# A Comparison

# of

# Relational and Network

# Data Base Management Systems

# as Implemented on

# the

# HP/3000

by

Thomas R. Harbron

and

Christopher M. Funk

July 1981

**Christopher M. Funk & Co., Inc.**
22 North Second Street, P.O. Box 1249
Lafayette, IN 47902
(317) 423-2644

## INTRODUCTION

### Motivation

The "software crisis", which has been generally recognized in the last ten years, has created the need for tools that allow programmers and users of computers to be more productive. The traditional tools (compilers, editors, file systems, etc.) are not adequate to keep pace with the growing power of computers and the expectations of those who use and pay for them.

Consequently, the past few years have seen a multitude of products introduced which claim to improve programmer productivity or, in a few instances, eliminate the need for programmers, or at least coders. Specifically, in the HP/3000 product line, Image, Query, DEL, KSAM, and V/3000 have been introduced by HP. Outside vendors have added to this list with products which, while frequently improving on the HP products, are more imitative than inovative. For example, there are several "Query like" products available from independent vendors which extend the functions of Query and remedy several of its obvious deficiencies, but do not offer a fundamentally different kind of tool.

More recently, several inovative tools have appeared on the market. Among these are two relational database management systems, Relate/3000[1] and Rel*Stor. The inovative aspect of these products is that they are based on the relational model rather than the network model of Image.

### Objectives

The purpose of this paper, and the study on which it is based, is to compare these products with Image both in concept and implementation to determine the strengths and weaknesses of each. The goal is to select one of the three as the basis of further development of software tools.

The authors do not presuppose that one of these products will be clearly superior to the others or that one would be the best choice under all circumstances. However, this study should serve as the basis for a rational decision.

### Scope

To do a thorough analysis of these products, one should probably use each for a year or more in a variety of applications. Since this is not feasible, the authors have elected to evaluate them on the basis of:

1. The published specifications and user manuals;

2. The mapping of a small, but demanding database onto each system;

3. Performance on the HP/3000 as indicated by carefully chosen tests.

This analysis is further complicated because:

1. Both Rel*Stor and Relate/3000 are still under development with modules and features not yet implemented;

2. Their manuals are likewise under development, and not always in step with the product;

3. One product (Rel*Stor) was not made available for testing.

Therefore, the reader should be cautioned not to accept this study as the last word on these products.

## BACKGROUND

### Database Models

Most authors[2-7] list three different models for databases. These are idealized models of how data is naturally structured and do not consider questions of implementation or efficiency. Rather, the models are based on mathematical principles.

These three models are known as the network model, the hierarchical model, and the relational model. Virtually all database systems are based on one of these three models. Moreover, an important step in designing a specific database is modeling it in one of these three forms.

Each form has its own peculiar strengths and weaknesses. These are discussed briefly below. One problem found in discussing models and database systems is that each, generally, has a unique vocabulary. This is confusing enough when considering them one-at-a-time. When three models and three systems are discussed in one paper, it is hopeless. Therefore a "generic" vocabulary will be employed here as listed below. The authors apologize to those who may find these terms imprecise or contrary to standard usage:

| | |
|---|---|
| Entity | - an object or "thing" about which information is stored in a database. |
| Attribute | - a characteristic of an entity. Only attributes of an entity can be stored, not the entity itself. |
| Field | - the physical representation of an attribute. |
| Record | - the physical representation of an entity consisting of the fields that hold the attributes of that entity. |
| Key | - a set of attributes that distinguishes one entity from other similar entities. |
| File | - the physical representation of a group of similar entities consisting of the records representing those entities. |
| Relationship | - a logical connection between entities. For example, between a parent and children, or between a vendor and purchase orders to that vendor. |

The adjective "logical" will be applied to the terms field, record, key, and file when discussing the corresponding parts of the models.

A "standard" database problem will be used as an example throughout the remainder of this paper. This problem is a simplified accounting system. The entities and their attributes are as follows:

| Entity | Attributes |
|---|---|
| Department | Dept # - a unique number assigned to each department. |
| | Dept name - the common name of the department. |
| | Dept head - the name of the manager of the department. |
| Expense | Exp # - a unique number assigned to each expense type. |
| | Expense description - a description of the expense type. |
| Account | Account # - a unique number assigned to each account, consisting of a dept # concatenated with an expense number. |
| | Budget amount - the dollar amount budgeted for this account. |
| | YTD credit amount - the total dollar amount of all transactions credited to this account. |
| | YTD debit amount - the total dollar amount of all transactions debited to this account. |
| Transactions | CR Account # - the number of the account to which this transaction is credited. |
| | DB Account # - the number of the account to which this transaction is debited. |
| | Amount - the dollar amount of the transaction. |
| | Date - the date of the transaction. |
| | Reference - the account reference of the transaction. |

## Network Model

The network model is characterized by logical files, each of which represents an entity type. The logical files are connected by relationships that show how entities in one logical file are related to entities in other logical files.

The relationships are usually restricted to one-to-N or 1:N types. This means that exactly one entity in one logical file is related to N(zero or more) entities in another logical file. This is customarily noted by an arrow pointing from the "1" entity to the "N" entity. For example, a department entity may be related to many accounts while an account entity must be related to exactly one department.

More than one relationship may exist between two entities. For example, each transaction is related to exactly one account as a "credit account" and to exactly one account as a "debit account." This is done as two 1:N relationships from account to transaction.

A convenient way to represent a network model is by a "data structure diagram." Such a diagram is shown in Fig. 1 for the accounting problem. Note that entities are usually linked together by a shared attribute value. The name of the shared attribute is shown on the arrow in the diagram. The underlined attributes are keys.

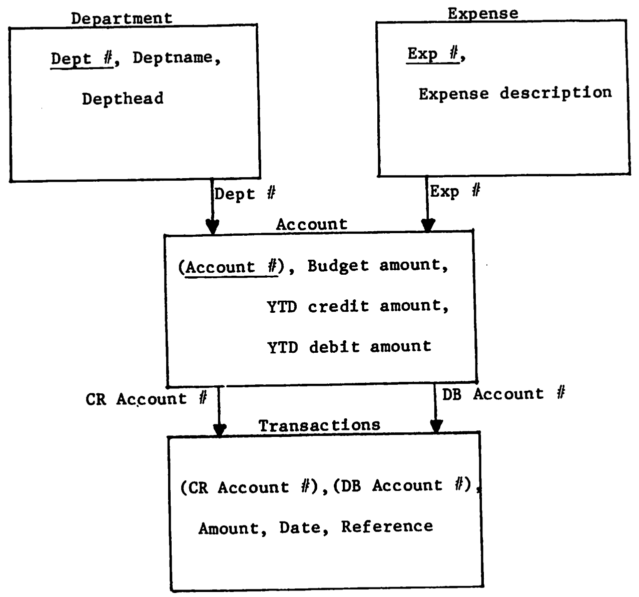

Figure 1

Notice that some of the attributes appear in parentheses. These are the same attributes that are used for the relationship linkage. Thus, it is redundant to show them as attributes; however, this is done in parentheses for logical completeness.

The network model is probably the most general of the three models. The network formed by the relationships can take on any topography and the links show the entity relationships. The other models are more restrictive.

