ACKNOWLEDGEMENTS

I would like to express my appreciation to my supervisor, Dr. B.C. Desai, who picked up the guidance of this research midstream and in the writing of the thesis. I am also indebted to Dr. J.J. Juergens, my former supervisor, under whom this work began.

I would like to thank Drs. J.W. Atwood and C.Y. Suen for their assistance during my research. I would also like to express my gratitude to Miss Phoebe Pang and Miss Selina Wong for their patience and love in typing up part of my thesis.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

AFT	active file table
CDD	command device directory
CFSS	DML command free space stack
CLT	currency location table
CMV	cumulative modified value array
CODASYL	programming language comittee on Data Systems Languages
CPU	central processing unit
CRU	current record of the run unit
DBA	data base administrator
DBMS	data base management system
DBMSC	DBMS control module
DB-STATUS	data base status register
DBTG	Data Base Task Group
DDL	data definition language
DDLC	Data Description Language Committee
DFD	device file directory
DHFD	device head file directory
DIAM	data independence access model
DML	data manipulation language
DSDL	data storage description language
dtcs	date and time a DML command starts
dtm	date and time modified
dtrs	date and time run unit starts
ELB	encoding level buffer
HISR	head index storage record

I/O	input/output
ISR	index storage record
LFS	logical file system
LLB	location level buffer
LRBC	logical rollback control module
MCDBMS	multilevel CODASYL DBMS
MV	modified value array
OS	operating system
PASR	pointer array storage record
PCLT	physical currency location table
PFS	physical file system
PRBC	physical rollback control module
RCT	record currency table
RDD	run unit device directory
RFSS	run unit free space stack
RTD	record type directory
SCT	set currency table
SLB	structural level buffer
SLR	structural level record
SPARC	Standard Planning and Requirements Committee
STC	start and termination modules
UWA	user working area

LIST OF FIGURES

LIST OF TABLES

Chapter I
INTRODUCTION

Data base technology has advanced by leaps and bounds in the last decade. A significant number of research papers have been published on data bases. The industrial world is not slow in their implementation. However, one area in data base technology is still lagging behind. This area is the recovery of data bases.

Recovery of data bases is a difficult subject for research. The unpredictable nature of errors contributes to the difficulty. Rapid technological advances in other disciplines of data bases leave recovery far behind the frontier of data base research. Any research to push recovery a step further is still way beyond the current DBMS technologies and may not be applicable to current industrial DBMS. This alone would discourage researchers to venture into this field. This thesis is intended to improve some of the recovery techniques as applied to CODASYL data bases. This may serve as an invitation for further research on the topic.

1.1 DIFFERENCES BETWEEN FILE AND DATA BASE RECOVERY

Each file is an isolated collection of data. There is no relation between files in general. If a file is damaged due to a failure, reloading it from a dump of an earlier version would not affect the other files in a filing system.

However, the case is much more complicated for data bases. Assuming that the data base is stored in the form of files, the data base files are interrelated. Damage to a data base file A may require reloading of an earlier version of the file. However, a record in file A and another record in file B may be related to each other. Such a relation may not have been established in the earlier version of file A but is indicated by the present version of file B. Inconsistency arises as a result of replacing file A by its earlier version alone.

Therefore the data base is seen as an integrated whole. This is especially true for the CODASYL type of network data bases where a relation can exist between any two entities. If a portion of the data base requires rollback due to damage, the entire data base has to be restored to a certain consistent point in time. A checkpoint for a data base serves as a point in time for rollback recovery. Unfortunately such recovery procedure takes intolerably long time for most users and applications.

A busy DBMS with concurrent run units or application programs amplifies the difficulty in recovery. Data modified by a run unit may be used by other run units as a basis for further modification of other parts of the data base. Run units are no longer isolated units. They can be considered as interacting with each other via the data base. The well-known domino efffect may use up all the recovery points of the interacting run units [Rand75]. If a DBMS is operating around the clock with concurrent run units, it may be hard to establish a checkpoint common to all the run units.

1.2 CODASYL DATA BASE RECOVERY

According to Date[1], 'a transaction is defined as a unit of work that is atomic from the point of view of the enterprise'. A bank clerk transferring an amount X from account A to account B may enter the command

```
    TRANSFER  X=100.00, A=123456, B=654321.
```

or the commands

```
    WITHDRAW  X=100.00, A=123456.
    DEPOSIT   X=100.00, B=654321.
```

The first option is regarded as an atomic transaction while the second one is regarded as two distinct transactions. If the sum has been subtracted from account A but not added to account B because of an error during execution, the data base is regarded as inconsistent for the former option but is otherwise consistent for the latter option.

The above analysis shows the similarity of the first option with a CODASYL Data Manipulation Language (DML) command. The first option involves modification of more than one account. Either the operation is to be performed

1 Date, C.J. An Introduction to Database Systems, (2nd Ed.). Addison Wesley, USA, 1977, p.400.

on both accounts or none of them should be affected. By the same token a DML command may involve modification of more than one record. Either the records involved are modified or none of them should be affected.

The smallest unit of recovery is inherent in the DML specification. Rollback of a DML command requires undoing of all the work done prior to the error condition throughout the execution of the DML command. The recovery must adhere to this requirement.

1.3 THE SCOPE OF THIS THESIS

Basically two main types of damage to the data base could occur. The first type is physical damage, for example, a disk head crash may cause a number of tracks to be inaccessible. The second is the erronous modification of the data base due to bugs in the user program, the DBMS or the operating system (OS).

This thesis assumes that general recovery techniques are available. These techniques are described in chapter two. Within the vast possibilities of damage and repair, this thesis aims at providing DML command rollback and run unit rollback. If a DML command is unable to proceed after changing part of the data base, a command rollback is needed. If a run unit is unable to continue even after a command rollback, a run unit rollback is required. This thesis also assumes a large computer system with a general purpose operating system. In order to avoid tackling the domino effect, a single user DBMS is assumed.

The goal of this thesis is twofold. First, a multilevel type CODASYL DBMS is to be designed. Second, a rollback recovery system is to be constructed to improve the recovery time and reliability of the two rollback options. The rollback recovery system is produced by a combination of the careful replacement and the differential file recovery techniques.

Chapter 2 describes the background material in two main parts, namely, a discussion on CODASYL DBMS and the recovery techniques available to data bases. A general overview of the designed model is given in chapter 3. The multilevel model is described in terms of the functions performed by its levels. Based on the functional description of the model in the previous chapter, chapter 4 gives a particular implementational design of the model. The data structures and the modules required are listed. A rollback recovery subsystem for fast DML command rollback and run unit rollback is described in chapter 5. Chapter 6 provides a functional description of the modules of the rollback recovery subsystem. Chapter 7 gives the conclusion. Appendix A contains a collection of tables which describes the modules implementing the proposed multilevel CODASYL DBMS.

Chapter II
BACKGROUND

This chapter first traces the evolution of CODASYL DBMS. The characteristics of such DBMS are described briefly, followed by a survey of the implementation of different industrial systems. After which the recovery techniques available to DBMS, including techniques borrowed from file recovery and operating system recovery, are summarized. Finally, the recovery techniques available in some well-known CODASYL DBMS are listed.

2.1 THE DEVELOPMENT OF CODASYL DBMS

This section traces the development of DBMS with particular emphasis on CODASYL DBMS. The evolution of DBMS may be roughly divided into three overlapping stages [Fry76]. The first stage is the early developments prior to 1964. The second stage is the evolution of families during the period 1964 to 1968. The third stage is the vendor/CODASYL developments from 1968 to the present.

The drive for the development of DBMS in the first stage comes mainly from users in the government, especially in the military and intelligence areas. These isolated prototype developments provided the starting points for several significant DBMS families.

In the second stage isolated developments faded away and full-scale DBMS families appeared. A family may cross organizational boundaries because of the communication of ideas. The most significant families are the MITRE/Auerbach family, the Postley/MARK IV family, the Bachman/IDS family and the Formatted File/GIS family. The Integrated Data Store (IDS) was developed by Bachman and his colleagues at the General Electric Company in 1964. It combined random access storage technology with highly procedural languages to provide a network data model. It has evolved into a new version IDS-II marketted by Honeywell in 1975. In 1966 Dodd and his colleagues at General Motors Research developed APL (Associated PL/I). This development is similar to IDS. It provided data management functions for a computer-aided environment. It also separates logical relationships of owner and member groups from their physical implementation.

The third stage moves from family-oriented activities to proprietary vendor development. Advances are kept as industrial secret. Literatures often confine themselves with the mathemetical and theoretical aspects of DBMS only. Three major families appear in this period: the CODASYL/DBTG family for network data bases, the IMS family for hierarchial data bases and the Inverted File family. The Inverted File family is represented by System 2000 of MRI Corporation.

Having IDS and APL as its starting point, the Programming Language Committee on Data Systems Languages (CODASYL) started a task group to extend COBOL to handle data bases. The first proposal was made by the Data Base Tsak Group (DBTG) in 1969. A subsequent report with improvements was published in 1971 [DBTG71]. This report constitutes a landmark in the development of data base technology [Tayl76]. Later, a Data Description Language Committee (DDLC) was formed to deal extensively with the data description. The DBTG evolved into the data Base Language Task Group (DBLTG) to deal only with COBOL extensions. The DDLC published a report in 1978 [CODA78]. A Data Storage Description Language (DSDL) is also included in the report. A number of authors [Mano78], [OCon78], [Stac78], [Toze78] welcome the DSDL with great appreciation. The development of the CODASYL family is shown in Fig. 2.1. The leftmost margin of numbers denotes the year. Next to this margin as well as the rightmost margin are the company names which marketted the models shown in the middle of the figure. However, when the name 'CODASYL' appears, the middle part of the figure represents CODASYL specifications or task groups. All entries on the same row belongs to the same year.

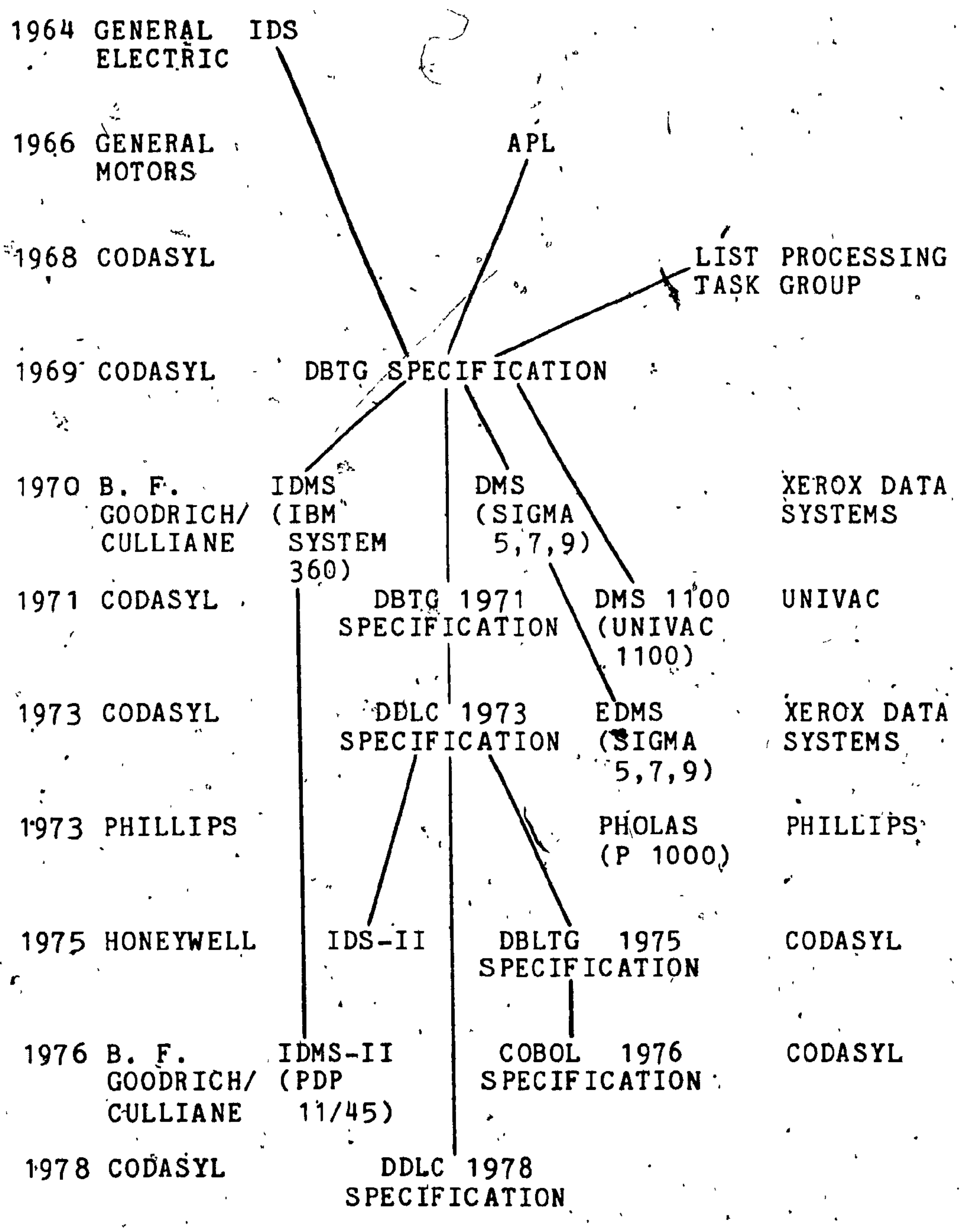


Fig. 2.1 The CODASYL Family

2.2 THE CHARACTERISTICS OF CODASYL DBMS

A number of papers [Tayl76] and textbooks [Date77], [Kroe77], [Mart77], [Olle78], [Spro76], [Tsic77], [Ullm80] discuss the characteristics of the CODASYL proposal. The majority of authors base their description on the DBTG's 1971 report [DBTG71]. Among them Olle's textbook provides a detailed tutorial on the subject. The DDLC 1978 specification [CODA78] is the most recent publication for the DDL proposal. Many concepts in the 1971 proposal are retained in this specification. Manola [Mano78] gives a detailed discussion on the transition of the 1971 report to the 1978 DDL proposal.

A CODASYL DBMS contains five main components as follows:

1. Schema Data Description Language (DDL),
2. Subschema Data Description Language,
3. Data Manipulation Language (DML),
4. Data Base Control System (DBCS),
5. Storage Schema Data Description Language (DSDL), formerly known as Device Media /control Language (DMCL).

The schema DDL is used to describe a data base independent of the programming languages used for the various applications to access the data base. Part or all of the data base which is of interest to a user is available to him via the subschema DDL. There is only one schema for a data base but there can be many subschemas for different users accessing different parts of the data base. The DML is the language employed by a user to access and update the data base. The storage schema DSDL is a storage device and operating system independent language to describe a storage environment for the data base and maps the schema to the storage environment.

Apart from the familiar COBOL terms of data item, data aggregate and record, the schema DDL also contains the concept of set. A set represents a named 1 to N link among record types. A set type consists of one owner record type and one or more member record types. A set can be ordered under different options. In other words, the member records within a given set occurrence have predetermined method of insertion into the set. Each member record of a set type has a storage class and a removal class for insertion or removal of a member record into or from a set occurrence. The set selection criteria determines which occurrence of the owner record type is to be the owner of a member record. Each run unit has associated with it a set of currency indicators for the current set type and the current record type. These indicators play an important role in DML execution, set selection and set insertion.

The data base is subdivided into smaller units called areas of realms. However, the CODASYL proposal does not specify the way to subdivide the data base.

LEVEL	COBOL JOD
Realm	READY, FINISH
Set	ORDER
Record only	ERASE, FIND, STORE
Record and item	GET, MODIFY
Linkage	CONNECT, DISCONNECT
Currency Indicators	ACCEPT
Conditional	IF
Declarative	USE
Concurrency	FREE, KEEP, REMONITOR

Fig. 2.2 COBOL JOD DML Statements

```
                                            +-------------+
                                           /| Subschema 1 |
                                          / +-------------+
+----------------+     +--------+ /   .          .
| Storage Schema |-->| Schema |    .          .
+----------------+     +--------+ \   .          .
                                          \
                                           \ +-------------+
                                            \| Subschema n |
                                             +-------------+
```

Fig. 2.3 The 1978 CODASYL Data Description Framework

Fig. 2.2 shows the DML statements of COBOL 1976. These statements manipulate the data base records. The User Working Area (UWA) is a buffer for loading or unloading data for the run unit. The 1978 CODASYL data description framework is shown in Fig. 2.3.

2.3 THE IMPLEMENTATION OF CODASYL DBMS

Due to industrial secrecy, there are few papers describing the physical implementation of a data base based on CODASYL. Appendix A shows the systems implementing CODASYL type data bases. Systems that implement CODASYL DBMS are based on the DBTG 1971 report. The DBTG 1971 DDL and DML languages are used in the following description. The implementation of CODASYL proposals will be discussed under the following sections: relation of the DBMS with the application programs and the OS, the data base, the schema and the user working area.

2.3.1 Relation of the DBMS with Application Programs and the Operating System

Schenk [Sche74] gives a comprehensive view on the relation of DBMS with application programs and the OS. He suggested several possibilities as discussed below.

Case one. Here the application to which the DBMS is linked can be an asychronous process (multi-tasking). However the host language has to support the multi-tasking and the data communication facilities. This is supported by the multiprogramming version of PHOLAS [Douq75], in which a copy of the system is link-edited to each user program requiring it. Fig. 2.4 shows the layout. Only the EXCLUSIVE mode is allowed for the READY statement. IDS-I [Bibb75], [John75] also needs a separate non-sharable set of IDS routines to be loaded with each program accessing the data base. The high core overhead is compensated by the powerful CDC 6000 with fast roll-in and roll-out feature.

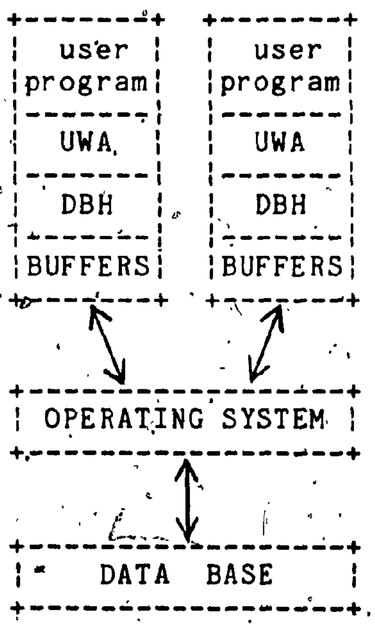


Fig. 2.4 Multiprogramming linked version of PHOLAS

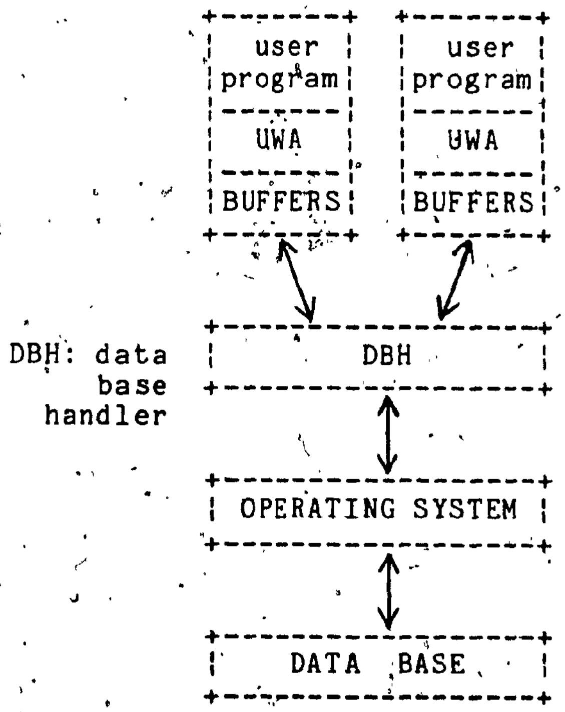


Fig. 2.5 Multiprogramming linked version of PHOLAS with shared code

Fig. 2.5 shows an improved version of the first case. The multiprogramming OS supports the shared code of the DBMS routines. A number of applications may run simultaneously, each of them having been linked to their own DBMS logically. Actually the DBMS is present only once in the core. The user programs are unaware of each other so that concurrency problems do arise. Two run units may update the same data base record but the OS is unaware of the situation. PHOLAS also supports this case. The data base handler (DBH) is a shared segment. The OS provides the segmentation.

Case 2. In this method the DBMS acts as an independent job under the OS. The concurrency problems can be solved in this case by the OS which provides locks for data base records. Unfortunately, the transfer of control through the OS and the amount of overhead involved is quite significant. The control can be maintained in three levels:

1) the OS controls the programs running under it. The DBMS acts as an application program as viewed from the OS. This choice encounters communication problems. No existing system is known to implement its DBMS in this way.

2) The DBMS controls the application programs running under it. This is supported by the full central data base handler version of PHOLAS. Several user programs interface with the DBH. The DBH has its own buffer pools and accesses the data base via the OS, as shown in Fig. 2.6. Apart from PHOLAS, the central version option of IDMS [Hack75], [Peck75] also supports this option. The centralized access monitor program (CAMP) threads requests to IDMS and regulates the use of the data base resources in the multiuser environment.

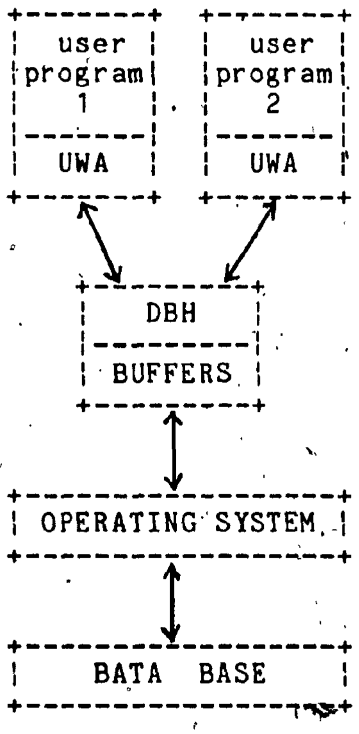


Fig. 2.6 Multiprogramming independent version of PHOLAS

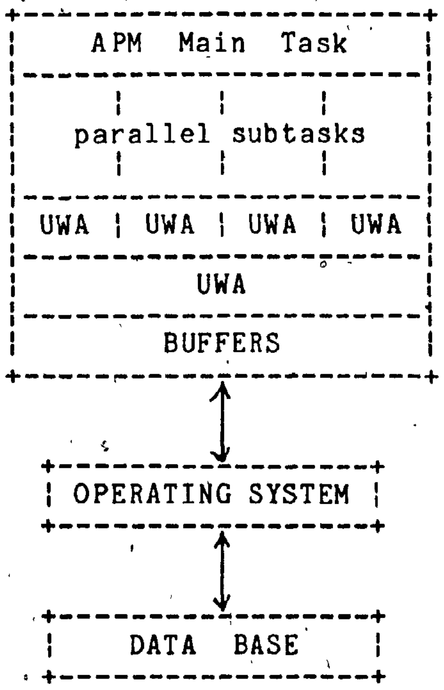


Fig. 2.7 Multitasking linked version of PHOLAS

3) The application controls tasks if the asynchronous processing technique is employed. The PHOLAS multitasking linked version supports this approach. This multitasking is not implemented at the operating system level but at the user program level. A special package known as the asynchronous process monitor (APM) is linked upon request by the compiler to the program. This will perform subtasking within the program, as shown in Fig. 2.7. These subtasks within a single module are invisible to the OS. Because the data base handler is linked to the program, the COBOL compiler automatically provides a UWA for each subtask. Each subtask can be considered as a run unit in the CODASYL

proposals.

Case 3. Here, the DBMS is an integrated part of the OS. Up to the present, no OS in use can achieve this stage because of the complexity of the problem involved.

Case 4. In this scheme, the DBMS is a back-end computer. This concept is explained in deatil by Canaday et al [Cana74], and is implemented as XDMS which stands for Experimental Data Management system. The DBMS is implemented on a separate machine which has exclusive access to the data base. The back-end computer, as opposed to the front-end computer, serves as an interface between the host computer and its data base. Canaday's paper also discusses the pros and cons of this approach.

2.3.2 The Data Base

Within the data base three constructs can be distinguished, namely realm (area in the DBTG 1971 language), record and set.

2.3.2.1 Realm or area

Realms are subdivisions of the data base. They are required to be logically and phusically non-overlapping compartments. Areas are usually divided into a number of physical blocks or pages. In IDS-I [John75] a data base is divided into subfiles. These subfiles are physical subdivisions of storage devices.

IDS-I employs a paging technique for all physical I/O and logical maintenance. The IDS program executing the STORE command does not cause the record (that is, page) to be written back to the data base. A 'must write' switch is set in the page. Each page of the data base contains a page header as shown in Fig. 2.8. A utility maps the storage devices into sequentially numbered pages.

```
     +---------------------------------------------------+
     | PAGE  |RECORD| CALC |EMPTY| MUST  |RECORD|
 +-->|       |      |CHAIN |SPACE| WRITE | I.D. |
 |   |NUMBER| TYPE  | NEXT |COUNT|SWITCH | FLAGS|
 |   +---------------------------------------------------+
 |
 |
 |              +------------------------------+
 +--------------|////////|         |     |     |
                |------------------------------|
                |    |        |            |   |
                |------------------------------|
                |            |         |       |
                |-----------------             |
                |          AVAILABLE           |
                |            SPACE             |
                +------------------------------+
```

Fig. 2.8 IDS-I page header record

The DMS-1100 [MacD75], [Robi75] implemented on UNIVAC 1100 divides the data base into several named areas. Each area contains a number of pages. Different areas can have different page sizes and different number of pages. FORDATA [Mack77] implemented on CDC 3600 and GPLAN/DMS [Hase74] also use a paging scheme. FORDATA contains map pages to account for the availability of data pages and whether a page has been referenced previously.

