MASTER IN COMPUTER SCIENCE

Design and implementation of a database management system based on the entity-relationship model

Tollenaere, Patrice

Award date:
1983

Awarding institution:
University of Namur

DESIGN AND IMPLEMENTATON OF A DATABASE MANAGEMENT SYSTEM

BASED ON THE ENTITY-RELATIONSHIP MODEL

PATRICE TOLLENAERE

Thesis presented in order to obtain
the Master's degree in Computer Science

Academic year 1982-1983

I now wish to acknowledge those people who supported me in this project. First I would like to thank Professor F. Bodart of the University of Namur who served as my mentor and thesis sponsor. His assistance was of great help to me for the completion of this thesis.

I am also grateful to Professor D. Teichroew of the University of Michigan who gave me the opportunity to work in the ISDOS project at that university. To Professor K. Kang of the University of Michigan who offered me so much support during my term in Michigan, I also extend my gratitude.

Final thanks go to the ISDOS project team in Namur. Their technical knowledge of the ISDOS project was of great service to me.

CONTENTS

0 INTRODUCTION.....1

1 INTRODUCTION TO THE ISDOS SOFTWARE

1.1 Introduction.....3

1.2 Life Cycle Support System.....3

1.2.1 Introduction.....3

1.2.2 Life Cycle.....3

1.2.3 Life Cycle Support System (LSS) Definition.....5

1.2.4 The LSS Generator.....5

1.3 Architecture of the ISDOS Software.....6

1.4 Information System Language Description Manager (ISLDM)....8

1.4.1 ISLDM Language.....8

1.4.2 ISLDM System.....8

1.5 The System Encyclopedia Manager (SEM).....10

1.5.1 Command Language Interface (CLI).....11

1.5.2 Language Processors: RP, IP, DP, and the Delete-Object Processor.....11

1.5.3 The Query System and the Name-Selection Processor.....11

1.5.4 Analysis and Display Processor.....12

1.5.5 The Database Interface.....13

1.6 ADBMS: A Data Base Management System for the ISDOS Software.....14

1.6.1 The Data Description Language: DDL.....14

1.6.2 ADBMS Overview.....16

1.6.2.1 The Database Tables (DBT).....16

1.6.2.2 The Database File (DBF).....17

1.6.2.3 The Database Control System (DBCS).....17

1.6.2.4 ADBMS Utility Programs.....18

2 OVERVIEW OF SEM PERFORMANCE

2.1 Introduction.....20

2.2 Performance of the IP and RP Processors.....20

2.2.1 Performance of the IP Processors.....20

2.2.2 Performance of the RP Processor.....23

2.3 ADBMS Performance.....24

2.3.1 The DDL for a SEM Data Base.....24

2.3.2 Implementation Evaluation.....27

2.3.3 Implementation of the Codaysl Model: a Synthesis.....29

2.3.3.1 Location Mode.....29

2.3.3.2 Set Mode.....29

2.3.3.3 Effect of Storage Structure on Processing Efficiency...33

2.3.4 Directions for Improvement.....35

3 LOGICAL VIEW OF AN ENTITY-RELATIONSHIP DATABASE MANAGEMENT SYSTEM (DBMS)

3.1 Introduction.....37

3.2 The Entity-Relationship Data Model.....37

3.3 An E-R Data Definition Language (DDL).....41

3.3.1 Introduction.....41

3.3.2 Basic Constructs.....41

3.3.3 Description Content.....44

3.3.3.1 Notation.....44

3.3.3.2 Special Meanings.....44

3.3.3.3 Entity Type Section.....46

3.3.3.4 Relationship Section.....48

3.4 A Data Manipulation Language.....49

3.4.1 Introduction.....49

3.4.2 Subroutines Description of the DBUSER Level.....49

4.5 DBMS Overview.....85

4.5.1 DBMS Structure Overview.....85

4.5.1.1 The Database Tables (DBT).....85

4.5.1.2 The Database Files (DBF).....85

4.5.1.3 The Database Control System (DBCS).....86

4.5.1.4 DBMS Utility Programs.....87

4.5.2 DBMS Structure.....89

4.5.2.1 The Database Tables.....89

4.5.2.2 The Database Entity Type Name Table (ETNTAB).....91

4.5.2.2.1 Entity Name Description Block (IEB).....91

4.5.2.3 The Database Relationship Type Name Table (RTNTAB).....92

4.5.2.3.1 Relationship Description Block (IRB).....92

4.5.2.4 The Database Type Description Table (TYTAB).....92

4.5.2.4.1 Primary Key Description Block (IPKB).....93

4.5.2.4.2 Attribute Description Block (IADB).....94

4.5.2.4.3 Value Set Description Block (IVSB).....95

4.5.2.4.4 Allowable Value Description Block (IIRB, ICHB).....98

4.5.2.4.5 Secondary Key Description Block (ISKB).....98

4.5.2.4.6 Attribute Part Description Block (IAPB).....99

4.5.2.4.7 Relationship Description Block (IREB).....99

4.5.2.4.8 Entity Related Description Block (IERB).....100

4.5.2.4.9 Character Array NAMES.....101

4.5.2.4.10 DBT Structure as Written by DDLA.....102

4.5.2.4.11 The Object Schema Parameters.....102

4.5.2.4.12 NAMES.....102

4.5.2.5 Dynamic Hashing Tables.....103

4.5.2.5.1 The Database Hashing Table for the Primary Key (HPKTAB).....103

4.5.2.5.1.1 Primary Key Hashing Description Block (IHPB).....103

4.5.2.5.2 The Database Hashing Table for the Secondary Key and the Relationship (HATTAB).....104

4 IMPLEMENTION OF AN E-R MODEL

4.1 Introduction.....58

4.2 Presentation of the Problem.....58

4.3 Overview of New Hashing Schemes.....60

4.3.1 Introduction.....60

4.3.2 Conventional Hashing Files.....60

4.3.3 Linear Hashing With Partial Expansions [Larson 1980].....63

4.3.3.1 Introduction.....63

4.3.3.2 Linear Hashing Scheme.....64

4.3.3.3 Improvement Proposed by [Larson 1980]:
Linear Hashing With Partial Expansions.....67

4.3.3.3.1 Introduction.....67

4.3.3.3.2 Presentation of Linear Hashing With Partial
Expansion.....67

4.3.3.3.3 Control Function.....71

4.3.3.3.4 Performance.....71

4.3.4 Dynamic Hashing Scheme for Secondary Key File
[Lloyd, Ramamohanorao and Thom 1983].....72

4.3.4.1 Introduction.....72

4.3.4.2 Definition of a Partial Match Query.....72

4.3.4.3 Definition of a Simple Partial-Match Retrieval Scheme
(When the File is Known) Based Purely on Hashing.....72

4.3.4.4 A Descriptor Scheme.....74

4.3.4.4.1 Descriptor, Page Descriptor, File Descriptor.....74

4.3.4.4.2 Constructing a Descriptor.....74

4.3.4.4.3 Using a Descriptor File.....76

4.3.4.5 Extension of the Scheme to Dynamic Files.....78

4.3.4.5.1 Presentation of the Scheme.....78

4.3.4.5.2 Choice Vector.....79

4.3.4.5.3 File Descriptor.....81

4.3.4.6 Performance.....81

4.4 How to Use These Schemes to Store Entities and
Relationships.....83

4.5.2.5.2.1 Attribute Hashing Description Block (IHSB).....104
4.5.2.5.3 Optimal Parameters Table (OPATAB).....105
4.5.2.5.3.1 The Optimal Header Description Block (IOHB).....105
4.5.2.5.3.2 The Optimal Parameter Description Block (IOPB).....106
4.5.2.5.3.3 The Choice Vector Description Block (IOAB).....107
4.5.2.6 File Organization.....107
4.5.2.7 Database Control Block.....110
4.5.2.7.1 Identification of a Page.....110
4.5.2.7.2 Data Area.....110
4.5.2.7.3 Storage Allocation.....111
4.5.2.7.4 DBCS Page Management System.....111
4.5.2.7.4.1 The DBCS Random I/O Routines (DBHRAN).....112
4.5.2.7.4.2 The Page Buffer (BUFPAG).....112
4.5.2.7.4.3 The Current Page.....112
4.5.2.7.4.4 Reading a New Page from the DBF into Main Memory...113
4.5.2.7.5 The Database Control System (DBCS).....114
4.5.3 Implementation Evaluation.....115
5 CONCLUSION.....117
6 BIBLIOGRAPHY.....118
Annex 1 Overview of New Hashing Schemes
Annex 2 Description of the Routines

INTRODUCTION

During the last ten years the demand for computer based information processing systems has grown steadily. The main reason for this is that the crisis in software development becomes increasingly important. Indeed, at the same time as the cost of hardware has dropped, the cost of software has risen dramatically. The major reason is that in the majority of cases, the software development stayed at an artisanal stage.

The main difficulties of such a development were :

- the lack of uniformity in writing the specifications and documentations of applications
- impossibility to update documentation manuals
- difficulties in consultations of these manuals
- lack of control over the specifications

To remedy these problems, a number of tools have been designed which assist in software design, construction, operation and maintenence.

Among these, the ISDOS project has been developed since 1969 at the University of Michigan under the supervision of Professor Teichroew.

The first chapter of this thesis will be dedicated to a brief presentation of the tools proposed by the ISDOS project.

As the evaluation and amelioration of any software system as large as the ISDOS project is of prime importance to anyone involved with the system, we outline in chapter two some levels of software and associated data structures which are critical for the performance. Specifically, the the performances of the database management system provided to handle datas are reviewed.

The different criticisms discussed in chapter two have focused attention on the need to change the current database management system. Among the several possible directions we have chosen one which employs the Entity-Relationship (E-R) model as a data model and which directly implements a database management system based on this model. Some aspects of the logical view of an E-R database management system are reviewed in chapter three.

As a database management system is a complex and huge task, we have limited our reflection only to new methods of direct access of the records for the database. These methods can at times be very efficient, but can not be the only support of a database management system. These methods and the physical structure of the database management system are proposed in chapter four.

Chapter 1 : INTRODUCTION TO THE ISDOS SOFTWARE

1. INTRODUCTION TO THE ISDOS SOFTWARE

1.1. Introduction

In the first section are discussed the different concepts and goals of the ISDOS software. In particular, the notions of life cycle support system (LSS), LSS generator and LSS processor will be explained. The second section is concerned with the LSS generator of the ISDOS project (ISLDM) and the third section introduces the generalized software (SEM) proposed by the ISDOS project to implement the LSS processor. Finally the fourth section presents the data base management system used to manage the data in the ISDOS software.

1.2. Life Cycle Support System

1.2.1. Introduction

In the early 70's a great number of medium and large organizations created a new organizational unit called "systems department". This new department was established to handle the information processing systems (IPS) inside the organizations.

An IPS was defined by [Yamamoto 1981] as "the subsystem of the information system in which data is recorded and processed following a formal procedure". Two kinds of IPS may be distinguished : manual and computer based.

Manual are those in which all operations are performed manually while computer based are those in which some operations are performed by a computer.

The creation of the systems department allowed a more centralized and controlled environment for the software development and operation. The management aspect of such development was carefully studied and the concept of "life cycle" results from such studies.

1.2.2. Life Cycle

The concept of "life cycle" is a management concept for controlling the process of software development and operation.

There exist many guidebooks for managing software projects based on the life cycle concept [Metzger 1973], [Biggs, Birks and Atkins 1980].

A software life cycle is divided into distinct phases. The exact breakdown into phases and the terms used to represent each phase differ from author to author.

The phases shown in fig 1.1 were defined in [Teichroew,Hershey and Yamamoto 1977]

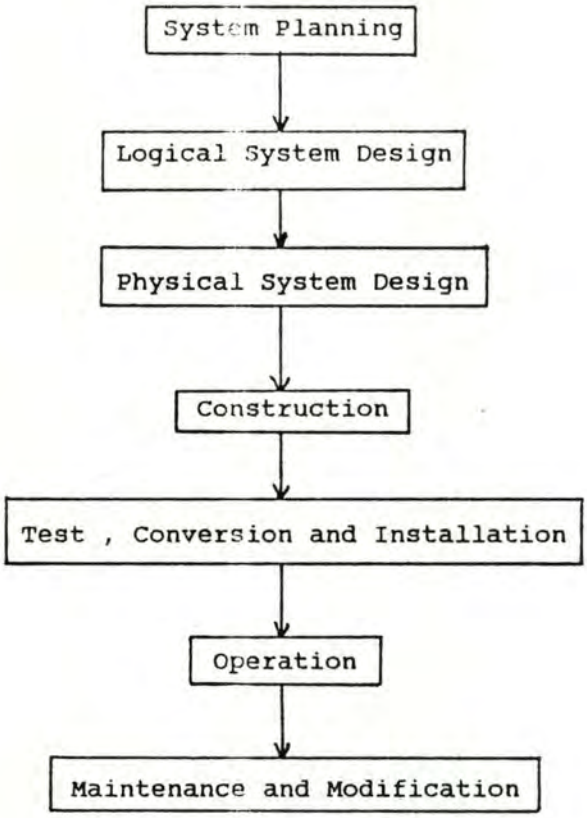


Fig 1.1 Software Development Phases

The output of a life cycle phase is often called documentation or system description. At each phase, the verification of completeness, consistency and unambiguousness of system description is an important point because the system description of one phase serves as the input to the next phase of the life cycle.

Tools to support the activities involved in the software life cycle were created. Surveys on the different tools proposed can be found in [Reifer 1975], [Reifer and Trattner 1977] and [Teichroew, Hersey and Yamamoto 1977].

Among these tools, one called "life cycle support system "(LSS) has been developed lately [Teichroew, Hershey and Yamamoto 1977].

1.2.3. Life Cycle Support System (LSS) Definition

The definition given by [Yamamoto 1981] of a life cycle support system is the following :

> "A life cycle support system is a class of computer based tools for software development that supports the applications in the systems department in one or more phases of the life cycle. "

The functions of a LSS are :

- accept system description in some predefined notation
- maintain a data base containing the system description
- perform analyses on the system description
- produce documentation and reports based on the system description
- perform control functions of the life cycle activities

To accomplish these functions, a LSS must have a tool system. The tool system consists of :

- the data base that stores the system description that has been described (the LSS database)
- the system description language that is used to express relevant information about an IPS (the LSS language)
- the processor that takes the system description expressed in the LSS language and updates the data base. It also performs analysis and produces reports based on information in the data base (the LSS processor).

1.2.4. The LSS Generator

To develop LSS's a specialized computer aided tool has been built : the LSS generator.

The LSS generator allows the LSS developer to specify a target LSS in a formal language and automatically generate as much of the LSS software and documentation as possible.

The LSS generator needs a model of IPS models that is applicable to a wide variety of conceivable LSS. This model is usually called the Meta-Model. Once the Meta model has been specified, a targetLSS model may be specified as a specific instance of the Meta-Model. In the next section we will explain the structure of the LSS generator proposed by the ISDOS project.

1.3. Architecture of the ISDOS Software

The ISDOS software has two important components : the Information System Language Description Manager (ISLDM) and the System Encyclopedia Manager (SEM). Figure 1.2 shows the general structure of the ISDOS software.

ISLDM is the name of the ISDOS's LSS generator processor. ISLDM is composed of the ISLDM language and the ISLDM system. The ISLDM system has four components :

1. the language processor
2. the documentation producer
3. the global analyzer
4. the table producer

SEM is the generalized software provided by the ISDOS project to implement the LSS processor. The main components of SEM are :

1. the Command Language Interface (CLI)
2. the language processors : IP, RP, DP and the Delete-Object processor
3. the extract processor : the Query system and the Name-Selection processor
4. the analysis and display processor which consists of three types of reports : FS, STR and EP reports
5. a database interface composed of a Data Management Interface (DMI) and a META table interface (MTI)

A database management system called ADBMS is provided to handle the data in the databases of the system.

The next sections describe each of these components.

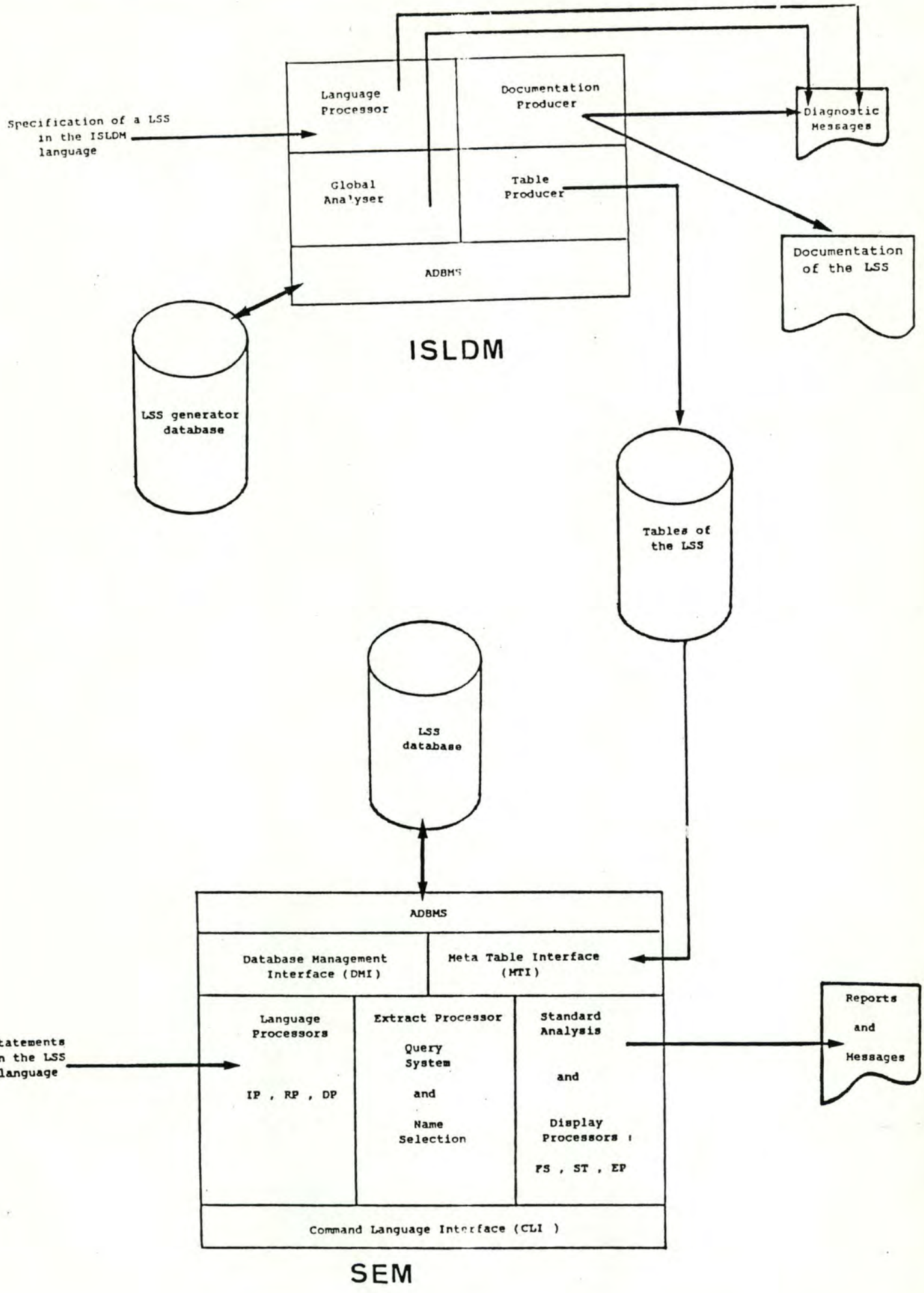


Fig 1.2 General Structure of the ISDOS Software

1.4. Information System Language Description Manager (ISLDM)

ISLDM is the name of the ISDOS's LSS generator processor. ISLDM is composed of the ISLDM language and the ISLDM system.

1.4.1. ISLDM Language

The ISLDM language (or META language) is a language for describing a formal specification of a target LSS. The META language can express information about an IPS according to a model called the META model. It is based on the observation that there exists a sufficient "commonality" with a wide range of LSS to derive a model.

The Meta model chosen by the ISDOS project is the Entity-Relationship (E-R) model proposed by [Chen 1976]. The names employed in the META model are different from those presented by CHEN. CHEN 's model will be briefly reviewed in chapter three, but for more information the reader is suggested to refer to the existing literature [Chen 1976], [Chen 1977], [Sakai 1980].

The main concepts of the E-R model are those of object, relationship and property. An object is an atomic element of the universe of the discourse. A property is a characteristic of an object or a relationship. A relationship represents a connection among one or more objects.

The choice of the E-R model was mainly motivated by the fact that the E-R representation of the world is quite natural and close to the LSS user's view.

1.4.2. ISLDM System

ISLDM system accomplishes the functions of language processing, global analysis, documentation generation. This system is composed of four main components :

1. the language processor
2. the global analyser
3. the documentation producer
4. the table producer

The person who specifies in the ISLDM language the components of a target LSS may enter the statements in any order and change the definition of the LSS as many times as desired.

The statements are processed by the language processor which can detect any syntax error and can produce

diagnostic messages. The global analyser detects inconsistencies of a more global nature.

The documentation producer allows documentation generation for the LSS's users once the LSS's developer is satisfied with the LSS definition.

The definition of the LSS is stored in a LSS generator database through a database management system called ADBMS.

The LSS processor is generated according to the informations contained in the database. The table generator produces tables containing the definition of the LSS. These tables are used by the generalized software to produce the LSS processor.

Figure 1.3 shows the structure of the ISLDM language and system

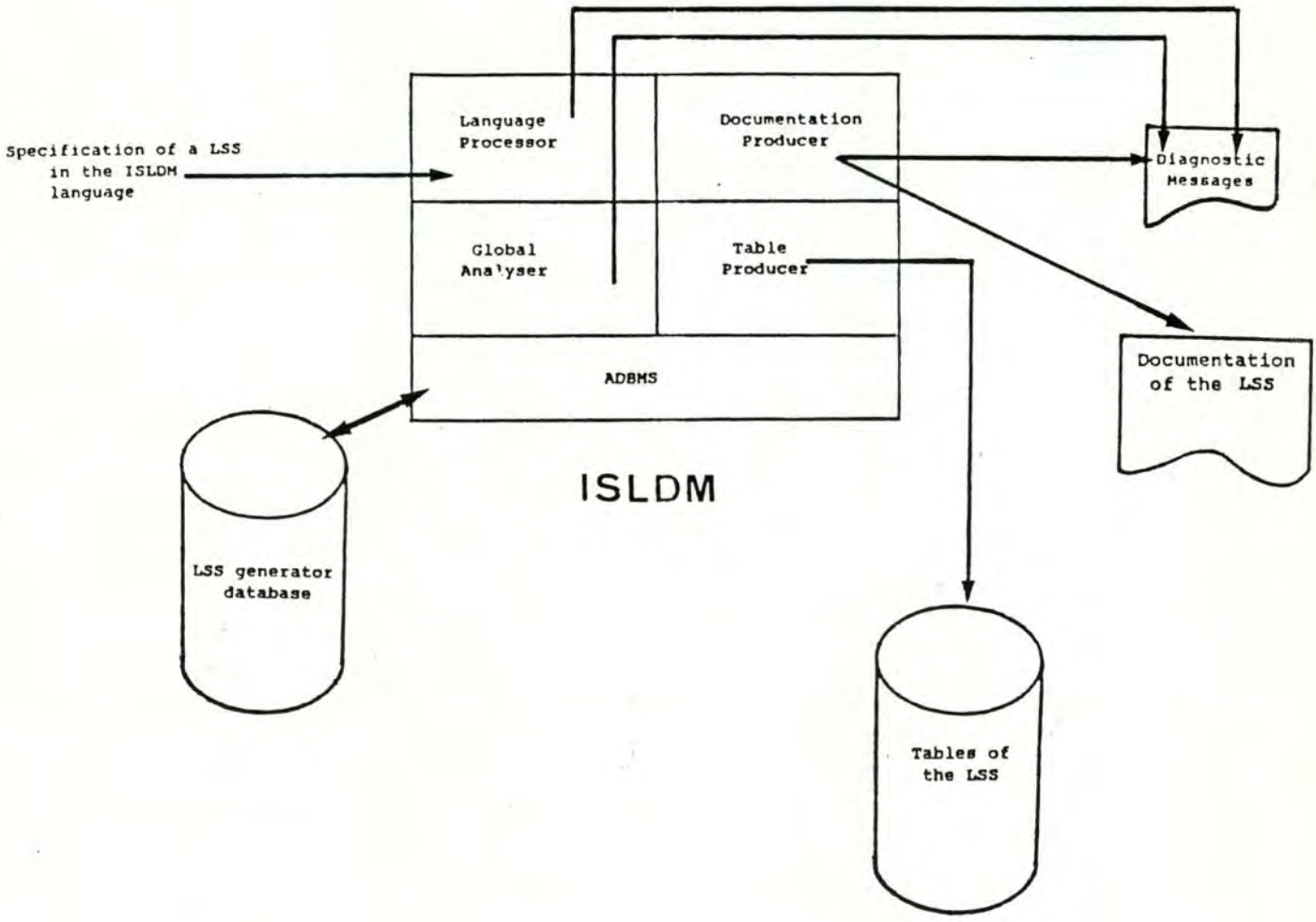


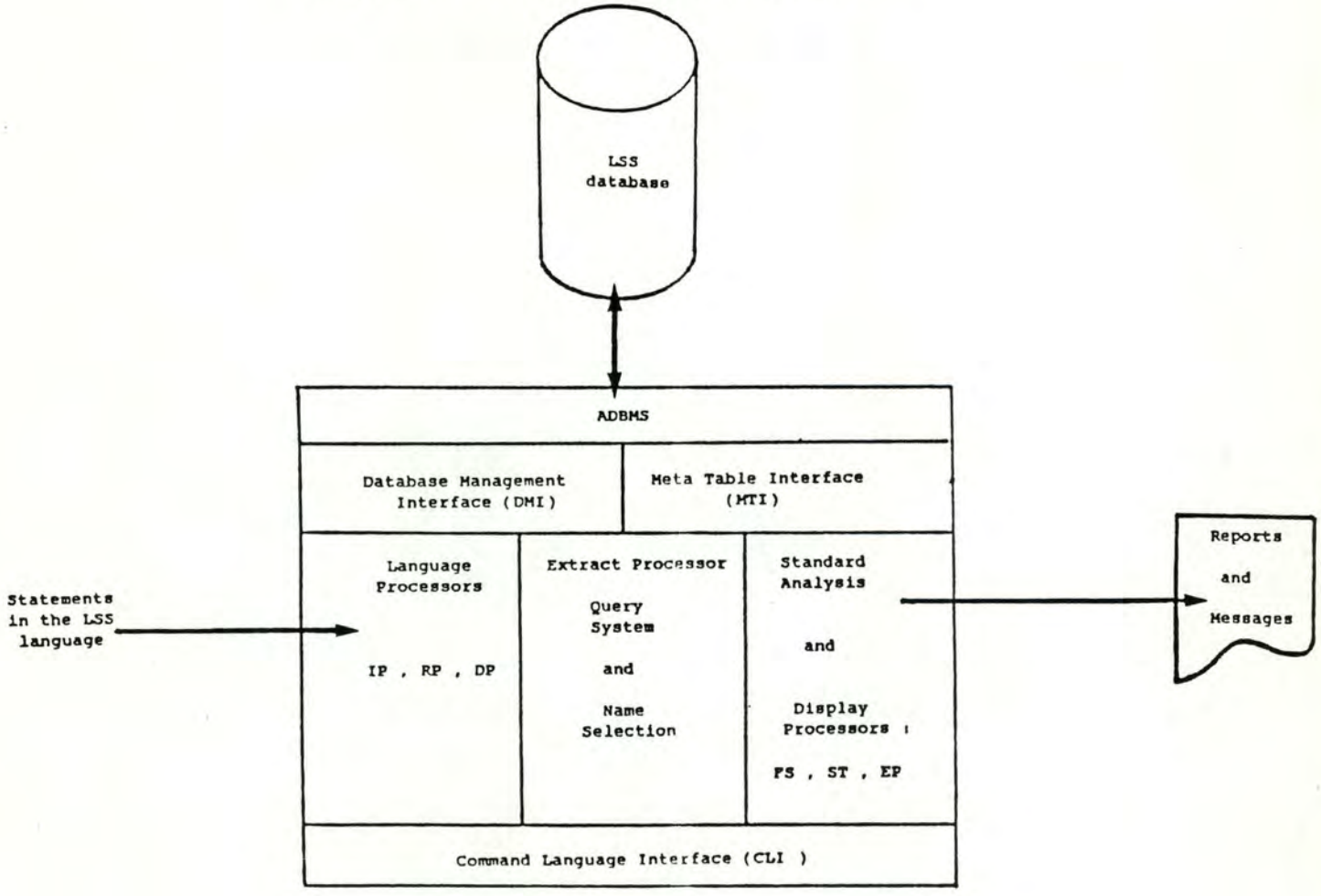
Fig 1.3 ISLDM system

1.5. The System Encyclopedia Manager (SEM)

SEM is the generalized software provided by the ISDOS project to implement the LSS processor. The main components of SEM are :

1. the Command Language Interface (CLI)
2. the language processors : IP,RP,DP and the Delete-Object processor
3. the extract processor : the Query system and the Name-selection processors
4. the analysis and display processor which consists of three types of reports : FS, STR and EP reports
5. a Database Management Interface (DMI) and a Meta Table Interface (MTI)

Figure 1.4 shows the structure of SEM.



SEM

Fig 1.4 SEM system

We are now seeing all these components in detail

1.5.1. Command Language Interface (CLI)

Thanks to the CLI, the LSS's user can have access to the functions of the LSS processor.

For a specific LSS, a set of LSS processor "macros" are defined by the ISLDM system that invoke an appropriate sequence of programs with necesary parameters.

Like this, the LSS processor can be used interactively by interpreting commands in the LSS command language

1.5.2. Language Processors : RP,IP,DP and the Delete-Object Processor

Language processors are used to update the SEM database. They take SEM language statements as input. Checks are made on these statements to preserve the integrity of the database. Some error messages are produced if an incorrect statement is detected by the processor.

There exist three levels of checking on the statements. First, a syntactical checking is made on the statements. A second checking is made on a syntactically correct statement to know if the statement is allowed or not in the current context. The third level of checking is a consistency check against what is already in the data base.

In SEM there are three language processors :

1. The Input Processor (IP)

2. The Delete Processor (DP)

3. The Replace Processor (RP).

IP adds the information specified in the input to the data base. DP deletes the specified information from the database. RP replaces the information in the database with the information supplied.

Objects are never deleted from the database by the language processors because instances of objects must be deleted together with all connections and properties. The Delete-Object processor checks the existence of an object and deletes from the data base.

1.5.3. The Query System and the Name-Selection Processor

Through the extract processors, Query system and Name-Selection, the LSS's user can select pieces of information from the database that meet certain user specified criteria.

The Name-Selection (NS) selects names of the objects in

the database according some selection criterion.

The major purpose of the Query system is to answer various inquiries on the description of an IPS stored in the database. The main features of the Query system are :

- the selection criterion can include any legal object type and any legal statement
- selection criteria can be named and stored for later use
- the set name which satisfy a criterion can be named and stored
- boolean operations (AND,OR,NOT) may be performed on sets of names

The difference between the Query system and Name-Selection is that Name-Selection may only specify selection criteria in terms of object types while the Query system provides a substantially more powerful facility to select objects from the database according to user-defined criteria.

1.5.4. Analysis and Display Processor

The analysis and display processor consists of two kinds of reports : standard and specialized reports. The specialized reports are dedicated to a particular LSS. On the contrary, the standard reports are based on a common software for all the LSS's.

In SEM there are three types of standard reports :

1. The Formatted Statement Reports (FS)
2. The Structure Reports (STR)
3. The Extended Picture Reports (EP)

The Formatted Statements (FS) processor selects all the statements in the LSS language for each object specified in the input. The reports contain the language statements, for each object named, that would recreate the database if IP were applied to the database. The statements are produced in a predefined order (FS). This means that the user has no control over the order of, nor the kind of statement to be produced. If the user wants to have these possibilities he must use the Selective FS (SFS) report.

The Structure Report (STR) presents in the form of indented lists and matrix a hierarchy associated with one or more objects. The user specifies the object types as well as the relationship types that define the hierarchy. For every name given as input a structure is displayed in an indented

list.

The Extended Picture (EP) Report is similar, in content, to the Structure Report in that it displays hierarchy. The hierarchy is represented in form of a tree. The tree is displayed with the root node placed at the left hand side and extending from left to right.

1.5.5. The Database Interface

The database interface is made of the Data Management Interface (DMI) and the Meta Table Interface (MTI).

DMI is the interface between the processor and the LSS database. All accesses to the LSS database are handled through this interface. The DMI is not used directly by the LSS's users but it is used by the processor programs. The DMI isolates the low level (implementation level) organization from the conceptual level (E-R model). This isolation allows the choice of a low level database handler (database management system).

The generalized software accesses the tables whenever it needs information about a particular LSS through the Meta Table Interface (MTI). MTI hides the physical implementation details from the upper level components. It is also a "procedure" interface to the information that characterizes the particular LSS.

1.6. ADBMS : A Database Management System for the ISDOS Software

This section presents the main features of the database management system used to handle data in the different parts of the ISDOS software. To have a detailed understanding of the data description language (DDL) and how ADBMS works, the reader is suggested to consult the following papers [ISDOS 1981a], [ISDOS 1978].

1.6.1. The Data Description Language : DDL

ADBMS is transportable general purpose database management system based upon the CODASYL 1971 DBTG model for the network database approach [Engles 1971].

The description of the database is known as the schema. The schema used is based on the Language Journal of Development [CDDL 1974].

A schema written in the DDL contains four types of entries :

1. one schema entry which identifies the schema
2. one or more areas entries which define the grouping of records into areas
3. record entries which define record types specifying details of their data items and data aggregates
4. set entries which define the grouping of record types into set types

An area in the database allows users to segregate the pages on which different record types are stored. This will often improve the search time for these records. Each record type which is described can be assigned to an area. If no record types are assigned to specific areas then all the record types are assumed to be in the same area.

A record in the database is conceptually like a record in a file, in the sense that it is a collection of data items. There can be an arbitrary number of records of a record type stored in the database. The data items and data aggregates are decribed in a fashion similar to COBOL.

The basic construct of the language is called a set. A set is an occurence of a named collection of records and each set type can represent a relationship between two or more record types as shown in fig 1.5.

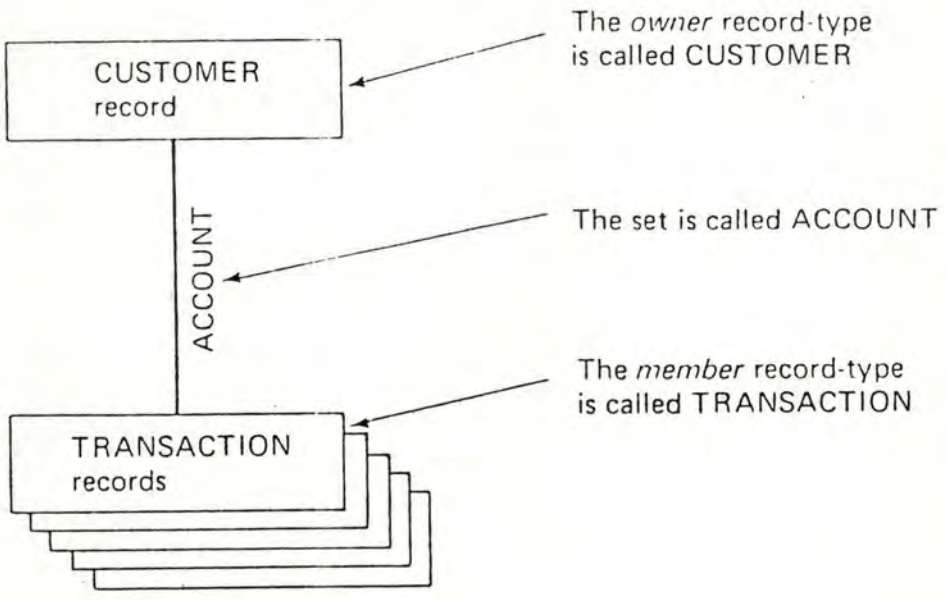


Fig 1.5 An occurence of a set (From [MARTIN 1975])

A set type can have one or more record types declared as its members. Each set must contain one occurence of its owner record type. A record type may be both an owner and a member of the same set type.

In the database, each owner record has associated with it its member records. Each of those members can in turn be defined to be the owner and/or member of other sets. These member record instances may not, however be a member of more than one instance of any set type, althought it may simultaneously be a member of several different set types. In the database a set is identified by specifying the owner record.

Each set type has an order specified for its member records in order to control the sequence of retrieval of the records and to control the storage of new member records into the sets of that set type. The set ordering criteria definition specified in the schema for each set type determines the logical ordering of the member records in each set of that set type.

The physical order of the member records in the database is independent of the logical ordering. In addition, the set ordering definition for each set type can be different so that a record that is member of two different set types could be subject to retrieval in two different orders. There exist four ordering criteria :

1. The order may be immaterial. This means that it makes no difference in what order the member records are stored or retrieved.

2. The records can be inserted in the order of storage of records, either FIFO or LIFO.

3. The records can be inserted immediately prior to or after a record which is current at the time of reference.

4. The records can be inserted in ascending or descending sequence of a key value.

1.6.2. ADBMS Overview

ADBMS is a database management system consisting of four parts :

1. the Database File (DBF) which consists of the data that is to be accessed

2. the Database Tables (DBT) which is the logical description of the structure of the database and is stored in the beginning of the DBF

3. the Database Control System (DBCS) which consists of a collection of subroutines to be called from FORTRAN. This is the actual programmed interface to the database.

4. the database utility routines which are three stand-alone programs. These programs aid the analyst in creating and maintaining a collection of databases.

1.6.2.1. The database Tables (DBT)

DBT which contains the database tables is generated by a program named DDLA whose input is a database description written in the DDL. DBT consists of three tables (ARETAB, RECTAB and SETTAB), a character vector of DDL names (NAMES) and some fullwords of control information. It is placed in the first pages of the DBF by a program named DBIN whose input is the output from DDLA.

ARETAB can be viewed as a linear vector containing area description blocks (ADB) describing the different areas of the schema.

SETTAB consists of set description blocks (SDB), owner description blocks (ODB) and member description blocks (MDB). These three types of control blocks describe the logical structure of the records in the

database.

RECTAB can be viewed as a linear vector containing two types of object schema control blocks, the record description blocks and the item description blocks associated.

Each object in the schema (records,sets...) is identified into the character array NAMES.

1.6.2.2. The Database File (DBF)

The information stored in the database is placed by the DBCS into the DBF. The DBF consists of physical pages usually defined at a computer installation to be equal in size to some unit of storage for that installation.

The DBF is initialized by a program named DBIN which sets up the first page (more if necessary) to contain information from the DBT produced by DDLA. Starting at the top of the following page in the DBF, the remaining pages are initialized for general database use.

1.6.2.3. The Database Control System (DBCS)

The DBCS is a collection of FORTRAN routines which interface with the user's program and with the ADBMS utility programs. They are divided into four groups classified by functions. These groups are :

1. DBUSER
2. DBLOW
3. DBTAB
4. DBRAND

DBUSER is the highest level of the DBCS system. The routines contained in DBUSER directly interface with the user's programs. The routines give the user program control over what database is to be accessed, what information is to be stored, in which sets, etc. The user's program does not need to do anything to insure that the correct database page is accessible, to locate available database holes for record storage, to determine the physical location of a record or an item.

The routines in DBLOW are low level routines used by DBUSER which do much of the actual work involved in manipulating the data in the database. They find control blocks, set pointers, compare items, etc. DBLOW also takes care of much of the page management

facilities needed by DBUSER. DBLOW relies heavily upon DBTAB and DBRAND to access the database.

DBRAND is used to get database pages in and out of main memory.

DBTAB is a set of routines which are used by DBUSER and DBLOW to access the database tables.

1.6.2.4. ADBMS utility programs

There are three utility programs available for use with ADBMS. Each is a stand alone program which calls routine in the DBCS. The programs are the following :

1. The Data Description Language Analyser (DDLA)

   This program generates the database tables (DBT) from a DDL. It also produces a FORTRAN block data subprogram which is used by the DBCS. In addition a DDL summary is produced for every record type and set type in the database.

2. The Database Initialization Program (DBIN)

   This program uses the DBT generated by DDLA to initialize a database (DBF).

3. The Database Summary Program (DBSM)

   DBSM produces for a populated database summary information on the sizes and percentage utilization of database holes and records.

Chapter 2 : OVERVIEW OF SEM PERFORMANCE

2. OVERVIEW OF SEM PERFORMANCE

2.1. Introduction

The evaluation and improvement of performance in any software system as large as SEM is of prime importance to everyone involved with the system, that is to say, the designers, implementors, maintainers and users.

Such an evaluation is a very complex task because SEM is used by several kinds of users with varying degrees of experience, job volume, performance requirements and computing resources. In addition SEM must be maintainable and transportable to a large number of computing environments.

In this chapter an attempt is made to isolate some levels of software and associated data structure which are critical for the performances. First is presented an evaluation of the performance of the IP and RP processors. After, the performance of ADBMS is studied.

2.2. Performance of the IP and RP Processors

2.2.1. Performance of the IP Processor

IP is the command where the performance problem first comes to the user's attention. A large amount of resources are used in IP and therefore the major effort in performance must be directed toward IP. In the next part of this section we will present conclusions of a study [ISDOS 1981b] about the performance evaluation of IP.

IP can be viewed as having three logical functions. First, the SEM input statements must be parsed and checked for syntax errors. Then, the logical consistency of the SEM statements must be checked against the content of the current SEM database. Finally, the current SEM database must be updated if no syntax or consistency errors were detected. Measurements were made at the level of each of these logical functions.

The first logical function is performed by a module called SNTX. This module contains the high level software to perform the parsing function. It utilizes also the MTLIB and DMLIB libraries as interface to the Meta database and LSS database.

The highest level subroutine in the SNTX module is the PARSER subroutine. All other subroutines in SNTX are called by the PARSER.

Ideally, the resources consumed by the PARSER should be independent of the SEM database size and other physical constraints. Unfortunately, the current implementation of the PARSER uses the Dynamic Storage subsystem (DS)

extensively.

This subsystem is a general dynamic storage system used by the ISDOS software and whose functions are the following:

- allocation and deallocation of storage
- storage and retrieval of character strings, logical values...
- copying of strings or blocks of words from one dynamically allocated area to another
- comparison of strings
- searching a dynamically allocated area from a given word

The use of such a system can cause contention for memory ,in some cases, depending on the SEM database size and user work constraints. The figure 2.1 shows that as the size of a SEM database increases, the average number of statements by CPU minute decreases in PARSER.

Measurements were made on a file with 7543 PSL statements (PSL is a language defined at the University of Michigan for describing IPS). The statements result in a database of 119 pages (4096 bytes).

More studies should be performed to better isolate the causes of increasing consumption as the size of the database increases. In particular, the SNTX module subroutines should be investigated to find the calls which are causing the observed increases.

For the two other logical functions, D.Childs has arrived at the conclusion that there exists a nonlinearily increasing dependence of IP consistency and updating functions on the database size as shown in figure 2.2. Measurements were made on a file with 4419 PSL statements.

The cause of such a dependence must be studied carefully. For this purpose it is necessary to examine IP algorithms and ADBMS software and interface (DMI) to determine the causes of the nonlinear increases in resource consumption.