## Hierarchical Model

The hierarchical model is, structurally, a subset of the network model; i.e. any hierarchical structure can be built under the rules of the network model. The difference is that additional constraints are imposed on the hierarchical model. These have to do with the relationships between entities and are as follows:

1. There is a unique entity type called the "root" where the hierarchical network begins.

2. Each entity, except those in the root, has exactly one "parent." A parent is another entity of a different type at a higher level.

3. Each entity, except those at the lowest level of the hierarchy, may have multiple "children." A child is another entity of a different type at a lower level.

The account example does not map easily into the hierarchical model because both the "account" and "transaction" entities have multiple parents. A better example is the bill of materials problem shown in Figure 2:

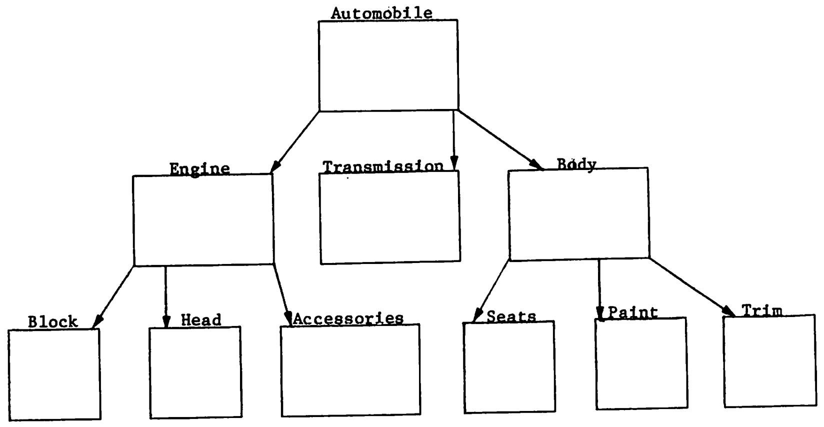

Figure 2

The hierarchical model is rightfully popular in situations where the data is naturally hierarchical. Otherwise, most of its usage seems to result from the dominance of several early database management systems based on this model. Many problems, including our elementary accounting example, would require unnatural restructuring to fit this model.

## Relational Model

While the network and hierarchical models are similar to each other, the relational model is totally different from them. The database consists of multiple logical files (called relations). Each logical file has one or more keys by which the logical records may be retrieved. There are no entity relationships of any kind connecting the logical files. The entity relationships can only be determined by comparing values of attributes of different entities.[8]

A simple list of the attributes of each logical record type, with an indication of the keys is sufficient to describe a model. For example, the following describes the relational model of the accounting problem:

| Logical File | Attributes |
|---|---|
| Department | <u>Dept #</u>, Deptname, Depthead |
| Expense | <u>Exp #</u>, Expdesc |
| Account | <u>Acct #</u>, Budamt, YTDCR, YTDDB |
| Transaction | <u>CRAcct #</u>, <u>DBAcct #</u>, Amt, Date, Ref |

Notice that transaction has two keys. Multiple keys are allowed in the relational model.

This model is unquestionably the simplest in appearance, and that is probably its greatest strength. It also has a firm mathematical foundation. Some authors[9] regard it as the most fundamental of the three models. However, it is probably the least implemented model because of two problems.

The first is a flaw in the model - the lack of explicit entity relationships. This can lead to what are called "insertion anomalies" and "deletion anomalies." For example, if a transaction is inserted, in our accounting problem, for which no credit account exists in the account logical file, the model will accept it. Likewise, a department could be deleted, thus "orphaning" the accounts associated with that department. Thus the rules necessary to avoid these anomalies must be imposed externally to the model.

The second problem is not so much with the model as with the implementations. The model makes it easy to request operations which are logically simple, but which require considerable resources and time to execute. Thus relational systems have earned a reputation for inefficiency.

## Normalization

This is one of the most important and poorly understood steps in designing a database. Normalization is essentially the process of discovering and isolating the entities represented by the data. There are three levels of normalization and the topic is discussed by most authors[2-7] with varying degrees of clarity. Atre[2] presents an unusually lucid discussion of normalization.

Normalization is nearly always presented in mathematical terms which, unfortunately, discourages some from investigating it further. A complete discussion is beyond the scope of this paper. However, normalized data will have the following advantages over intuitively designed, or unnormalized data:

1. Numerous types of insertion and deletion anomalies will not occur. These anomalies are of the type where the insertion or deletion of one entity has an unexpected or undesirable effect on another entity.

2. All entities will be readily accessible. Functions thought of after the database is designed will not require restructuring of the database.

3. A higher degree of data independence is possible. Programs are less likely to need change as the database is changed.

Neither the models nor the database systems have any way to enforce normalization. However, failure to normalize the data will inevitably create serious problems.

## Mapping

Mapping is a series of transformations on the structure of the database from its inception to the final, physical, database. There are five states in which the data is structured:

1. Initial data description

2. Normalized data description

3. The database model

4. The schema

5. The database

The mapping from the initial form to the normalized form is called "normalization" as described earlier. Normalization actually includes three separate transformations.

The mapping from the normalized form to the model is frequently accompanied by some compromises. If, for example, a hierarchical model is used, and the data is not inherently hierarchical, an artificial constraint is placed on the structure. Likewise, it may be necessary to "unnormalize" the data to some degree to fit the model.

Mapping from the model to the schema is really two activities that are done in parallel. First, the model must be mapped to the actual database systems. Some systems will be very close to the model and present little difficulty. Others may impose either structural or efficiency constraints which cause the structure to be altered significantly from that of the model. For example, the two-level limitation of Image requires compromises from the network model.

Second, the data structure must be expressed in a form acceptable to the database system. The form is called a "Data Description Language" or DDL. All database systems have a DDL. Some have a formal syntax, such as Image, while others may be conversational, such as Relate/3000. The data structure description expressed in a DDL is called a "schema."

The final transformation, of the schema into a database, is done by the database system. In most systems it is automatic with feedback in the form of error messages, status reports, and statistics.

## Implementation Considerations

The following items are factors to consider in judging the merit of a particular database system. Until the perfect system is developed, some will always be better than others on specific points. Different users will weigh these factors differently. However, all should be considered before a selection is made.

1. Mapping: What constraints are imposed when mapping from the model to the schema? Do significant changes have to be made? Are some things allowed, but not done because of performance considerations?

2. Data Manipulation Language (DML): The DML is the form in which requests are transmitted to the database system. Is the DML powerful? Is it easy to understand? Is it flexible? Can it be used from an application program? Is there a "stand-alone" mode?

3. Performance:

   a) Run efficiency: Are efficient search algorithms used? Is response time good? Are (logically) unnecessary accesses to secondary storage required?

   b) Storage efficiency: Is most storage space used for data? Do indexes, pointers, or other "non-data" items use up excessive space?

4. Concurrency: Has adequate thought been given to the problem of multiple users updating the database? What penalties or complications arise from shared access?

5. Restructuring: What needs to be done to change the database structure? What resources are required? What effect does restructuring have on existing applications? What must be done to initially load the database?

6. Security: How well protected is the data from unauthorized access? At what level or levels is security imposed: database, file, record, or item?

