# ADDING DATABASE MANAGEMENT TO ADA[R]

Patrick A.V. Hall, System Designers Int. Ltd, 105 Fleet Road, Fleet, Hants, England.

## 1. INTRODUCTION

Systems developed using Ada will clearly need to store complex data over long periods. There is a need for some database management capability. The traditional approach has been to develop seperate packages for use alongside the main programming language (eg DATE 75, CODA 71). That is not the way forward.

Programming languages and database management must come together. There have been some attempts to define database facilities as extensions of existing programming languages. These attempts have either been general explorations of abstract data types (LOCK 79), or been addition of data types to existing languages (SCHM 77, SCHM 79, AMBL 79, ALAG 81), notably Pascal. This has also been done for Ada (SMIT 81), adding to the language the functional data model capabilities of DAPLEX (Ship 81). But is this really necessary?

Atkinson, Chisholm and Cocksholt have pointed the way with their PS-Algol (ATKI 82). I had also been working on these lines from a base of POP.2, Pascal and PL/1, asking myself why, when the programming language already had a rich capability for storing and manipulating data using records and pointers, do you need a database management facility? Why can't the heap be simply paged off into disc? Isn't this the way to do it in Ada?



The advantages of this approach are threefold, all economic

- people need learn only one set of facilities
- storage of only one set of run-time software is necessary
- speed should increase with no need for dynamic translation of data.

This paper outlines the approach required for Ada. Ada will be viewed as frozen in its form defined in the reference manual (LEDG 81, ALSY 83) though it should be noted that Ada has in fact undergone (small) change while undergoing standardisation, and more substantial changes have been proposed (LEDG 82, SKEL 82). Note that the ADAPLEX approach (SMIT 81) violates the frozen nature of Ada.

The fundamental requirements for database implementations will be discussed in successive sections, and it will be shown how the requirement can be met within Ada as it is currently defined.

## 2. ADEQUACY FOR DATA MODELLING

In this section I will argue the adequacy of the data declarations of Ada for data modelling by using the entity-relationship conceptual models of Chen (CHEN 76) implementing them using records and access types in Ada. The declaration of the records and associated types would constitute the Schema of database practice, and would naturally be embedded in a package.

* R Ada is a registered trademark of the U.S. Government Ada Joint Program Office.

The Chen conceptual model represents information about the world as entities and their attributes, together with relationships which link together entities. The relationships may themselves have attributes. The simplest implementation would be to have a record type for each entity type, representing the attributes of the entity as fields, with the relationship type also being a record type with two (or more) access-type fields to the entities that are party to the relationship, plus fields for the attributes.

All the essential properties are captured by this, but there are two problems. Firstly, accesses are undirectional, and in conventional processing of Ada this makes it impossible to traverse from one Entity record to another through a Relationship record. Two conventional solutions to this to add the bidirectionality would be to use rings, or pointers arrays. Many other strategies are possible (e.g. HALL 75) and may be neccessary depending upon restrictions in use of pointers in the language.

The second problem will be taken up again in the next section.

An alternative implementation strategy will become worth considering. This is to use "associative" linkages rather than the "physical" access mechanisms of the programming language. This approach directly corresponds to the entity relationship approach of Chen (CHEN 76) and myself and colleagues (HALL 76). Each Entity is given an invariant identifying "surrogate" preferably by the system, and relationships are pairs (or more) of surrogates. The associative link is directional in the sense that it requires an index to be built (see section 3) on the surrogate in the relationship record.

## 3. ASSOCIATIVE ACCESS

In some computing problems, the processing of data can start at one of a small number of starting points or "roots", and follow paths through the data from there. However commonly, and in particular in commercial data processing, no particular record from a set is preferred, and it is necessary to find a record "out of the blue" given the value(s) of one (or more) of its field(s). It is this requirement for associative access that appears to have led to the definition of extra types for PASCAL (e.g. SCHM 77).