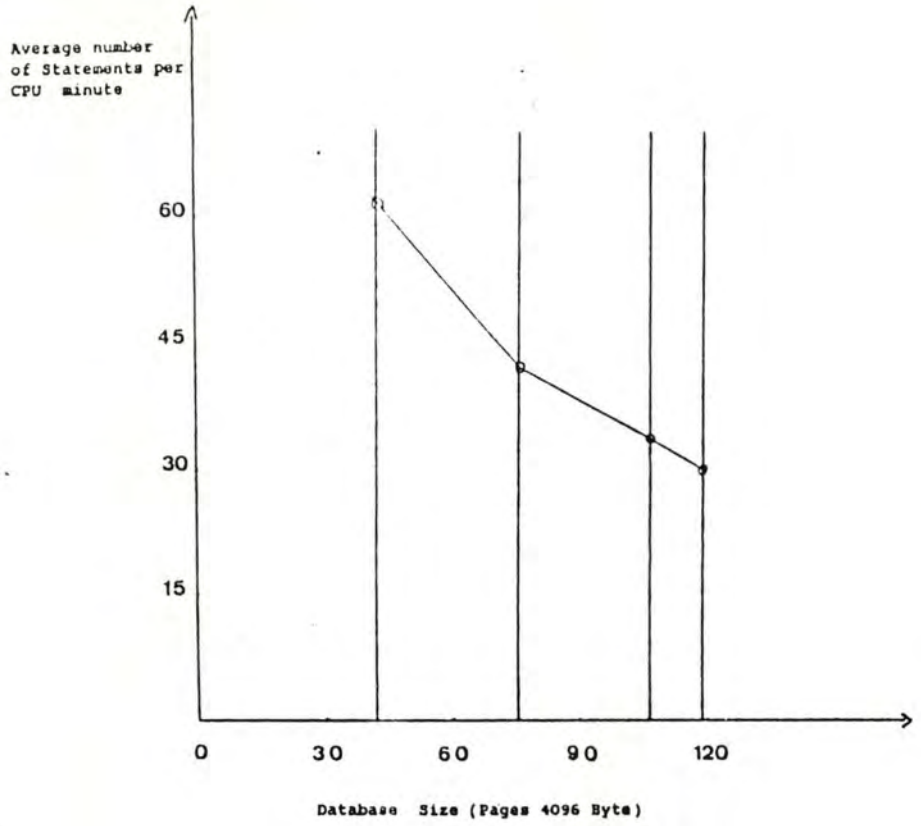


Fig 2.1 Syntactical function

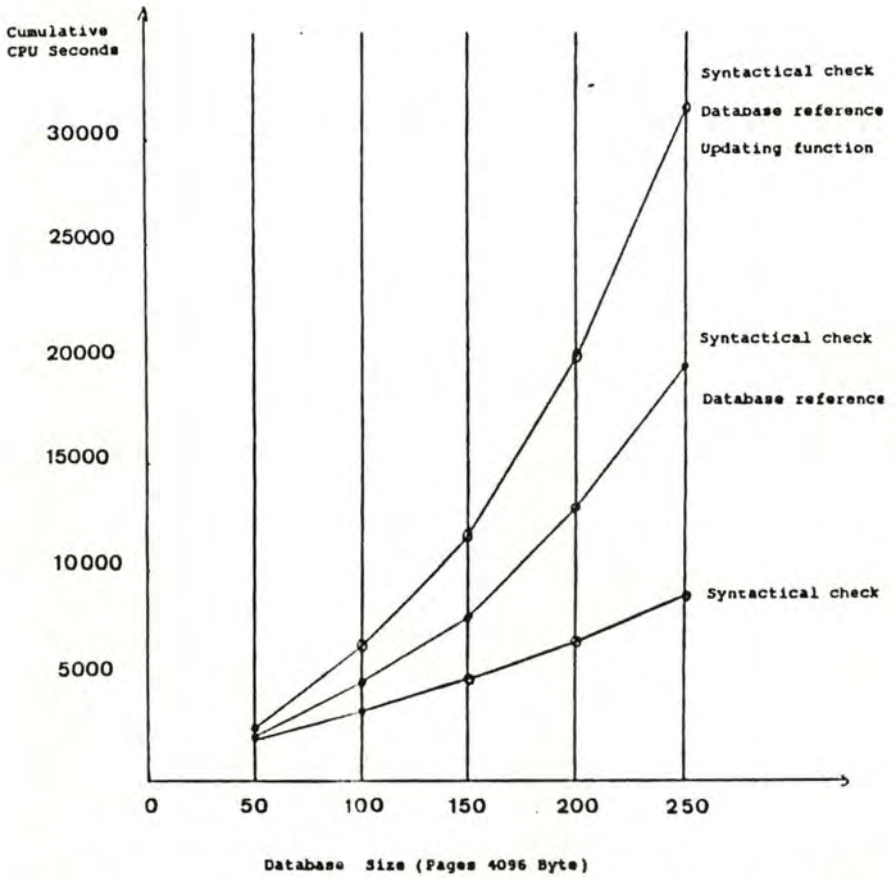


Fig 2.2 Syntactical , Consistency and Updating functions

2.2.2. Performance of the RP Processor

The performance of SEM reports commands come to the users attention only recently. This is probably due to the fact that not many reports have been run from large databases. In the future, more concern about the performance reports can be expected and thorough studies should be performed.

Recent measurements on the stucture report processor (STR) have been made at the University of Namur and have shown that the greatest part of the CPU time was spent in the DS subsystem (20 % of the CPU time) and the I/O routines (25 % of the CPU time).

As the DS subsystem performs all the allocations and deallocations of areas such as the retrieval,the copying and the comparisom of areas, it is understandable that much time is spent on the structure report because we must produce a hierarchy associated with one or more objects and thus search the entire database..PP 1 The I/O routines perform the transfer of pages from the secondary memory and the main memory. The time spent in these routines can be anticipated depending on the database size. In effect the greater the database is, the greater is the number of page faults and thus the need for transfer.

Furthermore, the transfer of a page from the secondary memory to the main memory is a slow operation which is made slower by the fact that they are performed in using the format statement of FORTRAN. These statements are interpreted in FORTRAN and this causes an increase in CPU time spent. A major improvement would be achieved if these routines were written in Assembler.

2.3. ADBMS Performance

2.3.1. The DDL for a SEM Database

The DDL [ISDOS 1981c] proposed by ADBMS is a rather restricted subset of the Codasyl specifications for network Database Management System (DBMS). It contains the minimum of the possibilities needed to describe a Codasyl database. The current DDL allows one, of course, to describe records, items and sorted sets, but the majority of the options proposed by the Database Task Group Report [DBTG 1971] are not allowed in the current implementation. Section 2.3.3 describes some mechanisms which are commonly in use in most database management system.

These mechanisms improve greatly the design of a database because they allow the user to take into account physical considerations to design a database following the user's performance requirements.

The DDL for a SEM database is given in figure 2.3. There are twelve different record types and eleven different set types used in a SEM database :

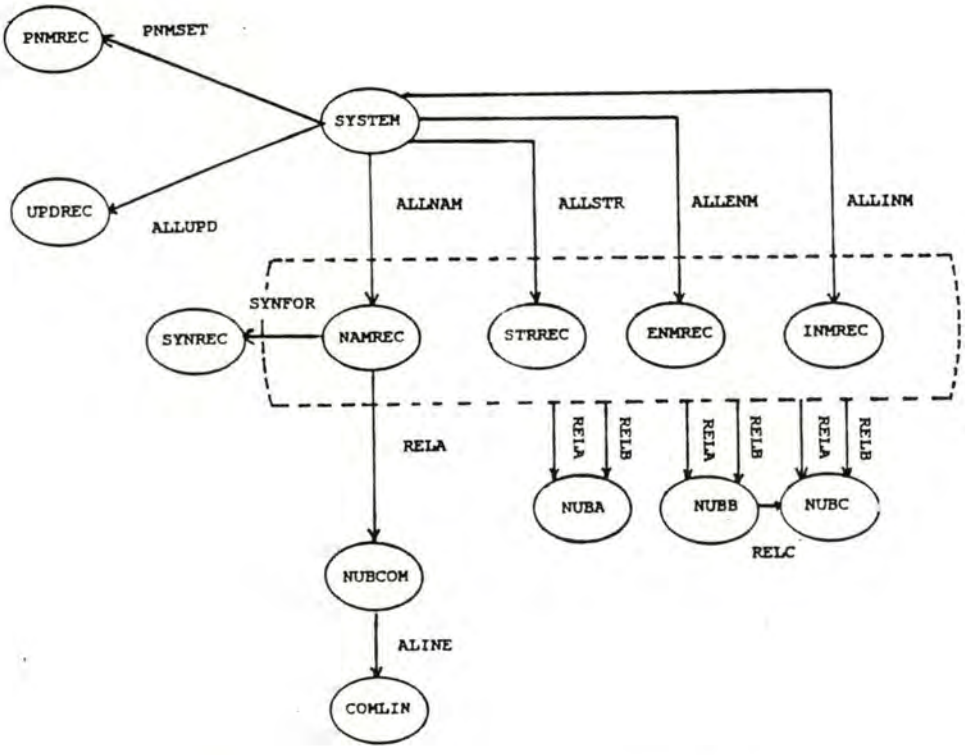


Fig 2.3 Schema of a SEM database

1. NAMREC Record

Each defined SEM name is stored in a NAMREC record. There are three items in the NAMREC record. The first contains the coded number which indicates the object types this record represents. The second is the character form of the name and the third item is the sequence number of the update.

2. SYNREC Record

SYNREC contains a synonym defined by the user for a SEM name. SYNREC has two items. One contains the coded number of the object type and the second represents the character representation of the synonym name.

3. NUBA, NUBB, NUBC and NUBCOM Records

NUBA, NUBB, NUBC and NUBCOM are of a generic type of records called a nub. The nubs are used because the database system does not allow many to many relationships directly. Also the nubs are used to indicate the type of connection between the objects. For instance when there is a simple relationship between two objects, a NUBA record is used.

When the connection is more complicated NUBB and NUBC records are used. These records contain a RELTYP item which has a code which designates the type of relationship the record is being used to represent.

4. INMREC, ENMREC and STRREC Records

INMREC and ENMREC are used whenever an integer number(INMREC) or a real number(ENMREC) needs to be stored in the database. They possess two items. One indicates the object type and the other contains the value of the number.

STRREC is used whenever a character string needs to be stored in the database. STRREC contains three items. One represents the object type, the second is the representation of the character string itself and the third item is the number of characters in the string.

5. COMLIN Record

Whenever a comment entry needs to be stored in the database, the lines which make up the comment entry are stored in a COMLIN record.

6. UPDREC Record

The UPDREC record is used to keep track of all the modifications made to the SEM database. Whenever a SEM modifier command is used to change the content of the database, an instance of a UPDREC record is created. There are four items in a UPDREC record. Two

indicate the data and time of the update, one indicates the numerical code corresponding to the commands used to update the database and the last item is the sequence number of the update.

7. PNMREC Record

The PNMREC record has a single item which contains the character representation of the name given to the database.

8. ALLNAM Set

This set is used to locate an object based on its name. It specifies an alphabetical ordering of all the basic names and synonyms the user defined.

9. RELA Set

This set has objects, numbers and character strings (NAMREC, INMREC, ENMREC and STRYEC records) as owner and various types of Nubs as members. It is used to specify the "left hand side" of "simple" and "complex" connections.

A connection between two records is called "simple" and only involves a NUBA record between them. A connection in more than two records is called a complex connection and requires the use of a NUBA and a NUBC record.

10. RELB Set

This set specifies the "right hand side" of connections between records, objects numbers and character strings which are the owner and various types of nubs are members.

11. RELC Set

This type of set is used when specifying "complex" connections involving more than two records. This set relates a particular NUMB record to a NUBC record.

12. ALINE Set

This set specifies the relationship between a NUBCOM record and its associated COMLIN records.

13. SYNFOR Set

Each name which the user defines as a basic name can have an arbitrary number of synonyms. Synonyms are stored in SYNREC records. The relationship between the basic name and its synonyms is desigated by the set SYNFOR with the basic name as owner and all the synonyms as member.

14. ALLUPD Set

This set contains all UPDREC records. It is a system set and the members are sorted by the value of the sequence of the order.

15. PNMREC Set

This set contains a single record of the type PNMREC. If no name has been assigned to the database this set will be empty.

16. ALLINM, ALLENM and ALLSTR Set

These sets are system sets. The sets ALLINM, ALLENM and ALLSTR contain the records INMREC, ENMREC and STRREC respectively.

2.3.2. Implementation Evaluation

All the sets in ADBMS are implemented as a doubly linked chain as shown in fig 2.4.

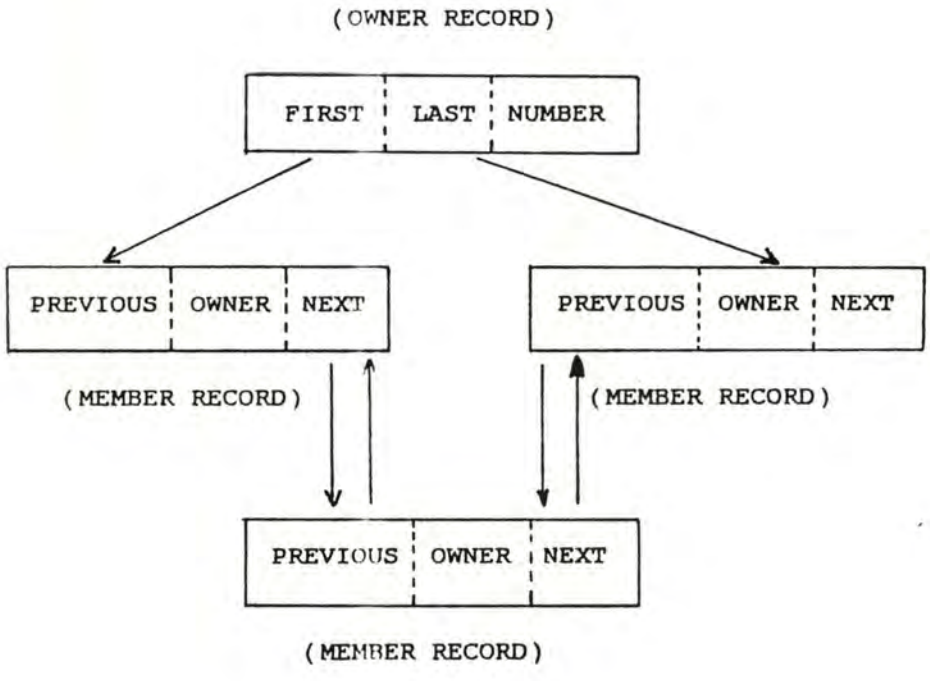


Fig 2.4 Set implemented with a doubly linked chain

For a set, all the searches are made linearly. The search algorithm for sorted sets works in the following manner. The search begins at the current member of the set, if there is a current member, or the first member if the

member currency is nul.

If a current member exists, then a comparison of the current member's sort key and the desired member's sort is made to determine whether to search forward or backward in the set. The search continues in the right direction until either the record is found or it can be determined that the member is not present.

If the current member is nul, the search begins with the first member and continues forward until the member is found or determined to be missing.

Regardless of other performance considerations, we can predict that this current procedure for searching sorted sets is inefficient and can cause an overhead in CPU time and database page faults.

Studies were performed by David Childs [ISDOS 1981b] on the ADBMS commands which make sequential searches on sorted sets. These commands are AMSK (add a member to a set based on key) and FMSK (find a member of a set based on key).

AMSK requires a sequential search, based on a logical sort key, of the members of a given set type instance defined by the current owner until the correct position in the sequence of logical sort keys is found for the new members.

FMSK requires a sequential search based on a logical sort key, of the members of a given set type instance defined by the current owner until the designated member is or is not found.

The results of the studies made show that the CPU time consumed by FMSK and AMSK on the ALLNAM set during an execution of the IP command is increasing nonlinearly. We can conclude that the searching procedure used by FMSK and AMSK to search sorted sets is dependent on the database size. This dependency was expected upon examination of the searching algorithm utilized by the two routines.

Furthermore, in certain cases in IP, the ALLNAM set is searched twice for a single name. The first search is to determine if the name exists in the database (this is made by FMSK). If the name is not in the database the ALLNAM set is searched again with AMSK for the first instance that sorts after the instance to be inserted so that the new instance will be inserted at the correct place in the set. The first task requires on the average N/2 comparisons (N is the number of member in the set) to search a sorted set. On the average the second task takes also N/2 comparisons.

David Childs has shown that these routines were utilized extensively by the IP command so that the resource consumption dependency of the two routines on the database is very significant in terms of performance. Thus a part of the poor performance such as the nonlinear increase in consumption of CPU time, the rapid decrease in the average number of statements processed per CPU minute and the rapid

increase in CPU time consumed as the number of basic names increase in the database can be attributed to the search algorithm.

A major improvement will be achieved in systematicaly constructing an index table for each of the system owned sets, i.e, ALLNAM,ALLSTR,ALLINM,ALLENM.

2.3.3. Implementation of the Codasyl Model : a Synthesis

The goal of this section is to describe some mechanisms which are commonly proposed by most of the database management systems.

The current implementation of ADBMS is a rather restricted subset of the CODASYL specifications for network DBMS [DBTG 1971]. Consequently the physical design of any database system is severely limited by the absence of clauses such as the Location mode and the Set mode as they are defined by the Codasyl Database Task Group report.

In this section we review some mechanisms which are usually implemented in most Codasyl DBMS and which greatly improve the design and performance of a database.

2.3.3.1. Location Mode

Location mode is a clause which allows specifications of the access methods for each record entry appearing in the schema. In particular, one useful mechanism is to have the possibility to declare LOCATION MODE IS CALCULATION.

Thanks to this, retrieval is based on the values supplied by the program for the data items which are contained in the record sought and which have been declared as CALCULATION keys. The DBMS transforms the values provided into a unique identifier and retrieves the record on the basis of that identifier.

2.3.3.2. Set Mode

Many DBMS [Pholas 1974], [UDS 1977] provide ways to improve the physical organization and placement of data. For this purpose, the data description language may not be used because to do that would destroy the independence between the logical and physical view of a database.

Generally, a Storage Structure Language (SSL) is provided which allows to specify physical parameters to improve the physical design of a database.

The set mode clause is one important part of this language because it allows one to choose ways to

implement the set concept. For each set type the designer must choose a trade off between time and space. This selection depends heavily on the characteristics of the processing to be performed.

As it could not be forecast at the time the DBMS was developed, the schema SSL allows choice, from among several alternatives provided by the DBMS, of the approach to be employed for any given set type.

In general three set modes are provided : CHAIN, POINTER ARRAY and LIST.

1. Set Declared as CHAIN

For each occurrence of a set declared to have a mode of CHAIN, a chain of pointers is created which provides serial access to all records in the set occurence. The pointers may or may not be embedded in the records themselves.

The owner record contains a pointer to the first member record in the logical sequence defined. The member records contain a pointer to the logical next member record unless the optional clause LINKED TO PRIOR is used. In this case, additinal links in the reverse direction are also provided.

In addition, the occurence of any of the member record types specified for a set may be declared to be linked to the owner. This causes the owner record of the set occurences to be accessible directly from each of the member record occurences as illustrated in fig 2.5

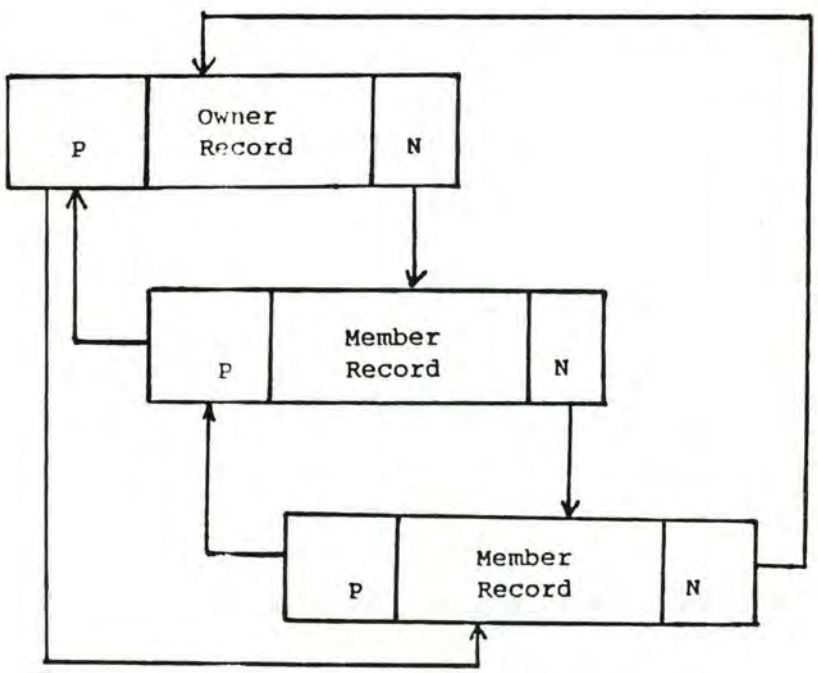


Fig 2.5 Set declared as CHAIN

If in addition the clause ORDER IS SORTED INDEXED has been specified for a set to be stored as chain, an index is created and maintained to retrieve the member records directly.

In this case, the chain of pointers in the records is used for sequential processing while directly accessed records will be retrieved via this index.

2. Set Declared as POINTER ARRAY

A pointer array is the functional equivalent of a chain. In sets declared as pointer arrays, member record occurences do not contain pointers to each other, but however they contain pointers to their owner record occurences.

Each owner record occurence is associated with a list of all of its member record occurrences as shown in fig 2.6.

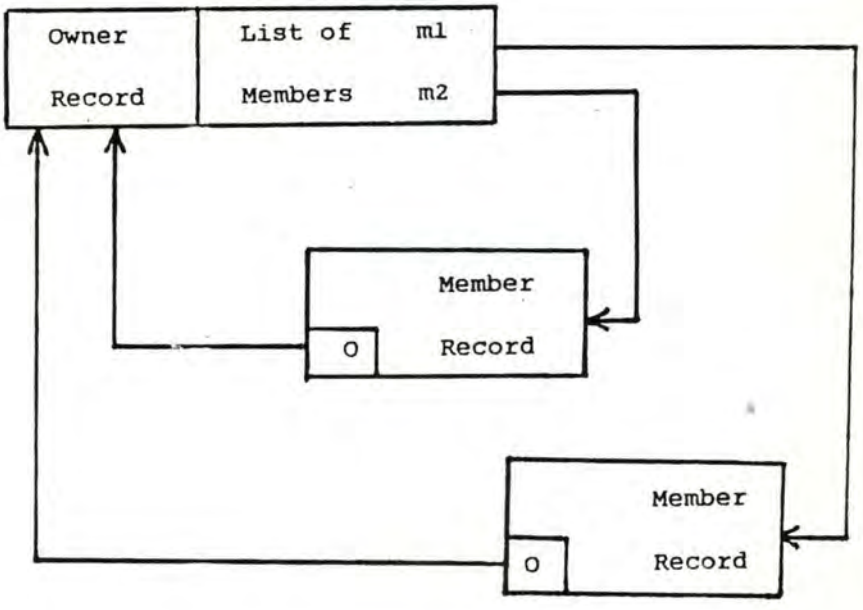


Fig 2.6 Set declared as POINTER ARRAY

The main advantage of such a technique is that the list of member record occurrences developed for pointer arrays allows some logical operations to be performed on the member records listed without the necessity of accessing the records themselves.

An advantage, for example, is to perform

logical AND and OR operations on the members of two or more sets occurrences without accessing any of the member records.

When the mode is POINTER ARRAY for a set, for each occurrence of the set an array is built containing a reference to the member records of the set occurrence. The references in the array are stored and maintained in the logical order specified for the set.

3. Sets Declared as LIST

When the lists are used as a storage technique, the set occurrence can be represented as indicated in fig 2.7

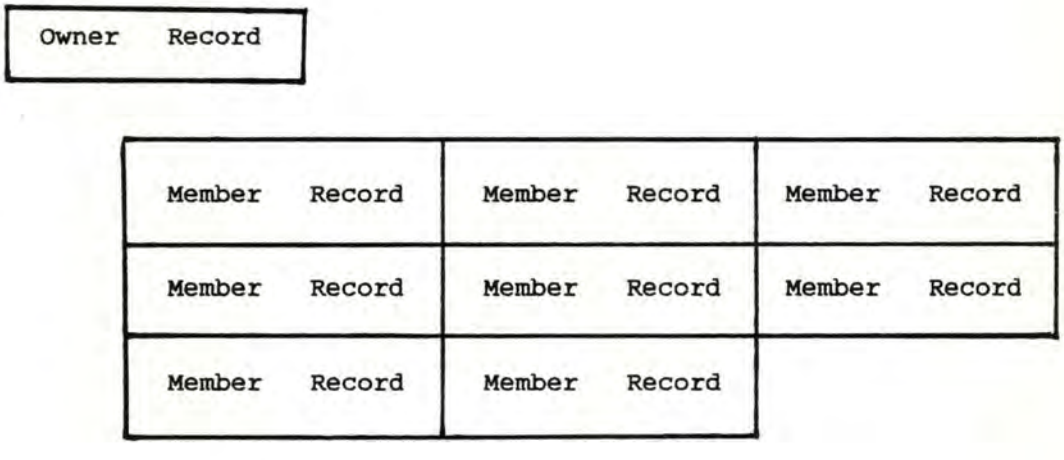


Fig 2.7 Set declared as LIST

The member records can be considered as sequentially stored in the logical order specified by the user. When the member records can not all reside in one page the participating pages are connected.

Since the list is treated as a table, an index is built with the highest ascending or descending key in each of the participating page. When we want to retrieve a particular record, the index is first used to determine in which page the record resides and after a scan is done through the page to retrieve the record.

2.3.3.3. Effect of Storage Structure on Processing Efficiency

When a user program is running, it executes storage and access operations on database. The time required to execute a program will depend heavily on the type of storage structure. This section gives a general comparison of the three set modes.

1. POINTER ARRAY

In the case of sequencial retrievals, an entry of the pointer array will be read for each record and will be followed by a disk access to the member record itself.

It is assumed that the pointer array is in the core. This will be the case when a member record of the set occurrence has already been accessed via the set.

For a direct retrieval, the page number of the part of the pointer array in which the entry of the sought record must reside, will be selected. When the proper entry has been found, the member record will be accessed. For insertions and deletions, they both require access to the pointer array and an additional disk access either to store the member record or to rewrite the page to be changed.

We can conclude that "MODE IS POINTER ARRAY" results in fast sequential retrievals and fast direct retrievals and updates.

2. CHAIN

In the case of sequential retrievals, one disk access will normally be needed for each member record (if no other optimization has been specified for the set). For a direct retrieval we must distinguish two cases.

If the " SORTED INDEXED " option has been specified for the set, the member record can be found by means of a sort key table. This table will be accessed to retrieve the correct entry and a disk access will then normally be needed for retrieving the record.

If the " SORTED INDEXED " option has not been specified, the member record in the chain must be accessed by processing the chain sequentially. This will result in many accesses.

For insertions and deletions, two actions must be distinguished. First, the correct insertion or deletion point must be determined within the selected set occurrence and secondly,

the new member record must be inserted or deleted.

If the chain is only SORTED, half the number of member records must on average be read to find the required insertion or deletion point. If the chain is "SORTED INDEXED", the correct insertion point can be found through the index table which then be accessed. When the new record is inserted or deleted, the pointer in the previous (and sometimes next) record must be updated.

In conclusion we can say that the mode "CHAIN" will result in fast sequential retrievals even, faster than for the mode "POINTER ARRAY". Fast direct retrievals are also available if the option ORDER IS SORTED INDEXED has been specified. Insertions ,deletions and updates will be slower than for the mode " POINTER ARRAY ".

3. LIST

As soon as the page in which the list of member records is known, only one disk access will be needed if the list is stored in one page.

For direct retrieval of a member record in a list, the right entry will found via the index.

For insertion or deletion, the position at which the record must be inserted or deleted within the selected set occurrence can be found by direct access to the index if the option ORDER IS SORTED INDEXED has been specified.

When the ORDER IS FIRST, LAST, NEXT, PRIOR or IMMATERIAL, the currency information determines the correct insertion or deletion point. Insertion can cause a number of member records to be repositioned.

We can conclude that MODE IS LIST offers the fastest sequential retrieval method. When updating involves a change of sequence it is slower than for POINTER ARRAY.

4. Conclusion

Some Codasyl DBMS (PHOBAS, UDS) provide a storage structure language (SSL) which allows facilities for specifying how data records and set relationships must be stored physically.

In addition to the possibilities described in the previous sections, these languages often provide mechanisms which allow the user to control the placement of indexed, sort key, search key table as well as the lists and pointers arrays. These and other possibilities have not been decribed in this dissertation because this demands a much longer discussion.

2.3.4. Directions for Improvement

The performance consideration outlined above must be the first steps of a much more accurate and thorough study of all levels of the ISDOS software.

But as changes to SEM software will in general require modifications of all SEM installations, we must perform projections for performance improvement, completion times, effects on existing software, effects on the transportability and maintainability.

As far as concern the SEM reports and modifier commands, it is suggested that one perform a detailed architecture analysis of all reports and commands, and evaluate their logical design. This will help to locate inefficiencies and redundancies.

The extent for resource contention must also be examined among the SEM subsystems. For example we have already seen (in section 2.2.2) that in some cases, the DS subsystem and ADBMS contend for memory space.

Improvement in ADBMS would be the most beneficial since ADBMS is used extensively throughout all of SEM. Different areas can be investigated to try to meet this goal. We could for example replace ADBMS with a commercial DBMS and compare the performances. We could also implement major modifications to ADBMS such as hashing procedures, indexed sequential procedures or other records retrieval methods.

Another approach is to say that the informations processed by SEM are relational rather than network type of model.

A SEM language is an occurence of a Meta language based on the E-R model. This leads to poor performance since we have to provide an E-R interface to the ADBMS network database.

This function is served by the DM subsystem which allows the user to access to a target database. All the subroutines and functions of the DM subsystem work at the Meta language level in terms of objects, properties and relationships. Thus the user need never know the actual physical structure of the database.

One idea would be to use the SEM model (E-R model) as a data model and to implement a database management system based on this model. In opposition to ADBMS, hashing procedures should be use to improve the performance.

The next chapter is devoted to the description of such a system.

3. LOGICAL VIEW OF AN ENTITY-RELATIONSHIP MANAGEMENT SYSTEM (DBMS)

3.1. Introduction

The several criticisms of the ADBMS have focused attention on the need to change the current Database Management System (DBMS) of the ISDOS software.

One approach that has been selected is to construct an Entity-Relationship (E-R) DBMS with very high performances. The system should be required to support large databases and to involve many database retrievals and updates with fast response times.

In this chapter we summarize a logical view of an E-R DBMS, i.e, the E-R model, a data description language (DDL) and a data manipulation language (DML). The detailed definition and formalization of the E-R model can be found in [Chen 1976], [Sakai 1980].

3.2. The Entity-Relationship Data Model

1. Entities and Entity Set

An entity is a thing which can be distinctly identified: a student, a course , etc. The definition of an entity depends only on the perception of the world by the user.

Entities are classified into different entity sets. A predicate is associated with each entity set to test whether an entity belongs to the set. An entity has a set of attribute values which characterize it.

An entity set can be expressed by the following notation :

E(A1,.....,An) where E is the name of the set and each Ai is an attribute typeto which a value set V(Ai) is associated.

2. Relationship, Relationship Set, Role

A relationship is an association between entities. The existence of a relationship depends upon the existence of the entities connected by this relationship. Each relationship can have attributes.

A relationship set Ri, can be viewed as a mathematical relation among n entity sets. A relationship set can be expressed by the following notation :

R(E1,.....,En ; A1,.....,An) where R is the name of the relationship set and Ei and Aj are an entity set and an

attribute type respectively.

Each entity in the relationship plays a specific role. The role of an entity is the function that this entity performs in the relationship.

The connectivity of a relationship set represents one of its essential characteristics and makes part of its description.

Assume we have a relationship set :

R(E1 ,.....,En ; A1,.....,An)

the connectivity of R will be defined by the affectation of minimum and maximum values, Cmin and Cmax respectively, for each entity Ei which belongs to R.

Cmin (Ei) is the minimum number of times that each instance of Ei may be implied in an instance of the relationship R. If Cmin (Ei) = 0, one instance of Ei may exist independently of instances of R. However if Cmin (Ei) = 1.....n, one instance of Ei cannot exist without making part of 1 to n instances of R.

Cmax(Ei) is the maximum number of times that each instance of Ei may be implied in an instance of R.

3. Attribute, Value and Value Set

The informations about entities are expressed by means of attribute values : "5" , "John", "green" are values. Values are classified into value sets.

An attribute can be formally defined as a function which maps from an entity set or a relationship set into a value set or a cartesian product of value set :

f : Ei or Ri V i or Vi1 $\times$ ..... $\times$ Vin

Fig 3.1 illustrates some attributes defined on the entity set employee. The attribute Name maps into value sets First-Name and Last-Name and the attribute Student-Number maps into the value set Student-Number.

4. Entity Key, Primary Key

A set of attributes called an entity key identify one entity set. This set is such that the mapping from the entity set to the corresponding group of value sets is one to one.

If there are several entity keys, we arbitrarily select one (the most meaningful) as the identifier of the entity set. This identifier is called a primary key.

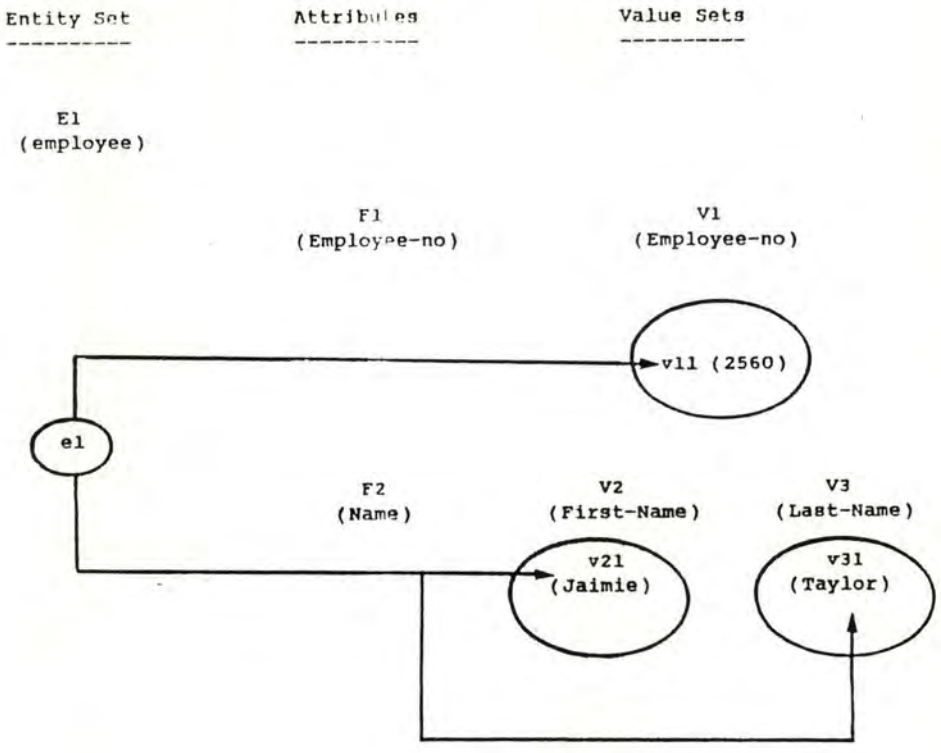


Fig 3.1 Attributes defined on the entity set student

5. Weak Entity and Relationship

Sometimes, the entities in an entity set cannot be uniquely identified by the values of their own attributes. We must use a relationship to identify these entities.

For example, if we consider two entity sets WORKER and DEPENDENT (dependent represents the persons which are supported by the worker). Let R be the relationship set that connects these two entity sets. This relationship set indicates the fact that a worker supports the dependent with which he is relied on. Dependents are identified by their names and by the values of the primary key of the employee supporting them. Dependent is called a weak entity set.

In summary there exist two kinds of entity and relationship types. If relationships are used for identifying one entity type, it will be called a weak entity. Otherwise, it will be called a regular entity.

In the same way, if some entities in a relationship are identified by other relationships ,it will be called a weak relationship. Otherwise, it is a regular relationship.

6. Integrity Constraints

Constraints for maintaining data integrity exist. A constraint is a predicate on an element of the E-R model and which must be checked during a given period of time.

These constraints limit both the possible instances of the element of the model and the set of operations that we can realize on it.

The usual constraints of integrity are the following [Chen 1976] :

- connectivity of a relationship set
- constraints on allowable values for a value set
- constraints on permitted values for a certain attribute:

  All allowable values in a value set are not permitted for some attributes. For example, the age of a worker can be limited between 18 and 60.

- conditions and time of existence of an entity, an association or attribute

7. Restrictions to P.Chen's Model

For simplicity reasons, we have chosen not to implement the weak entities and relationships. For the same reasons , constraints of integrity will be limited. The user can choose a minimum and maximum value for a value set. He can also define a set of values for a value set. The connectivity can also be stated.

3.3. An E-R Data Definition Language (DDL)

3.3.1. Introduction

An E-R data definition language is explained in this section. The DDL allows a description of the logical view of an E-R data model.

Unfortunately, this DDL is not totally independent of the physical implentation and this for two reasons.

First, at each entity and relationship set are associated some files (see chap 4).The initial size and the load factor of the file are parameters which the user can vary and this for performance reasons. The description of these parameters are included in the DDL.

Furthermore, the description must be stated in a certain order. First the entities set and after the relationships. Inside the description of an entity set, the primary key must be stated first followed by the secondary key description and terminated by the attribute part. This last section contains the items for which a particular retrieval technique has not been implemented.

We think that these constraints are easily acceptable because the DBMS is in its prototype stage and has been constructed to enter the schema of an E-R model, to implement and test the physical level as soon as possible. When the DBMS is revealed to be efficient, a new DDL more flexible for the users must be built.

Basic constructs are covered in paragraph 3.3.2 and description of the DDL can be found in paragraph 3.3.3.

3.3.2. Basic Constructs

1. DDL Primitives

   The description of a target database in the DDL is in terms of DDL primitives: entity type and relationship type.

2. DDL Sections

   The description for each instance of the primitives is given in an DDL "section". For example, if two entity types are to be decribed, two entity types sections are given.

3. Characters Set

   The DDL characters set is divided into four classes (see fig 3.2). Characters in class 1 and 2 are used for forming DDL names ; the only difference is

that the characters in class 2 are not allowed as the first character in a name. Characters in class 3 are DDL punctuation symbols that have specific meanings.

4. DDL Names

A DDL name must begin with a character in class 1 followed by zero or more characters from class 1 and 2. A DDL name may not be longer than thirty characters.

5. DDL Numbers

There are two types of number allowed in the DDL: integers and real numbers. Real numbers contain one (and only one) period as the decimal point. The maximum value and precision depends on the computing environment.

6. DDL Strings

A string is a sequence of characters from any class delimited by primes (') or double quotes (") on both sides. The maximum length of a string (not included the delimiters at the ends) is thirty characters.

7. Uniqueness of DDL Names

All names for the entity sets, the relationship sets, the attribute sets , and value sets must be unique within their class.

- CLASS 1 :

  A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z (upper case alphabet)

  a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z (lower case alphabet)

  ! (exclamation mark)

  $ (currency sign)

  % (percent sign)

  @ (at sign)

  - (underscore)

- CLASS 2 :

  0, 1, 2, 3, 4, 5, 6, 7, 8, 9 (digits)

  + (plus sign)

  - (minus sign)

  / (slash)

  < (less than)

  > (greater than)

  = (equal)

- CLASS 3 :

  ; (semi-colon)

  ( ) (parentheses)

- CLASS 4 :

  Any character available in the computing environment that is not included in class 1, 2 or 3.

Fig 3.2 Characters set

3.3.3. Decription Content

3.3.3.1. Notation

Each DDL section is described separately in section 3.3.3.3 and 3.3.3.2. There are three parts for each decription :

- Purpose
- Syntax
- Usage Rules

"Purpose" states what the DDL language section is intended to define. "Syntax" shows the prototype for all the statements allowed in a DDL section. "Usage rules" includes explanations and comments on the statements.

3.3.3.2. Special Meanings

- Undesrscored Words

  If the first word of a DDL statement is underscored then that statement is required for the DDL section in which it appears, otherwise, it is optional.

- Words in Upper-Case Letters

  A word in the prototype is printed in upper-case letters if that word is a DDL keyword and must be spelled in upper case letters exactly as given.

- Words in Lower-Case Letters

  Words printed in lower-case letters and not within angle brackets call for the language user to give names to replace these words. The lower-case words describe what kind of name is to be given.

- Words enclosed in braces ({ })

  When words or phrases are enclosed in braces ({ }), a choice from two or more entries must be made. The choices are separated by a vertical bar (:)

- Words Enclosed in Angle Brackets (< >)

  Only four words are enclosed in angle brackets (<

›) : ‹integer›, ‹real›, ‹string› and ‹value›.

If ‹integer› is specified then any integer must replace ‹integer›. ‹real› and ‹string› are interpreted in the same way.

‹value› means that the value which must replace ‹value› is either integer, real or string type depending on the type defined in a statement above.

- Semi colon (;)

The semi colon indicates the end of each statement.

3.3.3.3. Entity Type Section

- Purpose :

To define an entity type and its characteristics.

- Syntax :

```
ENTITY-TYPE entity-type-name ;

PKEY <integer> <integer> <integer> ;

ATTRIBUTE attribute-name ;

VALUE-SET value-set-name {INTEGER : NUMBER : CHAR}
<integer> ;

RANGE <value> THRU <value> ;

VALUE-ALL value ;

SKEY <integer> <integer> ;

ATTRIBUTE attribute-name ;

VALUE-SET value-set-name {INTEGER : NUMBER : CHAR}
<integer> ;

RANGE <value> THRU <value> ;

VALUE-ALL <value> ;

ATTRIBUTE PART ;

ATTRIBUTE attribute-name ;

VALUE-SET value-set-name {INTEGER : NUMBER : CHAR}
<integer> ;

RANGE <value> THRU <value> ;

VALUE-ALL <value> ;
```

- Usage Rules :

Each entity set is divided into three groups: primary key, secondary key and attribute part.

The primary key part is used to define a set of attributes which form a unique identifier for the entity set. The PKEY statement has three parameters which indicate some initial parameters for a hashing file (for the signification of these parameters see chap 4) : the first represents the initial group size,

the second indicates the number of buckets inside a group and the third is the load factor of the file.

Each PKEY statement must be followed by one or more attribute parts. Each attribute part is composed of one ATTRIBUTE statement and one or more value set parts.

The ATTRIBUTE statement defines the name of the attribute and must be followed by one or more value set parts.

A value set part is composed of one VALUE-SET statement optionaly followed by a RANGE statement and one or more VALUE-ALL statement. The RANGE statement indicates a range for the value set defined. The VALUE-ALL statement defines a value allowed for the value set. These two statements are optional. The integer at the end of the VALUE-SET statement indicatess the number of character representing a value for a value set.

The SKEY statement has also two parameters. The one indicates the exponent of 2 for the initial number of pages (see chap 4) and the other is the load factor for the file.

The SKEY stated the attributes of an entity for which a special technique is implemented to accelerate the search for these attributes. The rest of the description of the secondary key is the same than for the primary key.

The ATTRIBUTE PART statement is described in the same way as the PKEY and SKEY statement. This third part represents the attributes of the entity for which no special technique has been implemented.

3.3.3.4. Relationship Section

- Purpose :

to define a relationship type and its characteritics.

- Syntax :

RELATIONSHIP TYPE relationship-type-name <integer> <integer> ;

ENTITY-RELATED role-name entity-type-name <integer> <integer> ;

ATTRIBUTE PART ;

ATTRIBUTE attribute-name ;

VALUE-SET value-set-name {INTEGER : NUMBER : CHAR} <integer> ;

RANGE <value> THRU <value> ;

VALUE-ALL <value> ;

- Usage rules :

The description of a relationship set is composed of two parts : the description of the entity sets related and the description of its attribute part. The attribute part is defined in the same way as for an entity.

The RELATIONSHIP TYPE statement has two parameters. The one indicates the exponent of 2 for the initial number of pages (see chap 4) and the other is the load factor of the file .

A RELATIONSHIP TYPE statement must be followed by more than one ENTITY-RELATED statements. The two integers at the end of one ENTITY RELATED statement represent the connectivity.

There must be a value specified for the integer which represents the minimum number of times that an instance of the entity can be implied in the relationship.

The second integer must be >= to the first integer and if no values are specified , this means that there is no maximum number of times that each instance of the entity may be implied in an instance of the relationship.

3.4. A Data Manipulation Language

3.4.1. Introduction

A Data Manipulation Language (DDL) must be provided for the application programmers to manipulate the database via the E-R model.

The DML is a set of commands for retrievals, insertions, modifications, and deletions of data items in the data base. The interface between the application programs and the database is called the DBUSER level.

The set of commands belonging to this level are described below.

3.4.2. Subroutines Description of the DBUSER Level

1. CRENIN

Calling Convention :

CALL CRENIN (entyna, bufpar, ptrpar, ierr)

Purpose :

to create an entity instance

Description :

An entity whose values are contained in a buffer is added to the set of the entities identified by the given entity type name. The subroutine checks if the primary key already exists or is acceptable.

