A MACHINE INDEPENDENT IMPLEMENTATION OF A DATA STORAGE DESCRIPTION LANGUAGE

by

Anne Louise Zörner BSc.

A thesis submitted to the Council for National Academic Awards in partial fulfilment of the requirements for the degree of Doctor of Philosophy

SPONSORING ESTABLISHMENT: Department of Computer Studies
Sheffield City Polytechnic

January 1987

PREFACE

I would like to dedicate this work to my family and friends without whose patience and love the production of this thesis would have been impossible. I would like to thank Dr. J M Kerridge of Sheffield City Polytechnic and the Science and Engineering Research Council because of whom the undertaking and completion of this project was feasable. I would like to thank Jon again for being a friend and excellent tutor and for not letting go when things were slow and rough. I would also like to thank colleagues both at Sheffield City Polytechnic and at the Rutherford Appleton Laboratory for having faith in me and providing the necessary environment for the completion of this thesis.

ABSTRACT

A Machine Independent Implementation of a Data Storage Description Language

Anne L. Zörner

This thesis presents the methods, results and conclusions of a design and implementation of a Data Storage Description Language (DSDL). The DSDL chosen was the CODASYL Network DSDL. The design supports storage independent manipulation, for access and reorganisation of partitioned schema records, sets and indexes. The production of a Table Generator to compile the DSDL provided the basic structure and mechanisms of a run-time system for the support of dynamic incremental reorganisation. The project developed storage constructs and techniques for a machine independent Data Storage Description Language and evaluated these ideas through an implementation.

The particular objectives of the project included the evaluation of the efficiency of the techniques regarding the criteria of the storage space of tables and records, time for processing, and ease of reorganisation. By developing a run-time system to control dynamic reorganisation of a database given a new version of the storage schema for the same database.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

The concept of the Conference on Data Systems Languages (CODASYL) was formed during discussions of the Common Business Oriented Language (COBOL) in 1959. By 1965 with the success of COBOL under its belt CODASYL was looking to extend its activities; this they did by creating the List Processing Task Force to develop list processing capabilities for COBOL. Since the term 'list processing' did not fit in with the concept of COBOL, the group renamed themselves the Data Base Task Group (DBTG).

The group's first developments, the Data Description Language (DDL) and the Data Manipulation Language (DML), the schema for data description and the language for host access, were presented in an interim report to the CODASYL Programming Languages Committee in 1969{16}. The revised report{17} published in 1971 was to form the basis of what is now known as the network database system model and has been fundamental to several commercial implementations.

As a result of the interest stirred by the DBTG report, CODASYL, in late 1971, formed the Data Description Languages Committee (DDLC) whose initial role was to clarify the functionality of the schema DDL. Two years later in 1973 the DDLC published the schema DDL as a Journal of Development{B4}. At this time the Data Base Administration Working Group (DBAWG), a working group of the British Computer Society (BCS), were incorporated under the auspices of the DDLC.

It was thought to be desirable to have the description of the data separate from the description and control of its storage criterion. Therefore the DBAWG initiated modifications to the schema DDL to remove those storage related aspects, which were present. The DBAWG developed these storage constructs into a language for use by the Data Base Administrator (DBA) which was first published as an appendix to the 1978 DDLC Journal of Development Schema DDL {B5}.

The Data Storage Description Language (DSDL), into which this language developed, provides the DBA with the ability to tune the logical storage capabilities, including indexes, without affecting

the schema, subschema or application programs. The ability to reorganise on a dynamic incremental basis was provided in the 1981 JOD{B6}. It was the aim of the project reported in this thesis to show that this functionality was as feasible as it was desirable.

The development of a compiler/processor for the DSDL was seen as the first aim of the project. Started in 1980 it was hoped to produce two or more variations on the possible implementations of this compiler/processor. The first implementation to produce tables to be stored in some form of 'dictionary', the second as 'assembler' type code with processors to satisfy the DML requests put to the Data Base Management System (DBMS). Using these tables and/or code a simulated run-time system was to be produced to investigate dynamic incremental reorganisation.

There were various stages to the production of the run-time system. The design of the data storage and access methods. The design and implementation of the DBMS, the core of the run-time system. The specification of methods by which dynamic reorganisation could be included. Implementation of these reorganisation facilities, and evaluation of the techniques that were required in the light of reorganisation. Finally to investigate and possibly simulate the effect of reorganisation on the storage structures, system structures and access methods.

Discussions within the DBAWG led to many changes in both the syntax and semantics of the DSDL. These changes were frequently directly related to the correctness of the DSDL definition and as such had to be incorporated in this implementation. This movement in the target and complexity of functionality implied revision of the time scales involved. During the latter stages of the compiler development and the initial DBMS development American National Standards Institue(ANSI) committee X3H2 produced from the 81 CODASYL DDL what has become the Network Data Language (NDL) ANSI standard{B2}. The ANSI version has been adopted by the International Organisation for Standardisation (ISO) as an International Standard Database Language NDL{B20}. The decision was taken by DBAWG to develop the DSDL toward this possible standard. It was felt that the project reported in this thesis could help in this adaptation if the basis was changed to NDL.

However this proved to be an even more mobile target, because of which a fix was taken on the 1983 version of the ANSI-NDL{B1}.

1.1 Structure of the Thesis

Chapter 2 gives a brief description of the various types of database model, and their Reorganisation and Restructuring capabilities, with references to past and present implementations. Presented in Chapter 3 is the functionality of the CODASYL Model including a functional description of the DML, DDL/NDL and DBMS. The DSDL is described in Chapter 4 with particular detail paid to the reorganisation capabilities provided.

The example in Chapter 5 presents the development of an application through the design of a schema and storage schema using the DDL, and DSDL, the effect of the NDL changes are considered, together with the syntactical development of Version 2 of the example.

The Compiler/Table generator is presented in Chapter 6 reflecting on the effect this detailed investigation had on the DSDL.

The run-time system is split into interface and storage mechanisms in chapters 7 and 8, with the effect of reorganisation on the example and run-time system storage structures discussed in chapter 9. The conclusions which have developed during the course of this project are presented in chapter 10.

To understand the concept of a 'database' and the term 'dynamic incremental reorganisation' it is necessary to investigate the various types of database model. Each model is presented together with a comparison of their access and storage methods, with particular reference to implementations. The terms restructuring and reorganisation are defined together with their relationships to the models and implementations.

The term database is said to originate in the 1960's in papers concerning defence systems presented at a symposium sponsored by the System Development Corporation{B26}. The initial definition of the term involved files, entries, keys, and data, but has evolved from simply the means of data retrieval into one which involves storage, definition, access and manipulation of data, and referential and integrity constraints. The latter are rarely part of the integral definition and implementation of a database system.

There are three categories of data model: hierarchical, network and relational.

2.1 Hierarchical

The hierarchical model can be seen as a special case of the network model, where the resultant data structure diagram is an ordered tree or simple set (see Figure 1). Such a restricted data structure diagram is referred to as a hierarchical definition tree. The root nodes can be accessed in order of the hierarchical key by scanning or directly, using either an index or by hashing. From each parent a child record may be accessed either sequentially or via a pointer mechanism. The Information Management System (IMS) is an implementation of such a hierarchical system released in 1968 by Rockwell/IBM and provides HSAM, HISAM, HIDAM, and HDAM access

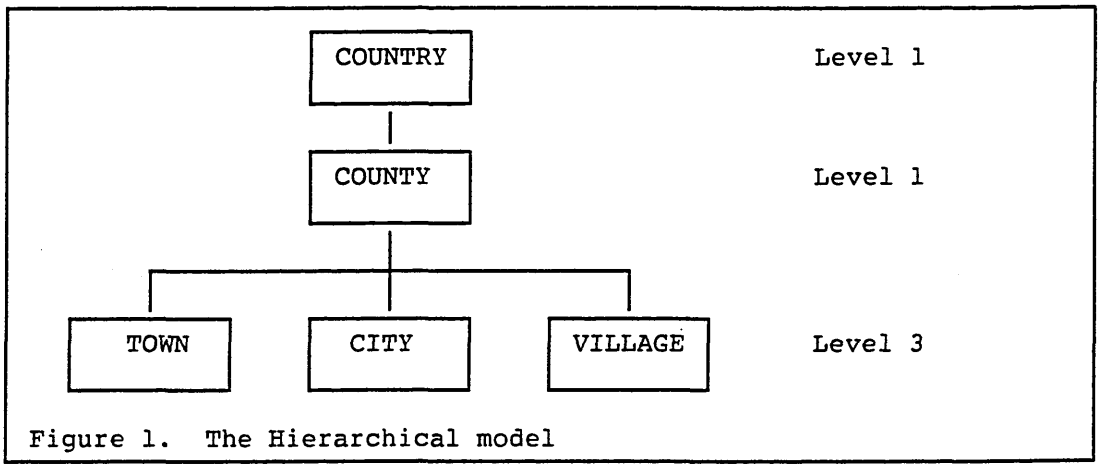


Figure 1. The Hierarchical model

methods{2,6,7,12,14,B35}. Other hierarchical implementations include BDMS{74} and SYSTEM 2000{12} which uses inverted files{6}.

2.2 Network

The network data model provides additional relationships over the simple set of the hierarchy: related entity sets, cycles, multi-participating record to set relationships and the ability to represent many to many relationships. The representation of many to many and cyclic relationships was fundamental to the 1981 CODASYL network model (see Figure 2).

In CODASYL terminology a set type is the definition of the relationship and a set is an occurrence of the set type. A record type may participate as a tenant in many set types, but an occurrence may not participate as a member of more than one occurrence of the same set type. Each set may be an empty set. A set type may be owned by the system in which case there is only one set occurrence of that type. A cyclic relationship is one where a record type acts as owner and possibly indirectly a member within the same cycle of sets. In Figure 2 there are two cyclic sets, namely: the PATIENT to PATIENT and PATIENT, DOCTOR, WARD relationships. There are also two many to many relationships PATIENT to DOCTOR and DOCTOR to WARD. One of these the PATIENT to DOCTOR relationship has already been broken down into two, one to many sets using the link record P-D. Normally in

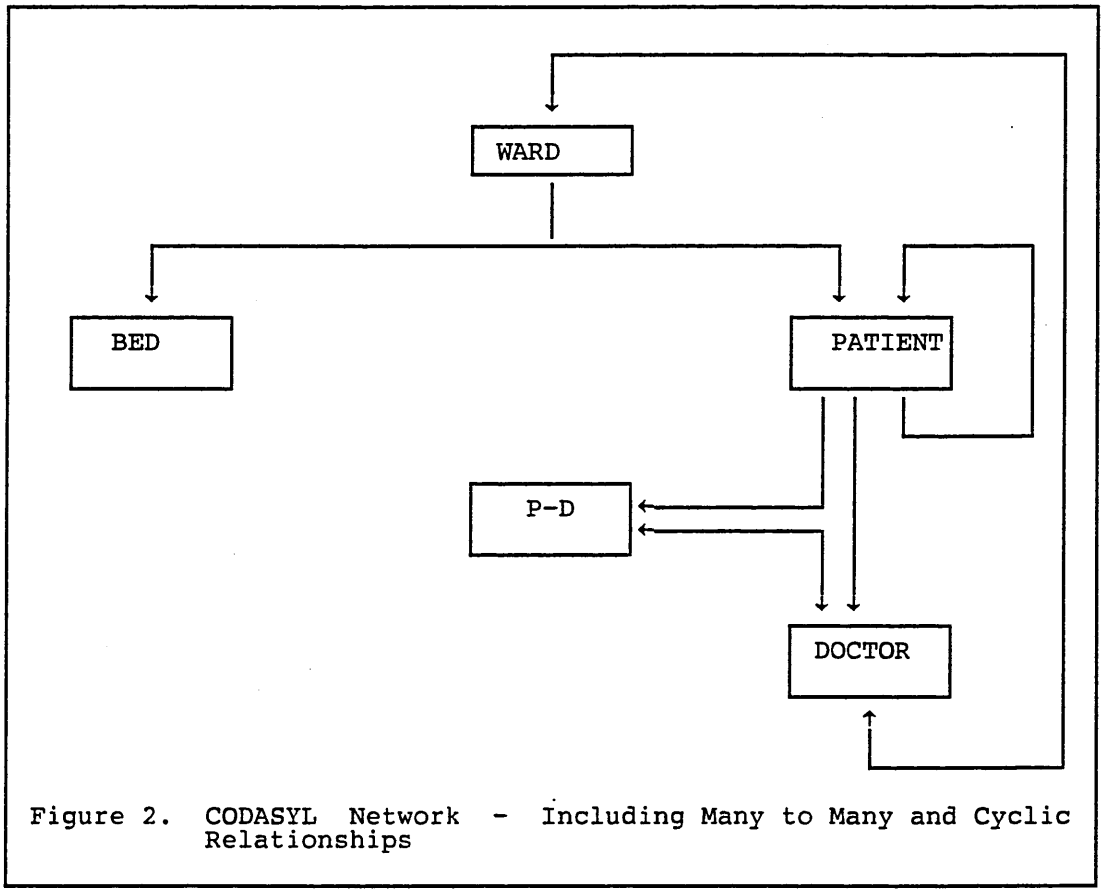


Figure 2. CODASYL Network - Including Many to Many and Cyclic Relationships

the network model these relationships or set types are given names such as DOCTORED BY, where PATIENT DOCTORED BY DOCTOR.

There may be many access paths, and the performance of the database is dependant upon the links maintained. These paths are maintained by the pointer/key system adopted between the owner and member(s) of the sets - where owner and member are particular record types. Detailed expansion of the CODASYL network model follows in Chapter 3.

The predominant disciple of the network model is IDMS {B13,B32,B34}, produced by Cullinane with a second version by International Computers Limited (ICL). The IDMS architecture follows the CODASYL 1971 report of the DBTG. Cullinane has since upgraded it to meet the 1975 CODASYL specification. ICL upgraded their version to include some aspects of the storage schema mentioned in the 1978 appendix to the DDLC JOD, however, it is not known whether this includes any dynamic reorganisation facilities. Indeed the schema still includes area specifications, a storage facility removed from the schema to the storage schema in 1978 as a result of DBAWG representations to

the DDLC. Other CODASYL network type implementations include UNIVAC's DMS1100{B35} (based on the 1969 DBTG report) and BOEING IPAD {B35}.

2.3 Relational

There are now many books and papers written on the subject of relational DBMS's and models, some theoretical others performance related. It is not possible here to develop the total concept of relational theory merely to give substance to the storage perspectives which will be described later. Nor can all books and papers be referenced, however, some of those which either give an overview of all models or performance between models are referenced, from which references containing explicit details may be found.

The true relational model has the underlying mathematical concept of the set-theoretic relation {B34}, which is a subset of the Cartesian product of a list of domains. A domain is a set of values. The members of a relation are termed a tuple, a tuple may be represented by rows of a table, for example SQL/DS{B17} and ORACLE{B24}. All rows should be distinct on the prime key, however, this is not always adhered to, as in IBM's SQL/DS{B17}.

A relational DBMS is one which supports a relational model and its access language may be based on a relational algebra or the relational calculus. The data represented in the model is access path independent, which is achieved by the process of normalisation. The degree of independence is defined by conformity to one of the levels of normalisation. Codd defined three levels of normalisation, Boyce-Codd Normal Form is now named as a fourth {12,14}. Formal definitions of the four normal forms may be found in {B11}. Basically a relation is said to have achieved fourth normal form if:

a. every attribute in a relation is based on a simple domain, i.e. repeating groups have been removed.

b. each non-prime attribute is fully dependent upon every key.

c. all transitive dependencies of non-prime attributes on keys is removed.

d. any multi-valued dependencies which are not also functional dependencies are eliminated.

Third normal form which conforms to the first three points is the most usually adopted form. There are said to be degrees of relational systems, conformity of such systems to the relational data model are categorised explicitly by Schmidt and Brodie {B29} on the work of the Relational Task Group (RTG) of the American National Standards Institute (ANSI). Schmidt and Brodie states that a system should not be called "relational" unless it satisfies the following minimum subset of conditions:

a. 'All information in the database is represented as values in tables.

b. There are no user-visible navigation links between these tables.

c. The system supports at least the select, project, and equi-join or natural join operators of the relational algebra - in whatever syntax is found convenient, but without resorting to commands for iteration or recursion, and with the provision that none of these operators is restricted by whatever access paths have been predefined.'

The result of this investigation of the current system functions was the definition of a language which describes the functionality of a 'tabular relational' database system. This is called SQL after the IBM relational query language SEQUEL (Structured English Query Language) and its developed system SQL/DS (VM/CMS), DB2 (MVS). The language has been produced by the ISO TC97/SC21/WG3 committee as the International Standard, Database Language SQL{B21}.

Of hardware based database systems there appear to be three: IDM 500 (Briton Lee) {B29,12,94,95}; DIRECT (University of Wisconsin) {11}; and CAFS (Content Addressable File Store) from ICL. The relation language of IDM (IDL) resembles QUEL of INGRES.

It is not clear which was the first relational system, however the Peterlee Relational Test Vehicle (PRTV) {B29,B35,12} was said to have four versions from 1970 - 78, and must surely have been one of the first. 1975 had seen the production of RAPID (origins STATPAK)

{B29,12} and MRDS/LINUS {B29,12}. Amongst the next wave were ORACLE {B29,B34,94,95}; PASCAL/R {B29,12,74}; QBE {B29,B34,B36}; ASTRAL {B29}; MRS {B29,12,94,95}; followed by IDAMS {B29}; IDM {B29,12,94,95}; NOMAD and NOMAD2 {B29,B34}; RAPPORT {B29,12}; and SQL/DS{B17} which was based on SYSTEM R{B29,B32,B34,B35,9,10,12,14,20,67,93,94},

Relational systems use a tuple of a relation like an occurrence of a record or a row of a table, and a primary key value could be seen as the owner of a value based set for equivalent foreign keys. The storage adopted by SQL/DS is not far removed from the CODASYL IDMS logical representation, but without the set pointers, in that it puts individual rows in pages in a dbspace, either sequentially or using an index, where a dbspace is simply a logical collection of logical pages taken from a storage pool of VM minidisks. The access methods used are what it terms a relational scan, or a segment scan in System R{9}, where the entire dbspace is searched for all records of the table, order is immaterial, or via a user defined INDEX, which is a b-tree type implementation stored with the table on the same pages. Access should be, according to relational theory, through the Primary Key{B10,B11,B12} however this is not adhered to in SQL/DS where an INDEX can be created on any column or group of columns, uniqueness is not enforced and there can be more than one per table. A simpler implementation by the Science and Engineering Research Council of R-EXEC{66}, based on G-EXEC an earlier implementation at the NERC (1973-1974){40}, uses the one table per file structure and predicate calculus to access tuples from the file, where each file self describes the table structure.

2.4 Restructuring and Reorganisation

It is common to confuse the meanings of 'to restructure' and 'to reorganise' {75,80}, therefore it is necessary to define this terminology in order to provide a basis for the rest of the discussion. The most common, but most underdeveloped{89} concept is that of restructuring the logical entities and relationships of the

schema. For example changing the logical structure of the schema DDL by the addition of a column or the redefinition of a table or record and its set access path mechanisms - in the case of the network or hierarchical models. Obviously the underlying logical to physical mapping is also affected as a consequence of such a restructuring.

The aim of reorganisation, however, is to improve performance by providing hidden mechanisms and storage control to improve the access and storage of the data. As such it should have no impact on the logical structure of the schema or associated application programs.

Conversely a restructure implies changes to be made to the schema, subschemas, application programs and to the storage mapping specifications. Wilson {89} describes some of the problems latent in the restructuring of a CODASYL type database and how neither the structure nor the implementations at the time provided these capabilities. In fact no implementation achieved the full capabilities described by the CODASYL system. It was conceivable that future CODASYL type systems would evolve further restructuring capabilities, however the development of relational systems has reduced the desirability of this functionality because of the necessary complexity of network systems.

Hierarchical and Network based systems such as ICL's IDMS, Cullinane's IDMS and IBM's IMS {89,B13} still require unloading and reloading all or part of the database, to implement restructuring. Restructuring is one of the desirable attributes of a Relational system, where there are none of the restrictions of set relationships between the third normal form relations to be maintained. Systems such as SQL/DS, MRS, NOMAD, PRTV and RAPID all provide the facilities required to restructure. For example in SQL/DS new tables may be created at any time, provided the creator has the necessary permissions, and columns may be added using the SQL command ALTER. Columns may only be added to the end of a stored row/tuple. There are no facilities for changing the implied domain of an attribute.

Reorganisation is the collective term by which none logical changes which improve access and storage of data is termed. There are two distinct types of change. First, strategy reorganisation which is the collective term for those changes used to control the allocation of

record occurrences and set linkages to the storage media space. Secondly, physical placement reorganisation which includes garbage collection of dead space and the re-location of record occurrences to make retrieval more efficient.

Further reorganisation has been more highly developed for network systems. Traditionally any reorganisation that systems provided was statically achieved via the use of unload and reload varying the new storage criterion, strategy 2 as described by Sockut {80,81}, which implies a block on all user access of the data. A variation of strategy 2 is to reorganise in place (strategy 1) which also blocks user read and update.

An alternative is to reorganise the database dynamically with usage, which may be achieved in one of two ways. By a background concurrent utility (strategy 4 of Sockut) which reorganises a finite part of the database allowing access to the rest of the database. Or incrementally as structures are referenced by the system maintaining full user access (strategy 3). The most common functionality provided by systems is garbage collection, many papers on reorganisation {32,79,82,83,92} refer primarily to algorithms and modelling techniques for performance monitoring and improvement of this type of reorganisation.

Factors which differentiate between the types are based on the finiteness of operation, the time of stoppage, locking and journalising requirements and resource usage costs.

Static reorganisation is a clearly defined finite operation it requires application software to be stopped, potentially for long periods, however techniques such as sorting{20} may be used to improve access performance.

Background reorganisation is a clearly defined operation with an explicit termination point. Applications may be run concurrently as data is reorganised however the data being reorganised must be locked from user access, thus requiring enhanced locking and journalising to control integrity.

Incremental reorganisation runs concurrently with applications, data is reorganised, as necessary, when used, there is no explicit termination point. This causes the most commonly used portion of the database to be reorganised first. As will be shown later this can add a considerable overhead to the application access if not held in check. However the overall cost may be less than a total background reorganisation, in overall resource usage. Not all forms of reorganisation may be incremental, for example new indexes must be created statically.

The main reason for choosing the network architecture was the association with and close theoretical knowledge of the DBAWG's network data storage description language, which provided the desired reorganisation facilities. At no time has a commercial system been developed which provides the full range of these database administrator facilities for dynamic incremental reorganisation.

The main elements to the CODASYL architecture are the database management system (DBMS), the associations in the real world described by the schema (DDL) and the storage definition (DSDL) of the entities in the schema.

3.1 The conceptual framework

A rununit is a user activation of an application program. Using DML statements, the rununit makes a call for data to the Database Management System (DBMS). The DBMS has access to the information necessary to process the request via links to the data description and storage description as can be seen in Figure 3. The DBMS processes the request, obtaining further information from the schema DDL and storage schema DSDL, as required.

The DBMS then uses this information to locate the required record. Physical Input/Output operations executed by the call to the operating system make available to the DBMS the data required. The data required is transferred to the User Work Area (UWA) of the rununit originating the call. The DBMS completes the call by providing status information on the results to the rununit. Information may be required by the DBMS of which the rununit has no knowledge.

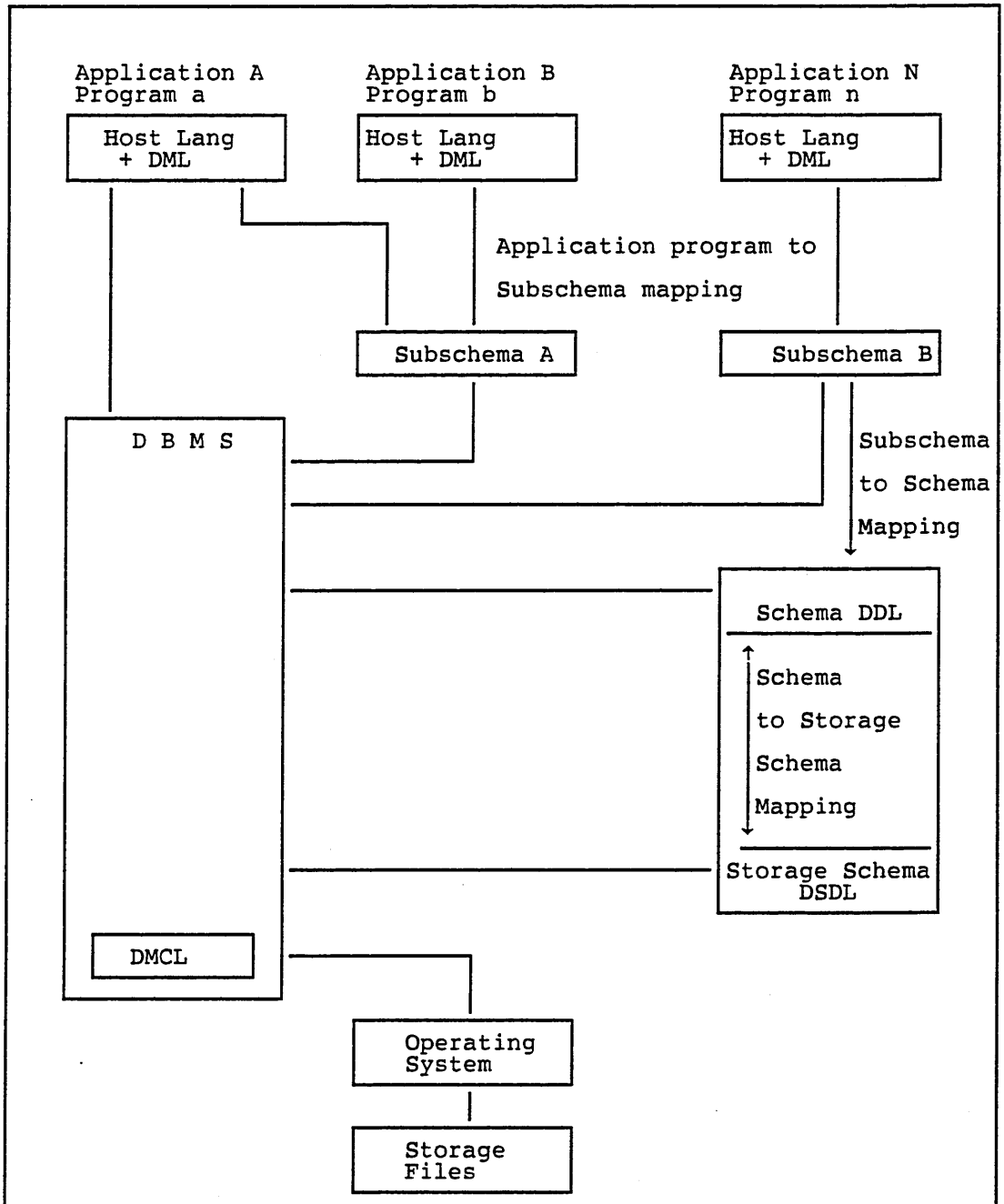


Figure 3. A CODASYL DBMS Architecture

3.2 The Data Manipulation Language (DML)

In commercial systems such as IDMS the Data Manipulation Language (DML) is normally embedded in a host language, such as COBOL. As such it is used to READY the database, or parts thereof, for access via the verbs FIND and GET, FETCH, MODIFY, STORE, and ERASE on records or items and CONNECT, DISCONNECT and RECONNECT on records to sets. The

data used is then COMMITted back to the database, and exited via the FINISH statement. Access is given to the database error conditions and keys.

A record is selected by the use of the record selection expressions within a FIND or FETCH. The selection expressions are used to find the required record for example the NEXT RECORD IN THE SET. Full details may be found in {B8} and are described in {B11,B23,B32,B15,B22}.

3.3 The Subschema/ Schema DDL

The subschema provides the program application view of the database, which may be the same as the schema view. Conversely it may be a subset of the overall schema view, where the schema view could take account of all data which may be held in the future not just that which is currently required.

The schema DDL is used to describe a logical database, and may be shared by many programs written in many languages. This description is in terms of the names and characteristics of the Data Items, Data Aggregates, Records and Sets included in the database and the relationships that exist and must be maintained between occurrences of those elements in the database.

Sets consist of collections of records one of which may be declared as the owner record type with one or more member record types.

A RECORD is described in terms of data items and data aggregates. This description is independent of any host language.

A DATA ITEM is the smallest named unit of data and may logically be of numeric, Boolean, string or implementer defined type. Items can be further collected into one of two kinds of data aggregate, vectors or repeating groups. An item's type, description and source may be specified, details of the clauses may be found in{B5,B6}

A vector is a one dimensional sequence of data items, all with identical characteristics. A repeating group is a collection of data of differing characteristics which occur more than once. The latter facility is provided by the use of the level numbers nested within an occurs clause, the former by the occurs clause on the same level as the item.

A SET is defined by specifying an owner, an ordering method, error and access control routines and member information. A set may be either owned by a record type or the system. A record type may own any number of sets, but an occurrence of a record, may only own one set occurrence of a particular set type. A record occurrence may be a member of only one set occurrence of a set type, but may also be an owner of a different occurrence of the same set type.

The members of a set may be ordered using any of a number of record ordering strategies (eg NEXT, PRIOR) or they can be held in a sorted order{B28}.

Key's in the schema DDL may also be defined for a record type or a set type with participating records ordered by keys.

3.3.1 The Database Language NDL and Associated DML

The Network Database Language (NDL) is the proposed American National Standards Committee (ANSC) Network Database Draft Standard{B2} as devised by the ANSC X3H2 subgroup using the CODASYL 1981 JOD {B5} as their base document. The NDL has since become the basis for an ISO International Standard, IS 8907{B20}.

This is based upon the CODASYL network design but the overall database concepts have changed slightly providing a stricter, more modular approach for the application programmer. This modularisation

may be used to provide an access control tool with a further tool for transaction control.

Prior to its publication, the DBAWG had removed from the base document for the standard , all references to storage and physical device structures and control mechanisms. However the NDL uses the same fundamental concepts, apart from one or two minor constructs, and thus requires underlying logical to physical mapping, either by implementation controls or by using some form of data storage mapping, such as that provided by the DSDL.

3.3.2 The Differences Between DDL and NDL

During the course of conversion of the DDL into the NDL by the ANSI X3H2 Technical Committee on Database, many changes were made. Some of the more important differences which affect the production of an NDL related DSDL, rather than one based upon the current CODASYL DSDL, are: schema record keys, conditional data items, source and result, data check clause on items, null, level numbers for records, and set selection. All of which occur in the DDL but are not supported by the NDL, of these only the removal of set selection, schema record key, and level numbers removes any real functionality from the DSDL, although functionality is removed from the schema. The NDL expanded the idea of sets by including the concept of a structural set.

The removal of set selection implies that the structural application programmer must provide the routes through and the logical maintenance of the network. For a STORE on a record, for example, all the set occurrences into which the automatic member has been inserted must be the current ones of the associated set types. The implementation of the Database Management System (DBMS), using the DDL guidelines, performed this positioning of the set occurrences required, obtaining the item values for the set selection clause from the application work area.

The removal of the schema key phrase for a schema record means that the DSDL can no longer directly provide indexes which support next and prior searches on the record type. (A duplicates restriction could also be placed on a record type for this key). However a similar result can be achieved for the record type, if it is a single member type in a system owned automatic set, with sorted order on the same key. A further limitation is that unique keys cannot be enforced.

The removal of the level numbers implies that all the items are now on the same level, removing the data aggregate and repeating group facility from the schema. Thus enforcing some similarity in data modelling techniques to the normalisation process for relational data systems. For example, the following description of the schema record Tetra, for the entity TETRAHEDRON, can be represented in the DDL by one record Tetral see Figure 4.

```
RECORD Tetral
  01 Weight       FLOAT 6,2
  01 Sides        OCCURS 4 TIMES
    02 Colour     CHARACTER 10
    02 Length     FLOAT 6,2
    02 Angle-Elev FLOAT 5,2
```

Figure 4. DDL Tetrahedron schema description.

In the NDL this can be represented in one of two ways either by a set type and two record types or one large description. The former description is given in Figure 5 by the set Atetra and the record types Tetra2 and Side.

```
RECORD Tetra2
  ITEM Weight       FLOAT 6,2

RECORD Side
  ITEM Colour     CHARACTER 10
  ITEM Length     FLOAT 6,2
  ITEM Angle-Elev FLOAT 5,2

SET Atetra
  OWNER Tetra
  ORDER FIRST
  MEMBER Side
     INSERTION MANUAL
     RETENTION FIXED
```

Figure 5. NDL(1) Tetrahedron schema description.

The latter description is given in Figure 6 as the large and cumbersome record type Tetra3.

```
RECORD Tetra3
  ITEM Weight        FLOAT 6,2
  ITEM S1-Colour      CHARACTER 10
  ITEM S1-Length      FLOAT 6,2
  ITEM S1-Angle-Elev FLOAT 5,2
  ITEM S2-Colour      CHARACTER 10
  ITEM S2-Length      FLOAT 6,2
  ITEM S2-Angle-Elev FLOAT 5,2
  ITEM S3-Colour      CHARACTER 10
  ITEM S3-Length      FLOAT 6,2
  ITEM S3-Angle-Elev FLOAT 5,2
  ITEM S4-Colour      CHARACTER 10
  ITEM S4-Length      FLOAT 6,2
  ITEM S4-Angle-Elev FLOAT 5,2
```

Figure 6. NDL(2) Tetrahedron schema description.

