A FLEXIBLE DATABASE MANAGEMENT SYSTEM FOR A VIRTUAL MEMORY MACHINE

by

JANE BARCLAY GRIMSON

Ph.D. Thesis
University of Edinburgh
1980

ABSTRACT

A Database Management System, EDAMS, is described, which is designed to run on the Edinburgh Multi-Access System, EMAS.

EDAMS is based on the 1971 CODASYL DBTG Proposals, but gives the user greater flexibility. It allows the formation of subschema logical records, whose fields can be drawn from any number of records defined in the parent schema. New sets may also be created by the user in the subschema. A device known as a database map, which contains all the set pointers and pointers to the schema record sources of the subschema logical records, facilitates this high degree of flexibility.

In addition, EDAMS provides an efficient algorithm for handling the problems of concurrent update in a database. The operation of this algorithm is assessed on a small test database.

Finally, the effects of designing a database management system for a virtual memory Operating System, such as EMAS, are examined.

INDEX

PART II THE DESIGN OF EDAMS

PART I

DATABASE MANAGEMENT SYSTEMS

CHAPTER 1

THE DEVELOPMENT OF DATABASE MANAGEMENT SYSTEMS

1.1 Introduction

The volume of information recorded in the world is increasing daily. The efficient running of any enterprise - government, banking, insurance, etc. - is critically dependent on having the relevant information at the right place and at the right time. Thus many agencies resorted to the computer to solve their information handling problems. At first the simple Information Storage and Retrieval Systems were able to meet the situation. But gradually many enterprises came to realize that in order to make efficient use of the computer, the computer was forcing them to structure their information in a certain rigorous way, which was not necessarily natural to that enterprise. Furthermore, each department within an enterprise maintained its own separate files with consequent problems of data redundancy and accuracy. For example, employees names and addresses had to be repeated across several different files, e.g. payroll, personnel; if an employee notifies one department of his change of address, that department's file will be updated with the new address, but all other files will have the old and now incorrect address. The need therefore arose for a system which would reflect the real-life situation and act as slave to the management and flow of information, rather than as master of it. The integrated corporate database with the database management

system to support it, represents the attempt to meet these needs.

1.2 Definition of a database

There are many definitions of the terms database and database management system. An early definition of a database as a set of logically related files is no longer considered sufficient; in fact, there is a definite attempt to get away from thinking of a database in terms of a large file or set of files.

The CODASYL Report [1] defines a database as follows:

'A database consists of all the record occurrences, set occurrences and areas which are controlled by a specific schema.'

This definition is useful only within the context of the Report itself. R.F.Schubert [2] defines a database in the following terms:

'A database must be viewed as a generalized, common, integrated collection of company or installation owned data which fulfills the data requirements of all applications which access it. In addition, the data within the database must be structured to model the natural data relationships which exist in a company.'

The drawback of this definition is that it hinges upon the identification of the company or installation which is not always easy to recognize.

The true nature of the database concept includes the following:

(a) integrated collection of data

(b) contains data pertaining to several applications without unnecessary duplication

(c) formal definition of the data

(d) independence of physical storage from logical views of the data.

A database management system (DBMS) is the name given to the software to support the database and is assumed to provide for:

(a) maintenance of data structures

(b) languages for storage, retrieval and update of data

(c) facilities for ensuring data integrity and security

(d) reporting facilities for the Database Administrator (DBA)

(e) separation of physical and logical data structures

(f) simultaneous access to the database by many users, including those who are altering the data (concurrent update).

1.3 Functional development of database management systems

At the end of the 1960's and the early 1970's there was a great surge of interest in the field of DBMSs. Software manufacturers and users alike hurredly designed systems which were not always successful.

The first computer files were simple sequential files on magnetic tape. The records on the file were usually sorted into a specific order and updating such files was often very costly. Even if only one record was to be altered the whole file had to be recopied, which led to the use of batch updates. In a batch update, several updates were grouped together in

a file sorted in the same order as the records on the master file and a new version produced. Such file systems were easy to use and worked well in small, relatively static situations. If, however, the files were large with frequent updates, those systems could become too slow and inflexible.

Then came the direct access disc with manufacturer-supplied access methods such as the Index Sequential Access Method (ISAM). ISAM allowed records to be processed both sequentially and randomly (based on the ISAM key) and updating a single record was possible without recopying the entire file. Whitney [3] sees this era as the first generation of data management systems.

However, as computers became increasingly used for more complex file applications, more sophisticated storage and accessing methods were required. For example, consider a file of student records with student number as the ISAM key, name, address, etc. together with the course(s) the student is taking. To access information about a particular student given the student number is easy. To process the file for a group of students (sorted by student number) is also easy. However, to extract the names of all the students enrolled in a particular course is both time-consuming and awkward. Hence the development of the inverted file which would contain, for example, all the courses together with a list of all the addresses of the records in the master file of students enrolled in each course. A master file can be inverted on any number of key fields, e.g. course, faculty. These inverted files can therefore be quite large and so it is necessary to structure them in such a way that they can be

accessed quickly. The general approach is to separate the keyword (e.g. course name in the above example) from its list of record addresses. These keywords are placed in a keyword dictionary, which can be structured as a binary tree, for example, or accessed by means of a hashing function. This was the era of the Information Storage and Retrieval Systems and Report Generators, eg. RPG, MARK IV, EASYTRIEVE [4], designated by Whitney as the second generation of data management systems.

Although these systems do represent a great improvement with non-procedural user languages and so on, they do not solve all the problems. The cost in terms of storage and maintenance of these massive inverted files, which together often exceed the size of the master file, is considerable. Thus database management systems were developed, Whitney's third generation of data management systems. The aim of the database management system (DBMS) is to provide:

(a) more general and efficient management of large amounts of data
(b) better backup/recovery mechanisms
(c) the elimination of unnecessary, redundant data
(d) perhaps the most important aim, to provide a much higher degree of data independence.

The old file systems were very sensitive to changes in the programs processing the data and vice versa. When each user application maintained its own separate file, this did not matter since each user could change his file of programs without affecting other users; this of course led to inconsistency between files. Once all the applications

are grouped into a single database, a means must be found to maintain this apparent independence from the user's point of view. Thus DBMSs are intended to separate data processing programs from the actual data. Changes made to the overall logical structure of the data should not affect those data processing programs, which are not directly involved. This is known as logical data independence [5]. Furthermore changes made to the physical layout and organization of the data should not necessitate changes to either the overall logical structure of the data or to the data processing programs. This is known as physical data independence.

The importance of data independence in DBMSs cannot be overstressed. If new data items are added, application programs should be independent of these changes. It is also desirable for the environment in which the application programs are run to remain constant, so that if the DBMS is to be run on a different Operating System or even on a different machine, the application system will be unaffected. Clearly, it is not feasible for the DBMS itself to be independent of such a change, but the cost of the reimplementation can be amortized over many applications.

Whitney's third generation of data management systems represents the first generation of true DBMSs such as IDMS, DMS 1100, IMS. It is interesting to note that some so-called DBMSs required report generation and query languages to provide the interface with the user (e.g. GIS [6] and TDMS [7]). Thus while there was no improvement in user interface between the second and third generations of data management systems, the latter provided a better foundation

for higher level facilities. In recent years there have been major developments in the establishment of a theoretical foundation for DBMSs based on Child's relational approach [8,9] and extended by Codd [10].

The relational approach to data systems has been used in deductive question/answer systems for several years. It was not until the late 1960's that its applicability to large, shared data banks was suggested by Codd. The main aim of this approach is to ensure data independence, it also provides the user with a powerful algebraic language to operate on the data.

There has been considerable controversy over whether the relational approach will in fact gain wide commercial acceptance, ultimately replacing the CODASYL DBTG approach. Michaels et al [11] in their comparison between the two concluded that neither represents the complete solution to the database management problems of the entire user community. Indeed, it seems probable that an amalgam of the two systems will emerge as being the most acceptable, to form the fourth generation of data management systems, the second generation of DBMSs.

However, at present most of the implementations of the relational approach are being carried out on a purely experimental scale in universities and research establishments, whereas there are a number of large, commercially-available partial implementations of the CODASYL proposals.

Finally, the mode of use of DBMSs has changed in recent years from batch to interactive. This has had profound effects on both the design and implementation of these systems.

1.4 The objective of this thesis

The starting point for this thesis was the April 1971 CODASYL DBTG Report and the Edinburgh Multi-Access Operating System, EMAS. It is intended to show that:

(a) it is feasible to implement a CODASYL-type DBMS on a virtual memory, multi-access Operating System

(b) it is possible, within the overall CODASYL framework, to provide the user with much greater flexibility in his use of the data in the database by allowing him to form his own logical records, whose fields can be drawn from all over the database without restriction

(c) an efficient and simple algorithm can be devised for solving the problems of contention between users during concurrent update of the database.

1.4.1 Layout of the thesis

This thesis is divided into three parts. Part I consists of an overview of the field of Database Management Systems together with a detailed discussion of the application of DBMSs to Hospital Information Systems. Part II outlines the design of a DBMS called EDAMS, which is based on the CODASYL proposals, but which provides the user with much greater flexibility, and which uses a new approach to concurrent update (see above). Part III contains the details of the implementation of EDAMS on the Edinburgh Multi-Access System, EMAS.

CHAPTER 2

A DATABASE MANAGEMENT SYSTEM APPLICATION

2.1 Introduction

Database management systems (DBMSs) are used in a very wide variety of applications ranging from Airline Systems (including the highly successful passenger seat reservation systems) [12], Production Control Systems [13], Management Information Systems [14] to Hospital Information Systems. The Hospital Information System (HIS) has been selected for special study in this thesis to provide a background against which to design a DBMS for the Edinburgh Multi-Access System (EMAS). The HIS has been chosen because it is comparatively new area of application for DBMSs, especially in the U.K., and because the benefits to be derived from it are practical (improvement in the quality of patient care) as well as financial (better use of resources).

2.2 Hospital Information Systems

The remainder of this chapter is concerned with a detailed examination of one application for a DBMS - namely, the Hospital Information System (HIS). Much of the material is based on a survey carried out at the Royal Infirmary, Edinburgh (RIE).

A HIS is a computer system for on-line processing with real-time responses of in-patient and out-patient data for one or more hospitals. The use of computers in hospitals

is still only in its early stages. Even in the United States their use is aimed at increasing the cost-effectiveness, through more efficient patient billing and accounting systems, rather than to improving the quality of patient care. If the public and the medical profession can be convinced that computer systems can ensure the privacy of medical data, there is undoubtedly a great potential in the field of HIS. Moreover, as the process of providing medical care becomes more and more complex, so the need for systems to handle patient records is becoming increasingly urgent, especially in large hospitals. Greenes et al [15] feel that it is now a matter of the highest priority to develop computer-based management systems for handling patient data. Moreover, such systems could automatically incorporate both the administrative and the research functions.

The basic aim of the HIS can therefore be summarized as follows:

(a) to provide the medical staff with all the information required in the provision of medical care, i.e. handling of patient records, laboratory reports, X-ray reports, etc.

(b) to provide the administrative staff with all the information required for the efficient management of the hospital, i.e. handling of admission procedures, bed census, menu planning, accounts, personnel and payroll (where appropriate) etc.

(c) scheduling and resource allocation

(d) as an off-shoot, to facilitate research into the diagnosis and treatment of disease.

2.2.1 Effects of the HIS

Having decided what the basic aims of the HIS are and what type of information is to be processed, it is necessary to consider the effects of the system by posing three questions:

(a) who will the system help and in what way?

(b) who might suffer?

(c) what are the relative economics of the HIS versus the system which existed prior to the introduction of the HIS?

The answers to the first two questions are critical. If, for example, the HIS results in a deterioration in the standard of medical care, then it is totally unacceptable, no matter how marvellous it is for the medical and administrative staff. Great care must be taken not to decrease the quality of patient care and it would not be unreasonable to expect it to improve as a result of the more timely provision of medical data. It was found at the Texas Institute for Rehabilitation and Research [16] that their system for on-line scheduling of patient care activities was, in some ways, too efficient; the computer was able to fill the patient's day so completely that he was exhausted by the end of it! Furthermore, users of the system (doctors, nurses, etc.) tended to depend entirely on the computer system at the expense of verbal communication both among themselves and with the patients, which is a vital part of medical care. On the other hand, the Ward Information Management System at the John Hopkins Hospital [17] has shown that the computerization of doctors' orders (for drugs, diet, investigations, etc.) resulted in a substantial reduction in the

number of errors in carrying out these orders (previously 15% of orders were not carried out correctly). This must surely represent a highly desirable effect of the HIS, which will result in an improvement in the quality of patient care.

Another potential pitfall and undesirable effect of the HIS is that workloads could be increased to unacceptable levels. For example, doctors might be required to spend long periods of time at computer terminals typing in their observations, orders and so on. This activity is purely clerical and doctors' skills would be far better employed elsewhere. However, in order to ensure a low error rate in the input data, it is always best to capture the data at source. Doctors should supervise the entry of their own clinical data and verify it immediately so that it may be corrected on-line. A Cathode Ray Tube (CRT), preferably with light-pen as well as keyboard, is the most widely used terminal device in hospitals. When large volumes of data have to be entered into the system (e.g. patient registration), this can be done by data preparation personnel, thereby keeping the typing by medical staff to a minimum.

As regards the relative economics of the two systems, manual or computer, it is unlikely that the computer system would work out any cheaper. The capital expenditure on the equipment required to support a HIS would take several decades to recoup. Moreover, the number of staff - in this case administrative staff - is hardly likely to decrease. Indeed if the experience of industry is anything to go by, the introduction of a computer results in an increase in the

number of staff required, but hopefully also with improved service.

2.3 The medical record

The most fundamental part of any HIS, whether manual or on a computer, is the medical record. The medical record contains all the relevant information about a person's health and consists of three main parts:

(a) personal information

(b) medical history

(c) current treatment

It is the processing of parts(b) and (c) which has proved to be a major stumbling block in the development of computerized systems. There is no standard format or terminology for recording this clinical information. The doctor very often uses a personal form of shorthand together with short pieces of text and aides-de-memoires. To transfer this information directly onto the computer, even in the form of English narrative, would be very wasteful and would result in the computer being used as a very extravagant filing system. Furthermore, it would probably be considerably more tedious to use than the manual system it replaced.

The personal information section of the medical record is quite straightforward, consisting of name, address, sex, place and date of birth and so on. This type of information is common to all personnel files, whether or not they are making use of computers; its structure is known in advance and is constant for all patients.

The recording of the medical history of a patient, however, is much more difficult. The information to be recorded will vary dramatically from one patient to another and, as indicated above, there is no standard terminology for recording items such as doctor's observations, physical examinations and so on.

It is not difficult to handle the recording of the major medical events in a patient's life, e.g. date, diagnosis, treatment, with details of periods spent in hospital etc. In addition, it would probably be helpful to record the name of the doctor who treated the patient and where further information about the illness and treatment can be found.

Some research has been done into the use of computers which interact with the patient by means of a question/answer system in order to obtain his medical history. The computer asks the patient a question and, according to the answer given, follows one of a number of paths of further questioning. If, for example, the patient is asked to indicate whether or not he has ever suffered from chest pains and he answers in the negative, then the computer might go on to ask whether or not he has ever had liver disease. If the answer to the question regarding chest pain is positive, then the computer will ask further questions pertaining to the chest pain before going on to ask about liver disease.

A summary of the patient's medical history could then be printed immediately. The doctor examining the patient can then ask the patient for further details and enter them into the system, if necessary. At this point, the doctor

should be given the alternative of using either the question/answer system or to enter his remarks in the form of unstructured narrative.

The major drawbacks of such systems for obtaining medical histories is their unreasonable reliance on the patient's memory and knowledge; indeed, some may be so confused as to be unable to reproduce their names consistently. However, the alternative of a national databank in which the major medical events in the lives of every member of the population are recorded is some way off. In normal circumstances, when the patient can be identified, the medical histories of incoming patients at least for the immediate past, would be available to the hospital from the patient's G.P.

The current treatment section of the medical record will contain a mixture of both structured and unstructured data. Among the structured data will be admission details for in-patients, for example: date, by whom referred, doctor-in-charge, diagnosis (if any), ward number, together with results of any number of laboratory tests in varied, but well-defined formats and X-ray reports. The unstructured data will include symptoms, doctor's observations and orders and nurses notes.

As with the taking of medical histories, a question/answer system with CRT, light-pen and keyboard could be used to capture the data. It is even more important that the doctor be permitted to use narrative as an alternative to the answers supplied. Abrams et al [18] quote as an example the situation where a doctor wishes to record the condition

of a patient relative to the last consultation. He would choose one of the following alternatives displayed on the CRT:

CURED / BETTER / SAME / WORSE / VERY MUCH WORSE / DEAD / OTHER

It is by selecting the 'OTHER' category that the doctor can enter narrative as a response, not simply because he feels that the patient's condition did not fall into any of the listed categories, but because he wished to elaborate further. The drawback in using the question/answer systems is that they could tend to lead the doctor too much, rather than allowing him to use his own knowledge and experience.

2.4 Patient identification

One of the main problems associated with a medical record database is that of patient identification. The simple and most straightforward method is to use the patient's name. It is unlikely that a patient will forget his name, assuming that he is conscious and even if he is unconscious his name can usually be ascertained without too much difficulty. It should be noted that a patient's name can change, e.g. on marriage, and cannot therefore be regarded as absolutely invariant. The survey in the Accident and Emergency (A&E) Department of RIE has shown that with the exception of patients injured in road traffic accidents and who have collapsed in the street, the names of the vast majority of patients can be ascertained immediately on arrival, either from the patient

himself or from a relative or friend. However, there are many obvious problems associated with the use of the name as an identifier - it is very far from unique (e.g. in the index for past in-patients at RIE, there are 90 patients called Alexander Smith), it is prone to mis-spelling and in manual systems to mis-filing.

An alternative to the use of the name as the basic key to patient identification is to use the patient's date of birth. This is the system which is currently in use at the Central Medical Records Department at RIE. The main library of medical records is filed by date of birth, in chronological order; within any given birthdate, records are stored alphabetically according to name (surname first). A separate card index is maintained to access the main library. This index is in alphabetical order of patient name (surname first) with date of birth as the secondary key.

The A&E Department at RIE assigns a unique number to each new patient (pre-printed on the registration form) and uses a file of names and addresses as an index. With many patients changing address from one visit to the next and with the non-uniqueness of names, this system is also unsatisfactory for general patient identification. At least the date of birth system has the merit that a patient's medical records can be retrieved without reference to any other documents. No-one can be expected to remember a completely arbitrary string of digits, as used in the A&E Department.

It is possible to envisage some far-fetched system which could incorporate names with mother's or grandmother's maiden name or date of birth, which could identify a large

population almost uniquely. However, a friend may well not know a patient's mother's maiden name, let alone his grandmother's! Systems based on place, time and date of birth have also been proposed, which can guarantee almost complete uniqueness, but which suffer from the same disadvantages.

All the solutions proposed above are unsatisfactory from one point of view or another. Moreover, none of them solves the problem of the unconscious patient who is brought into A&E alone without a friend or relative to give any information. A solution which is often put forward half-seriously is that everyone should wear an identification bracelet with a unique number on it which was assigned at birth. It is even proposed, though less seriously, that the number should be tatooed somewhere on the body. What happens, however, if the bracelet is lost or the number partially obliterated?

There is no simple answer to the problem of patient identification. It is certainly desirable for people who suffer from chronic diseases or who are allergic to certain drugs to wear an identification bracelet and/or carry an identification card at all times. Although these people form only a small percentage of the total population, they are a very significant percentage because of the high risk involved if they are not correctly identified. The general population, however, would not be so well motivated to carry the necessary identification.

Assuming, therefore, that a patient's name, sex and approximate age are known, it should be possible to devise an algorithm which could search rapidly through the patient

indexes stored in the computer in order to identify him and ascertain whether any details of his medical history are known. If an exact match is not found given the identification information available, a list of the closest matches found could be printed.

In the majority of cases in A&E at RIE, it is not strictly necessary to match up a patient's notes - in fact, at present, this is done in less than 13% of new cases. If a patient comes into A&E in March with a broken arm and then returns in November with a cut toe, the previous case notes would not be relevant. If, on the other hand, the patient had broken the same arm in November, the doctor might want to consult the March case notes and X-rays. In this case, the computer would have to consult the database immediately for details of the March episode. If the old case notes were not required, the computer would still have to link up the two episodes eventually. Such linkage could be carried out when the computer is not busy. However, with all the problems of patient identification outlined above, it is possible that the two episodes cannot be linked reliably by computer without any human intervention.

2.5 Concluding remarks

In this section some broad conclusions will be made regarding the requirements which a HIS imposes on the DBMS which supports it. Clearly, in order to draw detailed conclusions an exhaustive study of existing procedures in a

full hospital activity analysis would have to be carried out, which is not within the scope of this thesis.

There are two distinct aspects to the automation by computer of the information processing activities in hospitals. The first is the design of the HIS and the second is the design of the DBMS to support the HIS. Ideally, the HIS should be designed first and the DBMS should be constructed in such a way as to meet the requirements imposed by the HIS.

The design and implementation of a DBMS involves several man-years of effort and the hope is that a particular DBMS will be applicable in a wide variety of situations. Most of the effort today is being directed towards the design of these general-purpose DBMSs. This approach, therefore, is based upon the premise that the information handling requirements of the various applications are similar. Consider, for example, airline systems; they are designed as special-purpose DBMSs and as such they could be of use only to another airline, but certainly not for a complete HIS. However, a superficial comparison between the passenger seat reservation system alone and the appointments system in an out-patient department reveals certain similarities. The two processes of making an appointment and booking a seat are alike. A patient makes an appointment (sometimes many months ahead) for a particular clinic, on a particular day, at a particular time, while a passenger usually books a seat for a specific flight, on a specific day, at a specific time. A significant difference between the two systems is that whereas the patient will generally take the first available appointment, the passenger usually wants to book a seat on a specified flight.

In a comparison between financial systems and Hospital Information Systems, Dr. Reekie [19] showed that while the privacy requirements of the two systems are the same, the volume of transactions per service (laboratories, X-rays, etc.) is quite different. On average, each patient makes only one call on each service per day. Thus each service will have at most somewhere over a thousand transactions per day. Moreover, experience has shown that there are peaks of activity in a hospital between 9 a.m. and 11 a.m. with a smaller peak in the afternoon. Financial systems also suffer from peaks in the transaction rate and in both systems it is difficult to spread the load evenly throughout the day and night.

The distinction is made between special-purpose and general-purpose DBMSs. Although, as stated previously, most of the research is currently focussed on general-purpose DBMS, it is undoubtedly true that given a particular application (and sufficient resources), it is always possible to design a more efficient special-purpose DBMS, which is tailor-made for that application, than to use even the very best general-purpose system.

It is difficult to separate the requirements which a HIS imposes on the DBMS from those it imposes on the Operating System and hardware. Increasingly, the logical and physical aspects of DBMSs are being separated. Thus the logical aspects of the DBMS design involve the data structures, user interface, data protection and security and so on. The physical aspects are concerned with the volume of information

to be handled, activity rates and so on.

To conclude, the requirements imposed by the HIS on the DBMS can be summarized as follows:

(1) system reliability - both the hardware and software of a computer system supporting a HIS have to achieve almost 100% reliability. They have to be available 24 hours a day, 7 days a week and 52 weeks a year. In order to do this, experience with airline systems has shown that every item from CPU to data record must at least be duplicated; indeed most systems are triplicated. Such a dual system would be essential in a hospital which relied completely on a large central computer. It is well worth examining the possibility of using a network of mini-computers located in the various departments throughout the hospital, each supporting its own small database. A patient attending a number of different departments in the hospital might have a number of different specialist clinical records with a central identification, history and summary section "passed round" the relevant departments. The mini-computer network would be linked together in such a way that if one breaks down another can take over its urgent on-line work, in addition to its own. Such an approach has the added advantages (apart from enhancing the reliability of the system) that each department would have control over its own portion of the database and it would also be cheaper than a system which required a lot of built-in redundancy.

(2) storage hierarchy - the DBMS must be able to support a database which is spread over a number of different

storage devices, e.g. disc, drum, tape. Records would be moved automatically by the DBMS, according to rules specified by the application programs, from one level in the hierarchy to another. For example, the records of in-patients would remain at the top level of the hierarchy, i.e. on an on-line storage device, until the patient is discharged, when his record would automatically move to a lower level until required for the patient's check-up later. Out-patient records will not move to the top level until the day (or maybe even the hour) of their appointment. This is exactly analagous to the present manual system at RIE where case-notes are "pulled" from Central Records for out-patient clinics a few days ahead of the clinic. The lowest level of the hierarchy would represent archival storage. Presumably most of the information contained in these records could be safely destroyed after the patient had been dead for a number of years, retaining only those details which would be relevant for research purposes.

(3) Foreground and background operation - the DBMS would have to support both high-speed on-line operation and background batch work. On-line operation would get priority. Moreover, it might be desirable to have a priority attatched to each request, based on the type of request and its source. For example, a doctor in A&E urgently requesting a patient's case notes would be serviced before a radiologist updating a patient's record with the result of a non-urgent X-ray. In some situations, the priority system might not be practical as it could take longer

to establish the priority than to service the request.

(4) privacy - it is clearly of the utmost importance to ensure the confidentiality of medical data. At the London Hospital [20], where a small computer system is in use for admissions, it was felt that the records stored in the computer were better protected than the traditional case-note folders. In spite of the fact that the folders are not supposed to be handled by any unauthorized person, including the patient himself, folders are often left lying around for anyone to read. However, the London Hospital Project does take a more positive attitude to privacy and security than this might suggest. The video screens are located in rooms to which patients and members of the public do not have access. The casual snooper would have to know how to log on to the system to obtain any information. The consultants can specify at the time the patient is placed on the waiting list, whether or not their medical data is to be displayed at all. Finally, the screens fade very rapidly when not in use. Thus the DBMS would be required to provide privacy facilities down to the data item level. These facilities could take the form of one word keys or of privacy routines which could check the identity and authority of the user. It has been suggested that, in a nationwide medical database, the patient himself should be given the key to access his own medical record. While this would violate the currently held principle that patients should not be allowed to see their own medical records, it is in keeping with modern thinking

on civil rights. Thus anyone who records information about someone else (e.g. government agencies, credit rating firms, hospitals, etc.) should allow the subject of the information to access any factual data. In this way, cases of ill-justice due to incorrect information can be reduced.

CHAPTER 3

ELEMENTS OF DATABASE MANAGEMENT SYSTEMS

3.1 Introduction

In this chapter a number of aspects of DBMSs will be examined. In particular, data structures, data independence, data integrity, privacy and security will be discussed in detail.

A well-defined hierarchy of users of a DBMS can be identified and the significance of, for example, data integrity will vary according to the user's position in this hierarchy. Broadly speaking, the users of a DBMS can be divided into the following categories:

Level 1 (DBMS implementor) - perhaps not strictly 'user'

2 entire database description implementor - CODASYL schema writer

3 individual application description implementor - CODASYL subschema writer

4 application programmers

5 high level users - terminal enquiry etc.

Figure 3.1 Hierarchy of DBMS users

It should be noted that where a general framework is required in which to discuss, for example, data independence, the CODASYL April 71 DBTG Report [1] will be used.

3.2 Data structures

The term data structure is used in DBMSs to describe the user's view of the data and excludes details of storage techniques [21]. It therefore spans the data from the level of individual data item to the complete database. However, the level at which the greatest divergence in the approach taken by individual DBMSs arises, is the level of the group data structure; i.e. what structures the system employs to enable the user to describe relations between groups of data in the database. The term group relation, rather than simply relation is used in order to exclude the implicit association between data items and fields in an individual record.

There are three main classes of group relation data structures in DBMSs:

(a) network or set type

(b) hierarchical

(c) relational

3.2.1 The network or set data structure

A network data structure is one which permits a many-to-many relationship between records of which the CODASYL set [1] is an example. Although the CODASYL set is strictly speaking a one-to-many relationship, it can be used to represent a many-to-many relationship (see below). The CODASYL database consists of many different record types with related records grouped together into sets. Each set must have one owner record and one or more member records. There will be many occurrences of the same set type in the database and to avoid

confusion and ensure database integrity, a record occurrence cannot appear in more than one occurrence of the same set, i.e. a member record occurrence can have only one owner record occurrence in a set and owners are all distinct. It is this restriction which implies that the set is only a one-to-many relationship, but a many-to-many relationship can be represented by the simple introduction of a link record. Thus the set is regarded as a network structure. In the April 71 CODASYL Report [1], a second restriction was imposed which did not allow a record type to be both owner and member in the same set, but this restriction has been removed in the 1978 Journal of Development [22].

Membership of sets can be either MANDATORY (i.e. permament), in which case the record occurrence will only cease to be a member of the set when it is deleted from the database (or altered in such a way that it no longer qualifies as a member of that set), or OPTIONAL (i.e. temporary). In addition, set membership can be defined as AUTOMATIC, when records are inserted into sets automatically by the DBMS, or MANUAL, when records are linked into sets by specific user command.

3.2.2 The hierarchical structure

The hierarchical structure, as the name implies, is a father/son tree structure representing a one-to-many relationship only. An example of a DBMS using this class of data structure is IBM 's IMS/2, which is used as an illustration here [23, 24].

The basic data element in the IMS database is the segment. A segment is of fixed length and contains one or more logically

related data fields. These segement types are then joined together into a hierarchical tree structure known as the logical data base record. The IMS database thus consists of a number of logical data base records. Each application forms its own individual view of the database by specifying the segements to which it is sensitive. This is analagous to including certain record and set types of a parent schema in a subschema. An application program cannot access those segments to which it is not sensitive.