If the primary key already exists we can not insert the entity because a primary key uniquely identifies an entity. The primary key is not acceptable means that one of the values of the primary key does not belong to the range of allowable values defined for this entity type.

Arguments :

NAME	USAGE	TYPE	DESCRIPTION
entyna	input	char	entity type name
bufpar	input	char	buffer containing the values of the items

ptrpar	input	int	pointer into bufpar where the description of the entity begins
ierr	output	int	return code

2. CHENIN

Calling Convention :

CALL CHENIN (dbkey, attnam, vsnam, newval, ierr)

Purpose :

change the value of an attribute of an entity

Description :

An entity identified by a data base key (a data base key indicates the location ,i.e, the page number and a displacement within the page, of the primary key) is modified in the following way: the old value of a value set identified by its name and an attribute type name is replaced by a new value. If the value set is not a part of the primary key ,there is no consequence.

Otherwise we must change the primary key of this entity in all the relationships in which the entity is involved.

Arguments :

NAME	USAGE	TYPE	DESCRIPTION
dbkey	input	int(2)	data base key
attnam	input	char	attribute type name
vsnam	input	char	value set name
newval	input	char	new value
ierr	output	int	return code

3. DELENT

Calling Convention :

CALL DELENT (dbkey, ierr)

Purpose :

delete an entity

Description :

An entity identified by a database key is removed from the database. All relationships involving this entity are also deleted.

Arguments :

NAME	USAGE	TYPE	DESCRIPTION
dbkey	input	int(2)	database key
ierr	output	int	return code

4. RETENT

Calling Convention :

CALL RETENT (entnam ,pkey ,dbkey ,ierr)

Purpose :

to retrieve an entity

Description :

An entity identified by its primary key and its type is searched for in the database.The database key locating the entity is given as output. If the entity is not found then the database key is = 0.

Arguments :

NAME	USAGE	TYPE	DESCRIPTION
entnam	input	char	entity type name
pkey	input	char	primary key
dbkey	input	int(2)	database key
ierr	output	int	return code

5. RETVEN

Calling Convention :

CALL RETVEN (dbkey,attnam,vsnam,value,ierr)

Purpose :

to retrieve a value from an entity

Description :

The value item of an entity identified by a database key is retrieved and put into an integer array value. The item to retrieve is identified by an attribute name and a value set name.

Arguments :

NAME	USAGE	TYPE	DESCRIPTION
dbkey	input	int(2)	database key
attnam	input	char	attribute type name
vsnam	input	char	value set name
ierr	output	int	return code

6. RETVRE

Calling Convention :

CALL RETVRE (dbkey,attnam,vsnam,value,ierr)

Purpose :

to retrieve a value of an attribute of a relationship

Description :

The value item of a relationship identified by a database key is retrieved and put into an integer array value. The item to retrieve is identified by an attribute name and a value set name.

Arguments :

NAME	USAGE	TYPE	DESCRIPTION
dbkey	input	int(2)	database key
attnam	input	char	attribute type name
vsnam	input	char	value set name
ierr	output	int	return code

7. CREREL

Calling Convention :

CALL CREREL (relnam,bufpar,ptrpar,ierr)

Purpose :

to create a relationship

Description :

A relationship is created. Bufpar contains the primary keys and the attributes of the relationship.

Arguments :

NAME	USAGE	TYPE	DESCRIPTION
relnam	input	char	relationship type name
bufpar	input	char	buffer containing the values for the relationship
ptrbuf	input	int	pointer to the bufpar where the description begins
ierr	output	int	return code

8. CHREAT

Calling Convention :

CALL CHREAT (dbkey,attnam,vsnam,newval,ierr)

Purpose :

change the value of a relationship attribute

Description :

A relationship identified by a database key is modified in the following way : the old value of a value set is identified by its name and an attribute name is replaced by the new value.

Arguments :

NAME	USAGE	TYPE	DESCRIPTION
dbkey	input	int(2)	database key
attnam	input	char	attribute type name
vsnam	input	char	value set name
newval	input	char	new value
ierr	output	int	return code

9. DELREL

Calling Convention :

CALL DELREL (dbkey,ierr)

Purpose :

to delete a relationship

Description :

A relationship instance identified by a database key is removed from the database.

Arguments :

NAME	USAGE	TYPE	DESCRIPTION
dbkey	input	int(2)	database key
ierr	output	int	return code

10. RETREL

Calling Convention :

CALL RETREL (relnam ,pklist,dbklis,ierr)

Purpose :

retrieve a relationship instance

Description :

Given a relationship type name,and a list of primary keys which are related by this relationship (the name of the role precedes each primary key),this subroutine returns a list of database keys which correspond to the primary keys stated as inputs.

Arguments :

NAME	USAGE	TYPE	DESCRIPTION
relnam	input	char	relationship type name
pklist	input	char	primary key list
dbklis	output	int	database key list
ierr	output	int	return code

11. IDOP

Calling Convention :

IDOP (liobuf ,use,ierr)

Purpose :

open a database

Description :

The integer function IDOP is used to open a database. Liodbf designates the logical I/O unit to which the database is attached. If use is specified "0" then the database is only open for read operations.

Arguments :

NAME	USAGE	TYPE	DESCRIPTION
liobuf	input	int	logical I/O number
use	input	int	read/write flag
ierr	output	int	return code

12. DCLOSE

Calling Convention :

DCLOSE (ierr)

Purpose :

close the current database

Description :

the current database is closed

Arguments :

NAME	USAGE	TYPE	DESCRIPTION
ierr	output	int	return code

Chapter 4 : IMPLEMENTATION OF AN E-R MODEL

4. IMPLEMENTATION OF AN E-R MODEL

4.1. Introduction

There are two important problems to solve in order to provide an efficient access to the data structure.

First, the problem of dynamic files is a critical point for the usefulness of a system. Dynamic means that the size of the file may grow and shrink rapidly as the file undergoes insertions and deletions.

The second problem that arises is finding efficient techniques in terms of access to the data in the database. In particular, the number of page faults is a critical point for the efficiency of a system.

In the next section are presented the requirements of access to the data in the database as well as the addressing techniques that are available nowadays to provide efficient access to the data.Among these techniques, we have chosen two which seem to provide good performance. These techniques will be explained in section 4.3. We will outline in section 4.4 what are the advantages and the disadvantages of these techniques.

Section 4.5 will be dedicated to the description of the data base management system.

4.2. Presentation of the Problem

For the entities, we must be able to insert an entity instance, retrieve, and change the values of an entity. The problem is that you can retrieve one entity either by stating the value of a primary key or by stating the values of certain attributes. Typically, this is a multiple key retrieval problem.

For the relationships one must be able to retrieve an entire relationship by specifying some values of the primary keys that this relationship relates and/or by specifying some values of the attributes.

Usually, one key is used to uniquely identify records and is referred to as the primary key while the other keys are referred to as secondary keys. In most multiple key files today, the prime key determines the physical positioning of the records just as it does with most single key files.The secondary key addressing method is generally a technique which does not depend on the physical position of the record.

Overview of the classical techniques employed in the multiple key retrieval schemes [MARTIN 1975]

- Single Chains

  Single chains or rings through all items with a key kj have slow retrieval performance, but the index

size is small.

- Multi-List Organization

  Multiple chains or rings through items with key kj have faster retrieval, but the index is many times larger.

- Secondary Inverted List

  This classical file organization has a number of defects. Normally, if the secondary indices can be examined rapidly , the inverted list organization gives the fastest response to real-time inquiries because no chains have to be followed. On the other hand, the indices can be enormous and the organization of the indices themselves becomes a major file problem.

  This file organization also has the unfortunate property that the more fields that are specified in a query, the larger the amount of work that is needed to answer a query (more lists have to be intersected).

  Furthermore, it is not appropriate for dynamic files because of the expense of updating inverted indices. Due to the complex nature of the maintenance operations, most inverted file systems are updated off-line and, if possible, not too frequently.

  The periodic updates require the time-consuming process of sorting and reconstructing the tables.

- Associative Memories

  Associative memories would probably become one of the most powerful organizations that will change storage structure drastically in the future.

  However, until now, hardware associative memories large enough for database use have not been available for most systems.

  Software associative memories (file storage with properties similar to associative memories and using software techniques) are slow, clumsy, and often error-prone compared with their hardware equivalent.

The multiple key retrieval scheme is commonly implemented by techniques that do not provide good average performance when there are many records in the system.

The main reason is that these techniques generally use an index file to retrieve records by a secondary key. The size of the index can be very large, sometimes larger than the size of the main file itself. If the time to access a record is, in average, acceptable, the maintenance algorithm can be complicated in the case of many insertions and deletions.

Many of the difficulties that arise in maintaining a

database are a result of periodic reorganization of the records. Furthermore, files are becoming much larger as computer applications grow and storage costs drop. With very large files, periodic reshuffling of the records is time consuming and costly.

As the current techniques do not seem to provide sufficient performance, we have chosen to solve these problems by the employment of new hashing techniques that have recently appeared in the literature (see [Larson 1980], [Litwin 1980], [Litwin 1978], [Lloyd and Ramamohanorao to be published].

Among these techniques, the Larson scheme has been chosen to handle the retrieval of an entity via its primary key. This scheme has good performance in terms of insertions, deletions and retrievals. The only disadvantage is that the entities are not stored in the order of a sort key.

Unfortunately, this scheme is only a primary key scheme and does not allow the existence of secondary keys. So we use an other scheme [Lloyd and Ramamohanorao 1982] to store secondary keys. This scheme is suitable for secondary keys and allows the retrieval a set of items which have a specified value.

We have chosen to implement only the direct access method in the DBMS. Though the methods decribed below possess good performances , they can not be sufficient in a real DBMS which normally provides some other mechanisms to access the data (indexed sequential , chain , lists ...). This limitation has been introduced because the other methods can not be investigated in this thesis .

4.3. Overview of New Hashing Schemes

4.3.1. Introduction

Recently, there appeared in the literature hashing schemes which intend to provide high performance for files which grow and shrink rapidly.

Among these schemes two are reviewed in parts 2 and 3 of this section. The first scheme is suitable for files with a single key and the second is suitable for multiple keys retrieval files.
The first part of this section recalls some principles of the conventional hashing files.

4.3.2. Conventional Hashing Files

Hashing is an ingenious and useful form of address calculation technique. A simple pseudo-random function called hashing function (H) converts the item of a record (called the primary key) into a near random number and this

number is used to determine where the records are stored.

The records are stored in places called buckets. A bucket can hold one or more records and the set of buckets is called the address space of the file.

The record is inserted into the bucket H(c) (where c is the primary key), unless the bucket is already full. The search for c always starts with an access to the bucket H(c). If the bucket is full when c should be stored, a "collision" occurs.

A collision resolution method, which stores c in a bucket M such that M H(c), is then applied. The record c then becomes an overflow record and the bucket M is called an overflow bucket for c. The overflow records are often handled by a method called "bucket chaining"

Bucket chaining is the method in which overflow records are stored by linking one or more overflow buckets from a separate storage area to an overflowing bucket. Each overflowing bucket has its own separate chain of overflow buckets.

A search for an overflow record requires at least two accesses. If all collisions are resolved only by overflow records creations, as it was assumed recently, access performance rapidly deteriorates when primary buckets become full.

Fig 4.1 shows a conventional hashing file where the records are handled by bucket chaining.

Hashing is recognized as providing, in practice, the fastest random access to a file. Theoretical analysis indicates that access time to a hash table is independent of the number of records, but depends on the four factors listed below.The factors affecting efficiency are the following :

1. the bucket size
2. the load factor, i.e, the number of records stored in home buckets divided by the maximum number of records that could be stored in them
3. the hashing function used
4. the method of handling overflows

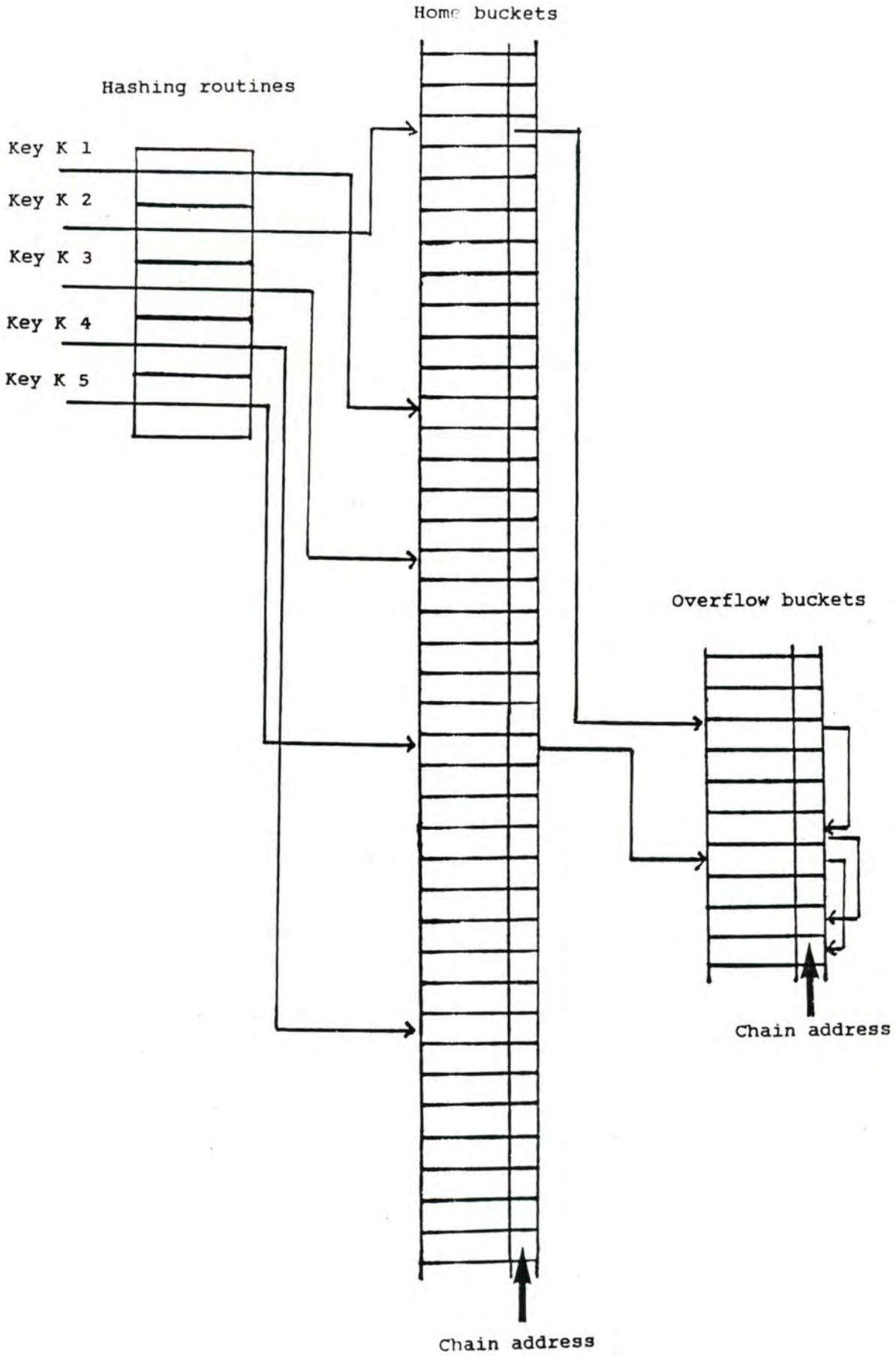


Fig 4.1 Conventional Hashing scheme

In contrast to the fast O(1) access time, hashing is burdened with two disadvantages that prevent its use in many applications :

- Hashing does not allow sequential processing of a file according to the natural order of the key (sequential processing requires a O(n * lnn) operation which makes the fast random access useless).

- Traditional hash files are not extendibles and their sizes are tied to the hash function used which often must be redefined.

  Thus, if the file grows by a large factor or if the records distribution over the available storage space is not uniform, the number of overflow records grows and the record retrieval time increases considerably.

  A high estimate of the number of records implies a costly rehashing operation (new hash function, table size, relocation of all records, etc.). Shrinkage of the file or a low estimation of the number of records implies under-utilized storage space.

If one can design adaptable hashing schemes that remain balanced as pages are added and deleted, the suitability of hashing for secondary storage devices would be greatly enhanced.

During the seventies, new file organizations which are based on hashing and which overcome the second disadvantage of conventional hashing were presented. These schemes are suitable for files whose size may grow and shrink rapidly. Their main characteristic is that the storage space allocated to the file can be increased and reduced without reorganizing the whole file.

These schemes are called "dynamic hashing schemes". The rest of this section presents two of these "dynamic hashing schemes".

4.3.3. Linear Hashing with Partial Expansions [Larson 1980]

4.3.3.1. Introduction

Linear hashing with partial expansions [Larson 1980] is a generalisation of the Linear hashing scheme developed by [Litwin 1980]. In paragraph 4.2.3.2 a brief review of the scheme proposed by Litwin is presented. Paragraph 4.2.3.3 contains a description of the improvement introduced by Larson.

4.3.3.2. Linear Hashing Scheme

In linear hashing, we have as starting point a traditional hash file where overflow records are handled by bucket chaining (see 4.2.2).

Assume that the insertion of a record with a key s leads to a collision and no records already stored in the bucket H(s) could become overflow records. The record may then be stored in a primary bucket only if a new hashing function is chosen. The new function that we shall then call H' should assign new addresses to some of the records hashed with H on H(s) and the file should be reorganized in consequence. If H = H' for all other records, the reorganizing needs to move only a few records and so may be performed dynamically.

The new function is called the dynamic hashing function. The modification to the hashing function is called a split address. The moving of some records of a bucket into a new bucket which has been added to the file is called a splitting.

The idea in Linear hashing is to use the splits in order to avoid the accumulation of overflow records and the splits are typically performed during some insertions.

Linear hashing allows a gradual increase in the storage space by splitting the home buckets in an orderly fashion : first bucket 0, then bucket 1 and so on.

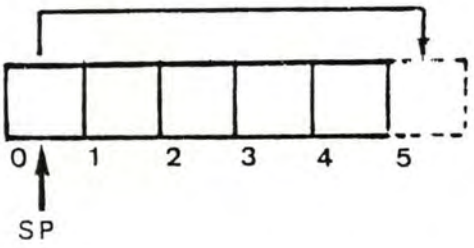
If m is the address of a collision and n the address of a split to be performed in the course of the resolution of this collision, the values of m are random while those of n are predefined (n =/ m).

We assume that a pointer P keeps track of which bucket is the next to be split. For the first N collisions, the buckets are pointed in the linear order 0,1,2,...N-1 where N is the original size of the file.

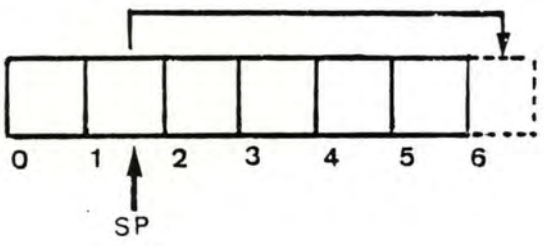
The size of the file doubles when all the original buckets have been split. In this case the pointer is reset to 0 and the process can starts again.

Fig 4.2 shows a Linear file at different stages of the splitting process (N = 5)

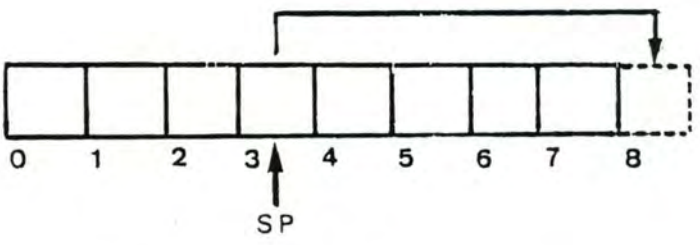
The file size becomes progressively larger, including the buckets N+1, N+2,...2N-1 one after another. A record to be inserted undergoes a split usually not when it leads to the collision, but with some delay. The delay corresponds to the number of buckets which have to be pointed while the pointer travels up, from the address indicated in the moment of



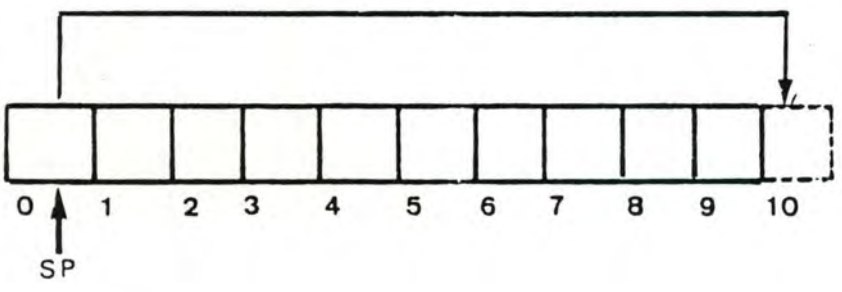
(a) At an initial stage (number of buckets=5)



(b) After one split



(c) After three splits



(d) After the doubling of the file size

Fig 4.2 Linear Hashing file

collision to the address of this collision.

With this mechanism, no matter what the address of the first collision is, let it be m1, LH performs the first split using H1 and for the address 0. The records from the buckets 0 are randomly distributed between bucket 0 and a new bucket N, while, unless m1=0, an overflow record is created for the bucket m1.

The second collision, no matter what its address is, m2 for example, leads to an analogous result, except that first it splits for the address 1 and appends the bucket N+1. Next, it may constitute the delayed split for the first collision, suppressing therefore the corresponding overflow record. This process continues for each of the N first collisions, thus moving the pointer step by step up to the bucket N-1. Sooner or later, the pointer points to each m and the splits, despite being delayed , move most of the overflow records to the primary bucket. We may therefore reasonably expect that, for any m , only a few overflow records exist.

It results from the above principles that first, the address space increases linearly and is as large as needed. Next, for any number of insertions, most of overflows records are moved to the primary buckets by the delayed splits.

After the splitting of a bucket, it should be possible to locate all the records which were moved to a new bucket without having to access the old bucket. The difficulty in this scheme is to find an algorithm which allows one to determine which records have to remain in the old bucket and which records must be transferred to the new bucket.

This algorithm must be such that approximately half the records are moved to the new bucket. The reader will find further information about this algorithm in in the appendix to this chapter.

The important point is that given the key of a record it is always possible to access the home buckets of a record without accessing any other bucket. If the record is not in the home bucket, it must be on the overflow chain emanating from the home bucket.

4.3.3.3. Improvement Proposed by [Larson 1980] : Linear Hashing with Partial Expansions

4.3.3.3.1. Introduction

Generally, the best performances are achieved in hashing techniques when the records are distributed as uniformly as possible among the buckets in the file.

Unfortunately, the record distribution of Linear hashing does not reach this goal because the load factor of a bucket already split is only half the load factor of a bucket not yet split.

The idea of Larson is to achieve an even greater load in doubling the file size in a series of partial expansions.

4.3.3.3.2. Presentation of Linear Hashing With Partial Expansion

In this scheme, the difference with linear hashing is due to the fact that the doubling of the file size is done in a series of partial expansions.

Initially the file consists of n0 * N buckets logically subdivised into N groups of n0 buckets each, n0 >= 1, N >= 1. A group i consits of the buckets (i, N + i, 2N + i ,....., (no-1) N + i) with i = 0,1,....,N-1.

Fig 4.3 shows a file at an initial stage with N = 3 and no = 3.

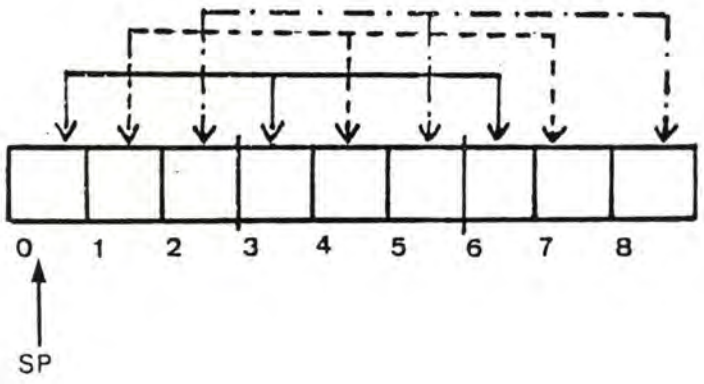


Fig 4.3 Linear Hashing file with partial expansions (N=3 and n0=3) at an initial stage

group 0 contains the buckets (0,3,6)

group 1 contains the buckets (1,4,7)

group 2 contains the buckets (2,5,8)

A full expansion results in a doubling of the file size and is accomplished by a sequence of n0 partial expansions increasing the size of each group to n0 + 1, n0 + 2 ,...., 2n0 respectively.

A partial expansion is carried out stepwise by adding one bucket to each group always in a predefined order : group 0, group 1 ,..., group N * 2 exp L - 1. Each complete partial expansion increases the file size by N * 2 exp L buckets.

The number of full expansions which have been accomplished is indicated by a variable L. The smallest file size on level L is no * N * 2 exp L buckets and the buckets (j, N 2 exp L + j, 2 * N 2 exp 1 + j ,....., (no-1) * N * 2 exp 1 + j) with j = 0, 1 ,.... ,N * 2 * exp L - 1 forms N * 2 exp L group of buckets.

Fig 4.4 shows a linear hashing file with two partial expansions (n0 = 2). The number of groups is 3 (N = 3).

In the example represented by the figure 4.4, the doubling of the file size is done in two steps; the first expansion increases the file size to 1.5 times the original size ,

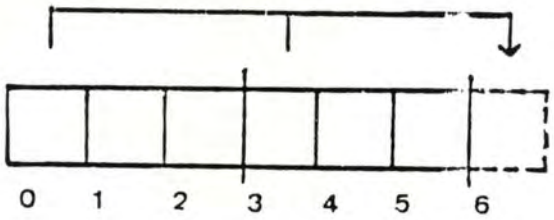
while the second expansion increases it to twice the original size.

We start with a file of 6 buckets, logically subdivided into 3 pairs of buckets, where the pairs are (j,j+N) j=0,1,2.

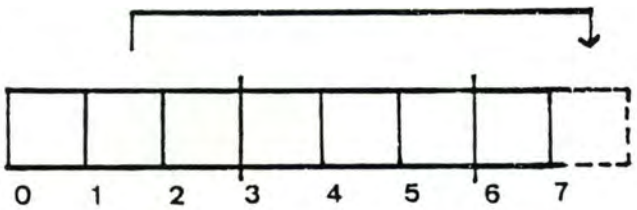
When new storage is needed, according to some rule , the file is expanded by one bucket, bucket 6, and part of the record in bucket 0 and bucket 3 are moved to bucket 6. When more space is required, the pair (1,4) is expanded.

When the last pair (2,5) has been expanded, the file size has increased from 6 to 9. Thereafter the second expansion starts , the only differences being that now 3 groups of 3 buckets are considered (0,3,6), (1,4,7), (2,5,8).

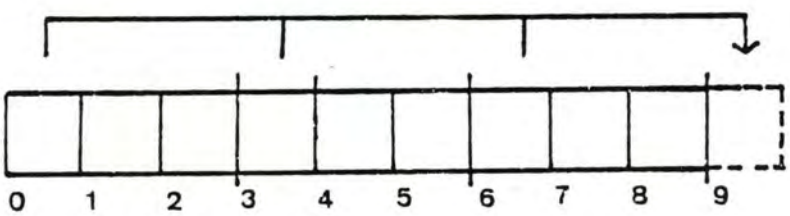
When the second partial expansion has been completed, the initial file size has doubled from 6 to 12.



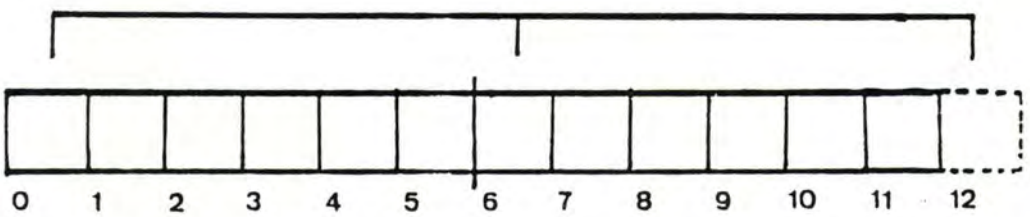
(a) At the initial stage



(b) After one split



(c) After one partial expansion



(d) After two partial expansions (doubling of the file)

Fig 4.4 Linear Hashing with partial expansions at different stages

However, we do not wish to continue expanding groups of four buckets, then five... We want to come back to groups of two buckets (j, j+6) j = 0,1,2,3,4,5 (figure 4.3).
If this were not the case, the cost of expanding a group would steadily increase and it would soon become prohibitively large.

Another important point is that when a group of buckets is expanding, it should not be necessary to rearrange records among the old buckets. We must simply scan through the old bucket and collect only the records which are to be reallocated to the new bucket.
In this way, the expansion can be made in one scan, and no jumping back and forth is necessary. The solution for this last point is quite simple and uses the rejection technique.

The rejection technique assumes that H(K) = (h1(K), h2(K), h3(K)..) is a sequence of hashing functions, where each hi hashes uniformly and independently over {0,1,2,...n-1}.
Furthermore, we suppose that a fixed set of records is to be stored in a file consisting of only $m' < m$ buckets. To find the address of a record with a certain key K, we compute the numbers...h1(K), h2(K)... and takes as the address of the record the first number which is a valid address, i.e, which is less than m' among the buckets $0,1,...m'-1$.
Once the file is extended by one bucket, m', and the address of every record in the set in recomputed, the same hashing functions h1,h2,... are used , but this time any number less than $m'+1$ is a valid address. When the number of buckets pass from $m'-1$ to m', the address of some of the records has changed to m' :

Example: if m=4 and $m'=2$, we consider the sequences

H(11)=(1,3,2,...), H(12)=(2,3,0,...),
H(13)=(3,1,2,...), and H(14)=(3,2,1,...)

If the file has 2 buckets, the records will be assigned to bucket 1,0,1,1, but if m' is increased to 3, the addresses of the records are 1,2,1 and 2. In this case, records 2 and 4 must be moved to the new bucket. If $m'=4$, the addresses are 1,2,3 and 3. This time record 3 and record 4 must be moved.

If you want to reduce the size of the file by one bucket, the new addresses are computed for the records located in the bucket which is to be deleted and the records are inserted into their new bucket. No other records have to be moved.

4.3.3.3.3. Control Function

Larson suggests that the expansion of the file should be controlled solely by the load factor When a record has been inserted into the file, load factor is checked and if it is higher than some fixed threshold w, $0<w<1$, the file is expanded by one bucket. This implies that we have a control function that keeps track of the number of records in the file and the number of overflow buckets.

This control function seems optimal because when we use Linear hashing with partial expansion, there is always a trade-off between storage utilization and the expected length of successful searches.

The higher the storage utilization is, the longer the searches are expected to be. Both factors cannot be controlled simultaneously. Larson suggests that we first control the storage utilization by requiring that it should always be $>=$ the threshold, but once this threshold has been reached, we minimize the time of searches by keeping the storage utilization as close to the threshold as possible.

4.3.3.3.4. Performance

A detailed performance analysis can be found in [Larson 1980] The analysis reveals that an average search length in the range of 1.1-1.2 accessses can be achieved with the same parameters.

Furthermore , we can say that the expected number of accesses required to insert a record also includes accesses required to physically store a record and to update the record counter, the accesses required to rearrange records in the old bucket reach between 4.37 and 6.37 accesses for load storage as high as 85%-90% and a bucket size of 50 records. The choice of two partial expansions seems to be a good compromise.

In summary, Linear hashing with partial expansions offers a new and simple technique for organizing dynamic files. Retrieval of a record is very fast by any standard and files have a constant storage utilization up to 0.90 with excellent performance. The performances deteriorate rapidly if the storage utilization is further increased.

4.3.4. Dynamic Hashing Scheme for Secondary Key File [Lloyd, Ramamohanorao and Thom 1983]

4.3.4.1. Introduction

The scheme presented in the section above has the disadvantage of letting the user access a record only by means of a primary key.

In the case of secondary keys (we want to find the location of a record via keys which do not uniquely identify one record), the pure hashing schemes are not very efficient. To solve this problem, we will describe a partial match retrieval scheme based on hash functions and descriptors See [Lloyd 1980],[Lloyd and Ramamohanorao 1982], [Lloyd, Ramamohanorao and Thom 1983], [Pfaltz, Berman and Cagley 1980].

4.3.4.2. Definition of a Partial Match Query

Each record in the file consists of a number of fields (secondary keys) which may be specified in a query.

Assume that there are K fields f1,....,fk which may be specified in the query. Then a partial-match query is a specification of the value of one or more of the fields f1,...,fk. An answer to a query is a listing of all records in the file which have the specified values for the specified fields.

4.3.4.3. Description of a Simple Partial-Match Retrieval Scheme (When the File is Known) Based Purely on Hashing

The records of the file are contained in a number of pages. We suppose first that the file is static and consists of 2 exp(d) pages. (d is a fixed, non-negative integer). The pages are numbered 0,1,...2 exp(d)-1.

There are K hashing functions hi, the ith function mapping from the key space of the field fi to the set of the strings of di bits where each di is a non-negative integer and d1 +...+ dk = d.

The page in which a particular record is to be stored is computed as follows. Each field fi is hashed to a string of di bits. The string resulting from the concatenation of these strings (in order) gives the page number.

The hashing functions should be chosen to distribute the records as evenly as possible amongst

the pages.

The problem at this stage is to minimize the average number of home pages which have to be accessed to answer a query.

Let Q be a query, so that Q in 1,2,...,k. We denote by the probability that the query is specified. The P_Q's for a particular system are determined by the use made of that system.

Then the average cost of a query is :

$$A = \sum_{Q} P_Q \left(\prod_{i \in Q} 2^{di} \right)$$

The optimization problem is thus to find d1,...,dk that minimize the objective function A and satisfy d1 +...+ dk = d.

As such, it is too simplistic because another difficulty arises with the key space of each field.

For example, often fields have rather small key spaces, and thus must be allocated only one or two bits. Constraints of the form di <= dmax naturally arise where 2 exp(dmax) is the number of values that a particular field can take. For example, sex can have only two values and thus no more than one bit should be allocated to it.

On the other hand, if the field f1 has a large key space of 2 exp(c1) values, the optimization problem assigns d1 bits to f1 with c1>>d1. A pure hashing scheme cannot cope with it because the total number of bits, d, allocated is determined by the size of the file space and there are a number of fields competing for bits.

As the number of bits allocated to f1 is severely limited, the hash function will map many different values of the same field in the same bit string. A large amount of information is lost in this case. One suggestion is to use a small, simplified, descriptor file, built on top of a hashing scheme, so that before any page is accessed, a check on its descriptor is made. A scheme using descriptors was developed by [Pfaltz, Berman and Cagley 1980].We briefly review this scheme.

4.3.4.4. A Descriptor Scheme

4.3.4.4.1. Desciptor, Page Descriptor, File Descriptor

At each file is associated a descriptor file. This decriptor file is composed of a set of descriptors pages.

A descriptor page is a set of descriptors where each descriptor is related to a page of the main file. A descriptor is simply a bit string of W bits (fixed length).

Each record R in the data file has a descriptor Dr associated with it. This descriptor is derived from the values (V1,V2,...,Vk) of the k attributes of the record R.

Fig 4.5 shows the basic scheme of a file and its associated descriptor file.

4.3.4.4.2. Constructing a Descriptor

One possibility in constructing the descriptor is to employ the method of disjoint coding [Pfaltz, Berman and Cagley 1980].

Disjoint coding begins by dividing each descriptor into F disjoint fields (each record has F attributes). Each field Fj consists of Wj bits and the sum of all values Wj from j=1 to j=f is equal to W.

Each of the attributes has an associated transformation Tj which maps from the key space of Fj to the subset of bit strings of length Wj.

To describe a record R, these transformations are applied to each of the attribute values of R and the Tj(vj)th bits in Wj is set to 1 while the remainder Wj-1 bits are set to 0.

Each descriptor will have exactly F bits set to 1. In a partial-match query, attribute values are specified for only a subset of the attributes. If Q<=F is such a subset, the query descriptor is built in the same way as the record descriptor.

The transformation Tj is applied to the attribute value Vj to determine which bit in Wj(Q) is set to 1. If all the possible values are not specified in the query, the bits corresponding to the fields not specified are set to 0.

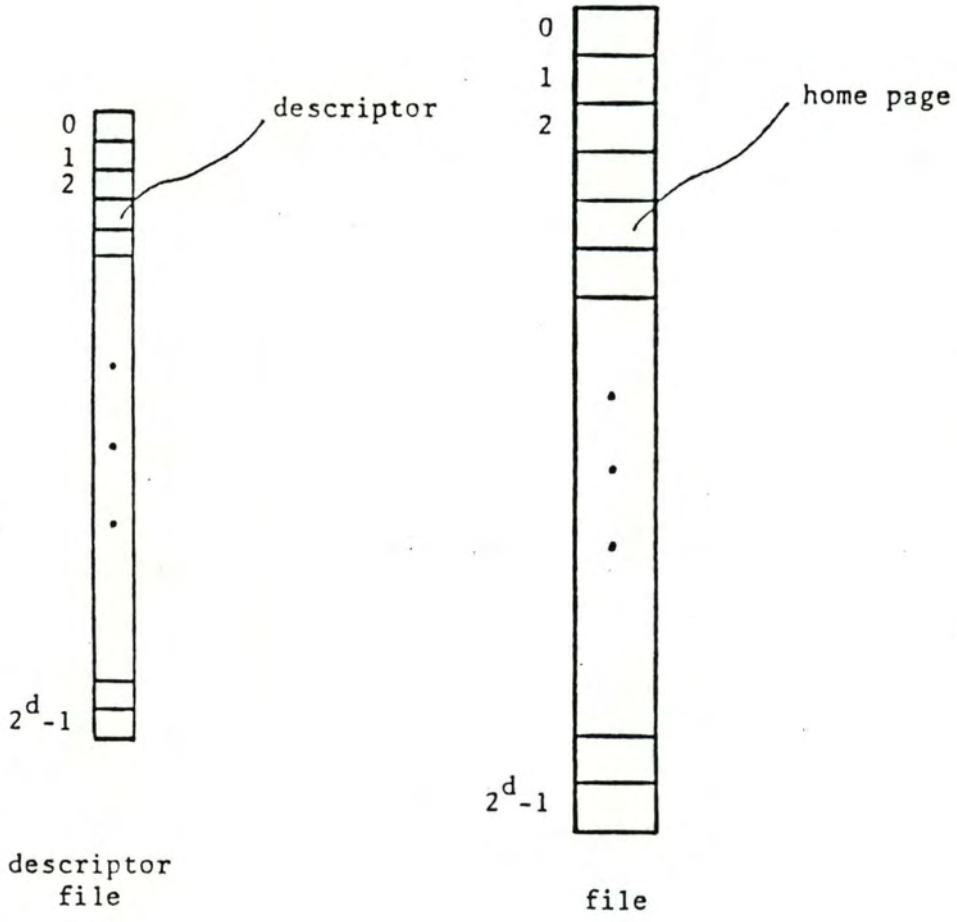


Fig 4.5 Basic scheme

As the descriptor of Dr and Q have been constructed using the same transformation, we can make the following propositions :

- If R satisfies the partial match query, then Q <= Dr.

- If Q is not a part of Dr, then R does not satisfy the partial match query.

- If Q is a part of Dr, then R may or may not satisfy the query.

"Q is a part of Dr" means that every bit position which is 1 in Q is also a 1 in Dr, and " Q is not a part of Dr" means that there is at least one bit position which is 1 in Q and 0 in Dr. With these propositions, we can construct a descriptor file which allows us to check if an information is contained in a page and thus access this page only if we are sure that we can find the information in this page.

In the scheme proposed by [Lloyd, Ramamohanorao and Thom 1983] ,a page descriptor is constructed by applying the logical function OR to the descriptors of the records contained in the pages of the main file and any overflow page.

4.3.4.4.3. Using a Descriptor File

The descriptor file is used as follows. Let Q be a query. Using the hash function Hi on the specified fields, a set of addresses of pages which can contain records in the answer to the query is generated.

However, before these pages are accessed, we check the descriptor file. Corresponding to Q, there is an associated query descriptor (with the same structure as a page descriptor), which is obtained by transforming the fields specified in Q using the Ti's and making up the remainder of the query description with 0's in the bit positions corresponding to the unspecified fields.

Then before accessing a page, we compare the query descriptor with the descriptor for that page. If the query descriptor has a 1 in a bit position where the page has 0, then the page cannot possibly contain a record in the answer to Q and hence, the page does not have to be accessed.

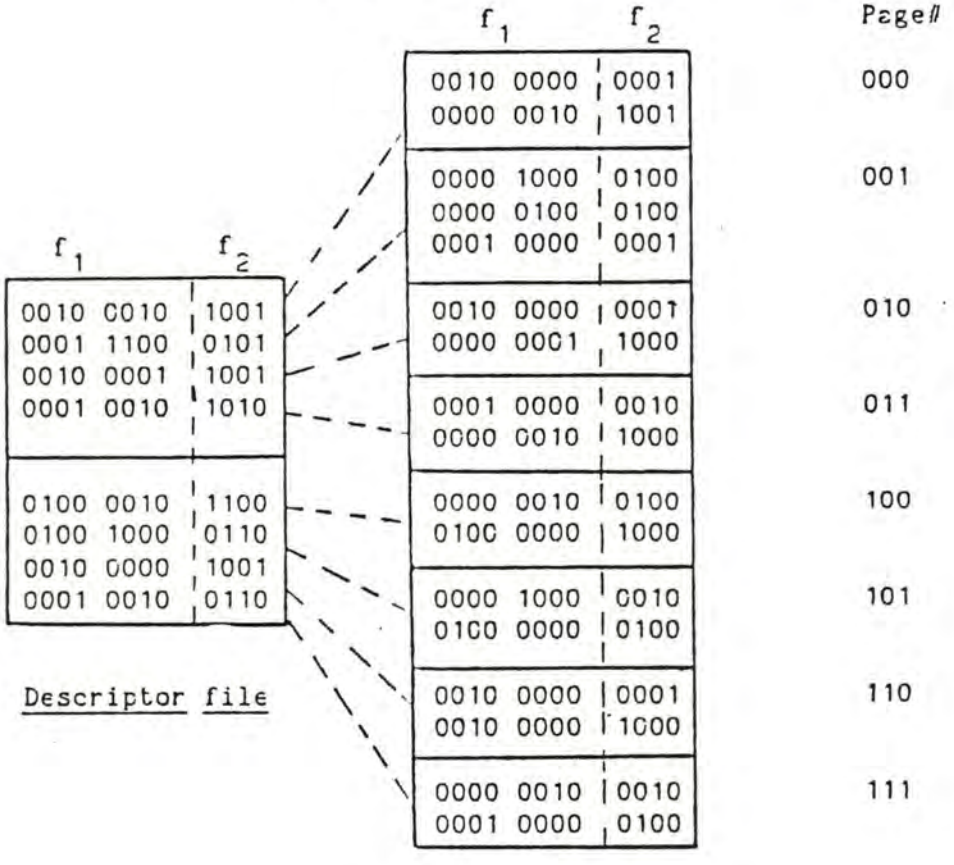
The advantage of descriptors is that they can add more knowledge about records actually present in the file. When a record is added to or deleted from a page, the descriptor must be updated.

Example:

Assume that we have a record type with two fields and that a query was made on the first field of this record type with the value V1.

If only one bit is allocated to this field then the hashing function can hash to the set of addresses beginning by 1 or 0. If we assume that the hashing function hashes to 1 then the set of home pages to be searched without the descriptor file is the following : pages 100, 101, 110, 111.

Suppose the query Q gives the query descriptor 001000000000. Page 100, which has a descriptor 010000101100 does not have to be accessed, since it has a 0 in a bit position where the query descriptor has a 1. On the other hand, it will be necessary to access page 110 with descriptor 001000001001.



Main File

d=3 k=2 d1=1 d2=2 W1=8 W2=4 W=12

It has to be noted that record descriptors are shown in the main file only for explanation purpose.

4.3.4.5.1. Presentation of the Scheme

The partical-match retrieval scheme described in the section above is only suitable for static files, but it is easy to extend it to dynamic files by utilizing the linear hashing scheme discussed earlier. The scheme used here is linear hashing.