Before 1981 there existed in the DDL the construct of 'set selection is by STRUCTURAL constraint' each member used in this way had to have in its specification a structural constraint clause for equality with the owner's data items. In December 1980 the DDLC removed this facility from the DDL. The similar functionality of the structural set was added to the NDL as part of the INSERTION clause. If insertion is structural, then the owner record is selected by the DBMS to have values of specified data items equal to those of the record to be inserted.

In general terms the CODASYL Data Storage Description Language (DSDL) defines how data described in a schema may be organised in terms of an operating system and device independent storage environment. As specified by the DBAWG charter, the DSDL is merely a tool for the Database Administrator and was designed to affect the performance of an application program but not to alter its results. There is, however, no direct relationship between the data contained in the database and the storage schema declarations except that the contents of a record may affect the storage representation.

This chapter provides an historical and conceptual account of the CODASYL DSDL and its subsequent associations with the ANSI NDL.

The DBAWG was set up jointly by the British Computer Society (BCS) and the CODASYL Data Description Language Committee(DDLC). The historical overview outlines the course of events from 1971 to the present day; however it must be noted that the major thrust of this project took account only of these changes that occurred pre-1983.

The major concepts of the DSDL are then described with reference to the syntactical description of the DSDL which is presented in the form of syntax graphs in Appendix E. The concepts are enhanced by reference to non-dynamic objectives by functional categorisation, followed by detailed discussions on records, sets and indexes. The effect of the NDL on this functional specification is then described.

To dynamically tune the database the facilities of reorganisation, as described in Chapter 2 are required, these aids are described with reference to type and occurrence descriptions.

The chapter is concluded by a discussion on the effect of the NDL on these reorganisation concepts.

4.1 Historical overview

Two conferences were held by the British Computer Society in 1970 and 1971, the former to discuss the 1969 CODASYL Database Task Group (DBTG) report the latter their 1971 report. As a result of these conferences several working groups were established by the BCS Advanced Programming Specialist Group. The forerunner of the DBAWG investigated "Implementations of the CODASYL DBTG proposals".

Developing these proposals further the group contacted and made presentations to the CODASYL DDLC and Data Base Language Task Group (DBLTG). By 1973 the group had turned its attention to the facilities that must be provided by an implementer but were not as yet covered. Thus the group became the "Development of the CODASYL data base proposals". In September of that year the group presented a paper entitled "Facilities for use by the Database Administrator" to the DDLC. Later that year the CODASYL DDLC approved the charter{18}:

"The DBAWG will develop tools for the use of the database administrator to control the efficient and reliable use of the database."

for the Data Base Administration Task Group and asked the working group to become its nucleus. Wishing to preserve their ties with BCS the group called itself the BCS/CODASYL DDLC Data Base Administration Working Group (DBAWG).

In June 1975 the DBAWG produced a discursive account{4} on storage mapping, integrity control, statistics, restructuring and reorganisation. At this time any references to these constructs were explicitly defined in the CODASYL Data Description Language (DDL). The aim was therefore to remove all references to such storage structures from the DDL.

These structures formed the base definitions of the Device Media Control Language(DMCL). Which was later renamed the Data Storage Description Language and published as an appendix to the DDL in 1978{B5}. The reason for the change was because there are two levels below the schema level :

a. The logical description of the data storage comprising the linkages and structures required to support the schema, provided by the DSDL, and

b. the actual device area mappings and physical descriptions.

The DMCL provides the mapping between the DSDL descriptions and the actual physical devices. The DMCL as a concept seen in the 1971 report of the DBTG{17} is still required in conjunction with the DSDL, however it may be embedded in the database operating system.

The DSDL was first published in 1978 and provided mapping and fragmentation only, no reorganisation capabilities were presented. Further enhancements were undertaken to both the DDL and DSDL and these were published in 1981.

Some of the facilities included in the 1981 DDL were: conditional expressions, checks for owner and member records, Boolean and conditional data items. Arithmetic expressions had been added but not published. Those features which had been removed but encompassed within the 1981 DSDL are: areas, measurement, picture clause, structural, tuning from functional and language categories. Also added to the 1981 DSDL was the concept of reorganisation. Altered but unpublished is the reformed set entry syntax which eradicates the conflict between the OWNER and MEMBER clauses. This error was discovered during the course of this project (see Appendix E.2,F,H).

In 1985 with the demise of the CODASYL DDLC the group dropped the term CODASYL from its name. Work progresses not only on storage schemas, but also Access Control, Distributed Databases {45}, and moving towards the relational arena acting as a primary source of British comment on the ISO SQL. Storage schemas for a relational system are also under investigation.

4.2 Static Functionality of the DSDL

The storage schema is divided syntactically into the entries of storage schema, areas, mappings, storage records, sets and indexes as shown by the syntax graphs in Appendix E. The ordering given to these entries is shown diagramatically in Appendix E.2.

Entries for mapping, storage records and sets are subdivided into sub-entries and then all are subdivided into clauses see Appendix E.2 and Appendix F. Functionally the storage schema can be said to be divided into mapping, storage structures, representation, placement and resource allocation categories. Some of these categories are described further through their association with records, sets and indexes.

4.2.1 Functional and Syntactical Description

The main functions of a storage schema are to control the mapping of records to storage structure representations, to control placement and resource allocation of the structures, and maintain the links supporting access mechanisms. The five entries and the functionality they provide are described briefly.

The storage schema entry merely defines the link to the schema and the version of the storage schema. (See 4.4).

A mapping entry defines the relationship between storage records and schema records. A schema record may be represented implicitly or explicitly by a 1:1 mapping and / or explicitly fragmented by a mapping to n storage records. A conditional mapping is one such that more than one explicit mapping is used depending on some condition(s). The schema record Lens (see Figure 7) is mapped conditionally depending on whether it is a macro lens. This figure is taken from the example system which is to be developed subsequently in Chapter 5.

```
MAPPING FOR Lens
IF Macro=true THEN
   STORAGE RECORDS ARE L-Model,Description,Macro
ELSE
   STORAGE RECORDS ARE L-Model,Whole-desc
```

Figure 7. 1:N Conditional Mapping of schema record Lens.

The storage area entry identifies and defines the characteristics of a storage area within the database. The size of this area is defined both by the number and size of a page and whether the area is expandable. Any type of entity may be defined as placed within an area.

The storage record entry defines a storage record type, its contents and the placement criteria of any occurrence of that type.

A set entry specifies how storage records are to be connected to support a schema set.

An index entry names and specifies the name and type of an index and its placement within a storage area. The details of storage records, sets and indexes are now described further.

4.2.2 Entity-relationship support within the DSDL

Entities and relationships are supported in the DSDL by storage records, sets and indexes.

4.2.2.1 Storage Record organisation

A storage record is the logical representation of the physical description of all or a part of a schema record. Only those records which map the whole of a schema record which are said to be mapped 1:1, may be seen to be directly accessible from the schema, however an intelligent DBMS could identify when a subschema record maps

through to a storage record and therefore only return that storage record not the whole of the schema record.

A storage record is defined in terms of the data items, its links to other storage records (within a partitioned mapping), its placement criteria, of, and in the, storage record, and the effects of reorganisation on and of it.

Data items may be explicitly or implicitly represented in a storage record, all items in a schema record may be represented implicitly or explicitly in at least one implicit or explicit supporting record. The position of a data item is controlled through alignment and the format controls its representation.

Link pointers are those pointers between storage records participating in the same multi-mapping. Link pointers are defined by the LINK clause of the STORAGE RECORD clause. These pointers may be direct or indirect, when there is a storage key index for the record pointed to then that pointer is indirect, for details see 4.2.3. These pointers must provide a closed circuit, that is from any storage record within a multi-mapping it must be possible through one or more other storage records to reach any other storage record of that mapping.

An example of such a link mechanism can be seen by the following description of the storage records L-model, Description, Macro and Whole-desc. The mapping of schema record Lens is multi-conditional to L-model, Description and Macro or L-model and Whole-desc (see Figure 7)

```
STORAGE RECORD L_Model
   LINK TO Description, Macro IS DIRECT
   LINK TO Whole_desc IS INDIRECT
   :

STORAGE RECORD Description
   LINK TO L-Model
   :

STORAGE RECORD Macro
   LINK TO L-Model
   :

STORAGE RECORD Whole-desc
   LINK TO L-Model
   :
```

Figure 8. Specification of link for a conditional mapping.

The implication of the INDIRECT on Whole-desc is that Whole-desc will have an index which is used as its storage key index.

Placement of the data items within a record is ordered as within the storage record definition and specified, justified and aligned depending on the clauses of the data subentry.

A storage record may be assigned to an area and optionally page tuned by means of gross placement and fine tuning. The particular strategy of placement may be chosen to be dependent on values within the schema record, this is controlled by the surrounding IF (condition) THEN placement of the Placement Subentry. For example:

```
STORAGE RECORD Description
   LINK TO L-Model
   IF Type='ZOOM' THEN
      DENSITY IS 2 STORAGE RECORDS PER PAGE
      PLACEMENT IS SEQUENTIAL ASCENDING Upper
      WITHIN Equipment FROM PAGE 1000 THRU 6000
   ELSE
      DENSITY IS 6 STORAGE RECORDS PER PAGE
      PLACEMENT IS SEQUENTIAL ASCENDING Lower
      WITHIN Equipment FROM PAGE 1 THRU 999
```

Figure 9. Conditional placement of Description.

Placement is calculated subject to the fine and gross placement criteria subject to the density specifications. However if the fine adjustments, specified by CALC, CLUSTERED or SEQUENTIAL cannot be adhered to then placement is subject to the gross placement criteria of the WITHIN clause. This can occur when there is insufficient space on the page where the record should have been placed.

CALC placement as specified by the CALC clause is calculated from a defined or implied routine, for dispersion randomly about the page range of the WITHIN clause, if specified. This method provides for the fastest retrieval of ad hoc queries provided all parameters of the function are known.

CLUSTERED placement as specified by the CLUSTERED clause provides clustering via a set. If a schema record is the first of its type within that set occurrence then it is placed near the owner, if NEAR OWNER was specified, then the storage record chosen is the one which acts as the destination of pointers. If a storage record of the same type already exists as a member of this set occurrence then the

storage record will, if possible, be placed on the same page as another of the same type representing its closest position in the set order. Otherwise placement is implementer defined. An ad hoc find on a clustered record could be very expensive if no occurrence is known.

SEQUENTIAL placement specified within a storage record description (Figure 9) ensures that within the specified page range subject to the density specifications any record of that type must be stored in such a way that the system may retrieve those records in the manner specified by the ascending/descending key sequence. This storage method need not be physical but may be kept logically by the system. To retrieve such a record may or may not prove to be expensive.

In fragmenting a record to allow optimal retrieval, the benefits must be weighed against the cost required to recombine such a record, taking into account the efficiency of any optimiser.

4.2.2.2 Sets

Relationships are defined in the schema by sets, which must be supported by the storage schema. The means to support the order and content of a set occurrence is defined in a storage set definition by means of pointers, indexes or identifier values.

Two types of sets may be defined: value based or pointer based sets. Value based sets are connected only at request time, pointer based sets are connected by a combination of logical or physical pointers and indexes.

Such connections may be direct or indirect, where indirect implies the support of a storage key index. The storage allocation for a pointer may be fixed or dynamic. Dynamic implies that no physical record space is retained in anticipation of the record having a set where as fixed allocation implies that space is created for the pointer even if the record is not a member of that set type. The types of pointer are shown in Figure 10.

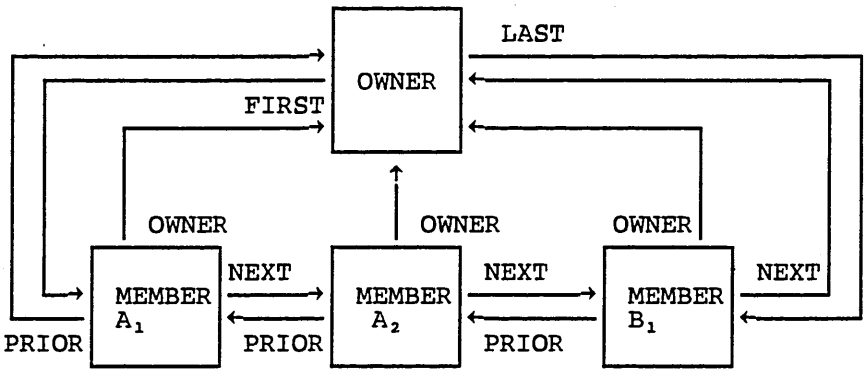


Figure 10. Pointer formations of a logical set occurrence

To support the order of a set occurrence as specified in the schema DDL (see Chapter 3) at least the minimal set of pointers and indexes must be defined. One such minimal set for a non-singular set is an index pointed to by the owner which supports every member record type and owner pointers for each member (Figure 11)

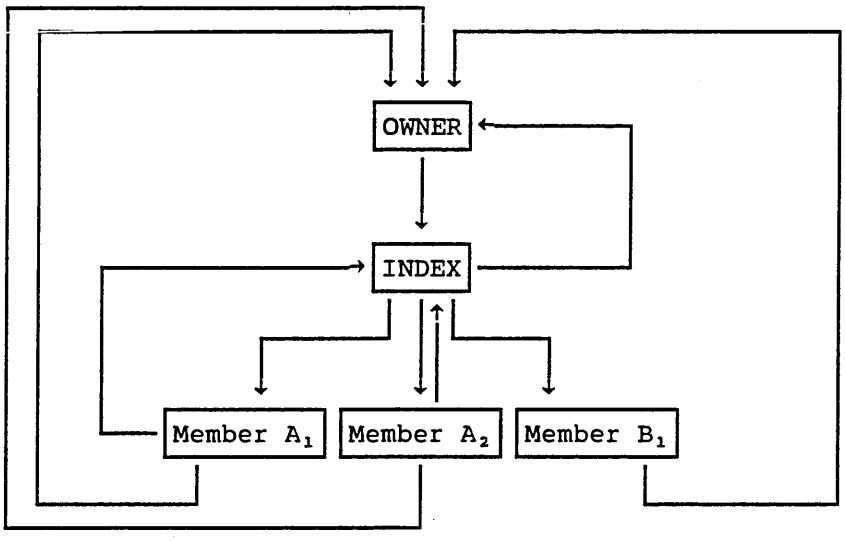


Figure 11. A minimum pointer combination set

The combinations of pointers must be used carefully when supporting the order of a set, it is possible to provide support which will be detrimental to the speed of retrieval. For example a set is completely linked when each member has a prior pointer and the owner has a last pointer. However if it is used to support a set whose order is ascending on a key then to find the first member of the set and follow its sequence requires a highly intelligent system to perform efficiently. In Figure 12 we see that having found the owner

the first record may only be found after passing through the entire chain. It should be remembered also that this simple pass is further complicated by fragmenting the schema records into storage records and the linking provided between them, an expansion of these problems is provided in Chapter 9.

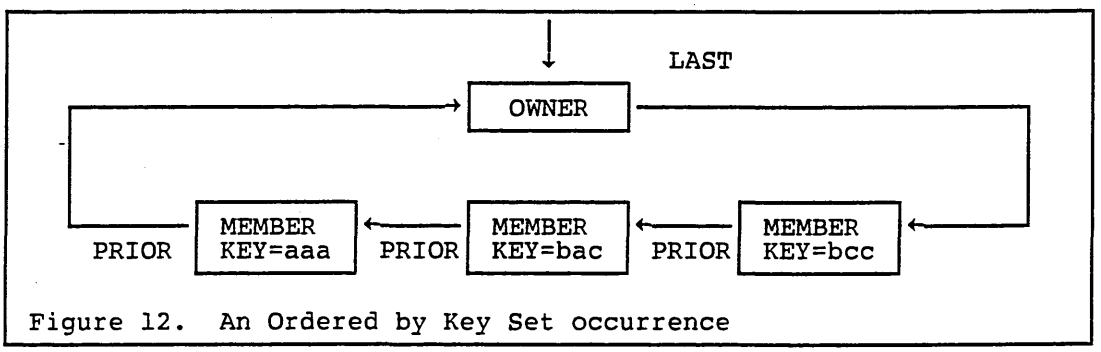


Figure 12. An Ordered by Key Set occurrence

4.2.3 Indexes

There are two types of schema key: those specified to support a schema record and those to support the order within a set occurrence, both types are supported in the storage schema by indexes.

There are three types of indexes; storage key, record key and set indexes.

Storage key indexes are used to support indirect addressing. A storage key points directly to the storage record it is defined as supporting, this index supports only one storage record type. Any storage record linked indirectly to this storage record will not be affected should the record be moved.

Record key indexes support schema record keys or keyed access to schema records using any combination of schema data items, which may imply some order. The latter two providing extra functionality for system record retrieval to that provided by the schema.

Set indexes support schema set representation. Options allow omitted member types, different key types such as the use of sort keys and search keys.

The specification of the USED clause given in Figure 13 is correct according to the syntax and semantics of the DSDL. As the set Manufactures is sorted so all the rules defined in the storage schema (including updates up to December 1980) are adhered to, however after recent debate with the members of the DBAWG it was difficult to see whether the format in the figure is correct or desirable.

```
USED FOR SET Manufactures
     MEMBER Camera
          KEY Model
     MEMBER Lens
     MEMBER Film
          KEY
```

Figure 13. An alternative USED clause to that in Version 1 of the Photographic Storage Schema.

The stored representation of an index is implementer defined however an index is considered to be a sequence of key/pointer pairs or pointers only if no key is involved.

4.3 NDL conformation of the DSDL with the functional changes

As stated in Chapter 3 historically the NDL has as its base the CODASYL DDL, consequently the DBAWG felt it necessary to provide a DSDL which could support a schema defined in NDL. 3.3.2 discussed the changes that were made to produce the NDL, these changes are now discussed as to their affect on the DSDL, and consequently the changes to the DSDL that were necessary.

Facilities which were removed included: schema record keys, set selection and level numbers. The structural set was the main addition.

Schema record keys were supported by the SCHEMA KEY option of the RECORD option of the USED clause. These keys provided a direct link between record access and the indexes of the storage schema, any link between records and indexes for records must now be calculated by an optimiser.

Set selection of the schema provided logical access routes which had to be maintained by the DBMS. The cascade effect of this automatic search and selection had a definite effect on reorganisation, this will be described later in 4.5.

The use of level numbers in the DATA subentry of STORAGE RECORD, enabled the DBA to specify a part of a hierarchy/repeating group with or without specifying individual data items, now the specification of DATA ALL is only at the whole storage record and therefore whole schema record level.

The initial removal of the structural constraint concept by the DDLC caused the value based option for a storage set to have a tenuous place in the storage schema. The rewrite of the syntax rules of SET for value based sets was not resolved. The replacement of insertion by structural set in the NDL provided an excellent hook for value based sets. The main syntax rule specified that value based could only be used if each MEMBER clause of the set type

a. as a structural specification and

b. has a retention clause for specifying FIXED or MANDATORY.

It is not necessary to have a structural set supported by a value based rather than a pointer based structure.

4.4 The DSDL Reorganisation Facilities

'Reorganisation' is the ability to alter the physical structure or access support mechanisms of the database. These changes have already been categorised in Chapter 2 into two types of reorganisation; Strategic and Physical Placement and the methods by which these changes are performed were described as STATIC, DYNAMIC BACKGROUND and DYNAMIC INCREMENTAL reorganisations.

The specifications of the 1981 CODASYL DSDL have been defined so it is possible to use strategic reorganisation and each method of

reorganisation as circumstances dictate. The separation into logical and physical description by schema and storage schema provides the necessary environment for both static and dynamic reorganisation to be performed without affecting the logical structure or application programmes.

The DSDL uses what are termed 'versions' of a storage schema to enable a system to provide reorganisation. A DSDL has a syntactical construct called a VERSION clause. A reorganisable object when initially defined is given a version number which is equivalent to the current version number of the storage schema via a VERSION clause. The current version of a storage schema contains all versions of the object definitions, and the new changed definitions. An object's definition may not change between versions unless it is given the new version number. A version of an object may not be omitted from a storage schema unless all previous definitions are omitted and there are no objects currently stored using that version. However, areas must be statically changed as they do not have versions.

With this version control the DSDL provides the following types of strategic reorganisation.

a. Changes to the mapping of record types on to storage areas, including fragmentation, distribution and data item storage specifications.

b. Changes to the mapping of set types on to storage areas including, access through the storage structure of a set type and the ORDER and SORTED clause storage structures.

c. Changes to the mapping of record occurrences and indexes onto storage areas including, record and index occurrence placement.

d. Changes to the placement of objects relevant to each other.

The DSDL does not provide the means by which garbage collection and index compaction may be controlled.

These changes are now explored by means of the functional descriptions of the objects: Mapping, storage area, storage record, set and index, the triggers of reorganisation of objects will be explored in Chapter 8.

4.4.1 Mapping

MAPPING controls the mapping of storage records to schema records. A new mapping may be an extension of an old mapping or a replacement. Unconditional may become conditional, simple one to one may be fragmented. Any new storage record participating in a mapping must be defined as the object of a storage record entry. The new mapping may also cause new versions of; sets for which this record is a member, indexes used for this schema record, and any storage record which now participates in a more complicated mapping.

4.4.2 Storage Area

A change to a storage area must be incorporated immediately, this is a static reorganisation, but may be done in background (strategy 4 of Sockut) provided it is defined before population.

4.4.3 Storage Record

There are three types of change for a storage record which leads to reorganisation; changed linkages, placement and data storage description. If a storage record now participates in a group or enhanced group mapping then links have to be defined to complete the group linkage. Any indirect pointers to new storage records would require the definition of new storage key indexes. The number of

pointers reserved may change should the number of pointers declared as "allocation dynamic" be altered.

There are no restrictions other than the normal DSDL static rules that a change of placement must abide to. Changes may be to either or both of the gross or fine placement strategies, for example a different area or SEQUENTIAL changed to CALC. Unconditional placement may become conditional to relieve storage pressure points.

Data storage description is also only limited to schema definition restrictions, in that a record could include more or less all of, specify transformations for and include filler space between the items. The space occupied by the record could change, thus implying a change in placement.

All the types of change to a storage record, basically imply a re-calculation of the storage record, unless complicated pre-optimisation is performed.

4.4.4 Storage set

The definition changes to a storage set are based around the linkage mechanisms adopted and the storage records that are used as the containers of the pointers. A set may be changed from value based to hard linkage based, the linkages may be all pointers or a mixture of index(es) and pointers. The storage records which act as the destination and location of the pointers may be changed to others of a mapping or replaced entirely. Whether the pointers always occupy space is dependant on the allocation method specified for each POINTER clause. If the destination record pointer type is changed to indirect a storage key index specification is required.

There are four parts to an index specification; the naming clause, PLACEMENT clause, USED clause and WITHIN storage area clause. The rules for reorganisation state that any new version of an index must contain an identical USED clause (see 3.7.1 INDEX Syntax Rule for Reorganisation 4{B6}), the other clauses may all be changed. Only

those indexes used for a non-singular set may be reorganised dynamically, all other forms must be reorganised statically or in the background.

Fine placement changes may be specified for indexes used to support a non-singular set. The Gross placement within an area (and optional page range) may be specified for all types of indexes, but only those used to support a non-singular set may specify STORAGE AREA OF OWNER.

The necessity for an identical USED clause to be specified has some unforeseen implications, for example any storage record which participates as a destination record for a DIRECT POINTER from an index used for RECORD, may not later be changed to be the destination of indirect pointers from anywhere.

Those records which have an index with the option USED FOR RECORD specified may not be changed to a new mapping, unless a method of removing indexes is provided. This is because an object cannot be removed from the storage schema until there are no occurrences using that version's description, and an index once created can merely have new versions which for example have a different placement strategy.

Let us take a schema record Shop and map that onto two storage records Gen-det and S-addr. A schema key Shop-code for Shop can be supported by the index Ind-shop, with direct pointers to the storage record Gen-det. In version 2 another set named Inventory needs reorganising and may well require the owner record Shop to be moved about. For this purpose a storage key index is defined for Gen-det. This would then imply that pointers to Gen-det in the schema key index Ind-shop would have to be changed to be indirect, a possibility not allowed for in the rules which govern the USED clause.

To show another example of this inflexibility let us say that in version 1 we have a default mapping for the schema record Manufacturer and an index which is USED FOR RECORD Manufacturer as in Figure 14 The default MAPPING is MAPPING FOR Manufacturer STORAGE RECORD Manufacturer.

```
INDEX Ind-man
    USED FOR RECORD Manufacturer
         SCHEMA KEY Manu-code
    WITHIN M-s
```

Figure 14. Version 1 of the index Ind-man

For Version 2 we try to create a new mapping for Manufacturer (Figure 15) using the storage records Man-addr and Man-det and then change the index to point to the new storage record Man-addr.

```
MAPPING VERSION 2 FOR Manufacturer
     STORAGE RECORD NAME Man-addr,Man-det
     :

INDEX Ind-man VERSION 2
     USED FOR RECORD Manufacturer
          SCHEMA KEY Manucode
          POINTER TO Man-addr
             :
```

Figure 15. An invalid specification for Version 2 of Manufacturer

However this has involved changing the USED clause to include a POINTER part, which according to the rules of reorganisation is not allowed, however recent discussions of the DBAWG agreed that this inflexibility was not desirable and that merely the type of index should not be changeable. For example Ind-man created to be used to support the schema key Manucode could not be changed to be USED FOR STORAGE KEY on Manufacturer.

4.5 The effect of the NDL on reorganisation - its simplifying factors

As was described in 4.3 the changes required of a DSDL to support the NDL rather than the DDL were few. The effect of the NDL on the reorganisation facilities provided by the DSDL was merely consequential, in that the removal of set selection from the NDL removed the automatic search criterion for the current set. When a system has to provide navigation through the sets to find the current

set then there was the possibility of providing set reorganisation of the pointers involved, which could cascade through all the records of the set occurrences in question. One of the reasons for this is the necessity for preserving the connectivity of the set occurrences. Therefore one of the ways in which reorganisation could be detrimental to an insert command is removed. Detail of the trigger effect one reorganisation may have on another is provided in Chapter 8.

5.1 A Conceptual Photographic Schema

It is hoped that by providing an example of sufficient complexity the major facilities, together with their advantages and disadvantages may be demonstrated. The wish of the author is that the unusual nature of the example, a pet subject, should not deter a reader from understanding the complexities but provide an interesting background. The nature of the example has been adapted to demonstrate the facilities of the DSDL, but not to describe a complete application development.

There are many entities which could be included in an example based upon photographic supplies and suppliers. The entities; Camera, Lens, Film, Chemical, Manufacturer and Shop have been chosen to best demonstrate the features and capabilities of the DSDL. However, all possible relationships between the entities have not been fully developed as this would only confuse.

Lenses may have different Mounts which allow them to be used on different Cameras. Processability of Films by Chemicals is dependant only on manufacturers' specifications as to suitability for example colour slide films of the E-6 type should be processed only by chemicals of the E-6 process.

All these items form part of the inventory of at least one shop.

All Items are produced by a single Manufacturer, but a Manufacturer may produce a variety of Items.

These entities and relationships together form the photographic schema which is represented in the Figure 16 and expanded into the specification of the schema DDL given in Appendix A.

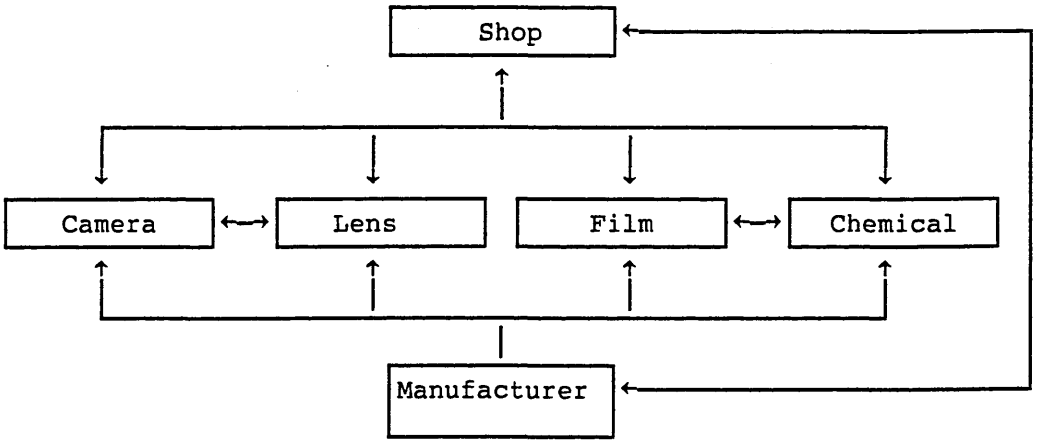


Figure 16. Photographic Entity Diagram

5.2 The Photographic Schema

In order to map the photographic entity diagram in Figure 16 onto a network schema each many to many relationships must be converted to a one to many relationship. Each relationship may then be represented by a set.

The environment from which this example is taken includes many other entities and relationships, these include such entities as distributors and sub-manufacturers. Some manufacturers often produce the same items but under different labels. These relationships are ignored here. In addition not all relationships between entities present in the schema are mapped. The relationships which are present in the conceptual schema are represented as sets in the specification of the Photographic Schema (Figure 17)

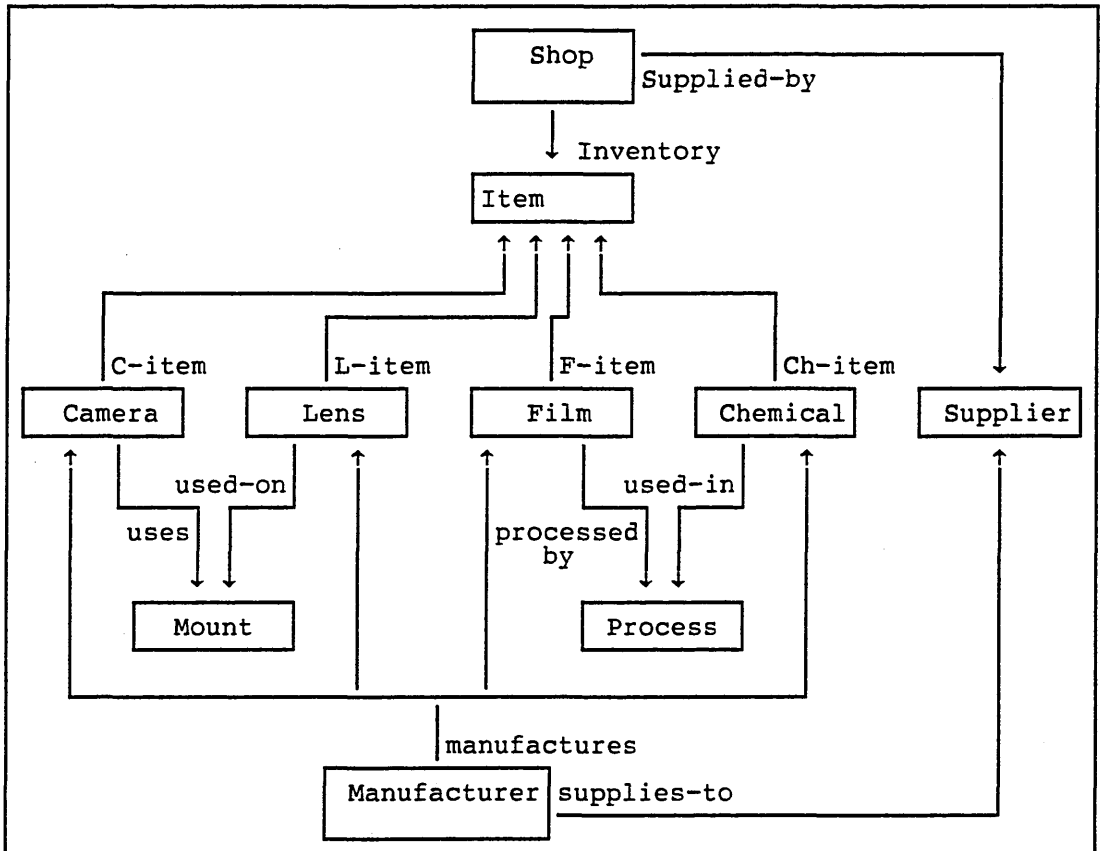


Figure 17. Photographic Schema

There are factors other than the representation of entities and relationships which affect the specification of the DSDL. These include the specification of schema keys for records and order on members of a set. Set selection however may be simulated by occurrence driven selection, and since it has been removed from the NDL we will not consider it in this example. Schema keys for records have also been removed from the NDL but these are a feature it was felt necessitated inclusion here, in order to provide simple direct keyed access. The ordering and key types for sets have been chosen with the specific intent of enabling a full description of the complexities and effect of reorganisation to be demonstrated.

5.3 The Photographic Storage Schema

It is envisaged that a storage schema could be designed in order to improve the access times for queries, or the storage of records, or

both. However the time required to store records might be sacrificed to improve retrieval times. Let us say that for this example we wish to provide a retrieval oriented database. Some of the queries could be :"What chemicals can I use to process this film?" "What can I purchase in that shop?"; "What lenses can I buy for this camera?". These queries could be refined using item values for selection criteria.

