D A T A B A S E   R E F L E C T I O N S

Alfredo Rego and Stan Sieler

Adager
Apartado 248
Antigua
Guatemala

D a t a b a s e   r e f l e c t i o n s

Alfredo Rego and Stan Sieler

Adager
Apartado 248
Antigua
Guatemala

Leslie Keffer de Rego, Editor

Special thanks to Bob Green and Elizabeth Hanckel.

DATABASE! The word itself originates fierce battles: How do we SPELL it? How do we PRONOUNCE it? What does it MEAN?

Whether we like it or not, we all use databases in one way or another (or have some of our private data squirreled away in somebody else's databases). Inevitably, then, we are interested in understanding the current state of affairs.

As a first step, let us enjoy the rare pleasure of admiring a conceptual THEORETICAL IDEAL. As a second and sobering step, let us analyze the more pedestrian matters associated with IMPLEMENTATIONS OF DATABASE MANAGEMENT SYSTEMS. And, as a third step, let us review some of the PRACTICAL CONSIDERATIONS that we face in our dealings with ANY database.

To illustrate our discussion, we use a concrete database management system as our frame of reference. The bulk of our experience as members of the world-wide HP3000 family has been with Hewlett- Packard's IMAGE/3000 databases. Therefore, it seems reasonable to use IMAGE as our frame of reference and the HP3000 series of computers as our universe. IMAGE has been used for more than a decade in thousands of HP3000 computers throughout the world. It combines a solid design with a solid implementation. Many influential people realize its potential and IMAGE continues to be the subject of major investments in research and development. We can certainly learn a lot about database systems in general by carefully studying IMAGE in particular.

As a convenience, we will use "DBMS" instead of "Database management system."

THEORETICAL IDEAL: A database is a computerized model of reality.

An EFFECTIVE model should be, at least:

- Faithful.
- Descriptive.
- Predictive.
- Reliable.
- Efficient.
- Flexible, while being stable.
- Convenient.
- Economically justifiable.
- Secure.
- Easy to use.

To qualify as an effective model, a database usually consists of two complementary parts, supplied by a DBMS: the internal STRUCTURE and the USER INTERFACE. The structure is for PERFORMANCE's sake. The user interface is for SANITY's sake.

The structure as well as the user interface of a DBMS are no better than the operating system that supports them. Fortunately, Hewlett-Packard has managed to provide us with a stable operating system with more than 10 years of constant growth and constant preservation of compatibility. A user program that runs on any HP3000 computer hardware (large or small, old or new) will run on any newer HP3000 computer; and most user programs that run on newer HP3000 computers will also run on older HP3000 computers. The required changes, if any, are trivial to make. The HP3000's range of age, size, price and processing power is unprecedented in the computer industry. This is quite nice to know, since we are interested in a solid foundation for our DBMS.

Structure

The structure of a database is not "strictly" necessary, but it has dramatic effects on the performance of certain functions, such as FINDING and REPORTING.

Different people have different preferences when it comes to structures. Just witness the variety of tastes regarding buildings, airplanes, girlfriends, boyfriends, automobiles, etc. Database structures (or lack thereof) are not an exception, of course. Some people are satisfied with piles of junk while other people delight in crystal-like organizations where everything is disciplined to the utmost degree. Let us see some examples that will give us an idea of the RANGE OF STRUCTURAL POSSIBILITIES.

At one extreme, we see that a garbage dump (a "heap") does not have any internal organization or discipline whatsoever. We can easily and quickly add anything to it. However, we may spend an eternity to locate the specific item that interests us. Most likely, we will have to examine EVERYTHING. At the other extreme, we see that an ideal library would have a very large collection of books, a very sophisticated indexing system that would allow us to find any book by means of any attribute, and a very efficient staff that would NEVER place any book in the wrong shelf.

The same is true with databases. A system with great response time during data entry usually has an unacceptable response time during ad-hoc inquiries. Conversely, a system with great response time during inquires will probably have a slower response time during data entry, due to all the maintenance effort required by the underlying structure. And even the best possible implementation of a DBMS will eventually misplace some vital piece of information!

