Technical Report

UCAM-CL-TR-10
ISSN 1476-2986

Number 10

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Prediction oriented description of database systems

Mark Theodore Pezarro

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted October 1978 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Darwin College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

SUMMARY

A descriptive model of database systems is presented. The model is intended to provide a general framework for the description of database systems which is not limited to any particular DBMS or even to any of the three mainstream approaches to DBMS architecture. This generality is derived from a new analysis of file organisation methods on which the model is based. The model concentrates on the aspects of a database system relevant to first-order performance prediction. These include database structure, the hardware and software used in implementing the system, the size of the database at various points in its lifetime, and its known or anticipated usage. Particular attention has been devoted to arriving at a general treatment of the details of database systems at the physical level, including access paths and their encoding, storage devices and their operating characteristics, and the mapping of data representations to storage devices.

A formal language has been devised in which to write textual descriptions of a database system in terms of the model. In addition, an experimental prediction program has been written which accepts a description of a database system expressed in the language and produces performance estimates for the described activity using computational methods based on expected-value formulae. Some preliminary results obtained by comparing estimates given by the program with measurements of an operational database system are presented. Further experimentation that would allow a definitive evaluation of the prediction program is outlined and a review is made of the current limitations of the model and program with suggestions for further research.

CONTENTS

FIGURES

TABLES

1. INTRODUCTION

The research described here is concerned with the development of a general model for the description of database systems. Identification of the aspects of a database system most significant with regard to performance prediction has been emphasized. This work has two principal motivations.

The first is a belief that a general model is an aid to a better understanding of existing systems. Describing these systems in terms of a neutral canonical model enables a clearer understanding of their similarities and differences. The benefits of such an exercise should include making it easier to appreciate and learn from results published about other systems and to recognise techniques employed elsewhere that could be usefully implemented in one's own system.

The second is to explore the possibility of developing a non-"DBMS-specific" prediction program based on computational methods and expected-value estimates. The entire database system is to be textually, rather than procedurally, described and equations, rather than simulation, are to be used to estimate the average time taken in performing basic database accessing functions. There are two aims here. The primary one is to build a program that is cheap to use in the sense that it runs in seconds rather than minutes or hours. A secondary one is to keep the amount of input data to the minimum necessary to get useful results.

1.1 Scope of application

This research is intended to be applicable to what are usually called integrated database systems. A *DBS* (database system) is here

defined as a computerised system for managing a large collection of data. The term computerised system is meant to refer to the collection of hardware, software, and data (stored on the hardware by the software) which altogether acts as a vehicle for the storage, retrieval and updating of facts. An *integrated DBS* is one in which the data is used for diverse purposes by a number of different users. These users are assumed to belong to some sort of common organisation.

There are three important implications arising from the above definitions.

1. The database is a communal resource and so must be managed in a way that optimises overall performance for everyone.

2. The database is a *large* collection of data which means that there are genuine difficulties in arriving at a physical database organisation that permits efficient and rapid access and is at the same time cost-effective.

3. Points 1 and 2 imply a large scale design effort is worthwhile. A large scale design effort is costly; the existence of a common organisational framework means the necessary financial support can be found.

These points have touched on the design and physical organisation of databases. This ties in with the previously mentioned emphasis on performance prediction.

An interest in design and performance evaluation suggests the concept of a general model for describing integrated DBSs and a program to accept such descriptions and produce performance estimates for the described DBSs. Focusing on physical organisation and physical database design leads to a particular effort to accommodate as rich a variety of physical organisation and designs as possible within the framework of the model. Figure 1.1 illustrates the use envisaged for

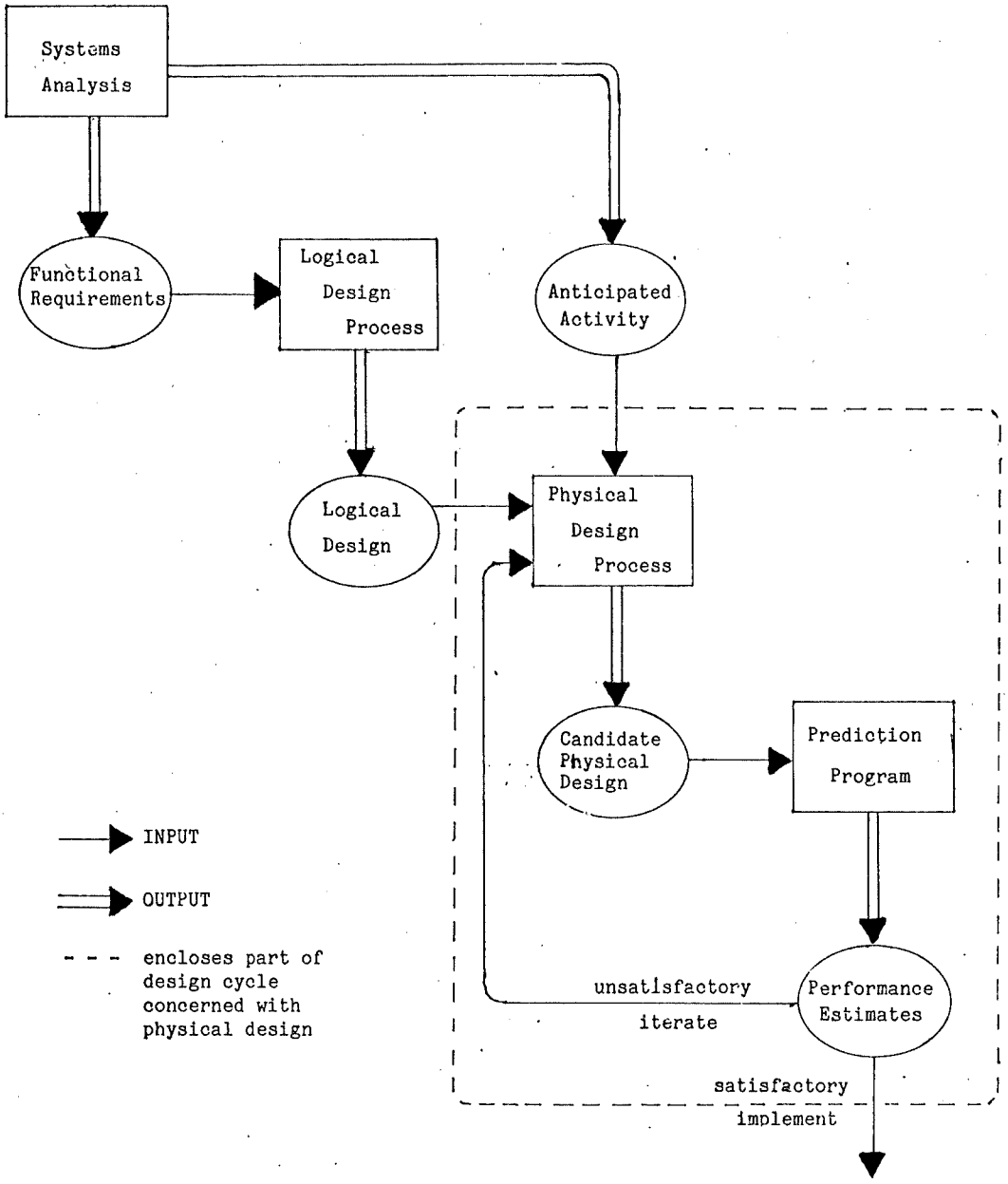


Figure 1.1: An Idealised Database Design Cycle

such a prediction program in an idealised database design cycle. The section of the design cycle concerned with physical database design and performance prediction is enclosed by a broken line. The role of the prediction program is to evaluate candidate physical designs produced by some physical design process. That process might be a specific design methodology or simply the work of an experienced practitioner. The

general model provides the framework for describing both the candidate physical designs and the logical design for which a suitable realisation is being sought.

1.2 Basic concepts and terminology

This thesis assumes that the reader is familiar with basic file organisation techniques and the three main approaches to DBMS* architecture (relational, hierarchical and network). A suitable introduction to file organisations can be found in [Dodd69]; a good introductory treatment of DBMSs is provided by the *Computing Surveys* special issue on that topic (vol. 8/1, 1976).

Some basic terminology will now be introduced. These terms and all others defined in subsequent chapters are listed in a glossary which follows the last chapter. The definition of the first three terms follows that of Dodd [Dodd69].

A piece of raw information stored, retrieved or otherwise processed by a DBS is an *elementary data item*. This will usually be shortened to *data item* or *field*. A collection of one or more data items accessed together is called a *record*. Data items may be grouped into records to indicate logical relationships or for performance considerations. A *key* is a group of data items from the same record used to represent the record in some way (for searching, indexing, sorting etc.). A *primary key* is one which uniquely identifies the full record, otherwise it is a *secondary key*. A *file* is a collection of records, usually of the same type. *Access* is used throughout this dissertation as a generic term for the

* DBMS stands for Data Base Management System.

the "touching" of data, whether for the purpose of retrieval or update.

An integrated DBS is built around a database which generally consists of a number of files. An *access path* is some mechanism for accessing a set of records which requires less effort than the scanning of all records in the database. Access paths are of two types [Nijs74], associative and positional. An *associative access path* leads to a record (or group of records) identified by some key value. A *positional access path* leads to a record which is in some sense "next" relative to a record of known location. A *file organisation* is the combination of structures and algorithms used to provide an access path. Some examples of file organisations are a file of sorted records accessed by binary search, a file of records grouped into secondary storage buckets addressed by a hashing function, or an inverted file.

1.3 Overview of thesis

The remainder of the thesis comprises seven chapters and two appendices. The next chapter, Chapter 2, overviews the work of other researchers which has served as a point of departure for or has influenced the development of this research. It concludes with a summary of directly related work.

Chapter 3 introduces the problem of modelling a DBS and then develops some ideas about the nature of databases and an analysis of file organisation methods which have influenced the DBS model. Chapters 4 through 6 go on to explain the model in detail with the aid of an example.

Chapter 7 describes the prediction program built around the model

and presents some preliminary results obtained from experimentation with an operational hospital patient administration system. Chapter 8 discusses further experimentation possible with the prediction program as it stands, suggests directions for further research to remove some of the present limitations of the model and program, and makes some observations based on the author's experience in developing them.

Appendix A describes the hospital DBS using the general DBS model. Appendix B is a complete BNF grammar defining the syntax of a language in which to write DBS descriptions accepted by the prediction program.

2. RELATED RESEARCH ON DBSs

A recent survey [Senk77] has identified two divergent streams of research in the area of DBSs. The first, which might be termed DBS theory, deals with data models, database accessing languages and data dictionaries. The second, which could be called system performance research, is concerned with the description of system workloads, the design, evaluation and optimisation of file organisations and the evaluation of DBSs constructed using file organisation techniques. The author's research has been influenced by results from both streams. Some of the research on DBS theory was used as a starting point for the development of a general model of DBSs. Other researchers' work on the evaluation of file organisations has been used to guide that development and in the writing of an experimental prediction program. This chapter surveys the related work from both streams with particular attention devoted to previous efforts at DBS evaluation.

2.1 Contributions from DBS theory

The directly relevant research from the stream of DBS theory concerns concepts from the organisation and architecture of DBSs. The last ten years have seen a number of proposals for viewing a DBS in terms of levels of abstraction.

Application of this approach in the late 1960s led to the formulation of two levels of abstraction known as logical and physical. At the logical level one is interested in establishing a correct and complete representation of the real world system to be modelled by the database. Correctness and completeness are judged in the context of anticipated database usage. At the physical level attention shifts

to the performance and efficiency issues of maintaining and manipulating the database representation of the real world. The motivation behind the logical-physical separation is to allow the designer to solve the logical aspects of his database problem first and then consider separately the selection of a physical storage structure that provides satisfactory performance.

Several years later, Earley proposed a tri-level definition of data structures [Earl72]. His three levels were the *relational level* at which only the relationships between data items are specified, the *access path level* which describes how data can be accessed and the *implementation level* which defines how a data structure is to be realised in a machine or on a storage medium. Earley thought of the relational and access path levels as being sub-levels of what he called the *semantic level* which defines, in a representation-independent way, the data in a data structure, how it can be accessed and in what ways the structure can be altered.

A major advance was then made by Senko and some other researchers at IBM San Jose with the introduction of the Data Independent Accessing Model (DIAM)[Senk73]. DIAM defined four levels of abstraction:

1. the Entity Set Level
   -- to provide a structured model of the real-world information in the database. This model is independent of all implementation details.

2. the String (Access Path) Level
   -- to describe access paths through the data constructed for reasons of efficiency

3. the Encoding Level
   -- to describe the bit-level encoding of access paths

4. the Physical Device Level
   -- to define the placement of the bit-level encodings on physical devices

The DIAM entity set level corresponds to the logical level of abstraction. The other three DIAM levels are sub-levels of the physical level of abstraction. Although originally conceived as a general descriptive model of database structure, DIAM evolved to become a DBMS architecture complete with two accessing languages, RIAL and RDAL. DIAM is further described in [Senk72, Senk76].

In an interesting theoretical paper Sundgren [Sund74] identified the function of a database as modelling some object system. He went on to make a useful distinction between:

a) the object system
b) information about the object system
c) data representations of that information
d) the storage media on which those representations are recorded

Database design can then be characterised as the choosing of three mappings: a)->b), b)->c) and c)->d). Logical database design defines the first mapping a)->b); mappings b)->c) and c)->d) are the province of physical database design. DIAM provides a good treatment of b)->c) and c)->d). A DBS model oriented towards performance prediction must concentrate on the physical level of abstraction. Hence DIAM was chosen as the starting point for the development of the general model of DBSs described in chapters 3 through 6.

2.2 Contributions from file organisation evaluation

Initial work in this area concentrated on the evaluation of one file organisation at a time. Later, as the underlying principles became better understood, attention shifted to formulating general models of file organisations and using them to develop generalised

equations for file organisation evaluation.

2.2.1 Evaluation of single file organisations

As early as 1957, Petersen used empirical methods in an exhaustive study of hashing with linear probing [Pete57]. In 1971, Lum et. al. published the results of a major study of hashing algorithms based on extensive experimentation with live rather than artificially generated data [Lum71]. Knuth [Knut73] provides analytical formulae for estimating the expected number of secondary storage accesses incurred in accessing records on direct-access storage devices by several different hashing techniques.

An early paper by Bloom [Bloo69] presented an analytical evaluation of a large inverted file organisation. The FOREM model developed by a group at IBM [Senk68] calculated timings for file organisations built around IBM indexing and direct-access methods. In 1970, Collmeyer and Shemer [Coll70] published analytical formulae for the retrieval performance of file organisations providing access on a primary key through binary search, indexing or hashing. Cardenas has developed detailed equations for predicting the retrieval performance of inverted files [Card75a] and doubly-chained trees [Card75b]. Siler has covered similar ground using stochastic and Monte Carlo techniques [Sile76]. A common factor in all this work is that it deals with retrieval only and does not consider CPU costs. The rationale for neglecting CPU costs is that they are assumed to be several orders of magnitude smaller than secondary storage access times and can be ignored for practical purposes. Another point in common is that a different set of equations or programmed evaluation module must be used for each type of file organisation.

2.2.2 Generalised file organisation evaluation

In order to develop general equations for file organisation evaluation it was first necessary to develop general models. An influential early paper in this regard was that of Hsiao and Harary [Hsia70] who proposed a list-oriented classification of file organisations. A file organisation is viewed as partitioning a file into lists of records with the same key value. To this end there is a directory containing each distinct key value and a list of the heads of sublists of records with the key value. The parameters describing a file organisation are the average number of records having a particular key value and the average number of sublists per value. For example, a primary index has one sublist of length one for every primary key value, an inverted file has k sublists of length one for every secondary key value (where k is the average number of records per secondary key value) and a multilist has one sublist of length k for every secondary key value (k is again the average number of records for a secondary key value). The major limitation of Hsiao and Harary's model is that it cannot differentiate between lists implemented by physical contiguity and those implemented by pointers. In fact, the use of pointers to link list elements is implicitly assumed. Consequently, some important file organisations such as sequential and index sequential cannot be described.

A second influential model in this area is due to Severance [Seve72, Seve75]. Severance proposed that file organisations be regarded as lists of record nodes. Each record node has two connections, one linking it to its list successor and another linking it to its data items. In each case the connections may be represented by physical contiguity or by a pointer. The parameters of the model are the

proportion of direct (contiguous) connections to indirect (pointer) ones for both the inter-record (record successor) and intra-record (data location) connections. In a sequential file all inter-record and intra-record connections are direct, in one of the inverted lists of an inverted file all inter-record connections are direct and intra-record ones indirect, and in one of the lists for a particular key value in the multilist organisation all inter-record connections are indirect and intra-record ones direct. An indexed sequential file is viewed as falling somewhere between a sequential file and a list in that some inter-record connections are direct and others indirect, the exact proportion depending on the amount of overflow. The principal limitation of Severance's model is that it lacks something equivalent to the notion of a directory which enabled Hsiao and Harary to combine lists for each key value into a complete file structure.

Building on the work of both Hsiao & Harary and Severance, Yao [Yao77a] has recently published details of a general model which describes file organisations in terms of three basic parameters: the number of index levels, the expected search length at each index level and at the level of the data records, and the proportion of direct to indirect connections at each level. Yao's model can describe a wide range of file organisations including multi-level primary and secondary indexing, multilists, cellular lists, index sequential and B-trees. Generalised equations for estimating secondary storage access times for random retrievals and updates have been developed as well. This work represents a significant advance since all these organisations can be described and evaluated in terms of the one model. However, evaluation is still limited to a single file organisation at a time and CPU costs are not taken into account. Teory and Das [Teor76]

have built a program called the File Design Analyzer which is based on Yao's model with some extensions to take in sequential and batch processing.

The culmination of work in the area of generalised file organisation evaluation is research leading to procedures for the design of optimal file organisations. Kennedy [Kenn73] has studied the use of access frequency information to optimise file organisation performance. Severance [Seve72] developed the parametric model of file organisations referred to earlier and presented cost equations using those model parameters and a design algorithm to determine "good" solutions for a restricted class of problems. Yao [Yao74] used an earlier version of his general model described above as a basis for a restricted optimisation procedure. Duhne [Duhn77] extended Severance's models and equations to include a wider class of implementation alternatives and then developed a semi-automatic design procedure based on branch and bound techniques.

2.2.3 Estimation of expected number of record or block accesses

Another relevant area of file organisation evaluation treats statistical questions involving batch accessing of files. Usually the quantity of interest is the expected number of record or block accesses. Papers in this category include those of Pezarro [Peza76] and Yao [Yao77b]which deal with the problem of estimating the number of blocks accessed in retrieving k records of known device address from a file, one by Schneiderman and Goodman [Schm76]which provides estimates of the savings in record accesses gained from the batching of queries to sequential or tree structured files, and one by Kollias [Koll78] which supplies an estimate for the proportion of a file which must be accessed in processing a batch of records ordered by device address.

2.3 Performance prediction for DBSs

All the work on performance prediction mentioned in the previous section shares a common characteristic -- the evaluation of file organisation techniques as applied to a single file only. Performance prediction for DBSs presents a considerably more difficult problem. An integrated database may be thought of as a number of files inter-related by a lattice of logical and physical connections. The retrieving or updating of data may necessitate travelling a path leading across several files and following an arbitrary combination of connections. It is dealing with this possibility which makes the task of performance evaluation for a DBS much more difficult. However, results from the study of file organisation evaluation can serve as a foundation for the more complex systems which predict DBS performance.

Prediction systems for DBSs can be divided into two classes: analytic and event-driven simulation. Analytic systems are usually predicated on the assumption of single-threaded accessing of the database. This simplification makes it possible to estimate performance with equations, resulting in a prediction system that runs much faster than real time and is relatively inexpensive to use. Event-driven simulation systems, on the other hand, can be designed to model the complex effects of concurrent access, channel contention and operating system overheads arising from multiprogramming. Consequently, they provide much greater accuracy at the price of a much higher cost in running time. Event simulation systems also require considerably

more detailed input.

Six DBS prediction systems that have been reported in the literature will now be looked at briefly. Besides the analytic/simulation classification, they also can be classified as *DBMS-specific* or *DBMS-independent*. DBMS-specific is taken to mean applicable to DBSs implemented with a specific DBMS whereas a DBMS-independent prediction system covers a range of DBMSs. The DBMS-specific ones will be reviewed first.

Several DBMS-specific prediction systems have been developed by the vendor of the DBMS concerned. An example is IBM's DBPROTOTYPE [IBM74]. DBPROTOTYPE is a package of six programs that permit detailed simulation of DL/1 calls against a skeleton IMS database. One of the programs builds the skeleton database which is specified by listing all root and child segments and providing a statistical description of segment occurrence frequencies. The others generate DL/1 calls against the skeleton database and report the results. DBPROTOTYPE as described in [IBM74] does not model concurrent access.

Other DBMS-specific prediction systems have been proposed or developed by large users. One of these is a system under development by Shell-Holland for evaluation of Univac's DMS-1100. As described by de Beer and Smit [deBe76], the Shell-Holland system accepts a DDL schema as input and produces internal data structures from which storage requirements are computed. De Beer and Smit also outline some formulae for estimating the number of block accesses to transfer a random selection of records to or from secondary storage. They do not propose to deal with concurrent access. Database usage is to be modelled at the level of DBTG DML commands, for example "FIND a record occurrence

using its CALC key".

An analytic DBMS-specific system which has actually been built is described by Teory and Oberlander [Teor78]. Called the Database Design Evaluator (DBDE), it produces expected-value performance estimates for applications running under Honeywell's IDS [Hone71]. Input to the DBDE includes a description of the IDS database, of the hardware (limited to one type of secondary storage device) and of the application as a sequence of IDS DML operations (RETRIEVE, STORE etc.). Principal output is an estimate of I/O processing time made up of seek time, rotational delay and data transfer time, and CPU time which includes database search time, I/O initiation and termination overheads and moving data between IDS buffers and the user work area. The DBDE seeks to provide bounds on I/O processing time in a multi-threading environment by suppling two estimates for this figure: one for "single access", defined as a dedicated device with no seek time and average rotational delays between consecutive sequential accesses, and "multi-access", modelled as a shared device with all sequential accesses for an individual application becoming effectively random because of interference from other processes. Other features of the DBDE are computation of overflow probabilities on the basis of database growth and the distribution of record sizes, evaluation of the effect of multiple buffering on I/O processing time* and modelling of dependent sequences of IDS operations involving currency. No figures are given for the DBDE's size or average running time. One validation test is reported using a live database of 155,000 records of three different types. A comparison was made between estimates of total elapsed time+ for a batch of retrieval

* This is not done analytically but by a kind of limited simulation.

\+ obtained from a central server queuing model of the test system using the DBDE's estimates of application CPU and I/O times.

applications and measured elapsed time. On this basis the DBDE estimate differed from the observed one by 21%.

A final example of DBMS-specific systems is the one developed by the MITRE corporation for NASA and described by Spitzer [Spit76]. It was built around an already existing simulation system for Exec-8 on the Univac 1108 to evaluate the performance of System 2000 in that machine/operating system environment. The 1108 simulator is driven by a workload of tasks characterised in terms of CPU time, I/O transfers and primary store requirements. The approach followed was to select eight primitive database operations out of which to construct sequences that represented System 2000 transactions. Each primitive database operation was described in terms of five parameters that had been identified by a separate study as being the most significant in determining System 2000 performance. A series of experimental runs was made, varying the five database parameters to produce graphs of CPU time and I/O transfers as a function of the database parameters. Using these graphs, a sequence of database operations representing a DBS transaction could be converted into a sequence of simulator tasks. Output from the simulator then gave total response time broken down into time waiting for processing by the DBMS, time waiting for primary store, swapping time, time spent in primary store (waiting to execute and executing) and I/O wait time. The MITRE system does model concurrent access and the interference effects arising from database locking to synchronise such access.

Two DBMS-independent prediction systems have recently been reported in the literature. The first is described by Nakurama, Yoshida and Kondo of Hitachi [Naka75]. It is a comprehensive simulation system with a two-part organisation. One part consists of a textual description of the hardware and database being modelled. The other is a procedural

modelling of the operating system, DBMS and application programs that together comprise the software driving the DBS. Task scheduling, I/O interrupt processing, communication processing, message scheduling, database access processing, buffer management and disc space management are all simulated. Detailed reports on all these activities are available as output from the simulator. The limitations of this system are that it can only describe hierarchical database structures and that a large reprogramming effort would be involved in applying it to a different host system (or even another set of applications) because of its use of procedural modelling. The latter limitation appears to be an inevitable consequence of an effort to obtain a very high degree of accuracy. No precise running times were given,but as a rough guide the authors reported that the simulator runs about 10 to 20 times faster than real-time.

The last system to be reviewed was developed by the Martin Marietta Group for NASA [Schn75]. It is a DBMS-independent simulator based on an implementation of the DIAM model mentioned earlier. Input to the simulator consists of a database description in terms of the DIAM model and a set of queries expressed in the DIAM RIAL accessing language. The storage devices and operating system were modelled procedurally as far as was thought necessary to calculate response time and resource usage. NASA decided not to use the simulator after extensive evaluation [NASA76] because it was judged very difficult to use and because the response time predictions were not accurate enough. However, it was felt that most of the deficiencies could be corrected at the cost of a major reprogramming effort.

Table 2.1 shows a categorisation of the six prediction systems based on the two classifications of analytic/simulation and DBMS-

specific/DBMS-independent. The position of SEER, the experimental prediction program developed as part of the author's research, is also shown within the scheme of categorisation. As Table 2.1 shows, it is

	DBMS-specific	DBMS-independent
analytic	Shell-Holland DBDE	SEER
event-driven simulation	DBPROTOTYPE MITRE	Hitachi* Martin Marietta

* hierarchical database structures only

Table 2.1: Categorisation of DBS Prediction Systems

the only analytic, DBMS-independent prediction system of which the author is aware.

3. MODELLING A DBS

The problem to be addressed is how does one describe a DBS for the purposes of performance prediction. The first decision to be made is what measure of system performance should be used. Following that decision, the next step is to identify the aspects of a DBS that must be described to estimate performance in terms of the selected metric. The first three sections of this chapter expand on these points to present an overall picture of the descriptive model developed by the author. The last three sections introduce some general ideas about the nature of databases, an analysis of file organisation methods and an example of a simple database. These are used in chapters 4 to 6 which explain the descriptive model in detail.

3.1 A measure of DBS performance

If a DBS is thought of as a repository of facts of interest to the organisation using it, then there are two principal activities which it must support. These are the *retrieval* of facts stored in the database and the *updating* of facts to keep them in line with external reality. Updating can be divided into three distinct activities: *modification*, *insertion* and *deletion*. A request to the DBS to perform one of these four activities is termed an *operation request*. Thus the DBS is viewed as a kind of "black box" system which stores facts, retrieves them later, can be asked to modify or delete them, or add new ones. This way of looking at things is depicted in Figure 3.1.

A reasonable goal is to aim at producing first-order or expected-value estimates of the time taken by a DBS to perform an operation request. This will be known as *database access time*. It should be

emphasized that this is not what is known as response time but rather is a component of response time. Response time in an online, interactive system typically consists of three basic components: teleprocessing and communication delays, multi-user overheads (O/S overheads and I/O contention) and time spent in accessing the database itself. The work outlined in this dissertation limits itself to the last of these.

Figure 3.1 suggests an application program communicating requests to a DBS either as a batch program or as a result of interaction with a user. Performance of an operation request is more precisely defined

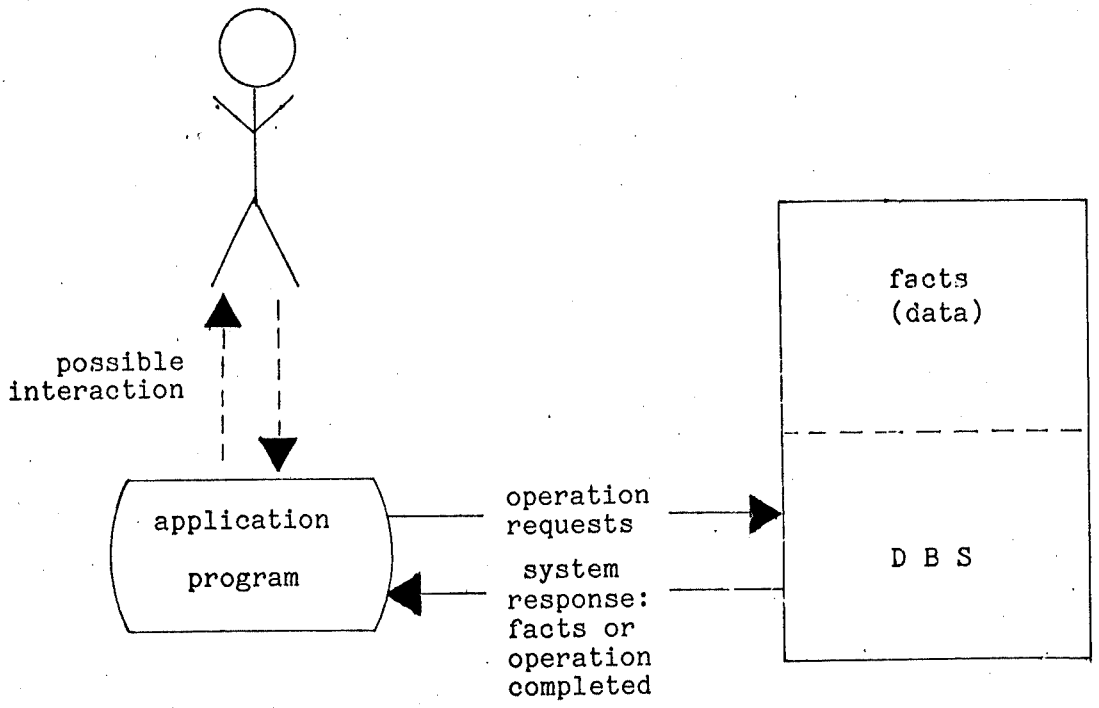


Figure 3.1: DBS as "black box" System

according to this view of matters as follows. In the case of retrieval, it is the location of all specified facts (if any) and their return to

the address space of the application program. In the case of update, it is the copying of all supplied facts (if any) to the address space of the DBS and execution of the requested operation. Thus database access time has two components: *secondary storage accessing time* and *CPU processing time*.

Two restrictions are imposed as simplifying assumptions. In both instances this is because there is no way of analytically treating the complications that arise in the unconstrained general case within the limitations of an expected-value model. The first restriction is that only single-threaded accessing of a database will be modelled. Proper treatment of multi-threading does not appear possible without the use of queuing or simulation models, a conclusion also reached by other researchers [Teor76]. The second restriction is that the model does not attempt to account for any reduction in secondary storage accesses that might be gained by the use of multiple page buffers. Trying to model this effect analytically involves making assumptions about the access patterns of application programs and it is difficult to see what assumptions would be reasonable. Extensive experimentation akin to that reported by Saltzer [Salt74] and Greenberg [Gree74] in postulating a linear relation between mean headway between page faults and the size of paging memory would be required. The author's experience in implementing a prediction program has led to some ideas about relaxing these restrictions. They are discussed in chapter 8.

In the context of the simplifying assumptions described above, secondary storage accessing time is taken to mean the sum of the time required to position a device at the starting point for a transfer and the time to transfer data from/to DBS buffers. Estimates of CPU processing time are intended to model overheads common to any DBMS,

such as the time to move data between DBS and application program buffers and the time required to compare index keys or evaluate hashing functions. As suggested in Figure 3.1, no estimate is made of the CPU time used by the application program because it is a function of program behaviour which is difficult to quantify analytically.

Database access time has now been selected as the measure of DBS performance. The other important quantity to be estimated is the total storage requirement of a described system.

3.2 Aspects of DBS description

Having chosen database access time and total storage requirements as the quantities to be estimated, it is now possible to identify the aspects of a DBS that must be described. These are:

1. database structure
   -- the type of information stored in the database and how it is represented on the available storage media.

2. database content
   -- the volume of data items and distribution of data values.

3. hardware configuration
   -- the performance and capacity of the available storage devices and limited details of CPU performance.

4. DBS software
   -- the CPU cost of executing certain basic DBS functions.

5. database activity
   -- a representative set of operation requests.

Prediction of database access time entails evaluating the cost of executing operation requests. Execution of an operation request is thought of as navigating a route through the database leading from some known starting point to the records of interest. Once these target records have been located, they may be retrieved, modified, deleted or

have new records inserted before (or after) them. With this model the relevance of each of the five aspects mentioned above can easily be explained.

A description of database structure is, in a sense, a map of the database. It enables a route to be selected or, equivalently, defines how to go about answering a query or making an update. This means providing sufficient information to decide 1) if the operation request is valid in the sense of being directed at facts of a type known to be in the database, 2) which records are of interest and how to get to them and 3) how to interpret their contents.

Given a route to the records of interest, the next step is to estimate how many records have to be visited in travelling it and to sum the expected costs of their accessing and processing. The result will be an estimate of database access time. Estimates of the number of records visited are made on the basis of information about database content. The cost of record accessing and processing is predicted from the details about the hardware configuration and DBS software.

Specification of the first four aspects thus provides enough information to evaluate a single operation request. For evaluating a DBS as a whole, a representative set of operation requests for the entire user community is needed. A description of database activity supplies such a set of requests which is similar in concept to the instruction-mix often used in the evaluation of CPU performance. This analogy is especially appropriate when one considers that the instruction set of a "database machine" would almost certainly include the four fundamental operations of retrieval, modification, insertion and deletion [Cana74, Ozka75, Su75].

Storage requirements can be estimated from the specification of the number of bits used to store individual values and information about the number of values stored.

Figure 3.2 shows the structure of the general descriptive model which has been proposed. It has five components, one for each aspect to be described. The following sub-sections discuss these components

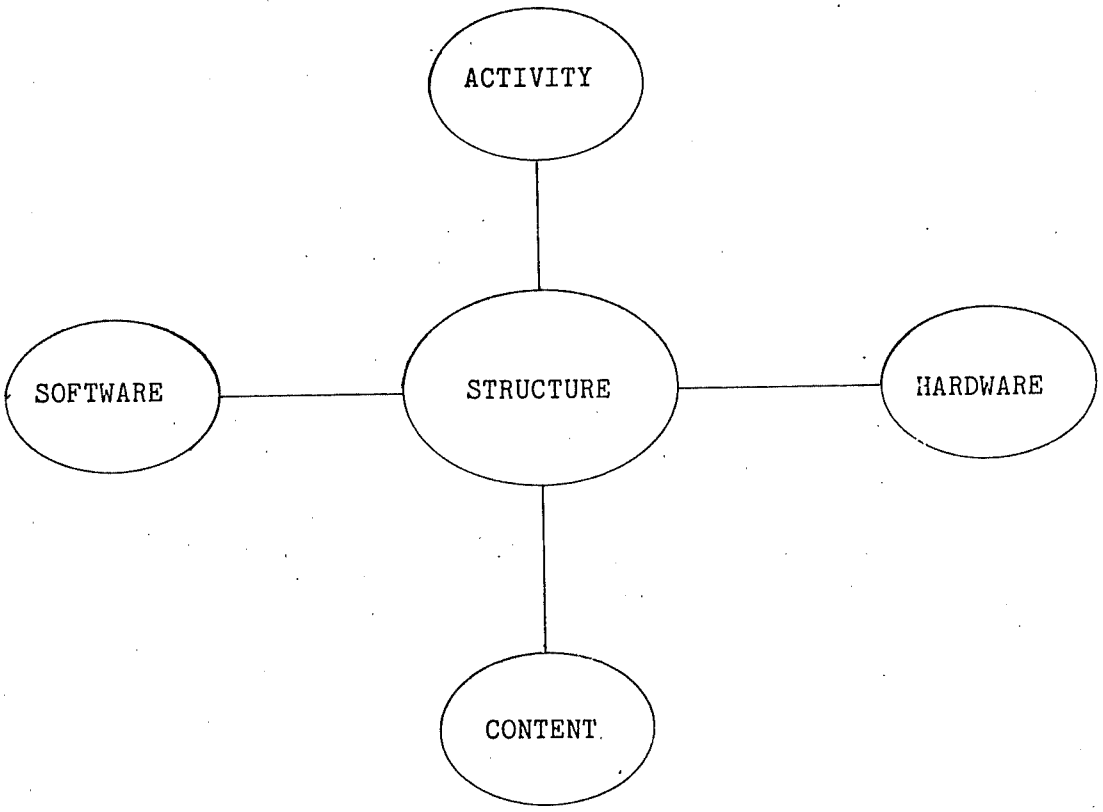


Figure 3.2: Components of a DBS Descriptive Model

in turn.

3.2.1. Structural description

Database structure was previously likened to a map of a database providing three kinds of information. These were: 1) an indication

of what types of facts are in the database, 2) a specification of which records hold which facts and of the access paths leading to records, and 3) a definition of the bit-level representation of facts. The above decomposition suggests structure can be described in terms of three hierarchic levels. These are an *information level* corresponding to 1), an *access level* corresponding to 2) and an *encoding level* corresponding to 3).

The information level is the conceptual level of database description. It specifies which entities are modelled by a database and which of their properties are represented in it. This is known as the *information structure* of a database. Information structure defines the types of fact in a database. *Information content* refers to the actual collection of facts in a database at some instant in time (they will be of type defined by the information structure).

The access level describes the *access structure* of a database. Access structure refers to the network of logical paths in a database which lead from known starting points to individual data values. A description of access structure declares such logical paths and specifies which data values represent which entity properties and the mechanisms for accessing those data values. The term *logical path* is used to emphasize that the access level is concerned with which logical connections exist and not how those connections are represented.

Data values are thought of as being stored in one or more virtual address spaces. In turn, the virtual address spaces are physically located on the devices described by the hardware configuration component of a DBS description. The encoding level specifies how many bits are allocated to store data values and the representation of logical

connections (direct by contiguity or indirect by pointer, for example).

