PROGRESS TOWARD AUTOMATING THE DEVELOPMENT OF DATABASE SYSTEM SOFTWARE *

D. S. Batory

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

TR-84-03 February 1984

*To appear in *Query Processing in Database Systems*, Springer-Verlag, 1984, W. Kim, D. Reiner, D. S. Batory, editors.

Progress Toward Automating The Development of Database System Software *

D.S. Batory
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712

ABSTRACT

Recent advances in physical database modeling suggest that it may be possible to develop significant portions of a DBMS's software, specifically the physical database component, automatically from a small set of specifications. These advances are based on the Unifying Model and the Transformation Model, both of which are reviewed in this paper. The Unifying Model deals with file structures and record linking mechanisms; the Transformation Model deals with conceptual-to-internal mappings. We illustrate their concepts with models of the storage architectures of two very different commercial DBMSs: INGRES and RAPID. We then explain how the Unifying Model/Transformation Model framework can provide a basis for automating the development of physical database software.

1. Introduction

In recent years it is 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 ([IEEE84]), CAD and engineering databases ([IEEE82]), textual databases ([Ston82]), and databases for artificial intelligence ([IEEE83]) 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 has been customized in two ways: existing systems have been enhanced ([Hask82], [Ston83]) and new DBMSs have been developed ([John83]). In either case, it is well-known that developing special-purpose database software is an exceedingly difficult, costly, and not always successful undertaking. There is a definite need for tools that simplify and aid the development of database software. Presently such tools are lacking.

Within the next ten years, we may witness a revolution in the way database system software is developed. Recent work in physical database modeling points the way to these advances.

Physical database software handles the placement and retrieval of data on secondary storage. The topic of physical databases has traditionally dealt with file structures and record

* This research was supported by the National Science Foundation under Grant MCS-8317353.

linking mechanisms, but more generally it encompasses buffer management, query processing, concurrency control, and recovery algorithms.

Models of physical databases have progressed to the point where they are accurate enough to be used as blueprints for physical database software development. The models to which we refer are the Unifying Model (UM) and the Transformation Model (TM), both of which are reviewed in this paper ([Bato82b], [Bato84b]). It is believed that a technology based on these models can be realized in which the physical database software for a DBMS can be developed automatically from a small set of specifications. The impact of such a technology is potentially significant. Special-purpose DBMSs that are optimized to handle specific classes of applications can be built quickly and inexpensively; modifications to existing DBMSs (constructed using this technology) could be accomplished just as easily. As an added benefit, it would facilitate the development of prototype DBMSs in which new algorithms (e.g., concurrency control, file structure, query processing, buffer optimization) could actually be tested and evaluated, thereby providing a vehicle to help tie physical database theory to practice.

Earlier models of physical databases did not exhibit such possibilities. A progression of increasingly more sophisticated and realistic general-purpose models were proposed prior to 1983 ([Hsia70], [Seve72], [Senk73], [Yao77], [Marc81], [Bato82b]). [1] Unfortunately, none could accurately account for the diversity and variety of structures and algorithms that are present in operational DBMSs in a simple and comprehendible way. Even so, these works were quite important for they laid the foundation for the modeling techniques that we present in this paper.

In the following sections, we will review concepts that are fundamental to physical database modeling. We will show how these concepts have been used to model the physical databases, or *storage architectures,* of two operational DBMSs: INGRES and RAPID. We will then explain how these concepts can underly a technology which may help automate the production of physical database software.

2. The Unifying Model

Since 1970, many studies have contributed to the articulation and understanding of major performance and design issues in files and databases. Such studies have addressed hash-based files ([Seve76a], [Bato82a]), B+ trees ([Naka78], [Bato81]), transposed files ([Hoff76], [Marc83]), batched searching ([Shne76]), performance evolution of files as records are inserted and deleted ([Bato82a], [Heym82]), index selection ([Schk75], [Ande77]), differential files ([Seve76b]), generalized access path structures ([Haer78]), and file reorganization ([Bato82a]), among others. Although many studies seemed quite unrelated, it was shown that all were instances of a single framework, called the Unifying Model (UM) [Bato82b].

The UM had two distinct submodels. One enabled the file structures and physical databases (i.e., networks of interconnected files) that had been investigated in earlier works to be described parametrically. Among the parameters that were used were file structure type (e.g., dynamic hash-based, indexed-sequential, etc.), file size, blocking factors, average length of overflow chains, and file structure height. The other submodel was a set of cost equations which utilized these parameters to predict database performance. These equations were shown to generalize earlier analyses. In this paper, we will concern ourselves only with the descriptive

---

[1] It is worth noting that at one time DIAM was believed to be sufficiently general and accurate to be used as the basis for DBMS software production ([Senk73]). Since then, it has become evident that DIAM's modeling constructs have two fundamental limitations. First, they are inadequate to represent the conceptual-to-internal mappings of existing DBMS software. Second, they are simply too general to be useful in describing the complexities of DBMSs in a comprehendible way. As a result, few research contributions on physical databases have relied on the DIAM framework.

submodel; its basics and subsequent generalizations are explained below.

Physical databases can be *decomposed* into a collection of internal files and internal links. An *internal file* is a file of records that are instances of a single *internal record type*. Records of internal files are actually stored. A relationship between one or more internal files is an *internal link*. (We draw a distinction here between conceptual files and links, which are defined in database schemas, from internal files and links. We will see later that there is a significant difference between files and links that are conceptual and those that are internal).

The basic structures of a physical database are simple files and linksets. A *simple file* is a structure that organizes records of one or more internal files. Classical simple file structures include hash-based, indexed-sequential, B+ trees, dynamic hash-based, and unordered files. A *linkset* is a structure that implements one or more internal links. Classical linkset structures include pointer arrays, inverted lists, ring lists, and IMS's hierarchical sequential lists ([Date82]). [2] Catalogs of known simple files and linksets are given in [Bato84b].

Because internal files, internal links, simple files, and linksets are easy to comprehend, the structure of a physical database can be specified by a straightforward procedure. First, the physical database is decomposed into its constituent internal files and internal links. Second, the linkset implementation of each internal link is specified, and third, the simple file implementation of each internal file is specified. Here are two examples.

Consider a file of records of type DATA. A DATA record has n fields $F_1 \cdots F_n$. This file is stored as an inverted file with attributes F_j and F_k indexed. Decomposition of the inverted file reveals three internal files and two links. There is the DATA file and two index files $INDEX_j$ and $INDEX_k$, one for each of the indexed attributes. Each INDEX file is connected to the DATA file by precisely one link. Figure 1.dsd shows these relationships graphically in a *data structure diagram (dsd)*, where boxes represent files and arrows are links (drawn from the parent file to the child file).

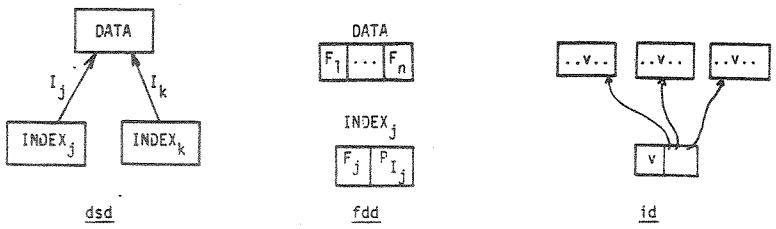


Figure 1. Decomposition of an Inverted File

A typical implementation of an inverted file has each INDEX file organized by a separate B+ tree and the DATA file organized by an unordered or heap file structure. In our example, there would be three simple files in all (i.e., a data file structure and two index file structures). The internal links would, of course, be implemented by inverted lists or pointer arrays.

To describe the implementation in more detail, two additional diagrams are used. One is a *field definition diagram (fdd)*, which shows the fields of the internal record types. Figure 1.fdd shows the DATA record type to consist of data fields $F_1 \cdots F_n$. It also shows the $INDEX_j$ record type to have two fields: a data field F_j and an inverted list field P_{I_j}. P_{I_j} is called the *parent field* of linkset I_j. The $INDEX_k$ type has a similar format to $INDEX_j$.