A DBMS without any internal organization whatsoever seems like a contradiction in terms. Even the "purest" of theoretical flat-table systems has some MINIMAL structure. Otherwise, we must do a full serial search through EVERY single bit of the database just to find one or two pieces of information. If we have intelligent peripherals that can do these serial searches blazingly fast, we can get away with a few thousand or a few million entries, but there is a limit to how many entries we can search serially in an effective way. A SEARCH WITHOUT INTERNAL STRUCTURE MEANS A SERIAL SCAN. This is unacceptable for on-line types of applications.

Regardless of our preference for structure (or lack thereof), the basic atomic entity in any digital computer system is the BIT. The word "bit" comes from collapsing the words "binary digit" and means "something which can be in one of two stable states". The "something" and the "two stable states" can be practically ANYTHING. These are just some of the many possibilities: on/off, right/left, up/down, positive/negative, forwards/backwards. Our choice depends more on convenience than on necessity. The absence or presence of a hole in an IBM punched card is an example of a bit. The direction of a microscopic magnetic field inside the HP3000's Intelligent Network Processor (INP) is another example of a bit. Despite their obvious differences in sophistication and technology, both types of bits are mathematically equivalent.

In general, we parcel out contiguous groups of bits and we call them nibbles, bytes, words, sectors, tracks, disc packs, memory banks, records, files, tape reels, diskettes, cassettes, punched cards, punched paper tape, rows, tables, columns, vectors, matrices, cylinders, segments, trees, lists, heaps, magnetic cards, drums, stacks, or whatever. The parceling, grouping and naming of these bunches of bits have been the result of arbitrary choices taken in the last few years by sublime scientists and mercenary promoters alike. It is VERY important to be aware of this, since some

people may think that these entities are sacred and worthy of our unquestioning acceptance.

These bunches-of-bits (whether we like them or not) cover a wide spectrum that ranges from the VERY inconvenient to the SOMEWHAT convenient. In general, we may consider them as different classes of computer memory. What is amazing is that we have been able to use these rudimentary computer-memory elements to model and represent practically everything known to humanity!

There are various ways to implement structures using these different hierarchies of computer memory. In general, faster memory is more expensive and more volatile. The trick is to try to optimize our cost/benefit ratios. CPU registers are the fastest, but a computer (ANY computer) has only a handful of them. Central memory is much slower, but a computer like the HP3000 can afford to have a few million central-memory locations (usually measured in terms of megabytes). Disc memory is orders of magnitude slower, but a computer like the HP3000 can have a few billion disc locations (usually measured in terms of gigabytes). Tape memory is dramatically slower but allows access to an unlimited amount of data, given enough time and human-operator patience. Since tape is the least expensive and the least volatile, it is mostly used for backing up databases (which generally reside on discs and are moved back and forth from discs to central memory for updating and inquiries).

Regardless of their physical location in the memory hierarchy, these computer-memory parcels contain information which is of interest to us. The challenge is to relate, somehow, those parcels that contain "related" pieces of information even though they may be physically quite apart from each other.

At this moment, let us take a small detour regarding fashion.

A casual cruising through the literature of the times might leave us with the impression that only the so-called "relational" systems can relate. Nothing could be more unrelated to the truth. Relating IS the whole idea behind ANY DBMS. This strange twisting of the language is analogous to what happened to the term "America".

America is the name of a CONTINENT that "begins" with Alaska and Canada in the north and "ends" with Argentina and Chile in the south. America IS NOT the name of a country. As a matter of fact, all kinds of countries, like Guatemala, Colombia, the United States of Brazil, the United States of Venezuela, the United States of Mexico and the United States of America, are scattered throughout this large continent. If you were born in Peru, then you are a Peruvian AND also an American; similarly, you are a Costa Rican AND an American, a Canadian AND an American, a U.S.A.'ian AND an American, and so on for every country in the American continent.

