TRICKS WITH(IN) IMAGE

Jan Janssens
Cobelfret N.V.
Antwerpen, Belgium

Summary

IMAGE-databases are very attractive for storing information: they are very reliable, they are easy to use, they can be accessed by querylanguages and they can be easy managed. Allmost every application on a HP3000, which is very important or deals with a huge amount of data, is forced to use IMAGE for its data-storage. (Of course there are other possibilities, but that seems to be more like re-inventing the wheel).

Unfortunately IMAGE/3000-databases have also some disadvantages: they can become slow if chainlengths or datasetcapacities are growing (10.000 entries is fine, but 1.000.000 entries is something else !!), they don't provide in a indexed access, their structure is sometimes too simple or rigid for the given problem.

This paper describes a method for "inventing tricks" to bypass the problem (which by the way can be applied to all kind of problems) and gives detailed (non-PM!!!) methods to overcome some IMAGE/3000-problems.

1. The "finding tricks"-technology

We will illustrate this technology through the use of the following example:

- A certain bookkeeping application had some databases and some other files in a separate account per company for which bookkeeping was done by a certain group of users. The files were builded by the financial on-line program and formed the input of a complex batchprogram that runned at night.

- If a user wanted to switch to another company, he had to log off and log on in the appropriate account. This involved a lot of overhead, and had to be done for every single inquiry in another account (company).

- The programs could easily be adapted to access the database and files in another account, BUT could not create (:BUILD) the inputfiles in another account (where the batchprogram and on-line programs of other users expected them) unless they should use PM.

1.1 Getting started

The first thing that has to be done is to FORMULATE THE PROBLEM. For the above example this seems to be "building files into another account". This is really THE problem, because if you could build

files into another account, the users could "swap" easily from the bookkeeping system of one company to that of another one (reinitialisation + close/open of database and files).

1.2 Think it over

The second step is : "WHY DO WE WANT TO SOLVE THIS PROBLEM?". Why do you want something to do that is very difficult, impossible or has some disadvantages ? In this case the reason is that programs expect to find the files in the "home-account", but that you can't build them from another one.

1.3 Look for "weak" and "strong" points

When we've located the real problem, we can attack it. It is comparable with the situation of an army that has surrounded a fortress and wants to take it over.
One could take the fort by attacking the big entrancedoor with a lot of soldiers. But is very likely that the 'current fortress owners' have made a fine defence system around the door and that the door is made of a very good material. The door is a "strong" point of the problem and its better to avoid it. Perhaps there are more less obvious ways in entering the fortress, but giving the same result. The defence system is perhaps not so strong if you could go in by the system that provides the fortress in fresh water.

In our case the implicit expectation of finding/building files in the homeaccount is the "strong point".
The "weak point" is that it is not necessarily to have the files in a certain account: as long as we are in the possibility to refind all the files that belong to one company, everything is ok!

1.4 Bypass the "weak" point

In the above case the problem can be bypassed by adding a dataset to the company database with a key=FILENAME and a field which has the fully qualified filename (filename + group + account) of the file under which name it was really saved.
All programs must first read in the (already open) database to find the actual filename! They also must do a delete or an update to it when the file is deleted or renamed.

2. Example of an own database system within an IMAGE database

2.1 Definition of the problem

The setup of a database system for a bookkeeping application posed the following problems (after the "think-it-over" phase):

- it must be possible to do extremely fast lookup in very long chains (i.e. give all entries for a big customer from 18 JUNE till 28 JULY)
- it must be possible to locate very fast all records belonging to a certain period (the period was unpredictable)
- the user-input, which consisted sometimes of more than 50 lines, must be registrated in a detail data-set with 3 keys (DOCUMENT, CUSTOMER, FILE), so that "up-to-second" consulting was possible.Allmost all transactions (say 98 %) were done on this data-set.
  Some chains exceeded the IMAGE/3000 upperlimit of 65535 entries.
- a payment of a single customer could cover more than 1500 entries, which must be updated on-line to avoid double usage. It must be possible to interrupt a transaction and to go on with it the next working day.
- the "accounts payable" and "accounts receivable" must be located in separate datasets to speed up batchjobs (serial read) and on-line transactions (reorganisation of the records to minimize DISCIO when consulting via the CUSTOMER-path (a frequent transaction))
- a certain department required such a complex presentation of their "accounts payable and receivable" that it was nearly impossible to do all that work at inquiry-time.(A lot of lookups in other data-sets was needed to define the sortcriteria and presentation lay-out).
  It must also be possible to run their job with a minimum elapsed time.
- it must be possible to use querylanguages on the developed datastructure.

