# GENESIS: A RECONFIGURABLE DATABASE MANAGEMENT SYSTEM

D. S. Batory, J. R. Barnett, J. F. Garza, K. P. Smith,
K. Tsukuda, B. C. Twichell, T. E. Wise

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

TR-86-07 March 1986

# GENESIS: A Reconfigurable Database Management System [*]

D.S. Batory, J.R. Barnett, J.F. Garza, K.P. Smith,
K. Tsukuda, B.C. Twichell, T.E. Wise

## Abstract

We present a novel, yet simple, technology which enables customized database management systems to be developed very rapidly. Over the last few years, a theory of database implementation was developed to explain the storage architectures of many commercial DBMSs (i.e., how these systems store and retrieve data) [Bat85a]. The theory identified basic components of DBMS software, required all components to have the same interface, and showed that component composition can be achieved in a simple manner.

We are designing a reconfigurable DBMS, called GENESIS, which is based on this theory. A prototype is now operational. The goal of GENESIS is to take a specification of a DBMS's storage architecture, and to have the system reconfigure itself to store and retrieve data according to this architecture.

DBMS software components can be written in a few months. When all components for a target DBMS are present, writing the architecture specification of the DBMS and reconfiguring the system takes a few hours and can be accomplished with negligible cost. Building the same DBMS from scratch can take many man-years and cost hundreds of thousands of dollars.

We believe that the reconfigurable software technology proposed herein embodies an important advance in tailoring database management systems to specialized applications. We also explain how extensions of this technology may have applications beyond DBMS software development.

[*] This work was supported by the National Science Foundation under grant MCS-8317353.

## 1. Introduction

Database management systems (DBMSs) have proven to be cost-effective tools for organizing and maintaining large volumes of data. In recent years it has become evident that there are many important database applications that do not conform to the familiar debit-credit scenario of business-oriented transactions. Statistical databases ([IEE84a]), CAD and engineering databases ([IEE82], [IEE84b]), textual databases ([Sto82], [Fal85]), and databases for artificial intelligence ([IEE83]) are examples. Owing to their unusual requirements, it is not surprising that existing 'general-purpose' DBMSs do not support these applications efficiently. Special-purpose database management systems are needed.

Database system software is presently customized in one of two ways: systems are developed from scratch ([Joh83], [Tur79]) or existing systems are enhanced ([Has82], [Sto83]). It is well-known that both approaches are exceedingly difficult, costly, and not always successful. There is a definite need for tools that simplify and aid the development of database system software.

A number of researchers have begun to address these problems in the context of specific DBMSs ([Day84], [Car84], [Sto86]). Concurrent with their work, our research has concentrated on the development of an encompassing and practical theory of DBMS implementation ([Bat82], [Bat84a], [Bat85a-b]). The theory provides a common framework to relate disparate results on a wide range of topics on database research, and reveals the basic components of DBMS software to be modules that realize simple files (file structures), linksets (record linking structures), and elementary transformations (conceptual-to-internal mappings). The storage architectures of commercial DBMSs (i.e., how systems store and retrieve data) are explained by compositions of these building blocks.

We are designing a reconfigurable database management system, called GENESIS, which is based on this theory. A prototype is now operational. GENESIS can be reconfigured into a DBMS that stores and retrieves data according to a specified storage architecture. Reconfiguration is accomplished by synthesizing the target DBMS from a library of software modules that correspond to the components of the theory. The library is extensible, so new modules can be added as needed.

Once a storage architecture has been designed, only the modules that are not present in the library must be written. As all modules are reusable, we anticipate the need for adding new modules will decrease as the library enlarges. When all modules are present, the time it takes to write the specification and to reconfigure GENESIS is a matter of hours, and can be done with negligible cost. This is in sharp contrast to the way customized DBMSs are presently developed. If the same DBMS was built from scratch, it could take many man-years and cost hundreds of thousands of dollars.

Constructing software from existing components is an old idea, and certainly every DBMS has been developed in a modular fashion. However, what distinguishes our approach from others is the way we define modules. Every module in our system is plug compatible with all other modules in that all support exactly the same interface. This enables modules to be composed quickly in many different ways by a simple linking process.

The boundaries of modules that comprise existing DBMSs are drawn differently than ours, and reflect the ad hoc nature of DBMS software design. Adding new capabilities to an existing DBMS quite often requires significant rewriting of modules. In contrast, new capabilities are usually encompassed by a single module in GENESIS, and to add a new module/layer to a storage architecture is a simple task because of plug-compatible interfaces.

The theory behind reconfigurable database systems is not yet widely known. We review the theory in Section 2, and illustrate it with the storage architecture of the MRS database management system ([Kor79]). An in-depth presentation of the theory is given in [Bat85a], along with examples of other storage architectures.

We present the design and mechanics of GENESIS in Section 3 by showing how the MRS architecture has been implemented. (Other storage architectures would be realized in an identical manner). The future of our work is outlined in Section 4. We specifically examine how concurrency control, query processing, and semantic data models fit into our framework. The MRS architecture is used to illustrate these extensions. We also outline a way that our technology might impact software development beyond the confines of a DBMS setting.

## 2. Background

Files and links are fundamental concepts in databases. A *file* is a set of records that are instances of a single record type. A relationship between two or more files is a *link*, which is a generalization of a CODASYL set. Each link relates records of one file, called the *parent file*, to records of other files, called *child files*. Links are more general than CODASYL sets in that they can express M:N relationships and that a file can serve as both parent and child in the same link.

Files and links are logical concepts; that is, their implementation is unspecified. To explain the spectrum of their implementations, two models are used: the *Transformation Model (TM)* and the *Unifying Model (UM)*. The TM formalizes the notion of conceptual-to-internal mappings and the UM codifies file structures and record linking mechanisms.

### 2.1 The Transformation and Unifying Models

A primary function of a DBMS is to map conceptual files and operations to their internal counterparts. INGRES [Sto76], for example, maps relations to inverted files. RAPID [Tur79] and SYSTEM R [Ast76] also begin with relations, but RAPID maps to transposed files and SYSTEM R maps to inverted files with record clustering. The software that performs conceptual-to-internal mappings is the physical database component of a DBMS.

An intuitive understanding of conceptual-to-internal mappings is gained by recognizing that a mapping is a sequence of database definitions that are progressively more implementation-oriented. The sequence begins with definitions of the conceptual files and their links, and ends with definitions of the internal files and their links. Each intermediate definition contains both conceptual and internal elements, and thus can be identified with a *level of abstraction* that lies *between* the conceptual and internal levels. In this way, physical databases can be viewed at different levels of abstraction.

Distinguishing different levels in a DBMS and mapping from one level to an adjacent level is usually straightforward. In the DBMSs that we have studied, only ten different primitive mappings, henceforth called *elementary transformations*, have been used. Elementary transformations map files and links from one level of abstraction to a lower level. Classical transformations include indexing, encoding, transposition, segmentation, record fragmenting (also called division), the CODASYL representation of n:m relationships, and horizontal partitioning. It follows that the conceptual-to-internal mappings of a software-based DBMS can be modeled by 1) taking a generic description of the conceptual files and conceptual links that the DBMS supports, and 2) applying a well-defined sequence of elementary transformations to produce the generic internal files and internal links of the DBMS. In the case of INGRES, SYSTEM R, and RAPID, all begin with the same conceptual files (i.e., relations), but each is distinguished by different sequences of transformations and hence different sets of internal files and internal links. Modeling storage architectures using elementary transformations is the basis of the *Transformation Model (TM)* [Bat85a].

Once the internal files and internal links are known, the storage structures of the DBMS's internal database are specified using the *Unifying Model (UM)* [Bat82], which distinguishes file structures from link structures. A *simple file* is a storage structure that organizes records of one or more internal files. Classical simple files include hash-based, indexed-sequential, B+ trees, dynamic hash-based, and unordered files. A *linkset* is a storage structure that implements one or more internal links. Classical linksets include pointer arrays, inverted lists, ring lists, hierarchical sequential lists, and record clustering (i.e., 'store near' [Dat82]). It follows that the storage structures of an internal database are specified by assigning each internal file to a simple file and each internal link to a linkset. Catalogs of recognized simple files and linksets are given in [Bat85a].

It is important to recognize that conceptual-to-internal mappings and elementary transformations are not artificial concepts. Each elementary transformation can be realized by a simple layer of software (i.e., an abstract data type). In turn, the physical database software of a DBMS can be seen as a sequence of these layers, where the software of different DBMSs are described by different sequences (i.e., different nestings of abstract data types). The idea of level of abstraction corresponds to the files and links of a DBMS that are *visible at a particular level in its software*. Thus, conceptual-to-internal mappings and elementary transformations are fundamental to the way DBMS software is actually written *or can be written*.

We illustrate the TM and UM by modeling the operational database system MRS ([Kor79]). We choose MRS because of its simplicity. Models of INQUIRE, ADABAS, SYSTEM 2000 are given in [Bat85a], IDMS is described in [Bat84a], INGRES and RAPID are found in [Bat85b], and preliminary models of DMS-1100, IMS, TOTAL, SPIRES, and CREATABASE are outlined in [Bat84b] and [Cas86].

As aids to explain the transformation model of MRS, we will use three different diagrams: *data structure diagrams (dsd)* show the relationships among files and links at a particular level of abstraction, *field definition diagrams (fdd)* indicate the fields of record types for each file shown in a dsd, and *instance diagrams (id)* illustrate both dsds and fdds (see Figures 2.2 - 2.4 for examples). Boxes in data structure diagrams represent files and arrows are links. Boxes in field definition diagrams represent record types. (There are no arrows in fdds). Boxes in instance diagrams represent record instances and arrows are pointers.

In addition to the usual conventions for drawing dsds, we use two others. First, abstract objects (typically files) are indicated by dashed outlines in data structure diagrams. Figure 2.1 shows a data structure diagram of an abstract file W and its materialization as the concrete files F and G and concrete link L.

We apply the terms *abstract* and *concrete* to files and links that are relative to a given level of abstraction, where an abstract file or link is realized by its concrete counterparts. The terms *conceptual* and *internal* represent the most abstract and most concrete representations of files and links in a storage architecture.

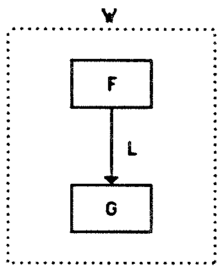


Figure 2.1. Materialization of Abstract File W

Second, pointers to abstract records arise naturally in storage architectures. In order to give such pointers a physical realization (i.e., a physical address or symbolic key), they must ultimately reference internal records. To define how pointer references are mapped, we rely on the orientation of record types within a dsd. The orientation of F and G in Figure 2.1 shows that file F is above file G. We say that F *dominates* G. This means that a pointer to an abstract record of type W will actually reference its corresponding concrete record of type F. For almost all transformations, there is a 1:1 correspondence between abstract records and their dominant concrete records; the only exception of which we are aware is full transposition (see [Bat85a]). Note that the dominance concept is recursive; that is, a pointer to a W record is the same as the pointer to its F record, which is the same as the pointer to the dominant record of the F record, and so on. In this way, pointers to abstract records are mapped to internal records.

## 2.2 A Model of the MRS Database System

MRS is a relational database management system that was implemented at the University of Toronto ([Kor79]). MRS creates a distinct internal database for each conceptual file that is defined by a user. Relationships between different conceptual files are realized by joins rather than by storage structures. The underlying storage architecture of MRS, therefore, can be revealed by examining how records of a single conceptual file are stored.

The generic CONCEPTUAL record type that is supported by MRS is shown in Figure 2.2. It consists of n data fields, $F_1 \cdots F_n$, where n is user definable. Each data field has a fixed length, so CONCEPTUAL records have fixed lengths. CONCEPTUAL files are the files that are defined in MRS schemas; CONCEPTUAL records are the records that are processed by MRS users. The id of Figure 2.2 shows a CONCEPTUAL record with the value $v_1$ in field $F_1$ and value $v_n$ in field $F_n$.

The internal files of MRS are materialized in the following way. MRS first maps CONCEPTUAL files to inverted files. The mapping produces a DATA file, where DATA records are in one-to-one correspondence

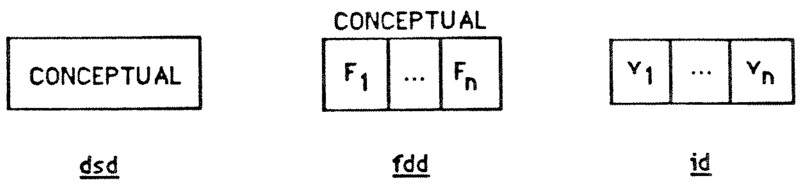


Figure 2.2. The CONCEPTUAL Record Type of MRS

with CONCEPTUAL records. In addition, for each field $F_j$ that is to be indexed, an ABSTRACT_INDEX$_j$ file is created. It is connected to the DATA file by link $I_j$ (see Fig. 2.3). $I_j$ is implemented by an inverted list.

If r fields are indexed, there will be one DATA file and r ABSTRACT_INDEX files each connected to the DATA file by precisely one link. The dsd of Figure 2.3 shows this relationship. (Note the notation $(\ )^j$ in Figure 2.3 means that zero or more fields may be indexed, each with a different value for j). In the case that no fields are indexed, CONCEPTUAL records are mapped directly to DATA records.

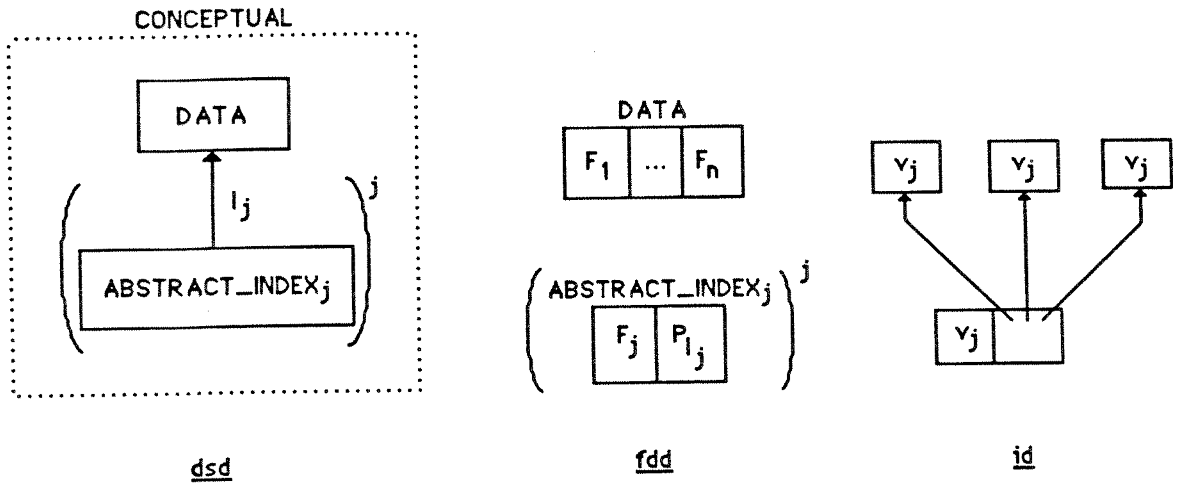


Figure 2.3. Indexing of CONCEPTUAL Fields

In the dsd of Figure 2.3, note that the DATA record type is identical to its CONCEPTUAL counterpart. The ABSTRACT_INDEX$_j$ record type has fields $F_j$ and $P_{I_j}$. $P_{I_j}$ is called the *parent field* of linkset $I_j$. It contains the parent structures (i.e., inverted list) of $I_j$ (see id of Figure 2.3).

