IMAGE/COBOL: Practical Guidelines

David J. Greer
Robelle Consulting Ltd.
Aldergrove, B.C., Canada

SUMMARY

This document presents a set of practical "rules" for designing, accessing, and maintaining IMAGE databases in the COBOL environment. This document is designed to aid systems analysts, especially ones who are new to the HP3000, in producing "good" IMAGE database designs. Each "rule" is demonstrated with examples and instructions for applying it. Attention is paid to those details that make using the database trouble-free for the COBOL programmer, and maintaining the database easier for the database administrator.

CONTENTS

DATABASE DESIGN

IMAGE/3000 is the database system supplied by Hewlett-Packard;[6] it is used to store and retrieve application information. A database does not suddenly appear out of thin air; it develops through a long and involved process. At some time, a logical database design must be translated into the actual schema that implements a physical IMAGE database. This phase is the most difficult of the database development cycle.[7] The IMAGE/3000 Reference Manual[6] contains a sample database called STORE, which demonstrates most of the attributes of IMAGE. Throughout this document, the STORE database will be referenced when examples are needed.

Logical Database Design

The foundation of a new database is a logical design, which is created by examining the user requirements for input forms, for on-line enquiries, and for batch reports. The database should be viewed as an intermediate storage area for the information that comes from the input forms and is eventually displayed on the output reports.[9] [10]

Database design is normally done from the bottom up, as opposed to structured program design, which is usually done from the top down. The starting point for a database is the elements (items) that will be stored in the database. These data elements represent the user's information. In the early stages, the size and type of these elements are not needed, only the name and values.

Rule: *Start your logical database design by naming each data item, then identify what values it can have and where it will be used.*

Here is an example of a subset of data items for the STORE database:

```
CUST-STATUS      Two characters, attached to each customer record.
                     Valid values are: 10=advanced, 20=current,
                     30=arrears and 40=inactive.

DELIV-DATE       Six numeric characters; Date, YYMMDD, attached to
                     every sales order as the promised delivery date.

ON-HAND-QTY      Seven numeric characters, attached to every inventory
                     record to show the current quantity of an
                     inventory item available for shipping.

PRODUCT-PRICE    Eight numeric characters, (6 whole digits, 2 decimal
                     places), attached to every sales record.  This is
                     the price of a product sold, on the date that
                     the sale was made.
```

As the logical database design develops to deeper levels of detail, the elements needed should eventually reach a stable list. These elements should then be combined into records by grouping logically related items together.

It is important that "repetition" be recognized early in the design. An example of this is a customer's address. The most flexible method of implementing addresses is a variable number of records associated with the customer account number. Another method is to make the address field an X-type variable (e.g., X24) repeated 5 times (e.g., 5x24). Repeated items are often the most natural way to represent the user's data, so the use of repeated items is encouraged.

After the records are designed, enquiry paths must be assigned. During the early stages of database design, it is important to use elements that are readable and easy to implement with the tools at hand. This permits testing of the database using tools such as QUERY, AQ, and PROTOS.

Physical Database Design

After the local database is designed, the IMAGE schema must be developed. The restrictions of IMAGE must now be worked into the database design.

IMAGE requires that all items needed in the database be defined at the beginning of the schema, and a size and type must be associated with each. Initially, declare each item as type X (display); later, the data type may be altered.

Records are implemented as IMAGE datasets. Start by treating each record format as a master dataset.

Rule: *If a record is uniquely identified by a single key value, start by making it a master dataset (e.g., customer master record keyed by a unique customer number).*

The STORE database assumes that each CUST-ACCOUNT field is unique. Furthermore, there is only one customer record for each CUST-ACCOUNT. All of the information describing one customer is gathered together to result in the M-CUSTOMER dataset:

```
NAME:  M-CUSTOMER,        MANUAL (1/2);        <<PREFIX = MCS>>
ENTRY:
       CITY
      ,CREDIT-RATING
      ,CUST-ACCOUNT(1)        <<KEY FIELD>>
      ,CUST-STATUS
      ,NAME-FIRST
      ,NAME-LAST
      ,STATE-CODE
      ,STREET-ADDRESS
      ,ZIP-CODE
      ;
CAPACITY:  211;     <<M-CUSTOMER,PRIME; ESTIMATED>>
```

Rule: *If a "natural" master dataset will require on-line retrieval via an alternate key, drop it down to a detail dataset.*

The detail dataset will have two keys: the key field of the original master dataset, and the alternate key. You will have to create a new automatic master for the original key field, and you may have to create an automatic master for the alternate key (unless you already have a manual master dataset for that item).

Take the M-CUSTOMER dataset as an example. Assume that in addition to needing on-line access by CUST-ACCOUNT, it is also necessary to have on-line access by NAME-LAST. The following dataset structure would result:

```
NAME:  A-CUSTOMER,        AUTOMATIC (1/2),     <<PREFIX = ACS>>
ENTRY:
       CUST-ACCOUNT(2)       <<KEY FIELD>>
      ;
CAPACITY:  211;     <<A-CUSTOMER,PRIME; ESTIMATED>>

NAME:  A-NAME-LAST,       AUTOMATIC (1/2),     <<PREFIX = ANL>>
ENTRY:
       NAME-LAST(1)       <<KEY FIELD>>
      ;
CAPACITY:  211;     <<A-NAME-LAST,PRIME; CAP(A-CUSTOMER)>>

NAME:  D-CUSTOMER,        DETAIL (1/2);        <<PREFIX = DCS>>
```

```
ENTRY:
      CITY
     ,CREDIT-RATING
     ,CUST-ACCOUNT(!A-CUSTOMER)      <<KEY FIELD, PRIMARY>>
     ,CUST-STATUS
     ,NAME-FIRST
     ,NAME-LAST(A-NAME-LAST)         <<KEY FIELD>>
     ,STATE-CODE
     ,STREET-ADDRESS
     ,ZIP-CODE
     ;
CAPACITY:  210;     <<D-CUSTOMER; CAP(A-CUSTOMER)>>
```

Rule: *If an entry can occur several times for the primary key value, store it in a detail dataset.*