7. Integrity: Is the database prone to develop internal inconsistencies (broken chains, missing records, etc.)? Do aborts or crashes cause problems? What provisions are there for checkpointing (back-up copies) and journaling (transaction logging) and recovery?

8. Data Independence: Are application programs isolated from the physical storage considerations? Can changes be made in the physical or logical structure of the database without changing existing programs?

These implementation considerations, together with the strengths and weaknesses of the model on which it is based, will be used to judge each of the systems considered in the next section.

## Three Implementations

This section of the paper will consider three database systems: Image, Relate/3000, and Rel*Stor. For each of these systems, the strengths and weaknesses of the model and the implementation considerations will be discussed. Finally vendor information will be provided.

### Image/Query

Image[10] is based on the network model. As such it enjoys the benefits of explicit entity relationships of the 1:N variety, and even extends the concept by allowing the N entities to be ordered by the value of an attribute of that entity.

The DDL uses a formal, but concise, syntax. The mapping is straightforward with one glaring exception: a logical file (called a "set" in Image) cannot both be on the "1" side of some entity relationships and on the "N" side of others. This limits the system, physically, to two levels.

The problem is caused by the distinction between two kinds of files: masters and details. Masters are direct access files where a record is located by hashing on a single key. The hashing algorithms are effectively implemented and work with good efficiency. Detail files are essentially sequential-chronological files. Entity relationships are implemented using pointers to form a linked list or "chain" linking all related records. Each chain starts and ends on one record in a master set. The chain links any number of records (64K maximum) in a detail set. One master record may originate up to 16 different chains. One detail record may be linked into as many as 16 different chains. Detail records are normally accessed by following a chain from a master record. Sequential access is possible for both master and detail records.

The two-level structure causes difficulties, and requires compromises when mapping from model to schema. For example, the following data structure diagram represents the Image implementation of the accounting problem. Trapezoids are used to represent master sets while rectangles represent detail sets. Chains are represented by solid arrows while logical relationships (implemented programatically) are shown by broken arrows.

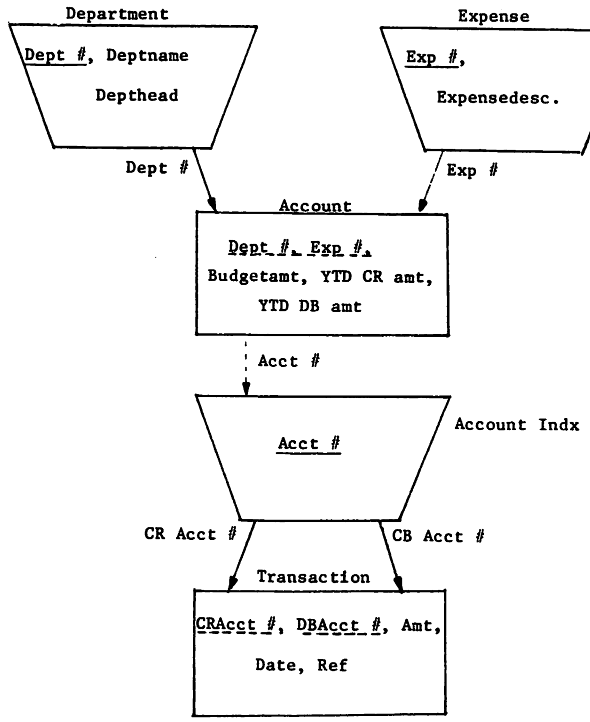

Figure 3

The broken underlines indicate that the underlined data item is a key only via the associated master. Image, however, requires that such fields be physically present in spite of the logical redundancy.

The ACCOUNT and ACCOUNTINDX sets are logically the same, but two are

required to circumvent the two-level problem. This requires redundant storage and additional access to secondary storage.

The schema for this data base is as follows:

```
BEGIN DATA BASE ACCTDB;
PASSWORDS:  <<NONE>>
ITEMS:
   DEPT,          X4;  <<DEPT #>>
   EXP,           X4;  <<EXP #>>
   DEPTNAME,     X20;  <<DEPARTMENT NAME>>
   DEPTHEAD,     X20;  <<DEPT HEAD'S NAME>>
   EXPDESC,      X20;  <<EXPENSE DESCRIPTION>>
   BUDAMT,        I2;  <<BUDGET AMOUNT>>
   YTDCR,         I2;  <<YEAR-TO-DATE CREDIT TOTAL>>
   YTDDB,         I2;  <<YEAR-TO-DATE DEBIT TOTAL>>
   ACCT,          X8;  <<ACCT # = DEPT#EXP#>>
   ACCTCR,        X8;  <<CREDIT ACCT #>>
   ACCTDB,        X8;  <<DEBIT ACCT #>>
   AMT,           I2;  <<TRANSACTION AMOUNT>>
   DATE,          I2;  <<TRANSACTION DATE>>
   REF,           X6;  <<ACCOUNTING REFERENCE>>

SETS:
   NAME:  DEPARTMENT, MASTER;
      ENTRY:   DEPT(1),
               DEPTNAME,
               DEPTHEAD;
      CAPACITY:  23;

   NAME:  EXPENSE, MASTER;
      ENTRY:   EXP(1),
               EXPDESC;
      CAPACITY:  23;

   NAME:  ACCOUNT, DETAIL;
      ENTRY:   DEPT (DEPARTMENT),
               EXP (EXPENSE),
               BUDAMT,
               YTDCR,
               YTDDB;
      CAPACITY:  100;

   NAME:  ACCOUNTINDX, MASTER;  <<LOGICALLY PART OF 'ACCOUNT'>>
      ENTRY:   ACCT(2);
      CAPACITY:  101;

   NAME:  TRANSACTION, DETAIL;
      ENTRY:   ACCTCR (ACCOUNTINDX),   <<CREDIT ACCT #>>
               ACCTDB (ACCOUNTINDX),   <<DEBIT ACCT #>>
               AMT,
               DATE,
               REF;
      CAPACITY:  6000;

END.
```

A variety of data types may be defined in the DDL. These are:

- 16 bit signed integer
- 32 bit signed integer
- 64 bit signed integer
- 16 bit unsigned integer
- 32 bit floating point
- 64 bit floating point
- Character string
- Zoned decimal
- Packed decimal

No provision is made to add user defined data types.

Image's DML consists of procedure calls which are compatible with all the standard languages. A summary of the procedures and their functions appears in Figure 4. An application program called "Query" is supplied with Image.[11] This program can be used interactively to access a database. It also serves as a report generator. Its usefulness is limited by its inability to look at more than one file at a time. However, it works very well otherwise and is simple enough to be used by non-programmers. Manipulations such as sorts and totals may be specified.

The run efficiency of Image is generally good. Performance problems are usually the result of design errors. For example, adding a detail record to a long ordered chain requires a sequential search of the chain. If there are 1000 detail records on the chain, 500 of them will (on the average) have to be read to determine the logical placement of the new record. Normally this would require 500 disk accesses! Thus, long ordered chains should be avoided.

