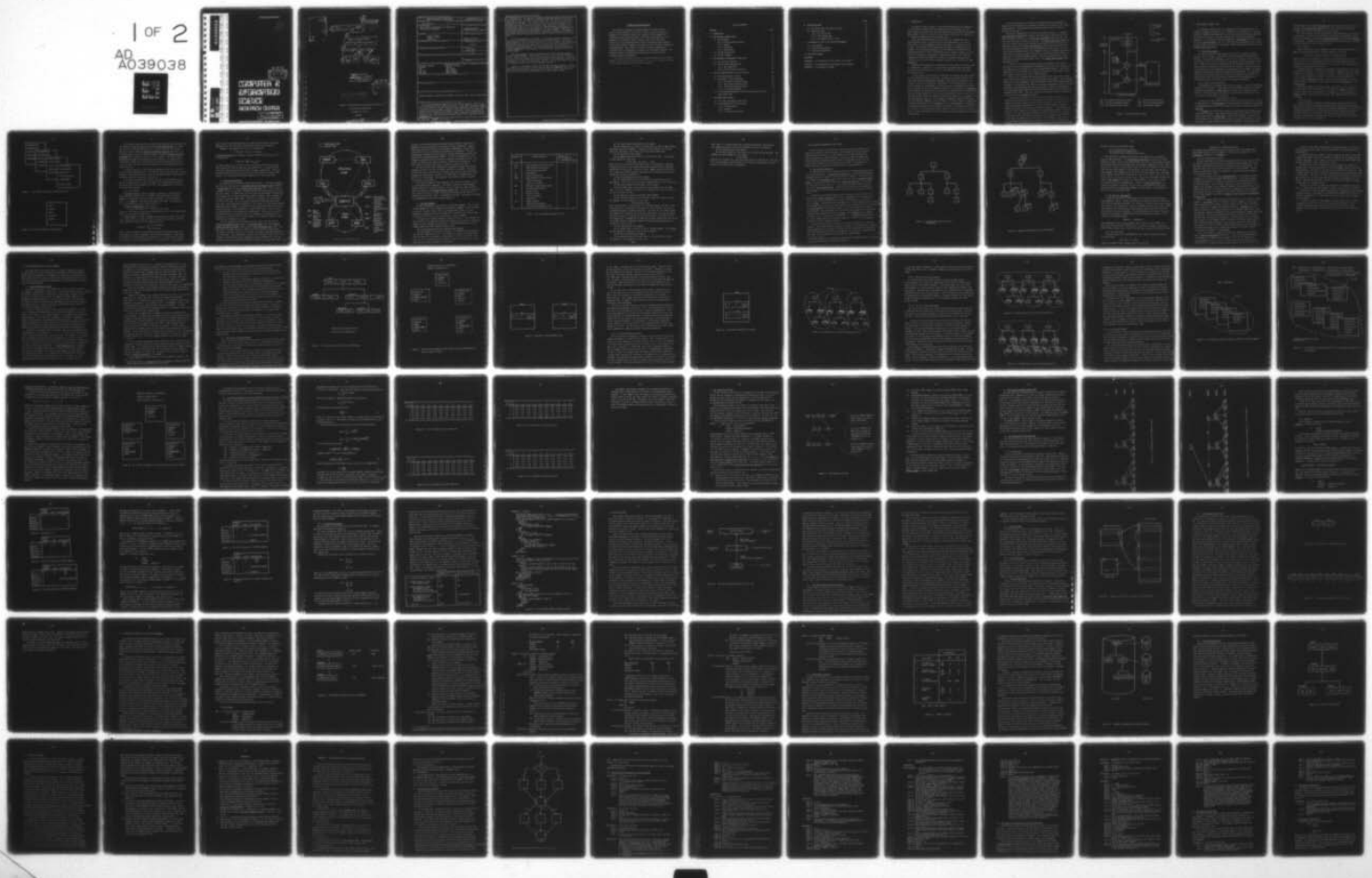
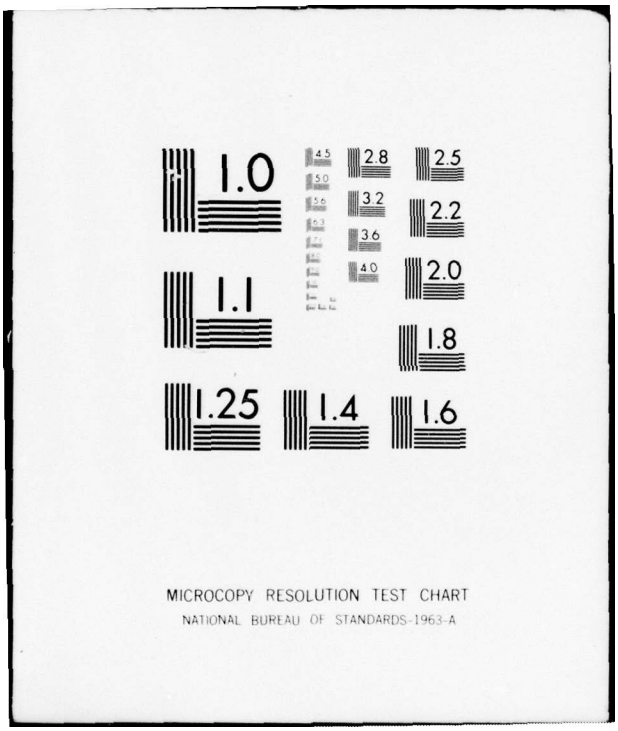
AD-A039 038 OHIO STATE UNIV COLUMBUS COMPUTER AND INFORMATION SC--ETC F/G 5/2
DBC SOFTWARE REQUIREMENTS FOR SUPPORTING HIERARCHICAL DATABASES--ETC(U)
APR 77 D K HSIAO, D S KERR, F K NG N00014-75-C-0573
UNCLASSIFIED OSU-CISRC-TR-77-1 NL

1 OF 2
AD A039038

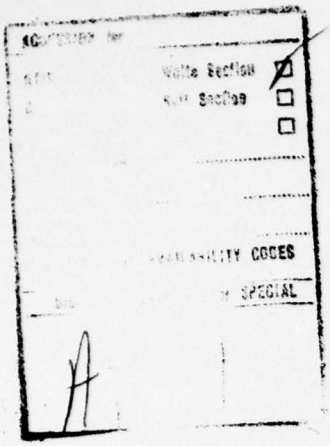



1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

TECHNICAL REPORT SERIES

COMPUTER & INFORMATION SCIENCE RESEARCH CENTER

THE OHIO STATE UNIVERSITY COLUMBUS, OHIO



(14) OSU-CISRC-TR-77-1

(9) Technical Rept.

(1)

(6)

DBC SOFTWARE REQUIREMENTS FOR SUPPORTING HIERARCHICAL DATABASES

by

(10) David K. Hsiao, Douglas S. Kerr and Fred K. Ng

(11) Apr 77 (12) 105 p.

Work performed under
(15) Contract N00014-75-C-0573
Office of Naval Research

Computer and Information Science Research Center
The Ohio State University
Columbus, OH 43210

April 77

407586

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER OSU-CISRC-77-1	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) "DBC Software Requirements for Supporting Hierarchical Databases"		5. TYPE OF REPORT & PERIOD COVERED Technical Report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) David K. Hsiao Douglas S. Kerr Fred K. Ng		8. CONTRACT OR GRANT NUMBER(s) N00014-75-C-0573
9. PERFORMING ORGANIZATION NAME AND ADDRESS		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 4115-A1
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE April 1977 13. NUMBER OF PAGES 98
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited.

Scientific Officer	DDC New York Area
ONR BRO	ONR 437
ACO	ONR, Boston
NRL 2627	ONR, Chicago
ONR 102IP	ONR, Pasadena

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Database Computer; hierarchical database, DL/1 calls, IMS; symbolic identifiers.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report shows the capability of the database computer (DBC) for supporting hierarchical data models and systems. Since IBM's Information Management System (IMS) is the most widely used data management system which supports a hierarchical view of data, we intend to show that the DBC can support IMS databases and IMS application programs. Furthermore, this study provides us with a case for comparing the merits of using a conventional general-purpose computer versus a special-purpose database computer (i.e., the DBC) for data management.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

CONTINUED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

It is shown that it is possible to design an interface, known as IMSI between IMS users and the DBC. The IMSI can faithfully execute the DL/1 calls (the data manipulation language of IMS) issued by IMS users. The design of the IMSI is considered in two phases. In the first phase, we show how to represent an IMS database utilizing the built-in (hardware) data structure of the DBC. This representation makes use of the concept of embedding symbolic identifiers into all dependent segments of an IMS database. The use of symbolic identifiers increases the degree of data independence of the stored database. Furthermore, the storage requirement for the symbolic identifiers is substantially offset by the removal of the conventional address pointers currently used in an IMS database.

In the second phase of the design, a translation process is designed to emulate every IMS operation (as specified in terms of DL/1 calls) on the transformed database. The two phases are therefore closely related. The translation algorithms are presented to show, first, that any DL/1 call can automatically be supported by the IMSI and, second, that the algorithms can be written in a fairly straightforward manner. The latter demonstrates that the degree of the software complexity and the amount of software support required by the IMSI is rather minimal.

The data management functions performed by the IMSI, although complete, require little software. In particular, the IMSI is freed from performing any kind of content-searching, thus eliminating the software required to search the system buffer. The content-addressable capability of the DBC makes possible the elimination of software buffer search. The use of the DBC can therefore result in a reduction of the role of data management software.

Finally, a comparative study of IMS and DBC performance is given. In the conclusion, we try to point out that not only IMSI can **outperform** IMS, but the DBC can provide additional services such as advanced security and clustering which are found neither in IMS software nor in IBM 370/360 hardware.

PREFACE AND ACKNOWLEDGEMENT

This work was supported by Contract N00014-75-C-0573 from the Office of Naval Research to Dr. David K. Hsiao, Associate Professor of Computer and Information Science, and conducted at the Computer and Information Science Research Center of The Ohio State University. The Computer and Information Science Research Center of The Ohio State University is an interdisciplinary research organization which consists of the staff, graduate students, and faculty of many University departments and laboratories. This report is based on research accomplished in cooperation with the Department of Computer and Information Science. The research contract was administered and monitored by The Ohio State University Research Foundation.

The authors would like to thank Mr. Javanta Banerjee for his careful reading of the manuscript and his comments of the work. Thanks is also due to Mr. Lorenzo Aquilar for helping out on examples.

TABLE OF CONTENTS

1. INTRODUCTION

We intend to issue a series of reports which demonstrate the capability of the database computer (DBC) for supporting known data models (such as the hierarchical, network and relational models) and their related data management systems. This is the first of these reports which shows the capability of the DBC for supporting the hierarchical model.

IBM's Information Management System (IMS), which implements a hierarchical model of data, is chosen for the following reasons. IMS is the most widely used data management system which supports a hierarchical view of data. The extensive IMS documentation also makes it possible for us to compare the anticipated performance of an IMS database on the DBC with the present performance of IMS on IBM 360/370 computer systems. Finally, the fact that IMS is available on IBM 360/370 computers provides us with a case for comparing the merits and demerits of using a conventional general-purpose computer versus a special-purpose database computer (i.e., the DBC) for data management.

Background descriptions of the DBC and of IMS are given in Sections 2 and 3, respectively. However, for a more detailed and authorative description on the DBC, one should refer to [1,2,3]. For reference to IMS, one should look into [4,5,6,7].

In this report, we show that it is possible to design an interface, known as IMSI between IMS users and the DBC. The IMSI can faithfully execute the DL/1 calls (the data manipulation language of IMS) issued by IMS users. The design of the IMSI is considered in two phases. In the first phase, we show how to represent an IMS database utilizing the built-in (hardware) data structure of the DBC so that the information stored in the IMS database is preserved. By preservation of information, we mean that any information that can be derived from an IMS database by DL/1 calls can also be derived by the same calls from the DBC database without any change of information content (i.e., semantics). The DBC representation makes use of the concept of embedding symbolic identifiers into all dependent segments of an IMS database. The use of symbolic identifiers increases the degree of data independence of the stored database. Furthermore, the storage requirement for the symbolic identifiers is substantially offset by the removal of the conventional address pointers currently used in an IMS database. The DBC representation and the resulting storage requirements are discussed in Section 4.

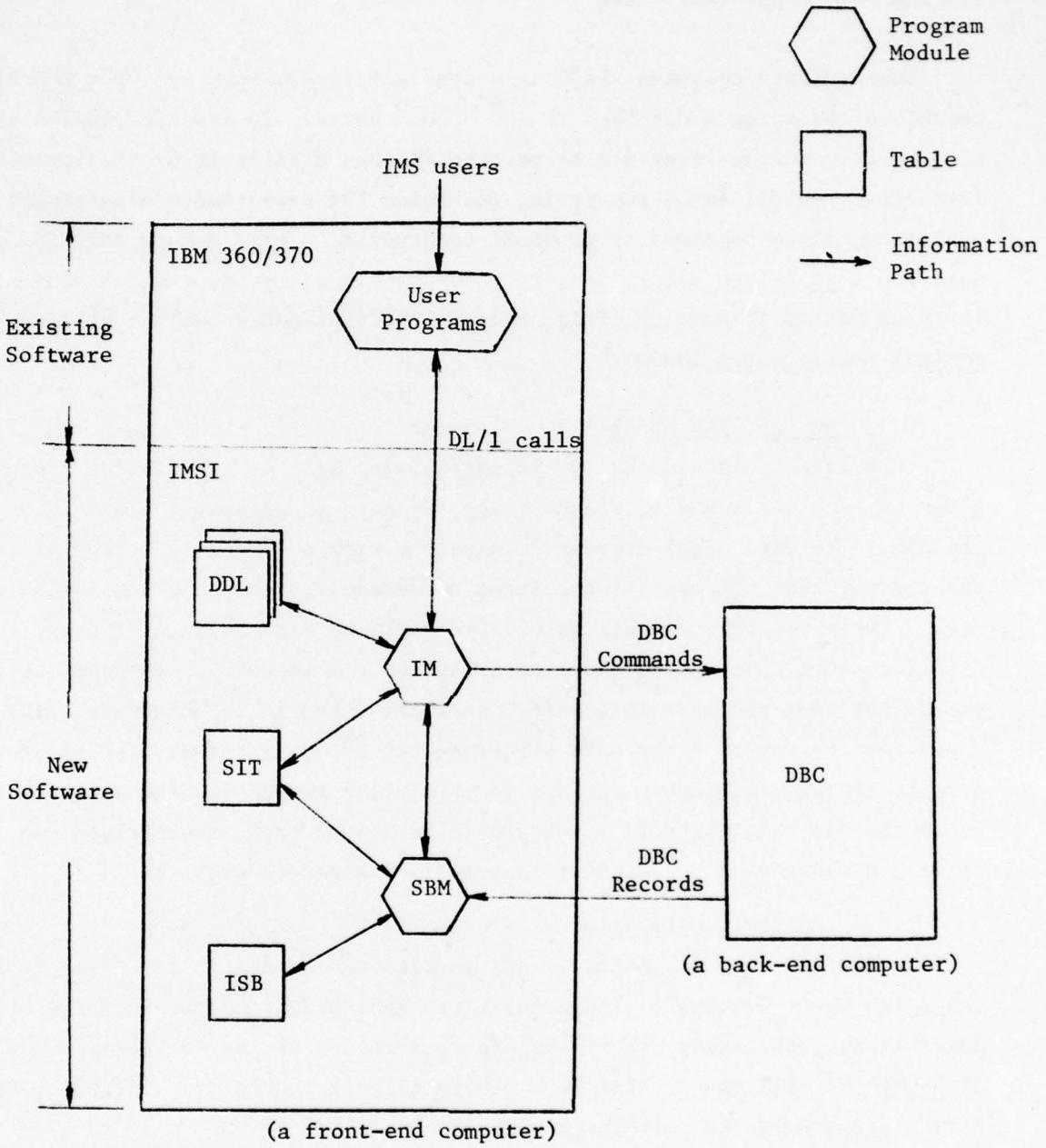
In the second phase of the design, a translation process is designed to emulate every IMS operation (as specified in terms of DL/1 calls) on the transformed database. The two phases are therefore closely related. These are discussed in Section 5.

The overall organization of the IMSI is depicted in Figure 1. The database description library (DDL) contains coded information about the IMS database as defined by the user. The interface system buffer (ISB) contains segments that have been retrieved from the DBC. The status information table (SIT) gives the current status of the interface system buffer (ISB). When the IMSI receives a DL/1 call, the DL/1 interface module (IM) decodes and executes the call using information stored in the aforementioned library, buffer and table. If the status information table (SIT) indicates that the interface system buffer (ISB) does not contain the needed information, the interface module (IM) will issue the necessary DBC commands needed to fetch DBC records and will insert them into the interface system buffer (ISB). The system buffer manager (SBM) maintains the ISB.

In Section 5, the translation algorithms are presented to show, first, that any D1/1 call can automatically be supported by the IMSI and, second, that the algorithms can be written in a fairly straightforward manner. The latter demonstrates that the degree of the software complexity and the amount of software support required by the IMSI is rather minimal.

In Section 6, we discuss the storage organization of the interface system buffer (ISB) and the functions of the system buffer memory (SBM). The discussion shows that the data management functions performed by the IMSI, although complete, are simple. In particular, the IMSI is freed from performing any kind of content-searching, thus eliminating the software required to search the system buffer. The content-addressable capability of the DBC makes possible the elimination of software buffer search. The use of the DBC can therefore result in a reduction of the role of data management software.

A performance evaluation of the IMSI design in given in Section 7. A comparative performance study is made between IMS and IMSI. The study is concentrated on several typical cases of search and update. The result of the study is gratifying, indicating that IMSI with the DBC support is indeed a favorable alternative to hierarchical database management. The efficiency of using the DBC to support an IMS database is, nevertheless, limited by the use of DL/1. Since the transformed database is stored in the DBC in a more data-independent manner, more efficient processing of the database can be achieved if new calls, in addition to the DL/1 calls, are introduced. Thus we also show the additional capabilities of using the DBC to support a hierarchical database when the data manipulation language employed is not restricted to DL/1.



DDL: The Database Description Library
SIT: The Status Information Table
ISB: The Interface System Buffer
IM: The DL/1 Interface Module
SBM: The System Buffer Manager
DBC: The Database Computer

Figure 1. The IMS Interface (IMSI)

2. THE DATABASE COMPUTER (DBC)

The database computer (DBC) is a specialized back-end computer which is capable of managing a database of 10^9 - 10^{10} bytes. In addition to its intended purpose of handling large databases, the DBC has a built-in protection mechanism for access control and a clustering mechanism for performance enhancement. Basically, it is composed of two main components, a mass memory (MM) for storing data and a structure memory (SM) for storing directory information about the data. The mass memory is made up of fixed-length content-addressable partitions called minimal access units (MAUs).

2.1 The DBC Data Model

In order to discuss the DBC representation and the translation process given in Sections 4 and 5, respectively, we need to present the data model of the DBC. The data model represents a user's view of the data stored in the DBC and the way that the user is permitted to manipulate the data according to that view. There are four aspects associated with the data model of the DBC: the data structure, the query, the clustering and the security. The data structure is the way that the user information is represented in the hardware. The query specifies the way that the data structure can be manipulated. The clustering effects the way the data structure is physically stored and the security controls the way that the data structure is protected from unauthorized use. We will discuss each of the four aspects in a separate subsection.

2.1.1 Built-in Data Structures

The definition of a database starts with two terms: a set AT of "attributes" and a set VA of "values". These terms are left undefined to allow the broadest possible interpretation. A DBC record is a subset of the Cartesian product AT x VA. We will assume that in a record all attributes are distinct. Thus, R is a set of ordered pairs of the form:

(an attribute, a value)

The set of all DBC records which are physically stored in the DBC is called the DBC database. The DBC database may be partitioned into subsets called files. To distinguish among several files, each file is given a unique name called its file name.

The keywords of a DBC record are those attribute-value pairs which characterize the DBC record. Other attribute-value pairs of the record, if any, are collectively called the record body. A DBC record, therefore, consists of a set of keywords and a (possibly empty) string of characters referred to as the record body. The DBC recognizes two types of keywords: those stored in the

structure memory (SM), called directory type (Type D),and those not stored in the SM, called non-directory type (Type N). The selection of keywords for storage in the SM will be discussed in a later section.

DBC records having the same attributes are often singled out for discussion. In such cases, we show only the attributes without their corresponding values. The set of attributes is called the attribute template (or, for short, template) of the records. Examples of records having identical attributes and the corresponding template are shown in Figures 2 and 3.

2.1.2 Queries

Queries are used in access commands to retrieve and update DBC records. A query is a Boolean expression of keyword predicates of the form:

<attribute, relational operator, comparative value>

where attribute is an attribute of a keyword, a relational operator is one of the set $\{=,\neq,<,\leq,>,\geq\}$ and comparative value is the value against which the keyword value of the specified attribute is to be tested. The queries will be expressed in disjunctive normal form.

A keyword predicate is true for a DBC record if some keyword in the record satisfies the predicate. A conjunct of predicate is true for a record if each predicate in the conjunct is true for the record. A query is true for a record if one or more conjuncts in the query is true for the record; such a DBC record is said to satisfy the query. The set of all DBC records in a file of the DBC database that satisfy a query will be called its response set or the response data.

Some simple examples of queries follow. The query $K_1 \wedge K_2$ is true for a record R when K_1 and K_2 are both in R. The query $K_1 \wedge$ (Salary $<$ 10,000) is true for R when K_1 is in R and there is a keyword in R whose attribute is Salary and whose value is less than 10,000. More elaborate queries can also be formed.

2.1.3 Clustering

The DBC, instead of supporting a fixed record placement scheme for all DBC records, has a record placement mechanism which carries out record placement polices supplied by the DBC users and database administrator. The clustering mechanism allows a DBC user to have some control over the physical placement of a DBC record when it is inserted into the mass memory (MM) of the DBC. Physical placement means assigning a DBC record to a MAU (a unit of physical closeness) in which it can be placed.

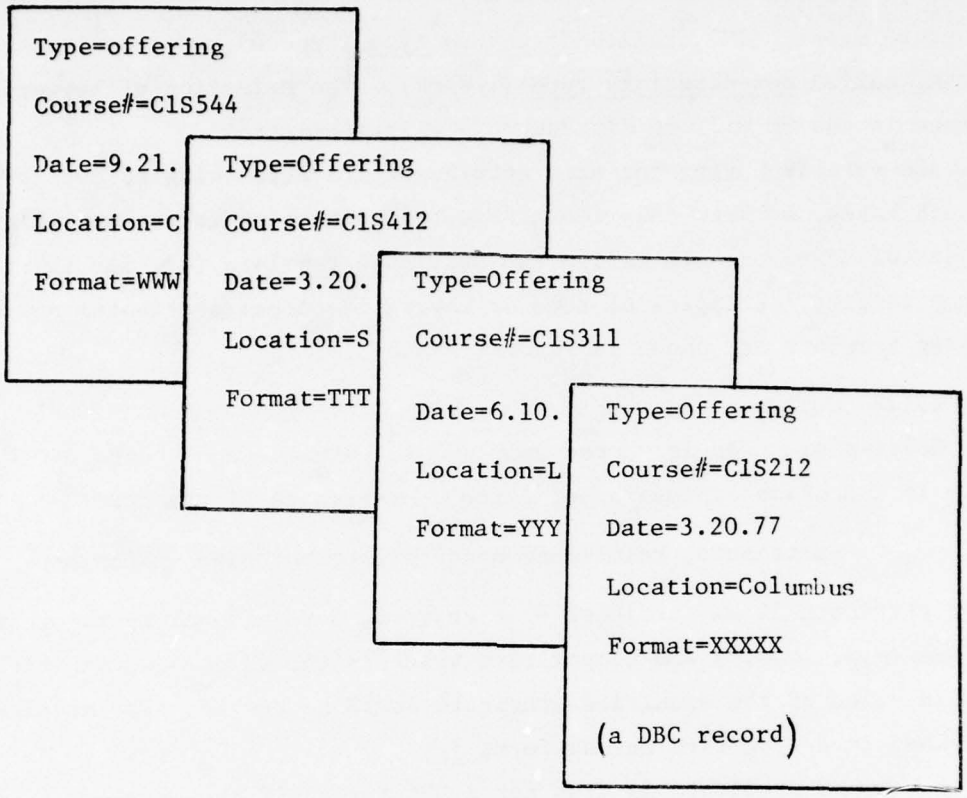


Figure 2. A set of DBC records with common attributes.

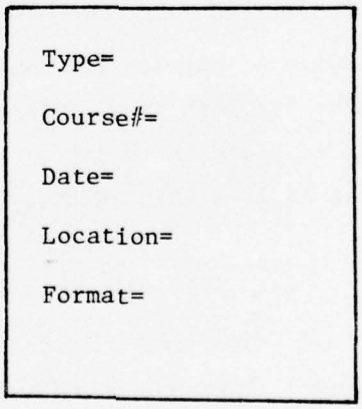


Figure 3. An attribute template of DBC records.

For each DBC record, there are certain type-D keywords which are designated for record placement purposes and are called clustering keywords. In order for the DBC to determine which MAU to be used for insertion, the DBC user may give a query, called the mandatory clustering condition (MCC), and a set of ordered pairs of the form (query, integer), called the optimal clustering conditions (OCCs), whenever a record is to be inserted. The MCC is used to determine which MAUs are eligible for record insertion. An eligible MAU is a MAU which contains one or more records that satisfy the MCC of the record to be inserted. The OCCs are used to choose one of the eligible MAUs determined by the MCC.

In order to make use of the clustering mechanism of the DBC, one has to choose a record placement policy. For each DBC record to be inserted, one must designate those keywords which are used as clustering keywords and specify the clustering conditions according to the policy.

In utilizing the same query facility to specify a clustering condition, any set of DBC records which can be retrieved by a single query can also be placed physically close together by the DBC. This demonstrates the power of the clustering mechanism.

2.1.4 The Security

A database access or simply an access is the name of a DBC operation which transfers information to or extracts information from the database. Examples of accesses are retrieve, insert and delete. Let ACC denote the set of all names of the accesses available in the DBC. Let a member of ACC be represented by a and a subset of ACC by A.

A security specification is a relation

$$S: \ DB \rightarrow 2^{ACC} \text{ where } 2^{ACC} \text{ is the power set of ACC.}$$

Thus, for a DBC record R, the security specification, S(R) = A, indicates which subset A of accesses is permitted on R.