Let us first examine the query "What chemicals can I use to process this film?" In order to find the chemicals we need first to locate the record which details which type of film it is that we wish to process. The records Film, Process and Chemical are simple records for which we will default the mapping of each onto its default storage record. The entry is through the Film and therefore we will use CALC placement for quick retrieval of the record Film. To ensure that the Process records are near by that can be used on this film, we use set CLUSTERING near owner via the set Processed-by, and store the Chemical records so that they may be found through the set Used-in or scanned using the SEQUENTIAL order of Type and Make.

```
STORAGE RECORD Film
     DENSITY ONE STORAGE RECORD PER 3 PAGES
     PLACEMENT CALC USING ASA
     WITHIN M-s FROM 1 THRU 4999
     DATA ALL

STORAGE RECORD Process
     DENSITY 3 RECORDS PER PAGE
     PLACEMENT IS CLUSTERED VIA SET Processed-by
     NEAR OWNER
     WITHIN F-ch FROM 1 THRU 4999
     DATA ALL

STORAGE RECORD Chemical
     PLACEMENT IS SEQUENTIAL ASCENDING Type, Make
     WITHIN F-ch FROM 5000 THRU 9999
     DATA ALL
```

Figure 18. Version 1 of the storage records Film, Process and Chemical

If the query had merely requested the process by which this film could be developed, then the query should be answerable from the Process record without recall to the Chemical records.

The records within the set Processed-by are stored in the default order with the simplest link mechanism of FIRST and NEXT pointers.

The details of the Chemical is then found by retrieving the owner of the set that this Process is Used-in.

```
SET Processed-by
     OWNER
          STORAGE RECORD Film
          POINTER FOR FIRST
     MEMBER
          STORAGE RECORD Process
          POINTER FOR NEXT

SET Used-in
     OWNER
          STORAGE RECORD Chemical
          POINTER FOR FIRST
     MEMBER
          STORAGE RECORD Process
          POINTER FOR NEXT
```

Figure 19. Version 1 of the sets Processed-by and Used-in

The storage area mapping then looks like that depicted in Figure 21 in the storage area F-ch, where all Films, Chemicals and the linking Process are stored. The legend given in Figure 20 identifies the characters used to define the pointers for the storage records in the figures. Figure 21, Figure 24, and Figure 27 detail the storage of records within the areas as defined for version 1 of the example storage schema(see Appendix B). The storage area is depicted by the large box inside the figure, in which each schema record is depicted by a box internal to the storage area. A mapping to the schema record, is then represented internally by its constituent storage records and the links between them. At this stage no record is divided across storage areas. Set linkages are represented by FIRST, LAST, NEXT and PRIOR pointers or the destination record for all OWNER pointers. Those pointers which are across storage areas are named externally to the storage area box but internal to the figures.

```
O - Owner
F - First          N - Next
L - Last           P - Prior
        I - Index
        D - destination
```

Figure 20. Legend for Figure 21 Figure 24 Figure 27

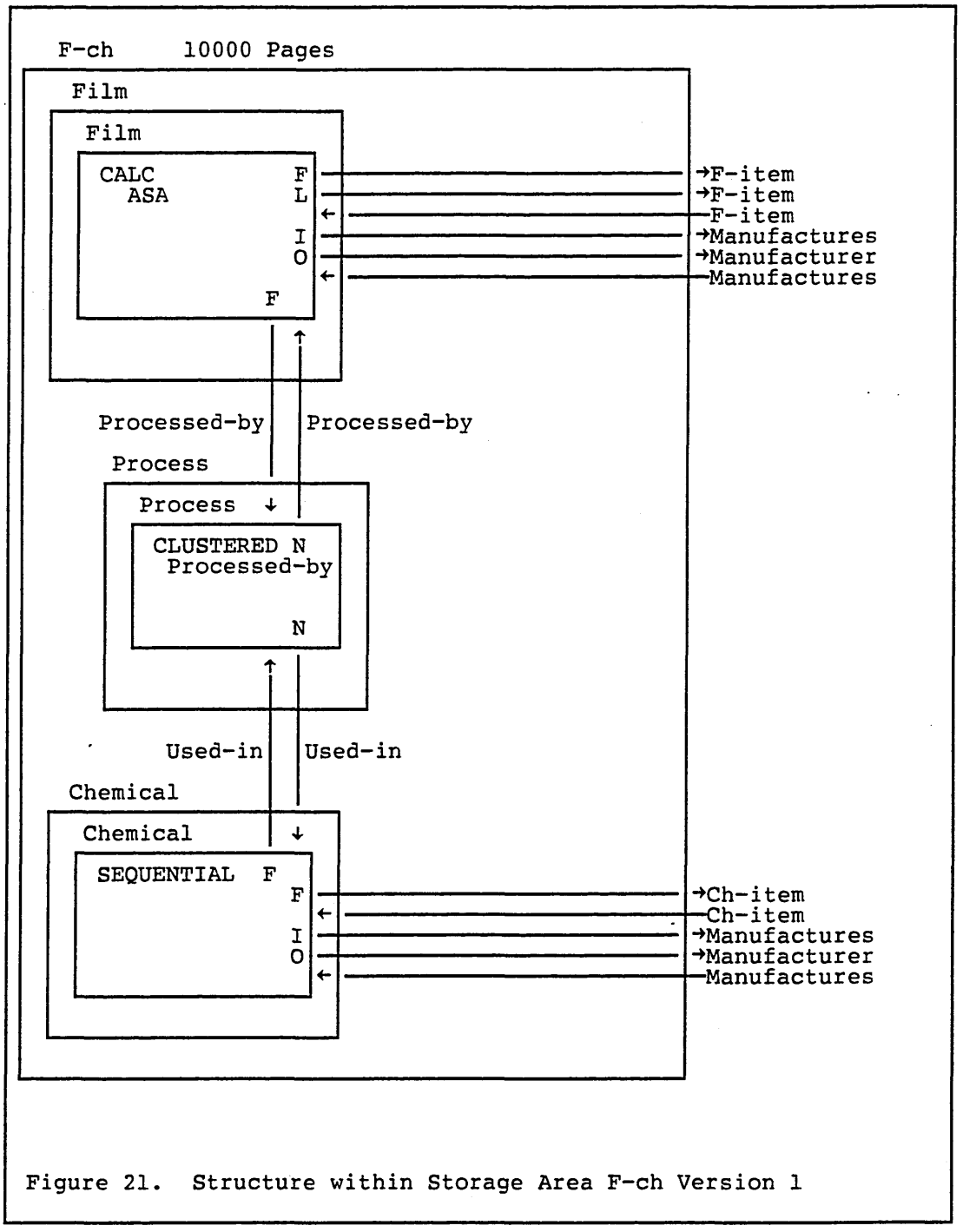


Figure 21. Structure within Storage Area F-ch Version 1

Let us examine the necessary criterion for a storage schema which supports the query "What items can be purchased in that shop?". Another way of looking at the query is to say "What is that shop's inventory?". Let the query be supported by access from the Shop record but that all the necessary information for this query can be satisfied by access to the Item records in the set Inventory. This should just produce a list of the items but no details. The details

may be found by locating the owners, the actual Items, through the sets F-item, C-item, Ch-item and L-item.

The shop could be found either through its Town or Chainname by splitting the record into two storage records, Gen-det and S-addr. Let Gen-det hold the general information such as the chainname of the shop and the VAT number, and S-addr all the address information.

```
MAPPING FOR Shop
     STORAGE RECORDS ARE C-model,Mod-det

STORAGE RECORD NAME IS Gen-det
     LINK TO S-addr IS DIRECT
     PLACEMENT IS CALC Chain USING Chainname
     WITHIN M-s FROM 500 THRU 999
     01 Chainname
             :

STORAGE RECORD NAME IS S-addr
     LINK TO Gen-det
     PLACEMENT IS CALC Town USING Town
     WITHIN M-s FROM 500 THRU 999
     01 Address
          DATA ALL
```

Figure 22. Mapping and storage record specifications for Shop.

Both storage record types are placed in M-s, the area that holds the information on Manufacturers, Shops and Inventory information.

Having found the Shop, using the CALC key Chainname, the set Inventory is searched. The set can be searched in Item-code order, the defined order as specified in the schema, either ascending or descending because a full pointer specification in the storage schema can be specified as in Figure 23

```
SET Inventory
     OWNER
          STORAGE RECORD Gen-det
               POINTER FOR FIRST,LAST MEMBER
               DESTINATION OF DIRECT POINTERS
     MEMBER
          STORAGE RECORD Item
               POINTER FOR NEXT,PRIOR TENANT
               DESTINATION OF DIRECT POINTERS
```

Figure 23. Set Inventory version 1

A similar query "Which shops does a Manufacturer supply?" has the same format, although this time a different placement strategy has been used. The intermediate record Supplier must be used to fulfil

the many to many mapping between Manufacturer and Shop. The resultant storage area configuration is then given in Figure 24.

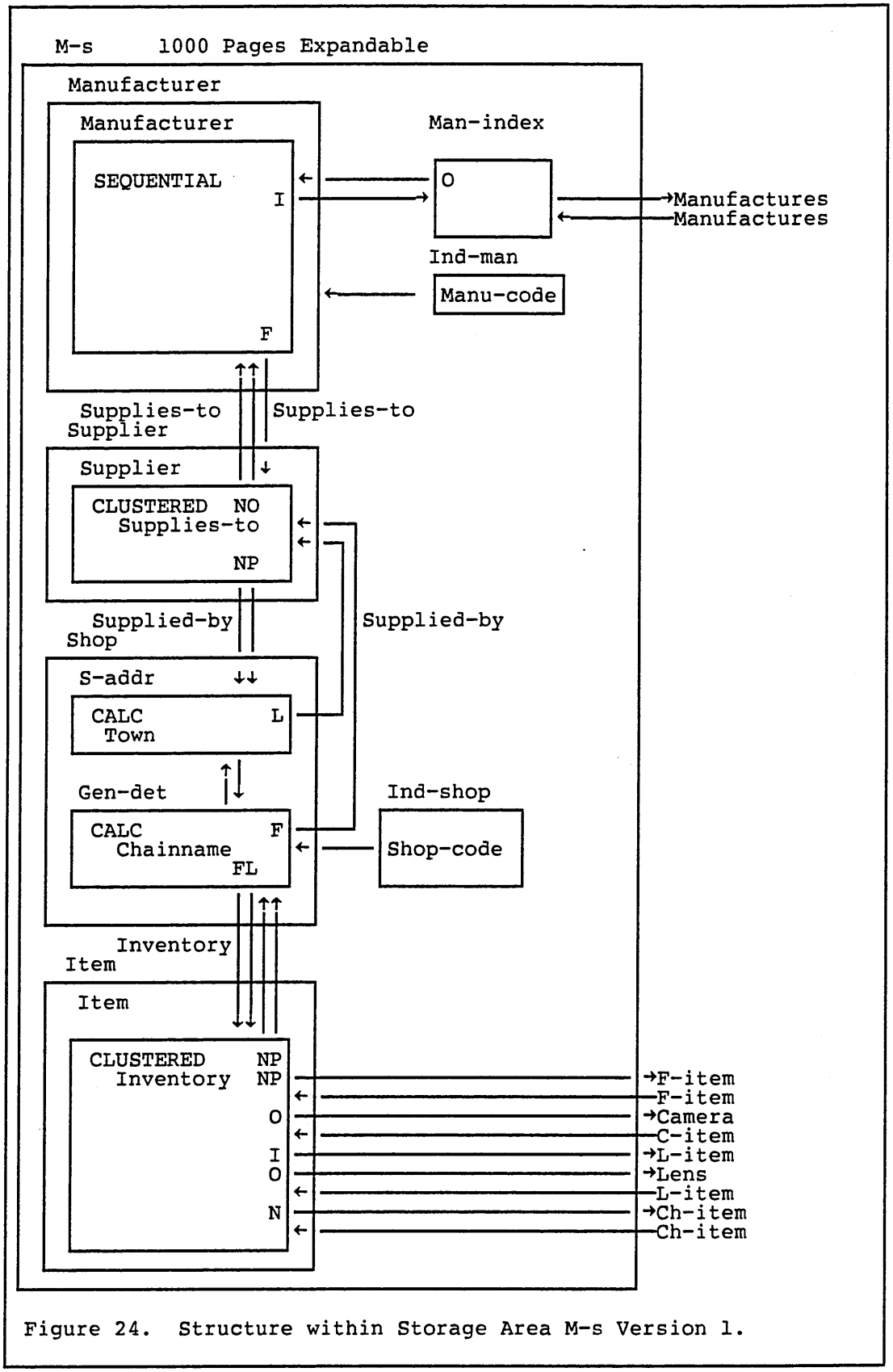


Figure 24. Structure within Storage Area M-s Version 1.

For our last query we will examine the query "What lenses can I buy for this Camera" in order to discover whether Tamron produce a Lens to fit my Camera. The link between Camera and Lens as shown in Figure 16 is many to many and had to be replaced in Figure 17 by the two sets Uses and Used-on, where Camera Uses Mount and Lens is Used-on Mount. The set name Used-on is required to represent both fixed and separate mounted lenses. It is likely that many types of query will require entry through Cameras and Lens', to this end each has a complicated mapping allowing storage and possible retrieval using CALC keys via Mode or Model for Camera and via Model or Min-max for Lens.

```
MAPPING FOR Camera
     STORAGE RECORDS ARE C-model,Mod-det

MAPPING FOR Lens
     IF Macro= 'TRUE'
     THEN STORAGE RECORDS ARE L-model,Description,Macro
     ELSE STORAGE RECORDS ARE L-model,Wholedesc
```

Figure 25. Mapping Description for Cameras and Lenses Version 1

To enable this multi-mode retrieval, multiple mappings were used (see Figure 25). In Figure 26 the storage records and indexes that are required to support the mapping of Lens are shown.

```
STORAGE RECORD C-model
     LINK TO Mod-det
     PLACEMENT IS CALC Mode USING Mode
     WITHIN Equipment FROM 1 THRU 499
     01 Brand-name
     01 Model
     01 Mode OCCURS
     01 Rec-retail-prc

STORAGE RECORD Mod-det
     LINK TO C-model
     DENSITY IS 5 STORAGE RECORDS PER PAGE
     PLACEMENT IS CALC Model2 USING Model
     WITHIN Equipment FROM 499 THRU 998
     01 Model
     01 ASA-range
       DATA ALL
     01 Speed-range
       DATA ALL
     01 Flash-sync-speed OCCURS
     01 Rec-Retail-prc

INDEX Cit-index
     PLACEMENT IS NEAR OWNER DISPLACEMENT 2 PAGES
     USED FOR SET C-item LINK TO OWNER
     WITHIN STORAGE AREA OF OWNER

INDEX Ind-cam
     USED FOR RECORD Camera
          SCHEMA KEY C-name
          POINTER IS DIRECT TO Mod-det
     WITHIN Equipment
```

Figure 26. Storage records for Camera and its indexes

If a similar exercise is done for schema records Mount and Lens producing the syntax as in Appendix B, then we could represent the resulting structure of the storage area Equipment and the object placement within it as in Figure 27.

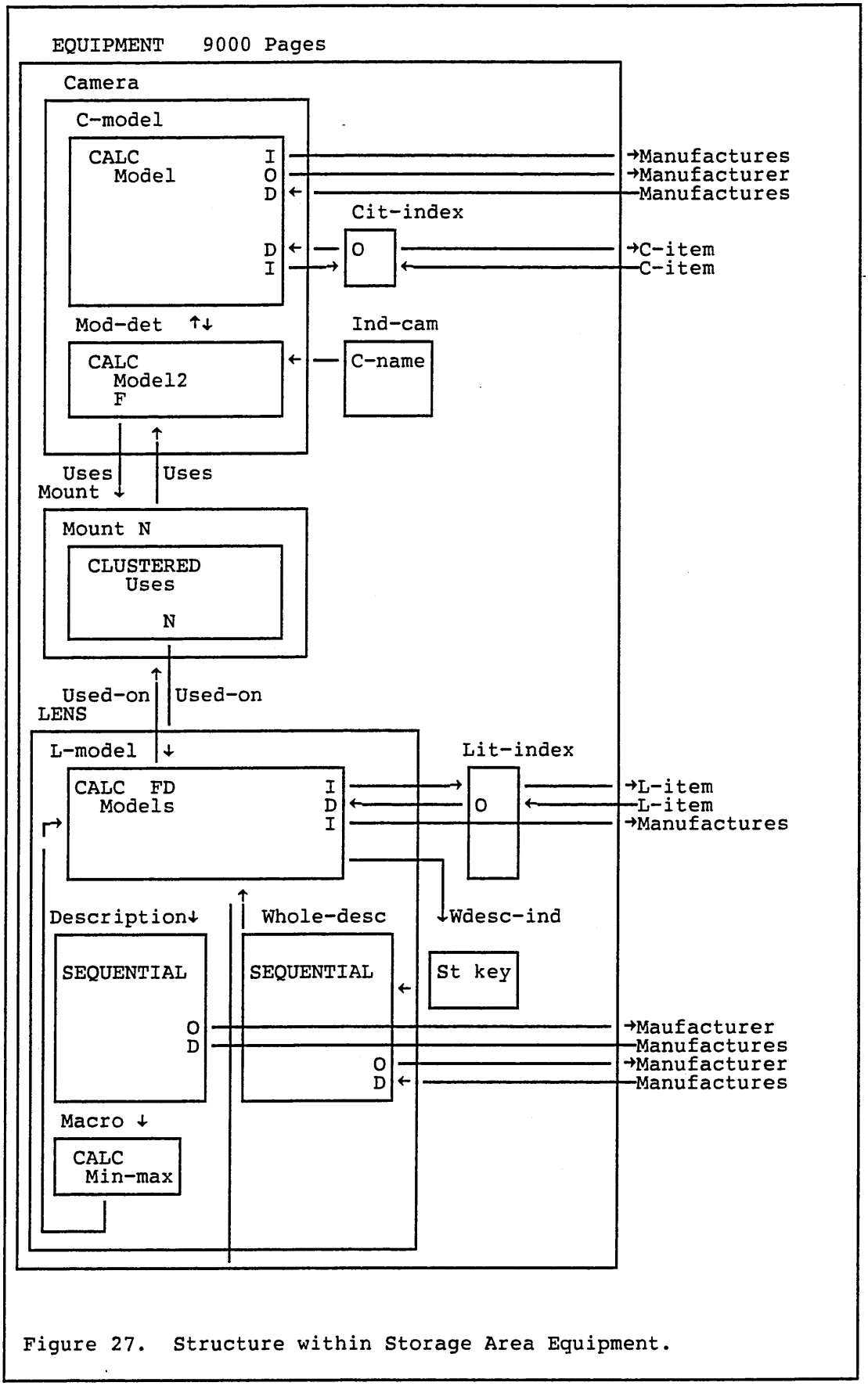


Figure 27. Structure within Storage Area Equipment.

5.4 Version 2 of the Storage Schema

Let us now investigate the syntax change requirements for two types of performance tuning. The first is that storage area M-s has become overcrowded, the second that the access time required to find where a particular chemical can be bought is too great.

The former requires that some record type, or part of, be removed from the area M-s. From Figure 24 we can see the possible records and indexes that could be involved. Since the other areas are possibly full enough we will create a new storage area. Let this storage area be named Items, with the definition as specified in Figure 28

```
STORAGE AREA Items
   INITIAL SIZE IS 1000 PAGES
      EXPANDABLE
   PAGE SIZE IS 256 WORDS
```

Figure 28. Storage Area Items

In order to populate this area and relieve the pressure within M-s, the record type Item was chosen for reorganisation. However this record has close links with the schema record Shop through the set Inventory. It was therefore decided to create a new multiple mapping for Item(see Figure 29).

```
MAPPING VERSION 2 FOR Item
   STORAGE RECORDS ARE S-item, I-item
```

Figure 29. Mapping for schema record Item

The two storage records S-item and I-item are defined so that S-item stays near the Shop and I-item is placed in the new area. Neither could be left to default, since by default all items would be in each storage record, and this would not relieve the space. The definitions for the two storage records are given in Figure 30.

```
STORAGE RECORD S-item VERSION 2
   LINK TO I-item IS INDIRECT
   PLACEMENT CLUSTERED VIA SET Inventory
      NEAR OWNER
   WITHIN M-s
   S-name
   S-code-no
   S-price

STORAGE RECORD I-item VERSION 2
   LINK TO S-item IS DIRECT
   PLACEMENT IS
      CALC CalcIcode USING Item-code
   WITHIN Items
   Item-type
   Item-code
```

Figure 30. Version 2 Description of new storage records S-item and I-item.

The definition of the mapping of Item requires that some set definitions require redefining. This is due to the explicit use of the destination of set pointers being defined in storage record Item version 1. Any new record of the type Item to be stored will be stored as the 2 storage records S-item and I-item, where I-item has a storage key index associated with it. The destination clauses of the affected sets must therefore be changed to point to S-item or I-item either implicitly or explicitly. The implicit record is S-item, which is the record first in the list in the mapping set. The new sets for C-item and L-item are specified in Figure 31 and implicitly, similar alterations for the sets Ch-item and F-item as for C-item.

```
SET C-item VERSION 2
   OWNER
     STORAGE RECORD C-model
        POINTER FOR INDEX Cit-index
        DESTINATION OF DIRECT POINTERS
   MEMBER
     STORAGE RECORD S-item
        POINTER FOR INDEX Cit-index, OWNER

SET L-item VERSION 2
   OWNER
      STORAGE RECORD L-model
        DESTINATION OF DIRECT POINTERS
        POINTER FOR INDEX Lit-index
   MEMBER
      STORAGE RECORD I-item
        DESTINATION OF INDIRECT POINTERS
        POINTER FOR INDEX Lit-index, OWNER
```

Figure 31. Version 2 of the sets C-item and L-item

The set Inventory has the minor change for Version 2 to point to the new destination record S-item. The storage key index defined for I-item is specified in Figure 32.

```
INDEX Iit-ind
    USED FOR STORAGE KEY I-item
    WITHIN Items
```

Figure 32. New storage key index defined for I-item

The latter type of reorganisation, that was required to improve the access search time of the schema record Chemical arises due to the only applicable access method being via a sequential search in storage area F-ch from pages 5000 thru 9999. This was not thought to be a popular enquiry.

Any change which includes the definition of multiple storage records might require changes to the sets Used-in, Ch-item, and Manufactures. Let us assume that the only change we wish to make is to define a new placement criteria to Chemical, that of CALC within the new storage area Items. The method used to do the CALC requires as input the name of the chemical process. The resulting change is simply a new version of the storage record Chemical(see Figure 33).

```
STORAGE RECORD Chemical VERSION 2
   PLACEMENT IS CALC Codes USING Code
   WITHIN Items FROM PAGE 500 THRU 1000
   DATA ALL
```

Figure 33. Version 2 of storage record Chemical

The consequences of these changes will be explored in Chapter 9.

The models which provide the format for a database system to incorporate dynamic incremental reorganisation have been described in chapters 3 and 4, the example which will be used to demonstrate a systematic course through dynamic reorganisation was introduced in chapter 5. Chapter 6 introduces the implementation part of this project describing the DSDL implementation undertaken as the first part in the production of a system to support the network database languages.

Given the DSDL {B5} the primary target was to design a three pass compiler, using standard compiler techniques{B16}. However due to hardware and software limitations, in particular no universal lexical or syntax analyser like LEX or YACC{51} , and the quantity of intricate semantic rules required the three passes became four: Lexical Analysis, Syntax Analysis, Semantic Analysis and Code/Table generation. For reorganisation it was seen that the structure of the output of the Code/Table generation needed to be finalised before the historical checking required for version analysis could be processed. Version analysis therefore formed a fifth pass. The comparison of the current storage schema and that proposed requires that the two must be of comparable format to this end the current tables are required together with the symbol tables from the dictionary. Code/Table generation and version analysis formed a macro-processor because like Pascal P-code{1,36} once the intermediate language is obtained from the syntax analysis phase it is possible to create any type of code or table generator as required.

The output from the version analysis becomes on initialisation into the runtime system the new storage schema provided the reorganisation rules were followed.

Each process does not begin unless the previous pass was error free. Errors in lexical, syntax and semantic analysis do not call a halt to the pass merely note their presence and find the next continuation point. Table generation will have no errors other than of capacity.

Version analysis reports errors pertaining to reorganisation and ensures that the new storage schema cannot be used.

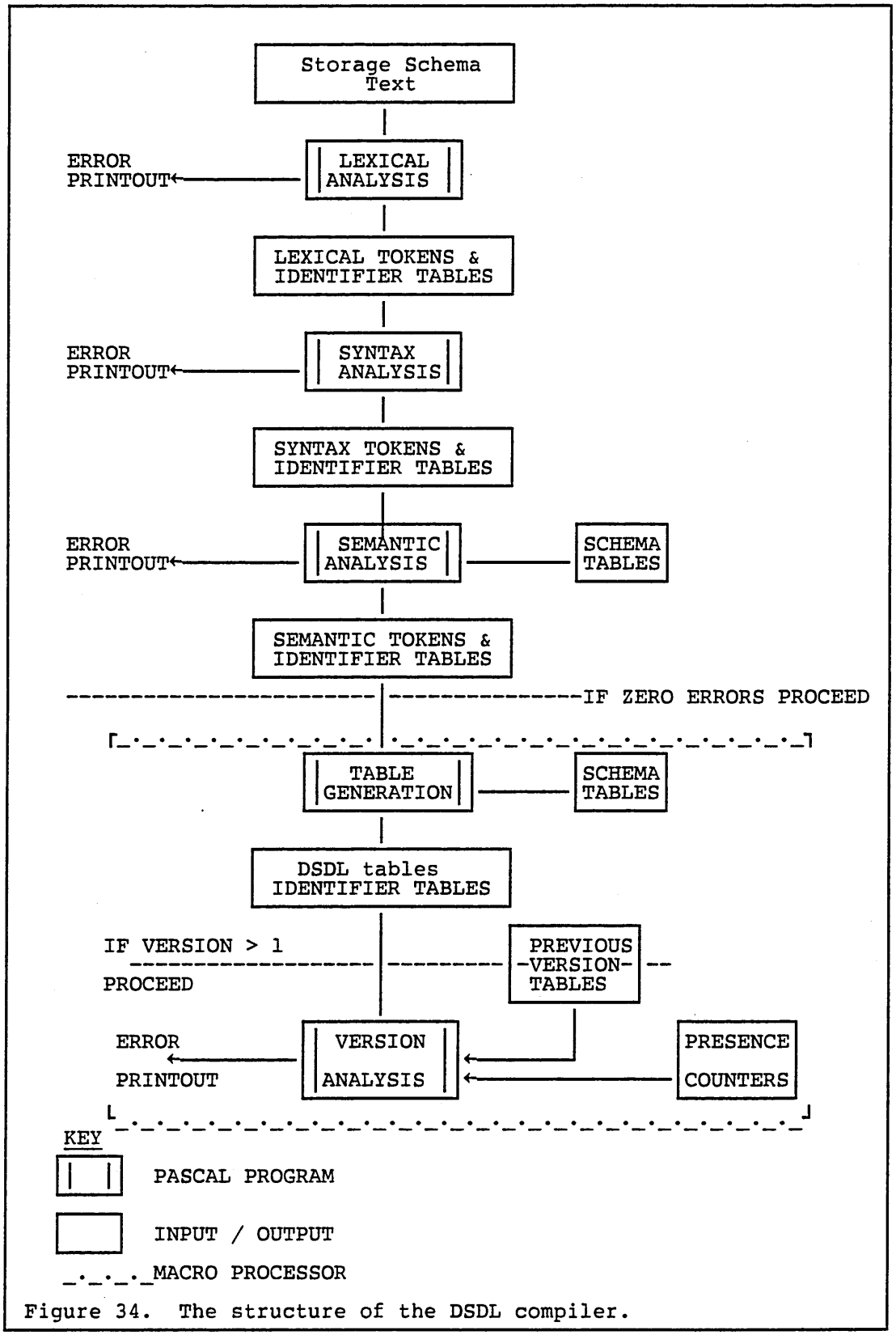


Figure 34. The structure of the DSDL compiler.

6.1 Lexical Analysis

The input to Lexical Analysis is for example the file of characters which form the text for the example Photographic Storage Schema in Appendix B. There were two ways in which to approach the design of lexical analysis one to enforce type dependence on variables the other type independence.

The latter ignores at the lexical analysis stage the types of the identifier, adopting one symbol table where the type of an identifier and its duplicity are realised in syntax analysis. Since a name may be used to represent more than one type, the syntax checking for such a symbolic language would be more complex. Names may also be used before they are defined (by juxtaposition) as being of a particular type.

The former adopted view required a symbol table for each identifier type, this method solved the need to repeat the syntax/lexical checking for duplicate names in syntax analysis. Simple syntax juxtaposition was required to identify the types of each identifier.

The initial representation of the internal symbol code was to consist of integers, negative for reserved words, positive for identifier types which were followed by the symbol table code.

The numeric representation had to be changed to an internal symbol set representation due to Pascal case statement requirements. The types of table considered for the symbol table were hash, binary and b+-trees{5}. At this stage it was decided that a binary tree system would be the simplest to implement, since it was not a requirement, it was thought, to delete symbols from the tree. The numeric symbol of a name was created from its depth and breadth within the tree.

The output of lexical analysis was therefore the reserved word and identifier trees, and the symbol stream.

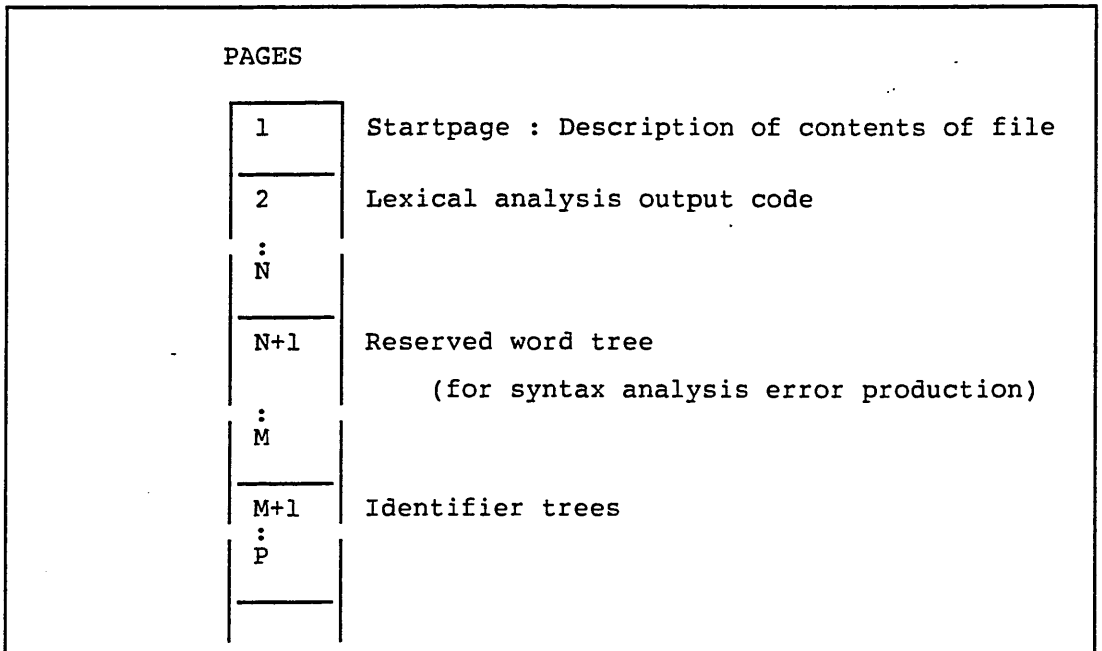


Figure 35. Lexical Analysis Output file structure

6.2 Syntax Analysis

Syntax analysis checks the symbol stream and produces a syntactically correct reduced symbolic representation of the DSDL source text. It requires as input a file which contains the symbols, reserved word tree and identifier trees, and produces as output a file containing the representation of the DSDL and the identifier and reserved word trees.

The file is direct access and controlled via the first page where the internal structures of the file are described.

The input symbols are of the form described as output from lexical analysis. They are passed through a syntax checker, constructed from the syntax of the DSDL grammar. Tables were produced to check that the grammar was LL(1), with a few suggestions to DBAWG of alterations this was achieved, and the results passed as papers (Appendix H) to the DBAWG.

The syntax of the DSDL can be seen from the graphs (Appendix E,F) to be logically divided into entries, subentries, clauses and or phrases. Once these entries had unique starting points (from LL(1) derivation) they provided the structure for the syntax analyser.

The main routine is simply :

```
Storage Schema Entry;
Mapping Entry;
starters = first in storage area entry
CASE symbol in starters
  area: storage area entry (starters);
  record: storage record entry (starters);
  set: set Entry (starters);
  index: index entry (starters)
end;
```

The starters are those reserved word or identifier symbols which represent those words which are valid next in the syntax. The graphs (see Appendix E) were used in conjunction with LL(1) grammar theory to obtain the possible starters, for example those following mapping entry were different to those following an index entry. The starters were therefore controlled on exit from an entry.

In places within the symbol stream are structures called conditions they have the same formulae throughout the DSDL, and were treated as a separate routine. The syntax for a condition was changed for the DDL in 1980 to include arithmetic expressions. The resulting syntax was not LL(1) and therefore required an intermediary rollback facility. Implementation of the checking of this syntax was via a stack.

The reserved word and identifier trees required by lexical analysis were passed to syntax analysis for use in error analysis, they then form part of the output to semantic analysis.

The output file is of very similar structure to the input file except that conditions and subscripts are now kept as separate tables within the file.

