1986

A physical schema derivation for network databases

Leszek A. Maciaszek
University of Wollongong

A PHYSICAL SCHEMA DERIVATION FOR NETWORK DATABASES

Leszek A. Maciaszek

University of Wollongong
Department of Computing Science
P.O.Box 1144, Wollongong, NSW2525
Australia

ABSTRACT

A methodology for the design of a physical database schema in the network model environment is presented. The methodology is integrated with the conceptual and logical designs as presented inter alia in the earlier reports (Maciaszek, 1986; Maciaszek *et al.*, 1986a). As the physical design problem has the potential of being untractable in polynomial time, we have reduced its complexity in two orthogonal ways: (1) by applying a theory of separability on user views, and (2) by imposing strict ordering on design steps (with feedback). The methodology adheres to the latest standardization efforts for the network model and is consistent with the currently most recognized network Database Management Systems. The methodology is meant to serve as an initial and preliminary specification for a computer-assisted design tool, which - when integrated with other interactive tools for conceptual and logical designs - will be a component of the Intelligent Database Design Kit (IDDK). The IDDK is being developed for network and relational systems and is partly operational.

Keywords: H.2. Database Management; E.2. Data Storage Representations; H.3. Information Storage and Retrieval.

Categories: Database Design, Physical Schema, Network Model.

CONTENTS

Principle:

"A data storage description language defines how data described in a schema may be organised in terms of an operating system independent and device independent storage environment. Such a description is known as a storage schema. A storage schema has no effect on the results of application programs but only affects their performance."

Report, 1978

1. INTRODUCTION

We have been working on a methodology for database design and development. The methodology consists of five phases: (1) conceptual design (conceptualization), (2) logical design (or formalization), (3) physical design (or materialization), (4) application software design (or realization), and (5) maintenance and evolution. The methodology has been partly automated in the conceptualization and formalization phases. The prototype versions of computer-aided design tools to derive feasible conceptual structures and convert them to logical structures for network databases are operational. The materialization methodology, as described in this report, is manual and is meant to serve as a blueprint for another computer-assisted design tool. An overall interactive design support environment has been labelled as the Intelligent Database Design Kit *(IDDK)*. Starting from the formalization phase, the methodology is customized separately for the relational and network databases. Figure 1 presents the successive stages of the first three phases of the methodology for network databases. The three phases deal in fact with data structures of a database, as opposed to its algorithms. The algorithm design is handled by the fourth phase and a software conversion part of the last phase (another part addresses database restructuring and reorganization).

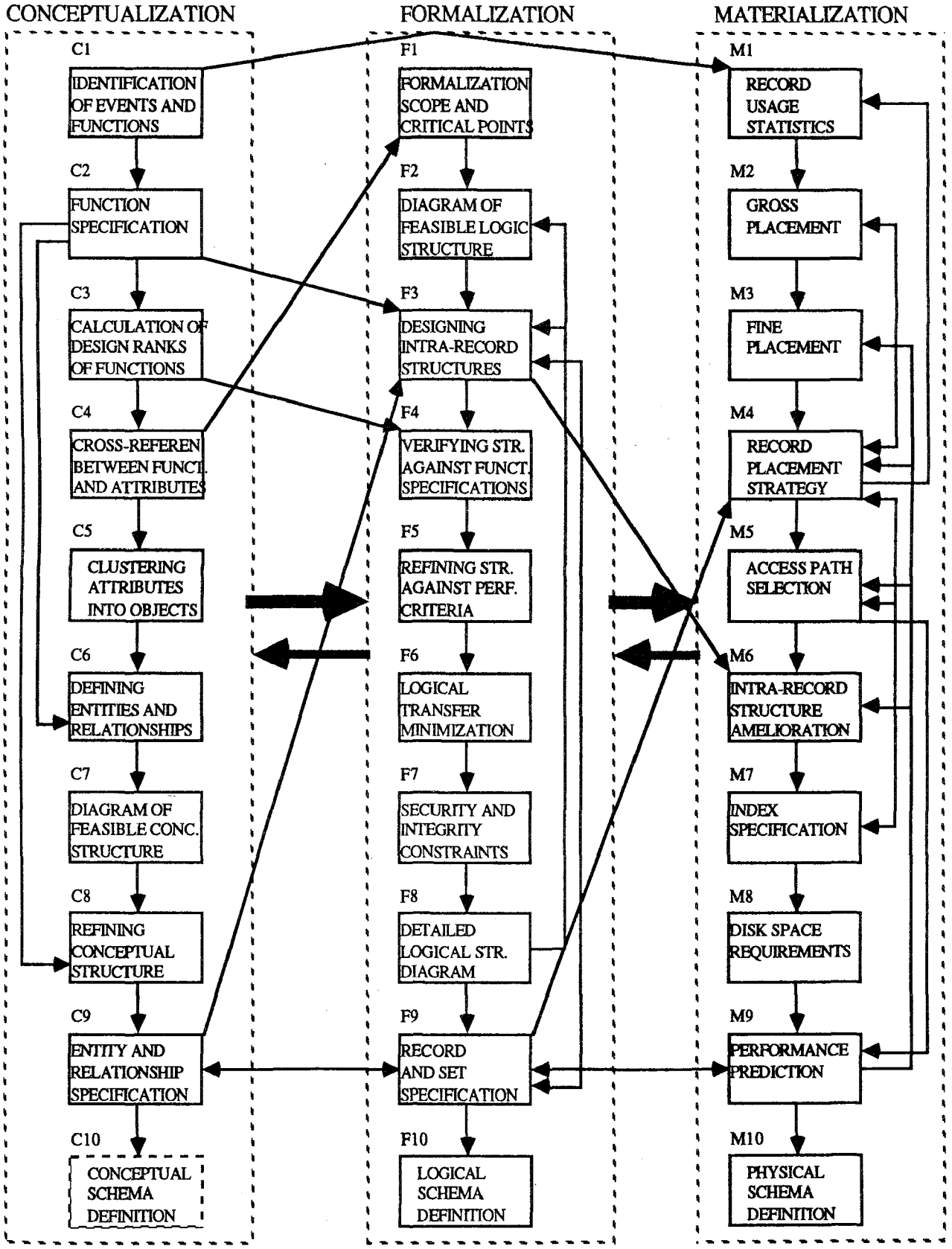


Figure 1. Data Structure Design Process for Network Databases.

A problem of physical database design (called hereafter materialization) deals with "... finding an optimal configuration of physical files and access structures - given the logical access paths that represent the interconnection among objects in the data model, the usage pattern of those paths, the organizational characteristics of stored data, and the various features provided by a particular database management system *(DBMS)*" (Whang *et al.*, 1982).

In this report we refer to an important subset of DBMS's that evolved around the network (CODASYL) database model. Network DBMS's have proved especially attractive and reliable for high-volume production applications and have formed a de facto standard for commercial databases. The first CODASYL specifications (1969 and 1971) have been followed, among others, by the comprehensive report in 1978 (Report, 1978) , two reports in 1981, and most recently by the ANSI standard draft proposal (Draft, 1985). Originally, the definition of logical and physical structures of database were combined together in a framework of a common language called Data Description Language *DDL*. However, in 1975 work began on separating logical and physical aspects of database definition in order to achieve greater data independence. This work culminated in the production of the appendix to the 1978 report in which a draft specification of a Data Storage Definition Language *DSDL* was given. As this work has been used as a blueprint for DBMS software development (most notably IDMS), we adhere to the 1978 DSDL definition in this report unless stated to the contrary (ANSI standard does not specify DSDL). We also cash in on our experiences in using the following commercial network DBMS-s: IDMS (Cullinet product that runs, among others, on IBM and ICL mainframes), DMS-1100 (Univac), VAX DBMS (DEC), UDS (Siemens) and RODAN (RIAD and IBM). Finally, we have drawn on material from Draft (1985) whenever a reference to other than DSDL interfaces has been needed.

Though we are concerned with the network database design, it is important to make the point that much of the discussion in this report is also relevant to the relational databases. The relational model is purely logical (if definition statements like CREATE INDEX are left aside) and leaves the physical considerations to the database implementors, i.e. they are hidden from a Database Administrator *DBA* but are inherent in a DBMS. Hence, one of the possible solutions is to apply physical design techniques acquired in the network database context. In fact, we currently observe a trend to implement relational database interfaces on top of the network DBMS's, e.g. IDMS/R on top of IDMS (IDMS/R, 1984) or RDMS-1100 on top of DMS-1100 (RDMS, 1985).

A significant amount of research has been devoted to the physical design of files and databases (a good survey of this research can be found e.g. in Merrett (1984), Teorey and Fry (1982), Wiederhold (1983)). The research to date has provided a variety of analytic, simulation and other tools for single-file design as well as for partial and specialized aspects of multiple-file (database) design. The usual approach employed in the developed models has been to minimize access cost over some limited solution space. Few efforts have addressed a more realistic database environment in which complex interconnections of data structures and access patterns have been considered (e.g. Jain (1984) and Whang(1985) in the case of network databases). However, these approaches again fall short of the expectations of a practically-minded DBA - the underlying assumptions are obvious tradeoffs to their applicability. (Some not exhaustive examples: Jain (1984) neither allows multiple storage records for a logical record nor spreading occurrences of a record type over more than one area. Whang (1985) assumes that occurrences of all record types are stored in one area and excludes the clustering of member record occurrences near their owner record occurrence.) As a result of the lack of design methodologies, practicing DBAs capitalize on experience and intuition and make their living by performing the pertinent cost-benefit tradeoffs (more or less on a trial-and-error basis).

To the best of our knowledge, the most comprehensive and practically viable physical design methodology for network databases has been developed within the DATAID project (Orlando *et al.*, 1985; Staniszkis and Rullo, 1982; Staniszkis *et al.*, 1982; Staniszkis *et al.*, 1983 etc.). That methodology, though slightly outdated (it is based on the early (1971 and 1973)

CODASYL specifications) is attractive for a number of existing DBMS's (the methodology uses storage formulae implemented in RODAN (RODAN, 1979)). In general, the DATAID methodology seems to be better suited for evaluation of existing databases than for designing new systems (in particular, as far as its physical part is concerned).

This report demonstrates an independent methodology carefully integrated with the IDDK. The methodology is meant to be practical, comprehensive, iterative, current and tractable by CAD tools.

One of the interesting features of our methodology is that it practically applies and extends the ideas behind the theory of separability as introduced in Whang *et al.* (1981) for the relational model and extended in Whang *et al.* (1982) for the network model. The theory says that if certain conditions hold the problem of designing the optimal physical database can be partitioned into subproblems such that the techniques developed for single-file design can be utilized to solve those subproblems. In its ultimate form, the separability property allows for reducing the global optimization design problem to a local optimization problem. We apply the theory in stage M1 of the methodology (Figure 1) by a vertical partitioning of design scope according to function specifications and by utilizing so called relative design importance (RDI) of record types. In stages M2 through M8 we extend this approach by imposing an iterative horizontal partitioning over the vertically divided design scope. Final integration of solutions is carried out during stage M9 and implemented in stage M10.

2. BASIC TERMINOLOGY

The three phases of the data structure design process refer to different - though interdependent - structuring notions. The conceptualization deals with relationships, entities and attributes. The formalization is concerned with sets, records and data-items. The materialization refers to areas, files, pages, blocks, storage records, etc. As far as the materialization notions are concerned, we further distinguish between the notions of physical storage structure and physical allocation structure. The storage structure definition is a responsibility of the DBMS and its DSDL, whereas the allocation structure is determined according to the device and file management provisions of an Operating System *OS* and its Command Language *CL*. Figure 2 shows the basic derivation dependencies between the data structure notions. The definition of notions for conceptual and logical structure is given in Maciaszek *et al.* (1986a). We presuppose the reader's knowledge of terminology relevant to the physical allocation structure, as it is well explained in many references on operating systems (e.g. Calingaert, 1982) and file management (e.g. Claybrook, 1983). On this understanding, we proceed to define the terms specific to physical storage structure for network databases.

An *area* is a largest named subdivision of the stored database. Its allocation structure equivalent is called a *file* - a portion of external storage media space controlled as an entity by the OS. In some DBMS-s (e.g. IDMS) the areas can be mapped on to the files in a variety of ways, in other systems (e.g. DMS-1100) the areas are related to the files in strictly 1:1 correspondence. At run time, one or more areas are readied in groups called *realms*. Thus, the realm is a subschema counterpart of the schema area. An area is either initialized (created) by the DBMS when it is opened for initial load or preinitialized by means of a utility routine of the DBMS. Its size is specified in the DSDL in the number of pages. The DSDL provides also for possible extensions with a specified step up to a maximum size.

A *page* is a fixed-length portion of an area. Ultimately, a page becomes a unit for transfers between external storage medium and DBMS buffer space, and it is then called a *block (physical record)*. (However, exceptions are possible. For instance, a page in VAX DBMS consists of one or more disk blocks of 512 bytes each). Thus, all pages have the same size in a given area. No doubt,

this is a severe restriction on the part of the DSDL in the light of complex distribution of record occurrences belonging to many different record types and connected by a truly network blend of CODASYL's sets. This hinders the optimization of performance, especially when several record types are hashed (calced) to the same area.

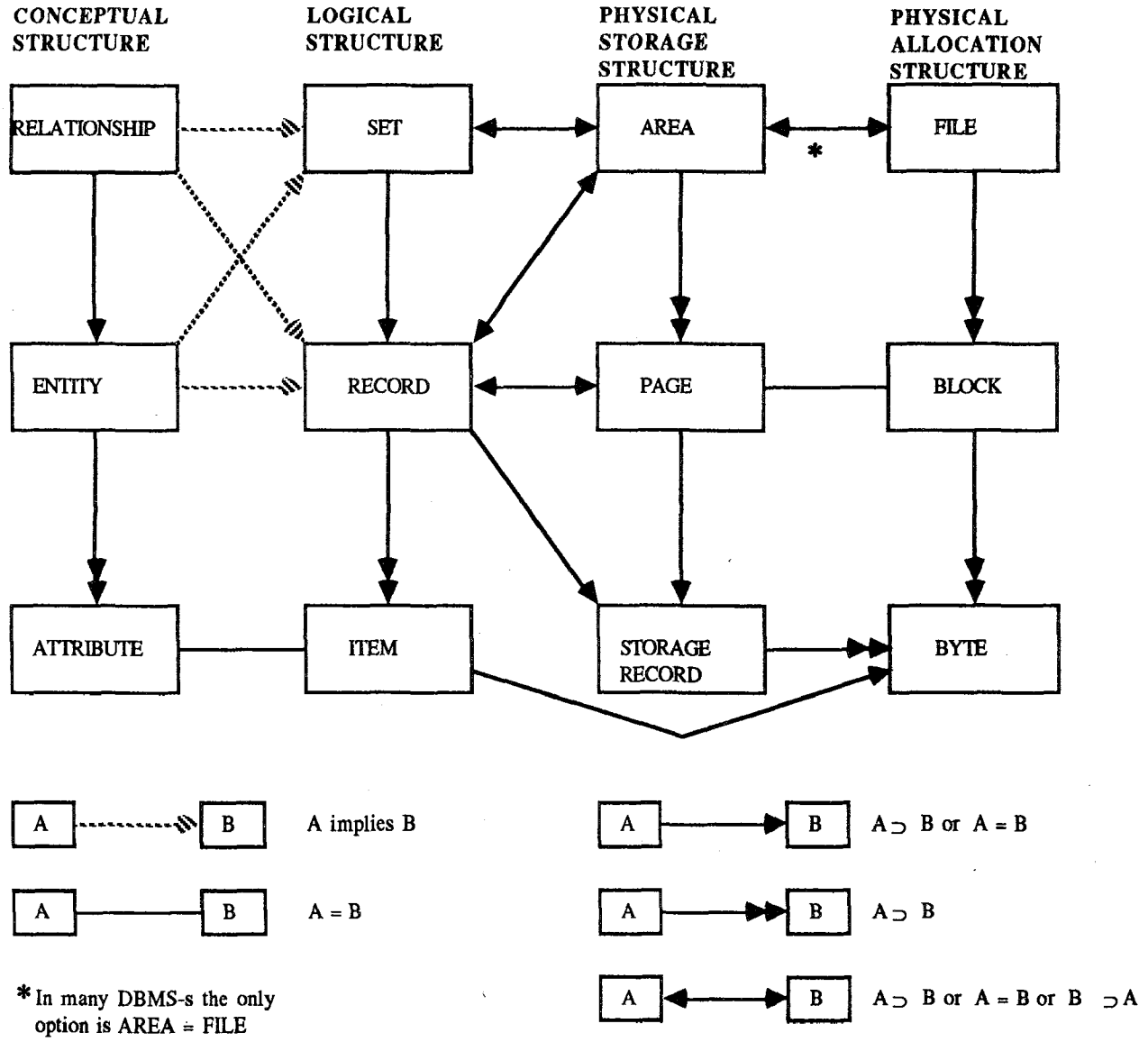
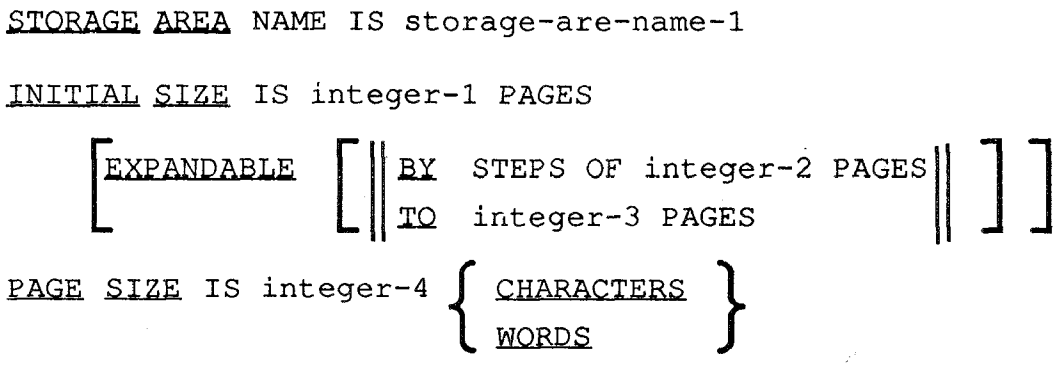


Figure 2. Derivation Dependencies Between Data Structure Notions.

The storage area entry of the DSDL is presented in Figure 3 (Report, 1978). (A DSDL is not discussed in Draft (1985).)

A *storage record* is a direct result of transforming a logical record to an actual storage format. It includes a subset (not necessarily proper, and possibly empty) of the data items from the logical record. It also contains all necessary pointers, record length information, and other control data. This implies that a given logical record type may be mapped onto more than one type of storage record. In this case, the storage records may also have overlapping (redundant) data items

from the logical record. All storage records related to a certain logical record are linked together, either by direct or indirect (i.e. through an index) pointers. The storage records can only be addressed by the DBMS, not by application programs. The DBMS is solely responsible for resolving space allocation problems caused by the growth of storage records within a page (and, possibly, for further splitting of storage records into smaller units).



[a / b] at least no occurrences, at most one occurrence (a or b)

{ a / b } at least one occurrence (a or b), at most one occurrence (a or b)

|| a / b || at least one occurrence (a or b), at most one occurrence of each (both a and b)

Figure 3. Format of Storage Area Entry.

Figure 4 represents excerpts from the DSDL entries (Report, 1978) which are relevant to the aforementioned description of the storage record.

Access paths to storage records can be supported by *indexes*, which consist of pointers and, optionally, keys. A *set* is a collection of schema records exhibiting some relationship amongst them. One record from each set is designated as the *owner* and every other record of the set is a *member* of that set. Sets are not defined as such in the storage schema by means of the DSDL, but are materialized by pointer chains and indexes, which are explicitly declared.

Before we proceed to the next section, a caveat related to the existing DBMS products. The notion of the storage record as described above has at least been implemented in one commercial DBMS (VAX DBMS). IDMS and RODAN introduce instead a concept of *fragment (segment)*. A record is fragmented if there is insufficient space available to hold the complete record on a page. The fragments are placed in more than one page and chained together. The DBA defines a minimum fragment length. The records are fragmentable only if declared as such. DMS-1100 adopts still another solution. When a record occurrence does not fit in a page, the page is compacted. Records are shifted in the page and vacant entries accumulated to make space.

We have described the basic terminology for physical storage structures. We have not attempted to address the full DSDL (and pertinent DDL) capabilities. In particular, the entries of interest to the derivation process of physical schema have not been demonstrated. We will introduce them gradually in the subsequent sections of the report.

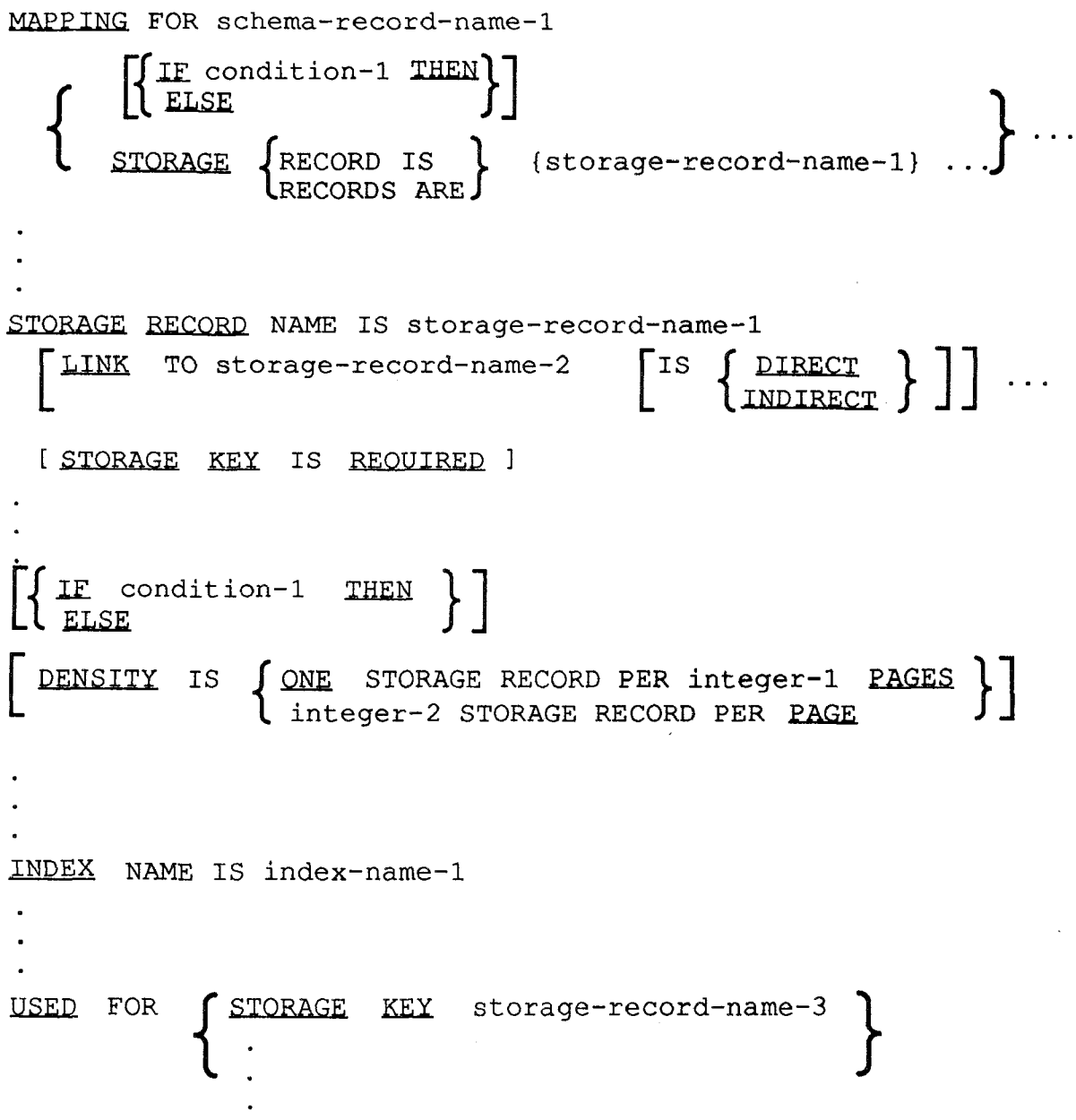


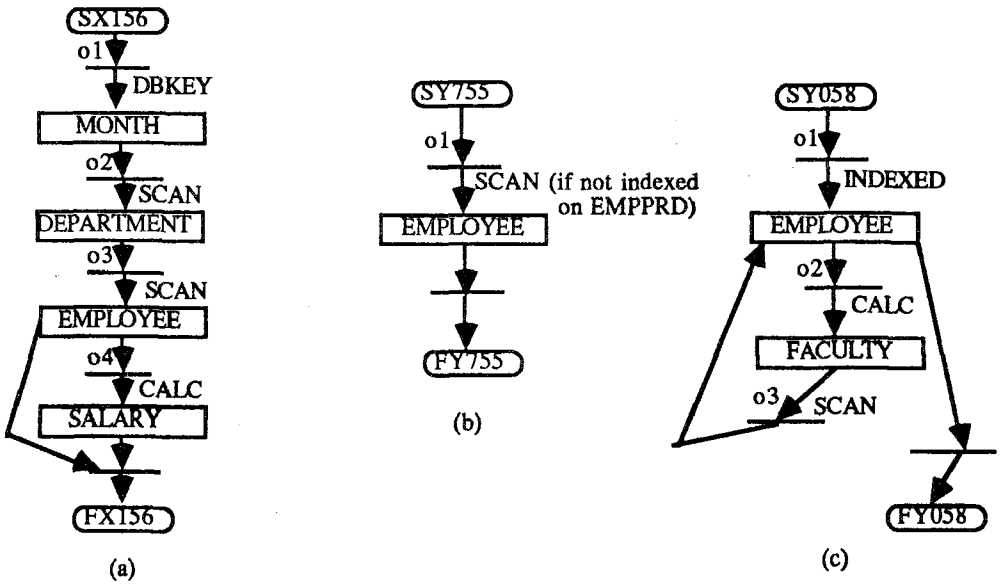
Figure 4. Format of Entries Relevant to Storage Record Definition.