Another source of performance problems can be record-level locking. Image uses dynamic locking to handle the concurrency. The locking may be done at the database level, file level, or record level. The lower the level, the greater the complexity[12] and the greater the overhead involved. The overhead at the database level is negligible; at the record level it is considerable.

Much of the time, blocking of records does not help reduce disk accesses. There is a provision to store detail records, that share a chain, in the order in which they occur on the chain. This physical ordering can only be done as part of restructuring and is not dynamically maintained. Thus there is usually a requirement for one physical access for each logical access.

Insertions and deletions require considerably more than one access. To insert a detail requires a minimum of four accesses for each chain involved as well as the write to put the record out. A deletion will usually require six accesses per chain.

The designer can consider these factors in mapping the schema and the resulting Image implementations can be as efficient as corresponding non-database applications.

| PROCEDURE | FUNCTION |
|---|---|
| DBOPEN | Initiates access to a data base. Sets up user's access mode and user class number for the duration of the process. |
| DBLOCK | Locks one or more data entries, a data set, or an entire data base (or a combination of these) temporarily to allow the process calling the procedure to have exclusive access to the locked entities. |
| DBFIND | Locates the first and last entries of a data chain in preparation for access to entries in the chain. |
| DBGET | Reads the data items of a specified entry. |
| DBBEGIN | When logging, designates the beginning of a transaction and optionally writes user information to the logfile. |
| DBMEMO | When logging, writes user information to the logfile. |
| DBPUT | Add new entries to a data set. |
| DBUPDATE | Updates or modifies the values of data items that are not search or sort items. |
| DBDELETE | Deletes existing entries from a data set. |
| DBEND | When logging, designates the end of a transaction and optionally writes user information to the logfile. |
| DBUNLOCK | Releases those locks obtained with previous calls to DBLOCK. |
| DBCLOSE | Terminates access to a data base or a data set, or resets the pointers of a data set to their original state. |
| DBINFO | Provides information about the data base being accessed, such as the name and description of a data item. |
| DBEXPLAIN | Examines status information returned by an IMAGE procedure that has been called and prints a multi-line message on the $STDLIST device. |
| DBERROR | Supplies an English language message that interprets the status information set by any callable IMAGE procedure. The message is returned to the calling program in a buffer. |
| DBCONTROL | Allows program operating in exclusive mode to enable or disable the "deferred update" option. |

Figure 4

The storage efficiency of Image is generally good, but again there are exceptions. Each chain requires 10 bytes for chain information in each master record, and 8 bytes in each detail record. Where several chains are involved this can become significant. Where the amount of data per record is small and the number of chains is high, the total storage required can be several times that required by the data. However, this is not typical. A modest "root file" is required to contain the schema and statistical information, but this is negligible in size.

As noted above, concurrency is handled by dynamic locking at several different levels. As with any dynamic locking situation, care must be exercised to avoid lockouts or deadlocks. Experience has shown that a dozen or so interactive users can share access to a database locking at the database level without serious contention problems. More can probably be accommodated with lower levels of locking.

Restructuring of an Image database is awkward at best and nearly impossible at worst. Essentially, fields may be added to records, and existing fields may be redefined. The maximum capacity of files may be changed, and chains may be added or deleted. Sometimes, but not always, new files may be added.

Restructuring is done with the utilities DBUNLOAD and DBLOAD. DBUNLOAD dumps records from the old database to magnetic tape. The old database is then purged and the new one is created from the revised schema. DBLOAD then loads the data from the tape to the new database. All pointers and chains are built anew by DBLOAD and the process can be very slow. The new database must be very similar to the old as no structure information is carried on the tape.

Security is very good with Image. In addition to the MPE file security, Image has an internal security mechanism that is very flexible. Up to 63 user classes, with associated passwords, may be defined. For each file and/or field, it is possible to specify which classes are allowed read access and which are allowed read/write access. All other user classes have no access. Image files are "privileged files" and cannot be accessed except through Image or by a privileged mode user.

The integrity of Image is unusually high. Crashes seem to cause a problem only when caused by a catastrophic hardware failure. Even then the problem can usually be fixed by deleting and replacing the record(s) involved. At worst a DBUNLOAD/DBLOAD will repair all structural damage.

Checkpointing can be done easily by using the utilities DBSTORE/DBRESTOR. These dump the files, with pointers, to tape and from tape to disk. Since no restructuring is done, they are very fast and efficient.

Journaling may be done with the transaction logging feature of Image. A utility is available to process the logged transactions to a checkpointed version of the database to recover all processing.

The degree of data independence can vary widely depending on the application programs themselves. At the low end of the spectrum, a program can, on a DBGET, request a physical record. At the other end, it can

request the specific fields wanted and the order in which they are delivered. A useful option is that of asking for the same list of variables used on the previous access of that file. This permits a logical "view" to be defined by the initial access(es). Thereafter the program sees this same view.

Image is structured as a set of user-callable procedures and several utilities plus Query. The utilities are only needed to restructure or recover the database and are not used for routine functions. All procedures are part of the application program's process. However, an extra data segment is created for each database that each process has open. In addition, all processes using a database share an extra data segment that serves as a common buffer and locking mechanism.

Image is a well established product and is nearly error free. It is available from:

Hewlett-Packard Co.
19447 Pruneridge Avenue
Cupertino, CA 95014

## Relate/3000

Relate is based on the relational model. Thus it enjoys the benefits of simplicity at the expense of losing the explicit entity relationships found in the other models. It is an unusually faithful implementation. with all standard features.

The DDL is conversational and informal. No "database" per se is defined. However, files (called relations in this model) are defined along with the name, and internal and external description of each field. These descriptions are stored in the "user label" area of each file. Thus the files are independent of one another. A database consists of those files a user has open at any given time.

Most relational systems provide for two types of relations or logical files. A "primary relation" is a permanent part of the database and is usually implemented as a physical file. A "derived relation" is one created during the run of an application and is usually not permanent. These derived relations are of two kinds: a "snapshot" is usually created by copying data from a primary relation to a new file. Thereafter it is independent of the original data. An "evolving view" is a rule that says how the derived relation is formed from the primary relations. The data remains in the primary relation.

Relate allows snapshots to be created at any time. In addition, evolving views may be created in two ways. A temporary, core resident view may be specified with the SELECT command. A permanent evolving view may be specified by the CREATE VIEW command. These are stored in separate, short files which contain the definition of the view, but no data.