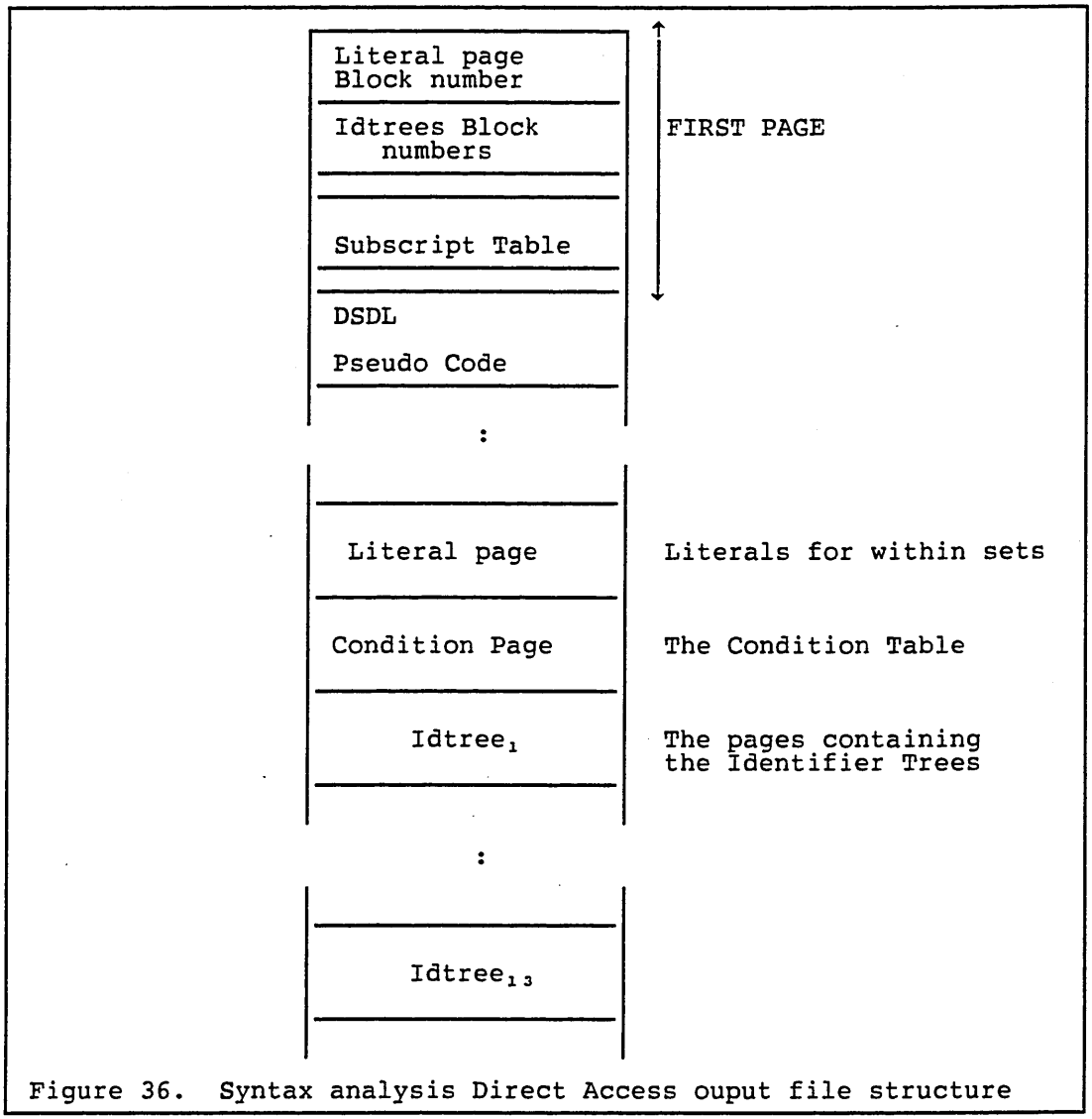


Figure 36. Syntax analysis Direct Access ouput file structure

6.3 Semantic Analysis

As has been described in Syntax Analysis the structure of the input file is formed from a string of integers which are mapped to form, a first control page, the syntax and identifier trees, the condition and subscript table and the symbol stream itself. Also required as input to semantic analysis are the schema tables. These may be accessed via internally declared routines or else by the use of the DBMS routines. To simplify the procedure, since the DBMS routines had not been defined it was decided to keep the routines internal but such that they could easily be replaced by calls to the DBMS since

apart from these externally defined controlled tables the first three passes form a machine independent macro processor, it is not significant how the tables or code would be defined and stored. But the same guidelines of syntax were used to simulate the DDL tables by inputting a stream of integers with integer references to a list of identifiers, and storing them in internal tables, in this way the syntax and semantics of the schema had to be assumed to be correct.

The semantic rules were obtained from the syntax and general rules for each entry, clause and so forth. To enable some of the rules to be fulfilled temporary Pascal structures were created which simulated the structure so that the connections between entries could be checked. For example 3.5.6 LINK Syntax Rule 1{B6} and rule 35 in the graphs in Appendix F.

"Storage-record-name-1 must appear in the same STORAGE clause in a Mapping Subentry as the storage record specified in the STORAGE RECORD clause to which this clause is subordinate"

As has been stated in Chapter 4 and can be seen from the basic syntax structure of the DSDL{B6}, there is no forced ordering on the entries. For where indexes and sets are concerned, the only rule is that no storage records may be defined after an index entry. This forces the check of 3.7.3 USED Syntax Rule 7{B6} to be at the end. The rule states that:

"An Index Entry defined with the SET option must be specified for each index type named in a Set Entry POINTER clauses for the set type named by schema-set-name-3."

It may be implemented by storing the names of those set types and the index name required, and then when an index used for SET for that index name is found a flag is set to say that the index was found. The list may at the end be traversed to check that all the required indexes were declared.

During the plotting of the rules to the syntax it was noted that some rules of a fundamental nature were missing, others such as the Member Ambiguity(Appendix H) were more inherent to the correctness of the semantics.

The output of the semantic analyser is the unchanged input file.

6.4 Table Generation

The fourth pass produces the final form of the tables and or code in the format required by the DBMS. Checking of the reorganisation criteria is omitted until the fifth pass Version Analysis.

Table Generation requires as input the output file from semantic analysis and the schema tables. The schema tables are required for direct substitution of subscript to definition of a schema record as defined in the schema and so that default mappings and storage records may be defined in the storage schema tables. Initially two forms of output were designed, tables and code since tables correlate closer to the design of the language, it was decided to develop a version of table generation. Another factor in the decision was the suitability of storing the information in a type of data dictionary. The tables are set up containing pointers to link across the tables. These are changed to integer array references for output. The reason for the change is the none transportability between programs of Pascal pointers.

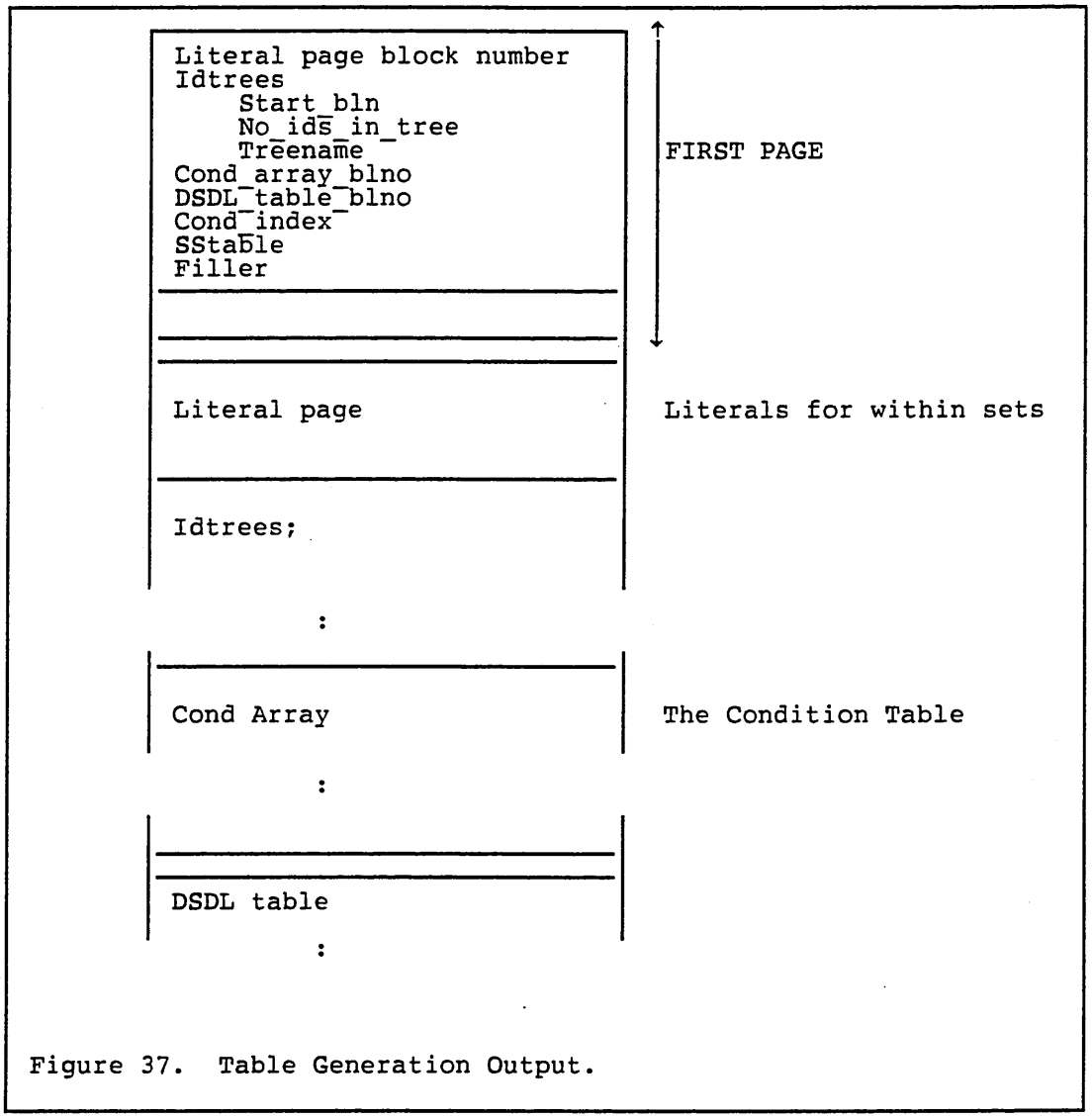


Figure 37. Table Generation Output.

The output is therefore tables arranged as an array of array structures and mapped onto a two dimensional array from a file of integers. Integers are used here because only tokens are held in the tables, real data is held in the identifier trees, condition and subscript tables.

6.5 Version Analysis

Similar in structure to semantic analysis this pass is completely dependant on the form of the output from table or code generation. The output from table generation and the previous version of the

storage schema compilation are compared according to the rules of reorganisation specified in the DSDL. The input is of code or table orientation for both the current version and the previous version of the storage schema. This pass is only applied to storage schemas of version 2 or more.

The rules of reorganisation are of 2 types - the syntax rules for reorganisation which control the creation, alteration and deletion of the elements, and the general rules for reorganisation, most of the general rules apply to the actions of the DBMS, however one set state that:

> "it is an error to remove an 'object' version from a storage schema if the database contains pointers to 'objects' of that version."

If the counters are supplied, of the current state of the contents of the database, to version analysis then this error can be checked at compile time rather than run-time.

This pass therefore performs a simple presence check for all objects both for old definitions and new definitions in the proposed storage schema. Counters relating to each object and the versions used are provided as output from the runtime system. All an objects versions prior to the current version must be in the new storage schema or else their counters must be zero. If an objects version is removed then all previous versions must also be removed from the proposed storage schema as well as all their counters being zero.

Required also is the USED clause check, mentioned previously in section 4.4, this should check that the format of a used specification does not change between versions of that index. Discussions which have taken place recently show that although that was the letter of the rule it was not the intention, and that this check should be replaced in version analysis by a simpler one. In that only the type specified; storage key, record key, or set should not be allowed to change.

The idea of a compiler making reference to output from a previous compilation and checking the relationship is understood to be totally novel, no reference has been found of a similar occurrence.

To simplify the comparison between versions, the new identifier symbols had to be converted to be comparable with the old symbols. If access to the DBMS is available then use of the system symbols would be most beneficial, thus avoiding a double conversion of the symbols.

7.1 Introduction

The DML verb is used as the basis in formulating the system response. The DML rules which call upon the DDL and DSDL table information which form the bases of the run-time system.

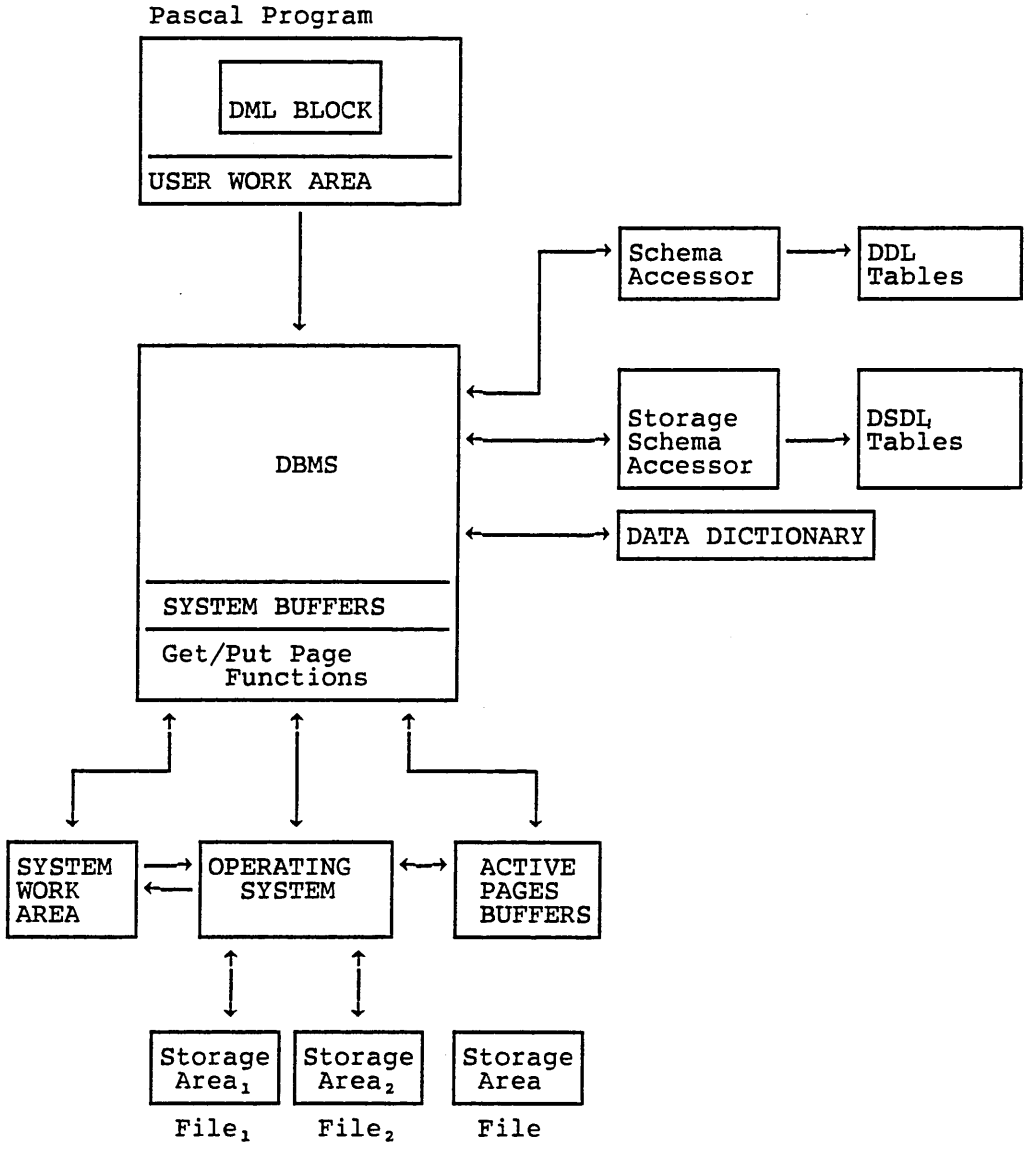


Figure 38. The experimental runtime system

The diagram(Figure 38) shows how the tables developed during the table generation phase are used from the run-time system via the Storage Schema Accessor. The run-time system will also be instrumental in achieving dynamic reorganisation. During access to records the run-time system will automatically convert existing storage records to the latest version. No such functionality currently exists in any commercially available software.

The application passes to the system details of the DML verb it requires to be executed, together with a pointer to an area containing information to support the request. The system processes this request, converting it to a request for information from the DDL and DSDL tables. The system uses this information to send an appropriate request to the GET PAGE routine to obtain the required page, via the operating system, from storage. This could have required other pages, for example index or associated pages through pointers.

At the time when this run-time system was designed the facilities for intercommunication between virtual machines in a VM environment were not provided. After further investigation it was decided that the complexities involved in locking and 2 phase commit were too complex to be undertaken here. Without these facilities the production of a multi-user system would have no meaning. Rollback, although more complicated for a multi-user system, should also be provided for any single user system. However without some form of logging this proved difficult.

Therefore the system was designed as a single user system consisting of a series of modules, the schema tables and storage schema tables being resident in memory, but the run-time DML routines had to be overlayed. This chapter describes the interfaces, exposing the relationship between the run-time system; the DML, the Schema Table Accessor, the Storage Schema Table Accessor, and the operating system.

7.2 The DML simulator interface

In order that a DML compiler/analyser did not have be written, because of time constraints, a solution was adopted which involved designing what could have been the output block description of a DML analyser. This block description was then used by a Pascal module which performed calls to the run-time system modules in the guise of DML calls.

The DML verbs perform the following groups of actions

a. Retrieve a record by some access path

b. Store a record by means of the criteria specified in the DDL and DSDL.

c. Change the contents of one or more data items in a schema record.

d. Alter set criterion.

e. Commit or Rollback changes.

The DML verbs used to retrieve a record are FIND, and FETCH, FIND merely locates the record, FETCH makes the values available to the application. The access path to the record is initialised by the type of record selection expression used.

For example a record selection expression of the form "ANY recordname USING idlist", is of the 'adhoc' type form. This is shown by the example in Chapter 5 for finding a particular shop record in order to find out what it sells. If there is no schema key for this idlist then it is hoped that the list fulfills the placement value criteria.

The record selection expression "PRIOR recordname WITHIN setname" indicates that we have a current pointer on that set and we wish to find or fetch the record PRIOR in the set order. The efficiency of this request will depend on whether the set is stored with PRIOR pointers or an index which can be so used.

If the record selection expression had specified NEXT instead of PRIOR then we could use it to complete the example referenced above by finding all the Items in the set Inventory, using NEXT calls, that this Shop sells.

Let us now examine the basic structure of the DML verbs in order to define the structure of the run-time system. For example the verb STORE has the structure:

```
STORE recordname [ RETAINING [| RECORD  |] CURRENCY ]
                             [| setname |]
```

Analysis showed that the block structure in Figure 39 was all that was necessary to support this and any other DML verb and was therefore defined using the Pascal block structure given in Figure 40. @Workarea(Figure 39) points to the area into which either the retrieved record was placed, or, in the case of a store, where the data to be stored was put by the application program.

In order to obtain values of the identifiers specified in the record selection expressions, the structures that mirror the schema records must also exist in the user work area. The values are then used to fulfill the record selection criterion. It is these values which must be used to satisfy any chosen storage access path.

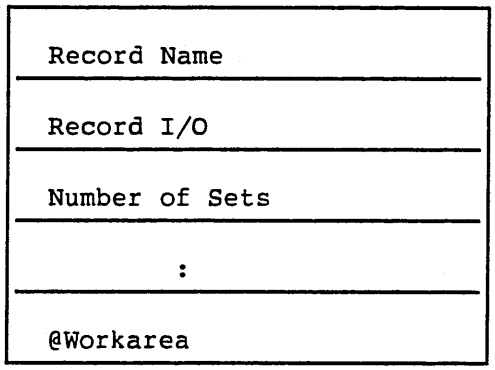


Figure 39. DML verb STORE block structure interface

```
DML-stblock = RECORD
     DML-RecName : INTEGER;
     DML-RecordF : BOOLEAN;
     DML-SetNo   : INTEGER;
     DML-SetName : ARRAY (.1..10.) of INTEGER;
     FILLER      : ARRAY (.1..15.) of INTEGER;
     DML-WAptr   : DB-PtrWorkArea
END
```

Figure 40. Pascal block structure for DML verb STORE interface

7.3 The DDL Structure and Interface

The Data Description Language looks very similar to the DSDL(Appendix A.1). The resultant description after compilation by a compiler, it was thought, could be either code or tables, as for the DSDL. It was envisaged that if the DSDL was held as tables, then to hold the DDL as anything other than tables would complicate their access. In order that a compiler did not have to be produced a compilation simulation was done, for the example by hand(Appendix A.1), to obtain what could have been the output of such a compilation. This output was envisaged to be in a similar format to that produced by the DSDL semantic analyser. A table creator was then produced which creates the same sort of table structure as that for the DSDL in table generation. As was mentioned in Chapter 6 this DDL table structure was also required as input into both semantic analysis and table generation of the DSDL. The schema table accessor routines were used as the means of accessing the the DDL by these DSDL compilation routines. The schema accessor contains routines which ask for information from the schema tables, such as

a. Given the record type, has it got a schema key defined with duplicates NOT allowed. It would return a boolean for schema key found, the schema key name, and a boolean for duplicates NOT allowed.

b. Given a record type, find all those sets for which this record is an automatic member and their ordering criteria. It would return

a pointer to a linked list which contains the names of the sets and their ordering criteria.

7.4 The DSDL Interface and Structures

The DSDL table on input to the run-time system is extended to include direct pointers to the schema tables and further to include identifier within record offset specifications. The access mechanisms developed to create the tables in table generation and to support the verification performed in version analysis in the DSDL compiler were extended by including retrieval of these offsets to create the DSDL table accessor. The storage schema accessor returns information such as

a. Given the schema key name and the record name find the index that is used for this record for this key. It returns a pointer to this definiton and the possible location of the index.

7.5 The Data Dictionary Structure and Interface

The data dictionary defined here was merely a repository for the data identifiers and their symbols from the identifier trees as output from storage and schema compilation. Access to these facilities could be provided to the DSDL as mentioned in Chapter 6. The more common desirablity of the Data Dictionary is that it should contain all meta-data and relationship representation as in the DDL and DSDL. These tables would therefore be part of the data dictionary. Maskell{59} describes the LEXICON Data Dictionary system as providing just that sort of interface. It would be possible to build up this system to include the validation of the DSDL and DDL. Reorganisational checking could be made part of the transitionary part of the system, or as an add-on version analyser. However the structure of the database system may change with connections to the data dictionary being via the application program. Such a system

configuration would probably be undesirable, since then the data dictionary would not be the integral part of the run-time system as was desired.

7.6 The Operating System Structure and Interface.

All input and output to the files that make up the storage areas is through the GET and PUT functions. These functions form the interface between the operating system and the DBMS. The operating system should not simply be that provided by any computer operating system, but be one devised as performing the DMCL style procedures as mentioned in Chapter 4. The operating system performs garbage collection and page optimisation through buffer management, retaining the most frequently used pages in the buffers. It was recently stated at a DB2 (an IBM SQL product) user group meeting that the future of Input/Output reduction would be by providing extended storage memory for these buffers{64}.

7.7 The DBMS structure

The DBMS as shown in Figure 38 is central to the runtime system. A call to the DBMS is made from the application program via the DML verbs using the DML block and passing with it a pointer to the user work area. The data required to satisfy the DML verb are copied from the user work area and placed in the system buffers.

The DML verb then acts as the construct around which the actions of the DBMS are formed, obtaining information as required from the DDL and DSDL tables. The data dictionary is accessed only when the string that is an identifier's name is required.

The actual storage structures and mechanisms provided by the DBMS will be enlarged upon in Chapter 8, but here we will develop further the action taken by the DBMS around the DML verbs.

For example let us investigate the actions that are required to STORE or FETCH a record.

For a STORE, the schema record is created by the application in the user work area. The data is then transferred with the DML call to the system buffers. Information is then obtained from the schema tables and storage schema tables so that the constituent storage records can be defined. Once the storage records have been found, a list is formed from the schema tables for which this schema record is an automatic tenant. Then for each set: a check is made that there is an appropriate occurrence exists; that no duplicate if DUPLICATES NOT ALLOWED is specified; and an owner pointer is created.

Then if all the sets can be satisfied we start to construct the storage record and its prefix information. The creation of the blank prefix for a storage record is described in Chapter 8, this includes creating blank pointers for links and sets.

The data of the storage record must then be concatenated with the prefix in the system work area in order to evaluate the size of each storage record. Once the storage record is constructed the system can initiate finding its placement, and the page upon which to place the record. If there is not enough room on the page specified by the placement criteria then an alternate page is found according to the overflow techniques specified for each type of placement. Before the records can be LINKed together all indirection must be fulfilled, thus locking the required index postions. The link pointers are then completed for each prefix. The structure for the set pointers created during prefix compilation can be now filled in with the pointers to the destination records for set memberships. These records for which insertion is via the CURRENT for their types and sets will already be in the system work area and their pointers may also be updated. This allows for sets whose ordering must be found prior to completing set pointers.

Once all sets and indexes have been fulfilled all storage records and their prefixes which have changed must be placed or replaced onto the pages of the active system pages. All CALC set pointers and page footer values must also be updated. At commit time all changed pages

are PUT via the operating system to the files. Details of the pages' before images are kept in the log so that rollback may be performed.

For FIND or STORE parts of the underlying routines must obviously be similar. However the main difference is the means by which a system, given the placement criteria defined and the system structures supported, may find a storage record. In a parallel processing environment, a multiply mapped record could have a processor per storage record or even per legal placement criteria. The placement retrieval mechanisms will be described in Chapter 8.

Once any one storage record has been retrieved, the optimiser having hopefully used the optimum route, it becomes a simple job to recreate the schema record as in the case of a FETCH or for a FIND to stop there having found one or more of the storage records.

8.1 Introduction

Chapter 7 described the structure of the run-time system by means of its actions which supports DML requests and the interactions with its interfaces. The interfaces developed were those between the DBMS and the DML, DDL, DSDL, Data Dictionary and Operating System. Chapter 8 further develops the structures of the run-time system by means of the storage structures and mechanisms which fulfil the standard picture of a DBMS. The structures and mechanisms required to support the reorganisation facilities, as referred to in chapter 4, are described together with their triggers. The chapter concludes with discussion on when and how far to cascade reorganisation.

8.2 Storage Structures

There are two types of storage structure that are required by a database system, namely those structures that are defined for the use of the system and those required to contain and access the data stored by the system.

The former describes such structures as the Currency pointers, the User and System Work areas, the System Buffers, the Active Page list, and the Page Buffers. The latter are an interpretation of the requirements that are necessary to store the data of such a system. These requirements are the different types of indexes, the Data Dictionary, the Page Description, the Prefix and Record descriptions.

The former are shown diagramatically in Figure 41. This gives detail of an example situation showing the current state of some of the pointers, the constructs of which will now be described.

The Currency Pointers are those defined in the DML as being required to be maintained for access to all records that are current for the set types and record types defined as local to the run unit. The set of Currency Pointers is maintained from within the schema tables, and data dictionary. The information held includes: a pointer to the next block, the record type, a pointer back to the schema table that holds the definition of that type,a pointer to the storage schema record mapping which is dependant on the version number and condition specification, a pointer to the Current Record Array (see Figure 41) and information as to the index usage. The difference between record and set currency pointers is simple: the extra pointers to the set definitions as provided by the schema and storage schema tables, and an indication as to whether the record is the owner or a member of the set.

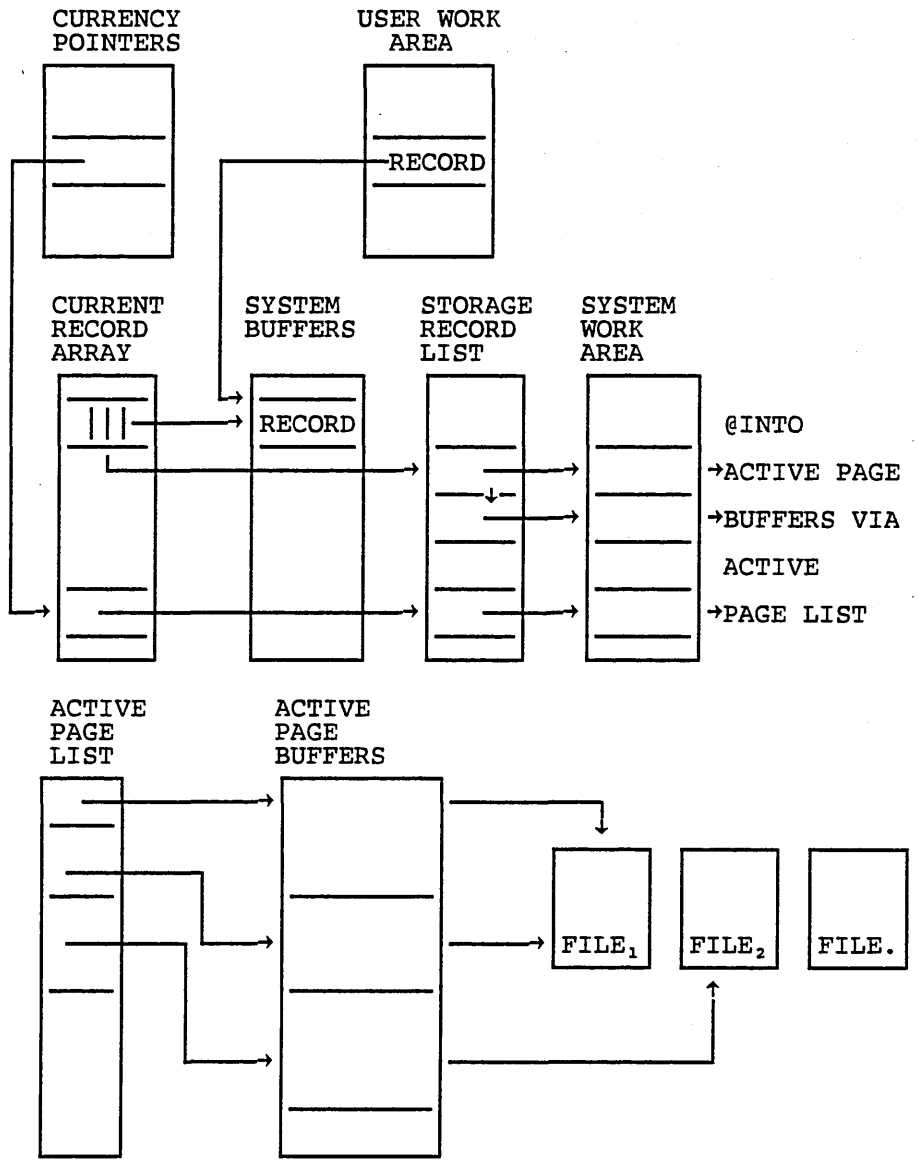


Figure 41. The Storage Structures required during Input and Output.

As stated in Chapter 7 the User Work Area (UWA) is an array which can be mapped by bytes or words, to be used by the application as the depository for values to be required as input to the run-time system, and as the depository of schema records that have been retrieved by the run-time system for the application. The record is copied to and from the UWA to the System Buffers.

The Current Schema Record Array is used by the system to maintain pointers to the schema records in the System Buffers and the storage

records in the System Work Area via the Storage Record List. The Current Record Array is a structured array, where each structure contains information similar to that defined for the currency pointers. This is because of the obvious requirement that not every schema record currently local to the run unit will be the current of that record or set type. The structure also points to the location in the Storage Record List of the first storage record in a mapping.

The System Buffers and the System Work Area are of the same structure as the User Work Area, that is, an array mapped by directly accessible bytes or words.

The Storage Record List is a linked list of linked list structures. A header record is required as the first element and it then points to a list of storage record details. The Storage Record List contains details for each storage record currently known by the run-unit that resides in the System Work Area. Examples of the details required are the start and finish pointers to the prefix and storage record, and pointers to the storage schema tables from where all information for the blank prefix is obtained. When known it will also contain details of which active page in the page list the record is or will be stored. A similar structure is also required for the indexes that are to be stored in pages.

The Active Page List may contain pointers to the pages either holding storage records or index records. It contains information on the file the page comes from, the storage area the file represents, the page number of the page within the area, the page number within the file and points to the Storage Schema details of the area.

The Active Page List and the Active Page Buffer Stack are controlled by the operating system part of the DBMS.

The Active Page Buffer Stack is represented by an array of structures. A Page is described by an array of structures. The Page described in Figure 42 also maps onto this structure, and is mapped by fullword, halfword and byte arrays, so that information may be obtained and stored directly to the word, halfword or byte.