ABSTRACT_INDEX$_j$ records are variable in length. As the file structures that MRS uses cannot store variable-length records, ABSTRACT_INDEX$_j$ records are *divided* into one or more fixed-length fragments. The first fragment, called an INDEX$_j$ record, contains the data field $F_j$ and the first pointer of the inverted list. The other fragments, called LIST$_j$ records, contain the remaining pointers. (A motivation for this division was that if the data value was a primary key, no LIST$_j$ records would be needed). INDEX$_j$ is connected to LIST$_j$ by link $L_j$. $L_j$ is implemented as a linear list. The id of Figure 2.4 shows a division of an ABSTRACT_INDEX$_j$ record that resulted in three fragments: an INDEX$_j$ record and two LIST$_j$ records. Note that in the fdd of Figure 2.4 fragments of field $P_{I_j}$ are denoted by $+P_{I_j}$. $P_{L_j}$ is the parent field of link $L_j$ (it contains the pointer to the first LIST$_j$ record) and $C_{L_j}$ is the *child field* of link $L_j$ (it contains a pointer to the next LIST$_j$ record).

DATA, INDEX$_j$, and LIST$_j$ are the internal files of MRS and $I_j$ and $L_j$ are the internal links. As mentioned earlier, the $I_j$ links are implemented by inverted lists and the $L_j$ links by linear lists. Each internal file is stored in its own file structure. DATA records are organized by an unordered file. For each j, INDEX$_j$ and LIST$_j$ records are respectively organized by a B-tree and unordered file. Thus, if r fields are indexed, there would be a total of 2*r+1 file structures.

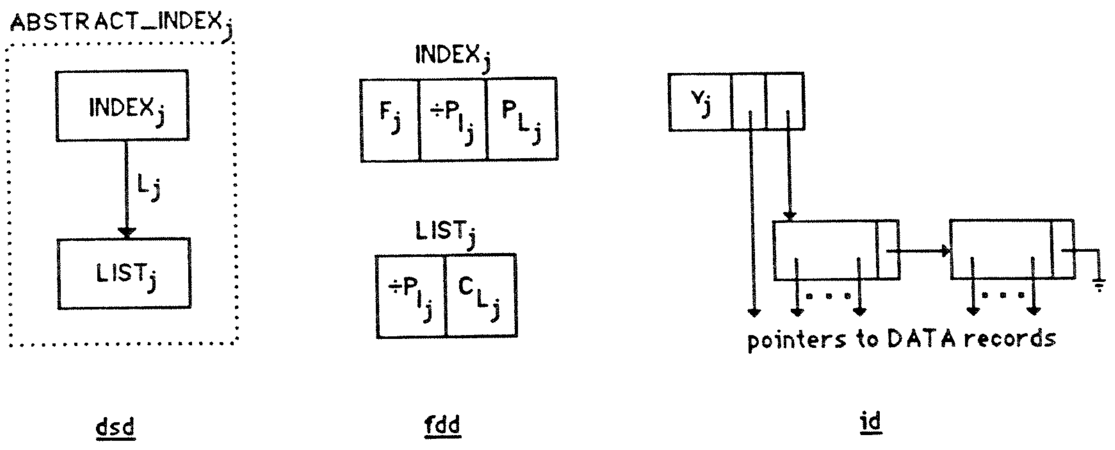


Figure 2.4. Division of ABSTRACT_INDEX

Figure 2.5 summarizes the storage architecture of MRS. The four tables list how abstract files are mapped to concrete files (Fig. 2.5b), and how internal files and internal links are mapped to simple files and linksets (Fig. 2.5c-e).

## 2.3 Comments

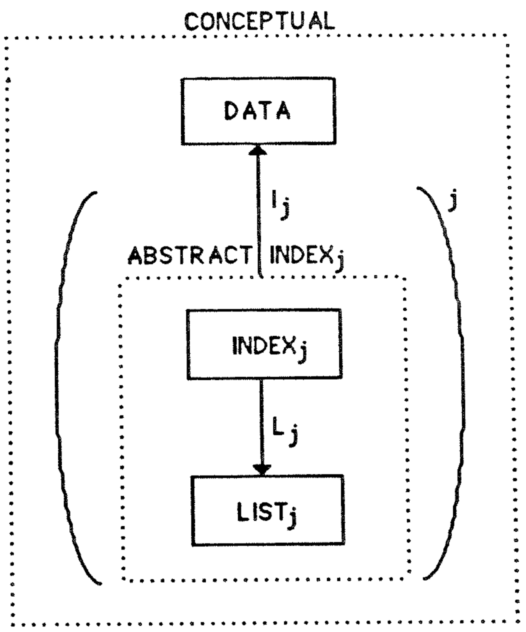
The model of MRS accounts for a considerable amount of implementation detail, all of which are essential to the construction and function of MRS software. It is this level of detail that enables our models to be used as blueprints for DBMS storage architectures.

Architecture models express the conceptual-to-internal mappings of data in a DBMS. They also can be used to explain the mappings of operations. MRS, for example, maps operations on CONCEPTUAL files to operations on DATA and ABSTRACT_INDEX files; operations on ABSTRACT_INDEX files are mapped to operations on INDEX and LIST files.

We are familiar with many database storage and retrieval algorithms that have been published in the last fifteen years, and are not aware of any practical example that does not fit our layered paradigm. We are convinced that layering has been an implicit part of explaining and developing database algorithms in the past. Our research demonstrates the importance of explicit layering.

In the next section, we show how GENESIS implements the TM and UM.

(a)



(b)

| Abstract File | Elementary Transformation |
|---|---|
| CONCEPTUAL | MRS_INDEX (extraction) |
| ABSTRACT_INDEX$_j$ (for all j) | MRS_DIVISION (division) |

(c)

| Link | Linkset |
|---|---|
| $I_j$ (for all j) | inverted list |
| $L_j$ (for all j) | multilist |

(d)

| Internal File | Simple File |
|---|---|
| DATA | Data_FS |
| INDEX$_j$ (for all j) | Index_FS$_j$ |
| LIST$_j$ (for all j) | List_FS$_j$ |

(e)

| Simple File | Implementation |
|---|---|
| Data_FS | unordered |
| Index_FS$_j$ (for all j) | B- tree |
| List_FS$_j$ (for all j) | unordered |

Figure 2.5. The Storage Architecture of MRS

## 3. The GENESIS Prototype

A fundamental precept of the TM is the uniformity with which abstract and concrete files and links are treated. That is, the record types that can be defined and the operations that can be performed at the conceptual level are exactly the same as those at the internal and intermediate levels. Thus, a single DDL and DML is sufficient to express the files, links, and operations that occur at every level of abstraction.

Defining a DDL and DML that is suitable for use at every level is perhaps the most important and difficult task in the design of GENESIS because not just any interface can be used. The DDL and DML of existing systems (e.g., CODASYL) are much too specific or clumsy. For this reason, the present GENESIS interface is synthetic. The DDL is based on an amalgam of concepts taken from existing DDLs and programming languages. The DML is primarily based on embedded-SQL ([Cha76], [Gra78]) and the extension of SQL to handle long fields ([Has82]).

We begin with an overview of the GENESIS prototype. We then outline the interface that is shared by every layer of software that realizes an elementary transformation, and explain the implementation of each layer in terms of data mappings and operation mappings. We conclude by showing how the composition of mappings is achieved.

*The basic ideas of each section are summarized in the introductory paragraphs. Further details, for those who are interested, are presented in the paragraphs that follow the bullet ( • ) marker.*

### 3.1 Organization and Overview

Figure 3.1 shows the components of the GENESIS prototype. Their function and relationship, in addition to the people who interact with GENESIS, are summarized below.

The *Database Architecture Administrator (DAA)* is responsible for selecting a storage architecture for the target DBMS. He may choose an existing architecture or create a new one. The DAA reconfigures GENESIS to this architecture by writing an *architecture program.* This program compiles conceptual schemas using the *DDL compiler* and maps the data definitions of conceptual files and conceptual links to their internal counterparts. The mappings are accomplished by prewritten procedures called *type transformers* which realize the abstract-to-concrete data definition mappings of elementary transformations. Unless the DBMS architecture needs to be changed, there is no need to alter an architecture program once it is written. Different storage architectures are realized by different architecture programs.

The *Database Administrator (DBA)* is responsible for database design. He develops conceptual schemas in terms of the GENESIS DDL, and runs the architecture program to convert these schemas into an internal representation called *storage architecture tables.* If schemas are modified after they have been compiled, they must be recompiled in order to update their storage architecture table representations.

*Database users* write transactions to process database retrievals and updates. The host language is C [Ker78]. The record types that can be defined in GENESIS are more general than the types supported by C, so users are supplied routines to read and manipulate buffer-resident GENESIS records. These routines compose the *trace manager.* Records are transferred between main memory buffers and secondary storage by *file operations* which are accessed via the Grand Central module.

*Grand Central* serves as a routing circuit to modules called *expander layers.* Each expander layer defines the abstract-to-concrete operation mappings for an elementary transformation. The number of expander layers equals the number of elementary transformations needed by a storage architecture. When an operation O on an abstract file is to be executed and the abstract file is materialized by the elementary transformation T, Grand Central causes operation $O_T$ of the expander layer for T to be executed. Every expander layer calls the trace manager to aid in the reading and manipulation of main-memory resident GENESIS records.

Abstract operations are eventually mapped to operations on internal files, which are processed by JUPITER, the file management system of GENESIS. JUPITER supports both single-keyed and multi-keyed simple files through a single interface, and handles all I/O between main-memory and secondary storage.

In summary, data definition mappings of elementary transformations are handled by type transformers. Operation mappings are handled by operation expanders. A database schema is compiled and mapped by an architecture program, which calls the DDL compiler and type transformers. A transcript of these mappings is stored in architecture tables. A transaction initiates operations on conceptual files and conceptual links. Grand

Central uses the contents of architecture tables to direct the translation of conceptual operations to their internal counterparts. Operation expanders are called to perform the level-by-level mappings.

The ANSI/SPARC role of database users, who write and execute transactions, and the DBA, who designs and writes database schemas, remain unchanged ([Tsi78]). Reconfigurable DBMSs, such as GENESIS, requires an additional party, the DAA, who is responsible for the construction and customization of a DBMS.

We estimate that it may take several days or weeks for the DAA to determine a satisfactory storage architecture for a given class of applications. It has been our experience that a typical expander layer and its transformer can be designed, coded, and debugged in one or two months. Once a storage architecture has been chosen, and when all of the expanders and transformers that are needed are available, GENESIS can be reconfigured to store and retrieve data according to the target architecture in the time it takes to write the DBMS's storage architecture program. This can done in a few hours.

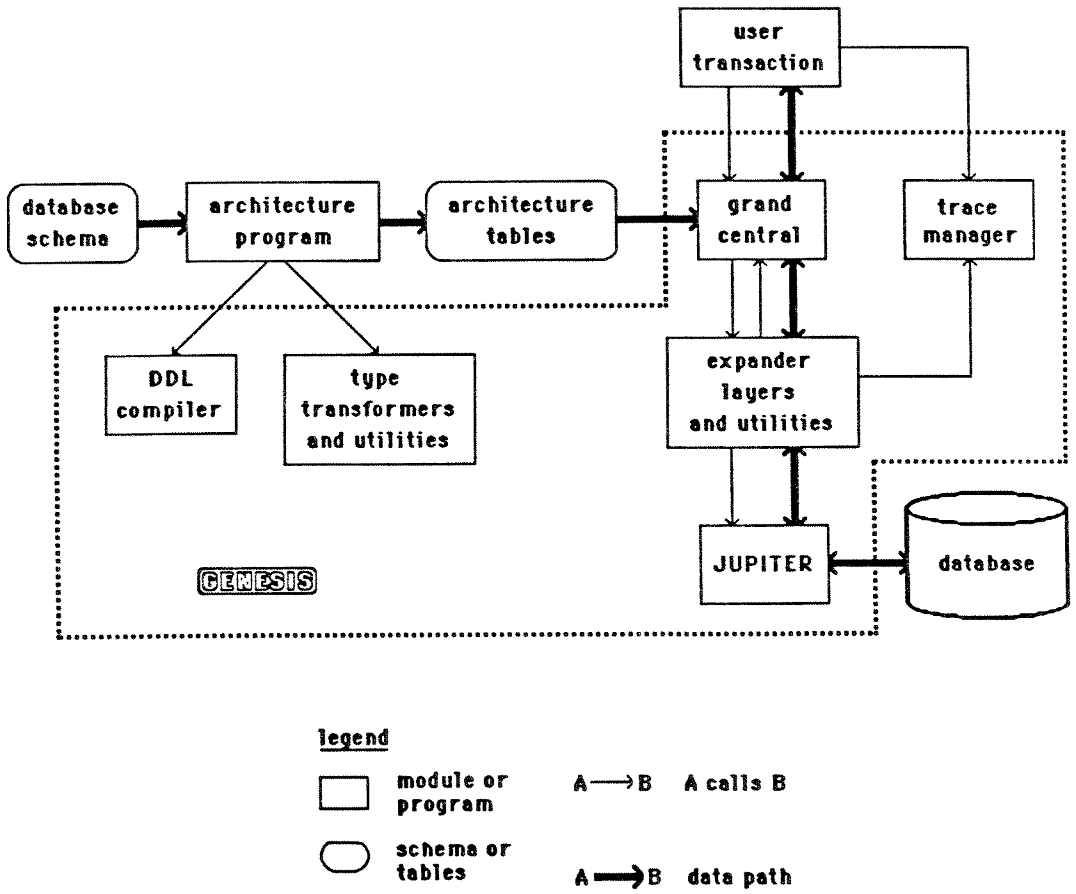


Figure 3.1 The Configuration of the GENESIS Prototype

## 3.2 The Layer Interface

### 3.2.1 DDL

The GENESIS DDL is a mixture of record structuring concepts taken from the C, PASCAL, and COBOL programming languages, and the set constructs of DBTG. GENESIS record types can contain fixed-length and variable-length fields, scalar and set-valued attributes (henceforth called *scalar fields* and *repeating fields)*, nested scalar and nested repeating fields, and matrices. These constructs underly *non-first normal form relations* ([Osz85], [Rot84], [Jae82]). Links support N:M relationships, information carrying sets ([Tsi77]), and multimember sets ([Dat82]).