A file (or relation) is created by the CREATE FILE command while keys are specified by the CREATE INDEX command. Examples of these commands are perhaps the most concise way to describe the DDL. The accounting problem will be used in these examples. Note that file names are limited to seven characters. Comments are enclosed in brackets { } but are not part of the session dialog; computer output is underlined:

```
{Create the department file}
> CREATE FILE DEPART; RECORDS=23
ENTER FIELD NAME, TYPE, LENGTH{.DECIMALS}
? DEPT, ALPHA, 4
? DEPTNAME, ALPHA, 2Ø
? DEPTHEAD, ALPHA, 2Ø
? //
THE "DEPART" FILE HAS BEEN CREATED AS A PERMANENT RELATE/3ØØØ FILE.

{Create the expense file. Here the description for each field is contained
 in the CREATE command}
> CREATE FILE EXPENSE;RECORDS=23;FIELDS=(EXP,ALPHA,4)(EXPDESC,ALPHA,2Ø)
THE "EXPENSE" FILE HAS BEEN CREATED AS A PERMANENT RELATE/3ØØØ FILE.

{A command may extend over multiple lines as follows}
> CREATE FILE ACCOUNT;RECORDS=1ØØ;FIELDS=&
&> (DEPT,ALPHA,4),&
&> (EXP,ALPHA,4),&
&> (BUDAMT,DOUBLE,13;COMMA=YES),&
&> (YTDCR,DOUBLE,13;COMMA=YES),&
&> (YTDDB,DOUBLE,13;COMMA=YES)
THE "ACCOUNT" FILE HAS BEEN CREATED AS A PERMANENT RELATE/3ØØØ FILE.

{Finally the transaction file is created}
> CREATE FILE TRANS;RECORDS=6ØØØ
ENTER FIELD NAME, TYPE, LENGTH{.DECIMALS}
? DEPTCR,ALPHA,4
? EXPCR,ALPHA,4
? DEPTDB,ALPHA,4
? EXPDB,ALPHA,4
? AMT,DOUBLE,13,COMMA=YES
? DATE,REAL,8;FORMAT="MM/DD/YY"
? REF,ALPHA,6
? //
THE "TRANS" FILE HAS BEEN CREATED AS A PERMANENT RELATE/3ØØØ FILE.

{Next the keys are defined with the CREATE INDEX command. The SET PATH
 command defines the "current" file for indexing}
> SET PATH DEPART
> CREATE INDEX BY DEPT;UNARY
{The "unary" specifies that keys must be unique}
> SET PATH EXP
> CREATE INDEX BY EXP;UNARY
{The following index contains one key formed by concatenating two fields}
> SET PATH ACCOUNT
> CREATE INDEX BY DEPT,EXP;UNARY
```

```
{The following indexes each have two keys, each of which is the concaten-
ation of two items; neither key must be unique}
> SET PATH TRANS
> CREATE INDEX BY DEPTCR,EXPCR
> CREATE INDEX BY DEPTDB,EXPDB
```

Each data file has one index file associated with it. Up to nine indexes may be defined for each data file. All indexes for one data file are stored in the one index file. The indexes are structured as "B-trees."[14] The B-tree is a tree structure which neatly solves the problems of making additions and deletions to the index, and is very efficient for retrieval. Appendix B of the KSAM manual[15] has a good presentation on B-tree indexes.

Eight different data types may be specified for the fields. These are:

- Character String
- 16 bit unsigned integer
- 16 bit signed integer
- 32 bit signed integer
- 32 bit floating point
- 64 bit floating point
- Packed decimal
- Zoned decimal

There is currently no provision for user defined data types, but this feature is under consideration.

An external format is also specified which has several options and some nice features. It is not as flexible as the PICTURE clause of COBOL or the FORMAT statement of FORTRAN, but better than the facilities found in Query.

The DML consists of two parts: commands and procedure calls. By far the greatest power and flexibility is in the commands. The procedures provide a better interface for application programs, in some cases, and somewhat more flexibility. Both commands and procedures may be accessed from an application program. The commands can also be used with Relate running as an interactive program as in the examples above.

The diagram in Figure 5 illustrates the program structure of Relate. The program Relate is the workhorse of the system. It is all that is needed when Relate is run as an independent program. When Relate is used from an application program, the "host language interface" library procedures are called by the program. These procedures, in turn, create a son process which runs the Relate program. All calls to these procedures are passed to the son process (Relate) for execution.

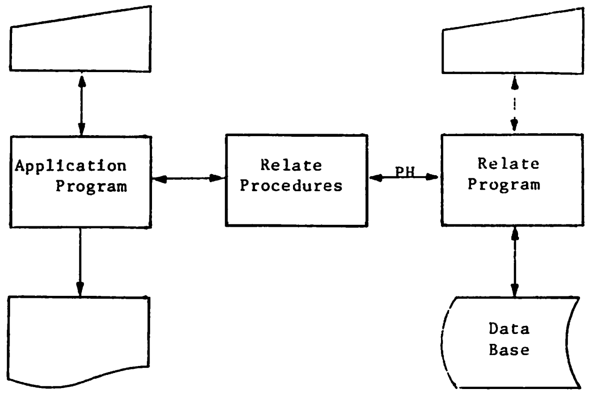

Figure 5

The following table lists the Relate commands with a brief description of the function of each. These commands may be used in the stand-alone mode or from an application program.

| Command | Function |
|---|---|
| ADD | Adds a record to the current file. |
| ALLOW | Sets the capabilities of different users. |
| CHANGE* | Modifies record(s) in a file. |
| CLOSE | Closes files or databases. |
| COMPARE | Compares contents of two files and selects either matching or unmatched records. |
| CONSOLIDATE* | Creates a subset of the current file. |
| COPY* | Copies the current file to another. |
| CREATE FILE | Creates a Relate/3ØØØ file. |
| CREATE INDEX | Creates an index for the current file. |
| CREATE VIEW | Creates an "evolving view" and stores its description in a file. |
| DELETE* | Purges selected records from current file. |
| DISABLE SECURITY | Releases Relate security. |

| Command | Function |
|---|---|
| DISALLOW | Inverse function of ALLOW. |
| ENABLE SECURITY | Turns on Relate security. |
| END | Terminates Relate program. |
| EXIT | Terminates Relate program. |
| EXECUTE | Causes Relate commands in a file to be executed. |
| HELP | Displays information about commands. |
| LABEL* | Prints records in label format. |
| LET* | Makes arithmetic or alphabetic assignments. |
| MODIFY | Changes the field formats or descriptions for the current file. |
| NOTE | The note command begins a comment line. |
| OPEN {Database / Path / File} | Opens the named database (Image) or file. "Path" is an alternate name for a file. |
| PRINT* | Displays selected data from current file on $STDLIST. |
| PURGE INDEX | Purges an index from the current file. |
| PURGE VIEW | Purges the named view. |
| RECOVER* | Restores records that have been logically, but not physically deleted. |
| REDO | One line edit function for previous command. |
| REORGANIZE | Physically removes logically deleted records and, optionally, changes file capacity. |
| SELECT* | Creates an "evolving view" and holds it in main memory for use by subsequent command(s). |
| SET INDEX | Specifies which index is to be used. |
| SET PATH | Specifies which file is to be used. |
| SHOW | Displays information about open files and indexes. |
| SORT* | Copies selected records from the current file to another file in order specified by sort key(s). |
| SUM* | Totals one or more fields in selected records. |

| Command | Function |
|---|---|
| SYSTEM | Sets global parameters. |
| TERMINAL | Specifies terminal characteristics. |
| UPDATE | Merges selected files. |

Those commands marked with an asterisk in the table above allow subsets of the records in the current file to be selected in either or both of two different ways. The first is by a range or ranges of values of fields that are indexed. For example:

```
    > SET FILE PEOPLE
    > SET INDEX NAME
    > "A"/"G" PRINT
```

will print the records for all people whose names are in the range A-G.