2.3.2.2 Record

The CODASYL DBTG gives a precise specification for a record. The only problem is with the data base key. Reorganization renders it impossible to assign the device address as the data base key. The data base key is usually implemented in three ways. The first method divides the data base key into two main parts. The first part contains the record type identification. The second part contains a number sequentially assigned by the DBMS per record type. A transformation table maps the value into the physical address.

The second method used in IDS-I assigns a line number to every record stored within a page. This line number is a relative address with respect to the page. A page number together with a line number uniquely identifies each storage record within the data base. The combined values constitutes a data base key. GPLAN/DMS also employs a similar scheme which has a page number and an offset in page.

The third method is used by PHOLAS. The data base key has both a logical and a physical part. The physical part always enables the file contaning the record to be located. The physical address can be found via the data base key translation table using the logical part. Apart from the data base key, a record can also be located by a search key in SIBAS [Lie75]. The search keys are index keys. They can be elementary or group data items.

2.3.2.3 Set

Schenk suggests a better definition of a set as 'an access path through a number of related records' [Sche74]. From the point of view of physical implementation, this definition is quite appropriate.

Douque [Douq75] presents four basic primitives relating to the sequence and level of indirection as shown in Fig. 2.9 from D. Severance's thesis. The sequence can be physical sequential (PHS) or pointer sequential (PTS). The user data can be data direct (DD) or data indirect (DI). The combinations of these make up techniques for data storage in sets supported by PHOLAS.

Bachman [Bach74] listed nine basic forms of set implementation based on fourteen different set manipulation functions. They are single level record array, multiple level record array, pointer array, boolean array (prohibit ordering of set members), packed boolean array, list, chain or ring, binary tree and phantom. Bachman also gives a table of comparison for the performance characteristics of these nine basic forms. The nine basic forms can be combined to form further structures for set implementation.

Each record in GPLAN/DMS contains three pointer fields, 'previous', 'owner' and 'next' for member records, and 'first', 'last', 'number' (of members) for owner records.

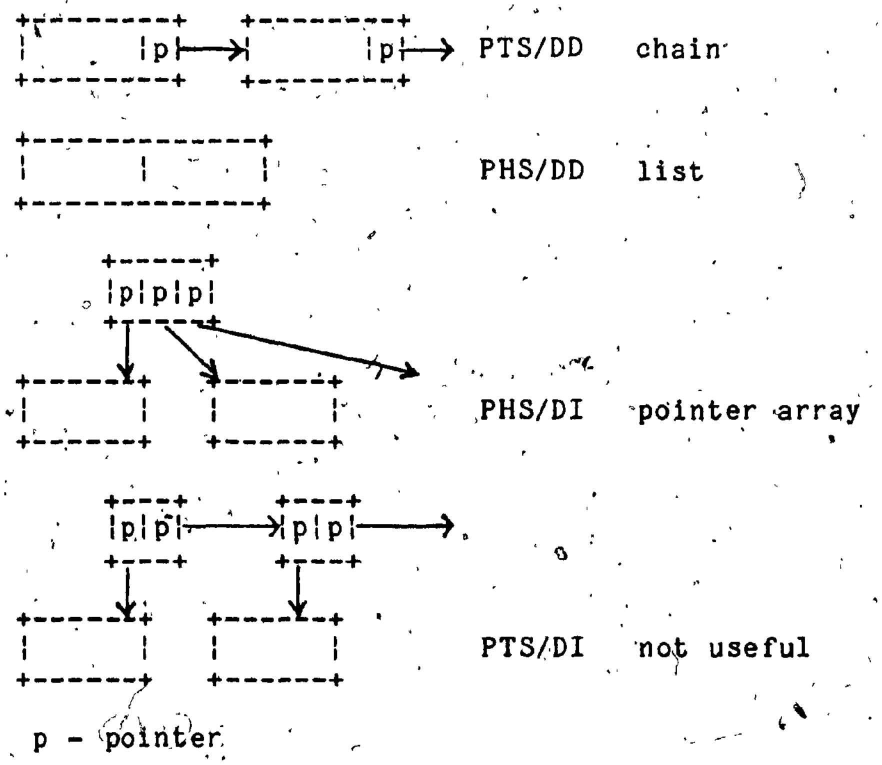


Fig. 2.9 Storage structure of PHOLAS

The pointers are all data base keys. The IDS-I chain is only a one way chain. User of IDS-I and DMS 1100 can specify logically related records to cluster in the same or adjoining pages.

Taylor [Tayl75] investigated three methods of implementing dynamic pointer arrays.

2.3.3 The Schema

The information describing user, data and storage structures of the data base has to be interrogated frequently at run time. The efficiency of schema and subschema storage is very important. The schema of GPLAN/DMS contains two tables generated by the DDL analyzer.

The record table contains a group of concise record descriptions. The system also incorporates the currency of the record types into this table. These currencies are the only value that changes in this table during execution time. The set table contains a group of set type descriptions. The current owner pointer and the current member pointer is present in the table for each set type. The schema is stored in core during exectuion time.

In FORDATA the schema is represented as a collection of records stored in area zero. It is stored in the same way as the rest of the data base. The schema is organized in the form of sets and records. However, since the schema is a collection of records in an area, it may be swapped to the backing store. It also incurs overhead bacause the schema contains currency indicators which have to be updated often.

2.3.4 The User Working Area (UWA)

The UWA is a loading and unloading zone. This is the place where data is transformed from the DBMS as a result of program call. Data to be delivered to the DBMS is also placed in the region. The UWA is set up for the subschema. Locations in the UWA are assigned to every data item that is included in the subschema.

Scheck suggests two methods of implementation of the UWA. The static approach determines the layout of the UWA at subschema compile time. The disadvantage of this approach is the possible waste of core space because not all record areas are always required. Moreover, a change in the subschema will require recompilation and relinking.

The dynamic approach determines the layout of the UWA at run time. The address for a record to be delivered or obtained by the DBMS is one of the parameters passed to the DBMS. The space allocation for the UWA is the responsibility of the DBMS from the start to the end of a DML command.

2.4 DATA BASE RECOVERY

Recovery techniques are used to restore a system to a usable state. Recovery data are redundant data maintained to make recovery possible. A recovery technique structures, organizes and manipulates data structures and recovery data to make recovery possible. A failure is an event which causes the system not to perform according to the specifications. Failures can be caused by hardware faults, software faults or human errors. The aim of recovery is to restore the data base into a consistent state. The consistent state consists of no spurious data, although some information may have been lost.

2.4.1 Theory

Bjork and Davies have investigated the theoretical aspects of data base recovery. They have worked out the concept of commitment and sphere of control [Bjor73], [Davi73]. Commitment refers to the guarantee that the information previously provided will be available again, unmodified. If a page is to be updated, committing the page to modification requires access to the original page if there is a failure during update. A sphere of control is a bound within which a process can have exclusive use of information. It is similar to the critical region bounded by the P and V operations of a semaphore in an operating system. Hierarchial spheres of control which result in hierarchial recovery are also discussed. The characteristics of classical batch, multi-thread and interactive processing pertinent to recovery and integrity are also surveyed.

2.4.2 File Recovery

A number of authors [Dave77], [EDP76], [Info77], [Verh78] have provided surveys of the literature on recovery of data bases. Some of the techniques applied in file recovery can be used in the recovery of data bases. Drake [Drak71] and Gibbons [Gibb76] give a comprehensive discussion of file recovery techniques. Oppenheimer and Clancy [Oppe68] survey the technique for file protection and recovery from hardware failures in a multiaccess, multiprogramming, single processor system. Smith and Holden [Smit72] investigated several models to compare I/O and recovery overhead in file recovery. Johnston [John76] five case studies. The Infotech report [Info76] reviews the performance and reliability trade-offs for various recovery techniques used in a number of on-line systems.

2.4.3 Salvation Program

A salvation program is used after a crash to restore the data base to some consistent state. It sacrifices deletion of files or data for consistency. The HIVE system use salvation program for recovery. Lockemann and Knutsen [Lock68] suggest a salvation program which provides recovery of the contents of a volume on an IBM 2311 disk drive in less than one minute. Daley and Neumann [Dale65] suggest an on-line storage procedure for mild failures. The procedure reads through directories and storage assignment tables to delete all erronous files and correct inconsistent information by maintaining two directory entries pointing to every file. Fraser [Fras69] mentions a start-up procedure for the Cambridge filing system. The procedure is used at the start of the day and after an error to make a thorough consistency check of the file directories and other administrative data. Erronous file and file directories are deleted. Once consistency has been achieved, the necessary recovery procedures are invoked. The Multics system [Ster74] has a salvaging program which, apart from the above mentioned functions, also notifies the users of the situation.

2.4.4 Dumping

Incremental dumping involves the copying of updated files onto archival storage after a job has finished or at regular intervals. It creates checkpoints for updated files. Backup copies of files can be restored after a crash to bring the files to their previous consistent state. A secondary dump collects all files and directories that have been incrementally dumped later than some specific time in the past. Only the latest version of the incremental dumps appear in a secondary dump. A secondary dump which supersedes all previous incremental dumps is called a complete secondary dump.

The primary system of the Cambridge filing system makes incremental dumps on magnetic tapes every twenty minutes. The secondary system maintains a compact record of all the files with a delay of one week. Daley and Neumann also investigated the time used for dumping.

Stern's thesis [Ster74] proposes an efficient scheme for incremental and secondary dumping for the Multics system with a hierarchial file directory. His dumper can run as a multiprogramming job to dump files concurrent with normal processing because a shadow copy of a file can be created to make sure that the incremental dumper dumps a consistent version of the file. The dumper can avoid unnecessary searching in the file directory tree because of the use of time stamps and a 'modified below' bit for all ancestors of a modified file. The reloading process can take place concurrently with normal processing after the file directory has been restored. Only users accessing files that requires reloading are delayed.

System R [Verh78] has a current page table and a segment page table. Within the page tables there are shadow bits, cumulative shadow bits and long term shadow bits. These bits are essential for incremental and secondary dumping. Gibbons [Gibb76] suggests 'grandfather', 'father' and 'son' versions of file dumping.

2.4.5 Journal

There are three main types of journal. After-image journals or new copy journals are copies of the section of the file after it has been modified by a transaction. Before-image journals or old copy journals are copies of the section of the file before it is modified by a transaction. A transaction journal is equivalent to an audit trail. An audit trail records sequences of actions performed on files. Audit trails can be used for rollback and rollforward. It can also be used to certify system integrity [Bjor75]. Gibbons [Gigg76] has investigated journals in considerable detail. He also lists the data which is to be recorded in the journals. He combines different types of journals for recovery purposes. Wiederhold [Wied77] suggests using activity thread as a transaction log for a message. Wilkes [Wilk72] even suggests recording keyboard strokes.

2.4.6 Careful Replacement

Careful replacement avoids updating data 'in place'. The update is performed on a copy of a component which replaces the original only if the update is successful. The copy is kept until the replacement is made successfully. Two instances of the same data structure exist only during update. This technique is different from differential file [Seve76] which accumulates update requests for unaltered original. The CMIC system [Gior76] employs the careful replacement technique. This approach assumes a directory tree structure in which new copies are made from leaf to root for replacement. Such an updating sequence is known as leaf first rule.

2.4.7 Recovery Block

Recovery block is a concept borrowed from the recovery of operating systems [Horn74]. An acceptance test is executed on exit from the primary block to confirm acceptable performance. If the primary block is accepted, the program control is transferred to the next recovery block. Otherwise a recursive cache machine invokes an alternate block with the initial data values restored. On exit from the alternate, the acceptance test is again performed. This may result in further invocation of alternate blocks if the result is still undesirable. The recovery block can be considered as an implementation of the sphere of control concept. Verhofstad [Verh77] finds mechanisms to implement the recovery block scheme for a filing system on a B1700 computer. Randell [Rand75] extends the recovery block concept to interacting processes and multilevel systems.

The system error recovery policy of operating systems [Denn76] can be applied to the design of a DBMS. Processes within a level of the multilevel system implement new operations for the level above. An error recovery routine exists within each level. If level i discovers an error and the error has propagated down to level i-1, the recovery routine of level i calls the recovery routine of level i-1 to put the abstract machine of level i-1 into a consistent state. The recovery routine of level i then places the data in its level into a consistent state. It reports to its caller in the level above the degree of success in repairing the error. The system error recovery policy may be considered as a particular implementation of the hierarchial sphere of control or of nested recovery blocks.

Parnas [Parn72c] suggests the use of software traps when an error is detected. The error is reflected upwards and the control is passed to the level where the error originates. This level is presumably the best place to correct the error. Sugar [Suga76] has designed an error handling feature for a disk I/O system based on this system error recovery concept.

2.4.8 Reverse Execution

Maryanski [Mary77] proposed a rollback and recovery scheme for distributed DBMS taking the CODASYL proposal as an example. He suggests collecting recovery data in such a format as to enable the execution of reverse DML statements to rollback the operations performed. For example, a CONNECT statement is performed for a previously executed DISCONNECT statement.

2.4.9 Industrial Recovery Systems

DMS 1100 [Foss74], [Lava73] provides before look, quick before look and after look for a page. The before look and after look store pages in a journal while the quick before look stores a page in the random access storage device. The DBMS provides quick recovery for simple errors and long recovery which needs reestablishment of the data base from a previous dump. Selective recovery is provided so that the users accessing the undamaged areas can access their data along with the necessary recovery action.

FORDATA maintains a transaction journal and a before-image journal for recovery. IDMS has backup facilities for system failure. SIBAS has logging and rollback facilities for recovery as well as consistency check for errors. IDS-II contains quick rollback and rollforward rebuild recovery capabilities.

Limited information is available concerning the implementation of recovery strategies in industrial DBMSs. Some recovery strategy may depend on the structure of the DBMS. Therefore, a description of the DBMS in chapters 3 and 4 preceeds the discussion of the rollback recovery system in chapters 5 and 6.

Chapter III
GENERAL DBMS MODEL DESIGN

This chapter describes a model to implement a DBMS with the specifications proposed by CODASYL. It also examines the design goals of the model. The approach in the design of the model is discussed, taking into account the assumptions involved. A general overview of a multilevel CODASYL DBMS (MCDBMS) model is presented together with some general design considerations, including error handling features. Finally, a description of the file system on which the MCDBMS resides is given.

3.1 BACKGROUND

The design goals of a CODASYL DBMS model are fourfold. First, to propose a general model which can have the freedom to allow different implementational strategies of the functional specification of the model. Second, the model should be sufficiently specific to enable design and specification of recovery policies based on it. Third, the model should allow further expansion of the DBMS to adopt new features. Data independence is the fourth design goal for the model. The first two aims would result in a compromise leading to a functional design in the present chapter. The next chapter contains a specific design of the model described in this chapter.

The proposal by the Data Base Task Group [DBTG71] for schema data definition language (DDL) lacks data independence and has been severely critized at this point. The proposal in 1978 [CODA78] includes a data storage description language (DSDL) along with the schema DDL. This serves as a significant step towards data independence. Toh et al [Toh77] distinguishes further logical data independence which means that application programs are stable from changes of data structure such as data items, data attributes, data arrangement and data relationshp representation. Physical data independence means that both application programs and data structures are free from changing storage structures such as data placement strategies and index construction methods. The proposed model is directed to achieve these two types of data independence.

The relationship between the DBMS and the operating system has been discussed in section 2.3.1. Two basic approaches to DBMS design are possible, the first being to design an operating system which facilitates handling of data base management requests. The other approach would be to design a DBMS to interface with an existing operating system in an installation. The pros and cons of both approaches have been studied by Bayer et al [Baye78]. The latter approach is taken for several reasons. First of all, an operating system is already a large and complicated piece of software. Adding data base handling facilities into it would make the design and implementation difficult and error prone. Secondly the software for such a system would be more expensive than merely adding a DBMS on top of an operating system. Thirdly, the problem of incompatibility comes in when the system is switched to adapt data base processing. An individual installation may have the OS package modified to provide special features to its non-database users. Changing to a new software of DBMS and OS integrated together as a new package would require modification of the software again to produce a compatible system for former non-database users. Fourthly, conventional von Neumann type computers were designed for the preparation and execution of compute bound programs and for very simple data processing only [Info78a], [Hsia78]. A typical conventional computer would utilize much of its power to interpret data base management requests instead of actually executing them. This is an inherent architectural drawback rather than the problems associated with the software design. Present research in data base machines is directed to transfer data base management functions to specialized hardware to improve performance. Therefore it does not necessarily follow that writing a vastly superior operating system would significantly improve the performance of data base application.

On the other hand taking the second approach is not free from problems. There are many operating systems available commercially, each with its own characteristics and applications. A general purpose operating system providing minimum service essential to the operation of the DBMS is assumed. Basically the model proposed in this chapter assumes that the file system of the operating system can handle random access files. This assumption is justified because most existing operating systems provide data handling facilities much more sophisticated than this requirement.

The research is limited to designing a single user DBMS. The single user DBMS is link-edited to the user application program requiring it. The routines in the DBMS model are implemented as library subroutines and DBMS requests in the application program are transformed into subroutine calls by a preprocessor.

The structuring approach is used to build up the DBMS model. There are two distinct ways of structuring a system [Horn73]. The first technique is to regard the system as comprising of interacting subsystems, where each subsystem is reponsible for part of the operation of the total system. The second method is to represent the system by a sequence of models, each conveying details of behaviour relevent to a chosen level of abstraction of its operation.

The first technique has been discussed extensively by Parnas [Parn72a], [Parn72b] and also by Myers [Myer76]. It is termed modularization. A system is composed of modules. Each module is specified by its input and output parameters and the set of their possible values, the initial values of its local variables, the effect and the action to be taken when an error occurs. A module is therefore a black box with sufficient information to interact with the outside world. This method allows one module to be written with minimal knowledge of the code in another module. It also allows modules to be reassembled and replaced without affecting other modules of the system. Apart from these advantages, another major benefit provided by modularization is that one can test the specifications of the modules long before the programs for the modules are coded. However, as Parnas remarked, the modularization approach is good for a small piece of software only.

The second technique of structuring, termed incremental design or hierarchial structuring, is to build up a system level by level, bottom-up. This technique has been discussed in detail by Goos [Goos75]. Each level defines a virtual machine or abstract machine to be used by higher levels [Habe76]. A given virtual machine provides a set of apparently atomic facilities which are used to construct another set of facilities that constitute a higher level virtual machine interface [Rand75]. Therefore, each virtual machine is an abstraction of the virtual machine below it. The system can be designed and understood separately in terms of a single level and two virtual machine interfaces that bound the level. Such a system increases its comprehensibility. Furthermore, it enhances system error recovery [Denn76]. The THE [Dijk68], [McKe76] and the BSM operating system [Goos72] are examples of this technique. Habermann further distinguishes the fact that hierarchial structuring is based on functions whereas the THE system is based on processes. The latter approach is not desirable because it increases execution time [Habe76].

Information modules consist of some data structures and a set of functions which share the knowledge of a particular design decision, for example, the details of the data structures. A level is a set of function names which are implemented by functions in lower levels. As Habermann points out, the two concepts are distinct. A module can span several levels and a level can contain a number of modules [Habe76].

Some of the concepts discussed above are for the design of operating systems while others are for general software design. They are borrowed and employed in the design of the MCDBMS. The approach taken is to achieve a nearly decomposible, hierarchial structure. Within each level further functional sublevels may be identified. Within each sublevel modules are specified. A module can be a subsystem and may be further decomposed into submodules performing atomic functions.

The model designed is based on CODASYL's 1978 specification of the schema DDL [CODA78] and the DSDL associated with it. The advantages and application of DSDL for data independence have been investigated by Tozer [Toze78], O'Connell [OCon78] and Stancey [Stan78]. Multilevel structures of data bases have been studied at both the logical and physical levels [Senk73], [Senk75a], [Olle75], [Mana76], [Toh77], [SDDT77]. In particular, the Data Independence Accessing Model (DIAM) has a strong influence on the present model.

3.2 GENERAL OVERVIEW

This section gives a general overview of the MCDBMS model. It also compares this model with other models.

3.2.1 General Model Description

Fig. 3.1 shows a pictorial view of the MCDBMS model. Rectangles represent constructs at various levels while ovals represent interlevel mappings between constructs. The six MCDBMS levels are labelled on the right while the corresponding three levels of CODASYL levels are labelled on the left. This representation of the model resembles that of DIAM II presented by Socket and Goldberg [Sock79].

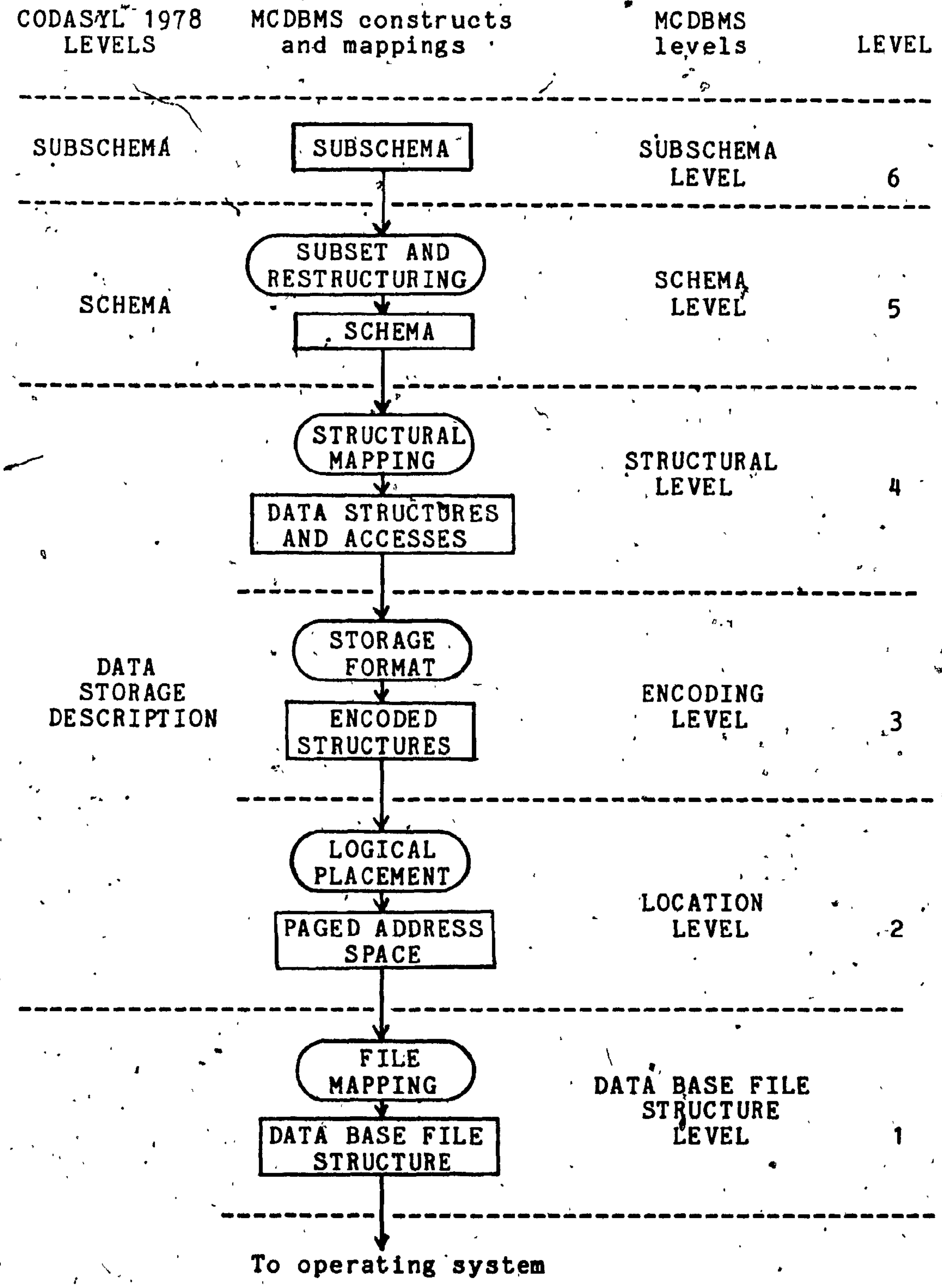


Fig. 3.1 Multilevel CODASYL DBMS Model (MCDBMS)

The multilevel structure of the MCDBMS model provides a series of abstractions to the level beside it. The subschema presents part of the schema to a user. It is a level which is "more abstract" than the schema level [Ullm80]. For example, a subschema field requiring age of employees need not mean that the employee's age be stored in the data base. Otherwise this attribute has to be updated frequently. The schema may contain the birth date of individual employees. Simple computation with the present date can meet the requirement of the subschema. Entities and their relationships are described by the subschema DDL.

The schema level contains the CODASYL network data model. The schema DDL defines the attributes and relationships among them. It also defines the linkages among relationships, for example, the ordering of members within a given set type. With the definition of linkages, the access paths independent of the set representation are also determined.

Employing the definition of Horowitz and Sahni [Horo76], the structural level is a representation of the record and set data structures of the CODASYL schema DDL. The structural level DDL describes the data structures in this level. It corresponds to the majority of clauses in the DSDL 1978 specification. Set relationships are usually represented by chain, pointer array or index. Access paths are most clearly seen at this level. The structural level provides a level of abstraction to the level above. The user of the level above need not be concerned with the ways in which a set is represented. A record in this level is known as a structural level record (SLR). Apart from the data items specified in the schema DDL, a SLR also contains pointer data items for set relationships. Header information such as record data base key and record length are also included in the SLR.

The encoding level reduces all primitives of the structural level, namely record, pointer array and index, into a basic data structure -- storage record. This level therefore describes how the data structures above are represented in one dimensional bit streams. The encoding level DDL describes the data structures in this level. The definition of storage records from schema records in DSDL 1978 is an example similar to the function of the encoding level DDL. Starting from this level the logical aspect of the data base fades away and the physical aspect comes in. This level provides an abstraction of the implementation of primitives of the structural level. It is fascinating to note that this level also provides an abstraction to the level below. The level below handles a storage record as a bit string which is an atomic unit.