Detail datasets are for repetition and multiple keys. Master datasets can only contain one entry per unique key value. An example of repetition in a detail dataset is a customer address field. The customer address can be stored as a repeated field in a master dataset, but eventually there will be an address that will not fit into the fixed-size repeated field. Instead of a repeated field, use a detail dataset to store multiple lines of an address. For example:

```
ADDRESS-LINE,           X24;    << An individual line of address.  This
                                   item is used in D-ADDRESS to provide an
                                   arbitrary number of address lines for
                                   each customer.
                                >>
CUST-ACCOUNT,           Z8;     << Customer account number.  This field
                                   is used as a key to the M-CUSTOMER

                                     IMAGE/COBOL:  Practical Guidelines

                                   and D-ADDRESS datasets.
                                >>
LINE-NO,                X2;     << Used to keep address lines in D-ADDRESS
                                   in the correct order.  This field also
                                   provides a unique way of identifying
                                   each address line for every
                                   CUSTOMER-ACCOUNT.
                                >>

NAME:  D-ADDRESS          DETAIL (1/2);        <<PREFIX = DAD>>
ENTRY:
       ADDRESS-LINE
      ,CUST-ACCOUNT(!M-CUSTOMER(LINE-NO))  <<KEY FIELD, PRIMARY PATH>>
      ,LINE-NO                             <<SORT FIELD>>
      ;
CAPACITY:  844;     <<D-ADDRESS; 4 * CAP(M-CUSTOMER)>>
```

Dataset Paths

The following definition of PATHs and CHAINs comes from Alfredo Rego:[11]

> A PATH is a relationship between a MASTER dataset and a DETAIL dataset. The master and the detail must contain a field of the same type and size as a common "bond," called the SEARCH FIELD. A path is a structural property of a database.
>
> A CHAIN, on the other hand, contains a MASTER ENTRY and its associated DETAIL ENTRIES (if any), as defined by the PATH relationship between the master and the detail for the particular DETAIL SEARCH FIELD. . . . A chain is nothing more than a collection of related entries (for instance, a bank customer would be the master entry and all of this customer's checks would be the detail entries; the "chain" would include the master AND all its details; the chain for customer number 1 would be completely different from the chain for customer number 2).

Paths provide fast access at a certain cost: adding and deleting records on the path is expensive. The more paths there are, the more expensive it gets.[1] Another restriction of paths is that there can be a maximum of 64,000 records on a single path for a single key value. This sounds like a large number, but it can be very easy to expand a chain to this size if a key is specified for a specific, reporting summary program (e.g., billing cycle, in monthly billing transactions).

Rule: *Avoid more than two paths into a detail dataset.*

There are some instances where three paths are necessary, but these should be avoided as much as possible. Before adding a path, examine how the path is going to be used. If it is added just to make one or two batch programs easier to program, the path is not justified. The batch programs should serially read and sort the dataset, then merge the sorted dataset with any other necessary information from the database.

The date paths of the SALES dataset of the STORE database are good examples of unnecessary paths. Because the chain lengths of paths organized by date are almost always very long, such a chain is rarely allowed. Also, users are often interested in a large range of dates (such as a month, quarter or year), not just a specific day.

In order to obtain the same type of reporting by date, it is possible to do one of the following: (1) read the database every night and produce a report of all records entered every day; (2) keep a sequential file of all records added to the dataset on a particular day. This file can then be used as an index into the database.

These are not the only solutions to removing the date paths, but they indicate the kind of solutions that are possible. Because of the high volume and length of the average chain, date paths are prime candidates for removal from a database.

The following example demonstrates how the SALES dataset should have been declared:

```
<<  The D-SALES dataset gathers all of the sales records
    for each customer.  The primary on-line access is by customer,
    but it is necessary to have available the product sales
    records.  The PRODUCT-PRICE is the price at the time
    the product is ordered.  The SALES-TAX is computed based
    on the rate in effect on the DELIV-DATE.
>>

NAME:  D-SALES,            DETAIL (1/2);        <<PREFIX = DSA>>
ENTRY:
         CUST-ACCOUNT(!M-CUSTOMER)    <<KEY FIELD, PRIMARY PATH>>
        ,DELIV-DATE
        ,PRODUCT-NO(M-PRODUCT)        <<KEY FIELD>>
        ,PRODUCT-PRICE
        ,PURCH-DATE
        ,SALES-QTY
        ,SALES-TAX
        ,SALES-TOTAL
        ;
CAPACITY:  600;     <<D-SALES; 3 * CAP(M-CUSTOMER)>>
```

Rule: *Avoid sorted paths.*

Because sorted paths can require very high overhead when records are added or deleted, they should be avoided as much as possible. There are some instances when a sorted path makes the system and program design much easier, but this convenience must be traded off against the highest cost of maintaining sorted chains.

The most important criteria in evaluating sorted chains are: (1) whether the chain is needed for batch or on-line access. In batch, it is possible to read and sort the dataset, rather than relying on sorted chains. In an on-line program, this is usually not possible, so sorted chains are required. (2) How long is the average chain going to be? The longer the chain, the more expensive it is to keep sorted. If chains have fewer than 10 entries per key value on average, sorted chains can be permitted. (3) How are records being added to the dataset? If a sorted chain is present, and data is added to the dataset in sorted order, there is very little extra overhead in the sorted chain. If, on the other hand, data is added in random fashion, there is a very high cost associated with the sorted chain.[11–13]

Locking Strategy

Early in the database design, it is important to identify the locking necessary for the application. The easiest choice is to use database locking. Unless specific entries are going to be modified by many users, database locking should work. Remember: locking is only needed when updating, adding, or deleting entries from the database, not when reading entries. Never leave the database locked when interacting with the terminal user.

The next level of locking to be considered is dataset locking. This takes more programming, but provides for a more flexible locking strategy.

Rule: *Never permit MR capability to programmers;* instead, use lock descriptors (and a single call to DBLOCK) to lock all datasets needed.

For very complicated systems (e.g., an inventory system with inventory levels that must be continually updated), record locking should be used. The database design should help the application programmer by making the easiest possible locking strategy available for each program.[2]

Passwords

Most application systems go overboard in their use of database passwords. The simplest scheme to implement is a two-password system. The database is declared with one password for reading and one for writing. Each password is applied at the dataset level; and item-level passwords are not used.

Rule: *Use the simplest password scheme that does not violate the database integrity.*

The advantages to this scheme are that there are fewer passwords to remember, IMAGE is more efficient (because all security checks are done at the dataset level, instead of the data item level), and the user can still use tools such as QUERY, by being allowed the read-only password.