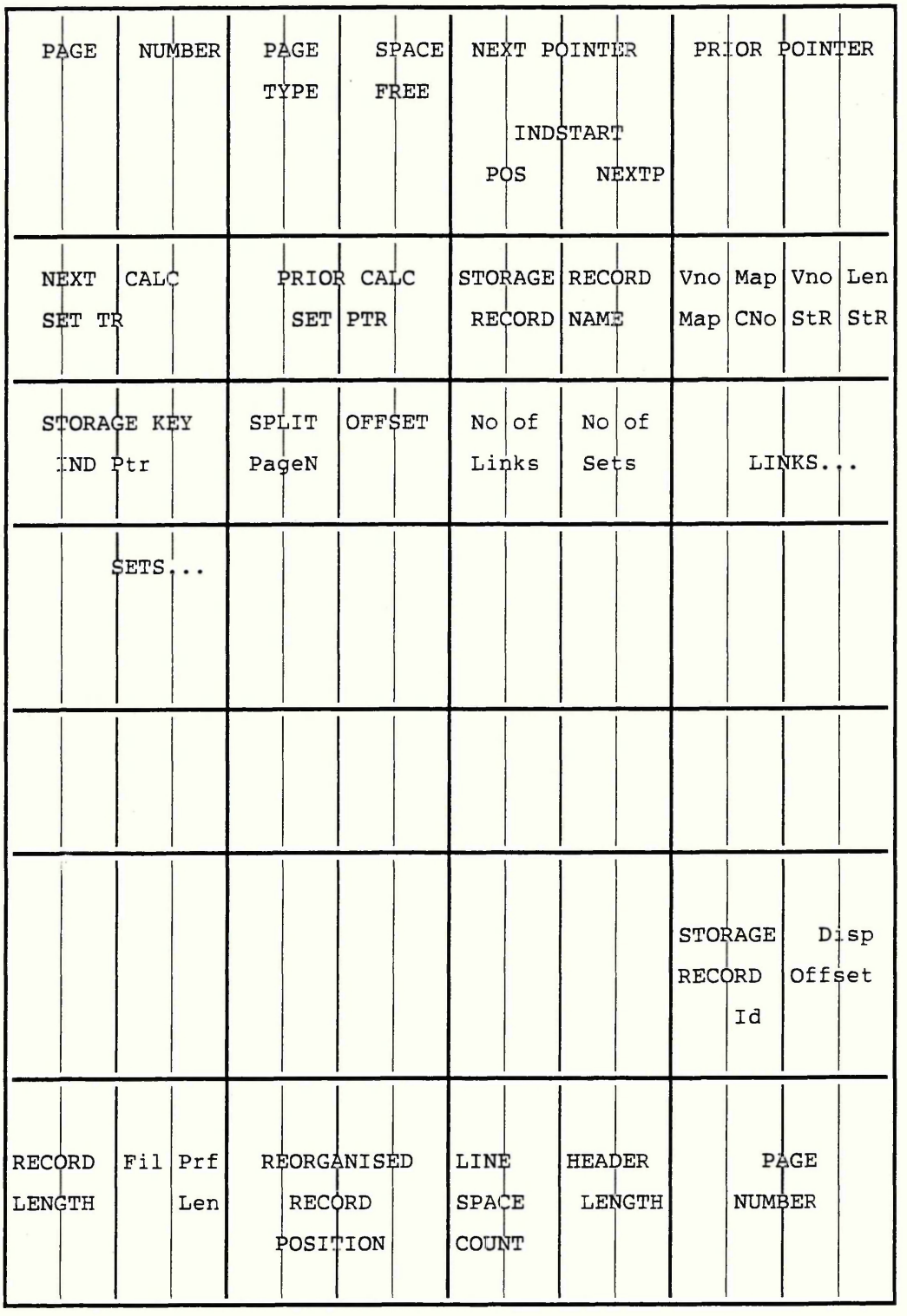


Figure 42. Physical Page Format

The Page Description in Figure 42 gives the structure that is required to store the user data. The physical page format is 256 words, where a word boundary is defined by the heavy lines, a half

word boundary by the medium lines, and the fine lines define the intermediary byte boundaries.

The first 4 words represent the header for each page which consists of the Page Number(1 word), the Page Type(½ word), the amount of the Free Space(½ word), and for those records which are CALCed onto the page, the Next and Prior Calc set pointers. For an index page the Next Pointer is replaced by the Index Start word consisting of a pointer to the start of the Index and a pointer to the next page for this Index.

The last 2 words of the page known as the footer consists of 1 byte for the line space count, 1 byte for the header length and 1 word for the page number.

The basic page format takes as its basis the description of the page used in ICL's IDMS{B9}. However where the CALC set in IDMS was not definable by the user, the CALC set here contains all those records which have been placed on this page via the CALC procedures. It is possible that many different sorts of records could be placed here if this page is in the range specified by the WITHIN clause for the record's placement strategy.

			offset within prefix
Next Calc Set Pointer	1 word		0 (word map)
Prior Calc Set Pointer	1 word		1
Storage Record Name	1 word		2
Version No of Mapping	1 byte		12 (byte map)
Mapping Choice No	1 byte		13
Version No of Storage Record	1 byte		14
Length of Record Segment (in bytes) (used for when the record is split)	1 byte		15
Storage Key Index	1 word		4
Split (page,offset)	1 word		5
No of Links=m	½ word		12 (half word map)
No of Sets =n	½ word		13
Links	1 word	* m	7 + (m-1)
Sets - Set name	1 word		7 + m+2p*(n-1)
Pointer Types	½ word		(8 + m+2p*(n-1))*2
Version No	1 byte		(8 + m+2p*(n-1))*4 +2
No of Pointers=p	1 byte		(8 + m+2p*(n-1))*4 +3
Pointers	1 word	* p *n	

Figure 43. Specification of Prefix Map

A line index is used to define where on the page a record and its prefix are to be found. There is one line index per record, which is

either on this page or has overflowed onto some other page. The first line index is placed adjacent to the Line Space Count, each subsequent line index is placed on decreasing sets of 3 words on the page(see Figure 42). The line index consists of: the Storage Record Identifier($\frac{1}{2}$ word); the displacement offset of the prefix in relation to the start of the page(in words) ($\frac{1}{2}$ word); the length of the entire record(prefix and storage record)($\frac{1}{2}$ word), a byte filler, the prefix length (1 byte) and 1 word to be retained for reorganisation.

As has been stated previously, a stored record, consists of a prefix and a data part. Within the prefix are details pertaining to the definition of the storage record and also to the logical linkages that are maintained for the database.

The prefix is described by the Prefix Map in Figure 43, the minimum size of a prefix is 5 words where the mapping of a schema record to storage record is one to one and it is not defined as being a tenant of an set. Dynamic set pointers are not maintained for this system.

The parts referring to version numbers will be described later in this chapter under the heading Reorganisation Structures.

If a record is defined as being a tenant of a set then the set linkage is maintained by the structure consisting of a setname, pointer types (bit flags), the number of pointers and the actual pointers. The bit flags indicate what sort of pointer each pointer represents, in the same order for all sets. The possible pointers for each set are for the owner: FIRST, LAST and INDEX and for the member: NEXT, PRIOR, INDEX and OWNER. Where the lowest bit is 0 for owner and 1 for member set pointers, and the next 3 or 4 bits respectively, represent 0 for pointer absent and 1 for pointer present. '0101' represents an owner record with pointers for FIRST and INDEX.

If a record cannot fit onto the same page as its prefix then the split page number gives the offset number of pages that page resides from this page and Offset gives the line index number which contains the details of where the record has been placed on that page.

The storage for an index on a page does not take into account what type of index it holds. Logically, as described in Chapter 4, there are 3 types of index: storage key, record key and set indexes.

A storage key index has one occurrence per index description in the storage schema. It is basically used by the system to speed retrieval and help to protect storage records from others which are undergoing movement of some form, in that they are used to support indirect addressing.

A record key index is one which was either defined in the storage schema, or to be used to support some schema record order, or for the use of the system to provide hidden fast access to a record. There is one occurrence per storage schema specification where each element points to just one of the storage records that might be supporting the schema record. It may be implemented as either explicitly holding the key, or else the key may be obtained from access to the storage record(s).

A set index provides the linkage mechanism and supports the order within a set. It may contain keys either implicitly or explicitly. There is one occurrence per occurrence of the set type that has been defined as using the index.

There has been much discussion on the different types of index that can be created. B-tree, B+-tree, AVL tree, linked lists, inverted index are examples of such types.

Comer{21} describes the B-tree and B*-tree, making much use of IBM VSAM facilities. They indicate that a balanced, multiway B-tree is efficient, versatile, simple and easily maintained, while the B+-tree also allows efficient sequential processing of the file. A B+-tree is defined as one where all keys reside in the leaves. Of interest to this thesis is the discussion on optimal pagination of B-trees with variable-length items{27} because of the obvious necessity for an index to be able to cope with values other than the integers used in most other examples. It is suggested that for a B+-tree the roadmap contains a prefix of the key which is of sufficient characters to distinguish the key from its neighbour to the right. This type of tree is termed a prefix B+-tree.

Arnow{3} compares B-trees, compacted B-trees and multiway trees, stating that compact B-trees degrade with insertion,and therefore are best for static data. Multiway trees are storage inefficient at high orders, whereas compact B-trees use significantly less storage than B-trees. Compact B-trees have faster retrieval but slower insertions and that for database implementation the B-tree provides the best overall performance. For indexes which are to be maintained in main memory, only output occasionally, Dewitt{25} compares AVL with B+-trees, to find that the B+-tree is of more use because AVL trees are too highly main memory tuned for keyed access. Dewitt concentrates on keyed access to tuples but it is surmised that the same results would be obtained for storage records.

Bell and Deen{5} compare a form of hash key over primary key tree indexing, and state that where there is high activity of insertions and or deletes an H-tree performs better than a B-tree. For a level greater than 2 B-tree, direct access to a storage record is more costly than through an H-tree.

If this discussion on trees had to conclude on the requirements for a concurrent manipulation of B-trees then the results of Kung and Lehman{48} would require further investigation as the details describing the locking mechanisms within Binary search trees would be very pertinent.

However the conclusions which were drawn from this somewhat small investigation were that there are many ways in which indexes can be created and maintained. It would also appear that some forms of indexes are better than others for larger numbers of records. If this was an investigation into the effects of the structures used in a database rather than the effect of reorganisation, then the following is true. If the records are read only then a different type of index may be used. By specifying a relationship between numbers of records and the storage structure of an index, reorganisation could be used to improve the performance of the index.

The result obtained from this investigation was that since only a minimal number of records would be likely to be stored, therefore the types of indexes would not prove critical. However, a simplistic approach would seem to be the better course of action. Therefore it

would be better to use simple linked lists for set indexes, and provide B-trees for storage key indexes and record key indexes. Where the root node for each storage key index and record key index would be maintained in memory, updates to the database as well as to the root node would be required.

To complete this discussion on the storage structures required by the DBMS we take a look at the different types of pointer as defined by the DSDL. There are 3 types of pointer; link pointers, set pointers, and index pointers, where each may be DIRECT or INDIRECT.

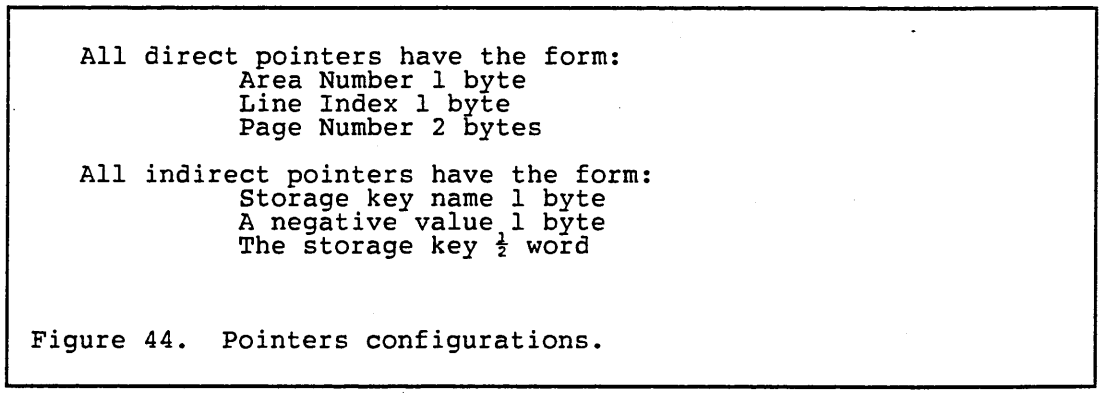


Figure 44. Pointers configurations.

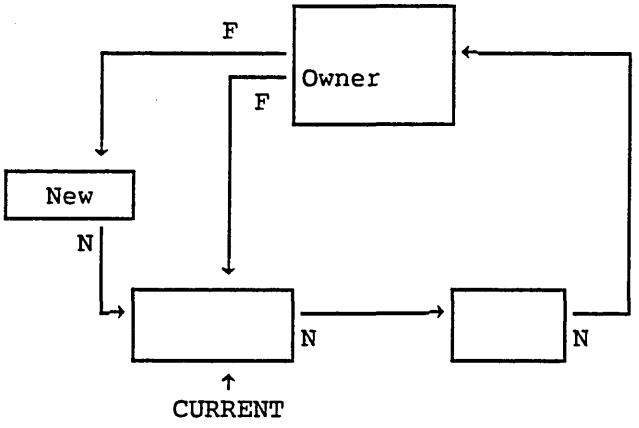
Only the storage key index cannot contain indirect pointers. The only difference between the pointer types is that for set pointers, the first part is negated when the destination is the owner of the set.

8.3 Storage Mechanisms

As for the storage structures, the mechanisms supporting the storage of data in a database can be regarded as of two types, those mechanisms which control the access paths and those directly related to the storage of the data. The former include insertion and maintenance of the order within sets, fragmentation, control of pointers, and the control of the placement strategies. The latter includes such mundane aspects such as efficiency and garbage collection as well as logging and the creation and placement of the prefix and storage record.

Most of the mechanisms will be common among most types of network database systems. In some form or another we will therefore concentrate on those particular to the support of the newer DSDL functions, such as cyclic sets, and then follow this by the support required for reorganisation. The storage schema also provides extra mechanisms through which sets can be supported, for example set indexes.

The simplest linkage within a set is for the owner to have a FIRST and the members to have NEXT pointers as in Figure 45. However if the set was sorted on a descending key, then to obtain a particular record according to its key could involve a search in ascending order, if both FIRST, NEXT and LAST, PRIOR pointers are not supported. The search would then have to retrieve each record, on the way looking at the data item values that make up the record. The problem is further complicated if the record is fragmented. Especially if the destination record does not contain all of the key.



New.Next = Old.First
Old.First = (NEW)

Figure 45. Insertion into a Set Order is PRIOR, supported by FIRST, NEXT pointers

The search algorithm is as follows:

```
Follow the pointer
   : fetch the storage record
   : if the key is not in the storage record
      : then fetch all the storage records
   : find the key
   : if its less than the required key
      : then find the next pointer for this set type
   : repeat.
end
```

It is not possible to have prior knowledge of which destination record version this could be, since there could be many mappings and many versions. If there are not then this could be one of the many optimising points.

'Fetch all the storage records' is another way of completing a mapping, which could arise when a record is fragmented. If fragmentation has been used it must be remembered that not only may we be providing many different ways of recovering the record but that for each storage record, the placement strategy is made more complex. For ad hoc queries rather than those using the network, a record must be found using the placement strategies. For multi-version storage schemas, a decision must be made as to the version number at which to start using the definitions, and then to follow it by other versions until all possible choices are negated.

For example, consider the ad hoc query "What items can be purchased in that shop?" from the discussions on the example in Chapter 5. It is first necessary to locate which shop it is. If we said any shop, then we could look in the storage schema to see that both types of storage record Gen-det and S-addr are placed within M-s, pages 500 to 999. Using this information the system would proceed with a sequential search through the pages. Since both placements were of type CALC there are two possible search methods. One follows CALC set pointers for each page until a record of the particular type is found. The other uses the line index to locate each record in turn on the page. Both use the prefix to discover the identity of the storage record. Alternatively if the shops Chainname had been mentioned then using the CALC routine Chain a storage record for that Chain could be located from the CALC set. If no record of the type exists, the first query will have searched all occupied pages in the range; the latter only the pages of the CALC set.

The order in which pointers are updated is very important, especially when cyclic sets are involved, otherwise members of sets or even whole sets could be lost, or retained when cascade deletion is required.

In considering the mechanisms to support the storage of data, we first consider the activity of garbage collection, if only to discard

it, as being a part of the DMCL, which most systems provide, if not all that efficiently.

The logging adopted here is simply used to retain a before image for all pages accessed by the current logical unit of work. A rollback merely ensures that the database has not been changed for these pages.

As stated previously in this chapter, mechanisms are required to control the creation and placement of the prefix and storage record. From the data in the storage schema, in particular, the data subentry of the current storage record, if it exists, or from the data description of the schema record, a map is created which will be used to obtain from or place values in the storage record. The complicated mappings of identifiers to storage types are ignored by the system.

At this stage the blank prefix, as described earlier in this chapter, must be created, this is done by obtaining data from the current LINK clauses and finding all necessary set information from the storage schema. The LINK clause count returns the number of links required to be maintained and the order that they must be in, represented in the storage schema tables by their associations. The sets in the storage schema table are searched for those which the current schema record is a tenant. A linked list of all of these sets is then formed for each whether this record is the owner, member automatic, member ordinary, or both and link them into either an owner list, a member list, a member ordinary list or a tenant list.

The main list contains back pointers to the storage schema tables, and indicates the version number used for the set. This will be discussed later. Before the pointers can be filled and mapped into the prefix, any indexes into which the(se) record(s) are to be placed must be positionally locked. From the storage key indexes the indirect pointers can be obtained. It is now possible to fulfil all pointers in the prefixes for a schema record's storage records. When a record is placed in a page the CALC set is updated for those records to be stored CALC. A line index is also created for each storage record. If the record is split because it did not fit onto the page, then the overflow procedures take over, updating the prefix Split pointer to point to the chosen overflow page. The chosen page

is implementer defined within the bounds of the DENSITY and WITHIN clauses.

So completes the discussion on the structures and mechanisms provided by the run-time system to control the storage of data in a network database system that provides a user definable storage schema. The following sections will try and complete the picture by describing those structures and mechanisms that are required extra to the basic system, in order to support dynamic incremental foreground reorganisation.

8.4 Structures to support reorganisation

Discussions on the structures reported previously in this chapter have omitted the facilities required to support reorganisation, and these will now be described. There are however, no grand structures necessary to support reorganisation. The storage required is very small. It is the mechanisms that have greatly increased, and these will be described in the last part of this chapter.

The structures required for reorganisation are merely extensions of the original structures. For example, we consider the structures required for input and output, and the page and prefix structures.

The former requires an increased mechanism in order to control the additional 'update' of records or set pointers to their current versions. It is also possible that an extension of the storage schema table to include object version combination counts could alleviate the problem of the removal of an object's definition from the storage schema. The only object not so relieved would be an index. To remove an index requires extra functionality not provided in the DSDL, that of dynamically dropping an index. The currency pointers, it was stated in the first part of this chapter contain information which points to the storage schema tables. Any pointer to the storage schema must take into account the version number and therefore version definition of the object. Similarly for all other storage schema table information where versions are involved.

The physical page format(see Figure 42) was described in the first part of this chapter. It described, a header, a footer, a line index, the prefix, and the storage record. The header for a record type page makes no reference to any versions and did not require any modification to support reorganisation. For an index however the version number of the structure is maintained as the 13th byte on the page. The footer also required no modification.

The line index is that part of a storage record's placement that resides in situ even if the storage record has been moved during reorganisation. The Reorganised Record Position details the location of the real storage record. In this way records can be reorganised without affecting others.

When a record is reorganised the Displacement Offset, Record Length and Prefix Length are zeroed. The storage record identifier remains the same, but cannot be relied upon to equal the identifier of the new storage record. The effect on new storage records and mappings is described in the next part of this chapter.

The prefix is the means by which linkages are made and also provides the complete description of the attached storage record, it must provide version information. Because the record could be part of a mapping for which there are many versions, the Mapping Clause number must be qualified by the Version Number of the Mapping. Similarly there must be a Version Number for the storage record type. In this way the combination may be located from within the storage schema to uniquely define the storage record. Further it is not necessary to update set pointers when a record is updated, so a version number is required for each set that the record is a tenant of, described later.

8.5 Mechanisms to support reorganisation

As stated in 8.4 the structures required to support dynamic reorganisation are few but the mechanisms and decisions are many. Not only are the actual mechanisms required, but the decisions of when

and how far to reorganise are very complex. It could be said, that the first decision as to when to reorganise is the simplest. Only reorganise when writing to the database. However this would imply that only when records are written or updated would a new version be applied. If, as in the case of the example, an area is found to be full, or the read access time is found to be too great and another access path needs to be created, then we can see that without putting through needless updates a reorganisation might not take place.

There are two ways in which this can be resolved. The first to allow reorganisation on read. The second is that some background utility performs this sort of reorganisation. Only the first method which was within the scope of this project was investigated.

Let us now investigate the triggers of reorganisation and the depths to which they affect the rest of the data. Using the storage objects; area, mapping, storage record, set and index, we will see what triggers the changes for new versions as described in Chapter 4.4.

The triggers could be seen as the reorganisations themselves and are categorized by, an update to a record, a store of a record, manual insertion or disconnection to a set, a record read, or a set traversed. The 'set traversed' is the most controversial as it may cause any number of reorganisations, and for what appeared to be a simple read operation could become very costly.

As stated previously, a new area is a static reorganisation and is created at the startup of the run-time system when the new storage schema is introduced.

Let us say that there is a new mapping. One condition produces the same storage records as before. The other provides one original storage record plus two new ones. If a record is updated, that was stored with a previous version of the mapping, then there are two choices. Either the record when restored is changed to the new mapping or the record's storage records stay the same unless individually they have a new version. The latter may then trigger the new mapping (see storage record reorganisation). If the former is chosen then it would depend on which condition was used to map the record. The former condition would not trigger storage record

reorganisation but the latter would. If a new mapping changes the destination record of set and or index pointers, then either the reorganised part of the record can be left to point to the new destination record or else those pointers in the set and indexes which are affected could be changed. The latter would of course take a lot of locating. If the new MAPPING caused a new SET version then for the action required (see set reorganisation).

8.5.1 Storage Record Reorganisation

If there is a new version of a storage record then its invocation could be triggered by a new mapping or simply via an access by an update or read. All new records are stored with the most recent version of a storage record. A new set version could cause a new storage record version to be adopted as will be shown in set reorganisation. In formulating a record to get its data item values, a storage record could be written back using the new version, this must be controlled to prevent untoward deadlock. As stated in Chapter 4.4.2 a storage record reorganisation could be separated into three sorts of change; changed linkages, placement and data description. To change the linkage without adopting any other changes to the storage record would require version numbers to be stored with the links. It could also invalidate the data in a record. For example, some identifiers are removed from the current storage record in a new version, a new mapping is created which involves changing to this new storage record and another storage record which includes these identifiers. To change to the new storage record definition without invoking in this case the new mapping would lose data. However a change to the placement of the storage record or a change to provide extra LINKs could be done so long as the DATA subentry remained the same. However changing to part of a new mapping is not valid.

8.5.2 Set Reorganisation

A new version of a set can be adopted in bits only if a new linkage mechanism is not chosen. That is, changing from FIRST, NEXT to LAST, PRIOR support would cause all set pointers in the set to be changed. Changing a mapping and therefore the destination and description of the storage record need not imply the rest of the set need also be reorganised. But if any of the affected records had a new mapping which affected the set pointers, then these would also be triggered. However for those in which a new version of a constituent storage record only had a new version, the reorganisation is optional. The storage records however may require their prefixes to be updated with new pointer values, at which time since a write is necessary, then to reduce Input/Output an optimiser could update the storage record or mapping version. The cascades of reorganisation which can follow especially with cyclic sets is optional.

A change to the pointers could merely involve the creation, or addition to, of an index. Although only non-singular set indexes may be reorganised dynamically, the storage key index providing indirect pointers could hide reorganisations that are taking place. Any pointers which are direct to a changed storage record or mapped record would have triggered their originators to be changed, had they been indirect. A temporary form of indirection can be produced using the line index reorganisation flag, the Reorganised Record Position(RRP). Any pointer that points to the RRP, when used, is changed to point to the new address. The problem remains, however, when to remove this line index. One way could be to have a count for those pointers which point directly to a line index, and subtract those pointers which are changed subsequently. If the RRP option is not allowed then all the sets for which the record is currently a tenant, including indexes, will be affected.

8.5.3 Reorganisation Triggers

The next problem to solve, having worked out in general what types of reorganisation and forms of access act as triggers, to decide at what time to do the reorganisations. Investigated were the options; to reorganise during a commit, or to reorganise during the creation of the storage records. Also to what extent are rollback and deadlock affected.

If a record is not reorganised until commit time then this would have involved not performing any placement checks or set completions until that point in time. If a set reorganisation were to cascade to other sets and or records then not only would the records already affected be updated, but that new records could have to be read in order to complete reorganisations.

If reorganisation is performed while the new storage records are created or updated then reorganisations are separated into small parts during the life of the logical unit of work. However at the end the logical unit of work must be rolled back, and then all the reorganisation must also be undone. It could also be that when a record is truly read thus obtaining its values, that at that time it too could be reorganised. Then would rollback affect it, since the only change to it would be a reorganisational change.

Although this is a single user system the decision for a multi-user system as to what would be locked is most vital. For example are those reorganisational changes, on read, hidden from other users, that is locked, or could they be visible. This depends in part on what the effect of a rollback is for these storage records.

9.1 Introduction

The details of the example and runtime system having been defined in Chapters 5, 7 and 8, we now look at the effect that the method and support of reorganisation had on the example, and the effect of reorganisation on the runtime system specification.

9.2 On the Example

In Chapter 5 we saw that performance tuning was required in two areas. The first was that the storage area M-s had become overcrowded, the second that the access time required to find where a particular Chemical could be brought had become too great.

The requirements involved generating definitions for a new area called Items, and a new mapping for Item consisting of the two storage records S-item and I-item. Consequently, the new set versions for C-item, L-item, F-item and Ch-item and a new placement for the storage record Chemical are required.

The reorganisations once implemented, let us investigate the effect they are having on the queries specified in Chapter 5. These queries were: "What Chemicals can I use to process this Film?"; "What can I purchase in that Shop?"; "What lenses can I buy for this Camera?".

Let us first investigate the effect on the query "What Chemicals can I use to process this film?". As stated in Chapter 5, to find the Process and Chemicals the record detailing the required Film needs to be accessed. This first access is not affected once the Film is found. The first Process to Chemical combination is found by following the FIRST pointer. The record found happens to be a Photocolor II developer (see Figure 46). If the owner of the Used-in

set is found, then the actual Chemical can be looked at. This Chemical record is of the type that could be reorganised. There were three options at this stage: it is already reorganised, it was reorganised but the pointer to it was not updated, or it has not been reorganised yet. If it was already reorganised then the pointer just followed is updated to point to the reorganised record, or in the first case, nothing is required to happen. The reorganised record is found by just one extra page read. The update of the pointer implies that this transaction is no longer read only, but this should be invisible to the application. If the set is supported by more than one link mechanisms, then the system must ensure that all the pointers are updated.

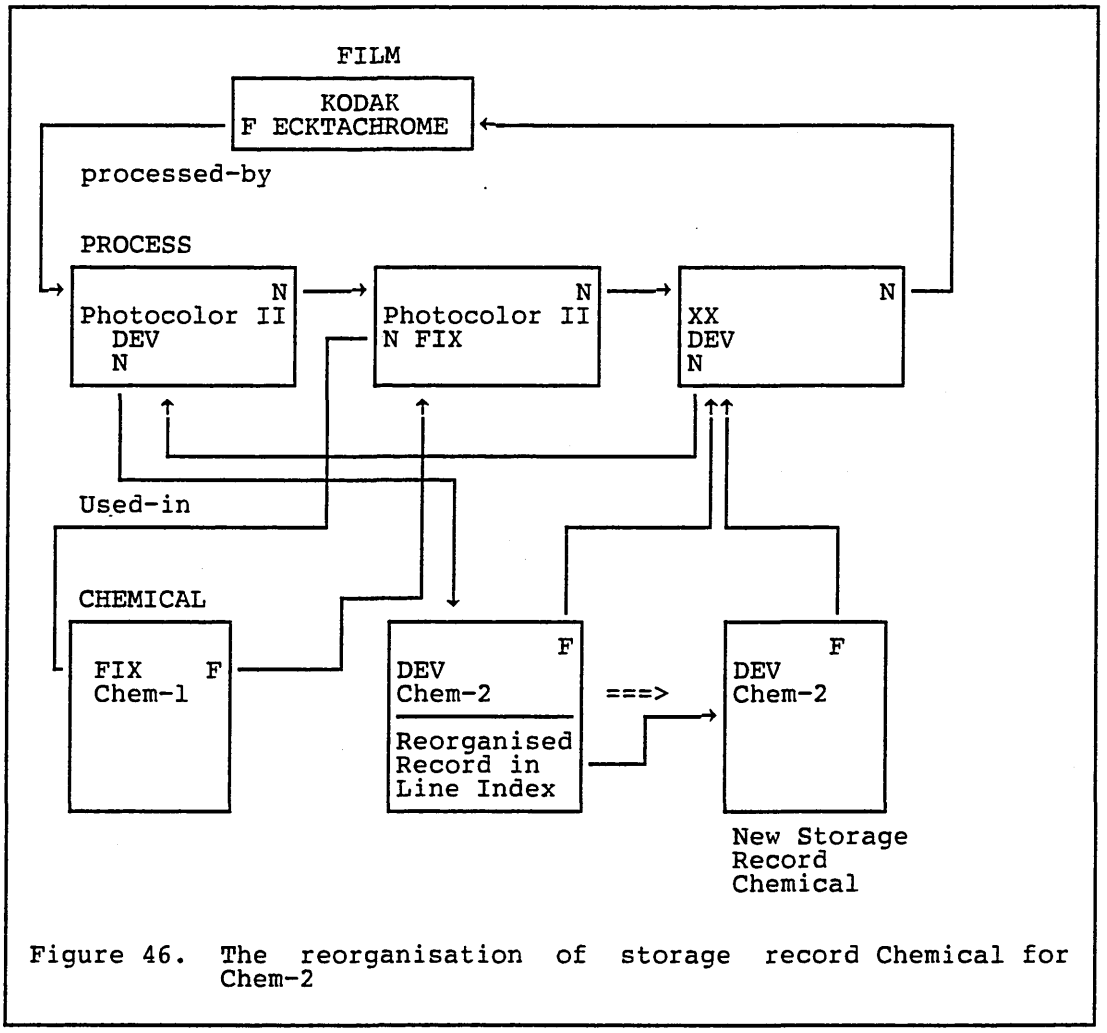


Figure 46. The reorganisation of storage record Chemical for Chem-2

If the record had not been reorganised, then because detail from within the record is required it is possible to trigger the reorganisation immediately. If the Reorganised Record Position is not adopted then the set pointers for Ch-item and Manufactures must

also be searched and updated. All the records of Ch-item are affected but only the index of Manufactures is affected. If the cascade option is not withdrawn then this could affect the occurrence of Ch-item to which this Chemical belongs, since there is a new version of the set. However it is not necessary to update set pointers simply because of movement of a destination record.

The system as implemented uses the Reorganised Record Position pointer in the line index, therefore if the record had not been reorganised then the actions portrayed in Figure 46 are enforced. However although it is possible to change the NEXT pointer in the Process record, Photocolor II developer, the reorganisation does not at this stage affect Ch-item or Manufactures.

For the second query, "What can I purchase in that Shop?" or, as specified in Chapter 5, "What's on the Inventory of that Shop?, we will see that some 'simple' reorganisation can in fact be quite complicated.

In Figure 47 we see that the particular Shop we wish to find was named Jessops of Sheffield, and that the Chainname was Jessops. In order to find this Shop record, the system uses the CALC Chain on Jessops and then searches the CALC set on that page until the Gen-det record for Jessops of Sheffield is found. The system can calculate that it is not necessary to formulate all the record by obtaining all the storage records because the next part of the program only uses the pointers in Gen-det. The next part of the query informs the system to use the set Inventory to obtain the first Item record. Once located the system formulates the record. The reorganiser notes that there is a new Item mapping specified in the storage schema. As the system is reorganising on read as well as write, it reorganises the single storage record Item into the two storage records I-item and S-item, and creates the storage key index Iit-ind to support indirect pointers to I-item(see Figure 48).

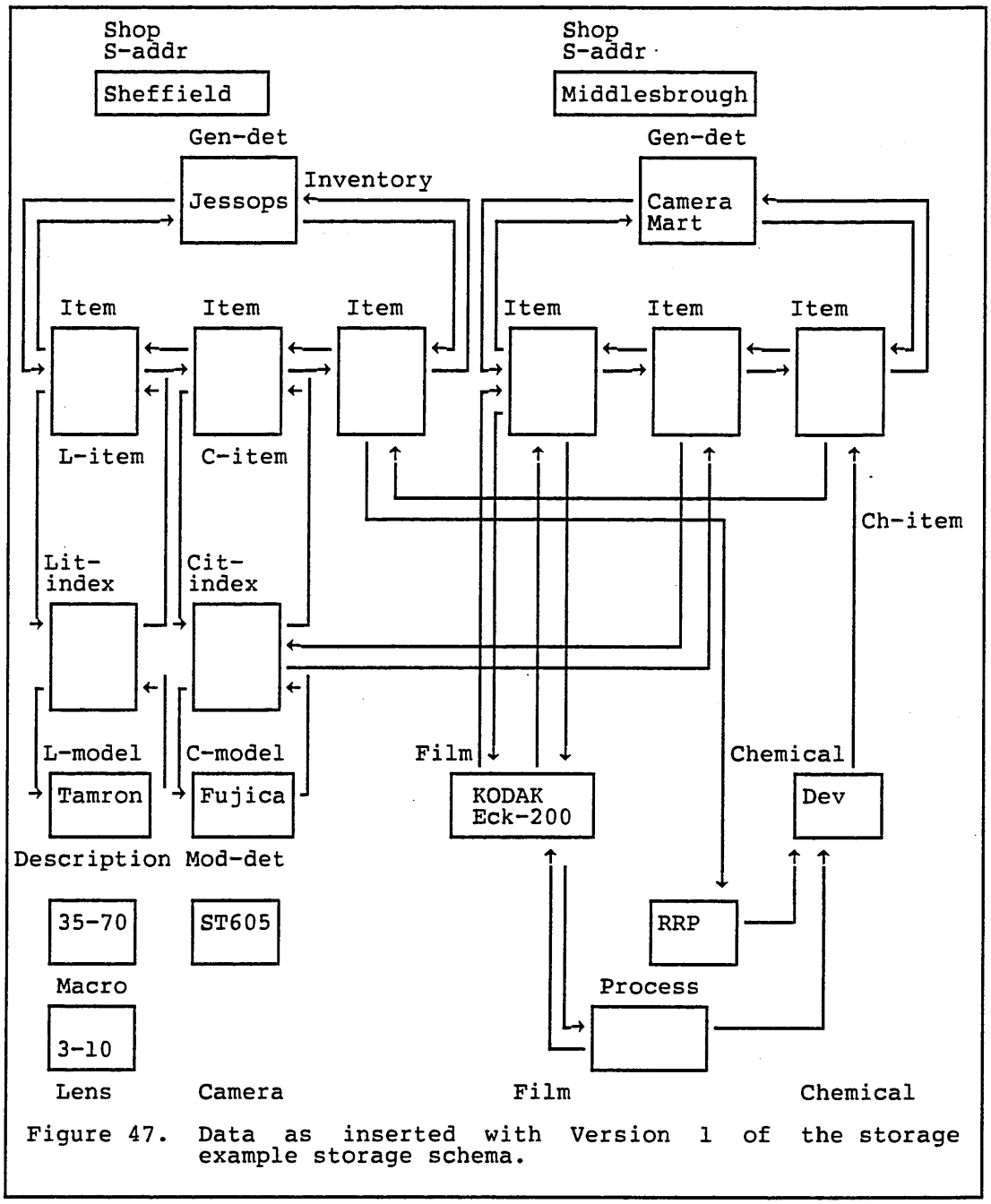


Figure 47. Data as inserted with Version 1 of the storage example storage schema.