Figure 3.3 adapted from [Yao74] is a simple example of the

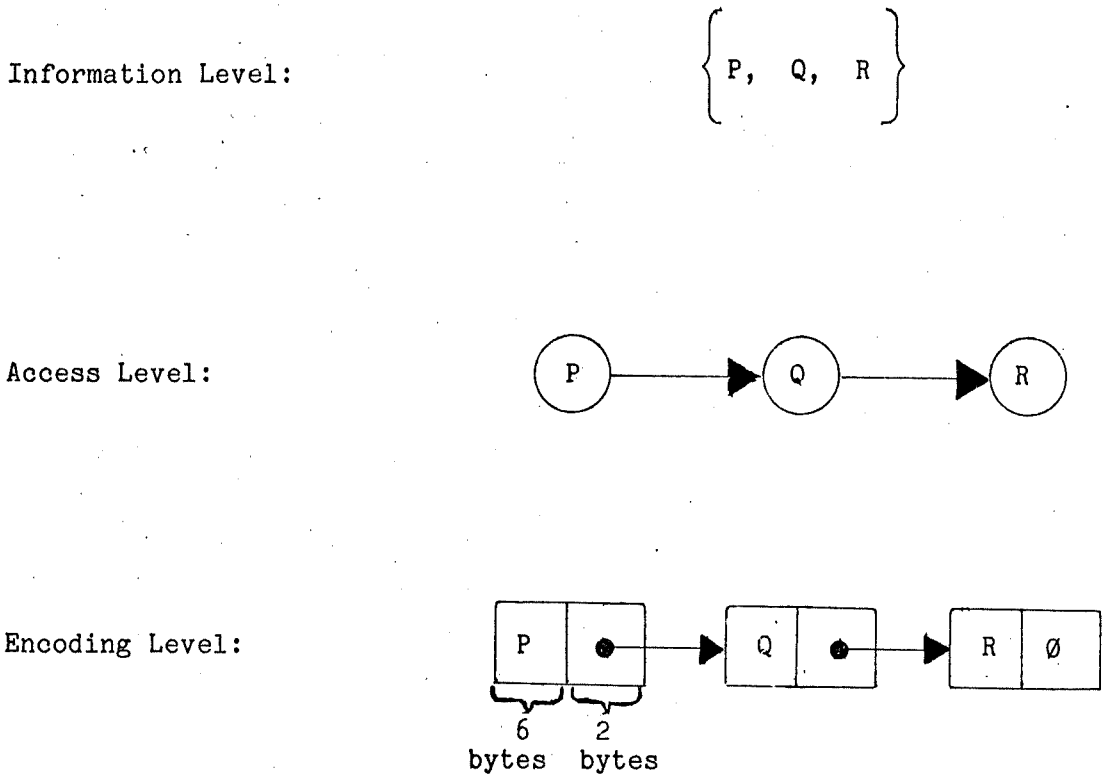


Figure 3.3: Levels of Structural Description

different levels of structural description. Imagine that there is a collection of three properties of some entity. At the information level these are seen as a set, {P,Q,R}, in which no ordering is implied. One possible path at the access level is P->Q->R. At the encoding level that access path could be represented by a linked list of elements with six bytes to hold a property value and two bytes to point to the next element. Equally well, the elements could be stored contiguously, in which case pointers would not be required.

Figure 3.4 sums up the above approach to structural description.

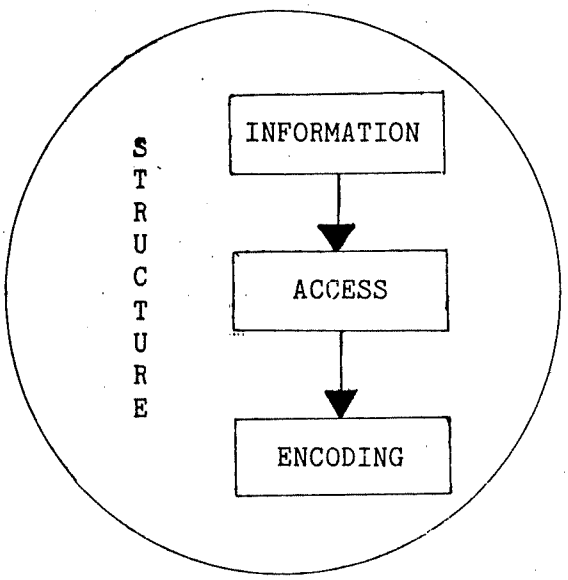


Figure 3.4: Tri-level Structural Description

Structure is described by one component of the DBS model. That component is split into three hierarchical levels, each an expansion of its parent providing an increasingly detailed specification of database structure.

Each level represents a freezing of a certain class of design parameters. The information level reflects a selection from all the entities and their properties in the real world, the access level a choice as to what access paths to provide to the values of entity properties and the encoding level a decision as to the number of bits to allocate to the storage of individual data values and the representation of logical connections. Thus for a given information level there are a number of possible access levels and, similarly, each one of these could be encoded in several different ways.

3.2.2 Content description

Database content is described separately from structure because the amount of data in a database usually varies as additions or

deletions are made to reflect changes in the real world. An exact specification of database content describes a database frozen at some instant in time. For the purposes of first-order performance prediction it is more useful to have a rough statistical description which is a reasonable approximation of database content over some time interval. The length of this interval is primarily a function of database volatility. In the case of a database which is expected to change significantly in size, it is advisable to provide content specifications corresponding to several stages in its growth. By this strategy one can determine when a planned access structure is likely to become inadequate and a new one will have to be devised.

A more concrete specification of the sort of content information required may be given in terms of the concepts of record and field. One needs to know such things as the number of occurrences of a particular type of record and the range of possible values for a particular field of a record. As far as the model described in the next three chapters is concerned, content description specifies how many instances of the various structural elements of the model are expected.

3.2.3 Hardware description and the SDM

Hardware description characterises the storage devices of the DBS in terms of their performance and capacity. Device performance and capacity suffice for the prediction of secondary storage access times and a check that total data volume is within total device capacity. There is one device description per type of device. Specification of the secondary storage configuration does not extend to identifying individual devices because that would be unnecessarily detailed for the purposes of this model. Limited details of CPU performance are

included for use in estimating CPU overheads.

In the sub-section on structural description it was mentioned that data values were thought of as being encoded in virtual address spaces which were themselves stored on physical devices. The assignment of virtual address spaces to devices depends on both database structure and hardware configuration. Therefore, there is a sixth component of a DBS description, not previously mentioned, to define this assignment. It is called the *SDM* (Structure-Device Mapping).

3.2.4 Software description

The main reason for making an attempt at software description is as an aid in estimating CPU processing costs. The minimal requirement is a means of specifying average CPU time for the execution of various primitive functions (hashing for example), the cost of which cannot be solely estimated from the simple CPU description mentioned above. Another reason is to provide a limited escape mechanism for indicating the CPU cost of algorithms or procedures other than the standard ones assumed by the prediction software.

3.2.5 Activity description

The components of the model described so far are sufficient to estimate DBS performance for a single access request. However, as was mentioned in chapter 1, the performance of an integrated DBS must be globally optimised taking into account the needs (and their relative importance) of all users. Hence a mechanism for generating a representative set of operation requests is needed to serve as a characterisation of database usage.

Operation requests can generally be classed as either *routine* or *non-routine*. Routine requests are those which occur frequently. They are exemplified by those generated by transaction processing, batch database applications or the processing of stored relational views as in System R [Astr76]. Routine requests define frequently used access patterns which can be accurately identified and profitably supported by file organisation techniques.

Non-routine operation requests are typically generated by high-level query language processors. The facts to be accessed are specified by arbitrary Boolean selection expressions which can range over any entity property represented in the database. Consequently, it is very difficult to identify access patterns which will be heavily enough used to justify the expense of building and maintaining access paths to support them. The most promising approach for selecting cost-effective access structures for DBSs in which non-routine requests predominate seems to lie in the direction of learning systems such as Stocker and Dearnley's self-organising DBMS [Stoc74] and Hammer and Chan's prototype system for automatic secondary index selection [Hamm76].

For the above reasons the model adopted here restricts itself to a simple strategy for generating routine operation requests. A list of pre-determined operation requests with associated usage weightings is supplied. The prediction program then estimates the average database access time for each operation request and also combines the results using the usage weightings to produce an estimate of average database access time across all database activity.

3.3 PRODD

Considering the inter-relationships of all six description components yields a diagram like Figure 3.5. The next three chapters describe a scheme for specifying the six components and their inter-relationships through a model called PRODD (for *Pr*ediction *O*riented *D*atabase *D*escription). The placing of structure in the middle is

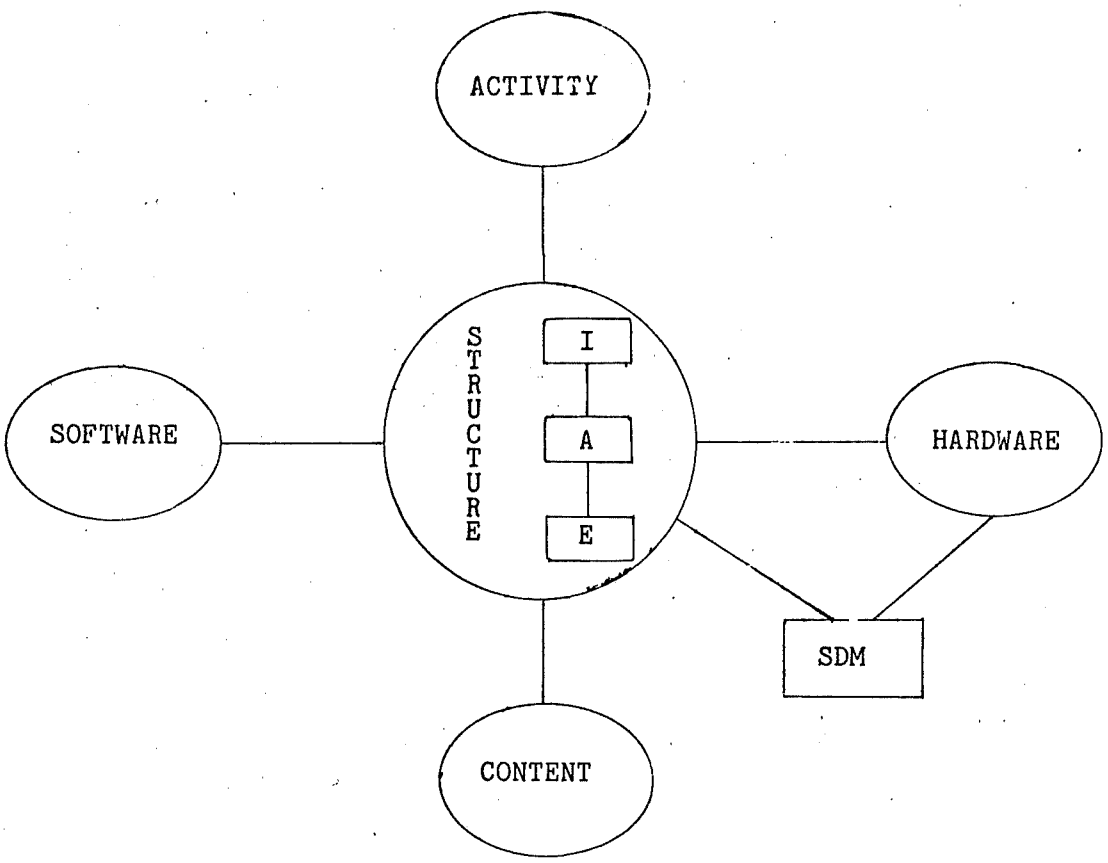


Figure 3.5: Components of PRODD

deliberate. It reflects the central role it plays in defining the procedures for database accessing. Once an access route has been determined, the three components of software, hardware and content description plus the SDM component supply the details needed to estimate the cost of travelling it. The activity component sits on top and directs the overall evaluation process by indicating which access

routes are to be costed.

3.4 What is a database?

In developing a general descriptive model of DBSs some assumptions about the nature and purposes of databases must be made. Mealy [Meal67] has observed that there are three realms of interest in data processing, namely, the real world itself, ideas about it existing in people's minds and symbols representing these ideas recorded on paper or some other medium. Following this classification, a database is a repository for recording symbols representing ideas about the real world. Logical database design is concerned with selecting which ideas about the real world to include in the database. Physical database design deals with choosing an appropriate symbolic representation of those ideas on the available storage media.

These ideas can be made more precise by introducing the concept of *entities* which are objects of interest in the real world. They may be people, places, things or relationships. Entities have properties which describe them. A *property* is a correspondence between an entity and some value describing it. The properties of an entity chosen for inclusion in a database are grouped into one or more entity descriptions. A database now becomes a collection of the representations of entity descriptions.

One might then consider how these representations of entity descriptions are going to be accessed, in particular how they are going to be addressed or located. It is here asserted that a fundamental characteristic of databases is that data is addressed on the basis of content, at least from the conceptual point of view. Any deviations

from this principle can be explained as performance optimisations of one kind or other. This principle leads to a simple conceptual model of database accessing. If new entity descriptions, i.e. new data, are to be added they are presented to the DBS to place where it sees fit. Existing entity descriptions, i.e. existing data, are addressed by supplying values for one or more of their properties. Any descriptions (possibly none) matching the content criterion will then be processed according to the operation specified (retrieval, modification or deletion). It is the responsibility of the DBS to find the stored representations of the target entity descriptions by whatever means are available.

One algorithm which can always be used is sequential search of the entire database, that is all entity descriptions. Although satisfactory from a theoretical standpoint, sequential search is too slow for most practical purposes. Consequently, a number of file organisations have been developed to speed up content addressing. The basic principle underlying all of these is a reduction of the search space in one way or another, a principle neatly expressed in Senko et. al's definition of a file organisation as "... the preprocessing and prestructuring of information so as to reduce the amount of scanning for the kinds of transactions that are anticipated" [Senk73].

3.5 Analysis of file organisation methods

Reduction of the size of the search space has just been identified as the basic principle underlying all file organisations. Why this should be so is easily seen by considering the formula for search time:

$T = V/R$

where T is search time

V is search space volume (number of bits to be scanned)

R is rate at which the search space can be scanned.

Clearly search time (T) can be made smaller either by reducing the number of bits to be scanned (V) or increasing the rate of scanning (R). R is determined by hardware factors. Current efforts in the direction of increasing R are centred around associative hardware. V can be reduced by software techniques and this is where file organisations come in.

Software methods for reducing V fall into two categories:

1. reduction of the number of records in the search space that must be examined
2. reduction of the size of records in the search space

In all cases the objective is to reduce the number of bits that must be scanned. Methods in category 1 are often used by themselves; those in category 2 are generally used in conjunction with a method from the first category.

3.5.1 Reduction of number of records scanned

Methods in category 1 will be examined first. The three principal ones discussed here are:

- partitioning
- restriction
- ordering

Partitioning divides a collection of records into equivalence classes on the basis of a function of a record key. All records with the same function value are in the same equivalence class and there are as many equivalence classes as there are unique function values. A familiar example of partitioning is provided by hashing. All records hashing to the same bucket or table entry are in the same equivalence class and it is only necessary to examine them when searching on a given key value.

There is an important special case of partitioning where the partitioning function is Boolean and members of only one of the two equivalence classes are of interest. Therefore, an explicit access path is defined only for the members of the collection corresponding to that equivalence class. This is known as *restriction*. It limits the search space to a subset of the orginal collection of records with the Boolean function acting as the membership predicate for the subset. Subsequent searching among records known to match that criterion can be satisfied by considering only that subset. Restriction may be profitable when it is known that there will be a lot of activity directed to a certain subset of the file with very little to the remaining records. The expense of maintaining the access path for frequently accessed records is then justified and none is incurred for the infrequently accessed ones. A simple example would be keeping a list of orders for the current week (day) in a file of pending orders.

Ordering is the arranging of a collection of records in some particular order, usually defined on the basis of their key values. The advantage of ordering on key value is that sequential search on the ordering key can stop as soon as the key value or the first value after it is found (for successful and unsuccessful searches respectively).

If the collection of records has the additional property that the i^{th} record can be directly accessed without touching any other record in the collection, then algorithms such as binary or Fibonaccian search can be used to further reduce the number of records to be scanned.

3.5.2 Reduction of size of records scanned

Two methods in category 2 will be considered. They are:

- abbreviation
- compression

Abbreviation is the shortening of a record to one or perhaps several keys and providing a link to the rest of the fields stored elsewhere. The keys in the shortened record represent the record for the purposes of scanning. Abbreviation is normally used with one or more of the category 1 methods because by itself it does not usually reduce search space volume sufficiently. Abbreviation is particularly profitable when the data has to be transferred across memory levels in order to carry out the scanning and comparison operations since it cuts down the amount of data that has to be transferred (data transfer usually being an expensive operation relative to comparison in primary memory). It may also be advantageous in dealing with variable-length records. If their key is fixed-length, or can be made to appear to be fixed-length, then some kind of algorithm better than sequential search can be used by ordering the keys. A one-level primary index is an example of the use of ordering in conjunction with abbreviation. A secondary index in which the accession lists* are ordered is an example of partitioning and ordering plus abbreviation. An example of abbreviation

* a list of records with a particular key value

used by itself is the proposed use of an associative disc to provide partial inversion described in [Coul72]. The file to be partially inverted is stored on a larger, slower and cheaper moving-head disc. The fields selected for inversion and the address of the complete record on the moving-head disc are stored on the associative disc. It can then be searched for abbreviated records containing the desired key values and a link to the full record on the larger disc.

Compression [Ruth72] is the transformation of a string of bits or characters into a shorter string. Although at first sight compression meets the criterion of reducing record size, it is not generally used in conventional file organisations because compressed data normally has to be decompressed before comparison. Thus the number of bits to be scanned effectively remains the same. The only possible gain is a reduction in transfer time and this is offset by the time required for decompression. Compression is often used in DBSs as an encoding level technique, however, because of its potential for increasing the capacity of secondary storage devices, channels and communication lines.

3.5.3 Connecting record collections

The preceding sub-sections have frequently mentioned record collections. Often the difference between two file organisations lies only in the way in which membership in the record collections it defines is represented. For example, a multilist and inverted list are logically the same in that for each secondary key value there is an associated list of records having that value. The list defines a logical collection of records. The difference between the two organisations lies solely in the way the list is represented, i.e. in the encoding of the logical connections. If the original record is

regarded as pure data and a new super-record is defined which contains the data plus linkage information concerned with representing its membership in record collections, then the difference can be conveniently stated in terms of Severance's ideas about inter-record and intra-record connections. In the multilist organisation, inter-record connections are indirect and intra-record ones direct; in an inverted file it is the converse, inter-record connections are direct and intra-record ones indirect. These points suggest that a powerful and general model for describing file organisations, and thus access paths, can be developed by combining the ideas of this section on reducing search space volume with Severance's ideas concerning record connections. This is the basis for the access and encoding levels of PRODD structural specification described in the next chapter and the foundation of their generality.

3.6 An example for demonstrating the use of PRODD

This example has been adapted from the March 1976 special issue of *Computing Surveys* (vol. 8/1, 1976) on DBMSs. It is known as the presidential database example and is introduced in Fry and Sibley's paper [Fry76]. For detailed exposition of PRODD's descriptive mechanisms, a section of the presidential database will be used to provide a more manageable and compact example. This section consists of information about presidents, states and administrations. It will be known as *the PSA example*. Figure 3.6 is a diagram of the entities and relationships using P. P.-S. Chen's conventions [Chen76]. Three entities are modelled: presidents, states and administrations. Presidents have headed one or more administration and are native sons of one state. Each state was admitted during a particular administration with the exception of

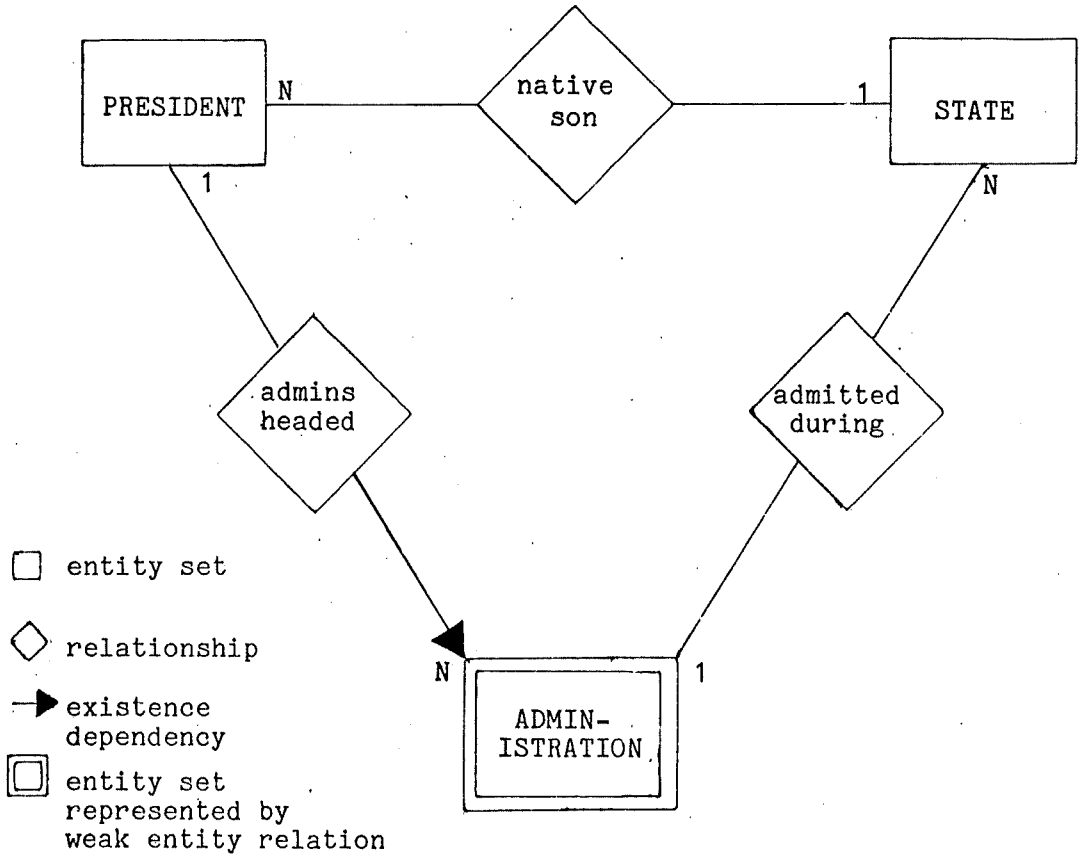


Figure 3.6: The PSA Example

the original thirteen states of the Union. The ADMINISTRATION entity set is represented by a weak entity relation to show its primary key includes the primary key of PRESIDENT. Furthermore, the directed edge linking relationship admins-headed and entity set ADMINISTRATION indicates an existence dependency of ADMINISTRATION on PRESIDENT. This reflects the fact that information about administrations is only relevant in the presence of information about the president who headed those administrations.

4. STRUCTURAL SPECIFICATION

Structural specification was earlier identified as the central component of a DBS description since its function is to define which operation requests are legitimate and how legitimate requests can be executed. Thus it is the first component of a PRODD DBS description and the first component to be discussed in detail. The discussion will be in three main sections, one for each of the three levels of structural description introduced in Chapter 3. These were:

- the information level
- the access level
- the encoding level

DIAM was mentioned in chapter 2 as the starting point for the development of the PRODD model. More precisely, it served as the starting point for the development of the structural specification component of PRODD. The overall organisation of DIAM and the PRODD structural component is the same but the mechanics of description in PRODD have been completely rethought. It was decided, however, that PRODD constructs with the same or similar function as in DIAM should retain their DIAM names to aid the reader already familiar with DIAM and its concepts. A more detailed review of the similarities and differences between PRODD and DIAM is given in the last section of this chapter.

4.1 A network realisation of the PSA example

PRODD is intended as a tool for describing actual DBSs. Therefore, something more concrete than the collection of entity sets and relationships introduced as the PSA database in the previous chapter is needed as an example. A network (DBTG) realisation of the PSA example

has been chosen for this purpose. Its subschema may be found in the tutorial paper on Codasyl DBTG systems [Tay176] in the *Computing Surveys* special issue. In those cases where a particular DBMS has to be assumed, Xerox's EDMS* [Xero73] will be taken as the standard.

A network, rather than relational or hierarchical, realisation has been chosen because the network data model is the most complex of the three. Its constructs are a superset of those of the other two models [Date76]. A demonstration that PRODD can describe a reasonable approximation to an arbitrary network-type DBS should also serve as evidence that the same could be done for hierarchical or relational-type DBSs. A further argument for concentrating on a network realisation is that the network model is widely known and has been successfully implemented in a number of commercial systems.

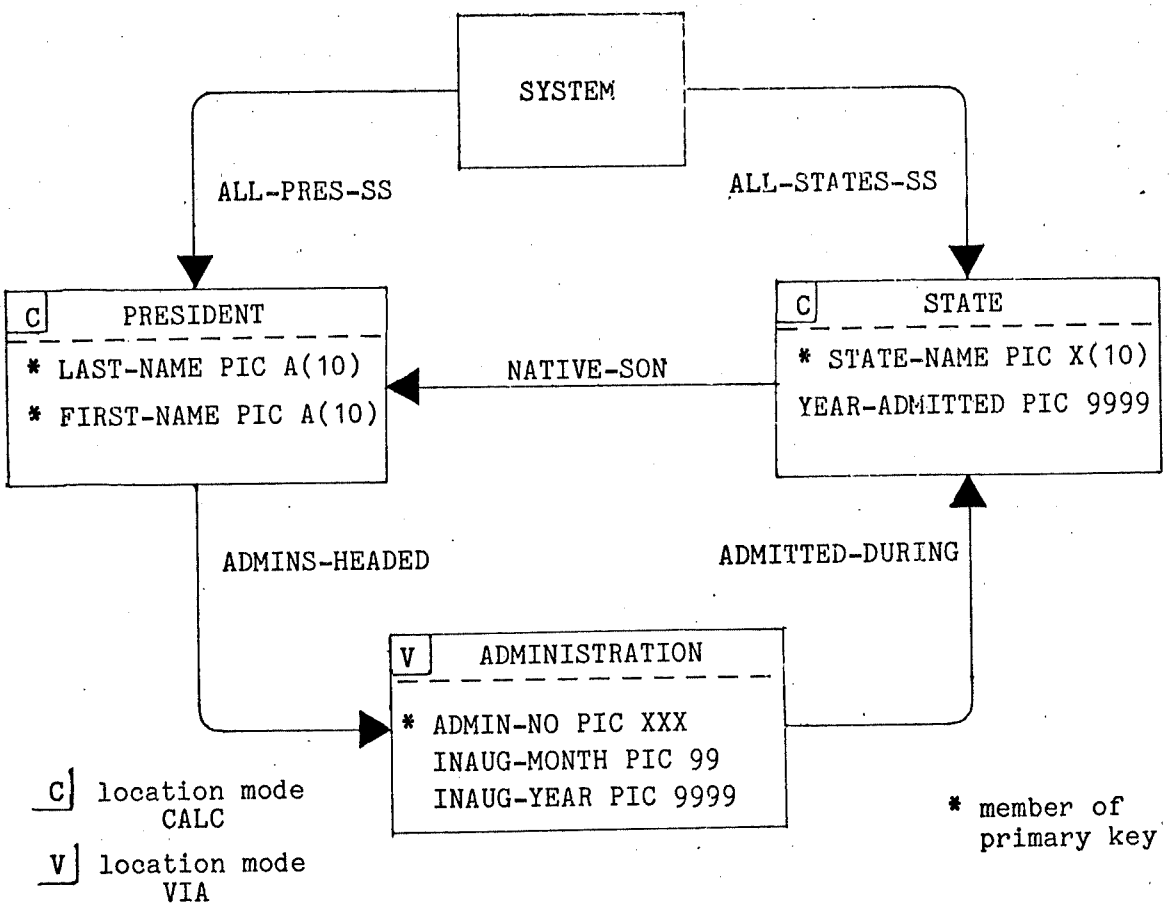


Figure 4.1: Network Realisation of the PSA Example

* EDMS is an implementation of the 1971 DBTG report [Coda71].

The augmented data structure diagram shown in Figure 4.1 can be used to summarise the structural information provided by a network schema. A PRODD description of such a structure must capture all details relevant to the previously stated goals of first-order prediction of database access time and storage requirements. In the following discussion the term *coset* due to Nijssen [Nijs75] will be used in referring to DBTG sets to avoid confusion with the use of set in its customary mathematical sense. The names of DBTG constructs will always be written in capital letters -- for example, record type PRESIDENT and coset type ADMINS-HEADED -- and the names of PRODD constructs appearing in plain text will be enclosed in double angle brackets -- description set <<rams>> for example. Figure 4.1 will now be examined with a view to seeing what information it provides corresponding to the three levels of PRODD structural description.

Starting with information structure, Figure 4.1 defines a database holding facts about classes of entities and their inter-relationships. The three record types represent three entity classes and the three coset types indicate relationships between entities from those classes. Entity properties are represented by the data items associated with each record type. Every record type has a set of data items making up a primary key.

Access structure is embodied in the cosets and the CALC location modes. Singular cosets ALL-PRES-SS and ALL-STATES-SS and CALC location mode for record types PRESIDENT and STATE define *entry points* or known starting points for accessing data item values. Singular cosets ALL-PRES-SS and ALL-STATES-SS are positional access paths while CALC location modes for PRESIDENT and STATE establish two associative access paths. There are another three positional access paths indicated by the

cosets NATIVE-SON, ADMINS-HEADED and ADMITTED-DURING.

Encoding-level information for data values is directly specified by the PIC declarations for the various data items. Different encodings of instances of logical access paths, i.e. coset occurrences, can be specified with the DBTG concept of set mode*. Set modes are not shown in Figure 4.1 but for the purposes of this discussion they will be assumed to be CHAIN with NEXT pointers only.

4.2 The information level

Returning to the model of a DBS as a "black-box" repository of facts, the first concern is to define which facts can be placed in the repository. The mechanism in PRODD for doing this is simple because, as explained in Chapter 1, it is primarily intended for use in the context of physical database design. It was therefore considered unnecessary to attempt to include as much semantic information as is expressed by some recent data models such as Chen's entity-relationship model [Chen76].

PRODD uses the following model of the real world in formulating an information-level description. There is a universe of entities or objects. Each entity has properties, some distinguishing it from and some shared in common with other entities. The information level specifies which entities are described by the database by listing the properties of those entities chosen for representation in the database. A *fact* can now be defined as a value for one property of one entity. From the information-level viewpoint the database maintained by a DBS is simply a collection of facts.

To aid in organising a database into a usable form it is desirable

* Set mode was included in the 1971 DBTG report [Coda71] but was subsequently dropped in the 1973 Codasyl JOD [Coda73] revision of the 1971 report.

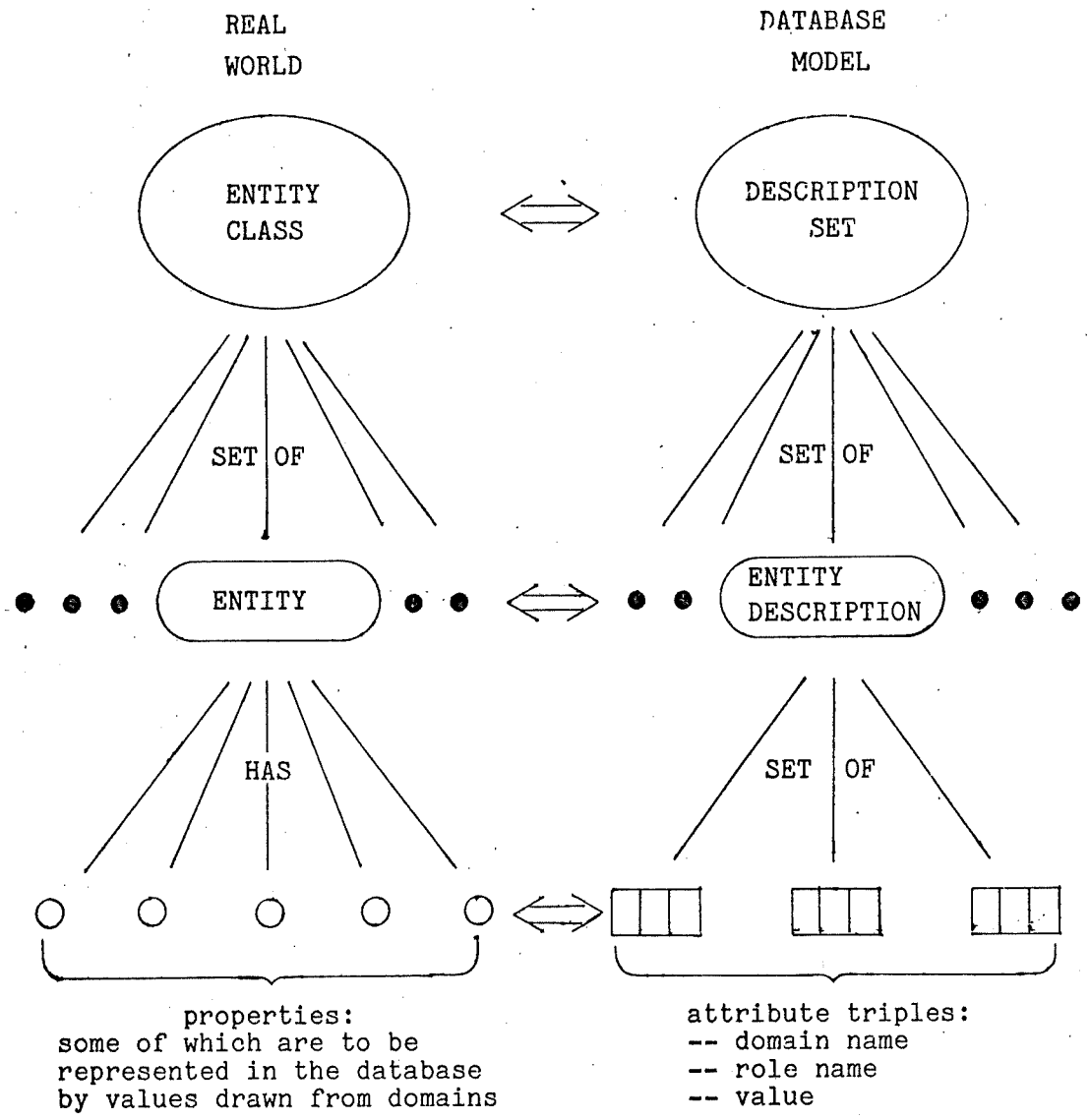


Figure 4.2: Correspondence between Entities and Description Sets

to impose a structure on the jumble of facts defined above. A first step is to observe that entities can be grouped into classes on the basis of their properties chosen for inclusion in the database. An *entity description* is the set of property representations stored in the database for some entity. Entity classes are formed by grouping together all entities having the same set of properties chosen for description. Entity descriptions are similarly grouped into *description sets*. There is a natural 1:1 mapping between entity classes and the description sets which define them. Figure 4.2 illustrates the correspondence between entities in the real world and their representation by descriptions stored in a database.

An example may help to clarify these points. Suppose the owner of a large sheep farm decided to use a database to keep track of his animals. The basic entity to be described is a sheep. Properties of a sheep chosen for description might be

(name, age, weight, colour, sex).

All sheep entities have the same description and are therefore in the same class. There is one description set <<sheep>> having all sheep descriptions as members. This picture of things is illustrated by Figure 4.3.

Now suppose that the owner wants to co-ordinate his breeding program instead of just keeping track of his animals. In this event he will probably want to record additional information besides the five properties listed above. Furthermore, these additional properties will depend on an animal's sex. Number of offspring and year first bred might be relevant for ewes while for rams the pertinent fact might be whether they are to be retained for breeding or to be sold.

entity: sheep
application: keep track of sheep
one description set:

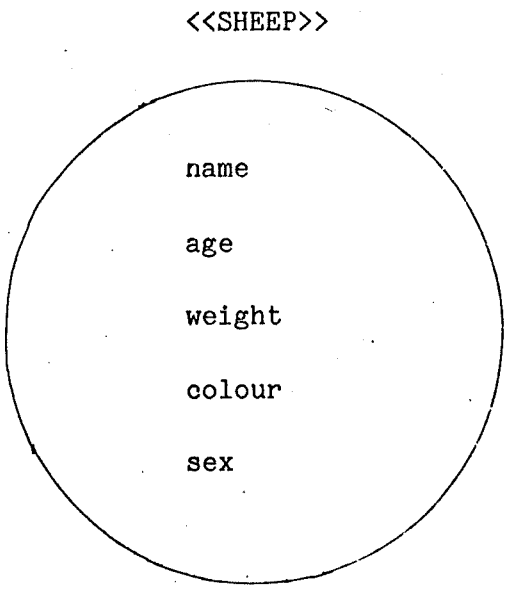


Figure 4.3: The <<sheep>> Description Set

Now there are two description sets, <<rams>> and <<ewes>>, and correspondingly two classes of sheep entities. The revised picture of things is shown in Figure 4.4.

An entity description was earlier defined as a set of property representations for some entity. How is an entity property represented for storage in a database? In PRODD, the representation of the entity property is defined by an *attribute*. Accordingly, an entity description is now redefined as a set of attributes describing some entity.

An attribute is a triple consisting of a role name, domain name and data value. A *domain* is simply a set of possible data values. Naming a domain specifies the set from which attribute values can be

entity: sheep
application: co-ordinate breeding program
two description sets:

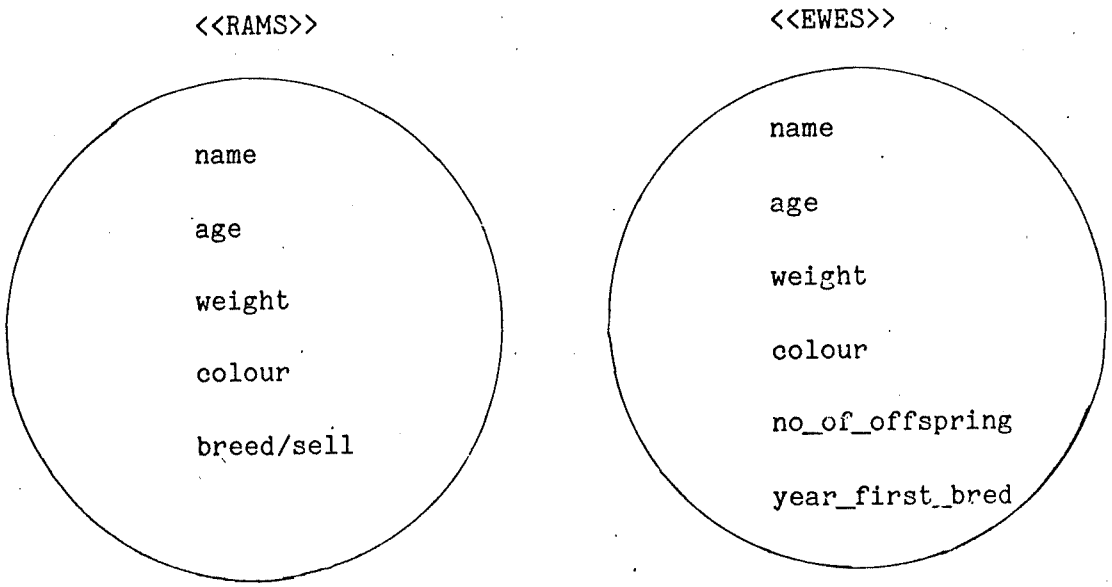


Figure 4.4: The <<rams>> and <<ewes>> Description Sets

drawn. Domains are said to *underlie* attributes. The role name identifies the role played by the associated value in describing the entity. An entity description is a set and hence unordered; a role name acts as a selector for one element of the description and allows several attributes to draw values from the same domain without confusion.

There is one more information level construct, the *identifier*. An identifier for a particular description set is a minimal set of its attributes which uniquely identifies every possible description in that set*. By minimal it is meant that removal of any attribute

* The set of all possible descriptions in a description set is the cross-product of the domains underlying the attributes of the descriptions in that set. The requirement of unique identification for every possible description ensures that a set of attributes classed as an identifier will have that property throughout the lifetime of the database and for any possible combination of entity descriptions.

from the set would nullify its property of unique identification. The PRODD model demands the existence of at least one identifier for every description set. This guarantees a logical access path to every entity description and its attributes and thus to every fact in the database.

It may be helpful to mention that a description set is equivalent to what Codd calls a relationship or domain-unordered relation [Codd70]. Attribute, role name and domain name have the same meanings as in Codd's relational model with the proviso that here domain is restricted to what Codd calls a simple domain[+]. An identifier is equivalent to a candidate key [Codd71].