To say that ONLY natives of the U.S.A. are Americans is as much of a sacrilege as to say that ONLY "relational" systems relate. Every native of any of the twenty-odd countries of the American continent is an American. Every DBMS relates. Period.

Let us now return from our small detour regarding fashion.

We were saying that, regardless of their physical location in the memory hierarchy, these computer-memory parcels contain information which is of interest to us. The challenge is to relate, somehow, those parcels that contain "related" pieces of information even though they may be physically quite apart from each other.

We may relate at the time when we enter the information, or at the time when we report the information, or at any time in between. We may use techniques such as hashing and indexing to go (more or less directly) to the locations of parcels that contain master or key pieces of information. We may use techniques based on pointers to link certain parcels to other far-away or close-by parcels, usually based on common values for keys or search fields. And we can do the linking in a hierarchical way, in a networking way, or in any other way that we happen to like. (It is interesting to note that a hierarchy is only a special case of a network). Regardless of the structure, the linking may be in one direction (singly-linked lists), or in two directions (doubly-linked lists), or whatever, based on tradeoffs dictated by our priorities: space? performance? reliability? We can also choose NOT to do any linking whatsoever.

The issue of structure (or lack of structure) becomes important when we have to face reality. It is one thing to deal with academic examples that consider only a few hundred (if that many) entries. It is another ball game, altogether, to deal with operational systems that deal with billions of related pieces of information. We must remember that IMAGE currently supports up to 99 datasets per database with up to 127 fields (simple or compound) per dataset and up to 2,139,094,785 entries per dataset, for a maximum database field count of ... oops ... our HP calculator has just overflowed and has gone into scientific powers-of-ten notation! Not to mention that IMAGE and MPE allow us to have many databases per computer and many computers per network, linked via satellites hovering high above the surface of the Earth!

Does anybody still claim to be able to manage all this without the benefit of structure?

User Interface

The USER INTERFACE of a database system is responsible for helping authorized human beings perform (a) addition, modification, deletion and reporting of information stored by means of the database STRUCTURE, as well as (b) general maintenance of the structure itself.

A DBMS must have, at the very least, a set of intrinsics (fundamental procedures, routines or functions) that allow basic operations on the database's structure and data. We can then write programs that call, in turn, these intrinsics. In the specific case of the HP3000 computer, we can access IMAGE/3000 databases by means of intrinsics such as DBOPEN, DBGET, DBUPDATE, and so on, which we can call from programs written in languages such as BASIC, COBOL, FORTRAN, PASCAL, RPG and SPL.

The user interface consists of the set of programs that call the intrinsics. Some of these programs will be specially tailored for certain applications and some will be generalized. Some of the generalized ones may, in turn, be interpretive in nature. Some will provide only data entry, others will provide only reporting or browsing, still others may provide every kind of access to a database. Some of the interface programs may have rigid syntax rules and some may have extremely friendly approaches. Some may guide the user by means of examples and some may take examples from the user as sufficient "commands" to do database operations. Some may allow a limited number of relations and others may allow a large number of relations and operations.

The user interface drives the database structure just as a concert pianist plays a concert grand piano. The artist and the instrument must be perfectly matched and balanced. Otherwise, everybody suffers through a painful performance!

Theoretical conclusion:

If we imagine "data" and "information" as members of a "population", we can define a DBMS by analogy with a nation: "A protected territory with well-guarded borders and well-disciplined citizens who live in a climate of stability, mutual respect and cooperation". In IMAGE's terms, we get "A structured collection of protected (privileged) files which interact with the operating system in such a way that our separate pieces of information are safely kept and meaningfully related when we ask for them."