As mentioned earlier, the GENESIS DDL is used to declare a spectrum of schemas from the conceptual to the internal levels. As an example of a conceptual schema, Figure 3.2 shows a typical MRS (relational) schema, i.e., one that contains no nested definitions, LINK, or SET constructs. Figure 3.3 is an example of an internal schema, which describes an inverted file (at a level of detail comparable to Figure 2.3).

• A schema consists of a sequence of four declarations: OPTIONS, TYPES, FILES, and LINKS ([Smi85]). There are two FILES in the database of Figure 3.3: Employee and Ename_Index whose types are Employee_Type and Ename_Index_Type, respectively. The Employee_Type lists three scalar fields: Emp#, Ename, Salary, where Ename is a variable-length string of at most 12 characters. The Offspring field is repeating and can contain at most 20 elements; the repeating unit is defined by the Dependent_Type, which itself contains a variable-length and a fixed-length field.

The Ename_Index_Type has two fields: Ename (the Employee field which is being inverted) and Inverted_List. The Inverted_List field is an *unbounded set* of POINTERs (i.e., a set whose cardinality has no limit). The LINKS statement declares Ename_Index to be the parent file and Employee the child file of link EI. The link key ([Bat82], [Bat85a]) for both files is Ename, and the parent field of link EI is Inverted_List.

The OPTIONS statement declares tags that can be associated with individual fields, files, or links. The option 'primary_key' tags fields Emp# of Employee_Type and Ename of Ename_Index_Type. 'unordered' tags the Employee file and 'bplus' tags Ename_Index.

OPTIONS are used as storage architecture directives. For example, 'bplus' and 'unordered' are used to tag files that are to be stored in B+ trees and unordered file structures, respectively. The role of OPTIONS in storage architecture design will be amplified in Section 3.3.2 (Type Transformers).

## 3.2.2 DML

The basic objects in a GENESIS database are records, files, and links. Each has its own data retrieval and manipulation operations. Record operations process records that are in main-memory, file operations transfer main-memory records to and from secondary storage, and link operations traverse and alter link occurrences.

*Cursors* are run-time mechanisms that are used to reference objects and to express operations on objects ([Gra78]). Records, files, and links have their own types of cursors.

In the prototype, only record and file cursors are explicitly supported. A link cursor can be realized by one or more file cursors, and link operations are primarily accomplished by calls to file operations. For this reason, ad hoc support of links will suffice for the short-term. However, a general-purpose link manager is a long-range goal.

The cursors and operations that are supported by GENESIS are reviewed in the remainder of this section.

### 3.2.2.1 Traces and Record Operations

A GENESIS record is an unnormalized relational tuple and is structured as an ordered tree of fields. The root represents an entire record. Its children are its immediate subfields, their children are their subfields, and so on. The ordering of nodes reflects the ordering in which fields appear in the record. Leaves correspond to scalar fields defined over primitive types, such as characters, bytes, floats, and integers, or strings of these types. Repeating groups and matrices are represented by non-leaf nodes, where their elements are treated as subfields.

Figure 3.4 is a tree representation of an Employee record, where the Employee type was defined in Figure 3.3. Beside each node is the name of the corresponding field and its ordinal number in parentheses. Under each leaf node is the data value that is contained in the field. The depicted record has Emp#=4179, Ename='Jones, Ed', Salary=40,000, and two Offspring: Al and Cindy, both of age 14.

The field name of a node cannot be used as its identifier, as many nodes may share the same name. Instead, nodes are identified by their *trace*, i.e., the path from the root to the node in question. The ordinal trace to the field containing 'Cindy' in Figure 3.4 is (1,4,2,1). Traces serve as cursors to fields of records in GENESIS.

Unlike many DDLs where there is a compatibility of the record types of the host language and DBMS, GENESIS record types are more general than those provided by the C language. Special routines, called *trace operations*, are used to read and update fields in GENESIS records *that are buffer resident*. The collection of all trace operations is the *trace manager*.

```
DATABASE      MRS_schema
{
    OPTIONS
        primary_key;     /* primary key option  */
        indexed;         /* index option        */

    TYPES
        Professor_Type =
        {
            Pname           STRING (10) OF CHAR   primary_key indexed;
            Office#         INT;
            Building        STRING (4) OF CHAR    indexed;
            Campus_Phone    INT;
        };

    FILES
            Professor       Professor_Type;
}.
```

Figure 3.2 An MRS (Relational) Schema

```
DATABASE      Inverted_File
{
    OPTIONS
        primary_key;        /* primary key option              */
        bplus;              /* b+ tree implementation          */
        unordered;          /* unordered file implementation   */

    TYPES
        Dependent_Type =
        {
            First_Name      STRING (8) OF CHAR;
            Age             INT;
        };

        Employee_Type =
        {
            Emp#            INT                           primary_key;
            Ename           STRING (12) OF CHAR;
            Salary          FLOAT;
            Offspring       SET (20) OF Dependent_Type;
        };

        Ename_Index_Type =
        {
            Ename           STRING (20) OF CHAR           primary_key;
            Inverted_List   SET (*) OF POINTER;
        };

    FILES
            Employee        Employee_Type                 unordered;
            Ename_Index     Ename_Index_Type              bplus;

    LINKS
        EI =
        {
            PARENT.LKEY  =  Ename_Index.Ename;
            PARENT.FIELD  =  Ename_Index.Inverted_List;
            CHILD.LKEY  =  Employee.Ename;
        };
}.
```

Figure 3.3 An Example GENESIS Schema

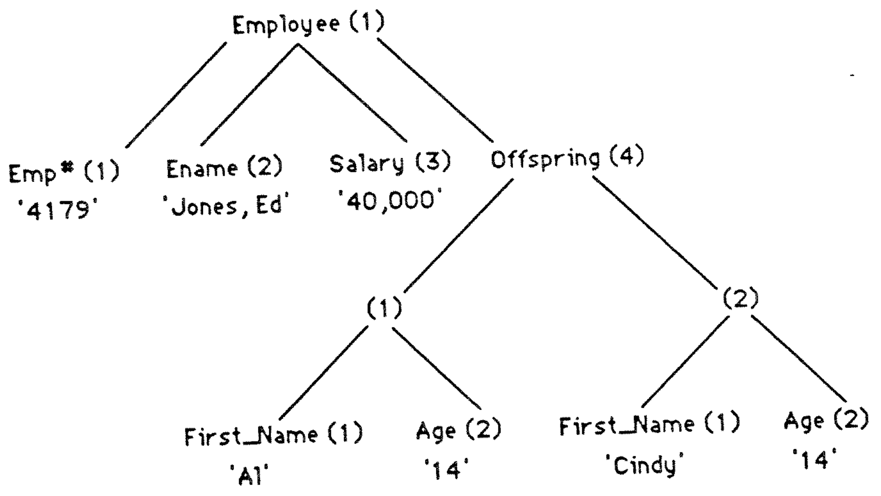


Figure 3.4. A Tree Representation of an Employee Record

| Operation Type | Trace Operation | Semantics |
|---|---|---|
| Navigation | | |
| | LEFT( trace ) | reposition trace to left sibling |
| | RIGHT( trace ) | reposition trace to right sibling |
| | UP( trace ) | reposition trace to parent node |
| | DOWN( trace, nth_child ) | reposition trace to the nth child |
| Manipulation | | |
| | RD( trace, buffer ) | read scalar field into buffer |
| | WR( trace, buffer ) | write buffer into scalar field |
| | AD( trace, buffer ) | add element in buffer to repeating field |
| | DL( trace, nth_child ) | delete nth_element of repeating field |
| Utility | | |
| | MAKE_TRACE( trace, field_name ) | create trace to specified field |
| | DROP_TRACE( trace ) | deallocate trace |
| | INIT_FIELD( trace, buffer ) | initialize field in buffer |
| | COUNT_CHILDREN( trace ) | return number of subfields of given field |
| | LEN( trace ) | return length of field |
| | LOC( trace ) | return starting address of field |

Figure 3.5. A Partial List of Trace Operations

There are three different groups of trace operations: navigational, manipulation, and utility. *Navigational* operations position traces, *manipulation* operations read and update individual fields, and *utility* operations create and delete traces dynamically and provide encoding information (e.g., the number of elements in a repeating field, the field length in bytes, etc.). A partial list of trace operations is given in Figure 3.5. A detailed description of the trace manager is given in [Smi85].

### 3.2.2.2 File Cursors and File Operations

GENESIS operations on files are patterned after embedded SQL operations ([Cha76]) and operations on long fields ([Has82]). A partial list of operations is given in Figure 3.6.

• Every file in a GENESIS database is assigned an identifier called a *file handle* or *mt_id*, which is used to identify a cursor with a file. Cursors can be dynamically allocated, deallocated, and reassigned to different files.

Records of a file that satisfy a query can be retrieved in two different ways. One way is to initialize a cursor for file searching (using the RET operation) and to advance the cursor repeatedly (using ADV) to return the sequence of qualified records one at a time. Note that queries involving joins (i.e., predicates over multiple files) are not handled by RET; such queries must be *decomposed* into simpler queries over single files prior to their processing ([Won76]). Query decomposition is discussed in Section 4.2.

A second way to retrieve records is to follow pointers. The INIT operation prepares a cursor for direct accessing, and the ACC operation retrieves a record given its pointer. The procedure to follow an inverted list, for example, is accomplished by INITializing a cursor, storing a pointer in the cursor, and ACCessing the record. The latter two steps are repeated for each pointer on the list.

Pointers to records can be symbolic or physical addresses, depending on how the records are stored in simple files. No distinction between pointer types is made by ACC or INIT. In this way, storage structures can change at the internal level without affecting higher-level algorithms. However, a distinction is made inside JUPITER, the file management system of GENESIS (see Section 3.3.1).

Other operations include modification (UPD) and deletion (REM) of the current record of a cursor, and record insertion (INS), which repositions a cursor to the inserted record.

Repeating fields can be retrieved and updated in fragments by the GET and PUT operations. These operations use *long field* cursors to indicate the sequence of elements in the field that are to be transferred. Fragments always contain an integral number of elements.

Figure 3.6 summarizes these operations. There are other operations, such as database creation, open, and close; transaction start, abort, and commit; and file loading and sorting. They have straightforward semantics and are not considered here.

| Operation | Semantics |
|---|---|
| MAKE_FILE_CURSOR( F_cursor, mt_id ) | create file cursor for file with handle mt_id |
| DROP_FILE_CURSOR( F_cursor ) | delete file cursor |
| REASSIGN( F_cursor, new_mt_id ) | reassign file cursor to another file |
| RET( F_cursor, query, into_list, position ) | prepare F_cursor for record retrieval. query is a selection predicate, into_list contains trace-buffer pairs for the input and output of individual fields, position specifies initial positioning of F_cursor (before_first or at_first record) |
| ADV( F_cursor ) | advance F_cursor to point to next record to be retrieved |
| INIT( F_cursor, query, into_list ) | prepare F_cursor for pointer following |
| ACC( F_cursor ) | follow pointer in F_cursor to access record |
| REM( F_cursor ) | delete record referenced by F_cursor |
| UPD( F_cursor ) | update record referenced by F_cursor; trace-buffer pairs that are flagged on F_cursor's into_list indicate the fields to be updated and the buffers containing their new values |
| INS( F_cursor, hold_option ) | insert record into the file of F_cursor; field values of record are referenced in F_cursor's into_list; hold_option is true if record is to be updated shortly after insertion |
| MAKE_FIELD_CURSOR( F_cursor, trace, L_cursor ) | create a cursor to a long field |
| DROP_FIELD_CURSOR( L_cursor ) | delete cursor to long field |
| GET( L_cursor, #_of_elements, buffer, buf_size ) | long field retrieval |
| PUT( L_cursor, #_of_elements, buffer ) | long field update |

Figure 3.6 A Partial List of File Operations

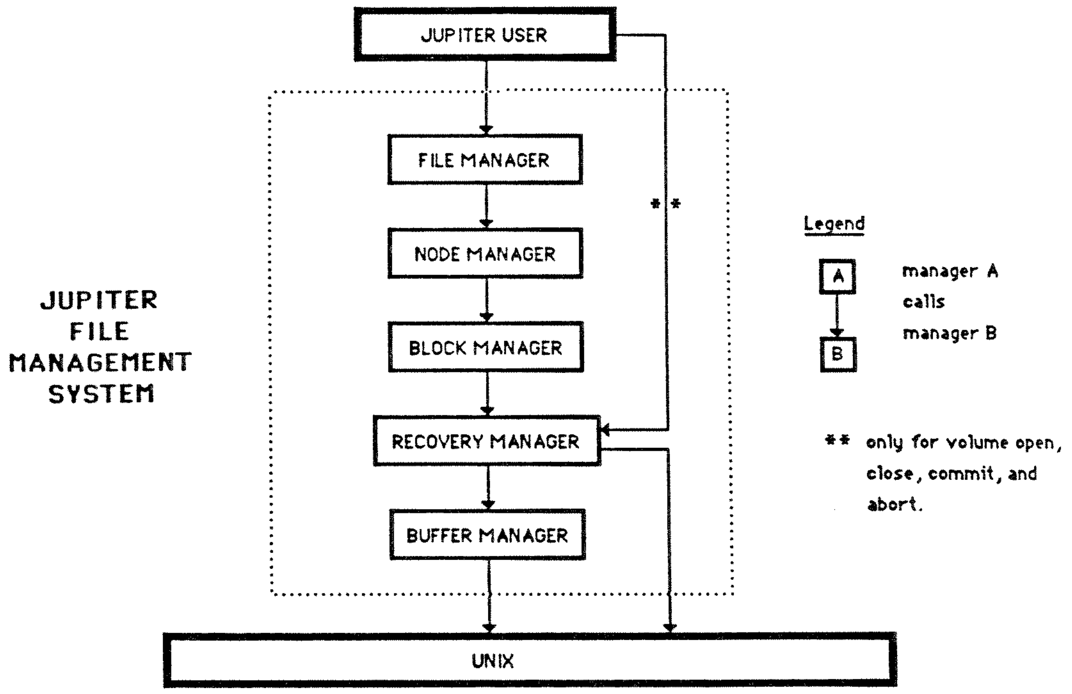
## 3.3 The Layer Implementation

Every layer of software that realizes an elementary transformation features the interface described in Section 3.2. We explain in the following sections how layers are implemented.