2.2 "Strong" and "weak" points

The "strong" points were:

- the complete system must work with the same efficiency for big and small databases.
- the long chains couldn't be changed into a lot of small ones with each a different key.
- the large transactions couldn't be broken up into small ones.
- if IMAGE was not used then a lot of maintenance-, management- and inquiry-programs needed to be written

The "weak" points were:

1:- all transactions were done in a chronological order: at each moment, input was done in only two (bookkeeping)periods, and the periods changed always in a chronological manner. Consulting was also done in chronogical order.

2:- info entered in the system, was never deleted to allow full auditing.(Besides of "clean-up" jobs after some years).

3:- the number of "outstanding records" was only a fraction of the complete history. People wanted to keep track of all bookings for at least 2-3 years (900.000 records), and the "outstandings" covered only 30.000 records.

4:- when entering information in the detail-dataset with 3 keys, most of the time 2 of 3 keys didn't change within the transaction. A lot of transactions also left some keyvalues zero or blank.

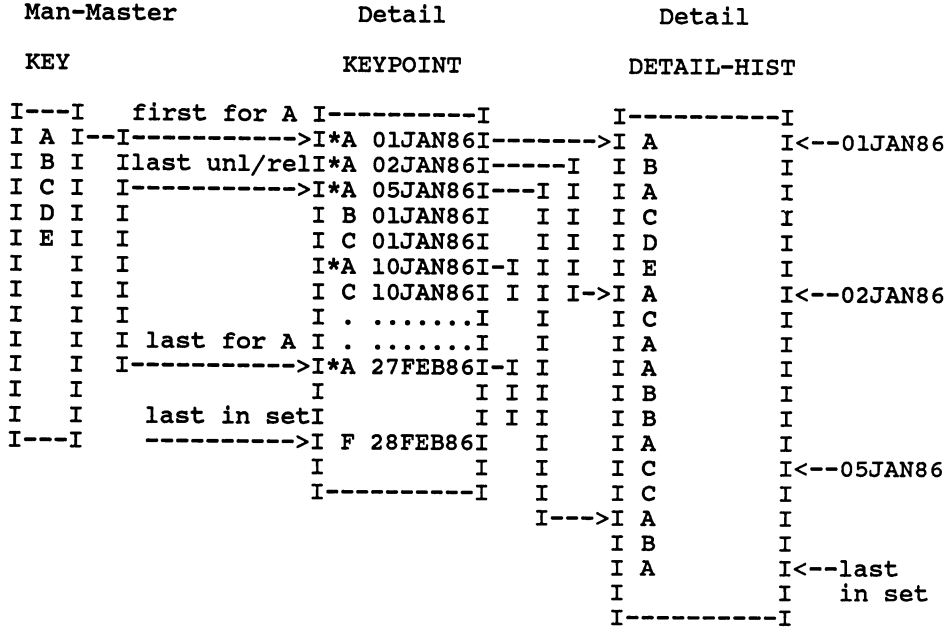
5:- the department that required the complex inquiries and jobs was a very important, but a small one.

6:- it was unnecessary and even unwanted to update immediately all payments. As long as one could not pay 2 times the same invoice, everything was ok. By not doing the update immediately users had the time to correct mistakes they encountered when closing their payments at the end of the day.

2.3 Bypassing the "weak" points

2.3.1 Data-structure for "historic" information

Working on the first 2 "weak points" the following design was drawn for the "history" detail-dataset:



- The detail data-set with 3 keys is a stand-alone detail-dataset. This dataset is always filled up with "empty" records. This gives us the opportunity to place the records where we want them by doing an DBUPDATE instead of a DBPUT. To find all records within a given period, we only have to read all records starting with the first record of the given "startdate" till we find a record with a date greater than the "enddate".An index is maintained to keep track of the first record for each date and the last "used" record. Locating ALL records of ALL keys within a given period out of a dataset of more than 1.000.000 entries, is now possible in a fraction of a second!!!!!