The location level maps the storage records to a paged address space. Conceptually one can visualize this in two stages. A bit string is first mapped onto a one dimensional address space. The one dimensional address space is then mapped to a paged address space. However, a restriction is imposed on the second stage of mapping: no bit string can cross data base page boundaries. The composite mapping is described by the location level DDL which basically corresponds to the PLACEMENT clause of the DSDL 1978. The next lower level handles pages without knowing what each page means. The lower level is unaware of the DBMS above it. This is an abstraction provided by the location level to the level below.

The data base file structure level serves as an interface between the MCDBMS and the operating system. It maps the paged address space into records of files so that the operating system can recognize them. Access and modification of data base pages is reduced to reading and writing of records of a file by a user program. The mapping is described by the data base file structure DDL, a subset of which corresponds to the STORAGE AREA -- INITIAL SIZE clause of the DSDL 1978. This level therefore abstracts away the concept of data base pages from the operating system. The operating system cannot distinguish the difference between the DBMS and other user programs.

It can be seen that requirements of a more complicated nature have to be met at both ends of the MCDBMS. At the top the user requirements have to be satisfied. At the bottom the operating system specifications have to be fulfilled. It is therefore reasonable to have a level in between the two extremes to have a basic data structure as a primitive building block. In the MCDBMS model this level is the encoding level of storage records. Abstract machines are added both upwards and downwards as one proceeds in both directions. The model designed can be considered as having achieved the goal of hierarchial structuring.

3.2.2 Comparison with DIAM II and ANSI/SPARC

Fig. 3.2 compares the MCDBMS model with the data independent access model (DIAM II) model and the ANSI SPARC specifications [Yoem76]. The end-user level of DIAM II is not compatible with the subschema level of MCDBMS, neither is the information level compatible with the schema level. This is because the end-user level and the information levels deal with entities and their relationships, for example, employer and position, while the subschema and the schema levels are specific models to represent the entities and their relationships. Records are used to represent entities and sets are used to denote relationships. For example, a position is an owner record while all employees are member records of the set mapping position to employees.

The string level of DIAM II consists of three primitives, atomic strings (A-strings), entity strings (E-strings) and link strings (L-strings) [Senk73]. These are representations just as the use of data items, records

DIAM II LEVELS	MCDBMS LEVELS	ANSI/SPARC LEVELS
end-user level		
	subschema level	external schema
information level		
string level	schema level	conceptual schema
encoding level	structural level	internal schema
	encoding level	
	location level	
physical device level	data base file structure level	

Fig. 3.2 Comparison of the level structure of DIAM II, MCDBMS and ANSI/SPARC

and sets of the schema DDL. The subschema and the schema of MCDBMS correspond to the external schema and the conceptual schema of the ANSI SPAR specifications.

The encoding level of the DIAM II model consists of encoding the constructs of the string level and mapping them onto a linear address space. This corresponds to the structural level, the encoding level and the location level of the MCDBMS model.

The physical device level, which is further expanded by the SDDTTG [SDDT77], would correspond to the data base file structure level and the operating system. The ANSI SPARC specifications classify everything below its conceptual schema as internal schema. This categorization is similar to the CODASYL 1978 specification of the DSDL, though the DSDL does not include the operating system.

An overall view of the MCDBMS has been presented. Wulf says that to describe a multilevel model by the top-down approach is to explain the higher level concepts, which is most abstract, in terms of primitives which await to be defined as one goes top-down. Conversely, the bottom-up approach is to build a vocabulary without letting the reader know where he is being led [Wulf75]. The reader has been given a bird's eye view on the functional aspects of individual levels of the model, how two adjacent levels relate to each other and how the model meets requirements at both ends. The next chapter gives a functional description of the modules and the data structures assumed in each level of the MCDBMS. The description proceeds in a bottom-up direction because the reader knows where the implementational design is heading towards. However, before the detailed description, some general considerations regarding the model are given below.

3.3 GENERAL IMPLEMENTATIONAL CONSIDERATIONS

3.3.1 DML Statements

The MCDBMS described above only intends to implement a subset of the CODASYL DDLC 1976 and 1978 proposals. The DML statements that are to be implemented are shown in Table 3.1. They are typical operations of a DBMS.

CATEGORY	COBOL JOD DML
record only	FIND ERASE STORE
record and item	GET MODIFY
linkage	CONNECT DISCONNECT
currency indicators	ACCEPT

Table 3.1 COBOL JOD DML statements implemented

The major features that are not implemented are shown as COBOL JOD DML statements in Table 3.2. The DML statements for realm and currency are not implemented because the system is a single user system while these two categories are concerned with concurrent processing. The realm category is also related to privacy which is not implemented to simplify the model. The others are also omitted for the purpose of simplification of the model.

CATEGORY	COBOL JOD DML
realm	READY FINISH
set	ORDER
conditional	IF
declarative	USE
concurrency	KEEP FREE REMONITOR

Table 3.2 COBOL JOD DML statements not implemented

3.3.2 Levels and Modules

The MCDBMS contains six levels. Each level is further subdivided into sublevels of abstraction. A sublevel contains modules of the same or different classes. A class serves as a boundary within which modules share the data structures local to its class only. This confinement partitions modules of the same sublevel further because a class is confined to modules of the same sublevel only.

Services provided by a sublevel to the sublevel above can be classified as general and specific. Services of a general nature can be used by different classes of modules in the sublevel above. Services of a specific nature can be utilized by only one class of modules. Therefore there is partitioning between modules of different sublevels. Modules in sublevel zero of all levels are information providing modules, giving mainly the information on the DDL of the level. All modules in all sublevels of a level are allowed to call these sublevel zero modules. In general, however, the aim of the design is that modules of a sublevel call modules of the sublevel immediately below them only. Services provided by a level, which is implemented in the uppermost sublevel of that level, are available to all modules on all sublevels of the level immediately above it.

Each module is given a module number of five digits. The most significant digit is the level number for the MCDBMS. The next two significant digits constitute the sublevel number of the level within which the module resides. Each sublevel can hold at most one hundred modules, indicated by the two least significant digits of the module number. The one hundred values, from 00 to 99, are further subdivided into ten groups. Each group contains ten modules. Group one modules have the two least significant digits from 00 to 09, group two from 10 to 19, and so on. Modules of the same group belong to the same class.

The modules are not primarily designed for high efficiency processing, rather, they are devised to demonstrate that the model proposed can satisfy the CODASYL 1978 specifications.

In general, modules exist as a necessary functional component. They serve to provide abstractions to the level above. However, some modules exist not as an absolutely necessary component but to reduce the size and complexity of higher level modules.

3.3.3 Error Handling

It is assumed that syntactical errors and other compilation errors are checked out during preprocessing and compilation time. Execution time error indication is still an important aspect in the MCDBMS design. Each sublevel has a status register for error indication. The status registers are named by their level name and sublevel number. For example, EL3-STATUS refers to the status register of sublevel three of the encoding level.

Each status register contains two components or fields. The first field is a two digit number which coincides with the two least significant digits of the module number of the executing module in the sublevel. The second field is a four digit number indicating the error type upon execution of that module.

Error types of all modules are indicated by powers of two starting from zero, for example, 1, 2, 4, 8, 16,... up to 2048. A module can have a maximum of twelve error types. This number serves to set a limit on the size of the modules. Each module has its own particular type under the same error number convention. Choosing error numbers in powers of two enables us to accumulate error numbers for the occurrence of more than one error type during execution. Mathematically,

$$\sum_{i=0}^{n} 2^{i} < 2^{n+1} \quad \text{for } n > 0$$

Errors can be classified as fatal or non-fatal. When a module encounters a non-fatal error, it attempts recovery and resumes execution with the error recorded in the second field of the status register. When a fatal error occurs, the error is indicated in the status register and control is returned to the higher level module calling it.

A fatal error may or may not require rollback action. If there is an attempt to modify a SLR, one or more storage records may have to be modified. If the SLR is mapped to two storage records, failure to modify the first storage record and write it back into the data base would result in fatal error but no rollback action is necessary. On the other hand, failure to modify the second storage record after the first storage record has been successfully written back results in a fatal error which requires rollback to maintain the data base consistency.

The example in the preceeding paragraph gives us some insight concerning rollback determination. The higher level module examines the conditions under which the fatal error occurs in the lower level module. It then determines whether rollback is required or not. Conversely, if a lower level module makes two types of modifications, it may also determine whether rollback is required. Once a module at a certain sublevel determines that rollback is required, the rollback requirement is propagated upwards and control is returned to higher level calling modules until the highest level calling module receives the message. Rollback indication is denoted by adding the value 4096 to the second field of the status register. This is why the error number is up to 2048 only. Any status register with the second field exceeding the value 4096 signifies that rollback is indispensable. In other words, a '1' in the twelth bit (starting from bit zero) of the second field indicates the rollback requirement.

The modules are generalized so that unsuccessful execution need not lead to an error condition. For example, the search for a particular record in a set should not give rise to an error if the search is unsuccessful. The search may have been intended to ensure that the record is not already in the set before inserting the record into it. The result of execution is indicated as an output parameter, which will be used by the calling module to decide whether an error condition exists.

If there are no errors the second field of the status register will remain zero. The first field contains the value of the module executed on the sublevel. Thus error recording scheme enables the DBA to identify precisely at which calling sequence the error occurs . It also facilitates error analysis if all values of local variables, parameters, and status registers of the modules in the calling sequence are recorded in an error report.

3.3.4 Data Base Key

Unique schema record occurrences are identified by unique data base keys. The DBTG specification [DBTG71] requires data base keys to be unique throughout the life time of the DBMS. This policy is followed in the MCDBMS model. There is a one to one mapping between schema records and SLRs. Hence data base key also uniquely identifies a SLR. The structure of the data base dey is as shown in Fig.3.3.

```
+------------------------------------+
| CODED DBK TYPE | ACCESSION NUMBER|
+------------------------------------+
```

Fig. 3.3. Structure of the data base key

A data base key consists of two parts. The first part is a coded data base key type (cdbkt) which identifies the record type of the SLR. Record types are represented numerically in the compiled schema DDL. The second part is an accession number. Thus number increases by one every time a new occurrence of the record type is created. The unique value of data base key is assigned to every SLR during creation.

The data base key concept is extended to identify other data structures of the structural level, the pointer array and the index. To distinguish such a convention from that of the CODASYL DDLC 1978 specification [CODA78], data base key is renamed logical data base key in the MCDBMS. Pointer arrays and indices are constructs in the logical level of a data base. The renaming is therefore consistent with the data structures it identifies.

The data structure for a general logical data base key is as follows:

```
01  rec-ldbk, PA-ldbk, IND-ldbk    record, pointer array or
              index logical data base key
    02  cdbkt           coded data base key type
        03  cdt         coded data type
        03  crst        coded record/set type
    02  an              accession number
```

The value of the coded data type determines which data structure the logical data base key represents. For the MCDBMS model the representation is as follows.

cdt	ldbk
0	rec-ldbk
1	PA-ldbk
2	IND-ldbk

Table 3.3 Representation of the coded data type in the logical data base key

Two bits are sufficient for this subfield with a possible expansion of adding one more data structure in the structural level.

The coded record or set type (crst) is the record type number or set type number. Set types are numbered as record types in the compiled schema DDL. If the value of this subfield is zero, the logical data base key identifies a SLR. If the the value of this subfield is one or two, the logical data base key identifies a pointer array or an index.

The accession number increments by one every time a new occurrence of a SLR is created. This value is shared by the pointer array and the index as well. The CODASYL DDLC 1978 specification requires one set type to have one and only one owner record type. One pointer array or one index is used to implement a set occurrence, therefore no problem of non-uniqueness would occur. There is a one to one mapping between the owner record logical data base key and the corresponding pointer array or index logical data base key. The accession number of the owner record occurrence coincides with that of the pointer array or the index. Knowing the value of the pointer array or index logical data base key and the set type can enable us to generate the owner record logical data base key easily. This would facilitate consistency checking.

A SLR can be split into two or more storage records. The concept of logical data base key cannot be carried into storage records, otherwise the problem of non-uniqueness would occur. A storage data base key is formed based on an extension of the logical data base key. A field of storage code (sc) is concatenated onto the logical data base key, as shown in Fig. 3.4.

```
+-------------------------------------------------------+
| CODED DBK TYPE | ACCESSION NUMBER | STORAGE CODE |
+-------------------------------------------------------+
```

Fig. 3.4 Structure of the storage data base key

When a SLR is split into two or more storage records, the storage records have the same logical data base key but different numeric value of the storage code. As storage records for a pointer array are created, the storage code increases stepwise by one. The accession numbers of various pointer array storage records representing the pointer array coincides with that of the pointer array occurrence, so does the coded data base key type. An index occurrence is a two level structure. The upper level is an index header. The storage code of the storage data base key of this index header storage record is zero. The accession number of the index header storage record is the same as that of the index occurrence. The lower level contains index storage records which have their storage code increased stepwise by one as they are created, just as that of the pointer array storage records. The accession number of the index storage records are the same as that of the index occurrence.

3.3.5 Linear and Paged Address Space

There are two strategies to implement the data structure central to the Location level. The first one is mapping storage records to a linear address space while the second one is mapping them to a paged address space. Mapping to a linear address space is conceptually simple but mapping to a paged address space is more suitable from a practical point of view. The former strategy is discussed first.

An owner record may have many member records. If the placement option of VIA SET NEAR OWNER is used for member records using hashing, a significant number of rehashing is required. Moreover, a record may cross file buffer boundary because different data base record types may differ in length. The linear address space approach is therefore not desirable.

The problem of locating the storage record is pushed from the file system to the DBMS. In the MCDBMS model this is the location level. The difficulty with a linear address space lies in the decision of determining the record length for access. Because different types of storage records vary in length, it is conceivable that a storage record would cross file record boundary. Storage record synthesis is required.

A paged address space can bypass the problems mentioned above. The linear address space is subdivided into pages numbered consecutively. Storage records are mapped onto pages. No absolute addressing is required to access a storage record. Only the page number is required. A routine to search for the presence of the desired storage record within a page solves the problem above. Another routine to access neighbouring pages for searches if the search failed in the initial target page can implement the VIA SET NEAR OWNER option. Furthermore, with the requirement that no data base record can cross page boundaries, the complexity of record handling can be reduced.

3.4 THE FILE SYSTEM

The file system is the foundation on which the MCDBMS is built. Thus an overview of the file system is essential before giving the detail description of the MCDBMS modules.

Madnick and Alsop wrote a classical paper on file system titled 'A modular approach to file system design' [Madn69]. Madnick and Donovan based their file system on the Madnick and Alsop paper and elaborated it in their book 'Operating Systems' [Madn74]. The file system described by the latter pair of authors will be discussed. This model would act as the file system to provide facilities to the DBMS. A slight modification is made in command transfer between various levels of the file system. File system of the MCDBMS requires a subset of the access methods of thier system. The model is described below.

Fig. 3.5 shows the structure of the file system, in which six levels can be identified. The file system is called by the CALL SFS command, containing arguments of the filename,the function required (read/write), 'next' or 'rec#' for sequential or random access, the process number making the call and the main memory location to which the logical record is to be transferred.

The Symbolic File System (SFS) translates a filename into a unique file identifier. A unique file identifier is needed because different users may access the same file under different names and different users can give the same name to different files. The Basic File System (BFS) locates the file directory entry from the file identifier and puts it into the Active File Table (AFT). The Access Control Verification (ACV) makes use of techniques such as password, access control matrix, cryptography to check for access rights.

The Logical File System (LFS) is concerned with mapping the structure of logical records onto the linear byte-string view of a file provided by the Physical File System (PFS). The LFS supports sequential or direct access to both fixed and variable length records. Based on the record

```
                  | CALL SFS(fln,fn, next ,p#,MMloc)
                  V                 rec#
----------------------------------------------------------------
+---------+
|SYMBOLIC|
|  FILE  |
| SYSTEM |
+---------+       | CALL BFS(fid,fn, next ,p#,MMloc)
                  V                 rec#
----------------------------------------------------------------
+-------+
| BASIC|
| FILE |
|SYSTEM|
+-------+         | CALL AVC(AFTentry#,fn, next ,p#,MMloc)
                  V                        rec#
----------------------------------------------------------------
+------------+
|   ACCESS   |
|  CONTROL   |
|VERIFICATION|
+------------+    | CALL LFS(AFTentry#,fn, next ,p#,MMloc)
                  V                        rec#
----------------------------------------------------------------
+--------+
|LOGICAL|
|  FILE |
| SYSTEM|
+--------+        | CALL PFSi(AFTentry#,fn,lba,lbl,p#,MMloc)
                  V
----------------------------------------------------------------
+---------+
|PHYSICAL|
|  FILE  |
| SYSTEMi|
+---------+       | CALL DSMi(fn,dev#,pbn,buf[ptr],#blks,p#)
                  V
----------------------------------------------------------------
+-----------+  +---------+  +----------+  +--------+
|ALLOCATION|  | DEVICE  |  |   I/O    |  | DEVICE|
| STRATEGY |  |STRATEGY|  |INITIATOR|  |HANDLER|
|  MODULEi |  | MODULEi|  |    i     |  |   i   |
+-----------+  +---------+  +----------+  +--------+
----------------------------------------------------------------
fln -- filename                     fn  -- function (READ,WRITE)
rec# - record number                p#  -- process number
MMloc  -- main memory location      fld -- file identifier
AFTentry# -- Active File Table entry number
lba -- logical byte address         lbl -- logical byte length
pbn -- physical block number        dev# - device number
#blks  -- number of blocks to be transferred
```

Fig. 3.5 A six level file system

specification 'next' or 'rec#' and the file structure from the AFT, the LFS converts a request for a record into a request for a byte string.

A file is treated as a sequential byte string without any explicit record format by the PFS. The PFS uses file buffering to minimize I/O by keeping track of records in the same unit of I/O transfer. If the function (fn) is READ, the device sends data to a file buffer in the PFS in the unit of a physical block.. The logical record is transferred from the file buffer to the user buffer in the main memory. If the function is WRITE, the logical record is transferred from the user buffer to the file buffer and then the entire physical block is written onto the storage device. All data transfers with lower levels are via the file buffer. PFS also allows the logical record size to be independent of the physical block size. It also allows non-contiguous allocation of file space by chained blocks or file maps. It also transfers the appropriate bit string to the user's buffer once the block is loaded into the file buffer.

The PFS calls the Allocation Strategy Module (ASM) for assigning or releasing storage space. It also calls the Device Strategy Module (DSM), Input/Output Initiator (IOI) and the Device Handler (DH). These modules function as follows

1) mapping physical address to device address
2) create channel programs

3) request input/output (I/O scheduler)

4) handle I/O interrupts and error conditions.

Modules implementing the file system described above are assumed to be available. The file system described here serves as a starting point of the MCDBMS which uses it and is described from the bottom up in the next chapter.

Chapter IV
DETAIL DBMS MODEL DESIGN

This chapter describes the implementational design of the MCDBMS described in the previous chapter. The model is described from the bottom, level by level. Some levels are subdivided into sublevels which are also described from the bottom. The data structures required are described in the form of declarations in the Working Storage Section of COBOL. They are followed by the syntax of commands to operate on the data structures. The modules are described in terms of the functions they perform in Appendix A. This chapter only gives a brief description of the functions performed by each class of modules within each sublevel.

The function of this chapter is to verify on paper that the MCDBMS model described in the preceeding chapter can be implemented. The actual implementation would take an enormous amount of time. Partial implementation of a small subset of the system does not necessarily imply that the entire system can be implemented. A paper design of the model is therefore given.

4.1 THE DATA BASE FILE STRUCTURE LEVEL

The data base file structure level is the bottom level of the MCDBMS. It interfaces the DBMS with the operating system. The function of this level is two-fold. First, it maps the paged address space of the location level into records of files. Second, it transforms data base page manipulation requests to record manipulation requests of files. This level assumes two requests that can be recognized by the operating system: a read request and a write request. Both requests require the specification of the file name and the record number intended for the operation. Upon receiving the command, the operating system supplies the process number and the main memory location to which the record is to be transferred. It then passes the command to the file system.

The data base page size corresponds to the size of a physical page. Different storage devices may have different physical page sizes. The physical page size for a storage device is constant. A physical page or a block of an integral number of physical pages is a unit of I/O transfer. The choice of the data base page size is intended to simplify the computation process for random accesses of the physical records of a file. A file is restricted to reside within a storage device and therefore each data file contains fixed length records. Even though a number of storage devices with different page sizes are used, the page size of a file remains fixed.

This level contains two sublevels. Sublevel zero contains the module DBF-MAP to map the data base pages to the files and records. Sublevel one transforms a command to load a data base page to a read request for the operating system. Similarly, it transforms a command to store a data base page to a write request for the operating system. The modules LOAD-PG and STORE-PG perform these functions. File name and key are supplied by the sublevel one modules as parameters for the call to the operating system. All accesses to the storage records take place in the level above. The operating system is not burdened with the identification of which file a storage record resides in and where it resides within the file. It is simply supplied with a file name, a record number and the function to be performed by the data base file structure level. The modules for this level are shown in Appendix A, Table A.1.

Apart from providing an abstraction to the operating system mentioned in section 3.2, this level also provides an abstraction to the location level. The location level is unaware of how the operating system handles the data base pages or where a data base page dwells.

4.2 THE LOCATION LEVEL

The location level is level two of the MCDBMS. There are six sublevels within this level. Modules in this level share a common data structure, the location level buffer (LLB), which has its length equal to the largest page size of the paged address space. Data structures local to a sublevel are defined just before the functional descriptions of the modules of the sublevel.

4.2.1 Sublevel Zero

Sublevel zero contains two classes of modules. Class one contains one module to provide information on placement strategies of storage records as defined by the location level DDL. The location level DDL specifies that the pointer array and index storage records are to be placed as close as possible to the owner storage record occurrence to which they belong. The single module of class two provides free pages to the modules requesting one. The module DB-SPACE is in class one while the module LL-INFO is in class two. Appendix A, Table A.2 shows the characteristics of these two modules.

4.2.2 Sublevel One

This sublevel contains two classes of modules. One class of modules manipulates the physical currency table. The other class of modules performs operation on the direct index directory of storage records.

4.2.2.1 Physical currency location table

A physical currency location table (PCLT) is maintained for different pointer array and index storage record types. An example of the application of the PCLT is as follows. A pointer array storage record is accessed. The DBMS obtains a logical data base key from a pointer array element. The SLR is loaded and the values of one or more of its fields are examined. If the values do not satisfy the desired condition, the next element of the pointer array is retrieved. However, the pointer array storage record has to be retrieved first. Since the pointer array storage record last accessed has its page number stored in the PCLT, a search of the PCLT is sufficient to obtain the page number of the pointer array storage record. Otherwise the owner record has to be located, then the pointer array for the appropriate set is located. The PCLT reduces the amount of I/O considerably.

The above example shows the advantage of storing the pointer array and index storage records in the data base. However, SLRs do not have their storage records listed in the PCLT. Repeated accesses of the same SLR are much less common as compared with that of the pointer array and indices. Continual update of the PCLT for SLRs while traversing the searching path wastes processing time.

The structure of the PCLT is indicated below

```
01  PCLT                      physical currency location table
    02  num-of-entries        number of entries in the table
    02  entry-pair
        03  sdbk              storage data base key
            04  cdbkt         coded data base key type
            04  an            accession number
            04  sc            storage code
        03  sdbk-pg#          page number where the storage
                              record resides
```

The syntax of commands to access the entries of this table are as follows

```
[GET] num-of-entries [FROM] PCLT
[PUT]                [INTO]
```

and

$\begin{bmatrix}\text{GET}\\ \text{PUT}\end{bmatrix}$ (entry-num)TH ENTRY PAIR (sdbk, sdbk-pg#) $\begin{bmatrix}\text{FROM}\\ \text{INTO}\end{bmatrix}$ PCLT

where entry-num is the entry number of the entry pair in the table.

Three modules provide operations on the PCLT. They are DELETE-CUR-LOC, INSERT-CUR-LOC and SEARCH-CUR-LOC. They serve to delete, insert and search the table. These three modules from a class of modules in sublevel one. They are shown as the first three modules in Appendix A, Table A.3.

4.2.2.2 Direct index directory

A placement option of some storage record is CLUSTERED NEAR SET VIA OWNER. The owner and member records defined in the schema DDL have their relationships carried down and redefined at the location level for storage records. Locating a storage record with this placement option requires the storage data base key of its owner storage record. However, the required storage data base key may not be available to the MCDBMS.

In order to tackle this problem, there is a direct index for each storage record type that requires an indication of the location of its storage records. Each entry of the direct index contains an identification of a storage record and the page number where the storage record resides. Record types with a placement option of CALC have their own hashing algorithms. They do not need direct indices. A direct index directory (DID) is required to determine the page number of each direct index so as to locate the appropriate direct index for a given storage record type. The DID is stored in the data base. The data structure of a DID page is

```
01 DID PG              direct index directory page
   02 header info      header information
   02 next-diry-pg#    page number of the next page
                       of the DID
   02 num-of-entries   number of entries in the page
   02 entry OCCURS max-entries TIMES
   03 cdbkt            coded data base key type
   03 sc               storage code
   03 pg#              page number of the first direct
                       index page of the storage
                       record type
```

where max-entries is the maximum number of entries that a DID page can hold. The syntax of commands to handle the level 02 data items other than the entry is

$$\left[\begin{matrix}\text{GET}\\\text{PUT}\end{matrix}\right] \text{data-name-1} \left[\begin{matrix}\text{FROM}\\\text{INTO}\end{matrix}\right] \text{DID PG}$$

the syntax of commands to handle the entry is

$$\left[\begin{matrix}\text{GET}\\\text{PUT}\end{matrix}\right] \text{(entry-num)TH ENTRY (cdbkt, sc, pg\#)} \left[\begin{matrix}\text{FROM}\\\text{INTO}\end{matrix}\right] \text{DID PG}$$

where data-name-1 is the data name of level 02 data items without sublevel 03 items entry-num is the entry number of the entry within the DID page.