A Linear Hashing file is shown at a typical stage in its existence in fig 4.6

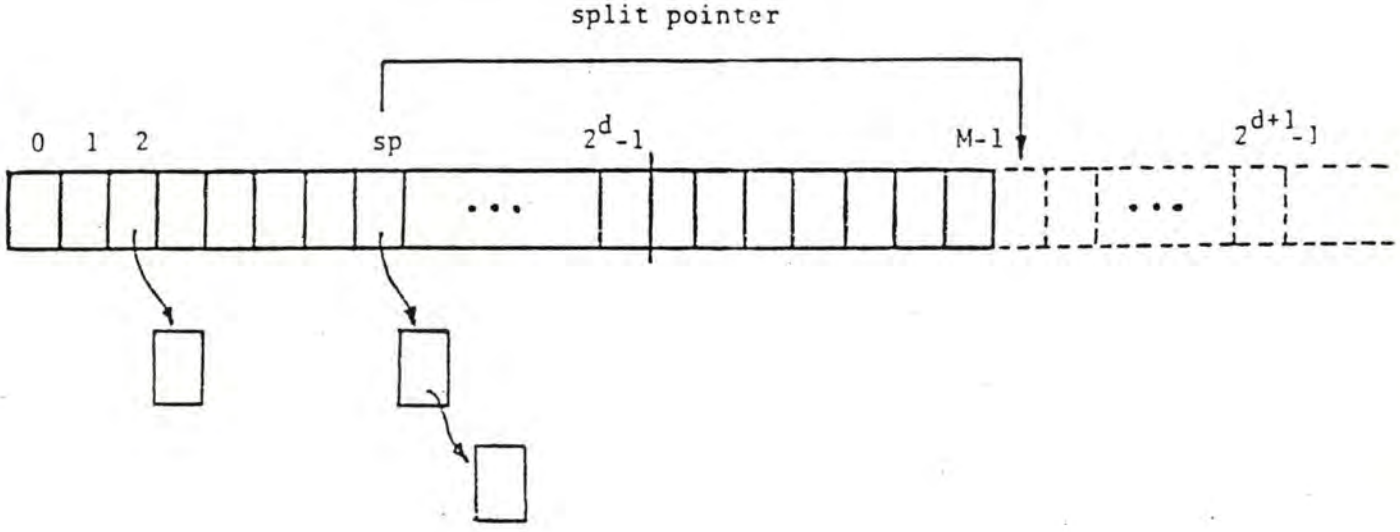


Fig 4.6 Typical stage of a Linear Hashing scheme

The file is currently stored in M pages, numbered from 0 up to M-1. Some pages have short overflow chains containing records that would not fit into the home pages.

The file also has a split pointer which indicates the next page to be split. This page is numbered sp. Note that the page to be split is independent of where collisions occur. The split pointer moves in a very systematic way, first page 0, then page 1 ,... splitting each page in turn.

When page 2 exp(d-1) splits, the split pointer returns to page 0. On the next run, the split pointer will go up to 2(exp(d+1))-1. Thus, the file doubles during a complete run of the split pointer. d is now a variable called the depth of the file.

The splitting policy employed here is to

split after every L insertions into the file. L is called the load control and must be carefully chosen to maintain a desired load factor.

When the page sp is split, the following occurs :

- For each record in the page sp, and for any associated overflow records, a new hash address is computed.

- For each such record, the new address will either be sp (the old address) or M=SP+2 exp(d). A new page, numbered M, is then appended to the end of the file and the records with hash address M are put into this page.

Since pages split in this very systematic way, the need for a directory is obviated. Furthermore, even though the file may have grown and the record moved since it was first inserted, it is still possible to calculate directly the home page of any record in the file.

The important part of the extension is the choice of the hash function. A more complicated way of constructing the hash function is needed because the file is no longer static.

For each Fi, we have a hash function Hi mapping from the key space of Fi to bit strings of an suitably length. We now no longer concatenate the strings Hi(Vi) as before, but we compute the new address with what we call the "choice vector ".

4.3.4.5.2. Choice Vector

We are now explaining what a "choice vector" is and how to use it to compute the address of a record. After, we will see from where it comes.

A " choice vector " contains numbers (I1, I2 ,.....) which are integers between 1 and K. Each integer indicates which field is taken into account to compute the address of a record. Let us see with an example how this "choice vector " is used :

Example :

Assume the choice vector is (4,5,4,3,2,3...) and K=6.
The right-most bit of the bit string forming the address is the first bit in the string H4(V4), the second from right is the first bit in the string H5(V5), the third is the second bit in the string H4(V4), and so on.

In general, the ith bit from right in the bit string forming the address will be the first so far unused bit in the string Him (Vim). In this case, the record is said to hash to an interlaced bit string.

This choice vector comes from the optimization problem. The problem is to minimize :

$$A = \sum_{Q} P_Q \left(\prod_{i \notin Q} 2^{d_i} \right)$$

subject to $\sum d_i = d$ where each di is a non negative integer.

[Lloyd, Ramamohanorao and Thom 1983] proposed in their paper an algorithm which computes the optimal number of bits and the optimal Di and Wi at each depth.

This algorithm is not used in the current system because it is complex and not sufficiently explained in the paper. We use a rather simpler way to compute the optimal bits. We say that there are three levels of probability that a query was made on a specified value of a item : high, middle and low. We allocate bits proportionally to these probabilities. For example if four bits must be allocated between 2 items declared as having a high and low probability respectively, then 3 bits will be allocated to the first and 1 to the second respectively.

What can happen is that a particular field's allocation of bits in the optimal solution at one depth can be higher than its allocation at the next highest depth.

This implies removing a bit from the middle of a hash address when the depth changes, but in this case we can avoid a complete reorganization of the file during the change in depth to handle this.

Thus, the allocation of di values, as the depth increases, should have the following property : If di bits are allocated to a field fi at depth di and di' at depth d + 1, then di <= di'. This property is called the monotonicity property.

The algorithm provided in [Lloyd, Ramamohanorao and Thom 1983] computes also the "choice vector".

4.3.4.5.3. File Descriptor

The descriptor file grows and contracts in parallel with the linear hashing file. However, no matter what the depth of the linear hashing file, the descriptor size is a constant W bits.

The construction and the information appearing in the descriptor are similar to those of section 4.3.4.4. Maintenance of a descriptor is also easy and is made in parallel with the maintenance of the LH file. Fig 4.7 shows a Linear hashing scheme and its descriptor file .

4.3.4.6. Performance

The descriptor of a page must be updated whenever a record is inserted into a linear hashing file. This involves computing the descriptor of the record and applying the logical function OR to the old descriptor associated to the page where the new record has been inserted and the new descriptor.

If a record is deleted the cost is slightly more expensive because the descriptor must be recomputed.

The cost of maintaining the descriptor file for an insertion or deletion is 2 disk accesses : one to read the descriptor, and one to write it. For a split, the cost is 4 disk accesses : one to read the descriptor, one to read the page and two to write the two new descriptors.

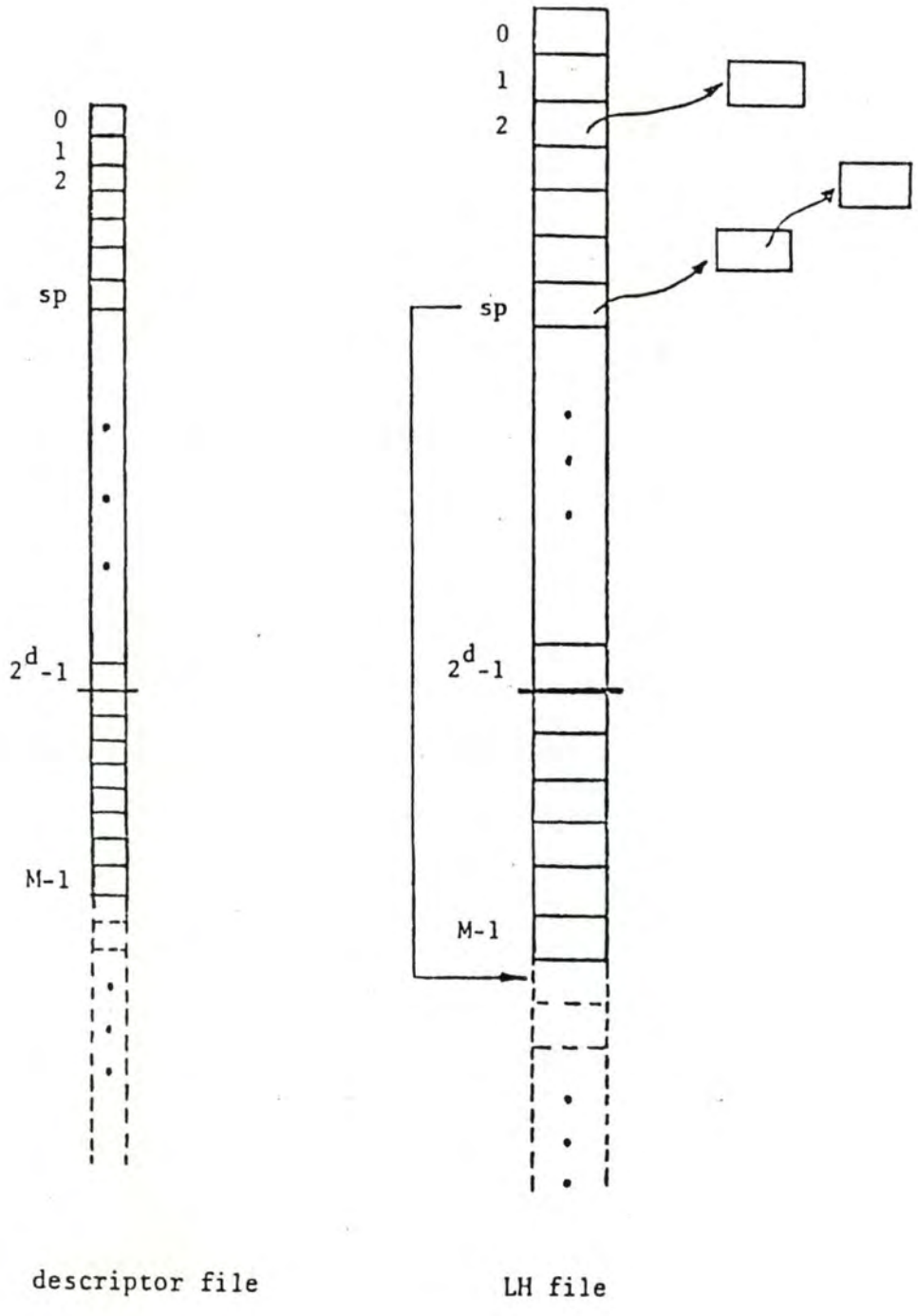


Fig 4.7 Linear Hashing scheme and descriptor file

4.4. How to Use These Schemes to Store Entities and Relationships

The first scheme (see section 4.3.3) presents good performance for files with primary keys. So all the primary keys of one entity set will be stored in one file which will be handled by the scheme of Larson.

The secondary keys and the attribute part will be stored in another file and will be handled by the second technique presented above. The relation between the primary key and the secondary keys is illustred by the figure below :

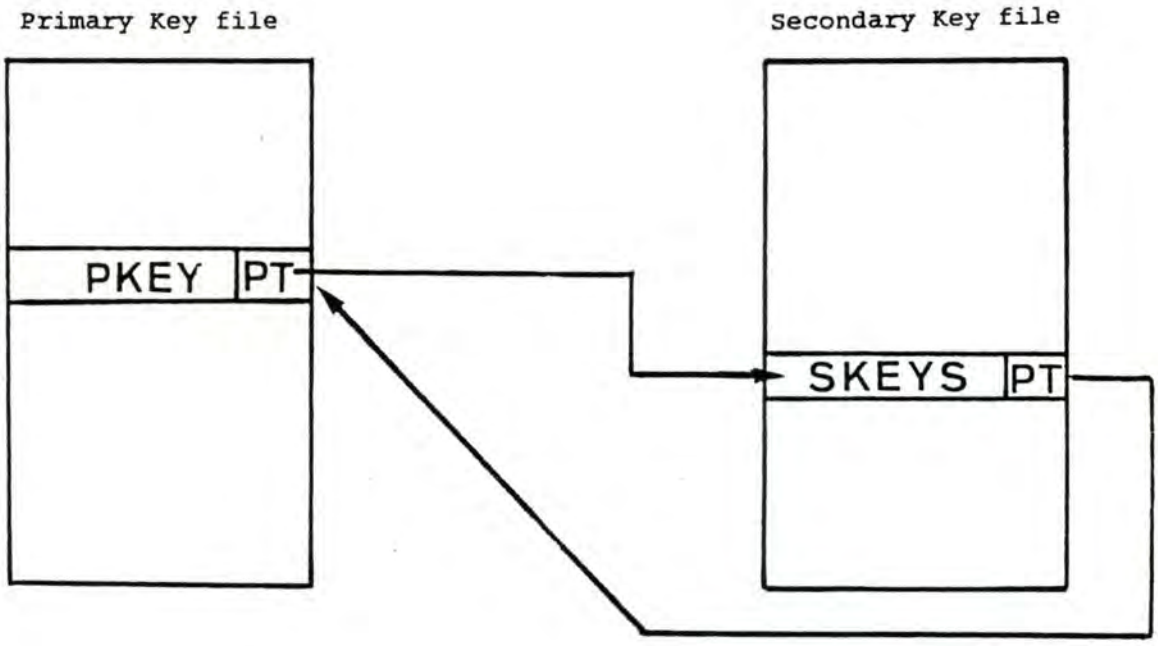


Fig 4.8 Relationship between a primary key file and its corresponding secondary keys

In this way with a primary key, you can access via a pointer all the other attributes of the entity. If you have accessed one secondary key you can also access its primary key.

The relationships are also implemented following the second scheme. When we want to retrieve one relationship we give all the primary keys of the entities implied in this relationship and thanks to the second scheme the relationship can easily be retrieved. We can also The following figure illustrates this process.

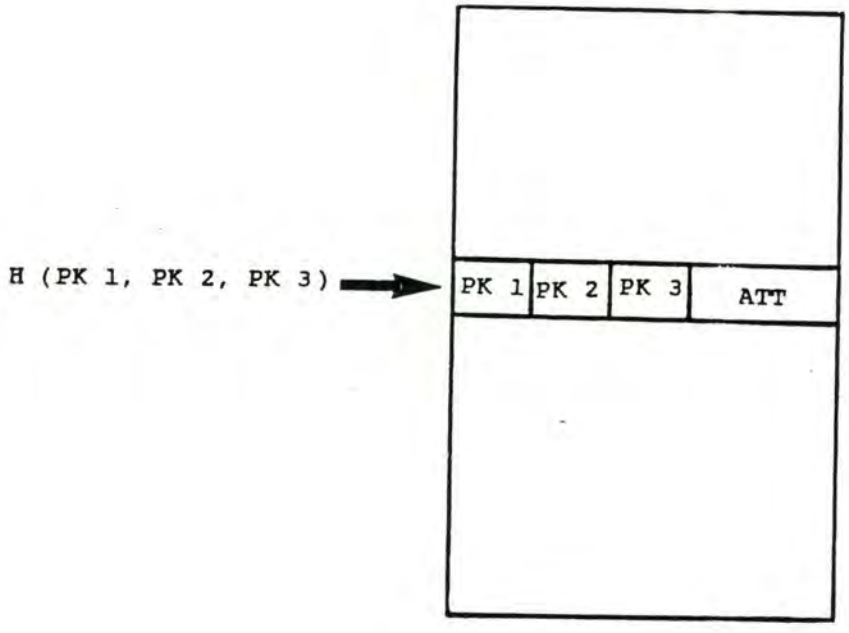


Fig 4.9 Relationship file

Even if this second scheme provides good performance, it seems that a combination of these techniques and pointers would be more efficient especially for queries of the type : we want to have all the relationships in which an entity is implied. In this case, once the primary key has easily (thanks to the first scheme) been retrieved, we followed the chain of pointers to have all the instance of the relationship which have this entity relied on.The second scheme will be used only to store and delete the relationship. The following figure illustrates this process.

We have chosen to implement only the two first schemes. The scheme with the pointers must be studied much more deeply before being implemented. In particular, we must verify that the update of the pointers is not too high a price to pay to improve the retrieval performance.

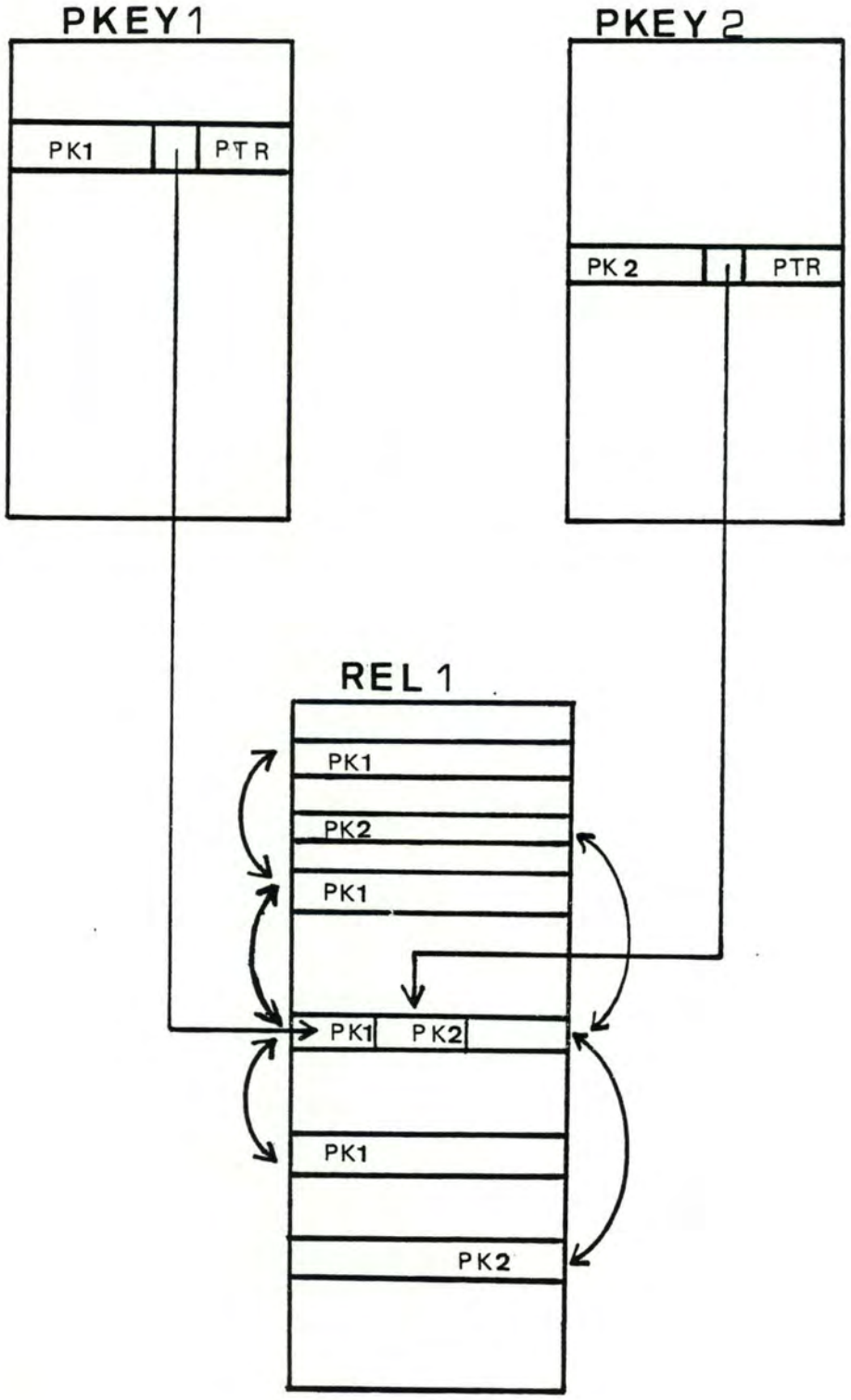


Fig 4.10 Combination of schemes

4.5. DBMS Overview

4.5.1. DBMS Structure Overview

The Database management system that we want to build consists of four parts :

1. the database files (DBF), which consist of the data that is to be accessed
2. the database tables (DBT), which contain the logical description of the structure of the database and is stored in the beginning of the DBF
3. the database control system (DBCS) which consists of a collection of subroutines that can be called from FORTRAN This is the actual programmed interface to the database.
4. The database utility routines which aid the analyst in creating and maintaining a collection of databases

4.5.1.1. The Database Tables (DBT)

The Database Tables (DBT) are generated by a program named DDLA whose input is a database description written in the Data Description Language (DDL).

The DBT consists of six description tables (ETNTAB, RTNTAB, TYDTAB, HPKTAB, HATTAB,OPATAB) and a character vector of DDL names (NAMES). They are placed in the first pages of the DBF by DDLA.

4.5.1.2. The Database Files (DBF)

The information stored in the database is placed by the DBCS into the DBF.

With each entity type is associated two files. One contains the primary key of each instance of the entity type and a pointer to the secondary key related to the primary key.The second file contains the secondary key of each instance of the entity type and a pointer to the primary key related to the secondary key. Finally there is a descriptor file associated with each secondary key file. With each relationship set is associated 2 files: a file which contains all the relationships and a descriptor file. The DBF consists of physical pages which depend on the computer installation. The DBF are initialized by DDLA.

4.5.1.3. The Database Control System (DBCS)

The DBCS is a collection of Fortran routines which interface with the user's program and with the DBMS utility programs. They are divided into five groups, classified by function. The five groups are :

1. DBHUSE
2. DBHLOW
3. DBHTAB
4. DBHRAN
5. DBHLIB

The DBHUSE routines are the only routines that directly interface with the user's program.

The DBHLOW routines are the lower level routines used by DBHUSE to access the database; they also contain the programs for the database storage allocation system and for the database management system.

DBHTAB is a collection of Fortran integer functions which return control block fields, and Fortran subroutines which update the control block fields. They are heavily used by DBHUSE in referencing the database tables.

DBHRAN consists of random input/output routines used by DBHUSE and DBHLOW to transfer pages of the database between main memory and the DBF.

DBHLIB consists of routines which allow one to manipulate the buffer in which are stored the pages of the DBF. These routines can store and retrieve strings, logical value, words and halfwords. They can also compare strings and words. They are used by DBHLOW and DBHRAN.

4.5.1.4. DBMS Utility Programs

There must be two utility programs available for use with DBMS. Each must be a stand-alone program which calls routines of the DBCS. The programs are :

1. The Data Description Language Analyzer (DDLA)

   This program generates the database tables (DBT) from a DDL and put it into an initial database. It also produces a Fortran block data source subprogram which is used by the DBCS.

2. The Database Summary Program (DSUM)

   This program must generate statistics on the number of instances of each entity and relationship types as also statistics on the load

of each file.

The figure 4.12 shows the structure of the system.

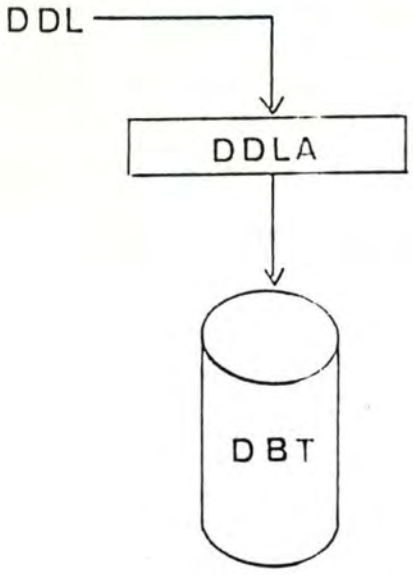


Fig 4.11 Generation of tables by DDLA

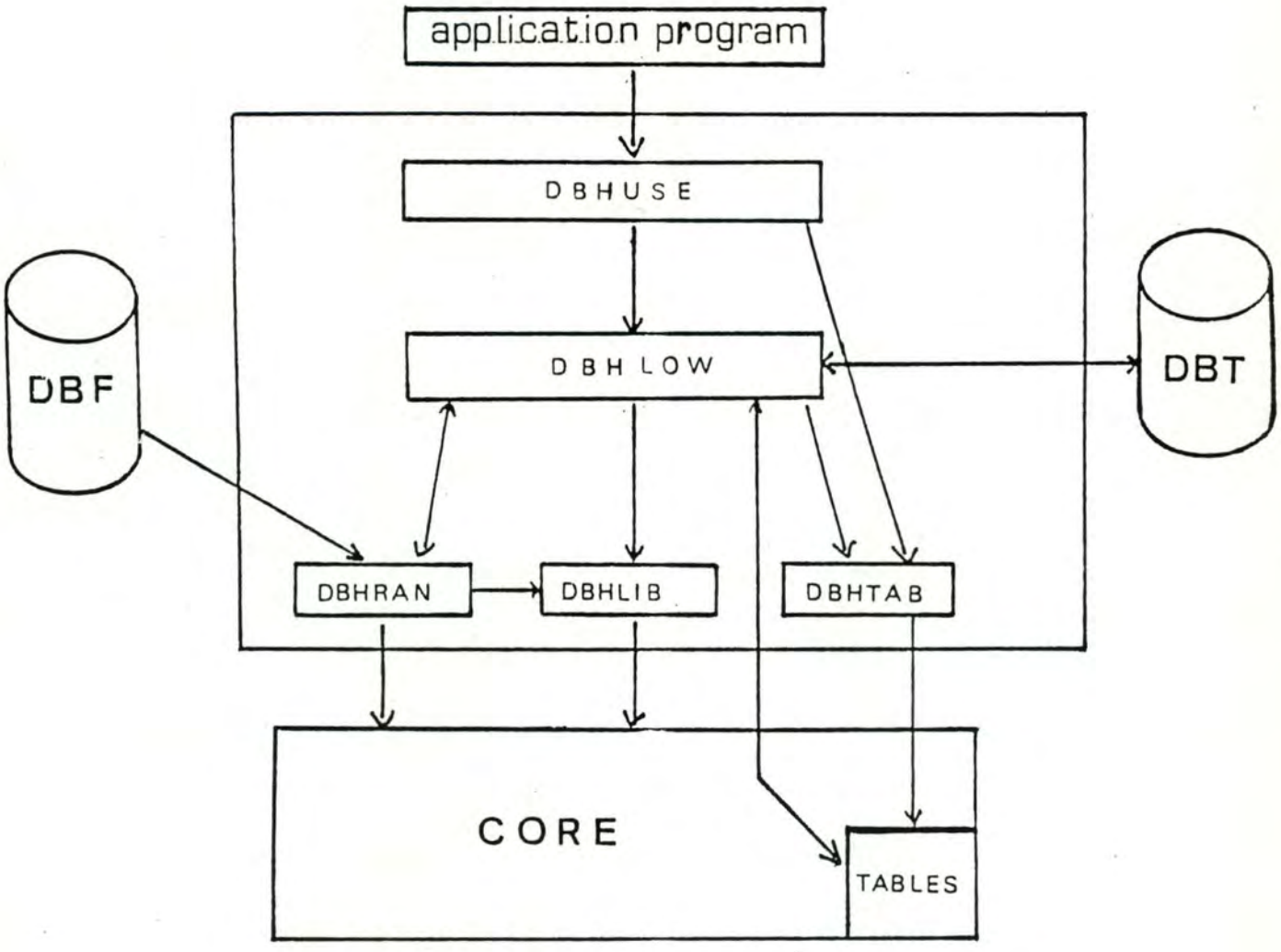


Fig 4.12 General structure of the system

4.5.2. DBMS Structure

4.5.2.1. The Database Tables (DBT)

The database tables contain the description of the structure of the database. There are six description tables : ETNTAB, RTNTAB, TYDTAB, HPKTAB, HATTAB, OPATAB.

The first three tables listed above are tables which describe the logical description of the database. The last three tables contain parameters which are used to handle files in the data base. They will be explained later.

The following figure represents the relationship between the entity type names table (ETNTAB) and the type description table (TYDTAB) containing the description of the entity types.

The relationship between the relationship type name table (RTNTAB) and the type description table (RTDTAB) has exactly the same structure as the above figure.

To obtain any information about an entity type, the name of this entity type is hashed in the table containing all the entity type names.

If the name in the table does not correspond to the entity type name passed, then we access the overflow area to find the right name. If the name is right, we access, via a pointer, the description of this entity type in the type description table (TYDTAB) and we can get any information concerning this entity. If the name is not right we follow the overflows chain until we have found the right name or the chain of pointers is terminated. The same method is used to access the information concerning a relationship type.

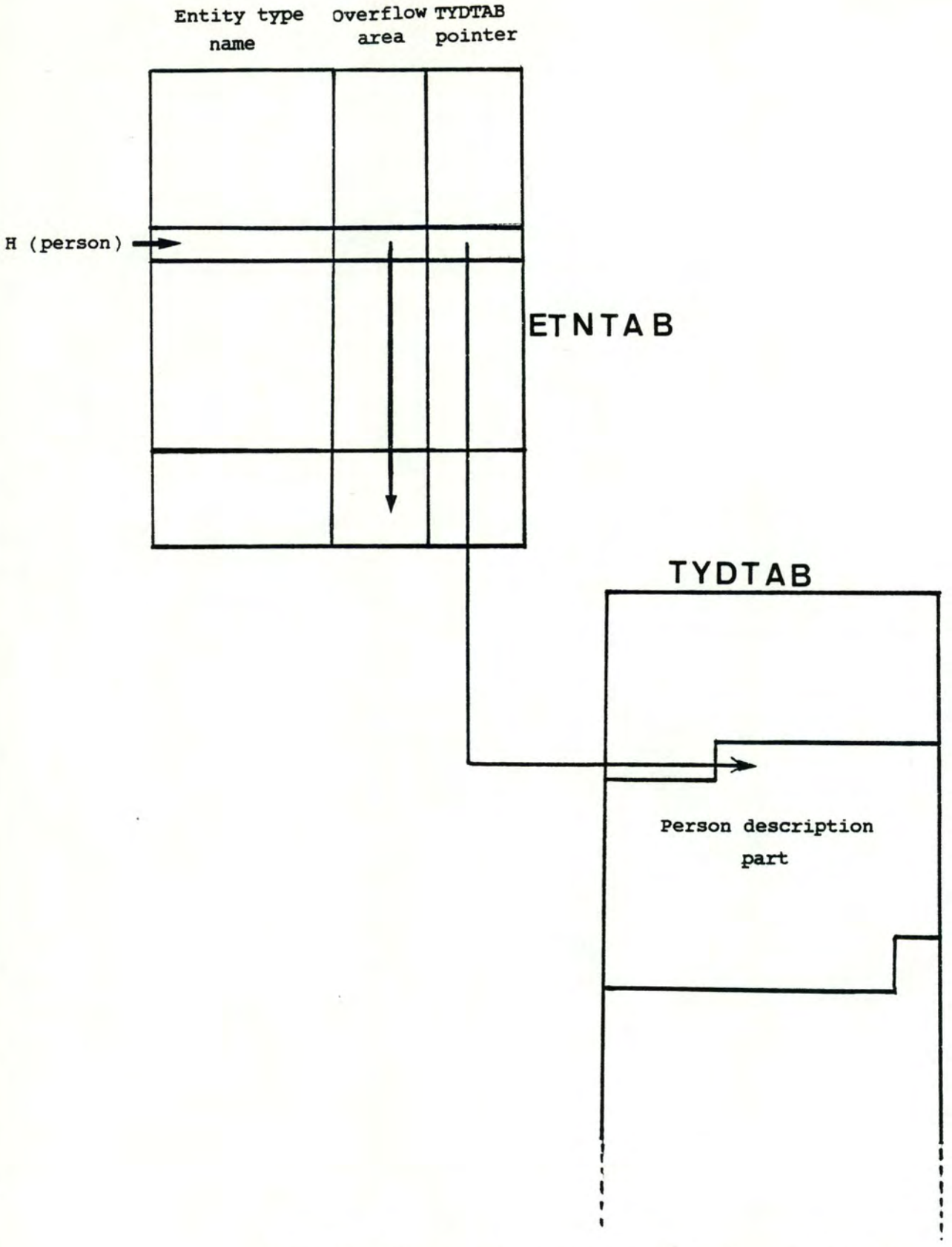


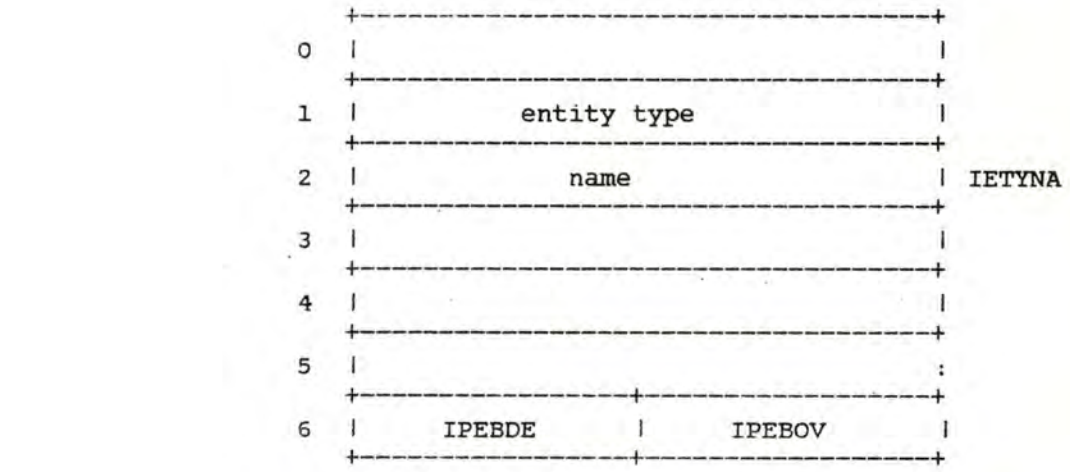
Fig 4.13 Relationship between the ETNTAB and the TYDTAB

4.5.2.2. The Database Entity Type Name Table (ETNTAB)

The database entity type name table is made up of entity name description block (IEB). There is one IEB for every entity type described in the DDL.

4.5.2.2.1. Entity Name Description Block (IEB)

An IEB consists of seven contiguous full words of information to describe an entity type name in the database. The six first words are used to store the entity type name (which has a maximum of 30 characters). The next halfword is a pointer in the type description table which corresponds to the description of the entity type in the ETNTAB table. The second half word is an index in the overflow area in cases where two names hash to the same address in the table.



Field Name	Size	Description
IETYNA	6 words	Entity Type Name (30 characters)
IPEBDE	Halfword	Pointer to the description of an entity type in the description table
IPEBOV	Halfword	Pointer to the overflow area of the ETNTAB table

4.5.2.3. The Database Relationship Type Name Table (RTNTAB)

Like the entity type name table, the relationship type name description table is made of relationship name description blocks (IRB) which have the same configuration as an IEB; only the field names (IRTYNA, IPRBDE, IPRBOV) are different.

4.5.2.3.1. Relationship Description Block (IRB)

An IRB consists of seven contiguous fullwords of information to describe a relationship type name in the database and its associated pointers. The structure and the information are the same as for an IEB.

Field Name	Size	Description
IRTYNA	6 words	Relationship type name (30 characters)
IPRBDE	Halfword	Pointer to the relationship description table
IPRBOV	Halfword	Pointer to the overflow area of the RTNTAB table

4.5.2.4. The Database Type Description Table (TYDTAB)

The database type description table (TYDTAB) is made up of primary key description blocks (IPKB), secondary description blocks (ISKB) , attribute part descriptions blocks (IAPB), relationship description blocks (IREB), attribute description blocks (IADB), value set description blocks (IVSB), allowable value description blocks (IIRB or ICHB), and relationship part description blocks (IERB).

There is one IPKB, ISKB, IAPB for each entity described in the DDL. There is one IREB for each relationship described in the DDL. For every attribute associated with an entity type or a relationship there is an IADB. Each IADB is followed by one or more IVSB which decribe the value set type belonging to this attribute. Each IVSB can be followed in turn of one or more IIRB or ICHB. These blocks decribe the values allowed for each value set. If all the values are possible than there is no IIRB or ICHB block.

The IPKB, ISKB, IAPB, IADB, IVSB, IIRB, ICHB, IREB, IERB are stored in the order in which they appear in the DDL.

For each entity there is first one IPKB. This IPKB is followed by one or more IADB, IVSB, IIRB, or

ICHB. Following the IADB, IVSB, IIRB, or ICHB for one IPKB, there is an ISKB which is the secondary key description block. This ISKB is itself followed by one or more IADB, IVSB, IIRB, or ICHB in the same way as for an IPKB.

After the description of the secondary key there is the description of the attributes and value sets in the same way as for the primary key.

Following the secondary key description part is an IADP desciption block. This block describes the part of the entity (attributes) for which no special techniques have been implemented. The attributes and value sets are described in the same way as before.

For a relationship type there is one IREB followed by some IERB. After the relationship part , there is one IAPB followed by one or more IADB, IVSB , IIRB, ICHB as for an entity type.

Inside the TYDTAB table, the user can access either the primary key description, the secondary description or the attribute part description. The user can also access the next attribute description of an attribute description and the next value set description of a value set description. For this purpose, the length of each description is given in IPKB, ISKB, IAPB, IADB, IVSB.

4.5.2.4.1. Primary Key Description Block (IPKB)

An IPKB consists of two contiguous fullwords of information used to describe one primary key description part in the database. The physical structure of an IPKB is the following :

There are four halfword integers of storage contained in an IPKB.

Field Name	Size	Description
IPKBLD	Halfword	Length of the description of a primary key (words)
IPKBNA	Halfword	Number of attribute for the primary key

IPKBID	Halfword	Index of the entity description
IPKBLP	Halfword	Length of the primary key on a page

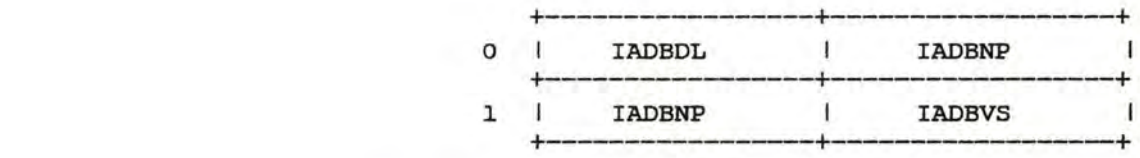
IPKBLD is the length in words of the description of a primary key. This length is equal to the following expression :

> number of attributes for the primary key * the length of each of the attribute description part.

IPKBNA is the number of attributes for the primary key. IPKBID is the sequence number of the description of an entity type in the table. For example if an entity type had been inserted after six other entity types IPKBID will have the sequence number 7. IPKBLP gives the length in words of a primary key. This sequence number is used to access the parameters in the hashing tables.

4.5.2.4.2. Attribute Description Block (IADB)

One IADB consists of two contiguous fullwords of information used to describe one attribute type in the database. The following diagram describes the physical structure of an IADB :



There are two half-word integers of storage contained in IADB. The following table describes the meaning of each storage location :

Field Name	Size	Description
IADBDL	Halfword	Length of the description of an attribute (words)
IADBNL	Halfword	Length of an attribute name (BYTES)
IADBNP	Halfword	Attribute name pointer
IADBVS	Halfword	Number of value set

IADBDL represents the length of the description of

IERBMI	Halfword	Minimum number of instances
IERBMA	Halfword	Maximum number of instances

IERBNP represents the pointer into the character array NAMES, where all the role names of the entities related by this relationship are stored. IERBNL is the role-name length.

IERBPD is a pointer into the type description table where all the entity type descriptions are stored. This entity corresponds to one of the entity types associated to this relationship.

IERBMI, IERBMA represent respectively the minimum and the maximum number of occurrences a particular entity type in a relationship type. IERBPC is the probability that the entity is specified in a query.

4.5.2.4.9. Character Array NAMES

The fourth table that is generated from a DDL is a character array where the names of the all the attributes, value sets, allowable values ,and roles are stored. The array is used by the database routines when searching the database for a particular type.

The index into the table is not kept in one place; rather each IADB, IVSB, ICHB, IERB has a name pointer field and a name length field. The name pointer is the displacement (in bytes) into NAMES where the name begins, and the name length field has the value of the name length (also in bytes).

The following table illustrates the same types that are stored in NAMES, and where the name pointer and lengths are stored.

Name	Type Name	Pointer Name	Length Where Stored
Attribute	IADBNP	IADBNL	IADB
Value set	IVSBNP	IVSBNL	IVSB
Allowable value	ICHBNP	ICHBNL	ICHB
Role	IERBNP	IERBNL	IERB

4.5.2.4.10. DBT Structure as Written by DDLA

This section will describe the physical order of the database control blocks, as written into the first page of the DBF by DDLA. There are eight logical records of table information :

1. The object schema parameters
2. ETNTAB
3. RTNTAB
4. TYDTAB
5. HATTAB
6. NAMES
7. HPKTAB
8. OPATAB

4.5.2.4.11. The Object Schema Parameters

There are a certain number of parameters which must be contained in the first line of the DBT.

The object schema parameters, as written into the DBF, are: ETNLEN(ICRNDB) the length in words of ETNTAB, TYDLEN(ICRNDB) the length in words of TYDTAB ,HPKTAB(ICRNDB) the length in words of HPKTAB, HATLEN(ICRNDB) the length in words of HATTAB, NAMLEN(ICRNBD) the length in words of NAMES, OPALEN(ICRNDB) the length in words of OPATAB ,RTNLEN (ICRNDB) the length in words of RTNTAB ; PAGIND index in the first page where ETNTAB begins ;BUFDAT the date that DDLA was executed to produce the DBT ; BUFTIM the time that DDLA was executed to produce the DBT.

4.5.2.4.12. NAMES

The names of all the objects described in the DDL, except for the entity type and relationship type names, are stored in the array NAMES.

That is to say : the name of the attribute type, value set type, and allowable values.

The names are stored in the order found in the DDL, except when an attribute, value-set, or allowable value is described which has the same name as a previous attribute, value-set, or allowable value. In this case, only the first

instance of the name is stored, and the pointer for the duplicate name will point to the first occurrence of the name.

4.5.2.5. Dynamic Hashing Tables

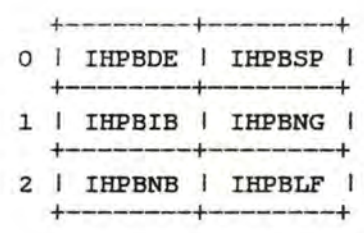
For the primary key, secondary key and the relationship file, we must have tables to store and update the different parameters of each file in the database.

4.5.2.5.1. The Database Hashing Table for the Primary Key (HPKTAB)

The hashing schemes used to handle the primary key file are explained in section 4.3.3. To store the information needed to manage these schemes, we have constructed the HPKTAB table. The database hashing table for the primary key is made of primary key description hashing blocks IHPB. There is one IHPB block by entity type. The IHPB are put in the table in the order of the entity type description (TYDTAB). The first IHPB corresponds to the first entity type described in TYDTAB and so on...

4.5.2.5.1.1. Primary Key Hashing Description Block (IHPB)

One IHPB consists of three contiguous fullwords of information used to describe the current state and the initial parameters of a primary key file..PP 1 The physical structure of an IHPB is the following :



The following table represents the meaning of each storage location.

Field Name	Size	Description
IHPBDE	Halfword	Depth of a file
IHPBSP	Halfword	Split pointer for a file

IHPBIB	Halfword	Initial number of bucket
IHPBNG	Halfword	Initial number of group
IHPBNB	Halfword	Number of bucket in a group not yet expanded
IHPBLF	Halfword	Load factor

IHPBDE represents the number of completed full expansions the file has undergone (initially IHPBDE = 0) and IHPBSP is the split pointer representing the next page to be split.

IHPBNG represents the initial number of groups in a file and IHPBIB indicates the initial number of buckets in a group.

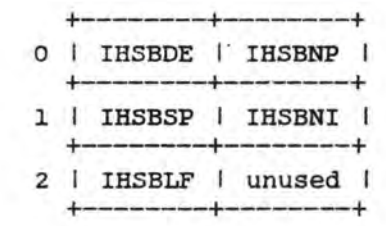
IHPBNB is the number of buckets in a group not yet expanded during a partial expansion. (initially IHPBNB = IHPBIB). IHPBLF represents the desired load factor of the file.

4.5.2.5.2. The Database Hashing Table for the Secondary Key and the Relationship (HATTAB)

This table is made up of the attribute hashing description block IHSB. There exists an IHSB for each secondary key file and each relationship file. They are stored in the order they appear in the DDL.