The current I-item address is inserted into the index. As there is a full set of NEXT and PRIOR pointers, the system can easily update the set pointers to point to the new S-item record rather than the Item Reorganised Record Position (RRP) pointer. The rest of the pointers are separated between the storage records; S-item being linked to Inventory, C-item, F-item and Ch-item, and I-item being linked to L-item. Note that completing the set Inventory pointers did not trigger the rest of the Item mapping reorganisations.

The next part of the query involves finding the owner of the current Item for the set L-item. This is done by following the index pointer to Lit-index and getting the owner pointer, to find the L-model storage record for the record Lens. The record is obtained from the amalgamation of the storage records L-model, Description and Macro, to see that the record details a Tamron 35-70 Macro Lens. Although there is new version for the set index L-item, it need not be triggered. Thus we arrive at the diagram of the records accessed and reorganised as seen in Figure 48.

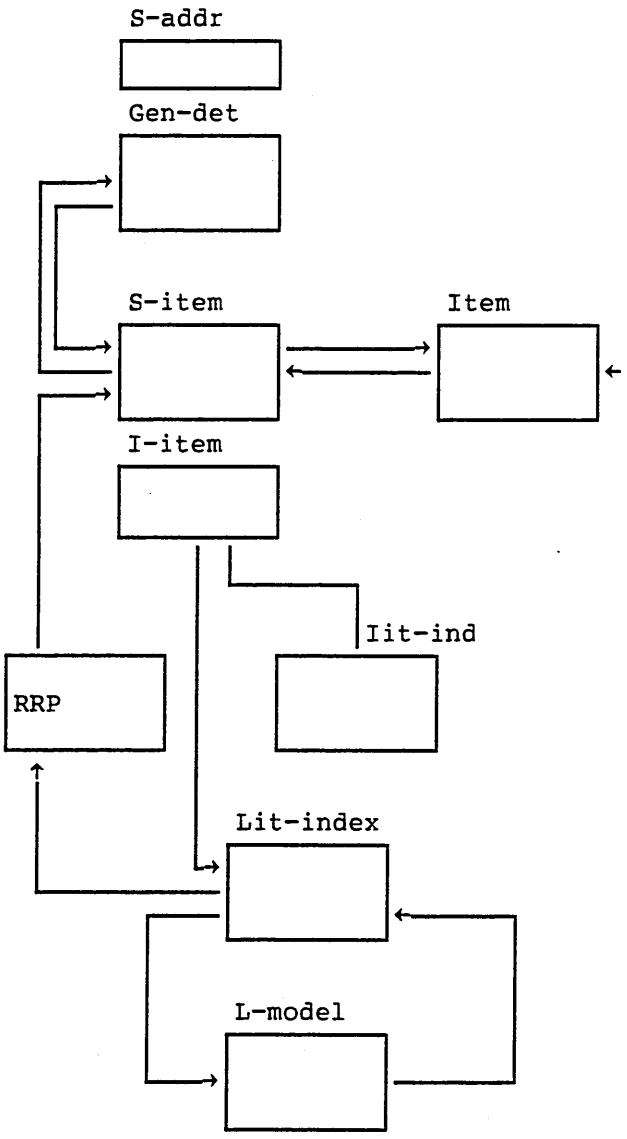


Figure 48. The status of the pointers after the first access to the L-model storage record.

For each NEXT in the set Inventory the above reorganisation is triggered, thus each Item on the Inventory is split into two storage records and a Reorganised Record Position pointer is created. In this way the second objective of clearing space in the storage area M-s is achieved.

The third query "What lenses can I buy for this Camera?" accesses the records Camera and Lens using the sets Uses and Used-on. The use of these sets does not trigger any reorganisation, even though the storage record L-model participates in the set L-item which has a new version for it, and similarly for C-model and the set C-item.

None of the queries mentioned so far involved ad hoc accesses to storage records which have more than one possible placement specification or version. These records are the ones for which it is more difficult to find out if they exist. This sort of access obviously becomes more expensive to run as time progresses when reorganisation is involved. This is because more versions of the placement specifications simply increase the complexity of the FIND.

9.3 On the Storage Structures and Mechanisms at Runtime

Chapter 7 introduced all the parts of the run-time system, and in particular the DBMS and the other parts which had to be written in order to provide the vehicle on which to test the feasibility of dynamic incremental reorganisation for a network database system. The run-time system was then expanded in Chapter 8, detailing those structures and mechanisms which were required in order to support a standard network system. Those facilities were specified by the DDL/NDL 1980 and DSDL 1981 specifications, but did not at that stage support the specified reorganisations.

Chapter 8 then went on to expand those structures to include those parts necessary to support dynamic incremental reorganisation. Some of the mechanisms which were required to support these structures together with the general rules as provided in the 1981 DSDL were then described. We will now look at the effect these structures had

on storage requirements, and the effect that the reorganisation mechanisms had in terms of the code requirements.

As can be surmised from Chapter 8 the extra storage required to support reorganisation is negligible. The page break down is as follows.

The prefix consists of two parts, a fixed part which is the same for all storage records, and a part which is dependant on the number of sets of which the storage record is specified as being a member and / or owner.

The change required to the fixed part consisted of a half word which represents an increase of 7%. For each set the version number requires 1 byte which gives an increase of approximately 8%. The header and footer do not change. This leaves the line index, which contains the Reorganised Record Position pointer which is set only when a storage record together with its prefix is reorganised to another position. When such a reorganisation occurs, the complete line index is left on that page as well as their being a line index for the record's new position. If the choice is made to include having a counter in the line index to keep a count of the number of pointers whose destination is that line index, then the filler(Fil) already present in the line index can be used. Thus the line index requires 50% more space per storage record to support dynamic incremental reorganisation, in the ratio of 3 to 2 words.

When considering the code requirements for reorganisation, the first suit of programs that was produced was the DSDL table compiler. The compiler, as described in Chapter 6, consisted of five passes. The PASCAL code requirements for each pass were: 2200 lines for Lexical Analysis; 3400 lines for Syntax Analysis; 4700 lines for Semantic Analysis; 2700 lines for Table Generation; and 4,500 lines for Version Analysis. The extra pass, Version Analysis, was required to compare the old and new versions of the storage schema, as described in Chapter 6. This code, which was required to verify dynamic incremental reorganisation specifications, was totally new, making about 25% of the total code required for the compiler.

It was thought that the initial specifications of the run-time system to support a none reorganisable storage schema, would consist of upwards of 20,000 lines of Pascal code. The increase that was required to support the reorganisation triggered by the DML verb STORE consisted of about 3,000 lines of code, about 40% of which could be used by the other DML verbs. Therefore we calculate that the increase in total would have been about 40% to 50% on top of that required to support the standard storage schema, that is about 30,000 lines in total.

The aim of this project was to show that the reorganisation facilities as provided by the 1981 JOD{B6} were as feasible as they were desirable. That is to provide the DSDL ability to reorganise on a dynamic incremental basis the objects in a network database system. To this end a compiler for the DSDL, and a run-time system were designed.

The implementation of the compiler/processor for the DSDL as presented in Chapter 6 was completed first. The production of tables, required in total five passes, Lexical Analysis, Syntax Analysis, Semantic Analysis, Table Generation and Version Analysis. The latter being the unique requirement for valid reorganisation analysis. Problems incurred during the production of the compiler included; the errors in the syntax and semantics in the JOD, the none LL(1) grammar and the changing nature of the JOD.

Had the facilities of a lexical and syntax analyser been available, such as LEX and YACC, then it estimated that the compiler production time would have been reduced by a third, allowing enough time for the investigation of the production of the code related DSDL mentioned in Chapter 1. Also the discovery of the syntax faults in the DSDL would have been quicker. The novelty of the compiler would not have been changed with the new compiler techniques and any similar production of a compiler for a relational DSDL would require a similar type of Version Analysis.

Previously in the discussion a relational optimiser has been eluded to. Obviously such an optimiser must make appropriate decisions. It is possible to have not just a logical optimiser such as those provided by current management systems but also one which pertains to both storage and access mechanisms. The latter is required because the method of access provided by current systems is not always suitable to the type of data involved. During the discussion on the provision of indexes in Chapter 8 we saw that the amount and type of data accessed altered the characteristics of the indexes. To this

end, interaction between the optimiser and the database administrator is required, alone it can not do the job proficiently.

The production of the run-time system was designed around the requirements of the DML verbs, based on their relationship with the data storage and access methods taking into account the environmental restrictions imposed by the operating system. However, it was found that a simulation was not adequate because it was the details that provided the real reorganisation and the problems associated with it.

The basic structures and mechanisms required to support a DSDL were described in Chapters 7 and 8. The investigation has shown that it is not only necessary to have logical to physical description changes, but the basic structures such as indexes must be allowed to change if circumstances dictate. A small network or relational system can be adequately supported by the simplest of index structures. Growth of a system must indicate that complex structures may be required, such as those described in Chapter 8.

One of the main problems, was in deciding what does, and when is, reorganisation triggered. Also to what extent should the actions of reorganisation trigger other reorganisations. All these facilities were described in Chapters 8 and 9.

The main problem with undertaking an implementation of a run-time system for a network database system was its sheer size. Therefore, in order to simplify the production of the run-time system, although all DML verbs were investigated, the design concentrated mainly on the requirements of the verbs STORE and FIND or FETCH. Using these verbs, the effect that the access of records and the storage of records has, on reorganisation was investigated. The example in Chapter 5 and the results presented in Chapter 9 have highlighted some of the problems that arose. The first verb to be encoded was STORE, however in order to store a record it is frequently necessary to 'find' other records. For example, CLUSTERED storage for a storage record when near owner, requires the owner to be found, and that record(s) need not be the CURRENT of the set. Thus when investigating the effect of a STORE the system also generated a lot of the functionality for FIND and FETCH.

What materialised from the investigations into the triggers of reorganisations was that although possible, reorganisation could be costly in terms of storage, and very costly in terms of disk input/output. That is the pages required to be local to a run-unit could be increased by as much as 70%. Also the requirements of many systems would force, as for the example in Chapter 5, the run-time system to create FIND or FETCH forced reorganisations. Since a read only query may become more popular thus forcing extra or better access paths to be created.

The experimental evidence seems to show that although possible in a single user environment, the effects of reorganisation in a multi-user environment, could require either advanced deadlock protection, or split level reorganisation. The later would then be incremental back ground reorganisation rather than foreground reorganisation. The triggers being the units of a record accessed the actions taking place behind the scenes. However it is still difficult to see how affected records would be effected by multi-user access. For example a record is reorganised that triggers another, which is required by another run-unit, the latter updates it, the former rolls back. If the first record is not reorganised, then the second record which it undoubtedly points to, or is pointed to by that record could have an invalid prefix.

Although for the NDL this trigger effect does cascade to too many records, if many record and set types are reorganised at any one time then as for record deletion there will still be some effect on connecting sets and storage records. The same can be said for a relational system for which referential integrity constraints are defined. The best way to avoid a cascade is by using indirect pointers via storage key indexes.

The higher the number of record types and so forth participating in a new version, then the more likely they are to be affected by cascade reorganisation. Thus increasing the resource usage, wait time and the possibility of deadlock (for a multi-user system) for an application. It is believed that for a multi-user network system that the occurrence of deadlock, during high dynamic foreground reorganisation activity, is increased dramatically, possibly out of control. In effect cascaded reorganisations should be done in the background, but

ABSTRACT

A Machine Independent Implementation of a Data Storage Description Language

Anne L. Zorner

This thesis presents the methods, results and conclusions of a design and implementation of a Data Storage Description Language (DSDL). The DSDL chosen was the CODASYL Network DSDL. The design supports storage independent manipulation, for access and reorganisation of partitioned schema records, sets and indexes. The production of a Table Generator to compile the DSDL provided the basic structure and mechanisms of a run-time system for the support of dynamic incremental reorganisation. The project developed storage constructs and techniques for a machine independent Data Storage Description Language and evaluated these ideas through an implementation.

The particular objectives of the project included the evaluation of the efficiency of the techniques regarding the criteria of the storage space of tables and records, time for processing, and ease of reorganisation. By developing a run-time system to control dynamic reorganisation of a database given a new version of the storage schema for the same database.

that a reorganisation process should still be triggered by the access of the record. For a STORE however the record must be stored using the most recent version of the storage records, mapping and index, therefore it is difficult to see how any such reorganisation might not effect other storage objects.

The original intention of dynamic reorganisation is still valid, with the demise of network database systems there has come the flourishing but uncontrollable relational database system. Systems such as SQL/DS, IBM's VM/CMS relational database management system, would still benefit from the inclusion of some form of DSDL. It can be seen, from information{B18} that the basic storage structure available in the SQL database, is not that dissimilar to that described by this thesis. The fact that indexes can benefit from reorganisations, into such structures as defined in chapter 8, is without doubt. Although, when referential integrity is introduced into the SQL then there would arise the same cascade problems as for a network database. In other words permanent DIRECT links should be avoided when considering foreground incremental reorganisation. Current relational systems already have advanced deadlock detection facilities, provided that reorganisation across referential integrity constraints into other tables is detectable then there can be only fewer problems than those found for a network system.

Further work that could be envisaged would include the investigation of the implementation of such a DSDL for a relational system {23} as that being defined by the DBAWG. An implementation of such a system could include the manipulation of tables using predicate calculus or an SQL type language. Any such implementation should not be deterred from making use of current technology, especially in the area of transputers{B19} and parallel computing technology. Where it is felt that the DSDL fragmentation and multiple placement techniques would be greatly improved. In this way providing some form of a CAFS type interface, allowing the hardware to improve retrieval times. Thus removing some of the difficulties from ad hoc retrieval of multiply placed multiply versioned records.

A.1 DDL Example Schema

(error recovery and access control omitted)

```
SCHEMA NAME IS Photo-Schema
   RECORD Manufacturer
        KEY Manu-code is M-name
            DUPLICATES ARE NOT ALLOWED
        01  M-name TYPE IS CHARACTER 20
        01  Address
          02 Number          FIXED     6
          02 Road            CHAR      20
          02 Town            CHAR      20
          02 County          CHAR      20
          02 Postcoad-Zip    CHAR      10
          02 Country         CHAR      20
        01   VAT-no           CHAR      10

   RECORD Shop
        KEY Shop-code IS S-name
            DUPLICATES FIRST
        01   S-name           CHAR      20
        01   Address
          02 Number          FIXED     6
          02 Road            CHAR      20
          02 Town            CHAR      20
          02 County          CHAR      20
          02 Postcode-Zip    CHAR      10
          02 Country         CHAR      20
        01   VAT-No           CHAR      10
        01   Chainname        CHAR      20


   RECORD Supplier
        01   S-name           CHAR      20
        01   M-name           CHAR      20


   RECORD Camera
        KEY C-name IS Brand-name,model
            DUPLICATES NOT ALLOWED
        01   Brand-name       CHAR      20
        01   Model            CHAR      10
        01   Mode             CHAR      5 OCCURS 3
        01   ASA-range
          02  Top              FIXED      5
          02  Bottom           FIXED      3
        01   Speed-range
          02  Top              FIXED      5
          02  Bottom           FIXED      3
        01   Flash-sync-spd FIXED      5 OCCURS 2
        01   Rec-retail-prc FLOAT      6,2


  RECORD LENS
      KEY L-name IS Brand-name, model
        01   Brand-name       CHAR      20
        01   Model            CHAR      10
        01   Type             CHAR      10
        01   Macro            BOOLEAN
        01   Min-max          CHAR      10
        01   mm
          02  Lower            FIXED      6
          02  Upper            FIXED      6
        01   F-Stop
          02  Minimum          FLOAT      5,2
          02  Maximum          FLOAT      5,2
        01   Rec-ret-price  FLOAT        6,2
```

```
RECORD Film
     01   Make              CHAR       20
     01 Code                CHAR        5
     01 Size
       02 num               DECIMAL     5,2
       02 Measure           CHAR        6
     01 ASA                 DECIMAL     5
     01 Rec-ret-price       FLOAT       6,2
     01 Type                CHAR       20
     01 Number              DECIMAL     3

RECORD Chemical
     01 Code                CHAR       20
     01 Make                CHAR       20
     01 Type                CHAR       20
     01 Size
       02 num               DECIMAL     5,2
       02 Measure           CHAR        6
     01 REC-ret-price       FLOAT       6,2

RECORD Item
     01 S-name              CHAR       20
     01 S-Code-no           CHAR       10
     01 S-Price             FLOAT       6,2
     01 Item-type           CHAR       10
     01 Item-Code           CHAR       10

RECORD Mount
     01 C-Brandname         CHAR       20
     01 C-Model             CHAR       10
     01 L-Brandname         CHAR       20
     01 L-Model             CHAR       10

RECORD Process
     01 F-Make              CHAR       20
     01 F-Code              CHAR        5
     01 Ch-Make             CHAR       20
     01 Ch-Code             CHAR       20
     01 Ch-Type             CHAR       20
     01 Process             CHAR       20
     01 Result              CHAR       10

SET NAME IS Inventory
    OWNER IS Shop
      ORDER FOR INSERTION IS
      SORTED BY DEFINED KEYS
    MEMBER IS Item
      MEMBERSHIP IS CONTROLLED INSERTION IS AUTOMATIC
                               RETENTION IS FIXED
      KEY IS Item-Code
      SET SELECTION IS
          THRU Inventory OWNER IDENTIFIED BY APPLICATION

SET Supplied-by
    OWNER Shop
      ORDER FIRST
    MEMBER IS Supplier
      MEMBERSHIP IS CONTROLLED INSERTION IS AUTOMATIC
                               RETENTION IS MANDATORY
      SET SELECTION
         THRU Supplied-by OWNER IDENTIFIED BY APPLICATION

SET Supplies-to
    OWNER Manufacturer
      ORDER FIRST
    MEMBER IS Supplier
      MEMBERSHIP IS AUTOMATIC RETENTION IS MANDATORY
      SET SELECTION
         THRU Supplies-to OWNER IDENTIFIED BY APPLICATION

SET Manufacturers
    OWNER Manufacturer
      ORDER SORTED DEFINED RECORD
        SEQUENCE IS Camera,Lens,Film,Chemical
    MEMBER IS Camera
      MEMBERSHIP INSERTION AUTOMATIC OPTIONAL
      KEY ASCENDING Model DUPLICATES NOT
      SELECTION THRU Manufacturer  APPLICATION
    MEMBER IS Lens
```

```
      MEMBERSHIP INSERTION AUTOMATIC OPTIONAL
      KEY ASC Model Duplicates NOT
      SELECTION THRU Manufacturer  APPLICATION
    MEMBER IS FILM
      MEMBERSHIP INSERTION AUTOMATIC OPTIONAL
      KEY ASC Code DUPLICATES FIRST
      SELECTION THRU Manufacturer  APPLICATION
    MEMBER IS Chemical
      MEMBERSHIP INSERTION AUTOMATIC OPTIONAL
      KEY ASC Code DUPLICATES LAST
      SELECTION THRU Manufacturer  APPLICATION

SET C-item
    OWNER Camera
      ORDER NEXT
    MEMBER Item
      MEMBERSHIP INSERTION AUTOMATIC FIXED
        SELECTION THRU C-item APPLICATION

SET L-item
    OWNER Lens
      ORDER NEXT
    MEMBER Item
      MEMBERSHIP INSERTION AUTOMATIC FIXED
        SELECTION THRU L-item APPLICATION

SET F-item
    OWNER Film
      ORDER NEXT
    MEMBER Item
      MEMBERSHIP INSERTION AUTOMATIC FIXED
        SELECTION THRU F-item APPLICATION

SET Ch-item
    OWNER Chemical
      ORDER NEXT
    MEMBER Item
      MEMBERSHIP INSERTION AUTOMATIC FIXED
        SELECTION THRU Ch-item APPLICATION

SET Uses
    OWNER Camera
      ORDER DEFAULT
    MEMBER Mount
      MEMBERSHIP INSERTION MANUAL OPTIONAL
      SELECTION THRU Can-use APPLICATION

SET Used-on
    OWNER Lens
      ORDER DEFAULT
    MEMBER Mount
      MEMBERSHIP INSERTION MANUAL OPTIONAL
      SELECTION THRU Can-be-used-on APPLICATION

SET Processed-by
    OWNER Film
      ORDER DEFAULT
    MEMBER IS Process
      MEMBERSHIP INSERTION MANUAL OPTIONAL
      SELECTION THRU Processed-by APPLICATION

SET Used-in
    OWNER Chemical
      ORDER DEFAULT
    MEMBER IS Process
      MEMBERSHIP INSERTION MANUAL OPTIONAL
      SELECTION THRU Can-process APPLICATION
```

APPENDIX B. VERSION 1 OF THE PHOTOGRAPHIC STORAGE SCHEMA

```
STORAGE SCHEMA NAME IS Photographic VERSION 1
   FOR Photo-Schema SCHEMA

MAPPING FOR SHOP
   STORAGE RECORDS ARE Gen-det, S-addr

MAPPING FOR Camera
  STORAGE RECORDS ARE C-Model, Mod-det

MAPPING FOR Lens
   IF Macro= 'TRUE'
   THEN STORAGE RECORDS ARE L-model, Description, Macro
   ELSE STORAGE RECORDS ARE L-model, Wholedesc

STORAGE AREA Equipment
  INITIAL SIZE IS 9000 PAGES
  PAGE SIZE IS 1024 CHARACTERS

STORAGE AREA F-ch
  INITIAL SIZE IS 10000 PAGES
  PAGE SIZE IS 1024 CHARACTERS

STORAGE AREA M-s
  INITIAL SIZE IS 1000 PAGES
  EXPANDABLE BY 100 PAGES
  PAGE SIZE 256 WORDS

STORAGE RECORD NAME IS Gen-Det
   LINK TO S-Addr IS DIRECT
   PLACEMENT IS CALC Chain USING Chainname
   WITHIN M-S FROM 500 THRU 999
   01 S-Name
   01 Vat-No
   01 Chainname

STORAGE RECORD S-addr
   LINK TO Gen-det
   PLACEMENT IS CALC Town USING Town
   WITHIN M-S FROM 500 THRU 999
   01 Address
      DATA ALL

STORAGE RECORD C-model
   LINK TO Mod-det
   PLACEMENT IS CALC Mode USING Mode
   WITHIN Equipment FROM 1 THRU 499
   01 Brand-name
   01 Model
   01 Mode  OCCURS
   01 Rec-retail-prc


STORAGE RECORD Mod-det
   LINK TO C-model
   DENSITY IS 5 STORAGE RECORDS PER PAGE
   PLACEMENT IS CALC Model2 USING Model
   WITHIN Equipment FROM 499 THRU 998
   01 Model
   01 ASA-range
       DATA ALL
   01 Speed-Range
       DATA ALL
   01 Flash-sync-speed OCCURS
   01 Rec-retail-prc

STORAGE RECORD L-model
   LINK TO Description, Macro is DIRECT
   LINK TO Whole-desc IS INDIRECT
   PLACEMENT IS CALC Models USING Model
   WITHIN Equipment FROM 999 THRU 1498
   01 Brand-name
   01 Model
   01 Type

STORAGE RECORD Description
   LINK TO Macro IS DIRECT
```

```
    DENSITY IS 2 STORAGE RECORDS PER PAGE
    PLACEMENT IS SEQUENTIAL ASCENDING Model
    WITHIN Equipment FROM 7000 THRU 8000
    01  Model
    01  Type
    01  Macro
    01  F-stop
        DATA ALL
    01 mm
        DATA ALL
    01  Rec-ret-price

STORAGE RECORD Macro
    LINK TO L-model IS DIRECT
    PLACEMENT IS CALC Min-max USING Lower, Upper
    WITHIN Equipment FROM 1499 THRU 1998
    01  Model
    01  Type
    01  Min-max
    01  F-stop
        DATA ALL

STORAGE RECORD Wholedesc
    LINK TO L-model
    PLACEMENT SEQUENTIAL ASCENDING Model
    WITHIN Equipment FROM 6000 THRU 6999
    01  Model
    01  Macro
    01  Min-max
    01  F-stop
        DATA ALL
    01 mm
        DATA ALL
    01  Rec-ret-price

STORAGE RECORD Manufacturer
    DENSITY IS 2 STORAGE RECORDS PER PAGE
    PLACEMENT IS SEQUENTIAL ASCENDING M-name
    WITHIN M-s FROM 1 THRU 499
    DATA ALL

STORAGE RECORD Supplier
    PLACEMENT CLUSTERED VIA SET Supplies-to
        NEAR OWNER
    WITHIN M-s FROM 1 THRU 499
    DATA ALL

STORAGE RECORD Film
    DENSITY ONE STORAGE RECORD PER 3 PAGES
    PLACEMENT CALC USING ASA
    WITHIN F-ch FROM 1 THRU 4999
    DATA ALL

STORAGE RECORD Chemical
     PLACEMENT IS SEQUENTIAL ASCENDING Type, Make
     WITHIN F-ch FROM 5000 THRU 9999
     DATA ALL

STORAGE RECORD Item
     PLACEMENT IS CLUSTERED VIA SET Inventory
        NEAR OWNER
     WITHIN M-s
     DATA ALL

STORAGE RECORD Mount
     DENSITY 3 RECORDS PER PAGE
     PLACEMENT IS CLUSTERED VIA SET Uses
        NEAR OWNER
     WITHIN Equipment
     DATA ALL


STORAGE RECORD Process
     DENSITY 3 RECORDS PER PAGE
     PLACEMENT IS CLUSTERED VIA SET Processed-by
        NEAR OWNER
     WITHIN F-ch FROM 1 THRU 4999
     DATA ALL

SET Inventory
```

```
    OWNER
      STORAGE RECORD Gen-det
         POINTER FOR FIRST,LAST MEMBER
         DESTINATION OF DIRECT POINTERS
    MEMBER
      STORAGE RECORD Item
         POINTER FOR NEXT,PRIOR TENANT
         DESTINATION OF DIRECT POINTERS

SET Supplied-by
    OWNER
      STORAGE RECORD Gen-det
         POINTER FOR FIRST MEMBER
      STORAGE RECORD S-addr
         POINTER FOR LAST MEMBER
         DESTINATION OF DIRECT POINTERS
    MEMBER
      STORAGE RECORD Supplier
         POINTER FOR NEXT,PRIOR TENANT
         DESTINATION OF DIRECT POINTERS

SET Supplies-to
    OWNER
      STORAGE RECORD Manufacturer
         POINTER FOR FIRST MEMBER
    MEMBER
      STORAGE RECORD Supplier
         POINTER FOR NEXT TENANT,OWNER

SET Manufacturers
    OWNER
      STORAGE RECORD Manufacturer
         POINTER FOR INDEX Man-index
    MEMBER RECORD Camera
      STORAGE RECORD C-model
         POINTER FOR INDEX Man-index,OWNER
         DESTINATION OF DIRECT POINTERS
    MEMBER RECORD Lens
      STORAGE RECORD Description
         POINTER FOR OWNER
         DESTINATION OF DIRECT POINTERS
      STORAGE RECORD L-model
         POINTER FOR INDEX Man-index
      STORAGE RECORD Wholedesc
         POINTER FOR OWNER
         DESTINATION OF INDIRECT POINTERS
    MEMBER RECORD Film
      STORAGE RECORD Film
         POINTER FOR INDEX Man-index, OWNER
    MEMBER RECORD Chemical
      STORAGE RECORD Chemical
         POINTER FOR INDEX Man-index,OWNER

SET C-item
    OWNER
      STORAGE RECORD C-model
         POINTER FOR INDEX Cit-index
         DESTINATION OF DIRECT POINTERS
    MEMBER
      STORAGE RECORD Item
         POINTER FOR INDEX Cit-index,OWNER
         DESTINATION OF DIRECT POINTERS

SET L-item
    OWNER
       STORAGE RECORD L-model
          DESTINATION OF DIRECT POINTERS
          POINTER FOR INDEX Lit-index
    MEMBER
       STORAGE RECORD Item
          DESTINATION OF DIRECT POINTERS
          POINTER FOR INDEX Lit-index, OWNER

SET F-item
    OWNER
       STORAGE RECORD Film
          POINTER FOR FIRST,LAST
    MEMBER
       STORAGE RECORD Item
          POINTER FOR NEXT, PRIOR
```

```
SET Ch-item
    OWNER
       STORAGE RECORD Chemical
          POINTER FOR FIRST
    MEMBER
       STORAGE RECORD Item
          POINTER FOR NEXT

SET Uses
    OWNER
       STORAGE RECORD Mod-det
          DESTINATION OF DIRECT POINTERS
          POINTER FOR FIRST
    MEMBER
       STORAGE RECORD Mount
          POINTER FOR NEXT

SET Used-on
    OWNER
       STORAGE RECORD L-model
          DESTINATION OF DIRECT POINTERS
          POINTER FOR FIRST
    MEMBER
       STORAGE RECORD Mount
          POINTER FOR NEXT

SET Processed-by
    OWNER
       STORAGE RECORD Film
          POINTER FOR FIRST
    MEMBER
       STORAGE RECORD Process
          POINTER FOR NEXT

SET Used-in
    OWNER
       STORAGE RECORD Chemical
          POINTER FOR FIRST
    MEMBER
       STORAGE RECORD Process
          POINTER FOR NEXT

INDEX Wdesc-ind
    USED FOR STORAGE KEY Whole-desc
    WITHIN Equipment

INDEX Man-index
    PLACEMENT IS NEAR OWNER DISPLACEMENT -2 PAGES
    USED FOR SET Manufacturers
       KEY
       LINK TO OWNER
    WITHIN STORAGE AREA OF OWNER

INDEX Cit-index
    PLACEMENT IS NEAR OWNER DISPLACEMENT 2 PAGES
    USED FOR SET C-item LINK TO OWNER
    WITHIN STORAGE AREA OF OWNER

INDEX Lit-index
    PLACEMENT IS NEAR OWNER
    USED FOR SET L-item LINK TO OWNER
    WITHIN STORAGE AREA OF OWNER

INDEX Ind-man
    USED FOR RECORD Manufacturer
       SCHEMA KEY Manu-code
    WITHIN M-S


INDEX Ind-shop
    USED FOR RECORD Shop
       SCHEMA KEY Shop-code
       POINTER IS DIRECT TO Gen-det
    WITHIN M-S FROM PAGE 100 THRU 200

INDEX Ind-cam
    USED FOR RECORD Camera
       SCHEMA KEY C-name
       POINTER IS DIRECT TO Mod-det
```

APPENDIX C. VERSION 2 OF THE PHOTOGRAPHIC STORAGE SCHEMA

```
STORAGE SCHEMA NAME IS Photographic VERSION 2
    FOR Photo-Schema SCHEMA

MAPPING FOR SHOP
    STORAGE RECORDS ARE Gen-det, S-addr

MAPPING FOR Camera
    STORAGE RECORDS ARE C-Model, Mod-det

MAPPING FOR Lens
    IF Macro= 'TRUE'
    THEN STORAGE RECORDS ARE L-model, Description, Macro
    ELSE STORAGE RECORDS ARE L-model, Wholedesc

MAPPING VERSION 2 FOR Item
    STORAGE RECORDS ARE S-item, I-item

STORAGE AREA Equipment
    INITIAL SIZE IS 9000 PAGES
    PAGE SIZE IS 1024 CHARACTERS

STORAGE AREA F-ch
    INITIAL SIZE IS 10000 PAGES
    PAGE SIZE IS 1024 CHARACTERS

STORAGE AREA M-s
    INITIAL SIZE IS 1000 PAGES
    EXPANDABLE BY 100 PAGES
    PAGE SIZE 256 WORDS

STORAGE AREA Items
    INITIAL SIZE IS 1000 PAGES
    EXPANDABLE
    PAGE SIZE IS 256 WORDS

STORAGE RECORD NAME IS Gen-Det
    LINK TO S-addr IS DIRECT
    PLACEMENT IS CALC Chain USING Chainname
    WITHIN M-S
    01 S-Name
    01 Vat-No
    01 Chainname

STORAGE RECORD S-addr
    LINK TO Gen-det
    PLACEMENT IS CALC Town USING Town
    WITHIN M-S
    01 Address
        DATA ALL

STORAGE RECORD C-model
    LINK TO Mod-det
    PLACEMENT IS CALC Mode USING Mode
    WITHIN Equipment
    01 Brand-name
    01 Model
    01 Mode  OCCURS
    01 Rec-retail-prc

STORAGE RECORD Mod-det
    LINK TO C-model
    DENSITY IS 5 STORAGE RECORDS PER PAGE
    PLACEMENT IS CLUSTERED
    01 Model
    01 ASA-range
        DATA ALL
    01 Speed-Range
        DATA ALL
    01 Flash-sync-speed OCCURS
    01 Rec-retail-prc

STORAGE RECORD L-model
    LINK TO Description, Macro is DIRECT
    LINK TO Whole-desc IS INDIRECT
    PLACEMENT IS CALC Models USING Model
    WITHIN Equipment
```