In sensitive applications, a separate dataset or database can be used to isolate data requiring special security. This still permits the simplest password scheme possible, with an extra level of security. The following example shows how to declare passwords for read-only access and read/write access on a dataset level:

```
    PASSWORDS:      1 READER;
                    2 WRITER;
```

The declarations for the M-CUSTOMER and the D-SALES datasets contain "(½)" on the line that declares the name of the datasets. The "(½)" indicates that the READER and WRITER passwords are in effect for the whole dataset.

Early Database Testing

The early database design should allow the user or analyst to experiment on the database design with test data. User tools such as QUERY or AQ should be used to access the database. At this stage, the item types may be left approximate, so long as the user or analyst gets a chance to interact with the database design. The analyst should check that all requirements of the user can be met by the database design.

Rule: *Build your test databases early. Use an application tool to verify that the database design is correct.*

In some cases, the end user may not be able to access the database, but the database designer must go through this testing process. This examination of the database design may uncover design flaws which can be fixed easily at this early stage. After the logical database design has been roughly packaged as an actual IMAGE database and verified against the user requirements, the design should be optimized and the finishing touches added (see next section).

Very Complex Databases

IMAGE has a number of size restrictions that it imposes on the database design. For example, the number of items in a database is limited to 255, and the number of datasets in a database is limited to 99. For many applications, these limits pose no problems; but with the larger databases being designed today, it is not difficult to imagine databases which exceed these lmits. What can you do to get around this problem?

Bottom-Up Design

The design method outlined above must be extended. For small projects, it is adequate to simply group related data items into datasets, because the entire application will fit into one database. However, for large projects, another step is required: related datasets must be grouped into separate databases.

Multiple databases introduce new problems for the application programmer. These include larger programs, which result in larger data stacks, as well as problems with locking. In designing a multiple database system, it is best to minimize the number of programs that must use more than one database.

If an application decomposes into independent subunits, few programs will require more than one database. The design of the system and the database may have to be revised to increase the independence of the sub-systems.

POLISHING DATABASE DESIGN

The database designer has two main concerns in completing the database design. Will the application programs be able to access the database within the defined limits of the HP3000? Does the database take best advantage of COBOL and other tools available?[3-8-11-13]

Overall Performance

Rule: *Always make a formal estimate of on-line response times and elapsed times for batch jobs.* If the project is going to require additional hardware resources, it is better to know it before the project goes into production.

The following material is taken from *On Line System*

Design and Development,[9] with comments and examples to expand on the original. The HP3000 is able to perform approximately 30 I/Os per second. On various machines under different operating systems, it may be possible to obtain more than this. Because it is extremely difficult to obtain the theoretical maximum of 30 I/Os per second, it is best to plan for a maximum of 20 I/Os per second.

Each IMAGE procedure results in a specific amount of I/O. Before going ahead with a large application, the total I/O required for the application must be computed and compared against the maximum. This is done by estimating the I/O for each on-line function, then summing the I/Os of the functions that might reasonably occur concurrently. Also, the total elapsed time for batch jobs must be estimated to ensure that they will complete in the time available.

The following gives an approximate measure of the number of I/Os necessary for each IMAGE procedure in an on-line environment:

Procedure	I/O
DBGET	1
DBFIND	1
DBLOCK	0
DBUNLOCK	0
DBUPDATE	1
DBPUT	2 + 2 * Number of keys in the dataset.
DBDELETE	2 + 2 * Number of keys in the dataset.

The figures for DBPUT and DBDELETE do not take into account sorted chains. If sorted chains are kept short, the above figures will work. If sorted chains are long, the following formula gives an approximate measure of how many I/Os are required to add records in random fashion to a sorted chain:

```
2 + 2 * number of keys + (average chain length / 2)
```

All of the above figures for the number of I/Os for each IMAGE procedure are the same in batch, with one exception. If a batch program reads a dataset serially, the I/Os required will be:

```
Serial DBGET I/Os = number of records / blocking factor
```

If the batch program also does a sort of all of the selected records from the serial DBGET, the number of I/Os will be increased.

Batch Calculation Example

The following example computes how long a specific batch program will take to run; the program makes the following IMAGE calls:

```
125,000 DBGETs serial; blocking factor is 5.
 80,000 DBPUTs to a detail dataset with two keys.
 80,000 DBFINDs.
 80,000 DBGETs to the dataset with the DBFIND.
 80,000 DBUPDATEs.

 Total I/Os required =

 I/Os for DBGET  (205,000 / 5) plus
 I/Os for DBFIND ( 80,000 X 1) plus
 I/Os for DBPUT  ( 80,000 X 6) plus
 I/Os for DBUPDATE(80,000 X 1).

 equals   681,000 I/Os.

 We can do approximately 20 I/Os a second so

      681,000
      -------    =  34,050 seconds  =  9.5 hours
        20
```

If the batch program also is intended to run overnight, but is unlikely to finish in one evening, because time is also needed for backup and other daily functions.

Improving Performance

How can the total time of this program example be reduced to 3.9 hours? One way is to replace the DBPUT with a DBUPDATE. In many instances it is possible, through changes in the application and database design, to use a DBUPDATE instead of a DBPUT. This is especially true in environments where there are recurring monthly charges, which change only slowly over time.

There is another advantage to using DBUPDATE. For each DBPUT, a record is added to the database, and this record must later be deleted using DBDELETE. Because it takes as long to delete the record as it did to add it in the first place, the DBUPDATE can provide as much as an eight-fold decrease in running time, compared with DBPUT/DBDELETE.

COBOL Compatibility

When designing a database, keep in mind how the database is going to be used (COBOL, QUERY, AQ, PROTOS, etc.). The following rules apply to item types and should be used throughout the database.

Numeric Fields

When the database was first designed, all fields were initially declared as type X (display). By now you should know the likely maximum value for each data item. Once the size of each data item is fixed, the time has come to specify a more efficient data type for numeric fields.