- For each of the 3 keys a pointerchain is maintained in the detaildataset. Because no information will be deleted and consulting doesn't need a "backchained read", a "forward"-pointer is enough. The pointerchain gives us the possibility to do a "chained get" a certain record of a chain is read.

- For each key a "KEYPOINT"-data-set is maintained which gives the recordnumber of the first record of that key with a date greater or equal to a given date. This data-set is also a stand-alone detaildataset prefilled with "empty"-records and contains again a pointerchain for each keyvalue. The masterdata-set "KEY" has pointers to the first and last entry for that key in the "KEYPOINT"-data-set. In addition the last "free" entry in KEYPOINT must be kept aside.

  In the KEYPOINT-dataset are pointers written if:

  - the number of records written in DETAIL-HIST, since the last record which has a pointer to it in KEYPOINT, exceeds a given number. The recordcounter and its upperlimit are also stored in the KEY-dataset.

  AND

  - the date differs from the date in the last record for the key in DETAIL-HIST.

  This system allows to define how many "entrypoints" you want to the chain of a given key. The number of entrypoints grows dynamically with the number of entries in the detail-dataset.

- To locate the first entry for a key for a date greater or equal to a given date, the search can be done in the data-set KEYPOINT.
  Once a record is read with a value greater than the given date, one must "backtrack" one record, read in DETAIL-HIST at the location given by the pointer of KEYPOINT and read further in DETAIL-HIST by following the pointers of the appropriate key until a record is read with date greater or equal to the asked startingdate.
  Because KEYPOINT contains less records than DETAIL-HIST and has a greater blocking factor, locating of a record in the chain goes much faster than a chained read in DETAIL-HIST.

- If the number of records for a given key in KEYPOINT becomes great, localizing records in DETAIL-HIST will slow down (although it will still be faster than the chained read in DETAIL-HIST).
  To overcome this situation an "unload-reload" of the data-set KEYPOINT can be done, so that records with the same keyvalue become adjacent.
  This gives already a very good improvement by eliminating a lot of disc IO (KEYPOINT can have a blocking factor of 80-100).
  The "unload-reload" can be done by writing a simple program. Because this program can write its records from the end to the beginning (only forward pointers are maintained) and only DBUPDATE's are done, processing of it goes very fast.

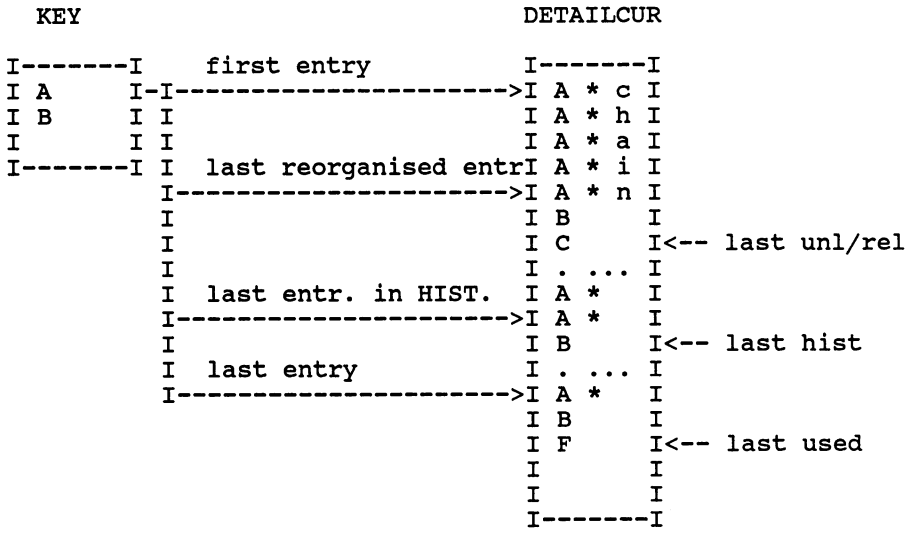
- It is also possible to do a "partial unload-reload" : one could reorganize only those entries after the last record of the last "full unload-reload". This action puts all records of the same key together in two blocks instead of one (one "partial reload" after a "full reload").
  Because we have only forward pointers, one must have a pointer to the last "fully unload-reloaded" record in KEYPOINT for each key.
  The effort is minimal, timesavings are high and results can be great: users are often consulting the most recent history and a "full reload" with a few "partial reloads" gives nearly the same effect as a set of "full reloads".
  This concept of "partial reloads" can also be applied to "normal" IMAGE-datasets. It's a pitty that such a smart reload is not yet available.