Figure 4.5 illustrates an information-level view of the PSA

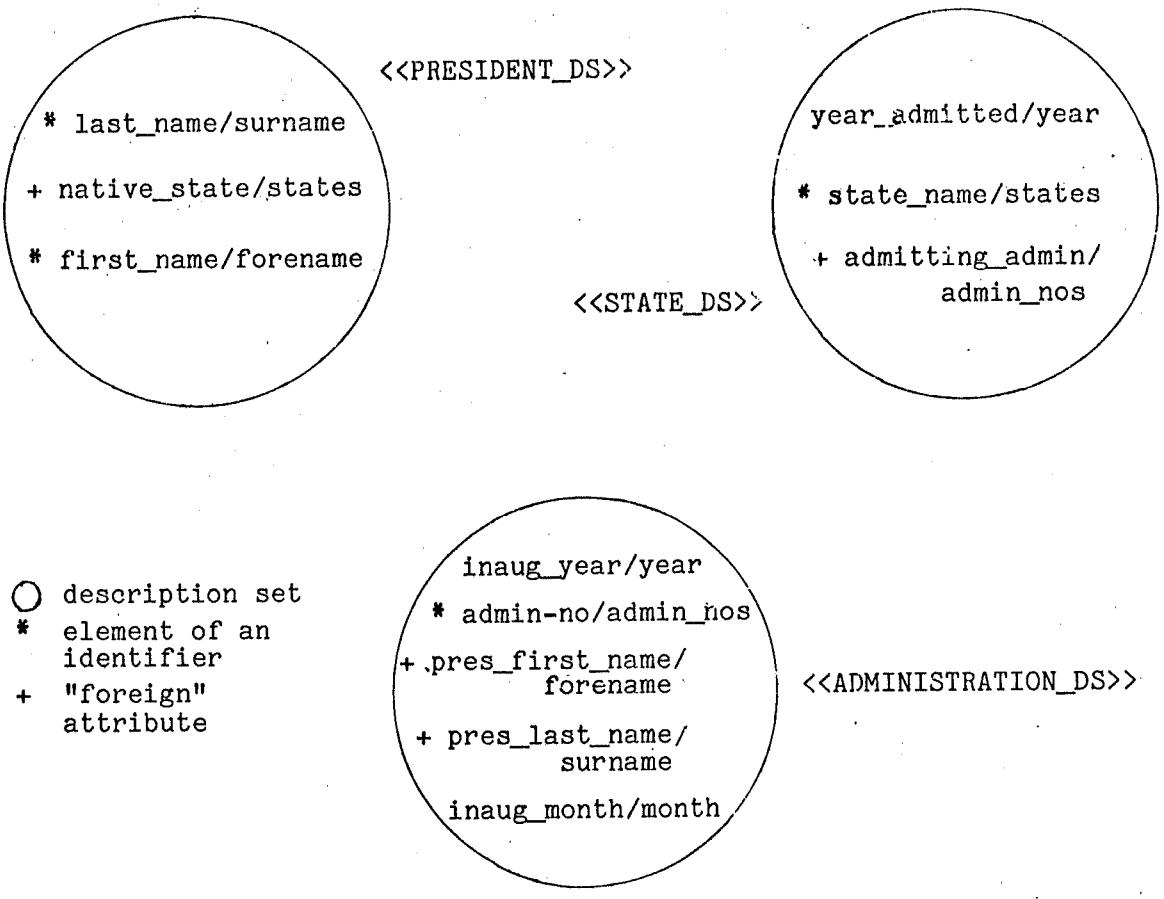


Figure 4.5: Information Level for the PSA Example

[+] This restriction effectively forces the information level to be in a form equivalent to the relational first normal form. There is no loss of generality, however, as there is a simple algorithm for going from unnormalised to first normal form which is given in [Codd71].

example. There are three description sets: one for presidents, one for states and one for administrations. Within the circle representing each description set appear the names of its attributes. No particular ordering is implied. There is one attribute for each data item in the corresponding network record type. In addition there are a number of "foreign" attributes (marked with a '+') used to model membership in information-bearing cosets [Meta75] representing 1:n relationships between record types. 1:n relationships can be modelled at the PRODD information level by including an identifier of the "owner" description set among the attributes of the "member" description set. At the access level it will be possible to indicate that these foreign attributes are not actually present as accessible values but are instead represented by permanent access paths. Each attribute appears in the diagram as a role name, domain name pair with '/' acting as a separator. In the PSA example each description set has exactly one identifier; the attributes comprising the identifier are marked with a '*'.

A PRODD information-level description is thus built out of four primitive constructs:

- the description set
- the domain
- the attribute
- the identifier

A formal, textual specification of the information level for a DBS consists of the declaration of all domains followed by the declaration of all description sets. Domains are simply named at present, although additional information defining valid values and other consistency/integrity constraints could conceivably be included. Description sets are declared

by listing all their attributes and specifying one or more identifiers.

Figure 4.6 is a fragment of a PRODD description declaring description set <<president_DS>> and its underlying domains. The first few lines place the information-level fragment within the structural specification component of a complete DBS description. These constructs are all named to make it possible to refer to different versions of them.

```
DBS_DESCRIPTION  PSA_example

  STRUCTURE_SPECIFICATION  network_realisation

    INFORMATION_LEVEL  PSA_entities
      .
      .
      .
      DOMAIN forename
      DOMAIN surname
      DOMAIN states
      .
      .
      .
      DESCRIPTION_SET  president_DS
        ATTRIBUTE  first_name / forename
        ATTRIBUTE  last_name / surname
        ATTRIBUTE  native_state / states

        IDENTIFIER  [last_name,first_name]
      END_SET
      .
      .
      .
  END_LEVEL
  .
  .
  .
```

Figure 4.6: Information Level Fragment

4.3 The access level

After the facts to be stored in the database have been declared, the next step is to describe how they can be accessed. PRODD has four primitive constructs for building a description of access structure:

- the AVC
- the A_string
- the C_string
- the L_string

These four constructs make it possible to describe database access structure as a combination of the fundamental methods of file organisation previously explained in the context of files, records and fields. The information-level view of a database consists of description sets, entity descriptions and attribute values and those methods will now be restated in these terms.

At the information level a database is a simple, unordered collection of entity descriptions. Clearly, any particular description in the database could be located by scanning all entity descriptions and picking out the one having the specified attribute value or values. Once again, this would be too slow for any database of reasonable size and so some form of organisation must be introduced. All the methods mentioned for records and fields can be applied equally well to entity descriptions and attributes. A collection of entity descriptions can be partitioned, restricted or ordered and individual descriptions can be abbreviated to their identifiers or some other subset of their attributes.

At the information level one thinks of a single stored instance

of each attribute value. However, in using file organisation techniques to speed up access, it may be necessary to store additional copies of some attribute values. This leads to the concept of *AVCs* (attribute value copies) and *A_strings* (AVC strings) which declare them. The A_string construct can be used to describe the use of abbreviation as a means of reducing search space volume.

Another construct exists to describe the organisation of collections of entity descriptions using partitioning, restriction and ordering. This is the *C_string* (collection string). Finally, there is a construct to link various collections together, for example, a collection of full descriptions to a collection of abbreviated ones, the latter perhaps being further organised by partitioning or ordering. This is the role of the *L_string* (link string). These four constructs will now be explained in detail using the PSA example as a common point of reference.

A *minimal access structure* for a given information structure is one which provides for exactly one instance of every unique fact presented to the DBS for storage and exactly one access route to the representation of each fact. An *access route* is a path beginning at a known starting point. A minimal access structure is sufficient to perform any legitimate operation request directed to the DBS. Hence a basic constraint on any proposed access structure is that it contains a minimal access structure. While this constraint guarantees that any legitimate operation request can be correctly dealt with, it does not provide any assurance that database access times are likely to be satisfactory. The access level describes how the two basic techniques of redundant storage of facts and multiple access paths are used to provide acceptable database access times.

An example of a fact in the PSA database is the year in which a state was admitted to the Union. In the network realisation there is exactly one instance of this fact, the value of the YEAR-ADMITTED data item in the STATE record type. While this particular fact is only recorded in one place, there are several paths to it from different known starting points. For example, one path starts at ALL-STATES-SS and can be followed to an individual STATE record occurrence which gives access to all data item values stored therein including YEAR-ADMITTED. The four access-level constructs will now be explained in detail using the PSA example as a common point of reference.

4.3.1 The A string and AVC

The purpose of the A_string is to declare the existence of individual copies of the values of the attributes of a description set. Since the access level is concerned with defining access paths along which scanning can logically proceed, a definite ordering must exist. Consequently, an A_string defines a sequence of AVCs. It is said to be *over* them and they *under* it. Over and under are similarly used when referring to the other string constructs for organising collections of access-level elements.

AVCs are said to *cover* the attributes for which they are value copies. The minimal access structure constraint implies that every attribute declared at the information level must have at least one copy of its value, i.e. an AVC, declared at the access level. Thus the first part of the minimal access requirement may be rephrased as "every attribute must be covered".

The A_string and other strings are essentially type declarations. Actually existing in the database are instances of those types. In the case of A_strings there is one potential instance for every distinct set of values of the AVCs it defines*. The A_string defines a logical path along which the AVCs can be visited in the order of their declaration. Figure 4.7 shows the access-level diagram for the PSA example with the

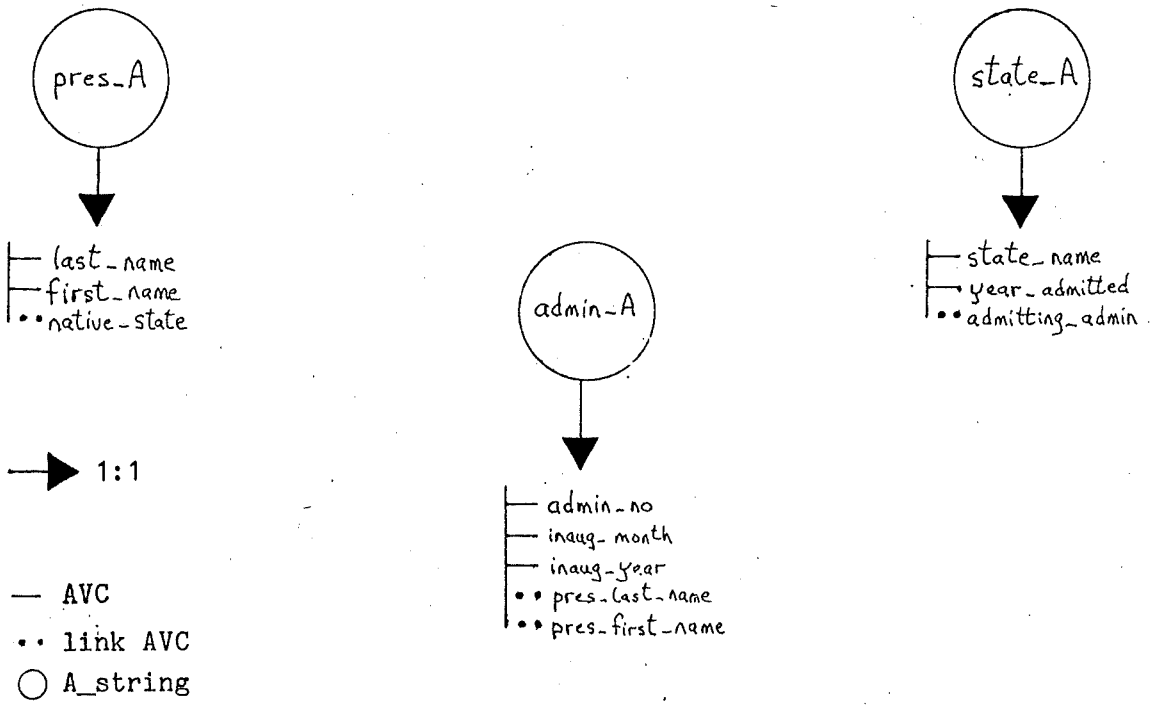


Figure 4.7: A_strings and AVCs for the PSA Example

appropriate A_strings and AVCs.

* Therefore, in order to have one A_string instance for each description in a description set, the set of attributes covered by the AVCs of the A_string must include at least one identifier.

Two types of AVC appear in Figure 4.7, *actual* and *link*. There is a third type, not shown,termed *derived*. The type of an AVC depends on the role it plays in the access structure. The simplest type of AVC is actual. This is the case in which existence of the AVC is simply represented by physical storage of the appropriate value.

Derived AVCs declare the existence of attribute value copies which are derived from the values of other attributes present in the database. Derivation may be defined as simple copying of an AVC for some other attribute or as the execution of a procedure taking AVCs as arguments. There is a further choice between storing derived values or rederiving them each time the AVC is accessed. That choice is relegated to the encoding level. Derived AVCs are basically similar to the DBTG notion of SOURCE and RESULT data items with the PRODD equivalent of choosing between ACTUAL and VIRTUAL being left to the encoding level.

Link AVCs are neither physically present in the database nor accessible for retrieval. Their function is to define the connections to be made when building information-bearing access paths as a consequence of update activity. They are necessary because all grouping and linking of strings at the access level is defined in terms of AVC values, i.e. on the basis of content. The value of a link AVC is defined in the same way as for derived AVCs.

Figure 4.8 is a fragment of a PRODD access-level declaration of A_string <<pres_A>> which defines the AVCs covering all the attributes of description set <<president_DS>> (as shown in Figure 4.7). An AVC name is formed by concatenating the name of its defining A_string with the role name of the covered attribute and using ':' as a separator. Every string declaration must list all strings defined over the string

```
A_STRING  pres_A
  OVER  president_DS :
        (  last_name,
           first_name,
           native_state  LINK
                         SOURCE IS state_A:state_name
        )
  UNDER president
END_STRING
```

Figure 4.8: A_string Declaration

being declared. This makes extra checking by the translator possible and forces the PRODD user to think more carefully about the structures he is describing. This list is known as the UNDER entry and appears as the last entry in the string declaration as in Figure 4.8. The string <<president>> named there will be properly introduced in the following sub-section describing C_strings.

4.3.2 The C string

The C_string is a powerful grouping construct which can define various combinations of the category 1 file organisation methods. An example will help motivate the presentation of the C_string. Figure 4.9 depicts the set of all instances of A_string <<admin_A>>. There is one instance of <<admin_A>> for each administration entity which has had its entity description recorded in the database. This is guaranteed by the fact that one of its AVCs is <<admin_no>> which covers the attribute of the same name acting as an identifier of description set <<administration_DS>>. Some typical <<admin_A>> instances are represented. Each instance is a triple of three values: one for <<admin_no>>, one for <<inaug_month>> and one for <<inaug_year>>. These value triples are shown in the diagram. Now suppose that for historical purposes one is interested in all administrations inaugurated before 1960, that it is furthermore

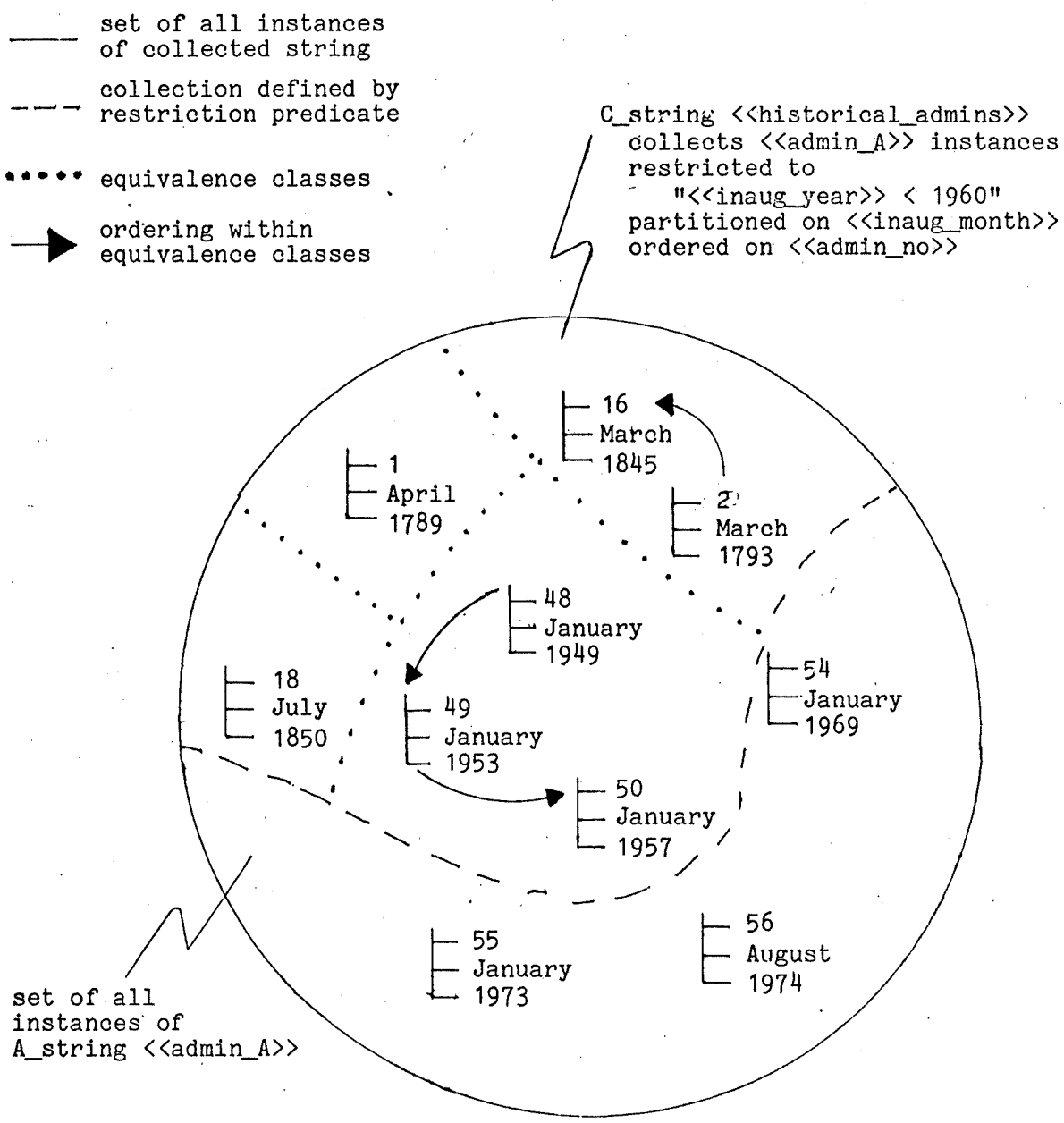


Figure 4.9: C_string Partitioning

desirable to group these historical administrations by month of inauguration, and that within those groupings it would be convenient to have administrations ordered by administration number. All this can be accomplished with one C_string.

A C_string can restrict, partition and order the instances of the string over which it is defined, although not all of these functions need necessarily be exercised. All three functions are defined in terms of accessible AVCs. An AVC, A, is *accessible* from a string, S, if there exists a sequence of strings $<S_1, S_2, \ldots, S_n>$ (n>=1) such that S_i is over S_{i+1} for i=1,...,n-1, S_1=S and S_n is an A_string over A. Intuitively, an AVC is accessible from the A_string which defines it and thence from any higher level string(s) giving access to that A_string.

The first thing a C_string declaration specifies is which string it is defined over, i.e. of which string it is *collecting* instances. In Figure 4.9 C_string <<historical_admins>> has been declared over A_string <<admin_A>>. The next point to be considered is whether all instances of the string are to be collected or only a certain subset. To this end a *restriction predicate* is provided. The restriction predicate may be null, in which case all instances are collected. Alternatively, a disjunctive normal form combination of comparisons on accessible AVCs or a named function with accessible AVCs as actual parameters may be used. In Figure 4.9 the restriction predicate is "RESTRICT ON admin_A:inaug_year < 1960". This is an example of the simplest form of a disjunctive normal form expression used as a restriction predicate.

After restriction the next function of a C_string is to partition the selected instances into equivalence classes. A list of accessible AVCs is specified in the *partitioning list*. A Cartesian cross-product, C, can be defined over the domains of the AVCs in the partitioning list. Each element of C is then a distinct set of possible values for the AVCs in the partitioning list. All instances of the collected string with the same set of values for the partitioning list are grouped into

an equivalence class represented by the element of cross-product C with that set of values. Not all elements of the cross-product will necessarily represent a non-empty equivalence class at any one instance in the DBS's lifetime. Partitioning thus declares a set of logical paths, each logical path being a sequence connecting all the instances in one equivalence class and identified by an element of C. Each such sequence is one instance of the C_string, with the number of C_string instances thereby being equal to the number of non-empty equivalence classes. The ordering which transforms an equivalence class (a set) into a sequence is defined by the C_string's ordering function described subsequently.

Figure 4.9 illustrates the effect of a partitioning list consisting of the single AVC <<admin_A:inaug_month>>. Partitioning on <<admin_A: inaug_month>> results in twelve different equivalence classes and therefore twelve instances of <<historical_admins>> (one for each month of the year). It is also possible to declare partitioning on the basis of a function with an actual argument list of accessible AVCs. There will be one equivalence class associated with each unique value in the range of values returned by the function. Hashing to secondary storage buckets is naturally described by function partitioning. Buckets are equivalence classes represented by the integers in the range of the hashing function. Assuming the ideal case of no overflow, a record is placed in the bucket (equivalence class) identified by the result of applying the hashing function to one or more of its field values. A final possibility is that the partitioning list may be empty. This is taken to mean that there is one equivalence class comprising the entire collection and so only one instance of the C_string. A C_string with an empty partitioning list is said to be *null-partitioned*.

The C_string has been described as declaring a set of sequences of instances of another string. The word "sequence" implies that an ordering exists. This is because the function of strings is to define access paths along which scanning can proceed and scanning is essentially a sequential operation (at least for an individual processor). Furthermore, those access paths are going to be mapped onto linear address space and some ordering will have to be used in performing that mapping.

An explicit ordering may be indicated by supplying a list of accessible AVCs called the *ordering list*. Each sequence over the members of an equivalence class is then ordered using the elements of the ordering list as sort keys. For Figure 4.9 the ordering list is (admin_A:admin_no/A). The '/A' indicates that the sequence is in ascending order by <<admin_no>>. '/D' after an AVC would specify descending ordering on that AVC. The first element of the ordering list specifies the primary sort order, the second the secondary sort order and so on. An empty ordering list is used to indicate that no ordering based on key values exists. In this case one of two possibilities can be specified: last in, first out or first in, first out. A final alternative is to declare an ordering opposite to that of another C_string with an identical restriction predicate and partitioning list. This option permits the description of doubly linked lists or chains such as DBTG cosets with NEXT and PRIOR pointers.

Figure 4.10 is an access-level fragment illustrating the declaration of C_string <<historical_admins>>. The UNDER list has been left unspecified, it not being relevant to this example.

The C_string has one optional entry, the RING entry. It is assumed by default that the sequence linking the members of each equivalence class

```
C_STRING historical_admins
  OVER admin_A
  RESTRICT ON (admin_A:inaug_year < 1960)
  PARTITION ON (admin_A:inaug_month)
  ORDER ON (admin_A:admin_no/A)
  UNDER ( . . . )
END_STRING
```

Figure 4.10: C_string Declaration

is a one-way linear structure and that when its end is reached no further travel is possible. The ring entry exists to describe the case where the last element of the sequence is connected to another string instance producing a one-way circular structure instead. The RING entry can be used to describe DBTG cosets where a coset occurrence is effectively a circular linked list with the owner record occurrence as the list head.

Returning to the PSA example, it was observed earlier that accessing a record occurrence gave access to two types of information: 1) its data item values and 2) coset linkages, i.e. the member occurrences of any coset occurrences owned by that record occurrence. The first type of information has been modelled by the A_strings shown in Figure 4.7. A C_string partitioned on AVC(s) can be used to model the second type. For example, the grouping of PRESIDENT record occurrences into occurrences of the coset NATIVE-SON can be modelled by defining a C_string <<sons>> over the A_string <<pres_A>> with partitioning on the link AVC <<pres_A:native_state>>. Partitioning on a link AVC is here being used to model an information-bearing coset. Two similarly defined C_strings, <<headed>> and <<admitted>>, model ADMINS-HEADED and ADMITTED-DURING. The access-level view of the PSA network with these C_strings included is illustrated by Figure 4.11.

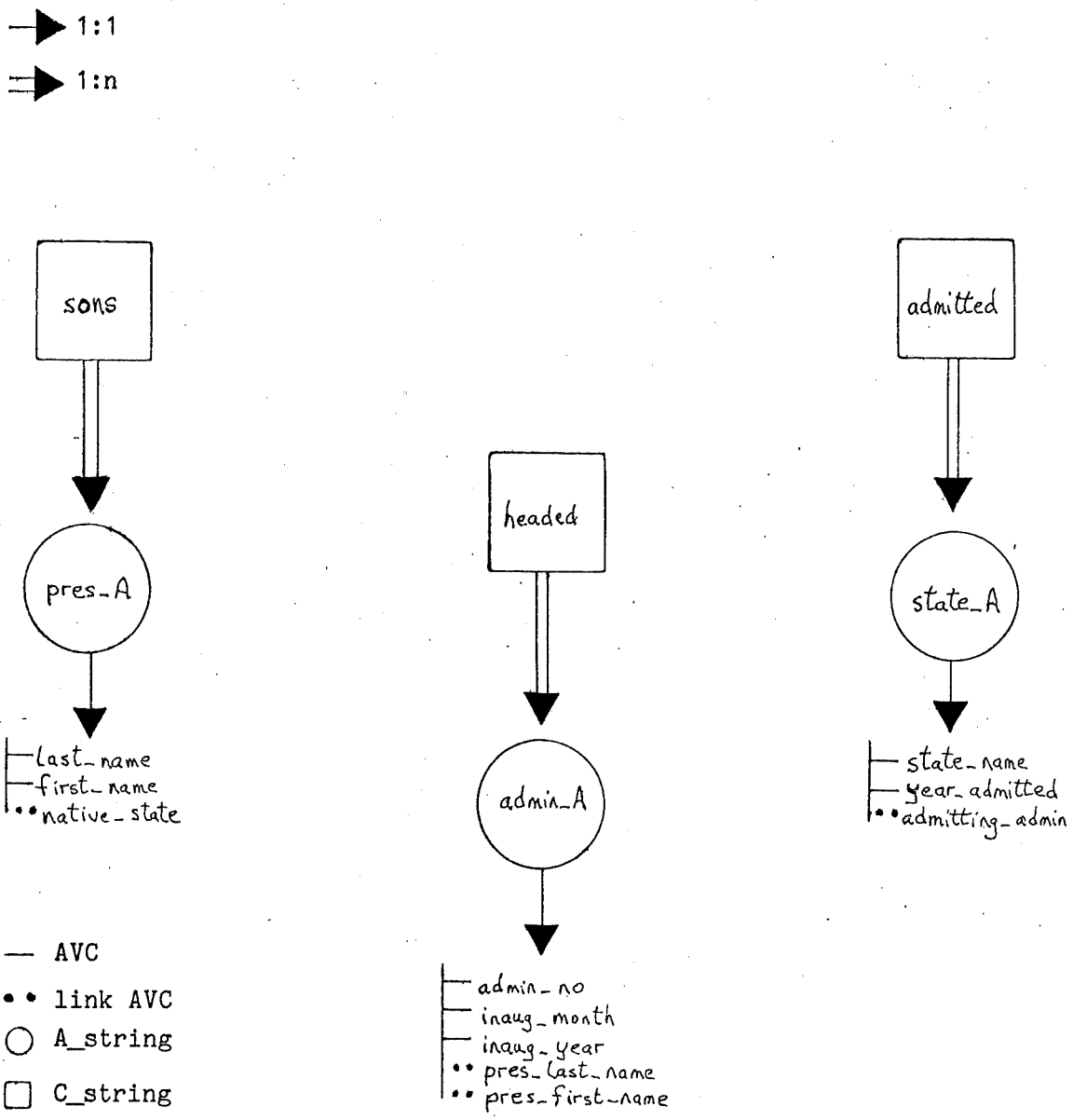


Figure 4.11: C_strings for the PSA Example

4.3.3 The L_string

The idea of the L_string is to establish links between individual instances in different collections of string instances which represent information about the same entities. This is indicated by the fact

that they provide access to AVCs with matching values. An L_string is defined over a sequence of strings of distinct type said to be *linked* by that L_string. The linked strings are related by each having a list of accessible AVCs, called *matching lists*, with values drawn from domains of the same type*. All matching lists must be of the same length. The set of matching lists for all linked strings is known as the *matching criterion* for the L_string.

An instance of an L_string is a sequence of instances of the strings over which it is defined, there being exactly one instance of each linked string in the sequence. Every instance in a particular sequence has the same values for the AVCs in its matching list. For the sake of simplicity at the encoding level, there is a constraint that the matching lists must determine a 1:1 mapping between L_string instances and linked instances. In other words, for a given L_string instance there must be at most one instance of each linked string and every instance of each linked string must only be eligible for inclusion under at most one L_string instance+.

* The values must be drawn from domains of the same type in order that meaningful comparisons can be made.

+ More formally, an L_string, L, is defined over a sequence of other strings $\langle S_1, S_2, \ldots, S_n \rangle$ such that $n \geq 2$, $L \neq S_i$ for $i = 1, 2, \ldots, n$ and no $S_j = S_k$ for $j \neq k$, $1 \leq j$, $k \leq n$. Furthermore, each S_i has associated with it a matching list $ML_i = (A_{i1}, A_{i2}, \ldots, A_{im})$ where A_{ij} is an AVC accessible from S_i for $1 \leq j \leq m$ and m is constant for all i. The matching criterion for L is the set $\{ML_1, ML_2, \ldots, ML_n\}$. Each instance L^I of L is a sequence $S_1^I, S_2^I, \ldots, S_n^I$ such that $ML_1^I = ML_2^I = \ldots = ML_n^I$ where $ML_i^I = ML_j^I$ if the two lists are element-wise equal (i.e. $A_{ik} = A_{jk}$ for $k = 1, 2, \ldots, n$). There is a further requirement that there be a partial 1:1 mapping between the L_string instances and the linked string instances. This constraint may be slightly relaxed in the case of a two-element L_string. Under certain encodings it will be sufficient if the mapping between instances of the L_string and the second linked string is a partial function. This implies that each instance of a linked string can be in at most one sequence, except instances of the second linked string of a two-element L_string which may be at the end of several sequences provided that one of a restricted class of encodings has been chosen.

An example can be constructed by considering the building of a secondary index on president first names. To do this an abbreviated <<president_DS>> description consisting of just the <<first_name>> attribute is defined by an A_string called say <<name_key>>. Instances of <<name_key>> cannot be directly linked to <<pres_A>> instances with the same value of <<first_name>> because more than one president may have the same first name, thereby violating the 1:1 mapping constraint. The solution is to partition the <<pres_A>> instances on <<first_name>> with a C_string called <<same_first_name>>. Now <<name_key>> instances can be linked to <<same_first_name>> instances by an L_string <<index-entry>> matching on <<first_name>>. It only remains to collect all the <<index_entry>> instances together with a C_string <<first_name_index>> and the secondary index is complete. Figure 4.12 is a diagram of the structure just described.

Returning to the PSA example, there are two types of information for each record type, data items and coset linkages. It has already been explained how A_strings can be used to model the data items and C_strings to model the coset linkages. Still to be demonstrated is how to link the data item and coset linkage information for each record type. This is done with an L_string.

Two strings which have already been introduced for representing information about presidents are A_string <<pres_A>> and C_string <<headed>>. Instances of <<pres_A>> are uniquely identified by the AVC list

(pres_A:last_name, pres_A:first_name).

C_string <<headed>> defines partitioning of the <<admin_A>> instances on the partitioning list

(admin_A:pres_last_name, admin_A:pres_first_name).

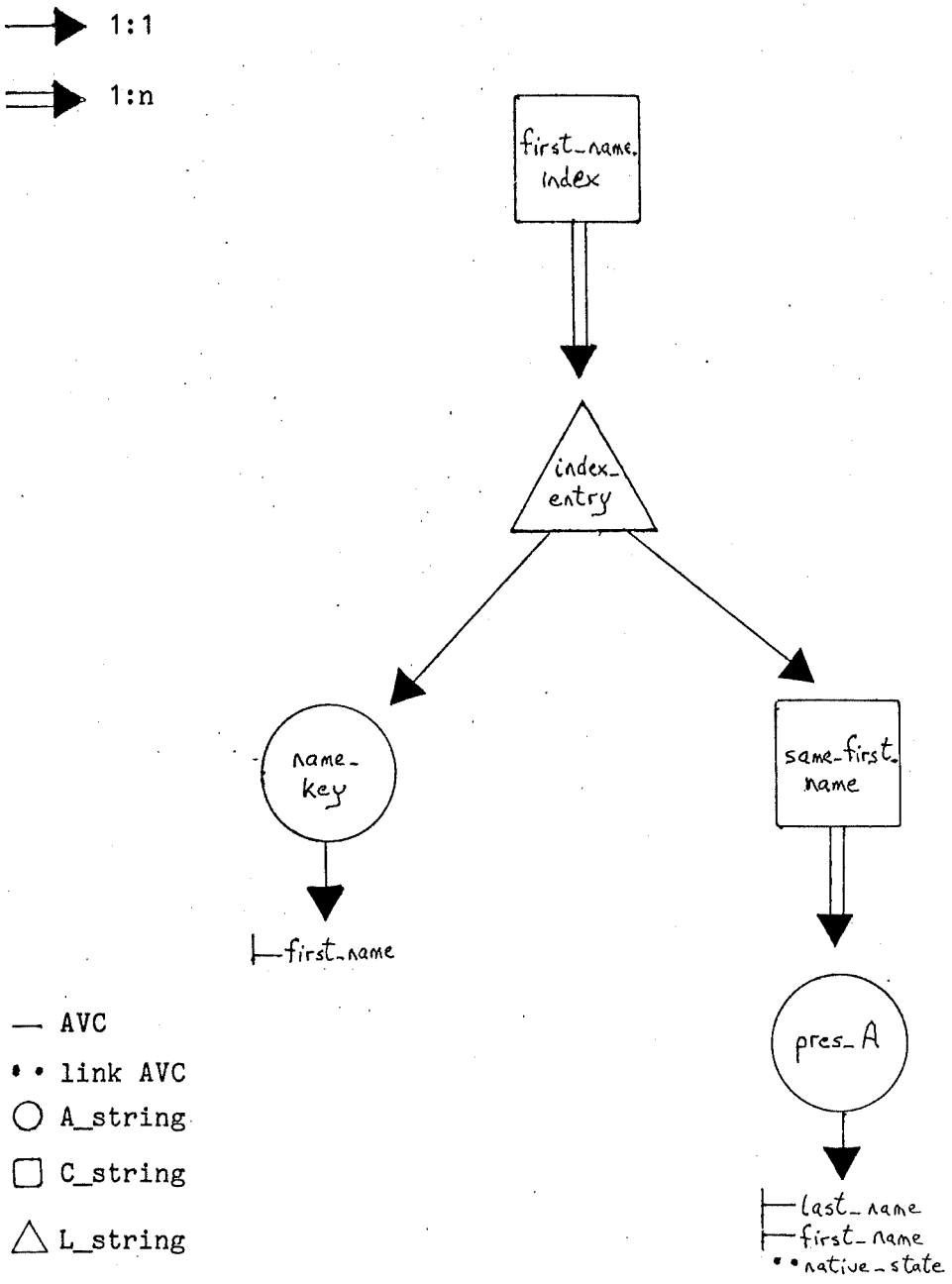


Figure 4.12: Secondary Index on President First Names

The underlying domains for both these lists are <<surname>> and <<forename>>. Hence an L_string defined over the sequence <pres_A,headed> with matching criterion

(pres_A:last_name, pres_A:first_name) =

(admin_A:pres_last_name, admin_A:pres_first_name)

will model access to the same types of information as the PRESIDENT record type. Figure 4.13 illustrates this state of affairs. L_string

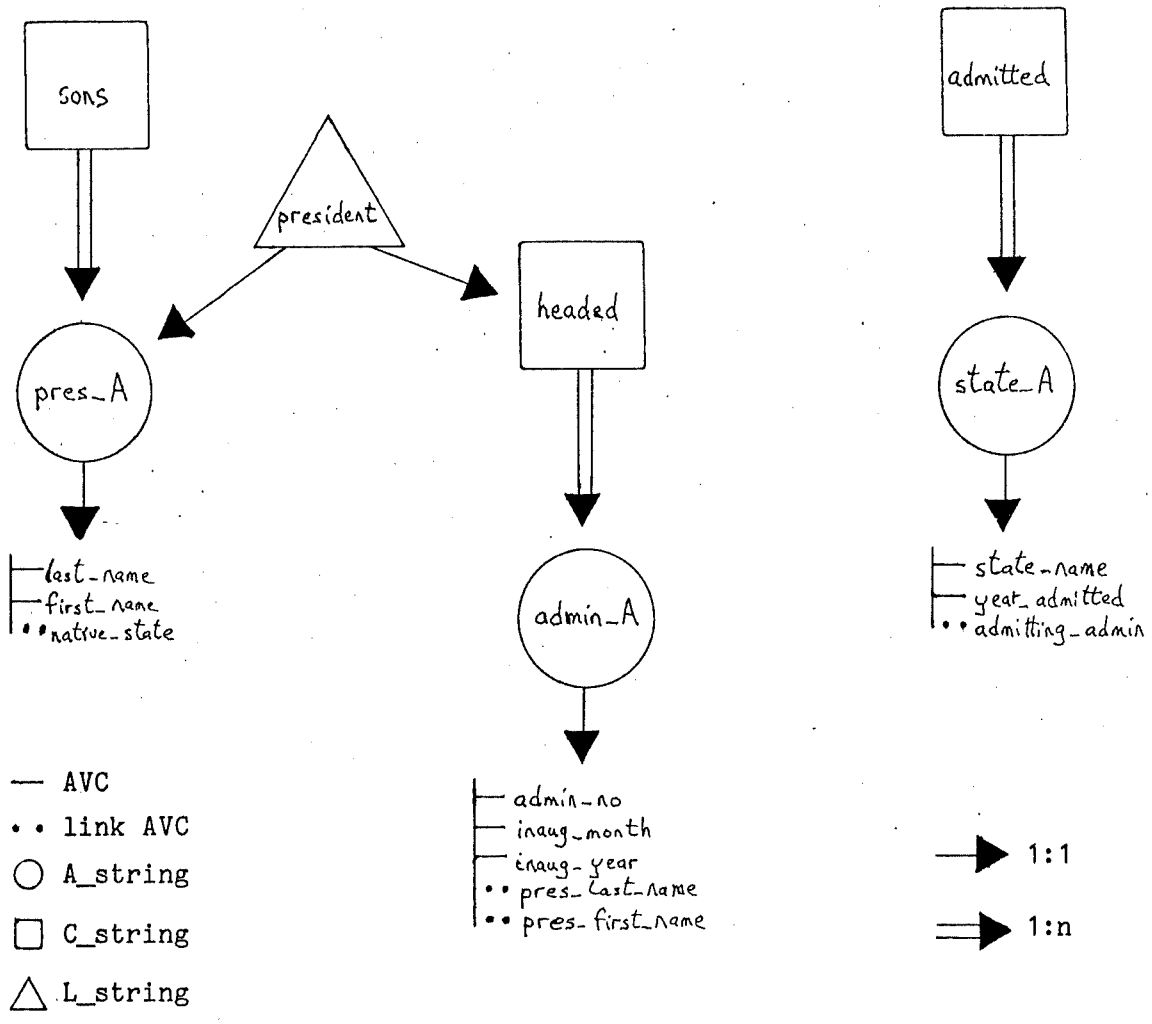


Figure 4.13: First Attempt at Modelling PRESIDENT Record

<<president>> links instances of <<pres_A>> (modelling PRESIDENT data items) with instances of <<headed>> partitioning the instances of <<admin_A>> (modelling ADMINISTRATION data items) into groups headed by a particular president. An example of the role of link AVCs can be seen here. When a new <<admin_A>> instance is added to the database, values must be supplied for the link AVCs, <<pres_last_name>> and <<pres_first_name>>, in order that the DBS can select the correct equivalence class corresponding to an instance of C_string <<headed>>* in which to place the new <<admin_A>> instance. Once this has been done the values of the link AVCs are equivalently represented by collection of the <<admin_A>> instance under the appropriate <<headed>> instance.