A segment of information can participate in more than one logical data structure, analagous to permitting a record type to be a member of more than one CODASYL set. The segment data itself exists only once in the database. In one structure, the duplicated segment will be replaced by a pointer to the actual segment where the data is stored:

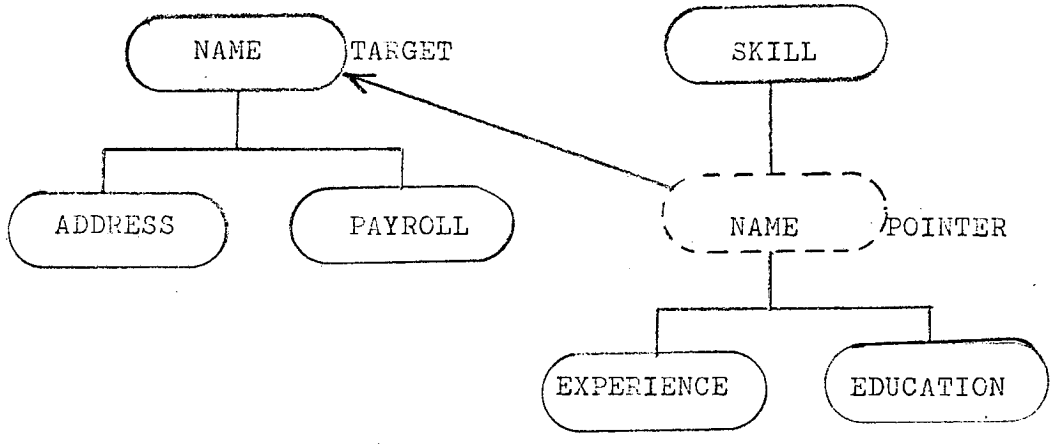


Figure 3.2 Target segment in an IMS database

There is a total of six retrieval functions:

(a) GET UNIQUE (GU)

(b) GET HOLD UNIQUE (GHU)

(c) GET NEXT (GN)

(d) GET HOLD NEXT (GHN)

(e) GET NEXT WITHIN PARENT (GNP)

(f) GET HOLD NEXT WITHIN PARENT (GHNP)

A GET UNIQUE call is used to retrieve a unique segment or path of segments; it is a useful means of establishing position in the database after which GN and/or GNP calls are used.

A GET NEXT retrieval request returns the next segment to which the run-unit is sensitive. The ordering of segments, corresponding to Knuth's pre-order traverse [25] as shown in Figure 3.3

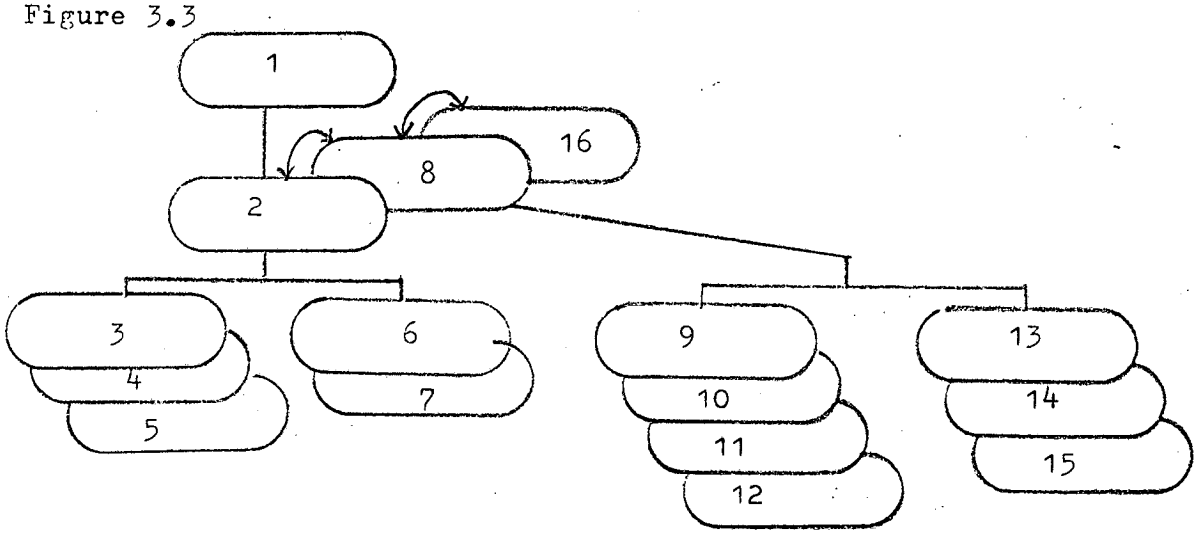


Figure 3.3 Segment order in an IMS database

A GET NEXT WITHIN PARENT call will obtain the next segment(s) within the family of a parent segment. The appropriate parent is established from the last GU or GN, which must have been successful.

The use of the HOLD options for a retrieval request is used to indicate that the user intends to delete or update

the segment; the rules for interpreting the functions remain unaltered. Under IMS/2, the feature is redundant since it is forbidden for two run-units to operate concurrently which have indicated that they intend to delete or update the same segment(s) in the database.

3.2.3 The relational data structure

The relational model of data developed by Codd [10] is based upon the mathematical theory of relations: given sets S1, S2, ... Sn, R is a relation on these n sets if it is a set of n-tuples, each of which has its first element from S1, its second from S2 and so on, i.e.

$$R = \{\langle e1, e2, \ldots e_n\rangle, \langle e1, e2, \ldots e_n\rangle, \ldots \langle e1, e2, \ldots e_n\rangle\} \mid e1 \in S1,\ e2 \in S2,\ \ldots\ e_n \in Sn.$$

The set Sj is defined as the jth domain of R.

Each relation has a primary key associated with it. A primary key is a domain (or group of domains) in the relation which uniquely identifies each tuple in the relation.

Consider the following example of a relation, supply, of degree 4, where the first domain consists of suppliers, the second of parts, the third of projects and the fourth of quantities:

supply(supplier	part	project	quantity)
1	2	5	17
1	3	5	23
2	3	7	9
2	7	5	4
4	1	1	12

Figure 3.4 The supply relation

The relation represents shipments in progress of parts, in specified quantities, from suppliers to projects. The primary key for the relation supply would be (supplier, part, project), all three domains being necessary to identify each tuple.

Although not strictly part of the relational data structure itself, it should be noted that this model of data automatically supplies functions and a language to operate on the data.

3.3 Data independence

One of the major reasons for an organization to adopt a DBMS is that system's ability to mirror the real-life situation within the organization. Of particular importance is the ability of the DBMS to handle the ever-changing demands of the enterprise. For example, radical restructuring of the database, as a result of new company policies, will be necessary from time to time. It is essential that existing application systems should be unaffected by these changes and this insulation is known as data independence.

There are four levels in a DBMS which must be insulated from one another:

(a) physically stored data

(b) database administrator's logical view of the whole database (schema)

(c) application's view of the subset of the data (subschema)

(d) application program itself.

The distinction is made between logical data independence and physical data independence [5]. A DBMS which provides physical data independence will allow the physical layout and organization of the data (level a) to be changed without affecting either the logical structure of the data (levels b and c) or the application programs (level d). The provision of logical data independence, on the other hand, permits the logical structure of the data (b and c) to be altered without changing the application programs (d). Of course, many alterations to the database will necessitate changes to all levels of the database management system (e.g. addition of new data item), but data independence is intended to ensure that the only elements requiring alteration in the system are those which are directly and logically involved in the alteration.

A change in the method of physical data storage, e.g. the reorganization of the data on secondary storage to increase efficiency, should not in any way affect the application programs. Whether or not such a change will affect levels b and c, the schema and subschema, will depend on how the system is implemented. Ideally, however, it is only the interface between b and a, presumably in the form of tables, which would require alteration.

Consider next the elimination of all the records of a given type. Such a change is bound to have repercussions at every level, but all application programs and subschemas which do not use the eliminated record type, should not be affected. First, the data records must be removed at level a, their descriptions and any reference to them in sets etc.

removed at both levels b and c, and of course, in the application programs themselves at level d. It is not necessary to physically remove the deleted records from the database; it would be more efficient to leave this to the next restructuring of the database. It is necessary to consider very carefully what happens to sets in which the deleted record participates. For example, if the deleted record is the only member of a set, the set could be deleted from the schema and/or subschema or simply appear as a memberless set. There is clearly no obvious answer to these problems, but an agreed standard would clearly be an advantage for those who want portable programs.

The next case to be examined is the addition of a new field to a record type. Again, the physical changes must be made to the database 'simultaneously' with the corresponding changes to the schema. Data independence should then guarantee that no more changes will be necessary either to the subschemas or to the application programs, even though they may use the record type involved, but are not interested in the new field. Naturally, those application programs which wish to use the new field, would have to be amended along with their subschemas.

3.3.1 Binding

The degree of data independence of application programs will be affected by when the binding between the user reference to the data and the physical access to it takes place [26]. Traditionally, data is bound to programs at compilation time (sometimes even at program design or coding time), whereas for maximum independence, binding should take place

as late as possible, i.e. at command execution time. Most DBMSs adopt a mixed approach to binding with some taking place at compilation time, some when files (realms) are accessed for the first time and the remainder at command execution time [21], resulting in a compromise between maximizing data independence and maximizing efficiency.

Finally, it is worth mentioning that although the main aim of data independence is to provide flexibility in the DBMS to enable it to adapt readily to the changing demands of the users, a by-product is also the provision of a measure of protection; users will not be aware of or have access to data outside the data defined in their own subschemas. This approach is also less demanding on the user, since he only learns those details of the database which are directly relevant to him.

3.4 Database integrity

It is clearly of fundamental importance that the data in a database is correct and time-consistent and that the linkages between related data items are correct. If the database were to be frozen at any point in time when no changes were being made to the database, it should be a valid picture of the real-life situation it represents.

There are several aspects to ensuring the integrity of a database:

(a) logical consistency checks

(b) validation of input to the database

(c) protection against interference between concurrent run-units, in particular during update

(d) backup and recovery measures

(e) consistency of multiple copies of data

3.4.1 Logical consistency checks

A database consists not only of data, but also of relationships between the data, which together form the data structure (see Section 3.2). Apart from the fact that a relationship between one or more records may form the basis of the storage/retrieval of a record, the relationship itself carries information implicitly, e.g. father/son, owner/member. It is therefore of vital importance to the overall integrity of the database that these relationships are logically consistent. Thus, for example, in the CODASYL system, it would be essential to ensure that an ownerless set had not evolved or, alternatively, that a record had been made inaccessible by virtue of the deletion of all pointers to it. The detection of such logical inconsistency over the entire database is clearly very costly. However, much of the checking can be done when updates are being carried out, especially where the alteration of relational pointers is involved. Since it is not possible for high level terminal enquiry users, application programmers or even subschema writers (levels 6,5,4 of Figure 3.1) to be aware of the indirect effects of their updates on other users, the responsibility of providing the logical consistency checks falls on the DBMS implementor and the schema writers (levels 1 and 2 of Figure 3.1).

3.4.2 Validation of data

No matter how elaborate the mechanisms in the DBMS for ensuring database integrity are, they will be totally useless if the input to the system is incorrect. At first, the question of the validation of input data would appear to be more the concern of the organization whose data is being stored in the database, rather than of the DBMS itself. However, when there are many different users of the data, the traditional approach of each user program validating its own input becomes insufficient. Instead, it is necessary to incorporate validity checking routines within the Data Definition Languages. For on-line systems it may be more efficient to display the information for immediate verification before transmitting it to the DBMS. There would still need to be a further check within the DBMS before finally storing the data in the database. Thus the validation of data involves both DBMS and the application program; some aspects may only be visually checked by the high level terminal user (level 6 in Figure 3.1).

3.4.3 Concurrent update

The subject of concurrent update of a database is discussed in detail in Chapter 5. In this section, the problems which arise when more than one run-unit is updating the database at the same time will be explained, but the solutions will be left mainly to Chapter 5.

One of the important aims of a DBMS is to allow more than one user, each with his own view of the data, to access

the database simultaneously. Concurrent data retrieval presents no problem of interference, but severe difficulties can arise when concurrent update is permitted. Apparently successful updates can be overwritten thus leaving the database in an invalid state.

There are a number of different forms which the interference between run-units concurrently updating the database can take. They depend upon the type of update being performed. The simplest situation is:

Run-unit A reads version 1 of record 1
Run-unit B reads version 1 of record 1
Run-unit A updates record 1 changing version 1 to version 2
Run-unit B updates record 1 changing version 1 to version 3

The update of run-unit A is lost as run-unit B overwrites it. Run-unit B should have been informed that the record had been changed after it had read it, or it should have been prevented from reading a record which had been read for update, or this conflict should have been resolved in some other way.

The standard approach to this problem is to use locks. A run-unit which wishes to update the database can, before it reads a record, prevent other users from accessing it until the update is complete. This is done by applying a lock to the record, thereby claiming exclusive right of access to the record.

Run-unit A locks and reads version 1 of record 1
Run-unit B attempts to lock and read record 1, but is
    queued awaiting release of the record by run-unit A
Run-unit A updates record 1 changing version 1 to version 2
Run-unit A unlocks record 1

Run-unit B locks and reads version 2 of record 1
Run-unit B updates record 1 changing version 2 to version 4
Run-unit B unlocks record 1

Provided a run-unit is limited to claiming one lock at a time, i.e. it must release one record before claiming another, this simple approach works well and is easy to implement. However, if a run-unit can claim more than one lock at a time, deadlock can occur (see below).

A more subtle form of interference can occur when run-units are updating groups of records, i.e. reading a number of records and on the basis of certain criteria updating one or more of the records. Consider the following example:

Run-unit A reads version 1 of records 1 and 2 and validates transaction a against version 1 of record 1
Run-unit B reads version 1 of records 1 and 2 and validates transaction b against version 1 of record 2
Run-unit A uses transaction a to update record 2 changing version 1 to version 2
Run-unit B uses transaction b to update record 1 changing version 1 to version 2

Both transactions passed the validation checks against version 1 of records 1 and 2, but due to the updates neither would pass against version 2 of the records. Thus an inconsistent database has resulted.

Again, the application of locks will avoid this type of interference:

Run-unit A locks and reads records 1 and 2
Run-unit A validates transaction a against record 1
Run-unit B attempts to lock and read records 2 and 1 and is queued awaiting run-unit A
Run-unit A uses transaction a to update record 2
Run-unit A unlocks records 1 and 2
Run-unit B locks and reads records 2 and 1

Run-unit B validates transaction b against the new record 2, but the transaction is rejected

Run-unit B unlocks records 2 and 1

The example above is of a consistent series of updates, i.e. a process requires a time-consistent view of a number of records before deciding which to update. By locking all the records involved, even if only one is to be updated, no interference can arise.

Once a process is allowed to claim more than one resource (record) in a random order, deadlock can occur. The typical case is:

1 Run-unit A reads and locks record 1
2 Run-unit B reads and locks record 2
3 Run-unit A attempts to lock record 2 and is queued awaiting run-unit B
4 Run-unit B attempts to lock record 1 and is queued awaiting run-unit A

Neither run-unit A nor B can continue. In order to resolve the deadlock, either A or B must be pre-empted and its resources released.

The problems which arise when deadlock occurs are by no means trivial. In order to pre-empt run-unit A in the above example, it is necessary to position it prior to its issuing the lock and read request for record 1. This repositioning is known as rollback. However, run-unit A may have made changes to other records in the database in the meantime and all these changes would have to be reversed as well as its own internal variables. What happens to other processes which have been affected by these changes (i.e. which have used the altered records) is often not considered in existing systems. Ideally, they too have to be rolled

back and so the problem mushrooms.

In general, the designers of DBMSs tend to prefer to adopt solutions to the concurrent update problem which do not give rise to deadlock or which enable rollback to take place to a predetermined place known as a checkpoint, in the program, without rolling back other run-units.

Deadlock need not occur directly between two run-units, but also through a chain of intervening run-units. For example:

Ra = {R1, R2, R3, R4} and Wa = {R5}
Rb = {R6, R7, R8} and Wb = {R1}
Rc = {R9, R10, R11} and Wc = {Rb}
Rd = {R12, R13, R5} and Wd = {R9}
where Ri = set of records currently locked by run-unit i
and Wi = set of all records for which run-unit i is currently queued

The deadlock is between run-units a and d through the intervening run-units b and c. The detection of this type of chain deadlock is not straightforward. An algorithm based on a graph theoretic model of the database involving loop detection is proposed by King and Collmeyer [27]. However, even having detected the deadlock, there still remains the problem of which run-unit to pre-empt and how.

3.4.4 Backup and recovery measures

There are two ways in which data can be lost completely:

(a) hardware error, e.g. at input terminal, transmission line, disc head crash.

(b) writing of data to an area outwith the control of the database, including data lost due to inaccessibility following corruption of pointers to the data.

There is little that the DBMS can do to guard against hardware faults, but it must ensure that users are notified as soon as possible and that adequate recovery measures can be taken by the system.

As regards the second manner in which data can be lost, it is assumed that the DBMS is incapable of setting up the links between the data incorrectly or of storing the data at the wrong address. If the data links become corrupted thus leaving the data inaccessible, then the restoration of the links following recovery should also automatically restore the data.

There are three aspects to backup and recovery measures:

(a) backup copies of the database or portions of it

(b) journal file of database transactions

(c) checkpoints

The traditional approach was to maintain father/son/grandfather copies of data files on tape. In the event of failure, the entire file was then restored from tape. This would be impractical in the huge databases of today. This is well illustrated in the Infotech State of the Art Report on Database Management [28] where the example is given of the time it would take to dump the entire warranty files of the Detroit car manufacturers - namely, 48 hours each day. The solution therefore is to divide the database into several physical areas on different storage devices, so that only one disc or drum, say, has to be restored following system failure. Backup copies (dumps) are made of certain highly active and vital portions of the database at frequent intervals, supplemented by less frequently taken copies of the entire

database. Although this is a time-consuming exercise during which the portion of the database being copied will not be available to users, it is a convenient time to carry out at least a partial database reorganization. This reorganization can take the form of simply compacting empty spaces but it can also consist of radical restructuring of the database to increase efficiency.

In addition to general backup files, it is also necessary to keep copies on a Journal File of all the transactions on the database. The entry on the journal file can be made either before the update or after the update when the altered page is being written back to the database or, more probably, a combination of the two. Generally, the journal file is made on tape, which means that it will be quite slow during recovery and is a major limiting factor on the speed of recovery. DMS 1100 [29] allows the Database Administrator to specify that copies will be made on a Random Access file which clearly greatly speeds up the recovery operation. On the other hand, Random Access storage devices in the past were more liable to suffer hardware failures then sequentail devices, though this is becoming less true.

The final aspect of backup and recovery systems is the checkpoint. When a checkpoint is made, a copy of central storage is made and the position on the journal files marked. Checkpoints can be initiated either by the DBMS, e.g. at the start of a run-unit or from within the application program, e.g. at the start of an update. The use of checkpoints enables rollback and recovery to take place automatically and quickly. It is preferable for checkpoints to coincide

with quiescent points, i.e. points at which there is no transaction active. The consequences of inadequate backup and recovery measures are potentially so serious that the provision of full facilities is becoming one of the most important aspects of DBMS design [30].

3.4.5 Consistency of multiple copies of data

It was stated in Chapter 1 that an important aim of the DBMS is to control data redundancy, i.e. the unecessary duplication of data in the database. It should be noted, however, that it is sometimes desirable to incur the overhead of the extra storage required by repeating a data field in order to greatly increase efficiency.

McCall in the Infotech Report [28] quotes the example of where it is much cheaper to duplicate customers' names and addresses at a cost of £7000 extra for the second record rather than to incur the cost of the extra processor usage which would be required to obtain the information from two different places.

The problem with data redundancy in DBMSs, just as in the older systems, is the difficulty of ensuring that all copies of the field in question are the same at any time. If they are not identical, then there may be no way of telling which copy is the correct one. Thus if one copy of a duplicated field is updated, all other copies must also be updated automatically and 'simultaneously'. The question of the consistency of multiple copies of data therefore becomes a question of consistency during a group update, which was discussed in Section 3.4.3. Thus all duplicated fields

must be locked together. The user (application programmer and high level user, levels 4&5 of Figure 3.1) should of course be unaware of the chain reaction of his update which will be carried out automatically by the system.

3.5 Privacy

The main threat of computers as seen by the layman is their use in establishing huge databanks in which all information on an individual is integrated. This information would be gathered from many different sources, e.g. bank, income tax, mortgage companies, job applications, police, educational institutions and so on. Thus the provision of adequate privacy controls becomes of vital importance to the designer of the DBMS. Even the most elementary controls are going to cost something, both in real terms and in terms of performance. The analogy can be drawn with the physical protection of valuables - the more valuable the items, the stronger the safe used and the more elaborate the security arrangements. Similarly, it can be expected that the more sensitive the information stored in the database, the more expensive the provision of security controls will be.

Before the teleprocessing era, the provision of security for a computer system was really simply a question of ensuring the physical security of the computer room and associated disc and tape libraries. Modern teleprocessing systems are much more vulnerable. Apart from the difficulty of ensuring the security of hundreds of terminals, sophisticated bugging

devices enable the communication lines themselves to be tapped. Assuming therefore that the snooper manages to log on to the system, the next line of defence must come from the DBMS itself. The final line of defence is the Operating System and hardware. If the DBMS provides a high degree of security, then the skilled, professional spy will attempt to bypass the DBMS and possibly also the Operating System to gain access to the database. To frustrate such spies an elaborate code could be used to encode the data when it is stored and then decoded by the DBMS when the data is retrieved. In this way, meaningful access would be expensive other than through the authorized DBMS routines. The code used must change in an unpredictable way because the longer the code is in use, the greater the chance of someone breaking it and the greater the gain for him if he succeeds.

No matter how secure the system may be, it is important to make provision for the detection of anyone who does succeed in accessing the database illegally. In order to do this, it is essential to maintain an activity log of all events on the system, which is regularly and fully analyzed.

3.5.1 Terminal security

Terminals connected to the DBMS could be kept locked with keys or access cards held only by authorized personnel. To log on to the system, users would be required to give a password, which would either not be displayed at all at the terminal or else be overtyped. Such an approach has the merit of being cheap, but it would only be effective against the curious snooper and not the skilled professional.

To frustrate the line-tappers all data using the communication links to the DBMS could theoretically be encoded by a hardware device at the terminal and then decoded by a reciprocal device at the computer. It would also be possible to store all the data in the database in coded form. However, the problem of how to distribute the current encryption key securely over an entire teleprocessing network is far from being satisfactorily solved.

3.5.2 Physical data protection

The most straightforward case of data protection is to ensure that no-one accesses those fields for which they have no right of access, i.e. physical data protection. There are a number of different approaches to this problem:

(a) DBMS can maintain, as part of the Data Definition Language, a list of authorized users of each field/record; if a user's name is not on the list then the DBMS will not allow him to access the field/record (or the inverse of this specifying the range of permitted access for each user)

(b) each sensitive field/record can have a lock associated with it and those wishing to access it must first give the correct key; again this would be specified in the DDL

(c) execution of a database procedure to determine whether or not the user is permitted to access the field/record.

In the case of databases which are stored on removable devices, e.g. tapes, discs, header labels can be checked for access permission. This would also ensure privacy in the event

of an operator accidentally mounting the wrong tape or disc.

3.5.3 Logical data protection

An increasingly important aspect of protection to which little attention has been paid is that of logical data protection. It is possible to have a situation whereby a user is permitted to access the name field in the personnel record and the salary field in the payroll record, but he would not be permitted to link the two fields together, i.e. he would not be able to find out how much a particular person earns. Even if database procedures were available to monitor a user's activities, it might still be possible for him to list the two sets of data and associate them outside the system using his knowledge of the real world. The DBMS could not reasonably be expected to do anything about this.

CHAPTER 4

CONCURRENT UPDATE IN DATABASES

4.1 Introduction

The difficulties which arise when more than one run-unit is concurrently updating the database were explained in Section 3.4.3. In this chapter, the general aims to be achieved by a solution to the update problem will be discussed and the approaches taken by some existing and proposed systems will be examined.

4.2 Guidelines for solution to concurrent update problem

The following is a list of the desirable attributes of a solution to the concurrent update problem (see also [31]); note that these attributes are ideals and not necessarily simultaneously realizable as is discussed in Section 4.2.1.

(a) The basic aim of a solution to the concurrent update problem is to detect and avoid interference between concurrent users of the database. This must be totally transparent to the users and must give each user the illusion that he alone is accessing the database - or at least the portion in which he is interested. Thus solutions of the type which inform a user that a record has been changed by another user since he first read it are unsatisfactory.

(b) Users should have the illusion that they are permitted free

and unrestricted access, both for reading and writing, to those portions of the database in which they are interested, subject, of course, to any privacy constraints. Users should ideally not have to specify in advance what operations they wish to perform. For example, they should be allowed to step unconstrained through the database reading and updating records.

(c) Solutions which necessitate rollback are unsatisfactory in an on-line environment due to the unrepeatability of the work. The exception to this is any system in which processes are not updating the database when they are pre-empted or rolled back. If an actively updating process is rolled back through a change in data which might affect the decisions made by other users accessing that data, it is possible that these users would not still be logged on to the system. Rollback in a batch environment, however, is quite satisfactory; the user simply indicates the beginning and end of his group updates and need not be aware of whether or not rollback has taken place. Using a differential file and resetting all local and global variables, the system rolls the process back to the start of the update. Such an approach can be useful to the programmer in that it could be used to initiate a voluntary rollback in the event of an error being detected.

(d) The solution must guarantee that all users will eventually be able to run. If a user's resource demands are considerable, he may have to wait until there are virtually no other users of the system. Such users effectively

run their programs in batch mode, when, in general, the problem of concurrent update does not arise. If, however, the transaction to be performed is urgent (e.g. flight cancellation) the demands must be met quickly and therefore a priority system may be required.

(e) The solution must not involve too high an overhead especially for simple operations. In many applications, updates are simple in structure and involve only a single record, i.e. group updates are comparatively rare, although this may well be because they are difficult to program.

(f) Only those records which are logically involved in the update should be locked, i.e. a process should claim and be given no more resources than it actually needs and should release them at the earliest possible moment consistent with the logic of the update.

4.2.1 Discussion of the requirements

The requirements listed above are not logically compatible. The aim of giving each user apparently his own view of the database while at the same time maximizing the concurrency are in effect contradictory. If only one user at a time is accessing the database, then he can simply read and update records freely, even for group updates. However, once other users are allowed to access the database at the same time, interference can easily occur as illustrated in the examples in Section 3.4.2.

In order to avoid possible interference between concurrent updaters of a database, it is necessary to ensure that they are accessing disjoint portions of the database. However,

requirement (b) stipulates that users should not ideally have to specify in advance what their access requirements are. Thus the DBMS would have to examine each users demands in order to ascertain whether they overlap with another concurrent user. It is not possible for the system to deduce the individual record occurrences required by a user (especially when requests are content-based) without actually executing the user program. Thus the DBMS could only deduce the user's requirements in broad terms, e.g. realm or record type, from the subschema DDL and/or declaratives in the application program. Even if the user simply wished to update a single record, the system would only be able to state in advance that the program would require exclusive access to, for example, all the record occurrences of that type or all the records in the realm in which the desired record is located. It would therefore issue locks on that basis. A concurrent user wishing to update a single different record in the same realm, or of the same type, would therefore have to wait until the first user terminated. Such an approach runs contrary to requirement (f) which states that no process should be given more resources than it logically needs and that it should not retain those resources for longer than necessary.

Thus at the very least the DBMS must know before a process reads a record of its possible intention to subsequently update the record. However, this is not sufficient since even this information is not enough for the system to guarantee no interference between users. Hence a system of locks is introduced which must be claimed by a process prior to reading

a record which it intends subsequently to update. This lock can be claimed by the process explicitly using a special LOCK command or automatically by the system when the process issues a special type of READ (e.g. GET HOLD in IMS). This approach works well when users are restricted to claiming a single record at a time, i.e. they must release a lock prior to obtaining the next one. This is not an unreasonable restriction for some users, but it is totally impractical for the remainder who perform group updates. To handle group updates, it is necessary to allow processes to hold more than one lock at a time and to release them separately or all together. However, if the user is allowed to step freely through the database claiming locks and updating records, deadlock can easily occur. Requirement (c) prohibits solutions of this kind.

It is therefore necessary to compromise even further in order to perform group updates successfully. Users must specify in advance all their requirements which form part of logically consistent updates.

4.3 Existing approaches to concurrent update

In this section the solutions adopted by CODASYL, IMS/2, DMS 1100, PRIME and the proposal by Chamberlin et al in [32] will be discussed.

4.3.1 CODASYL

CODASYL allows for two levels of locking - at the area

level (using DML OPEN command with qualifiers) and at the record level (DML KEEP/FREE commands).

A run-unit may open an area for EXCLUSIVE use - either update or retrieval - which prohibits all other users from accessing the area for the duration of that run-unit or until it issues a CLOSE on that area. The KEEP command on a record is used to notify the DBMS of the intention of the run-unit to re-access that record. While a KEEP on a record is in force (i.e. until a corresponding FREE is issued), any attempt by that run-unit to update the record will be successful only if the record has not been changed by other run-units since the KEEP was issued. Such a system is clearly easy to implement but it places the onus entirely on the user to decide what action to take if the update is unsuccessful. This system has been generalized since the April 71 Report to recognize two modes:

(a) monitored mode

(b) extended monitored mode.

Only the current record (i.e. the record most recently accessed) of a run-unit can be in monitored mode, but any record (including the current) can be in extended monitored mode. The current record is placed in monitored mode automatically and remains in this mode until it ceases to be the current record or is the object of a REMONITOR statement. The execution of a KEEP statement on the current record of the run-unit alters its mode to extended monitored mode. Extended monitored mode continues until a FREE statement removes the record from that mode or a REMONITOR statement references the record or the realm in which it is stored is removed from the ready

mode. The purpose of a REMONITOR statement is to alter the records currently in extended monitored mode and to ensure that the current record continues to be monitored even after it ceases to be the current record of the run-unit.

Although this system is more precise than the straightforward KEEP/FREE of the April 71 Report, the effect from the user's point of view is the same; namely, the user is notified if a monitored or extended monitored record is altered by a concurrent run-unit since the record entered monitored or extended monitored mode. It should be noted that, as with many other aspects of CODASYL, the role of the KEEP/FREE command is under review.