The selection may be further qualified by a "FOR" clause. For example:

```
    > "A"/"G" PRINT FOR DEPT = "SALES" AND SALARY > 25ØØØ
```

will print the records for all people whose names are in the range A-G, who work in the sales department, and whose salary is greater than 25ØØØ. Both ranges and FOR conditions may be compounded.

Again referring to the accounting problem the following dialog illustrates the power of some commands. As before, comments are in brackets, computer output is underlined.

```
{Open department file and add entries to it}
> OPEN FILE DEPART
> ADD
ENTER DEPT, DEPTNAME, DEPTHEAD
DEPT?     {Here the one field, or all three fields, may be entered.  If
           fields have been specified in a hierarchical form, only fields
           that differ from record-to-record need be entered.}
DEPT? //  {Returns control to command interpreter.}

{Next, a SELECT command is used to define an evolving view that combines
 data from two different files.  The COPY command then copies this view to
 another file.}
> SELECT DEPART.DEPT,EXPENSE.EXP
{This will select the dept # from the department file and the exp # from
 the expense file.  There are 8 records in the department file and 9 in
 the expense file.  A total of 8x9=72 combinations are possible, and that
 many records will be copied by the next command.}
> COPY TO ACCOUNT
{We now have 72 accounts in the ACCOUNT file.  We would next like to
 create one transaction for each combination of credit acct # and debit
 acct # except that the same account number may not appear in both places.
 This will give 72x72-72=5112 transactions.  There may be more elegant
 ways to do this, but the following sequence worked.  It was necessary to
 create a temporary file similar to ACCOUNT but with different field names.}

> CREATE FILE TEMP; STRUCTURE=ACCOUNT
{This creates a file of the same size and with the same field definitions
 as ACCOUNT.}
> SET PATH ACCOUNT
> COPY TO TEMP
{TEMP now contains the same data as ACCOUNT.}
> SET PATH TEMP
> MODIFY DEPT; NAME=DEPTDB
> MODIFY EXP; NAME=EXPDB
{The field names in TEMP have been renamed.}
> SET PATH ACCOUNT
> MODIFY DEPT; NAME=DEPTCR
> MODIFY EXP; NAME=EXPCR
{The field names in ACCOUNT have been temporarily renamed.  Names of fields
 in ACCOUNT and TEMP now correspond to those in TRANS.  Next, an evolving
 view will be defined as the product of account numbers in these two files,
 excluding cases where the two account numbers are identical.  These 5112
 transactions will then be created with the copy command.}
> SELECT ACCOUNT.DEPTCR,ACCOUNT.EXPCR,TEMP.DEPTDB,TEMP.EXPDB WHERE&
&>  ACCOUNT.DEPTCR <> TEMP.DEPTDB OR ACCOUNT.EXPCR <> TEMP.EXPDB
> COPY TO TRANS
```

Space does not permit a more complete display of the use of the commands. A rather nice demonstration package is available from CRI that shows more of the commands. A few hours "playing" with the system is also very instructive.

The programmatic interface consists of the eleven procedure calls listed in Figure 6. Notice that any of the commands may be used through the RELATE procedure. A "cursor" is a file control block. Multiple cursors may be used allowing multiple files to be processed concurrently.

The big problem with relational systems has always been run efficiency. Part of the problem comes from the apparent simplicity of the model - it is very easy to give a logically simple command that requires enormous resources. One of the authors, while experimenting with Relate, inadvertently gave a command that required $72^3$ = 373,248 logical file accesses. It required about 40 minutes to execute and, but for good blocking efficiency could have required much longer. A better way was found to do the same function in a few seconds.

The other source of low efficiency has been the indexing system. The B-tree structure has solved this nicely and this particular implementation is nearly optimally efficient. For example, a new index was created for the TRANS file of 5112 records in 78.5 seconds of CPU time. This works out to 15.4 milliseconds per record which is excellent.

Blocking factors are automatically chosen, and again, seem to be nearly optimal. Disk accesses appear to be the minimum possible in most cases tested.

| Procedure | Function |
|---|---|
| RELATE | Passes a command to the RELATE/3ØØØ data base management system. |
| RDBADD | Adds a new record to the file associated with the passed cursor. |
| RDBBIND | Binds a memory location for a return value or a substitution variable. |
| RDBCLOSE | Closes a cursor. |
| RDBDELETE | Deletes the current record from the file associated with the passed cursor. |
| RDBERROR | Returns information on an error condition that exists in a cursor. |
| RDBINFO | Returns information on the current file or status of the system. |
| RDBINIT | Initializes a cursor. |
| RDBPOINT | Positions a pointer to a specific record for reading. This call does not function on views or selections. |
| RDBREAD | Reads the next record from the associated cursor. |
| RDBUPDATE | Updates the current record on the file associated with the passed cursor. |

Figure 6

In short, both storage and run efficiencies seem to be very close to the ideal. Where performance problems arise, they can likely be traced to the ease with which some very awkward operations can be requested.

Concurrency is handled by dynamic locking of files. It may be enhanced beyond the file level in subsequent releases. Even at the file level, experience with Image indicates that it should be effective.

Restructuring is certainly one of the strong points of Relate. New files are easily defined and data from one or more files can readily be copied to the new file, with undefined fields zeroed or blanked. Moreover, Relate may be used with Image, KSAM, or MPE files as well as Relate files. Many of the commands including OPEN, COPY, PRINT, and CLOSE will work with these other file types. Not only does this give the user the option of combining these other file types into a Relate database, but it also makes the task of converting from the other file types to Relate files a trivial exercise. After one has struggled with restructuring Image, Relate seems almost too good to be true!

Security is defined with the ALLOW command. Essentially this specifies which users or groups of users can execute various groups of commands. For example use of the CREATE and PURGE commands can be restricted to one user. However, these security features only are effective for users going through the Relate system. Only the MPE file security is effective for a user who goes into a Relate file through the MPE file system.

An option exists to considerably strengthen the security by making the Relate files privileged files as Image does. However, this presently involves giving PM capability to the account and to the database administrator. Other options are under study to improve the security including encryption.

The integrity is similar to Image except that journaling (logging to a tape) and recovery from the log tape are not presently implemented. They could be done through application programming.

Data independence is also similar to Image. The user may accept all fields in the record as they physically occur or specify the fields and their order.

Relate/3ØØØ is a new product and, as with any new product of this complexity, can be expected to have some bugs in it. However, it appears to be soundly conceived, efficiently implemented, and a very effective tool. It is available for a one-time fee of $18,500 which includes the first year's maintenance. Maintenance after the first year will be 15% of the (then) current selling price. Relate is available from:

Computer Resources, Inc.
2750 El Camino Real
Mountain View, CA 94040
(415) 941-4646

## Rel*Stor

Rel*Stor is also based on the relational model and, with Relate, shares the particular strengths and weaknesses of that model. Unlike Image and Relate, which were created specifically for the HP/3ØØØ, Rel*Stor is implemented on other systems.

The DDL for Rel*Stor is slightly more formal than that for Relate, but closer to Relate's than Image's. The DEFINE command is used to create a file (relation or "table" as it is called in the Rel*Stor manual[16]). A database is created using the DEFINEDB command. This command specifies the name of the database and gives upper limits for the number of data files and users. Three directories are then created to store information on the database and its users, but no data files are specified or created.

