querying of relations.

We present data base programs first as they appear to the user. Then we briefly discuss their implementation so that performance limitations and future extensibility can be evaluated.

CONTENTS

I INTRODUCTION

This report describes the file system for the experimental large file management support system currently being implemented at SRI. INTERLISP, an interactive, development-oriented computer programming system, has been augmented to support applications requiring large data bases maintained on secondary store. The data base support programs are separated into two levels: an advanced file system and relational data base management procedure.

The file system allows programmers to make full use of the capabilities of on-line random access devices using problem related symbolic primitives rather than page and word numbers. It also performs several useful data storage functions, such as data compression, sequencing, and generation of symbols that are unique for a file.

The data base management procedures provide a high-level relational data base facility. Programmers define the logical contents of their data base as a set of relations declared as a formal data base definition, or "schema." This schema is then interpreted by general purpose functions that provide for creation, updating, and querying of relations.

We will present the data base programs first as they appear to the user. Then we will briefly discuss their implementation so that performance limitations and future extensibility can be evaluated.

II CONCEPTUAL OUTLINE OF THE DATA BASE FACILITY

The data base facility for INTERLISP has been designed in two layers. The purpose for explicit layering is that the first layer appears to be of value to several applications that do not need a complete relational data

base management system. The first layer is a set of procedures that constitutes a "record-structured direct access method" for INTERLISP with variable-sized records, and indexed-sequential retrieval. A comparable software package available elsewhere is IBM's VSAM (Virtual Storage Access Method).

In terms of concepts more familiar to LISP users, a direct file can be thought of as a hash-array on secondary storage that is global to individual programs and processes, with the virtue that it can be made arbitrarily large while taking up only a small, fixed amount of the user's virtual address space.

What a direct file has that a standard INTERLISP (prettyprint) file doesn't have is record-structured storage and retrieval.

A prettyprint file must be created all at once. In order to change any part of the file, or add to it, the whole file must be recreated. This recreation process is somewhat improved by automatic INTERLISP system functions that copy verbatim those pieces of old files that have not been changed without having to regenerate the symbolic external form for their INTERLISP expressions. A direct file, on the other hand, is created one record at a time, and individual records can be deleted or rewritten at any time without modifying the rest of the file. Thus we see that record structure allows for <u>data segmentation</u>. Due to the incremental growth characteristic of record-structured files, the opening and closing of files takes on the additional notation of being a checkpoint. File additions or deletions during a session do not take effect until a file is closed. If a system failure occurs while a file is open, no activity that preceded the opening of the file during the session is affected, and no activity that took place after the file was opened is in effect.

There are two basic access modes for the direct file system. In "direct" access mode, files contain a collection of records that are "randomly" accessible via symbolic keys. This is analogous to accessing values of hash items, where items correspond to keys and values correspond to records. In "sequential" access mode, files are processed sequentially in a manner analogous to (but more powerful than) the operation of MAPHASH. In accordance with the hash-array analogy, direct access mode is typically used when searching a file associatively for a particular fact, while sequential access mode is most appropriate when a set of data is being accessed and processed linearly.

In general, keys and records are arbitrary INTERLISP s-expressions whose elementary constituents are literal atoms, numbers, or strings. Keys are restricted further to be noncircular s-expressions, while records can be circular (i.e., print structure can be infinite due to backwards pointers) and can also be arrays. Procedures are also provided for reading and writing records from arrays and hash-arrays, so that any INTERLISP data structure that can exist in core can exist on a direct file. However, since file data structures inhabit secondary store, discipline in the use of pointers is necessary to obtain good performance.

The second layer of data base management programs is a set of relational data base management procedures. Data is stored in a relational, or tabular data structure, where columns correspond to data "attributes" and rows are recorded associations of attribute values. This data structure has been discussed at length by Codd* and his disciples, and we will avoid further conceptual presentation here.

---

*E.F. Codd, "A Relational Model for Large Shared Data Banks," CACM, Vol. 13, p. 377 (1970).

The data base is designed to have an existence independent of any particular programs. In fact, when we use the term "data base," we imply a set of data structures that are maintained principally on secondary storage media and that can be accessed by any program written at any time relative to the time of data base creation, so long as these programs have proper authorization. Access is achieved via procedures provided by the data base management system. In order to make the data base program-independent, but meaningfully consistent and efficient when viewed by several programs, we must maintain a data base declaration, or schema. The schema contains information that programs need in order to make logically correct and efficient use of data that has been stored by other programs.

When we talk about a particular program's data structure, we imply the local and global variables used by it in the course of its computations. Every programming language contains statements or substatements that serve to declare the types of variables and to indicate what procedures or blocks they are local to. INTERLISP's declarative facilities include the argument list to LAMBDA and NLAMBDA functions which serves to limit the scope of variables, and information determining an atom's type which is associated with every data value, so that data is declared at an "atomic" rather than "generic" level. (Note that no large-scale data bases have atomic level type labeling because of the tremendous inefficiencies that typically result.) Many powerful languages such as PL/1, PASCAL, and SIMULA '67 have quite extensive declarative sublanguages that allow the user to specify in a non-procedural manner the size of data structures, permitted values for data items, hierarchical organization of data structures, and so on. Our data base architecture also uses a declarative, nonprocedural language to define

its principal data structures in such a way that programs can properly access data stored by other programs. We refer to a data base description in this language as a schema.

At the top level, the schema stores the names of data items and their groupings, along with the information needed for data base access functions to locate these data items in the files. The schema also stores detailed information about each data item, including its primitive type, units, statistical type (such as continuous, discrete, normally distributed), and the retrieval files that offer alternate access paths to this item.

The schema also stores a set of procedures, or "demons" that are executed when data are stored or modified in certain relations. The purpose of these demons is to alert interested modules of occurrences significant to them that have been recorded in the data base, and to maintain useful summary information about particular relations.