The use of the area locking mechanism can lead to inefficient sharing. Although, in theory, the records involved in group updates (i.e. inter-dependent updates of a number of records) should be located in the same area, in practice, with large databases and many users with conflicting requirements for record placement, this may not be possible. Hence one run-unit could lock a single portion of the database even if it was only updating one record, which particular record depending on several other records in different areas, all of which would have to be locked together.

4.3.2 IMS/2

IBM's IMS/2 [23,24] in a sense avoids the problem of concurrent update altogether by simply restricting concurrent usage of the data to disjoint portions of the database. This is based on the specification of segment sensitivity for each run-unit in the Job Control Language (see Section

3.2.2). If a run-unit has indicated that it intends to delete or update a segment which another run-unit has also indicated it may wish to delete or update, then IMS will ensure that the two programs will not be scheduled together (cf. CODASYL OPEN command for areas). This approach greatly limits the degree of concurrency in the system since even if the two run-units only have one segment occurrence to be updated in common, the second run-unit will have to wait until the first one has terminated.

Under IMS/2 the DML HOLD option on retrieval requests is redundant, but under IMS/VS it will enable locks to be applied at block level. A locked block being updated will not be released and written back to the database until a FREE command is issued or the run-unit terminates. This system can give rise to deadlock which will be resolved by rollback of one of the run-units involved.

4.3.3 DMS 1100

UNIVAC's DMS 1100 [29] implements the area locking mechanisms as proposed in the CODASYL April 71 Report. However, the operation of the DML KEEP/FREE commands is slightly different. The KEEP statement places a lock on a page of the database and the FREE command releases it. While one process holds the lock for a page, other processes cannot access it.

Deadlock can occur and a rollback mechanism is put into operation when it is detected. The user specifies in his application program a rollback paragraph which must

be executed when rollback is required. Only a single process is rolled back and its effects on the behaviour of other processes is not considered. Furthermore, if no entries on the random access Audit Trail (quick-before-looks) have been specified in the schema for the areas involved, then the database can be left in an inconsistent state following rollback.

4.3.4 PRIME

Although based on the CODASYL DBTG proposals, the PRIME DBMS [33] takes an individual approach to the concurrent update problem. PRIME introduces a unit known as an update Database Transaction (DBT), which is initiated by the application program by means of a START TRANSACTION command and terminated by an END TRANSACTION command or an ABORT TRANSACTION command. All logically related updates are grouped together into an update DBT. The system makes use of before-images of blocks which are taken before each block is updated. These before-images can then be used to rollback the transaction when the user aborts the transaction or when it is aborted automatically. If a user attempts to read or write a block that has been modified by a concurrent update DBT, the system will order him to abort his transaction. If the user complies, using an ABORT TRANSACTION command, he may perform his own recovery before aborting, but if he fails to comply, the system will abort the transaction automatically. It is felt that concurrent conflicts are transient and usually clear quickly. Hence, after aborting a transaction, a user can start a new update DBT immediately and try again.

4.3.5 Chamberlin et al's solution

At the 1974 IFIP Congress a paper was presented by Chamberlin, Boyce and Traiger [32], in which a deadlock free scheme was put forward as a solution to the concurrent update problem. The main complications attributable to resource-sharing in large databases are seen as:

(a) non-unique resource names

(b) non-static resource categories - a process operating on a resource may change its nature

(c) interdependent locks - further lock requests may be issued on the basis of the first set of lock requests

(d) increased complexity - to maximize concurrency the basic lockable unit must be small, e.g. a record, but this approach implies millions of lockable resources.

In their solution Chamberlin et al assume the existence of SEIZE and RELEASE primitives in the application programming language. The code between the SEIZE and its END statement is known as a seize block. Within the seize block, no procedure can be carried out except the claiming of records - in particular, no changes can be made to the database. It is also permissable to issue lock requests which are dependent on the data values of records.

```
SEIZE;
  X=EMPLOYEES WHERE SALARY >'10000';
  DEPARTMENTS WHERE DEPTNO=X.DEPTNO;
END;
```

The reason for the restriction on the type of operation which can be carried out within the seize block is obvious - namely, that a process can be pre-empted safely within this block without affecting other processes. Once outside its

seize block, a process cannot be pre-empted and has exclusive access to the records it has locked. It relinquishes all locked records simultaneously using the RELEASE statement. In this way changes it has made to the database will appear as a single logically-consistent unit. Clearly, all records locked in a seize block must be released before the next seize block is entered.

The algorithm for locking records envisages a search engine which can examine records and set locks on the ones which qualify. It can also examine the non-updated version of locked records. If the search engine for one process wishes to lock a record which is already locked by another process, the requesting process is said to be blocked and must wait until the record is released. For every record, there is an ordered queue of processes - the process at the head of the queue holds the record and the remainder are blocked waiting for it.

Clearly, it is possible for deadlock to occur. Chamberlin et al say that this can be detected easily using King and Collmeyer's method [27] and can be prevented by defining a priority ordering among processes. Thus if P1 requests a record held by P2, the record is pre-empted if and only if P1 has higher priority than P2 and P2 is still in its seize block. Record queues are held in priority order. However, such a scheme can lead to unnecessary pre-emption and it would be better to pre-empt only when deadlock has actually occurred. To avoid the possibility of one process being blocked indefinitely, it is possible to favour a process in such a way as to guarantee it will run. It should be pointed out that

the favouring of a process involves a further overhead for the algorithm. Chamberlin et al propose the following modification to their algorithm:

(a) when process P1 requests a record which is locked by P2, the record is pre-empted if and only if:

(i) P1 is favoured and P2 is blocked or

(ii) P2 is not favoured and P1's queueing behind P2 would result in deadlock

Otherwise P1 queues immediately behind the favoured process P3, if and only if P3 is on the queue, else immediately behind P2

(b) When a process requests a free record, it is immediately granted a lock and placed at the top (holder position) of the queue for that record

(c) When a process P1 becomes blocked, it releases to the favoured process P3 all of its records for which P3 is queued and places itself next in line for these records

(d) When a process becomes favoured then wherever it appears on the queue, it moves to the top of the queue if the record is held by a blocked process, pre-empting the record, or to the second position in the queue if the holding process is not blocked (it could be outside its seize block)

(e) A record when released is given to the next process in the queue.

When a process wishes to release its records, it must wait until all other processes are either blocked or outside their seize blocks in order to ensure that a consistent view of the database is always available to all processes.

Chamberlin insists that if two processes A and B are simultaneously updating records of the same type that the 'snapshot' obtained by process A will reflect either all of the updates made by process B or none of them. All the updated records are checked against the locking predicates of blocked processes. Two situations are of interest:

(a) One of the newly released records may be found to meet the locking predicate of several processes, Pi. In this case to avoid deadlock, a total ordering of processes is generated which is consistent with all the existing queues. The processes Pi are placed on the queue for the newly released record in positions consistent with the total ordering

(b) One or more of the blocked processes may be queued for a newly released record, but may now discover that it no longer meets their locking predicates. These processes delete themselves from the queue for the record.

In both the above situations, the interdependencies of the locking predicates may necessitate re-examination of all the predicates and pre-emption of all the records held by that process as it is rolled back to the start of its seize block.

4.4 Summary of approaches to concurrent update

An examination of the approaches given above to the concurrent update problems reveals that they fall into two categories - minimum locking and over-locking. Sometimes a system uses a combination of these two approaches. Minimum

locking involves only those records which are logically involved in the update and over-locking involves (in general) locking more than is necessary, but which is easier to implement.

4.4.1 Minimum locking

The critical feature of minimum locking is the type of locking predicates which are allowed. If these are restricted to specific identification of records by means of database key, then the system is easy to implement and operate. The important aspect of this restriction is that the set of records requested is invariant, i.e. it does not depend upon the state of the database.

If, however, time-varying locking predicates which are dependent on the content of the database are allowed, the problem is infinitely more complex. For example, requests of the type:

LOCK EMPLOYEE RECORDS WHERE DEPARTMENT=X

will depend upon which employee records have department=X at a given time. Requests of this type are quite reasonable and should be handled by the system.

In order to evaluate such locking predicates, a time-consistent snapshot of all the records involved is required. However, it takes a finite length of time to evaluate the locking predicates. This time can be considerable when requests of the form:

LOCK PATIENT RECORDS WHERE SYMPTOM=Y

are made and there is no inverted symptom file. It is worth noting that if, as seems likely, CODASYL provides for non-

disjoint realms, then with locks applied to complete realms, requests of this form could be handled quite efficiently. While the process is evaluating the locking predicates, other processes can be making changes to the database which might affect the evaluation. If the process is restricted to examining only those records which are not currently locked and if the locking predicates have been correctly written to include all records which are logically involved in the update, then in theory there should be no problem. Practically, however, this means that in the case of a symptom request of the type given above, the locking predicate would fail even if only one patient record in the entire database were locked. Thus the entire locking predicate would have to be re-evaluated. Clearly, with this type of request, it would be much more sensible simply to keep track of all the patient records with SYMPTOM=X and check each newly updated record as it is released, until all patient records have been examined.

In general, a process whose locking predicates are content-dependent can be thought of as tracing a time-varying path through the database from record R1 to Rn. Having reached a record node Ri, the path to be followed from Ri, i.e. the next node Ri+1 to be selected, depends on the value of a field in Ri, or, more generally, on $\{R1,R2,\ldots,Ri\}$. Thus the entire time-varying set of records $\{R1,R2,\ldots,Rn\}$ are logically involved in the update and must all be locked by the process before it can be released. Clearly, with an operation of this type, it is not possible to continue the evaluation of locking predicates once a locked node Rj is reached. If records $\{R1,R2,\ldots.Rj-1\}$

have been locked as each node is reached, then this set is still valid and can be retained by the process until Rj is released and the locking predicate evaluation continued. With a long and complex path through the database, a process could be locked for a very long time, while at the same time, preventing other processes which might require a single record from the locked set $\{R1, R2, \ldots . Rj-1\}$ from being released. To avoid this, it is preferable not to lock the records $\{R1, R2, \ldots\ldots, Rj-1\}$ as the path is being traced through the database. However, in this case once a locked record Rj is reached, the process is blocked and the records $\{R1, R2, \ldots., Rj-1\}$ can be released to other waiting processes, if required. In this way, the entire path from R1 would have to be re-evaluated since any change in record $Rf \in \{R1, R2, \ldots\ldots, Rj-1\}$ may well affect $\{Rf+1, Rf+2, \ldots, Rj-1\}$ such that a different path will be followed.

It is important to realize that in complex path tracing algorithms, the logic required in the seize block will probably have to be repeated again outside the block when the records are actually being processed and updated.

4.4.2 Over-locking

The essential feature of over-locking is that it involves locking more than is actually required and therefore the IMS scheme based on segment type and the CODASYL scheme based on areas are examples of this type.

How efficiently the area locking mechanism works is entirely dependent on how close the areas are to those portions of the database used by individual application programs.

The CODASYL areas physically resemble the files of traditional data management systems and therefore there could be a good correlation between areas and portions of the database required by particular application programs. However, one of the fundamental reasons for the introduction of the DBMS was to eliminate the unnecessary redundancy in the traditional multiple file systems. In general, it was standard practice to design the files such that the payroll program used one or two files, the personnel program another file and so on, even though there might be considerable duplication of information (e.g. employee name and address) across files. Given therefore that all these files are merged into a single database with the elimination of most of the dulicated data and that the database is divided into non-overlapping segments, it is unlikely that these segments will correspond neatly to the original files.

An alternative to the area as the basic locking mechanism is the record type. On consideration of traditional filing systems in which each file was composed of a single record type, this approach may well be quite logical. Thus, for example, the payroll program will be concerned with the payroll record, the personnel program with the personnel record and so on. Clearly, in order to avoid the disadvantage of non-overlapping areas, it is the logical record type which is used. It is the responsibility of the DBMS to translate this into one or more physical record types. A queuing mechanism will avoid deadlock - if an application program required more than one logical record type, it must claim them all together. All processes are guaranteed to run although

complex path tracing algorithms involving many different record types may well have to wait a long time before being released.

CHAPTER 5

THE CODASYL PROPOSALS

5.1 Introduction

It is undoubtedly true that the publication which has had the greatest impact on the field of Database Management Systems is the 1969 Report of the CODASYL Data Base Task Group [34] together with its sequel, the April 1971 Report [1]. CODASYL (Conference on Data Systems Languages) is a voluntary organization composed mainly of users and implementors and was set up in 1959. It is this organization which was responsible for the development of COBOL. One of its three main committees, the Programming Languages Committee (PLC), is concerned with approving changes to COBOL. The Data Base Task Group (DBTG) was a sub-committee of the PLC. The April 1971 Report of the DBTG was intended to discuss enhancements to COBOL to incorporate more sophisticated data management facilities. The report has since been reworked with various modifications and incorporated into the COBOL Journal of Development [35] and the CODASYL Data Definition Language Committee (DDLC) Journal of Development, 1978 [22]. In spite of this, the original 1971 report and its subsequent alterations in the JODs is seen, if not as a proposal for a DBMS, at least as a discussion of the sort of facilities a DBMS should be expected to handle. Above all, the 1971 Report provided a clear and well-defined framework as well as a terminology in which to discuss DBMSs. It is not proposed

to describe the CODASYL proposals in detail here, but rather to give a brief description of them and then to discuss some aspects more fully.

The CODASYL view of DBMSs is a continually evolving process with many working parties which examine all the various aspects in detail and make recommendations for changes to be made in the two Journals of Development. Apart from its great initial impact, the dynamic nature of CODASYL has maintained its vital role in the field of DBMSs today. However, the ultimate aim of CODASYL is to provide a "standard". The field of DBMSs is still developing rapidly and to impose a standard which necessarily has to be fairly static, could be detrimental.

5.2 Elements of the CODASYL Proposals

The two main elements of the CODASYL 1971 Proposals are:

(a) Data Description Language (DDL)

(b) Data Manipulation Language (DML)

A third language, the Device/Media Control Language (DMCL) is also briefly mentioned. The DMCL provides the mapping between the physical database and the physical storage devices, whereas the DDL and DML are concerned mainly with the logical database.

5.2.1 The Data Description Language

The Data Description Language is used to describe the

data and the relationships between the data at two distinct levels - the schema and the subschema. The schema is seen as a logical description of the entire database, i.e. of all the data items, records and relationships between them (sets in CODASYL terminology). The subschema, on the other hand, is a description of only a portion of the database as required and viewed by a particular application. Thus each application has its own subschema. The subschema is really just a subset of its parent schema, since it may differ from it in only relatively minor ways, e.g. the omission or renaming of certain areas, records and sets (see Section 5.2.3 for definitions of these terms) and the ordering and/or characteristics of data items within records.

The subschema is host language dependent at least at the data item level. Thus each host language, e.g. FORTRAN, PL/I, COBOL requires its own subschema DDL, e.g. FORTRAN subschema DDL etc.

5.2.2 The Data Manipulation Language

The Data Manipulation Language is the language used to access the database. It consists of a variety of commands embedded in a host language. Initially, COBOL was the only host language which was discussed in any detail but since then a FORTRAN DML JOD has been published [36].

5.2.3 Data structures

The smallest unit of named data in the CODASYL proposals is the data-item; an occurrence of a data item is a representation of a value. A data-aggegrate is a named collection of

data items within a record. There are two types - vectors and repeating groups. A vector is a one-dimensional ordered collection of data-items, all of which have identical characteristics. A repeating-group is a collection of data that occurs an arbitrary number of times within a record occurrence and may consist of data-items, vectors or repeating groups. A record is a collection of zero, one or more data-items or data-aggregates and is the basic addressable unit in the DBMS. There may be an arbitrary number of occurrences in the database of each record type specified in the schema for that database. In the April 1971 Report, each record has a unique identifier called a database key, which is assigned when the record occurrence is first stored in the database and remains its permament identifier until that record occurrence is deleted. Database keys are assigned by the system according to rules specified for that record type in the schema and arguments, if any, supplied by the process adding the record occurrence to the database. The keys are available to the program. In the latest DDLC JOD 1978 [22], database keys are for system use only and are no longer accessible to the application program; they are in use for the duration of the program and not throughout the life of the record.

A set is a named collection of record types. As such, it establishes characteristics of an arbitrary number of occurrences of a named set. Each set type specified in the schema must have one record type declared as its owner and one or more record types declared as its member records. Each occurrence of a set must contain an occurrence of its owner record and may contain an arbitrary number of occurrences

of each of its member record types. An area is a named subdivision of the addressable storage space in the database and may contain occurrences of records and sets or parts of sets of various types. Areas may be opened by a program with USAGE MODES which permit or do not permit concurrent programs to open the same area. Since the April 1971 Report, the area has been complemented by the realm and the storage-area. A realm is a logical subdivision of the database and the storage-area is a subdivision of physical storage.

A database consists of all the record occurrences, set occurrences and areas which are controlled by a specific schema.

5.2.4 The set concept

The CODASYL set concept has already been discussed as an example of a network data structure in Section 3.2.1. It is interesting to note that although many aspects of the April 1971 Report have been changed or modified, the set has remained intact.

The April 1971 Report describes two different modes in which sets can be implemented, namely CHAIN and POINTER ARRAY. The members of a chained set are linked together by a system of pointers known as NEXT pointers, which starts with the owner, then passes through each member in turn and ends with the owner as shown in Figure 5.1.

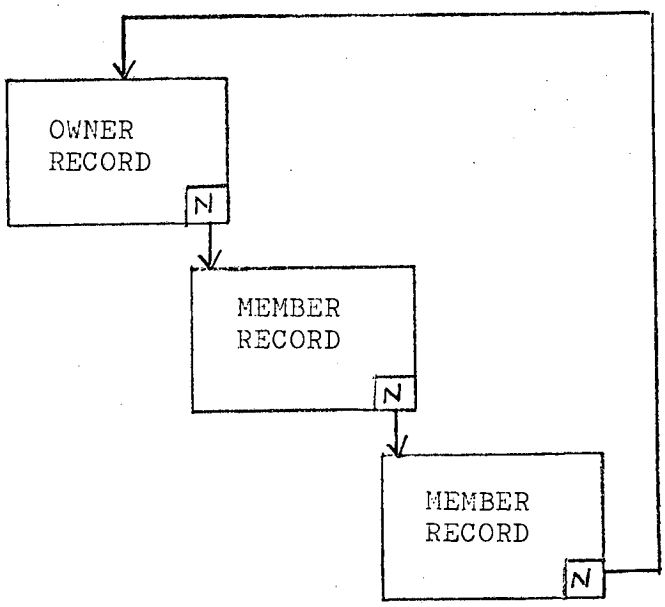


Figure 5.1 A chained set with NEXT pointers

In addition to NEXT pointers, the LINKED TO PRIOR option can be used to include PRIOR pointers to link backwards as well as forwards through the set. Finally, each member can be linked individually to the owner using the LINKED TO OWNER option. Figure 5.2 shows all the possible pointers.

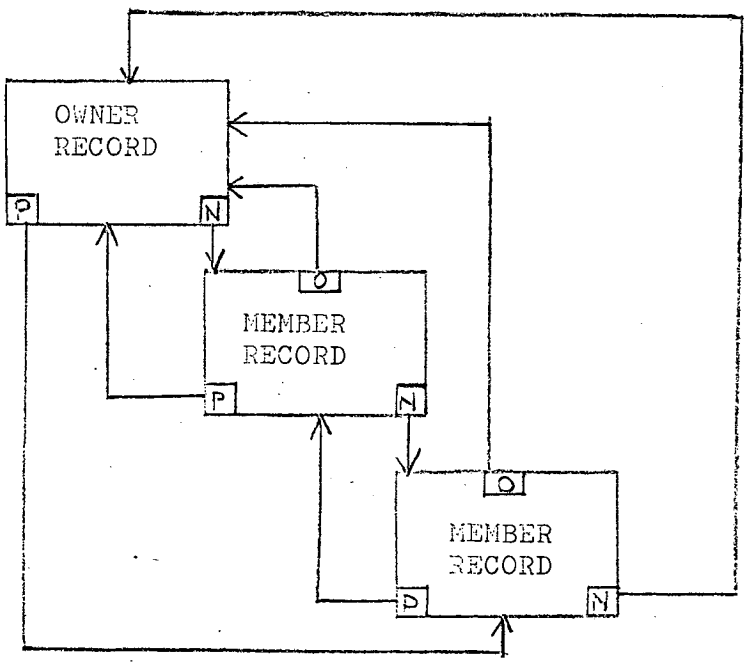


Figure 5.2 A chained set with NEXT, PRIOR and OWNER pointers

In the POINTER ARRAY mode, the NEXT pointers are stored not in member records but in the owner records; the only pointers allowed in the records themselves are the pointers to the owner as shown in Figure 5.3.

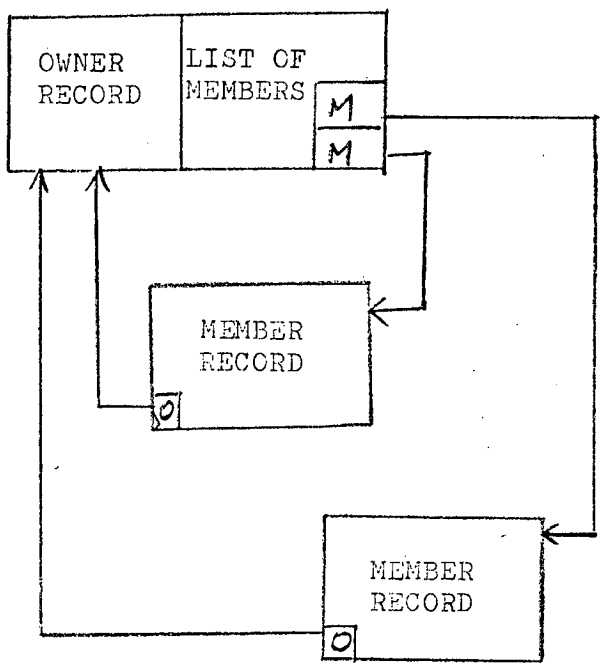


Figure 5.3 Pointer array set

One of the most difficult features to understand in the CODASYL April 1971 Report is the SET OCCURRENCE SELECTION clause of the DDL. It is this clause which governs how the particular occurrence of a set is to be selected from all the other occurrences of the set. A set can be identified by its owner record, so assuming the owner can be located directly, then the appropriate set is selected. Alternatively, the CODASYL currency indicators can be used. CODASYL maintains several currency indicators during database processing which show which occurrence of each area, set type or record type was last accessed. Thus the current occurrence of the

particular set is the one to be selected. Apart from the hierarchical relationship within sets between owner and member records, it is clearly possible for sets themselves to be organized in a hierarchy. Thus a member record of one set becomes the owner of another set one level down the hierarchy. A third method of set selection depends on selecting the root set (using either of the methods above), set 1, and providing sufficient identifiers to trace down the hierarchy from set 1, set 2, ... in such a way that the owner of set 2 is a member of set 1 etc., until the required set is found. The decision of which method of set selection to be adopted must rest with the Database Administrator and the application programmer's task is to supply the necessary parameters to the database procedure.

5.2.5 The storage-schema and Data Storage Description Language

A significant structural development from the April 1971 Report to the present CODASYL position is the introduction of the storage-schema in the 1978 DDLC JOD [22] and the Data Storage Description Language by the Database Administration Working Group (DBAWG) [37 & appendix to 22]. The subschema is the application programmer's view, the schema is the Database Administrator's logical view and the storage-schema is the DBA's physical view. The storage-schema would be written in Data Storage Description Language (DSDL) and is used to describe a storage environment for a database and an associated schema to storage mapping. The schema is defined first and it describes all the data in the database. A subschema describes a local view and the mapping between that

view and the schema. The storage-schema defines a physical view and defines a mapping between this view and the schema. Since both subschema and storage-schema map on to the schema, the subschema to schema mapping is independent of the schema to storage-schema mapping and application program independence from storage structure may be improved.

A storage-record is a variable length record which is stored physically contiguously within a page of a storage-area. A storage-area can be considered to consist of both an integral number of pages and an integral number of storage-records. A storage-record is of variable length. Thus a single schema record may be mapped directly onto a storage-record or several schema records may share several storage-records. It would also be possible for a schema-record to span several storage-records. The particular mapping chosen would depend upon consideration of storage and retrieval efficiency. The flexible nature of the mapping (both one-to-many and many-to-one) means that schema records may be designed without considerations of the efficiency constraint that they be stored as a single unit. Hence the schema records may be designed according to the logical application requirements.

In addition to the DSDL, DBAWG have also described other extensions to the original April 1971 Report concerned with data administration aids. These include facilities for integrity control, gathering statistics on database use and restructuring and reorganization of the database [37].

5.3 An assessment of the CODASYL Proposals

The CODASYL April 1971 DBTG Report was intended as a set of very carefully worked out proposals, which were to open to discussion and criticism. It is certainly true to say that it generated considerable interest and stimulated much debate on the subject of DBMSs. It is proposed in this section to present some of the criticisms which have been made of the report.

5.3.1 The AREA concept

The April 1971 Report outlines possible uses of an area as:

(a) a means whereby the Data Administrator could conveniently subdivide a larger database into smaller and more manageable sections - this can be exploited for selective duplication, backup and recovery

(b) the placement of complete areas can be controlled in order to lead to more efficient storage and retrieval - an unused area could, for example, be stored off-line in archival storage.

The strong association with the physical storage structure (e.g. (b) above) points to the traditional file concept. For example, in the DMS 1100 implementation of the CODASYL Proposals [29], areas have a one-to-one relationship with the standard Operating System file.

Apart from its storage role, the area also acts as the basic access and locking mechanism. The choice of the area to fulfil this role undoubtedly makes the writing of

application programs more difficult. The WITHIN clause, which defines in which area a record occurrence is to be placed, allows for more than one area to be specified for a single record type, the actual area name being given by the value of the data-base-area-name when the record occurrence is being stored in the database. For example, it is required to retrieve record occurrence R, which was defined as being stored WITHIN AREA-A or AREA-B. Prior to executing the FIND command, the program must initialize the data-base-area-name to either AREA-A or AREA-B. In order to do this, the programmer must know in which of the two areas the record R was actually placed when the STORE command for R was originally issued. It should not be necessary for an application programmer to know such details.

Considering now the use of areas as the basic locking mechanism of the DBMS; it is clearly wasteful for a run-unit to have control over more of a resource than it actually needs, although it can be safer for group updates. By requiring a run-unit to lock at the area level, it can therefore have control over the whole area even though it may only be updating one record. This can lead to very inefficient sharing and limit concurrency. As was indicated in Section 5.2.3, the area has now been replaced by the realm and the storage-area. The role of the realm is still evolving, but it is possible that the final result together with the storage-area will remove the anomalies described above.

5.3.2 The role of schema and subschema

CODASYL regards the schema as a description of the

entire database and the subschemas as descriptions of portions of it required by various applications. The main objective here is to give the users access only to the data they actually require, both in order not to confuse them with irrelevant data and also to provide a certain measure of security. The hope is also that such a structure will provide a degree of data independence, i.e. that changes made to a database which do not involve the data used by particular application programs should not necessitate changes to those programs. However, Dee et al [28] found that as their CODASYL database grew and the schema was altered, programs had to be changed which did not use the new data.

Essentially, a CODASYL subschema consists of portions of DDL copied from its parent schema with a few minor alterations, e.g. privacy information, attributes of data items, method of selection of member records of sets (see Section 5.2.1). This is very restrictive. If, the subschema is intended to represent truly the view of the database by a particular application, it is not unreasonable to expect greater flexibility. It would be desirable to allow the user to define new sets in the subschema. Also the only major difference allowed between the subschema record and its parent schema record is the omission of certain fields in the subschema record. The order of fields may also be changed and the attributes of data items. It would be useful to be able to form new record types in the subschema whose fields may be drawn from a number of different parent schema records without restrictions. A natural extension to this new subschema record type would be to allow the definition of new sets

in the subschema. The ramifications if this are discussed in more detail in a later chapter.

5.3.3 Sets

In [38] Professor King cites the example of a restriction in the CODASYL Report on the use of sets. Consider a database containing peoples names and their interests. There would be two record types, PERSON and INTEREST and two set types; PERSON-INTEREST with owner PERSON and member INTEREST, which links one person to all his interests and the inverse INTEREST-PERSON with owner INTEREST and member PERSON, which links one interest to all the people with that particular interest (see Figure 5.4).