However, this representation of the network record is not yet accurate enough. Following an ADMINS-HEADED coset occurrence leads to a set of ADMINISTRATION record occurrences, each in turn providing access to a set of data item values and linkage to STATE record occurrences through the ADMITTED-DURING coset. As presently defined, L_string <<president>> gives access to what corresponds to the set of ADMINISTRATION data items (A_string <<admin_A>>), but not the the representation of the ADMITTED-DURING coset linkage provided by C_string <<admitted>>. This difficulty can be resolved by introducing two more L_strings, <<administration>> and <<state>>, defined analogously to L_string <<president>>. C_strings <<headed>>, <<admitted>> and <<sons>> are now defined over L_strings <<administration>>, <<state>> and <<president>> respectively with the result shown in Figure 4.14. L_string <<presidents>> still links instances of A_string <<pres_A>> and C_string <<headed>> as before. The difference

* <<headed>> partitions <<admin_A>> instances on exactly those AVC values.

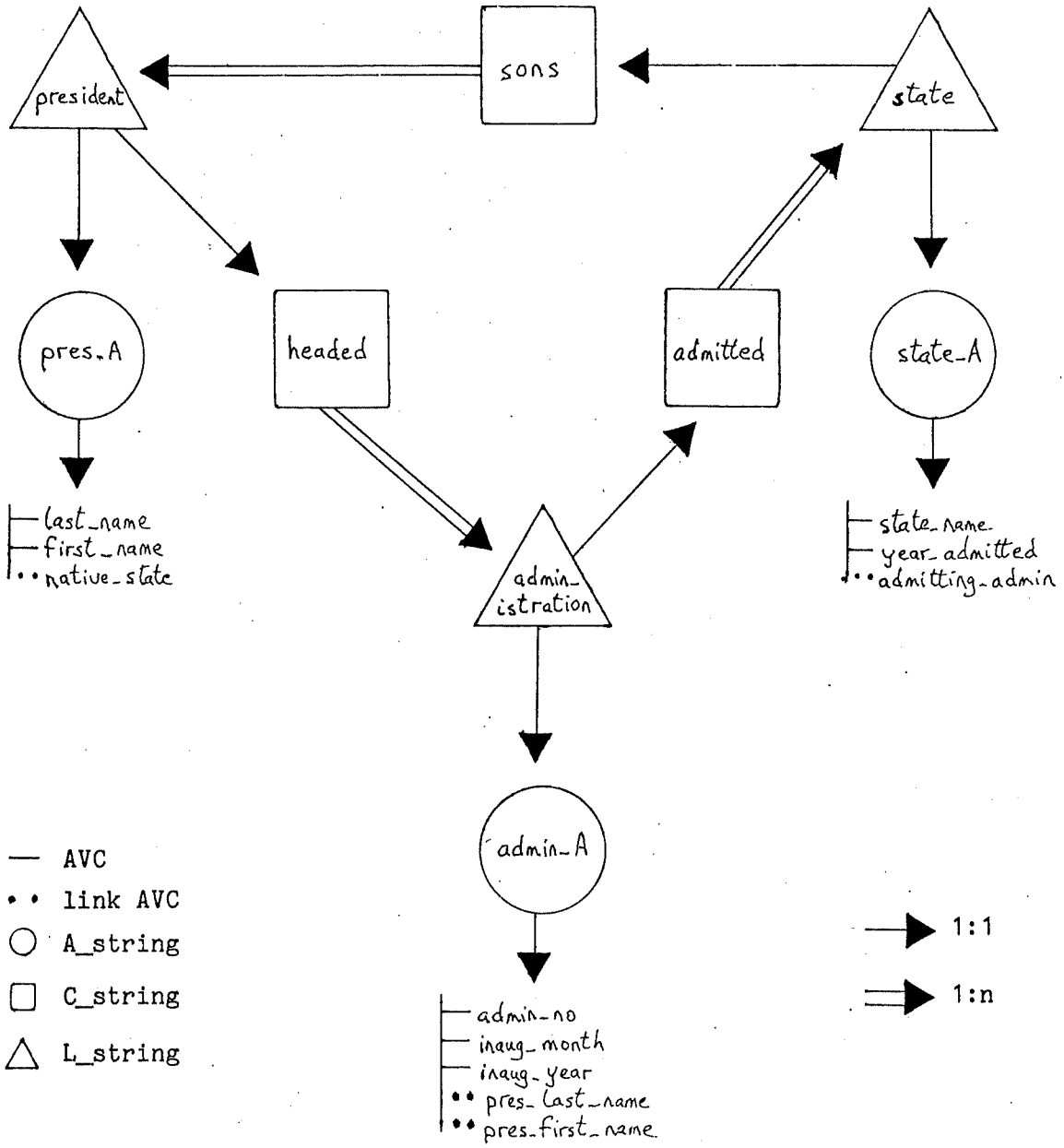


Figure 4.14: L_strings for the PSA Example

is that <<headed>> now collects instances of L_string <<administration>> which are pairings of <<admin_A>> instances with C_string <<admitted>> instances. The coset linkage to STATE record occurrences defined by ADMITTED-DURING is now correctly modelled by the link to <<admitted>> through <<administration>>.

As a further example, Figure 4.15 is an access-level fragment declaring L_string <<president>>.

```
L_STRING president
  OVER (pres_A,headed)
  MATCH ON ((pres_A:last_name,pres_A:first_name) =
            (admin_A:pres_last_name,admin_A:pres_first_name))
  UNDER (sons, . . . )
END_STRING
```

Figure 4.15: L_string Declaration

It is now possible to introduce the notion of identifiers for strings as an extension of the identifier concept for description sets. A string identifier is thus a minimal set of AVCs accessible from a string which uniquely identifies every instance of the string. The partitioning list of a C_string and the matching list of an L_string* are examples of string identifiers. A partitioning list is an identifier by definition because there is at most one instance of a C_string corresponding to an element of the cross-product of the domains underlying the AVCs in the partitioning list. A similar comment applies to matching lists.

4.3.4 Key-access C strings and entry points

Figure 4.14 as it stands models all the positional access paths in the PSA network. Modelling of entry points and associative access paths remains to be described.

The two singular cosets, ALL-PRES-SS and ALL-STATES-SS, are naturally modelled by two null-partitioned C_strings, <<all_pres_SS>>

* In the case of a two-element,partial function L_string the matching list is an identifier of the first linked string only. However, there will still be at least one identifier for the L_string, that being the identifier of the first linked string.

and <<all_states_SS>>. These are defined over L_strings <<president>> and <<state>> respectively, collecting all instances (i.e. the restriction predicate is null) and ordering them on their identifiers. <<all_pres_SS>> and <<all_states_SS>> are themselves linked together under an L_string <<system>> representing the system owner of singular sets.

A special type of C_string, the key-access C_string, is used in describing the associative access paths provided by CALC location mode for the PRESIDENT and STATE record types. Although LOCATION MODE CALC is usually taken to imply hashing this is not a requirement of the DBTG proposals [Date75]. What is required is that the implementor supply a CALC procedure which accepts a CALC-key as input and produces an appropriate database key as output. With this definition the CALC concept is one of general key-to-address transformation or software content-addressing.

A similar concept is embodied in the key-access C_string of PRODD. The idea is that collected instances will be partitioned into equivalence classes on the basis of a list of accessible AVCs, as for standard C_strings. This list is called a *key-access list*. However, when supplied with a set of values for the key-access list the DBS will provide immediate access to the beginning of the sequence linking the elements of the equivalence class identified by that set of values. *Key-access* is declared to be either *primary*, meaning the key-access list is an identifier for the collected string and each equivalence class will contain one instance or be empty, or *secondary*, in which case an equivalence class may hold zero to many instances. The mechanism employed by the DBS to support immediate key-access is not specified at the access level. This is done at the encoding level instead, in accordance with the principle that the access level deals with the logical paths which exist and not with their

realisation.

Returning to the description of CALC location mode, primary key-access C_strings <<pres_C>> and <<state_C>> are declared to model CALC location mode for the PRESIDENT and STATE record types respectively. Their key-access lists consist of the AVCs corresponding to the CALC-keys of the two record types.

An access-level fragment declaring <<system>>, <<all_pres_SS>> and <<pres_C>> is illustrated in Figure 4.16. <<system>> and <<pres_C>>

```
L_STRING system
  OVER (all_pres_SS,all_states_SS)
  MATCH ON ()      || because children are 1-instance
  KNOWNPOINT
END_STRING

C_STRING all_pres_SS
  OVER president
  RESTRICT ON ()
  PARTITION ON ()
  ORDER ON (pres_A:last_name/A,pres_A:first_name/A)
  UNDER system
END_STRING

C_STRING pres_C
  OVER president
  RESTRICT ON ()
  PRIMARY KEY_ACCESS ON (pres_A:last_name,pres_A:first_name)
  ORDER ON ()
  KNOWNPOINT
END_STRING
```

Figure 4.16: Entry Point Strings

are declared as KNOWNPOINTs instead of providing an UNDER list as in the earlier string declaration examples. A KNOWNPOINT is a known starting point for access routes through the database. Therefore, only key-access C_strings or strings known to have just one instance can be declared as KNOWNPOINTs. The latter are termed *1-instance*. A 1-instance string is either a null-partitioned C_string or an L_string defined over 1-instance

strings.

Aside from known starting points, there is another method of locating a beginning for an access route. This is the scanning of all instances of a given string looking for a match on accessible AVC values. Thus any string (A, C or L) can be designated as a SEARCHPOINT, meaning it can be the starting point of an access route where the appropriate string instance will be located by sequential search. Every DBTG record is effectively a SEARCHPOINT since one of the variants of the FIND command permits scanning of all occurrences of a record type within a specified AREA.

All access routes must start at a KNOWNPOINT or SEARCHPOINT string instance. Hence such strings are collectively referred to as *entry points*. In both cases there is a known algorithm for finding a single instance of a string at which to begin traversal of the access route.

Figure 4.17 presents a diagram of the complete network of strings and AVCs modelling the access structure of the PSA example. A formal PRODD description of access structure declares the strings and AVCs corresponding to all fact instances and the logical access paths to them.

4.3.5 String structure

A network like Figure 4.17 is called a *string structure*. It is constructed from four primitives -- AVCs, A_strings, C_strings and L_strings -- collectively known as *SSEs* (string structure elements). A string may be under several other strings, as are L_strings <<president>> and <<state>> in Figure 4.17. AVCs are always under one string only, the

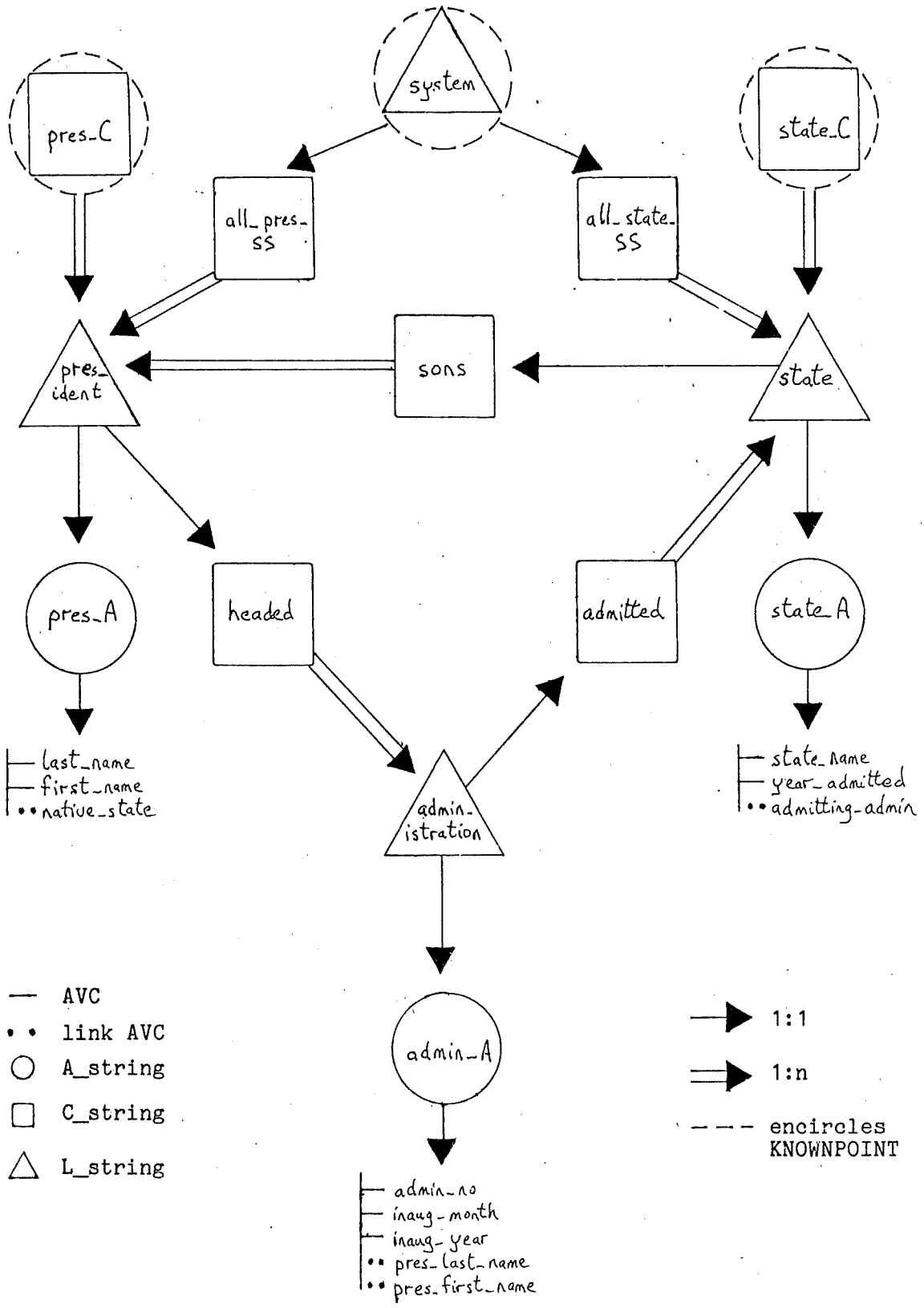


Figure 4.17: Complete Access Level for the PSA Example

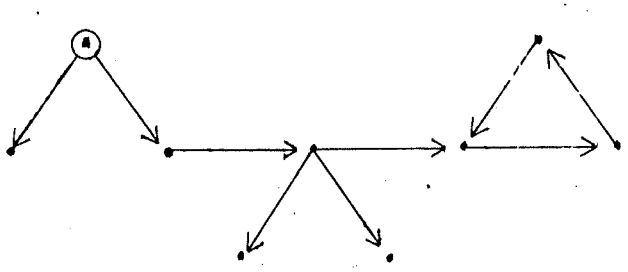
A_string which defines them. Strings declared to be KNOWNPOINTs are not under any string. A string which is a SEARCHPOINT may or may not be under another string or strings.

A string structure can be regarded as a directed graph and it is often convenient to use terminology from graph theory in discussing one. Most of the terms used in this dissertation are clearly defined in [Aho72]. Several terms not mentioned there are defined here. The *filial set*, F, of a node, N, of a directed graph is the set of all direct descendants of N. N is a *parent* of each node in F. Each member of F is a *child* of N and a *sibling* of every other node in F. For the remainder of this dissertation *graph* will be taken to mean *directed graph* unless explicitly stated otherwise.

Considering a string structure in graph-theoretic terms, KNOWNPOINTs are base nodes because, having no parents, their in-degree is zero. All other strings are interior nodes and AVCs are leaf nodes. Accessibility can now be easily defined for all SSEs. A SSE, B, is *accessible* from another SSE, A, if there is a path of length zero or greater from A to B.

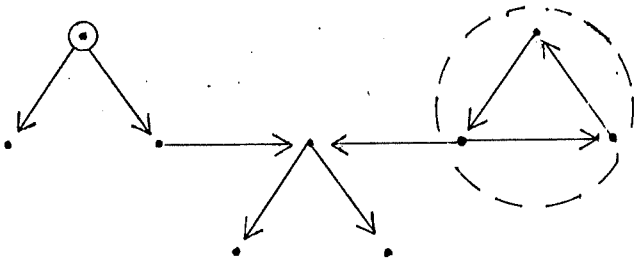
The minimal access constraint was earlier expressed as "every fact must have at least one instance in the database and every fact instance must be reachable by at least one access route". In the section discussing AVCs the first part of the constraint was rephrased as "every attribute must be covered". The second part can now be rephrased as "every AVC must be accessible from an entry point". The minimal access constraint thus becomes "every attribute must be covered by at least one AVC and every AVC must be accessible from at least one entry point".

In addition to satisfying the minimal access constraint, the further requirement of connectivity is demanded of a valid string structure. A string structure is termed *connected* if there is at least one entry point and every SSE which is not an entry point is accessible from at least one entry point. Figure 4.18 depicts two string structures represented as graphs. Both have one base node and in both all leaves are accessible from the base node. However, graph A is connected and graph B is not.



GRAPH A -- "connected"

⊙ base node



GRAPH B -- "unconnected"

Figure 4.18: String Structure Connectivity

Connectivity could be established in graph B if one of the nodes in the three node cycle encircled by a broken line were designated a SEARCHPOINT. The point of the connectivity requirement is that it guarantees the existence of an access route to every SSE in the structure.

4.4 The encoding level

The function of the encoding level is to define the bit-level representation of the fact instances and logical access paths defined at the encoding level. The encoded representations are thought of as being stored in one or more virtual address spaces, themselves physically located on the devices described by the hardware specification component of PRODD.

The abovementioned virtual address spaces are called *LASs* (linear address spaces). An LAS is an addressable stream of units of storage. Description of an LAS first specifies which of the basic storage units recognised by PRODD is to be the unit of space allocation within the LAS. PRODD recognises three basic storage units: the bit, the byte and the word. A byte is taken to be eight bits. The number of bits per word is included as part of the hardware specification.

LAS units of storage are grouped into pages which are the logical units of organisation for secondary storage accessing. The LAS description also specifies page size and whether deletion is logical or physical (space reclaimed immediately). This is an LAS parameter because different policies may be appropriate for different LASs depending on the update activity directed at the data stored in an individual LAS. An LAS description also gives percentage page fill at database loading time. Figure 4.19 illustrates the description of the one LAS for the PSA example. Here the LAS is being used to approximate the DBTG concept of AREA/REALM. The parameter values are from a typical EDMS database. A separate component of PRODD, the SDM, defines the allocation of LASs to physical devices.

```
LAS presidential area
  STORAGE_UNIT IS BYTE
  PAGE_SIZE IS 512 WORDS
  DELETION IS PHYSICAL
  PAGE_FILL IS 65%
END_LAS
```

Figure 4.19: An LAS for the PSA Example

4.4.1 The BEU: Fundamentals

The encoding of SSEs (string and AVCs) is defined by means of a single parameterised format, the *BEU* (basic encoding unit). There must be one BEU defined for every SSE declared at the access level with one exception. Link AVCs do not require an encoding because their values are not actually stored. It is assumed in PRODD that all instances of a single SSE are placed in one LAS. This assumption made the initial implementation of a prediction program easier but is not essential. It could be removed later if it should prove unduly restrictive. Thus the first part of a BEU is the LAS entry which names the LAS in which all instances of that SSE are to be placed.

Strings have been described as giving access to other strings or to AVCs. At the encoding level this means that an encoded string instance identifies the start of an ordered collection or sequence. In PRODD terminology this is the *value* of a string instance. The collection of SSE instances is called a *value sequence*. The value sequences of A and L_strings are made up of a fixed number of SSE instances, one for each SSE in their filial sets. C_string instances may have a variable number of child instances in their value sequences. The notion of value sequence can be extended to AVCs by defining the value of an AVC to be the start of a collection of storage units holding its value. This collection may be fixed or variable in number depending on whether the data value is

stored in a fixed or variable-length field.

So the second point to be described by each BEU is where the start of an SSE instance's value sequence is located. This is roughly equivalent to Severance's intra-record connection describing the location of the data portion of a record. By defining a *CDG* (contiguous data group) as a group of SSE instances always stored or fetched together three possibilities can be identified:

1. the value sequence may start in the same CDG (Severance's direct connection)
2. the value sequence may start in the next CDG in the same LAS
3. the value sequence may start in a randomly located CDG in any LAS (Severance's indirect connection).

The significance of the above classification is that:

1. means that no further accessing of secondary storage is required to access the SSE instance or storage unit which begins the value sequence
2. implies only a sequential access is needed
3. necessitates a time-consuming random access

In each one of these cases a pointer field might exist to indicate the exact location of the start of the value sequence. Therefore, in addition to specifying one of same, next or any CDG, it is possible to declare the existence of a value start field of fixed size.

In order to evaluate the performance of an operation request, it is necessary to have some information about the length of access routes. Hence the length of a value sequence, known as the *value length*, must be indicated. A value length may be fixed, in which case the value is supplied as part of the structural specification. On the other hand, it may be variable with an average value supplied by the content

specification component. It may be the case that a field exists to record the value length. Therefore, it is possible to declare a value length field of fixed size.

Figure 4.20 illustrates a partial BEU for the C_string <<headed>> The LAS and VALUE entries are shown. A VALUE entry has two sub-entries:

```
BEU FOR headed
  LAS IS presidential_area
  .
  .
  .
  VALUE START IS IN ANYCDG
              PTR IS FIELD OF 4 BYTES
        LENGTH IS VARIABLE
END_BEU
```

Figure 4.20 Partial BEU Declaration

a START sub-entry giving the location of the first element of the value sequence relative to its parent (SAMECDG, NEXTCDG or ANYCDG) and a LENGTH sub-entry describing the length of the value sequence.

The string structure for the PSA example demonstrated that a string might be under several other strings and thus a single string instance may be an element of several value sequences. Accordingly, there is one NEXT entry for each value sequence in which the SSE instance may appear. For each of these value sequences the associated NEXT entry specifies where the next element of that value sequence is located. Here is the approximate equivalent of Severance's inter-record connection which describes the location of the next record in a record collection. Normally there must be one NEXT entry for each parent of the SSE*.

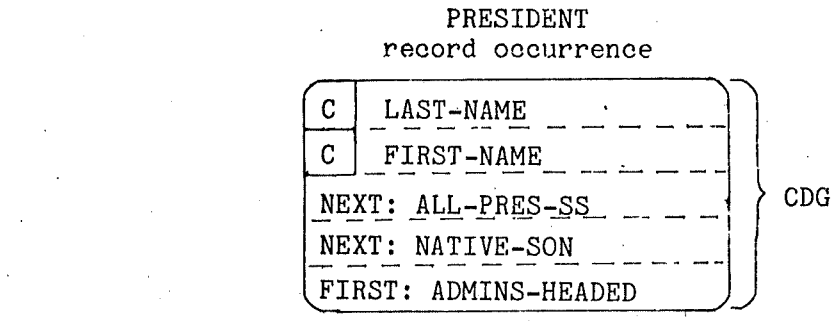
* It is optional in the case of the last stored element of an A or L_string, or in the case of the child of a C_string partitioned on an identifier of the child string because in both cases they will be the last or only element of their parent's value sequence.

A NEXT entry provides the same details as the VALUE START sub-entry.

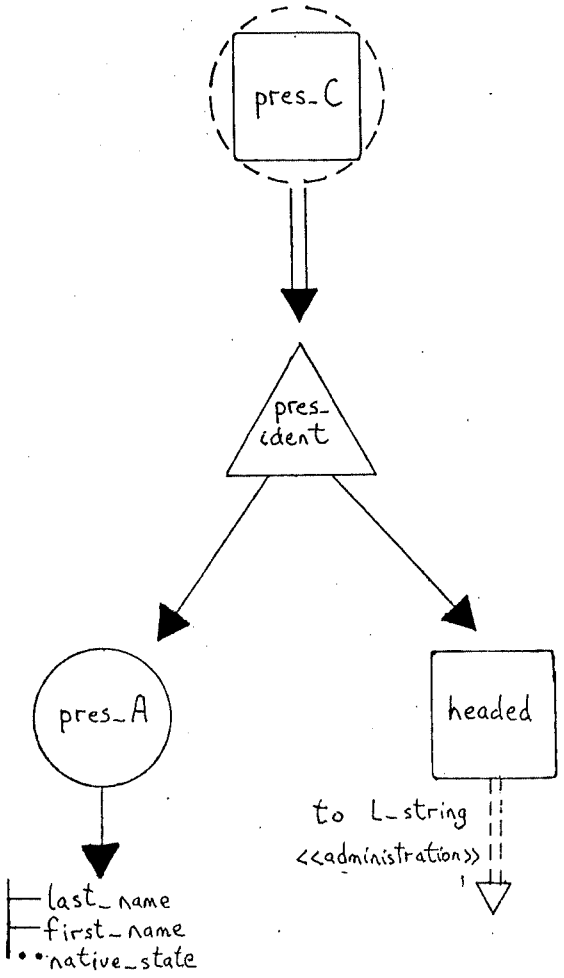
Figure 4.21 represents an approximate encoding-level description of the network PSA example. At the top is a schematic representation of a PRESIDENT record occurrence. It has fields for LAST-NAME and FIRST-NAME plus a NEXT pointer for each of the two cosets of which it is a member and a FIRST pointer for the set occurrence which it owns. On the left below is the section of the network string structure corresponding to the PRESIDENT record type. It has been isolated from Figure 4.17. Instances of the strings and AVCs shown on the left of Figure 4.21 make up a CDG corresponding to the PRESIDENT record occurrence.

Below right in Figure 4.21 is a simplified version of a formal encoding-level specification for the string structure diagrammed on the left. All SSE instances are assigned to one LAS, <<presidential_area>>, corresponding to the DBTG REALM PRESIDENTIAL-AREA. The BEU for <<pres_C>> is a special case because <<pres_C>> is a key-access C_string. Its VALUE START sub-entry must specify the mechanism to be used in transforming key values into addresses. At present PRODD supports three alternatives: hashing, indexing and binary search. In the case of <<pres_C>> the choice is hashing and the name of a hashing function is supplied. The software specification component of PRODD will provide details concerning the CPU cost of executing the hashing function.

The BEU for L_string <<president>> declares the start of its value sequence to be in the same CDG as the <<president>> instance with a length fixed by its access-level definition. This will be two instances, one for A_string <<pres_A>> and a second for C_string <<headed>>. It also has two NEXT entries which declare the next elements in the value sequences of C_strings <<all_pres_SS>> and <<sons>> to be in any CDG



STRING STRUCTURE



ENCODING

LAS for all BEUs is
presidential_area

BEU for pres_C
VALUE is found by hashing
using DBS_hash

BEU for president
VALUE is in same CDG with
length of 2 instances
NEXTs are in any CDG with
4 byte pointers

BEU for pres_A
VALUE & NEXT in same CDG,
length is 3 instances

BEU for headed
VALUE is in any CDG with
4 byte pointer

BEU for pres_A:last_name
VALUE & NEXT in same CDG,
length is 10 bytes

BEU for pres_A:first_name
VALUE is in same CDG with
length of 10 bytes

Figure 4.21: Encoding Level Description of PRESIDENT record

identified by a four byte pointer field. A_string <<pres_A>>'s BEU declares its VALUE START as being in the same CDG with a length again fixed by its access level definition, this time as the number of AVCs under <<pres_A>>. It has a NEXT entry for the value sequence of its one parent, L_string <<president>>, defining the next element (an instance of C_string <<headed>>) to be in the same CDG.

BEUs for the other SSEs are constructed in a similar manner. However, note that BEUs for C_string <<headed>> and AVC <<pres_A:first_name>> do not have a NEXT entry because they are the last encoded elements of L and A_strings respectively (link AVCs such as <<pres_A:native_state>> not being encoded). Note also that the VALUE LENGTHs for AVCs <<last_name>> and <<first_name>> are in terms of storage units, in this case bytes, instead of SSE instances as for strings.

4.4.2 Optional BEU entries

There is one BEU entry reserved for derived AVCs which must be the first entry when present. The DERIVATION entry declares one of three options for the computation and storage of a derived AVC's value. Derivation may be *immediate* meaning that whenever the value of an SSE instance on which derivation is based is changed the derived value is immediately recomputed. *Delayed* derivation means that recomputation of the derived value is delayed until it is accessed. In both cases the derived value is physically stored. *Virtual* derivation means that the derived value is recomputed upon each access and therefore is not actually stored between accesses.

It is well-known that one of the crucial factors affecting performance is clustering of related records on the same page so as to

reduce the number of secondary storage accesses required in processing them sequentially. Consequently, many DBMSs have a facility which enables a database designer to specify a placement strategy to exploit this kind of physical organisation. An example can be seen in the DBTG VIA location mode which is usually implemented so that the DBMS tries to store the members of a set occurrence near the owner record, preferably on the same page [Bake75]. Another example is found in System R [Astr76] where images (indices) and binary links (equivalent to DBTG cosets with manual membership) may be declared to have the clustering property.

PRODD provides for description of the use of such a facility by means of the CLUSTER entry of a BEU. Instances of a SSE may be declared to be clustered with respect to the value sequence of at most one of the SSE's parents. That value sequence must be connected by pointers since it does not make sense to refer to clustering of SSE instances already related by contiguity. In the PSA example the BEU for L_string <<administration>> would declare clustering under C_string <<headed>> to model the fact that the ADMINISTRATION record has LOCATION MODE VIA the coset ADMINS-HEADED.

There are several other BEU entries which enable a more exact modelling of the encoding of access structure in a particular DBS. The first is the notion of labelling. A BEU for a SSE may declare labelling, meaning that every instance will be prefixed by a label field identifying its SSE type*. The LABEL entry declares the size of the label field. If an LAS is to be processed sequentially then at least some of the SSE instances will have to be labelled so that the stream of storage units can be correctly interpreted as distinct SSE instances. Declaring labelling is also a convenient mechanism for representing the existence of other

* Labelling is compulsory for SEARCHPOINT strings.

DBS control information. Having mentioned labelling, the average size of an SSE instance can now be defined as the sum of the size of the LABEL field, the VALUE START and VALUE LENGTH fields, and any NEXT pointer fields. If any of these fields have not been specified their size will be reckoned as zero for this calculation.

In order to aid the PRODD translator in identifying the membership of CDGs and in carrying out various semantic checks, the concept of a CDGHEAD has been introduced. An SSE is a CDGHEAD if its instances are first in (at the head of) the CDGs in which they are placed. All CDGHEADs must be declared as such by their BEUs.

There is an optional entry for describing alignment to permit precise estimation of space usage. Each LAS declares the basic storage unit used in defining encoding and space allocation within that LAS. All BEUs for SSEs assigned to that LAS must specify field sizes in terms of that storage unit or multiples of it. By default an SSE instance is assumed to be aligned on the basic storage unit of the LAS to which it is assigned. The ALIGN entry permits the specification of alignment on some larger storage unit. Looking at Figure 4.21 the storage unit for LAS <<presidential_area>> was declared to be the byte and all SSE instances except those of L_string <<president>> would be considered to be aligned on byte boundaries by default. The BEU for <<president>> however, declares word alignment because in many DBMSs, EDMS in particular, record occurrences are constrained to begin and end on word boundaries.

The use of compression is described by another optional entry. A sub-sequence of the value sequence of an A_string can be designated as a compression sequence. One AVC is nominated as the COMPRESSIONHEAD. Its BEU names the compression and decompression functions and defines the

length of the compressed sequence in the standard way. All other AVCs in the compression sequence specify compression under the COMPRESSIONHEAD in lieu of the usual NEXT entry. Their VALUE entries must specify VALUE START in the same CDG while VALUE LENGTH is used to indicate the uncompressed length of their value sequences. When route costing is being carried out, the entire compression sequence is considered decompressed when the COMPRESSIONHEAD is reached and so other AVCs in the compression sequence can be visited without further cost.

As a final example, Figure 4.22 presents a complete BEU for the L_string <<president>>.

```
BEU FOR president
  LAS IS presidential_area
  ALIGN ON WORD
  LABEL IS FIELD OF 3 BYTES

  NEXT UNDER all_pres_SS
       IS IN ANYCDG
       PTR IS FIELD OF 4 BYTES

  NEXT UNDER sons
       IS IN ANYCDG
       PTR IS FIELD OF 4 BYTES

  VALUE
    START IS IN SAMECDG
    LENGTH IS FIXED BY DEFN
           IN FIELD OF 1 BYTE
END_BEU
```

Figure 4.22: BEU for L_string <<president>>

Table 4.1 summarises the various BEU entries and their functions. They appear in the same order as in an actual BEU declaration.

Three primitive constructs for describing encoding structure have now been introduced:

entry name	optional/ compulsory	function
DERIVATION	only for derived AVCs	required and allowed only for derived AVCs: specifies whether derivation is immediate, delayed or virtual
LAS	compulsory	names LAS in which SSE instances are to be placed
CDGHEAD	optional	declares this BEU as a CDGHEAD
ALIGN	optional	declares unit of storage on which SSE instances are to be aligned
LABEL	optional	declares a label field of fixed size
COMPRESSION HEAD	optional	declares this BEU as the head of a compression sequence
COMPRESSED UNDER sse	optional	declares this BEU to be in the compression sequence headed by sse
NEXT UNDER parent	one for each parent SSE	specifies location of next element in parent's value sequence; optionally may specify a fixed-size pointer field
CLUSTER UNDER parent	optional, one parent only	specifies SSE instances are stored "near" the previous element in the parent's value sequence
VALUE START	compulsory	specifies location of first element of SSE's value sequence; optionally may specify a fixed-size pointer field
VALUE LENGTH	compulsory	specifies length of SSE value sequence; optionally may specify a fixed-size length field

Table 4.1: BEU Entries

- the LAS
- the BEU
- the CDG

A formal PRODD encoding-level specification consists of the declaration of all LASs in which SSE instances can be placed followed by one BEU for

each SSE defined at the encoding level (excepting link AVCs).

4.4.3 Factoring of control information

In the encoding-level examples presented earlier certain SSEs had constants for the values of some of their BEU entries. Examples have included VALUE START in same CDG or VALUE LENGTH fixed by definition. In such cases it is usual to record the information in some kind of system catalogue rather than in the data stream. This is called *factoring*. The more regular the access structure, the more factoring is possible. In cases of great structural regularity, such as a sequential file of fixed-length records, all the structural connections are represented by contiguity and hence are factored. Only the actual data values appear in the data stream. In general, the amount of storage consumed by control information can be estimated by noting that only the value sequences of AVCs represent pure data; all storage devoted to LABEL fields, NEXT and VALUE pointer fields and LENGTH fields represents the access structure governing access to that data.

4.5 The relationship between DIAM and PRODD

The PRODD information level is a merging of concepts from the DIAM entity set level, Codd's relational model [Codd70] and Mealey's ideas about the nature of data [Meal67]. The access level is a rationalised and more powerful version of the DIAM string level based on the analysis of file organisation methods in the previous chapter. The notions of AVCs and the possibility of derived data values, entry points and access routes, the minimal access constraint and string structure connectivity

were introduced by the author, as was the terminology of over, under and covering used in describing access structure. The PRODD C_string only bears a limited resemblance to the DIAM E_string which is why it was renamed. PRODD A_strings and L_strings are more rigorously defined than their DIAM counterparts. The PRODD encoding level only superficially resembles the DIAM encoding level which has been systematically overhauled following the analysis of the details necessary for analytic performance prediction presented in chapter 3. DIAM's encoding level seems primarily designed to allow a precise description of storage structure down to fine details such as the bit pattern stored in a particular field of the last record in a chain to mark its end. Thus, although the LAS, BEU and CDG have kept their DIAM names because their broad conceptual functions are the same, the way in which they are used is quite different. Finally, a complete language and formal syntax for describing database structures with PRODD have been developed, something which was never done for DIAM.

This chapter has demonstrated how database structure can be described with considerable precision through the use of a small number of general and powerful primitive constructs. It has been shown in detail how database structure defines what facts exist in a database and how their representations are to be accessed and encoded. The next chapter, chpater 5, discusses the four components of PRODD which provide the details necessary to cost operation request performance. Chapter 6 then explains how operation requests are actually described in PRODD and how that description is used to estimate database access time.

5. CONTEXT SPECIFICATION

Structural specification was earlier likened to providing a map of a database. Such a map is adequate in an abstract algorithmic sense; it defines whether or not a particular type of fact can be found in the database and how to go about accessing instances of it. However, it is like a map without scale, one on which all cities are represented as being equidistant and of equal size and all roads as being of the same standard. Before total storage requirements can be estimated and the travelling of access routes costed, it will be necessary to place a description of database structure within the context of a particular DBS implementation. There are four components of PRODD used in context placement:

- content specification
- software specification
- hardware specification
- SDM (Structure-Device Mapping) specification

These will be discussed in turn.

5.1 Content specification

As mentioned in chapter 3, database content is specified separately from structure in order to allow for changes in the number of facts stored in a database. It is possible to imagine a completely static database in which no update activity occurs after its initial loading, but this would seem to be the exception rather than the rule. The contents of a static database could be specified in the course of structural description by declaring all value sequences to be of fixed length and providing a constant or average length as appropriate. The static

case is thus easily dealt with and will not be further discussed.

PRODD provides for the description of dynamic database content by allowing variable-length value sequences to be specified for C_strings and AVCs. Supplying an average length for every value sequence of variable length has the effect of fixing database content at some instant in time. Depending on the database's volatility this will be a reasonable approximation to reality for an interval of greater or lesser duration. If volatility is high, it will be short. If volatility is low, it will be relatively long.

These ideas suggest that it would be desirable to specify a database's contents at a number of different times so as to be able to evaluate the effect of database growth on performance and storage requirements. This is the approach adopted in PRODD. Database content is declared to be described at a number of points in a database's lifetime known as *evaluation points*.

The first evaluation point is conventionally at the time of initial bulk loading of the database. It is called the *loading point*. Other evaluation points are numbered 1,2,.... The lengths of value sequences are defined in terms of the average number of collected instances per equivalence class in the case of C_strings and the average number of units of storage per data value in the case of AVCs. The values at the loading point are known as *loading values*. Subsequent evaluation points may be defined in the same manner or, alternatively, in terms of percentage increases/decreases from previous evaluation points, whichever is more convenient. If percentage changes are used they may be specified as being with respect to the initial loading values or to those computed for the previous evaluation point.