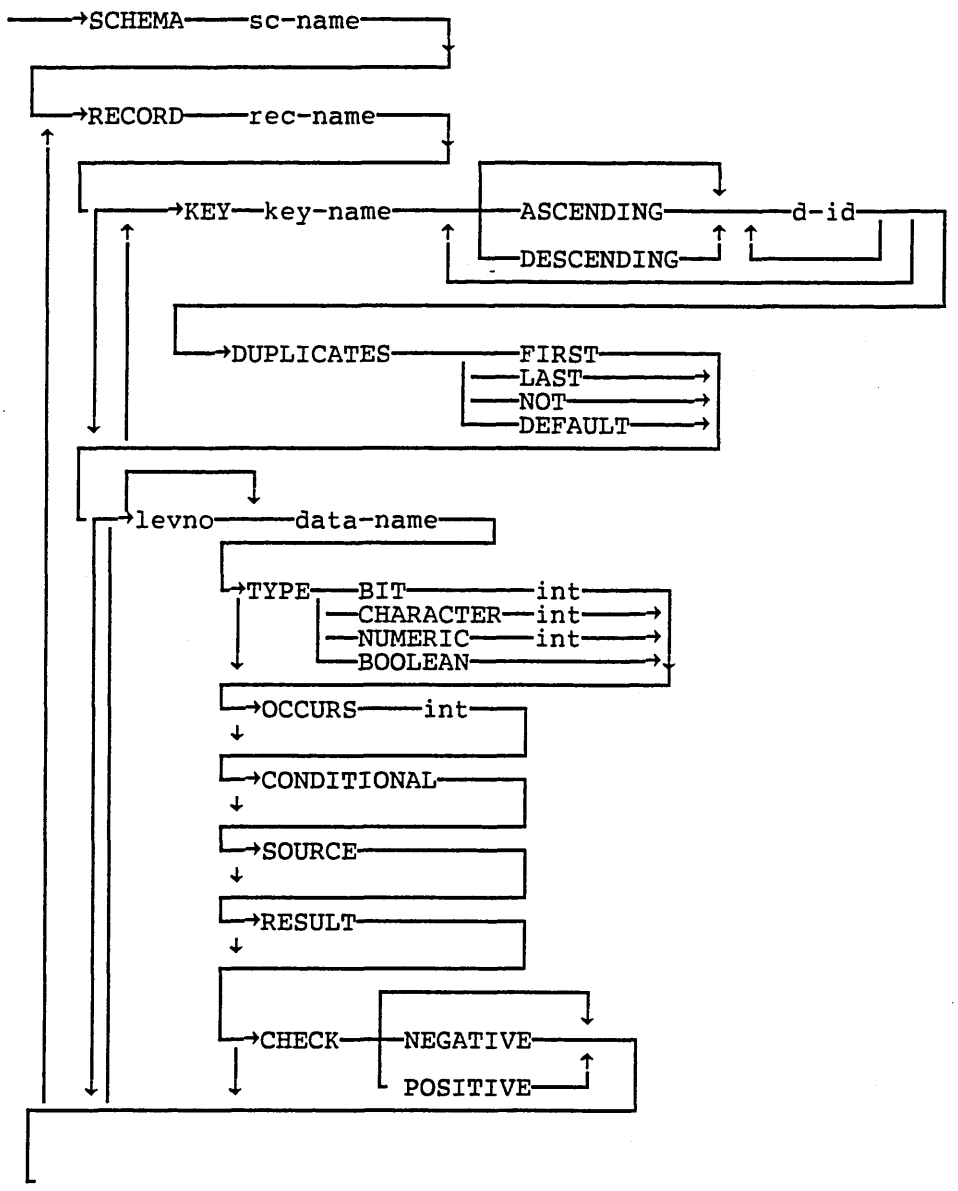
```
    01 Brand-name
    01 Model
    01 Type

STORAGE RECORD Description
    LINK TO Macro IS DIRECT
    DENSITY IS 2 STORAGE RECORDS PER PAGE
    PLACEMENT IS CLUSTERED VIA SET Manufacturers
       NEAR OWNER
    WITHIN Equipment
    01  Model
    01  Type
    01  Macro
    01  F-stop
      DATA ALL
    01 mm
      DATA ALL
    01  Rec-ret-price

STORAGE RECORD Macro
    LINK TO L-model IS DIRECT
    PLACEMENT IS CALC Min-max USING Lower, Upper
    WITHIN Equipment
    01  Model
    01  Type
    01  Min-max
    01  F-stop
      DATA ALL

STORAGE RECORD Wholedesc
    LINK TO L-model
    PLACEMENT SEQUENTIAL ASCENDING Model
    WITHIN Equipment
    01  Model
    01  Macro
    01  Min-max
    01  F-stop
      DATA ALL
    01 mm
      DATA ALL
    01  Rec-ret-price

STORAGE RECORD Manufacturer
    DENSITY IS 2 STORAGE RECORDS PER PAGE
    PLACEMENT IS SEQUENTIAL ASCENDING M-name
    WITHIN M-s FROM 1 THRU 499
    DATA ALL

STORAGE RECORD Supplier
    PLACEMENT CLUSTERED VIA SET Manufacturers
       NEAR OWNER
    WITHIN M-s FROM 1 THRU 499
    DATA ALL

STORAGE RECORD Film
    DENSITY ONE STORAGE RECORD PER 3 PAGES
    PLACEMENT CALC USING ASA
    WITHIN F-ch FROM 1 THRU 499
    DATA ALL

STORAGE RECORD Chemical
    PLACEMENT IS SEQUENTIAL ASCENDING Type, Make
    WITHIN F-ch FROM 5000 THRU 9999
    DATA ALL

STORAGE RECORD Chemical VERSION 2
    PLACEMENT IS CALC Codes USING Code
    WITHIN Items FROM PAGE 500 THRU 1000
    DATA ALL

STORAGE RECORD Item
    PLACEMENT IS CLUSTERED VIA SET Inventory
       NEAR OWNER
    WITHIN M-s
    DATA ALL

STORAGE RECORD S-item VERSION 2
    LINK TO I-item IS INDIRECT
    PLACEMENT CLUSTERED VIA SET Inventory
       NEAR OWNER
```

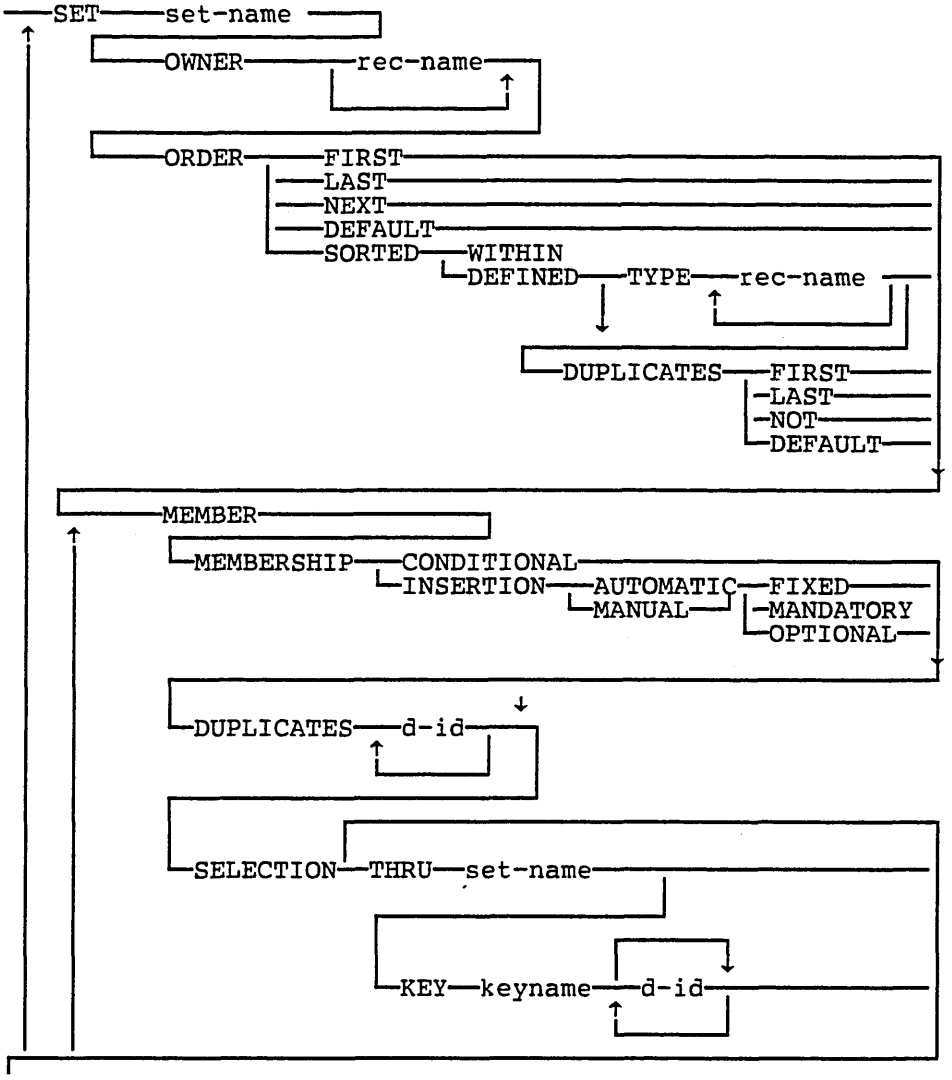
```
    WITHIN M-s
    01 S-name
    01 S-code-no
    01 S-price

STORAGE RECORD I-item VERSION 2
    LINK TO S-item IS INDIRECT
    DENSITY IS 3 STORAGE RECORDS PER PAGE
    PLACEMENT IS
       CALC calcIcode USING Item-code
    WITHIN Items FROM 1 THRU 401
    Item-type
    Item-code

STORAGE RECORD Mount
    DENSITY 3 RECORDS PER PAGE
    PLACEMENT IS CLUSTERED VIA SET Uses
       NEAR OWNER
    WITHIN Equipment
    DATA ALL


STORAGE RECORD Process
    DENSITY 3 RECORDS PER PAGE
    PLACEMENT IS CLUSTERED VIA SET Processed-by
       NEAR OWNER
    WITHIN F-ch FROM 1 THRU 4999
    DATA ALL

SET Inventory
   OWNER
     STORAGE RECORD Gen-det
        POINTER FOR FIRST,LAST MEMBER
        DESTINATION OF DIRECT POINTERS
   MEMBER
     STORAGE RECORD Item
        POINTER FOR NEXT,PRIOR TENANT
        DESTINATION OF DIRECT POINTERS

SET Inventory VERSION 2
   OWNER
     STORAGE RECORD Gen-det
        POINTER FOR FIRST,LAST MEMBER
        DESTINATION OF DIRECT POINTERS
   MEMBER
     STORAGE RECORD S-item
        DESTINATION OF INDIRECT POINTERS
        POINTER FOR NEXT, PRIOR MEMBER

SET Supplied-by
   OWNER
     STORAGE RECORD Gen-det
        POINTER FOR FIRST MEMBER
     STORAGE RECORD S-addr
        POINTER FOR LAST MEMBER
        DESTINATION OF DIRECT POINTERS
   MEMBER
     STORAGE RECORD Supplier
        POINTER FOR NEXT,PRIOR TENANT
        DESTINATION OF DIRECT POINTERS

SET Supplies-to
   OWNER
     STORAGE RECORD Manufacturer
        POINTER FOR FIRST MEMBER
   MEMBER
     STORAGE RECORD Supplier
        POINTER FOR NEXT TENANT,OWNER

SET Manufacturers
   OWNER
     STORAGE RECORD Manufacturer
        POINTER FOR INDEX Man-index
   MEMBER RECORD Camera
     STORAGE RECORD C-model
        POINTER FOR INDEX Man-index,OWNER
        DESTINATION OF DIRECT POINTERS
   MEMBER RECORD Lens
     STORAGE RECORD Description
        POINTER FOR OWNER
```

```
        DESTINATION OF DIRECT POINTERS
      STORAGE RECORD L-model
        POINTER FOR INDEX Man-index
      STORAGE RECORD Wholedesc
        POINTER FOR OWNER
        DESTINATION OF INDIRECT POINTERS
    MEMBER RECORD Film
      STORAGE RECORD Film
        POINTER FOR INDEX Man-index, OWNER
    MEMBER RECORD Chemical
      STORAGE RECORD Chemical
        POINTER FOR INDEX Man-index,OWNER

SET C-item
    OWNER
      STORAGE RECORD C-model
        POINTER FOR INDEX Cit-index
        DESTINATION OF DIRECT POINTERS
    MEMBER
      STORAGE RECORD Item
        POINTER FOR INDEX Cit-index,OWNER
        DESTINATION OF DIRECT POINTERS

SET C-item VERSION 2
    OWNER
      STORAGE RECORD C-model
        POINTER FOR INDEX Cit-index
        DESTINATION OF DIRECT POINTERS
    MEMBER
      STORAGE RECORD S-item
        POINTER FOR INDEX Cit-index, OWNER

SET L-item
    OWNER
       STORAGE RECORD L-model
         DESTINATION OF DIRECT POINTERS
         POINTER FOR INDEX Lit-index
    MEMBER
       STORAGE RECORD Item
         DESTINATION OF DIRECT POINTERS
         POINTER FOR INDEX Lit-index, OWNER

SET L-item VERSION 2
    OWNER
       STORAGE RECORD L-model
         DESTINATION OF DIRECT POINTERS
         POINTER FOR INDEX Lit-index
    MEMBER
       STORAGE RECORD I-item
         DESTINATION OF INDIRECT POINTERS
         POINTER FOR INDEX Lit-index, OWNER

SET F-item
    OWNER
       STORAGE RECORD Film
         POINTER FOR FIRST,LAST
    MEMBER
       STORAGE RECORD Item
         POINTER FOR NEXT, PRIOR

SET F-item VERSION 2
    OWNER
       STORAGE RECORD Film
         POINTER FOR FIRST, LAST
    MEMBER
       STORAGE RECORD S-item
         DESTINATION OF DIRECT POINTERS
         POINTER FOR NEXT

SET Ch-item
    OWNER
       STORAGE RECORD Chemical
         POINTER FOR FIRST
    MEMBER
       STORAGE RECORD Item
         POINTER FOR NEXT

SET Ch-item VERSION 2
    OWNER
       STORAGE RECORD Chemical
```

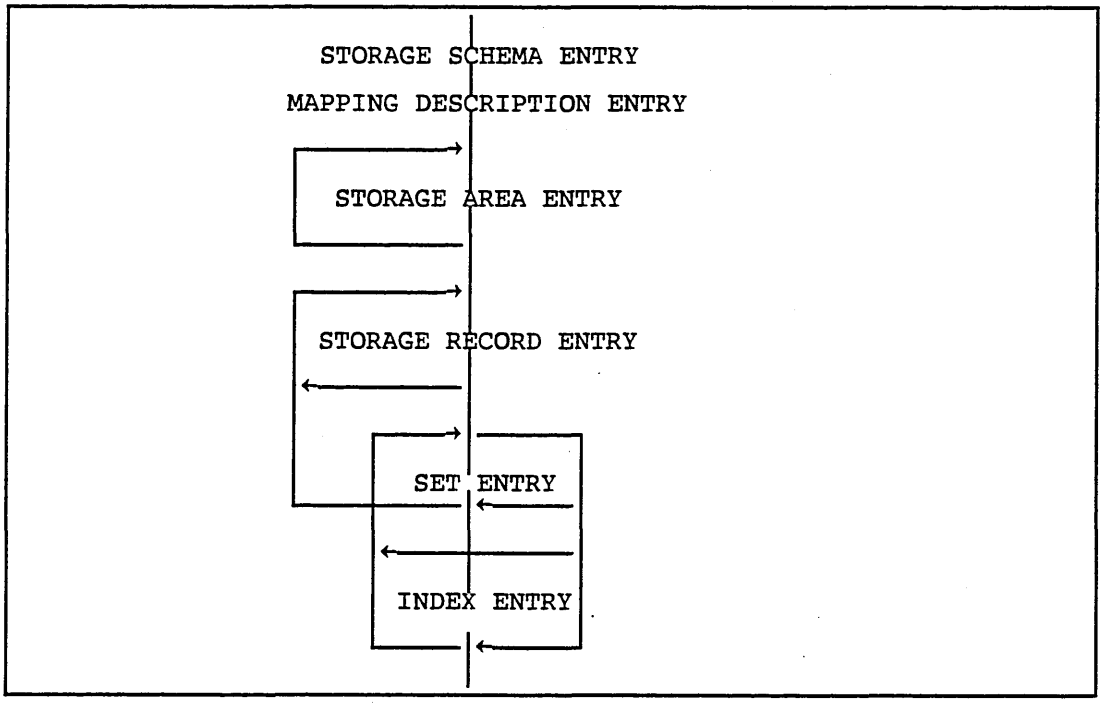
```
        POINTER FOR FIRST
    MEMBER
       STORAGE RECORD S-item
         POINTER FOR NEXT

SET Uses
    OWNER
       STORAGE RECORD Mod-det
         DESTINATION OF DIRECT POINTERS
         POINTER FOR FIRST
    MEMBER
       STORAGE RECORD Mount
         POINTER FOR NEXT

SET Used-on
    OWNER
       STORAGE RECORD L-model
         DESTINATION OF DIRECT POINTERS
    MEMBER
       STORAGE RECORD Mount
         POINTER FOR NEXT

SET Processed-by
    OWNER
       STORAGE RECORD Film
         POINTER FOR FIRST
    MEMBER
       STORAGE RECORD Process
         POINTER FOR NEXT

SET Used-in
    OWNER
       STORAGE RECORD Chemical
         POINTER FOR FIRST
    MEMBER
       STORAGE RECORD Process
         POINTER FOR NEXT

INDEX Wdesc-ind
    USED FOR STORAGE KEY Whole-desc
    WITHIN Equipment

INDEX Man-index
    PLACEMENT IS NEAR OWNER DISPLACEMENT -2 PAGES
    USED FOR SET Manufacturers
       KEY
       LINK TO OWNER
    WITHIN STORAGE AREA OF OWNER

INDEX Cit-index
    PLACEMENT IS NEAR OWNER DISPLACEMENT 2 PAGES
    USED FOR SET C-item LINK TO OWNER
    WITHIN STORAGE AREA OF OWNER

INDEX Lit-index
    PLACEMENT IS NEAR OWNER
    USED FOR SET L-item LINK TO OWNER
    WITHIN STORAGE AREA OF OWNER
```

APPENDIX D. SCHEMA NDL ORIENTED TOKENS IGNORING COMPLICATED SET SELECTION



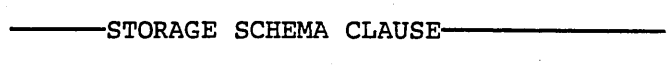

SET
set-name
OWNER
rec-name
ORDER
FIRST
LAST
NEXT
DEFAULT
SORTED
WITHIN
DEFINED
TYPE
rec-name
DUPLICATES
FIRST
LAST
NOT
DEFAULT
MEMBER
MEMBERSHIP
CONDITIONAL
INSERTION
AUTOMATIC
MANUAL
FIXED
MANDATORY
OPTIONAL
DUPLICATES
d-id
SELECTION
THRU
set-name
KEY
keyname
d-id

E.1 DSDL Overall Structure Syntax Graphs



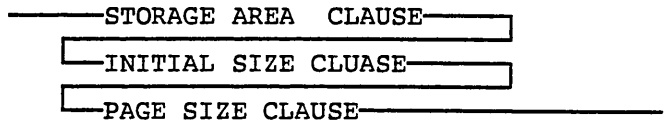
E.2 DSDL Overall Subentry Structure Graphs

STORAGE SCHEMA ENTRY

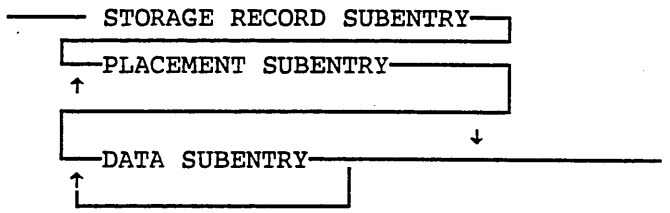


MAPPING DESCRIPTION ENTRY

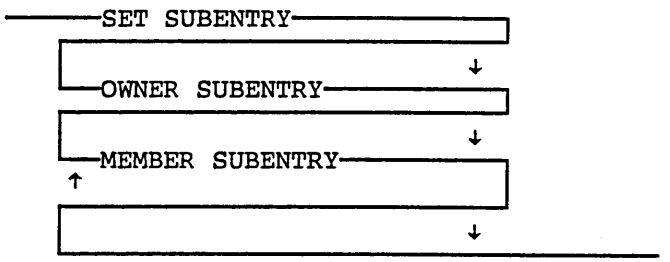
STORAGE AREA ENTRY



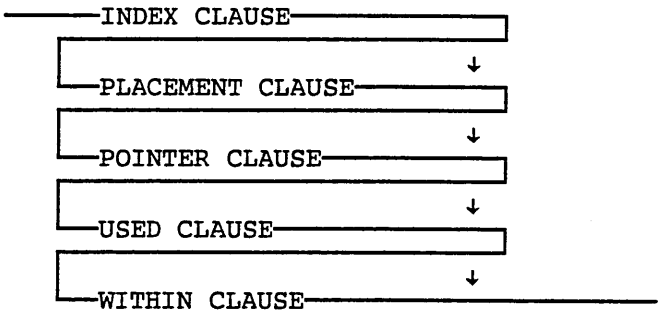
STORAGE RECORD ENTRY

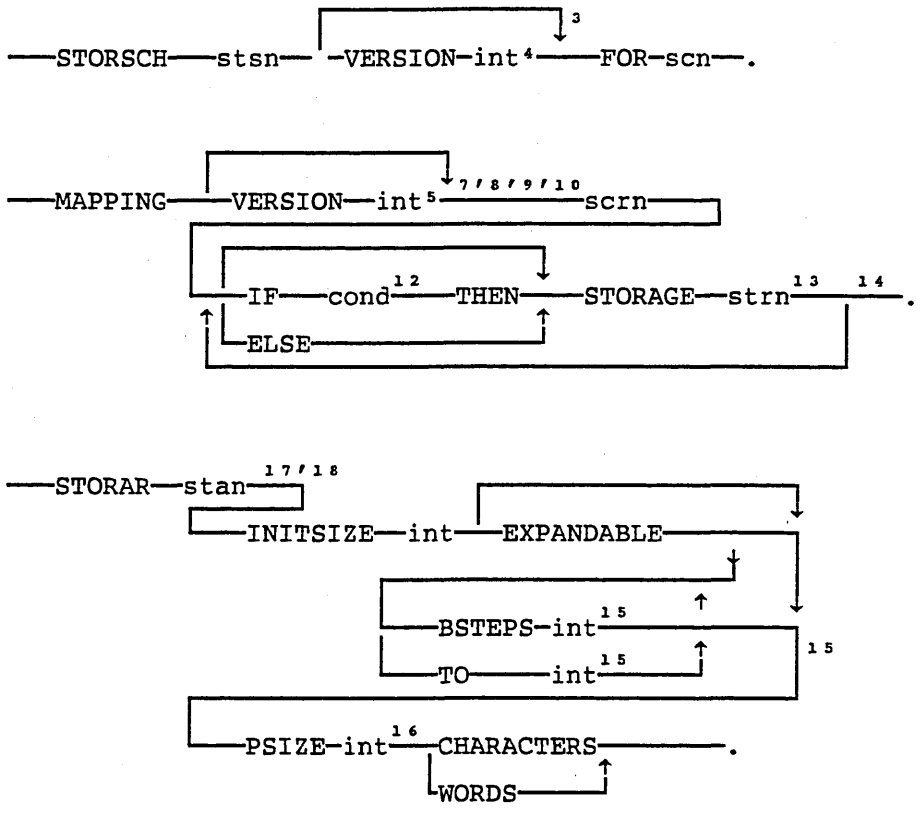


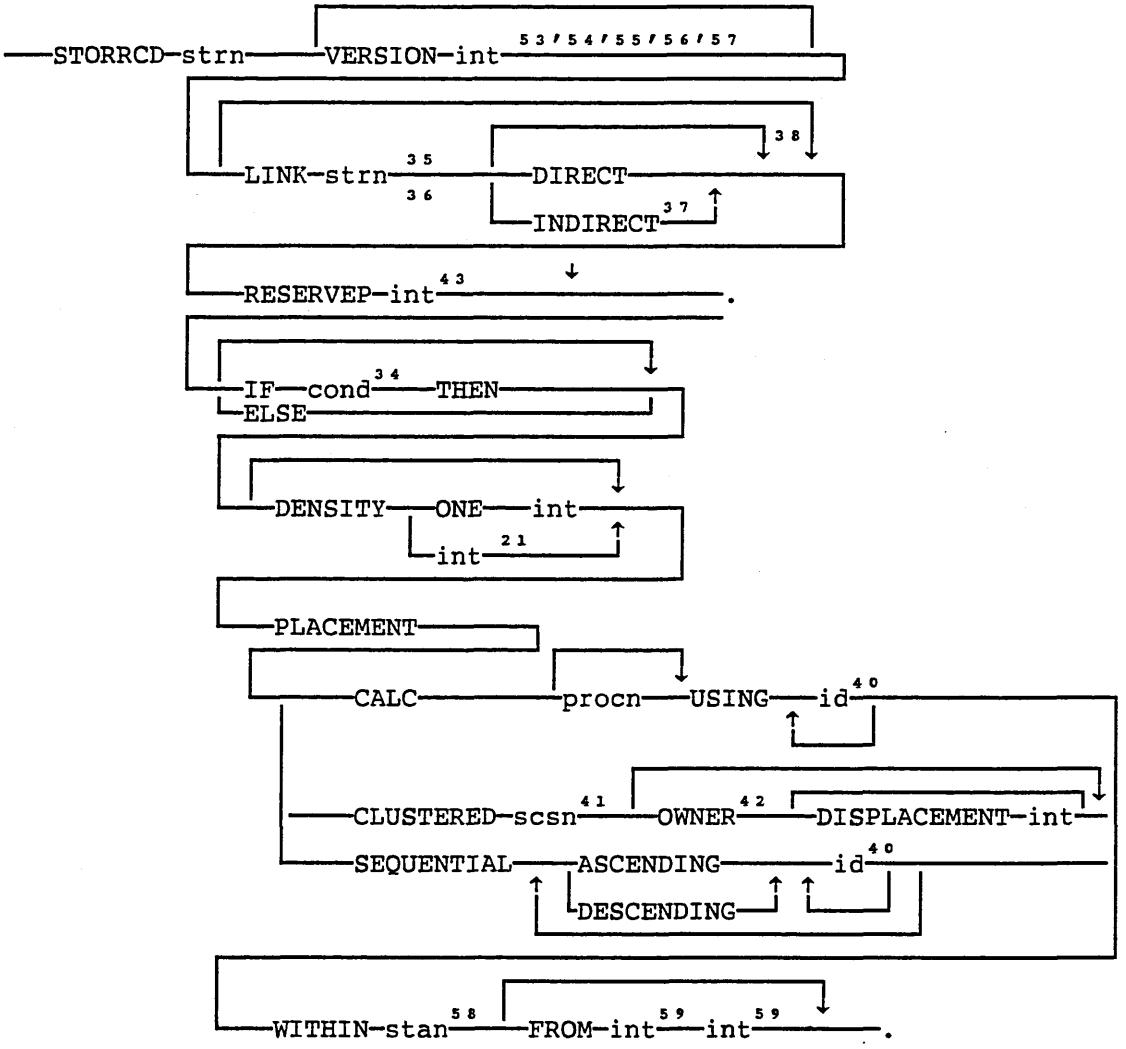
SET ENTRY

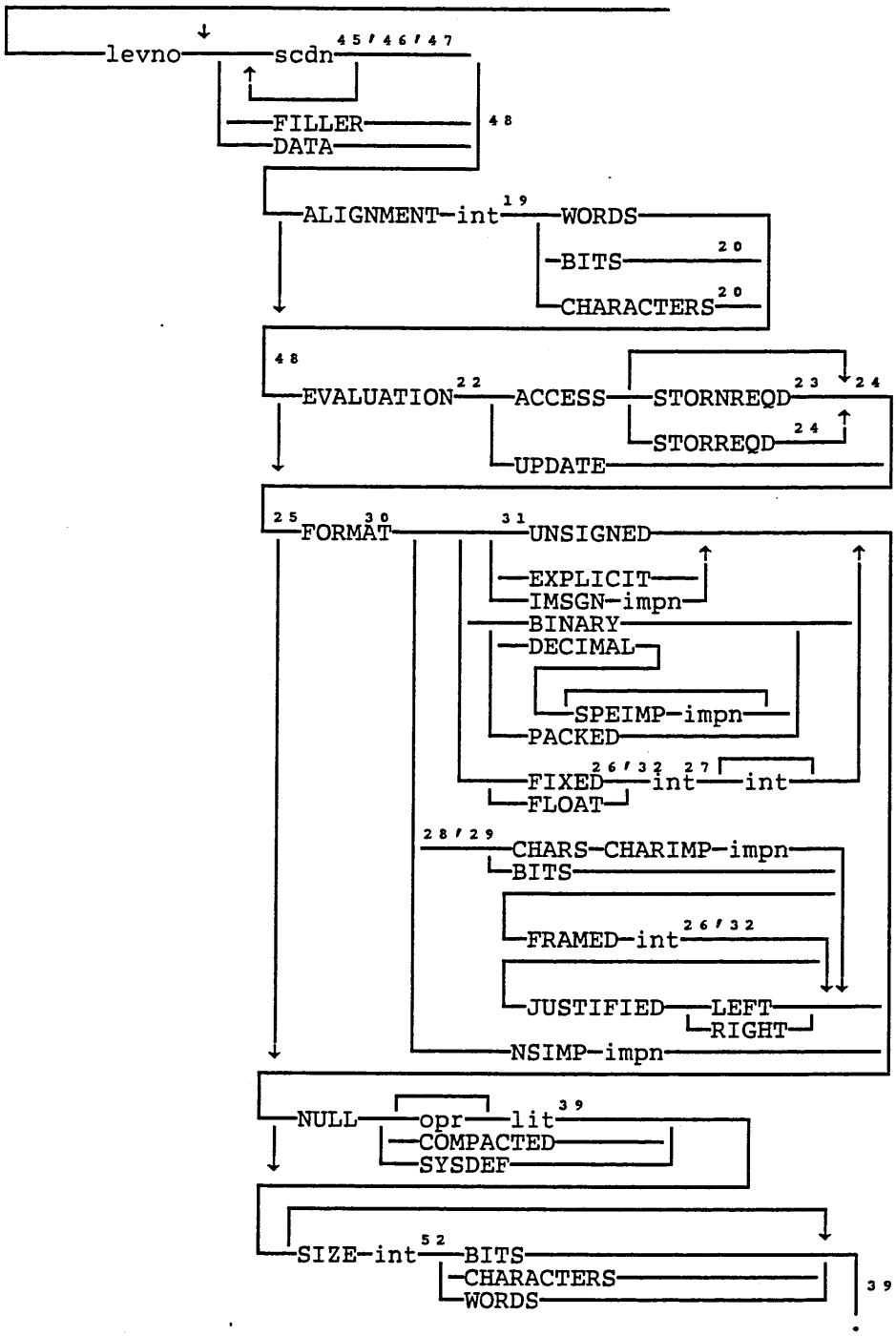


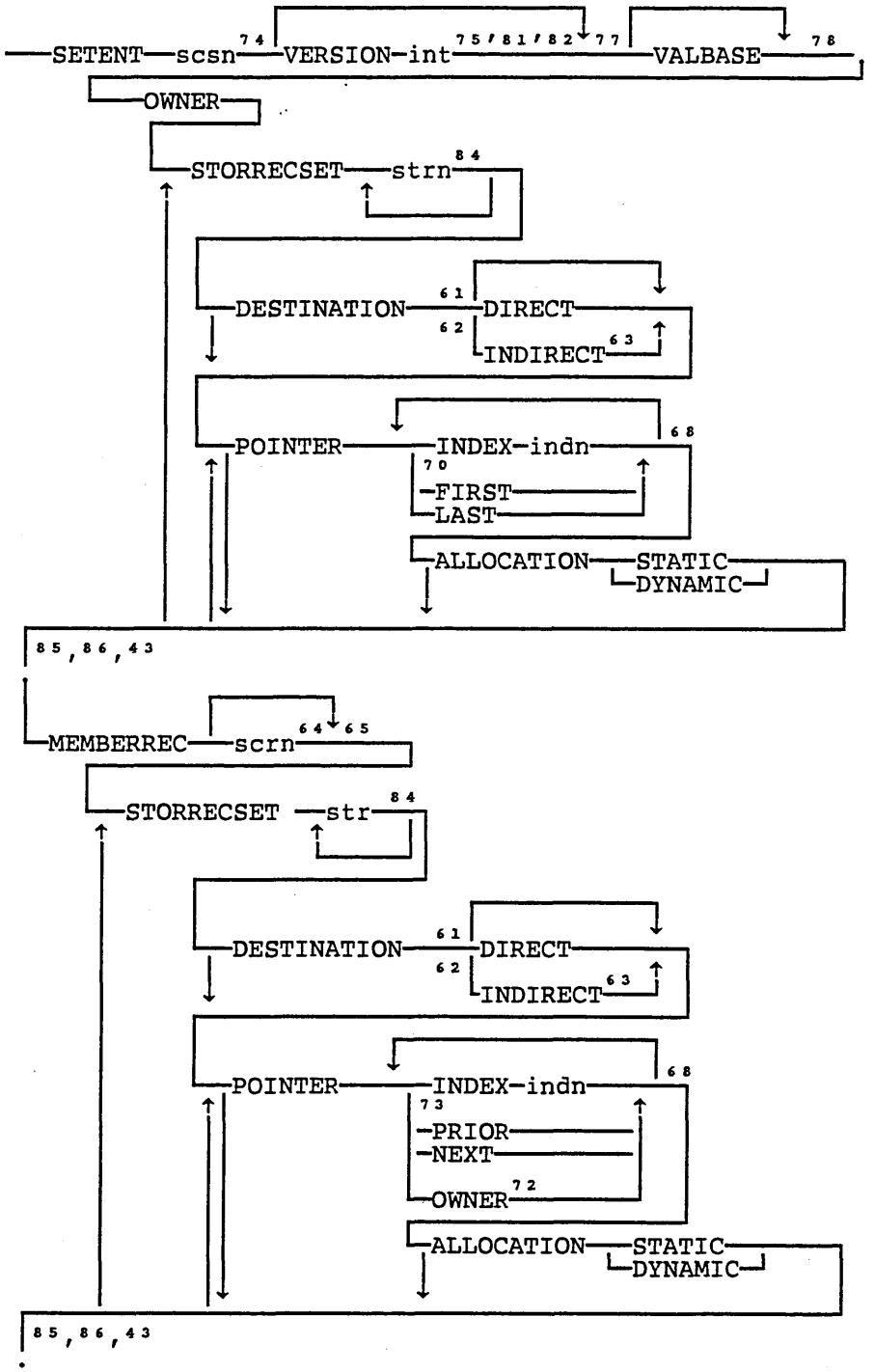
INDEX ENTRY




STORSCH
stsn
VERSION
int
FOR
scn
MAPPING
VERSION
int
scrn
IF
cond
THEN
STORAGE
strn
ELSE
STORAR
stan
INITSIZE
int
EXPANDABLE
BSTEPS
int
TO
int
PSIZE
int
CHARACTERS
WORDS