Wow! This sounds nice (or terrifying, according to your beliefs on how society should -- or should not -- be organized). Regardless of your reaction, the fact is that pieces of information, just like people, are unpredictable. Any attempt to organize them neatly into "perfect countries" is doomed to fail. We must be aware, then, that we are dealing with structures created and maintained by humans and, like every human thing, these structures will be full of errors, omissions, sins, blind alleys, blatant loopholes, wasteful use of scarce resources, injustices, and so on. Quite normal, we think, if we consider that there are infinite implementation choices, with each of these choices being followed, in turn, by an infinite amount of other sets of infinite choices! Too much, really, for the human mind.

Our theoretical conclusion then is: "The different DBMS's are only different in terms of the implementation choices made by their designers. Deep down inside, all DBMS's are the same: They are just frail human attempts to organize unruly pieces of information."

This theoretical conclusion serves as our guiding light as we begin our study of practical realities: The implementations of DBMS's.

==

PRACTICAL REALITY: A database is a frail human attempt to organize unruly pieces of information.

==

Frail or not, databases should put up a minimum battle to be considered worthy contenders. Let us examine some tricks of the trade that we consider standard for such battles.

Availability of a minimum set of access possibilities

We deal with a database for two main reasons: to do general maintenance work on the STRUCTURE and to do general maintenance work on the DATA. We access the structure by means of the DBMS's utilities (like DBUTIL and ADAGER). We access the data by means of the DBMS's intrinsics (like DBUPDATE).

We can access a database's data in an updating or a browsing mode. In updating mode, we can add, delete and modify entries. In browsing mode, we can only look at existing entries. In either mode, we are restricted by the database's security layout, as specified by the database's design and maintenance staff. For instance, in a payroll database some people may not be allowed to modify or browse the salary fields.

We have the following possible kinds of access to a database's data:

- Exclusive update, so that nobody else accesses the database when we update it. This is not very sociable and we should only use this access mode under extreme circumstances.
- Non-exclusive update. In this mode, we can still choose whether or not we allow concurrent updating or concurrent browsing or a mixture of both.
- Exclusive browsing, so nobody else accesses the database while we read its entries.
- Non-exclusive browsing. In this mode, we can choose whether or not we allow concurrent updating or concurrent browsing or a mixture of both while we read the database's entries.

IMAGE allows several access methods, including:

- SERIAL (forwards or backwards), according to the physical location of the entries regardless of their search-field values. We access the NEXT or the PREVIOUS physical entry.
- CHAINED DETAIL (forwards or backwards), according to a chain of common detail entries with the same search-field values even though the involved entries may be quite apart physically. We access the NEXT or the PREVIOUS logical entry (which may be located close by or thousands of PHYSICAL entries away from the current entry).
- CHAINED MASTER (forwards or backwards), according to a chain of common master entries with the same search-field hash-value even though the involved entries may be quite apart physically. We access the NEXT or the PREVIOUS logical entry (which may be located close by or thousands of PHYSICAL entries away from the current entry).
- HASHED, according to the value of a master search-field. With hashed access, we specify "give me the entry that corresponds to Gonzalez" (without any mention of the LOCATION of Gonzalez's entry; as a matter of fact, such an entry could very well be the most active migrating secondary in our database!)
- DIRECTED, according to the entry number. We access entry number 1541, or entry number 1978, or entry number 780903, regardless of its contents.

IMAGE gives us "location-addressing" mechanisms (serial and directed access methods) as well as "contents-addressing" mechanisms (hashed and chained access methods).

These are the individual access modes INTO the database. To orchestrate them, IMAGE and MPE (the operating system for the HP3000 family of computers) allow us:

- Concurrent access by different processes within the same time frame. Some processes may be updating, some adding, some deleting, some simply browsing, some doing everything.
- Access by the same process throughout various time frames, even as the database's structure evolves. This requires data-independent programming, which is slower and more painstaking but more flexible than hard-wired programming.
- Access by a mix of online and batch processes.
- Access by a mix of local and remote processes.
- User-defined granularity of access (i.e., the ability to zoom in on what interests us: no more and no less). Shall we examine one or more databases? one or more datasets? one or more entries? one or more fields? Shall we deal exclusively with real entries or shall we construct virtual entries that project certain fields from certain datasets in certain databases and then join certain subsets in a Boolean nightmare? The topic of granularity is very important for intelligent tradeoffs among criteria such as access, privacy, security, concurrency, locking, and performance.