The other diagram is an *instance diagram (id)*, which is used to illustrate both the data structure and field definition diagrams. Figure 1.id shows an $INDEX_j$ record with data value v and its inverted list which references all DATA records that have v as its F_j value.

As another example, suppose the DATA records were organized as a multilist file, where

again fields F_j and F_k are indexed. Decomposition results in the same data structure diagram as in the inverted file example (Fig. 2.dsd). However, their distinction lies in the link implementations: multilist files use multilist linksets. The distinction can also be seen in the field definition and instance diagrams. Figure 2.fdd shows DATA records to have two additional fields C_{I_j} and C_{I_k}. These fields are respectively the *child fields* of links I_j and I_k. Their purpose is to contain linkset pointers to the next DATA record on a list of DATA records. Figure 2.id shows the same link instance of Figure 1.id, except that a multilist structure connects an $INDEX_j$ record to its DATA records.

As a general rule, the presence and function of parent and child fields in record types that are linked is determined solely by the linkset that materializes the link. In the case of inverted list linksets (Fig. 1), a parent field appears in every parent record. For multilist linksets (Fig. 2), both fields are present. IMS logical parent pointers are linksets that are implemented solely by parent pointers ([Date82]); only child fields are used. Sequential linksets do not require either parent or child fields (i.e., parent and child records are linked by contiguity). Thus, a linkset can introduce parent fields, child fields, both, or neither.

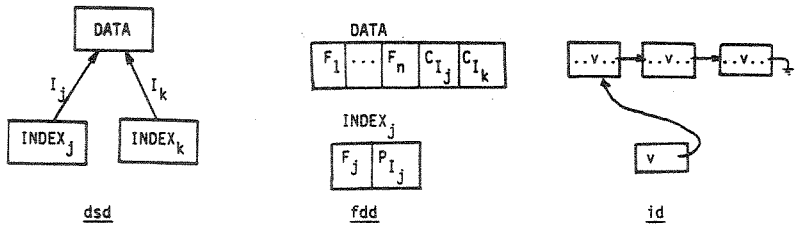


Figure 2. Decomposition of a Multilist File

The decomposition approach to modeling physical databases is intuitive and seemed quite powerful. Virtually all works published prior to the UM used similar, though not as extensive, techniques for modeling database structures. However, common to all models (including the UM) was the assumption that conceptual to internal mappings were simple. That is, given a data structure diagram of the conceptual record types and links, the mapping to the corresponding internal record types and links was obvious. In many commercial and specialized DBMSs, this is definitely not the case. New modeling concepts are required in order to capture the conceptual-to-internal mappings of actual DBMSs. The Transformation Model (TM) was developed in response to this need.

3. The Transformation Model

A primary function of a DBMS is to map conceptual files and operations to their internal counterparts. INGRES [Ston76], for example, maps relations and relational operations onto inverted files. RAPID [Turn79] and SYSTEM R [Date82] also begin with relations, but RAPID maps to transposed files and SYSTEM R maps to inverted files with record clustering.

An intuitive understanding of conceptual-to-internal mappings comes from recognizing that mappings can be modeled as 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 'pure' conceptual and 'pure' internal levels. It follows that physical databases can be modeled at different levels of abstraction.

What does it mean for a DBMS to have different levels of abstraction? It means that the DBMS's physical database software (the software that accomplishes the conceptual-to-internal

mappings) was written (or could have been written) by nesting abstract data types. That is, the outer-most types define the conceptual record types, their links, and operations. These types are defined in terms of 'primitive' abstract data types, and these primitive types in turn are based on even more primitive types, and so on, terminating with abstract data types that define the internal record types, their links, and operations. Thus, levels of abstraction have a practical interpretation. [2]

Recognizing different levels in a DBMS and mapping from one level to an adjacent level turns out to be straightforward. In the DBMSs that the author is aware, only ten different primitive mappings, here called *elementary transformations,* have been utilized. Elementary transformations are used singly or in combination to map records and links from one level of abstraction to the next lower level. In principle, this means that the conceptual-to-internal mappings of a software-based DBMS can be modeled by 1) taking the generic conceptual record types and links that the DBMS supports, and 2) applying a well-defined sequence of elementary transformations to produce the internal record types and links of the DBMS. For example, in the case of RAPID, INGRES, and SYSTEM R, all three begin with the same conceptual record types (i.e., relations), but are distinguished by different sequences of transformations (and hence different sets of internal record types and links). We will briefly describe each of the known elementary transformations later.

The use of elementary transformations to define conceptual-to-internal mappings is the basis of the Transformation Model (TM) [Bato84a-b]. This model is related to the UM in the following way. The UM relies on decomposition to identify the internal files and links of the physical database. Implementations for each internal file and link can then be specified. In contrast, the TM starts with the conceptual files and links that are supported by a DBMS and shows how their underlying internal files and links are derived. Thus, the TM supplants the intuitive process of physical database decomposition with a conceptual-to-internal mapping process that, it is believed, closely models the way DBMS software is written.

Ten elementary transformations were discovered as a natural consequence of modeling the conceptual-to-internal data mappings used in the ADABAS ([Gese76]), CREATABASE ([NDX81]), DMS-1100 ([Sper75]), IDMS ([Cull81]), IMS ([IBM80]), INGRES ([Ston76]), INQUIRE ([Info79]), RAPID ([Turn79]), SPIRES ([Stan73]), SYSTEM 2000 ([Casa81]), and TOTAL DBMSs ([Cinc79]), among others. Each transformation maps an abstract structure (e.g., record type, file, link) to one or more concrete structures. (It is possible for a record type/file to be 'concrete' at one level of abstraction and 'abstract' at a lower level). A brief description of each transformation is given below. More complete descriptions are given in [Bato84b]. Abstract records can be:

- *augmented* by metadata (e.g., delete flag, record type identifier, etc.);
- *encoded* for purposes of compression, encryption, or searching (e.g., SOUNDEX encoding);
- *collected* onto a single link occurrence (similar to the DBTG concept of a singular set);
- *segmented* along field boundaries to produce two or more (sub)records. A link occurrence connects the primary record to all of its secondary records;
- *divided* without respect to field boundaries to produce two or more (sub)records. A link occurrence connects the primary record to all of its secondary records;
- mapped directly to a concrete record by the ***null*** transformation. Null transformations arise when the application of a sequence of transformations is conditional.

---

[2] Conversely, levels of abstraction also provide new insight into how DBMSs can be implemented using abstract data types. Although this particular topic has been studied before (e.g., [Hamm76], [Rowe79], [Baro81]), the mechanism by which modular design concepts are applied at the internal level is still not well understood. Improvements and clarifications of such mechanisms - based on our notion of levels of abstraction - are presented in [Wise83].

Fields of abstract records can be:

- *extracted.* This produces an index or dictionary for the field's data values. Index or dictionary records each contain a distinct value and are connected via a link to all abstract records that have that value.

Links between abstract record types can be:

- *actualized.* That is, the materialization of an abstract link is expressed in terms of one or more concrete links.

DBMS storage architectures can be

- *layered.* A page at one level of abstraction is mapped to a distinct record at a lower level; the address of the page becomes the primary key of the record. Thus a page fetch is translated to a record retrieval, and a page write is translated to a record update.

Files of abstract records can be:

- ***horizontally partitioned*** into two or more subfiles.

To illustrate and explain the effects of elementary transformations, we again use data structure, field definition, and instance diagrams. Besides the usual conventions, there are two additions. First, abstract objects (typically record types) are indicated by dashed outlines in data structure diagrams. Figure 3 shows a data structure diagram of an abstract record type W and its materialization as the record types F and G and link L.