The type of field used for numeric values depends on the maximum size of the number to be stored (i.e., the number of digits, ignoring the sign). The following table should be used in determining numeric types:

```
Number of Digits  IMAGE Data type

      <5                 J1
      <10                J2
      >=10         Packed-decimal of the appropiate size.
```

Rule: *For numeric fields, use J1 for fewer than five digits; use J2 for fewer than ten digits; otherwise, use a P-field (packed-decimal) of the appropriate size.*

In COBOL, an S9(2)V9(2) COMP variable is considered to have a size of 4, or J1. The one exception to this rule is sort fields. All sort fields must be type X. If a numeric sort field is required, it must be declared as type X and redefined as zoned in all COBOL programs. Remember that packed fields in IMAGE are always declared one digit larger than the corresponding COBOL picture (S9(11) COMP-3 becomes P12) and must be allocated in multiples of four.

COBOL databases must not contain R-fields, because R-fields have no meaning in the COBOL language. Instead of an R-type field, a J-type or P-type field must be used. The STORE database contains an R-type field, CREDIT-RATING, which should have been declared as:

```
CREDIT-RATING,        J2;     << Customer credit rating.  The larger
                                 the number, the better the customer's
                                 credit.  Used to five decimal places.
                              >>
```

Key Types

Every key, whether in a master or detail dataset, must be hashed to obtain the actual data associated with the key value. Hashing is a method where a key value, such as customer number 100, is turned into an address. The method used tries to generate a different address for every key value, but in practice this is never possible. The choice of the type of key has a large bearing on how well the hashing function will work.

Rule: *Always use X-type, U-type, or Z-type keys, and never use J-type, R-type, P-type, or I-type keys.* Type X, type U, and type Z keys give the best hashing results.

When using a Z-type for a key, leave it as unsigned in all COBOL programs. Because key values rarely have negative values, there is no effect on the application by removing the sign from a zoned field. The advantages to leaving off the sign are: (1) displaying the field in COBOL or QUERY results in a more "natural" number, and (2) problems between positive, signed, and unsigned zoned numbers are avoided.

Date Fields

Rule: *Dates must be stored as J2 (S9(6) COMP) in YYMMDD format.*

This format provides the fastest access time in COBOL and takes the least amount of storage. Use a standard date-editing routine to convert from internal to external format and vice versa.[4]

The only exception to this is when a detail chain must be sorted by a date field. Because IMAGE does not allow sorting on J2 fields, X6 is used. For the chain to be sorted correctly, the date must still be stored in YYMMDD format.

Other Item Types

The only item types that should be used are J- or P-types for numeric values, and X-, U- or Z-types for keys. The K-, I- and R-types should never be used in a commercial application where COBOL is the primary development language.

Example

Earlier, in the discussion of logical database design, four items were described: CUST-STATUS, DELIV-DATE, ON-HAND-QTY, and PRODUCT-PRICE. The following example gives the actual IMAGE declaration for each of these items, according to the rules of this section:

```
CUST-STATUS,          X2;    << Defined state of a particular customer
                                account.  The valid states are:
                                10 = advance
                                20 = current
                                30 = arrears
                                40 = inactive
                             >>
DELIV-DATE,           J2;    << Promised delivery date.
                             >>
ON-HAND-QTY,          J2;    << Amount of a specific product currently
                                onhand.  Only updated upon
                                confirmation of an order.
                             >>
PRODUCT-PRICE,        J2;    << Individual product price, including
                                two decimal points.
                             >>
```

Primary Paths

Rule: *Assign a primary path to every detail dataset.*

IMAGE organizes the database so that accesses along the primary path are more efficient than along other paths. The primary path should be the path that is accessed most often in the dataset.

If there is only one path in a detail dataset, it must be the primary path. If there are two paths that are accessed equally often, but one is used mostly in on-line programs and the other mostly in batch programs, assign the primary path to the one that is used in on-line programs. A primary path is indicated by an exclamation point (!) before the dataset name that defines the path. A path with only one entry per chain should not be selected as a primary path.

The Schema

The IMAGE schema is the method by which you tell both IMAGE and the programmers what the database looks like. The schema should be designed with maximum clarity for the programmer, because IMAGE is only partly concerned with the schema's layout.

Rule: *The schema file name is always XXXXXX00, where XXXXXX is the name of the database.*

This naming convention makes locating the schema easier for all staff. The file is always located in the same group and account as the database. If the database name was STORE and the STORE database was built in the DB group of the USER account, the schema name would be STORE00.DB.USER.

Layout

A clear layout of the schema makes the programmer's job easier. Some requirements of the layout are imposed by IMAGE, but there are still a number of things that the database designer can do to make the schema more understandable.

Every database schema should start with a $CONTROL line. The $CONTROL line must always contain the TABLE and BLOCKMAX parameters. The default BLOCKMAX size of 512 should always be used when first implementing the database. Later, after careful consideration, the BLOCKMAX size may be changed. When first designing the database, $CONTROL NOROOT should be used.

The $CONTROL line should be followed by the name of the database. This is followed by a header comment. This comment describes the designer of the database, the date, the conventions used in designing the schema, abbreviations that are used within IMAGE names, and sub-systems with which the database is compatible and incompatible.

The following are the opening lines of the example STORE database:

```
$CONTROL TABLE,BLOCKMAX=512,LIST,NOROOT
BEGIN

DATA BASE    STORE;

<<                    STORE DATABASE FROM THE IMAGE MANUAL

AUTHOR:      DAVID J. GREER, ROBELLE CONSULTING LTD.

DATE:        DECEMBER 15, 1981

CONVENTIONS:

This schema is organized in alphabetic order.  All master datasets are
listed before detail datasets, and automatic masters come before
manual master datasets.

All dates are stored as J2, YYMMDD, except where they are used as
sort fields.  If a date is a sort field, it is stored as X6, YYMMDD.

The following abbreviations are used throughout the schema:

NO      = Number
CUST    = Customer
QTY     = Quantity

This database can be accessed by COBOL, QUERY, AQ and PROTOS.  Note
that the STREET-ADDRESS field is incompatible with QUERY, but AQ
can correctly add and modify the STREET-ADDRESS field.
>>
```

Naming of Items and Sets

Rule: *Names must be restricted to 15 characters; the only special character allowed in names is the dash (-).* This ensures that the names are compatible with V/3000 and COBOL.

The percent sign (%) should be replaced with the abbreviation "-PCT", and the hash sign (#) should be replaced with the abbreviation "-NO".