We will now proceed to a detailed description of the data base package. We present the direct file procedures in Section III, and the relational data base procedures in Section IV. In Section V, we briefly discuss certain implementation details that should help future users to determine the inherent limitations and extensibility of the programs.

III DIRECT FILES: GENERAL FUNCTIONAL DESCRIPTION

A direct file is a TENEX file created and opened by the function DFOPEN and closed after a series of accesses by the functions DFCLOSE or DFCLOSEALL. The reason for opening a file is to allocate core space for processing a window of the complete file. The reason for closing a file is to ensure that linkage in the file and its symbol table are updated from this window to reflect the activity since it was opened.

A direct file contains a collection of data records, each record consisting of an s-expression or array datum that is retrieved by its symbolic address or key in the file. Each distinct key can have only a single record associated with it in a particular file. For example, a direct file might contain a personal telephone directory. In this case the keys of the file might be names of friends, while the data records might contain their phone numbers. A program could look up the phone number of Joe Egghead by requesting that file routines get the datum stored under the key (Egghead Joe), and obtain the answer "314-1592."

A file is opened in one of three protection modes: INPUT, OUTPUT, or BOTH. This guarantees that, until it is closed, only an appropriate subset of the file operations will be allowed. The protection mode of a file can be changed by closing it and reopening it. The only caveat is that files must originally be created with a protection mode of OUTPUT or BOTH.

Basic Direct Access

There are three basic access functions: DFPUT, DFGET, and DFDELETE. DFPUT stores a data record on a specified file under a given symbolic key. If a new record is stored under a used key, it replaces the old record stored there. DFPUT is a legal file operation only if the file has been opened for OUTPUT or BOTH.

For example, Joe's phone number might have been originally stored in the file FRIENDS.PHONE;1 by the following function call:

```
(DFPUT (QUOTE FRIENDS.PHONE;1) (QUOTE (EGGHEAD JOE)) "314-1592")
```

If Joe subsequently changes his phone number due to harassing anonymous phone calls, the following function call might be used to update his phone number:

```
(DFPUT (QUOTE FRIENDS.PHONE;1) (QUOTE (EGGHEAD JOE)) "271-8281")
```

DFGET retrieves a record from a specified file that was stored under a given symbolic key. DFGET is a legal file operation only if the file has been opened for INPUT or BOTH. To retrieve Joe's current number we would evaluate:

```
(DFGET (QUOTE FRIENDS.PHONE;1) (QUOTE (EGGHEAD JOE)))
```

The value of this function would be the record stored under the given key, i.e. "271-8281."

DFDELETE removes permanently the record stored under a key. DFDELETE is a legal file operation only if the file has been opened for OUTPUT or BOTH. If Joe changes his phone number again, and this time refuses to tell the author his new number, he must execute:

```
(DFDELETE (QUOTE FRIENDS.PHONE;1) (QUOTE (EGGHEAD JOE)))
```

Subsequently a DFGET for this key will return the value NIL.

Basic Sequential Access

When DFPUT and DFGET are given key values of NIL, they perform sequential access. The sequence of records in a file is determined by arranging its keys in "key order." Key order is defined as ascending "ALPHORDER" for atomic keys, and a car-first, recursive application of the rule "atom < list" for non-atomic keys. The record returned by a sequential DFGET is the successor of the last record accessed either sequentially or directly with DFGET. When a file is opened, a sequential DFGET will return the lowest order key before the first direct DFGET is performed.

When a sequential DFPUT is executed, it automatically generates a file key sequence number, which is a unique, identifying key (global to

the file's symbol table) for the record. In this case the value of DFPUT is a list, the first element of which is the generated key. The algorithm for assigning file key sequence numbers is explained in Section V. File key sequence numbers are ordinary numeric keys, and programmers who mix sequential and direct output to files can write over records previously written sequentially by doing a DFPUT to a previously assigned sequence number.

Comparison with Hash-Arrays

Note that the basic direct access routines described above have the following INTERLISP analogs:

```
(DFPUT FILE KEY RECORD) ≡ (PUTHASH KEY RECORD FILE)
       (DFGET FILE KEY) = (GETHASH KEY FILE)
    (DFDELETE FILE KEY) ≡ (PUTHASH KEY NIL FILE)
```

There are four crucial differences:

- Hash-arrays are totally core-resident, while direct files exist mostly on secondary storage. The core space taken up by an open direct access file, called the "window" into the file, is just enough to accommodate a segment of the file's symbol table. The size of this window, and consequently the storage overhead for an open file is limited by virtual memory size. The actual file size, and hence the number of associations that can be stored on a direct file, is only limited by the available disk storage in the system.
- Since file records reside on secondary access, the wait time required to execute DFPUT or DFGET can be upwards of three orders of magnitude greater than PUTHASH or GETHASH.

- Files exist independent of processes so that several procedures may use the same files, and the file symbol table will be maintained automatically on the file itself. Only one process will be permitted write access to a file at a time, however.
- Hash-arrays can store associations between any INTERLISP data types by linking a pair of pointers to their physical locations in the core image. This generality can be achieved with direct files by introducing a "pointer" data type (identified by a unique prefix character such as "#") and thereby allowing records to point at other records in a standard way. At present this generalization has not been implemented.

Note that in the hash-table analog we can think of MAPHASH as being a rudimentary sequential access mode for hash-tables, where the key order is random.

Detailed Procedure Descriptions

In the function descriptions below, FILE is a partial or complete TENEX file name from which DFOPEN determines a complete direct file name (including a version number) under which it opens a file. FILENAME is the complete direct file name that is returned as the value of DFOPEN, and is the name that all subsequent routines referencing the file must use. Records and keys can be any INTERLISP s-expression whose atomic elements are literal atoms, numbers, or strings. Keys are further restricted to be noncircular. Records can also be arrays. All the procedures below are lambda functions, so all arguments that are given as literals must be quoted.