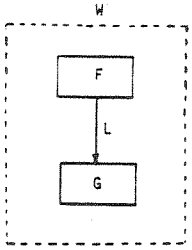


Figure 3. A Materialization of Abstract Record Type W

Second, pointers to abstract records arise naturally. These pointers, however, must ultimately reference internal (concrete) records. To indicate how pointer references are transformed, we rely on the orientation of record types within a dsd. The orientation of F and G in Figure 3, for example, shows that F *dominates* G. This will mean that a pointer to an abstract record of type W will actually reference its corresponding internal (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 known exception, to be considered later, is full transposition). The dominant concept is recursive; that is, if F records are abstract, a pointer to an F record will reference its dominant concrete record, and so on. In this way, pointers to abstract records are mapped to concrete records.

We will illustrate the TM approach in the following sections by presenting models of the storage architectures of two very different relational database systems: INGRES and RAPID. During this exercise, the reader should note the level of detail that is captured using the UM and TM framework. Later we will examine the role this framework can play in the automation of physical database software.

4. Applications

4.1 INGRES

INGRES was among the first major DBMSs that were based on the relational model. It was developed in the mid-1970's at the University of California, Berkeley, and is now marketed by Relational Technology, Inc.

The generic CONCEPTUAL record type supported by INGRES consists of n scalar and elementary fields (see Fig. 4). n is user-definable. Data values and their respective fields have fixed lengths. Relationships between two or more CONCEPTUAL record types are realized by join operations, rather than by physical structures. Thus, the underlying storage structures used by INGRES can be understood by examining how records of a single conceptual type are stored.

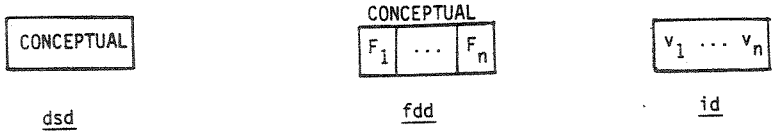


Figure 4. Generic CONCEPTUAL Record Type of INGRES

CONCEPTUAL records are materialized in the following way. INGRES allows elementary and compound fields to be indexed. (A *compound field* is defined in INGRES to be a field that consists of two to six elementary fields). Field F_i is indexed by *segmenting* it with duplication from CONCEPTUAL records. (That is, the field to be indexed is 'copied', leaving the CONCEPTUAL record intact). This produces an *ABSTRACT_INDEX*, record type connected to an ABSTRACT_DATA record type by linkset I_i (see Fig. 5). I_i is a singular pointer. (A *singular pointer* is a pointer array that contains precisely one pointer. The value of this pointer is called the *tuple id* of the CONCEPTUAL or ABSTRACT_DATA record). All fields are indexed in this manner. (The notation ( )... in Figure 5.dsd means that zero or more fields may be indexed).

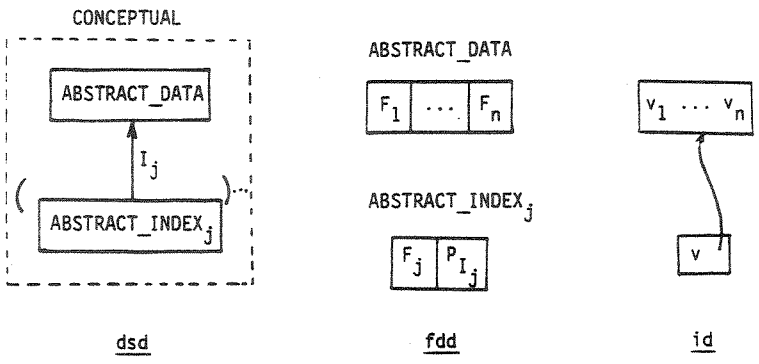


Figure 5. Segmentation of CONCEPTUAL Record Type

INGRES treats ABSTRACT_INDEX files as special normalized relations consisting of a data value field and a pointer field. This treatment actually simplifies the implementation of INGRES, for ABSTRACT_INDEX and ABSTRACT_DATA records are materialized by the same sequence of transformations. To avoid defining this transformation sequence twice, we will define it in terms of a generic record type, ABSTRACT_REC.

Let ABSTRACT_REC consist of m fixed length fields $G_1 \cdots G_m$ (see Fig. 6). ABSTRACT_REC is materialized by *segmenting* fields $G_1 \cdots G_m$ from ABSTRACT_REC. The LINE_REC and REC record types connected by link L are produced as a result (see Fig. 7). LINE_REC is fixed-length and contains only the field P_L. REC is identical to its ABSTRACT_REC counterpart. L is a singular pointer with special parent-child clustering properties (to be explained shortly).

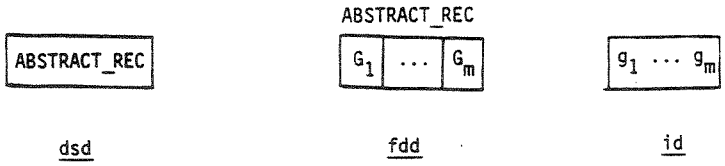


Figure 6. ABSTRACT_REC Type

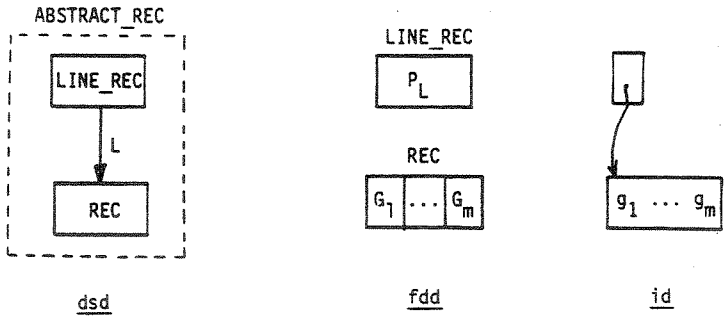


Figure 7. Segmentation of ABSTRACT_REC Type

The database administrator can declare whether or not instances of REC are to be compressed. (INGRES compresses records by eliminating trailing blanks from character fields). If REC is compressed, it is materialized by the *encoding* transformation. The COMPRESSED_REC type results. If compression is not requested, REC is materialized by the *null* transformation. The UNCOMPRESSED_REC type results (see Fig. 8).

The above composite sequence of transformations for materializing ABSTRACT_REC will be referred to as the *INGRES transformation*. This transformation is used to materialize both ABSTRACT_INDEX and ABSTRACT_DATA records. The conceptual-to-internal mappings of INGRES are summarized in Figure 9. As a general rule, INDEX records are not compressed.

The simple files that are used to organize the internal records of INGRES can be understood in terms of the INGRES transformation. When an ABSTRACT_REC is materialized, two internal record types result. One is LINE_REC. The other is either COMPRESSED_REC or UNCOMPRESSED_REC. In either case, INGRES *always* stores related records of both types on the same page (see Fig. 10). The storage locations of LINE_REC records are fixed