A file sanction or simply a sanction is defined as the couple (Q,A) where Q is a query, and A is a subset of ACC. A sanction (Q,A) induces a relation $S.FS_{Q,A}$ over DBC records R of the database such that

$$S.FS_{Q,A}(R) = \begin{cases} A \text{ if R satisfies Q.} \\ ACC, \text{ otherwise.} \end{cases}$$

Thus, a sanction induces a security specification which indicates that only the accesses in A may be performed on the records satisfying Q. When R does not satisfy Q, all accesses may be performed on it. In this case we say that no sanctions of (Q,A) are applicable to R. The sanction is a very powerful

type of security specification since it allows the full power of the query language (i.e., Q) to be used to specify DBC records to be protected.

Consider a file named F and a set of sanctions where

$$S = \{(Q_1,A_1), (Q_2,A_2),\ldots,(Q_m,A_m)\}.$$

A database capability (F,S) induces a security specification $S.DC_{F,S}$ over R of F such that

$$S.DC_{F,S}(R) = \bigcap_{i=1}^{m} S.FS_{Q_i,A_i}(R)$$

In words, $S.DC_{F,S}(R)$ is the set of all access granted for R by one or more file sanctions in S and not denied by any sanction of S. Security specifications are therefore stored in the DBC as database capabilities. The database capabilities specify exactly what access operations are allowed on DBC records. The DBC maintains database capabilities for each active user.

2.2 The DBC Architecture

The architecture of the DBC is depicted in Figure 4. The three components that form the data loop are the database command and control processor (DBCCP), the mass memory (MM) and the security filter processor (SFP). When a query in a command is sent to the DBCCP, the DBCCP decodes the query using the structure loop (see below). The DBCCP then uses the structural information returned from the structure loop to form localized mass memory commands. These commands are then sent to the mass memory (MM). The MM is the repository of the database of 10^9 or 10^{10} bytes. It is realized by modifying conventional moving head disks with content-addressable capability. The concept of a partitioned content-addressable memory (PCAM) is utilized to access large partitions and to perform content-searching of the partitions in a cost-effective way. Only a part of the database needs to be searched for a given query. This is due to the information provided by the structure loop. The response set resulting from the operations of the MM is sent to the SFP where it is checked for security clearance and then forwarded to the user.

The four components which form the structure loop of the DBC are the keyword transformation unit (KXU), the structure memory (SM), the structure memory information processor (SMIP) and the index translation unit (IXU). The KXU converts keywords sent by the DBCCP into their internal representation. The primary function of the SM is to retrieve and update structural information of the database. This information is likely to be large (10^7 - 10^9) bytes. Furthermore, the operations on this information must be performed at a rate commensurate with that of database operations performed by the mass memory (MM). The

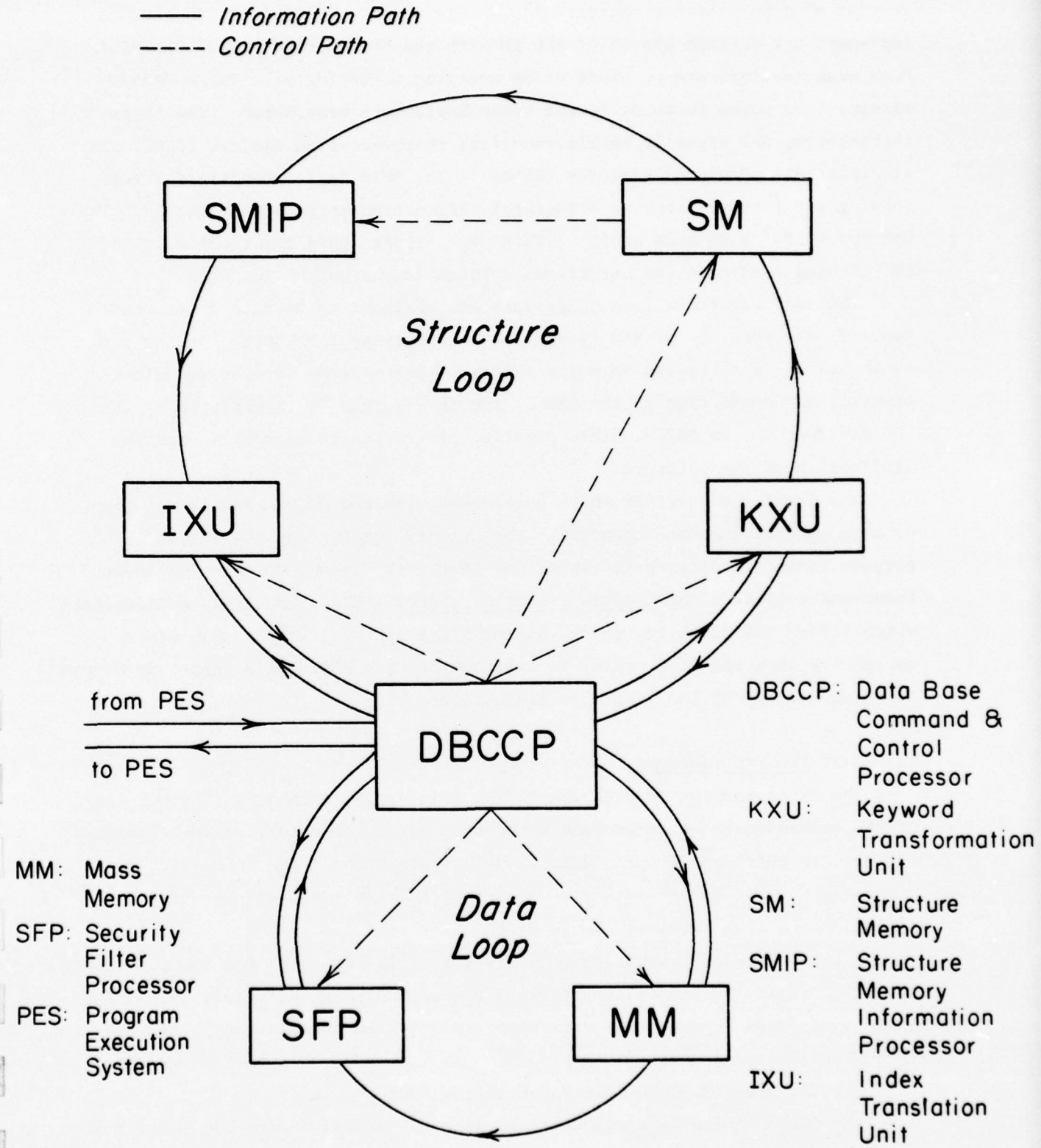


Figure 4. Architecture of DBC

concept of a partitioned content-addressable memory (PCAM) is again used to implement the storage system of the SM with the above properties. Powerful PCAM organizations are possible using emerging technologies. Three design alternatives using three different technologies are considered. The three technologies are magnetic bubble memories, charge-coupled devices (CCDs) and electron beam addressed memories (EBAMs). The SMIP is responsible for performing set intersections on structural information retrieved by the SM. The concept of PCAMs is once again utilized to perform rapid intersection. The IXU is used to decode the structural information output by the SMIP.

The four structure loop components are designed to operate concurrently. Keywords are sent to the KXU at regular intervals by the DBCCP. The output of the KXU is sent to the SM which retrieves index terms for the transformed keywords and sends them to the SMIP. The SMIP output is interpreted by the IXU and sent to the DBCCP. This pipeline of processors results in maximum utilization of the hardware.

The front-end computer which interfaces with the DBC is likely to be one or more general-purpose computers. The primary function of these general-purpose computers is program execution. For this reason, we term all those front-end computers the program execution system (PES). The IMSI, an interface which allows IMS users to query hierarchical databases in the DBC and to manipulate them via DL/1 calls, is intended for the PES. This report is devoted to an exposition of the IMSI (See again Figure 1).

2.3 The DBC Commands

The PES communicates with the DBC by issuing DBC commands. The two types of DBC commands are access commands and preparatory commands. Access commands are used to retrieve, insert, delete, and update DBC records in a file. Preparatory commands are issued to manage file information and security specifications so that the user can facilitate the access commands.

A repertoire of the DBC commands is listed in Table I. The commands that will be used in the translation process are marked by an asterisk. Simplified formats of these commands are presented. A detailed description of all the command formats and their usage are given in [3].

(1) The Open-Database-File-for-Creation (ODFC) Command

The ODFC Command is required to be sent to the DBC before DBC records of the file are loaded into the DBC. With respect to a file, this command provides information on the number of attributes of the file, the number of MAUs that need to be allocated initially, and the number of MAUs that may be allocated if the initial allocation is insufficient.

Mnemonics	Command Function	Command Type (A - access type P - preparatory type)
ODFC	*Open Database File for Creation	P
LAI	*Load Attribute Information	P
	Load Security Descriptor	P
LR	*Load Record	A
CDF	*Close Database File	P
ODFA	*Open Database File for Access	P
	Retrieve by Query	A
	Retrieve by Pointer	A
RQP	*Retrieve by Query with Pointer	A
	Retrieve Within Bounds	A
IR	*Insert Record	A
DQ	*Delete by Query	A
	Delete by Pointer	A
	Delete File	P
RR	*Replace Record	A
	Retrieve MAU Addresses	P
	Load Creation Capability List	P

Table I List of Commands Recognized by DBC

(2) The Load-Attribute-Information (LAI) Command

The LAI Command must be issued to the DBC after the ODFC or ODFA command and before any access commands. It supplies the attributes of a file and causes the attributes to be loaded into the structure memory (SM).

(3) The Load-Record (LR) Command

The LR command loads DBC records of a file into the (MM). The argument of this command is of the form:

$$\text{record, MCC, } OCC_1, OCC_2, \ldots, OCC_n$$

where record is a DBC record to be loaded. MCC is the mandatory clustering condition and OCCs are the optional clustering conditions used to determine the MAU for the DBC record. Execution of this command causes the DBC record to be loaded into the DBC database according to the clustering policy specified by the clustering conditions.

(4) The Close-Database-File (CDF) Command

The CDF command indicates to the DBC that the specified file may be deactivated. There will be no more access commands from the user on this file.

(5) The Open-Database-File-for-Access (ODFA) Command

The ODFA command opens a file which has been created in the DBC database. The access commands (discussed in the following) may then be issued to the DBC to process the file.

(6) The Retrieve-by-Query-with-Pointer (RQP) Command

The RQP command retrieves from a file DBC records that satisfy a query. The argument for this command is of the form:

sort-attribute, query

where sort-attribute is the attribute of the keyword used by the DBC to sort the response set on the basis of the values of the attribute. The query is a Boolean expression of keyword predicates. Execution of this command yields the response set containing all DBC records that satisfy the query in sorted order with respect to the values of the sort-attribute. Each DBC record in the response set is assigned by the DBC with a pointer which allows fast access to the DBC record for subsequent update and deletion operations without involving the structure loop.

(7) The Insert-Record (IR) Command

The IR command adds a record to a file in the DBC database. The argument of this command is identical to that of the LR command.

(8) The Delete-by-Query (DQ) Command

The DQ command deletes DBC records from a file that satisfy a query. The argument of this command is of the form:

query

where query is a Boolean expression of keyword predicates. Execution of this command causes all records that satisfy the query to be deleted.

(9) The Replace-Record (RR) Command

The RR command replaces a DBC record in a file of the DBC database. The argument for this command is of the form:

<ptr, record>

where ptr is a pointer for a DBC record and record is a DBC record. The execution of this command causes the DBC record located by the pointer to be replaced by the one given in the argument.

3. THE INFORMATION MANAGEMENT SYSTEM (IMS)

In this section, we introduce only those IMS facilities which will be used for discussion in later sections. The introduction will be centered around the data structure of an IMS database and the data language DL/1 used to manipulate the data structure. We will not, however, address the actual application program structure and its relationship to IMS, both of which are not relevant to the interface with the DBC. For simplicity, a slightly different syntax of the DL/1 language is employed herein.

3.1 The IMS Data Structure

The definition of an IMS database begins with the term segment type which will be left undefined. A logical data structure is a hierarchical structure of segment types, an example of which is shown in Figure 5. The segment type A is called the root segment type and the others are called dependent segment types. Each dependent segment type has a parent segment type. For instance B is the parent segment type of E. Similarly, each parent segment type has one or more child segment types. The successive parent-child relationships define levels. A is at the first level, B and G are at the second level and so on.

A logical data structure defines an IMS database in which there may be zero or more segment occurrences (or simply, segments) for each segment type in the logical data structure and each segment occurrence may contain one or more fields. Associated with an occurrence of a parent segment type are zero or more occurrences of each of its child segment types, collectively called the children (or child segments) of the parent segment. Each child segment has a unique parent segment. All occurrences of a particular child segment type which share a common parent segment are said to be twins. The descendants of a segment occurrence are its children, their children, etc. A database record consists of a root segment and all its dependent segments. Finally, an IMS database consists of all the database records.*

A convention for representing an IMS database schematically is shown in Figure 6. Each labeled square represents a segment. The relationship between a parent segment and its children of a given type is represented by a line from the parent segment to the first child segment. The twins of a particular segment type are illustrated by stacking them one after another.

---

*The reader should note the difference between an IMS database record as defined here and a DBC record as defined in Section 2.1.

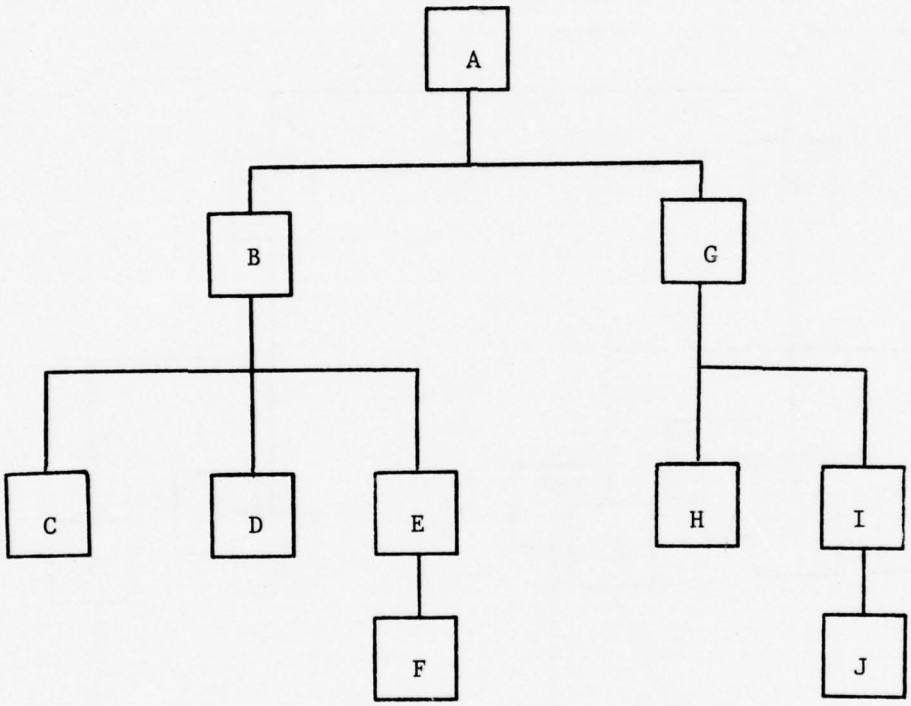


Figure 5. A Logical Data Structure for an IMS Database

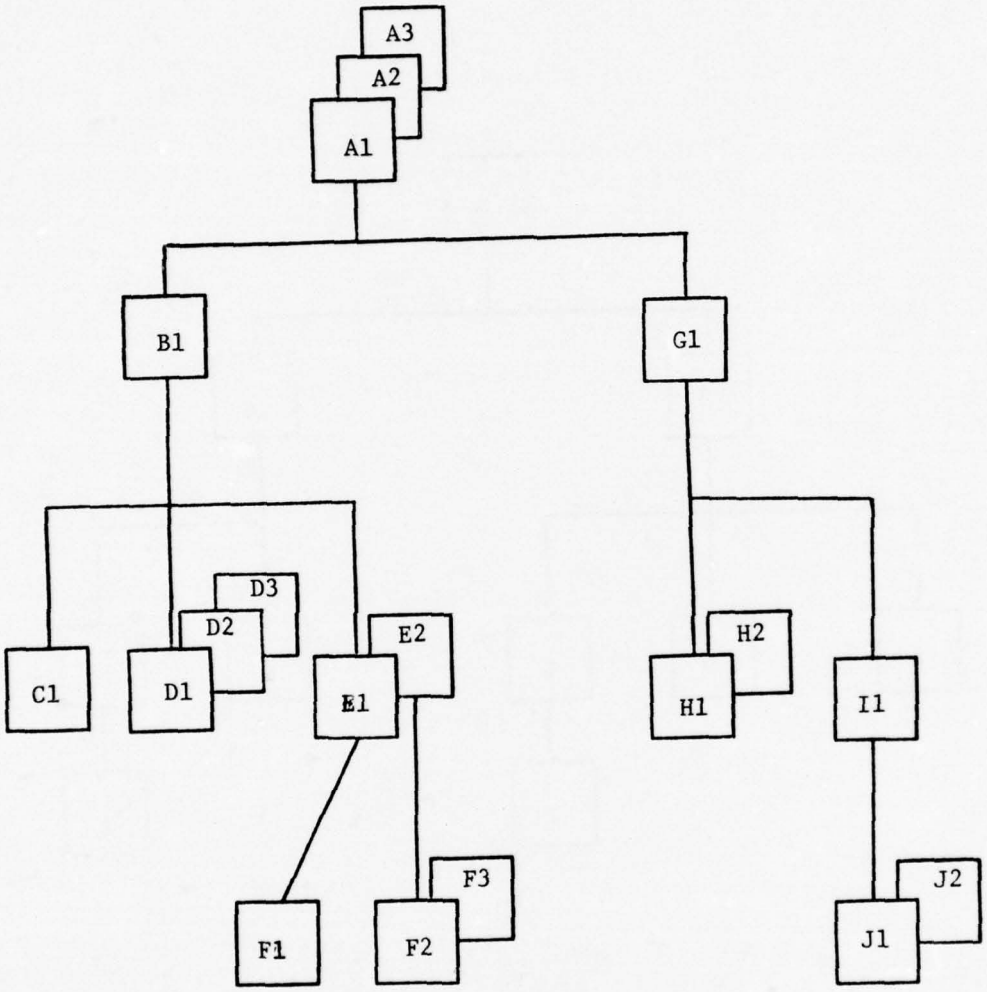


Figure 6. Schematic Representation of an IMS Database

Thus the following relationships exist.

A1 is the parent of B1 and G1.
H1, H2 and I1 are children of G1.
E1 and E2 as well as F2 and F3 are twins.

IMS application programs must traverse the segments of the database in order to make retrievals. The convention of traversing is from top to bottom (parent to child), front to back (among twins) and left to right (among children). More specifically, at every segment seek the leftmost segment in the next lower level (i.e., a child), if none exists, seek the next twin segment; if none exists, seek the next right segment at the same level; if none exists, seek the next right segment to the segment last traversed at the next higher level. The database in Figure 6 would be traversed in the order A1, B1, C1, D1, D2, D3, E1, F1, E2, F2, F3, G1, H1, H2, I1, J1, J2, A2, A3. Notice that the traversal order defines a next segment with respect to a given segment. This concept is used extensively in the data language DL/1. Finally, a hierarchical path is a sequence of segment occurrences, one per level, reading directly from a segment at one level to a particular segment at a lower level. For example A1, G1, I1, J2 is a hierarchical path.

3.2 The IMS Data Language DL/1

An IMS user processes an IMS database with application programs using Data Language/1 (DL/1). The DL/1 operations are invoked by means of subroutine calls from an application program. For clarity, we will not present the original DL/1 syntax in this discussion. Rather, a simplified form of the syntax is given which excludes the specification of the program communication block (PCB) and the I/O area in the call arguments. The specification of PCB and I/O area are mainly for the purpose of communication with the application program. A DL/1 call has the following format:

FUNCTION SEARCH-LIST

where FUNCTION is one of insert (ISRT), delete (DLET), replace (REPL) or a form of get (GET) and where SEARCH-LIST is a sequence of segment search arguments (SSA), possibly one per level which are used to select a hierarchical path.

3.2.1 The Search List

The basic function of the SEARCH-LIST is to narrow the field of search. It has the form

$$SSA_1 \quad SSA_2 \quad \ldots\ldots \quad SSA_n$$

where each segment search argument (SSA) is of the form

<segment-type> <Boolean expression>

with Boolean expression relating values of fields of the given segment type. The Boolean expression need not appear, in which case we say that the SSA is unqualified; otherwise it is qualified.

3.2.2 DL/1 Processing Functions

A brief description of the DL/1 processing functions of get, insert, delete, and replace is given in this section. The purpose of this section is to give the reader some idea of how each DL/1 function call can be interpreted. The rules for each of the function calls are omitted in the discussion. The discussion is informative rather than exhaustive. (For more detailed treatment of the subject, refer to [5]).

In the IMS context, the term "current position" in the database refers to a segment in the traversal sequence described earlier. After each retrieval or insertion operation, a position is established on the traversal sequence of the IMS database. For a retrieval operation, this position refers to the segment just retrieved; for an insertion operation, this position refers to the segment just inserted. Positions may also be established on the hierarchical path leading from the root segment to the "current position" in the database Each of these segments is called the "segment on which position is established at that level."

There are several forms of the get statement each of which returns a single segment. A get-unique (GU) call retrieves a specific segment by starting at the root segment type and finding the first segment at each level i satisfying SSA_i, retrieving the segment satisfying the last SSA. A get-next (GN) call starts the search at the "current position" in the database and proceeds along the traversal sequence satisfying the SSA's and retrieving the segment satisfying the last SSA. The basic difference between GN and GU calls is the starting position used in traversing the database. The execution of a GN call without SSAs returns the next segment (maybe of the some or different segment type) on the traversal sequence relative to the "current position" in the database. The execution of a GN call with an unqualified SSA returns the next segment on the traversal sequence of the segment type specified in the SSA, relative to the "current position" in the database.

It is also possible to restrict the number of segments to be searched using a get-next-within-parent (GNP) call. A GNP call restricts the search to descendants of a specific parent segment. Thus IMS also maintains a "parent position" which is set at the last segment that was retrieved by a GU or GN call. The parent position remains constant for successive GNP calls.

In order to lock (hold) a segment for future update there is a similar set of calls get-hold-unique (GHU), get-hold-next (GHN) and get-hold-next-within-parent (GHNP).

The get-hold calls are similar to their respective get calls but are used to obtain the segment before the contents of a segment can be changed through a DLET or REPL call. In this report, we will treat the get-hold calls as semantically equivalent to their respective get calls.

The ISRT call is used to initially load the segments for creation of a database and to add new occurrences of an existing segment type into an established database. The format of the ISRT call is identical for either use. When a segment occurrence is to be inserted, the parent occurrence must already exist in the database. The SSAs of the ISRT call specifies the complete hierarchical path from the root to this parent and also the type of the segment to be inserted. IMS will enter the new occurrence at the correct position as defined by the value of its sequence field.

The DLET call is used to delete the occurrence of a segment from a database. It deletes the specified segment occurrence and also all its children.

The REPL call is used to modify the content of a segment occurrence through program processing. The segment to be modified and replaced must first be obtained by a get-hold call. A REPL call can then be issued after the segment has been modified.

In order to simplify our discussion, we require that both GU and ISRT functions must have SSA for each level from the top level down. The GN and GNP functions may have either no SSA or have an SSA for each level starting at the top level. The DLET and REPL functions do not involve SSAs at all.

4. THE DBC REPRESENTATION OF AN IMS DATABASE

The first phase of the IMSI design is to propose a method to represent an IMS database in the DBC preserving the information from the IMS database. Recall that by preservation of information, we mean that any information in an IMS database that can be stored, retrieved, and manipulated by the DL/1 calls can also be stored, retrieved, and manipulated in the same manner by the same calls on the DBC database.

4.1 The Representation Problem

We intend to represent an IMS database by a file in the DBC in which an IMS segment is represented by a DBC record. Since a DBC record consists of keywords and attribute-value pairs, a natural way of representing a segment by a DBC record is to represent each field in the segment by either a keyword or a non-keyword attribute-value pair. We recall, at this point, that keywords are used by the DBC to perform content-addressing of the DBC records, whereas non-keywords are not. Normally the latter attribute-value pairs consist of the textual information of the record.

In order to determine whether a field should be represented by a keyword or by a non-keyword attribute-value pair, we must know whether or not such a field is to be used in a DL/1 search argument. If the field is to be used in a search argument, then it should be represented by a keyword. Otherwise, it should be represented by a non-keyword attribute-value pair. Hence, when an IMS database description is defined and stored in the database description library (DDL), it is necessary to define those fields which will be used as search arguments in a DL/1 call so that the IMSI can facilitate the representation.

In addition, information about the segment type and its relationship with other segments (i.e., parent-child and twin relationships) must also be represented as attribute-value pairs in the DBC record. For example, if the segment type is OFFERING, then the DBC record is augmented by the attribute-value pair (TYPE, OFFERING) where TYPE is the name of the attribute for segment type. In order to discuss the representation of the relationship among the segments, we first introduce some terminology. An identification field of a segment is a field of the segment whose value is distinct from all other values of the same field appearing in the twins of this segment (all root segments are considered as twins). If each segment has an identification field, one can simply "normalize" the hierarchical structure as follows. For each segment, embed the identification fields of each of its ancestors (i.e.,

parents, grandparents, and so on). In doing so, the parent-child and twin relationships are preserved even though the explicit hierarchical structure is removed. The only ambiguity that may occur is when the identification fields of different types of segments have the same field name. This can easily be resolved by qualifying the field name with the segment type. For simplicity, we assume distinct field names are used. The only complication is that we must know a priori what is an identification field for a segment. How then to select an identification field for each segment so that the hierarchical structure can be normalized? There are three cases to be considered:

First, if the segment has a sequence field* and its values are distinct among the twins, then the sequence field can be selected as the identification field of the segment.

Second, if the segment has a sequence field but its value is not distinct among the twins, then a new field is formed by augmenting the sequence field value so as to identify the twins uniquely. The augmented values can be generated, for instance, by the computer clock. The augmented sequence field is then designated as the identification field of the segment. It should be noted that the ordering of the segments defined by the augmented field is the same as that defined by the sequence field.