Item Layout

The easiest layout to implement, maintain and understand is to declare everything in the database sorted in alphabetic order. The items in the database should begin with a $PAGE command to separate the items from the header comment. Each item appears sorted by its name, regardless of the item's type or function.

In many IMAGE applications, the schema also acts as the data dictionary. For this reason, it is very important that every part of the database design be completely documented in the schema. Document each item as it is declared. To make each item stand out, the following layout should be used:

```
CUST-NO,                 Z10;      << The customer number is used as a
                                      key field in the M-CUSTOMER dataset.
                                      It is also the defining path in
                                      the D-ORDER-DETAIL dataset.
                                   >>
```

The item name, its type, and the comment start in the same column for every item. Each part of the item definition will stand out, and because the item names are in sorted order, the applications programmer can easily find a particular item.

Dataset Layout

Every dataset declaration must be preceeded by a header comment that describes the use of the dataset and any special facts that the programmer should be aware of.

When accessing the dataset from a COBOL program, it will be necessary to have a COBOL record which corresponds to the dataset. In order to prevent confusion between two occurrences of the same item as a field in several datasets, a prefix will be assigned to each of the variables in the COBOL buffer declaration. This prefix is selected by the database designer and must appear on the same line as the name of the database. For example:

```
<<  The M-CUSTOMER dataset gathers all of the static information
    about each customer into one dataset.  A customer must exist
    in this dataset before any sales are permitted to the
    customer.  This dataset also provides the necessary path
    into the D-SALES dataset.
>>

      NAME:   M-CUSTOMER,            MANUAL (1/2);  <<PREFIX=MCS>>
```

The AUTOMATIC, MANUAL or DETAIL keyword must always appear in the same column. This makes reading the schema easier, and by searching the file for a string (by using \L"NAME:" in QEDIT) it is possible to produce a nice index of dataset names, their types, and their prefixes. The following example prints an index of the STORE dataset names:

```
:RUN QEDIT.PUB.ROBELLE
/LQ STORE00.DB "NAME:"
NAME:  M-CUSTOMER,          MANUAL (1/2);         <<PREFIX = MCS>>
NAME:  M-PRODUCT,           MANUAL (1/2);         <<PREFIX = MPR>>
NAME:  M-SUPPLIER,          MANUAL (1/2);         <<PREFIX = MSU>>
NAME:  D-INVENTORY,         DETAIL (1/2);         <<PREFIX = DIN>>
NAME:  D-SALES,             DETAIL (1/2);         <<PREFIX = DSA>>
```

Rule: *Automatic master datasets have names that start with "A-".*

They must be declared immediately after the item declarations, separated from item declarations by a $PAGE command, and they must appear in alphabetic order.

Rule: *Manual master datasets have names that start with "M-".*

The manual master datasets follow the automatic master datasets, again preceded by a $PAGE command. Like the automatic masters, the manual master datasets must be declared in alphabetic sequence.

Rule: *Detail dataset names start with "D-".*

The detail datasets follow the manual master datasets, and the two are separated by a $PAGE command. The detail datasets also appear in alphabetic order.

Field Layout

Without exception, the fields in every dataset must be declared sorted alphabetically. There is a strong tendency to try to declare the fields within a dataset in some other type of logical grouping. Because this logical grouping exists only in the mind of the database designer and cannot be explicitly represented in IMAGE, it should never be used. By declaring fields in sorted order, the applications programmer can work much faster with the database, since no time has to be spent searching for fields within each dataset.

The database designer can still group fields together in a dataset by starting each field with the same prefix. If a dataset contains a group of costs, they might be called VAR-COSTS, FIX-COSTS and TOT-COSTS. To group these items together in the dataset, call them COSTS-VAR, COSTS-FIX and COSTS-TOT. This maintains the sorted field order in each dataset, while allowing for logical grouping of fields.

Most datasets contain one or more key fields. A key field is specified by following it with (). Because the () pair is sometimes hard to see, a comment should be included beside every key field, indicating that the field is a key. In a detail dataset, the primary key should include a comment to that effect. The following example shows how to declare the fields in a dataset:

```
<<  The D-SALES dataset gathers all of the sales records
    for each customer.  The primary on-line access is by customer,
    but it is necessary to have available the product sales
    records.  The PRODUCT-PRICE is the price at the time
    the product is ordered.  The SALES-TAX is computed based
    on the rate in effect on the DELIV-DATE.
>>

NAME:  D-SALES,          DETAIL (1/2);       <<PREFIX = DSA>>
```

```
ENTRY:
       CUST-ACCOUNT(!M-CUSTOMER)    <<KEY FIELD, PRIMARY PATH>>
      ,DELIV-DATE
      ,PRODUCT-NO(M-PRODUCT)        <<KEY FIELD>>
      ,PRODUCT-PRICE
      ,PURCH-DATE
      ,SALES-QTY
      ,SALES-TAX
      ,SALES-TOTAL
      ;
CAPACITY:  600;     <<D-SALES; 3 * CAP(M-CUSTOMER)>>
```

Capacities

Analysis of the data flow of the application should result in an approximate capacity for each dataset.

Rule: *The capacity of master datasets must be a prime number.*

To see if a number is prime :RUN the PRIME program contributed by Alfredo Rego. Master datasets should never be more than 80% full (see DBLOADNG below, under "Database Maintainence"), and detail datasets should never be more than 90% full.

The line with the capacity must be formatted in the following way:

```
CAPACITY:   211;     <<M-CUSTOMER,PRIME; ESTIMATED>>
```

The comment after the capacity gives a method for determining the approximate capacity of the dataset. Most detail datasets have a capacity that is related to the master datasets having paths into the detail datasets. These relationships should be described in the capacity comment.