- By maintaining in the data-set KEY the last record in KEYPOINT which is "fully unloaded-reloaded" another improvement can be done in speeding up consulting: If the given startdate is less than the date given by the last "unloaded-reloaded" record, a binary search can be done in KEYPOINT between the first record for that key and the last "unloaded-reloaded" record of it.

- For 1 of the 3 keys (DOCUMENTNR) a different approach was made because the number of entries was rather small (1 to 100) and all records were always introduced in one single transaction. (This is in fact another "weak" point).
  Only a pointer to the first and last record are provided.

2.3.2 Data-structure for "current" information

Considering "weak points" 3 and 4, a separate dataset (DETAILCUR) is used which holds all outstandings yet in DETAILHIST and all input of the day.



- For each key the following pointers are provided in the master KE

  - first entry in this dataset for the given key
  - last entry in this dataset for the given key
  - last entry in this dataset for the given key that is already present in DETAILHIST. (lasthistoryponter)
  - last entry in this dataset for the given key that was "unloaded-reloaded"

  In addition a pointer to the last "used" record, the last "fully unloaded-reloaded" and the last record already in DETAILHIST of this dataset itself must be kept aside.
  In the dataset DETAILCUR all entries belonging to the same key are linked together with forward pointers.

  Here we don't need the KEYPOINT-dataset because:

  - most of the time all outstandings are asked (no selection on date)
  - the number of entries is rather small and and the dataset itself can be "unloaded-reloaded" for the most used path (CUSTOMER) in a rather small amount of time. The trick of a "partial unloadreload" can also be used.

- Consulting of "outstandings" can be done in the same manner as explicated above for the KEYPOINT dataset. The trick of consulting with the binary search mechanism only works for the path to which reorganizing was done!!!!

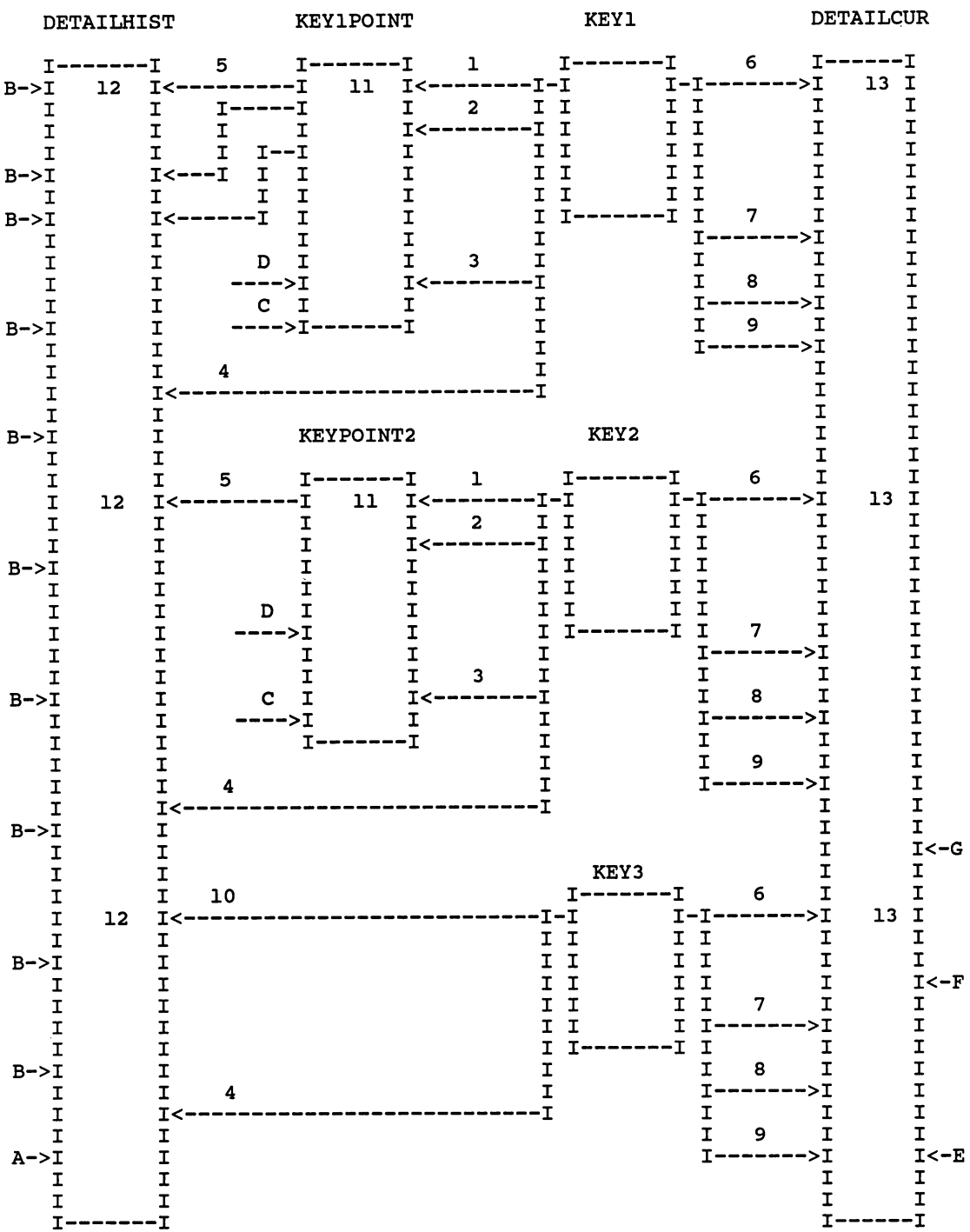
- All input is done to DETAILCUR-dataset, even if it's not "outstanding".
  All those records receive a special code and will be skipped when consulting the outstandings.
  No input is done in the DETAILHIST-dataset. When consulting the history for a given key, one starts reading in the DETAILHIST-dataset untill the end of the chain is reached, reads the record in DETAILCUR indicated by the "lasthistorypointer", and continues reading the next record of the chain.