3. GATHERING RECORD USAGE STATISTICS

The physical design process commences with the stage M1 in which the record usage statistics are calculated. At this initial stage, the choice of the record as a measure for usage statistics is the only practical alternative, despite the record being a logical notion. No physical notion, for example a storage record or a page, can be used because neither is defined in this design stage yet. From this point of view our understanding of record usage statistics, i.e. as a sort of interface between the logical and physical view of the database, verges upon the notion of access path as defined by Katz and Wong (1982) and Katz and Wong (1983); it also resembles a logical level traversal type of Effelsberg and Loomis (1984), a transaction definition language of Staniszkis *et al.* (1982), and search strategies of Batory and Gotlieb (1982). However, and regretably, none of

the above notions could have been used in our model directly at this stage since the technicality of the problem and its purpose are slightly different and not that of performance prediction (for the time being). We merely aim at calculating a coefficient called a relative design importance of a record in the physical design considerations.

The relative design importance *(RDI)* of a record is inferred from the logical record access counts. Note that since the intra-record structures were already determined during formalization (F3), it is easy to define access patterns of functions (transactions) to logical records. A simple abstraction process can do this job by modifying (in fact simplifying) the function specifications in such a way that they now traverse the logical records rather than attributes (C2). Figure 5 is an example of "Petri-net-like" access structures of three function specifications defined in Maciaszek (1986).



(a) EMPSLR01 - Determine the global number of employees and the total salary in a given month for each department and then list surnames, first names and monthly salaries of the employees.

(b) EMPSTF01 - Get surnames, first names, number of publications, periods of employment and ranks of the university staff members.

(c) EMPSTF02 - Enter doctors on position of a section chairman into the Faculty Council and list members of the Faculty Council according to the surnames, first names, degrees, ranks, and positions.

Figure 5. Access Structures of the Three Functions: (a) EMPSLR01, (b) EMPSTF01, (c) EMPSTF02.

However, the access graphs of functions are not sufficient to get the logical record access counts. Traversal techniques (search strategies) to individual records are also required to be known. Fortunately, this task is taken care of earlier during formalization (F5). Though these strategies are likely to be modified later in the materialization process (M4 & M5), they give a sound basis for access counts considerations at these initial materialization stages. Four traversal techniques, i.e. the ways of reaching a logical record in a database, are recognized here:

(1) CALC/DBKEY - uses a CALC key, database key, or currency indicator value;

(2) INDEXED - uses an index defined either for a set (e.g. an ordinary or index-pointer array) or for a record type (e.g. a B-tree).
(3) SCAN - accesses all the record occurrences by means of pointers within a set or by means of physical contiguity within an area.
(4) PARTIAL SCAN - similar to SCAN but terminates as soon as the criteria of searching (response set) have been satisfied.

In an integrated and shared database system a record can be retrieved in a variety of ways. SCAN and PARTIAL SCAN, when based on the physical contiguity within an area, can be applied to all database records. Member records of the sets can be always reclaimed by either SCAN or PARTIAL SCAN or INDEXED applied to those sets. If a record has been located as CALC or DIRECT, it can be accessed randomly. Moreover, a secondary search can be provided on some data items of the record (usually by means of a B-tree).

In general, the CALC/DBKEY and INDEXED techniques are oriented toward the entry point search. Thus, they mainly apply to the problem of finding an individual record fast and only rarely are used for the set traversal (i.e. as a navigational support). On the other hand, the SCAN and PARTIAL SCAN techniques are almost exclusively applied as the tools of a navigational search and are concerned with the efficient processing of a subset of member records within the set, such that the subset involved satisfies the search criteria.

Having known how an occurrence (or occurrences) of record type R_i is going to be retrieved from secondary storage by the function F_j, the DBA is in a position to calculate the expected cardinality (i.e. the number of record occurrences of R_i) in processing F_j. We shall call this measure a **search length** SL_j. In case of the CALC/DBKEY and INDEXED techniques, the search lengths are determined for both the entry point and the navigational support. In case of the SCAN and PARTIAL SCAN, we define the search lengths in terms of navigation through either the set or the area. However, we note an important difference between the SCAN and PARTIAL SCAN - the cardinality of the subset records satisfying the search criteria for SCAN can be and most commonly is greater than 1, whereas for the PARTIAL SCAN the cardinality is almost always equal to 1 (this is caused by the very nature of PARTIAL SCAN that is normally used to scan for a record occurrence responding to the search criteria and perhaps even being the entry point in the next step of the function processing - when no more efficient technique, i.e. CALC/DBKEY or INDEXED, can be applied).

The following are the search length formulas for the four traversal techniques (in the case of the CALC/DBKEY and INDEXED technique, the length of the entry point search is given on the left-hand side of the expression and the length of the navigational search is shown on the right-hand side):

(1) CALC/DBKEY: (1)

$$1 + p_o - p_b \cong SL_j \cong C_{R(i)} / 2, \text{ where:}$$

$C_{R(i)}$ - cardinality of records in areas in which R_i can be placed,
p_o - probability of overflow of calc-chain to another page (zero for DBKEY),
p_b - probability of finding the record occurrence in the buffer.

(2) INDEXED: (2)

$$H_I \geq SL_j \cong C_{R(i)} / 2, \text{ where:}$$

$C_{R(i)}$ - cardinality of R_i in the database,
H_I - height of the index or the number of secondary storage accesses required to search the

index (in fact, an assumption is made here that a master index will not require a disk access since it will have been placed in main memory when the area is opened; however, this gain of one access is neutralized later by a need to access the data area after scanning the whole index area); in general, the height of an index is on the order of $O(\log_b C_{R(i)})$, where b is the branch factor, i.e. the number of key values in a node (or, in the case of B-trees, one greater than the number of key values).

(3) SCAN: (3)

$SL_j \cong C_{R(i)} * (1 - p_r)$, where:

$C_{R(i)}$ - cardinality of R_i in the set or in the database,

p_r - probability that two or more record occurrences R_i are placed in one page.

(4) PARTIAL SCAN: (4)

$SL_j \cong (C_{R(i)} * (1 - p_r)) / 2$, where:

$C_{R(i)}$ - cardinality of R_i in the set or in the database,

p_r - probability that two or more record occurrences R_i are placed in one page.

The stage of record usage statistics is concluded by finding the **relative design importance of record types** RDI_{ij}. This coefficient has a profound influence on the further materialization stages. The materialization as a whole is centered around the record design (let us recall that even sets are materialized in records). Thus, attaching design priorities to the record types cannot be overemphasized in the process of considering design alternatives. Figure 6 presents an example in which a tabular form is used to calculate the RDI_{ij} of five record types R_i in the pre-canned function environment consisting of five functions F_j. The formulas necessary to arrive at the RDI_{ij} are also shown. RDR_i stands for the **relative design rank of a function** and expresses the relative importance of a function (transaction) in the set of functions under consideration ($\Sigma\ RDR_j = 1$). The RDR_j are calculated during stage C3 of conceptualization (Maciaszek, 1986). They lend themselves as important factors to identify and supress the effect of interferences among user views on the design process. They also represent a direct response to the requirements of the theory of separability and by leading to the vertical partitioning of a design scope make the problem tractable.

4. GROSS PLACEMENT

Two related mechanisms to control a storage scheme are recognized - gross placement and fine placement. Gross placement is aimed at designing the areas and fine placement is targetted towards the choice of a page in an area. Gross placement control involves a number of decisions, that can be classified as follows: (1) record to area mapping, (2) area to file mapping, (3) area size, and (4) page range.

The gross placement has not been treated with due attention by the researchers. Most often this topic is either neglected or only pointed out in the methodologies (including DATAID). As an exception we can mention the discussion in Teorey and Fry (1982) where a simplified heuristic algorithm, partly relevant to the problem, is investigated.

Record Types \ Functions	RR(j)	0.2	0.1	0.3	0.25	0.15	RDI(ij)
		F1	F2	F3	F4	F5	
R1	SL(j)	20	40	40		5	
	RP(j)	0.3448	0.4444	0.3809		0.2500	
	RI(j)	0.0690	0.0444	0.1143		0.0375	0.2652
R2	SL(j)	1	15	3	4		
	RP(j)	0.0174	0.3333	0.0286	0.5714		
	RI(j)	0.0035	0.0333	0.0086	0.1428		0.1882
R3	SL(j)	5		7		8	
	RP(j)	0.0682		0.0667		0.4000	
	RI(j)	0.0172		0.0200		0.0600	0.0972
R4	SL(j)	30		50	3		
	RP(j)	0.5172		0.4762	0.4386		
	RI(j)	0.1034		0.1429	0.1071		0.3534
R5	SL(j)	2	10	5		7	
	RP(j)	0.0345	0.2222	0.0476		0.3500	
	RI(j)	0.0069	0.0222	0.0143		0.0525	0.0959
Σ SL(j)		58	45	105	7	20	Σ = 1
Σ RP(j)		1	1	1	1	1	

SL(j) - average search length for R(i) in processing F(j)
RP(j) - relative priority of retrieval of R(i) with respect to F(j)
RI(j) - relative importance of R(i) with respect to F(j)
RDI(ij) - relative design importance of R(i) with respect to all functions F(j)

$RP(j) = SL(j) / \Sigma SL(j)$

$RI(j) = RR(j) * RP(j)$

$RDI(ij) = \Sigma RI(j)$

Figure 6. Tabular Aid to Calculate the RDI of Record Types (example).

The task of the record-to-area mapping is clearly related to the problem of record placement strategy (M4). It follows that the knowledge about record placement is inherent in the mapping problem and vice versa. To resolve the conflict, an iterative reasoning must be applied. At the outset, it should be noticed that the preliminary decisions on placement strategies have already been taken during the phase of formalization. Those decisions were based on the access patterns of the set of functions to the records and we can safely recognize them as a starting-point.

According to the 1978 Draft Specification of DSDL (Report, 1978), three placement strategies are available to the DBA:
1. CALC,
2. CLUSTERED,
3. SEQUENTIAL.

The placement subentry format is demonstrated in Figure 7. A word of explanation is necessary. The placement strategies in Figure 7 refer to the storage records. Therefore, in order to continue to use the DSDL of 78'Report as a basis for our discussion in this Section, we have to make clear the consequences of relating placement strategies to the logical records instead. These consequences are not too significant (after all, in CODASYL specifications prior to 1978, the placement strategy - called then a location mode - was aimed at logical records). In some rare situations, the choice of placement for storage records representing the same logical record may be made conditional on the content of the logical record. As an example, one would permit high activity storage records to be placed using a different strategy to low activity storage records. As we can neglect this possibility (without loss of generality), we feel free to discuss the placement

strategies in terms of logical records.

```
[{ IF condition-1 THEN }]
 { ELSE                }

 [DENSITY IS {ONE STORAGE RECORD PER integer-1 PAGES }]
             {integer-2 STORAGE RECORD PER PAGE      }

PLACEMENT
  {CALC [procedure-name-1] USING (identifier-1) ...                    }
  {CLUSTERED VIA SET schema-set-name-1                                 }
  {     [NEAR OWNER storage-record-name-1 ...  ]                       }
  {     [    [DISPLACEMENT IS integer-3 PAGES] ]                       }
  {    [WITH storage-record-name-2]                                    }
  {SEQUENTIAL {{ASCENDING } {identifier-2} ... } ... }
  {           {{DESCENDING }                     }   }

WITHIN storage-area-name-1
    [ FROM PAGE integer-4 THRU integer-5 ]
```

Figure 7. Format of Placement Subentry.

Since it is allowed for an area to contain occurrences of several record types and for occurrences of a record type to span several areas, a potential solution space is enormous. In fact, the number of design alternatives is the product of power sets of the set of record types and the set of areas minus a negligable constant to eliminate some repeating or useless combinations (such as no record types in the area). Thus, the number of possible mappings is proportional to $O(2^n * 2^m)$, where n and m represent the number of record types and areas, respectively.

The pertinent part of the Report (1978) definition is shown in Figure 8. Surprisingly enough, the definition is in fact an element of DDL rather than DSDL. This illustrates how difficult it is to separate some logical and physical aspects of schema definition. In Draft (1985) an area is no longer defined within the framework of DDL (called a schema definition language). However, as already mentioned, DSDL is not specified in Draft (1985) and its definition is left to the implementor of a DBMS.

```
RECORD  NAME IS record-name-1
WITHIN  {  {ANY  AREA       } [ AREA-ID IS parameter-name-1 ]  }
        {  {area-name-1}...}                                   }
        {          [USING PROCEDURE procedure-name-1 ]]        }
        {                                                      }
        {  AREA OF OWNER OF set-name-1                         }
```

Figure 8. Format of a Part of Record Subentry (DDL).

It is not difficult to conclude that the problem of record to area mapping is not manageable by well-known linear optimization techniques or other noniterative algorithms because of some conflicting conditions in the implementation alternatives (e.g. a member having two owners in two sets can only be clustered with respect to one set) and since some alternatives can override the effects of the other design choices. In these circumstances, a solution to the problem lies in a stepwise heuristic algorithm or, at the best, in techniques of integer linear programming such as branch-and-bound algorithms (originally conceived as back-track programming) and perhaps also in techniques of dynamic programming (v. Reuter and Kinzinger (1984) for some conclusions from experiences of using heuristic and analytic methods in the physical database design).

The following more or less quantitative rules-of-thumb are formulated for our heuristic algorithm (note that most of them tighten the solution space):

(1) Consider the relative design importance RDI of records in the process of record-to-area mapping (Figure 6).
(2) A cluster of record occurrences consisting of an owner and member CLUSTERED NEAR OWNER is stored within one area (though not necessarily all clusters of the pertinent set type have to be mapped into one and the same area).
(3) Record occurrences of a particular type with PLACEMENT SEQUENTIAL are stored in one area.
(4) It follows from (2) and (3) that only records located PLACEMENT CALC are candidates, albeit unlikely, to be spread over more than one area.
(5) An area does not run over more than one subschema (this ensures that only a subset of applications is affected by a failure in an area).
(6) The number of areas linked by sets should be minimized (to facilitate recovery if not for other reasons).
(7) While keeping the number of areas as small as possible (according to our experiences more than 15-20 areas per subschema introduces an inadmissible overhead on the part of the I/O transfers and memory requirements), the areas must not extend over more than one disk volume and perhaps with the load factor not exceeding 70% (for system availability and recovery reasons).
(8) By definition, the areas of MODE INDEX or POINTER are separate from DATA areas and as such they can influence rule (7).
(9) A subschema includes owner record types if the pertinent member record types are of INSERTION AUTOMATIC.
(10) An owner of singular sets is located in an area where the member record type (of one of those singular sets) having the highest relative design rank RDR is stored.

From what has been said one can conclude that the following sets of objects are involved in the process of record-to-area mapping:

(1) Set of record types:
$$R = \{R_1, ..., R_i, ..., R_r\}$$
(2) Set of areas:
$$A = \{A_1, ..., A_i, ..., A_a\}$$
(3) Set of subschemata:
$$X = \{X_1, ..., X_i, ..., X_x\}$$
(4) Set of set types:
$$S = \{S_1, ..., S_i, ..., S_s\}.$$

The cardinalities of sets R, X, S as well as the mutual connections and overlappings among those sets are assumed to be known a priori. The cardinality of A can only be determined in the process of relating records to areas. Thus, an initial design situation can be expressed in a way exemplified in Figure 9. The example comprises nine record types, n(R) = 9, twelve set types, n(S) = 12, and seven subschemata, n(X) = 7. The RDI of record types and their placements are also

shown - C stands for CALC, V - CLUSTERED VIA, S- SEQUENTIAL. Moreover the record type R_1 is named SYSTEM, i.e. it is the owner of singular sets, and the record R_2 is the AUTOMATIC member in the set type S_4.

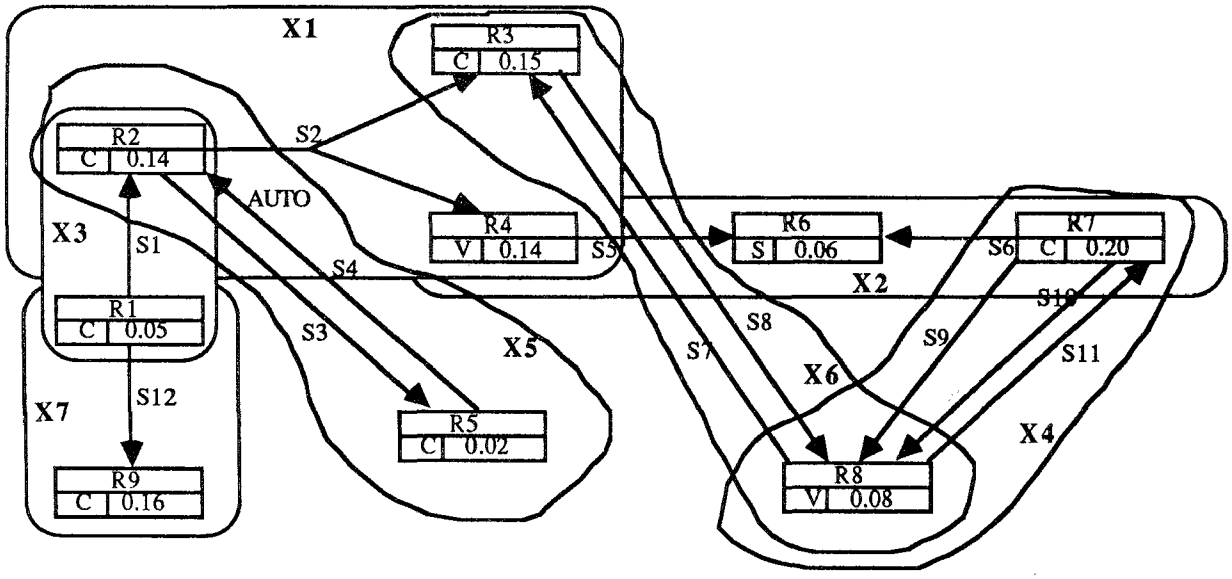


Figure 9. Initial Design Situation for a Record to Area Mapping (example).

It is our belief that the example of Figure 9 can be considered as typical for a real-life, though scaled, design problem. Therefore the example can serve as a validation vehicle of plausability of our rules-of-thumb. Perhaps the first observation should concern rule (5) - it can be seen from the diagram that this rule, though otherwise justified, cannot be complied with in the mapping process. As shown, in order to be consistent with that rule a separate area would have to be created for each of the nine record types involved. Evidently this would be contrary to the very idea of a database as opposed to a conventional file system.

However, we do not ignore the rule (5) entirely. The subschemata express collections of user functions (transactions) and in our approach to database design the user needs are of highest priority. A possible solution to this deadlock is to combine the rule (5) with the rule (1). To this end we determine **subschema-driven relative design importance of records** *XRDI* by simply adding RDI-s of all record types embraced by a pertinent subschema X_i (Figure 10).

Subschema	Records in the Subschema	XRDI
X1	R2, R3, R4	0.43
X2	R4, R6, R7	0.40
X3	R1, R2	0.30
X4	R7, R8	0.28
X5	R2, R5	0.27
X6	R3, R8	0.23
X7	R1, R9	0.21

Figure 10. Subschema-Driven RDI of Record Types (example).

As a consequence, the process of record-to-area mapping commences with the record types belonging to subschemata of XRDI = max, unless there is a contradiction accruing from rule (9). In our example the highest XRDI, equal to 0.43, is attached to the subschema X_1. Rule (9) is relevant for X_1 because the record type R_2 is the AUTOMATIC member of the set type S_4. Thus the scope of X_1 would need to be extended by including a pertinent owner R_5 or else precautions have to be taken not to run programs that store, delete or update in keys the record R_2.

We are now in a position to formulate a stepwise and recursive heuristic algorithm for a record-to-area mapping in the network database environment:

STEP1. Draw a diagram of an initial design situation as shown in Figure 9.

STEP2. Determine the subschema-driven relative design importance of record types XRDI and sort them in a descending order $XRDI_1$, ... $XRDI_x$, where x stands for the total number of subschemata and $XRDI_i \rightarrow X_i$ (i=1,2, ..., x).

STEP3. For any X_i, i=1,2, ..., x, determine the cardinalities of record types with placements:
(a) CALC - n(C),
(b) CLUSTERED - n(V),
(c) SEQUENTIAL - n(S).

STEP4. If for X_i the cardinality of CALC record types $n(C) \leq 1$, then define an area A_i as being consistent with the boundaries of X_i, i.e. $A_i = X_i = \{R_i: R_i \in X_i\}$.

STEP5. If for X_i the cardinality of CALC record types $n(C) > 1$, and all CALC record types are owners of CLUSTERED VIA or SEQUENTIAL record types invoked within the same subschema, then define an area A_i as being consistent with the boundaries of X_i, i.e. $A_i = X_i = \{R_i: R_i \in X_i\}$.

Special cases of STEP4 & STEP5 and possible iterations:

(a) If there is a reason to assign more than one area to a CALC record type as mentioned in the rule-of-thumb (4) (e.g. record STUDENT could be scattered over two areas in order to process separately the male and female students). Note, however, that such situations are considered exceptional - if each record type is assigned to one area only, then the complexity of the design problem becomes lower and it is on the order of $O(2^n)$, where n stands for the number of record types.

(b) If SEQUENTIAL record types $R_s \in X_i$ are independently enclosed by another subschema X_j, perhaps as the only record types in this subschema, then it is justifiable to separate R_s out and to locate R_s in a distinct area $A_j = X_j = \{R_s\}$. The above action is relinquished if XRDI of X_j is lesser than XRDI of X_i by an order of magnitude.

(c) If VIA record types $R_v \in X_i$ can be CLUSTERED with more than one owner record type R_o, then consider the relative design ranks RDR of all the owners involved $\{R_o\}$ and cluster R_s with respect to the owner of the highest RDR even though this owner could be outside of the X_i scope. Then apply the rule-of-thumb (2) and define the area

$A_j \supseteq \{R_v, R_o\}$. If the owner R_o is from outside of the X_i scope then define $A_i = X_i - R_v$.

STEP6. After each decision that relates a record type to an area, modify the design situation accordingly and repeat the steps 2 - 6. Refine and verify those steps with regard to the rules-of-thumb (6) - (10).

We believe that the disciplined approach presented above allows the mapping of record types to areas in the least awkward and the most fair manner with respect to the whole database user community. The following shows how this algorithm arrives at the set of areas for our example of Figures 9 and 10.

1. The Subschema X_1 has the largest $XRDI_1 = 0.43$. There are three record types embraced by this subschema $X_1 = \{R_2, R_3, R_4\}$, and n(C) = 2, n(V) = 1. Moreover, rule-of-thumb (9) is relevant to this subschema since R_2 is the AUTOMATIC member in the set S_3. After applying STEP5 and rule (9) to the subschema X_1 the defined area is $A_1 = \{R_2, R_3, R_5\}$. This situation is illustrated in Figure 11.

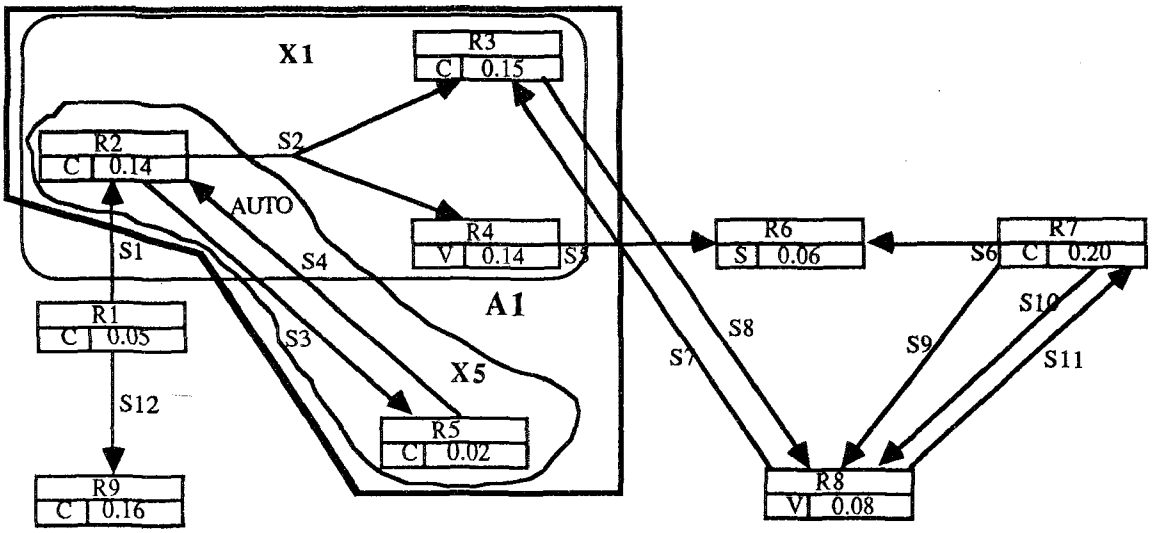


Figure 11. Design Situation After the First Iteration (Area A1 is Defined).