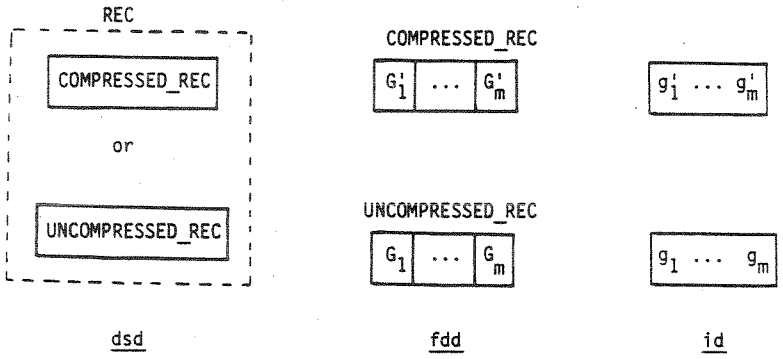


Figure 8. Compression of Null Transformation of REC Type

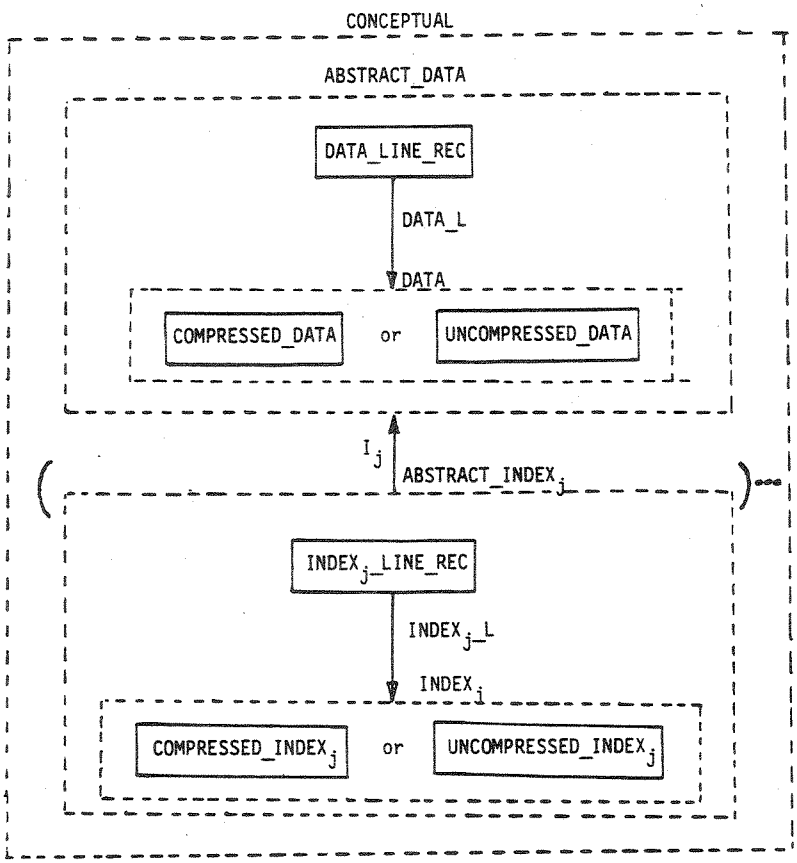
(but for an exception described below) and do not change with time; the storage location (within a page) of COMPRESSED_REC and UNCOMPRESSED_REC records may change with time. With this notion of parent and child clustering in mind, instances of related record pairs may be organized by an indexed-aggregate, hash-based, or unordered file structure. [3] As a general rule, COMPRESSED_INDEX and UNCOMPRESSED_INDEX records are usually organized by index-aggregate files.

It is worth noting that LINE_REC records maintain the 1:1 correspondence between tuple ids and the storage location of COMPRESSED_REC and UNCOMPRESSED_REC records. This correspondence allows COMPRESSED_REC and UNCOMPRESSED_REC records to be relocated (within the pages in which they are stored) without altering the INDEX records that reference them. (Recall the pointers of INDEX records are tuple ids). Relocations within blocks occur naturally as a consequence of updates, insertions, and deletions. Relocations beyond block boundaries occur when there is no room in a block to accommodate an expanded COMPRESSED_REC record. Such expansions happen when a CONCEPTUAL record is modified. In such cases, the CONCEPTUAL record is assigned a new tuple id. This means that the corresponding COMPRESSED_REC is moved to another block which can accommodate it and its LINE_REC record. Furthermore, all INDEX records that reference the CONCEPTUAL record must be updated to reflect the change in tuple id. This particular design reflects the belief that interblock movements do not occur often.

The storage architecture of INGRES is summarized in Figure 9. Source materials used in the derivation were [Held75], [Ston76], and [Butt83].

---

[3] An *indexed-aggregate* structure is a variation of indexed-sequential structures. When a data record is stored, it is never moved and thus has a fixed storage address. In indexed-sequential files, records are maintained in primary key sequence, and hence are often moved.



Internal and Aggregate Internal Record Types and Links	Implementation
ABSTRACT_DATA aggregate, ABSTRACT_INDEX$_j$ aggregate	hash-based, indexed-aggregate, or unordered
I_j, DATA_L, INDEX$_j$_L	singular pointer

Figure 9. The Storage Architecture of INGRES

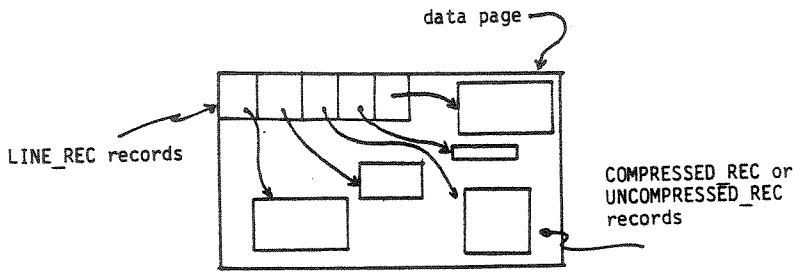


Figure 10. Clustering of LINE_REC, COMPRESSED_REC, and UNCOMPRESSED_REC Record

4.2 RAPID

RAPID was developed by Statistics Canada to support the processing of major surveys and censuses conducted by the Canadian Government. It is among the best known and most sophisticated statistical DBMSs presently in operation.

RAPID has a two-layer architecture. Both layers have well-defined conceptual record types and internal record types. We begin at the higher layer.

RAPID is based on the relational model. Its generic CONCEPTUAL record type is identical to that of INGRES (Fig. 11). CONCEPTUAL records are normally referenced by their contents. However, in RAPID they may also be directly referenced by their *record indexes*, RAPID's name for a tuple identifier.

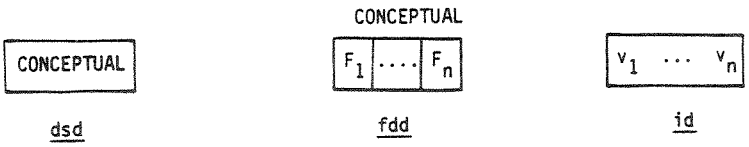


Figure 11. Generic CONCEPTUAL Record Type of RAPID

RAPID allows selected fields of CONCEPTUAL records to be compressed by an index encoding technique ([Alsb75], [Bato83]). When a CONCEPTUAL record type is defined in a RAPID schema, the values of a field's domain can be listed by enumeration. The index position at which a data value is located in the list is its index code. RAPID substitutes fixed-length index codes for data values thereby compressing CONCEPTUAL records.

Index encoding field F_i is modeled by *extracting* F_i from CONCEPTUAL. *DICTIONARY*$_i$ and IE_CONCEPTUAL (Index-Encoded CONCEPTUAL) record types, connected by link D_i, are formed as a result (see Fig. 12). D_i is an index encoded linkset, a linkset that is implemented solely by parent pointers. (The parent pointers in this case are index codes). All fields are index encoded in this manner. [4] Note that the notation ( )... in Figure 12.dsd means that zero or more fields may be extracted (i.e, index encoded).

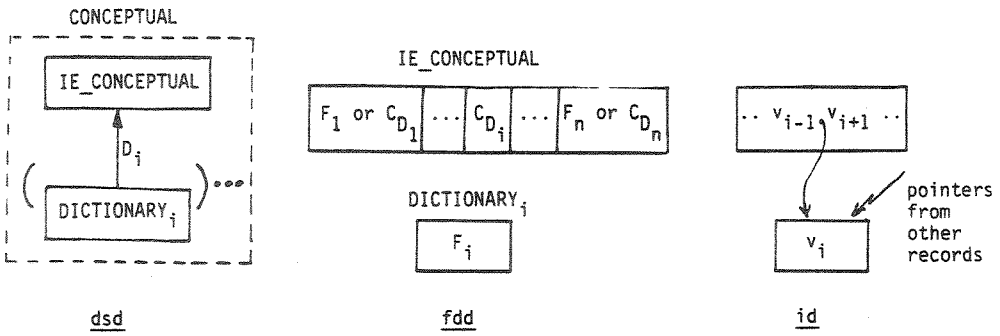


Figure 12. Extraction of CONCEPTUAL Fields

[4] Note that the extraction is performed partially by the database administrator when the domain is enumerated during schema definition. Although some DBMSs, such as CREATABASE ([NDX81]), have internal routines that perform the entire task of index encoding, RAPID's design reflects a typical statistical application need to define the set of data values associated with a given field *prior* to database loading ([McCa82]).