DFOPEN[FILE; PROTECTION; NEWFLG; FORMAT]

This function opens FILE with protection mode given by PROTECTION, which may be INPUT, OUTPUT, or BOTH. FILE may be any acceptable TENEX file name.

If PROTECTION is OUTPUT or BOTH and NEWFLG is NEW, a new empty version of the file is created. If NEWFLG is OLD, an old version is updated. If NEWFLG is OLD and no old version of the file exists, DFOPEN will return with value NIL. If PROTECTION is INPUT, the value of NEWFLG will be disregarded; and if no old version of the file exists, DFOPEN will return with value NIL. If NEWFLG is NIL, OLD will be assumed if PROTECTION is INPUT or BOTH, and NEW will be assumed if PROTECTION is OUTPUT.

FORMAT is a list of file format parameters in the following order: (CORERES NKEYS NBYTES). If CORERES < 1 it gives the fraction of the file's index (symbol table) which will be kept core resident during file accessing. If CORERES is an integer greater than 1, it gives the number of index blocks that will be kept core resident. NKEYS is the maximum number of keys per index block, and NBYTES is the maximum number of bytes per index block. NKEYS must be greater than 1, and NBYTES must be greater than 200. A file's format is set when it is first opened, and it remains the same for that version of the file. A new version of the same file with a different format can be obtained using REORG. A file format tailored to a specific application may significantly improve

performance, and a good format can be determined interactively using the function DFADVISE. If FORMAT is NIL when a file is first opened, a reasonable default format will be assumed.

The value of DFOPEN is a complete direct file name.

DFCLOSE[FILENAME]

This function closes the direct file specified by FILENAME, its complete direct file name. If the file is not open, the value of DFCLOSE is NIL; otherwise, its value is the complete direct file name.

DFPUT[FILENAME; KEY; RECORD]

FILENAME is a complete direct file name that must be open for OUTPUT or BOTH. This function stores RECORD under KEY. If a record is already written under this key, the old record will be replaced. If KEY is NIL, a file key sequence number is generated and RECORD is stored under the generated key.

If KEY is non-NIL, the value of DFPUT is RECORD. If key is NIL, its value is a two-element list: the first element is the generated file key sequence number, and the second is RECORD.

DFGET[FILENAME; KEY]

FILENAME is a complete direct file name that must be open for INPUT or BOTH. This function returns as its value the record stored under KEY in the specified file. If no such record exists, it returns NIL.

If KEY is NIL, DFGET reads the next sequential file record after the last record touched by DFGET, DFGETARRAY, DFNEXTKEY, or DFLASTKEY. The value of DFGET after a sequential access is then a two-element list in which the first element is the next key in key order and the second element is the record stored there. When a file is first opened, a sequential read returns the record with lowest order key, and at the end of the file it returns NIL.

DFDELETE[FILENAME; KEY]

FILENAME is a complete direct file name that must be open for OUTPUT or BOTH. DFDELETE deletes a record previously written on the file. It is logically equivalent to (DFPUT FILENAME KEY NIL) except that it actually physically deletes the record, reducing the size of the file after REORG. The value of DFDELETE is NIL.

DFPUTARRAY[FILENAME; KEY; ARRAY; HASHFLG; FIRSTELT; LASTELT; COMPRESSFLG]

This function is like DFPUT, except that for its record it writes out a segment of ARRAY between indices FIRSTELT and LASTELT. If HASHFLG is non-NIL, ARRAY is a hash-array and the complete set of hash-links is written out. Otherwise ARRAY is a LISP-array. If FIRSTELT or LASTELT are NIL or out of bounds, they default to the first and last elements of ARRAY respectively.

One of the features provided for writing out arrays is "data compression." The reason for data compression is that arrays are often sparse, i.e., many of their entries are NIL. Sparse arrays can be written out in a compact print form that reduces the amount of secondary storage space used and trades increased

CPU time for decreased I/O time. If COMPRESSFLG = 0 or NIL, no data compression is performed. If COMPRESSFLG = 1, "NIL compression" is performed, reducing the amount of space required to represent each null array entry to one byte. If COMPRESSFLG = 2, "pile compression" is performed. With pile compression only non-null elements are stored, but each non-null element takes up at least two more bytes than it requires with compression of types 0 or 1.

Generally, compression of type 1 is reasonable for all applications that do not need files to be stored in a readable form (when printed on the teletype or line printer), and compression of type 2 is desirable when more elements of an array are null than non-null. Compression types can be mixed on a file, since they are associated with each independent record. DFPUTARRAY stores the compression type with each record so that DFGETARRAY will always know how to decompress. If DFPUTARRAY is used to output a LISP-array, its value is the number of elements in the array segment written out, or a two-element list containing the generated key and the number of elements for a sequential PUT. If DFPUTARRAY is used to output a hash-array, its value is a LISP pointer to the hash-array, or a two-element list containing the generated key and the pointer for a sequential PUT.

DFGETARRAY[FILENAME; KEY; ARRAY; FIRSTELT; LASTELT]

This function is like DFGET, except that it reads its record directly into ARRAY. If the record contains a hash-array, the hash associations on the record are added to those already in ARRAY. The value of DFGETARRAY is ARRAY, or a two-element list

containing the next key in key order and ARRAY for a sequential GET. If the record was written from a dotted pair of a hash-array pointer and a growth factor, and ARRAY is not a dotted pair, then the DFGETARRAY will return a dotted pair of an array address and the old growth factor instead of the value of ARRAY.