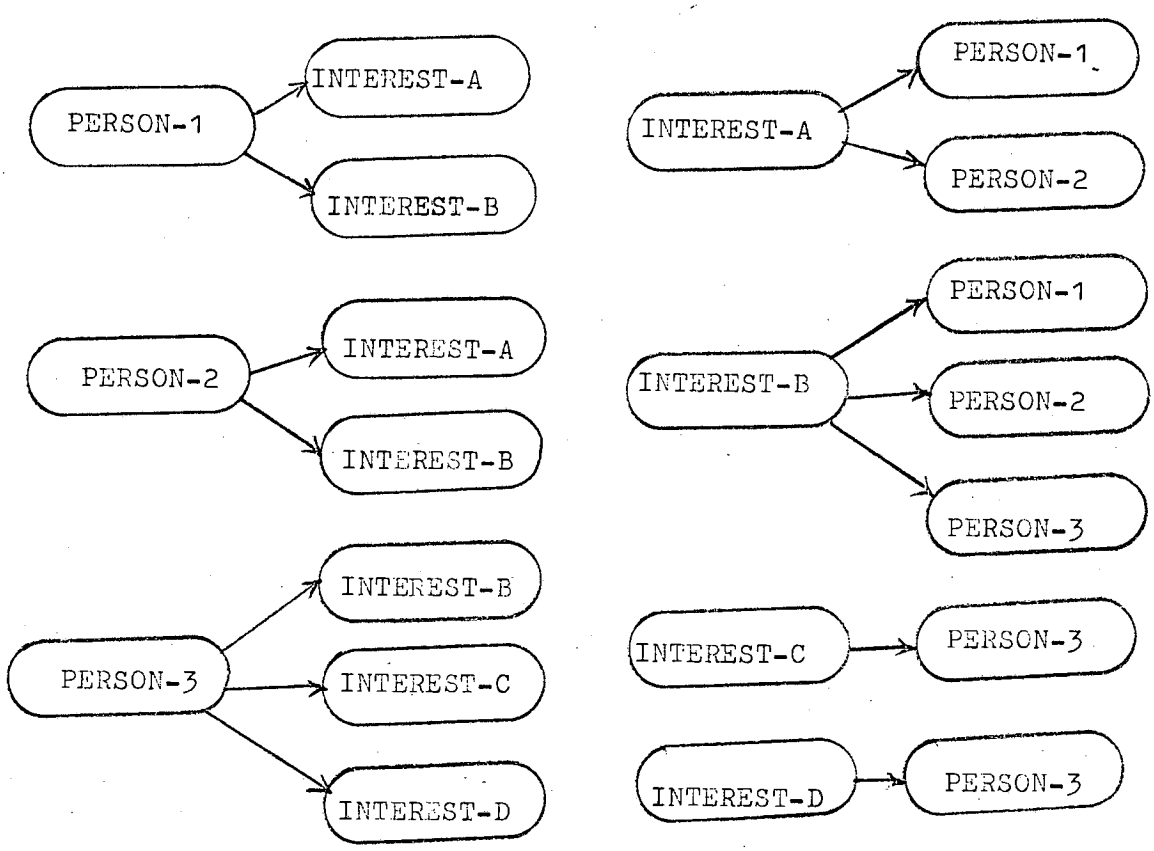


Figure 5.4 Occurrences of the PERSON-INTEREST set and the INTEREST-PERSON set

Take member record occurrence INTEREST-B (of the PERSON-INTEREST set) and it will be found to be a member of three occurrences of the PERSON-INTEREST set, owned by different owner record occurrences, PERSON-1, PERSON-2 and PERSON-3. The same is true in reverse in the INTEREST-PERSON set. Such a situation is expressly forbidden by CODASYL. The reason for this restriction is said to be that if member record INTEREST-B had been selected and the DBMS was then asked to find its owner, the system would not know which of these owners to choose. It has been shown in [38] that the problem can be circumvented by the introduction of a redundant relation record. Such a solution is not within the spirit of a DBMS which aims at the elimination of as much redundancy in the database as possible. A proviso should be added here that in some situations such a link record may have valuable significance and be an important part of the logical structure of the database.

5.3.4 Index structures

A major omission from the CODASYL proposals which has received widespread criticism is the lack of any provision for an index structure or associative mechanism. Such a facility would be based on records themselves using record keys and would be independent of how the sets themselves are chained together (see Section 5.2.4). Index structures such as the Index Sequential file organization, inverted files and associative mechanisms such as hashing techniques are well-known and widely used and could be employed to great advantage in a DBMS.

CODASYL does provide for a record location mode (CALC), a type of hashing function, which could be implemented as an Index Sequential organization, but the necessary removal (since the April 1971 Report) of database keys from the user's view, means that it would not be possible for an application program to exploit this knowledge. Alternatively, an index mechanism on a sorted system-owned set could be used to equate to ISAM.

CODASYL also makes no provision for the implementation of a content-addressing mechanism. The provision of such a facility is becoming increasingly important as users move further away from viewing data in terms of physical representation on storage towards seeing it in terms of its representation of the real world.

CHAPTER 6

VIRTUAL MEMORY AND DATABASE MANAGEMENT SYSTEMS

6.1 Introduction

During the last ten years there has been a general move away from conventional operating systems towards virtual memory systems. This trend is not so apparent in the literature on DBMSs (e.g. CODASYL proposals) and yet the type of operating system underlying the DBMS is of vital importance to the design and efficient operation of the DBMS. This chapter consists of a brief discussion of some of the aspects of virtual memory systems which are significant from the point of view of a DBMS.

6.2 Virtual memory systems

In the early days of computing the only memory device directly available to the executing program was main memory (core storage). The programmer therefore divided his program into a number of sections which would overlay one another in main memory. With the advent of high level programming languages and increasingly complex overlay strategies, an automatic storage management system became essential and a consequence of multiprogramming. The introduction of multi-programming systems with their associated problems of resource sharing, in particular, the memory resource, together with the desire to achieve independence for programs led to the development of a storage

allocation system which became known as virtual memory [39].

In a virtual memory system, the programmer has the illusion that he has available to him a very large one-level store, which appears to him as main memory. In fact, this virtual memory consists of a hierarchy of storage devices composed of main memory and usually magnetic drums and discs. All addresses references in the program are virtual addresses and it is only when the program is actually executing that the system translates them into physical machine addresses.

Of fundamental importance in a virtual memory system is the concept of a page, which is the unit of storage which is transferred between the levels in the storage hierarchy. Thus if an executing program requests a particular piece of data, the whole page on which the data is to be found will be brought into main memory. Clearly, the choice of page size is vital. A small page size could minimize the amount of unnecessary information brought into main storage, whereas a large page size could be more efficient [40].

Much of the literature on DBMSs and virtual memory systems is concerned with the effects of using buffer pools in an attempt to reduce I/O accesses to the database [41, 42, 43, 44]. These pools are commonly used in non-virtual systems by programs requiring a lot of I/O. Sherman and Brice [41] point out that an increase in the buffer space may cause a decrease in performance due to increased competition for real memory between program and buffer. They analyze the effects of different algorithms for buffer management and page replacement as well as the effects of varying the size of buffer space and real memory. The results are compared on the basis of

the cost of running a DBMS, where cost is defined to be the sum of the number of database faults and page faults. A database fault occurs when a requested database address is not found in the virtual buffer, while a page fault occurs when a requested virtual memory address is not found in real memory. The use of buffers in virtual memory systems can therefore give rise to a phenomenon known as double paging which occurs when a database request gives rise to both a database fault and a page fault. Sherman and Brice concluded that the advantages, in terms of increased efficiency, of virtual buffers can overcome the disadvantages of double paging resulting from their use.

A detailed study of the effects of different page replacement algorithms for relational databases has also been done by Casey and Osman [45].

6.3 Direct mapping of the entire database onto virtual memory

The theory and literature on virtual memory [e.g. 46] is mostly concerned with the analysis of program behaviour rather than data usage. The principle of locality, which has been observed experimentally, states that a program favours a subset of its pages and that this set of favoured pages changes membership slowly. The aim of the Database Administrator is to establish just such a locality in the physical mapping of the database to secondary storage.

A goal of a virtual memory system is to minimize the number of page faults (i.e. the number of times an executing

program requests a page which is not currently in main memory); each fault requires an access to secondary storage, albeit to a fairly fast device such as a magnetic drum. In the same way, a goal of the Database Administrator is to minimize the number of accesses to secondary storage.

In any DBMS, the method of mapping of the data to secondary storage is critical to the efficiency of the system. An intuitive approach to this mapping in a virtual memory system would be to map the database onto the whole virtual memory and leave the virtual memory system to handle the entire physical management of the data. There are four main reasons why such an approach would be undesirable:

(a) limitation of the size of the database to the size of virtual memory

(b) non-locality of access

(c) privacy constraints

(d) data integrity problems.

6.3.1 Database size

The database would be limited to the size of virtual memory less the space required by the program and the system. Although the 32-bit address machines now available would accomodate the majority of databases in use today, there would still remain a few which were too big. The number of these very large databases is bound to grow, but at the same time the vast majority of new databases will be much smaller. Also, it is quite conceivable that 2-dimensional virtual memory systems will be introduced which have 32 bits to identify the segment and 32-bit addresses within each segment. These systems would undoubtedly

accomodate all the databases to be designed in the foreseeable future.

However, the virtual memory would have to contain not only the database, but also the DBMS routines, application program, tables, indexes etc. plus system routines and data.

6.3.2 Non-locality of access

As was stated earlier, it has been shown that programs do exhibit locality of access [46], but it seems unlikely that the same would be true of database usage. For example, by definition, transaction processing on a large database, shows no locality of access. Thus mapping the database directly onto virtual memory derives no advantage from the automatic memory management facilities in the virtual memory Operating System, which depend, in part, for their efficiency, on locality.

6.3.3 Privacy constraints

Most virtual memory (VM) Operating Systems (e.g. the Edinburgh Multi-Access System [47]) have more than one level of access to a process' virtual memory. For example, the system may access the entire VM, while the user process may access only part of it. The users of the DBMS do not have uniform rights of access to all the data in the database. Thus the database could not be mapped directly onto a single level of virtual memory. In fact, several levels would be required and with privacy controls operating at area, record and field level, this could be very complex. The DBMS would still have its own privacy controls (see Section 3.5) in addition to the automatic security provided by the VM OS through the various

levels of access. However, if a sensitive data field, record or area is mapped directly onto a process' VM, it is easier to bypass the DBMS and so gain illegal access.

6.3.4 Data integrity

Of all the reasons given above for not mapping the database directly onto the VM, perhaps the most important is the fourth, namely, the difficulty of ensuring data integrity. Consider, for example, a transaction which involved several changes to the database, which together formed a single logical unit. In order to guarantee the integrity of the database, either all the updates involved in the transaction are completed or none. In a VM OS this would be impossible. An update operation is complete and secure only after the page involved has been written back from VM to secondary storage. In a group transaction, altered pages will be written back to secondary storage at irregular time intervals, depending upon many factors, including page fault patterns, processor allocation etc. It would therefore not be possible to ensure that all the updated pages involved in the transaction are written back to secondary storage at the same time.

6.4 The subdivision of database for storage mapping

Since it is not advisable to map the entire database directly onto VM, it is necessary to subdivide the database into units for storage. In the same way, the Database Administrator (DBA) running on a non-virtual memory system must

divide the database into CS files. In fact, the difficulties are the same for both systems - namely, the conflicting requirements of the various applications for physical record placement and the DBA's desire for overall efficiency.

Having divided the database into several large physical sections, the VM system itself becomes significant. In a non-VM system, records in files are grouped together into I/O blocks, each block being the same size in order to reduce secondary storage accesses. This is no different from the VM system dividing the files into fixed size pages. Thus although, from the programmer's view, the entire file appears to be in main memory and all records equally rapidly accessible, in reality, as Stacey in [48] points out, the two-level storage environment still exists with the penalty of secondary storage accesses.

It is worth noting the usefulness of the concept of a Frame, as described by Senko in [49]. The frame provides a unit for the physical grouping of space allocation, record control fields etc., which may map onto one or more pages in VM.

6.5 Concluding remarks

The VM system gives the application programmer the illusion of a one-level storage system with all advantages. The DBMS designer and DBA, however, have to take into account the fact that the storage system only appears to consist of a single level, whereas in reality it is composed of at least two levels. Thus the penalties of data transfer between secondary and primary storage which exist in non-VM systems, must still be considered.

The problems of devising efficient methods for both the subdivision of the database into storage units (files) and for physical record placement, still exist whether or not the DBMS is running on a VM system.

In the final analysis, however, although the VM system may not solve any of these difficult problems for the DBMS designer and the DBA, it undoubtedly makes the solutions simpler. Thus while they must bear the multi-level storage environment in mind, the DBMS files can still be handled through the VM system.

PART II

THE DESIGN OF EDAMS

CHAPTER 7

THE OVERALL DESIGN OF EDAMS

7.1 Introduction

The second part of this thesis is concerned with a description of a database management system called EDAMS (EMAS Database Management System) designed to run on the Edinburgh Multi-Access System, EMAS (see Chapter 10).

EDAMS is based on the CODASYL proposals [1, 19, 32] although there are several fundamental differences. Rather than describe EDAMS in detail, the main differences between EDAMS and CODASYL will be explained and discussed in this part of the thesis.

7.2 The role of the EDAMS schema and subschema

The role of the CODASYL schema and subschema was discussed in Section 5.3.2 and also the changes proposed by DBAWG [34] in Section 5.2.4 following the introduction of the storage schema. EDAMS takes a different view of the relationship between the parent schema and its subschemas.

The EDAMS schema is a description of all the entire database. For simplicity, in the initial version of EDAMS, the schema is seen as a description of the complete physical database, i.e. of the records themselves and the fields they contain. This is not an essential restriction, however, since a DBAWG-

type storage-schema could easily be placed underneath, thereby providing three levels of data description - storage-schema, schema and subschema. Thus the EDAMS schema is a description, in terms of records containing fields, of the pool of data available to the user community. With this view of the schema it becomes irrelevant whether or not one schema record is physically stored as a single storage record.

In order for the user to be able to uniquely identify each schema record, it is necessary for the DBA to define one field of each schema record type as a schema record key. The keys must be distinct for all records of the same type. It is most unlikely that the DBA will have to add an extra field to a record for the key, since good database design generally insists that records be distinguishable from within (i.e. apart from system-assigned database keys).

The EDAMS subschemas are descriptions of the logical portions of the database required by various applications. All operations on the data in the database are carried out via a subschema. No direct access through the schema is possible. It is important to realize that the subschema does not simply provide the user with a window into the database. If this were not so, then the storage and deletion of data in the database would only involve making the window bigger or smaller, i.e. adding or removing data from the user's view. It would not involve physical changes being made to the database itself.

It is more accurate to think of the EDMAS subschema as providing the user with a door into the database through which the user can see, but can also gain access, if he has the right key.

There are two major consequences of these altered roles for the EDAMS schema and subschema:

(a) introduction of the subschema logical record

(b) alteration of the role of the set in the schema

7.3 The EDAMS subschema logical record

It was pointed out in Section 4.3.2 that the rules governing the derivation of subschema records from parent schema records are very restrictive under the CODASYL proposals. The order of fields in the subschema record may be altered, certain fields may be omitted completely and the attributes of data items may be changed. EDAMS removes these restrictions entirely by introducing the subschema logical record.

A subschema logical record is a record whose fields may be drawn from a number of different parent schema records as shown in Figure 7.1.

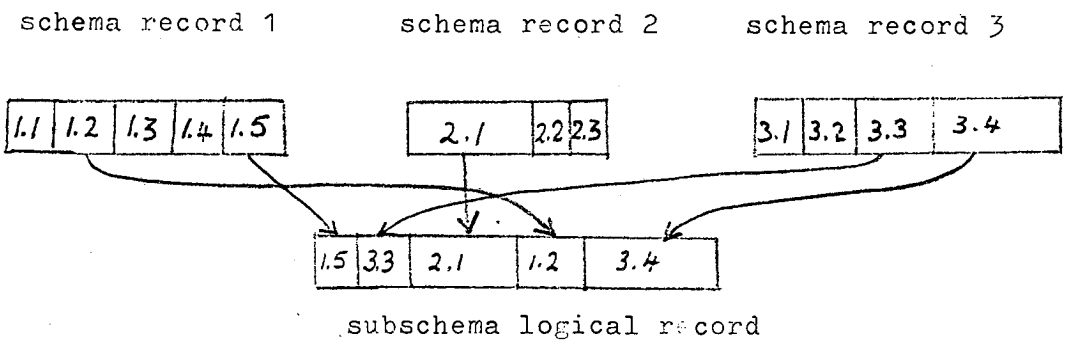


Figure 7.1 Derivation of subschema logical record

The method of formation of subschema logical records is discussed in detail in Chapter 8.

All records in the EDAMS subschema are regarded as

logical, in the sense used above. A CODASYL-type subschema record can be defined by simply selecting the fields of the logical record from a single schema record as shown in Figure 7.2.

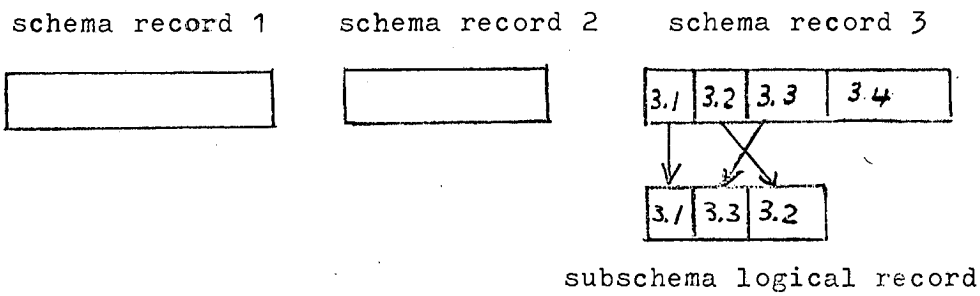


Figure 7.2 Subschema logical record derived from single schema record

7.4 Sets in EDAMS

In the CODASYL proposals, sets may only be defined in the schema; a subschema may use the sets of its parent schema, but may not create new ones. The introduction of the subschema logical record in EDAMS requires that this structure is altered. EDAMS therefore allows the user to create new sets in the subschema to link together the logical records.

It was stated earlier that all EDAMS subschema records are regarded as being logical and are treated in the same way. The question therefore arises as to whether or not sets should be removed entirely from the schema. It is clearly not essential to confine the definition of sets to the subschema. But in order for the schema set to have meaning in the subschema, it would be necessary for the owner and member records in the

subschema to be subsets of the owner and member records in the schema. In other words, the subschema logical records would have their fields drawn from one and only one schema record, as illustrated in Figure 7.2.

EDAMS treats each subschema separately. No sharing of logical records or sets between subschemas is possible. There are two main reasons for this. Firstly, the first subschema to define the new logical record or set would in a sense be dictating its structure to a second subschema which wished to use that record. Such an arrangement would be satisfactory if the second subschema were a subset of the first, but this would not generally be the case. The second problem associated with subschema sharing is the difficulty of deciding what rules should apply to such sharing. Should the logical records be identical in all respects, including field order and attribute? Or should the rules which operate between the CODASYL schema records and its derived subschema records apply? For example, suppose subschema SS1 defined a record composed of fields F1, F2, F3 and F4, which subsequently subschema SS2 discovered would also be useful, but omitting field F2. If the CODASYL-type rules applied then subschema SS2 could use the record. If, on the other hand, it was subschema SS2 which had first defined a logical record composed of fields F1, F3 and F4, then with CODASYL-type rules subschema SS1 could not use the record. Of course, it would be possible using the EDAMS rules for the formation of logical records for subschema SS1 to form a "logical" logical record, so to speak, by adding field F2. A hierarchy of subschemas could be envisaged but it is easy to see how confusing the situation could become.

Given the structure of EDAMS, if several subschemas wish to use the same logical record (or even just a group of fields e.g. F1, F3 and F4 in the above example) then it would surely be more efficient to reorganize the schema so that the fields in the logical record, or part thereof, are grouped together to form a new schema record. The advantages of an underlying storage-schema, whose records are based on subschema records, become clear in such situations.

The same problems also arise with the sharing of sets between subschemas which make it impractical in EDAMS. Returning therefore to the question of whether or not sets should be allowed in the EDAMS schema, it becomes apparent that unless they are permitted, no sharing of relationships between data will be possible in EDAMS - other than the mere juxtaposition of fields in a record (physical or logical). This is clearly unsatisfactory and could lead to an unacceptable level of duplication between subschemas. Moreover, although the set is a logical concept and therefore does not perhaps belong in the schema, the fact remains that certain relationships between records are inherently part of the structure of the database. For example, all the various records pertaining to one employee, or one project do belong together, no matter what the application. In other words, the set itself carries information which has to be stored somewhere. Thus EDAMS retains the set at schema level while, at the same time, allowing the user to define new sets in the subschema.

7.4.1 Use of schema sets in subschemas

It is necessary now to examine the rules which govern

the use of a schema set in a subschema. CODASYL rules restrict the subschema records to be subsets of their parent schema records. Thus sets which link together schema records will be meaningful in the subschema. The situation in EDAMS is complicated by the SLR. Consider the diagram below in Figure 7.3 of schema set S.

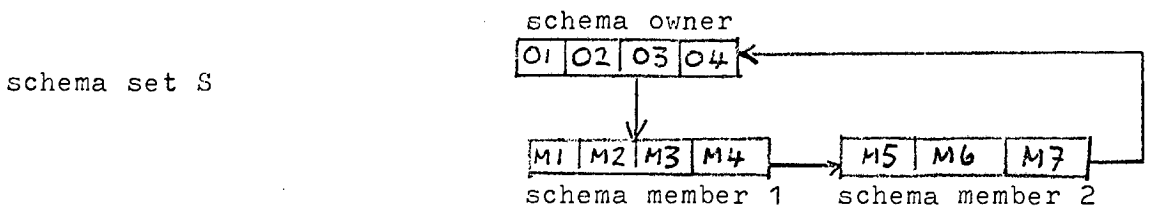
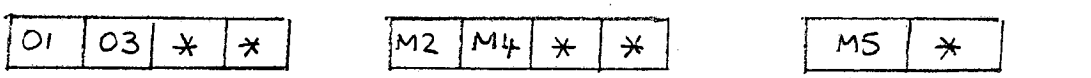


Figure 7.3 Schema set S

Suppose three SLRs which contain fields from the owner and member records of schema set S, as shown in Figure 7.4



subschema record 1 subschema record 2 subschema record 3

where * indicates fields from records which are not part of schema set S

Figure 7.4 SLRs derived from records in schema set S

In Figure 7.4, two fields in subschema record 1 are taken from the schema owner, two fields in subschema record 2 are taken from schema member 1 and one field in subschema record 3 from schema member 2. Thus the schema set S could still be meaningful in the subschema, as shown in Figure 7.5.

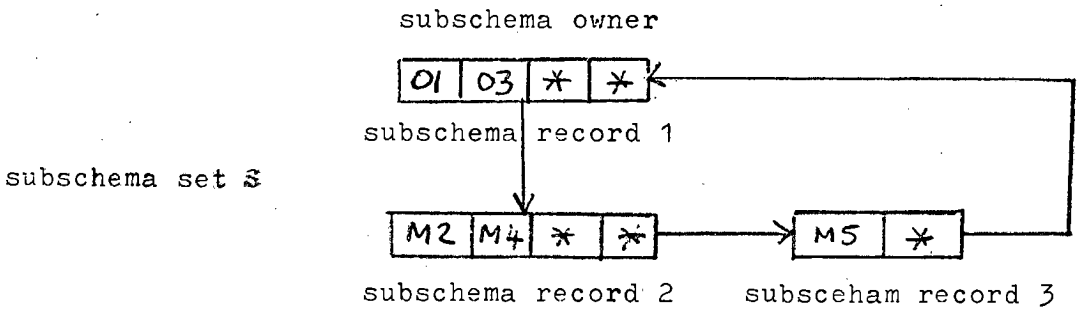


Figure 7.5 Subschema set S

If, however, the SLRs contained a mixture of fields from the schema owner and schema member records, the use of the schema set S in the subschema is confusing, as shown in Figure 7.6.

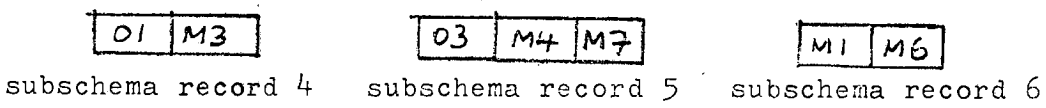


Figure 7.6 Alternative SLRs derived from records in schema set S

Subschema records 4 and 5 each contain fields from both the schema owner and member record of set S. It is therefore not obvious which should be the owner and which the member, if the subschema set S were to be established.

The purpose of retaining the schema set in EDAMS was to enable sharing of sets as well as data across subschemas. Thus, it would not be illogical to restrict the subschema records defined as forming part of the schema set, to be subsets of their parent schema records, i.e. single-source SLRs. As Figure 7.5 shows, this restriction is more severe than is absolutely necessary. It would be possible, for example, to insist that the subschema owner record contained at least one field from the parent schema owner record and that each subschema member

record contained at least one field from its parent schema member record. Other fields in the subschema records can be drawn from anywhere in the database. However, the more severe restriction, of single-source SLRs, is more straightforward. Above all, however, this restriction also applies to other uses of SLRs, as is shown in Chapter 9. It is clearly much simpler to have one restriction applied in all necessary situations, rather than one restriction in one group of situations, another restriction elsewhere and so on. The restriction is, in fact, the same as that which applies between CODASYL schema and subschema records.

7.5 Areas in EDAMS

In Section 5.3.1 the difficulties associated with CODASYL areas were discussed. The area performs directly or indirectly all the following functions:

(a) provides the basic access and locking mechanism
(b) divides the database into both logical and physical sections
(c) provides the mapping between the database and the Operating System files.

The area is basically a physical concept, yet CODASYL requires that the user is aware, in certain circumstances, in which of a number of areas, the record he requires is located. The user should not be required to possess such information.

The replacement of the area by the realm and storage-area in the current CODASYL position, has helped to remove some of

the anomalies. The realm is moving further away from physical storage and is being seen more as a logical subdivision of the database. In this way, the idea of overlapping realms, as mentioned in Section 4.4.1, becomes feasible.

In EDAMS the role of the area/realm is more complex because of the introduction of SLRs. However, the distinction between realm and storage-area becomes sharper. The storage-area is a physical entity and is defined as a subdivision of physical storage. Thus the storage-area belongs in the EDAMS schema, at least initially, when there is no separate EDAMS storage-schema. The realm, on the other hand, as a logical concept, belongs in the EDAMS subschema. EDAMS SLRs may be assigned to one or more realms. This assignment is defined when the logical record is defined in the subschema DDL. The rule for the assignment can be based upon a number of criteria:

(a) logical record type - all logical records of that type are assigned to one realm

(b) set membership - all members (and owners) of a set are placed in a given realm

(c) field values - realm assignement is based upon the value of a particular field in the logical record.

Hence realms may overlap. The EDAMS realm can therefore be thought of as a shorthand for referring to a group of (logical) records, other than by content. It can be used as a logical device, in addition to the algorithm which is discussed in detail in Chapter 10 and as a unit for privacy control.

CHAPTER 8

THE FORMATION OF SUBSCHEMA LOGICAL RECORDS

8.1 Introduction

The EDAMS subschema logical record is composed of fields drawn from one or more parent schema records. Such a tool is potentially very powerful but if the database were badly designed, severe inefficiencies could result. One logical access to the database could require several physical accesses to collect the fields together comprising the logical record. In a two-level architecture, i.e. schema and subschema in the initial version of EDAMS, the aim would be to ensure that the fields of frequently used logical records are drawn from a single parent schema record. The advantage of a three-level architecture comes from relating subschema and storage-schema records, i.e. not schema and storage-schema or schema and subschema. The reason for this is that the database is always accessed through the subschema.

One of the main objectives of this thesis is to provide the user with much greater flexibility at subschema level. The EDAMS subschema logical record (SLR) plays a major role in the provision of this increased flexibility. Before discussing the use (retrieval and update) of SLRs by the application programmers and high level users, it is essential to examine the method of formation of SLRs as defined in the subschema DDL.

8.2 The use of the relational approach

Consider the portion of a sample database given in Figure 8.1 below.

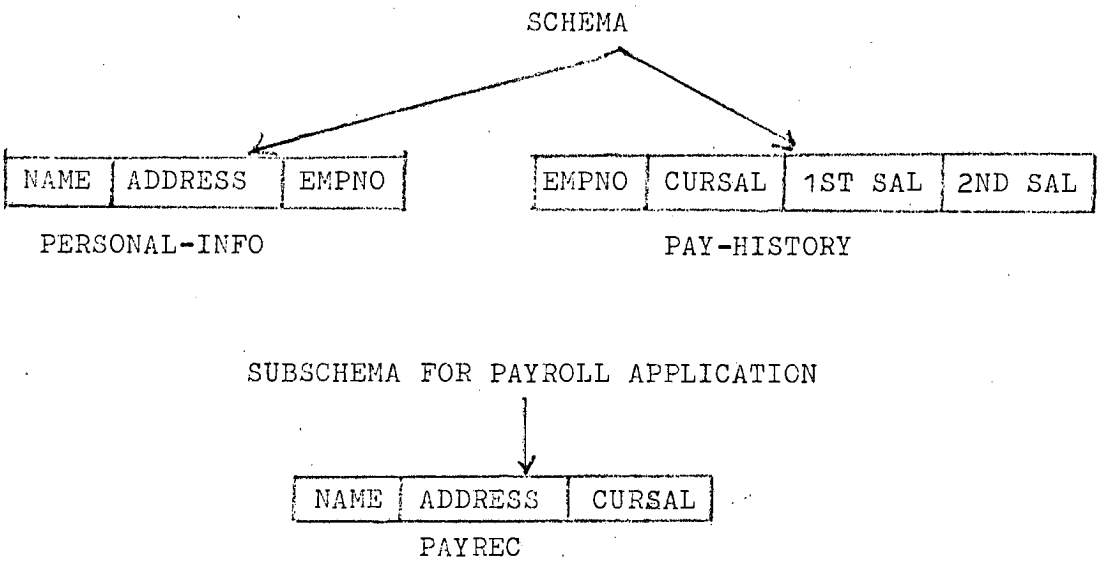


Figure 8.1 Portion of a sample database

A structural definition of the SLR PAYREC would be given in the PAYROLL subschema DDL as follows:

DEFINE RECORD TYPE PAYREC;

FIELD 1 IS NAME; SOURCE IS NAME FIELD OF RECORD TYPE PERSONAL-INFO

FIELD 2 IS ADDRESS; SOURCE IS ADDRESS FIELD OF RECORD TYPE PERSONAL-INFO

FIELD 3 IS CURSAL; SOURCE IS CURSAL FIELD OF RECORD TYPE PAY-HISTORY

Figure 8.2 Definition of SLR structure

This definition defines the source record types, namely

PERSONAL-INFO and PAY-HISTORY, for the formation of the PAYREC SLR. It does not, however, specify the rules for associating a particular occurrence of a PERSONAL-INFO record with a particular occurrence of a PAY-HISTORY record to generate the corresponding occurrence of the PAYREC SLR.

EDAMS uses two methods, which can be used separately or together, to solve the problem of source record identification. Both methods incorporate some useful features of the relational model [10]. The first method is based on records and the second on sets.