Derived relations can be created as snapshots through the RETRIEVE command. The RANGE command allows the data for the snapshot to be gathered from more than one file. There is no provision for "evolving views."

The DEFINE command is functionally and syntactically similar to the CREATE command of Relate. For example, the ACCOUNT file would be created by the following:

```
DEFINE ACCOUNT
          DEPT    STRING   4,
          EXP     STRING   4,
          BUDAMT  INTEGER 1Ø,
          YTDCR   INTEGER 1Ø,
          YTDDB   INTEGER 1Ø,
          PRIME=2
          SIZE=23;
```

The PRIME=2 clause indicates that the first two fields (concatenated) constitute a unique key for each record. There is no provision for files, such as the transaction file, where no combination of fields yields a unique key. (Two transactions could be identical in all respects.) Likewise there is no provision for multiple keys. The single key may include several fields (in order of declaration starting with the first). The indexes are structured as B-trees with the advantages of ease of change and efficient retrieval.

There are only three data types specified, and data is stored in external (ASCII) form rather than internal (Binary) form. Data type appears to matter only when arithmetic operations are performed. The three types and the upper and lower limits on their "width" are:

| | |
|---|---|
| Integer | 4 to 12 bytes |
| Real | 8 to 12 bytes |
| String | --- |

These widths would allow both 16 and 32 bit integers, but only 32 bit floating point. No formatting capability is included.

The DML is quite rich and nearly as complex as a programming language such as Basic. This richness could be seen as Rel*Stor's strongest point. Its complexity could be a weak point.

Before proceeding to the DML it is necessary to understand the programmatic components of Rel*Stor and how they function. A program called the Relational Data Handler or RDH is the main workhorse of the system. The RDH is run as a son process of the user's process. The user's process may either be an application program or a program called the Terminal Interface Process or TIP. See Figure 7.

TIP serves as a conversational interface, text editor, and other miscellaneous functions. It operates in three different modes: control mode, edit mode, and administrative mode. The control mode allows the user to formulate queries, send them to the RDH, and see the results. The edit mode allows the user to compose, edit, modify and save queries. The administrative mode allows qualified users to define new databases, grant users access to databases and retrieve data base statistics. There are 45 TIP commands altogether as shown in Figure 8.

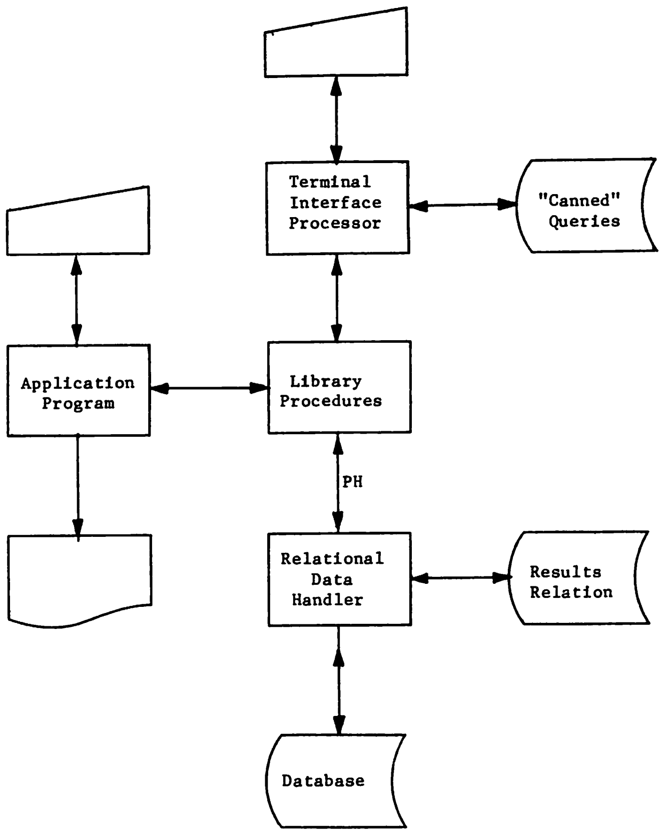

Figure 7

TIP Control Commands

| Name | Meaning |
|---|---|
| APPEND | append, insert, or change data in a relation |
| BYE | terminate a session with a database |
| COMPILE | translates Buffer to code and checks syntax |
| CONTINUE | continue query to retrieve next page of data |
| DBA | switch from Control to DBA mode |
| EDIT | switch from Control to Edit mode |
| END | return control to the operating system |
| GO | COMPILE and RUN the Textual Buffer |
| HELLO | initiate a session with a database |
| LISTREL | list relations and their data attributes |
| PRINT | change automatic PRINTRR mode |
| PRINTIO | print the number of disk accesses in your last request |
| PRINTRR | print the Result Relation |
| RUN | send the COMPILED Buffer |
| SAVERR | save the Result Relation, formatted for a LOAD command |
| SEND | send a canned query for processing |
| SETPAGE | set the maximum rows per page in the Result Relation |
| SETRR | reset the size of Result Relation file |
| SETWSR | reset the size of workspace relation file |
| SHOWME | shows the filenames and parameters of present session |
| STATISTICS | print the statistics associated with your last query |
| SYSTEM | print the system configuration parameters |

Figure 8 (contd)

TIP Control Commands

| Name | Meaning |
|---|---|
| TIME | print the time required to complete last query |
| // | terminates APPEND command |
| \ | terminates APPEND command |

TIP Edit Commands

| | |
|---|---|
| ADD | add lines to the Buffer |
| DELETE | delete lines from the Buffer |
| END | return to control mode |
| JOIN | JOIN together the Buffer and a KEEPed file |
| KEEP | save the contents of the Buffer in a permanent file |
| LIST | list lines of the Buffer |
| PURGE | delete a file created by the KEEP command |
| REPLACE | replace lines of text in the Buffer |
| TEXT | bring a permanent file into the Buffer |
| // | terminate ADD or REPLACE command |

TIP Administrator Commands

| | |
|---|---|
| ABORT | set query control point to inactive |
| ALTUSER | change or add to the capability of a user |
| COOL | to be implemented |
| DEFINEDB | define a new data base |
| DEFINEI | initial installation of RELATE |
| END | return to control mode |
| LISTDBS | list data base statistics or all data base names |

Figure 8 (contd)

TIP Administrator Commands

| Name | Meaning |
|---|---|
| LISTUSER | list directory for one user or all users |
| NEWUSER | enter a new user with RETRIEVE capability |
| PURGEUSER | remove a user from a data base directory |
| REMOVEDB | remove a data base |
| RESTRICT | to be implemented |
| WARM | to be implemented |

Figure 8

The programmatic interface to the RDH consists of four library procedures:

| Procedure | Function |
|---|---|
| STARTRDH | Creates the RDH process |
| TXTBFMGR | Sends a query to RDH |
| PROJECT | Returns one field at a time from the result of the query |
| GETDESC | Returns a description of the result of a query |

Queries are processed in a "batch" mode by the RDH; i.e. a query is passed to the RDH, the result of the search is placed in a temporary file called the "result relation." The results may then be interrogated or even saved as a new, permanent file.