2. The next subschema to be considered is $X_2 = \{R_4, R_6, R_7\}$ with $XRDI_2 = 0.40$. However, the record type R_4 has already been assigned to the area A_1 and a new subschema-driven relative design importance of the remaining record types R_6 and R_7 is equal to $XRDI_2' = 0.06 + 0.20 = 0.26$. On this basis we rather proceed now to the subschema with the larger XRDI (that is the subschema X_3) and defer slightly considerations concerning X_2.

3. The subschema $X_3 = \{R_1, R_2\}$ with XRDI = 0.30. However, again the record type R_2 has already been assigned to the area A_1, thus diminishing the value of the subschema-driven relative design importance of the record types to $XRDI_3' = 0.05$. We also note that R_1 is

the owner of the two singular set types S_1 and S_{12} and the only other subschema involved is X_7. Thus, per rule-of-thumb (10), we define a new area $A_2 = \{R_1, R_9\}$. This decision is even further motivated by the SEQUENTIAL placement of R_9. The design situation is visualized in Figure 12.

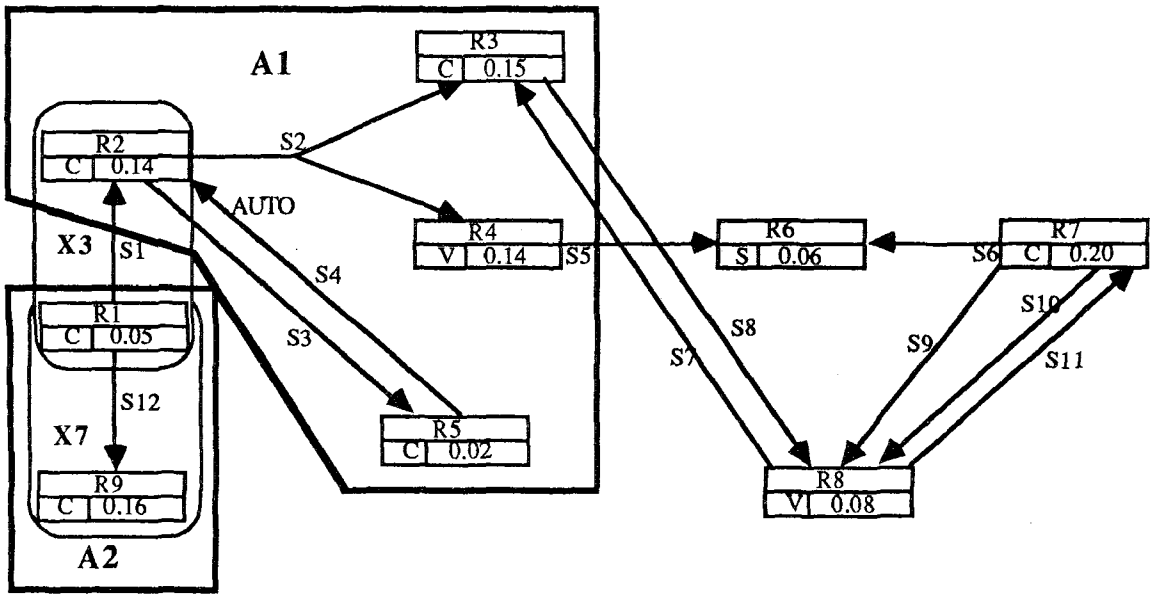


Figure 12. Design Situation After the Second Iteration (Area A2 is Defined).

4. The next subschema of interest is $X_4 = \{R_7, R_8\}$ with $XRDI_4 = 0.28$ and n(C) = 1, n(V) = 1. After applying STEP4 and considering the special case (c), another area is determined $A_3 = \{R_7, R_8\}$ (Figure 13).

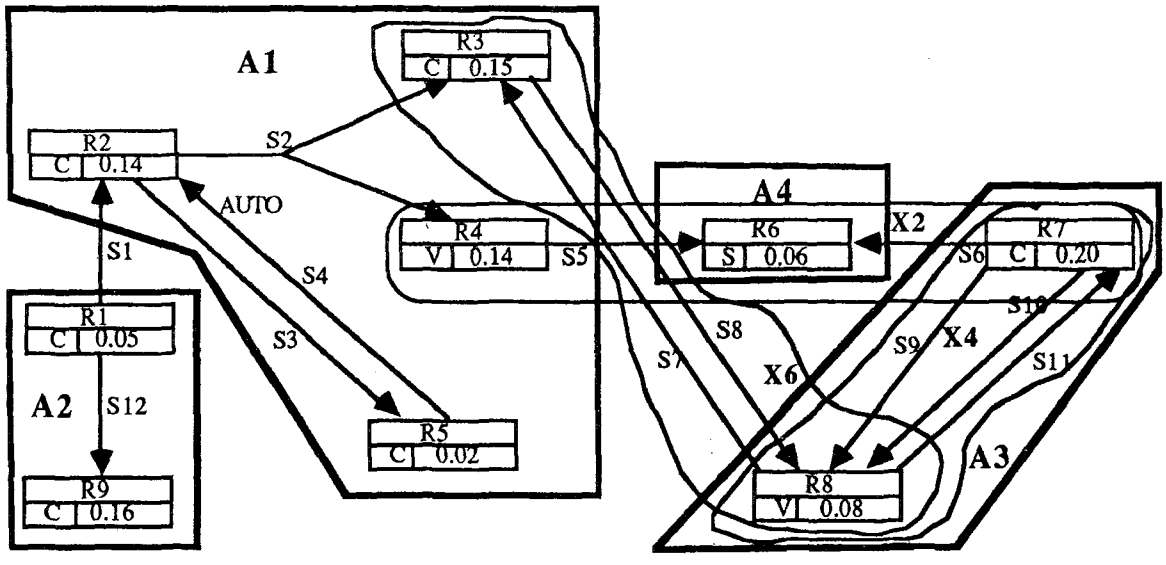


Figure 13. Final Design Solution After the Third and Fourth Iteration (Areas A3 and A4 Are Defined).

5. As a result, the only record type left, i.e. not included in an area, is R_6. Being of SEQUENTIAL placement, the record type R_6 is subjected to the special case (b) and, therefore, the last area defined is $A_4 = \{R_6\}$. Figure 13 shows the final design solution of the record-to-area mapping for our example.

Certainly, the problem of record to area mapping underlies the other problems of gross placement: area to file mapping, area size, and page range. Nevertheless, we now sketch our approach to the remaining gross placement issues.

The problem of assigning files to areas lends itself to a scheme that could be envisioned as a one to one (onto) function. Indeed, there is no reason (except perhaps for a test database) why the areas should not remain in the one to one correspondence to the files, albeit some DBMS's allow areas to be mapped to files in a variety of ways. (However, such a flexibility of a DBMS can well be deceptive and is sometimes applied just to hide away a rigid file placement policy of the OS (cp. IDMS, 1982).) We strongly advocate a one to one correspondence between areas and files not only for the sake of a design clarity but mostly in the interest of such design concerns as recovery, integrity, concurrency, and security. For example, one to one mapping from areas to files can be advantageous if distinct access priviledges are specified for each area and/or if applications are restricted to certain areas to enhance performance.

Area sizes are declared in the DSDL in terms of the number of pages they contain. However, in our systematic design approach, page characteristics are concerns of fine rather than gross placement. Therefore, the sizes of areas can be estimated only roughly at this stage. Feedbacks from the fine placement will permit necessary refinements in the stage M3.

The following expression is derived to approximate the size of a storage area:

$$AS_k = (\Sigma_{(\forall i \in k)} (RS_i + CD_i) * NRO_i) / PD_k \qquad (5)$$

where

RS_i - total length of user data items in the *i*th schema record type assigned to area *k* ;

CD_i - estimated length of control data items (mainly 4-bytes long set pointers) in the *i*th record type;

NRO_i - expected number of occurrences of the *i*th record type in area *k*;

PD_k - initial packing density of area *k* in a percentage, if applicable.

A word of explanation on packing density. Depending on the DBMS, this factor may only be applicable when the DBA chooses to not allow an area to be dynamically expandable from its initial size (or if a rigid DBMS only permits static area sizes). It is widely accepted that most areas should be 70 - 80 percent packed when they are set up, but lower densities can be appropriate for very volatile areas to which many additions can be expected.

The packing density is often specified outside of the DSDL by means of a command language of a database 'create-and-load' utility. The area extensions, however, are usually defined within DSDL (cp. Figure 3). Irrespective of the method of extension specification, there are tradeoffs. Although one can start up the database with small area sizes, the automatic expansions are prone to shortcomings with the CALC range; CALC records are likely to continue to be distributed only within the original space allocation for an area. A slow but continuous deterioration of performance can only be corrected by reorganizing and reloading the database (this problem is a subject of the last phase of our methodology).

As accruing from the philosophy of our design methodology, and from the process of record to area mapping in particular, the records will usually be allocated a full page range of their area (cp. Figure 7). We recognize, however, three cases when smaller page ranges can be desirable (cp. IDMS, 1982):

(1) to optimise access to those schema records in an area for which PLACEMENT SEQUENTIAL has been chosen and sequential access is predominant for the record type at stake (in that case a larger page size and fewer buffers may also be exercised);

(2) to simplify integrity and recovery if different record types are assigned to the same area but the user queries are not expected to process them in the interrelated fashion (i.e. the occurrences of different record types are not to be clustered);

(3) to apply different access paths mechanisms to two or more storage records as a result of the fragmentation of a logical record (this could lead, for instance, to a different placement technique for high and low activity storage records and would also benefit the dumping, archiving and recovery processing).

5. FINE PLACEMENT

Fine placement control involves the choice of a page in the storage area. This implies a need for the design steps to determine: (1) page size, (2) record to page mapping, and (3) storage record to record mapping.

The size of a page, which is fixed within a storage area, is specified in terms of characters or words. This decision depends both on the allocation of records to the storage area and the physical characteristics of the storage media (e.g. the capacity of a disk track). The latter aspect, when taken to its logical conclusion, makes the discussion below more relevant for block-addressable as opposed to sector-addressable devices.

The problem of assigning records to pages is NP-hard. Essentially, the problem is to assign the records to the pages in such a way that the total number of pages accessed, in response to a set of user queries with various probabilities of submission, is minimized. A recent paper by Yu *et al.* (1985) describes an interesting adaptive clustering algorithm to solve this problem. Unfortunately, under the unacceptable, for network databases, assumption of the same length for all records. Another recent paper due to Lirov and Daunov (1985) uses simulation instead of an analytic approach. However, the underlying assumptions are even stronger - the logical schema is "leveled" in a hierarchical fashion such that, effectively, only CLUSTERED VIA OWNER placement strategy is considered.

Determining a page size involves several complex parameters that in practice are tradeoffs to each other. The most important tradeoffs are listed below(cp. IDMS (1982), Lirov and Daunov (1985), Staniszkis *et al.* (1983), VAX (1984)):

1A. A larger buffer pool can result in performing virtual paging of the pool (thrashing), thus causing an additional I/O by the OS to page fault the buffer into the working set (apart from an I/O by the DBMS to read the page into the buffer pool).

1B. A smaller buffer pool can result in more database I/O.

2A. A larger buffer size allows more data to be read with each I/O. This is usually advantageous for processing CLUSTERED and SEQUENTIAL records.

2B. A smaller buffer size means a larger buffer number for the same buffer pool. This increases a chance for the DBMS to retain previously referenced records in its buffer pool for a run unit. This, in turn, can decrease an overall number of I/O if the database contains many CALC records.

3A. A larger page size is convenient for processing CLUSTERED and SEQUENTIAL records (time to transfer a page tends to be small compared with the time to search the disk, which includes queueing time, seek time, and latency). However, larger page sizes increase CPU overhead due to looking for a requested record, searching for a free slot in the page to store a record, or page shuffling when record size changes.

3B. A smaller page size increases the likelihood of record fragmentation (not to be confused with a deliberate fragmentation of a logical record to a number of storage records).

A page is a DBMS equivalent of an OS block (with the exception of VAX DBMS - v. Section 2). It follows that pages are stored in blocks and although a block size may in special cases exceed a page size this is of no concern to the DBMS (and to the DBA at large). Typically, a page format is as shown in Figure 14 (cp. DMS (1984), IDMS (1980)).

We propose an alternative method to handle fine placement which is tailored to the physical storage structure (rather than to the physical allocation structure and the interfaces between storage and allocation structures) (v. Figure 2). Therefore, we do not refer directly to such notions as blocks or buffers. Moreover, we consider that a page size and record to page mapping are inseparable issues and they are treated accordingly.

Our approach to record to page mapping and to selection of a page size is based on two constraints (which are usually an implementor's recommendations):

1. The maximum page size must not exceed a disk track length.

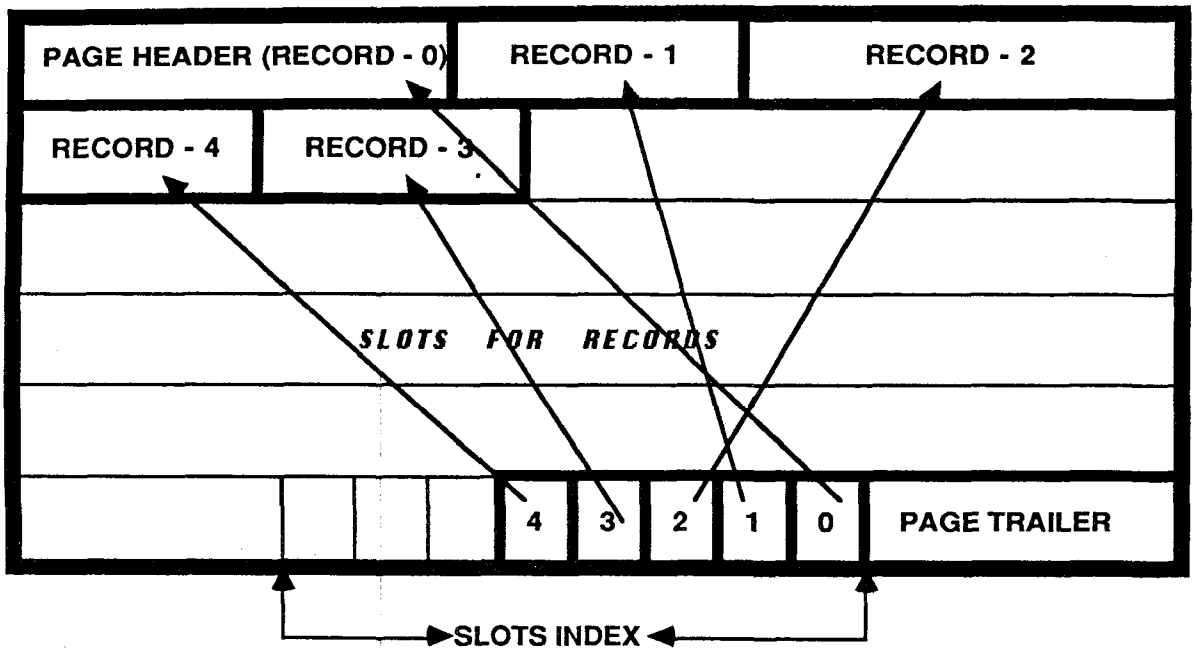
2. The minimum page size should accomodate the largest record from among the record types assigned to the pertinent area (plus the lengths of page header, page trailer and slots index) and should yield an integral number when divided into the disk track length.

A page size is a function of some average "basket" of records per page, that is determined by considering: (a) the expected configuration or pattern of records stored and processed in a dependent or clustered fashion, and (b) the average length of records. The existence of the former feature (i.e. clustering) establishes a 'recognizable pattern' (i.e. the physical ordering) of records in an area. If such a pattern is not known then a trial and error extension of a multiple size analysis technique by Oberlander (v. Teorey and Fry, 1982) is proposed. We shall call it a 'non-recognizable pattern' problem. In both cases, an average time until page overflow can be considered as an important design hint (cp. Heyman (1982), Cooper and Solomon (1984)). In what follows a recognizable pattern problem is discussed first.

A *recognizable pattern problem* will typically fall into one of the following categories:

1. There is only one record type in the area. The record size distribution is known for the record type of variable length. If the distribution is not known the uniform distribution can provide a good approximation.

2. There are *n* record types in the area but all of them placed as SEQUENTIAL and the loading strategy is known.



PAGE HEADER - Typically 24 - 40 bytes of control information (page number, pointers to the beginning and end of the calc-chain(s), overflow usage, etc.).

PAGE TRAILER - Typically 8 bytes of further control information (page number, slots index length, page trailer length) plus 4-bytes pointers to calc-chains allocated in excess of the first one.

SLOTS INDEX - Expandable index of typically 4 to 8-bytes entries, one entry for each record on the page and occasionally for records stored externally (pointer to the record, record identifier, record size, length of the record prefix).

SLOTS FOR RECORDS - A variable number of slots holding records (or fragments, or pointers), possibly followed by the free space.

NOTE: RECORD-3 follows RECORD-4 due to updates. RECORD-4 was shifted up in the page after some RECORD-X was physically erased (usually the record is physically erased immediately after DELETE is executed if the DBMS knows PRIOR records to the object record in the sets in which the object record (i.e. RECORD-X) participates, or it is physically erased with some delay, when the set is next traversed and the necessary areas are opened for UPDATE usage). The position of slot number in the slots index is the same, thus the database-key of a shifted record does not change.

Figure 14. Page Format.

3. The member record types in the area are CLUSTERED NEAR one OWNER without DISPLACEMENT. (The clustering with DISPLACEMENT is a non-recognizable pattern.) An owner record type can have any PLACEMENT. The loading strategy and a distribution of set lengths are known - especially a minimal cardinality or, better, the mode of cardinalities.

4. This case is a combination of the last two categories. A recognizable pattern algorithm can be applied as long as the loading strategy is clear.

5. The member record types in the area CLUSTERED NEAR *k* different OWNERS and the owner record types have no overlapping page ranges. Alternatively, the member record types in the area are CLUSTERED WITH the first member record stored (which in turn is CLUSTERED NEAR owner) and all such "first members" have no overlapping page ranges. For both cases, the loading strategy and distribution of set lengths are known. This is practically a generalized case of Category 3.

The challenge now is to determine a stepwise and iterative heuristic algorithm for the above categories of recognizable pattern problem. For simplicity we assume that record types are not spread accross more than one area. As an aside, we note (and it is not suprising) that the recognizable pattern problem reveals a consecutive retrieval property (except for Category 1 when the object record is located by CALC). It follows, therefore, that due to the performance characteristics of secondary storage devices the minimization of the number of page accesses should have the higher priority than the transfer overhead. Thus, the page sizes for the recognizable pattern problem should be relatively large.

We propose the following heuristic algorithm to choose a page size for a recognizable pattern problem:

STEP1. Consider cardinality of record occurrences of record types NRO_i. Determine the modal group of cardinality of record occurrences in the sets MRO_j by assuming a grouped frequency distribution with the intervals equal to five (say). However, if the grouped distribution is multimodal, then calculate instead an average number of record occurrences in a set type $ARO_j = NRO_m/NRO_o$, where NRO_m and NRO_o, respectively, are the expected number of member (m) and owner (o) occurrences of set type (j). Define also the lengths of records: $RL_i = RS_i + CD_i$. For variable length records, an average record length is assumed (unless there are special pros or cons regarding fragmentation, thus justifying lengths longer or shorter than average).

STEP2. Consider the possible loading strategies. If more than one loading strategy is applicable try and choose the one which is most attractive from a viewpoint either of loading efficiency or record sequence as requested by some set of functions with the highest design ranks (v. Maciaszek, 1986).

STEP3a. For Categories 1 through 3 choose the page size by adding the lengths of records in a basket (as indicated by a pertinent Category of the recognizable pattern problem) such that the page size is approximately equal to the mid-range between the smallest MINPS and the largest MAXPS physically permitted page sizes, i.e.:

$$MR = (MAXPS + MINPS) / 2 \qquad (6)$$

$$IPS = \Sigma_{(i=1...n)} RL_i \quad \text{where n is such that IPS} < MR \qquad (7)$$

$$EPS = IPS + PH + PT + SI \approx MR \quad (8)$$

where:
MR - mid-range between MAXPS and MINPS,
$RL_i = RS_i + CD_i$ - lengths of records in the basket,
IPS - intermediate page size,
PH - page header length,
PT - page trailer length,
SI - slots index length,
EPS - effective page size.

STEP3b. For Categories 3 through 5 a page size, which is based on the sum of record lengths in the modal or average set length, ought to be taken. The sum cannot, however, exceed the maximum page size MAXPS, i.e.:

$$IPS = \Sigma_{(\forall i \in MRO(j))} RL_i \text{ or } \Sigma_{(\forall i \in ARO(j))} RL_i \quad (9)$$

$$EPS = IPS + PH + PT + SI \leq MAXPS \quad (10)$$

As mentioned, for a *non-recognizable pattern problem* we propose an algorithm which basically is a trial and error extension of a multiple record size analysis technique due to Oberlander (cp. Teorey and Fry, 1982). (Note that Oberlander's algorithm was used for a different purpose; computing the expected value of used space in each block of the database in order to get a database size.)

There is quite a number of possible categories for the non-recognizable pattern problem - the simplest case being such that there is only one record type in the area but neither the distribution of record sizes nor the expected record size is known. Record types with PLACEMENT CALC will dominate in this pattern. Therefore, due to the performance characteristics likely to be revealed in such cases, the page sizes should be relatively small in order to decrease the transfer overhead.

A trial and error algorithm for a non-recognizable pattern problem is now proposed:

STEP1. Consider cardinality of record occurrences of record types in the area NRO_i. Assume that the size of any record occurrence in the area is independent of its neighbours. As a result, assume the uniform distribution of record sizes with a mean record size μ_i (i denotes record type).

STEP2. Calculate the mid-range between the minimum page size MINPS and the mid-range of the minimum MINPS and the maximum MAXPS page sizes, i.e.:

$$IPS = (MINPS + (MINPS + MAXPS) / 2) / 2 \quad (11)$$

Use the intermediate page size IPS as the first approximation of a final page size. (However, if MAXPS >> MINPS or an overwhelming majority of record types are placed CALC, an alternative method of computing IPS can be chosen, not excluding IPS = MINPS.)

STEP3. Apply Oberlander's technique by enumerating the possible arrangements of records in a page of IPS size. (This can be done manually with ease because the computational complexity is here polynomial and only if the number of record types exceeds 10 (very unlikely in practice!) or when IPS >> RL_i, would a computer implementation of the

algorithm be needed.) Determine the probabilities of arrangements P_i, i = 1,2,...,n and used space on the page for each arrangement $USED_i$, i = 1,2,...,n. Then compute the expected value of used space on the page by the formula:

$$EUSED = \Sigma (P_i * USED_i) \quad (12)$$

Figure 15 gives further explanation by means of the example.

STEP4. Repeat ***STEP2*** and ***STEP3*** for IPS = (IPS - NUSED)+ $min\mu_i$ and for IPS = (IPS - NUSED) - $min\mu_i$, where NUSED is the length of never used space for any page arrangement (v. Figure 15) and $min\mu_i$ is the minimal mean record size.

STEP5. Choose the IPS which yields the largest ratio of used to unused space in a page. Compute EPS = IPS + PH + PT + SI, where PH, PT, and SI are determined on the proportional basis of the number of record occurrences of record types involved. Round EPS in order to achieve an integral number of EPS on a track.

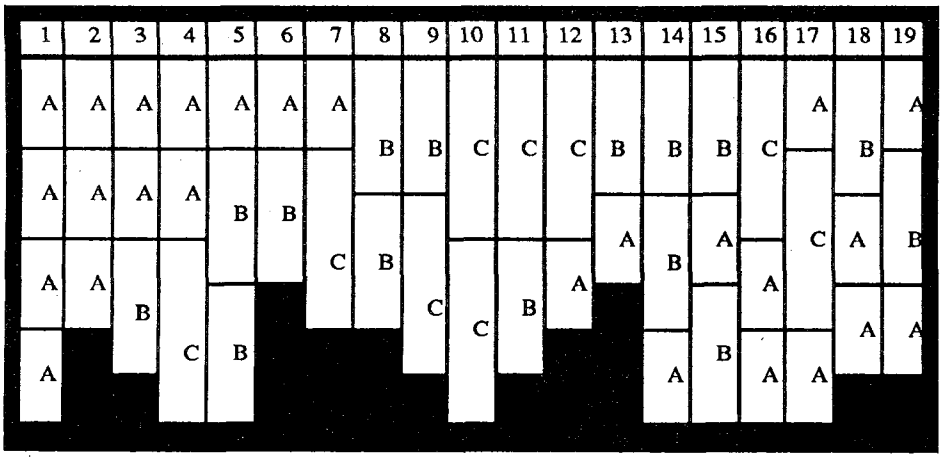
In the foregoing discussion, we have more or less tacitly assumed the one-to-one correspondence between a record and a storage record. Although this assumption can be expected to hold in most realistic databases, there are a few cases where representing a logical record type by more than one storage record type is desirable. For a particular mapping, the contents of the storage records may be disjoint or overlapping with the logical records. The pertinent mapping is determined based on a condition specified in the DSDL (Figure 4). In general, the mapping onto two or more storage record types may be useful to filter out highly confidential or highly volatile data. It will almost always penalize update operations and favour retrieval and recovery processing.