If the record contains a LISP-array, the elements of the segment of ARRAY from FIRSTELT to LASTELT are filled with the first (LASTELT - FIRSTELT + 1) elements of the array stored at KEY. If FIRSTELT or LASTELT are NIL or out of bounds, they default to the first and last elements of ARRAY respectively. If the array stored at KEY is smaller than the array segment it is being read into, the top part of the array remains unchanged, just as elements with index less than FIRSTELT or greater than LASTELT remain unchanged. The value of DFGETARRAY is the number of array elements read, or a two-element list containing the next key in key order and the number of array elements read for a sequential GET.

DFKEY[FILENAME; KEY]

FILENAME is a complete direct file name. DFKEY is a predicate that returns with value KEY if a record with the specified key is stored in the specified file and NIL otherwise. This is a way of checking which records exist in a file without actually reading them. If KEY is NIL, DFKEY returns the key of the last record touched by DFGET, DFGETARRAY, DFNEXTKEY, or DFLASTKEY. If DFKEY is called directly after DFBEGINSEQ with KEY=NIL, it returns the key of the first record in the file. Also, it is often desirable to

know if a key has an associated record, since DFPUT overwrites the records associated with previously used keys.

DFSAVE[FILENAME]

This function is equivalent to closing and reopening a file, and its raison d'être is that the system might crash. Hence it may be desirable to save a file several times during a long updating session to make sure updates are properly reflected in the file's index. Unfortunately, because of TENEX's poor file reliability scheme, even doing multiple DFSAVEs does not guarantee that updates during a session are completely safe if a crash happens soon after executing DFSAVE, because the system maps file pages back to the disk at its leisure.

DFSTATUS[FILENAME]

FILENAME is a complete direct file name. DFSTATUS returns a list of status information about a file. At present, the first element of this list is the total number of keys in the file, and the second element is the file format.

DFREORG[FILENAME; FORMAT]

FILENAME must be a complete TENEX file name including a version number, which may be open or closed. DFREORG creates a new version of this file with wasted space compressed out and with a new format if one is given. Both versions of the file will be closed when DFREORG is finished. The conditions under which wasted space enters a file are described in Section IV. If FORMAT is NIL, the old file format is retained. The value of DFREORG is the complete direct file name of the new version created.

DFADVISE

DFADVISE is a function of no arguments that carries on a dialog with the user, at the end of which it formulates a good format for his direct file based on the way he describes his application. The value of DFADVISE is the suggested format.

DFGENKEY[FILENAME]

This function generates a new file key sequence number for the specified file without writing out a record. The record can then be sequentially written at a later time by executing (DFPUT FILENAME KEY RECORD), where KEY is the value returned by DFGENKEY. The value of DFGENKEY is guaranteed to be a unique sequential key value for the file.

DFBEGINSEQ[FILENAME]

After calling DFBEGINSEQ, the next sequential read in the specified file will be the first record in key order.

DFNEXTKEY[FILENAME; KEY]

FILENAME is a complete direct file name. If KEY has a value equal to a key in the named file, then DFNEXTKEY returns this key as its value. If no such key exists in the file, the first key in the file greater (in key order) than the value of KEY is returned. If KEY is NIL, the next sequential key in the file since the last DFGET, DFGETARRAY, DFLASTKEY, or DFNEXTKEY is returned. In no case is the file record read; only the key value is returned. When the next key is the end of the file, NIL is returned.

DFLASTKEY[FILENAME]

FILENAME is a complete direct file name. DFLASTKEY returns the value of the key directly preceding in key order the last key read

from the named file by DFGET, DFGETARRAY, DFNEXTKEY, or DFLASTKEY. In no case is the file record read; only the key value is returned. When the preceding key is the beginning of the file, NIL is returned. If DFLASTKEY is called directly after DFBEGINSEQ, the first key is the last key in the file.

IV RELATIONAL DATA BASE PROCEDURES: GENERAL FUNCTIONAL DESCRIPTION

The relational data base procedures will be presented here in three logical groups. First, we will consider the set of functions for generating and modifying the data base schema. Once a schema is defined, it becomes possible to store, retrieve, and modify data in the schematized relations. This is accomplished with the access functions RELATION.GET, RELATION.FIX, and RELATION.PUT. RELATION.PUT stores a row of data specified as an a-list of associated attribute-value pairs in a relation (a-list is short for association list, which is described by example in the INTERLISP manual). RELATION.GET retrieves data after performing a pattern match of a "query template" against the data stored in a relation. This query template is, in essence, a filter through which the whole relation is passed. From a programming point of view, RELATION.GET is a CONNIVER style generator function which returns a "possibilities list" that can be used to successively produce rows of the subrelation defined by the template.* A third set of generator functions is provided that allows the set operations "and," "or," and "relative complement" to be applied to subrelations defined by possibility lists. The result is a completely general facility for forming subrelations limited only by the query pattern matching logic, which is complete, but is efficient for only certain types of queries.

*See Chapter 12 of the INTERLISP manual for a description of CONNIVER style generators and possibility lists.

In Figures 1 and 2, later in this section, we give a small data base example from our primary initial application, a management support system, to further explicate the concepts of data base schemas and relations.

Schema Definition and Modification

The complete data base schema is the union of the subschemas for each relation. The subschemas are defined and modified independently with the functions DEFINE.RELATION and REDEFINE.RELATION. The list of all currently existing relations is returned by the function RELATIONS, and the subschema for a relation is returned by the function SUBSCHEMA. The function ATTRIBUTE.DESCRIPTION retrieves the schema description of a single attribute. The operation of these functions is described below.

DEFINE.RELATION[RELATION.DESCRIPTION]

This function creates the subschema for a new data base relation. RELATION.DESCRIPTION is a list of the form:

(<relation name> <attribute description 1>
<attribute description 2> ...)

where <relation name> is a symbolic reference name for the relation, and each attribute description is an a-list with the required feature NAME giving a symbolic attribute name, and a set of optional attribute features that may include PRIM.TYPE, STAT.TYPE, UNITS, LIMIT, and others. The attribute name ROW.ID is reserved for system use and cannot be specified in a subschema. The full set of optional attribute features and their meanings is described in the appendix.