One DID page occupies one data base page. No other storage record is allowed in this page. The DID pages are chained together. The first DID page is preassigned in physical storage. The entries are not ordered. A sequential search is necessary for finding a specific storage record type. A sequential search could be sufficient for data bases that do not have many record types. Deletion of an entry in a DID page results in overwriting the entry to be deleted by the last occupied entry. The number of entries of the page is then reduced by one. Insertion results in entering the entry into the location immediately following the last occupied entry. The number of entries of the DID page is increased by one. If the DID page is full, the next DID page is accessed for insertion. Search for an empty entry space starts from the first DID page. If all the DID pages are full, a free page is requested from the data base for insertion. The new DID page is inserted to the rear of the DID page chain. Five modules to handle the DID constitute the class two modules of sublevel one. They are DELETE-DIRY-ENTRY, INSERT-DIRY-ENTRY, MODIFY-DIRY-ENTRY, RETRIEVE-DIRY-ENTRY and SEARCH-DIRY-ENTRY. The last five modules of Appendix A, Table A.3 shows the modules of this sublevel.

4.2.3 Sublevel Two

Sublevel two consists of three classes of modules. The single class one module CUR-LOC-OP makes use of the service provided by sublevel one to manipulate the PCLT. All requests to the PCLT are channelled through this module. Class two modules provide five operations to manipulate direct index which locates the page where a storage record resides. The single class three module searches for a storage record given a data base page.

4.2.3.1 Direct index

Class two modules are reponsible for manipulating direct indices. Each direct index (DI) contains the information of all storage record occurrence of a storage record type currently in the data base. Each entry of the DID points to the DI. The data structure for a DI page is as follows

```
01  DI-PG#                  direct index page
    02 header info          header information
    02 next-ind-pg#         page number of the next DI page
    02 num-of-entries       number of entries in the page
    02 entry occurs max-entries TIMES
    03 an                   storage record accession number
    03 pg#                  page number where the storage
                            record resides
```

where max-entries is the maximum number of entries that a DI page can accomodate.

The data structure of a DI page is very similar to that of the DID page, so is the syntax of commands to handle data items within the DI page. For this reason the syntax is not repeated here. The modules which operate on the DI also bear great resemblance to those of the DID. The searching can be further improved from sequential to binary. However, for this research, it is sufficient to retain this inefficient structure for recovery considerations. The modules which operate on the DI are DELETE-DIR-INDEX, INSERT-DIR-INDEX, MODIFY-DIR-INDEX, RETRIEVE-DIR-INDEX and SEARCH-DIR-INDEX. They are shown as the second to the sixth modules in Table A.4 of Appendix A.

4.2.3.2 Storage record

The data structure for a storage record is as follows

```
01 sto-rec               storage record
   02 rec-sdbk           storage data base key of the
                         storage record
   02 rec-length         length of the storage record
   02 other header info other header information
   02 content
```

The syntax of commands to access the entries of the storage record is

$\begin{bmatrix}\text{GET}\\\text{PUT}\end{bmatrix}$ data-name $\begin{bmatrix}\text{FROM}\\\text{INTO}\end{bmatrix}$ STO REC

where data-name is the name of the level 02 data item of the storage record.

4.2.3.3 Data base page

The size of the LLB is such that it can hold the largest page of the storage devices. Data base pages are stored as records of a file with one page per record. Accessing a record of a file by the data base file structure level results in the transfer of a record which is a data base page into the LLB. Operations on the data base page can be considered as operations on the LLB. The data structure of a data base page is as follows

```
01 data base page
   02 pg-header        page header
   02 pg-number        data base page number
   02 free space list header
      03 size             size of the free space list header
      03 link             points to the next free space block
                          in the page
```

The data structure of a free space block within a page is as follows

```
01 free space block
   02 free space node
      03 size              size of the free space block
      03 link              points to the next free space
                           block in the page
```

The syntax of commands to manipulate the data items are

Format 1 $\begin{bmatrix}\text{GET}\\\text{PUT}\end{bmatrix}$ data-name-1 $\begin{bmatrix}\text{FROM}\\\text{INTO}\end{bmatrix}$ LLB

Format 2 $\begin{bmatrix}\text{GET}\\\text{PUT}\end{bmatrix}$ data-name-2 (var-name-1) $\begin{bmatrix}\text{FROM}\\\text{INTO}\end{bmatrix}$ LLB

where

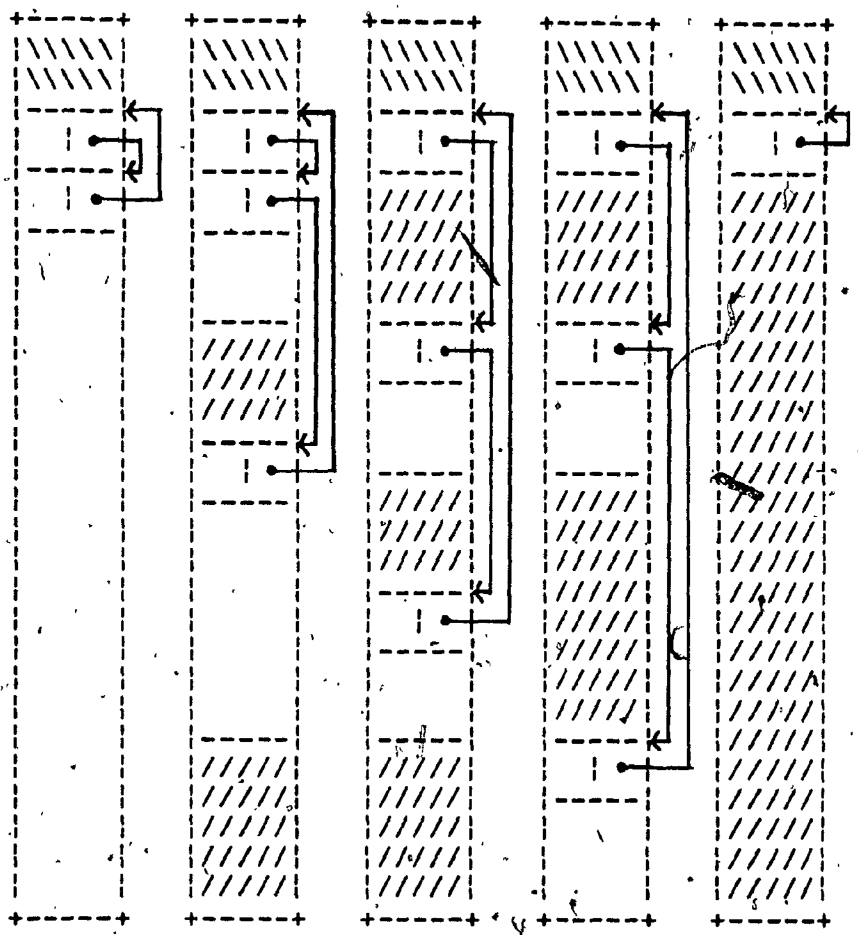
data-name-1 is the data name for level 02 data items without further sublevels,

data-name-2 is the data name for level 03 data items of the free space list header or the free space block,

var-name-1 is the variable name with value equal to the starting address of the free space node or the free space list header.

The location of the free space list header is fixed. The value of var-name-1 is known. As for free space blocks, the value of var-name-1 identifies the free space block we are considering. All space not identified by the free space list is filled.

The searching algorithm for a storage record in a page must ensure that the storage record to be located is not a free space block and that the searching has not exceeded the page boundary. There are five cases that the searching algorithm has to be aware of. The five cases are shown in Fig. 4.1. The SEARCH-STRING-IN-PAGE module performs the search. It is shown as the last module in Appendix A, Table A.4.



▧ page header

☐ free space

▨ occupied

Fig. 4.1 Five cases to consider for a storage record search in a page

4.2.4 Sublevel Three

The modules in sublevel three all belong to the same class. The functions performed by them are the following: space allocation, alternate page placement of a storage record, definition of the boundary within which a storage record type may reside and a first estimate of the location of a storage record. The modules are known as ALLOCATE, ALT-PLACEMENT, APPROX-PLACEMENT and BOUND. They are shown in Appendix A, Table A.5. The modules of this level serve to simplify the modules of sublevel four at the next higher sublevel.

The structure of the data base page is based on the one proposed by Horowitz and Sahni[1] Free space manipulation follows their scheme. Consider a request comes from a run unit to store a bit string of a given length. The space allocation algorithm assigns the first free space block with sufficient size to hold a storage record during a search for free space. The remaining free space is reduced to a free space block of smaller size. If the remaining free space cannot hold an entire free space node, the free space block is not assigned and another one must be chosen.

If a page does not contain free space suitable for assignment, an alternative page is selected. Alternate

[1]Horowitz, E.; and Sahni, S. Fundamentals of Data Structures, Computer Science Press, CA, USA, 1976, pp. 142-155.

pages are selected on the basis of proximity. The initial target page is used as a focal point and neighbouring pages are searched as the radius increases stepwise by one in both directions. This strategy is used for both storing and searching of storage records. A module implements this strategy.

If the storage record to be located or placed is of placement option CALC, the hashing function is applied to the storage data base key of the storage record to obtain the focal page number. Searching for the storage record or for a free space block to accomodate the storage record starts from this focal page.

A module to supply a first estimate of the data base page for a record first searches the currency location table for the presence of the storage record if it is of type index or pointer array. If the search is unsuccessful or the storage record is not of the above types, the page in the LLB is examined. If the result is still negative, the location level DDL is checked to see if the placement option is CALC. If all of the attempts above yields an unsatisfactory result, the page number of the storage record is retrieved via the direct index.

4.2.5 Sublevel Four

number of data base pages in order to fulfill their mission. A module updates the free space list of a page to adapt a storage record. Another module is invoked for storage record deletion, modification or retrieval. Both modules ensure that the required data base page is in the LLB before they return control to the calling modules. The modules in the first class are FIND-PAGE-SPACE and FIND-STRING.

The second class of modules transfer storage records between the ELB and the LLB without any data conversion. The transfer is merely a bit string transfer as viewed from this sublevel. The modules are known as COPY-ELB-TO-LLB and COPY-LLB-TO-ELB.

There is a single module in the third class. Previously occupied storage space is returned to the page in the LLB. Merging of free space blocks takes place if there an adjacent free space block on one or both sides of the storage space to be released. The single module in this class is known as DEALLOCATE. Appendix A, Table A.6 shows the modules within this sublevel.

4.2.6 Sublevel Five

Sublevel five modules implement the basic primitives to delete, modify, retrieve and store a storage record. The module to delete a storage record from the base deallocates the storage space and updates the PCLT if the record is in the table. Modification of a storage record changes the content of the storage record but not its length. The PCLT may have to be updated for retrieving or storing a storage record. Storing a storage record requires finding a target page where the record should be placed. Then find a neighbouring page to put the record if the target page does not contain sufficient space. The space is allocated and the record is transferred from the ELB to the page. Appendix A, Table A.7 is transferred from the ELB to the page. The four modules of this sublevel are DELETE-STO-REC, MODIFY-STO-REC, RETRIEVE-STO-REC and STORE-STO-REC. Appendix A, Table A.7 shows the modules of this sublevel.

4.2.7 Summary

The modules of the location level are shown in Fig.4.2. The numbers at the left are sublevel numbers. There is a total of twenty-nine modules in this level. Modules enclosed in the same rectangular box belong to the same class.

The location level presents an abstraction to the data base file structure level. All details of the page structure, the empty spaces inside a page, the number of storage records currently in the page are irrelevent to the lower level. Modification of the storage record placement strategies does not affect the modules of the data base file structure level if the placement does not cross the boundary

```
     +----------------+
     |DELETE-STO-REC  |
     |MODIFY-STO-REC  |
     |RETRIEVE-STO-REC|
     |STORE-STO-REC   |
   5 +----------------+
---------------------------------------------------------------------------
     +-----------+
     |FIND-PAGE- |   +---------------+
     |     SPACE |   |COPY-ELB-TO-LLB|   +----------+
     |FIND-STRING|   |COPY-LLB-TO-ELB|   |DEALLOCATE|
   4 +-----------+   +---------------+   +----------+
------------------------------------------------------------------
     +----------------+
     |ALLOCATE        |
     |ALT-PLACEMENT   |
     |APPROX-PLACEMENT|
     |BOUND           |
   3 +----------------+
-----------------------------------------------------------------
                  +------------------+
                  |DELETE-DIR-INDEX  |
                  |INSERT-DIR-INDEX  |
                  |MODIFY-DIR-INDEX  |   +---------------+
     +----------+ |RETRIEVE-DIR-INDEX|   |SEARCH-STRING- |
     |CUR-LOC-OP| |SEARCH-DIR-SPACE  |   |         IN-PG |
   2 +----------+ +------------------+   +---------------+
---------------------------------------------------------------------------
                      +--------------------+
                      |DELETE-DIRY-ENTRY   |
     +--------------+ |INSERT-DIRY-ENTRY   |
     |DELETE-CUR-LOC| |MODIFT-DIRY-ENTRY   |
     |INSERT-CUR-LOC| |RETRIEVE-DIRY-ENTRY |
     |SEARCH-CUR-LOC| |SEARCH-DIRY-SPACE   |
   1 +--------------+ +--------------------+
---------------------------------------------------------------------------
       +--------+         +-------+
       |DB-SPACE|         |LL-INFO|
   0   +--------+         +-------+
---------------------------------------------------------------------------
```

Fig. 4.2 Modules in the location level

of files in which the storage record type is allowed to reside.

4.3 THE ENCODING LEVEL

The encoding level defines how data structures in the structural level are encoded as bit strings to be placed in an address space. The placing of bit strings is considered at the lower level and as such it is not discussed here. The storage record structure is superimposed on the bit string. This name coincides with that of the DSDL 1978 specifications because they share similar functions, namely, splitting a record in the logical level to a number of storage records and and encoding data item. The difference is that these functions are performed on the schema record for the DSDL 1978 definition while that of the encoding level are performed on the structural records. On top of this, index and pointer arrays are also reduced to storage records.

There are three sublevels in this level. Sublevel zero contains a module to provide information on the encoding level DDL. Both sublevels one and two contain three classes of modules each. These sublevel one and two modules implement three structural level data structures. For both sublevels one and two, their class one and two modules implement the SLR, the class two modules construct the pointer array and the class three modules implement the

```
 PUT            INTO
```

where data-name is any of the level 02 elementary items. The syntax to handle entry pairs in the ISR is as follows

```
GET  (entry-num)TH ENTRY PAIR (key-value, rec-ldbk)  FROM  ISR
PUT                                                  INTO
```

where entry-num is the entry number relative to the ISR in the encoding level buffer.

The ISRs of a set form a doubly linked list. The two ends of the list have zero pointer value. A max-min-key-value field is maintained to reduce unnecessary scanning through the ISR. The index is an ordered index. If the index is in ascending order of key values, the field contains the maximum key value. A search for an entry pair with key value greater than the maximum key value of the ISR retrieved would result in skipping over the present ISR to the next ISR to continue the search. The index is arranged in ascending key values allowing duplicate key values. The ISRs can be considered as arranged in ascending key values. If the key is declared to be in descending key values, searching starts from the end with the the largest key value. The max-min-key-value field then contains the smallest key value in the ISR.

The ISRs can be created and deleted depending on the number of elements in them and the operation to be performed on them. Certain information is required to handle an index. The ISR containing the smallest key for ascending key or the ISR containing the largest key for descending key. Second, the mapping of the index logical data base key to an index storage record. This mapping will vary as ISR becomes empty and is deleted. A head index storage record (HISR) is created to solve the two problems. It has a storage code value of zero. Its data structure is as follows:

```
01 HISR                head index storage record
   02 HISR-sdbk        storage data base key of the I-I ISR
   02 HISR-length      length of the HISR
   02 l-sdbk           storage data base key of the ISR
                       containing lowest key value
   02 h-sdbk           storage data base key of the ISR
                       containing highest key value
```

The syntax to handle data items of the HISR is

$$\begin{bmatrix}\text{GET}\\\text{PUT}\end{bmatrix} \text{ data-name } \begin{bmatrix}\text{FROM}\\\text{INTO}\end{bmatrix} \text{ HISR}$$

where data-name is any level 02 data item of the HISR.

The part of the logical data base key of the HISR-sdbk is the logical data base key of the index. The HISR contains two pointers pointing to both ISRs containing the lowest and highest key value. Changing the ordering of key from ascending to descending or vice versa does not alter the data structure of the index nor the modules managing the ISR. The index structure in the encoding level is shown in Fig. 4.4.

Appendix A, Table A.11 shows the modules used to implement index management in sublevels one and two. Sublevel one contains the modules DELETE-ISR, FIND-ISR and SPLIT-ISR. Sublevel two contains the modules CREATE-INDEX,

```
                    +----+
+-------------------|HISR|------------------+
|                   +----+                  |
|                                           |
|                                           |
V                                           V
+---+ ---> +---+ -->      --> +---+
|ISR|      |ISR|      ...     |ISR|
+---+ <--- +---+ <--      <-- +---+
```

Fig. 4.4 Encoding level storage records to implement a two level index

DELETE-INDEX, DELETE-INDEX-ENTRY, INSERT-INDEX-ENTRY, RETRIEVE-INDEX-ENTRY, RETRIEVE-FL-INDEX-ENTRY, RETRIEVE-NP-INDEX-ENTRY and SEARCH-INDEX-CONTENT. These modules provide the operations to manipulate the index structure. Duplicate key values in an index can occur depending on the specification of the schema DDL. This option affects the module to look for the appropriate index storage record. The module contains a number of options. Before describing the options, it is worthy to note that an index consists of pairs of key values and data base key value. A key is the value of a data item of a record. The first option requires a match of the key values. The second option requires a match in the logical data base key value. The third option is to find an ISR with a key value which is just greater than the input key value for insertion in ascending order. The fourth option is the counterpart of the third one, i. e. an ISR with a key value which is just smaller than the input key value for insertion in descending order. The final option is to match both the key value and the logical data base key value of an entry pair. Note that the search can be in the direction of ascending or descending key values as specified by an input parameter. Duplicate key searches are allowed but the number of duplicates must be specified. A duplicate number m is the m'th entry pair with the input key. The above three modules form the basis for building sublevel two index handling modules.

The length of the index can be varied dynamically by the index storage record deletion and splitting modules. A module to create the HISR and the first ISR is called when an owner record is created and the owner record contains a set type implemented by index.

The owner record always maintains a HISR and an ISR for an index set throughout its lifetime. Insertion of an entry pair takes place at the rear of a list of entry pairs having the same key value in the direction of search depending on the key ordering. The next and prior entry pairs are defined under the same schema so are the first and last elements of the index. For example, if the key is defined in descending order, the first entry pair contains the highest key value and the prior entry pair has key larger than or equal to the present key value.

4.3.5 Summary

There is a total of twenty-eight modules in the encoding level. They are shown in Fig. 4.5. The vertical partitioning shows the restriction of the upper sublevel modules of a class to call the lower sublevel modules of the same class. The number on the left are sublevel numbers.

Storage record is the construct of the encoding level. Different applications access and update only certain parts of the schema record. Partitioning of a SLR may cause additional processing for record creation and deletion. However, the policy improves performance for access. Tozer [Toze78] has discussed the partitioning of schema records of the DSDL 1978 proposal.

The encoding of SLRs is preferred to that of schema records for records splitting. Users do not access data values only, they would also issue DML commands to find subschema records. The data structure in the structural level includes access paths. Some access paths are not available to a subschema because some sets are not within the subschema description. Record splitting to form storage records need not include the pointer fields for the set outside the scope of the subschema. Hence encoding of structural records gives the designer a better picture of the situation. Moreover, addition of pointer fields increases the length of the schema record and so gives a better consideration for splitting.

The encoding level provides a level of abstraction to the level above. The data base administrator (DBA) concerned with the logical operations of the structural

```
                      |                 |
                      |                 |
                      |                 | +---------------+
                      | +-------------+ | |CREATE-INDEX   |
                      | |CREATE-PA    | | |DELETA-INDEX   |
                      | |DELETE-PA    | | |DELETE-INDEX-  |
   +----------------+ | |DELETE-PA-   | | |        ELEMENT|
   |CREATE-STR-REC  | | |      ELEMENT| | |INSERT-INDEX-  |
   |DELETE-STR-REC  | | |INSERT-PA-   | | |          ENTRY|
   |REPLACE-STR-FLD | | |      ELEMENT| | |RETRIEVE-INDEX-|
   |REPLACE-STR-REC | | |RETRIEVE-PA- | | |          ENTRY|
   |RETRIEVE-STR-REC| | |      ELEMENT| | |RETRIEVE-FL-   |
   +----------------+ | |RETRIEVE-FL- | | |    INDEX-ENTRY|
                      | |   PA-ELEMENT| | |RETRIEVE-NP-   |
                      | |SEARCH-PA-   | | |    INDEX-ENTRY|
                      | |      CONTENT| | |SEARCH-INDEX-  |
                      | +-------------+ | |        CONTENT|
 2                    |                 | +---------------+
----------------------+-----------------+-------------------
                      | +------------+  |     +-----------+
      +------------+  | |CREATE-PASR |  |     |DELETE-ISR |
      |XFER-DECODE |  | |SHIFT-PASR- |  |     |FIND-ISR   |
      |XFER-ENCODE |  | |     ELEMENT|  |     |SPLIT-ISR  |
 1    +------------+  | +------------+  |     +-----------+
----------------------+-----------------+-------------------
        +-------+
        |EL-INFO|
 0      +-------+
------------------------------------------------------------
```

Fig. 4.5 Modules of the encoding level

level is free from consideration of how the structural record, pointer array and index are built. Moreover, the DBA need not be concerned with the size of an index or a pointer array, whether the constructs are full or how many levels does an index have. Variation of the number of storage records to implement the three data structures does not affect the modules or the data structures above. This provides stability to the structural level to a certain extent.

The current level also presents a level of abstraction to the location level. All link fields and content fields disappeared as viewed from the level below. Only the storage data base key and the storage record length remains. The location level is concerned with these two fields to which is appended a third field of data. Interchanging the position of the content and the link fields would not affect any module of the level below. Even modifying the coding of fields of the structural level data structures would not affect the location level statistically. The effect only reveals itself dynamically. In other words no location level module is affected by the variation of size of a storage record. It is the actual placement during the execution of the location level modules that results in the change. The variation in size is limited by the page size and hence there is a limit to this change. The storage records whose size have to be varied and are existing in the data base require reorganization effort that is smaller for a decrease in storage record size than for an increase in size.

4.4 THE STRUCTURAL LEVEL

The structural level is the most complicated level of the MCDBMS. The requirement to meet the specification of CODASYL DDLC 1978 accounts for the complication. This is because DML statements that act on records at this level can be highly procedural.

Several design decisions are made concerning the structural level modules and the data structures. All modules in this level contain parameters belonging solely to this level. For example, a request to transfer a field from a record in the UWA to the schema record mentions the field under its subschema defined name. The request is transformed to one requiring transfer of an address in a UWA to a field under its SLR defined name. This transformation takes place in the higher levels. This policy reduces interdependency between levels.

4.4.1 Convention of Parameter Passing

The CONNECT, DISCONNECT, FIND RETAINING, MODIFY and STORE DML statements may involve specification of one or more set names to be operated upon. The subschema set names are converted to schema set names in the subschema level.

Passing these set names to the structural level modules requires a dynamic data structure. This is because the number of set names to be operated depends on the DML command.

Set types are compiled and represented by increasing set numbers, a module in the schema level encodes the sets involved into an accumulative set of set number n has its encoded set number 2^n. Summing the encoded set numbers gives us the accumulated set number. Passing this value to the structural level is simpler than passing a dynamic data structure of a link of list of set names or set numbers. Decoding is performed by the structural level modules.

Apart from specifying the names of sets involved in an operation, the CODASYL COBOL JOD 1976 [CODA76] specification of DML statements also allows the user to specify the ALL option of sets. The schema sets are numbered chronologically from zero onwards increasing by steps of one. Assuming that N is the value of the accumulated set number, the minimum value of N is zero. The value zero in N is used to represent the ALL option without ambiguity. This technique is also applied to an input of data identifiers of a subschema record.

It is also assumed that the syntax of the DML statements are checked during compile time. Whether or not the operations intended are within the boundary of the subschema is also checked during this time. The schema level module also supplies appropriate parameters, for example, address of a field in UWA to be transferred, before the call is made to the structural level.

4.4.2 Record and Set Representation

Sets are implemented by chain, pointer array or index, depending on the set ordering criteria. Table 4.1 shows the possible strategies for implementation.