- Speeding up registration of a booking is done as follows:

  - the pointers for a key are only updated if the key changes in the next record. For most of the bookings our dataset now acted as a detaildataset with only one key.
  - no pointers are created if a key is omitted (say zero or blank).
  - only "DBUPDATE"'s are done instead of DBPUT's.
  - registration is only done in 1 dataset instead of 2.

- Input of the day is posted to the DETAILHIST-dataset by a batch-job that runs overnight. When no "reorganisation" is done the job can start at the record just after the one indicated by the "lasthistorypointer" of this dataset.
  All modifications to already existant records in DETAILCUR are also redone in the DETAILHIST-dataset. Because modifications mostly exist of "payments of outstanding records" and those records are linked together with another "chain", one can easily retrieve the modified records. Other modifications are tracked by writing away the documentnr in a separate dataset.

- The complete system now becomes:



1: A pointer to the first record in KEYPOINT.
In KEYPOINT all records of the same key are linked together and are always sorted on date. This is done by the overnight batchjob.

2: A pointer to the last "fully unloaded-reloaded" record in KEYPOINT.
This pointer is maintained by the "fully unloadreload" program.

3: A pointer to the last record in KEYPOINT.
A record in KEYPOINT is written if :

- the current pointer = 0 (no entries in history yet for key)

OR

- the number of records written in DETAILHIST since the last record in KEYPOINT exceeds a certain value (both to be registrated in dataset KEY) and the date differs from the date of the last record in DETAILHIST.

4: A pointer to the last record in DETAILHIST.
The pointer is maintained by the overnightjob.

5: For each combination of KEY and DATE : a pointer to the first record in DETAILHIST. Written by the overnightjob.

6: A pointer to the first record of the key in DETAILCUR. Updated by the on-line programs and by the "reorganisation"-job of DETAILCUR.

7: A pointer to the last record of the key in DETAILCUR that is already available in DETAILHIST. Updated by the overnightjob.

8: A pointer to the last record of the key that is "fully unloaded-reloaded".
Updated by the "reorganisationjob".

9: A pointer to the last record of the key in DETAILCUR. This pointer is updated by the on-line programs and by the "reorganisationjob".

10: A pointer to the first record of the key (DOCUMENTNR) in DETAILHIST.
Updated by the overnightjob.