A partial demonstration of content specification for the PSA example should help to clarify these ideas. If it is imagined that the database became operational just after the 1956 presidential elections, then at loading time there would have been 50 states, 50 administrations and 33 presidents (Eisenhower was the 33rd president and his 1956-59 term of office the 50th administration). Now suppose evaluation points are fixed at just after the 1976 and 1996 elections. In 1976 Carter became the 38th president and his 1976 administration the 57th. Assuming a similar increase in the number of presidents and administrations over the following twenty years, by 1996 there will have been 43 presidents and 64 administrations. The number of states can reasonably be assumed to remain constant. Table 5.1 presents the above figures in tabular form.

year	number of presidents	number of states	number of administrations
1956	33	50	50
1976	38	50	57
1996*	43	50	64
% change per 20 yrs. (1956 base)	+15.15%	+0.0%	+14.0%

* 1996 figures estimated assuming same rate of growth as 1956 - 1976.

Table 5.1: PSA Occurrence Figures

Figure 5.1 illustrates the content specification for strings <<all_pres_SS>>, <<headed>> and <<sons>>, all of which have variable-length value sequences. First the number of evaluation points after loading is indicated. Next is the specification for <<all_pres_SS*>

```
CONTENT_SPECIFICATION    PSA_1965_to_1996

  EVALUATION_POINT OCCURS 2 TIMES

  VALUE LENGTH

      .
      .
      .
    FOR all_pres_SS
      AT LOADING IS 33.0 INSTANCES
      AT EVALUATION_POINT 1 IS + 15.15% FROM LOADING
                          2 IS + 30.30% FROM LOADING
    END_FOR

    FOR headed
      AT LOADING IS 1.52 INSTANCES
      AT EVALUATION_POINT 1 IS 1.5  INSTANCES
                          2 IS 1.49 INSTANCES
    END_FOR

    FOR sons
      AT LOADING IS .66 INSTANCES
      AT EVALUATION_POINT 1 IS + 15.15% FROM LOADING
                          2 IS + 30.30% FROM LOADING
    END_FOR
      .
      .
      .
```

Figure 5.1: Content Specification

which is the simplest. From 1956 to 1976 there was an increase of 5 in the number of presidents or a change of +15.15%. Under the assumption of an equal increase over the following 20 years the value of evaluation point 2 (1996) will be up 30.30% from the initial loading value.

The calculations for <<headed>> and <<sons>> are slightly more complex. At loading time there were 33 presidents and 50 adminsitrations. Therefore, on average there were 1.52 administrations per president. At evaluation point 1 this has fallen to 1.5 and to 1.49 by evaluation point 2. The figures for <<sons>> depends on the ratio of presidents to states. At loading time there were 33 presidents and 50 states and thus an average of 0.66 presidents per state. Since the number of states is constant, the percentage change in value lengths for instances

of the C_string <<sons>> will be exactly proportional to that for C_string <<all_pres_SS>>. Therefore, evaluation points 1 and 2 can be fixed without further calculation at increases of 15.15% and 30.30% from loading respectively.

Once value sequence lengths have been specified for all strings, either as fixed at the encoding level or as variable with a set of values provided by content specification, database content is almost fully described. One small loophole remains, however. This arises in some cases concerning SEARCHPOINT strings.

Generally, all SSEs are either known to be under other strings or to be 1-instance KNOWNPOINTS. In the first case, the number of SSE instances can be estimated from the information about the length of their parent's value sequences. In the second, the number of instances is known to be one by definition. However, for a string which is a SEARCHPOINT and which has no parents neither of these considerations apply. Consequently, there is a second optional section of content specification which is used to indicate the number of instances of such strings at each evaluation point. The form for doing so is exactly the same as for specifying value lengths. Figure 5.2 presents a fragment illustrating this for L_string <<president>>. Strictly speaking this would be

```
SEARCHPOINT  COUNT

  FOR president
    AT LOADING IS 33.0 INSTANCES
    AT EVALUATION_POINT 1 IS + 15.15% FROM LOADING
                        2 IS + 30.30% FROM LOADING

  END_FOR
    .
    .
    .
```

Figure 5.2: Searchpoint Counts

unnecessary since the number of instances of <<president>> can be estimated from the length of the value sequence for C_string <<all_pres_SS>> which collects all instances of L_string <<president>>. However, it is included for the sake of illustration.

A complete content specification thus guarantees two things: a value is known for the (average) length of every SSE instance's value sequence and for the number of instances of every entry point. Using these figures, an estimate of the number of instances of every SSE at each evaluation point can be computed. Once this stage has been reached, it is possible to estimate the volume of SSE instances since their average size will be known from the structural specification.

There is one minor complication remaining before LAS volumes can be computed. If logical deletion is specified for an LAS, an estimate of dead space at each evaluation point must be supplied. Since a BEU assigns all instances of an SSE to one LAS, the volume of live data in each LAS can be estimated. The number of pages which must be allocated to achieve the specified initial page fill factor can also be calculated. The volume figures are computed at every evaluation point (taking dead space into account if appropriate) and then printed for each LAS. A warning can be issued if projected volume at any evaluation point would overflow the initial page allocation determined by the loading volume and initial page fill factor. Page fill factors can be calculated for all other evaluation points and used to estimate page overflow probabilities.

Implementation of the PRODD model in a prediction program revealed an oversight in the scheme described in the preceding paragraphs. It becameclear that it was necessary to distinguish between two values for the "average length of a value sequence", these being 1) an unweighted

average taken over all instances of the string in the database, and 2) a weighted average reflecting the frequency of accesses to the different instances of a string. If all instances of a string are accessed with equal probability then these two are the same. This is not always the case,as is described in more detail in chapter 7 where experience with using the prediction program is discussed. If a difference does exist then 1) must be used in estimating storage requirements and 2) in costing access routes. The content specification component was accordingly extended to allow specification of both 1) and 2).

5.2 Software specification

The purpose of software specification is to give approximate CPU costs for some of the basic functions provided by the software with which a DBS has been implemented. These functions are introduced in a number of places at the access and encoding levels:

1. C_string selection may be defined as the result of a procedure.
2. C_string partitioning may be defined as the result of a function.
3. the value of a link or derived AVC may be defined as the result of a function.
4. if the method chosen to support key access is hashing, then the name of a hashing function must be supplied.
5. if compression is specified then the names of compression and decompression procedures must be supplied.

The role of software specification is to supply approximate timings for the procedures which have been named as performing those functions.

A procedure timing is expressed in terms of two components: a

fixed overhead for every procedure call and a variable overhead depending on the combined length in bytes of all arguments of the procedure call. Either, but not both, of these components may be zero. This formulation has been found satisfactory for describing procedures such as hashing or compression routines; it is certainly not claimed to be suitable for characterising the execution time of procedures in general.

Figure 5.3 illustrates the specification of a procedure timing for the function DBS_hash introduced in Chapter 4. DBS_hash was declared as the hashing function used in finding the start of value sequences for

```
SOFTWARE_SPECIFICATION  EDMS_functions

  PROCEDURE DBS_hash
    TIMING IS 129.2 uSEC
            +  16.0 uSEC PER BYTE
  END_PROCEDURE
    .
    .
    .
```

Figure 5.3: Procedure Timing

C_string <<pres_C>>, <<pres_C>> having been declared to model location mode CALC for the PRESIDENT record type. The figures used here were obtained by hand-timing of the assembly code for the standard hashing function of EDMS [Xero73].

5.3 Hardware specification

The PRODD component for hardware specification characterises the CPU and storage devices on which a DBS runs. Limited particulars of CPU performance and architecture are supplied for use in estimating CPU overheads. Details of device performance and capacity define average secondary storage access times under various conditions.

5.3.1 CPU description

A CPU is described in a simple fashion by four parameters. These are:

- the number of bits per byte
- the number of bits per word
- the time to perform a one byte comparison (operands in store)
- the time to copy one byte from one store location to another

Byte size and word size are used to calculate storage requirements expressed in bytes. Byte comparison and copying times are used in calculating CPU overheads for operations such as index searching, binary search and transferring data between application program and DBS buffers.

Figure 5.4 illustrates the description of the Xerox Sigma 6 CPU [Xero71]. The first line of text is the declaration of a hardware

```
HARDWARE_SPECIFICATION  Xerox_Sigma6_configuration

  CPU
    BYTE IS  8 BITS
    WORD IS 32 BITS
    COMPARISON IS 8.5 uSEC PER BYTE
    COPYING    IS 7.4 uSEC PER BYTE
  END_CPU
```

Figure 5.4: CPU Description

specification component, the first element of which is a CPU description.

5.3.2 The transducer-surface model

The description of storage devices in PRODD is based on an abstract view of such devices called the *transducer-surface model*. This model has been formulated with two objectives in mind: 1) to capture only those details relevant to first-order performance prediction and 2) to embrace as wide a range of secondary storage devices as possible.

The transducer-surface model pictures a device as one read/write transducer operating over a single surface of some storage medium. The surface is divided into strips of width determined by the resolution characteristics of the transducer. Operation of the device is imagined to proceed as follows. The transducer can be positioned over any strip and then a known point on that strip brought into position under the transducer. After positioning, data can be read from or written to a section of the strip. By convention the transducer is considered to move across strips and the surface is considered to move under the transducer once it has been positioned over a particular strip. However, as far as the actual device being described is concerned, only one of the two may move or there may be no macroscopic movement at all. Figure 5.5 shows a schematic representation of the idealised device.

Secondary storage accessing time can now be seen to have three components:

- transducer positioning time or *seek time*
- surface positioning time or *latency time*
- data read/write time or *transfer time*

Seek time and latency time together make up *positioning time*, the delay incurred before data transfer can begin.

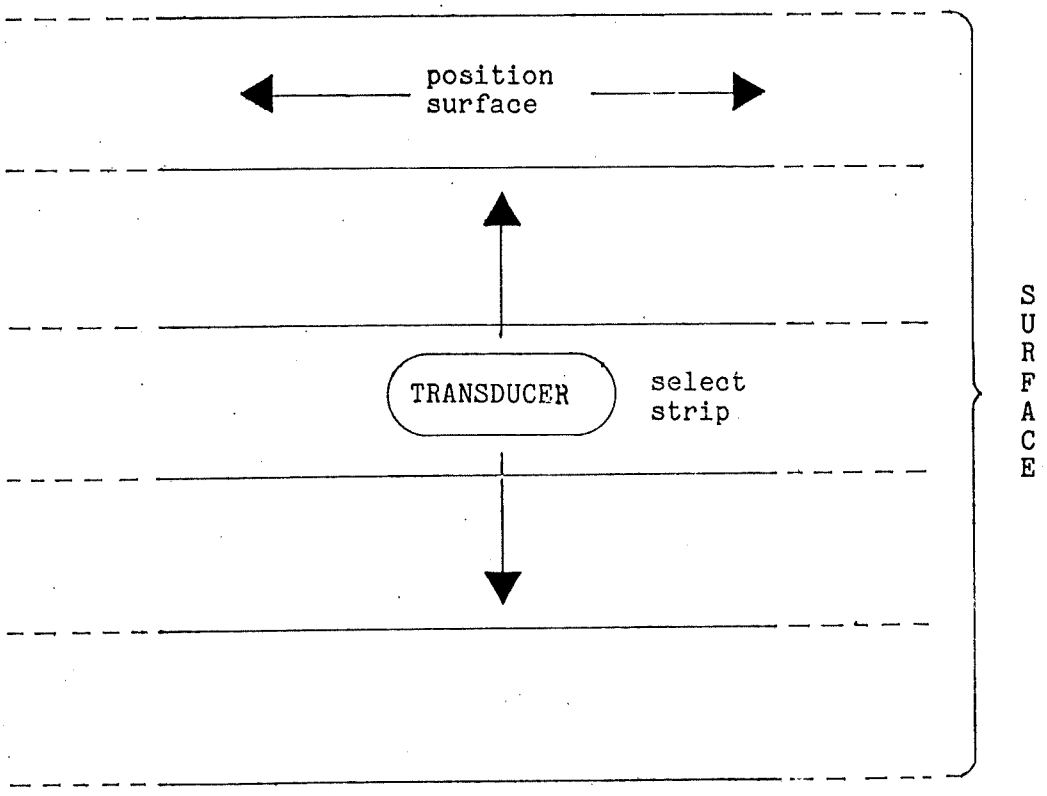


Figure 5.5: The Transducer-Surface Model

Two groupings of units of secondary storage follow naturally from the above model. The first is the *block*, here defined to be the minimum number of units of storage that can be read or written in one physical operation. The second is the *track*, defined as the maximum number of units of storage accessible once the transducer has been positioned over a certain strip. A track is usually equivalent to one strip. In this model a track always comprises an integral number of blocks and a device an integral number of tracks.

As mentioned in chapter 1, database accesses conceptually fall into two categories: associative and positional. An associative access is one in which an entity description is located through supplying a

value for one of its identifiers whereas a positional access addresses the next description in some sequence. At the hardware level this division is continued with two modes of physical access: random and sequential. *Random access* is taken to mean the accessing of a block of known address starting from a random positioning of transducer and surface. *Sequential access* is understood to mean the accessing of the next block relative to the current positioning of transducer and surface.

The above classification suggests that there are two types of positioning, random and sequential. A random positioning is considered to consist of a random seek and a random latency delay. For the purposes of modelling, the time for those two operations will be represented by the average random seek and latency times respectively. Sequential positioning will normally incur one sequential latency delay. However, in the case where the next block is in a different track, it is considered to consist of a sequential seek plus a random latency delay. This convention was suggested by Yao [Yao77a]. It is based on the assumption that one sequential seek is made to position the transducer over the next track followed by one random latency delay in positioning the first block of the track under the transducer. Thus, the other two relevant parameters for estimating positioning time are the average sequential seek and latency times.

Transfer time is determined by a combination of the time required to read or write one block, known as *block transfer time*, and the seek and latency times for the device concerned. Table 5.2 defines the calculation of transfer time for a contiguous sequence of b blocks where b is greater than or equal to 1 and less than the total number of blocks on the device.

b=1, i.e. the amount of data to be transferred is <= one block	transfer time is: 1*(block transfer time)
b>1, but all blocks are in the same track	transfer time is: b*(block transfer time) + (b-1)*(sequential latency time)
b>1 and the b blocks lie in T>1 tracks	transfer time is: b*(block transfer time) + (T-1)*(sequential seek time) + (T-1)*(random latency time) + (b-T)*(sequential latency time)

Table 5.2: Transfer Times for Block Sequences

It has been found useful to introduce a second category of sequential latency delay in connection with devices having substantial latency times. Consider the problem of estimating secondary storage access time for the retrieval of a record stored by hashing with linear probing. This will consist of one primary access to the home bucket and a number (possibly zero) of secondary accesses to overflow buckets. The primary access can be modelled accurately enough as random. However, if latency time is large, CPU time to search a bucket is small and the device is a rotating one, then the latency delay between secondary accesses will be much closer to the maximum than the average. With a moving-head disc this can mean a difference of as much as 10-15 msec. which is too much to ignore. Accordingly, a second type of latency delay is defined, known as *secondary latency*, to deal with the above class of accesses.

When a device is being used for random accessing of relatively small amounts of data (relative to total device capacity) seek time becomes the dominant component of access time. Attempts to reduce seek time overhead can be observed in two common storage devices: moving-head and fixed-head discs. In the terminology of the transducer-surface

model a moving-head disc has multiple surfaces with one transducer per surface. The transducers move together so that they are always positioned over the same strip of their respective surfaces. The advantage of this arrangement is that the number of units of storage accessible from one transducer positioning has now been increased by a factor of m, m being the number of surfaces. Consequently, the seek time overhead can be shared among a larger volume of data than was the case with only one surface. In terms of the transducer-surface model, track size has been increased m-fold without paying any penalty in increased random latency time. A more expensive solution with higher performance is represented by the fixed-head disc. There one transducer is allocated per strip thereby eliminating seek time altogether.

A wide range of devices may be approximately described with the transducer-surface model by:

1. allowing zero values for seek time and/or latency time
2. allowing a track to contain exactly one block with the possibility that blocks may be only one unit of storage.

Table 5.3 illustrates the classification of a number of classes of storage devices in terms of the model.

device	no. of transducers	no. of tracks	no. of surfaces	seek time	latency time
tape drive	1	1	1	no	yes
moving-head disc	m	n	m	yes	yes
fixed-head disc or drum	n	n	1	no	yes
bubble memory (major/minor loop)	1	n	1	no	yes
EBAM (block addressed)	1	n	1	yes	no
RAM	word size	1	1	no	no

Table 5.3: Storage Device Classification

5.3.3 Device type description

Hardware specification does not identify the individual devices present in the secondary storage configuration. A description of each type of device in terms of the transducer-surface model and a declaration of how many devices of each type are available online was judged adequate for the purposes of first-order prediction. It also simplifies the preparation of a complete DBS description.

As an example of the application of the transducer-surface model to device description, Figure 5.6 shows the specification for the Xerox RXD-71 disc unit. The RXD-71 has 400 cylinders of 20 surfaces, with a surface consisting of tracks of 3 blocks of 512 words (2048 bytes)

```
DEVICE_TYPE  RXD_71
  OCCURS 3 TIMES

  DEVICE IS 400 TRACKS
  TRACK  IS  60 BLOCKS        || a cylinder
  BLOCK  IS 512 WORDS

  BLOCK_TRANSFER_TIME IS 8.3 mSEC

  SEEK TIME
    SEQUENTIAL IS  7.5 mSEC
    RANDOM     IS 29.0 mSEC

  LATENCY TIME
    SEQUENTIAL IS  0.0 mSEC
    SECONDARY  IS 25.0 mSEC
    RANDOM     IS 12.5 mSEC

END_DEVICE
```

Figure 5.6: The RXD-71 Moving-Head Disc

each. It rotates at 2400 rpm. Notice that track size has been set to cylinder size since all blocks in one cylinder can be accessed without head movement. The phrase "OCCURS 3 TIMES" indicates that the online configuration includes three devices of this type.

5.4 SDM specification

After the hardware configuration has been described, it remains to define the mapping of LASs to the devices of that configuration. This mapping serves to bind LASs to actual devices so that real-time estimates of secondary storage access times can be made.

The actual mapping specified by a PRODD SDM component is not very detailed. It does not go beyond indicating which type of device a particular LAS is stored on. More detailed mappings in which an actual address range on a particular device is specified can be imagined, as in Senko et. al.'s Physical Device Level [Senk73], but were judged unnecessary for the goals of the PRODD model. At the level of detail chosen for PRODD, the structure-device mapping might be more accurately termed a *structure-device type* mapping.

Figure 5.7 is a PRODD fragment declaring the SDM component for the PSA example. This consists of just one entry because the example makes use of only one LAS. In general there would be one entry for every LAS

```
SDM_SPECIFICATION  PSA_realm
  LAS presidential_area IS ON DEVICE_TYPE RXD_71
END_SPECIFICATION
```

Figure 5.7: Structure-Device Mapping

declared at the encoding level.

Using the figures generated for maximum LAS size, the total storage requirement for a device type can be calculated at each evaluation point and a warning given if it exceeds that available. The available online

storage capacity for each type is taken to be the product of the number of devices of that type times the individual capacity; both these figures are provided by the hardware specification component.

6. ACTIVITY SPECIFICATION AND EVALUATION

The components of PRODD discussed in the last two chapters serve to characterise a DBS in a passive sense. They describe how it is put together, how big it is, what devices it is stored on and how long it takes to perform certain basic functions. But they do not provide any information about how it is used. This is the role of activity specification and is discussed in the first two sections of this chapter. The third section explains how the description of database usage provided by activity specification is processed to yield estimates of database access time.

6.1 The activity set

In chapter 3 a "black box" view of a DBS (c.f. Figure 3.1) was introduced in which a DBS was seen as responding to four classes of operation request, namely, retrieval, modification, insertion and deletion. A natural consequence of this view is to describe the usage of a DBS in terms of a representative set of operation requests called an *activity set*.

The first problem to be faced is how to determine the activity set. Chapter 3 proposed a classification of operation requests as either routine or non-routine. Routine operation requests clearly ought to be included in the activity set. It remains only to choose a suitable way of indicating their relative significance as components of DBS activity. The solution adopted with PRODD is to assign a weighting to each operation request in the activity set. The sum of the weightings should be one. Normally the weighting would be the number of times a particular request occurred within a representative period

of time expressed as a fraction of the total number of operation request occurrences within that period.

Non-routine operation requests present a more difficult problem. It is possible to devise schemes for their generation based on statistical descriptions of the frequency with which requests of the four different classes occur and the likelihood of accessing different types of facts. A strategy of this kind was used for query generation in the Martin Marietta simulator described in Chapter 2. Since the research described here is aimed at DBSs in which routine operation requests predominate, it will be assumed that the contribution of non-routine ones to overall activity is minimal and hence can be ignored for practical purposes. Accordingly, at this time no method of including them in the activity set has been implemented.

6.2 Specification of routine operation requests

Activity specification has now been reduced to the problem of describing an activity set made up of routine operation requests. The idea of weighting operation requests in accordance with their frequency of occurrence has already been introduced. The next requirement is a method for specifying individual operation requests. The one described here has been designed with the primary objective of making it easy to get an experimental prediction program off the ground.

A major simplification was not to include any path selection or route finding capability in the program. Instead the specification for each operation request fully defines the elements of the access structure which must be visited to carry out the operation request. This decision is further justified by the fact that the method used for

path selection various from DBS to DBS and is generally a complex algorithm. Therefore, the only practicable way of describing path selection in a general fashion would be procedural modelling, something which the PRODD model explicitly seeks to avoid.

As mentioned in chapter 3, the performance of an operation request can be thought of as following a route through the database. The route begins at some known starting point and leads to the representations of the relevant facts. The cost of following this route in terms of database access time has been adopted as the metric of DBS performance. Thus a natural and convenient way to specify an operation request is to describe the route followed in its execution.

6.2.1 Route following and pre-order visiting

A route must begin at some known starting point, i.e. one of the entry points of the access structure. From there one or more of its children may be visited. Looking at the database at the access level, this means that the path from an entry point string may be followed to one or more of its child SSEs. In turn any of their children may be visited and so on recursively.

Route following can be described as partial pre-order visiting of a section of the string structure graph. *Pre-order visiting* of an ordered graph may be recursively defined in a manner analogous to the definition of pre-order traversal of trees. For each node visited, process the node itself and then visit each child in turn. *Partial pre-order visiting* means that not every child of a visited node must be visited but only a possibly empty sub-sequence of the filial sequence. Every non-empty sub-sequence must start with the first child.

For the ordered graph shown in Figure 6.1 a pre-order visit beginning at entry point N1 would pass through nodes

N1, N2, N3, N4, N5, N6

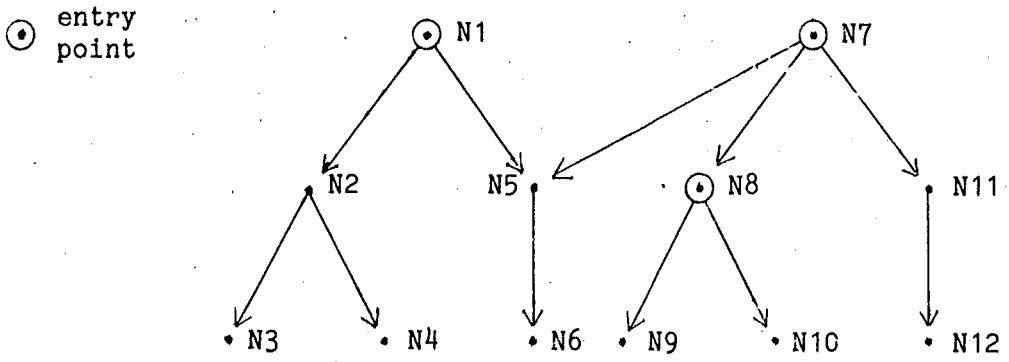


Figure 6.1: Pre-order Visiting of an Ordered Graph

whereas one of the partial pre-order visits possible from node N1 would be

N1, N2, N3, N5

Route following is now defined as partial pre-order visiting beginning at an entry point of the string structure graph.

In actual fact it is SSE instances rather than SSEs which are being visited. This consideration does not lead to any complication with regard to the children of A or L_strings because an A or L_string instance has a value sequence made up of exactly one instance of each child SSE in the string's filial set. Therefore, visiting a child of an A or L_string corresponds to visiting the one instance of that child in the value sequence of a parent string instance. However, when visiting the child instances of a C_string instance, there are a number of instances of the collected string in the value sequence. Consequently,

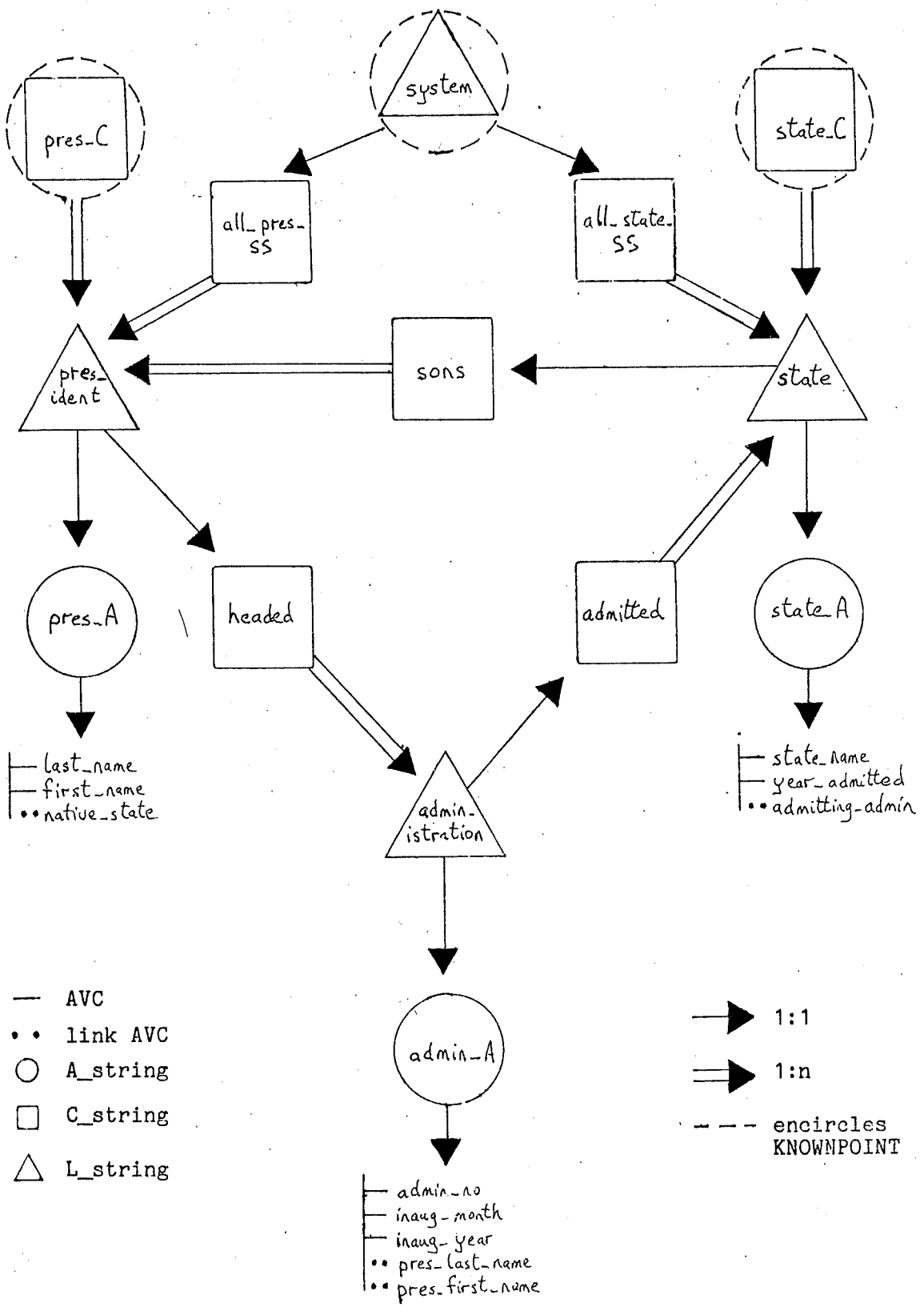


Figure 6.2: Complete Access Level for the PSA Example

an indication as to how many of these are actually visited must be supplied.

In describing pre-order visiting, some processing of each node visited was mentioned. This referred to the examination of the encoded representation of the SSE instance in order to find its value sequence. Selected SSE instances along the route may also be marked as having additional operations performed on them. These operations are comparison, retrieval, modification, deletion or insertion.

6.2.2 Simple retrieval

As an aid to following the description of access routes with PRODD, the diagram of the complete access level for the PSA example (Figure 4.17) has been reproduced here as Figure 6.2.

Figure 6.3 shows the PRODD description of the access route for the operation request "retrieve year admitted for a given state"*. The route begins at the entry point <<state_C>> which defines primary key-access to instances of L_string <<state>>. After accessing the correct instance of <<state>> , its value sequence is followed to A_string <<state_A>> in turn giving access to the AVCs <<state_name>> and <<year_admitted>>. The appearance of '/R' after <<year_admitted>> marks it as being retrieved. The symbol '=|' indicates the end of the route.

* In terms of DBTG DML, a simple procedure to do this would be to FIND the STATE record occurrence for the specified state using the CALC key STATE-NAME and then GET the STATE record occurrence. The value of the YEAR-ADMITTED data item would then be available in the UWA (user work area).

```
ENTRY AT
state_C
=> SELECT EXACTLY 1.0 OF
   state
   => state_A
      => ( state_A:state_name,
           state_A:year_admitted/R
         )
=|
```

Figure 6.3: A Route for Simple Retrieval

6.2.3 The "selection-entry"

When visiting the child instances of a C_string it is necessary to indicate how many of the usually variable number of instances are actually to be visited. A similar consideration applies when a route begins with a SEARCHPOINT since it is possible that more than one of the SEARCHPOINT string instances may satisfy the matching condition on their accessible AVC values. The construct for supplying this figure is called a "selection-entry". It has a number of forms which will be discussed in the remainder of this sub-section.

The first classification of these forms is on the basis of whether the child instances to be visited are identified by position or by content. The first group of forms which describe selection by position includes selection of the first, selection of the last or selection of all child instances. When the last child instance is selected it is assumed that all other child instances must be passed through in following the next sibling connections leading to the last instance. This leads to a division of visited instances into *selected instances* which are of real interest and *scanned instances* which are examined in the course of reaching the selected instances. Another possibility included in the first group is that in which the

locations of the instances to be visited are known beforehand. This is known as *exact selection* because only selected instances are visited, no scanning being necessary. An example of exact selection appears in Figure 6.3 where following <<state_C>> is assumed to provide the address of exactly one instance of <<state>> because <<state_C>> is declared to provide primary key-access on state name.

The second group of forms comprises those in which the target instances are selected on the basis of content, i.e. on the basis of their content or the content of instances to which they provide access. There are two possibilities within this second group, one for the cases in which the number of instances to be selected is known beforehand and a second for those in which it is only known after all of the child instances have been visited. These are known as *a priori* and *a posteriori* respectively. The reason for distinguishing between them is that in the a priori case, visiting of the child instances can terminate as soon as the a priori known number of selected instances are found. Otherwise, all child instances will have to be visited, that being the only way of ensuring that all child instances satisfying the selection criterion have been visited.

An example should make the distinction clear. Suppose that it is the case that one of the collected instances will be selected on the basis of its content. If this is known before the collection is searched, i.e. a priori (because the selection criterion is matching on a primary key or something similar), and advantage is taken of this knowledge to terminate the search as soon as that one instance is found, then only half of the collected instances will be visited on average. On the other hand, if knowledge of the distribution of data values provides

the estimate that, on average, one instance will be found to satisfy the selection criterion, then this will be described as a posteriori selection of one instance because it will have been necessary to visit all collected instances checking them against the selection criterion.

Six forms of selection have now been mentioned. These were first, last, all, exact, a priori and a posteriori. The first three implicitly define how many of the child instances are to be visited,but with the last three it is necessary to include an estimate of the number of instances selected. This estimate may be expressed as a number,or as a percentage of the value length or number of SEARCHPOINT instances. On the basis of the form of selection and the number of instances selected an estimate is made as to the number of instances scanned. The sum of the two gives the number of instances visited.

An example of the use of the selection-entry construct in describing more complex operation requests is presented in Figure 6.4.

```
SEARCH FROM
president SELECT A_POSTERIORI 3.2
=> ( pres_A
     => ( pres_A:last_name /R,
          pres_A:first_name /C
          |                 || end of selection section
        ),
     headed
     => SELECT ALL OF
        administration
        => admin_A
           => ( admin_A:admin_no /R,
                admin_A:inaug_month /R,
                admin_A:inaug_year /R
              )
   )
=|
```

Figure 6.4: A Route Involving Selection

It illustrates the route followed to perform the operation request "find

all presidents with a certain first name and retrieve for each administration they headed its number and the month and year of inauguration".

The first use of the selection-entry construct in Figure 6.4 is in describing how many instances of <<president>> will be selected by a SEARCHPOINT scan on <<pres_A:first_name>>. For each selected instance of <<president>> the route visits its first child <<pres_A>> and in turn the two children <<last_name>> and <<first_name>> of <<pres_A>>. The symbol '|' marks the end of the section of an access route followed to reach the value used in selection by content. Partial pre-order visiting for the scanned instances will terminate at the node immediately before the '|'. The '/C' after <<first_name>> indicates that on every visit from a <<president>> instance the value of instances of this SSE will be compared to determine selection. The '/R' after <<last_name>> means that in the case of a visit from a selected <<president>> instance the value of the <<last_name>> instance will be retrieved (implying a CPU overhead to copy it out of the DES buffers). Parentheses have been used for grouping when more than one child is visited, as with <<pres_A>> and <<headed>> visited from <<president>> and <<last_name>> and <<first_name>> visited from <<pres_A>>. Pre-order visiting from the selected instances of <<president>> will continue past the '|' and on through <<headed>> to select all <<administration>> instances linked to the selected <<president>> instance. From those <<administration>> instances the route contines through <<admin_A>> to its children <<admin_no>>, <<inaug_month>> and <<inaug_year>> which are retrieved as required.

6.2.4 Specification of update operation requests

The previous two examples have been concerned with retrieval. In dealing with update operation requests there is an additional factor to take into account, namely the cost of updating the various file organisations used to speed up access. An example of this kind of housekeeping is the updating of secondary indices after a record has been added or deleted.

If an update operation request is regarded as affecting a certain group of target SSE instances, then its execution can be broken down into two phases. The first is the location and processing of the target instances; this is called the *primary phase*. The second is the location and processing of all SSE instances which must be altered to keep them consistent with the update; this is called the *secondary phase*. It is the secondary phase which performs the housekeeping chores referred to above. In the case of retrieval operation requests there is only a primary phase; no secondary phase is necessary since no changes are made.

Figure 6.5 is a PRODD specification of a route for the operation request "add a new president who is a native son of a certain state". The route followed in the primary phase is shown first. Referring to the PSA example one again, PRESIDENT has CALC location mode and so its placement will be determined by the CALC routine. This is described by entry at <<pres_C>> leading to the selection of 0.0 instances because the cost of locating the place at which to insert a new instance is the same as an unsuccessful search for an instance with the same primary key*. Next one instance of a CDG headed by a <<president>>

* This is true of all three primary key-access methods -- hashing, indexing and binary search -- currently specifiable at the encoding level.

```
PRIMARY
ENTRY AT
pres_C
=> SELECT EXACTLY 0.0 OF
   president /ADDCDG
=|

SECONDARY
FOR all pres_SS
ENTRY AT
system
=> all_pres_SS
   => SELECT A_PRIORI 1.0 OF
      president /M
      => pres_A
         => ( pres_A:last_name /C,
              pres_A:first_name /C
            |                  ||  end of selection section
            )
=|

FOR sons
ENTRY AT
state_C
=> SELECT EXACTLY 1.0 OF
   state
   => (  state_A,
         sons
         => SELECT A_PRIORI 1.0 OF
            president /M
            => pres_A
               => pres_A:last_name /C,
                  pres_A:first_name /C
               |                  ||  end of selection section
      )
=|
```

Figure 6.5: A Route for an Update Operation Request

instance is added. Addition and deletion are specified in terms of CDGs because, as defined in Chapter 4, a CDG is a group of data values always stored or retrieved together.

The secondary phase involves following two additional routes, one for each of the other two parents of <<president>> (<<all_pres_SS>> and <<sons>>). The processing described by those two routes is intended to approximate that entailed in updating the coset occurrences of ALL-PRES-SS and NATIVE-SON when a new PRESIDENT occurrence is added.

The first route specified leads to the correct position at which to insert the new <<president>> instance in the chain of <<president>> instances under <<all_pres_SS>>. It begins at entry point <<system>> and then searches through the chain of <<president>> instances comparing the values of <<pres_A:last_name>> and <<pres_A:first_name>> accessible from those instances. Once the <<president>> instance which should immediately precede the new instance has been located, its NEXT pointer field for the <<all_pres_SS>> value sequence will have to be changed. This is indicated by marking the one selected instance of <<president>> with a '/M' for modify. The second route is basically the same. It describes the processing to find the correct position at which to insert the new <<president>> instance in the <<sons>> value sequence under the <<state>> instance for the named native state.

There is one further refinement concerning the possibility of conditional processing. Conditional processing refers to the situation in which one of a number of routes may be followed depending on the data values present in the database. The simplest way of describing this is to list all possible routes that might be followed in carrying out the operation request along with the probability of their being used. The expected database access time for the operation request can then be calculated from the average database access time for each route and the associated probabilities. This is the approach adopted in PRODD.

An example can be constructed as follows. Suppose one of the transactions carried out on the PSA database has the description "retrieve year admitted for a certain state and the names of its native sons if there are any". If the number of states with one or more native sons is 34 then .68 of the states are in that category. Bearing these figures in mind, Figure 6.6 illustrates the PRODD description of the

possible routes that could be followed. This example additionally

```
OPERATION_REQUEST  state_info
  WEIGHTING IS .___
  ROUTE  no_1      || no sons exist
    WEIGHTING IS .32
    PRIMARY
    ENTRY AT
    state_C
    => SELECT EXACTLY 1.0 OF
       state
       => ( state_A
            => ( state_A:state_name,
                 state_A:year_admitted /R
               ),
            sons /C
          )
    =|
  END_ROUTE
  ROUTE  no_2      || one or more sons do exist
    WEIGHTING IS .68
    PRIMARY
    ENTRY AT
    state_C
    => SELECT EXACTLY 1.0 OF
       state
       => ( state_A
            => ( state_A:state_name,
                 state_A:year_admitted /R
               ),
            sons /C
            => SELECT ALL OF
               president
               => pres_A
                  => ( pres_A:last_name /R
                       pres_A:first_name /R
          )          )
     =|
  END_ROUTE
END_REQUEST
```

Figure 6.6: Conditional Processing

illustrates the full syntax for specifying an operation request. The first weighting indicates the frequency of occurrence of this operation request amongst all requests in the activity set. Next is a specification of the route followed when no sons exist. It ends with a comparison

on the value of the C_string <<sons>> instance which reveals that no <<president>> instances are collected under this particular <<sons>> instance. The second route is for the case in which sons do exist. In this case the route continues on to visit all <<president>> instances and retrieve the name information accessible from them.