4.5.2.5.2.1. Attribute Hashing Description Block (IHSB)

One IHSB consists of three contiguous fullwords of information used to describe the current state and the initial parameters of a secondary key or relationship file. The physical structure of an IHSB is the following:



The following table represents the meaning of each storage location:

Field Name	Size	Description

IHSBDE	Halfword	Depth of the file
IHSBNP	Halfword	Number of pages for this file
IHSBSP	Halfword	Split pointer
IHSBNI	Halfword	Initial number of page
IHSBLF	Halfword	Load factor

IHSBDE represents the depth of the file ,i.e, the number of expansions the file has undergone. IHSBNP represents the current number of pages of the file.

IHSBSP represents the split pointer (the next page to split). IHSBNI is the exponent of 2 to reach an initial number of pages. For example if we want to have an initial file of eight pages then IHSBNI = 3 because 2 EXP 3 = 8. IHSBLF is a desired load factor for the file.

4.5.2.5.3. Optimal Parameters Table (OPATAB)

The optimal parameters table is used to store all the optimal parameters to manage the secondary key files and the relationship files in a way described in section 4.3.

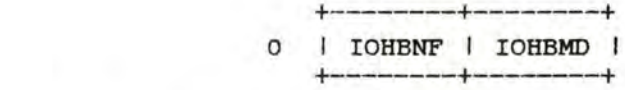
The optimal parameters table is made up of the optimal parameters description block IOPB and the optimal header description block IOHB.

In the OPATAB table, there is one IOHB corresponding to each secondary key relationship defined. For each IOHB there are (number of fields * * maximum depth of the file) IOPB. After each IOPB there is the choice vector (see section 4.3.4.5.2) which is comprised of a set of IOABCV blocks.

4.5.2.5.3.1. The Optimal Header Description Block (IOHB)

One IOHB consists of one contiguous fullword of information used to describe the optimal parameters needed to compute the address of a record type and its associated descriptor in an optimal way.

The physical structure of an IOHB is as follows:



The following table represents the meaning of each storage location.

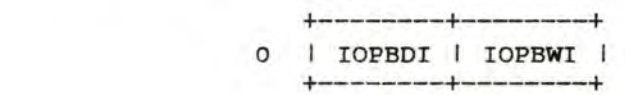
Fiel Name	Size	Description
IOHBNF	Halfword	Number of field
IOHBMD	Halfword	Level of the file

One IOHBNF represents the number of fields of a secondary key or a relationship. IOHBMD represents the level of the file, i.e., the maximum number of expansions in a file.

4.5.2.5.3.2. The Optimal Parameter Description Block (IOPB)

One IOPB consists of one contiguous fullword of information used to describe the optimal parameters of a file. There is an IOPB for each field of the file.

The physical structure of an IOPB is the following:



The following table represents the meaning of each storage location.

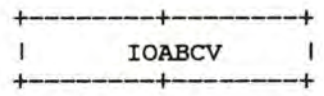
Field Name	Size	Description
IOBPDI	Halfword	Number of optimal bits for a file
IOPBWI	Halfword	Number of optimal bits for a descriptor

IOPBDI represents the optimal number of bits allocated to a field at a particular level of the file. IOPBWI represents the number of optimal bits for a field of a descriptor file. This field is the same that the field of IOPBDI.

4.5.2.5.3.3. The Choice Vector Description Block (IOAB)

One IOAB consists of one contiguous fullword of information used to describe the part of a choice vector. It is necessary to remember that the choice vector is a set of numbers whereby each number represents the field to take into account when computing the address of the entire record.

The physical structure of an IOAB is as follows:



IOABCV is an integer between 0 and the number of fields of a secondary key or relationship.

4.5.2.6. File Organization

There are four kinds of files in the DBMS; a primary key file, a secondary key file , a relationship file and a descriptor file. Each secondary key and relationship file is associated with a descriptor file. There are as many primary key files and secondary key files as there are entity types.

Each primary key in a primary key file is associated with the address (within page no. and displacement on the page) of the corresponding secondary key and attribute part in a secondary key file. Each secondary key and attribute part is, in turn, associated with its primary key. The primary key files are handled using the techniques defined in section 4.3.3 and 4.3.4 and relationship files are handled using the techniques defined in section 4.3.4 .

The following figure represents the files organization for one entity type.

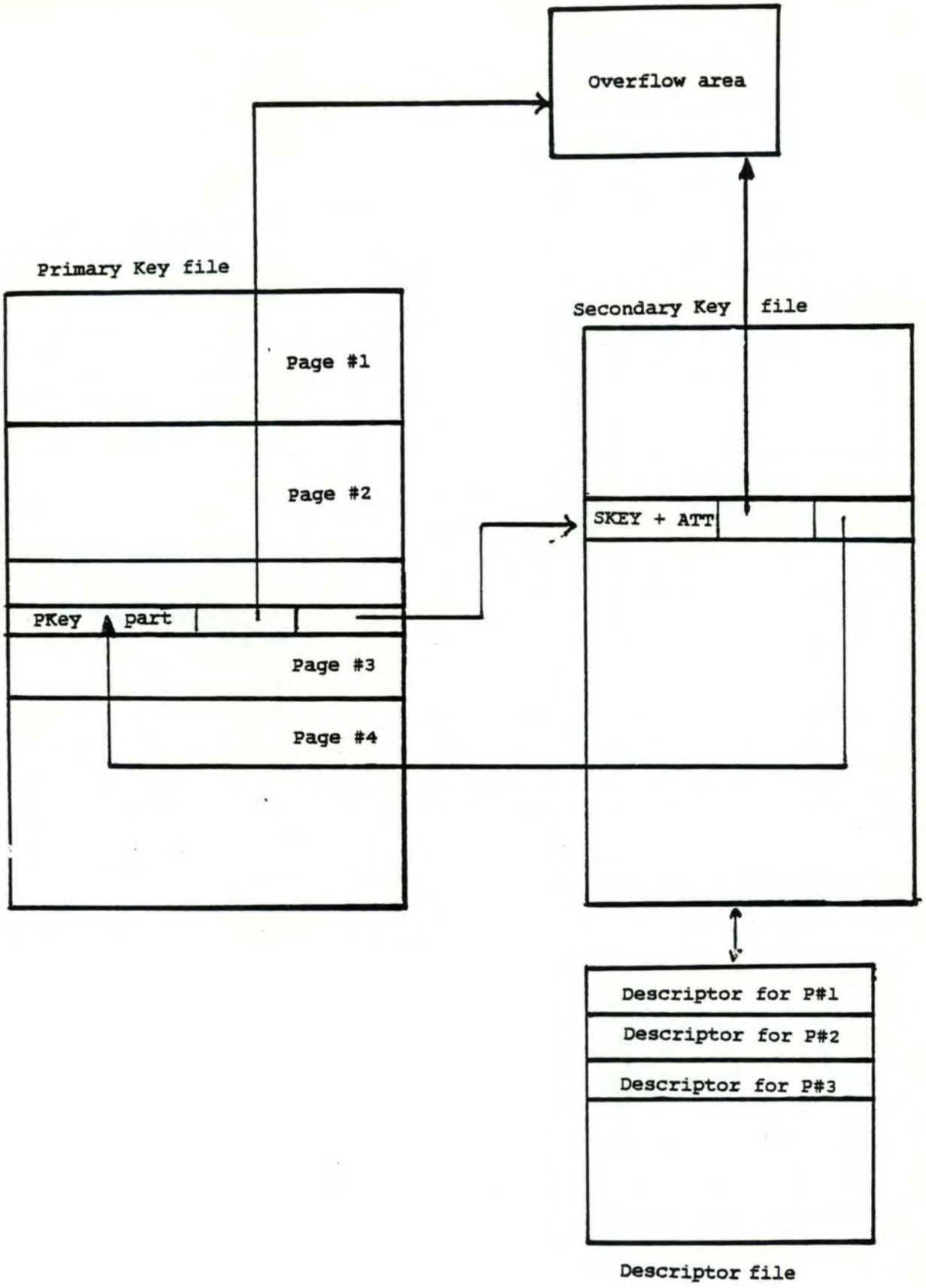


Fig 4.14 File organization for an entity type

The following figure represents the file organization for one relationship type.

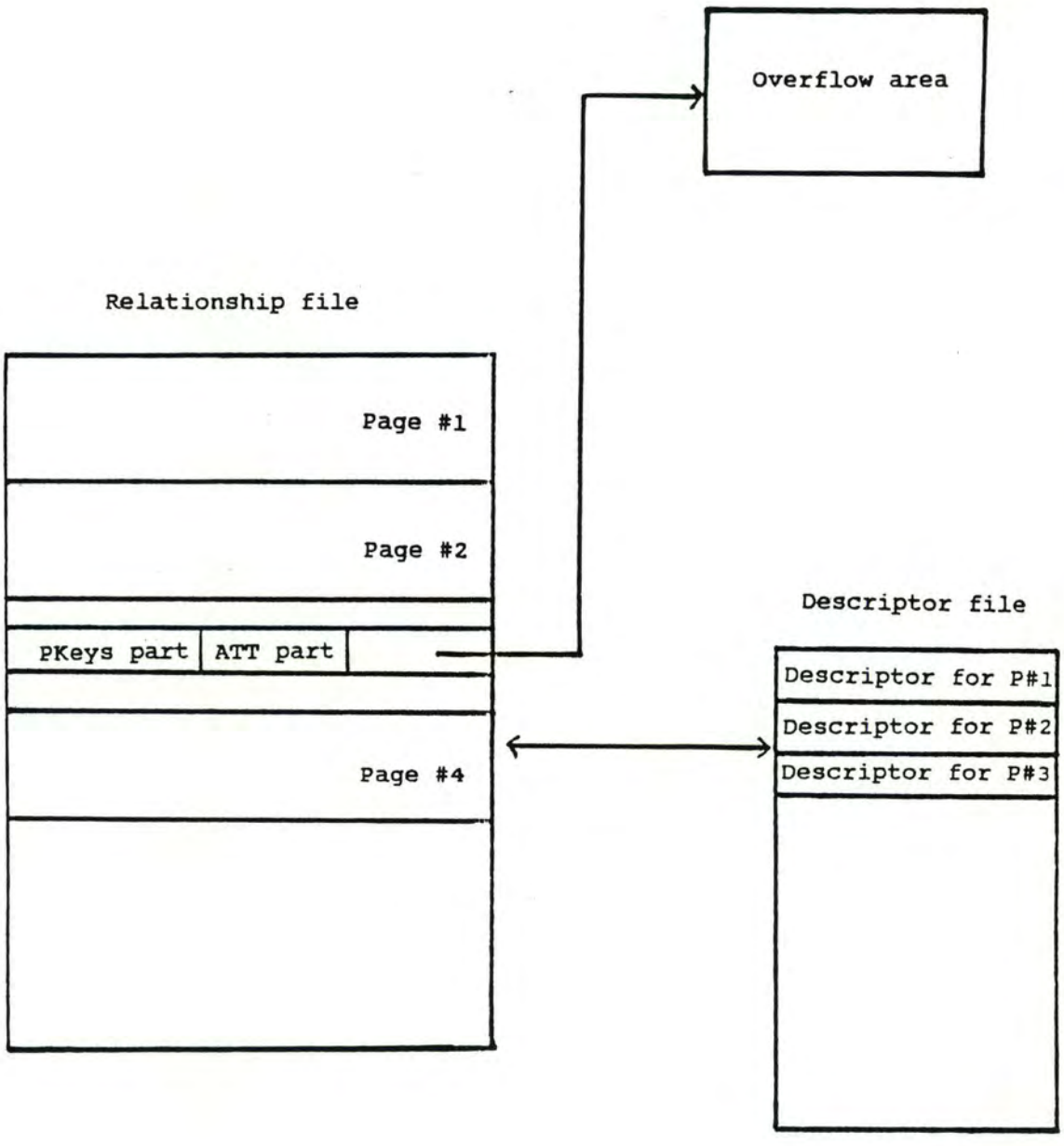


Fig 4.15 File organization for a relationship type

4.5.2.7. Database control block

As informations are stored in the database, it is structured by the DBCS into physical record and is stored in the DBF. This section describes the structures of the physical records which appear in the DBF.

4.5.2.7.1. Identification of a Page

The DBF consists of a certain number of physical pages (4096 bytes each) which belong to a certain page type. A particular page in DBF is identified by the type of the file (descriptor, primaty key, secondary key ,relationship), the instance number of a file type (for example if there are eight primary keys defined than there will be eight primary key files and we must identify each of these files) and the sequence number of a page in a particular file. One way to solve this problem of identification would be to have the address on one word where the first halfword would be used to identify the file and the second halfword would be used to identify the page number.

Unfortunately this leads to very high numbers (though less than the highest integer permitted) which are not always accepted by the file handler. Theoretically any integer could identify a record(page) in a random file but in reality this is not the case.

So we have segregated the DBF in rows of 1000 pages: 450 pages are reserved for the primary key file, 450 for the secondary key file and 100 for the associated descriptor file. If it is a relationship file then 900 pages are reserved for the relationship file and 100 for its descriptor file.

Later, this possibility of truly dynamic files must be added.

4.5.2.7.2. Data Area

Each page can be seen as a table because all the records on a page have the same length. With each record on a page is associated 3 words. The first two words indicate the address (page # and displacement) in an overflow area. The third word indicates if the record is used or not. In addition, the primary key record and the secondary key record have two words associated with them which indicate the address of the corresponding

secondary or primary key

4.5.2.7.3. Storage Allocation

When the DBCS searches a page to store any record type, it first computes the addresses where this type of data will be inserted.

Once the address is computed, we can read the page in which to insert this record. The page number is formed by the concatenation of the number which represents the file type and the index for this page.

We thus access to the location of the record inside this page. We first check if the data area is not present at this address or in the associated overflow pages. If not, we can insert the data area either in the main page or in an overflow page. If yes, we do not insert the data area.

To add a record to an overflow area, we must find a pointer equal to 0 in the chain of pointers and then find the next available place, insert the record, and update the pointer.

This means that we need a current overflow address which indicates the page number and the displacement in the page from which the information can be retrieved. If the number of insertions and deletions is higher than a certain load factor (normally the page size), then we get one or more pages and we redistribute some records from a certain page to pages that we have obtained.

4.5.2.7.4. DBCS Page Management System

Many computer installations limit the number of pages in main memory that a program may use at any particular time. Since the size of the database can be many times this limit, the DBCS contains a set of routines which will control which pages in the database are kept in main memory. If a page is needed that is not in main memory at this time, the DBCS will go to the DBF and read in the page.

4.5.2.7.4.1. The DBCS Random I/O Routines (DBHRAN)

The basis of the page management system is a set of random input, output routines contained in the DBHRAN These routines can read selected pages of the database into main memory and, in their proper sequence, write them out again into the specified DBF.

4.5.2.7.4.2. The Page Buffer (BUFPAG)

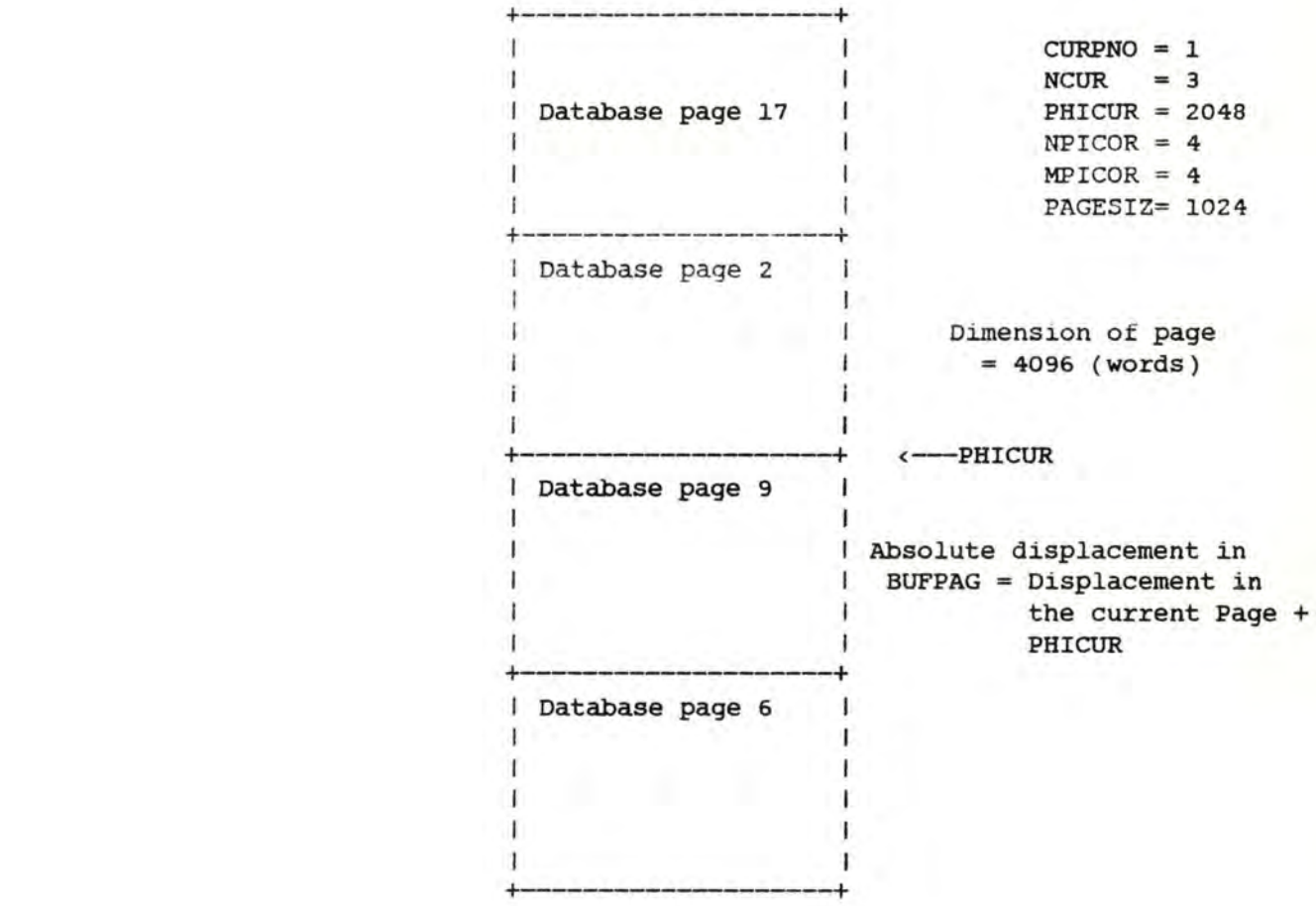
The database pages, when in main memory, are kept in a single one-dimensional integer array called BUFPAG. BUFPAG is dimensioned to a value equal to page size (in words) multiplied by the maximum number of pages allowed in main memory at one time or MPICUR.

4.5.2.7.4.3. The Current Page

The DBCS flags one of the pages in main memory as the "current page." The page number of the current page is kept in an integer fullword named CURPNO; several DBCS routines reference the current page rather than specify one particular page.

Associated with CURPNO is a fullword integer variable, PHICUR. PHICUR is a pointer to the beginning of the current page. Because the current page may be in one of several places in BUFPAG, a displacement into the current page is not necessarily the same as the displacement of the same location in the BUFPAG vector.

The following diagram gives an example of a database at a particular point execution of a user program. MPICOR for this example is four pages, and page size is 1024 words per page. This implies that page is dimensioned to 4096. There are four database pages in main memory: pages 17, 2, 9, and 6. NPICOR, the number of database pages in main memory, has a value of 4. Page 9 is the current page, which is sequentially the third page in main memory. NCUR, the sequence number of the current page in main memory is then 3. PHICUR, the pointer into BUFPAG for CURPNO is 2048 (this is numerically equal to (NCUR-1) * PAGESIZE).



When the current page changes, CURPNO and NCUR are changed, and PHICUR is recalculated. This means that the DBCS can always reference the correct location given the displacement on the current page.

4.5.2.7.4.4. Reading a New Page from the DBF into Main Memory

There is a particular algorithm used to read a page into main memory from the database. The DBCS first checks to see if we have the desired page in the core already. If so, the DBCS sets it to be the current page and returns. If the DBCS does not find the desired page, it then has two alternatives to try in order to obtain a new current page.

First the DBCS checks to see if NPICOR is equal to MPICOR. If so, there is no room in BUFPAG for a new database page and we must select a new page and be overwritten.

To do this the DBCS, utilizes a table named PREF and a counter named NDBKF. PREF must be dimensioned to be at the least the value of MPICOR. Each time the current page is reset, NDBKF is incremented and stored in the location in

PREF that is associated with the page reference number in main memory. In this way, the page in main memory which corresponds to the lowest value in PREF is the page that was least recently referenced. This is the page that the DBCS selects to be removed. Since all the other pages in main memory have been referenced more recently, they normally have a greater chance of being referenced than of being removed. Before removing the page, the DBCS checks to see if that page was modified. If so, the old page in rewritten into in the DBF. If the old page was not modified since being read in, it does not need to be written out into the DBF. The new page which is read in replaces the old page, and becomes the current page.

If NPICOR < MPICOR, it means that there is still room in the page for a new database page. The NPICOR is incremented by one, and the new page becomes the current page.

4.5.2.7.5. The Database Control System (DBCS)

The DBCS is a set of FORTRAN routines which are used by the user's program to access a database. The routines are divided into the five following groups clarified roughly by function.

1. DBHUSE

   These routines are the highest level routines that interface directly with the user's program.

2. DBHLOW

   These are the low-level routines used by DBHUSE. They do most of the actual "work" of the DBCS.

3. DBHTAB

   These routines are used by DBHUSE and DBHLOW to access the database tables and control blocks.

4. DBHRAN

   These are the DBCS random I/O routines which are used to open, read from, write onto, rewrite onto, and close the DBF.

All these routines will be described in annexe 2.

4.5.3. Implementation Evaluation

The ISDOS project has adopted conventions of programming for all the software that is developed inside the project [ISDOS 1979a], [ISDOS 1979b]

The goal of these conventions is to provide facilities for the development, maintenance and portability of the software. The main characteristics of these conventions are the following :

- An Extended Fortran language has been developed. This language mainly allows the declaration of a character variable, the substitution of named constant and the substitution of relational and boolean operators.

  Each routine must contain as heading a brief description of their purposes and input/output parameters.

- One and only one "return" is allowed in each subroutine. "Return" is always preceded by the label "900".

- All the labels must be ordered.

- The comment lines are highly recommended.

Up until now, only a part of the system has been implemented and tested. The main reason is that the different levels of such a complex system as a database management system are highly integrated. This means that the system can be tested with efficiency only if the majority of the routines in the different levels of the system have been written. The list below is the description of what has already been done and what is to be done :

- DDLA program has been implemented and tested. This program reads the description of a database, constructs the tables and inserts an initial database.

- DBHLIB library has been implemented and tested. These routines in this library control the operations inside the buffer BUFPAG. Particulary, they store and retrieve character strings, logical values, words and halfwords. They also perform operations of comparisons.

- DBHRAN level (I/O routines) have been also implemented and tested.

- DBHTAB level has also been implemented and tested.

- In DBHLOW only the following routines have been tested the other routines of this level have been constructed but not yet tested.

  1. IDPAGE routine
  2. REPAGE routine
  3. WHPAGE routine
  4. NEPAGE routine
  5. DBHTFI routine
  6. DBHERR routine

  - DBHUSE level hasnot been yet implemented .

CONCLUSION

The goal of this thesis was to isolate some levels of the ISDOS software which are critical for its performance and try to improve as much as possible these points.

In a first phase, we studied the components of the ISDOS software and we tried to evaluate their performance. The database management system (ADBMS) is revealed to be one of the most important points of the system. We outlined the cases in which its performance decreases considerably. Some solutions are possible to remedy this situation.

One approach is to say that the informations processed by SEM are a relational(E-R) rather than network type (ADBMS) of model. This leads to poor performance since we have to provide an E-R interface to ADBMS. Thus one solution would be to replace the actual database management system by an E-R database management system .

As the design and the implementation of such a system is a huge task we limited our reflection to direct access techniques. These techniques are based on new hashing techniques which have recently appeared in the literature. They often offer better performance than conventional hashing but can not alone support a database management system.

The design of a system was proposed in chapters 3 and 4. The implementation has not yet been achieved and the next step of the work will be to complete it. Next an evaluation of the performances must be performed to know if for certain these techniques are more efficient than the classical ones, and if so, in which cases.

Annex 1 gives an overview of the new hashing techniques that have recently appeared and annex 2 describes the routines which must be implemented for the system.

Thought working in a project whose goal is to help in maintaining the documentation of the information systems, we do not have used (for time reason) the tools proposed to describe our system.

BIBLIOGRAPHY

[Biggs, Birks, and Atkins 1980]

Charles L. Biggs, Evan G. Birks, and William Atkins, Managing the Systems Development Process ,Touche Ross Management Series, Prentice-Hall , 1980.

[CDDL 1974]

National Bureau of Standard Handbooks 113, CODASYL Data Description Language Journal of Development , U.S. Department of Commerce, National Bureau of Standards , Washington D.C., 1974.

[Chen 1976]

Peter P. Chen, "The entity-relationship model - toward a unified view of data", ACM Transactions on Database Systems , Vol. 1, no. 1 (March 1976), pp. 9-36.

[Chen 1977]

Peter P. Chen, "The entity-relationship model - a basis for the enterprise view of data", Proceedings, National Computer Conference , 1977, Vol. 46, pp. 77

[DBTG 1971]

CODAYSL Data Base Task Group Report, April 1971, Correspondence: Chairman, CODAYSL Data Base Task Group, P.O. Box 124, Monroeville, Pennsylvania, 15146.

[Engles 1971]

R.W. Engles, "An Analysis of the April 1971 Data Base Task Group Report", Proc. ACM SIGFIDET Workshop on Data Description, Access and Control, 1971.

[ISDOS 1978]

ISDOS Project, ADBMS Program Logic Manual, Version D3.0 , ISDOS Ref. # 7830-0193-0, Department of Industrial and Operations Engieering, the University of MIchigan, Ann Arbor, Michigan, 48109, January 1978.

[ISDOS 1979a]

E.A. Hershey III, Y. Yamamoto, E. Chikosky, and D. Mielta, ISDOS Project, ISDOS Software Conventions , ISDOS Ref. # 267, Department of Industrial and Operations Engineering, the University of Michigan, Ann Arbor, Michigan, 48109, July, 1979.

[ISDOS 1979b]

E.A Hershey III, Y. Yamanoto, E. Chikosky and D. Mielta, ISDOS Project, Extended Fortran (EF) ,ISDOS Ref. # 265, Department of Industrial and Operations Engineering, the University of Michigan, Ann Arbor, Michigan, 48109, July, 1979.

[ISDOS 1981a]

ISDOS Project, A Database Management System (ADBMS) Based Upon DBTG 71 ISDOS Ref. # 81D32-0191-1, Department of Industrial and Operations Engineering, The University of Michigan, Ann Arbor, Michigan, 48109, February 1981.

[ISDOS 1981b]

ISDOS Project, Technical Memorandum 379, Department of Industrial and Operations Engineering, the University of Michigan, Ann Arbor, Michigan, 48109, September 18, 1981.

[ISDOS 1981c]

ISDOS Project, The Structure and Contents of a PSA Data Base , ISDOS Ref. # 81A52-0273-1, Department of Industrial and Operations Engineering, the University of Michigan, Ann Arbor, Michigan, 48109, January 1981.

[Larson 1980]

P. Larson, "Linear Hashing With Partial Expansions", Proceedings, Sixth International Conference on Very Large Databases , 1980, pp. 224-232.

[Litwin 1978]

Witold Litwin, "Virtual Hashing : A Dynamically Changing Hashing", Proc. Fourth Conference on Very Large Data Bases, West Berlin, September, 1978, pp. 517-523.

[Litwin 1980]

Witold Litwin, "A New Tool for File and Table Addressing", Proc. on Very Large Databases, Montreal, 1980, pp.212-224.

[Lloyd 1980]

John W. Lloyd, Optimal Partial-Match Retrieval, BIT 20, 1980, pp. 406-413.

[Lloyd and Ramamohanorao to be published]

John W. Lloyd and K. Ramamohanorao, "Dynamic Hashing Schemes", Department of Computer Science, the University of Melbourne, to be published.

[Lloyd and Ramamanoroa 1982]

John W. Lloyd and K. Ramamohanorao, "Partial-Match Retrieval for Dynamic Files, BIT" ,Department of Computer Science, the University of Melbourne, BIT 22 (1978), pp. 150-168.

[Lloyd, Ramamohanorao and Thom 1983]

John W. Lloyd, K. Ramamohanorao, and James A. Thom, "Partial-Match Retrieval Using Hashing and Descriptors", Department of Computer Science, the University of Melbourne, to appear in ACM Transactions on Data Base Systems, 1983.

[Martin 1975]

James Martin, Computer Data Base Organization , Prentice Hall, 1975

[Metzger 1973]

Phillip W. Metzger, Managing a Programming Project , Prentice-Hall, 1973.

[Pfaltz, Berman and Cagley 1980]

John L. Pfaltz, William J. Berman and Edgar M. Cagley, "Partial-Match Retrieval Using Indexed Descriptor Files", Communications of the ACM, Vol. 23, No. 9, September, 1980.

[Pholas 1974]

Data Systems Pholas Schema DDL and SSL .

[Reifer 1975]

Donald J Reifer, Interim Report on Aids Inventory Project , SAMSO-TR-75-184, July 16, 1975.

[Reifer and Trattner 1977]

Donald J. Reifer and Stephen Trattner, "A Glossary of software tools and techniques", Computer (IEEE Computer Society), July 1977, Reprinted in [Miller 1979].

[Sakai 1980]

Hirotaka Sakai, "Entity-Relationship Approach to the Conceptual Schema Design", Hitachi Institute of Technology, Hitachi, LTD., Tokyo, Japan, 1980.

[Teichroew, Hershey and Yamamoto 1977]

Daniel Teichroew, Ernest A Hershey lll, and Yuzo Yamamoto, "Computer-aided software development", Software Reliability , Infotech International LTD., Maidenhead, Berkshire, England, 1977, Part 2, pp.299-366.

[UDS 1977]

Universal Database Management System, Schema DDL and SSL, Reference Manual.

[Yamamoto 1981]

Yuzo Yamamoto, "An Approach to the Generation of Software Life Cycle Support Systems", Ph.D. dissertation, the University of Michigan, 1981.

ANNEX 1 : Overview of new hashing schemes

Table of Contents

ABSTRACT.....3

1. Introduction.....4

2. Hashing Schemes Using an Index.....6

2.1 Extendible Hashing Scheme [Fagin 1979].....6

2.1.1 Introduction.....6

2.1.2 Extendible Hash Tables Principle.....6

2.1.3 A Particular Extendible Hashing Scheme.....7

2.1.4 Finding, Inserting, and Deleting Records.....9

2.1.5 Performance.....9

2.2 Dynamic Hashing Scheme [Lloyd, Ramamohanorao and Thom 1983], [Scholl 1981].....10

2.2.1 Presentation of the Scheme.....10

2.2.2 Dynamic Hashing Scheme With Deferred Splitting.....13

2.3 Linear Splitting.....16

2.3.1 File Structure.....16

3. Virtual Hashing Schemes [Aho 1979] [Litwin 1978].....19

3.1 First Virtual Hashing Scheme.....19

3.1.1 Presentation of the Scheme.....19

3.1.2 Performance Consideration.....21

3.2 Improvement of Virtual Hashing: Linear (Virtual) Hashing.....21

3.3 Introduction.....21

3.4 Presentation of the Scheme.....21

3.4.1 Performance Consideration.....25

3.4.2 Control Functions.....25

4. Linear Hashing With Partial Expansions [Larson 1980].....26

4.1 Introduction....26

4.2 Presentation of Linear Hashing With Partial Expansion....26

4.3 Control Function....29

4.4 Performance....30

5. Variation of Larson and Litwin's Scheme [Lloyd and Ramamohanrao to be published]....30

5.1 Introduction....30

5.2 Dynamic Hashing Scheme With Round-Up Pages....30

6. Dynamic Hashing Scheme for Secondary Key File [Lloyd, Ramamohanorao and Thom 1983]....32

6.1 Introduction....32

6.2 Definition of a Partial Match Query....32

6.3 Description of a Simple Partial-Match Retrieval Scheme When the File is Known) Based Purely on Hashing....32

6.4 A Descriptor Scheme....33

6.5 Extension of the Scheme to Dynamic Files....38

6.5.1 Presentation of the Scheme....38

6.5.2 Choice Vector....39

6.5.3 File Descriptor....40

6.6 Performance....40

ABSTRACT

This paper gives an overview of new hashing techniques which allow an address space to grow or shrink dynamically. A file or table based on these techniques may support any number of insertions and deletions without access or memory load performance distortion. This paper is not intended to include the mathematical development of these techniques, nor is it intended to present the algorithms or a complete performance analysis. The aforementioned information that has been excluded from this paper may be found in the references cited in the bibliography.

1. INTRODUCTION

Hashing is an ingenious and useful form of address calculation technique. A simple pseudo-random function called hashing function (H) converts the item of a record (called the primary key) into a near random number and this number is used to determine where the record is stored.

The records are stored in places called buckets. A bucket can hold one or more records and the set of buckets is called the address space of the file.

The record is inserted into the bucket H(c) (where c is the primary key), unless the bucket is already full. The search for c always starts with a access to the bucket H(c). If the bucket is full when c should be stored, a "collision" occurs.

A collision resolution method, which stores c in a bucket M such that M =/ H(c), is then applied. The record c then becomes an overflow record and the bucket M is called an overflow bucket for c. The overflow records are often handled by a method called "bucket chaining "

Bucket chaining is the method by which overflow records are stored by linking one or more overflow buckets from a separate storage area to an overflowing bucket. Each overflowing bucket has its own separate chain of overflow buckets.

A search for an overflow record requires at least two accesses. If all collisions are resolved only by overflow records creations, as it was assumed recently, access performance rapidly deteriorates when primary buckets become full.

Fig 1.1 shows a conventional hashing file where the records are handled by bucket chaining.

Hashing is recognized as providing, in practice, the fastest random access to a file. Theoretical analysis indicates that access time to a hash table is independent of the number of records, but depends on the four factors listed below [MARTIN 1975].

The factors affecting efficiency are the following :

1. the bucket size
2. the load factor, i.e, the number of records stored in home buckets divided by the maximum number of records that could be stored in them
3. the hashing function used
4. the method of handling overflows

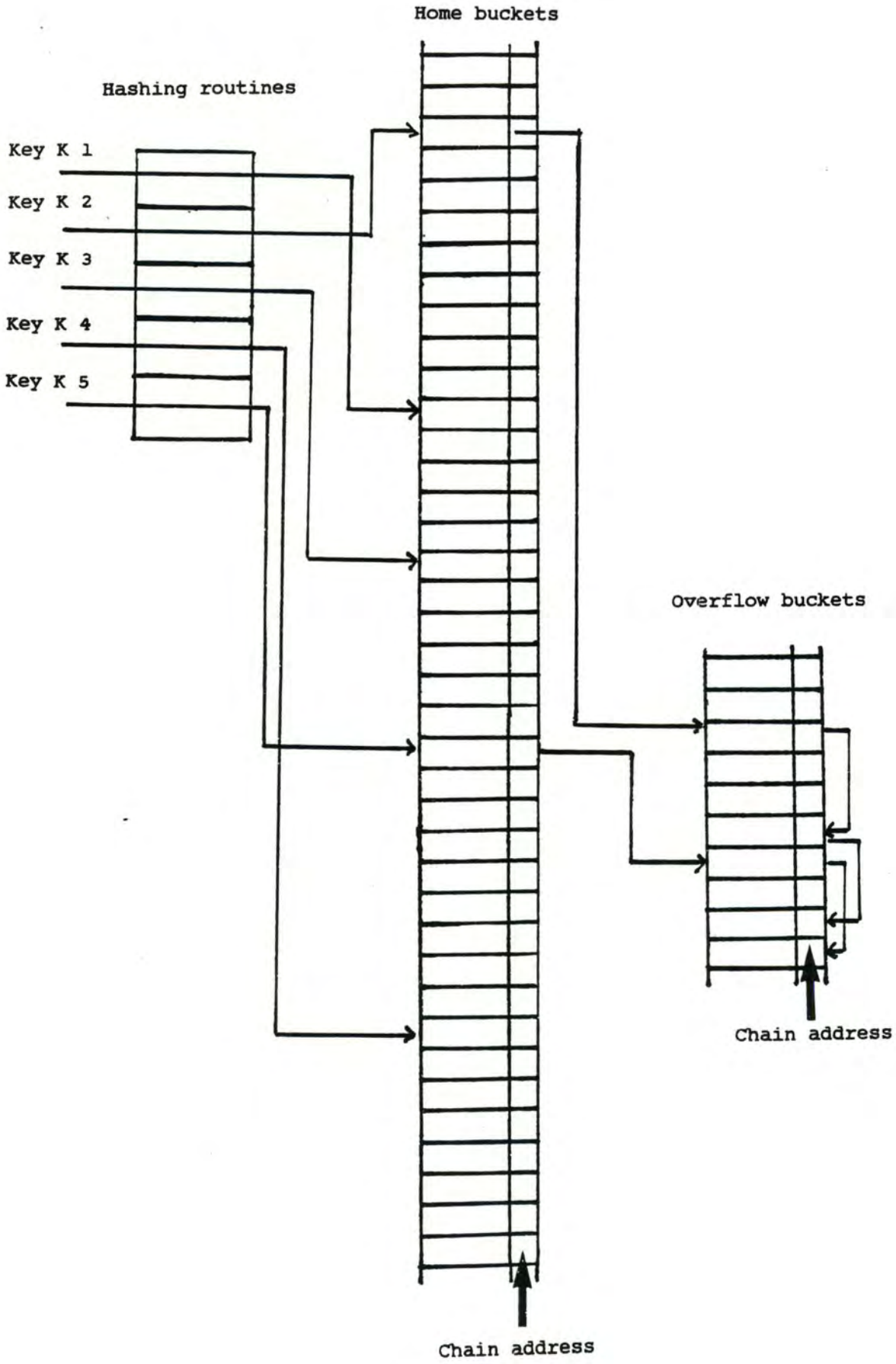


Fig 1.1 Conventional Hashing scheme

In contrast to the fast O(1) access time, hashing is burdened with two disadvantages that prevent its use in many applications :

- Hashing does not support sequential processing of a file according to the natural order of the key (sequential processing requires an O (n * ln n) operation which makes the fast random access useless.

- Traditional hash files are not extendibles and their sizes are tied to the hash function used which often must be redefined.

  Thus, if the file grows by a large factor or if the records distribution over the available storage space is not uniform, the number of overflow records grows and the time to retrieve a record increases considerably.

  A high estimate of the number of records implies a costly rehashing operation (new hash function, table size, relocation of all records, etc.). Shrinkage of the file or a low estimation of the number of records implies under-utilized storage space.

If one can design adaptable hashing schemes that remain balanced as pages are added and deleted, the suitability of hashing for secondary storage devices would be greatly enhanced.

During the seventies, new file organizations which are based on hashing and which overcome the second disadvantage of conventional hashing were presented. These schemes are suitable for files whose size may grow and shrink rapidly. Their main characteristic is that the storage space allocated to the file can be increased and reduced without reorganizing the whole file.

These schemes are called "dynamic hashing schemes" and can be divided into two types :

1. those making use of some kind of index whose size varies with the file size

2. those without an index

The first types of schemes developed were: virtual hashing by W. Litwin [Litwin 1978], dynamic hashing by P. Larson [Larson 1978], and extendible hashing by R. Fagin [Fagin 1979]. So far there has been only one scheme of the second type, called linear (virtual) hashing and which was developed by W. Litwin [Litwin 1980] and P. Larson [Larson 1980]. These hashing schemes are reviewed in the following sections.

2. HASHING SCHEMES USING AN INDEX

2.1. Extendible Hashing Scheme [Fagin 1979]

2.1.1. Introduction

Extendible hashing is a new access technique in which the user is guaranteed no more than two page faults to locate the data associated with a given unique identifier or a key. Unlike conventional hashing, extendible hashing has a dynamic structure that grows and shrinks gracefully with the database size. This approach solves the problem of making hash tables that are extendible.

2.1.2. Extendible Hash Tables Principle

If we consider a hash table as a directory with an address space A and each entry of the directory pointing to a bucket of fixed size (figure 2.1) , we will see that this traditional picture has the disadvantage of not having the capability of making the file extendible. If a bucket overflows because too many keys K arrive with H(k) equal to an address W, there is no alternative to rehashing.

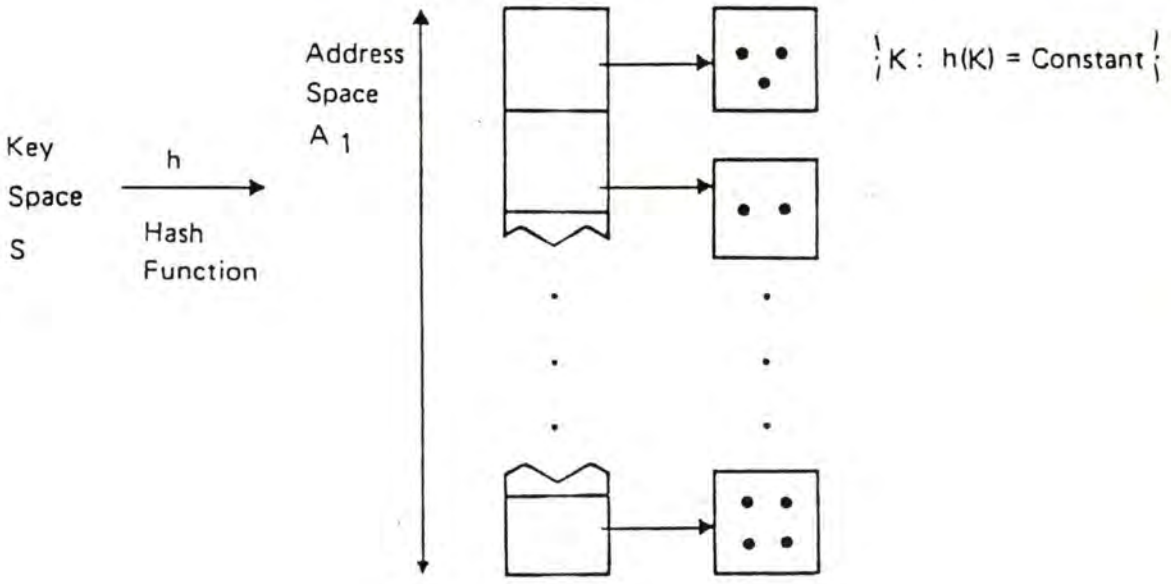


Fig 2.1 Hashing into a directory

We shall now show that if we separate the hash address space from the directory address space, hash tables can be made extendible.

The hash function maps the key space S on to a large address space A. A partition T splits A into m blocks ; each

block has one bucket allocated for its use and the directory implements the correspondence between blocks and buckets. If the partition T is defined by m + 1 boundaries W0 ,W1 ,..... ,Wm, the bucket Li contains all keys k with W(i-1) <= H(k) < Wi. In this scheme, if a bucket overflows, we are able to change the partition by shifting one boundary Wi and relocating only those keys that are affected by this shift. Furthermore, H does not need to be changed (see figure 2.2).