The normal solution to this requirement is to build "indexes", nominating one or more fields in the record as "keys". In database management systems these indexes are an integral part of the system. We will have to program the indexes ourselves, hiding these from the applications processes by placing them inside the packages that manage the database: various tools could assist this, and performance controls (later section) should enable us to build effective indexes which exploit the inherent structure of backing storage devices.

There are a variety of well-established indexing techniques we could use (e.g. HALL 75). These could be programmed seperately for each index, but Ada has the facility for generic procedures and packages which should

enable us to program the indexes once and define new indexes as required. It would also make the schema, where the indexes are actually declared, more readable.

Several approaches to indexing are possible, depending upon how much is done within the indexing procedures.

A first method is to supply a set of procedures for

- creating a new record in the index given its key
- finding a record with a given key, setting a currency pointer to this record, and returning the pointer.
- getting the next record in sequence following an operation of finding a record.
- deleting a record from the index.

This package has limitations, since it only allows one field of the record to be indexed. It enforces uniqueness of key. Fields of the record can be updated using assignments, since the actual record in the database is returned. However this also allows the key field to be updated and there is no protection from this, the applications programs having to be trusted in this respect.

A second method would arrange the procedures to return a copy of the record. Taking this course has several consequences:

- fields can be changed using the normal Ada assignments, but must eventually be updated in the database by a REPLACE procedure or similar
- the REPLACE procedure can enforce that keys are not changed since it can exploit the redundancy embodied in the currency pointer
- pointers to other records cannot be followed directly, or else the real record will be found,and thus pointers cannot be returned with the records, and links must be accessed via suitable database procedures.

This last point suggests the use of associative linkages.

A third method is to completely hide the record through a set of procedures, one for accessing each field of the record, and one for updating each field. A particular record would be obtained, as before by a FIND operation which sets a currency pointer to the record, which is then processed. It is very straightforward to remove any ability to update the key field, and more general access rights can be tightly controlled as will be seen in section 6.

## 4. PERMANENCE OF DATE

A Database constitutes a permanent collection of data that must persist for months or years. In database management systems this is usually achieved by storage on non-volatile mangetic media like disc, treating the data as external to the computer.



However in Ada it is here proposed to make the database an integral part of the active programs, using the data declarations of the language.

In Ada the heap is global and not tied to a particular task or other process but nevertheless is tied to a collection of tasks or programs or packages. This means that the first step in Ada must be to make some task or process run forever, or a package persist forever.

Making processes run forever requires that the operating system or run-time system should recognise, in addition to the operator (or job control) commands for starting and stopping (aborting) a process, also commands for suspending and restarting a process. Suspending and aborting both require the facility to interrupt an active process, and suspending further requires the retention of the current state of execution of the interrupted process, retaining this in primary memory or backing it off onto secondary storage as appropriate. There is nothing particularly novel in these facilities, though their motivation may appear unusual. Consideration must also be given to hardware failure particularly if the primary storage is volatile. The principle necessary to secure oneself against hardware failure and recover from it are taken up again in section 8. Storage of the data on backing storage is not fundamental, though it is the usual first step in providing recovery.

Making the system run forever brings additional complications. It must be possible to amend the system, either for repairing bugs or for modifying and extending the system, without compromising the data. That means on-line changes to the software, not usual in normal systems. Nevertheless on-line changes are possible, and the run-time system of Ada must enable these.

## 5. DATA VOLUME

Databases in some applications like process control and computer aided design of electrical circuits are small enough to fit entirely within primary storage, but this is unusual. Usually databases are very large, and some form of backing storage is necessary.

Now this is the point at which it appears, historically, that the data of programming languages and the data of databases became disassociated. The ability to handle large volumes of data has traditionally been provided through i/o facilities to magnetic tape and disc.

The solution proposed here is to back the heap off into secondary storage, either through a virtual machine for the complete software, or through a virtual machine specialised to the heap. The important thing is that this would be entirely invisible in Ada. The use of i/o is not necessary.