SET ORDERING CRITERIA,INSERTION	POSSIBLE IMPLEMENTATIONS
FIRST	chain, pointer array
LAST	chain, pointer array
NEXT	chain
PRIOR	chain
SORTED BY DEFINED KEYS	chain, index

Table 4.1 Possible ways to implement the set concept

The representative data structure of a SLR is as follows

```
01  SLR                      structural level record
    02  rec-ldbk             logical data base key of the SLR
    02  rec-length           length of the SLR
    02  rec-info             other information of the SLR
    02  rec rt-1             record type of the SLR
        03  next-ldbk        ldbk of the next record of the
                             same record type rt-1
    02  set-flds             set fields
        03 set st-1          set type st-1 for owner record of
                             a set chain
            04  first-ldbk      ldbk of the first member
                                record in set st-1
            04  last-ldbk       ldbk of the last member
                                record in set st-1
        03  set st-2         set type st-2 for owner record of
                             a pointer array set
            04  PA-ldbk         ldbk of the pointer array
            04  num-of-ele      number of elements in the
                                pointer array
        03  set st-3         set type st-3 for owner record of
```

```
                        an index set
         04  IND-ldbk         ldbk of the index
      03 set st-4       set type st-4 for member record
                        of chain set
         04  prior-ldbk       ldbk of the prior member
                              record of set st-4
         04  next-ldbk        ldbk of the next member
                              record of set st-4
         04  owner-ldbk       ldbk of the owner record
                              of set st-4
      03 set st-5       set type st-5 for member record
                        of index or pointer array set
         04  owner-ldbk       ldbk of the owner record
                              of set st-5
   02 content-flds      content fields
      03 fld-name OCCURS num-of-flds TIMES
                        different field names within the
                        content fields
```

The syntax of command to handle the level 04 data items of the set type and level 03 of the record type are as follows

$$\begin{bmatrix}\text{GET}\\\text{PUT}\end{bmatrix} \text{data-name OF } \begin{matrix}\text{rec rt-i}\\\text{set st-i}\end{matrix} \begin{bmatrix}\text{FROM}\\\text{INTO}\end{bmatrix} \text{SLBj}$$

where data-name is the sublevel data items of record or set, the values of i is between one and five shown above, value of j is either 1 or 2. The two values of j are carried down for all the syntax of command for data items in the SLR. The syntax of command to handle data items of the content field is as follows

$$\begin{bmatrix}\text{GET}\\\text{PUT}\end{bmatrix} \text{fld-value OF FIELD fld-name } \begin{bmatrix}\text{FROM}\\\text{INTO}\end{bmatrix} \text{SLBj}$$

or

$$\begin{bmatrix}\text{GET}\\\text{PUT}\end{bmatrix} \text{fld-value OF FIELD NUM fld-num } \begin{bmatrix}\text{FROM}\\\text{INTO}\end{bmatrix} \text{SLBj}$$

where fld-value is the value of the field,

fld-name is the name of the field,

fld-num is the field number.

The syntax of command to handle the first two fields is

```
[GET] rec-ldbk    [FROM] SLBj
[PUT] rec-length  [INTO]
```

The data structure is only representative of a SLR because it encompasses all types of set implementation and the record is treated as both an owner record and a member record of different sets. It is not necessary that all level 03 items are present in the set fields. Any level 03 item can occur more than once if several sets are implemented by the same structure. The record type pointer is a singly linked list of all records of the same record type. It is absent when all records of the same record type is implementd by an index or a pointer array.

The data structure shows that each member record has a pointer to its owner record for every set it participates in. This field enables fast access to the owner record. It also indicates whether the member record is currently connected to a set of MANUAL insertion option. If the value of the field is zero, it is not connected.

The present level has two buffers, SLB1 and SLB2. These two buffers are of the same size, each can accomodate the largest SLR in the data base. In general, SLB1 is used as if it were the only buffer in the structural level. SLB2 is used in three occasions. First, a subschema record is always transferred to the SLB2. Second, when a record is created, it is stored in SLB2. SLB1 is used as a buffer to retrieve records so as to determine the owner records of sets with which the created record is an AUTOMATIC member. The link fields of the created record in SLB2 is updated as each automatic set owner is determined. Third, the set selection may require comparison of certain fields of the member and owner records.

The syntax to transfer the content between SLB1 and SLB2 is

$$\text{TRANSFER} \begin{bmatrix}\text{SLB1}\\ \text{SLB2}\end{bmatrix} \text{TO} \begin{bmatrix}\text{SLB2}\\ \text{SLB1}\end{bmatrix}$$

4.4.3 Record Type Directory

A directory is required for various structural representation of the SLR type occurrences. Such a directory provides a means to examine all current existing record occurrences of a particular record type. The owner record selection of a member record requires the MCDBMS to go through all owner record occurrences to see which one satifies the set criteria. A record type directory is therefore essential. This directory is accessed by modules of class three and modified by the module of class four. It is assumed to be loaded in the main memory. A copy of which is also stored at preassigned pages in the data base. The data structure of the record type directory (RTD) is as follows:

```
01  RTD              record type directory
    02  entry  OCCURS max-rec-type TIMES
        03  rt    record type
        03  rep   representation of the record type
```

```
          03  type-ldbk  logical data base key of a
                         header record, a pointer
                         array or index.
```

where rep has value 0 for chain, 1 for pointer array and 2 for index, max-rec-type is the maximum number of record type available to the run unit.

The syntax of command to access an entry of the RTD is

$$\begin{bmatrix}\text{GET}\\\text{PUT}\end{bmatrix}\ \text{(entry-num)TH ENTRY (rt, rep, type-ldbk)}\ \begin{bmatrix}\text{FROM}\\\text{INTO}\end{bmatrix}\ \text{RTD}$$

The header record for the chain representation has the following data structure

```
01  header-rec          record type header record
    02  rec-ldbk        logical data base key of the
                        header record
    02  length          length of the header record
    02  header-info     header information
    02  next-ldbk       logical data base key of the
                        next record
```

The syntax to access the level 02 data items is

$$\begin{bmatrix}\text{GET}\\\text{PUT}\end{bmatrix}\ \text{data-name}\ \begin{bmatrix}\text{FROM}\\\text{INTO}\end{bmatrix}\ \text{SLB1 HEADER REC}$$

where data-name represent the level 02 data items.

The header record is required to maintain the stability of the RTD and modifications of the chain would not affect the RTD entry. The RTD is modified only when a new schema record type is added or an existing one is deleted.

As can be seen from the header record and the data structure of the SLR, the link field for the chain representation of a record type is a singly linked list instead of a doubly linked list. When a new record is created, it is added to the first position of the chain to minimize processing.

4.4.4 Sublevel Zero

Sublevel zero supplies information on the structure of the SLRs, indices and pointer arrays. A module in this sublevel provides such information. Modules in higher levels also require information of the content fields of the SLRs. However, this information is available in the schema DDL. Another module provides service to enquiries to the content fields. Certain DML statements require knowledge of the subschema records residing in the UWA. A third module provides information to subschema records and the conversion of the the schema and subschema data identifiers.

The module to supply subschema information should theoretically reside in sublevel zero of the schema level with the schema level buffer holding a schema record. By the same token, the module to supply subschema information should reside in sublevel zero of the subschema level. The schema level buffer is unnecessary because the content fields of the SLR coincides with the schema record, thus reducing both processing time and memory requirement. The structural level modules are responsible for direct transfer between subschema records and SLRs. This adds to the complexity of the structural level but reduces processing time. The three modules in this sublevel, S-INFO, SL-INFO and SS-INFO, are shown in Appendix A, Table A.12.

4.4.5 Sublevel One

Sublevel one contains three modules responsible for the transfer of a data identifier or a variable between the structural level and the UWA. The modules are XFER-ID-FROM-UWA, XFER-ID-TO-UWA and XFER-VAR-FROM-UWA. This sublevel provides the primitives of data transferrence to be used by modules of the sublevel above. Appendix A, Table A.13 describes the modules of this sublevel.

4.4.6 Sublevel Two

Sublevel two contains two classes of modules. The first class of modules are supported by the modules of the first class in the sublevel below. They require the transfer of an entire subschema record or a number of data identifiers of the subschema record between the UWA and the SLB2. These modules are called by modules in the highest sublevel of the structural level. There are three first class modules. They are XFER-IDS-FROM-UWA, XFER-REC-FROM-UWA and XFER-REC-TO-UWA.

The second class of modules provide totally different functions from that of the first. They perform comparisons for various options of the set selection criteria as defined in the schema DDL. The owner record of a set is assumed to reside in SLB1 and the member record in SLB2. A module compares the values of certain fields of the SLB1 record with some fields of the SLB2 record or with some variables supplied by the user in the UWA. It is assumed that a key consists of a single data item only. Another module in the second class matches a key field of the subschema member record with the key field of the owner record. The owner record type has an index with the key values of all its owner record occurrences. Comparison takes place with the key values in the index instead of loading all such record occurrences and retrieving its key field for matching. This module is set aside instead of being embodied in the former module because of a number of reasons. First, no record is required in SLB1 and SLB2 for comparison. Second, it need not call any sublevel zero modules while the other module requires such a call to get the address of the field for comparison. Third, it assumes that an index for the owner record type exists while the other module does not. The algorithms for the two modules differ significantly. The two modules are MATCH-FLDS and MATCH-KEY. These two modules are supported by the second class modules of sublevel one. Appendix A, Table A.14 shows the modules of this sublevel.

4.4.7 Sublevel Three

Sublevel three contains several classes of modules. Class one and two contain three modules each. The Modules within these two classes have similar algorithms. They provide facilities to handle currency tables. It is assumed that two currency tables are present in the main memory. They are the record currency table (RCT) and the set currency table (SCT). It is further assumed that the currency tables are created during compile time. The tables contain just sufficient space to hold all currencies required during the execution time of any run unit within a subschema. All entry pairs have their second field initialized to zero.

The data structures for the currency tables are as follows:

```
01  RCT                record currency table
    02  entry OCCURS max-rec times
        03  rt-in-RCT         record type
        03  an-in-RCT         accession number

01  SCT              set  currency  table
    02  entry  OCCURS  max-set  TIMES
        03  st-in-SCT             set type
        03  ldbk-in-SCT           logical data base key of the
                                  current record of the set type
```

where max-rec and max-set are the maximum of schema record types and set types available to the run unit.

The syntax of commands for handling RCT and SCT entries are

```
[GET] (entry-num-1)TH ENTRY (rt-in-RCT, an-in-RCT) [FROM] RCT
[PUT]                                              [INTO]
```

and

```
[GET] (entry-num-2)TH ENTRY (st-in-SCT, ldbk-in-set) [FROM] SCT
[PUT]                                                [INTO]
```

where entry-num-1 and entry-num-2 are entry numbers of the entry pairs in RCT and SCT respectively. The level 02 items rt-in-RCT and an-in-RCT correspond to those of cdbkt and an of the logical data base key.

Class one modules are designed to delete, modify and retrieve an entry pair in the RCT while class two modules provide the same three functions to an entry pair in the SCT. Each record type and each set type is associated with a record type number or set type number as described in section 3.3.4. Simple sequential search takes place for accessing the entries of the currency tables. The modules DELETE-RCT-ENTRY, MODIFY-RCT-ENTRY and RETRIEVE-RCT-ENTRY belong to the first class while the modules DELETE-SCT-ENTRY, MODIFY-SCT-ENTRY and RETRIEVE-SCT-ENTRY belong to the second class.

Besides the two currency tables, there is also a currency indicator for the current record of the run unit (CRU). This indicator contains the logical data base key of the current record. The syntax to access this value is as follows

$$\begin{bmatrix}\text{GET}\\\text{PUT}\end{bmatrix}\ \text{cur-ldbk}\ \begin{bmatrix}\text{FROM}\\\text{INTO}\end{bmatrix}\ \text{CRU}$$

where cur-ldbk is the logical data base key of the current record of the run unit.

Class three contains three modules to implement search routines to trace through the set members. These modules are used for identification of the owner record under the set selection criteria. The three modules provide service to traverse the chain, index or pointer array of the owner record type. As they travel along, they call the module of the second class of sublevel two to determine whether the record selected is the owner record. Modules of the second class of sublevel two for field comparisons are available to those of the third class of sublevel three only. These three modules are REC-SEARCH-CHAIN, REC-SEARCH-IND and REC-SEARCH-PA.

Updating the records type in chain, pointer array or index are the functions performed by the modules in the fourth class of this sublevel. When a record is created or erased, the information in the record type is to be modified. The modification involves either an insetion or deletion of a record type occurrence. The modules to perform the operations of the fourth class are UPDATE-REC-CHAIN, UPDATE-REC-INDEX and UPDATE-REC-PA. The modules of class two of sublevel two are available to this class of modules only. Appendix A, Table A.15 shows the modules of the present sublevel.

4.4.8 Sublevel Four

Sublevel four contains five classes of modules. The primitives for index and pointer array manipulation have been implemented in the encoding level, but those for the set chain have not been designed. Class one modules provide service to connect a SLR to a set chain under various options specified by the schema DDL. The five modules of this class performs connection in the first, last, next, prior position and according to the sorted key ordering of the set chain. The modules are SET-CHAIN-FIRST-CON, SET-CHAIN-LAST-CON, SET-CHAIN-NEXT-CON, SET-CHAIN-PRIOR-CON and SET-CHAIN-SORTED-CON. The modules implementing the first two and the last connection types assume that the owner SLR is in SLB1. Notice that the last choice allows ordering to be in ascending or descending key values with or without duplicates. All five modules only modifies the link fields of the member records in the set chain. The member record itsely is not modified. The remaining job is performed by a higher level module.

The second class of modules exist to reduce the size of the modules in the sublevel above. They retrieve the first member and the next member in a set of chain, index or pointer array implementation and to retrieve the last member of a set chain. They are known as FIND-LAST-SET-CHAIN, RETRIEVE-FIRST-IN-SET and RETRIEVE-NEXT-IN-SET.

The third class of modules implements three selection criteria of the schema DDL. The options are selection by structural constraint, by key, and by matching some fields of an owner record with some fields of the member record or with some variables. The three modules are CONS-SET-SEL, KEY-SET-SEL and FLDS-SET-SEL. These three modules are supported by the third class modules in the sublevel below. Class three modules of sublevel three is available to class three modules of sublevel four only.

A single module, UPDATE-REC-TYPE, exists in the fourth class of this sublevel. It acts as a switch to determine which fourth class module to call in sublevel three. Any creation or erasure of a record would invoke this module to update the record type information.

The fifth class of module, UPDATE-CUR, updates the RCT and the SCT. It is called by some modules of the highest sublevel of the structural level. It allows specification of retaining part or all of the currencies from being updated. This module is supported by the modules of the first and second classes in sublevel three. Appendix A, Table A.16 shows the modules in this sublevel.

4.4.9 Sublevel Five

Sublevel five contains two classes of modules, one module in each class. Both modules act as a switch to determine which option of the schema DDL specification is required and the appropriate lower sublevel module to be called.

The module of the first class connects a member record into a set chain. It is known as SET-CHAIN-ORD-CON. The choice of lower level module to call depends on the mode of connection of a member record to a set chain as required by the schema DDL. Modules of class one of the sublevel below supports this module alone.

The other module selects an owner record of a set for a given member record occurrence. The THEN THRU option in not implemented. Olle [Olle78] doubts the effectiveness of this option and remarks that only few systems have implemented it. The modules of class three of the sublevel underneath dedicates their service solely to this module, which is known as SET-SEL. Appendix A, Table A.17 shows these two modules.

4.4.10 Sublevel Six

Sublevel six consists of three classes of modules. Class one modules concern themselves with set connections according to a specific set implementation. All three modules of class one assume that the member record to be connected resides in SLB2. To ensure that the member record is not in the set already, its owner record pointer is examined before connection. The member record has its pointer field updated after connection but it is not stored back into the data base because the member record may be required to be connected to more than one set. Storing the record after all the necessary connections are made reduces I/O. The three class one modules are SET-CHAIN-CON, SET-IND-CON and SET-PA-CON.

Classes two and three contain one module each. The class two SET-CHAIN-DISCON module simply disconnects a member record from the set chain. Disconnection of a member record from a set index and a set pointer array is not implemented at this level because they are available in the level below. Such a stategy is followed because a set chain is within the structural level view. Connections and disconnections are done externally. A set index or set pointer array is simply a data structure allowing any number of members in the structural level theoretically. Their actual implementation lies in the encoding level, so are the modules implementing them. Nevertheless, modules in any sublevel of the structural level can call these modules. The class three module SET-OWNER-SEL selects the owner record of a set given a member record. It provides the necessary parameters for calling the set selection module of sublevel five. It also checks if calling the set selection module can be avoided because the owner record may be available in the SCT. Appendix A, Table A.18 shows the modules of this sublevel.

4.4.11 Sublevel Seven

Sublevel seven contains two classes of modules. The two modules of the first class, SET-CON and SET-DISCON, deal with set connection and disconnection. The set connection module identifies the owner record of the member record within a set. It then finds out the set implementation. Next, it calls appropriate modules in the sublevel below to perform connection. Modules of classes one and three of the sublevel below are used to support this module. This module can be called under different occasions. It can be invoked purposely for connecting a member record to a set. It is called when a record occurrence is created and connection of the record to all sets in which it participates as AUTOMATIC member is needed. It can also be called if a field of a record is modified and the field value affects the set membership. The member record in SLB2 has its pointer field modified but it is not stored into the data base after the execution of this module. This is because the member record may have to be connected to more than one set. Storing the member record after modification of all its pointer fields reduces I/O and thus improving the DBMS performance.

The set disconnection module disconnects a member record from a set. It identifies its owner record from one of its pointer fields. It also identifies the set implementation strategy. Disconnection is then performed. As in set connection, all modifications are reflected in the data base except storing the member record. The basis for this policy is the same as that of the set connection module.

The class two module SET-OWNER creates an empty index or pointer array when an owner SLR is first created. The module first identifies the set implementation as defined in the structural level DDL. Then the creation of the construct takes place. Appendix A, Table A.19 contains a description of the modules of this sublevel.

The modules of this sublevel provide an abstraction of the set structure to the sublevels above. All modules in the sublevels above consider the set as a data structure with ordering but the implementation is unknown to them. Altering the set implementation does not affect the upper sublevel modules.

4.4.12 Sublevel Eight

Sublevel eight contains two classes of modules. The first class contains two modules, one to connect a member record to a number of sets and the other to disconnect a member record from a number of sets. The member record is not stored into the data base after modification of its pointer fields because it may be further modified by the modules manipulating it. It remains in SLB2 after connection is performed. An output parameter of both modules reflects the result of connection or disconnection. When the connection is successful, the value of the parameter is zero. Otherwise it contains the accumulative set number of the sets in which the member record has already been connected or disconnected for a request for connection or disconnection respectively. The modules in this class are ACC-SET-CON and ACC-SET-DISCON.

The second class of module ACC-SET-OWNER calls the same class of module in the sublevel below. It provides creation of structural level set constructs to the set types in which a new record occurrence participates as owner. It is therefore called when a new record occurrence is created. Appendix A, Table A.20 shows the modules of this sublevel.

4.4.13 Sublevels Nine and Ten

Sublevels nine and ten contain one module each. They are both used to implement the module for record erasure in sublevel eleven. The module of sublevel nine erases one SLR from the data base. The record to be erased has no member record currently. When a record is erased, all the set implementational constructs, empty pointer arrays and indices, are erased as well. This module also disconnects the record from all sets in which it is currently a member. Therefore the module to disconnect a record from a number of sets is called. This explains why this module is one sublevel higher then the accumulative set disconnection module.

Sublevel ten is a recursive procedure to erase a record. The module traverses every set in which the record is an owner. If the set is non-empty, each member record is to be erased. Each member record then becomes the input parameter of the recursive procedure and the procedure calls itself again. If the record contains no member records, the module in the sublevel in the sublevel immediately below is called to erase the record. The module takes care of the ERASE ALL option as specified in Codasyl JOD 1976 specification. The module ERASE-REC is in sublevel nine while the module RECURSIVE-ERASE is in sublevel ten. Appendix A, Table A.21 shows the modules in these two sublevels.

4.4.14 Sublevel Eleven

Sublevel eleven is the highest sublevel of the structural level. It implements the DML statement on the structural level basis. The module names coincide with the module commands. The DML statements implemented have been listed in Table 3.1 of Section 3.2. Appendix A, Table A.22 describes the modules of this sublevel.

It is assumed that the current record of the run unit is in SLB1 both in the beginning and at the end of the command, except when an error occurs.

The member record is moved to SLB2 in modules to implement the CONNECT and DISCONNECT statements before calling lower level modules. The member record is stored into the data base after all pointer fields are updated in the member record.

Only two options of the ERASE statement is implemented. They are ERASE and ERASE ALL MEMBERS. The first option requires that no member record is currently present in any of the sets in which the record to be erased is an owner. The other option allows the presence of member records but all member records will be erased. The other two options specified in the CODASYL COBOL JOD 1976 depend on the removal class. Olle [Olle78] criticizes the other two options to be semantically complex and expects them to disappear sooner or later. They are therefore not implemented in the MCDBMS.

No cycle of sets is allowed. The two cases shown in Fig. 4.6 are forbidden. This is due to the creation of an infinite loop with the recursive erasure module in sublevel ten. It is also difficult to determine whether other member records of C should be erased or not.

```
    +-------+                  +-------+
    |       |                  |       |
    |       V                  |       V
    |   +--------+             |   +--------+
    |   |   A    |             |   |   A    |
    |   +--------+             |   +--------+
    |       |                  |       |
    |       V                  |       V
    |   +--------+             |   +--------+
    |   |   B    |             |   |   B    |
    |   +--------+             |   +--------+
    |       |                  |       |
    +-------+                  |       V
                               |   +--------+
                               |   |   C    |
                               |   +--------+
                               |       |
                               +-------+

          case 1                     case 2
```

Fig. 4.6 Forbidden set relations of the schema record types.

The seven formats of the FIND statement are quite straightforward. They to modify a record may trigger a large amount of I/O. The module may modify the set membership of a record or the contents of the record or both. If a data item is used as the CALC key, it cannot be modified. The record must be erased and recreated.

The module to implement the STORE statement stores a record into the data base. Associated with the storage is the creation of all pointer arrays and indices for set types of which the record type is the owner. The created record is also connected to all the set occurrences in which it participates as an AUTOMATIC member. A new record occurrence of the record type is entered into the record type information apart from updating the currency inducators involved.

4.4.15 Summary

Fig. 4.7 and 4.8 show the modules of the structural level. The numbers on the left are sublevel numbers. There is a total of sixty-five modules in this level. The large number of sublevels in this level is partially attributed to the highly procedural DML statements. Sublevels eight to eleven can be considered as breaking the DML statements into smaller units that can be handled conveniently. Ten modules are specifically dedicated to implement the set selection criteria. Connection of a record to a set also requires ten modules.

This level provides a level of abstraction to the schema level. If the structural representation of a schema record

```
       +-----------+
       |ACCEPT     |
       |CONNECT    |
       |DISCONNECT |
       |ERASE      |
       |FIND1      |
       |FIND2      |
       |FIND3      |
       |FIND4      |
       |FIND5      |
       |FIND6      |
       |FIND7      |
       |GET        |
       |MODIFY     |
       |STORE      |
 11    +-----------+
-------------------------------------------------------------
       +----------------+
       |RECURSIVE-ERASE|
 10    +----------------+
-------------------------------------------------------------
       +----------+
       |ERASE-REC|
  9    +----------+
-------------------------------------------------------------
       +---------------+
       |ACC-SET-CON    |   +--------------+
       |ACC-SET-DISCON|   |ACC-SET-OWNER|
  8    +---------------+   +--------------+
-------------------------------------------------------------
       +-----------+
       |SET-CON    |   +----------+
       |SET-DISCON|   |SET-OWNER|
  7    +-----------+   +----------+
-------------------------------------------------------------
       +--------------+
       |SET-CHAIN-CON|   +-----------+   +-----------+
       |SET-IND-CON  |   |SET-CHAIN-|   |SET-OWNER-|
       |SET-PA-CON   |   |    DISCON|   |       SEL|
  6    +--------------+   +-----------+   +-----------+
-------------------------------------------------------------
       +-----------+
       |SET-CHAIN-|   +--------+
       |   ORD-CON|   |SET-SEL|
  5    +-----------+   +--------+
-------------------------------------------------------------
```

Fig. 4.7 Modules of sublevles 5 to 11 of the structural level

or set changes, the schema DDL is unaffected. Apart from

```
 +-----------+
 |SET-CHAIN- |
 | FIRST-CON | +-----------+ +-----------+
 |SET-CHAIN- | |FIND-LAST- | |CONS-SET-  |
 |  LAST-CON | | SET-CHAIN | |        SEL| +--------+
 |SET-CHAIN- | |RETRIEVE-FI| |FLDS-SET-  | |UPDATE- | +--------+
 |  NEXT-CON | | RST-IN-SET| |        SEL| |    REC-| |UPDATE- |
 |SET-CHAIN- | |RETRIEVE-NE| |KEY-SET-   | |    TYPE| |     CUR|
 | PRIOR-CON | | XT-IN-SET | |        SEL| +--------+ +--------+
 |SET-CHAIN- | +-----------+ +-----------+
 |SORTED-CON |
4+-----------+
-------------------------------------------------------------
   +-----------+ +-----------+ +----------+ +----------+
   |DELETE-RCT | |DELETE-SCT | |REC-SEARC | |UPDATE-RE |
   |     -ENTRY| |     -ENTRY| |  H-CHAIN | |  C-CHAIN |
   |MODIFY-RCT | |MODIFY-SCT | |REC-SEARC | |UPDATE-RE |
   |     -ENTRY| |     -ENTRY| |  H-INDEX | |  C-INDEX |
   |RETRIEVE-  | |RETRIEVE-  | |REC-SEARC | |UPDATE-RE |
   | RCT-ENTRY | | SCT-ENTRY | |      H-PA| |      C-PA|
3  +-----------+ +-----------+ +----------+ +----------+
-------------------------------------------------------------
   +----------+
   |XFER-IDS- |
   | FROM-UWA | +-----------+
   |XFER-REC- | |MATCH-FLDS |
   | FROM-UWA | |MATCH-KEY  |
   |XFER-REC- | +-----------+
   |    TO-UWA|
2  +----------+
-------------------------------------------------------------
   +-------------+
   |XFER-ID-FROM |  +----------+
   |         -UWA|  |XFER-VAR- |
   |XFER-ID-TO-  |  | FROM-UWA |
   |         -UWA|  +----------+
1  +-------------+
-------------------------------------------------------------
   +-------+     +--------+    +--------+
   |S-INFO |     |SL-INFO |    |SS-INFO |
0  +-------+     +--------+    +--------+
-------------------------------------------------------------
```

Fig. 4.8 Modules of sublevels 0 to 4 of the structural level

abstraction this level also provides stability to the schema level module.