While the assumption of equivalence of logical and storage records is slightly crude, our stepwise derivation process and practical circumstances make it justifiable. After all, addressing the problem now would cause a need for immediate feedback (and refinement) to most of the issues of gross and fine placement. Moreover, there is a limit to the number of questions we can tackle at once, and these problems about multiple storage records are not the most pressing here. Nevertheless, we will come back to the issue in the later design stages and will emphasize feedback requirements (Figure 1).

In the meantime, however, we would like to point out that analytical models of structuring database records are available and are now accurate enough to be used as blueprints for the IDDK. What remains to be done is to integrate and customize them in the overall methodology aimed towards network databases. We believe that the main burden of this activity should rest with the performance prediction stage (M9). An overview of record structuring techniques is given in March (1983), a structuring due to different volatility of data (80-20 principle) is discussed in March and Severance (1977), and an impact of record segmentation on recovery processes is dealt with in March and Scudder (1984).

RECORD TYPE	BYTES	n(R)	P(record)
A	1000	200	2/5
B	1500	100	1/5
C	2000	200	2/5

MINPS = 2600 bytes
MAXPS = 9000 bytes
IPS = 4200 bytes



} - NUSED

CASE	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
NEXT RECORD IN RANDOM SEQUENCE	A B C	B C	A B C	A B C	A B C	C	B C	B C	A B C	A B C	A B C	B C	C	A B C	A B C	A B C	A B C	A B C	A B C
P(CASE)	16/625	24/625	4/125	8/125	2/125	4/125	12/125	3/125	2/25	4/25	2/25	12/125	4/125	2/125	2/125	8/125	8/125	4/125	4/12
USED (in thousands)	4	3	3.5	4	4	2.5	3	3	3.5	4	3.5	3	2.5	4	4	4	4	3.5	3.5

EUSED = 3500 bytes

Method of Calculation (Examples):
P(2) = P(AAA$\overline{A}$) = 2/5 * 2/5 * 2/5 * 3/5 = 24/625
P(6) = P(ABC) = 2/5 * 1/5 * 2/5 = 4/125

Figure 15. Example of STEP2 and STEP3 of the Algorithm for Non-Recognizable Pattern Problem.

5. RECORD PLACEMENT

In Report (1978) the control over record allocation was removed from the DDL (where it was called LOCATION MODE) to the DSDL (PLACEMENT subentry - Figure 7). The main consequence of this change is that the record placement is applied to storage rather than logical records. With this in mind, in the gross and fine placement designs (Section 4 and 5) we assumed the DBA's preliminary knowledge on placement strategies for logical record types. (In fact, this knowledge is also indispensable for many formalization stages (Figure 1)). Naturally, storage record placements chosen in this stage (M4) may imply different logical record placements to those used in the stages M2 and M3 (let alone the formalization phase). If such is a case, the design decisions of stages M2 and M3 have to be iteratively validated and possibly modified.

The decision concerning placement of a record occurrence is based on the record's access patterns. However, this decision has to be carefully integrated with the storing procedures of the DBMS and the database loading strategy as determined by the user.

To store a record occurrence in the database, the DBMS first determines a target page according to the placement strategy for the record type. Therefore, it is important for the DBA to be aware beforehand about the DBMS operations to store the record occurrence in the page and wisely influence the choice of the target page. In particular, the storing procedures will set and/or modify a number of pointers in the prefix of the subject record occurrence and in the related record occurrences. For the sake of clarity, we distinguish the following kinds of pointers:

(1) **placement pointers** (calc-chain or record key index pointers),

(2) **link pointers** (direct or indirect (storage key index) pointers),

(3) **fragmentation pointers** (segmentation or division pointers in the terminology of Batory's Transformation Model (Batory, 1984)),

(4) **set pointers** (chain pointers or set index pointers).

Among the kinds of pointers listed above, only the fragmentation pointers are not controlled by the DSDL statements. The placement pointers are implicitly subjected to the placement subentry (Figure 7) and the index entry (Figure 16). The link pointers are govern by the storage record subentry (Figure 4) and the index entry (Figure 16). The set pointers are controlled by the set pointer subentry (Figure 17) and the index entry (Figure 16).

Calc-chain pointers originate in a system-owned record type kept in the page header. In practice, this system record type is the owner of the special CALC set. If the hashing algorithm randomizes to a page number, the members of the CALC set are occurrences of all the record types that have the placement CALC. If the hashing algorithm randomizes to a slot (line) number, the members of the CALC set are the synonym (overflow) record occurrences. (Depending on the system, the CALC set can only have one occurrence per page or can have a distinct occurrence for each member record type placed CALC.) In the former case, the search for a record occurrence is entered in the owner CALC record occurrence and follows the calc-chain until the record occurrence is located, or until a record occurrence of the same type with a higher database-key value is located (record-not-found). In the latter case, the search begins in the slots index and continues - if the record occurrence overflowed - in the calc-chain. In both cases, the search can involve I/O for other pages if the record occurrences in the calc-chain overflowed beyond their home page. It should be emphasized again that the DBA's control over calc-chain is only implicit - the CALC set types are neither declared in the DDL nor explicitly referenced in the DSDL.

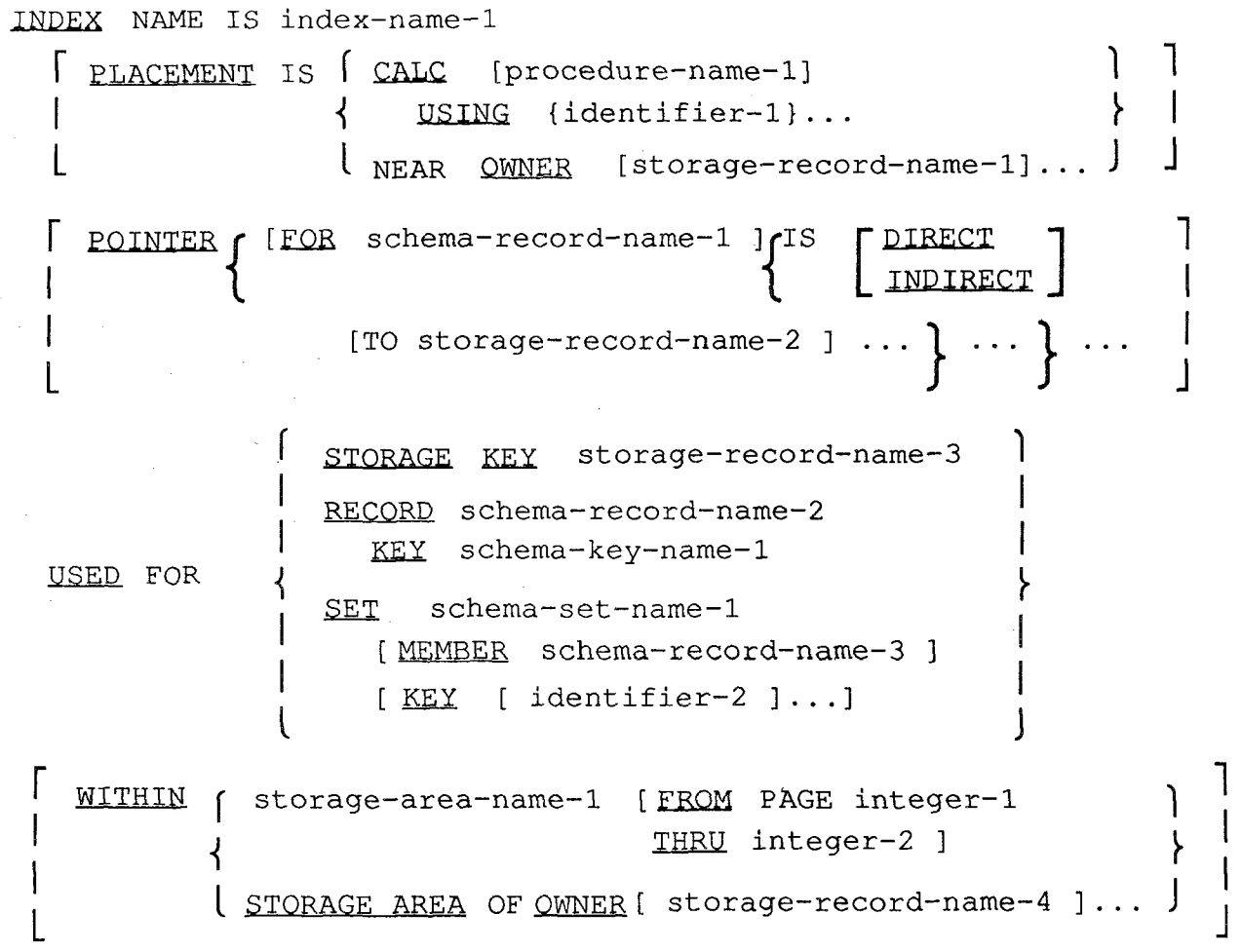


Figure 16. Format of Index Entry.

Record key indexes support record keys defined in the logical schema by means of the DDL. In the absence of an indexed placement strategy in Report (1978), the record key indexes are used for the secondary (possibly order) keys. Subject to the indexing technique applied (Stage M7), the data records may or may not have supporting pointers (for example, to handle overflow in a way similar to IBM's ISAM). In general, however, the question of record key index pointers in the data records is either negligible or even not existent. As far as Stage M4 is concerned, of much more significance is the placement strategy applied to the index records themselves (Figure 16) and how this strategy interferes with the placement of data records (ref. Section 9).

Link pointers are incorporated in the storage records whenever more than one storage record constitutes a logical record. Link pointers allow easy retrieval and processing of the entire logical record. They can be either direct (thus creating a chain of storage records) or indirect through a one-level index (that is, pointers in the storage key indexes must be direct). As in the case of the record key indexes, the link indexes must be placed within a DBA-specified page range of a storage area (Figure 16). (Incidentally, the same is true for the indexes supporting singular sets.)

Fragmentation pointers are not controlled by the DBA. In a practical system the records defined as variable-length can be subjected to fragmentation. This can happen when the entire storage record occurrence does not fit on the page and it has to be fragmented and stored on more

than one page. In such a case, the DBMS will place extra pointers in the prefixes of fragments in order to chain them together. After Batory (1984), we distinguish between the segmented and divided fragments (division, unlike segmentation, is done without respect to the data-item boundaries). Naturally, the need for variable-length records in a database can be alleviated in the first place by the fact that the logical records can be mapped to more than one storage record.

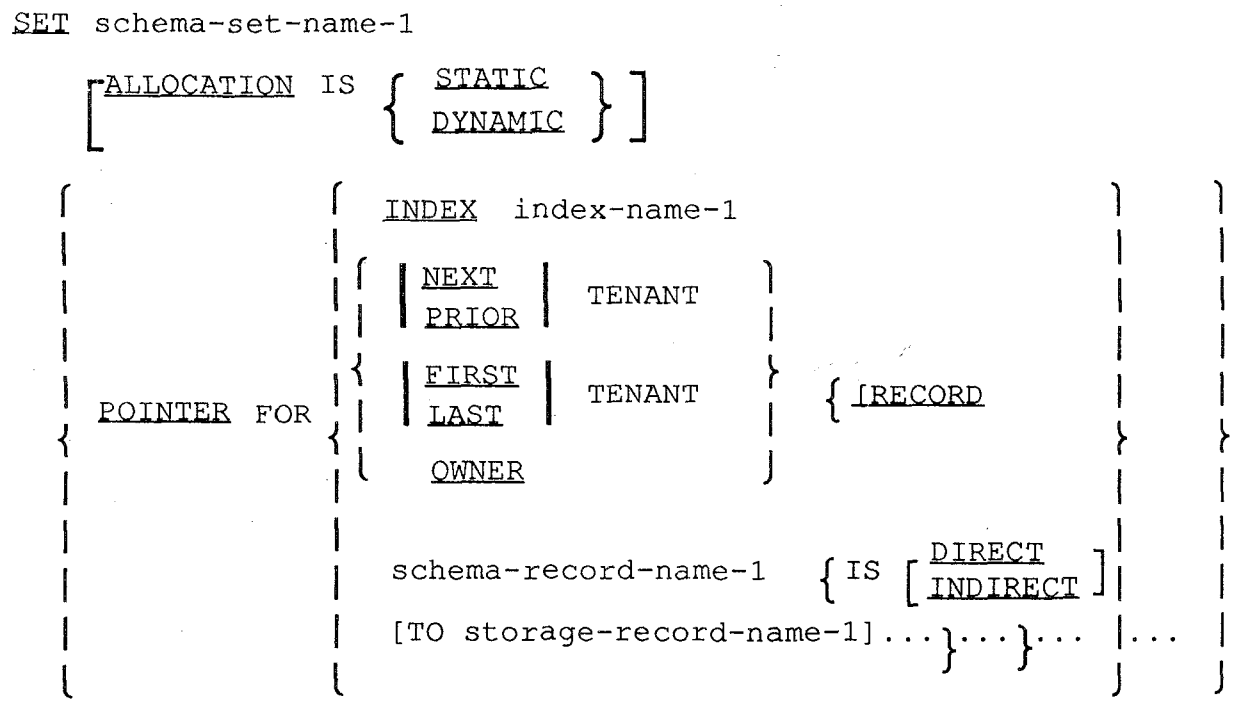


Figure 17. Format of Set Pointer Subentry.

Set pointers deserve a special consideration as factors in the process of record design. They tend to occupy the largest portion of a record prefix, and sometimes of an entire storage record. Their role and number depend on the set implementation (Figure 17). The two set modes are distinguished: chain and pointer-array (we prefer not to use again the term "index" for obvious reasons; v. Section 9). In either case, an owner can contain forward (FIRST) and reverse (LAST) pointers; the members - forward (NEXT), reverse (PRIOR), and back-to-owner (OWNER) pointers. A set in the pointer-array mode will additionally consist of an index built on sorted pointers (database-keys) or on record-key - pointer pairs. We call the former the **ordinary pointer-array**, the latter - the **keyed pointer-array**. The keyed pointer-array can be, and usually is, sorted on a record key (sort key). The pointer-array mode usually compels the DBA to also specify the owner pointers.

The database loading strategy is another factor to be considered in choosing the placement of record types. Ideally, the loading strategy should be envisioned or even decided upon in parallel with the record placement decisions. Its importance cannot be overemphasized in the database situation - the sequence in which record occurrences of various types are loaded is just as responsible for the final distribution of records in the database as the record placement strategies themselves. It is our opinion that the loading process is by far the most difficult and complex activity in the database processing (in which, incidentally, UPDATE is the only valid access mode). As such, it must be treated with due attention and supported by the DBMS Load/Unload facility. In the remainder of this Section, however, we only implicitly address the loading issue. An explicit

discussion would only be possible after giving first consideration to the loading process per se. As this is one of the main subjects in the next design phase (realization), we choose to defer the integration problem between the record placement and the loading strategy until our stand on the latter issue is clearly declared and described.

We are now in a position to formulate the principles for revising our logical record placement decisions and for specifying storage record placements. We repeat that three mutually exclusive strategies exist (cp. Figure 7):
1. CALC (C),
2. CLUSTERED (V),
3. SEQUENTIAL (S).

It turns out, and it is not fortuitous, that the CLUSTERED placement consists of many options, motivated by the coherence of this placement with set definitions. The cluster of records is chosen according to the set owner or to the first member of the set. In the latter case, the placement of the first member may or may not be specified relative to the owner. The CALC and SEQUENTIAL placements are determined independently of a record participation in the sets.

A subconclusion is evident - the record types which are processed predominantly in a set-oriented (navigational) fashion are candidates for CLUSTERED placement and the record types exhibiting a record-oriented (entry-point) processing are placed as CALC or SEQUENTIAL. Hence, the first problem to overcome is to determine quantitatively for each record type its predominant usage pattern - navigational or entry-point. To this aim, we first refer back to stages C2 and C3 of conceptualization (Figure 1) in order to consider again the function specifications and their relative design ranks RDR (Maciaszek, 1986). Then, we refer to stage M1 in order to reconsider the access structures of the functions (Figure 5) in terms of storage records and set definitions. (The placement strategies in access structures of M1 are subject to modification as a result of M4 - the modification will involve an iterative design refinement in stages M1 through M3.) On this basis, we construct for each individual record type its usage graph as shown in Figure 18.

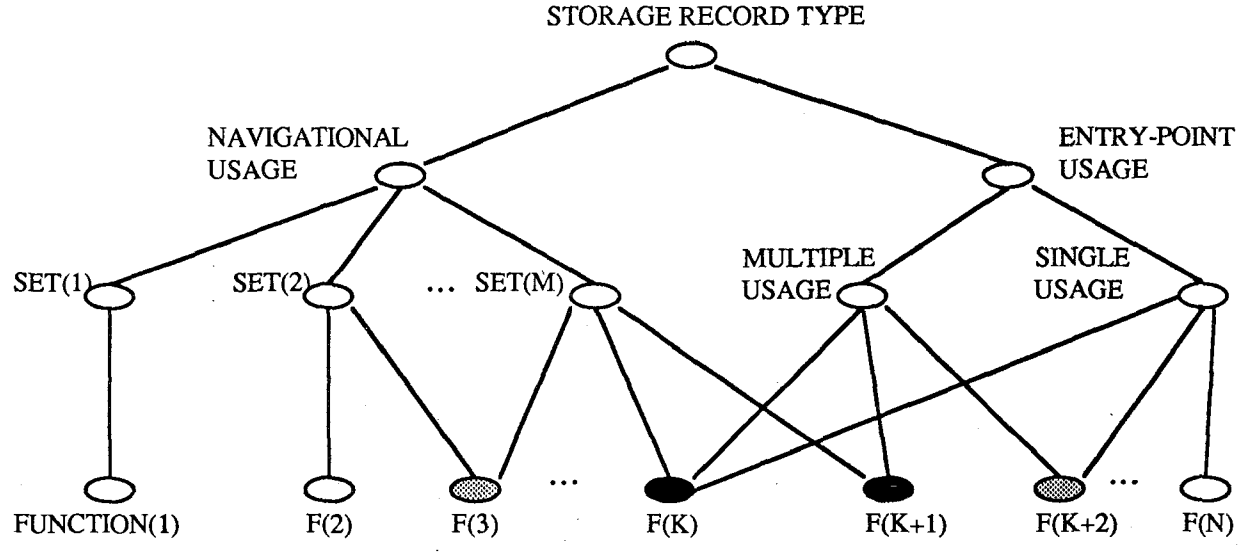


Figure 18. Graph of Storage Record Usage.

According to Figure 18, it is possible, that for a storage record type its occurrences are processed by functions in both navigational and entry-point ways. These functions are indicated in the graph by black circles - they do not contribute to our knowledge about the predominant record usage and are eliminated in relevant calculations. The dotted circles are used for those functions that process the record in either navigational or entry-point fashion, but in the former case they navigate through more than one set type and in the latter case they exhibit both multiple and single record usage. The multiple usage occurs when an entry-point search involves, in a next step, processing of some more record occurrences of the same type. The single usage takes place in case of a search for a single record occurrence (possibly with duplicates) satisfying particular conditions.

We formulate the following stepwise algorithm to determine the placement strategy for storage record types:

STEP1. Draw a graph of storage record usage for the record with the highest relative design importance RDI (cp. Figure 6). (Several graphs may be required if the logical record is mapped to several storage records.)

STEP2. Calculate the sum of relative design ranks of functions (RDR) in the navigational usage subtree (NRR) and in the entry-point usage subtree (ERR). Omit the ambiguous RDRs in the calculation (black circles in Figure 18).

STEP3. If NRR $\geq$ ERR then assign to the record type the placement CLUSTERED. Otherwise, the placement is determined as follows: (1) compute ERRs for the multiple usage (MERR) and the single usage (SERR) - include the RDRs of black and dotted circles in the sums, (2) if MERR > SERR, then assign to the record type the placement SEQUENTIAL, (3) otherwise, if MERR $\leq$ SERR, assign to it the placement CALC.

STEP4. If the record type was assigned CLUSTERED placement, a further decision is needed as to which set type will be used for clustering. To this aim, we compute the set-driven NRRs (S/DNRR) as the sums of NRRs of functions utilizing pertinent sets (clearly, the functions indicated by the black and dotted circles are also considered). The obvious implication is that the owner of the set with the largest S/DNRR is chosen for clustering.

STEP5. Repeat steps STEP1 through STEP4 for the remaining record types in the descending order of RDIs. Continue with STEP6.

STEP6. Draw a simplified physical database structure as exemplified in Figure 19 and consider the following special cases which can require modifications and/or special detailed approach to the placement strategies decided in STEP2 and STEP3.

(a) If a record type such as R3 is CLUSTERED VIA S2 and S2 is placed SEQUENTIAL (or CLUSTERED) and if there exists another set type for R3 (such as S1) present in the R3 usage graph, located CALC and having the next smaller S/DNRR in comparison with the set S2, then make R3 CLUSTERED VIA S1 and minimize the "side-effect" of such decision by ensuring that the owner pointers for S2 exist in R3.

(b) If a conflict arises because of CLUSTERED placement for two or more set types that are hierarchically related, such as S2 and S3 in Figure 19, then the ideal clustering is impossible and the pertinent placement strategies should be reconsidered (unless the sizes of the sets are extremelly small and for example all record occurrences fit in one page). One possibility is to change the placement of the record type such as R3 to CALC. Another possibility is to decide which set type (S2 or S3) takes priority and enforce the "best" sequence of records by means of a careful database loading and

taking advantage of DISPLACEMENT and WITHIN options of PLACEMENT subentry (Figure 7). In either case, the decision is arbitrary.

(c) If a conflict arises because of multiple member record types in a set type such as S4 and the members are CLUSTERED VIA this set type. The conflict is even harder to resolve if the owner record type is itself located as CLUSTERED (or SEQUENTIAL). In fact, this situation is a special case of case (b) and should be treated likewise.

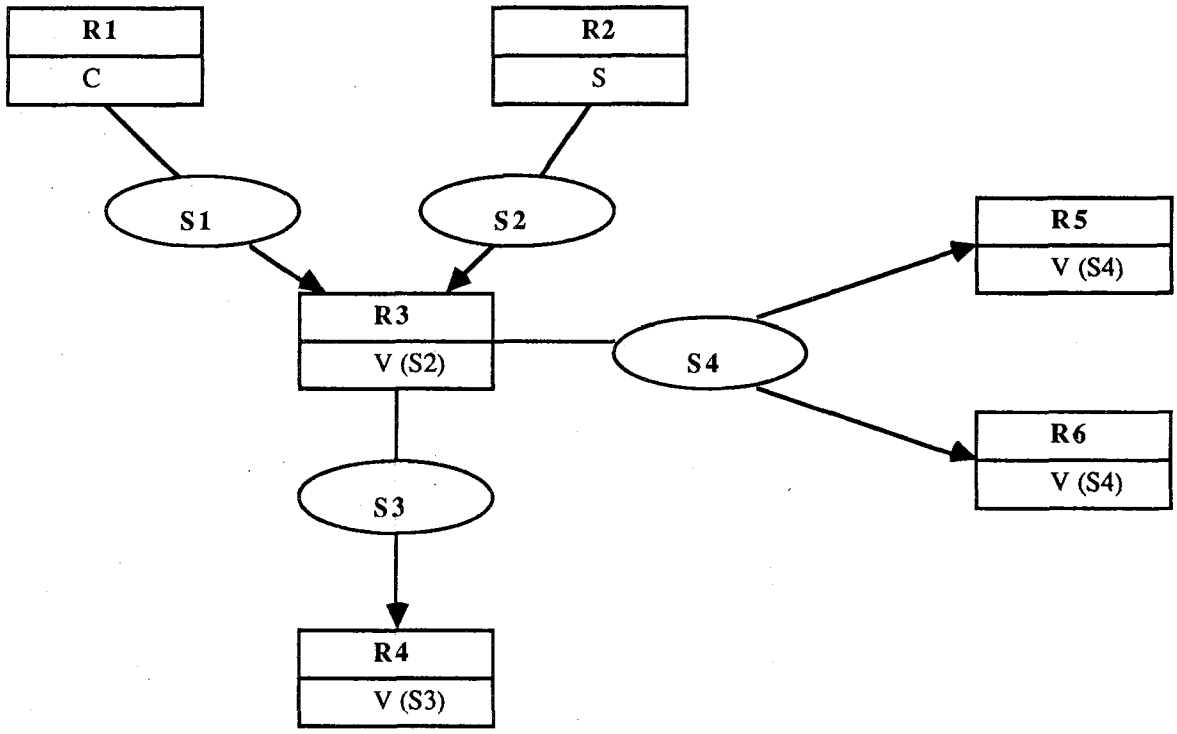


Figure 19. Simplified Physical Database Structure (example).