Finally, if the segment has no sequence field, then a completely new field is formed. A problem with this solution is that since no sequence field is explicitly defined, the ordering of the twins may be defined implicitly by their position in the IMS database. This ordering information would be lost if the new field is assigned an arbitrary value. Hence one must assign a value in such a way so that the order can be preserved. There are three positions where a segment with no sequence field can be inserted into the IMS database - as the first twin, as the last twin or between two twins. The method of assigning value to preserve the order of insertion is similar to the Dewey decimal notation (e.g., to insert a segment between 12.34 and 12.35, number it 12.34.1).

In the subsequent discussion, unless otherwise noted, we assume that each segment has a sequence field and its value is distinct among the twins of the segment type. It is evident that the identification field of a segment together with the identification fields of each of its ancestors (if any) uniquely identify the segment of a given type. In short, we will call this group of fields the symbolic identifier of the segment.

We now summarize the representation of an IMS database in the DBC. For

---

* The sequence field is a designated field for defining an ordering among the twins. This is an IMS terminology and convention.

each segment in an IMS database, the keywords and non-keyword attribute-value pairs of a DBC record are formed in one of the following four ways.

1. For each field in the segment which will be used in a DL/1 search argument, form a keyword using the field name as the attribute and the field value as the value.
2. Form a keyword of the form (TYPE, segtype) where TYPE is a literal and segtype is the segment type of the segment in consideration.
3. For each identification field in the symbolic identifier of the segment, form a keyword using the field name as the attribute and the field value as the value. Since a symbolic identifier may have one or more field name-value pairs, there will be one or more such keywords.
4. For each field which will not be used in a DL/1 search argument, form a non-keyword attribute value pair using the field name as the attribute and field value as the value.

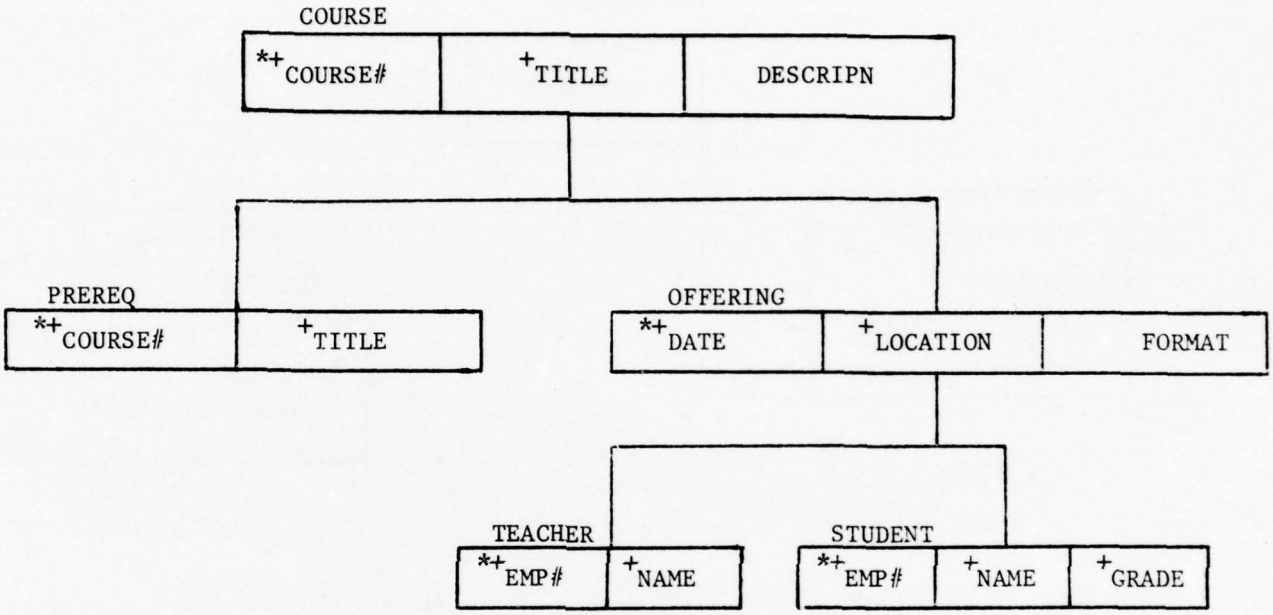
In the above process, many fields are represented by keywords. However, not all these keywords will be designated as Type-D keywords. In fact, as will be shown in Section 4.2, the number of Type-D keywords can be chosen to be relatively small without affecting DBC performance, if the appropriate clustering policy is chosen.

For example, consider the IMS database in Figure 7. We show its representation by different DBC records in Figure 8. For simplicity, no values are given; only the segment type and attribute templates are depicted. Even though the attribute TYPE appears in each DBC template, its values are different. Qualification is used in three occasions to resolve ambiguity because some of the field names used to form symbolic identifiers are not distinct. Notice that the COURSE# field in PREREQ and the EMP# fields in TEACHER and STUDENT need qualification.

4.2 The Choice of Type-D Keywords

Type-D keywords are keywords that are stored in the structure memory (SM) of the DBC. These keywords are update variant. By minimizing the number of Type-D keywords, we can reduce the SM storage required and the amount of SM update operations performed.

In certain cases, the reduction of the number of Type-D keywords will increase the number of MAUs to be searched. This can be shown by a simple example. Figure 9 shows 2 MAUs, MAU1 containing a DBC record R1 with keywords K1 and K2 and MAU2 containing a DBC record R2 with keywords K1 and K3. If K2 and K3 are chosen as Type-D keywords, then the directory entry for K2 will have the form <K2 : MAU1> meaning that all DBC records of the file containing K2 can be located in MAU1. Similarly, the directory entry for K3



Sequence field is marked with *

Search field is marked with +

Figure 7. The Logical Data Structure of an IMS database.

Symbolic identifier is underlined

Keyword is marked with @

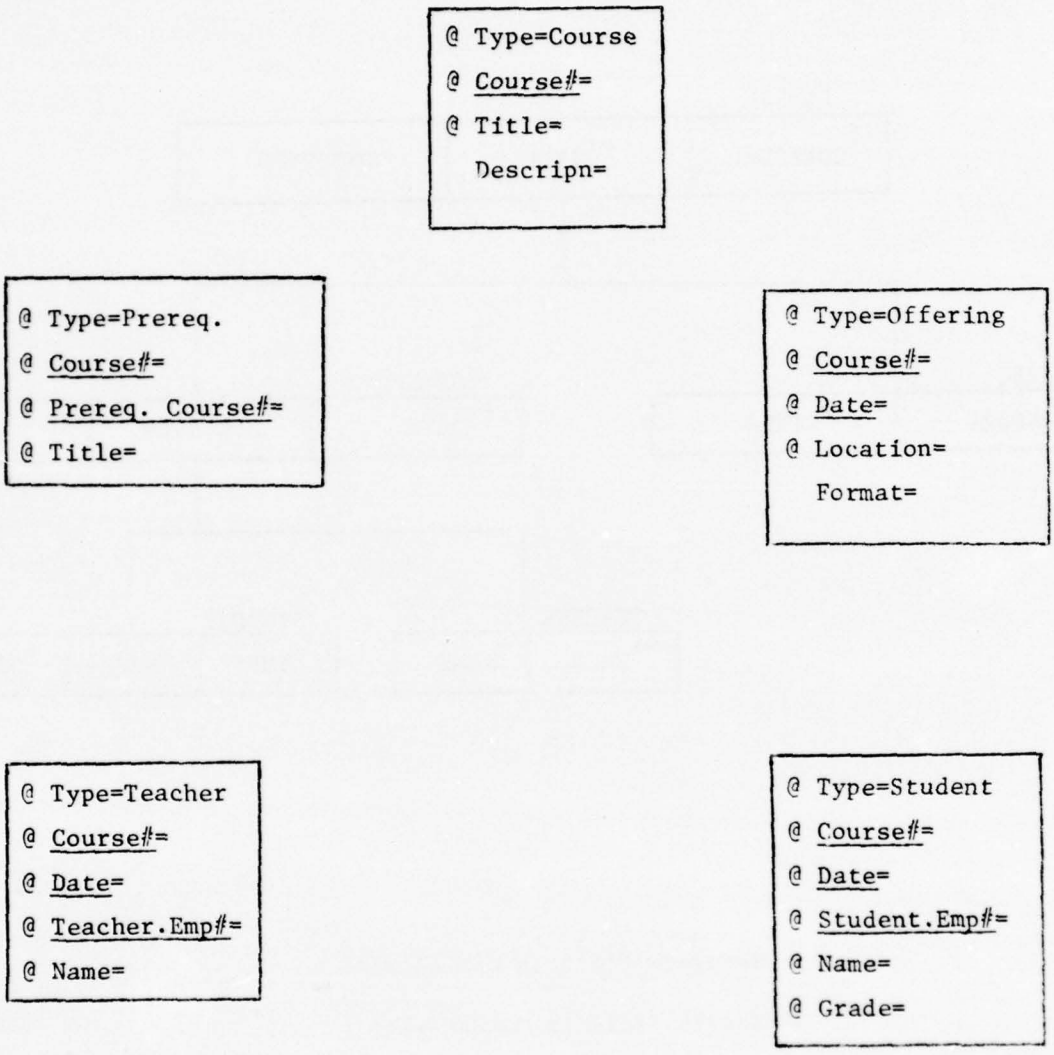


Figure 8. The attribute templates of DBC records for the transformed segments as shown in Figure 7.

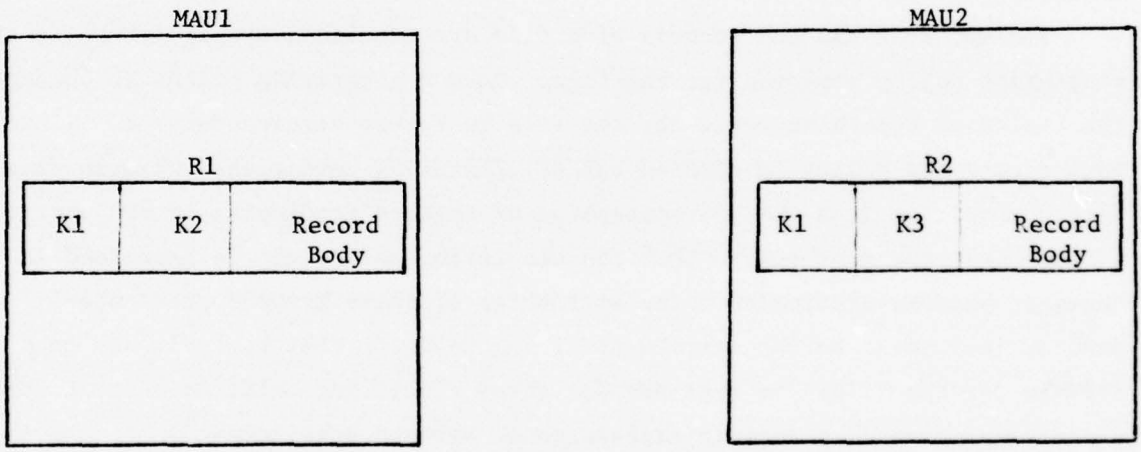


Figure 9. Two MAUs of the mass memory (MM).

is <K3 : MAU2>. To locate R1, K2 can be used as the query. Using the information provided by the SM, only MAU1 must be searched. Similarly, to locate R2, only MAU2 must be searched. However, if K1 is chosen (instead of K2 and K3) as the only Type-D keyword, then the directory entry for K1 is <K1 : MAU1, MAU2> reducing the number of directory entries in the SM. However, in order to retrieve either R1 or R2 by way of K1 both MAU1 and MAU2 must be searched. Hence the reduction of the number of Type-D keywords in this case will decrease the precision of the search and degrade the performance of the DBC.

On the other hand, if MAU1 contains both R1 and R2 as depicted in Figure 10, the choice of K1 as a Type-D keyword will still limit the search to MAU1. Hence the precision of the search is unaffected by a reduction of Type-D keywords. This example shows that if DBC records are clustered in an appropriate way, the number of Type-D keywords used can be reduced without affecting the DBC performance.

The way that the DBC records of a file are retrieved should determine the clustering policy employed for the file. Once a clustering policy is chosen, the choice of Type-D keywords for the file is rather straightforward. Since each clustering policy is carried out by clustering conditions and each clustering condition is a Boolean expression of keyword predicates in disjunctive normal form, one must ensure that the clustering policy can be expressed in terms of keyword predicates. The attributes of these keyword predicates should, therefore, be the attributes of the keywords that have already been defined for the file. We next propose three clustering policies each of which can be expressed in a Boolean expression of keyword predicates.

4.2.1. The First Clustering Policy.

The first policy (see Figure 11) clusters all DBC records which represent segments belonging to the same IMS root segment. This clustering policy is a natural one since segments are normally accessed by way of their root segments. If the DBC records representing an IMS root segment and all its dependent segments can be contained in a single MAU, then access to dependent segments of the root segment requires only one MAU access. A disadvantage of this policy is that if several root segments must be accessed collectively, several MAU accesses may be required since different root segments are not clustered in the same MAU. This situation may happen quite frequently in a GU call if the qualification statement given for the root segment type is satisfied by more than one root segment. The second disadvantage is that if the DBC records represent-

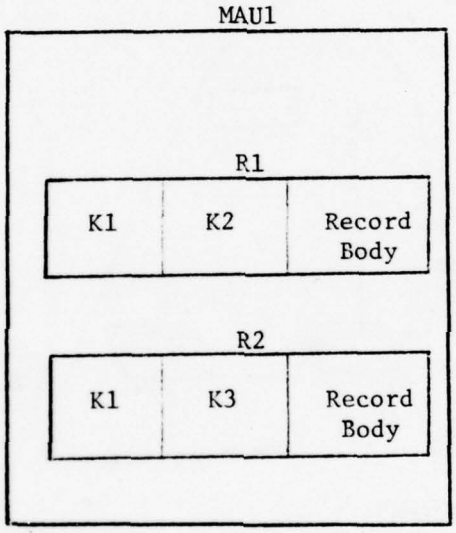


Figure 10. Two records clustered into one MAU.

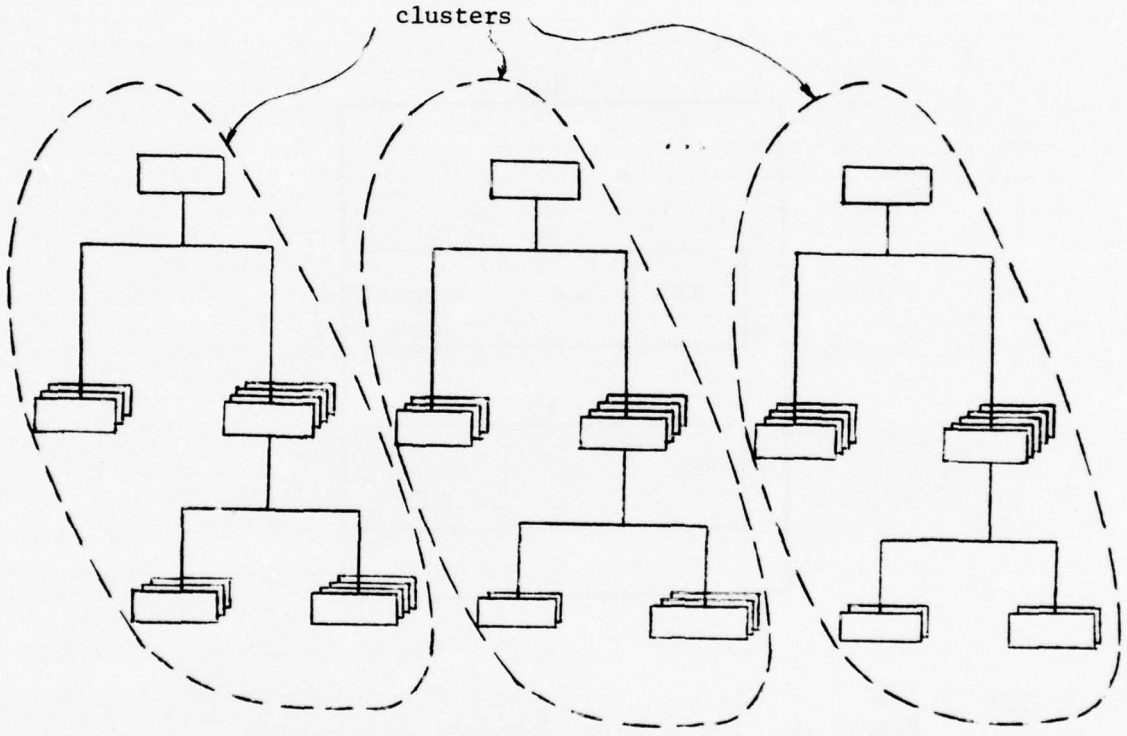


Figure 11. An Application of the First Clustering Policy

ing all the dependent segments of a root segment cannot be entirely contained in a single MAU, then access to dependent segments may require more than one MAU access.

4.2.2. The Second Clustering Policy

The second policy (see Figure 12) first clusters the DBC records which represent all the IMS root segments and then clusters the DBC records which represent all dependent segments. This policy takes care of the first disadvantage of the previous policy since all root segments now belong to the same cluster. The fact that a root segment is not clustered with its dependent segments does not affect the performance. This is because the root segment must be accessed before any dependent segment is accessed and hence the same number of DBC accesses are required to get to the dependent segments when either policy is used.

4.2.3. The Third Clustering Policy

The third policy (Figure 13) clusters the DBC records by segment type and is employed when the average size of the DBC records representing a root segment and all its dependent segments is larger than or comparable to the size of a MAU. This policy produces smaller clusters which can fit into MAUs.

A proposed clustering algorithm for hierarchical databases [8] used the known frequency of access pattern to produce an optimal clustering. However, that algorithm is not applicable to IMSI due to the way the DBC handles its records. In the system discussed in that paper, unit retrieved from the database is a page. When a parent segment, for example, must be found, the page containing that segment will be retrieved. If the child segment of the parent were stored in the same page, then subsequent access to these child segments will not require an additional access to the database if the page is still in main memory. Therefore, it is advantageous to cluster both the parent and its child segments in the same page.

The strategy of transferring data from the DBC database to the front-end computer is different. The notion of paging is absent from the DBC. Although it is possible to transfer the content of an entire MAU (the counterpart of a page) to the front-end computer, it is undesirable for the following reasons. First, a MAU is generally 2 orders of magnitude (i.e., 100 times) larger than a page. If the entire MAU contents must be accommodated, large amounts of buffer

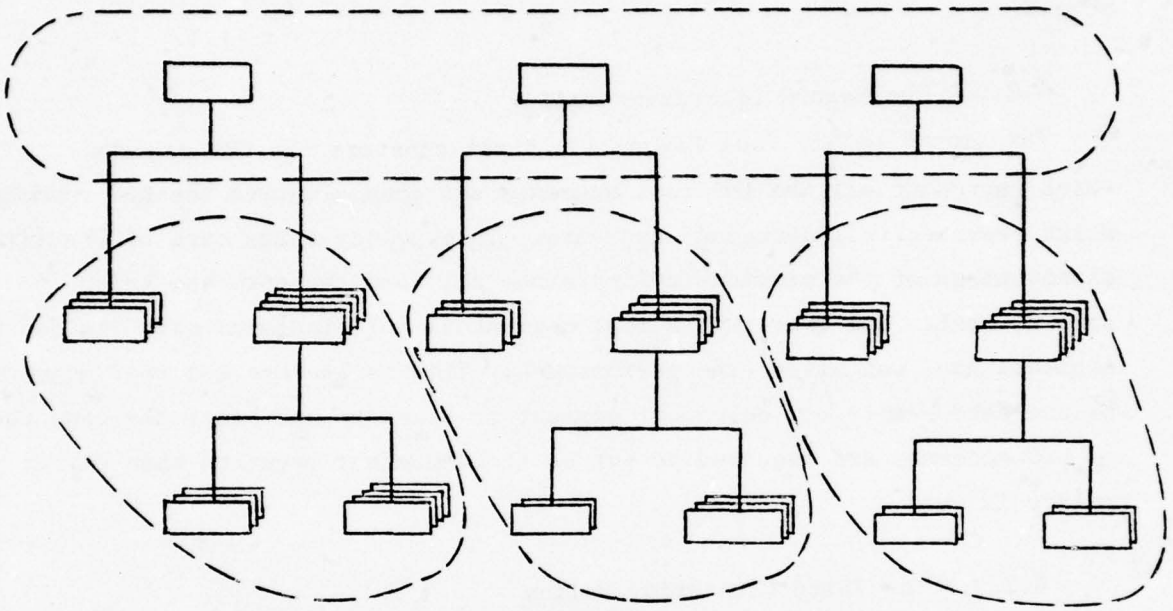


Figure 12. An Application of the Second Clustering Policy

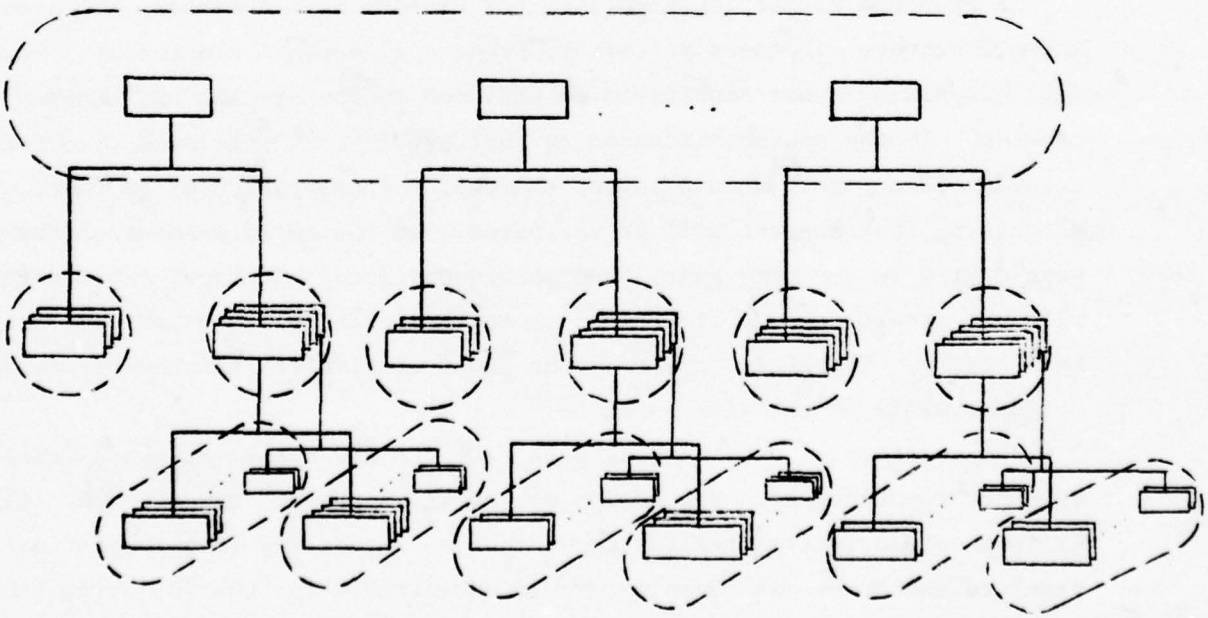


Figure 13. An Application of the Third Clustering Policy

storage must be provided. Second, the mass memory (MM) of the DBC is designed to search data stored in a MAU efficiently with its hardware content-addressing capability. If such capability is not utilized and the content is transferred to the front-end computer, then the IMSI will have to search the data thereby increasing the amount of IMSI software required. In addition, search performance would be degraded since this would be software performance not hardware performance.

Since two MAU accesses are needed to first fetch the parent segment and then its children, it is only necessary to cluster the children in one MAU. Clustering both the parent and its children in the same MAU will not improve the performance. Therefore, an elaborate clustering algorithm such as that proposed in [8] is not required.

In summary, the basic requirement of a clustering policy employed by the IMSI is that all the twin segments of a given type should be clustered. The reader should see that the first clustering policy discussed above violates this basic requirement, whereas it is satisfied by the second and third clustering policies. In fact, the clusters formed by the second and the third policies tend to be large, a desirable property which cuts down the number of clustering keywords used to define the clusters. Smaller clusters require more clustering keywords. However, one should not form clusters larger than one MAU. Thus in choosing a clustering policy which meets the basic requirement, one should minimize the number of clustering keywords and assure that the individual clusters are smaller than a MAU.

4.2.4 A Clustering Example

In this report, we choose the second clustering policy based on the assumption that the average size of an IMS database record (i.e., a root segment and all its dependent segments) is smaller than the size of an MAU (about 500K bytes). Once the clustering policy is chosen, we can determine the Type-D keywords for the file which will be the clustering keywords.