6.3 Evaluation of operation requests

This section explains how estimates of database access time (DAT) are obtained for the operation requests described by activity specification. The complete form of an operation request description is sketched in Figure 6.7. Figure 6.7 illustrates that, in general, an operation request is performed by following one of a number of routes, R_1, R_2,..., R_n with associated frequency weightings W_1, W_2, ..., W_n.. The average database access time for an operation request is then given by

$$DAT[OR] = \sum_{i=1}^{n} W_i * DAT[R_i] \qquad (6.1)$$

A route in turn will generally consist of one primary sub-route with average access time $DAT[P_i]$ and s secondary sub-routes with average access time $DAT[S_{ij}]$, j=1,...,s. Therefore the average access time for a route R_i is given by

$$DAT[R_i] = DAT[P_i] + \sum_{j=1}^{s} DAT[S_{ij}] \qquad (6.2)$$

It now remains to explain how database access time for a sub-route ($DAT[P_i]$ or $DAT[S_{ij}]$ is estimated. As mentioned earlier in this chapter a sub-route is described in terms of partial pre-order visiting of the string structure graph. This can be conveniently represented by

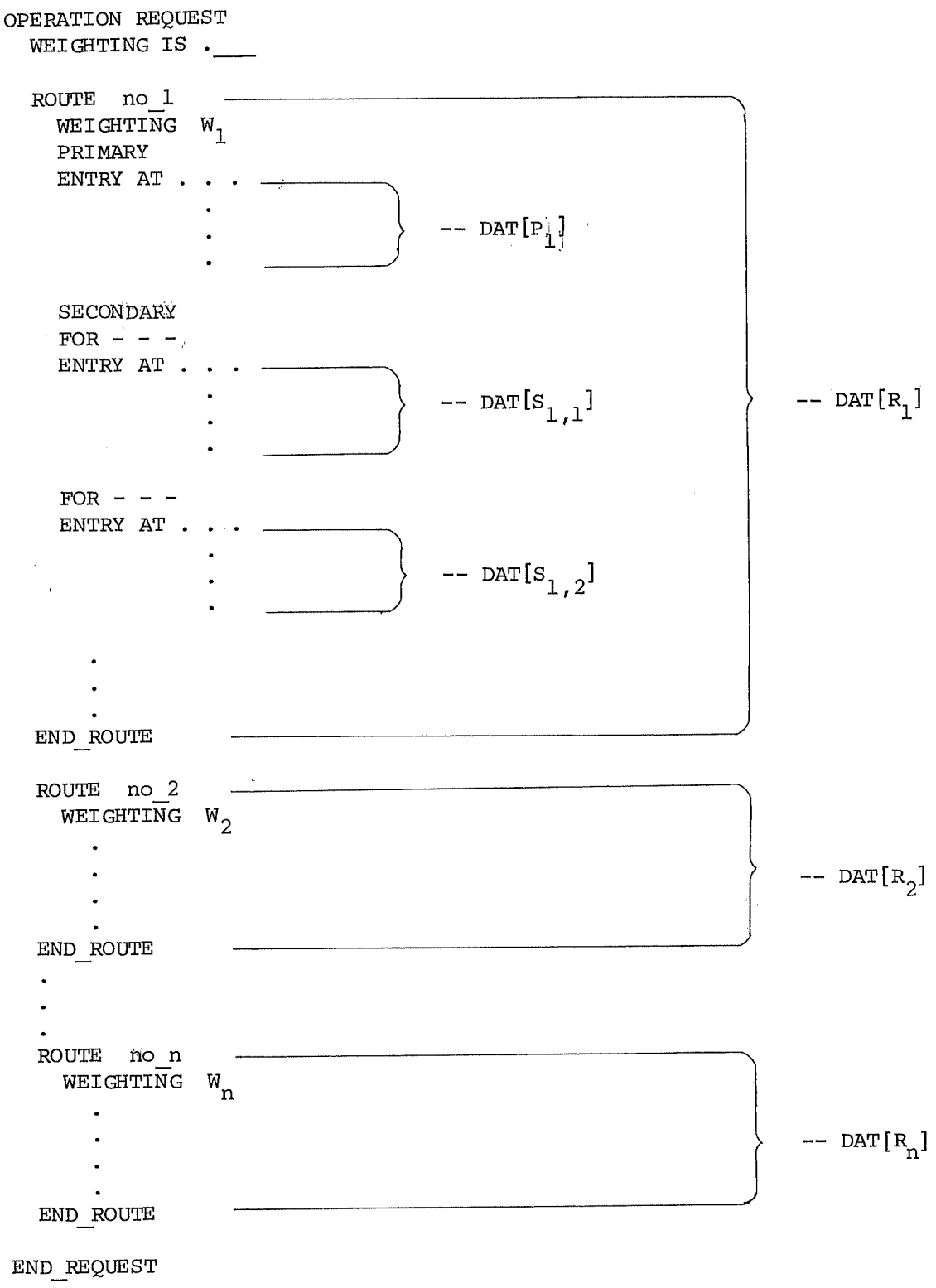


Figure 6.7: The General Form of an Operation Request

a *doubly-chained tree* [Suss63]. The doubly-chained tree is a way of representing a j-ary tree using nodes with a fixed number of pointer fields. In its simplest form each node has two pointer fields, one for the first child in its filial set and another for one of the node's siblings. Figure 6.8 shows a j-ary tree and its equivalent representation as a doubly-chained tree.

Figure 6.9 illustrates the doubly-chained tree representation of the operation request of Figure 6.3 which retrieved year admitted for a given state. This is a simple example in which only one instance of

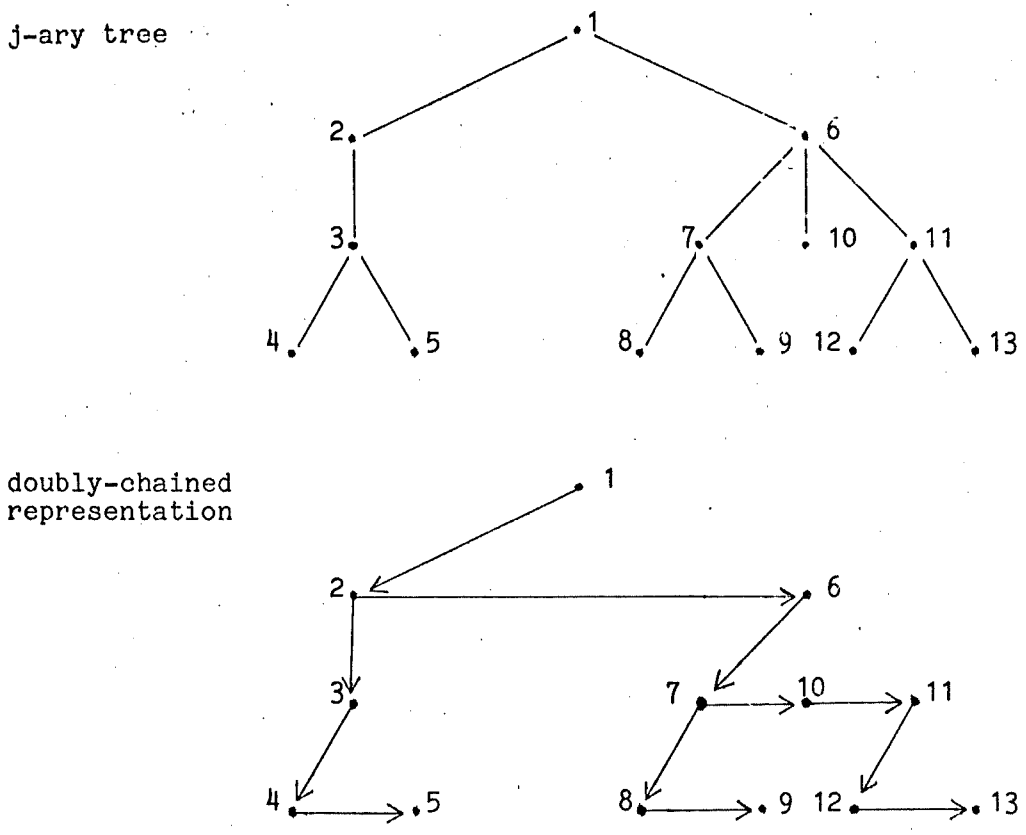


Figure 6.8: Representation of a J-ary Tree as a Doubly-Chained Tree

every SSE on the route is visited. Each SSE is represented by a node with six fields. The first two are pointer fields, indicating the nodes for the first child and sibling to be visited respectively. A

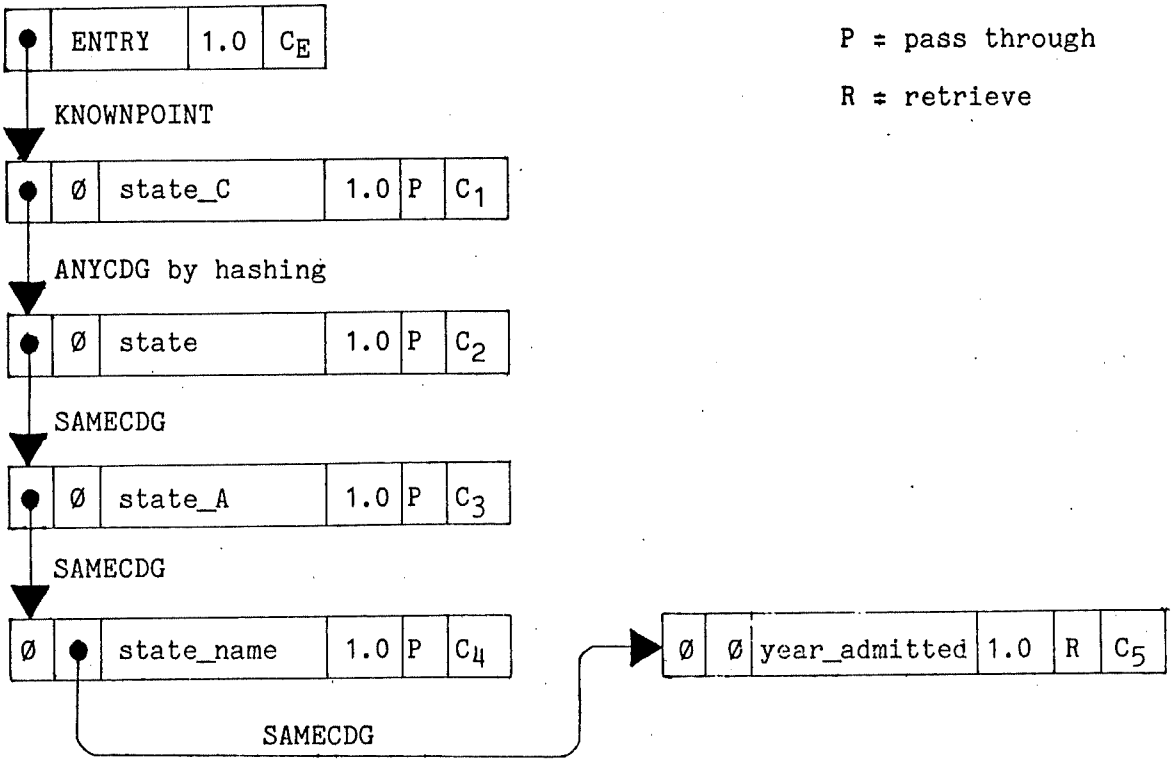


Figure 6.9: Doubly-Chained Tree for an Operation Request

null pointer is indicated by the symbol 'Ø'. The third field contains the name of the SSE. The fourth is a pointer to a vector of multipliers, one per evaluation point, indicating how many instances of that SSE are to be visited. In the example all the vectors are unit vectors. The fifth field records the operation to be carried out on those instances and the sixth points to a vector of costs, again one per evaluation point. The cost of processing a node has two components, CPU processing time and secondary storage accessing time. These are determined by the operation to be performed on the SSE instance(s) represented by that node. If it is passing through or retrieval then the secondary storage accessing component will be zero. Costing is done for each evaluation point because the secondary storage access times incurred by the update operations of insertion, modification and deletion will vary with

database content.

To arrive at the total cost of travelling a (sub-)route is a relatively simple matter of adding up the costs of visiting each node and of following the connections between nodes. The cost of following a connection depends on its type. If it is SAMECDG then it is assumed to be nothing. If it is NEXTCDG then it is the cost of one sequential access times the probability that the next CDG is not on the same page. If it is ANYCDG then it will be the cost of one random access. Normally the cost of following a connection will only have the secondary storage component of database access time. However, in the case of connections established by one of the key-access methods there will be a CPU component as well, for example the time to evaluate a hashing function.

Figure 6.10 presents a mathematical formulation of route costing, It defines the cumulative cost of visiting a node and all its descendants.

Let the cost of processing node N be p(N) and the cost of following a pointer P linking elements of a value sequence be f(P). Then the cumulative cost, C, of visiting node N and all its descendants may be defined recursively as follows.

If node N has m children with cumulative costs

$$C_1, C_2, \ldots, C_m$$

and multipliers

$$M_1, M_2, \ldots, M_m$$

linked by pointers

$$P_1, P_2, \ldots, P_m$$

where P_i is the pointer leading to the i^{th} child

then

$$C_N = p(N) + \sum_{i=1}^{m} f(P_i) + M_i C_i + (M_i - 1) f(P_{i+1}) \qquad (6.3)$$

Note for a leaf node L, $C_L = p(L)$

If the processing cost for the special ENTRY node E is defined to be zero, the DAT for a (sub-)route is C_E.

Figure 6.10: Cumulative Costing

The cumulative cost of the root node plus the cost of accessing the entry point instance represented by the root node is the database access time for the whole route. In expressing the general cumulative cost formula some conventions have been used. All the descendants of an A or L_string have unitary multipliers. Thus in their case (6.3) reduces to

$$C_N = p(N) + \sum_{i=1}^{m} [f(P_i) + C_i] \qquad (6.4)$$

In (6.4) C_N is the sum of processing a node plus the cost of reaching each of its child instances and their cumulative costs. C_strings only have one child so m=1. There are potentially two types of connection in their value sequences, however. These are P_1, the type of connection linking a C_string instance with its first child instance, and P_2, the type of sibling (NEXT) pointer connecting collected instances. For a C_string, (6.3) reduces to

$$C_N = p(N) + f(P_1) + M_1C_1 + (M_1-1)f(P_2) \qquad (6.5)$$

In (6.5) C_N is the sum of processing a node and reaching its first child instance plus the product of the cumulative cost for a child instance times the number of child instances (the multiplier, M_1) plus the cost of following the sibling connection to reach all child instances visited after the first one.

In Figure 6.9 all the multipliers were 1.0. A simple example using non-unitary multipliers can be demonstrated by changing the operation request of Figure 6.3 slightly to become "retrieve state name and year admitted for all states". A route for this operation request is illustrated in Figure 6.11.

```
ENTRY AT
system
=> ( all_pres_SS
     all_states_SS
     => SELECT ALL OF
        state
        => state_A
        => ( state_A:state_name /R,
             state_A:year_admitted /R
           )
   )
=|
```

Figure 6.11: A Route Visiting All <<state>> Instances

Throughout the lifetime of the PSA database the number of states will be constant at 50. Hence the multiplier vector for the node for <<state>> instances will be a constant vector with all elements set to 50. Figure 6.12 depicts the doubly-chained tree representation of the route.

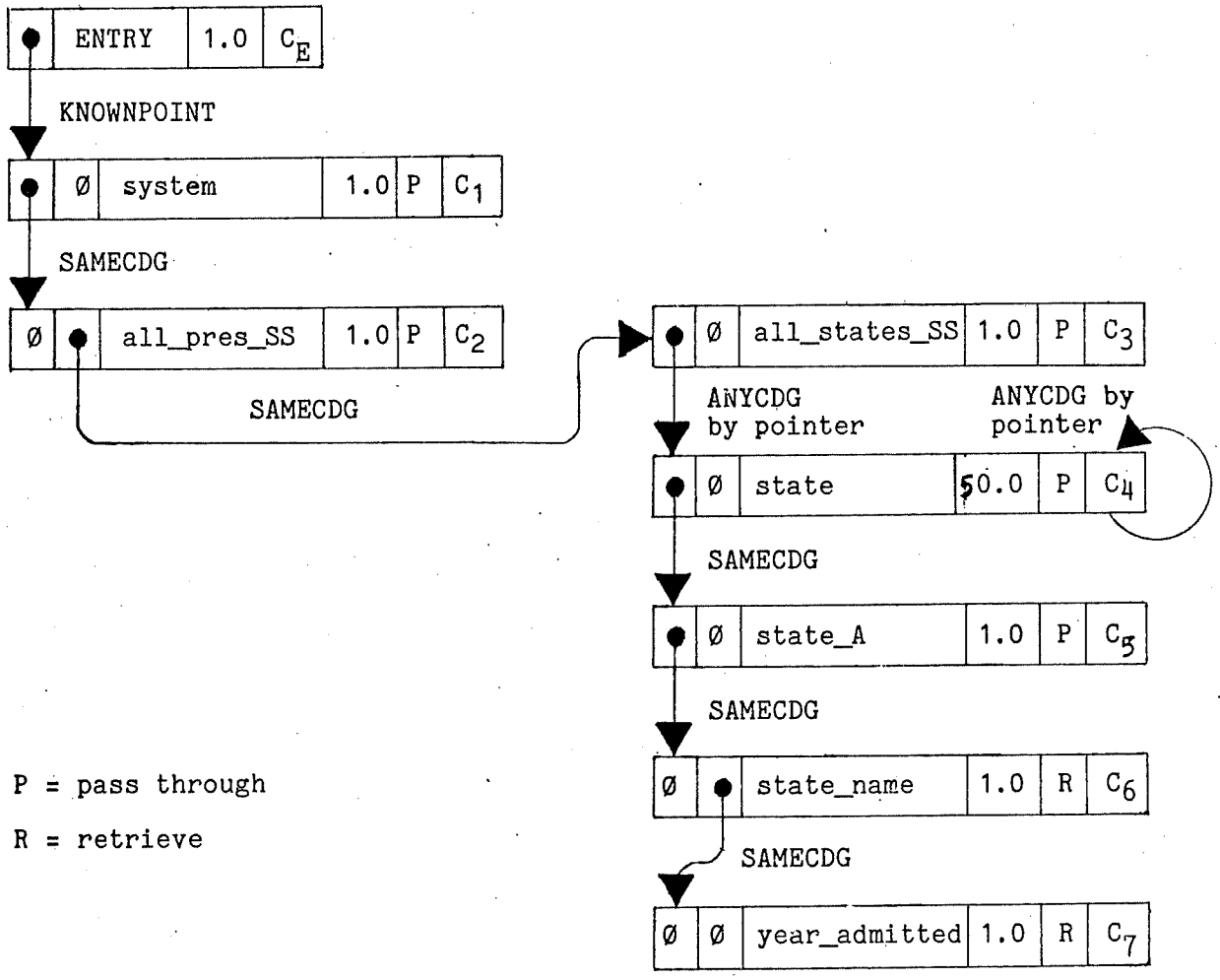


Figure 6.12: Tree for a Route Visiting All <<state>> Instances

7. SEER -- EXPERIMENTATION WITH A PREDICTION PROGRAM

This chapter falls into two parts. The first describes SEER, the experimental prediction program developed using the concepts of the PRODD model. The second presents some preliminary results from experimentation with SEER on an operational DBS.

7.1 SEER

SEER has two principal components: a *translator* and a *cost estimator*. The translator accepts a DBS description written in the PRODD language informally introduced in the examples of the preceding three chapters and formally defined by the BNF grammar of Appendix B. It generates an internal representation of that description for use by the cost estimator. If the description is syntactically correct and satisfies a number of semantic checks the cost estimator proceeds to evaluate the access routes defined by activity specification and thus arrives at estimates of database access time. Figure 7.1 depicts this organisation.

7.1.1 The translator

The translator is based on parsing by the top-down technique of recursive descent [Grie71]. There is approximately one routine to every BNF rule in Appendix B. Part of the commentary for each routine is a statement of the BNF rule or rules it is to translate. Appendix B has been produced by extracting those rules from the source listing of the translator.

Recursive descent was chosen as the parsing technique because it is relatively easy to program and, more importantly, it is easy to make syntactic changes by adding or removing the routines corresponding to

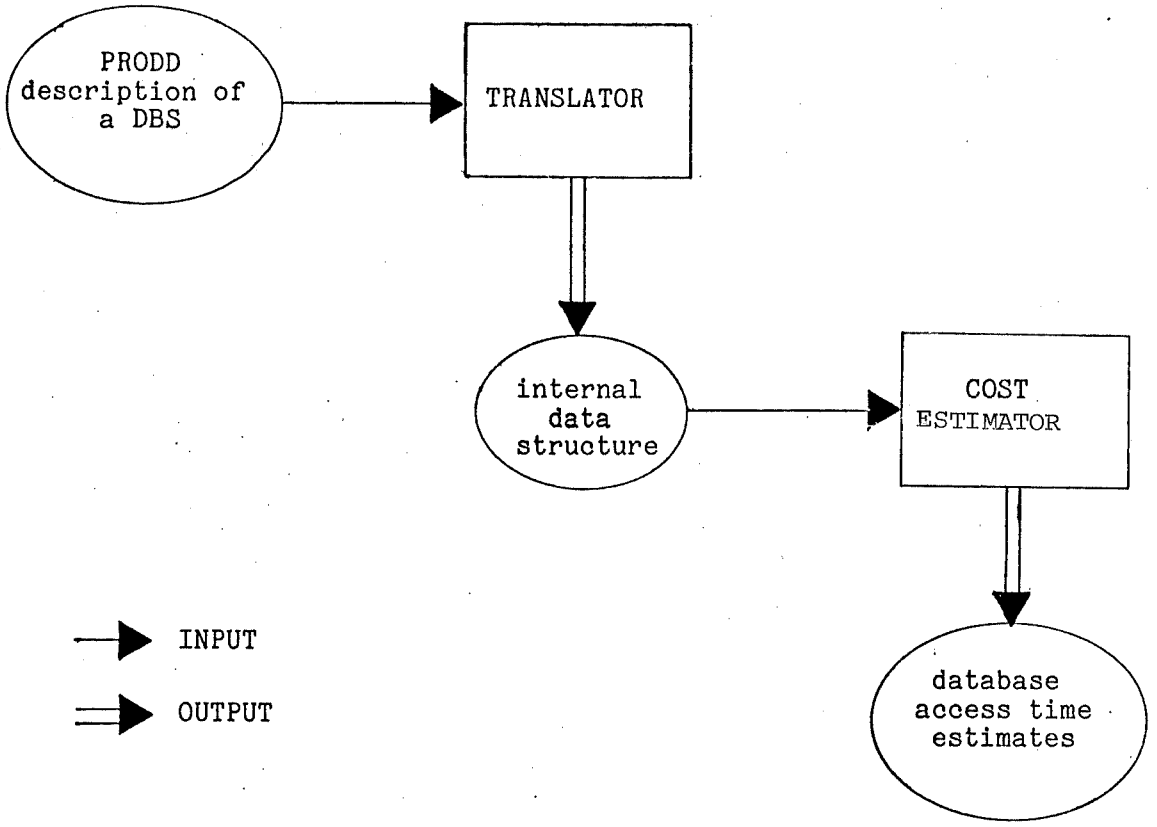


Figure 7.1: SEER components

new or deleted syntactic rules. Recursive descent also makes it possible to produce accurate error messages, a feature which the author found helpful in debugging, aside from its obvious desirability from the user's standpoint.

In general the translator makes as many semantic checks as it can. There are two reasons for this policy. The first is so that the cost estimator routines can "trust" the data structure produced by the translator to be complete. An example would be that, if a BEU specifies the length of a value sequence as being variable then the translator checks that a length is indeed specified for each evaluation point by the content specification component. Consequently, the cost estimator routines can assume without checking that these values exist when it

finds in the BEU description that they should exist. The second reason is that a PRODD description of any non-trivial example is reasonably complicated and it is probably as easy to make mistakes as when writing in a high-level programming language. It is clearly desirable that the translator trap as many of these as it can. For example, the translator ensures that the minimal access constraint is satisfied and that every SSE declared at the access level does in fact have an encoding (BEU) specified for it at the encoding level. More subtly, it checks points such as that members of a clustered value sequence have had the location of their siblings (via the NEXT entry) specified as being in any CDG.

7.1.2 The cost estimator

The cost estimator is based on the principle of solving a large and difficult problem by dividing it into smaller, more manageable pieces. Here the problem is costing an operation request. Chapter 6 showed that this could be reduced to the problem of costing an access route which was in turn reduced to costing the visiting of instances of the SSEs making up the access route. Costing the visiting of an SSE instance is a question of estimating quantities such as the expected number of probes in retrieving a record stored by external hashing*. Equations for these quantities are either available in the literature on file organisation evaluation (surveyed in chapter 2) or have been derived by the author (cf. [Peza 76]). Values to be substituted for the parameters of the equations are obtained from the information in a translated DBS description. A method for combining the costs of "small pieces" into an overall estimate for an operation request is defined by equations (6.1), (6.2) and (6.3) of chapter 6 (cf. section 6.3).

* Knuth's term for using hashing to store records on direct-access storage devices such as discs or drums [Knut73].

An example of the costing of "small pieces" in SEER is the routine to estimate the CPU and secondary storage access costs of using hashing with linear probing to realise a key-access C_string. These consist of:

1. the CPU cost of evaluating the hashing function -- supplied by the software specification component and a calculation of the total size in bytes of the hash key(s).

2. the CPU cost of scanning the CDGs hashing to the same home bucket -- the average number of CDGs scanned is calculated using formulae from [Knut73].

3. the secondary storage accessing cost of reading the pages on which the scanned CDGs reside -- the average number of pages read is interpolated from a table from [Seve76].

7.1.3 SEER size, development and running time

SEER is a large program and represents a considerable programming effort. The initial design work was begun in 1976. Two years later it consists of just under 19,000 lines of BCPL, a high-level systems programming language [Rich73]. The object code generated for an IBM 370/165 is slightly more than 165K bytes*. Table 7.1 is a breakdown of the distribution of almost 1,000 hours of the author's time among the various tasks involved in SEER's development. These hours were logged daily and represent actual productive time spent on the project. For the most part the captions should be self-explanatory. Shared refers to time spent on various utility routines used throughout the system. Testing covers the construction of test data and its input, the running of tests and subsequent debugging and time spent in making minor changes to correct bugs revealed by testing. Time spent on major changes is included under the headings of coding etc. for the appropriate component. Maintenance includes the revision of existing code owing

* This figure does not include the BCPL runtime system.

	translator hours	%	cost estimator hours	%	shared hours	%	task totals hours	%
design	57.05	11.9%	13.70	12.8%	33.20	8.7%	103.95	10.7%
coding	175.15	36.7%	35.40	33.0%	69.15	18.1%	279.70	28.9%
input/edit	103.95	21.8%	14.85	13.8%	80.95	21.2%	199.75	20.6%
desk checking	28.30	5.9%	2.95	2.8%	5.05	1.3%	36.30	3.8%
compiling	2.95	.6%	2.05	1.9%	16.50	4.3%	21.50	2.2%
testing	102.55	21.5%	37.25	34.7%	94.95	24.8%	234.75	24.3%
maintenance	8.00	1.7%	1.25	1.2%	82.50	21.6%	91.75	9.5%
component totals	447.95	49.4%	107.45	11.1%	382.30	39.5%	967.70	100.%
component size (lines of BCPL)	9,500	50.2%	2,400	12.7%	7,000	37.1%	18,900	100.%

percentages for component and task totals are of total hours (967.7)

Table 7.1: Breakdown of Time Spent on SEER's Development

to changes in program specifications, the reorganisation of modules as the program grew and the bookkeeping involved in keeping track of module versions (source and object code) and test output. The figures for design do not reflect the full design effort because they do not include the time spent on deciding PRODD's syntax nor on formulating the strategy for route costing. Those hours really ought to be included since the translator is driven by the syntax rules it has to translate and the cost estimator is a relatively straightforward implementation of the route costing strategy outlined in chapter 6. The figures for the cost estimator component are on the low side because not all of it

has been written. Due to time pressure, some of the low-level cost estimator routines not actually called into play in evaluation of the test case have been left uncoded. The necessary theoretical work to write them has been done and the translator generates the parameters required to drive them, but at the moment if called they abort the run with an appropriate message.

The major portion of execution time in a SEER run is spent in the translation phase. Translation of the full PSA example (about 780 lines) takes just over 1.6 seconds on an IBM 370/165. For the complete master index description (Appendix A) it is slightly more than 1.8 seconds. The costing phase is much quicker since much of the work has already been done by the translator. Costing of the six operation requests in the PSA example at three evaluation points takes .343 seconds. The costing phase for the two operation requests modelling the MI and MI0003 transactions at five evaluation points is .170 seconds. Thus costing takes about .02 seconds per data point.

It may be interesting to compare the above figures with similar ones for the File Design Analyzer (FDA) of Teory and Das mentioned in chapter 2. The FDA is a program which produces estimates of secondary storage overhead and I/O processing time by analytic methods for a wide range of file organisations. It is written in ANS/FORTRAN and consists of about 1,800 source lines compiling to 96K bytes of object code. Execution time on an IBM 370/168 is approximately .1 seconds per data point.

7.2 Background to experimentation

This section and the next discuss the author's experience with

SEER in predicting the performance of several transactions running on an operational DBS. SEER has been used to make performance predictions for the system as it currently stands. Those predictions are compared with actual measurements obtained from a built-in monitoring facility. The use of SEER in predicting the effect of a proposed change to the DBS is also discussed. These results should be interpreted as preliminary. Unfortunately,lack of time has prevented the full range of experimentation that had been intended.

7.2.1 The test case

The DBS used as a basis for comparison is a patient administration system running at Addenbrooke's Hospital, Cambridge. The system is intended to maintain online information about patients currently receiving treatment at the hospital and to provide for the retrieval from archival storage of the medical history of any patient previously treated at the hospital. Online information is envisaged to include patient registration details (address, family physician), outpatient appointments, waiting lists and bed occupancy, results of laboratory tests and theatre scheduling. Applications currently running at Addenbrooke's cover a master index of patients, registration of patients and outpatient appointments. A general review of the aims of the patient administration system and the database design evolved to meet them is given by Baker et. al. [Bake74].

The Addenbrooke's system has been developed using Xerox's Extended Data Management System (EDMS) [Xerox73]. EDMS is an implementation of the 1971 Codasyl DBTG report [Coda71]. The hospital computer unit has a Xerox Sigma 6 computer with 512K bytes of primary memory and 147M bytes of disc store. Most of the work is done during the day

using online transaction processing. Typically, some 2,500 transactions are handled in an 8-hour day. A number of batch programs are run during the evening and night to carry out document printing and maintenance functions.

The online patient administration system is resident in two partitions. One partition is occupied by a terminal interface controller which polls terminals and places messages on a queue for the transaction processing monitor resident in the other partition. The transaction processing monitor initiates the loading of the correct application program to carry out a transaction. EDMS calls are made by the application programs, of which at most one will ever be active (eligible to run) at a given moment. The effect is that database accessing initiated by the online system is single-threaded, a fact which makes the Addenbrooke's system particularly suitable as a basis for comparison with SEER's predictions.

7.2.2 The master index

The master index contains an entry for every patient treated at Addenbrooke's since 1948. Besides such details as name, address, sex and date of birth, each entry contains the hospital number (a unique serial number) assigned to every patient and used throughout the rest of the system to identify the patient in all other processing. The master index is designed to serve two functions. The first is a search on a person's surname and sex and, optionally, forename and date of birth, to determine if the individual concerned has ever been treated at Addenbrooke's before. If so, an index entry will be found giving a hospital number which can be used to retrieve the person's medical history and which will be used in any other transactions run in dealing

with him or her. If not, a new hospital number is assigned, registration details collected and the person added to the index. The second function is to provide a translation from patient hospital number to personal details, i.e. name, address, age and so on.

These two functions of the master index mean that it will be consulted extremely often. Great care was therefore taken in designing a suitable structure for it. This included writing programs to make statistical measurements on the file of patient entries (converted from manual records) to be taken on at initial database loading. The considerations and reasoning that led to the final design are clearly described in an interesting paper by Fenlon [Fenl75]. Two factors that made it a difficult problem were the highly skewed distribution of surnames among patients, 1% of surnames accounting for 41.5% of patients, and the fact that the master index can be expected to increase monotonically in size since a patient's index entry is never deleted.

Figure 7.2 is a simplified data structure diagram of the master index structure chosen to support searching on name, sex and date of birth. There is one SURNAME-SEX record for every distinct surname-sex combination among the registered patients. SURNAME-SEX records have CALC location mode on SURNAME and SEX data items. There is an INDEX record for every patient, holding the rest of patient details and stored in an INDEX-SET occurrence owned by the appropriate SURNAME-SEX occurrence. INDEX records have VIA location mode through INDEX-SET and therefore will usually be placed on the same page as the owner SURNAME-SEX occurrence. INDEX-SET is ordered on forename. A *primary search* finds the appropriate SURNAME-SEX record by hashing on the supplied surname and sex and then searches through the associated

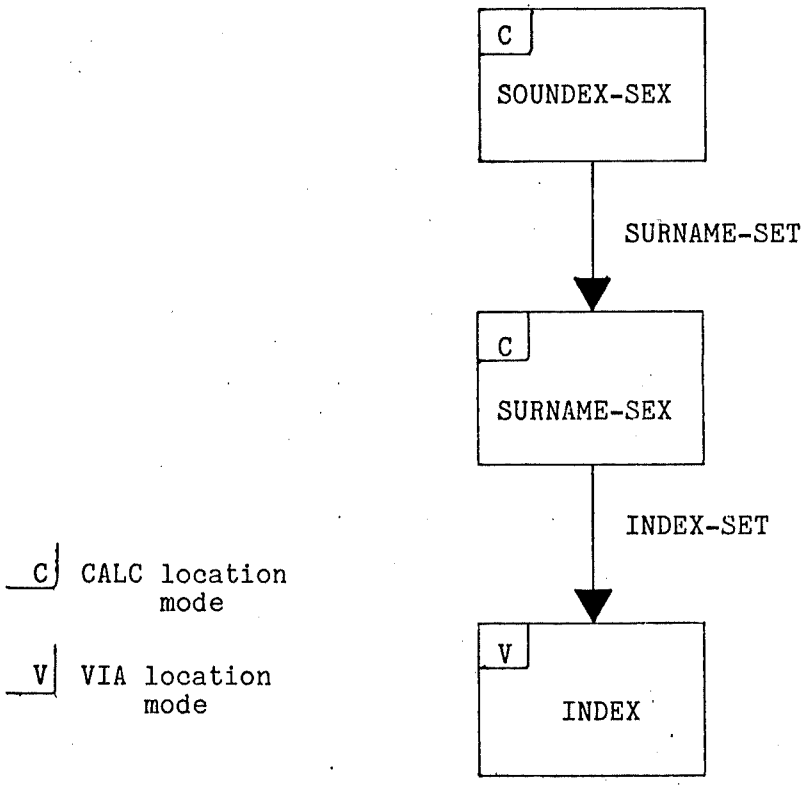


Figure 7.2: Master Index Structure

INDEX-SET occurrence looking for matches on forename and date of birth. The ordering of INDEX records by forename within INDEX-SET is used to terminate a search as soon as a forename higher than the specified one is found.

There is a secondary search mechanism to deal with possible mis-spelling of surnames. Each surname is assigned a Soundex code using the Soundex scheme (described in [Knut73]) which attempts to assign the same four-character code to names which sound the same but are spelled differently. SURNAME-SEX records with the same Soundex code are linked together by membership in an occurrence of SURNAME_SET owned by a SOUNDEX-SEX record having their common Soundex code and sex as

CALC data items. A *secondary search* locates a SOUNDEX-SEX record by computing the Soundex code for the supplied surname and then hashing on Soundex code and sex. All SURNAME-SEX records in the SURNAME-SET owned by that SOUNDEX-SEX record are visited and the INDEX records owned by each of the SURNAME-SEX records are searched as for a primary search. Since the SURNAME-SEX records have CALC location mode they are spread out across the master index page range and there will be one page access for each one visited.

The highly skewed distribution of surnames means that some INDEX-SET occurrences will be extremely long. These long coset occurrences also tend to be frequently visited. It is the surnames with over 150 occurrences which are the 1% accounting for 41.5% of patients. The long INDEX-SET occurrences owned by these common surnames can be expected to be visited 41.5% of the time on the assumption that, over all, patient entries tend to be accessed with equal probability. A strategy of constructing separate INDEX-SET occurrences for the most frequently occurring initials of common names was adopted to speed up the searching of these worst cases. A threshold of 150 patients of the same surname-sex was determined experimentally. At the time of initial loading of the master index in 1975, all surname-sex combinations exceeding this threshold had their most common initial split off and stored under a new SURNAME-SEX record with a modified surname key formed by concatenation of surname and initial. This procedure was repeated until the number of remaining INDEX records dropped below 150. The new surname-initial records are said to be *scattered*. A flag is set in the original SURNAME-SEX record for each initial scattered. Scattered SURNAME-SEX records are not included in any SURNAME-SET occurrence to keep secondary search times down and stop them finding

too many matches.

The master index is stored in a separate EDMS AREA of 15,000 pages. An EDMS page is one disc block of 512 words. Table 7.2 shows the number of occurrences of the various record types as of 1 April 1978. These figures are based on linear interpolation of exact counts taken on

record	count
SOUNDEX-SEX	7,968
SURNAME-SEX	75,203
INDEX	543,748
total	626,919

load factor = .792
page capacity = 53.2

Table 7.2: Master Index Record Counts

15 January 1978 and 15 June 1978. The load factor is the total volume of data space in the 15,000 pages (each page has a 3 word header) divided by the total volume of all record occurrences. Page capacity is the ratio of data space on a single page to the average size of a record occurrence.

Table 7.3 presents some statistics about the distribution of surnames among the patient population prior to database loading in September 1975. In the absence of exact information it has been assumed that the percentage proportions have remained the same over the growth of the master index, i.e. that the 184 surnames accounting for 28.8% of patients in 1975 still did so on 1 April 1978. Again owing to lack of exact information it has been assumed that surnames having

threshold	no. of surnames >= threshold	no. of patients with those surnames	proportion
150	450	179,659	1.0% account for 41.5%
300	184	124,274	0.4% account for 28.8%

total of surnames was 45,481

total of patients was 432,028

male/female ratio was $\frac{208,345}{223,682} = .931$

Table 7.3: Surname Frequencies as of September 1975

more than 300 occurrences also had more than 150 occurrences of each sex. There is a further implicit assumption underlying all calculations discussed in this chapter. It is that inquiries about patients are randomly distributed among the master index population. More precisely, it is assumed that inquiries are randomly distributed among the patients currently undergoing treatment and, at a lower level of activity, among those persons who have received treatment in the past and so have entries in the index.