RAPID allows elementary or compound fields to be indexed (see Fig. 13). Field F_k is indexed by *segmenting* it with duplication from IE_CONCEPTUAL records. This produces the KEY_COLUMN_k record type which is connected to the ABSTRACT_DATA type by link I_k. I_k is a singular pointer. Other fields are indexed in a similar manner. Note that ABSTRACT_DATA is identical to its IE_CONCEPTUAL counterpart.

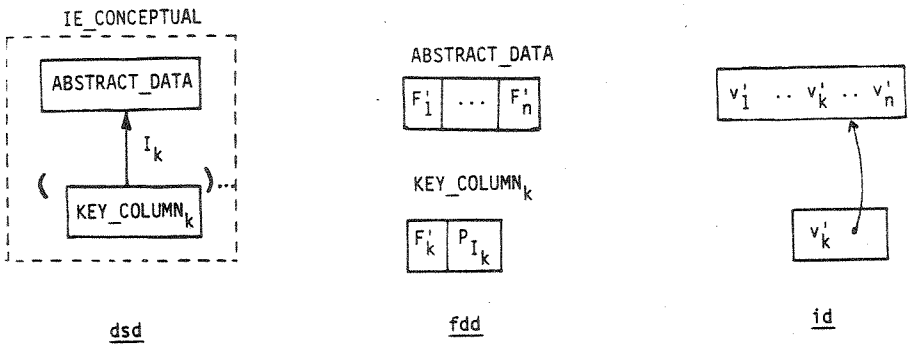


Figure 13. Segmentation of IE_CONCEPTUAL Type

The ABSTRACT_DATA record of Figure 13 is materialized in two steps (see Fig. 14). First, a metadata field containing a delete flag is *augmented* to each record. The flag is cleared initially and is set when the record is deleted. Second, each field is *segmented* from all other fields. (This is commonly known as *full transposition).* n+1 record types result; each contains precisely one field. Because all fields are treated identically, the resulting record types are not distinguished as being either 'primary' or 'secondary'. RAPID refers to the record type that contains data field *F*, as *COLUMN*, and the DELETE_FLAG field as $$$$ROOT.

Link R, which connects the COLUMN and $$$$ROOT record types, is a transposed link-set. A *transposed linkset* does not connect related records of different types together by actual pointers. Rather, a connection is implied by the sharing of a common record index. That is, related records share the same relative index position within their respective files. Thus, there is precisely one record instance for each of the $COLUMN_1 \cdots COLUMN_n$ and $$$$ROOT types for

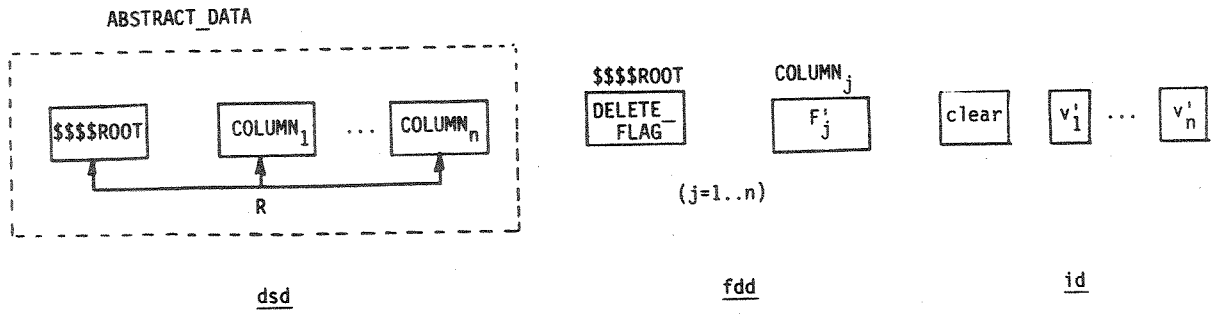


Figure 14. Augmentation and Transposition of ABSTRACT_DATA Type

every ABSTRACT_DATA record. The concatenation of the n+1 records that have record index i reconstructs the ABSTRACT_DATA record (with augmented delete flag) whose record index is i. Owing to this arrangement, a pointer to an ABSTRACT_DATA record may be treated as a pointer to any or all of its $COLUMN_1 \cdots COLUMN_n$ or $$$$ROOT records.

Finally, uncompressed data fields that contain text, such as comments, are now compressed by RAPID. This involves the elimination of trailing blanks. The *TEXT_COLUMN*$_j$ records that result from the *encoding* (compression) of *COLUMN*$_j$ records are variable in length. *COLUMN*$_j$ records whose fields do not contain text are mapped to *NONTEXT_COLUMN*$_j$ records by the *null* transform (see Fig. 15).

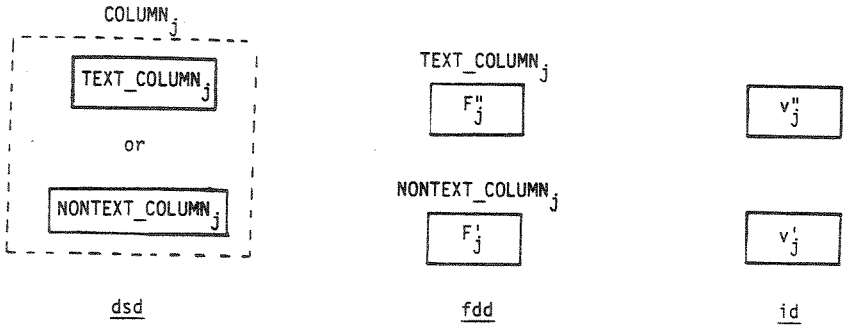


Figure 15. Encoding or Null Transformation of COLUMN Record Types

The internal record types of RAPID at this layer are the KEY_COLUMN, DICTIONARY, TEXT_COLUMN, NONTEXT_COLUMN, and $$$$ROOT. A distinct file structure is used to organized the records of each type. KEY_COLUMN records are organized by B+ trees; DICTIONARY, $$$$ROOT, TEXT_COLUMN, and NONTEXT_COLUMN records are organized by unordered files. [5] Records in an unordered file are addressed by their relative location keys (i.e., their index positions relative to the start of the file). The record index of a CONCEPTUAL record is the relative location key of its corresponding $$$$ROOT record. Similarly, the relative location key of a DICTIONARY record is its index code.

The storage architecture of the upper layer of RAPID is summarized in Figure 16.