By doing a \L"CAPACITY", it is possible to obtain quickly an index of the capacity of each dataset in the schema. Because the capacity is always the last line of each dataset declaration, doing a \L"M-CUSTOMER" will identify the beginning and ending declarations for the M-CUSTOMER dataset. The following example lists the capacity of the datasets in the STORE database:

```
:RUN QEDIT.PUB.ROBELLE
/LQ STORE00.DB "CAPACITY:"
CAPACITY:  211;     <<M-CUSTOMER,PRIME; ESTIMATED>>
CAPACITY:  307;     <<M-PRODUCT,PRIME; ESTIMATED>>
CAPACITY:  211;     <<M-SUPPLIER,PRIME; ESTIMATED>>
CAPACITY:  450;     <<D-INVENTORY; 2 * CAP(M-SUPPLIER)>>
CAPACITY:  600;     <<D-SALES; 3 * CAP(M-CUSTOMER)>>
```

Final Checkout

After the schema is entered into a file, it must be :RUN through the schema processor, and any typing mistakes should be eliminated:

```
:FILE DBSTEXT=STORE00.DB
:FILE DBSLIST;DEV=LP;CCTL
:RUN DBSCHEMA.PUB.SYS;PARM=3
```

The table produced at the end of the schema should be studied. The following anomalies should be checked:

1. Large-capacity master datasets with a blocking factor less than four (either increase the BLOCKMAX size to 1024, or change the master dataset to a detail dataset with an automatic master dataset).
2. The blocksize is too small (IMAGE optimizes the blocking factor to minimize disc space); use REBLOCK of ADAGER to increase the blocking factor. The blocksize of all dataset blocks should be as close to the BLOCKMAX size as possible.
3. Are there more than two paths into a detail dataset? If there are, can some of them be deleted?

Establishing the Programming Context

By using IMAGE, the COBOL programmer's job should be simplified, since all access to the database is done through the well-defined IMAGE procedures. Like most powerful tools, IMAGE (and COBOL) can be abused by the unsuspecting user.

Rule: *Define a standard IMAGE communication area* *and put this area in the COPYLIB.*

The starting point for using IMAGE is the standard parameter area, which includes the IMAGE status area, the various access modes used, a variable for the database password, and a number of utility variables which are needed when using IMAGE. For example:

```
05  DB-ALL-LIST                PIC X(2) VALUE "@ ".
05  DB-SAME-LIST               PIC X(2) VALUE "* ".
05  DB-NULL-LIST               PIC S9(4) COMP VALUE 0.
05  DB-DUMMY-ARG               PIC S9(4).
05  DB-PASSWORD                PIC X(8).
05  DB-MODE1                   PIC S9(4) COMP VALUE 1.
05  DB-KEYED-READ              PIC S9(4) COMP VALUE 7.
05  DB-STATUS-AREA.
    10  DB-COND-WORD           PIC S9(4) COMP.
        88  DB-STAT-OK             VALUE ZEROS.
        88  DB-END-OF-CHAIN        VALUE 15.
        88  DB-BEGIN-OF-CHAIN      VALUE 14.
        88  DB-NO-ENTRY            VALUE 17.
        88  DB-END-FILE            VALUE 11.
        88  DB-BEGIN-FILE          VALUE 10.
    10  DB-STAT2               PIC S9(4) COMP.
    10  DB-STAT3-4             PIC S9(9) COMP.
    10  DB-CHAIN-LENGTH        PIC S9(9) COMP.
        88  DB-EMPTY-CHAIN     VALUE ZEROS.
    10  DB-STAT7-8             PIC S9(9) COMP.
    10  DB-STAT9-10            PIC S9(9) COMP.
```

Rule: *Establish naming standards for all variables associated with IMAGE databases.*

Standard prefixes must be used on all database variables, including the database, dataset, data field and dataset buffer declarations. A suggestion is to start all database variables with "DB-", all dataset names with "DB-SET-", and all database buffer declarations with "DB-BUFFER-". Data field names are prefixed by the special dataset prefix (which the designer established in the schema), so that each field has a unique name. For example:

```
01  DATASET-M-PRODUCT.

    05  DB-SET-M-PRODUCT                PIC X(10) VALUE "M-PRODUCT;".

    05  DB-BUFFER-M-PRODUCT.
        10  MPR-PRODUCT-DESC            PIC X(20).
        10  MPR-PRODUCT-NO              PIC 9(8).
```

Field Lists

The selection of the type of field lists depends on the answer to this question: Can your total application be recompiled in a weekend?

Rule: *Use "@" field list is you can recompile in a weekend (prepare a COPYLIB member for each dataset); use "*" field list otherwise and hire a DBA!*

If the answer to the question is "yes," the at ("@") field list and full buffer declarations should be used when accessing the database. This method requires that all dataset buffers be declared and added to the COPYLIB. If a dataset changes, the buffer declaration must be changed in the COPYLIB, and all affected programs must be recompiled. The simplest solution is to recompile the complete application system whenever a dataset changes.

There must be two complete COPYLIBs available for every application. One is for production, and one is for development.

Rule: *Use a test COPYLIB during development. Double-check that all existing programs will recompile and :RUN correctly before moving the new COPYLIB into production!*

When a database is restructured, the buffer declarations are first changed in the development COPYLIB. When the new database is put into production, the development COPYLIB is also moved into production, as well as any programs that required modification or recompilation.

If the application system is so large that it cannot be recompiled in a weekend, it should use partial field lists and the same ("*") field list. This requires that an application program declare a matching field list and buffer area for each dataset that it accesses. The field list declares the minimum subset of the dataset that the application program needs.

Because partial field lists are more expensive at run time, the applications programmer must code a one-time call to DBGET for every dataset that the application program will use. The same ("*") field list is used on all subsequent DBGET calls. Note that this can cause problems if a common subroutine is called that uses one of the same datasets, but with a different field list.

In order to maintain an application with partial field lists, there must be a way to cross reference every program/dataset relationship. When a dataset changes, the cross reference system is checked to see which programs use the dataset. Each of these programs must be examined to see if it is affected by the change to the dataset. It is not enough to fix the COPYLIB and recompile, since the field declarations are in the individual source files, not in the COPYLIB file.

Dataset Buffers

The database designer assigns a short, unique prefix to each dataset of each database. These prefixes are used in the declaration of the database buffers for the datasets. In addition, dataset buffer declarations must include all 88-level definitions for flags, and sub-definitions for IMAGE fields that are logically subdivided within the application.