In normal virtual machines there is still a severe limitation on data volume by virtue of the addressing limitations of the underlying hardware. But this link to the hardware for the pointers need not be maintained. Pointers of arbitrary range could be used. These pointers could be related to addresses on secondary storage, viewing the primary storage component of the heap as a cache or buffer pool. Alternatively the pointers could be purely logical, with a logical to physical conversion table being maintained. Logical pointers have extra attractions with compactifying garbage collection, and in recovery.

In Ada it may well be appropriate to make the data types subject to backing storage controllable. This is easily done through a pragma - already pragmas have been defined for declaring bounds on volumes to assist in storage allocation, and this would be a natural extension of that capability.

## 6. CONTROLLING SHARED ACCESS

Databases are commonly shared between several people or processes, and it is normal to restrict access to only the data required, to prevent either accidental or malicious use or modification of other data. Such restriction also limits the impact of change, a very important consideration in large systems. For the moment we are only considering static aspects of sharing, as if all processes sharing the data run sequentially. In section 7 the extra complexities of concurrent sharing will be addressed.

For processes and programs it is usual to restrict knowledge of the data available through a limited set of declarations. In conventional data management systems these are known as "subschema". We require that the complete set of declarations of the database are out of scope of the process, while the data itself is accessible, but only in so far as is permitted by the local declarations.

Ada is not really designed for subschema, though some way towards this should be possible through the use of packages to segment the data declarations. Unfortunately for interlinked datastructures, this strategy does not work. Ada avoids recursive definitions, and requires that where recursive definitions might be necessary, incomplete type declarations are used to enable the access type to have been declared before it is used within a record. Now of course incomplete type declarations have to be completed before objects of the type can be declared, but the Ada Reference manual requires that they are completed within the same package. However the reference manual is not clear on this matter, and we could quite reasonably have incomplete type declarations carried outside the package in which they are declared for completion in some other package. Ada compilers clearly could take this route without compromising the language elsewhere.

Nevertheless a form of subschema is possible using subtypes as an intermediary between the schema and its use.

The alternative approach to data modelling used associative linkages. The schema there is not interlinked, and clearly it can be segmented using packages.

We would like, however, to have more control over what functions particular programs or people are allowed to perform on the data. In addition to limiting exposure to a subset of records within the schema, we would like

- restriction on the fields visible
- restrictions to particular records by range of some field
- restriction by users to particular records individually identified
- restriction of type to access



All these controls for shared access could easily be enforced through a procedural interface between applications and database, as in the second and third methods of indexing in section 4. Each subschema would then offer just those parts of the data and operations that are desired.

## 7. CONCURRENT ACCESS

In section 6 we discussed shared access, but assumed sequential access if simultaneous access could cause problems. However, in most database applications concurrent access to the data is essential. A first requirement is clearly the ability to have several applications taking place at once, interacting with a shared database. The programming primitive to enable this is the "task" of Ada (LEDG 81). Thus in enabling concurrent access to a database, a first step must be to make the database and each seperately executing application process into seperate tasks. However, that is not the end of the story.

Concurrent accesses to data require controls to prevent interaction which could lead to unpredictable results. Recently a comprehensive survey was given by Kohler (KOHL 81), and a description of several subtle problems is given by Engles (ENGL 76).

The solution is to recognise that some sequences of operations must succeed or fail as a unit. These are the transactions of databases. Changes to database resources must not be visible until the transaction completes and 'commits' the changes (see also next section).

The usual implementation employs locks. However locks bring with them the new problem of deadlock. Locking is by no means the only method, but is so universally used that it is often thought to be the only method. Adding locks to the database in Ada can be done in one of two ways. Either locks can be explicitly programmed using extra fields or separate tables - or the already existing mutual exclusion methods associated with concurrent processes of the language can be exploited. If the former approach is taken then all requests for data must pass through a central control point for setting, checking, and unsetting of locks. One or more tasks could be used. The methods for handling such locks are well known (eg. KOHL 81). By contrast the latter approach, of exploiting the mutual exclusion mechanisms designed for processes, does appear novel. The mechanisms for communication between concurrent processes can lead to fairly complex programs, as evidenced by the examples of Welsh et al (WELS 81). The mechanisms of Ada have particular properties which, regrettably, make them unsuitable.