4.5 THE SCHEMA AND THE SUBSCHEMA LEVEL

The schema level includes data types of set and record. It provides a level of abstraction to the DBA. The DBA need not be concerned with the implementation of the data types. A global view of the data types is provided for subschema definition, monitoring, reorganization and recovery purposes. The schema DDL is unified language to define data types of the entire data base. A very different subschema DDL can be used to describe the data relationships in the subschema catered to the need of the particular application. However, this would not affect the schema DDL as far as the data and its relationships are concerned. If the data representation changes, only the interface need be changed in encoding and decoding between transfers

The schema level contains the module DMLP to transfer the sets and data identifiers into accumulative set number and accumulative field number for the structural level. The module also classifies the data management requests to call the appropriate module of sublevel eleven of the level below. Appropriate parameters are supplied from the commands.

The subschema level has data types known as set and record. Most operations are based on these data types.

There can be many subschemas for a single schema. This level provides an abstraction to the user because he need not be concerned with complex data relationships of the entire data base. The module in this level merely transforms an operation from the user's view to the DBA's view of the schema level abstract data types. The module transforms subschema data identifier, record and set names to their schema correspondences. The module is known as SS-TO-S-XFORM. Appendix A, Table A.23 shows the modules of levels five and six.

4.6 GENERAL REMARKS ON THE MCDBMS MODEL

There are three modules in the data base file structure level, thirty in the location level, twenty-six in the encoding level, sixty-five in the structural level and one for both the schema and the subschema level, resulting in a total of one hundred and twenty-six modules. In the design process, the lower level modules are implemented bottom up because the requirements to be met are well specified. The structural level modules are implemented as a result of iterations. The lower sublevel modules provide the primitives to the basic procedures so as to implement the highly procedural DML commands. The commands also determines what particular type of modules are required, such as the modules dedicated for set selection.

The advantages of levels of abstraction have been applied to the MCDBMS model from both the stability and the abstraction point of view. The stability concept is important because the software and hardware technologies advances by leaps and bounds today. In particular, the subschema, the schema and the data structures of the structural level are independent of physical devices on the which the data base resides.

Storage records are stored page by page. This renders reconstruction an easy task. Free space can be gathered together and storage records can be closely packed on a page by page basis. Such page by page relocation would not affect the placement mapping algorithms.

Theoretically speaking, introducing the DBMS requires no modification of the operating system. Practically speaking, however, the CPU scheduling module may have to be modified to give higher priority to the DBMS or to allot a greater time slice to the DBMS. For an operating system supporting the multi-user environment, the single-user DBMS can be considered as one of the application programs. Security measures as applied to the general file system would suffice if the other application programs are not allowed to access data base files.

With a multi-level structure, accesses can be slowed down by subroutines calling subroutines together with a number of table look-ups. However, this is the price paid to achieve data independence. This is precisely the reason for CODASYL DBTG 71 to be transformed to CODASYL DDD 78 with the introduction of a Data Storage Description Language.

An interesting point concerning the design process is worth mentioning. After specifying the function of each level, the modules of the MCDBMS is built bottom-up level by level. Within each level, the interface of the top sublevel with the level above is specified. The data structure required for the level is then designed. Modules starting from sublevel zero are then built upwards to satisfy the requirements of the top sublevel modules. This method works well with the encoding level. The sublevel modules of the encoding level encounter a difficulty in meeting the specifications of the top sublevel. Some new data structures have to be added and modules manipulating them are built from sublevel one. The new data structures are the direct index and the direct index directory. An iteration is required to build up the modules instead of a strictly bottom-up implementation. A number of iterations is required to implement the sublevels of the structural level. The design of modules to implement the set selection criteria consists of a detailed top-down specification of the interface and a bottom-up synthesis of the modules to meet the need. The ERASE module is designed top-down up to the ERASE-REC module.

With the MCDBMS design, the more the number of sublevels within a level, the more is the number of iterations performed in the implementation. However, the general trend of top-down design for the interfaces and the bottom-up implementation with respect to levels is valid for the MCDBMS model. The design process verifies that the level of abstraction approach is a powerful tool to reduce complexities, to reduce interdependencies between modules and to enforce a clear interface for the highest sublevels to meet the requirements. The class concept also provides good partitioning to reduce considerations of the modules of a sublevel. The total number of modules in the structural level is greater than the total number of modules in all other levels. The time spent in designing the structural level modules is also greater than that spent in designing modules of all other levels.

The MCDBMS design follows the hierarchial structuring approach. The modules of the level above can only call the modules of the level immediately below it. This strict structuring policy is not followed in the sublevels of the same level. A module in a sublevel can call modules one or more sublevels below it, provided they are within the same level. Strict structuring can be achieved by putting modules in between sublevels. However, the sole function of such modules is to make the system look strictly hierarchial. The inefficiency caused by their presence would make one hesitate to put them into the MCDBMS.

Chapter V
ROLLBACK RECOVERY SUBSYSTEM

This chapter describes a fast rollback recovery subsystem. The goals, approach and assumptions of the recovery system are discussed. This is followed by a description of the modified rollback strategy. Two implementation strategies are listed and compared. The modules of the rollback subsystem are described. Finally the rollback recovery system is evaluated and its limitations are listed.

5.1 BACKGROUND

5.1.1 Goals

The recovery aspect dealt with follows logically from the requirements of the 1976 CODASYL specifications of the DML commands. When a DML command cannot proceed for some reason, all changes made to the data base since the start of the command must be undone. This type of command rollback is one of the goals of the recovery subsystem.

Expanding the command rollback concept would result in run unit rollback. The run unit may have executed a number of DML commands successfully. These commands could result in modification of part of the data base. The run unit comes to a point where a DML command fails to function as expected. Several alternate paths are tried but the efforts seems futile. The run unit cannot proceed any longer. A run unit rollback is required to undo all changes made to the data base by the successfully executed DML commands from the start of the run unit. Run unit rollback is therefore another goal of the subsystem.

The conventional method of command and run unit rollback is to store the before images into a before image journal tape as described in section 2.4.5. Rollback requires rewinding the tape and replacing the data base pages by the before image pages. This process can be speeded up by storing the before image pages in a random access storage device instead of on a tape. This is known as "dumping on the fly" [Toze77]. The time for command rollback may be tolerable but run unit rollback could take unacceptably long time. The goal of the present rollback subsystem is to design a fast rollback subsystem for both the DML command and the run unit.

5.1.2 Approach

The careful replacement technique [Gior76], [Verh77], [Verh78] is chosen as the starting point for the design of the rollback recovery subsystem. Careful replacement is applied to physical storage so as to keep the update sequence as safe as possible. The probability of having an inconsistent file is reduced to a minimum. The details of the careful replacement scheme have been described in section 2.4.6.

If all updates from a DML command can be reflected in the data base by complementing a bit, fast command rollback can be achieved by leaving the bit as before. If a DML command executes successfully, the bit is complemented at the termination of the command.

By the same token, if the run unit terminates successfully, a bit is simply complemented to put all changes made by the run unit into effect. This bit flipping technique provides a fast way of recovery rollback.

However, any update to the data base is not limited to a single page but would affect a number of pages. These pages constitute a logical unit of recovery, either for command or run unit rollback. The number of pages to be modified by each command is unpredictable. It is necessary to extend and modify the concept of careful replacement from the physical scope to the logical scope of application.

5.1.3 Assumptions

The goals of the proposed rollback subsystem have been presented. The subsystem, however, requires an environment within which it can operate. The environment assumed is the MCDBMS and the file system described in the two preceeding chapters. The MCDBMS is assumed to be able to detect the error condition so as to initiate rollback recovery. The errors that the rollback recovery subsystem is tackling are logical type of errors caused by the execution of the DML commands. Failure of the MCDBMS to detect an error condition would not give rise to rollback recovery.

Other techniques are assumed to be available to handle other types of errors, such as a disk head crash. The rollback recovery subsystem is not designed to tackle such hardware failures.

The MCDBMS and the file system assumed requires extension to adapt recovery modules. It is also assumed that there are at least two file buffers in the PFS. Each buffer can be of the size of a physical block or of a unit of I/O transfer.

The description of the PFS described in section 3.4 allows non-contigious allocation of file space by chained blocks or file maps. Verhofstad [Verh78] remarks that a file implemented in chain blocks would propagate replacements if the careful replacement technique is used. If a block is to be replaced, the link pointing to it has to be updated. Careful replacement requires that the link is updated in a copy of the block containing the link. Replacing a block would result in replacing all preceeding blocks in the link list of blocks of a file. Chained block file structure are therefore inhibited.

The file system is assumed to have file directory similar to that of OS 360/370, as described by Madnick and Donovan[1]. Each storage device contains its own device file directory (DFD). The DFD is a hierarchial directory for all files stored in the storage devices. A special storage device contains its own DFD with a catalog entry. Each catalog entry contains a file identifier and the corresponding storage device number of the device on which the file resides. The Active File Table entry also contains the storage device number (dev#) where the file is stored. The special storage device with the catalog entry is defined as device zero, DEV(0). The other storage devices holding data base files are identified as devices one to N, or abbrivated by DEV(1), ..., DEV(N).

A further restriction is that there is no pointer from a child page to its parent page. This is because if a child page needs careful replacement, its parent needs to be carefully replaced as well. A copy of the parent page is made to update the pointer to the new child page. All other child pages of the updated parent would require a copy to be made so as to update their parent pointers to point to the new parent page. This would result in replacement propagation as described before. The restriction is therefore set to avoid the disaster.

[1] Madnick, S.E.; and Donovan, J.J. Operating Systems, McGraw-Hill, N.Y., USA, 1974, pp. 520-523.

With a separate DFD for each storage device, an inherent restriction is that no file can cross storage device boundaries. In the event that a file is too large for a single storage device, it is split into two or more files.

5.2 GENERAL CONSIDERATIONS

This section lists the data structures used for recovery. It also describes the recovery strategy and two alternate implementations. The pros and cons of the alternatives are discussed. Further data structures to tackle problems such as free space management and distinguishing whether a page is a copy of the original one are described.

5.2.1 Data Structures for Rollback Recovery

5.2.1.1 Physical blocks

Each storage device has a hierarchial DFD. The root page of the DFD is known as the device head file directory (DHFD). According to the careful replacement policy, when a page is modified, a copy of the page is made. Modification takes place on the copy. The directory page pointing to the original page also requires a copy to point to the new page. Therefore in a file hierarchy all ancestors require careful replacement, including the DHFD page.

In order to maintain the stability of the catalog of DEV(0), the catalog is not replaced. The entries in the catalog only contains the device numbers corresponding to the file identifiers. The exact location of the file is found in the DFD of the device. As long as replacement takes place in the same storage device, the catalog need not be modified.

In the MCDBMS one physical block is used to hold one data base page. Careful replacement gives rise to copies of the DHFD page. A method is needed to indicate the location of the DHFD page. Viewed from the PFS, a storage device is composed of consecutive blocks. The first five physical blocks are reserved for careful replacement of the DHFD page. The blocks are numbered from zero to four.

Blocks two to four are used to hold different versions of the DHFD page. One block holds the DHFD page before the start of the executing run unit. Another block holds the DHFD page after the completion of the most recent DML command but before the execution of the present one. The third one holds the DHFD page during the execution of the current DML command. Block zero is used as a pointer to the block containing the DHFD page before the start of the executing run unit. Block one is another pointer to point to the DHFD page of the most recently completed DML command.

```
 BLK(0) BLK(1) BLK(2) BLK(3) BLK(4)
+-----------------------------------+
|333333|444444|      |      |      |
|333333|444444|      |      |      |
|333333|444444| DHFD | DHFD | DHFD |
|333333|444444| page | page | page |
|333333|444444|      |      |      |
|333333|444444|      |      |      |
+-----------------------------------+
```

BLK(0) -	points to the DHFD block of the most recent run unit
BLK(1) -	points to the DHFD block of the most recent DML command
BLK(2) -	DHFD page of the current DML command
BLK(3) -	DHFD page of the most recent run unit
BLK(4) -	DHFD page of the most recent DML command

Fig. 5.1 An example of the first five physical blocks of a storage device

As an example consider the situation in Fig. 5.1. Block zero contains the repeated value three. This value indicates that block three contains the DHFD page of the most recent run unit. Block one contains the repeated value four. This value indicates that block four contains the DHFD page of the most recent DML command. The block that is not being pointed to is block number two and it contains the DHFD page of the current DML command.

The syntax of command to store or load a physical block is as follows:

```
LOAD BLK(i) OF DEV(j) INTO BUF(k)
STORE BUF(k) INTO BLK(i) OF DEV(j)
```

where

BLK(i) is block number i, i is the physical block number of the storage device,

DEV(j) is the storage device j, the value of j ranges from 0 to N,

BUF(k) is the file buffer number k, the value of k is either 1 or 2.

The data structure for physical blocks zero and one of storage devices 0 to N is as follows:

```
01  PHYBLK                          physical block
    02  VALUE p OCCURS M TIMES
```

where M is the maximum number of integers a physical block can hold.

in other words, the entire block is filled with repeated numbers of the same value. This value is the physical block number where the DHFD page resides.

The syntax of commands to retrieve or store the value of p when blocks zero and one are loaded into the file buffer are as follows:

```
GET VALUE p FROM BUF(k)
FILL BUF(k) WITH VALUE p
```

Note that blocks zero and one have to be in the buffer only during the retrieval operation.

5.2.1.2 Device Directory (DD)

Two arrays are used in the file system to indicate the values of blocks zero and one of each storage device containing data base files. The first array is the Run unit Device Directory array (RDD) of the most recent run unit.

The second array is the DML Command Device Directory array (CDD) of the most of the most recent DML command. RDD(i) contains the value stored in block zero of device i. Similarly CDD(j) contains the value stored in block one of device j. Both arrays have N+1 elements for devices zero to N.

The two arrays reduce I/O considerably. Consider the case when these two arrays are not included in the rollback recovery subsystem. In order to locate the block number containing the DHFD page of the current DML command, blocks zero and one are loaded into the recovery subsystem. Values in the two blocks are extracted. As in the example of Fig. 5.1, if block zero contains the value three and block one contains the value four, the remaining block is block two. Block two therefore contains the DHFD page of the current DML command. However, the RDD and the CDD arrays contain the values which blocks one and two contain. Examining the appropriate elements of these two arrays would also give the desired remaining block number. These arrays can be made to occupy only a small amount of the main memory space. Each array element only requires a minimum of two bits to identify the three blocks involved.

When a DML command finishes execution, each device with its data base page modified requires updating their block one value. The value of block one is switched to point to the block containing the new DHFD page of the terminating

DML command. A Modified Value array (MV) for devices 0 to N is used to indicate the devices that have been modified within a DML command. The elements of the MV array are initialized to zero at the beginning of a DML command. Every time device j is modified, MV(j) is incremented by one. At command termination time, all storage devices with their corresponding MV array element greater than zero has their block one value switched to point to the new DHFD block.

Similarly, an array is required to indicate the devices which have been modified during the lifetime of the run unit. The array is known as Cumulative Modified Value array (CMV). The elements of the CMV array are initialized to zero when a run unit starts. Every time when a block of a device is modified, the corresponding CMV element is incremented by one together with the MV array element. When the run unit terminates, all storage devices with the corresponding CMV element value greater than zero have their block zero modified to point to the new DHFD block. Both MV and CMV arrays have N+1 elements for devices 0 to N. When a run unit terminates, block zero of a number of storage devices may require updating to point to the new DHFD block. If a system crash occurs during this process of update and the content of main memory is lost, it is difficult to determine which storage devices have their block zeros updated and which ones do not. The data base would be left in an inconsistent state.

The problem described in the preceeding paragraph can be tackled by introducing some data structures into device zero. Blocks two to four of device zero are used to hold various versions of the device directory. One of the three blocks holds the RDD array for the most recent run unit. Another block holds the CDD array of the most recent DML command. The block left is used for switching. Block zero contains the block number of the block holding the RDD array. Block one contains the block number of the block holding the CDD array. Fig. 5.2 shows an example of the

```
BLK(0) BLK(1) BLK(2) BLK(3) BLK(4)
+---------------------------------+
|444444|222222|     |      |     |
|444444|222222| CDD | used | RDD |
|444444|222222|     | for  |     |
|444444|222222|     |switch|     |
|444444|222222|array| ing  |array|
|444444|222222|     |      |     |
+---------------------------------+
```

BLK(0) - pointer to the RDD block
BLK(1) - pointer to the CDD block
BLK(2) - block containing the CDD array
BLK(3) - block used for switching
BLK(4) - block containing the RDD array

Fig. 5.2 An example of the first five blocks of storage device zero

first five blocks of storage device zero.

Basically, when a command terminates, a module switches block one to point to the new DHFD block for devices with MV element greater than zero. After the update is completed, the new value of CDD array is stored into the buffer used for switching in device zero. Block one is then set to point to the new block containing the up-to-date CDD values. If an error occurs during updates of block one of various devices, the CDD values pointed to by block one of device zero are retrieved and used to reset the block one values of various devices for command rollback.

The syntax of commands for data transfer between the RDD, CDD arrays and the DD in the file buffer are as follows:

$$\text{TRANSFER DD IN BUF}(k) \text{ TO } \begin{bmatrix}\text{CDD}\\ \text{RDD}\end{bmatrix}$$

$$\text{TRANSFER } \begin{bmatrix}\text{CDD}\\ \text{RDD}\end{bmatrix} \text{ TO BUF}(k)$$

where CDD and RDD resides in main memory, and the transfer is the entire length of the DD directory. The array lengths of DD, CDD and RDD are the same.

In addition, block five of device zero is used to hold the currency location table (CLT) before a DML command is executed. The syntax of command for CLT transfer is as follows:

$$\begin{bmatrix}\text{PUT}\\ \text{GET}\end{bmatrix} \text{ CLT } \begin{bmatrix}\text{INTO}\\ \text{FROM}\end{bmatrix} \text{ BUF}(k)$$

the PUT command copies the entire CLT into buffer k and the GET command copies the CLT stored in the buffer into the CLT in main memory. We assume that a CLT can be stored by a physical block. If the size of a CLT is larger than a physical block, this command can easily be expanded to satisfy the requirement.

Block six of device zero is used to hold the content of a data base record or a variable in the UWA if the DML command requests data to be transferred to the UWA. The data structure of block six of device zero is as shown below.

```
01  PHYBLK6        physical block six
    02  addr       starting address of the data to be
                   transferred to the UWA
    02  len        length of data to be transmitted
    02  content    content of data to be transmitted
```

Since a DML command transfers a record or a variable at a time, block six only needs to have a maximum length of one subschema record.

The syntax of command for transferring data from the UWA to the buffer is as follows

```
TRANSFER ADDR addr LENGTH len FROM UWA TO BUF(k)
```

where addr and len are the address and length of data to be transferred in the UWA. Note that the values of addr and len are also stored on BUF(k) together with the content specified by them.

In order to reload the data back to the UWA, the following commands are used.

```
GET ADDR addr LENGTH len FROM BUF(k)
TRANSFER CONTENT OF BUF(k) TO ADDR
    addr LENGTH len OF UWA
```

5.2.2 Implementation Strategies

5.2.2.1 Physical implementation

The careful replacement scheme requires that every update is performed on a copy of the component which replaces the original only if the update is successful. In order to achieve careful replacement on a number of pages modified instead of on a single page, a combination of the careful replacement and differential file techniques is used. The differential file replaces the original by careful replacement. The unit for the differential file is a DML command or a run unit.

Under the careful replacement scheme, whenever a page is to be modified, a copy is made and modification takes place on the copy. This strategy is modified such that only pages that are not modified within the same DML command needs a copy for update. Pages that have been modified in the same DML command would result in update in place on the modified copies.

Consider the device file directory as shown in Fig. 5.3. The diagram shows a binary tree structure. In reality the tree is n-ary, where n is the number of entries that can be held in a directory page. For purpose of illustration the file hierarchy is assumed binary. Block A is the DHFD block. Assume that when a DML command starts executing, the data base page in block D is to be modified. A copy of block D is created and modification takes place on the copy.

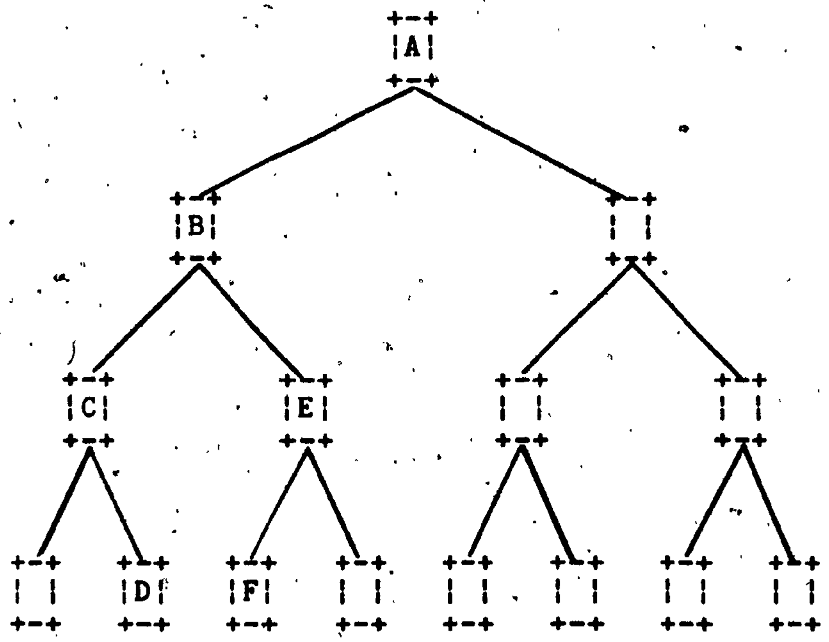


Fig. 5.3 A device file directory as a binary tree

A copy of block C is then created. The appropriate pointer entry is updated to point to the copy of block D. Similarly a copy of block B is created to point to the new block C. Since the DHFD block is itself a copy, update in place on block A is made to point to the new block B. Copies are made from leaf to root according to the leaf first rule.

If block D is to be modified again within the same DML command, the operation is performed on the copy of block D previously used in the DML command. Update in place occurs on the copy just as in the differential file technique. Blocks A, B and C are not affected.

If block F needs modification within the same DML command, a copy is created and modification takes place on the copy. A copy of block E is then created and modified to point to new block F. The copy of block B, which was created before, is updated in place to point to the new block E.

In this example it can be seen that if strictly careful replacement is to be used, a second update to block D within the same DML command would result in the creation of copies of blocks D, C, B and A as well. However, the modified scheme performs update in place on block D only. Blocks C, B and A are not affected. An update to block F would result in duplicates of blocks F, E, B and A in strict careful replacement while only copies of blocks F and E are created in the modified scheme. The modified careful replacement scheme reduces updates. It also explains the data structures of blocks two to four for storing the DHFD pages of a device. Only one block is used to store the DHFD of an executing DML command. This block is always updated in place within the DML command.

When a DML command terminates the MV array is checked. For all the devices with the MV element value non-zero, the pointer in block one is switched to point to the DHFD block of the terminating command. The corresponding CDD elements are also updated. After updating block one of the modified devices, the value of CDD is stored in device zero. Block one of device zero is also switched to point to the block containing the new CDD array.

The command termination procedure described above produces careful replacement of a DML command as a logical unit. This means that all blocks modified by the command are carefully replaced. The careful replacement of blocks modified by a DML command takes place in two stages. First, the DHFD block of each device with modified blocks are carefully replaced by switching block one of the device to point to the new DHFD block. Second, the new block one values are stored in the CDD. The new CDD is stored in device zero and block one of device zero is switched to point to the new CDD block. This last switching results in careful replacement of the blocks modified by the DML command as a logical unit.

Note that the modified careful replacement scheme can also be applied to a run unit instead of a DML command. The run unit termination procedure is similar to that of the command termination. Updates are made based on the CMV array instead of the MV array. Block zero is switched instead of block one. Finally, block zero of device zero is switched to point to the block containing the new RDD array.

5.2.2.2 Logical implementation

There are basically two ways to implement the modified careful replacement scheme. The previous section describes an implementation that works in the physical level. Physical blocks of storage devices are used for switching.

An alternative to the above is to perform modified careful replacement in the DBMS level as described below.

The file size of every data base file is extended. A request to update a data base page results in transforming the request to write onto an empty page of the same file so as to create a modified duplicate. Extra space in the file is used as temporary pages for modified duplicates. This accounts for one of the two reasons for extending the length of a data base file.

Another reason to extend the file size is to allow space for three tables. An example illustrates the situation best. Consider the case when page 6 is to be modified. The modified duplicate copy of page 6 may reside in page 22. When the run unit terminates a table is required to keep the reference of page 6 to page 22. This is because the CALC algorithms map data base records to absolute page numbers in the MCDBMS. If page 6 is the absolute page number, the operating system has to be directed to load page 22 instead of page 6 if a record in absolute page number 6 is required in the next run unit. A table is necessary to consolidate all duplicate pages with respect to the absolute page number up to the last run unit.

Just as blocks two to four in storage devices for the DHFD page, three tables are needed in the data base files for the logical implementation; one is for the page correspondence for the most recent run unit, another for the most recent DML command, and a third for the executing DML command. Each entry of the tables contains two integers. One integer is the absolute page number. The other integer is the page number where the page is actually stored. Two more pages are needed to act as blocks zero and one to point to the appropriate tables.

5.2.2.3 Comparison of the two implementations

The physical implementation lays the burden of the modified careful replacement on the file system while the logical implementation lays the burden on the MCDBMS. The physical implementation requires addition of recovery modules into the filing system while the logical one does not. The LFS has to be able to distinguish between a database file and a non-database file for appropriate transfer of control. If the MCDBMS is delivered as a package together with its recovery facilities, the logical implementation is preferred. The technical support group of the companies would not like to have any modules added into their file system in order to adapt the new package.

On the other hand, the logical implementation has two efficiency drawbacks as compared with the physical one. First, extra searching is involved in the MCDBMS level for the appropriate page for loading. Searching of a physical block takes the same number of accesses in the file system for both implementations, assuming, a tree device file directory. However, the logical implementation adds onto it searches of which page correspond to the absolute page number for loading or storing. This search may require loading of a number of pages. Each page to be retrieved results in searching through the file hierarchy for the appropriate physical block. Loading of a logical page therefore gives rise to a number of accesses of file directory blocks. The search increase processing time and the loading increases I/O traffic.