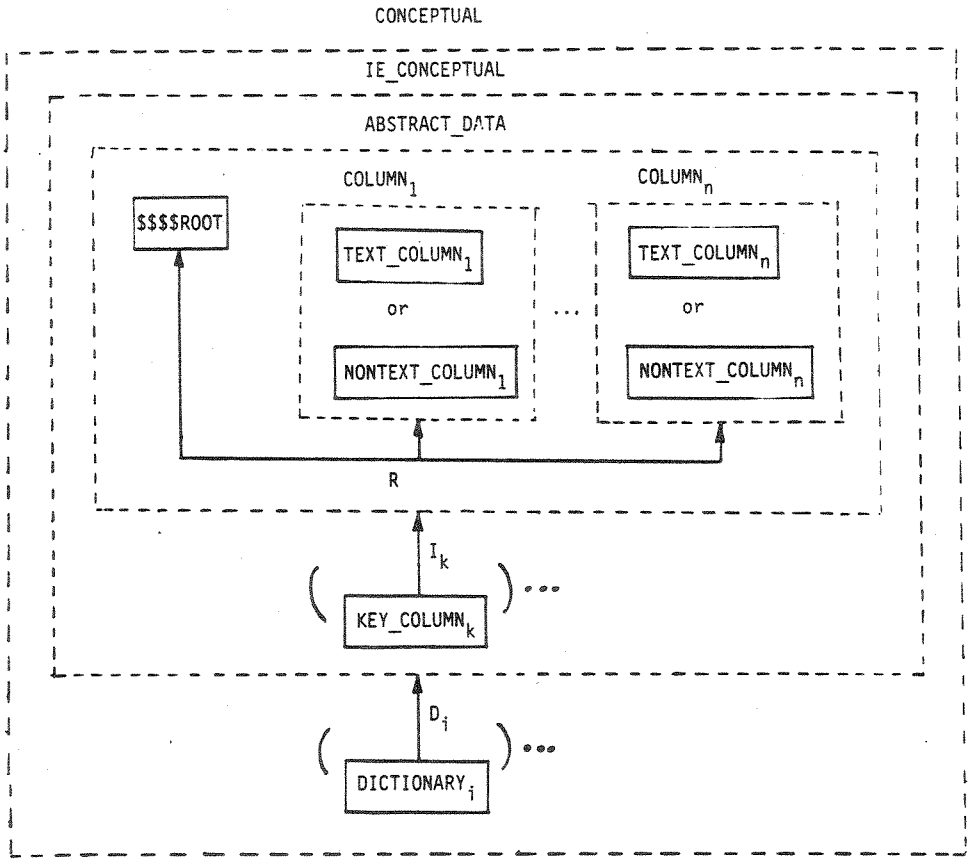
The file structures used by RAPID rely on the physical sequentiality of their blocks. This allows the addressing of a record in an unordered file, for example, to be accomplished by a simple indexing operation. Because RAPID files are expected to grow, allocating disk space would not be a difficult problem if there were but one file. However, there are at least n+1 growing files and preserving the sequentiality of their blocks is no longer a simple problem. The designers of RAPID adopted a two-layer architecture as a solution.

The address space of the upper layer was assumed to be virtual and of enormous size. By assigning the starting block addresses of sequentially growing files far enough apart, the interference among the storage requirements for different files could be eliminated. However, virtual secondary storage addresses ultimately have to be mapped to actual addresses. This was accomplished in the following way.

A 1:1 correspondence was defined between blocks on the upper layer and conceptual records of type ABSTRACT_BLOCK on the lower layer. (This is modeled by the *layering* transformation). A ABSTRACT_BLOCK record consists of two fields: BLOCK_ADDRESS and BLOCK_CONTENTS (Fig. 17).

---

[5] The DICTIONARY, $$$$ROOT, and NONTEXT_COLUMN structures resemble BDAM (1-level unordered) files. TEXT_COLUMN structures resemble RSDS (multilevel unordered) files.



Internal Record Types and Links	Implementation
KEY_COLUMN_k	B+ tree
$\$\$\$\$ROOT$, $TEXT\_COLUMN_j$, $NONTEXT\_COLUMN_j$, $DICTIONARY_i$	unordered
R	transposed linkset
I_K	singular pointer
D_i	relational linkset with parent pointers

Figure 16. Upper Layer Storage Architecture of RAPID

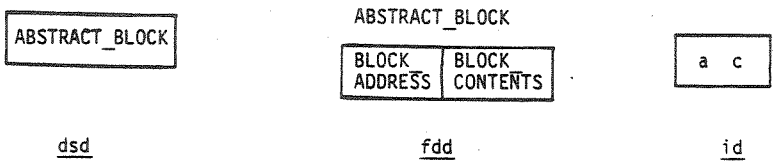


Figure 17. Lower Layer Generic Record Type

ABSTRACT_BLOCK is materialized by segmenting BLOCK_ADDRESS from BLOCK_CONTENTS (see Fig. 18). This produces the internal record types ADDRESS and BLOCK, which are connected by link AB. AB is a singular pointer.

ADDRESS records are organized by a B+ tree and BLOCK records are organized by an unordered file. Figure 18 summarizes the lower layer architecture of RAPID.

Source materials used in the derivation were [Turn79], [Stat81], and [Hamm82].

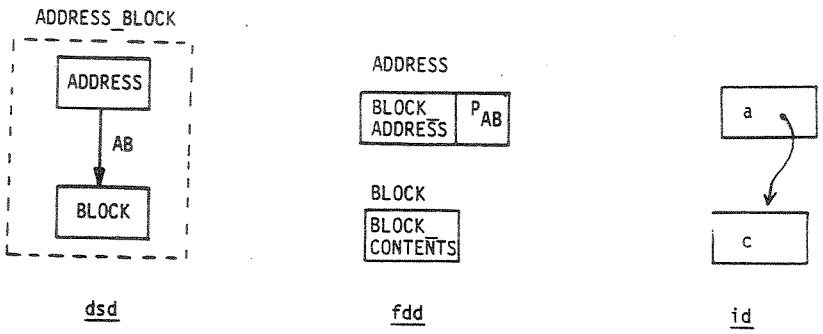


Figure 18. Lower Layer Storage Architecture of RAPID

4.3 Comments

The models of INGRES and RAPID are typical of those of other commercial DBMSs; storage architectures are described by sequences of elementary transformations. It is natural to ask how such sequences are generated, and what, if any, is the justification for using a given sequence. Clearly such questions are significant for they raise a fundamental point about what storage architectures are better than others. No answer can yet be given. The state of the research at this moment is to examine as many storage architectures as possible. Once a sufficient knowledge base has been collected, it is hoped that the underlying methodology for generating and justifying transformation sequences will become evident.

5. Data Operations, Locking, Query Processing, and DBMS Compilers

Elementary transformations, abstract data types, and automated software development are closely related. The idea of abstract data types is to encapsulate a record type definition and operations on the type with their mappings to lower level types and operations. We have explained elementary transformations as rules for mapping abstract structures to concrete structures. Alternatively, each transformation could have been explained in terms of mapping abstract operations (e.g., retrieval, updates, locks, etc.) to their concrete counterparts. It therefore seems possible that the data and operation mappings associated with each elementary transformation could be encapsulated as an abstract data type. It follows that the software which implements the storage architectures of commercial DBMSs may be described as a composition of abstract data types. It is believed that this connection between elementary transformations and abstract data types will lead to an automated development of internal DBMS software.

With this in mind, the models of the previous section can be used as blueprints for writing 'INGRES' and 'RAPID' software. In the process of writing such software, one will end up defining:

1) how conceptual operations on data are mapped to their internal counterparts,

2) how conceptual record and conceptual file locks are mapped to their internal counterparts, and

3) how query processing is accomplished.

At first glance, it is not clear how any of these processes can be automated or introduced into

our modeling framework. However there is a way, and we will consider each of the above points in turn.

Consider the division transformation and its rule for mapping data. This transformation takes an abstract record and divides it into one or more segments. One segment is the primary segment; the remaining are secondary segments. The primary segment is connected to its secondary segments by a link occurrence. Figure 19 shows the data mapping induced by the division transformation on record type REC. In Figure 19.id, the link is implemented by a list linkset, and the example REC record was divided into three segments; one primary and two secondary.