## 8. RECOVERING FROM ERRORS

Recovery from errors is an extremely important issue in computing, and has been covered in several recent survey articles (RAND 78, VERH 78, KOHL 81, GRAY 81). When there is a failure of part or all of the computer system, it is important that the data is not left in a corrupt or unpredictable state. A secondary consideration is that only a minimal amount of work should need to be repeated. Two classes of error must be accommodated: system errors and application errors.

System errors are those that are completely outside the control of any application process, such as hardware failures. For these errors the integrity of the data is usually assured by the taking of "back-ups" or "checkpoints" of the data at "tidy points" such as the end of day or end of week. In the event of failure, processing is restarted at the most recent checkpoint. For a finer grained restart capability, at the end of each complete processing step or "transaction" the updates can be dumped or "logged", so that to restart, the system returns to the start of the processing steps actually in operation at the time of failure.

Since system errors are outside the control of the application, recovery from these errors should be entirely invisible to the application. To make transaction logging in Ada possible we would need to identify those units of processing that are "transactions". At the end of the transaction the updates made during that block would be dumped to the transaction log. Suitable units of processing in Ada could be tasks, or even blocks between "begin" and "end" which would allow nested transactions.

Applications errors are those detected by the application process itself. These errors could be local or transient failures, such as deadlock or resource shortage, or could be some situation discovered in the data. In these circumstances the application process must abort itself, leaving the database in the same condition as if it had never started.

As for system errors, we need to recognise recoverable units of processing, called variously above "transactions" and "applications processes". Again, in Ada, these are most readily equated with blocks. The language construct for "aborting" a block would be the raising of an exception. But we now require an additional implementation technique: updates during the transaction must be made "tentatively" and only be visible to this transaction and no other. Only on successful completion of the transaction should the updates be "committed" and made visible to other transactions.

If the act of commitment is associated with the end of block processing then raising an exception should be able to neatly step over this. A mechanism for making tentative updates is not to make them in place, but in new storage, using logical pointers rather than physical pointers - then simple manipulation of the logical to physical pointer mapping enables local visibility during the transaction and commitment at the end. Ideally the act of commitment should be single uninterruptable step.

Note that this leads to nested transactions. These are unusual, but the locking protocols have been described (MOSS 81).

## 9. VERSIONS

Data exists in a sequence of versions as it undergoes change during its lifecycle. These versions are usually ignored in conventional database management systems, though there have been some discussions of the merit of retaining version history (SCHU 77, COPE 80).

Versions are useful in supporting transactions - if a transaction fails, then the version at the start of the transaction persists: if the transaction succeeds, a new version is produced. Normally the versions would not be visible to the user - only the latest version is available, and update appears (at least) to be made in-place.

However there is a lot of merit in making versions visible, and this could be done in Ada through the addition of extra (system generated) fields to hold the version identification. This version identification should clearly be user meaningful, and would most sensibly be a time stamp.

## 10. CONTROL OVER PERFORMANCE

In database management software it is usual to be given some control over performance which is not directly visible to the programmer and does not require changes to any programs or data definitions. These database administrator facilities control placement of data on discs, details of representation of the data, and similar.

In Ada there is a measure of control over representation, both through specific constructs of the language, and through 'pragmas'. These give both compile time and run-time controls. For databases it is specifically run-time controls that are of concern, and pragmas provide a flexible escape mechanism to influence the behaviour of the run-time system. Any of the extensions to the run-time system discussed here would be controllable through this mechanism.

## REFERENCES


ALAG 81 Alagic S. and Kulenovic A. "Relational Pascal Data Base Interface" Computer Journal Vol 24 No. 2 pp. 121-127 May 1981.

ALSY 83 Alsys "Reference Manual for the Ada Programming Language" ANSI/MIL-STD 1815A January 1983

AMBL 79 Amble T, Bratbergsengen K, Risnes E "ASTRAL-A structured and Unified Approach to Data Base Design and Manipulation" IFIP TC-2 Working Conference on Database Architecture. North Holland 1979.