Secondly, considerable storage space can be utilized to store the extended data base files. This extension is partly due to the storage of the three tables mentioned above and partly due to the necessity of pages for modified duplicates. A simple example is sufficient to illustrate the situation.

Assume that a storage device has storage capacity of 10000 units. Fifty data base files, each requiring 100 units for storing its contents, are stored in the device. These files would occupy a total of 5000 units. Suppose one run unit would modify up to a maximum of 100 units of the data base. In order to implement careful replacement, each file requires an extension of 100 units. This increase does not include the space for the three tables. The increase takes place in each file because a run unit can perform its updates all on a single file or scatter its updates over a number of files. Fifty files occupying 200 units each would use up the entire storage device.

In the physical implementation only 100 units of storage space is required to hold the modified duplicates of a run unit. This results in a total saving of 49% of the device space. This illustrative example points out the space utilization of the two implementations.

Subsequent design of the rollback recovery system is based on the physical implementation approach. This is because of the above drawbacks of the logical implementation. The inefficiency in processing and I/O transfer outweighs the modification to the file system from an economic point of view.

5.2.3 Time Stamp and Free Space Management

This section describes how the recovery system distinguishes pages which are already a duplicate and which are not. The problem of free space management is also discussed.

5.2.3.1 Time stamp

Modified careful replacement of directory and data base pages requires a means to distinguish whether the page to be updated has been modified or not within the executing DML command. A time stamp is employed to serve the purpose. Each DFD page and each data base page contains a time stamp.

The time stamp indicates the date and time a page is last modified (dtm).

Besides the time stamp the date and time at which a DML command starts (dtcs) and a run unit starts (dtrs) are also recorded. If the dtm of a page to be modified is greater than that of dtcs, the page is already a duplicate and update in place can be performed. If dtm is smaller than both dtcs and dtrs, a duplicate is created for update. If dtm is between dtcs and dtrs, the page has been modified by a previous DML command but not by the present one. It is necessary to create a copy for the present command. Otherwise command rollback is impossible because the before image is overwritten.

Both the DFD page and the data base page contains a page header. The time stamp is stored in the page header. The syntax of command to manipulate the time stamp is as follows:

```
[GET] dtm [FROM] PG-HDER OF BUF(j).
[PUT]     [INTO]
```

where PG-HDER is the page header and j has value either one or two. Note that the page is assumed to be in BUF(j) already.

5.2.3.2 Free Space Management

As duplicates of DFD pages and data base pages are needed, empty pages are allocated. The old pages cannot be freed immediately as copies are created, otherwise they would be assigned and overwritten. They have to be kept until the command terminates. If the old page has dtm between dtrs and dtcs, they are freed at command termination. This is because they are already copies of old pages with respect to the present run unit. These pages are entered into a free space stack for the DML command known as the Command Free Space Stack (CFSS). All elements of this stack are integers formed by the concatenation of the device number and the physical block number. All elements of this stack are released at the termination of the DML command.

In order to update a page with dtm smaller than both dtrs and dtcs, a copy of the page is created. The device number and block number of the old page are concatenated and stored in a free space stack for a run unit known as the Run unit Free Space Stack (RFSS). The elements of this stack are returned as free space at the termination of the run unit. Note that whenever a DML command terminates, an end marker is stored into this stack.

The syntax of command to access and update the elements of the free space stack is as follows:

$$\begin{bmatrix}\text{GET}\\ \text{PUT}\end{bmatrix} \ \text{dev\#//blk\#} \ \begin{bmatrix}\text{FROM}\\ \text{INTO}\end{bmatrix} \begin{Bmatrix}\text{CFSS[CT]}\\ \text{RFSS[RT]}\end{Bmatrix}$$

where dev# is the device number, blk# is the block number, CT and RT are pointers to the top of the stacks CFSS and RFSS respectively. The slashes between dev# and blk#

denotes concatenation.

5.3 Overview of Recovery Modules

Recovery modules can be divided into two main types. One type is responsible for recovery data collection. The other type controls the rollback action. Chapter 6 describes the recovery modules in detail.

Recovery data collection modules can be further subdivided into three types. The first type of modules make preparation for recovery data collection at the start of a run unit and of a DML command. They are responsible for resetting values, initiation of variables etc.

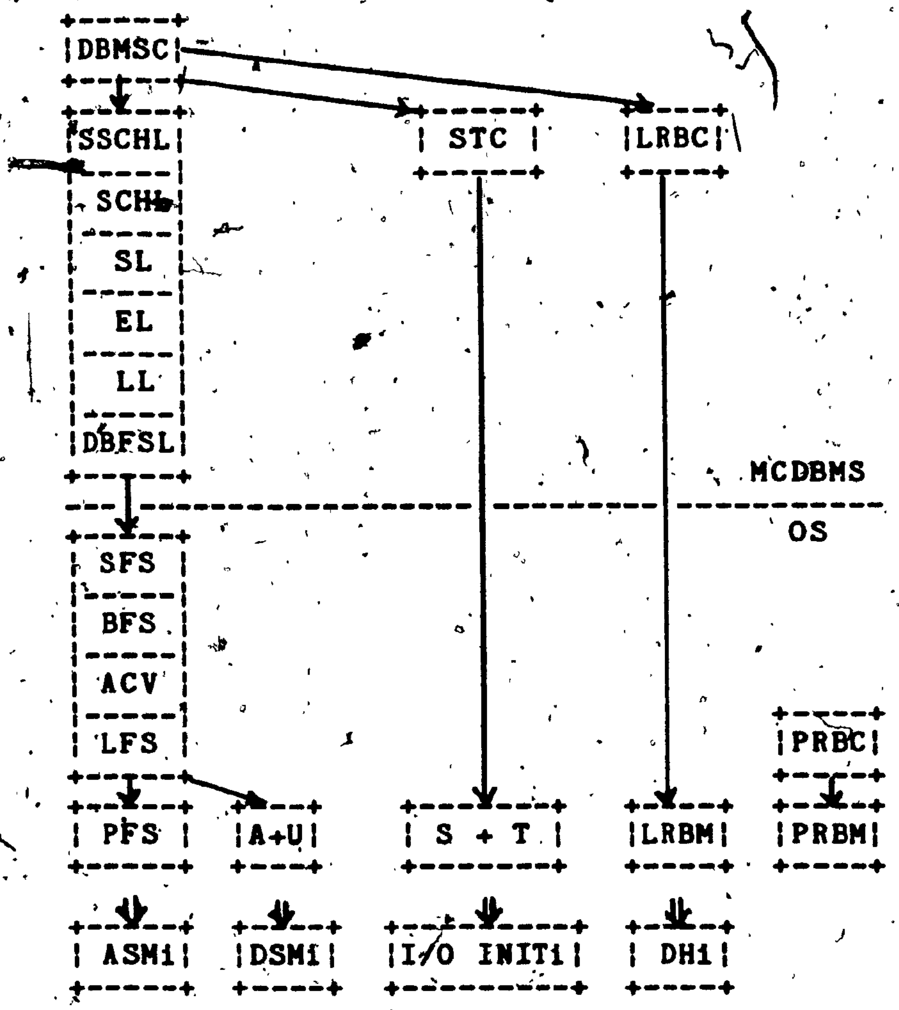
The second type of modules for recovery data collection deals with run unit and command termination. They are responsible for switching old DHFD to new ones and to return free space to the new DFD of each device. The first and second type of modules are represented by the module S+T in Fig. 5.4.

The third type of modules for recovery data collection provides service to access and update requests. Retrieval requests require traversal through the most current DFDs. Update requests require examination of the DFDs and perform modified careful replacement of the directory pages and the data base pages. These modules are represented by the module A+U in Fig. 5.4.

In general all three types of modules issue commands for physical block manipulation. The PFS level of the file system is the one which views the device in terms of physical blocks. The LFS above this level can only see logical bit strings. The physical blocks are abstracted away. The three types of recovery data collection modules have to reside in the PFS. The type three modules replace the function of the PFS for data base files.

In the MCDBMS, only read and write requests are issued to the file system. The LFS therefore calls the access and update modules only. A module is needed in the MCDBMS to issue special calls to the S+T module via the operating system in terms of supervisor calls. The module to issue such calls is known as STC. The operating system is extended to recognize these calls. Fig. 5.4 shows a brief layout of the recovery rollback system in relation to the MCDBMS and the file system. Single arrows show that the module pointed to is called by the one directly above it. Double arrows show that the module can be called by more than one module.

The modules for rollback recovery are divided into logical rollback and physical rollback. Logical rollback control module (LRBC) is initiated by the DBMSC module. The logical rollback modules (LRBM) residing in the PFS level provides command rollback and run unit rollback. Command rollback is more complicated than run unit rollback because



DBMSC	- MCDBMS control module		
STC	- Start and termination control module		
LRBC	- Logical rollback control module		
SCCHL	- Subschema level	SFS	- Symbolic file system
SCHL	- Schema level	BFS	- Basic file system
SL	- Structural level	ACV	- Access Control Verif.
EL	- Encoding level	LFS	- Logical file system
LL	- Location level	PFS	- Physical file system
DBFSL	- Data base file structure level		
A+U	- Access and update modules		
S+T	- Start and termination modules		
LRBM	- Logical rollback modules		
PRBC	- Physical rollback control modules		
PRBM	- Physical rollback modules		
ASMi	- Allocation strategy module of DEV(i)		
DSMi	- Device strategy module of DEV(i)		
I/O INITi	- Input/output initiation module of DEV(i)		
DHi	- Device handler of DEV(i)		

Fig. 5.4 General overview of the rollback recovery system

the Current Location Table has to be restored. If the UWA has been modified by the command, its contents are restored as well. Restoration of the CLT and the UWA buffer are not necessary for run unit rollback.

If a failure occurs during the execution of the command termination module or the run unit termination module, the operating system calls the physical rollback control (PRBC) module. The PRBC module in turn calls appropriate modules for rollback recovery, depending on whether the content of the main memory is retained or lost.

The DBMSC module acts as a control module for the MCDBMS and recovery modules. When the MCDBMS detects an error and the command cannot continue, the error condition propagates up until the DBMSC module is reached. After analysis the DBMSC module calls the LRBC module for rollback. When a run unit or a DML command starts or terminates, this module also invokes the STC module for appropriate action.

Chapter VI
ROLLBACK RECOVERY SUBSYSTEM MODULES

This chapter gives the details of the recovery modules. The functions of each recovery module are described. The modules of the rollback recovery subsystem are described top-down. Finally, some comments are made concerning the rollback recovery system.

6.1 RECOVERY CONTROL MODULES IN THE MCDBMS

6.1.1 DBMS Control

The DBMS control (DBMSC) module centralizes the control of the MCDBMS and the recovery modules. It interfaces the MCDBMS with the run unit. When a run unit starts to execute, the run unit transfers control to the DBMSC module. This module calls the start and termination control (STC) module which in turn calls the run unit starting (RU-START) module. The RU-START module performs initializations of the run unit. Control is transferred back to the STC module and then to the DBMSC module. Finally, the DBMSC module returns control to the run unit.

When a run unit encounters a DML command, control is transferred to the DBMSC module again. The DBMSC module first checks that the run unit has reset all the error conditions before it executes the next DML command. If there is no error condition, the DBMSC module calls the STC module which in turn calls the command starting (CMD-START) module. The command starting module makes preparation for modified careful replacement.

The DBMSC module then transfers control to the MCDBMS modules. If an error occurs, the error is propagated up to a level which determines the nature and extent of the error. The error is then propagated upwards until the DBMSC module is reached. The DBMSC module summarizes the error conditions indicated by the status registers of various sublevels of the MCDBMS. The error summary is recorded in the DB-STATUS register.

If rollback is needed, the DBMSC module calls the logical rollback control (LRBC) module to initiate rollback. If no rollback is needed and there is a non-fatal error, control is passed back to the run unit which can examine the DB-STATUS register for error information. If the command executes successfully without error, control is passed back to the DBMSC module. After confirming that the execution is successful by examining the status registers of various sublevels, the DBMSC module calls the STC module again. The STC module in turn calls the command termination (CMD-TERM) module to change block one pointers of modified storage devices to point to the new DHFD blocks.

When a run unit terminates the run unit passes control to the DBMSC module. The DBMSC module ensures that all error conditions are reset before the run unit is allowed to terminate. If the DB-STATUS register indicates that there is no error, the STC module is called to initiate the run unit termination (RU-TERM) module. If there is an error the LRBC module is called to initiate run unit rollback.

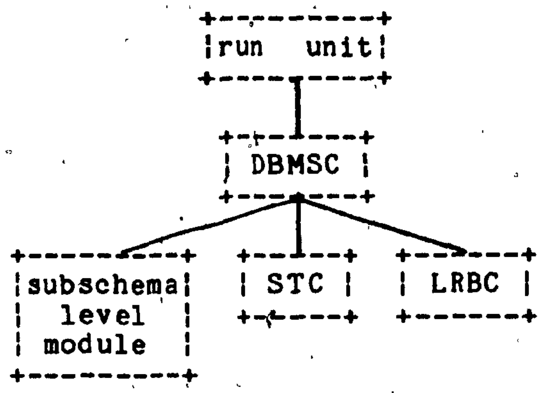


Fig. 6.1 Interaction of the DBMSC module with other modules

Fig. 6.1 shows the interaction of the DBMSC module with other modules. The DBMSC module interacts with the run unit. It calls the subschema level module, the STC and the LRBC modules. These three modules are in the MODBMS level.

6.1.2 Start and Termination Control

The start and termination (STC) module is called exclusively by the DBMSC module. Its function can be seen clearly from the preceeding section. Basically it calls two starting modules, RU-START and CMD-START, and two termination modules, RU-TERM and CMD-TERM. The starting modules make preparation for modified careful replacement of physical blocks. The termination module puts the new copies of modified blocks into effect.

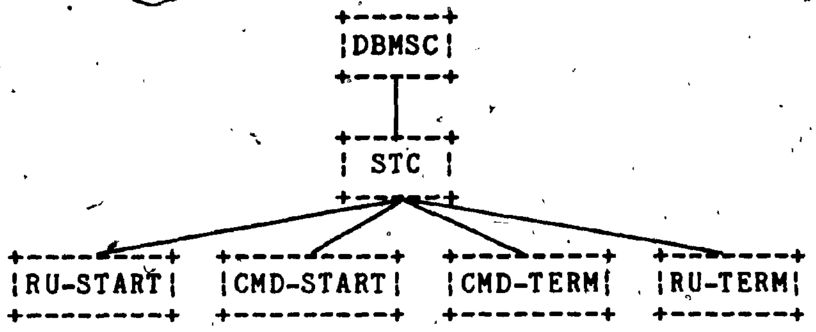


Fig. 6.2 Interaction of the STC module with other modules

Fig. 6.2 shows the interaction of the STC module with other modules. The STC module is in the MCDBMS and the four modules which it calls are in the PFS level.

6.1.3 Logical Rollback Control

The logical rollback control (LRBC) module resides in the MCDBMS. It is responsible for rollback of a DML command and a run unit. The request for rollback originates from the DBMSC module. This module deals with logical rollback as opposed to physical rollback. Logical rollback is initiated by the MCDBMS. The MCDBMS is aware of the rollback action. Physical rollback deals with situations where the rollback action is invisible to the MCDBMS. Physical rollback is initiated in the PFS level. The LRBC module is called exclusively by the DBMSC module.

Fig. 6.3 shows the interaction of the LRBC module with other modules. The LRBC module calls the CMD-RB and RU-RB

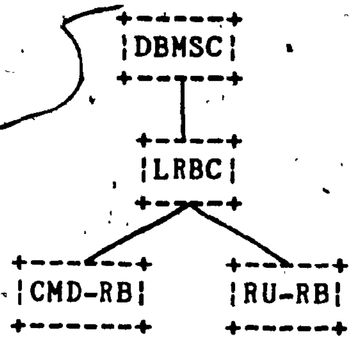


Fig. 6.3 Interaction of the LRBC module with other modules

modules for command and run unit rollback. These two modules reside in the PFS level.

6.2 STARTING MODULES

There are two starting modules in the rollback recovery subsystem. They are run unit and DML command starting modules. These two modules are called by the STC module in the MCDBMS.

6.2.1 Starting of a Run Unit

The RU-START module is responsible for initializations when a run unit starts. The date and time when a run unit starts (dtrs) is recorded when the run unit starts execution. At this time block zero of devices 1 to N points to the DHFD block of the most recent run unit. Block zero of device zero points to the block containing the RDD array. The RU-START module produces a copy of the CDD array from the RDD array as initialization in device zero. The array copy acts as the CDD array for the most recent DML command.

Block one of device zero is set to point to the CDD block.

Block zero of devices 1 to N points to the DHFD block of the most recent run unit. For all storage devices which the run unit can modify, two copies of the DHFD block is made in blocks two to four of the devices. Block one is made to point to one of the copies. The copy which block one points to serves as the DHFD block of the most recent DML command. The other copy acts as the DHFD block for the current DML command. The pointers for free space stacks RFSS and CFSS are initialized to zero. The array element of the MV and CMV arrays are also initialized to zero. The RU-START module deals with physical blocks and so it resides in the PFS level. After all initialization have been performed the RU-START module returns control to the STC module calling it.

6.2.2 Starting of a DML Command

The module DML-START is called by the STC module when every DML command starts executing. The date and time the command starts (dtcs) is recorded. The CLT is copied onto block five of device zero. If the DML command involves transferring data to the UWA, the data to be overwritten is first stored in block six of device zero in case rollback is needed.

Devices 1 to N has been initialized so that blocks two to four contains the DHFD pages. Consider the case when a device has some of its blocks modified in the last DML command. On command termination block one has been switched to point to the new DHFD block. Block zero still points to the DHFD block of the most recent run unit. The block left within blocks two to four has to be initiated by copying the DHFD block of the most recent DML command onto it. If a device has not been modified by the last DML command, then blocks two to four are not altered. No creation of copies of the DHFD page is necessary. The former case is indicated by a positive value in the corresponding element of the MV array. After creation of copies of the DHFD blocks, the MV array elements are reset to zero. The latter case has value zero in the MV elements.

6.3 DML COMMAND OPERATING MODULES

Two modules, one for access while the other for update of the data base, are described within this section. The modules performs operations on the data base using the modified careful replacement strategy.

6.3.1 DML Command Access

The module CMD-ACCESS can be called many times within a DML command. Its function is purely to retrieve a physical block containing the required data base page. It takes the place of the PFS and maps a logical record to a physical block. The data base is not modified by this module. This module performs the search of the required block through the

DHFD ~~of~~ the executing DML command down the DFD until the physical block is found. The LFS of the file system calls this module whenever the function (fn) is READ and the file is a data base file.

6.3.2 DML Command Update

A DML command to update a data base page requires a number of operations. If the block has previously been modified within the current DML command, update in place occurs. If no changes were previously made on the block, a copy is created and the update is performed on the copy. If the latter condition occurs, copies of the DFD pages leading to the data base page are made if no copies exist already within the same command. This is similar to the situation described in section 4.2.2.1.

The CMD-UPDATE module assumes that the CMD-ACCESS module has been executed and the block containing the data base page to be modified is in the file buffer. Modification takes place within the CMD-UPDATE module. Note that every time the CMD-UPDATE module is called, the appropriate elements of both the MV and CMV arrays are incremented by one.

As the module traverses down the DFD, copies of blocks without duplicates are made. The existence of duplicates depends on whether dtm is greater than dtcs or not. If the dtm value of a block is greater than dtcs, a duplicate of the block exists within the current DML command. If dtm is between dtrs and dtcs, the block has been modified by former DML commands but not by the present one. This block number is entered into the free space stack CFSS to be returned as free space at the termination of the current DML command. A new block is assigned as duplicate for update. If dtm is smaller than dtrs, the block has not been modified within the present run unit. The block number is entered into the free space stack RFSS to be returned as free space at the termination of the run unit. A new block is assigned as duplicate for update.

6.3.3 Summary

The operations of the two modules for data base

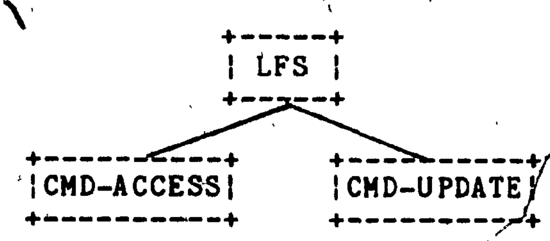


Fig. 6.4 DML command access and update modules

operatios have been described above. Fig. 6.4 shows the CMD-ACCESS and CMD-UPDATE modules are called by the LFS above. The access and update modules takes the place of the PFS for data base file access and update. They reside in the PFS level of the file system.

6.4 TERMINATION MODULES

6.4.1 DML Command Termination

Theoretically speaking, a command termination indicates that the MCDBMS has been transformed from one state to another. This transition to a new state is indicated by the state variables. The CMD-TERM module serves to assign values to the state variables to indicate that the new state has been reached. In other words, the module updates the data base to reflect the successful completion of the DML command.

The CMD-TERM module first returns all free space of the CFSS array to the devices. For all devices with modified blocks, their block one are set to point to the new DHFD blocks. The free space stack pointer for DML command is reset to zero. A zero marker is added to the free space stack RFSS. This marker is used for rollback. Its function is described in detail in section 6.5.1. The CDD array is updated for the command and stored into device zero. Block one of device zero is switched to point to the new CDD block.

6.4.2 Run Unit Termination

The run unit termination module, RU-TERM, consolidates all changes made to the data base within the current run unit. A copy of the most recent DHFD page is made for each storage device with modified blocks within the run unit. The blocks indicated by the free space stack RFSS are returned to the devices. Block zero of all the modified devices are then updated to point to the new DHED block. The new values of the RDD array are updated. The pointer for the RFSS array is reset to zero. The new RDD array is then stored in device one. Block zero of device one is switched to point to the new RDD block.

6.5 LOGICAL ROLLBACK RECOVERY

Logical rollback recovery is visible and is initiated by the MCDBMS. The DBMSC module detects the error condition and issues a command to the LRBC module to perform either command or run unit rollback.

6.5.1 DML Command Rollback

DML command rollback is responsible to undo all changes made to the data base from the start of the command up to the point of request for rollback. A command rollback request is assumed to be issued in the process of executing the DML command. In other words, the DBMSC discovers that the command is unable to proceed any longer. This situation implies that the CMD-TERM module has not been called when the CMD-RB module is called.

The command rollback action is fast because the switching action of the careful replacement strategy is simply not performed. This would retain the data base as it was before the command was executed. The only work left is to replace the variables that have been modified by the command. These variables include the CLT, the buffer contents of UWA, the MV and CMV arrays and the free space stack.

The function of the end marker to distinguish blocks returned by a DML command to the RFSS array can be clearly seen here. In general the recovery system does not know the number of entries that the current DML command has entered into the RFSS array. The end marker is inserted only by the CMD-TERM module. The CMD-TERM module has not been executed when the present rollback module is called. Therefore the first end marker found in the RFSS array tracing down from the top must be the one entered at the termination of the most recent command. If the top element is not the end marker, entries must have been added onto the stack by the current DML command. Elements are popped from the stack until the end marker is reached. If the top element is already the end marker, the current DML command has not push any entry into the array.

6.5.2 Run Unit Rollback

Run unit rollback undoes all changes made to the data base from the start of the run unit up to the point of request for rollback. This module is called exclusively by the LRBC module. The LRBC module is in turn called by the DBMSC module. Neither the CLT nor any buffer value in the UWA need be reloaded. As in the CMD-RB module, the execution of the run unit rollback (RU-RB) module and the RU-TERM module are mutually exclusive.

The RU-RB module is in general called after a successful DML command execution when the MCDBMS cannot proceed. This situation is identified when an error condition has not been reset by a run unit in the DB-STATUS register and it attempts to execute the next DML command. The RU-RB module may also be called just before run unit termination when the error condition in the DB-STATUS register has not been reset. The DBMSC checks the error condition of the DB-STATUS register before calling CMD-START or RU-TERM module. The RU-RB module simply returns control to the run unit without executing the RU-TERM module. This leaves block zero of all storage devices unaltered. Block zero still contains pointers to the DHFD blocks or the RDD block with respect to the last run unit.

6.6 PHYSICAL ROLLBACK RECOVERY

Physical rollback recovery is invisible to the MCDBMS. A failure can occur during the execution of the CMD-TERM or RU-TERM module. This type of failure occurs in the operating system level. Abnormal termination or interruption of run unit execution can occur as a result of power failure or the operating system crash. It is necessary to restore the data base into a consistent state. Resumption of execution of the run unit at the point where it left off is perferred. Such a rollforward action is possible only if the content of the main memory is not lost. This specific type of recovery rollback is not of the logical nature visible to the MCDBMS.

6.6.1 Physical Rollback Recovery Control

Physical rollback recovery takes place only in the operating system. It is invisible to the MCDBMS. The physical recovery modules are called by the file system and not by the MCDBMS. The physical rollback recovery (PRBC) module determines which type of rollback action is required.

Error during the execution of the CMD-TERM or RU-TERM modules would result in transfer of control to the operating system. The operating system calls the PRBC module. The operating system also sets the condition that determines

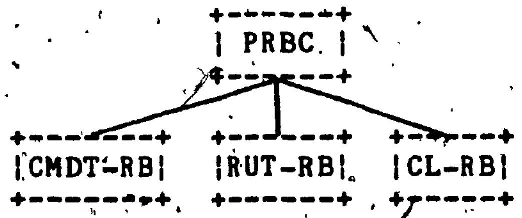


Fig. 5.5 Modules called by the PRBC module

whether or not the contents of the main memory are retained. Fig. 6.5 shows the modules called by the PRBC module. The CMDT-RB module performs rollback on the command executed. The RUT-RB module attempts to rollback and then to execute the RU-TERM module to complete the termination process. The CL-RB module performs rollback when the contents of the main memory are cleared or ruined.