The special case of STEP6 exemplifies some of the design conflicts which are unmanageable algorithmically and have to be left to an intuitive decision that can only be verified to some extent in the performance prediction stage (M9). However, the decision -though intuitive - can be supervised by integrating it with other design parameters and objectives. Some of them - e.g. storing procedures and loading strategies - have been considered in this Section. The gross and fine placement are other issues which are inseparable from the record placement. Hence, we advocate throughout the iteration and integration aspects of our approach. This, incidentally, is more of an advantage than a restriction in the light of the IDDK featuring an expert system behaviour.

7. ACCESS PATH SELECTION

In this Section, we discuss the issues involved in deciding on the access paths for user functions that minimize the expected number of page accesses (and, therefore, the response times). No distinction is made between query and update functions as the latter also require to retrieve data before modifying it. We consider functions on an individual basis as if the system was entirely

dedicated to the processing of the function at hand. Later, during the performance prediction stage (M9), an attempt will be made to minimize the mean weighted response time for a workload made up of a set of functions. (The weighted response time is expressed as the ratio between the actual and the stand-alone dedicated response time to a function (Ferrari *et al.*, 1983).)

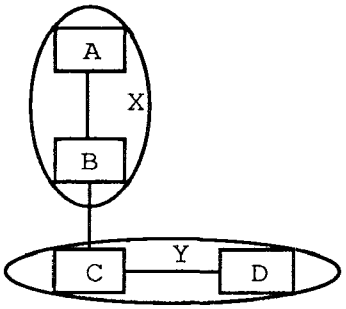
In a sense, the access path selection is a recapitulation of the placement decisions taken in the previous stages. Its aim is to advise the DBA how to design access algorithms of the application programs (and the interactive transactions) fulfilling the functions. The algorithms are determined based on available characteristics of logical and physical schemas, in particular - the implementation of sets. The algorithms specification is expressed as a pseudo-code showing the sequence of FIND commands to be issued against the database. Figure 20 presents the SET SELECTION clause of DDL'78 (Report, 1978). (In Schema Definition Language SDL'85, the SET SELECTION clause is not included, presumably removed to DSDL which, however, is not specified in Draft (1985).) Figure 21 presents the BNF syntax of the FIND statement of DML'85 (Draft, 1985) and Figure 22 represents an example in which different formats of the FIND command are used to search the same simple database. The example utilizes the syntax of VAX DBMS and is by no means exhaustive (other valid FIND formats are perfectly possible). The example is meant to be self-explanatory. It shows that the intricacy of the SET SELECTION clause and the sheer range of FIND commands are reasons for a variety of possible access paths to execute even a simple function. In particular, the example demonstrates that the network DBMS requires the execution of a FIND operation: (1) the currency indicators that come from the system area, or (2) the values of database identifiers that come from the user work area (UWA), or (3) the keeplist tables (system area), or (4) the sequential access to pertinent areas of a database, or (5) combinations of (1) through (4). (Incidentally, the major enhancement of DML'85 over the earlier specifications is the introduction of the WHERE clause, which is functionally equivalent to the predicate of the selection operator in the relational model (Deen, 1985). The WHERE clause, however, would not influence the logic behind the FIND statements as used in Figure 22.)

```
SET  SELECTION IS

[ THRU set-name-1 OWNER IDENTIFIED BY                                    ]
|    [ SYSTEM                                                           ]  |
|    | APPLICATION                                                       |  |
|    | KEY  key-name-1 [ data-identifier-1 EQUAL TO                      |  |
|    {                                                                   }  |
|    |    { data-identifier-2 } ]  ...                                   |  |
|    | parameter-name-1       }                                          |  |
|    [ SELECTION  DEFINED FOR record-name-1                             ]  |
|                                                                          |
{    [ THEN THRU set-name-2 WHERE OWNER IDENTIFIED BY                    ]  }
|    |     { data-identifier-3  [ EQUAL  TO                              |  ...
|    |                                                                   |
|    [          { data-identifier-4 } ] } ...                           ]
|                { parameter-name-2 }
|
|  BY  PROCEDURE procedure-name-1
|
|  BY  STRUCTURAL  CONSTRAINT
[                                                                        ]
```

Figure 20. Format of SET SELECTION Clause (DDL).

```
<find statement> ::=
    FIND
        <find specification>
        [<find intent>]
        [<find cursor disposition>]

<find specification> ::=
        <database key identifier>
    |   <search specification>

<search specification> ::=
    <search orientation>
    <domain specification>
    [WHERE <condition>]

<search orientation> ::=
    FIRST | LAST | NEXT | PRIOR
    | {ABSOLUTE | RELATIVE} <signed integer>

<domain specification> ::=
        <record type domain>
    |   <set domain>
    |   <subschema domain>

<record type domain> ::=
    <record view name>

<set domain> ::=
    [<record view name>] IN <set view name>

<subschema domain> ::=
    SUBSCHEMA RECORD

<find intent> ::=
    FOR {RETRIEVE | UPDATE}

<find cursor disposition> ::=
        RETAIN ALL
    |   <find specification disposition>

<find specific disposition> ::=
    [<position number>]
    [<find specific retention>]

<position number> ::=
    AS MEMBER <set view name>...

<find specific retention> ::=
        RETAIN RECORD
    |   RETAIN SET <set view name>...
    |   RETAIN RECORD SET <set view name>...
```

Figure 21. Syntax of FIND Statement (ANSI'85).



A1	B5	B2	A7	B9	B1
B4	B8	B6	B3	B10	A5
C2	C1	C3	C8	D2	C9
C7	C10	C4	D1	C11	C5

Currency Indicators:

R U	X	A-B	B-C	A	B	Y	D-C	C	D	KEEPLIST1	
B2	B2	B2	B3	A7	B3	C8	C5	C1	D2	1 2 3 4	B10 C11 D2 D1

Command	Record Found	Search Based On Current of	UWA value
FIND FIRST A	A1		
FIND LAST D	D1		
FIND NEXT B	B10	B	
FIND PRIOR C	C2	C	
FIND ANY B	B8		B-KEY=B8
FIND DUPLICATE B	Not-Found	B	
FIND RELATIVE -3 B	B4	B	
FIND +3 B	B9		
FIND FIRST WITHIN X	A1		
FIND NEXT WITHIN X	A7	X	
FIND FIRST WITHIN Y	C2		
FIND NEXT WITHIN Y	D2	Y	
FIND FIRST WITHIN A-B	B1	A-B	
FIND NEXT WITHIN A-B	B3	A-B	
FIND FIRST WITHIN B-C	C2	B-C	
FIND NEXT WITHIN B-C	C2	B-C	
FIND FIRST B WITHIN X	B5		
FIND NEXT B WITHIN X	B9	X	
FIND FIRST B WITHIN A-B	B1	A-B	
FIND NEXT B WITHIN A-B	B3	A-B	
FIND FIRST B USING B-KEY	B9		B-KEY=B9
FIND NEXT B WITHIN A-B USING B-KEY	No-Key	A-B	B-KEY=B2
FIND OWNER WITHIN A-B	A7	A-B	
FIND CURRENT	B2	R U	
FIND CURRENT WITHIN A	A7	A	
FIND CURRENT WITHIN A-B	B2	A-B	
FIND CURRENT WITHIN X	B2	X	
FIND FIRST WITHIN KEEPLIST1	B10	} Keeplist	
FIND LAST WITHIN KEEPLIST1	D1		
FIND OFFSET 2 WITHIN KEEPLIST1	C11		

Figure 22. Example of FIND Usage.

The access path selection is used here as a milestone in the verification of the physical design in the performance prediction stage. However, a number of other usages are foreshadowed. In particular, a non-procedural query interface to the network database may be the straightforward outcome of an efficient access path selector (optimizer). The pseudocode in which an access path is expressed can be used as a specification method in the realization phase (application software design). Also, the maintenance and evolution phase will take advantage of the tuning properties of the access path optimizer.

We begin by defining an access path evaluation model. To this aim, we first distinguish the categories of **atomic access units** *AU* and state the formulas to estimate the expected number of page accesses for each atomic unit. Then, we show by means of an example the pseudocode specification and the graphical notation for function (access path) representation to be used in the selection process. The interdependence between the cost of a given strategy for evaluating a function and the various parameters of the physical storage structure is indicated. We summarize by giving a stepwise algorithm for access path selection. While we exploit a few important contributions on access path selection in the network environment (Gerritsen, 1975; Dayal and Goodman, 1982; Jain, 1984; Whang, 1985), we go beyond the previous works not only in integrating the access path selection in an overall design methodology but also in comprehensiveness and specificity of treatment of the subject matter.

Atomic access units are used to build up the access path graphs for the functions. Having shown ways in which the atomic units may be connected, cost equations for the different access paths may be obtained. Figure 23 represents the structural chart of the network database access model. The leaf nodes of the chart correspond to atomic access units. Formulas (13) - (28) are the analytical expressions to estimate the expected number of page accesses #P (cp. search length notion SL_j in Section 3) required for each atomic access unit. The formulas are preceded by the definition of design parameters.

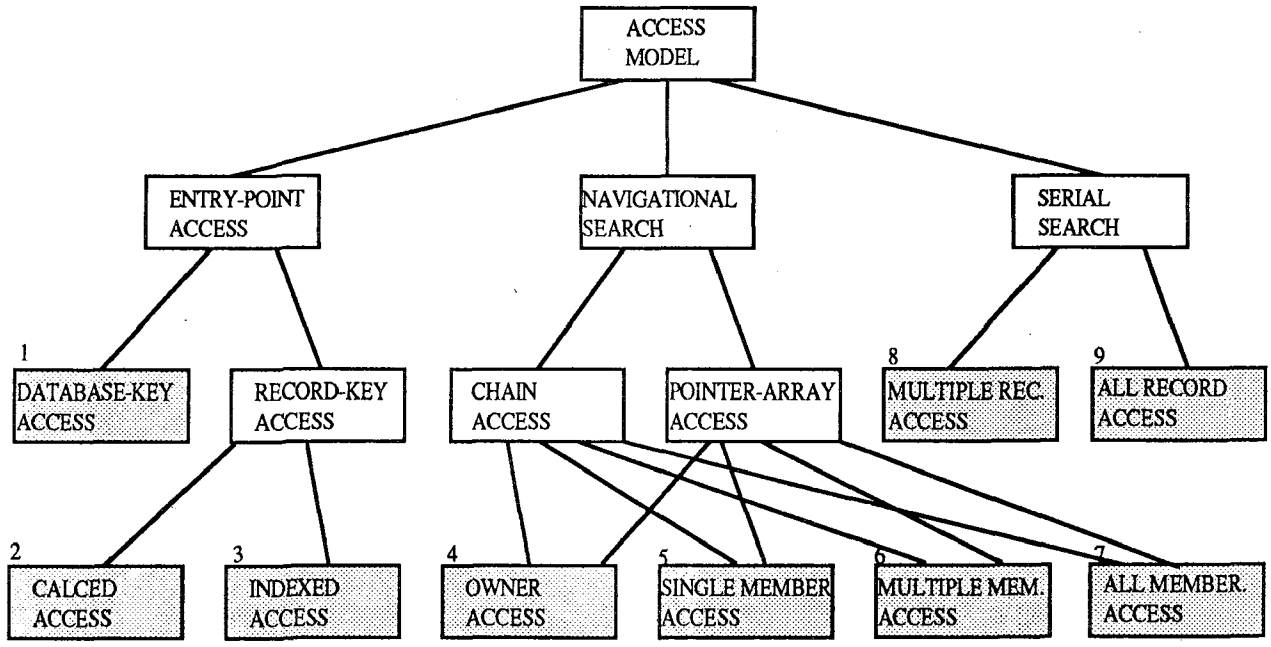


Figure 23. Structural Chart of Network Database Access Model.

1. Cardinality information (necessary to derive other parameters):

card(R)	- cardinality of record types in the database
$\|R_i\|$	- cardinality of record occurrences of type *i*
card(S)	- cardinality of set types in the database
$\|S_j\|$	- cardinality of set occurrences of type *j*
card(A)	- cardinality of areas in the database

2. Statistical information:

$\|MRO_{ij}\|$	- average cardinality of member record occurrences of type *i* in *j*th set type
$\|NRO_{ik}\|$	- expected cardinality of occurrences of *i*th record type in *k*th area
p_b	- probability of finding the record occurrence in the buffer, i.e. the ratio between average number of records in the buffer to the expected cardinality of records in the area
p_o	- probability of overflow of calc-chain to another page; this is a function $f(PD_k)$
p_r	- probability that two or more record occurrences of type R_i are placed in one page; this is a function $f(EPS_k,RL_i,AS_k,g(R,A))$, where g(R,A) expresses the mapping of record types to areas (Section 4)

3. Size information:

EPS_k	- effective page size in *k*th area
RL_i	- average length of record occurrences of type *i*
AS_k	- size of *k*th area
PD_k	- initial packing density of *k*th area in percentages (assumed to be constant throughout the design procedure)

4. Zero-one test information:

OT_{ij} - owner pointer test = $\begin{cases} 1 & \text{, if } i\text{th member record type in } j\text{th set type has owner pointers} \\ 0 & \text{, otherwise} \end{cases}$

AT_{ik} - area test = $\begin{cases} 1 & \text{, if } i\text{th record type is assigned to } k\text{th area} \\ 0 & \text{, otherwise} \end{cases}$

5. Selectivity information:

$SEL(P_q,R_i)$	- selectivity of predicate P of query q when applied to *i*th record type, i.e. the probability that a given record occurrence r_i satisfies the selection predicate

(1) Database-Key Access:

$$\#P_1 = 1 - p_b \tag{13}$$

(2) Calced Access:

$$\#P_2 = 1 + p_o * p_b \tag{14}$$

(3) Indexed Access:

$$\#P_3 = \log_b|R_i| \tag{15}*$$

b - the branch factor, i.e. the number of key values in a node (or, in the case of B-trees, one greater than the number of key values)

* the master index is assumed to be in the main memory

(4) Owner Access:

(i) through chain:

$$\#P_{41} = \begin{cases} 1 - p_b & \text{,if } OT_{ij} = 1 \\ (\#P_{71} + 1) * 0.5 \text{ or} & \\ (\#P_{72} + 1) * 0.5 & \text{, if } OT_{ij} = 0 \end{cases} \tag{16}$$

(ii) through pointer-array:

$$\#P_{42} = 1 - p_b \tag{17}*$$

* OT_{ij} is required to be equal 1

(5) Single Member Access:

(i) through chain:

$$\#P_{51} = |MRO_{ij}| * (1 - p_r) * 0.5 \tag{18}$$

(ii) through pointer-array:

$$\#P_{52} = \begin{cases} |MRO_{ij}| * 0.5 & \text{for ordinary pointer arrays} \\ \log_b|MRO_{ij}| + 1 & \text{for keyed and sorted pointer arrays} \end{cases} \tag{19}$$

(6) Multiple Member Access:

(i) through chain or ordinary pointer array:

$$\#P_{61} = |MRO_{ij}| * (0.5 / SEL(P_q,R_i)) \tag{20}$$

(ii) through sorted chain or keyed sorted pointer array (P_q = f(sort key)):

$$\#P_{62} = ((|MRO_{ij}| * RL_i) / (EPS_k * PD_k)) / SEL(P_q,R_i) \tag{21}$$

(7) All Member Access:

(i) through chain (if member occurrences of type *i* are clustered via the set *j*):

$$\#P_{71} = ((|MRO_{ij}| * RL_i) / (EPS_k * PD_k)) \quad (22)^*$$

* it is assumed that the number of record occurrences r_i in a page is not restricted by means of the DENSITY clause (Figure 7)

(ii) through chain (if the placement of member occurences r_i is clustered via a set other than *j* (or the placement is calc or sequential):

$$\#P_{72} = |MRO_{ij}| * (1 - p_r) \quad (23)$$

(iii) through pointer-array:

$$\#P_{73} = |MRO_{ij}| \quad (24)$$

(8) Multiple Record Access:

(i) if the records are sequentially placed:

$$\#P_{81} = \Sigma_k AT_{ik} * ((|NRO_{ik}| * RL_i) / (EPS_k * PD_k)) / SEL(P_q,R_i) \quad (25)$$

(ii) if the records are not sequentially placed:

$$\#P_{82} = \Sigma_k AT_{ik} * (|NRO_{ik}| * 0.5) / SEL(P_q,R_i) \quad (26)$$

(9) All Record Access:

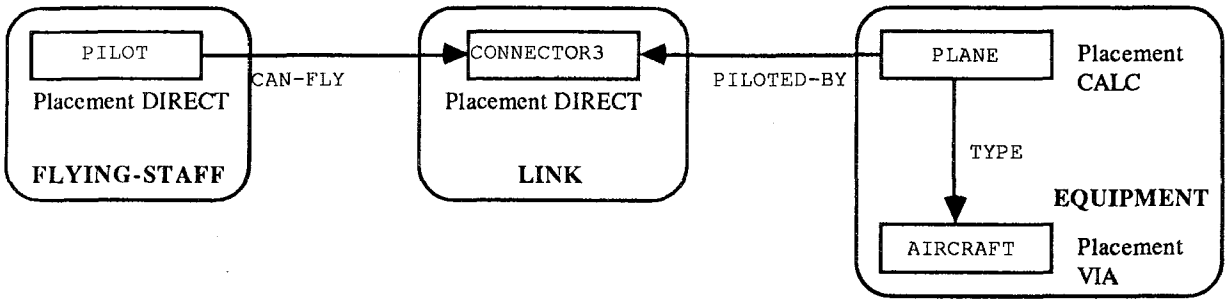
(i) if the records are sequentially placed:

$$\#P_{91} = \Sigma_k AT_{ik} * ((|NRO_{ik}| * RL_i) / (EPS_k * PD_k)) \quad (27)$$

(ii) if the records are not sequentially placed:

$$\#P_{92} = \Sigma_k AT_{ik} * (AS_k / EPS_k) \quad (28)$$

We are now in a position to derive the costs of different access paths which can be used in response to a user function. At run time, the minimal cost access path will be selected to proceed with the execution of the function. At design time, the selection of the best path for execution must be made possible by including all involved record types, set types and areas in the subschema through which the function communicates with the database (cp. Section 4 and realization phase). Technically speaking, the distinction must be made between on-line and batch functions. For on-line functions, the logic of path selection is partly incorporated in the subschema definition by adding to it the so called path section (Figure 24 shows an example for QLP of DMS-1100). The path section determines all possible paths in a desirable order (from best to worst). These paths are used by the query language processor to generate access to database as a response to a function. For batch functions, the logic of path selection is built into an application program. That is, the sequence of execution of DML commands determines the database accesses.




```
IDENTIFICATION DIVISION
SUBSCHEMA NAME QLPSUB IN FILE LESDBS OF SCHEMA AIRLINE
HOST LANGUAGE QLP
DATA DIVISION
DATA NAME SECTION
    DATA NAMES EQUIPMENT,AKEY1,FLYING-STAFF,AKEY2,LINK
AREA SECTION
    AREAS EQUIPMENT,FLYING-STAFF,LINK
RECORD SECTION
    RECORDS PLANE,AIRCRAFT,PILOT,CONNECTOR3
SET SECTION
    SETS TYPE,CAN-FLY,PILOTED-BY
QLP SECTION
PATH NET1
    ROOT PLANE THRU PILOTED-BY TO CONNECTOR3
               THRU CAN-FLY TO PILOT
PATH NET2
    ROOT PILOT THRU CAN-FLY TO CONNECTOR-3
               THRU PILOTED-BY TO PLANE
PATH SIMPLE
    ROOT PLANE THRU TYPE TO AIRCRAFT
PATH PATH1
    ROOT PLANE THRU PILOTED-BY TO CONNECTOR3
PATH PATH2
    ROOT PILOT
    DIRECT CURRENCY ASSUMED
    AREA-NAME IS FLYING-STAFF
    PAGE-NUM IS 1
    RECORD-NUM IS 1
    THRU CAN-FLY TO CONNECTOR3
PATH PATH3
    ROOT PILOT THRU CAN-FLY TO CONNECTOR3
PATH PATH4
    ROOT PLANE FETCH NEXT CURRENCY ALLOWED
PATH PATH5
    ROOT CONNECTOR3 FETCH NEXT CURRENCY ALLOWED
PATH PATH6
    ROOT PILOT FETCH NEXT CURRENCY ALLOWED
```

Figure 24. Subschema Diagram and Definition for Query Language (QLP-1100).

Since the IDDK has evolved around the assumption of a pre-canned function environment (Maciaszek, 1986), the determination of the best access paths for individual functions paves the way for the global "optimization" consideration in the performance prediction stage (M9).

Consider the network schema, which was derived by one of the IDDK tools in Maciaszek *et al.* (1986a), and is repeated here in Figure 25. Assume that the user is interested in the

> "names of employees of Department of Fairyland who taught courses taken by Donald Duck during the last six semesters".

It is clear that the schema is not particularly supportive for execution of this query.

There are many possible strategies to solve the above query depending on the way and sequence in which the available atomic access units are utilized in building up the access path. We compare two access strategies with respect to the following assumptions on actual data values. For simplicity, we assume that all involved record types are placed in one area and their blocking factor is one. Moreover, the paths through LINK05 are not taken advantage of (this virtually means that the same set types are used in both strategies). The placements of records are evident from the pseudocode that follows (in the syntax of VAX DBMS - cp. Figure 22). We also assume - contrary to the fact that the access path selection is but one stage of uncompleted design process - that the gross, fine, and record placement characteristics are not subject to changes. We now specify the values of the parameters for path selection, the pseudocodes for the two access strategies, their access graphs (Figures 26 and 27), and their costs.

$|R_1|$ = 10 departments
$|R_2|$ = 400 employees
$|R_3|$ = 20 semesters
$|R_4|$ = 5000 students
$|R_5|$ = 500 courses
$|R_6|$ = 4000 LINK03 occurrences
$|R_7|$ = 4800 LINK04 occurrences (contain SEMNUM)
$|R_8|$ = 6000 LINK05 occurrences
$|R_9|$ = 40000 LINK 06 occurrences

S_1 = DPRT-EMPL	$MRO_{21} = 40$
S_2 = EMPLOY-L04	$MRO_{72} = 12$
S_3 = SEMEST-L04 (no owner pointers)	$MRO_{73} = 240$
S_4 = SEMEST-L03 (owner pointers)	$MRO_{64} = 200$
S_5 = COURSE-L03 (owner pointers)	$MRO_{65} = 8$
S_6 = COURSE-L06 (pointer array)	$MRO_{96} = 80$
S_7 = STUDEN-L06 (pointer array)	$MRO_{97} = 8$

Strategy 1:

```
        MOVE 'DONALD DUCK' TO SNAME IN STUDENT
1.      FIND ANY STUDENT USING SNAME IN STUDENT
        loop1 (until no more LINK06)
2.          FIND FIRST (NEXT) LINK06 WITHIN STUDEN-L06
```

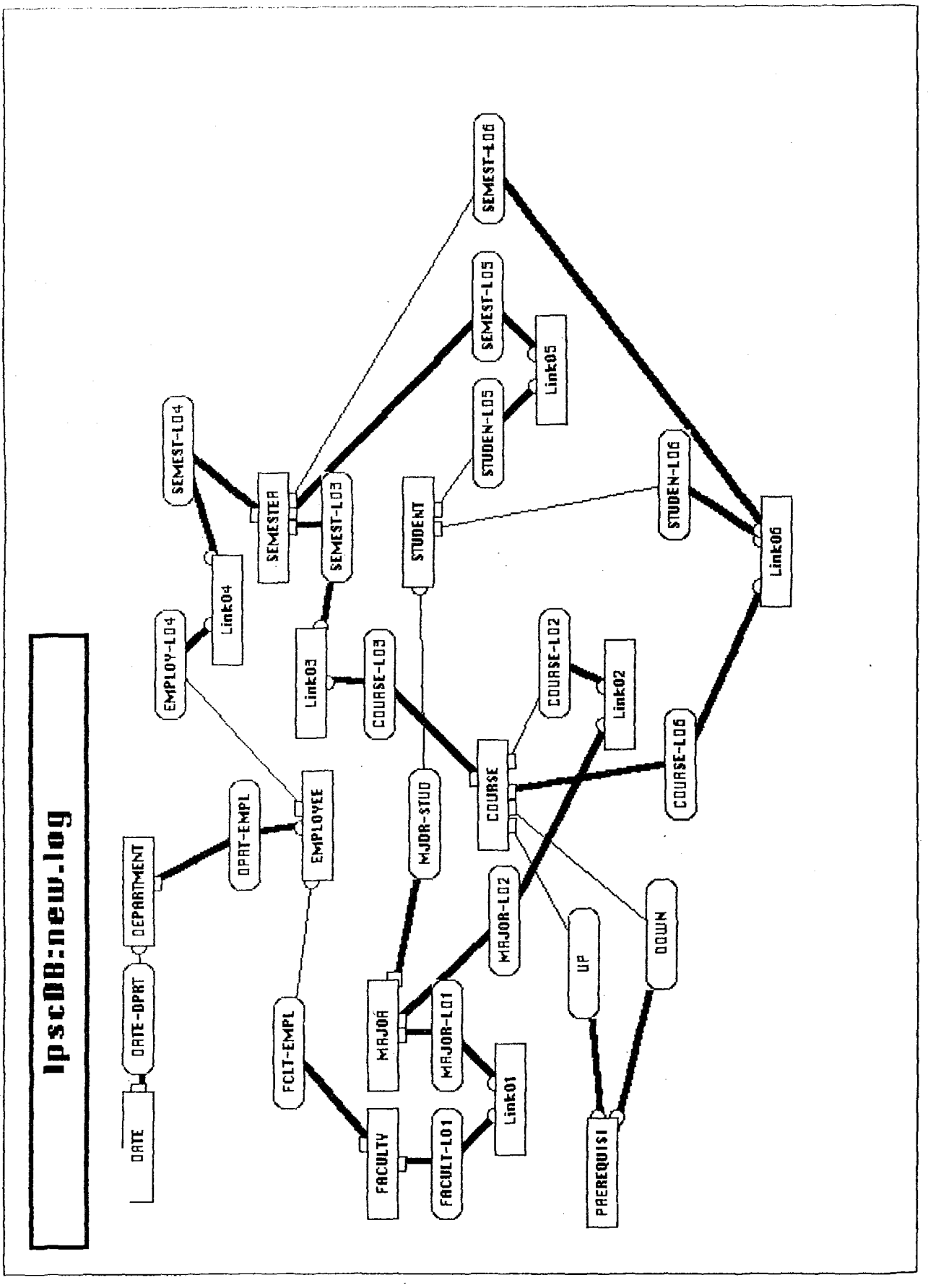


Figure 25. Diagram of the University Database Schema.

```
3.          FIND OWNER WITHIN COURSE-L06
            KEEP CURRENT COURSE USING KEEPLIST1
        endloop1
        MOVE 'DEPARTMENT OF FAIRYLAND' TO DNAME IN DEPARTMENT
4.      FIND ANY DEPARTMENT USING DNAME IN DEPARTMENT
        loop2 (until no more EMPLOYEEs)
5.          FETCH FIRST (NEXT) EMPLOYEE WITHIN DPRT-EMPL
            loop3 (six times)
6.                  FIND FIRST WITHIN KEEPLIST1
                    FREE FIRST WITHIN KEEPLIST1
                    loop4 (until no more LINK03)
7.                      FIND FIRST (NEXT) LINK03 WITHIN COURSE-L03
8.                      FIND OWNER WITHIN SEMEST-L03
                        KEEP CURRENT SEMESTER USING KEEPLIST2
                    endloop4
            endloop3
            loop5 (until end of KEEPLIST2)
9.                  FETCH FIRST WITHIN KEEPLIST2
                    FREE FIRST WITHIN KEEPLIST2
                    loop6 (until no more LINK04)
10.                     FETCH FIRST (NEXT) LINK04 WITHIN EMPLOY-L04
                        IF SEMNUM IN LINK04 = SEMNUM IN SEMESTER
                           DISPLAY 'EMPLOYEE IS ' ENAME OF EMPLOYEE
                    endloop6
            endloop5
        endloop2
```