Thus, in spite of the numerous TIP commands, it is the RDH commands that constitute the true DML, and even include most of the DDL. Figure 9 lists the RDH commands and operators which may be combined in the usual ways.

The RETRIEVE command is the primary read verb and the WHERE clause the qualifier. For example, the following query would retrieve all transactions debited to department #42 for expense categories 10-29 if the amount exceeds $100.00.

```
RANGE X = ACCOUNT
RETRIEVE X.DEPTDB, X.EXPDB, X.DEPTCR, X.EXPCR, AMT, DATE, REF
WHERE
  X.DEPTDB = 42 AND X.EXPDB >=1Ø AND X.EXPDB < 3Ø
  AND AMT > 1ØØ.ØØ
```

The LOAD command may be used to load data to a Rel*Stor file from an MPE file. It also may be used to add new records to a Rel*Stor file. There is no update verb in the RDH commands. The APPEND command in the TIP control mode will update. The only other option is to copy the entire file, modifying the necessary records. This seems to reflect a strong "batch" orientation rather than an on-line orientation.

Little can be said about performance since the product was not available for testing. The batch orientation of the DML, however, makes it likely that interactive processing, for any but a read only mode, would be awkward at best and could be very inefficient.

Storage efficiency is likewise less than optimum due to the storage of external rather than internal forms of the data.

Concurrency is not implemented at the present time. There are plans to add it in 1982. At present only one user at a time may access a database.

RDH Commands and Operators

| Name | Meaning |
|---|---|
| AND | logical (Boolean) operator |
| AVG | COLLECT function - averages real or integer |
| BUT | logical (boolean) operator - same as AND |
| COLLECT | perform function (COUNT, SUM, MIN, MAX, AVG) |
| COUNT | COLLECT function - counts rows |
| DEFINE | create a new data base table |
| DELETE | delete rows from a data base table |
| DELETEQ | DELETE but save DELETEd rows in Result Relation |
| LOAD | bulk load data into a table |
| MAX | COLLECT function - finds maximum in column |
| MIN | COLLECT function - finds minimum in column |
| NOT | logical (Boolean) operator |
| OR | logical (Boolean) operator |
| RANGE | specify table and assign relation variable(s) |
| REMOVE | purge a data base table |
| RENAME | rename a data base table |
| RETRIEVE | get information from a data base |
| SAVE | save the Result Relation as a data base table |
| SUM | COLLECT function - adds real or integer column |
| WHERE | introduce qualification clause to query |
| +, -, *, / | arithmetic operators - plus, minus, times, divided by |
| () | parenthesis - determine order of operation |
| := | value of expression is assigned to variable |
| <, <=, =, >=, > | relational operators |
| # | relational operator - is not equal to |
| % | relational operator - is included in - string only |

Figure 9

Security is defined by the NEWUSER and ALTUSER commands. These allow the administrator to control the users having access to each database, and the commands that each user may use. However, these security provisions are only effective for users going through the Rel*Stor system. Only the MPE file security is effective for a user who goes into a Rel*Stor file through the MPE file system.

Integrity is similar to Image except that there is no provision for journaling (logging transactions to tape) and recovery from the log tape. This could be implemented by application programs, however.

Data independence is also similar to Image. The user may accept all fields in the record as they occur, or specify the fields and their order.

Rel*Stor is a new product, with some features not scheduled for implementation until next year. As with any new product of some complexity, it can be expected to have some bugs. It was designed to work on a variety of computer systems and this may well cause some compromises in its design - for example the "batch" orientation of the RDH.

Rel*Stor is available for a one time fee of $20,000 which includes delivery, installation, three copies of the documentation, maintenance/update service, and hot-line consultation for 12 months. Maintenance/update service, and hot-line consultation costs $3000 per year after the first year. Rel*Stor is available from:

GTE Products Corporation
P.O. Box 188
Mountain View, CA 94042
(415) 966-2371

## SUMMARY

Any time that complex systems are compared, it is difficult to be totally objective because the designers of the systems had different goals and viewed various features with differing importance. Likewise, users of these systems will have varying needs. Thus it is not possible to rank these systems - any one of them might be the best choice for some applications. The grading chart, shown following, is an attempt to objectively assess salient features of each system. Even here, however, each individual grade must be a subjective judgment made after considering a variety of dissimilar factors.

## Grading

In the table below, the features of each of the three systems have been graded in ten categories. These grades are solely the judgment of the authors. The grade meanings are as follows:

A - excellent, outstanding
B - above average
C - average
D - below average but useable
F - essentially useless or non-existent
X - unable to determine

| | Image | Relate/3000 | Rel*Stor |
|---|---|---|---|
| 1. Mapping to system | D | B | B |
| 2. DDL and data types | B | B | C |
| 3. DML | C | A | D |
| 4. Run performance | B | B | X |
| 5. Storage efficiency | C | A | D |
| 6. Concurrency | B | C | F |
| 7. Restructuring | D | A | B |
| 8. Security | B | D | D |
| 9. Integrity | A | B | B |
| 10. Data independence | C | C | C |

## Future Developments

This paper is the result of one small step in a project to bring together a cohesive and effective set of program development tools. The keystone of this project is a data dictionary which would be shared by all components. At present each system (Image, Relate/3000, Rel*Stor, V/3000, etc.) has its own data dictionary (or fragment thereof) contained within it.

The time has come for a centralized data dictionary. The dictionary would necessarily be complex enough to require a database to store it. Systems could share the information and the data conversion procedures instead of each having its own (different) version. The authors plan to do future work in this direction.

## Bibliography

1. Newsletter of the HP General Systems Users Group, p. 24, April, 1981.
2. Atre, S., Data Base: Structured Techniques for Design, Performance, and Management. New York: John Wiley & Sons, 1980.
3. Date, C.J., An Introduction to Database Systems. Reading, Mass.: Addison-Wesley Publishing Co., 1977.
4. Kroenke, David, Database Processing. Palo Alto: Science Research Associates, Inc., 1977.
5. Tsichritzis, Dionysios C., and Lochovsky, Frederick H., Data Base Management Systems. New York: Academic Press, 1977.
6. Ullman, Jeffrey D., Principles of Database Systems. Potomac, Md.: Computer Science Press Inc., 1980.
7. Wiederhold, Gio, Database Design. New York: McGraw-Hill Book Co., 1977.
8. Date, op.cit., p. 53.
9. Atre, op.cit., p. 130.
10. Image Data Base Management System Reference Manual. Hewlett-Packard Co., Cupertino, CA., Part No. 32215-90003.
11. Query Reference Manual. Hewlett-Packard Co., Cupertino, CA., Part No. 30000-90042.
12. Image, op.cit., p. 4-17.
13. Relate/3000 Database Management System Reference Manual. Computer Resources, Inc., Mountain View, CA, July 1, 1981.
14. Comer, Douglas, "The Ubiquitous B-Tree," Computing Surveys, Vol II, No. 2, June 1979, pp 121-137.
15. KSAM/3000 Reference Manual. Hewlett-Packard Co., Cupertino, CA, Part No. 30000-90079.
16. User Reference Manual for Rel*Stor. GTE Products Corp., Mountain View, CA, March 1981.