We begin with a brief description of JUPITER, the file management system and the lowest layer of GENESIS. Implementing elementary transformations requires the abstract-to-concrete mappings of both data definitions and operations. We show how data definitions are mapped by procedures called type transformers, and how file operations are mapped by procedures called operation expanders. We conclude by explaining how expanders relate to software layers and how operation mappings are composed.

### 3.3.1 The JUPITER File Management System

JUPITER is a general-purpose file management system ([Gar85]). It is composed of five layers and resides on UNIX (See Fig. 3.7). The bottom layer is the *buffer manager*. It handles block I/O and coordinates the usage of buffers via an LRU replacement strategy. The next higher layer is the *recovery manager*, which uses Lorie's shadowing algorithm ([Lor77]). Above the recovery manager is the *block manager*. It handles the storage and retrieval of records within buffer-resident blocks. Records can either be fixed-length or variable-length, and can be either anchored (i.e., have fixed storage locations) or unanchored. In all, four different block formats are handled.



JUPITER USER
FILE MANAGER
NODE MANAGER
BLOCK MANAGER
RECOVERY MANAGER
BUFFER MANAGER
UNIX
JUPITER FILE MANAGEMENT SYSTEM
**
Legend
A
B
manager A calls manager B
** only for volume open, close, commit, and abort.

Figure 3.7 The Organization of JUPITER

*Nodes* or *frames* have been shown to be the basic components of file structures ([Mar81], [Bat82]). The *node manager* provides nodes and operations on nodes as primitives, and relies on the block manager for lower-level support. Four different types of node implementations are available: 1) primary block only, 2) primary block with unshared overflow blocks , 3) primary block with shared overflow blocks, and 4) unshared overflow blocks only. Each is illustrated in Figure 3.8. (Note that shared overflow blocks contain records of different nodes, whereas unshared overflow blocks contain records of a single node).

The highest layer is the *file manager*. It presents a uniform interface to all single-keyed and multi-keyed file structures. By doing so, a measure of data independence is achieved. Simple file implementations in a DBMS storage architecture can be changed without forcing modifications to the software that reference them. JUPITER presently features indexed-sequential, indexed-aggregate, B+ tree, deferred B+ tree, unordered,

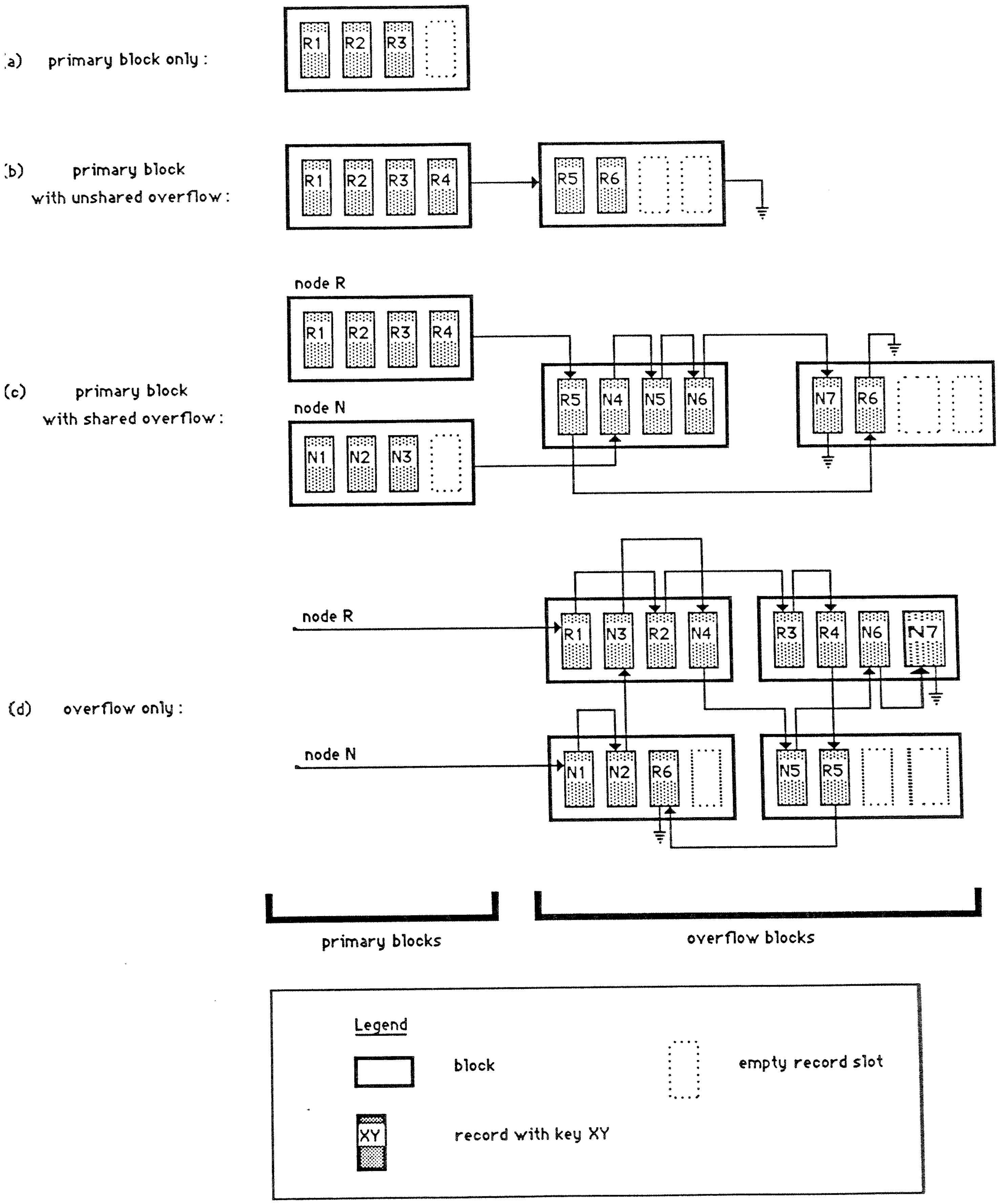


Figure 3.8 Node Implementations in JUPITER

hash-based, heap, and multi-keyed hashing file structures. (Useful variations of these structures can be generated by altering their node and block implementations ([Gar85]). This is easily done by resetting the block and node implementation flags prior to file creation).

JUPITER is extensible. Different buffer and recovery management algorithms, such as those described in ([El184], [Eff84]), have been introduced by replacing the existing buffer or recovery manager while retaining the same interface. (Recovery using logs would require more extensive changes). New file structure algorithms can be added easily. Furthermore, they can be coded quickly because many of the difficult-to-write primitives are already provided by the block and node managers. A detailed description of JUPITER is forthcoming.

## 3.3.2 Type Transformers and Architecture Programs

A formalization of conceptual-to-internal mappings are *architecture programs*, which translate the data definitions of conceptual files (e.g., record types) to their internal counterparts using *type transformers*. ([Tsu85]). Type transformers are procedures that handle the details of entering architecture specifications into tables, called *storage architecture tables*. These tables are used to direct the conceptual-to-internal mappings of file operations.

An architecture program for MRS is shown in Figure 3.11. All architecture programs are very simple and very short, and hence can be written quickly. An explanation of this program and architecture programs in general is given below.

- A *volume* in GENESIS is an area in secondary storage. (A volume corresponds to a UNIX file in the current version of JUPITER). One or more JUPITER files can be stored in a volume, where a JUPITER file is the union of the records of one or more internal files. Each JUPITER file is organized by a separate simple file structure.

An architecture program declares all volumes, JUPITER files, and JUPITER file implementations by calls to the VOL_DCL, JUPITER_FILE_DCL, and FILE_IMPL_DCL procedures. Each procedure enters its declaration into an unused row of one of three tables: the volume table (VT), the JUPITER file table (JFT), and the file implementation table (FIT). The row number at which a declaration is stored is returned by the procedure as the *handle* or *id* to that volume, JUPITER file, or file implementation. Subsequent references to a declaration is made via its handle. (When a JUPITER file is declared, the handle of the volume in which the file is to be stored and the handle of its file structure implementation must be specified. Thus, volumes and simple file implementations are declared prior to JUPITER files. The assignment of internal files to JUPITER files is accomplished by INTERNAL, a procedure which will be discussed shortly).

The DDL_COMPILE procedure compiles GENESIS schemas into an internal representation that is stored in several tables, the most important of which is the *field definition table (FDT)*. Every record type defined in a schema is identified with a row in the FDT, called its *FDT handle (fdt_id)*.

An architecture program encodes the data definitions of a schema and its conceptual-to-internal mappings in the *mapping table (MT)*. Every file that is encountered in the conceptual-to-internal translation is identified with a row of the MT. The number of the row is the *mapping table identifier (mt_id)* of the file. As we saw in Section 3.2.2.2 (File Cursors), mt_ids are used as file handles.

The MT row of a file contains the FDT handle of the file's record type, the identifier of the elementary transformation which maps the file to one or more concrete files, the number of concrete files to which it is mapped, the mt_id of the first (the dominant) of these concrete files (the mt_ids of the subordinate files have consecutively higher mt_id numbers), and the FST handle of the simple file in which records of this file are stored (applicable if the file is internal).

MT entries are made by procedures called *type transformers* which map an abstract file to its concrete file and concrete link counterparts according to an elementary transformation. In the following, we illustrate type transformers by showing how MRS schemas are mapped. Note, however, that it is the internal representation of schemas that are actually transformed in the prototype.

Let MRS_INDEX(a_mtid, c_mtid, nindex, indx_opt) be the transformer that MRS uses to accomplish indexing. It takes as input the handle a_mtid of an abstract file and produces the handle c_mtid of the concrete data file. The handles of the index files which are created are assigned consecutive numbers beginning with c_mtid+1. The number of index files that are created is returned in parameter nindex.

MRS_INDEX looks at the OPTIONS assigned to each field of the abstract record type to determine which fields are to be inverted. If a field is flagged by the indx_opt option, another input parameter to MRS_INDEX, then an index file is created.

As an illustration, let a_mtid be the handle of an abstract file whose record type definition is shown in Figure 3.9a. The type consists of n fields $F_1 \cdots F_n$, where field $F_2$ and $F_n$ are tagged with the 'index' option. (Actually, any number of fields could be tagged; here we have chosen $F_2$ and $F_n$ as examples).

Figure 3.9b shows the result of performing MRS_INDEX(a_mtid, c_mtid, nindex, 'index'). Record types for a data file and two index files are produced, along with the declarations for their connecting links. The handle of the data file is returned in c_mtid and the handle of the index files are c_mtid+1 and c_mtid+2. The value of parameter nindex is 2. Note that the primary key fields of both index record types are tagged by the 'primary_key' option. This tagging is important for subsequent transformations.

As a general rule, OPTIONS in database schemas specify storage architecture directives such as which fields to encode or index, which file structures to use, etc. OPTIONS serve as schema-defined parameters to transformers.

Let MRS_DIVIDE(a_mtid, c_mtid, nptr) be the transformer which accomplishes the division that is used by MRS. It takes as input an abstract index file whose handle is a_mtid and produces an index file and list file whose handles are c_mtid and c_mtid+1. Parameter nptr is the maximum number of pointers to be stored in a list record. The link that connects index records to their list records is implemented as a linear list.

As an illustration, let a_mtid be the handle of an abstract index file whose record type definition is given in Figure 3.10a. The type has two fields: a primary key field F and an inverted list field $P_I$. [1]

Figure 3.10b shows the result of performing MRS_DIVIDE(a_mtid, c_mtid, nptr). Record types for an index file and a list file are produced, along with the declaration of their connecting link. Note that the index record type contains room for one pointer of the inverted list and each list record can contain as many as nptr pointers. Futhermore, field F of the index record type is tagged with the 'primary_key' option, whereas the fields of the list record type have no tags.

Let INTERNAL(i_mtid, fid , pkey_opt) be the transformer which assigns an internal file with handle i_mtid to the JUPITER file whose handle is fid. If the simple file implementation of the JUPITER file requires primary keys, as is the case with B+ trees, hash-based files, and indexed-sequential structures, internal file i_mtid must have a primary key. INTERNAL recognizes the primary key field by the 'pkey_opt' option. [2]

With the aid of some additional procedures (transformer utilities), an architecture program can be written that maps an MRS schema - consisting of one or more CONCEPTUAL files - to its internal representation. Figure 3.11 shows such a program written in C ([Ker78]). It accepts a file 'mrs_schema' containing an MRS schema as input and outputs a file 'mrs_schema_tables' containing the architecture tables for this schema. This program is a transcription of the MRS architecture presented in Section 2.2.

As an example of how an architecture program maps the record types of a specific schema, Figure 3.12 illustrates the mapping of the Professor record type defined in the MRS schema of Figure 3.2. A more detailed explanation of this example, including a listing of architecture tables, is given in Appendix I.

It is worth noting that unless a storage architecture is modified, an architecture program does not change once it is written. Different architectures are described by different architecture programs.

MRS has a rather simple storage architecture. Although other architectures can be considerably more complicated (e.g., IDMS), their architecture programs are only a few lines longer than the one presented. It is because of this simplicity that architecture programs can be developed and modified very quickly.

1 Abstract index records can also have augmented fields. We don't consider these fields here for simplicity. However, we do include them in the prototype.

2 Note that unimplementable storage architectures arise when an internal file has no primary key and its JUPITER file implementation requires one. A similar error is if two or more internal files are stored in the same JUPITER file and both internal files have different primary keys (i.e., different lengths or different starting addresses). Transformers can provide assistance in recognizing these and other errors.