The clustering keywords chosen are those keywords corresponding to the sequence fields of the root segments and those keywords representing the segment types. In order to see that these are the only clustering keywords required, we present the MCCs for those clusters shown in Figure 12. The first MCC defines the cluster of root segments (Figure 14). There is then a separate MCC for each cluster of dependent segments of a given root segment. For example, Figure 15 shows the MCC for the dependent segments of the database record with root segment

MCC_1 : Type=Course

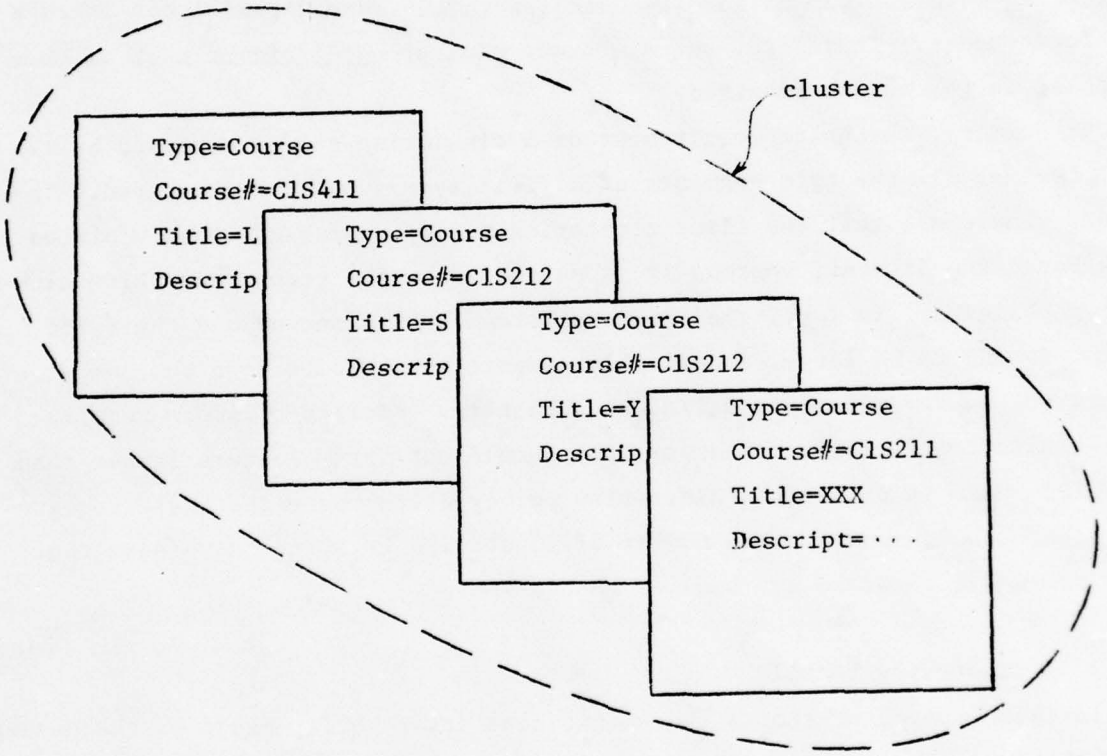
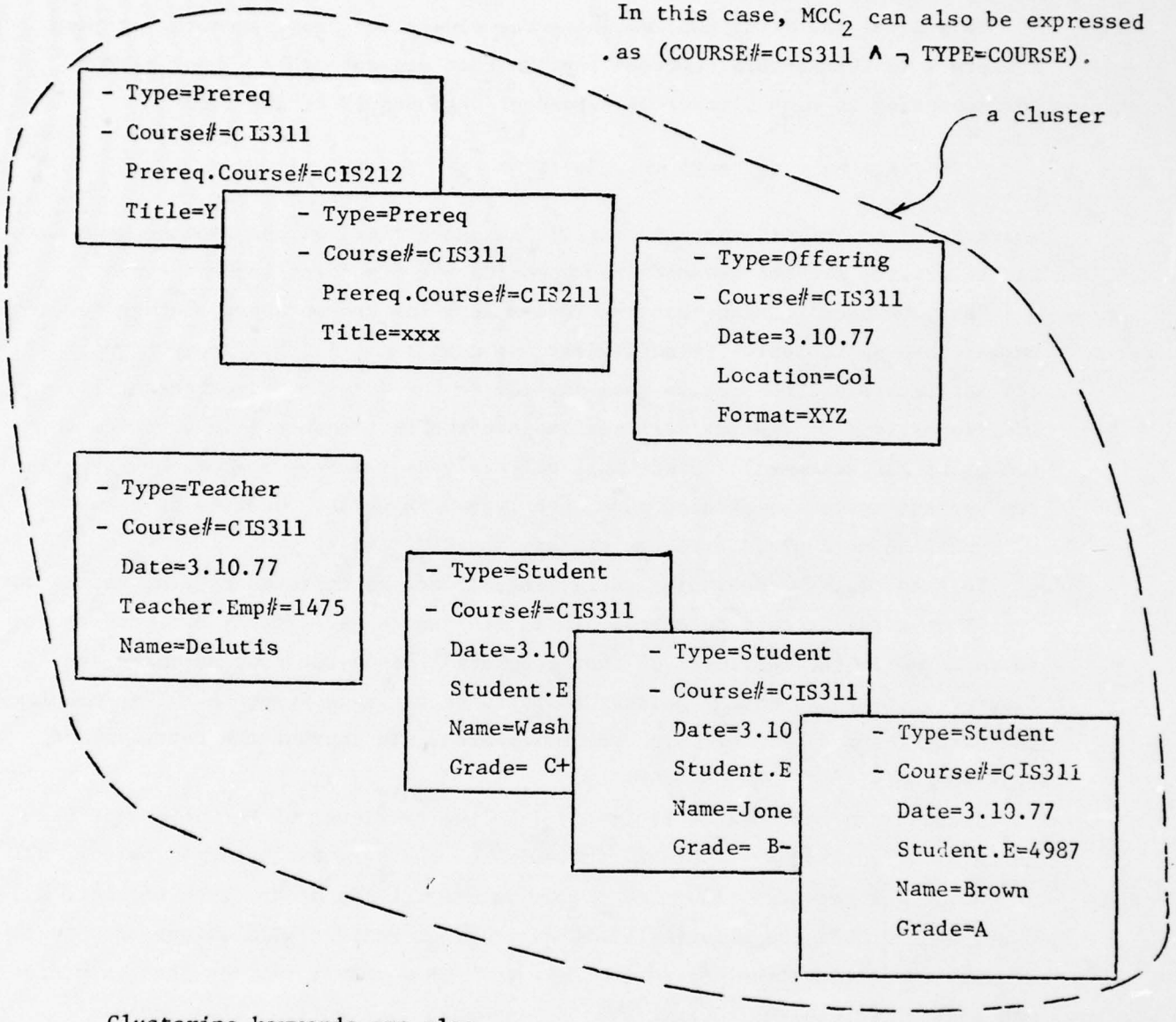


Figure 14. The Mandatory Clustering Condition (MCC) for the Root Segments.

MCC_2 = (Type=Prereq ∧ Course#=CIS311) ∨ (Type=Offering ∧ Course#=CIS311) ∨ (Type=Teacher ∧ Course#=CIS311) ∨ (Type=Student ∧ Course#=CIS311)

In this case, MCC_2 can also be expressed as (COURSE#=CIS311 ∧ ¬ TYPE=COURSE).



Clustering keywords are also marked with -.

Figure 15. The Mandatory Clustering Condition for the Dependent Segments of a Root Segment.

containing Course#=CIS311. In addition, Figure 16 shows the Type-D keywords (clustering keywords), the symbolic identifiers and the other keywords.

In general the first MCC, defining the cluster of root segments, is just K where K is the keyword representing the root segment type. Then the MCCs corresponding to each cluster of dependent segments is of the form

$$(K_1 \wedge \hat{K}) \vee (K_2 \wedge \hat{K}) \vee \ldots \vee (K_t \wedge \hat{K})$$

where $\hat{K}$ is the keyword representing the sequence field of the root segment and $K_1, K_2, \ldots, K_t$ are the keywords representing the dependent segment types.

Next, we need to show that the Type-D keywords chosen above, though few in number, are sufficient. By sufficient, we mean that (1) the Type-D keywords can facilitate all DL/1 calls when applied to the transformed database, (2) any additional Type-D keywords will not improve the performance (i.e., reduce the number of MAU accesses). Since DL/1 calls always retrieve segments by type and the segment types are already made into Type-D keywords. Queries in terms of Type-D keyword predicates can emulate the DL/1 calls.

To show that the searching performance cannot be improved by designating more Type-D keywords is more involved. There are two cases. First, can performance be improved in the retrieval of root segments? Since the root segments are clustered, it takes one MAU access to retrieve the root segments. Thus nothing, including the addition of more Type-D keywords, can improve the retrieval of root segments.

Second, can performance be improved in the retrieval of the dependent segments of a root segment? Since, according to the second clustering policy, all dependent segments are clustered by the sequence field of the root segment, we have to show that the sequence field of the root segment will always be used in a query to retrieve the dependent segments. This result implies that only one MAU access is required to retrieve the dependent segments so that again no improvement is possible. It remains to show that the sequence field of the root segment will always be used in a query to retrieve the dependent segments. Since the parent-child relationships are preserved by embedding the symbolic identifier of the parent into its child segments, whenever the child segments are to be retrieved, the symbolic identifier of the parent has to be used in the query. However, the sequence field of the root segment appears in every symbolic identifier of its dependent segment. Therefore, the sequence field of the root segment will always be used in a query to retrieve the dependent segments.

Symbolic identifier is underlined.

Keyword is marked with @.

Type-D keyword is marked with _ .

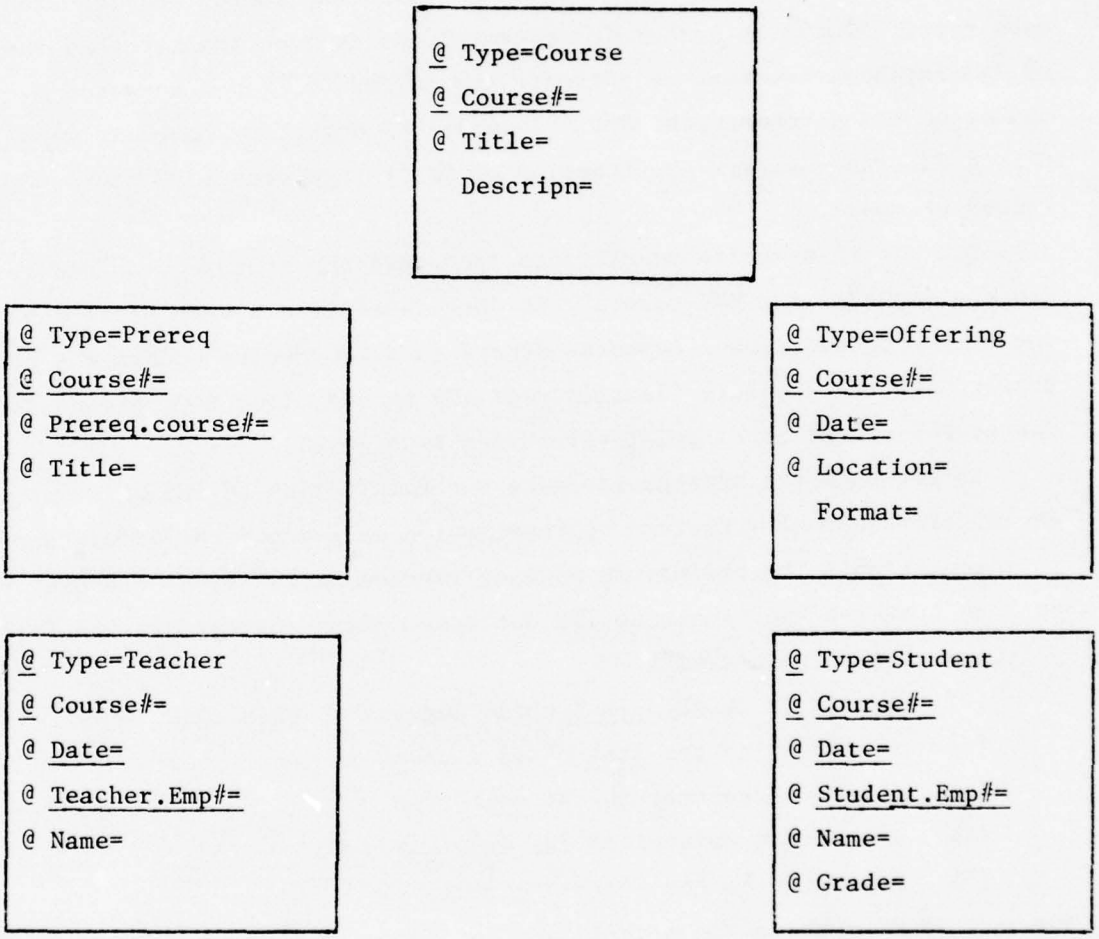


Figure 16. The attribute templates of DBC records showing Type-D Keywords.

In summary, we have proposed a way of choosing a sufficient set of Type-D keywords for a file. Even though the number of Type-D keywords chosen is small, the performance has not been compromised.

4.3. The Storage Requirement of the Structure Memory(SM) and Mass Memory(MM)

The amount of SM storage required depends on the number of Type-D keywords. Using the second clustering policy, there is one Type-D keyword for each root segment (i.e. for each IMS database record) and for each segment type. Since the number of segment types is much smaller than the number of IMS database records, we estimate the number of Type-D keywords by the number of IMS database records. Since an IMS database record is much larger than a directory entry, the fraction of SM storage versus MM storage will indeed be small.

The use of symbolic identifiers increases the storage required to store a segment as a DBC record. At each level of a hierarchical data structure, the number of additional keywords stored in a DBC record equals the number of keywords in the symbolic identifier of the parent, i.e. zero at the root level, one at the second level and (i-1) at the i-th level.

To estimate the storage saved by the elimination of IMS pointers, consider an IMS hierarchical structure representation called the child/twin pointer representation. The child/twin pointer representation gives the user the minimal path to traverse and update an IMS database. Each segment has the following pointers to its "relatives":

(1) A pointer to the first child segment of each type.

(2) A pointer to the last child segment of each type.

(3) A forward pointer to the next twin.

(4) A backward pointer to the previous twin.

(5) A pointer to the parent segment.

Hence, assuming there are m child segment types related to this segment, there are 2m + 3 pointers.

To compare the storage requirement for IMS pointers and DBC symbolic identifiers, the following terms need to be defined. The fanout of a segment type is the number of its child segment types. The depth of an IMS database is the maximum number of levels. For simplicity, we assume that the fanout of each segment type is a constant m and similarly that the number of twin occurrences of a child segment type (including the root segment type) is a constant y. It follows that every segment occurrence, except at the lowest level, has m*y child segment occurrences. Thus level 1 has y segments, level 2 has my^2 segments and in general level i has $m^{i-1}y^i$ segments. The number of pointers

and additional keywords used at the i-th level will be $m^{i-1}y^i(2m+3)$ and $m^{i-1}y^i(i-1)$, respectively. Hence the total number of pointers N_p is given by:

$$N_p = \sum_{i=1}^{n} m^{i-1}y^i(2m+3)$$

and the total number of additional keywords N_k is given by:

$$N_k = \sum_{i=1}^{n} m^{i-1}y^i(i-1).$$

We then define the storage ratio R as:

$$R = \frac{N_k}{N_p} x\ k \qquad (1)$$

where k is the ratio of the average length of a sequence field to the length of a pointer. When $R < 1$, more space is required for the pointers than for the additional keywords.

By using some algebraic manipulation and the following formulas:

$$S_p(x) = \sum_{i=0}^{p} x^i = \frac{1-x^{p+1}}{1-x}$$

$$T_p(x) = \sum_{i=1}^{p} ix^i = \frac{S_p(x) + x-(1+px^{p+1})}{1-x}$$

it can be shown that for $x=my$

$$R = \frac{k}{(2m+3)}\left\{\frac{1}{1-my} - \frac{my-1}{(my)^n-1} + \frac{n-1}{1-1/(mn)^n}\right\}$$

Assuming $1/(mn)^n \doteq 0$, R can be approximated by

$$R \doteq \frac{k}{(2m+3)}\left\{\frac{1}{1-my} + (n-1)\right\} \qquad (2)$$

Further assuming my is sufficiently large, then (2) can be simplified to

$$R \doteq k\frac{n-1}{2m+3} \qquad (3)$$

According to (3), the storage ratio R between symbolic identifier and pointers decreases when either the depth n decreases or the fanout m increases.
In Tables II, III, IV, and V, we show different values of R for various n, m, and k. Since IMS uses 4 byte pointers and k may have values of 1, 2, 4, or 8, the length of an average sequence field is 4, 8, 16, or 32 bytes, respectively.

n \ m	1	2	3	4	5	6	7	8	9	10
1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
2	0.20	0.14	0.11	0.09	0.08	0.07	0.06	0.05	0.05	0.04
3	0.40	0.28	0.22	0.18	0.15	0.13	0.12	0.11	0.10	0.09
4	0.60	0.43	0.33	0.27	0.23	0.20	0.18	0.16	0.14	0.13
5	0.80	0.57	0.44	0.36	0.31	0.27	0.24	0.21	0.19	0.17

Table II. R as a function of n and m where k=1

n \ m	1	2	3	4	5	6	7	8	9	10
1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
2	0.40	0.28	0.22	0.18	0.15	0.13	0.12	0.11	0.10	0.09
3	0.80	0.57	0.44	0.36	0.31	0.27	0.24	0.21	0.19	0.17
4	1.20	0.86	0.67	0.54	0.46	0.40	0.35	0.32	0.29	0.26
5	1.60	1.14	0.89	0.73	0.62	0.53	0.47	0.42	0.38	0.35

Table III. R as a function of n and m where k=2.

n \ m	1	2	3	4	5	6	7	8	9	10
1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
2	0.79	0.57	0.44	0.36	0.31	0.27	0.23	0.21	0.19	0.17
3	1.59	1.14	0.89	0.73	0.61	0.53	0.47	0.42	0.38	0.35
4	2.39	1.71	1.33	1.09	0.92	0.80	0.71	0.63	0.57	0.52
5	3.19	2.28	1.78	1.45	**1.23**	**1.07**	0.94	0.84	0.76	0.70

Table IV. R as a function of n and m where k=4.

n \ m	1	2	3	4	5	6	7	8	9	10
1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
2	1.58	1.14	0.89	0.73	0.61	0.53	0.47	0.42	0.38	0.35
3	3.18	2.28	1.77	1.45	1.23	1.07	0.94	0.84	0.76	0.70
4	4.78	3.42	2.66	2.18	1.84	1.60	1.41	1.26	1.14	1.04
5	6.38	4.57	3.55	2.91	2.46	2.13	1.88	1.68	1.52	1.39

Table V. R as a function of n and m where k=8.

In summary, the storage requirement for the symbolic identifiers is compensated for in many cases by the removal of the conventional address pointers. It also suggests that the design of a hierarchical database with smaller number of levels and larger fanout will result in storage conservation when it is stored in the DBC. Furthermore, the use of symbolic identifiers eliminates the needs of creating secondary indexes when the user wishes to enter the database through the dependent segments, contributing a further reduction in storage.

5. THE TRANSLATION PROCESS

The DL/1 interface module (IM) is the component of the IMSI which interfaces with an IMS user program by translating and executing DL/1 calls. The IM is designed to fully utilize the DBC capabilities so that the IMSI need not perform any content-addressing, resulting in a reduction of the software cost for implementing and running the IMSI.

The information obtained in the course of executing a DL/1 call is maintained in the interface system buffer (ISB) and consists of DBC records which are retrieved from the DBC in the execution of the DL/1 call. Since each DBC record represents a segment in an IMS database, we will use the terms DBC record and segment interchangeably whenever there is no confusion. We shall call all the meaningful information in the ISB at any given time the content of the ISB.

In the following example, we use the IMS database which was defined in Figure 7. Suppose the DL/1 call to be processed is:

```
GU      COURSE    (TITLE='MATH')
        OFFERING  (LOCATION='STOCKHOLM')
        STUDENT   (GRADE='A')
```

This get-unique (GU) call has three segment search arguments (SSAs), COURSE (TITLE='MATH'), OFFERING (LOCATION='STOCKHOLM'), and STUDENT (GRADE='A'). It retrieves a specific segment by starting at the COURSE segment type and finding the first segment satisfying the SSA at each level, and then retrieving the segment satisfying the last (i.e., the third) SSA. The content of the ISB after the execution of the above call is shown in Figure 17. For clarity, we do not show how the information is managed in the ISB (see Section 6). At this point we assume that the ISB has unlimited memory. It contains segments of three types, ordered according to the values of their sequence fields. The segments at different levels marked as 'x', 'y', or 'z' are the segments on which position is established at that level (see Section 3.2) and are referred to as the current segment of the given type. The STUDENT segment marked 'z' is also established as the current position in the database and is sent to the user I/O area.

Now we shall proceed to illustrate how the content of the ISB is established during the execution of the above GU call.

(1) Starting with the first SSA in the call, i.e., COURSE (TITLE='MATH'), the COURSE segments which satisfy the qualification TITLE='MATH' are retrieved from the DBC and put into the ISB. These segments are retrieved by the DBC query (TYPE=COURSE $\wedge$ TITLE='MATH') and are sorted by the DBC according to the values of their sequence fields.

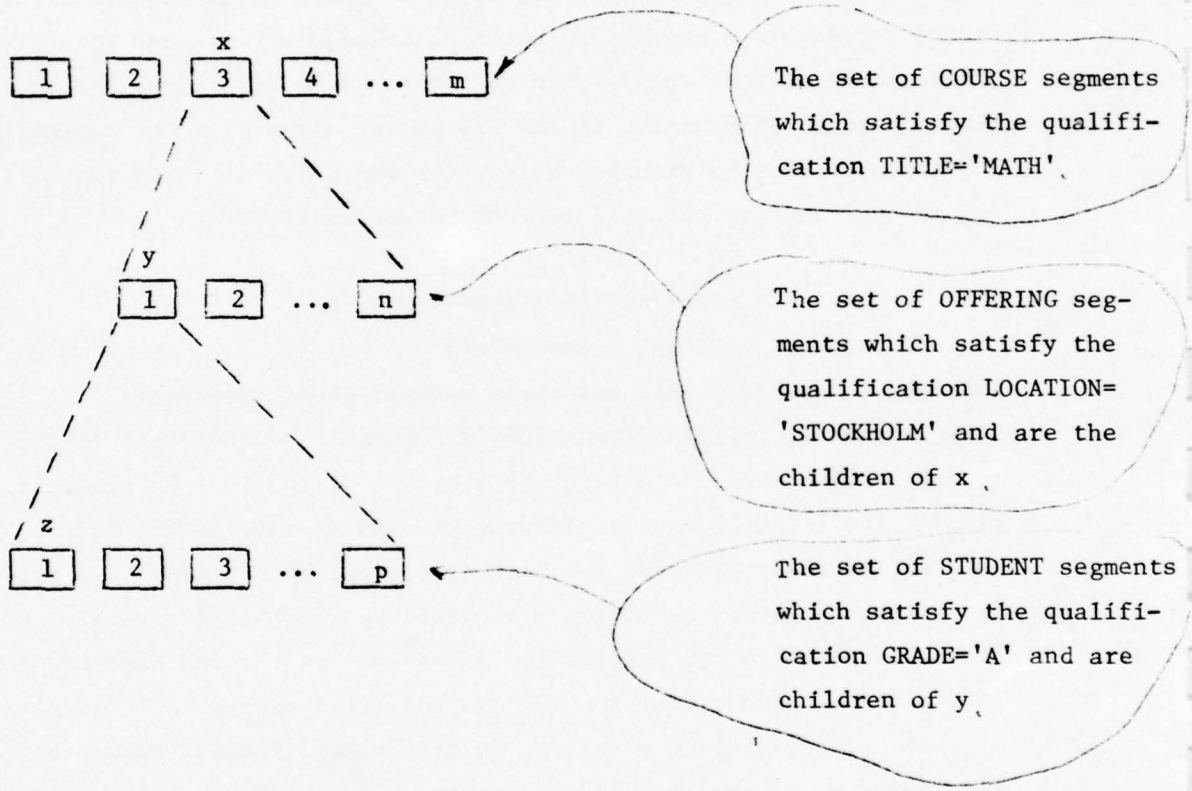


Figure 17. The content of the ISB.

(2) The first COURSE segment in the ISB is taken as the current COURSE segment.

(3) The OFFERING segments, children of the current COURSE segment, are then retrieved with the qualification (LOCATION='STOCKHOLM') and stored in the ISB in the order defined by their sequence field. If the symbolic identifier of the current COURSE segment were (COURSE#=M5), then the DBC query created for this retrieval would be(TYPE=OFFERING ∧ COURSE#= M5 ∧ LOCATION='STOCKHOLM').

(4) If no segment can be retrieved in step 3, then the next COURSE segment in the ISB is established as the current COURSE segment and step 3 is repeated.

(5) Suppose some OFFERING segments are retrieved and stored in the ISB, the first OFFERING segment in the ISB is taken as the current OFFERING segment.

(6) A process similar to step 3 and step 4 is performed to retrieve the first segment with qualification (GRADE='A').

(7) The first of the retrieved STUDENT segments is sent to the user I/O area.

It should be noted at this point that the content of the ISB established by the above GU call could be used to process the next DL/1 call, for example to retrieve the next student who has an A grade in a MATH course offered in Stockholm. In this case the next student segment (with respect to the current STUDENT segment) in the ISB is transferred to the user I/O area without accessing the DBC. In order to process the next DL/1 call, it is essential for the IM to know the content of the ISB. Thus a status information table is established.

The example discussed above has only one hierarchical path at a given time. (i.e., a COURSE segment with TITLE='MATH', its child OFFERING segment with LOCATION='STOCKHOLM' and its child STUDENT segment with GRADE='A'). IMS also allows multiple hierarchical paths as long as all segment occurrences at the same level have the same parent segment. Thus a second hierarchical path to a TEACHER segment is allowed. These hierarchical paths define a current segment for each segment type.

5.1 The Status Information Table (SIT)

The IMSI maintains information about the content of the ISB in the status information table. Making use of the SIT, the IM can determine what changes, if any, have to be made to the content of the ISB in order to execute a DL/1 call. The information stored in the SIT is not used in the execution of a GU call since each GU call executes independently of the previous contents of the ISB. However, the SIT is a vital tool used to execute the GN or GNP call since the result of such a call depends on the previous content of the ISB. For each segment type the SIT contains a COUNT of the number of occurrences of that segment type in the ISB, a number between 1 and COUNT indicating the CURRENT SEGMENT of that segment type and a Boolean expression (QUALIFICATION) satisfied by all the segments of that type in the ISB.

The CURRENT SEGMENT field is used to facilitate sequential traversal (i.e., the retrieval of the "next" segment). The COUNT field is used to determine whether or not the current segment is the last segment in the ISB. The QUALIFICATION field is used to tell whether or not the content of the ISB can be used to satisfy the DL/1 call.

5.2 The Translation of the Get Calls

The various DL/1 calls have been discussed in Section 3. Since the get calls are the most sophisticated calls in DL/1, we first discuss how a get call is executed by the IM. The execution of the delete, replace and insert calls is straightforward and will be discussed later in Appendix A.

5.2.1 An Observation

It can be shown that GU and GN can be viewed as cases of GNP. Therefore any GU, GN or GNP call can be treated as a GNP call. This observation allows us to use basically the same algorithms to handle all the get calls. To illustrate this point consider the IMS database shown in Figure 18 having three levels and three root segments. we now introduce a fictitious level (called level zero), as shown in Figure 19. Each root segment becomes a child segment of the segment (the zero segment) introduced at level zero. We will refer to the IMS database given in Figure 18 and Figure 19 as DB1 and DB2 respectively. Notice that DB2 has only one database record.