The relational data model has already been discussed in Section 3.2.3, but the operations which can be performed on relations were not discussed in that section. Two operations, JOIN and PROJECTION, are of interest. The JOIN operation is simply a means of combining two relations on a common domain (field) and PROJECTION is a means of selecting desired domains from a relation. Consider the example given in Figure 8.3

supp(supplier	part)
1	1
2	2
2	3

part(part	project)
1	1
2	4
3	2

Figure 8.3 Two joinable relations

The join of supp and part would be the relation R in Figure 8.4

R(supplier	part	project)
1	1	1
2	2	4
2	3	2

Figure 8.4 The join of relations supp and part

Consider the relation supply(supplier, part, project, quantity) as shown in Figure 8.5.

supply(supplier	part	project	quantity)
1	2	5	17
1	3	5	23
2	3	7	9
2	7	5	4
4	1	1	12

Figure 8.5 The supply relation

The projection of the supply relation over the domains (supplier, project) would be the relation S shown in Figure 8.6.

S(supplier	project)
1	5
2	7
2	5
4	1

Figure 8.6 The projection of supply over (supplier, project)

8.2.1 Record-based formation of subschema logical records

The record-based formation of SLRs regards the schema record types as relations and forms a series of joins (and

projections, if necessary) on them in order to form a new relation, the required set of SLR occurrences.

Consider the sample database given in Figure 8.1 above. To form the PAYREC SLR, the following statement is all that is required:

JOIN PERSONAL-INFO, PAY-HISTORY ON EMPNO TO FORM PAYREC;

This would result in a set of records containing the NAME and ADDRESS fields from the PERSONAL-INFO record, EMPNO from both (common field) and CURSAL, 1ST SAL and 2ND SAL from PAY-HISTORY. It is therefore necessary to select the required fields. This can be done either by means of individual field listings as in Figure 8.2 placed before the JOIN command or, alternatively, by making use of the relational operation of PROJECTION:

JOIN PERSONAL-INFO, PAY-HISTORY ON EMPNO TO FORM TEMPREC;
PROJECT TEMPREC OVER NAME, ADDRESS, CURSAL TO FORM PAYREC;

Although projection is undoubtedly a much shorter way to describe the selection operation, the more verbose DDL of Figure 8.2 might be useful when information other than the field's inclusion in the SLR is required, e.g. privacy information, field characteristics where they are different from the schema and so on.

In the above example, the EMPNO field will be a unique identifier for both sets of records. Thus two different employees could not have the same EMPNO. For every value of EMPNO, there will be only one matching pair of PERSONAL-INFO and PAY-HISTORY records. Such a join is known as an equijoin. If, however, the "joining" field is non-unique, then the EDAMS rule is to generate all possible pairs.

Consider the following example given in Figure 8.7 below.

	Record Type A			Record Type B	
Occurrences	Fields				
	F1	F2	F3	F4	F5
1	7	3	2	7	9
2	8	4	9	7	6
3	8	1	0	7	3
4	9	2	1	9	5
5	9	7	6	9	4

Figure 8.7 Two "joinable" relations A & B where cardinality increases

The result of a join operation on the above relations, Record type A and Record type B:

JOIN RECORD TYPE A, RECORD TYPE B ON F1 TO FORM RESULT;

is given in Figure 8.8 below.

	Result			
Occurrences	Fields			
	F1	F2	F3	F5
1	7	3	2	9
2	7	3	2	6
3	7	3	2	3
4	9	2	1	5
5	9	2	1	4
6	9	7	6	5
7	9	7	6	4

Figure 8.9 Join of relations A & B

Thus every possible combination of records is produced based on the common field over which the join takes place.

8.2.2 Set based formation of subschema logical records

The second approach to the formation of SLRs is based upon the set membership structure of the parent schema records from which the SLRs are derived. Once again, the relational model is used except that in this case the "handle" for the join operation is a set type rather than a field type.

Consider the employee database given in Figure 8.1 and suppose there is a schema set called EMPLOYEE of which PERSONAL-INFO is the owner record and PAY-HISTORY a member, as shown in Figure 8.10.

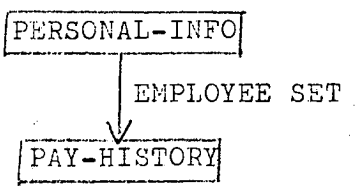


Figure 8.10 Employee schema set structure

The subschema DDL for defining the PAYREC SLR could then be:

```
DEFINE RECORD TYPE PAYREC
  FIELD 1 IS NAME;  SOURCE IS NAME FIELD OF RECORD TYPE
    PERSONAL-INFO OWNER OF EMPLOYEE SET;
  FIELD 2 IS ADDRESS;  SOURCE IS ADDRESS FIELD OF RECORD
    TYPE PERSONAL-INFO;
  FIELD 3 IS CURSAL;  SOURCE IS CURSAL FIELD OF RECORD TYPE
    PAY-HISTORY MEMBER OF EMPLOYEE SET;
JOIN PERSONAL-INFO, PAY-HISTORY THRU SET EMPLOYEE TO FORM
    TEMPREC;
PROJECT TEMPREC OVER NAME, ADDRESS, CURSAL TO FORM PAYREC;
```

Figure 8.11 DDL for set-based formation of PAYREC SLR

The fact that PERSONAL-INFO and PAY-HISTORY have a common field

is irrelevant. The join operation will cause the two schema records PERSONAL-INFO and PAY-HISTORY, to be merged to form the SLR PAYREC, according to the schema records occurrences in the EMPLOYEE set.

8.2.3 Selection expressions

Not only is it possible to use a combination of the two approaches described above, but also to introduce selection expressions to produce subsets of the join. For example,

JOIN PERSONAL-INFO, PAY-HISTORY ON EMPNO WHERE CURSAL < 10000 TO FORM TEMPREC;

Only those PAY-HISTORY records for which the CURSAL is less than 10000 will be included in the join.

The implementation of the EDAMS SLR is therefore making extensive use of the relational approach to DBMSs. This has the advantage of retaining the flexibility and data independence of the relational model without detracting from the CODASYL user model. It is worth noting that the relational sublanguage described above is highly relevant to data retrieval and query languages.

8.3 Derived fields

It is debatable whether derived fields (SOURCE and RESULT) should be permitted in the EDAMS schema at all. To allow the existence of VIRTUAL field would be confusing considering that all EDAMS subschema fields are VIRTUAL in one sense. The CODASYL ACTUAL SOURCE and RESULT fields are physically stored

in the database and hence their inclusion in the EDAMS schema is reasonable. CODASYL insists that a SOURCE field must be derived from a field in its owner record but such a restriction would be insufficient in EDAMS, since not all schema records belong to sets. Record type alone is not enough to uniquely identify the source record. An EDAMS ACTUAL SOURCE field can be derived from a field in any schema record type. To uniquely identify the source record, EDAMS uses the schema record key (see Section 7.2). The paramters for an EDAMS ACTUAL RESULT field can be taken from anywhere in the physical database, as in the CODASYL proposals.

8.3.1 Time of calculation of ACTUAL derived fields

The time of calculation of derived data items is important since it affects not only the efficiency of retrieval and update, but also the integrity of the database. It is not necessary, nor is it possible, to insist that the values of two duplicated fields be identical at all times, but rather only when they are expected to be identical, i.e. whenever an application program needs them. However, in order to ensure the integrity of the database, it is necessary to insist that if the two fields differ at any moment, e.g. after system failure, there must be some rigorous means of telling which of the two versions is correct. CODASYL allows the derived data item to be updated, hence altering the source as well. This therefore implies that the two fields have equal status. Thus the only way to ascertain which version is correct in the event of a disagreement is to use the journal tapes, which is in keeping with the resolution of other update anomalies which might occur following

a system failure.

There are two possible approaches to when the value of a derived field (source or result) should be calculated, namely:

(a) only when the derived field is actually accessed

(b) every time the source is altered for a source field and every time any parameter is altered for a result field.

The first of these two alternatives (a) has the advantage that re-calculation of the derived data item takes place only when absolutely necessary. However, which of the two approaches operates more efficiently overall depends on whether the derived item is read more often or written more often. If the item is written more often, then the first approach would be better. However, this raises the question as to why the duplicated field was necessary, if it is not used very often. The main reason for the repetition of fields is when it is worthwhile because of high access frequency (see Section 3.4.4) in order to improve the efficiency of retrieval. If the derived field is recalculated only when the field is accessed, then this will add an overhead to the retrieval operation. At the very least, a check will have to be made as to whether or not the source field has changed since the derived field was last updated. If the source field has altered, then it is necessary to change the derived field. There is also the overhead of storing flags of some sort to indicate changes to the source. On the other hand, no such operations are required if the second approach (b) is adopted, namely the recalculation of the derived field takes place at the same time as the source field or result parameters are updated; this would be treated as a group update. Thus the derived field

always contains the up-to-date version of the item.

These considerations therefore favour the second approach, which is adopted by EDAMS (and also by CODASYL), i.e. derived fields are calculated every time the source is altered for a SOURCE field and every time any parameter is altered for a RESULT field. Furthermore, EDAMS also allows the derived field to be updated with the automatic updating of the source field taking place at the same time.

8.3.2 Time of calculation of VIRTUAL derived fields

In EDAMS all subschema fields in the logical records correspond to the CODASYL VIRTUAL SOURCE or to the CODASYL VIRTUAL RESULT. Since the values of VIRTUAL fields are not physically stored within the record, calculation of their values can only take place when a GET command involving those fields is executed. Given the relationship between the EDAMS subschema and its schema, it is not meaningful to restrict the derivation of VIRTUAL SOURCE fields to fields from the owner record of the set as in the CODASYL proposals. Note that the majority of subschema fields (in logical records) will be VIRTUAL SOURCE. The EDAMS VIRTUAL RESULT field will operate in the same way as its CODASYL counterpart - namely, parameters may be drawn from any record(s) in the database (i.e. defined in the EDAMS schema) and calculation takes place only when the field is actually retrieved. An EDAMS VIRTUAL RESULT field cannot be the subject of a STORE or MODIFY command.

8.4 Rules for encoding and decoding

For every field in a CODASYL database, encoding/decoding procedures can be specified in the schema. The procedures can be invoked every time the field is stored (encoding) or retrieved (decoding) or only when the attributes of the data item differ from schema to subschema (USAGE clause). The encode/decode facility can be used for:

(a) encryption/decryption

(b) data compression/expansion

(c) unit changes, attribute variation etc.

In terms of the three-level data description structure, both encryption/decryption and data compression/expansion could operate between schema and storage-schema as well as between subschema and schema. EDAMS allows their specification in both the schema and subschema DDL. However, there would seem little point in using the facility at both levels for the same field.

Where the third use of the encode/decode facility is concerned, namely for unit changes and attribute variation, it would not be logical to allow this to be specified in the EDAMS schema. EDAMS therefore restricts the use of this facility to the subschema DDL.

8.5 Privacy information

Another ramification of the introduction of the concept of logical records in EDAMS is the specification of privacy

information. In the original April 1971 DBTG Report [1], privacy locks at subschema level overrode those at schema level. A rule such as this is necessary to avoid confusion. This approach, though not entirely logical, has the merit of being quite straightforward. The EDAMS schema represents the DBA's view of the database and it is his responsibility to apply locks to sensitive data items and records and to supply approved users with the appropriate keys. The question therefore arises as to whether those locks are applied at subschema or schema level (see Section 3.1, Figure 3.1 for a definition of the hierarchy of DBMS users). Consider the portion of a sample database given in Figure 8.1. Suppose that the current salary, CURSAL, field is sensitive and access restrictions are placed on it. If locks were set on CURSAL in both the schema and subschema, the payroll application would then have to give two keys each time the CURSAL field was accessed. This is confusing since the user sees the database through the subschema only and is not really concerned with the schema at all. The situation in EDAMS is further complicated by the fact that subschema records can be composed of parts of several different schema records. Thus multiple keys might be required to satisfy the schema locks as well as a subschema lock on the logical record and its fields.

In EDAMS privacy controls exist at two levels:

(a) between subschema and schema

This is to permit the inclusion of sensitive fields and records in the subschema logical record and could take the form of restriction of access to the Data Directory. Alternatively, locks could be set in the schema DDL for

which keys, to be checked by the subschema DDL compiler, would be given in the subschema DDL.

(b) between application program (and high level users) and subschema

These locks or privacy procedures are set in the subschema and are satisfied at execution time - they therefore operate in the same way as the CODASYL subschema locks. Apart from locks on individual fields and records in the subschema, it would also be necessary (as in CODASYL) to have a lock on the subschema itself. Thus only authorized users could gain access to the subschema.

CHAPTER 9

OPERATIONS ON SUBSCHEMA LOGICAL RECORDS

9.1 Introduction

There are four basic operations which can be performed on data in a database:

(a) retrieval

(b) update - in the sense of the alteration of the value of an existing field

(c) creation of a new record occurrence

(d) deletion of an existing record occurrence.

All access to the EDAMS database is via a subschema. Thus all the above operations must be carried out through subschema logical records. Before discussing the operations on SLRs, it is necessary to describe briefly how SLRs are implemented; the detailed description of their implementation is left to Part III of the thesis. The SLR as it is physically stored in an EDAMS database, consists of a series of pointers to (logical addresses of) the source schema record fields from which the SLR fields are derived. Figure 9.1 illustrates, diagrammatically, how the sample database given in Figure 8.1 would be physically implemented in EDAMS.

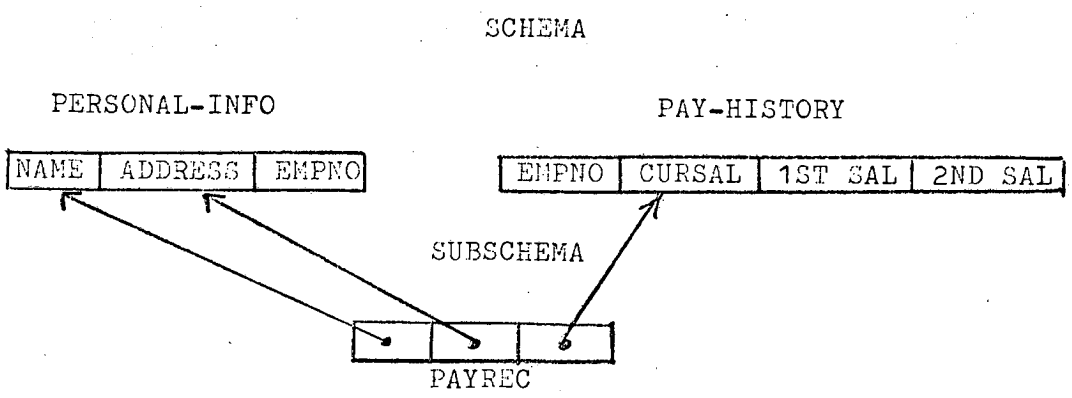


Figure 9.1 Implementation of PAYREC SLR

These pointers are established when the SLR is defined in the subschema DDL and become part of the permament database as an entity in the database map (see Section 13.2), until the SLR is deleted. As far as the high-level user of EDAMS is concerned, however, the PAYREC SLR looks like Figure 8.1 not Figure 9.1. In other words, the pointers are transparent to the users; they do not concern him and he does not have access to them.

9.2 Retrieval

The retrieval of an SLR is straightforward. As an example, consider the retrieval of an occurrence of the PAYREC SLR given in Figure 9.1, as shown in Figure 9.2.

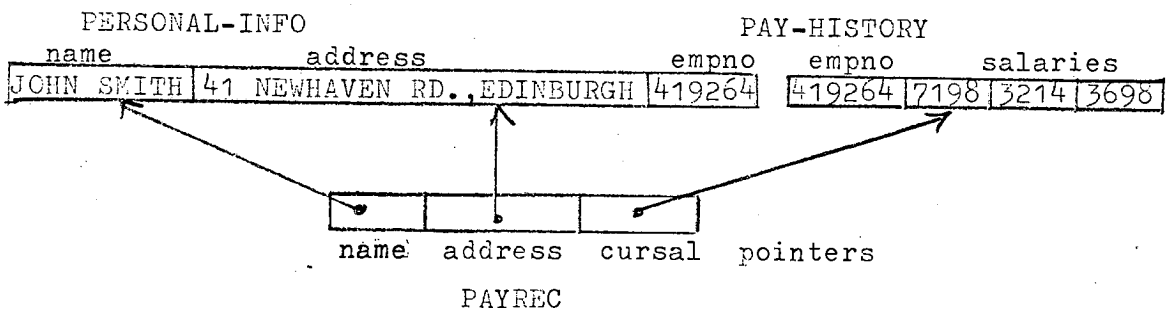


Figure 9.2 Diagrammatic representation of an occurrence of PAYREC SLR

For simplicity, assume that the user has fixed position in the database. A request to

GET NEXT PAYREC

will retrieve the one belonging to JOHN SMITH in Figure 9.2. To satisfy this request, EDAMS locates the particular PAYREC, extracts the pointers (logical addresses) and uses them to access the physical database in order to extract the required fields. The user is then presented with the record shown in Figure 9.3 below.

name	address	cursal
JOHN SMITH	41 NEWHAVEN RD.,EDINBURGH	7198

PAYREC

Figure 9.3 The retrieved occurrence of PAYREC SLR

9.3 Update

Update of an SLR, in the sense of the alteration of an existing field value, is also straightforward. For example, suppose JOHN SMITH in Figure 9.2 changed his address. The user would specify

MODIFY PAYREC; ADDRESS=48 MARCHMONT RD.,EDINBURGH

To obey this command, EDAMS follows the address pointer in the PAYREC SLR and alters the corresponding address field in the PERSONAL-INFO schema record. The resulting position of the database is given in Figure 9.4

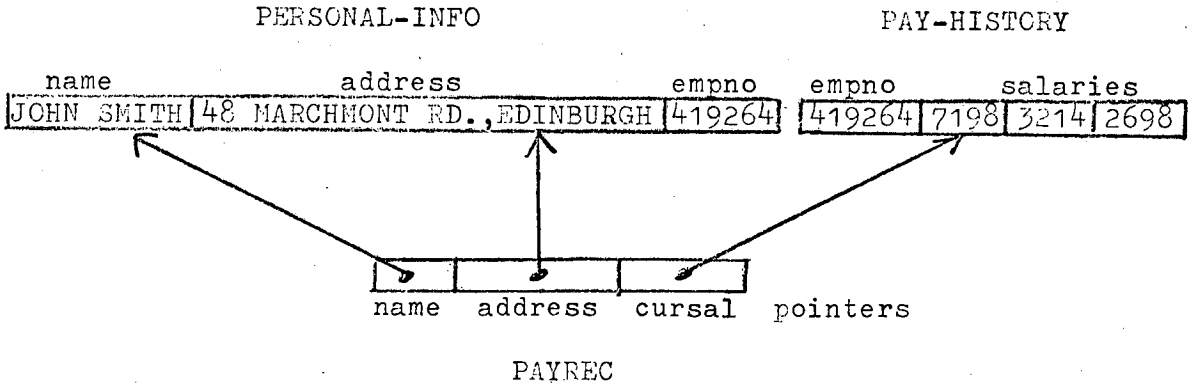


Figure 9.4 Updated PAYREC SLR

Note that the PAYREC SLR itself has remained unaltered.

9.3.1 Effects of the update

Even a simple update operation such as the one described above can have repercussions, which may require data in the database to be altered in addition to the single field which was the subject of the update operation. Consider the following portion of a company database.

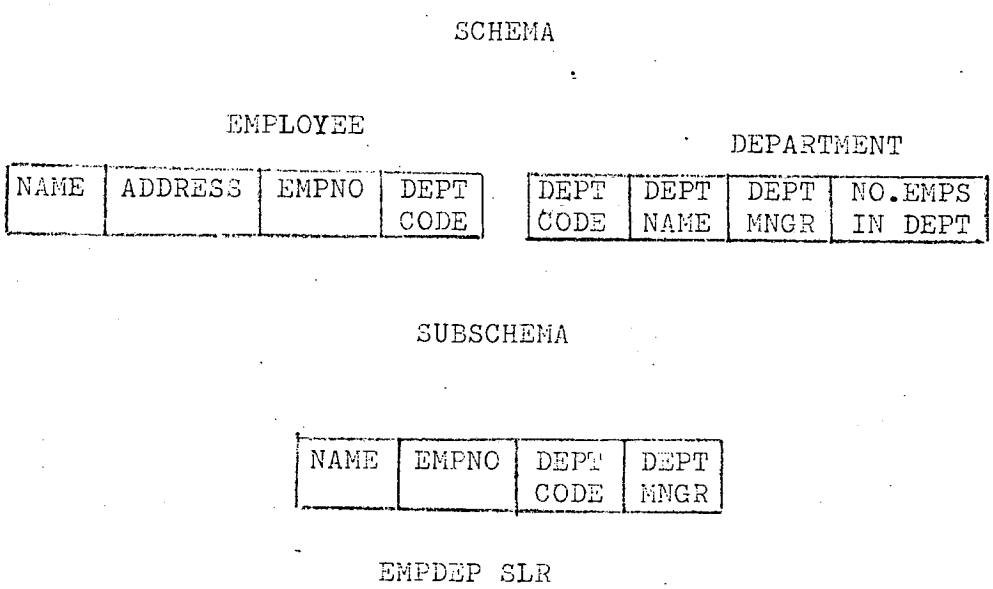


Figure 9.5 Portion of COMPANY database 1

Suppose that the SLR had been formed by use of the following DDL:

JOIN EMPLOYEE, DEPARTMENT ON DEPTCODE

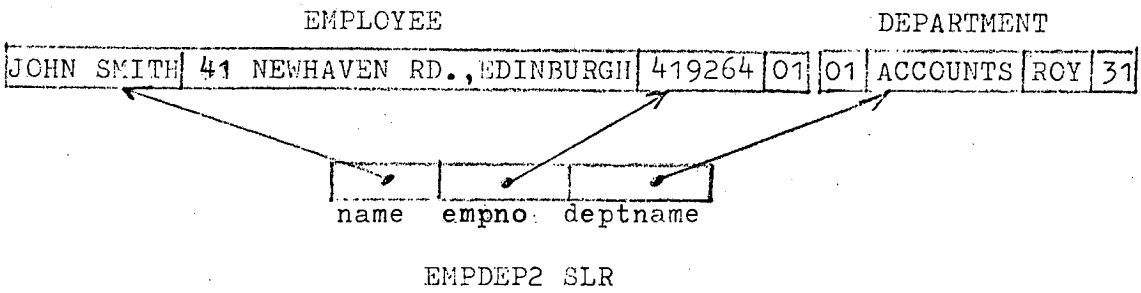
The EMPDEP SLRs are set up when this subschema DDL is executed. Suppose, however, that employee TOM BROWN is transferred from the ACCOUNTS department (code 01) to the PAYROLL department (code 02). Clearly, the simple alteration in the DEPTCODE field via the SLR and hence in the DEPARTMENT schema record would result in an invalid database.

Thus before executing an update, EDAMS must first examine the field to be updated to ascertain whether it is a key in the formation of that (or indeed of any other) SLR. In the above example, therefore, EDAMS must consult the DDL definition tables for the formation of EMPDEP SLR. Using this information, EDAMS scans the DEPARTMENT records for the one with DEPTCODE=02. It will then join this with the original EMPLOYEE record for TOM BROWN to form a totally new occurrence of the EMPDEP SLR. In this case, the formation rules for the EMPDEP SLR are relatively simple, but if they involved a nest of join operations, the whole process could become quite involved.

9.3.2 The update anomaly

Where the subject of the update is the key to the join operation, it is always possible for EDAMS to interpret the formation rules for the new SLR correctly. It is, however, possible to envisage a situation where the update of a field, which is not itself a key to the join operation, results nevertheless in an incorrect database.

Consider, for example, an amended EMPDEP SLR, called EMPDEP2 as shown in Figure 9.6.



EMPDEP2 SLR

Figure 9.6 Portion of COMPANY Database 2

Note that as before the EMPDEP2 SLR is formed using the following DDL

JOIN EMPLOYEE, DEPARTMENT ON DEPTCODE

If JOHN SMITH is moved from the Accounts to the Payroll Department, EDAMS treats the deptname field in EMPDEP2 as a normal, non-key field and updates it accordingly. The resulting database is shown in Figure 9.7

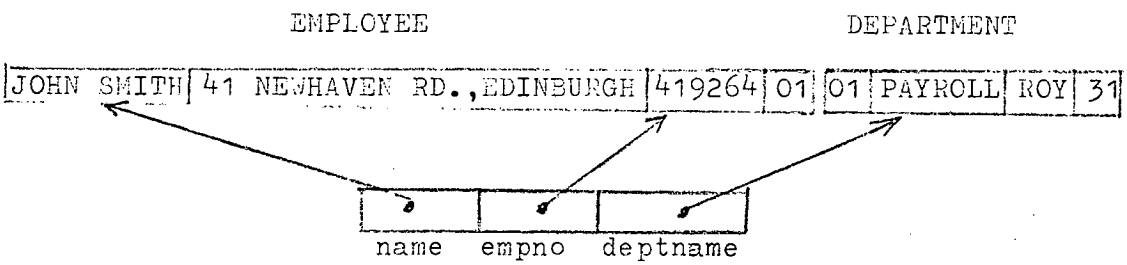


Figure 9.7 Incorrect COMPANY Database 2 arising from the update of EMPDEP2

On further examination, it becomes clear that, in general, the alteration of any field in an SLR which is formed as a result of a join operation on two or more schema records,

can result in an invalid database. The one exception to this is in fact the join key field itself, since EDAMS has sufficient information to select the new schema record to form the new SLR.

There are two possible solutions to this problem, namely:

(a) to rely on the DBA and users not to specify updates in a form which could result in an invalid database

(b) to disallow all update operations on SLRs other than those which are strict subsets of a single parent schema record.

The first solution of relying on the user and the DBA to police the system, is clearly totally impractical and can be dismissed. Therefore the second solution of restricting update to simple SLRs must be adopted. This is in some ways an unfortunate restriction, since it does remove a degree of flexibility at the subschema level. Moreover, many update operations can be carried out on SLRs without problems, such as the change of address in the example above. However, the restriction is clearly essential to safeguard the integrity of the database. Furthermore, it will also prove useful in the third database operation, that of the creation of a new record occurrence, which is discussed below.

9.4 Creation of a new record occurrence

In EDAMS a differentiation is made between the addition of a new logical record occurrence and the storage of one. The addition of a new logical record simply consists of establishing the pointers for the logical records to link in to the existing fields in the schema records, i.e. no new physical data is

added to the database; the user is simply adding more data to his own logical view. The storage of a new logical record, on the other hand, results in new physical data being added to the database.

To illustrate the distinction between these two operations, consider the following portion of a physical EDAMS database:

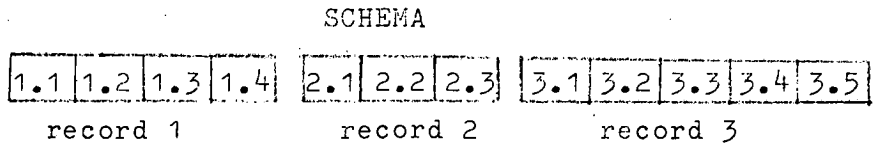


Figure 9.8 Portion of an EDAMS database

The addition operation is represented by the user who wishes to add a new logical record to his subschema which consists of fields 1.1, 1.3, 2.2, 3.2 and 3.5 as shown in Figure 9.9.

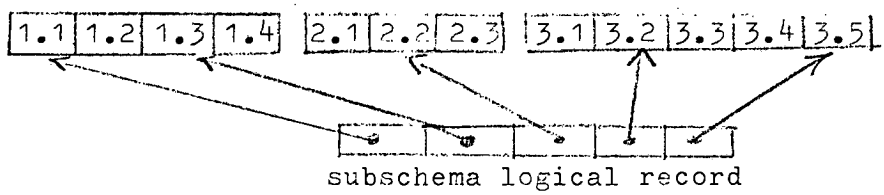
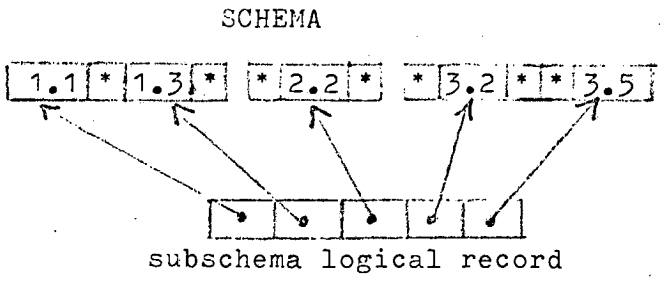


Figure 9.9 Addition of new subschema logical record

The second type of operation, storage, is quite different. Suppose that the new subschema logical record represents data on an entirely new entity. Hence schema records 1, 2 and 3 in Figure 9.8 would not exist. The result of the storage of the new SLR which consisted of five fields, formed by definition in the subschema DDL by a join operation on schema record 1, schema record 2 and schema record 3, followed by a projection to select the required fields, is shown in Figure 9.10.



where * indicates an unassigned field

Figure 9.10 Storage of new subschema logical record

The EDAMS storage operation corresponds to the CODASYL STORE. Fields in the schema record are unassigned if they do not appear in the corresponding CODASYL subschema record definition for which the STORE command was issued. Of course, the CODASYL STORE operation can only result in unassigned values being recorded in one schema record, whereas EDAMS can generate as many new schema records as there are source records for the SLR. It is as a result of this that a problem analagous to the update anomaly discussed in the previous section arises. The difficulty occurs when another SLR is stored which contains not only some of the unassigned fields in Figure 9.10, for example, but also some of those which have already been assigned as a result of the STORE on the first SLR. If these already assigned fields are updated with the new values, an incorrect database could result; however, not to update them but at the same time assign values to the previously unassigned fields could also result in an invalid database.

The simple solution to this problem is the same as to the update anomaly, namely to restrict the storage of new records to those which are a strict subset of a single parent schema record.

9.5 Deletion

Corresponding to the addition and storage operations, there are the removal and deletion operations, although the distinction is not so clear-cut.

The removal of a subschema logical record occurrence implies only its removal from the user's view. It cannot involve the deletion of any physical data from the database even if the fields involved are not referenced by any other subschema.