The following is the full buffer declaration for the M-CUSTOMER dataset of the STORE database. Note that each variable is prefixed with "MCS-", which is the prefix that was assigned by the database designer.

```
01  DB-BUFFER-M-CUSTOMER.
    05  MCS-CITY                        PIC X(12).
    05  MCS-CREDIT-RATING               PIC S9(4)V9(5) COMP.
    05  MCS-CUST-ACCOUNT                PIC 9(10).
    05  MCS-CUST-STATUS                 PIC X(2).
        88  MCS-CUST-ADVANCE            VALUE "10".
        88  MCS-CUST-CURRENT            VALUE "20".
        88  MCS-CUST-ARREARS            VALUE "30".
        88  MCS-CUST-INACTIVE           VALUE "40".
    05  MCS-NAME-FIRST                  PIC X(10).
    05  MCS-NAME-LAST                   PIC X(16).
    05  MCS-STATE-CODE                  PIC X(2).
    05  MCS-STREET-ADDRESS              PIC X(25) OCCURS 2.
    05  MCS-ZIP-CODE.
        10  MCS-ZIP-CODE-1              PIC X(3).
        10  MCS-ZIP-CODE-2              PIC X(3).
```

Repeated items should be declared with an occurs clause, or sub-divided, whichever the application requires. For example, a cost field may be declared as a repeated item representing fixed, variable, overhead, and labor costs. Rather than declare the costs field as a repeated item in the actual buffer declaration, subdivide it into the four costs. For example, assume a declaration for costs such as:

```
COSTS,            4J2;     <<Cost of an item.  Each cost has two
                             decimal points and the cost item
                             is broken down as follows:
                             COSTS(1) = Variable costs
                             COSTS(2) = Fixed costs
                             COSTS(3) = Overhead costs
                             COSTS(4) = Labour costs
                           >>
```

Assuming that the COSTS field was declared in the D-INVENTORY dataset, which has a prefix of "DIN", the following buffer declaration would be used for the COSTS field:

```
01  DB-BUFFER-D-INVENTORY.
    05  DIN-COSTS.
        10  DIN-VARIABLE-COSTS          PIC S9(7)V9(2) COMP.
        10  DIN-FIXED-COSTS             PIC S9(7)V9(2) COMP.
```

```
        10  DIN-OVERHEAD-COSTS          PIC S9(7)V9(2) COMP.
        10  DIN-LABOUR-COSTS            PIC S9(7)V9(2) COMP.
```

Rule: *Prepare sample COBOL calls to IMAGE in source files, with one IMAGE call per file.*

The sample IMAGE calls should be organized with one parameter per line. When programming, these template IMAGE calls must be copied into the COBOL program and modified with the database name, dataset name, and any other necessary parameters.

General purpose SECTIONS, declared in the COPYLIB, should NOT be used for the IMAGE calls. These SECTIONS obscure the meaning of the COBOL code. In addition, they can cause unnecessary branches across segment boundaries.

A scheme for handling fatal IMAGE errors must be declared, and the sample IMPAGE calls should refer to the fatal-error section. Here is a sample call to the IMAGE routine DBFIND:

```
    CALL "DBFIND" USING DB-
                        DB-SET-
                        DB-MODE1
                        DB-STATUS-AREA
                        DB-KEY-
                        DB-ARG-
     IF NOT DB-STAT-OK AND NOT DB-NO-ENTRY THEN
         PERFORM 99-FATAL-ERROR.
```

The fatal-error section (99-FATAL-ERROR) should call DBEXPLAIN. It should also cause the program to abort, and the system job-control word should be set to a fatal state. Note that just using STOP RUN will not set the system job-control word to a fatal state. The following is an example of a fatal-error section. The routine MISQUIT calls the QUIT intrinsic, which causes the program to abort.

```
$PAGE "[99]  FATAL ERROR"
*************************************************************
*  THIS SECTION DOES THE FOLLOWING:                         *
*  1.  CALLS DBEXPLAIN WITH THE IMAGE STATUS AREA.          *
*  2.  CALLS MISQUIT TO ABORT THE PROGRAM.                  *
*                                                           *
*  NOTE:  THIS MODULE MUST ONLY BE CALLED AFTER A FATAL ERROR*
*         HAS OCCURED WHEN CALLING AN IMAGE ROUTINE.        *
*                                                           *
*************************************************************

 99-FATAL-ERROR                SECTION.

     CALL "DBEXPLAIN" USING DB-STATUS-AREA.

     CALL "MISQUIT" USING DB-COND-WORD.

 99-FATAL-ERROR-EXIT.  EXIT.
```

Rule: *Avoid tricky data structures,* especially if they cannot be easily retrieved and displayed with the available tools (QUERY, AQ, PROTOS, QUIZ, etc.).

Some examples of data structures to avoid: (1) julian dates; (2) bit maps; (3) alternate record structures (REDEFINES); (4) implied and composite keys/paths; and (5) implied description structures. The more complicated the database structure, the more likely it is that programming or system errors will be created as a result of the database design.

Database Maintenance

There are a number of steps that the database administrator must take in order to guarantee that a database remains clean after it is implemented. A number of standard programs must be run against each production database at least once a month; others must be run daily.

Backup

A number of other people have commented on the backup problem of databases,[12] but the problem is important enough to deserve comment again. Most HP3000 shops do a full backup once a week and a partial backup once a day. This is normally sufficient for most purposes (e.g., source files, PUB.SYS, utilities), but it is not adequate for most IMAGE applications. An IMAGE database consists of several interrelated files. A database that is missing one dataset is nearly useless.

Rule: *EVERY backup tape should include ALL of the* *files of ALL of the database that are used in day-to-day applications.*

There should be an easy way to store complete databases onto partial backup tapes, without having to do selective stores. The BACKUP program (available from the San Antonio Swap Tape) helps solve this problem. The BACKUP program is run once a day against every production database. It accepts the database name as input and causes the last-modified date to be changed to today's date on every file of the database. This causes the entire database to be included on the daily partial backup.

In addition, the BACKUP program prints a listing with the following information: the dataset name, the current number of entries in the dataset, and the capacity of the dataset. Further, the BACKUP program examines the relationship between the number of entries and the capacity of each dataset, and prints a warning if it thinks the capacity is too small. This listing must be checked daily, in order to have time to expand the capacity of a dataset before it is exceeded.

Measuring Database Performance (DBLOADNG)

The performance of a given database will change as the database matures.

Rule: *The performance of every application database should be measured at least once a month.*

There is one program that will measure, in great detail, the performance of an IMAGE database. This program is DBLOADNG,[1–12] and it is available from the HPIUG contributed library.

DBLOADNG examines the performance of both master and detail datasets, and reports a large number of statistics. The most important are the percentage of secondaries in master datasets, and the elongation of detail datasets.