7.2.3 A PRODD Description of the master index

The information-level view of the master index is very simple. There is just one description set, <<patient>>, with the attributes shown in Figure 7.3.<<soundex_code>> is included as an attribute to model the secondary search mechanism. A fully specified primary search might be described as "find all patient descriptions with <<surname>>=X and <<forename>>=Y and <<sex>>=Z and <<date_of_birth>>=N". A secondary search would be phrased as "find all patient descriptions such that

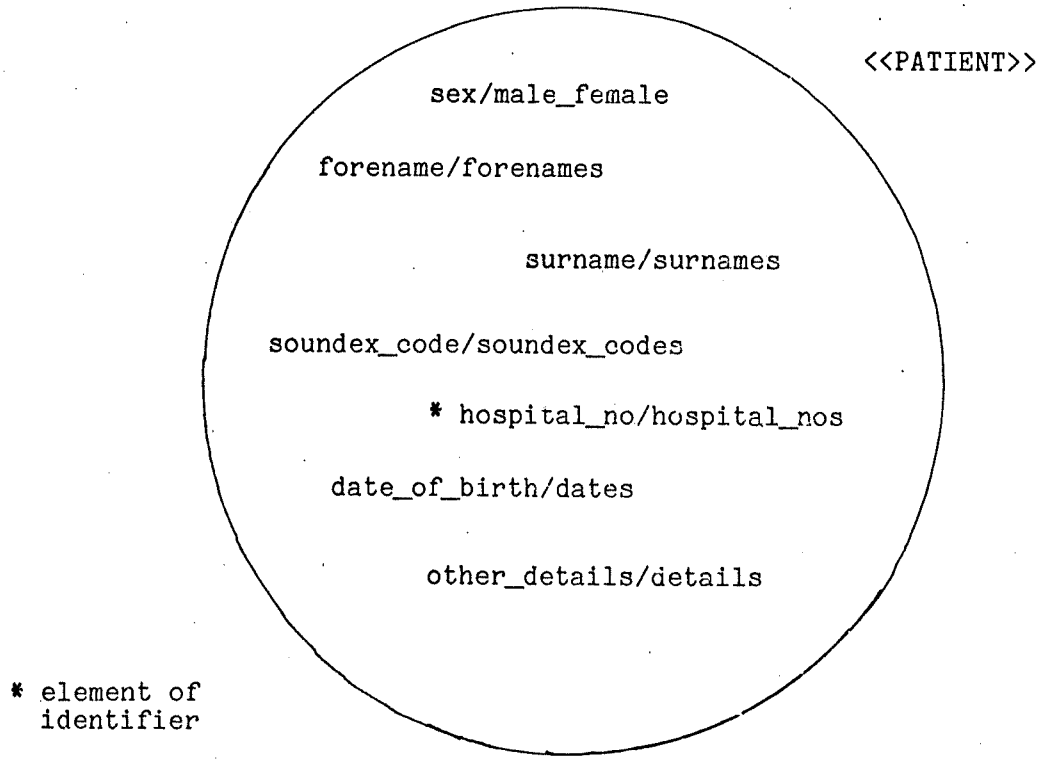


Figure 7.3: Information-Level View of Master Index

<<soundex_code>>=S and <<forename>>=Y and <<sex>>=Z and <<date_of_birth>>=N".

The access-level view is pictured in Figure 7.4. <<soundex_sex>>, <<surname_sex>> and <<index>> model the record types of the same name. Coset linkages have been modelled as in the PSA example. A full PRODD description of the master index appears in Appendix A.

7.2.4 Timings for master index transactions

The two transactions chosen for modelling were those that do primary and secondary searching, known as MI and MI0003 respectively. They were picked because they are run frequently and therefore average figures for them have significance. They have the additional property

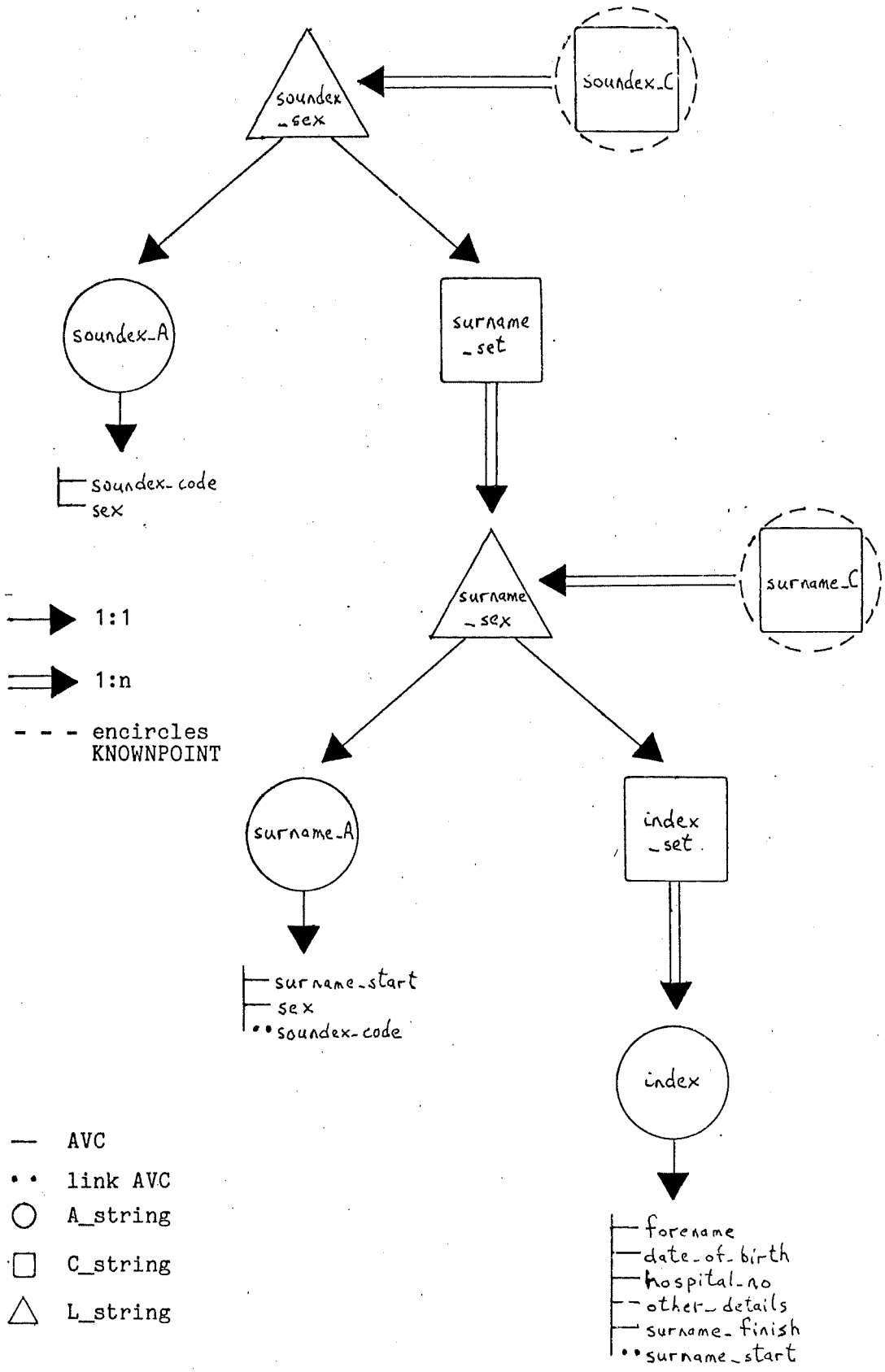


Figure 7.4: Access-Level View of Master Index

of being self-contained and not referencing other parts of the database aside from a small scratchpad structure for which a correction factor can be easily computed.

Measurements were available for the three days 31 March 1978, 1 April 1978 and 3 April 1978. Those for MI and MI0003 are shown in the first two columns of Table 7.4. A third column with the same measurements for all transactions run over the three days is included to give an idea of overall system performance and workload. All figures are averages apart from the first row giving the number of times

	MI	MI0003	all
no. of times ran	431	294	7900
records visited	112.320	89.959	92.401
records inserted	8.631	1.017	1.987
records deleted	2.796	12.910	1.243
page reads	7.786	7.636	12.883
page writes	1.077	1.163	1.799
transient accesses	1.077	1.163	1.799
overlay accesses	2.900	2.820	1.361
journalling writes	3.457	3.272	3.196
user CPU (sec)	.716	.578	.539
total I/O (sec)	.695	.760	1.022
database I/O (sec)	.448	.518	.789
search I/O (sec)	.341	.402	n/a

Table 7.4: Measurements for MI, MI0003 and All Transactions

the transaction ran. Records visited is a count of how many records occurrences were scanned in processing the EDMS calls for the transaction. Records inserted and deleted are exactly that. Page reads and page

writes are the number of page reads and writes done by EDMS*. Transient accesses are page writes done to a temporary disc area for quick rollback of an individual transaction. Overlay accesses are those incurred in bringing in overlays. These are organised so as to always be sequential reads. Journalling writes are sequential tape writes to the journal tape.

User CPU is the CPU time used by the application program and EDMS put together. This includes many factors not taken into account in SEER's estimation of the CPU processing component of database access time. In particular, SEER does not make any attempt to estimate the CPU time used by the application program. As a result there is not much correspondence between CPU overhead as predicted by SEER and the user CPU figure produced by the monitoring module. These points will be taken up in more detail when SEER's predictions are compared with the measurements from Table 7.4.

Total I/O time is not actually recorded by the monitoring module. What is available are user CPU time, teleprocessing monitor time and elapsed time from transaction iniation to termination. Total I/O time has been computed as elapsed time minus user CPU time and teleprocessing monitor time. It therefore includes a component corresponding to the delay incurred in waiting to regain the CPU after losing it to do an I/O request. Unfortunately, there is no way of estimating the size of this component from the information available.

* It is interesting to note the significantly higher number of page reads done overall than by either MI or MI0003 separately, even though the number of records visited in both their cases is about the same or higher than the overall figure for records visited. This undoubtedly reflects the extra care taken in designing the master index structure.

SEER attempts to predict the secondary storage accessing component of database access time. This is not equivalent to the total I/O time for a transaction. Therefore, some factors contributing to total I/O have been subtracted out. First to be removed are transient accesses, overlay accesses and journalling writes*. The result has been called database I/O. A final correction is made to account for the fact that both transactions use a small DBTG coset structure as a scratchpad for recording search results. All update activity for MI and MI0003 shown in Table 7.4 is due to operations on this scratchpad structure. It has not been explicitly modelled with SEER because SEER does not model the effect of multiple buffers (for reasons explained in chapter 3) which is instrumental in this case in ensuring that normally only one page read is necessary to read the scratchpad in at the beginning of a transaction and one page write required to write it out at the end. Consequently, SEER would considerably overestimate the number of page accesses involved in manipulating the scratchpad. It is easy to estimate a correction factor for the MI and MI0003 transactions because all page writes are attributable to scratchpad manipulation and the number of scratchpad reads will always equal the number of writes. Thus the final correction is to subtract the product of twice the number of page writes times an average random access (49.8 msec) from database I/O. The result has been called search I/O and should be close to what SEER estimates as secondary storage accessing time. Search I/O has not been shown in the column for all transactions since there are some transactions that do permanent updating as well as scratchpad manipulation.

* Transient accesses are assumed to be random (average time of 49.8 msec), overlay accesses to be sequential (average time of 20.925 msec) and journalling writes to be sequential from current tape position (average time of 38.3 msec).

7.3 SEER predictions

In making predictions for the MI and MI0003 transactions it is necessary to distinguish carefully between the average length of a coset occurrence and the average length of a visited coset occurrence, i.e. the average length of the coset occurrences actually traversed. They are not always the same. The former is easily computed as the number of owner records divided by the number of member records. Estimating the latter is more difficult. Some assumptions about the underlying distributions of data values and their effect on which coset occurrences are accessed must be made. It is important to get those assumptions at least approximately correct because it is the average length of visited coset occurrences which dominates the performance of MI and MI0003. In the following discussion the term *average visited length* will be used for the average length of visited coset occurrences.

An example shows just how great a difference may exist. The database side of an MI transaction consists of locating one SURNAME-SEX record by hashing and then looking around its set of INDEX records for matches on forename and date of birth. Clearly the important factor here is the average length of INDEX-SETs traversed in MI transactions. From the information in Table 7.2 the average length of an INDEX-SET occurrence is easily calculated as 7.23. However, as pointed out in the discussion of the master index structure, it is the long INDEX-SET occurrences which are most frequently visited. In fact the average visited length of an INDEX-SET occurrence as calculated from measurements is over 140. There will be a great difference in performance predictions depending on which of these figures is used.

Proceeding along these lines, the discussion of SEER's performance

predictions is presented in four sub-sections. The first shows what happens when "naive" assumptions are made about average visited lengths. It is intended as a point of reference -- what is the worst estimate that can be made? The second presents results obtained by making "intelligent" assumptions about the average length of visited cosets. It is intended to show the kind of results an experienced analyst who understood the problem and the prediction program's limitations might be able to obtain. This class of results should be obtainable before actually building the DBS. The third demonstrates what is possible if actual measurements are used to estimate the average visited length. The effect is to tune the predictions to the particular DBS being studied. Those predictions can then be compared with SEER's predictions of the consequences of changing the DBS to yield an estimate of relative changes in performance. The fourth sub-section concludes with an example of using SEER to do just that in the case of the master index.

7.3.1 Predictions based on "naive" assumptions

Table 7.5 shows the results obtained assuming that the average visited lengths of INDEX-SET and SURNAME-SET occurrences are the same as their average lengths. This is equivalent to assuming a uniform

	MI	error	MI0003	error
user CPU (O)	.716	n/a	.578	n/a
CPU overhead (Pn)	.005		.060	
search I/O (O)	.341	-85.3%	.402	+29.4%
SSAT (Pn)	.050		.520	

all times are in seconds

(O) = observed
(Pn) = predicted "naively"
SSAT = secondary storage = accessing time

Table 7.5: Predictions Based on "Naive" Assumptions

distribution of surnames among patients and Soundex codes among surnames. The latter is not unreasonable but the former is not particularly realistic as the results bear out. Error has been expressed as a percentage of observed time with direction indicated by + (high) or - (low). This convention is used throughout the rest of the chapter.

The CPU time figures will be discussed first. SEER tries to estimate the CPU overhead component of database access time by estimating such factors as the time taken to transfer data from DBMS to application program buffers, the time needed to make comparisons on search keys and the time needed to reorganise a page to reclaim space released by deletion. Factors of this kind should be common to all DBMSs and have a place in a DBMS-independent prediction program like SEER. The large difference between the CPU overheads predicted by SEER (which must be regarded as something in the nature of theoretical minima) and the measured user CPU figures are indicative of just how much other processing is being done by EDMS and the application program. For example, EDMS spends time administering its page buffers and interpreting the translated version of the subschema which describes record formats and field sizes. Examining the COBOL listing of the MI program reveals that about 95% of the procedure division is concerned with details that are not directly related to database accessing at all, such as data validation, reformatting for display and moving data around in internal buffers to do character and string processing. Thus measured CPU time and SEER's estimate of CPU overhead cannot be directly compared and, unfortunately, there is insufficient information to make corrections as was done with the I/O time measurements. Table 7.5 shows CPU overhead to be estimated as being an order of magnitude less than secondary storage access time for both transactions. The same result can also be observed in the SEER

estimates based on "intelligent" assumptions and on the use of measurement to estimate average visited lengths*. A similar order of magnitude difference can be observed between the estimates for CPU and I/O time produced by Teory and Oberlander's IDS model, which suggests that SEER's CPU estimates are reasonable even though direct comparison with observed timings has not been possible.

Turning to secondary storage access time, the predicted time for MI bears no relation to observed time. This was anticipated given the highly skewed distribution of surnames among patients. On the other hand, the result for MI0003 is quite promising and shows that the assumption of a uniform distribution of surnames to Soundex codes is not unreasonable.

7.3.2 Prediction based on "intelligent" assumptions

Examining MI (the primary search transaction) first, the basic problem is to arrive at a better estimate of the average visited length of INDEX-SET occurrences. A method suggested by Heising's "80-20 principle" [Heis63] can be used for this purpose. Heising suggested that in many commercial applications 80% of record activity deals with only the most active 20% of a file. In the case of MI, from the statistical measurements recorded in Table 7.2 one can estimate that 41.5% of INDEX records will be in 900 INDEX-SET occurrences+ and the remaining 58.5% distributed among 74,303 INDEX-SET occurrences. Combining the estimates of the average coset lengths for the 900 frequently visited INDEX-SET occurrences and the remaining infrequently visited ones

* Tabulation of the results under the last two classes of assumptions has been limited to the I/O estimates because only they permit meaningful comparison, but the relationship mentioned was observed in the estimates produced by SEER under all three classes of assumptions.

\+ 900 because there will be two SURNAME-SEX records for every common surname.

using weightings of 41.5% and 58.5% respectively yields an estimate of 108.33 for average visited length, considerably more than that of 7.23 obtained under the "naive" assumption of uniform distribution.

In order to improve on the prediction for MI0003 it is necessary to arrive at some estimate of how many SURNAME-SEX records were scattered and use this to arrive at a better estimate of visited average lengths for SURNAME-SET and INDEX-SET occurrences. As already mentioned, it is assumed that the 184 surnames exceeding 300 occurrences among the patient population as measured in 1975 also fall above the scattering threshold of 150 in each sex. The next problem is to estimate how many initials will have to be split off on average to bring the 368 common surname-sex combinations under the threshold. No measurement was made of this figure but an estimate can be made using Zipf's Law [Knut73] which states that the nth most common event (here an event is an initial) occurs with frequency inversely proportional to n*. Assuming a Zipfian distribution for the initials of forenames leads to an estimate that 5 initials per common surname, or 1840 SURNAME-SEX records in all, will be scattered. Average SURNAME-SET length is recalculated excluding the scattered SURNAME-SEX records and used as an estimate for average visited length since the assumption of a uniform distribution of surnames among Soundex codes appears to be a reasonable one. By similar reasoning the average length of visited INDEX-SET occurrences in a secondary search is estimated as the average length of INDEX-SET occurrences owned by non-scattered SURNAME-SEX records.

Table 7.6 presents the predictions obtained on the basis of these assumptions. The prediction for MI has increased markedly in accuracy.

* Zipf's law was originally formulated for and tested on the distribution of words in natural language text but has since been found useful in a large number of other contexts.

	MI	error	MI0003	error
search I/O (O)	.341	n/a	.402	n/a
SSAT (Pn)	.050	-85.3%	.520	+29.4%
SSAT (Pi)	.197	-42.2%	.508	+26.4%

all times are in seconds

(O) = observed
(Pn) = predicted "naively"
(Pi) = predicted "intelligently"

SSAT = secondary storage accessing time

Table 7.6: Predictions Based on "Intelligent" Assumptions

It is now out by less than a factor of 2 in comparison with a factor of 7 under the "naive" assumption. The improvement in the prediction for MI0003 is not so dramatic because the "naive" assumption was already a fairly sensible one.

7.3.3 Predictions based on actual measurement

The method here was to make use of the statistics on records visited, inserted and deleted to estimate average visited lengths for the access pattern of each transaction. By examination of a detailed trace of 25 transactions it was possible to develop formulae for the average visited lengths based on the three measured figures and an estimate of the number of record probes involved in hashing. Table 7.7 shows the results obtained.

Once again an increase in accuracy can be observed. It is difficult to draw any definite conclusions from these figures. Further experimentation would be required first. Nonetheless, it is the author's

	MI	error	MI0003	error
search I/O (O)	.341	n/a	.402	n/a
SSAT (Pn)	.050	-85.3%	.520	+29.4%
SSAT (Pi)	.197	-42.2%	.508	+26.4%
SSAT (Pm)	.251	-26.4%	.375	- 6.7%

all times are in seconds

(O) = observed
(Pn) = predicted "naively"
(Pi) = predicted "intelligently"
(Pm) = predicted using measurements

SSAT = secondary storage accessing time

Table 7.7: Predictions Based on Measurements

feeling that it should be possible to achieve results over a wide range of systems that fall within the 25-40% range based on "intelligent" assumptions and within the 10-25% range based on measurements. To put these in perspective, researchers working with analytical models of single file organisations have reported their predictions as being within 8% [Lum74], 5% [Duhn77] and 10% [Teor76]. The work reported in [Lum74] and [Duhn77] used files of artificially generated data as a basis for comparison whereas in [Teor76] the comparison was made using a file of live data.

7.3.4 Predicting the effect of change

As demonstrated in the previous sub-section, once a DBS has been implemented and its performance can be measured, it is possible to tune SEER's predictions using those measurements. Those improved predictions can be used to estimate the effect of changes to the DBS while holding content and activity constant. The types of changes that can be

investigated are those affecting aspects of a DBS described by the other four components of PRODD. These would include adding an index (access level), changing record boundaries (encoding level), adding a new or faster device (hardware specification), changing the allocation of files on existing devices (SDM) and using a faster hashing algorithm (software specification).

A convenient example can be found in Addenbrooke's master index. The computer unit will soon acquire a new disc unit. It is proposed that it be used to improve the performance of transactions referencing the master index structure because that is at the heart of the entire online system. The easiest way to do this, and one which involves no reprogramming, is to increase the number of pages allocated to storage of the master index. A lower load factor will result and EDMS should have greater success in placing INDEX records on or near the same page as the SURNAME-SEX record owning the INDEX-SET occurrence to which they belong.

Table 7.8 illustrates the predicted effect of increasing the number of pages allocated to the master index structure from 15,000 to 23,000*, an increase of 63.3% and lowering the load factor from .792 to

* This is a slight simplification. The actual proposal is to store the master index structure in two separate EDMS AREAs (for technical reasons not relevant here). One would be for the records supporting searching on surname-sex and the other for the records supplying inversion on hospital number. The predictions in Table 7.8 are based on modelling the existence of two such AREAs. The overall effect, however, is to increase the total number of pages allocated to the master index structure to 23,000, as stated in the main body of the text.

to .563. These results can be explained by considering the dominating

	MI	improvement	MI0003	improvement
SSAT (15,000 pages)	.251	24.7%	.375	0%
SSAT (23,000 pages)	.189		.375	

all times are in seconds
SSAT = secondary storage accessing time

Table 7.8: Effect of Increasing Master Index Page Range

factors in primary and secondary search times. In primary searches it is the expected number of page accesses required to search an INDEX-SET occurrence which is a function of their average visited length and the load factor. Decreasing the load factor will decrease the average number of pages a visited INDEX-SET occurrence spreads across, but the long ones will still spill over somewhere in the order of ten or more pages because their average visited length is unchanged. In secondary searches the dominating factor is the average visited length of SURNAME-SET occurrences. This has not changed at all and is the reason why SEER predicts no improvement for MI0003.

8. EVALUATION AND FURTHER RESEARCH

This chapter begins with a program of further experimentation which would be carried out if time permitted. A review of the limitations of the existing PRODD model and SEER prediction program and a discussion of avenues of research to remove them follows. The chapter concludes with some observations about the conflicting demands of producing a complete theoretical model and building a practical design tool.

8.1 Further experimentation

This section describes further experimentation which could be performed with SEER as it stands at the moment. The first step would be to model an update transaction. One of these was actually selected from among those running on the Addenbrooke's system and the necessary PRODD description written and SEER estimates produced. Unfortunately, the Addenbrooke's systems staff had a mistaken impression of the frequency with which it ran. In the three day sample from which the measurements for MI and MI0003 were taken it only ran twice, not often enough to allow any meaningful comparisons to be made. Preliminary work (statistics gathering, sketching a PRODD description) has also been carried out on the modelling of several transactions drawn from the outpatient appointments application.

Provided that the cost estimator routines that were left uncoded were finished, some limited experimentation with relational systems could be done. The phrase "limited experimentation" is used because PRODD/SEER does not currently have any mechanism for describing "dynamic organisation" of data undertaken in the course of database accessing. Sorting of intermediate relations is an example of this kind of processing which is an important component of database access

time in relational systems. The next section explores how this limitation might be removed.

An opportunity for experimentation with relational-type DBSs exists in a special-purpose relational system under development in the Laboratory by T.J. King. Incorporating data pipelining concepts from PRTV [Todd77], the system provides for the storage and manipulation of a large collection of 16th-18th century parish records used in historical research by a group of anthropologists. Current plans are to extend the prototype system by exploiting indexing techniques to increase its speed. It would be possible to experiment with PRODD/SEER in predicting the effect of introducing these optimisations.

Another avenue for experimentation with SEER would lie in modelling the DML calls of ADABAS [Soft71], a "flat-file" system permitting cross-linking of files [Tsic77]. ADABAS is supported by the University of Cambridge Computing Serivce and so is readily available. It is being used for production purposes by several research projects and it might prove possible to obtain live data for testing purposes. Experimentation with ADABAS would allow detailed checking of PRODD/SEER's accuracy in modelling partial inversion, the file organisation on which ADABAS is based.

It is estimated that nine months to a year would see the above program through. The results would permit some definite statements about the potential accuracy and utility of a general, analytic prediction program instead of the necessarily tentative ones to which this dissertation limits itself.

8.2 Present limitations and further research

The limitations of PRODD/SEER are of two types: 1) minor ones in the sense that they could be dealt with by known methods and 2) major ones requiring further research. A review of the latter will be used to motivate the discussion of directions for further research aimed at their removal.

8.2.1 Minor limitations

There are three of these. The first is that the current definition of secondary storage accessing time does not include any of the overheads due to maintaining database integrity and providing for error recovery. Adding a small section to the PRODD description to specify recovery precautions taken at the page level during updates would enable the modelling of overheads such as tape journalling and quick rollback page writes, which were subtracted from total I/O in making the comparisons presented in chapter 7. A related point is that no allowance is made for the CPU overhead to set up and finish off I/O operations. It was initially left out because it was felt that it is not an easy figure to obtain without detailed measurement of the system (usually the host operating system) responsible for individual physical I/O transfers. Also since SEER reports its estimates of the number of secondary storage accesses as well as the time taken to perform them, it is easy to adjust the CPU overhead estimates if the figure is known. On balance it is now felt that the CPU cost of I/O initiation/termination should be optionally specifiable as part of the CPU description.

The second is that current estimates are averages only with no indication of the range in which values can be expected to fall. Best

and worst case predictions would also be useful. This would mean tripling the amount of data required for content specification and hardware description (maximum, minimum and average would be required instead of just averages) and entail some careful thought in developing equations for the boundary cases.

The third is the previously mentioned inability to describe "dynamic organisation" of records in the course of database accessing. PRODD could be revised to accommodate dynamic organisation by extending the specification of access routes to allow for a second type of node, a processing node. The cost of processing could be either fixed or it could be defined in terms of the number of SSE instances visited in the section of the access route below the node. The latter information could be made available as part of the process of cumulative costing. This extension has the further attraction of making it possible to estimate application program processing time, or at least that component of it directly related to processing of the data returned by or passed to the DBS. The main obstacle would lie in estimating execution times for the sections of code concerned. If it is written in assembly code, hand-timing is feasible, but if it is written in a high-level language the problem is more difficult. Three possible solutions are: 1) a method for "hand-timing" high-level language code developed by Wichman [Wich72], 2) the use of built-in compiler tracing options where they exist or 3) external measurement techniques as described by Storey and Todd [Stor77].

8.2.2 Major limitations requiring further research

There would seem to be three. Two of them are the principal simplifying assumptions mentioned in chapter 3, non-consideration of the

effects of multiple buffering and the restriction to predictions for single-threaded accessing only. The third is not strictly speaking a limitation but is more in the nature of an inelegancy in the formalisation of the real world by the PRODD model. Specifically it is the somewhat piecemeal approach to the handling of overflow in PRODD/SEER.

Multiple buffering was excluded from consideration in chapter 3 because there was no known method of evaluating its effect analytically. There are two possibilities for further research. The first is a follow-up to Greenberg's work [Gree74] to make the same kind of measurements on DBSs as were made on Multics. Saltzer [Salt74] had postulated that a relation of the form

$$MHBF = cZ \quad \text{where } c \text{ is a constant} \tag{8.1}$$

existed between mean headway between page faults (MHBF) and paging memory size (Z). Greenberg carried out extensive measurements of Multics and found this simple linear model to be an adequate characterisation of the paging behaviour of Multics for primary memory sizes less than 16 Mbytes. It seems likely that a similar relation might exist between total DBS buffer size and number of secondary storage accesses,but experimentation would be required to confirm this and obtain values for the constant c in (8.1). Greenberg was careful to add the qualification "We hypothesize that the reference patterns observed, and the headway function derived are characteristic of a large-scale computer utility being used by an academic community through interactive consoles". The second possibility is to take advantage of the fairly detailed description of access routes demanded by PRODD's activity specification component to simulate buffer usage in an average case. A description of an access route can also be regarded as a description of an access (reference) pattern and so is

well-suited to this purpose. Although this strategy would considerably increase the complexity of the low-level cost estimator routines, no insurmountable difficulties are foreseen in implementing it. Again experimentation would be necessary to evaluate the effectiveness of such an approach.

Consideration of multi-threading was also ruled out because there was no established means of estimating its impact analytically. Teory and Das [Teor76] propose a strategy for allowing for the effects of multi-access interference on sequential accessing of secondary storage blocks. They suggest assuming that all accesses to secondary storage are effectively random, even when an individual application is accessing sequential blocks, because of interference from other application programs accessing the database concurrently. Similar reasoning would appear to suggest the assumption that when the number of buffers is relatively small* multi-access interference will reduce performance to that obtainable from single buffering on the grounds that a phenomenon akin to thrashing will occur. Teory and Das do not cite any results from experiments with live data and operational DBSs to support their hypothesis. It is the author's feeling that, despite the intuitive appeal of the above ideas, this must be done before they can be accepted as practically applicable.

The remark made above about the somewhat piecemeal approach to overflow was meant to refer to the lack of a general mechanism for describing the action to be taken on overflow. Instead,each SEER cost estimator routine which deals with an operation where overflow might occur must make allowance for it individually. Schneider [Schn76] has

* relatively small could be made more precise by introducing the notion of an application program having a "working set" of database pages .

suggested two ways of looking at overflow: 1) overflow occurs when a space is too small for the collections of records assigned to it and the solution is to modify the description of the space to increase its capacity or 2) overflow occurs when the collection is too large for the space assigned to its storage and the solution is to redirect the overflowing records somewhere else. In PRODD terms 1) implies associating overflow rules with the LAS and 2) implies associating them with the BEU. Conventional practice favours the former but the latter has advantages when considering the possibility of different overflow rules for SSEs of different types assigned to the same LAS. It is not clear to the author at this time which approach is better. Further study would be necessary.

8.3 Some final thoughts

As defined in chapter 1 the aims of this research were twofold, namely, to develop a general canonical model for the description of DBSs and to explore the feasibility of building an analytic, DBMS-independent prediction program. When I began this project I was unaware of the full magnitude of the divergence between the two goals of producing a complete theoretical model and a practical design tool. The PRODD/SEER system falls somewhere between the two with a tendency to lean towards the former. If one were interested in building a production tool for physical database design it would be desirable to streamline PRODD considerably. Possibilities for doing so include eliminating the information level and revising the encoding level to focus on the CDG (i.e. record) rather than the SSE (i.e. data item). At the same time, having the two goals was a definite asset because work on one stimulated thought on and provided inspiration for work on the other.

I also found it a very valuable experience to work with real data from a real system. There is nothing quite like confrontation with reality for forcing a sharpening of one's ideas and clarification of ill-defined concepts. Furthermore, it was reassuring to see that approximations to reality such as the uniform distribution and Zipf's Law did yield realistic estimates. Many researchers in this field have used artificially generated data as a basis of comparison. There is a certain risk here in that the data is usually generated to conform to the statistical assumptions of one's model. The degree of correlation between predictions and the results of tests made with such artificial data is only really indicative of the confidence that can be placed in predictions for such ideally distributed data. It seems more interesting to observe what happens when the techniques are applied in real life.

The DIAM model has been widely acknowledged as being extremely general in its descriptive scope, as witnessed by the decision of the Codasyl SDDTTG* to adopt it as the starting point for developing its model and language [Coda77] for describing and translating data to be communicated between different computing environments. It has also been described as computationally intractable [Duhn77] precisely because of its extreme generality. This thesis has developed a complete reworking of DIAM in the light of a new analysis of file organisation methods to become the component describing database structure of a general model for describing DBSs. Some preliminary experimental results were presented which indicate that it should be possible to develop an analytic, DBMS-independent prediction program based on the model which would produce reliable, useful estimates. It could not be claimed that

* Stored Data Description and Translation Task Group

SEER as it stands meets those criteria but I would contend that the research described here demonstrates that it can be done and moreover lays the foundations for doing so.

REFERENCES

Aho72 Aho, A. V. & Ullman, J. P.
The Theory of Parsing, Translation and Compiling,
Prentice-Hall, Inc., Englewood Cliffs, N. J., vol. 1,
pp. 37-43 (1972).

Astr76 Astrahan, M. M., Blasgen, M. W., Chamberlain, D. D., Eswaran, K. P. Gray, J. N., Griffiths, P. P., King, W. F., Lorie, R. A., McJones, P. R., Mahl, J. W., Putzolu, G. R., Traiger, I. L., Wade, B. W. & Watson, V.
"System R: Relational approach to database management",
ACM Transactions on Database Systems 1/2, pp. 97-137 (1976).

Bake74 Baker, G. J., Gardiner, S. W. & Gradwell, D. J. L.
"A database for four hospitals in the United Kingdom",
Medinfo 74, North-Holland, London, pp. 323-327 (1974).

Bake75 Baker,G. J.
"The correct use of Codasyl DBTG sets",
Database Journal 6/2, pp. 19-21 (1975).

Bloo69 Bloom, B. H.
"Some techniques and trade-offs affecting large data base retrieval times",
Proc. 24th ACM Natl. Conf., pp 83-95 (1969).

Cana74 Canaday, R. H., Harrison, R. D., Ivie, E. L., Ryder, J. L.
"A back-end computer for data base management",
CACM 17/10, pp. 575-582 (1974).

Card75a Cardenas, A. F. & Sagamang, J. P.
"Modelling and analysis of data base organization",
Information Systems 1/2, pp. 57-67 (1975).

Card75b Cardenas, A. F.
"Analysis and performance of inverted data base structures",
CACM 18/5, pp. 253-263 (1975).

Chen76 Chen, P. P.-S.
"The entity-relationship model -- Towards a unified view of data".
ACM Transactions on Database Systems 1/1, pp. 9-36 (1976).

Coda 71 Codasyl Data Base Task Group
April 1971 Report,
ACM, New York (1971).

Coda73 Codasyl Data Description Language Committee
Data Description Language Journal of Development,
Document C13.6/2:113, U.S. Govt. Printing Office, Washington, D.C. (1973).

Coda77 SDDTTG of the Codasyl Systems Committee,
"Stored-data description and data translation: A model and language",
Information Systems 2/3, pp. 95-148 (1977).

Codd70 Codd, E. F.
"A relational model of data for large shared data banks",
CACM 13/6, pp. 377-387 (1970).

Codd71 Codd, E. F.
"Further normalisation of the data base relational model",
in *Data Base Systems*, Courant Computer Science Symposium 6,
(ed.) R. Rustin, Prentice-Hall (1972).

Coll70 Collmeyer, A. J. & Shemer, J. E.
"Analysis of retrieval performance for selected file organisation techniques",
Proc. 1970 IFIPS FJCC, pp. 201-210 (1970).

Coul72 Coulouris, G. F., Evans, J. M. & Mitchell, R. W.
"Towards content-addressing in data bases",
Computer Journal 15/2, pp. 95-98 (1972).

Date75 Date, C. J.
An Introduction to Database Systems,
Addison-Wesley, Reading, Mass. (1975).

Date76 Date, C. J.
"An architecture for high-level language database extensions",
Proc. 1976 SIGMOD Conf., pp. 101-122 (1976).

deBe76 deBeer, A. & Smit, G. L.
"The performance of data base systems in relation to data and storage structures",
Proc. of the European Comp. Conf. on Computer Performance Evaluation, Online Conference Ltd., Uxbridge, England,
pp. 129-147 (1976).

Dodd69 Dodd, G. G.,
"Elements of data management systems",
Computer Surveys 1/2, pp. 117-133 (1969).

Duhn77 Duhne, R. A.
Optimal Design of a Generalized File Organization,
Ph.D. Dissertation, Cornell University (1977).

Earl72 Earley, J.
"On the semantics of data structures",
in *Data Base Systems*, Courant Computer Science Symposium 6, (ed.)
R. Rustin, Prentice-Hall, pp. 23-32 (1972).

Fenl75 Fenlon, S.
"The on-line patient index at Addenbrooke's Hospital, Cambridge",
Database Journal 6/7, pp. 22-26 (1975).

Fry76 Fry, J. P. & Sibley, E. H.
"Evaluation of data-base management systems",
Computing Surveys 8/1, pp. 7-42 (1976).

Gree74 Greenberg, B. S.
"An experimental analysis of program reference patterns in the Multics virtual memory",
Project MAC Report MAC-TR-129, M.I.T. (1974).

Grie71 Gries, D.
Compiler Construction for Digital Computers,
Wiley & Sons, Inc., New York, pp. 97-100 (1974).

Hamm76 Hammer, M. & Chan, A.
"Index selection in a self-adaptive data base management system",
Proc. 1976 SIGMOD Conf., pp. 1-8 (1976).

Heis63 Heising, W. P.
"Note on random accessing techniques",
IBM Systems Jouranl 2/2, pp. 112-116 (1963).

Hone71 Honeywell Information Systems, Inc.
Integrated Data Store, BR69 (1971).

Hsia70 Hsiao, D. & Harary, F.
"A formal system for information retrieval from files",
CACM 13/2, pp. 67-73 (1970).

IBM74 International Business Machines, Corp.
DBPROTOTYPE, Program Description/Operations Manual,
SH20-1303-1, IBM Palo Alto Development Centre, 1501 California Avenue, Palo Alto, California 94394 (1974).

Kenn73 Kennedy, S. R.
The Use of Access Frequencies in Data Base Organisation,
Ph.D. Dissertation, Cornell University (1973).

Knut73 Knuth, D. E.
The Art of Computer Programming,
vol. 3: Sorting and Searching,
Addison-Wesley, Reading, Mass. (1973).

Koll78 Kollias, J. G.
"An estimate of seek time for batched searching of random or index sequential structured files",
Computer Journal 21/2, pp 132-133 (1978).

Lum71 Lum, V. Y., Yuen, P. S. T. & Dodd, M.
"Key-to-address transform techniques: A fundamental performance study on large existing formatted files",
CACM 14/4, pp. 228-239 (1971).

Lum74 Lum, V. Y., Senko, M. E., Ling, H. & Barlow, J. H.
"Quantitative timing analysis and verification for file organization modelling",
in *Information Systems*, COINS IV, (ed.) J. J. Tou, Plenum, New York, pp. 377-386 (1974).

Meal67 Mealey, G. H.
"Another look at data",
Proc. 1967 AFIPS FJCC, pp. 525-534 (1967).

Meta75 Metaxides, A.
"Information bearing and non-information bearing sets",
in *Data Base Description*, (eds.) B. C. Douque & G. M. Nijssen,
pp. 363-368, North-Holland, Amsterdam (1975).