The deletion of a logical record, on the other hand, does involve the physical removal of the data from the database. The source fields for all the fields in the SLR are deleted from the source schema records, i.e. they are flagged as deleted. As with the update and storage of multi-source SLRs, difficulties can also arise with their deletion, when fields which are keys to join operations for other SLRs. As before, therefore, it is necessary to restrict the deletion of SLRs to those which are strict subsets of a single schema record.

9.6 Summary of operations on SLRs

The only operations which can be performed on multi-source SLRs are:

(a) retrieval

(b) creation

(c) removal

However, single-source SLRs can in addition to the above, be the subject of:

(d) update

(e) storage

(f) deletion.

CHAPTER 10

CONCURRENT UPDATE IN EDAMS

10.1 Introduction

In the light of all the difficulties associated with existing solutions to the concurrent update problem in DBMSs, a new algorithm is proposed for EDAMS.

The system is not dissimilar to the Chamberlin et al scheme [32] described in Section 4.3.5. There are three undesirable features of the Chamberlin et al scheme:

(a) by allowing blocked processes to hold locks for records, single record updaters could be discriminated against and caused to wait an unnecessarily long time

(b) the algorithm is tedious to implement with a proliferation of small queues, one for each locked record which has been requested by another process

(c) arbitrary method of favouring processes.

10.2 The EDAMS algorithm

Under this new method instead of a queue of processes for each record, there is a single queue of blocked processes awaiting the release of locked records by other processes. As soon as a process has all the records it has requested, it will be released, regardless of its position in the queue. All records involved in a group update must be claimed in a

single seize block, operations within the block being restricted in the same way as in the Chamberlin et al scheme. The position of processes in the queue is solely determined by their time of arrival (at a seize block). It is necessary to insist that once the search engine has been allocated to a process, that process will run until completion of the seize block or until it is blocked. Consider two concurrent processes P1 and P2:

Search engine allocated to P1

P1 reads and locks records R1 and R2

P1 reads R3, but decides not to lock it

Search engine allocated to P2

P2 reads and locks R3

End of search engine for P2

P2 updates record R2

P1 reads and locks R4 and R5

End of search engine for P1

While executing the update, P1 finds that R3 has been altered in such a way that it now satisfies its locking predicates. Thus in order to ensure that P1 does obtain a time-consistent snapshot of the database the search engine must be allocated to P1 until it satisfies all its locking predicates in a single attempt or until it is blocked because it wants to examine or claim a locked record.

Consider the following example:

PROCESSES ON QUEUE	SET OF LOCK REQUESTS
Pa	Wa = {R1,R2,R3,R4,R5}
Pb	Wb = {R2,R5,R6,R7,R8}
Pc	Wc = {R1,R5,R6,R9,R10}

where Wi = set of lock requests made by process Pi

Process Pa is at the head of the queue and assume that all records are initially unlocked. Hence process Pa can be released. The search engine will then examine all the locking predicates for processes Pb and Pc for the first time and will ascertain that both are blocked and place them on the queue.

PROCESSES	SET OF RECORDS LOCKED BY PROCESS Pi	SET OF RECORDS REQUESTED BY PROCESS Pi
* Pa	Ra = {R1,R2,R3,R4,R5}	Wa = ϕ
Pb	Rb = ϕ	Wb = {R2,R5,R6,R7,R8}
Pc	Rc = ϕ	Wc = {R1,R5,R6,R9,R10}

* indicates executing process

Note that the set Ri of records currently locked by process Pi is null for all processes within their seize blocks. Thus a process is not granted the lock for any record unless it can obtain all the records it requires in one go and be released. Assume for simplicity that Pa releases all its records simultaneously, although this is not an essential restriction as in Chamberlin's algorithm. Under Chamberlin's algorithm, processes are permitted to examine the non-updated versions of locked records and hence if records were released singly instead of in one go, a concurrent updater could obtain a snapshot of records some of which are updated versions (released records) and some of which are not (locked records) (see Section 4.3.5). After the simultaneous release of Pa's record , the search engine re-executes the locking predicates of Pb and discovers that all its requests can be met and it is released. The locking predicates of Pc are then examined, but this process is still blocked.

EXECUTING Pb Rb = {R2,R5,R6,R7,R8} Wb = ϕ

Pc Rc = ϕ Wc = {R1,R5,R6,R9,R10}

Pb will then release all its records, allowing Pc to run.

For clarity in the above example, static Wi sets have been used. However, in the general case, the locking predicates will depend upon database content and hence will vary with time. The algorithm is still valid in this situation.

10.3 Indefinite blocking of a process

It is possible under this new algorithm for a process to be indefinitely blocked, even though it is at the head of the queue. Consider the case where two processes, Pa and Pb, are concurrently updating the database (their lock sets must of course be distinct). A third process, Pc, is the first process on the queue. Pa releases its records so Pc's locking predicates are re-examined in the light of the newly-released records. Pc finds that it is still blocked as it requires some records currently held by Pb. The requirements of a fourth process, Pd, in position two in the queue, are then examined and the search engine finds that all its requests can be met, so it is released. Process Pb terminates, so once again the locking predicates of Pc are examined, but it is still blocked since it requires some of the records now held by Pd. The search engine will then move down the queue and release the next process, if possible. In theory, therefore, it is possible for a process such as Pc to be blocked indefinitely. Although this is unlikely to occur in practice,

the fact that the system cannot guarantee that all processes will be released eventually (short of being the only process in the system) is unacceptable. In order to avoid this, it is necessary to maintain the queue discipline throughout and not release a process until it is at the head of the queue. As in the case of Chamberlin et al scheme, this could lead to unacceptable and totally unneccesary delays for processes which only want a single known record.

A variation of this situation would result if processes are allowed to release records one by one instead of all together. Such an approach attempts to meet the requirement that no process should retain a resource for longer than is absolutely necessary. Consider a process Pa which is updating the database having locked records {R1,R2, ...,Rn}. Pa releases R1, but for consistency must retain {R2,R3,...,Rn} until the update is complete. Process Pb at the head of the queue requires {R1,R2} so it cannot be released, while process Pc further down the queue requires only R1 and can therefore be released. It is possible for Pb to remain blocked indefinitely. Consider, for example, the case where Pb's lock set is identical to Pa's, namely {R1,R2,...,Rn} and Pa releases each record separately.

It would be impractical to take the attitude that the user who carries out the type of operation which demands a lot of resources is anti-social and will just have to wait until those resources are available, i.e. effectively, giving him a very low priority. In the ultimate extreme, this could mean waiting until all other updating processes had logged off the system, which might never happen!

10.3.1 Favoured processes

The first approach to the problem of the indefinite blocking of a process, is the one taken by Chamberlin et al, namely of arbitrarily and externally favouring a process to guarantee that it will run. Apart from solving the problem of the indefinite blocking of a process, this approach has the advantage of giving the DBA some direct control over potentially extravagant users of the database. Moreover, the method of favouring can be used in certain urgent operations which require a time-consistent view of the database, e.g. calculation of the bedstate in a hospital, daily totalling of credits and debits in a financial system. Both these operations can be carried out very rapidly once the resources (records) are available. To introduce a system of favouring a particular process to the algorithm effectively means that until that process has built up its complete lock set, no other users can be allowed to lock records, i.e. enter seize blocks. This is necessary in order to guarantee that the process will be released given the potentially time-varying nature of locking predicates. Say, for example, that process Pa is favoured and Pb enters its seize block and requests a lock for record R10. Furthermore, assume that record R10 has been examined by process Pa but rejected as it did not meet any of its locking predicates. Now if the system were to grant process Pb the lock for record R10, it is quite possible that process Pa (still in its seize block) could decide as a result of some other newly released record, that it wishes to re-examine R10 only to find it locked by Pb. Thus in order to ensure that Pa, the favoured process, will run, it is essential to prevent any other process from entering a seize

block. Gradually, those processes which currently hold locks on records, will release them and the favoured process will be able to build up its entire lock set and be released. Since the permission to favour a process would only be given by the DBA, it is natural to assume that it would only be used in extreme cases where it is important. A favoured process is therefore a special process. In addition to the command to favour a process, the DBMS must also be supplied with a list of all the logical record types involved in the update. Thus other processes using logically disjoint portions of the database could be allowed to continue unaffected.

10.3.2 Waiting time priority system

An alternative to the external favouring of a process to solve the problem of the indefinite blocking of a process, is an internal priority system which requires no outside trigger to guarantee the release of a process. Under the waiting time priority system, each process is allocated a priority based on the time spent waiting. Thus the longer the process has been waiting (blocked) the higher will be its priority. A threshold value limits the difference between the process at the head of the queue (i.e. the one which arrived at the seize block first) and the process to be considered next for release. No special priority queue is required to implement this system as the EDAMS algorithm can simply use its standard process queue, since it is ordered purely by time of arrival at a seize block. Thus when a process joins the queue it is allocated a priority of zero, which is incremented the longer it has to wait. Eventually, then the priority difference of the process at the head

of the queue which has been blocked for a long time, will become so high that no other processes can execute their seize blocks and the process will then be released. The effect of this system is the same as the external favouring of a process (see above), except that it has the advantage of being automatic and less arbitrary. EDAMS therefore adopts this approach.

10.3.3 "Overlocking" for special purposes

It is the realization that favoured processes are special processes which leads to a third approach to the problem of the indefinite blocking of a process. It is not intended as an alternative to the priority system described above and adopted by EDAMS, but rather as a supplement to it. This approach consists of locking potentially more records than are actually required by the logic of the update, in one go, rather than evaluating a locking predicate one record at a time to build up the lock set. The obvious choice for the specification of this "overlocking" is the logical record type. Thus the locking predicate will simply be:

LOCK ALL RECORDS OF TYPE-X

Note that the evaluation of this locking predicate does not involve any examination of the database itself. The search engine merely has to ascertain whether any other process currently holds locks for any records of TYPE-X. If not, then the process can be released immediately. If so, then it is placed on the queue of blocked processes in the normal way.

Clearly, however, in order to guarantee that processes using this "overlocking" facility will be released quickly, there

would still have to be a waiting-time priority system as outlined above. The use of the "overlocking" facility would have to be regulated and only available to special processes.

For example, consider the calculation of the bedstate of a hospital. This involves the very brief examination of all the current in-patient records, which would presumably form a single logical record type. Thus no "overlocking" would be involved and the process would be released as quickly as possible with minimum overhead (no delay to evaluate locking predicates). The fact that all the records of a given logical type are required simultaneously for the bedstate calculation is typical of those processes which require a snapshot of a large portion of the database, e.g. daily totalling of credits and debits in a financial system. Thus the specification of the locking predicate by means of the logical record type will greatly increase the efficiency of these processes.

To summarize, therefore, the over-locking facility is not an alternative to the priority system, but rather an extension of it in order to increase the efficiency of certain special processes, where for example,

(a) the number of records to be considered for locking is large

or

(b) the number of unsuccessful attempts before the locking predicates are satisfied is large.

10.4 Repeated evaluation of locking predicates

Associated with the problem of the indefinite blocking of

a process is the problem of the repeated evaluation of locking predicates of such processes each time they are considered for release. This subject has already been discussed in detail in a general context in Section 4.2.2. In the context of the EDAMS algorithm, the problem will be alleviated either by the use of the waiting-time priority system or the "overlocking" facility.

PART III

THE IMPLEMENTATION OF EDAMS

CHAPTER 11

AN OVERVIEW OF EMAS

11.1 Introduction

In this chapter a brief description of the Edinburgh Multi-Access System (EMAS) will be given with particular emphasis on those aspects which affect a DBMS implemented on EMAS. EMAS is a general-purpose virtual memory time-sharing system for the ICL System 4-75 computer [47,50,51]. The paging unit provides 256 segments of 16 pages each, each page being 4096 bytes. Each user has his own virtual memory of up to 256 segments of 2↑16 bytes each. Segments 0-31 of each virtual memory are used by the Director processes (see Section 11.2) and are not available to the user. In EMAS, the distinction is made between the heart of the system provided by the system software and the part which is more visible to the user, the subsystem software. The aspect of the system software which is of direct interest to a DBMS is the Director.

11.2 Director

Each user process has a director process which can access the user's entire virtual memory. The main purpose of the director process is to perform file system and console communication services for the user. It is stored on a replaceable disc unit and is paged in and out to drum and core as required.

All director processes share the same code and access the same physical copy. Thus the director is almost always either in core or on drum all the time.

The File System provided by director contains all user files. Each file consists of an arbitrary, but integral, number of pages of totally unstructured information. All files are stored on-line on disc and are accessed by connecting them into the user's virtual memory, i.e. mapping the complete file onto a segment (or several contiguous segments) of the user's virtual memory. While a file is connected, the system pages it between disc and drum and core as required. Thus once the file has been connected the virtual address is used to reference it. Files can be shared between users and in this case all users access the same physical copy.

Files can be connected in one of four modes - read unshared, read shared, read and write shared, and read and write unshared.

The facility exists in EMAS for messages to be sent from one process to another which, as will be shown later, is of particular significance for a DBMS. The user service PON is used to place the message (limited to 32 bytes) onto the queue and the receiver removes the message by issuing a POFF request. If no message is on the queue, the receiver is suspended until the message is available. The receiver can then send a reply, if appropriate, using PON and the sender receives the reply using POFF.

11.3 The standard EMAS subsystem

The standard EMAS subsystem provides users with a variety of facilities including virtual memory management (except for the first 32 segments containing director), file organization conventions by means of file headers, command interpretation and so on.

Although all these functions are vital to the DBMS, many, e.g. command interpretation, can be taken for granted. The File Directory Package (FDP) which is responsible for virtual memory management is however significant. All file requests to director are via the FDP which maintains a map of virtual mamory and information concerning the size and mode of access of all files currently connected in the user's virtual memory.

11.4 Updating EMAS files

An important feature of EMAS from the point of view of a DBMS is how EMAS updates files; in particular, at what point are the altered pages in core transferred to disc? It is only when this transfer is complete that the update can be regarded as successfully executed. It is important to ensure, for example, that all the altered pages associated with a group update are written back to disc "simultaneously".

There are three situations when pages in core which have been updated by a process are written onto disc:

(a) when the file is disconnected (this would also include the user logging off, when all files currently connected are

automatically disconnected

(b) when the system wishes to reduce or change the processes' working set

(c) when the user service Make Disc Consistent (MDC) is requested

In all three situations, EMAS guarantees to ensure consistency by writing all pages altered by a process back to disc at the same time.

CHAPTER 12

THE EDAMS MASTER PROCESS

12.1 Introduction

The obvious starting point for the design of a CODASYL-type DBMS for EMAS was the conceptual DBMS given in the April 71 Report [1] which is reproduced in Figure 12.1.

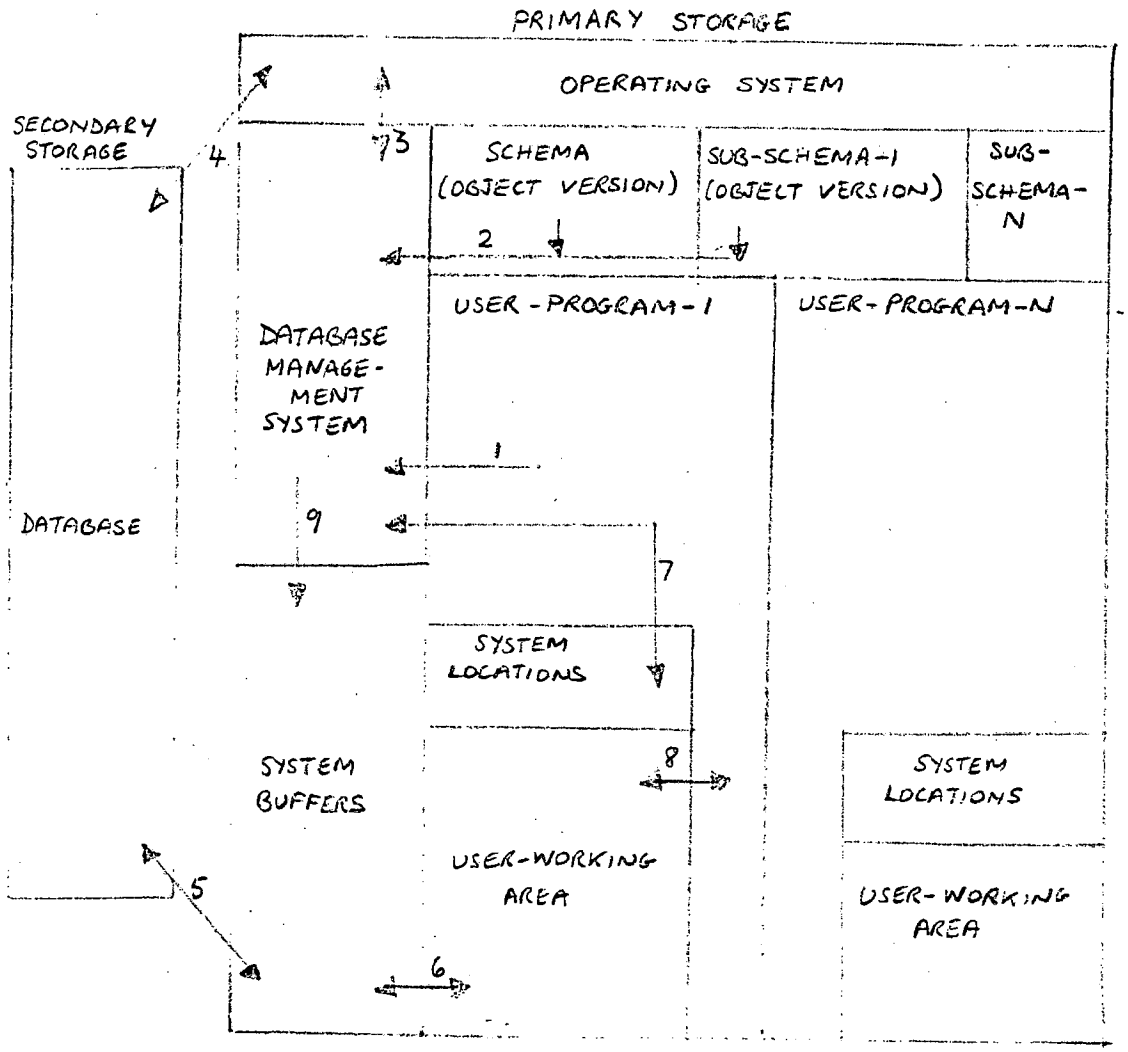


Figure 12.1 CODASYL's conceptual DBMS

The operations designated by the numbers 1 to 9 in Figure 12.1 are explained below.

1 a call for data by a user program to the DBMS. All calls for the services of the DBMS are made in the DML

2 the DBMS analyzes the call and supplements the arguments provided in the call itself with information contained in the object version of the schema for the database, and in the object version of the subschema invoked by the user program originating the call. The schema describes the database in terms of the characteristics of the data and the implicit and explicit relationships between data items. The subschema is a subset of the schema. It describes the data known to the program invoking it in the form in which the DBMS makes it available, and expects to find it, in that program's USER WORKING AREA (UWA). In this conceptual system it is assumed that the object version of the subschema contains only the differences from the schema and is not complete in itself. The source form of the schema is written in the schema DDL and the source form of the subschema is written in the subschema DDL.

3 on the basis of the call for its services and information obtained from the object version of the schema and subschema, the DBMS requests physical I/O operation, as required to execute the call, from the Operating System

4 The Operating System interacts with secondary storage

5 The Operating System transfers data between secondary storage and system buffers

6 The DBMS transfers data, as required to fulfill the call, between the system buffers and the UWA of the programs

originating the call. Any required data transformations between the representation of the data as it appears in secondary storage and the representation of the data as it appears in the program's UWA, are handled by the DBMS.

7 the DBMS provides status information to the calling program on the outcome of its call. The information provided is currency status information, error status condition codes, area name, record name.

8 data in a program's UWA may be manipulated as required, using the facilities of the host language.

9 the DBMS administers the System Buffers. The System Buffers are shared by all programs serviced by the DBMS. User programs interact with the System Buffers entirely through the DBMS.

It is clear from Figure 12.1 that the CODASYL DBTG has assumed that the DBMS would be implemented on a non-virtual Operating System. Under EMAS the user has access to his entire virtual memory (except the first 32 segments which are used by Director). It should be noted that other VM systems may have more than one protected area. In particular, in EMAS the user would have access to the System Buffers of Figure 12.1, in which EMAS would place the data retrieved from the database. CODASYL envisages that these buffers would be available only to the DBMS which would translate the data in them, according to the information contained in the schema and subschema, into the form required by the user and place it in the UWA. Furthermore, since the schema, subschemas, database indexes, etc. are required by the DBMS to service user requests, the files in

which they are contained would have to be connected, at least in read-only mode, in the user's virtual memory as would the database itself. This negates the fundamental concept of CODASYL, or indeed of any DBMS, that the user should only be permitted to access the data to which he is entitled. There are two possible solutions to this problem. The first is to place all, or at least part of EDAMS, in a priviledged (protected) section of the Operating System. The second is to introduce a separate EDAMS process which could communicate with the user process.

12.2 Placing EDAMS in a protected area of EMAS

The obvious choice for a protected area of EMAS is the director. The files containing all the privileged information could then be connected only in Director and the user process would not be able to access them directly. However, there would be at most 4 segments of Director available to EDAMS. The EDAMS routines themselves could be connected in read-only mode to the user's portion of VM, but the database itself (or those portions of it required at any moment), schema, subschemas, indexes, tables, backup files would all have to be connected into this comparatively small area. Obviously, they could not all be connected simultaneously and therefore the overhead of frequent file connection and disconnection would have to be considered. File connection is not an expensive procedure as it only involves checking access permission and noting the mapping information in the Director for that process. No access to

secondary storage is required until the file is actually used and no supervisor calls are made. However, there is potentially quite a high overhead involved for file disconnection. The mapping information in the Director is removed and a call made to the Supervisor to remove any pages written to in the file back from core or drum to disc. In fact, all pages belonging to that process which have been written to are removed back to disc for consistency.

12.2.1 Expansion of Director

Even if the EDAMS schema was not required on-line during execution (cf. IDMS) the saving of Director space would not be sufficient. It is therefore worth considering the possibility of expanding the Director to accomodate all the necessary information. Such an approach would be feasible given the structure of EMAS, but was rejected in favour of the simpler and neater system described in Section 12.3.

12.3 The EDAMS Master Process

A neater and more efficient way to implement EDAMS is to introduce a separate process, the EDAMS Master Process (EMP). This process would contain the entire DBMS, schemas, sub-schemas, tables, indexes and would also be the unshared owner of the database itself.

In order to implement the EMP, it would be necessary to make use of the EMAS inter-process communication facilities

(see Section 11.2). The 32 bytes allowed by EMAS for the message is clearly not enough to give even the simplest DML command. A Communication Area (CA), equivalent to the User Working Area of Figure 12.1, is required which would simply be a standard EMAS file connected in read and write shared mode by both the EMP and the user process. There would be a separate CA for each user process. The PON and POFF commands indicate the service and destination of the messages, i.e. from a user to EMP or vice versa. The EMAS message area itself gives the name of the CA. Details of users access rights, subschema in use, etc. would be contained in his CA. Also, the details of the DML request would be placed in the CA. A message indicating that a request had been made would be PONned on the message queue. The EMP would then POFF the information, execute the request using EDAMS, place the result in the CA and PON a message to the user indicating that the reply is available to be POFFed by the user.

Although the overhead of communicating with EDAMS via a message queue is greater than, say, simply calling an external routine, it is still considerably less than the connection/disconnection overhead of the Director solution. Moreover, the EMP provides a neater solution which fits in with the architecture of EMAS (only very minor alterations to Director are required to enable a process to use the inter-process communication facilities) and with the architecture of CODASYL-type DBMSs. It was therefore decided to implement EDAMS using an EDAMS Master Process.

The EDAMS Master Process operates in the same way as Brinch Hansen's monitors [52,53,54]. Monitor data is only

accessible to the monitor procedure and only one process can be progressing in a monitor at a time, during which time it has exclusive access to the monitoring data. In the same way, EMP data is accessible only to the EMP, and the EMP handler requests messages from the user processes one at a time.

It should also be noted that the use of a separate EDAMS Master Process has two further advantages:

(a) it is easy to ensure that seize blocks cannot be interrupted (see Section 10.2)

(b) it would be easier to move an EMP out to a stand-alone filestore which could be useful in highly-shared situations.

CHAPTER 13

STORAGE MAPPING IN EDAMS

13.1 Introduction

The mapping between subschema through schema to the physical database is of vital importance to the efficiency of EDAMS. In this chapter, the various levels of mapping will be discussed and the concept of the database map will be introduced.

13.2 Database map

The April 71 CODASYL Report implied that the owner/member, member/member set pointers should be embedded within the data records (except for pointer arrays), although the current DDLC JOD [22] makes no reference to implementation. However, most implementations of the CODASYL proposals (e.g. DMS 1100 [29]) do embed the set pointers in the records themselves. One notable exception is the PRIME DBMS [33]. Engles [55] has pointed out that such chained structures suffer from two main disadvantages:

(a) they are only as strong as the weakest link

(b) they can be time-consuming to search on direct access devices.

However, such structures do have many advantages. For example, records can be added and deleted relatively easily without moving other records about. It would therefore seem desirable

to incorporate the flexibility of chained structures, but to separate the data records themselves from the links which connect them. The PRIME DBMS [33] is an example of such a system; it uses B-trees to describe the set structures and these are stored in entirely separate files from the data. As regards Engles' first criticism of chained structures, the normal DBMS backup facilities for the data would of course also apply to the pointers regardless of where they are stored.

It is therefore proposed in EDAMS to store all pointers separately from the records themselves in a database map. A database map is a representation of a database (or portion of it) giving its structure and the relationships which exist between records, but where the actual data records themselves are replaced by their database keys. The EDAMS database key is similar to the 1971 CODASYL database key in that each EDAMS record is assigned a unique key for all time. It is envisaged that these keys would be generated by, say, a hashing function on record type, etc. In order to obtain the physical address of the record with a specified database key, a high-speed table look-up technique is required. For example, the hybrid technique described in [56], which is a combination of a hash and a binary tree, could be used.

There will be one database map per subschema. However, certain parts of the map may be common to several subschemas - namely information relating to schema sets and their owner and member records.

Consider, for example, the situation where user a, via subschema S1, adds a new member record occurrence to schema-defined set s. This addition must be visible to all users

of the set s, both those who access the database via subschema S1 and those who use other subschemas. It would be both difficult and costly to ensure that all the necessary changes were made to all the appropriate database maps. There are two possible solutions to this problem of database sharing and integrity:

(a) a common root section for all subschema database maps which describes the records and sets defined in the schema

(b) a separate database map for the schema.

The difficulty of the first approach is that it provides the subschema and hence, indirectly, the user with information which may not be relevant to the particular application. This is contrary to one of the main aims of the schema/subschema structure - namely, both for security and for achieving data independence, to give the user access only to the data which he actually requires for his own database application.

The second approach does not suffer from this particular difficulty as the schema database map is owned and used only by the DBA. Individual database maps will access the schema database map by means of cross references to it. Thus only those parts of the schema database map which are relevant to the particular subschema will be cross referenced.

The main objective of the database map (db map) is to make access to the database more efficient and also to simplify consistency (integrity) checking of the database. It also provides a convenient method of introducing sets at the subschema level.

It is envisaged, for example, that access keys could be contained in the nodes of the database map as well as database

keys. Thus access paths through the database could be traced until the desired record is found, thereby greatly reducing the number of accesses to secondary storage. The PRIME DBMS also permits the inclusion of search keys in its B-trees.

It is difficult to illustrate the concept of a db map diagramatically because of the multiplicity of pointers in all directions. The map can be thought of as being obtained by removing all the data from the record occurrences in a chained database and replacing it by a pointer (the database key) to the data. In EDAMS, the map would be based on subschema records.

Figure 13.1 shows the general set linkage structure in EDAMS

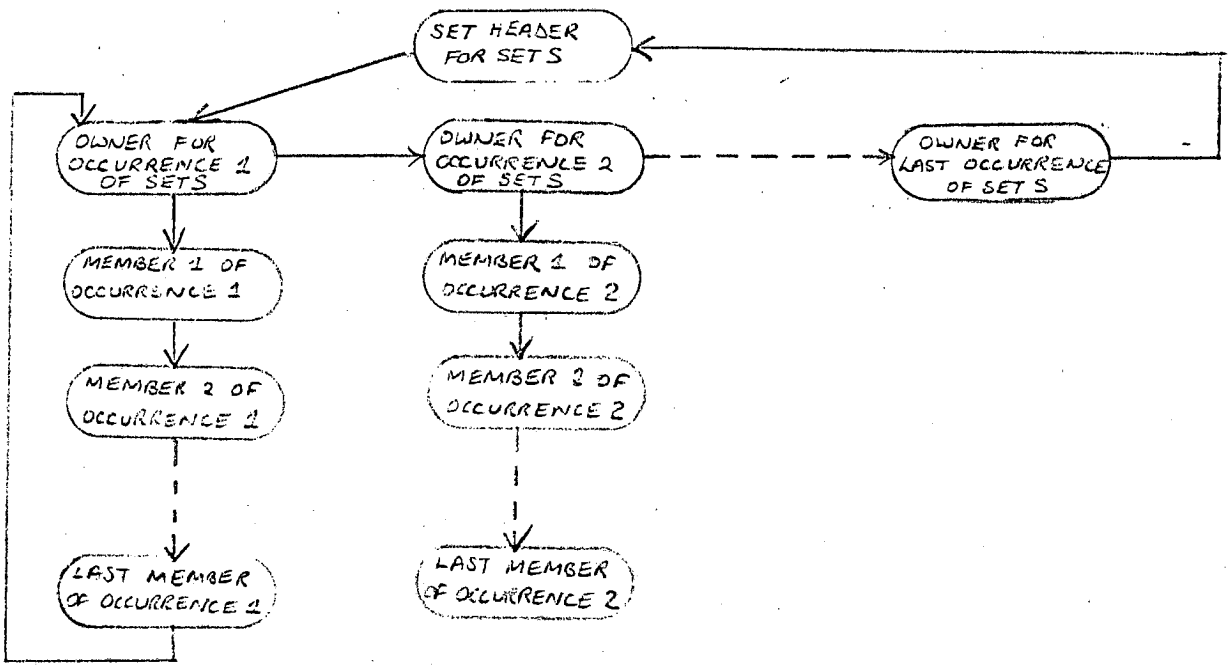


Figure 13.1 Set linkage structure in EDAMS

In addition, each member record could be linked to its owner and predecessor, as well as to its successor; also each owner could be connected back to the set header. This could also be implemented either by placing the nodes contiguously or by storing them in a tree structure.