ATKI 82 Atkinson M., Chisholm K., Cockshott P. "PS-algol: an algol with a Persistent Heap" SIGPLAN Notice Vol 17 No 7 July 1982 PP 24-31.

CHEN 76 Chen P_S.P. "The Entity - Relationship Model - Toward a Unified View of Data", ACM Transactions on Database Systems, Vol. 1, No. 1, pp. 9-36 March 1976.

CODA 71 CODASYL "Data Base Task Group Report", April 1971.

COPE 80 Copeland G. "What is mass storage were free? "5th Workshop on Computers Architecture for Non-Numeric Processing. ACM March 1980.

DATE 75 Date C.J. "An Introduction to Database Systems", Addidon-Weseley, Massachusetts 1975.

ENGL 76 ENGLES R.W. "Currency and Concurrency in the COBOL Date Base Facility" in Modelling in Data Base Management Systems (Ed Nijssen G.M.) North Holland 1976. pp 339-363.

GRAY 81 Gray J., McJones P., Blasgen M., Lindsay B., Lorie R., Price T., Putzolu F., Traiger I. "The Recovery Manager of system R Database Manager" ACM Computing Surveys Vol. 13 No. 2 June 1981.

HALL 75 Hall P.A.V. "Computational Structures" MacDonald and Janes/American Elsevier, 1975.

HALL 76 Hall P.A.V., Owlett J., and Todd S.J.P., "Relations and Entities" Modelling in Data Base Management Systems (Ed Nijssen G.M.) North Holland 1976 pp. 201-220.

KOHL 81 Kohler W.H.. "A Survey of Techniques for Syncronisation and Recover in Decentralised Computer Systems" ACM Computing Surveys Vol. 13 No. 2 pp. 149-188 June 1981.

LEDG 81 Leggard H.. "ADA Introduction. Ada Reference Manual (July 1980)" Springer, 1981.

LEDG 82 Ledgard H. and Singer A. "Scaling Down Ada (or Towards a Standard Ada Subset)" Comm.ACM, Vol. 25 No. 2 February 1982, pp. 121-125.

LOCK 79 Lockeman P.C., Mayr H.C., Weil W.H., and Wholleber W.H. "Data abstraction for Database Systems" ACM Trans. on Database Systems Vol. 4 No. 1 March 1979.

MOSS 81 Moss, J.E.B., "Nested Transactions: An approach to reliable distributed computing." MIT thesis MIT/LRS/TR-260, April 1981.

RAND 78 Randell B., Lee P.A., Treleaven P.C. "Reliability Issues in computing Systems Design", ACM Computing Surveys, Vol. 10 No. 2 pp. 123-165 June 1978.

SCHM 77 Schmidt J.W. "Some high level language constructs for data of type relation" ACM trans. on Database systems Vol. 2 No. 3 September 1977.

SCHM 79 Schmidt J.W. "Data Type Concepts for Databases" pp. 154-175 in Infotech State of the Art Conference "Data Design" London 10-12 September 1979.

SCHU 77 Schueler B.M. "Update reconsidered" IFIP TC-2 working Conference Modelling in Data Base Management Systems, Nice January 1977.

SHIP 81 Shipman D. "The Functional Data Model and the Data Language DAPLEX" ACM Trans on Database Systems, March 1981 pages 140-173.

SKEL 82 Skelly P.G. "The ACM Position on Standardisation of the Ada Language", Comm. ACM, Vol. 10 No. 2 pp. 167-195, June 1978

SMIT 81 Smith J.M., Fox S., Landers T. "Reference Manual for Adaplex" Technical Report CCA-81-02, Computer Corporation of Amenia', January 1981.

VERH 78 Verhofstad J.S.M. "Recovery Techniques for Database Systems" ACM Computing Surveys, Vol. 10 No. 2 pp. 167-195, June 1978.

WELS 81 Welsh J., Lister A., "A Comparative Study of Task Communication in Ada" Software Practice and Experience Vol. 11 pp. 257 - 290, 198