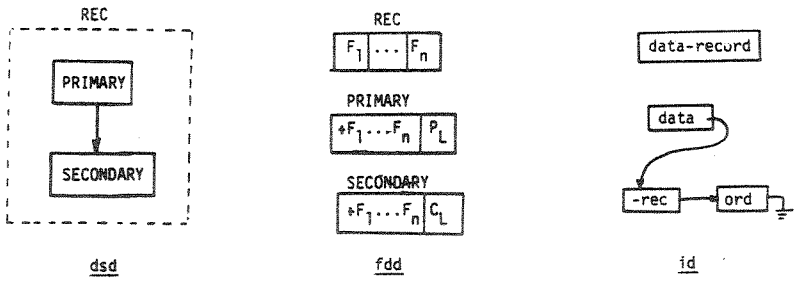


Figure 19. The Division Transformation

Consider the 'Access REC at rec-addr' operation which retrieves the record (of type REC) that is identified with the physical address rec-addr. At the level where REC is a concrete record type, this operation is a primitive. At a lower level (Fig. 19), the operation becomes:

1) Access PRIMARY at rec-addr;
2) Follow the link L occurrence to retrieve all related SECONDARY records;
3) Remove the parent and child fields of linkset L from the PRIMARY and SECONDARY records. Concatenate the records and return the result.

As this example shows, supplying a rule that defines how an abstract operation (Access REC) is mapped to a sequence of one or more concrete operations (e.g., Access PRIMARY) is fairly straightforward. It is often the case that several different rules can be supplied for each transformation. For example, in each of the following works [Kenn73], [Hoff76], [Niam78], [Bato79], a different algorithm for searching transposed files was presented. Each of these algorithms could be expressed as an operation mapping rule for the full transposition version of the segmentation transformation. The same holds for other elementary transformations.

In order for such rules to be valid at all levels of abstraction, the set of operations on abstract records (structures) must be identical to the set on concrete records (structures).

To see how this is possible, recall the basic ideas of the Unifying Model. At any level of abstraction, one is dealing with a physical database whose internal record types and links can be modeled by a data structure diagram. Just as physical databases can be decomposed into their constituent internal record types and links, so too can operations on physical databases be decomposed into a sequence of basic operations on internal record types and links. The total number of basic operations is quite small. Basic operations on record types include record retrieval, modification, insertion, access (as defined above), and deletion. Basic operations on links include the retrieval of the child records of a given parent, the retrieval of the parent record(s) of a given child record, and record linking and unlinking. With the aid of supplementary operations (e.g., record concatenation), level-independent transformation rules for mapping operations can be specified. The first phase of this research has already been completed

([Wise83]).

Now consider the 'Lock REC at rec-addr using lock-type' operation which locks the record (of type REC) that is identified with the physical address rec-addr. The locking mode is lock-type (e.g., exclusive or shared). Again, at the level where REC is concrete, this operation is a primitive. At a lower level (Fig. 19), the operation becomes:

Lock PRIMARY at rec-addr using lock-type.

There is no need to lock the corresponding SECONDARY records since the only way such records can be accessed is through their PRIMARY record. Rules for other transformations can be defined in a corresponding way.

Not all locks originate at the conceptual level. Locks on index records, for example, are not requested by DBMS users (since index records are invisible at the conceptual level). Instead, they are set by internal level routines. DBMSs also use special locks, called *cursor* or *currency* locks, which are imposed on records when a internal routine is examining a record as a candidate for retrieval ([Cull81], [Gray78]). (Retrievals often require a DBMS to examine many more records than are actually returned). A cursor lock is converted to a shared or exclusive lock if the record is returned to the user; otherwise it is released once the cursor (currency) is changed. Research is presently underway to define protocols for internal locking mechanisms and general rules for mapping locks according to elementary transformations.

How will such research differ or contribute to existing results? Many locking protocols have already been advanced in the literature. Some protocols are suitable only for restricted data structure diagrams, such as trees ([Silb80]); others are not restricted, such as two-phase locking ([Eswa76]). In our framework, these protocols are seen as concurrency control mechanisms that apply at the conceptual level. Protocols for lock mapping rules apply at levels below the conceptual. In this way, research on lock transformations should complement existing research.

How will research on query processing fit into the framework? The Unifying Model starts with the assumption that operations (e.g., retrievals, joins, etc.) on physical databases can be expressed as sequences of basic operations on internal record types and links. How these operations are sequenced in order to traverse multiple record types and links is often decided by a query optimizer which has knowledge of conceptual records and available links and indices. (The optimizer also relies on cost modules which predict the costs of traversing these links and accessing indices and 'conceptual' files. Such predictions *are* based on knowledge of the 'pure' internal level). Therefore, many query processing and query optimization algorithms work at a level of abstraction which lies immediately below the 'pure' conceptual level, but usually much above the 'pure' internal level. Further research on this topic is necessary.

It is evident that many difficult problems must be solved before physical database software development can be automated. But it should also be evident that software automation is possible. It is envisioned that a compiler of DBMSs would accept as input the transformation sequence that defines the conceptual-to-internal mappings of the target DBMS. Descriptive specifications about the transformations, such as limits on record lengths, compression algorithms to use, and internal record type and link implementations, would also be input. The output of the compiler would be the physical database component of the target DBMS, i.e., the software that stores, retrieves, etc. conceptual files and links according to the specified storage architecture. The generated software could include concurrency control and recovery mechanisms.

The compiler would rely on an extensive library of software modules. These modules would contain the code of different transformation rules, algorithms for different data compression techniques, support for each simple file and linkset structure (e.g., there would be module

for index-sequential files, B+ trees, pointer arrays, etc.), a general recovery manager (using shadow paging or logging), different lock managers (based on different protocols), and different query processors (for different processing strategies), etc. The compiler would use its input to select among the available modules and synthesize special software (based on the transformation sequence) to integrate them. If a special module is needed (e.g., support code for a special-purpose file structure), it would have to be added to the library before it could be used in DBMS development.

6. Conclusions

An advance in physical database modeling has been the recognition that conceptual-to-internal mappings can be explained as a sequence of primitive mappings called elementary transformations. This has enabled the storage architectures (i.e., physical databases) of many different operational database management systems to be modeled in a uniform, simple, and comprehendible way. Prior to the introduction of elementary transformations, it was difficult - and in many cases impossible - to account for the complexity of the storage architectures of commercial systems.

We have reviewed two models of physical databases in this paper: the Unifying Model (UM) and the Transformation Model (TM). The UM defines and characterizes file structures and record linking mechanisms; the TM explains conceptual-to-internal mappings. Taken together, they were used to describe the storage architectures of two commercial DBMSs: INGRES and RAPID.

It is believed that models of storage architectures are now accurate enough to be used as blueprints for DBMS (physical database) software development. In this paper we have indicated how this may be done. However, many problems still remain to be solved. Further advances in query processing, physical database modeling, and concurrency control are needed before this goal is satisfactorily attained.

7. References

[Alsb75] P.A. Alsberg, 'Space and Time Savings Though Large Database Compression and Dynamic Restructuring', *Proc. IEEE,* 63 #8 (August 1975), 1114-1122.

[Ande77] H.D. Anderson and P.B. Berra, 'Mimimum Cost Selection of Secondary Indexes for Formatted Files', *ACM Trans. Database Syst.,* 2 #1 (March 1977), 68-90.

[Baro81] A.J. Baroody and D.J. DeWitt, 'An Object-Oriented Approach to Database System Implementation', *ACM Trans. Database Syst.,* 6 #4 (December 1982), 509-539.

[Bato79] D.S. Batory, 'On Searching Transposed Files', *ACM Trans. Database Syst.,* 4 #4 (December 1979), 531-544.

[Bato81] D.S. Batory, 'B+ Trees and Indexed Sequential Files: A Performance Comparison', *ACM SIGMOD 1981,* 30-39.

[Bato82a] D.S. Batory, 'Optimal File Designs and Reorganization Points', *ACM Trans. Database Syst.,* 7 #1 (March 1982), 60-81.

[Bato82b] D.S. Batory and C.C. Gotlieb, 'A Unifying Model of Physical Databases', *ACM Trans. Database Syst.,* 6 #4 (December 1982), 509-539.