In Figure 13.1 each node is shown separately, whereas in the database map, there would only be one node per subschema record occurrence. Figure 13.2 is an attempt to illustrate the database map for the example given in Figure 5.4.

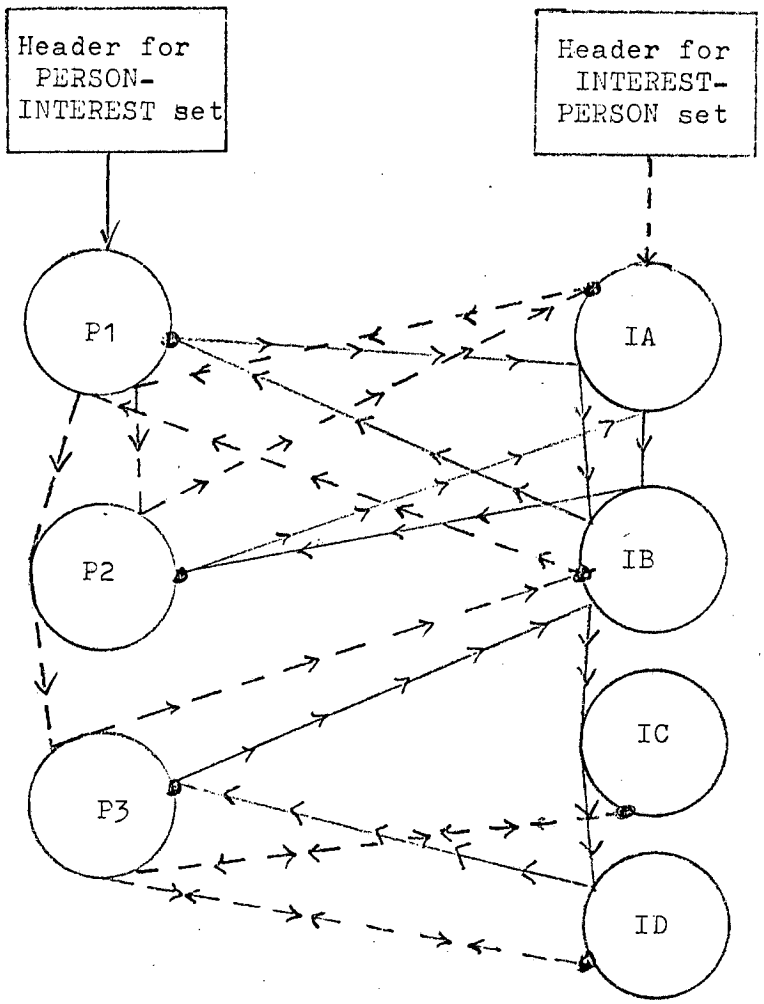


Figure 13.2 Diagrammatic representation of a sample database map

Thus each node in the map will have a number of pointers each one linking a records into a particular set occurrence. The

key to the representation of these pointers is the unique identification of every set and of every occurrence of every set across the database and the distinction made between owner and member records. Further classification of pointers would be necessary to indicate whether the pointer is a forward pointer or a backward pointer (for doubly linked sets) or a pointer to the owner. For the purposes of this example, only forward pointers will be considered. Note that in Figure 13.3 the pointer values following the identification information refer to table addresses. In reality, the database key would be used.

TABLE ADDRESS	DB KEYS	POINTERS			
1	P1	PI1-O-F-4	IP1-M-F-2	IP2-M-F-2	
2	P2	PI2-O-F-4	IP1-M-F-4	IP2-M-F-3	
3	P3	PI3-O-F-5	IP2-M-F-5	IP3-M-F-6	IP4-M-F-7
4	IA	PI1-M-F-5	PI2-M-F-5	IP1-O-F-1	
5	IB	PI1-M-F-1	PI2-M-F-2	PI3-M-F-6	IP2-O-F-1
6	IC	PI3-M-F-7	IP3-O-F-3		
7	ID	PI3-M-F-3	IP4-O-F-3		

where
PIi denotes the ith occurrence of PERSON-INTEREST set
IPj denotes the jth occurrence of INTEREST-PERSON set
O denotes owner record; M denotes member record
F denotes forward pointer

Figure 13.3 Tabular representation of sample database map

The number of entries in the entire database map shown in Figure 13.3 is equal to the number of records in the subschema database, which participate in sets. The possibility of

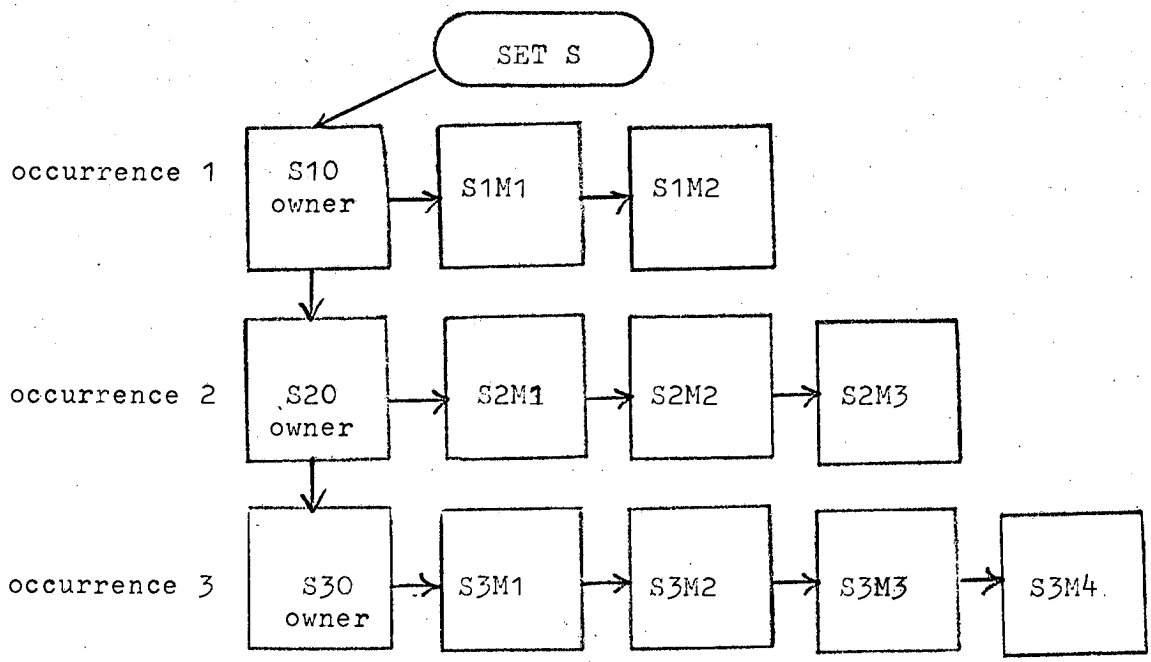
records in an EDAMS database which do not belong to any set is not precluded and they could readily be incorporated into the database map. Thus the map will be very large and some type of structure to facilitate speedy lookup based on database key will be essential [e.g. 56].

The CODASYL restriction that a record occurrence cannot appear in more than one occurrence of the same set does not apply to EDAMS. All EDAMS set occurrences are treated separately in the database map and thus there can be no ambiguity in interpreting the data structure.

However, this does alter the use and interpretation of DML commands such as FIND NEXT and FIND OWNER.

13.3 Interpretation of EDAMS DML

In the CODASYL proposals, DML commands such as FIND OWNER and FIND NEXT for a given set type, identify unique occurrences of records. This is not true in EDAMS since record occurrences may appear in more than one occurrence of the same set. Consider the diagram shown in Figure 13.4 overleaf.



where SiO is the owner record of the ith occurrence of set S
and SiMj is the jth member record of the ith occurrence of set S

Figure 13.4 Set occurrence structure

Now replace the symbolic record names, SiMj, by actual records as shown in Figure 13.5 below.

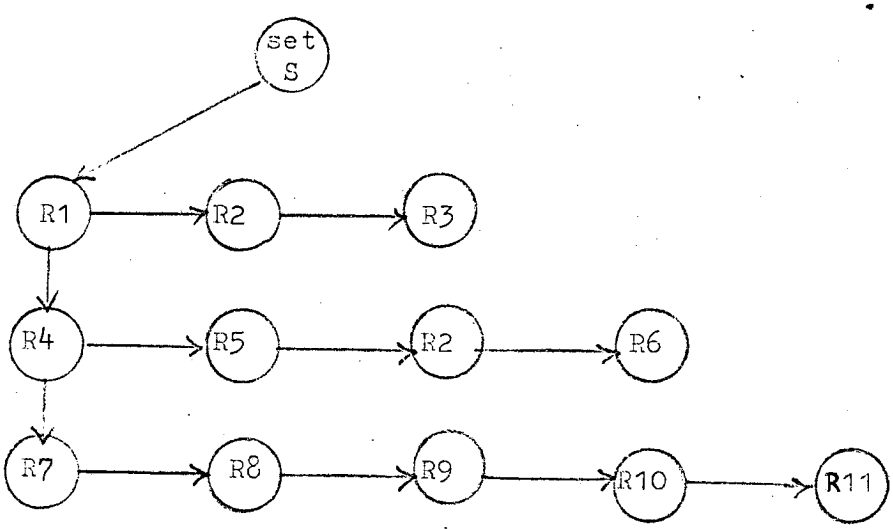


Figure 13.5 Example of actual set occurrences

Note that the record R2 appears in both the first and second occurrences of the set. A FIND OWNER for R2 (in the context of set S) under the CODASYL proposals would not know which of R1 or R4 to select.

If the application program is processing the database in the context of the first occurrence of the set S (EDAMS naturally maintains currency/context indicators in the same way as CODASYL) and issues a command to FIND OWNER of R2 (in set S) then the system will return R1. If, on the other hand, the program context is the second occurrence of the set, then R4 will be returned. However, if R2 has been reached either directly or in the context of its participation in another set, a request to

FIND OWNER OF SET S

will return R1. To locate R4, a new EDAMS command

FIND NEXT OWNER OF SET S

can be used. Whenever a FIND (NEXT) OWNER command is encountered, in addition to the owner record itself, a flag will also be returned. This flag will be set if another owner record occurrence is found for the particular member record occurrence. The cost of setting the flag is minimal and avoids an extra access to the database map when a FIND NEXT OWNER command is issued and there is only one owner. FIND NEXT OWNER can be used repeatedly to locate all the owners until the flag is returned unset.

An analogous situation exists in EDAMS for the FIND NEXT command. If the context of the set occurrence is clear, then there is no confusion. Thus, in the context of the first occurrence of the set S in Figure 13.5 (owner R1), with R2

the current record,

FIND NEXT RECORD OF SET S

will return R3. But if the context is the second occurrence of set S (owner R4) then R6 would be returned. As in the case of FIND OWNER, when R2 is not reached through set S,

FIND NEXT RECORD OF SET S

is ambiguous. EDAMS' solution is to return the next record in the first set occurrence in which R2 participates, i.e. R3. A flag is set to indicate that R2 participates in more than one occurrence of the set. To locate the next record in this second occurrence, R6, the user must issue a

FIND ALTERNATIVE NEXT RECORD OF SET S

As with the FIND (NEXT) OWNER command, EDAMS sets a flag to indicate that an alternative 'next record' exists.

13.4 EDAMS realms

In Section 5.3.1, the difficulties associated with CODASYL areas were discussed. The area performs directly or indirectly both the following functions:

(a) basic access and locking mechanism (concurrent update in addition to KEEP/FREE DML statements)

(b) provides the mapping between the database and Operating System files.

In Section 7.5, it was stated that in EDAMS a rigid distinction between the realm and the storage-area would be made. The realm is a logical subdivision of the database and exists only at the subschema level. Subschema records may

be assigned to one or more realms and thus realms may overlap. The EDAMS realm can therefore be thought of as a shorthand for referring to a group of logical records and will therefore be useful for privacy controls, concurrency controls and so on. In fact, the EDAMS realm can be treated and implemented as an ownerless set, forming part of the database map. Normal set operations can therefore be used to manipulate records within a realm, e.g.

FIND NEXT IN REALM R

FIND LAST IN REALM R

FIND FIRST IN REALM R

13.4.1 Mapping of EDAMS database to physical storage

The second role of the original CODASYL area has been replaced in EDAMS by the modern CODASYL storage-area. The mapping of the database to physical storage is part of the physical description of the database and is placed initially in the EDAMS schema. The aim of this mapping is to divide the database into segments which can be mapped conveniently onto EMAS files. The specification of the mapping must be based on physical rather than on logical entities and must be transparent to the user. The allocation of subschema records to realms corresponds to the allocation of schema records (or storage-schema records) to the storage-area.

CODASYL gives the DBA three alternative ways of specifying the record placement strategy:

(a) DIRECT

The database key specified determines the placement of the record.

(b) CALC

A database key is formed from the parameters of the command using either a user or system defined procedure.

(c) VIA

The placement of the record is determined by its membership of a set. To evaluate this option, the DBMS must use the SET OCCURRENCE SELECTION clause for the set.

In addition, for every record type defined in the schema, the user must specify a WITHIN clause indicating in which of one or more areas the record should be stored. In situations where an option is given, the application program must initialize the appropriate parameter with the correct area-name. In other words, the programmer is required to know in which area a particular record was stored.

Such an approach would not be possible in EDAMS since storage-areas are completely transparent to the user. Thus the system must be able to decide from its own information into which of a number of possible storage-areas a given record should be placed.

There is no natural way to map the logical database onto physical EMAS files. The simplest approach would be to allow the DBA total flexibility in the placement in storage of physical records. Thus in the schema DDL, DBA procedures could be specified which would decide in what storage-area to place an occurrence of a record. In these procedures, the DBA can make use of record type, the schema record key (see Section 7.2) or set membership details, for example. The full CODASYL VIA SET option could not be available under EDAMS because of the potential difficulty of uniquely identifying an owner record.

CHAPTER 14

DATABASE CONSISTENCY DURING UPDATE

14.1 Introduction

An important feature of a DBMS is how it ensures the consistency of the database in the event of a system failure and, in particular, of failure during an update operation.

To alter an item in a database may involve not only the changing of the data itself, but also the updating of tables and indexes. It is of vital importance that, if failure occurs, the database can be restored to the state which existed prior to the start of the uncompleted update.

EMAS alone cannot ensure this degree of consistency and hence EDAMS must provide the necessary facilities.

14.2 The effects of the on-line environment

An update in EDAMS is not secure until the Make Disc Consistent (MDC) routine (see Section 11.4) has been executed, i.e. updated page copied back to disc. Other concurrent users of an altered page see the new version in core and not the old version on disc.

In a batch environment, such a situation does not present a problem since if failure occurs before the update has been secured, other processes, which have used the "new" version of the record can be rolled back automatically. The majority

of users of EDAMS will be on-line and hence automatic rollback would be very difficult. For example, the system might find that a user who should be rolled back, or at least notified of failure, has logged off. The overhead involved in the execution of an MDC is not insignificant. However, in order to ensure consistency in an on-line environment, an MDC should be issued when the lock on a record, or group of records, is released.

In this way, each update (individual or group) will be complete in itself. However, what should happen in the situation where a program, which has been updating the database and whose updates are already secured, aborts? Clearly, in an on-line environment, there is no definitive answer to this question. As was indicated above, users who have logged off the system cannot be rolled back and indeed rollback of any interactive process, even if it is still active, is difficult. The solution to be adopted will depend more upon the application system than the DBMS. The best the DBA can hope to achieve is to insist on high and rigorous programming standards for users of the database, especially those permitted to alter its contents. A common approach, even in batch systems, is to debug application programs on a specially designed test database which incorporates as many of the "deviations" as possible in the main database. This was the system adopted in the University of Toronto Information System [57], for example. A deferred update system, such as is used in PRIME [33], whereby updates are written to a temporary file and the database is only updated when the group transaction is complete, can alleviate some of the difficulties of working in an on-line environment.

The only EMAS facility available to EDAMS is the MDC, which is inadequate. EDAMS really requires an MDC operation which is not page oriented, but operates only on specified extents in virtual memory, rather than on altered pages. However, EDAMS was designed to run on EMAS as it stands, without alteration. The problems with EMAS, as far as EDAMS is concerned, are discussed in detail in the last chapter of the thesis.

14.2.1 Simple update

Consider the problem of altering one data field within a record, assuming that this requires no movement of data or that such movement is confined to a single page.

(a) Run-unit (RU) obtains update lock on record

(b) RU processes record and calls on EDAMS to make change

(c) EDAMS through EMAS makes change - note that page is still in core

(d) RU releases lock on record.

The immediate execution of an MDC will make the update secure. Unfortunately, when the MDC is executed, the entire page is written to disc. Thus, changes made to that page by other partially completed transactions will be written back also. Although this does not necessarily present an integrity problem, it makes rollback in the event of failure considerably more complex, as will be explained in the next section.

14.2.2 Complex update

A complex update is one which involves consistent changes

to more than one page in the database, whether they are data pages and/or pointer/index pages. The problem arises if failure occurs during the MDC operation (whether automatically or manually triggered), e.g. pointer page updated, but not data page. The user can be made aware of what has happened but since he is not concerned with pointer files, indexes, maps, etc., he will not be in a position to do anything about it. Hence EDAMS will have to handle the situation and ensure the consistency of the database with the aid of a Journal File.

Traditionally, the Journal File was stored on magnetic tape, since this medium was much less vulnerable to failure than, say, magnetic disc and was also considerably cheaper. However, this has become less true and it is an increasingly common practice to use a small, dedicated disc for journalling. Each group update is assigned a unique transaction sequence number (TSN). The following sequence of events takes place:

(a) user successfully executes seize block and holds locks on required records

(b) start transaction block marker for TSN set on Journal File

(c) Record entry made on Journal File as follows:

TRANSACTION SEQUENCE NO.	RECORD ID	OPERATION TYPE	BEFORE/AFTER IMAGES OF RECORD

where

(i) there is an arbitrary number of these entries per transaction

(ii) RECORD ID includes page number in VM

(iii) OPERATION TYPE indicates update, deletion, etc.

(iv) BEFORE/AFTER IMAGES contain the minimum portion of

the record for update only.

(d) update performed

When user requests release of locks on records

(e) MDC executed

(f) End transaction block marker for TSN set on Journal File to indicate the successful completion of the update

(g) Locks on all records released.

During recovery the end transaction block markers can be checked. If the marker is not set, then EDAMS must examine the database using the information given in the before and after images of the record on the Journal to ascertain whether or not the update has in fact been carried out successfully. If not, then EDAMS must either complete the update or reset the record(s) (and tables/indexes) to their original state.

As in the case of the simple update, the execution of the MDC will also cause the changes made to the particular pages by other incomplete transactions to be written back to disc. Since the update is not officially complete (i.e. records are still locked) until the user releases the locks after execution of the MDC, the logical integrity of the database is ensured. However, in the event of failure, rollback can become quite complex. Any given page may contain the results of completed and partially completed transactions. In order to facilitate roll-back, it would be helpful to include a list of all pages involved in an MDC when the end of transaction block marker is set.

In addition to the Journal File (which must be 100% reliable) a Log of all other database activities - retrieval requests, console activity etc. - is maintained to provide a

complete record of DBMS usage for statistical purposes. Although of importance, the Log is not so vital to ensuring database integrity.as the Journal.

As part of the recovery facilities of EDAMS, it is envisaged that dumping of the entire database or of selected portions of it will be carried out at regular intervals. In the event of catastrophic failure, the database can be restored by rolling forward from the dump using the Journal File.

CHAPTER 15

IMPLEMENTATION OF CONCURRENT UPDATE ALGORITHM

15.1 Introduction

In order to fully evaluate the EDAMS algorithm for handling concurrent update, it was necessary to implement the algorithm using a test database. To do this, a basic core of EDAMS consisting of a Master Process and message communication facilities was required. A small test database containing 26 records of 5 different types was set up. In many database applications, 90% of the accesses are made on 10% of the data and the purpose of this implementation is to extract this 10% and scale down. The Master Process maintained the search engine, process queues and lock lists for the concurrent update algorithm.

15.2 Message communication

In order to implement the concurrent update algorithm, it was necessary for the EDAMS Master Process (EMP) to handle four types of service requests from user processes:

(a) service indicating that the user had entered a seize block
(b) service indicating a lock request (locking predicate)
(c) service indicating the end of a seize block
(d) service indicating the release by a process of all its currently locked records.

Thus the sequence of requests by any one user process would be abb...cd. The sequence a to d constitutes one transaction. Note that any number of lock requests, (b), can be enclosed between the beginning and end of a seize block. There could also be an abort transaction service, but this was not implemented. As was explained in Section 10.2, it is not necessary to insist that a process release all its records simultaneously, as the EDAMS search engine does not examine locked records. Thus the snapshot obtained by an updating process will automatically reflect none of the updates (if it arrives first at the seize block) or all of the updates of a second concurrently updating process. However, for simplicity, in this implementation it was decided to release all records simultaneously (service (d)).

To implement the four service requests, four routine calls are required at the DML level:

(a) SEIZE

(b) LOCK

(c) ENDSEIZE

(d) RELEASE

15.3 Time clock

In a complete EDAMS system, a number of users would be using the system simultaneously. Hence service requests would be arriving at the EMP in a pseudo-random fashion from all users. Clearly, the time interval between service requests for a given user will vary greatly and will depend, among other things, upon the type of request. Thus, for example, one

would expect a certain time interval, t1, between the user logging on and entering his first seize block. This would be followed by a probably shorter time interval, t2, prior to the issue of the first lock request. There would be an average interval, t3, between lock requests with a shorter interval, similar to t2, before the ENDSEIZE. One would expect a much longer time interval, t4, before the user releases all his locked records. It is during this time that the actual update is carried out.

Broad assumptions could be made as to the relative lengths of the various time intervals t1, t2, t3 and t4, but other factors such as "thinking time", typing speed etc. if the user is working truly interactively will play an important part. Rather, therefore, than attempting to devise an elaborate time clock mechanism, it was decided simply to use a random number generator to decide from which user the next message to the EMP would come. Naturally, the messages from each individual user must follow the sequence described above.

15.4 Actions required by EMP

When a process enters a seize block, the only action taken by the EMP is to place the process on the queue and set its status to active, i.e. not blocked.

When a process issues a lock request, the request is placed in a buffer by the EMP and control is passed back to the user. This continues until the user process issues an ENDSEIZE request, at which time the EMP will service all the lock requests (in that seize block) for that user. Essentially, this

consists of ascertaining whether or not the lock request(s) can be granted. If so, the database keys of the requested records are placed together with the user name of the requesting process on the list of currently locked records, namely the lock list. If the request(s) cannot be granted because one or more of the records is already locked by another process (i.e. it appears on the lock list with another user name), then the status of the requesting process is set to blocked on the queue. Furthermore, the process is rolled back to the first lock request in the seize block and all records currently held by that user in the lock list (as a result of previous successful requests within the same seize block) are taken away.

Three types of update were considered:

(a) basic type consisting of a list of the records the user wishes to lock

(b) content-based lock request, e.g.

LOCK ALL EMPLOYEE RECORDS FOR WHICH DEPARTMENT FIELD = 40

(c) path-tracing - lock all records on a content-dependent path; this type of request would also cover, for example, locking an entire set occurrence.

A basic lock request of type (a) consists of a list of the records to be locked. Each record in the list is immediately and uniquely identifiable without involving access to the database itself, e.g. using a key which can be translated directly into a database key. The action required by the EMP is simply to check this (generated) list of database keys against the lock list. If any one record appears under a different user name, the failure of the request is signified, otherwise success.

Lock request type (b) consists of a record type name

followed by the name of a field within that record, followed by an upper and lower bound for the value into which the field must fall to satisfy the lock request e.g.

EMPLOYEE RECORD DEPARTMENT FIELD BETWEEN 0 AND 10

This will attempt to lock all employee records whose department code is in the range

$$0 \leq \text{DEPARTMENT CODE} \leq 10$$

The action taken by the EMP is to examine all the department fields of all employee records and make a list of the database keys of all those which fall within the given bounds. This list is then checked against the lock list to ascertain whether or not the request is successful.

The lock request type (c) consists of the unique identification of the record at which the path tracing algorithm is to start followed by the length of the path. This is a somewhat artificial representation of the real situation, where the user would fix position in the database, move along the path and finish when a particular record is reached. However, as regards the concurrent update algorithm, a path length represents an analogous method of terminating the lock request. Moreover, for testing purposes, a random number generator was used to determine each node in the path. The database key of each node is noted and then checked against the lock list as in type(b) above.

When the EMP can successfully grant all the lock requests for a user in a seize block, the user is granted the locks and allowed to proceed outside the seize block. The process is removed from the queue.

For ease of implementation, the processes release all their currently locked records simultaneously before entering another

seize block. This is easily accomplished with the structure of EDAMS. Firstly, all the process' entries on the lock list are removed. A message is sent to the user to proceed. Secondly, the location of the first blocked process, if any, on the queue is found. Its status is then set to active and the process restarted, i.e. instead of using the random number generator to calculate where the message is coming from, the implementation forces it to be the first blocked process on the queue. The reason for this is that it is only when a process releases locked records that there is any point in restarting blocked processes.

If the process remains blocked then the search engine finds the next blocked process on the queue. If its priority difference with the head of the queue is less than a certain threshold value, the search engine will attempt to release it and so on down the queue until there are no more blocked processes or the threshold is reached. On completion of its set of lock requests, a process is assigned a time priority of zero. Thus if the priority of the process at the head of the queue is less than the threshold, the search engine will attempt to release the incoming process, otherwise it is placed at the end of the queue.

To restart the seize block for a process is a simple matter, which is completely transparent to the user, EDAMS must store all the lock requests for each user in a buffer until the ENDSEIZE command is reached. Thus to restart a seize block all that is required is simply to reposition the message pointer to the beginning of the buffer for that user.

15.5 Results for test runs of concurrent update algorithm

The algorithm was run with a random mix of concurrent update request in the test database with

(a) 5 users

(b) 10 users

(c) 15 users

The occasional request to lock almost the entire database was inserted.