11: In KEYPOINT are all records with the same key linked together.

12: In DETAILHIST are all records with the same key linked together.

13: In DETAILCUR are all records with the same key linked together.

A: Pointer to the last used record in DETAILHIST.

B: Pointer to the first record of a given date in DETAILHIST

C: Pointer to the last used record in KEYPOINT.

D: Pointer to the last "fully unloaded-reloaded" record of KEYPOINT.

E: Pointer to the last used record in DETAILCUR.

F: Pointer to the last record of DETAILCUR that is also in DETAILHIST.

G: Pointer to the last "fully unloaded-reloaded" record of DETAILCUR.

2.3.3 Speeding up complex inquiries

To allow fast processing of the jobs of the (small) important department and to solve their complex inquiries the following modifications were done on the above design:

- the complete structure is made in double : one for the important department and one for all the others. This reduces the capacity of all datasets for that department by 3000 % (compared with the other departments).
  The result on the elapsed time of jobs that have to do a serial read is in the same order.

- a separate small file is builded (by a now fast running job) which contains the "outstandings" in a complex presentation-layout. The job runs overnight and at the quiet periods in the day.
  A separate masterdataset is builded which contains pointers to the file for each key and a pointer to the last record of DETAILCUR that is already in the file. When consulting is done on that file, all records of the given key are given, and eventualy additional records in DETAILCUR complete the overview.
  Whenever the jobs runs during the day, records are always added to the file and the appropriate pointers are changed after adding all records to the file. Consulting stays possible when the job is running!!!
  The EOF of the file is reset to zero when the jobs starts overnight.
  A file was chosen because:

  - writing in it goes faster than writing in a database
  - all info is safely placed in the database
  - all info can be reconstructed by rerunning the job
  - reading in it goes faster than reading in a database

2.3.4 Large interruptable transactions

The registration of the complex payments (update of over 1500 entries in a single transaction with the possibility to "undo" it within the same working day was handled in the following way:

- A (central) bitmap for the DETAILCUR-dataset is created. Each bit in it corresponds with an (outstanding) record in DETAILCUR with a relative recordnumber equal to the bitnumber in the bitmap. When the bit = 0, the corresponding record is "free" for payment, otherwise it is "used".

- When the user starts the input of a certain payment, a copy of the bitmap is taken into an XDS (local bitmap).
  When payment references to an already registrated document, the records of it are read in DETAILCUR. If the corresponding bit (in the local bitmap) is zero, one can go on, otherwise the document is already used in another transaction.

The recordnumber of each line in the workfile or scroll is added to the line itself.
No bits are set in the local bitmap.

- When a user terminates a transaction (i.e. payment of a customer) a module is called to set the appropriate bits in the central bitmap.
(His workfile or scroll will be scanned to gather the right bitnumbers). The results of this module can be:

  - Everything ok: the transaction of the user can now (temporarily) be stored outside the database in a normal MPE-file.
Further payments on referenced documents in it, are now disabled by the bitmap.
1500 or more DBUPDATE's are changed to a few discaccesses to UPDATE the bitmap!!!!

  - Double usage of bits : the user entered two times the same document. The workfile or scroll is scanned to retrieve them (relative recordnumber) and a message is given to the user.

  - A bit changed to 1 : another user referenced the same record before this user could registrate his input.

- When a user deletes a transaction the same module is called to reset the referenced bits in the central bitmap to zero. The appropriate records are again free.
If some of the corresponding bits are already zero, the current bitmap is "corrupt", and must be recovered starting from the MPE-files (the user-input).

- When a user wants to change a transaction, the same is done when deleting one (but the user-info is not cleared to allow modifications).
When registrating the modified transaction, the bits are resetted
- After validating the complete input (eventually after hours, or even at the end of the day), a job runs to post the user-input in the MPEfiles into the database. This can be done at the quiet periods or even overnight.

2.3.5 Query

The use of querylanguages was rather easy to imply:

- Because all data is located in IMAGE/3000 databases, reporting itself imposes no problem, once the records are retrieved.

- Modules were build to "localate" records in DETAILCUR and DETAILHIST, using the available pointers in the datasets KEY, KEYPOINT, DETAILCUR and DETAILHIST. Those modules build a "selectfile" that holds the relative recordnumbers of the retrieved entries and is used as input for the query-reporter.