DEFINE.RELATION creates a new relation file called <relation.name>.RL, on which it stores the descriptive schema information in

preparation for use by the access functions. Each relation exists on a separate direct file called <relation name> and must conform to the restrictions on the initial field of a TENEX file designator.

REDEFINE.RELATION[RELATION.DESCRIPTION]

RELATION.DESCRIPTION has the same form as DEFINE.RELATION. This function modifies the subschema for a previously defined data base relation. The new schema definition is compared, an attribute at a time, with the old. The user is informed of any old attributes that have been deleted or had features changed, while the file is prepared to accept any newly defined attributes. Attribute descriptions are deleted at the end of the schema redefinition process, and a user confirmation is requested before attribute description records are deleted from the relation file. The new subschema description replaces the old one. The usual mode for redefining relations is to obtain the subschema of a relation with SUBSCHEMA, edit the returned value with EDITV, and then redefine the subschema with this edited value.

SUBSCHEMA[RELATION.NAME]

This function returns the current subschema for the named relation in the same form as the relation description passed to DEFINE.RELATION. The attribute descriptions are returned in the same order as they were given for the last DEFINE.RELATION or REDEFINE.RELATION operation.

ATTRIBUTE.DESCRIPTION[RELATION.NAME; ATTRIBUTE.NAME]

This function returns the attribute description for a named attribute of a named relation in the form given to DEFINE.RELATION. Note that

one of the attribute features returned is the "internal attribute number" ATT.NO, assigned by the system.

DELETE.RELATION[RELATION.NAME]

This function removes from existence the named relation and its associated retrieval files (not actually garbage collected until the next TENEX EXPUNGE operation). It returns NIL.

RENAME.RELATION[OLD.NAME; NEW.NAME]

This function renames the relation called OLD.NAME to have the relation name NEW.NAME henceforth.

Access Functions

The basic access functions for relations are RELATION.PUT and RELATION.GET, which store and retrieve rows of data in relations. When a row is added to a relation the system generates a unique row identifer which is the key under which the row is stored in the relational file. This row identifier is returned as the value of the reserved attribute name ROW.ID whenever a row is returned to a program in the form of an a-list of attribute-value pairs. Using this row identifer, individual rows of a relation can be read, deleted, or modified.

RELATION.PUT[RELATION.NAME; ROW]

This function stores a row in the named relation. ROW is an a-list that associates symbolic attribute names with their assigned values for this row of the relation. NIL will be stored as the value of all unassigned attributes. RELATION.PUT does type and value checking under schema control, described in the appendix. It prints error messages and returns NIL if it is given invalid data, and returns the value of ROW otherwise.

RELATION.GET[RELATION.NAME; QUERY.PATTERN]

The value of RELATION.GET is a possibilities list that can be used to generate one by one the rows of the named relation that satisfy QUERY.PATTERN. The rows are returned as a-lists of attribute name and value pairs. QUERY.PATTERN is used to select which rows should be returned, and it is an a-list of the form:

((<attribute #1 name> <condition #1>)
(<attribute #2 name> <condition #2>) ...)

The conditions are lists in one of the following possible forms:

Limits: (L . (<low value> . <high value>))
(L . (-INF . <high value>))
(L . (<low value> . INF))
(L . (-INF . INF))

Values: A single value given by an atom or a list of values prefixed by the atom "*."

Predicate: (P . <predicate name>)

RELATION.GET returns only those rows for which the conjunction of all attribute conditions are satisfied. A "limit" condition is satisfied if the value of the attribute in the given row is greater than or equal to the low value and less than or equal to the high value given. -INF and INF stand for negative and positive infinity respectively. A "value" condition is satisfied if the value of the attribute in the given row is one of the values specified. A "predicate" condition is satisfied if the named predicate applied to the value of the attribute returns non-NIL. In addition to attribute-condition pairs, QUERY.PATTERN can also contain a pair of the form (P . <predicate>) in which case the named predicate is applied to the row being considered and it must return a non-nil value for the two to match.

RELATION.NEXT[POSS.LIST]

Given a possibilities list returned by RELATION.GET or one of the functions for combining generators, RELATION.NEXT returns the next generated row of the relation it applies to. RELATION.NEXT returns NIL after the last row.

RELATION.READ.ROW[RELATION.NAME; ROW.ID]

In the named relation, read the row with the specified row identifier. The value is the row, returned as an a-list in the same form as that returned by RELATION.NEXT. If ROW.ID is NIL, read the next sequential row in the relation. To position the record pointer at the first row for sequential reading, use RELATION.FIRST.ROW. RELATION.READ.ROW returns NIL if the row does not exist.

RELATION.DEL.ROW[RELATION.NAME; ROW.ID]

In the named relation, delete the row with the specified row identifier. The value is T if the row exists, and NIL if it does not.

RELATION.FIX.ROW[RELATION.NAME; ROW.ID; NEW.ROW]

In the named relation, replace the row having the specified row identifier with NEW.ROW. NEW.ROW is an a-list like the one given to RELATION.PUT. The value of RELATION.FIX.ROW is the new row returned as an a-list if the row to be replaced existed, and NIL otherwise.

RELATION.FIRST.ROW[RELATION.NAME]

This function places the record pointer at the first row of the named relation so that the next call to RELATION.READ.ROW with ROW.ID=NIL will read the first row of the relation.

Functions for Combining Generator Sets

The value of RELATION.GET is a possibilities list that can be used to generate a set of rows of a relation. In order to provide a fully general method for selecting subrelations from relations, we allow these generators to be combined with logical set operations. Note that the universal set of rows for a given relation can be obtained by executing:

(RELATION.GET <relation name> NIL).