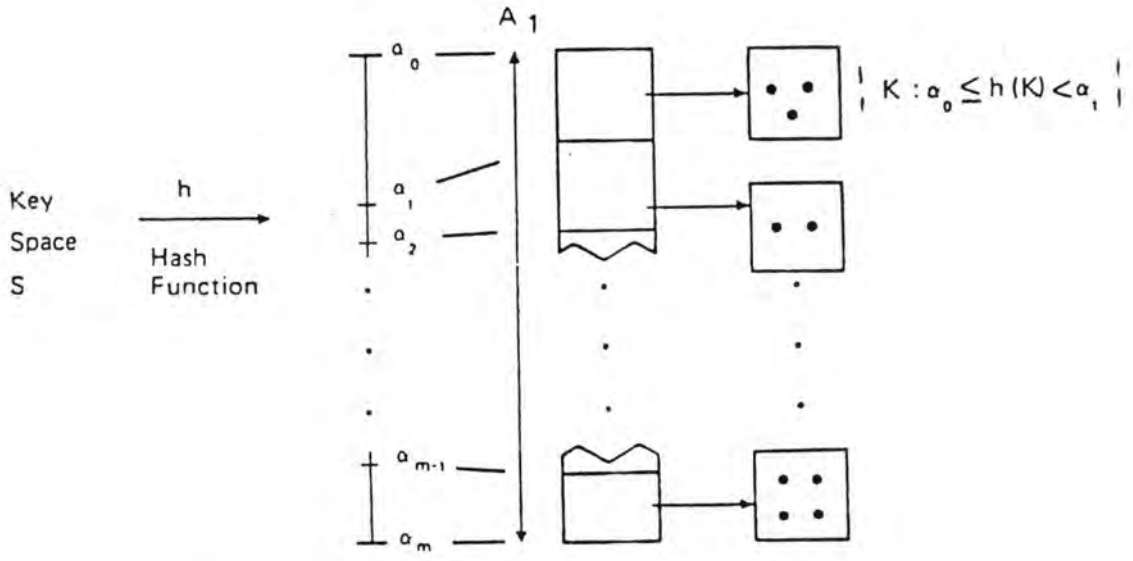


Fig 2.2 Hashing into a large address space

We have thus made the hash table extendible by varing a partition T on a large address space A, while keeping the hash function unchanged.

2.1.3. A Particular Extendible Hashing Scheme

Assume that we are given a fixed hash function H. If K is a key, then we call F = H(k) the pseudo-key associated with K. We choose pseudo-keys to be of fixed length, such as 32 bits. We must choose the hash function so that whatever the distribution of the keys, we can expect that the pseudo-keys will be distributed nearly uniformly : half the pseudo-keys have first bit 0, a quarter start with bits 01 and so on as figure 2.3 demonstrates.

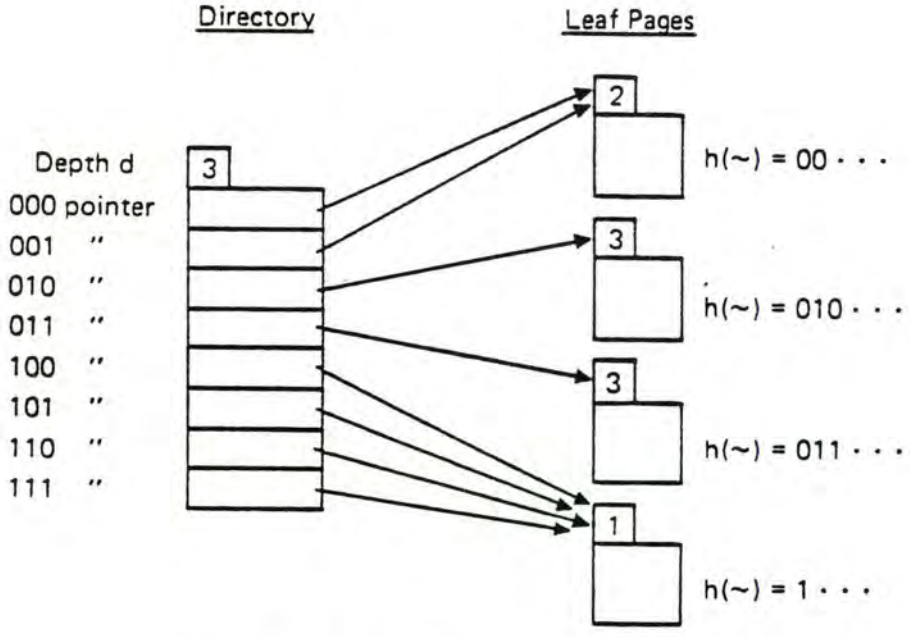


Fig 2.3 An extendible hashing scheme

The file is structured into two levels : directory and leaf pages.

1. Directory

The directory has a header in which a value called "the depth of the directory" is stored. After the header, there are pointers to a leaf page.

The pointers are built as follows. The first pointer points to a leaf page which contains all keys K for which the pseudo-key F' = H(k) starts with D consecutive zeros.

This pointer is followed by a pointer for all keys whose pseudo-keys begin with D bits 0.....01 and so on until the pointers for all keys whose pseudo-keys begin with D consecutive 1.

Thus, the depth D indicates the number of bits of the pseudo-keys which are taken into account to locate the pointer to a leaf page. For example, there are 2 exp(3) pointers not necessarily distinct in figure 2.3

2. Leaf Pages

Each leaf page has a header that contains a local depth D' for the leaf page. Local depth means that this leaf page contains all keys (and associated

information) whose pseudo-keys begin with the same D' bits.

In the example of figure 2.1, local depth 1 means that this leaf page contains not only all keys whose pseudo-keys begin with 100, but also all keys whose pseudo-keys begin with a 1. The depth of a leaf page is <= to the depth of the directory.

2.1.4. Finding, Inserting and Deleting Records

1. Finding a record

We must calculate H(k), where K is the key of the record we want, and find its first D bits. Do a simple address computation to find the location in the directory of the pointer that corresponds to the D-bit prefix, and follow this pointer to find a leaf page.

2. Inserting a record

If a leaf page is finally overfilled, we split this page into two leaf pages, each with depth = D' + 1. All keys whose pseudo-key begins with the D' pseudo-key + 0 are on the first new leaf page. All other keys whose pseudo-key begins with the D' pseudo-key + 1 are on the other page.

If a leaf page is overfilled and the local depth of the leaf page already equals the depth of the directory, the directory doubles in size, its depth increases, and the leaf page splits.

3. Deleting a Record

When a leaf page becomes underfilled because of deletions ,the corresponding block is merged with its buddy (if the latter has enough room). The buddy is the page which begins with the same D'-1 bits as the underfilled page.

2.1.5. Performance

If there are so many keys that the directory is in secondary storage, it can be streamed into main memory in large blocks since the directory is stored contiguously.

If there are a few million keys when the directory doubles and if the secondary storage device has a data transferral rate of approximately a million bytes per second (IBM 3330 disk), the time involved in doubling the directory is less than a second if there were 400 keys per leaf page.

The leaf pages can be organized as usual hash tables

with any standard collision resolution technique such as open addressing or chaining, as long as it stores colliding keys within the same page.

There is :

- one page fault in locating the appropriate directory page
- at most one page fault in obtaining the appropriate leaf page

No more than two page faults are necessary to locate a key and its associated information. In many cases, the directory will be so small that it can be kept resident in main memory, i.e., if the page size is 4K bytes, keys are 7 bytes long, and a pointer to page is 3 bytes long, and after one million have been inserted, the directory can be expected to be the size of 3 pages. The storage allocation provided by this scheme is approximatively equal to ln 2 = 69.

2.2. Dynamic Hashing Scheme [Larson 1978],[Scholl 1981]

2.2.1. Presentation of the Scheme

This scheme was presented by Larson [Larson 1978] and was refined by Scholl [Scholl 1981]. The dynamic hashing scheme is based on the same principle as the extendible hashing scheme, but the implementation is different.

Thanks to the dynamic hash function, a bucket is associated with the given unique record's key K and the bucket's location is identified by searching through one index whose size shrinks according to the data volume. Therefore, only one access to the secondary storage is necessary (if the index is available in the core).If the file grows steadily the index will be partly stored in secondary storage (2 accesses in this case).

An initial hashing function ,Ho, distributes the records among M initial index entries. The dynamic hashing index is implemented by means of a tree structure which grows and shrinks more smoothly than the extendible hashing index, but the index nodes are larger than those of the extendible hashing index entry. Each leaf of the tree contains a pointer to a bucket.

When a bucket overflows, the corresponding index leaf becomes an internal node to which two new leaves are appended, the left leaf pointing towards the original bucket, and the right leaf pointing towards a new bucket.

When the brother buckets are underfilled, they are merged into one bucket and the corresponding index leaves are deleted, their father now pointing towards the resulting

bucket.

With each record 's key k is associated a unique path when scanning down the tree as follows:H(k), where H is any hash function, is the seed for a pseudo-random generator which, when successively called, returns a binary sequence which uniquely determines the path through the three until a leaf is reached.

The index grows with time to a forest of M trees.Each index entry is either an internal node (TAG=0) which contains pointers to his father and sons, or an external node (TAG=1) which contains, in addition to the pointer to its father, a pointer to a bucket in the data file (BKT) and the number of records actually stored in the bucket (RCDS) (see Figure 2.4),

The performance of this scheme is nearly the same as in extendible hashing excepted that the index nodes are larger than that of the extendible hashing.

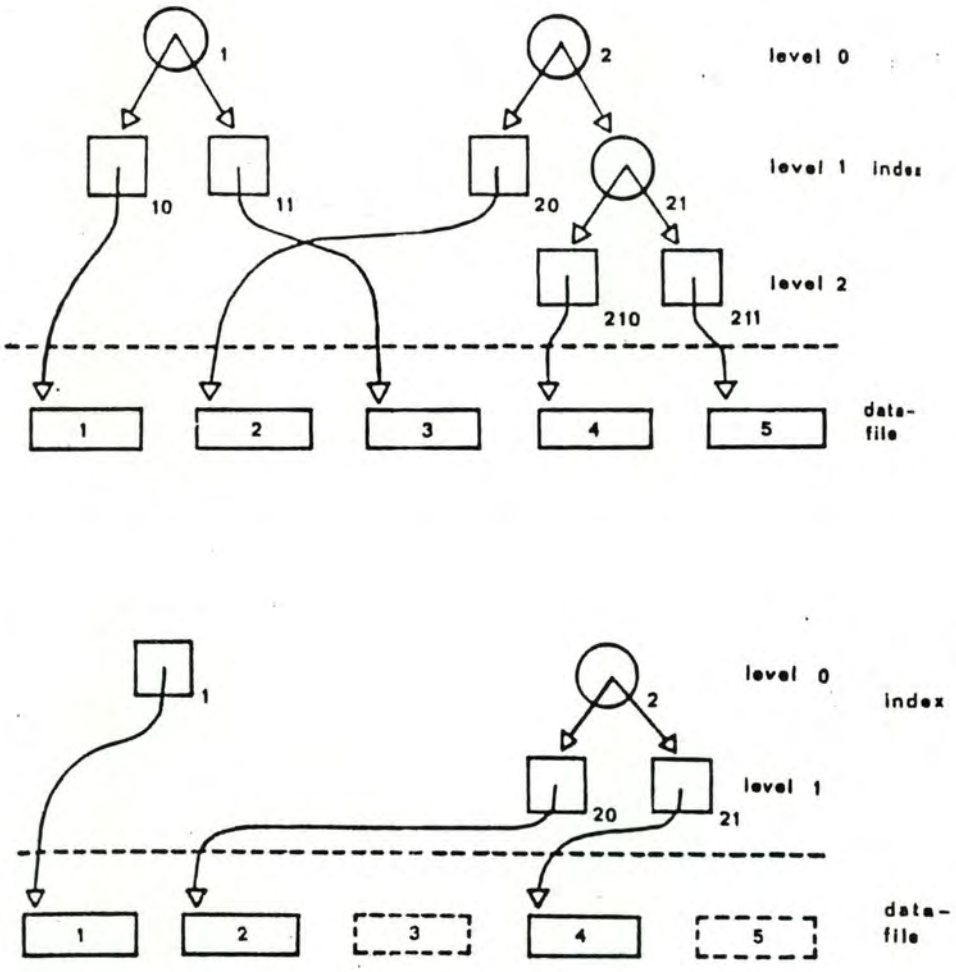


Fig 2.4 Dynamic Hashing scheme

2.2.2. Dynamic Hashing Scheme With Deferred Splitting

Scholl [Scholl 1981], in his first scheme, introduced the idea of systematically splitting pages, not necessarily when the collisions occured.

The idea is that the splitting of any bucket is deferred until B x b records (B>1) have been addressed to this bucket.

First consider the case where B<=2. When trying to insert a record into the primary bucket j which already contains b records, a new "overflow" bucket is allocated (bucket f) and chained to bucket j. The new record is inserted into this overflow bucket and any other record candidate to be inserted into bucket j will be inserted into the overflow bucket until the latter contains (B-1)b records.

When trying to insert a new record into a bucket whose overflow bucket already contains (B-1)b records, splitting occurs exactly as the dynamic hashing, a new bucket, say L, is allocated, and the Bb+1 records are distributed between bucket i and bucket j. Bucket F is freed and the index updated (see figure 2.5).

In the second case, if B > 2 when the first overflow bucket is full, another overflow bucket is allocated and chained to the first one and so on until Bb records have been inserted (see figure 2.6).

For all values of B, when splitting occurs, the index is updated; that is, the former external node (of level r in the tree) becomes an internal node pointing toward two external nodes (its sons) of level r+1 in the tree, each of them pointing towards a chain of one or more buckets containing the records. Then the Bb+1 records are distributed among the original chain of buckets and a new chain.

In the worst case, the number of accesses to recording storage required to find a record actually stored in the file (if the index is available in main memory) is equal to B. For the index size, the expected number of internal and external nodes are given by the following approximations.

It is shown in Larson [Larson 1978] that when the number of records, N, is large, because of the close connection with trees [Knuth 1973] in the case of dynamic hashing, the expected number of internal and external nodes are given by the following expressions.

Dynamic hashing :

- E(internal nodes) = (N/b*ln2) - M
- E(external nodes) = N/b*ln2

Dynamic hashing with deferred splitting :

- E(internal nodes) = (N/B*b*ln2) - M

- E(external nodes) = N/B*b*ln2

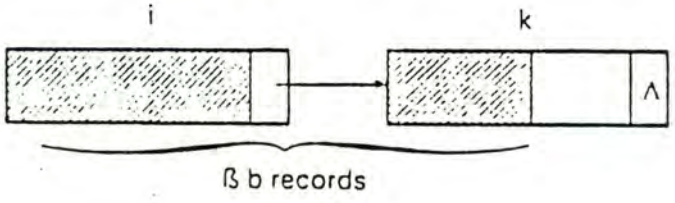
N is the number of records actually stored in the file at a given time.

M is the number of index entries.

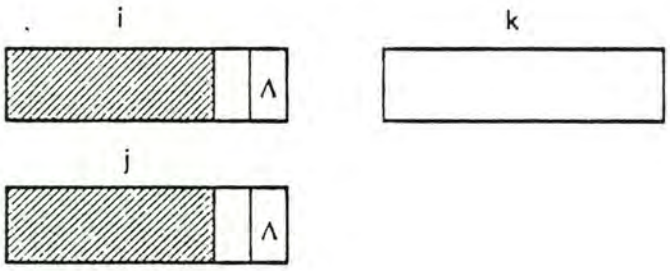
The index size of dynamic hashing with deferred splitting is approximately decreased by a factor of B.

Other existing improvements :

If a bucket j has overflowed, instead of allocating a separate overflow bucket to bucket j, we allocate the second half of bucket k as an overflow area for bucket j, but this leads to considerable storage utilization improvement at the expense of increased complexity in bucket management, while the index size is unchanged by the above modification (see fig 2.7)



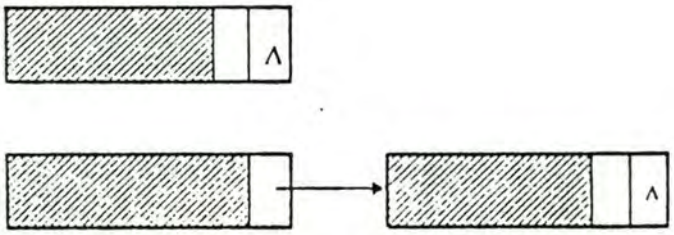
(a) B < 2



(b) Just after the split : k is freed ; B < 2

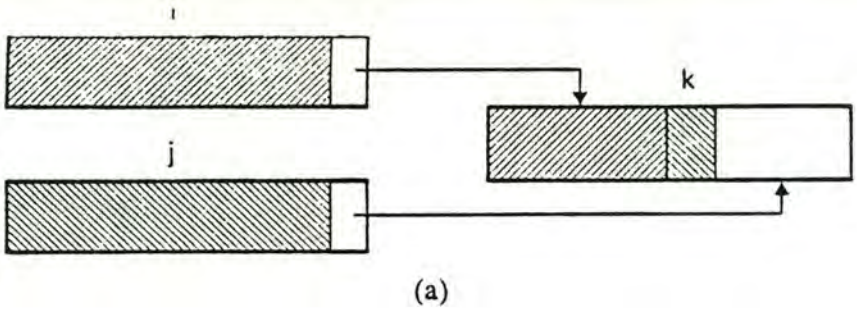
Fig 2.5 Deferred splitting B<2

(c) B > 2



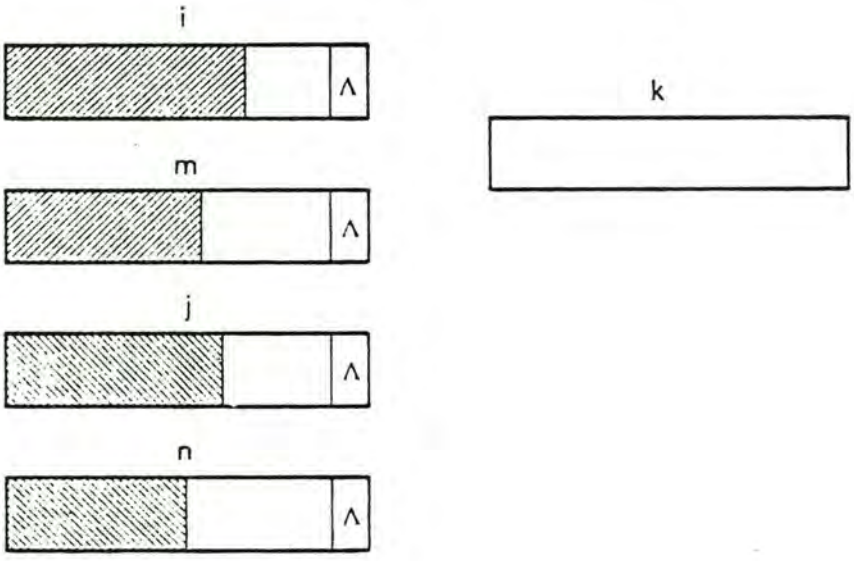
(d) B > 2

Fig2.6 Dynamic Hashing scheme with deferred splitting



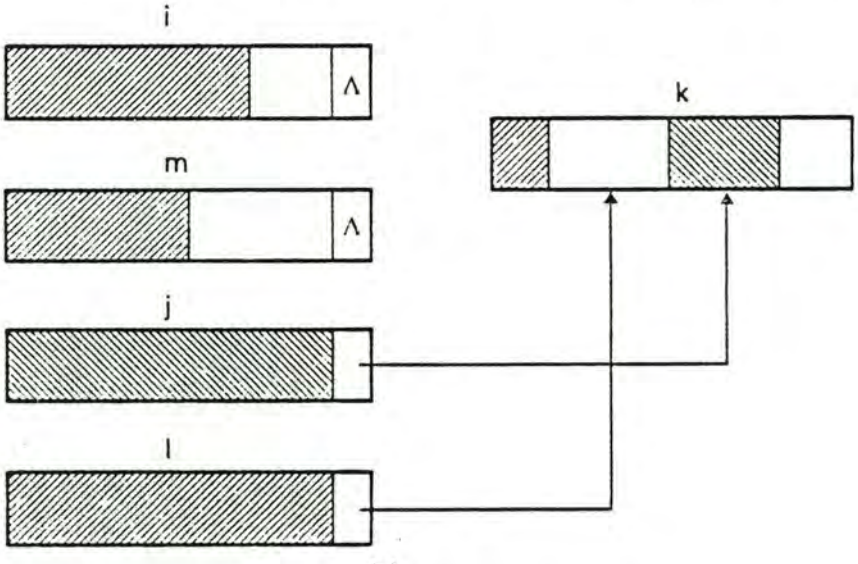
(a)

(a) Before splitting of bucket



(b)

(b) After the splitting of bucket j, k is freed



(c)

(c) k is not freed

Fig 2,7 Dynamic Hashing with deferred splitting : one overflow bucket is shared by two primary buckets

2.3. Linear Splitting

2.3.1. File Structure

4. Introduction

Linear splitting is the second scheme proposed by Scholl [Scholl 1981]. In this technique, splitting is performed every y x b insertions, and the bucket to be split is not necessarily the one in which the last insertion occurred. Compared with the above schemes, this new technique provides a smaller index size and higher storage utilization while the access cost degradation is not dramatic.

5. Presentation of the Scheme

An initial function Ho associates the record's key K with one among M lists, say L. We associate k with a unique entry E in the Lth list. E contains a pointer BKJ to the secondary storage bucket where the record is to be stored, see figure 2.8

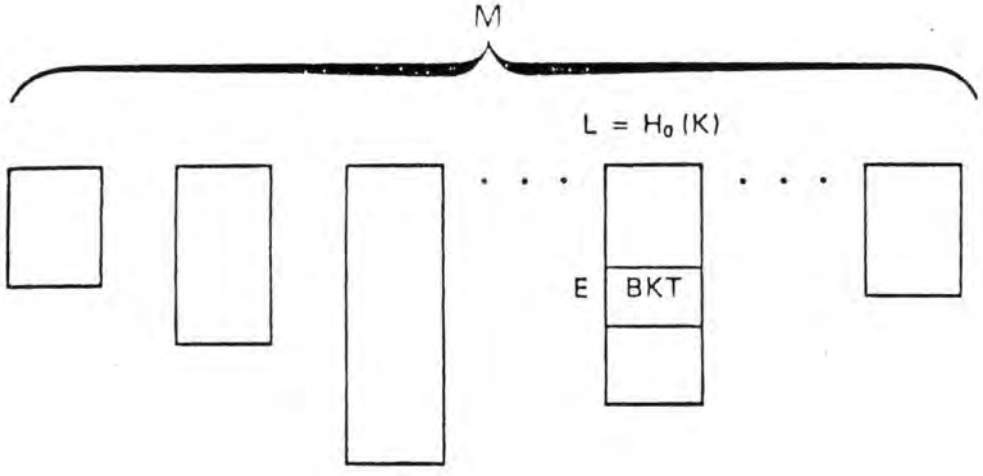


Fig 2.8 Linear splitting : index structure

The initial index is composed of M cells (each list has only one cell), M buckets being initially allocated to the file. After Y x b records have been inserted in the Lth bucket denoted by B1 (Yb records hashed to cell L), bucket B1 is split into two buckets and the records are distributed between B1 and a new bucket B2 whose addresses are contained in two new cells, L(2) and L(3), appended to the Lth list,

initially composed of one cell, L(1). L(1) is deleted.

After Yb new insertions B1 is split once more into two buckets, B1 and B3 whose addresses are contained in L(4) and L(5) appended to the Lth index list and L(2) is deleted. After Yb new insertions, B2 is in turn split into two buckets, B2 and B4, L(3) is deleted, and L(6) and L(7) are appended, which point towards B2 and B4 respectively.

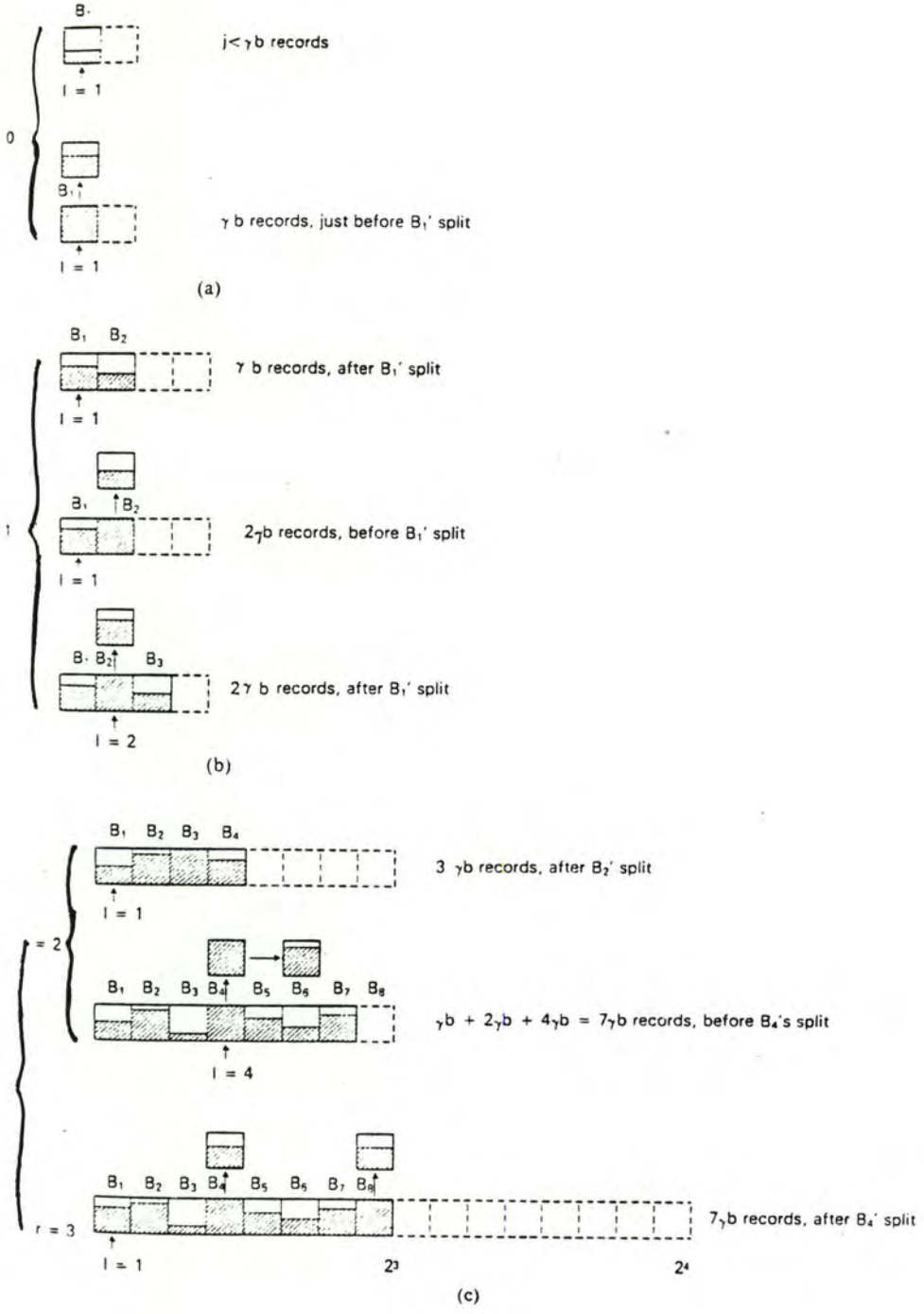
Eventually, as the file grows, any buckets may overflow before they are, in turn, split. Then, the overflow records are stored in one or more overflow buckets. When splitting occurs, one or more new buckets are allocated, while one or more buckets (in the initial chain) are possibly freed.

If the file shrinks, two buckets are merged, the records are gathered into buckets (into one or more overflow buckets), and the index is updated (see figure 2.9).

6. Performance

The file structure with a linear splitting technique may support a much larger number of records than with dynamic hashing before the index overflows on secondary storage. If Y=2.1, retrieval of a record takes on the average 1.8 accesses to secondary storage with linear splitting while with the same Nmax number of records, the dynamic hashing index is mostly stored in secondary storage and retrieval of a record may be slowed down considerably.

In summary, as the number of records actually stored in the file steadily increases, linear splitting not only improves storage utilization, but also uses an index smaller in size while the access cost degradation is not dramatic and in some cases linear splitting may provide better access performance.


B·
j<γb records
l = 1
0
B1'
γb records, just before B1' split
l = 1
(a)
B1 B2
γb records, after B1' split
l = 1
B1 B2'
2γb records, before B1' split
l = 1
B· B2' B3
2γb records, after B1' split
l = 2
(b)
B1 B2 B3 B4
3 γb records, after B2' split
l = 1
B1 B2 B3 B4' B5 B6 B7 B8
γb + 2γb + 4γb = 7γb records, before B4's split
l = 4
B1 B2 B3 B4' B5 B6 B7 B8'
7γb records, after B4' split
l = 1
2³
2⁴
(c)

3. Virtual Hashing Schemes (Hashing Schemes Which Do Not Use Any Kind of Index)

The virtual hashing (VH) schemes proposed by Litwin [Litwin 1978] are similar to extendible hashing, but do not employ any index. Retrieval of a record may then require only one access to secondary storage.

3.1. First Virtual Hashing Scheme

3.1.1. Presentation of the Scheme

Assume that the insertion of a record with a key s leads to a collision and no records already stored in the bucket H(s) could become overflow record. The record may then be stored in its primary bucket only if a new hashing function is chosen.

The new function we shall then call H' should assign new addresses to some of the records hashed with H on H(S) and the file should be reorganized in consequence. If H=H' for all other records, the reorganizing needs to move only a few records and so may be performed dynamically.

The new function is called the dynamic hashing function. The modification to the hashing function is called a "split address". The idea in virtual hashing is to use the splits in order to avoid the accumulation of overflow records and the splits are typically performed during some insertions. All splits result from the application of split functions. The basic split functions are defined as below.

If S is the key space, let ho: S -> {o,1,...N-1} to be the function that is used to load the file. The functions h1, h2, following requirements :

$$hi: S \rightarrow \{0,1,\ldots\ 2\ exp(i)\ N-1\}$$

For any S, either :

$$hi(S) = h(i-1)(S) \qquad (1)$$

or :

$$hi(S) = h(i-1)(S) + 2\ exp(i-1)\ N \qquad (2)$$

For this, we suppose that each hi (i=0,1,...) hashes randomly. This means that the probability that S is mapped by hi to a given address is 1/2exp(i)N. This also means that (1) and (2) are equiprobable events.

The idea of this split is illustrated by the following example :

We suppose that a file F is created with the hashing by

split and is a dynamic hashing function.

Figure (b) shows what has happened to the file. No records other than those hashed to 0 have been moved. The addresses of approximately half the number of the records have been changed and it comes from (1) (2) that all these records had the same new address (which is 100 in the example).

The reorganization is made by applying h1 as the split function and we have performed the split for the address 0. The hashing function h results from the splits and is a dynamic hashing function.

In contrast to what could be accomplished if a classical hashing were used, the split has resolved the collision without creating an overflow record and without access performance degradation.

3.1.2. Performance Consideration

Litwin has shown that a record may be found typically in one access while the load during insertions oscillates between 45 % and 90 % and is 67.5 % on average.He showed also that the average load during insertions may be always greater than 63% and almost always greater than 85% if we accept that the average successful search requires 1.6 accesses.

3.2. Improvement of Virtual Hashing : Linear (Virtual) Hashing

3.3. Introduction

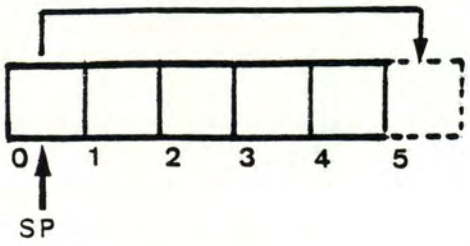
The natural idea in the previous scheme is to split the bucket which undergoes the collision, but split addresses must then be random and this must lead to dynamic hashing functions using tables.

Dynamic hashing functions which do not need tables may be obtained only if the split addresses are chosen in a predefined order instead of splitting the bucket which undergoes the collision. The idea of performing splits in some predefined order is the basis of a new kind of Virtual Hashing technique which is called "linear hashing" [Litwin 1980].

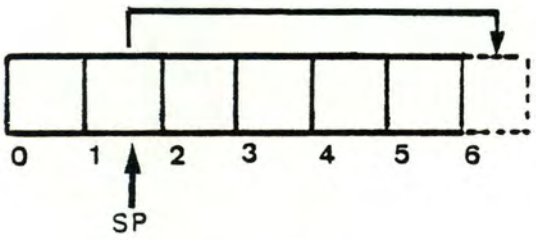
3.4. Presentation of the Scheme

Linear hashing (see fig 3.2) increases the storage space gradually by splitting the primary buckets in an orderly fashion: first bucket 0, then bucket 1,...

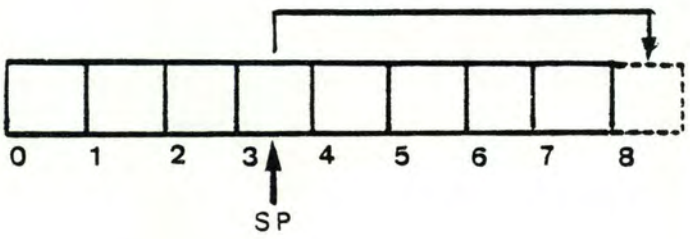
If m is the address of a collision and n the address of a split to be performed in the course of the resolution of this collision, the values of m are random while those of n are predefined (n =/ m). We assume that the new record is stored as



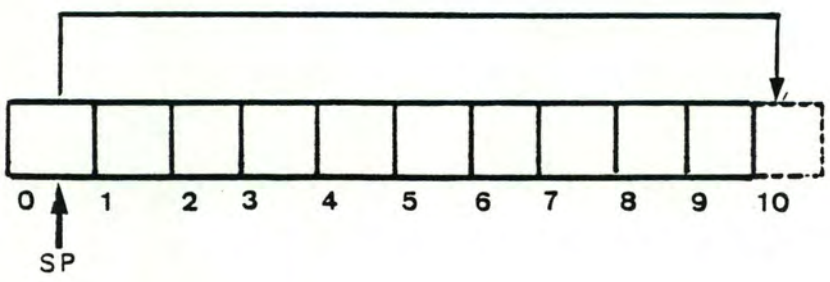
(a) At an initial stage (number of buckets=5)



(b) After one split



(c) After three splits



(d) After the doubling of the file size

overflow record by a usual classic collision resolution method such as chaining for example.

Next, we assume that a pointer P keeps track of which bucket is the next to be split. For the first N collisions, the buckets are pointed in the linear order 0,1,2,...N-1 and all splits use h1.

(2) (see section 3.1) implies then that the file becomes progressively larger, including the buckets N+1, N+2,...2N-1 one after another. A record to be inserted undergoes a split usually not when it leads to the collision, but with some delay. The delay corresponds to the number of buckets which have to be pointed while the pointer travels up, from the address indicated in the moment of collision to the address of this collision.

With this mechanism, no matter what the address is of the first collision, let it be m1, LH performs the first split using H1 and for the address 0. The records from the buckets 0 are randomly distributed between bucket 0 and a new bucket N, while, unless m1=0, an overflow record is created for the bucket m1.

The second collision, no matter what its address is, m2 for example, leads to an analogous result, except that first it splits for the address 1 and appends the bucket N+1. Next, it may constitute the delayed split for the first collision, suppressing therefore the corresponding overflow record. This process continues for each of the N first collisions, thus moving the pointer step by step up to the bucket N-1. Sooner or later, the pointer points to each m and the splits, despite being delayed, move most of the overflow records to the primary bucket.

We may therefore reasonably expect that, for any m < N, only a few overflow records exist. After N collisions, we have H = H1.

The function hi :

$$S \longrightarrow \{0,1,\ldots\ 2\ \exp(i)\ N-1\}$$

implies then that, instead of the hashing on N addresses, we now hash on 2N addresses. The 2 conditions (1), (2) imply that h2 has on the hashing with h1, the action analogous to that of h1 on the hashing with h=ho, except that it hashes on 4N addresses.

We therefore assume that n=0 again, now that we split with h2, and that the upper bound on n is 2N-1. For further insertions, we use h3, h4,... hj, while the pointer travels each time from 0 to 2 exp(j-1).

It results from the above principles that first,the address space increases linearly and is as large as needed. Next,for any number of insertions, most of the overflow records are moved to the primary buckets by the delayed splits.

3.4.1. Performance Consideration

The choice of file parameters may lead to a mean number of accesses per successful search close to 1 while the load stays close to 60%.

It may also lead to a load staying equal to 90% while the successful search requires 1.35 accesses in the average.

3.4.2. Control Functions

We must have rules for deciding when the splitting of the next bucket is to take place. Several alternatives are possible. Litwin has investigated two strategies called "uncontrolled" and "controlled splitting".

Uncontrolled means that the next bucket is split whenever an inserted record is placed in an overflow bucket.

Controlled splitting allows splitting to take place only when a record is placed in an overflow bucket and the storage utilization is above some predetermined threshold, e.g, 75%.

This leads to considerable differences in performance. Uncontrolled splitting results in low storage utilization (e.g. 60%) and fast retrieval. By controlled splitting, better storage utilization can be achieved, but retrieval is slowed down.Performance figures based on simulations can be found in [Litwin 1980].

4. Linear Hashing With Partial Expansions [Larson 1980]

4.1. Introduction

Linear hashing with partial expansions presented by P. Larson [Larson 1980] is a generalization of linear hashing developed by W. Litwin.

In hashing techniques, the best performances are achieved if records are distributed as uniformly as possible among the buckets in the file.

Unfortunately, the record distribution of linear hashing does not reach this goal because the load factor of a bucket already split is only half the load factor of a bucket not yet split. This new technique tries to remedy this disadvantage and thus increase the performance of the file.

4.2. Presentation of Linear Hashing With Partial Expansion

In this scheme, the difference with linear hashing is due to the fact that the doubling of the file size is done in a series of partial expansions.

Initially the file consists of n0 * N buckets logically subdivised into N groups of n0 buckets each, n0 >= 1, N >= 1. A group i consists of the buckets (i, N + i, 2N + i ,....., (n0-1) N + i) with i = 0,1,....,N-1. Fig 4.1 shows a file at an initial stage with N = 3 and n0 = 2.

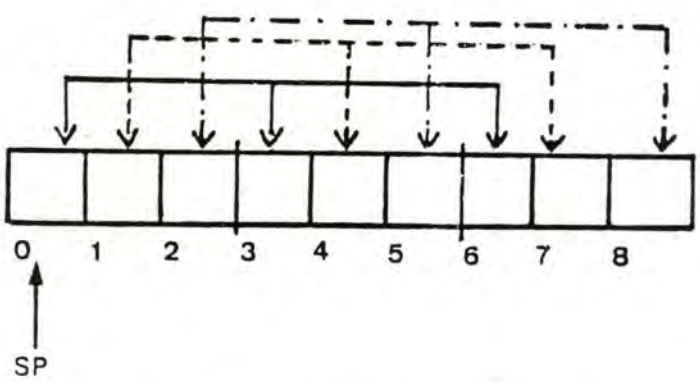


Fig 4.1 Linear Hashing file with partial expansions (N=3 and n0=3) at an initial stage

A full expansion results in a doubling of the file size and is accomplished by a sequence of n0 partial expansions increasing the size of each group to n0 + 1, n0 + 2 ,....., 2n0 respectively.

A partial expansion is carried out stepwise by adding one bucket to each group always in a predefined order : group 0, group 1 ,..., group N * 2 exp L - 1. Each complete partial expansion increases the file size by N * 2 exp L buckets.

The number of full expansions which have been accomplished is indicated by a variable L. The smallest file size on level L is no * N * 2 exp L buckets and the buckets (j, N 2 exp L + j, 2 * N 2 exp 1 + j ,....., (no-1) * N * 2 exp 1 + j) with j = 0, 1 ,.... ,N * 2 * exp L - 1 forms N * 2 exp L group of buckets.

Fig 4.2 shows a linear hashing file with two partial expansions (n0 = 2). The number of groups is 3 (N = 3).

In the example represented by the figure 4.2, the doubling of the file size is done in two steps ; the first expansion increases the file size to 1.5 times the original size, while the second expansion increases it to twice the original size.

We start with a file of 6 buckets, logically subdivided into 3 pairs of buckets, where the pairs are (j,j+N) vj=0,1,2.

When new storage is needed, according to some rule, the file is expanded by one bucket, bucket 6, and part of the record in bucket 0 and bucket N are moved to bucket 6. When more space is required, the pair (1,4) is expanded.

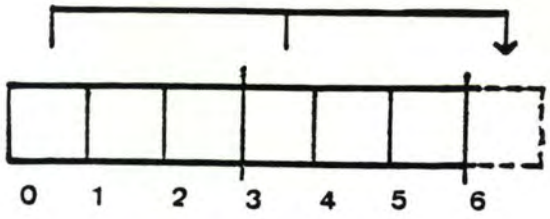
When the last pair (2,5) has been expanded, the file size has increased from 6 to 9. Thereafter the second expansion starts, the only difference being that now 3 groups of 3 buckets are considered (0,3,6), (1,4,7), (2,5,8).

When the second partial expansion has been completed, the file size has doubled from 6 to 12.

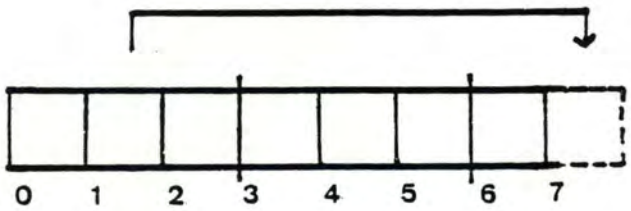
However, we do not wish to continue expanding groups of four buckets, then five... We want to come back to groups of two buckets (j, j+6) j = 0,1,2,3,4,5 (figure 1.3). If this were not the case, the cost of expanding a group would steadily increase and it would soon become prohibitively large.

Another important point is that when a group of buckets is expanding, it should not be necessary to rearrange records among the old buckets. We must simply scan through the old buckets and collect only the records which are to be reallocated to the new bucket. In this way, the expansion can be made in one scan, and no jumping back and forth is necessary. The solution for this last point is quite simple and uses the rejection technique.

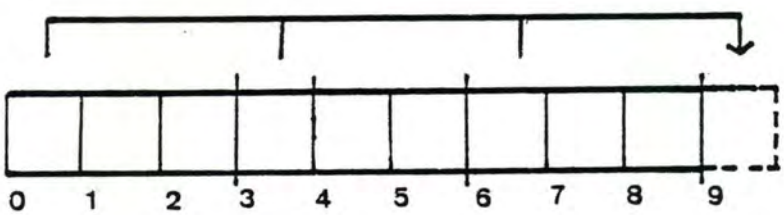
The rejection technique assumes that H(K) = (h1(K), h2(K), h3(K)..) is a sequence of hashing functions, where each hi hashes uniformly and independently over {0,1,2,...n-1}.
Furthermore, we suppose that a fixed set of records is to be stored in a file consisting of only $m' < m$ buckets. To find the address of a record with a certain key K, we compute the numbers...h1(K), h2(K)... and takes as the address of the



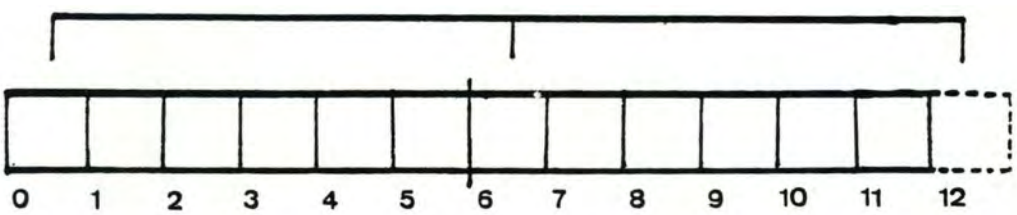
(a) At the initial stage



(b) After one split



(c) After one partial expansion



(d) After two partial expansions (doubling of the file)

FIG 4.2

record the first number which is a valid address,i.e, which is less than m' among the buckets 0,1,...m'-1.
Once the file is extended by one bucket, m', and the address of every record in the set in recomputed, the same hashing functions h1,h2,... are used, but this time any number less than m'+1 is a valid address. When the number of buckets pass from m'-1 to m', the address of some of the records has changed to m' :

Example: if m=4 and m'=2, we consider the sequences

H(l1)=(1,3,2,...), H(l2)=(2,3,0,...), H(l3)=(3,1,2,...), and H(l4)=(3,2,1,...)

If the file has 2 buckets, the records will be assigned to bucket 1,0,1,1, but if m' is increased to 3, the addresses of the records are 1,2,1 and 2. In this case, records 2 and 4 must be moved to the new bucket. If m'=4, the addresses are 1,2,3 and 3. This time record 3 and record 4 must be moved.

If you want to reduce the size of the file by one bucket, the new addresses are computed for the records located in the bucket which is to be deleted and the records are inserted into their new bucket. No other records have to be moved.

4.3. Control Function

Larson suggests that the expansion of the file should be controlled solely by the overall storage utilization (including the number of overflow buckets in use). When a record has been inserted into the file, storage utilization is checked and if it is higher than some fixed threshold w, $0<w<1$, the file is expanded by one bucket. This implies that we have a control function that keeps track of the number of records in the file and the number of overflow buckets.