(a)

```
TYPES

Abstract =                          /* abstract file */
{     F1        ...
      F2        ...                 indexed;
      ...

      Fn        ...                 indexed;
};
```

(b)

```
TYPES

Data =                              /* dominant concrete file */
{     F1        ...
      F2        ...                 indexed;
      ...

      Fn        ...                 indexed;
};

F2_Index =                          /* index record type for field F2 */
{     F2        ...                 primary_key;
      PI_F2     SET (*) OF POINTER;     /* inverted list */
};

Fn_Index =                          /* index record type for field Fn */
{     Fn        ...                 primary_key;
      PI_Fn     SET (*) OF POINTER;     /* inverted list */
};

LINKS

I_F2 =                              /* inverted list linkset */
{     PARENT.LKEY = F2_Index.F2;
      PARENT.FIELD = F2_Index.PI_F2;
      CHILD.LKEY = Data.F2;
};

I_Fn =                              /* inverted list linkset */
{     PARENT.LKEY = Fn_Index.Fn;
      PARENT.FIELD = Fn_Index.PI_Fn;
      CHILD.LKEY = Data.Fn;
};
```

Figure 3.9 Data Mappings of MRS_INDEX

(a)

```
TYPES

Abstract_Index =                    /* abstract index record type */
{     F         ...                 primary_key;    /* data field */
      PI        SET (*) OF POINTER;                 /* inverted list field */
};
```

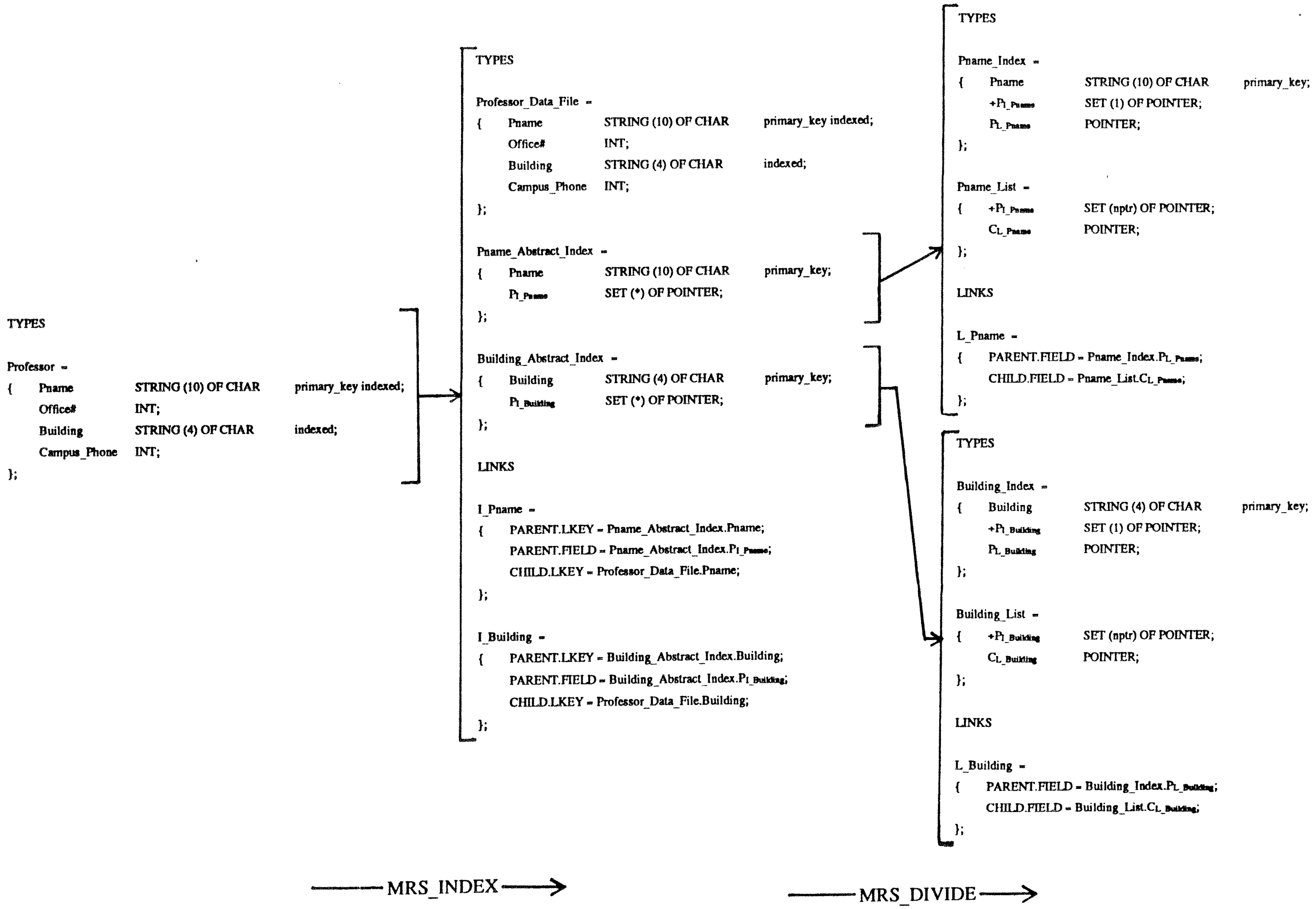
(b)

```
TYPES

Index =                             /* index record type */
{     F         ...                 primary_key;    /* data field */
      +PI       SET (1) OF POINTER;                 /* 1 pointer */
      PL        POINTER;                            /* list head */
};

List =                              /* list record type */
{     +PI       SET (nptr) OF POINTER;              /* nptr pointers */
      CL        POINTER;                            /* next pointer */
};

LINKS

L =                                 /* list linkset */
{     PARENT.FIELD = Index.PL;
      CHILD.FIELD = List.CL;
};
```

Figure 3.10 Data Mappings of MRS_DIVIDE

```
program                                                   comments
--------------------------------------------------------------------------------------------

mrs_architecture_program( mrs_schema, mrs_schema_tables )
{
  INIT_TABLES;                                            clear architecture tables

  DDL_COMPILE( mrs_schema );                              compile schema

  VOL_DCL( mrs_vol, ... );                                declare database volume

  FILE_IMPL_DCL( unordered, ... );                        declare file implementations
  FILE_IMPL_DCL( bplus, ... );

  CONCEPTUAL_FILES( nfiles );                             enter definitions of each conceptual
                                                          file in the schema into the first
                                                          rows of the mapping table; the
                                                          number of conceptual files defined
                                                          in the schema is returned in 'nfiles'.

  for (c = 1; c ≤ nfiles; c++)                            for each conceptual file c do..
  {
    MRS_INDEX( c, d, nindx, 'index' );                    map file c to its inverted file
                                                          counterparts. 'd' is the mt_id of data file,
                                                          nindex is the number of abstract_index files
                                                          created, 'index' is OPTIONS tag to flag
                                                          fields to be indexed.

    for ( ai = d+1; ai ≤ d+nindx; ai++ )                  for each abstract index file ai do..
    {
      MRS_DIVIDE( ai, i, nptr );                          divide abstract_index file
                                                          'i' is mt_id of index file;
                                                          'i+1' is mt_id of list file

      JUPITER_FILE_DCL( index_file, bplus, mrs_vol, ... );    declare B+ tree file structure
      INTERNAL( i, index_file, 'primary_key' );               store index file in B+ tree

      JUPITER_FILE_DCL( list_file, unordered, mrs_vol, ... ); declare unordered file structure
      INTERNAL( i+1, list_file, );                            store list file in unordered file

    };                                                    end abstract_index loop

    JUPITER_FILE_DCL( data_file, unordered, mrs_vol, ... );   declare unordered file structure
    INTERNAL( d, data_file, );                                store data file in unordered file

  };                                                      end conceptual loop

  WRITE_TABLES( mrs_schema_tables );                      save architecture tables
};
```

Figure 3.11 An Architecture Program for MRS



```
TYPES

Professor =
{       Pname           STRING (10) OF CHAR     primary_key indexed;
        Office#         INT;
        Building        STRING (4) OF CHAR      indexed;
        Campus_Phone    INT;
};
```

```
TYPES

Professor_Data_File =
{       Pname           STRING (10) OF CHAR     primary_key indexed;
        Office#         INT;
        Building        STRING (4) OF CHAR      indexed;
        Campus_Phone    INT;
};

Pname_Abstract_Index =
{       Pname           STRING (10) OF CHAR     primary_key;
        PI_Pname        SET (*) OF POINTER;
};

Building_Abstract_Index =
{       Building        STRING (4) OF CHAR      primary_key;
        PI_Building     SET (*) OF POINTER;
};

LINKS

I_Pname =
{       PARENT.LKEY = Pname_Abstract_Index.Pname;
        PARENT.FIELD = Pname_Abstract_Index.PI_Pname;
        CHILD.LKEY = Professor_Data_File.Pname;
};

I_Building =
{       PARENT.LKEY = Building_Abstract_Index.Building;
        PARENT.FIELD = Building_Abstract_Index.PI_Building;
        CHILD.LKEY = Professor_Data_File.Building;
};
```

```
TYPES

Pname_Index =
{       Pname           STRING (10) OF CHAR     primary_key;
        +PI_Pname       SET (1) OF POINTER;
        PL_Pname        POINTER;
};

Pname_List =
{       +PI_Pname       SET (nptr) OF POINTER;
        CL_Pname        POINTER;
};

LINKS

L_Pname =
{       PARENT.FIELD = Pname_Index.PL_Pname;
        CHILD.FIELD = Pname_List.CL_Pname;
};
```

```
TYPES

Building_Index =
{       Building        STRING (4) OF CHAR      primary_key;
        +PI_Building    SET (1) OF POINTER;
        PL_Building     POINTER;
};

Building_List =
{       +PI_Building    SET (nptr) OF POINTER;
        CL_Building     POINTER;
};

LINKS

L_Building =
{       PARENT.FIELD = Building_Index.PL_Building;
        CHILD.FIELD = Building_List.CL_Building;
};
```

Figure 3.12 MRS Conceptual-to-Internal Mapping of the Professor Record Type

### 3.3.3 Operation Expanders

An *operation expander* is a procedure that maps an operation on an abstract file to a sequence of operations on concrete files, as prescribed by an elementary transformation. Expander mappings do *not* rely on how concrete operations are realized at lower levels; they express operation mappings that are *independent* of the elementary transformations which may be used to materialize concrete files and their operations. As we noted in Section 2.3, this is a simple requirement to meet. Moreover, it is this independence which permits different DBMS storage architectures to be constructed from a common pool of software components.

Again consider MRS. The MRS_INDEX transformation maps an abstract file to an inverted file (i.e., a concrete data file and concrete index files). Operations on inverted files are well-understood. When abstract records are retrieved their corresponding (dominant) data records are retrieved, along with zero or more index records. (The index records were used to locate the data records). When an abstract record is updated or deleted, its corresponding data record is updated or deleted, in addition to the modification of affected index records.

The MRS_DIVIDE operation mappings are also straightforward. Retrieving an abstract index record involves the retrieval of its (dominant) index record and the following of a linear-list linkset to access all of its list records. By concatenating these records, the abstract index record is reconstructed. An update or deletion of an abstract index record requires the update or deletion of its corresponding index and list records.

The INTERNAL transformation translates operations on files directly to operations on JUPITER files. As such, operations on internal files can be considered primitives. (Actually, JUPITER has a slightly more primitive set of operations than the operations listed in Section 3.2.2.2. INTERNAL algorithms perform this translation).

In the remainder of this section, we show how two simple operations - MAKE_FILE_CURSOR and DROP_FILE_CURSOR - are mapped by expanders. Operation mappings for the more complicated operation of record retrieval (RET and ADV) are presented in Appendix II.

- We will use the following notation to express the mappings of operation expanders:

$\text{OPERATION}_{\text{transformation}} \rightarrow$

OPERATION-1;
OPERATION-2;
...
OPERATION-n;

The operation to the left of → is an operation on an abstract file. Its subscript indicates which elementary transformation produces this mapping. The sequence of operations to the right of → are operations on concrete files and have unspecified transformation subscripts (to denote no reliance on how concrete operations are implemented at lower levels).

Consider the MAKE_FILE_CURSOR operation and the MRS_INDEX transformation. Suppose that at most two cursors are needed at any one time to process operations on an inverted file; one is permanently assigned to the data file and the other is reassignable to any of the index files. It follows that when a cursor is created for the abstract file, a data file cursor and an index file cursor are also created. The mapping for this operation is:

$\text{MAKE\_FILE\_CURSOR}_{\text{MRS\_INDEX}}(\text{ A\_cursor, A\_mtid }) \rightarrow$

MAKE_FILE_CURSOR( D_cursor, D_mtid );
MAKE_FILE_CURSOR( X_cursor, X_mtid );

A_mtid and D_mtid denote the handles of the abstract file and the data file, and X_mtid is the handle of one of the index files. (Note that the mapping table MT provides the means to determine D_mtid and X_mtid from A_mtid).

Although it is not shown in this mapping, D_cursor and X_cursor are actually linked as 'subcursors' to A_cursor. Subsequent transformations may create subcursors to D_cursor and X_cursor, giving rise to a tree of cursors rooted at A_cursor. The leaves of the tree are cursors on internal (JUPITER) files. With this construction, it is possible to find the concrete cursors of any abstract cursor quickly. We show an example cursor tree at the end of the next section (Grand Central).

In a similar manner, DROP_FILE_CURSOR has the expansion:

```
DROP_FILE_CURSOR$_{\text{MRS\_INDEX}}$( A_cursor ) $\rightarrow$

    DROP_FILE_CURSOR( D_cursor );
    DROP_FILE_CURSOR( X_cursor );
```

Now consider the mapping of these operations by MRS_DIVIDE. Creating (dropping) a cursor on an abstract index file requires the creation (deletion) of cursors on the corresponding concrete index and concrete list files. Thus, MAKE_FILE_CURSOR and DROP_FILE_CURSOR have simple expansions:

```
MAKE_FILE_CURSOR$_{\text{MRS\_DIVISION}}$( X_cursor, X_mtid ) $\rightarrow$

    MAKE_FILE_CURSOR( I_cursor, I_mtid );
    MAKE_FILE_CURSOR( L_cursor, L_mtid );

DROP_FILE_CURSOR$_{\text{MRS\_DIVISION}}$( X_cursor ) $\rightarrow$

    DROP_FILE_CURSOR( I_cursor );
    DROP_FILE_CURSOR( L_cursor );
```

X_mtid is the handle of the abstract index file, and I_mtid and L_mtid are the handles of the index and list files.

As mentioned earlier, MAKE_FILE_CURSOR$_{\text{INTERNAL}}$, DROP_FILE_CURSOR$_{\text{INTERNAL}}$, and other INTERNAL operations can be considered primitives which create and destroy JUPITER cursors.

Expanders for other operations and elementary transformations can be written in a straightforward manner (see Appendix II).

### 3.3.4 Grand Central

Conceptual-to-internal mappings of operations are realized as a composition of abstract-to-concrete mappings of operations. This composition is accomplished at run-time by routing the output of one expander layer to the input of another. The mapping table (MT) contains the routing information. The routing procedures are collectively called *Grand Central.* A more advanced method of composition is discussed in Section 4.3.

• Grand Central is a set of procedures, one procedure for each file operation. Each procedure is a case statement, where the case indicator identifies an elementary transformation. The number of cases equals the number of expander layers that are needed for a specific architecture. Suppose an abstract file A has the handle A_mtid. Let MT[A_mtid].xform be the identifier of the elementary transformation which maps A to its concrete counterparts. (MT.xform is the column of the mapping table MT which contains this identifier). It follows that the Grand Central procedures for MAKE_FILE_CURSOR and RET look like:

```
MAKE_FILE_CURSOR( A_cursor, A_mtid )
{
    switch( MT[ A_mtid ].xform )
    {
    case MRS_INDEX :
        MAKE_FILE_CURSOR$_{\text{MRS\_INDEX}}$( A_cursor, A_mtid );

    case MRS_DIVIDE :
            MAKE_FILE_CURSOR$_{MRS\_DIVIDE}$( A_cursor, A_mtid );

        case INTERNAL :
            MAKE_FILE_CURSOR$_{INTERNAL}$( A_cursor, A_mtid );
        };
};


RET( A_cursor, A_query, A_into_list, position)
{
        switch( MT[ A_cursor.mtid ].xform )
        {
        case MRS_INDEX :
            RET$_{MRS\_INDEX}$( A_cursor, A_query, A_into_list, position);

        case MRS_DIVIDE :
            RET$_{MRS\_DIVIDE}$( A_cursor,  A_query, A_into_list, position);

        case INTERNAL :
            RET$_{INTERNAL}$( A_cursor, A_query, A_into_list, position);
        };
};
```

Procedures for other operations follow the same pattern.