We now show that any GU, GN and GNP call given to DB1 can be treated as a GNP call given to DB2 if the current position and parent position in DB2 are initialized appropriately.

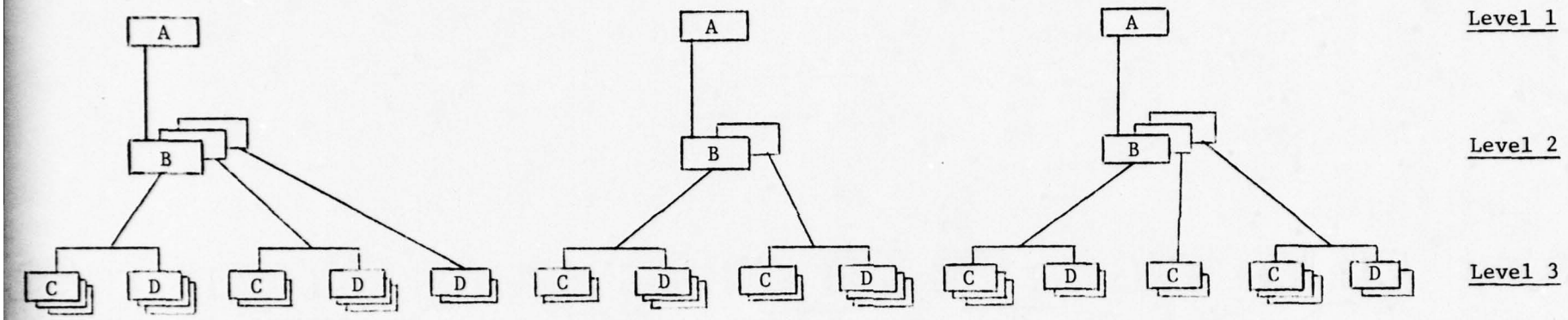


Figure 18. An IMS database with 3 database records.

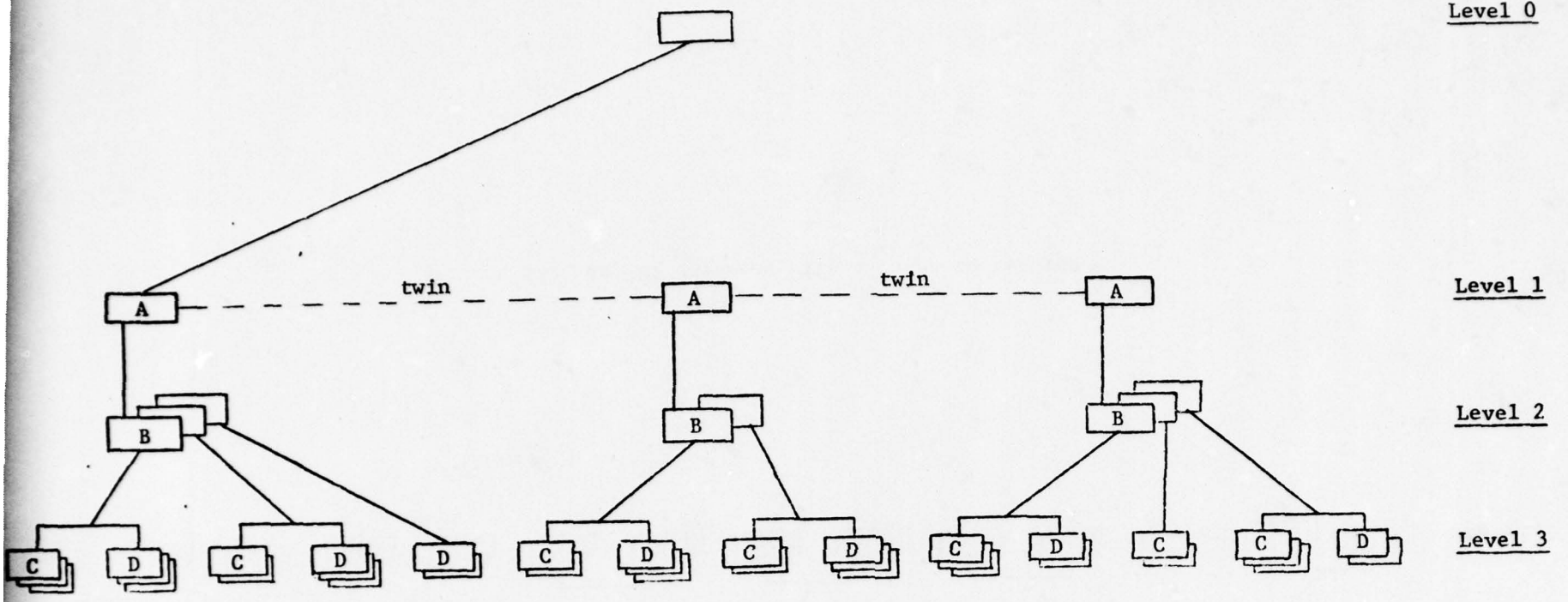


Figure 19. The IMS database transformed by the introduction of a universal parent.

First, to see that a GN call to DB1 is equivalent to a GNP call to DB2, we set the parent position on the zero segment in DB2 and set the current position in DB2 to that in DB1. The effect of executing a GN call on DB1 is the same as executing a GNP call with the same search arguments on DB2.

Second, to see that a GU call to DB1 is equivalent to a GNP call to DB2, we set the current position and parent position in DB2 to the zero segment and execute the GNP call with the same segment search arguments as those in the GU call.

Finally, a GNP call to DB1 is equivalent to a GNP call to DB2 as long as the current position and the parent position are the same.

5.2.2 Examples

These examples are based on the IMS database defined in Figure 7.

Example 1: To process the call:

```
GU      COURSE
        OFFERING (LOCATION='STOCKHOLM')
```

A get-unique call must find the first occurrence of a segment satisfying the SSAs. Thus no information in the ISB is of use. Therefore, the SIT must be initialized so that processing may begin at level 1 (Figure 20). The DBC query for retrieval

```
(TYPE = COURSE)
```

is created to load the ISB with all COURSE segments. Assuming there were, for example, a total of 15 such segments then the ISB should be changed to that shown in Figure 21. The next step is to retrieve OFFERING segments which are children of the current COURSE segment and which satisfy the qualification LOCATION='STOCKHOLM'. Suppose the current COURSE segment has the symbolic identifier K. The DBC query created for retrieval is:

```
(TYPE=OFFERING)∧ K ∧(LOCATION='STOCKHOLM')
```

Suppose that 4 OFFERING segments are retrieved from the DBC and stored in the ISB. The changes in the SIT are given in Figure 22. The processing is completed and the first OFFERING segment is transferred to the user I/O area.

Example 2: To process the get-next call:

```
GN      COURSE
        OFFERING    (LOCATION='STOCKHOLM')
        STUDENT     (GRADE='A')
```

	CURRENT SEGMENT	COUNT	QUALIFICATION
0	1	1	NULL
(course) 1			
(prereq) 2			
(offering) 3			
(teacher) 4			
(student) 5			

Figure 20. Initialized SIT.

	CURRENT SEGMENT	COUNT	QUALIFICATION
0	1	1	NULL
(course) 1	1	15	NULL
(prereq) 2			
(offering) 3			
(teacher) 4			
(student) 5			

Figure 21. SIT after retrieval of COURSE segments.

	CURRENT SEGMENT	COUNT	QUALIFICATION
0	1	1	NULL
(course) 1	1	15	NULL
(prereq) 2			
(offering) 3	1	4	LOCATION='STOCKHOLM'
(teacher) 4			
(student) 5			

Figure 22. SIT after retrieval of OFFERING segments.

assuming the previous call is the one given in Example 1. Since the SSAs for COURSE and OFFERING are identical, processing begins at level 3 by bringing into the ISB the STUDENT segments which are the children of the current OFFERING segment satisfying the qualification GRADE='A'. Suppose the symbolic identifier of the current OFFERING segment is composed of keywords K_1, K_2, ... K_j. The DBC query created for retrieval is:

$$(\text{TYPE=STUDENT}) \wedge K_1 \wedge K_2 \wedge \ldots \wedge K_j \wedge (\text{GRADE='A'})$$

Suppose that 5 STUDENT segments satisfy the query. The changes in the SIT is shown in Figure 23. The processing is completed and the first STUDENT segment is transferred to the user I/O area.

Now suppose that the OFFERING occurrence has no STUDENT child segments with Grade='A'. Then it is necessary to examine the next OFFERING occurrence, which is known to be in the ISB since the OFFERING SSAs for this and the previous queries are the same. Thus the STUDENT segments which are children of the next OFFERING must be retrieved.

Example 3: To process the get-next call:

```
GN      COURSE
        OFFERING
        STUDENT     (GRADE='B')
```

assuming the previous call as given in Example 2. Since the SSA for STUDENT is GRADE='B' whereas the previous was GRADE='A', the DBC must be asked to retrieve new STUDENT segments which are children of the current OFFERING but which follow the current STUDENT. Thus suppose that the symbolic identifier of the current OFFERING segment is composed of the keywords K_1, K_2,..., K_j and the sequence field of the current STUDENT segment is SEQFLD= x. Then the DBC query created for retrieval is:

$$(\text{TYPE=STUDENT}) \wedge K_1 \wedge K_2 \wedge \ldots \wedge K_j \wedge (\text{SEQFLD} \geq x) \wedge (\text{GRADE='B'})$$

Suppose that 8 STUDENT segments are retrieved and stored into the ISB. The changes in the SIT is shown in Figure 24. The processing is completed and the first STUDENT segment is transferred to the user I/O area.

Finally suppose that the current OFFERING occurrence has no STUDENT child segments with GRADE='A'. As in the similar situation in Example 2, it is then necessary to examine the next (in the traversal sequence) OFFERING occurrence. However the new SSA for OFFERING is NULL, while the old was

	CURRENT SEGMENT	COUNT	QUALIFICATION
0	1	1	NULL
(course) 1	1	15	NULL
(prereq) 2			
(offering) 3	1	4	LOCATION='STOCKHOLM'
(teacher) 4			
(student) 5	1	5	GRADE='A'

Figure 23. SIT after retrieval of 5 STUDENT segments.

	CURRENT SEGMENT	COUNT	QUALIFICATION
0	1	1	NULL
(course) 1	1	15	NULL
(prereq) 2			
(offering) 3	1	4	LOCATION='STOCKHOLM'
(teacher) 4			
(student) 5	1	8	GRADE='B'

Figure 24. SIT after retrieval of 8 STUDENT segments with GRADE='B'.

LOCATION='STOCKHOLM'. Thus there is no guarantee that the next occurrence is in the ISB so that the DBC must be asked to retrieve all the OFFERING segments that are children of the current COURSE segment and have sequence fields greater than the current OFFERING.

5.2.3 A Translation Strategy

The processing of a get-call will now be described briefly. The complete algorithms are described in Appendix A.

The status information table (SIT) is used to process any get-calls. Before any calls are processed the SIT must be initialized to indicate that no segments of any type are in the interface system buffer (ISB) except the zero segment (whose presence is fictitious). After a get-call has been processed the SIT contains information about the segment occurrences in the ISB. Recall that IMS identifies a current occurrence of one or more segment types. One of the properties of these occurrences is that all occurrences of different segment types at the same level must be children of the same segment occurrence at the next higher level.

Suppose that a get-call of the following form is received by the IM:

$$\begin{array}{ll} \text{Get} & (S_1, Q_1) \\ & (S_2, Q_2) \\ & \vdots \\ & (S_n, Q_n) \end{array}$$

where S_i is the segment type and Q_i is the corresponding qualification at level i. This call is first transformed by taking Q_0=NULL and adding the pair (S_0,Q_0) representing the zero segment to obtain

$$\begin{array}{ll} \text{Get} & (S_0, Q_0) \\ & (S_1, Q_1) \\ & \vdots \\ & (S_n, Q_n). \end{array}$$

The problem is then to determine which of the current segment occurrences satisfy the new SSAs at each level and then whether the ISB contains all the segment occurrences which might satisfy the new SSA by comparing the Q_i with the corresponding QUALIFICATION in the SIT.

Assuming that the current occurrences of segment types S_0, S_1, ...,S_i

satisfy the corresponding qualifications Q_0, Q_1, Q_2,...,Q_i, then the following table shows what can be determined about $(S_{i+1}, Q_{i+1}) = (S, Q)$ by information in the SIT. In that table the notation "Q implies QUALIFICATION" means the Boolean expression Q implies the Boolean expression QUALIFICATION which is an entry in SIT, so that a segment occurrence satisfying Q must also satisfy QUALIFICATION. Thus, since the ISB contains all the segments satisfying Q. However, if QUALIFICATION implies Q but they are not identical, there may be segments satisfying Q which are not in the ISB and which may need to be retrieved from the DBC.

The previous discussion suggests the algorithm RETRIEVE (I, SUCCESS), shown in Figure 25, which retrieves segments satisfying Q_i, Q_{i+1}, ... , Q_n that are descendents of segments satisfying Q_0, Q_1, ... , Q_{i-1}. SUCCESS is set to TRUE if the segments are found, otherwise it is set to FALSE. Retrieval of segments satisfying Q_0, Q_1, ... , Q_n is accomplished by calling RETRIEVE (0, SUCCESS). RETRIEVE uses two other procedures, BUFFER (I, SUC) and NEXT (I, SUC). BUFFER (I, SUC) retrieves segments of type S_I satisfying Q_I from the DBC and places them into the ISB. SUC is set to TRUE if a non-empty set is retrieved; otherwise, to FALSE. NEXT (I, SUC) advances the current segment of type S_I and sets SUC to FALSE if there are no more segments, and to TRUE, otherwise. Unless Q_I = $QUALIFICATION_I$ (i.e., the desired segment is in the ISB), the next occurrence of the segment type S_I is retrieved from the DBC and placed in the ISB. There is no attempt to determine if Q_I implies $QUALIFICATION_I$ (i.e., the desired segment may already be in the ISB).

	Current occurrence satisfies Q	All occurrences that satisfy Q are in the ISB
1. no current segment of type S	no	no
2. current segment of type S and Q=QUALIFICATION	yes	yes
3. current segment of type S and Q≠QUALIFICATION and (QUALIFICATION=NULL or Q implies QUALIFICATION)	maybe	yes
4. current segment of type S and Q≠QUALIFICATION and (Q=NULL or QUALIFICATION implies Q)	yes	not necessarily
5. otherwise	maybe	not necessarily

```
RETRIEVE (I, SUCCESS)

     /*Retrieves segments satisfying Q_I, Q_I+1, ... , Q_N that are descendents*/
     /*of segments satisfying Q_0, Q_1, ... , Q_I-1.  Set SUCCESS=TRUE if seg- */
     /*ments are found, otherwise FALSE.                                       */
     if no current segment of type S_I or current segment of type S_I does not
        satisfy Q_I
        then do;
           call BUFFER (I, SUC);
           if ¬ SUC then SUCCESS=FALSE, return;
        end;
     if I=N
        then do;
           call NEXT (I, SUC);
           if ¬ SUC then SUCCESS=FALSE, return;
        end;
        else do;
           RETRIEVE (I+1, SUCCESS);
           do while ¬ SUCCESS;
               call NEXT (I, SUC)
               if ¬ SUC then SUCCESS=FALSE, return;
               RETRIEVE (I+1, SUCCESS);
           end;
        end;
     end RETRIEVE;


BUFFER (I, SUC)

     /*Retrieve segments of type S_I satisfying Q_I from DBC and place into ISB.*/
     if no current segment of type S_I in ISB
        then do;
           retrieve from DBC segments of type S_I which satisfy Q_I into ISB;
        else do;
           retrieve from DBC segments of type S_I which satisfy Q_I and their
           sequence field values are greater that that of the current segment
           of type S_I;
     if no segment can be retrieved
        then SUC=FALSE;
        else SUC=TRUE;
     QUALIFICATION_I=Q_I;
     end BUFFER;


NEXT (I, SUC)
     if Q_I≠QUALIFICATION_I
        then do;
           call BUFFER (I, SUC)
           if ¬ SUC then return;
        end;
     if the current segment of type S_I is the last segment in the ISB
        then SUC=FALSE, return;
        else do;
           advance current segment of type S_I;
           SUC=TRUE
           return;
        end;
     end NEXT;
```

Figure 25. The algorithms RETRIEVE BUFFER, AND NEXT.

6. BUFFER MANAGEMENT

The interface system buffer (ISB) is created and managed in the IMSI to reduce the number of accesses to the DBC. It is designed to be implemented in a virtual memory system environment as shown in Figure 26. More specifically the ISB resides in the virtual memory space of the computer system which supports the IMSI, whereas the buffer partition is the partition of main memory allocated to the ISB for paging. Since virtual memory management is conventional, we will not address it here.

Before introducing our concept of buffer management, we first discuss what buffer management is like when a conventional general-purpose computer is used to support database management. The buffer manager is the set of modules that manages blocks (or physical records) of information. A block contains records (logical records) which are seldom relocated. Due to this static nature and the fact that records in a block may have diverse characteristics, not every record that is in a block will be relevant for a particular application. Hence when a block is transferred from the database to the buffer, not all information stored in the block will actually be utilized, thus effectively increasing the amount of I/O activity since some irrelevant data are transferred. Furthermore, the presence of irrelevant data decreases the effective buffer size and consequently increases the frequency of data transfer from the database to the buffer.

Due to the diverse characteristics of records stored in a block, the buffer manager has little or no specific knowledge about the content of a block. This results in some undesirable effects. First, when a particular record is retrieved by its contents, the entire buffer must be searched in order to determine if the record resides in the buffer so that on the average, half of the buffer needs to be searched. Since the buffer is implemented in a virtual memory environment, the I/O activity induced by paging is considerable. Second, the buffer manager has little knowledge to predict which block residing in the buffer will be used again in future references. A general policy known as a block replacement algorithm, must then be devised to replace the blocks stored in the buffer. Examples of block replacement algorithms are least-recently used (LRU), first-in-first-out (FIFO) and random. These algorithms are intended to increase the hit ratio of the blocks stored in the buffer. Since these algorithms are general in nature, the hit ratio can seldom be close to the ideal.

The buffer management concept employed in the IMSI is different from the conventional concept. When information is transferred to the ISB, it is retrieved

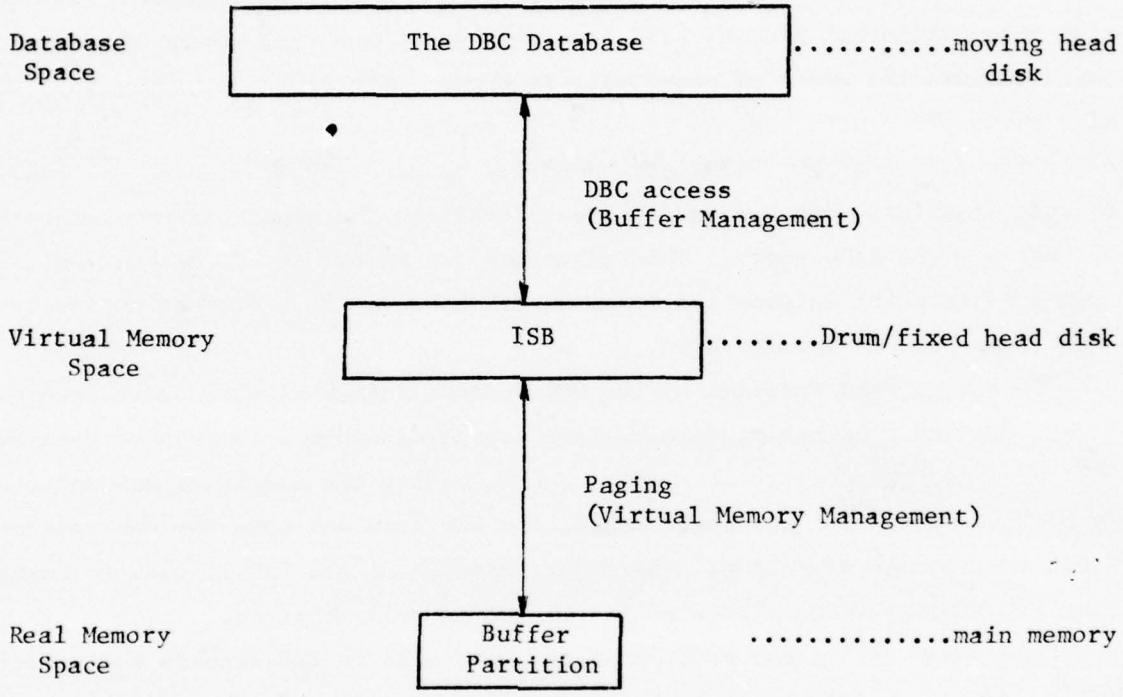


Figure 26. The virtual memory environment of the ISB.

from the DBC according to content, i.e. all the DBC records which are transferred to the ISB at one time satisfy a DBC query. Therefore, the system buffer manager (SBM) has specific knowledge on the content of the ISB. Furthermore, the processing of these DBC records follows an order which is defined by a keyword representing a sequence field of a segment. Using the content information and the ordering information available to it, the SBM can predict what will be the future references if the user continues to process these DBC records. The SBM also knows which DBC records have been referenced and will not be referenced again because the order of processing is known. Therefore, it can always determine which DBC records can be removed for replacement and a general replacement algorithm like the LRU, FIFO, etc. need not be used. Furthermore, all DBC records transferred to the buffer are relevant to the application because they all satisfy the same query. This minimizes the amount of I/O activities required to transfer information from the database to the buffer as no irrelevant data takes part in the transfer.

To sum up, the function of the SBM becomes fairly simple for two reasons. First, the SBM need not perform content searching since this is performed by the DBC. Second, the process of loading and unloading the ISB is determined completely by the translation process. Hence, the SBM need not have any generalized block replacement algorithm. The major function of the SBM is then to ensure that all virtual space allocated to the ISB is well utilized.

The space allocation problem arises when all the DBC records that should be placed in the ISB exceed the size limit of the ISB. This situation is possible, for the number of DBC records which satisfies a query is not fixed. When there is insufficient space in the ISB, the SBM must have a policy to temporarily unload some DBC records from the ISB even though they have not been referenced. Such a policy is described in the following section.

6.1. The Buffer Space Allocation Strategy

Virtual space is allocated to the ISB in terms of pages. The size of the ISB depends on the total virtual space available to the user and should be chosen to be as large as possible for the following reason. If we define the I/O cost as the sum of the cost of DBC access and the paging cost, and if we assume that the paging cost is less than the cost of DBC access, then a larger buffer size will decrease the I/O cost. This is because a larger ISB allows more DBC records to be contained in the virtual memory, thus effectively reducing the I/O cost when insufficient space in the ISB results in additional DBC access. The assumption that paging cost is less than the cost of DBC access is reasonable as the paging mechanism usually employs storage devices which have

no seek time, such as a drum, while the DBC employs moving head disks, which have seek time delay.

How is ISB space allocated internally? Segments of different types could be stored in the ISB (Note: when we say segment in the ISB, we actually mean the DBC record representing that segment). Hence it is necessary to allocate a number of pages in the ISB to each segment type defined in a logical data structure of the IMS database. Such allocation should be based on a priori knowledge about the average number of pages used by each segment type in the ISB. If no such knowledge is available, the allocation could be proportional to the percentage of the database space occupied by a segment type. The purpose of the allocation is to provide a policy to decide at least how much ISB space each segment type can have. We call this amount of space allocated to a segment type a quota.

At any given time, some segment type may not utilize part or all of its quota. There are two such possibilities. First, the space currently occupied by this type that is less than its quota will be referred to as the "left-over" space of this segment type. Second, the quota space not yet used at all will be referred to as the "not-yet-used" space of the segment type. To fully utilize the ISB space, the "left-over" space should always be consumed by other segment types whose space requirement exceeds its own quota. The "not-yet-used" space of a segment type can also be used by other segment types. Thus space wastage can be avoided due to under utilization of space by some segment types. However a buffer space reclamation problem is thus created. Steps must be taken to redistribute the buffer space if a segment type wants to reclaim the buffer space allocated to its quota, but the space is being used by other segment types. There are two steps in the reclamation process. The first step is garbage collection. Since the processing of the segments is in the order defined by their sequence field, the segments that have already been processed can be regarded as garbage. The garbage collection step will free all such garbage space. If this step results in sufficient space for the reclamation, then no further action is taken. Otherwise, the second step of reclamation is carried out to deallocate the space over-used by some segment types. Any segment type that has over-used its quota is a candidate for deallocation. The question now is how to choose a segment type for deallocation? The segment types in the highest level(i.e. level closest to the top level) should be deallocated first for the following reason. In the IMS database traversal process, segments at a higher level will not be required until all lower-level segments have been

traversed. Thus the removal of some segments in the higher level has a less immediate affect on the traversal process.

In the following subsection, we shall present the data structures necessary to implement the above buffer space allocation strategy.

6.2. Data Structures

The SBM maintains 3 tables for the management of the ISB. They are the ISB bit map, the ISB page table, and the segment control table. The ISB bit map shows which pages in the ISB are in use or not in use. It is used to locate the unused pages for space allocation. The ISB page table shows which pages are allocated to each segment type. The pages allocated to a segment type will have corresponding entries in the ISB page table linked by pointers. The list of linked entries in the ISB page table also gives the order in which the pages are allocated to a segment type, thus defining the order that these pages will be referenced. The segment control table contains information for each segment type. It includes some pointers to the ISB page table. The relationship between these three tables is shown in Figure 27.

6.2.1. The ISB Bit Map

Each bit in the ISB bit map corresponds to a page in the virtual space allocated to the ISB. A bit is set to 1 if the corresponding page in the ISB is allocated for use, otherwise it is set to zero. If the SBM wants to allocate a certain number of pages for a segment type, the ISB bit map will be searched sequentially for unallocated pages. Since there is one bit for each page in the ISB bit map, the size of the ISB bit map will be N/8 bytes where N is the total number of pages in the ISB. For example if N is 1000 then N/8 would be 125.

6.2.2. The ISB Page Table

The format of an ISB page table entry shown in Figure 28. Each entry corresponds to a page in the ISB. The pointer fields FPTR and BPTR are pointers to other entries in the ISB page table. They define a sequence of pages allocated to a segment type and are used to traverse the sequence of pages in a forward and backward direction, respectively. They are also used for the deallocation and garbage collection process discussed in Appendix B. Assuming that the ISB has no more than 1000 pages, then it will allow a conveniently large 4000K byte. address space if each page is 4K bytes. Hence the FPTR and BPTR fields will both need 10 bits to address the ISB page table. Therefore, the maximum size of this table is 2.5K bytes (=1000 x 20/8).