RELATION.INTERSECTION[POSS.LIST1; POSS.LIST2]

POSS.LIST1 and POSS.LIST2 are possibilities lists for subrelations (produced by RELATION.GET or combinations of generators produced by RELATION.GET). The value of RELATION.INTERSECTION is a possibilities list that can be used to generate the rows common to both of the input generators.

RELATION.UNION[POSS.LIST1; POSS.LIST2]

POSS.LIST1 and POSS.LIST2 are possibilities lists for subrelations (produced by RELATION.GET or combinations of generators produced by RELATION.GET). The value of RELATION.UNION is a possibilities list that can be used to generate the rows selected by either of the input generators.

RELATION.DIFFERENCE[POSS.LIST1; POSS.LIST2]

POSS.LIST1 and POSS.LIST2 are possibilities lists for subrelations (produeced by RELATION.GET or combinations of generators produced by RELATION.GET). The value of RELATION.DIFFERENCE is a possibilities list that can be used to generate the rows in POSS.LIST1, but not in POSS.LIST2.

Relation Demons

Relation demons provide a machanism for implementing alert functions and maintaining summary information. Demons are created and exterminated with the functions described below.

RELATION.MAKE.DEMON[RELATION.NAME; DEMON.FN; FIRING PATTERN; OPTIONAL.ARGUMENTS]

This function creates a demon that watches over the named relation. Every time a row is added to the relation with RELATION.PUT, deleted from a relation with RELATION.DEL.ROW, or modified with RELATION.FIX.ROW, if it satisfies FIRING.PATTERN, then DEMON.FN is called with four arguments in the following order: the relation name, the added row, CHANGEFLG, and OPTIONAL.ARGUMENTS. FIRING.PATTERN has the same form as QUERY.PATTERN, described for RELATION.GET. CHANGEFLG will have the quoted value PUT if the row is being stored with RELATION.PUT, DEL if the row is being deleted with RELATION.DEL.ROW, and FIX if the row is being modified with RELATION.FIX.ROW. The demon will remain activated over any number of sessions or processes until it is deleted, because it is stored with the data base. If the demon is made active over several sessions, it is up to the programmer to be sure that DEMON.FN is an available function during each session. The value of RELATION.MAKE.DEMON is the two element list:

(RELATION.NAME DEMON.FN)

RELATION.KILL.DEMON[RELATION.NAME:DEMON.FN]

Kill the named demon watching over the named relation. The value of the function is NIL.

An Exemplary Subschema for a Management Support System

Figure 1 illustrates a subschema specification for a management support system, which is our primary data base application. This relation records the time required to perform each subactivity of an aircraft mission run from a naval aircraft carrier.

```
(MST
     ((NAME . MISSION.ID)
      (PRIM.TYPE . NUMERIC)
      (STAT.TYPE . SEQUENTIAL)
      (RETRIEVAL . INDEX))
     ((NAME . PREFLIGHT.CHKOUT.TIME)
      (PRIM.TYPE . NUMERIC)
      (STAT.TYPE . CONTINUOUS)
      (UNITS . MIN)
      (RETRIEVAL . INDEX))
     ((NAME . BRIEFING.TIME)
      (PRIM.TYPE . NUMERIC)
      (STAT.TYPE . CONTINUOUS)
      (UNITS . MIN)
      (RETRIEVAL . TRANSPOSED)
     ((NAME . FUELING.TIME)
      (PRIM.TYPE . NUMERIC)
      (STAT.TYPE . CONTINUOUS)
      (UNITS . MIN))
     ((NAME . LAUNCH.TIME)
      (PRIM.TYPE . NUMEROUS)
      (STAT.TYPE . CONTINUOUS)
      (UNITS . MIN)
      (RETRIEVAL . TRANSPOSED)))
```

Figure 1. Subschema for MST (Mission Subactivity Timing)

Example of Data Stored in a Relation

In Figure 2 we show some exemplary data that might be stored in the relation schematized in Figure 1.

Relation: MST

Mission Id.	Preflight Checkout Time	Briefing Time	Fueling Time	Launch Time
#2931	10 mins.	15 mins.	10 mins.	7 mins.
#2932	15 mins.	25 mins.	12 mins.	6 mins.
#2933	12 mins.	20 mins.	8 mins.	9 mins.
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.

Figure 2. An Exemplary Relation

V DISCUSSION OF DATA BASE IMPLEMENTATION

In this section we will briefly discuss the implementation of the data base procedures described in the previous sections. More detailed implementation information is available from the author in working notes on direct access files and data base architecture.

Direct File Implementation

Direct files were implemented using the following INTERLISP primitives for random access to TENEX symbolic files: INPUT, INFILE, IOFILE, SFPTR, PEEKC, READC, READ, LINELENGTH, PRIN1, and PRIN2. A direct file actually consists of a header, followed by a collection of interspersed index and record blocks. The header and index blocks keep a symbol table for the file, while record blocks store data records.

The logical structure of a direct file index is a B-tree, which is a blocked, multilevel tree structure that requires at most $1 + \log_{\lceil b/2 \rceil} (N + 1)/2$ block references during searching and insertion, where b is the order of the tree and N is the number of keys in the tree.* A typical B-tree is shown in Figure 3. We call the highest level index block of this tree the root block.

*B-trees are discussed at some length in D. Knuth, The Art of Computer Programming, Vol. 3, pp. 473-480.