Strategy 2:

```
        MOVE 'DEPARTMENT OF FAIRYLAND' TO DNAME IN DEPARTMENT
1.      FIND ANY DEPARTMENT USING DNAME IN DEPARTMENT
        loop1 (until end of set DPRT-EMPL)
2.          FETCH FIRST (NEXT) EMPLOYEE WITHIN DPRT-EMPL
            loop2 (until end of set EMPLOY-L04)
3.                  FIND FIRST (NEXT) LINK04 WITHIN EMPLOY-L04
4.                  FETCH OWNER WITHIN SEMEST-L04
                    if one of the six semesters then loop3 (until end of set SEMEST-L03)
5.                      FIND FIRST (NEXT) LINK03 WITHIN SEMEST-L03
6.                      FIND OWNER WITHIN COURSE-L03
                        loop4 (until end of set COURSE-L06)
7.                          FIND FIRST (NEXT) LINK06 WITHIN COURSE-L06
8.                          FETCH OWNER WITHIN STUDEN-L06
                            IF SNAME = 'DONALD DUCK'
                                DISPLAY 'EMPLOYEE IS ' ENAME OF EMPLOYEE
                        endloop4
                    endloop3
            endloop2
        endloop1
```

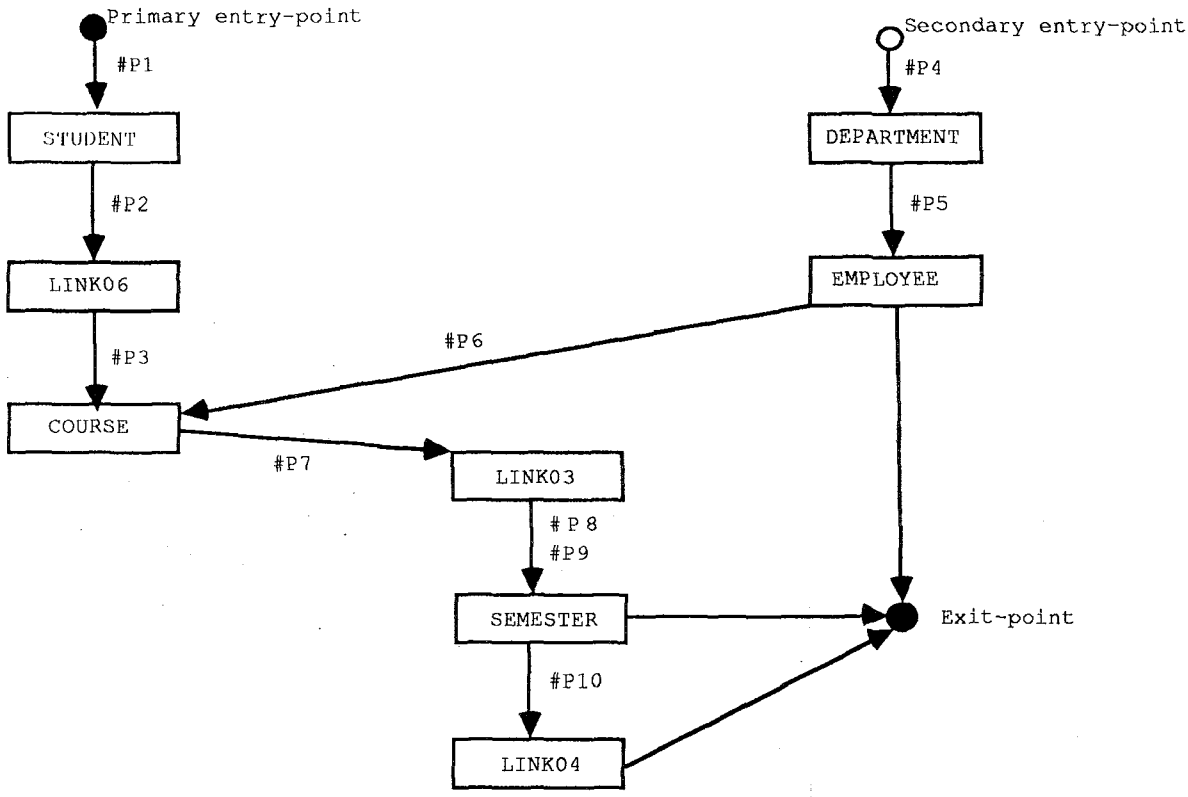


Figure 26. Access Graph for Strategy 1.

Cost of Strategy 1 (ref. costs of atomic access units AU in Formulas 13 - 28):

1.	AU2	#P1 = 1
	loop1	
2.	AU7	#P2 = 8
3.	AU4	#P3 = 1 * 8 = 8
	endloop1	#endloop1 = 1 + 8 + (8 * 1) = 17
4.	AU2	#P4 = 1
	loop2	
5.	AU7	#P5 = 40
	loop3	
6.	AU1	#P6 = 1 * 6 = 6
	loop4	
7.	AU7	#P7 = 8
8.	AU4	#P8 = 1 * 8 = 8
	endloop4	#endloop4 = 16
	endloop3	#endloop3 = 6 * 16 = 96
	loop5	
9.	AU1	#P9 = 8
	loop6	
10.	AU7	#P10 = 12
	endloop6	#endloop6 = 12
	endloop5	#endloop5 = 8 * 12 = 96
	endloop2	#endloop2 = 40 * (96 + 96) = 7,680
	end	#Pend = 7680 + 17 = 7,697

Answer: Approximately 7,697 accesses are needed to respond to the query according to Strategy 1.

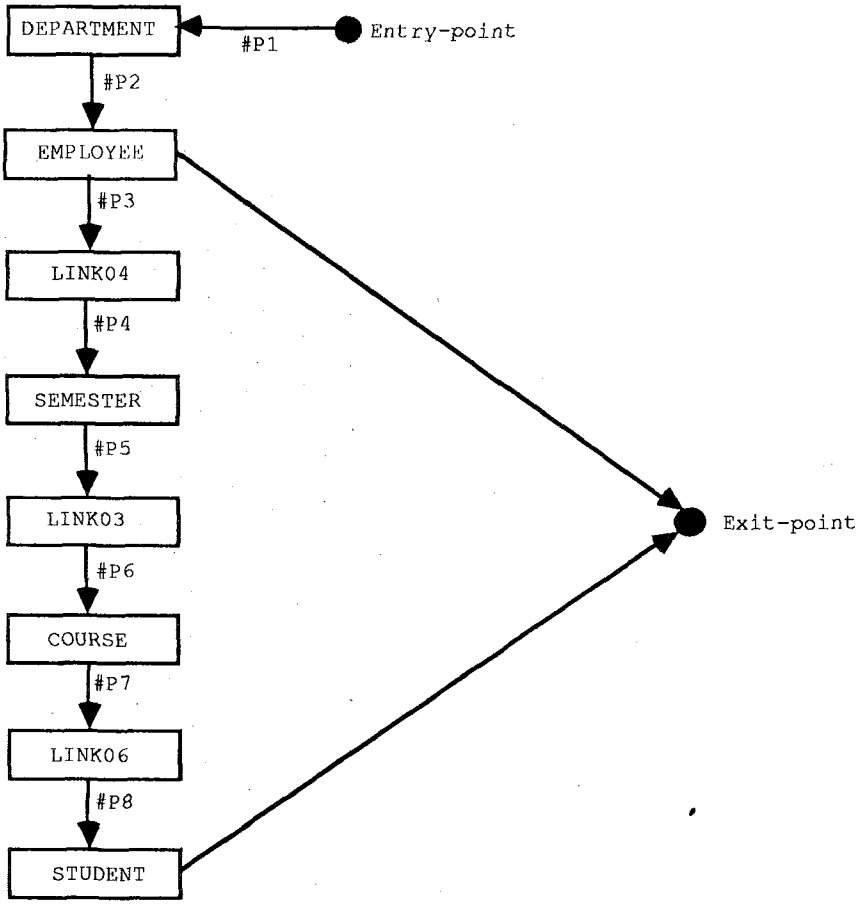


Figure 27. Access Graph for Strategy 2.

Cost of Strategy 2 (ref. costs of atomic access units AU in Formulas 13 - 28):

1.	AU2	#P1 = 1
	loop1	
2.	AU7	#P2 = 40
	loop2	
3.	AU7	#P3 = 12
4.	AU4	#P4 = 120 * 12 = 1440
	loop3	
5.	AU7	#P5 = 200
6.	AU4	#P6 = 1 * 200 = 200
	loop4	
7.	AU7	#P7 = 80
8.	AU4	#P8 = 1 * 80 = 80
	endloop4	#endloop4 = 160
	endloop3	#endloop3 = 400 * 160 = 64,000
	endloop2	#endloop2 = 1452 * 64000 : (20/6) semesters = 27,878,400

endloop1 #endloop1 = 40 * 27878400 = 1,115,136,000

end #Pend = 1,115,136,000

Answer: Approximately 1,115,136,000 accesses are needed to respond to the query according to Strategy 2.

The astronomical cost of Strategy 2 results from the fact that it is a nested approach which - although straightforward - is quite orthogonal to the schema. The deeply nested approaches tend to exhibit exponential complexities. Hence, for example, if we allowed for owner pointers in SEMEST-L04, then the cost of the strategy would reduce to 10,752,000 page accesses (a significant improvement). Moreover, this still unrealistic number should be reduced by relaxing the assumption of blocking factor equal to one. Let #P1 (#P2) be the number of one-record pages read in the outer (inner) loop. #P1 + #P1 * #P2 secondary storage accesses are required to evaluate the query. If, however, a main memory buffer can hold one or more many-record pages such that all required occurrences of one of the record types can be kept in the buffer, then only #P1 + #P2 accesses are necessary. Note also that further improvements can result from some basic alterations to the schema definition.

We conclude this Section with a list of steps of the overall procedure for access path selection:

STEP1. Build a cost model of atomic access units.

STEP2. Based on the schema definition and function specifications, define the reasonable logical access path strategies for evaluating the function. This has been demonstrated in this Section by means of pseudocodes and access graphs, but in the IDDK environment we will provide for a special Function Definition Language FDL. The specifications in FDL will constitute inputs to the performance prediction system (stage M9).

(As an aside, we note that our approach differs from that of the relational query optimization which requires to generation of (rather than specification of) all access paths. This is because the relational query languages are nonprocedural. However, our methodology can be readily extended to incorporate nonprocedural interfaces for network databases. This can be done by generating some (rather than all) access strategies which will exploit the superior semantic power of the network model.)

STEP3. Augment the logical access strategies with the details of physical representation of the database (gross, fine, and record placement). In this Section, the physical characteristics have been considered in the pseudocode and access graphs. And again, the physical information will constitute an input to the performance prediction tool of IDDK (Section 11). In fact, this step makes the difference between the selection and optimization models for access evaluation. In an automated environment of IDDK, the access evaluation can be tuned up by varying values for physical database parameters.

STEP4. Choose the cheapiest access path strategy by applying a cost model of atomic access units to various possible paths.
Execute this step in two sub-steps:

STEP4A. Find all "redundant" paths such that these paths give rise to a network flow problem with the identical source and sink nodes (first and last accessed record, respectively). The objective is to minimize the flow from source to sink. That is, the shortest path should be determined and all other "redundant" paths should be rejected at this sub-step. In other words the problem is:
"Given a connected values graph $\mathbb{G}$ where the positive arc-values #P represent cost of

the arcs, find the cost-minimal path from source to sink."
There exist well known algorithms to solve that problem, such as the algorithms of Dijkstra or Dantzig (Eiselt and Frajer, 1977). Those algorithms can also be used for handling multiple sources and multiple sinks by means of the concepts of super-source and super-sink (Daellenbach *et al.*, 1983). Our example could be made relevant to this sub-step by including LINK05 in determining alternate paths.

STEP4B. Find the cost-minimal access path from among all "nonredundant" paths, as shown in our example.

Finally, we must point out that the power and pragmatism of our approach is mostly due to its integration with the overall methodology. Since the design process starts with and is based on the function specifications, the conceivable access path strategies and the underlying database structures are de facto pre-selected. This means that the unreasonable access strategies are virtually precluded from being considered in the selection process. At this juncture it is perhaps relevant to draw another distinction between query optimization in the network and relational settings. Let us quote P.M.D. Gray: "In fact Codasyl databases are usually designed to have access paths (sets) that suit common queries and one could even consider these sets as "precomputed joins" and say that the Codasyl model is the best known way of storing very large relations on existing hardware so as to perform joins efficiently!" (Gray, 1984).

8. INTRA-RECORD STRUCTURE AMELIORATION

At the logical level design (formalization), the intra-record structures have been defined and viewed as sequences of data-items (v. stage F3 in Figure 1). Apart from the elementary data items, the group and repeating items have been allowed (cp. Maciaszek *et al.*, 1986a). The repeating items could lead to variable length records. Such view of intra-record structure has been quite adequate, since the functional content was a determinant of the design process. At the physical level, however, the efficiency issues of retrieval and update time, and the storage space become important. We have extensively dealt with the access considerations in the previous Section. In this Section we refer to the techniques for refinement of intra-record structures within the boundaries of the DSDL'78 data subentry specification. Hence, we leave aside two related issues: logical-to-storage record mapping and transparent compression. The former have been briefly addressed in stages M3 (Section 5) and M4 (Section 6) (see also March and Scudder, 1984). The latter is beyond the DBA's influence as the transparent compression refers to the routines built-in to a DBMS, which improve system performance but otherwise the database user is not affected (it has been shown that such compression techniques reduce secondary storage requirements in typical commercial databases by 30 - 90%; cp. Aronson (1977), Cormack (1985), Severance (1983), Welch (1984) and others). Figure 28 presents the general format of data subentry - the last subentry of storage record entry (Report, 1978).

The data subentry complements the transparent compression by letting the DBA define the storage representation parameters. The parameters include: different storage formats; alignment (synchronization); derived data items; null compaction to supress zeros, blanks, or both; implicit or explicit sign.

Let us emphasize the importance of storage representation by means of a simple practical example set up in the DMS-1100 environment (Figure 29). (In fact, synchronization (SYNC) of Version 3 steps beyond the syntax of the DDL of DMS-1100, but it is allowed in the host language environment.) It can be seen that the size of the record differs significantly for the three versions (from 1057 to 1664 bytes). It may be of some surprise that in some situations the binary synchronized formats (Version 3) do not offer space savings over the display (character per byte) representations (Version 1) - in fact, in the example, the most space consuming format is the

synchronized one. Bearing in mind that thousands of occurrences of this record type might be kept in the database, the space saving can be considerable. Moreover, there is no trade-off to speed. If anything, the shortest record version will speed up processing since the computational data items are stored in the binary format.

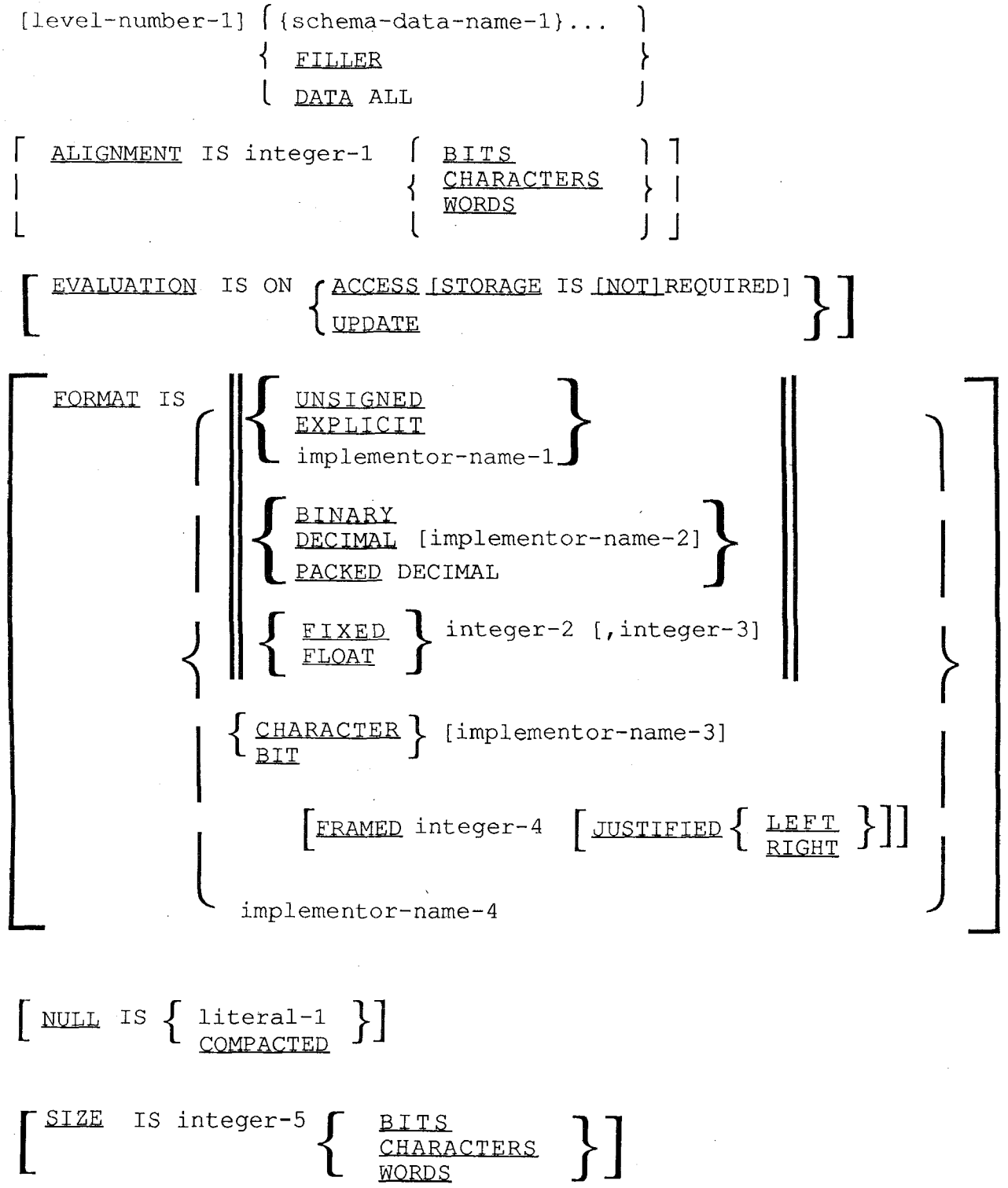


Figure 28. Format of Data Subentry.

Version 1:

```
RECORD NAME IS CUSTOMER
    LOCATION MODE IS CALC HASH-CUST
            IN AREA5 USING CUSTOMER-NAME DUPLICATES NOT ALLOWED
    WITHIN AREA5
    RECORD MODE IS ASCII
    02 CREDIT-ACCOUNT-NUMBER        PIC X(4)
    02 CUSTOMER-NAME                PIC X(20)
    02 CUSTOMER-ADDRESS             PIC X(20)
    02 CREDIT-RATING                PIC 9
    02 AMOUNT-BORROWED              PIC 9(7)
    02 LOAN-ACCOUNT-NUMBER          PIC 9(4)
    02 CURRENT-BALANCE              PIC 9(7)
    02 NUMBER-OF-TRANSACTIONS       PIC 9(4)
    02 TRANSACTIONS
            OCCURS 1 TO 100 TIMES
            DEPENDING ON NUMBER-OF-TRANSACTIONS
            03 TRANSACTIONS-DATE    PIC 9(6)
            03 TRANSACTION-AMOUNT   PIC 9(7)
            03 TRANSACTION-CODE     PIC X
```

$RS_1 = 4 + 20 + 20 + 1 + 7 + 4 + 7 + 4 + (6 + 7 + 1) * 100 = 1467$ bytes

Version 2:

```
RECORD NAME IS CUSTOMER
    LOCATION MODE IS CALC HASH-CUST
            IN AREA5 USING CUSTOMER-NAME DUPLICATES NOT ALLOWED
    WITHIN AREA5
    RECORD MODE IS ASCII
    02 CREDIT-ACCOUNT-NUMBER        PIC X(4)
    02 CUSTOMER-NAME                PIC X(20)
    02 CUSTOMER-ADDRESS             PIC X(20)
    02 CREDIT-RATING                PIC 9
    02 AMOUNT-BORROWED              PIC 9(7) USAGE COMP
    02 LOAN-ACCOUNT-NUMBER          PIC 9(4)
    02 CURRENT-BALANCE              PIC 9(7) USAGE COMP
    02 NUMBER-OF-TRANSACTIONS       PIC 9(4) USAGE COMP
    02 TRANSACTIONS
            OCCURS 1 TO 100 TIMES
            DEPENDING ON NUMBER-OF-TRANSACTIONS
            03 TRANSACTIONS-DATE    PIC 9(6)
            03 TRANSACTION-AMOUNT   PIC 9(7) USAGE COMP
            03 TRANSACTION-CODE     PIC X
```

$RS_2 = 4 + 20 + 20 + 1 + 3 + 4 + 3 + 2 + (6 + 3 + 1) * 100 = 1057$ bytes

Figure 29. Impact of Data Item Formats on Record Sizes (to be continued).

Version 3:

```
RECORD NAME IS CUSTOMER
    LOCATION MODE IS CALC HASH-CUST
           IN AREA5 USING CUSTOMER-NAME DUPLICATES NOT ALLOWED
    WITHIN AREA5
    RECORD MODE IS ASCII
    02 CREDIT-ACCOUNT-NUMBER          PIC X(4)
    02 CUSTOMER-NAME                  PIC X(20)
    02 CUSTOMER-ADDRESS               PIC X(20)
    02 CREDIT-RATING                  PIC 9
    02 AMOUNT-BORROWED                PIC 9(7) USAGE COMP SYNC
    02 LOAN-ACCOUNT-NUMBER            PIC 9(4)
    02 CURRENT-BALANCE                PIC 9(7) USAGE COMP SYNC
    02 NUMBER-OF-TRANSACTIONS         PIC 9(4) USAGE COMP SYNC
    02 TRANSACTIONS
          OCCURS 1 TO 100 TIMES
          DEPENDING ON NUMBER-OF-TRANSACTIONS
          03 TRANSACTIONS-DATE       PIC 9(6)
          03 TRANSACTION-AMOUNT      PIC 9(7) USAGE COMP SYNC
          03 TRANSACTION-CODE        PIC X
```

RS_3 = 4 + 20 + 20 + 1 + 3 slack bytes + 4 + 4 + 4 + 4 + (6 + 2 slack bytes + 4 + 1 + 3 slack bytes) * 100 = 1664 bytes

Figure 29. Impact of Data Item Formats on Record Sizes (continued).