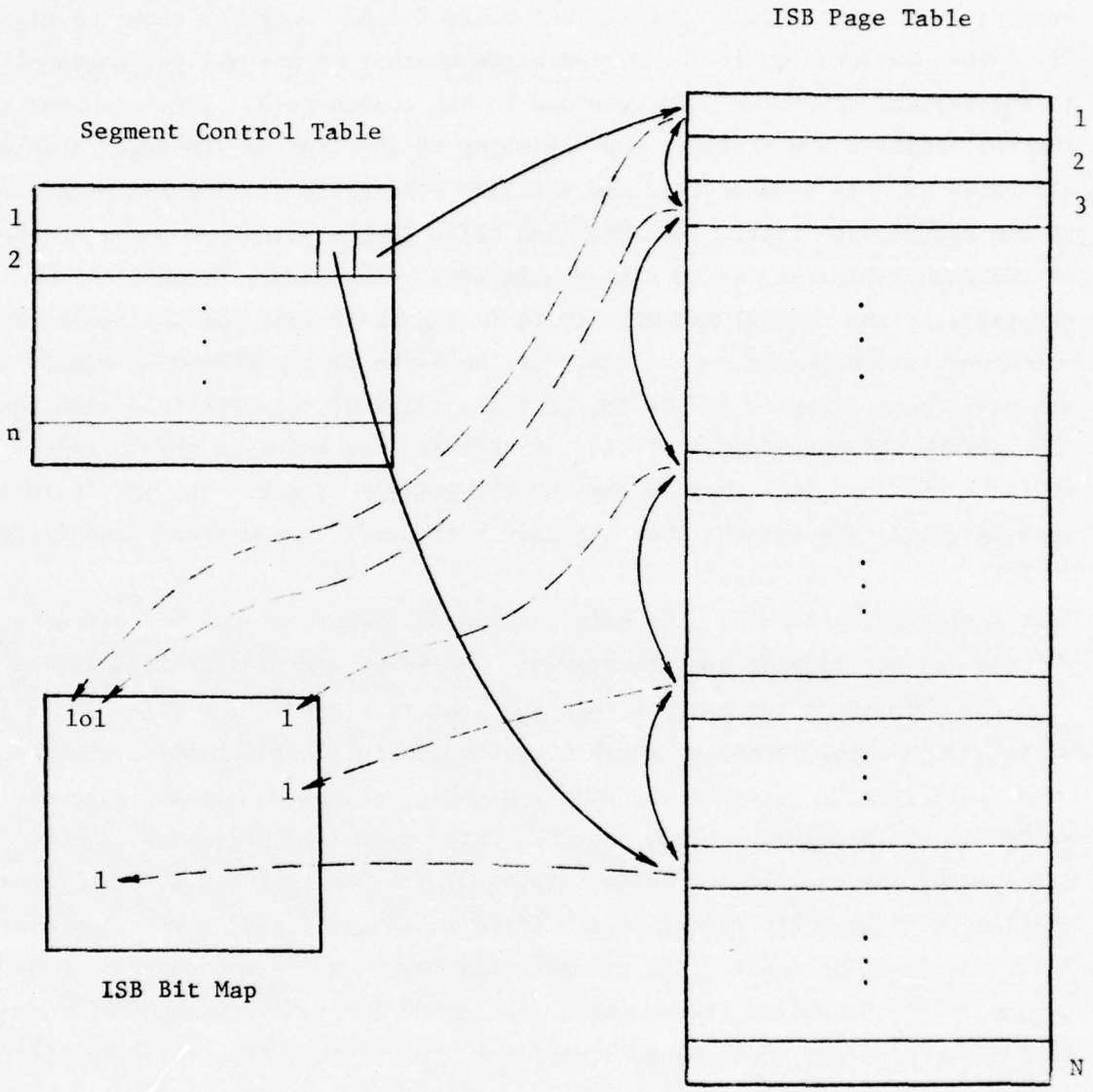


Figure 27. Segment control table, bit map, and ISB page table.

6.2.3 The Segment Control Table

Each entry in the segment control table contains information about a segment type. The format of a segment control table entry is shown in Figure 29. The deallocation field (D) indicates whether or not all the pages allocated to the segment type have been returned to the common pool. The list head field (LHEAD) contains a pointer to the beginning of the list of ISB page table entries allocated to this segment type and the list tail field (LTAIL) contains a pointer to the end of this list. The LHEAD and LTAIL fields allow traversal of the list of ISB page table entries in either direction. The old CS field (OCS) contains the value of the CURRENT SEGMENT field in the status information table(SIT) which was last accessed by the SBM. If the value of the CURRENT SEGMENT field has never been advanced by the IM, then the value of the OCS field will equal that of the CURRENT SEGMENT field. Otherwise, the value of the OCS field would be one less than that of the CURRENT SEGMENT field. The OCS field is used to recall the segment that was last retrieved. The current page field (CPAGE) contains the page number of the page containing the segment that was last retrieved by the IM. The offset field (OFFSET) contains the offset within the page of the segment last retrieved. The CPAGE and OFFSET fields together form the address of the segment that was last retrieved. The total field (TOTAL) contains the total number of pages allocated to this segment type. The used page field (UPAGE) contains the number of pages which precede the page indicated by CPAGE. These pages contain segments which have been processed and could be released by the garbage collection process. The length field (LENGTH) contains the length of each DBC record stored.(Here we assume fixed length segments). The sequence value field (SEQ) contains the value of the sequence field in a segment. The SEQ field is used when the ISB does not have enough space to contain the entire set of segments transferred to the IMSI from the DBC. A solution to this problem is to discard some of the segments transferred to the IMSI. Since the segments with higher sequence field values will be referenced at a later time, such segments should be discarded. Suppose that all segments whose sequence fields have values less than or equal to some value, say x, can be stored in the ISB, then the SEQ field will have the value x. When all the segments stored in the ISB have been referenced, then the SEQ field is used to retrieve the rest of the segments which have been discarded. This process can be repeated any number of times. In order to tell whether the SEQ field contains meaningful information, a field C is used to indicate its validity.

The maximum length of a segment control table entry is 44 bytes and the

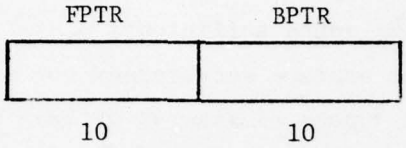


Figure 28. The format of an ISB page table entry

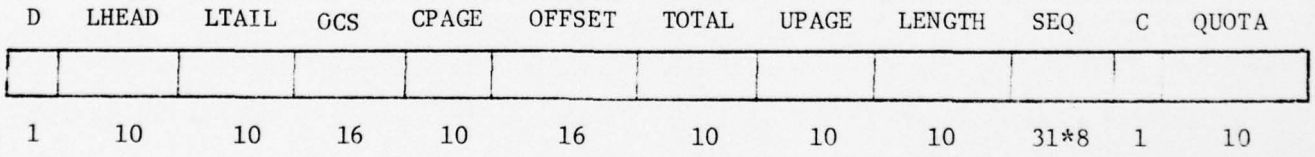


Figure 29. The format of a segment control table entry.

maximum number of segment types is 255. Therefore, the maximum storage requirement of the segment control table is approximately 10K bytes. However, this estimate is based on the worst case. In reality, a segment control table of 1K bytes should normally be quite sufficient.

In summary, the total storage requirement for maintaining these 3 tables should normally be within a page of size 4K bytes. Hence the access of these tables will not incur additional paging cost to the IMSI. The manipulation of these tables is described in Appendix B.

7. A COMPARATIVE STUDY OF IMS AND DBC PERFORMANCE

An analysis of DBC performance relative to IMS is made. To support the analysis, cases covering a variety of processing situations are constructed. We then compare the relative performance between IMS and DBC via these cases. Although these cases do not give a precise measure of performance since DBC has not yet been constructed, they do provide us with a better insight into the strengths of using the DBC in different processing environments.

A measure of performance used in these cases is considered in the double paging environment as discussed in Section 6. In the double paging environment, data are transferred from the database space to the virtual memory space in the (front-end) computer system, which is then subject to paging. For simplicity, the measure of performance will only be based on the cost of transferring data from the database space to the virtual memory space. This cost is measured by the number of database disk accesses to retrieve or store the required data. Thus, the measure does not take into account the software overhead (i.e., paging, buffer searching) for managing data stored in the virtual memory of the front-end computer. Such software requirements have been shown (in Section 6) to be favorable to the interface because the paging cost is greatly reduced by eliminating buffer searching.

Hierarchical data is represented in IMS by one of four organizations, known as HSAM, HISAM, HDAM, and HIDAM.* The first two, HSAM and HISAM are sequential representations while HDAM and HIDAM are direct. In our examples, we assume that hierarchical data is represented in IMS by the HIDAM (Hierarchical Indexed Direct Access Method) organization. The direct representation is chosen instead of the sequential representation because it offers more rapid access to segments within a database record than the sequential access methods. It also allows the DBC to be compared with the best possible IMS access method. The only difference between HIDAM and HDAM lies in the fact that the root segments are located by an index in HIDAM and by a hashing scheme in HDAM.

HIDAM allows root segments to be processed by an index and dependent segments to be processed using pointers. A HIDAM database, therefore, actually consists of two files--one contains the data and the other contains the index. Indexing is done on the sequence fields of the root segments. To locate a segment in the data file, the index file is used to provide an entry point to the data file. Within the data file, child/twin pointers (see Section 4.3) are

---

* A description of HIDAM is given in Appendix C.

used to represent the hierarchical structure. The data file is divided into blocks of equal length. A block is the unit of data that is transferred to and from the buffer pool area in the virtual memory. Each block transfer requires a disk access to the database. In the upcoming examples, we intend to measure the number of block transfers required to process a user transaction.

The IMS database used in the cases is shown in Figure 30. We assume that it has 1,000 root segments and the length of each segment is 200 bytes (including pointers, etc.). Each root segment has 30 children (i.e., OFFERING segment occurrences) of length 100 bytes each. Each OFFERING segment has 50 children (i.e., STUDENT segment occurrences) of length 100 bytes each. The structure of the IMS database is, in fact, a simplification of that given in Figure 7. This simplified structure is sufficient for the illustration since each DL/1 call involves only one hierarchical path. The entire IMS database is linearized according to its traversal sequence and segments are loaded in this order into the blocks of the data file. For instance, the first m segments in the sequence are stored in the first block of the data file. The next m segment will be stored in the second block, and so on.

An IMS database record is a course segment and all of its dependent segments and is, therefore, 153,200 (=200 + 30 x 100 + 50 x 30 x 100) bytes. Assuming the block size is 4K bytes, which is a favorable page size, the IMS database record will spread across 39 (=153,200/4000) blocks.

The examples used in the comparison involve a variety of processing situations including retrieval of a specific segment, retrieval of a number of segments, sequential traversal of the entire database, and addition of segments to and deletion of segments from the IMS database. The comparison is based on the number of disk accesses required by IMS and by the DBC. An analysis of the results of the comparison will be given later.

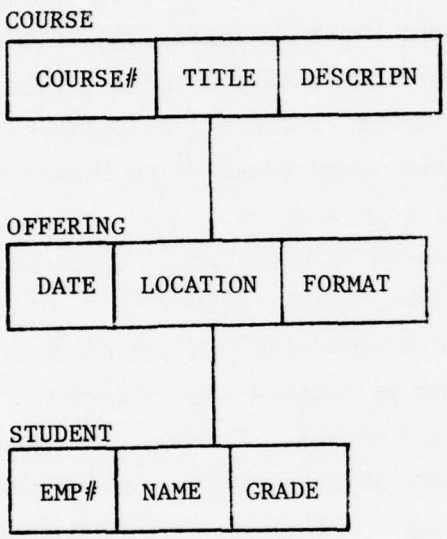
We also touch upon other performance issues such as the security.

7.1 Case Studies

Case 1: To retrieve a specific STUDENT segment.

```
GU    COURSE      (COURSE#=CIS211)
      OFFERING    (DATE=730105)
      STUDENT     (EMP#=1684)
```

In the IMS environment: We will consider the best case and the worst case. If the STUDENT segment which satisfies the call is the first STUDENT segment in the IMS database record, then the number of disk accesses can be calculated as follows:



	Length (bytes)	Occurrences
COURSE	200	1000
OFFERING	100	30(per course)
STUDENT	100	50(per offering)

Figure 30. The database structure used in the examples.

(1) One disk access[1] to the index database to locate the block containing the root (i.e., COURSE) segment.

(2) One disk access to retrieve the block containing the root segment.

Since the required STUDENT segment is stored in the same block as the root segment, no more database accesses are required. Hence the total disk accesses are two.

If the STUDENT segment which satisfies the call is the last STUDENT segment in the IMS database record, then the number of disk accesses can be calculated as follows:

(1) One disk access to the index database to locate the block containing the root segment.

(2) 30 more disk accesses to traverse from the first OFFERING segment to the last using the twin pointers (Since there are 30 OFFERING segments and since we may assume that each of them is located in a different block, there will be 30 disk accesses. The justification of assuming different blocks is as follows. On the average, there are 50 students per offering, each STUDENT segment requiring 100 bytes. Thus the average physical distance (in bytes) between two adjacent OFFERING segments is 5K bytes, which is larger than a page. We may, therefore, expect at least one disk access per OFFERING segment).

(3) One disk access to traverse from the last OFFERING segment to the last STUDENT segment (since the OFFERING segment and its last STUDENT segment are located in different blocks).

Hence, the total is 32 disk accesses. A rough estimate of the median can be calculated from the two extreme cases as 17 (=(32+2)/2).

In the DBC environment: The number of disk accesses is calculated as follows:

(1) One disk access to retrieve the root segment.

(2) One disk access to retrieve the OFFERING segment.

(3) One disk access to retrieve the STUDENT segment.

---

1. For simplicity, it is assumed that only one disk access is required to retrieve index information since the amount of index information for this example is small.

The total is 3 disk accesses. These results are summarized in the following table:

No. of Accesses	IMS	DBC
minimum	2	3
maximum	32	3
approximate median	17	3

Case 2: To retrieve a number of STUDENT segments.

```
        GU      COURSE      (COURSE#=CIS211)
                OFFERING    (LOCATION=LONDON)
                STUDENT     (GRADE='B')
LOOP:   GN      COURSE      (COURSE#=CIS211)
                OFFERING    (LOCATION=LONDON)
                STUDENT     (GRADE='B')
        GO TO LOOP
```

In the IMS environment: We will again consider the best case and the worst case. If there is no OFFERING segment in the database record which satisfies the qualification (LOCATION=LONDON), then the number of disk accesses required is calculated as follows:

(1) One disk access to the index database to locate the block containing the root segment.

(2) 30 more disk accesses to traverse from the first OFFERING segment to the last (see explanation in the previous example).

Hence the total disk accesses required to process the above transaction is 31.

On the other hand, if each OFFERING segment in the database record satisfies the qualification (LOCATION=LONDON), then the number of disk accesses required is given by:

(1) One disk access to the index database to locate the block containing the root segment.

(2) 39 more disk accesses to traverse the entire database record (since a database record resides in 39 blocks).

The total is 50. Hence the approximate median is 36 (=(31+40)/2).

In the DBC environment: The number of disk accesses is calculated as follows:

(1) One disk access to retrieve the root segment.

(2) One disk access to retrieve all the OFFERING segments which satisfies (LOCATION=LONDON). Let the number of OFFERING segments retrieved be x (where $0 \leq x \leq 30$).

(3) x disk accesses to retrieve the STUDENT segments under each OFFERING segment retrieved.

The total is x + 2. Hence the best case is 2 (when x=0) and the worst case is 32 (when x=30). The approximate median is 17.

These results are summarized in the following table:

No. of accesses	IMS	DBC
minimum	31	2
maximum	40	32
approximate median	36	17

It should be noted that if the sequence field of the COURSE segment is not used as a search assignment in the DL/1 call, then, for IMS, each COURSE segment would have to be examined to determine if it satisfies the qualification. This would require 1000 more disk accesses since there are 1000 COURSE segments each of which is located in a different block. However, for DBC, the number of disk accesses remain the same since it only requires 1 disk access to retrieve any COURSE segment(s).

Case 3: To sequentially traverse the entire IMS database.

```
        GU   COURSE
LOOP:   GN
        GO TO LOOP
```

In the IMS environment: As shown in Case 2, to traverse a database record 39 disk accesses are needed. Since there are 1000 database records, the total number of disk accesses would be 39000 (39 x 1000). The access to the index database is negligible since it may only take one or two disk accesses to transfer the entire index into the memory.

In the DBC environment: The number of disk accesses is calculated as follows:

(1) One disk access to retrieve all the root segments.

(2) For each root segment, one disk access is required to retrieve its dependent OFFERING segment. Since there

are 1000 root segments, the number of disk accesses required to retrieve the OFFERING segments is 1000.

(3) For each OFFERING segment, one disk access is required to retrieve its dependent STUDENT segment. Since there are a total of 30000 (=30 x 1000) OFFERING segments, the number of disk accesses is 30000.

Hence the total is 31001.

Case 4: To insert a new STUDENT segment.

```
ISRT    COURSE    (COURSE#=CIS211)
        OFFERING (DATE=730105)
        STUDENT
```

In the IMS environment: The number of disk accesses required to locate the (logical) position where the STUDENT segment can be inserted is the same as that calculated in Case 1, to retrieve a student segment, i.e., the minimum is 2. The maximum is 32 and the median is 17. It will also be necessary to store the segment in a new block since no space is assumed to be available in the existing blocks. Therefore, one more disk access is needed to actually insert the segment. The number of disk accesses for the insertion is:

3	(minimum)
33	(maximum)
18	(median)

In the DBC environment: The segment can actually be inserted in one disk access since the sequence fields of the ancestors are given in the DL/1 call. The DBC record can simply be formed using the sequence fields given in the DL/1 call and only one disk access is required to actually place the DBC record in an MAU. However, if the sequence fields of the ancestor are not given in the DL/1 call, they must be retrieved first. It takes one disk access to retrieve the root segment and one disk access to retrieve the OFFERING segment. After retrieving these two segments, the DBC record for the STUDENT segment can be formed using the symbolic identifier of its parent. It can then be inserted into the DBC in approximately one disk access. Therefore, insertion requires a minimum of 1 and a maximum of 3 accesses.

Case 5: To delete a COURSE segment.

```
GHU     COURSE     (COURSE#=CIS211)
DLET
```

In the IMS environment: Since all the dependent segments of the COURSE segment must also be deleted, the entire database record must be traversed in order to complete the database. The number of disk accesses is 40 (one disk access to the index database, 39 disk accesses to traverse the database record).

In the DBC environment: Only one DBC access is needed to delete the root. Since, according to the second clustering policy, the root and its dependent segments are clustered in two different MAUs, the total number of disk accesses is actually 2.

7.2. A Performance Analysis

A summary of the results of the preceding comparisons is given in Figure 31. Based on these observations, an analysis is made on the merits of using the DBC to support hierarchical databases.

The DBC has superior performance in updating operations. First, it simplifies the process of inserting a segment into an IMS database. In IMS, inserting a segment requires searching of the IMS database for a (logical) position in which the segment can be placed. Address pointers are then adjusted in the IMS database to support the actual insertion. This process is time-consuming since the IMS system must search the database to establish the (logical) position for insertion. In DBC, the segment can be placed anywhere in the DBC database and there is no need to search the DBC database if the symbolic identifier of its parent can be determined from the insert call. Furthermore, there is no need to fix pointers for an insertion operation.

Second, DBC also simplifies the process of deleting a segment. The argument is similar to that of insertion. Furthermore, the dependent segments of the parent segment being deleted can be deleted at the same time with little overhead, because the symbolic identifier of the parent appears in each of its dependent segments. Hence, a DBC deletion command with a symbolic identifier as the parameter will delete all segments having that symbolic identifier, thereby automatically deleting the parent and all of its dependent segments in one operation. On the other hand, for IMS to delete all

Case study		Environments		
			IMS	DBC
1.	To retrieve a specific segment	min	2	3
		max	32	3
2.	To retrieve a number of segments	min	31	2
		max	40	32
		"med"	36	17
3.	To traverse the entire database		39000	31000
4.	To insert a segment	min	3	1
		max	33	2
		"med"	18	3
5.	To delete a segment		40	2

Note: "med" = (min + max)/2

Figure 31. Summary of results

the dependent segments of the parent, each dependent segment must be located individually and then deleted, a time-consuming process.

With regard to retrieval operations, as seen in Cases 1 and 2, the DBC performs well in retrieving a specific segment or a small number of segments requiring a search of a large portion of the database. In Case 1, a specific segment is retrieved. For IMS, the number of disk accesses depends on the length of the (pointer) path which is used to traverse from the entry point of the database to the required segment. If this path spans many blocks, the number of disk accesses required would be large. However, in DBC, the notion of a (pointer) path does not exist. By embedding the symbolic identifier of the parent into the child segments, each child segment can be searched independently and, therefore, it is path independent. The number of disk accesses is sensitive to the number of levels that are traversed in reaching the segment.

In Case 2, a number of segments are retrieved. For IMS, the number of disk accesses again depends on the length of the (pointer) path required for traversal. For DBC, it depends (in this case) on the number of OFFERING segments, x, which satisfies the qualification. If x is small, then the number of disk accesses will be small.

The capability of the DBC for sequential processing is demonstrated in Case 3. Even though the performance gap between IMS and DBC is narrowed in comparison to Cases 1 and 2, the DBC, which is not designed for sequential processing, can still out-perform IMS, which is designed for sequential processing.

In addition tc the previously described benefits, the DBC allows more flexibility in retrieval operations. In IMS, in order to retrieve a segment, the system must enter the database through an entry point. Normally, the entry points are limited to the root segments because an index (called primary index) is created and maintained automatically only for the root segments. Although the user has the option to define secondary indices for the dependent segments for the purpose of entering the database in places other than the root segments (see Figure 32), the use of secondary indices requires extra amounts of storage and maintenance for the index files. In DBC, however, every segment can be used as an entry point to the database because segments stored in the DBC are not located by pointers or by adjacency. Each segment, stored in the DBC, can be located individually by its contents, without depending on its position in the database, thus providing

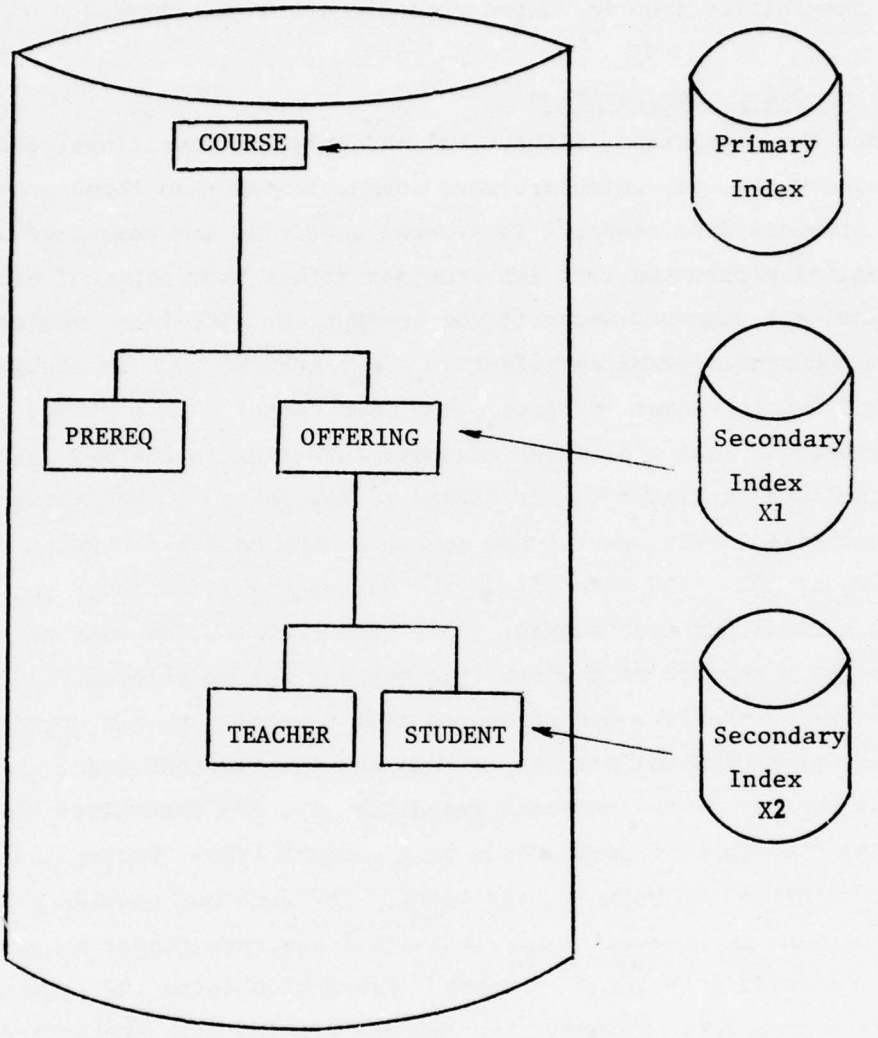


Figure 32. Indexing a database with secondary indices.

retrieval flexibility even if secondary indices are not created.

7.3. Security Consideration

Besides the advantages in retrieval and updating operations, the DBC offers security features which are more sophisticated than those provided in IMS. IMS provides data security in several ways. We are concerned with the kind of logical protection that IMS provides rather than physical protection, like terminal and password security, or encryption. The first logical protection concerns segment sensitivity. If a segment type is designated to be not sensitive to a user (program), the user cannot have any kind of access to the segments of that type. For example, referring to the IMS database structure defined in Figure 7, if PREREQ is designated to be insensitive to user A, then effectively, user A has access to the database structure as shown in Figure 33. IMS also allows the database administrator to specify processing options for each segment type. For example, the user may be allowed to get a segment of a given type but may not be allowed to perform delete, insert, or replace operations on that segment type. In other words, segments are protected collectively within the same segment type. By overlooking the content of the segments expeditiously, IMS determines whether the user has the right to have access to a segment type. Protection is said, therefore, to be on the segment type level. IMS does not provide protection below the segment type level, i.e., individual segments cannot be protected differently according to their contents. Protection below the segment type level is more involved. However, the DBC can provide such protection by hardware.