Here's how Grand Central works. Each procedure of Grand Central directs the expansion of an operation on abstract files. If an abstract file is materialized by the MRS_INDEX transformation, then the RET, ADV, etc. operations on the abstract file would be translated into the $\text{RET}_{\text{MRS\_INDEX}}$, $\text{ADV}_{\text{MRS\_INDEX}}$, etc. operations on concrete files. In general, if an abstract file is materialized by transformation T, the abstract operation O on this file is mapped to operation $O_T$ by Grand Central.

Users issue operations on conceptual files in the MRS architecture. These operations are translated by MRS_INDEX mappings to operations on abstract index files and data files. Operations on abstract index files are translated by MRS_DIVIDE mappings to operations on index and list files. Operations on data, index, and list files are translated by INTERNAL to operations on JUPITER files. All of these translations are directed by Grand Central.

As mentioned in Section 3.3.3 (Operation Expanders), a tree of file cursors is created when a cursor on a conceptual file in an MRS database is created. Figure 3.13 shows this tree. Again, it is a result of composing $\text{MAKE\_FILE\_CURSOR}_{\text{MRS\_INDEX}}$, $\text{MAKE\_FILE\_CURSOR}_{\text{MRS\_DIVIDE}}$, and $\text{MAKE\_FILE\_CURSOR}_{\text{INTERNAL}}$. As mentioned earlier, the leaves of the cursor tree are JUPITER cursors (i.e., cursors on INTERNAL files).

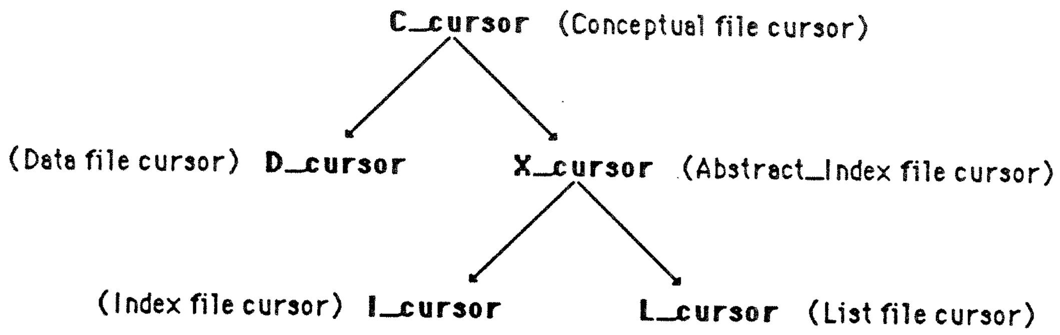

C_cursor (Conceptual file cursor)
(Data file cursor) D_cursor
X_cursor (Abstract_Index file cursor)
(Index file cursor) I_cursor
L_cursor (List file cursor)

Figure 3.13 A Cursor Tree for the MRS Storage Architecture

## 4. Future Research

Reconfigurable database systems require a technology that will take years to perfect. We have taken the first steps to consolidate theoretical results and practical achievements. The next steps will require basic research both in databases and software development.

Among the outstanding database issues that remain to be incorporated into the prototype are concurrency control, query processing, and customized end-user interfaces. Although we are investigating all three, we discuss the first two in Sections 4.1 and 4.2 to show how layering forces a novel interpretation and generalization of existing results.

Customizing end-user interfaces is an essential feature of reconfigurable DBMSs. It can take the form of allowing alternative data models and data languages, allowing new data types and operations to be defined, or both. A considerable amount of research on these topics is already available ([Shu77], [Hou77], [Tay79], [Su81], [Dem85], [Sto83], [Car84], [Day84], [Lor85], [Sto85]). Our preliminary investigations suggest that existing results can be accommodated into our framework in a straightforward manner. For this reason, we direct readers to these references.

From the side of software development, a technology is needed to compose layers of software at compile time (not at run-time as we are now doing). Compile-time composition has the potential of eliminating unneeded generality in expanders through code simplification. Furthermore, there is the possibility of applying our technology to other areas of software development. These topics are the subject of Section 4.3.

*As before, the basic ideas of each section are summarized in the introductory paragraphs. Additional details follow the bullet ( • ) marker.*

### 4.1 Concurrency Control

Concurrency control is an integral component of multiuser database systems. The most common method to achieve concurrency control is locking; i.e., a transaction can access a data item only if it holds an appropriate lock on that item ([Kor86]). As a general rule, locking protocols have been developed for reading and writing objects that are visible through a single interface. Reconfigurable database technology requires more sophistication because a series of interfaces are crossed in conceptual-to-internal mappings. At each interface, objects in addition to conceptual objects may appear (e.g., indices, fragment files, etc.), and protocols for reading and writing them are needed. It is often the case that protocols, such as two-phase locking (2PL), which are appropriate at the conceptual level may not be optimal at the internal level.

An accurate way to characterize the locking protocol of a DBMS is by a composition of the protocols that are applied at each layer in its storage architecture. This concept is illustrated below.

• Placing and releasing locks on records are special database operations. Consequently, they are subject to mappings. It is not unusual, for example, that locking a conceptual record may generate locks on one or more internal records. A simple set of rules appears to govern the mapping of record locks, so that consistency at a higher level is preserved at lower levels ([Cul85]):

(1) A lock on an abstract record is always mapped to a lock on its dominant concrete record.

(2) A record lock becomes a page lock if mapped by the internal or layering transformation.

(3) Secondary records created by division and segmentation are indirectly locked by locking their primary record. That is, locking the primary path to secondary records prevents them from being accessed by others.

Note that these rules do not say *when* locks should be placed or released (this is the responsibility of the locking protocol), but rather *how* a lock on an abstract object is inherited by concrete objects. Thus we distinguish between locking protocols and lock mappings.

Lock mapping in MRS occurs in the following way (see Fig. 4.1). A lock on a CONCEPTUAL record becomes a lock on its DATA record by rule (1). A lock on an ABSTRACT_INDEX record becomes a lock on its INDEX record, again by rule (1). LIST records need not be locked, even if they are read or updated, by rule (3). Locks on DATA and INDEX records are locks on internal records. Lock mappings can either stop here, or they can be translated into page locks by rule (2). (This means that two different versions of the MRS architecture could be constructed, each with a different lock granularity, i.e., records vs. pages).

MRS uses 2PL for concurrency control at the conceptual level. However, a non-2PL protocol is used for locking ABSTRACT_INDEX records and their corresponding INDEX records. As mentioned above, LIST records are not locked at all. Thus, if the locking protocol of MRS were explained solely in terms of internal records, it would be an odd mixture of rules which would not correspond at all with recognized protocols. Viewed in a layered manner, where the semantics of the objects that are being locked are taken into account, the MRS protocol can be easily seen as a composition of familiar results.

A reconfigurable DBMS could provide a selection of protocols at any given layer. System R, for example, has a software layer similar to MRS_INDEX. However, it uses 2PL to lock ABSTRACT_INDEX records. IDMS uses a variant of 2PL at the conceptual level.

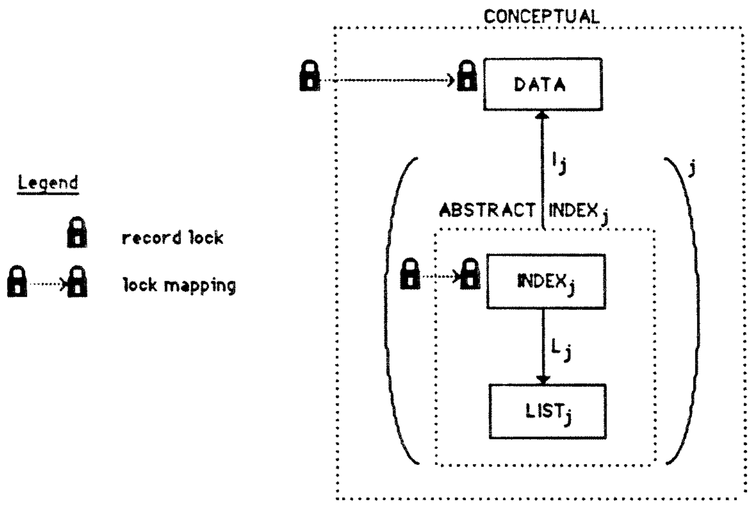


Figure 4.1 Conceptual-to-Internal Lock Mappings in MRS

It appears possible to integrate concurrency control into expander layers, so that multiuser DBMSs can be constructed from components just as easily as that for single user systems. A general theory of lock mappings will be needed to provide the required groundwork.

## 4.2 Query Processing

Query processing is a fundamental area of database research. The seminal results have concentrated on relational DBMSs (e.g., [Sel79], [Won76]). Extensions have generalized their scope so that query processing in network, hierarchical, and relational databases are subsumed as special cases (e.g., [Che84], [Wha83], [Ros82]). Current research addresses the challenges of non-1NF relations, abstract data types, complex objects, and recursive queries (e.g., [Rot85], [Sto85], [Lor85], [Ull85]).

Reconfigurable DBMS technology requires query processing algorithms to be consolidated and recast into a layered framework. This calls for known algorithms to be decomposed into layers, so that each processing strategy and optimization strategy is localized to a specific layer. Thus, a query processing algorithm which covers multiple layers would be reconstructed by the composition of the optimizations and strategies of the layers that it references.

Queries are usually posed at the external level; it is the responsibility of the DBMS to map queries to the internal level. A layered model of query processing must account for *external-to-internal* mappings in a unified way. An example of layered decomposition is given below.

• Consider the conceptual Employee and Department record types (relations) which both have indices on the Dept field:

```
TYPE     Employee =
{
     E#        INT                      primary_key;
     Ename     SET (10) OF CHAR;
     Salary    FLOAT;
     Dept      SET (8) OF CHAR          indexed;
};

TYPE     Department =
{
     D#        INT                      primary_key;
     Dept      STRING (8) OF CHAR       indexed;
     Building  STRING (10) OF CHAR;
     City      STRING (10) OF CHAR;
};
```

A view file (relation) of the equi-join of Employee and Department is declared in SQL as:

```
DEFINE  VIEW  Empdept  AS
SELECT   E#, Ename, Salary, Employee.Dept, D#, Building, City
FROM     Employee, Department
WHERE    Employee.Dept = Department.Dept
```

The query to retrieve employee names and department numbers of employees who earn more than $30,000 and work in Denver is expressed through this view as:

```
SELECT   Ename, D#
FROM     Empdept
WHERE    Salary > 30000  AND  City = 'Denver'
```

One way that this query could be processed is by a merge-join of the Dept indices ([Bla77]). That is, a Dept index record of the Employee file is joined with a Dept index record of the Department file if both have the same Dept value. The join produces pairs of inverted lists; one list points to Employee records and the other points to Department records. An inverted list pair is processed by following the pointers of each list and applying the restriction predicates to the referenced records. The Employee and Department records that qualify are joined unconditionally, and the Ename and D# attributes are projected. The query is evaluated by processing each inverted list pair in the above manner.

Decomposing this algorithm into layers requires us to introduce a new layer to the MRS architecture and a new file operation . The new layer is called VIEW and is responsible for external-to-conceptual mappings. It resides on top of MRS_INDEX and incorporates standard query modification and query simplification algorithms to map operations on view relations to operations on conceptual relations.[2] Figure 4.2 is a dsd that shows the *external-to-internal* data mappings of the Empdept view relation.

The new file operation is called JOIN. It binds a cursor to a 'view' file which is produced by the join of two concrete files. A parametric definition of JOIN is given in Figure 4.3. A cursor initialized by JOIN can be

[2] The implementation of the VIEW layer may be different than other MRS layers. A preprocessor can accomplish the external-to-conceptual mappings for embedded DML statements at program compile time. In such cases, the VIEW 'layer' is actually in the preprocessor expansions. For high-level query languages, the VIEW expansions are done at run-time. The VIEW layer would then constitute part of the query language subsystem of the DBMS.

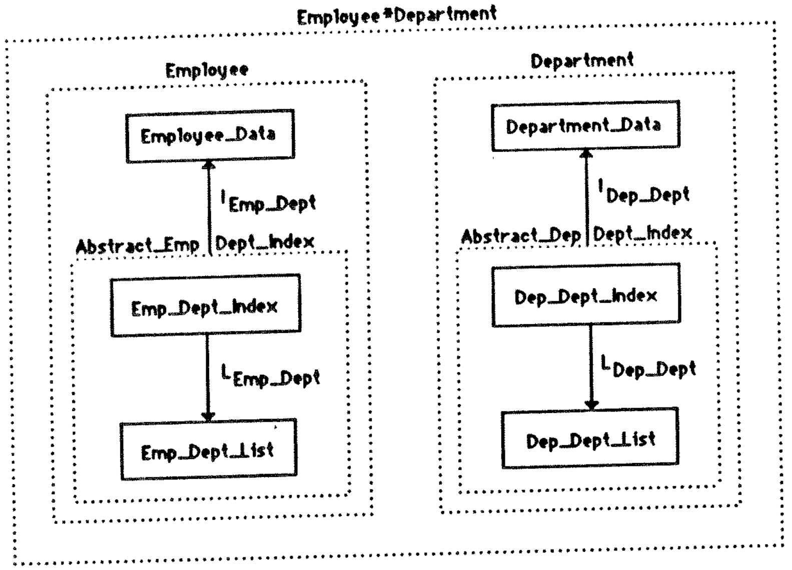


Figure 4.2 External-to-Internal Mapping of Data in MRS

Operation:

JOIN( J_cursor,
mtid$_1$, query$_1$, into_list$_1$,
mtid$_2$, query$_2$, into_list$_2$,
join_clause, position )

Semantics:

join concrete file mtid$_1$ with concrete file mtid$_2$.
J_cursor is a cursor over the records of the joined files.
query$_i$ and into_list$_i$ are the qualification predicate
and field-buffer pairs for data input and output for file mtid$_i$
join_clause gives the joining condition, and
position specifies initial positioning of J_cursor
(before_first or at_first record).