An interesting feature of a database environment, not present in conventional file systems, is the possibility to evaluate a data item value on access or update (Figure 28). This is applicable only to the data items which were declared as being SOURCE or RESULT in the logical schema (the value of the SOURCE data item must be the same as the value of another data item, whereas the value of the RESULT data item is computed according to a predefined algorithm). The EVALUATION clause not only ensures that the value of the object data item is current on each access or update but also can ensure that the data item value will not take space (STORAGE IS NOT REQUIRED) and will be created each time it is required. (We must note, however, that SOURCE and RESULT clauses were dropped in ANSI'85 specifications due to implementation difficulties (Draft, 1985).)

The storage representation yields similar benefits and disadvantages to the transparent compression. Thus, it can reduce storage cost, buffer requirements, and transfer time. On the other hand, it can require additional processing time due to compression and decompression operations, handling variable-length records and bit-level manipulations. Further research is needed to establish an interdependence between intra-record structure parameters and relevant characteristics of database applications (the size of the database, the amount and type of data redundancy, the nature and frequency of user functions, etc.).

9. INDEX SPECIFICATION

Indexes in a network database can support record types and set types (see Figures 16 and 23). For record types, they must be keyed on either a logical record key (Figure 30) or on a storage key (Figure 4). (It is important to remember that a logical record type can have more than one key defined by separate KEY clauses and that the DBMS allocates a unique storage key to each

occurrence of the storage record to be indexed.) For set types, the indexes are called here pointer arrays and they can be keyed (pointer plus a key) or ordinary (pointer minus a key). The presence of indexes has already been considered by us in Stage M4 (Section 5) and M5 (Section 6). In this Section we provide a more detailed discussion on this issue.

```
KEY  key-name-1 IS { [ ASCENDING  ]  data-identifier-1 } ...
                   [ DESCENDING ]

     DUPLICATES  ARE { FIRST
                       LAST
                       NOT ALLOWED
                       SYSTEM-DEFAULT }

[ FREQUENCY  OF || DIRECT     || RETRIEVAL  IS  HIGH ]
                || SEQUENTIAL ||
```

Figure 30. Format of Record KEY Clause (DDL).

Figure 31 presents a taxonomy of indexing techniques as a basis for index specification. The interpretation of the graph is that the concatenated and feasible properties of an index are those that are linked together and thus, create an uninterrupted path. The path is broken (or finished) by putting a black circle at the end of the link. For example, it is feasible to specify a pointer array (set index) that is keyed, based on a single key such that it consists of several simple items and is not unique. However, it is impossible to specify a record index which is ordinary and specify a database index built on several keys.

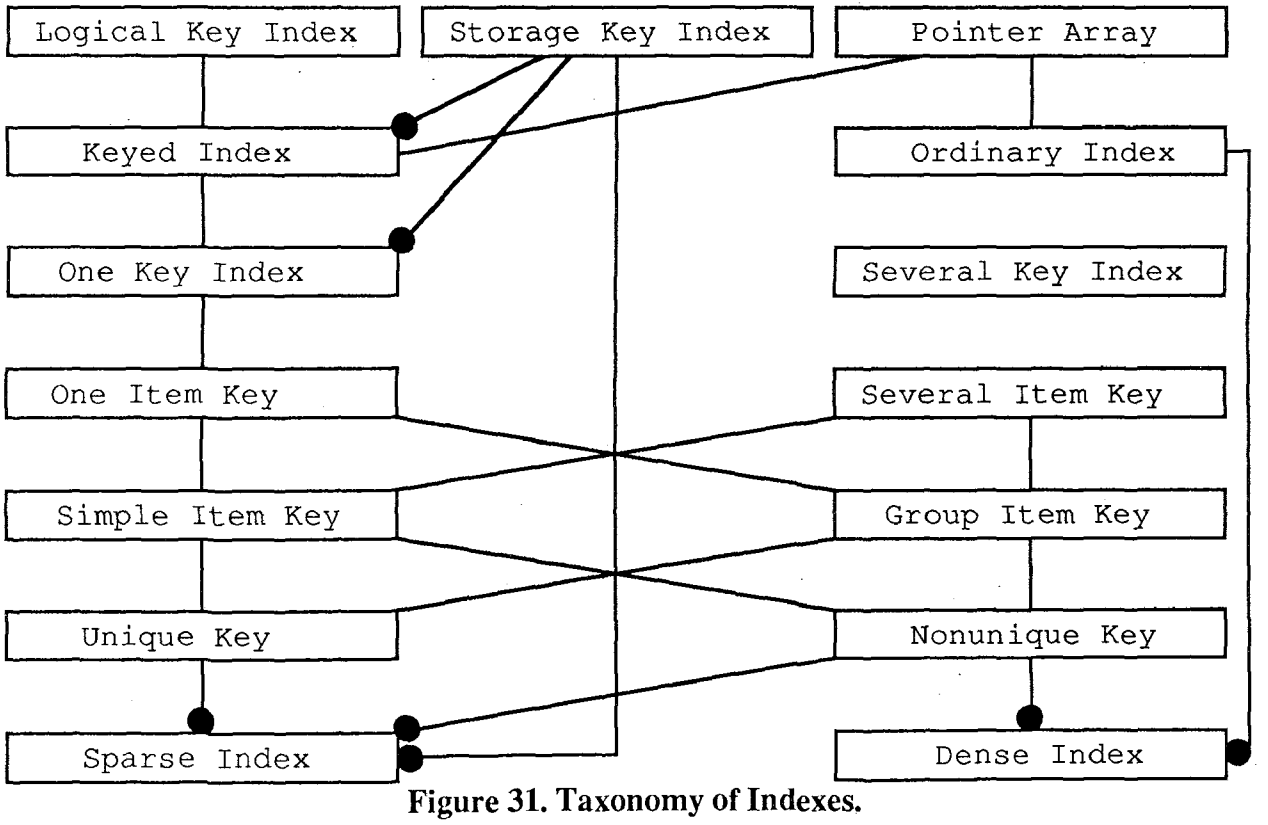


Figure 31. Taxonomy of Indexes.

The taxonomy of Figure 31 is meant to be useful to the DBA in customizing an index definition. However, some of the index categories may not be amenable to the DBA's decision due to a specific DBMS implementation and some as a matter of principle. Let us elaborate on these problems a liitle.

The DBA is given a choice of using a record index or a pointer array. A keyed index may be specified for logical record type, storage record type, and set type. A logical record key to be indexed must be chosen from among the key(s) defined in the KEY clause for the record type (Figure 30). (This clause was removed in the ANSI-85 proposal and, instead, an optional (and less specific) uniqueness clause was introduced (Draft, 1985).) A storage record key for the index is established by means of the obligatory specification STORAGE KEY REQUIRED in the storage record subentry (Figure 4), but apart from that the DBA is not involved.

A keyed index can only be built on one key. A several key index (often referred to as combined index) is not supported by the index entry (Figure 16). In a several key index, each index entry would contain a pointer to a record (or a list of records) which simultaneously satisfies all key values. Theoretically, several key indexes could be applicable to logical keys, since in the DDL there is a provision to define several keys for a record type (by means of writing several KEY clauses in the same record entry; cp. Figure 30). This provision, however, is not extended on the index entry. Supposedly, the maintenance problems with several key indexes have been regarded as outweighing the potential benefits of speeding up some of the more complex queries. After all, these sorts of indexes have not seen much practical use even in the dedicated (as opposed to governed by universal DBMS) database systems (cp. Claybrook, 1983).

A keyed index can further be classified as consisting of one data item or several data items (this division is not applicable to storage key indexes, as shown in Figure 31). To implement a several item key, the DBA is required to use several data-identifiers:

(a) in the record KEY clause in the case of a logical key index (Figure 30) or
(b) in the SET ... KEY option of index entry in the case of a pointer array (Figure 16).

Each of the data-identifiers can in turn designate either a simple or a group item. This gives rise to the next classification in Figure 31. In fact, an index based on a several item key, such that the items are group items, can be considered as a nearly perfect simulation of a several key index.

The logical key indexes and pointer arrays can be constructed on unique keys (DUPLICATES NOT ALLOWED) or on nonunique keys (DUPLICATES FIRST, LAST or SYSTEM-DEFAULT). The record KEY clause is used to this aim in the case of the logical key index. For pointer arrays, the same can be accomplished by omitting in the index entry the KEY option and stating in the DDL that the set is SORTED (with the appropriate DUPLICATES clause).

A natural implementation technique for unique keys is a sparse index, for nonunique keys - a dense index. However, since the DBMS can ensure alternative ways to locate duplicates, the sparse indexes are typically used in both cases. Ordinary pointer arrays are by necessity implemented as dense indexes.

Apart from sparse and dense indexing, the taxonomy of Figure 31 does not address implementation techniques for indexes, as this problem is beyond the DBA's decision scope. Broadly speaking, from an implementation point of view, all known index structures can be classified on those that organize the specific set of data (e.g. B-trees) and those that organize the embedding space in which the data is located (e.g. grid files). While the former techniques are still not seriously challenged by the latter in the database market place, there has been (due to intensive research in recent years) a significant progress in search techniques that organize the embedding space (cp. Nievergelt *et al.*, 1984). Regretably, this subject, though exciting, remains beyond the scope of this report.

We now turn to the placement of index records. As with data records, the facilities available can be classified into those controlling gross placement (cp. Section 3) and those relevant to fine placement (cp.Section 4). With reference to the index entry (Figure 16), the gross placement is addressed by the WITHIN clause and the fine placement by the PLACEMENT clause.

As far as the gross placement is concerned, the main decision to be made is whether the page range for the index records should coincide with (or be separated from) the page range for the supported data records. A quick insight into the problem dictates the following heuristic rule that has been confirmed by experience (cp. IDMS, 1982).

HEURISTIC. For record key indexes, the use of a separate page range or area is preferable. For pointer arrays, it is usually more efficient for the index records to have the same page range as the member records if they are clustered, otherwise - the same page range as the owner of the set.

The reasons why a separate page range or area is usually better for record key indexes are threefold (IDMS, 1982):

(1) The data records can be clustered more closely together.
(2) The index records may be given pages to themselves, and can therefore grow to their maximum size without having to be relocated.
(3) Contention for the area or page range containing the index can be reduced, as the pages are not locked just because of operations (mainly updates) on data records that may not even be related to the index.

The pointer array records will normally benefit from sharing the page range with the CLUSTERED member records, since a particular data record can often be retrieved together with the pointer array records that lead to it. The risk of pointer array relocation can be kept under control by the wise use of a DISPLACEMENT clause for data records (see Figure 7). Putting the pointer array records in the page range of the owner of the set containing unclustered members should be self-explanatory in view of the above discussion.

It should be mentioned, however, that the gross placement of index records can turn out to be academic in face of the restrictions imposed by some older DBMS-s. For example, DMS-1100 requires the specification of separate areas for record key indexes, pointer arrays, and data records (DMS, 1984).

Fine placement facilities are strictly correlated with the applied gross placement; to the extent that record key indexes can only be controlled by the gross placement facilities and fine placement is left entirely to the system (the DBMS attempts to distribute index key records evenly over the page range by using a randomizing algorithm). The scope of DBA decisions in the case of fine placement for pointer arrays is also minimal.The PLACEMENT clause (Figure 16) allows either for hashing on a data item within the owner record type or for placing the pointer array records near a storage record representing the set owner.

10. DISC SPACE REQUIREMENTS

We have already dealt with space calculation on many occasions - during the gross placement (Section 4), fine placement (Section 5), and access path selection (Section 6). The primary reason for this Stage is to put on previous discussion in a systematic framework and extend it by providing a consistent survey of the calculation formulas.

Calculating the total amount of disc space required for database files is simple once the database structure is determined. A recommended practice today is to produce a stub schema and

make the system calculate the sizes. This can be done by running a schema report generator (one of the processors of any self-respecting DBMS.) For example, the schema report generator SRT of DMS-1100 calculates record sizes in 36-bit words (Figure 32) and area sizes in 1792-word tracks. More specifically, the record size is given in terms of control data length, user data length, and total length. For variable length records, the maximum record length is calculated and the designer should convert it to the average length instead. The storage assignment requirements for each area (not shown in Figure 32) are calculated as follows: the number of words allocated for the area (the allocated pages multiplied by the words per page) is divided by 1792 words per track to produce the number of tracks required for the area (DMS, 1984).

The schema report generator, however, is a generalized tool, not particularly suited for more detailed space calculations. Therefore, we formulate now a specific method which provides the basis for constructing yet another CAD tool in our IDDK. The method calculates the space needed using the following steps:
(1) space for an average occurrence of each data record type,
(2) space for occurrences of all data record types,
(3) space for an index,
(4) space for all indexes,
(5) number of pages required for database area,
(6) total disc space requirements.

As mentioned, the space for an average occurrence of the record type can be obtained from the schema report. The calculation formula is straightforward (cp. Formula (5)):

$$RL_i = RS_i + CD_i \tag{29}$$

By way of illustration, we show in Figure 33 the DMS-1100 approach to establish RL_i (DMS, 1984).

The space required for occurrences of all data record types DS is estimated by the formula which rounds up RL_i to a multiple of four and multiplies it by the expected number of occurrences of *i*th record type NRO_i. The products for all record types are then summed.

$$DS = \Sigma_{\forall i} (\lceil RL_i \rceil * NRO_i) \tag{30}$$

The calculation of the space required for an index IS_{ind} is system-dependent, as the index implementation in Report (1978) is left to the implementor (it is likely, however, that the implementation will be of B-tree type with a dynamic data split technique, as opposed to a static overflow area technique). We propose the following generalized recursive formula to estimate IS:

$$IS_{ind} = \Sigma_{\forall x} IS_{ind}(x) \tag{31}$$

where:

$$\begin{cases} IS_{ind}(0) = NRO_{ind} * IES_{ind} \\ \\ IS_{ind}(x+1) = \log_b(IS_{ind}(x)) \qquad \text{until } IS_{ind}(x+1) \le 1 \end{cases}$$

such that the branch factor b (the number of key values in the node) equals:

$$b = MIS_{ind} / IS_{ind}$$

where:

RECORD NAME	LOCATION MODE	AREAS IN WHICH THE RECORD MAY BE STORED
PERSONNEL	CALC	FLYING-STAFF GROUND-STAFF

SET PARTICIPATION (AS A MEMBER)	POINTERS	LOCATION CRITERIA AS A MEMBER
PERSONAL-FILE (AUTOMATIC)	NEXT PRIOR	EMP-NO (ASCENDING)
IS-A (MANUAL)	NEXT	SET ORDERED NEXT MEMBER OF TYPE FOLLOWING CURRENT OF SET IS SELECTED

SET PARTICIPATION (AS AN OWNER)	POINTERS
ASSIGNED-TO	NEXT PRIOR

RECORD LENGTH IN WORDS		
CONTROL	DATA	TOTAL
7	15	22

POINTERS	
2	OWNER POINTER(S)
2	AUTOMATIC POINTER(S)
1	MANUAL RESERVED POINTER(S)
1	CALC POINTER(S)

Figure 32. Record Report Produced by SRT (DMS-1100).

$IS(0)$	the space occupied by index records on the bottom index level,
NRO_{ind}	the number of data record occurrences that can be accessed through the index,
IES_{ind}	the index entry size (this equals the key length (which is zero for ordinary pointer arrays) plus four bytes for the pointer (plus two bytes for the record identifier for pointer arrays used for more than one member record type)),
MIS_{ind}	the maximum index record size.

```
Record Header (Fixed)                                          4 bytes

Owner Pointers
     Owner of _____ sets with next links * 4 bytes             __ bytes
     Owner of _____ sets with prior links * 4 bytes            _______
     Owner of _____ sets with order last * 4 bytes             _______
     Owner of _____ pointer array sets * 4 bytes               _______

Automatic Member Pointers
     Member of _____ sets with next links * 4 bytes            _______
     Member of _____ sets with prior links * 4 bytes           _______
     Member of _____ sets with owner links * 4 bytes           _______

Manual Flag Control (present only if defined as a
                                    manual member)             4 bytes

Manual Member Pointers
     Number of pointers reserved * 4 bytes                     _______

Chain Pointers (calc or index-sequential)
     Add 4 bytes for single links, 8 for double                _______

User Data (RS_i)                                               _______

System Pointers
     Add 4 bytes if record is modified and written
     on overflow page                                          _______
```

Figure 33. Record Space Calculation (DMS-1100).

The space required for all indexes IS is the sum of sizes of all record indexes and pointer arrays declared in the database:

$$IS = \Sigma_{\forall ind} (IS_{ind}) \tag{32}$$

The number of pages required for *k*th database area $\#P_k$ is equal to:

$$\#P_k = (DS + IS)_k / EPS_k * PD_k \tag{33}$$

where:

EPS_k	the effective page size as derived in the Stage M3 (Section 5),
PD_k	the initial packing density of area *k* in a percentage.

The total disc space requirement should be given in tracks rather than pages. It must be also remembered that space is required for the directory, test databases, recovery journals, space management records, and for the files that contain codes of the logical schema, physical schema, subschemas, and application programs. Special consideration should be given to a predicted growth factor of a database (possibly including such details as the space required for logically deleted member records in the sets without prior pointers). For these reasons, our formula for the calculation of TS is only approximate:

$$TS = (\Sigma_{\forall k} (\#P_k / \lfloor TC / EPS_k \rfloor)) + ES \qquad (34)$$

where:

TC	the track capacity,
EPS	the effective page size in *k*th area,
ES	the extra space required for the directory, programs, etc.

11. PERFORMANCE PREDICTION

The performance prediction stage is the culmination of the physical design phase. This is where the partial solutions obtained in the previous stages are integrated and the global efficiency issues are considered. In a sense, the earlier stages have served merely to establish feasible solutions and to promote candidate implementations. The paths of investigation that have had little chance of success have been eliminated, leaving, however, a still large enough design region for significant differences in operational performance due to variations in the physical database parameters. In this Section, we propose an overall framework for the Physical Evaluation and Selection (PES) tool aimed at assessing and predicting the effect of physical design decisions on resource consumption, transaction throughputs, response times, etc. The specification of the PES has been influenced by:

* the Layered Model of Database System Performance due to Sevcik (1981),
* the EOS and EROS systems of DATAID project (Staniszkis and Rullo, 1982; Staniszkis *et al.*, 1982; Orlando *et al.*, 1985),
* the physical database design environment described by Teorey and Fry (1982), and
* the measurement and tuning techniques for computer systems presented in Ferrari *et al.*, (1983).

We use Data Flow Diagrams (DFD) to represent the PES graphically in Figures 34 and 35. The specific notation is consistent with that described in Jeffrey and Lawrence (1984). The directed line represents a data flow, the circle portrays a data transformation, the straight line expresses a data store, and the rectangle refers to an originator or receiver of data. Each data flow is named, except for single flows to or from a data store. Each of the bubbles (circles) represents a transformation of the incoming data flows by a processor of the PES. By convention, only the net flow to or from a data store is shown on the diagrams (hence, for example, in an update process only the flow to the file is shown, despite I-O processes being involved in reality).

Figure 34 presents the major categories of inputs and outputs for the performance prediction. The framework of the PES is restricted to the physical design and does not provide for modifications to the conceptual or logical designs (Maciaszek, 1986; Maciaszek *et al.*, 1986a). According to Sevcik's (1981) classification, the PES is an evaluation and selection system, but it is not an optimizer. The PES is capable of evaluating the performance of a specific design with respect to certain measures. It can also assist in selecting among a small number of design alternatives. However, it is not aimed at producing the set of parameter values that optimize performance with respect to a given performance measure (or group of measures).

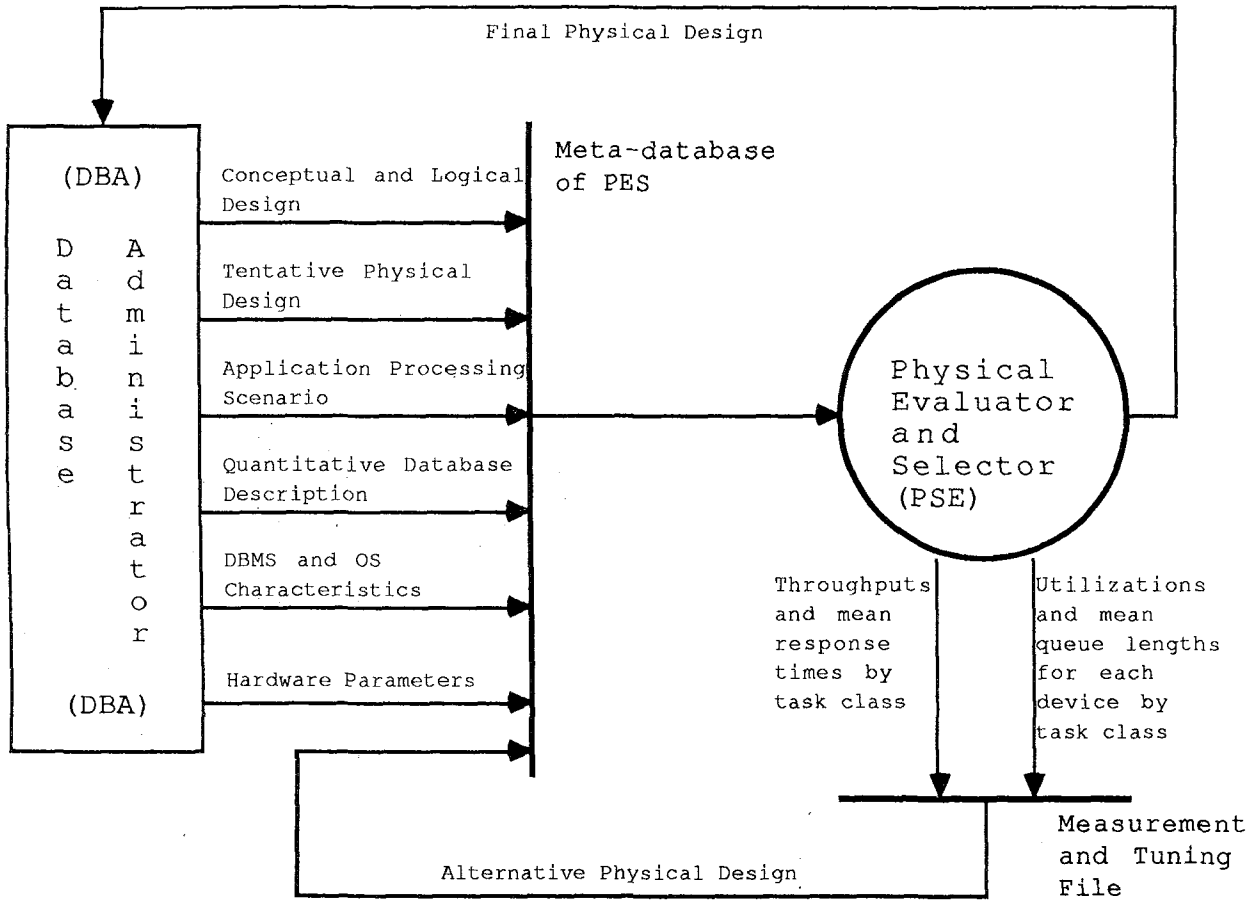


Figure 34. Physical Evaluator and Selector (Data Flow Diagram: level 0).

The PES is a design-time system. Nevertheless, with minor changes to the interfaces, it can be used for performance prediction of existing databases under changing workload characterizations and with tuning of physical parameters. Indeed, the accuracy of the PES could be best verified by comparing its prediction results with the performance of a real database system. In any case, it is reasonable to assume that the inaccuracies due to the modelling assumptions in the PES will generally be insignificant in comparison with uncertainties in workload characterization and workload forecasting (Sevcik, 1981). This motivated us to give special consideration to the way a workload and its characterization are decided upon. We do not rely entirely on the probabilistic approach which represents the workload by random variables distributions of which are assumed to be statistically independent. Instead, we draw from the knowledge on conceptual and logical designs and generate a likely application processing scenario. This approach responds to the research results that have shown that the uniformity and independence assumptions in database design lead to cost estimations that are often pessimistic (Christodoulakis, 1984).

The database environment for the PES is a multiuser and shared system (however, a non-database workload is not considered). This means that disc space is shared by many users, so that the position of the read/write head on the device can be considered as a random variable. As pointed out by Teorey and Fry (1982), this can have the effect (in the worst case) of making each block access, a random access, regardless of where a user's previous block access occurred. We believe that relaxing the requirement of multiuser and shared system for database performance prediction would be unreasonable.