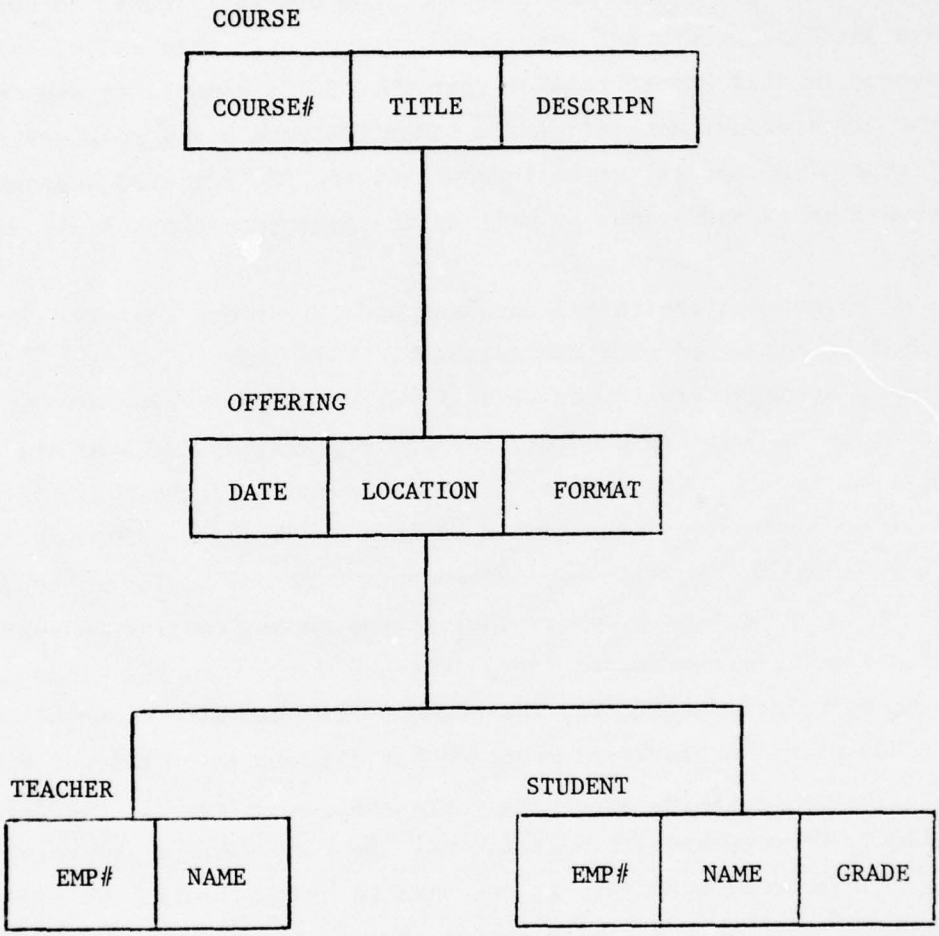


Figure 33. A case of access control.

8. CONCLUDING REMARKS

As a multi-model database computer, the DBC is intended to support various database models and associated systems. To this end, we have endeavored in this report to show that the DBC is capable of supporting the hierarchical data model and IBM's IMS data management system. In two forthcoming reports, we will show that the DBC can also support the relational model and system as well as the CODASYL network model and system.

To support a hierarchical database model, the original IMS database must be converted to a DBC database. This one-time database conversion is straightforward and cost-effective. By straightforward, we mean that the algorithms provided for the conversion, known as the representations methods are simple. The main purpose of these algorithms is to remove address-pointers embedded in the IMS database and to introduce the symbolic identifiers. Address-pointers are no longer needed in the DBC since the mass memory (MM) of the DBC is content-addressable. Symbolic identifiers preserve the parent-child and twin relationships. By using symbolic identifiers, IMS segments can not only be accessed in accordance with the traversal sequence but also can be retrieved directly. In other words, both the sequential-oriented search for and the random access to individual IMS segments are facilitated. Because the storage requirement for symbolic identifiers is compensated by the removal of address-pointers and because the random access capability has made the need for secondary indices unnecessary, the DBC database is cost-effective.

In addition to database conversion, the DBC requires a software interface, known as IMSI, to support IMS application programs. IMSI enables IMS users to run IMS (application) programs in the same computer system environment on which they were developed (say, an IBM 360/370 computer system) while utilizing DBC's storage and search capabilities (via communication lines) for IMS database records without any modification of the user programs and without the presence of the IMS data management system. It is interesting to note that the software requirement for the interface is minimal. Essentially, IMSI intercepts the user programs' DL/I calls, converts the calls to equivelent DBC commands, sends the commands to DBC for execution, keeps track of the IMS segments passed by the DBC, manages the buffer areas which accomodate the segments,

and clears and resets buffers and tables. Because the DBC accepts queries in the form of a Boolean expression of keyword predicates and because it has commands in all storage, retrieval and update modes, the conversion of DL/1 calls to the DBC queries and commands is straightforward. Due to DBC's content-addressability, segments retrieved from the mass memory and placed in the buffer are all 'valid' segments. There is no need for the buffer management routine to have an elaborate segment searching algorithm. Furthermore, the size of the buffer need not be large since it contains no invalid segments.

There are other advantages in utilizing the DBC which are not available if the IMS database were running in a conventional computer system environment:

(1) The DBC can concurrently host several types of data models and interface with different data management systems, making it possible to communicate among different models and systems.

(2) The built-in security mechanisms are far more advanced than IMS can offer, allowing users to enjoy more adequate access control and data protection.

(3) The built-in clustering mechanisms can improve performance. Since the partitioned content-addressable memories (PCAMs) utilize very large partitions (in mass memory, the size of the partition is the size of the cylinder), clustering is easy and effective. Furthermore, only the relevant segments of a cluster in a partition are output. Therefore, one is not concerned with the kind of problems associated with small page size in a conventional computer system with virtual memory.

(4) The DBC can provide overall throughput improvements over conventional computer systems, since the DBC is destined to support very large databases of 10^{10} bytes with good cost/performance. It is difficult for a conventional computer system to support a growing database application by adding more disks and software. Furthermore, the DBC can relieve the general-purpose computer from using much needed CPU cycles for data management tasks.

REFERENCES

1. Baum, R.I., Hsiao, D.K., and Kannan, K., "The Architecture of a Database Computer--Part I; Concepts and Capabilities", The Ohio State University, Tech. Rep. No. OSU-ClSRC-TR-76-1, (September, 1976).
2. Hsiao, D.K. and Kannan, K., "The Architecture of a Database Computer--Part II: The Design of Structure Memory and Its Related Processors", The Ohio State University, Tech. Rep. No. OSU-ClSRC-TR-76-2, (October,1976).
3. Hsiao, D.K. and Kannan, K., "The Architecture of a Database Computer--Part III: The Design of the Mass Memory and Its Related Components", The Ohio State University, Tech. Rep. No. OSU-ClSRC-TR-76-3, (December,1976).
4. IBM, Information Management System/Virtual Storage (IMS/VS) Version 1, General Information Manual, GH20-1260-4.
5. IBM, Information Management System/Virtual Storage (IMS/VS) Version 1, Application Programming Reference Manual, SH20-9026-4.
6. IBM, Information Management System/Virtual Storage (IMS/VS) Version 1, System/Application Design Guide, GH 20-9025-4.
7. IBM, Information Management System/Virtual Storage (IMS/VS) Version 1, System Programming Reference Manual, SH20-9027-4.
8. Schkolnick, M., "Clustering Algorithm for Hierarchical Structures", ACM Trans. Database Systems 2, 1(March 1977), 27-44.
9. Sherman, S.W. and Brice, R.S., "Performance of a Database Manager in a Virtual Memory System", ACM Trans. Database Systems 1,4 (December 1976), 317-343.
10. Tuel, W.G., "An Analysis of Buffer Paging in Virtual Storage Systems", Research Rep. RJ 1421, IBM Research Laboratory, San Jose, Calif., July 1974.
11. Rodriquez-Rosell, J and Hildebrand D., "A Framework for Evaluation of Data Base Systems", Research Rep. RJ 1587, IBM Research Laboratory, San Jose, Calif., May 1975.

APPENDIX A - THE ALGORITHMS FOR THE TRANSLATION PROCESS

This section provides details of how each DL/1 call is processed. It should be noted that before any DL/1 call can be processed by the IMSI, the (DBC) file containing the IMS database must be opened using the preparatory DBC command given in Section 2.3.

Since the interface system buffer (ISB) is managed only by the system buffer manager (SBM), the DL/1 interface module (IM) communicates with the SBM by subroutine calls which then perform the functions requested by the IM on the ISB. These functions include retrieving deleting, replacing and inserting a segment in the ISB, loading and unloading of the ISB.

For implementation purposes, the status information tables (SIT) discussed in Section 5 would require an additional field to each segment type. This field, the VALIDITY field, having value 1 or 0 is used to indicate whether the information given in the entry is meaningful or not. For abbreviation, V, CS and QUAL will be used to stand for the VALIDITY, CURRENT-SEGMENT and QUALIFICATION fields, respectively. We present a description of these functions in the following.

(1) Fetch-current.

The parameters of the call are k (segment type), and addr (a storage location in the IM). The execution of the call causes the current segment of type k (i.e. the segment indicated by CP(k) in the status information table) to be retrieved from the ISB and transferred to the location addr.

(2) Replace-current.

This call is used to replace a segment stored in the ISB by the segment supplied as an argument. The parameters of the call are k (the segment type) and addr (the storage location of the segment used to replace the one in the ISB). The segment replaced is the current segment of type k.

(3) Delete-current.

The parameter of the call is k (the segment type). The execution of the call deletes the current segment of type k from the ISB.

(4) Insert-as-current.

The parameters of the call are k (the segment type) and addr (a storage location in IM). The execution of the call inserts the segment

addressed by addr into the ISB and establishes this segment in the ISB as the current segment of type k.

(5) Release-buffer-space.

The parameter is k (the segment type). The execution of the call releases all buffer space allocated to segment type k.

(6) Load-buffer.

The parameters are k (the segment type) and command-ID (i.e., a DBC command identifier). This call causes the SBM to load the response set from the DBC identified by command-ID into the segment type k portion of the ISB. This call is issued to the SBM after the IM has issued a DBC retrieve command to the DBC.

A.1 Processing the Get Calls

The get-unique (GU), get-next (GN) and get-next-within-parent (GNP) calls have been shown to be similar except for the initial setting of the parent position and the current position in the IMS database. Hence the processing of each of the three types of call is described by basically the same set of algorithms except for the initialization part. The get-hold calls will be treated as semantically equivalent to their respective get calls.

In the algorithms, we assume the SSAs for the get call are (So, Qo), (S_1, Q_1),..., (Sn, Qn). Furthermore, the variable parent contains a segment type indicating the parent position. Similarly, the variable CPDB contains a segment type indicating the current position in the database. Notice that parent and CPDB contain only segment types and do not directly address the actual segments. The actual segments can be located by the respective CS fields in the SIT.

The subroutine structure of the get call is depicted in Figure 34. The initialization process for three get calls is performed by algorithms A (for GU), B (for GN) and C (for GNP). This initialization process includes the setting of the appropriate parent position and current position in the database. Algorithm D first determines at which level processing should start by comparing the input qualifications with the qualifications stored in the SIT. It then, calls Algorithms E, F or G depending on the cases (to be discussed) to complete the processing. Essentially they bring in a set of segments on each level starting from the level determined by algorithm D down to the lowest level specified by the get

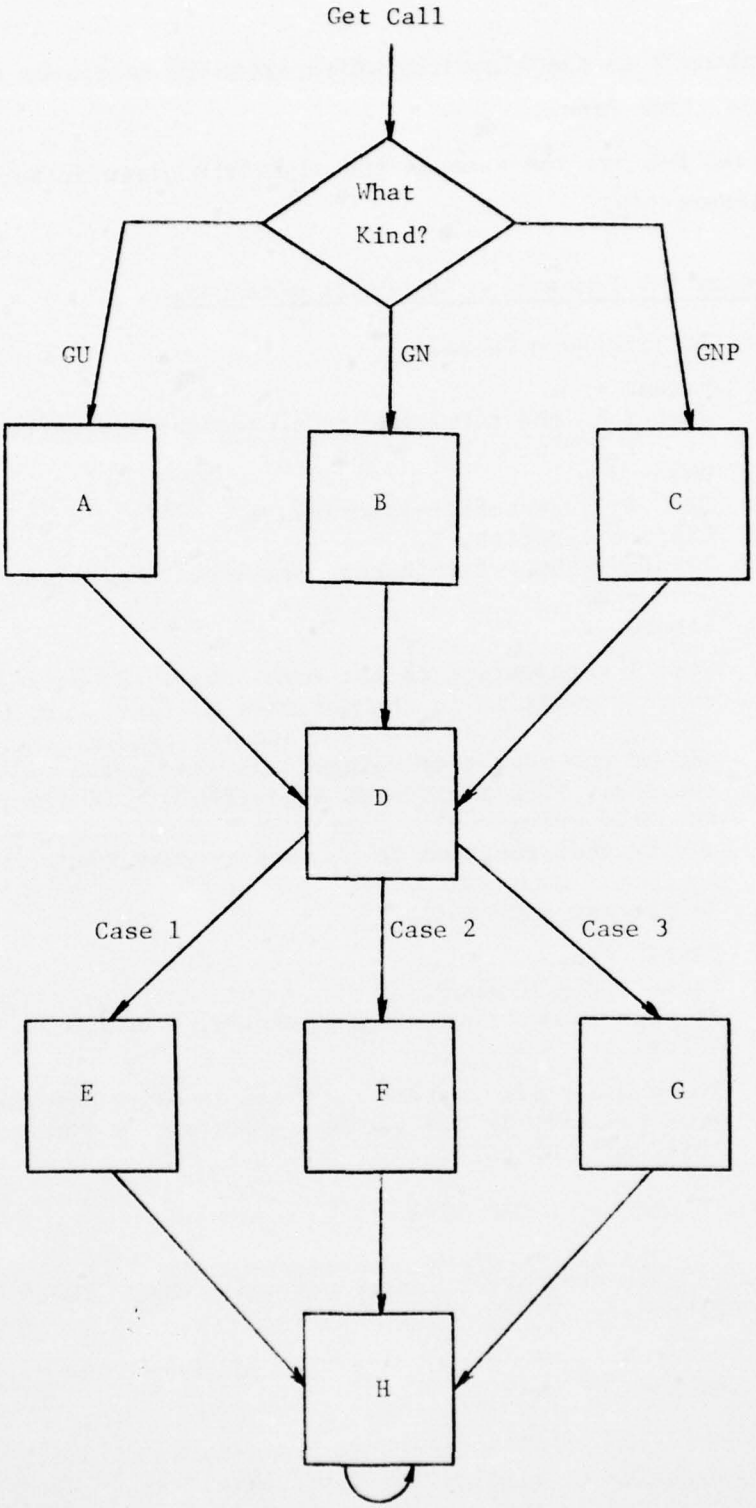


Figure 34. The Subroutine Structure of the Get Call.

call. Algorithm H is the algorithm which actually retrieves a set of segments at a given level.

Algorithms D-H are the same as the algorithm given in Section 5 although specified differently.

A.2 Algorithms for Processing Get Calls with SSAs

ALGORITHM A: To process a GU call.

Step 1: Parent ← o.
Step 2: (Let r be the total number of segment types) For i=1, 2, ..., r,perform step 3 and 4.
Step 3: V(i) ← o.
Step 4: Call Release-buffer-space(i).
Step 5: Execute algorithm D.
Step 6: If algorithm D terminates normally, then Parent ← Sn; CPDB ← Sn.
Step 7: Return.

Notes: Step 1 sets Parent to the zero level. Steps 2-4 set the current position in the database to level zero by clearing the validity field for each segment type greater than zero and at the same time release all the spaces allocated the ISB. Step 5 executes algorithm D. If algorithm D is executed successfully,then step 5 resets the parent position and current position to be used by subsequent calls.

ALGORITHM B: To process a GN call.

Step 1: Parent ← o.
Step 2: Execute algorithm D.
Step 3: If algorithm D terminates normally, then Parent ← CPDB ← Sn.
Step 4: Return.

Notes: These steps are similar to those in algorithm A. The current position in the database need not be reset because this was a GN call.

ALGORITHM C: To process a GNP call.

Step 1: Execute algorithm D.
Step 2: If algorithm D terminates normally, then CPDB ← Sn.
Step 3: Return.

Note: In step 2, the parent position is not reset due to the rule of the GNP call.

ALGORITHM D: This algorithm initializes the search of the IMS traversal sequence by finding the first level, if it exists, where QUAL(Si) is incompatible with Qi(i.e., Qi≠QUAL(Si) and Qi≠NULL). It then distinguishes three cases and makes calls to algorithms E, F, and G respectively. The cases are:

E. QUAL(Si) is incompatible with Qi for some i.
F. QUAL(Si) is not incompatible with Qi for i=0, 1, ..., m and m<n.
G. QUAL(Si) is not incompatible with Qi for i=0, 1, ..., m and m=n.

Step 1: $i \leftarrow 1$.
Step 2: If $V(Si) = o$, then go to Step 5.
Step 3: $i \leftarrow i + 1$.
Step 4: If $i \leq n$, then go to Step 2.
Step 5: $m \leftarrow i - 1$.
Step 6: For $i = 1, 2, \ldots, m$, perform Step 7.
Step 7: If Qual(Si) is incompatable with Qi then execute algorithm E with argument i; return with condition from algorithm E.
Step 8: If $m < n$, then execute algorithm F; else, execute algorithm G.
Step 9: Return with condition.

Notes: Steps 1-5 locate the lowest level n such that the SIT contains meaningful data on the segment types So, Si, ... ,Sn. Steps 6-7 locate the smallest i such that Qual(Si) is incompatable with Qi. If such an i exists, then algorithm E is executed. Otherwise, either algorithm F or G is executed depending on whether or not m is less than n.

ALGORITHM E:

Input parameter: i, a segment type.

Step 1: Issue a Fetch-current call to the SBM to fetch the current segment of segment type Si.
Step 2: Extract the value of the sequence field of the segment into x (assume SEQ is the field name).
Step 3: Let $K_1, K_2, \ldots, K_t$ be the keywords of the symbolic identifier of the parent of the segment obtained in Step 1.
Step 4: Form the DBC query: $(TYPE = Si) \wedge (SEQ \geq x) \wedge K_1 \wedge K_2 \wedge \ldots \wedge K_t \wedge Qi$.
Step 5: Issue a retrieve-by-query-with-pointer command to the DBC using the query created in Step 3 and the sort attribute SEQ.
Step 6: Issue a Load-buffer call to the SBM to load the response set of the above command into the ISB.
Step 7: If the SBM indicates an empty response set, then perform Steps 8-12; else go to step 13.
Step 8: If S_{i-1} = Parent, then return 'not-found'.
Step 9: $CS(S_{i-1}) \leftarrow CS(S_{i-1}) + 1$.
Step 10: Execute Algorithm H with input argument i.
Step 11: If a 'not-found' condition is returned from Step 10, then return 'not-found'.
Step 12: Go to Step 15.
Step 13: Qual(Si) $\leftarrow$ Qi.
Step 14: V(Si) $\leftarrow$ 1.
Step 15: For all j such that Sj is a dependent of Si, perform Step 16-17.
Step 16: V(Sj) $\leftarrow$ o.
Step 17: Call Release-buffer-space(S_j).
Step 18: CS(Si) $\leftarrow$ 1.
Step 19: If $i < n$, then go to Step 23; else perform Steps 20-22.

Step 20: Issue a Fetch-current call to the SBM to fetch the current segment of segment type Sn.
Step 21: Send the segment to the user.
Step 22: Return.
Step 23: i ← i + 1.
Step 24: Execute algorithm H with input argument i.
Step 25: If 'not-found' is returned from Step 24, then return 'not-found'; else go to Step 19.

Notes: This algorithm implements the first case discussed in Algorithm D. Steps 1-6 retrieve a new set of segments under the current parent with sequence fields having values greater than or equal to that of the current segment at level i. Steps 13-18 change the content of the status information table to reflect the changes in the ISB from by Steps 1-6. If Steps 1-6 retrieve an empty response set, then Steps 7-12 are executed to retrieve a new set of segments at level i under the segment which is next to the current parent at level i-1. Steps 19-25 retrieve a new set of segment at level i+1 under the current parent at level i. These last steps are repeated zero or more times until the level n is reached.

ALGORITHM F:

Step 1: i ← m.
Step 2: i ← i + 1.
Step 3: Execute algorithm H with input argument i.
Step 4: If 'not-found'condition is returned from Step 3, then return 'not-found'.
Step 5: If i < n, then go to Step 2.
Step 6: Issue a Fetch-current call to the SBM to fetch the current segment of segment type Sn.
Step 7: Send the segment to the user.
Step 8: Return.

Notes: This algorithm implements the second case discussed in Algorithm D. It retrieves a new set of segments from level m + 1 to level n by calling algorithm H.

ALGORITHM G:

Step 1: i ← m.
Step 2: CS(Si) ← CS(Si) + 1.
Step 3: If CS(Si) > COUNT(Si), then perform Steps 4-11; else go to Step 9.
Step 4: If i=1, then return 'not-found'.
Step 5: If S_{i-1}=Parent, then return 'not-found'.
Step 6: $CS(S_{i-1})$ ← $CS(S_{i-1})$ + 1.
Step 7: Execute algorithm H with input arguement i.
Step 8: If 'not-found' condition is returned from Step 7, then return 'not-found'.
Step 9: Issue a Fetch-current call to the SBM to fetch the current segment of segment type m.
Step 10: Send the segment to the user.
Step 11: Return.

Note: This algorithm implements the third case as discussed in Algorithm D.

ALGORITHM H:

Input argument: i, a level number. Retrieves the first sets of segments of types S_o, S_1, ..., S_i which satisfy Q_o, Q_1, ..., Q_i starting from the current segment of segment type S_{i-1}.

Step 1: If $CS(S_{i-1}) > COUNT(S_{i-1})$, then go to Step 11.
Step 2: Issue a Fetch-current call to the SBM to fetch the current segment of segment type S_{i-1}. If the segment is "deleted", then $CS(S_{i-1}) \leftarrow CS(S_{i-1}) + 1$ and go to Step 1.
Step 3: Let K_1, K_2, ..., K_t be the keywords of the symbolic identifier of the segment fetched in Step 2.
Step 4: Form the DBC query: $(TYPE = Si) \wedge K_1 \wedge K_2 \wedge \ldots \wedge K_t \wedge Qi$.
Step 5: Issue a retrieve-by-query-with-pointer command to the DBC using the query created in Step 4 and the sort attribute is the sequence field name of Si.
Step 6: Issue a Load-buffer call to the SBM to load the response set of the above command into the ISB.
Step 7: If the SBM indicates an empty response set,then go to Step 8;else go to Step 30.
Step 8: If $Qual(S_{i-1}) = Q_{i-1}$,then go to Step 9;else.go to Step 16.
Step 9: $CS(S_{i-1}) \leftarrow CS(S_{i-1}) + 1$.
Step 10: If $CS(S_{i-1}) \leq COUNT(S_{i-1})$,then go to Step 2.
Step 11: If $S_{i-1} = o$,then return 'not-found'.
Step 12: If S_{i-2} = Parent,then return 'not-found'.
Step 13: $CS(S_{i-2}) \leftarrow CS(S_{i-2}) + 1$.
Step 14: Execute algorithm H with input argument (i-1).
Step 15: If 'not-found' condition is returned from Step 14,then return 'not-found';else.go to Step 2.
Step 16: Issue a Fetch-current call to the SBM to fetch the current segment of Si-1.
Step 17: Extract the value of the sequence field, say SEQ, of the retrieved segment into x.
Step 18: Let K_1, K_2, ..., K_t be the keywords of the symbolic identifier of the parent of the retrieved segment.
Step 19: Form the DBC query: $(TYPE = S_{i-1}) \wedge (SEQ \geq x) \wedge K_1 \wedge K_2 \wedge \ldots \wedge K_t \wedge Q_{i-1}$.
Step 20: Issue a retrieve-by-query-with-pointer command to the DBC using the query created in Step 19 and the sort attribute SEQ.
Step 21: Issue a Load-buffer call to the SBM to load the response set of the above command into the ISB.
Step 22: If the SBM indicates an empty response set,then go to Step 12.
Step 23: $Qual(S_{i-1}) \leftarrow Q_{i-1}$.
Step 24: $CS(S_{i-1}) \leftarrow 1$.
Step 25: $V(S_{i-1}) \leftarrow 1$.
Step 26: For all j such that S_j is a dependent of S_i perform Steps 27-28.
Step 27: $V(S_j) \leftarrow o$.
Step 28: Call Release-buffer-space(Sj).

Step 29: Go to Step 2.
Step 30: Qual(S_i) ← Q_i.
Step 31: V(S_i) ← 1.
Step 32: CS(S_i) ← 1.
Step 33: For all j such that S_j is a dependent of S_i, perform Steps 34-35.
Step 34: V(S_j) ← o.
Step 35: Call Release-buffer-space(S_j).
Step 36: Return.

Notes: This algorithm retrieves a set of segments of type Si (lets call them child segments) which satisfy Qi. First, it must fetch the parent so that its symbolic identifier can be used to retrieve the child segments. There are two cases (tested for in Step 1). First, the parent under consideration does not reside in the ISB, then algorithm H calls itself to retrieve the set of parent segments (i.e.,of type S_{i-1}). This is performed in Steps 11-15. In the second case, the parent in consideration resides in the ISB. Then, Step 2 is executed to fetch the parent from the ISB and Steps 3-6 are performed to retrieve the child segments using the parent's symbolic identifier. However the situation is more complex if Steps 3-6 retrieve an empty set of child segments. Then the "next" parent segment should be established as the new parent. It is possible that the set of parent segments residing in the ISB is not suitable to be used to provide the "next" parent. This is the case when Qual(S_{i-1}) is not equal to Q_{i-1}. This case is tested in Step 8. If Qual(S_{i-1}) is equal to Q_{i-1}, then the "next" parent segment can be established as the new segment (Steps 9-10). Otherwise, Steps 16-22 are executed to retrieve the parent segments from the DBC.
Steps 23-28 change the SIT to reflect the retrieval of a new set of parent segments. Steps 30-35 change the SIT to reflect the retrieval of the child segments.