Naka75 Nakamura, F., Yoshida, J. & Kondo, H.
"A simulation model for data base system performance evaluation",
Proc. 1975 AFIPS NCC, pp. 459-465 (1975).

Nijs74 Nijssen, G. M.
"Data structuring in the DDL and relational model",
in *Data Base Management*, (eds.) J. W. Klimbie & K. L. Koffeman,
North-Holland, Amsterdam, pp. 363-384 (1974).

Nijs75 Nijssen, G. M.
"DDL illustrated with data structure diagrams",
Proc. IFIP IC-2 Special Working Conf. on DDL, Namur, (preprints)
pp. 98-177 (1975).

NASA76 NASA, Lyndon B. Johnson Space Center,
Internal Memorandum 76-FO83-215 on "Completion of generalised
data management system RTOP, Applied Research" (1976).

Ozka75 Ozkarahan, E. A., Schuster, S. A. & Smith, K. C.
"RAP -- An associative processor for data base management",
Proc. 1975 AFIPS NCC, pp. 379-387 (1975).

Pete57 Peterson, W. W.
"Addressing for random-access storage",
IBM Journal of Research and Development 1/2, pp. 130-146
(1957).

Peza76 Pezarro, M. T.
"A note on estimating hit ratios for direct-access storage
devices",
Computer Journal 19/3, pp. 271-272 (1976).

Rich73 Richards, M.
"The BCPL Programming Manual",
Internal report, Computer Laboratory, University of Cambridge
(1973).

Ruth72 Ruth, S. S. & Kreutzer, P. J.
"Data compression for large business files",
Datamation 18/9, pp. 62-66 (1972).

Salt74 Saltzer, J. H.
"A simple linear model of demand paging performance",
CACM 17/4, pp. 181-186 (1974).

Schn75 Schneider, L. S.
*Generalized Data Management System, Math Model Simulator, User
Guide*,NASA Contract NAS9-13951, Institutional Data Systems
Division, Lyndon B. Johnson Space Center, NASA, Houston, Tx (1975).

Schm76 Schneiderman, B. & Goodman, V.
"Batched searching of sequential and tree structured files",
ACM Transactions on Database Systems 1/3, pp. 268-275 (1976).

Schn76 Schneider, L. S.
"A relational view of the Data Independent Accessing Model",
Proc. 1976 SIGMOD Conf., pp.75-90 (1976).

Senk68 Senko M. E., Lum, V. Y. & Owens, P. J.,
"A file organization model (FOREM)",
Proc. 1968 IFIP Cong., pp. C19-23 (1968).

Senk72 Senko, M. E., Altman, E. B., Astrahan, M. M., Fehder, P. L. & Wang, C. P.
"A data independent architecture model 1: Four levels of description from logical structures to physical search structures",
IBM Research Report RJ 982 (1972).

Senk73 Senko, M. E., Altman, E. B., Astrahan, M. M. & Fehder, P. L.
"Data structure and accessing in data-base systems",
IBM Systems Journal 12/1, pp. 30-93 (1973).

Senk76 Senko, M. E. & Altman, E. B.
"DIAM II and levels of abstraction. The Physical Device Level: A general model for access methods",
Proc. 2nd Intl. Conf. on Very Large Databases, Brussels, (preprints) pp. 79-94 (1976).

Senk77 Senko, M. E.
"Data structures and data accessing in data base systems past, present and future",
IBM Systems Journal 16/3, pp. 208-257 (1977).

Seve72 Severance, D. G.
Some Generalized Modelling Structures for Use in Design of File Organizations,
Ph.D. Dissertation, University of Michigan (1972).

Seve75 Severance, D. G.
"A parametric model of alternative file structures",
Information Systems 1/2, pp. 51-55 (1975).

Seve76 Severance, D. G. & Duhne, R.
"A practitioner's guide to addressing algorithms",
CACM 19/2, pp. 314-326 (1976).

Sile76 Siler, K. F.
"A stochastic evaluation model for database organisations in data retrieval systems",
CACM 19/2, pp. 84-95 (1976).

Soft71 Software AG
ADABAS General Information Manual,
Software AG, West Germany (1971).

Spit76 Spitzer, J. F.
"Performance prototyping of data management applications",
Proc. of 1976 ACM Natl. Conf., pp. 287-291 (1976).

Stoc74 Stocker, P. M. & Dearnley, P. A.
"A self-organising data base management system",
in *Data Base Management*, (eds.) J. W. Klimbie & K. L. Koffeman, North-Holland, Amsterdam, pp. 337-349 (1974).

Stor77 Storey, T. & Todd, S. J. P.
"Performance analysis of large systems",
Software Practice & Experience 7/3, pp. 363-370 (1977).

Su75 Su, S. Y. W. & Lipovski, G. J.
"CASSM: A cellular system for very large data bases",
Proc. 1st Intl. Conf. on Very Large Databases, pp.456-472 (1975).

Sund74 Sundgren, B.
"Conceptual foundation of the infological approach to data bases",
in *Data Base Management*, (eds.) J. W. Klimbie & K. L. Koffeman, North-Holland, Amsterdam, pp. 61-96 (1974).

Suss63 Sussenguth, E. H.
"Use of tree structures for processing files",
CACM 6/5, pp. 272-279 (1963).

Tayl76 Taylor, R. W. & Frank, R. L.
"Codasyl data-base management systems",
Computing Surveys 8/1, pp. 67-103 (1976).

Teor76 Teory, T. J. & Das, K. S.
"Application of an analytical model to evaluate storage structures",
Proc. 1976 SIGMOD Conf., pp. 9-19 (1976).

Teor78 Teory, T. J. & Oberlander, L. B.
"Network database evaluation using analytical modelling",
Proc. 1978 AFIPS NCC, pp. 833-842 (1978).

Todd77 Todd, S. J. P.
"The Peterlee Relational Test Vehicle -- A system overview",
IBM Systems Journal 15/4, pp. 285-308 (1977).

Tsic77 Tsichritzis, D. C. & Lochovsky, F. H.
Data Base Management Systems,
Academic Press, New York (1977).

Wich72 Wichman, B. A.
"Estimating the execution speed of an ALGOL program",
SIGPLAN Notices 7/8, pp. 24-44 (1972).

Xero71 Xerox Data Systems,
Xerox Sigma 6 Computer, Reference Manual 901713 (1971).

Xero73 Xerox Corporation,
Xerox Extended Data Management System (EDMS), Reference Manual 903012A (1973).

Yao74 Yao, S. B.
Evaluation and Optimization of File Organizations through Analytic Modelling,
Ph.D. Dissertation, University of Michigan (1974).

Yao77a Yao, S. B.
"An attribute based model for database cost analysis",
ACM Transactions on Database Systems 2/1, pp. 45-67 (1977).

Yao77b Yao, S. B.
"Approximating block accesses in database organisations",
CACM 20/4, pp. 260-261 (1977).

Erratum

In Appendix A the phrase

"SELECT ON ( )"

should be replaced by

"RESTRICT ON ( )"

wherever it appears in a C_string declaration.

APPENDIX A: A PRODD Description of the Addenbrooke's Master Index

```
DBS_DESCRIPTION  hospital_database

STRUCTURE_SPECIFICATION  EDMS_network

INFORMATION_LEVEL  master_index

  DOMAIN  forenames
  DOMAIN  surnames
  DOMAIN  male_female
  DOMAIN  soundex_codes
  DOMAIN  hospital_nos
  DOMAIN  dates
  DOMAIN  details
  DOMAIN  inversion_nos

  DESCRIPTION_SET  patient

    ATTRIBUTE forename / forenames
    ATTRIBUTE surname_start / surnames      || surname split into two parts:
    ATTRIBUTE surname_finish / surnames     || 1st nine letters + remainder
    ATTRIBUTE sex / male_female
    ATTRIBUTE date_of_birth / dates
    ATTRIBUTE hospital_no / hospital_nos
    ATTRIBUTE other_details/details
    ATTRIBUTE soundex_code / soundex_codes
    ATTRIBUTE inversion_no / inversion_nos      || for inversion on
                                                || hospital number

    IDENTIFIER  hospital_no

  END_SET

END_LEVEL


ACCESS_LEVEL  EDMS_access_paths

  || equivalent of SOUNDEX-SET
  || ENTRYPOINT equivalent to ownership by MI area

  C_STRING  soundex_set
    OVER soundex
    SELECT ON ()
    PARTITION ON ()
    ORDER ON (soundex_A:soundex_code/A,soundex_A:sex/A)
    KNOWNPOINT
  END_STRING

  || equivalent of SOUNDEX-SEX & SURNAME-SET
  C_STRING   soundex_C
    OVER soundex
    SELECT ON ()
    PRIMARY KEY_ACCESS ON (soundex_A:soundex_code,soundex_A:sex)
    ORDER ON ()
    KNOWNPOINT
  END_STRING

  L_STRING  soundex     || link data items and set pointers
    OVER (soundex_A,surname_set)
    MATCH ON ((soundex_A:soundex_code,soundex_A:sex) =
              (surname_A:soundex_code,surname_A:sex))
    SEARCHPOINT UNDER (soundex_set,soundex_C)
  END_STRING
```

```
A_STRING  soundex_A      || data items
  OVER patient : (soundex_code,sex)
  UNDER soundex
END_STRING

C_STRING  surname_set      || establish set pointers
  OVER surname
  SELECT ON ()             || not quite correct because doesn't
                           || account for scattered SURNAME records
  PARTITION ON (surname_A:soundex_code,surname_A:sex)
  ORDER ON surname_A:surname_start/A
  RING TO soundex
  UNDER soundex
END_STRING


|| define equivalent of SURNAME-SEX & INDEX-SET

C_STRING  surname_C
  OVER surname
  SELECT ON ()
  PRIMARY KEY_ACCESS ON (surname_A:surname_start,surname_A:sex)
  ORDER ON ()
  KNOWNPOINT
END_STRING

L_STRING  surname      || link data items & set pointers
  OVER (surname_A,index_set)
  MATCH ON ((surname_A:surname_start,surname_A:sex) =
            (index:surname_start,index:sex))
  SEARCHPOINT UNDER (surname_set,surname_C)
END_STRING

A_STRING  surname_A      || data items
  OVER patient : (
    surname_start,
    sex,
    soundex_code IS LINK
                 SOURCE IS soundex_A:soundex_code )
  UNDER surname
END_STRING

C_STRING  index_set      || establish set pointers
  OVER index
  SELECT ON ()
  PARTITION ON (index:surname_start,index:sex)
  ORDER ON ()
  RING TO surname
  UNDER surname
END_STRING


|| define equivalent of INDEX

A_STRING  index          || no L_STRING necessary because index
  OVER patient : (       || is not a set owner
    forename,
    date_of_birth,
    hospital_no,
    other_details,       || address, duplicate hospital no.,
                         || registration status, dead or alive flag
    surname_finish,
    sex,
    surname_start IS LINK
                  SOURCE IS surname_A:surname_start,
    inversion_no IS LINK
                 RESULT OF compute_century (index:hospital_no)
    )
  SEARCHPOINT UNDER (index_set,inversion_ptr)
END_STRING
```

```
  |*
    The following five strings are here to model the INVERSION
    record which is in the same AREA as the other master index
    records.  The INVERSION record provides inversion of the
    INDEX records on hospital no.  It must be included to get
    load factors and page occupancies for the master index
    AREA right.
  *|

  C_STRING  inversion_C
    OVER inversion
    SELECT ON ()
    PRIMARY KEY_ACCESS ON inversion_A:inversion_no
    ORDER ON ()
    KNOWNPOINT
  END_STRING

  L_STRING  inversion      || link century no. & list of index ptrs.
    OVER (inversion_clist,inversion_A)
    MATCH ON (inversion_A:inversion_no = index:inversion_no)
    UNDER inversion_C
  END_STRING

  C_STRING  inversion_clist      || group ptrs to all members with same
    OVER inversion_ptr           || inversion no. (a century)
    SELECT ON ()
    PARTITION ON index:inversion_no
    ORDER ON ()
    UNDER inversion
  END_STRING

  C_STRING  inversion_ptr      || provide ptrs to all members of a
    OVER index                 || given century
    SELECT ON ()
    PARTITION ON index:hospital_no
    ORDER ON ()
    UNDER inversion_clist
  END_STRING

  A_STRING  inversion_A      || to hold inversion (century) no.
    OVER patient : inversion_no
    UNDER inversion
  END_STRING


END_LEVEL


ENCODING_LEVEL EDMS_encoding


  LAS  MI
    STORAGE_UNIT IS BYTE
    PAGE_SIZE IS 512 WORDS
    PAGE_FILL IS 60.%
    PAGE_LABEL IS FIELD OF 3 WORDS
  END_LAS


  BEU FOR soundex_set
    LAS IS MI
    CDGHEAD
    ALIGN ON WORD
    LABEL IS FIELD OF 3 BYTES
    VALUE START IS IN ANYCDG
                PTR IS FIELD OF 4 BYTES
          LENGTH IS VARIABLE
  END_BEU

  BEU FOR soundex_C
    LAS IS MI
    VALUE START IS FOUND BY HASHING
                   USING EDMS_calc WITH PROBING
          LENGTH IS VARIABLE
  END_BEU
```

```
BEU FOR soundex
  LAS IS MI
  CDGHEAD
  ALIGN ON WORD
  LABEL IS FIELD OF 3 BYTES
  NEXT UNDER soundex_set
    IS IN ANYCDG
    PTR IS FIELD OF 4 BYTES
  NEXT UNDER soundex_C
    IS IN ANYCDG
    PTR IS FIELD OF 4 BYTES
  VALUE START IS IN SAMECDG
        LENGTH IS FIXED BY DEFN
              IN FIELD OF 1 BYTE
END_BEU

BEU FOR soundex_A
   LAS IS MI
   NEXT UNDER soundex
        IS IN SAMECDG
   VALUE START IS IN SAMECDG
         LENGTH IS FIXED BY DEFN
END_BEU

BEU FOR soundex_A:soundex_code
  LAS IS MI
  NEXT UNDER soundex_A
       IS IN SAMECDG
  VALUE START IS IN SAMECDG
        LENGTH IS FIXED AS 3 BYTES
END_BEU

BEU FOR soundex_A:sex
  LAS IS MI
  VALUE START IS IN SAMECDG
        LENGTH IS FIXED AS 1 BYTE
END_BEU

BEU FOR surname_set
  LAS IS MI
  VALUE START IS IN ANYCDG
              PTR IS FIELD OF 4 BYTES
        LENGTH IS VARIABLE
END_BEU

BEU FOR surname_C
  LAS IS MI
  VALUE START IS FOUND BY HASHING
                 USING EDMS_calc WITH PROBING
        LENGTH IS VARIABLE
END_BEU

BEU FOR surname
  LAS IS MI
  CDGHEAD
  ALIGN ON WORD
  LABEL IS FIELD OF 7 BYTES     || includes 4 bytes for surname vector
  NEXT UNDER surname_set
       IS IN ANYCDG
       PTR IS FIELD OF 4 BYTES
  NEXT UNDER surname_C
       IS IN ANYCDG
       PTR IS FIELD OF 4 BYTES
  VALUE START IS IN SAMECDG
        LENGTH IS FIXED BY DEFN
              IN FIELD OF 1 BYTE
END_BEU

BEU FOR surname_A
  LAS IS MI
  NEXT UNDER surname
       IS IN SAMECDG
  VALUE START IS IN SAMECDG
        LENGTH IS FIXED BY DEFN
END_BEU
```

```
BEU FOR surname_A:surname_start
  LAS IS MI
  NEXT UNDER surname_A
       IS IN SAMECDG
  VALUE START IS IN SAMECDG
       LENGTH IS FIXED AS 9 BYTES
END_BEU

BEU FOR surname_A:sex
  LAS IS MI
  VALUE START IS IN SAMECDG
       LENGTH IS FIXED AS 1 BYTE
END_BEU

BEU FOR index_set
  LAS IS MI
  VALUE START IS IN ANYCDG
             PTR IS FIELD OF 4 BYTES
       LENGTH IS VARIABLE
END_BEU

BEU FOR index
  LAS IS MI
  CDGHEAD
  ALIGN ON WORD
  LABEL IS FIELD OF 3 BYTES
  NEXT UNDER index_set
       IS IN ANYCDG
       PTR IS FIELD OF 4 BYTES
  CLUSTER UNDER index_set
  VALUE START IS IN SAMECDG
       LENGTH IS FIXED BY DEFN
             IN FIELD OF 1 BYTE
END_BEU

BEU FOR index:forename
  LAS IS MI
  NEXT UNDER index
       IS IN SAMECDG
  VALUE START IS IN SAMECDG
       LENGTH IS FIXED AS 12 BYTES
END_BEU

BEU FOR index:date_of_birth
  LAS IS MI
  NEXT UNDER index
       IS IN SAMECDG
  VALUE START IS IN SAMECDG
       LENGTH IS FIXED AS 7 BYTES
END_BEU

BEU FOR index:hospital_no
  LAS IS MI
  NEXT UNDER index
       IS IN SAMECDG
  VALUE START IS IN SAMECDG
       LENGTH IS FIXED AS 6 BYTES
END_BEU

BEU FOR index:other_details
  LAS IS MI
  NEXT UNDER index
       IS IN SAMECDG
  VALUE START IS IN SAMECDG
       LENGTH IS FIXED AS 63 BYTES
END_BEU

BEU FOR index:surname_finish
  LAS IS MI
  NEXT UNDER index
       IS IN SAMECDG
  VALUE START IS IN SAMECDG
       LENGTH IS FIXED AS 19 BYTES
END_BEU
```

```
  BEU FOR index:sex
    LAS IS MI
    VALUE START IS IN SAMECDG
          LENGTH IS FIXED AS 1 BYTE
  END_BEU

  BEU FOR inversion_C
    LAS IS MI
    VALUE START IS FOUND BY HASHING
                    USING EDMS_calc WITH PROBING
          LENGTH IS VARIABLE
  END_BEU

  BEU FOR inversion
    LAS IS MI
    CDGHEAD
    ALIGN ON WORD
    LABEL IS FIELD OF 3 BYTES
    NEXT UNDER inversion_C
         IS IN ANYCDG
         PTR IS FIELD OF 4 BYTES
    VALUE START IS IN SAMECDG
          LENGTH IS FIXED BY DEFN
                IN FIELD OF 1 BYTE
  END_BEU

  BEU FOR inversion_clist
    LAS IS MI
    NEXT UNDER inversion
         IS IN SAMECDG
    VALUE START IS IN SAMECDG
          LENGTH IS FIXED AS 100 INSTANCES
  END_BEU

  BEU FOR inversion_ptr
    LAS IS MI
    NEXT UNDER inversion_clist
         IS IN SAMECDG
    VALUE START IS IN ANYCDG
                PTR IS FIELD OF 4 BYTES
          LENGTH IS FIXED AS 1 INSTANCE
  END_BEU

  BEU FOR inversion_A
    LAS IS MI
    VALUE START IS IN SAMECDG
          LENGTH IS FIXED BY DEFN
  END_BEU

  BEU FOR inversion_A:inversion_no
    LAS IS MI
    VALUE START IS IN SAMECDG
          LENGTH IS FIXED AS 4 BYTES
  END_BEU


END_LEVEL


END_SPECIFICATION
```

```
CONTENT_SPECIFICATION  March_31_1978_estimates

|*
  These figures are based on linear interpolation of record occurrence
  counts for mid January '78 and mid June '78 to estimate figures
  for the end of March of the same year.  The set occupancy figures
  are for five different cases:

  loading point:       assuming uniform distribution
  evaluation point 1:  adjusted for MI using Heisings's 80-20 rule
                   2:  "                     " group access statistics
                   3:  adjusted for MI0003 using Zipf's law
                   4:  "                         " group access statistics
*|


  EVALUATION_POINT OCCURS 4 TIMES


  VALUE LENGTH

  FOR soundex_set
    AT LOADING IS 7968. INSTANCES
    AT EVALUATION_POINT 1 IS 7968. INSTANCES
                        2 IS 7968. INSTANCES
                        3 IS 7968. INSTANCES
                        4 IS 7968. INSTANCES
  END_FOR

  FOR soundex_C
    AT LOADING IS 7968. INSTANCES
    AT EVALUATION_POINT 1 IS 7968. INSTANCES
                        2 IS 7968. INSTANCES
                        3 IS 7968. INSTANCES
                        4 IS 7968. INSTANCES
  END_FOR

  FOR surname_set
    AT LOADING IS 9.438 INSTANCES
    AT EVALUATION_POINT 1 IS 9.438 INSTANCES
                        2 IS 9.438 INSTANCES
                        3 IS 9.207 INSTANCES
                        4 IS 6.535 INSTANCES
  END_FOR

  FOR surname_C
    AT LOADING IS 75203. INSTANCES
    AT EVALUATION_POINT 1 IS 75023. INSTANCES
                        2 IS 75023. INSTANCES
                        3 IS 75023. INSTANCES
                        4 IS 75023. INSTANCES
  END_FOR

  FOR index_set
    AT LOADING IS 7.230 INSTANCES
    AT EVALUATION_POINT 1 IS 108.333 INSTANCES
                        2 IS 141.586 INSTANCES
                        3 IS   6.227 INSTANCES
                        4 IS   6.227 INSTANCES
  END_FOR

  FOR inversion_C
    AT LOADING IS 5288. INSTANCES
    AT EVALUATION_POINT 1 IS 5288. INSTANCES
                        2 IS 5288. INSTANCES
                        3 IS 5288. INSTANCES
                        4 IS 5288. INSTANCES
  END_FOR

END_SPECIFICATION
```

```
SOFTWARE_SPECIFICATION  EDMS_timings

  FUNCTION EDMS_calc
    TIMING IS 129.2 uSEC + 16. uSEC PER BYTE
  END_FUNCTION

  FUNCTION compute_century
    TIMING IS 31.8 uSEC
  END_FUNCTION

END_SPECIFICATION


HARDWARE_SPECIFICATION  Xerox_Sigma6_configuration

  CPU
    BYTE IS 8 BITS
    WORD IS 32 BITS
    COMPARISON IS 8.5 uSEC PER BYTE
    COPYING    IS 7.4 uSEC PER BYTE
  END_CPU

  |*
    copying per byte is timed as:

        LB   2.5     load byte indexed
        STB  3.1     store byte indexed
        BDR  1.8     branch on decrementing register (branch)
             ---
             7.4 uSEC

    comparison per byte is timed as:

        LB   2.5     load byte indexed
        CB   2.2     compare byte indexed
        BCS  2.0     branch on conditions set (no branch)
        BDR  1.8     branch on decrementing register (branch)
             ---
             8.5 uSEC
  *|


  DEVICE_TYPE  RXD_71
    OCCURS 3 TIMES

    DEVICE IS 400 TRACKS
    TRACK  IS  60 BLOCKS      || a cylinder
    BLOCK  IS 512 WORDS

    BLOCK_TRANSFER_TIME IS 8.3 mSEC

    SEEK_TIME
      SEQUENTIAL IS 7.5 mSEC
      RANDOM     IS 29.0 mSEC

    LATENCY_TIME
      SEQUENTIAL IS 25.0 mSEC      || one rotation because EDMS only set
                                   || up to read 1 page at a time
      RANDOM     IS 12.5 mSEC
  END_DEVICE

END_SPECIFICATION


SDM_SPECIFICATION  MI_area

  LAS MI IS ON DEVICE_TYPE RXD_71

END_SPECIFICATION
```

```
ACTIVITY_SPECIFICATION  Master_Index_Transactions

OPERATION_REQUEST  MI0000
|*
  Primary searching on surname and sex, optionally forename and
  date of birth as well.
*|
  WEIGHTING IS .594      || 431/725

  ROUTE  basic
    WEIGHTING IS 1.0
    PRIMARY
    ENTRY AT
    surname_C
    => SELECT EXACTLY 1.0 OF
       surname
       => ( surname_A
            => surname_A:surname_start /R ,
            index_set
            => SELECT A_PRIORI 1.0 OF
               index
               => ( index:forename /C,
                    index:date_of_birth /C
                    |,
                    index:hospital_no /R,
                    index:other_details /R
                  )
          )
    =|
  END_ROUTE

END_REQUEST

OPERATION_REQUEST  MI0003
  WEIGHTING IS .406      || 294/725
  ROUTE basic
    WEIGHTING IS 1.0
    PRIMARY
    ENTRY AT
    soundex_C
    => SELECT EXACTLY 1.0 OF
       soundex
       => ( soundex_A,
            surname_set
            => SELECT ALL OF
               surname
               => ( surname_A,      || no need to examine surname
                    index_set
                    => SELECT A_POSTERIORI 1.0 OF
                       index
                       => ( index:forename /C,
                            index:date_of_birth /C
                            |,
                            index:hospital_no /R,
                            index:other_details /R
                          )
                  )
          )
    =|
  END_ROUTE
END_REQUEST
```

```
END_SPECIFICATION

END_DESCRIPTION


STATISTICS_SPECIFICATION
LAS MI 15000  632207. 632207. 632207. 632207. 632207.
CDGHEAD  soundex_set  1  1. 1. 1. 1. 1.
                         6. 6. 6. 6. 6.
CDGHEAD  soundex  1  7968. 7968. 7968. 7968. 7968.
                     20. 20. 20. 20. 20.
CDGHEAD  surname  1  75203. 75203. 75203. 75203. 75203.
                     32. 32. 32. 32. 32.
CDGHEAD  index  2  surname  543748. 543748. 543748. 543748. 543748.
                            36.751 36.751 36.751 36.751 36.751
CDGHEAD  inversion  1  5288. 5288. 5288. 5288. 5288.
                       312. 312. 312. 312. 312.
BEU index_set IN surname
END_SPECIFICATION
```

APPENDIX B: PRODD Syntax

The syntax of a language for describing DBSs in terms of the PRODD model is defined below using a variant of BNF. The following rules define the formalism for syntax definition informally:

1. Syntactic constructs (i.e. nonterminals) are named by identifiers without being enclosed in any form of brackets. An identifier for this purpose is any sequence of lower case alphanumeric symbols (possibly including '_') beginning with an alphabetic symbol.

   e.g. attribute
        descriptionset_name

2. '->' separates a construct identifier from its definition and '|' separates alternative definitions.

   e.g. vehicle -> car | lorry | motorcycle

3. Terminal symbols occur as upper case words or as the symbols themselves. 'C' indicates the occurrence of the character C itself as a terminal symbol.

   e.g. DOMAIN
        /
        'C'

4. The sequence "** . . .**" encloses a right hand side definition in plain text.

   e.g. idfr -> ** a sequence of alphanumeric characters beginning with an alpahbetic character **

5. Parentheses are used to group symbols and square brackets are used to enclose parts of a definition which may be omitted.

   e.g. equipment -> lamp pump [(bell | horn)]

   is equivalent to

   equipment -> lamp pump |
                lamp pump bell |
                lamp pump horn

6. The format

   < x S ... >

   where x is some sequence of symbols and S is a single terminal symbol (S may be omitted) indicates possible repetition; it stands for

   x | xSx | xSxSx | ...

   e.g. name_list -> < name , ... >

7. There are three hyper rules to define various alternatives for writing a list of items. Each upper case word appearing on the left hand side of the rule can be replaced on both sides of the rule by any lower case nonterminal to produce a definition of that nonterminal in standard form. The three hyper rules are:

   ELEMENT_list -> ELEMENT |
                   '(' [< ELEMENT , ... >] ')'

```
ELEMENT_SEPARATOR_pack  ->  ELEMENT  |
     '('  [< ELEMENT  SEPARATOR  ... >]  ')'

ELEMENT_LMARKER_RMARKER_SEPARATOR_package  ->  ELEMENT  |
     LMARKER  [< ELEMENT  SEPARATOR  ... >]  RMARKER
```

For example the first hyper rule could be used to specify a definition for a list of names by writing the lower case sequence "name_list". Substitution of the nonterminal "name" for the upper case sequence "ELEMENT" yields the rule:

```
name_list  ->  name  |
               '('  [< name  ,  ... >]  ')'
```

A simplified version of the BCPL [Rich73] comment facility has been provided. The sequence "||" introduces a comment extending to the end of the line in which it appears. The sequence "|* . . . *|" is a bracketed comment which may span a number of lines.

The rules for the PRODD language now follow:

```
dbs_description  ->  DBS_DESCRIPTION  idfr
                       structure_spec
                       content_spec
                       software_spec
                       hardware_spec
                       sdm_spec
                       activity_spec
                     END_DESCRIPTION

structure_spec  ->  STRUCTURE_SPECIFICATION  idfr
                      information_level
                      access_level
                      encoding_level
                    END_SPECIFICATION


information_level  ->  INFORMATION_LEVEL idfr
                         <domain_stmt ... >
                         <description_stmt ... >
                       END_LEVEL

domain_stmt  ->  DOMAIN domain_name

description_stmt ->  DESCRIPTION_SET descriptionset_name
                       <attribute_stmt ... >
                       identifiers_stmt
                     END_SET

attribute_stmt  ->  ATTRIBUTE  role_name  /  domain_name

identifiers_stmt  ->  IDENTIFIER(S)  identifier_list

identifier  ->  rolename_leftbracket_rightbracket_commasymbol_package

domain_name  ->  idfr

descriptionset_name  ->  idfr

rolename  ->  idfr
```

```
access_level  ->  ACCESS_LEVEL  idfr
                    < string_stmt ... >
                  END_LEVEL

string_stmt  ->  a_string  |
                 c_string  |
                 l_string

a_string  ->  A_STRING  astring_name
                astg_over_stmt
                under_stmt
              END_STRING

c_string  ->  C_STRING  cstring_name
                OVER  string_name
                restrict_stmt
                (key_access_stmt  |  partition_stmt)
                order_stmt
                [ring_stmt]
                under_stmt
              END_STRING

l_string  ->  L_STRING  lstring_name
                OVER  string_name_list
                match_stmt
                under_stmt
              END_STRING

astg_over_stmt  ->  OVER descriptionset_name : (ALL | avc_name_list)

avc_name_entry  ->  rolename  [IS]
                      [(LINK | DERIVED)
                       (SOURCE  [IS]  avc_name  |  function_ref)]

restrict_stmt-> RESTRICT [ON]  (function_ref | dnf_expn)

dnf_expn  ->  cnf_expn_orsymbol_pack

cnf_expn  ->  comparison_lbracket_rbracket_andsymbol_package

comparison  ->  avc_name  relop  cvalue

relop  ->  =  |  =  |  >  |  <  |  >=  |  <=

cvalue  ->  integer  |  string

order_stmt  ->  ORDER  [IS]  ( LIFO |
                               FIFO |
                               [ON]  order_avc_list |
                               OPPOSITE  [TO]  cstring_name )

key_access_stmt  ->  (PRIMARY  |  SECONDARY)
                     KEY_ACCESS  [ON]  avc_name_list

partition_stmt  ->  PARTITION  [ON]  (avc_name_list  |  function_ref)

ring_stmt  ->  RING  [TO]  string_name

match_stmt  ->  MATCH  [ON]  matchlist_equalsymbol_pack

matchlist  ->  avc_name_list
```

```
under_stmt  ->  KNOWNPOINT  |
                SEARCHPOINT  |
                SEARCHPOINT  UNDER  ostring_name_list  |
                UNDER  ostring_name_list

ostring_name  ->  cstring_name  |  lstring_name

string_name  ->  astring_name  |  cstring_name  |  lstring_name

astring_name  ->  idfr

cstring_name  ->  idfr

lstring_name  ->  idfr

avc_name  ->  astring_name  :  rolename

order_avc  ->  avc_name  /  (A | D)

function_ref  ->  RESULT  [OF]  function_name  avc_name_list


encoding_level  ->  ENCODING_LEVEL  idfr
                      < las_stmt ... >
                      < beu_stmt ... >
                    END_LEVEL


las_stmt  ->  LAS  las_name
                STORAGE_UNIT  [IS]  unit
                PAGE_SIZE  [IS]  (unit_count  |  LAS)
                DELETION  [IS]  (PHYSICAL | LOGICAL)
                PAGE_FILL  [IS]  percentage
                [PAGE_LABEL  [IS]  field_dcln]
              END_LAS


beu_stmt  ->  BEU  [FOR]  sse_name
                [derivation_entry]
                las_entry
                [head_entry]
                [align_entry]
                [label_entry]
                [(chead_entry  |  cunder_entry)]
                [< next_entry ... >]
                [cluster_entry]
                value_entry
              END_BEU


derivation_entry  ->  DERIVATION  [IS]  (VIRTUAL | IMMEDIATE | DELAYED)

las_entry  ->  LAS  [IS]  las_name

head_entry  ->  CDGHEAD

align_entry  ->  ALIGN  [ON]  unit

label_entry  ->  LABEL  [IS]  field_dcln

chead_entry  ->  COMPRESSIONHEAD

cunder_entry  ->  COMPRESSION  UNDER  sse_name


next_entry  ->  NEXT  UNDER  string_name
                  address_dcln

cluster_entry  ->  CLUSTER  UNDER  string_name


value_entry  ->  VALUE
                   vstart_dcln
                   vlength_dcln
```

```
vstart_dcln  ->  START  [IS]  (address_dcln | keyaccess_dcln | COMPUTED)

vlength_dcln  ->  LENGTH  [IS]
                    (FIXED  ([BY]  DEFN | [AT]  count_dcln)  |  VARIABLE)
                    [[IN]  field_dcln]


keyaccess_dcln  ->  FOUND  [BY]
                      (HASHING  [USING]  function_name
                                [WITH]  (PROBING | CHAINING)  |
                       INDEXING  [LEVEL  [IS]  integer]
                                 PTR  [IS]  field_dcln        |
                       BINARYCHOP  )

address_dcln  ->  [IN]  (SAMECDG  |  NEXTCDG  |  ANYCDG)
                  [PTR  [IS]  field_dcln]

field_dcln  ->  FIELD  [OF]  unit_count

las_name  ->  idfr

sse_name  ->  string_name  |  avc_name


content_spec  ->  CONTENT_SPECIFICATION  idfr
                    evaluation_count
                    [searchpoint_counts]
                    [value_lengths]
                    [deadspace_counts]
                  END_SPECIFICATION

evaluation_count  ->  EVALUATION_POINT  occurs_count

searchpoint_counts  ->  SEARCHPOINT  COUNT
                          < searchpoint_entry ... >

value_lengths  ->  VALUE  LENGTH
                     < variablesse_entry ... >

deadspace_counts  ->  DEADSPACE  COUNT
                        < deadlas_entry ... >


searchpoint_entry  ->  FOR  string_name
                         load_point
                         evaluation_stmt
                       END_FOR

variablesse_entry  ->  FOR  (cstring_name | avc_name)
                         load_point
                         evaluation_stmt
                       END_FOR

deadlas_entry  ->  FOR  las_name
                     load_point
                     evaluation_stmt
                   END_FOR


evaluation_stmt  ->  [AT]  EVALUATION_POINT  < evaluation_point ... >

load_point  ->  LOADING  [IS]  real_count

evaluation_point  ->  integer  [IS]
                        ( sign  percentage  [OF]  (LOADING | PREVIOUS)  |
                          real_count )
```

```
software_spec  ->  SOFTWARE_SPECIFICATION  idfr
                     < function_stmt ... >
                   END_SPECIFICATION

function_stmt  ->  FUNCTION  function_name
                     timing_entry
                   END_FUNCTION

timing_entry  ->  TIMING  [IS]  real_time  [+  byte_time]

function_name  ->  idfr


hardware_spec  ->  HARDWARE_SPECIFICATION  idfr
                     cpu_stmt
                     <devicetype_stmt ... >
                   END_SPECIFICATION

cpu_stmt  ->  CPU
                BYTE  [IS]  bitcount
                WORD  [IS]  bitcount
                COMPARISON  [IS]  byte_time
                COPYING     [IS]  byte_time
              END_CPU

bit_count  ->  integer  BITS

devicetype_stmt  ->  DEVICE_TYPE  devicetype_name
                       occurs_count
                       DEVICE  [IS]  integer  TRACKS
                       TRACK   [IS]  integer  BLOCKS
                       BLOCK   [IS]  unit_count
                       BLOCK_TRANSFER_TIME  [IS]  real_time
                       SEEK_TIME  seqran_entry
                       LATENCY_TIME  seqran_entry
                     END_DEVICE

seqran_entry  ->  SEQUENTIAL  [IS]  real_time
                  [SECONDARY  [IS]  real_time]
                  RANDOM  [IS]  real_time

devicetype_name  ->  idfr


sdm_spec  ->  SDM_SPECIFICATION  idfr
                < allocation_stmt ... >
              END_SPECIFICATION

allocation_stmt  ->  LAS  las_name  [IS]  [ON]
                       DEVICE_TYPE  devicetype_name


activity_spec  ->  ACTIVITY_SPECIFICATION  idfr
                     < or_stmt  ... >
                   END_SPECIFICATION

or_stmt  ->  OPERATION_REQUEST  idfr
               weighting_entry
               < route_stmt  ... >
             END_REQUEST

weighting_stmt  ->  WEIGHTING  [IS]  real
```

```
route_stmt  ->  ROUTE  route_name
                  weighting_entry
                  primary_route
                  [secondary_routes]
                END_ROUTE

primary_route  ->  PRIMARY  sse_trip

secondary_routes  ->  SECONDARY
                        < secondary_route ... >

secondaryroute  ->  FOR  string_name
                      sse_trip


sse_trip  ->  starting_point  visit_children  '=|'

starting_point  ->  (ENTRY  [AT]  knownpoint_string  |
                     SEARCH  [FROM]  searchpoint_string  selection_entry)

selection_entry  ->  SELECT  (FIRST  |  LAST  |  ALL  |  selectivity_estimate)  [OF]

selectivity_estimate  ->  (EXACTLY  |  A_PRIORI  |  A_POSTERIORI)
                          (real  |  percentage)

visit_children  ->  '=>'  (selected_instances  |  child_list)

selected_instances  ->  selection_entry  child

child  ->  child_entry  ['|']  recursive_visit

recursive-visit  ->  [[starting_point]  visit_children]

child_entry  ->  child_sse  [operation]

operation  ->  /  (ADDCDG  |  DELCDG  |  C  |  R  |  M)

knownpoint_string  ->  ostring_name

searchpoint_string  ->  string_name

child_sse  ->  sse_name


percentage  ->  real  %

units  ->  BIT  |  BITS  |  BYTE  |  BYTES  |  WORD  |  WORDS

i_unit  ->  (INSTANCES  |  units)

time_unit  ->  uSEC  |  mSEC  |  SEC

occurs_count  ->  OCCURS  integer  TIMES

unit_count  ->  integer  units

count_dcln  ->  integer i_unit

real_count  ->  real  i_unit

real_time  ->  real  time_unit

byte_time  ->  real_time  [PER]  BYTE

sign  ->  '+'  |  '-'

real  ->  integer  '.'  integer  |
          integer  '.'  |
          '.'  integer

integer  ->  < digit ... >

digit  ->  1 | 2 | ... | 9
```

```
idfr  ->  letter  [< (letter | digit | '_') ... >]

letter  ->  A | B | ... | Z | a | b | ... | z

string  ->  ** any sequence of characters excluding blanks **  |
            "  ** any sequence of characters excluding
                  double quotes **  "                          |
            '  ** any sequence of characters excluding
                  single quotes **  '

orsymbol  ->  \/

andsymbol  ->  /\

commasymbol  ->  ,
```