This control function seems optimal because when we use linear hashing with partial expansion, there is always a trade-off between storage utilization and the expected length of successful searches.

The higher the storage utilization is, the longer the searches are expected to be. Both factors cannot be controlled simultaneously. Larson suggests that we first control the storage utilization by requiring that it should always be >= than the threshold, but once this threshold has been reached, we minimize the time of searches by keeping the storage utilization as close to the threshold as possible.

4.4. Performance

A detailed performance analysis can be found in [Larson 1982] The analysis reveals that an average search length in the range of 1.1-1.2 accessses can be achieved with the same parameters.

Furthermore, we can say that the expected number of accesses required to insert a record, also including accesses required to physically store a record and to update the record counter, the accesses required to rearrange records in the old bucket, reach between 4.37 and 6.37 accesses for load storage as high as 85%-90% and a bucket size of 50 records. The choice of two partial expansions seems to be a good compromise.

In summary, linear hashing with partial expansions offers a new and simple technique for organizing dynamic files. Retrieval of a record is very fast by any standard and files have a constant storage utilization up to 0.90 with excellent performance. The performances deteriorate rapidly if the storage utilization is further increased.

5. Variation of Larson and Litwin's Scheme [Lloyd and Ramamohanorao to be published]

5.1. Introduction

Recently, two variations of the schemes presented above appeared in [Lloyd and Ramamohanorao to be published]. These schemes seem to be the most powerful dynamic hashing schemes up to now. They have certain performance advantages over earlier schemes because they implement the ideas of Larson and Litwin in a simpler way.

5.2. Dynamic Hashing Scheme With Round-Up Pages

As the file grows, it goes through a series of expansions. At the beginning of such an expansion (see Figure 5.1.a) the file consists of certain home pages plus their associated overflow pages. Each home page has its own, possibly void, chain of overflow pages, which contain records that would not fit into the home page.

The home pages are divided into S groups of g pages. S is the current segment size. g is a parameter, characteristic of the file, called the group size. The home pages are indexed by 0,1,...,gs-1. The g home pages indexed by j, j+s,..., j+(g-1)S together form a group of buddy pages (j=0,1,..S-1). Litwin's scheme can be obtained by pulling g=1.

To add a new home page, we need a pointer called the split pointer, SP, which indicates the next group of buddy pages to be split. At the start of an expansion, SP points to the first group of g buddy pages indexed by 0,S,...,(g-1)S.

After exactly L insertions, this group is split and an extra

home page is appended to the end of the file. Then, the records in the home pages 0,S,...(g-1)S plus the records in the overflow pages associated with these home pages are redistributed, according to a certain hash function, amongst the g+1 home pages 0,S,..gS.

If the hash function is effective, we can expect overflow chains for these home pages to be reduced, and, perhaps, disappear. The split pointer is then moved forward to the next group of buddy pages 1, S+1,... (g-1)S+1. When the split pointer moves to the last group of buddy pages S-1,...2S-1 and these pages are split, the split pointer will return to the beginning of the file (so that SP=0). During the expansion, the file grew from gS home pages to (g+1)S home pages and exactly SL records were inserted.

However, what we would like is to divide the current file into groups of g buddy pages and enlarge the segment size. The problem in this scheme is that the current file size in terms of home pages will not generally be divisible by g. To make it possible, we append if necessary, r extra home pages to the end of the file, where 0 <= r < g. The extra pages added are called round-up pages.

Larson's method of splitting a group of buddies is more complicated because the file goes through a series of partial expansions which together constitute a full expansion.

In each partial expansion the group size is different and thus the effect on this is to give variable performance during full expansions. In keeping the group size constant, this leads to more uniform performance.

The only complication in this scheme is that we need round-up pages, but the number of round-up pages added in each is bound by a small constant which is certainly negligible compared with the file size. This scheme is the natural generalization of Litwin's scheme.

6. Dynamic Hashing Scheme for Secondary Key File [Lloyd, Ramamohanorao and Thom 1983]

6.1. Introduction

The scheme presented in the section above has the disadvantage of letting the user access a record only by means of a primary key.

In the case of secondary keys (we want to know how the location of a record via keys which do not uniquely identify one record), the pure hashing schemes are not very efficient. To solve this problem, we will describe a partial match retrieval scheme based on hash functions and descriptors. See [Lloyd and Ramamohanorao to be published], [Lloyd 1980], [Lloyd and Ramamohanorao 1982], and [Lloyd, Ramamohanorao and Thom 1983].

6.2. Definition of a Partial Match Query

Each record in the file consists of a number of fields (secondary keys) which may be specified in a query.

Assume that there are K fields f1,....,fn which may be specified in the query. Then a partial-match query is a specification of the value of one or more of the fields f1,...,fk. An answer to a query is a listing of all records in the file which have the specified values for the specified fields.

6.3. Description of a Simple Partial-Match Retrieval Scheme (When the File is Known) Based Purely on Hashing

The records of the file are contained in a number of pages. We suppose first that the file is static and consists of 2 exp(d) pages. (d is a fixed, non-negative integer). The pages are numbered 0,1,...2 exp(d)-1.

There are K hashing functions hi, the ith function mapping from the key space of the field fi to the set of the strings of di bits, where each di is a non-negative integer and d1 +...+ dk = d.

The page in which a particular record is to be stored is computed as follows. Each field fi is hashed to a string of di bits. The string resulting from the concatenation of these strings (in order) gives the page number.

The hashing functions should be chosen to distribute the records as evenly as possible amongst the pages.

The problem at this stage is to minimize the average number of home pages which have to be accessed to answer a query.

Let Q be a query, so that Q is included in 1,2,...,k. We denote by PQ the probability that the query Q is specified. Thus, P =1 and P(Q)>=0 for all Q. The P 's for a particular

system are determined by the use made of that system.

Then the average cost of a query is :

A =

The optimization problem is thus to find d1,...,dk that minimize the objective function A and satisfy d1 +...+ dk = d.

Like this, it is too simplistic because another difficulty arises with the key space of each field.

For example, often fields have rather small key spaces, and thus must be allocated only one or two bits. Constraints of the form di <= dmax naturally arise where 2 exp(dmax) is the number of values that a particular field can take. For example, sex can have only two values and thus no more than one bit should be allocated to it.

On the other hand, if the field f1 has a large key space of 2 exp(C1) values, the optimization problem assigns d1 bits to f1 with C1>>d1. A pure hashing scheme cannot cope with it because the total number of bits, d, allocated is determined by the size of the file space and there are a number of fields competing for bits.

As the number of bits allocated to f1 is severely limited, the hash function will map many different values of the same field in the same bit string. A large amount of information is lost in this case. One suggestion is to use a small, simplified, descriptor file, built on top of a hashing scheme, so that before any page is accessed, a check on its descriptor is made. A scheme using descriptors was developed [Lloyd and Ramamohanorao to be published]. We briefly review this scheme.

6.4. A Descriptor Scheme

1. Desciptor, Page Descriptor, File Descriptor

At each file is associated a descriptor file. This decriptor file is composed of a set of descriptors page.

A descriptor page is a set of descriptors where each descriptor is related to a page of the main file. A descriptor is simply a bit string of W bits (fixed length).

Each record R in the data file has a descriptor Dr associated with it. This descriptor is derived from the values (V1,V2,...,Vf) of the F attributes of the record R. Fig 6.1 shows a basic scheme of a file and its descriptor.

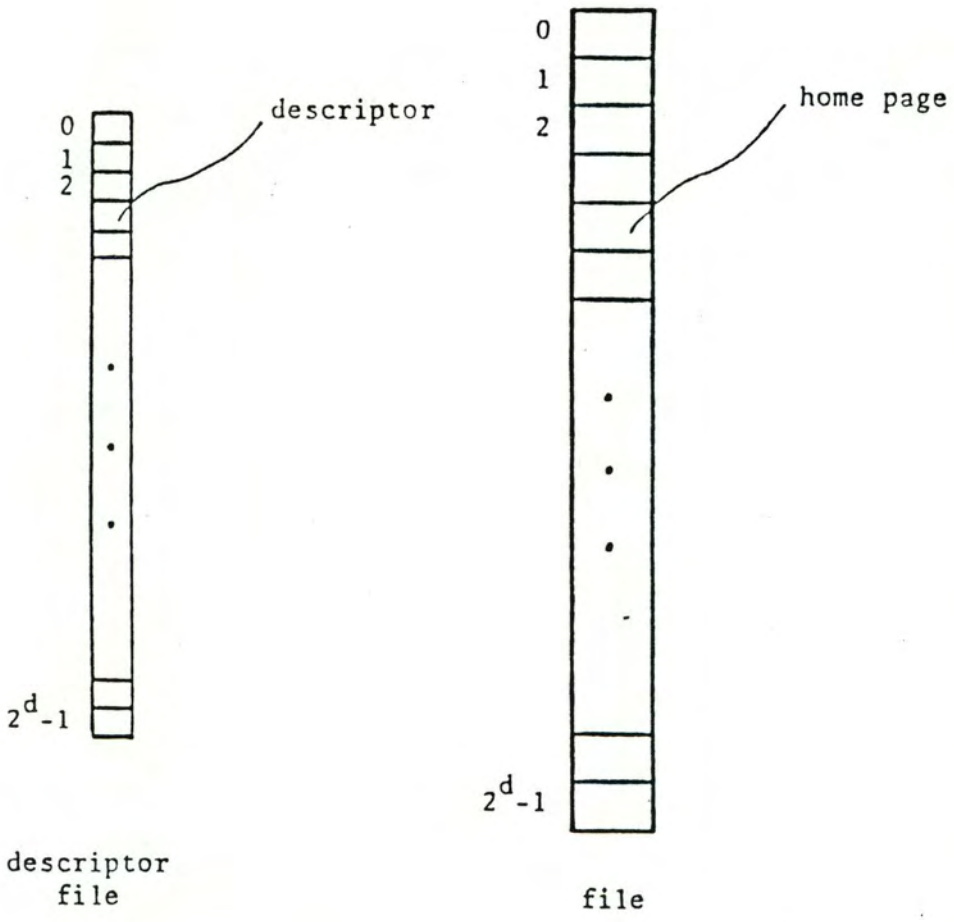


Fig 6.1 Basic scheme

2. Constructing a Descriptor

One possibility in constructing the descriptor is to employ the method of disjoint coding.

Disjoint coding begins by dividing each descriptor into F disjoint fields (each record has F attributes). Each field Fj consists of Wj bits and the sum of all values Wj from j=1 to j=f is equal to W.

Each of the attributes has an associated transformation Tj which maps from the key space of Fj to the subset of bit strings of length Wj.

To describe a record R, these transformations are applied to each of the attribute values of R and the Tj(vj)th bits in Wj is set to 1 while the remainder Wj-1 bits are set to 0.

Each descriptor will have exactly F bits set to 1. In a partial-match query, attribute values are specified for only a subset of the attributes. If Q<=F is such a subset, the query descriptor is built in the same way as the record descriptor.

The transformation Tj is applied to the attribute value Vj to determine which bit in Wj(Q) is set to 1. If all the possible values are not specified in the query, the bits corresponding to the fields not specified are set to 0.

As the descriptor of Dr and Q has been constructed using the same transformation, we can make the following propositions :

- If R satisfies the partial match query, then Q <= Dr.
- If Q is not a part of Dr, then R does not satisfy the partial match query.
- If Q is a part of Dr, then R may or may not satisfy the query.

"Q is a part of Dr" means that every bit position which is 1 in Q is also a 1 in Dr, and "Q is not a part of Dr" means that there is at least one bit position which is 1 in Q and 0 in Dr. With these propositions, we can construct a descriptor file which allows us to check if an information is contained in a page and thus access this page only if we are sure that we can find the information in this page.

In the scheme proposed by [Lloyd and Ramamohanorao 1982] ,a page descriptor is constructed by applying the logical function OR to the descriptors of the records contained in the pages of the main file and any overflow page.

3. Using a Descriptor File

The descriptor file is used as follows. Let Q be a query. Using the hash function Hi on the specified fields, a set of addresses of pages which can contain records in the answer to the

2. Constructing a Descriptor

One possibility in constructing the descriptor is to employ the method of disjoint coding.

Disjoint coding begins by dividing each descriptor into F disjoint fields (each record has F attributes). Each field Fj consists of Wj bits and the sum of all values Wj from j=1 to j=f is equal to W.

Each of the attributes has an associated transformation Tj which maps from the key space of Fj to the subset of bit strings of length Wj.

To describe a record R, these transformations are applied to each of the attribute values of R and the Tj(vj)th bits in Wj is set to 1 while the remainder Wj-1 bits are set to 0.

Each descriptor will have exactly F bits set to 1. In a partial-match query, attribute values are specified for only a subset of the attributes. If Q<=F is such a subset, the query descriptor is built in the same way as the record descriptor.

The transformation Tj is applied to the attribute value Vj to determine which bit in Wj(Q) is set to 1. If all the possible values are not specified in the query, the bits corresponding to the fields not specified are set to 0.

As the descriptor of Dr and Q has been constructed using the same transformation, we can make the following propositions :

- If R satisfies the partial match query, then Q <= Dr.
- If Q is not a part of Dr, then R does not satisfy the partial match query.
- If Q is a part of Dr, then R may or may not satisfy the query.

"Q is a part of Dr" means that every bit position which is 1 in Q is also a 1 in Dr, and "Q is not a part of Dr" means that there is at least one bit position which is 1 in Q and 0 in Dr. With these propositions, we can construct a descriptor file which allows us to check if an information is contained in a page and thus access this page only if we are sure that we can find the information in this page.

In the scheme proposed by [Lloyd and Ramamohanorao 1982] ,a page descriptor is constructed by applying the logical function OR to the descriptors of the records contained in the pages of the main file and any overflow page.

3. Using a Descriptor File

The descriptor file is used as follows. Let Q be a query. Using the hash function Hi on the specified fields, a set of addresses of pages which can contain records in the answer to the

query is generated.

However, before these pages are accessed, we check the descriptor file. Corresponding to Q, there is an associated query descriptor (with the same structure as a page descriptor), which is obtained by transforming the fields specified in Q using the Ti's and making up the remainder of the query description with 0's in the bit positions corresponding to the unspecified fields.

Then before accessing a page, we compare the query descriptor with the descriptor for that page. If the query descriptor has a 1 in a bit position where the page has 0, then the page cannot possibly contain a record in the answer to Q and hence, the page does not have to be accessed.

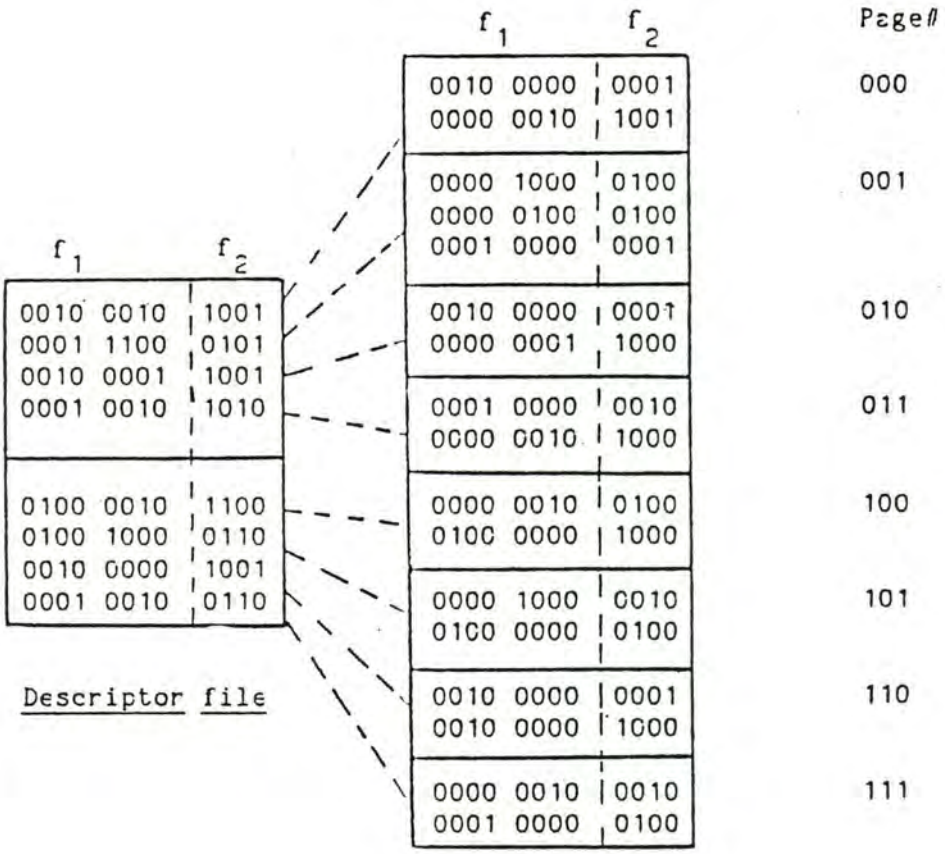
The advantage of descriptors is that they can add more knowledge about records actually present in the file. When a record is added to or deleted from a page, the descriptor must be updated.

Example:

Assume that we have a record type with two fields and that a query was made on the first field of this record type with the value V1.

If only one bit is allocated to this field then the hashing function can hash to the set of addresses beginning by 1 or 0. If we assume that the hashing function hashes to 1 then the set of home pages to be searched without the descriptor file is the following : pages 100, 101, 110, 111.

Suppose the query Q gives the query descriptor 001000000000. Page 100, which has a descriptor 010000101100 does not have to be accessed, since it has a 0 in a bit position where the query descriptor has a 1. On the other hand, it will be necessary to access page 110 with descriptor 001000001001.



d=3 k=2 d1=1 d2=2 W1=8 W2=4 W=12

It has to be noted that record descriptors are shown in the main file only for explanation purposes.

6.5. Extension of the Scheme to Dynamic Files

6.5.1. Presentation of the Scheme

The partical-match retrieval scheme described in the section above is only suitable for static files, but it is easy to extend it to dynamic files by utilizing the linear hashing scheme discussed earlier. The scheme used here is linear hashing.

A Linear Hashing file is shown at a typical stage in its existence in fig 6.2. The file is currently stored in M pages, numbered from 0 up to M-1. Some pages have short overflow chains containing records that would not fit into the home pages.

The file also has a split pointer which indicates the next page to be split. This page is numbered sp. Note that the page to be split is selected independently from where the collisions occur. The split pointer moves in a very systematic way, first page 0, then page 1,... splitting each page in turn.

When page 2 exp(d-1) splits, the split pointer returns to page 0. On the next run, the split pointer will go up to 2(exp(d+1))-1. Thus, the file doubles during a complete run of the split pointer. d is now a variable called the depth of the file.

The splitting policy employed here is to split after every L insertions into the file. L is called the load control and must be carefully chosen to maintain a desired load factor.

When the page sp is split, the following occurs :

- For each record in the page sp, and for any associated overflow records, a new hash address is computed.

- For each such record, the new address will either be sp (the old address) or M=SP+2 exp(d). A new page, numbered M, is then appended to the end of the file and the records with hash address M are put into this page.

Since pages split in this very systematic way, the need for a directory is obviated. Furthermore, even though the file may have grown and the record moved since it was first inserted, it is still possible to calculate directly the home page of any record in the file.

The important part of the extension is the choice of the hash function. A more complicated way of constructing the hash function is needed because the file is no longer static.

For each Fi, we have a hash function Hi mapping

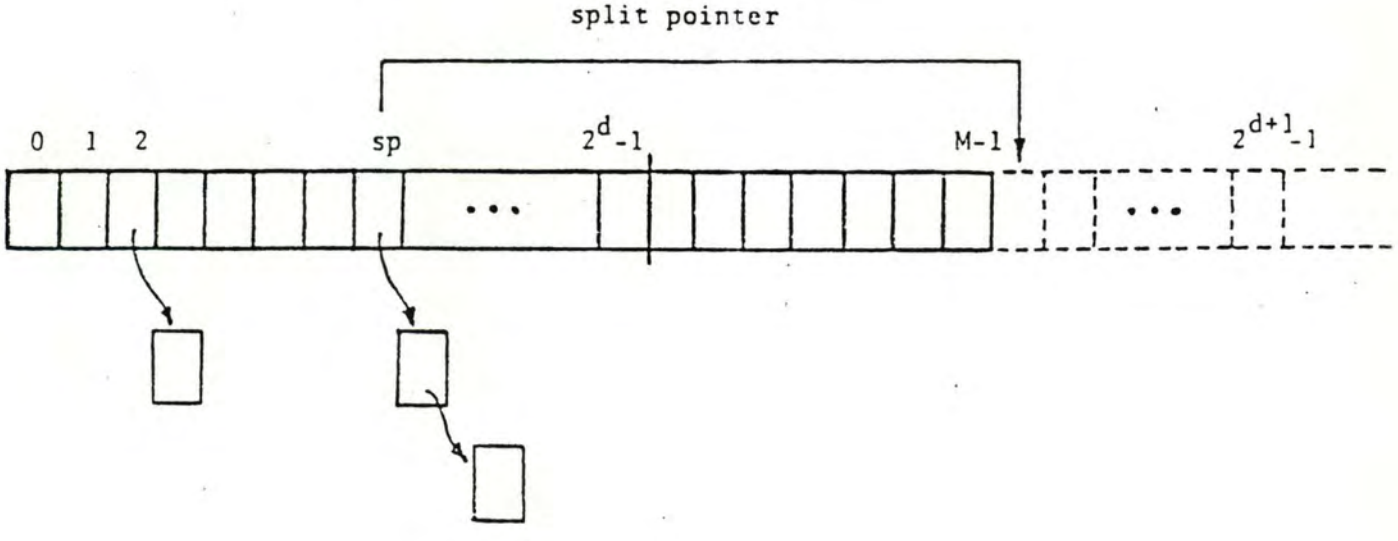


Fig 6.2 Typical stage of a Linear Hashing scheme

from the key space of Fi to bit strings of an suitably length. We now no longer concatenate the strings Hi(Vi) as before, but we compute the new address with what we called the " choice vector ".

6.5.2. Choice Vector

We are now explaining what a "choice vector" is and how to use it to compute the address of a record. After we will see from where it comes.

A " choice vector " contains numbers (I1, I2 ,.....) which are integers between 1 and K. Each integer indicates which field is taken into account to compute the address of a record. Let see with an example how this "choice vector " is used :

Example :

Assume the choice vector is (4,5,4,3,2,3...) and vK=6. The right-most bit of the bit string forming the address is the first bit in the string H4(V4), the second from right is the first bit in the string H5(V5), the third is the second bit in the string H4(V4), and so on.
In general, the ith bit from right in the bit string forming the address will be the first so far unused bit in the string Him (Vim). In this case, the record is said to hash to an interlaced bit string.

This choice vector comes from the optimization problem. The problem is to minimize :

subject to where each di is a non negative integer.

[Lloyd and Ramamohanorao 1982] proposed in their paper an algorithm which computes the optimal number of bits, and the optimal Di and Wi at each depth.

What can happen is that a particular field's allocation of bits in the optimal solution at one depth can be higher than its allocation at the next highest depth.

This implies removing a bit from the middle of a hash address when the depth changes, but in this case we can avoid a complete reorganization of the file during the change in depth to handle this.

Thus, the allocation of di values, as the depth increases, should have the following property :

If di bits are allocated to a field fi at depth
di and di' at depth d + 1, then di <= di'.

This property is called the monotonicity property.

The algorithm provided in [Lloyd and Ramamohanorao 1982] computes also the "choice vector".

6.5.3. File Descriptor

The descriptor file grows and contracts in parallel with the linear hashing file. However, no matter what the depth of the linear hashing file, the descriptor size is a constant W bits.

The construction and the information appearing in the descriptor are similar to those of section 6.4. Maintenance of a descriptor is also easy and is made in parallel with the maintenance of the LH file. Fig 6.3 shows a Linear hashing file with its descriptor file.

6.6. Performance

The descriptor of a page must be updated whenever a record is inserted into a linear hashing file. This involves computing the descriptor of the record and applying the logical function OR to the old descriptor and the new descriptor associated to the page where the new record has been inserted and the new descriptor.

If a record is deleted the cost is slightly more expensive because the descriptor must be recomputed.

The cost of maintaining the descriptor file for an insertion or deletion is 2 disk accesses: one to read the descriptor, and one to write it. For a split, the cost is 4 disk accesses: one to read the descriptor, one to read the page, and two to write the two new descriptors.

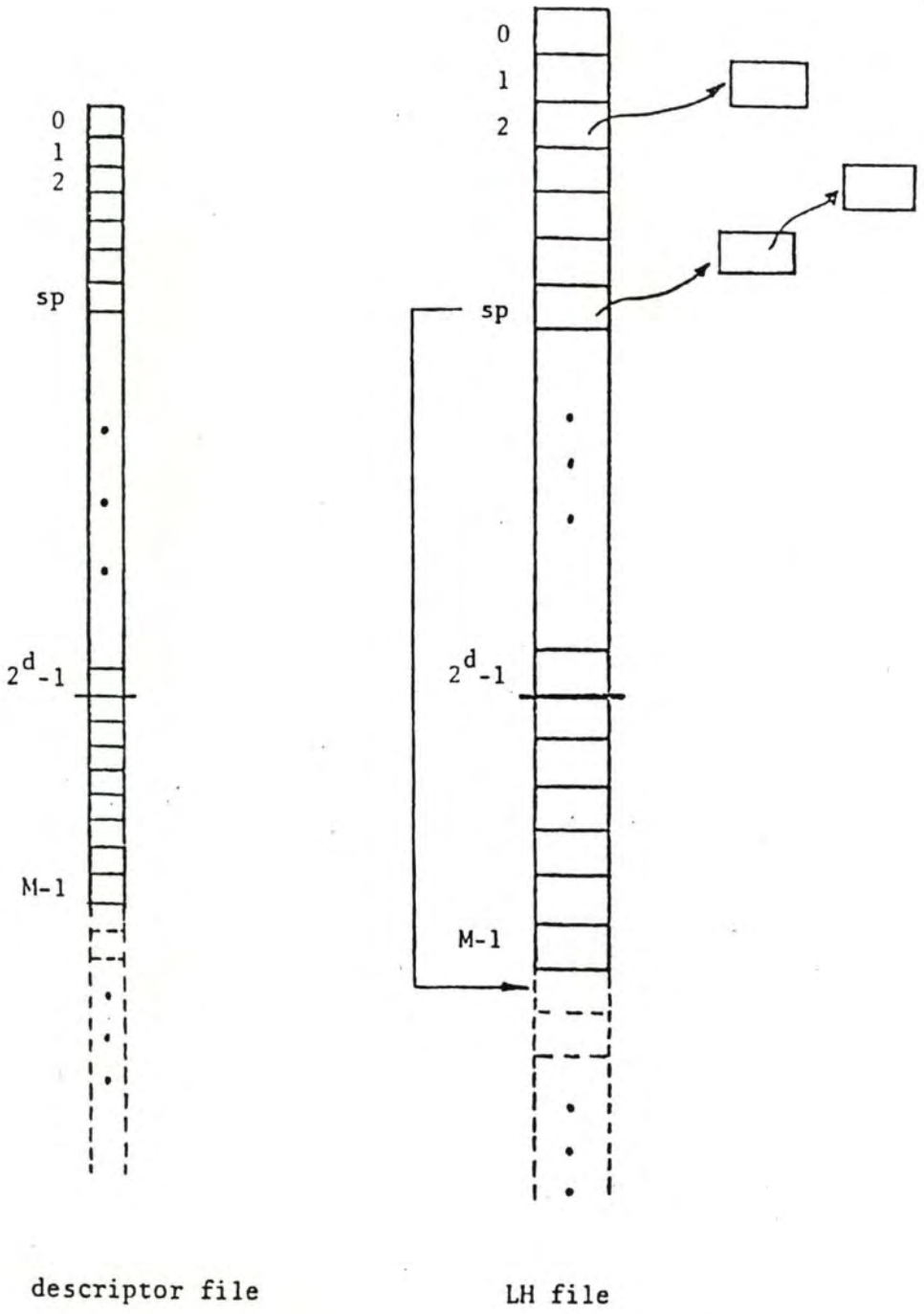


Fig 6.3 Linear Hashing scheme and descriptor file

7. REFERENCES

[Aho 1979]

Aho, A.V. and J.D. Ullman. "Optimal Partial-Match Retrieval When Fields are Independently Specified," ACM Transactions on Data Base Systems, Vol. 4, No. 2, June 1979, pp. 168-179.

[Fagin 1979]

Fagin, Ronald (IBM Research Laboratory), Jurg Nievergelt (Institut Informatik), Nicholas Pippenger (IBM T.J. Watson Research Center), and H. Raymond Strong (IBM Research Laboratory). "Extendible Hashing - A Fast Access Method for Dynamic Files," in ACM Transactions on Data Base Systems, Vol. 4, No. 3, September 1979, pp. 315-344.

[Knuth 1973]

Knuth, D.E. The Art of Computer Programming, Vol. 3: Sorting and Searching. Addison-Wesley, Reading, MA, 1973.

[Larson 1978]

Larson, P. "Dynamic Hashing," BIT 18 (1978), pp. 184-201.

[Larson 1980]

Larson, P. "Linear Hashing With Partial Expansions," Proceedings, 6th International Conference on Very Large Databases, 1980, pp. 224-232.

[Litwin 1980]

Litwin, Witold. "A New Tool for File and Table Addressing," in Proceedings on Very Large Databases, Montreal, 1980, pp. 212-224.

[Litwin 1978]

Litwin, Witold. "Virtual Hashing: A Dynamically Changing Hashing," Proc. 4th Conference on Very Large Data Bases, West Berlin, Sept. 1978, pp. 517-523.

[Lloyd 1980]

Lloyd, John W. Optimal Partial-Match Retrieval, BIT 20, 1980, pp. 406-413.

[Lloyd and Ramamohanorao to be published]

Lloyd, John W. and K. Ramamohanorao. "Dynamic Hashing Schemes," Department of Computer Science, University of Melbourne, to be published.

[Lloyd and Ramamohanorao 1982]

Lloyd, John W. and K. Ramamohanorao. "Partial-Match Retrieval for Dynamic Files, BIT," Department of Computer Science, University of Melbourne, BIT 22 (1982), pp. 150-168.

[Lloyd, Ramamohanorao and Thom 1983]

Lloyd, John W., K. Ramamohanorao, and James A. Thom. "Partial-Match Retrieval Using Hashing and Descriptors," Dept. of Computer Science, University of Melbourne, to appear in ACM Transactions on Data Base Systems, 1983.

[Martin 1975]

Martin, James. Computer Data Base Organization, Prentice Hall.

[Scholl 1981]

Scholl, Michel. "New File Organizations Based on Dynamic Hashing," in ACM Transactions on Data Base Systems, Vol. 6, No. 1, March 1981, pp. 194-211.

ANNEX 2 : Description of subroutines

1. DBHUSE

These routines are the highest level routines that interface directly with the user's program.

1. CRENIN

Calling Convention :

```
CALL CRENIN (entyna, bufpar, ptrpar, ierr)
```

Purpose :

to create an entity instance

Description :

An entity whose values are contained in a buffer is added to the set of the entities identified by the given entity type name. The subroutine checks if the primary key already exists or is acceptable.

If the primary key already exists we can not insert the entity because a primary key uniquely identifies an entity. The primary key is not acceptable means that one of the values of the primary key does not belong to the range of allowable values defined for this entity type.

Arguments :

NAME	USAGE	TYPE	DESCRIPTION
entyna	input	char	entity type name
bufpar	input	char	buffer containing the values of the items
ptrpar	input	int	pointer into bufpar where the description of the entity begins
ierr	output	int	return code

2. CHENIN

Calling Convention :

```
CALL CHENIN (dbkey, attnam, vsnam, newval, ierr)
```

Purpose :

change the value of an attribute of an entity

Description :

An entity identified by a data base key (a data base key indicates the location ,i.e, the page number and a displacement within the page, of the primary key) is modified in the following way: the old value of a value set identified by its name and an attribute type name is replaced by a new value. If the value set is not a part of the primary key ,there is no consequence.

Otherwise we must change the primary key of this entity in all the relationships in which the entity is involved.

Arguments :

NAME	USAGE	TYPE	DESCRIPTION
dbkey	input	int(2)	data base key
attnam	input	char	attribute type name
vsnam	input	char	value set name
newval	input	char	new value
ierr	output	int	return code

3. DELENT

Calling Convention :

CALL DELENT (dbkey, ierr)

Purpose :

delete an entity

Description :

An entity identified by a database key is removed from the database. All relationships involving this entity are also deleted.

Arguments :

NAME	USAGE	TYPE	DESCRIPTION
dbkey	input	int(2)	database key
ierr	output	int	return code

4. RETENT

Calling Convention :

CALL RETENT (entnam ,pkey ,dbkey ,ierr)

Purpose :

to retrieve an entity

Description :

An entity identified by its primary key and its type is searched for in the database.The database key locating the entity is given as output. If the entity is not found then the database key is = 0.

Arguments :

NAME	USAGE	TYPE	DESCRIPTION
entnam	input	char	entity type name
pkey	input	char	primary key
dbkey	input	int(2)	database key
ierr	output	int	return code

5. RETVEN

Calling Convention :

CALL RETVEN (dbkey,attnam,vsnam,value,ierr)

Purpose :

to retrieve a value from an entity

Description :

The value item of an entity identified by a database key is retrieved and put into an integer array value. The item to retrieve is identified by an attribute name and a value set name.

Arguments :

NAME	USAGE	TYPE	DESCRIPTION
dbkey	input	int(2)	database key

attnam	input	char	attribute type name
vsnam	input	char	value set name
ierr	output	int	return code

6. RETVRE

Calling Convention :

CALL RETVRE (dbkey,attnam,vsnam,value,ierr)

Purpose :

to retrieve a value of an attribute of a relationship

Description :

The value item of a relationship identified by a database key is retrieved and put into an integer array value. The item to retrieve is identified by an attribute name and a value set name.

Arguments :

NAME	USAGE	TYPE	DESCRIPTION
dbkey	input	int(2)	database key
attnam	input	char	attribute type name
vsnam	input	char	value set name
ierr	output	int	return code

7. CREREL

Calling Convention :

CALL CREREL (relnam,bufpar,ptrpar,ierr)

Purpose :

to create a relationship

Description :

A relationship is created. Bufpar contains the primary keys and the attributes of the relationship.

Arguments :

NAME	USAGE	TYPE	DESCRIPTION
relnam	input	char	relationship type name
bufpar	input	char	buffer containing the values for the relationship
ptrbuf	input	int	pointer to the bufpar where the description begins
ierr	output	int	return code

8. CHREAT

Calling Convention :

```
CALL CHREAT (dbkey,attnam,vsnam,newval,ierr)
```

Purpose :

change the value of a relationship attribute

Description :

A relationship identified by a database key is modified in the following way : the old value of a value set is identified by its name and an attribute name is replaced by the new value.

Arguments :

NAME	USAGE	TYPE	DESCRIPTION
dbkey	input	int(2)	database key
attnam	input	char	attribute type name
vsnam	input	char	value set name
newval	input	char	new value
ierr	output	int	return code

9. DELREL

Calling Convention :

CALL DELREL (dbkey,ierr)

Purpose :

to delete a relationship

Description :

A relationship instance identified by a database key is removed from the database.

Arguments :

NAME	USAGE	TYPE	DESCRIPTION
dbkey	input	int(2)	database key
ierr	output	int	return code

10. RETREL

Calling Convention :

CALL RETREL (relnam ,pklist,dbklis,ierr)

Purpose :

retrieve a relationship instance

Description :

Given a relationship type name,and a list of primary keys which are related by this relationship (the name of the role precedes each primary key),this subroutine returns a list of database keys which correspond to the primary keys stated as inputs.

Arguments :

NAME	USAGE	TYPE	DESCRIPTION
relnam	input	char	relationship type name
pklist	input	char	primary key list
dbklis	output	int	database key list
ierr	output	int	return code

11. IDOP

Calling Convention :

IDOP (liobuf ,use,ierr)

Purpose :

open a database

Description :

The integer function IDOP is used to open a database. Liodbf designates the logical I/O unit to which the database is attached. If use is specified "0" then the database is only open for read operations.

Arguments :

NAME	USAGE	TYPE	DESCRIPTION
liobuf	input	int	logical I/O number
use	input	int	read/write flag
ierr	output	int	return code

12. DCLOSE

Calling Convention :

DCLOSE (ierr)

Purpose :

close the current database

Description :

the current database is closed

Arguments :

NAME	USAGE	TYPE	DESCRIPTION
ierr	output	int	return code

2. DBHLOW

These routines are the low level routines used by DBUSER. They do most of the actual work of the DBMS. In this part, we only describe the routines corresponding to the management of the entities. As the secondary key part is also managed with a technique used for the secondary keys of an entity they are similar to certain routines described below.

13. IDPAGE

CALLING CONVENTION :

```
I = IDPAGE( INDCUR,TYPE,OVDES,NOPAGE )
```

PURPOSE :

A number identifying a page is returned . This number is computed by combining the kind of file , the type of the page , the index of the description of an object type and the page number in a file .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
INDCUR	INPUT	INTEGER	Index of the description of an object type
TYPE	INPUT	INTEGER	Kind of file
OVDES	INPUT	INTEGER	Type of a page
NOPAGE	INPUT	INTEGER	Page number
IDPAGE	INPUT	INTEGER	Number identifying the page

14. CVCS

CALLING CONVENTION :

```
CALL CVCS( BUFNAM,PTRNAM,LENNAM,INTNUM,IERR )
```

PURPOSE :

Convert a character string to a integer

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION

BUFNAM	INPUT	INTEGER	Array where the character form of the name is
PTRNAM	INPUT	INTEGER	Pointer into BUFNAM where the name start
LENNAM	INPUT	INTEGER	Lenght of name
INTNUM	OUTPUT	INTEGER	number representing the name
IERR	OUTPUT	INTEGER	Error code

15. GETPAG

CALLING CONVENTION :

CALL GETPAG

PURPOSE :

Find a free page in BUFPAG

ARGUMENTS :

16. WHPAGE

CALLING CONVENTION :

CALL WHPAGE

PURPOSE :

Write the current page (if it has been modified)

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION

17. NEPAGE

CALLING CONVENTION :

CALL NEPAGE(PAGADD,IERR)

PURPOSE :

Get and initialize a new page

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
PAGADD	INPUT	INTEGER	Address of the new page
Ierr	INPUT	INTEGER	Error code

18. CVENBU

CALLING CONVENTION :

CALL CVENBU(BUFTCO,PTRBUF,LENSTG,IERR)

PURPOSE :

Convert the integer,real,character parameter of BUFTCO.This buffer contains all the parameters stated for an entity instance .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
BUFTCO	INPUT	CHAR	Buffer containing the parameters
PTRBUF	INPUT	INTEGER	Where to start
LENSTG	INPUT	INTEGER	Length of string
IERR	INPUT	INTEGER	Error code

19. CVPAEN

CALLING CONVENTION :

CALL CVPAEN(BUFTCO,BUFCON,PTRBUF,NBRATT,IERR)

PURPOSE :

Convert the character form of a part of an entity (either the primary key, the secondary key part and the attribute part) .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION

BUFTCO	INPUT	CHAR	Array where the char form of the description of an entity is.
BUFCON	OUTPUT	CHAR	Array where the convert form of the description of an entity is.
PTRBUF	I/O	INTEGER	Pointer into BUFTCO where the desciption of the part begins and ends.
NBRATT	INPUT	INTEGER	Number of attribute for the part .
IERR	OUTPUT	INTEGER	Error code

20. CHECKI

CALLING CONVENTION :

CALL CHECKI(NUM,IERR)

PURPOSE :

Check if an integer value belongs to those defined

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
NUM	INPUT	INTEGER	Number to check
IERR	OUTPUT	INTEGER	Error code

21. CHECKE

CALLING CONVENTION :

CALL CHECK(ENUM,IERR)

PURPOSE :

Check if an integer value belongs to those defined

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
ENUM	INPUT	REAL	Number to check
IERR	OUTPUT	INTEGER	Error code

22. CHECKC

CALLING CONVENTION :

CALL CHECKC(STRARR, STRSTA, IERR)

PURPOSE :

Check if a character string is right

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
STRARR	INPUT	CHAR	Character string to check
STRSTA	INPUT	INTEGER	Where to start
IERR	OUTPUT	INTEGER	Error code

23. CHEEPK

CALLING CONVENTION :

CALL CHEEPK(BUFSTR, ADDPKE, ADDSKE, EXIST)

PURPOSE :

Check the existence of the primary key in core

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
BUFSTR	INPUT	CHAR	Buffer containing the primary key
ADDPKE	I/O	INTEGER	Address of the primary key
ADDSKE	OUTPUT	INTEGER	Address of the secondary key (if it exists)
EXIST	OUTPUT	LOGICAL	Say if the primary key exists or not

24. INSKEY

CALLING CONVENTION :

CALL INSKEY(ADDSKE, IERR)

PURPOSE :

Insert the secondary key in core .The secondary key is contained in a common CURPAR which contains the current entity and

relationship.

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
ADDSKE	INPUT	INTEGER	Address of the secondary key
IERR	OUTPUT	INTEGER	Error code

25. INPKAD

CALLING CONVENTION :

CALL INPKAD(ADDSKE,ADDPKE)

PURPOSE :

Insert the primary key address related to a secondary key

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
ADDSKE	INPUT	INTEGER	Address of the secondary key
ADDPKE	INPUT	INTEGER	Address of the primary key

26. INSKAD

CALLING CONVENTION :

CALL INSKAD(ADDPKE,ADDSKE)

PURPOSE :

Insert the secondary key address related to a primary key .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
ADDPKE	INPUT	INTEGER	Address of the primary key
ADDSKE	INPUT	INTEGER	Address of the secondary key

27. UPDESC

CALLING CONVENTION :

CALL UPDESC(ARRAY,ADDARR)

PURPOSE :

Update the descriptor of a page when a new record has been inserted.

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
ARRAY	INPUT	CHAR	Character array containing the record for the computation of the descriptor
ADDARR	INPUT	INTEGER	Address of the record

28. CODESC

CALLING CONVENTION :

CALL CODESC(ARRAY , DESCR)

PURPOSE :

Compute the descriptor of an array

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
ARRAY	INPUT	CHAR	Array containing the fiels of a record
DESCR	OUTPUT	INTEGER	Descriptor

29. COADES

CALLING CONVENTION :

CALL COADES(ADDSKE,ADDESC)

PURPOSE :

Compute the address of the descriptor

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION

ADDSKE	INPUT	INTEGER	Address of the secondary key
ADDESC	OUTPUT	INTEGER	Address of the descriptor

30. CASKEY

CALLING CONVENTION :

CALL CADSKEY(ADDARR)

PURPOSE :

Compute the address of the current secondary key

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
ADDARR	OUTPUT	INTEGER	Addres of the secondary key in core

31. SPSKEY

CALLING CONVENTION :

CALL SPSKEY

PURPOSE :

Make the split of a page of the current secondary key file

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION

none

32. SPLREC

CALLING CONVENTION :

CALL SPLREC(BUFPAR , ADDBUF)

PURPOSE :

Transfert a record on a splitted page

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
BUFPAR	INPUT	CHAR	Record to transfert
ADDBUF	OUTPUT	INTEGER	Address where to insert

33. UPDEPA

CALLING CONVENTION :

CALL UPDEPA(ADDPAG)

PURPOSE :

Update the descriptor of a page which has split

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
ADDPAG	INPUT	INTEGER	Address of the page

34. INPKEY

CALLING CONVENTION :

CALL INPKEY(PKADD , IERR)

PURPOSE :

Insert a new primary key in the database

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
PKADD	INPUT	INTEGER	Address of the primary key
IERR	OUTPUT	INTEGER	Error code

35. CAPKEY

CALLING CONVENTION :

CALL CAPKEY(ADDPKE)

PURPOSE :

Compute the address of the current primary key

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
ADDPKE	OUTPUT	INTEGER	Address of the primary key

36. SPPKEY

CALLING CONVENTION :

CALL SPPKEY

PURPOSE :

Split the current primary key file

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION

none

37. RANGEN

CALLING CONVENTION :

CALL RANGEN(INTNUM)

PURPOSE :

Generate a row of random number , this row is determined by INTNUM .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
INTNUM	INPUT	INTEGER	Number which determined the row to be generated .