A.3 Algorithms for Processing Get Calls without SSAs

The algorithms given in the previous section apply to any get call with SSAs. This section presents the algorithms for processing a GN or GNP call without SSAs. In the former, the IM has to use only the segment types given in the SSAs to process the call. However,in the latter, any segment type in the database is eligible. The IM simply follows the traversal sequence and retrieves the "next" segment on the sequence.

The following definitions are used in the algorithms that follow. Leftmost-child(S) denotes the leftmost child segment type under the segment type S. It has a NIL value if S has no child segment types. Next-brother(S) denotes the segment type which is on the same level as S and is next right to S. It has a NIL value if S has no segment type on its

right side. Parent#(S) denotes the segment type of the parent segment of S.

ALGORITHM A: To process a GN call with no SSA.

Step 1: Parent ← o.
Step 2: Execute algorithm C.
Step 3: If algorithm C terminates normally,then Parent ← CPDB.
Step 4: Return.

ALGORITHM B: To process a GNP call with no SSA.

Step 1: Execute algorithm C.
Step 2: Return.

ALGORITHM C:

Step 1: S ← CPDB.
Step 2: r ← Leftmost-child(S)
Step 3: If r = NIL,then go to Step 8.
Step 4: Temp ← r.
Step 5: r ← S.
Step 6: S ← Temp.
Step 7: Go to Step 25.
Step 8: If S = Parent,then return 'not-found'.
Step 9: If Qual(S) = NULL,then go to Step 17.
Step 10: Issue a Fetch-current call to the SBM to fetch the current segment of segment type S.
Step 11: Extract the value of the sequence field, say SEQ, of the retrieved segment into x.
Step 12: Let K_1, K_2, ..., K_t be the keywords of the symbolic identifier of the parent of the retrieved segment.
Step 13: Form the DBC query: $(TYPE = S) \wedge (SEQ \geq x) \wedge K_1 \wedge K_2 \wedge \ldots \wedge K_t$.
Step 14: Issue a retrieve-by-query-with-pointer command to the DBC using the query created in Step 13 and the sort attribute SEQ.
Step 15: Issue a Load-buffer call to the SBM to load the response set of the above command into the ISB.
Step 16: If the SBM indicates an empty response set,then go to Step 19;else,go to Step 31.
Step 17: CS(S) ← CS(S) + 1.
Step 18: If CS(S) ≤ COUNT(S),then go to Step 37.
Step 19: r ← Next-brother(S).
Step 20: If r ≠ NIL,then go to Step 23.
Step 21: S ← Parent#(r).
Step 22: Go to Step 8.
Step 23: S ← r.
Step 24: S ← Parent#(S).
Step 25: Issue a Fetch-current call to the SBM to fetch the current segment of segment type r.
Step 26: Let K_1, K_2, ..., K_t be the keywords of the symbolic identifier of the retrieved segment.
Step 27: Form the DBC query: $TYPE = S \wedge K_1 \wedge K_2 \wedge \ldots \wedge K_t$.
Step 28: Issue a retrieve-by-query-with-pointer command to the DBC using the query created in Step 27 and the sort attribute is the sequence field name of S.

Step 29: Issue a Load-buffer call to the SBM to load the response set of the above DBC command into the ISB.
Step 30: If the SBM indicates an empty response set then go to Step 19.
Step 31: Qual(S) ← null.
Step 32: V(S) ← 1.
Step 33: CS(S) ← 1.
Step 34: For each j such that S_j is a dependent of S perform Steps 35-36.
Step 35: V(S_j) ← o.
Step 36: Call Release-buffer-space (S_j).
Step 37: CPDB ← S.
Step 38: Issue a Fetch-current call to the SBM to fetch the current segment of type S.
Step 39: Send the segment to the user.
Step 40: Return.

Notes: The algorithm first tries to get the "next" segment to satisfy the call by seeking the leftmost child of the current position in the database. This is done in Steps 1-2. If there is a leftmost child, then Steps 4-7 and Steps 25-40 are executed to process the call. Otherwise, the algorithm seeks the twins of the current position in the database to satisfy the call. This is performed in Steps 8-18. However, if there are no twins, then the brothers (segment types under the same parent segment type) are sought to satisfy the call. This is performed in Steps 19-20 and Steps 23-40. If there are no brothers, then the "uncle" of the current position is sought. This is performed in Steps 21-22.

A.4 Processing a Delete Call

The delete call is issued to delete the occurrence of a segment from the database. The segment to be deleted must first be obtained by issuing a get-hold call. Hence the segment to be deleted is always the current position in the database. The deletion of a segment has a side-effect if the segment is a parent. All segment occurrences beneath the parent are deleted as well.

The delete call can be processed quite easily by a DBC command call. Since the symbolic identifier of the parent appears in all of its dependent segments, a single DBC delete command with the symbolic identifier as the parameter will effectively delete the parent and all of its dependent segments.

ALGORITHM A: To process a delete call.

Step 1: Issue a Fetch-current call to the SBM to fetch the current segment of the segment number CPDB.
Step 2: Let K_1, K_2, ..., K_t be the keywords of the symbolic identifier extracted from the retrieved segment.

Step 3: Form the DBC query: $K_1 \wedge K_2 \wedge \ldots \wedge K_t$.
Step 4: Issue a delete-by-query command to the DBC using the query created in Step 3.
Step 5: Issue a Delete-current call to the SBM to mark the current segment of segment type CPDB as "deleted".
Step 6: For each j such that S_j is a dependent of CPDB, perform Steps 7-8.
Step 7: $V(S_j) \leftarrow o$.
Step 8: Call release-buffer-space(S_j).
Step 9: Return.

Notes: Steps 1-4 delete the segment and its dependent segments in the DBC. Step 5 deletes the segment in the ISB. Steps 6-8 delete its dependent segments in the ISB.

A.5 Processing a Replace Call

The segment to be replaced must first be obtained by a get-hold call. Hence the segment to be replaced is always the current position in the database. In describing the algorithm, we assume that the segment used in the replace call is stored in the storage location addressed by addr in the IM.

ALGORITHM A: To process a replace call.

Step 1: Issue a replace-by-pointer command to the DBC using the segment addressed by addr and its (DBC record)pointer as the parameters.
Step 2: Issue a replace-current call to the SBM to replace the current segment of segment type CPDB in the ISB by the segment addressed by addr in the IM.
Step 3: Return.

A.6 Processing an Insert Call

An insert call has the following format:

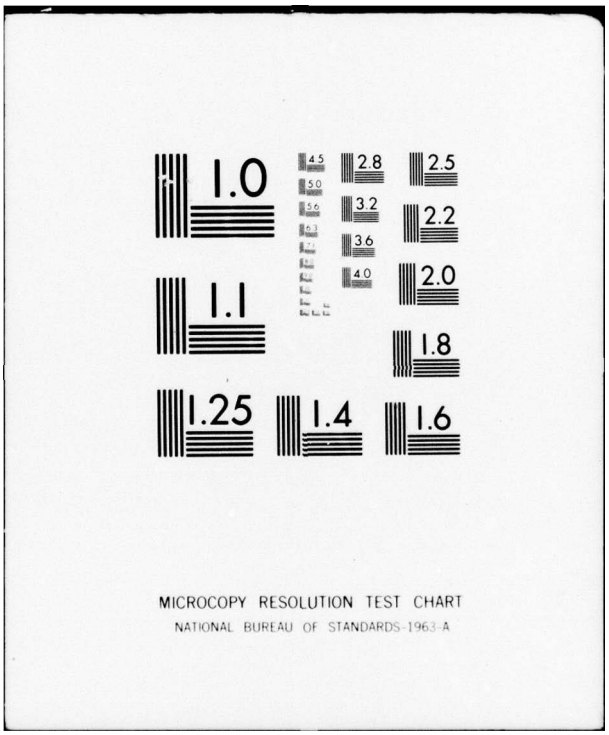
$$\text{ISRT} \quad [S_1, Q_1 \quad S_2, Q_2 \quad \vdots \quad S_{n-1}, Q_{n-1}] \quad S_n$$

where the last unqualified SSA specifies the segment to be inserted into the database. Hence in the above format, S_n represents the segment to be inserted. The specification of the SSAs above S_n is to position the data base for the insert call. Up to the level n-1, the SSA evaluation and positioning for the insert call is exactly the same as that for a get-unique call of the following format:

AD-A039 038 OHIO STATE UNIV COLUMBUS COMPUTER AND INFORMATION SC--ETC F/G 5/2
DBC SOFTWARE REQUIREMENTS FOR SUPPORTING HIERARCHICAL DATABASES--ETC(U)
APR 77 D K HSIAO, D S KERR, F K NG N00014-75-C-0573
UNCLASSIFIED OSU-CISRC-TR-77-1 NL

2 OF 2
AD A039038

END
DATE FILMED
5-77


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

GU S_1, Q_1
S_2, Q_2
:
S_{n-1}, Q_{n-1}

If the sequence S_1, S_2, ..., S_{n-1} is not specified at all, then the current position in the database is used to determine the hierarchical path for the insert call. Since the positioning of the database for an insert call is the same as that for a get-unique call, the algorithms for positioning the database for an insert call will not be given in this section. The reader is referred to Section A.1 for the algorithms.

Once the positioning for an insert call has been set, the actual insertion operation can be performed in a fairly simple manner. A DBC record is created for the given segment using the rules given in Section 4.1. Then the mandatory clustering condition (MCC) for physical placement of the DBC record is created (in our discussion, we use the second clustering policy given in Section 4.2). The DBC record is then inserted into the MM by a load-record (LR) or an insert-record (IR) command, depending on whether the current operation is creating or updating the file. Notice that we use only the IR command in the algorithm for simplicity.

ALGORITHM A: To process an insert call.

Step 1: If the sequence S_1, S_2, ..., S_{n-1} is not specified, then go to Step 4.

Step 2: Execute the get unique algorithm using (S_1, Q_1), (S_2, Q_2), ..., (S_{n-1}, Q_{n-1}).

Step 3: If a 'not-found' condition is returned from the get-unique algorithm then return 'not-found'.

Step 4: If $V(S_{n-1}) = o$, then return 'not-found'.

Step 5: Create a DBC record by performing Steps 6-11.

Step 6: For each field in the segment which will be used as a search argument in a DL/1 call, form a type-N keyword using the field name as the attribute of the keyword and the field value as the value of the keyword.

Step 7: For each field in the segment which will not be used as a search argument in a DL/1 call, form a nonkeyword attribute-value pair using the field name as the attribute and field value as the value of the attribute-value pair.

Step 8: Form a type-D keyword of the form (TYPE, segtype) where segtype is the segment type of the segment to be inserted. Designate this keyword as a clustering keyword.

Step 9: If S_n is the root segment, then change the keyword formed in Step 6 for the sequence field of the root segment to a type-D keyword and designate it as a clustering keyword. Skip the next two steps.

Step 10: Issue a Fetch-current call to the SBM to fetch the current segment of the segment number S_{n-1}.

Step 11: Let K_1, K_2, ..., K_t be the keywords of the symbolic identifier of the segment retrieved in Step 10. Use these keywords as the keywords for the segment.

Step 12: If S_n is the root segment, then form the MCC: $(TYPE = S_n)$, otherwise form the MCC: $(K_1 \wedge K^1) \vee (K_2 \wedge K^1) \vee \ldots \vee (K_t \wedge K^1)$ where K^1 is the keyword representing the sequence field of the root segment and K_1, K_2, ..., K_t are the keywords representing all the dependent segment types.

Step 13: Issue an insert-record command using the DBC record created in Step 4 and the MCC created in Step 12.

Step 14: Issue an insert-as-current call to the SBM to insert the DBC record created in Step 5 into the ISB.

Step 15: $CP(S_n) \leftarrow 1$.

Step 16: $COUNT \leftarrow 1$.

Step 17: $V(S_n) \leftarrow 1$.

Step 18: $QUAL(S_n) \leftarrow$ sequence key of the inserted segment.

Step 19: Return.

Notes: Step 2 positions the database for the insert call. Steps 5-11 create the DBC record by forming the appropriate keywords. The keywords representing the sequence field of the root segment and all the segment types are designated as type-D and clustering keywords. The rest of the keywords are designated as type-N keyword. Step 12 forms the appropriate MCC for physical placement. Steps 14-18 update the ISB and the SIT.

APPENDIX B - THE ALGORITHMS OF THE SYSTEM BUFFER MANAGER (SBM)

The SBM executes the subroutine calls from the DL/1 interface module (IM). These calls are executed according to the buffer management policy discussed in Section 6.1 and using the data structures given in Section 6.2. The description of the function of each of the calls was given in Appendix A. The function names are listed as follows.

(1) Load-buffer.

(2) Fetch-current.

(3) Replace-current.

(4) Delete-current.

(5) Insert-as-current.

(6) Release-buffer-space.

B.1. The Load-buffer Call.

ALGORITHM A: To execute the Load-buffer call.

Input arguments: 1. A segment type k.

2. A DBC command identifier, C-ID.

Notations: Let m be the number of DBC records retrieved.

Step 1: Wait until the response set identified by C-ID is ready for transmission from the DBC.
Step 2: If response set is empty, then return "empty".
Step 3: Call Load(k). [see Algorithm B]
Step 4: COUNT(k) $\leftarrow$ m.
Step 5: Return.

Note: Step 4 sets the COUNT field of the status information table (SIT) to the total number of DBC records retrieved.

ALGORITHM B: The Load Algorithm.

Input argument: A segment type k.

Notations: 1. Let m be the number of DBC records retrieved.
2. Let n be the total number of segment types for the IMS database.
3. Let N be the size of the ISB (in pages).

Step 1: Calculate the number of pages needed to store the DBC records using m and LENGTH(k). Let this number be t.
Step 2: If $\sum_{i=1}^{n}$ TOTAL(i) + t > N, then go to step 5.
Step 3: Call Load-record(k,t,o,). [see algorithm C]
Step 4: Return.
Step 5: Call Garbage-collection. [see algorithm D]
Step 6: If $\sum_{i=1}^{n}$ TOTAL(i) + t $\leq$ N, then go to step 3.
Step 7: i $\leftarrow$ 1.

Step 8: $y \leftarrow TOTAL(i) - QUOTA(i)$.
Step 9: If $y \leq 0$, then go to step 14.
Step 10: If $\sum_{i=1}^{n} TOTAL(i) + t - N \geq y$, then go to step 13.
Step 11: Call Deallocation (i, ΣTOTAL(i) + t - N). [see algorithm E]
Step 12: Go to step 3.
Step 13: Call Deallocation(i,y).
Step 14: $i \leftarrow i + 1$.
Step 15: If $i \leq n$, then go to step 8.
Step 16: Call Load-record (k,QUOTA(k),1).
Step 17: Return.

Note: Step 3 is executed if the ISB has enough space for the response set. Otherwise, step 5 is executed for garbage collection. If the ISB still has not enough space after garbage collection, then steps 7-15 are executed for deallocation of ISB space. The segment types on the higher levels will be deallocated first. Step 8 calculates how much space occupied by the segment type should be returned to the common pool. When step 16 is executed, it indicates the only space available is the space allocated to its quota.

ALGORITHM C: The Load-record Algorithm.

Input argument: 1. A segment type k.
2. The number of pages x to be loaded.
3. An indicator d.

Step 1: $i \leftarrow 1$.
Step 2: Find a page y using the ISB page map and set the corresponding bit to zero.
Step 3: Transfer DBC records to the page y.
Step 4: Make adjustment to the pointer fields in the ISB page table and the segment control table to reflect the allocation of this page to the segment type k.
Step 5: $i \leftarrow i + 1$.
Step 6: If $i \leq x$, then go to step 2.
Step 7: If d=1, then enter the value of the sequence field of the last segment stored into SEQ(k).
Step 8: TOTAL(k) $\leftarrow$ x.
Step 9: UPAGE(k) $\leftarrow$ o.
Step 11: CPAGE(k) $\leftarrow$ LHEAD(k).
Step 12: OFFSET(k) $\leftarrow$ o.
Step 13: C $\leftarrow$ d.
Step 14: Return.

Notes: This algorithm stores the DBC records into x pages of the ISB. DBC records are stored in a specified order. If all DBC records cannot be stored in x pages, the DBC records with largest sequence field values will not be stored.

ALGORITHM D: The Garbage-collection Algorithm.

Notations: Let n be the total number of segment types.

Step 1: $i \leftarrow 1$.
Step 2: If UPAGE(i) = 0, then go to step 10.
Step 3: $j \leftarrow 1$.

Step 4: m ← LHEAD(i).
Step 5: Set the m^{th} bit in ISB bit map to 1.
Step 6: j ← j + 1.
Step 7: If j < UPAGE(i), then LHEAD(i) ← FPTR(m) and go to step 10.
Step 8: m ← FPTR(m)
Step 9: Go to step 5.
Step 10: i ← i + 1.
Step 11: If i ≤ n, then go to step 2.
Step 12: Return.

Notes: This algorithm returns the pages occupied by any segment type which have been processed to the common pool.

ALGORITHM E: The Deallocation Algorithm.

Input arguments: 1. A segment type k.
2. The number of pages x to be returned to the common pool.

Step 1: i ← 1.
Step 2: m ← LTAIL(k).
Step 3: Set the m^{th} bit in the ISB page table to 1.
Step 4: i ← i + 1.
Step 5: If i > x, then LTAIL (k) ← BPTR(m) and return.
Step 6: m ← BPTR(m).
Step 7: Go to step 3.

Notes: This algorithm returns the last x pages of the segment type k to the common pool.

B.2. The Fetch-current Call.

ALGORITHM A: To execute the Fetch-current call.

Input arguments: 1. A segment type k.
2. An address, addr, to which the retrieved DBC record is transferred.

Step 1: If OCS(k) ≠ CS(k), then go to step 4.
Step 2: Retrieve the DBC record in the ISB addressed by (CPAGE(k), OFFSET(k)).
Step 3: If "deleted" is indicated on the DBC record, then return "deleted" else return the DBC record to addr.
Step 4: OCS(k) ← CS(k).
Step 5: OFFSET(k) ← OFFSET(k) + LENGTH(k).
Step 6: If OFFSET(k) > page size, then go to step 9.
Step 7: Retrieve the DBC record in the ISB addressed by (CPAGE(k), OFFSET(k)).
Step 8: If the retrieved record indicates the end-record, then return "error" else return the DBC record to addr.
Step 9: CPAGE ← FPTR(CPAGE).
Step 10: If CPAGE = "Null", then go to step 14.
Step 11: OFFSET ← 0.
Step 12: UPAGE ← UPAGE + 1.
Step 13: Go to step 7.
Step 14: If C(k) = 0, then return "error".
Step 15: Form a DBC query: (TYPE = i) ∧ (ATT > SEQ) ∧ QUAL(k), where ATT is the field name of the sequence field.

Step 16: Issue a DBC retrieve-by-query-with-pointer command using the query created in step 15.
Step 17: If the response set is empty, then return "error".
Step 18: Call Load-buffer(k,C-ID) [algorithm B in Section B.1] where C-ID is the DBC command identifier used in step 16.
Step 19: OFFSET(k) ← OFFSET(k) + LENGTH(k).
Step 20: Retrieve the DBC record in the ISB addressed by (CPAGE(k), OFFSET(k)).
Step 21: Return the DBC record to addr.

Notes: Steps 2-3 are executed if the segment last retrieved by the SBM is the same as the current segment. Otherwise steps 5-13 are executed to retrieve the next DBC record with respect to the one addressed by (CPAGE(k), OFFSET(k)). If the DBC record addressed by (CPAGE(k), OFFSET(k)) is the last DBC record stored in the ISB, then steps 15-21 are executed to fetch more DBC records from the DBC.

B.3. The Replace-current Call.

ALGORITHM A: To execute the Replace-current call.

Input arguments: 1. A segment type k.
2. An address, addr, to a DBC record used to replace the one stored in the ISB.

Step 1: Store the DBC record addressed by addr into the ISB addressed by (CPAGE(k), OFFSET(k)).
Step 2: Return.

B.4. The Delete-current Call.

ALGORITHM A: To execute the Delete-current call.

Input argument: A segment type k.

Step 1: Mark the DBC record in the ISB addressed by (CPAGE(k),OFFSET(k)) as "deleted".
Step 2: Return.

B.5. The Insert-as-current Call.

ALGORITHM A: To execute the Insert-as-current call.

Input arguments: 1. A segment type k.
2. An address, addr, to a DBC record to be inserted into the ISB.

Step 1: Call Deallocation(k, TOTAL(k)).
Step 2: Find a page x using the ISB bit map and set the corresponding bit to zero.
Step 3: Store the DBC record R addressed by addr into page x starting from location 0.
Step 4: Store an "end-record" after R in page x.
Step 5: D(k) ← C(k) ← UPAGE(k) ← OFFSET(k) ← 0.
Step 6: LHEAD(k) ← LTAIL(k) ← CPAGE(k) ← x.

Step 7: BPTR(x) ← FPTR(x) ← "null".
Step 8: TOTAL(k) ← 1.
Step 9: Return.

Notes: This algorithm releases all pages allocated to the segment type and allocates a new page for the DBC record to be inserted. The page only contains this DBC record. All the fields in the segment control table entry are set appropriately in steps 5-8.

B.6. The Release-buffer-space Call.

ALGORITHM A: To execute the Release-buffer-space Call.

Input argument: A type code k.

Step 1: Call Deallocation(k, TOTAL (k)).
Step 2: D(k) ← 1.
Step 3: Return.

Notes: This algorithm releases all pages allocated to the segment type k. The D field is set to 1 to indicate the segment control table entry is deactivated.

APPENDIX C - A DISCUSSION OF HIDAM

There are two database organizations in IMS: Hierarchical Sequential (HS) and Hierarchical Direct (HD). Each of these two physical organizations is supported by two database access methods: HSAM and HISAM for HS organization and HDAM and HIDAM for HD organization. We will restrict our attention to Hierarchical Indexed Direct Access Method (HIDAM).

HIDAM is used for indexed access to the root segments of a hierarchical database. The index is stored in an index area and no user data segments exist within this area. The user data segments are stored in a separate area. Fig. 35 illustrates a possible HIDAM physical storage of the IMS database of Fig. 6.

When a HIDAM database is created, the user presents the segments of each database record to IMS in proper sequence. The data area is then used in a purely sequential order to load all segments presented. As each root segment is presented, the system automatically creates an indexing segment and places it in the index area. Notice that because of the allocation of blocks in sequential order, the logical adjacency of segments is often reflected by physical adjacency as well (although pointers have to be maintained, nevertheless).

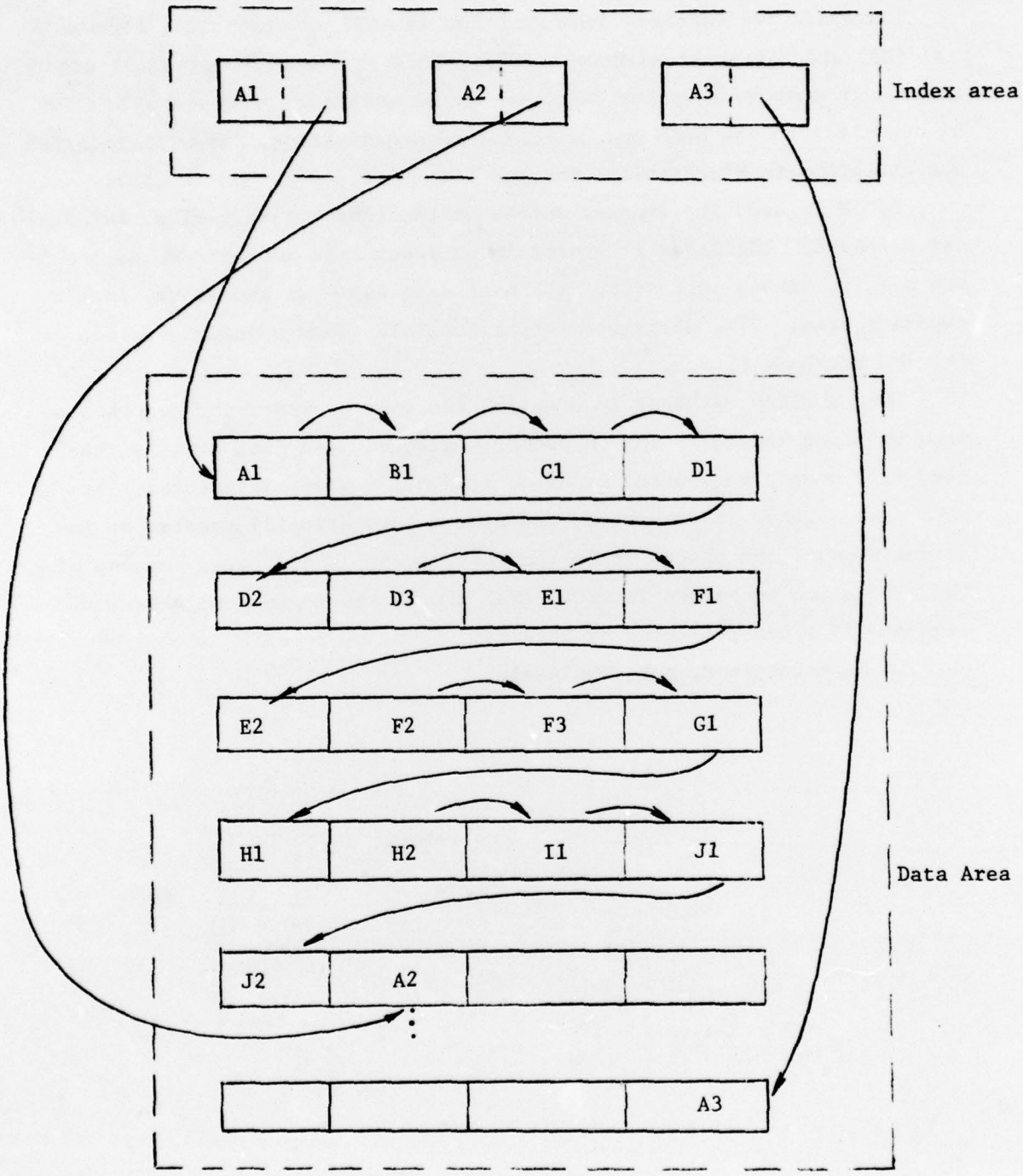


Fig. 35 HIDAM physical storage of the database of Fig. 6.