Reporting

All these mind-boggling combinations of access modes have one ultimate goal: to allow us, eventually, the ability to report the information contained by the data that we maintain in the database. This eventual reporting must, in turn, be efficient and convenient.

Reporting, of course, does not have to be limited to the boring kind. Piles of computer printouts listed on 132-column paper are not the only category of "allowed" reporting. With the existence of plotters, laser printers, graphics terminals, voice devices for input and output of information, etc., the range of reporting capabilities for IMAGE and the HP3000 computer is practically unlimited.

The function of REPORTING may very well be the most interesting one, from a user's viewpoint. We may actually assemble totally new, "virtual" entries by shuffling, joining, projecting, including or excluding actual "physical" entries that we have stored in the database's structure. Unfortunately, if we want better performance during the reporting function, we may have to pay a high price and we may be forced to suffer an unacceptable level of performance during other functions like addition or deletion of entries. When interested parties tell us about their wonderful reporting capabilities, we may want to ask them "How long does this jewel take if we have 95 different files with 2 billion entries each that have to be joined, projected and then massaged through some arithmetical number crunching?"

There is no such thing as a free report!

Flexibility, in the face of stability

As circumstances change quickly and unpredictably, we must be able to adapt our database (both its structure and its user interface) to keep up with the reality it is supposed to model. However, just as we want to be adaptable and flexible, we also want to have stability. This means that we MUST preserve the meaning of the information we already have in our database.

If we design our databases and procedures with data independence in mind, we may very well discover that a change in our database requires only minor changes (if any) to our existing procedures. More importantly, we will be able to develop new procedures without having to do any maintenance work WHATSOEVER on current ones. In IMAGE, for instance, we declare our independence by means of field lists and calls to DBINFO for the resolution of run-time definitions.

Some DBMS's require us to determine the logical structure of our database ONCE, at the beginning and forever. This is hopelessly impractical, since God only knows what will happen in the future. With IMAGE ands its utilities (especially ADAGER), we are able to change the practical implementations of our designs and we do not have to pay outrageous prices just because we were human and could not foresee absolutely everything when we began our database projects.

Optimization of Performance

Performance is a relative term. For instance, raw performance in the addition of new entries is totally meaningless if the performance of finding those entries, later on, is pitiful. Even a system with excellent overall performance in storage and retrieval may still be unacceptable if the human effort required to learn to use it turns out to be of heroic proportions. For a DBMS, we must be aware of the tradeoffs in performance for these functions:

- Entering data.
- Finding entries that meet a certain set of criteria.
- Reporting the entries found.
- Deleting entries.
- Modifying entries.
- Adding entries.
- Storage requirements for data and structural information.
- Recovering in case of computer, media or site failure.
- Reorganizing the database structure.
- Backing up and restoring to/from offline media (like tapes).
- Logging and auditing.
- Designing the database.
- Implementing the database.
- Revising the database structure.
- Concurrency control.
- People's interaction with the database (gruesome or friendly?)
- Security & privacy of data.
- Training.
- General baby-sitting required by the database.

Minimization of expenditures

A DBMS should consume a minimum of resources to produce a maximum of legitimate access to its data.

IMAGE and MPE use central memory for run-time control blocks, tables, buffers and execute-only code segments. IMAGE and MPE use disc memory for long-term tables and structures. MPE uses disc memory for "virtual memory" (temporary storage of run-time data which needs to be swapped out of central memory).

It is interesting to note that a database may have a simple structure but a lot of data entries, thereby occupying little central memory for run-time control blocks but huge areas of disc for data. Also, a database may have a very complex structure but few data entries, thereby occupying large areas of central memory for run-time control blocks but small amounts of disc for data.