If there are a large number of secondaries in master datasets, either the hashing algorithm is not working well, or the capacity of the dataset needs to be increased. Note that the hashing performance of a key, such as customer number, can be improved by adding a check digit to every customer number.

The "elongation" of a detail dataset indicates whether logically related records are being stored physically adjacent. For primary paths, the elongation factor should be very small (1=perfect), since IMAGE tries to place records of a primary-path in the same disc block (see the DBLOADNG documentation and *Optimizing IMAGE: An Introduction.*[1]

If the performance of detail datasets is very poor because logically related records have been spread around the disc, there is only one solution: RELOAD the database using DBUNLOAD/DBLOAD. This will cause the detail dataset to be organized along the primary path, and could result in significant performance improvements.

Logical Database Maintenance

During the design phase of an IMAGE database, many logical assumptions are made about the data in the database. Some assumptions might be: (1) status fields, which are two characters long in a detail dataset, but have a long description in a master dataset; (2) keys that are stored in detail datasets, but do not have an explicit path into a master dataset; and (3) IMAGE chains that are limited to a specific length (e.g., one address per customer) or a range of lengths (e.g., no more than 10 items per order).

Rule: *When designing a database, keep a list of logical assumptions.*

These assumptions are dangerous, because they must be maintained by the application software, not by IMAGE.

Rule: *A program to check logical assumptions should be implemented for every application system.*

This program is often called DBREPORT, and its purpose is to check these logical assumptions. DBREPORT is often left until last, and often never implemented. This is unfortunate, since the DBREPORT program is *the* most important program in an application system.

In Alfredo Rego's paper, *DATABASE THERAPY: A practitioner's experiences*[12], he describes periodic checkups for a database. The following is taken from his paper:

> Please notice that a good diagnosis system must be nasty and sadistic by nature. It has as its primary objective to FIND ERRORS, not to certify a system as being error-free (there is no such system anyway!). A good diagnosis system must also be extremely patient and humble, since it will fail many times. Please keep in mind that there is a psychological inversion in effect here: A good diagnosis system fails if it does not detect any errors. And most of the time it will not detect any errors, since we hope and assume that the entity being tested is reasonably error-free."[12]

The DBREPORT program must be designed with Alfredo's philosophy in mind. It should check EVERY dataset in an application, and it should check EVERY record for logical consistency. This includes simple checks to see that every field in every dataset is within a reasonable limit. Examples of this are status fields that take on values from 1 to 10, but which are implemented as J1. A J1 variable can take on values from −32768 to +32767, which is certainly a larger range than 1 to 10.

The DBREPORT program must check all logical dataset relationships. What happens if every customer record has its address in a detail dataset? If the system crashes while the user is adding a new customer, the address record may not be added. DBREPORT must check for these types of relationships (what will your billing program do when it can't find an address?).

ADAGER

Rule: *If an application system is going to depend on IMAGE, ADAGER is a requirement, not an option.*

ADAGER provides all of the restructuring facilities necessary to maintain IMAGE databases; these transformations cannot be accoplished with DBLOAD/DBUNLOAD. Without ADAGER, numerous conversion programs must be written.

While DBLOAD/DBUNLOAD can be used for some simple database restructuring, it is prone to err. ADAGER is designed to be friendly to the end user, but, more importantly, ADAGER guides the user through every phase of the database restructuring process.

ADAGER provides a powerful facility, but it can also be misused by the unsuspecting. In order to make ADAGER changes effectively, test them first on a development database. Following changes to the database structure, the application programs must be recompiled (with buffers changed in the development COPYLIB), and each program must be tested against the new database.

Currently, ADAGER cannot be run from batch (at least, not conveniently), nor does it produce a hard-copy audit trail of the changes to a database.

Rule: *ADAGER must be run on a printing terminal.*

Keep the listing of the ADAGER changes to the test database. Use it to verify that the changes to the production database match exactly the changes to the test database. After changing the production database, move the development COPYLIB into production and recompile all affected programs. File the hard-copy listing of the ADAGER changes and keep it for future reference.

Because the schema is also used as the data dictionary, it must be modified to indicate the new database design. ADAGER's SCHEMA function can be used to double check that all schema changes were made properly. When modifying the database schema, be sure to apply all of the rules in the Schema section of this paper.

BIBLIOGRAPHY

To gain a complete understanding of IMAGE, study the references in this bibliography. A suggested order of study is: References 6, 7, 9, 10 and 11 for more ideas on database design; 5 for some hints on common programming errors; and 1, 3, 8, 12 and 13 for notes on optimizing IMAGE databases and application systems in general. Reference 1 is an excellent introduction to database optimization, and it includes a discussion of the DBLOADNG program.

[1]Rick Bergquist, *Optimizing IMAGE: An Introduction,* HPGSUG 1980 San Jose Proceedings.

[2]Gerald W. Davidson, *Image Locking and Application Design,* Journal of the HPGSUG, Vol. IV, No. 1.

[3]Robert M. Green, *Optimizing On-Line Programs,* Technical Report, second edition, Robelle Consulting Ltd.

[4]Robert M. Green, *SPLAIDS2 Software Package,* contains date editing routine (SUPRDATE) available from Robelle Consulting Ltd.

[5]Robert M. Green, *Common Programming Errors With IMAGE/3000,* Journal of the HPGSUG, Vol. I, No. 4.

[6]Hewlett-Packard, IMAGE/3000 Reference Manual.

[7]Karl H. Kiefer, *Data Base Design – Polishing Your Image,* HPGSUG 1981 Orlando Proceedings.

[8]Jim Kramer, *Saving the Precious Resource – Disc Accesses,* HPGSUG 1981 Orlando Proceedings.

[9]Ken Lessey, *On Line System Design and Development,* HPGSUG 1981 Orlando Proceedings.

[10]Brian Mullen, *Hiding Data Structures in Program Modules,* HPGSUG 1980 San Jose Proceedings.

[11]Alfredo Rego, *Design and Maintenance Criteria for IMAGE/3000,* Journal of the HPGSUG, Vol. III, No. 4.

[12]Alfredo Rego, *DATABASE THERAPY: A practitioner's experiences,* HPGSUG 1981 Orlando Proceedings.

[13]Bernadette Reiter, *Performance Optimization for IMAGE,* HPGSUG 1980 San Jose Proceedings.