Figure 4.3 A Parametric Definition of the JOIN Operation

(a) operation at the external level:

RET$_{VIEW}$( V_cursor, 'Salary > 30,000 and City = Denver', V_into_list, position )

(b) RET$_{VIEW}$ expansion:

JOIN$_{MRS\_INDEX}$( J_cursor,
Employee_mtid, 'Salary > 30000', Employee_into_list,
Department_mtid, 'City = Denver', Department_into_list,
'Employee.Dept = Department.Dept', position )

(c) JOIN$_{MRS\_INDEX}$ expansion:

JOIN$_{MRS\_DIVIDE}$( JAX_cursor,
Abstract_Emp_Dept_Index_mtid, null, Abs_Emp_Dept_Index_into_list,
Abstract_Dep_Dept_Index_mtid, null, Abs_Dep_Dept_Index_into_list,
'Abstract_Emp_Dept_Index.Dept = Abstract_Dep_Dept_Index.Dept', position );

INIT$_{INTERNAL}$( Employee_Data_cursor, 'Salary > 30000', Employee_Data_into_list );
INIT$_{INTERNAL}$( Department_Data_cursor, 'City = Denver', Department_Data_into_list );

(d) JOIN$_{MRS\_DIVIDE}$ expansion:

JOIN$_{INTERNAL}$( JX_cursor,
Emp_Dept_Index_mtid, null, Emp_Dept_Index_into_list,
Dep_Dept_Index_mtid, null, Dep_Dept_Index_into'list,
'Emp_Dept_Index.Dept = Dep_Dept_Index.Dept', position );

INIT$_{INTERNAL}$( Emp_Dept_List_cursor, null, Emp_Dept_List_into_list );
INIT$_{INTERNAL}$( Dep_Dept_List_cursor, null, Dep_Dept_List_into_list );

Figure 4.4 External-To-Internal Mapping of JOINs in MRS

ADVanced, in the same way as RET, to read records of a view file one at a time.

Figure 4.4 shows how the RET operation at the external level is mapped to operations at the internal level. Each step in the mapping is highlighted below.

1) *External Level.* A user initiates a RET on the Empdept view relation. Grand Central translates this operation into $RET_{VIEW}$ (Fig. 4.4a).
2) *External-to-Conceptual Mappings.* VIEW maps the RET on Empdept relation to a JOIN of the Employee and Department files (Fig. 4.4b). The mapping is accomplished by view substitution algorithms.
3) *Conceptual-to-Inverted_File Mappings.* MRS_INDEX maps the JOIN of Employee and Department to a JOIN of the abstract Dept index files, and an INITialization of the cursors on the internal Employee_Data and internal Department_Data files (Fig. 4.4c). The data file cursors are used for following pointers of inverted lists and applying restriction predicates to data file records. It is in MRS_INDEX where the algorithm to process inverted list pairs, which was described above, resides.
4) *Abstract_Index-to-Internal Mappings.* MRS_DIVIDE maps the JOIN of the abstract Dept index files to a JOIN of the internal index files, and an INITialization of cursors to the internal list files (Fig. 4.4d). The INITed cursors are used for pointer following to access list file records. It is in MRS_DIVIDE that inverted lists are reconstructed from their fragments.
5) *Internal Level.* INTERNAL materializes the JOIN of the internal index files by the merge-join algorithm.

Once an external cursor has been bound to the view file (which is what the above mappings accomplish), view records are read one at a time by ADVancing the external cursor. The mapping of the external ADV operation is straightforward and is not presented here.

As indicated earlier, there are many other ways of processing this query; each would have its own distinctive external-to-internal mappings. Although it may not be obvious from this example, specific strategies and optimization decisions can be identified with each layer. The VIEW layer, for example, determines the ordering in which conceptual files are to be joined. (No decision was necessary in our example). The MRS_INDEX layer determines whether or not indices should be used. (An alternative strategy to the one considered above would be to ignore indices altogether and to perform a JOIN on the Employee_Data and Department_Data files directly. A JOIN of the conceptual files would still result, but perhaps at a cost of lower performance.) Some layers, such as MRS_DIVIDE, don't involve optimization of JOINs as there is only one obvious strategy to pursue.

A layered model of query processing, based on the above approach, is forthcoming.

## 4.3 Software Development Technologies

If one compares the code that GENESIS produces with the actual hand-written code of MRS, one finds substantial differences. GENESIS code has a considerable amount of layering between the conceptual and internal levels, whereas hand-written code has minimal layering. GENESIS code relies heavily on intermediate results (such as cursors to ABSTRACT_INDEX files in Figure 3.13), whereas hand-written code does not. GENESIS layers (expanders) embody generalized algorithms, whereas hand-written code embodies subalgorithms.[3] The overhead of extra layering, materialization of intermediate results, and algorithm generality causes GENESIS code to be more verbose and to run slower than hand-written code. Despite these differences, both accomplish exactly the same mappings.

The drawbacks of GENESIS-produced code are not intrinsic to reconfigurable DBMS technology; they are actually a consequence of composing modules at run-time. Compile-time composition of modules offers a

[3] The MRS_INDEX module in the GENESIS prototype, for example, handles the inversion of both scalar and repeating fields. Although repeating fields in conceptual records are excluded from the MRS architecture, they may be featured in other architectures. The extra generality enables the MRS_INDEX module to be used in the construction of diverse architectures.

means by which these problems can be eliminated.

- Hand-written code expresses a compact form of conceptual-to-internal mappings. We are convinced that programmers are mentally composing the mappings of expanders and simplifying the resulting compositions without realizing the layering processes that are involved. Preliminary investigations support the hypothesis that just as arithmetic equations can be expanded and simplified by plugging in definitions of lower-level functions and collecting terms, so too can efficient code - resembling hand written - be produced by macro-expanding operation expanders and then simplifying. Obvious simplification techniques are the elimination of dead code and the replacement of unnecessary computations with their resulting values. Such techiques have been used optimizing compilers ([Lov76], [Car77], [Sch77]).

Compile-time composition and simplification of software modules seems destined to become an important technology for software development. Libraries of prewritten and general-purpose modules can be tapped by software designers to eliminate the tedium and cost of writing code for well-understood routines. Compile-time composition of referenced modules will enable customized software to be developed rapidly and the code which is produced to be efficient. Thus, the utility of a software composition technology will not be limited to the construction of database management systems. However, the same can be said for the technology on which GENESIS rests.

Data structures are main-memory storage structures. Preliminary investigations reveal that layering occurs in data structures much in the same way as it does in DBMS storage architectures. Thus, there should be data structure counterparts to the UM and TM. As data structures are used in virtually every major software application (of which DBMSs are a small subset), it seems reasonable to believe that a companion technology to GENESIS can be devised to handle a much broader scope of software development problems than we are presently considering.

The software composition technology and the reconfigurable DBMS technology are complimentary and independent. It is our belief that the impact of one on commercial software systems development is likely to be strongly influenced by the maturity of the other. We have begun to explore these more general technologies.

## 5. Conclusions

Database systems have largely been built in an ad-hoc manner. Typical DBMS software is monolithic; to affect modifications and upgrades is either very costly or impractical. Constructing DBMSs has been more of an art practiced by few, rather than a science understood by many. This will not always be the case.

We have developed a novel, yet simple, technology which enables customized DBMSs to be produced cheaply and quickly. The technology for reconfiguring DBMSs is based on theoretical models ([Bat82], [Bat85a]) which 1) identified basic building blocks of DBMS software, 2) revealed that all building blocks could be referenced through a common interface, and 3) showed that operational DBMSs could be described by compositions of these building blocks. GENESIS, an operational prototype of a reconfigurable DBMS, is an implementation of these ideas.

The merit of our research lies in both its practical and theoretical contributions. From a practical perspective, once a storage architecture has been designed, the software to support it i can be produced rapidly. If a prototype architecture doesn't work out, it can be changed easily. Both capabilities are in stark contrast to the present state-of-the-art where the exact opposite is true.

Another practical advantage is the method by which basic software components of DBMSs can be developed. Each component (e.g., an expander-transformer pair) describes simple mappings and can be coded and debugged in isolation. This contrasts with current methods of DBMS software construction where many of our components (layers) are fused and therefore must be debugged simultaneously.

From a theoretical perspective, our research points a way to which disparate academic results and practical achievements on database implementation (i.e., storage structures, query processing, concurrency control, etc.) can be unified. Furthermore, we see the potential of our technology to be generalized to areas of software development beyond DBMSs.

We believe that the development of a reconfigurable DBMS technology will result in a fundamental advance in understanding and simplifying the construction of database systems.

*Acknowledgements.* We thank Hank Korth and Arthur Keller for their comments on an earlier draft of this paper.

## 6. References

[And76] H.D. Anderson and P.B. Berra, 'Minimum Cost Selection of Secondary Indices for Formatted Files', *ACM Trans. Database Syst.* 2,1 (March 1977), 68-90.

[Ast76] M.M Astrahan, et al., 'System R: A Relational Approach to Database Management', *ACM Trans. Database Syst.* 1,2 (June 1976), 97-137.

[Bat82] D.S. Batory and C.C. Gotlieb, 'A Unifying Model of Physical Databases', *ACM Trans. Database Syst.* 7,4 (Dec. 1982), 509-539.

[Bat84a] D.S. Batory, 'Conceptual-To-Internal Mappings in Commercial Database Systems', *ACM PODS 1984*, 70-78.

[Bat84b] D.S. Batory, 'Notes on Commercial DBMS Architectures', unpublished manuscript, 1984.

[Bat85a] D.S. Batory, 'Modeling the Storage Architectures of Commercial Database Systems', *ACM Trans. Database Syst.*, 10 #4, (Dec. 1985), 463-528.

[Bat85b] D.S. Batory, 'Progress Toward Automating the Development of Database System Software', *Query Processing in Database Systems*, Springer-Verlag 1985, W. Kim, D. Reiner, and D.S. Batory (ed), 261-296.

[Bla77] M.W. Blasgen and K.P. Eswaren, 'Storage and Access in Relational Database Systems', *IBM Syst. Jour.* 16,4 (1977), 363-377.

[Car77] J.L. Carter, 'A Case Study of a New Code Generation Technique for Compilers', *CACM* 20,12 (Dec. 1977), 914-920.

[Car84] M. Carey and D. DeWitt, 'Extensible Database Systems', *Islamorda Workshop on Large Scale Knowledge Base and Reasoning Systems*, Feb. 1985, 335-352.

[Cas86] I.R. Casas, 'Performance Prediction of Data Base Systems', Ph.D. Diss., Department of Computer Science, University of Toronto, 1986.

[Cha76] D.D. Chamberlin, et al., 'SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control', *IBM Jour. Res. and Dev.* 20,6 (Nov. 1976), 560-575.

[Che84] H. Chen and S.M. Kuck, 'Combining Relational and Network Retrieval Methods', *ACM SIGMOD 1984*, 131-142.

[Cul85] R.E. Culler, 'Locking and the Transformation Model', M.Sc. Report, Dept. Computer Sciences, The University of Texas, Austin, Texas, 1985.

[Dat82] C.J. Date, *An Introduction to Database Systems*, Addison-Wesley, 1982.

[Day84] U. Dayal and J. Smith, 'PROBE: A Knowledge-Oriented Database Management System', *Islamorda Workshop on Large Scale Knowledge Base and Reasoning Systems*, Feb. 1985, 103-138.

[Dem85] G.B. Demo and S. Kundu, 'Analysis of the Context Dependency of CODASYL Find-Statements with Application to Database Program Conversion', *ACM SIGMOD 1985*, 354-363.

[Eff84] W. Effelsberg and T. Haerder, 'Principles of Database Buffer Management', *ACM Trans. Database Syst.* 9,4 (Dec. 1984), 560-595.

[Elh84] K. Elhard and R. Bayer, 'A Database Cache for High Performance and Fast Restart in Database Systems', *ACM Trans. Database Syst.* 9,4 (Dec. 1984), 503-525.

[Fal85] C. Faloutsos, 'Signature Files: Design and Performance Comparison of some Signature Extraction Methods', *ACM SIGMOD 1985*, 63-83.

[Far75] J.H.G. Farley and S.A. Schuster, 'Query Execution and Index Selection for Relational Databases', CSRG Tech. Rep. 53, University of Toronto, 1975.

[Gar85] J.F. Garza, 'Design and Implementation of Jupiter: A General File Management System', M.Sc. Thesis, Dept. Computer Sciences, University of Texas, Austin, Texas, 1985.

[Gra78] J. Gray, 'Notes on Database Operating Systems', research rep. RJ2188, IBM Laboratory, San Jose, 1978.

[Has82] R.L. Haskin and R.A. Lorie, 'On Extending the Functions of a Relational Database System', *ACM SIGMOD 1982*, 207-212.

[Hou77] B.C. Housel, 'A Unified Approach to Program and Data Conversion', *VLDB 1977*, 327-335.

[IEE82] *IEEE Database Engineering*, Design Data Management, R. Katz (ed), June 1982.

[IEE83] *IEEE Database Engineering*, Expert Systems and Database Systems, W. Kim (ed), December 1983.

[IEE84a] *IEEE Database Engineering*, Statistical Database Management, D. Batory (ed), March 1984.

[IEE84b] *IEEE Database Engineering*, Engineering Database Management, R. Katz (ed), June 1984.

[Jae82] G. Jaesche and H. Schek, 'Remarks on the Algebra of Non First Normal Form Relations', *ACM PODS 1982*, 124-138.

[Joh83] H.R. Johnson, J.E. Schweitzer, and E.R. Warkentine, 'A DBMS facility for Handling Structured Engineering Entities', *ACM Database Week: Engineering Design Applications 1983*, 3-12.

[Ker78] B.W. Kernighan and D.M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, 1978.

[Kor86] H.F. Korth and A. Silberschatz, *Database System Concepts*, McGraw-Hill, 1986.

[Kor79] J.Z. Kornatowski, 'The MRS User's Manual', Computer Systems Research Group, University of Toronto, 1979.

[Lor77] R.A. Lorie, 'Physical Integrity in a Large Segmented Database', *ACM Trans. Database Syst.* 2,1 (March 77), 91-104.

[Lor85] R.A. Lorie, et al., 'Supporting Complex Objects in a Relational System for Engineering Databases', in *Query Processing in Database Systems*, W. Kim, D.S. Reiner, and D.S. Batory (ed), Springer-Verlag, 1985, 145-155.

[Lov76] D.B. Loveman, 'Program Improvement by Source to Source Transformation', *JACM* 24, 1 (Jan. 1977), 121-145.

[Mar81] S.T. March, D.G. Severance, and M. Wilens, 'Frame Memory: A Storage Architecture to Support Rapid Design and Implementation of Efficient Databases', *ACM Trans. Database Syst.* 6,3 (Sept.

1981), 441-463.