We can think of other computers, linked to a given computer through a local or remote network like Hewlett-Packard's Distributed Systems (DS), as another kind of "memory", since we can send data to remote computers and request data from them.

It is fascinating to see how the most economical solution to a problem can also be the most elegant and fruitful. In the specific case of the HP3000 and IMAGE, the minimization of central-memory expenditure turns out to aid tremendously in maximizing performance, concurrency, security and consistency. By sharing central-memory buffer pools in the database control block, IMAGE centralizes all data transfers between central memory and disc, between central memories in the case of networked computers, and within a given central memory in the case of local processes which access the same database concurrently. This centralization of control allows IMAGE to guarantee the correctness of important functions such as logging, recovery, auditing, locking, etc.

Protection, Security, & Privacy

Privileged protection

In the good old days of vacuum tubes, every location accessible to the computer was accessible to anybody. With the advent of higher-level architectures, operating systems and languages, some locations in main memory as well as in auxiliary storage were taken away from the standard user. These areas were reserved for the operating system and were baptized "privileged".

The enforcement of privileged protection is a MANAGEMENT CHOICE. Some "higher up" human decides which accounts, groups and users are given the privileged-mode capability. Since the internal tables of an operating system or a DBMS are so complex and sensitive, standard users should not be allowed to access them directly. In the HP3000 computer, both software and hardware mechanisms make sure that a user-mode process accesses privileged

areas only by means of a management-approved intermediary. In the case of IMAGE, these approved intermediaries are the IMAGE intrinsics and the IMAGE utilities (like DBUTIL and ADAGER).

Passwords

Passwords are the minimum acceptable kind of filter to control access to sensitive information. IMAGE uses table-driven passwords. To make them more tamper-proof, IMAGE allows combinations of upper and lower case letters, special characters, escape sequences, and a host of non-printing ASCII bit patterns.

Logging (for both recovery and auditing purposes)

IMAGE controls logging globally, by database and not by user. We can use IMAGE utilities (standard MPE as well as DBAUDIT) to analyze the log files on disc, tape, or other serial media. We can detect suspicious activity, both in terms of specific accesses to our databases and in terms of specific accesses by given users by means of given terminals (local or remote) or of given batch jobs. The HP3000 operating system also keeps track of sessions and jobs, together with their logon hardware devices and any mounting or dismounting of tapes for backup purposes. By combining these checks with strict discipline in the computer room, we may prevent the loss of database privacy by means of unauthorized copying to removable media or by means of unauthorized migration to remote computers through the distributed-systems network.

Intimate cooperation with the operating system

There are literally thousands of cooperating protocols between a DBMS and the computer's operating system. Some of these protocols are little, some are big; some are obvious and some are extremely subtle. Some are documented in the user manuals and some are undocumented and uncallable by non-privileged processes. Let us examine a couple of such protocols just to taste the general flavor.

Whenever an IMAGE intrinsic operates, it marks the calling process as CRITICAL. A critical MPE process cannot be aborted externally (through the :ABORT or :ABORTJOB commands) and if the process itself initiates an abort due to its own errors, MPE causes a system failure. IMAGE uses this feature to protect the integrity of the database, since any IMAGE intrinsic will be able to proceed uninterrupted until its logical conclusion unless it happens to be blown up by an "independent" system failure or by a failure in the intrinsic's own code or in the database's tables).

To serialize sensitive operations, IMAGE issues a request for the IMAGE SIR before it opens or closes a database on behalf of a given process. We can think of a SIR as an operating-system-wide lock upon a certain resource. In the case of the IMAGE SIR, this certain resource is the ability to open or close IMAGE databases. When it obtains the lock, IMAGE knows that no other processes are in the midst of opening or closing a database ANYWHERE in the system. Hence, IMAGE can proceed with the business at hand, safe from confusion by other processes.

Reliability

A DBMS is welcome to have all the bells and whistles that its creators desire. However, if it is not reliable, it is not worth using at all. Reliability, for us, means AVAILABILITY and, in the event of unavailability, effective RECOVERABILITY.