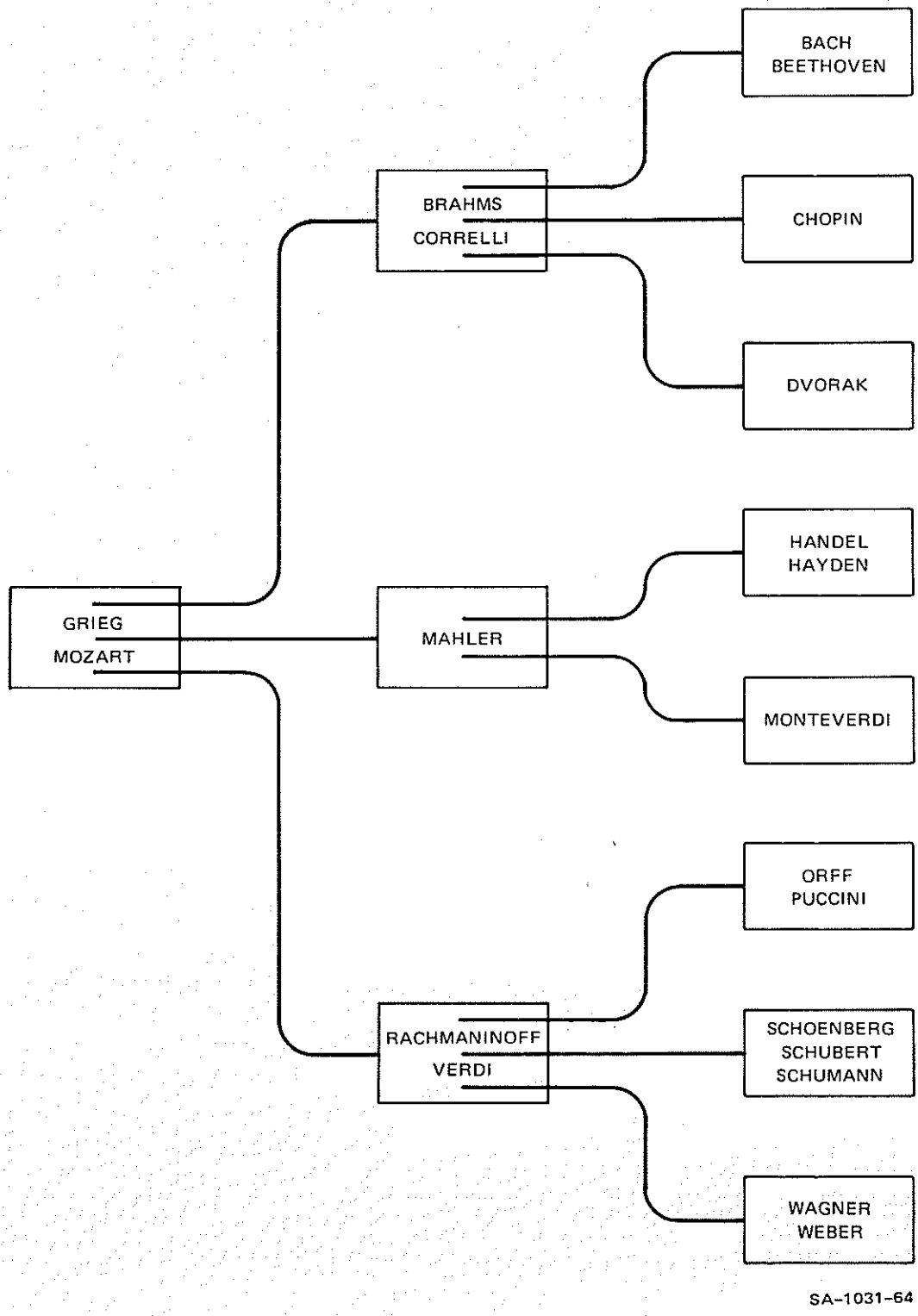

FIGURE 3 A TYPICAL B-TREE OF ORDER 4 WITH THREE LEVELS

The B-tree insertion algorithm used by the direct file package is Knuth's insertion algorithm with a restriction on both the maximum number of keys, NKEYS, and the maximum number of bytes, NBYTES, in a block. Both these limits are specified by the file format established when the file is first opened. The limits serve two purposes. Primarily they guarantee that reading an index block requires no more than one disk read, and secondarily they limit the size of the "window" into the file that resides in the user virtual address space. This window consists of index blocks that have been read into core and that remain there for processing.

One other format parameter, CORERES, combines with NKEYS and NBYTES to determine the total window size for an open direct file. CORERES determines the number of index blocks that are kept core resident while the file is active, either by giving the number of index blocks in the window, or by indicating what fraction of the complete index will remain in core. Index blocks are then demand-paged into the window with a least recently used replacement policy. Note that the root index block always remains in core, since it is the starting point of all searches.

Instead of Knuth's deletion algorithm, entries are deleted from the index by simply tagging them as deleted. Similarly, when a record is deleted the record block is simply tagged as being free. This deletion convention reduces the amount of time required to delete keys and records, as well as leaving allocated space for replacing deleted entries. However, it also implies that a significant amount of wasted space can accumulate in index blocks after several DFDELETE's, and wasted space can accrue in the file when records are replaced with larger records.

The function DFREORG compacts a direct file by removing wasted index and record space. It stores records in key order, thereby reducing the number of

disk pages that have to be read during sequential processing. DFREORG will also write the file with a new format, if desired, to improve its performance. It is intended that periodically, when files start to fill with wasted space due to heavy updates, DFREORG should be applied to them.

Assignment of File key Sequence Numbers

When records are written sequentially, DFPUT must generate a unique key, global to the file, under which to store the new record. Furthermore, the generated key should be greater in key order than the last sequentially generated key so that a file will be read sequentially in the same order that it was written sequentially. Both of these objectives are accomplished by the algorithm KEY SEQUENCE described below. KEY SEQUENCE generates numeric keys that can be accessed by the user like other keys, but that are guaranteed not to match any other key in the file at their time of creation.

Algorithm: KEY SEQUENCE

- When a file is created, store in its header a counter, DFGENCT, set to 0.
- Whenever a nonsequential DFPUT is executed with a numeric key, check that DFGENCT is greater than or equal to this key. If not, set DFGENCT to the value of this numeric key.
- When a sequential DFPUT is executed, increment DFGENCT. The increment value is the generated key sequence number.