STORRCD-strn
VERSION-int
53/54/55/56/57
LINK-strn
35
36
DIRECT
38
INDIRECT
37
RESERVEP-int
43
.
IF
cond
34
THEN
ELSE
DENSITY
ONE
int
int
21
PLACEMENT
CALC
procn
USING
id
40
CLUSTERED-scsn
41
OWNER
42
DISPLACEMENT-int
SEQUENTIAL
ASCENDING
DESCENDING
id
40
WITHIN-stan
58
FROM-int
59
int
59
.


levno
scdn
45/46/47
FILLER
DATA
48
ALIGNMENT-int
19
WORDS
BITS
20
CHARACTERS
20
48
EVALUATION
22
ACCESS
STORNREQD
23
24
STORREQD
24
UPDATE
25
FORMAT
30
31
UNSIGNED
EXPLICIT
IMSGN-impn
BINARY
DECIMAL
SPEIMP-impn
PACKED
FIXED
26/32
int
27
int
FLOAT
28/29
CHARS-CHARIMP-impn
BITS
FRAMED-int
26/32
JUSTIFIED
LEFT
RIGHT
NSIMP-impn
NULL
opr
lit
39
COMPACTED
SYSDEF
SIZE-int
52
BITS
CHARACTERS
WORDS
39


SETENT
scsn
74
VERSION-int
75/81/82
77
VALBASE
78
OWNER
STORRECSET
strn
84
DESTINATION
61
DIRECT
62
INDIRECT
63
POINTER
INDEX-indn
68
70
FIRST
LAST
ALLOCATION
STATIC
DYNAMIC
85,86,43
MEMBERREC
scrn
64
65
STORRECSET
str
84
DESTINATION
61
DIRECT
62
INDIRECT
63
POINTER
INDEX-indn
68
73
PRIOR
NEXT
OWNER
72
ALLOCATION
STATIC
DYNAMIC
85,86,43

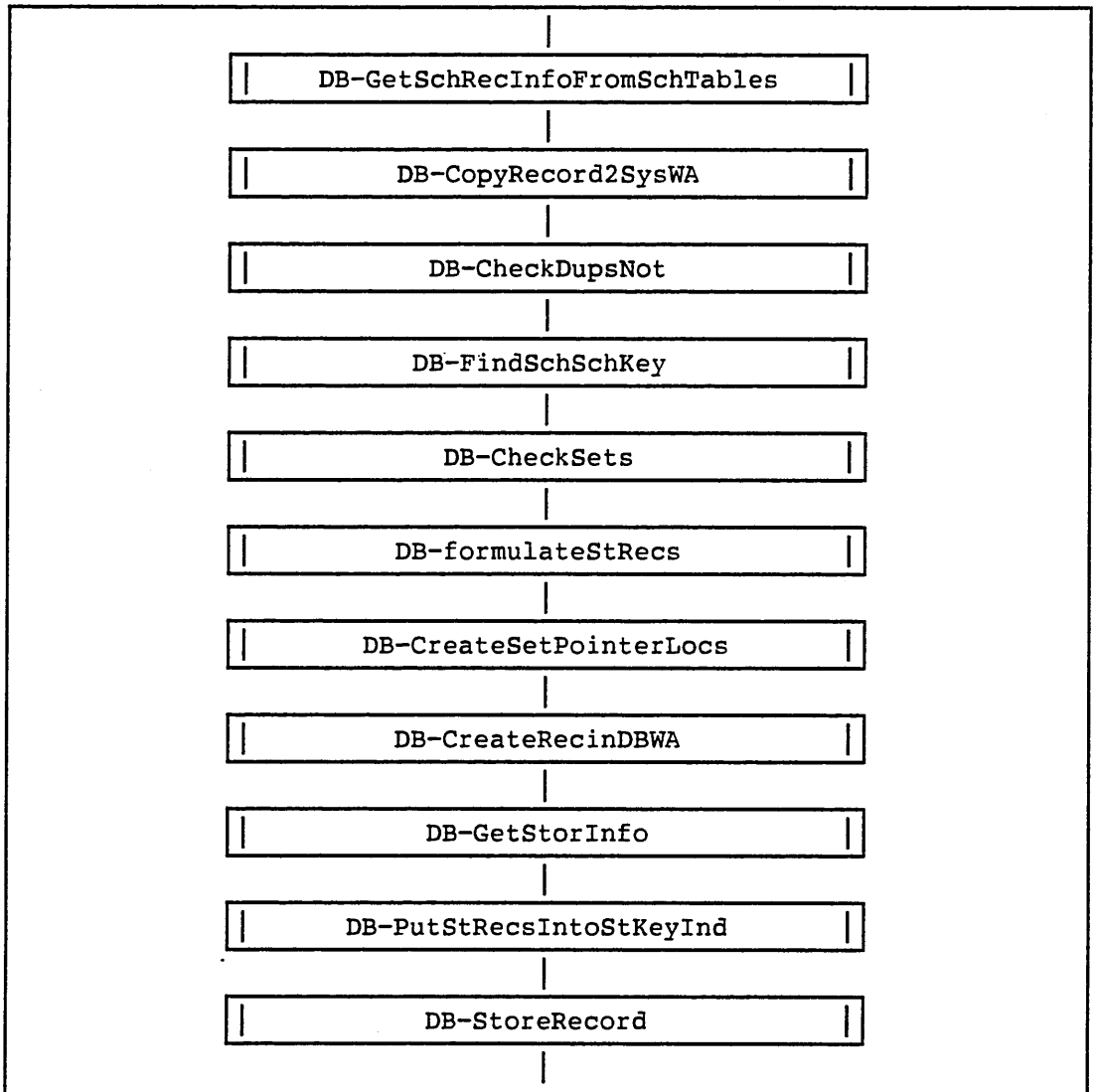


Figure 49. Modular Diagram of the DML verb STORE

APPENDIX H. PAPERS WRITTEN BY THE AUTHOR

ALZ-WP-8001 A fourteen page document of DSDL editing errors and questionable ambiguities. (Put to the DDLC via DBAWG and ABERDEEN proposals.)

ALZ-WP-8101 Reference is made to the DDL for all conditions, but words used in conditions are not present in the DSDL reserved word list. (DBAWG-8104)

ALZ-WP-8102 The 'MEMBER' ambiguity.

ALZ-WP-8103 The ambiguous syntax points of the DSDL.

ALZ-WP-8104 The Set clause (syntax rule 4 part b) referenced a constraint no longer present in the DDL.

ALZ-WP-8105 The Within clause, demonstrated an inconsistency in its integer specification.

ALZ-WP-8106 The Syntax description of the Set Entry was incorrect, a variety of correct syntaxes were proposed.

ALZ/SJC-WP-8201 Further to ALZ-WP-8106 the draft for a DBAWG paper on the Set Entry as subentries was complied. This includes the Member ambiguity.

ZORNER,A.L. "A Design for an Implementation of a Runtime System to Support Dynamic Incremental Foreground Reorganisation in a Network Database System" Proc. of the Third British National Conference on Databases.11-13 July 1984.

APPENDIX I. BIBLIOGRAPHY

a. *Papers and Articles*

1. Addyman,A.M. et al "A Draft Description of PASCAL." Software - Practice and Experience Vol 9 1979 pp381-424

2. Anzelmo,F.D. (member IEEE) "A Data Storage Format for Information System Files." IEEE Trans on Computers Vol C-20 No1 Jan 1971

3. Arnow,D. & Tenenbaum,A.M. "An Empirical Comparison of B-trees, compact B-trees, and multiway trees." Sigmod Record Vol 14 No 2 pp33-46 Proceedings of Annual Meeting Boston MA June 18-21 1984

4. B.C.S./CODASYL DDLC DBAWG "Database Administration working group June 1975 report" June 1975.

5. Bell,D.A. & Deen,S.M. "Hash Trees versus B-Trees." The Computer Journal Vol 27 No3 1984

6. Belyeu,S.M. "Hardware Approach to Storage Mapping." IBM Technical Disc. Bulletin Vol 21 No1 June 1978

7. Bernstein,P.A. & Goodman,N. "Multiversion Concurrency Control - Theory and Algorithms." ACM trans on Database Systems Vol 8 No4 Dec 1983 pp465-483

8. Blank,B.G. "Distributed Quasi-ordered and Ordered Data Storage Structures in Direct Access Store." Programmirovanie No5 Sept-Oct 1979 pp56-65

9. Blasgen,B.W. & Eswaran,K.P. "Storage and Access in Relational Databases." IBM Syst 3 No4 1977 pp363-377

10. Blasgen,M.W. et al "System R: An architectural Overview." IBM Syst 3 1981 Vol 20 No1 pp41-62

11. Boral,H., Delwitt,D.J., Friedland,D., Jamell,N.F. & Wilkinson,W.K. "Implementation of the database machine DIRECT." IEEE Trans on Software Engineering Vol SE-8 No6 Nov 1982 pp533-543

12. Brodie,M.L. "Research on the translation and Standardisation of Relational and Network Type Database Management Systems." US Army Research Final Report date 19 March 1981

13. Brodie,M.L. et al "ANSI/X3/SPARC DBS-SG Relational Database Task Group (final report)." Standards Institute New York Sept 81 NBS/GCR-82/379 PB82-170051

14. Buneman,P., Frankel,R.E. & Nikhil,R. "An Implementation Technique for Database Query Languages." ACM Trans Database Syst Vol 7 No2 June 1982 ppl64-180

15. Pin-Shan Chen,P. "The Entity-Relationship Model - Toward a Unified View of Data." ACM Trans on Database Syst Vol 1 No1 March 1976 27 pages

16. CODASYL Data Base Task Group. "CODASYL DBTG Report." Oct 1969

17. CODASYL Data Base Task Group "CODASYL DBTG Report" Apr 1971

18. CODASYL DDLC. Standing Paper 14 "Charter for the Data Base Administration Task Group" Aug 1974.

19. Chen,P.P. & Bingyao,S. "Design & Performance Tools for database systems." Pro Conf on Very Large Databases - Tokyo Oct 1977 pp3-15

20. Chen,Huei-huang & McCure Kuck,S. "Combining Relational and Network Retrieval Methods." ACM 1984 ppl31-142

21. Comer,D. "The Ubiquitous B-Tree." Computer Surveys Vol 11 No2 June 1979 ppl21-137

22. Crennell,K.M. et al "Report of the SRC Working Party on Databases and Database Management Systems." Rutherford Appleton Laboratories Oct 1980

23. DBAWG. "DBAWG-SP24.1 Draft SQL Data Storage Description Language" Sep 1986.

24. Dewitt,D.J. & Hawthorn,P.B. "A Performance Evaluation of Database Machine Architectures." University of Wisconsin Madison - Computer Sciences Technical Report No 437 June 1981

25. Dewitt,D.J. et al "Implementation Techniques for Main Memory Database Systems." Sigmod Record Vol 14 No2 pp1-8 Proceedings of Annual Meeting Boston MA June 18-21 1984

26. Dewitt,D.J. & Hawthorn,P.B. "Performance of Database Machine Architectures.(U) Wisconsin University

27. Diehr,G. & Faaland,B. "Optimal Pagination of B-trees with variable length items." Comms of ACM Vol 27 No3 Mar 1984 pp241-247

28. Elmasri,R., Devor,C. & Rahimi,S. "Notes on DDTS - An Apparatus for Experimental Research in Distributed Database Management Systems." Honeywell Corporate Computer Sciences Centre - Bloomington Minnesota pp32-49

29. Feborowicz,J. "A Zipfian Model of an Automatic Bibliographic System: An application to medicine." J. of American Society for Information Science July 1982 pp223-232

30. Flower,R.A. "An Analysis of Optimal Retrieval Systems with Updates." National Technical Information Service Report no. TR-488 MIT June 1975 14 refs

31. Fung,K.T. "A Reorganisation Model based on the Database Entropy Concept." The Computer Journal Vol 27 No1 1984 pp67-71

32. Fung,K.T. "An approach to reorganisation in a Database (Conference paper)." Soc for Gen Syst Res Procs 26th Annual Meeting of Soc for Gen Syst Res (with AAAS) Jan 1982 Vol 1,61 pp311-314

33. Germans,F. & Higgenbotham,S. "A Student Use Hierarchical Database Management System." Dept. of Computer & Information Sciences, Temple University, Philadelphia

34. Graef,N., Kretschonar,H., Loehr,K.P. & Morawetz,B. "How to Design and Implement Time-Sharing Systems Using Concurrent PASCAL." Software - Practice and Experience Vol 9 1979 pp17-24

35. Hall,D.E., Scherrer,D.K. & Sventek,J.S. "A Virtual Operating System." Comms of ACM Sept 1980 Vol23 No9 pp495-502

36. Hanson,P.B. "The Programming Language Concurrent PASCAL." IEEE Trans on Software Engineering Vol SE-1 No2 1975 pp199-207

37. Held,G.D., Stonebraker,M.R. & Wong,E. "INGRES - A Relational Database System." Vol 1 Database Management Systems AFIPS Press pp37-44

38. Hong,Y.C. & Su,S.Y.W. "Associative Hardware and Software Techniques for Integrity Control." ACM Trans on Database Systems Sept 1981 Vol 6 No3 pp416-440

39. James,E.B. "The User Interface." The Computer Journal Vol 23 No1 pp25-28

40. Jeffery,K.G.,Gill,E.M. "The Design Philosophy of the G-EXEC System." Computers and Geosciences, Vol2 1976 pp345-346

41. Katz,R.H. & Wong,E. "Resolving Concepts in Global Storage: Design through replication." ACM Trans on Database Systems March 1983 Vol 8 No1 pp110-135

42. Katz,R.H. & Lehman,T.J. "Database Support for Versions and Alternatives of Large Design Files." IEEE Trans on Software Engineering March 1984 Vol SE-10 No2 pp191-200

43. Kay,M.H. "An Assessment of the CODASYL DDL for use with a relational subschema." Computer Laboratory Univ. of Cambridge Sept 1984

44. Kerridge,J.M. "A FORTRAN Implementation of Concurrent PASCAL." Software - Practice and Experience Vol 12 1982 pp45-55

45. Kerridge,J.M. "An Architecture and Syntax for Distributed Databases." North Holland Computers & Standards 3 1984 pp33-56

46. Kollias,J.G. "File Organisations and their Reorganisation." Inform Systems 1979 Vol 4 pp49-54

47. Kompelmakher,V. & Listovets,V.A. "Database models and loading methods." Translated from Programmirovarie Vol 5 No5 Sept 1979 pp348-353

48. Kung,H.T. & Lehman,P.L. "Concurrent Manipulation of Binary Search Trees." ACM Trans of Database Systems 1980 Vol 5 No3 pp354-382

49. Leung,C.H.C. & Choo,Q.M. "The Effect of Fixed Length Record Implementation on file system Response." Acta Informatica Vol 17 1982 pp399-409

50. Leung,C.H.C. "Optimal Database Reorganisation: some practical difficulties." Inf Processing Letters 19 Aug 82 Vol 15 Nol

51. Lions,J. "Experiences with the UNIX Time-Sharing System." Software - Practice and Experience Vol 9 1979 pp701-709

52. Lochovsky,F.H. & Tsichritzis,D.C. "Teaching Data Management using an Educational Database Management System." Computer Systems Research Group, University of Toronto,Canada.

53. Lucking,J.R. "Database Language, in particular DDL, development at CODASYL ICL no details

54. Major,J.B. "Processor, I/O Path, and DASD configuration capacity." IBM Syst 3 1981 Vol 20 Nol pp63-85

55. Managaki,M., Doine,T., Toshimoto,H. & Katayama,H. "A Structural Model for System Implementation and its Application to CODASYL-DBTG." Vol 1 Database Management Systems AFIPS Press pp51-58

56. Manhood,D.W. "Storage Level Control of a CODASYL database: Part I." Computer Bulletin Sept 1980 pp6-10

57. Manhood,D.W. "Storage Level Control of a CODASYL database: Part II." Computer Bulletin Dec 1980 pp4-5

58. March,S.T., Severence,D.G. & Wilins,M. "Frame Memory: A Storage Architecture to Support Rapid Design and Implementation of Efficient Databases." ACM Trans on Database Systems Sept 1981 Vol 6 No3 pp441-463

59. Maskell,R. "Lexicon - An established Data Dictionary System." Database Journal Vol 6 No7 pp15-21

60. McLeod,D. & Heimbigner,D. "A federated architecture for Database Systems." National Computer Conference 1982 pp283-289

61. Narayana,K.T., Prasad,V.R. Joseph,M. "Some Aspects of Concurrent Programming in CCNPASCAL." Software - Practice and Experience 1979 Vol 9 pp749-770

62. Neal,D. & Wallentine,V. "Experiences with the Portability of Concurrent PASCAL." Software - Practice and Experience 1978 Vol 8 pp341-353

63. Oppen,D.C. "Reasoning About Recursively Defined Data Structures." JACM Vol 27 No3 July 1980 pp403-411

64. Personal communications. "At inaugural DB2 User Group Meeting" London 5th Dec 1986.

65. Prowse,P.H. & Johnson,R.G. "A Natural Language Database Interface to the User." The Computer Journal Vol 23 No1 pp22-25

66. Read,B.J., "A Relational Data Handling System for Scientists." Proc. of the Fifth British National Conf. on Databases (BNCOD5),1986,pp23-41

67. Reisner,P., Boyce,R.F. & Chamberlin,D.D. "Human Factors Evaluation of two data base query languages - Square and Sequel." Vol 1 Database Management Systems AFIPS Press pp71-76

68. Roussopoulos,N. "The Logical Access path Schema of a Database." IEEE Trans on Software Engineering Vol SE-8 No6 Nov 1982 pp563-573

69. Rustin,R. "Database Management: An Overview." Chase Manhattan Bank,NY no further details

70. Sale,A.H.J. "Strings and Sequence Abstraction in PASCAL." Software - Practice and Experience Vol 9 1979 pp671-683

71. Seaman,R.P. "Algorithm for Mapping self-describing records." IBM Technical Disc. Bulletin Vol 18 No11 April 1976

72. Senko,M.E. & Altman,E.B. "DiamII and levels of Abstraction The physical device level: a general model for access methods." Systems for Large Databases. North Holland Publishing Co. 1976 pp79-83

73. Sethi,R. & Tang,A. "Constructing Call-by-Value Continuation Semantics." JACM July 1980 Vol 27 No3 pp580-597

74. Sherman,S. & Werth,J. "Relationship between Database Systems and Operating Capabilities: Stage One - the Survey." National Bureau of Standards, Washington 25 Mar 1982 NTIS PB82-238932

75. Shu,N.C., Housel,B.C. & Lum,V.Y. "CONVERT: A High Level Translation Definition Language for Data Conversion." ACM comms Oct 1975 Vol 18 No10

76. Sibley,E.H. & Taylor,R.W. "A Data Definition and Mapping Language." ACM comms Dec 1973 Vol 16 No12 pp750-759

77. Sicherman,G.L., de Jonge,W. & Van de Riet,K. "Answering Queries without Revealing Secrets." ACM Trans Dist. Syst Vol 8 No1 March 1983 pp41-59

78. Smith,D. et al "A Component Architecture for Database Management Systems (interim report)." Computer corp. of America, Cambridge, MA, 18 June 1980 NBS-GCR-81-340 PB82-203621

79. Sockut,G.H. "A Performance Model for Computer Database Reorganisation performed Concurrently with usage." Operations Research 1978 pp789-804

80. Sockut,G.H. & Goldberg,R.P. "Database Reorganisation - principles and practice." NBS Special Publication, US Dept of Commerce, National Bureau of Standards pp500-547

81. Sockut,G.H. & Goldberg,R.P. "Database Reorganisation - principles and practice." ACM Computer Surveys Vol 11 No4 Dec 1979

82. Soderlund,L. "Concurrent Database Reorganisation - Assessment of a Powerful Technique through Modelling." Procs 7th Int. Conf. on Very Large Databases, Cannes Sept 1981 pp499-509

83. Soderlund,L. "Evaluation of concurrent physical database reorganisation through simulation modelling." ACM Sigmetrics Vol 10 No3 Sept 1981 pp19-32

84. Su,S.Y.W., Nguyen,L.H., Emam,A. & Lipovski,G.J. "Architectural Features and Implementation Techniques of the Multicell CASSM." IEEE Trans on Computers Vol C-28 No6 June 1979

85. Stonebraker,M. et al "Performance Enhancements to a Retrieval Database System." ACM Trans Database Systems Vol 8 No2 June 1983 pp167-185

86. Uemura,S., Yuba,T., Kokuba,A., Ooomote,R. & Sugawara,Y. "The Design and Implementation of a Magnetic-Bubble Database Machine." Procs of the IFIP Congress Information Processing, Tokyo Oct 1980 pp433-438

87. Uemura,S., Yuba,T., Kokubu,A. & Ooomote,R. "Database Machine with Magnetic-Bubble Memory." Technologies North-Holland, Amsterdam 1982 pp39-50

88. Whang et al "Separability - An Approach to Physical Database Design." IEEE Trans on Computers Vol C-33 No3 March 1984 pp209-222

89. Wilson,T.B. "Database Restructuring: Options and Obstacles." Euro IFIP, North Holland Publishing Co. 1979 pp567-573

90. Wilson,T.B. "The Description and Usage of Evolving Schemas." IEEE Oct 1980

91. X3H2 ANSC "Overview of DBCS/Programming Language Interface." ANSC X3H2 Feb 1982 Doc X3H2-22-8(R)

92. Yao,S.B., Das,K.S. & Teorey,T.J. "A Dynamic Database Reorganisation Algorithm." ACM Trans on Database Systems Vol 1 No2 June 1976 pp159-174

93. Yao,S.B. "Optimisation of Query Evaluation Algorithms." ACM Trans on Database Systems June 1979 Vol 4 No2 pp133-135

94. Yao,S.B, Henver,A.R. & Romeo,T. "Performance Evaluation of Database Systems - a benchmark methodology." US Dept of Commerce, NBS Washington, DC NBS-GCR 84-467 May 1984 PB84-217504

95. Yao,S.B. et al "Analysis of Three Database System Architectures using Benchmarks (final report)." Software Systems Technology Inc. College Park, MD May 1984 NBS/GCR-84/468 PB84-217512

96. Zahle,T.U. "Scan - A simple record-at-a-time DML for the relational data model."

97. Zave,P. "The operational versus the conversational approach to software development." Comms. of the ACM Feb 1984 Vol 27 No2

98. Zorner,A.L. "A Design for an Implementation of a Runtime System to support Dynamic Incremental Foreground Reorganisation of a Network Database System." Proc. of the Third British National Conference on Databases. 11-13 July 1984.

b. *Books and Manuals.*

1. American National Standards Committee X3H2-83-121. "April 1983 (Draft Proposed) Network Database Language"

2. American National Standards Committee X3H2-86-26. "March 1986 (Draft Proposed) Network Database Language"

3. Bourne,F.R. "The UNIX system" Addison-Wesley Pub Comp. 1982

4. CODASYL Data Definition Language Committee. "DDLC Journal of Development." Aug 1973

5. CODASYL. "Data Description Language Committee DDL & DSDL JOD" Secretariat of the Canadian Government EDP 1978

6. CODASYL. "Data Description Language Committee DDL & DSDL JOD" Secretariat of the Canadian Government EDP 1981

7. CODASYL. "COBOL Committee JOD" Secretariat of the Canadian Government EDP 1981

8. CODASYL. "Data Manipulation Language and Subschema " Secretariat of the Canadian Government EDP 1981

9. ICL IDMS Appendix E: Space Management.

10. Date,C.J. "Database - A Primer." Addison-Wesley Popular Series 1983

11. Date,C.J. "An Introduction to Database Systems." 2nd & 3rd editions Addison-Wesley 1981 Syst Prog Services Chps 2,26

12. Date,C.J. "An Introduction to Database Systems." Vol II Addison-Wesley 1983

13. Davis,B. "The Selection of Database Software." NCC Publications 1977

14. Draffan,I.W.&Poole,F. "Distributed Data Bases : An advanced course." Cambridge University Press 1981

15. Dean,S.M. "Fundamentals of Database Systems." Macmillan Press LTD 1977

16. Gries,D. "Compiler Construction for Digital Computers." Wiley International Edition 1971

17. IBM. "SQL/Data System Terminal User's Reference for VM/System Product" Release 3 pn 5748-XXJ

18. IBM. "CA77 SQL/DS Implementation & Management in a VM Environment - student notes" UK33-5551 1985

19. INMOS. "Transputer Reference Manual" INMOS LTD. 1985

20. ISO 8907-1987(E) "Database Language NDL" November 1986

21. ISO 9075-1987(E) "Database Language SQL" November 1986

22. Larson,J.A. "Database Management System Anatomy." Lexington Books 1982 D.C. Heath & co

23. Olle,T.W. "The CODASYL Approach to Database Management." John Wiley 1980

24. ORACLE. "ORACLE SQL/UFI Reference Guide" 1984 ORACLE Corp., Menlo Park, California

25. Palmer,I. "Database Systems: A practical Reference." CACI 1975

26. Proc. "Proc. of Symposium on Developement and Management of a Computer-centered Data Base 1963" System Development Corporation, Santa Monica, California. 1964

27. Rohl,J.S. "An introduction to compiler Writing." Macdonald and Jane's/American Elsevier 1975

28. Robinson,H. "Database Analysis and Design. Chartwell-Bratt 1981

29. Schmidt,J.W. & Brodie,M.L. "Relational Database Systems Analysis and Comparison." Springer-Verlag

30. Shneiderman,B. (ed) "Database Management Systems." Vol I The Information Technology Series AFIPS Press

31. Tsichritzis,D.C. & Bernstein "Operating Systems" Academic Press 1974

32. Tsichritzis,D.C. & Lochovsky,F.H. "Database Management Systems." Academic Press 1977

33. Uemura,S. et al "Database Machine with Magnetic-Bubble Memory." 1.3 Computer Science and Technologies, North-Holland, Amsterdam 1982

34. Ullman,J.D. "Principles of Database Systems." 2nd edition Pitman Publishing Ltd 1983

35. Wierderhold,G. "Database Design." 2nd edition McGraw-Hill

36. Zloof,M.M. "Query by Example." Vol 1 Database Management Systems AFIPS Press pp63-70

APPENDIX J. DETAILS OF RELATED STUDIES

The references given in Appendix I have all been accessed as part of the course of guided study.

The candidate has attended 34 meetings of the DBAWG, 11 meetings of the BSI and 4 meetings of ISO TC97/SC21/WG3. These meetings involved contributions to the development of database languages and standards.

- 13-14 Nov 1980 BCS/CODASYL DBAWG at BCS HQ. (LONDON)
- 13-14 Jan 1981 BCS/CODASYL DBAWG at Prime (BEDFORD)
- 15-16 Mar 1981 BCS/CODASYL DBAWG at Sheffield City Polytechnic
- 4- 5 Jun 1981 BCS/CODASYL DBAWG at Hatfield Polytechnic
- 27-28 Aug 1981 BCS/CODASYL DBAWG at ERCC
- 8-11 Oct 1981 BCS/CODASYL DBAWG at The Burn (ABERDEEN)
- 18-18 Dec 1981 BCS/CODASYL DBAWG at Open University (LONDON)
- 11-12 Feb 1982 BCS/CODASYL DBAWG at UNIVAC (LONDON)
- 15-16 Apr 1982 BCS/CODASYL DBAWG at D.C.E. (HOLLAND)
- 10-11 Jun 1982 BCS/CODASYL DBAWG at Woolwich Polytechnic
- 16-19 Sep 1982 BCS/CODASYL DBAWG at The Burn (ABERDEEN)
- 18-19 Nov 1982 BCS/CODASYL DBAWG at ICL (BRACKNELL)
- 13-14 Jan 1983 BCS/CODASYL DBAWG at SCICON
- 10-11 Mar 1983 BCS/CODASYL DBAWG at OU (Milton Keynes)
- 26-27 May 1983 BCS/CODASYL DBAWG at Sheffield City Polytechnic
- 7- 8 Jul 1983 BCS/CODASYL DBAWG at OUCC (OXFORD)
- 1- 4 Sep 1983 BCS/CODASYL DBAWG at The Burn (ABERDEEN)
- 10-11 Nov 1983 BCS/CODASYL DBAWG at Software Sciences
- 19-20 Jan 1984 BCS/CODASYL DBAWG at SPERRY (LONDON)
- 22-23 Mar 1984 BCS/CODASYL DBAWG at CAST (EDINBURGH)
- 31- 1 Jun 1984 BCS/CODASYL DBAWG at DCE (HOLLAND)
- 6- 9 Sep 1984 BCS/CODASYL DBAWG at The Burn (ABERDEEN)
- 8- 9 Nov 1984 BCS/CODASYL DBAWG at ICL (LONDON)
- 23 Nov 1984 BSI DBMS Panel at BSI HQ (LONDON)
- 7 Jan 1985 BSI DBMS Panel at BSI HQ (LONDON)
- 24-25 Jan 1985 BCS/CODASYL DBAWG at NCC (MANCHESTER)
- 5- 8 Feb 1985 ISO WG5-15 at Bougeval (FRANCE)
- 14-15 Mar 1985 BCS/CODASYL DBAWG at Sheffield City Polytechnic
- 13 May 1985 BSI DBMS Panel at BSI HQ (LONDON)
- 23-24 May 1985 BCS/CODASYL DBAWG at OXFORD
- 18-19 Jul 1985 BCS/CODASYL DBAWG at Hatfield Polytechnic
- 5- 8 Sep 1985 BCS/CODASYL DBAWG at The Burn (ABERDEEN)
- 7 Oct 1985 BSI DBMS Panel at BSI HQ (LONDON)
- 22 Oct 1985 BSI SQL Rapp. Gp. at ICI (MANCHESTER)
- 5- 8 Nov 1985 ISO TC97/SC21/WG3 at Gaithesburg (USA)

- 21-22 Nov 1985 BCS/CODASYL DBAWG at CCTA (NORWICH)
- 7 Jan 1986 BSI SQL Rapp. Gp. at ICI (MANCHESTER)
- 10 Feb 1986 BSI SQL Rapp. Gp. at ICI (MANCHESTER)
- 3- 4 Apr 1986 BCS/CODASYL DBAWG at DCE (HOLLAND)
- 21-24 Apr 1986 ISO DBL Rapp. Gp. at SEIMANS (MUNICH)
- 29-30 May 1986 BCS/CODASYL DBAWG at ERCC (EDINBURGH)
- 10 Jun 1986 BSI SQL Rapp. Gp. at ICI (MANCHESTER)
- 17-18 Jul 1986 BCS/CODASYL DBAWG at Sperry (LONDON)
- 29 Jul 1986 BSI SQL Rapp. Gp. at ICI (MANCHESTER)
- 28 Aug 1986 BSI SQL Rapp. Gp. at ICI (MANCHESTER)
- 4- 7 Sep 1986 BCS/CODASYL DBAWG at The Burn (ABERDEEN)
- 15-19 Sep 1986 ISO TC97/SC21/WG3 at Royal Holloway College
- 17 Oct 1986 BSI SQL Rapp. Gp. at ICI (MANCHESTER)
- 13-14 Nov 1986 BCS/CODASYL DBAWG at NCC (MANCHESTER)