3. "Various" tricks

3.1 Speeding up registration in a detaildataset

A lot of transactions work on detaildatasets in the following way:

- If there are entries available in the detaildataset for a given key, they are shown to the user.
- The user can add, delete and insert lines or change existing lines.
- Registration in the database.

Because inserting and deleting records in a chain is a rather difficult thing to achieve within IMAGE, the easiest programming way for this problem is deleting all existing entries and then writing the new records.

However, as long as the keyvalues remain the same the thing can easily speeded up in the following way:

```
clear flag fend
<findfrom,db="BASE",ds="SET",di="ITEM",from="ITEMVALUE">
if $image = 17 then set flag fend
   <getchain,db="BASE",ds="SET",di="ITEM">
   if $image = 15 then set flag fend
   while info available in user workfile or scroll
      move values from workfile or scroll to buffer
      if flag fend
         <putdb,db="BASE",ds="SET">
      else
         <updatedb,db="BASE",ds="SET">
         <getchain,db="BASE",ds="SET",di="ITEM">
         if $image = 15 then set flag fend
      endif
   endwhile
endif
ifnot flag fend
   while $image = 0
   <deletedb,db="BASE",ds="SET">
   <getchain,db="BASE",ds="SET",di="ITEM">
endif
```

Changing information becomes now quicker than introducing new information!!

3.2 Locking in a PH-environment and use of MR-capabilities

To avoid "overlapping updates" on data, there are 2 possibilities:

--> METHOD 1 <--	--> METHOD 2 <--
transaction WITHOUT permanent lock	transaction WITH permanent lock
==================================	================================

- get data
- modification by user
- lock data
- re-get data
- if re-getted data = old data
'update'
else
appropriate action or
give message to user
endif
- unlock data
	- lock data
	- get data
	- modification by user
	- 'update' data
	- unlock data

The first method leaves the database free of "locks" until the update really takes place. It assures a minimum of lockingproblems (not avoiding deadlocks, but avoiding that a user has to wait for another one (who is entering his information or is drinking a cup of coffee while he has something locked that prevents the other user to registrate his data).
The disadvantage is that the processing can become difficult or even impossible.

Method 2 has the advantage that it is easy. It even gives no problems if locking can be done on ITEM-level. The problem arises when one has to lock one or more datasets or eventually the complete database. (Remark: locking on dataset-level is required when updating MASTER-datasets).
In a PH-environment where a process stays alive when activating another one (perhaps in in the middle of a transaction that has a lock on the database) this method can not be used and can cause deadlocks (because MR-capability must be used).

The problem can be solved by applying first a "soft lock" and later on a "hard lock". The "soft lock" takes a lock at item-level in a complete separate database. This database consists only of 1 standalone detail-dataset with just 1 field and has no records in it.
When a user wants to add, modify or delete certain data, a lock at itemlevel is taken on the dummy database. The lockdescriptor is composed as follows:

- descriptionidentifier (i.e. "CUSTOMERNR") - value (i.e. "62417")	--> "CUSTOMERNR62417"

Although the modifications itself require locking on several items, datasets or even databases, only one "logical lock" in the dummy database is required.

(Data of given CUSTOMER can be stored in several records in several datasets and even several databases).
"Logical or soft" locks should be defined for the whole application system, and all programs should use them in the following way:

```
- conditional lock on dummy database at ITEM-level.
- if lock is not granted
     user-access in application program changed to "READ" as
     long as user works on this data.
  else
     user is allowed to make his modifications
     lock data ("hard lock" as required by IMAGE)
     update data
     unlock data ("hard lock")
     unlock dummy database ("soft lock")
  endif
```

This system has two advantages:

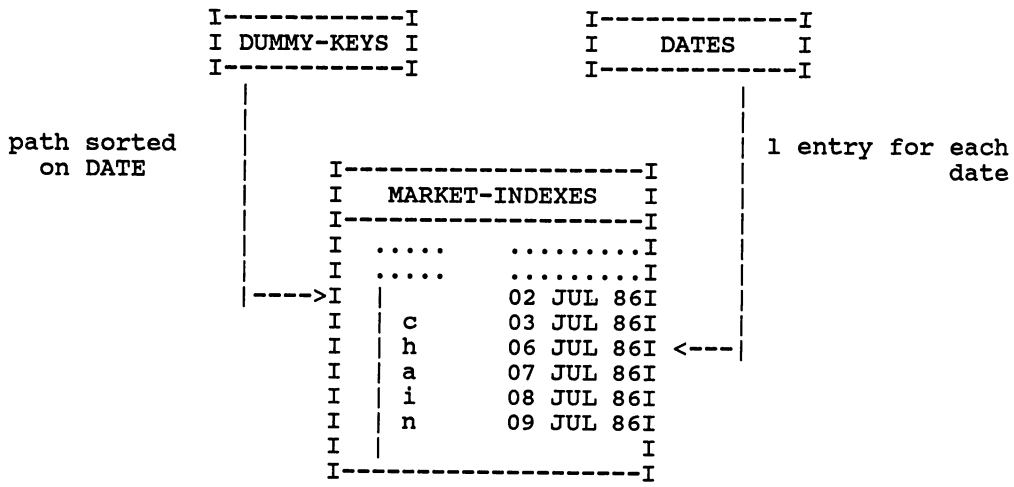
- only specific "working data" is locked, leaving "THE" database free of locks, so that waiting time for obtaining locks is minimalized.
- no complicate processing when doing the "real" update, so that the number of quiet periods on the database can stay at a good level.

Remark: as long as there are no transactions that lock the complete database, or work on more than on database, the "soft-locking" can be done on a dummy dataset of the database itself.

3.3 The use of dummy sorted paths

IMAGE lacks the capability to do an indexed look-up with the possiblity to retrieve the next or prior value for a given item. If all your keys follow a certain pattern (i.e. DATE, DOCUMENTNR ...) and if entries are most of the time entered in "good" sequence, the following approach can be taken:

Suppose a follow-up program for certain stockmarket indexes: each day all indexes are registrated and the program must give a quick overview of the evolution of them, starting with a date greater or equal to a given date:



DUMMY-KEYS is an automatic master with just one entry. The path to MARKET-INDEXES is sorted on DATE(YMD). DATE is also an automatic master.
All transactions simply work on the dataset MARKETINDEXES. Because most figures are inputted in sequencial order (the input are the figures of the day), there is little overhead due to the sortpath.

To localize the first entry and read the next entries

```
get in dataset DATES with the given date
while no record found
  compute next day
  get record in dataset DATES with computed date
endwhile
read corresponding entry in MARKET-INDEXES for path DATE
reread same entry in MARKET-INDEXES for path DUMMY-KEY
chained read in MARKET-INDEXES for path DUMMY-KEY
```

Remark: the application program has to check if the startdate falls in "a region of values" that is "supported" by the database. Otherwise, locating the first record can take too much time.

3.4 "Bulk"-handling of information

If one has to store a huge amount of userdata in one single transaction, and if re-screening of it must be done very fast, it is impossible to write all information record by record into a database.

However, if the records are compacted to one big record, the improvement can be astonishing ! Suppose you have a workfile or scroll that has to be saved, and which has 200 lines in it, each of 80 characters wide. The maximum entrysize in IMAGE is 2047 words = 4094

bytes. Up to 50 lines of your workfile or scroll will fit in such one big IMAGE-record, and the complete information can be stored in 4 IMAGE-records!
A very easy compression technique like compression of blanks will further reduce the amount of records with a factor 2 (a lot of data has blanks in it) and even reduce CPU-usage by avoiding extra IMAGE-calls. See further to 3.1 to increase speed when writing to a detaildataset.
Retrieving data goes at the same spectacular speed. At our site, storing and restoring information in this manner requires on a loaded machine less than 1 second for a "screen" with 200 lines.

The price that is paid, is that the data is now in its denormalized form. This can be overcome by writing the modified key-values to a message file.
A second process can now in background read the messagefile and reread the saved (denormalized) entries and write them to a second "CLONE"-database in its normalized form.
Programs that work on information "in the way" it was stored, can still work on the fast denormalized database. The others must work on the normalized one.

Biography

Jan Janssens is since 1980 system manager at Cobelfret N.V. - Antwerp, where he worked exclusively with HP3000-computers. He is a civil engineer of the KU-Leuven and followed MBA at the university of Ghent.