Availability

Availability (the percentage and contiguity of UP time) must be very high. The percentage and contiguity of DOWN time must be very low. The HP3000 computer in general, and the IMAGE DBMS in particular, may very well hold the world's record in the availability category.

Recoverability

Any of the components of a working database system may fail:

- CPU.
- Memory.
- I/O channels.
- Mass storage devices.
- Offline (backup) media.
- Site (fire, flood, etc.)
- Communications lines (for remote access to a centralized database or for remote access among various databases in the case of a computer network).
- Operating system.
- DBMS software, hardware, or firmware.
- Applications software.
- Malicious (or innocent but equally catastrophic) misuse of QUERY languages.

A robust DBMS must be able to recover from any and all of these kinds of failures. Webster's Dictionary defines "ROBUST" as "Having or exhibiting strength or vigorous health: POWERFUL, MUSCULAR, VIGOROUS. Firm and assured in purpose, opinion, or outlook. Exceptionally sound. Strongly formed or constructed: STURDY. Requiring strength or vigor. FULL-BODIED, STRONG. Synonym: see HEALTHY."

IMAGE is certainly a robust DBMS: it is able to function even after having suffered considerable structural damage and it offers several alternatives (some standard and some optional) for recovering back to a consistent state, including:

- Intrinsic Level Recovery (ILR).
- Transaction logging & DBRECOV.
- DBSTORE & DBRESTOR.
- DBUNLOAD & DBLOAD (both chained and serial).
- On-line reconstruction (ADAGER).

As with everything else in life, there is no such thing as a free lunch. The price you pay for robustness includes certain degradation of performance and periodic lack of availability while you do your robustness-oriented tasks. But it all becomes worthwhile the day when you really need to recover from a bad catastrophe.

Concurrency

The challenge of allowing concurrent access to a database includes these criteria:

- Giving access to as many users and processes as possible, in a mix of batch/online, local/remote, update/browse.
- Minimizing the waiting time for each accessor.
- Maximizing the protection of the database (from malicious users, from bona-fide users that enter incorrect data or transform/delete valuable data, etc.)
- Minimizing the total impact on other users and processes, not associated with the concurrently-accessed database.
- Scheduling database-maintenance activities in such a way that their impact on concurrent users is minimal.
- Eliminating or minimizing deadlock situations, when two processes wait forever for each other.

In the specific case of the HP3000 computer, MPE and IMAGE implement concurrency in several ways, some controlled by user processes and some controlled by the database and operating-system processes.

The most obvious way is the mechanism of LOCKING. IMAGE allows temporary locks at the database, dataset and data entry levels. These locks are based on predicates kept in the single database control block, resident in memory and shared by all accessors of a given database. Since predicate locking is LOGICAL and not PHYSICAL in nature, it is extremely efficient. Remember that IMAGE provides only the locking mechanisms. It is up to the user to take advantage of these locking mechanisms in an intelligent way to reach an acceptable level of performance. The operating system will guarantee deadlock-free operation for users without multiple-rin (MR) capability. However, the operating system will let users with MR capability lock as many databases in as many ways as they wish, with the understanding that deadlocks may arise and the user, then, will be responsible for restarting the whole system.

A more subtle way to implement concurrency is to share some of the internal resources of the computer, of the operating system and of the DBMS, both in hardware and in software. Let us look at a couple of examples to get an idea of these techniques, as implemented in the HP3000.

The HP3000 computer defines all executable code as shared and re-entrant. Only one copy of the IMAGE code exists in the computer, regardless of the number of databases accessed or the number of database accessors. When the memory-resident space occupied by a part of the IMAGE code needs to be used by some other code segment, the memory manager simply overwrites the IMAGE code. When the operating system needs that "destroyed" part of the IMAGE code, the memory manager simply copies it back from disc, usually recycling the memory space that is no longer used by some other code segment. Note that code segments NEVER need to be written to disc, since they are not modified at all. What is true for the IMAGE code is true for any other code (systems or applications) in the computer.