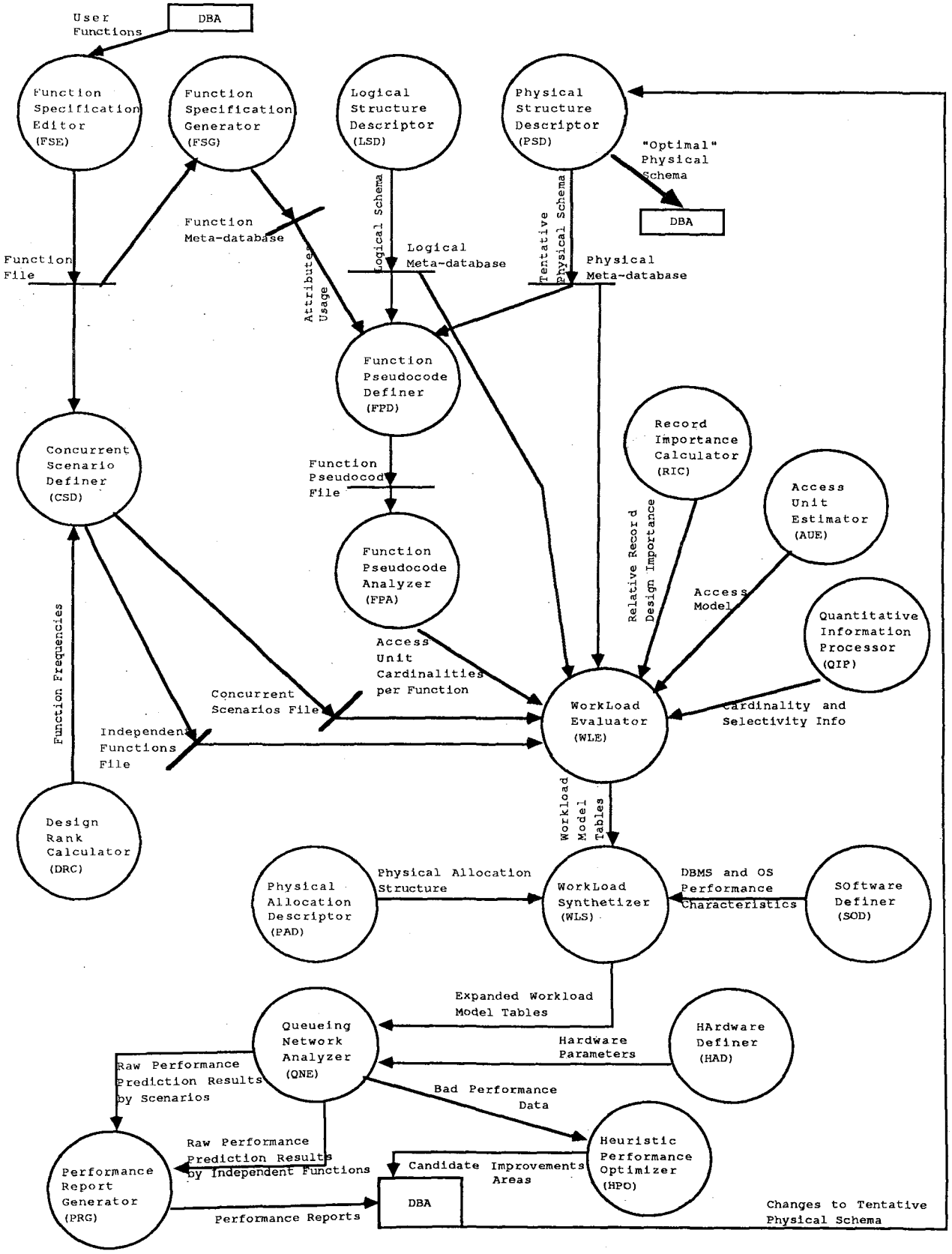


Figure 35. Physical Evaluator and Selector (Data Flow Diagram: level 1).

The PES results in a group of measurement indexes that provide diversified views of the database performance. It is left mostly up to the DBA to make a trade-off analysis when different individual indexes result in conflicting design decisions. It must be remembered, though, that some indexes are beyond the DBA's control and they can only constitute a background information from the PES (e.g. CPU and I/O queue waiting time, communication delay). Other indexes can be tuned by the PES and the DBA (e.g. I/O service time, secondary storage space) or they can be controlled in a limited scope (e.g. lockout delays, CPU time, main storage space) (Teorey and Fry, 1982).

In general, the architecture of the PES, as shown in Figure 35, is designed to support the following performance indexes:

* response time,
* throughput (or productivity),
* secondary storage cost and utilization,
* turnaround time,
* CPU memory cost and CPU utilization,
* overlap.

Response time is defined as the elapsed time between the interactive user function initiation and the instant the corresponding reply begins to appear at the terminal. It consists of a CPU service time, CPU queue waiting time, I/O service time, I/O queue waiting time, lockout delay, and communications delay (Teorey and Fry, 1982). It seems that the I/O service time is most susceptible to the DBA's control and as such should be reported separately. The global response time for a user function should be further divided in order to show the execution time of DML commands used in the pseudocode for the function. The content of the reports is expected to resemble those of the DATAID project (Orlando *et al.*, 1985). The statistical description of response times for all functions should also be provided (standard deviation or percentiles of the distribution of response times (Ferrari *et al.*, 1983)). (An observed response time is the pth percentile if p percent of all response times are below the observed time.)

Throughput is measured as the amount of work performed by a system in a given unit of time (Ferrari *at al.*, 1983). Its value is to be expressed in the number of functions processed per unit of time and in the amount of I/O transfers per unit of time. The value of throughput must be analyzed in strong correlation with a given workload and the system's capacity (the maximum theoretical value that the throughput can reach). Throughput is an important measure in calculating the system's usage cost. It provides the global indication of the system's power.

Secondary storage cost is expressed in the disc space required by the database. The **utilization** is a percentage of the disc active time for the workload. The utilization reports should be enriched by the calculation of the related factors: mean service time, visits (I/O transfers), average waiting time, average queue length.

Turnaround time is defined as the time interval between the instant a batch user function (batch program) is submitted to the system and the instant its execution ends (Ferrari *et al.*, 1983). It provides an indication of processing efficiency, especially when the weighted turnaround time and the mean weighted turnaround time are obtained. The weighted turnaround time is the ratio between the turnaround time and the program's processing time in a stand-alone (dedicated) environment.

The **CPU memory cost** is expressed in the main memory space needed for the workload (the DBA has some control over buffer allocation, but little else). The **CPU utilization** is defined as the percentage of operating time during which the CPU is active (Ferrari *et al.*, 1983). The analysis of the CPU utilization should also show the time spent for system overhead (for Operating System and DBMS functions). The CPU utilization can also be used as one of the throughput indices. The frequency with which the programs constituting the workload generate references to

information items not present in main memory (page faults) should also be obtained.

The **overlap** is described as the percentage of system's operating time during which two or more resources are simultaneously busy (Ferrari *et al.*, 1983). This index should reflect all the possible combinations of channels and the CPU.

The PES is an integral part of the IDDK. In fact, some of the PES processors (Figure 35) are being implemented outside of the PES and are expected to easily interface with the other PES tools. The PSE, FSG, DRC are already operational in prototype forms on Macintosh Plus workstations (Maciaszek *et al*, 1986b). The CSD, FPD and FPA provide the workload characterization. The LSD is a part of the logical level IDDK (Maciaszek *et al.*, 1986a). The RIC and AUE are the IDDK tools relevant to the earlier stages of materialization (M1 and M5, respectively). The QIP is a consequence of a broad range of activities in all design phases addressed by the IDDK (conceptualization, formalization, and materialization). The SOD and PAD includes settings for DBMS tunning parameters and for operating system environment. The HAD specifies the hardware, in particular disc characteristics. The actual measurement and performance prediction is done by the WLE, WLS, and QNE (with operational properties similar to those in DATAID (Staniszkis and Rullo, 1982; Staniszkis *et al.*, 1982). The tuning is supported by the HPO which will be built around the heuristic algorithms described more or less explicitly in the previous sections of this report. Finally, the PRG will produce tabular and graphical analysis of the evaluation and selection results. The overall process is iterative and leads to a semi-optimal physical schema.

12. PHYSICAL SCHEMA DEFINITION

The physical schema definition is the tangible outcome of the physical design process. In practice, it is given in the DSDL. The object physical schema is consulted during the database accesses. During the design process, however, the DSDL definition should be supported by the graphical representation of the physical schema. It is foreshadowed that the IDDK will first produce the graphical display of the schema (using the tool named the Physical Diagram Constructor PDC), and then the DSDL definition will be automatically generated (under the assumption that the IDDK is customized for a given DSDL).

The graphical schema representation is consistent with the logical diagrammatic technique already operational in a prototype version within the IDDK (Maciaszek *et al.*, 1986a) and resembles the diagrammatic notation developed by us in Maciaszek (1981). Figure 36 uses the "fork" set type to show the graphical notation for the physical schema definition. We assume that the diagram is self-explanatory. Because of the growing complexity of the diagrams for larger structures, the PDC will only automatically generate small diagrams (up to ten record types, say), but it will provide separate graphs for each set type. For large diagrams, the PDC should be supported by the Physical Diagram Editor (PDE) which will allow the DBA to move shapes around and design the output layout.

13. CONCLUSION

In this report we have described the methodology to derive a network physical database schema based on the conceptual and logical designs. The methodology is accurate enough to be used as a blueprint for the development of a computer-assisted design tool. Such a tool will be integrated in the Intelligent Database Design Kit (IDDK), which is a set of computer-assisted tools of expert system flavour for analysing, constructing and documenting a database design. The IDDK is partly operational in prototype form in the Apple Macintosh Plus environment (conceptual and logical design scope).

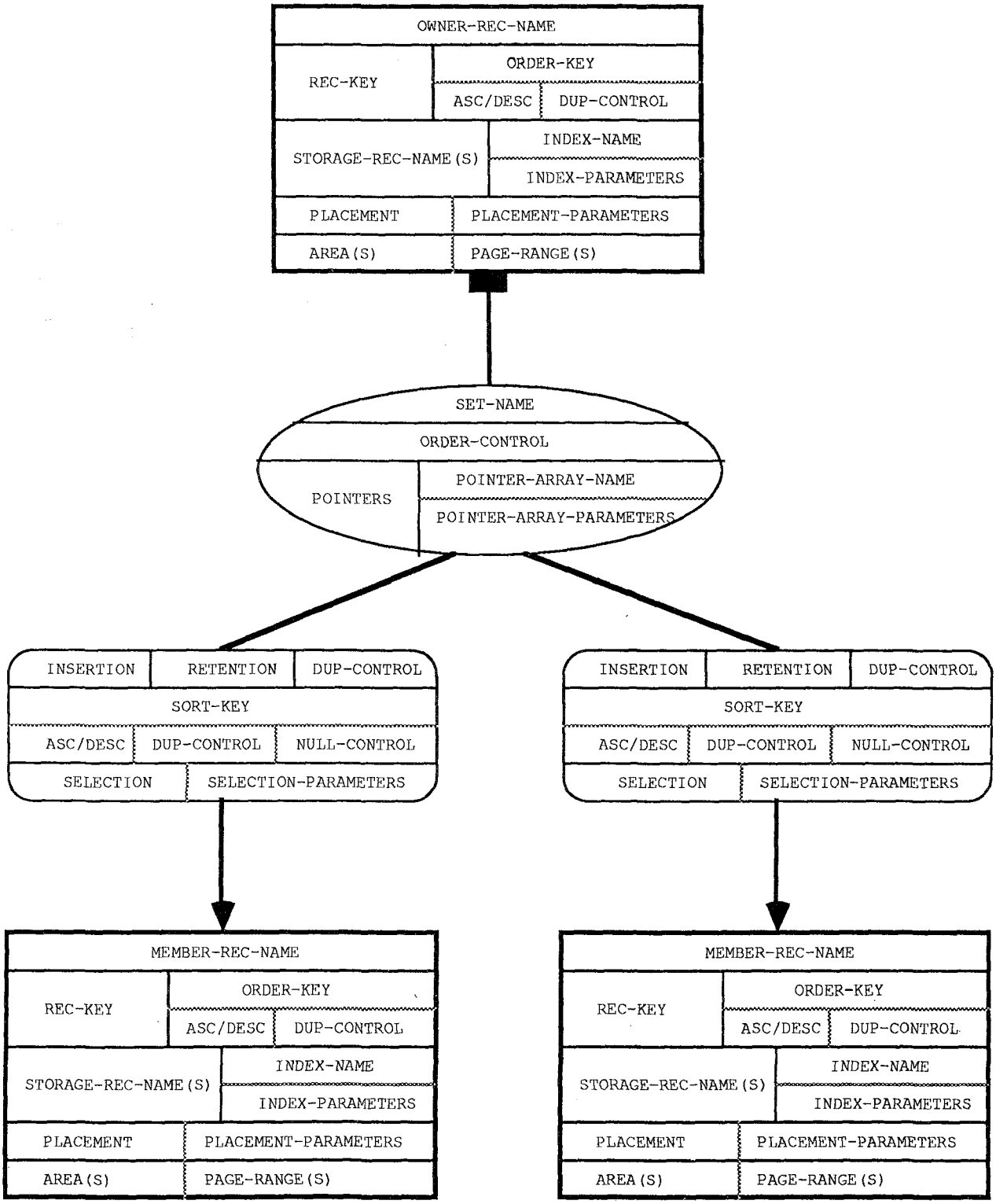


Figure 36. Graphical Definition of Physical Schema (Fork Set).

It is our opinion that the IDDK, when completed, will be of commercial value. With the rising sophistication of DBMS-s, there is a growing demand for computerized design tools. Currently, there is no network or relational production DBMS which provides for such tools even in the limited scope. Some of the network DBMS-s, however, are well prepared for that purpose as they interface with active Data Dictionary/Directory Software (e.g. IDMS). (Obviously, the IDDK, to be commercially viable, will have to be interfaced with a DD/D Package.)

We are of the opinion that most of the research in database design is too oversimplified to be directly applicable. The best one can say is that there is a number of interesting partial solutions which are not integrated in consistent methodologies. The only comprehensive approaches have been, to the best of our knowledge: (1) the DATAID project financed by the Italian Research Council (e.g. Orlando *et al.*, 1985), (2) the Interactive System for Database Design and Integration centered around the Functional Data Model (e.g. Yao *et al.*, 1985) and more recently (3) the Database Designer's Workbench based on the methodology of Teorey and Fry (1982) conducted by Computer Corporation of America and Harvard University (e.g. Reiner *et al.*, 1986). From all DBMS vendors, Cullinet has always stood out in providing reasonable design manuals for its IDMS and IDMS/R software. And only recently (October 1986), Cullinet announced that it contracted the development of a database software engineering tool. The scope of that tool is not known to this author at the time of writing, but it is believed to be inclined towards physical design.

Our methodology is similar in scope to the DATAID project (except that we do not address distributed databases). We believe, however, that our approach is better structured, evolves from a carefully defined extended entity-relationship approach, and is up to date with the ANSI standardization efforts. As well, it provides for versatile feedbacks and multiple iterations, and applies strong heursitics that eliminate (at early stages) paths of investigation that have little chance of success. Moreover, high-resolution graphics capabilities of readily available personal computers, i.e. Apple's Macintosh Plus, are being used for the IDDK implementation.

ACKNOWLEDGMENTS

The author wishes to thank William Bowie, Neil Gray, Michael Farnan, Wayne Findlay, Stephen Lucas, Mark Paine, Peter Roan and George Zamroz for their contribution to the implementation of the IDDK. I am grateful to Plenum Publishing Corporation for permission to reproduce in this report (Sections 3 and 4) much of Maciaszek (1985). I also thank Michael Farnan, Gary Stafford and Peter Strazdins for their comments on the draft of this report and for suggesting improvements to the presentation.

INDEX OF FIGURES

REFERENCES

ARONSON, J. (1977): *Data Compression - A Comparison of Methods,* Institute for Computer Sciences and Technology, National Bureau of Standards, U.S. Department of Commerce, June, 31pp.

BATORY, D.S. and GOTLIEB, C.C. (1982): A Unifying Model of Physical Databases, *ACM Trans. Database Syst.*, 4, pp.509-539.

BATORY, D.S. (1984): *Modelling the Storage Architectures of Commercial Database Systems*, Tech. Report TR-83-21, University of Texas at Austin, 58pp. (also, in a slightly modified version, in *ACM Trans. Database Syst.*, 1985, 4, pp.463-528).

BEIGHTLER, C.S. PHILLIPS, D.T. and WILDE, D.J. (1979): *Foundations of Optimization*, Prentice-Hall, 487pp.

CALINGAERT, P. (1982): *Operating System Elements. A User Perspective*, Prentice-Hall, 240pp.

CHRISTODOULAKIS, S. (1984): Implications of Certain Assumptions in Database Performance Evaluation, *ACM Trans. Database Syst.*, 2, pp.163-186.

CLAYBROOK, B.G. (1983): *File Management Techniques*, John Wiley & Sons, 247pp.

COOPER, R.B. and SOLOMON, M.K. (1984): The Average Time Until Bucket Overflow, *ACM*

Trans. Database Syst., 3, pp.392-408.

CORMACK, G.V. (1985): Data Compression on a Database System, *Comm. ACM*, 12, pp.1336-1342.

DAELLENBACH, H.G. GEORGE, J.A. and MCNICKLE, D.C. (1983): *Introduction to Operations Research Techniques*. Second Edition, Allyn and Bacon, 705pp.

DAYAL, U. and GOODMAN, N. (1982): Query Optimization for Codasyl Database Systems, *Proc. Int. Conf. on Management of Data SIGMOD*, Orlando, Florida, U.S.A., pp.138-150.

DEEN, S.M. (1985): *Principles and Practice of Database Systems*, MacMillan, 393pp.

DMS (1984): *Series 1100 Data Management System Schema Definition DMS 1100 Level 9R1*, Data Administration Reference, p. several number, Sperry.

DRAFT (1985): *Draft Proposed American National Standard Network Database Language*, Project 355-D, p. 142, Technical Committee X3H2 - Database, X3 Secretariat/CBEMA.

EFFELSBERG, W. and LOOMIS, M.E.S. (1984): Logical, Internal, and Physical Reference Behaviour in CODASYL Database Systems, *ACM Trans. Database Syst.*, 2, pp.187-213.

EISELT, H.A. and FRAJER VON, H. (1977): *Operations Research Handbook. Standard Algorithms and Methods with Examples*, MacMillan Prss, 398pp.

FERRARI, D. SERAZZI,G. and ZEIGNER, A. (1983): *Measurement and Tuning of Computer Systems*, Prentice-Hall, 523pp.

GERRITSEN, R. (1975): A Preliminary System for the Design of DBTG Structures, *Comm. ACM*, 10, pp.551-556.

GRAY, P.M.D. (1984): Implementing the Join Operation on Codasyl DBMS, in: *Database - Role and Structure. An Advanced Course*, ed. P.M.Stocker, P.M.D.Gray, M.P.Atkinson, Cambridge University Press, pp.185-205.

HEYMAN, D.P. (1982): Mathematical Models of Database Degradation, *ACM Trans. Database Syst.*, 4, pp.615-631.

IDMS (1980): *IDMS Part 2 Database Establishment* (IDMS.200/IDMSX.200), Reference, p. several number, ICL 2900.

IDMS (1982): *IDMS Part 5 Database Design* (IDMS.320/IDMSX.320), Reference, p. several number, ICL 2900.

IDMS/R (1984): *IDMS/R Database Operations* (Release 10.0, TDDB-0220-100G0), p. several number, Cullinet.

JAIN, H.K. (1984): A Comprehensive Model for Storage Structure Design of CODASYL Databases, *Inform. Syst.*, Vol.9, pp. 217-230.

JEFFREY, D.R. and LAWRENCE, M.J. (1984): *Systems Analysis and Design*, Prentice-Hall of Australia, 225pp.

KATZ, R.H. and WONG, E. (1983): Resolving Conflicts in Global Storage Design Through

Replication, *ACM Trans. Database Syst.*, 1, pp.110-135.

KENT, W. (1978): *Data and Reality. Basic Assumptions in Data Processing Reconsidered*, North-Holland, 211pp.

LIROV, Y. and DAUNOV, N. (1985): Heuristic Approach to Network Database External Parameters Design, *Inform. Syst.*, Vol.10, pp.311-316.

MACIASZEK, L.A. (1981): Database Design, in: *Design of Information Systems for Unified Computer Systems*, ed. E.Niedzielska, AE, pp.205-252 (in Polish).

MACIASZEK, L.A. (1985): Record to Area Mapping in a CODASYL Environment, *Proc. 2nd Int. Conf. on Foundations of Data Organization*, Kyoto, Japan, pp. 297-304 (the revised version in: *Foundations of Data Organization*, ed. S.P.Ghosh, Y.Kambayashi, K.Tanaka, Plenum Publ., 1986).

MACIASZEK, L.A. (1986): *An Enhanced Conceptual Structure Derivation*, University of Wollongong, Department of Computing Science, Preprint 86-1, 44pp.

MACIASZEK, L.A. BOWIE W.S. and LUCAS, S.K. (1986a): *On Derivation of Network and Relational Schemas from an Enhanced Conceptual Structure*, University of Wollongong, Department of Computing Science, Preprint 86-3, 38pp.

MACIASZEK, L.A. FARNAN, M.J. FINDLAY, W.T. PAINE, M.D. ROAN, P.M. and ZAMROZ, G.B. (1986b): *Computer-Assisted Derivation of a Feasible Conceptual Structure (Detailed Abstract)*, typescript, 14pp.

MARCH, S.T. and SEVERANCE, D.G. (1977): The Determination of Efficient Record Segmentations and Blocking Factors for Shared Data Files, *ACM Trans. Database Syst.*, 3, pp.279-293.

MARCH, S.T. (1983): Techniques for Structuring Database Records, *Comput. Surv.*, 1, pp.45-79.

MARCH, S.T. and SCUDDER, G.D. (1984): On the Selection of Efficient Record Segmentations and Backup Strategies for Large Shared Databases. *ACM Trans. Database Syst.*, 3, pp.409-438.

MERRETT, T.H. (1984): *Relational Information Systems*, Reston Publishing Co., 507pp.

NIEVERGELT, J. HINTERBERGER, H. and SEVCIK, K.C. (1984): The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM Trans. Database Syst.*, 1, pp.38-71.

ORLANDO, S. RULLO, P. SACCA, D. and STANISZKIS, W. (1985): Integrated Tools for Physical Database Design in CODASYL Environment, in: *Computer-Aided Database Design. The DATAID Project*, ed. A. Albano, V. De Antonellis, A. Di Leva, Elsevier Science Publishers B.V., pp.131-153.

RDMS (1985): *Series 1100 Relational Data Management System RDMS 1100*, Level 1R2, Release Description, p. several number, Sperry.

REINER, D. BRODIE, M. BROWN, G. CHILENSKAS, M. FRIEDELL, M. KRAMLICH, D. LEHMAN, J. and RESENTHAL, A. (1986): A Database Design and Evaluation Workbench: Preliminary Report, in : VESELY, E.G.: *The Practitioner's Blueprint for Logical and Physical Database Design*, Prentice-Hall, pp.268-275.

REPORT (1978): Report of the CODASYL Data Description Language Committee, *Inform. Syst.*, Vol.3, pp. 247-320.

REUTER, A. and KINZINGER, H. (1984): Automatic Design of the Internal Schema for a CODASYL Data Base System, *IEEE Trans. on Soft. Eng.*, 4, pp.358-375.

RODAN (1979): *Universal Database Management System RODAN*, User Manuals, p.several number, CPiZI (in Polish; English version available from CPiZI, Warsaw, Poland).

SEVCIK, K.C. (1981): Data Base Performance Prediction Using an Analytical Model, *Proc. 7th Int. Conf. Very Large Data Bases*, Cannes, France, pp.182-198.

SEVERANCE, D.G. (1983): A Practitioner's Guide to Data Base Compression. Tutorial, *Inform. Syst.* Vol.8, No.1, pp.51-62.

STANISZKIS, W. and RULLO, P. (1982): *Transaction Workload Analysis in the CODASYL Data Base Performance Predictor EOS*, Rapporto CRAI 82-17, 109pp.

STANISZKIS, W. RULLO, P. and GAUDIOSO, M. (1982): *Probabilistic Approach to Evaluation of Data Manipulation Algorithms in a CODASYL Data Base Environment,* Rapporto CRAI 82-16, 52 plus pp.

STANISZKIS, W. SACCA', D. MANFREDI, F. and MECHIA, A. (1983): Physical Data Base Design for CODASYL DBMS, in: *Methodology and Tools for Data Base Design*, ed. S. Ceri, North-Holland, pp.119-148.

TEOREY, T.J. and FRY, J.P. (1982): *Design of Database Structures,* Prentice-Hall, 495pp.

VAX (1984): *VAX DBMS Database Design Guide*, Order No. AA-Y311A-TE, p. several number, Digital Software.

WELCH, T.A. (1984): A Technique for High-Performance Data Compression, *IEEE Comp.*, June, pp.8-19.

WHANG, K.-Y. WIEDERHOLD, G. and SAGALOWICZ, D. (1981): Separability - an Approach to Physical Database Design, *Proc. 7th Int. Conf. Very Large Data Bases,* Cannes, France, pp.320-332.

WHANG, K.-Y. WIEDERHOLD, G. and SAGALOWICZ, D. (1982): Physical Design of Network Model Database Using the Property of Separability, *Proc. 8th Int. Conf. Very Large Data Bases,* Mexico City, Mexico, pp.98-107.

WHANG, K.-Y. (1985): Property of Separability in Physical Design of Network Model Databases, *Inform. Syst.*, Vol.10, pp.57-63.

WIEDERHOLD, G. (1983): *Database Design*, Second Edition, McGraw-Hill, 751pp.

YAO, S.B. WADDLE, V. and HOUSEL, B.C. (1985): An Interactive System for Database Design and Integration, in: *Principles of Database Design Volume I Logical Organizations,* ed. S.B.Yao, Prentice-Hall, pp.325-360.

YU, C.T. SUEN, C.-M. LAM, K and SIN, M.K. (1985): Adaptive Record Clustering, *ACM Trans. Database Syst .*, 2, pp. 180-204.