Implementation of Relations

Relation files are implemented using the record-structured, direct access file procedures. Here we describe the format of a relational file as a direct file structure.

Each relational file contains several records that store subschema information used to table-direct RELATION.PUT and RELATION.GET. Under the key RELATION.LOCAL, a list of "local" information for each relation is stored. At present, the only local items stored are the number of assigned attributes in this relation and a list of the names of active demons and their firing patterns.

Under the name of each attribute of the relation, an attribute description is stored as an a-list of features. One of the features of an attribute is "internal" to the system--it associates an "attribute number" with each attribute. The attribute number is assigned by incrementing a counter as new attributes are defined with DEFINE.RELATION and REDEFINE.RELATION, and by storing the last assigned attribute number on the RELATION.LOCAL record.

The rows of a relation are stored at sequential keys and written from an array whose length is the number of assigned attributes in the relation, which we will refer to as the "buffer" array. The correspondence between attribute names and stored values is therefore maintained positionally, with the assigned attribute number corresponding to the position in the buffer array of an attribute value.

Ultimately the performance of RELATION.GET may be improved by using retrieval files, which store information from the primary relational files redundantly and in a form that is somewhat more difficult to update incrementally. The two retrieval files currently planned are INDEX and TRANSPOSED files. The schema language is used to specify which parameters should be maintained in INDEX and TRANSPOSED files with the RETRIEVAL feature (see the Appendix).

The INDEX file stores one record for each value of each indexed column of its relation containing the row identifiers of all rows that have this

value for this column. If columns get quite large and have few distinct values, it may be desirable to break the pointer sets for each value into several subrecords. This reduces the amount of core required to process index blocks.

The TRANSPOSED file stores one record for each column of its relation. The TRANSPOSED file reduces the amount of time required to process one or a few full column(s) of a relation, e.g. to find the mean and standard deviation of a column. If only a few columns are processed using the primary relational file, it is necessary to read the values of all columns for each row, only to discard most of these values as irrelevant to the computation.

Figures 4 through 6 illustrate graphically the file structure of the primary relational file and its associated retrieval files. The information in the files is based on the example given in Figures 1 and 2.

Key	Record
MISSION.ID	((NAME . MISSION.ID) (PRIM.TYPE . NUMERIC) ... (ATT NO. 1))
PREFLIGHT.CKOUT.TIME	((NAME . PREFLIGHT.CHKOUT.TIME) (PRIM.TYPE . NUMERIC) ... (ATT NO. 2))
.	.
.	.
.	.
(Row Identifier) #1	2931 10 15 10 7
#2	2932 15 25 12 6
.	. . . . .
.	. . . . .
.	. . . . .

Figure 4. The Primary Relational File: MST.RL

Key	Record		
(MISSION.ID . 2931)	1		
(MISSION.ID . 2932)	2		
.	.		
.	.		
.	.		
(PREFLIGHT.CHKOUT.TIME . 10)	1	15	35
(PREFLIGHT.CHKOUT.TIME . 12)	1	10	
.	.		
.	.		
.	.		

Figure 5. The Index File: MST.INDX

Key	Record			
BRIEFING.TIME	15	25	20	...
LAUNCH.TIME	7	8	9	...

Figure 6. The Transposed File: MST.TRN

VI CONCLUSIONS

The data base package presented here is quite general, and it provides applications support at two levels: the file system and data base management levels. It is the only implemented powerful secondary-storage oriented data base management package for INTERLISP of which we are aware. We anticipate that the procedures described here may be useful to several research projects underway, and we will be glad to furnish source programs and working notes to interested parties.

Appendix

THE RELATIONAL SCHEMA LANGUAGE

The possible features that an attribute description can have, and the meanings and effects of assigning values to these features are described below. The NAME feature is the only "required" feature of an attribute description in the sense that if NAME is not specified, DEFINE.RELATION and REDEFINE.RELATION will fail.

Feature Name	Feature Value
NAME	The value of this feature is a symbolic reference name for its attribute.
PRIM.TYPE	NUMERIC. Values are fixed or floating point numbers. RELATION.PUT checks that NUMBERP is true for this attribute's value. STRING. Values are character strings. RELATION.PUT checks that STRINGP is true for its value. SEXPR. Values are LISP atoms or s-expressions composed of LISP atoms.
STAT.TYPE	DISCRETE. Values are discrete, meaning that intermediate values are meaningless. Thus, statistics such as the mean of a discrete data item are meaningless.

CONTINUOUS, NORMAL, EXPONENTIAL. These specifications indicate the underlying distribution of a data item so that appropriate statistics and tests will be applied by general-purpose analysis programs.

UNITS — The value of this feature is a standard unit description for this attribute, such as "IN", "FT", "LBS", "SEC", "MIN".

RETRIEVAL — The value of this feature is either INDEX, TRANSPOSED, or (INDEX TRANSPOSED). It tells what retrieval files this attribute will be redundantly stored on.

LIMIT — Only allowed for attributes with data type NUMERIC. The value of this feature is a dotted pair of values. RELATION.PUT will check that the value of this attribute is greater than or equal to the CAR of the pair, and less than or equal to its CDR. Thus this feature is used for data validation.

VALUE — The value of this feature is a list of all the possible values for this attribute. RELATION.PUT will check that any value entered for this attribute is one of the values given. This feature may be used either for data validation, or as input to a histogram program.

ACKNOWLEDGEMENTS

I would like to thank Richard Fikes, who participated in the system design and in the report preparation; Marshall Pease, who participated in the system design; and Kazu Tachibana, who participated in the system design and implemented the relational data base procedures.

The research reported herein was primarily sponsored by the National Aeronautics and Space Administration under Contract NASW-2086, and the Office of Naval Research under Contract N00014-71-C-0294.