The problems associated with a realistic time clock and hence of the priority threshold system have already been discussed in Section 15.3. On completion of a set of lock requests in a seize block, a user is assigned a priority of zero. This is incremented by one for every incoming command to the EMP (issued by other active users) until the user is released. Two threshold values - 5 and 10 units - were selected and were compared with a threshold of zero, which corresponds to a first-come-first served operation (FCFS). Note that a very high threshold value corresponds to the situation where the search engine attempts to release all users on the queue in order, irrespective of their priority relative to the head of the queue.

seize block number	user no.	records requested	no. of failures	priority on release
1	3	2 7	0	0
2	4	8 9 10 11 22 4	0	0
3	5	7 15 18	0	0
4	1	1 15 13 20	0	0
5	2	1 8 12 23 9 14 22 26 25 24 3 10 20 18	1	2
6	1	1 2 12 19	0	0
7	4	9 11 5 25 12 2 13	0	0
8	3	1 3 4 6 12 15 24	0	0
9	4	21 2 4 26 1 15 23 20 17 6	0	0

Figure 15.1 Results for 5 users for all threshold values - 0, 5 and 10

seize block number	user no.	records requested	no. of failures	priority on release
1	5	7 15 18	0	0
2	1	1 19 13 20	0	0
3	9	entire database	1	14
4	8	5 4 8 23	0	12
5	2	1 8 12 23 9 14 22 26 25 24 3 10 20 18	1	11
6	4	8 9 10 11 22 4	1	11
7	3	2 7	0	9
8	7	7 2 13 24	1	10
9	10	1 8	1	11
10	6	1 9 11	2	11
11	1	1 2 12 9	1	1
12	9	3 4 5	0	0
13	2	9 11 5 25 12 2 13	1	6
14	8	2 6	1	5
15	6	1 2 3 4	1	4
16	3	1 3 4 6 12 15 24	1	1
17	4	21 2 4 26 1 15 23 20 17 6	1	2
18	8	24 25 26	0	0

Figure 15.2 Results for 10 users for threshold = 0 (first-come-first-served

seize block number	user no.	records requested	no. of failures	priority on release
1	5	7 15 18	0	0
2	1	1 19 13 20	0	0
3	8	5 4 8 23	0	0
4	9	entire database	2	13
5	2	1 18 12 23 9 14 22 26 25 24 3 10 20 18	3	9
6	3	2 7	1	7
7	4	8 9 10 11 22 4	3	12
8	7	7 2 13 24 1 26	2	8
9	10	1 8	2	11
10	6	1 19 11	1	10
11	1	1 2 12 19	1	1
12	9	3 4 5	0	0
13	8	2 6	0	0
14	4	9 11 5 25 12 2 13	2	7
15	6	1 2 3 4	1	2
16	8	24 25 26	1	1
17	3	1 3 4 6 12 15 24	2	2
18	4	21 2 4 26 1 15 23 20 17 6	1	1

Figure 15.3 Results for 10 users for threshold=5

seize block number	user no.	records requested	no. of failures	priority on release
1	5	7 15 18	0	0
2	1	1 19 13 20	0	0
3	8	5 4 8 23	0	0
4	3	2 7	0	0
5	4	8 9 10 11 22 4	1	4
6	2	1 8 12 23 9 14 22 26 25 24 3 10 20 18	3	9
7	9	entire database	5	19
8	7	7 2 13 24 1 26	1	11
9	10	1 8	2	12
10	6	1 9 11	2	10
11	1	1 2 12 19	1	1
12	4	9 11 5 25 12 2 13	1	1
13	8	2 6	1	4
14	3	1 3 4 6 12 15 24	0	0
15	9	3 4 5	1	5
16	8	24 25 26	1	4
17	6	1 2 3 4	3	5
18	4	21 2 4 26 1 5 23 20 17 6	4	4

Figure 15.4 Results for 10 users for threshold=10

seize block number	user no.	records requested	no. of failures	priority on release
1	4	8 9 10 11 22 4	0	0
2	3	2 7	0	0
3	11	1 9 26 2 4 8	2	8
4	9	entire database	1	22
5	12	3 5 7 1	1	34
6	13	9 8 7 6	1	40
7	10	1 8	1	38
8	5	7 15 18	0	37
9	15	1 3 5 7 9 11	2	34
10	8	5 4 8 23	1	27
11	2	1 8 12 23 9 14 22 26 25 24 3 10 20 18	1	27
12	1	1 19 13 20	1	25
13	4	26 19 24 1 9	1	23
14	7	7 2 13 24 1 26	1	22
15	11	8	0	19
16	6	1 19 11	2	26
17	4	9 11 5 25 12 2 13	1	27
18	3	1 3 4 6 12 15 24	1	25
19	9	3 4 5	1	24
20	11	2 7	0	4
21	14	4 8 22	2	6
22	1	1 2 12 19	0	4
23	6	1 2 3 4	2	5
24	8	2 6	1	6
25	11	1 2 3 4	1	4
26	4	20 2 4 26 1 15 23 20 17 6	1	3
27	8	24 25 26	0	0

Figure 15.5 Results for 15 users for threshold=0 (first-come-first-served

seize block number	user no.	records requested	no. of failures	priority on release
1	4	8 9 10 11 22 4	0	0
2	3	2 7	0	0
3	12	3 5 7 1	1	4
4	11	1 9 26 2 4 8	3	9
5	9	entire database	4	25
6	10	1 8	1	26
7	5	7 15 18	1	25
8	13	10 9 8 7 6	3	40
9	15	1 3 5 7 9 11	2	35
10	8	5 4 8 23	1	27
11	1	1 19 13 20	1	22
12	2	1 8 12 23 9 22 26 25 24 3 10 20 18	2	28
13	14	26 19 24 1 9	1	22
14	11	8	0	16
15	7	7 2 13 24 1 26	2	23
16	4	8 9 10 11 22 4	0	16
17	6	1 9 11	4	29
18	9	3 4 5	0	21
19	3	1 3 4 6 12 15 24	2	24
20	11	2 7	0	6
21	14	4 8 22	1	1
22	1	1 2 12 19	1	1
23	4	21 2 4 26 1 15 23 20 17 6	1	3
24	6	1 2 3 4	1	3
25	11	1 2 3 4	1	3
26	8	2 6	1	2
27	8	24 25 26	0	0

Figure 15.6 Results for 15 users for threshold=5

seize block number	user no.	records requested	no. of failures	priority on release
1	4	8 9 10 11 22 4	0	0
2	3	2 7	0	0
3	12	3 5 7 1	1	4
4	11	1 9 26 2 4 8	3	9
5	9	entire database	4	25
6	10	1 8	1	26
7	5	7 15 18	1	25
8	13	10 9 8 7 6	3	40
9	15	1 3 5 7 9 11	3	35
10	8	5 4 8 23	3	27
11	1	1 19 13 20	1	22
12	2	1 8 12 23 9 14 22 26 25 24 3 10 20 18	4	28
13	14	26 19 24 1 9	4	22
14	11	8	1	16
15	7	7 2 13 24 1 26	5	23
16	4	8 9 10 11 22 4	3	16
17	9	3 4 5	1	15
18	6	1 9 11	5	29
19	3	1 3 4 6 12 15 24	3	24
20	11	2 7	0	6
21	14	4 8 22	1	1
22	1	1 2 12 19	1	1
23	4	21 2 4 26 1 15 23 20 17 6	1	3
24	6	1 2 3 4	2	3
25	11	1 2 3 4	1	3
26	8	26	1	2
27	8	24 25 26	0	0

Figure 15.7 Results for 15 users for threshold=10

Threshold	No. of failures 0	1
all values	8	1

Figure 15.7 Breakdown of number of unsuccessful attempts to execute seize blocks for 5 users - 9 seize blocks

Threshold	No. of failures 0	1	2	3	4	5
0	6	11	1			
5	5	6	5	2		
10	5	7	2	2	1	1

Figure 15.8 Breakdown of number of unsuccessful attempts to execute seize blocks for 10 users - 18 seize blocks

Threshold	No. of failures 0	1	2	3	4	5
0	7	15	5			
5	7	11	5	2	2	
10	4	11	1	6	3	2

Figure 15.9 Breakdown of number of unsuccessful attempts to execute seize blocks for 15 users - 27 seize blocks

Threshold	Average no. of failures per seize block	Average priority on release
all values	0.1	0.2

Figure 15.10 Average number of failures and average priority on release per seize block for 5 users

Threshold	Average no. of failures per seize block	Average priority on release
0	0.6	6
5	1.2	4.7
10	1.4	4.9

Figure 15.11 Average number of failures and average priority on release per seize block for 10 users

Threshold	Average no. of failures per seize block	Average priority on release
0	0.9	18.1
5	1.3	15.2
10	2.0	15.0

Figure 15.12 Average number of failures and average priority on release per seize block for 15 users

15.6 Analysis of the results

In order to fully evaluate the efficiency of an algorithm for handling concurrent update, it would be preferable to do it in a "live" situation. As this was not possible, it was decided to use a small test database with several users whose record requests overlapped considerably. Even in a large database with several users running concurrently, one algorithm will perform much the same as another if their requests do not overlap. However, there is evidence to show that in many applications there is considerable clustering of update requests, both in time and locality. For example, when a horse closes its entry for a race [58], the horse's racing history and the owner's and trainer's accounts must be updated and the details for the race altered. Such transactions arrive at the rate of one per second throughout Thursday and Friday mornings and at a higher rate before a Bank Holiday weekend. This is in addition to the other normal activity in the Horse Racing Administration System at Wetherbys, such as foalings, registrations, etc. Moreover, two or more horses may close for the same race at the same time and often for the same owner or trainer for different races. Thus the test situation used in EDAMS is not totally unrealistc with several users updating a small number of records.

The reason for the double peaks of activity in each of the runs (e.g. Figures 15.3 and 15.4) is that the experiment was conducted in such a way that all users start from scratch by entering seize blocks and finish by releasing locked records. Although the individual commands from users arrive at random,

the command sequence is the same for each user. Thus, at the beginning of each run all users will be entering seize blocks, then locking records and then releasing them, mainly before the second set of seize blocks for each user is started. In a "live" situation at any point in time, one would expect that users would be at various different stages in execution, i.e. not all entering seize blocks. It would have been preferable to use a randomly staggered start and collect statistics in the middle of the test run.

15.6.1 First-come-first-served

The first-come-first-served (FCFS) operation corresponds to an EDAMS Priority System with threshold=0; i.e. only the head of the queue can be released even if the lock requests of processes further down the queue are distinct. Thus no attempt is made to evaluate a user's lock requests unless it is at the head of the queue. In this way, the number of unsuccessful evaluations of seize blocks is minimized, but so also is the degree of concurrency with only one user active most of the time. The results show (Figures 15.1, 15.2, 15.5, 15.7-15.12) that even in a moderately concurrent situation, represented by 10 users, the average priority on release is considerably higher for FCFS than for the set threshold in the EDAMS Priority System.

15.6.2 EDAMS Priority System

In spite of the rather artificial priority mechanism used, the trend of the results for the EDAMS Priority System with increasing threshold value is clear. A very high (infinite)

value of the threshold corresponds to the situation where the lock requests for all processes on the queue are checked each time a user releases locked records. The effect of this is very clear in the 15 user run (Figures 15.9 and 15.12). A small reduction in the average priority on release is accompanied by a very large increase in the number of unsuccessful executions of the seize blocks.

CHAPTER 16

CONCLUSIONS

16.1 Introduction

The objectives of this thesis as outlined in Section 1.4 are threefold, namely, to show that

(a) it is feasible to implement a CODASYL-type DBMS on a Virtual Memory (VM), multi-access Operating System, such as the Edinburgh Multi-Access System (EMAS),

(b) it is possible, within the overall CODASYL framework, to provide the user with much greater flexibility in the creation of logical records whose fields can be drawn from all over the database without restriction,

(c) an efficient and simple algorithm can be devised for solving the problem of contention between users during concurrent update of the database.

In this final chapter, the degree to which these objectives have been met in the thesis through the design of EDAMS (EMAS Database Management System), will be discussed, together with the difficulties encountered in meeting them.

16.2 The implementation of EDAMS on EMAS

In Chapters 11,12 and 13, it was shown how a CODASYL-type DBMS could be implemented on a VM system such as EMAS. The VM system offers the DBMS designer many advantages, especially

with regard to the automatic management of memory and ease of implementation programming. However, a number of difficulties were encountered, which will be summarized below.

16.2.1 Privacy and security

EMAS has two levels of access to a process' VM. The first 32 segments (0-31) of a user's VM contain the Director to which only the system has access. The remaining segments (32-255) can be accessed by both the user and the system. Such a two-tiered structure presents privacy problems for the DBMS designer. For example, consider a user process requesting a record which for simplicity is identical to a physical record in the database. If the EMAS file mapping facilities were used, then the entire file containg the requested record, would be mapped onto (connected into) the user's VM. In this way, the user could have unrestricted access to the whole file.

In order to ensure privacy, it is therefore necessary to map the database, or portions of it, onto parts of the VM to which the user does not have direct access. In EMAS, there are two possible solutions to this problem, namely to place the data in either

(a) Director or

(b) another process' VM.

These two approaches are discussed in detail in Chapter 12. Essentially, the problem with the first solution, that of mapping the database onto Director, is the limited space available. Some of the 32 segments are already used for system and file information and there would not be sufficient left for database

connection as well as the indexes, subschema and schema tables and so on, which would also have to be protected. It would be possible to expand Director, but it was decided to adopt the second solution, namely the use of a second process. This process is called the EDAMS Master Process, EMP. The EMP can be regarded as the DBMS. All requests for service by the DBMS are passed to the EMP via the inter-process message communication facilities in EMAS. All data, tables, indexes, etc. are connected into the EMP's VM, before being passed back to the user. A Communication Area is set up between the EMP and each user process (simply an EMAS read-write shared file) to contain database requests, replies and so on.

16.2.2 Database integrity

One of the main problems encountered when implementing the DBMS on EMAS concerned the difficulty of ensuring database integrity during update of the database. An update in a VM system cannot be considered complete and secure, until all the pages involved have been written back to secondary storage. The EMAS service of significance is known as the Make Disc Consistent (MDC). When requested, this service writes back to disc all pages altered by a particular process. EMAS automatically uses the MDC when either the process' working set of pages in core changes or when a user file is disconnected (including when a user logs off).

The MDC, as it stands, is too blunt an instrument for direct use by the DBMS to ensure DB integrity during update. The pages written back to disc as a result of the MDC could contain partially completed as well as totally completed

transactions.

The solution adopted by EDAMS is to make use of the Journal File. Every update transaction is assigned a Transaction Sequence Number, TSN. Once a user process holds the locks on all the records involved in the transaction, an entry is made on the Journal File containing the TSN, before and after images of the record(s), page number in VM, and so on. Once the entire update is complete and secure (MDC for the process executed), an End of Transaction for that TSN is set on the Journal File. Thus in the event of failure, rollback can be initiated.

As a result of the fact that the MDC is page-oriented, such rollback will be quite complex as those pages written back to disc following an MDC, may contain partially completed transactions belonging to other processes as well as completed transactions. The situation would be greatly simplified if the MDC could be much more selective, based on extents in VM. In this way, only the actual records involved in the transaction will be written back to disc. It is understood [59] that such an extent-based MDC routine could be incorporated into EMAS and this would greatly facilitate the maintenance of DB integrity during update, especially when rollback following failure is required.

16.3 Flexibility of the EDAMS data model

A major contribution of EDAMS to the design of a CODASYL-based DBMS is to provide the user with much greater flexibility.

This flexibility is brought by allowing the user to form subschema logical records (SLRs) which can be composed of fields taken from any record(s) in the parent schema. An obvious extension of this is to allow the user to define new sets in the subschema to link the SLRs together.

The introduction of the SLR poses a number of problems for the design of EDMAS such as:

(a) inclusion/exclusion of sets in the schema

(b) identification of source schema records for definition of SLRs

(c) operations on SLRs.

16.3.1 Sets in schema

With the introduction of the SLR and subschema sets, the question arose as to whether or not sets in the EDMAS schema were necessary. It was felt that the relationships between the data (either implicit or explicit) are as much part of the database as the data itself. If schema sets were excluded, then the information concerning these relationships would have to be repeated in each subschema which required them. Moreover, a fundamental concept in the use of databases is that of sharing and the elimination of unnecessary redundancy. It was therefore decided to retain the schema set in EDMAS and to augment it by allowing new sets to be defined in the subschema.

However, the retention of the schema set poses problems in relation to its use in the SLR environment of the subschema. For example, if a group of SLRs contained a mixture of fields from both the owner and the member records of a schema set, the

use of that set to link together the SLRs could be confusing. It would be difficult to identify which SLR should be the owner and which a member. Thus the subschema records defined as forming part of a schema set must be subsets of their parent schema records, i.e. single-source SLRs. In this way, there will be no ambiguity as to the use of the set in the subschema.

16.3.2 Definition of SLRs

In order to define a new SLR type in the subschema DDL, it is necessary to identify the source schema records and when the new set of SLRs is generated to uniquely identify the particular group of schema record occurrences which provide the sources of a given SLR occurrence (cf. CODASYL set occurrence selection).

The solution to this problem of source record identification adopted by EDAMS incorporates some useful features of the relational approach to DBMS. There are two methods, the first is based on records and the second on sets.

The record-based approach consists of expressing the rules for the formation of a set of SLRs in terms of the relational JOIN and PROJECT operations. For example, consider the sample database given in Figure 16.1.

SCHEMA

NAME	ADDRESS	EMPNO

PERSONAL-INFO

EMPNO	CURSAL	1stSAL	2ndSAL

PAY-HISTORY

SUBSCHEMA FOR PAYROLL APPLICATION

NAME	ADDRESS	CURSAL

PAYREC SLR

Figure 16.1 Portion of a sample database

Having defined the source fields for PAYREC, e.g.

DEFINE RECORD TYPE PAYREC;

FIELD 1 IS NAME; SOURCE IS NAME FIELD OF RECORD TYPE PERSONAL-INFO; etc.

the relational operators are used:

JOIN PERSONAL-INFO, PAY-HISTORY ON EMPNO TO FORM TEMPREC;

PROJECT TEMPREC OVER NAME, ADDRESS, CURSAL TO FORM PAYREC;

The result of the JOIN operation is a set of records, merged on the EMPNO field, each containing

NAME, ADDRESS, EMPNO, CURSAL, 1stSAL, 2ndSAL.

The PROJECT operator is then used to select only those fields required, namely NAME, ADDRESS and CURSAL. Note that in this example, the JOIN is an EQUIJOIN, i.e. only one pair of schema records for each value of EMPNO. If, however, the "joining" field is non-unique, then the EDAMS rule is to generate all possible pairs.

The second approach to the formation of SLRs is based upon the set membership structure of the parent schema records from which the SLRs are derived. Suppose the following set

structure existed for the sample database given in Figure 16.1 above:

PERSONAL-INFO

EMPLOYEE SET

PAY-HISTORY

Figure 16.2 EMPLOYEE set structure in schema

The source field definition of the PAYREC SLR in the DDL would be slightly different, e.g.

DEFINE RECORD TYPE PAYREC;

FIELD 1 IS NAME; SOURCE IS NAME FIELD OF RECORD TYPE PERSONAL-INFO OWNER OF EMPLOYEE SET; etc.

The set based JOIN and PROJECT commands for forming PAYREC are then:

JOIN PERSONAL-INFO, PAY-HISTORY THRU SET EMPLOYEE TO FORM TEMPREC;

PROJECT TEMPREC OVER NAME, ADDRESS, CURSAL TO FORM PAYREC;

16.3.3 Operations on SLRs

All access to the EDAMS database must be via a subschema. Thus all storage, retrieval and update operations are carried out on SLRs.

Retrieval of an SLR is straightforward and consists merely of retrieving the fields from the source records and putting them together to form the SLR before passing it to the user.

Update, in the sense of the alteration of an existing SLR field value, is also apparently straightforward. However, it can have undesirable repercussions. Suppose, for example, the field is the key to a JOIN operation in the formation of

that, or any other SLR, then it is clear that inconsistencies could result. Indeed, it is shown in Section 9.3.2 that the field to be updated need not even be the key to a JOIN operation for problems to occur. It was found that the only way to guarantee the integrity of the database was to restrict update operations to single-source SLRs. It is recognized that this is an unfortunate restriction as many updates can be carried out successfully on multi-source SLRs and it does remove a degree of flexibility.

EDAMS distinguishes between two types of operations to create an SLR. The first, called addition, simply involves the establishment of the links between a new SLR and its source schema records, which already exist in the database. This is quite straightforward. The second creation operation, called storage, is quite different in that it involves the physical addition of new data to the database. The source schema records for the SLR have to be created and fields in those source records which do not form part of the SLR, assigned null values. This could result in a proliferation of schema records in the database whose fields are largely unassigned. Problems analagous to those which arise with update can also occur when a second SLR is stored which contains some of the unassigned values and some of those assigned by the first SLR. Once again, the solution is to restrict the store operation to SLRs which are strict subsets of a single parent schema record.

Corresponding to addition and storage of SLRs are removal and deletion. Removal only involves the removal of the SLR from the user's view, with no physical deletion of data from the database. Deletion, on the other hand, does involve

the physical deletion of data from the database. As before, difficulties can arise, so the deletion operation is once again restricted to single-source SLRs.

16.3.4 Database maps

In order to facilitate the formation of SLRs and the definition of sets in EDAMS, pointers indicating source fields, set linkages and so on are stored separately from the data in a database map. Essentially, a database map can be regarded as a representation of a subschema's view of the database, but where the actual data is replaced by pointers to where it is stored.

Much of the overhead in database processing involves following pointers, looking up indexes and so on, even before any physical data is retrieved from the database. Any approach which can enable this table look-up to be speeded up will increase the overall efficiency of the DBMS. The database map, which will of necessity be quite large, is intended to do this. Moreover, it is anticipated that since it will be used so frequently, it will be permamently connected in the EMP's VM (either in core or on drum). There is one database map per subschema plus a root map for the schema.

16.4 Concurrent update in EDAMS

A completely new algorithm is given for solving the problem of contention between users of a database. The aim of the algorithm is to maximize concurrency without imposing

too high an overhead.

The algorithm makes use of locks on records and a user must hold all the locks for all the records involved in an update transaction before being released to perform the actual update. The claiming of these locks is done in a special section of the application program, known as a seize block. Only one user can be executing a seize block at a time - this is facilitated by the use of the EMP described above. If a process A, currently executing a seize block, attempts to lock a record which is already locked by another process B, then process A is suspended and all locks which it has already claimed in that seize block released. Furthermore, process A is placed on a queue of blocked processes, the ordering of the queue being determined by the process' time of arrival at a seize block. Once process B has completed its update, it will release all the locks it holds simultaneously. The system will then go down the queue of blocked processes, attempting to satisfy their locking predicates and release them. If the process at the head of the queue is still blocked (i.e. a third process C holds the locks required by the head of the queue), then the system will attempt to release the next process on the queue and so on until the difference between the waiting time priority of the process to be considered for release and the waiting time priority of the head of the queue exceeds a given threshold. A process is assigned a waiting time priority of zero when it is first placed on the queue of blocked processes and it increases with time spent in the queue. In this way processes cannot be held up indefinitely, while at the same time, processes down the queue whose lock requirements are simple, will not be held up

unnecessarily by processes wishing to perform complex updates involving large numbers of records.

16.4.1 Evaluation of the algorithm

The degree of concurrency to be achieved by EDAMS will depend upon several interdependent factors including:

(a) number of users concurrently updating the database

(b) extent to which the lock sets of users overlap

(c) timing of lock requests.

For example, consider the extreme case of only two users concurrently updating a database of 1 million records. Statistically, the chances of these two processes wanting to update the same record at the same time are very small, but yet, it is feasible that they could hold each other up continuously if there lock sets happen to overlap in a certain way. Moreover, it is likely that in a database of 1 million records, there would be areas of the database which would be much more active, at any given time, than others. At the other extreme, it is possible to imagine several users all wanting to update the same single record, but their timings, although close, are such that they never interfere with one another, i.e. one process releases the record just before the next one ends the seize block in which it requests to lock the record.

It is therefore very difficult to compare one concurrent update algorithm with another. The yardstick against which the EDAMS algorithm was measured was the straightforward first-come-first-served (FCFS) system. The results show that the EDAMS algorithm performs considerably better overall. An operational comparison between say the Chamberlin et al algorithm described

in Section 4.3.5 and EDAMS would be interesting, but the results would be difficult to evaluate. The overhead of the Chamberlin algorithm with its proliferation of small queues for individual records, is clearly higher than the EDAMS system. Furthermore, the Chamberlin algorithm necessitates the totally arbitrary favouring of a process in order to ensure its release, whereas the EDAMS Priority System automatically and logically guarantees that every process will be released within a reasonable period of time (threshold), while at the same time allowing more than one process to hold records simultaneously. On the other hand, it could be argued that the Chamberlin et al. algorithm might not involve as many re-evaluations of entire seize blocks. In this connection, however, it should be pointed out that EDAMS allows for "over-locking" based on realm for certain processes. Requests of the form

LOCK ALL RECORDS IN REALM R

require no access to the database in the seize block. All that is required is a quick scan through the list of currently locked records against the records in realm R. If any record appears on both lists, then the locking predicate fails.

In conclusion it is felt that the EDAMS Priority System does provide an efficient and simple algorithm for solving the problems of contention between users during concurrent update of the database.

16.5 Future work

Only the basic core of EDAMS has been implemented in

order to evaluate the operation of the concurrent update algorithm. Future work on EDAMS should therefore involve a full-scale implementation of the system with live data.

It would be interesting to evaluate, if possible, how efficiently the database map concept works in practice. The extent to which users benefit from the increased degree of flexibility offerred by EDAMS through the SLR and subschema sets, is also worthy of examination.

In order to assess more fully the operation of the concurrent update algorithm, it would be useful to replace it, in the full EDAMS implementation, by other solutions to the problem (e.g. Chamberlin et al, CODASYL) using the same database and the same set of user requests.

REFERENCES

1. CODASYL DBTG Report, British Computer Society, April 1971.

2. Schubert R.F., Basic concepts in Data Base Managment Systems, Datamation, July 1972, pp42-47.

3. Whitney K.M., Fourth generation data management systems, AFIPS, 1973, pp239-244.

4. Postley J.A., The MARK IV System, Datamation, Jan. 1968, pp28-30.

5. Martin J., Principles of data-base management, Prentice Hall, 1976.

6. Bryant J.H., Semple P. GIS and file management, Procs. ACM National Meeting, 1966, pp97-107.

7. Bleier R.E., Treating hierarchical data structures in the SDC time-shared data management system (TDMS), Procs. ACM National Meeting, 1967, pp41-49.

8. Childs D.L., An information algebra, CACM, Vol.5, No.4, pp190-204.

9. Childs D.L., Feasability of a set-theoretic data structure, A general structure based on a reconstituted definition of a relation, IFIP, 1968, Vol.1, pp420-430.

10. Codd E.F., A relational model of data for large shared data banks, CACM, Vol.13, No.6, June 1970, pp377-387.

11. Michaels A.S., Mittman B., Carlson C.R. A comparison of the Relational and CODASYL approaches to Data-Base Management, Computing Surveys, Vol.8, No.1, March 1976, pp125-151.

12. Fowling J.R., Controlling BEA Seat Reservations, The computer Bulletin, Vol.10, No.1, June 1966, pp48-50.

13. Stross C.C.M., Operation of a disc data base, The Computer Journal, Vol.15, No.4, pp290-297.

14. Emery J.C., An overview of Management Information Systems, Data Base, Vol.5, 1973, pp1-15.

15. Greenes R.A., Pappalardo A.N., Marble C.W., A system for clinical data management, AFIPS, FJCC, 1969, pp297-305.

16. Beggs S., Vallbona C., Spencer W.A., Jacobs F.M., Baker R.L., Evaluation of a system for on-line computer scheduling of patient care activities, Computers & Biomedical Research, Vol.4, 1971, pp634-654.

17. Simborg D.W., MacDonald L.K., Ward Information Management System - An Evaluation, Computers & Biomedical Research, Vol.5, 1975, pp484-497.

18. Abrams M.E., Bowden K.F., Chamberlin J., A computer-based general practice and health centre information system, Journal of the Royal College of General Practictioners, Vol.16, 1968, pp415-427.

19. Reekie D., Computers in the health service - fact or fiction?, Meeting of the Edinburgh Branch of the British Computer Society, Feb.6, 1974.

20. The London Hospital Computer System, A case study in the installation of a major real-time system, Procs. of conferences held on Nov. 27, 1973 and April 24, 1974.

21. Feature analysis of generalized DBMS, CODASYL, May 1971.

22. CODASYL DDLC Journal of Development, 1978, Available from Canadian Government.

23. IBM IMS/360 Version 2 General Information Manual, GH20-0765-3.

24. IBM IMS/360 Version 2 Application Programming Reference Manual, SH20-0912-3.

25. Knuth D.E., The Art of Computer Programming, Vol. 1, Fundamental Algorithms, Addison-Wesley, 1968.

26. Palmer I., Database Management, Scicon, 1973.

27. King P.F., Collmeyer A.J., Database Sharing - an efficient method for supporting concurrent processes, AFIPS, 1973, Vol.42, pp271-275.

28. Infotech State of the Art Report No. 15 on Data Base Management, Infotech Information Ltd., 1973.

29. DMS 1100 Schema Definition Reference Manual, UNIVAC.

30. Kroenke D., Database Processing Fundamentals, Modeling, Applications, Science Research Associates Inc., 1977.

31. Everest G.C., Concurrent update control and database integrity in Data Base Management, ed. J.W.Klimbie & K.L.Koffeman, North Holland/American Elsevier, 1974.

32. Chamberlin D.D., Boyce R.F., Traiger C.L., A deadlock-free scheme for resource sharing in a data-base environment, Proc. IFIP Congress 1974, pp340-343.

33. IDR 3042, PRIME COMPUTER User's Guide for the Database Administrator, July 1977.

34. CODASYL DBTG Report 1969, British Computer Society.

35. CODASYL COBOL Journal of Development 1976, British Computer Society.

36. CODASYL FORTRAN DML Journal of Development 1978, available from the Canadian Government.

37. BCS/CODASYL DDLC DBAWG Report, Jan. 1975, British Computer Society.

38. BCS October 1971 Conference on April 1971 Report, British Computer Society.

39. Kilburn T., Edwards D.B.G., Lanigan M.J., Sumner F.H., One-level storage system, IRE Transactions EC-11, No.2, April 1962.

40. Joseph M., An analysis of paging and program behaviour, Computer Journal, Vol. 13, No. 1, Feb. 1970, pp48-54.

41. Sherman S.W., Brice R.S., Performance of a Database Manager in a Virtual Memory System, ACM Trans. on Database Systems, Vol. 1, No. 4, Dec. 76, pp317-343.

42. Brice R.S., Sherman S.W., An extension of the performance of a database manager in a virtual memory system using partially locked virtual buffers, ACM Trans. on Database Systems, Vol.2, No. 2, June 77, pp196-207.

43. Lang T., Wood C., Fernandez I.B., Database buffer paging in virtual storage systems, ACM Trans. on Database Systems, Vol. 2, No. 4, Dec. 77, pp339-351.

44. Tuel W.G., An analysis of buffer paging in virtual storage systems, Research Report RJ 1421, IBM Research Lab., San Jose, California, July 1974.

45. Casey R.G., Osman I.H., Generalized page replacement algorithms in a relational data base, Proc. ACM-SIGFIDET Workshop on Data Description, Access & Control, May 1974, ACM, pp104-124.

46. Denning P.J., Virtual memory, Computing Surveys, Vol. 2, No. 3, Sept. 1970, pp153-189.

47. Whitfield H., Wight A.S., EMAS - The Edinburgh Multi-Access System, The Computer Journal, Vol. 16, No. 4, Nov. 1973, pp331-346.

48. Stacey G.M., The role of virtual memory in the handling of application files, Information Processing Letters, Vol. 1, 1971, pp1-3.

49. Senko M.E., Altman E.B., DIAM NOTE 1, A Framework mode for implementing a record storing facility, Research Report RJ 1365, IBM Research Lab., San Jose, California, March 1974.

50. Millard G.E., Rees D.J., Whitfield H. The standard EMAS subsystem, The Computer Journal, Vol. 18, No. 3, Aug. 1975, pp213-219.

51. Rees D.J., The EMAS Director, The Computer Journal, Vol. 18, No. 2, May 1975, pp122-130.

52. Hensen P.B., A program methodology for operating system design, IFIP, 1974, pp394-397.

53. Hensen P.B., The programming language concurrent Pascal, IEEE Trans. on Software Engineering, Vol. SE-1, No. 2, June 1975, pp199-207.

54. Casey L.M., Computer Structures for distributed systems, Ph.D. Thesis, Dept. of Computer Science, University of Edinburgh.

55. Engles R.W., A tutorial on data-base organization, IBM Technical Report, TR-00.2004, March 20, 1970.

56. Grimson J.B., Stacey G.M., A performance study of some directory structures for large files, Information Storage & Retrieval, Vol. 10, pp357-364.

57. Tsichritzis D.C., Lochovsky F.H., Database Management Systems, Academic Press (Computer Science & Applied Maths Series), 1977.

58. Atkinson M., Private communication, 1978.

59. Yarwood K., Private communication, 1979.