[Bato83] D.S. Batory, 'Index Encoding: A Compression Technique for Large Statistical Databases', *Proc. Workshop on Statistical Database Management 1983,* 306-314.

[Bato84a] D.S. Batory, 'Conceptual-to-Internal Mappings in Commercial Database Systems', *ACM PODS 1984,* 70-78.

[Bato84b] D.S. Batory, 'Modeling the Physical Structures of Commercial Database Systems', to appear in *ACM Trans. Database Syst.,* Also TR-83-21, Department of Computer Sciences, University of Texas at Austin, 1983.

[Butt83] P. Butterworth, (Relational Technology, Inc.), Technical Discussion, 1983.

[Casa81] I. Casas-Raposo, 'Analytic Modelling of Database Systems: The Design of a System

2000 Performance Predictor', M.Sc. Thesis, Department of Computer Science, University of Toronto, 1981.

[Cinc79] Cincom Systems, Inc., 'TOTAL PDP-11 Programmers Reference Manual', Cincinnati, Ohio, 1979.

[Cull81] Cullinane Database Systems, Inc., 'IDMS Database Design and Definition Guide', Westwood, Massachusetts, 1981.

[Date82] C.J. Date, *An Introduction to Database Systems, 3rd Ed.* Addison-Wesley, 1982.

[Eswa76] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger, 'The Notions of Consistency and Predicate Locks in a Database System', *Comm. ACM,* 19 #11 (November 1976), 624-633.

[Gese76] Gesellshaft fur Mathematik und Datenverarbeitung, 'ADABAS: Database Systems Investigation Report, Vol. 2 Part 1', Istitute fur Informationssysteme, Bonn, West Germany, 1976.

[Gray78] J.N. Gray, 'Notes on Database Operating Systems', Report RJ2188, IBM San Jose, 1978.

[Haed78] T. Haeder, 'Implementing a Generalized Access Path Structure for a Relational Database System', *ACM Trans. Database Syst.,* 3 #3 (September 1978), 285-298.

[Hamm76] M. Hammer, 'Data Abstractions for Databases', *Proc. Conf. Data: Abstractions, Definition, and Structure, SIGPLAN Notices 11 (1976), 58-59.*

[Hamm82] R. Hammond, (Statistics Canada), technical discussion, 1982.

[Hask82] R. Haskin and R. Lorie, 'On Extending the Functions of a Relational Database System', *ACM SIGMOD 1982,* 207-212.

[Held75] G. Held and M. Stonebraker, 'Storage Structures and Access Methods in the Relational Database Managment System INGRES', *Proc. ACM Pacific 1975,* 26-33.

[Heym82] D.P. Heyman, 'Mathematical Models of Database Degradation', *ACM Trans. Database Syst.,* 7 #4 (December 1982), 615-631.

[Hoff76] J.A. Hoffer, 'An Integer Programming Formulation of Computer Database Design Problems', *Information Sciences,* 11 (1976), 29-48.

[Hsia70] D. Hsiao and F. Harary, 'A Formal System for Information Retrieval from Files', *Comm. ACM,* 13 #2 (February 1970), 67-73.

[IBM80] IBM, 'IMS/VS Version 1: General Information Manual', 1980.

[IEEE82] *IEEE Database Engineering,* June 1982, R. Katz, editor.

[IEEE83] *IEEE Database Engineering,* December 1983, W. Kim, editor.

[IEEE84] *IEEE Database Engineering,* March 1984, D. Batory, editor.

[Info79] Infodata Systems, Inc., 'INQUIRE Basic Training Course', Pittsford, New York, 1979.

[John83] H.R. Johnson, J.E. Schweitzer, and E.R. Warkentine, 'A DBMS Facility for Handling Structured Engineering Entities', *1983 SIGMOD Engineering Design Applications,* 3-12.

[Kenn73] S.R. Kennedy, 'The Use of Access Frequencies in Database Organizations', Ph.D. Th., Cornell U., Ithaca, N.Y., 1973.

[Marc81] 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 (September 1981), 441-463.

[Marc83] S.T. March, 'Techniques for Structuring Database Records', *ACM Comp. Surveys, 15 #1 (March 1983), 45-80.*

[McCa82] J.L. McCarthy, 'Metadata Management for Large Statistical Databases', *Proc. VLDB 1982,* 234-243.

[Niam78] B. Niamir, 'Attribute Partitioning in a Self-Adaptive Relational Database System', Report MIT/LCS/TR-192, Laboratory for Computer Science, M.I.T., 1978.

[Naka78] T. Nakamura and T. Mizoguchi, 'An Analysis of Storage Utilization Factor in Block Split Data Structuring Scheme', *Proc. VLDB 1978,* 489-495.

[NDX81] NDX Retrieval Systems, Inc., 'CREATABASE Performance Manual', Houston, Texas, 1981.

[Rowe79] L.A. Rowe and K.A. Shoens, 'Data Abstractions, Views, and Updates in RIGEL', *ACM SIGMOD 1979,* 71-81.

[Schk75] M. Schkolnick, 'The Optimal Selection of Secondary Indices for Files', *Information Systems,* 1 (1975), 141-146.

[Senk73] M.E. Senko, E.B. Altman, M.M. Astrahan, and P.L. Feder, 'Data Structures and Accessing in Database Systems', *IBM Syst. Jour.,* 12 #1 (1973), 30-93.

[Seve72] D.G. Severance, 'Some Generalized Modeling Structures for use in Design of File Organizations', Ph.D. Thesis, University of Michigan, 1972.

[Seve76a] D.G. Severance and R. Duhne, 'A Practitioner's Guide to Addressing Algorithms', *Comm. ACM,* 19 #6 (June 1976), 314-326.

[Seve76b] D.G. Severance and G.M. Lohman, 'Differential Files: Their Application to the Maintenance of Large Databases', *ACM Trans. Database Syst.,* 1 #3 (September 1976), 256-267.

[Silb80] A. Silberschatz and Z. Kedem, 'Consistency in Hierarchical Database Systems', *Jour. ACM,* 27 #1 (1980), 72-80.

[Shne76] B. Shneiderman and V. Goodman, 'Batched Searching of Sequential and Tree Structured Files', *ACM Trans. Database Syst.,* 1 #3 (September 1976), 268-275.

[Sper75] Sperry Rand Corporation, 'DMS 1100 Schema Definition: Data Administrator Reference', 1975.

[Stan73] Stanford University, 'Design of SPIRES: Vol. I and II', Center for Information Processing, Stanford University, 1973.

[Stat81] Statistics Canada, 'RAPID Internals Manual', Ottawa, Ontario, 1981.

[Ston76] M. Stonebraker, E. Wong, P. Kreps, and G. Held, 'The Design ad Implementation of INGRES', *ACM Trans. Database Syst* 1 #3 (September 1976), 189-222.

[Ston82] M. Stonebraker, et al., 'Document Processing in a Relational Database System', Report UCB/ERL M82/32, Electronics Research Laboratory, University of California, Berkeley, 1983.

[Ston83] M. Stonebraker, B. Rubenstein, and A. Guttman, 'Application of Abstract Data Types and Abstract Indices to CAD Databases', Report UCB/ERL M83/3, Electronics Research Laboratory, University of California, Berkeley, 1983.

[Tsic77] D.C. Tsichritzis and F. Lochovsky, *Data Base Management Systems,* Academic Press, New York, 1977.

[Turn79] M.J. Turner, R. Hammond, and P. Cotton, 'A DBMS for Large Statistical Databases', *Proc. VLDB 1979,* 319-327.

[Wise83] T. Wise, 'A Technique to Model and Design Physical Database Structures', M.Sc. Thesis, Department of Computer and Information Sciences, University of Florida, 1983.

[Yao77] S.B. Yao, 'An Attribute Based Model for Database Access Cost Analysis', *ACM Trans. Database Syst.* 2 #1 (March 1977), 45-67.