6.6.2 Failure During Command Termination

If a failure occurs during the execution of the CMD-TERM module, some storage devices may have their block one values modified while others have not. If the contents of the main memory are lost, a run unit rollback is required. This is because the operating system is unable to identify the DML command it has last executed. If the contents of the main memory are retained, a command rollback can be performed and the DML command is executed again.

The block in device one containing the CDD array is replaced by the modified careful replacement strategy. This array can be used as a reference point to compare block one of all storage devices. This array is loaded into main memory. Block one of each storage device where the array MV has corresponding element value greater than zero is examined. If the values differ, the value of the CDD element is copied onto block one and stored. The array MV can be used in deciding which device to check because it is not altered within the CMD-TERM module. After restoring block one values of devices 1 to N, the CMD-RB module is called. This module is called because the CLT, the UWA buffer value, CMV, MV, RFSS arrays all need rollback.

6.6.3 Failure during Run Unit Termination

The RUT-RB module for recovery of failure during RU-TERM execution assumes that the last DML command has been executed successfully; otherwise RU-TERM module is not executed. It is necessary for the RUT-RB module to perform rollback and then to execute the RU-TERM module to terminate the run unit. Rollback is required to bring the data base to the state just before the RU-TERM module is called. Rollforward is required to complete the run unit

If the execution of the RU-TERM module has not come to the point to reset the pointer to the free space stack RFSS, the run unit termination modude is executed again. The reexecution takes place after restoring the RDD array to the values of the last run unit. If the execution has reached the point where the RFSS pointer has been reset, it only remains to store the result of the new RDD into device zero.

6.6.4 Failure with Contents of Main Memory Damaged

When the contents of the main memory is lost or is partially damaged, the run unit can no longer continue. It is impossible to identify the DML command last executed or the command the MCDBMS has been executing. A run unit rollback is needed. In the worst case the run unit may be executing the RU-TERM module. Some devices may have the value of their block zero updated while others are not yet altered.

The CL-RB module performs recovery for abnormal terminations with the contents of main memory lost or unreliable. The module performs a run unit rollback based on block zero of device zero. The block of device zero containing the RDD array is first loaded into the main memory. Block zero of device one to N is loaded into file buffer one by one. If the value of block zero of a device is the same as that of the corresponding RDD array element, no action is taken. If the values differ, the value of the array element is stored into block zero of the device. The module reduces the data base into the state of the most recent run unit.

6.7 SUMMARY AND COMMENTS

6.7.1 Summary

A rollback recovery system has been proposed for the MCDBMS. The recovery system is based on a combination of both the careful replacement and the differential techniques. Three modules are added onto the MCDBMS for recovery initiation. One of the three modules is designed for centralized recovery control. The second one invokes the recovery data collection and consolidation modules. The third one issues commands for logical rollback.

A total of twelve modules are added onto the file system for recovery data collection and recovery action. Out of the twelve modules, six deal with rollback recovery. Four modules out of six are responsible for physical rollback which is invisible to the MCDBMS. The access and update modules replace the PFS of the file system for data base files. Fig. 6.6 shows the relationships of the modules in the rollback recovery system.

```
                    +------+
                    |DBMSC|
                    +------+
                       |-----------------------+
                       |                       |
                       V                       V
                    +------+              +-------+
                    | STC |               | LRBC |
                    +------+              +-------+
                       |                       |
                       |                       |          MCDBMS
                       |                       |
------------------------+-----------------------+-----------------
                       |                       |
                       |                       |             OS
                       |                       |
      +---+            |                       |          +----+
      |LFS|            |----------+            |          |PRBC|
      +---+            |          |            |          +----+
        |              |          |            |            |
        V              V          V            V            V
+-----------+ +----------+ +---------+ +-------+ +--------+
|CMD-ACCESS| | RU-START| | RU-TERM| | RU-RB| |   CL-RB|
|CMD-UPDATE| |CMD-START| |CMD-TERM| |CMD-RB| | RUT-RB|
+-----------+ +----------+ +---------+ +-------+ |CMDT-RB|
                                                 +--------+
```

Fig. 6.6 Modules of the rollback recovery system

There is one potential infinite loop hidden in the rollback recovery system. It is shown in Fig. 6.7. When a failure occurs during the execution of the run unit termination module, the PRBC module is invoked. After checking the error condition, the PRBC module calls the RU-RB module. The latter module restores the environment

```
+----------+
|   PRBC   |----------
+----------+         |
     ^               V
     |          +---------+
     |          | RUT-RB  |
     |          +---------+
     |               |
+----------+         |
| RU-TERM  |<---------
+----------+
```

Fig. 6.7 Potential infinite loop in the rollback recovery subsystem

before execution of the RU-TERM module and calls the module again if the pointer to the RFSS array has not been reset. The operating system is responsible to check out this infinite loop and call CL-RB if the run unit cannot be terminated properly.

When a failure occurs during the execution of the CMD-RB or RU-RB modules, the operating system tries to reexecute the module for a number of times. If the execution still fails, the MCDBMS is stopped for analysis. After the system is restored, the CL-RB module is executed.

6.7.2 Pros and Cons

The advantages of the rollbck recovery subsytem are as follows

1. Fast command rollback.
2. Very fast run unit rollback.
3. Run unit rollback can be performed even if the run unit terminates abnormally with the contents of the main memory damaged. The subsytem is crash resistant.

4. If the contents of the main memory are retained during a crash, rollback of the executing DML command can be achieved. The run unit does not have to start from the beginning.

5. The recovery subsytem can be initiated by the MCDBMS for logical type of errors that arise during the run unit execution.

6. The modules CMD-RB and RU-RB can be executed again if another crash or a failure arises during the execution of these two modules. They do not create any overhead or generate any information that has to be undone in the subsequent rollback.

7. The rollback subsytem is faster than pure careful replacement as fewer pages need to be replaced. This has been explained in section 4.2.2.1.

8. The CL-RB module, which performs run unit rollback when the contents of the main memory is lost, is one which one can always turn to if all attempts of logical rollback failed.

9. The rollback subsytem replaces the old blocks by the new ones as a run unit executes. This is different from the differential file technique which accumulates modified blocks. No separate period of time is needed to consolidate the differential file into the master file.

The disadvantages of the rollback recovery subsytem are as follows

1. Updating a page results in updates of a number of pages. The amount of I/O traffic is increased as compared with a subsytem with conventional rollback facilities.

2. More time is required to execute a DML command because of the overhead involved. The time taken would be even longer for commands that update a number of data base pages.

3. The file sytem has to be modified at the PFS level. Modification of a large piece of software could give rise to errors unless the software is well structured and the interfaces are well defined.

6.7.3 Limitations

The limitations of the rollback recovery subsytem are as follows

1. The subsytem is designed for a single user DBMS only. No concurrent processing of the run units is possible.

2. The subsytem cannot handle hardware crashes such as disk head crash.

3. The unit of rollback is either a DML command or a run unit. There could be cases when a user wants to rollback a number of DML commands once he finds that a certain DML command cannot function as expected. He may want to retreat to a former point in the application program and to follow an alternate path. The rollback recovery subsytem fails to provide such a feature.

Chapter VII
CONCLUSION

A multilevel CODASYL DBMS model has been designed. The model has six levels. Each level acts as a virtual machine and provides abstraction to the level above.

The subschema level interfaces with the user via the DBMS control module. This level provides a local view of the data base to the user. The schema level provides a global view to the DBA. The two data structures in this level are record and set. They represent entities and their relationships in the real world.

The structural level contains structural level record, index and pointer array to implement the data structures of the schema level. The schema level user is not concerned with the way in which a set is implemented. The structural level therefore provides a level of abstraction to the level above. The encoding level implements the data structures of the structural level by a primitive construct, the storage record. An occurrence of a structural level record, an index or a pointer array can be represented by one or more occurrences of storage records which may belong to different storage record types. The encoding level presents a level of abstraction to the structural level. The structural level is unaware of how its data structures are constructed.

The encoding level also provides a level of abstraction to the location level below. The location level maps bit strings onto a paged address space. The location level abstracts away the bit string content of pages to the data base file structure level below it. It requests accesses or updates to a page as a unit rather than bit strings. The data base file structure level interfaces with the operating system. It maps the paged address space to the records of a file. A record here is equivalent to the physical block size of a device. This level interfaces the MCDBMS with the operating system.

The structural level, encoding level and the location level of the MCDBMS are an expansion of the data storage description of the CODASYL 1978 proposal. These three levels provide clear and well-defined structuring of the DBMS. The data base file structure level provides an interface to the operating system just as any other user process does. The limitation of this design is to assume that the MCDBMS serves a single user and is link-edited to the run unit requiring it. With respect to the design goals set up, the MCDBMS has fulfilled the goals.

Each sublevel of each level of the MCDBMS has its own status register for error reporting. Error conditions propagate upwards until the DBMS control module is reached. The DBMS control module calls rollback modules if the DB-STATUS indicates the need for rollback.

The detail modular design of the MCDBMS model requires a total of 126 modules. The possibility of realizing the MCDBMS design is verified using the modular design method.

A rollback recovery system has also been designed. The system is based on a combination of the careful replacement and the differential file techniques. The system assumes that file directories are in the form of a directory tree in each storage device. The rollback recovery system can provide fast command rollback and run unit rollback. Even when the content of the main memory is lost, run unit rollback can still be achieved in a short time. Three recovery control modules are added into the MCDBMS level. Six recovery data collection and six recovery rollback modules are added into the physical file system. The design goals of the rollback recovery system have been achieved. Although the rollback recovery system is intended to apply to the MCDBMS, in principle it can be extended to other data models using different data sublanguages. The DML is a highly procedural language capable of updating more than one page at a time.

The MCDBMS system and the rollback recovery system offer a number of possibilities for further research. First, the MCDBMS can be implemented to compare its performance with existing DBMSs. Second, the rollback recovery system can be implemented to compare the performance with systems using careful replacement. The time required for processing,

state of the art report on On-Line Data Bases, v.2, Invited papers, Infotech Intl. Ltd., Maidenhead, England, 1977, pp. 339-354.

[Toze78] Tozer, E.E. "The data storage description language (DSDL)" in Infotech state of the art report on Data Base Technology, v.2, Invited papers, Infotech Intl. Ltd., Maidenhead, England, 1978, pp. 385-421.

[Ullm80] Ullman, J.D. Principles of Database Systems, Computer Science Press, Maryland, USA, 1980, ch. 1.

[Verh77] Verhofstad, J.S.M. "Recovery and crash resistance in a filing system" in ACM SIGMOD Procs., D.C.P. Smith (Ed.), Aug 1977, pp.158-167.

[Verh78] Verhofstad, J.S.M. "Recovery techniques for database systems", ACM Comput. Surv., <u>10</u>, 2 (June 1978), 167-195.

[Wied77] Wiederhold, G. Database Design, McGraw-Hill, Computer Science Series, MsGraw-Hill, NY, USA, 1977.

[Wilk72] Wilkes, M.V. "On preserving the integrity of data bases", Comp. J., <u>15</u>, 3 (Aug 1972), 191-194.

[Wulf76] Wulf, W.A. "Structured programming in the basic layers of an operating system" in Language Hierarchies and Interfaces, F.L. Bauer and K. Samelson (Eds.), Lecture notes in computer science, v.46, Springer-Verlag, Berlin, 1976, pp. 293-344.

[Yorm76] Yormark, B. "The ANSI/X3/SPARC/SGDBMS architecture" in The ANSI/SPARC DBMS Model, Procs. 2nd SHARE Working Conf. On DBMS, Montreal, Canada, Apr 26-30, 1976, D.A. Jardine (Ed.), North-Holland, Amsterdem, 1976, pp. 1-21.

APPENDIX A

MODULE NAME	MODULE NUMBER	DESCRIPTION
DBF-MAP	10000	Maps data base pages to records of files
LOAD-PG	10100	Loads a data base page from the data base into the location level buffer
STORE-PG	10101	Stores a data base page from the location buffer into the data base

Table A.1 Modules of the data base file structural level

MODULE NAME	MODULE NUMBER	DESCRIPTION
DB-SPACE	20000	This module accepts an input logical data base page number and outputs a free page number closest to that of the input. It keeps track of all the page assignments of the data base internally
LL-INFO	20010	Provides information on the mapping of storage records into the paged address space and the set relations between the storage records

Table A.2 Modules in sublevel zero of the location level

MODULE NAME	MODULE NUMBER	DESCRIPTION
DELETE-CUR-LOC	20100	Deletes an entry from the PCLT
INSERT-CUR-LOC	20101	Inserts an entry into the PCLT
SEARCH-CUR-LOC	20102	Searches the PCLT for all or part of a given storage data base key
DELETE-DIRY-ENTRY	20110	Deletes an entry from a DID page
INSERT-DIRY-ENTRY	20111	Inserts an entry into a DID page
MODIFY-DIRY-ENTRY	20112	Modifies an entry of a DID page
RETRIEVE-DIRY-ENTRY	20113	Retrieves the page of the direct index from the DID page given the storage record type
SEARCH-DIRY-SPACE	20114	Searches for a DID page with at least one empty entry space. If the DID is full, a new DID page is created

Table A.3 Modules of sublevel one of the location level

MODULE NAME	MODULE NUMBER	DESCRIPTION
CUR-LOC-OP	20200	Provides operation for deletion, insertion and searching of the PCLT
DELETE-DIR-INDEX	20210	Deletes an entry from a DI page
INSERT-DIR-INDEX	20211	Inserts an entry into a DI page
MODIFY-DIR-INDEX	20212	Modifies an entry of a DI page
RETRIEVE-DIR-INDEX	20213	Retrieves the page number of the storage record in the DI given the storage data base key
SEARCH-DIR-INDEX	20214	Searches for a DI page with at least one empty entry space. If the DI is full, a new DI page is created
SEARCH-STRING-IN-PAGE	20220	Searches for a storage record within a data base page in the LLB

Table A.4 Modules of sublevel two of the location level

MODULE NAME	MODULE NUMBER	DESCRIPTION
ALLOCATE	20300	Searches the free space list of the LLB for a block size sufficient to hold a storage record
ALT-PLACEMENT	20301	Generates an alternative page number that the input storage record may reside
APPROX-PLACEMENT	20302	Produces the focal page number around which the stirage record may reside
BOUND	20303	Gives the boundary within which a storage record may reside

Table A.5 Modules of sublevel three of the location table

MODULE NAME	MODULE NUMBER	DESCRIPTION
FIND-PAGE-SPACE	20400	Finds the page with free space to adapt the given size, loads the page into the LLB and updates the free space list to adapt the string to be stored
FIND-STRING	20401	Searches for a storage record with a given storage data base key and loads the page containing it into LLB
COPY-ELB-TO-LLB	20410	Copies a storage record from ELB to LLB
COPY-LLB-TO-ELB	20411	Copies a storage record from LLB to ELB
DEALLOCATE	20420	Returns previously occupied space to the page as free space

Table A.6 Modules of sublevel four in the location level

MODULE NAME	MODULE NUMBER	DESCRIPTION
DELETE-STO-REC	20500	Deletes a storage record from the data base
MODIFY-STO-REC	20501	Modifies a storage record in the data base
RETRIEVE-STO-REC	20502	Retrieves a storage record from the data base
STORE-STO-REC	20503	Stores a new storage record into the data base

Table A.7 Modules of sublevel five of the location level

MODULE NAME	MODULE NUMBER	DESCRIPTION
EL-INFO	30000	Provides information on the mapping of structural level data structures to storage records

Table A.8 Module in sublevel zero of the encoding level

MODULE NAME	MODULE NUMBER	DESCRIPTION
XFER-DECODE	30100	Transfers a storage record in the ELB to the appropriate fields of the SLR in the SLB1 with decoding if necessary
XFER-ENCODE	30101	Option one- transfers one field of a SLR to a correcponding field in a storage record. Option two- transfers fields of a SLR to all corresponding fields of a storage. Both options may have encoding if necessary.
CREATE-STR-REC	30200	Creates the storage records correspond-ing to a new SLR
DELETE-STR-REC	30201	Deletes all the storage records corres-ponding to a SLR
REPLACE-STR-FLD	30202	Replaces a field of the SLR resulting in the replacement of a field of a storage record with encoding if necessary
REPLACE-STR-REC	30203	Replaces the storage records correspond-ing to a SLR with encoding if necessary
RETRIEVE-STR-REC	30204	Retrieves a SLR by retrieving all storage records corresponding to the SLR and available to the subschema of the run unit. Decoding takes place if necessary.

Table A.9 Modules in sublevels one and two of the encoding level for structural level record (SLR) manipulation

MODULE NAME	MODULE NUMBER	DESCRIPTION
CREATE-PASR	30110	Creates an occurrence of a PASR
SHIFT-PASR-ELEMENT	30111	Shifts the elements of a PASR up one element space
CREATE-PA	30210	Creates the pointer array
DELETE-PA	30211	Deletes a pointer array resulting in deletion of all PASRs associated with the pointer array
DELETE-PA-ELEMENT	30212	Deletes an element from the pointer array
INSERT-PA-ELEMENT	30213	Inserts an element to the end of the pointer array
RETRIEVE-PA-ELEMENT	30214	Retrieves an element in the pointer array
RETRIEVE-FL-PA-ELEMENT	30215	Retrieves the first or last element of the pointer array
SEARCH-PA-ELEMENT	30216	Searches through the pointer array to find an element with value that matches with the input

Table A.10 Modules in sublevels one and two of the encoding level to handle pointer arrays

MODULE NAME	MODULE NUMBER	DESCRIPTION
DELETE-ISR	30120	Deletes an ISR from the data base
FIND-ISR	30121	Finds an ISR satisfying the input specification and loads it into the encoding level buffer.
SPLIT-ISR	30122	Splits a fully packed ISR into two by creating a new ISR and copying half of the entry pairs of the original ISR to the new ISR.
CREATE-INDEX	30220	Creates the HISR and the first ISR
DELETE-INDEX	30221	Deletes all ISRs and the HISR corresponding to a particular set occurrence
DELETE-INDEX-ENTRY	30222	Deletes an entry pair of an ISR. This may recult in further deletion of an ISR if it is empty except for the case that it is the only storage record remaining in the set
INSERT-INDEX-ENTRY	30223	Inserts an entry pair into an ISR. This may result in splitting of ISR if it is fully packed already
RETRIEVE-INDEX-ENTRY	30224	Retrieves the entry pair of an index with a supplied key value
RETRIEVE-FL-INDEX-ENTRY	30225	Retrieves the first or last entry pair of an index
RETRIEVE-NP-INDEX-ENTRY	30226	Retrieves the next or prior entry pair of an index relative to the supplied logical data base key of an entry pair
SEARCH-INDEX-CONTENT	30227	Tests whether a given logical data base key is in the index

Table A.11 Modules in sublevels one and two of the encoding level for index management

MODULE NAME	MODULE NUMBER	DESCRIPTION
S-INFO	40000	Provides information of the schema DDL
SL-INFO	40001	Provides information of the constructs of the structural level
SS-INFO	40002	Provides information of the subschema DDL

Table A.12 Modules of sublevel zero of the structural level

MODULE NAME	MODULE NUMBER	DESCRIPTION
XFER-ID-FROM-UWA	40100	Transfers an identifier from a record in the UWA to the corresponding field of the SLR in SLR2
XFER-ID-TO-UWA	40101	Transfer a field of the SLR in SLR2 to the corresponding data identifier of a record in the UWA
XFER-VAR-FROM-UWA	40110	Transfers a variable or a data identifier of a record in the UWA to the structural level

Table A.13 Modules of sublevel one of the structural level

MODULE NAME	MODULE NUMBER	DESCRIPTION
XFER-IDS-FROM-UWA	40200	Transfers a number of data identifiers from a subschema record in the UWA to SLB2
XFER-REC-FROM-UWA	40201	Transfers a subschema record from the UWA to SLB2
XFER-REC-TO-UWA	40202	Transfers a record from SLB1 to the corresponding record buffer in UWA
MATCH-FLDS	40210	Matches the fields of an owner SLR in SLB1 with certain fields of the record in SLB2 or with some variables in the UWA
MATCH-KEY	40211	Matches an index of a SLR with a key value in a field of the subschema member record

Table A.14 Modules of sublevel two of the structural level

MODULE NAME	MODULE NUMBER	DESCRIPTION
DELETE-RCT-ENTRY	40300	Nullifies the currency indicator of a record type in the RCT
MODIFY-RCT-ENTRY	40301	Modifies an entry in the RCT for change of currency of a record type
RETRIEVE-RCT-ENTRY	40302	Retrieves the logical data base key of the current record of a given record type
DELETE-SCT-ENTRY	40310	Nullifies the currency indicator of a set type in the SCT
MODIFY-SCT-ENTRY	40311	Modifies an entry in the SCT for change of currency of a set type
RETRIEVE-SCT-ENTRY	40312	Retrieves the logical data base key of the current record of a given set type
REC-SEARCH-CHAIN	40320	Searches through a record type chain for matching of fields to be used in set selection
REC-SEARCH-IND	40321	Searches through a record type index for matching of fields to be used in set selection
REC-SEARCH-PA	40322	Searches through a record type pointer array for matching of fields to be used in set selection
UPDATE-REC-CHAIN	40330	Updates the record type chain for insertion or deletion of an occurrence of the record type
UPDATE-REC-INDEX	40331	Updates the record type indes for insertion or deletion of an occurrence of the record type
UPDATE-REC-PA	40332	Updates the record type pointer array for insertion or deletion of an occurrence of the record type

Table A.15 Modules of sublevel three of the structural level

MODULE NAME	NUMBER	DESCRIPTION
SET-CHAIN-FIRST-CON	40400	Connects a member record to the first position of a set chain
SET-CHAIN-LAST-CON	40401	Connects a member record to the last position of a set chain
SET-CHAIN-NEXT-CON	40402	Connects a member record to the next record position of a set chain
SET-CHAIN-PRIOR-CON	40403	Connects a member record to the prior record position of a set chain
SET-CHAIN-SORTED-CON	40404	Connects a member record to a set chain according to a predefined ordering of the key
FIND-LAST-SET-CHAIN	40410	Finds the last member in a set chain
RETRIEVE-FIRST-IN-SET	40411	Retrieves the first element of a given set
RETRIEVE-NEXT-IN-SET	40412	Retrieves the next member record of a given set
CONS-SET-SEL	40422	Selects an owner record of a set based of structural constraint
FLDS-SET-SEL	40421	Selects an owner record of a set by matching certain fields of the member record with the owner record or with some variables
KEY-SET-SEL	40422	Selects an owner record of a set with a specific key value
UPDATE-REC-TYPE	40430	Updates the record type information when a structural record occurrence is created of erased
UPDATE-CUR	40440	Updates the currency table of the record and all sets in which the membership of the record changes

Table A.16 Modules of sublevel four of the structural level

MODULE NAME	MODULE NUMBER	DESCRIPTION
SET-CHAIN-ORD-CON	40500	Connects a member record into a set chain according to the set ordering criteria as specified in the schema DDL
SET-SEL	40510	Selects an owner record of a set for a given member record occurrence

Table A.17 Modules of sublevel five of the structural level

MODULE NAME	MODULE NUMBER	DESCRIPTION
SET-CHAIN-CON	40600	Connects a member record into a set chain
SET-IND-CON	40601	Connects a member recore into a set index
SET-PA-CON	40602	Connects a member record into a set pointer array
SET-CHAIN-DISCON	40610	Disconnects a member record from a set chain
SET-OWNER-SEL	40620	Selects the owner record of a set given the member record

Table A.18 Modules of sublevel six of the structural level

MODULE NAME	MODULE NUMBER	DESCRIPTION
SET-CON	40700	Connects a member record to the appropriate owner record of a set
SET-DISCON	40701	Disconnects a member from its owner record in a set
SET-OWNER	40710	Creates the structural level set constructs for an owner SLR. It is

Table A.19 Modules of sublevel seven of the structural level

MODULE NAME	MODULE NUMBER	DESCRIPTION
ACC-SET-CON	40800	Connects a member record to a number of sets
ACC-SET-DISCON	40801	Disconnects a member record from a number of sets
ACC-SET-OWNER	40810	Creates the structural level set constructs for a number of sets of an owner record

Table A.20 Modules of sublevel eight of the structural level

MODULE NAME	MODULE NUMBER	DESCRIPTION
ERASE-REC	40900	Erases a record from the data base. The record does not contain any set members currently
RECURSIVE-ERASE	41000	Erases a record with recursive call due to the presence of member records to be erased along with the owner

Table A.21 Modules in sublevels nine and ten of the structural level

MODULE NAME	MODULE NUMBER	DESCRIPTION
ACCEPT	41100	Finds the logical data base key of the current record of the record type, a set type or of the run unit
CONNECT	41101	Connects the current record of the run unit to a number of sets
DISCONNECT	41102	Disconnects the current record of the run unit from a number of sets
ERASE	41103	Erases a record from the data base. All its member records may be erased.
FIND1	41104	Implements the first format of the FIND statement. It also updates the currency indicators as specified
FIND2	41105	Finds a record with a given calculate key value and a record type
FIND3	41106	Finds a record with the same value as some data identifiers of the current record of the run unit within a given set
FIND4	41107	Finds a record relative in position with respect to the current record of the run unit within a given set
FIND5	41108	Finds the current record of a given set
FIND6	41109	Finds the owner record of a member record within a given set. The member record is a tenant of the set and is the current record of the set type
FIND7	41110	Finds a record of a given record type within a given set type based on certain user supplied conditions
GET	41111	Transfers part of a SLR into the subschema record area
MODIFY	41112	Implements the two formats of the MODIFY statement as specified in the CODASYL COBOL JOD 1976
STORE	41113	Stores a record into the data base

Table A.22 Modules of sublevel eleven of the structural level