The data segments assigned to a process by the HP3000 (run-time stack plus extra data segments) are normally private and controlled by each individual process. Certain extra data segments can be defined as sharable among members of the same process family or among completely unrelated processes, as is the case with IMAGE's run-time control blocks. The fact that IMAGE's control blocks are shared by all accessors to the same database turns out to be very convenient for concurrency, since hierarchical locking and enqueue/dequeue logic handle conflicting requests.

Ease of use

If getting all the previous benefits means getting through a maze of incomprehensible gibberish, nobody is going to use (willingly) any DBMS.

IMAGE is implemented by means of data structures that are simple, well-defined and easy-to-understand. Its atomic user interface (upon which all kinds of friendly user interfaces can be built) is also simple and consists of a handful of intrinsics. These intrinsics have standard calling sequences that make them accessible to ANY language in the HP3000. When things go wrong (either in the database itself or in the calling sequence), these intrinsics return meaningful messages and status flags that distinguish unexpected conditions from erroneous conditions. IMAGE has straight-forward utilities to cover all aspects of database maintenance and tuning.

Ok. So far, so good. Database management systems certainly look like very cost-effective methods to manage information. But how about our PARTICULAR reality? How do we model it using a DBMS?

PRACTICAL CONSIDERATIONS

Regardless of the specific DBMS that we may have chosen to model our reality, we have to do certain standard tasks to design and implement a database. Let's review some of these tasks and some of their possible pitfalls.

- Identify those aspects of reality that will be modeled by the database. Remember that only a small part of reality can be mirrored by a database. It turns out that defining WHAT to model, HOW to model it and WHY we want to model it are more difficult than we think. Therefore, we must be prepared to REVIEW and CHANGE our opinions (and implementations) throughout the life of our database.

- Make some notes of the relationships that exist among the entities to be modeled. Different folks have different forms. Some are normal forms and some are abnormal forms. It does not matter. Most likely, what we want to model does not fit into somebody else's pet theories anyway! The important question is: "Can we define, redefine or cancel these relationships at any time during the life of the database?" For performance reasons, we may want to specify that some OBVIOUS relationships be "hard-wired" in the database's structure (for instance, by means of PATHS in IMAGE). But we do not want to be stuck for life!

- Translate these informal notes into some kind of formal language that we understand and that the computer understands too. We want to be able to translate our INITIAL notes as well as the notes that we develop DURING the life of our database. We do not want this to be a one-time affair!

- Parcel out the computer storage required to build the database's structures. Some DBMS's require the user to specify the areas of storage in sickening detail, down to the cylinder, track and sector level, thereby ensuring the survival of operating system gurus. This was OK in the 1950's but we are adults now! The modern combination of IMAGE and ADAGER reserves and formats ALL disc space at database-creation time, automatically and in a fool-proof manner. At any time during the life of an IMAGE database, we can physically move specific datasets from one disc to another for performance fine-tuning, if we so desire. And we can easily change the capacities of the datasets to keep up with reality.

- Add, delete, modify and report entries. Fine tune things in such a way that we reach a reasonable compromise between the RESPONSE TIME for any of these functions and the global THROUGHPUT for the whole transaction load.

- Baby-sit the database through its daily (and nightly!) maintenance needs. Some DBMS's assume that only the DATA will change and make no provisions for dynamic changes to the STRUCTURE. IMAGE and ADAGER allow us to do all kinds of changes to the data AND to the structure while preserving the meaning of the information, at any time during the life of the database.

None of these tasks is easy, and each task is essential if we want to maintain the mutual reflections of REALITY and DATABASE. It is the DBMS's responsibility, as our tool, to make sure that these tasks are PRACTICAL IMPLEMENTATIONS of SOUND THEORETICAL IDEALS.

Our reflections have considered IMAGE/3000 as our frame of reference. We look forward to hearing from you regarding your reflections and your frames of reference. Thank you.