38. DBHTFI

CALLING CONVENTION :

CALL DBHTFI(RETCOD)

PURPOSE :

Extract the data base tables from the data base

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
RETCOD	OUTPUT	INTEGER	Return code

39. DBHERR

CALLING CONVENTION :

CALL DBHERR(RTNNAM, IERR)

PURPOSE :

Handle database errors

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
RTNAM	INPUT	CHAR	Routine Name
IERR	INPUT	INTEGER	Error number

40. INIPAR

CALLING CONVENTION :

CALL INIPAR(PTRTAB, TYPE)

PURPOSE :

Initialize the current parameter

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
PTRTAB	INPUT	INTEGER	Pointer into the type description table
TYPE	INPUT	INTEGER	Integer to say if the current parameter is an entity (1) or a relationship (2)

3. DBHTAB library

DBHTAB is a collection of Fortran integer functions which return control block fields, and Fortran subroutines which update the control block fields. They are heavily used by DBHLOW and DBHUSE in referencing the database tables

41. IETYNA

CALLING convention :

```
I = IETYNA (BUFNAM , PTRNAM  , LENNAM )
```

PURPOSE :

Integer function which returns the value of the pointer into the TYDTAB description table .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
BUFNAM	INPUT	CHAR	buffer containing the entity type name
PTRNAM	INPUT	INTEGER	pointer where the name begins in BUFNAM
LENNAM	INPUT	INTEGER	length of the entity type name
IETYNA	OUTPUT	INTEGER	pointer to the description of the entity into TYDTAB

42. IRTYNA

CALLING convention :

```
I = IRTYNA (BUFNAM , PTRNAM  , LENNAM )
```

PURPOSE :

Integer function which returns the value of the pointer into the TYDTAB description table .This pointer points to the the beginning of the description
of a relationship .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
BUFNAM	INPUT	CHAR	buffer containing the entity type name

PTRNAM	INPUT	INTEGER	pointer where the name begins in BUFNAM
LENNAM	INPUT	INTEGER	length of the entity type name
IRTYNA	OUTPUT	INTEGER	pointer to the description of the relationship into TYDTAB

IPEBDE

CALLING convention :

I = IPEBDE (IPEBPT)

PURPOSE :

Integer function which returns the value of the pointer into the TYDTAB description table from the entity type name identified by the IPB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IPBPTR	INPUT	INTEGER	IPB pointer to a entity type control block
IPEBDE	OUTPUT	INTEGER	entity description pointer

IPRBDE

CALLING convention :

I = IPRBDE (IPRBPT)

PURPOSE :

Integer function which returns the value of the pointer into the TYDTAB description table from the relationship type name description identified by the IPB pointer

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IPBPTR	INPUT	INTEGER	IPB pointer to a relationship type control block
IPEBDE	OUTPUT	INTEGER	entity description pointer

IPEBOV

CALLING convention :

I = IPEBOV (IPRBPT)

PURPOSE :

Integer function which returns the value of the pointer into the overflow space of the ETNTAB .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IPBPTR	INPUT	INTEGER	IPB pointer to a entity type control block
IPEBOV	OUTPUT	INTEGER	overflow pointer

IPRBOV

CALLING convention :

I = IPRBOV (IPRBPT)

PURPOSE :

Integer function which returns the value of the pointer into the overflow space of the RTNTAB .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IPBPTR	INPUT	INTEGER	IPB pointer to a relationship type control block
IPRBOV	OUTPUT	INTEGER	overflow pointer

43. IPKBLD

CALLING convention :

I = IPKBLD (IPKBPT)

PURPOSE :

Integer function which returns the value of the primary key description length from the IPKB identified by the IPKB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION

IPKBPT	INPUT	INTEGER	IPKB pointer to a primary key type control block
IPKBLD	OUTPUT	INTEGER	description length of the primary key type

44. IPKBNA

CALLING convention :

I = IPKBNA (IPKBPT)

PURPOSE :

Integer function which returns the value of the number of attribute from the IPKB identified by the IPKB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IPKBPT	INPUT	INTEGER	IPKB pointer to a primary key type control block
IPKBNA	OUTPUT	INTEGER	number of attribute of a primary key type

45. IPKBLP

CALLING convention :

I = IPKBLP (IPKBPT)

PURPOSE :

Integer function which returns the value of the primary key length from the IPKB identified by the IPKB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IPKBPT	INPUT	INTEGER	IPKB pointer to a primary key type control block
IPKBLP	OUTPUT	INTEGER	length of the primary key on a page

46. ISKBLD

CALLING convention :

```
I = ISKBLD (ISKBPT)
```

PURPOSE :

Integer function which returns the value of the secondary key description length from the ISKB identified by the ISKB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
ISKBPT	INPUT	INTEGER	ISKB pointer to a secondary key type control block
ISKBLD	OUTPUT	INTEGER	description length of a secondary key type

47. ISKBNA

CALLING convention :

```
I = ISKBNA (ISKBPT)
```

PURPOSE :

Integer function which returns the value of the number of attribute from the ISKB identified by the ISKB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
ISKBPT	INPUT	INTEGER	ISKB pointer to a secondary key type control block
ISKBNA	OUTPUT	INTEGER	number of attribute of secondary key type

48. ISKBLP

CALLING convention :

```
I = ISKBLP (ISKBPT)
```

PURPOSE :

Integer function which returns the value of the secondary key length from the ISKB identified by the ISKB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
ISKBPT	INPUT	INTEGER	ISKB pointer to a secondary key type control block
ISKBLP	OUTPUT	INTEGER	length of the secondary key on a page

49. IAPBLD

CALLING convention :

I = IAPBLD (IAPBPT)

PURPOSE :

Integer function which returns the value of the attribute part description length from the IAPB identified by the IAPB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IAPBPT	INPUT	INTEGER	IAPB pointer to an attribute part type control block
IAPBLD	OUTPUT	INTEGER	description length of the attribute part type

50. IAPBNA

CALLING convention :

I = IAPBNA (IAPBPT)

PURPOSE :

Integer function which returns the value of the number of attribute from the IAPB identified by the IAPB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IAPBPT	INPUT	INTEGER	IAPB pointer to a attribute part type control block

IAPBNA	OUTPUT	INTEGER	number of attribute of a attribute part type

51. IAPBLP

CALLING convention :

I = IAPBLP (IAPBPT)

PURPOSE :

Integer function which returns the value of the attribute part length from the IAPB identified by the IAPB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IAPBPT	INPUT	INTEGER	IAPB pointer to a attribute part type control block
IAPBLP	OUTPUT	INTEGER	length of the attribute part on a page

52. 53. IAPBLD

CALLING convention :

I = IAPBLD (IAPBPT)

PURPOSE :

Integer function which returns the value of the attribute part description length from the IAPB identified by the IAPB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IAPBPT	INPUT	INTEGER	IAPB pointer to an attribute part type control block
IAPBLD	OUTPUT	INTEGER	description length of the attribute part type

54. IAPBNA

CALLING convention :

I = IAPBNA (IAPBPT)

PURPOSE :

Integer function which returns the value of the number of attribute from the IAPB identified by the IAPB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IAPBPT	INPUT	INTEGER	IAPB pointer to a attribute part type control block
IAPBNA	OUTPUT	INTEGER	number of attribute of a attribute part type

55. IADBDL

CALLING convention :

I = IADBDL (IADBPT)

PURPOSE :

Integer function which returns the value of the description length from the IADB identified by the IADB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IADBPT	INPUT	INTEGER	IADB pointer to a attribute type control block
IADBDL	OUTPUT	INTEGER	length of the attribute on a page

56. IADBNL

CALLING convention :

I = IADBNL (IADBPT)

PURPOSE :

Integer function which returns the value of the attribute name length from the IADB identified by the IADB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION

IADBPT	INPUT	INTEGER	IADB pointer to a attribute type control block
IADBNL	OUTPUT	INTEGER	attribute name length

57. IADBNP

CALLING convention :

I = IADBNP (IADBPT)

PURPOSE :

Integer function which returns the value of the pointer into NAMES from the IADB identified by the IADB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IADBPT	INPUT	INTEGER	IADB pointer to a attribute type control block
IADBNP	OUTPUT	INTEGER	pointer name

58. IADBVS

CALLING convention :

I = IADBVS (IADBPT)

PURPOSE :

Integer function which returns the number of value set from the IADB identified by the IADB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IADBPT	IVSUT	INTEGER	IADB pointer to a attribute type control block
IADBVS	OUTPUT	INTEGER	number of value set

59. IVSBTY

CALLING convention :

I = IVSBTY (IVSBPT)

PURPOSE :

Integer function which returns the type of value set from the IVSB identified by the IVSB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IVSBPT	INPUT	INTEGER	IVSB pointer to a value set control block
IVSBTY	OUTPUT	INTEGER	value set type

60. IVSBNP

CALLING convention :

I = IVSBNP (IVSBPT)

PURPOSE :

Integer function which returns the name pointer of value set from the IVSB identified by the IVSB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IVSBPT	INPUT	INTEGER	IVSB pointer to a value set control block
IVSBNP	OUTPUT	INTEGER	name pointer

61. IVSBNL

CALLING convention :

I = IVSBNL (IVSBPT)

PURPOSE :

Integer function which returns the name length of a value set from the IVSB identified by the IVSB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IVSBPT	INPUT	INTEGER	IVSB pointer to a value set control block
IVSBNL	OUTPUT	INTEGER	value set name length

62. IVSBMD

CALLING convention :

I = IVSBMD (IVSBPT)

PURPOSE :

Integer function which returns the modifier value of value set from the IVSB identified by the IVSB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IVSBPT	INPUT	INTEGER	IVSB pointer to a value set control block
IVSBMD	OUTPUT	INTEGER	value set modifier value

63. IVSBNV

CALLING convention :

I = IVSBNV (IVSBPT)

PURPOSE :

Integer function which returns the number of value allowed from the IVSB identified by the IVSB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IVSBPT	INPUT	INTEGER	IVSB pointer to a value set control block
IVSBNV	OUTPUT	INTEGER	number of value allowed

64. IVSBMI

CALLING convention :

I = IVSBMI (IVSBPT)

PURPOSE :

Integer function which returns a minimum value (if integer type) from the IVSB identified by the IVSB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION

IVSBPT	INPUT	INTEGER	IVSB pointer to a value set control block
IVSBMI	OUTPUT	INTEGER	minimum value for a value set

65. IVSBMA

CALLING convention :

I = IVSBMA (IVSBPT)

PURPOSE :

Integer function which returns the maximum value from the IVSB identified by the IVSB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IVSBPT	INPUT	INTEGER	IVSB pointer to a value set control block
IVSBMA	OUTPUT	INTEGER	maximum value for a value set

66. IVSBIL

CALLING convention :

I = IVSBIL (IVSBPT)

PURPOSE :

Integer function which returns the minimum character length (if character type)from the IVSB identified by the IVSB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IVSBPT	INPUT	INTEGER	IVSB pointer to a value set control block
IVSBIL	OUTPUT	INTEGER	minimum character length

67. IVSBAL

CALLING convention :

I = IVSBAL (IVSBPT)

PURPOSE :

Integer function which returns the maximum character lenght from the IVSB identified by the IVSB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IVSBPT	INPUT	INTEGER	IVSB pointer to a value set control block
IVSBAL	OUTPUT	INTEGER	maximum character length

68. IVSBAP

CALLING convention :

I = IVSBAP (IVSBPT)

PURPOSE :

Integer function which returns the maximum character string pointer from the IVSB identified by the IVSB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IVSBPT	INPUT	INTEGER	IVSB pointer to a value set control block .
IVSBAP	OUTPUT	INTEGER	character string pointer

69. IVSBDS

CALLING convention :

I = IVSBDS (IVSBPT)

PURPOSE :

Integer function which returns the displacement of value set in the data area from the IVSB identified by the IVSB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IVSBPT	INPUT	INTEGER	IVSB pointer to a value set control block
IVSBDS	OUTPUT	INTEGER	displacement of the value set

70. IIRBAV

CALLING convention :

```
I = IIRBAV (IIRBPT)
```

PURPOSE :

Integer function which returns an allowable (integer or real type) value of a value set from an IIRB identified by the IIRB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IIRBPT	INPUT	INTEGER	IIRB pointer to an allowable value control block
IIRBAV	OUTPUT	INTEGER	an allowable value

71. ICHBNL

CALLING convention :

```
I = ICHBNL (ICHBPT)
```

PURPOSE :

Integer function which returns an allowable value lenght (character type) of a value set from an ICHB identified by the ICHB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
ICHBPT	INPUT	INTEGER	ICHB pointer to an allowable value (character) control block
ICHBNL	OUTPUT	INTEGER	a allowable character length

72. ICHBNP

CALLING convention :

```
I = ICHBNP (ICHBPT)
```

PURPOSE :

Integer function which returns an allowable value pointer (chharacter type) of a value set from an ICHB identified by the ICHB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
ICHBPT	INPUT	INTEGER	ICHB pointer to an allowable value (character) control block
ICHBNP	OUTPUT	INTEGER	a allowable character pointer

73. IREBLD

CALLING convention :

I = IREBLD (IREBPT)

PURPOSE :

Integer function which returns the description lenght from an IREB identified by the IREB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IREBPT	INPUT	INTEGER	IREB pointer to a relationship control block
IREBLD	OUTPUT	INTEGER	the relationship description length

74. IREBLR

CALLING convention :

I = IREBLR (IREBPT)

PURPOSE :

Integer function which returns the lenght from an IREB identified by the IREB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IREBPT	INPUT	INTEGER	IREB pointer to a relationship control block
IREBLR	OUTPUT	INTEGER	the relationship length

75. IREBNR

CALLING convention :

I = IREBNR (IREBPT)

PURPOSE :

Integer function which returns the number of entity related from an IREB identified by the IREB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IREBPT	INPUT	INTEGER	IREB pointer to a relationship control block
IREBNR	OUTPUT	INTEGER	number of entity related

76. IREBID

CALLING convention :

I = IREBID (IREBPT)

PURPOSE :

Integer function which returns the sequence number of the description from an IREB identified by the IREB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IREBPT	INPUT	INTEGER	IREB pointer to a relationship control block
IREBID	OUTPUT	INTEGER	sequence number

77. IERBNP

CALLING convention :

I = IERBNP (IERBPT)

PURPOSE :

Integer function which returns the role name pointer from an IERB identified by the IERB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION

IERBPT	INPUT	INTEGER	IERB pointer to an entity related control block
IERBNP	OUTPUT	INTEGER	role name pointer

78. IERBNL

CALLING convention :

```
I = IERBNL (IERBPT)
```

PURPOSE :

Integer function which returns the role name length from an IERB identified by the IERB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IERBPT	INPUT	INTEGER	IERB pointer to an entity related control block
IERBNL	OUTPUT	INTEGER	role name length

79. IERBPD

CALLING convention :

```
I = IERBPD (IERBPT)
```

PURPOSE :

Integer function which returns the entity description pointer from an IERB identified by the IERB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IERBPT	INPUT	INTEGER	IERB pointer to an entity related control block
IERBPD	OUTPUT	INTEGER	entity description pointer

80. IERBMI

CALLING convention :

```
I = IERBMI (IERBPT)
```

PURPOSE :

Integer function which returns the minimum connectivity from an IERB identified by the IERB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IERBPT	INPUT	INTEGER	IERB pointer to an entity related control block
IERBMI	OUTPUT	INTEGER	connectivity minimum

81. IERBMA

CALLING convention :

```
I = IERBMA (IERBPT)
```

PURPOSE :

Integer function which returns the maximum connectivity from an IERB identified by the IERB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IERBPT	INPUT	INTEGER	IERB pointer to an entity related control block
IERBMA	OUTPUT	INTEGER	maximum connectivity

82. IERBPC

CALLING convention :

```
I = IERBPC (IERBPT)
```

PURPOSE :

Integer function which returns the probability that the IERB identified by the IERB pointer is implied in a query .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IERBPT	INPUT	INTEGER	IERB pointer to an entity related control block
IERBPC	OUTPUT	INTEGER	probability that the entity is implied in a query

83. IHPBDE

CALLING convention :

I = IHPBDE

PURPOSE :

Integer function which returns the depth of a primary key file identified by the current index in the table HPKTAB .

ARGUMENTS :

none

84. IHPBSP

CALLING convention :

I = IHPBSP

PURPOSE :

Integer function which returns the value of the split pointer of a primary key file identified by the current index in the table HPKTAB .

ARGUMENTS :

none

85. IHPBIB

CALLING convention :

I = IHPBIB

PURPOSE :

Integer function which returns the initial number of bucket of a primary key file identified by the current index in the table HPKTAB .

ARGUMENTS :

none

86. IHPBNG

CALLING convention :

I = IHPBNG

PURPOSE :

Integer function which returns the number of groups of a primary key file identified by the current index in the table HPKTAB .

ARGUMENTS :

none

87. IHPBNB

CALLING convention :

I = IHPBNB

PURPOSE :

Integer function which returns the current number of buckets not yet expanded for a primary key file identified by the current index in the table HPKTAB .

ARGUMENTS :

none

88. IHPBNB

CALLING convention :

I = IHPBNB

PURPOSE :

Integer function which returns the current number of buckets not yet expanded for a primary key file identified by the current index in the table HPKTAB .

ARGUMENTS :

none

89. IHSBDE

CALLING convention :

I = IHSBDE

PURPOSE :

Integer function which returns the current depth of a secondary Key file identified by the current index in the table HATTAB .

ARGUMENTS :

none

90. IHSBNP

CALLING convention :

I = IHSBNP

PURPOSE :

Integer function which returns the current number of pages of a secondary key file identified by the current index in the table HATTAB .

ARGUMENTS :

none

91. IHSBSP

CALLING convention :

I = IHSBSP

PURPOSE :

Integer function which returns the current split pointer of a primary key file identified by the current index in the table HATTAB .

ARGUMENTS :

none

92. IHSBNI

CALLING convention :

I = IHSBNI

PURPOSE :

Integer function which returns the initial number of pages of a secondary key file identified by the current index in the table HATTAB .

ARGUMENTS :

none

93. IOHBNF

CALLING convention :

I = IOHBNF

PURPOSE :

Integer function which returns the number of fields of a secondary key or relationship file identified by the current index in the table OPATAB .

ARGUMENTS :

none

94. IOHBMD

CALLING convention :

I = IOHBMD

PURPOSE :

Integer function which returns the maximum of a secondary key or relationship file identified by the current index in the table OPATAB .

ARGUMENTS :

none

95. IOPBDI

CALLING convention :

I = IOPBDI (I)

PURPOSE :

Integer function which returns the number of bits allocated to a field (identified by I) of a secondary key or relationship file identified by the current index in the table OPATAB .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
I	INPUT	INTEGER	index of field
IOPBDI	OUTPUT	INTEGER	optimal number of bits

96. IOPBWI

CALLING convention :

I = IOPBWI (I)

PURPOSE :

Integer function which returns the number of bits to allocated to a descriptor field (identified by I) of a secondary key or relationship file identified by the current index in the table OPATAB .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
I	INPUT	INTEGER	index of field

IOPBWI	OUTPUT	INTEGER	optimal number of bits for a descriptor field

97. IOABCV

CALLING convention :

I = IOABCV (I)

PURPOSE :

Integer function which returns the choice vector pointer (identified by I) of a secondary key or relationship file identified by the current index in the table OPATAB .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
I	INPUT	INTEGER	index of choice vector
IOABCV	OUTPUT	INTEGER	choice vector pointer

98. SETYNA

CALLING convention :

I = SETYNA (BUFNAM , PTRNAM , LENNAM)

PURPOSE :

put the entity type name contain in BUFNAM into the ETNTAB .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
BUFNAM	INPUT	CHAR	buffer containing the entity type name
PTRNAM	INPUT	INTEGER	pointer where the name begins in BUFNAM
LENNAM	INPUT	INTEGER	length of the entity type name
IERR	OUTPUT	INTEGER	Error code

99. SRTYNA

CALLING convention :

I = SRTYNA (BUFNAM , PTRNAM , LENNAM , IERR)

PURPOSE :

put the relationship type name contained in BUFNAM into the RETYNA .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
BUFNAM	INPUT	CHAR	buffer containing the entity type name
PTRNAM	INPUT	INTEGER	pointer where the name begins in BUFNAM
LENNAM	INPUT	INTEGER	length of the entity type name
IERR	OUTPUT	INTEGER	Error code

SPEBDE

CALLING convention :

SPEBDE (SPEBPT , BDE)

PURPOSE :

SET the value of the pointer into the TYDTAB description table from the entity type name identified by the IPB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IPBPTR	INPUT	INTEGER	IPB pointer to a entity type control block
BDE	INPUT	INTEGER	pointer to the TYDTAB

IPRBDE

CALLING convention :

SPRBDE (SPRBPT , SPR)

PURPOSE :

Set the value of the pointer into the TYDTAB description table from the relationship type name description identified by the IPB pointer

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IPBPTR	INPUT	INTEGER	IPB pointer to a relationship type control block
BDE	OUTPUT	INTEGER	entity description pointer

SPEBOV

CALLING convention :

SPEBOV (SPRBPT , BOV)

PURPOSE :

Set the value of the pointer into the overflow space of the ETNTAB table .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IPBPTR	INPUT	INTEGER	IPB pointer to a entity type control block
BOV	INPUT	INTEGER	overflow pointer

SPRBOV

CALLING convention :

CALL SPRBOV (SPRBPT)

PURPOSE :

Set the value of the pointer into the overflow space of the RTNTAB .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IPBPTR	INPUT	INTEGER	IPB pointer to a relationship type control block
BOV	INPUT	INTEGER	overflow pointer

100. SPKBLD

CALLING convention :

CALL SPKBLD (IPKBPT , BLD)

PURPOSE :

Set the value of the primary key description length from the IPKB identified by the IPKB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IPKBPT	INPUT	INTEGER	IPKB pointer to a primary key type control block
BLD	INPUT	INTEGER	description length of the primary key type

101. SPKBNA

CALLING convention :

CALL SPKBNA (IPKBPT , BNA)

PURPOSE :

Set the value of the number of attribute from the IPKB identified by the IPKB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IPKBPT	INPUT	INTEGER	IPKB pointer to a primary key type control block
BNA	INPUT	INTEGER	number of attribute of a primary key type

102. SPKBLP

CALLING convention :

CALL SPKBLP (IPKBPT , BLP)

PURPOSE :

Set the value of the primary key length from the IPKB identified by the IPKB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IPKBPT	INPUT	INTEGER	IPKB pointer to a primary key type control block
BLP	INPUT	INTEGER	length of the primary key on a page

103. SSKBLD

CALLING convention :

```
CALL SSKBLD (ISKBPT , BLD)
```

PURPOSE :

Set the value of the secondary key description length from the ISKB identified by the ISKB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
ISKBPT	INPUT	INTEGER	ISKB pointer to a secondary key type control block
BLD	INPUT	INTEGER	description length of a secondary key type

104. SSKBNA

CALLING convention :

```
CALL SSKBNA (ISKBPT)
```

PURPOSE :

Set the value of the number of attribute from the ISKB identified by the ISKB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
ISKBPT	INPUT	INTEGER	ISKB pointer to a secondary key type control block
BNA	INPUT	INTEGER	number of attribute of secondary key type

105. CALL SSKBLP

CALLING convention :

SSKBLP (ISKBPT , BLP)

PURPOSE :

Set the value of the secondary key length from the ISKB identified by the ISKB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
ISKBPT	INPUT	INTEGER	ISKB pointer to a secondary key type control block
BLP	INPUT	INTEGER	length of the secondary key on a page

106. SAPBLD

CALLING convention :

CALL SAPBLD (IAPBPT , BLD)

PURPOSE :

Set the value of the attribute part description length from the IAPB identified by the IAPB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IAPBPT	INPUT	INTEGER	IAPB pointer to an attribute part type control block
BLD	INPUT	INTEGER	description length of the attribute part type

107. SAPBNA

CALLING convention :

CALL SAPBNA (IAPBPT , BNA)

PURPOSE :

Set the value of the number of attribute from the IAPB identified by the IAPB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IAPBPT	INPUT	INTEGER	IAPB pointer to a attribute part type control block
BNA	INPUT	INTEGER	number of attribute of a attribute part type

108. SAPBLP

CALLING convention :

CALL SAPBLP (IAPBPT , BLP)

PURPOSE :

Set the value of the attribute part length from the IAPB identified by the IAPB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IAPBPT	INPUT	INTEGER	IAPB pointer to a attribute part type control block
SAPBLP	INPUT	INTEGER	length of the attribute part on a page

109. 110. SAPBLD

CALLING convention :

CALL SAPBLD (IAPBPT , BLD)

PURPOSE :

Set the value of the attribute part description length from the IAPB identified by the IAPB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IAPBPT	INPUT	INTEGER	IAPB pointer to an attribute part type control block
BLD	INPUT	INTEGER	description length of the attribute part type

111. SAPBNA

CALLING convention :

CALL SAPBNA (IAPBPT , BNA)

PURPOSE :

Set the value of the number of attribute from the IAPB identified by the IAPB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IAPBPT	INPUT	INTEGER	IAPB pointer to a attribute part type control block
BNA	INPUT	INTEGER	number of attribute of a attribute part type

112. SADBDL

CALLING convention :

CALL SADBDL (IADBPT)

PURPOSE :

Set the value of the description length from the IADB identified by the IADB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IADBPT	INPUT	INTEGER	IADB pointer to a attribute type control block
BDL	INPUT	INTEGER	length of the attribute on a page

113. SADBNL

CALLING convention :

CALL SADBNL (IADBPT , BNL)

PURPOSE :

Set the value of the attribute name length from the IADB identified by the IADB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IADBPT	INPUT	INTEGER	IADB pointer to a attribute type control block
BNL	INPUT	INTEGER	attribute name length

114. SADBNP

CALLING convention :

CALL SADBNP (IADBPT , BNP)

PURPOSE :

Set the value of the pointer into NAMES from the IADB identified by the IADB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IADBPT	INPUT	INTEGER	IADB pointer to a attribute type control block
BNP	INPUT	INTEGER	pointer name

115. SADBVS

CALLING convention :

CALL SADBVS (IADBPT , BVS)

PURPOSE :

Set the number of value set from the IADB identified by the IADB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IADBPT	IVSUT	INTEGER	IADB pointer to a attribute type control block
BVS	INPUT	INTEGER	number of value set

116. SVSBTY

CALLING convention :

CALL SVSBTY (IVSBPT , BTY)

PURPOSE :

Set the type of value set from the IVSB identified by the IVSB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IVSBPT	INPUT	INTEGER	IVSB pointer to a value set control block
BTY	INPUT	INTEGER	value set type

117. SVSBNP

CALLING convention :

CALL SVSBNP (IVSBPT , BNP)

PURPOSE :

Set the name pointer of value set from the IVSB identified by the IVSB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IVSBPT	INPUT	INTEGER	IVSB pointer to a value set control block
BNP	INPUT	INTEGER	name pointer

118. SVSBNL

CALLING convention :

CALL SVSBNL (IVSBPT , BNL)

PURPOSE :

Set the name length of a value set from the IVSB identified by the IVSB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IVSBPT	INPUT	INTEGER	IVSB pointer to a value set control block

BNL	INPUT	INTEGER	value set name length

119. SVSBMD

CALLING convention :

CALL SVSBMD (IVSBPT , BMD)

PURPOSE :

Set the modifier value of value set from the IVSB identified by the IVSB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IVSBPT	INPUT	INTEGER	IVSB pointer to a value set control block
BMD	INPUT	INTEGER	value set modifier value

120. SVSBNV

CALLING convention :

CALL SVSBNV (IVSBPT , BNV)

PURPOSE :

Set the number of value allowed from the IVSB identified by the IVSB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IVSBPT	INPUT	INTEGER	IVSB pointer to a value set control block
SVSBNV	INPUT	INTEGER	number of value allowed

121. SVSBMI

CALLING convention :

CALL SVSBMI (IVSBPT , BMI)

PURPOSE :

Set a minimum value (if integer type) from the IVSB identified by the IVSB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IVSBPT	INPUT	INTEGER	IVSB pointer to a value set control block
BMI	INPUT	INTEGER	minimum value for a value set

122. SVSBMA

CALLING convention :

CALL SVSBMA (IVSBPT , BMA)

PURPOSE :

Set the maximum value from the IVSB identified by the IVSB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IVSBPT	INPUT	INTEGER	IVSB pointer to a value set control block
BMA	INPUT	INTEGER	maximum value for a value set

123. SVSBIL

CALLING convention :

CALL SVSBIL (IVSBPT , BIL)

PURPOSE :

Set the minimum character length (if character type)from the IVSB identified by the IVSB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IVSBPT	INPUT	INTEGER	IVSB pointer to a value set control block
BIL	INPUT	INTEGER	minimum character length

124. SVSBAL

CALLING convention :

CALL SVSBAL (IVSBPT , BAL)

PURPOSE :

Set the maximum character lenght from the IVSB identified by the IVSB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IVSBPT	INPUT	INTEGER	IVSB pointer to a value set control block
BAL	INPUT	INTEGER	maximum character length

125. SVSBAP

CALLING convention :

CALL SVSBAP (IVSBPT , BAP)

PURPOSE :

Set the maximum character string pointer from the IVSB identified by the IVSB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IVSBPT	INPUT	INTEGER	IVSB pointer to a value set control block
BAP	INPUT	INTEGER	character string pointer

126. SVSBDS

CALLING convention :

CALL SVSBDS (IVSBPT , BDS)

PURPOSE :

Set the displacement of value set in the data area from the IVSB identified by the IVSB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION

IVSBPT	INPUT	INTEGER	IVSB pointer to a value set control block
BDS	INPUT	INTEGER	displacement of the value set

127. SIRBAV

CALLING convention :

CALL SIRBAV (IIRBPT , BAV)

PURPOSE :

Set an allowable (integer or real type) value of a value set from an IIRB identified by the IIRB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IIRBPT	INPUT	INTEGER	IIRB pointer to an allowable value control block
BAV	INPUT	INTEGER	an allowable value

128. SCHBNL

CALLING convention :

CALL SCHBNL (ICHBPT , BNL)

PURPOSE :

Set an allowable value lenght (character type) of a value set from an ICHB identified by the ICHB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
ICHBPT	INPUT	INTEGER	ICHB pointer to an allowable value (character) control block
BNL	INPUT	INTEGER	a allowable character length

129. SCHBNP

CALLING convention :

CALL SCHBNP (ICHBPT , BNP)

PURPOSE :

Setan allowable value pointer (chharacter type) of a value set from an ICHB identified by the ICHB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
ICHBPT	INPUT	INTEGER	ICHB pointer to an allowable value (character) control block
BNP	INPUT	INTEGER	a allowable character pointer

130. SREBLD

CALLING convention :

CALL SREBLD (IREBPT , BLD)

PURPOSE :

Set the description lenght from an IREB identified by the IREB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IREBPT	INPUT	INTEGER	IREB pointer to a relationship control block
BLD	INPUT	INTEGER	the relationship description length

131. SREBLR

CALLING convention :

CALL SREBLR (IREBPT , BLR)

PURPOSE :

Set the lenght from an IREB identified by the IREB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IREBPT	INPUT	INTEGER	IREB pointer to a relationship control block
BLR	INPUT	INTEGER	the relationship length

132. SREBNR

CALLING convention :

CALL SREBNR (IREBPT , BNR)

PURPOSE :

Set the number of entity related from an IREB identified by the IREB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IREBPT	INPUT	INTEGER	IREB pointer to a relationship control block
BNR	INPUT	INTEGER	number of entity related

133. SREBID

CALLING convention :

CALL SREBID (IREBPT , BID)

PURPOSE :

Set the sequence number of the description from an IREB identified by the IREB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IREBPT	INPUT	INTEGER	IREB pointer to a relationship control block
BID	INPUT	INTEGER	sequence number

134. SERBNP

CALLING convention :

CALL SERBNP (IERBPT , BNP)

PURPOSE :

Set the role name pointer from an IERB identified by the IERB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION

IERBPT	INPUT	INTEGER	IERB pointer to an entity related control block
BNP	INPUT	INTEGER	role name pointer

135. SERBNL

CALLING convention :

CALL SERBNL (IERBPT , BNL)

PURPOSE :

Set the role name length from an IERB identified by the IERB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IERBPT	INPUT	INTEGER	IERB pointer to an entity related control block
BNL	INPUT	INTEGER	role name length

136. SERBPD

CALLING convention :

CALL SERBPD (IERBPT , BPD)

PURPOSE :

Set the entity description pointer from an IERB identified by the IERB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IERBPT	INPUT	INTEGER	IERB pointer to an entity related control block
BPD	INPUT	INTEGER	entity description pointer

137. SERBMI

CALLING convention :

CALL SERBMI (IERBPT , BMI)

PURPOSE :

Set the minimum connectivity from an IERB identified by the IERB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IERBPT	INPUT	INTEGER	IERB pointer to an entity related control block
BMI	INPUT	INTEGER	connectivity minimum

138. SERBMA

CALLING convention :

CALL SERBMA (IERBPT , BMA)

PURPOSE :

Set the maximum connectivity from an IERB identified by the IERB pointer .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IERBPT	INPUT	INTEGER	IERB pointer to an entity related control block
BMA	INPUT	INTEGER	maximum connectivity

139. SERBPC

CALLING convention :

CALL SERBPC (IERBPT , BPC)

PURPOSE :

Set the probability that the IERB identified by the IERB pointer is implied in a query .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
IERBPT	INPUT	INTEGER	IERB pointer to an entity related control block
BPC	INPUT	INTEGER	probability that the entity is implied in a query

140. SHPBDE

CALLING convention :

CALL SHPBDE (BDE)

PURPOSE :

Set the depth of a primary key file identified by the current index in the table HPKTAB .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
BDE	INPUT	INTEGER	number of depth

141. SHPBSP

CALLING convention :

CALL SHPBSP (BSP)

PURPOSE :

Set the value of the split pointer of a primary key file identified by the current index in the table HPKTAB .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
BSP	INPUT	INTEGER	split pointer

142. SHPBIB

CALLING convention :

CALL SHPBIB (BID)

PURPOSE :

Set the initial number of bucket of a primary key file identified by the current index in the table HPKTAB .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
BID	INPUT	INTEGER	initial number of buckets

143. NHPBNG

CALLING convention :

CALL SHPBNG(BNG)

PURPOSE :

Set the number of groups of a primary key file identified by the current index in the table HPKTAB .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
BNG	INPUT	INTEGER	number of groups

144. SHPBNB

CALLING convention :

CALL SHPBNB(BNB)

PURPOSE :

Set the current number of buckets not yet expanded for a primary key file identified by the current index in the table HPKTAB .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
BNB	INPUT	INTEGER	Number of buckets not yet expanded

145. SHSBDE

CALLING convention :

CALL SHSBDE(BDE)

PURPOSE :

Set the current depth of a secondary Key file identified by the current index in the table HATTAB .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
BDE	INPUT	INTEGER	depth of a secondary file

146. SHSBNP

CALLING convention :

CALL SHSBNP(BNP)

PURPOSE :

Set the current number of pages of a secondary key file

identified by the current index in the table HATTAB .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
BNP	INPUT	INTEGER	Number of pages

147. SHSBSP

CALLING convention :

CALL SHSBSP(BSP)

PURPOSE :

Set the current split pointer of a primary key file identified by the current index in the table HATTAB .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
BSP	INPUT	INTEGER	split pointer

148. SHSBNI

CALLING convention :

CALL SHSBNI(BNI)

PURPOSE :

Set the initial number of pages of a secondary key file identified by the current index in the table HATTAB .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
BNI	INPUT	INTEGER	Initial number of pages

149. SOHBNF

CALLING convention :

SOHBNF(BNF)

PURPOSE :

Set the number of fields of a secondary key or relationship file identified by the current index in the table OPATAB .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
BNF	INPUT	INTEGER	number of field

150. SOHBMD

CALLING convention :

CALL SOHBMD(BMD)

PURPOSE :

Set the maximum of a secondary key or relationship file identified by the current index in the table OPATAB .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
BMD	INPUT	INTEGER	maximum depth

151. SOPBDI

CALLING convention :

CALL SOPBDI (I , BDI)

PURPOSE :

Set the number of bits allocated to a field (identified by I) of a secondary key or relationship file identified by the current index in the table OPATAB .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
I	INPUT	INTEGER	index of field
BDI	INPUT	INTEGER	optimal number of bits

152. SOPBWI

CALLING convention :

CALL SOPBWI (I , BWI)

PURPOSE :

Set the number of bits to allocated to a descriptor field (identified by I) of a secondary key or relationship file identified by the current index in the table OPATAB .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
I	INPUT	INTEGER	index of field
BWI	INPUT	INTEGER	optimal number of bits for a descriptor field

153. SOABCV

CALLING convention :

CALL SOABCV (I , BCV)

PURPOSE :

Set the choice vector pointer (identified by I) of a secondary key or relationship file identified by the current index in the table OPATAB .

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
I	INPUT	INTEGER	index of choice vector
BCV	INPUT	INTEGER	choice vector pointer

4. DBHRAN

DBHRAN consists of random input/output routines used by DBHUSE and DBHLOW to transfer pages of the database between the main memory and the DBF.

RANDCL

CALLING CONVENTIONS :

```
CALL RANDCL (FDESG)
```

PURPOSE :

To close a file opened for random I/O

ARGUMENTS : NAME	USAGE	TYPE	DESCRIPTION
FDESG	INPUT	INTEGER	FILE designator returned by RANDOP

RANDOP

CALLING CONVENTIONS :

```
CALL RANDOP (LIONUM , FDESG , IRC)
```

PURPOSE :

To open a file for random I/O

ARGUMENTS : NAME	USAGE	TYPE	DESCRIPTION
LIONUM	INPUT	INTEGER	Logical I/O unit number to use for the DBF
FDESG	OUTPUT	INTEGER	FILE designator other DBRAND routines
IRC	OUTPUT	INTEGER	Return code

RANDRD

CALLING CONVENTIONS :

```
CALL RANDRD (FDESG,PAGNO,BUFFER)
```

PURPOSE :

Read a given page from a random I/O file into a given buffer

ARGUMENTS : NAME	USAGE	TYPE	DESCRIPTION
FDESG	INPUT	INTEGER	FILE designator returned by RANDOP
PAGENO	INPUT	INTEGER	Page number to read
BUFFER	OUTPUT	INTEGER	Array dimentioned to the page size into which the page is read .

154. RANDOQ

CALLING CONVENTIONS :

CALL RANDOQ (LIONUM,FDESG,IRC)

PURPOSE :

Open a random file

ARGUMENTS : NAME	USAGE	TYPE	DESCRIPTION
LIONUM	INPUT	INTEGER	Logical I/O number
FDESG	INPUT	INTEGER	FILE designator returned by RANDOP
IRC	OUTPUT	INTEGER	Error Code

RANDRW

CALLING CONVENTIONS :

CALL RANDRW (FDESG,PAGENO,BUFFER)

PURPOSE :

Rewrite a given page into a random I/O file from a buffer . This page is assumed to exist in the DBF.

ARGUMENTS : NAME USAGE TYPE DESCRIPTION

FDESG	INPUT	INTEGER	FILE designator returned by RANDOP
PAGENO	INPUT	INTEGER	Page number to be rewritten
BUFFER	INPUT	INTEGER	Array dimentioned to the page size from which the page is rewritten

155. RANDWT

CALLING CONVENTIONS :

```
CALL RANDWT (FDESG ,PAGENO,BUFFER,IRC)
```

PURPOSE :

Write a given page into a random I/O file from a buffer. This page is a new page being added to the DBF .

ARGUMENTS : NAME	USAGE	TYPE	DESCRIPTION
FDESG	INPUT	INTEGER	FILE designator returned by RANDOP
PAGENO	INPUT	INTEGER	Page number to be written
BUFFER	INPUT	INTEGER	Array dimentioned to the page size from which the page is written
IRC	OUTPUT	INTEGER	Return code

5. DBHLIB

DBHLIB consists of routines which allow one to manipulate the buffer in which are stored the pages of the DBF. These routines can store and retrieve strings, logical value, words and halfwords. They can also compare strings and words. They are used by DBHLOW and DBHRAN.

156. DBGETA

CALLING CONVENTION :

```
CALL DBGETA(DBID,CDISP,BUF,PTR,LEN)
```

PURPOSE :

Obtain a character string from BUFPAG

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
DBID	INPUT	DBWORD	Pointer into BUFPAG
CDISP	INPUT	INTEGER	Character DISP where to start in BUFPAG
BUF	I/O	CHAR	Where to return string
PTR	INPUT	INTEGER	Where to start in BUF
LEN	INPUT	INTEGER	Number of characters

157. DBGETV

CALLING CONVENTION :

```
CALL DBGETV (DBID,WDISP,NVEC,VEC)
```

PURPOSE :

Obtain a vector from BUFPAG

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
DBID	INPUT	INTEGER	Pointer into BUFPAG
WDISP	INPUT	INTEGER	where to start in BUFPAG

NVEC	INPUT	INTEGER	Number of words to obtain
VEC	OUTPUT	WORD	Where to put values from BUFPAG

DBGETW

CALLING CONVENTION :
CALL DBGETW(DBID,WDISP,VAL)

PURPOSE :
Obtain value of a word from allocated BUFPAG

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
DBID	INPUT	INTEGER	Pointer into BUFPAG
WDISP	INPUT	INTEGER	displacement in words
VAL	I/O	INTEGER	Value obtained from BUFPAG

158. DBPTLW

CALLING CONVENTION :
Call DBPTLW(DBID,WDISP,LVALUE)

PURPOSE :
Put something in left half of word

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
DBID	INPUT	INTEGER	Pointer into BUFPAG
WDISP	INPUT	INTEGER	displacement into BUFPAG
LVALUE	INPUT	INTEGER	Value to put into left half

159. DBPUTA

CALLING CONVENTION :
CALL DBPUTA(DBID,CDISP,BUF,PTR,LEN)

PURPOSE :
Put character string into BUFPAG

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
DBID	INPUT	INTEGER	Pointer into BUFPAG
CDISP	INPUT	INTEGER	Displacement into BUFPAG (char)
BUF	INPUT	CHAR	Where to get string
PTR	INPUT	INTEGER	Where to start in BUF
LEN	INPUT	INTEGER	Number of characters

160. DBPUTV

CALLING CONVENTION :
CALL DBPUTV(DBID,WDISP,NVEC,VEC)

PURPOSE :
Put vector into BUFPAG

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
DBID	INPUT	INTEGER	Pointer into BUFPAG
WDISP	INPUT	INTEGER	Displacement into BUFPAG
NVEC	INPUT	INTEGER	Number of words to put in
VEC	INPUT	WORD	Where to get values to put in BUFPAG

161. DBPUTW

CALLING CONVENTION :
CALL DBPUTW(DBID,WDISP,VAL)

PURPOSE :

Change the value of a word in BUFPAG

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
DBID	INPUT	INTEGER	Pointer into BUFPAG
WDISP	INPUT	INTEGER	Displacement into BUFPAG
VAL	INPUT	INTEGER	Value to put in BUFPAG

162. IDBCMA

CALLING CONVENTION :

```
I = IDBCMA(DBID,CDISP,BUF,PTR,LEN)
```

PURPOSE :

Compare character string with a string into BUFPAG

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
DBID	INPUT	INTEGER	Pointer into BUFPAG
CDISP	INPUT	INTEGER	Displacement into BUFPAG (char)
BUF	INPUT	CHAR	String to compare
PTR	INPUT	INTEGER	Where to start in BUF
LEN	INPUT	INTEGER	Number of characters

163. IDBFND

CALLING CONVENTION :

```
I = IDBCMA(DBID,WDISPS,WDISPE,IVAL)
```

PURPOSE :

Find a word in BUFPAG

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
DBID	INPUT	INTEGER	Pointer into BUFPAG

WDISPS	INPUT	INTEGER	Integer where to start
WDISPE	INPUT	INTEGER	Integer where to end
IVAL	INPUT	INTEGER	Integer value to look for

164. IDBGET

CALLING CONVENTION :
I = IDBCMA(DBID,WDISP)

PURPOSE :
Obtain a value of a word in BUFPAG

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
DBID	INPUT	INTEGER	Pointer into BUFPAG
WDISP	INPUT	INTEGER	Integer displacement (words)

165. LDBGET

CALLING CONVENTION :
I = LDBCMA(DBID,WDISP)

PURPOSE :
Obtain a logical value of a word in BUFPAG

ARGUMENTS :

NAME	USAGE	TYPE	DESCRIPTION
DBID	INPUT	INTEGER	Pointer into BUFPAG
WDISP	INPUT	INTEGER	Integer displacement (words)