[Osz85] Z.M. Ozsoyoglu and L-Y. Yuan , 'A Normal Form for Nested Relations', *ACM PODS 1985*, 251-260.

[Ros82] A. Rosenthal and D. Reiner, 'An Architecture for Query Optimization', *ACM SIGMOD 1982*, 246-255.

[Rot84] M.A. Roth, H.F. Korth, and A. Silberschatz, 'Theory of Non-First-Normal-Form Relational Databases', Tech. Rep. 84-36, Department of Computer Sciences, The University of Texas at Austin, 1984.

[Rot85] M.A. Roth, H.F. Korth, and D.S. Batory, 'The SQL/NF Query Language', Department of Computer Sciences, The University of Texas at Austin, 1985.

[Sch77] R.W. Scheifler, 'An Analysis of Inline Substitution for a Structured Programming Languages' *CACM* 20,9 (Sept. 1977), 647-654.

[Sel79] P.G. Selinger, et al., 'Access Path Selection in a Relational Database Management System', *ACM SIGMOD 1979*, 23-34.

[Smi85] K. Smith, 'Design and Implementation of the GENESIS Record Manager', M.Sc. Thesis, Dept. Computer Sciences, The University of Texas, Austin, Texas, 1985.

[Sto76] M. Stonebraker, E. Wong, P. Kreps, and G. Held, 'The Design and Implementation of INGRES', *ACM Trans. Database Syst.* 1,3 (Sept. 1976), 189-222.

[Sto82] M. Stonebraker, et al., 'Document Processing in a Relational Database System', Report UCB/ERL M82/32, Electronics Research Laboratory, University of California, Berkeley, 1982.

[Sto83] M. Stonebraker, B. Rubenstein, and A. Guttman, 'Application of Abstract Data Types and Abstract Indices to CAD Data Bases', *ACM Database Week: Engineering Design Applications 1983*, 107-114.

[Sto85] M. Stonebraker, 'Inclusion of New Types in Relational Data Base Systems', Report UCB/ERL M85/67, Electronics Research Laboratory, University of California, Berkeley, 1985.

[Sto86] M. Stonebraker and L. Rowe, 'The Design of POSTGRES', *ACM SIGMOD 1986*, to appear.

[Shu77] N.C. Shu, et al., 'EXPRESS: A Data Extraction Processing and Restructuring System', *ACM Trans. Database Syst.*, 2 #2, (June 1977), 134-174.

[Su81] S.Y.W. Su, H. Lam, and D.H. Lo, 'Transformation of Data Traversals and Operations in Application Programs to Account for Semantic Changes of Databases', *ACM Trans. Database Syst.*, 6 #2, (June 1981), 255-294.

[Tay79] R. Taylor, et al., 'Database Program Conversion: A Framework for Research', *VLDB 1979*, 299-312.

[Tsi77] D.C. Tsichritzis and F. Lochovsky, *Data Base Management Systems*, Academic Press, New York, 1977.

[Tsi78] D.C. Tsichritzis and A. Klug, 'The ANSI/X3/SPARC DBMS Framework: Report of the Study Group on Database Management Systems', *Info. Syst.*, 1978, Vol 3, 173-191.

[Tsu85] K. Tsukuda, 'Mapping of Record Types in the GENESIS Database Management System', M.Sc.

Thesis, Dept. Computer Sciences, The University of Texas, Austin, Texas, 1985.

[Tur79] M.J. Turner, R. Hammond, and P. Cotton, 'A DBMS for Large Statistical Databases', *VLDB 1979*, 319-327.

[Wie83] G. Wiederhold, *Database Design*, McGraw-Hill, 1983.

[Wha83] K-Y. Whang, 'A Physical Database Design Methodology Using the Property of Separability', Ph.D. Diss., Dept. Computer Science, Stanford University, Stanford, California, 1983.

[Won76] E. Wong and K. Youssefi, 'Decomposition - A Strategy for Query Processing', *ACM Trans. Database Syst.* 1,3 (Sept. 1976), 223-241.

[Ull85] J.D. Ullman, 'Implementation of Logical Query Languages for Databases', *ACM Trans. Database Syst.*, 10 #3, (Sept. 1985), 289-321.

## Appendix I. Storage Architecture Tables

The five most important storage architecture tables are the field definition table (FDT), the mapping table (MT), the volume table (VT), the file implementation table (FIT), and the Jupiter file table (JFT). There are other tables, but they are not relevant here. In the following paragraphs, we show the contents of five tables for the MRS schema of Figures 3.2 and 3.12 after it has been processed by the architecture program of Figure 3.11.

Field definitions are stored in the FDT. Each row of the FDT of Figure A1 describes a single field by its option flags, type, bounds (if an array or repeating group), the row of the FDT of its first subfield, the number of subfields, and the name of the field itself. Row 0 always contains the name of the conceptual schema. The rows whose type is 'FILE' contain record type definitions. For example, the Professor record type is stored in row 1 and the Pname_Abstract_Index record type is stored in row 8. Rows 0-6 were entered by the DDL compiler when the MRS schema of Figure 3.2 was processed. Rows 7-13 were added by the MRS_INDEX transformer. Rows 14-20 and 21-27 were added in successive executions of the MRS_DIVIDE transformer.

Files are defined in the MT. Each row of the MT of Figure A2 describes a file by its name, the row of the FDT at which its record type is stored, the identifier of the elementary transformation that maps it to its concrete counterparts, the row of its dominant concrete file, the number of concrete files to which it is mapped, and the row of the JFT specifying where records of this file are stored (applicable if the file is internal). For example, the Professor file (row 0) is mapped by the MRS_INDEX transformation to the Professor_Data_File (row 1) and the abstract index files for Pname and Building (rows 2 and 3).

The tables of Figure A3 are relevant to JUPITER. MRS databases are stored in a single volume. The VT of Figure A3 contains only one volume definition ('mrs_vol'). Two different file structures are used by MRS: unordered and B+ trees. They are defined in the FIT. The MRS database of Figure 3.2 generates five internal files: two index files, two list files, and one data file. They are defined in consecutive rows of the JFT.

## Appendix II. RET and ADV Operation Expanders for the MRS Architecture

Consider the MRS_INDEX transformation and its mapping of the RET and ADV operations. For purposes of simplicity, we will assume that queries are single clauses of the form (field = value). Algorithms for processing/mapping more complicated queries are given in [Sel79], [And76], and [Far75].

When RET prepares an abstract cursor A_cursor for retrieval, a strategy for processing the specified query is decided. The data file is scanned if there is no index to process the query, else the index record identified by the query is retrieved and the pointers of its inverted list are followed. This strategy is expressed by:

```
RET_MRS_INDEX( A_cursor, A_query, A_into_list, position ) →

        if A_query can be processed by indices then
        {
(1)             X_into_list := ( inverted_list_trace, inverted_list_buffer );
(2)             RET( X_cursor,  A_query, X_into_list, at_first );
(3)             INIT( D_cursor, null, A_into_list );
        }
        else   /* scan data file */
        {
(4)             RET( D_cursor, A_query, A_into_list, before_first );
        };
(5)     if position = at_first then ADV_MRS_INDEX( A_cursor );
```

Lines (1-3) deal with the case of processing A_query using an index. At line (1), 'inverted_list_buffer' is declared to be the buffer that is to contain the inverted list of the retrieved index record. The record (and its inverted list) is retrieved at line (2), where the index record satisfies A_query (i.e., 'field = value'). The D_cursor is initialized for pointer following in line (3). Note that the fields to return are the same as that for the abstract file.

| fdt_id | opt_flags | type | bounds | 1st_child | n_child | name |
|---|---|---|---|---|---|---|
| 0 | | DB | | 1 | 1 | MRS_Schema |
| 1 | | FILE | | 2 | 4 | Professor |
| 2 | F1+F2 | STRING | 10 | 6 | 1 | Pname |
| 3 | | INT | | | | Office |
| 4 | F1 | STRING | 4 | 6 | 1 | Building |
| 5 | | INT | | | | Campus_Phone |
| 6 | | CHAR | | | | |
| 7 | | FILE | | 2 | 4 | Professor_Data_File |
| 8 | | FILE | | 9 | 2 | Pname_Abstract_Index |
| 9 | F1 | STRING | 10 | 6 | 1 | Pname |
| 10 | | SET | * | 11 | 1 | /* pointer array */ |
| 11 | | POINTER | | | | /* pointer type */ |
| 12 | | FILE | | 13 | 2 | Building_Abstract_Index |
| 13 | | STRING | 4 | 6 | 1 | Building |
| 14 | | FILE | | 15 | 2 | Pname_Index |
| 15 | F1 | STRING | 10 | 6 | 1 | Pname |
| 16 | | SET | 1 | 11 | 1 | /* set (1) of pointer */ |
| 17 | | POINTER | | | | /* first child pointer */ |
| 18 | | FILE | | 18 | 2 | Pname_List |
| 19 | | SET | nptr | 11 | 1 | /* set (nptr) of pointer */ |
| 20 | | POINTER | | | | /* next child pointer */ |
| 21 | | FILE | | 22 | 2 | Building_Index |
| 22 | F1 | STRING | 10 | 6 | 1 | Building |
| 23 | | SET | 1 | 11 | 1 | /* set (1) of pointer */ |
| 24 | | POINTER | | | | /* first child pointer */ |
| 25 | | FILE | | 26 | 2 | Building_List |
| 26 | | SET | nptr | 11 | 1 | /* set (nptr) of pointer */ |
| 27 | | POINTER | 0 | 0 | 0 | /* next child pointer */ |

note: F1 denotes the 'indexed' option and F2 denotes the 'primary_key' option.

Figure A1. The Field Definition Table (FDT)

| mt_id | file_name | fdt_id | xform | 1st_child | n_child | jft_id |
|---|---|---|---|---|---|---|
| 0 | Professor | 1 | MRS_INDEX | 1 | 3 | |
| 1 | Professor_Data_File | 7 | INTERNAL | | | 5 |
| 2 | Pname_Abstract_Index | 8 | MRS_DIVIDE | 4 | 2 | |
| 3 | Building_Abstract_Index | 12 | MRS_DIVIDE | 6 | 2 | |
| 4 | Pname_Index | 14 | INTERNAL | | | 1 |
| 5 | Pname_List | 18 | INTERNAL | | | 2 |
| 6 | Building_Index | 21 | INTERNAL | | | 3 |
| 7 | Building_List | 25 | INTERNAL | | | 4 |

Figure A2. The Mapping Table (MT)

| VT | | FIT | | JFT | | | |
|---|---|---|---|---|---|---|---|
| vt_id | vname | fit_id | file_type | jft_id | simple_file_name | vt_id | fit_id |
| 1 | mrs_vol | 1 | unordered | 1 | Pname_Index_SF | 1 | 2 |
| | | 2 | b+ tree | 2 | Pname_List_SF | 1 | 1 |
| | | | | 3 | Building_Index_SF | 1 | 2 |
| | | | | 4 | Building_List_SF | 1 | 1 |
| | | | | 5 | Professor_Data_File | 1 | 1 |

Figure A3. The Volume Table (VT), File Implementation Table (FIT), and Jupiter File Table (JFT)

Line (4) handles the case of data file scanning. D_cursor is initialized by using the query and into_list specified for the abstract file. (Again, data records are identical to their abstract record counterparts).

At line (5), if A_cursor is to be positioned at the first qualified record, $ADV_{MRS\_INDEX}$ is called. The expansion of $ADV_{MRS\_INDEX}$ is:

```
ADV_MRS_INDEX( A_cursor ) →

        if A_query is being processed using indices then
        {
(1)     loop: get next pointer p from inverted_list_buffer;
(2)           if no more pointers exist, set STATUS := EOF and return;

(3)           D_cursor.pointer := p;   /* D_cursor now points to record p */
(4)           ACC( D_cursor );         /* follow pointer */
        }
        else  /* scan data file */
        {
(5)           ADV( D_cursor );
        };
```

Lines (1-4) get the next pointer from the inverted_list_buffer and access the referenced record. At line (5), an advance of an abstract cursor translates to an advance of the data file cursor.

MRS_INDEX mappings for other operations (e.g., INS, REM, UPD, etc.) can be written in an analogous manner. We omit their details.

Now consider the MRS_DIVIDE transformation. Retrieving an abstract index record involves the retrieval of its (dominant) index record and the following of a linear-list linkset to access all of its list records. The index and list records are concatenated to reconstruct the abstract index record. Let X_mtid be the mtid of an abstract index file, and I_mtid and L_mtid be the mtids of its index and list files. The retrieval algorithm for MRS_DIVIDE is expressed by:

```
RET_MRS_DIVIDE( X_cursor, X_query, X_into_list, position ) →

(1)   I_into_list := ( trace to entire index record, index_buffer );
(2)   RET( I_cursor, X_query, I_into_list, before_first );

(3)   L_into_list := ( trace to entire list record, list_buffer );
(4)   INIT( L_cursor, null, L_into_list );

(5)   if position = at_first then ADV_MRS_DIVIDE( X_cursor );


ADV_MRS_DIVIDE( X_cursor ) →

(1)   ADV( I_cursor );
(2)   if STATUS = EOF then return;
(3)   get child pointer p from index record (in index_buffer);
(4)   while p <> null do
(5)   {     L_cursor.pointer := p;          /* L_cursor now points to record p */
(6)         ACC( L_cursor );
(7)         add pointers of list record (in list_buffer) to the pointers
                  of the index record (in index_buffer);
(8)         get next child pointer p from list record (in list_buffer);
      };
```

```
(9)    return fields of the abstract index record (reconstructed in
               index_buffer) that are specified in X_into_list;
(10)   set STATUS := successful;
```

In $\mathrm{RET}_{\mathrm{MRS\_DIVIDE}}$, the index cursor I_cursor is prepared for record retrieval, where a RETrieved index record is to be placed in index_buffer (lines (1-2)). The list cursor L_cursor is prepared for pointer following in lines (3-4), where an ACCessed list record is to be placed in list_buffer. $\mathrm{ADV}_{\mathrm{MRS\_DIVIDE}}$ is performed in line (5) if the first abstract index record is to be retrieved.

In $\mathrm{ADV}_{\mathrm{MRS\_DIVIDE}}$, lines (1-3) retrieve an index record and make preparations to follow a chain of list records. Lines (4-8) follow the chain and concatenate the pointers of the list records onto the index record. The abstract index record is reconstructed in the index_buffer. Lines (9-10) return the fields requested by X_into_list and set a successful status code.

MRS_DIVIDE mappings for other operations can be written in a straightforward manner, and will not be presented here.

As mentioned earlier, INTERNAL level operations, such as $\mathrm{RET}_{\mathrm{INTERNAL}}$ and $\mathrm{ADV}_{\mathrm{INTERNAL}}$, are primitives and are not given expansions.